Revisiting Data Partitioning for Scalable RDF Graph Processing

Jorge Armando Galicia Auyón

THÈSE

pour l'obtention du Grade de

Docteur de l'Ecole Nationale Supérieure de Mécanique et d'Aérotechnique

(Diplôme National - Arrêté du 25 mai 2016)

Ecole Doctorale : Sciences et Ingéniérie pour l'Information et Mathématiques
Secteur de Recherche : Informatique et Applications

Présentée par :

Jorge Armando GALICIA AUYÓN

****

Revisiting Data Partitioning for Scalable RDF Graph Processing

****

Directeur de thèse : **Ladjel BELLATRECHE**
Co-encadrant de thèse : **Amin MESMOUDI**

****

Soutenue le 12 Janvier 2021
devant la Commission d'Examen

****

JURY

Président :	
Emmanuel GROLLEAU	Professeur, ISAE - ENSMA, Poitiers
Rapporteurs :	
Yannis MANOLOPOULOS	Professor, Open University of Cyprus, Chypre
Farouk TOUMANI	Professeur, Université Blaise Pascal, Clermont-Ferrand
Membres du jury :	
Genoveva VARGAS-SOLAR	Chargée de Recherche, CNRS, Grenoble
Carlos ORDONEZ	Associate Professor, University of Houston, USA
Patrick VALDURIEZ	Directeur de Recherche, INRIA, Montpellier
Amin MESMOUDI	Maître de Conférences, Université de Poitiers, Poitiers
Ladjel BELLATRECHE	Professeur, ISAE - ENSMA, Poitiers

It always seems impossible until it's done.

— Nelson Mandela

Acknowledgements

This thesis would not have been possible without the inspiration and support of so many individuals:

- I would like to thank my supervisors Prof. **Ladjel Bellatreche** and Dr. **Amin Mesmoudi** for their guidance, patience, and support during my thesis. The insightful discussions I had with each of you definitely enlightened my way while exploring new ideas. Their careful editing contributed greatly to the production of this manuscript. You have helped me learn how to do research.

- I would also want to show my sincere gratitude to Professors **Yannis Manolopoulos** and **Farouk Toumani** for taking their precious time to be *rapporteurs* of my thesis. I greatly appreciate your feedback that I carefully considered to improve the quality of my work. In addition, I would like to thank Professors **Genoveva Vargas-Solar, Patrick Valduriez, Carlos Ordonez**, and **Emmanuel Grolleau** who have honored me to be part of the examination committee and whose questions have enriched my study from many different angles.

- I also thank all the members of the LIAS laboratory who welcomed me and with whom I shared so many moments of conviviality throughout these three years. Many thanks to **Allel Hadjali, Brice Chardin, Henri Bauer, Michael Richard**, and **Yassine Ouhammou** for mentoring me during the first steps of my teaching career. I wish to extend my special thanks to **Mickaël Baron** for the technical support throughout my work, for his availability and enthusiasm. I sincerely thank **Bénédicte Boinot** for her kindness, generosity, and all the administrative assistance to coordinate my thesis project.

- I want to thank my past and present Ph.D. colleagues and friends. **Abir, Amna, Anaïs, Anh Toan, Chourouk, Fatma, Houssameddine, Ibrahim, Ishaq, Issam, Khedidja, Louise, Matheus, Mohamed, Nesrine, Réda, Richard, Sana, Simon**, and

Soulimane. Many thanks to everyone for all the shared moments in which we exchanged ideas of our academic works and for the diverse conversations that cleared my mind when I needed it most.

- Many thanks to my group of friends, **Anita, Abril, Gad, Gledys, Laura, Lina, Marcos, Mariana, Melissa, Moditha**, and **Suela** on whom I was able to unload a lot of pressure thanks to so many shared laughs. Your sense of humor was the ray of sunshine for many cloudy days during this journey.

- Finally, my deep and sincere gratitude to my family for their unconditional and unparalleled love, help, and support. Many thanks to my parents, **Veronica** and **Jorge**, my sisters **Daniela** and **Jimena**, brother **José**, and aunt **Carmen** for always being there to restore my optimism when I was missing. I am forever indebted to **Régis, Marie France, Caroline, Thomas**, and **Marie José** who became my family and I never felt alone while being so far from my loved Guatemala. Finally, I would like to express my deepest gratitude to **Gabriel**, who encouraged, supported, and loved me without fail and stood by my side every time. I will be forever grateful to you, this journey would not have been possible if not for all your support.

¡Muchas gracias a tod@s!

Table of Contents

Introduction

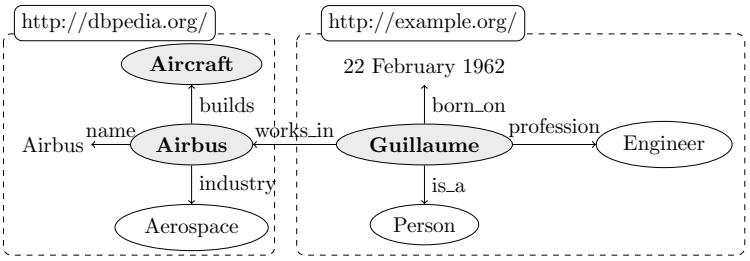
Context

Nowadays, we live in a hyper-connected world in which large amounts of data and knowledge are issued from several providers such as social media, mobile devices, e-commerce, sensors, the Internet of Things, and many others. Much of this data is produced and available on the Internet. To date, the Web holds more than 1.8 billion websites[1] and its size is steadily expanding. The data on the Web is characterized, among others, by its *volume* (e.g., 4PB of data created by Facebook daily), *variety* of sources with different formats (e.g., text, tables, blog posts, tweets, pictures, videos, etc.), *veracity*, where the following issues have to be dealt: data origin, authenticity, uncertainly, imprecision, completeness, data quality, ambiguity, etc. and *velocity* (the fast generation of data must correspond to the speed of its processing). The information and knowledge associated to this data, if well prepared, can be valuable to *consumers* (companies, governments, private users, etc.) who can obtain relevant *knowledge* and eventually useful *wisdom*.

However, the availability of the data on the Web does not imply its direct and straightforward exploitation. Tim Berners-Lee, the inventor of the World Wide Web, was aware of this since the mid-1990s. His vision was that computers could manipulate and, more importantly, interpret the information published on the Web. The concept flourished in the following years and the Semantic Web emerged from this vision. It seeks to organize the Web, composed of hyperlinked documents and resources, based on semantic annotators described and formalized in ontologies. The presence of ontologies offers reasoning capabilities over this data. These annotations let different actors such as scientific communities, companies, governments, end-users, etc. to link, exchange, and share data on the Web.

To satisfy these noble objectives, the data on the Web must be represented and queryable in an intuitive and clear way. With this motivation in mind, the World Wide Web Consortium (W3C) led many efforts to specify, develop and deploy guidelines, data models, and languages that lay the foundations for the semantic interconnection of Web data. Among the set of guidelines, the W3C published a recommendation [W3C04] presenting some principles, constraints, and good practices for the core design components on the Web. In this document, the W3C established how to identify Web resources using a single global identifier: the URI (Unique Resource Identifier). URIs, whose generalization in IRIs (International Resource Identifiers) was defined later in [W3C08], are the base component of the Semantic Web architecture [iIaC05]. When it comes to data models for the Semantic Web, the W3C proposed and largely promoted the Resource Description Framework (RDF) [RC14] to express information about resources. This data model is the cornerstone of other W3C's models and languages such as **(i)** RDF Schema (RDFS) [DB14] providing classes and vocabularies to describe groups of related resources and their relationships, **(ii)** SKOS (Simple Knowledge Organization System) [AM09] a common data

[1] `https://www.internetlivestats.com/`



(a) RDF Graph

```
@prefix db:<http://dbpedia.org/> .
@prefix ex:<http://example.org/> .

SELECT ?x ?y WHERE {
?x ex:works_in db:Airbus .
?x ex:profession ?y .
}
```

(b) SPARQL query

Figure 1: Examples for RDF and SPARQL

model to represent knowledge systems like taxonomies and classification schemas, and **(iii)** the Web Ontology Language (OWL) [Gro12] giving means to define and represent ontologies.

All the previous models and languages are built using RDF triples. A triple is the *smallest unit of data* in RDF. A triple models a single statement about resources with the following structure $\langle subject, predicate, object \rangle$. It indicates that a relationship identified by the predicate (also known as property) holds between the subject and the object depicting Web resources (things, documents, concepts, numbers, strings, etc.).

Example 1. The statement *"Airbus builds aircrafts"* can be represented by a triple as $\langle Airbus, builds, Aircraft \rangle$[2]. This triple can be represented logically as a graph where two nodes (subject and object) are joined with a directed arc (predicate) as shown in Figure 1a.

Besides, the interconnection characteristic gives RDF the ability to link triples from different datasets via their IRIs. The result of merging triples constitutes a *Knowledge Graph* (KG). For instance, our triple $\langle Airbus\ , builds, Aircraft \rangle$ can be easily linked to $\langle Guillaume, works\_in, Airbus \rangle$[3] through the predicate $works\_in$ (Figure 1a).

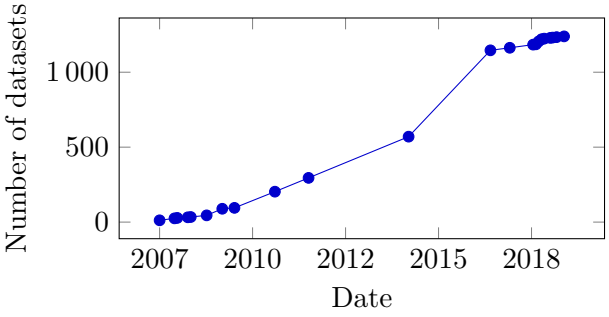
To query KGs and RDF datasets, the W3C defined SPARQL [SH13a] as the standard query language for RDF. SPARQL allows expressing queries across diverse datasets. The simpler SPARQL queries are those formed as a conjunction of triple patterns (known as *Basic Graph Patterns BGP*). Triple patterns are similar to RDF triples except that the subject, predicate, or object may be a variable.

Example 2. The query in Figure 1b asks for all the engineers working at the Airbus company. The solution of this query is a subgraph of the queried graph(s) in which the variable terms are mapped to the values of the resulting subgraph. Processing a SPARQL query can be viewed as a subgraph matching problem. The results for the previous query are the mappings of $?x \rightarrow$ `<http://example.org/Guillaume>` and $?y \rightarrow$ `<http:// example.org/Engineer>`.

The popularity of RDF is due to its *flexibility*, *simplicity*, and *availability* of a query language. Therefore, several organizations and governments spent a lot of effort to publish their data on the Web in RDF. This phenomenon is known as the Linked Open Data (LOD). The LOD cloud is one of the most well-known initiatives that allows available data sets on the Web to be referenced. The number of datasets in the LOD cloud has increased rapidly. Currently, it counts more than 1260 datasets, where a large subset of them contains several billions of triples[4] as shown in Figure 2. These datasets are accessed via SPARQL endpoints (RESTful web services to expose RDF data queried with SPARQL) or downloading the data as data dumps.

Furthermore, the spread of RDF and SPARQL is evidenced by their usage to process and query recent valuable data to increase the decision power and crisis management. For example, with the recent Covid-19 pandemic, different organisms, local health authorities, press structures, research laboratories, etc. generated large amounts of data on a daily basis. The U.S.

[2]This triple is a part of DBpedia's Knowledge Graph.
[3]A part of a KG: http://example.org/Guillaume
[4]`https://lod-cloud.net/`



Dataset	Number of triples (Millions)
DBLP	207
Yago	284
Freebase	2 000
DBPedia'14	3 000
Wikidata	12 000

Figure 2: Linked Open Data (LOD) evolution

Government's open data portal published daily stats in RDF format to facilitate their integration[5]. Another example is the work developed by the INRIA research team Wimmics[6]. They developed a Knowledge Graph named Covid-on-the-Web [WRT20] merging academic articles from the COVID-19 Open Research Datasets. The graph counts more than 674 million triples describing parts of scientific publications (title, abstract, and body), and information that is enriched by annotations obtained from other sources like *DBpedia*.

Today, Knowledge Graphs (*KG*s) have become popular means for academia and industry for *capturing*, *representing*, and *exploring* structured knowledge. The pioneer *KG*, *DBpedia* [LIJ+15], was released in the beginning of 2007 by a group of academia. It aims at collecting semi-structured information available on Wikipedia. It was followed by Freebase [BEP+08] which served as the basis for Google's *KG*, Yago [SKW07], Wikidata [Vra12], and many other academic graphs. The launch of the first commercial *KG* by Google in 2012 coined the term *Knowledge Graph* and established its crucial role in web data management. Google's team uses it to enhance its search engine's results with *infoboxes* (presenting a subset of facts about people, places, things, companies, and many other entities that are relevant to a particular query). The information in Google's *KG* is harvested from a variety of sources and today it relates facts from billions of entities that are exploited not only by their search engine but also by smart devices like Google Assistant and Google Home. Since then, several Web contenders have presented their *KG*. Among them we can mention, the Yahoo *KG* [Tor14] in 2013, the Bing *KG* [Tea15], the LinkedIn *KG* [He16], and many more. This large panoply of *KG*s can be divided into three main classes [BSG18]:

(i) *Generalized KGs* whose sources cover a variety of topics, here we find *KG*s such as Freebase, Wikidata, NELL, DBPedia, YAGO, MetaWeb, Prospera, Knowledge Vault, GeoNames, ConceptNet, etc.;

(ii) *Specialized KGs* gathering information from similar subject matters, such as the Facebook *KG* (social graph with people, places, things combined with information from Wikipedia), Amazon Knowledge Graph (which started as product categorization ontology), Wolfram Alpha (linking world facts and mathematics), LinkedIn *KG*, Recruit Institute of Technology (connecting people, skills, and recruiting agencies);

(iii) *Enterprise KGs* like those developed specifically for some companies such as Banks and Credit Rating Agencies.

This variety and the wealth of *KG*s motivate researchers from other Computer Science fields to integrate them in different phases of their life cycles. Recently, *KG*s have been used in *recommendation systems* for explanation purposes of recommended items [GZQ+20]. In *Business Intelligence systems*, *KG*s have also been used to cure the data and augment data warehouses to

[5] `https://catalog.data.gov/dataset/covid-19-daily-cases-and-deaths`
[6] `https://team.inria.fr/wimmics/`

reach high-value [BBKO20]. In Question-Answering Systems, *KG*s are used in several phases of their life cycle [SRB+18]. In [BBB19], they are used for handling ambiguity problems of Natural Language queries.

The Ecosystem of Knowledge Graphs

The above presentation shows the complexity of the environment of the *KG*. From research and learning perspectives, it has become urgent to deeply understand this environment in order to identify its different elements. Making explicit these elements contributes in understanding some research issues. The main elements of a *KG* are: *actors*, *requirements*, *constraints*, *services*, *storage* and *deployment infrastructures*. In Figure 3, we propose a detailed ecosystem of a *KG* using the Entity-Relationship Diagram.

The following comments this diagram: A *KG* is the entry point of our diagram. It has a name (KG_Name) and a nature (certain/uncertain like NELL [CBK+10] and Knowledge Vault [DGH+14]). A *KG* concerns one or several domains/topics (Medicine, Environment, Education, etc.), and it is built from one or several data sources. It is defined and designed by one or multiple creators. A creator owns the *intellectual property* of the *KG*. It can be any organization or user. A creator has a name (Creator_Name) and a type (academic/ industrial). A *KG* is administered by a manager that can be different from its creator. The manager is in charge of implementing internal (e.g., storage, loading, maintenance, archive, security) and external (e.g., access, availability, extraction) services. Each service has a name (Service_Name) and a means to manipulate and reach this data (using either a SPARQL endpoint or data dump in the case of external services). The *KG* is stored in a *Storage Infrastructure* (Triple Store). It has a name (Store_Name) and information about its internal storage (native, non-native). This store is deployed in a given platform that may be either centralized, distributed, or parallel. Finally, a *KG* is designed to be exploited by consumers using the external services that extract fragments/views of the referred *KG*. The extracted data can be materialized at the consumer

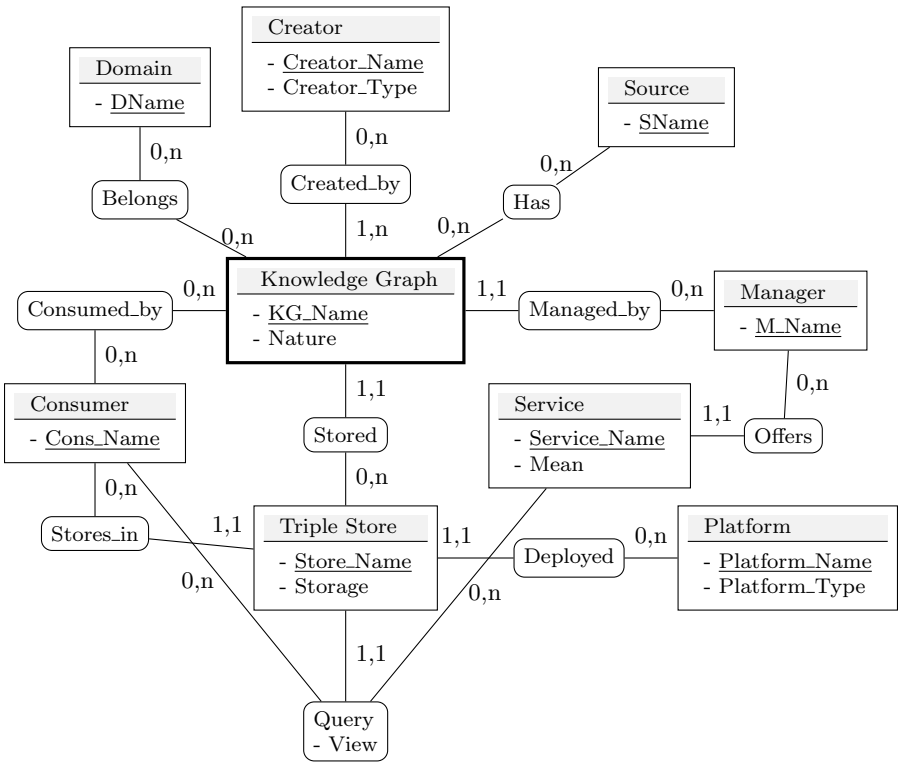


Figure 3: Knowledge Graphs Ecosystem

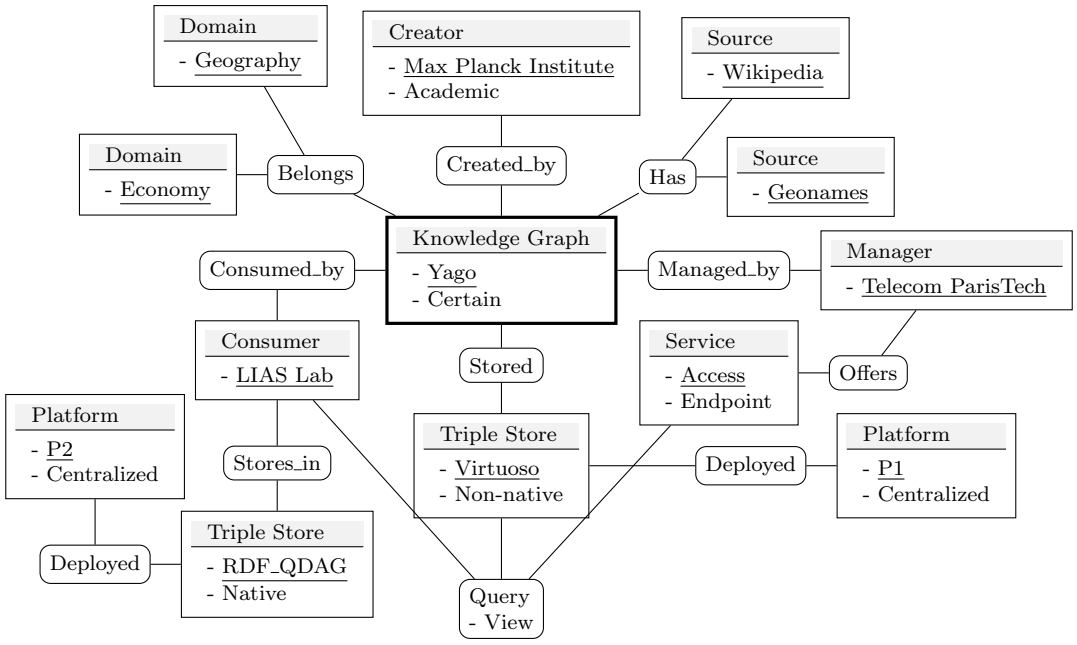


Figure 4: An Example of an instantiation of our Model: The Case of the Yago KG

level in a different triple store. A consumer can be any entity covering normal users, bots, learning institutions, researchers, industrials, etc.

Figure 4 gives an instantiation of our model for the Yago's KG, where we assume that the LIAS Laboratory is one of its consumers.

Triple Stores with some numbers The storage infrastructure that includes triple stores is an important element of the KG ecosystem, since it has to provide the following services:

- A scalable storage of the Web data deluge (e.g., *Academic KG*: Yago2 → 284 million triples, DBLP → 207 million triples, DBPediaEN → 538 million triples, *Commercial KG*: eBay product KG → 1 billion triples, Google's KG → 500 billion triples),
- A quick response time of SPARQL queries to meet consumer requirements,
- A high availability of this data and services (e.g., DBpedia SPARQL endpoint receives more than 177k queries per day) [VHM+14].

The Race for Efficient Triple Stores

As suggested in the above sections, it is important to notice that the Triple Store is at the heart of the KG' ecosystem. This situation motivates and pushes researchers from the *Core Databases*, *Semantic Web*, and the *Infrastructure for Information Systems* to develop triple stores that can be classified into two main groups: *non-native* and *native* as illustrated in Figure 5.

Non-native Systems The managers of these systems choose existing storage solutions to build their triple stores. The Relational Database Management Systems (RDBMSs) were widely used to design such systems. Virtuoso [EM09] is one of the most popular triple stores adopting this strategy. Its storage model consists of a single table including naturally the three columns for the attributes of the RDF triple (subject S, predicate P, and object O). At query run-time,

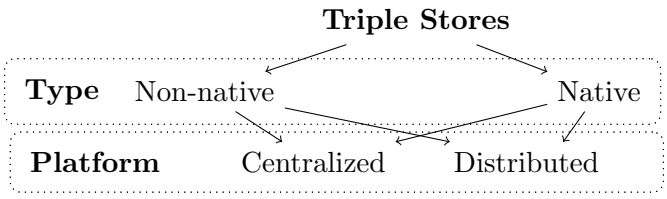


Figure 5: Triple stores classification

any SPARQL query is translated to SQL. This strategy generates multiple self-joins (one for each triple pattern) of the single table that significantly degrades the system's performance. To reduce the number of self-joins induced by this type of approach, Jena [Wil06] proposed a new storage of the traditional triple as a flat table (called *property table*) including the subject and all predicates as attributes. This storage strategy generates the problem of NULL values and stresses to represent multi-valued properties.

To overcome the above drawbacks, other systems used one of the oldest optimization techniques known under the name of data partitioning. SW-Store [AMMH09] is one of these systems. It vertically partitions the above-cited property table into L fragments (where L represents the number of predicates). Each fragment is associated to one predicate and composed of two columns (subject, object), known as *Binary Tables*. This strategy solved the storage problems caused by NULL values, but suffered overhead costs when many predicates were involved in a single SPARQL query. The emerging schemas generated by a strategy proposed in [PPEB15] are the result of splitting the triple table into several three-column tables according to induced predicate's sets.

Summary Non-native systems are considered as naive solutions to store RDF data thanks to different services offered by RDBMSs in terms of storage, optimization, maintenance, etc. Due to the limited schema in a triple (having only three attributes), indexing techniques were widely used to speed up SPARQL queries. Despite this, few attempts were proposed to vertically partition the triple table. This partitioning is performed in the structure of triples and surprisingly ignores the *graph structure of the KG* that represents its *logical schema*.

Native systems The managers of these systems do not rely on existing RDBMSs and build the infrastructure of RDF stores from scratch. Two main storage strategies are used in these systems: *table-* and *graph-driven* strategies. Systems using table-driven strategies store triples in tables. Contrary to non-native storage systems, the data in native systems is encoded and eventually compressed. RDF-3X [NW08] is the most representative system in this category. It stores triples in three-column tables in compressed clustered B+trees. For performance issues, intensive indexes are built considering all the permutations of the columns (e.g., SPO, OPS, PSO) and more others. Systems following graph-driven strategies, like gStore [ZÖC+14] use physical implementations of graphs such as adjacency lists. The optimizations of these systems are strongly dependent on the graph implementation structures (e.g., V*-tree indexes in gStore).

Summary Native systems follow the same philosophy as the non-native ones in using intensive indexing techniques to speed up SPARQL queries. In both systems, optimizations are performed at the physical level. Logical optimizations (such as data partitioning) that have been widely used in relational databases did not have their rightful place.

Researchers and managers of triple stores have grasped the great opportunity offered by the large availability of new programming paradigms, big data frameworks, cloud computing

platforms to develop scalable stores. Unfortunately, this opportunity was not available in the golden age of Relational Databases. In the following, a short introduction is given for distributed and parallel triple stores.

Distributed and Parallel Triple Stores

For scalability issues, a large panoply of distributed and parallel triples stores exists. The life cycle of developing such a store can be easily inspired by the one used for designing traditional distributed and parallel databases [BBC14]. It has the following steps: (1) choosing the hardware architecture, (2) partitioning the target KG (that we call the **fragmenter**), (3) allocating the so-generated fragments over available nodes (called **allocator**), (4) replicating these fragments for efficiency purposes, (5) defining efficient query processing strategies, (6) defining efficient load balancing strategies, and (7) monitoring and detecting changes. Data partitioning is a precondition to ensure the deployment, efficiency, scalability, and fault tolerance of these systems [ÖV11]. The sensitivity of RDF data partitioning and its impact on the performance of RDF stores has been recently discussed in a paper presented in a SIGMOD Workshop in 2020 [JSL20].

Data partitioning has to be transparent for the consumers of triple stores, in the sense that they *should not care about how the data was partitioned*. In contrast, for the managers, data partitioning is *not transparent*. This is because they have to define, evaluate, and tune different data partitioning techniques that fit their data. By analyzing the major distributed and parallel triple stores, we propose to classify them into two main categories based on the degree of transparency granted to the designers of these systems: *transparent*[7] and *opaque* systems. In transparent systems, the data partitioning is delegated to the host (like a cloud provider) offering services to manage the triple store. For instance, the *Cliquesquare* system [GKM+15] is built on top of the Hadoop framework who is in charge of performing the data partitioning. In these systems, the RDF data partitioning is hidden from local managers. In contrast, in opaque systems, all elements of the data partitioning environment are mastered by the managers. The different partitioning modes of RDF data are part of this environment (e.g., *Hashing*, *Graph partitioning*, and *Semantic hashing* [JSL20]). These systems are usually deployed in a master/slave architecture in which a *fragment* of triples is distributed on each slave (e.g., RDF-3X). At the query runtime, the master node (which has a catalog of the data) sends queries to each slave and if necessary, combines the results to produce the final answer.

From the above presentation including centralized, distributed, and parallel triple stores, three main lessons are learned:

1. A strong demand from managers and consumers for developing triple stores by exploiting the technological opportunity in terms of software and hardware is noticed. More concretely, the DB-Engine website[8] ranks 19 RDF stores in October 2020. These stores span centralized and parallel triple store systems.

2. Any initiative to use data partitioning in triple stores should benefit from the large experience of academia, commercial DBMSs editors, and Open Source developers (e.g., the case of PostgreSQL that supports nicely data partitioning) in defining and using data partitioning in the context of traditional databases. This experience has to be capitalized to augment the *reuse* and *reproducibility* of the findings of data partitioning in RDF data. This capitalization is done by an exploitation process of the whole environment of data partitioning.

3. The necessity for developing an RDF data partitioning framework for managers willing to design centralized and parallel triple stores.

[7] Transparency is associated to the data partitioning
[8] `https://db-engines.com/en/ranking/rdf+store`

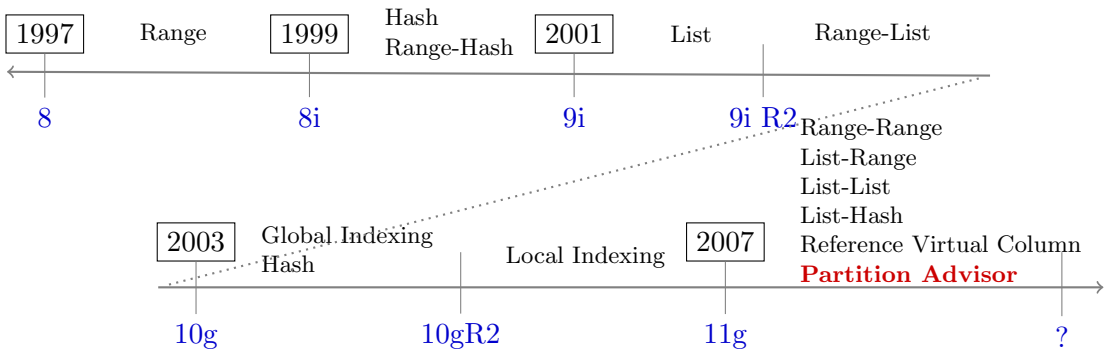


Figure 6: Horizontal partitioning evolution in Oracle RDBMS

Data Partitioning in Triple Stores: Why reinvent the wheels?

To explicit the different elements of RDF data partitioning, let us overview data partitioning in the context of traditional databases in order to *reuse* its success stories. Data partitioning has been first considered as a logical technique for designing relational databases and then as a physical optimization after the amplification of physical design motivated by the arrival of the data warehousing technology [ANY04]. It is used either in centralized and distributed/parallel databases. Three main types of data partitioning exist (vertical, horizontal, and hybrid). These types will be largely discussed in Chapter 1. Several partitioning modes have been proposed and evolved after each generation of databases. Two types of modes exist single (range, list, and hash partitioning) and composite (combining the single ones) modes. Figure 6 shows the spectacular adoption and services offered by editors of DBMSs such as Oracle. Native data definition language support is available for horizontal data partitioning (e.g., `CREATE TABLE ... PARTITION BY RANGE(...)`). The decision of partitioning a table is delegated to the database administrator. The main characteristic of data partitioning is its ability to be combined with some redundant optimization techniques such as indexes and materialized views. This implies that the declaration of the partitions of a table is performed before deploying a given database.

Two versions of horizontal partitioning are available [CNP82]: primary and derived horizontal partitioning. Primary horizontal partitioning is performed to a single table T using simple predicates defined on T. A simple predicate has the following form: $Attr\ \theta\ value$, where $Attr$ is an attribute of T, $\theta \in \{<, >, =, \leq, \geq\}$ and $value \in domain(Attr)$. It can be performed using the different fragmentation modes above cited. Derived horizontal partitioning is the result of the propagation of a partitioning schema of a table T on a table R. The derived horizontal partitioning is feasible if a parent-child relationship exists between T and R. Due to the difficulty of selecting an optimal data partitioning scheme for a given database, several algorithms with different strategies (including affinity-based [NCWD84], graph-based [NR89], minterm-based [ÖV11], cost model-based [RZML02]) have been proposed to satisfy a set of non-functional requirements such as query performance, network transfer cost, number of final fragments, etc. Two main classes of these algorithms are distinguished:

(i) *Query-driven algorithms:* that require an a priori knowledge of a representative workload as an input [ÖV11, NR89];

(ii) *Data-driven algorithms:* which are independent of the representative query workload.

Due to the complexity of performing data partitioning, several academia and DBMS editors have proposed wizards for assisting designers (database administrators) in their day-to-day tasks when managing partitions. The most important advisors are shown in Table 1. From this table we can figure out that data partitioning and indexes are available in all mentioned advisors. This is due to their quality in improving queries.

Name	**Type**	**Supported optimizations**			
		P[a]	I[b]	MV[c]	C[d]
SQL Database Tuning Advisor (DTA)	Commercial	✓	✓	✓	
Oracle SQL Access Advisor		✓	✓	✓	
DB2 Index Advisor tool [RZML02]		✓	✓	✓	✓
Parinda [MDA+10]	Academic	✓	✓		
SimulPh.D [BBA09]		✓	✓		

[a]Partitioning, [b]Indexes, [c]Materialized views, [d]Clustering

Table 1: Partitioning advisors in [Bel18]

The abundance and the popularity (in terms of research and education) of data partitioning in traditional databases allow us to propose its environment described in an Entity-Relationship diagram in Figure 7 that is commented as follows:
Data Partitioning (*DP*) is the entry point of our diagram. A *DP* has a name (Part_Name), a type (Part_Type, i.e. horizontal, vertical or hybrid), a version (i.e. primary or derived) and an inventor described by either the title of her/his published paper(s) describing this method or the name of the system implementing it. A *DP* strategy is applied to one or many *Entities*. An Entity is identified by a schema (e.g., table, class) and an extension (e.g., tuples in a table, instances of a class) stored in a given Entity Store (e.g., DBMS) that is deployed in a platform. A *DP* strategy is designed to meet one or many Objective functions measured by mathematical cost models [OOB18]. Each Cost Model has a Name (Cost_Name) and a Metric corresponding, for instance, to CPU, Inputs/Outputs, and Transfer costs. A *DP* strategy has to satisfy one or several Constraints and must be implemented by a set of algorithms. An Algorithm has a name (Algo_Name) and a type (Algo_Type, i.e. Data-driven or Query-driven). Finally, a *DP* strategy has one or many partitioning modes (Mode). A Mode has a name (Mode_Name) and a type (Mode_Type, i.e. simple or composite).

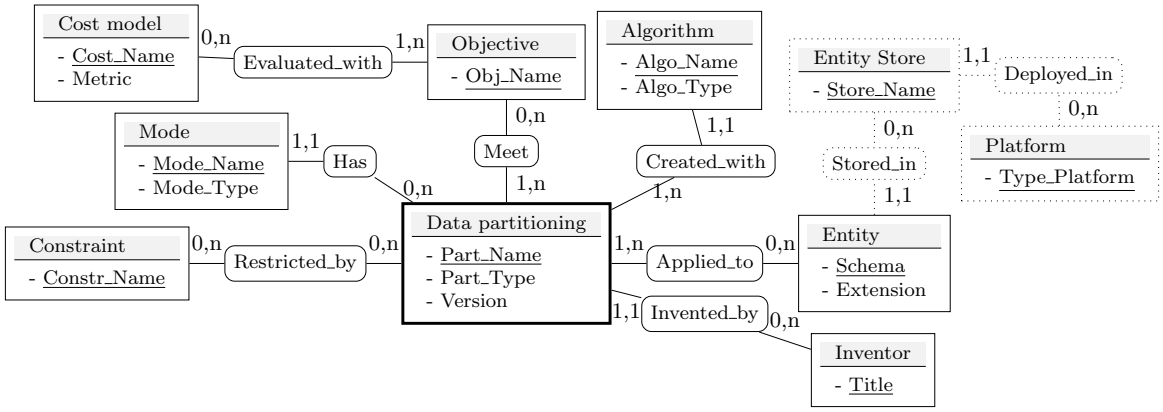


Figure 7: Partitioning environment in relational databases

Figure 8 gives an excerpt instantiation of our model for the Horizontal Partitioning method defined in [BBRW09]. The authors in this paper proposed primary and derived horizontal partitioning schemes for a relational data warehouse schema stored in a centralized RDBMS. A cost-driven algorithm is given to generate a number final fragments of a fact table (which must not exceed a threshold fixed by the data warehouse administrator). The obtained fragments have to minimize the number of inputs-outputs (I/Os) when processing a set of OLAP queries known in advance. The traditional modes (range, list, and hash) are applied to the obtained final fragments of fact and dimension tables.

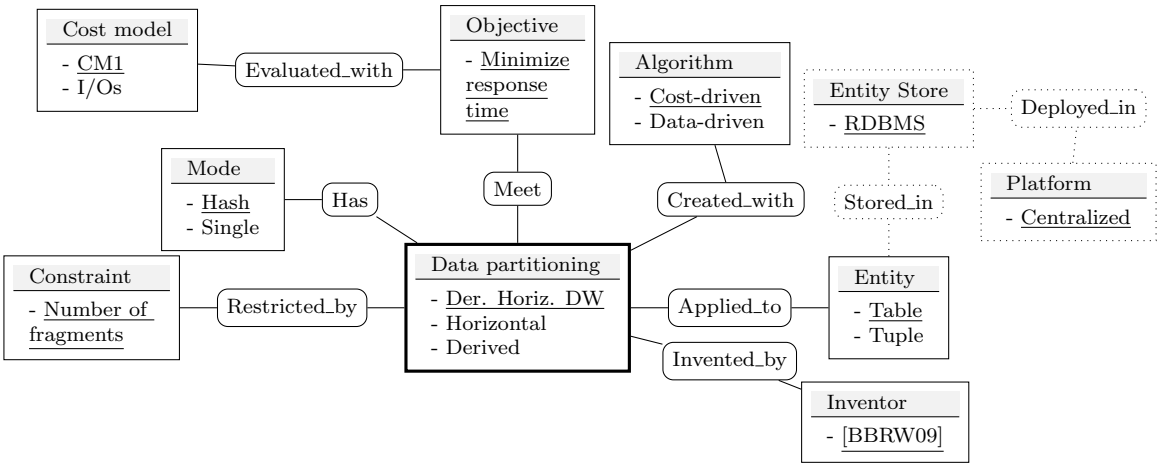


Figure 8: Horizontal partitioning in [BBRW09]

The Reuse of Data Partitioning in Relational Databases in other Generations

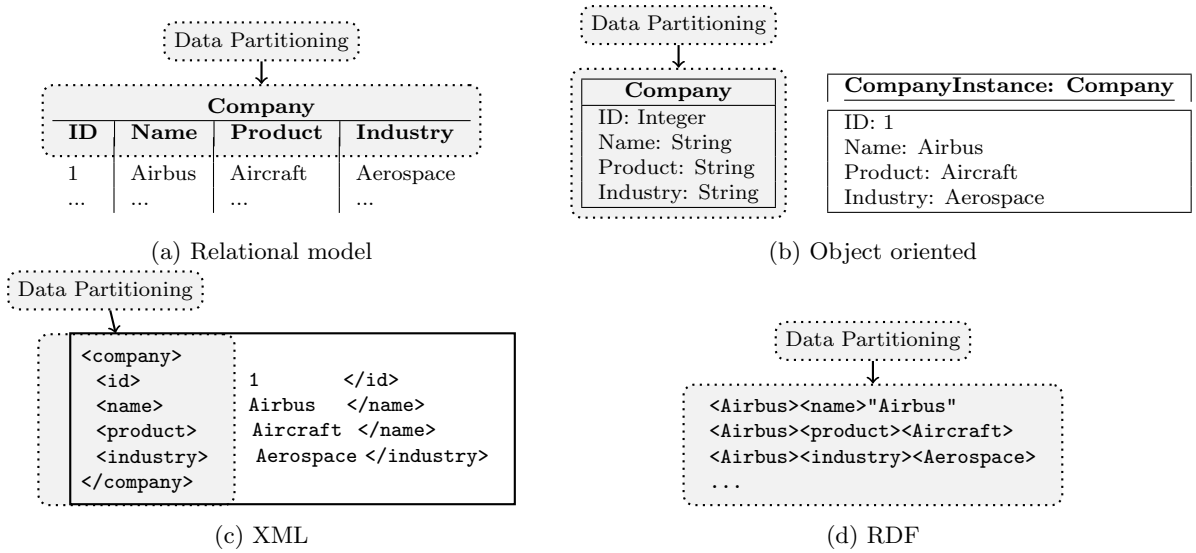
The lessons of relational databases served as basis for many other data models that came after. Let us take for example the object-oriented model whose partitioning (of classes) was initially explored by [CWZ94] and then implemented in [KL95, KL00]. Later, horizontal and vertical partitioning methods for classes were proposed in the literature by [BKS97] (referential horizontal partitioning [BKL98]) and [FKL03] respectively. Similar examples can be mentioned in data warehouses (e.g., horizontal fragmentation of the fact table in a star schema [SMR00] and the use of referential partitioning in the dimensional data model [BBRW09]) and semi-structured models like XML (e.g., [BLS09]).

So, we wonder, why this type of approach has not been seen so far in RDF databases? Because without a doubt, data partitioning is currently at the center of the debate regarding its great influence on the performance of triple stores. Making a partitioning environment *explicit* for triple store managers can help not only to understand current strategies but also to set the grounds for future extensions. The Entity-Relationship diagram shown in Figure 7 does not have any component related to the physical storage structure of the data. Indeed, *DP* strategies are performed to the entities that *logically* represent the data. The manager of a relational database should not worry about how exactly the data are physically represented on the disk. In fact, the physical storage structures are proper to each RDBMS. The *key* of data partitioning in these models is their logical layer. This layer allowed the definitions of algorithms, cost models, constraints, etc. that are independent of particular system implementation. In the following, we address how this layer could be introduced for RDF datasets.

The Logical Layer as a Key to RDF Partitioning

To analyze what was the factor that allowed many models to take ideas, techniques, algorithms, etc. from the relational model and adapt them to their liking let us consider the example illustrated in Figure 9. In this Figure, we compare the partitioning of a table, a class, a document, and a set of RDF triples. In Figures 9a, 9b, and 9c we see that data partitioning is applied first from a the logical level, regardless of the tuples inside the table, the instances of the class or the values inside the tags of the XML document. Data partitioning is then applied to high-level entities (such as a table, a class, or the schema of the XML document). This characteristic allowed to *naturally* reuse/map/adapt the partitioning strategies of relational databases to all these other models.

In contrast, in RDF, the data are partitioned at a much finer granularity (as shown in Figure 9d). Let us remember that the triple is the basic representation unit in RDF. This gives the



(a) Relational model

(b) Object oriented

(c) XML

(d) RDF

Figure 9: Partitioning example by data model

model a lot of flexibility, but it certainly complicates the reuse of data partitioning techniques as it has been done in other models. In RDF, the data are partitioned with merely physical strategies using the triple as fragmentation unit. This does not mean that RDF triples do not have any schema. This schema exists, but contrary to the models illustrated in Figures 9a, 9b, and 9c, it is *implicit*. This implicit schema can be identified with various techniques based on the hierarchy of classes and ontologies (using the RDFS and OWL schemas for example) of the predicates, or relying only on the structure of the data. So, if logical entities can be detected, why not using/adapting them to propose partitioning techniques inspired by the relational model so widely studied?

This motivates us to provide triple store's designers with a framework to partition RDF data from a logical level first. We do not seek to change the features that made RDF flourish but to incorporate a common logical layer to the design process of RDF partitions. The logical layer will allow designers to work on structures with a higher level of abstraction than that offered by the triples. To regroup the triples, we propose two strategies: by instances and attributes. These structures are defined in Chapter 3 of this thesis. The first strategy analyzes each node of the RDF graph and its outgoing edges. The set of predicates of a subject node characterize it [NM11]. In the example of Figure 9d, the Airbus node is characterized by the predicates `has_name, has_product` and `industry`. We refer to a node and its outgoing triples as a *forward data star*. Making an analogy with the object-oriented model, a forward data star is an instance of some class. To identify to which high-level entity a *forward data star* belongs, we can apply two types of techniques that use structural or semantic clustering. The groups of forward data stars form a *forward graph fragment* and they harmonize with the notion of horizontal partitions. Vertical partitions on the other side are identified by grouping the nodes with their *ingoing* edges in *backward data stars*. The backward stars are further grouped in *Backward graph fragments*. These structures allow defining an explicit schema for RDF data. They are the baseline of our framework allowing triple store managers to partition RDF graphs based on logical structures. Our framework allows managers (designers) to partition in an *informed* way, considering the inherent graph structure of the data, their connectivity, and other constraints related to the partitioning environment (e.g., system's constraints). In the following sections, we describe the components of this partitioning framework.

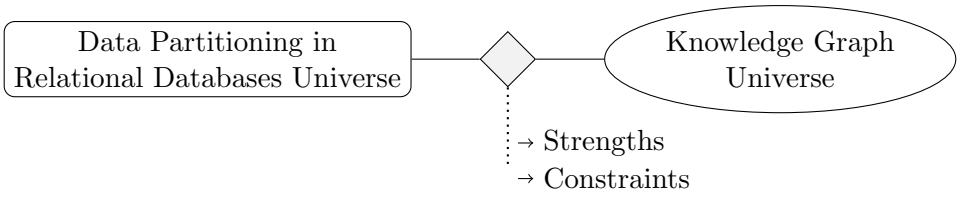


Figure 10: Fusing data partitioning to both worlds

Thesis Vision

In this thesis, we position ourselves as a manager of a triple store to satisfy the creator needs described in a manifest (Figure 11). This manifest contains the KG, nonfunctional requirements, constraints, and offered services imposed by the creator. We favor the usage of data partitioning in designing triple stores by reusing its strong points identified in the traditional databases. Our RDF data partitioning is proposed to be used either in new triple stores (the case of our RDF_-QDAG store currently developed in our LIAS Laboratory) or in already designed ones (such as gStoreD).

We claim that the implementation of our vision passes through the following tasks:

1. A deep understanding of data partitioning proposed in the context of centralized/parallel/distributed traditional databases.
2. The reuse of its strong points in terms of techniques and tools. To reach this objective, an establishment of a concise and complete survey of data partitioning is required followed by a reproduction of its principles and strong aspects to RDF data. Figure 10 shows the connection between data partitioning and KG environments.

The objectives that we set in our thesis are:

(i) The definition of a common framework for centralized and parallel triple stores with comprehensive components implementing our vision.

(ii) The proposal of efficient data-driven partitioning algorithms defined on logical structures of RDF as in relational databases.

(iii) The instantiation of our framework in the centralized and parallel triple stores by taking into account the requirements of consumers in terms of constraints and nonfunctional requirements such as the performance of SPARQL queries.

(iv) Making tools available for designers including a fragment manipulation language and a wizard (an advisor) to assist them in their tasks. Both are based on the logical fragments that we previously defined.

Contributions

The main contributions of our work are:

- With the motivation of increasing the reproduction and the reuse of important findings of the data partitioning in the of traditional centralized and parallel databases in mind, we propose a complete survey of this problem covering the important generations of the database world.
- The definition of a Framework, called *RDFPartSuite* illustrated in Figure 11. It supports our vision in designing centralized and parallel triple stores. *RDFPartSuite* can be personalized to the type of the platform. One of the main characteristics of our RDFPartSuite

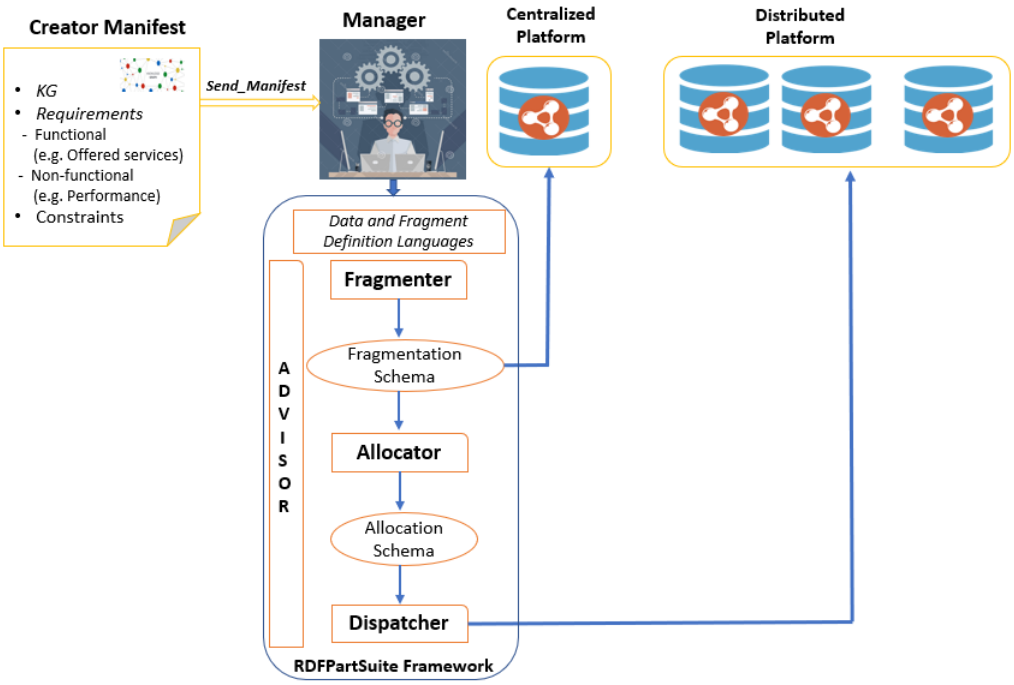


Figure 11: RDFPartSuite Framework

is that it follows the paradigm outside a DBMS [Ord13], in which all technical efforts are performed externally from the triple stores. Based on the type of the platform two scenarios are possible:

1. If the platform is centralized, our framework activates the module fragmenter in charge of partitioning triples based on their logical representation. The definition of these fragments is sent to the target triple store. After that, triples of the creator KG are loaded in this store (in our case RDF_QDAG).
2. When the platform is distributed or parallel, the fragmenter sends the obtained fragmentation schema to the allocator component. This latter role is to assign the different fragments to nodes of the target platform. This allocation is performed based on a *data-driven* approach (i.e. considering the *inherent connectivity between fragments*). Once the allocation schema is obtained, the dispatcher component sends the detailed description of this schema to different nodes and then loads the triples according to their definitions.
3. Assistance tools are proposed to help managers in their tasks. These tools take into account the expertise level of the manager. If she/he is an expert in the studied problems, she/he can easily perform different tasks offered by the framework. For that she/he needs a language to describe and define the different schemes (fragmentation/allocation/dispatcher). In the case, she/he is not an expert, we propose her/him an advisor as in traditional DBMS editors.

- The definition of a set of logical entities (named graph fragments) used to perform the fragmentation of RDF datasets at a logical level in centralized and parallel environments. This is a step forward with respect to the current strategies used by the triple stores that use merely physical fragmentation strategies that depend on the specific storage structure used to represent the triples.

- The deployment of this framework to the loading module of a centralized system (RDF_QDAG) and of a distributed triple store (gStoreD). [PZÖ+16]).

Thesis Outline

The reminder of our manuscript is composed of two main parts as illustrated in Figure 12.

First part: State-of-the-art and Objectives

The first part presents our state of the art and background concepts revolving around two chapters. In Chapter 1, we develop a complete survey of data partitioning. We conduct a comprehensive review of the literature on the data partitioning problem with the aim of letting the reader get a complete overview of the problem in terms of definitions, variants, and constraints. We define the data partitioning foundations along with ten dimensions to classify the spectrum of works related to this manner. It sets the reader in the whole history of the evolution of the data-partitioning concept.

Then, Chapter 2 starts giving a general overview of the graph data model. We detail its logical and storage structures, query processing strategies, and languages. Next, to better delimit our work, we focus on the many existent RDF systems. We start by giving some background concepts about RDF storage, processing, and partitioning strategies. We classify the existent systems according to their fragmentation, allocation, and replication strategies. We conclude this section by comparing the partitioning strategies seen in Chapter 1 and those of RDF systems discussed in Chapter 2. This comparison serves as the basis for setting out the objectives of our contributions.

Second Part: Contributions

We detail in this section our two main contributions.

In Chapter 3, we describe the fragmenter and allocator components of our framework. We start by formally defining the logical entities used to partition a KG. We detail the algorithms to generate them from raw RDF datasets. Then, we defined the allocation problem of the fragments identified previously. We showed that the connectivity of graph fragments can be expressed in a graph. Thereby, we proposed some heuristics to partition the fragments based on graph-partitioning techniques.

In Chapter 4, we implement our framework in a centralized and parallel triple store. We integrate first our findings into the loading module of a graph-based centralized triple store. This system, RDF_QDAG [KMG+20], stores the triples using the logical structures defined in the previous chapter. We start by giving an overview of this system describing its storage, optimization, and processing modules. Then, we evaluate RDF_QDAG in three main stages: data loading, fragmentation, and allocation. We compared its loading costs and query performance with respect to other representative systems of the state of the art. We use real and synthetic datasets with variable sizes to test the system's scalability. Then, we show the integration process of our framework to the loading stage of the data to gStoreD[PZÖ+16]. Finally, we describe the present Assistance tools as extensions to our Framework. These tools include a Data Definition Language and an RDF partitioning advisor.

Last Part: Conclusions and Perspectives

This chapter concludes the thesis by providing a summary and an evaluation of the presented work. Lastly, this chapter discusses several opportunities for future works.

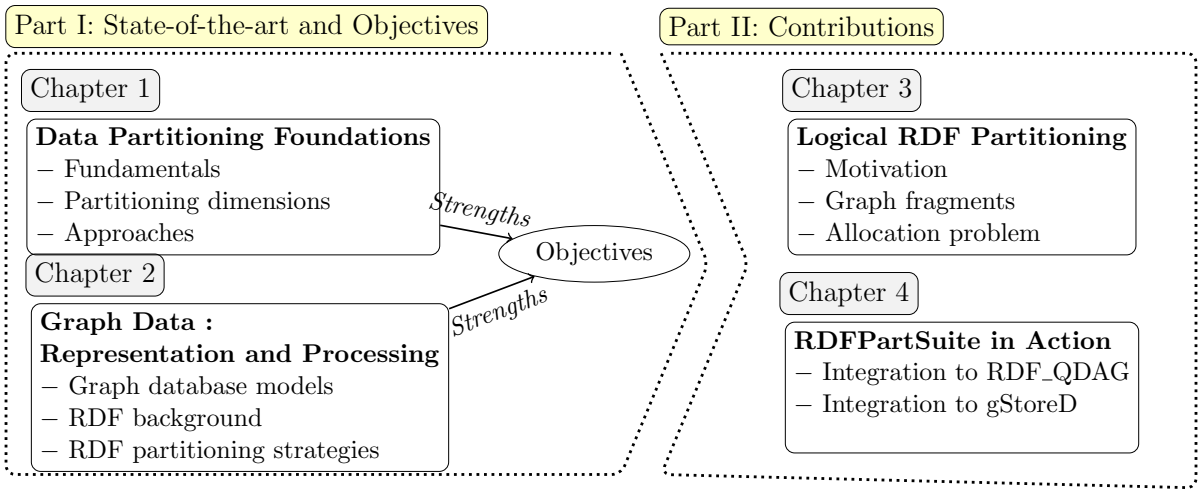


Figure 12: Breakdown of thesis chapters

Publications

International journals

1. Abdallah KHELIL, Amin MESMOUDI, Jorge GALICIA AUYON, Ladjel BELLATRECHE, Mohand-Saïd HACID, Emmanuel COQUERY. Combining Graph Exploration and Fragmentation for RDF Processing. Information System Frontiers (Q1), Springer, (2020), DOI: 10.1007/s10796-020-09998-z.

International conferences

1. Ishaq ZOUAGHI, Amin MESMOUDI, Jorge GALICIA AUYON and Ladjel BELLATRECHE. Query Optimization for Large Scale Clustered RDF Data. Proceedings of the 22nd International Workshop in Design, Optimization, Languages and Analytical Processing of Big Data - DOLAP (Core B), Copenhagen, Denmark, (2020), pp. 56–65, URL: http://ceur-ws.org/Vol-2572/paper18.pdf.

2. Jorge GALICIA AUYON, Amin MESMOUDI and Ladjel BELLATRECHE. RDFPartSuite: Bridging Physical and Logical RDF Partitioning. Proceedings of the 21st International Conference in Big Data Analytics and Knowledge Discovery - DaWaK (Core B), Vienna, Austria (2019), pp. 136–150, DOI: 10.1007/978-3-030-27520-4_10.

3. Jorge GALICIA AUYON, Amin MESMOUDI, Ladjel BELLATRECHE and Carlos ORDONEZ. Reverse Partitioning for SPARQL Queries: Principles and Performance Analysis. Proceedings of the 30th International Conference in Database and Expert Systems Applications – DEXA (Core B), Vienna, Austria, (2019), pp. 174–183, DOI: 10.1007/978-3-030-27618-8_13.

4. Abdallah KHELIL, Amin MESMOUDI, Jorge GALICIA AUYON and Mohamed SENOUCI. Should We Be Afraid of Querying Billions of Triples in a Graph-Based Centralized System? Proceedings of the 9th International Conference in Model and Data Engineering -MEDI, Toulouse, France, (2019), pp. 251–266, DOI: 10.1007/978-3-030-32065-2_18.

National conferences

1. Abdallah KHELIL, Amin MESMOUDI, Jorge GALICIA, and Ladjel BELLATRECHE. EXGRAF: Exploration et Fragmentation de Graphes au Service du Traitement Scalable

de Requêtes RDF, 16e journées EDA "Business Intelligence & Big Data", Lyon, France (2020), pp.47-60

2. Jorge GALICIA AUYON, Amin MESMOUDI and Ladjel BELLATRECHE. RDFPartSuite: Bridging Physical and Logical RDF Partitioning. Actes de la 35th Conférence Internationale sur la Gestion de Données – Principes, Technologies et Applications -BDA, Lyon, France, (2019).

Submitted papers

1. Ishaq ZOUAGHI, Amin MESMOUDI, Jorge GALICIA AUYON and Ladjel BELLATRECHE. GoFast: Graph-based Optimization for Efficient and Scalable Query Evaluation. Information Systems (Q1).

2. Jorge GALICIA AUYON, Amin MESMOUDI, Ishaq ZOUAGHI and Ladjel BELLATRECHE. The Wisdom of Logical Partitioning as a Foundation for Large-Scale RDF Processing. Data and Knowledge Engineering (Q2), Elsevier.

Part I

Preliminaries

Chapter 1

Data Partitioning Foundations

Contents

Summary In our Introduction, we have discussed the importance of describing in detail the partitioning environment for designing *opaque* systems. In this chapter we conduct a comprehensive survey of the literature regarding the partitioning problem in relational databases. We look at the fundamentals of the problem and its evolution over all the database generations. Our aim is to give a complete overview of how the problem was treated in these systems identifying its main elements and strong points. The chapter is organized as follows. We start in Section 1.2 citing the works that led to the explicit definition of partitioning stating the data partitioning foundations. Then, in Section 1.3 we define the problem and its main dimensions that are ergo used to classify the surveyed works. Next, Section 1.4 organizes the works according to these dimensions giving a high-level comparison and classification of existing partitioning approaches. We introduce the partitioning approaches in the most recent large-scale platforms in Section 1.5. Finally in Section 1.6 we give insights to open problems and compare the approaches presented in this chapter.

1.1 Introduction

Within the most studied optimization techniques in databases, data partitioning occupies a very important spot not only for its effectiveness to improve the performance of the database, but also because in modern large-scale parallel and distributed architectures it is a mandatory stride. Data partitioning notions were introduced in the seventies in [HS75] shortly after the definition of database indexes. Along with materialized views and indexes, data partitioning was at an early stage part of the same pool of optimization strategies sketched during the physical design phase [LGS+79]. These strategies seek to improve the speed of retrieval operations reducing the data transferred in the network in a distributed database, or between primary and secondary storage in a centralized system.

In relational databases, partitioning refers to splitting what is logically one large table into smaller physical pieces. Similar to indexes, the data partitioning problem has been covered in all generations of databases from deductive [Spy87, NH94], objected-oriented [BKS97] to data warehouses [SMR00]. Still, contrarily to other optimization strategies and before the introduction of dynamic partitioning techniques, partitions are created during the declaration of the database tables. This feature distinguishes data partitioning from other optimization techniques that can be created on-demand while the database is running. Recreating partitions after its definition is costly and unpractical. Numerous tools have been developed to facilitate the creation of partitions, for instance physical design advisors (e.g [RZML02], DB2 advisor [ZRL+04]) and data definition languages offered by some Database Management Systems (e.g., Oracle, PostreSQL, MySQL).

For example, let us consider the table *Airplane* in Figure 1.1 which will be used throughout this chapter. This table partitioned on attribute Dev is declared in PosgreSQL with following Data Definition Language (DDL) as:

```
CREATE TABLE Airplane
(
ID integer PRIMARY KEY,
Model varchar(15),
Dev varchar(15),
Length real,
Cost real
)
PARTITION BY LIST (Dev)
```

In this statement, the table *Airplane* is partitioned listing the key values for each partition creating two groups based on the airplane's developer (*Dev*): Airbus and Boeing. The `PARTITION` statement in this DDL should not be confused with the SQL operator `PARTITION BY` used in OLAP window functions. Although declaring partitions can be done very simply, choosing the attribute(s) and the partitioning method that optimizes the performance and the efficiency at query runtime is a very complex problem due to the large number of variables and possible solutions to consider.

ID	Model	Dev	Length	Cost
1609	A320	Airbus	37.57	101
1752	A350-900	Airbus	64.75	317.4
1909	A340-600	Airbus	75.36	300.1
1912	B737-800	Boeing	39.5	102.2
2212	B777-300	Boeing	73.86	361.5

Figure 1.1: Aircraft table

We characterized the partitioning problem according to its:

- **Maturity**: the problem has existed almost since the definition of the first database systems and has been largely explored,
- **Coverage**: it has been implemented for all generations of databases,
- **Evolution:** it has been adapted to the different architectures, needs and constraints, and
- **Complexity:** finding an optimal partition scheme needs to consider several alternatives and sometimes mutually exclusive objective functions.

The research devoted to this topic has been very rich and extensive. Some surveys exist in the literature devoted to specific partitioning types. For example, [GDQ92, MS98] are devoted to certain partitioning strategies whose main objective is called declustering.

In this chapter we conduct a comprehensive review of the literature of the partitioning problem with the aim of letting the reader get a complete overview of the problem. We start in Section 1.2 citing the works that led to the explicit definition of partitioning stating the data partitioning foundations. Then, in Section 1.3 we define the problem and its main dimensions that are ergo used to classify the surveyed works. Next, Section 1.4 organizes the works according to these dimensions giving a high-level comparison and classification of existing partitioning approaches. We introduce the partitioning approaches in the most recent large-scale platforms in Section 1.5. Finally in Section 1.6 we give insights to open problems and compare the approaches presented in this chapter.

1.2 Data Partitioning Fundamentals

In this section we recall the main features and characteristics of the partitioning problem to unify the terminology used throughout the thesis. We first mention the works that led to the definition of what we called the *core* partitioning problem. Then we give definitions and examples of the different variants of partitioning in relational databases. Each variant is motivated highlighting its performance impact at query runtime. Finally, we explain the dimensions used to classify the partitioning algorithms.

The database community strives to provide systems that store, update, administrate and retrieve information ensuring the integrity, consistency and security of the stored data and transactions offering a reasonable performance. Performance optimization strategies for database systems have been largely studied especially the ones applied during the database design stage. Three optimization strategies for centralized systems were clearly distinguished already in 1976 in [ES76]:

1. *Hardware*: physically increase the speed at which data are transferred between primary and secondary memory,
2. *Encoding*: improve data encoding strategies to increase logically the information content transferred to main memory,
3. *Physical design*: selectively transfer only those physical records or data attributes which are actually required by the application.

Even if originally these strategies targeted centralized database systems, they fit to the parallel and distributed contexts when the network constraint is considered as well at each point. The last category of techniques (*Physical design*) comprises indexes, materialized views and partitions all being part of the physical design process conceived by the database designer. Partitioning was defined as an optimization stage that evolved into a mandatory design step in distributed and parallel systems. Next, we introduce the definition and evolution of the partitioning problem.

1.2.1 Partitioning definition and development overview

Let us define the partitioning process in general as the action or state of dividing or being divided into parts[1]. In the database context the data are divided to form partitions. The earliest partitioning studies in information systems dealt with the optimal distribution of entire files to nodes on a computer network. These works (e.g., [Chu69, Esw74]) solve the problem which arises in the design of distributed information systems where the data files are shared by a number of users in remote locations.

Partitioning in relational databases was introduced later disclosing how partitioning a relation contributes to improve the system's performance and *implicitly* describing how partitions are built. In 1975 an article by Jeffrey Hoffer and Denis Severance [HS75] motivated data partitioning as a technique to reduce the query execution time and implicitly introduced two types of partitioning configurations. These configurations use clustering notions to distribute the data in subfiles. They showed that the query response time in a centralized database is minimized if each relation is partitioned either at the attribute or at the record level, presenting the first notions of horizontal and vertical partitions. Their main motivation is the awareness that the query's retrieval time is dominated by the time spent to transport data from the disk to main memory before processing. They formalized their ideas and proposed an algorithm clustering the attributes of a relation in a centralized database system. This first formal definition to a partition is shown in Definition 1.1.

Definition 1.1 ***Partition*** *[HS75] Given a set of attributes A of a relation R, consider the collection of subfiles $S = \{(A_i, R_i)\}$ and $A_i \subseteq A$, $R_i \subseteq R$, specifying the attributes and entities represented in a subfile i. S clusters the data in two basic forms:*

1. *By attributes in which $R_i = R$ and $\bigcup_{i=1}^{n} A_i = A$ for $i = 1...n$.*
2. *By records in which $A_i = A$ and $\bigcup_{i=1}^{n} R_i = R$ for $i = 1...n$.*

A cluster S in which $A_i \cap A_j = \emptyset (R_i \cap R_j = \emptyset)$ for all $i \neq j$ is named an attribute (record) partition for the database.

The same team of researchers extended its work proposing an integer programming formulation for the problem considering the case when the database is shared by a group of users [Hof76, ES76, MS78]. The terms *horizontal* and *vertical* partitions were adopted by the database community to define partitions performed at the record and attribute level of a relation respectively. The works of Eisner M., Hoffer J. and Severance D. described previously were the first efforts towards creating partitioning algorithms for centralized database systems.

The previous findings work for both centralized and distributed database systems. In both cases the main objective is to benefit data locality by *clustering* the records or attributes according to a defined degree of togetherness. Centralized and distributed partitioning strategies promote data locality, a strategy in which related records/attributes are often stored in close proximity to minimize performance overhead in access path navigation. In centralized systems, attribute and record clustering are used to gather in main memory the most frequently accessed records or records subsets to reduce the number of input/output (I/O's) to the primary storage. In a distributed database system, the objective is as well to cluster attributes or records but with the aim of decreasing the number of distributed transactions considering that for these systems the network traffic is the bottleneck of query processing.

A special kind of distributed database systems that exploit the parallelism to deliver high-performance and highly available systems was introduced not long after the boom of distributed databases in the eighties. This kind of database system, named *parallel* is formed by a set of processing units connected by a very fast network to process large databases efficiently. Three

[1]Taken from the Oxford dictionary

Table 1.1: High-level variants of the partitioning problem

<table>
<tr><th>Stage</th><th>Platform</th><th>Type</th><th>Mechanism</th><th>System element</th></tr>
<tr><td>Optional</td><td>Centralized</td><td rowspan="2">Horizontal
Vertical</td><td rowspan="2">Clustering</td><td>Memory & Disk</td></tr>
<tr><td rowspan="2">Mandatory</td><td>Distributed</td><td rowspan="2">Memory, Disk & Network</td></tr>
<tr><td>Parallel</td><td>Horizontal</td><td>De-clustering</td></tr>
</table>

forms of parallelism are exploited in these systems: i)inter-query parallelism which enables the parallel execution of multiple queries, ii)intra-query parallelism making the parallel execution of independent operations in a single query and iii)intra-operation parallelism in which the same operation is executed as many sub-operations [ÖV96]. Contrarily to distributed databases, localizing the query execution in a single node is not the preferred solution. In this case, the performance might be degraded due to excessive queuing delays at the node and the parallelism will not be exploited. Data partitioning for parallel databases implements the so-called *declustering*, a strategy which spreads the data among the processing nodes to ensure parallelism and load balance. The declustering term was introduced in 1986 in [FLC86] to solve the multi-disk data allocation problem. This term was later used in the early parallel database systems (e.g., Gamma [DGG+86]) that horizontally partitioned and distributed the data across multiple processing nodes.

Still, in many situations the system is really a hybrid between parallel and distributed, specially in shared-nothing architectures. As mentioned in [ÖV96], the exact differences between parallel and distributed database systems are somewhat unclear. In this thesis we distinguish the partitioning strategies in both systems, but as seen in the next sections, it is the main objective of the system which guides the choice for the best distribution strategy. Table 1.1 organizes the information formerly described. In the following section we explain the evolution of the problem and the dimensions that we applied to describe the partitioning environment.

1.2.2 Partitioning concept evolution

Partitioning a relational database implies dividing tables into parts (or partitions). As we previously mentioned, the partitions are subsets of the relation at the attribute or record levels. The core definition of the problem has not changed over time, however, the problem has evolved with the consideration of additional constraints, different platforms and ergo new objectives. The first notions of data partitioning are related to the distribution of entire files in a centralized or distributed system. Then, as a result of the work of [HS75], the notion of partition as a subset of a relation starts being considered. Formerly, partitioning was used as a strategy to optimize the system in the same way as indexes in a non-mandatory process. With the introduction of parallel systems, the partitioning main objective changed and therefore the partitioning algorithms were adapted to this type of systems. In this section we detail the works leading to the definition of partition and its types. We mention as well the first algorithms for each partitioning type.

1.2.2.1 File allocation

Partitioning has its origins in file allocation models performed on the basis of entire files. Distributed computer systems were introduced before the relational databases entered the market. Optimal file allocation models seek to minimize the overall operating costs of storage and transmission of distributed files across the nodes of a computer network. In 1969 W. Chu modelized file allocation in a computer network as an integer programming problem [Chu69], his work was later enriched by K. P. Eswaran in [Esw74]. These works, along with others, served as bases to

develop the very early works on partitioning algorithms for relational databases.

1.2.2.2 Implicit partitioning definition

The work of J. Hoffer and D. Severance published in 1975 [HS75] was the first work that intuitively showed that subdividing the relation in sub-files (at the attribute or at record levels) minimized the retrieval, storage and maintenance costs of data in a centralized database. They developed a method to cluster the relation's attributes and vertically partition the relation, but no method was established to generate sub-files at the record level (i.e. horizontal fragments). The problem was mathematically formalized by J. Hoffer the next year in [Hof76]. The approach modeled the allocation of subsets of a relation on different physical areas of the system (i.e. primary and secondary memory) using a weighted combination of operational costs for secondary storage, read-only and update requests summed across memory areas. M. Eisner and D. Severance published in 1976 an article [ES76] with a model using a network graphical representation of the user's interactions reducing the retrieval cost in a database shared by multiple users. The work was extended in [MS77] and a complete mathematical approach of the automatic selection of database design was presented by the same authors in [MS78].

1.2.2.3 Explicit partitioning definition

Fragments were explicitly defined as the sub-relations used as distribution units among the nodes of a distributed database system in 1977 in the technical report describing the distributed system SDD-1 [JG77]. The term was reused by Stonebraker et al.[ESW78] in 1978 to describe the partitioning stage in the distributed version of INGRES. In the same year a database design workshop [LGS$^+$79] reunited many database researchers to discuss important issues and future research problems. The terms horizontal and vertical partitions were established at this workshop, defining them as the clustering (or fragmentation using the term introduced in [JG77]) strategies to perform data partitioning and placement. Horizontal partitions refer to record partitioning while vertical partitions refer to partitions at the attribute level. It is important to remark that the term *fragmentation* was used in distributed database systems to separate the processes of partition creation for a relation and the allocation process of physical files to nodes. The separation of the data partitioning process in fragmentation and allocation stages is still carried out in distributed systems [ÖV11, RG00].

Former partitioning algorithms We briefly recall the first algorithms proposed to create vertical and horizontal partitions. Vertical partitioning algorithms were introduced sooner than algorithms to partition the database at the record level. The first algorithm using a similarity measure to group attributes according to their togetherness was proposed by J. Hoffer et al. [HS75] in 1975. The algorithm measured the attribute's affinity using the bond energy algorithm BEA developed in [JSW72]. Based on [HS75]'s findings, S. Navathe et al. published in 1984 [NCWD84], a two phase approach for vertical partitioning. Their algorithm was extended by Cornell and Yu in [CY87]. The detailed list of vertical partitioning algorithm is found in Section 1.4.1.1 of our work. Here we present only the works that served as the fundamentals for other approaches.

Horizontal partitioning algorithms were introduced a few years after the definition of horizontal fragments in 1978. In 1982, S. Ceri et al. published their work [CNP82] presenting a horizontal partitioning algorithm based on the *min-term predicates* defined in their paper. Given a relation R with n attributes represented as $R(A_1, .., A_n)$, a simple predicate p_j is defined as $p_j : A_j \theta\ Value$ where $\theta \in \{=, \neq, <, \leq, >, \geq\}$ and the $Value$ is chosen from the domain of A_j. A *min-term predicate* is a conjunction of simple predicates (or its negation) describing a horizontal partition (or fragment as called in distributed databases) of the relation R. In general, the algorithm consists on finding the minimal set of min-term predicates describing the queries of a

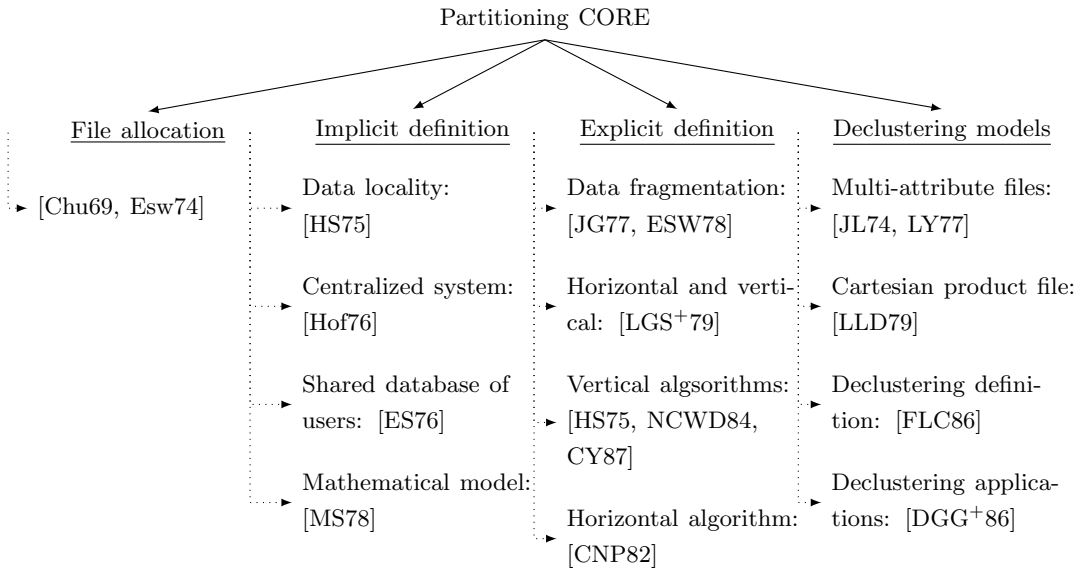


Figure 1.2: Database partitioning origins

workload to afterwards create horizontal partitions based on them. This algorithm was extended by Özsu and Valduriez in [ÖV11]. Before [CNP82], several works were published concerning file allocation (a complete survey is found in [For83]) but none of them proposed a characterization of the access patterns defining a horizontal partition. The complete list of horizontal partitioning approaches is presented in Section 1.4.1.2.

Declustering algorithms In parallel to the works handling horizontal and vertical partitioning in centralized and distributed databases, approaches dealing with the organization and distribution of data on multiple-disk systems started its development from the seventies. These works served subsequently as starting point to the development of partitioning approaches for parallel database systems. Parallel databases exploit hardware architectures in multiprocessor environments. In general, individual database servers are connected with a fast network connection exploiting parallelism (especially in I/O) by distributing database files across multiple processors and/or disks. The partitioning problem in parallel systems exhibits similarities with centralized and distributed database systems. An evident one is that a relation is divided as well into parts to increase the performance. However, on the contrary of distributed and centralized systems that privilege data locality, data are rather spread among the nodes to process in parallel avoiding execution skewness. As it was formalized by Sacca D. and Wiederhold G. in [SW85] and mentioned by [ÖV11] the partitioning problem in these systems is much more complex.

Before describing partitioning strategies for multiple-disks systems let us first mention some multi-attribute file organizations whose fundaments were used to establish the partitioning strategies for parallel database systems. These file organizations contributed to the development of Cartesian product files to which several partitioning heuristics were devoted. Rothnie et al. introduced in 1974 [JL74], a file organization strategy hashing multiple keys for a record. In 1976, J.H et al. [LY77] proposed a multi-dimensional file structure together with multi-dimensional attribute's indexes to reduce the retrieval, update and storage costs of data in which the record address is determined by multiple keys. Their proposal combines a multi-dimensional directory (MDD) and a single key index to find efficiently the solutions for a partial match-queries in a files with attributes having discrete domains. Let us consider for example a file storing records like $(A_1, .., A_d)$ with d attributes. A partial match query defined on this file is query $q : A_i = a_i \wedge A_j = a_j \wedge \cdots \wedge A_k = a_k$. The best strategies of some multi-attribute file

organizations were unified in the specification of Cartesian product files in 1978 [LLD79].

A Cartesian product file, as defined in [MS98], stores records as an ordered *d-tuple* $(a_1, .., a_d)$ of values where each attribute a_i has a finite domain. Let us define D_i as the domain of the *i-th attribute*. A d-attribute file is a subset of $D_1 \times D_2 \times \cdots \times D_d$. When this file is stored on disk, the records are partitioned into buckets or pages. The file is called a Cartesian product file if when each attribute's domain is partitioned into m subsets, the records in $D_{1m_1} \times \cdots \times D_{1m_d}$ are stored in a single bucket. When a multi-attribute query is defined on the file, the buckets containing records qualified by the query are retrieved from the secondary storage. The query cost depends then on the number of buckets retrieved from the disk. To improve the performance, a Cartesian product file is stored on multiple disks, and since each disk is assumed to be independently accessible, the time to respond a query correspond to the maximum number of blocks retrieved by a disk of the system instead of the total amount of blocks retrieved. The problem of distributing files across multiple disks is called *declustering*. The term was introduced in 1986 in [FLC86] although parallel systems like Gamma [DGG+86] already used declustering strategies in their first versions. A more detailed description of the declustering strategies will be given in Section 1.3.3.

It took around 10 years to establish the bases of the partitioning problem. The works leading to its definition are organized in Figure 1.2 in which we show the structure of the works we previously described. In the following section we detail the dimensions used to describe the partitioning features used to organize the partitioning approaches described in Section 1.4.

1.3 Partitioning dimensions

We modeled the partitioning environment using a star-schema diagram shown in Figure 1.3. The dimensions with a double border in this Figure correspond to ones defining the core partitioning problem. We organize the partitioning approaches in the following sections by the means of the ten dimensions detailed below that are used as diversification criteria.

1.3.1 Type

This dimension describes the alternative ways of dividing a table into smaller ones. Two alternatives are clearly noticeable: dividing it *horizontally* or *vertically*. In brief, logical entities are represented as relations in the relational model. Each relation groups sets of tuples that are instances of a logical entity which are depicted by a set of attributes. For example, the entity Aircraft in Figure 1.1 is defined by the attributes: model, developer, length (in m), cost (im M$) and an identifier. The table is divided as shown in Figures 1.4a and 1.4b at a record level (horizontally) or at an attribute level (vertically) respectively.

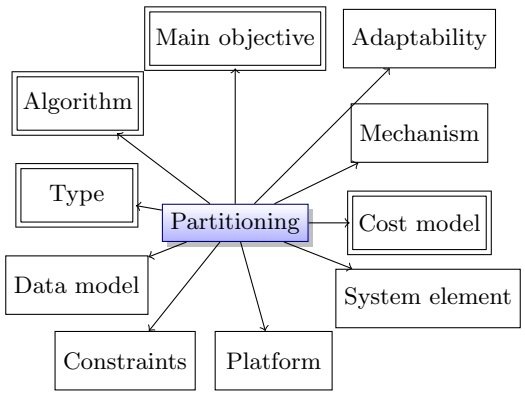


Figure 1.3: A star-schema partitioning environment

ID	Model	Dev	Length	Cost
1609	A320	Airbus	37.57	101
1752	A350-900	Airbus	64.75	317.4
1909	A340-600	Airbus	75.36	300.1

ID	Model	Dev	Length	Cost
1912	B737-800	Boeing	39.5	102.2
1212	B777-300	Boeing	73.86	361.5

(a) Horizontal partitions

ID	Model	Dev
1609	A320	Airbus
1909	A340-600	Airbus
1912	B737-800	Boeing
1212	B777-300	Boeing

ID	Length	Cost
1609	37.57	101
1909	75.36	300.1
1912	39.5	102.2
1212	73.86	361.5

(b) Vertical partitions

Figure 1.4: Partitioning types for table Airplane

The primary key is replicated on each partition in Figure 1.4b to enable the reconstruction of the original table if necessary. This condition is known as the *reconstruction* rule. More details about the correctness fragmentation rules are found in [ÖV11, RG00, SKS+97] . Horizontal partitioning has been explored more widely by commercial DBMSs and there are many versions of it. First, we distinguish horizontal partitions performed in a single table named *single horizontal partitioning*. This strategy is supported by most of commercial database systems in their latest versions. The single-level partitioned methodologies are:

- Range: as stated in [BBRW09], range partitioning is defined by a tuple (c, V), where c is a column type and V is an ordered sequence of values from the domain of c. A relation is then split according to a range of values for a given set of columns.
- Hash: this mode decomposes the data applying a hashing function that is provided by the system to the partitioning columns.
- List: this mode splits a table according to a list of discrete values for the partitioning key of a column.

Another horizontal type combining the above single-level partitioning modes is named *composite horizontal partitioning*. In this strategy, a table is partitioned by one data distribution method and then each partition is subdivided into sub-partitions applying a secondary distribution method. This strategy comprises the following strategies Range-Hash, Range-List, List-Hash, Range-Range, and all possible combination of the single level methods. For example, the Aircraft table of our example could be partitioned by Developer first (list partitioning) and then apply a range partitioned on the Length attribute.

The *referential horizontal partitioning* allows to partition a table by leveraging an existing parent-child relationship [BBRW09] established according to a foreign key between two relations named the member and owner. It was introduced by Ceri et al. at the beginning of the eighties [CNW83, CNP82] to optimize equi-join queries between the member and owner relations. The drawback of this partitioning mode is the fact that member tables may only be partitioned using a unique owner, even if its values reference to more than one owner tables. More details about this strategy are given in Section 1.4.1.2.

The list of partitioning types in this section is not exhaustive, however the missing strategies are variations of the types already presented. In the Table 1.2 we show the supported partitioning types in the most popular DBMSs. The hybrid partitioning column refers to a *virtual* column-based partitioning that allows the partitioning key to be an expression built on the row partitioning and it is efficient for storing columns individually in partitions.

DBMS	Version	Partitioning Strategy					
		Hash	Range	List	Composite	Derivate	Hybrid
Oracle	19c	✓	✓	✓	✓	✓	
MySQL	8	✓	✓	✓	✓		
Microsoft SQL-Server	2019		✓				
PostreSQL	9.4		✓	✓			
IBM DB2	11.5	✓	✓	✓			
Teradata	15.1	✓	✓			✓	✓

Table 1.2: Partitioning type by DBMS

A third partitioning type named *hybrid* is a combination of the previous partitioning techniques. The hybrid partitioning process can be represented as a tree structure, as seen in [ÖV11] in which vertical partitioning is followed by an horizontal fragmentation or vice-versa. This partitioning strategy creates much finer partitioning files and are a good fit for some applications needing the information at a lower granularity. A hybrid partitioning applied to the Aircraft table and its partition tree are shown in Figures 1.5a and 1.5b respectively. This partitioning strategy is supported by systems like Teradata as described in [ASAB16].

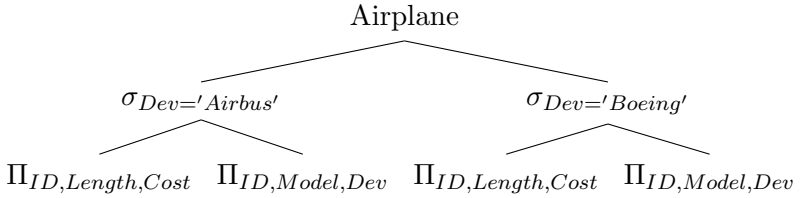
ID	Model	Dev
1609	A320	Airbus
1752	A350-900	Airbus
1909	A340-600	Airbus

ID	Length	Cost
1609	37.57	101
1752	64.75	317.4
1909	75.36	300.1

ID	Model	Dev
1912	B737-800	Boeing
1212	B777-300	Boeing

ID	Length	Cost
1912	39.5	102.2
1212	73.86	361.5

(a) Hybrid partitioning of Aircraft table



(b) Tree structure

Figure 1.5: Hybrid partitioning example

All of the approaches mentioned above are summarized in Figure 1.6.

1.3.2 Main objective

The problem of optimally partitioning data over a centralized, distributed or parallel database system could at a first glance seem like the same. Even though if the expected output is a set of horizontal or vertical partitions, the main objective to create each partition is quite different according to the system's requirements. For example, let us consider a partitioning strategy in a distributed database replicating most of the tables in all sites to ensure a good performance in terms of retrieval query time. This strategy would produce a very poor result if the partitioning objective was to partition the system to improve its performance in terms of storage efficiency. We categorize the partitioning objectives as:

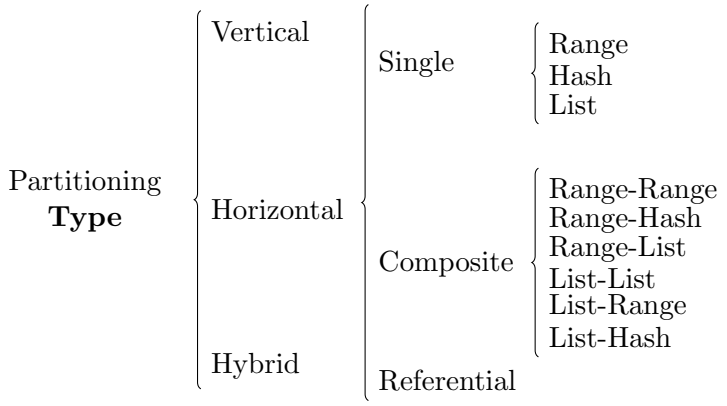


Figure 1.6: Type dimension schema

- *Response-time:* the main objective consists to reduce the query access costs. This is measured as an average response time to solve a workload (queries and frequencies). We distinguish approaches that seek to minimize time in the most frequent queries, and others that look for covering the greatest spectrum of queries with acceptable performances.

- *Concurrency:* this objective deals with executing the maximum number of transactions in the minimum time. It is measured in terms of the system's throughput.

- *Maintenance:* The storage and update criteria are included in this category. This objective seeks to maximize the efficiency in terms of the utilization of resources like the hard-drive, network or primary memory. It also considers for example, the cost reduction of idle machines in a distributed systems.

- *Combined:* This objective considers the minimization of custom cost models (e.g., weighted combinations) of the operational costs and measures of the query response time. They provide a much more balanced solution but to the cost of complexity.

1.3.3 Mechanism

This dimension refers to the means used by partitioning techniques to accomplish the previously mentioned objectives. It is comprised of two alternatives:

- *Clustering:* Partitioning in centralized database systems seeks to minimize the transfer costs between the primary and secondary memory retrieving the records or attributes that are pertinent to the user's request. In distributed database systems the main goal is to avoid costly network communications by localizing the executions at the distributed system nodes where the data resides. In both cases, the data is *clustered* to form groups of tuples or attributes that are usually retrieved together. For centralized systems the record or attribute clusters are created in such way that ideally only the clusters containing the data requested by the user are loaded to main memory avoiding unnecessary disk access. In distributed systems, the data clusters are created in such a way that distributed transactions are averted.

- *Declustering:* In parallel database systems there is no need to maximize local processing at each node. This strategy could, on the contrary, be self-defeating to the system's load balance making one node to perform all the work while all the others remain idle. Data partitioning in parallel systems come to a trade-off between maximizing the response time for each individual query but at the cost of execution skewness or maximizing the intra-

query[2] and inter-query[3] parallelism by means of partitioning. The partitioning strategy for these systems is called full partitioning or *declustering* as its main objective. Declustering strategies partition each relation across all the nodes of the system letting parallel database systems to exploit the I/O bandwidth of multiple disks by reading and writing them in parallel [DG92].

1.3.4 Algorithm

Finding the right set of partitions has been shown to be a very complex problem. The horizontal partitioning problem was proven to be NP hard [SW85, SW83], likewise the problem of finding vertical partitions applying an affinity graph [LOZ93]. Horizontal and vertical partitioning were mathematically formalized and posed as optimization problems whereof solution does not scale properly without the use of heuristics. We categorize the solution algorithms with the following classes:

- *Exact solutions*: integer programming formulations for the partitioning problem. Firstly introduced to create vertical partitions using attribute clusters measuring their pairwise affinity [ES76]. Then other mathematical techniques for record partitioning were proposed. All of these formulations served as basis for heuristics used to prune alternative solutions and boost the algorithms' scalability. The objective function and constraints for these algorithms belong to the categories detailed in Sections 1.3.5 and 1.3.6 respectively.
- *Heuristic solutions*: these approaches do not guarantee to be optimal but find an approximate solution in a reasonable time.
  - Cost based: These algorithms use heuristics to prune the solution space of alternatives to evaluate in a cost model as detailed in Section 1.3.5. Some alternatives use the query optimizer cost model to rank solutions to their performance.
  - Sophisticated: these methods adapt heuristics from other domains to solve the data partitioning problem. For instance, graph partitioning heuristics are used by some partitioning algorithms to create fragments of attributes or records.

1.3.5 Cost Model

To objectively asses if a partitioning strategy improves the performance of the database system a set of metrics are established to define the criteria that determine a *performant* system. After all, without a well-defined metric it is impossible to decide which partitioning alternative is better than another one. Also, cost models are essential when predicting the real execution costs of a query *a priori*, without actually evaluating it. Cost models provide a simplified vision of a system, and are related to the objectives described above in the main objective dimension. We categorize the cost models as:

- *Response-time:* these cost models use the query response time to measure the system's performance based on the query access costs. The query response time is usually measured as the weighted average response time to solve a workload. The response time can be directly measured or estimated as a linear combination of the processing, I/O and network costs as shown in the equation[RZML02]:

$$Cost(Q) = \alpha \cdot Cost_{CPU} + \beta \cdot Cost_{I/O} + \gamma \cdot Cost_{Network}$$

  This equation assumes that there is some overlap among the three components. To calculate the costs, the number of rows are estimated based on statistics. Among them we found

[2]Decomposing the query into smaller tasks that execute concurrently on multiple processors.
[3]Several queries execute concurrently on multiple processors to improve the overall throughput of the system.

the table cardinality, distinct values per column, number of pages in a table, and more data collected from histograms. For centralized systems the response time is dominated by the I/O costs that are estimated by the number of loaded disk blocks. The communication costs (i.e. $Cost_{Network}$) are the bottle neck in distributed database systems. For parallel database systems, the performance is dominated by the slowest machine of the cluster.

For a given workload, the costs are calculated for each of the queries and the global cost is given by:

$$Cost(w) = \sum_{q \in W} w(q) * Cost(q)$$

The $w(q)$ corresponds to the weight of each query in the workload which can be for example the frequency.

- *Throughput*: the traditional database problem is to maximize throughput subject to constraints on response time. As claimed by [GHK92], the problem could be also formulated as minimizing the response time subject to the constraints of the throughput. In this context, a sheer use of resources to reduce only the response time can lead to penalize the overall system's performance. To limit this phenomenon, some models have been proposed to limit the resources devoted to reduce the response time. Similarly, [RJ17] proposed the following relation to be minimized between the assigned and relative performance of the sites (B):

$$\frac{assignedLoad(B)}{load(B)}$$

- *Maintenance-based:* the storage and update costs are included in this category. These cost models will penalize replication and measure how efficient in terms of the utilization of resources like the hard-drive, network or primary memory.

- *Global-system:* these cost models are weighted combinations of the operational costs for secondary and primary storage and a measure of the query response time. They provide a much more balanced solution but to the cost of complexity. Let us consider for example the cost model in [PCZ12]:

$$Cost(\mathcal{D}, \mathcal{W}) = \frac{(\alpha \times CoordinationCost(\mathcal{D}, \mathcal{W})) + (\beta \times SkewFactor(\mathcal{D}, \mathcal{W}))}{\alpha + \beta}$$

In this case, the coordination cost is combined with a factor measuring the skewness of the partitioning strategy.

1.3.6 Constraints

The partitioning algorithms seek to optimize an objective function with respect to some variables in the presence of constraints. The following list summarizes the most common constraints:

- *Partition size:* this constraint is related to the limit in space, or processing resources.

- *Partition number*: this value is associated to the number of sites in the distributed or parallel architectures that will store the final data.

- *Redundancy*: whether the system allows to have multiple copies of the data to improve the query performance or ensure fault tolerance.

- *Workload*: this constraint refers to whether a predefined workload is known *a priori* to design the partitions.

1.3.7 Platform

This dimension considers the three types of database architectures:

- *Centralized*: only the interactions between the processor, primary and secondary memory are considered.
- *Distributed*: this dimensions include the Peer-to-Peer, Client/Server and Multi - Database Systems.
- *Parallel*: sub-divided in shared nothing, shared memory, shared disk and hybrid architectures.

1.3.8 System Element

In this dimension we consider whether the hardware element targeted by the partitioning approach. Among the components we can mention: primary & secondary memory, local area network, internet, cache and the processor.

1.3.9 Adaptability

This dimension classifies partitioning strategies into static and dynamic. The difference depends on whether the partitioning describes the procedure that is performed when declaring the database and it never changes, or if it includes describes strategies carried out after executing several workloads and the system is able to adapt its partitioning. The details of both strategies are given below.

- *Offline:* the database designer decides at the creation of the database how the data will be partitioned. This is achieved thanks to a requirement analysis phase in which the needs of the final users and the available resources are gathered. Most recently, automatic physical design advisors automatically select the proper partitioning configuration for a given workload. These advisors are supported by many commercial and academic database systems to help non-expert designers to select the most adequate partitions, indexes and materialized views. As mentioned in the introduction of this chapter, recreating partitions can be very expensive since the database needs to be disrupted and this is not always possible.
- *Online:* approaches following this strategy monitor and periodically adapt the database to fit the observed workload/incoming data. Ideally the system is able to adapt to changing workloads and data maintaining the database up-to-date at a reasonable performance.

1.3.10 Data model

In this section we classify the partitioning strategies according to the data model used to logically represent the data. We include in this category the relational model and the models directly derived from this. We include then:

- *Relational:* introduced in the early seventies by Edgar Codd in [Cod70] where the data are represented as tuples grouped in relations. Most of relational database systems use the SQL definition query language providing a declarative method for specifying data and declaring queries. The entity relationship (ER) data model introduced by Peter Chen in [Che76] is well suited to represent relational data. Its components are readily translated to relations. ER modelling is based on two concepts:
  - Entities: defined as tables that hold information and,

  - Relationships: which are associations or interactions between entities.

- *Object-oriented:* in this data model information is represented as objects, as it is done in object-oriented programming. This data model, introduced since the early eighties, is a hybrid of both the relational and object-oriented programming.

- *Deductive:* the deductive data model involves the application of formal logic to the problems of data definition, manipulation and integrity. The model provides the possibility to retrieve not only explicitly stored data but logically inferred data as well. The model was introduced in 1984 in the work of [Rei82] which gave the declarative semantics of deductive databases. The query and declaration language used for this model is based on the declarative programming language Datalog, a subset of the logic programming language Prolog. Among the query language we have the language ESQL introduced by Galdarin and Valduriez in [GV90].

- *Dimensional:* it is the model embraced in the data warehouse design. Introduced by Ralph Kimball in [KS95] as a solution to decision support systems and business intelligence in contrast the entity relationship data modeling already established in the nineties. In this work, the author claimed that Entity-Relation modelling works fine for Online Transaction Processing (OLTP) workloads but fails to solve analytical queries. As described in [GMR98], the dimensional data model consists of a set of fact schemes whose basic elements are facts, measures, dimensions and hierarchies. Facts model an event occurring in the company (e.g., a sale), measures are typically numerically valued attributes describing a fact and dimensions are hierarchically organized discrete values to facts.

1.4 Partitioning approaches

This section focuses on describing partitioning approaches organized with the partitioning dimensions described in Section 1.3. All of the dimensions are summarized in Table 1.3, in which dimensions are summarized in two groups: i)dimensions contextualizing the problem and ii) dimensions describing the solution. We start in Section 1.4.1 (whose schema is given in Figure 1.7) giving the fundamental algorithms chronologically by Type, each type is subdivided by platform and by partitioning mechanism. At each subsection we describe the exact and heuristic algorithms presented on each approach. Then, in Section 1.4.2 we present the approaches by data model and by constraints. Then in Section 1.4.3 we present the approaches by adaptability detailing some partitioning wizards. Finally in Section 1.4.4 we present a summary of the approaches by constraints.

Table 1.3: Partitioning approaches

	Dimension	
Problem	**Platform**	• Centralized • Distributed • Parallel
	Main objective	• Response-time • Concurrency • Maintenance • Combined
	System element	• Disk • Memory • Network
	Data model	• Relatinal • Object-oriented • Deductive • Dimensional
	Constraints	• Partition size • Partition number • Redundancy • Workload
Solution	**Type**	• Horizontal • Vertical • Hybrid
	Mechanism	• Clustering • Declustering
	Algorithm	• Exact • Heuristics
	Adaptability	• Offline • Online
	Cost model	• Response-time • Throughput • Maintenance-based • Global system

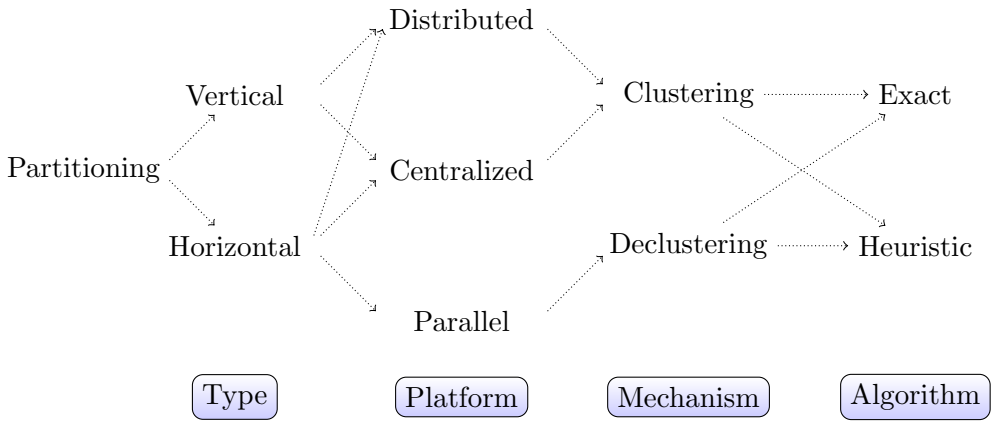


Figure 1.7: Organization of schema Section 1.4.1

1.4.1 Partitioning by type, platform and mechanism

The schema of this section is given in Figure 1.7. We start explaining the the vertical partitioning algorithms in distributed and centralized systems. Then we detail horizontal approaches in distributed and centralized architectures. Finally we discuss horizontal declustering strategies in parallel platforms. We highlight whether the approach is a heuristic or exact algorithm. In what follows we use the words fragment and partition indistinctly to denote each of the sub-relations obtained after a partitioning process.

1.4.1.1 Vertical

This partitioning strategy divides a relation on its attributes. Specifically, given a relation R vertical partitioning produces the subrelations $R_1 ... R_v$ such that each R_i contains a subset of R's attributes and its primary key attribute. Formally, consider $C(R) = \{c_1^*, \cdots, c_n\}$ as the set of R's attributes and c_1^* as the primary key attribute. For simplicity we consider only single attribute primary key, if there are more keys the c_1^* contains the set of primary key attributes. The set of sub-tables of R is $\{g_1, ..., g_k\}$ such that each g contains a disjoint subset of the non-primary key attributes as well as the primary key of R [ÖV11, ANY04] and $g_1 \bowtie g_2 \cdots \bowtie g_k = R$. The primary key in each g is used to be able to rebuild the original table. This partitioning strategy is more complex than horizontal fragmentation because of the bigger number of available choices [ÖV11]. It is useful when the application queries access only to a small subsets of columns in a table reducing the amount of data to be scanned to answer the query [ANY04].

Approaches to create vertical partitions exist since the seventies when Hoffer and Severance [HS75] measured the affinity between pairs of attributes to cluster them according to their pairwise affinity. They found the attribute's clusters using a *heuristic* based on the bond energy algorithm. Their motivation is to allow queries to deal with smaller relations, loading sub-relations with only the necessary attributes in main memory when a memory hierarchy is supported. This approach served later as basis to the algorithm detailed in [ÖV11] and the work of [MASB18]. Their work was modeled later as an integer programming problem in [Hof76]. The exact algorithm presented in this paper does not use any heuristic to be solved and consequently does not scale for bigger relations. Eisner and Severance proposed in [ES76] a heuristic technique using a network representation of the interactions of the users with the attributes of each record. The complexity of the problem was studied by Niamir in 1978 in [Nia78] which proved that the number of partitions equals to the $B(m)$ (m^{th} Bell number). For instance, a relation of 10 attributes, can be partitioned in more than 115,000 ways.

Niamir B. and Hammer M. proposed in [HN79] a system to automatically select attribute partitions using the hill-climbing heuristic to select the candidate partition whose evaluation cost is minimal. The candidates are obtained using a pairwise grouping-attribute regrouping

Attributes	ID	Model	Dev	Length	Cost
ID		15	20	30	40
Model	15		5	4	12
Dev	20	5		2	4
Length	30	4	2		22
Cost	40	12	4	22	

Figure 1.8: Affinity matrix of Airplane table

heuristic, named *grouping* (also known as bottom-up approaches), which starts assigning each attribute to one fragment and at each step joins some of the fragments to the initial one if the cost is minimal. Similar to this work, thirty years later the work named HillClimb [HP03] focused on data layouts proceeds as follows: it starts finding two partitions (initially single attributes) which, when merged, provide the best improvement in terms of expected query costs. The algorithm stops iterating when there is no improvement in expected query costs.

Similarly, [SW85] propose grouping heuristics for centralized and distributed databases respectively. A most recent modeling was proposed in [Amo10] in which the costs in terms of read, written and transferred bytes are estimated with a cost model given a database schema and a workload. The proposed cost model is extended to handle load balancing as well instead of just minimizing the sum of transfer/access costs. The author showed that finding the minimum cost model is an NP-hard problem and proposed two solution strategies: quadratic programming and simulated annealing. Their experiments performed using the H-Store database showed the feasibility and effectiveness of the cost model.

The results of the teams of Hoffer and Severance were extended by Navathe et al. in 1984 in [NCWD84]. In this work the authors propose a two-phase approach to obtain vertical partitions. The method performs clustering using an affinity matrix of attributes as the one shown in Figure 1.8. The numbers on the table represent the number of transactions in which both attributes appear together, for example the attributes ID and Model appear together in 15 transactions of an assumed workload. Then, with this result, the method uses a binary partitioning technique on the clustered attributes to determine the partitions. The work is used as basis in [CY87] where it is adapted to relational databases and later in [CPW89] the work was extended decomposing the design process into several sub-design problems. These heuristic algorithms, named *splitting* (also known as top-down) heuristics, start in general with a relation and partitions it at each step based on some objective function. Most recently, the grouping approach detailed in [BCN17] proposes an algorithm that independently of the workload and using only the database logical scheme derives functional dependencies between the attributes with TANE — a popular functional dependency extraction algorithm.

More sophisticated approaches to create vertical partitions using graphical algorithms are found in the works of Navathe et al. in [NR89]. In their work the authors mapped the affinity matrix to a graph with weighted edges that is partitioned using a technique expanding the graph's spanning tree. Similar techniques are applied in [LZ93]. Most recently, more heuristics have been considered to obtain vertical fragments. A genetic algorithm was used in [DAB06] to explore the set of candidate solutions. Besides association rules from data mining have been used to create clusters of partitions in [RG06] for instance. All of the approaches described above seek to minimize the response time by clustering the attributes according to a workload. A more recent approach [DPP+19] consider the feasibility of a general machine learning techniques, specifically deep reinforcement learning, to find the right vertical partitioning scheme matching a workload.

Some works present automatic database attribute clustering for single and distributed computers. For example, [LG12] proposed an algorithm called AutoClust to automatically vertically partition a database. Their algorithm is based on closed item sets mined from queries of the

workload and their attributes using rule mining. It can be divided in five steps: 1. Attribute usage matrix generation. 2. Attributes mining. 3. Primary key addition to the proposed partitions. 4. Generate execution tree, with a candidate attribute clustering in the leafs. 5. Submit solutions to the query optimizer to select the optimal vertical partitioning schema. Their algorithm is optimizer-integrated, meaning that it uses the query optimizer as a black box to perform the optimization using *what-if* calls.

A complete experimental survey of some of the work mentioned in this section was done in [JPPD13]. This work discusses six vertical partitioning algorithms under the same experimental setting. They concluded that the HillClimb algorithm presented in [HP03] is the best one in terms of the criteria defined in their paper. Some of the partitioning techniques in this study are presented in the sections 1.4.3 and 1.5 where we detail online partitioning algorithms and partitioning in recent large-scale platforms. The results of this section are summarized in Table 1.4.

Table 1.4: Vertical partitioning approaches

Approach	Algorithm	Strategy	Details
[Hof76]	Exact	Splitting	Integer programming formulation
[Nia78]		Grouping	Complexity study
[Amo10]		Grouping	Global cost model
[HS75]	Heuristic	Splitting	Bond energy algorithm (BEA)
[ES76]			Network representation
[NCWD84]			Affinity matrix
[NR89]			Graph partitioning
[LZ93]			Graph partitioning
[BCN17]			Functional dependency
[HN79]		Grouping	Hill-climbing
[HP03]			Hill-climbing
[DAB06]			Genetic algorithm
[RG06]			Association rule mining
[LG12]			Association rule mining
[DPP+19]			Deep reinforcement learning

1.4.1.2 Horizontal

Horizontal partitioning splits a relation along its tuples, it was explicitly defined shortly after vertical partitioning in 1978. However, contrarily to the works describing vertical partitions, the algorithms to partition the data horizontally were introduced a few years later.

Centralized and distributed platforms The efforts towards the characterization of horizontal fragments started with Ceri et al. in 1982 in [CNP82] with the introduction of the *min-term predicates*. A min-term predicate is the combination of simple predicates whose definition is given in Definition 1.2. A *minterm predicate* m_j is the conjunction of simple predicates in their natural or negated form (i.e. $\neg p_j$) such that m_j does not contain contradicting simple predicates.

Definition 1.2 *(Simple predicate) [ÖV11] Given a relation R and one of its attributes c_i, a simple predicate p_j is defined on R as $p_j : c_i \, \theta \, V$ where $\theta \in \{=, \neq, <, >, \leq, \geq, \}$ and V is any value chosen from the domain of c_i.*

The horizontal partitioning process of a relation R is described as the set of tuple groups $G = \{g_1, \cdots, g_k\}$ such that each group $g \in G$ is defined by a set $M = \{m_1, \cdots, m_k\}$ of min-term

predicates. Each g_i corresponds to a partition of the relation R if $\bigcup_{g_i \in G} g_i = R$ and $\bigcap_{g_i \in G} g_i = \emptyset$. The previous conditions are known as completeness and reconstruction fragmentation rules respectively. Ideally the partition groups are balanced and they correspond to the application groups queried by the application, however the selection of the optimal minterm predicates has been showed to be an NP hard [SW83] problem.

The min-term predicates algorithm was complemented by Özsu and Valduriez in [ÖV11] where the *COM_MIN* and *PHORIZONTAL* algorithms are described. In simple words, the main steps of the algorithm are:

1. Generation of min-term predicates (*COM_MIN* algorithm),
2. Simplification of min-terms eliminating redundant and useless terms (*PHORIZONTAL* algorithm), and
3. Generation of partitions (or fragments).

Following the example of Figure 1.4a, the predicates $\sigma_{Dev='Airbus'}$ and $\sigma_{Dev='Boeing'}$ are the min-predicates of the partitions shown in the Figure. To avoid redundancy, in this section we discuss only the generalities of horizontal partitioning approaches. Specific horizontal strategies are discussed in Section 1.4.2 to avoid redundancy.

Parallel platforms In parallel processing systems there is no need to localize the query execution in a single node. The objective of parallel systems is to use partitioning for parallelizing and load balancing query executions across the nodes of the cluster as a mean to improve the database performance when answering a workload. The problem was firstly formalized by Sacca D. and Wiederhold G. in [SW83]. In this work, the differences between parallel and distributed system were clearly stated. The authors proved that finding a feasible and optimal solution for the partitioning problem in a parallel environment is NP-hard. An important trade-off should be considered when only a few tuples satisfy the selection operator of a query. Ideally in this situation, the partitioning strategy should localize the tuples that satisfy the query across only a few processors minimizing the communication overhead associated with the synchronization and scheduling. Conversely, for complex queries a full declustering may seem like the most interesting strategy. This compromise makes partitioning in parallel systems much more complex [ÖV11]. We distinguish clearly two main horizontal partitioning strategies in parallel platforms: *single-attribute* and *multi-attribute* declustering.

The first high performance DBMS multiprocessor databases introduced in the eighties comprise systems like Gamma [DGG+86], Bubba [WB87] and Volcano [Gra94]. In such systems, relations are generally horizontally partitioned across multiple processors. When the entire relation is distributed among all the nodes of the cluster the mechanism is called declustering as we previously mentioned. The most widely single-attribute declustering strategies are the following [DG92]:

- *Round-robin*: this strategy sequentially sends the i^{th} tuple to the ($i \mod n$) partition.
- *Range*: distributes the tuples according to intervals of some attribute.
- *Hashing*: a hashing function specifying the placement of a tuple in a particular disk is applied to a tuple's attribute.

The decision on which hashing function to apply or which range to use in a specific key are left to the database designer's criteria. The strengths and drawbacks of each strategy are summarized in Table 1.5.

A major drawback of the hashing and range strategies supported by parallel systems is that neither can de-cluster a relation on more than one attribute. Several multi-attribute strategies

Table 1.5: Single-attribute horizontal declustering strategies

Strategy	**Strenghts**	**Drawbacks**
Round robin	Sequential access queries. Uniform data distribution.	Direct access to individual tuples.
Hash	Exact match queries	Range queries
Range	Exact match queries Range queries	Partition size disparity

were proposed to overcome this issue. We classify these approaches based on their objectives as presented in [GGGK03]. They are organized in Table 1.6, we give an overview of each of the approaches.

Table 1.6: Multi-attribute declustering

Objective	**Strategy name**	**Reference**
Localize execution to as few nodes as possible.	MAGIC BERD	[GD94] [BAC+90]
Reduce execution skewness.	Disk Modulo Fieldwise XOR Error Correcting Codes (ECC) Hilbert Curve Allocation	[DS82] [KP88] [FM91] [FB93]
Optimize processing of join operators.	DYOP Multi-attribute partitioning	[OO85] [HL90]

Let us start with the strategies striving to localize to a few nodes the execution of a query referencing a partitioning attribute. This group of strategies is appropriate for systems that suffer from the overhead to coordinate multi-site queries. We describe two of the main works: the Multi-Attribute GrId deClustering (MAGIC) [GD94] and Bubba's Extended Range Declustering (BERD) [BAC+90]. MAGIC is an extension of the Hybrid-Range partitioning strategy published in [GD90] that strikes a compromise between the sequential execution paradigm of range declustering and the intra-query parallelism achieved with hash and round-robin.

MAGIC builds a grid directory on a relation such that each entry in the two-dimensional grid represents a relation's fragment. To determine which attributes and ranges should be used to build the grid, MAGIC uses the frequencies of queries containing individual attributes and the average resource requirements (e.g., disk accesses, network). An example of such grid is shown in Figure 1.9 where each entry of the grid represents a fragment of the relation. Each partition would be assigned to a single processor if to enhance the system's throughput.

		Developer	
		Boeing	Airbus
Cost	0-150	1	2
	150-300	3	4
	300-450	5	6

Figure 1.9: MAGIC grid example

BERD [BAC+90] fully partitions a relation across the nodes of the cluster using the primary partitioning attribute value in the first place. Then for each of the secondary attributes an auxiliary relation is formed from the attribute's values with their respective identifiers and

location. The tuples on these relations are range partitioned over multiple locations and the partitions at each location are indexed in the form of a B-tree index. When a query is submitted to the system, if it involves the primary attribute it is directed to the relevant location. For any other attribute, the system uses first the auxiliary relation to determine the appropriate location of the tuples. A study comparing both BERD and MAGIC is presented in [GDQ92] in which the superiority of MAGIC over BEARD is manifested.

The second type of approaches is appropriate for workloads in which the overheads due to parallelism are shaded by the performance gain of splitting the work on different nodes. This approaches include: Disk Modulo (DM) [DS82], Fieldwise Xor (FX) [KP88], Error Correcting Codes (ECC) [FM91] and the Hilbert Curve Allocation Method (HCAM) [FB93]. They are clearly described in [MS98] using Cartesian product files, a multi-attribute file structure for partial match and best match queries very similar to the grid files presented before.

Let us start with the Disk Modulo method, which assigns a bucket $[i_1, i_2, ..., i_d]$ to a disk unit as:

$$i_1 + i_2 + ... + i_d \ mod \, M$$

where M is the number of available disks. This strategy is optimal for partial match range queries. For the matrix of Figure 1.9, before adding up the values of each i, the values (or range) are mapped to an integer.

The XOR declustering is optimal when the number of disks and the size of each field is a power of two. The assignment formula of this approach applies the bitwise XOR operator $\otimes$ to the binary values of the bucket coordinates $[i_1, i_2, ..., i_d]$ as:

$$i_1 \oplus i_2 \oplus ... \oplus i_d \ mod \, M$$

where M is the number of available disks.

The vector-based declustering generates a pair of integer vetors for a given number of disks and allignts the buckets in a Cartesian product file with the vectors. The Hilbert Curve Allocation (HCAM) uses the Hilbert space-filling curve to impose a linear ordering on the buckets in the Cartesian product files. Then it traverses the buckets in the order assigning each bucket to a disk unit in a round-robin way. As shown in [MS98], this strategy outpeforms the previous strategies for small range queries and large number of disks.

Finally we mention the group of strategies striving to optimize the processing of the join operator. The DYOP technique [OO85] partitions the data by repeatedly sub-dividing the tuple space of multiple attribute's domains. In order to execute a hash-join query efficiently, the size of each partition equals the aggregate memory of the processors in the system, the tuple's order is preserved as well. [HL90] builds a grid file on the attributes used to join the partitioned relation with others. For example, let us consider the relations $R(A, B, C)$, $S(B, D)$ and $T(C, E)$. Building a grid-file on the B and C attributes to partition R would minimize the number of tuples of R that are redistributed when it is joined with either S or T.

A simulation study of data placement issues in a shared-nothing system is presented in [MD97]. They performed experiments in a simulation written in CSIM/C++ that was configured to simulate a SN database of 128 nodes. The drawback of their model is that the cost to shuffle data on the network is disregarded. They performed experiments to establish the optimal degree of Declustering (number of partitions). They presented results of declustering, response time for joins, indexes, skewness and different CPU configurations for parallel and sequential systems. They concluded that full declustering is a viable strategy for placing relations in a SN parallel database system. The evaluation is done using the average response time as metric.

Referential horizontal partitioning This strategy uses the database schema, specifically the member-owner relation between tables to generate partitions. The member-owner relation is illustrated in Figure 1.10 which diagrams two tables in which a relation called the Member is linked to the information stored in another relation called the Owner. An integrity constraint

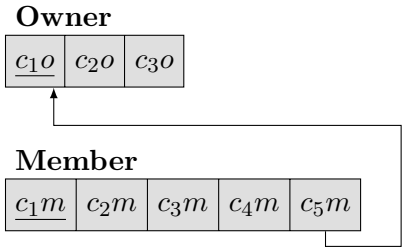


Figure 1.10: Owner and member relations

is defined between the two tables using the primary key of the owner relation and an attribute in the member relation (e.g c_1o and c_1m in Figure 1.10 respectively). It was first described by Ceri et al. in [CNW83]. In this partitioning method, the owner relation is initially partitioned and the partitions of the member relation are obtained with a semi-join operation.

Formally let us consider two linked relations *O, M* being the owner and member relations respectively. If O is partitioned and $\{o_1, \cdots, o_w\}$ are the partitions, the horizontal partitioning of M is defined as $M_j = M \ltimes O_j, 1 \leq j \leq w$. When the member table has foreign keys pointing to more than one relation, the choice of the attribute used to partition the member relation is done according to: i) the attribute that would maximize performance for most applications and ii) the attribute that minimizes the joins costs between both relations [ÖV11].

This partitioning strategy is supported by the commercial DBMS Oracle since its 11gR1 version. Its implementation is described in [ECS+08] which highlights the key concepts of referential partitioning, the implementation challenges and an experimental study showing the performance gain sand other benefits of this partitioning strategy. They also explained how referential partitioning could be used to simulate vertical partitioning by breaking a table into two or more smaller tables with less columns in a parent-child relationship.

1.4.1.3 Hybrid

As described previously, this partitioning also known as nested occurs when a table is vertically partitioned and it is then partitioned horizontally (or vice versa). As mentioned in [ÖV11], the number of levels of nesting can be large but it stops when each fragment consists only of a single tuple.

The only commercial DBMS supporting this partitioning strategy is Teradata. Their strategy detailed in [ASAB16] relies on the file system storage layer which handles either row, column or combined partitions in the same internal way. Contrarily to the horizontal or vertical systems which are row or column-based respectively, Teradata file system is *row-id* based. A row-id identifies a row or parts of it, and at runtime the data are searched using this id. In this way, the file system is agnostic to the partitioning scheme. When a row is inserted in a table, a row-id is generated based on a hash value of the primary index columns.

1.4.2 Partitioning by data model

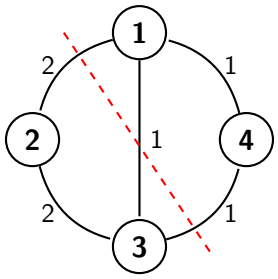
In this section we present the most relevant approaches by data model. We start chronologically with the relational data model, presenting the approaches applied to centralized and distributed platforms since the parallel approaches were described in the previous section. Then we move to object-oriented and deductive models and finally the dimensional data model.

1.4.2.1 Relational data model

Here we detail the most recent horizontal partitioning algorithms. We do not include vertical approaches in this section to avoid redundancy since they were fully discussed in Section 1.4.1.1.

TupleID	QueryID
1	1,2,3,6
2	1,4,6
3	1,2,4,5
4	3,5

(a) Workload as inverted table



(b) Schism graph

Figure 1.11: Schism data representation

For horizontal partitioning, we have already discussed the basic approaches in which the main concepts and algorithms are introduced. Let us start describing the approaches to deal with systems characterized by numerous short on-line transactions, in which partitioning seeks generally to increment the system's throughput. This is achieved mostly by reducing the number of distributed transactions. The approaches presented in this section are summarized in Table 1.7 in which we highlight the main strategy, its inputs and main objectives.

A very influent workload-aware partitioning approach named *Schism* was proposed by Curino et al. in [CZJM10]. Their system partitions the database using graph partitioning techniques and decision trees. They first considered an OLTP workload containing read and update queries and map the database to a graph. In their model, a node corresponds to a tuple and the edges connect tuples that are used within the same query. The nodes' weights correspond to the number of times the tuple is called in the workload or their size in bytes. This is exemplified in Figure 1.11. The interactions between the tuples 1 to 4 are shown in the inverted table showing the ID of the tuples and the queries they are part of (Figure 1.11a). The edges weights represent the number of times in the workload both nodes are connected. They expanded the graph to consider replication by adding n more copies to each node, where n is the number of edges connected to the node. When a partition is created, if two nodes of the same tuple are in the same partition, they are clearly not replicated. Contrarily, if two nodes of the same tuple are found in two different partitions then the tuples are replicated to ensure data locality. Using this graph, a partitioning heuristic (e.g., METIS) is applied and the data are divided in partitions. A decision tree is used to explain the obtained partitions, and to recreate a path leading to the final partitions using simple predicates. The red dotted line of Figure 1.11b represents the partitions after applying the graph partitioning heuristic. They evaluated their work using TPC-E and other databases taken from social networks.

In [PCZ12] Pavlo et al. presented an approach named Horticulture to automatically create horizontal partitions. Its main objective is to optimize throughput minimizing the number of distributed transactions and minimizing the access skew across servers. Their workload-aware partitioning algorithm is based on an adaptation of the large neighborhood search technique in which a cost model is used to estimate the coordination cost and load distribution for a simple workload. Their method starts with an initial best solution, obtained using measures from the workload and access patterns. It generates a set of neighbor solutions (called relaxing the initial solution) that are evaluated against the cost model (composed of the coordination and skewness cost). If they improve then the cost model then the solution is considered as best and it continues iterating.

Consens M. et al. presented divergent designs in [CILP12]. A divergent design installs a different physical configuration (e.g., indexes and materialized views) with each database replica, specializing replicas for different subsets of the workload. Queries are routed to the most suitable replica at runtime. They formalized the problem representing the cost of evaluating the workload and then using this cost formula to decide the best solution. In their experiments they used the DB2 Design Advisor to generate candidate solutions, and later each one was evaluated using

the total cost formula that considers the trade off between load balance and specialized replicas. They presented a pruning strategy that was tested with two TPC workloads.

The system HOPE presented in [CGZT14] applies an hyper-graph clustering algorithm to generate partitions of a database considering a transactional workload. In their work, they generated tuple groups using the *min-term* predicates. These groups of tuples are the nodes of the hyper-graph, weighted with two values: the size of the tuple group and the number of transactions in the workload accessing the group. Each transaction in the workload is mapped to a weighted hyper-edge of the graph. HOPE partitions the hyper-graph using the hMetis[4] package. The skewness of the partitions was regulated by a skewness factor.

Tran K. et al. proposed a Join Extension, Code-Based Approach in [TNST14]. Their method examines the workload and the database schema to derive a partitioning strategy. It is called Code-Based since it inspect the SQL code of stored procedure to define join paths used to partition. The algorithm leverages partitioning by foreign key relationships to automatically identify the best way to partition tables using the attributes. After it combines the solution for each transaction in the workload to find the best global partitioning solution.

Zamanian et al. presented in [ZBS15] a partitioning scheme called predicate-based reference partition (PREF) that allows co-partition sets of tables. Their method is a modified version of the referenced horizontal partition, but it ensures full data locality replicating tuples in different partitions. Besides, they presented two partitioning algorithms (one schema-driven and other workload-aware) that incorporate PREF to find automatically the best partitioning scheme for a database. In the schema-driven algorithm they modeled the given database schema as an undirected graph, having as edges the referential constraints of the schema. Based on this graph, they decide which relations will be initially partitioned (using a hashing function for instance), and the other relations that will be partitioned following the same schema. They maximize data locality, trying to reduce the data replication.

In [DLL+17], the authors present an approach to partition data and exploit the fact that in many recent database systems, data is replicated to increase the robustness and availability. Their method replicates data but each replica has a different partition structure. They start calculating the distance between queries of the workload based on the shared attributes. They later used a k-medoids to cluster the workload and generate a partitioning plan for each cluster and organize replicas with these plans. They assumed that the number of cluster is equal to the number of replicas.

Rabl and Jacobsen presented in [RJ17] an allocation strategy in shared-nothing data clusters. Their model allows replication to improve the performance of read only queries but it balances it to avoid low performances in update queries. Their model seeks to maximize the throughput while secondarily minimizing update and disk consumption overhead due to replication. For this, each query is run in a single node and replication of data between nodes is needed. Their model starts classifying the update or read queries of the workload according to the data they access. Each class is a different type of data fragment (e.g., horizontal or vertical fragment). Based on this classification, each group of queries is assigned to one or more data nodes. The allocation is calculated using heuristics to solve a linear programming problem balancing the load of query classes across the nodes and reducing the overall data replication.

Most recently, authors in [GLL+20] proposed a general strategy named AlCo (AlCo, Allocate fragments based on Cost) for allocating fragments in a distributed database. AlCo evaluates multiple candidate allocation plans based on a cost model, which is realized by a modified genetic algorithm employed by PostgreSQL. Their cost model synthetically considers various factors to enable a generalization ability. Also, to reduce the risks caused by randomization of the genetic algorithm, AlCo provides an upper bound computed through current heuristic methods to improve the algorithm's robustness.

[4] `http://glaros.dtc.umn.edu/gkhome/metis/hmetis/overview`

Table 1.7: Recent horizontal partitioning approaches

Approach		Main strategy	Inputs		Replication	Objective		
			Schema	**Workload**		**Max Locality**	**Min Redundancy**	**Min Workload skew**
Schism	[CZJM10]	• Graph partitioning.		✓	Partial	✓		
Horticulture	[PCZ12]	• Iterative neighbor search.	✓	✓	Full	✓	✓	✓
Divergent designs	[CILP12]	• Iterative refinement. • Use of query optimizer.		✓	Full	✓		
SWORD	[QKD13]	• Hypergraph partitioning. • Repartitioning technique.		✓	Partial	✓		
HOPE	[CGZT14]	• Hyper graph partitioning.		✓	Partial	✓		✓
JECB	[TNST14]	• Join-extension code-based. • Partial solutions combination.	✓		Partial	✓		
PREF	[ZBS15]	• Referential predicate-based. • Maximum spanning tree.	✓	✓	Partial	✓	✓	
Replica-aware	[DLL+17]	• K-medois to attributes. • Branch and bound.		✓	Partial	✓		
Query centric	[RJ17]	• Linear programming. • Greedy allocation strategy.		✓	Partial	✓	✓	
AlCo	[GLL+20]	• Custom cost model. • modified genetic algorithm.		✓	No	✓		

1.4.2.2 Object-oriented data model

The partitioning concept was smoothly adapted to this kind of databases in which an entity corresponds to a class instead of a relation. The partitioning strategies were explored firstly explored by Cook J. et al. in [CWZ94] investigating methods to improve the performance of the garbage collector, in which a subset of the entire database is collected independently from the rest.

A framework for class partitioning in OODB was proposed by Karlapalem K. and Li Q. in [KL95, KL00]. Their framework devises partitioning schemes based on different types of methods and their classification. The horizontal fragmentation process in OODB is defined as a process for reducing the number of disk entries to execute a query reducing the number of irrelevant objects accessed. Vertical fragmentation strives to reduce the irrelevant attributes accessed when querying a class.

Horizontal class partitioning: It was pioneered by Bellatreche et al. in [BKS97] and extended in [BKS00]. In these works the authors presented horizontal fragmentation based on a set of queries and develop strategies for primary horizontal partitioning. In [BKL98], the problem of derived class data partitioning has also been studied.

Vertical class partitioning: A cost-driven approach to study the effectiveness of data partitioning in OODBs in terms of reducing the number of disk accesses when executing a query was proposed by Fung C. et al. in [FKL03]. [CG97] proposes a unified view of the vertical partitioning problem and a set of transformation rules for various vertical partitioning methods.

1.4.2.3 Deductive data model

The partitioning problem in deductive database was covered by Mohania M. et al. in [MS94]. Deductive database addresses the design of distribution of both the database and the rules. In this work the authors considered the minimization of data communication cost as the primary rules from allocation. The problem is modeled as a directed acyclic graph, where nodes represent rules or relations and arcs represent dependencies or usage of relations by rules. A heuristic is then proposed based on successively combining adjacent nodes.

Another approach was presented by Neumann K. et al. in [NH94]. This paper presents the algorithms necessary to partition a deductive database represented as an Extended Predicate connection Graph (EPCG). The objectives that the system seeks to attain are the equalization of storage and processing costs, distributing the base relation nodes in an effective manner and preserving the locality.

1.4.2.4 Dimensional data model

As it was mentioned in Section 1.3.10 this data model is embraced by data-warehouses. This data model illustrated in Figure 1.12 in which a fact table (Maintenance table) is surrounded by the dimension tables (Aircraft, Developer, Date and Procedure) in a star-schema. Data warehouses had its boom in the nineties and it was at the end of this decade that the first partitioning strategy for parallel data-warehouses was proposed by Stöhr T. et al. in [SMR00]. In their work, the authors presented an allocation that considers relational data warehouses based on a star schema and utilizing bitmap index structures. Their experiments were developed in a "Shared Disk" architecture. They proposed a multi-dimensional hierarchical fragmentation of the fact table based on multiple dimension attributes. To fragment, they used a technique called "point fragmentations" that creates one horizontal fragment per value of the dimension. If it is done with for example two dimensions, having m and n as number of different values respectively, the maximum number of fragments is $n * m$. Bitmap indexes are fragmented using

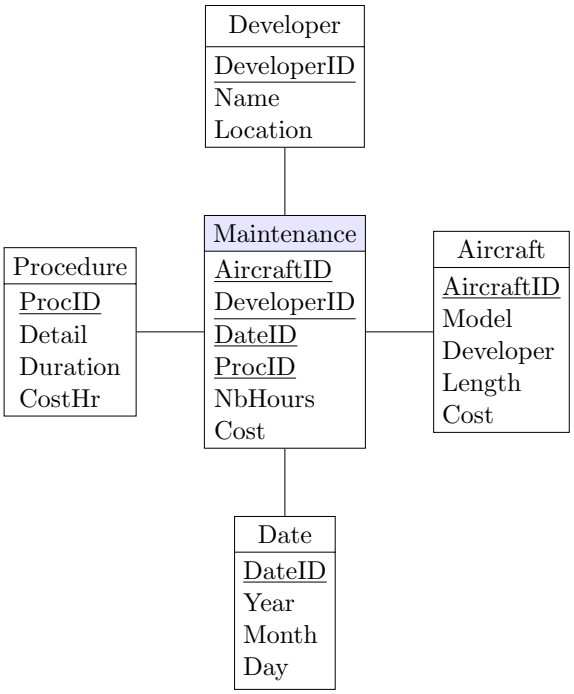


Figure 1.12: Star-schema example

the same logic as the fact table. A simple round robin allocation of fact fragments to the disks is used.

In parallel, [BKMS00] presented an algorithm for fragmenting the tables of a star schema (e.g., Figure 1.12). They mentioned that during the fragmentation process, the choice of the dimension tables used to fragment the fact table plays an important role in the overall performance. They develop a greedy algorithm to choose "best" dimension tables to perform the partitioning of the fact table.

Bellatreche et al. [BBM07] followed Stöhr's direction focusing on horizontal partitioning and bitmap join indexes in the context of centralized data warehouses. They proposed an algorithm to select simultaneously these structures in order to reduce the query processing cost. The algorithm uses the horizontal partitioning schema injected to a genetic algorithm to prune the search space when selecting the bitmap join indexes. Also, they proposed a greedy algorithm to select bitmap join indexes under a storage bound.

In [BBRW09] the referential horizontal partitioning was formalized for the dimensional data model. In this study, the authors gave a formalization of the referential fragmentation schema selection problem in the data warehouses studying its hardness to select an optimal solution. They proposed as well two heuristics to the selection problem: hill climbing and simulated annealing with several variants to select a near optimal partitioning schema.

The work of [LMV10] introduces a fined grained virtual partitioner which dynamically adapts partition sizes, without any knowledge about the database and the DBMS. Their proposal is to have an initial number of virtual partitions greater than the number of participating nodes. In this way each cluster node processes a set of small, light-weight sub-queries. In their algorithm, each node in the shared-nothing cluster contains a copy of the database. The idea is to partition each query (e.g., by range) on each node, such that each node is in charge of processing one specific query range. In this way, at query runtime many sub queries of the original query are produced per node. The goal is to achieve intra-query parallelism in OLAP query processing. To ensure good performance, there must be a clustered index associated to the partitioning attribute and also a uniform value distribution on the partitioning attribute.

Finally, a more recent work [NKH18] proposed a novel graph-based database partitioning

Table 1.8: Partitioning advisors

Advisor		Optimization Strategy				
		P[a]	**I**[b]	**M**[c]	**C**[d]	**S**[e]
DB2 Index Advisor	[RZML02]	✓	✓	✓	✓	
Parallel DB2 Index Advisor	[ZRL+04]	✓				
Microsoft SQL Server	[ANY04]	✓	✓	✓		
AutoPart	[PA04]	✓				
Parallel Microsoft SQL Server	[NB11]	✓	✓			
Parinda	[MDA+10]	✓	✓			
Oracle 11g		✓	✓	✓		✓
SOAP	[CZC15]	✓				
Deep Reinforcement Learning	[HBR19]	✓				

[a]Partitioning, [b]Index, [c]Materialized views, [d]Clustering, [e]Storage management

method called GPT that improves query performance with lower data redundancy. The authors claim that existing partitioning methods, specifically PREF [ZBS15], have a few major drawbacks such as a large amount of data redundancy and not supporting join processing without shuffle in many cases despite their large data redundancy. They pointed that this partitioning strategy is not optimal to treat analytic queries (OLAP) and that GPT is more suited for star and snowflake schemas in data-warehouses. They elucidate that the drawbacks arise from the tree-based partitioning schemes in PREF.

1.4.3 Partitioning by adaptability

We enumerate the offline and online partitioning strategies. We start with the offline partitioning strategies, specifically the available design advisors to initially partition a relational database. Partitioning wizards are a complement of the strategies previously presented. Then we depict dynamic partitioning, or on-line data reorganization as called in [SKS+97].

1.4.3.1 Partitioning advisors

Due to the complexity of the problem of selecting the optimal partitioning design for a database, a number of commercial and academic tools have been proposed to assist administrators and non-expert users in the task of choosing the most adequate partitioning schema for a workload. As it was mentioned in Section 1.2.1 data partitioning is not the only optimization technique to enhance the system's performance. Indexes and materialized views are suggested as well by some advisors considering the impact of the interaction and dependencies between these optimization strategies. The cited works are summarized in Table 1.8 in which we mention for each advisor the optimization strategies that it proposes. As it is shown in the table, the partitioning technique is the only optimization technique supported by all advisors.

The first automatic selection of table partitioning in parallel shared-nothing database systems was presented by Rao J. et al. in [RZML02]. They presented a solution that given a workload and their frequency of occurrence, determines the most optimal partitioning using an integrated solution with the query optimizer of the DB2 DBMS. In contrast to what was previously proposed, they built a partition advisor to automate the process of partition selection exploring the cost model of the query optimizer itself. To do this, they added two additional modes to the query optimizer: the recommend partition and evaluate partition modes. In the first mode, the optimizer will generate candidate partitions for each table involved in a query of the workload. The candidate set of partitions is not limited to this set, in recommend mode the optimizer generates candidate partitions considering replication, and combinations of individual

columns in the tables involved in a query. After, the query optimizer generates a plan based on these candidate partitions and it stores the plan in a table. The cost of each partition is evaluated by the optimizer using two different strategies: sampling and deriving. Both strategies adjust the value obtained when applying the cost function to the original statistics of the database. The evaluate partition mode performs a Rank-based enumeration of the whole set of partitioning configurations proposed in the candidate partitions table. Then using a customized cost function proposed the best partitioning model. They explored other ways of combining the models like genetic and other search algorithms but their results worked better in rank-based enumerations.

The DB2 advisor presented in [ZRL+04] was the first tool recommending indexes, materialized views and partitions by considering their interactions. The authors identified two types of dependencies between indexes, materialized query tables, data partitioning, and multi-dimensional clustering: *weak* and *strong*. A technique t_i is "strongly" dependent on technique t_j, if a change in selection of t_j often results in a change in that of t_i. Otherwise, we say t_i "weakly" depends on t_j. They showed that a weak dependency exists between data partitioning, indexes and multi-dimensional clustering. In contrast to the strong dependency between data partitioning and materialized query tables, knowing this interaction allows coupling data partitioning and other optimization techniques. For instance, in the case, when data partitioning cannot optimize all queries (due to the constraint representing the number of final fragments generated by a data partitioning algorithm), it will be then augmented by other techniques.

The work in Fractured Mirrors by Ramamurthy R. et al. detailed in [RDS02] is not specifically a partitioning advisors. However, their work was used later in the advisor proposed in [ANY04]. The Fractured Mirrors paper presents a partitioning model that takes advantage of redundant storage used in some database systems to provide tolerance to disk failures. They considered two disks in a mirror being logically identical but physically different. In particular, one copy of each table is stored using the Decomposition Storage Model (or vertical), and one is stored in a N-ary Storage Model (or horizontal). In their work they first revisited some performance problems associated with the vertical model. After they propose an indexing strategy to overcome the drawbacks presented in last section, specially the reconstruction algorithms. Their strategy based on a B-Tree showed a dramatic increase of performance over the naive implementation. that considered both horizontal and vertical partitions.

Agrawal S. et al. proposed a workload-aware solution to the problem of automating physical design in a single node in [ANY04]. Their solution takes both performance and manageability into account. In their paper they presented an integrated approach to automate physical design, considering horizontal and vertical partitioning, indexes and materialized views. They presented the complexity proof of the problem and the need to apply pruning techniques to reduce the large search space of solutions. They defined the physical design problem and presented the interactions arising from inclusion of horizontal and vertical partitioning. Their system architecture is composed of 4 main modules: 1) Column group restriction, 2) Candidate selection, 3) Merging, and 4) Enumeration. In the column group restriction step, the system selects column groups that are relevant for the workload (a column is relevant if it can be used to answer one or more queries in the workload). This step eliminates columns that will never help to optimize the workload. The candidate selection step uses a Greedy (m,k) algorithm to select for each query a relative optimal configuration. In the merging step, new physical structures are added to the set of candidates. This new physical structures are generated merging vertical partitions that are output of Candidate Selection. Next for each vertical partition, they merged all indexes and materialized views relevant for that vertical partition taking horizontal partitioning into account. The idea is to avoid over-specialized physical design structures and to build structures that benefit the entire workload. The Enumeration step takes as input the candidates and produces the final solution.

Nehme R. et Bruno N. proposed in [NB11] a partitioning advisor for parallel database systems

that recommends the best partitioning design for an expected workload. Their tool recommends which tables should be replicated, and which ones should be distributed according to specific columns. The developed techniques are deeply integrated with the parallel query optimizer, having a much more accurate recommendation in a shorter time. They assumed that when a table is partitioned it is hash-partitioned in a single column and also that the database statistics for cost estimation are always available. They claim that similar approaches do not use the query optimizers as they do, since previous works consider it as a black box and only use it to perform what if analysis (e.g., Rank-Based and Genetic algorithms to generate possible solutions to be evaluated by the optimizer). In their solution the query optimizer is an active part of the recommender, they use the physical and logical trees produced for each query to generate a data structure containing the candidate tables to be replicated and partitioned using the interesting columns (defined also on the paper).

Parinda [MDA+10] is a partitioning advisor for an open source DBMS (PostgreSQL). As the previously mentioned advisors, it uses the query optimizer to estimate the benefit of the design structure simulating the design features efficiently. Their algorithm uses the work of AutoPart [PA04], an algorithm that automatically partitions the database tables to optimize sequential access assuming prior knowledge of a representative workload. The simulations are achieved not achieving by writing the actual optimization features to disk but rather modifying the statistics used by the optimizer to solve a query. The advisor proposes both partitions and indexes to the database administrator.

From Oracle 11g a Partition Advisor[5] is part of the SQL Access Advisor recommending a partitioning strategy for a table based on a supplied workload of SQL statements.

Most recently, the work of [HBR19] introduced a learned partitioning advisor for analytical OLAP workloads based on Deep Reinforcement Learning (DRL). The main idea is that a DRL agent learns based on experience by monitoring the behavior of different workloads and partitioning schemes. Their experiments showed that their advisor is able to adapt to different deployments and it outperforms existing automatic partitioning approaches.

1.4.3.2 Online partitioning

The dynamic partitioning approaches treat the problem that the load on the DBMS does not remain static fluctuating constantly according to the change on the user's needs. The changes come either from the workload or from new data that is being added and that should be assigned to a specific partition. Next, we present partitioning strategies that allow the creation of elastic systems automatically re-organizing the data depending on the load or data changes. The approaches are summarized in Table 1.9.

Let us start with the work of [LMV10] previously detailed in Section 1.4.2.4. This work depicts a virtual partitioner to efficient OLAP query processing tuning the partition sizes without requiring any knowledge from the database or the DBMS.

A dynamic partitioning method was proposed by [JD11]. In their work the authors proposed AutoStore, a system that monitors the current workload and partitions automatically the data at checkpoints time intervals. They create the partitions based on the data obtained from an affinity matrix of attributes calculated for each query of the workload. They considered changes in the workload but no changes in the data. The cost model denote the execution cost of workload estimated by a cost-based optimizer.

In [LAP+13], the Liroz-Gistau et al. proposed *DynPart* (introduced in [LAP+12]) and *DynPartGroup*, two dynamic partition algorithms for continuously growing datasets. In their paper the authors formalized the partitioning problem, defining and imbalance factor to leave some flexibility to the size of each partition. Their partitioning method is workload-aware defining an operation that based on the relevant fragment for a query, calculates the efficiency of

[5] `https://docs.oracle.com/database/121/VLDBG/GUID-E864E9E2-0456-4FB5-860B-44444337D7D8.htm`

Table 1.9: Online partitioning approaches

Aproach		Details	Changing Workload	Changing Data
Virtual part.	[LMV10]	• Full database replication. • Specialized partitions for OLAP queries.	✓	
Autostore	[JD11]	• Attribute affinity matrix per query. • Partition at checkpoints time intervals.	✓	
DynPart	[LAP+13]	• Partition guided by eficiency measure: $\frac{\#relevant fragments}{\#fragments\_accessed}$		✓
SOAP	[CZC15]	• Cost-based repartition. • Trigger system's throughput threshold.	✓	
Cumulus	[FMS15]	• Time series tools to predict the workload. • Graph-based heuristic repartitining.	✓	
Clay		• Identify clumps of tuples to migrate.	✓	
GridFormation	[DPP+18]	• Reinforcement learning.	✓	

a partitioning for that respective query. Their defined efficiency is intuitively the ratio between the minimum number of relevant fragments of the query and the number of fragments that are actually accessed under the given partitioning. The second method groups the data to be added before hand. Contrarily to Clay, the workload remains static while the data is added constantly to the original dataset. They proposed a measure to calculate the affinity between the data added and the fragments.

The approach named SOAP, a system framework for scheduling online database repartitioning for OLTP workloads, was presented in [CZC15]. This system looks to minimize the time frame of executing the repartition operations while guaranteeing the performance of the current transactions running in the system. The repartitioner determines at which moment the database should be repartitioned extracting periodically the frequency of transactions and the partitions accessed by them. A cost-based repartition is triggered if the system's throughput estimated with the previous information is under a predefined threshold. A repartitioning task could be to move individual or groups of tuples (based on ranges for instance). To enable the database to continue running while a data are being moved, the system relies on replication. To this end, the system generates three types of repartitioning operators: replica creation, replica deletion and objects migration. Their prototype is built on top of PostgreSQL and conducted an experimental study on Amazon EC2 to validate SOAP's significant performance advantages.

Cumulus [FMS15], an adaptive data partitioning approach which is able to identify characteristic access patterns of transactions and use them to initially partition and dynamically repartition if these access patterns change. This strategy is specifically tailored for applications that need strong consistency guarantees with OLTP workloads. Partitions in this strategy are created with the objective to reduce distributed transactions and at the same time distribute the load across sites. First, Cumulus collects data from the current workload and uses the exponential moving average (time series prediction technique) to anticipate future access patterns. Then, a filter is applied to only preserve the most frequent access patterns that are organized in a weighted graph. The graph is partitioned using the METIS [KK98a] heuristic and finally a cost model is presented to decide if it is worth to repartition based on the gain between the cost to repartition and the reduced number of distributed transactions.

The online partitioning system called Clay is presented in [STE+16]. This system dynamically creates blocks of tuples to migrate among servers during repartitioning. It does not consider replication. The system identifies and groups in clumps the tuples that they called *hot tuples*.

These are the tuples that the DBMS wants to migrate to another partition. The clump is enlarged with the frequently accessed tuples of the hot tuple, and finally it moves the clump to other node.

Most recently [DPP+18] presented a preliminary work in an online self-managing partitioning and layout selection system based on reinforcement learning. Their goal isto build a general solution capable of leveraging experience and mimicking specialized methods. The paper presents the agents, environments and actions but since it is at an early stage, they do not offer a prototype or comprehensive evaluations.

1.4.4 Partitioning by constraints

In this section we summarize all of the previously described partitioning techniques by constraint. The complete list is organized in Table 1.10 in which the main dimensions of the partitioning problem are mentioned. The approaches whose constraint column is marked with a ✓mean that they consider the workload, the number of replicas (higher than 1) or the database schema as part of the problem's inputs. The schema constraint referes to the database schema, or the platform schema indicating for instance the number of replicas desired.

Table 1.10: Partitioning approaches

Type	Data model / Approach	Workload-aware	Replication	Schema	Response-time	Concurrency	Load balance	Combined	Clustering	Declustering	Exact	Heuristics	Online	Offline
		Constraints			**Main objecive**				**Mech.**		**Alg.**		**Adapt.**	
Vertical	[Hof76]	✓				✓			✓		✓	✓		✓
	[Nia78]	✓			✓						✓			
	[Nia78]	✓				✓			✓		✓			✓
	[HN79]	✓				✓			✓			✓		✓
	[NCWD84]	✓				✓			✓			✓		✓
	[CY87]	✓				✓			✓			✓		✓
	[CPW89]	✓				✓			✓			✓		✓
	[NR89]	✓				✓			✓			✓		✓
	[LZ93]	✓				✓			✓			✓		✓
	[RG06]	✓				✓			✓			✓		✓
	[DPP+19]	✓				✓			✓			✓		✓
	[Amo10]	✓				✓			✓		✓	✓		✓
	[BCN17]			✓		✓			✓			✓		✓
	Object oriented													
	[FKL03]	✓				✓			✓			✓		✓
	[CG97]	✓				✓			✓			✓		✓
Hybrid	[ASAB16]	✓				✓			✓			✓		✓
Advisors	[RZML02]	✓				✓			✓			✓		✓
	[ZRL+04]	✓				✓			✓			✓		✓
	[RDS02]	✓	✓		✓	✓			✓			✓		✓
	[ANY04]	✓	✓		✓	✓			✓			✓		✓
	[NB11]	✓	✓		✓	✓			✓			✓		✓
	[MDA+10]	✓				✓			✓			✓		✓
	[HBR19]	✓				✓			✓			✓		✓

Type	Data model / Approach	Workload-aware	Replication	Schema	Response-time	Concurrency	Load balance	Combined	Clustering	Declustering	Exact	Heuristics	Online	Offline
		Constraints			**Main objecive**				**Mech.**		**Alg.**		**Adapt.**	
Horizontal	*Relational*													
• Parallel	[SW83]	✓				✓	✓			✓	✓			✓
	[DG92]	✓				✓	✓			✓		✓		✓
	[GD94]	✓		✓		✓				✓		✓		✓
	[BAC+90]	✓				✓				✓		✓		✓
	[DS82]					✓	✓			✓		✓		✓
	[KP88]					✓	✓			✓		✓		✓
	[FM91]					✓	✓			✓		✓		✓
	[FB93]					✓	✓			✓		✓		✓
	[OO85]			✓	✓					✓		✓		✓
• Centr. & Distr.	[HL90]			✓	✓					✓		✓		✓
	[CNP82]	✓			✓	✓			✓			✓		✓
	[CNW83]				✓				✓			✓		✓
	[ECS+08]				✓				✓			✓		✓
	[CZJM10]	✓	✓			✓	✓		✓			✓		✓
	[PCZ12]	✓				✓	✓		✓			✓		✓
	[CILP12]	✓	✓			✓	✓		✓		✓	✓		✓
	[CGZT14]	✓				✓	✓		✓			✓		✓
	[TNST14]	✓				✓	✓		✓			✓		✓
	[ZBS15]	✓	✓	✓		✓	✓		✓			✓		✓
	[DLL+17]	✓	✓			✓	✓		✓			✓		✓
	[RJ17]	✓	✓			✓	✓		✓		✓	✓		✓
	[JD11]	✓				✓			✓			✓	✓	
	[CZC15]	✓	✓			✓			✓			✓	✓	
	[FMS15]	✓				✓			✓			✓	✓	
	[STE+16]	✓	✓			✓			✓			✓	✓	
	[DPP+18]	✓				✓			✓			✓	✓	
	[GLL+20]		✓			✓			✓			✓		✓
	Object oriented													
	[CWZ94]	✓				✓			✓			✓		✓
	[KL95]	✓				✓			✓			✓		✓
	[BKS00]	✓				✓			✓			✓		✓
	[BKL98]	✓				✓			✓			✓		✓
	Deductive													
	[MS94]	✓				✓			✓			✓		✓
	[NH94]	✓				✓			✓			✓		✓
	Dimensional													
	[SMR00]	✓			✓				✓			✓		✓
	[BKMS00]	✓			✓				✓			✓		✓
	[BBM07]	✓			✓				✓			✓		✓
	[BBRW09]	✓			✓				✓			✓		✓
	[LMV10]	✓	✓		✓				✓			✓	✓	
	[NKH18]	✓			✓				✓			✓		✓
	[LAP+13]	✓	✓		✓				✓			✓	✓	

1.5 Partitioning in large-scale platforms

Back in the early 2000s, the need to incorporate, store and query large amounts of data from the Web, with different sources, formats and at the lowest cost, contributed to the development of new kinds of data management tools. Systems based on the relational model, especially parallel databases, scale and can cope with vast volumes of data as demonstrated in [PPR+09]. However, the relational model is too rigid to adapt to the variability of sources and formats of current data. Also, since the data schema is rarely available beforehand, it is impossible to perform the conceptual design stage which is essential when building a relational database. Therefore, systems applying a *schema-later* strategy have been largely accepted. These systems store initially the data in its raw format and parse it at runtime. The pioneer presenting a novel scalable data management tool was Google, introducing the Google File System [GGL03] and the processing model MapReduce [DG04]. Their works served as base to Hadoop, one of the most popular large-scale data framework until today. In this section, we start giving a brief overview of Hadoop focusing on the partitioning strategy applied by this framework. Then, we explain and detail the most relevant systems built on top of Hadoop and describe their partitioning strategies.

1.5.1 Hadoop ecosystem

As defined in [Had], the Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across shared-nothing clusters of computers using simple programming models. Inspired by the Google File System and Map reduce papers, it is designed to scale up from single servers to thousands of machines not relying in main memory to execute computations. Additionally, the framework runs programs while ensuring fault-tolerance frequently encountered in distributed environments. Hadoop consists of two main modules: *i)* Storage and distribution layer (Hadoop Distributed File System - HDFS) and, *ii)* Data processing layer (Hadoop MapReduce and YARN). Part of its architecture is shown in Figure 1.13 in which the systems contoured in blue are the ones part of the Hadoop ecosystem.

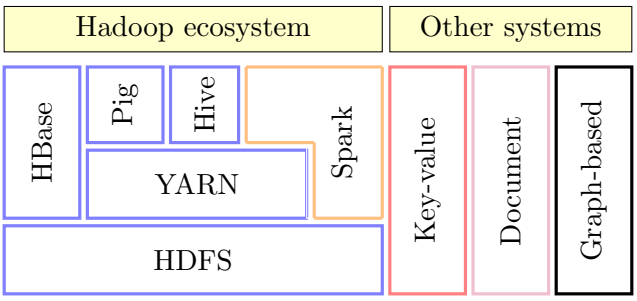


Figure 1.13: Large-scale systems

1.5.1.1 Data storage layer

The Hadoop Distributed File System (HDFS) is a distributed data storage system based on a master-slave architecture (the nodes are called Name and Data nodes for master and slaves respectively). HDFS is designed to store structured and unstructured data in a scalable, highly available, and fault tolerant way. Files are split into blocks of a configurable size (128MB by default) which are distributed and stored among the data nodes. To ensure fault tolerance and load balancing, data blocks of the same file are replicated and stored on different nodes. The name node (master) manages the file system and regulates the access from file blocks to locations using the file's metadata.

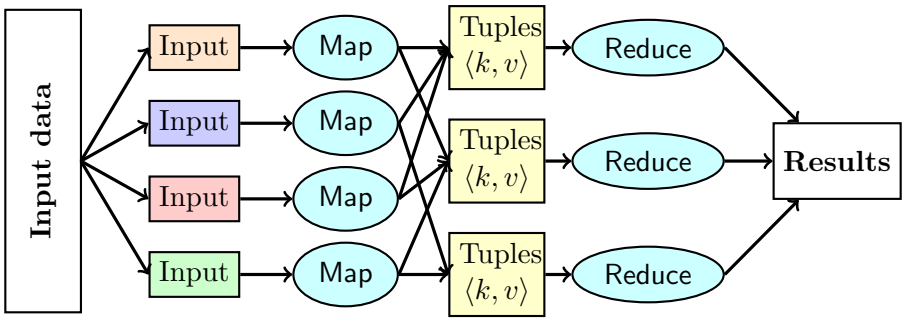


Figure 1.14: MapReduce schema

Data partitioning When a file is imported into HDFS, it is sequentially split into small blocks of a fixed storage size. Each block is replicated in a customisable fixed number of copies and a hash function is used to distribute the data to the nodes. This strategy is quite simple and allows to distribute the data controlling its balance. However it does not consider the schema of the data and it may produce some network overheads at retrieval.

1.5.1.2 Data processing layer

Since Hadoop version 2, the processing layer is divided in two main components: MapReduce and YARN. Both are designed to manage job scheduling, resources and the cluster. However in version 1 MapReduce provided both resource management and data processing. In Hadoop 2, the resources of the cluster are managed by YARN to eand the data processing models, including MapReduce, lean on YARN. In this way, YARN can be used with different processing models while taking advantage of HDFS functionalities.

MapReduce It is a programming framework following a master-slave architecture to process in a parallel, reliable and fault-tolerant manner data on large clusters. The programmer designs a job that must specify custom-built Map and Reduce functions. When the job is executed, the parallelization of resources and data are transparent to the user. First the Map function splits some input data into independent chunks that constitute key-values. These results are processed by the Mapper component in charge of distributing the chunks to the processing nodes, collecting the intermediate key-value pairs, sorting and grouping them by key before sending them as input to the Reducer. Next, the Reduce function aggregates the values associated to a key according to the predefined program. This procedure is illustrated in Figure 1.14. The first version of MapReduce was based on two components: i) Job Tracker (master) performing the resource managing and job scheduling, and ii) the Task Tracker (slave) supervising the execution of the Map and Reduce functions. In the latest version (called YARN), the Job Tracker was split into two separate daemons: i) A global Resource Manager and ii) a per-application Application Manager.

YARN Its name stands for Yet Another Resource Negotiator. The purpose of this resource management and task scheduling technology is to allocate the system's resources to the different applications running in a Hadoop cluster. Before adding YARN, Hadoop could only run MapReduce applications, adding YARN greatly increased the potential use of the framework. It separates the resource management and planning of the MapReduce data processing component enabling Hadoop to support more applications and different types of processing. In concrete, the JobTracker of Hadoop 1 is separated into two separate daemons: 1) ResourceManager that allocates and manages resources across the cluster, and 2) Application Master designed to schedule tasks, to match them with TaskTrackers and to monitor their progress.

1.5.1.3 Data querying layer

Several projects were built on top of the Hadoop architecture to avoid the implementation of MapReduce programs to query and analyze data stored in HDFS. If at the beginning MapReduce sought to be an alternative to relational databases, little by little notions of relational databases have been incorporated into the framework. For instance, SQL-like languages have been created to avoid the constraint of programming Map and Reduce functions. Also, other systems built on top of Hadoop incorporate some of the ACID (Atomicity, Consistency, Isolation, Durability) properties. In this section we give an overview of some of the most relevant projects, Apache Hive and Apache Pig, focusing on the partitioning strategy applied by these approaches.

Hive [6] it is a datawarehouse built on top of Hadoop that facilitates reading, writing, and managing large datasets using a SQL-like declarative language (HiveQL). At execution time, queries are translated as MapReduce jobs sent to the Job Tracker in a Hadoop cluster. Tables and databases are declared with a Data Definition Language (DDL) and then the data are loaded into the specified location. Tables in Hive are either internal or external, the first type *owns* its data, meaning that the data are deleted when the table is dropped. Hive supports only horizontal fragmentation of the data. To partition, it creates subdirectories in the HDFS reflecting the partitioning structure. For example, horizontally partitioning the Aircraft table of Figure 1.1 with two attributes will create the following directory in the HDFS:

```
...
.../aircraft/Dev=Airbus/Model=A320
.../aircraft/Dev=Airbus/Model=A350-900
.../aircraft/Dev=Airbus/Model=A340-600
...
.../aircraft/Dev=Boeing/Model=B737-800
.../aircraft/Dev=Boeing/Model=A777-300
```

Data in Hive are therefore only fragmented since allocation is controlled by the distributed file system. The HDFS splits and replicates the files on each directory into blocks that are distributed among the nodes of the cluster. Partitions are declared in a static (at the creation of the table) or dynamically (when inserting the data).

Pig! [7]: it is an alternative for the use of Hive, introducing an ETL-like language (named Pig Latin) rather than a query language like HiveQL. Pig Latin statements are transparently transformed into MapReduce jobs to perform the desired transformations to the data. Pig does not allow to create partitions on the working files. Still, it allows to read only the relevant sections for a given job if the data are stored in the HDFS partitioned with a tool like HCatalog[8] (a table and storage management layer built on top of Hive and incorporates Hive's DDL).

1.5.2 Apache Spark

Spark[9] is an unified analytics engine for large-scale data processing. It is based in a master-slave architecture with three main components, the driver program and cluster manager (master) and the worker nodes. Contrarily to Hadoop, Spark is based on in-memory strategies to improve performance. The main Spark's abstraction is a resilient distributed dataset (RDD), which is an immutable collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. RDDs are created gathering information from a distributed system, in general

[6] https://hive.apache.org/
[7] `https://pig.apache.org/`
[8] `https://cwiki.apache.org/confluence/display/Hive/HCatalog`
[9] `https://spark.apache.org/`

HDFS, hence data partitioning is not controlled by Spark directly. When the data is already in main memory, an RDD can be partitioned and repartitioned using different methods, such as hash, range, and custom partitioning. As Hadoop, the Spark project consists of a core version and several components were developed on top. The main components are:

Spark SQL This module is specialized for working with structured data. It allows to use custom SQL commands to query external data sets with complex analytics. Concretely, users can run queries over both imported data from external sources (e.g., Hive Tables) and data stored in existing RDDs.

MLlib It is Spark's machine learning (ML) library providing already implemented classification, regression, clustering and other machine learning algorithms. Quite similar to Mahout[10] for Hadoop.

GraphX Library for manipulating graphs and executing graph-parallel computations offering already implemented graph operators (e.g., subgraph and mapVertices and graph algorithms (e.g., PageRank).

Streaming This component provides automatic parallelization, as well as scalable and fault-tolerant streaming processing in Spark.

1.5.3 NoSQL stores

As mentioned in the introduction of this section, the increase of the data collected from the web, social networks, mobile and connected devices motivated the need for other data management systems. These systems sought to achieve horizontal scalability, high availability fault tolerance and database schema maintainability. However, achieving all of the above requirements through the traditional relational database systems is not a simple task and building newer systems with a different paradigm was necessary. The need was then to create systems that could cope to the evolution of schemas being able to handle at the lowest cost the massive data volumes. Some of the above requirements are mutually exclusive, and for some applications one may be ready to make trade-offs giving up one objective for another. This resulted in the emergence of NoSQL stores, some built on top of Apache Hadoop, which offer flexibility and relax some of the traditional RDBMs to boost the performance another(s). The following classification is taken from the work of [DCL18].

1.5.3.1 Key-value

In this systems the data are represented as *(key,value)* pairs stored in highly efficient lookup data structures. They guarantee very fast lookups and are suitable for applications using as single key to access the data. Data partitioning in these systems is performed in general by hashing the key.

Redis [11] It is an in-memory data structure store, used as a database, cache and message broker. It supports data structures like strings, hashes, lists, sets, sorted sets with range queries, bitmaps, etc. The database keeps the data in main memory but allows to persists it if some conditions are satisfied.

[10] https://mahout.apache.org/
[11] https://redis.io/

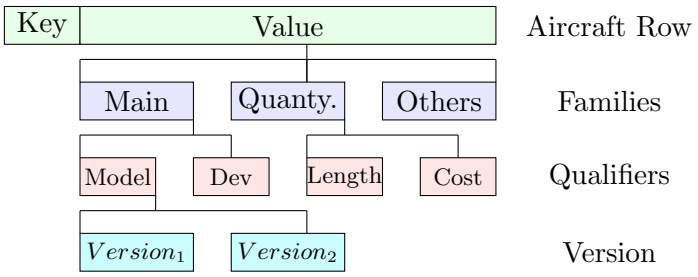


Figure 1.15: Internal structure of HBase row example

1.5.3.2 Column-family

In this type of systems the data are represented as rows and a fixed number of column families. A column-family is a number of attributes (columns) logically related to each other and that are mostly accessed together. They are inspired from another Google project, the Bigtable system [CDG+08]. Data partitioning in these systems is done either horizontally and vertically (by column families). We describe one of the most popular column-stores: HBase.

HBase [12]: it is a distributed and scalable data store in which data are stored with key-value pair structures. The key is the row identifier and the value contains the row's attributes. The row's attributes are further organized into column families, which are physically stored together in a single file in the distributed file system. Also, HBase keeps a customizable number of versions of the attribute's values of each row. HBase uses HDFS as storage layer but other distributed file systems could be used. Data are fragmented applying a hybrid approach. Firstly, horizontal fragments (called regions) are performed on the keys. These fragments are used as balancing unit by HBase. Then, the row's attributes are vertically partitioned in terms of column families, storing each family in a different file. An example of the data organization of HBase is illustrated in Figure 1.15.

1.5.3.3 Document stores

These systems are more complex key-value tores in which the value is represented in a structure named a *document* encoded instandard semi-structured formats such as XML or JSON.

MongoDB [13] This system is the most popular document store which provides a powerful query language that allows to filter and sort by any document's field. As key-value stores it partitions the data based on their key. It uses the key associated to the collection to partition the data into chunks. A chunk consists of a subset of sharded data and each chunk has a inclusive lower and exclusive upper range based on the key.

1.5.3.4 Graph stores

These systems store datasets represented as graphs in an efficient manner providing effective operations for their querying and analysis. These stores represent a graph as a set of vertices representing entities and edges representing relationships between them. They are divided in native and non-native graph stores according to whether they are built on top of a non-graph data store (e.g., relational database, document store, key-value store). The most popular native system is Neo4j[Web12] which is a highly scalable native graph database, purpose-built to leverage not only data but also data relationships. More details about graph-storage will be given in Chapter 2.

[12] `https://hbase.apache.org/`
[13] https://www.mongodb.com/

1.5.4 Hybrid architectures

In a traditional relational database management system, a triple is retrieved sending a query request to the database. The query is forwarded to the file system (controlled by the database) which retrieves the corresponding disk blocks and sends them back to the system. The DBMS has a much more direct control over the location and the way the data are partitioned among the disk(s). In the previous approaches, the file system (HDFS) and the query engine (e.g., MapReduce, HBase, Spark) do not form a single unit, and even if some approaches fragment the data on distinct files, the data allocation and replication tasks are left to the distributed file system. In this section we overview some hybrid architectures that place the data intentionally in specific nodes of the distributed file system or another type of storage layer.

HadoopDB [ABA+09] is another hybrid architecture that uses an existing RDBMS to store the data in each worker, instead of the HDFS which is used by Hive. This allows to exploit partitioning, indexing, buffering and compression capabilities offered by DBMSs. Partitioning the data is done using the horizontal approaches presented in last section.

The approach in [ETÖ+11] named CoHadoop, locates data intentionally on specific data nodes. This is achieved extending the HDFS with a file-level property (locator) in such a way that all files with the same locator are placed on the same set of Data Nodes. The NameNode is augmented with an index (i.e. location table) specifying where the data must be placed. Partitioning is then decided by the user (using the workload for example). Their approach is similar to [DQJ+10], Hadoop++ supporting as well data copartition.

The Trojan Data Layouts were proposed in [JQD11]. This approach internally organizes data blocks into attribute groups according to the workload in order to improve data access times. Their system fully preserves the fault tolerance properties of MapReduce, since they kept the blocks of replicated data but only adapted the way each block organizes data internally. Their algorithm applies first a column grouping method to cluster queries in a workload according to access patterns. Then they map each resulting query group to one data block replica as to compute the Trojan layout for such replica.

Other approaches have proposed improvements to the Hadoop framework based on the physical data layouts in the HDFS. Data in HDFS are usually organized using a row-oriented layout since a pure column store has severe drawbacks as the data for different columns may reside on different nodes leading to high network costs [DQ12]. Still, some systems proposed to store the data as columns. Llama [LAC+11] store the data as columnar files. Its performance is comparable to databases, in which the performance if better in queries involving small number of attributes. This system was outperformed by [Che10], which also employs a columnar format to store the data but implements a partition the partitions attributes across (PAX [ADHS01]) layout, in which all data belong to a record are stored in the same block.

In [RHAF15], Romero et al. proposed a system to solve analytic queries with a cost-based approach. Their system is implemented in a three level architecture running HDFS at the lowest level, HBase as the storage manager and MapReduce as the query execution engine. In their approach, the records were indexed and partitioned in HBase.

1.6 Conclusion

With the data deluge and the explosion of deployment architectures, the data partitioning has become the sole technique that database actors (designers, administrators, architects, analysts, students, researchers) have to know in detail. At the same time, data partitioning has been studied from the arrival of databases evolving with its generations. In this chapter we sought to give a historic overview of the data partitioning problem, its history, evolution, constraints and how it behaves after the arrival of each new data representation and deployment architecture. To do so, we proposed a data partitioning foundation that includes: implicit and explicit definitions and types, as well as data partitioning dimensions comprising: (1) partitioning type,

(2) algorithm, (3) main objective, (4) adaptability, (5) mechanism, (6) cost model, (7) system element, (8) platform, (9) constraints and (10) data model. These dimensions offer a graphical representation (a star-schema like) of data partitioning. For each dimension, we described its characteristics and depicted its elements and labels.

We have focused on presenting the state-of-the-art of data partitioning in relational databases and other close models in centralized, distributed and parallel architectures. We cited the works that led to an explicit definition of the partitioning problem and showed the gradual development of the main data partitioning forms. Subsequently, we exposed the algorithms, strategies and heuristics introduced to partition the database according to a set of formerly defined dimensions. We offer a comprehensive review on data partitioning and can be used as a torchlight for readers and actors to work in this exciting and durable problem. Finally, we show how the problem has evolved in the current cloud platforms and how it adapted to cope with large-scale data. In the following chapter we show how the problem has been treated specifically for treating graphs.

Chapter 2

Graph Data : Representation and Processing

Contents

Summary In the previous chapter we surveyed the partitioning problem in relational databases. Our goal is to provide a partitioning environment to managers of Knowledge Graphs. To do this, we cannot disregard how the Knowledge Graphs have been treated in the literature. In this chapter, we start giving a general overview of the graph data model. In Section 2.2, we detail the logical and storage structures for graphs , describing the query processing strategies and languages (Sect. 2.2.1 - Sect. 2.2.4), and reporting some of the most relevant currently available systems. We dedicate also Section 2.7 to describe the graph partitioning algorithms used by some parallel systems. Next, in Section 2.3, we focus on the Resource Description Framework (RDF). We start by giving some background concepts (Sect. 2.3.1), and similarly to what we did with the pure graph systems, we describe RDF storage (Sect. 2.3.2), processing (Sect. 2.3.3) and its partitioning strategies (Sect. 2.3.4). We conclude summarizing the presented works and comparing the strengths of both worlds (relational and graphs).

2.1 Introduction

In the previous chapter we detailed the partitioning problem in relational databases. We showed its evolution and how it was adapted to the object oriented, deductive and multi-dimensional data models. We explained the development of large-scale platforms and how they adapted to the recent need to efficiently deal with big data volumes. However, the data models discussed derive directly from the relational design so widely studied. In this chapter we detail a data model that allows describing more complex relationships. Specifically, graph-oriented models representing the data as graphs whose manipulation is expressed as graph-oriented operations. They have gain momentum in the past years with the development of the Semantic Web, Social Networks, and with the increase in the ability to capture data from different disciplines like Biological and Transportation networks. In these applications, the interconnectivity of the data is a key feature and it is as important as the data itself. Modeling the data as a graph allows a more natural way of handling these applications. In graph-based systems, queries refer directly to the original graph structure and graph operations like finding the shortest path can be directly expressed with specific query languages. In this chapter we start giving more insights about the graph data model in general (Section 2.2), discussing their storage, processing and partitioning. Then, in Section 2.3, we focus on the RDF model introduced to represent the data in the Semantic Web giving some main definitions and describing the most relevant storage and processing approaches.

2.2 Graph database models

The research around graph database models is not new, actually it had a peak in the late eighties/early nineties but it was eclipsed by the development of tree-based structures like XML modeling in the first generation websites. The use of graphs in computer science dates back to the beginning of relational databases in the mid-1970s in which graph models were proposed to expand the information stored in databases with semantic networks (firstly defined in [Sim72]). Also, graphs were used by the legacy data models (hierarchical and network models) to depict the navigation of its records. The historically influential network data model [TF76] represents data as physical records that are navigated using a graph-based structure (a network). This model, also known as CODASYL, browses through the record's parents (owners) and children (members) remaining tied to the physical implementation of the data. In what follows we give a brief overview of the former graph database approaches. The list of approaches is not exhaustive, we mention only a few of them to give a general idea of the concept evolution. A complete survey is found in [AG08]. In chronological order and according to the system in which they collaborate, the first graph database models appeared in the following systems:

Semantic networks The work of [RM75] proposed the use of a semantic network extending the original Codd's model to represent knowledge about the database. The semantic network defines the meaning of the data in a labeled directed graph (described in Section 2.2.1) where both nodes and edges are labeled. This network contributes to the generation of the relational schema and to the definition of semantic operators. Similarly, a semantic network was used by [Shi81] to describe the database schema in their functional data model and language named DAPLEX. DAPLEX's basic components are entities and functions, in the graph structure the nodes correspond to entity types and the arrows depict functions. The work of [KV84] used a directed graph to represent the database structure before defining a logic and an algebraic query language generalizing the relational, hierarchical and network data models. In this line, the system G-Base [Kun87] represented the database in terms of knowledge structures using a labeled directed graph introducing as well a data manipulation language for the stored graph.

Object-oriented As it was mentioned in the previous chapter, object-oriented databases (OODBs) had its boom in the late eighties. Graph-oriented models for OODBs were proposed in the systems O_2 and GOOD described in [LRV88] and [GPdBG94] respectively. A directed graph is used in both systems to represent the objects and relationships between them. GOOD (graph-oriented object database) is a theoretical support for systems in which the data representation and its manipulation are graph-based. GOOD introduces basic operations and a language based on pattern matching to manipulate graph-shaped object bases. This work served as basis to several research projects, for example G-Log [PPT95] introducing a declarative query language for graphs.

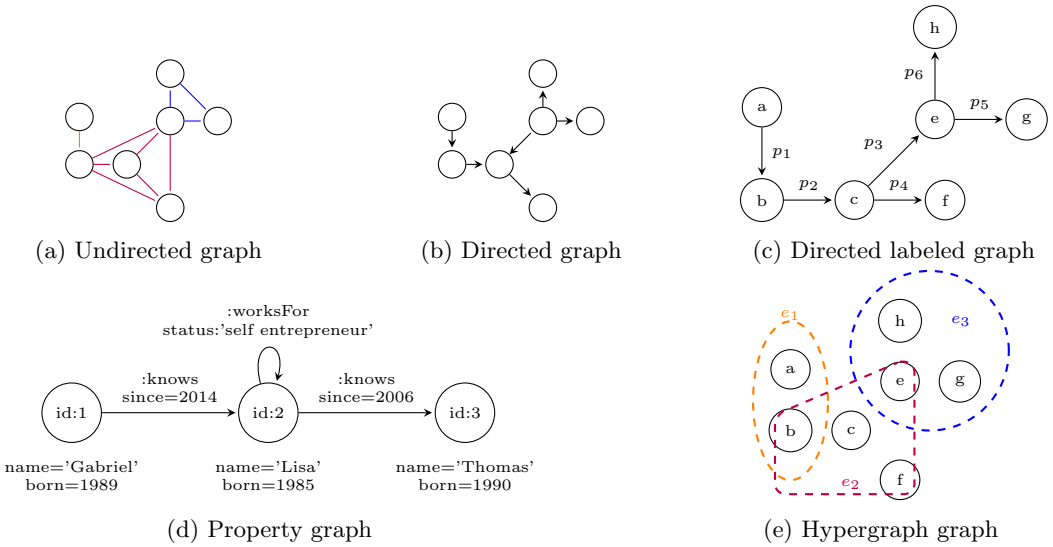
Former graph models The GraphDB system [Güt94] proposed a data model and a query language for graphs in a standard database environment. Their goal is to smoothly integrate graph notions into the popular database systems of the time (object-oriented specifically). To achieve this, they defined a graph database and a query language on top of classes and hierarchies derived from OODBs. In parallel, the work of [GPST94] presented at the same conference, introduced Graph Views to define and manipulate graphs independently of the system used to persist the data (e.g., relational, object-oriented and file systems). To manipulate the graphs, they proposed derivation operators defining new graph views upon existing ones. Other works used hyper-graphs to represent the data in a much more general model, for example Hy^+ [CM93] provides a user-interface to formulate queries to a set of graph patterns.

The beginning of the 1990s marked the emergence of the best known aspect of the Internet today: the Web as a set of pages in HTML format incorporating text, links and images, addressable via a URL and accessible using the HTTP protocol. The Internet boom undoubtedly diverted the efforts devoted to graph-based models to semi-structured models originally used to publish data on the Web. The success and exponential growth of Web content forced the creation of standards that allowed to order and exploit the information available on the Internet. The World Wide Web Consortium (W3C) founded in 1994 is in charge of developing and maintaining these standards within which we find HTML, XML, RDF and SPARQL. The Web is naturally modeled as a graph, Web pages are connected nodes interconnected via hyperlinks. The famous PageRank search algorithm used by Google apply this modeling to measure the importance of webpages in its search engine. Web data integration standards (e.g., RDF) also use a graph to model the interconnection of resources on the Web. More recently, the massive use of social networks and the possibility to recover data from many domains pushed the return and development of graph models. In the following section, we explain the logical graph structures used to represent graphs.

2.2.1 Logical graph data structures

As it was previously mentioned, graph data models represent the data as graphs providing efficient traversal operations to query and analyze them. Graphs are composed of entities representing something that exists as a single or composed unit, and relations establishing connections between two or more entities. In this section we describe the data structures used to model entities and relations in the graph data model. These structures are not mutually exclusive, some structures are a combination of some of them. A few of the structures are illustrated in Figure 2.1.

- **Directed/Undirected graphs** In an undirected graph like the one shown in Figure 2.1a, all relationships are symmetric. Contrarily, the edges on directed graphs illustrated in Figure 2.1b have a single source and a destination vertex. The number of incoming/outgoing edges define the in/out degree of each node respectively.



(a) Undirected graph (b) Directed graph (c) Directed labeled graph

(d) Property graph (e) Hypergraph graph

Figure 2.1: Graph data structures

- **Labeled graphs** In this type of structure vertices and/or edges are tagged with labels representing their respective roles, types or some metadata. A particular type of labeled graphs, named edge-labeled graphs, in which labeled directed edges indicate the different type of relationships is shown in Figure 2.1c. This amalgam is the one embraced by the Resource Description Framework described in Section 2.3.
- **Multigraph** These graphs allow multiple edges between the same vertices and self-loops.
- **Attributed graphs** In this structure, the nodes and edges have properties expressed as variable list of key-value pairs. An attributed, directed, labeled multigraph named *property graph* is given in Figure 2.1d. The property graph model is implemented by many graph-based systems and is used as reference model for several research. An in-depth description of property graphs is given in [BFVY18].
- **Hypergraphs** This model is said to be a generalization of a graph in which an edge can join any number of vertices instead of only one. The set of nodes connected with the same edge is called an hyperedge, when the hyperedge is directed, the set of nodes is ordered. A simple hypergraph with 3 hyperedges is shown in Figure 2.1e.

2.2.2 Data storage

Whether the system was built specifically to treat graphs or if it was built on top of another platform, each of the structures described previously is stored according to the strategies described in this section.

- **Adjacency matrix** This matrix is a 2-dimensional array whose size equals the $n \times n$ where n is the number of vertices in the graph. The matrix stores a 1-bit in cell $m_{i,j}$ if it exists a relationship between a vertex i to vertex j and a 0-bit otherwise. It is the representation with the simplest implementation, however it suffers from space overheads even for sparse graphs. An example of this structure is given in Figure 2.2a. The figure shows a variation of the adjacency matrix in which the edge's ids between two nodes are indicated directly in the cell.
- **Adjacency list** This representation stores the graph as a set of lists. Each row of the array corresponds to a vertex paired with a list of its neighboring nodes. Usually, it stores a node with its out-neighbors, however some systems enable replication storing

both (in/out)-neighbors for a node. Efficient index structures over vertices and edges are created so that random access to single elements is fast. This storage scheme is ideal for graph systems built on top of key-value stores and systems prioritizing online transaction processing workloads (OLTP) [DCL18]. An outward adjacency list is illustrated in Figure 2.2b for the directed graph.

- **Edge list** Contrarily to the adjacency list, this storage structure represents a graph as a list of edges indicating for each edge its corresponding incident vertices. This structure in which all the information associated to an edge is clustered is useful to the representation of hypergraphs in which an edge is associated with a set of edges. Some recent popular graph stores, like Neo4j [Web12] use this representation. This representation is also preferred by some RDF processing systems. An example of such structure is given in Figure 2.2c.

- **Compressed sparse row (CSR)** It is one of the most widely used storing structures for its efficient lookup structure. To store a graph it uses two arrays of integers. More arrays could be added to encode more information about a node and edges (e.g weights), this is the representation used by the Metis[KK98a] graph partitioner discussed in Section 2.2.5. The first array (named array of edges) stores the adjacency lists of all vertices together storing all edges continuously. The size of the array of edges is therefore the total number of edges in the graph. The second array, named vertex array, is an index to the previous array mapping a vertex ID into the ID of the first outgoing edge. Let us consider the example in Figure 2.2d, in which the nodes and edges labels are encoded before appending them to the edge and vertices array. In this example, the CSR represents the directed labeled graph of Figure 2.1c.

- **Complex indexes**: This category comprises more sophisticated index structure than a regular adjacency list. We can mention for instance the compressed bitmap indexes used in the system DEX [MMG$^+$07] used to efficiently combine multiple adjacency lists with bit operations.

2.2.3 Query and manipulation languages

Among graph data models there are several works on query languages. These works present collections of operators and inference rules to manipulate and query the graph data structure. Considering the vast number of approaches available and the scope of this thesis, in this section we describe only the most recent and popular graph query languages. An in-depth analysis of query languages for graph databases is given in [AAB$^+$17]. We start describing the main types of query workloads and then giving an overview of the query languages.

2.2.3.1 Query workloads

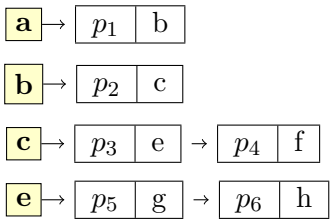
Similar to relational databases, the workloads can be divided in transactional and analytical. In the first type of queries, small fractions of the graph are explored and a quick response time is needed. They can be classified in:

- **Path queries**: This type seeks to determine whether a vertex v can be reached from another vertex w given a set of conditions. A path query has the form $P = x \xrightarrow{\alpha} y$ where α specifies the path conditions and x and y can be variables or specific nodes. For example, satisfying a regular expression over the set of node's or edge's labels or finding a shortest path between both nodes. A path query using regular expressions are known as regular path queries.

- **Navigational/Pattern matching queries**: These queries are more complex structures than the previous ones since their objective is to find sub-graphs in data graph that are

	a	b	c	e	f	g	h
a	0	1	0	0	0	0	0
b	0	0	2	0	0	0	0
c	0	0	0	3	4	0	0
e	0	0	0	0	0	5	6
f	0	0	0	0	0	0	0
g	0	0	0	0	0	0	0
h	0	0	0	0	0	0	0

(a) Adjacency matrix



(b) Adjacency list

Edge	$node_A$	$node_B$
p_1	1	2
p_2	2	3
p_3	3	4
p_4	3	5
p_5	4	5
p_6	4	6

(c) Edge list

Vertex encoding

Vertex	ID
a	0
b	1
c	2
e	3
f	4
g	5

Edge encoding

Predicate	ID
p_1	0
p_2	1
p_3	2
p_4	3
p_5	4
p_6	5

CSR

edge-array=[1, 2, 3, 4, 4, 5]
node-array=[0, 1, 2, 4, 5, 6]

(d) Compressed sparse row

Figure 2.2: Storage structures for graph of Figure 2.1c

isomorphic to a given query. They are graph-structured queries that should be matched versus a database.

The other type of graph queries express analytical graph computations, where a significant fraction of the vertices and edges of the graph are accessed. Similar to SQL in the relational databases some languages offer aggregation operators offering the possibility to group values and compute statistics over these groups.

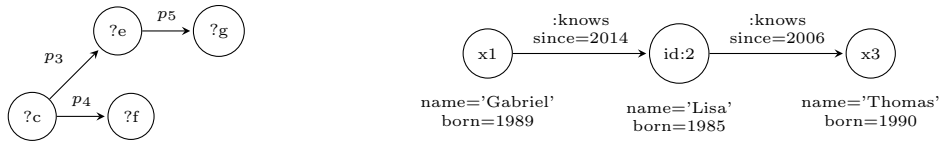
2.2.3.2 Query languages

In this section we give for the most popular modern query languages their main characteristics, strengths and drawbacks. All of the languages presented here share the graph pattern matching operations.

- **SPARQL** [SH13a]: This language was standardized by the W3C in 2008 to query RDF graphs. Its basic graph structure are triple patterns, which are RDF triples composed of a subject, a predicate and an object that may be bounded values or variables. Triple patterns can be combined using conjunctions to form basic graph patterns. Additional to the evaluation of graph matching queries, it offers graph creating functionality with the `CONSTRUCT` operator. The language does not allow query composition (i.e. use the output of a query as the input of another query without persisting the result). An example query for the query represented in Figure 2.3a is:

```
SELECT ?c ?f WHERE {
   ?c   p3  ?e  .
   ?c   p4  ?f  .
   ?e   p5  ?g  .  }
```

- **Cypher** [FGG+18]: This language developed by the developing team of the Neo4j graph system to query property graphs using patterns as its main building block. Patterns are



(a) RDF query (b) Property graph query

Figure 2.3: Graph query examples

expressed giving a graphical intuition to encode nodes and edges with arrows between them. The `MATCH` clause specifies the basic graph pattern, nodes and edges expressed inside parenthesis and brackets respectively. Specific values for properties are specified inside a node within braces and nodes' labels are separated with a `":"` symbol or can also be expressed in the `WHERE` clause. The `RETURN` clause expresses the projected output variables. The language supports selection, projection, grouping, aggregation and ordering. The Cypher query for the property graph query shown in Figure 2.3b is expressed as:

```
MATCH (x1)  - [:knows] -> (id:2 {name:"Lisa"})
MATCH (id:2) - [:knows] -> (x3)
RETURN x1.name AS name1, x3.name AS name2
```

- **Gremlin** [Pro19]: This language is part of the Apache TinkerPop3[1] graph computing framework. It is originally specified to query property graphs but it is quite different to the previous languages. Instead of having SQL-like operators forming a declarative language, it provides a much more programming-like interface focusing on graph traversal operators. For example in the following query using the property graph of Figure 2.3, the call `G.V()` returns the set of all nodes in the graph, then sequences of selections are applied on the set of nodes. The command `out` retrieves all nodes that can be reached with the label knows.

```
G.V().has('name','Gabriel').out('knows')
.has('name','Lisa').out('knows').has('name','Thomas')
```

- **Oracle PGQL**[vRHK+16]: graph query language built on top of SQL providing pattern matching capabilities. It is part of the Oracle Spatial and Graph products[2]. It also allows to create and manipulate existing graphs. The following query expresses the graph query shown in Figure 2.3b.

```
SELECT x1.name AS name1, x2.name AS name2
 FROM   MATCH (x1)  -[:knows]-> (id:2),
        MATCH (id:2) -[:knows]-> (x3)
 WHERE  x2.name = 'Lisa'
```

- **G-Core** [AAB+18]:it is a practical language developed by a industry and academia consortium with the aim of providing a natural syntax to express the most important graph query features. It allows not only to query but to create and compose new graphs. For example, a nested query combining a constructed graph pattern of Figure 2.1d is shown here:

[1] `https://tinkerpop.apache.org/`
[2] `https://pgql-lang.org/`

```
SELECT id(m), m2
MATCH (id:2)-[:knows]->(h),
      (h)   -[:knows]-> (m) ON (
      CONSTRUCT (h) -[:friendOf] -> (m2),
                (m2)-[:knows] -> (m)
     )
```

2.2.4 Query processing

The query processing strategies depend on the type of graph query targeted by the application. These strategies can be as simple as finding the direct neighbors of a node or finding isomorphic graphs given a query pattern, or more complex traversal paths used in algorithms like PageRank. In this section, we give an overview of graph processing strategies used in the subgraph matching queries which are the core of several query languages. Understanding how this type of query is solved serves as a base to understand how more complex graph queries are processed. A detailed description of each algorithm and more complex strategies dealing with reachability is found in [BFVY18].

Let us give some background concepts that will be used throughout this section. Given a direct labeled graph $G = (V, E)$, a *conjunctive query* CQ allows to identify substructures/patterns in G. It asks for all the subgraphs in G matching a given CQ pattern.

Definition 2.1 ***(Conjunctive query)*** *[BFVY18] Let $\mathcal{V}$ be a set of vertex variables. A conjunctive query graph is the expression:*

$$(z_1, ..., z_m) \leftarrow a_1(x_1, y_1), ..., a_n(x_n, y_n)$$

such that:

- $x_1 y_1, ..., x_n y_n \in \mathcal{V}$
- $a_1 ... a_n \in \mathcal{L}$ *(set of labels).*
- $\forall z_i \in \{x_1, y_1, ..., x_n, y_n\}$

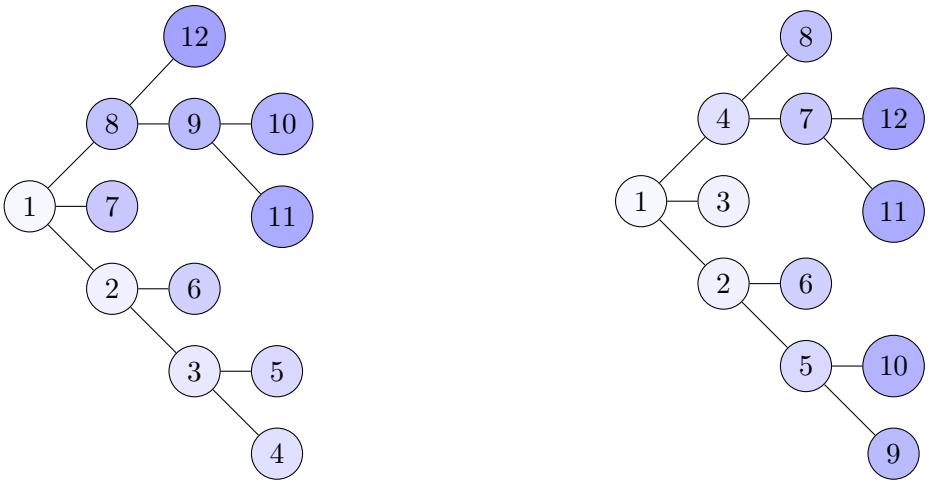
Now, let us define the *mapping* function that as indicated by its name maps nodes and edges in a query to nodes and edges in the graph. More formally:

Definition 2.2 ***(Mapping)*** *[BFVY18] A mapping is a function $\mu : V \mapsto \mathcal{V}$ such that $\forall_{1 \leq i \leq n} \exists e_i \in E$ where for each $e_i = (\mu(x_i), \mu(y_i))$ and a_i are labels of e_i. Evaluating a conjunctive query r over a graph G is expressed as:*

$$[\![r]\!]_G = \{(\mu(z_1), ..., \mu(z_m)) | \mu \text{ is mapping of } r \text{ on } G\}$$

Two graphs $K = (V_k, E_k)$ and $H = (V_h, E_h)$ are *isomorphic* if there is a mapping $\mu : V_k \mapsto V_h$ such that $(x, y) \in E_k \iff (\mu(x), \mu(y)) \in E_h$. In this case, μ maps edges in K to edges in H and non-edges in K to non-edges in H. On the other side, non-edges mappings are not enforce in *homomorphic* mappings such that $(x, y) \in E_k \Rightarrow (\mu(x), \mu(y)) \in E_h$.

Let us consider the graph given in Figure 2.1c and the following conjunctive query: $q = (?c, ?f, ?g) \leftarrow p_3(?c, e), p_4(?c, ?f), p_5(?e, ?g)$. The evaluation of the query on the given graph (named G) is $[\![q]\!]_G = \{(?c, c), (?e, e), (?f, f), (?g, g)\}$.



(a) Depth-first search (DFS) (b) Breadth-first search (BFS)

Figure 2.4: Nodes visiting order

Subgraph matching The process of finding all subgraphs of a given graph G that are iso(homo)morphic to a given conjunctive query r is called subgraph matching. In this case, each subgraph is encoded by a mapping μ of nodes/edges from r to G. In general, the sugraph matching process consists in combining partial mappings obtained first in order to obtain the final subgraph match. A partial mapping is formally defined as follows:

Definition 2.3 ***(Partial mapping)*** *[BFVY18] Given a graph G, a graph pattern $r[z_1, ..., z_m]$ and a mapping μ of r on G, a partial mapping is a sequence of pairs $\langle(v_1, \mu(v_1)), ..., (v_k, \mu(v_k))\rangle$ such that $v_1...v_k$ is a sequence of distinct vertices of r.*

Subgraph matching algorithms are based on depth-first search (DFS) or breadth-first search (BFS) techniques which are described in the following section.

2.2.4.1 Depth-first search (DFS)

DFS-based algorithms for subgraph matching are based on the backtracking principle. This principle is illustrated in Figure 2.4a in which starting from an arbitrary node as the root, the algorithm explores as far as possible along each branch before backtracking. Its implementation is usually based on recursion, but this implementation fails to treat large graphs in a reasonable time. Therefore, the implementation is emulated using an iterative graph search based on states. The searching is guided by a plan, which is an ordered list of vertices indicating the sequence of query nodes assigned to graph vertices while exploring.

An example of such algorithm is the matrix-based subgraph matching which models the query and data graph as adjacency matrices and solves a conjunctive query using matrix operations. For a given query and data adjacency matrices, it defines a permutation matrix P to express the assignment of query vertices to graph vertices. In this matrix, each row i contains exactly one cell p_{ij} equal to 1 (indicating a mapping between the variable node i and the data node j) while other entries are equal to 0. A permutation matrix for the graph and queries of Figures 2.1c and 2.3a respectively is shown in Figure 2.5a.

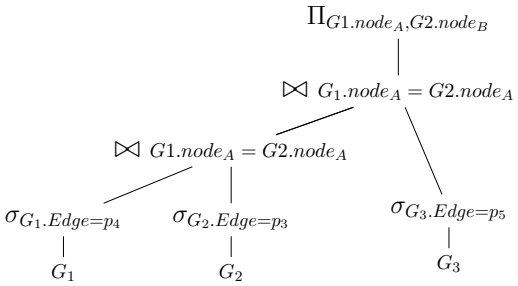
Even if for the previous query there is only a single valid permutation matrix, this is not the general case since a single query might have multiple valid permutation matrices. To check that the permutation matrix encodes an isomorphic mapping, the following expression must hold:

$$A_Q = PA_GP^T$$

where A_Q and A_G represent the query and graph adjacency matrices respectively. The algorithm to find all possible mappings relies on filling P progressively given that the isomorphic condition

	a	b	c	e	f	g	h
?c	0	0	1	0	0	0	0
?e	0	0	0	1	0	0	0
?f	0	0	0	0	1	0	0
?g	0	0	0	0	0	1	0

(a) Permutation matrix



(b) Relational logical query tree

Figure 2.5: Examples of Sections 2.2.4.1 and 2.2.4.2

must be fulfilled in partial permutation matrices. A naïve approach would start at the first row assigning 1 to the first row and column and all the other values to 0. If the isomorphic condition is fulfilled, it continues to the other row and the procedure is called recursively. If the isomorphism condition is violated, it backtracks to the previous row and set the next column to 1 and all the other values to 0. Then it calls the procedure again. It should be noted that the intermediate results are not stored in any structure making DFS-based algorithms memory efficient. More details are found in [BFVY18].

2.2.4.2 Breadth-first search (BFS)

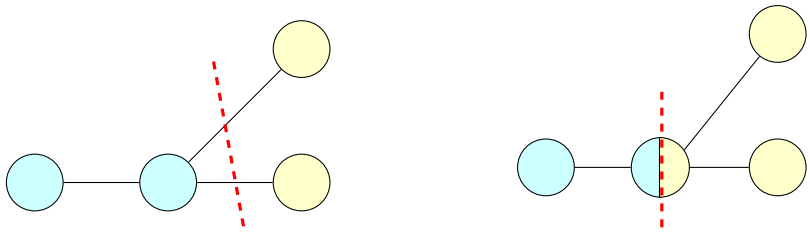
Several subgraph matching algorithms are based on a BFS searching strategy illustrated in Figure 2.4b. Contrarily to DFS, it does not use recursion and its iterative-based making it less complex. It can even be optimized with dynamic programming. However, it needs a structure to memorize the partial solutions that when combined could become a subgraph match. Algorithms based on the relational model follow a BFS graph exploration matching. Considering the query of Figure 2.3a, and assuming that the data is stored in a single table like the one shown in Figure 2.2c. Supposing that the data are stored in a relational database, one of the possible logical execution tree is given in Figure 2.5b. The self joins to the tables are executed pairwise in an equivalent BFS-like subgraph matching strategy. In this case, the intermediate results are stored in main memory causing excessive memory consumption. In the example of Figure 2.5b at least two intermediate result structures must be stored in main memory before projecting the values in the final result.

2.2.5 Graph partitioning

Over the past years, the data stored as graphs has increased considerably. For example, the size of the Facebook data graph counts with a billion nodes and about two hundred billion edges [LNP16]. Distributed and parallel solutions emerged to cope with the demand of efficient graph storage and processing. In these systems, cutting the graph into smaller pieces is a fundamental step to enable parallelism. Graph partitioning must find a balance between the following objectives:

- *Minimize communication overhead*: The communication between hosts dominates the query execution on large scale systems, optimizing this factor is key to optimize the performance. In general, this objective is mapped to the minimization of the total edge cuts (i.e. crossing edges having nodes in different partitions).
- *Maximize load balance*: this objective refers to avoid data and execution skewness, concentrating most of the data in just a few sites.

The problem described previously is also known as the *edge-cut* partitioning problem in which the vertices of a graph are divided into disjoint clusters such that each cluster has almost the



(a) Edge-cut (b) Vertex-cut

Figure 2.6: Partitioning problems

same size whereas the number of edges sharing vertices in two partitions is minimized. A different approach named *node-cut* consists in dividing the edges into equal size clusters such that both vertices of an edge are always assigned to the same cluster. However, in this approach the vertices are not unique since they might be replicated due to the distribution of the node's edges across distinct partitions. In this case, the goal is to: i) *Balance* the partitions in terms of number of edges and ii)*minimize* the number of replicated vertices. Both problems are illustrated in Figure 2.6.

The graph partitioning problem has been proven to be **NP-complete** [GJS76] and finding a reasonable solution to the balanced partitioning is very hard to estimate. In this section we give an overview of the most recent and essential works in *graph partitioning*. The mathematical foundations and a in-depth description of each approach is given in the following survey [BMS+16].

2.2.5.1 Algorithms

In this section we describe the most relevant graph-partitioning algorithms. We divide them into two groups, the first one considers an initial partitioning of the graph as input, these algorithms are known as *graph growing methods*. On the other hand, *global partitioning algorithms* start with the entire original graph and directly compute the partitions. This classification presents a summary of the one presented in [BMS+16]. A schema of the classification of the approaches described briefly in this section is shown in Figure 2.7.

Global partitioning algorithms As it was previously mentioned, these algorithms start with an entire graph and compute a solution directly. They are mainly used to partition small graphs due to its complexity. In fact, some of the approaches are used to produce the initial partitioning required in graph growing algorithms. We describe first *exact* approaches finding the optimal solution to the graph partitioning problem using exhaustive enumeration techniques (e.g., branch and bound). Due to its complexity, most of the methods are dedicated to bi-partition the graph, still they can be generalized to a k-way partitioning by recursion. Most of the exact algorithms use branch and bound to list the candidate solutions. To estimate the bounds of the optimal solution, the graph partitioning has been modeled as a linear program [BCR97] or quadratic program [HK00] just to name a few.

All of the previous methods are able to find the optimal partitions for graphs with very limited size. To speed-up the running time, some heuristics are proposed to find solutions that approximate the optimal graph partitioning. These solutions comprise:

- *Spectral partitioning:* This strategy is applied to split a graph in two groups, for this it analyzes the *spectrum* of a matrix representing the graph. The spectrum is the set of eigenvectors order by their magnitude and their corresponding eigenvalues. The strategy finds the Laplacian matrix of the graph and calculates the second smallest eigenvector which is proven to be a solution for a simplified version of the graph partitioning problem.

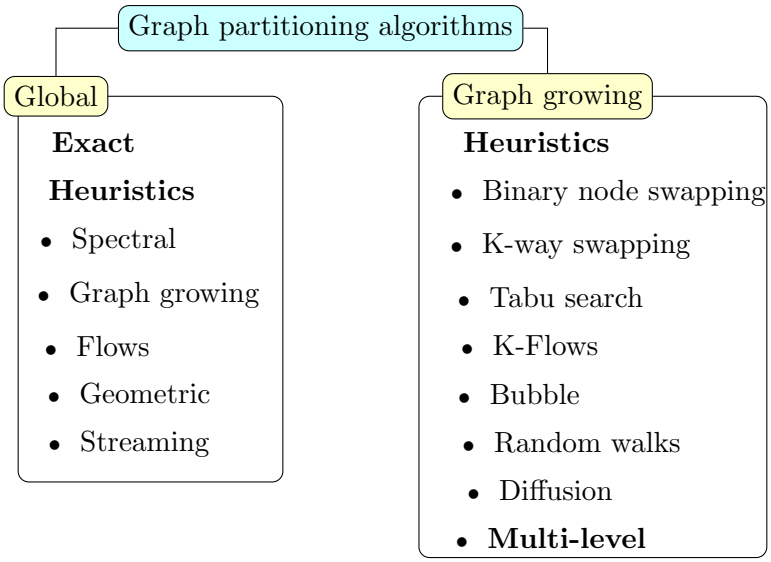


Figure 2.7: Graph partitioning algorithms

They were introduced in the seventies in [DH72] and later improved in works like [BS93] for example.

- *Graph growing:* The algorithms in this category (e.g., [GL81]) and explore the graph using a breath first search (BFS) assigning each visited node to the same group. The process stops when half of the original node weights are assigned to one partition and the remaining nodes are assigned to another.

- *Flows*: This type uses max-flow min-cut theorem to compute the partitions. The work of [BCLS87] for instance calculates the maximum flow between the nodes of the same partition but it ignores the load balance constraint.

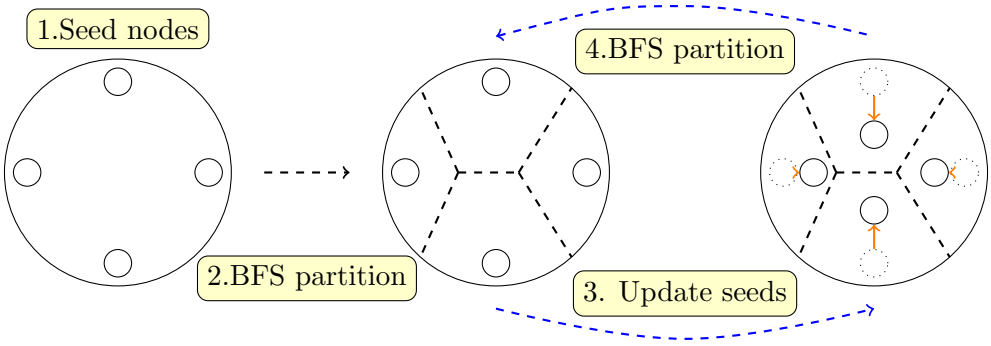
- *Geometric partitioning*: The approaches under this category use the coordinates of the graph in the space. In these approaches nodes geometrically close are mapped to the same partition with the aim of reducing the global edge-cut. They comprise for example the well known space-filling curves [PB94].

- *Steaming partitioning:* This strategy has gain importance in the past year with the development of large-scale streaming applications. These strategies grant more flexibility in terms of solution's quality to increase the partitioning speed. Among these strategies we find [NU13] and FENNEL [TGRV14].

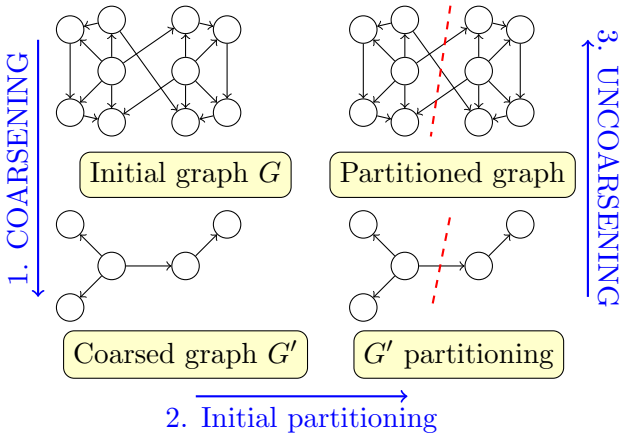
Graph growing algorithms This algorithms iteratively improve starting partitioning solutions. Most of current solvers use this type of heuristics. They can be classified as:

- *Binary node swapping*: This strategy iterates over a set of nodes and its neighbors and swaps nodes from each partition. With this action the total number of cuts decreases. The solution was first presented by Kernighan and Lin in [KL70] and later improved by many other authors.

- *K-way swapping*: These approaches are a workaround to the recursive application of the bisection methods. They propose to directly partition the graph in k parts by using priority queues [San93] or a global priority queue [KK98a] to control the number of swaps between nodes.



(a) Bubble schema



(b) METIS schema

Figure 2.8: Graph algorithms schemas

- *Tabu search*: As its name indicates, these strategies use tabu search to explore the set of solutions (e.g., [GBF11]). Their calculation is more expensive than the previous k-way swapping.
- *K-Flows*: this method presented in [SS11] improves a given bipartition by growing the area around the boundary nodes/cut edges.
- *Bubble*: This method presented in [DPSW00] is quite similar to k-means, the method randomly select k seed nodes that are evenly distributed over the graphs. Then, it uses BFS to grow each of the k-nodes, the seeds are then recalculated. The algorithm stops when it is impossible to no more improvement partition is found for more than 10 iterations or if the seed nodes does not change. A schema of this strategy is shown in Figure 2.8a.
- *Random walks and diffusion*: these approaches start at a node and choose randomly from its neighbors another node to visit using a probabilistic transition matrix. The idea of the approach is to detect dense graph regions based on the intuition that if a region is dense, it is quite difficult to leave the region after several steps. These intuition is described in [Sch07].
- *Multi-level*: The implementation of this partitioning approach (METIS [KK98a] is until today, one of the most successful and most applied graph partitioning technique. This technique splits the graph partitioning process in three stages. In the first phase it coarsens the original graph G to form a condensed representative graph of G. Then it applies a partitioning algorithm (K-way swapping for instance) to the compressed graph. Finally, the coarsed graph is goes back to its initial shape and the partitions are refined. A schema of this procedure is shown in Figure 2.8b.

Table 2.1: Dynamic graph partitioning approaches

System		Main technique	Changing		Replication
			Workload	Graph	
ParMETIS	[SKK97]	• Difussion schemes.		✓	
xDGP	[VCLM13]	• Greedy vertex migration.		✓	
Hermes	[NKDC15]	• Neighboring vertices.		✓	
Leopard	[HA16]	• Neighboring vertices.		✓	✓
Inc	[FLT+20]	• Incrementalization.		✓	
Fennel	[TGRV14]	• Stream graphs. • Neighboring vertices.		✓	
Spinner	[MLLS17]	• Pregel label propagation.		✓	
Sedge	[YYZK12]	• Two-level partition.	✓		✓
	[TÖ15]	• Graph access patterns.	✓		
TAPER	[FM17]	• Stability. • Random walks.	✓		
IOGP	[DZC17]	• Hasing, • Edge-cut and split.	✓		

In current large-scale graphs, the previous methods are unable to produce partitions in a reasonable time. Some efforts were maid towards parallelizing some of them, for example the distributed version ParMETIS [KK98b]. It exists other partitioning strategies highly efficient in terms of loading times and low memory costs, however they do not consider the edge-cut and load balance constraints. These approaches are mainly based on *hashing*, like for example the 2D-hash strategy of the GraphX framework [GXD+14].

The approach named Ja-Be-Ja was introduced in [RPG+15] claiming that it outperforms METIS for large graphs. The algorithm uses local search and simulated annealing to partition the graph using an *edge-cut* or *vertex-cut* strategy (both illustrated in Figure 2.6). The algorithm does not need to know the entire graph to generate the partitions since it is based on operations at the vertex level. Each node has information about its neighbors and it is initially assigned with a color (partition). At each step, a node's color can change according to the most prevalent color from its neighborhood.

Another similar scalable graph partitioning algorithm named Spinner was presented in [MLLS17]. It is based on the Label Propagation Algorithm on top of the large-scale graph platform Apache Giraph [Fou19] (described in Section 2.2.6). If k partitions should be formed, it starts assigning each vertex to a k_i partition randomly. Then, it iterates over a modified k-way balanced label propagation algorithm to assign the vertices' labels to its neighbors until it converges. Whenever data are added or removed from the graph, Spinner reiterates to find the converging assignment. This type of approaches considering changing workloads or variations to the original graph are named *Dynamic algorithms* and are described in the following section.

Dynamic algorithms Current graph-based applications deal with data which is not only large, but it is also updated. Additionally, the user's needs may also change leaving the initial workload out of date. Re-partition the graph is complex and expensive, hence partitioning the graph from scratch is not always an option. There have been some works in dynamic partitioning of graphs. The list is not exhaustive, more details on other dynamic algorithms is given in [DCL18]. All of the works presented in this section are summarized in Table 2.1.

The same creators of METIS introduced in [SKK97] a repatritioning technique for changing graphs based on diffusion. The works presented previosly on streaming techniques are also dynamic approaches. In these works the changes occur in the graph and not in the workload.

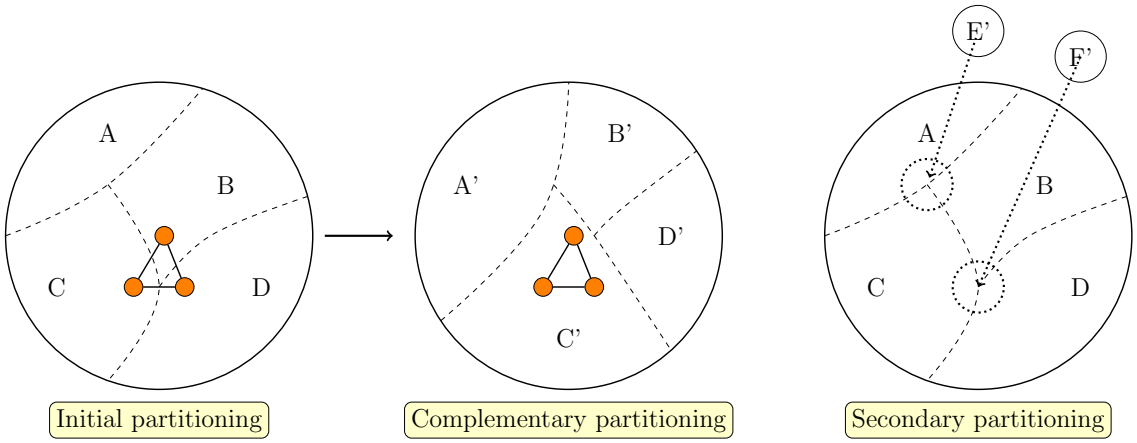


Figure 2.9: Partitioning in Sedge

The work of Tüfekçi and Özturan in [TÖ15], introduces a framework for changing workload using access patterns. The proposed system partitions a graph database and provides a fully functional distributed graph database system.

The dynamic replicated algorithm Sedge (Self Evolving Distributed Graph Management Environment) proposed in [YYZK12] creates new partitions or replicates to cope with variations on the workload. It proposes a two-level graph partitioning architecture with complimentary primary partitions and dynamic secondary partitions. The starting point is an initial partitioning, then based on the query workload it creates secondary partitions (as replicas) reducing the original cross partition edges. The secondary replicas are either full replicas of the initial partitions or replicas of the cross-partition hotpots. These replicas are either complimentary created to the initial partitioning or they are created on-demand. It is build to work specially in the in-memory computation model of Pregel. An illustration of the partitioning strategies presented in Sedge is shown in Figure 2.9.

In [VCLM13] the authors introduced xDGP, a system that dynamically repartitions massive graphs to adapt to structural changes in the graph inside a system based on Pregel. The system adopts an iterative vertex migration algorithm that relies on local information only, making complex coordination unnecessary. It starts hash partitioning the graph and defining a capacity constraint to avoid skewness. It uses a greedy vertex migration strategy that buffers the nodes that should be moved based on local information available to the vertex whose goal is to reduce the global edge-cut.

Another approach considering evolving graphs is Hermes [NKDC15] which is built on top of the Neo4j platform. This approach repartitions dynamically and on the fly on multiple servers the existent partitions of data. Their algorithm is based on the number of neighbors measure to decide whether a vertex should be moved.

The approach named Leopard [HA16] is lightweight a dynamic graph partitioning approach which simultaneously considers a replication mechanism to reduce edge-cut. The intuition behind the proposed method is to assign a vertex to the partition containing most of its neighbors, penalizing large partitions when they are too large. It continuously revisits vertices and edges that have already been assigned to keep vertices close to its neighbors.

In [FM17], Firth and Missier present TAPER, a graph partitioning system that is sensitive to evolving workloads. Their system takes as input any given initial partitioning and iterates to adjust the partitioning taking the workload into account. For this, it uses the intuition of stability used in community detection. When a partition is stable, a random walk inside that partition should remain in the same partition for a long time.

In IOGP [DZC17], the authors presented a partitioning algorithm optimized for transactional workloads. Its methodology is divided in three stages: quiet, vertex reasoning and edge splitting

stage. In the quiet stage, the vertices are placed using a hashing function. In the vertex reasoning phase, checks as more vertex are inserted how connected they are to other vertices to re-assign them if necessary. The last stage named edge-splitting splits edges of high degree into multiple servers to enable parallelism.

More recently, [FLT+20] proposed to incrementalize current batch partitioners instead of developing yet another dynamic graph partitioning method. In their work, they formalized the incremental graph partitioning problem adding an objective to the ones presented at the beginning of this section: minimizing the changes considering that a full repartition is expensive. They managed to increment some edge-cut and streaming partitioners and their experiments verified the effectiveness of their approach.

2.2.6 Graph databases

In the previous sections we characterized the following dimensions of the graph database model: i) Logical data structure, ii) Data storage, iii) Query and manipulation languages, iv) Query processing strategies, and v) Graph partitioning. In this section we use these dimensions to classify some of the most popular graph stores. Our list is not exhaustive, we considered only the most relevant approaches to show the evolution of the available graph-oriented technologies. Systems built to manage RDF data will be treated in detail in Section 2.3, hence we will not describe them in this section to avoid redundancy.

We classify the approaches in two major groups: *native* and *non-native*. The first group consists in systems specifically designed to store and query data represented as graphs. In contrast, non-native approaches are built on top of existent platforms like relational databases, key-value stores and other cloud systems. In this section we describe the most relevant systems from both categories by platform (centralized or parallel). A summary of all the presented systems is given in Table 2.2.

2.2.6.1 Centralized systems

These systems perform the graph computation only in one machine. The early graph-based systems described in the introduction of this section enter in this category (e.g., [Güt94]). We find here the leading graph database system that made the property graph data model popular, Neo4j [Web12]. Neo4j is a Java based system in which property graphs are represented through edge lists. Each edge has an entry composed of two linked lists, one for the source s and another for the destination d vertex, storing the previous and next edges connected to s and d respectively. The system allows to query the data directly with the Cypher language or via REST, Java, Javascript or Python APIs.

Before Neo4j a big number of RDF systems had already been developed. The system DEX introduced in [MMG+07] is an in-memory graph database allowing to query multiple sources. It was written in C++ (although an API is proposed to facilitate the access to data) and efficiently process graph operations using bitmaps. Another in-memory system, HypergraphDB [Ior10], proposed to model the data as hypergraphs. Its structures are stored as adjacency lists in the BerkleyDB key-value store [OBS99]. Queries are defined as traversal operations using iterator APIs.

More recently, the system GraphChi-DB [KG14] proposed an efficient data structure which is basically an immutable flat array named Parallel Adjacency Lists to manage and analyze graphs with billions of edges in a single machine. The data are stored in a CSR structure which is partitioned in P intervals such that a single partition P_i could fit into main memory. Each partition partitioned by destination vertices ID's and its ordered by source vertex ID. This strategy guarantees an efficient lookup of vertices when querying with Cypher or SPARQL query languages.

Table 2.2: Graph databases mentioned in Section 2.2.6

			Graph				Query	
System		**T**	**Struct.**	**Storage**	**Part.**	**Mech.**	**Lang.**	**Proc.**
DEX	[MMG+07]	N	DLG	Bit index	✗	Im	API	BFS
HypergrDB	[Ior10]	N	Hyp	Adj. list	✗	Im	API	BFS DFS
Neo4j	[Web12]	N	PG	Edge list	✗	D	Cypher API	BFS
Graphchi	[KG14]	N	DLG	CSR	✗	D	Cypher Sparql	BFS
Distributed								
Trinity	[SWL13]	Nn	PG	Adj. list	Hash	Im	API	K-v
TAO	[BAC+13]	Nn	PG	Edge list	Flex	D	API	Rel
Titan	[Dat15]	Nn	PG	Adj. list	Hash	D	Gremlin	K-v
GraphX	[XGFS13]	Nn	PG	CSR	Vertex-cut	Im	API	Spark
Pregel	[MAB+10]	N	DLG	Adj. list	Hash	Im	API	BSP
Giraph	[Fou19]	N	DLG	Adj. list	Hash	D	API	BSP Hadoop
GraphLab	[LGK+12]	N	DLG	Adj. list	Vertex-cut	Im	API	GAS

D:Disk, **DLG**: Directed labeled graph, **Hyp**: Hypergraph, **Im**: In memory, **K-v**: key-value store, **N**: Native, **Nn**: Non-native, , **Part:** Partitioning strategy, **Proc:** Processing strategy, **Rel**: Relational.

2.2.6.2 Parallel systems

The systems in this category can handle extremely large graphs with billions of edges on a cluster of machines by partitioning the graph so that each machine stores a part of the graph to process it in parallel.

Non-native The systems in this category are built on top of cloud platforms that avoid building systems from scratch simplifying the storage and processing. In these systems, the graph is stored using the specific data structures and operators of the cloud system. Microsoft Trinity [SWL13] for instance is built on top of an existent shared-memory key-value store. Trinity represents the graph in adjacency lists that are fully processed in-memory achieving low latency to solve queries. Similarly, Titan[Dat15] is built on top of HBase and Cassandra a key-value store and a wide-column system respectively. The system TAO [BAC+13] was introduced to to treat read-only queries on the Facebook's social graph. It is built on top of a distributed NoSQL database adding an intermediary layer to access the data stored in MySQL using caches. GraphX [XGFS13] is a graph computing library built on top of Apache Spark[3]. It stores vertices and edges

[3] `https://spark.apache.org/`

in two RDDs which are Spark's distributed collections. Graph operations are transformations applied to the vertices and nodes RDDs.

Native Most of the systems in this category are based on the Bulk Synchronous Parallel (BSP) programming model which uses a message passing interface (MPI). BSP computation consists on a series of supersteps that are synchronized when moving from one step to another (in different machines). Google researchers were the pioneers to use the BSP model in an in-memory graph system named Pregel [MAB+10]. The programming model used was later known as *vertex-centric* since it is based on message propagation from vertex to vertex. At runtime, messages are received and sent from/to the node's neighbors that update its current state and also the edges' state.

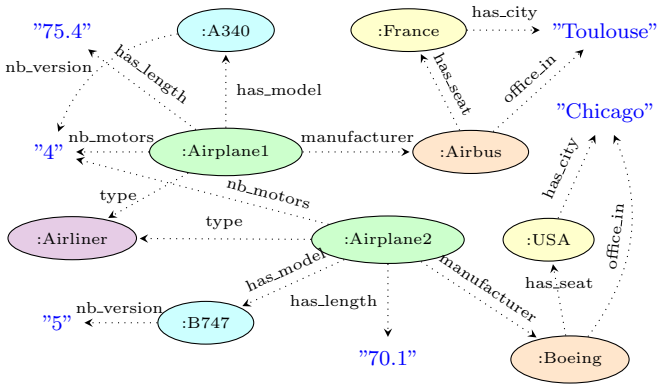
The source code of Pregel was not public, however many systems followed the execution model described in detail in the Pregel paper. Systems following this BSP are called *synchronous*. Apache Giraph [Fou19] is an open source Java implementation of Pregel. It stores the data as vertex adjacency lists in the Hadoop Distributed File System HDFS and the operations are run as map jobs. Several approaches have been built on top of Giraph, some extensions are for instance Giraph++ [TBC+13]. This system proposes a graph-centric paradigm in which the messages are propagated faster inside a graph close neighbors. An experimental survey about these systems is found in [HDA+14].

Asynchronous systems claim to be faster than the previous approaches but they are much more complex [CEK+15]. Graphlab [LGK+12], a system developed in C++ for shared-memory architectures, belongs to this category. The computation of this system is done through shared state instead of shared messages. A vertex is therefore read/updated from its own data or that of its neighbors instead of using messages.

Commercial graph databases The commercial offer of systems able to store and query graphs has considerably increased in the past years. The most popular graph database system is Neo4j[4] which offers a graph processing compliant with ACID transactions. It offers a reasonable performance for Cypher and traversal queries in a centralized environment. Distributed solutions comprise systems like DGraph[5] introducing a json-based query interface (GraphQL). Also, other distributed graph systems like Virtuoso[6] and Stardog[7] support RDF queries in SPARQL. Besides, several systems are part of multi-model databases built on top of recent cloud platforms, like for example: CosmosDB[8] (Microsoft Azure), Amazon Neptune[9] (AWS), OrientDB[10] (SAP), ArangoDB[11].

So far we have not discussed the systems specifically designed to store and query Web data standardized as a graph. In the following section we give an overview of this standard and detail the systems conceived to treat specifically this data model.

[4] https://neo4j.com/
[5] https://dgraph.io/
[6] https://virtuoso.openlinksw.com/
[7] https://www.stardog.com
[8] https://azure.microsoft.com/en-us/services/cosmos-db/
[9] https://aws.amazon.com/neptune/
[10] https://orientdb.com/
[11] https://www.arangodb.com/



(a) RDF graph

```
<:Airplane1> <:has_model> <:A340>.
<:Airbus> <:has_seat> <:France>.
<:Airbus> <:office_in> "Toulouse".
<:Airplane1> <:has_type> <:Airliner>.
<:Airplane20> <:has_length> "70.1".
<:Airplane20> <:manufacturer> <:Boeing>.
<:Airplane1> <:has_length> "75.4".
<:Airplane1> <:nb_motors> "4".
<:France> <:has_city> "Toulouse".
<:Airplane1> <:manufacturer> <:Airbus>.
<:Boeing> <:has_seat> <:USA>.
...
```

(b) RDF document

Figure 2.10: RDF example

2.3 Resource Description Framework

The Resource Description Framework (RDF) [RC14] has been widely accepted as the standard data model for data interchange in the Web. RDF is flexible enough to facilitate the integration of data with different schemas. The model uses triples consisting of a subject, a predicate and an object (s, p, o) as the core abstract data structure to represent information. A set of such triples forms a directed labeled graph (described in section 2.2.1) called an RDF graph. The nodes represent IRIs (international resource identifiers), blank nodes (unspecified resources) or literals. Edges on the other side characterize a resource by linking subject and object nodes by a predicate edge. An RDF graph is shown in Figure 2.10a. We formalize the previous concepts in Section 2.3.1, then we present the storage models in Section 2.3.2. The storage models described are used in centralized and distributed RDF system. Then, in Section 2.3.3 we classify RDF systems by processing strategy. Finally, in Section 2.3.4 we describe some of the distributed systems that scale to massive RDF datasets, highlighting the partitioning strategies applied by these systems.

The purpose of this entire section is to give an overview of RDF, focusing on the storage and partitioning techniques. Our goal is not to describe in detail each of the current RDF processing systems since there are several surveys dedicated to this (e.g., [KM15, Özs16, AHKK17, WHCS18, ASYNN20]). In this section, we mention only some of the most relevant systems.

2.3.1 Background

An RDF statement is a triple representing a relationship between the subject and object. It is defined as follows:

Definition 2.4 ***RDF Triple*** *[KM15] A triple (s, p, o) is a tuple from $(U \cup B) \times U \times (U \cup L \cup B)$ where s is known as the subject, o the object and p is a relationship between s and o known as the predicate. U, B, L are the sets of IRIs, blank nodes and literals respectively.*

As it was previously mentioned, *IRIs* are the identifiers of Web resources which can be reused in other datasets to represent the same resource and avoid redundancy. We will not delve further into its construction, more details are given in [W3C08]. *Blank nodes* represent anonymous resources in the subject or object position of the triple. Its syntax is not regulated by the RDF standard, its identification varies according to specific datasets. *Literals* are values like strings, numbers and dates. A literal is composed of a lexical form, a datatype IRI and a language tag even though some syntaxes may support simple literals with just the lexical form.

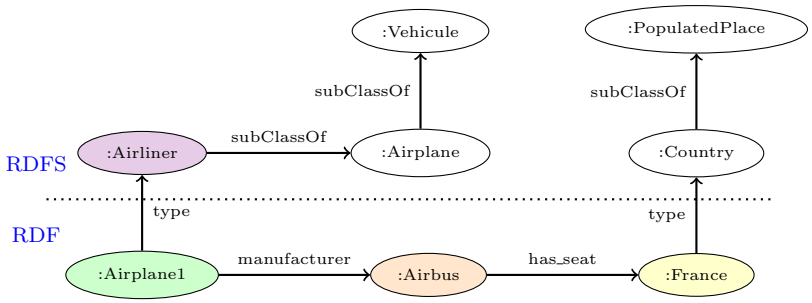


Figure 2.11: RDF-Schema associated to graph of Figure 2.10a

The IRI's of the nodes of the RDF graph shown in Figure 2.10a where simplified by adding a ":" representing the complete IRI. Also, the datatypes were omitted in the literal nodes. For example, the triple `(:Airplane1,has_length,"75.4")` is actually:

`(http://example.org#Airplane1,http://example.org#has_length>,"75.4"^^xs:decimal)`.

A set of RDF triples forms an RDF graph defined in [Özs16] as follows:

Definition 2.5 ***RDF Graph*** *[Özs16] An RDF graph is a six-tuple* $G = \langle V, L_V, f_V, E, L_E, f_E \rangle$ *such that:*

- V *and* L_V *are the set of vertices and their labels respectively.* V *corresponds to all subjects and objects in the data.*
- E *and* L_E *are the collection of directed edges connecting subjects and objects and their corresponding labels respectively. The label of an edge corresponds to the property connecting a subject and object in the data.*
- $f_V : V \to L_V$ *and* $f_E : E \to L_E$ *are bijective functions assigning vertices and edges with their respective label.*

An RDF graph is **encoded** in a document using concrete syntaxes such as N-Triples, Turtle, TriG and JSON-LD. The most basic syntax is N-Triples representing one triple per line as shown in Figure 2.10b.

The RDF data model was extended with annotators like the **RDF Schema** (RDFS)[DB14] and OWL [Gro12]. The RDFS language allows to define more precisely the schema of RDF data inspired in object-oriented programming models. It provides a vocabulary to describe resources with classes, class hierarchies and properties, adding semantics that enable reasoning over the data. Resources are divided in groups called classes, the members of a class are known as class' instances. Classes are themselves resources represented as triples which can be seen as a graph. The principal built-in class definitions are `rdfs:Class`, `rdfs:subClassOf`, `rdfs:domain` and `rdfs:range`. An example of RDFS annotations to the data of Figure 2.10a is given in Figure 2.11. If a more complete schema is needed, ontology languages offering more expresiveness like OWL [Gro12] could be a good solution.

The standard language to query RDF is **SPARQL** [SH13b] which supports to query required and optional graph patterns, using conjunctions, disjunctions, subqueries and aggregations. An example of a SPARQL query and its representation as a query graph is given in Section 2.2.3.2. SPARQL is based on graph pattern matching. The Basic Graph Patterns (BGP) queries are the most commonly used. A BGP query is a set of *triple patterns* (s, p, o) of the form $(U \cup B \cup \mathcal{V}) \times (U \cup \mathcal{V}) \times (U \cup L \cup B \cup \mathcal{V})$ where $\mathcal{V}$ is a set of variables. A sequence of triple patterns, with optional filters comprises a BGP query, its syntax (not considering `FILTER` constraints) is expressed in [KM15] as :

$$\texttt{SELECT}\ ?v_1\ \ldots\ ?v_m\ \texttt{WHERE}\ \{t_1\ \ldots\ t_n\}$$

where $\{t_1, ..., t_n\}$ is a set of triple patterns and $?v_1...?v_m$ is the set of variables occurring in $\{t_1...t_n\}$.

An example SPARQL query for the RDF graph of Figure 2.10a finding the airplanes whose length is greater than 65.0 meters is specified as follows:

```
SELECT ?a ?m ?l
WHERE {
   ?a :type :Airliner .
   ?a :has_length ?l .
   ?a :has_model ?m .
   FILTER (?l > 65.0)
}
```

In this query, the first tree lines in the `WHERE` clause consist of three triple patterns having variables such as `?a, ?l, ?m`. The `FILTER` clause restricts the solutions matching the triple pattern based on numeric values or strings (with regular expressions). The formal definition of a SPARQL query match is given in Definition 2.6.

Definition 2.6 ***(SPARQL match)*** *[Özs16] Considering an RDF graph G and a SPARQL query Q having $\{v_1, ..., v_n\}$ distinct subjects and objects. The set $\{u_1, ..., u_n\} \in G$ is called a match of Q if there exists a bijective function $\mu : G(U \cup B \cup L) \rightarrow Q(B \cup \mathcal{V})$ where $v_i = \mu(u_i) \forall_{1 \leq i \leq n}$ and $\forall_{v_i}$ the following conditions are fulfilled:*

- *Literals/IRIs: if v_i is a literal or an IRI $\rightarrow u_i$ has the same value in G.*
- *Variables: if v_i is a variable vertex $\rightarrow u_i$ must satisfy the filter constraint (if any) over the parameter v_i.*
- *Edges: all edges and edge's labels in Q must have a corresponding mapping in G, $\forall (v_i, v_j) \in Q \rightarrow (u_i, u_j) \in G$. Also, the edge label must satisfy the filter constraint if any.*

SPARQL supports more complex operators (e.g., `OPTIONAL`, `ASK`, `UNION`, `ORDER BY`, `LIMIT`) that are not considered in this chapter but whose details are found in [SH13b]. We focus our work on the core Basic Graph patterns supported by most of the state of the art systems described in the next section.

2.3.2 Storage models

In contrast to relational databases in which the storage layout is easily represented in a table, in RDF the storage structure depends on the system and the strategy used to further process the data. Let us recall the most popular relational databases storage layouts: *i) Row-wise* or traditional layout scheme (N-ary storage model NSM), *ii) Columnar* or decomposition storage model (DSM) [CK85], and *iii)* partitioning attributes across (PAX) [ADH02]. All of the previous layouts derive directly from the core data structure of the relational model, the table. They store the table's data row by row (NSM), column by column (DSM), or store first row by row and finally partition by attributes (PAX). The storage model of early RDF systems is based on the relational model. Such approaches rely on relational database techniques to efficiently solve SPARQL queries translated to SQL. These storage approaches are described in Section 2.3.2.1. More recently, some systems represent directly the data as a graph, not using an intermediate representation like a table. These approaches are detailed in Section 2.3.2.2.

2.3.2.1 Relational-based

This storage model stores triples (s, p, o) in relational database tables using one of the following strategies:

- *Triple table*: this approach initially embraced by Sesame [BKvH02] stores the data in a single table of three columns (subject, predicate, object). An example of this storage is illustrated in Figure 2.12a. In terms of space complexity it equals the size of the graph and without indexes lookups require scanning the whole table. This is in fact its major drawback, the processing of self-joins turns quite expensive when SPARQL queries become more complex. To improve performance, triple tables are lexicographically ordered in several optimization strategies have been proposed, for example:

  - *Indexing*: establishing a lexicographical order of the table is possible on single columns. If the query involves a lookup in one of the unordered columns, it may require to scan the entire table. The *naive* strategy to overcome this problem creates a primary index of the other columns despite replication. A cheaper option in terms of space are *secondary indexes* and *single-column indexes* adding a column to the triple table with pointers. The popular RDF system Viruoso [EM09] built on top of a relational database stores the data in a table of 4 columns (graph G, subject S, predicate P, object O). It is composed of a main index GSPO and a secondary OPGS bitmap index. Another popular indexing strategy stores the *permutations* of the single table (SPO, OPS, PSO, SOP, POS, OSP). This technique is strongly inspired from the indexing strategies proposed in the seventies in [Lum70, Shn77]. This techinque avoids a full scan of the table for any single query pattern. Also, the index could be built only on a subset of the columns (e.g., SP, PO). This strategy, named *projection indexes*, allows to solve queries using only merge joins. The most popular system using a combination of both strategies (exhaustive and projected indexes) is RDF-3X [NW08] which indexes all the permutations of the table's attributes and some of its projections. More recently, RDFox [NPM+15] uses only three index columns and three secondary projection indexes.

  - *Dictionaries & encoding*: to reduce the size of a triple table, which can become very large due to redundancy, several compression techniques are proposed to improve the reading efficiency. Using dictionary tables mapping each individual value to a dense domain of positive integers reduces considerably the lookup time, reducing it to a logarithmic search instead of a full scan. Deeper encoding techniques based on byte encoding reduce even more the space of the encoded data storing only the increment (delta) of one row to another. For example, in the triple table of Figure 2.12a, the subject of the first row will be encoded, while a 0 byte will be assigned to encode the subject of the second row since it is the same as the first one. The byte delta encoding technique is embraced by the RDF-3X [NW08] system.

- *Binary table:* this strategy creates a two-column table (subject, object) per predicate. It is illustrated in Figure 2.12b The strategy is applied by SW-Store[AMMH09] and HadoopRDF[DWNY12] for example. It suffers from join overheads when a query involves several predicates.

- *Property table:* this approach also known as pivited table aims to reduce the number of self-joins storing the data in a wider table whose dimensions correspond to the number of distinct subjects and predicates. The drawbacks of this approach are the great number of `null` values in the table causing storage overheads, the complexity to treat multi-valued properties and its fixed nature not supporting the schema-flexible nature of RDF (e.g., complex to add a triple with a new property). An excerpt of a property table is shown in Figure 2.12c. This strategy is applied by the Jena2 [Wil06] system. Many strategies have been proposed to alleviate some of the problems described previously within which we find:

Subject	Predicate	Object
Airplane1	has_model	A340
Airplane1	has_length	75.4
Airplane1	nb_motors	4
Airplane1	type	Airliner
...	...	...

(a) Triple table

nb_motors

Subject	Object
Airplane1	4
Airplane2	4

...

nb_version

Subject	Object
B747	4
A340	5

(b) Binary tables

	has_model	has_length	nb_motors	type	nb_version	...
Airplane1	A340	75.4	4	Airliner	null	
Airplane2	B747	70.1	4	Airliner	null	
B747	null	null	null	null	5	
...						

(c) Property table

Figure 2.12: Relational-based storage of the graph of Figure 2.10a

– *Emerging schemas:* this strategy reduces the number of null values in the table by partitioning it. Each partition gathers the data sharing a similar schema. The idea is that although RDF data does not define an explicit schema, its schema is implicit in the data and can be found with diverse approaches. An example of a property table partitioned using emerging schemas is shown in Figure 2.13a. Some of the most relevant approaches to find emergent schemas are:
  * The clustering strategy for RDBMS described by Chu et al. named *hidden schemas* in [CBN07] can be very useful to find emerging schemas. In their work, the authors mapped the table's attributes to a weighted graph and then used the k-nearest neighbor partitioning algorithm to create the sub-relations along with the Jaccard similarity to measure the strength of co-occurrence between the two attributes.
  * The term *emergent schema* for RDF was introduced in [PPEB15]. In this work, the authors used the characteristic sets *cs* defined in [NM11] and formally described in Definition 2.7 to cluster triples and create the implicit schema tables. The characteristic sets for the entire table are merged based on a semantic and structural scores. The structural score identifies discriminating properties on each *cs* using an adapted TF/IDF similarity score. The distributed RDF system EAGRE [ZCTW13] used as well the characteristic sets to distinguish entities and store the data of similar entities together.
  * More recently, the approach Cinderella [HVL14] proposed a strategy to partition partitioned incoming data on the fly based on their similarity schema and in an existent partitioning which can be obtained with either of the previously described methods. The system fix the maximum size for partitions and new entities are assigned to the closest partition.
– *Hashing:* This technique embraced by DB2 for RDF [BDK+13] maps the predicate and the object of a triple to a column group using a hashing function. The number of column groups is given as a parameter, collisions are stored in new rows (named spill row) adding a flag column as shown in Figure 2.13b. A reverse primary hash could be applied by hashing the values to groups from the triple's object.
– *Bit encoding:* The system BitMat [ACZH10] propose a bit-wise matrix representation of property tables. In this matrix, each cell is either 0 or 1 representing the absence or existence of that triple. An example of this strategy is shown in Figure 2.13c. During query processing, queries are executed on the compressed data.

Subject	has_model	has_length	nb_motors	type
Airplane1	A340	75.4	4	Airliner
Airplane2	B747	70.1	4	Airliner

...

Subject	has_seat	officie_in
Airbus	France	Toulouse
Boeing	USA	Chicago

(a) Emerging schemas

Subject	1		2		3		Spill
Airplane1	has_model	A340	has_length	75.4	nb_motors	4	1
Airplane1	null	null	type	Airliner	null	null	1
Airplane2	has_model	B747	has_length	70.1	nb_motors	4	1
Airplane2	null	null	Airliner	Airliner	null	null	1
Airbus	has_seat	France	officie_in	Toulouse	null	null	0
...							

(b) Primary hash

	has_model	has_length	nb_motors	type	nb_version	...
Airplane1	1 0 0 0 0 0 0	0 1 0 0 0 0 0	0 0 1 0 0 0 0	0 0 0 0 1 0 0	0 0 0 0 0 0 0	...
Airplane2	0 0 0 0 1 0 0	0 0 0 0 0 1 0	0 0 1 0 0 0 0	0 0 0 0 1 0 0	0 0 0 0 0 0 0	...
B747	0 0 0 0 0 0 0	0 0 0 0 0 0 0	0 0 0 0 0 0 0	0 0 0 0 0 0 0	0 0 0 0 0 0 1	...
...						...

*Each bit sequence represents the object's sequence `(A340, 75.4, 4, Airliner, B747,70.1,5)`

(c) Bit encoding

Figure 2.13: Property table optimizations of Figure 2.10a

Definition 2.7 ***(Characteristic set)*** *[NM11] Each subject s in an RDF graph G has a characteristic set defined as* $\overrightarrow{cs}(s) = \{p|\exists_o : (s, p, o) \in G\}$.

2.3.2.2 Graph-based

This storage strategy maintains the graph structure of RDF representing the data using the structures representing the adjacency of directed graphs defined in Section 2.2.2 of this chapter. At query runtime, these approaches exploit the graph nature of RDF and SPARQL reducing the query execution to a subgraph matching problem. These structures allow efficient lookups of the neighbording vertices for a given vertex. We will not detail each of the structures again to avoid redundancy. However, we will describe some of the most relevant optimization strategies used by some of the graph-based systems:

- *Encoding:* Similarly to triple tables, data in adjacency structures are compressed and grouped to streamline the lookup process. In this line, ordering the vertices lexicographically should be complemented with other ordering strategies that reflect data locality (close neighbors should be placed close in the data structure). The adjacency list could be encoded with dictionaries as in the relational-based approaches, using bit compression deltas to encode neighbors [BV04] or more complex tree-based structures (K^2-trees [BLN09]). GStore for example.

- *Indexing:* the simplest strategy indexes the keys of the adjacency lists in a B+Tree if the vertices are lexicographically sorted. More complex secondary structures concern the problem of finding all vertices that are reachable from a given vertex useful for long path and distance queries. Among these strategies we find the 2-Hop indexing scheme [CHKZ03] and the tree-based index FERRARI[SABW13].

The first system that considered RDF directly as a graph was presented in [BHS03].

This system was was built on top of an object-oriented database. Later, the system BitMap [ASH08] used an adjacency matrix. The matrix encodes the triples of an RDF graph completely in a bit matrix that is flattened in two dimensions.

Both of the optimization strategies described previously are applied in the graph-based RDF processing system gStore [ZÖC+14]. This system encodes the entities in adjacency lists grouping the vertex's and its adjacent edge labels into a fixed length bit string. The encoded data are stored in a VS*-tree index (height-balanced tree where each node is a bitstring that corresponds to an encoded vertex). An incoming SPARQL query is also encoded using a similar structure and the query problem is reduced to a sub-graph matching problem. Systems built on top of key-value stores also integrate this category. Trinity.RDF [ZYW+13] is built on top of the in-memory key-value store Trinity. For each node it stores the adjacency lists of outgoing and incoming edges helping the query evaluation seen as a graph exploration process. Similarly, the commercial graph system Stardog[12] supports SPARQL of data stored in the RocksDB[13] key-value store. The major issue confronted by these approaches is scalability to large RDF graphs [AHKK17, Özs16].

2.3.3 Processing strategies

Processing SPARQL queries is a complex task in which a wide variety of approaches are available. The former systems were built on top of relational databases storing triples in tables as described in the previous section and directly translating SPARQL to SQL. Later, systems built specifically to treat RDF data came to light implementing its own physical storage and operators. More recently, parallel RDF systems follow the same strategy building these systems on top of cloud platforms transforming SPARQL queries into equivalent operators of the platform (e.g., MapReduce jobs). In general, RDF systems can be divided into 2 large groups: *native* and *non-native* whose characteristics are given below:

- *Native systems* are specifically built to treat RDF data (e.g., RDF-3X [NW08], gStore [ZÖC+14], TriAD [GSMT14]). These systems use custom physical layouts, native indexing strategies, efficient communication protocols and explicit replication. They could use relational-based storage layouts but their execution core is independent from a relational database. They require certainly a greater effort for development, yet they have proven to outperform the non-native strategies [AHKK17].

- *Non-native systems* are built on top of existent technologies like relational databases (e.g., Virtuoso [EM09]), key-value stores (e.g., Trinity.RDF [ZYW+13]), cloud platforms like MapReduce (e.g., EAGRE [ZCTW13], SHARD [RS11]) and Spark[14] (e.g., S2RDF [SPSL16]) among others. As it was previously mentioned, the query processing in these systems consists to transform a SPARQL query to the *known* operators of the parent system. For instance, SPARQL is translated to SQL or to MapReduce jobs if the system is built on top of a relational database or a Hadoop[15] distribution respectively.

Complete surveys on both type of systems exist. For example, [KM15] surveys all the systems built on top of different cloud platforms. An experimental comparison of both type of systems was performed in [AHKK17]. More general surveys in which the systems are further classified according to their storage, indexing strategy, and query execution are [Özs16, WHCS18, ASYNN20]).

[12] `https://www.stardog.com/`
[13] `https://rocksdb.org/`
[14] `https://spark.apache.org/`
[15] `https://hadoop.apache.org/`

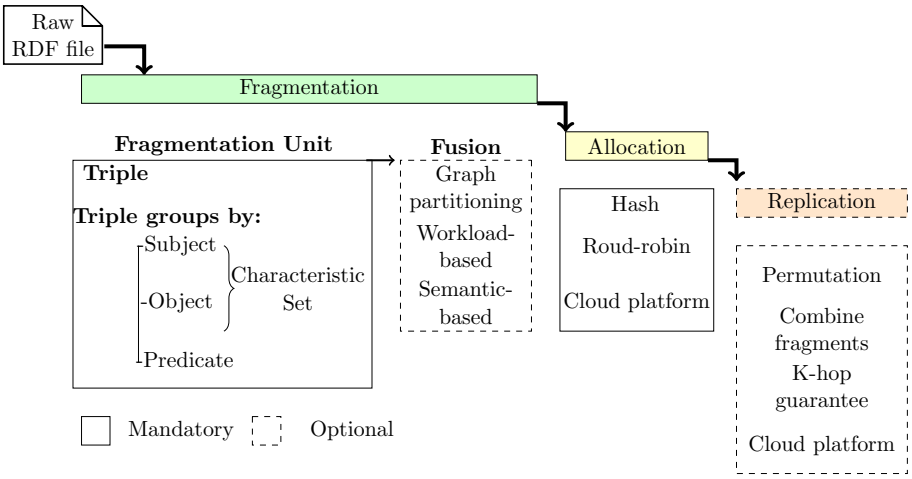


Figure 2.14: Partitioning Process in RDF Systems

2.3.4 Data partitioning

To cope with the increase in available RDF data and the need to process it efficiently, RDF systems resorted techniques of distribution and parallel processing extensively explored in relational databases. Within distributed RDF systems we find two groups, *federated* and *clustered* systems which are described as follows:

- *Federated systems* run SPARQL queries over multiple SPARQL endpoints. A SPARQL endpoint is a protocol service which allows to query data stored in RDF using the SPARQL language. Such systems require to perform data integration on-the-fly of multiple heterogeneous RDF sources. Federated systems are out of the scope of our study.
- *Clustered RDF systems* distribute the data among different data nodes that are part of a single RDF storage solution. For several years, a large number of distributed systems offering efficient RDF data processing have come to light. We focus our study in this type of systems.

In this section we classify these works according to the partitioning strategy applied to distribute the data. We present a scheme summarizing the process of data partitioning in RDF processing systems regardless of the execution model. We divide the process illustrated in Figure 2.14 in three phases: Fragmentation, Allocation and Replication. The dashed squares in the figure represent the optional phases of the process. We use this common framework inspired from the strategy used for distributed databases described in [ÖV11] to guide our classification. In what follows we detail the inputs, outputs and algorithms on each one of these stages.

2.3.4.1 Fragmentation

The fragmentation process consists in setting the fragmentation unit to be allocated over the sites of the distributed system. There is no consensus regarding how RDF data should be distributed. Most of current solutions use the triple as fragmentation unit. However, it has been claimed that grouping the triples first might improve the performance by discovering implicit structures in the input RDF graph [BDK+13, SGK+08]. Consequently, other solutions propose to group the triples by subject, predicate or object first and use these groups as fragmentation units. Other solutions go further and group the sets of triples (grouped by subject or object) in entities identified for instance with characteristic sets [PPEB15, NM11]. Regardless of how the data are physically represented by each system (e.g., triple table, property table), and of the execution engine (e.g., relational-based, subgraph matching), the fragmentation units shown in Figure 2.14 cover all the approaches of our state of the art.

Fusion This optional stage consists in creating broader groups of triples to be allocated over the sites of the distributed system. The fragmentation units are merged applying either of the following techniques:

- *Graph partitioning*: The fragmentation units are mapped to a graph to apply graph partitioning heuristics clustering the fragmentation units. The graph is built mapping the fragmentation units like triples (in H-RDF-3X [HAR11]), or groups of triples (entities in EAGRE [ZCTW13]) to nodes of the graph. The edges (which could be weighted) represent the connectivity within the fragmentation units. A partitioning graph heuristic, mostly METIS [KK98a] is used to generate the groups of fragmentation units.

- *Workload-based*: in these techniques the fragmentation units are grouped according to a given query workload. Triples that are queried together are gathered in the same fragments to optimize the query execution.

- *Semantic-based*: this approach gathers fragmentation units based on the semantic similarity (using ontologies for instance) of the concepts represented in the graph.

The fusion process is not applied in all the proposed systems and it is therefore an optional step (bounded by a dashed rectangle in Figure 2.14). Many systems choose to apply allocation strategies directly to the fragmentation units (e.g., triples, groups of triples by subject).

2.3.4.2 Allocation

The allocation stage consists in finding the distribution of a set of fragments $\{F_1, ..., F_n\}$ (which are the output of the previous stage) to a set of sites $\{S_1, ..., S_m\}$. The distribution of fragments is done applying the following techniques:

- *Hashing*: a hashing function is applied to the fragmentation unit or the fragment's attributes (e.g., gStoreD [PZÖ+16]).

- *Round-robin*: the fragments from last stage are placed on each site indistinctly. This is the case of the fragments obtained after a graph partitioning fusion.

- *Cloud-platform dependent*: in this case the allocation task is left to the cloud platform (e.g Hadoop, Spark) on which the RDF distributed system was built on top. Most of these platforms use distributed file systems, partitioning the data by blocks and finally sharding them to the worker nodes.

2.3.4.3 Replication

Data replication is a frequent yet not a mandatory optimization strategy applied by several clustered RDF distributed systems. The replication strategies are as follows:

- *Permutation*: in this strategy copies with different sort orders are maintained in systems representing the data as a single triple table. In general the copies are not exact replicas of the data, but encoded and compressed indexed versions. For example, RDF-3X [NW10] keeps 9 indexes (e.g., subject-predicate-object SPO, object-predicate-subject OPS) corresponding to the extended permutations of the triple table's attributes (subject, predicate object).

- *Combine fragments*: this strategy consists in gathering triples in different fragmentation units (e.g., grouping by subject, then grouping by object) and creating a copy of the data for each organization. This fragmentation strategy is embraced by the system RDF_QDAG [KMG+20]).

- *K-hop guarantee*: this strategy is used specifically in systems whose fragmentation units are merged with a graph partitioning strategy. For any vertex v assigned to a worker node f, all vertices up to k-hops away and their corresponding edges are replicated in f. Systems applying this strategy are H-RDF-3X [HAR11] and SHAPE [LL13].
- *Cloud platform*: this replication strategy takes place in systems built on top of a cloud platform, in this case the replication rules are established by the cloud platform. For example, the systems using the Hadoop Distributed File System HDFS replicate by default with a factor 3. Systems using HDFS as storage layer are Hadoop, Apache Spark or key-value stores like HBase[16]. Distributed RDF systems in this category are HadoopRDF [DWNY12], S2RDF [SPSL16], H2RDF[PKT+13] just to name a few.

Some efforts have also been made to build a system to benchmark partitioning strategies. The system Koral[JST17], for example, allows the integration of different RDF graph partitioning techniques to investigate their behavior.

In Table 2.3 we classify the systems of the state of the art based on the partitioning framework introduced previously. In the table we included approaches built on top of cloud platforms and also systems built specifically to deal with RDF data.

[16] `https://hbase.apache.org/`

Table 2.3: State of the art systems

		Fragmentation						**Allocation**			**Replication**			
		T	Group By			Fusion								
System	**Type**$^{\alpha,\beta}$		S	P	O	GP	WA	S	RR	H	CP	Perm	Comb	K-Hop	CP
Apache Rya [PCR15]	On top	✓									✓	✓			
AdPart [AAK+16]	Native	✓					✓		✓						
Clique-Square [GKM+15]	On top		✓	✓	✓								✓		✓
DiploCloud [WC16]	Native	✓						✓		✓				✓	
DREAM [HRN+15]	On top	✓								✓		✓			
EAGRE [ZCTW13]	On top		✓			✓				✓					
gStoreD [PZÖ+16]	Native		✓			✓				✓					
H-RDF-3X [HAR11]	On top	✓				✓				✓				✓	
H2RDF [PKT+13]	On top	✓									✓				✓
HadoopRDF [DWNY12]	On top			✓							✓				✓
JenaHbase [KKTC12]	On top			✓							✓				✓
Partout [GHS14]	On top	✓					✓			✓		✓			
PigSparql [SPL11]	On top	✓									✓				✓
S2RDF [SPSL16]	On top			✓							✓		✓		✓
S2X [SPBL15]	On top	✓									✓				✓
Sedge [YYZK12]	Native	✓											✓		
Sempala [SPNL14]	On top			✓							✓				✓
SHAPE [LL13]	On top	✓						✓		✓				✓	
SHARD [RS11]	On top		✓								✓				✓
PRoST [CFL18]	On top		✓	✓							✓				✓
SPARQLGX [GJGL16]	On top			✓							✓				✓
TriAd [GSMT14]	Native		✓		✓					✓		✓			
Trinity.RDF [ZYW+13]	On top		✓		✓	✓					✓				✓
WARP [HS13]	On top	✓					✓			✓				✓	

T: Triple, GP: Graph partitioning, W: Workload-based, SM: Semantic-based, RR: round-robin, H:Hash, CP: Cloud Platform
$^{\alpha}$Native: systems built specifically for RDF data, $^{\beta}$On top: systems built on top of cloud platforms.

2.4 Conclusion

In this chapter, we gave first a general overview of the graph data model. We detailed its logical and storage structures, described the query processing strategies and languages, and reported some of the most relevant current available systems. In addition, we dedicated a section to describe the graph partitioning algorithms used by some parallel systems. This section allowed us to understand the number of solutions available within systems merely treating data as graphs. We saw the advantages of the model but also its limits, especially due the variability of the methods used to partition and process the data. Next, to better delimit our work we focused on one of the many existent graph-based data models. Specifically on RDF, the data model for data interchange in the Web. We start by giving some background concepts, and similarly to what we did with the graphs in general, we described RDF storage, processing and partitioning strategies. In RDF, partitioning is dependent on the physical organization of the data (i.e. stored in a triple table, property table or graph structure). We showed that RDF partitioning strategies are conditional to a system and contrarily to relational databases, there is no common logical layer covering all existing partitioning variations. We say that RDF data partitioning is *system-driven*, in contrast to partitioning in relational databases that we named *data-driven*. In relational databases, there are tools (e.g., advisors, definition languages) giving comfort the database designer to create partitions according to the structure of the tables or a partitioning scheme that meets the demands of some workload. This is not the case in RDF, the data are charged into a system which, in general, automatically partitions the data with a predefined strategy. The process is similar to what happens in cloud platforms as described in last chapter, in which the data are split in blocks and hash distributed among the working nodes.

Table 2.4: Main strenghts relational and RDF partitioning

Relational	**RDF**
− Integration of data not following the same initial schema is complex.	+ Flexible integration of new data.
+ *DBMS-independent*: partitioning is independent of the database management system.	− *System-dependent*: the execution system and the partitioning strategy are coupled.
+ *Partitioning at a logical level*: Independent of the physical model used to store the data (e.g., NSM, DSM).	− Dependent on the physical storage model (e.g., triple-based, graph-based).
+ Native database definition languages support to declare horizontal partitions.	− Partitioning performed transparently during the loading stage.
+ Several automatic partitioning advisors are available help the designer to create partitions.	− Data partitioning is mostly imposed by the triple store.

The partitioning strategies adopted by RDF systems also have several advantages. For example, most approaches allow for simple data integration. Indeed, one of RDF assets is its flexibility to integrate data with different underlying schemas. The relational model, on the other hand, is often too rigid to handle data without giving an upfront schema. This contributes to tailor partitioning advising tools for the designer creating partitions based on the data structure and workload needs. Some approaches in RDF tried to model RDF data in a relational database schema before storing the data in a RDBMS. However, as it is demonstrated

by surveys, relational-based systems are not optimal when dealing with highly complex queries. The strengths and drawbacks (mostly mutually exclusive) of both data models are summarized in Table 2.4.

We consider necessary to provide tools for RDF system designers mimicking the comfort offered in relational database systems. This led us to fix the objective of this thesis which consists in proposing an RDF data partitioning framework that is based on implicit logical structures. The development of this framework will be covered throughout the next chapters.

Part II

Contributions

Chapter 3

Logical RDF Partitioning

Contents

Summary Based on the analysis of the literature presented in the first part of this thesis, we propose a partitioning environment for the design of opaque systems. We draw on the lessons learned in the relational model that partitions the data based on logical structures. In this chapter, we give the formal definition and detail the algorithms to create these logical entities that we named graph fragments ($\mathcal{G}f$). These entities harmonize with the notion of partitions by instances (horizontal) and by attributes (vertical) in the relational model. We start giving an overview of this logical layer in Section 3.2, comparing our strategy to the current state of the art. Then, we formally define the logical entities in Section 3.3. This section includes the algorithms to group RDF triples in graph fragments. Next, in Section 3.4 we discuss the usage of these fragments as physical storage structures. Section 3.5 formalizes the allocation problem of these fragments in opaque systems. Section 3.6 presents an complete example of the creation and allocation algorithms described in the previous sections. Then, Section 3.7 presents strategies to deal with large logical fragments. Finally, Section 3.8 concludes this chapter.

3.1 Introduction

The Resource Description Framework (RDF) has been widely accepted as the standard data model for data interchange in the Web. Formerly, RDF intended to model exclusively information in the Web but its flexibility allowed the standard to be used in other domains (e.g., genetics, biology). Currently, collections of RDF triples are extensive sources of information. The size of many RDF datasets can exceed more than a billion triples (e.g., DBPedia: 3 billion triples[1], Bio2RDF: 5 billion triples[2]) pushing the development of scalable processing systems. To this end, several systems rely on parallel and distributed architectures partitioning the data on distinct processing workers.

Most of the distributed and parallel RDF systems (RDFDS[3]) partition the data using the triples as distribution units and apply, for example, hashing functions sharding the triples at its finest granularity. These strategies do not allow to keep the graph structure of the model or triples belonging to the same entity together, impairing the system's performance. Furthermore, most systems are dependent on a single partitioning strategy.

We draw on the lessons learned in the relational model, in which data partitioning is performed at a logical level, independent of the storage approach. In the following we introduce a logical layer to the partitioning process of opaque RDFDS. We start giving an overview of this logical layer comparing our strategy to the current state of the art. Then, we formally define the logical entities and propose the algorithms to group RDF triples in graph fragments. Next, we discuss the usage of these fragments as physical storage structures and formalize their allocation problem. Finally we present a complete example of the creation and allocation heuristics defined in this chapter.

3.2 RDF partitioning design process

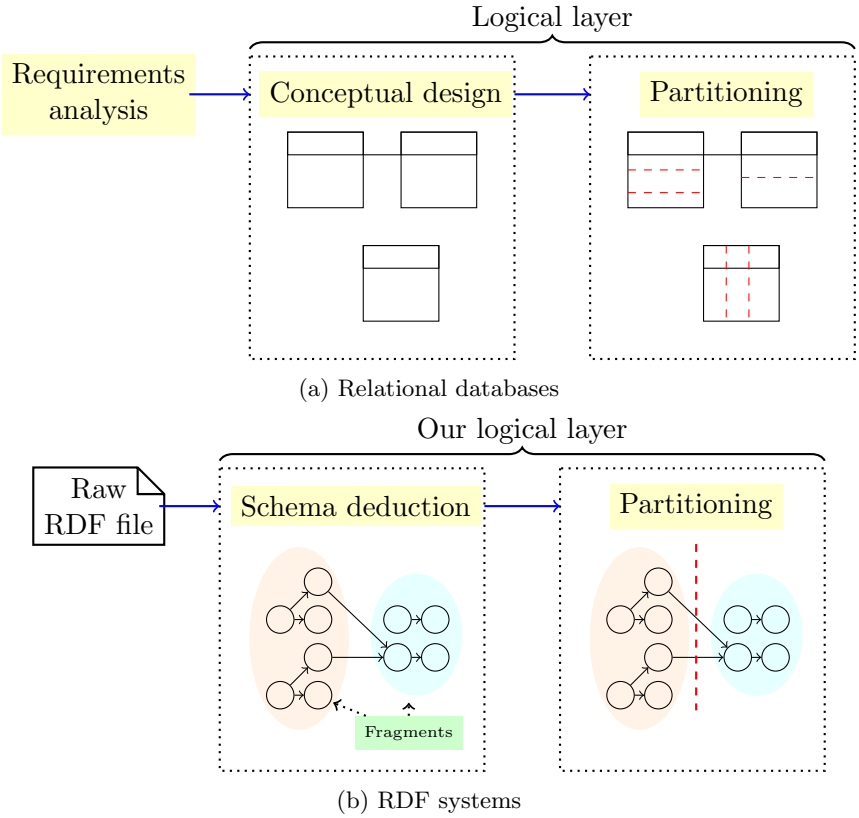
The design of a distributed system entails the distribution of data and programs to the sites of the computer network in which the system is deployed. We focus on the problem of data distribution seeking to find and allocate the optimal distribution units. In distributed relational databases, the design process starts similarly to centralized systems with a requirements analysis specifying the data and processing needs. The requirements are the input of the *conceptual design* stage that maps requirements to entities, attributes and relationships. The result of this stage is the global conceptual schema which is in turn the input of the *distribution design* step. This step determines the distribution units and their location. Entire tables could be used as distribution units, however it is preferable to use table subsets named partitions or fragments. To make this process more understandable, the distribution design is subdivided in [ÖV11] in two sub steps. The first one, named fragmentation, determines the distribution units and then the allocation step places them to the sites of the network. The previous process is summarized in Figure 3.1a.

The conceptual design phase explicitly declaring the high-level entities like in relational databases does not occur in RDF systems. Instead, the data are organized in a schema-free model using triples whose global structure is not explicitly stated. This characteristic gives RDF more flexibility to integrate data with different schemas but it pays the price during the physical design [AÖD14]. Furthermore, the data related to the same high-level entity can be easily scattered through the batch of data. When it comes to the distribution of RDF data, there is no consensus about how the data should be fragmented and allocated. Currently, the partitioning strategies are highly dependent on the backend system used to store the data. For example, the systems built on top of cloud platforms delegate the partitioning choice to the distributed file system which usually hashes the data by chunks. Specialized systems use triples

[1] `https://wiki.dbpedia.org/about/facts-figures`

[2] `https://www.lod-cloud.net/dataset/bio2rdf-pubmed`

[3] From this point we use the abbreviation RDFDS to denote both, parallel and distributed RDF systems



(a) Relational databases

(b) RDF systems

Figure 3.1: Partitioning design process

as distribution units and usually apply hashing functions to decide the triple's location.

There have been some efforts to explicitly add a class hierarchy schema to RDF through annotations (e.g., RDF schema and ontologies). Furthermore, as shown in [PPEB15], i) the entities in a single dataset can be described with multiple ontologies, ii) not all the entities in a dataset are annotated with the same metadata and, iii) not all SPARQL query patterns consider them. Using these annotations in the identification of implicit entities is not very effective.

The identification of higher-level entities in RDF has also been researched in the context of query optimization. Specifically, to collect statistics to estimate the cardinality of triple patterns. Instead of collecting statistics on the entire dataset, approaches like the characteristic set [NM11] for instance, collect statistics on higher-level entities. In [NM11], these entities are detected using the structure of the original dataset grouping the triples by subject first and further gathering the data according to their predicates. This strategy does not rely on the annotations (e.g RDFS) to identify entities and it allows to consider some of correlations missed by the other optimization approaches based global statistics. This technique has been reused to detect a relational schema to the data as we will see below.

Grouping the triples in higher level entities has also been explored in relational-based RDF systems. The goal of these approaches is to build a relational schema and map the triples to multiple relational tables. For example, the *hidden schemas* [CBN07] introduced in DB2RDF inferred an RDF schema to vertically partition a property table via attribute clutering. The *emergent schemas* approach in [PPEB15] combines the characteristic set with other clustering techniques to identify implicit structures within the data. More recently, [HVL14] introduced *Cinderella* to incrementally partition bulk data inserts assigning a schema with a similarity measure between attributes. The techniques described previously could be generalized in a framework allowing the discovery of implicit entities used by optimization strategies (like indexes) or as data fragments.

Our work introduces a logical dimension to the partitioning process of RDF graphs. We

draw on the lessons learned in the relational model, in which data partitioning is performed at a logical level, independently of the storage approach used to persist the data. This *wise* strategy has proven its success due to its great balance between conceptual and intuitive simplicity [OB19]. The proposed process is illustrated in Figure 3.1b. We borrow some of the techniques in the literature used to identify higher-level entities in RDF datasets and define logical RDF fragments. Then, based on these fragments we propose allocation strategies considering the initial connectivity of the data, keeping the original graph structure and avoiding data skewness.

3.3 Graph fragments

Many studies have shown that grouping triples contributes to improve the performance of centralized (e.g., [CBN07, PPEB15]) and distributed (e.g., [AHKK17]) RDF systems. In this section we propose to cluster the data in groups inspired from the partitions in the relational model. In relational databases, an entity (a table) is partitioned at the instance (horizontal) or attribute level (vertical). Indeed, horizontal and vertical partitions let to retrieve only relation's subsets avoiding full table scans. Considering that in RDF there is no conceptual design stage like in relational databases, our proposal looks to detect implicit entities based on the structure of the RDF graph. Such entities represent the implicit schema of an RDF graph

In fact, the characteristic set (defined in Definition 2.7) of a node in an RDF graph allows to group the instances of the same high-level entity to form a partition. In the following section we define two kinds of entities, following the idea of fragmenting the entities at the instance or attribute levels, but with an actual graph connotation.

3.3.1 Grouping the graph by instances

To gather the triples of the same high-level entity we group in the first place the triples by its subjects. These structures defined in Definition 3.1 are named *forward data stars*. Let us first define the functions $f_s(t) \rightarrow s$, $f_p(t) \rightarrow p$ and $f_o(t) \rightarrow o$ returning the subject, predicate and object of an RDF triple $t = \langle s, p, o \rangle$ respectively. Both functions are applied in some of the definitions of this chapter.

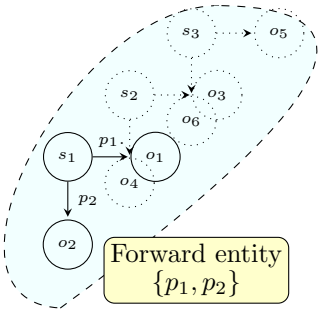
Definition 3.1 ***(Forward graph star)*** *A forward graph star denoted as $\overrightarrow{\mathcal{G}s}(s)$ is a subset of G in which the triples share the same subject s. Formally, $\overrightarrow{\mathcal{G}s}(s) \subseteq G$, such that $\overrightarrow{\mathcal{G}s}(s) = \{t | \forall_{i \neq j}(f_s(t_i) = f_s(t_j) = s)\}$. We name the subject s the head of the forward star.*

A forward graph star gathers all the properties associated with a node describing a single occurrence of a certain entity. As shown in Theorem 3.1, a triple can only belong to a single forward graph star. The entities can be uniquely identified by a subset of their emitting edges [NM11]. To characterize them we use the notion of characteristic set defined in Definition 2.7.

Theorem 3.1 *A triple belongs to one and only one forward graph star. Let t be a triple $t \in G$, if $t \in \overrightarrow{\mathcal{G}s}_i \Rightarrow \forall_{i \neq j}(t \notin \overrightarrow{\mathcal{G}s}_j)$.*

Proof. *To prove by contradiction. Let us assume that a triple $t \in \overrightarrow{\mathcal{G}s}_i$ and $t \in \overrightarrow{\mathcal{G}s}_j$. Let us consider two not equal triples t_1, t_2, both different from t, such that $t_1 \in \overrightarrow{\mathcal{G}s}_i$ and $t_2 \in \overrightarrow{\mathcal{G}s}_j$. Based on the forward entities definition, $f_s(t_1) = f_s(t)$ and $f_s(t_2) = f_s(t)$. Using the transitivity of the equality, $f_s(t_1) = f_s(t_2)$ which is impossible if $\overrightarrow{\mathcal{G}s}_i \neq \overrightarrow{\mathcal{G}s}_j$.*

A *forward graph fragment* formally described in Definition 3.2 gathers forward graph stars according to their characteristic set. As it was previously mentioned, we can characterize an entity by its emitting edges. A forward graph fragment is a *logical* structure used to gather the



(a) Graph

s	p_1	p_2
s_1	o_1	o_2
s_2	o_3	o_4
s_3	o_5	o_6

(b) Table

```
{ {s1 : {{p1:o1}, {p2:o2}}},
  {s2 : {{p1:o3}, {p2:o4}}},
  {s3 : {{p1:o5}, {p2:o6}}}
}
```

(c) Adjacency list

Figure 3.2: Forward graph fragment example

data related to the same high-level entity. As shown in Figure 3.2, this structure is independent of the storage layer of the systems. A forward graph fragment can be used only as a logical structure to distribute the data, or used to physically store the data in this format as shown in Figures 3.2b and 3.2c.

In general, two forward graph stars belong to the same entity if the labels of their emitting edges (i.e. the predicates) are the same. Yet, sometimes two instances belong to the same entity but their characteristic sets are not exactly the same. This is the case when two characteristic sets differ only in non-discriminating predicates. To avoid creating a great number of fragments with a very strict similarity criteria, we used a similarity score and a threshold. The similarity scores are described in Sect. 3.3.1.1. The characteristic set of a graph fragment is therefore the union of the characteristic sets of the graph stars of the fragments. To simplify the notation, the function $cs(\overrightarrow{\mathcal{G}f})$ returns the characteristic set (which is the identifier) of the fragment.

Definition 3.2 ***(Forward graph fragment)*** *A forward graph fragment is a set of forward graph stars $\overrightarrow{\mathcal{G}s}$ that have similar characteristic sets according to a threshold τ. A forward graph star $\overrightarrow{\mathcal{G}s}_i$ belongs to one and only one forward graph fragment $\overrightarrow{\mathcal{G}f}_i$.*

Formally, $\overrightarrow{\mathcal{G}f} = \{\overrightarrow{\mathcal{G}s(s)} | \forall_{\overrightarrow{\mathcal{G}s}(s_i) \neq \overrightarrow{\mathcal{G}s}(s_j)} (Sim(cs(s_i), cs(s_j)) \geq \tau), \}$.

3.3.1.1 Similarity scores

The similarity function $Sim(cs(s_i), cs(s_j))$ returns a score measuring how similar both characteristic sets are. In its calculation, it can consider just the structure of the characteristic set and their relationships or more complex semantic features. Among the structural similarity functions we find:

- *Supersets:* this similarity strategy merges two characteristic sets if there is a larger set that contains all the properties of the smaller one. It is calculated as follows:

$$Sim(cs(s_i), cs(s_j)) = \frac{|cs(s_i) \cap cs(s_j)|}{Min(|cs(s_i)|, |cs(s_j)|)}$$

  To ensure that only full subsets are merged with supersets, the threshold τ is fixed to 1.

- *Jaccard similarity:* this measure evaluates only how many predicates both sets have in common. Contrarily to the previous function, it considers not only the superset case.

$$Sim(cs(s_i), cs(s_j)) = \frac{|cs(s_i) \cap cs(s_j)|}{Max(|cs(s_i)|, |cs(s_j)|)}$$

- *Tf-idf* [PPEB15]: this criteria used in Information Retrieval expresses the frequency of a term taking into account how often the term is used in the corpus. The measure is

interesting to compare characteristic sets since it could be adapted to weight the predicates according to their discriminatory power, giving less weight to predicates shared by almost all the characteristic sets of the dataset. The *tf-idf* measure of a predicate in a characteristic set is calculated as follows:

$$tf\_idf(p, cs) = \frac{1}{|p|} \cdot log\Big(\frac{\#TotalCSs}{1 + \#countCSs(p)}\Big)$$

Where the tf factor is always 1 since every property occurs once per characteristic set, $\#TotalCSs$ is the total number of characteristic sets, and $\#countCSs(p)$ is the total number of characteristic sets having property p in the property list. The measure can be normalized to take into account the length of the characteristic set:

$$tf\_idf(p, cs) = \Big(0.5 + 0.5 * \frac{1}{|cs|}\Big) \cdot log\Big(\frac{\#TotalCSs}{1 + \#countCSs(p)}\Big)$$

Finally, the tf-idf values for each predicate are used to calculate cosine similarity as:

$$Sim(cs_i, cs_j) = \frac{\sum_{p \in (cs_i \cap cs_j)} tf\_idf(p, cs_i) \times tf\_idf(p, cs_j)}{\sqrt{\sum_{p_i \in cs_i} tf_i df(p_i, cs_i)^2} \times \sqrt{\sum_{p_j \in cs_j} tf_i df(p_j, cs_j)^2}}$$

On the other hand, if the similarity is based on semantic measures there are two alternatives:

- *Same label:* this strategy merges two characteristic sets if their semantic label (identified by the predicate `rdfs:label`) is the same. This similarity is binary, if both characteristic sets have the same label the similarity is 1 and 0 otherwise.
- *Ancestor-based:* This strategy checks at the hierarchy of classes in the ontology and assigns a similarity 1 if both sets share a common ancestor. However, the similarity score weights whether the common ancestor is too general in the ontology. Given a common ancestor, the similarity is calculated as follows:

$$Sim(cs_i, cs_j) = log\Big(\frac{\#TotalInstO}{\#InstC}\Big)$$

Where $\#TotalInstO$ is the total number of instances covered by the ontology and $\#InstC$ is the total number of instances covered by the ancestor.

Both of the previous similarities are calculated only if a common ontology exists between both cs, otherwise the structural similarity is preferred.

To assign a forward graph star $\overrightarrow{\mathcal{G}s}(s)$ to a forward graph fragment, a pairwise similarity is calculated between the characteristic sets of the forward star and the fragments already assigned to $\mathcal{G}f$. The graph star is assigned to the fragment with the highest similarity only if its similarity score is higher than the given threshold τ. Let us consider for example a forward graph star $\overrightarrow{\mathcal{G}s}(s_i)$ and the set of forward fragments $\overrightarrow{\mathcal{G}f} = \{\overrightarrow{\mathcal{G}f_j}, \overrightarrow{\mathcal{G}f_k}\}$. To calculate the similarity one can encounter the following cases:

(i) If $Sim(cs(s_i), cs(\overrightarrow{\mathcal{G}f_j})) > Sim(cs(s_i), cs(\overrightarrow{\mathcal{G}f_k})) > \tau$ then $\mathcal{G}s_i \in \overrightarrow{\mathcal{G}f_j}$.

(ii) If both scores have the same value, $Sim(cs(s_i), cs(\overrightarrow{\mathcal{G}f_j})) = Sim(cs(s_i), cs(\overrightarrow{\mathcal{G}f_k})) > \tau$, then $\overrightarrow{\mathcal{G}s_i}$ is assigned to either of the sets $\overrightarrow{\mathcal{G}f_j}$ or $\overrightarrow{\mathcal{G}f_k}$.

(iii) If the forward graph star has not a similarity score greater than the threshold with any of the graph fragments, formally expressed as:

$$\forall_{i \neq j}\Big(Sim(cs(s_i), cs(\overrightarrow{\mathcal{G}f_j}))\Big) < \tau)$$

then the the forward star $\overrightarrow{\mathcal{G}s}$ forms a forward graph fragment on its own ($\overrightarrow{\mathcal{G}f} = \overrightarrow{\mathcal{G}s_i}$).

3.3.1.2 Forward fragmentation algorithm

The generation of forward graph fragments is described in Algorithm 1. The inputs of the algorithm are an RDF file in SPO order and a similarity threshold τ. We consider an ordered file since many triple stores employ bulk loading techniques that efficiently encode and organize the data. The algorithm starts initializing an empty graph star, a variable storing the subject of the first triple and an empty map structure mapping characteristic sets to graph fragments. Then it makes a single pass through the triples of the dataset (Step 2). The loop creates a graph star set until it detects a change in the subject of the current triple (Step 3). At this point, it generates a key combining the set of ID's of the predicates that co-occur in a graph star with a hashing function (Step 6). If the key is already in the map of characteristic sets, it assigns the star directly to the fragment and appends it to the set of graph fragments with the function *appendGf* (Step 9). If there is no key in the map with the star's key then it uses the function *updateGf* which applies the similarity functions to decide to which fragment the star should be appended (Step 11). The functions *appendGf* and *updateGf* are described in Algorithms 2 and 3 respectively. The algorithm returns the set of graph fragments, the map of characteristic sets and fragments, and a map assigning a subject with the corresponding graph fragment (Step 18).

Algorithm 1 Generation of forward graph fragments

Input: RDF file D of $\{t_1, ..., t_n\}$ triples $t = \langle s, p, o\rangle$ in SPO order, threshold similarity τ
Output: Set of fragments $\overrightarrow{\mathcal{GF}} = \{\overrightarrow{\mathcal{GF}}_1, ..., \overrightarrow{\mathcal{GF}}_p\}$, characteristic set map $CSM\langle key, \overrightarrow{\mathcal{GF}}_k\rangle$, subject map $SM\langle s_{i-1}, \overrightarrow{\mathcal{GF}}_k\rangle$

1: **Initialize** $\overrightarrow{\mathcal{G}s} = \emptyset$, $s_{i-1} = f_s(t_1)$, $CSM\langle key, \overrightarrow{\mathcal{GF}}_k\rangle = \emptyset$,$SM\langle s_{i-1}, \overrightarrow{\mathcal{GF}}_k\rangle = \emptyset$
2: **for each** t_i in D **do**
3: **if** $s_{i-1} = f_s(t_i)$ **then**
4: $\overrightarrow{\mathcal{G}s}(s_{i-1}) = \overrightarrow{\mathcal{G}s}(s_{i-1}) \cup \{t_i\}$
5: **else**
6: $key = \mathcal{H}ash(cs(s_{i-1}))$
7: **if** $key \in CSM.keySet()$ **then**
8: $\overrightarrow{\mathcal{GF}}_k = CSM.get(key)$
9: $\overrightarrow{\mathcal{GF}} = appendGf(\overrightarrow{\mathcal{G}s}(s_{i-1}), \overrightarrow{\mathcal{GF}}_k, \overrightarrow{\mathcal{GF}})$
10: **else**
11: $(\overrightarrow{\mathcal{GF}}, CSM, \overrightarrow{\mathcal{GF}}_k) = updateGf(\overrightarrow{\mathcal{G}s}(s_{i-1}), \overrightarrow{\mathcal{GF}}, \tau, CSM)$
12: **end if**
13: $\overrightarrow{\mathcal{G}s}(s_{i-1}) = \{t_i\}$
14: $SM.put(s_{i-1}, \overrightarrow{\mathcal{GF}}_k)$
15: **end if**
16: $s_{i-1} = f_s(t_i)$
17: **end for**
18: **return** $\overrightarrow{\mathcal{GF}}$, CSM,SM

The *appendGf* (Alg.2) function takes a given graph star, and appends it to the indicated graph fragment (Step 1). Finally it updates the global set of graph fragments (Steps 2 - 3). The *updateGf* function (Alg. 3) uses the similarity functions described in Section 3.3.1.1 to decide whether to append the graph star to a given fragment or to create a new one. The algorithm considers the case in which two similarity functions are prioritized. For example, using a semantic similarity (which is a binary similarity) first and if it does not find any match, apply a structural similarity. The algorithm makes a single pass through the already assigned graph fragments (Step 2). It uses the first similarity measure and, if it finds a graph fragment whose similarity is 1, it assigns the graph star to the fragment and stops the search (Steps 3 - 5). The second similarity measure is used the case in which the first measure did not find any match

(Steps 7 - 9). If after going through all the assigned fragments it still does not find any match, it creates a new graph fragment (Steps 13 - 15). It returns the set of updated graph fragments, the updated characteristic set map and the fragment assigned to the graph star (Step 20).

Algorithm 2 *appendGf* function

Input: A graph star $\overrightarrow{\mathcal{G}s}(s)$, a graph fragment $\overrightarrow{\mathcal{G}f}_k \in \overrightarrow{\mathcal{G}f}$, a set of graph fragments $\overrightarrow{\mathcal{G}f}$
Output: Updated set of graph fragments $\overrightarrow{\mathcal{G}f}$
1: $\overrightarrow{\mathcal{G}f}_k = \overrightarrow{\mathcal{G}f}_k \cup \overrightarrow{\mathcal{G}s}(s)$
2: $\overrightarrow{\mathcal{G}f} = \overrightarrow{\mathcal{G}f} - \overrightarrow{\mathcal{G}f}_k$
3: $\overrightarrow{\mathcal{G}f} = \overrightarrow{\mathcal{G}f} \cup \overrightarrow{\mathcal{G}f}_k$
4: **return** $\overrightarrow{\mathcal{G}f}$

Algorithm 3 *updateGf* function

Input: A graph star $\overrightarrow{\mathcal{G}s}(s)$, a set of graph fragments $\overrightarrow{\mathcal{G}f}$, a similarity threshold τ, Map of graph fragment keys $CSM\langle key, \overrightarrow{\mathcal{G}f}_k\rangle$
Output: ($\overrightarrow{\mathcal{G}f}$, $CSM\langle key, \overrightarrow{\mathcal{G}f}_k\rangle$,$\overrightarrow{\mathcal{G}f}_{tmp}$)
1: **Initialize** $max = 0, \overrightarrow{\mathcal{G}f}_{tmp} = \emptyset$, $key = \mathcal{H}ash(cs(s_i))$
2: **for each** $\overrightarrow{\mathcal{G}f}_i$ in $\overrightarrow{\mathcal{G}f}$ **do**
3: **if** $Sim_A(cs(s), cs(\overrightarrow{\mathcal{G}f}_i)) = 1$ **then**
4: $\overrightarrow{\mathcal{G}f}_{tmp} = \overrightarrow{\mathcal{G}f}_i$
5: **break**
6: **else**
7: **if** $Sim_B(cs(s), cs(\overrightarrow{\mathcal{G}f}_i)) > \tau$ AND $Sim_B(cs(s), cs(\overrightarrow{\mathcal{G}f}_i)) > max$ **then**
8: $max = Sim_B(cs(s), cs(\overrightarrow{\mathcal{G}f}_i))$
9: $\overrightarrow{\mathcal{G}f}_{tmp} = \overrightarrow{\mathcal{G}f}_i$
10: **end if**
11: **end if**
12: **end for**
13: **if** $\overrightarrow{\mathcal{G}f}_{tmp} = \emptyset$ **then**
14: $\overrightarrow{\mathcal{G}f}_{tmp} = \overrightarrow{\mathcal{G}}\, s(s)$
15: $\overrightarrow{\mathcal{G}f} = \overrightarrow{\mathcal{G}f} \cup \overrightarrow{\mathcal{G}f}_{tmp}$
16: **else**
17: $\overrightarrow{\mathcal{G}f} = appendGf(\overrightarrow{\mathcal{G}s}(s), \overrightarrow{\mathcal{G}f}_{tmp}, \overrightarrow{\mathcal{G}f})$
18: **end if**
19: $CSM.put(key, \overrightarrow{\mathcal{G}f}_{tmp})$
20: **return** $(\overrightarrow{\mathcal{G}f}, CSM, \overrightarrow{\mathcal{G}f}_{tmp})$

3.3.1.3 Forward fragments as allocation units

In this section we prove that the set of forward graph fragments $\overrightarrow{\mathcal{G}f}$ partitions an RDF graph in a *correct* set of partitions. This is depicted in Theorem 3.2.

Theorem 3.2 *The set* $\overrightarrow{\mathcal{G}f} = \{\overrightarrow{\mathcal{G}f}_1, ..., \overrightarrow{\mathcal{G}f}_l\}$ *of all forward graph fragments for the graph G is a correct*[4] *partition set of the graph G.*

[4]According to the correctness fragmentation rules in [ÖV11]

Proof. *We will show that the three correctness fragmentation rules mentioned in [ÖV11] are enforced.*

- *Completeness: If* $t \in G \Rightarrow \exists \overrightarrow{\mathcal{G}s_i}$ *such that* $t \in \overrightarrow{\mathcal{G}s_i}$, $\overrightarrow{\mathcal{G}s_i} \in \overrightarrow{\mathcal{G}f_i}$, *and* $\overrightarrow{\mathcal{G}f_i} \in \overrightarrow{C}$. *By contradiction, if* $\forall \overrightarrow{\mathcal{G}s_i}, t \notin \overrightarrow{\mathcal{G}s_i}$ *then the triple's t subject* $f_s(t)$ *does not equal any of the subjects of the forward stars* $\overrightarrow{\mathcal{G}s}$. *Since the forward stars* $\overrightarrow{\mathcal{G}s}$ *are built by grouping first all triples of the graph G by subject, the triple t's subject is not equal to the subject of any triple in G, therefore* $t \notin G$.

- *Reconstruction: It is possible to define an operator* ∇ *such that* $G = \nabla \overrightarrow{\mathcal{G}f_i}, \forall \overrightarrow{\mathcal{G}f_i} \in \overrightarrow{C}$. *For the forward fragment classes, the operator* ∇ *equals the union operator* $\cup$. *In other words,* $\bigcup_{i=1}^{l} \overrightarrow{\mathcal{G}f_i} = G$. *By contradiction, if* $\exists t \in \overrightarrow{\mathcal{G}f_i}$ *such that* $t \notin G$, *then the subject of any triple in* $\overrightarrow{\mathcal{G}f_i}$ *does not belong to G. This is impossible because by definition, the forward stars are created grouping all triples from the initial graph G by subject. This would be possible only if* $t \notin \overrightarrow{\mathcal{G}f_i}$ *and therefore the set of forward entities would not be complete.*

- *Disjointness:* $\forall_{i \neq j}(\overrightarrow{\mathcal{G}f_i} \cap \overrightarrow{\mathcal{G}f_j} = \emptyset)$. *By contradiction, if* $\exists(\overrightarrow{\mathcal{G}f_i} \cap \overrightarrow{\mathcal{G}f_j} = \{t\})$ *then* $t \in \overrightarrow{\mathcal{G}s_i}$ *and* $t \in \overrightarrow{\mathcal{G}s_j}$ $(\overrightarrow{E_i} \in \overrightarrow{\mathcal{G}f_i}, \overrightarrow{\mathcal{G}s_j} \in \overrightarrow{\mathcal{G}f_j})$, *which is impossible unless the same triple had two different subjects.*

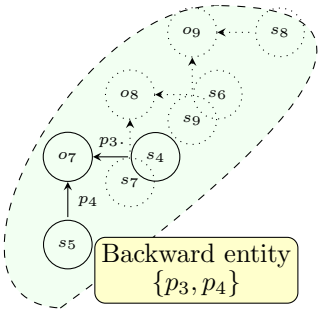
3.3.2 Grouping the graph by attributes

The organization of an RDF graph in forward graph fragments identifies the set of instances for implicit entities in the graph using characteristic sets. The fragments created with this strategy resemble the tuple groups generated when horizontally partitioning a relational database. As in the relational model, fragmenting a graph in forward fragments is useful for a certain type of queries. For example, if the workload is composed of star-shaped queries, this type of organization is ideal. However, this single fragmentation strategy does not optimize all query spectra. Partitioning an entity by its attributes in the relational model, known as vertical partitioning, optimizes other types of queries that involve a small number of attributes. Several RDF processing systems use a similar strategy, where a fragment is created to each property (e.g., SW-Store [AMMH09], S2RDF [SPSL16]). In relational-based systems, the strategy stores triples in different tables per property. This strategy is efficient when solving queries with a few attributes but suffers overheads when several predicates are joined in one query.

In this section we describe the creation of fragments by regrouping the nodes by their properties. Instead of strictly regrouping each property in a different fragment, we use the notion of characteristic sets to group the attributes affecting the same node. We name these structures *backward graph fragments*, whose construction is very similar to the fragments described in the previous section. We start defining a *backward graph star* which groups a node and its *incoming* edges. This structure allows to identify properties that affect the same node to later cluster them. It is formally defined in Definition 3.3. A triple belong to a single backward graph star as shown in Theorem 3.3.

Definition 3.3 ***(Backward graph star)*** *A backward graph star denoted as* $\overleftarrow{\mathcal{G}s}(o)$ *is a subset of the original RDF graph G in which the triples share the same object o. Formally* $\overleftarrow{\mathcal{G}s} \subseteq G$ *such that* $\overleftarrow{\mathcal{G}s} = \{t | \forall_{i \neq j}(f_o(t_i) = f_o(t_j))\}$. *We name the object o the head of the backward star.*

Theorem 3.3 *A triple belongs to one and only one backward graph star. Let t be a triple* $t \in G$, *if* $t \in \overleftarrow{\mathcal{G}s_i} \Rightarrow \forall_{i \neq j}(t \notin \overleftarrow{\mathcal{G}s_j})$.



(a) Graph

o	p_3	p_4
o_7	s_4	s_5
o_8	s_6	s_7
o_9	s_8	s_9

(b) Table

```
{ {o7 : {{p3:s4}, {p4:s5}}},
  {o8 : {{p3:s6}, {p4:s7}}},
  {o9 : {{p3:s8}, {p4:s9}}}
}
```

(c) Adjacency list

Figure 3.3: Backward graph fragment example

Proof *To prove by contradiction. Let us assume that a triple* $t \in \overleftarrow{\mathcal{G}}\, s_i$ *and* $t \in \overleftarrow{\mathcal{G}}\, fs_j$*. Let us consider two not equal triples* t_1, t_2*, both different from* t*, such that* $t_1 \in \overleftarrow{\mathcal{G}}\, s_i$ *and* $t_2 \in \overleftarrow{\mathcal{G}}\, s_j$ *. Based on the backward entity definition,* $f_o(t_1) = f_o(t)$ *and* $f_o(t_2) = f_o(t)$*. Using the transitivity property of the equality,* $f_o(t_1) = f_o(t_2)$ *which is impossible if* $\overleftarrow{\mathcal{G}}\, s_i \neq \overleftarrow{\mathcal{G}}\, s_j$*.*

The backward graphs stars are grouped using characteristic sets. We extend in Definition 3.4 the definition given in Chapter 2 to consider the characteristic set of objects. A *backward graph fragment* is therefore a group of backward graph stars whose heads (objects) share a similar characteristic set. This structure allows to gather the predicates that point to the same type of objects, creating broader groups than the ones obtained when splitting the dataset by predicates. We can retrieve the characteristic set of a backward fragment with the function $cs(\overleftarrow{\mathcal{G}f})$.

The backward graph fragments are illustrated in Figure 3.3a. Similarly, they could be physically stored in tables (Figure 3.3b) or adjacency lists (Figure 3.3c). They are formally defined in Definition 3.5.

Definition 3.4 ***(Characteristic set extension)*** *Each subject* s *and object* o *in an RDF graph* G *has a characteristic set defined as* $\overrightarrow{cs}(s) = \{p|\exists_o : (s,p,o) \in G\}$ *and* $\overrightarrow{cs}(o) = \{p|\exists_o : (s,p,o) \in G\}$*.*

Definition 3.5 ***(Backward graph fragment)*** *A backward segment* $\overleftarrow{\mathcal{G}f}$ *is a set of backward graph stars* $\overleftarrow{\mathcal{G}s}$ *with similar characteristic sets according to a threshold* τ*.*

Formally, $\overleftarrow{\mathcal{G}f} = \{\overleftarrow{\mathcal{G}s}(o)|\forall_{\overleftarrow{\mathcal{G}s}(s_i) \neq \overleftarrow{\mathcal{G}s}(s_j)} Sim(cs(o_i), cs(o_j\})) \geq \tau\}$*.*

The similarity functions used to group characteristic sets are the same functions described in Section 3.3.1.1, although structural functions are mainly used to form backward fragments.

3.3.2.1 Backward fragmentation algorithm

The generation of backward graph fragments is described in Algorithm 4. The inputs of the algorithm are the same as for Alg. 1 except that the input file must be in OPS *(object, predicate, subject)* order. This input file facilitates the creation of backward stars and fragments. The algorithm starts initializing an empty graph star, a variable storing the object of the first triple and an empty map structure mapping characteristic sets to graph fragments. Then it makes a single pass through the triples of the dataset (Step 2). The loop creates a graph star set until it detects a change in the object of the current triple (Step 3). At this point, it generates a key combining the set of ID's of the predicates that co-occur in a graph star with a hashing function (Step 6). If the key is already in the map of characteristic sets, it assigns the star directly to the fragment and appends it to the set of graph fragments with the function *appendGf* (Step 9). If there is no key in the map with the star's key then it uses the function *updateGf* which

applies the similarity function to decide to which fragment the star should be appended (Step 11). The function *appendGf* is not detailed in an algorithm, since the steps are the same as in 2 but for backward fragments. The function *updateGb* is described in Algorithm 3. The algorithm returns the set of graph fragments, the map of characteristic sets and graph fragments, and a map of objects with their corresponding backward fragment (Step 11).

Algorithm 4 Generation of backward graph fragments

Input: RDF file D of $\{t_1, ..., t_n\}$ triples $t = \langle s, p, o\rangle$ in OPS order, threshold similarity τ
Output: Set of fragments $\overleftarrow{\mathcal{G}f} = \{\overleftarrow{\mathcal{G}f}_1, ..., \overleftarrow{\mathcal{G}f}_p\}$,characteristic set map $CSM\langle key, \overleftarrow{\mathcal{G}f}_k\rangle$, object map $OM\langle o, \overleftarrow{\mathcal{G}f}\rangle$

1: **Initialize** $\overleftarrow{\mathcal{G}s} = \emptyset$, $o_{i-1} = f_o(t_1)$, $CSM\langle key, \overleftarrow{\mathcal{G}f}_k\rangle = \emptyset$
2: **for each** t_i in D **do**
3: **if** $s_{i-1} = f_s(t_i)$ **then**
4: $\overleftarrow{\mathcal{G}s}(o_{i-1}) = \overleftarrow{\mathcal{G}s}(o_{i-1}) \cup \{t_i\}$
5: **else**
6: $key = \mathcal{H}ash(cs(o_{i-1}))$
7: **if** $key \in CSM.keySet()$ **then**
8: $\overleftarrow{\mathcal{G}f}_k = CSM.get(key)$
9: $\overleftarrow{\mathcal{G}f} = appendGf(\overleftarrow{\mathcal{G}s}(o_{i-1}), \overleftarrow{\mathcal{G}f}_k, \overleftarrow{\mathcal{G}f})$
10: **else**
11: $(\overleftarrow{\mathcal{G}f}, Map, \overleftarrow{\mathcal{G}f}_k) = updateGb(\overleftarrow{\mathcal{G}s}(o_{i-1}), \overleftarrow{\mathcal{G}f}, \tau, CSM)$
12: **end if**
13: $\overleftarrow{\mathcal{G}s}(o_{i-1}) = \{t_i\}$
14: $OM.put(o_{i-1}, \overleftarrow{\mathcal{G}f}_k)$
15: **end if**
16: $o_{i-1} = f_o(t_i)$
17: **end for**
18: **return** $\overleftarrow{\mathcal{G}f}$, CSM, OM

The *updateGf* function is described in Alg. 5. The algorithm makes a single pass through the already assigned graph fragments (Step 2). Based on a similarity measure in the characteristic set, it searches for the most similar already assigned graph fragment(Steps 3 - 5). If after going through all the assigned fragments it still does not find any match, it creates a new graph fragment (Steps 8 - 10). It returns the updated set of backward fragments, the updated characteristic set map and the graph fragment assigned to the input graph star (Step 15).

The set of backward graph fragments forms also a partitioning set of the RDF graph as shown in Theorem 3.4.

Theorem 3.4 *The set* $\overleftarrow{\mathcal{G}f} = \{\overleftarrow{\mathcal{G}f}_1, ..., \overleftarrow{\mathcal{G}f}_m\}$ *of all backward segments for the graph* G *is a correct partition set of the graph* G.

The proof follows the same structure as the one for the forward graph fragments.

We have shown that the sets of forward and backward fragments $(\overrightarrow{\mathcal{G}f}, \overleftarrow{\mathcal{G}f})$ induce *correct* partitions (i.e. complete, disjoint and rebuildable) of the original RDF dataset G. The elements on each set correspond to *fragments* of the RDF graph G that become distribution units during the *allocation* step. The allocation problem is described and formalized in Section 3.5. in the next section we compare the creation of fragments with other techniques commonly used to persist RDF graphs.

Algorithm 5 *updateGb* function

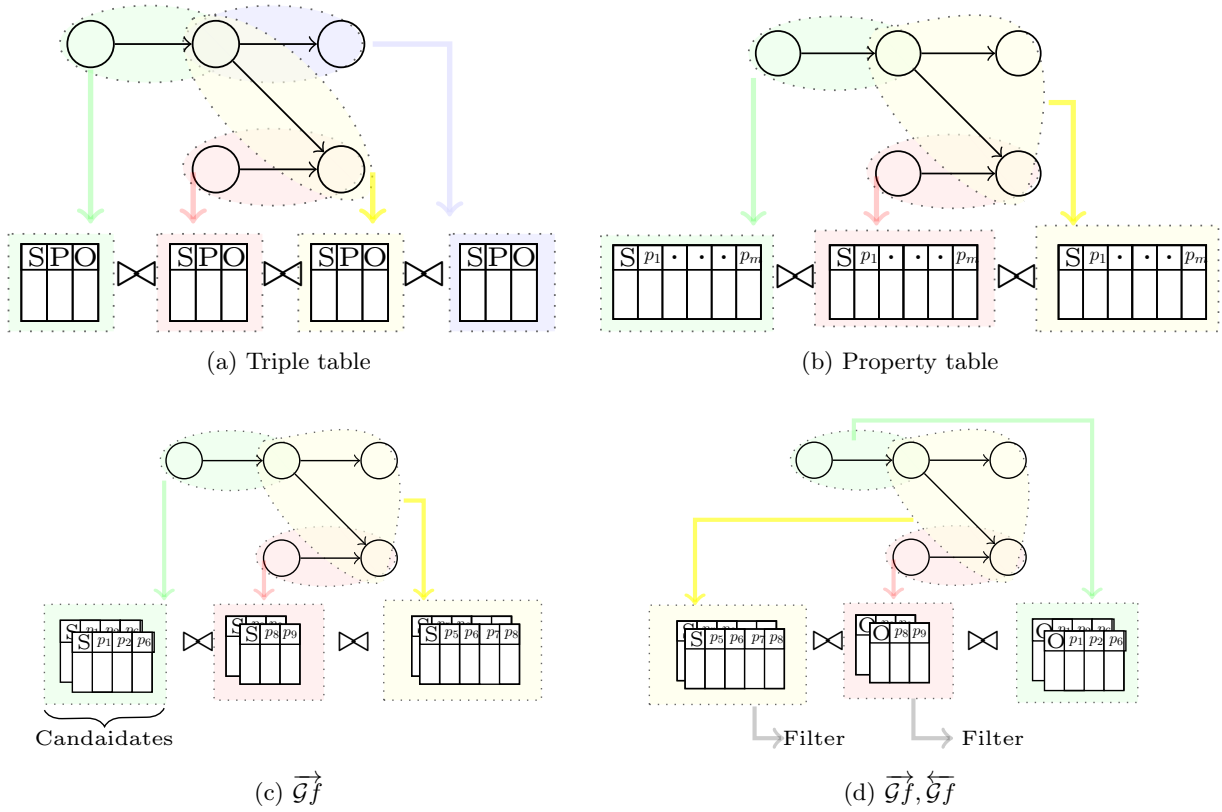
Input: A graph star $\overleftarrow{\mathcal{G}s}(o)$, a set of graph fragments $\overleftarrow{\mathcal{G}f}$, a similarity threshold τ, Map of graph fragment keys $CSM\langle key, \overleftarrow{\mathcal{G}f}_k\rangle$
Output: ($\overleftarrow{\mathcal{G}f}$, $CSM\langle key, \overleftarrow{\mathcal{G}f}_{tmp}\rangle$,$\overleftarrow{\mathcal{G}f}_{tmp}$)

1: **Initialize** $max = 0, \overleftarrow{\mathcal{G}f}_{tmp} = \emptyset$, $key = \mathcal{H}ash(cs(o_i))$
2: **for each** $\overleftarrow{\mathcal{G}f}_i$ in $\overleftarrow{\mathcal{G}f}$ **do**
3: **if** $Sim_B(cs(o), cs(\overleftarrow{\mathcal{G}f}_i)) > \tau$ AND $Sim_B(cs(o), cs(\overleftarrow{\mathcal{G}f}_i)) > max$ **then**
4: $max = Sim_B(cs(o), cs(\overleftarrow{\mathcal{G}f}_i))$
5: $\overleftarrow{\mathcal{G}f}_{tmp} = \overleftarrow{\mathcal{G}f}_i$
6: **end if**
7: **end for**
8: **if** $\overleftarrow{\mathcal{G}f}_{tmp} = \emptyset$ **then**
9: $\overleftarrow{\mathcal{G}f}_{tmp} = \overleftarrow{\mathcal{G}}s(o)$
10: $\overleftarrow{\mathcal{G}f} = \overleftarrow{\mathcal{G}f} \cup \overleftarrow{\mathcal{G}f}_{tmp}$
11: **else**
12: $\overleftarrow{\mathcal{G}f} = appendGf(\overleftarrow{\mathcal{G}s}(o), \overleftarrow{\mathcal{G}f}_{tmp}, \overleftarrow{\mathcal{G}f})$
13: **end if**
14: $CSM.put(key, \overleftarrow{\mathcal{G}f}_{tmp})$
15: **return** $(\overleftarrow{\mathcal{G}f}, CSM, \overleftarrow{\mathcal{G}f}_{tmp})$

3.4 From logical fragments to physical structures

The organization of RDF data in graph fragments allows to detect implicit logical entities that are used as distribution units in RDFDS. Furthermore, using these logical fragments as physical structures could significantly improve the performance of centralized and distributed systems. The work of Pham et al. in [PPEB15] showed that explicitly storing the data using a relational schema automatically discovered boosts the performance of Virtuoso [EM09], a relational-based triple store. Organizing the data into forward and backward graph fragments, regardless of the structure used to persist the data (e.g., tables, indexes or adjacency lists), avoids to scan the whole dataset many times with for a single query as it is done by most of the systems storing the entire RDF graph in a single data structure. This intuition is illustrated in Figure 3.4 comparing the query execution in systems with distinct storage structures. Let us start with the execution in systems storing the data in a single triple table as shown in Figure 3.4a. In these systems, the execution is in general divided by triple patterns (TPs). The query in this figure is divided in four triple patterns. The matches for each TP are found by scanning the entire table or index storing the whole dataset. Then, the matches for each TP are joined to find the final results. Some systems have optimized this strategy by indexing the data in different orders (e.g., SPO, OPS, PSO) to join the results of triple patterns using more efficient algorithms (e.g merge-join in RDF-3X [NW08]). Other strategies, like the property table allow to reduce the number of joins solving star-shaped patterns with a single scan. For example, the yellow pattern shown in Fig 3.4b composed of two single patterns is solved with a single scan of the property table. Still, the property table must be entirely scanned to solve every star-shaped pattern which may cause overheads.

Partitioning the RDF graph in physical fragments can avoid to scan the whole dataset with each query pattern. To do this, the relevant partitions must be identified based on the information available in the query. Forward and backward fragments allow to identify the relevant partitions based on the query's predicates, which are known in most SPARQL queries. The process to solve a query in a system persisting the data with forward graph fragments is illustrated in Figure 3.4c. We assume that the data are stored in tables, but the storage structure



(a) Triple table (b) Property table

(c) $\overrightarrow{\mathcal{G}f}$ (d) $\overrightarrow{\mathcal{G}f}, \overleftarrow{\mathcal{G}f}$

Figure 3.4: Query execution by storage

could have been any other (e.g., indexes, adjacency lists). If the data are organized as forward fragments, the query should be grouped in forward star-shaped patterns too (denoted as $\overrightarrow{SQ}$) so that each pattern group has a candidate set of forward fragments. The candidate set contains forward fragments in which a match to the query is likely to be found. To obtain them one could apply simple rules, for example that the characteristic set of the star-shaped pattern must be contained in the characteristic sets of any of the candidate fragments. More formally, the set of candidate fragments denoted as $\mathcal{C}(\overrightarrow{SQ})$ is defined as $\mathcal{C}(\overrightarrow{SQ}) = \{\overrightarrow{\mathcal{G}f}_k | cs(\overrightarrow{SQ}) \subseteq cs(\overrightarrow{\mathcal{G}f}_k)\}$. Ideally, the total number of triples read from the candidate set is much smaller than the total number of triples of the dataset. Since not all the data is read at the same time, the system's throughput is also improved. In addition, the fragments in the candidate set can be scanned in parallel, reducing the queries' response time. The order in which the triple patterns are executed plays also a very important role in the query performance. Storing the data in physical graph fragments allows to easily collect statistics about them to estimate the most efficient join order. Fragmented data also allows more efficient filtering from one pattern to another, as we will see below, using forward and backward graph fragments simultaneously allows to prune invalid scans from one triple pattern to another.

Combining forward and backward fragments The integration of horizontal and vertical partitions in the relational model has been explored by many researchers in the past. For example, the advisors of a database physical design described in [ANY04] or the Fractured Mirrors [RDS02]. Meanwhile, many massive processing systems propose replication strategies not only to recover and support fault tolerance but to improve the response time of queries. The Hadoop distributed framework for example, stores the data with a default replication factor of 3. We consider an approach similar to the Fractured Mirrors [RDS02] in which a system

stores two copies of the data. One copy organizes the data as forward graph fragments and another as backward graph fragments. This configuration is useful especially when the workload is unknown in the initial partitioning stage. Also, as it is shown in Figure 3.4d, considering both fragments allows to filter the data scanned at each candidate fragment considering the matches from the previous query pattern. For instance, let us consider that the execution starts finding the matches for the yellow star pattern in the Figure 3.4d. The matches found in this pattern could be used as filters to scan the rows (which are the objects) of the candidate backward fragments of the green and red patterns.

3.5 Allocation problem

The allocation of fragments is a mandatory stride in distributed systems. The problem consists in finding the optimal distribution of data fragments to the sites of a computer network. The problem was initially studied in the context of file distribution but later deepen in relational databases where it was proven to be NP-Complete [Esw74, SW85, LY80]. In the relational model, the estimation of how optimal a distribution strategy is depends on various criteria which are evaluated on the basis of cost models. These models estimate the storage and maintenance costs along with performance metrics like the system's throughput and response time. Since the complexity of the problem does not allow to calculate exact solutions in a reasonable time, a number of different heuristics used in operational research (e.g., knapsack problem [CMP82]) have been applied in RDBMS. These techniques are barely used in modern distributed architectures like Hadoop, that despite being based on other execution paradigms, face the same problem of data distribution. These systems split the data by chunks and use hashing functions to distribute them among the sites of the cluster since the data lacks a defined schema.

Similarly, most of RDFDS opt for very simple distribution solutions. These solutions do not guarantee that the triples closely related to each other would be in the same site. A query that merges intermediate results which are not found in the same machine are very inefficient mainly due to the high transfer costs. As in distributed relational databases, we consider the network costs as the processing bottleneck at query runtime. Therefore, the strategies to improve the system's performance should seek to maximize data locality. This can be achieved through indexing, partitioning and replication techniques.

As it was shown in the previous section, using graph fragments as physical storage structures reduces the disk costs and enable parallelism in centralized and distributed systems. In this section, we consider the use of forward and backward graph fragments to reduce the network cost in RDFDS. We define the allocation problem and its constraints. We seek to allocate triples as close as possible to its neighbors to prune the intermediate results generated for a query locally at each site. We assume that the workload is not available, as it is the case when analyzing RDF data. We build our distribution model based on innate data connectivity.

3.5.1 Problem definition

Inputs Let $V = \{V_1, ..., V_p\}$ be the set of forward, backward or both graph fragments $V = \{\overrightarrow{\mathcal{G}f_1}, ..., \overrightarrow{\mathcal{G}f_l}\} \cup \{\overleftarrow{\mathcal{G}f_1}...\overleftarrow{\mathcal{G}f_m}\}$ with cardinality $p = l + m$. The **sites** are represented by the set $S = \{S_1, ..., S_q\}$. We assume that the size of a single graph fragment is always inferior to the maximum capacity of a site $|S_i| \geq |V_j|$. The available space in a site S_k is expressed as $|S_k|$ and the average size of a triple is expressed as $\overline{|t|}$. An *imbalance factor* denoted by ϵ is used to avoid high imbalance in the number of triples between partitions.

Problem Let us consider the functions: $W_V : V \rightarrow \mathbb{N}^+$ returning the number of triples per graph fragment and $W_F : V \times V \rightarrow \mathbb{N}$ returning the number of triples *shared* by two graph fragments. The procedure to calculate the values of both functions is described in Section

3.5.1.1. The allocation function $x_{iS_k} : V \rightarrow \{0, 1\}$ is defined as:

$$x_{iS_k} = \begin{cases} 1 & \text{if } V_i \text{ is stored in site } S_j \\ 0 & \text{otherwise} \end{cases}$$

And $\oplus : x_{iS_k} \rightarrow \{0, 1\}$ is the X-OR operator.

The partitioning problem consists then in finding an allocation function $X_{iS_k} : V \rightarrow S$ that minimizes the total number of *shared* triples by two sites. Given a set of fragments V, sites S and an imbalance factor ϵ:

$$minimize \sum_{\substack{i,j \in \{1,\ldots,p\} \\ k \in \{1,\ldots,q\}}} \Big((x_{iS_k} \oplus x_{jS_k}) \cdot W_F(V_i, V_j) \Big) \tag{3.1}$$

Subject to:

(i) Imbalance constraint:

$$\forall_{k \in \{1..q\}} \Big| \sum_{i=1}^{p} \big(x_{iS_k} \cdot W_V(V_i) \big) - \sum_{j=1}^{p} \big(x_{jS_k} \cdot W_V(V_j) \big) \Big|_{i \neq j} \leq \epsilon \tag{3.2}$$

(ii) Replication factor:

$$\forall_{i \in \{1..p\}} \Big(\sum_{k=1}^{q} x_{iS_k} = R \Big) \tag{3.3}$$

(iii) Available space in sites:

$$\forall_{k \in \{1,\ldots,q\}} \sum_{i=1}^{p} x_{iS_k} \cdot \frac{W_V(V_i)}{|\overline{t}|} \leq |S_k| \tag{3.4}$$

The objective function seeks to minimize at each site the number of triples whose nodes (subject or object) belong to graph fragments located in distinct sites. The constraints are expressed in Equations 3.2, 3.3 and 3.4. They refer to the maximum imbalance allowed per site, to the number of copies of each fragment in the system (which by default is set to $R = 1$) and to the site's size constraint respectively.

3.5.1.1 Estimation of weights

To find the weights W_V and W_F it is necessary to make a second pass over the SPO (or OPS) RDF file to look up the information. The algorithm finding the weights between forward graph fragments is described in Algorithm 6. The algorithm takes as input an RDF file in SPO order, the map of characteristic sets and the map of subjects defined in Alg. 1. It iterates through the file and gets to which forward fragment the subject of the current triple belongs (Step 3). Then, it increments the map of vertex weights (Steps 4-9). Finally, it checks whether the object of the current triple is the subject of a triple in another fragment (Step 10). If so, it increments the counter and updates the map (Steps 12-17).

The procedure to estimate the weights of backward graph fragments is quite similar to the previous one, using the respective CSM and OM maps defined in Alg. 4. If the system considers both sets of fragments simultaneously, the weights are calculated according to the cases shown on Table 3.1.

Finding exact solutions is, as it is the case in relational-based systems, computationally unfeasible and therefore heuristics producing sub-optimal results are the most convenient strategies. In the following section we present a heuristic mapping the fragments to the nodes of a graph.

Algorithm 6 Generation of weights for forward fragments

Input: RDF file D of $\{t_1, ..., t_n\}$ triples $t = \langle s, p, o\rangle$ in SPO order, characteristic set map $CSM\langle key, \overrightarrow{\mathcal{G}f}_k\rangle$, subject map $SM\langle s, \overrightarrow{\mathcal{G}f}\rangle$
Output: $W_V : \langle(\overrightarrow{\mathcal{G}f}_{src}, \overrightarrow{\mathcal{G}f}_{dest}), count\rangle$ $W_F : \langle\overrightarrow{\mathcal{G}f}, count(t)\rangle$,

1: **Initialize** src =`null`, $dest$ =`null`, $W_V = \emptyset$, $W_F = \emptyset$, $count = 0$
2: **for each** $\langle s, p, o\rangle$ in D **do**
3: $\overrightarrow{\mathcal{G}f}_{src} = SM.get(s)$
4: **if** $\overrightarrow{\mathcal{G}f}_{src} \in W_V.keySet()$ **then**
5: $count = W_V.get(\overleftarrow{\mathcal{G}f}_k)$
6: $W_V.put(\overrightarrow{\mathcal{G}f}_{src}, count + 1)$
7: **else**
8: $W_V.put(\overrightarrow{\mathcal{G}f}_{src}, 1)$
9: **end if**
10: **if** $o \in SM.keySet()$ **then**
11: $\overrightarrow{\mathcal{G}f}_{dest} = SM.get(o)$
12: **if** $(\overrightarrow{\mathcal{G}f}_{src}, \overrightarrow{\mathcal{G}f}_{dest}) \in W_F.keySet()$ **then**
13: $count = W_F.get((\overrightarrow{\mathcal{G}f}_{src}, \overrightarrow{\mathcal{G}f}_{dest}))$
14: $W_F.put((\overrightarrow{\mathcal{G}f}_{src}, \overrightarrow{\mathcal{G}f}_{dest}), count + 1)$
15: **else**
16: $W_F.put((\overrightarrow{\mathcal{G}f}_{src}, \overrightarrow{\mathcal{G}f}_{dest}), 1)$
17: **end if**
18: **end if**
19: **end for**
20: **return** W_V, W_F

3.5.2 Graph partitioning heuristic

The set of graph fragments is mapped to an undirected weighted graph, transforming the allocation problem into a graph partitioning problem. The graph partitioning problem has been proved to be a very complex and computationally expensive problem. However, many efficient heuristics have been developed as detailed in Chapter 2 (e.g., METIS [KK98a]).

Let us map the set of graph fragments $\mathcal{G}f$ into a directed weighted graph represented by the quadruple $\mathcal{G} = (V, E, W_V, W_E)$. V corresponds to the set of nodes which are the sets of forward and backward graph fragments. The node's weights are represented in the set W_V, each element $w(V_i)$ corresponds to the number of triples on each fragment. E represents the set of edges and W_E the set of weights ($w(E_{ij})$ between the nodes i and j). The weights are calculated following the strategies presented in the previous section. An example graph is shown in Figure

Table 3.1: Edge's weights in fragment graph $\mathcal{G}$

Edge type	Condition[a]
$(\overrightarrow{\mathcal{G}f_i}, \overrightarrow{\mathcal{G}f_j})$	$object(t_i) = subject(t_j)$
$(\overleftarrow{\mathcal{G}f_i}, \overleftarrow{\mathcal{G}f_j})$	$subject(t_i) = subject(t_j)$
$(\overleftarrow{\mathcal{G}f_i}, \overleftarrow{\mathcal{G}f_j})$	$object(t_i) = subject(t_j)$
$(\overrightarrow{\mathcal{G}f_i}, \overleftarrow{\mathcal{G}f_j})$	$subject(t_i) = subject(t_j)$
$(\overrightarrow{\mathcal{G}f_i}, \overleftarrow{\mathcal{G}f_j})$	$object(t_i) = subject(t_j)$

[a]The weight of an edge $W(edge)$ is the number of triples t_i such that:
$t_i \in G_i, t_j \in G_j$, and $t_i \wedge t_j$ fulfill the condition in this column accordingly.

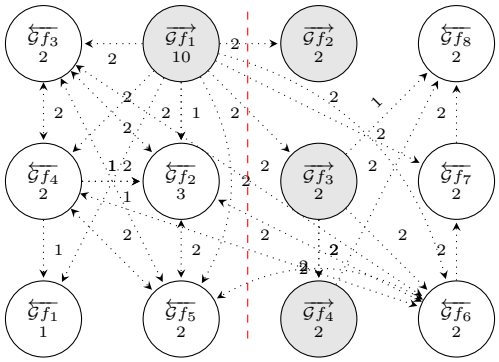


Figure 3.5: Graph partitioning example

3.5. In this case the graph is split in two partitions, the goal is to minimize the cuts to implicitly diminish the data that would be transferred if the fragments are joined.

Given a graph $\mathcal{G}$ and a given a number $p \in \mathbb{N}^+$ indicating the number of partitions, the graph partitioning problem asks for blocks of nodes $\mathcal{V}_1, ..., \mathcal{V}_p$ such that $\mathcal{V}_1 \cup ... \cup \mathcal{V}_p = V$ and $\mathcal{V}_i \cap \mathcal{V}_j \forall i \neq j$. As it was defined before, a balance parameter ϵ is used to regulate the size between partitions.

Objective function We seek a partition that minimizes the *total cuts*.

$$minimize \sum_{i<j} w(E_{ij}) \tag{3.5}$$

Subject to:

$$\forall_{k \neq j} (V_i \in \mathcal{V}_k \wedge V_j \in \mathcal{V}_l) \tag{3.6}$$

$$|w(E_i) - w(E_j)|_{\forall i \neq j} \leq \epsilon \tag{3.7}$$

The first constraint refers to the fact that a graph fragment should be assigned to only one block of nodes (i.e. partition). Replication is therefore not deemed in the graph heuristic approach. The second constraint refers to the imbalance factor between partitions. The constraint seeking not to exceed the size of a given partition defined in Eq. 3.4 is not considered by most of the graph partitioning heuristics. Accordingly, we do not explicitly declare it here. If the size of a partition exceeds the available space in a site, re-partitioning strategies should be considered. These strategies are described in Section 3.7.

3.6 RDF partitioning example

The following example gives an overview of our approach. Let us consider the RDF graph G illustrated in Figure 3.6. This graph is a miniature version of the graph presented in the previous chapter in Figure 2.10. We start identifying forward graph fragments. Next, we show how physically storing the data with this structure contributes at query runtime. Then, we build the backward graph fragments for this graph. Finally, considering forward and backward fragments, we show an example of our allocation heuristic.

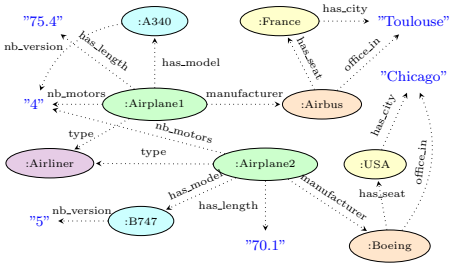


Figure 3.6: Miniature of graph G of Figure 2.10

Forward graph fragments Let us start gathering the triples in forward graph fragments. As it was described in Section 3.3.1 we group first the triples by its subjects. We obtain 8 groups of triples (`Airplane1, Airplane2, B747, A340, Airbus, Boeing, France` and `USA`). We name each of these groups a *forward graph star* $\overrightarrow{\mathcal{G}s}$ representing instances of higher-level entities (e.g., an Airplane, a Manufacturer, an Airplane Model). We observe that two instances of the same high-level entity share the same (or almost the same) set of predicates. In general, the forward graph fragments can be uniquely identified by the set of its emitting edges (i.e. characteristic set). In the example, the forward stars `Airplane1` and `Airplane2` share the same predicates (`has_model, has_length, manufacturer, nb_motors` and `type`). Consequently `Airplane1` and `Airplane2` belong to the same group that we name a *forward graph fragment* $\overrightarrow{\mathcal{G}f_i}$ of G. There are in total 4 $\overrightarrow{\mathcal{G}f}$ in our example, two of them are illustrated in Figure 3.7b showing forward graph fragments physically stored in a tabular form.

The query Q_1:

```
SELECT ?a ?m ?l
WHERE {
   ?a :type :Airliner .
   ?a :has_length ?l .
   ?a :has_model ?m .
   FILTER (?l > 65.0)
}
```

could be efficiently solved by looking for the relevant graph fragment whose characteristic set contains the predicates `type, has_length` and `has_model` avoiding a full scan of the data.

Still, an organization of the data in forward graph fragments is non-optimal when solving queries with a very reduced number of predicates like for instance the query Q_2:

```
SELECT AVG(?y)
WHERE {
   ?x :nb_motors ?y .
}
```

To solve this query, all the triples of the forward fragment `Airplane` must be read, even the triples with predicates other than `has_motors`. When exploring the graph for matches, 10 triples in the Airplane forward graph fragment must be scanned when only 2 triples are relevant.

Backward graph fragments The previous problem is quite similar to the one that motivated vertical partitions in relational databases. Partitioning a table by attributes can be more efficient for some queries, where they involve few attributes. In our case, we create groups of triples by attributes which are named *backward graph fragments*. To build them, we gather first the triples by its *incoming* edges. For the graph of Figure 3.6 we obtain 13 groups `"5", "4", "75.4", "70.1", Boeing, Airbus, France, USA, "Toulouse", "Chicago", A340, B747` and `Airliner`. We name these groups *backward graph stars* $\overleftarrow{\mathcal{G}s}$. Similarly to what was done for the forward fragment, we use the characteristic set to identify the groups of attributes. We name them *backward graph fragments* $\overleftarrow{\mathcal{G}f}$.

For the example graph of Figure 3.6 we obtain 8 backward graph fragments: the group of city names with the `has_city` predicate (2 triples), the group of manufacturers (2 triples), lengths (2 triples), locations (2 triples), models (2 triples) and types (2 triples). There is a group gathering the triples having only the `nb_version` predicate (1 triple), and another group gathering the triples with the predicates `nb_version` and `nb_motors` (3 triples). The predicates could stored

Subject	has_model	has_length	nb_motors	type
Airplane1	A340	75.4	4	Airliner
Airplane2	B747	70.1	4	Airliner

...

Subject	has_seat	officie_in
Airbus	France	Toulouse
Boeing	USA	Chicago

(a) Forward graph fragments stored as relations

```
{ "nb_version,nb_motors" :{
    "4" :{ "nb_version" :[A340], "nb_motors" :[Airplane1, Airplane2] },},
 "nb_version" :{
    "5" :{"nb_version" :[B747] }},
 "has_city,office_in" :{
    "Chicago" :{"has_city" :[USA], "office_in" :[Boeing]}, "Toulouse" :{"has_city" :[
        France],"office_in" :[Airbus] }},
    ... }
```

(b) Backward graph fragments stored in adjacency list

Figure 3.7: Graph fragments examples

physically in adjacency lists in a json file as shown in Figure 3.7b.

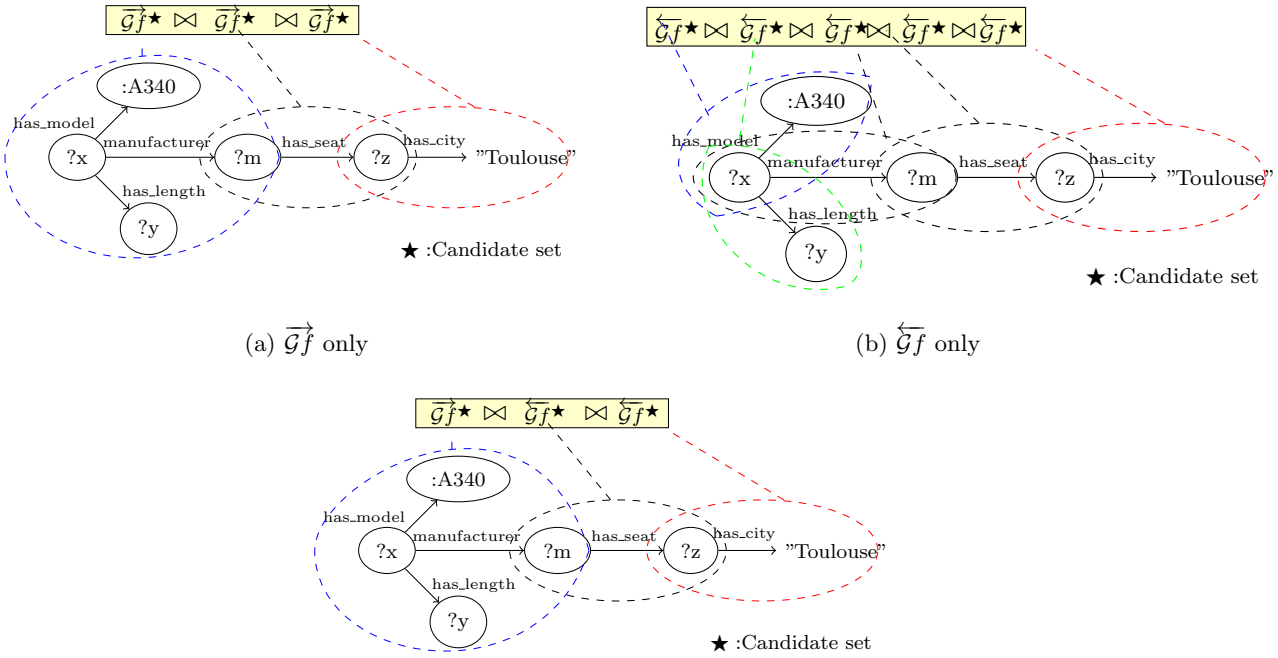
The solution to the query getting the average number of motors in an entity could be solved by scanning only the backward graph fragment storing triples with the predicate `nb_motors`. The execution engine checks which characteristic sets contain the given predicate and scans only the backward graph fragment (`nb_version, nb_motors`) $\overleftarrow{\mathcal{G}b_5}$ that gathers 3 triples. Comparing to the result obtained when the data are organized as *forward graph fragments*, the execution process on the backward fragments is more efficient. The process saves, in this case, the resources used to scan 7 triples. However, as it is the case for vertical partitions in the relational model, the performance can be degraded for SPARQL queries joining many single patterns.

Combining forward and backward fragments Some queries require to join several forward or backward graph fragments to find a solution.

For example, let us consider the query Q_3:

```
SELECT ?y ?z
WHERE {
   ?x  :has_model   :A340 .
   ?x  :has_length  ?y .
   ?x  :manufacturer ?m .
   ?m  :has_seat    ?z .
   ?z  :has_city    "Toulouse" .
}
```

If the data are physically stored as forward graph fragments, the query can be solved scanning at least 3 candidate sets as shown in Figure 3.8a. In this example, we start by scanning the fragments whose characteristic set contains the predicates `has_model`, `manufacturer` and `has_length`. The results of this matches are joined with the candidate fragments assigned to the nodes with outgoing edges in the query. On the other side, if the data are stored physically as backward graph fragments, the execution would involve many more joins. A candidate set of fragments is needed for each node with incoming edges in the query graph as shown in Figure 3.8b. Even if a backward graph fragment would be much more efficient to find the bounded value in the fifth query pattern of Q_3, the great number of joins needed impairs the overall performance. To fix this problem, a strategy presented in Section 3.4 considers storing the data using both types of physical fragments. The execution schema for Q_3 using this strategy is



(a) $\overrightarrow{\mathcal{G}f}$ only

(b) $\overleftarrow{\mathcal{G}f}$ only

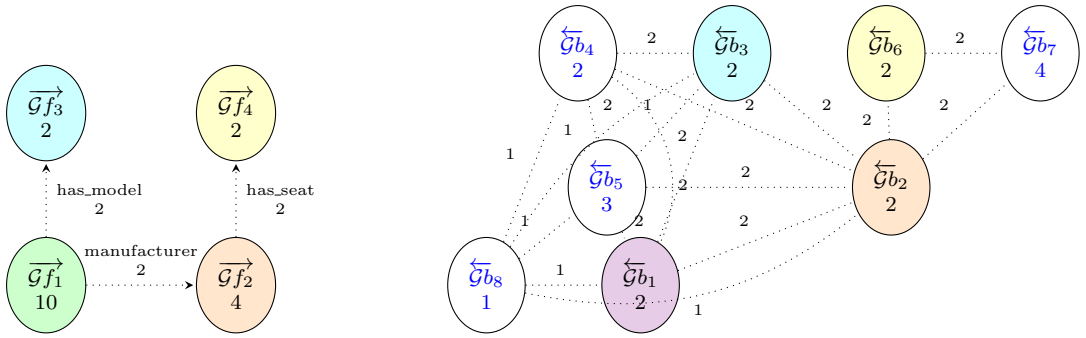
(c) $\overrightarrow{\mathcal{G}f}$ and $\overleftarrow{\mathcal{G}f}$

Figure 3.8: Execution strategies for query Q_3

shown in Figure 3.8c. We do not go into more details about the query execution modeling using the graph structures presented here and the selection of an optimal execution plan, both topics are out of the scope of this thesis.

Graph fragments allocation As it was seen in Section 3.5, forward and backward graph fragments can be used as *distribution units* to allocate triples in a RDFDS. We propose to use graph partitioning heuristics to find solutions to the allocation problem in a reasonable time. The graph shown in Figure 3.9a is the weighted graph of forward graph fragments of Figure 3.6. The graph has four nodes, one for each fragment, whose weight is the number of triples stored on each of them. An edge is drawn between two nodes if there is a triple on the source fragment whose object is the subject of the other fragment. The weights of each edge are the number of triples shared between both fragments. The edge's weights represent the number of triples that should be transferred between two fragments when they are joined. Let us consider for example the edge's weights between the forward graph fragments $\overrightarrow{\mathcal{G}f_1}$ and $\overrightarrow{\mathcal{G}f_2}$. The weight in this case is 2 since the subjects `:Airplane1` and `:Airplane2` stored in $\overrightarrow{\mathcal{G}f_1}$ are connected to the subjects `:A340` and `:B747` that are the subjects of triples stored in $\overrightarrow{\mathcal{G}f_2}$.

Figure 3.9b illustrates the weighted graph of backward graph fragments for the graph of Figure 3.6. The node's weights are the number of triples on each fragment and the edges are also drawn based on the data shared by triples in two distinct fragments. The cases are detailed in Table 3.1. The detail of the data on each fragment is found in the Appendix A. We do not show the weighted graph combining forward and backward graph fragments for readability. Finally, the allocations is decided using a graph partitioning heuristic like METIS [KK98a]. If the chosen graph partitioning heuristic works only for non directed graphs, the conversion is very simple.



(a) Weighted graph of $\overrightarrow{\mathcal{G}f}$ (b) Weighted graph of $\overleftarrow{\mathcal{G}f}$

Figure 3.9: Extract of weighted graphs of fragments for graph G

3.7 Dealing with large fragments

When the size of a forward or backward graph fragment exceeds a predifined threshold, the fragment needs to be repartitioned. The threshold can be set to the space available in a single site of a distributed system because if the size of a graph fragment surpasses the available space in a site, we will not be able to place it anywhere. To partition within a fragment, we apply a function that sub-partitions it keeping the data belonging to the same graph star together. In other words, the function must hold triples with the same subject or object in the same forward or backward graph fragment respectively. This allows to keep the previously found logical structure intact so that the allocation methods based on the data connectivity can still be applied. The re-partitioning function is either obtained by hashing the triple's subject or object, or by using a range strategy on the predicates as seen below.

Let us start defining the fragments that need to be re-fragmented. Given a set of sites $\{s_1, ..., s_s\}$ and graph fragments $\mathcal{G}f = \{\mathcal{G}f_1, ..., \mathcal{G}f_m\}$. If the site's capacity and a graph fragment's size are defined as $|s_i|$ and $|\mathcal{G}f_i|$ respectively. A graph fragment needs to be re-partitioned if $|\mathcal{G}f_i| > |s_i|$. The number of sub-partitions is obtained as:

$$n = \left\lceil \frac{|\mathcal{G}f_i|}{|s_i|} \right\rceil$$

Let us consider that the size of a graph fragment $\mathcal{G}f_i$ is bigger than the available space for a site. The *hashing* strategy applies a hashing function on every graph star such that:

$$\forall \mathcal{G}s(h) \in \mathcal{G}f_i : \mathcal{H}(\mathcal{G}s(h)) \mod (n)$$

where n is the number of sub-partitions. The advantage of this strategy is the creation of uniform partitions with similar sizes.

The next sub-partition strategy, named *range re-partitioning*, maps the forward or backward fragments to a partition according to the values of a predicate(s). If a fragment involving k predicates $(p_1, .., p_k)$ is too large, we propose to repartition it. To do so, we first consider one of its *numerical predicates* (having the following form: $p_n \theta value_n$, where $value_n$ is an integer or a float) and then re-split the initial fragment into two new fragments based on the domain of $value_n$. We iterate this procedure till having fragments with reasonable size. The information related to the domain of different predicates could be available in the triple store statistic module [ZMG+20]. In our case, we compute this information from the KG.

The process is iterative. For example, if after computing the re-partition operations on a fragment its size is still too large, the user can decide to use either of the previous strategies again.

3.8 Conclusion

In the relational databases world, data partitioning has been identified for a long time as a key optimization and manageability technique. In this context, data partitioning, characterized by its simplicity, is independent of the storage approach of the data. Consequently, a user does not have to deal with the physical storage layer of the system hosting the data, even though the partitions impact seriously the physical layer. These findings represent a wisdom of traditional relational data partitioning. In this chapter, we claim to reproduce this wisdom by tackling the partitioning problem of RDF data. Unlike traditional partitioning techniques, RDF techniques are dependent on the partitioning strategy and are difficult to generalize for different systems.

We draw on the philosophy adopted by the relational model to address partitioning within distributed RDF systems. Precisely, we introduced a logical layer to the merely physical distribution process of triples to a set of sites. We formalize and detail the algorithms used to create the logical entities that we named graph fragments ($\mathcal{G}f$). Our entities extend the notion of partitioning by instances and by attributes in the relational model and offer great balance between conceptual and intuitive simplicity, in addition to its logic expressiveness. We formalized the allocation problem and presented a graph-based heuristic to minimize communication costs.

Chapter 4

RDFPartSuite in Action

Contents

Summary In the previous chapter we defined a logical layer to partition RDF datasets. This layer allows partitioning RDF data independently of its physical storage implementation while preserving its logical graph structure. In this chapter, we present the partitioning framework *RDFPartSuite*. This framework is built upon the logical structures that we defined in the previous chapter. It is composed of three main modules: fragmenter, allocator and dispatcher. These modules provide functionalities to assist managers (triple stores' administrators) to partition Knowledge Graphs based on their requirements. We detail the incorporation of this framework to a centralized (RDF_QDAG) and a distributed (gStoreD) triple store. We conducted several experiments that demonstrate the virtues and the costs of incorporating our structures while loading and processing RDF data. Finally, we present a set of assistance tools proposed by our framework. These tools relieve managers from the partitioning design tasks with a declarative partitioning language and a partitioning advisor. The following chapter is organized as follows. We start in Section 4.2 giving an overview of the centralized triple store where we incorporated our framework. Then, in Section 4.3 we compare the loading costs of our framework with respect to other fragmentation strategies from the state of the art. Next, in Section 4.4 we evaluate our fragmentation strategies in terms of data coverage and query performance. The fragmentation process is evaluated in a centralized and also in a distributed triple store to which we incorporated our framework. In Section 4.5, we evaluate the allocation strategies of graph fragments. Then, in Section 4.6 we detail the assistance tools provided by *RDFPartSuite*. These tools are a partitioning language (Section 4.6), and an advisor (Section 4.7) . Finally Section 4.8 concludes this chapter.

4.1 Introduction

In the previous chapter we promoted a logical layer for RDF data partitioning. We defined a set of structures that allow partitioning RDF data regardless of how it is stored. These structures preserve the logical graph nature of RDF data and are used as fragmentation units. The logical fragments are placed to the sites of a distributed system according to data-driven algorithms that we have also detailed. In this chapter we introduce a framework named *RDFPartSuite*. The framework provides generic functionalities build upon the logical structures defined in the previous chapter. It can be adapted according to the manager's specifications (creator's requirements, triple store, available infrastructure, etc.). *RDFPartSuite* provides a standard way to build and deploy RDF partitioning schemas in a universal and *reusable* environment. The framework is composed of three main modules:

(i) *Fragmenter:* this component is in charge of partitioning the triples using their logical representation as graph fragments (i.e. $\overrightarrow{\mathcal{G}f}$ or $\overleftarrow{\mathcal{G}f}$). The fragments can be built using structural or semantic rules as described in Section 3.3.1.1.

(ii) *Allocator:* the allocator builds a distribution schema for the fragments built by the fragmenter component. This schema is built using the data-driven strategies detailed in Section 3.5.

(iii) *Dispatcher:* this component sends the fragments to the sites of a distributed system following the allocation schema produced the allocator. The dispatcher is also in charge of loading the data to the target triple store.

In addition to these modules, our framework integrates a set of assistance tools for the administrators (managers) of triple stores. These tools include a declarative partitioning language, to fragment, allocate, and dispatch a Knowledge Graph to a target triple store. And, for non-expert users, it provides a partitioning wizard to help them build a partitioning schema for their data.

This chapter is dedicated to this framework. We start describing how to incorporate our framework to a centralized triple store (*RDF_QDAG* [KMG+20]). This system uses data partitioning and graph exploration techniques to accelerate the execution of SPARQL queries in a centralized environment. We have also shown how to incorporate our framework to a distributed triple store (*gStoreD* [ZÖC+14]). We carried out a significant number of experiments that showed the feasibility of our partitioning strategies, its effectiveness and its limits. Finally, we presented *RDFPartSuite*'s assistance tools.
The sections of this chapter are organized as follows. We start in Section 4.2 detailing the incorporation of our framework to *RDF_QDAG*. We describe its main modules along with their storage and processing strategies. This system uses the logical fragments (i.e. $\overrightarrow{\mathcal{G}f}$ or $\overleftarrow{\mathcal{G}f}$) as physical storage units. This allows us to compare the performance gain of our model versus alternative partitioning strategies applied in other systems of the state-of-the-art. We detail our implementation choices and examine carefully the pre-processing costs (Section 4.3). In Section 4.4, we evaluated our fragmentation strategies in terms of query performance. We performed these experiments in *RDF_QDAG*, a centralized relational-based system, and a graph-based distributed triple store (*gStore*). Then, in Section 4.5 we largely discussed the allocation strategies supported by our system and the results of our experimental comparisons. Finally, in Sections 4.6 and 4.7 we detail our assistance tools which are the declarative partitioning language and the partitioning advisor respectively.

4.2 RDF_QDAG

Modern RDF processing systems can be distinguished in two groups as it was studied in Section 2.3.2. The first group uses the relational model to store RDF triples in tables with diverse con-

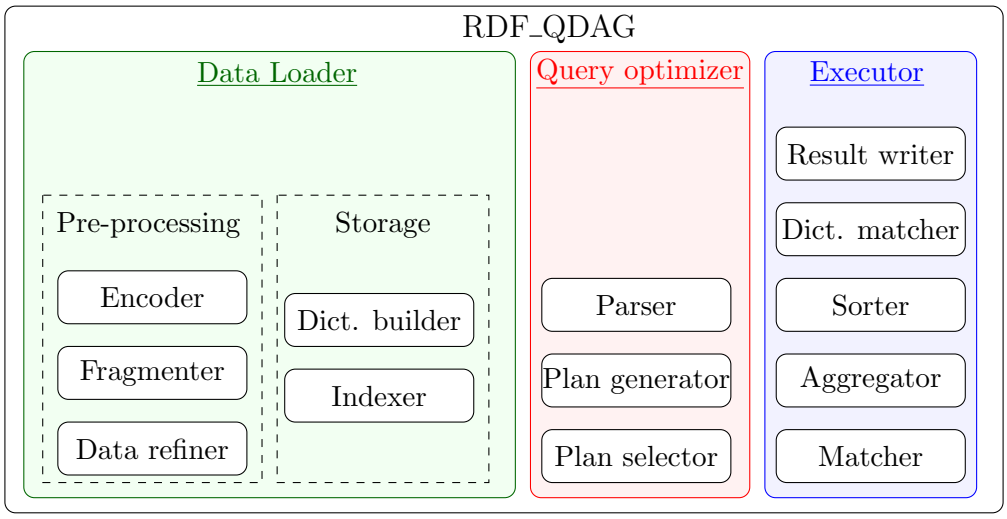


Figure 4.1: RDF_QDAG architecture

figurations. The systems under this group are more easily scalable as they can use optimization strategies such as indexes and partitions available for relational databases. Unfortunately, the performance in these systems degrades rapidly, specially when dealing with complex SPARQL queries. The relational model is not suitable for handling RDF data inherently represented as a graph. The second group comprises RDF processing systems maintaining the graph structure of RDF data. These systems store the triples in data structures specifically designed to store graphs (cf. Section 2.2.2). The major issue confronted by these approaches is scalability to large RDF graphs [AHKK17, Özs16]. They fail to efficiently manage the use of main memory and are not scalable in infrastructures with limited resources. *RDF_QDAG* [KMG+20] combines the virtues of data partitioning in relational-based systems with an execution model based on the exploration of the RDF graph. The system supports Basic Graph Pattern (BGP) SPARQL queries as well as wildcards, aggregation and sorting operators. The data in RDF_QDAG are physically partitioned in Forward and Backward graph fragments. It is able to prune irrelevant graph fragments using the predicate's bounded values in the query. This feature allows to explore only the fragments in which a match is likely to be found. As mentioned before, the query execution in this system is based on graph exploration. To avoid memory overflows, the execution engine of RDF_QDAG is based on the Volcano parallel query evaluation system [Gra94]. In what follows we give an overview of RDF_QDAG describing its general architecture, storage, optimization and execution layers.

4.2.1 System architecture

The systems is composed of three main modules: Data Loader, Query Optimizer and Executor. The architecture is illustrated in Figure 4.1. Below we describe each of the system's components.

4.2.1.1 Data loader

This module prepares, indexes and partitions the raw RDF data received as input. It is composed of two main components: pre-processing and storage. The first one transforms the RDF data (stored in N-Triples, N3 or Turtle files) encoding all the strings and arranging the encoded data in forward and backward graph fragments. The string transformation takes place in the *Encoder* unit. The result of this stage is given as input to the *Fragmenter* component that splits the data in forward and backward graph fragments using Algorithms 1 and 4. The *Data refiner* adds to each triple on each fragment information related to its direct neighbors as it will be detailed later in Section 4.2.2. The *pre-processing* module is entirely coded in Java. The storage component,

coded in C++, takes the encoded refined file from the pre-processing as input and sets up the data in indexes. The result of this step are a set of binary files used by the query optimizer and executor modules.

4.2.1.2 Query optimizer

This module is responsible for finding the optimal execution plan for a SPARQL query. It is composed of three main sub-modules: *parser, plan generator* and *plan selector*. The *parser* transforms the SPARQL query to a series of structures that are understandable by the *plan generator* sub-component. The *plan generator* enumerates a series of "acceptable plans" defined in Section 4.2.3 that are later ranked based on a cost model in the *plan selector* component. The best plan is selected using heuristics that take simple statistics from the fragmented data to estimate the execution cost of each plan. The optimization strategies applied by this component are described in detail in [ZMG+20].

4.2.1.3 Executor

This component receives the chosen execution plan from the optimizer and implements it. The *matcher* sub-component interrogates the B+Tree index (cf. Section 4.2.2) and gets the encoded results. This component might interrogate the string dictionary (using the *dictionary matcher* sub-component) during the execution to enhance the index search. However, the *dictionary matcher* is often called at the end of the query execution decoding the original string values. The data are aggregated and sorted on the respective sub-components when necessary. The result's output (e.g., in console, to a file) is managed by the *result writer*.

4.2.2 Storage model

As we mentioned previously, the data in RDF_QDAG are physically stored as graph fragments. The system stores two copies of the data, one as forward and another as backward graph fragments. This choice will be justified when presenting the execution model. In this section we present how the data are compressed and how the triples are physically represented on disk. Let us start describing the output files from the pre-processing phase. First, the predicates from the input dataset are gathered in a single file with their identifiers in a file named `pred_index`. An example of this file for the graph data of Figure 2.10 is shown in Figure 4.2a. Then, the identifiers of the forward and backward graph fragments are stored in separate files as shown in Figures 4.2b and 4.2c respectively. Lastly, the triples assigned to each graph fragment are stored in separate files. For each graph fragment (identified by a fragment id `fid`), the data are separated in three files:

- *Dictionary* (`fid.dic`): This file stores a map of the strings with their respective IDs of the triples assigned to the graph fragment. The strings are sorted in lexicographical order and the IDs are integer values. An example of this file is given in Figure 4.2d. The dictionary file is used as input by the Dictionary Builder component to store the data in a compressed string index.
- *Schema* (`fid.schema`): This file stores the list of predicates (encoded with the IDs from the pred_index file) along with the data type of the triple's object if the schema file corresponds to a forward graph fragment or the subject otherwise. The first line of the file stores the data type of the encoded subjects (in case of a forward fragment) or the encoded object (in case of a backward fragment) stored in the graph fragment. The schema file for the graph fragment in our running example is illustrated in Figure 4.2e.
- *Data* (`fid.data`): This file stores the encoded subject, predicate and object of each triple in the fragment. Also, since the evaluation method of RDF_QDAG is based on the exploration

Predicate	ID
has_city	0
has_length	1
has_model	2
has_seat	3
manufacturer	4
nb_motors	5
nb_version	6
office_in	7
type	8

(a) Predicates identifiers

Predicates	fid
0	1
6	2
3,7	3
1,2,4,5,8	4

(b) Forward fragments identifiers

Predicates	fid
1	5
2	6
3	7
4	8
6	9
8	10
0,7	10
5,6	11

(c) Backward fragments identifiers

4.dic	
:A340	10
:Airbus	20
:Airliner	30
:Airplane1	40
:Airplane2	50
:B747	60
:Boeing	70
Chicago	80

(d) Dictionary file

4.schema
int
1:float
2:int
4:int
5:int
8:int

(e) Schema file

4.data
40 0 2 10 6 2
40 0 1 75.4 5 0
40 0 5 4 12 0
40 0 8 30 10 0
40 0 4 20 8 3
50 0 2 60 6 6
50 0 1 70.1 1 0
50 0 5 4 12 0
50 0 8 30 10 0
50 0 4 70 8 0

(f) Data file

Figure 4.2: Storage files examples for graph of Figure 2.10

of the RDF graph, we materialized the information relative to where the direct neighbors of the subject and object of each triple are stored. The data on each fragment are first sorted by subject (or object in backward graph fragments). Then, each triple is stored in a different line of the data file with the following information separated by spaces:

- $Node_1$: it is the encoded subject (or object in a backward graph fragment). It is represented using 8 bytes.
- $\mathcal{G}f$: it is ID (fid) of the fragment (if any) storing the incoming (or outgoing edges in a backward graph fragment) edges of $Node_1$. If there is any this value equals zero.
- *Predicate*: this value is the predicate's ID of the triple. It is represented using 4 bytes.
- $Node_2$: it is the encoded object (or subject in a backward graph fragment). It is represented using 8 bytes.
- $\mathcal{G}f_{in}$: it is the id (fid) of the fragment storing the incoming edges (if any) of $Node_2$. It is represented using 4 bytes.
- $\mathcal{G}f_{out}$: it is the id (fid) of the fragment storing the outgoing edges (if any) of $Node_2$. It is represented using 4 bytes.

In the example of Figure 4.2f, the first line corresponds to the triple `<:Airplane1> <:has_model> <:A340>`, encoded as:

$$\overbrace{40}^{\texttt{:Airplane1}}\ \overbrace{0}^{\overleftarrow{\mathcal{G}f}}\ \overbrace{2}^{\texttt{:has\_model}}\ \overbrace{10}^{\texttt{:A340}}\ \overbrace{6}^{\overleftarrow{\mathcal{G}f}}\ \overbrace{2}^{\overrightarrow{\mathcal{G}f}}$$

The data file can be very dense and include many redundant information. The compression strategy for this file are shown in the following section.

Indicator	Predicate
17 bits	1 bit

$Node_1$	$\mathcal{G}f$	$Node_2$	$\mathcal{G}f_{in}$	$\mathcal{G}f_{out}$
0-8 bytes	0-4 bytes	0-8 bytes	0-4 bytes	0-4 bytes

Figure 4.3: Compression structure in RDF_QDAG

4.2.2.1 Data compression

The files storing the triples and its neighbors described in the previous section gather all the significant information to traverse the data from one fragment to another. To make the exploration even more efficient, each data fragment is stored in a clustered B+Tree index. To avoid the storage of redundant data and to limit the size of the indexes, the data are compressed using very similar techniques to the ones applied in RDF-3X [NW08]. Let us recall some features of the data stored in graph fragment file: i) First, the triples are sorted by subject (or object in backward graph fragments)and by predicate id, ii) The triples share the same characteristic set, iii) The characteristic set is unique for a single subject (or object in a backward graph fragment). These features allow to devote a single bit per triple (0 or 1) to indicate whether it has the same predicate as the previous one. Thus, it is not necessary to store the predicate value for each triple. This idea is reproduced to store $Node_1$ and $Node_2$, where the 0 bit is used to denote when their values do not change with respect to the previous triple. If the value has changed, 8 bytes are devoted to store the identifier or value for both $Node_1$ and $Node_2$. The 8 bytes are enough to support Integer $\rightarrow$ 4 bytes, Long $\rightarrow$ 8 bytes, Float $\rightarrow$ 4 bytes and Double $\rightarrow$ 8 bytes datatypes.

The values of $\mathcal{G}f$, $\mathcal{G}f_{in}$ and $\mathcal{G}f_{out}$ are represented using the same state logic. The state 0 indicates that the information does not change with respect to the previous triple or if the value does not exist (as in the first line of the example in Figure 4.2f). The following 4 bytes are used to indicate the identifiers of the fragments if any.

An indicator of 17 bits is assigned to each triple to state the number of bytes allocated to each element of the triples. The compression structure for each triple is shown in Figure 4.3.

4.2.3 Execution model

In this section we present the execution model of RDF_QDAG that, as we mentioned in the previous sections, is depicted as a graph exploration process. Let us start by recalling the evaluation model used by most of the state-of-the-art RDF systems. In these systems, the evaluation of a SPARQL query is modeled as a succession of triple patterns. Let us consider, for instance, the following SPARQL query:

```
SELECT ?x ?y ?m WHERE{
?x :has_length ?y .          #tp1
?x :manufacturer ?m .        #tp2
?m :office_in "Toulouse" .  #tp3
}
```

This query is composed of three single patterns, and the execution is modeled as a sequence of joins:

$$tp_1 \bowtie_{?x} tp_2 \bowtie_{?m} tp_3$$

The optimization problem in these systems consists in finding the optimal join order. On the other hand, the query execution in graph-based systems is modeled as a graph exploration process. These systems model the query as a graph as illustrated in Figure 4.4. An *exploration plan* defines the order in which the query vertices are assigned to graph vertices [BFVY18]. As with join-based plans, a graph-based plan determines the order of the operations performed by the executor component of the system to find the query matches.

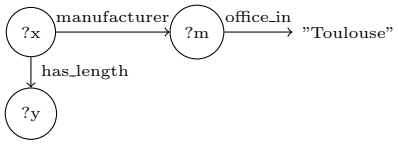


Figure 4.4: Query graph example

The nodes in the query graph can be grouped with their incoming or outgoing edges. This simplifies the execution plans and also allows modeling the query exploration with a similar logic to the one used to group triples in graph fragments in RDF_QDAG. We name these structures *Forward Query Stars* and *Backward Query Stars* if they group the query's nodes by their outgoing or incoming edges respectively. They are formally described in Definition 4.1. The sets of forward and backward query stars for the query in Figure 4.4 are $\overrightarrow{QS} = \{\overrightarrow{QS}(?x), \overrightarrow{QS}(?m)\}$ and $\overleftarrow{QS} = \{\overleftarrow{QS}(?y), \overleftarrow{QS}(?m), \overleftarrow{QS}(\text{"}Toulouse\text{"})\}$.

Definition 4.1 ***Query Star*** *Let Q be the SPARQL query graph. A Forward Query Star $\overrightarrow{QS}(x)$ is the set of triple patterns such that $\overrightarrow{QS}(x) = \{(x,p,o)|\exists_{p,o} : (x,p,o) \in Q\}$, x is named the head of the Query Star. Likewise, a Backward Query Star $\overleftarrow{QS}(x)$ is $\overleftarrow{QS}(x) = \{(s,p,x)|\exists_{s,p} : (s,p,x) \in Q\}$. We use $\overrightarrow{QS}, \overleftarrow{QS}$ to denote the set of forward and backward graph stars and qs to denote indistinctly a forward and backward query star.*

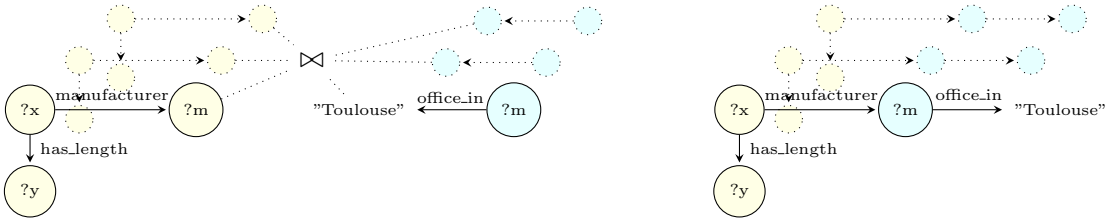
The query execution is then modeled as a sequence of Query Stars in an *Execution Plan* that we depict in Definition 4.2. Indeed, an execution plan for a query is not unique. It is the job of the query optimizer to rank the execution plans according to a given cost function and choose the one with the lowest cost.

Definition 4.2 ***Execution plan*** *An execution plan is an order function applied on a set of Query Stars. The function denotes the order in which the mappings for each Query Star will be found. We denote by $\mathcal{P} = [QS_1, QS_2, ..., QS_n]$ the plan formed by executing QS_1, then QS_2,..., and finally QS_n.*

In order for an execution plan to be acceptable it must fulfill both of the following conditions:

(i) *Coverage:* all the nodes and edges in the query graph must be covered by the sequence of query stars in the plan.

(ii) *Bounded heads:* the heads of all the query stars in the plan, excepting the first one, must have a bounded value before emerging in the plan. This condition avoids to perform a Cartesian product between the matches of two query stars and ensures that the results from the previous query stars prune irrelevant mappings in the query stars. This is illustrated in Figures 4.5a and 4.5b showing an acceptable and a non-acceptable query plan respectively.



(a) $[\overrightarrow{QS}(?x), \overleftarrow{QS}(\text{"}Toulouse\text{"})]$ (b) $[\overrightarrow{QS}(?x), \overrightarrow{QS}(?m)]$

Figure 4.5: Query execution plans example

The generation and selection of the optimal execution plan is out of the scope of this thesis. We only give an overview of the most relevant definitions to give the reader an idea of how the system works. A detailed description of the cost model and the plan selection algorithm is given in [ZMG+20]. In the following section we explain the execution of a graph-based execution plan.

4.2.3.1 Volcano execution model

Let us suppose that the chosen execution plan for a query is $\mathcal{P} = [QS(x_1), ..., QS(x_m)]$. For each query star of the plan, the execution engine must create a list of *candidate graph fragments*. The forward query stars admit only forward graph fragments as part of their candidate sets. Similarly, backward query stars admit only backward graph fragments as candidate sets. This set of candidates was already introduced in Section 3.6 but here we give a more formal definition. A graph fragment is part of the candidate set of a given query star if its characteristic set contains the characteristic set of the query star. More formally, $Cand(QS(x_i), \mathcal{G}f) = \{\mathcal{G}f_i | cs(QS(x_i)) \subseteq cs(\mathcal{G}f_i) \wedge \mathcal{G}f_i \in \mathcal{G}f\}$.

For each candidate graph fragment of a query star, the execution engine searches the relevant triples on the B+tree index used to store the data of each fragment. Since for every triple we stored the location (fragment identifier) of their incoming and outgoing edges, the engine knows exactly to which fragment send the intermediate mappings in the next query star. This is illustrated in Figure 4.6. In this case, the first query star ($\overrightarrow{QS}(?x)$) has two candidate forward fragments ($\overrightarrow{Gf}_{11}$ and $\overrightarrow{Gf}_{12}$). The matches found in these fragments are sent to the candidate fragments of the second query star ($\overrightarrow{QS}(?m)$). Knowing exactly to which graph fragment should be sent the result of the query star is possible because we stored the location of the direct forward and backward neighbors for each triple. Finally, the results are sent to the *Result Writer* without needing an extra coordination step (as it is the case in several systems).

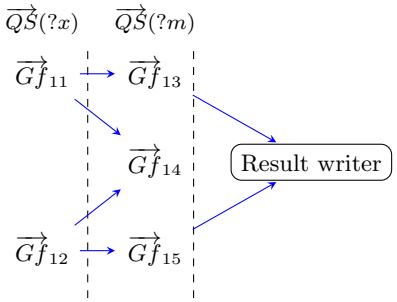


Figure 4.6: Execution pipeline example

The core execution engine is based on the Volcano parallel query evaluation system. The model presented in [Gra94] allows to control the amount of data loaded to main memory avoiding any overflow. RDF_QDAG uses two memory buffers per query star: input and output buffers. The input buffers load the data sequentially from the B+Tree indexes of each fragment. The matching triples are sent to an output buffer that sends the matching data to the following input buffer. This process is illustrated in Figure 4.7.

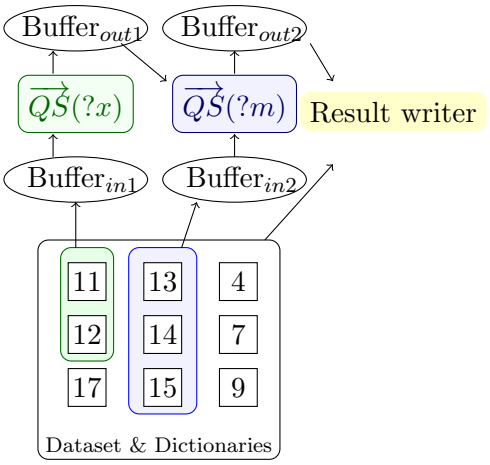


Figure 4.7: Volcano execution in RDF_QDAG example

4.3 Loading costs

The identification and formulation of forward and backward graph fragments in RDF_QDAG is certainly a more complex process than the one applied by other systems loading the data directly to a single relational table for instance. The dataset is read several times to identify and encode the data with the desired format. In this section we compare the pre-processing time of RDF systems with different storage and execution paradigms. We compared the loading module of RDF_QDAG against relational, graph and cloud-based systems. We compared them in terms of loading time and also in their ability to scale to large scale datasets in architectures with limited resources. Indeed, we noticed that the experimental comparisons of RDF systems are usually performed in platforms in which the datasets fit up to $3x$ times in main memory (e.g., in [AHKK17]). The loading module of RDF_QDAG is able to partition very large RDF graphs even when the entire data does not fit in the available main memory. The compared systems are: i) Virtuoso [EM09], ii) RDF-3X [NW08], iii) gStore [ZÖC+14] and iv) CliqueSquare [GKM+15] in a single node (from this point represent it as CliqueSquareS). We evaluated them using real and synthetic dataset described below.

4.3.1 Tested datasets

We used two real and two synthetic datasets with diverse sizes. Within the synthetic datasets we have LUBM[1] [GPH05], a customizable data generator describing information related to universities, departments and faculties. Similarly, WatDiv[2] [AHÖD14] is a data generator offering the possibility to customize the entities and associations of dataset triples. WatDiv also offers a SPARQL query generator to vary structural characteristics and selectivities of the tested queries. Among the real datasets, we have the DBLP dataset collecting an extract of the information in this computer science bibliography[3]. This dataset[4] is stored in the N-Triples format connecting research papers, authors and venues. Lastly, the Yago2 dataset[5] gathers information from Wikipedia, Wordnet and GeoNames. We used the version of this dataset published in [AHKK17].

The characteristics of the datasets are summarized in Table 4.1. We generated synthetic datasets with different sizes to test the scalability of the tested loading approaches.

[1] `http://swat.cse.lehigh.edu/projects/lubm/`
[2] `https://dsg.uwaterloo.ca/watdiv/`
[3] `https://dblp.uni-trier.de/`
[4] Available at: `http://dblp.l3s.de/dblp.rdf.gz`
[5] Available at `https://github.com/ecrc/rdf-exp`

Table 4.1: Experimental datasets

Dataset	Triples (M)	Size (GB)	# S (M)	# P (M)	# O (M)
Watdiv100M	109	15	5.21	86	9.76
Watdiv1B	1000	149	52.12	86	179.09
LUBM100M	100	17	16.27	17	12.10
LUBM500M	500	83	81.38	17	60.54
LUBM1B	1367	224	222.21	17	165.29
Yago	284	42	10.12	98	52.37
DBLP	207	32	6.84	27	35.52

M: Millions, GB: Gigabytes

4.3.2 Configuration setup

The tested systems perform the data pre-processing in a single site (the master node in a distributed system with a master/slave architecture). The master site in our system runs a 64-bit Linux with 32GB of RAM, an Intel(R) Xenon(R) Gold 5118 @ 2.30 GHz processor and 2TB of hard disk. As it was previously mentioned, the Data Loading module of RDF_QDAG is coded in Java (only the indexing sub-component loading the data in the B+Tree is coded in C++). We implemented a fragmentation module on Scala that runs on a Spark cluster. However, the tests in this section are performed in the centralized version of the fragmentation module that successfully managed to load all the datasets.

4.3.3 Pre-processing times

The pre-processing results are summarized in Table 4.2 and are illustrated in a bar chart comparing their logarithmic times in Figure 4.8. The loading times of RDF_QDAG are comparable to the ones obtained by other systems loading the data directly to triple tables. Clearly Virtuoso and RDF-3X loaded the data faster, yet the detection and division of data in graph fragments is not a process that makes the loading process a significantly more expensive step. As we will discuss later (in Section 4.4.3), partitioning the data in graph fragments enhances the performance of some queries. In addition, as we have discussed throughout this thesis, the identification of logical entities brings other benefits such as simpler design of other optimization strategies.

Furthermore, the loading module of RDF_QDAG was the only one capable of charging all the datasets given the limited main memory (32GB). The graph-based system gStore was unable to load datasets whose size is greater than the available main memory. The relational-based systems managed to load all the datasets except LUBM1B in which we got a memory overflow error. In addition, RDF_QDAG loaded more efficiently datasets with great number of predicates compared to Cliquesquare. Let us recall that Ciquesquare groups the triples in different files inside the Hadoop Distributed File System (HDFS) based on the triple's predicates. When the number of predicates augments, the partitioning becomes less efficient.

Table 4.2: Loading times comparisons

Dataset	Loading times (minutes)				
	Virtuoso	RDF-3X	gStore	CliquesquareS	RDF_QDAG
Watdiv100M	45	27	382	107	71
Watdiv1B	188	329	✗	✗	1,080
LUBM100M	284	20	✗	143	348
LUBM500M	1,390	114	✗	✗	1,490
LUBM1B	✗	✗	✗	✗	3,320
Yago	608	520	✗	✗	815
DBLP	97	59	✗	184	127

✗: Unable to load

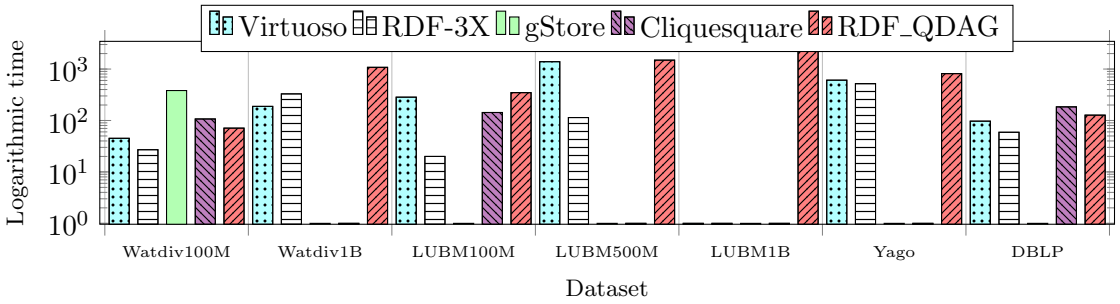


Figure 4.8: Logarithmic loading times

4.4 Evaluation of the fragmentation strategies

In this section we present an evaluation of the use of forward and backward graph fragments initially as logical entities and then as physical structures in RDF processing systems. We begin in Section 4.4.1 by verifying that the number of fragments generated for each dataset is reasonable. We show how many fragments are generated and what percentage of the dataset they represent. In addition, we show the number of fragments needed to cover the eighty percent of the data. This is to show that the groups of graph fragments are a large picture of the input data and could be later used to propose other optimization strategies. Then, in Section 4.4.2, we compare the performance of systems storing physically the data as forward or as backward graph fragments. We analyze the types of queries for which each fragmentation strategy works best. We named the section *exclusive comparison* since we compare separately forward and backward fragmentation strategies. Finally, in Section 4.4.3 we compare the performance of RDF_QDAG, which uses both types of fragments simultaneously, with representative systems of the state of the art.

4.4.1 Data coverage

The results of this section are shown in Table 4.3. The forward and backward graph fragments are obtained using $\tau = 1$ and considering the structural similarity measure. In this way, we analyze the boundary cases with the maximum number of graph fragments. For all the real datasets and for Watdiv, the number of forward graph fragments is greater that the number of backward graph fragments. This is due to the fact that in average, the outdegree of the vertices in these datasets is larger than their indegree. Thus, there are more possibilities to combine predicates as forward fragments and the number of outward characteristic sets is greater than the inward ones. For both synthetic datasets, the number of graph fragments covering 80% of the data is almost invariable with respect to the size of the dataset. This makes sense because our

Table 4.3: Data coverage per dataset

Dataset	$\overrightarrow{\mathcal{G}f}$ N^o of fragments		$\overleftarrow{\mathcal{G}f}$ N^o of fragments	
Watdiv100M	39,855	(1,028)	1,181	(17)
Watdiv1B	96,344	(1,030)	4,725	(19)
LUBM100M	11	(6)	13	(8)
LUBM500M	18	(9)	17	(9)
LUBM1B	36	(19)	33	(18)
Yago	25,511	(100)	1,216	(24)
DBLP	247	(10)	26	(14)

( ): Number of fragments covering 80% of the data

fragmentation strategy allows to detect the seed entities used to generate more data. Actually, the dataset's size is increased varying slightly the features of these seed entities identified as forward graph fragments ($\approx$ 1030 in Watdiv and $\approx$ 10 in LUBM). LUBM has a very limited number of predicates, and being a synthetic database, the generated data is not very variable. This influences definitely the number of forward and backward fragments found for this dataset (less than 40 fragments in the dataset with more than a billion triples). Overall, the number of fragments is always finite and reasonable with respect to the size of the dataset they represent. As we will see later when analyzing the allocation strategies, the use of fragments that hold 80% of the data facilitates the use of graph partitioning algorithms to decide the location of the fragments in the system.

4.4.2 Exclusive comparison of fragmentation strategies

The objective of this section is to analyze the impact of organizing the data in forward or backward graph fragments and to determine which type of fragmentation strategy is more pertinent for a query type. Let us recall that RDF_QDAG stores the data in forward and backward graph fragments simultaneously. At query runtime, the system moves through the inward (or outward) edges of each node in the graph to find matches for a given query pattern. This search is made very efficiently thanks to data partitioning in forward and backward graph fragments that include information of the direct neighbors of a node. However, in this section we are interested in evaluating both partitioning strategies separately. For this we use two scenarios, the first one described in Section 4.4.2.1 built in a relational database and the second one in Section 4.4.2.2 using a graph-based system.

4.4.2.1 Incorporating our framework to a relational-based centralized store

In this section we evaluate the influence of physically storing the data as backward graph fragments in a relational database. The results described in this section are part of the experimental study of the article *Reverse Partitioning for SPARQL queries* [GMBO19]. We stored RDF datasets into a relational database using three different strategies: i) single big table of three columns (subject, predicate, object) similar to RDF-3X's and Virtuoso's strategy, ii) vertical partitioning (one table per predicate) similar to the strategy applied by SW-Store [AMMH09] and iii) applying our reverse partitioning strategy gathering the data by incoming edges. We implemented a customized query parser transforming SPARQL statements into SQL in accordance with each of the previous schemas. We preferrred to create the query translator instead of using the systems implementing the previous approaches to evaluate strictly the influence of the storage configurations. We evaluated on each schema the execution time of queries with different forms (The tested queries are available in the Appendix B).

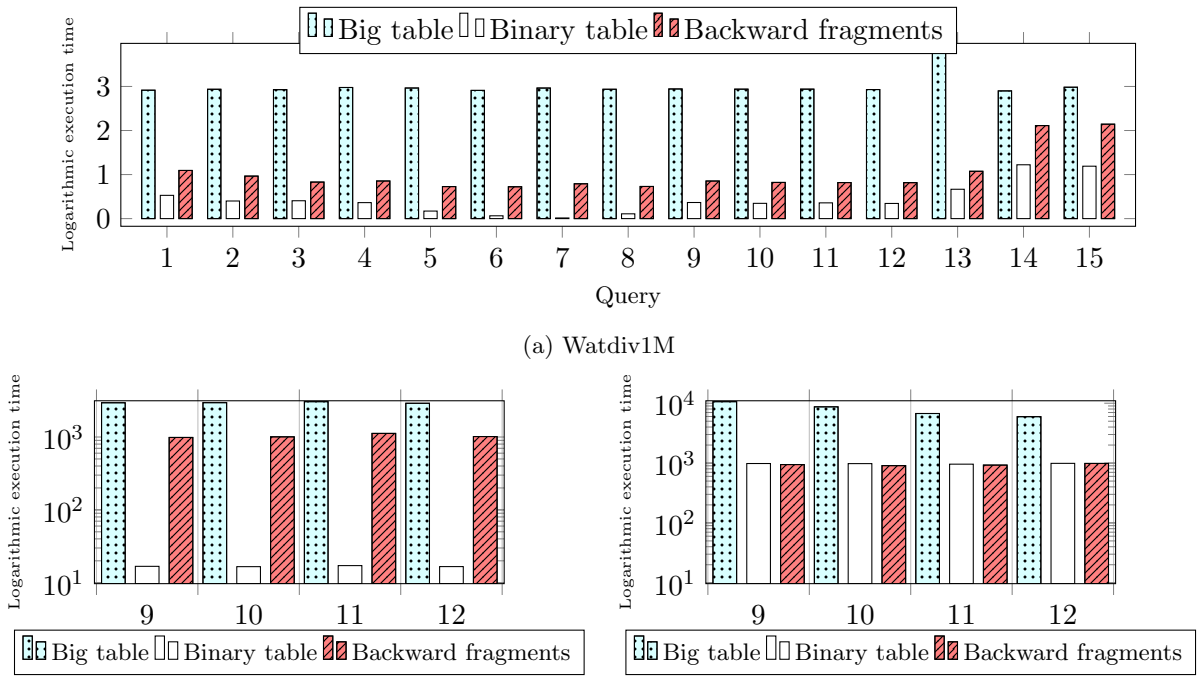
Table 4.4: Experimental datasets in relational-based system

Dataset	Size (GB)	#S	#P	#O	$\#\overleftarrow{\mathcal{G}f}$
Watdiv1M	0.148	52,505	87	105,492	222
Watdiv10M	1.54	521,585	87	1,003,136	587
Watdiv20M	3.28	1,042,785	87	2,473,723	641

M: millions

Configuration setup The partitioning module derived from the RDF_QDAG loading module is implemented in Java. The translation module from SPARQL to SQL was implemented also in Java and the data were stored on PostreSQL 11. The experiments were performed in the master node of the hardware described in Section 4.3.2. We tested our approach with the WatDiv framework since it offered a wide variety of queries. We generated datasets of 1, 10 and 20 million triples. More details about the datasets are shown in Table 4.4. For each of them we generated 80 queries (20 of each query type).

Experimental results The results are shown in Figure 4.9. Creating vertical partitions on the predicates gives the most performant execution times for the majority of queries considering that there was not an intense intermediary indexing strategy as it is the case for RDF-3X. The major drawback of the vertical partitioning strategy is data distribution on the tables. The performance of the data stored as backward graph fragments is almost as good as the vertical partitioning, especially when the dataset size is bigger and exploring a single table becomes more costly. Backward graph fragmentation performance suffers in queries with patterns in which the subject and object are unknown. In a relational-based system, storing the data in graph fragments is ideal to prune queries with patterns with bounded values at the object position. This is the case of queries 9-12.



(a) Watdiv1M

(b) Watdiv10M

(c) Watdiv20M

Figure 4.9: Performance of partitioning configurations in relational-based system

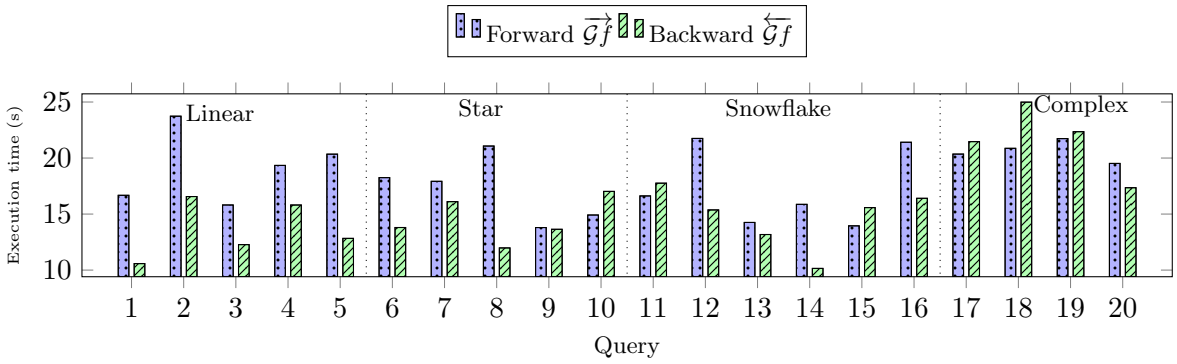


Figure 4.10: Query performance of forward and backward graph fragments

4.4.2.2 Incorporating our framework to a graph-based distributed store

In this section we compare the performance of organizing the data as forward or as backward graph fragments in a graph-based triple store. We determine for which type of queries organizing the data as forward graph fragments is more suitable than backward graph fragments and vice versa. To the best of our knowledge, there is no graph-based RDF system storing the data exclusively as backward graph fragments. Indeed, RDF_QDAG uses both structures (i.e. $\overrightarrow{\mathcal{G}f}$ and $\overleftarrow{\mathcal{G}f}$) and gStore stores the data in adjacency lists grouping the data by subjects. For the experiments in this section, we adapted the distributed version of gStore [PZÖ+16] to support the storage of forward graph fragments first and then of backward graph fragments. The results in this section are part of the experimental study of our framework: *RDFPartSuite: Bridging Physical and Logical RDF Partitioning* published in [GMB19].

Configuration setup gStoreD was deployed on a 5 machine cluster connected by a 10Gbps Ethernet switch. The cluster runs a 64-bit Linux and each site has a 8GB RAM, a processor Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz and 100GB of hard disk. The latest version of gStoreD[6] was configured to store the data on adjacency lists on the subjects, and a modified version storing the data on adjacency lists on the objects. The adjacency lists on the subjects are the default structures employed by gStore's V*-Tree index. At the loading stage, the user indicates in a file the location for each subject (data star head in our notation, cf. Definition 3.1) and all the triples with that subject are assigned to that location id. We make sure that the triples belonging to the same forward graph fragment are located in the same partition. Then, to load the data in adjacency lists by objects we slightly change the input file containing the raw RDF data. The data is injected to gStore in a file in *N-Triples* format whose triples are ordered as *(subject predicate object)*. To force the creation of adjacency lists from the object, we change the order of the input file to *(object predicate subject)*. We changed the file with the subject's location (giving the location for each distinct object) and the SPARQL queries accordingly. For example, a query pattern *?x p ?y* in the original query list is changed to *?y p ?x* in the queries used to test backward fragments. We performed the evaluation with the four query types (linear, star, snowflake and complex queries) of the Watdiv benchmark with 10 million triples. The list of queries is found in the Appendix B.

Experimental results The results are shown in Figure 4.10. The system has much better performance when the data are organized as backward graph fragments when solving linear queries. It performed in average 1.4x faster than when the data was organized as forward fragments. For star queries, the use of backward fragments enhanced up to 1.2x in average the performance. For snowflake shaped queries, there is no strategy that works better than the

[6]Available at `https://github.com/bnu05pp/gStoreD`

other. We found queries in which backward graph fragments worked better because there was a bounded value in the object position of the query, but if this condition was not fulfilled forward fragments worked better. Organizing the data as forward fragment was definitely better to solve the workload of complex queries (up to 1.2x faster for query 18). Both strategies complement each other in the processing of certain query types. In the following section we show that the combination of both structures in RDF_QDAG drives RDF data processing in centralized systems efficient and scalable.

4.4.3 Combining fragmentation strategies

In this section we present the results of an experimental study comparing RDF_QDAG with three representative systems of the state of the art. We compared them in terms of query performance (for BGP queries) and scalability. The compared systems are:

(i) *Virtuoso*[7] [EM09]: it is the relational-based system by excellence storing the data in a triple table.

(ii) *RDF-3X*[8] [NW08]: intensive index-based system able to scale to billions of triples. Used as baseline of many optimization approaches (e.g., characteristic sets [NM11]).

(iii) *gStore*[9] [ZÖC+14]: graph-based system storing the data in adjacency lists.

To evaluate their scalability we used datasets of several millions of triples. Specifically we used Watdiv100M, Watdiv1B, LUBM500M, Yago and DBLP whose characteristics are described in Section 4.3.1. The testes were performed on the master node of our testing platform described in Section 4.3.2. The results presented in this section are an excerpt from our paper describing RDF_QDAG main components in [KMG+20]. The queries are in the Appendix B of this thesis. For all the experiments, we run the queries 3 times (excluding the first cold start run time to avoid the warm-up bias).

Let us start discussing the results of the queries performed in the Watdiv benchmark shown in Table 4.5. As it was explained when we compared the loading times in Section 4.3.3, gStore was unable to load Watdiv1B. Furthermore, at query runtime, gStore loads the V*-Tree index to main memory before looking for query matches slowing down the execution of most of the queries. Complex queries in Watdiv100M are solved in average $5.3x$ faster in RDF_QDAG compared to Virtuoso and RDF-3X. For the dataset of more than a billion triples (i.e. Watdiv1B), RDF_QDAG is clearly faster to solve complex queries ($130x$ faster in average). For snowflake queries in the Watdiv100M dataset, Virtuoso, RDF-3X and RDF_QDAG performed very similarly. However, the same queries in the dataset of 1 billion triples were solved $4.9x$ and $12.8x$ faster in RDF_QDAG than in RDF-3X and Virtuoso respectively. Linear queries were solved $0.4x$ and $45x$ faster in our system than RDF-3X and Virtuoso for Watdiv1B, and all three performed similarly in Watdiv100M. Finally, star queries are solved faster in Virtuoso in the smaller dataset, but in average $4.9x$ faster than RDF-3X and $12.8x$ with respect to Virtuoso in the 1 billion dataset.

The other synthetic dataset, LUBM500M, whose results are shown in Table 4.6, could not be loaded by gStore for the same reasons as the previous one. RDF-3X performed slightly better for queries 8, 11, 12 and 14, that are snowflake shaped queries. Still, in all the other queries RDF_QDAG outperformed both systems solving queries $9.8x$ and $1.1x$ faster than Virtuoso and RDF-3X respectively.

[7] `http://vos.openlinksw.com/owiki/wiki/VOS`
[8] `https://gitlab.db.in.tum.de/dbtools/rdf3x`
[9] `https://github.com/pkumod/gStore`

Table 4.5: Watdiv BGP queries results in seconds

		Complex			Snowflake					Linear					Star			
		1	**2**	**3**	**1**	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**
Watdiv100M	**gS**	93.9	92.1	96.4	91.3	93.1	97.8	88.9	91.9	93.8	94.4	92.3	93.3	95.2	91.2	89.2	85.0	93.8
	3X	12.9	**0.7**	66.1	**0.1**	0.7	0.8	1.5	0.01	**0.1**	1.1	0.03	0.8	0.6	**2.0**	0.3	0.9	4.8
	V	13.3	2.2	69.3	0.4	0.7	**0.3**	**0.2**	0.3	0.3	**0.1**	0.01	**0.3**	**0.1**	✗	**0.1**	**0.2**	20.3
	QD	**1.4**	1.7	**45.0**	0.3	**0.4**	**0.3**	0.4	**0.001**	**0.1**	1.1	**0.001**	1.1	0.4	5.6	0.7	0.88	**1.9**
Watdiv1B	**3X**	6818.2	16.1	694.2	**0.4**	14.0	12.7	10.3	1.3	0.4	56.1	0.5	21.0	**19.1**	173.7	6	11.6	267.0
	V	152.4	111.9	921.8	4.7	56.2	10.4	**8.0**	4.6	33.7	**2.3**	30.8	33.1	37.6	✗	**1.5**	**1.8**	697.5
	QD	**9.1**	**13.4**	**382.2**	1.5	**2.2**	**4.6**	1.5	**0.3**	**0.2**	102.0	**0.3**	**19.1**	47.4	**35.9**	29.2	6.4	**15.0**

gS: gStore, 3X: RDF-3X, V:Virtuoso, QD: RDF_QDAG, ✗: Error, **Bold:** lowest execution time

Table 4.6: LUBM500M BGP queries results in seconds

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
3X	0.1	388.4	0.1	0.1	0.1	553.4	0.1	**0.2**	72.7	0.1	**0.4**	**0.2**	250.6	**533.1**
V	0.05	214.9	0.1	0.1	2.8	683.3	0.1	6.7	27.0	0.7	3.8	4.6	326.6	796.9
QD	**0.01**	**139.5**	**0.01**	**0.03**	**0.1**	**550.0**	**0.03**	3	**13.9**	**0.01**	2.6	3.0	**44.6**	550.0

3X: RDF-3X, V:Virtuoso, QD: RDF_QDAG, **Bold:** lowest execution time

Table 4.7: DBLP BGP queries results in seconds

	L1	**L2**	**L3**	**L4**	**S1**	**S2**	**S3**	**S4**	**S5**
3X	4226.37	0.40	0.15	0.10	2687.48	0.19	155.36	**2.68**	439.45
V	✗	0.05	0.48	0.13	✗	2.26	210.78	88.35	405.85
QD	**2023.34**	**0.01**	**0.01**	**0.005**	**537.50**	**0.01**	**145.1**	1.23	**237.9**

3X: RDF-3X, V:Virtuoso, QD: RDF_QDAG, **Bold:** lowest execution time

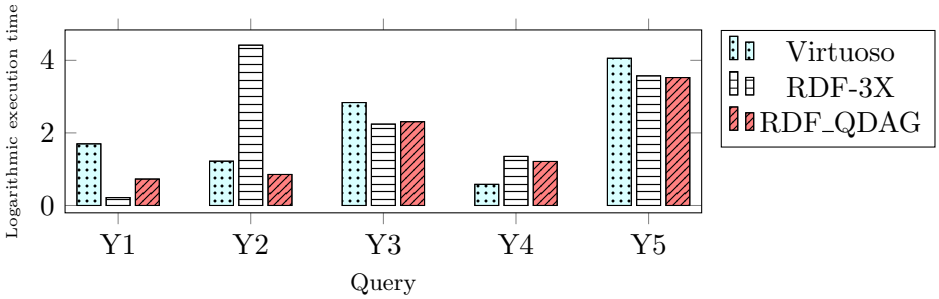


Figure 4.11: Yago BGP queries results

Among the real datasets, we designed the workload of DBLP in queries with different shapes and selectivities. As it is shown in Table 4.7, RDF_QDAG outperformed RDF-3X and Virtuoso for most of the queries ($1.6x$ faster than Virtuoso and $13.8x$ faster than RDF-3X). The queries in which the systems had a similar performance are those very selective (e.g., L1, S1 and S5). Finally, the results of Yago are plotted in a logarithmic scale in Figure 4.11. RDF_QDAG outperformed the three systems and managed to filter faster the results based on the known bounded values of the queries (e.g., query Y2). It achieved up to $4x$ faster performance compared to Virtuoso.

Summary The virtues of the logical partitioning strategy have been shown throughout these sections. Undoubtedly, a logical partitioning strategy grants much more freedom to designers and managers. They do not depend anymore on a specific system to partition their data. We have shown the feasibility of incorporating our framework into centralized and distributed triple stores. Furthermore, the experiments using RDF_QDAG reveal that the use of these structures in a centralized native triple store ensures its scalability and good performance for certain queries.

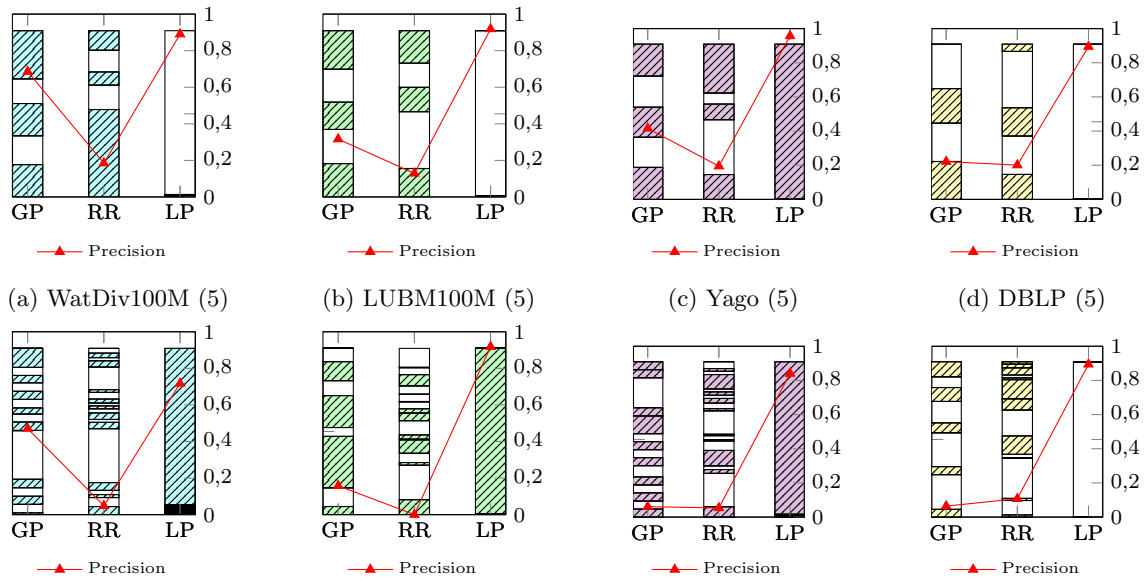
4.5 Evaluation of the allocation strategies

In the previous section we evaluated whether both *fragmentation* strategies proposed in this thesis are effective to cover the completeness of the data in a reasonable manner. Then, we compared their effectiveness at query runtime with other physical organization strategies of the state of the art. In this section, we evaluate the *allocation* strategies proposed to place the forward and backward graph fragments to the sites of a distributed triple store. We start in Section 4.5.1 comparing three allocation strategies in terms of data skewness prior the query execution. Then, in Section 4.5.2 we present an extensive study of the communication costs of the state-of-the-art partitioning strategies versus the allocation heuristic proposed for our logical structures. This study is based on the execution logs of RDF_QDAG. Finally, in Section 4.5.3, we compare the performances of a graph-based system, a cloud-based system and a system that applies the partitioning strategy proposed in this study. Part of the results in this section were published in our work [GMB19] introducing*RDFPartSuite Framework*.

4.5.1 Data skewness comparison

In this section we compare the round-robin, linear programming and min-cut allocation strategies in terms of data skewness. The first strategy assigns the fragments (forward and backward) to the sites in a round-robin manner. The linear programming uses a solver to find the optimal solution for the problem defined in Section 3.5.1. Finally, the min-cut strategy refers to the

graph-based heuristic described in Section 3.5.2. We give as inputs to the linear programming and min-cut strategies only the necessary fragments to cover the 80% of the data (cf. Table 4.3). The other 20% are assigned in a round-robin manner. The results are shown in Figure 4.12 in which we represent the distribution when 5 and 20 partitions are created for some of the datasets described in Section 4.3.1. For each partitioning strategy we show the proportion of the number of triples assigned to each site in a stacked bar and in a secondary axis the precision of the distribution.



(a) WatDiv100M (5) (b) LUBM100M (5) (c) Yago (5) (d) DBLP (5)

(e) WatDiv100M (20) (f) LUBM100M (20) (g) Yago (20) (h) DBLP (20)

Figure 4.12: Triple distribution by allocation method
GP: graph partitioning, RR: rond-robin, LP: linear programming

The linear programming solution, was implemented using the Gurobi[10] optimization library [GO18] . To obtain a solution with a reasonable time, we configured the imbalance parameter in the first constraint of the problem $\epsilon = 500M$ triples. Even with this big tolerance, the majority of the data was allocated to one partition. Also, we validated that the round-robin distribution strategy leads to data skewness problems compared with the graph partitioning heuristic. For example, for LUBM100M, the number of triples stored in the second site is significantly larger to the number of triples in all of the other partitions.

The precision of each partitioning strategy is shown in the secondary axis of the graphs of Figure 4.12. It is calculated as: $\frac{w(e_c)}{w(e)}$, in which $w(e_c)$ and $w(e)$ are respectively the sum of the weights of the cut edges and the total number of edge's weights. In other words, it is the ratio of the number of triples that share a subject (or object) in two fragments located in different sites with respect to the number of triples on the site. The results are shown in the red line of Figure 4.12. As expected the higher precision is obtained by the linear program, however the scalability on this program is very low and in the ends puts most of the data to a single site. The min-cut partitioning algorithm, used by systems like EAGRE [ZCTW13] to distribute forward fragments, gets better results than the round-robin strategy, not only in terms of data distribution but also on the precision.

4.5.2 Communication costs study

In this section, we present an extensive study of the communication costs induced by different allocation strategies. To evaluate fairly and isolate the impact of an allocation strategy at query

[10] Availabe at: `https://www.gurobi.com/downloads/gurobi-optimizer-eula/`

runtime, we run the queries on the same execution engine. We measured precisely for each query the number of intermediate results exchanged. For this purpose, we use the execution logs of RDF_QDAG [KMG+20]. Even though it is a centralized system, we collected for each query the number of data sent from one fragment to another as it was explained when we described Figure 4.6. Then, we calculate the total mappings exchanged between fragments in different sites.

We analyzed the communication costs for the following allocation techniques:

- *Hash on subject (H)*: as shown in the Related Work section, several systems follow this strategy applying a hashing function on the triple's subject.

- *Linear programming (LP):* the problem formalized in Section 3.5 was programmed and solved with the Gurobi [GO18] optimization library as in the previous section.

- *Graph partitioning (GP)*: the data are firstly grouped in graph fragments that are distributed using a graph partitioning heuristic (METIS [KK98a]).

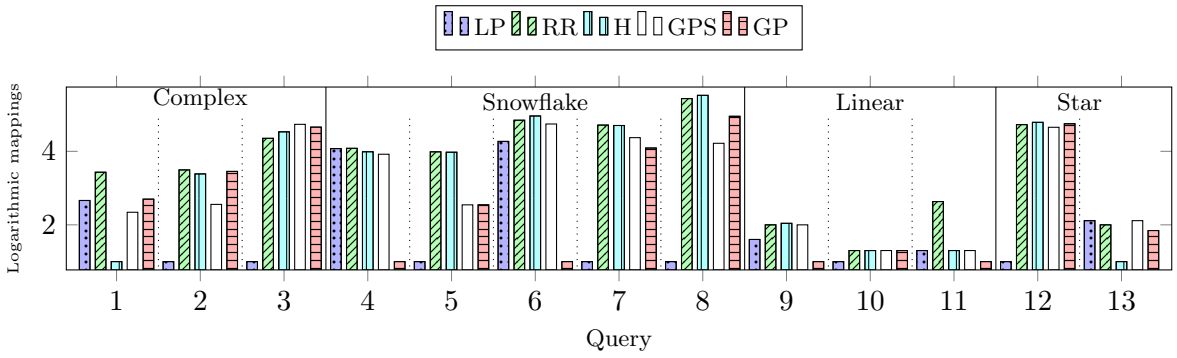
- *Graph partitioning (small groups) (GPS)*: in this strategy the size of the graph fragments were restricted to form as many fragments as possible. Then these smaller fragments are distributed with a graph partitioning heuristic. We seek to see whether regulating the size of the partitions influence in the overall performance of the system.

- *Round robin allocation (RR)*: data firstly grouped in graph fragments are distributed in a round-robin manner.

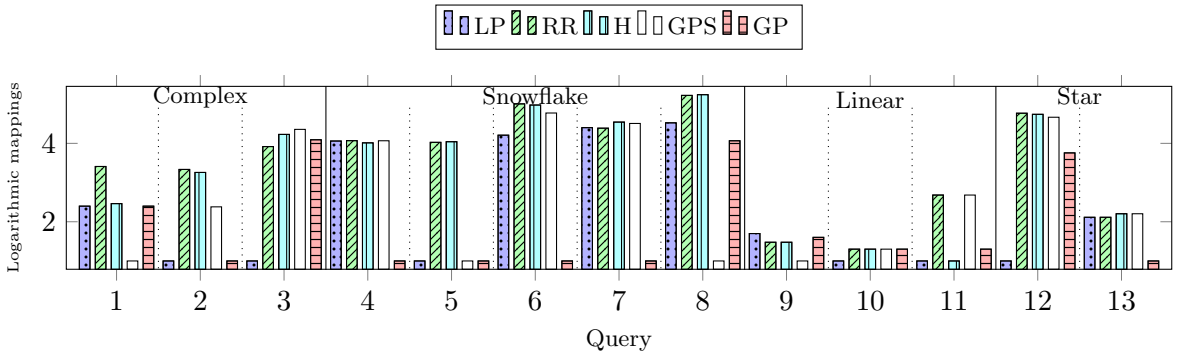
All of the previous allocation strategies were evaluated in the Watdiv benchmark since it offered queries with different structures (complex, snowflake, linear and star). We generated a dataset with about 100k triples to guarantee that the linear program finds a solution in a reasonable time. We partitioned the data in 5, 10 and 20 sites to evaluate the performance of the solutions as the number of sites grows. The results comparing the number of mappings exchanged by fragments in different sites for each query are shown in Figure 4.13.

The linear programming strategy (identified as LP in the graphs) is, as expected, the one with the lowest number of mappings exchanged between sites in most of the queries. There are still some queries where the graph partitioning strategy with smaller groups (GPS) or hashing by the subjects performed better. This is because the function to optimize in LP considers moving the initial graph fragments from one site to another, but does not consider making smaller fragments. For those queries, making smaller fragments reduced the number of exchanged mappings (e.g., queries 1 and 13 in the case of 5 partitions). The hashing strategy performed well for linear queries but it induced several exchanges for the other query types. In general, for linear queries we did not find an absolute winning strategy, since the queries are quite simple the allocation strategy did not varied considerably the number of mappings. The round robin allocation strategy was the one with the highest number of mappings for all of the query types in the tests with 5, 15 and 20 partitions. Furthermore, the graph partitioning strategy that does not modify the size of the groups offers a reasonable performance without modifying the sizes of the loaded groups. This strategy worked better in snowflake and star-shaped queries. Another interesting finding was that the number of generated partitions (e.g., 5, 10 or 20) did not influence the number of mappings sent between sites.

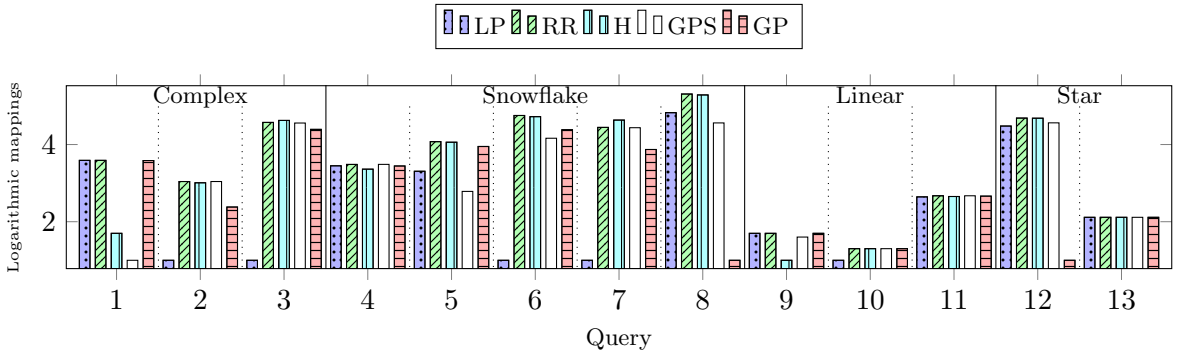
Our experiments show that allocation strategies like graph-partitioning heuristics are a good compromise to reduce the number of data interchange at query runtime. Still, if the queries are mostly simple linear queries, a hashing function has shown to have also an advantageous performance. Another essential insight is that regulating the size of the groups of graph fragments is crucial to achieve a good performance, as it was proven by the number of exchanged results in complex queries.



(a) 5 partitions



(b) 10 partitions



(c) 20 partitions

Figure 4.13: Mappings analysis for Watdiv100k
LP: Linear programming, RR: round-robin, H: hashing on the subject, GPS: graph partitioning - small groups, GP: Graph partitioning

4.5.3 Distributed experiments

We used synthetic and real datasets to compare three systems with distinct partitioning and execution strategies. Contrarily to the experiments detailed in Section 4.4.3, we compare RDF_-QDAG with two distributed triple stores. Our goal is not to compare the performance of systems storing physically the data with different structures as in Sect. 4.4.3. Here, our focus is to compare systems *distributing* the triples with different strategies. The compared distribution strategies are: i)cloud-based, ii)simple hashing and iii)our allocation strategy of graph fragments. The tested systems are:

(i) *gStoreD* [PZÖ+16]: graph-based system partitioning each node by subject. We use as well the centralized version of this system for reference.

(ii) *Cliquesquare* [GKM+15]: cloud-based system fragmenting the data by predicates. We configured the system in a single and multi node modes.

(iii) *RDF_QDAG* [KMG+20]: centralized system partitioning the data in forward and backward graph fragments.

4.5.3.1 Configuration setup

The experiments were conducted on a cluster of 5 machines connected by a 10Gbps Ethernet switch. The cluster runs a 64-bit Linux and each site has 16GB (32GB in the master node) of main memory, a processor Intel(R) Xenon(R) Gold 5118 @ 2.30 GHz and 100GB (2TB in the master node) of hard disk. As it was done in the communication study, we use Yago and DBLP as real datasets. We use Watdiv100M and LUBM100M as synthetic datasets. Also, to be able to compare with gStore which was unable to load LUBM100M, we also tested in LUBM with 20 million triples.

4.5.3.2 Experimental results

The results are shown in Tables 4.9-4.10. We use the following abbreviations for each system: Cliquesquare in a single node (Cs), Cliquesquare in multiple nodes (CsD), gStore (gS), distributed gStore (gSD) and RDF_QDAG. In the tables, we show the execution time in seconds, highlighting with bolds the best results for each query. Below we discuss the results for each dataset.

Watdiv For most of the queries, the execution time of RDF_QDAG is up to 60x less than all the other systems. It performed on average 400x, 40x, 348x, and 40x faster for linear, star, snowflake and complex queries respectively. However, for the complex query C3 which involves several graph patterns without any subject or object known was performed almost 2x faster by gStore. This dataset fitted in main memory and therefore, for in-memory systems (like gStore) the matching processing is faster. Especially in queries that are not very selective or that do not have a bounded subject or object that could be used by RDF_QDAG to prune. RDF_QDAG processing is based on the Volcano execution model, exchanging data between the disk and main memory to guarantee scalability and the same time ensuring a good performance. As it is shown next in the real datasets, gStore was not able to treat these datasets whose size is bigger than the available main memory.

LUBM The results for both datasets are shown in the Table 4.9. The system RDF_QDAG outperformed all the other tested systems (centralized or distributed versions) in most of the queries for both dataset sizes. As shown by the results in the Table 4.9, for the 20 million triple dataset Cliquesquare (single or multinode) is the slowest system due to the starting cost of the MapReduce engine which does not offset the performance gain of parallelism. Even if gStore

systems (centralized and distributed) are able to load the 20M dataset, RDF_QDAG performs better for all queries except the sixth which is a simple query but not very selective. However, the same query in the dataset of 100 million triples is solved 15x faster by RDF_QDAG. For the dataset of 100 million triples, RDF_QDAG is faster in almost 80% of the queries. The queries (2, 5, 6, 14) for which Cliquesquare performed in average 2.4x faster than RDF_QDAG were simple queries not selective with many results. In this case, the parallel treatment offered by MapReduce in Cliquesquare works best. Let us remind that RDF_QDAG partitioned the graph but worked in a centralized configuration, the parallel version of this system is part of our future perspectives.

Yago and DBLP The results are shown in Table 4.10. Cliquesquare and the centralized version of gStore were unable to process the tested queries. RDF_QDAG outperformed gStoreD by almost 1000x. Concerning DBLP, the distributed versions of gStore and Cliquesquare as well as RDF_QDAG were able to process queries on this dataset. RDF_QDAG performed in average 500x better than all the tested systems.

Summary These experiments complement those that evaluated data fragmentation. We show the advantages of our allocation strategies in terms of balanced data distribution and query performance. Our experimental study showed that incorporating our logical partitioning strategies to a centralized triple store can make them more performant than many of the current distributed systems. Our partitioning strategies enable a graph-based system (RDF_QDAG) to scale contrary to other centralized graph-based systems like gStore.

Table 4.8: Execution time (in seconds) for Watdiv100M queries

Sys.	C1	C2	C3	F1	F2	F3	F4	F5	L1	L2	L3	L4	L5	S1	S2	S3	S4	S5	S6	S7
Cs	124.5	181.1	138.4	✗	181.0	75.7	59.1	104.4	51.5	51.8	40.9	41.8	50.5	57.8	47.9	✗	40.6	46.5	42.7	44.2
CsD	69.9	93.0	91.7	✗	92.0	81.7	163.9	✗	30.4	30.3	26.4	26.3	31.4	125.7	37.5	✗	26.4	36.5	27.6	27.5
gS	93.9	92.1	96.4	91.3	93.1	97.8	88.9	91.9	93.8	94.4	92.3	93.3	95.2	91.2	89.2	90.8	88.8	90.5	85.0	93.8
gSD	139.8	55.0	**24.0**	22.0	33.0	29.3	59.4	73.1	27.6	22.6	21.9	22.8	23.3	22.9	21.8	21.8	21.8	21.9	21.9	21.8
QDAG	**1.36**	**1.68**	45.05	**0.27**	**0.44**	**0.63**	**0.41**	**0.001**	**0.09**	**1.15**	**0.001**	**1.11**	**0.45**	**5.57**	**0.78**	**0.29**	**0.95**	**0.24**	**0.76**	**1.91**

Table 4.9: Execution time (in seconds) for LUBM queries

Ds.	Sys.	1	2	3	4	5	6	7	8	9	10	11	12	13	14
20M	Cs	42.4	76.3	43.8	45.0	40.4	46.9	120.8	104.6	155.8	42.4	49.1	38.4	42.4	47.8
	CsD	38.3	67.6	159.2	29.1	28.0	31.0	505.1	49.2	108.4	29.0	31.1	27.0	29.0	30.0
	gS	26.0	25.0	26.2	26.5	32.2	47.9	25.9	25.7	26.6	25.3	25.7	26.8	34.2	49.0
	gSD	10.1	883.1	9.8	9.8	9.8	**17.1**	29.7	353.0	33.9	9.9	12.9	9.9	14.1	**17.3**
	QDAG	**0.01**	**6.0**	**0.01**	**0.02**	**0.09**	19.5	**0.03**	**0.47**	**0.99**	**0.004**	**0.02**	**0.015**	**7.9**	18.4
100M	Cs	73.3	143.8	86.4	81.5	45.9	87.7	388.6	301.8	600.9	78.7	56.6	39.5	40.9	89.8
	CsD	57.1	**67.5**	45.0	43.2	**30.0**	**43.0**	146.1	122.3	291.4	41.1	32.0	26.9	27.9	**43.1**
	gS	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	gSD	2684	✗	2815	1818	1801	1912	11063	22439	8747	3312	3455	1639	1834	1920
	QDAG	**0.005**	173.9	**22.5**	**4.1**	35.6	121.9	**0.039**	**1.2**	**0.05**	**0.006**	**1.04**	**0.80**	**12.9**	123.8

Table 4.10: Execution time (in seconds) for DBLP and YAGO queries

Ds.	Sys.	C1	C2	C3	L1	L2	L3	L4	S1	S2	S3	S4	Ds.	1	2	3	4	5
DBLP	Cs	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	**Yago**	✗	✗	✗	✗	✗
	CsD	✗	✗	278.5	**1365**	37.5	**29.4**	37.6	**424**	38.7	**98.5**	35.5		✗	✗	✗	✗	✗
	gS	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗		✗	✗	✗	✗	✗
	gSD	**41556.3**	**4458**	2779	✗	1767	1556	1483	✗	1649	2264	1692		579.84	417.59	933.90	373.26	399.70
	QDAG	✗	✗	**238**	2023	**0.01**	45.9	**0.01**	896	**0.01**	158.9	**2.81**		**0.54**	**0.07**	**20.35**	**1.63**	**335.20**

Dataset (Ds.), System (Sys.), Cliquesquare in a single node (Cs), Cliquesquare in multiple nodes (CsD), gStore (gS), distributed gStore (gSD) and RDF_QDAG (QDAG). The **bold** cells represent the lowest execution time and ✗ that the system was unable to perform the query.

4.6 Partitioning language

The creation of forward and backward graph fragments before partitioning a raw RDF file contributes to integrate a logical dimension to the purely physical partitioning process. This logical layer allows designers to deal with much more manageable structures than the one offered by single triples. These structures can be used not only by system's developers but also by Triple Store managers. In this section, we present a user-friendly extension to the RDFPartSuite framework. We propose a declarative partitioning language that allows designers, based on their expertise level, to manually partition a KG.

The declarations are done similarly to the ones of tables and partitions with a DDL (data definition language) available in some database management systems (e.g., Oracle[11], PostreSQL[12]). The language uses the logical structures and hides the implementation of the fragmentation and allocation algorithms presented in the previous chapter. In the following section we define the grammar and syntactical elements of this language. RDFPartSuite keeps a meta-data file with information related to the managed KGs. This includes, their names, fragments and predicates identifiers. The specifications of the infrastructure used by the triple store (number of sites, available space on each site, IP of each site, etc.) are also kept in a separate file by the framework.

4.6.1 Notations

Before defining the grammar, let us consider the notation used to define the syntax of the definition language. This declarative language is not case-sensitive. While describing the grammar we used some commands in uppercase just to facilitate their readability.

- ⟨element⟩: nonterminal element;
- `element`: terminal element;
- [element]: optional element;
- {element}: is an element that can be repeated 0 or n times;
- | : represents an alternative;
- ⟨element list⟩ is an element defined as ⟨element⟩{, ⟨ element ⟩}.

The main lexical elements are described as follows:

⟨non quote characters⟩	::=	{a-z}
⟨quote characters⟩	::=	'{a-z}'
⟨unsigned integer⟩	::=	{init digit}[{digit}]
⟨unsigned float⟩	::=	{init digit}[{digit}].{⟨digit⟩}
⟨digit⟩	::=	0|1|2|3|4|5|6|7|8|9
⟨init digit⟩	::=	1|2|3|4|5|6|7|8|9

The language is composed of seven main statements from the declaration of the KG to the allocation of graph fragments to the sites of the system. Each of these statements is described next.

[11] `https://docs.oracle.com/cd/E18283_01/server.112/e16541/part_admin001.htm`
[12] `https://www.postgresql.org/docs/10/ddl-partitioning.html`

4.6.2 `CREATE KG` statement

This statement creates an empty Knowledge Graph and stores the name in a metadata file maintained by the framework. If the *KG* exists already (identified by the *kg name*), it sends back an error.

⟨kg definition⟩ ::= `CREATE KG` ⟨kg name⟩
⟨kg name⟩ ::= ⟨non quote characters⟩

Let us consider for example the following statement declaring a *KG* named *yago*.

```
CREATE KG yago
```

4.6.3 `LOAD DATA` statement

This statement indicates that the data in a file (in .NT or .ttl format) belongs an existent knowledge graph.

⟨load data⟩ ::= `LOAD DATA INFILE` ⟨file name ⟩ `INTO` ⟨kg name⟩
⟨file name⟩ ::= ⟨non quote characters⟩

In the previous statement, the *filename* is the raw RDF file stored in the *working directory*[13] of the framework. For example, here we state that the triples of the file `yago-knowledge.nt` belong to the yago *KG* declared previously.

```
LOAD DATA INFILE yago-knowledge.nt INTO yago
```

4.6.4 `FRAGMENT KG` statement

This statement fragments the data in an existent *KG* in Forward or Backward graph fragments.

⟨fragment kg⟩ ::= `FRAGMENT KG` ⟨kg name⟩ `FORWARDLY` [⟨partition options⟩]
| `FRAGMENT KG` ⟨kg name⟩ `BACKWARDLY` [⟨partition options⟩]
⟨partition options⟩ ::= `BY` ⟨fragment mode⟩ [⟨max frag⟩][⟨similarity⟩]
⟨fragment mode⟩ ::= `SUPERSET`
| `JACCARD`
| `TFIDF`
| `LABEL`
| `ANCESTOR`
⟨max frag⟩ ::= `MAX` ⟨unsigned integer⟩
⟨similarity⟩ ::= `SIM` ⟨unsigned float⟩

The language allows to partition a *KG* (identified by *kg name*) in Forward or by Backward graph fragments. The structural partitioning (using characteristic sets only), a similarity threshold of 1, and the maximum size of 50 million triples per partition are the default parameter values. The fragmentation mode can be changed to either of the modes presented in Section 3.3.1.1. Also, it is possible to limit the maximum size of fragments (in number of triples) using the keyword `MAX`. Finally, it is also possible to change the similarity threshold (`SIM`) that is

[13] Indicated in a configuration file

defined in Sect 3.3.1.1. This value is a float between 0 and 1. If the user enters a similarity threshold greater than 1, the framework identifies it as an error and abort the command.

For example, let us consider that the *yago KG* declared previously. is partitioned by forward fragments, using the JACCARD MODE, restricting the number of triples in a fragments to 10,000 and choosing the similarity threshold to 0.8.

```
FRAGMENT KG yago FORWARDLY BY JACCARD MAX 10000 SIM 0.8
```

4.6.5 ALTER FRAGMENT statement

It is possible to re-split a given fragment because it is too large with the methods described in Sect. 3.7.

⟨alter fragment⟩	::=	`ALTER FRAGMENT` ⟨frag id⟩ `IN` ⟨kg name⟩ `SUBPARTITION BY` ⟨subpartition mode⟩
⟨frag id⟩	::=	⟨unsigned integer⟩
⟨subpartition mode⟩	::=	`HASH`
	|	⟨range⟩
⟨range⟩	::=	`RANGE` ⟨pred id⟩ `LESS THAN` ⟨range value⟩
⟨pred id⟩	::=	⟨unsigned integer⟩
⟨range value⟩	::=	⟨unsigned integer⟩
	|	⟨unsigned float⟩

All the fragments and predicates are identified with an integer identifier (*frag id* and *pred id* respectively) automatically generated by RDFPartSuite. A fragment can be sub partitioned using two main modes: hashing function or range partitioning as defined in Section 3.7.

Let us consider the example in which we want to partition a fragment in the *yago KG* identified with the $ID = 345$. The fragment is partitioned in two parts according to the values of the predicate identified with the $ID = 8$ (whose range are integers).

```
ALTER FRAGMENT 345 IN yago SUBPARTITION BY RANGE 8 LESS THAN 4000
```

4.6.6 ALLOCATE statement

This statement allocates the fragments of the previous instruction according to a round-robin rule or using the graph partitioning heuristic described in Section 3.5.2.

⟨allocate fragments⟩	::=	`ALLOCATE KG` ⟨kg name⟩ `BY` ⟨allocation mode⟩ ⟨nb sites⟩
⟨allocation mode⟩	::=	`ROUND-ROBIN`
	|	`METIS`
⟨nb sites⟩	::=	`SITES` ⟨unsigned integer⟩

For example, in our running example, we can allocate the fragments in the *yago KG* using the graph heuristic in 5 sites.

```
ALLOCATE yago BY METIS SITES 5
```

4.6.7 ALTER ALLOCATION statement

This statement allocates a given fragment (identified by a fragment ID) to a specific site (identified with a site ID).

⟨alter allocation⟩ ::= ALTER ALLOCATION FRAGMENT ⟨frag id⟩ IN ⟨kg name⟩ ⟨site id⟩
⟨site id⟩ ::= TO SITE ⟨unsigned integer⟩

In our example in the *yago KG*, we change the allocation of fragment 2 to the site identified with the ID 5.

```
ALTER ALLOCATION FRAGMENT 2 IN yago TO SITE 5
```

4.6.8 DISPATCH statement

This statement calls the dispatcher component of RDFPartSuite. It sends the fragments of a *KG* to the sites according to the allocation schema. It can be called only after the allocation schema has been created using the ALLOCATE statement. The data are sent to the sites indicated in the configuration file of the framework.

⟨dispatch allocation⟩ ::= DISPATCH KG ⟨kg name⟩

In our example, the *KG* is dispatched to the sites of the system as follows:

```
DISPATCH KG yago
```

4.6.9 Integration of the language to other systems

In this section we show that the proposed language is able to express the partitioning strategies used by most of distributed RDF systems. For example, HadoopRDF[DWNY12] applying a hashing function on the subject is mimicked creating forward star graphs fragments that are allocated using a round-robin. If there are 15 sites in the system, the partitioning is declared as:

```
FRAGMENT KG yago FORWARDLY
ALLOCATE yago BY ROUND-ROBIN SITES 15
```

If the system uses vertical partitions (e.g., SW-Store [AMMH09]), backward graph fragments are then declared. The declaration of fragments with maximum partition size of 1000 is for example:

```
FRAGMENT KG yago BACKWARDLY BY SUPERSET MAX 1000
ALLOCATE yago BY ROUND-ROBIN IN SITES 5
```

The gStoreD [PZÖ+16] system creates the forward entities and leaves to the user the freedom to choose the allocation strategy. Supposing that she/he chose a graph partitioning heuristic, the declaration is:

```
FRAGMENT KG yago FORWARDLY
ALLOCATE yago BY METIS SITES 15
```

The system RDF_QDAG[KMG+20] uses both, forward and backward segments, and allocate them with a graph partitioning heuristic declared as:

```
FRAGMENT KG yago FORWARDLY
FRAGMENT KG yago BACKWARDLY
ALLOCATE yago BY METIS SITES 15
```

4.7 RDF partitioning advisor

In this section we present a partitioning wizard that integrates in different modules each of the fragmentation and allocation approaches that have been proposed so far. Our goal is to provide user-friendly tools for experimented and non-expert users. An expert user (manager) can choose to integrate the interfaces of *RDFPartSuite* directly to her/his code. The user can also choose to manually partition the data using the declarative partitioning language that we proposed in the previous section. Our partitioning advisor is designed to assist non-expert designers or administrators of centralized or distributed triple stores. It produces a partitioning schema for a KG based on the input user's requirements. It provides a degree of *comfort* similar to the one offered by the partitioning advisors in the relational model (Sect. 1.4.3.1).

4.7.1 Main functionalities

RDFPartSuite's advisor functionalities are summarized as follows:

- *Collect manager's requirements:* the advisor provides means to managers of centralized and distributed triple stores to record the creator's requirements (functional and non-functional), constraints, and KGs to be considered in the fragmentation, allocation and loading processes.

- *Smart KG Fragmentation*: the advisor proposes a fragmentation schema for the input KG considering the manager's explicit requirements and using default parameters (e.g., similarity threshold).

- *Smart KG Allocation*: an allocation schema is proposed by the advisor. This schema considers the manager's requirements explicited in the first requirement.

- *Tune fragmentation and allocation*: the advisor leaves the final decision of how to fragment and how to allocate to the managers. The manager is able to choose different fragmentation and allocation strategies from the ones proposed automatically by the advisor. Additionally, it lets the manager change the values of the input parameters and let them sub partition manually large graph fragments.

- *Load the data to a triple store*: this functionality concerns the data transfer and loading to the target triple store. At the moment we support RDF_QDAG and gStoreD.

As summary of the functionalities of the system is shown in Figure 4.14. Basically, the manager seeks to fragment a given raw RDF dataset in graph fragments. The tool adjusts the partitioning strategy according to a set of configuration parameters (e.g., system catalog, constraints). It allows exploring the set of fragments and their distribution and if necessary, adjusting the fragmentation strategy or re-partitioning some fragments. Finally, the set of graph fragments are distributed/loaded to the sites of the system.

The following section describes the architecture of this system that we name *RDFPartSuite GUI*.

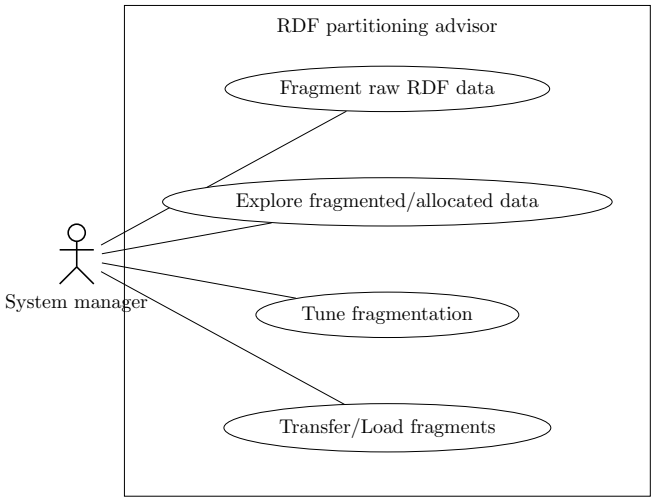


Figure 4.14: Partitioning advisor use case

4.7.2 System architecture

In this section we describe the architecture of *RDFPartSuite* advisor illustrated in Figure 4.15. It is composed of five modules: features extractor, fragmenter, allocator, dispatcher and the Graphic User Interface (GUI). The framework provides a Web User Interface to manage the partitioning process. This interface is similar to the ones available to administrate the current big data frameworks. The back-end layer was implemented using the Java programming language chosen for its robustness, portability and available libraries. The communication between the backend and the graphic interface is performed using a REST API. The backend is built following the MicroProfile specification using JAX-RS and CDI components. The client layer (i.e. front-end) is implemented with Vue.js and Bootstrap. Below, we detail each of the *RDFPartSuite* components.

4.7.2.1 Features extractor

This component is in charge of processing the information collected from the GUI. It gets for instance the configuration parameters for the fragmentation algorithms. For example, the similarity threshold, the type of similarity scores and the allocation methods. It also processes information related to the system's hardware (e.g., available space in hard disk, main memory available per site).

4.7.2.2 Fragmenter

This component is in charge of creating the forward and backward graph fragments. It gathers the data following Algorithms 1 and 4 respectively. By default it creates groups of triples using the structural similarity measure on the characteristic sets. The similarity threshold is initially fixed to 1, but it can be tuned in the next component. This component also calculates a set of statistics characterizing the fragments. It counts the number of triples per fragment and calculates the distribution of the data on the fragments. It also detects which fragments should be re-partitioned because their size exceeds the maximium available space in a single site. It communicates with both, the fragmenter and dispatcher components, since it collects information to create the plots guiding the user to choose the optimal fragmentation and allocation strategies. For the distributor component, it calculates the precision of the distribution calculating the relative number of cuts in the sites.

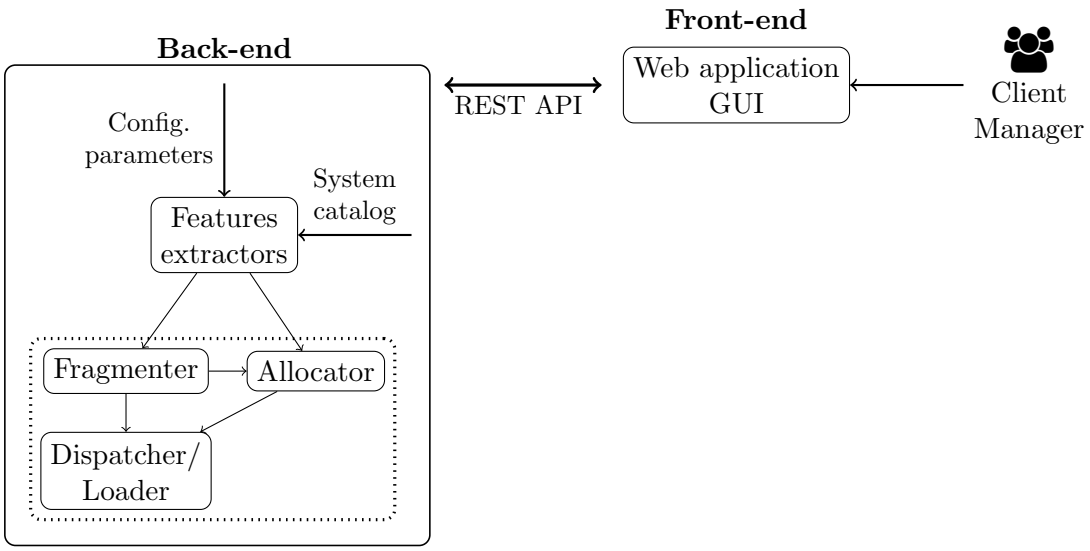


Figure 4.15: Partitioning advisor architecture

4.7.2.3 Allocator

This component calculates firstly the distribution of the final graph fragments (the ones created after re-partitioning) to the sites of the system. It lets the user choose among the graph partitioning heuristic and the traditional hashing function. It sends the number of data exchanged between two partitions to the statistics generator component which calculates the allocation's precision. The output of this component is a file indicating the site to which each fragment should be assigned.

4.7.2.4 Dispatcher/Loader

This component is in charge of sending the fragments to the sites of the distributed system according to the allocation schema produced by the previous component. This component connects to the loading module of a centralized (RDF_QDAG) or distributed triple store (gStore) and load the data according to the fragmentation (in centralized) or allocation (in distributed) schema.

4.7.3 Use case

In this section we present the functionalities of *RDFPart* following the sequence of screens in the web graphic user interface. We show for each screen the expected inputs, options and outputs from the processing modules in the back-end. The welcome screen of our interface is shown in Figure 4.16. In this interface, the user selects the raw RDF file to be partitioned (in N-Triples format). Then, it selects the directory to store the temporary files generated during the fragmentation stage. By default, this directory is set the same directory as the raw RDF file. Next, it collects information about the infrastructure in which the data will be partitioned. It gets the number of sites (workers and master), and their available space in hard disk and main memory. If the site's specifications are different in the master site (which sometimes have more available main memory and disk), it records this information in a different form. After this data is set, the *Start Partitioning* action button activates and the system performs an initial scan on the data to encode it and detect the distinct predicates. Finally, it allows to select the domain and range of each predicate, which by default are set to strings. After this stage, the fragmentation strategy using Algorithms 1 and 4 begins.

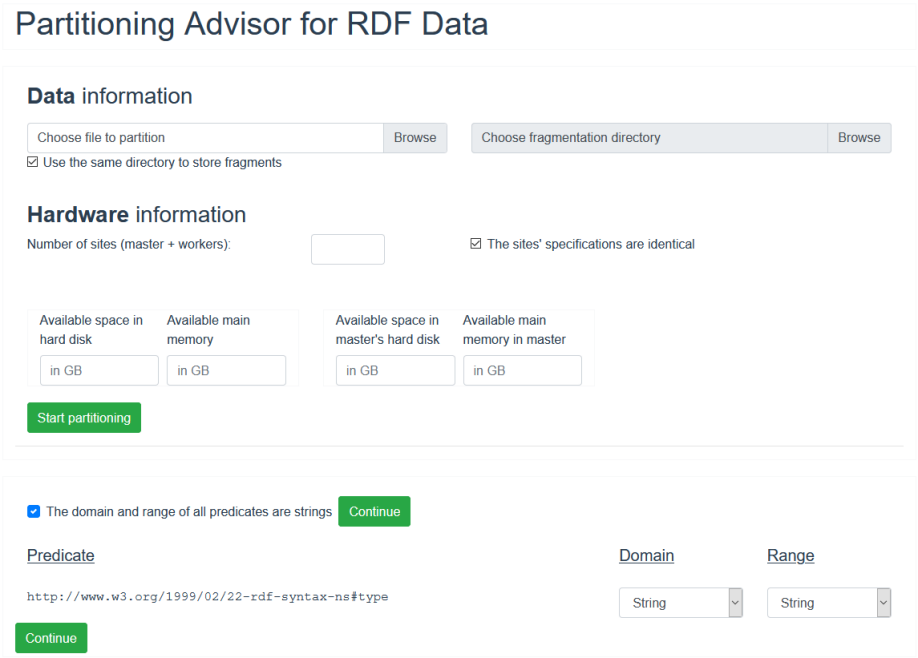
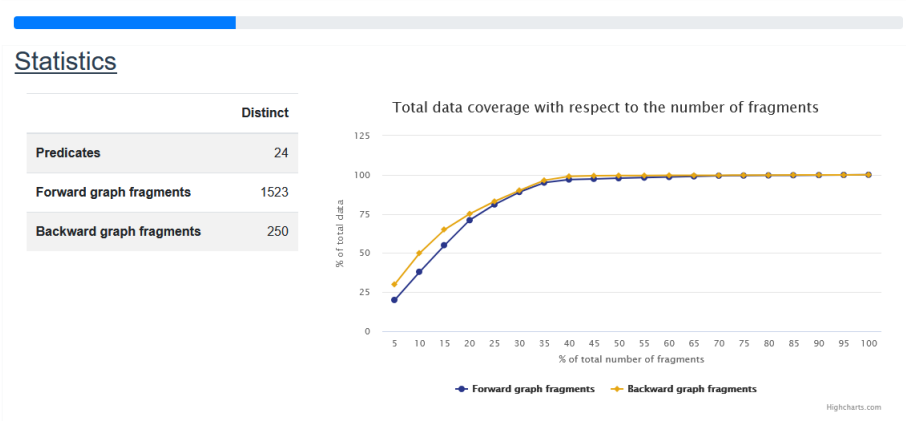


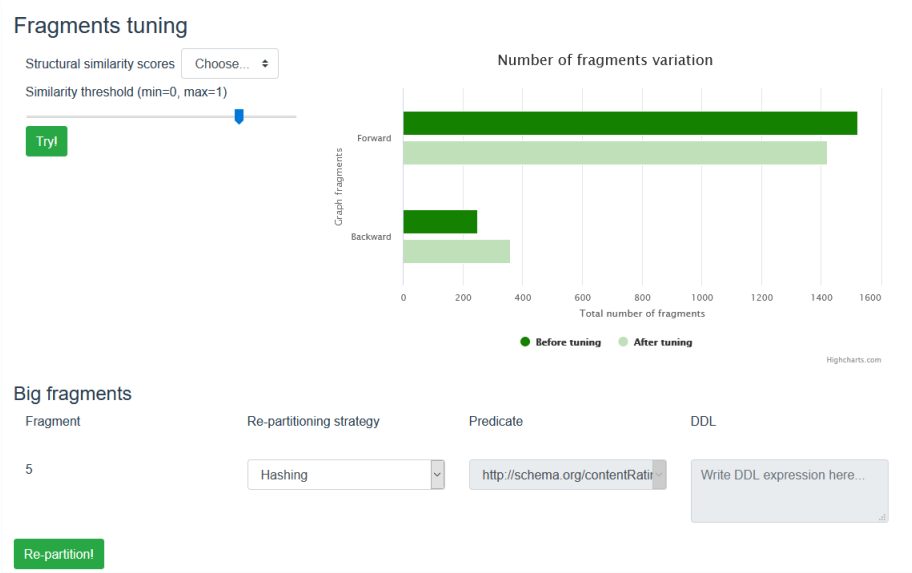
Figure 4.16: *RDFPart* advisor's welcome screen

The fragmentation progress can be tracked using the progress bar of the second screen shown in Figure 4.17a. Once the fragmentation process finishes, the forward and backward graph fragments are persisted to a temporary directory and a set of statistics are displayed. These statistics include the number of distinct predicates, forward and backward graph fragments. By default the fragmentation is performed using the structural similarity measure (which is by far the simplest). We displayed as well the distribution of the total data in the forward and backward fragments in a line chart. This graph shows from the total number of fragments (normalized to a percentage), what is the percentage of the total data stored on them. This visualization allows to detect if most of the data are condensed into a few fragments and should therefore be re-distributed. The re-fragmentation module is illustrated in Figure 4.17b. This screen allows to change the default structural similarity score and to select a distinct similarity threshold. Since the interface was developed with VueJs, the graph of Figure 4.17a would change when the similarity score is changed. A comparison of the total number of fragments of the default fragmentation and the tested one is shown in an horizontal bar chart in Figure 4.17b. Finally, if the set of fragments whose size is greater than the available space in a single site are shown in at the bottom of the screen in the section *Big fragments*. This menu allows to choose the re-partitioning strategy which by default is set to a hashing function as described in Section 3.7. However, the data from a fragment can be re-partitioned using a single predicate and the definition language of the previous section. To save the changes the user clicks on the *Re-partition!* button.

The final screen concerns the allocation of the forward and backward fragments to the sites of the distributed system. This screen is shown in Figure 4.18. When the workload is available, it selects the directory storing the SPARQL query files. The specification asks a single file per query. After clicking the *Test allocation* button it compares using a customized precision function the distribution of the case in which the fragments are distributed using a hashing function on the predicates, and the graph partitioning heuristic defined in Sect. 3.5. The final allocation method is selected from the right drop-down list and it generates in the temporary directory a file with the graph fragment identifier and its location id.



(a) *RDFPart* initial statistics



(b) *RDFPart* re-fragmentation module

Figure 4.17: *RDFPart* statistics and re-fragmentation modules

Figure 4.18: *RDFPart* allocation interface

4.8 Conclusion

In this chapter we presented our framework in action. We used the fragmentation and allocation modules to load the data into a centralized and distributed triple store. We performed a series of tests on each of the tested systems to show the strengths, features and limits of our framework. We started giving an overview of the centralized RDF_QDAG triple store. This system stores the data as forward and backward graph fragments. We integrated our findings to the loading module of this system. We compared the loading costs of RDF_QDAG with other representative systems of the state of the art in terms of time and scalability. We showed that the fragmentation process with our entities ($\mathcal{G}f$) is feasible, even in architectures with limited resources. Even though the loading times were high compared to other lighter fragmentation techniques (e.g., triple table), we saw the advantages at query runtime of this extra cost. Then, we performed a complete evaluation of the fragmentation strategies. We started showing that the number of fragments is reasonable with respect to the datasets' sizes. Next, we use our framework to fragment the data in forward and backward graph fragments in a graph-based distributed triple store (gStoreD). This system was used to compare Forward and Backward fragmentation strategies. Next, we showed that combining both types of fragments in a single system contributes not only for scalability but to enhance the query performance. We performed an extensive study on the communication costs comparing our partitioning strategy with the most relevant state-of-the-art techniques. We have shown that our strategies give a promising compromise between the allocation cost, measured in terms of loading time, distribution's quality, and query performance. Finally, we introduce a user friendly extensions to our framework. These extensions are a declarative partitioning language built on top of the logical structures of the previous chapter and a partitioning advisor.

Conclusions and Perspectives

Conclusion

Currently, our world is filled with data. A large part of this data is freely available on the Web. The W3C devoted many efforts to the development of standards that facilitate the exploitation, exchange, and *reuse* of this data. Among these standards, RDF excels for its flexibility, simplicity, and the expressiveness of its query language (SPARQL). RDF statements are triples logically represented in a graph. These graphs interconnect data coming from multiple datasets constituting a Knowledge Graph (KG). The complexity of this later concerns both its ecosystem that involves several actors (e.g., creators, consumers, managers) and infrastructures, and its size. To satisfy the requirements of these actors, each KG has to be efficiently stored and queried to facilitate the different processes of its exploitation. As quoted by Clifford Stoll, "Data is not information, information is not knowledge, knowledge is not understanding, understanding is not wisdom", reaching wisdom passes through the availability of both data and knowledge.

The storage infrastructure, and specifically the Triple Stores are a cornerstone to satisfy these needs. This pushed researchers from diverse communities (e.g., Semantic Web, Core Databases, Information Systems Infrastructures) to propose a large collection of triple stores. Some are built on top of existent solutions (like RDBMS), others are built from scratch using customized storage implementations (e.g., clustered indexes, adjacency lists). They all have in common the use of physical optimization strategies that are *specific* to each system. These strategies usually *ignore* the underlying graph-based *logical schema* of KGs. This is far from the optimizations based on logical structures broadly studied in relational databases. Within these strategies we find *data partitioning* in the first line.

Data partitioning grants managers means to enhance the performance, manageability, and availability of centralized triple stores by splitting the data in parts (fragments). Still, data partitioning in centralized triple stores has not had its rightful place in the design of these systems. To scale to large KGs, many parallel and distributed triple stores came to light. In these systems, data partitioning becomes a mandatory stride. Many systems offer managers the possibility to control the partitioning of their data (opaque systems) while others delegate this task to a host like a cloud provider (transparent systems). For the former systems, there are several partitioning modes (hashing, graph partitioning, semantic hashing, etc.). They are applied at the triple-level distributing the data at its finer granularity. Recently, there has been a research boom to study the performance impact of these data placement strategies in triple stores [JSL20, ANS18].

The need to address the problem of RDF data partitioning is evident and crucial for performance and manageability issues. Our vision encourages the reuse and the reproduction of strong points of data partitioning applied in different database generations (relational, object-oriented, XML, data warehouses, etc.) that preceded RDF databases. To do so, it is necessary to make the partitioning environment *explicit*, identifying its strengths, evolution, and how their main

elements can be reused in the context of *KG*s. We surveyed the data partitioning problem to establish the bases that will guide the development of our partitioning framework. It is important to notice that reproducibility is the hallmark of scientific research. A recent initiative from Information Systems Journal (Elsevier) identified that much of the research published in computer science journals and presented at conferences cannot be readily and fully reproduced. This motivates the Editors of this journal to establishing a new article type: the Invited Reproducibility Paper[14]. This reproducibility concerns only experiments. At the same time, several nice conceptual ideas and visions already published and implemented in systems/products in the last decades are not well reproduced in current trends in the database world. The wealth of data partitioning findings in the traditional databases in terms of availability of large body of partitioning techniques, algorithms, data structures, optimizations, deployment (in commercial and open-source DBMSs), advisors, connection with other optimization techniques (such as indexes, materialized views, etc.), fragment manipulation languages, adaptation for different database generations, and the large number of Ph.D. theses (a quick search using "Fragmentation Bases de Données Informatique" on `theses.fr` which is a Website that lists all Ph.D. thesis defended in France since 1985, we find around 875 theses). By this thesis, we attempt to promote the vision of the reproduction of findings of data partitioning in the context of triple stores.

This vision is embedded in our RDF data partitioning framework for the contexts of centralized, distributed, and parallel triple stores. The framework follows the paradigm of being *outside* the DBMS so it can be *reused*, improved, and extended as we will discuss later. Our framework works using a logical representation of the fragments. We define a set of logical entities that allow partitioning *KG*s independently on the storage representation of each system. Finally, we show how this framework was introduced into centralized and distributed triple stores. We conducted several experiments that showed some scenarios that demonstrate when these partition strategies are most efficient. Our contributions are summarized as follows.

- **Data partitioning survey**

We provide a complete survey of the partitioning problem in relational databases. We have cleared the foundation of the problem and its evolution over all the database generations. We characterized the problem thanks to a star-schema like the representation of data partitioning defining 10 main dimensions comprising: (1) Partitioning type, (2) Algorithm, (3) Main objective, (4) Adaptability, (5) Mechanism, (6) Cost model, (7) System element, (8) Platform, (9) Constraints and (10) Data model. These dimensions make the elements of the partitioning environment explicit and they are used to classifying the surveyed works. Secondly, since we are dealing with Knowledge *Graphs* we cannot disregard how graphs have been treated in the literature. We surveyed their logical representation, storage, and querying strategies of this data model. Then, we focus on RDF systems giving some background concepts and an overview of the current storage and partitioning strategies applied by triple stores. Thanks to both surveys, we found that the study of data partitioning in triple stores has not followed the same philosophy as in relational databases. The partitioning environment in RDF datasets is still un-explicit. Several methods exist to partition an RDF dataset, but they are hardly reproducible and extensible to other systems. This is due to the dependency of these methods on the physical storage structure of triples on a given triple store.

- **Foundations to RDF Logical Partitioning**

Drawing on the lessons learned from the partitioning environment in relational databases, our first concern was the definition of logical partitioning structures to partition a *KG*. These structures should (i) offer more expressiveness than single triples, (ii) represent the overall structure of the data and, (iii) maintain their graph connectivity. In this thesis, we formally defined these

[14] `https://www.elsevier.com/connect/new-article-type-verifies-experimental-reproducibility`

structures. We distinguish two structures: *forward graph fragments* $\overrightarrow{\mathcal{G}f}$ and *backward graph fragments* $\overleftarrow{\mathcal{G}f}$. They harmonize with the notions of horizontal and vertical partitions in the relational model, respectively. However, they have been defined on graph abstractions and not on a table. We proved that the partitioning process of a KG in graph fragments enforces the correctness rules of fragmentation [ÖV11]. Also, graph fragments are independent of the storage structure of a triple store (table- or graph-driven). Besides, since they are built from the original KG and not from the storage representation of the triples, they manage to keep the inherent graph structure of a KG.

- **RDFPartSuite Framework**

We define a common partitioning framework for centralized and parallel triple stores. Our framework follows the paradigm of outside the DBMS (outside the Triple Store in our case). This enables users to *reuse* and extend this framework independently from a given triple store. The framework is composed of three main modules: *fragmentor*, *allocator* and *dispatcher*. The first module partitions a given dataset in fragments according to their logical representation in forward or backward graph fragments. This logical representation is sent to the target centralized triple store (if the systems are centralized) or to the allocator module (in distributed systems). The allocator assigns the fragments to the different sites of the system according to data-driven algorithms. This module outputs an allocation schema that is sent to the dispatcher in charge of distributing the triples according to this schema. Finally, our framework offers managers support tools. These tools consider the manager's expertise level to design a partitioning scheme. For beginners, it offers a partitioning advisor that automatically chooses the partitioning scheme. If the manager is an expert, a declarative partitioning language is given to partition a KG manually.

- **RDFPartSuite in Action**

Our framework was first incorporated to the loading module of a centralized triple store. The chosen system was the centralized triple store RDF_QDAG [KMG+20], currently developed in our LIAS Laboratory. We showed the scalability of the fragmentor module and compared the performance of non-partitioned centralized stores with respect to this system leveraging data partitioning. Next, we used our framework to load the data to a distributed triple store (gStoreD[PZÖ+16]). We compared the query performance for configurations in which the data are loaded as forward and as backward graph fragments in this system. Our experimental study revealed that the use of these structures in triple stores not only grants designers more freedom, but also ensures the scalability and good performance for certain queries.

Final Takeaways

In this section, we share our own experiences that led to the establishment of our vision that was materialized in our partitioning framework. Our aim is to give future researchers an overview of the process followed in the development and implementation of our framework so they can reuse/improve their future works with our feedback. The need to create a partitioning framework arose after exploring the papers and source codes of several triple stores. Our aim was initially to reproduce and learn how the systems that had been tested in several surveys (e.g., [AHKK17, KM15]) works. Many of these systems had not been maintained for a long time and they have been barely used in real implementations. Testing every single system was impossible since the number of existing triple stores is huge. This because usually improving existent triple store systems can be more expensive than starting everything from scratch. At this point, we questioned why this has been the case for triple stores and not for RDBMS? Certainly, there are many RDBMSs on the market now, however studying their optimization strategies remains

constant to the relational model. We decided to reproduce a similar methodology for the design of triple stores. We focused on data partitioning because recently this optimization strategy has gained momentum in centralized and especially in parallel and distributed triple stores. Our goal was then, to overview in detail the main components of data partitioning to reuse them in the context of triple stores. We seek to imitate the methodology used to partition RDF data and not exactly the implementations used by RDBMS. Our idea is to take advantage of so many lessons learned over the years for the creation of partitions in the relational model and incorporate them into the creation of our framework. Our framework can be improved and extended. The main perspectives of our work are detailed in the following section.

Perspectives

Extensions to our framework

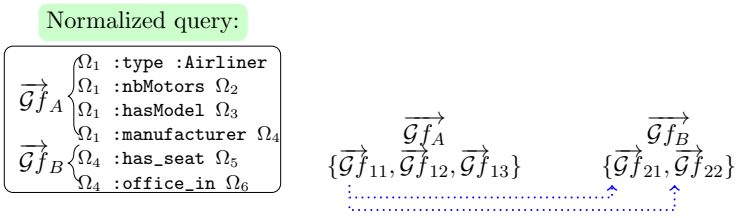
Calibrator component

The calibrator component would be in charge of monitoring the different schemes generated by our framework using data-driven approaches. This calibration is performed by the exploitation of workloads running on the target store. This allows us to have a mixed approach combing the connectivity of the data and workloads while designing our stores. We use the workload to *boost* the quality of the initial allocation. This task could be divided in two sub-modules: workload handler and smart re-allocator. In the following, we give a brief overview of how both modules could work.

Workload handler : This sub-component would be in charge of measuring the closeness between the graph fragments based on the information available on the workload. We could adapt some of the workload-aware partitioning algorithms of the state of the art. Let us for example describe how we could use the techniques of [HS13] and [GHS14] to extract the most relevant information from a representative workload. This is illustrated in Figure 4.19 in which a query (Fig. 4.19a) is normalized using the techniques of [HS13]. To measure the connectivity between graph fragments in the workload we can detect a set of *candidate* graph fragments for each part of the query. In the example shown in Fig. 4.19b we can calculate the interactions between the candidate fragments. This information is transmitted to the following module, that adjusts the cost model used by the allocator component.

```
SELECT ?a ?m ?z WHERE {
?a :type :Airliner .
?a :nbMotors "4".
?a :hasModel ?m  .
?a :manufacturer ?z .
?z :has_seat :France .
?z :office_in ?t .
FILTER regex(?t, "Toulouse") }
```

(a) Original query



(b) Query normalization

Figure 4.19: Normalization process

Smart re-partitioner this component considers the former data-driven partitioning proposed by the allocator and the insights from the workload handler to determine if a fragment should be transferred or replicated to a given site. To find a solution to the fragment re-allocation problem, one of the following actions must be performed on each fragment (V_i):

- *Replicate:* the fragment V_i is replicated to as many sites in S as long as the replication, space availability and imbalance constraints are met.

- *Move:* this action moves the fragment V_i from its former site S_j to another site in the network.
- *Keep:* this action maintains the fragment in its original location S_j.

The interaction between both sub-components is illustrated in Figure 4.20.

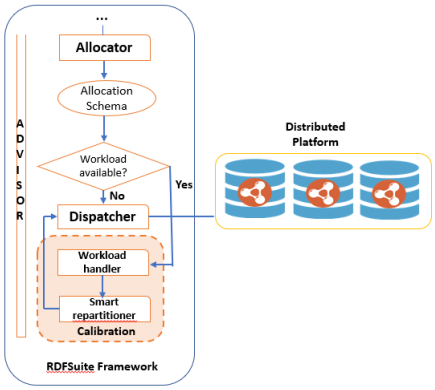


Figure 4.20: Calibration component extension

Other allocation rules

Currently, the allocator component of our framework implements data-driven algorithms to place graph fragments to the sites of a distributed system. This strategy is based on the inherent connectivity of the triples. However, we can imagine other data-driven rules to extend our framework. For example, for datasets containing geospatial information. Triple stores have been recently extended for the efficient management of spatial data. Thus, one may want to place graph fragments together according to geospatial variables. Another data-driven strategy to be considered is that useful for triple stores that support reasoning. This allocation strategy should consider the semantical closeness of their fragments with respect to a given set of ontologies. This will not only leave connected triples on the graph in the same site, but also triples that are semantically close but not directly with an arc.

Consideration of other optimization techniques

For the moment, our framework is used to partition KGs in triple stores. We are aware that data partitioning is just one of the pool of optimizations strategies to consider while designing efficient storage systems. Considering the impact of data partitioning and its interaction with indexes, replicas, materialized views, etc. is the next step. Following what happened in RDBMS, partitioning advisors proposed first a partitioning scheme that is tuned by subsequent optimization structures (e.g., indexes) [NB11]. Our framework can be used as a baseline for other components in charge of proposing indexes or replications of some fragments.

Implementation to transparent triple stores

We have introduced the notion of data transparency associated with data partitioning. In transparent distributed triple stores, the partitioning task is delegated to the host where the system is deployed. What if the host offered clients more visibility and control to partition their own data? Incorporating our partitioning framework into existent distributed frameworks should be considered. This need is evidenced, for example, by the overheads suffered by many Hadoop-based systems due to the high communications costs of MapReduce joins [AHKK17].

Dynamic optimization strategies

Today there are many very efficient main-memory distributed triple stores. In these stores, fast dynamic optimizations to the data distribution can be considered. These optimizations may consider query streams to decide on re-partitioning or replication strategies. A component can be added to our framework to deal specifically with this online reorganization.

Reproducibility of our framework to other types of data

In this thesis we present a vision that includes the design process of triple stores. Our vision can be reused/adapted to the contexts of other existent data types that exist today (e.g., property graphs) or data models that are yet to come.

A repository for data partitioning findings

In presenting our thesis, we did our best to promote the idea of reproductivity of the strong points of data partitioning in the traditional databases (deployed in centralized and parallel infrastructures) in RDF stores. If we deepen our analysis, the development of a repository (a knowledge graph-like) containing all findings of data partitioning in both traditional and RDF databases will be a great asset for students, academia, industry, developers of Open Sources systems, just to name a few. These findings have to be associated with their scientific papers and/or the systems implementing them. Such a repository will allow any user to navigate through different dimensions of this repository to find their needs. This repository can be complementary to a traditional survey.

Résumé

Introduction

Contexte

Nous vivons dans un monde hyper-connecté dans lequel de grandes quantités de données et de connaissances sont émises par plusieurs fournisseurs tels que les réseaux sociaux, les appareils mobiles, le commerce électronique, des capteurs, l'Internet des objets et bien d'autres. Une grande partie de ces données est produite et disponible sur Internet. À ce jour, le Web contient plus de 1,8 milliard de sites[15] et sa taille est en constante expansion. Les données issues du Web se caractérisent, entre autres, par leur *volume*, *variété*, *véracité* et *vélocité*. Les informations et les connaissances associées à ces données, si elles sont bien préparées, peuvent être précieuses pour les consommateurs (entreprises, gouvernements, utilisateurs privés, etc.) qui peuvent obtenir des *connaissances* pertinentes et éventuellement de la *valeur ajouté*.

Cependant, la disponibilité des données issues du Web n'implique pas leur exploitation directe. Les ordinateurs doivent manipuler et interpréter les informations publiées sur le Web. Cette idée a donné lieu au Web sémantique. Les données issues du Web doivent être représentées et interrogeables de manière intuitive et claire. Le World Wide Web Consortium (W3C) a mené de nombreux efforts pour spécifier, développer et déployer des directives, des modèles de données et des langages. Parmi ces modèles, le Resource Description Framework (RDF) [RC14] a été conçu pour exprimer des informations sur les ressources du Web. Ce modèle est la pierre angulaire des autres modèles et langages du W3C.

Le triplet est la *plus petite unité de données* dans RDF. Un triplet modélise une seule déclaration sur les ressources avec la structure suivante $\langle sujet, predicat, objet \rangle$. Le triplet indique qu'une relation identifiée par le prédicat (également connu sous le nom de propriété) est maintenu entre le sujet et l'objet représentant des ressources Web (entités, documents, concepts, etc.).Cette caractéristique d'interconnexion donne à RDF la possibilité de lier des triplets de différents ensembles de données via leur IRIs (International Resource Identifiers). Le résultat de la fusion des triplets constitue une *Graphe de Connaissances* (KG). Pour interroger les ensembles de données RDF et les KGs, le W3C a défini SPARQL [SH13a] comme langage de requête standard. SPARQL permet d'exprimer des requêtes sur divers ensembles de données.

La popularité de RDF est due à sa *flexibilité*, *simplicité* et *disponibilité* d'un langage de requête. Aujourd'hui, les graphes de connaissances (KGs) sont devenus des moyens populaires pour les chercheurs académiques et industrielles pour *capturer*, *représenter* et *exploiter* des connaissances structurées. Cette variété et la richesse de KGs a motivé des chercheurs d'autres domaines à les intégrer dans les différentes phases de leur cycle de vie.

[15] `https://www.internetlivestats.com/`

La course aux Triple Stores efficaces

Avec l'essor des graphes de connaissances, deux problèmes se posent: *(i)* stockage scalable et *(ii)* accès efficace aux KGs. Les systèmes en charge du stockage et de l'accès aux triplets sont appelés Triple Stores. Ces systèmes peuvent être classés en deux groupes principaux: *non-native* et *native*.

Non-native Ces systèmes choisissent des solutions de stockage existantes pour construire leurs triple stores. Les systèmes de gestion de bases de données relationnelles (SGBDR) ont été largement utilisés pour concevoir de tels systèmes. Les premiers systèmes stockent les triplets dans une seule table de trois colonnes (sujet S, prédicat P et objet O). Pour réduire le nombre d'auto-jointures induites par cette approche, d'autres systèmes utilisent une table plate appelée table de propriétés. Cette table stocke chaque propriété dans des colonnes différentes. Cette stratégie de stockage génère le problème des valeurs nulles et des contraintes pour représenter des propriétés à valeurs multiples. Pour surmonter les inconvénients ci-dessus, certains systèmes partitionnent verticalement la table de propriétés en L fragments (où L représente le nombre de prédicats). Chaque fragment est associé à un prédicat et composé de deux colonnes (sujet, objet).

Native Ces systèmes ont été spécialement conçus pour traiter les données RDF. Deux stratégies de stockage principales sont utilisées dans ces systèmes: les stratégies basées sur une structure d'arbre et les listes d'adjacence. Le système RDF-3X [NW08] est le système le plus représentatif dans la première catégorie. Il stocke les triplets dans des arbres B+Tree. Pour améliorer les performances, il stocke toutes les permutations des colonnes (par exemple, SPO, OPS, PSO). Les systèmes de la deuxième catégorie utilisent des implémentations physiques de graphes tels que des listes d'adjacence. Les optimisations de ces systèmes dépendent fortement des structures de mise en œuvre du graphe.

Triple Stores distribués et parallèles Pour assurer la scalabilité, il existe une grande nombre de triple stores distribués et paralleles. Le cycle du développement d'un tel système peut facilement s'inspirer de celui utilisé pour la conception des bases de données traditionnelles. Le partitionnement des données est une condition préalable pour assurer le déploiement, l'efficacité, la scalabilité et la tolérance aux pannes de ces systèmes [ÖV11]. La sensibilité du partitionnement des données RDF et son impact sur les performances dans les triple stores ont été récemment discutés dans un article présenté lors d'un workshop de SIGMOD 2020 [JSL20]. Nous classons ces systèmes en trois groupes: homogènes, hétérogènes et basés sur le cloud. Les systèmes homogènes et hétérogènes sont généralement implémentés dans des architectures master/slave. Les premiers clonent un triple store centralisé sur les nœuds d'un système distribué tandis que le second utilise des triple stores différents en chaque nœud. Dans les systèmes basés sur le cloud, le stockage et l'accès aux données sont délégués à l'hôte. Plusieurs modes de partitionnement pour les données RDF ont également été implémentés (par exemple, *Hashing*, *Partitionnement de graphes* et *Hashing sémantique*.

De la présentation ci-dessus, y compris les triple stores centralisés, distribués et parallèles, trois leçons principales sont tirées:

1. Une forte demande des gestionnaires et des consommateurs pour développer des triple stores en exploitant l'opportunité technologique en termes de logiciels et de matériels.

2. Toute initiative visant à utiliser le partitionnement des données dans des triple stores devrait bénéficier de la grande expérience dans les bases des données traditionnels. Cette expérience doit être capitalisée pour augmenter la *réutilisation* et la *reproductibilité* des résultats du partitionnement des données dans les données RDF.

3. La nécessité de développer un cadre de partitionnement de données RDF pour les gestionnaires désireux de concevoir des triple stores centralisés et parallèles.

Notre vision, objectifs et contributions

Dans cette thèse, nous visons d'abord à expliciter le problème du partitionnement pour les bases de données traditionnelles. Nous privilégions l'utilisation du partitionnement de données dans la conception de triple stores en réutilisant ses points forts. Notre vision est de construire un cadre de partitionnement basé sur l'analyse de forces et de contraintes des deux univers (bases de données traditionnels et KG). Pour cela une étude approfondie du partitionnement des données proposé dans le contexte des bases de données traditionnelles centralisées/parallèles/distribuées est nécessaire.

Les objectifs que nous avons fixés dans notre thèse sont:

(i) La définition d'un cadre commun pour les triple stores centralisés et parallèles avec des composants complets mettant en œuvre notre vision.

(ii) La proposition d'algorithmes efficaces de partitionnement pilotés par les données définis sur des structures logiques définis préalablement.

(iii) Montrer la faisabilité du framework en l'instanciant dans un triple store centralisé et distribué.

(iv) Mise à disposition d'outils pour les concepteurs, y compris un langage de manipulation de fragments et un assistant automatique de partitionnement, basés sur des fragments logiques.

Les principales contributions de notre travail sont:

- Avec la motivation d'augmenter la reproduction et la réutilisation des résultats importants du partitionnement des données dans les bases de données centralisées et parallèles traditionnelles, nous proposons une étude complète de ce problème couvrant les générations importantes du monde des bases de données.
- La définition d'un Framework, appelé *RDFPartSuite*. Il soutient notre vision dans la conception de triple stores centralisés et parallèles. *RDFPartSuite* peut être personnalisé en fonction du type de plateforme et il suit le paradigme en dehors d'un SGBD [Ord13], dans lequel tous les efforts techniques sont effectués en externe. En fonction du type de plateforme deux scénarios sont possibles:
  1. Si la plateforme est centralisée, notre framework active le module *fragmenteur* en charge du partitionnement des triples en fonction de leur représentation logique.
  2. Lorsque la plate-forme est distribuée ou parallèle, le *fragmenteur* envoie le schéma de fragmentation obtenu au composant *allocateur*. Ce dernier attribue les différents fragments aux nœuds de la plate-forme cible. Une fois le schéma d'allocation obtenu, le composant *dispatcher* envoie la description détaillée de ce schéma à différents nœuds, puis charge les triplets en fonction de leurs définitions.
  3. Des outils d'assistance (langage de manipulation de fragments et assistant automatique de partitionnement) sont proposés pour aider les gestionnaires dans leurs tâches.
- La définition d'un ensemble d'entités logiques (appelées fragments de graphe) utilisées pour effectuer la fragmentation d'ensembles de données RDF au niveau logique dans des environnements centralisés et parallèles.
- Le déploiement de ce framework sur le module de chargement d'un système centralisé (RDF_QDAG [KMG+20]) et d'un triple store distribué (gStoreD [PZÖ+16]).

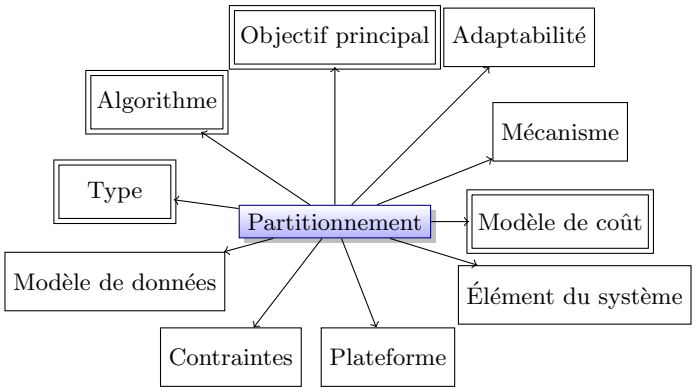


Figure 4.21: Schéma en étoile de l'environnement de partitionnement

Fondements du Partitionnement des Données

Dans les techniques d'optimisation les plus étudiées dans les bases de données, le partitionnement des données occupe une place très importante non seulement pour son efficacité à améliorer les performances de la base de données, mais aussi parce que dans les architectures parallèles et distribuées à grande échelle modernes, c'est un pas obligatoire. Les notions de partitionnement de données ont été introduites dans les années soixante-dix en [HS75] peu de temps après la définition des index de base de données.

Dans les bases de données relationnelles, le partitionnement consiste à diviser ce qui est logiquement une grande table en morceaux physiques plus petits. Comme pour les index, le problème du partitionnement des données a été couvert dans toutes les générations de bases de données: deductives [Spy87, NH94], orientée objet[BKS97] et entrepôts de données [SMR00]. Pourtant, contrairement à d'autres stratégies d'optimisation et avant l'introduction de techniques de partitionnement dynamique, des partitions sont créées lors de la déclaration des tables de base de données. Cette fonctionnalité distingue le partitionnement des données des autres techniques d'optimisation qui peuvent être créées à la demande pendant l'exécution de la base de données. Recréer des partitions après sa définition est coûteux et peu pratique.

Nous avons caractérisé le problème de partitionnement en fonction de son:

- **Maturité**: le problème existe presque depuis la définition des premiers systèmes de base de données et a été largement exploré,
- **Couverture**: il a été implémenté pour toutes les générations de bases de données,
- **Évolution:** il a été adapté aux différentes architectures, besoins et contraintes, et
- **Complexité:** trouver un schéma de partition optimal doit considérer plusieurs alternatives et parfois des fonctions objectives mutuellement exclusives.

Dimensions de partitionnement

Nous avons modélisé l'environnement de partitionnement à l'aide d'un schéma en étoile illustré dans la Figure 4.21. Les dimensions avec une double bordure dans cette Figure correspondent à celles définissant le problème de partitionnement de base. Nous organisons les approches de partitionnement dans ce chapitre en utilisant les dix dimensions détaillées ci-dessous qui sont utilisées comme critères de diversification. Ces schéma est utilisé pour organiser les travaux étudiés dans ce chapitre.

Type

Cette dimension décrit les façons alternatives de diviser une table en plus petites parties. Deux alternatives sont clairement visibles: la diviser *horizontalement* ou *verticalement*. Un troisième type de partitionnement nommé *hybride* est une combinaison des techniques de partitionnement précédentes. En bref, les entités logiques sont représentées sous forme de relations (tables) dans le modèle relationnel. Chaque relation regroupe des ensembles de tuples qui sont des instances d'une entité logique qui sont représentées par un ensemble d'attributs.

Le partitionnement horizontal a été exploré plus largement par les SGBD et il en existe de nombreuses versions. Tout d'abord, nous distinguons les partitions horizontales effectuées dans une seule table nommée *partitionnement horizontal simple*. Plusieurs modalités de partitionnement sont implémentes:

- Partitionnement par *plage*: une relation est divisée en fonction d'une plage de valeurs pour un ensemble donné d'attributs.
- Partitionnement de *liste*: cette stratégie divise une table selon une liste de valeurs discrètes pour la clé de partitionnement d'une colonne.
- Partitionnement *hash*: cette modalité décompose les données en appliquant une fonction de hachage aux attributs de partitionnement.

Un autre mode de partitionnement horizontal, appelé *composite*, combine les modes de partitionnement simple (par example, plage-hash, hash-liste). Enfin, le *partitionnement horizontal dérivée* permet de partitionner une table en exploitant une relation parent-enfant existante établie selon une clé étrangère entre deux relations [BBRW09].

Objectif principal

Le problème du partitionnement optimal des données sur un système de base de données centralisé, distribué ou parallèle pourrait à première vue sembler le même. Même si la sortie attendue est un ensemble de partitions horizontales ou verticales, l'objectif principal de créer chaque partition est très différent selon les exigences du système. Nous classons les objectifs de partitionnement comme suit: *(i) Temps de réponse*, *(ii) Concurrence*, *(iii) Maintenance* and *(iv) Combiné*.

Mécanisme

Cette dimension fait référence aux moyens utilisés par les techniques de partitionnement pour atteindre les objectifs mentionnés précédemment. Il est composé de deux alternatives: le clustering et le declustering.

- Le *clustering* fait référence au partitionnement dans des systèmes de base de données centralisés et distribués. Dans les premiers systèmes, le partitionnement des données cherche à minimiser les coûts de transfert entre la mémoire primaire et la mémoire secondaire en récupérant les enregistrements ou les attributs pertinents à la demande de l'utilisateur. Dans les systèmes de bases de données distribuées, l'objectif principal est d'éviter les communications réseau coûteuses en localisant les exécutions aux nœuds du système distribué où résident les données. Dans les deux cas, les données sont regroupées pour former des groupes de tuples ou d'attributs qui sont généralement récupérés ensemble.
- La stratégies de *declustering* est préférée par les systèmes de base de données parallèles où il n'est pas nécessaire de maximiser le traitement local à chaque nœud. Le partitionnement des données dans les systèmes parallèles se traduit par un compromis entre maximiser le temps de réponse pour chaque requête individuelle mais au prix d'une asymétrie d'exécution ou maximiser le parallélisme intra-requête.

Algorithme

Trouver le bon ensemble de partitions s'est avéré être un problème très complexe. Le problème de partitionnement horizontal s'est avéré être NP hard [SW85, SW83], de même que le problème de trouver des partitions verticales appliquant un graphe d'affinité. Le partitionnement horizontal et vertical ont été formalisés mathématiquement et posés comme des problèmes d'optimisation dont la solution est trouvée grâce à l'utilisation d'heuristiques. Nous classons les algorithmes en: *(i) Solutions exactes* et *(ii) Heuristiques.*

Modèle De Coût

Pour évaluer objectivement si une stratégie de partitionnement améliore les performances du système de base de données, un ensemble de métriques est établi pour définir les critères qui déterminent un système *performant.* Après tout, sans une métrique bien définie, il est impossible de décider quelle alternative de partitionnement est meilleure qu'une autre. De plus, les modèles de coûts sont essentiels pour prédire les coûts d'exécution réels d'une requête *a priori*, sans l'évaluer réellement. Les modèles de coûts offrent une vision simplifiée d'un système et sont liés aux objectifs décrits avant. Nous classons les modèles de coûts comme suit: *(i) Basé sur le temps de réponse*, *(ii) Basé sur le débit*, *(iii) Basé sur la maintenance* et *(iv) Coût global du système.*

Contraintes

Les algorithmes de partitionnement cherchent à optimiser une fonction objective par rapport à certaines variables en présence de contraintes. La liste suivante résume les contraintes les plus courantes: *(i) La taille maximale/minimale de la partition*, *(ii) Nombre de partitions*, *(iii) Redondance*, et *(iv) La charge de requêtes.*

Plateforme

Cette dimension considère les trois types d'architectures de base de données: *(i) Centralisée*, *(ii) Distribuée* et *(iii) Parallèle.*

Élément du Système

Dans cette dimension, nous considérons si l'élément matériel ciblé par l'approche de partitionnement. Parmi les composants, nous pouvons mentionner: mémoire primaire et secondaire, réseau local, processeur.

Adaptabilité

Cette dimension classe les stratégies de partitionnement en statiques et dynamiques. La différence dépend si le partitionnement décrit la procédure qui est effectuée lors de la déclaration de la base de données et qu'elle ne change jamais, ou s'il inclut décrit les stratégies effectuées après l'exécution de plusieurs charges de travail et que le système est capable d'adapter son partitionnement.

Modèle de données

Dans cette section, nous classons les stratégies de partitionnement selon le modèle de données utilisé pour représenter logiquement les données. Nous incluons dans cette catégorie le modèle relationnel et les modèles directement dérivés de celui-ci. Nous incluons alors: *(i) Modèle relationnel*, *(ii) Orientée objet*, *(iii) Déductive* and *(iv) Dimensionnelle.*

Partitionnement dans les plates-formes récentes à grande échelle

Au début des années 2000, la nécessité d'incorporer, de stocker et d'interroger de grandes quantités de données sur le Web, avec différentes sources, formats et à moindre coût, a contribué au développement de nouveaux types d'outils de gestion des données. Les systèmes basés sur le modèle relationnel, en particulier les bases de données parallèles, évoluent et peuvent gérer de vastes volumes de données, comme le montre [PPR+09]. Cependant, le modèle relationnel est trop rigide pour s'adapter à la variabilité des sources et des formats de données actuelles. De plus, étant donné que le schéma de données est rarement disponible au préalable, il est impossible d'effectuer l'étape de conception qui est essentielle lors de la construction d'une base de données relationnelle. Par conséquent, les systèmes appliquant une stratégie *schéma-après* ont été largement acceptés. Ces systèmes stockent initialement les données dans leur format brut et les analysent lors de l'exécution. Le pionnier présentant un nouvel outil de gestion de données évolutif était Google, introduisant le système de fichiers Google [GGL03] et le modèle de traitement MapReduce [DG04]. Leurs travaux ont servi de base à Hadoop, l'un des frameworks de données à grande échelle les plus populaires jusqu'à aujourd'hui. Dans cette section, nous commençons à donner un bref aperçu de Hadoop en mettant l'accent sur la stratégie de partitionnement appliquée par ce framework. Ensuite, nous expliquons et détaillons les systèmes les plus pertinents construits sur Hadoop et décrivons leurs stratégies de partitionnement.

Partitionnement dans l'écosystème Hadoop

Le stockage dans Hadoop est géré par le composant Hadoop Distributed File System (HDFS). Le HDFS est un système de stockage de données distribué basé sur une architecture maître-esclave. HDFS est conçu pour stocker des données structurées et non structurées de manière évolutive, hautement disponible et tolérante aux pannes. Les fichiers sont divisés en blocs d'une taille configurable (128 Mo par défaut) qui sont distribués et stockés entre les nœuds de données. Pour garantir la tolérance aux pannes et l'équilibrage de charge, les blocs de données du même fichier sont répliqués et stockés sur différents nœuds. Le nœud de nom (maître) gère le système de fichiers et régule l'accès des blocs de fichiers aux emplacements à l'aide des métadonnées du fichier.

Lorsqu'un fichier est importé dans HDFS, il est divisé séquentiellement en petits blocs d'une taille de stockage fixe. Chaque bloc est répliqué en un nombre fixe personnalisable de copies et une fonction de hachage est utilisée pour distribuer les données aux nœuds. Cette stratégie est assez simple et permet de distribuer les données contrôlant son équilibre. Cependant, il ne prend pas en compte le schéma des données et il peut produire des frais généraux de réseau à la récupération.

Conclusion

Avec le déluge de données et l'explosion des architectures de déploiement, le partitionnement des données est devenu la une technique que les acteurs de la base de données (concepteurs, administrateurs, architectes, analystes, étudiants, chercheurs) doivent connaître en détail. Le partitionnement des données a été étudié à partir de l'arrivée de bases de données évoluant avec ses générations. Dans ce chapitre, nous avons cherché à donner un aperçu historique du problème de partitionnement de données, de son historique, de son évolution, de ses contraintes et de son comportement après l'arrivée de chaque nouvelle architecture et plateforme de déploiement de données. Pour ce faire, nous avons proposé une base de partitionnement de données comprenant: des définitions et des types implicites et explicites, ainsi que des dimensions de partitionnement de données comprenant: (1) Type de partitionnement, (2) Algorithme, (3) Objectif principal, (4) Adaptabilité, (5) Mécanisme, (6) Modèle de coût, (7) Élément Système, (8) Plateforme, (9) Contraintes et (10) Modèle de données. Ces dimensions offrent une représentation graphique

(comme un schéma en étoile) du partitionnement des données. Pour chaque dimension, nous avons décrit ses caractéristiques et représenté ses éléments et ses étiquettes.

Nous nous sommes concentrés sur la présentation de l'état de l'art du partitionnement de données dans des bases de données relationnelles et d'autres modèles proches dans des architectures centralisées, distribuées et parallèles. Nous avons cité les travaux qui ont conduit à une définition explicite du problème du partitionnement et montré le développement progressif des principales formes de partitionnement de données. Par la suite, nous avons exposé les algorithmes, stratégies et heuristiques introduits pour partitionner la base de données selon un ensemble de dimensions précédemment définies. Nous offrons un examen complet sur le partitionnement des données et peut être utilisé comme un flambeau pour les lecteurs et les acteurs de travailler dans ce problème passionnant et durable. Enfin, nous montrons comment le problème a évolué dans les plates-formes cloud actuelles et comment il s'est adapté pour faire face aux données à grande échelle. Dans le chapitre suivant, nous montrons comment le problème a été traité spécifiquement pour le traitement de graphes.

Les Données Orientée Graphe : Représentation et Traitement

Dans le chapitre précédent, nous avons détaillé le problème de partitionnement dans les bases de données relationnelles. Nous avons montré son évolution et comment elle a été adaptée aux modèles de données orientés objet, déductifs et multidimensionnels. Nous avons expliqué le développement de plates-formes à grande échelle et comment elles se sont adaptées au besoin récent de gérer efficacement les grandes volumes de données. Cependant, les modèles de données discutés dérivent directement de la conception relationnelle si largement étudiée. Dans ce chapitre, nous détaillons un modèle de données qui permet de décrire des relations plus complexes. Plus précisément, des modèles orientés graphes représentant les données sous forme de graphe dont la manipulation est exprimée sous forme d'opérateurs de graphes. Ils ont pris de l'ampleur au cours des dernières années avec le développement du Web sémantique, des réseaux sociaux et avec l'augmentation de la capacité de capturer des données provenant de différentes disciplines telles que la Biologie et le Transport. Dans ces applications, l'interconnectivité des données est une caractéristique clé et elle est aussi importante que les données elles-mêmes. La modélisation des données sous forme de graphe permet une manière plus naturelle de gérer ces applications. Dans ce chapitre, nous commençons à donner plus d'informations sur le modèle de données de graphes en général. Ensuite, nous nous concentrons sur le modèle RDF introduit pour représenter les données dans le Web sémantique en donnant quelques définitions principales et en décrivant les approches de stockage et de traitement les plus pertinentes.

Modèles de bases de données orientées graphes

La recherche autour des modèles de base de données orientées graphes n'est pas nouvelle, en fait, elle a connu un pic à la fin des années 1980 début des années 1990, mais elle a été éclipsée par le développement de structures arborescentes comme la modélisation XML dans les sites Web de première génération. L'utilisation des graphes en informatique remonte au début des bases de données relationnelles au milieu des années 1970 dans lesquelles des modèles de graphes ont été proposés pour étendre les informations stockées dans des bases de données avec des réseaux sémantiques (d'abord défini dans [Sim72]).

Structures de données, langages et traitement des requêtes

Les modèles de données orientées graphes représentent les données sous forme de graphes fournissant des opérations de traversée efficaces pour les interroger et les analyser. Les graphes sont composés d'entités représentant quelque chose qui existe en tant qu'unité unique ou composée et de relations établissant des connexions entre deux entités ou plus. Quelques structures de

données utilisées pour modéliser les entités et les relations dans ce modèle sont: la matrice d'adjacence, la liste d'adjacence, la liste de bord, le compressed sparse row(CSR) et les indexes complexes.

Parmi les études de recherche, il existe plusieurs travaux sur les langages de requête. Ces travaux présentent des collections d'opérateurs et de règles d'inférence pour manipuler et interroger la structure de données orientées graphes. Compte tenu du grand nombre d'approches disponibles et de la portée de cette thèse, nous ne mentionnons dans cette section que les langages de requête de graphes les plus récents et les plus populaires. On retrouve alors: SPARQL, Cypher, Gremlin, Oracle PGQL et G-Core.

Les stratégies de traitement des requêtes dépendent du type de requête ciblé par l'application. Ces stratégies peuvent être aussi simples que de trouver les voisins directs d'un nœud ou de trouver des graphes isomorphes en fonction d'un modèle de requête, ou des chemins de traversée plus complexes utilisés dans des algorithmes tels que PageRank.Nous avons donné un aperçu des stratégies de traitement de graphes qui sont au cœur de plusieurs langages de requête. Une description détaillée de chaque algorithme et des stratégies plus complexes traitant de l'accessibilité se trouve dans [BFVY18].

Partitionnement de graphes

Au cours des dernières années, les données stockées sous forme de graphes ont considérablement augmenté. Par exemple, la taille du graphe de données Facebook compte un milliard de nœuds et environ deux cents milliards d'arêtes [LNP16]. Des solutions distribuées et parallèles ont vu le jour pour répondre à la demande de stockage et de traitement de graphes efficaces. Dans ces systèmes, couper le graphe en petits morceaux est une étape fondamentale pour permettre le parallélisme. Le partitionnement de graphes doit trouver un équilibre entre les objectifs suivants:

- Minimiser les coûts de communication, et
- Maximiser l'équilibre de charges.

Bases de données orientées graphes

Dans les sections précédentes, nous avons caractérisé les dimensions du modèle de base de données orientées graphe: i) Structure de données logique, ii) stockage de données, iii) langages de requête et de manipulation, et iv) stratégies de traitement des requêtes. Nous utilisons ces dimensions pour classer les systèmes les plus populaires. Nous classons les approches en deux grands groupes: *natif* et *non-natif*. Le premier groupe est constitué de systèmes spécialement conçus pour stocker et interroger des données représentées sous forme de graphes. En revanche, les approches non natives sont construites sur des plates-formes existantes telles que les bases de données relationnelles, les systèmes clé-valeur et d'autres systèmes comme par example le cloud.

Le Resource Description Framework (RDF)

Le Resource Description Framework (RDF) [RC14] a été largement accepté comme modèle de données standard pour l'échange de données issues du Web. RDF est suffisamment flexible pour faciliter l'intégration des données avec différents schémas. Le modèle utilise des triplets composés d'un sujet, d'un prédicat et d'un objet (s, p, o) comme structure de données abstraite de base pour représenter l'information. Un ensemble de tels triplets forme un graphe de connaissances (KG). Les nœuds représentent des IRIs (International Resource Identifiers), des nœuds vides (ressources non spécifiées) ou des littéraux. Les arcs caractérisent une ressource en reliant les nœuds sujet et objet par un prédicat. Ce chapitre classe les systèmes les plus représentatifs en natifs et non natifs. Nous nous concentrons ici sur les stratégies de partitionnement appliquées par la plupart de ces systèmes.

Partitionnement des données RDF

Pour faire face à l'augmentation des données RDF disponibles et à la nécessité de les traiter efficacement, les systèmes RDF ont eu recours à des techniques de distribution et de traitement parallèle largement explorées dans les bases de données relationnelles. Dans les systèmes RDF distribués, nous trouvons deux groupes, les systèmes *fédérés* et les systèmes en *cluster* qui sont décrits comme suit:

- *Systèmes fédérés*: ces systèmes exécutent des requêtes SPARQL sur plusieurs SPARQL endpoints[16]. De tels systèmes nécessitent d'effectuer l'intégration de données à la volée de plusieurs sources RDF hétérogènes. Les systèmes fédérés sont hors du champ de notre étude.

- *Systèmes en cluster*: ces systèmes distribuent les données entre différents nœuds qui forment une seule solution de stockage. Depuis plusieurs années, un grand nombre de systèmes distribués offrant un traitement efficace des données RDF ont vu le jour. Nous concentrons notre étude sur ce type de systèmes.

Nous divisons le processus de partitionnement en trois phases: Fragmentation, Allocation et Réplication. Ensuite, nous détaillons les entrées, les sorties et les algorithmes sur chacune de ces étapes.

- *Fragmentation:* Le processus de fragmentation consiste à définir l'unité de fragmentation à allouer sur les sites du système distribué. Il n'y a pas de consensus sur la façon dont les données RDF devraient être distribuées. La plupart des solutions actuelles utilisent le triple comme unité de fragmentation. Cependant,il a été affirmé que le regroupement des triples en premier pourrait améliorer les performances en découvrant des structures implicites dans le graphe RDF [BDK+13, SGK+08]. Par conséquent, d'autres solutions proposent de regrouper d'abord les triplets par sujet, prédicat ou objet et d'utiliser ces groupes comme unités de fragmentation.

- *Allocation:* L'étape de l'attribution consiste à trouver la distribution d'un ensemble de fragments $\{F_1, ..., F_n\}$ (qui sont le résultat de l'étape précédente) pour un ensemble de sites $\{S_1, ..., S_m\}$. La distribution des fragments se fait en appliquant les techniques suivantes: fonction de hachage, round-robin ou dépendant de la plateforme cloud.

- *Réplication:* La réplication des données est une stratégie d'optimisation fréquente mais pas obligatoire appliquée par plusieurs systèmes distribués RDF. Il permet également la tolérance aux pannes dans ces systèmes.

Conclusion

Dans ce chapitre, nous avons d'abord donné un aperçu général du modèle de données orientées graphes. Nous avons détaillé ses structures logiques et de stockage, les stratégies de traitement des requêtes, ses langages de requêtes, et enfin nous avons détaillé certains systèmes actuels. Nous avons vu les avantages du modèle mais aussi ses limites, notamment en raison de la variabilité des méthodes utilisées pour le partitionnement et traitement de données. Ensuite, nous nous sommes concentrés sur RDF. Nous avons donné quelques concepts de base, et aussi décrit les stratégies de stockage, de traitement et de partitionnement. Le partitionnement dans les triple stores dépend de l'organisation physique des données (c'est-à-dire stockées dans une seule table, une table de propriétés ou une liste adjacente). Nous avons montré que les stratégies de partitionnement RDF sont dépendantes aux systèmes. De plus, contrairement aux bases de données relationnelles, il n'y a pas de couche logique commune. Le partitionnement de données

[16]Services web RESTful pour exposer les données RDF interrogées avec SPARQL

RDF est *piloté par le système*, contrairement au partitionnement dans les bases de données relationnelles où il est *piloté par les données*. Dans les bases de données relationnelles, il existe des outils (par exemple, des systèmes automatiques de partitionnement, langages de définition) qui permettent au concepteur de base de données de créer des partitions en fonction de la structure des tables ou d'un schéma de partitionnement qui répond aux exigences d'une certaine charge de requêtes.

Relationnel	RDF
− L'intégration des données avec différents schémas est complexe.	+ Intégration flexible de nouvelles données.
+ *SGBD-indépendant*: le partitionnement est indépendant du système de gestion de base de données.	− *SGBD-dépendant*: le système d'exécution et la stratégie de partitionnement sont couplés.
+ *Partitionnement au niveau logique*: Indépendamment du modèle physique utilisé pour stocker les données (par exemple, NSM, DSM).	− Dépend du modèle de stockage physique (par exemple, basé sur les triplets).
+ Les langages de définition de base de données natifs prennent en charge la déclaration de partitions horizontales.	− Partitionnement effectué de manière transparente pendant la phase de chargement.
+ Plusieurs conseillers de partitionnement automatique sont disponibles pour aider le concepteur à créer des partitions.	− Le partitionnement des données est principalement imposé par le triple store.

Les stratégies de partitionnement adoptées par les systèmes RDF présentent également plusieurs avantages. Par exemple, la plupart des approches permet l'intégration des données. En effet, L'un des atouts de RDF est sa flexibilité pour intégrer des données avec différents schémas sous-jacents. Le modèle relationnel, en revanche, est souvent trop rigide pour gérer les données sans donner un schéma initial. Certaines approches de RDF ont essayé de modéliser les données RDF dans un schéma de base de données relationnelle avant de stocker les données dans un SGBDR. Cependant, comme le démontrent plusieurs études expérimentales, les systèmes relationnels ne sont pas optimaux lorsqu'ils traitent des requêtes très complexes. Les forces et les inconvénients (principalement mutuellement exclusifs) des deux modèles de données sont résumés dans le tableau précédent.

Nous considérons nécessaire de fournir des outils pour les concepteurs de systèmes RDF imitant le confort offert dans les systèmes de bases de données relationnelles. Cela nous a amené à fixer l'objectif de cette thèse qui consiste à proposer un framework de partitionnement de données RDF basé sur des structures logiques implicites. L'élaboration de ce framework (RDFPartSuite) sera abordée dans les prochains chapitres.

Partitionnement logique des triplets RDF

Nous nous appuyons sur les enseignements tirés du modèle relationnel dans lequel le partitionnement des données est effectué à un niveau logique, indépendamment de l'approche de stockage. Dans ce qui suit, nous introduisons une couche logique au processus de partitionnement des triple stores distribués homogènes. Nous commençons à donner un aperçu de cette couche logique en comparant notre stratégie à l'état actuel de l'art. Ensuite, nous définissons formellement les

entités logiques et proposons des algorithmes pour regrouper les triplets RDF en fragments de graphes. Ensuite, nous discutons de l'utilisation de ces fragments comme structures de stockage physique et formalisons leur problème d'allocation. Enfin, nous présentons un exemple complet des heuristiques de création et d'allocation définies dans ce chapitre.

La conception d'un système distribué implique la distribution de données et de programmes sur les sites du réseau informatique dans lequel le système est déployé. Nous nous concentrons sur le problème de la distribution des données en cherchant à trouver et à allouer les unités de distribution optimales. Pour rendre ce processus plus compréhensible, la distribution de données est subdivisée en [ÖV11] en deux sous-étapes. Le premier, nommé *fragmentation*, détermine les unités de distribution puis l'étape d'*allocation* qui place les fragments sur les sites du réseau.

La phase de conception déclarant explicitement les entités de haut niveau comme dans les bases de données relationnelles ne se produit pas dans les systèmes RDF. Au contraire, les données sont organisées dans un modèle sans schéma en utilisant des triples comme unités de distribution. Pourtant, les données relatives à la même entité de haut niveau peuvent être facilement dispersées dans le lot de données. Il n'y a pas de consensus sur la façon dont les données devraient être fragmentées et allouées. Actuellement, les stratégies de partitionnement dépendent fortement du système principal utilisé pour stocker les données. Il y a eu des efforts pour ajouter explicitement un schéma de hiérarchie de classes à RDF via des annotations (par exemple, schéma-RDFS et ontologies). De plus, comme indiqué dans [PPEB15], i) les entités d'un seul ensemble de données peuvent être décrites avec plusieurs ontologies, ii) toutes les entités d'un ensemble de données ne sont pas annotées avec les mêmes métadonnées et, iii) tous les modèles de requêtes SPARQL ne les considèrent pas. L'utilisation de ces annotations dans l'identification des entités implicites n'est pas très efficace.

Notre stratégie identifie des entités en utilisant les ensembles de caractéristiques [NM11]. Un ensemble de caractéristiques regroupe d'abord les triplets par sujet et collecte ensuite les données en fonction de leurs prédicats. Cette stratégie ne repose pas sur des annotations pour identifier les entités et elle permet de considérer certaines corrélations entre les triplets. Nous avons réutilisé cette notion pour détecter les entités décrites ci-dessous. Nous considérons nécessaire de fournir des outils pour les concepteurs de systèmes RDF imitant le confort offert dans les systèmes de bases de données relationnelles. Cela nous a amenés à fixer l'objectif de cette thèse qui consiste à proposer un framework de partitionnement de données RDF basé sur des structures logiques implicites. L'élaboration de ce cadre sera abordée dans les prochains chapitres.

Fragments de graphes

Dans cette section, nous proposons de regrouper les données inspirés du partitionnement relationnel. Dans les bases de données relationnelles, une entité (une table) est partitionnée au niveau de l'instance (horizontale) ou de l'attribut (verticale). Considérant qu'en RDF il n'y a pas d'étape de conception comme dans les bases de données relationnelles, notre proposition cherche à détecter les entités implicites dans un jeu de données RDF.

L'ensemble de caractéristiques proposé dans [NM11] permet de regrouper les instances d'une même entité de haut niveau pour former une partition. Nous définissons deux types d'entités: par instances et par attributs décrites ci-dessous.

Regroupement par instances

Pour rassembler les triples d'une même entité de haut niveau, nous regroupons en premier lieu les triples par ses sujets. Ces structures sont nommées *étoiles de données en avant*. Nous définissons d'abord les fonctions $f_s(t) \to s$, $f_p(t) \to p$ and $f_o(t) \to o$ renvoyant le sujet, le prédicat et l'objet d'un triplet RDF $t = \langle s, p, o \rangle$ respectivement. Ces deux fonctions sont appliquées dans certaines des définitions du présent chapitre. Une étoiles de données en avant rassemble toutes les propriétés associées à un nœud décrivant une seule occurrence d'une certaine entité. Les

entités peuvent être identifiées de manière unique par un sous ensemble de leurs bords émetteurs [NM11].

Un *fragment de graphe avant* rassemble des étoiles de données en avant selon leur ensemble caractéristique. Comme il a été mentionné précédemment, nous pouvons caractériser une entité par ses bords émetteurs. Un fragment de graphe avant est une structure *logique* utilisée pour rassembler les données liées à la même entité de haut niveau. Un fragment de graphe avant peut être utilisé comme structure logique pour distribuer les données.

En général, deux étoiles de données avant appartiennent à la même entité si les étiquettes de leurs bords émetteurs (c'est-à-dire les prédicats) sont les mêmes. Pourtant, parfois deux instances appartiennent à la même entité mais leurs ensembles caractéristiques ne sont pas exactement les mêmes. C'est le cas lorsque deux ensembles de caractéristiques ne diffèrent que par des prédicats non discriminants. Pour éviter de créer un grand nombre de fragments avec des critères de similarité très stricts, nous avons utilisé un score de similarité et un seuil. Pour simplifier la notation, la fonction $cs(\overrightarrow{\mathcal{G}f})$ retourne l'ensemble des caractéristiques (qui est l'identifiant) du fragment.

Scores de similarité La fonction de similarité $Sim(cs(s_i), cs(s_j))$ renvoie un score mesurant la similitude des deux ensembles de caractéristiques. Dans son calcul, il peut considérer uniquement la structure de l'ensemble de caractéristiques et leurs relations ou des caractéristiques sémantiques plus complexes. Parmi les fonctions de similarité structurelle, nous trouvons: Supersets, Tf-idf, ou enfin mesures sémantiques (même étiquette et basée sur les ancêtres).

Regroupement par attributs

L'organisation d'un graphe RDF en fragments de graphe avant identifie l'ensemble des instances implicites à l'aide d'ensembles de caractéristiques. Les fragments créés avec cette stratégie ressemblent aux groupes de tuples générés lors du partitionnement horizontal d'une base de données relationnelle. Comme dans le modèle relationnel, fragmenter au niveau instance est utile pour un certain type de requêtes. Par exemple, si la charge de travail est composée de requêtes en forme d'étoile, ce type d'organisation est idéal. Cependant, cette stratégie de fragmentation unique n'optimise pas tous les spectres de requêtes. Le partitionner d'une entité par ses attributs dans le modèle relationnel (partitionnement vertical), optimise d'autres types de requêtes qui impliquent un petit nombre d'attributs. Plusieurs systèmes de traitement RDF utilisent une stratégie similaire, où un fragment est créé pour chaque propriété (par exemple SW-Store [AMMH09], S2RDF [SPSL16]). Dans les systèmes relationnels, la stratégie stocke les triplets dans différentes tables par propriété. Cette stratégie est efficace lors de la résolution de requêtes avec quelques attributs, mais souffre de frais généraux lorsque plusieurs prédicats sont joints dans une requête.

Dans cette section, nous décrivons la création de fragments en regroupant les nœuds par leurs propriétés. Au lieu de regrouper strictement chaque propriété dans un fragment différent, nous utilisons la notion d'ensembles de caractéristiques pour regrouper les attributs affectant le même nœud. Nous nommons ces structures *fragments de graphe arrière* dont la construction est très similaire aux fragments de graphe avant décrits précédemment. Nous commençons par définir un *étoile de données arrière* qui regroupe un nœud et ses bords *entrants*. Cette structure permet d'identifier les propriétés qui affectent le même nœud pour les regrouper plus tard avec les mêmes seuils de similarité que pour les fragments de graphe avant.

Des fragments logiques aux structures physiques

L'organisation des données RDF dans des fragments de graphe permet de détecter des entités logiques implicites qui sont utilisées comme unités de distribution dans des triple stores distribués. De plus, l'utilisation de ces fragments logiques comme structures physiques pourrait con-

sidérablement améliorer les performances des systèmes centralisés et distribués. Les travaux de Pham et al. dans [PPEB15] a montré que le stockage explicite des données à l'aide d'un schéma relationnel découvert automatiquement améliore les performances de Virtuoso [EM09],un triple store non-natif. Organiser les données en fragments de graphe avant et arrière, quelle que soit la structure utilisée pour stocker les données (par exemple, tables, arbres ou listes d'adjacence), évite de scanner l'ensemble de données plusieurs fois avec une seule requête, comme cela est fait par la plupart des systèmes.

Partitionner le graphe RDF en fragments physiques peut éviter d'analyser l'ensemble des données avec chaque requête. Pour ce faire, les partitions pertinentes doivent être identifiées en fonction des informations disponibles dans la requête. Les fragments en avant et en arrière permettent d'identifier les partitions pertinentes en fonction des prédicats de la requête, qui sont connus dans la plupart des requêtes SPARQL.

Combinaison de fragments avant et arrière L'intégration des partitions horizontales et verticales dans le modèle relationnel a été explorée par de nombreux chercheurs dans le passé. Par exemple, les conseillers d'une conception physique de base de données décrite dans [ANY04] ou les miroirs fracturés [RDS02]. Pendant ce temps, de nombreux systèmes de traitement massifs proposent des stratégies de réplication non seulement pour récupérer et prendre en charge la tolérance aux pannes, mais pour améliorer le temps de réponse des requêtes. Nous considérons une approche similaire aux miroirs fracturés [RDS02] dans laquelle un système stocke deux copies des données. Une copie organise les données comme des fragments à l'avant et une autre en arrière. Cette configuration est utile en particulier lorsque la charge de travail est inconnue lors de la phase de partitionnement initiale.

Problème d'allocation

L'allocation de fragments est une étape obligatoire dans les systèmes distribués. Le problème consiste à trouver la distribution optimale des fragments sur les sites d'un réseau informatique. Le problème a d'abord été étudié dans le contexte de la distribution de fichiers, puis approfondi dans les bases de données relationnelles où il s'est avéré NP-complet [Esw74, SW85, LY80]. Dans le modèle relationnel, l'optimisation d'une stratégie dépend de divers critères qui sont évalués sur la base de modèles de coûts. Ces modèles estiment les coûts de stockage et de maintenance ainsi que des mesures de performance telles que le débit et le temps de réponse. Étant donné que la complexité du problème ne permet pas de calculer des solutions exactes dans un temps raisonnable, un certain nombre d'heuristiques différentes utilisées en recherche opérationnelle(e.g. knapsack problem [CMP82]) ont été appliqués dans le SGBDR. Ces techniques sont aussi utilisées dans les architectures distribuées modernes comme Hadoop, qui, bien qu'elles soient basées sur d'autres paradigmes d'exécution, font face au même problème de distribution des données.

De même, la plupart des triple stores optent pour des solutions de distribution très simples. Ces solutions ne garantissent pas que les triples étroitement liés les uns aux autres seraient dans le même site. Une requête qui fusionne des résultats intermédiaires qui ne se trouvent pas dans la même machine est très inefficace principalement en raison des coûts de transfert élevés. Comme dans les bases de données relationnelles distribuées, nous considérons les coûts de réseau comme le goulot d'étranglement de traitement lors de l'exécution de la requête. Par conséquent, les stratégies visant à améliorer le rendement du système devraient viser à maximiser la localisation des données. Cela peut être réalisé grâce à des techniques d'indexation, de partitionnement et de réplication.

L'utilisation de fragments de graphes comme structures de stockage physique réduit les coûts de disque et permet le parallélisme dans les systèmes centralisés et distribués. Nous considérons l'utilisation de fragments de graphes avant et arrière pour réduire le coût du réseau ces systèmes.

Nous cherchons à allouer des triples au plus près de ses voisins pour élaguer les résultats intermédiaires générés pour une requête localement sur chaque site. Nous supposons que la charge de travail n'est pas disponible et donc nous construisons notre modèle de distribution basé sur la connectivité innée des données. Le problème est soumis à une série de contraintes qui sont le déséquilibre, la réplication et l'espace disque disponible.

Heuristique de partitionnement de graphe

L'ensemble de fragments de graphe est mappé à un graphe pondéré non orienté, transformant le problème d'allocation en un problème de partitionnement de graphe. Le problème de partitionnement de graphes s'est avéré être un problème très complexe et coûteux en calcul. Cependant, de nombreuses heuristiques efficaces ont été développées comme par example le package METIS [KK98a]. Nous recherchons une partition qui minimise le nombre total de coupes de bord soumises aux mêmes contraintes définies précédemment.

Conclusion

Dans le monde des bases de données relationnelles, le partitionnement des données est identifié depuis longtemps comme une technique clé d'optimisation et de gestion. Le partitionnement des données dans les SGBDR est indépendant de la technique de stockage des données. Dans ce chapitre, nous prétendons reproduire cette stratégie en abordant le problème du partitionnement des données RDF. Contrairement aux techniques de partitionnement traditionnelles, les techniques RDF dépendent de la stratégie de partitionnement et sont difficiles à généraliser pour différents systèmes.

Nous nous appuyons sur la philosophie adoptée par le modèle relationnel pour aborder le partitionnement au sein de systèmes RDF distribués. Précisément, nous avons introduit une couche logique au processus de distribution simplement physique des triplets à un ensemble de sites. Nous formalisons et détaillons les algorithmes utilisés pour créer les entités logiques que nous avons nommées fragments de graphes ($\mathcal{G}f$. Nos entités étendent la notion de partitionnement par instances et par attributs dans le modèle relationnel et offrent un grand équilibre entre simplicité conceptuelle et intuitive, en plus de son expressivité logique. Nous avons formalisé le problème d'allocation et présenté une heuristique de graphes pour minimiser les coûts de communication.

Notre framework RDFPartSuite en Action

Dans le chapitre précédent, nous avons promu une couche logique pour le partitionnement des données RDF. Nous avons défini un ensemble de structures qui permettent de partitionner les données RDF indépendamment de la façon dont elles sont stockées. Ces structures préservent la nature logique de graphe des données RDF et sont utilisées comme unités de fragmentation. Les fragments logiques sont placés sur les sites d'un système distribué selon des algorithmes pilotés par les données que nous avons également détaillés. Dans ce chapitre, nous introduisons un framework nommé *RDFPartSuite*. Le framework fournit des fonctionnalités génériques basées sur les structures logiques définies dans le chapitre précédent. Il peut être adapté en fonction des spécifications du gestionnaire (exigences du créateur, triple store, infrastructure disponible, etc.). *RDFPartSuite* fournit un moyen standard de construire et de déployer des schémas de partitionnement RDF dans un environnement universel et *réutilisable*. Le framework est composé de trois modules principaux:

(i) *Fragmenteur:* ce composant est chargé de partitionner les triples en utilisant leur représentation logique sous forme de fragments de graphe (c'est-à-dire $\overrightarrow{\mathcal{G}f}$ or $\overleftarrow{\mathcal{G}f}$).Les fragments peuvent être construits en utilisant des règles structurelles ou sémantiques.

(ii) *Allocateur:* ce composant crée un schéma de distribution pour les fragments créés par le fragmenteur. Ce schéma est construit à l'aide des stratégies basées sur les données détaillées précédemment.

(iii) *Dispatcher:* ce composant envoie les fragments aux sites d'un système distribué suivant le schéma d'allocation produit par l'allocateur. Le dispatcheur charge aussi les données dans le triple store cible.

En plus de ces modules, notre framework intègre un ensemble d'outils d'assistance pour les administrateurs de triple stores. Ces outils comprennent un langage de partitionnement déclaratif, pour fragmenter, allouer et envoyer un graphe de connaissances à un triple store cible. Et, pour les utilisateurs non experts, il fournit un assistant de partitionnement pour les aider à créer un schéma de partitionnement.

Nous commençons par décrire comment intégrer notre framework à un triple store centralisé (*RDF_QDAG* [KMG+20]). Ce système utilise des techniques de partitionnement de données et d'exploration de graphes pour accélérer l'exécution des requêtes. Nous avons également montré comment intégrer notre framework dans un triple store distribué (*gStoreD* [ZÖC+14]). Nous avons réalisé un nombre important d'expériences qui ont montré la faisabilité de nos stratégies de partitionnement, son efficacité et ses limites.

Incorporation au RDF_QDAG

Les systèmes de traitement RDF modernes peuvent être distingués en deux groupes. Le premier groupe utilise le modèle relationnel pour stocker les triplets RDF dans des tables. Les systèmes de ce groupe sont plus facilement scalables car ils peuvent utiliser des stratégies d'optimisation telles que les indexes et les partitions disponibles pour les bases de données relationnelles. Malheureusement, les performances de ces systèmes se dégradent rapidement, en particulier lorsqu'il s'agit de requêtes SPARQL complexes. Le modèle relationnel ne convient pas pour gérer les données RDF représentées de manière inhérente sous forme de graphe. Le deuxième groupe comprend des systèmes de traitement RDF maintenant la structure de graphe des données RDF. Ces systèmes stockent les triplets dans des structures de données spécialement conçues pour les stocker. Le problème majeur auquel sont confrontées ces approches est la scalabilité [AHKK17, Özs16]. Ils ne parviennent pas à gérer efficacement l'utilisation de la mémoire principale dans les infrastructures avec des ressources limitées.

RDF_QDAG [KMG+20] il combine les vertus du partitionnement des données dans les systèmes relationnels avec un modèle d'exécution basé sur l'exploration du graphique RDF. Le système prend en charge les requêtes SPARQL Basic Graph Pattern (BGP) ainsi que les opérateurs de caractères génériques, d'agrégation et de tri. Les données dans RDF_QDAG sont physiquement partitionnés en fragments de graphe avant et arrière. Il est capable d'élaguer des fragments de graphe non pertinents en utilisant les valeurs des prédicats dans la requête. Cette fonctionnalité permet d'explorer uniquement les fragments dans lesquels une correspondance est susceptible d'être trouvée. Comme mentionné précédemment, l'exécution de la requête dans ce système est basée sur l'exploration de graphes. Pour éviter les débordements de mémoire, le moteur d'exécution de RDF_QDAG est basé sur le système d'évaluation de requête parallèle Volcano [Gra94].

Ce système est utilisé pour évaluer les temps de chargement, la couverture des données et pour comparer les performances par rapport à d'autres systèmes en utilisant différentes stratégies de partitionnement. Le module de chargement prépare, indexe et partitionne les données RDF brutes reçues en entrée. Il est composé de deux composants principaux: le module de prétraitement et le module de stockage. Le premier transforme les données RDF (stockées dans des fichiers N-Triples, N3 ou Turtle) codant toutes les chaînes de caractères et organisant les données en fragments de graphes avant et arrière. Le module de prétraitement est entièrement codé en Java. Le composant de stockage, codé en C++, prend le fichier affiné codé du Pré-traitement

en entrée et configure les données dans les index. Le résultat de cette étape est un ensemble de fichiers binaires utilisés par les modules en charge du traitement de requêtes.

Coûts de chargement L'identification et la formulation de fragments de graphes avant et arrière dans RDF_QDAG est certainement un processus plus complexe que celui appliqué par d'autres systèmes chargeant les données directement dans une seule table relationnelle par exemple. L'ensemble de données est lu plusieurs fois pour identifier et encoder les données avec le format souhaité. Dans cette section, nous comparons le temps de prétraitement des systèmes RDF avec différents paradigmes de stockage et d'exécution. Le module de chargement de RDF_QDAG était le seul capable de charger tous les jeux de données étant donné la mémoire principale limitée (32 Go). Le système basé sur des graphiques gStore n'a pas pu charger des ensembles de données dont la taille est supérieure à la mémoire principale disponible. Les systèmes relationnels ont réussi à charger tous les ensembles de données sauf LUBM1B dans lequel nous avons eu une erreur de débordement de mémoire.

Performance des requêtes Les vertus de la stratégie de partitionnement logique ont été montrées tout au long de ces sections. Sans aucun doute, une stratégie de partitionnement logique accorde beaucoup plus de liberté aux concepteurs et aux gestionnaires. Ils ne dépendent plus d'un système spécifique pour partitionner leurs données. Nous avons montré la faisabilité d'intégrer notre cadre dans des magasins triples centralisés et distribués. De plus, les expériences utilisant RDF_QDAG révèlent que l'utilisation de ces structures dans un triple store natif centralisé garantit son scalabilité et de bonnes performances pour certaines requêtes.

Incorporation au gStoreD

Nous comparons les performances de l'organisation des données sous forme de fragments de graphe vers l'avant ou vers l'arrière dans un triple store distribué. Nous déterminons pour quel type de requêtes l'organisation des données en tant que fragments de graphe en avant est plus approprié que les fragments de graphe en arrière et vice versa. À notre connaissance, il n'existe pas de système RDF basé sur des graphes stockant les données exclusivement sous forme de fragments de graphes en arrière. En effet, RDF_QDAG utilise les deux structures (c'est-à-dire $\overrightarrow{\mathcal{G}f}$ et $\overleftarrow{\mathcal{G}f}$) et gStore stocke les données dans des listes d'adjacence regroupant les données par sujets. Pour les expériences de cette section, nous avons adapté la version distribuée de gStore [PZÖ+16] pour prendre en charge le stockage des fragments de graphe avant d'abord, puis des fragments de graphe arrière. Les résultats de cette section font partie de l'étude expérimentale de notre étude: *RDFPartSuite: Bridging Physical and Logical RDF Partitioning* [GMB19].

Conclusion

Dans ce chapitre, nous avons présenté notre cadre d'action. Nous avons utilisé les modules de fragmentation et d'allocation pour charger les données dans un triple store centralisé et distribué. Nous avons effectué une série de tests sur chacun des systèmes pour montrer les forces et les limites de notre framework. Nous avons commencé à donner un aperçu du triple store RDF_-QDAG. Ce système stocke les données sous forme de fragments de graphe avant et arrière. Nous avons intégré nos résultats au module de chargement de ce système. Nous avons comparé les coûts de chargement de RDF_QDAG avec d'autres systèmes représentatifs de l'état de l'art en termes de temps et de scalabilité. Nous avons montré que le processus de fragmentation avec nos entités ($\mathcal{G}f$) est réalisable, même dans les architectures avec des ressources limitées. Même si les temps de chargement étaient élevés par rapport à d'autres techniques de fragmentation plus légères (par exemple, triple table), nous avons vu les avantages lors de l'exécution des requêtes de ce coût supplémentaire.

Ensuite, nous avons effectué une évaluation complète des stratégies de fragmentation. Nous avons commencé à montrer que le nombre de fragments est raisonnable par rapport à la taille des ensembles de données. Ensuite, nous utilisons notre framework pour fragmenter les données en fragments de graphes avant et arrière dans un triple store distribué basé sur des graphes (gStoreD). Ce système a été utilisé pour comparer les stratégies de fragmentation avant et arrière. Ensuite, nous avons montré que la combinaison des deux types de fragments dans un seul système contribue non seulement à la scalabilité, mais également à améliorer les performances des requêtes. Nous avons réalisé une étude approfondie sur les coûts de communication en comparant notre stratégie de partitionnement avec les techniques les plus pertinentes. Nous avons montré que nos stratégies offrent un compromis entre le coût d'allocation, mesuré en termes de temps de chargement, de qualité de distribution et de performances. Enfin, nous introduisons une extension conviviale à notre framework. Ces extensions sont un langage de partitionnement déclaratif construit sur les structures logiques du chapitre précédent et un conseiller de partitionnement.

Conclusions et Perspectives

Actuellement, notre monde est rempli de données. Une grande partie de ces données est disponible gratuitement sur le Web. Le W3C a consacré de nombreux efforts au développement de normes facilitant l'exploitation, l'échange et la réutilisation de ces données. Parmi ces standards, RDF excelle pour sa flexibilité, sa simplicité et l'expressivité de son langage de requête (SPARQL). Les déclarations RDF sont des triples logiquement représentés dans un graphe. Ces graphes interconnectent des données provenant de plusieurs ensembles de données constituant un graphe de connaissances (KG). La complexité de ce dernier concerne à la fois son écosystème qui implique plusieurs acteurs (par exemple, les créateurs, les consommateurs, les gestionnaires) et les infrastructures, et sa taille. Pour satisfaire les exigences de ces acteurs, chaque KG doit être efficacement stocké et interrogé pour faciliter les différents processus de son exploitation.

L'infrastructure de stockage, et en particulier les Triple Stores, sont une pierre angulaire pour répondre à ces besoins. Cela a poussé des chercheurs de diverses communautés à proposer une grande collection de ces systèmes. Certains sont construits sur des solutions existantes (comme les SGBDR), d'autres sont construits en utilisant des implémentations de stockage personnalisées. Ils ont tous en commun l'utilisation de stratégies d'optimisation physique qui sont *spécifiques* à chaque système. Ces stratégies généralement *ignorent* le *schéma logique* du KG. Ceci est loin des optimisations basées sur des structures logiques largement étudiées dans les bases de données relationnelles. Dans ces stratégies, nous trouvons *data partitioning* dans la première ligne.

Le partitionnement des données accorde aux gestionnaires des moyens d'améliorer les performances, la gérabilité et la disponibilité des triple stores centralisés en divisant les données en fragments. Pourtant, le partitionnement des données dans les triple stores centralisés n'a pas eu la place qui lui revient dans la conception de ces systèmes. Pour des raisons de scalabilité, de nombreux triple stores parallèles et distribués ont vu le jour. Dans ces systèmes, le partitionnement des données devient une évidence. Il existe plusieurs modes de partitionnement (hachage, partitionnement graphique, hachage sémantique, etc) qui sont appliqués au niveau des triplets. Récemment, il y a eu un boom de recherche pour étudier l'impact sur la performance de ces stratégies de placement de données dans les triple stores[JSL20, ANS18].

Notre vision encourage la réutilisation et la reproduction des points forts du partitionnement de données appliqué dans différentes générations de bases de données qui ont précédé les bases de données RDF. Pour ce faire, il est nécessaire de rendre l'environnement de partitionnement *explicite*, en identifiant ses forces, son évolution et la manière dont leurs principaux éléments peuvent être réutilisés dans le contexte de KGs. Nous avons étudié le problème du partitionnement des données pour établir les bases qui guideront le développement de notre framework de

partitionnement. Dans cette thèse, nous promouvons la vision de la reproduction des résultats du partitionnement de données dans le contexte des triple stores.

Cette vision est intégrée dans notre framework de partitionnement de données RDF pour triple stores centralisés, distribués et parallèles. Le framework suit le paradigme d'être *en dehors* du SGBD afin qu'il puisse être *réutilisé*, amélioré et étendu. Notre framework fonctionne en utilisant une représentation logique des fragments qui permettent de partitionner KGs indépendamment de la représentation de stockage de chaque système. Enfin, nous montrons comment ce framework a été implémenté dans des triple stores centralisés et distribués. Nous avons mené plusieurs expérimentations qui ont montré quand ces stratégies de partitionnement sont les plus efficaces.

Perspectives

Extensions de notre framework

Composant de calibration Ce composant serait chargée de surveiller les différents schémas générés par notre framework en utilisant des approches basées sur les données. Le calibrage est réalisé par l'exploitation des charges de requêtes en cours dans le triple store. Nous utilisons la charge de requêtes pour *améliorer* la qualité de l'allocation initiale. Cette tâche pourrait être divisée en deux sous-modules: gestionnaire de charge de travail et réaffectateur intelligent.

Autres règles d'allocation

Actuellement, le composant allocateur de notre framework implémente des algorithmes pilotés par les données pour placer des fragments de graphes sur les sites d'un système distribué. Cette stratégie est basée sur la connectivité inhérente des triples. Cependant, nous pouvons imaginer d'autres règles basées sur les données pour étendre notre cadre. Par exemple, pour les ensembles de données contenant des informations géospatiales.

Prise en compte d'autres techniques d'optimisation

Pour le moment, notre framework est utilisé pour partitionner KGs. Nous sommes conscients que le partitionnement des données n'est qu'une des stratégies d'optimisation à prendre en compte lors de la conception de systèmes de stockage efficaces. Tenir compte de l'impact du partitionnement des données et de son interaction avec les index, les répliques, les vues matérialisées, etc. est la prochaine étape.

Stratégies d'optimisation dynamique

Aujourd'hui, il existe de nombreux triple stores distribués à mémoire principale très efficaces. Dans ces systèmes, des optimisations dynamiques rapides de la distribution des données peuvent être envisagées. Ces optimisations peuvent prendre en compte les flux de requêtes pour décider des stratégies de re-partitionnement ou de réplication. Un composant peut être ajouté à notre cadre pour traiter spécifiquement de cette réorganisation en ligne.

Reproductibilité de notre framework à d'autres types de données

Dans cette thèse, nous présentons une vision qui inclut le processus de conception des triple stores. Notre vision peut être réutilisée/adaptée aux contextes d'autres types de données existants qui existent aujourd'hui (par exemple, les graphes de propriétés) ou des modèles de données qui sont encore à venir.

Un référentiel pour les résultats de partitionnement de données

En présentant notre thèse, nous avons fait de notre mieux pour promouvoir l'idée de reproductivité des points forts du partitionnement de données dans les bases de données traditionnelles (déployées dans des infrastructures centralisées et parallèles) dans les triple stores. Si nous approfondissons notre analyse, le développement d'un référentiel (un graphe de connaissances) contenant toutes les conclusions du partitionnement des données dans les bases de données traditionnelles et RDF sera un grand atout pour les étudiants, les universités, l'industrie, les développeurs de systèmes Open Sources, pour n'en nommer que quelques-uns.

References

[AAB+17] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. Foundations of modern query languages for graph databases. *ACM Comput. Surv.*, 50(5):68:1–68:40, 2017. (Cited in page 66)

[AAB+18] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter A. Boncz, George H. L. Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan F. Sequeda, Oskar van Rest, and Hannes Voigt. G-CORE: A core for future graph query languages. In *Proceedings of the International Conference on Management of Data, SIGMOD, Houston, TX, USA, June 10-15*, pages 1421–1432, 2018. (Cited in page 68)

[AAK+16] Razen Al-Harbi, Ibrahim Abdelaziz, Panos Kalnis, Nikos Mamoulis, Yasser Ebrahim, and Majed Sahli. Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning. *VLDB Journal*, 25(3):355–380, 2016. (Cited in page 90)

[ABA+09] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel J. Abadi, Alexander Rasin, and Avi Silberschatz. Hadoopdb: An architectural hybrid of mapreduce and DBMS technologies for analytical workloads. *PVLDB*, 2(1):922–933, 2009. (Cited in page 58)

[ACZH10] Medha Atre, Vineet Chaoji, Mohammed J. Zaki, and James A. Hendler. Matrix "bit" loaded: a scalable lightweight join query processor for RDF data. In *19th International Conference on World Wide Web, WWW, Raleigh, North Carolina, USA, April 26-30*, pages 41–50, 2010. (Cited in page 84)

[ADH02] Anastassia Ailamaki, David J. DeWitt, and Mark D. Hill. Data page layouts for relational databases on deep memory hierarchies. *VLDB J.*, 11(3):198–215, 2002. (Cited in page 82)

[ADHS01] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving relations for cache performance. In *Proceedings of 27th International Conference on Very Large Data Bases VLDB, September 11-14, Roma, Italy*, pages 169–180, 2001. (Cited in page 58)

[AG08] Renzo Angles and Claudio Gutiérrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, 2008. (Cited in page 63)

[AHKK17] Ibrahim Abdelaziz, Razen Harbi, Zuhair Khayyat, and Panos Kalnis. A survey and experimental comparison of distributed SPARQL engines for very large RDF data. *Proc. VLDB Endow.*, 10(13):2049–2060, 2017. (Cited in page 80), (Cited in

page 86), (Cited in page 99), (Cited in page 122), (Cited in page 128), (Cited in page 155), (Cited in page 157), (Cited in page 174)

[AHÖD14] Günes Aluç, Olaf Hartig, M. Tamer Özsu, and Khuzaima Daudjee. Diversified stress testing of RDF data management systems. In *Proceedings of the 13th International Semantic Web Conference - ISWC, Riva del Garda, Italy, October 19-23*, pages 197–212, 2014. (Cited in page 128)

[AM09] Sean Bechhofer Alistair Miles. Skos simple knowledge organization system. `https://www.w3.org/TR/skos-reference/`, 2009. (Cited in page 1)

[AMMH09] Daniel J. Abadi, Adam Marcus, Samuel Madden, and Kate Hollenbach. Sw-store: a vertically partitioned DBMS for semantic web data management. *VLDB Journal*, 18(2):385–406, 2009. (Cited in page 6), (Cited in page 83), (Cited in page 104), (Cited in page 131), (Cited in page 146), (Cited in page 171)

[Amo10] Rasmus Resen Amossen. Vertical partitioning of relational OLTP databases using integer programming. In *Workshops Proceedings of the 26th International Conference on Data Engineering, ICDE, March 1-6, Long Beach, California, USA*, pages 93–98, 2010. (Cited in page 36), (Cited in page 37), (Cited in page 51)

[ANS18] Adnan Akhter, Axel-Cyrille Ngonga Ngomo, and Muhammad Saleem. An empirical evaluation of RDF graph partitioning techniques. In *21st International Conference on Knowledge Engineering and Knowledge Management - EKAW, Nancy, France, November 12-16*, pages 3–18, 2018. (Cited in page 153), (Cited in page 176)

[ANY04] Sanjay Agrawal, Vivek R. Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18*, pages 359–370, 2004. (Cited in page 8), (Cited in page 35), (Cited in page 47), (Cited in page 48), (Cited in page 51), (Cited in page 108), (Cited in page 172)

[AÖD14] Gunes Aluc, M. Tamer Özsu, and Khuzaima Daudjee. Workload matters: Why RDF databases need a new design. *Proc. VLDB Endow.*, 7(10):837–840, 2014. (Cited in page 97)

[ASAB16] Mohammed Al-Kateb, Paul Sinclair, Grace Au, and Carrie Ballinger. Hybrid row-column partitioning in teradata. *PVLDB*, 9(13):1353–1364, 2016. (Cited in page 29), (Cited in page 41), (Cited in page 51)

[ASH08] Medha Atre, Jagannathan Srinivasan, and James A. Hendler. Bitmat: A main-memory bit matrix of RDF triples for conjunctive triple pattern queries. In *7th International Semantic Web Conference (ISWC), Karlsruhe, Germany, October 28*, 2008. (Cited in page 85)

[ASYNN20] Waqas Ali, Muhammad Saleem, Bin Yao, and Axel-Cyrille Ngonga Ngomo. Storage, indexing, query processing, and benchmarking in centralized and distributed RDF engines: A survey. 05 2020. (Cited in page 80), (Cited in page 86)

[BAC+90] Haran Boral, William Alexander, Larry Clay, George P. Copeland, Scott Danforth, Michael J. Franklin, Brian E. Hart, Marc G. Smith, and Patrick Valduriez. Prototyping bubba, A highly parallel database system. *IEEE Trans. Knowl. Data Eng.*, 2(1):4–24, 1990. (Cited in page 39), (Cited in page 52)

[BAC+13] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C. Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkateshwaran Venkataramani. TAO: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28*, pages 49–60, 2013. (Cited in page 78)

[BBA09] Ladjel Bellatreche, Kamel Boukhalfa, and Zaia Alimazighi. Simulph.d.: A physical design simulator tool. In *20th International Conference on Database and Expert Systems Applications, DEXA, Linz, Austria, August 31 - September 4*, pages 263–270, 2009. (Cited in page 9)

[BBB19] Wissem Bouarroudj, Zizette Boufaïda, and Ladjel Bellatreche. Welink: A named entity disambiguation approach for a QAS over knowledge bases. In *13th International Conference on Flexible Query Answering Systems, FQAS, Amantea, Italy, July 2-5*, pages 85–97, 2019. (Cited in page 4)

[BBC14] Soumia Benkrid, Ladjel Bellatreche, and Alfredo Cuzzocrea. A global paradigm for designing parallel relational data warehouses in distributed environments. *Trans. Large Scale Data Knowl. Centered Syst.*, 15:64–101, 2014. (Cited in page 7)

[BBKO20] Nabila Berkani, Ladjel Bellatreche, Selma Khouri, and Carlos Ordonez. A model-agnostic recommendation explanation system based on knowledge graph. In *31st International Conference on Database and Expert Systems Applications - DEXA, Bratislava, Slovakia, September 14-17*, pages 149–163, 2020. (Cited in page 4)

[BBM07] Ladjel Bellatreche, Kamel Boukhalfa, and Mukesh K. Mohania. Pruning search space of physical database design. In *Proceedings of the 18th International Conference on Database and Expert Systems Applications DEXA, Regensburg, Germany, September 3-7*, pages 479–488, 2007. (Cited in page 46), (Cited in page 52)

[BBRW09] Ladjel Bellatreche, Kamel Boukhalfa, Pascal Richard, and Komla Yamavo Woameno. Referential horizontal partitioning selection problem in data warehouses: Hardness study and selection algorithms. *IJDWM*, 5(4):1–23, 2009. (Cited in page 9), (Cited in page 10), (Cited in page 28), (Cited in page 46), (Cited in page 52), (Cited in page 163), (Cited in page XV)

[BCLS87] Thang Nguyen Bui, Soma Chaudhuri, Frank Thomson Leighton, and Michael Sipser. Graph bisection algorithms with good average case behavior. *Combinatorica*, 7(2):171–191, 1987. (Cited in page 73)

[BCN17] Nikita Bobrov, George A. Chernishev, and Boris Novikov. Workload-independent data-driven vertical partitioning. In *Proceedings of New Trends in Databases and Information Systems - ADBIS, Nicosia, Cyprus, September 24-27*, pages 275–284, 2017. (Cited in page 36), (Cited in page 37), (Cited in page 51)

[BCR97] Lorenzo Brunetta, Michele Conforti, and Giovanni Rinaldi. A branch-and-cut algorithm for the equicut problem. *Math. Program.*, 77:243–263, 1997. (Cited in page 72)

[BDK+13] Mihaela A. Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. Building an efficient RDF store over a relational database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, New York, NY, USA, June 22-27*, pages 121–132, 2013. (Cited in page 84), (Cited in page 87), (Cited in page 168)

[Bel18] Ladjel Bellatreche. Optimization and tuning in data warehouses. In *Encyclopedia of Database Systems, Second Edition, edited by Ling Liu and Tamer Özsu*. 2018. (Cited in page 9), (Cited in page XVII)

[BEP+08] Kurt D. Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Vancouver, BC, Canada, June 10-12*, pages 1247–1250, 2008. (Cited in page 3)

[BFVY18] Angela Bonifati, George H. L. Fletcher, Hannes Voigt, and Nikolay Yakovets. *Querying Graphs*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2018. (Cited in page 65), (Cited in page 69), (Cited in page 70), (Cited in page 71), (Cited in page 125), (Cited in page 167)

[BHS03] Valerie Bönström, Annika Hinze, and Heinz Schweppe. Storing RDF as a graph. In *1st Latin American Web Congress (LA-WEB 2003), Empowering Our Web, 10-12 November 2003, Sanitago, Chile*, pages 27–36, 2003. (Cited in page 85)

[BKL98] Ladjel Bellatreche, Kamalakar Karlapalem, and Qing Li. Derived horizontal class partitioning in oodbs: Design strategies, analytical model and evaluation. In *Proceedings of the 17th International Conference on Conceptual Modeling ER, Singapore, November 16-19*, pages 465–479, 1998. (Cited in page 10), (Cited in page 45), (Cited in page 52)

[BKMS00] Ladjel Bellatreche, Kamalakar Karlapalem, Mukesh K. Mohania, and Michel Schneider. What can partitioning do for your data warehouses and data marts? In *Proccedings of the International Database Engineering and Applications Symposium, IDEAS, September 18-20, Yokohoma, Japan,*, pages 437–446, 2000. (Cited in page 46), (Cited in page 52)

[BKS97] Ladjel Bellatreche, Kamalakar Karlapalem, and Ana Simonet. Horizontal class partitioning in object-oriented databases. In *Proceedings of the 8th International Conference on Database and Expert Systems Applications , DEXA, Toulouse, France, September 1-5*, pages 58–67, 1997. (Cited in page 10), (Cited in page 21), (Cited in page 45), (Cited in page 162)

[BKS00] Ladjel Bellatreche, Kamalakar Karlapalem, and Ana Simonet. Algorithms and support for horizontal class partitioning in object-oriented databases. *Distributed and Parallel Databases*, 8(2):155–179, 2000. (Cited in page 45), (Cited in page 52)

[BKvH02] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF schema. In *Proceedings of the First International Semantic Web Conference - ISWC, , Sardinia, Italy, June 9-12*, pages 54–68, 2002. (Cited in page 83)

[BLN09] Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. k2-trees for compact web graph representation. In *16th International Symposium on String Processing and Information Retrieval, SPIRE, Saariselkä, Finland, August 25-27*, pages 18–30, 2009. (Cited in page 85)

[BLS09] Rajesh Bordawekar, Lipyeow Lim, and Oded Shmueli. Parallelization of XPath queries using multi-core processors: challenges and experiences. In *12th International Conference on Extending Database Technology EDBT, Saint Petersburg, Russia, March 24-26*, pages 180–191, 2009. (Cited in page 10)

[BMS+16] Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning. In *Algorithm Engineering - Selected Results and Surveys*, pages 117–158. 2016. (Cited in page 72)

[BS93] Stephen T. Barnard and Horst D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing, PPSC, Norfolk, Virginia, USA, March 22-24*, pages 711–718, 1993. (Cited in page 73)

[BSG18] Luigi Bellomarini, Emanuel Sallinger, and Georg Gottlob. The Vadalog system: Datalog-based reasoning for knowledge graphs. *Proc. VLDB Endow.*, 11(9):975–987, 2018. (Cited in page 3)

[BV04] Paolo Boldi and Sebastiano Vigna. The webgraph framework I: compression techniques. In *Proceedings of the 13th international conference on World Wide Web, WWW, New York, NY, USA, May 17-20*, pages 595–602, 2004. (Cited in page 85)

[CBK+10] Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Estevam R. Hruschka Jr., and Tom M. Mitchell. Toward an architecture for never-ending language learning. In *Proceedings of the Twenty-Fourth Conference on Artificial Intelligence, AAAI, Atlanta, Georgia, USA, July 11-15*, 2010. (Cited in page 4)

[CBN07] Eric Chu, Jennifer L. Beckmann, and Jeffrey F. Naughton. The case for a wide-table approach to manage sparse relational data sets. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14*, pages 821–832, 2007. (Cited in page 84), (Cited in page 98), (Cited in page 99)

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, 2008. (Cited in page 57)

[CEK+15] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at Facebook-scale. *Proc. VLDB Endow.*, 8(12):1804–1815, 2015. (Cited in page 79)

[CFL18] Matteo Cossu, Michael Färber, and Georg Lausen. PRoST: Distributed execution of SPARQL queries using mixed partitioning strategies. In *EDBT*, pages 469–472, 2018. (Cited in page 90)

[CG97] Gajanan S. Chinchwadkar and Angela Goh. Method transformations for vertical partitioning in parallel and distributed object databases. In *Proceedings of the Third International Parallel Processing Conference Euro-Par, Passau, Germany, August 26-29*, pages 1135–1143, 1997. (Cited in page 45), (Cited in page 51)

[CGZT14] Yu Cao, Xiaoyan Guo, Baoyao Zhou, and Stephen Todd. HOPE: iterative and interactive database partitioning for OLTP workloads. In *IEEE 30th International Conference on Data Engineering ICDE, Chicago, IL, USA, March 31 - April 4*, pages 1274–1277, 2014. (Cited in page 43), (Cited in page 44), (Cited in page 52)

[Che76] Peter P. Chen. The entity-relationship model - toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976. (Cited in page 33)

[Che10] Songting Chen. Cheetah: A high performance, custom data warehouse on top of mapreduce. *PVLDB*, 3(2):1459–1468, 2010. (Cited in page 58)

[CHKZ03] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5):1338–1355, 2003. (Cited in page 85)

[Chu69] Wesley W. Chu. Optimal file allocation in a multiple computer system. *IEEE Trans. Computers*, 18(10):885–889, 1969. (Cited in page 23), (Cited in page 24), (Cited in page 26)

[CILP12] Mariano P. Consens, Kleoni Ioannidou, Jeff LeFevre, and Neoklis Polyzotis. Divergent physical design tuning for replicated databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Scottsdale, AZ, USA, May 20-24*, pages 49–60, 2012. (Cited in page 42), (Cited in page 44), (Cited in page 52)

[CK85] George P. Copeland and Setrag Khoshafian. A decomposition storage model. In *Proceedings of ACM SIGMOD International Conference on Management of Data, Austin, Texas, USA, May 28-31*, pages 268–279, 1985. (Cited in page 82)

[CM93] Mariano P. Consens and Alberto O. Mendelzon. Hy+: A hygraph-based query and visualization system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28*, pages 511–516, 1993. (Cited in page 64)

[CMP82] Stefano Ceri, Giancarlo Martella, and Giuseppe Pelagatti. Optimal file allocation in a computer network: a solution method based on the knapsack problem. *Comput. Networks*, 6(5):345–357, 1982. (Cited in page 109), (Cited in page 172)

[CNP82] Stefano Ceri, Mauro Negri, and Giuseppe Pelagatti. Horizontal data partitioning in database design. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Orlando, Florida, USA, June 2-4*, pages 128–136, 1982. (Cited in page 8), (Cited in page 25), (Cited in page 26), (Cited in page 28), (Cited in page 37), (Cited in page 52)

[CNW83] Stefano Ceri, Shamkant B. Navathe, and Gio Wiederhold. Distribution design of logical database schemas. *IEEE Trans. Software Eng.*, 9(4):487–504, 1983. (Cited in page 28), (Cited in page 41), (Cited in page 52)

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970. (Cited in page 33)

[CPW89] Stefano Ceri, Barbara Pernici, and Gio Wiederhold. Optimization problems and solution methods in the design of data distribution. *Inf. Syst.*, 14(3):261–272, 1989. (Cited in page 36), (Cited in page 51)

[CWZ94] Jonathan E. Cook, Alexander L. Wolf, and Benjamin G. Zorn. Partition selection policies in object database garbage collection. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA, May 24-27*, pages 371–382, 1994. (Cited in page 10), (Cited in page 45), (Cited in page 52)

[CY87] Douglas W. Cornell and Philip S. Yu. A vertical partitioning algorithm for relational databases. In *Proceedings of the Third International Conference on Data Engineering, February 3-5, 1987, Los Angeles, California, USA*, pages 30–35, 1987. (Cited in page 25), (Cited in page 26), (Cited in page 36), (Cited in page 51)

[CZC15] Kaiji Chen, Yongluan Zhou, and Yu Cao. Online data partitioning in distributed database systems. In *Proceedings of the 18th International Conference on Extending Database Technology, EDBT, Brussels, Belgium, March 23-27*, pages 1–12, 2015. (Cited in page 47), (Cited in page 50), (Cited in page 52)

[CZJM10] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 3(1):48–57, 2010. (Cited in page 42), (Cited in page 44), (Cited in page 52)

[DAB06] Jun Du, Reda Alhajj, and Ken Barker. Genetic algorithms based approach to database vertical partition. *J. Intell. Inf. Syst.*, 26(2):167–183, 2006. (Cited in page 36), (Cited in page 37)

[Dat15] DataStax. TITAN: Distributed graph database. `http://titan.thinkaurelius.com`, 2015. (Cited in page 78)

[DB14] R.V. Guha Dan Brickley. RDF schema 1.1. `https://www.w3.org/TR/rdf-schema/`, 2014. (Cited in page 1), (Cited in page 81)

[DCL18] Ali Davoudian, Liu Chen, and Mengchi Liu. A survey on NoSQL stores. *ACM Comput. Surv.*, 51(2):40:1–40:43, 2018. (Cited in page 56), (Cited in page 66), (Cited in page 75)

[DG92] David J. DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992. (Cited in page 31), (Cited in page 38), (Cited in page 52)

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8*, pages 137–150, 2004. (Cited in page 53), (Cited in page 165)

[DGG+86] David J. DeWitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, and M. Muralikrishna. GAMMA - A high performance dataflow database machine. In *Proceedings of the Twelfth International Conference on Very Large Data Bases VLDB, August 25-28, Kyoto, Japan.*, pages 228–237, 1986. (Cited in page 24), (Cited in page 26), (Cited in page 27), (Cited in page 38)

[DGH+14] Xin Dong, Evgeniy Gabrilovich, Geremy Heitz, Wilko Horn, Ni Lao, Kevin Murphy, Thomas Strohmann, Shaohua Sun, and Wei Zhang. Knowledge vault: a web-scale approach to probabilistic knowledge fusion. In *The 20th ACM International Conference on Knowledge Discovery and Data Mining SIGKDD, New York, NY, USA - August 24 - 27*, pages 601–610, 2014. (Cited in page 4)

[DH72] William E Donath and Alan J Hoffman. Algorithms for partitioning of graphs and computer logic based on eigenvectors of connection matrices. *IBM Technical Disclosure Bulletin*, 15(3):938–944, 1972. (Cited in page 73)

[DLL+17] Liming Dong, Weidong Liu, Renchuan Li, Tiejun Zhang, and Weiguo Zhao. Replica-aware partitioning design in parallel database systems. In *Proceedings of Euro-Par: Parallel Processing - 23rd International Conference on Parallel and Distributed Computing, Santiago de Compostela, Spain, August 28 - September 1*, pages 303–316, 2017. (Cited in page 43), (Cited in page 44), (Cited in page 52)

[DPP+18] Gabriel Campero Durand, Marcus Pinnecke, Rufat Piriyev, Mahmoud Mohsen, David Broneske, Gunter Saake, Maya S. Sekeran, Fabián Rodriguez, and Laxmi

Balami. Gridformation: Towards self-driven online data partitioning using reinforcement learning. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD, Houston, TX, USA, June 10*, pages 1:1–1:7, 2018. (Cited in page 50), (Cited in page 51), (Cited in page 52)

[DPP+19] Gabriel Campero Durand, Rufat Piriyev, Marcus Pinnecke, David Broneske, Balasubramanian Gurumurthy, and Gunter Saake. Automated vertical partitioning with deep reinforcement learning. In *Proceedings of New Trends in Databases and Information Systems, ADBIS, Bled, Slovenia, September 8-11*, pages 126–134, 2019. (Cited in page 36), (Cited in page 37), (Cited in page 51)

[DPSW00] Ralf Diekmann, Robert Preis, Frank Schlimbach, and Chris Walshaw. Shape-optimized mesh partitioning and load balancing for parallel adaptive FEM. *Parallel Comput.*, 26(12):1555–1581, 2000. (Cited in page 74)

[DQ12] Jens Dittrich and Jorge-Arnulfo Quiané-Ruiz. Efficient big data processing in hadoop mapreduce. *PVLDB*, 5(12):2014–2015, 2012. (Cited in page 58)

[DQJ+10] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *PVLDB*, 3(1):518–529, 2010. (Cited in page 58)

[DS82] David Hung-Chang Du and John S. Sobolewski. Disk allocation for cartesian product files on multiple-disk systems. *ACM Trans. Database Syst.*, 7(1):82–101, 1982. (Cited in page 39), (Cited in page 40), (Cited in page 52)

[DWNY12] Jin-Hang Du, Haofen Wang, Yuan Ni, and Yong Yu. HadoopRDF: A scalable semantic data analytical engine. In *8th International Conference on Intelligent Computing Theories and Applications (ICIC)*, pages 633–641, 2012. (Cited in page 83), (Cited in page 89), (Cited in page 90), (Cited in page 146)

[DZC17] Dong Dai, Wei Zhang, and Yong Chen. IOGP: an incremental online graph partitioning algorithm for distributed graph databases. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, HPDC, Washington, DC, USA, June 26-30*, pages 219–230, 2017. (Cited in page 75), (Cited in page 76)

[ECS+08] George Eadon, Eugene Inseok Chong, Shrikanth Shankar, Ananth Raghavan, Jagannathan Srinivasan, and Souripriya Das. Supporting table partitioning by reference in Oracle. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD, Vancouver, BC, Canada, June 10-12*, pages 1111–1122, 2008. (Cited in page 41), (Cited in page 52)

[EM09] Orri Erling and Ivan Mikhailov. Virtuoso: RDF support in a native RDBMS. In *Semantic Web Information Management - A Model-Based Perspective*, pages 501–519. 2009. (Cited in page 5), (Cited in page 83), (Cited in page 86), (Cited in page 107), (Cited in page 128), (Cited in page 134), (Cited in page 172)

[ES76] Mark J. Eisner and Dennis G. Severance. Mathematical techniques for efficient record segmentation in large shared databases. *J. ACM*, 23(4):619–635, 1976. (Cited in page 22), (Cited in page 23), (Cited in page 25), (Cited in page 26), (Cited in page 31), (Cited in page 35), (Cited in page 37)

[Esw74] Kapali P. Eswaran. Placement of records in a file and file allocation in a computer. In *IFIP Congress*, pages 304–307, 1974. (Cited in page 23), (Cited in page 24), (Cited in page 26), (Cited in page 109), (Cited in page 172)

[ESW78] Robert S. Epstein, Michael Stonebraker, and Eugene Wong. Distributed query processing in a relational data base system. In *Proceedings of the 1978 ACM SIGMOD International Conference on Management of Data, Austin, Texas, USA, May 31 - June 2, 1978*, pages 169–180, 1978. (Cited in page 25), (Cited in page 26)

[ETÖ+11] Mohamed Y. Eltabakh, Yuanyuan Tian, Fatma Özcan, Rainer Gemulla, Aljoscha Krettek, and John McPherson. Cohadoop: Flexible data placement and its exploitation in hadoop. *PVLDB*, 4(9):575–585, 2011. (Cited in page 58)

[FB93] Christos Faloutsos and Pravin Bhagwat. Declustering using fractals. In *Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems (PDIS), Issues, Architectures, and Algorithms, San Diego, CA, USA, January 20-23*, pages 18–25, 1993. (Cited in page 39), (Cited in page 40), (Cited in page 52)

[FGG+18] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the International Conference on Management of Data, SIGMOD, Houston, TX, USA, June 10-15*, pages 1433–1445, 2018. (Cited in page 67)

[FKL03] Chi-Wai Fung, Kamalakar Karlapalem, and Qing Li. Cost-driven vertical class partitioning for methods in object oriented databases. *VLDB J.*, 12(3):187–210, 2003. (Cited in page 10), (Cited in page 45), (Cited in page 51)

[FLC86] M. T. Fang, Richard C. T. Lee, and Chin-Chen Chang. The idea of de-clustering and its applications. In *Proceedings of the Twelfth International Conference on Very Large Data Bases VLDB, August 25-28, Kyoto, Japan.*, pages 181–188, 1986. (Cited in page 24), (Cited in page 26), (Cited in page 27)

[FLT+20] Wenfei Fan, Muyang Liu, Chao Tian, Ruiqi Xu, and Jingren Zhou. Incrementalization of graph partitioning algorithms. *Proc. VLDB Endow.*, 13(8):1261–1274, 2020. (Cited in page 75), (Cited in page 77)

[FM91] Christos Faloutsos and Dimitris N. Metaxas. Disk allocation methods using error correcting codes. *IEEE Trans. Computers*, 40(8):907–914, 1991. (Cited in page 39), (Cited in page 40), (Cited in page 52)

[FM17] Hugo Firth and Paolo Missier. TAPER: query-aware, partition-enhancement for large, heterogenous graphs. *Distributed and Parallel Databases*, 35(2):85–115, 2017. (Cited in page 75), (Cited in page 76)

[FMS15] Ilir Fetai, Damian Murezzan, and Heiko Schuldt. Workload-driven adaptive data partitioning and distribution - the cumulus approach. In *IEEE International Conference on Big Data, Big Data, Santa Clara, CA, USA, October 29 - November 1*, pages 1688–1697, 2015. (Cited in page 50), (Cited in page 52)

[For83] Surveyors' Forum. comments on "comparative models of the file assignment problem". *ACM Comput. Surv.*, 15(1):81–82, 1983. (Cited in page 26)

[Fou19] The Apache Software Foundation. Apache Giraph. `https://giraph.apache.org/`, 2019. (Cited in page 75), (Cited in page 78), (Cited in page 79)

[GBF11] Philippe Galinier, Zied Boujbel, and Michael Coutinho Fernandes. An efficient memetic algorithm for the graph partitioning problem. *Annals OR*, 191(1):1–22, 2011. (Cited in page 74)

[GD90] Shahram Ghandeharizadeh and David J. DeWitt. Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database machines. In *Proceedings of the 16th International Conference on Very Large Data Bases, August 13-16, , Brisbane, Queensland, Australia*, pages 481–492, 1990. (Cited in page 39)

[GD94] Shahram Ghandeharizadeh and David J. DeWitt. MAGIC: A multiattribute declustering mechanism for multiprocessor database machines. *IEEE Trans. Parallel Distrib. Syst.*, 5(5):509–524, 1994. (Cited in page 39), (Cited in page 52)

[GDQ92] Shahram Ghandeharizadeh, David J. DeWitt, and Waheed Qureshi. A performance analysis of alternative multi-attribute declustering strategies. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 2-5*, pages 29–38, 1992. (Cited in page 22), (Cited in page 40)

[GGGK03] Shahram Ghandeharizadeh, Shan Gao, Chris Gahagan, and Russ Krauss. High performance parallel database management systems. In *Handbook on Data Management in Information Systems*. Springer, Berlin, Heidelberg, 2003. (Cited in page 39)

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles SOSP, Bolton Landing, NY, USA, October 19-22*, pages 29–43, 2003. (Cited in page 53), (Cited in page 165)

[GHK92] Sumit Ganguly, Waqar Hasan, and Ravi Krishnamurthy. Query optimization for parallel execution. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 2-5*, pages 9–18, 1992. (Cited in page 32)

[GHS14] Luis Galárraga, Katja Hose, and Ralf Schenkel. Partout: a distributed engine for efficient RDF processing. In *WWW*, pages 267–268, 2014. (Cited in page 90), (Cited in page 156)

[GJGL16] Damien Graux, Louis Jachiet, Pierre Genevès, and Nabil Layaïda. SPARQLGX: efficient distributed evaluation of SPARQL with Apache Spark. In *International Conference on Semantic Web (ISWC)*, pages 80–87, 2016. (Cited in page 90)

[GJS76] M. R. Garey, David S. Johnson, and Larry J. Stockmeyer. Some simplified np-complete graph problems. *Theor. Comput. Sci.*, 1(3):237–267, 1976. (Cited in page 72)

[GKM+15] François Goasdoué, Zoi Kaoudi, Ioana Manolescu, Jorge-Arnulfo Quiané-Ruiz, and Stamatis Zampetakis. CliqueSquare: Flat plans for massively parallel RDF queries. In *ICDE*, pages 771–782, 2015. (Cited in page 7), (Cited in page 90), (Cited in page 128), (Cited in page 140)

[GL81] Alan George and Joseph W Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice Hall Professional Technical Reference, 1981. (Cited in page 73)

[GLL+20] Jin-Tao Gao, Wenjie Liu, Zhanhuai Li, Jian Zhang, and Li Shen. A general fragments allocation method for join query in distributed database. *Inf. Sci.*, 512:1249–1263, 2020. (Cited in page 43), (Cited in page 44), (Cited in page 52)

[GMB19] Jorge Galicia, Amin Mesmoudi, and Ladjel Bellatreche. RDFPartSuite: Bridging physical and logical RDF partitioning. In *Proceedings of the 21st International Conference on Big Data Analytics and Knowledge Discovery - DaWaK, Linz, Austria*, pages 136–150, 2019. (Cited in page 133), (Cited in page 136), (Cited in page 175)

[GMBO19] Jorge Galicia, Amin Mesmoudi, Ladjel Bellatreche, and Carlos Ordonez. Reverse partitioning for SPARQL queries: Principles and performance analysis. In *Proceedings of the 30th International Conference in Database and Expert Systems Applications - DEXA, Linz, Austria, August 26-29*, pages 174–183, 2019. (Cited in page 131)

[GMR98] Matteo Golfarelli, Dario Maio, and Stefano Rizzi. The dimensional fact model: A conceptual model for data warehouses. *Int. J. Cooperative Inf. Syst.*, 7(2-3):215–247, 1998. (Cited in page 34)

[GO18] LLC Gurobi Optimization. Gurobi optimizer reference manual, 2018. (Cited in page 137), (Cited in page 138)

[GPdBG94] Marc Gyssens, Jan Paredaens, Jan Van den Bussche, and Dirk Van Gucht. A graph-oriented object database model. *IEEE Trans. Knowl. Data Eng.*, 6(4):572–586, 1994. (Cited in page 64)

[GPH05] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Semant.*, 3(2-3):158–182, 2005. (Cited in page 128)

[GPST94] Alejandro Gutiérrez, Philippe Pucheral, Hermann Steffen, and Jean-Marc Thévenin. Database graph views: A practical model to manage persistent graphs. In *Proceedings of 20th International Conference on Very Large Data Bases VLDB, September 12-15, Santiago de Chile, Chile*, pages 391–402, 1994. (Cited in page 64)

[Gra94] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994. (Cited in page 38), (Cited in page 122), (Cited in page 127), (Cited in page 174)

[Gro12] W3C OWL Working Group. OWL 2 web ontology language. `https://www.w3.org/TR/2012/REC-owl2-overview-20121211/`, 2012. (Cited in page 2), (Cited in page 81)

[GSMT14] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. TriAD: a distributed shared-nothing RDF engine based on asynchronous message passing. In *International Conference on Management of Data, SIGMOD, Snowbird, UT, USA, June 22-27*, pages 289–300, 2014. (Cited in page 86), (Cited in page 90)

[Güt94] Ralf Hartmut Güting. Graphdb: Modeling and querying graphs in databases. In *Proceedings of 20th International Conference on Very Large Data Bases VLDB, September 12-15, Santiago de Chile, Chile*, pages 297–308, 1994. (Cited in page 64), (Cited in page 77)

[GV90] Georges Gardarin and Patrick Valduriez. ESQL: an extended SQL with object and deductive capabilities. In *Proceedings of the International Conference on Database and Expert Systems Applications, Vienna, Austria, August 29-31*, pages 299–306, 1990. (Cited in page 34)

[GXD+14] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow

framework. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI, Broomfield, CO, USA, October 6-8*, pages 599–613, 2014. (Cited in page 75)

[GZQ+20] Qingyu Guo, Fuzhen Zhuang, Chuan Qin, Hengshu Zhu, Xing Xie, Hui Xiong, and Qing He. A survey on knowledge graph-based recommender systems. *CoRR*, abs/2003.00911, 2020. (Cited in page 3)

[HA16] Jiewen Huang and Daniel Abadi. LEOPARD: lightweight edge-oriented partitioning and replication for dynamic graphs. *Proc. VLDB Endow.*, 9(7):540–551, 2016. (Cited in page 75), (Cited in page 76)

[Had] Welcome to Apache Hadoop! `https://hadoop.apache.org/`. (Cited in page 53)

[HAR11] Jiewen Huang, Daniel J. Abadi, and Kun Ren. Scalable SPARQL querying of large RDF graphs. *PVLDB*, 4(11):1123–1134, 2011. (Cited in page 88), (Cited in page 89), (Cited in page 90)

[HBR19] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. Towards learning a partitioning advisor with deep reinforcement learning. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD, Amsterdam, The Netherlands, July 5*, pages 6:1–6:4, 2019. (Cited in page 47), (Cited in page 49), (Cited in page 51)

[HDA+14] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M. Tamer Özsu, Xingfang Wang, and Tianqi Jin. An experimental comparison of pregel-like graph processing systems. *Proc. VLDB Endow.*, 7(12):1047–1058, 2014. (Cited in page 79)

[He16] Qi He. Building the linkedin knowledge graph. `https://engineering.linkedin.com/blog/2016/10/building-the-linkedin-knowledge-graph`, 2016. (Cited in page 3)

[HK00] Bruce Hendrickson and Tamara G. Kolda. Graph partitioning models for parallel computing. *Parallel Comput.*, 26(12):1519–1534, 2000. (Cited in page 72)

[HL90] Kien A. Hua and Chiang Lee. An adaptive data placement scheme for parallel database computer systems. In *Proceedings of the 16th International Conference on Very Large Data Bases, August 13-16, Brisbane, Queensland, Australia*, pages 493–506, 1990. (Cited in page 39), (Cited in page 40), (Cited in page 52)

[HN79] Michael Hammer and Bahram Niamir. A heuristic approach to attribute partitioning. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1.*, pages 93–101, 1979. (Cited in page 35), (Cited in page 37), (Cited in page 51)

[Hof76] Jeffrey A. Hoffer. An integer programming formulation of computer data base design problems. *Inf. Sci.*, 11(1):29–48, 1976. (Cited in page 23), (Cited in page 25), (Cited in page 26), (Cited in page 35), (Cited in page 37), (Cited in page 51)

[HP03] Richard A. Hankins and Jignesh M. Patel. Data morphing: An adaptive, cache-conscious storage technique. In *Proceedings of 29th International Conference on Very Large Data Bases, VLDB, Berlin, Germany, September 9-12*, pages 417–428, 2003. (Cited in page 36), (Cited in page 37)

[HRN+15] Mohammad Hammoud, Dania Abed Rabbou, Reza Nouri, Seyed-Mehdi-Reza Beheshti, and Sherif Sakr. DREAM: distributed RDF engine with adaptive query planner and minimal communication. *PVLDB*, 8(6):654–665, 2015. (Cited in page 90)

[HS75] Jeffrey A. Hoffer and Dennis G. Severance. The use of cluster analysis in physical data base design. In *Proceedings of the International Conference on Very Large Data Bases, September 22-24, Framingham, Massachusetts, USA.*, pages 69–86, 1975. (Cited in page 21), (Cited in page 23), (Cited in page 24), (Cited in page 25), (Cited in page 26), (Cited in page 35), (Cited in page 37), (Cited in page 162)

[HS13] Katja Hose and Ralf Schenkel. WARP: workload-aware replication and partitioning for RDF. In *Workshops Proceedings of the 29th IEEE ICDE*, pages 1–6, 2013. (Cited in page 90), (Cited in page 156)

[HVL14] Kai Herrmann, Hannes Voigt, and Wolfgang Lehner. Cinderella - adaptive online partitioning of irregularly structured data. In *Workshops Proceedings of the 30th International Conference on Data Engineering Workshops, ICDE 2014, Chicago, IL, USA, March 31 - April 4, 2014*, pages 284–291, 2014. (Cited in page 84), (Cited in page 98)

[iIaC05] W3C Office in Italy at C.N.R. Representing knowledge in the semantic web. `http://www.w3c.it/talks/2005/openCulture/slide1-0.html`, 2005. (Cited in page 1)

[Ior10] Borislav Iordanov. Hypergraphdb: A generalized graph database. In *International Workshops: Web-Age Information Management - WAIM IWGD, XMLDM, WCMT, Jiuzhaigou Valley, China, July 15-17*, pages 25–36, 2010. (Cited in page 77), (Cited in page 78)

[JD11] Alekh Jindal and Jens Dittrich. Relax and let the database do the partitioning online. In *5th International Workshop Enabling Real-Time Business Intelligence - BIRTE, 37th International Conference on Very Large Databases, VLDB, Seattle, WA, USA, September 2*, pages 65–80, 2011. (Cited in page 49), (Cited in page 50), (Cited in page 52)

[JG77] James B. Rothnie Jr. and Nathan Goodman. An overview of the preliminary design of SDD-1: A system for distributed databases. In *Berkeley Workshop*, pages 39–57, 1977. (Cited in page 25), (Cited in page 26)

[JL74] James B. Rothnie Jr. and Tomas Lozano. Attribute based file organization in a paged memory environment. *Commun. ACM*, 17(2):63–69, 1974. (Cited in page 26)

[JPPD13] Alekh Jindal, Endre Palatinus, Vladimir Pavlov, and Jens Dittrich. A comparison of knives for bread slicing. *Proc. VLDB Endow.*, 6(6):361–372, 2013. (Cited in page 37)

[JQD11] Alekh Jindal, Jorge-Arnulfo Quiané-Ruiz, and Jens Dittrich. Trojan data layouts: right shoes for a running elephant. In *ACM Symposium on Cloud Computing in conjunction with SOSP, SOCC, Cascais, Portugal, October 26-28*, page 21, 2011. (Cited in page 58)

[JSL20] Daniel Janke, Steffen Staab, and Martin Leinberger. Data placement strategies that speed-up distributed graph query processing. In *Proceedings of The International Workshop on Semantic Big Data, SBD@SIGMOD , Portland, Oregon, USA, June 19*, pages 2:1–2:6, 2020. (Cited in page 7), (Cited in page 153), (Cited in page 160), (Cited in page 176)

[JST17] Daniel Janke, Steffen Staab, and Matthias Thimm. Koral: A glass box profiling system for individual components of distributed RDF stores. In *2nd International Workshop on Benchmarking Linked Data and NLIWoD3: Natural Language Interfaces for the Web*, 2017. (Cited in page 89)

[JSW72] William T. McCormick Jr., Paul J. Schweitzer, and Thomas W. White. Problem decomposition and data reorganization by a clustering technique. *Operations Research*, 20(5):993–1009, 1972. (Cited in page 25)

[KG14] Aapo Kyrola and Carlos Guestrin. Graphchi-db: Simple design for a scalable graph database system - on just a PC. *CoRR*, abs/1403.0701, 2014. (Cited in page 77), (Cited in page 78)

[KK98a] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Scientific Computing*, 20(1):359–392, 1998. (Cited in page 50), (Cited in page 66), (Cited in page 73), (Cited in page 74), (Cited in page 88), (Cited in page 111), (Cited in page 115), (Cited in page 138), (Cited in page 173)

[KK98b] George Karypis and Vipin Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *J. Parallel Distributed Comput.*, 48(1):71–95, 1998. (Cited in page 75)

[KKTC12] Vaibhav Khadilkar, Murat Kantarcioglu, Bhavani M. Thuraisingham, and Paolo Castagna. Jena-HBase: A distributed, scalable and effcient RDF triple store. In *Proceedings of the ISWC 2012 Posters & Demonstrations Track, Boston, USA, November 11-15, 2012*, 2012. (Cited in page 90)

[KL70] Brian W. Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *Bell Syst. Tech. J.*, 49(2):291–307, 1970. (Cited in page 73)

[KL95] Kamalakar Karlapalem and Qing Li. Partitioning schemes for object oriented databases. In *Proceedings RIDE-DOM '95, Fifth International Workshop on Research Issues in Data Engineering - Distributed Object Management, Taipei, Taiwan, March 6-7*, pages 42–49, 1995. (Cited in page 10), (Cited in page 45), (Cited in page 52)

[KL00] Kamalakar Karlapalem and Qing Li. A framework for class partitioning in object-oriented databases. *Distributed and Parallel Databases*, 8(3):333–366, 2000. (Cited in page 10), (Cited in page 45)

[KM15] Zoi Kaoudi and Ioana Manolescu. RDF in the clouds: a survey. *VLDB J.*, 24(1):67–91, 2015. (Cited in page 80), (Cited in page 81), (Cited in page 86), (Cited in page 155)

[KMG+20] Abdallah Khelil, Amin Mesmoudi, Jorge Galicia, Ladjel Bellatreche, Mohand-Saïd Hacid, and Emmanuel Coquery. Combining graph exploration and fragmentation for scalable RDF query processing. *Information Systems Frontiers (2020)*, 2020. (Cited in page 14), (Cited in page 88), (Cited in page 121), (Cited in page 122), (Cited in page 134), (Cited in page 138), (Cited in page 140), (Cited in page 146), (Cited in page 155), (Cited in page 161), (Cited in page 174)

[KP88] Myoung-Ho Kim and Sakti Pramanik. Optimal file distribution for partial match retrieval. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 1-3*, pages 173–182, 1988. (Cited in page 39), (Cited in page 40), (Cited in page 52)

[KS95] Ralph Kimball and Kevin Strehlo. Why decision support fails and how to fix it. *SIGMOD Rec.*, 24(3):92–97, 1995. (Cited in page 34)

[Kun87] H. S. Kunii. DBMS with graph data model for knowledge handling. In *Proceedings of the Fall Joint Computer Conference on Exploring technology: today and tomorrow*, pages 138–142, 1987. (Cited in page 63)

[KV84] Gabriel M. Kuper and Moshe Y. Vardi. A new approach to database logic. In *Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, April 2-4, Waterloo, Ontario, Canada*, pages 86–96, 1984. (Cited in page 63)

[LAC+11] Yuting Lin, Divyakant Agrawal, Chun Chen, Beng Chin Ooi, and Sai Wu. Llama: leveraging columnar storage for scalable join processing in the mapreduce framework. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Athens, Greece, June 12-16*, pages 961–972, 2011. (Cited in page 58)

[LAP+12] Miguel Liroz-Gistau, Reza Akbarinia, Esther Pacitti, Fábio Porto, and Patrick Valduriez. Dynamic workload-based partitioning for large-scale databases. In *Proceedings of the 23rd International ConferenceDatabase and Expert Systems Applications - DEXA, Vienna, Austria, September 3-6*, pages 183–190, 2012. (Cited in page 49)

[LAP+13] Miguel Liroz-Gistau, Reza Akbarinia, Esther Pacitti, Fábio Porto, and Patrick Valduriez. Dynamic workload-based partitioning algorithms for continuously growing databases. *Trans. Large-Scale Data- and Knowledge-Centered Systems*, 12:105–128, 2013. (Cited in page 49), (Cited in page 50), (Cited in page 52)

[LG12] Liangzhe Li and Le Gruenwald. Autonomous database partitioning using data mining on single computers and cluster computers. In *16th International Database Engineering & Applications Symposium, IDEAS, Prague, Czech Republic, August 8-10*, pages 32–41, 2012. (Cited in page 36), (Cited in page 37)

[LGK+12] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, 2012. (Cited in page 78), (Cited in page 79)

[LGS+79] Vincent Y. Lum, Sakti P. Ghosh, Mario Schkolnick, Robert W. Taylor, D. Jefferson, Stanley Y. W. Su, James P. Fry, Toby J. Teorey, B. Yao, D. S. Rund, B. Kahn, Shamkant B. Navathe, D. Smith, L. Aguilar, W. J. Barr, and P. E. Jones. 1978 new orleans data base design workshop report. In *Proceedings of the Fifth International Conference on Very Large Data Bases, October 3-5, Rio de Janeiro, Brazil.*, pages 328–339, 1979. (Cited in page 21), (Cited in page 25), (Cited in page 26)

[LIJ+15] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. Dbpedia - A large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web*, 6(2):167–195, 2015. (Cited in page 3)

[LL13] Kisung Lee and Ling Liu. Scaling queries over big RDF graphs with semantic hash partitioning. *PVLDB*, 6(14):1894–1905, 2013. (Cited in page 89), (Cited in page 90)

[LLD79] W. C. Lin, Richard C. T. Lee, and David Hung-Chang Du. Common properties of some multiattribute file systems. *IEEE Trans. Software Eng.*, 5(2):160–174, 1979. (Cited in page 26), (Cited in page 27)

[LMV10] Alexandre A. B. Lima, Marta Mattoso, and Patrick Valduriez. Adaptive virtual partitioning for OLAP query processing in a database cluster. *JIDM*, 1(1):75–88, 2010. (Cited in page 46), (Cited in page 49), (Cited in page 50), (Cited in page 52)

[LNP16] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. Parallel graph analytics. *Commun. ACM*, 59(5):78–87, 2016. (Cited in page 71), (Cited in page 167)

[LOZ93] Xuemin Lin, Maria E. Orlowska, and Yanchun Zhang. A graph based cluster approach for vertical partitioning in database design. *Data Knowl. Eng.*, 11(2):151–169, 1993. (Cited in page 31)

[LRV88] Christophe Lécluse, Philippe Richard, and Fernando Vélez. O2, an object-oriented data model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 1-3*, pages 424–433, 1988. (Cited in page 64)

[Lum70] Vincent Y. Lum. Multi-attribute retrieval with combined indexes. *Commun. ACM*, 13(11):660–665, 1970. (Cited in page 83)

[LY77] J. H. Liou and S. Bing Yao. Multi-dimensional clustering for data base organizations. *Inf. Syst.*, 2(4):187–198, 1977. (Cited in page 26)

[LY80] K. Lam and Clement T. Yu. An approximation algorithm for a file-allocation problem in a hierarchical distributed system. In *Proceedings of the 1980 ACM SIGMOD International Conference on Management of Data, Santa Monica, California, USA, May 14-16*, pages 125–132, 1980. (Cited in page 109), (Cited in page 172)

[LZ93] Xuemin Lin and Yanchun Zhang. A new graphical method of vertical partitioning in database design. In *Advances in Database Research - Proceedings of the 4th Australian Database Conference, ADC Griffith University, Brisbane, Queensland, Australia, February 1-2*, pages 131–144, 1993. (Cited in page 36), (Cited in page 37), (Cited in page 51)

[MAB+10] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD, Indianapolis, Indiana, USA, June 6-10*, pages 135–146, 2010. (Cited in page 78), (Cited in page 79)

[MASB18] Shikha Mehta, Parul Agarwal, Prakhar Shrivastava, and Jharna Barlawala. Differential bond energy algorithm for optimal vertical fragmentation of distributed databases. *Journal of King Saud University - Computer and Information Sciences*, 2018. (Cited in page 35)

[MD97] Manish Mehta and David J. DeWitt. Data placement in shared-nothing parallel database systems. *VLDB J.*, 6(1):53–72, 1997. (Cited in page 40)

[MDA+10] Cristina Maier, Debabrata Dash, Ioannis Alagiannis, Anastasia Ailamaki, and Thomas Heinis. PARINDA: an interactive physical designer for PostgreSQL. In *Proceedings of the 13th International Conference on Extending Database Technology EDBT, Lausanne, Switzerland, March 22-26*, pages 701–704, 2010. (Cited in page 9), (Cited in page 47), (Cited in page 49), (Cited in page 51)

[MLLS17] Claudio Martella, Dionysios Logothetis, Andreas Loukas, and Georgos Siganos. Spinner: Scalable graph partitioning in the cloud. In *33rd IEEE International Conference on Data Engineering, ICDE, San Diego, CA, USA, April 19-22*, pages 1083–1094, 2017. (Cited in page 75)

[MMG+07] Norbert Martínez-Bazan, Victor Muntés-Mulero, Sergio Gómez-Villamor, Jordi Nin, Mario-A. Sánchez-Martínez, and Josep-Lluís Larriba-Pey. Dex: high-performance exploration on large graphs for information retrieval. In *Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management, CIKM, Lisbon, Portugal, November 6-10*, pages 573–582, 2007. (Cited in page 66), (Cited in page 77), (Cited in page 78)

[MS77] Salvatore T. March and Dennis G. Severance. The determination of efficient record segmentations and blocking factors for shared data files. *ACM Trans. Database Syst.*, 2(3):279–296, 1977. (Cited in page 25)

[MS78] Salvatore T. March and Dennis G. Severance. A mathematical modeling approach to the automatic selection of database designs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Austin, Texas, USA, May 31 - June 2*, pages 52–65, 1978. (Cited in page 23), (Cited in page 25), (Cited in page 26)

[MS94] Mukesh K. Mohania and Nandlal L. Sarda. Some issues in design of distributed deductive databases. In *Proceedings of 20th International Conference on Very Large Data Bases VLDB, September 12-15, Santiago de Chile, Chile*, pages 60–71, 1994. (Cited in page 45), (Cited in page 52)

[MS98] Bongki Moon and Joel H. Saltz. Scalability analysis of declustering methods for multidimensional range queries. *IEEE Trans. Knowl. Data Eng.*, 10(2):310–327, 1998. (Cited in page 22), (Cited in page 27), (Cited in page 40)

[NB11] Rimma V. Nehme and Nicolas Bruno. Automated partitioning design in parallel database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Athens, Greece, June 12-16*, pages 1137–1148, 2011. (Cited in page 47), (Cited in page 48), (Cited in page 51), (Cited in page 157)

[NCWD84] Shamkant B. Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou. Vertical partitioning algorithms for database design. *ACM Trans. Database Syst.*, 9(4):680–710, 1984. (Cited in page 8), (Cited in page 25), (Cited in page 26), (Cited in page 36), (Cited in page 37), (Cited in page 51)

[NH94] Kathleen Neumann and Lawrence J. Henschen. Partitioning algorithms for a distributed deductive databases. In *Proceedings of the ACM 22nd Annual Computer Science Conference on Scaling up: Meeting the Challenge of Complexity in Real-World Computing Applications, CSC '94, Phoenix, Arizona, USA, March 8-10*, pages 288–295, 1994. (Cited in page 21), (Cited in page 45), (Cited in page 52), (Cited in page 162)

[Nia78] Bahram Niamir. Attribute partitioning in a self-adaptive relational database system. *Massachusetts Institute of Technology, Technical Report 192*, 1978. (Cited in page 35), (Cited in page 37), (Cited in page 51)

[NKDC15] Daniel Nicoara, Shahin Kamali, Khuzaima Daudjee, and Lei Chen. Hermes: Dynamic partitioning for distributed social network graph databases. In *Proceedings of the 18th International Conference on Extending Database Technology, EDBT, Brussels, Belgium, March 23-27*, pages 25–36, 2015. (Cited in page 75), (Cited in page 76)

[NKH18] Yoon-Min Nam, Min-Soo Kim, and Donghyoung Han. A graph-based database partitioning method for parallel OLAP query processing. In *34th IEEE International Conference on Data Engineering, ICDE, Paris, France, April 16-19*, pages 1025–1036, 2018. (Cited in page 46), (Cited in page 52)

[NM11] Thomas Neumann and Guido Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *Proceedings of the 27th International Conference on Data Engineering, ICDE, April 11-16, Hannover, Germany*, pages 984–994, 2011. (Cited in page 11), (Cited in page 84), (Cited in page 85), (Cited in page 87), (Cited in page 98), (Cited in page 99), (Cited in page 134), (Cited in page 170), (Cited in page 171)

[NPM+15] Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. RDFox: A highly-scalable RDF store. In *14th International Semantic Web Conference- ISWC- , Bethlehem, PA, USA, October 11-15*, pages 3–20, 2015. (Cited in page 83)

[NR89] Shamkant B. Navathe and Minyoung Ra. Vertical partitioning for database design: A graphical algorithm. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Portland, Oregon, USA, May 31 - June 2*, pages 440–450, 1989. (Cited in page 8), (Cited in page 36), (Cited in page 37), (Cited in page 51)

[NU13] Joel Nishimura and Johan Ugander. Restreaming graph partitioning: simple versatile algorithms for advanced balancing. In *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD, Chicago, IL, USA, August 11-14*, pages 1106–1114, 2013. (Cited in page 73)

[NW08] Thomas Neumann and Gerhard Weikum. RDF-3X: a risc-style engine for RDF. *Proc. VLDB Endow.*, 1(1):647–659, 2008. (Cited in page 6), (Cited in page 83), (Cited in page 86), (Cited in page 107), (Cited in page 125), (Cited in page 128), (Cited in page 134), (Cited in page 160)

[NW10] Thomas Neumann and Gerhard Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, 2010. (Cited in page 88)

[OB19] Carlos Ordonez and Ladjel Bellatreche. Guest editorial-dawak 2018 special issue-trends in big data analytics. *DKE, (2019)*, 2019. (Cited in page 99)

[OBS99] Michael A. Olson, Keith Bostic, and Margo I. Seltzer. Berkeley DB. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference, June 6-11, Monterey, California, USA*, pages 183–191, 1999. (Cited in page 77)

[OO85] Esen A. Ozkarahan and Aris M. Ouksel. Dynamic and order preserving data partitioning for database machines. In *Proceedings of 11th International Conference on Very Large Data Bases, August 21-23, Stockholm, Sweden*, pages 358–368, 1985. (Cited in page 39), (Cited in page 40), (Cited in page 52)

[OOB18] Abdelkader Ouared, Yassine Ouhammou, and Ladjel Bellatreche. Qosmos: Qos metrics management tool suite. *Comput. Lang. Syst. Struct.*, 54:236–251, 2018. (Cited in page 9)

[Ord13] Carlos Ordonez. Can we analyze big data inside a dbms? In *Proceedings of the sixteenth international workshop on Data warehousing and OLAP, DOLAP*, pages 85–92, 2013. (Cited in page 13), (Cited in page 161)

[ÖV96] M. Tamer Özsu and Patrick Valduriez. Distributed and parallel database systems. *ACM Comput. Surv.*, 28(1):125–128, 1996. (Cited in page 24)

[ÖV11] M Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*. Springer Science & Business Media, 2011. (Cited in page 7), (Cited in page 8),

(Cited in page 25), (Cited in page 26), (Cited in page 28), (Cited in page 29), (Cited in page 35), (Cited in page 37), (Cited in page 38), (Cited in page 41), (Cited in page 87), (Cited in page 97), (Cited in page 103), (Cited in page 104), (Cited in page 155), (Cited in page 160), (Cited in page 170)

[Özs16] M. Tamer Özsu. A survey of RDF data management systems. *Frontiers Comput. Sci.*, 10(3):418–432, 2016. (Cited in page 80), (Cited in page 81), (Cited in page 82), (Cited in page 86), (Cited in page 122), (Cited in page 174)

[PA04] Stratos Papadomanolakis and Anastassia Ailamaki. AutoPart: Automating schema design for large scientific databases using data partitioning. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management SSDBM, 21-23 June, Santorini Island, Greece*, pages 383–392, 2004. (Cited in page 47), (Cited in page 49)

[PB94] John R Pilkington and Scott B Baden. Partitioning with space filling curves, technical report. 1994. (Cited in page 73)

[PCR15] Roshan Punnoose, Adina Crainiceanu, and David Rapp. SPARQL in the cloud using rya. *Information Systems*, 48:181–195, 2015. (Cited in page 90)

[PCZ12] Andrew Pavlo, Carlo Curino, and Stanley B. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24*, pages 61–72, 2012. (Cited in page 32), (Cited in page 42), (Cited in page 44), (Cited in page 52)

[PKT+13] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, Panagiotis Karras, and Nectarios Koziris. H2RDF+: high-performance distributed joins over large-scale RDF graphs. In *IEEE International Conference on Big Data*, pages 255–263, 2013. (Cited in page 89), (Cited in page 90)

[PPEB15] Minh-Duc Pham, Linnea Passing, Orri Erling, and Peter A. Boncz. Deriving an emergent relational schema from RDF data. In *Proceedings of the 24th International Conference on World Wide Web, WWW, Florence, Italy, May 18-22*, pages 864–874, 2015. (Cited in page 6), (Cited in page 84), (Cited in page 87), (Cited in page 98), (Cited in page 99), (Cited in page 100), (Cited in page 107), (Cited in page 170), (Cited in page 172)

[PPR+09] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Providence, Rhode Island, USA, June 29 - July 2*, pages 165–178, 2009. (Cited in page 53), (Cited in page 165)

[PPT95] Jan Paredaens, Peter Peelman, and Letizia Tanca. G-log: A graph-based query language. *IEEE Trans. Knowl. Data Eng.*, 7(3):436–453, 1995. (Cited in page 64)

[Pro19] ISO Graph Query Language Proponents. Gql standard. `https://www.gqlstandards.org/`, 2019. (Cited in page 68)

[PZÖ+16] Peng Peng, Lei Zou, M. Tamer Özsu, Lei Chen, and Dongyan Zhao. Processing SPARQL queries over distributed RDF graphs. *VLDB Journal*, 25(2):243–268, 2016. (Cited in page 13), (Cited in page 14), (Cited in page 88), (Cited in page 90), (Cited in page 133), (Cited in page 140), (Cited in page 146), (Cited in page 155), (Cited in page 161), (Cited in page 175)

[QKD13] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. SWORD: scalable workload-aware data placement for transactional workloads. In *Proceedings of EDBT/ICDT Conferences, Genoa, Italy, March 18-22*, pages 430–441, 2013. (Cited in page 44)

[RC14] Markus Lanthaler Richard Cyganiak, David Wood. RDF 1.1 concepts and abstract syntax. `https://www.w3.org/TR/rdf11-concepts/`, 2014. (Cited in page 1), (Cited in page 80), (Cited in page 159), (Cited in page 167)

[RDS02] Ravishankar Ramamurthy, David J. DeWitt, and Qi Su. A case for fractured mirrors. In *Proceedings of 28th International Conference on Very Large Data Bases VLDB, August 20-23, Hong Kong, China*, pages 430–441, 2002. (Cited in page 48), (Cited in page 51), (Cited in page 108), (Cited in page 172)

[Rei82] Raymond Reiter. Towards a logical reconstruction of relational database theory. In *On Conceptual Modelling, Perspectives from Artificial Intelligence, Databases, and Programming Languages, Book resulting from the Intervale Workshop 1982*, pages 191–233, 1982. (Cited in page 34)

[RG00] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*. McGraw Hill, 2000. (Cited in page 25), (Cited in page 28)

[RG06] Boris Rozenberg and Ehud Gudes. Association rules mining in vertically partitioned databases. *Data Knowl. Eng.*, 59(2):378–396, 2006. (Cited in page 36), (Cited in page 37), (Cited in page 51)

[RHAF15] Oscar Romero, Victor Herrero, Alberto Abelló, and Jaume Ferrarons. Tuning small analytics on big data: Data partitioning and secondary indexes in the hadoop ecosystem. *Inf. Syst.*, 54:336–356, 2015. (Cited in page 58)

[RJ17] Tilmann Rabl and Hans-Arno Jacobsen. Query centric partitioning and allocation for partially replicated database systems. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD, Chicago, IL, USA, May 14-19*, pages 315–330, 2017. (Cited in page 32), (Cited in page 43), (Cited in page 44), (Cited in page 52)

[RM75] Nick Roussopoulos and John Mylopoulos. Using semantic networks for database management. In *Proceedings of the International Conference on Very Large Data Bases, September 22-24, Framingham, Massachusetts, USA*, pages 144–172, 1975. (Cited in page 63)

[RPG+15] Fatemeh Rahimian, Amir H. Payberah, Sarunas Girdzijauskas, Márk Jelasity, and Seif Haridi. A distributed algorithm for large-scale graph partitioning. *ACM Trans. Auton. Adapt. Syst.*, 10(2):12:1–12:24, 2015. (Cited in page 75)

[RS11] Kurt Rohloff and Richard E. Schantz. Clause-iteration with mapreduce to scalably query datagraphs in the SHARD graph-store. In *Proceedings of the Fourth International Workshop on Data-intensive Distributed Computing, DIDC, San Jose, CA, USA, June 8*, pages 35–44, 2011. (Cited in page 86), (Cited in page 90)

[RZML02] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy M. Lohman. Automating physical database design in a parallel database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6*, pages 558–569, 2002. (Cited in page 8), (Cited in page 9), (Cited in page 21), (Cited in page 31), (Cited in page 47), (Cited in page 51)

[SABW13] Stephan Seufert, Avishek Anand, Srikanta J. Bedathur, and Gerhard Weikum. FERRARI: flexible and efficient reachability range assignment for graph indexing. In *29th IEEE International Conference on Data Engineering, ICDE , Brisbane, Australia, April 8-12*, pages 1009–1020, 2013. (Cited in page 85)

[San93] Laura A. Sanchis. Multiple-way network partitioning with different cost functions. *IEEE Trans. Computers*, 42(12):1500–1504, 1993. (Cited in page 73)

[Sch07] Satu Elisa Schaeffer. Graph clustering. *Comput. Sci. Rev.*, 1(1):27–64, 2007. (Cited in page 74)

[SGK+08] Lefteris Sidirourgos, Romulo Goncalves, Martin L. Kersten, Niels Nes, and Stefan Manegold. Column-store support for RDF data management: not all swans are white. *PVLDB*, 1(2):1553–1563, 2008. (Cited in page 87), (Cited in page 168)

[SH13a] Andy Seaborne Steve Harris. SPARQL 1.1 overview. `https://www.w3.org/TR/sparql11-overview/`, 2013. (Cited in page 2), (Cited in page 67), (Cited in page 159)

[SH13b] Andy Seaborne Steve Harris. SPARQL 1.1 query language. `https://www.w3.org/TR/sparql11-query/`, 2013. (Cited in page 81), (Cited in page 82)

[Shi81] David W. Shipman. The functional data model and the data language DAPLEX. *ACM Trans. Database Syst.*, 6(1):140–173, 1981. (Cited in page 63)

[Shn77] Ben Shneiderman. Recuced combined indexes for efficient multiple attribute retrieval. *Inf. Syst.*, 2(4):149–154, 1977. (Cited in page 83)

[Sim72] Robert F Simmons. *Semantic networks: their computation and use for understanding English sentences*. Department of Computer Sciences and Computer-Assisted Instruction Laboratory, University of Texas, 1972. (Cited in page 63), (Cited in page 166)

[SKK97] Kirk Schloegel, George Karypis, and Vipin Kumar. Multilevel diffusion schemes for repartitioning of adaptive meshes. *J. Parallel Distributed Comput.*, 47(2):109–124, 1997. (Cited in page 75)

[SKS+97] Abraham Silberschatz, Henry F Korth, Shashank Sudarshan, et al. *Database system concepts*, volume 4. McGraw-Hill New York, 1997. (Cited in page 28), (Cited in page 47)

[SKW07] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *Proceedings of the 16th International Conference on World Wide Web, WWW, Banff, Alberta, Canada, May 8-12*, pages 697–706, 2007. (Cited in page 3)

[SMR00] Thomas Stöhr, Holger Märtens, and Erhard Rahm. Multi-dimensional database allocation for parallel data warehouses. In *Proceedings of 26th International Conference on Very Large Data Bases, VLDB,Cairo, Egypt, September 10-14*, pages 273–284, 2000. (Cited in page 10), (Cited in page 21), (Cited in page 45), (Cited in page 52), (Cited in page 162)

[SPBL15] Alexander Schätzle, Martin Przyjaciel-Zablocki, Thorsten Berberich, and Georg Lausen. S2X: graph-parallel querying of RDF with graphx. In *Biomedical Data Management and Graph Online Querying Workshop*, pages 155–168, 2015. (Cited in page 90)

[SPL11] Alexander Schätzle, Martin Przyjaciel-Zablocki, and Georg Lausen. PigSPARQL: mapping SPARQL to Pig Latin. In *International Workshop on Semantic Web Information Management (SWIM)*, page 4, 2011. (Cited in page 90)

[SPNL14] Alexander Schätzle, Martin Przyjaciel-Zablocki, Antony Neu, and Georg Lausen. Sempala: Interactive SPARQL query processing on hadoop. In *International Conference on Semantic Web (ISWC)*, pages 164–179, 2014. (Cited in page 90)

[SPSL16] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. S2RDF: RDF querying with SPARQL on Spark. *Proc. VLDB Endow.*, 9(10):804–815, 2016. (Cited in page 86), (Cited in page 89), (Cited in page 90), (Cited in page 104), (Cited in page 171)

[Spy87] Nicolas Spyratos. The partition model: A deductive database model. *ACM Trans. Database Syst.*, 12(1):1–37, 1987. (Cited in page 21), (Cited in page 162)

[SRB+18] Kuldeep Singh, Arun Sethupat Radhakrishna, Andreas Both, Saeedeh Shekarpour, Ioanna Lytra, Ricardo Usbeck, Akhilesh Vyas, Akmal Khikmatullaev, Dharmen Punjani, Christoph Lange, Maria-Esther Vidal, Jens Lehmann, and Sören Auer. Why reinvent the wheel: Let's build question answering systems together. In *Proceedings of the World Wide Web Conference, WWW, Lyon, France, April 23-27*, pages 1247–1256, 2018. (Cited in page 4)

[SS11] Peter Sanders and Christian Schulz. Engineering multilevel graph partitioning algorithms. In *Proceedings of the 19th Annual European Symposium, Saarbrücken Algorithms - ESA , Germany, September 5-9*, pages 469–480, 2011. (Cited in page 74)

[STE+16] Marco Serafini, Rebecca Taft, Aaron J. Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. Clay: Fine-grained adaptive partitioning for general database schemas. *PVLDB*, 10(4):445–456, 2016. (Cited in page 50), (Cited in page 52)

[SW83] Domenico Saccà and Gio Wiederhold. Database partitioning in a cluster of processors. In *Proceedings of the 9th International Conference on Very Large Data Bases, October 31 - November 2, Florence, Italy*, pages 242–247, 1983. (Cited in page 31), (Cited in page 38), (Cited in page 52), (Cited in page 164)

[SW85] Domenico Saccà and Gio Wiederhold. Database partitioning in a cluster of processors. *ACM Trans. Database Syst.*, 10(1):29–56, 1985. (Cited in page 26), (Cited in page 31), (Cited in page 36), (Cited in page 109), (Cited in page 164), (Cited in page 172)

[SWL13] Bin Shao, Haixun Wang, and Yatao Li. Trinity: a distributed graph engine on a memory cloud. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, New York, NY, USA, June 22-27*, pages 505–516, 2013. (Cited in page 78)

[TBC+13] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From "think like a vertex" to "think like a graph". *Proc. VLDB Endow.*, 7(3):193–204, 2013. (Cited in page 79)

[Tea15] Bing Team. Bing announces availability of the knowledge and action graph api. https://blogs.bing.com/search/2015/08/20/bing-announces-availability-of-the-knowledge-and-action-graph-api-for-developers/, 2015. (Cited in page 3)

[TF76] Robert W. Taylor and Randall L. Frank. CODASYL data-base management systems. *ACM Comput. Surv.*, 8(1):67–103, 1976. (Cited in page 63)

[TGRV14] Charalampos E. Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. FENNEL: streaming graph partitioning for massive scale graphs. In *Seventh ACM International Conference on Web Search and Data Mining, WSDM, New York, NY, USA, February 24-28*, pages 333–342, 2014. (Cited in page 73), (Cited in page 75)

[TNST14] Khai Q. Tran, Jeffrey F. Naughton, Bruhathi Sundarmurthy, and Dimitris Tsirogiannis. JECB: a join-extension, code-based approach to OLTP data partitioning. In *International Conference on Management of Data, SIGMOD, Snowbird, UT, USA, June 22-27*, pages 39–50, 2014. (Cited in page 43), (Cited in page 44), (Cited in page 52)

[TÖ15] Volkan Tüfekçi and Can Özturan. Partitioning graph databases by using access patterns. In *Adaptive Resource Management and Scheduling for Cloud Computing - Second International Workshop, ARMS-CC, Held in Conjunction with ACM Symposium on Principles of Distributed Computing, PODC , Donostia-San Sebastián, Spain, July 20*, pages 158–176, 2015. (Cited in page 75), (Cited in page 76)

[Tor14] Nicolas Torzec. The yahoo knowledge graph, presented at the 10th semantic technology business conference. `https://www.slideshare.net/NicolasTorzec/the-yahoo-knowledge-graph`, 2014. (Cited in page 3)

[VCLM13] Luis M. Vaquero, Félix Cuadrado, Dionysios Logothetis, and Claudio Martella. xdgp: A dynamic graph processing system with adaptive partitioning. *CoRR*, abs/1309.1049, 2013. (Cited in page 75), (Cited in page 76)

[VHM+14] Ruben Verborgh, Olaf Hartig, Ben De Meester, Gerald Haesendonck, Laurens De Vocht, Miel Vander Sande, Richard Cyganiak, Pieter Colpaert, Erik Mannens, and Rik Van de Walle. Querying datasets on the web with high availability. In *13th International Semantic Web Conference - ISWC, Riva del Garda, Italy, October 19-23*, pages 180–196, 2014. (Cited in page 5)

[Vra12] Denny Vrandecic. Wikidata: a new platform for collaborative data collection. In *Proceedings of the 21st World Wide Web Conference, WWW, Lyon, France, April 16-20*, pages 1063–1064, 2012. (Cited in page 3)

[vRHK+16] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. PGQL: a property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, Redwood Shores, CA, USA, June 24 - 24*, page 7, 2016. (Cited in page 68)

[W3C04] W3C. Architecture of the world wide web, volume one. `https://www.w3.org/TR/webarch/#identification`, 2004. (Cited in page 1)

[W3C08] W3C. An introduction to multilingual web addresses. `https://www.w3.org/International/articles/idn-and-iri/`, 2008. (Cited in page 1), (Cited in page 80)

[WB87] W. Kevin Wilkinson and Haran Boral. KEV - A kernel for bubba. In *Proceedings of the 5th International Workshop in Database Machines and Knowledge Base Machines, Tokyo, Japan*, pages 31–44, 1987. (Cited in page 38)

[WC16] Marcin Wylot and Philippe Cudré-Mauroux. DiploCloud: Efficient and scalable management of RDF data in the cloud. *IEEE Trans. Knowl. Data Eng.*, 28(3):659–674, 2016. (Cited in page 90)

[Web12] Jim Webber. A programmatic introduction to neo4j. In *Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '12, Tucson, AZ, USA, October 21-25*, pages 217–218, 2012. (Cited in page 57), (Cited in page 66), (Cited in page 77), (Cited in page 78)

[WHCS18] Marcin Wylot, Manfred Hauswirth, Philippe Cudré-Mauroux, and Sherif Sakr. RDF data storage and query processing schemes: A survey. *ACM Comput. Surv.*, 51(4):84:1–84:36, 2018. (Cited in page 80), (Cited in page 86)

[Wil06] Kevin Wilkinson. Jena property table implementation, 2006. (Cited in page 6), (Cited in page 83)

[WRT20] Inria CNRS Wimmics Research Team, University Côte d'Azur. Covid-on-the-web dataset. `https://github.com/Wimmics/CovidOnTheWeb`, 2020. (Cited in page 3)

[XGFS13] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. GraphX: a resilient distributed graph system on Spark. In *First International Workshop on Graph Data Management Experiences and Systems, GRADES, co-located with SIGMOD/PODS, New York, NY, USA, June 24*, page 2, 2013. (Cited in page 78)

[YYZK12] Shengqi Yang, Xifeng Yan, Bo Zong, and Arijit Khan. Towards effective partition management for large graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Scottsdale, AZ, USA, May 20-24*, pages 517–528, 2012. (Cited in page 75), (Cited in page 76), (Cited in page 90)

[ZBS15] Erfan Zamanian, Carsten Binnig, and Abdallah Salama. Locality-aware partitioning in parallel database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4*, pages 17–30, 2015. (Cited in page 43), (Cited in page 44), (Cited in page 47), (Cited in page 52)

[ZCTW13] Xiaofei Zhang, Lei Chen, Yongxin Tong, and Min Wang. EAGRE: towards scalable I/O efficient SPARQL query evaluation on the cloud. In *29th IEEE ICDE, Brisbane, Australia, April 8-12*, pages 565–576, 2013. (Cited in page 84), (Cited in page 86), (Cited in page 88), (Cited in page 90), (Cited in page 137)

[ZMG+20] Ishaq Zouaghi, Amin Mesmoudi, Jorge Galicia, Ladjel Bellatreche, and Taoufik Aguili. Query optimization for large scale clustered RDF data. In *Proceedings of the 22nd International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data co-located with EDBT/ICDT, Copenhagen, Denmark, March 30*, pages 56–65, 2020. (Cited in page 116), (Cited in page 123), (Cited in page 127)

[ZÖC+14] Lei Zou, M. Tamer Özsu, Lei Chen, Xuchuan Shen, Ruizhe Huang, and Dongyan Zhao. gStore: a graph-based SPARQL query engine. *VLDB Journal*, 23(4):565–590, 2014. (Cited in page 6), (Cited in page 86), (Cited in page 121), (Cited in page 128), (Cited in page 134), (Cited in page 174)

[ZRL+04] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy M. Lohman, Adam J. Storm, Christian Garcia-Arellano, and Scott Fadden. DB2 design advisor: Integrated automatic physical database design. In *Proceedings of the Thirtieth International Conference*

on Very Large Data Bases, Toronto, Canada, August 31 - September 3, pages 1087–1097, 2004. (Cited in page 21), (Cited in page 47), (Cited in page 48), (Cited in page 51)

[ZYW+13] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A distributed graph engine for web scale RDF data. *Proc. VLDB Endow.*, 6(4):265–276, 2013. (Cited in page 86), (Cited in page 90)

Appendix A

Logic fragmentation example

A.1 Raw data & encoding

```
<:Airplane1> <:has_model> <:A340> .
<:Airplane1> <:has_length> "75.4" .
<:Airplane1> <:nb_motors> "4" .
<:Airplane1> <:type> <Airliner> .
<:Airplane1> <:manufacturer> <Airbus> .
<:Airplane2> <:has_model> <:B747> .
<:Airplane2> <:has_length> "70.1" .
<:Airplane2> <:nb_motors> "4" .
<:Airplane2> <:type> <Airliner> .
<:Airplane2> <:manufacturer> <Boeing> .
<:Airbus> <:office_in> "Toulouse" .
<:Airbus> <:has_seat> <:France> .
<:Boeing> <:office_in> "Chicago" .
<:Boeing> <:has_seat> <:USA> .
<:USA> <:has_city> "Chicago" .
<:France> <:has_city> "Toulouse" .
<:B747> <:nb_version> "5" .
<:A340> <:nb_version> "4" .
```

Predicate	ID
has_city	0
has_length	1
has_model	2
has_seat	3
manufacturer	4
nb_motors	5
nb_version	6
office_in	7
type	8

String	ID
:A340	10
:Airbus	20
:Airliner	30
:Airplane1	40
:Airplane2	50
:B747	60
:Boeing	70
Chicago	80
:France	90
Toulouse	100
:USA	110

A.2 Forward graph fragment

```
{ "0" :{
    "id":1,
    "90" :{ "0":[100]},
    "110" :{ "0":[80]}
  },
   "6" :{
   "id":2,
    "10" :{ "6":[4^int]},
    "60" :{ "6":[5^int]}
  },
"3-7" :{
   "id":3,
    "20" :{ "3":[90], "7":[100]},
    "70" :{ "3":[110], "7":[60]}
  },
"1-2-4-5-8" :{
    "id":4,
    "40" :{ "1":[75.4^float], "2":[10], "4":[20], "5":[4^int], "8":[30] },
    "50" :{ "1":[70.1^float], "2":[60], "4":[70], "5":[4^int], "8":[30]}
    }
}
```

A.3 Backward graph fragment

```
{  "1" :{
    "id":5,
    "75.4^float" :{"1":[40]},
    "70.1^float" :{"1":[50]}
  },
  "2" :{
    "id":6,
    "10" :{ "2":[40]},
    "60" :{ "2":[50]}
  },
  "3" :{
    "id":7,
    "90" :{ "3":[20]},
    "110" :{ "3":[70]}
  },
  "4" :{
    "id":8,
    "20" :{ "4":[40]},
    "70" :{ "4":[50]}
  },
  "6" :{
    "id":9,
    "5^int" :{ "6":[60]}
```

```
  },
  "8" :{
    "id":10,
    "30" :{ "8":[40,50]}
  },
  "0-7" :{
    "id":11,
    "100" :{ "0":[20],"7":[90]},
    "80" :{ "0":[70],"7":[110]}
  },
  "5-6" :{
    "id":12,
    "4^int" :{ "5":[40,50],"6":[10]}
  }
}
```

Appendix B

Queries

B.1 Watdiv

B.1.1 Prefixes

```
@prefix pr: <http://purl.org/stuff/rev#>
@prefix prg: <http://purl.org/goodrelations/>
@prefix foaf: <http://xmlns.com/foaf/>
@prefix sc: <http://schema.org/>
@prefix prt: <http://purl.org/dc/terms/>
@prefix pro: <http://purl.org/ontology/mo/>
@prefix rdfs: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix w: <http://db.uwaterloo.ca/~galuc/wsdbm/>
@prefix ogp: <http://ogp.me/ns#>
@prefix geo: <http://www.geonames.org/ontology#>
```

C: Complex
S: Star
F: Snowflake
L: Linear

B.1.2 Queries

C1

```
SELECT ?v0 ?v4 ?v6 ?v7 WHERE {
  ?v0 sc:caption ?v1 .
  ?v0 sc:text ?v2 .
  ?v0 sc:contentRating ?v3 .
  ?v0 pr:hasReview ?v4 .
  ?v4 pr:title ?v5 .
  ?v4 pr:reviewer ?v6 .
  ?v7 sc:actor ?v6 .
  ?v7 sh:language ?v8 .
  }
```

C2

```
SELECT ?v0 ?v3 ?v4 ?v8 WHERE {
  ?v0 sc:legalName ?v1 .
  ?v0 prg:offers ?v2 .
  ?v2 sc:eligibleRegion ?v10 .
  ?v2 prg:includes ?v3 .
  ?v4 sc:jobTitle ?v5 .
  ?v4 foaf:homepage ?v6 .
  ?v4 w:makesPurchase ?v7 .
  ?v7 w:purchaseFor ?v3 .
  ?v3 pr:hasReview ?v8 .
  ?v8 pr:totalVotes ?v9 .
  }
```

C3

```
SELECT ?v0 WHERE {
   ?v0 w:likes ?v1 .
   ?v0 w:friendOf ?v2 .
   ?v0 prt:Location ?v3 .
   ?v0 foaf:age ?v4 .
   ?v0 w:gender ?v5 .
   ?v0 w:givenName ?v6 .
   }
```

S1

```
SELECT ?v0 ?v1 ?v3 ?v4 ?v5 ?v6 ?v7
    ?v8 ?v9 WHERE {
   ?v0 prg:includes ?v1 .
   ?V2 prg:offers ?v0 .
   ?v0 prg:price ?v3 .
   ?v0 prg:serialNumber ?v4 .
   ?v0 prg:validFrom ?v5 .
   ?v0 prg:validThrough ?v6 .
   ?v0 sc:eligibleQuantity ?v7 .
   ?v0 sc:eligibleRegion ?v8 .
   ?v0 <sc:priceValidUntil ?v9 .
   }
```

S2

```
SELECT ?v0 ?v1 ?v3 WHERE {
   ?v0 prt:Location ?v1 .
   ?v0 sc:nationality w:Country14 .
   ?v0 w:gender ?v3 .
   ?v0 rdfs:type w:Role2 .
   }
```

S3

```
SELECT ?v0 ?v2 ?v3 ?v4 WHERE {
   ?v0 rdfs:type w:ProductCategory14 .
   ?v0 sc:caption ?v2 .
   ?v0 w:hasGenre ?v3 .
   ?v0 sc:publisher ?v4 .
   }
```

S4

```
SELECT ?v0 ?v2 ?v3 WHERE {
   ?v0 foaf:age w:AgeGroup1 .
   ?v0 foaf:familyName ?v2 .
   ?v3 pro:artist ?v0 .
   ?v0 sc:nationality w:Country1 .
   }
```

S5

```
SELECT ?v0 ?v2 ?v3 WHERE {
   ?v0 rdfs:type w:ProductCategory7 .
   ?v0 sc:description ?v2 .
   ?v0 sc:keywords ?v3 .
   ?v0 sc:language w:Language0 .
   }
```

S6

```
SELECT ?v0 ?v1 ?v2 WHERE {
   ?v0 pro:conductor ?v1 .
   ?v0 rdfs:type ?v2 .
   ?v0 w:hasGenre w:SubGenre83 .
   }
```

S7

```
SELECT ?v0 ?v1 ?v2 WHERE {
   ?v0 rdfs:type ?v1 .
   ?v0 sc:text ?v2 .
   w:User145869 w:likes ?v0 .
   }
```

F1

```
SELECT ?v0 ?v2 ?v3 ?v4 ?v5 WHERE {
   ?v0 ogp:tag w:Topic172 .
   ?v0 rdfs:type ?v2 .
   ?v3 sc:trailer ?v4 .
   ?v3 sc:keywords ?v5 .
   ?v3 w:hasGenre ?v0 .
   ?v3 rdfs:type> w:ProductCategory2 .
   }
```

F2

```
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 ?v7
    WHERE
 {
   ?v0 foaf:homepage ?v1 .
   ?v0 ogp:title ?v2 .
   ?v0 rdfs:type ?v3 .
   ?v0 sc:caption ?v4 .
   ?v0 sc:description ?v5 .
   ?v1 sc:url ?v6 .
   ?v1 w:hits ?v7 .
   ?v0 w:hasGenre w:SubGenre55 .
   }
```

F3

```
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 WHERE {
   ?v0 sc:contentRating ?v1 .
   ?v0 sc:contentSize ?v2 .
   ?v0 w:hasGenre w:SubGenre42 .
   ?v4 w:makesPurchase ?v5 .
   ?v5 w:purchaseDate ?v6 .
   ?v5 w:purchaseFor ?v0 .
   }
```

F4

```
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 ?v7
    ?v8 WHERE {
   ?v0 foaf:homepage ?v1 .
   ?v2 prg:includes ?v0 .
   ?v0 ogp:tag w:Topic71 .
   ?v0 sc:description ?v4 .
   ?v0 sc:contentSize ?v8 .
   ?v1 sc:url ?v5 .
   ?v1 w:hits ?v6 .
   ?v1 sc:language> w:Language0 .
   ?v7 w:likes ?v0 .
   }
```

F5

```
SELECT ?v0 ?v1 ?v3 ?v4 ?v5 ?v6 WHERE {
   ?v0 prg:includes ?v1 .
   w:Retailer28001 prg:offers ?v0 .
   ?v0 prg:price ?v3 .
   ?v0 prg:validThrough ?v4 .
   ?v1 ogp:title ?v5 .
   ?v1 rdfs:type ?v6 .
   }
```

L1

```
SELECT ?v0 ?v2 ?v3 WHERE {
   ?v0 w:subscribes w:Website18627 .
   ?v2 sc:caption ?v3 .
   ?v0 w:likes ?v2 .
   }
```

L2

```
SELECT ?v1 ?v2 WHERE {
   w:City107 geo:parentCountry ?v1 .
   ?v2 w:likes w:Product0 .
   ?v2 sc:nationality ?v1 .
   }
```

L3

```
SELECT ?v0 ?v1 WHERE {
   ?v0 w:likes ?v1 .
   ?v0 w:subscribes ?v2 .
   }
```

L4

```
SELECT ?v0 ?v2 WHERE {
   ?v0 ogp:tag w:Topic132 .
   ?v0 sc:caption ?v2 .
   }
```

L5

```
 SELECT ?v0 ?v1 ?v3 WHERE {
   ?v0 sc:jobTitle ?v1 .
   w:City24 geo:parentCountry ?v3 .
   ?v0 sc:nationality ?v3 .
   }
```

B.2 LUBM

B.2.1 Prefixes

```
@prefix rdfs: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix lb: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
@prefix u1: <http://www.Department0.University0.edu/>
@prefix u2: <http://www.University0.edu>
```

B.2.2 Queries

1

```
SELECT ?x WHERE {
?x rdfs:type lb:GraduateStudent .
?x lb:takesCourse u1:GraduateCourse0 .
}
```

2

```
SELECT ?x ?y WHERE {
?x rdfs:type lb:GraduateStudent .
?y rdfs:type lb:University .
?z rdfs:type lb:Department> .
?x lb:memberOf ?z .
?z lb:subOrganizationOf ?y .
?x lb:undergraduateDegreeFrom ?y .
}
```

3

```
SELECT ?x WHERE {
?x rdfs:type lb:Publication .
?x lb:publicationAuthor
    u1:AssistantProfessor0 .
}
```

4

```
SELECT ?x ?y1 ?y3 WHERE {
?x lb:worksFor u1 .
?x lb:name ?y1 .
?x lb:emailAddress ?y2 .
?x lb:telephone ?y3 .
}
```

5

```
SELECT ?x WHERE
{
?x lb:memberOf u1 .
}
```

6

```
SELECT ?x WHERE {
?x rdfs:type lb:UndergraduateStudent .
}
```

7

```
SELECT ?x ?y WHERE {
?x rdfs:type lb:UndergraduateStudent .
?x lb:takesCourse ?y .
?y rdfs:type lb:Course .
u1:AssociateProfessor0 lb:teacherOf
    ?y .
}
```

8

```
SELECT ?x ?y WHERE {
?x rdfs:type lb:UndergraduateStudent .
?x lb:memberOf ?y .
?x lb:emailAddress ?z .
?y rdfs:type lb:Department .
?y lb:subOrganizationOf u1 .
}
```

9

```
SELECT ?x ?y WHERE {
?x rdfs:type lb:UndergraduateStudent .
?y rdfs:type lb:FullProfessor .
?z rdfs:type lb:Course .
?x lb:advisor ?y .
?x lb:takesCourse ?z .
?y lb:teacherOf ?z .
}
```

10

```
SELECT ?x WHERE {
?x rdfs:type lb:GraduateStudent .
?x lb:takesCourse u1:GraduateCourse0 .
}
```

11

```
SELECT ?x ?y WHERE {
?x rdfs:type lb:ResearchGroup .
?x lb:subOrganizationOf ?y .
?y lb:subOrganizationOf u2 .
}
```

12

```
SELECT ?x ?y WHERE {
?x lb:headOf ?y .
?y rdfs:type lb:Department .
?y lb:subOrganizationOf u2 .
}
```

13

```
SELECT ?x WHERE {
?x lb:undergraduateDegreeFrom u2 .
}
```

14

```
SELECT ?x WHERE {
?x rdfs:type lb:UndergraduateStudent .
}
```

B.3 DBLP

B.3.1 Prefixes

```
@prefix foaf: <http://xmlns.com/foaf/0.1#>
@prefix el: <http://purl.org/dc/elements/1.1#>
@prefix o: <http://swrc.ontoware.org/ontology#>
@prefix te: <http://purl.org/dc/terms#>
@prefix owl: <http://www.w3.org/2002/07/owl#>
@prefix dblp: <http://dblp.uni-trier.de/rec/bibtex/series/cogtech/>
@prefix rdfs: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix book: <http://dblp.l3s.de/d2r/resource/publications/books/ph/>
@prefix author: <http://dblp.l3s.de/d2r/resource/authors/>
```

C: Complex
S: Star
L: Linear

B.3.2 Queries

C1

```
SELECT ?v0 ?v4 ?v6 ?v5 WHERE {
   ?v0 foaf:maker ?v1 .
   ?v3 el:creator ?v1 .
   ?v5 o:editor ?v2 .
   ?v2 te:tableOfContent ?v4 .
   ?v2 te:issued ?v6 .
   }
```

C2

```
SELECT ?v0 ?v4 ?v6 ?v5 WHERE {
   ?v0 te:issued ?v1 .
   ?v0 el:identifier ?v2 .
   ?v0 o:volume ?v3 .
   ?v5 el:type ?v4 .
   ?v6 el:subject ?v4 .
   }
```

C3

```
SELECT ?v0 ?v1 ?v2 WHERE {
   ?v0 el:title ?v1 .
   ?v0 owl:sameAs ?v3 .
   ?v0 te:bibliographicCitation ?v2 .
   }
```

L1

```
SELECT ?v0 ?v1 ?v2 ?v3 WHERE {
   ?v0 el:creator ?v1 .
   ?v0 foaf:maker ?v2 .
   ?v0 foaf:homepage ?v3 .
   }
```

L2

```
SELECT ?v0 ?v1 WHERE {
   ?v0 te:tableOfContent ?v1 .
   ?v0 te:bibliographicCitation
       dblp:Helbig2006 .
   }
```

L3

```
SELECT ?v0 ?v1 ?v2 ?v3 WHERE {
   ?v0 o:month ?v1 .
   ?v2 el:subject ?v1 .
   ?v3 o:isbn ?v1 .
   }
```

L4

```
SELECT ?v0 ?v1 WHERE {
   ?v0 rdfs:seeAlso book:Shasha92 .
   ?v0 te:tableOfContent ?v1 .
   }
```

S1

```
SELECT ?v0 ?v1 ?v2 ?v5 WHERE {
   ?v0 el:creator ?v1 .
   ?v0 foaf:maker ?v2 .
   ?v0 o:journal ?v3 .
   ?v0 rdfs:type ?v4 .
   ?v0 rdfs:label ?v5 .
   }
```

S2

```
SELECT ?v0 ?v4 ?v6 WHERE {
   ?v0 te:tableOfContent ?v1 .
   ?v0 o:editor
       author:Rodney_W._Topor .
   ?v0 o:number ?v3 .
   ?v0 rdfs:label ?v4 .
   ?v0 te:issued ?v5 .
   ?v0 te:bibliographicCitation ?v6 .
   }
```

S3

```
SELECT ?v0 ?v3 ?v1 WHERE {
   ?v0 rdfs:seeAlso ?v1 .
   ?v3 foaf:homepage ?v1 .
   }
```

S4

```
SELECT ?v0 ?v1 ?v3 WHERE {
   ?v0 rdfs:seeAlso ?v1 .
   ?v3 foaf:page ?v1 .
   }
```

B.4 Yago

B.4.1 Prefixes

```
@prefix yago: <http://yago-knowledge.org/resource/>
@prefix rdfs: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

X

B.4.2 Queries

1

```
SELECT ?GivenName ?FamilyName WHERE {
    ?p yago:hasGivenName ?GivenName .
    ?p yago:hasFamilyName ?FamilyName
        .
    ?p yago:wasBornIn ?city .
    ?p yago:hasAcademicAdvisor ?a .
    ?a yago:wasBornIn ?city .
    }
```

2

```
SELECT ?GivenName ?FamilyName WHERE {
    ?p yago:hasGivenName ?GivenName .
    ?p yago:hasFamilyName ?FamilyName
        .
    ?p yago:wasBornIn ?city .
    ?p yago:hasAcademicAdvisor ?a .
    ?a yago:wasBornIn ?city .
    ?p yago:isMarriedTo ?p2 .
    ?p2 yago:wasBornIn ?city .
    }
```

3

```
SELECT ?name1 ?name2 WHERE {
    ?a1 yago:hasPreferredName ?name1 .
    ?a2 yago:hasPreferredName ?name2 .
    ?a1 yago:actedIn ?movie .
    ?a2 yago:actedIn ?movie .
    }
```

4

```
SELECT ?name1 ?name2 WHERE {
    ?p1  yago:hasPreferredName ?name1
        .
    ?p2  yago:hasPreferredName ?name2
        .
    ?p1  yago:isMarriedTo ?p2 .
    ?p1  yago:wasBornIn ?city .
    ?p2  yago:wasBornIn ?city .
    }
```

5

```
 SELECT ?p WHERE {
    ?p yago:hasGivenName> ?gn.
    ?p rdfs:type ?h.
    ?p yago:wasBornIn ?city.
    ?city yago:isLocatedIn ?i.
    ?p yago:livesIn ?a.
    ?a yago:wasBornIn ?city2.
    ?city2 yago:isLocatedIn ?j.
    }
```

List of Figures

List of Tables

Résumé

Le Resource Description Framework (RDF) et SPARQL sont des standards très populaires basés sur des graphes initialement conçus pour représenter et interroger des informations sur le Web. La flexibilité offerte par RDF a motivé son utilisation dans d'autres domaines. Aujourd'hui les jeux de données RDF sont d'excellentes sources d'information. Ils rassemblent des milliards de triplets dans des Knowledge Graphs qui doivent être stockés et exploités efficacement. La première génération de systèmes RDF a été construite sur des bases de données relationnelles traditionnelles. Malheureusement, les performances de ces systèmes se dégradent rapidement car le modèle relationnel ne convient pas au traitement des données RDF intrinsèquement représentées sous forme de graphe. Les systèmes RDF natifs et distribués cherchent à surmonter cette limitation. Les premiers utilisent principalement l'indexation comme stratégie d'optimisation pour accélérer les requêtes. Les deuxièmes recourent au partitionnement des données. Dans le modèle relationnel, la représentation logique de la base de données est cruciale pour concevoir le partitionnement. La couche logique définissant le schéma explicite de la base de données offre un certain confort aux concepteurs. Cette couche leur permet de choisir manuellement ou automatiquement, via des assistants automatiques, les tables et les attributs à partitionner. Aussi, elle préserve les concepts fondamentaux sur le partitionnement qui restent constants quel que soit le système de gestion de base de données. Ce schéma de conception n'est plus valide pour les bases de données RDF car le modèle RDF n'applique pas explicitement un schéma aux données. Ainsi, la couche logique est inexistante et le partitionnement des données dépend fortement des implémentations physiques des triplets sur le disque. Cette situation contribue à avoir des logiques de partitionnement différentes selon le système cible, ce qui est assez différent du point de vue du modèle relationnel. Dans cette thèse, nous promouvons l'idée d'effectuer le partitionnement de données au niveau logique dans les bases de données RDF. Ainsi, nous traitons d'abord le graphe de données RDF pour prendre en charge le partitionnement basé sur des entités logiques. Puis, nous proposons un framework pour effectuer les méthodes de partitionnement. Ce framework s'accompagne de procédures d'allocation et de distribution des données. Notre framework a été incorporé dans un système de traitement des données RDF centralisé (RDF_QDAG) et un système distribué (gStoreD). Nous avons mené plusieurs expériences qui ont confirmé la faisabilité de l'intégration de notre framework aux systèmes existants en améliorant leurs performances pour certaines requêtes. Enfin, nous concevons un ensemble d'outils de gestion du partitionnement de données RDF dont un langage de définition de données (DDL) et un assistant automatique de partitionnement.

Mots-clés : Bases de données–Conception; Bases des données–Gestion; Graphes, Théorie des; Partionnement de graphes; Resource Description Framework (informatique); SPARQL (langage de programmation); Structure logique; Systèmes experts (informatique)

Abstract

The Resource Description Framework (RDF) and SPARQL are very popular graph-based standards initially designed to represent and query information on the Web. The flexibility offered by RDF motivated its use in other domains and today RDF datasets are great information sources. They gather billions of triples in Knowledge Graphs that must be stored and efficiently exploited. The first generation of RDF systems was built on top of traditional relational databases. Unfortunately, the performance in these systems degrades rapidly as the relational model is not suitable for handling RDF data inherently represented as a graph. Native and distributed RDF systems seek to overcome this limitation. The former mainly use indexing as an optimization strategy to speed up queries. Distributed and parallel RDF systems resorts to data partitioning. The logical representation of the database is crucial to design data partitions in the relational model. The logical layer defining the explicit schema of the database provides a degree of comfort to database designers. It lets them choose manually or automatically (through advisors) the tables and attributes to be partitioned. Besides, it allows the partitioning core concepts to remain constant regardless of the database management system. This design scheme is no longer valid for RDF databases. Essentially, because the RDF model does not explicitly enforce a schema since RDF data is mostly implicitly structured. Thus, the logical layer is inexistent and data partitioning depends strongly on the physical implementations of the triples on disk. This situation contributes to have different partitioning logics depending on the target system, which is quite different from the relational model's perspective. In this thesis, we promote the novel idea of performing data partitioning at the logical level in RDF databases. Thereby, we first process the RDF data graph to support logical entity-based partitioning. After this preparation, we present a partitioning framework built upon these logical structures. This framework is accompanied by data fragmentation, allocation, and distribution procedures. This framework was incorporated to a centralized (RDF_QDAG) and a distributed (gStoreD) triple store. We conducted several experiments that confirmed the feasibility of integrating our framework to existent systems improving their performances for certain queries. Finally, we design a set of RDF data partitioning management tools including a data definition language (DDL) and an automatic partitioning wizard.

Keywords : Database design; Database management; Graph theory; Knowledge graph; RDF (Document markup language); SPARQL (Computer program language); Logic design; Expert systems (Computer science); Performance

---

Secteur de recherche : Informatique et applications

Laboratoire d'Informatique et d'Automatique pour les Systèmes
Ecole Nationale Supérieure de Mécanique et d'Aérotechnique
Téléport 2 – 1 avenue Clément Ader – BP 40109 – 86961 FUTUROSCOPE CHASSENEUIL CEDEX
Tél : 05.49.49.80.63 – Fax : 05.49.49.80.64