ExpRalytics : Expressive and Efficient Analytics for RDF Graphs

Pawel Guzewicz

INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2021IPPAX063

Thèse de doctorat

ÉCOLE POLYTECHNIQUE

IP PARIS

ExpRalytics: Expressive and Efficient Analytics for RDF Graphs

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à l'École polytechnique

École doctorale n°626 École doctorale de l'Institut Polytechnique de Paris (ED IP Paris)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Palaiseau, le 6 octobre 2021, par

PAWEŁ GUZEWICZ

Composition du Jury :

Sihem Amer-Yahia Directrice de recherche, CNRS and Univ. Grenoble Alpes, Laboratoire d'Informatique de Grenoble (LIG)	Rapporteuse
Volker Markl Full professor, Technische Universität Berlin, Berlin Institute for the Foundations of Learning and Data, German Research Center for Artificial Intelligence	Rapporteur
Angela Bonifati Full professor, Université Claude Bernard Lyon 1, Laboratoire d'Informatique en Image et Systèmes d'information (Liris)	Examinatrice
Fabian Suchanek Full professor, Télécom Paris (Institut Polytechnique de Paris), Laboratoire Traitement et Communication de l'Information (LTCI)	Examinateur (Président)
Federico Ulliana Associate Professor, Montpellier University, Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM)	Examinateur
Benoît Groz Maître de Conférences, Université Paris-Saclay, Laboratoire de Recherche en Informatique (LRI)	Examinateur
Ioana Manolescu Directrice de recherche, Inria and École polytechnique (Institut Polytechnique de Paris), Laboratoire d'informatique de l'École polytechnique (LIX)	Directrice de thèse
Yanlei Diao Professeure, École polytechnique (Institut Polytechnique de Paris) and Inria, Laboratoire d'informatique de l'École polytechnique (LIX)	Co-encadrante de thèse
François Goasdoué Full Professor, Univ. Rennes 1, Le laboratoire français de recherche et d'innovation en sciences et technologies du numérique (IRISA)	Invité

PAWEŁ GUZEWICZ

EXPRALYTICS: EXPRESSIVE AND EFFICIENT ANALYTICS FOR RDF GRAPHS

PHD THESIS

EXPRALYTICS: EXPRESSIVE AND EFFICIENT ANALYTICS FOR RDF GRAPHS

by

PAWEŁ GUZEWICZ

THESIS TITLE:
ExpRalytics: Expressive and Efficient Analytics for RDF Graphs

PHD CANDIDATE:
Paweł Guzewicz

SUPERVISORS:
Ioana Manolescu, senior researcher at Inria, France
Yanlei Diao, professor at École Polytechnique, France

ACADEMIC INSTITUTION:
Institut Polytechnique de Paris
Département d'Informatique, de Données et d'Intelligence Artificielle
Laboratoire d'Informatique de l'École Polytechnique

LOCATION:
Palaiseau, France

TIME FRAME:
October 1, 2018 – September 30, 2021

DEFENSE:
October 6, 2021

In loving memory of my grandpa Adolf Guzewicz
And to my closest family:
To my dad Jarek and my mom Agata, to my sister Magda,
To my grandma Helena, and to my uncle Wiesław
In gratitude for their limitless support

Ku pamięci mojego dziadka Adolfa Guzewicza
Oraz mojej najbliższej rodzinie:
Tacie Jarkowi i mamie Agacie, siostrze Magdzie,
Babci Helenie oraz wujkowi Wiesławowi
W podzięce za Wasze bezgraniczne wsparcie

ACKNOWLEDGMENTS

I would love to thank both of my supervisors for their great aid, support, and mentorship throughout my PhD studies. I have learned a lot from them, and I very much appreciate it.

Thank you, Ioana, for your precise feedback, guidance, and your relentless effort into perfecting our work.

Thank you, Yanlei, for providing original insight into our research and your endeavor for clarity in our writing.

I also wish to thank my close collaborator, Mirjana Mazuran.

Thank you, Mirjana, for sharing your academic experience with me and for our numerous and fruitful research discussions.

On a personal note, I would like to add that I have tremendously enjoyed a plenitude of stimulating discussions with all of you, as well as with other members of the CEDAR team of Inria Saclay and LIX. I will always fondly remember this time.

ABSTRACT

Large (Linked) Open Data are increasingly shared as RDF graphs today. However, such data has not yet reached its full potential in terms of sharing and reuse. The main bottleneck here lies in the capacity of human users to explore, discover, and grasp the content and insights of RDF graphs, which are inherently heterogeneous and can be both large and complex.

In the first part of this thesis, we provide new methods to meaningfully summarize data graphs, with a particular focus on RDF graphs. One class of tools for this task is *structural RDF graph summaries*, which allow users to grasp the different connections between RDF graph nodes. To this end, we introduce our novel RDFQuotient tool that finds compact yet informative RDF graph summaries that can serve as first-sight visualizations of an RDF graph's structure. These summaries, based on the notion of quotient graphs, are easy to understand for casual users; they provide an overview of the complete structure of an RDF graph while being typically many orders of magnitude smaller. Our summarization algorithms have a linear time complexity in the size of the input graph. Further, we propose incremental summarization algorithms capable of bringing the smallest needed adjustments to a summary to reflect modifications in the input graph. We also propose novel algorithms for building the summaries in a parallel shared-nothing architecture and instantiate them to the Apache Spark platform.

In the second part of this thesis, we consider the problem of *automatically identifying the k most interesting aggregate queries* that we can evaluate on an RDF graph, given an integer k and a user-specified *interestingness function*. Aggregate queries are routinely used to learn insights from relational data warehouses, and some prior research has addressed the problem of automatically recommending interesting aggregate queries. However, the RDF setting presents several differences from the traditional data warehouse setting:

1. In an RDF graph, we are not *given* but we must *identify* the facts, dimensions, and measures that compose aggregate queries;

2. Relational OLAP algorithms for efficiently evaluating multiple aggregates cannot handle the presence of multi-valued dimensions for a given fact; such dimensions are quite frequently found in RDF data facts and may have zero, one, or more values for dimensions.

To address these challenges, we devise Spade, an *extensible end-to-end framework* that enables the identification and evaluation of interesting

aggregates based on MVDCube, our new *RDF-compatible one-pass algorithm for efficiently evaluating a lattice of aggregates*, and a novel *early-stop technique* (with probabilistic guarantees) that prunes uninteresting aggregates and, as a result, reduces the aggregate evaluation cost. Experiments using both real and synthetic graphs demonstrate the ability of our framework to find interesting aggregates in a large search space, the efficiency of our algorithms, and scalability as the data size and complexity grow.

RÉSUMÉ

Les données ouvertes sont souvent partagés sous la forme de graphes RDF, qui sont une incarnation du principe *Linked Open Data* (données ouvertes liées). De telles données n'ont toutefois pas atteint leur entier potentiel d'utilisation et de partage. L'obstacle pour ce faire réside principalement au niveau de la capacité des utilisateurs à explorer, découvrir et saisir le contenu et des graphes RDF ; cette tâche est complexe car les graphes sont naturellement hétérogènes, et peuvent être à la fois volumineux et complexes.

Dans la première partie de cette thèse, nous proposons de nouvelles méthodes pour résumer de grands graphes de données, avec un accent particulier sur les graphes RDF. Un outil particulièrement puissant pour cette tâche est *un résumé structurel d'un graphe RDF structurels* ; ce résumé informe les utilisateurs sur les différentes connexions entre les nœuds de graphe RDF. À cette fin, nous avons proposé une nouvelle approché pour la construction de résumés structurels de graphes RDF, à savoir RDFQuotient ; les résumés qu'il construit peuvent servir de première visualisation de la structure d'un graphe RDF, tout en étant plusieurs plus compacts, souvent de plusieurs ordres de grandeur. Nous avons identifié une famille de quatre tels résumés, utilisant différentes relations d'équivalence entre les noeuds et/ou utilisant de différentes manière les types éventuellement présents dans les graphes RDF. Nous avons proposé des algorithmes capables de construire ces résumés ; tous ces algorithmes sont très efficace puisque complexité de calcul dépend de façon linéaire de la taille du graphe. Nous avons aussi proposé des variantes incrémentales de nos algorithmes, qui le font évoluer de manière efficace en appliquant juste les modifications nécessaires afin de lui permettre de refléter des modifications dans le graphe d'entrée. Nous avons également proposé de nouveaux algorithmes pour construire les résumés dans une architecture parallèle, et les avons instanciés sur la plate-forme Apache Spark.

Dans la deuxième partie de cette thèse, nous considérons le problème *d'identifier automatiquement les requêtes d'agrégation les plus intéressantes* qui peuvent être évaluées sur un graphe RDF, étant donnée une *fonction d'intérêt* spécifiée par l'utilisateur. Les requêtes d'agrégation sont couramment utilisées pour analyser des entrepôts de données relationnelles, et certaines recherches antérieures ont abordé le problème de la recommandation automatique des requêtes d'agrégation les plus intéressantes. Cependant, le problème est assez différent dans le contexte d'un entrepôt de données RDF :

1. Dans un graphe RDF, nous devons *identifier* les faits, les dimensions et les mesures qui composent une requête d'agrégation,

alors que dans le cadre relationnel, ces informations sont déterminées par le schéma de l'entrepôt ;

2. Les algorithmes OLAP capable d'évaluer efficacement, conjointement, plusieurs requêtes d'agrégation ne s'appliquent pas en présence de dimensions à valeurs multiples pour un fait donné ; de telles dimensions sont assez fréquentes dans les données RDF (où un fait peut avoir zéro, une ou plusieurs valeurs pour chaque dimension).

Nous avons proposé Spade, un *approche nouvelle, complète et extensible*, qui permet l'identification et l'évaluation de requêtes d'agrégation intéressantes. Au coeur de l'exploration est MVDCube, notre nouvel *algorithme spécialement conçu pour RDF, capable d'évaluer efficacement un treillis d'agrégats*. Par ailleurs, nous avons proposé une nouvelle *technique d'arrêt précoce du calcul d'un aggrégat* (avec des garanties probabilistes) ; cette technique permet d'épargner du temps de calcul sur des agrégats qui s'avèrent sans intérêt, et, par conséquent, réduit les coûts associé à notre travail d'exploration de requêtes d'agrégation.

Nous avons mis en oeuvre cette approche dans le cadre d'une plateforme complète. Des expériences utilisant à la fois des graphes réels et synthétiques démontrent sa à trouver des agrégats intéressants dans un grand espace de recherche, l'efficacité de nos algorithmes (dont la performance est meilleure que celle de PostgreSQL pour des tâches où les systèmes sont comparables), et étudié leur évaluation lorsque la taille et la complexité des données augmentent.

PUBLICATIONS

This PhD thesis is based on my research work done under the supervision of Ioana Manolescu and Yanlei Diao and in collaboration with François Goasdoué, a professor at Université de Rennes, and Mirjana Mazuran, a Marie Skłodowska-Curie fellow in the CEDAR team from July 2018 to October 2020. The resulting scientific articles were published in international venues and also (for some) informally presented at BDA, the annual French national data management conference.

The research on the RDFQuotient project lead to three publications during my PhD thesis: [P1] (a demo paper), [P2] (a workshop paper), [P3] (a journal paper).

The subsequent work on the Spade project resulted in two additional publications: [P4] (a demo paper), and [P5] (a full-length conference paper).

Furthermore, ideas, text, and figures included in this thesis have appeared previously in the abovementioned publications:

[P1] François Goasdoué, Paweł Guzewicz, and Ioana Manolescu. "Incremental structural summarization of RDF graphs." In: *EDBT 2019 - 22nd International Conference on Extending Database Technology*. Lisbon, Portugal, Mar. 2019. URL: https://hal.inria.fr/hal-01978784.

[P2] Paweł Guzewicz and Ioana Manolescu. "Parallel Quotient Summarization of RDF Graphs." In: *SBD 2019 - International Workshop on Semantic Big Data*. Amsterdam, Netherlands, June 2019. DOI: 10.1145/3323878.3325809. URL: https://hal.inria.fr/hal-02106521.

[P3] François Goasdoué, Paweł Guzewicz, and Ioana Manolescu. "RDF graph summarization for first-sight structure discovery." In: *The VLDB Journal* 29.5 (Apr. 2020), pp. 1191–1218. DOI: 10.1007/s00778-020-00611-y. URL: https://hal.inria.fr/hal-02530206.

[P4] Yanlei Diao, Paweł Guzewicz, Ioana Manolescu, and Mirjana Mazuran. "Spade: A Modular Framework for Analytical Exploration of RDF Graphs (demonstration)." In: *PVLDB*. Also informally presented at BDA 2019, where it received the Best Demonstration Award. Los Angeles, California, USA: VLDB Endowment, Aug. 2019, pp. 1926–1929. DOI: 10.14778/3352063.3352101. URL: https://hal.inria.fr/hal-02152844.

[P5] Yanlei Diao, Paweł Guzewicz, Ioana Manolescu, and Mirjana Mazuran. "Efficient Exploration of Interesting Aggregates in RDF Graphs." In: *SIGMOD*. Association for Computing Machinery, June 2021. DOI: 10.1145/3448016.3457307. URL: https://arxiv.org/abs/2103.17178.

Concerning the RDFQuotient project, prior to the beginning of my thesis, the novel summaries $\mathtt{G}_{/\mathtt{W}}$ and $\mathtt{G}_{/\mathtt{TW}}$ were (informally) presented in a short "work in progress" paper [1], with procedural definitions (not as quotients). The preliminary work on RDF quotient summaries [2] was also demonstrated. The study of the interplay between summarization and RDF graph saturation was introduced in [3] (poster), without the sufficient condition for the shortcut procedure. I joined the project in the moment of its development that is best captured in the technical report (version 5) [4], where the formal shortcut results were stated for the first time. Before the beginning of the PhD program, during my master's end-of-study research internship, I co-authored the publication [5] (a workshop paper) on type hierarchies-based summaries; this piece of work is pertinent yet orthogonal to RDFQuotient, and it is not included in this thesis.

Regarding Spade, before my joining of the project, the line of research has been started with the Dagger system [6]. The name stands for "**D**igging for interesting **agg**regates in **R**DF graphs"; hence, the name of its successor, Spade, refers to "a better tool[1] for digging for RDF aggregates". Dagger has later been extended using sampling to Dagger+ [7], the work that is orthogonal to Spade.

1 Pun intended.

CONTENTS

1

INTRODUCTION

1.1 MOTIVATION AND OUTLINE

The Semantic Web is a vision of the Web as a vast repository of linked data sources; this vision has been launched and is still actively promoted by the World Wide Web Consortium (W3C, in short). By making data sources open and easy for automatic analysis by computers, the Semantic Web facilitates building data-driven applications. It also encourages the integration of data sources within larger ones and the development of online data repositories.

At the core of the W3C stack of standards and technologies lies the Resource Description Framework (RDF) [8], a graph-structured format for describing data. In an RDF graph, we describe resources (RDF nodes) by specifying the values of the properties that they might have. The basic unit of information in an RDF graph is thus a *triple* of the form `s p o`, stating that the *subject* `s` (an RDF node) has a *property* `p` (a graph edge) whose value is the *object* `o` (an RDF node).

The RDF standard itself is flexible and allows data sources to specify the properties that a node may or should have. If an RDF happens to have a given property, the property may take one or several distinct values. A special property identified in the RDF standard is the *resource type*; it is useful because it carries semantic information about the class of the typed resource. In addition, nodes may have no types or multiple types; those may be related, e.g., through a type hierarchy.

We may enrich the semantics of RDF graphs using an ontology, which captures the semantics of an application domain via classes, properties, and rules stating the relationship between them. Several ontology languages have been proposed; in particular, RDF Schema (RDFS), which is relatively simple, or OWL, which is significantly more complex.

We can query RDF graphs with the help of the standard SPARQL query language [9]. The elementary building block of a SPARQL query is a graph pattern, supporting a wide range of operations, such as selections, projections, joins (between triples having a common subject or object), and filtering. Moreover, the latest SPARQL version (SPARQL 1.1) also supports flexible querying through regular path expressions, as well as group-by and aggregation primitives. The latter

enables it to express *aggregate queries*, similar in spirit to their SQL counterparts, even if their semantics on heterogeneous data graphs is more involved [10], [11].

Heretofore, we have recalled the richness and flexibility of RDF graphs that offer new research opportunities. We now move forward to explore big *graph* data.

Motivating application. Computational Lead Finding (CLF) [12], [13] is one of the target applications of our work. For journalists, a "lead" is an idea based on which they may write an interesting article. Given a dataset, CLF aims to automatically identify the interesting leads from the data. Such tasks still pose immense challenges as, nowadays, large and complex data graphs surpass human cognitive capacities and are hard to exploit for users without automatic tools.

To address these problems, my thesis work makes contributions in two directions, aiming at facilitating the task of human users discovering the content of a graph.

RDF GRAPH SUMMARIZATION: RDFQUOTIENT Our first goal then is to get the "feel of the data". To lower the cognitive burden on users that have to comprehend the content of an RDF graph, many RDF graph summarization techniques have been proposed and described, for instance, in the recent survey [14]. Each summary is a compact representation of (some aspects of) the given RDF graphs. Some of these summaries mainly aim at summarizing the graph structure; others focus on finding frequent structural patterns (subgraphs); some target RDF graphs themselves, whereas others focus on summarizing very large ontologies (which are described as RDF graphs themselves). In this thesis, we study summaries of RDF graphs, which provide small synopses of the *structure* of an RDF graph. We target our summaries for first-time casual users willing to explore intricate RDF data through visualization. To this end, we provide efficient summarization algorithms and implement them in our RDFQuotient tool.

As a brief view of the type of results achieved in the thesis, Figure 1 shows a summary computed automatically, out of a large graph of DBLP bibliographic data, by the RDFQuotient tool which we will describe. The summary visualization is orders of magnitude smaller than that of the input graph and resembles an ER diagram. The summary drawing gives an abstract overview of the graph that is both meaningful and compact. Looking at such a summary, the user can quickly understand the structure of the underlying RDF graph. The summary is a directed graph itself that provides insight into the modeling of the data. Each of its nodes *represents* a set of resources (subjects in the RDF graph); for instance, N13 and N5 represent a set of nodes that have the types Article and Book, respectively. Summary edges reflect connections between nodes from the original graph represented by various summary nodes. For instance, edges labeled *references* and

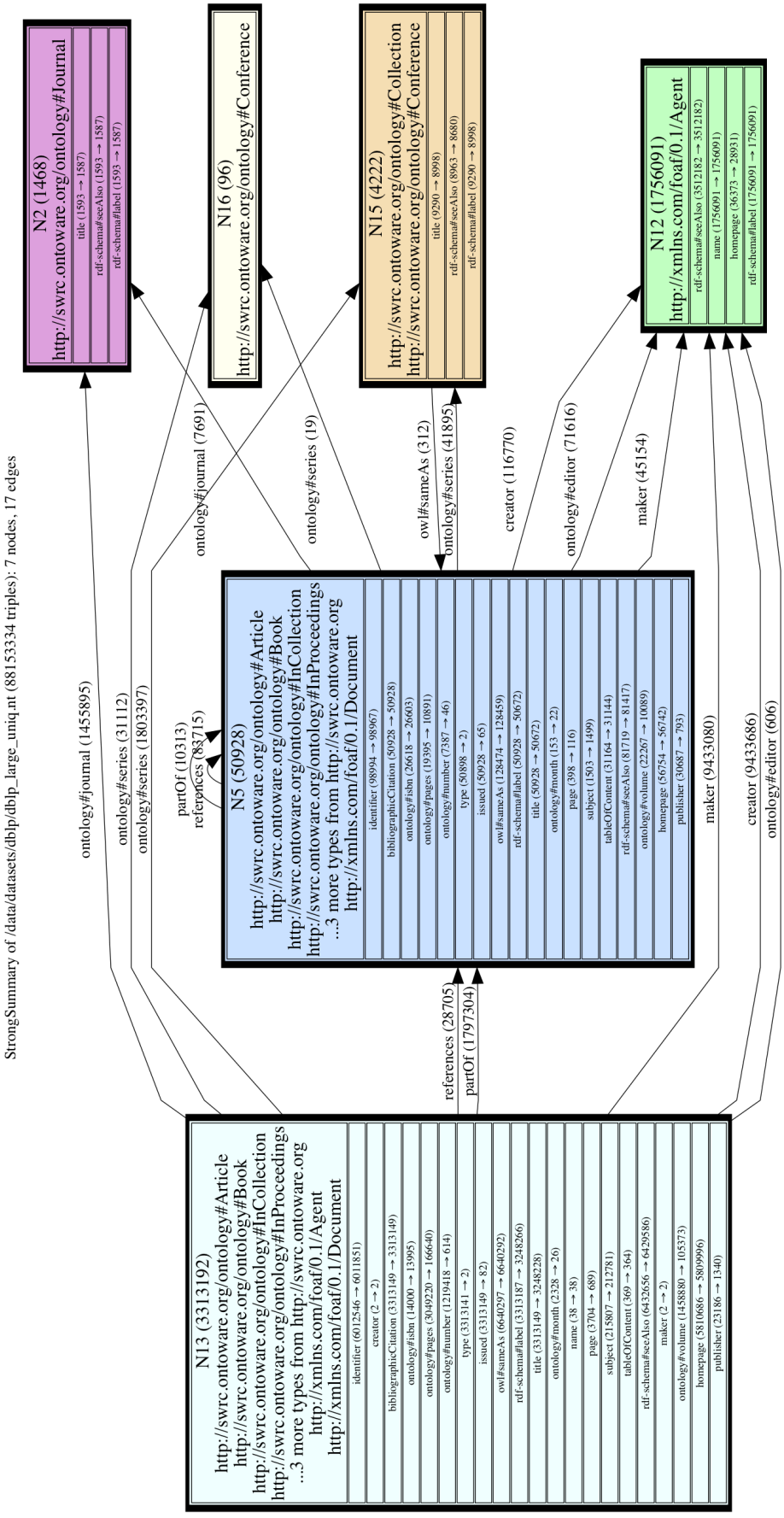


Figure 1: Strong summary of the DBLP graph computed by RDFQuotient.

partOf connect N13 to N5; similar edges connect to other summary nodes, e.g., N2 (journal articles), or N16 (conference articles). We display each such summary node in a box, where the header of the table contains the types of the input nodes that it summarized, and the table body shows its outgoing properties that go to leaf nodes in the graph. We observe that there are only a handful of different "logical" concepts captured in DBLP such as articles/books/documents, journals, conferences, collections of conference venues, as well as agents: this is useful for beginning a query-driven RDF graph exploration. Further, we can tell that the vast majority of the graph contents concentrates on broadly-understood scientific publications, and the frequencies of the nodes and edges (in parentheses) give us a hint about the properties and their co-occurrence on RDF nodes. We can also examine, at first glance, various connections (edges) between the main concepts such as publications having their editors. We delegate a more detailed explanation of how to interpret such a summary visualization, including the numbers in parentheses in the table entries, to Chapter 3.

EXPLORATORY ANALYTICS ON RDF GRAPHS: SPADE The second paradigm for information discovery in RDF graphs we study in this thesis consists of finding interesting insights in the form of multidimensional RDF aggregate queries on the graph. Specifically, we target aggregates whose result exhibits deviation from the prior knowledge, e.g., outliers, that makes them worthy of the user's attention. For instance, a data analyst or a journalist may probe the data with further queries to learn more about the interesting aggregate. We developed a comprehensive method for automatically and efficiently identifying such interesting aggregates given an RDF graph. Using our method, we can enumerate a large space of candidate multidimensional aggregates, efficiently evaluate them, and rank them based on their interestingness. We implemented this approach within Spade, an innovative system that advances the state of the art of Online Analytical Processing (OLAP) systems for RDF graphs.

Furthermore, Spade redefines core data warehouse concepts to adapt them to the specificities of the RDF data model, in that:

1. We aim at novice users who do not need to possess technical skills;

2. We leverage a quotient summary of an RDF graph to identify an analytical schema and determine a fact set to build aggregate queries on;

3. To further enlarge the space of multidimensional aggregate queries we explore, we enrich our queries by following graph paths and deriving new properties from text values, such as the language or keywords present in the literals (string constant RDF nodes);

4. We analyze the graph properties to determine the most useful dimensions and measures, and build aggregate lattices;

5. To surpass the limitations of current SPARQL processing engines, we design MVDCube, an efficient aggregate lattice evaluation algorithm based on a relational data warehouse counterpart. It is the first known *correct* algorithm for this task in the particular setting of heterogeneous RDF graphs. As we will show in Chapter 4, relational algorithms do not apply as they may introduce significant errors;

6. We further speed up the Spade computation using our novel early-stop technique for pruning unpromising aggregates;

7. Finally, we guide the search for the most interesting aggregates based on an interestingness function score.

For example, Figure 2 shows an interesting aggregate found by Spade in the DBLP dataset. The plot depicts the distribution of each keyword found in the DBLP publications' titles over the years. In this heat map, the redder the color, the more the articles in an aggregate group. From this visualization, we can learn that some core concepts, such as logic, system, and theory, have been present in the DBLP content since its earliest years (1936 in this plot). We also observe other insights provided by the same plot. For instance:

- Systems and networks have strong visibility in recent years; and
- The Web has become a topic of interest in the 1990s.

Crucially, the insights were found automatically in a large space of aggregates.

To conclude, both Figures 1 and 2 exemplify RDFQuotient and Spade's approaches to offer new insights into RDF graphs that we are only able to find using our new automatic techniques and which drastically enhance our understanding of the underlying data. We stress here that these are otherwise hard to find for users due to the complexity and abundant volumes of present-day heterogeneous RDF graphs.

1.2 MANUSCRIPT ORGANIZATION

The sequel of the thesis is divided into four chapters.

In Chapter 2, we first introduce basic notation used throughout the entire manuscript. We then present the key RDF concepts relevant to this work, focusing on data graph summarization. We also discuss existing aggregate lattice evaluation algorithms. Subsequently, we position our work w.r.t. main prior work in the areas of graph summarization and relational data warehouses.

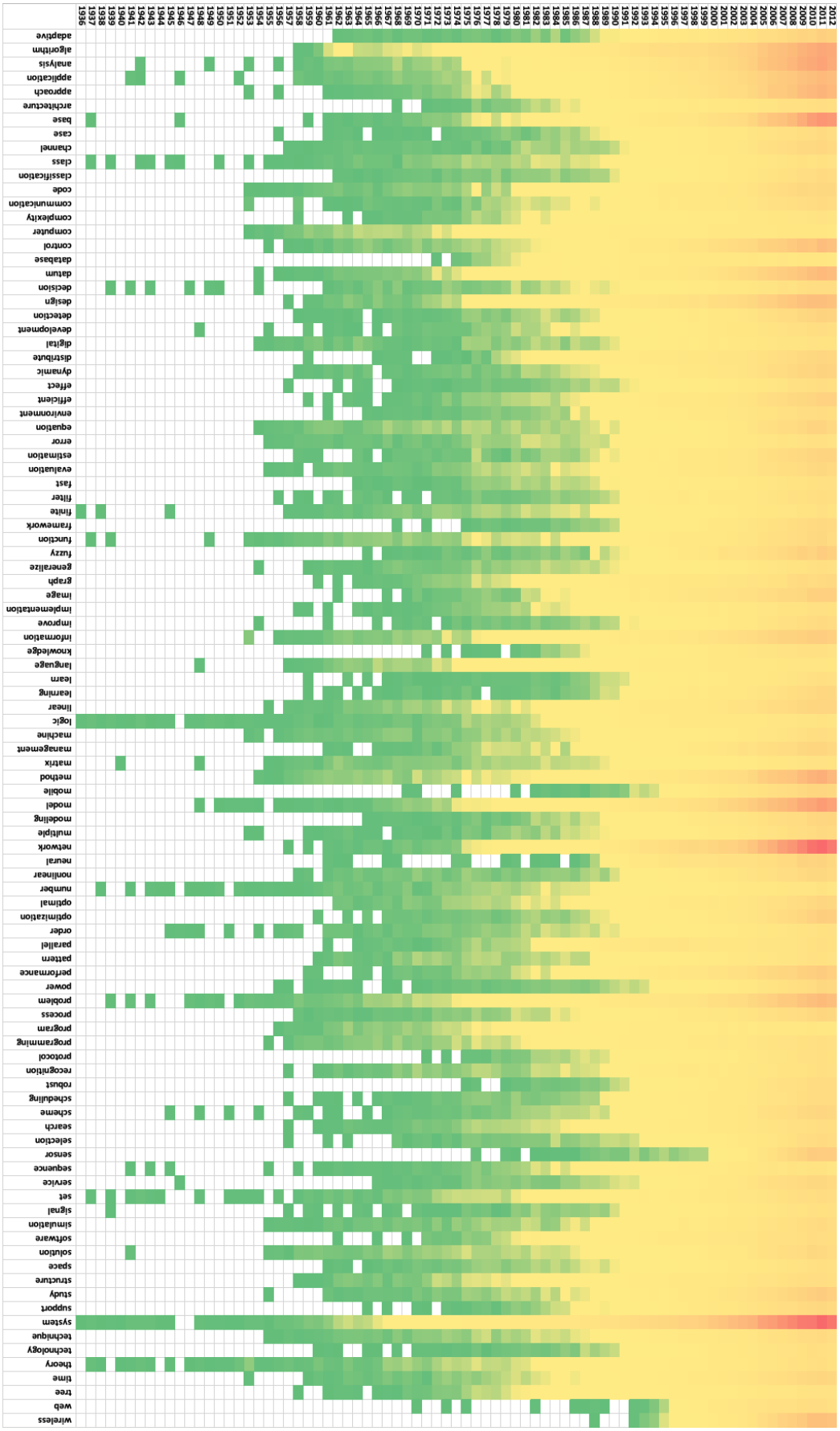


Figure 2: Sample interesting two-dimensional aggregate recommended by Spade: distribution of keywords in DBLP publication titles across the years.

Next, in Chapter 3, we show the limitations of the existing approaches and devise new RDF graph summaries to enable first-sight structure discovery in RDF graphs. We explain the foundation of our RDFQuotient tool for graph summarization and give a thorough study of quotient summaries, including our four novel summaries: weak, strong, typed weak, and typed strong. We also explore the semantics of RDF graphs based on ontologies in the summarization context. We exploit the ontology of an RDF graph to enrich the quotient RDF summary and speed up its computation based on our shortcut procedure that allows us to avoid expensive graph saturation. We describe our novel centralized and parallel algorithms, study their performance in our experiments, and demonstrate their merit.

Building on our summarization framework, in Chapter 4, we present our new Spade system that helps users find the most interesting aggregates in an RDF graph automatically. We show that Spade benefits from the summaries built by RDFQuotient to guide the automatic exploration of a large and rich space of multidimensional aggregates of an RDF graph. We provide new techniques for aggregate lattice evaluation, similar to the CUBE operator from relational data warehouses, yet crucially computing the correct results on RDF (which is not the case for existing relational tools). Finally, in our experimental evaluation, we prove both the efficiency and scalability of our MVDCube algorithm, with and without the early-stop technique.

We conclude the work that spans this thesis in Chapter 5.

ACKNOWLEDGMENTS Paweł Guzewicz is supported by the Agence Nationale de la Recherche under the ARN grant "A New Database Service for Interactive Exploration on Big Data – AIDE" (ANR-16-CE23-0010-01) and the European Research Council, H2020 research program under the ERC grant "BigFastData" (ERC 725561).

2

PRELIMINARIES

We recall here the starting points of this work: RDF graphs (Section 2.1), graph quotients (Section 3.2), SPARQL queries (Section 2.2), and relational data warehouses and aggregation (Section 2.3).

2.1 DATA GRAPHS AND THE RESOURCE DESCRIPTION FRAMEWORK

RDF DATA MODEL This work targets *directed graphs with labeled nodes and edges*. In particular, we focus on data graphs described in the Resource Description Framework (RDF) [8], the World Wide Web Consortium (W3C) standard for representing Web data. We consider only well-formed RDF data, as per the RDF specification [8], defined over three pairwise disjoint sets: the set of Uniform Resource Identifiers (URIs) $\mathcal{U}$, the set of typed or untyped literals (constants) $\mathcal{L}$, and the set of blank nodes (unknown URIs or literals) $\mathcal{B}$. Blank nodes are an essential feature of RDF that allows us to support unknown URI or literal tokens. They are conceptually similar to the labeled `nulls` or variables used in incomplete relational databases [15], as shown in [16].

RDF GRAPHS An *RDF graph* is a finite set of *triples* of the form $\mathtt{s}\ \mathtt{p}\ \mathtt{o}$, such that $\mathtt{s} \in (\mathcal{U} \cup \mathcal{B})$, $\mathtt{p} \in \mathcal{U}$, and $\mathtt{o} \in (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$. The triple $\mathtt{s}\ \mathtt{p}\ \mathtt{o}$ states that the *subject* $\mathtt{s}$ has the *property* $\mathtt{p}$, and the value of the property is the *object* $\mathtt{o}$. The RDF property *rdf:type* is used to attach types, i.e., classes, to an RDF node, which may have zero, one, or several types. Note here that an RDF type attached to an RDF node using the *rdf:type* property is an RDF node itself, whereas the types in typed literals are concatenated to their raw constant string. Also, since literals do not appear as subjects of *rdf:type* triples, they cannot be typed via the *rdf:type* property.

As our running example, Figure 3 shows a sample RDF graph $\mathtt{G}$ describing a university department including professors, graduate students, articles they wrote, and courses they teach and/or take. Here and in the sequel, RDF nodes shown in gray are classes (types). Further, RDF nodes whose labels appear enclosed in quotes, e.g., "d1", are literals, whereas the others are URIs.

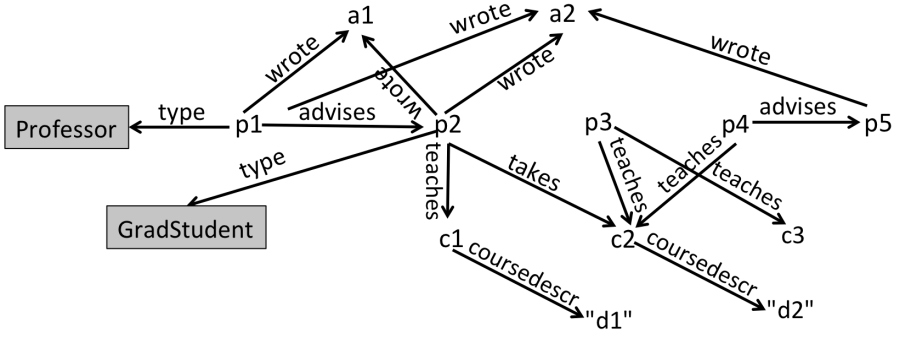


Figure 3: Sample RDF graph.

GradStud	*rdfs:subClassOf*	*Instructor*
Professor	*rdfs:subClassOf*	*Instructor*
takes	*rdfs:domain*	*Student*
takes	*rdfs:range*	*Course*
advises	*rdfs:subPropertyOf*	*knows*

Table 1: Sample ontology.

RDFS ONTOLOGY We can enhance the resource descriptions comprised in an RDF graph by declaring *ontological constraints* stating relationships among its types and properties. For instance, we may add to the graph $\mathtt{G}$ in Figure 3 the ontology O consisting of the triples listed in Table 1 to state that graduate students, respectively, professors are instructors, that anyone who takes a course is a student, what can be taken is a course, and that advising someone entails knowing them.

From ontological constraints and explicit triples, we may derive *implicit* triples. For instance, from $\mathtt{G}$ and O, it follows that p_1 is of type *Instructor*, which is modeled by the implicit triple p_1 *rdf:type Instructor*; we say this triple *holds in* $\mathtt{G}$, even though it is not explicitly part of it. Other implicit triples obtained from this ontology based on $\mathtt{G}$ are: p_2 *rdf:type Instructor*, p_2 *rdf:type Student*, and p_1 *knows* p_2.

Given a graph $\mathtt{G}$ (which may include a set of ontological constraints, denoted O), we obtain the *saturation* of $\mathtt{G}$ by adding to $\mathtt{G}$:

1. All the implicit triples derived from $\mathtt{G}$ and O, then
2. All the implicit triples derived from those in Step 1 and O, and so on

until a fixed point, denoted $\mathtt{G}^{\infty}$, is reached.

We add these triples based on *RDF entailment rules* from the RDF standard. In this work, we consider the widely-used *RDF Schema (RDFS) entailment rules*. They exploit the simple RDFS ontology language based on the four standard properties illustrated above, which we denote *subClass*, *subProperty*, *domain* and *range*, to add new ontological constraints or facts. The saturation $\mathtt{G}^{\infty}$ of a graph $\mathtt{G}$ comprising

RDFS ontological constraints is finite [17], unique and can be computed in polynomial time, e.g., by leveraging a database management system [16]. Crucially, $\mathtt{G}^{\infty}$ materializes the semantics of G.

TERMINOLOGY We call the triples from G whose property is *rdf:type* *type triples*, those whose property is among the standard four RDFS ones *schema triples*, and we call all the other *data triples*.

We say an RDF node is *typed* in G if the node is the subject of at least one type triple in G. In the graph shown in Figure 3, p_1 *rdf:type* $Professor$ is a type triple, hence the node p_1 is typed, $Professor$ *rdfs:subClassOf* $Instructor$ is a schema triple, whereas p_1 *advises* p_2 is a data triple.

Further, we say a URI from G is a *class node* if:

1. It appears as the subject or the object of an *rdfs:subClassOf* triple, or the object of an *rdfs:domain* triple or an *rdfs:range* triple; or
2. It appears as the object of an *rdf:type* triple; or
3. It appears as the subject of an *rdf:type* triple with the object *rdfs:Class*.

We call *property node* a URI appearing:

1. As the subject or the object of an *rdfs:subPropertyOf* triple, or as the subject of an *rdfs:domain* triple or an *rdfs:range* triple; or
2. As the subject of an *rdf:type* triple with object *rdf:Property*.

Together, the class and property nodes are the *schema nodes*; all non-schema nodes are *data nodes*. In Figure 3, Professor and GradStudent are class nodes. If we consider the aforementioned ontology O, *takes*, *advises*, and *knows* are property nodes. Finally, the a, p, c, and d RDF nodes are data nodes.

It is important to stress that not all nodes in an RDF graph are typed, e.g., this is only true for p_1 and p_2 in Figure 3. Further, some nodes may have several types, in particular due to saturation (e.g., p_2 is of types GradStudent and Instructor) but not only, e.g., p_1 could also be of type ForeignStudent, etc.

2.2 QUERIES ON RDF GRAPHS

We can query RDF graphs by means of SPARQL queries as defined in [9]. SPARQL is a declarative query language similar to SQL. A SPARQL query contains a graph pattern that is used for data selection. In this thesis, we focus on basic graph patterns (BGPs), i.e., sets of triple patterns. A query processing engine matches triples of a BGP against G. Each triple of the BGP may contain values (RDF nodes) or variables (strings preceded with the ? character). For example, the

query below finds all instances of the Professor class for the graph in Figure 3.

```
SELECT ?instance {
    ?instance rdf:type Professor .
}
```

The result of the query is $\{(p_1)\}$.

SPARQL queries may contain a prefix, which allows us to abbreviate URIs. For instance, common abbreviations recognized by SPARQL query processing engines, which are consistent with those we introduced in Section 2.1, are:

```
rdf   http://www.w3.org/1999/02/22-rdf-syntax-ns#
rdfs  http://www.w3.org/2000/01/rdf-schema#
```

Using SPARQL, we can express more sophisticated queries such as aggregates. The following query finds the number of courses per instructor for the graph in Figure 3.

```
PREFIX re: <http://running.example/>
SELECT ?instructor COUNT(?course) AS ?numberOfCourses {
    ?instructor re:teaches ?course .
}
GROUP BY ?course
```

The result of the query is $\{(p_2, 1), (p_3, 2), (p_4, 1)\}$. Note here that the results does not contain p_1 because this RDF node does not have a value for the property *teaches*.

Hereafter, when we discuss queries issued against an RDF graph $\mathtt{G}$, we refer to the semantics of SPARQL queries. As hinted in Section 2.1, a query processing engine assumes that the semantics of $\mathtt{G}$ (the dataset) is its saturation $\mathtt{G}^\infty$ (materialized or not). For instance, consider the graph $\mathtt{G}$ in Figure 3 with the ontology O stated in Table 1 and the following query Q:

```
SELECT ?instance {
    ?instance rdf:type Instructor .
}
```

The result of $Q(\mathtt{G})$ is empty, whereas the result of $Q(\mathtt{G}^\infty)$ returns nodes p_1 and p_2. Notice how this time, through the type inference, we find that there are two instructors. Interestingly, neither of the two queries about instructors finds all the p RDF nodes ($\{p_1, p_2, p_3, p_4\}$): we will explore this observation in Section 3.3.

Similarly, given that $\mathtt{G}^\infty$ contains more data, its summary may be different from the one of the original graph $\mathtt{G}$ (more on this also in Chapter 3). Therefore, from now on, we consider SPARQL queries issued against $\mathtt{G}$ as queries on $\mathtt{G}^\infty$ (unless explicitly stated otherwise).

2.3 RELATIONAL DATA WAREHOUSES AND AGGREGATION

We recall here a set of classical notions from the relational data management literature: Online Analytical Processing, aggregate queries, and aggregate lattices. Then, we recall one of the most efficient algorithms introduced in this relational setting for efficiently evaluating lattices of aggregate queries over large volumes of data that provides an efficient way of computing the result of a GROUP BY CUBE query over a multidimensional aggregate lattice. Even though this algorithm does not apply to the heterogeneous RDF graphs we consider in this thesis, we build upon it to devise our novel algorithm specifically for this setting (in Chapter 4).

ONLINE ANALYTICAL PROCESSING (OLAP) Online analytical processing (OLAP, in short) is a major class of applications of structured database management systems, and it has contributed to making data management the multibillion-dollar industry it is nowadays. OLAP systems enable businesses to derive critical insights from old and new data characterizing the functioning of a real-life process by efficiently evaluating *aggregate* queries that compute a global view from millions of individual data points.

In a typical application, managers of a retail chain use OLAP to synthesize and then analyze the sales of various products in various locations, stores over different periods. In an academic setting, we may use OLAP to analyze students' registration to individual courses, compare average grades across disciplines, etc.

FACTS, DIMENSIONS, AND MEASURES OLAP facilitates multidimensional analytics and is typically backed by a relational *data warehouse* (DW) [18]. Such a warehouse is a specially organized database, which features one or more fact tables. A fact table states *facts*, i.e., raw information about a specific topic, e.g., "a client buys a product in a given store at a given date and time", or "a student takes a course in a certain term in a certain year". Each fact is uniquely identified by a combination of *dimensions*, e.g., the client and the product in the former example, or the student, the course, the year, and the professor teaching the course that year, in the latter example.

Further, to each fact, there may be associated one or several *measures*, which depends on the aggregate query asked over the data warehouse. For instance, if the retail manager is interested in following product sales, the number of products bought in every sale is a good measure; instead, if we examine the financial aspect, the price at which the sale took place is the right measure to use. In the academic example, if we follow the attendance of a course, then each fact simply counts as 1; instead, if we study the average grades, the grade obtained by each student in each course is the good measure to use.

Last but not least, an aggregate query uses an *aggregate function* over the measure values corresponding to all the facts. In our example, one can *count* the sales or the students; *sum* the numbers of sold products or the number of student registrations across several years; *average* the grades of all students who took the same course at the same time, or even find the *maximum/minimum* grade given each year, etc.

MULTIDIMENSIONAL AGGREGATE LATTICE Experience has shown that the OLAP analysis needed by executives is rarely satisfied by running one aggregate query. Instead, we need many queries to find interesting insights into the data. Further, many aggregate queries that are asked on a given data warehouse have things in common. For instance, one may require the GPA of students in a course *by year when the course is given*, or *by year and student gender* (to see if there is a significant difference in performance by gender), or *by year and teaching assistant* if we want to see whether there are strong differences across the different lab groups, etc.

To this end, as one of its core data structures, OLAP employs a multidimensional data array called OLAP cube. The OLAP cube spans the data on its multiple dimensions and stores, in each multidimensional cell, a measure value. Typically, relational DWs organize the data in a star schema or a snowflake schema (a normalized star schema) with one or several fact tables. The performance of OLAP depends both on the data layout and the query evaluation algorithms. The latter may lead to expensive computation; thus, their design accounts for data volume and data complexity (in particular, the number of dimensions). Generally speaking, in a data warehouse where N dimensions have been identified, and if one fixes a measure and an aggregate function, given that each subset of dimension can be used for aggregation, there are 2^N possible ways to group the facts, thus, 2^N aggregates to compute.

An important insight that has revolutionized the efficiency of OLAP processing is the following. Let S_1 be a set of one or more dimensions, and $S_2 \supset S_1$ be a dimension set that is its strictly larger superset. Then:

- Any group of facts in the aggregate query determined by the dimension set S_1 is completely included in a group of facts in the aggregate query determined by S_2; this suggests reusing the effort needed to group facts on S_2 to group by S_1 at a lower computational cost;

- If the aggregate function is *sum*, *count*, *min*, or *max*, we can compute the aggregate function results for the aggregate query corresponding to S_1, *directly* from the results corresponding to S_2, with no need to consult the base data (the facts themselves) or their measure values; in the case of *avg*, we can reconstruct the aggregated result based on *sum* and *count*.

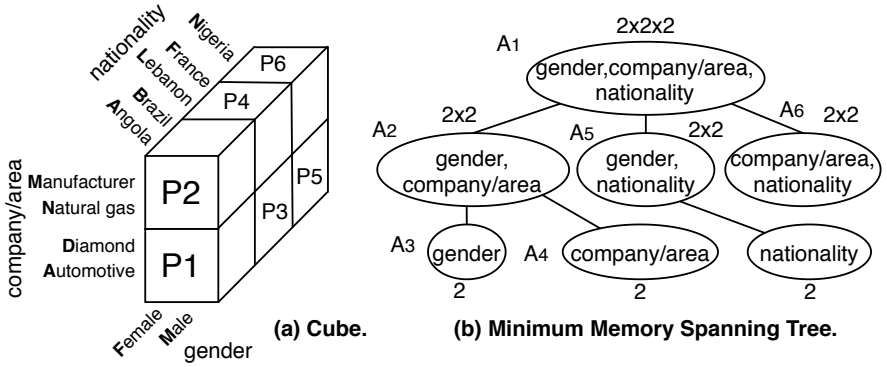


Figure 4: Multidimensional space and MMST for Example 3.

This has lead to the introduction of a so-called *multidimensional aggregate lattice* that compactly represents the 2^N aggregates that can be obtained from N independent dimensions, and also captures how each of them can be computed based on the results of the other. The node at the bottom of the lattice (usually not shown) corresponds to the empty set of dimensions: data is not aggregated at all (this node corresponds to the raw set of facts). At the top of the lattice, data is aggregated by all the N dimensions. At each intermediary level, we have nodes whose parents in the lattice have one more dimension, and whose children have one dimension less.

CLASSICAL ONE-PASS LATTICE COMPUTATION For efficiency, in relational DWs, nodes in a multidimensional lattice are often computed from one of their parents to reuse computation and limit the number of passes over the data. In some works, the reusing of the other aggregates' results is sometimes referred to as "summarizability" [19], [20]. Among the existing algorithms [21], ArrayCube [22] computes the whole lattice in a single pass over the data. Given a set of N dimensions, a measure, and an aggregate function, it relies on an *array representation* of data and evaluates 2^N nodes through a *Minimum Memory Spanning Tree*, as we recall below.

ARRAY REPRESENTATION OF DATA The distinct values of each dimension are *ordered*, leading to a set of *cells*, each corresponding to a unique combination of indices of values along the N dimensions (axes). In Example 3, assuming *nationality* $\in$ {A, B, F, L, N}, *gender* $\in$ {F, M} and *company/area* $\in$ {A, D, M, N} (we denote initials of the respective values in Figure 4), the multidimensional space has 40 cells, e.g., in cell 0, *nationality*=A, *gender*=F and *company/area*=A; in cell 1, *nationality*=B, *gender*=F and *company/area*=A. Each cell of the N-dimensional array contains the value of the aggregated measure over all facts in that cell; in Example 3, this is the count of CEOs. Further, cells are grouped in *partitions*: each partition is a contiguous part of the array, containing

A_1: count of CEOs by nationality, gender, company/area

tID	nationality	gender	company/area	count(*)
t1	Angola	Female	Diamond	1
t2	Angola	Female	Manufacturer	1
t3	Angola	Female	Natural gas	1
t4	Nigeria	null	Automotive	1
t5	Nigeria	null	Manufacturer	1
t6	France	null	Automotive	1
t7	France	null	Manufacturer	1
t8	Lebanon	null	Automotive	1
t9	Lebanon	null	Manufacturer	1
t10	Brazil	null	Automotive	1
t11	Brazil	null	Manufacturer	1

A_2: count of CEOs by gender, company/area

tID	gender	company/area	count(*)
t1	Female	Diamond	1
t2	Female	Manufacturer	1
t3	Female	Natural gas	1
t4	null	Automotive	4
t5	null	Manufacturer	4

A_4: count of CEOs by company/area

tID	company/area	count(*)
t1	Diamond	1
t2	Manufacturer	5
t3	Natural gas	1
t4	Automotive	4

tID	gender	count(*)
t1	Female	3

A_3: count of CEOs by gender

Figure 5: Relational aggregation.

the cells corresponding to a predefined number of distinct values along each dimension, e.g., if this is 2, the 40-cell array has 6 partitions. Figure 4(a) shows the array. Note that an initial pass over the data is required to bring it from the relational to the array representation.

MINIMUM MEMORY SPANNING TREE (MMST) The dimensions in Example 3 determine the lattice in Figure 4(b). To evaluate all nodes, ArrayCube chooses, for each non-root node A, a parent node in the lattice to compute the aggregate in A from, hence forming a spanning tree covering all the nodes in the lattice. The memory needed to evaluate all the aggregates in one pass over the data depends on the ordering of dimensions, their numbers of distinct values, and the partition size. ArrayCube chooses the tree that minimizes the overall memory needed; this is called the Minimum Memory Spanning Tree, or MMST in short.

LATTICE COMPUTATION The algorithm proceeds as follows. The MMST is instantiated, allocating to each node the required memory. Partitions are loaded from the array representation of data, one at a time, into the root of the MMST. The content of each cell in the root is propagated to the children and used to incrementally update the aggregated measures of all the nodes in the MMST. Once a partition is evaluated, *each node checks if it is time to store its memory content to disk*. For instance, after scanning partition P1 in Figure 4(a), the subarray with *nationality* $\in$ {A, B} and *company/area* $\in$ {A, D} is exhausted. Thus, the counts of CEOs with either of the two nationalities, and A or D company area are computed. Now, A_6 (Figure 4(b)) can store its result to disk and reuse the memory in the subsequent computation. Similarly, once processed, the two subarrays of P2, *nationality* $\in$ {A, B}, and *company/area* $\in$ {M, N}; *nationality* $\in$ {A, B}, are exhausted, and both A_6 and A_5 can store their results to disk. A_6 stores its result

to disk after every partition, whereas A_5 does this after every two partitions.

3

RDF GRAPH SUMMARIZATION

In this chapter, we introduce RDF graph summarization techniques. We begin by describing the motivation for summarizing RDF graphs, accompanied by an example visualization of one of our novel summaries. We then explain in detail our new approach and validate it with experiments.

3.1 MOTIVATION

RDF graphs enable describing large and heterogeneous datasets. Yet, their inherent structural heterogeneity makes it hard for the casual users to get acquainted with a graph's structure. To address this problem, many RDF *summarization* techniques have been proposed in the literature [14], some of which draw upon graph summarization techniques [23] proposed independently of RDF. Each of them builds, out of a given RDF graph, a compact structure that conveys the essential information of the graph, all that while being much more compact.

As stated in [14], RDF summarization techniques fall into four classes:

1. Structural methods are built considering first and foremost the graph structure, respectively, the paths and subgraphs present in the graph;

2. Pattern mining methods apply mining techniques to discover patterns in the data and use the patterns as a summary (synthesis) of the graph;

3. Statistical methods aim at extracting from the graph a set of quantitative measures or statistics; finally

4. Hybrid methods combine elements from more than one of the previous classes.

For what concerns summary applications, these range from (RDF) graph indexing, through query cardinality estimation, to helping users formulate graph queries, graph visualization, and exploration.

A large and useful class of structural graph summaries are based on defining an *equivalence* relation among graph nodes, and creating one summary node for each equivalence class (a set of nodes equivalent to each other in the original graph). Then, for every edge labeled p which goes from s to o in the graph, the summary has an edge labeled p from the summary node corresponding to the equivalence class of s, to the node corresponding to the equivalence class of p. Such summaries, also called *quotient summaries*, have many good properties, mainly due to the existence of a graph homomorphism from the original graph into its summary. Quotient summaries proposed for general graphs include [24], [25], [26], [27], [28], [29].

Summarizing RDF graphs raises two new questions w.r.t. to prior (non-RDF) graph summarization setting:

1. How to take into account the types that may be attached to the nodes (knowing that a node may have no type, or one type, or several)? On the one hand, types bring an opportunity to define node equivalence, since, intuitively, two nodes having the same type(s) are likely similar in sole way. On the other hand, they cannot be solely relied upon, because many RDF graphs lack types for many (or all) of their nodes; prior RDF summary quotients [30], [31], [32] answer this question in different ways;

2. How should a summary reflect the implicit triples that may hold in the graph due to the presence of an ontology? The prior poster paper [3] is the only work to have addressed this so far.

In this work, we make several theoretical and practical contributions to the area of quotient RDF graph summarization. Specifically:

1. We rely on the formal *RDF quotient summarization framework* [3], taking into account RDF-specific concepts such as ontologies.

2. We introduce *two novel equivalence relations* between RDF nodes, which rely on the transitive co-occurrence of properties on graph nodes. Based on them, we define *two novel summaries called weak and strong*, respectively, as well as two variants thereof that give priorities to types (for those nodes that have type information); we call these summaries *typed weak and typed strong*, respectively. The interest of these new equivalence relations is that they lead to summaries that are much more compact (that contain fewer nodes and edges) than quotient summaries previously studied in the literature [24], [25], [26], [27], [28], [29], [30], [31], [32]. This compactness comes at the price of some loss of accuracy. Nevertheless, they do preserve a significant amount of information from the input graph. In particular, for domain-specific graphs, describing applications from a specific area, our summaries are very convenient data discovery tools: a simple summary visualization helps understand the graph structure. This is why *this*

work's main target are domain-specific graphs. For encyclopedic graphs, such as DBPedia or YAGO, our quotients are very likely to be more compact than some studied in prior work, but still too large for human comprehension; non-quotient summaries, e.g., based on pattern mining [14], are more appropriate for such graphs.

3. In this thesis, we recall the results obtained in the scope of the RDFQuotient project to show that for a large set of RDF equivalence relations, one can build their corresponding quotient summary of an RDF graph *including its implicit triples, without materializing them*. Based on our framework, we elaborate the sufficient condition for an RDF equivalence relation that enables building this through our so-called *shortcut* method; its advantage is to reduce very significantly the summarization time. We state that *our weak and strong summaries satisfy this condition, whereas the typed weak and typed strong ones do not*. We include (without proofs) a set of similar results also on previously studied equivalence relations.

4. Our next contribution is a set of *novel algorithms for computing our summaries*. We propose three kinds of algorithms: global ones, which are a straight-forward memory-efficient but time-consuming implementation; novel *incremental* ones, which are able to reflect addition of triples to the graph, without re-traversing the rest of the graph; and finally novel *parallel* algorithms dedicated to computational clusters. All our centralized algorithms have *amortized linear complexity in the size of the graph*. We have implemented these algorithms and summary visualizations in the RDFQuotient system, available online in open source [33]. Our parallel algorithms are implemented on top of the Spark framework.

5. Finally, we demonstrate experimentally: the compactness of our summaries, the good performance of our summarization algorithms, and the performance benefits of the shortcut method. Moreover, our experiments confirm that we can benefit from parallelism to speed up execution and surpass the limits of memory-constrained environments in the case of large graphs whose memory needs grow otherwise linearly with the size of the graph.

SAMPLE SUMMARY VISUALIZATION Below, we show an example where our summarization techniques compress an RDF graph structure by many orders of magnitude while still supporting an informative visualization.

Figure 6 shows the summary of a WatDiv [34] benchmark graph of approximately 11 millions of triples. This visualization reflects the

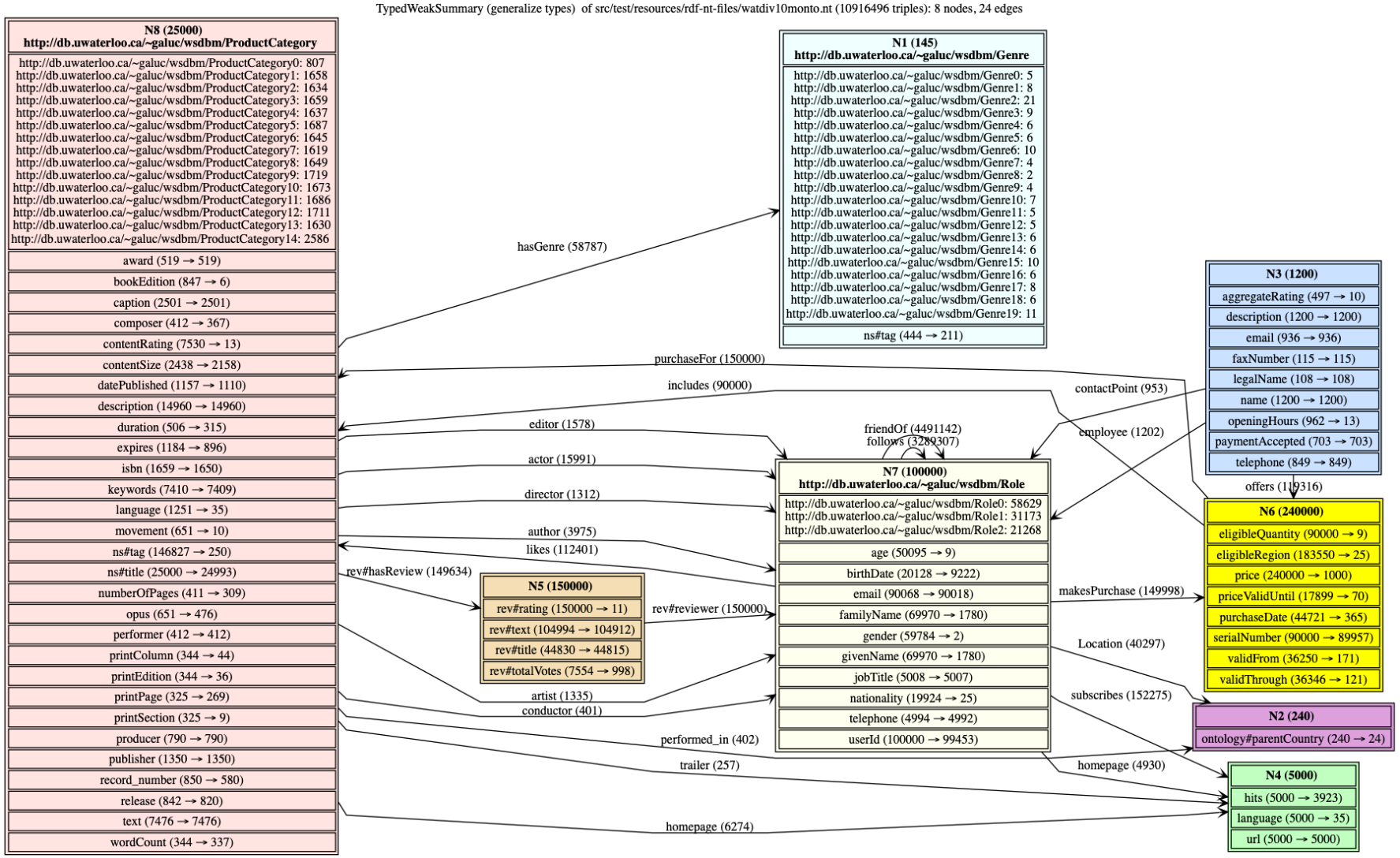


Figure 6: ER-style visualization built from one of our summaries.

complete structure of the graph, using only 8 nodes and 24 edges, comparable to a simple entity-relationship (ER) diagram. This summary reads as follows.

1. Non-leaf graph nodes belong to one of the eight disjoint entities, each represented by a summary node (box in Figure 6) labeled N_1 to N_8. The number of graph nodes in each entity appears in parenthesis after the label N_i of their representative, e.g., $25,000$ for N_8.

2. Each entity represents either graph nodes with types or without types. In the first case, the most general types of the graph nodes, according to which they have been grouped in the entity, are given in bold underneath the entity label; these coincide with the original graph node types when an ontology is not present. For instance, all the 25000 graph nodes represented by N_8 are (implicitly) of type ProductCategory due to the ontology at hand; their distribution according to their original typing is also shown, e.g., 807 are of type ProductCategory0. An entity that does not show types represents untyped graph nodes which have been grouped according to the relationships they have with others, using a novel *transitive* relation of co-occurrence of their properties, which we introduce in this thesis. For instance, N_4 represents both the web pages of the products of N_8 and these of the persons of N_7, because these web pages can be home pages for both: there are *homepage* edges from both N_8 to N_4 and from N_7 to N_4.

3. Graph nodes from an entity may have outgoing properties whose values are *leaf* nodes in the graph; the set of all such properties appears in the corresponding summary node box, one property per line. For each property, e.g., nationality for N_7, the summary node specifies how many graph nodes represented by this entity have it ($19,924$ in this case), and how many distinct leaf nodes are target of these edges (25 in this case).

4. Graph nodes from an entity may have outgoing properties whose values are *non-leaf* nodes in the graph. For each graph edge $n_1 \xrightarrow{a} n_2$, where n_1, n_2 are non-leaf graph nodes and a is the property (edge label), an a-labeled edge in the summary goes from the representative of n_1 to that of n_2. Next to a, that summary edge is also labeled with the number of graph edges to which it corresponds.

5. Properties from a small, fixed vocabulary are considered *metadata* (as opposed to *data*) and therefore are not used to split graph nodes in entities, e.g., *rdfs:comment* and *rdfs:label* in Figure 6.

More such visualization summaries can be found online [33]; an example leading from an RDF graph to its summary and then such

a visualization is worked out in this thesis. Most of the material presented here is new. The exceptions are: Theorem 2 appeared without proof in the poster [3]; our algorithms were outlined in the demonstration [35].

In the sequel of this chapter, in Section 3.2, we begin by outlining the background of the quotient RDF graph summarization. In Section 3.3 we introduce the novel notions of property cliques, based on which we define our summaries. In Section 3.4, and respectively, Section 3.5 we extend this to graphs comprising type triples, respectively, ontologies. In Section 3.6 we discuss summary visualization. In Section 3.7 we describe our summarization algorithms. In Section 3.8 we present our experiments. Finally, in Sections 3.10 and 3.11, we survey related work on non-quotient graph summarization and conclude, respectively.

3.2 BACKGROUND: QUOTIENT RDF GRAPH SUMMARIZATION

We recall here quotient RDF summaries as defined in prior work, outline existing work in this area, and discuss their limitations.

3.2.1 *Quotient graphs*

Given an RDF graph $\mathtt{G}$ and an equivalence relation[1] $\equiv$ over the RDF nodes of $\mathtt{G}$, the quotient of $\mathtt{G}$ by $\equiv$, denoted $\mathtt{G}_{/\equiv}$, is the graph having:

- An RDF node for each equivalence class of $\equiv$ (thus, for each set of equivalent $\mathtt{G}$ nodes); and
- For each edge $n_1 \xrightarrow{a} n_2$ in $\mathtt{G}$, an edge $n_1^{\equiv} \xrightarrow{a} n_2^{\equiv}$, where n_1, n_2 are the quotient summary nodes corresponding to the $\equiv$ equivalence classes of n_1, n_2, respectively (also called *representatives of* n_1 *and* n_2, respectively).

We call *representation function* a function that maps each graph node n to its representative ($n^{\equiv}$).

Quotients have several desirable properties from a summarization perspective:

SIZE GUARANTEES: By definition, $\mathtt{G}_{/\equiv}$ is guaranteed to have at most as many nodes and edges as $\mathtt{G}$. Some non-quotient summaries, e.g., Dataguides [36], cannot guarantee this.

PROPERTY COMPLETENESS: Every property (edge label) from $\mathtt{G}$ is present on some summary edges. This gives first-time users of the dataset a chance to decide, based on their interest and envisioned application, if it is worth further investigation. In some applications, e.g., when data journalists explore open data,

1 An equivalence relation $\equiv$ is a binary relation that is reflexive, i.e., $x \equiv x$, symmetric, i.e., $x \equiv y \Rightarrow y \equiv x$, and transitive, i.e., $x \equiv y$ and $y \equiv z$ implies $x \equiv z$ for any x, y, z.

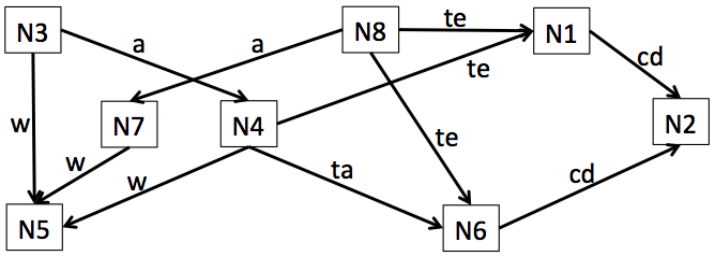


Figure 7: `1fb` summary of the RDF graph in Figure 3.

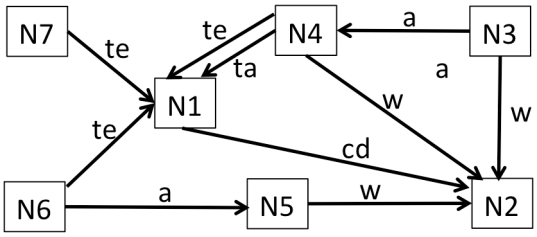


Figure 8: `1fw` summary of the RDF graph in Figure 3.

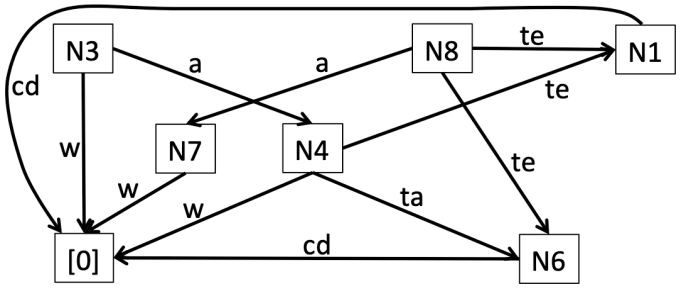


Figure 9: `ioa` summary of the RDF graph in Figure 3.

or physicians look for a rare diagnosis, it is important not to miss a "weak signal", encoded in RDF as a set of triples using infrequent properties.

STRUCTURAL HOMOMORPHISM: It is easy to see that the function f associating to each $\mathtt{G}$ node its representative in $\mathtt{G}_{/\equiv}$ is a homomorphism from $\mathtt{G}$ into its summary: any subgraph of $\mathtt{G}$ is "projected" by f into a subgraph of its quotient.

3.2.2 *Prior work*

Quotient graph summaries include, e.g., [25], [26], [28], [37], [38], [29], [39]; RDF quotient summaries are described in [30], [31], [32].

BISIMULATION Bisimilarity is behind most equivalence relations used in these works. Two RDF nodes n_1, n_2 are *forward bisimilar* [24] (denoted $\equiv_{\mathtt{fw}}$) iff:

- For every G edge $n_1 \xrightarrow{a} m_1$, G also comprises an edge $n_2 \xrightarrow{a} m_2$, such that m_1 and m_2 are also (forward) bisimilar, and
- A similar statement holds, replacing n_1, m_1 with n_2, m_2 and vice-versa.

Although forward bisimilarity focuses on *outgoing* edges only, two RDF nodes can also be bisimilar w.r.t. their *incoming* edges, i.e., *backward bisimilar* (denoted $\equiv_{\mathtt{bw}}$). Backward similarity of two RDF nodes n_1, n_2 is recursively defined similarly as above when considering G edges $m_1 \xrightarrow{a} n_1$ and $m_2 \xrightarrow{a} n_2$. Finally, two RDF nodes can be bisimilar based on both their incoming and outgoing edges, i.e., *forward and backward bisimilar* (denoted $\equiv_{\mathtt{fb}}$), when they are both *forward bisimilar* and *backward bisimilar*. We denote the bisimulation based summaries $\mathtt{G}_{/\mathtt{fw}}$ (forward), $\mathtt{G}_{/\mathtt{bw}}$ (backward), and $\mathtt{G}_{/\mathtt{fb}}$ (forward and backward, or "full bisimilarity"), respectively. They have been studied, in particular for indexing and query processing, in [30], [32], [31]. Computation of bisimulation summaries can also be parallelized [40].

BOUNDED BISIMILARITY Bisimulation summaries tend to be large, because bisimilarity is rare in heterogeneous graphs. For instance, in Figure 3, none of p_1, p_2, ... p_5 is bisimilar to the other, due to slight differences in their properties; similarly, the courses c_1, c_2, and c_3 are not bisimilar, because c_3 lacks a description, c_2 is the only one target of a *takes* triple, etc. Our experiments in Section 3.8 confirm this on many graphs.

To mitigate this problem, k-bisimilarity was introduced in [27]. For an integer k, nodes are k-forward (and/or backward) bisimilar iff they are bisimilar *within their k-bound neighborhoods*. One drawback of k-bisimilarity is that it requires users to guess the k value leading to the best compromise between compactness (favored by a low k, e.g., if $k = 0$, all the graph nodes are equivalent, and the summary consists of a single node) and structural information in the summary (a high k). RDF quotients based on k-bisimilarity are studied in [30], [32].

It turns out that even 1-bisimilarity is rare in heterogeneous graphs. For instance, Figure 7 shows the `1fb` summary of the sample graph in Figure 3[2]. Here and throughout this thesis, summary nodes are shown in rectangles and are labeled N_1, N_2, etc.; for brevity, we abridge property names to use *w* for wrote, *a* for *advises*, *te* for *teaches*, *ta* for *takes* and *cd* for *coursedescr*. Nodes N_3, N_4 and N_8 represent, respectively, p_1, p_2, p_3; the three courses are represented by N_1 and N_6. This summary is almost as complex as the input graph. Figure 8 shows the `1fw` summary of our sample G: it is smaller (only 5 nodes and 10 edges, whereas the `1fb` one has 8 nodes and 12 edges). However, as our

2 The summaries shown in the Figures 7, 8 and 9 ignore the type triples of G for readability and because they were not used for summarization in the referenced works.

experiments show, `1fw` summaries are still too large to be useful for visualization.

LEAF AND ROOT COLLAPSE We may encounter an issue when summarizing only according to the properties outgoing an RDF node, e.g., $\equiv_{\texttt{fw}}$. Namely, such summaries consider equivalent (thus, collapse) all RDF nodes lacking outgoing edges, that is, all the leaves of `G`. Such leaf nodes may have very little to do with each other. For instance, on the WatDiv graph summarized in Figure 6, all the values of *award*, *numberOfPages*, *keywords*, *email*, *faxNumber* would be summarized together, even though they are very different. We call this situation *leaf collapse through summarization*. Symmetrically, a summary whose relation only depends on nodes' incoming properties, e.g., $\equiv_{\texttt{bw}}$, automatically summarizes together all nodes with no incoming edges, although again they may represent very different things; we call this *root collapse*. We argue that a good summary should not systematically collapse leaves (respectively, roots), but do so only when they really are similar to each other.

In [32], each RDF equivalence relation (thus, any RDF quotient) considers all the leaf nodes, both URIs and literals, as equivalent and represents them by a single summary node denoted [0]. For instance, the equivalence called $\sim_{\texttt{b}}$ in [32] leads to a `b` summary that is obtained from `fb` by collapsing all leaves together. Thus, all those summaries automatically suffer from the leaf collapse issue.

The $\equiv_{\texttt{1fw}}$ equivalence, denoted $\sim_{\texttt{a}}$ in [32], is also very similar to grouping nodes into "characteristic sets" [41], [42]. This equivalence summarizes together all articles and course descriptions (N_2 in Figure 8). To avoid too many characteristic sets (or, equivalently, to reduce the number of summary nodes), we may apply a cardinality-based heuristic method proposed in [42], which merges them into at most r sets, for a user-specified threshold r. However, this burdens the user with choosing such a threshold. As our experiments show, some graphs are much more complex than others, thus it is not easy to set r, especially for users not yet acquainted with the data, such as the ones we target.

The $\equiv_{\texttt{1fb}}$ equivalence, denoted $\sim_{\texttt{ioa}}$ in [32], leads to the `ioa` summary shown in Figure 9. This may significantly reduce the number of summary nodes, since only one of them is a leaf. However, as reported in [32] and verified in our experiments, even after this reduction in the number of nodes, those bisimilarity-based summaries are still too large for visualization.

TYPE-DRIVEN SUMMARIZATION The remaining equivalence relations used in the literature to summarize RDF graphs are (also) based on *RDF types*.

In [32], two nodes are $\sim_{\mathtt{t}}$-equivalent if they have exactly the same types. This collapses all untyped nodes into a single summary node, e.g., all but p_1 and p_2 in Figure 3, and five out of the eight nodes in Figure 6 (N_2, N_3, N_4, N_5, and N_6), even though they are unrelated.

The $\sim_{\mathtt{ioat}}$ equivalence, also introduced in [32], considers two nodes equivalent iff they are $\equiv_{\mathtt{1fb}}$-equivalent and they have exactly the same types. Thus, a `ioat` summary has at least as many nodes and edges as the `ioa` one; our experiments confirm it is too large to be used for first-sight visualization.

3.2.3 *Limitations of the prior work*

To conclude, the equivalence relations used in prior RDF quotient summaries are based on:

1. Bisimilarity (possibly bounded to a maximum distance k), which leads to complex summaries with very high numbers of nodes and edges, unsuited for first-sight discovery.
2. Uni-directional bisimilarity (i.e., $\equiv_{\mathtt{fw}}$, $\equiv_{\mathtt{bw}}$ and their bounded variants), which suffer from leaf or root collapse;
3. Types alone: this collapses all untyped nodes, even when their data properties have nothing to do with each other;
4. Bisimilarity *and* having the same types; this leads to summaries at least as large as those based on bisimilarity alone.

Leaf collapse is also present in all the equivalences of [32].

Another limitation of prior work is not considering how summarization interacts with saturation, and instead simply assuming that $\mathtt{G}$ is already saturated. When this is not the case, obtaining the summary of $\mathtt{G}^{\infty}$ requires first computing this saturation, which may be costly in terms of run time and storage space.

To go beyond these limitations, in the sequel, we introduce our *novel equivalence relations*, leading to *compact and informative* quotient summaries of RDF graphs, whether they are fully typed, have no types at all, or are anywhere in between. Further, we provide *novel, advanced techniques for summarizing a graph's saturation without saturating it*; this can lead to speedups of orders of magnitude.

3.2.4 *Notation*

The notation we use in this chapter is compiled in Table 2.

3.3 DATA GRAPH SUMMARIZATION

We first consider graphs made of *data triples* only. We define the novel notion of *property cliques* in Section 3.3.1; building on them,

G	RDF graph (Section 2.1)
G^{∞}	the result of saturating G (Section 2.1)
type	*rdf:type* (Section 2.1)
subClass	*rdfs:subClassOf* (Section 2.1)
subProperty	*rdfs:subPropertyOf* (Section 2.1)
domain	*rdfs:domain* (Section 2.1)
range	*rdfs:range* (Section 2.1)
$\equiv_{\mathtt{fw}}$	forward bisimilarity [24]
$\equiv_{\mathtt{bw}}$	backward bisimilarity [24]
$\equiv_{\mathtt{fb}}$	forward and backward bisimilarity [26]
$\equiv_{\mathtt{1fw}}$	1-forward bisimilarity [27]
$\equiv_{\mathtt{1bw}}$	1-backward bisimilarity [27]
$\equiv_{\mathtt{1fb}}$	1-forward and backward bisimilarity [27]
$\sim_{\mathtt{b}}$	b equivalence (or $\equiv_{\mathtt{fb}}$ equivalence with leaf collapse) [32]
$\sim_{\mathtt{a}}$	a equivalence (or $\equiv_{\mathtt{1fw}}$ equivalence with leaf collapse) [32]
$\sim_{\mathtt{ioa}}$	input-output equivalence (or $\equiv_{\mathtt{1fb}}$ with leaf collapse) [32]
$\sim_{\mathtt{t}}$	typed equivalence [32]
$\sim_{\mathtt{ioat}}$	input-output and typed equivalence (or $\equiv_{\mathtt{1fb}}$ and typed with leaf collapse) [32]
$\equiv_{\mathtt{S}}$	strong equivalence (Definition 2)
$\equiv_{\mathtt{W}}$	weak equivalence (Definition 3)
$\equiv_{\mathtt{TS}}$	typed strong equivalence (Section 3.4.2)
$\equiv_{\mathtt{TW}}$	typed weak equivalence (Section 3.4.2)
$G_{/\mathtt{fb}}$	forward and backward bisimilarity summary of G [26]
$G_{/\mathtt{1fw}}$	1-forward bisimilarity summary of G [24]
$G_{/\mathtt{t}}$	typed summary of G [32]
$G_{/\mathtt{ioat}}$	input-output and typed summary of G [32]
$G_{/\mathtt{S}}$	strong summary of G (Definition 5)
$G_{/\mathtt{W}}$	weak summary of G (Definition 4)
$G_{/\mathtt{TS}}$	typed strong summary of G (Section 3.4.2)
$G_{/\mathtt{TW}}$	typed weak summary of G (Definition 6)
$\simeq$	strong homomorphism (Definition 7)

Table 2: Summary of the notation used in Chapter 3.

we devise new graph node equivalence relations and corresponding

graph summaries in Section 3.3.2. Summarization will be generalized to handle also *type triples* in Section 3.4.

3.3.1 *Data property cliques*

Let us consider the ways in which data properties (edge labels) are organized in a graph. The simplest relation is *co-occurrence*, when a node is the source (or target) of two edges carrying the two labels. However, as illustrated in Figure 3, two properties, such as *wrote* and *advises*, may co-occur on one node, whereas another one may have *wrote* and *teaches*. The main intuition behind this work is to consider *all* these properties (*wrote*, *advises*, *teaches*) related, as they *directly or transitively* co-occur on some nodes. Formally:

Definition 1 (Property relations and cliques) *Let* $\mathtt{p}_1$, $\mathtt{p}_2$ *be two data properties in* $\mathtt{G}$*:*

1. $\mathtt{p}_1, \mathtt{p}_2 \in \mathtt{G}$ *are* source-related *iff either:*
   - *A data node in* $\mathtt{G}$ *is the subject of both* $\mathtt{p}_1$ *and* $\mathtt{p}_2$*, or*
   - $\mathtt{G}$ *holds a data node that is the subject of* $\mathtt{p}_1$ *and of a data property* $\mathtt{p}_3$*, with* $\mathtt{p}_3$ *and* $\mathtt{p}_2$ *being source-related.*
2. $\mathtt{p}_1, \mathtt{p}_2 \in \mathtt{G}$ *are* target-related *iff either:*
   - *A data node in* $\mathtt{G}$ *is the object of both* $\mathtt{p}_1$ *and* $\mathtt{p}_2$*, or*
   - $\mathtt{G}$ *holds a data node that is the object of* $\mathtt{p}_1$ *and of a data property* $\mathtt{p}_3$*, with* $\mathtt{p}_3$ *and* $\mathtt{p}_2$ *being target-related.*

A maximal set of data properties in $\mathtt{G}$ *which are pairwise source-related (respectively, target-related) is called a* source *(respectively,* target*)* property clique*.*

In the graph in Figure 3, properties *advises* and *teaches* are source-related due to p_4 (Condition 1 in the definition). Similarly, *advises* and *wrote* are source-related due to p_1; consequently, teaches and wrote are source-related (Condition 2). Further, the graduate student p_2 *teaches* a course and *takes* another, thus *teaches*, *advises*, *wrote* and *takes* are all part of the same source clique. Table 3 shows the target and source cliques of all data nodes from Figure 3.

It is easy to see that the set of non-empty source (or target) property cliques is a *partition over the data properties of* $\mathtt{G}$. Further, if a node $n \in \mathtt{G}$ is a source of some data properties, they are all in the same source clique; similarly, all the properties of which n is a target are in the same target clique.

3.3.2 *Strong and weak node equivalences*

Building on property cliques, we define two *node equivalence relations* among the data nodes of a graph $\mathtt{G}$:

	Source clique and nodes having this source clique
SC_1	$\{advises, takes, teaches, wrote\}$: p_1, p_2, p_3, p_4, p_5
SC_2	$\{coursedescr\}$: c_1, c_2, c_3
SC_3	$\emptyset$: a_1, a_2
	Target clique and nodes having this target clique
TC_1	$\{advises\}$: p_2, p_5
TC_2	$\{teaches, takes\}$: c_1, c_2, c_3
TC_3	$\{coursedescr\}$: d_1, d_2
TC_4	$\{wrote\}$: a_1, a_2
TC_5	$\emptyset$: p_1, p_3, p_4

Table 3: Source and target cliques of G nodes (Figure 3).

Definition 2 (Strong equivalence) *Two data nodes n_1, n_2 of G are* strongly equivalent, *denoted $n_1 \equiv_S n_2$, iff they have the same source* and *target cliques.*

Strongly equivalent nodes have the same structure of *incoming and outgoing* edges. In Figure 3, nodes p_1, p_3 and p_4 are *strongly* equivalent to each other. Among these, note that p_1 and p_3 may seem very dissimilar: p_1 has the properties $\{wrote, advises\}$, whereas p_3 has only $\{teaches\}$. These nodes are strongly equivalent due to the node p_4 which has a common outgoing property with p_1, and one with p_3. Similarly, p_2, p_5 are strongly equivalent, and so are c_1, c_2 and c_3, etc. The transitivity built in strong equivalence through the use of cliques allows to recognize all these publication nodes as equivalent, and avoid separating them (as $\equiv_{fb}$ and $\equiv_{fw}$ do, recall Figures 7, 8 in Section 3.2). Thus, clique-based equivalence avoids the pitfall of leading to too many nodes for a readable visualization.

A second, weaker notion of node equivalence requests only that equivalent nodes share the same *incoming or outgoing* structure, i.e., they share the same source clique or the same target clique. Formally:

Definition 3 (Weak equivalence) *Two data nodes n_1, n_2 are* weakly equivalent, *denoted $n_1 \equiv_W n_2$, iff: (i) they have the same non-empty source or non-empty target clique, or (ii) they both have empty source and empty target cliques, or (iii) they are both weakly equivalent to another node of G.*

It is easy to see that $\equiv_S$ and $\equiv_W$ are equivalence relations and that strong equivalence implies weak equivalence, noted $\equiv_S \Rightarrow \equiv_W$.

In Figure 3, $p_1, \ldots, p_5$ are *weakly* equivalent to each other due to their common source clique SC_1; a_1, a_2 are *weakly* equivalent due to their common target clique, etc.

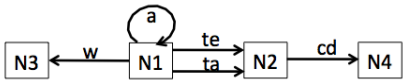


Figure 10: Weak summary of the RDF graph in Figure 3 (type triples excluded).

From the definitions above and the equivalence notions recalled in Section 3.2, it follows that $\equiv_{\texttt{1fb}}$ is more restrictive than $\equiv_{\texttt{S}}$ ($\equiv_{\texttt{1fb}} \Rightarrow \equiv_{\texttt{S}} \Rightarrow \equiv_{\texttt{W}}$). However, $\equiv_{\texttt{S}}$ and $\equiv_{\texttt{W}}$, which reflect outgoing *and* incoming properties, are in general incomparable with $\equiv_{\texttt{1fw}}$ and $\equiv_{\texttt{1bw}}$, which reflect only outgoing, respectively, incoming properties. The *transitive aspect* of property cliques is a radical departure from previously considered equivalence relations (Chapter 2). It gives $\equiv_{\texttt{S}}$ and $\equiv_{\texttt{W}}$ the flexibility to accept as equivalent structurally heterogeneous nodes, leading to summaries which are both meaningful and compact.

PROPERTY NODES EQUIVALENCE We make an important addition to the clique-based equivalences introduced above. A property node (Section 2.1), that is, a node (subject or object) labeled by a URI which also appears as a property of a data node, is only strongly and weakly equivalent to itself. Property nodes are pretty rare, but they do occur in some RDF graphs; an example would be the subject of a triple such as *takes rdfs:isDefinedBy* studentOfficerX, where *takes* appears as a data property in Figure 3. A triple comprising a property node can be seen as a form of *metadata*, which helps interpret/understand the graph's data triples. Thus, we consider that a property node is only equivalent to itself.

3.3.3 *Weak and strong summarization*

WEAK SUMMARIZATION The first summary we define is based on the weak equivalence:

Definition 4 (Weak summary) *The* weak summary *of a data graph* $\texttt{G}$, *denoted* $\texttt{G}_{/\texttt{W}}$, *is its quotient graph w.r.t. the weak equivalence relation* $\equiv_{\texttt{W}}$.

The weak summary of the graph in Figure 3 is depicted in Figure 10. N_1 represents all the people (p_1 to p_5), N_2 represents the courses, N_3 the articles and N_4 the course descriptions. Note the self-loop from N_1 to itself; it denotes that *some* nodes represented by N_1 advise *some* nodes represented by N_1. This summary has only 4 nodes and 5 edges; it is smaller (at most half as many edges) and much easier to grasp than the `1fb`, `1fw` and `ioa` ones, shown in Chapter 2. At the same time, it conveys the essential information that some nodes *advise*, *write*, also they *teach* and *take* something that *has course descriptions*.

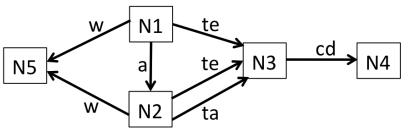


Figure 11: Strong summary of the RDF graph in Figure 3 (type triples excluded).

Generally speaking, node N_i in the weak summary $\mathtt{G}_{/\mathtt{W}}$ of a graph $\mathtt{G}$ represents *all the* $\mathtt{G}$ *nodes whose outgoing (respectively, incoming) properties are a subset of the outgoing (respectively, incoming) properties of* N_i.

The weak summary has the following important property:

Proposition 1 (Unique data properties [4]) *Each* $\mathtt{G}$ *data property appears exactly once in* $\mathtt{G}_{/\mathtt{W}}$.

We exploit this to efficiently build weak graph summaries (Section 3.7).

We remark that the weak summary $\mathtt{G}_{/\mathtt{W}}$ of a graph $\mathtt{G}$ has the *minimal size* (in the number of edges) among *all* the quotient summaries of $\mathtt{G}$: every property labeling a $\mathtt{G}$ edge appears *exactly once* in $\mathtt{G}_{/\mathtt{W}}$, while, by definition, it appears *at least once* in any quotient summary (Section 3.2). Our experiments show that $|\mathtt{G}_{/\mathtt{W}}|$ is typically 3 to 6 orders of magnitude smaller than $|\mathtt{G}|$.

STRONG SUMMARIZATION Next, we introduce the summary based on the strong equivalence:

Definition 5 (Strong summary) *The* strong summary *of the graph* $\mathtt{G}$, *denoted* $\mathtt{G}_{/\mathtt{S}}$, *is its quotient graph w.r.t. the strong equivalence relation* $\equiv_{\mathtt{S}}$.

The strong summary of the graph of Figure 3 is shown in Figure 11. Similarly to the weak summary (Figure 10), the strong one groups all courses together, and all articles together. However, it separates the person node in two: those represented by N_1 advise those represented by N_2. This is because the target clique of p_1, p_3 and p_4 is empty, whereas the target clique of p_2 and p_5 is $\{advises\}$ (Table 3). Due to this finer granularity, in $\mathtt{G}_{/\mathtt{S}}$, several edges may have the same label, e.g., there are two *teaches* and two *wrote* edges in Figure 11, whereas in $\mathtt{G}_{/\mathtt{W}}$, as stated in Proposition 1, this is not possible. Our experiments (Section 3.8) show that, even though $\mathtt{G}_{/\mathtt{S}}$ is often somehow larger than $\mathtt{G}_{/\mathtt{W}}$, it still remains many orders of magnitude smaller than the original graph.

By definition of $\equiv_{\mathtt{S}}$, equivalent nodes have the same source clique and the same target clique. This leads directly to the next result, exploited by our algorithms for building strong summaries (Section 3.7):

Proposition 2 (Strong summary nodes and $\mathtt{G}$ cliques) $\mathtt{G}_{/\mathtt{S}}$ *has exactly one node for each source clique and target clique of the same* $\mathtt{G}$ *data node.*

Proof *If two* $\mathtt{G}_{/\mathtt{S}}$ *distinct nodes had the same source and the same target clique, they would be strongly equivalent. This cannot be the case in a quotient summary obtained through* $\equiv_{\mathtt{S}}$*, since by definition, such a summary has one node for each* $\equiv_{\mathtt{S}}$ *equivalence class. Thus, any two distinct* $\mathtt{G}_{/\mathtt{S}}$ *have distinct source cliques and/or distinct target cliques.*

Now, let m *be a* $\mathtt{G}_{/\mathtt{S}}$ *node, and* $S_m = f_{\mathtt{S}}^{-1}(m)$ *be the set of all* $\mathtt{G}$ *nodes represented by* m*. By the definition of a quotient summary,* m *must be the target (respectively, the source) of an edge carrying each of the labels on the edges entering (respectively, going out of) any node* $n \in S_m$*. Thus,* m *is source of all the properties in the source clique shared by the nodes in* S_m*, and is target of all the properties in the target clique shared by the nodes in* S_m*. Thus,* m *has the source and target clique of any node from* S_m*; this concludes our proof.*

FROM THE STRONG TO THE WEAK SUMMARY Because strong equivalence implies weak equivalence, it follows that $(\mathtt{G}_{/\mathtt{S}})_{/\mathtt{W}} = \mathtt{G}_{/\mathtt{W}}$. For instance, the nodes N_1 and N_2 in Figure 11 have the same source clique, thus the weak summary of the graph in Figure 11 is exactly the one in Figure 10. Hence, one can get *both* $\mathtt{G}_{/\mathtt{W}}$ *and* $\mathtt{G}_{/\mathtt{S}}$ by building $\mathtt{G}_{/\mathtt{S}}$ and then weakly summarizing it to also get $\mathtt{G}_{/\mathtt{W}}$. This is (much) faster than re-summarizing $\mathtt{G}$, mainly because $\mathtt{G}_{/\mathtt{S}}$ is much smaller than $\mathtt{G}$. Another consequence is that $\mathtt{G}_{/\mathtt{W}}$, intuitively, compresses more (is more imprecise) than $\mathtt{G}_{/\mathtt{S}}$[3]; we demonstrate this also through experiments (Table 7 in Section 3.8).

PROPERTY NODE REPRESENTATION Since property nodes represent a form of metadata about the data graph, we decide that in *all* our quotient summaries they are always represented by themselves, i.e., a node labeled with the same URI.

GENERIC PROPERTIES Some special properties frequently used in RDF deserve a special treatment. First, the standard *owl:sameAs* property is used to denote that two URIs should be considered as being "the same"; in particular, the incoming/outgoing edges of one should also be considered as belonging to the other. To reflect this special semantics, we extend our notion of clique to *treat the properties incoming/outgoing two nodes connected by owl:sameAs (directly or indirectly) as if they occurred on the same node*. This ensures that any two nodes connected by *owl:sameAs* have the same source and target clique, thus they are weakly and strongly equivalent. Second, some generic properties such as *rdfs:label* are sometimes used to annotate RDF nodes with very different meaning. Building cliques based on the co-occurrence of such generic properties may consider too many

3 One example among many: the W summary of a BSBM 1M graph has just one node, whereas the S summary has 5 and is quite informative (https://rdfquotient.inria.fr/files/2019/11/bsbm1m_s_split_and_fold_leaves.png).

nodes equivalent. To avoid this, we build our cliques ignoring the triples whose properties are generic, construct $G_{/W}$ or $G_{/S}$ accordingly, then, for each triple of the form n_1 *rdfs:label* t (where t is some text), we add an edge N_1 *rdfs:label* N_2 to the weak or strong summary, where N_1 is the representative of n_1, and N_2 is a new summary node, representing t.

3.4 TYPED DATA GRAPH SUMMARIZATION

We now discuss the summarization of graphs with data and type triples. Types represent domain knowledge that the data producers found meaningful to describe it. However, some or all nodes of a graph may lack types.

Only a few RDF graph summarization works explicitly considered type triples. The equivalence relation $\sim_t$ introduced in [32] follows an approach we call *type-only*: nodes are equivalent if they have exactly the same types. This approach groups together all untyped nodes, which is problematic in graphs such as the one summarized in Figure 6. The same approach is taken in [43] which further assumes that all non-leaf nodes are typed, a hypothesis not borne out in practice (see again Figure 6). A different approach we term *data-and-type* is taken in [32]: for two nodes to be equivalent, they should both be equivalent according to a relation that only reflects their data properties, and have the same types. For instance, $\sim_{ioat}$ is based both on having the same input and output properties ($\sim_{ioa}$), and the same types ($\sim_t$). As explained in Chapter 2, this splits graph nodes into many equivalence classes (summary nodes), which is not desirable for first-sight visualization.

A better approach introduced in [32] to reflect types in a quotient summary built from an equivalence relation $\equiv$ is as follows: first, summarize G ignoring type triples, and second, for each triple n *rdf:type* c in G, add to $G_{/\equiv}$ a triple N *rdf:type* c, where N represents n in $G_{/\equiv}$. We call this quotient summarization approach *data-then-type*. It does not suffer from the disadvantages of *type-only* nor *data-and-type*. Instead, it allows to identify meaningful node groups even in graphs where some or all the nodes lack types.

Below, we start by formalizing the special treatment we argue should be given to class and property nodes in any RDF quotient summary. Based on this, we extend the *data-then-type* approach to our clique-based equivalence relations (Section 3.4.1), then present another novel approach which we call *type-then-data*. It gives priority to types when available, while still avoiding the pitfalls of type-only and data-and-type summarization (Section 3.4.2).

CLASS NODE EQUIVALENCE AND REPRESENTATION To ensure quotient summaries preserve the application knowledge encoded

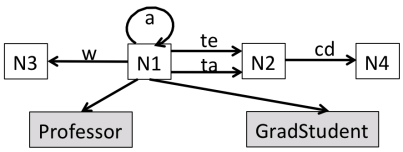


Figure 12: Weak summary of the graph in Figure 3.

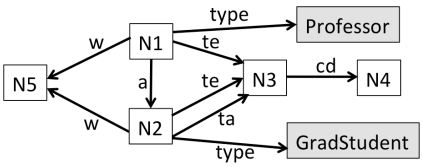


Figure 13: Strong summary of the graph in Figure 3.

within the classes, properties and ontology of a graph, we decide that in any equivalence relation $\equiv$, any class node is only equivalent to itself, and any class node is represented by itself, and similarly for property nodes. Hence, a typed data graph has the same class nodes, and the same property nodes (Section 3.3), as its summary.

3.4.1 *Data-then-type summarization*

We extend the `W`, respectively, `S` summaries to type triples, by stating that they follow the data-then-type approach.

Figure 12 illustrates this for $\mathtt{G}_{/\mathtt{W}}$; note that N_1 is attached both Professor and GradStudent types. Generally, a typed node N_i in a typed weak summary represents *all the* `G` *nodes whose incoming/outgoing properties are included in those of* N_i*, some of which may also have some of the types of* N_i; the `G` nodes represented by an untyped node N_j are the same as in a weak summary.

Figure 13 shows the strong summary of our sample graph when type triples are considered. Note that the *Professor* type is attached to N_1, the representative of p_1, p_3 and p_4 (those who *advise* someone), whereas *GradStudent* is attached to N_5, representing the *advise*es (p_2 and p_5).

3.4.2 *Type-then-data summarization*

This approach is novel. In contrast with data-then-types, it considers that node types are more important when deciding whether nodes are equivalent. However, it still relies on data properties to summarize untyped nodes. Thus, from an equivalence relation $\equiv$ (based on data properties alone), we derive a *novel typed equivalence relation* where two nodes are equivalent if:

- Both are typed, and they have the same set of types; or

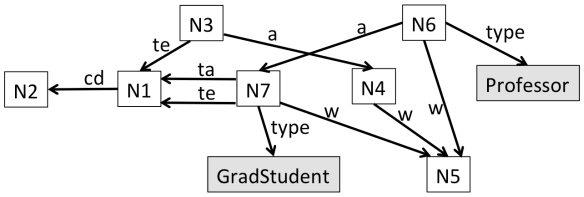


Figure 14: Typed weak summary of the graph in Figure 3.

- Both are untyped, and they are equivalent according to $\equiv$.

For weak summarization, this approach leads to:

Definition 6 (Typed weak summary) *Let* $\equiv_{\mathtt{TW}}$ (typed weak equivalence) *be an equivalence relation that holds between two data nodes* n_1, n_2 *iff:*

- n_1, n_2 *have no types in* $\mathtt{G}$ *and* $n_1 \equiv_{\mathtt{W}} n_2$*; or*
- n_1, n_2 *have the same non-empty set of types in* $\mathtt{G}$*.*

The typed weak summary $\mathtt{G}_{/\mathtt{TW}}$ *of a graph* $\mathtt{G}$ *is denoted* $\mathtt{G}_{/\mathtt{TW}}$.

Figure 14 shows the typed weak summary of our sample RDF graph. Unlike $\mathtt{G}_{/\mathtt{W}}$ (Figure 10), $\mathtt{G}_{/\mathtt{TW}}$ represents p_1 by N_1, separately from p_3, because p_3 is of type *Person*, whereas p_3 is untyped.

In a similar manner, we define **typed strong equivalence**, denoted $\equiv_{\mathtt{TS}}$, as in Definition 6 by replacing $\equiv_{\mathtt{W}}$ with $\equiv_{\mathtt{S}}$, and denoting by $\mathtt{G}_{/\mathtt{TS}}$ the **typed strong summary** of a graph $\mathtt{G}$. In our example, $\mathtt{G}_{/\mathtt{TS}}$ coincides with $\mathtt{G}_{/\mathtt{TW}}$.

FROM THE TYPED STRONG TO TYPED WEAK SUMMARY It is easy to see that if $n_1 \equiv_{\mathtt{TS}} n_2$, then also $n_1 \equiv_{\mathtt{TW}} n_2$; thus $(\mathtt{G}_{/\mathtt{TS}})_{/\mathtt{TW}} = \mathtt{G}_{/\mathtt{TW}}$. This also allows building $\mathtt{G}_{/\mathtt{TS}}$ and $\mathtt{G}_{/\mathtt{TW}}$ for almost the cost of building $\mathtt{G}_{/\mathtt{TS}}$ alone, as this summary is small and thus summarized quickly.

SUMMARY EQUALITY We now analyze when two of our summaries may coincide for a given graph $\mathtt{G}$, i.e., they are the same up to their data node labels N_1, N_2, etc. To formalize this, we define:

Definition 7 (Strong isomorphism $\simeq$) *A strong isomorphism between two RDF graphs* $\mathtt{G}_1, \mathtt{G}_2$*, noted* $\mathtt{G}_1 \simeq \mathtt{G}_2$*, is an isomorphism which is the identity for the class and property nodes.*

We remark that for visualization purposes, strongly isomorphic summaries can be seen as identical as they describe exactly the same structure.

3.5 SUMMARIZATION OF GRAPHS WITH RDFS ONTOLOGIES

We now turn to the general case of an RDF graph with an RDFS ontology.

First, observe that any summary of an RDF graph has the same RDFS ontology as this graph. This is because:

1. Every property node is only represented by itself (Section 3.3.2),
2. Every class node is represented by itself (Section 3.3.3), and
3. By definition of an RDF quotient, any ontology triple, which only connects two class or property nodes, is "copied" in the summary. We view ontology preservation as a desirable feature, since the ontology has crucial information about the meaning of the data[4].

Second, we identify two ways in which an ontology can impact summarization:

THROUGH TYPE GENERALIZATION Type-then-data summarization (Section 3.4.2) groups nodes by their sets of types. If the ontology features triples of the form c_1 *rdfs:subClassOf* c_2, it can be argued that c_2 can be used instead of c_1 to summarize a resource having the type c_1. We study this in Section 3.5.1.

THROUGH IMPLICIT TRIPLES As we explained in Chapter 2, the semantics of an RDF graph $\mathtt{G}$ includes its explicit triples, but also its *implicit* triples which are not in $\mathtt{G}$, but hold in $\mathtt{G}^{\infty}$ due to ontological constraints (such as the triples p_2 *rdf:type* Instructor, p_2 *rdf:type* Student, and p_1 *knows* p_2 in Section 2.1). An interesting question, then, is to determine the interplay between saturation and summarization: how is the summary of $\mathtt{G}^{\infty}$ related to that of $\mathtt{G}$, first, in general (for any quotient summary), and then, for the four summaries we introduced? The reminder of this section is devoted to this topic.

3.5.1 *Type-then-data summarization using most general types*

The most commonly used feature of RDFS ontologies is the subClass relationship, stating that any resource of a type c_1 is also of the type c_2; "subtype" and "supertype" are commonly used to denote c_1 and c_2 in such settings. The subgraph consisting of the subClass triples of a graph is typically acyclic[5]; we assume below that this is the case, thus

4 Even though we consider the ontology very important, our goal is to bring the much more numerous *data and type* (non-ontology) triples to a visually comprehensible size through summarization. When present, the ontology may help visualize the data; the ontology itself may be summarized, etc.

5 If a loop existed, all the types involved in the loop would be equivalent for all practical purposes and could be replaced with any among them.

the types present in G can be organized in a directed acyclic graph (DAG). Even though class nodes are very often much fewer than data nodes, they can still be too numerous for a small visualization to include or reflect all of them. For instance, there are more than 500 product types labeled ProductType1, ProductType2, etc., in a BSBM benchmark graph of 100M triples, a WatDiv benchmark graph of 10M triple comprises 14 product category types, or the real-life DBLP dataset includes a dozen types of scientific publications. Applying type-then-data summarization to such a graph would lead to a high number of nodes, one for each type; this appears shortsighted, given that in these examples, a natural common supertype can be found, e.g., ProductType, ProductCategory, and Publication, respectively.

To obtain compact type-then-data summaries even in the presence of such ontologies, we adopt the following practical solution:

- For each typed data node $x \in \mathtt{G}$, let $\tau(x)$ be the set of all types associated with x in G, and $\overline{\tau(x)}$ the set comprising the most general supertypes of the types in $\tau(x)$, which can be easily computed based on the subClass triples. We exclude a few "standard" root types, such as *rdf:resource* or *owl:Thing*, from the supertype hierarchy, as these would not bring useful information to summary users.

- Then, **type-then-data summarization based on most general types** uses $\overline{\tau(x)}$ instead of $\tau(x)$. This is how we obtained the graph in Figure 6: there, N_1 represents all the nodes whose most general type set comprises exactly `http://db.uwaterloo.ca/~galuc/wsdbm/Genre`, and similarly for N_8 and the type `http://db.uwaterloo.ca/~galuc/wsdbm/ProductCategory`.

This technique can be applied to both typed weak and typed strong summarization.

In our previous work [5], we defined a variant of type generalization based on type-then-data summarization; its algorithm adopts a bottom-up approach of traversing the type hierarchy until we find the lowest branching point, i.e., such an RDF type that has more than one supertype.

3.5.2 *Interactions between summarization and saturation*

In [3], the relationship between summarization and saturation has been studied, and this analysis has been furthered in the paper [44] that belongs to this thesis. We start by recalling the main results of [3] as follows.

First, does saturation *commute with* summarization? In other words, is $(\mathtt{G}^{\infty})_{/\equiv}$ strongly isomorphic (Definition 7) to $(\mathtt{G}_{/\equiv})^{\infty}$? Figure 15 shows that this is not always the case; sp denotes the standard property RDFS *rdfs:subPropertyOf* (Section 2.1). For a given graph G, the

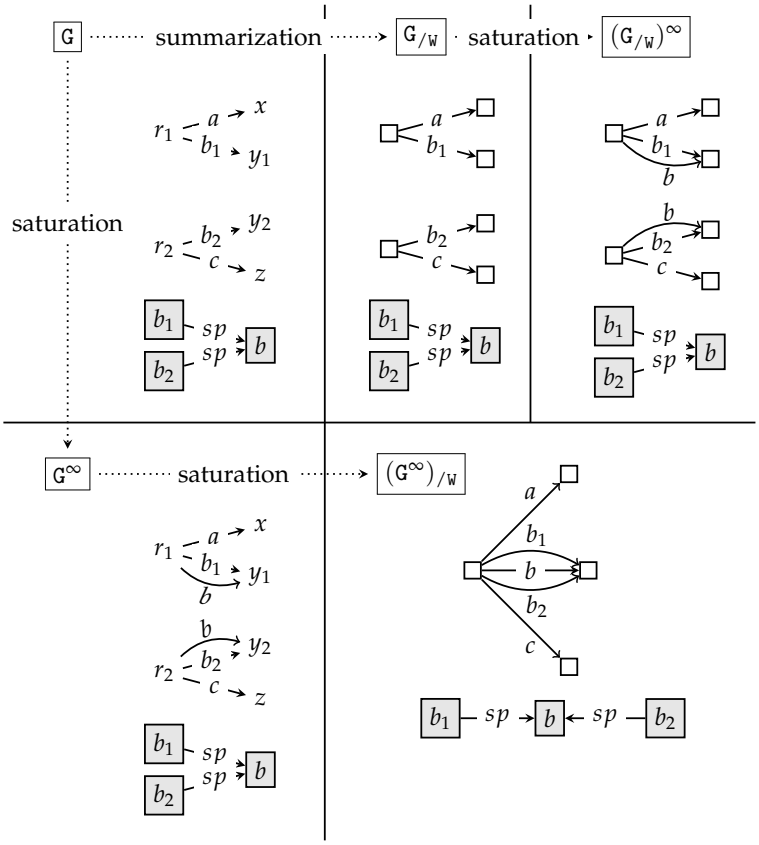


Figure 15: Saturation and summarization example.

figure shows its weak summary $\mathtt{G}_{/\mathtt{W}}$ and its saturation $(\mathtt{G}_{/\mathtt{W}})^{\infty}$, as well as $\mathtt{G}^{\infty}$ and its summary $(\mathtt{G}^{\infty})_{/\mathtt{W}}$. Here, saturation leads to b edges outgoing both r_1 and r_2 which makes them equivalent in $\mathtt{G}^{\infty}$. In contrast, summarization *before* saturation represents them separately; saturating the summary cannot unify them as in $(\mathtt{G}^{\infty})_{/\mathtt{W}}$ (recall from Section 2.1 that saturation can only *add edges* in a graph).

Even though $(\mathtt{G}^{\infty})_{/\equiv}$ and $(\mathtt{G}_{/\equiv})^{\infty}$ are *not* strongly isomorphic in general, we establish that they *always* relate as follows (see the diagram in Figure 16):

Theorem 1 (Summarization homomorphism [3]) *Let* $\mathtt{G}$ *be an RDF graph,* $\mathtt{G}_{/\equiv}$ *its summary and* f *the corresponding representation function from* $\mathtt{G}$ *nodes to* $\mathtt{G}_{/\equiv}$ *nodes. Then,* f *defines a homomorphism from* $\mathtt{G}^{\infty}$ *to* $(\mathtt{G}_{/\equiv})^{\infty}$.

Since $(\mathtt{G}_{/\equiv})^{\infty}$ is homomorphic to $\mathtt{G}^{\infty}$, would *their* summaries coincide, i.e., be strongly isomorphic? It turns out that this may hold or not depending on the RDF equivalence relation under consideration. When it holds, we call *shortcut* the following three-step transformation aiming at obtaining a summary strongly isomorphic to $(\mathtt{G}^{\infty})_{/\equiv}$, instead of $(\mathtt{G}^{\infty})_{/\equiv}$ itself: first summarize $\mathtt{G}$; then saturate its summary; finally, summarize it again in order to build $((\mathtt{G}_{/\equiv})^{\infty})_{/\equiv}$:

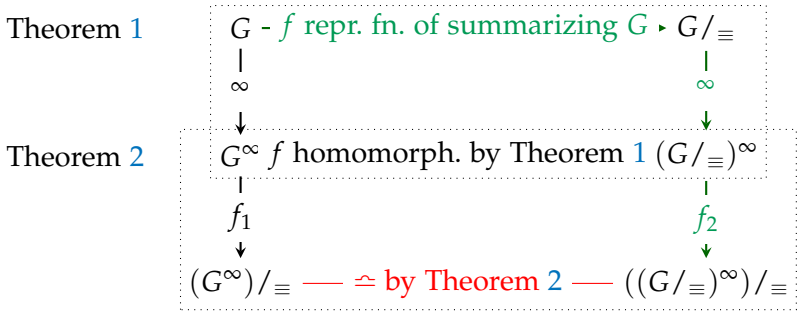


Figure 16: Illustration for Theorem 1 and Theorem 2.

Definition 8 (Shortcut [3]) *We say* the shortcut holds *for a given RDF node equivalence relation* $\equiv$ *iff for any* $\mathtt{G}$, $(\mathtt{G}^{\infty})_{/\equiv}$ *and* $((\mathtt{G}_{/\equiv})^{\infty})_{/\equiv}$ *are strongly isomorphic.*

Note that, from a practical viewpoint, hence for visualization, if the shortcut holds for $\equiv$, then $(\mathtt{G}^{\infty})_{/\equiv}$ and $((\mathtt{G}_{/\equiv})^{\infty})_{/\equiv}$ are equivalent as they differ just in their data node IDs (e.g., N_1, N_2, etc., in Figure 6), which carry no particular meaning.

Next, we establish one of our main contributions: a *sufficient condition* under which for *any* quotient summary based on an equivalence relation $\equiv$ as discussed above (where class and property nodes are preserved by summarization), the shortcut holds. In particular, as we will demonstrate (Section 3.8), the existence of the shortcut can lead to computing $(\mathtt{G}^{\infty})_{/\equiv}$ *substantially faster* by actually computing $((\mathtt{G}_{/\equiv})^{\infty})_{/\equiv}$.

Theorem 2 (Sufficient shortcut condition [3]) *Let* $\mathtt{G}_{/\equiv}$ *be a summary of* $\mathtt{G}$ *through* $\equiv$ *and* f *the corresponding representation function from* $\mathtt{G}$ *nodes to* $\mathtt{G}_{/\equiv}$ *nodes (see Figure 16).*

If $\equiv$ *satisfies: for any RDF graph* $\mathtt{G}$ *and any pair* (n_1, n_2) *of* $\mathtt{G}$ *nodes,* $n_1 \equiv n_2$ *in* $\mathtt{G}^{\infty}$ *iff* $f(n_1) \equiv f(n_2)$ *in* $(\mathtt{G}_{/\equiv})^{\infty}$, *then the shortcut holds for* $\equiv$.

Figure 16 depicts the relationships between an RDF graph $\mathtt{G}$, its saturation $\mathtt{G}^{\infty}$ and summarization $(\mathtt{G}^{\infty})_{/\equiv}$ thereof, and the RDF graphs that appear at each step of the shortcut computation. The intuition for the sufficient condition is the following. On any path in Figure 16, saturation adds edges to its input graph, whereas summarization "fuses" nodes into common representatives. On the regular path from $\mathtt{G}$ to $(\mathtt{G}^{\infty})_{/\equiv}$, edges are added in the first step, and nodes are fused in the second. On the shortcut (green) path, edges are added in the second step, whereas nodes are fused in the first and third steps. The two paths starting from $\mathtt{G}$ can reach $\simeq$ results only if $\mathtt{G}$ nodes fused on the shortcut path are also fused (when summarizing $\mathtt{G}^{\infty}$) on the standard path. In particular, *the first summarization along the shortcut path should not make wrong node fusions*, that is, fusions not made when considering the full $\mathtt{G}^{\infty}$: such a "hasty" fusion *can never be corrected later on along the shortcut path*, as neither summarization nor saturation split

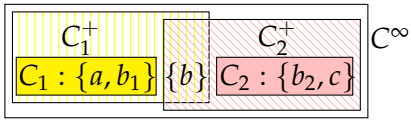


Figure 17: Two source cliques from the graph in Figure 15, their saturations, and their enclosing clique C^∞ in $\mathtt{G}^\infty$.

nodes. Thus, an erroneous fusion made in the first summarization step irreversibly prevents the end of the shortcut path from being $\mathrel{\widehat{=}}$ to $(\mathtt{G}_{/\equiv})^\infty$.

When the condition is met, summarizing $\mathtt{G}$, then saturating its summary, then summarizing the graph thus obtained leads to $(\mathtt{G}^\infty)_{/\equiv}$ (the same up to data node labels) *without the need to saturate* $\mathtt{G}$. The shortcut can be faster than saturating $\mathtt{G}$ then summarizing the result, because the shortcut avoids the cost to *find, store, and summarize* the implicit triples derived from $\mathtt{G}$; it only deals with the implicit triples derived from $\mathtt{G}_{/\equiv}$, which (depending on $\equiv$) may be much smaller than $\mathtt{G}$.

3.5.3 *Shortcut results*

As briefly described in the publication list following the abstract, the formal results in this section appear in our publication [44], the article of reference for RDFQuotient-related work. The implementation of the algorithms, including the shortcut procedure has been started during my master's end-of-study research internship and has been finalized during my PhD program. However, these theorems and lemmas have been established for the first time in the technical report [4] before my joining of the project. We state the formal claims here for completeness, whereas we skip the mathematical proofs and details that can be found in [44] (or [4]).

In order to establish shortcut results for our summaries, we start by investigating how property cliques are impacted by saturation.

In $\mathtt{G}^\infty$, every $\mathtt{G}$ node has all the data properties it had in $\mathtt{G}$; therefore, two data properties belonging to a $\mathtt{G}$ clique are also in the same clique of $\mathtt{G}^\infty$. Further, if the schema of $\mathtt{G}$ comprises subProperty constraints, a node may have a data property in $\mathtt{G}^\infty$ that it did not have in $\mathtt{G}$. As a consequence, each $\mathtt{G}^\infty$ clique includes one or several cliques from $\mathtt{G}$, which may "fuse" by acquiring more properties due to saturation with subProperty constraints. An example is given in Figure 17, where C_1^+ and C_2^+ are the saturations of the source cliques C_1, C_2, whereas $C^\infty = \{a, b_1, b, b_2, c\}$ is a source clique of the graph $\mathtt{G}^\infty$ (also in Figure 15).

Based on Theorem 2 and the above observations, we show:

Theorem 3 ($\mathtt{W}$ shortcut [44]) *The shortcut holds for* $\equiv_{\mathtt{W}}$.

For instance, on the graph in Figure 15, it is easy to check that applying summarization on $(\mathtt{G}_{/\mathtt{W}})^{\infty}$ (as prescribed by the shortcut) leads exactly to a graph strongly isomorphic to $(\mathtt{G}^{\infty})_{/\mathtt{W}}$.

Showing Theorem 3 is rather involved; we do it in several steps. We first show a technical lemma below that describes the relationships between a clique C of $\mathtt{G}$, its saturated version C^+, and the cliques of $\mathtt{G}^{\infty}$:

Lemma 1 (Saturation vs. property cliques [44]) *Let C, C_1, C_2 be non-empty source (or target) cliques of $\mathtt{G}$.*

1. *There exists exactly one source (respectively, target) clique C^{∞} of $\mathtt{G}^{\infty}$ such that $C \subseteq C^{\infty}$.*

2. *If $C_1^+ \cap C_2^+ \neq \emptyset$, then all the properties in C_1 and C_2 are in the same $\mathtt{G}^{\infty}$ clique C^{∞}.*

3. *Any non-empty source (or target) clique C^{∞} is a union of the form $C_1^+ \cup \cdots \cup C_k^+$ for some $k \geq 1$, where each C_i is a non-empty source (respectively, target) clique of $\mathtt{G}$, and for any C_i, C_j where $1 \leq i, j \leq k$ with $i \neq j$, there exist some cliques $D_1 = C_i, \ldots, D_n = C_j$ in the set $\{C_1, \ldots, C_k\}$ such that:*

$$D_1^+ \cap D_2^+ \neq \emptyset, \ldots, D_{n-1}^+ \cap D_n^+ \neq \emptyset$$

4. *Let $\mathtt{p}_1, \mathtt{p}_2$ be two data properties in $\mathtt{G}$, whose source (or target) cliques are C_1 and C_2. Properties $\mathtt{p}_1, \mathtt{p}_2$ are in the same source (respectively, target) clique C^{∞} of $\mathtt{G}^{\infty}$ if and only if there exist k non-empty source (respectively, target) cliques of $\mathtt{G}$, $k \geq 0$, denoted $D_1, \ldots, D_k$ such that:*

$$C_1^+ \cap D_1^+ \neq \emptyset,$$
$$D_1^+ \cap D_2^+ \neq \emptyset, \ldots, D_{k-1}^+ \cap D_k^+ \neq \emptyset, D_k^+ \cap C_2^+ \neq \emptyset.$$

Based on Lemma 1, we show:

Lemma 2 (Property relatedness in $\mathtt{W}$ summaries [44]) *Data properties are target-related (respectively, source-related) in $(\mathtt{G}_{/\mathtt{W}})^{\infty}$ iff they are target-related (respectively, source-related) in $\mathtt{G}^{\infty}$.*

Now, recall from Lemma 1 that:

$$SC_{\mathtt{W}}^{\infty} = (SC_{\mathtt{W}}^1)^+ \cup (SC_{\mathtt{W}}^2)^+ \cup \ldots \cup (SC_{\mathtt{W}}^m)^+ \text{ and}$$
$$TC_{\mathtt{W}}^{\infty} = (TC_{\mathtt{W}}^1)^+ \cup (TC_{\mathtt{W}}^2)^+ \cup \ldots \cup (TC_{\mathtt{W}}^n)^+$$

for some $\mathtt{G}_{/\mathtt{W}}$ source cliques $SC_{\mathtt{W}}^1, \ldots, SC_{\mathtt{W}}^m$ and target cliques $TC_{\mathtt{W}}^1$, $\ldots, TC_{\mathtt{W}}^n$.

Lemma 2 ensures that the data properties in $(SC_{\mathtt{W}}^1)^+ \cup \ldots \cup (SC_{\mathtt{W}}^m)^+$ are related in $\mathtt{G}^{\infty}$, and those of $(TC_{\mathtt{W}}^1)^+ \cup \ldots \cup (TC_{\mathtt{W}}^n)^+$ are related in $\mathtt{G}^{\infty}$.

Moreover, $n_{\mathtt{W}}$ was created in $\mathtt{G}_{/\mathtt{W}}$ from a set of weakly-equivalent $\mathtt{G}$ nodes having as source clique one among $SC^1_{\mathtt{W}}, \ldots, SC^m_{\mathtt{W}}$ and as target clique one among $TC^1_{\mathtt{W}}, \ldots, TC^n_{\mathtt{W}}$. In $\mathtt{G}^\infty$, these nodes connect the data properties of $SC^\infty_{\mathtt{W}}$ with those of $TC^\infty_{\mathtt{W}}$.

Finally, Theorem 3 follows directly from the above discussion, Theorem 1, and the proposition below:

Proposition 3 (Same cliques-W [44]) *$\mathtt{G}^\infty$ and $(\mathtt{G}_{/\mathtt{W}})^\infty$ have identical source clique sets, and identical target cliques sets. Further, a node $n \in \mathtt{G}^\infty$ has exactly the same source and target clique as $f_{\mathtt{W}}(n)$ in $(\mathtt{G}_{/\mathtt{W}})^\infty$.*

Theorem 4 ($\mathtt{S}$ shortcut [44]) *The shortcut holds for $\equiv_{\mathtt{S}}$.*

We prove this based on counterparts of statements established for $\mathtt{G}_{/\mathtt{W}}$. First, we show Lemma 3:

Lemma 3 (Property relatedness in $\mathtt{S}$ summaries [44]) *Data properties are target-related (respectively, source-related) in $(\mathtt{G}_{/\mathtt{S}})^\infty$ iff they are target-related (respectively, source-related) in $\mathtt{G}^\infty$.*

Then, from Theorem 1 and the above Lemma 3, we obtain the next proposition:

Proposition 4 (Same cliques-S [44]) *$\mathtt{G}^\infty$ and $(\mathtt{G}_{/\mathtt{S}})^\infty$ have identical source clique sets, and identical target clique sets. Further, a node $n \in (\mathtt{G}_{/\mathtt{S}})^\infty$ has exactly the same source and target clique as $f_{\mathtt{S}}(n)$ in $(\mathtt{G}_{/\mathtt{S}})^\infty$.*

Theorem 4 follows directly from Proposition 4.

Finally, we have:

Theorem 5 (No shortcut for $\equiv_{\mathtt{TW}}$ [44]) *The shortcut does not hold for $\equiv_{\mathtt{TW}}$.*

We prove this by exhibiting in Figure 18 a counter-example. In $\mathtt{G}$ and $\mathtt{G}_{/\mathtt{TW}}$, all data nodes are untyped; only after saturation a node gains the type C. Thus, in $\mathtt{G}_{/\mathtt{TW}}$, one (untyped) node represents all data property subjects; this is exactly a "hasty fusion" as discussed below Theorem 2. In $(\mathtt{G}_{/\mathtt{TW}})^\infty$, this node gains a type, and in $((\mathtt{G}_{/\mathtt{TW}})^\infty)_{/\mathtt{TW}}$, it is represented by a single node. In contrast, in $\mathtt{G}^\infty$, r_1 is typed and r_2 isn't, leading to two distinct nodes in $(\mathtt{G}^\infty)_{/\mathtt{TW}}$. This is not strongly isomorphic with $(\mathtt{G}_{/\mathtt{TW}})^\infty$ which, in this example, is strongly isomorphic to $((\mathtt{G}_{/\mathtt{TW}})^\infty)_{/\mathtt{TW}}$. Thus, the shortcut does not hold for $\equiv_{\mathtt{TW}}$.

Theorem 6 (No shortcut for $\equiv_{\mathtt{TS}}$ [44]) *The shortcut does not hold for $\equiv_{\mathtt{TS}}$.*

The graph in Figure 18 is also a shortcut counter-example for $\mathtt{TS}$.

Based on Theorem 2, we have also established:

Theorem 7 (Bisimilarity shortcut [44]) *The shortcut holds for the forward ($\equiv_{\mathtt{fw}}$), backward ($\equiv_{\mathtt{bw}}$), and forward-and-backward ($\equiv_{\mathtt{fb}}$) bisimilarity equivalence relations (recalled in Section 3.2).*

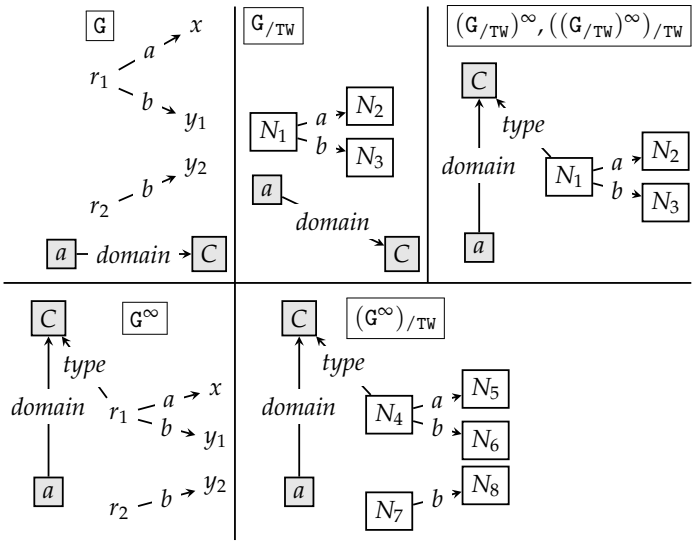


Figure 18: Shortcut counter-example.

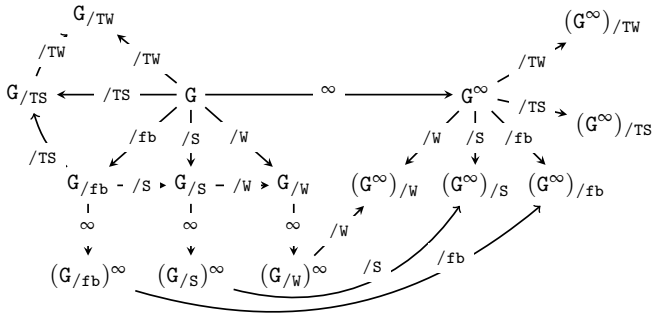


Figure 19: Relations between quotient summaries.

3.5.4 *Relationships between summaries*

From the definition of the weak and strong equivalences, it is easy to show that $(G_{/S})_{/W} = G_{/W}$, i.e., one could compute $G_{/W}$ by first summarizing G into $G_{/S}$, and then applying weak summarization on this (typically much smaller) graph; similarly, $(G_{/TS})_{/TW} = G_{/TW}$. It is also the case that $(G_{/W})_{/S} = G_{/W}$, i.e., strong summarization cannot compress a weak summary further, and similarly $(G_{/TW})_{/TS} = G_{/TW}$. Figure 19 summarizes the main relationships between G, G^∞, our summaries and bisimilarity-based ones.

3.6 FROM SUMMARIES TO VISUALIZATIONS

We now describe how to go from a quotient summary to a graphical visualization such as the one illustrated in the Introduction.

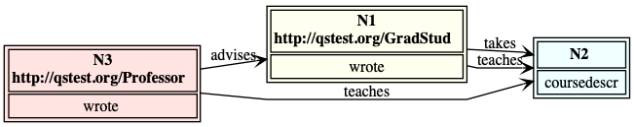


Figure 20: Visualization resulting from leaf and type inlining on the sample strong summary from Figure 13.

3.6.1 *Leaf and type inlining*

For structurally simple graphs like our sample G shown in Figure 3, quotient summaries have very few nodes and edges, and any node-link visualization method can be used. We explain here how we obtained our visualizations, illustrated in our *online gallery* [33].

To further simplify summaries, we apply *leaf and type inlining*, as follows. We remove type edges; instead, each type attached to a node in the summary is shown in the box corresponding to the node, after the node ID. Similarly, for each edge $n \xrightarrow{a} m$ where m is a *leaf*, we include a as an "attribute" of n, and do not render m (we say it has been "inlined" within n). A sizable part of an RDF graph's nodes are leaves; as we will show, inlining them into their parent nodes greatly simplifies the visualization.

Figure 20 illustrates inlining for the S summary (Figure 13) of our sample graph. This summary is extremely compact, yet rich with information; professors, students, and courses are visible at a glance. Articles have been inlined within their authors as they were leaves in $G_{/S}$. This simplification can also be seen as a small loss of information: Figure 20 does not immediately suggest that Professors may have written articles together with GradStudents. However, only leaf nodes are folded and after a first glance, users may pursue exploration by other means (e.g., queries to check for such joint articles). Thus, we consider that inlining is overall beneficial, and *systematically apply it on summaries before visualization*.

If type-then-data summarization is used based on the most general types (Section 3.5.1), the most general types are shown at the top of each typed summary node (immediately under the node ID), then the actual types of the graph nodes represented by the summary node are shown one per line, under the most general types. N_1, N_7, and N_8 in Figure 6 illustrate this.

3.6.2 *Summary statistics*

If users are (also) interested in a quantitative view of an RDF graph, our summaries can also plot a set of statistics. Below, we describe and illustrate them based on Figure 6. For each summary node N_i, we display:

- The number of G nodes represented by N_i, in parenthesis after "N_i" in the corresponding box, e.g., "N_8 (25000)";
- For each type c such that x *rdf:type* c for some x represented by N_i, the number of G nodes represented by N_i which are of type c, e.g., `http://db.uwaterloo.ca/~galuc/wsdbm/ProductCategory0:807`;
- For each data property p such that x p y for some x represented by N_i and all the objects of p triples whose subjects are represented by N_i are leaves in G, the number of such p triples, and the number of distinct targets of such triples. For instance, within N_8, "bookedition (847 $\rightarrow$ 6)" denotes that there are 847 bookedition triples whose subjects are represented by N_8, and they reach a total of 6 distinct objects (which are leaf nodes).

For each summary edge $N_i \xrightarrow{a} N_j$ where a is a data property, we display the number of x a y triples in G such that x is represented by N_i and y is represented by N_j. The label "hasGenre (58787)" on the edge from N_8 to N_1 is an example of such an edge statistic.

All these statistics can be gathered by our summary construction algorithms (Section 3.7), at no extra computational cost.

3.6.3 *Visualizing very large summaries*

For a very complex (e.g., encyclopedic) dataset, even the graph obtained from inlining may have too many nodes and edges for an effective visualization. If it has several connected components (one per domain), each of them can be viewed separately. Otherwise, the inlined graph can be split into several, possibly overlapping subgraphs, using any graph decomposition strategy (for instance, minimize the number of times the representatives of two nodes connected in G appear in different summary subgraphs, etc.). Information discovery in such graphs requires more computational and cognitive effort.

3.7 SUMMARIZATION ALGORITHMS

We now present summarization algorithms that, given as input a graph G, construct $G_{/W}$, $G_{/S}$, $G_{/TW}$, and $G_{/TS}$. We devise two types of algorithms: centralized and parallel ones. Further, all our centralized algorithms have an *amortized linear complexity in the size of* G: they can be built in just one (incremental) or two passes (global) over the data. Our incremental algorithms, capable of reflecting additions to G into its previously computed summaries, are the most involved.

3.7.1 *Global data graph summarization*

We begin by presenting algorithms that summarize *only the data triples* through *two graph traversals*. The first pass allows us to learn the equivalence relation and create the summary nodes. Our equivalence relations are defined *based on the triples of a given graph* G, thus when summarization starts, we do not know whether any two nodes are equivalent; the full equivalence relation is known only after inspecting all G triples. In the second one, we determine the representative of each G node and, as a consequence, add triples to the summary.

Our *global* W *summarization algorithm* (Algorithm 1) exploits Proposition 1, which guarantees that any data property occurs only once in the summary. To each data property p encountered, it associates a summary node (integer) s_p which will be the (unique) source of p in the summary, and similarly a node t_p target of p; these are initially unknown, and evolve as G is traversed. Further, the algorithm uses two maps op and ip that associate, with each data node n, the set of its outgoing, respectively, incoming data properties. These are filled during the first traversal of G (Step 1.). Steps 2. to 2.5 ensure that for each node n having outgoing properties and possibly incoming ones, s_p for all the outgoing ones are equal, and are also equal to t_p for all the incoming ones. This is performed using a function $fuse$ which, given a set of summary nodes, picks one that will replace all of them. In our implementation, summary nodes are assigned integer IDs, and $fuse$ is simply min; we just need $fuse$ to be distributive over $\cup$, i.e., $fuse(A, (B \cup C)) = fuse(fuse(A, B), fuse(A, C))$. Symmetrically, Step 3. ensures that the incoming properties of nodes lacking outgoing properties (thus, absent from op) also have the same target. In Step 4., we represent s and o based on the source/target of the property p connecting them. The $fuse$ operations in 2. and 3. have ensured that, while traversing G triples in 4., any data node n is always represented by the same summary node $f_W(n)$.

Our *global* S *summarization algorithm* (Algorithm 2) uses two maps sc and tc which store for each data node n, its source clique $sc(n)$, and its target clique $tc(n)$, and for each data property p, its source clique src_p and target clique trg_p. Further, for each (source clique, target clique) pair encountered during summarization, we store the (unique) corresponding summary node. Steps 1.-1.2. build the source and property cliques present in G and associate them to every subject and object node (in sc and tc), as well as to any data property (in src_p and trg_p). For instance, on the sample graph in Figure 3, these steps build the cliques in Table 3. Steps 2-2.2. represent the nodes and edges of G.

The correctness of algorithms **global-W** and **global-S** follows quite easily from their descriptions and the summary definitions.

Algorithm 1: Global W summarization of a graph

global-W(G)

1. For each $\mathtt{s}\ \mathtt{p}\ \mathtt{o} \in \mathtt{G}$:
1.1. Add p to $op(\mathtt{s})$ and to $ip(\mathtt{o})$.
2. For each node $n \in op$:
2.1. Let $X \leftarrow fuse\{s_\mathtt{p} \mid \mathtt{p} \in op(n)\}$.
2.2. If X is undefined:
2.2.1. Let $X \leftarrow$nextNode().
2.3. Let $Y \leftarrow fuse\{t_\mathtt{p} \mid \mathtt{p} \in ip(n)\}$.
2.4. If Y is undefined:
2.4.1. Let $Y \leftarrow$nextNode().
2.5. Let $Z \leftarrow fuse(X, Y)$.
2.6. For each $\mathtt{p} \in ip(n)$:
2.6.1. Let $s_\mathtt{p} \leftarrow Z$.
2.7. For each $\mathtt{p} \in op(n)$:
2.7.1. Let $t_\mathtt{p} \leftarrow Z$.
3. Repeat **2** to **2.7** swapping ip with op and $t_\mathtt{p}$ with $s_\mathtt{p}$.
4. For each $\mathtt{s}\ \mathtt{p}\ \mathtt{o} \in \mathtt{G}$:
4.1. Let $f_\mathtt{W}(\mathtt{s}) \leftarrow s_\mathtt{p}$, $f_\mathtt{W}(\mathtt{o}) \leftarrow t_\mathtt{p}$.
4.2. Add $f_\mathtt{W}(\mathtt{s})\ \mathtt{p}\ f_\mathtt{W}(\mathtt{o})$ to $\mathtt{G}_{/\mathtt{W}}$.

Algorithm 2: Global S summarization of a graph

global-S(G)

1. For each $\mathtt{s}\ \mathtt{p}\ \mathtt{o} \in \mathtt{G}$:
1.1. Check if $src_\mathtt{p}$, $trg_\mathtt{p}$, $sc(\mathtt{s})$ and $tc(\mathtt{o})$ are known; those not known are initialized with $\{\mathtt{p}\}$.
1.2. If $sc(\mathtt{s}) \neq src_\mathtt{p}$, fuse them into a new clique $src'_\mathtt{p} = sc(\mathtt{s}) \cup src_\mathtt{p}$; similarly, if $tc(\mathtt{o}) \neq trg_\mathtt{p}$, fuse them into $trg'_\mathtt{p} = tc(\mathtt{o}) \cup trg_\mathtt{p}$.
2. For each $\mathtt{s}\ \mathtt{p}\ \mathtt{o} \in \mathtt{G}$:
2.1. $f_\mathtt{S}(\mathtt{s}) \leftarrow$ the (unique) summary node corresponding to the cliques $(sc(\mathtt{s}), tc(\mathtt{s}))$; similarly, $f_\mathtt{S}(\mathtt{o}) \leftarrow$ the node corresponding to $(sc(\mathtt{o}), tc(\mathtt{o}))$ (create the nodes if needed).
2.2 Add $f_\mathtt{S}(\mathtt{s})\ \mathtt{p}\ f_\mathtt{S}(\mathtt{o})$ to $\mathtt{G}_{/\mathtt{S}}$.

3.7.2 *Incremental data graph summarization*

These algorithms are particularly suited for incremental *summary maintenance*: if new triples $\Delta^+_\mathtt{G}$ are added to G, it suffices to summarize only $\Delta^+_\mathtt{G}$, based on $\mathtt{G}_{/\equiv}$ and its representation function $f_\equiv$, in order to obtain $(\mathtt{G} \cup \Delta^+_\mathtt{G})_{/\equiv}$. Incremental algorithms are considerably *more complex*, since various decisions (assigning sources/targets to properties in W, source/target cliques in S, node representatives in both) must

Algorithm 3: Incremental W summarization of one triple

increm-W(s p o)

1. Check if s_p and o_p are known: either both are known (if a triple with property p has already been traversed), or none;
2. Check if $f_W(s)$ and $f_W(o)$ are known; none, one, or both may be, depending on whether s, respectively, o have been previously encountered;
3. Fuse s_p with $f_W(s)$ (if one is unknown, assign it the value of the other), and o_p with $f_W(o)$;
4. Update $f_W(s)$ and $f_W(o)$, if needed;
5. Add the edge $f_W(s)$ p $f_W(o)$ to $G_{/W}$.

be *repeatedly revisited* to reflect newly acquired information about G triples, as we shall see.

Each *incremental summarization* algorithm consists of an *incremental update method, called for every data triple*, which adjusts the summary's data structures, so that at any point, the summary reflects exactly the graph triples visited until then.

Incremental weak summarization (**increm-W**) is outlined in Algorithm 3. For example (see the figure below), let's assume the algorithm traverses the graph G in Figure 3 starting with: p_1 *advises* p_2, then p_1 *wrote* a_1, then p_4 *teaches* c_2. Then:

- When we summarize this third triple, we do not know yet that p_1 is equivalent to p_4, because no common source of *teaches* and *advises* (e.g., p_3 or p_4) has been seen so far. Thus, p_4 is found not equivalent to any node visited so far, and represented separately from p_1.
- Now, assume the fourth triple traversed is p_4 *advises* p_5: at this point, we know that *advises*, *wrote* and *teaches* are in the same source clique, thus $p_1 \equiv_W p_4$, and their representatives (highlighted in yellow) must be *fused* in the summary (Step 3.).
- More generally, it can be shown that the relation $\equiv_W$ *only grows* as more triples are visited; in other words: if in a subset G' of G's triples, two nodes n_1, n_2 are weakly equivalent, then this holds in any G'' with $G' \subseteq G'' \subseteq G$.

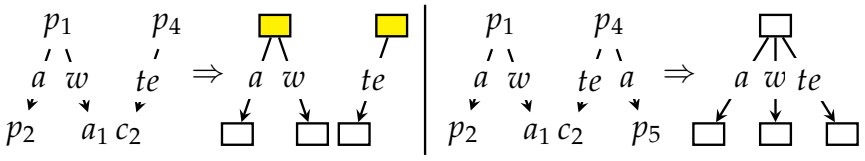


Figure 21: Incremental weak summary node fusion after the addition of a new triple.

Algorithm 4: Incremental S summarization of one triple

increm-S(s p o)

1. Check if we already know a source clique src_p (respectively, target clique trg_p).Either both are known (if a p triple has already been traversed), or none. Those not known are initialized with {p};
2. Check if $sc(s)$ (respectively, $tc(o)$) are known; those unknown are initialized with {p};
3. If $sc(s) \neq src_p$, fuse them into new clique $src'_p = sc(s) \cup src_p$, using Union-Find; similarly, if $tc(o) \neq trg_p$, fuse them into $trg'_p = tc(o) \cup trg_p$, and:
3.1 Replace $sc(s)$ and src_p with src'_p throughout the summary (respectively, replace $tc(o)$ and trg_p with trg'_p);
3.2 The above may entail summary node fusions; in this case, update f_S (use Union-Find) and the summary edges to reflect it;
4. If before seeing s p o s had been already represented and it had an empty source clique, then s needs to split, i.e., be represented separately from the nodes to which it was $\equiv_S$ previously; call **split-source(s)**. (Symmetric discussion for o, call **split-target(o)**).
5. Update $f_S(s)$ and $f_S(o)$, if needed;
6. Add the edge $f_S(s)$ p $f_S(o)$ to $G_{/S}$.

Summary node *fusion* dominates the algorithm's *complexity*. Let N_1, N_2 be two sets of G nodes, represented at a certain point by the distinct summary nodes m_1, m_2. When fusing them into a single m, we must also record that all the nodes in $N_1 \cup N_2$ are now represented by m. A naïve implementation leads to $O(N^2)$ complexity, where N is the number of data nodes, since each new node may lead to a fusion whose cost is $O(N)$; in the worst case N could be proportional to $|G|$ (the number of triples in G) leading to an overall complexity of $O(|G|^2)$ for the incremental weak summarization.

Instead, we rely on a *Union-Find* [45] (a.k.a. Disjoint Sets) data structure, with the *path compression* and *union by size* optimizations, which guarantee an overall *quasi-linear worst-case complexity* of our incremental weak summarization algorithm. The exact complexity is $O(N\alpha(N))$ where $\alpha(N)$, the inverse Ackermann's function, is smaller than 5 for any machine-representable input N. Assimilating this to *linear-time*, the algorithm's complexity class is in $O(|G|)$, which is also *optimal* because of a trivial lower bound of $\Omega(|G|)$ as summarization must fully traverse G.

Incremental strong summarization (**increm-S**) is outlined in Algorithm 4 through the incremental update of the S summary due to the traversal of the triple s p o. Conceptually, the algorithm is symmetric

Procedure 5: Splitting summary node s on source

split-source(s)

1. Collect all G edges adjacent to s into $transfer$ set.
2. For each s p o $\in transfer$, decrement by 1 the counter for f_S(s) p f_S(o) in the summary.
3. Update f_S(s).
4. For each s p o $\in transfer$, if such edge already exists in the summary, then increment its counter by 1, otherwise add f_S(s) p f_S(o) to the summary with counter equal to 1.

for the source (s) and target (o) of the edge, we only discuss the source side below:

- Steps 1. and 2. start by determining the source clique of s, based on its previously known source clique (if any) and the previously known target clique of p (if any).
- After Step 2., s's source (and target) clique, reflecting also the newly seen triple s p o, are completely known. Determining them may have involved fusing some previously separate cliques.
- For instance, on the graph in Figure 3, assume we first traverse the triple $p1$ *advises* p_2, then p_4 *teaches* c_2; so far we have the source cliques $\{advises\}$, $\{teaches\}$ and $\emptyset$. If the next traversed triple is p_4 *advises* p_5, we fuse the source cliques (Step 3.1) $\{advises\}$ and $\{teaches\}$ into $\{advises, teaches\}$. This requires fusing the summary node whose (source, target) cliques were $(\{advises\}, \emptyset)$ with the one that had $(\{teaches\}, \emptyset)$ (Step 3.2).

NON-MONOTONICITY OF STRONG EQUIVALENCE The last intricacy of incremental strong summarization is due to the fact that unlike $\equiv_W$, the relation $\equiv_S$ *may grow and shrink* during summarization. For instance, assume incremental strong summarization of the graph in Figure 3 starts with p_1 *wrote* a_1, p_2 *wrote* a_2, p_2 *takes* c_2 (see the figure below). After these, we know $p_1 \equiv_S p_2$; their source clique is $\{wrote, takes\}$ and their target clique is $\emptyset$. Assume the next triple traversed is p_3 *advises* p_2: at this point, p_1 is *not* $\equiv_S$ *to* p_2 *any more*, because p_2's target clique is now $\{advises\}$ instead of the empty $\emptyset$. Thus, p_2 *splits* from p_1, that is, it needs to be represented by a new summary node (shown in yellow below), distinct from the representative of p_1.

Further, note that the representative of p_1 and p_2 (at left above) had one *takes* edge (highlighted in red) which was *solely due to* p_2*'s outgoing takes edge*. By definition of a quotient summary (Section 3.2), that edge *moves* from the old to the new representative of p_2 (the yellow node). If, above at left, p_1 had also had an outgoing edge labeled *takes*, at right, both nodes in the top row would have had

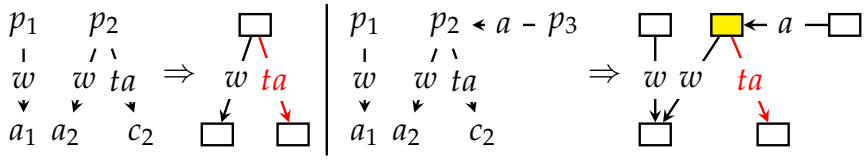


Figure 22: Incremental strong summary node split after the addition of a new triple.

an outgoing *takes* edge. It can be shown that *splits only occur in such cases*, i.e., o whose target clique becomes non-empty (respectively, s whose source clique becomes non-empty, and the node was previously represented together with other nodes; if it was represented alone, we just update the respective clique of its representative).

The procedure **split-source(s)** (Procedure 5) represents s separately, to reflect that it no longer has an empty target clique, and, for each outgoing edge of s, adds a corresponding edge to the new representative of s and checks if, as a consequence, an edge needs to be removed from its previous representative. We establish:

Proposition 5 (Algorithm correctness) *Applying algorithm* ***increm-W*** *(respectively,* ***increm-S****) successively on each triple of* G*, in any order, builds* $G_{/W}$ *(respectively,* $G_{/S}$*).*

Proof *All our algorithms (global or incremental) start by identifying the class and property nodes: this is done retrieving all the subjects and objects from schema triples, and also all the objects of type triples. As previously stated, triple stores routinely support such retrieval efficiently. Our algorithms start by representing these special schema nodes exactly by themselves, and copying in the summary all the schema triples. This exploits the observation made in Section 3.5 (*G *and* $G_{/\equiv}$ *have the same schema triples).*

Below, we show the correctness of incremental W and S summarization *on data triples. The proof of Proposition 6 (below) extends this also to type triples.*

The correctness of incremental W summarization *on data triples follows from the fact that the* ***increm-W*** *algorithm preserves a set of invariants. Let* G_k *be the first k triples of* G*, in the order in which they are traversed by the algorithm. For any* $1 \leq k \leq |G|$*, after applying* ***increm-W*** *on k data triples, the following invariants are preserved:*

1. *The source and target* src_p *and* trg_p *of any property* p *present in these k triples are known.*

2. *For any summarized triple* s p o*, we have* $f_W(s) = src_p$ *and* $f_W(o) = trg_p$*; further, the summary contains the edge* $f_W(s)$ p $f_W(o)$*.*

The preservation of these invariants is shown by considering all the cases which may occur for a given summarized triple s p o*: the subject* s *may have* already been seen *(in which case this triple may lead to a fusion),* or not

(in which case we create the new representative of $\mathtt{s}$*), and similarly for* $\mathtt{o}$*. For* $\mathtt{p}$ *there are also two cases (depending on whether we had already encountered it or not, we may create* $src_\mathtt{p}$ *and* $trg_\mathtt{p}$*, or just fuse them with preexisting representatives of* $\mathtt{s}$ *and* $\mathtt{o}$*). There are* 8 *cases overall. The replacements and fusions detailed in Algorithm 3 guarantee these invariants.*

For simplicity of presentation, the ***increm-W*** *algorithm considers the possible fusions due to* $\mathtt{s}$ *and* $\mathtt{o}$ *separately. However, in reality, given that they may impact the same node(s) (e.g., if* $f_\mathtt{W}(\mathtt{s}) = f_\mathtt{W}(\mathtt{o})$*), all the replacements are first computed, then reconciled into a* list of summary node substitutions*, applied in all the data structures. For instance, suppose we need to replace summary node* 3 *with* 1 *because of a fusion on the subject side, and also summary node* 5 *with* 3 *because of a fusion on the object side. In this case, the algorithm will replace* 5 *and* 3 *directly with* 1*. If the replacements were applied sequentially, e.g., first* 3 *with* 1*, the second replacement would leave* 3 *(not* 1*) instead of* 5*, which would be an error.*

Similarly, the correctness of incremental S summarization *on data triples follows from the fact that the* ***increm-S*** *algorithm preserves the following invariants after having been called on* k *successive data triples, with* $1 \leq k \leq |\mathtt{G}|$*:*

1. *The source and target clique* $sc(\mathtt{p})$ *and* $tc(\mathtt{p})$ *of any property* $\mathtt{p}$ *present in these* k *triples are known, and they contain* $\mathtt{p}$*.*
2. *For any summarized triple* $\mathtt{s}\ \mathtt{p}\ \mathtt{o}$*, we have* $f_\mathtt{S}(\mathtt{s}) = sc(\mathtt{p})$ *and* $f_\mathtt{S}(\mathtt{o}) = tc(\mathtt{p})$*; further, the summary contains the edge* $f_\mathtt{S}(\mathtt{s})\ \mathtt{p}\ f_\mathtt{S}(\mathtt{o})$*.*
3. *For any source clique* sc *and target clique* tc *of a node* n *appearing in the summarized triple, the summary contains exactly one node.*
4. *For any summary node* m*, the count* $m_\sharp$ *is exactly the cardinality of the set* $\{n \in \mathtt{G} \mid f_\mathtt{S}(n) = m\}$*.*
5. *For any summary edge* $m \xrightarrow{\mathtt{p}} m'$*, the count* $e_\sharp$ *is exactly the cardinality of the set* $\{n \xrightarrow{\mathtt{p}} n'$ *edge of* $\mathtt{G} \parallel f_\mathtt{S}(n) = m$ *and* $f_\mathtt{S}(n') = m'\}$*.*
6. *For any (subject, property) combination occurring in the summarized triples, the count* $(\mathtt{sp})_\sharp$ *is exactly the number of times this occurred in the triples. Similarly, for any (property, object) combination appearing in the summarized triples, the count* $(\mathtt{po})_\sharp$ *is exactly the number of times it appeared.*

Like for ***increm-W****, there are eight cases depending on whether* $\mathtt{s}$*,* $\mathtt{p}$ *and* $\mathtt{o}$ *have been previously seen. Further, in the four cases where* $\mathtt{s}$ *has been seen, we may need to split* $\mathtt{s}$*'s representative, or not, and similarly for* $\mathtt{o}$*; thus, the six original cases where at least one of them had been seen lead to* 12 *cases (to which we add the remaining two, where neither* $\mathtt{s}$ *nor* $\mathtt{o}$ *had been seen), for a total of* 14 *cases.*

Items 4, 5 and 6 are ensured during: the addition of an edge to the summary (this sets $e_\sharp$ *to* 1 *or increases it); the assignment of representatives to nodes*

(this sets $m_\sharp$ to 1 or increments it); the edge repartition during split (this subtracts from one edge $e_\sharp$ exactly the count that it adds to another new edge); and node replacements (which, when replacing u with v, either carry $u_\sharp$ into $v_\sharp$, if v did not exist in the summary previously, or add $u_\sharp$ to $v_\sharp$ if it did). Together, 4, 5 and 6 ensure the correctness of the ***split*** *procedure (explained in Section 3.7.2).*

The previous items are ensured by the creation of summary nodes (at most one exists at any time for a given source and target clique), fusing cliques (this guarantees each property is in the right clique, and remove cliques input to the fusion), and replacing / fusing summary nodes, as well as from the correctness of the split procedure.

Splitting requires inspecting the data edges attached to the node, in order to add edges to its new representative (such as p_2 *ta* c_2 above). We make an assumption, denoted $(\star)$, that the average number of edges incoming/outgoing a data node is small (and basically constant) compared to the size of G; this assumption holds for the graphs we have experimented with. Under the $(\star)$ assumption, using the above data structures (including Union-Find), *the complexity of incremental strong summarization is amortized constant per added triple*.

All our algorithms require $O(|\mathtt{G}|)$ space to store the summary, the representation function, and their other data structures. Encoding all graph and summary nodes as integers, however, reduces the actual memory needs significantly.

3.7.3 *Global and incremental typed graph summarization*

We now explain how to extend our incremental data triple summarization algorithms to type triples.

To extend W, respectively, S summarization to type triples in "data-then-type" fashion (Section 3.4.1), we run W, respectively, S summarization *first, over the data triples only*, as described in the preceding two sections. This assigns their (final) representatives to all data nodes. Then, for each s *rdf:type* c triple, we simply add to the summary the edge $f_\mathtt{W}(\mathtt{s})$ *rdf:type* c (respectively, $f_\mathtt{S}(\mathtt{s})$ *rdf:type* c); recall from Section 3.4 that any class node C is represented by itself.

For "type-then-data" summarization (Section 3.4.2), *we first traverse the type triples only*, compute all the class sets, and assign to each typed data node a representative based on its class set. Then, we run a *type-aware variant* of a W (respectively, S) algorithm, either global or incremental. The differences are:

1. In TW summarization, a data property p may lack an untyped source (and/or target), if p has only typed subjects (respectively, objects), e.g., a property e in the two-triples graph n_1 *rdf:type* c, n_1 e a_1. Similarly, in TS summarization, a property like e will lack a source clique, if it does not have an untyped source.

Whenever a clique-based representative for a property's source and target is missing, the algorithms will instead use the type-based representative, e.g., the representative of n_1 in n_1 *e* a_1 will be the one for the type set $\{C\}$.

2. Summarizing the data triple s p o does not fuse nor split the representative of s (respectively, o), if s (respectively, o) is typed; the representatives of typed nodes never change.

We establish:

Proposition 6 (Algoritm correctness) *Applying* ***global*****-**W *(respectively,* ***global*****-**S*) on* G*, or applying* ***increm*****-**W *(respectively,* ***increm*****-**S*) on each triple of* G*, extended as described above for data-then-type or type-then-data summarization leads, respectively, to* $G_{/W}$*,* $G_{/S}$*,* $G_{/TW}$*, and* $G_{/TS}$*.*

Proof *First, recall that* TW *and* TS *summarization start with the type triples, which means all typed nodes are detected and represented according to their class sets, before the data triples are summarized. This entails that among the cases which occur for* W *and* S *summarization (*8*, respectively,* 14*, see discussion in the proof of Proposition 5), those in which the subject, respectively, the object was already represented are further divided in two, depending on whether the subject, respectively, object was a typed node.*

This shows that incremental TW summarization handles a superset of the cases handled by the W one*, and similarly for* TS *and* TS*. Thus,* ***increm*****-**TW*, respectively,* ***increm*****-**TS *preserve all the invariants of* ***increm*****-**W*, respectively,* ***increm*****-**S[6]*, with some additions, which we highlight in italics below.*

ADDITIONS OF TW SUMMARIZATION W.R.T. WEAK SUMMARIZATION

1. *The source and target* src_p *and* trg_p *of any property* p *present* with an untyped source, respectively, an untyped target *in the summarized triples are known.*

2. *For any summarized triple* s p o*, we have* $f_W(s) = src_p$ if s is untyped *and* $f_W(o) = trg_p$ if o is untyped*; further, the summary contains the edge* $f_W(s)$ p $f_W(o)$.

ADDITIONS OF TS SUMMARIZATION W.R.T. STRONG SUMMARIZATION

1. *The source and target clique* $sc(p)$ *and* $tc(p)$ *of any property* p *present in these k triples* with an untyped source, respectively, with an untyped target *are known, and they contain* p.

6 Note that in the particular case of triples connecting untyped nodes, the algorithms coincide.

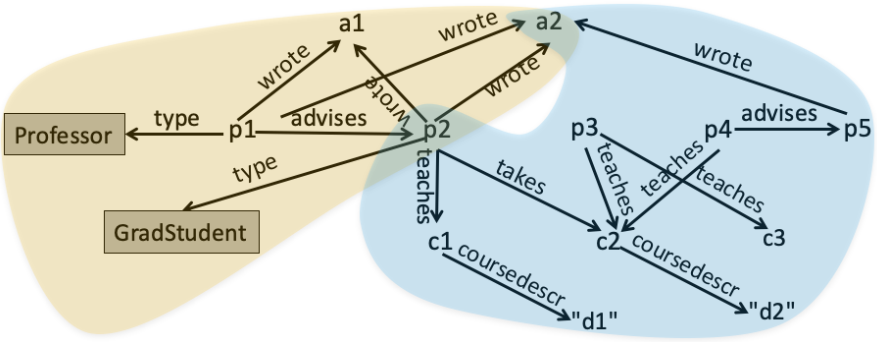


Figure 23: Sample RDF graph partitioned.

2. *For any summarized triple* $\mathtt{s\ p\ o}$, *we have* $f_{\mathtt{S}}(\mathtt{s}) = sc(\mathtt{p})$ if $\mathtt{s}$ is untyped, *and* $f_{\mathtt{S}}(\mathtt{o}) = tc(\mathtt{p})$ if $\mathtt{o}$ is untyped; *further, the summary contains the edge* $f_{\mathtt{S}}(\mathtt{s})\ \mathtt{p}\ f_{\mathtt{S}}(\mathtt{o})$.

Further, they also preserve:

1. *The summary contains one node for each set of classes belonging to some resource in the input.*

2. *For any node n with a non empty class set,* $f_{\mathtt{TW}}(n)$ *(respectively,* $f_{\mathtt{TS}}(n)$*) is the node corresponding to the class set of n.*

These invariants are ensured by the way in which we collect all class sets during the initial traversal of type triples (common to the TW *and* TS *algorithms). Further, during the* TW *and* TS *summarization, as said in Section 3.7.3, the representatives of typed nodes never fuse, and never split.*

The 6 invariants from the proof of Prop. 5 ensure the correct summarization of data triples when s *and* o *are untyped. Together with the two above, they also ensure the correct summarization of triples having a typed* s *and/or* o.

These algorithms need to work with the data and type triples *separately*. Fortunately, most popular RDF store allow such direct access. The space needed to also represent type triples remains linear in $|\mathtt{G}|$.

3.7.4 *Parallel summarization*

Lastly, we move to our novel parallel algorithms. A first idea is to simply partition G among the available nodes (machines), summarize each slice of the graph on its node, and then summarize again the union of these partial summaries to get the summary of G. Unfortunately, this may be incorrect. Assume that G is the graph in Figure 3, and that it has been partitioned into two machines, $\mathtt{G}_1$ and $\mathtt{G}_2$, as shown in Figure 23 with orange and blue regions, respectively. Notice that all *teaches* triples are in $\mathtt{G}_2$. Summarizing the two subgraphs separately, we obtain $(\mathtt{G}_1)_{/\mathtt{W}}$ that has a *wrote* edge, and $(\mathtt{G}_2)_{/\mathtt{W}}$ that has a *teaches* edge. However, the information that *wrote* and *teaches* had common

sources in $\mathtt{G}$ is lost: using only $(\mathtt{G}_1)_{/\mathtt{W}}$ and $(\mathtt{G}_2)_{/\mathtt{W}}$, one cannot compute $\mathtt{G}_{/\mathtt{W}}$. A similar reasoning holds for $\mathtt{G}_{/\mathtt{S}}$, $\mathtt{G}_{/\mathtt{TW}}$ and $\mathtt{G}_{/\mathtt{TS}}$.

Therefore, we devised new parallel algorithms for building our summaries (Sections 3.7.5, 3.7.6); they all assume a distributed storage and a MapReduce-like framework. Then, in Section 3.7.7, we show how to tailor our algorithms to the Spark framework used in our implementation. We assume the graph contains $|\mathtt{G}|$ triples and that we have M machines at our disposal. All the algorithms perform two preprocessing steps:

1. Build the sets of class and property nodes of $\mathtt{G}$, which must be preserved by summarization;
2. *Dictionary-encode* the RDF URIs and literals into integers, to manipulate less voluminous data. (This is quite standard in RDF data management works.)

3.7.5 *Parallel data graph summarization*

PARALLEL COMPUTATION OF THE STRONG SUMMARY We compute the strong summary through a sequence of parallel processing jobs as follows.

S1. We distribute *all (data and type)* triples of input graph equally among all the machines, e.g., using round robin approach, so that each m_i, $1 \leq i \leq M$ holds at most $\left\lceil \frac{|G|}{M} \right\rceil$ triples.

S2. In a Map job, each machine m_i for a given data triple $t = \mathtt{s\ p\ o}$ *emits* two pairs: $(\mathtt{s}, (\mathtt{source}, \mathtt{p}, \mathtt{o}))$ and $(\mathtt{o}, (\mathtt{target}, \mathtt{p}, \mathtt{s}))$, where $\mathtt{source}$ and $\mathtt{target}$ are two constant tokens (labels). *The data and type triples initially distributed to each machine $m_1, \ldots, m_M$ are kept (persisted) on that machine throughout the computation*. All other partial results produced are discarded after they are processed, unless otherwise specified.

S3. In the corresponding Reduce job, for each resource $r \in \mathtt{G}$, all the data triples whose subject or object is r are on the same machine m_i. For each such r, m_i can infer the relationships (same source clique, same target clique) that hold between the data properties of $\mathtt{G}$ appearing on incoming and outgoing edges of r. Formally, a *property relation information (or PRI, in short)* between two properties a, b of $\mathtt{G}$ states that they are in the same source clique, or that they are in the same target clique. For instance, if m_i hosts the blue triples from Figure 23, where $a = \mathit{teaches}$ and $b = \mathit{takes}$, the triples p_2 *teaches* c_1 and p_2 *takes* c_2 lead to a PRI of the form $(\mathit{teaches}, \mathit{takes}, \mathtt{source})$. We also emit PRIs for each property with itself, in order to prepare the necessary

information so that all cliques are correctly computed in the steps below (even those consisting of a single data property).

The PRIs resulting from all the data triples hosted on m_i are de-duplicated locally at m_i.

S4. Each machine *broadcasts* its PRIs to all other machines *while also keeping its own PRIs*. Observe that for k properties having, for instance, the same source, $\frac{k(k-1)}{2}$ PRIs can be produced. However, it suffices to broadcast $k-1$ among them (the others will be inferred by transitivity in Step (S6)).

S5. Based on this broadcast, each machine has the necessary information to compute the source and target cliques of $\mathtt{G}$ *locally*, and actually computes them[7].

At the end of this stage, the cliques are known *and will persist on each machine until the end of the algorithm*, but we still need to compute:

- All the (source clique, target clique) pairs which actually occur in $\mathtt{G}$ nodes,
- The representation function, and
- The summary edges.

S6. The representation function can now be locally computed on each machine as follows:

- For a given pair of source and target cliques (SC, TC), let N_{SC}^{TC} be a URI uniquely determined by SC and TC, such that a different URI is assigned to each distinct clique pairs: N_{SC}^{TC} will be the URI of the $\mathtt{G}_{/\mathtt{S}}$ node corresponding these source and target cliques.
- For each resource r stored on m_i, the machine identifies the source clique SC_r and target clique TC_r of r, and creates (or retrieves, if already created) the URI $N_{SC_r}^{TC_r}$ of the node representing r in $\mathtt{G}_{/\mathtt{S}}$.

S7. Finally, we need to build the edges of $\mathtt{G}_{/\mathtt{S}}$.

a) To summarize *data triples*, for each resource r whose representative N_r is known by m_i, and each triple (hosted on m_i) of the form $r\ p\ o$, m_i emits $(o, (\mathtt{p}, N_r))$. This triple arrives on the machine m_j which hosts o and thus already knows N_o. The machine outputs the $\mathtt{G}_{/\mathtt{S}}$ triple $N_r\ \mathtt{p}\ N_o$.

7 In practice: this can be implemented, e.g., using Union-Find; and this is redundant as only one of them could have done it and broadcast the result.

b) To summarize *type triples*, for each resource r represented by N_r such that the type triple r *rdf:type* c is on m_i, the machine outputs the summary triple N_r *rdf:type* c [8] .

The above process may generate the same summary triple more than once (and at most M times). Thus, a final duplicate-elimination step may be needed.

ALGORITHM CORRECTNESS The following observations ensure the correctness of the above algorithm's stages.

- Steps (S1) to (S4) ensure that each machine has the complete information concerning data properties being source-related or target-related. Thus, each machine correctly computes the source and target cliques.
- Step (S3) ensures that each machine can correctly identify the source and target clique of the resources r which end up on that machine.
- The split of the triples in Step (S1) and the broadcast of source and target clique ensure that the last steps (computation of representation function and of the summary triples) yield the expected results.

PARALLEL COMPUTATION OF THE WEAK SUMMARY The algorithm for weak summarization starts with the steps (S1)-(S3) as above, then continues as shown below. In a nutshell, this algorithm exploits the observation that by definition of the weak summary, each data property occurs only once.

W4. Instead of PRIs, the machines emit *Unification Decisions*. A unification decision between two data properties a, b, is of one of the following forms:

- a,b have the same source node in $G_{/W}$;
- a,b have the same target node in $G_{/W}$;
- The source of a is the same as the target of b.

For instance, in the blue region of Figure 23, two triples p_2 *takes* c_2, c_2 *coursedescr* d_1 lead to the UD "the target of *takes* is the same as the source of *coursedescr*"; similarly, p_4 *teaches* c_2, p_4 *advises* p_5 lead to the UD "the source of *teaches* is the same as the source of *advises*", etc. In the above, just like for the PRIs, a and b can be the same or they can be different.

W5. Each machine broadcasts its unique set of UDs while also keeping its own. The number of UDs is bound by the number of

8 There is no need to flip the triple and send it to another map job because the object of a type triple is already known to be a class node thus represented by itself.

properties pairs in $\mathtt{G}$. Not all of the combinations are sent; a transitive closure is applied in Step (W6).

W6. Each machine has the necessary information to compute the nodes and edges of $\mathtt{G}_{/\mathtt{W}}$ as follows:

- Assume that for two sets IP, OP of incoming, respectively, outgoing data properties, we are able to compute a unique URI W_{IP}^{OP}, which is different for each distinct pair of sets.
- Build $\mathtt{G}_{/\mathtt{W}}$ with an edge for each distinct data property p in $\mathtt{G}$; the source of this edge for property is $W_{\emptyset}^{\{p\}}$, while its target is $W_{\{p\}}^{\emptyset}$. All edges are initially disconnected, that is, there are initially $2 \times P$ nodes in $\mathtt{G}_{/\mathtt{W}}$.
- Apply each UD on the graph thus obtained, gradually fusing the nodes which are the source(s) and target(s) of the various data properties. This entails replacing each W node with one reflecting all its incoming and outgoing data properties known so far.

At the end of this process, each node has $\mathtt{G}_{/\mathtt{W}}$. We still need to compute the representation function.

W7. On each machine holding a triple $r_1\ p\ r_2$, we identify the W nodes W_p, W^p in $\mathtt{G}_{/\mathtt{W}}$ which contain p in their outgoing, respectively, incoming property set. We output the $\mathtt{G}_{/\mathtt{W}}$ triple $W_p\ p\ W^p$.

W8. The type summary triples are built exactly as in Step (S7b).

3.7.6 *Parallel typed graph summarization*

We now present the changes needed by the above algorithms to compute the typed counterparts of weak and strong summaries. The changes are needed in order to reflect the different treatment of the type triples. In particular, we introduce a new constant token `type` to be sent in Step (S2). We emit pairs corresponding to type triples only in the forward direction, e.g., we send $(\mathtt{s}, (\mathtt{type}, \mathtt{p}, \mathtt{o}))$ but not $(\mathtt{o}, (\mathtt{type}, \mathtt{p}, \mathtt{s}))$. We do not emit pairs with tokens `source` nor `target` for type triples, as typed nodes *do contribute to property cliques* (recall Chapter 2). The type triples are then cached (kept) at each machine, and not used until the Step (S6) in the strong summarization algorithm, respectively, (W6) in the weak one.

To determine the representative of a typed node, each machine that received some type triples groups them by the subject and creates a temporary class set IDs based on the types it knows for each subject. Then, a step similar to (S7a) is needed to disseminate information about each such typed nodes, say n at machine m. Any machine m' which has received some triples of whom n is a subject/object, but

has no type triples about n, needs to know it is typed, thus omit it from the clique computation.

3.7.7 *Apache Spark implementation specifics*

A Spark cluster has a set of *worker nodes* and a *driver*, which coordinates the run of an application and communicates with cluster manager (e.g., YARN scheduler). A Spark *executor* is a process working on a piece of data in a worker node. Each Spark application consists of *jobs* that are divided into *stages*, each divided into *tasks*.

BASIC SPARK TERMINOLOGY Spark relies on *Resilient Distributed Datasets (RDDs)*, which are fault-tolerant and immutable collections of data distributed among the cluster nodes. Spark also supports *broadcast variables*, i.e., collections of the data that are first gathered at the driver and then are broadcast (copies shipped over network) to all the cluster nodes. These collections are immutable and can only be used for local lookups.

ADAPTING OUR ALGORITHMS TO SPARK We adapt our parallel algorithms to Spark's RDD-based computation model as follows. Each step of an algorithm consumes an RDD and builds another one. First of all, we load the input graph into an RDD called `graph`. Then, we preprocess it in order to create the RDDs: `dictionary`, `reverseDictionary`, `nonDataNodesBlocklist` and `encodedTriples`. `dictionary` maps `G` nodes to their integer encodings, `reverseDictionary` is its reverse map and `encodedTriples` are the `iG` triples encoded into integers. We collect the class and property nodes in `schema-Nodes` RDD; this (very small) collection is broadcast to all nodes.

Then, we create an RDD called `nodesGrouped` that is a map from `G` nodes to the set of their incoming and outgoing edges. Next, in the weak algorithm we create a `unificationDecisions` RDD that is a collection of unification decisions; in the strong algorithm we create a respective `propertyRelationInformation` RDD. For those two RDDs we exclude (prefilter) class and property nodes, and in case of typed summaries we exclude typed nodes too.

The RDDs `unificationDecisions` and `propertyRelation-Information` are gathered at the driver, which computes the source and target cliques of `G`. The driver then broadcasts the source and target clique IDs of each property to all the nodes. This allows to create the summary nodes and to create the summary edges. The algorithms that build $G_{/S}$, $G_{/TW}$ and $G_{/TS}$ also use an RDD called `representationFunction`, which stores a mapping between the input graph node and its summary representative. This map is filled by each algorithm, and then used on each machine to emit, for each

triple of the form $n\ p\ m$ that it stores, the corresponding summary triple $n_{rep}\ p\ m_{rep}$, where n_{rep}, m_{rep} are the representatives of n and m, respectively. The weak summarization algorithm does not need this map; as explained in Section 3.7.5 (Step (W6) and below), it can create the summary triple representing $n\ p\ m$ directly based on p.

Finally, in all the algorithms we need to decode the properties (edge labels). We do it by joining the encoded summary triples with the `reverseDictionary` in order to replace each encoded property with its full value from `G`.

3.8 CENTRALIZED SUMMARIZATION EXPERIMENTS

COMPUTATIONAL ENVIRONMENT We ran all experiments on an Intel Xeon CPU E5-2640 v4 @ 2.40GHz, 40 cores (2 sockets with 10 physical cores each, hyper-threading enabled), running CentOS 7 with 90GB for JVM (OpenJDK 1.8) and 30GB of shared buffers for PostgreSQL 9.6 (with 640 MB working memory).

SYSTEMS We implemented the centralized algorithms in Java 1.8 (14.5k lines of code) and made them available online as the RDFQuotient tool [33]. RDFQuotient relies on OntoSQL 1.0.12, an efficient RDF storage and query answering platform on top of an RDBMS [46], [47], [48] (PostgreSQL in our case). OntoSQL stores RDF triples in an integer-encoded triple table, indexed by `s`, `p`, and `o`.

3.8.1 *Centralized algorithms compared*

For a broader view, we now position our work with respect to some best-known quotient summarization proposals. Concerning the centralized algorithms, in RDFQuotient, we implemented:

- Our *global* and *incremental* algorithms for building $G_{/W}$, $G_{/S}$, $G_{/TW}$, $G_{/TS}$ summaries (a total of eight).
- An algorithm which computes the `fb` (forward and backward bisimilarity or "full bisimilarity") summary of an RDF graph, used in many prior works, e.g., [26]. The `b` summary introduced in [32] is obtained from `fb` by collapsing all leaves together.
- Algorithms to build the k-bounded bisimilarity summaries for $k = 1$, denoted `1fw`, `1bw`, and `1fb` [27]. We do not use higher k values because, as we will show, even at this smallest k, bisimilarity summaries are too large for visualization; higher k would only increase the number of nodes and edges. The `a` summary in [32] and also the characteristic sets of [42]) correspond to the `1fw` summary with all leaves collapsed. Similarly, the `ioa` summary of [32] is obtained from `1fb` by collapsing all leaves.

Real datasets	$\|G\|$	$\|G^\infty\|$	u%	#p	#c
DBLP	88,153,334	88,153,334	87	26	14
DBpedia Person	7,889,268	7,889,268	63	9	1
Foodista	1,019,799	1,019,799	75	13	5
Nobel Prizes	87,549	119,457	71	45	26
Springer LOD	145,136	213,017	77	26	11

Synthetic datasets	$\|G\|$	$\|G^\infty\|$	u%	#p	#c
BSBM [49] 1M	1,000,708	1,009,138	68	38	159
BSBM 10M	10,538,484	10,628,484	66	38	593
BSBM 100M	104,115,556	105,315,556	61	38	2019
BSBM 138M	138,742,476	140,342,476	61	38	2,421
LUBM [50] 1M	1,001,658	1,227,984	34	18	45
LUBM 10M	10,728,460	13,147,069	34	18	45
LUBM 100M	106,778,632	130,843,944	34	18	45

Table 4: Datasets used in experiments.

- The best algorithms from [32] building the `t` and `ioat` summaries, but without the leaf collapse.

As explained in Chapter 2, the leaf collapse introduces a loss of information, and for this reason, we do not adopt it.

3.8.2 *Datasets*

We have experimented with real and synthetic graphs of up to 36.5GB. Table 4 shows for each graph its number of triples $|G|$, the number of triples in its saturation $|G^\infty|$, the percentage of untyped nodes in the graph u%, and the number of distinct data properties #p and classes #c. Each graph has at least 30% untyped nodes; these form a strong majority in all but the LUBM graphs. Note that in BSBM graphs, the number of classes grows with the data size.

3.8.3 *Summary size*

Table 5 reports the node and edge counts, denoted $(n|e)$, for the compared summaries, both directly, and after applying inlining (Section 3.6, denoted with the ^{in} superscript). We report only the data edges in the summaries, and omit schema triples, or metadata triples, e.g., C *dc:publisher* a_1 states that a_1 published class C, etc. These

G	$G_{/W}$	$G^{in}_{/W}$	$G_{/S}$	$G^{in}_{/S}$	$G_{/TS}$	$G^{in}_{/TS}$	$G_{/1fw}$	$G^{in}_{/1fw}$	$G_{/t}$	$G^{in}_{/t}$	$G_{/ioat}$	$G^{in}_{/ioat}$	$G_{/fb}$	$G^{in}_{/fb}$
BSBM1	**179**|**38**	**1**|**7**	184|57	5|9	281|1951	7|10	208|1017	48|573	264|2046	102|485	312|4477	61|617	279999|999436	-|-
BSBM10	**613**|**38**	**1**|**7**	618|68	5|11	1131|10274	7|10	642|1023	48|579	1114|10785	518|749	719|10320	86|740	-|-	-|-
BSBM100	**2039**|**38**	**1**|**7**	2044|70	5|11	3325|25639	7|10	2068|1023	48|579	3308|26918	-|-	2149|34503	105|895	-|-	-|-
BSBM138	**2441**|**38**	**1**|**7**	2446|71	5|11	3983|30759	-|-	2470|1023	-|-	3966|32294	-|-	2556|41963	110|892	-|-	-|-
DBLP	**23**|**26**	**2**|**8**	30|66	7|17	38|196	14|66	267|4344	62|938	55|237	13|188	439|8390	-|-	-|-	-|-
DB. Person	**4**|**9**	**1**|**2**	6|21	2|4	5|11	**1**|**2**	67|377	65|83	3|11	**1**|**2**	135|1651	112|91	-|-	-|-
Foodista	**11**|**13**	**3**|**6**	12|16	**3**|**6**	17|33	6|13	72|1959	25|975	12|36	6|33	191|4518	169|934	-|-	-|-
LUBM1	**82**|**18**	**3**|**12**	86|43	7|27	95|94	16|58	89|81	14|75	92|94	16|58	103|173	24|111	-|-	-|-
LUBM10	**82**|**18**	**3**|**12**	86|44	7|28	95|94	16|58	89|83	14|77	92|94	16|58	101|163	22|100	-|-	-|-
LUBM100	**82**|**18**	**3**|**12**	86|44	7|27	95|94	16|58	89|83	14|77	92|94	16|58	101|167	22|104	-|-	-|-
Nobel	**97**|**59**	**9**|**18**	110|110	21|36	105|85	16|33	117|555	48|434	83|82	14|65	164|697	70|316	22640|83206	10765|32671
Springer	**49**|**24**	**4**|**2**	49|24	**4**|**2**	49|24	**4**|**2**	40|51	8|9	37|24	**4**|**2**	73|90	8|9	65297|139566	15530|30028

Table 5: Summary sizes: direct and after inlining.

omitted triples are the same for all summaries[9]. Numbers are missing (-|-) when algorithms ran out of memory or longer than 3 hours.

`fb` summarization failed to complete within 3 hours, on all but the smallest graphs (Springer, Nobel Prizes and BSBM 1M). This is partially due to our simple, single-computer implementation, but computing `fb` is also intrinsically hard, as it requires many iterations. More efficient methods to build the `fb` summary are parallel [51]; existing algorithms to build the `t` and `ioat` summaries are based on MapReduce [32]. Here, we study the size, precision, as well as qualitative properties (see Section 3.2) of prior-work summaries, based on simple (if not the most efficient) centralized implementations. The `fb` results we obtained confirm the observation in Section 3.2 that such summaries are much too complex to be used for a first visualization.

Among the other summaries, *before inlining*, as theoretically expected (Proposition 1), $G_{/W}$ always has the fewest edges; $G_{/S}$ is close. In contrast, summaries that group nodes (also) by the types, such as $G_{/TS}$, $G_{/t}$ and $G_{/ioat}$ are much larger. This is particularly visible for larger BSBM datasets: as the schema complexity grows, these summaries have tens of thousands of edges. $G_{/ioat}$ remains too complex even for the simpler-schema graphs (excluding BSBM), with hundreds or thousands of edges, whereas $G_{/W}$ and $G_{/S}$ only have a few dozen edges. $G_{/1fw}$ is always smaller than $G_{/ioat}$ (this can be shown theoretically based on their definitions), but it remains much larger (by a factor of 2 in the case of Springer, up to 122 in the case of Foodista) than $G_{/S}$. Among the graphs other than BSBM, $G_{/t}$ has less edges than $G_{/S}$ on the DBpedia Person graph, and less nodes on the Springer graph; on all the others, $G_{/t}$ is several times larger than $G_{/S}$. Further, $G_{/t}$ has the qualitative drawback of considering all untyped nodes equivalent; none of our summaries has this problem.

After inlining, $G^{in}_{/W}$ and $G^{in}_{/S}$ are reduced to very few (1 to 21) nodes and 2 to 36 edges. In contrast, $G^{in}_{/t}$ has significantly more edges than $G^{in}_{/S}$ (up to 11× for BSBM10 or DBLP); it is smaller than $G^{in}_{/S}$ (by a small margin) only on DBPedia Person and Nobel. The summaries $G^{in}_{/1fw}$, $G^{in}_{/ioat}$ still remain very large, thus, not useful for first-sight visualization.

3.8.4 *Summarization time*

The times to build our summaries using the global and the incremental algorithms are plotted as a function of $|G|$ in Figure 24; both axes are in log scale. For each summary type, the summarization time is roughly linear in the size of G, confirming the expectations stated in Section 3.7. **Increm-**`W` is the fastest overall; it traverses G only once, thus it is faster than **global-**`W` which performs several passes. `S`, `TW`

9 In details, for the graphs in Table 5, we omitted: 1,008; 4,028; 13,352; 16,020; 19; 1; 5; 246; 246; 246; 171; respectively; 108 triples.

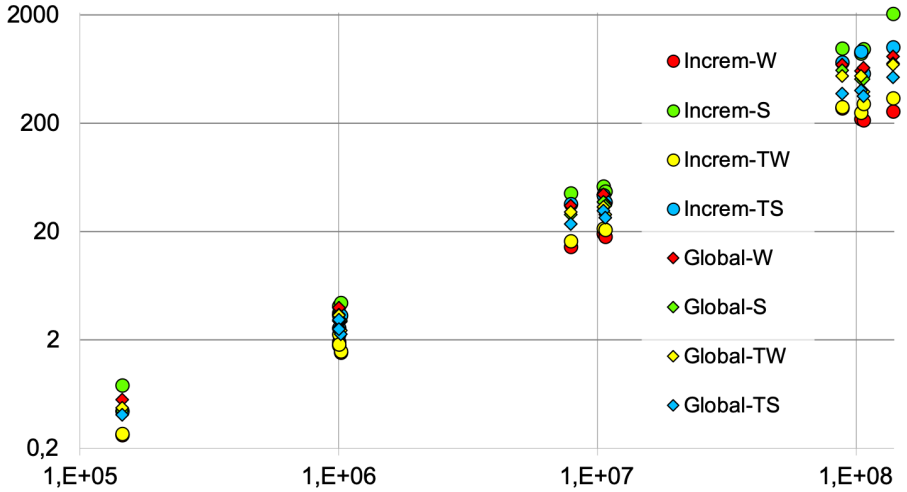


Figure 24: Summarization time (s) vs. graph size $|\mathtt{G}|$.

and TS, in this order, are more expensive, and finally incremental S which, as we explained, is quite complex. Since **increm-S** is rather expensive per-triple, it is more efficient to first summarize a graph using **global-S**, and call **increm-S** only to maintain it later. This is significantly faster: for instance, **global-S** on BSBM138M takes only 11.85 minutes, whereas **increm-S** takes 34.5. **Increm-TS** is often faster than **increm-S** because typed nodes do not lead to splits during TS summarization.

SHORTCUT SPEEDUP Table 6 shows the time to build $(\mathtt{G}^{\infty})_{/\equiv}$ in two ways:

- *Direct*, i.e., saturate G then summarize, denoted $dt_{\equiv}$, and
- *Shortcut* (Section 3.5.3), summarize G, then saturate the summary and summarize again, denoted $st_{\equiv}$.

It also shows the **shortcut speedup** $x_{\equiv}$ for $\equiv \in \{\mathtt{W}, \mathtt{S}\}$ defined as $(dt_{\equiv} - st_{\equiv})/dt_{\equiv}$. The speedup ranges between 39% and 94%in all cases, a direct consequence of $\equiv_{\mathtt{W}}$ and $\equiv_{\mathtt{S}}$ compression. Indeed, $dt_{\equiv}$ includes the time to summarize $\mathtt{G}^{\infty}$, whereas $st_{\equiv}$ includes the time to summarize $(\mathtt{G}_{/\equiv})^{\infty}$; the smaller this is, the higher $x_{\equiv}$.

3.8.5 *Summary precision*

We now attempt to quantify the loss of precision of our structural summaries. A simple measure is the fraction of summary subgraphs having no isomorphic counterpart in the data; intuitively, summary users may believe G exhibits such structures, but this is not true. For instance, node N_1 in Figure 10 has, among others, two outgoing edges labeled a and ta, whereas such a node does not exist in the original graph (Figure 3). We define the **precision loss at** l, or PL_l as the

Dataset	$dt_{\mathtt{W}}$ (s)	$st_{\mathtt{W}}$ (s)	$x_{\mathtt{W}}$ (%)	$dt_{\mathtt{S}}$ (s)	$st_{\mathtt{S}}$ (s)	$x_{\mathtt{S}}$ (%)
BSBM1M	6.56	2.12	0.68	9.52	4.37	0.54
BSBM10M	73.96	19.34	0.74	123.73	52.84	0.67
BSBM100M	797.50	218.83	0.73	1451.40	884.35	0.39
BSBM138M	1393.69	257.19	0.82	3627.19	2049.22	0.44
LUBM1M	24.53	1.99	0.92	27.47	3.76	0.86
LUBM10M	302.24	18.13	**0.94**	354.81	47.09	**0.87**
LUBM100M	3472.07	214.28	**0.94**	4247.83	974.20	0.77
Nobel	3.32	0.46	0.86	3.55	0.70	0.80
Springer	4.07	0.42	0.90	4.52	0.96	0.79

Table 6: Shortcut experiments.

fraction of connected l-edges subgraphs of the summary $\mathtt{G}_{/\equiv}$ without a counterpart in $\mathtt{G}$. Table 7 shows PL_2 scores for our datasets. By definition, `fb`, `1fb` and `ioat` have $PL_2 = 0$ for any graph, since they reflect completely node neighborhoods at distance 1. In contrast, $\mathtt{G}_{/\mathtt{W}}$ has the highest precision loss; $\mathtt{G}_{/\mathtt{TS}}$ is the most precise, much better than $\mathtt{G}_{/\mathtt{t}}$ which blindly collapses untyped nodes. `1fw` is quite imprecise in some cases, since it ignores incoming edges.

3.8.6 *Experimental conclusions*

Our four summaries can be built efficiently in linear time. They strongly reduce graph sizes and, through inlining, they lead to compact, understandable graphs which fit human comprehension capacity at first sight. If all non-leaf $\mathtt{G}$ nodes are typed, $\mathtt{G}_{/\mathtt{TS}}$ (or, equivalently, $\mathtt{G}_{/\mathtt{TW}}$), with type generalization, are the most informative and most precise, given that type information specified by humans carries precious information about the graphs. Otherwise, $\mathtt{G}_{/\mathtt{S}}$ strikes the best balance between concision and informative content, whereas $\mathtt{G}_{/\mathtt{W}}$ loses more precision. The shortcut speeds up `W` and `S` summarization of $\mathtt{G}^{\infty}$ by 39% to 94%.Thus, *we find the strong summary the safest and most interesting choice: it can be built efficiently and is most likely to lead to informative yet understandable RDF graph summaries*. More generally, the data publisher can build them all (first build $\mathtt{G}_{/\mathtt{S}}$ and $\mathtt{G}_{/\mathtt{TS}}$, then $\mathtt{G}_{/\mathtt{W}}$ from $\mathtt{G}_{/\mathtt{S}}$ and $\mathtt{G}_{/\mathtt{TW}}$ from $\mathtt{G}_{/\mathtt{TS}}$) and select the one(s) to share with potential data users.

3.9 PARALLEL SUMMARIZATION EXPERIMENTS

We implemented the parallel algorithms designed for the Spark framework in Scala 2.11; recall the implementation details discussed in Section 3.7.7.

Dataset	$G_{/W}$	$G_{/S}$	$G_{/TS}$	$G_{/1fw}$	$G_{/t}$
BSBM1	0.56	0.26	**0.01**	0.46	0.46
BSBM10	0.56	0.45	**0.01**	0.46	0.46
BSBM100	0.56	0.48	**0.01**	0.46	0.46
BSBM138	0.56	0.49	**0.01**	0.46	0.46
DBLP	0.23	0.09	**0.03**	0.32	0.37
DB. Person	0.09	**0.08**	0.09	0.28	0.31
Foodista	0.15	0.15	**0.00**	0.25	0.32
LUBM1	0.50	0.25	**0.00**	0.15	0.03
LUBM10	0.50	0.25	**0.00**	0.15	0.03
LUBM100	0.50	0.25	**0.00**	0.15	0.03
Nobel	0.27	0.04	**0.01**	0.59	0.49
Springer	**0.01**	**0.01**	**0.01**	0.59	0.59

Table 7: Precision loss experiments.

COMPUTATIONAL ENVIRONMENT Our experiments have been carried on the computational infrastructure described in Section 3.8. For our parallel experiments we used a cluster of 6 such machines. All machines in this cluster are connected to a switch using 10 Gigabit Ethernet. We used Spark 2.3.0 with Hadoop's YARN scheduler 2.9.0. We give to Spark and YARN 100GB of RAM and 36 cores at each machine. We leave a fraction of the memory (remaining 24GB) and 4 CPU cores for the operating system.

EXPERIMENTS OVERVIEW We now study our parallel algorithms for computing $G_{/W}$, $G_{/S}$, $G_{/TW}$, and $G_{/TS}$ summaries. We used the BSBM benchmark graphs of 1M, 10M, respectively, 100M triples. In these graphs, *61% to 68%* of the nodes were *untyped*, whereas the others have at least one type. On the one hand, this justifies the need for summaries (such as $G_{/W}$ and $G_{/S}$) which do not require all nodes to be typed; on the other hand, there is also a sizable share of typed nodes, which makes the computation of $G_{/TW}$ and $G_{/TW}$ significantly different, on these graphs, than that of $G_{/W}$ and $G_{/S}$. As we have seen in Table 5, the RDFQuotient summaries of the BSBM graphs are quite small; thus, the information (PRIs and UDs) broadcast by our algorithms is also quite compact.

3.9.1 *Configuration*

We recall here that M denotes the number of machines (Spark workers). Spark parameters relevant for our performance analysis were set as follows:

- The number of cores per machine C_M: we picked $C_M = 36$ (all available cluster resources). We pick all the available resources in order to maximize the performance.
- The number of cores per executor C_E. Following Spark guidelines, unless otherwise specified, we pick $C_E = 4$.
- The available memory per machine R_M: we set it to 100GB.
- The amount of memory per executor $R_E \geq 2.78$GB. There can be at most C_M executors per machine, so they will use at least $\frac{R_M}{C_M}$, in general $R_E = \frac{R_M \cdot C_E}{C_M}$. We set the lower bound for R_E as a minimal memory of the YARN container, within the memory limit for executor we need to hold out around 1GB for a memory overhead (memory for JVM in YARN).
- The number of executors per machine E_M, typically 9.
- The total number of executors E is computed as $E_M \cdot M$.
- The number of partitions $P = \alpha E$. Following existing recommendations[10], we set $\alpha = 4$.

3.9.2 *Speedup through parallelism*

We fix the following parameter values $M = 5$, $C_E = 4$, $E_M = 9$, $R_E = 11$GB, $P = 180$ and we vary E, the number of executors, by using more or less machines.

Figure 25 shows the run time (in minutes) of the parallel algorithms with respect to the number of executors, on a BSBM graph of 10M triples. The time here includes loading, preprocessing, summarization and saving the output file containing the graph summary. We can see that parallelization helps decrease the run time: there is a big gain from 9 to 18 executors, and smaller gains as the parallelism increases (in our setting, benefits basically disappear/amortize going from 36 to 45 executors).

Figure 26 zooms to show *only the summarization time* (in seconds) for the same set of computations. We see that the algorithm is overall efficient, i.e., summarization itself is quite fast (seconds as opposed to minutes for the overall computation). We investigated and found large parts of the run time in Figure 26 are spent:

1. Encoding the RDF triples into triples of integers; this is by far the dominant-cost operation. To do this, we need to identify the (duplicate-free) set of node labels from `G`, operation which we

10 http://spark.apache.org/docs/latest/tuning.html#level-of-parallelism, https://stackoverflow.com/questions/31359219/relationship-between-rdd-partitions-and-nodes

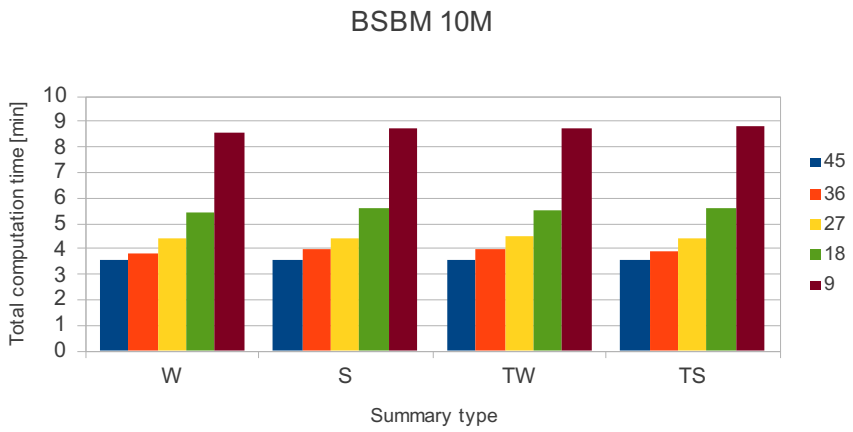


Figure 25: Total run time in the parallel setting for various number of executors.

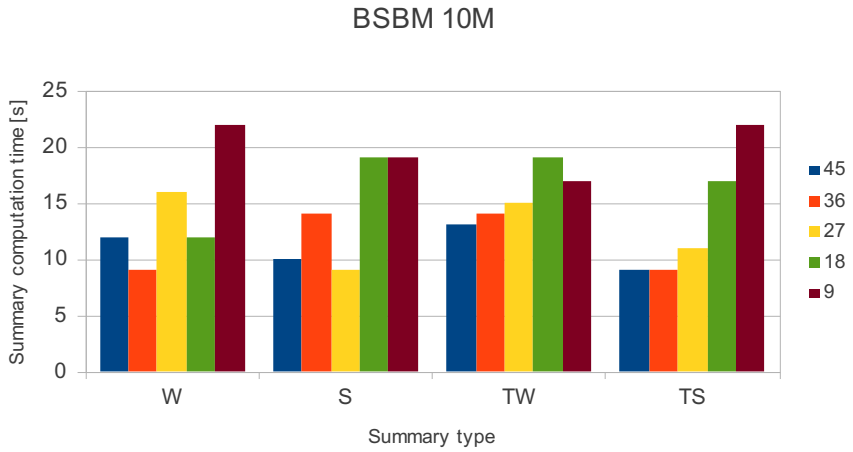


Figure 26: Summarization time in the parallel setting for various number of executors.

implemented using Spark's `distinct()` function, which eliminates duplicates from an RDD that is (as usual) spread across the nodes. It involves communication between nodes, thus its high latency. We note, however, that our initial implementation, which lacked the encoding and worked directly with URIs and literals from `G`, was way slower (and encountered memory issues). Thus, we believe the encoding cost is worth paying;

2. The precomputation of the RDF class and property nodes from a given graph `G`. This is also implemented by each machine adding its schema nodes to an RDD and then calling `distinct()`.

Coming back to Figure 26, we see that despite the need for a global synchronization step, parallelism (increasing the number of executors) clearly shortens the summarization time. A few data points are counter-intuitive, e.g., using 45 executors takes slightly more than using 36 for weak summarization, or using 36 takes more than using 27 for strong summarization. We believe these points are due to the (random) way in which the data is distributed among the executors, which in turn determines when the broadcast operation (Steps (S4) and (W5)) is finished. Even though this distribution is simply made in round-robin fashion, we find that overall parallelism clearly helps, which can be seen, e.g., by comparing the times for the lowest parallelism degrees (9, 18) and the highest (36, 45); this holds even more when put back in the perspective of the overall time (Figure 25).

3.9.3 *Scalability study*

Our next experiment studies the impact of the RDF graph size on the summarization time. We have repeated the above experiments for BSBM graphs of 1M and 100M triples, leading to time measures for 1M, 10M, and 100M triples, that we analyze together to determine how the algorithms scale. Figure 27 shows the total run time (in minutes), whereas Figure 28 depicts only the time to compute the summary. Note the logarithmic scale of the y axis in both graphs.

The total time (Figure 27), which is dominated by the data preprocessing and, thus, I/O-bound, grows linearly with the data size.

The summarization time alone (Figure 28) grows almost linearly with the data size for weak, typed weak and typed strong summarization, whereas the growth of the summarization time is super-linear for the strong summary. We have not been able to determine precisely the cause. Recalling also Figure 26, we believe there is some variability in Spark in-memory execution performance, that we were not able to control precisely. However, considering (also) the fact that summarization itself takes a relatively small part of the total time, we can conclude that overall our parallel algorithms scale up well (basically linearly) with the data size.

3.9.4 *Experimental conclusions*

We studied the performance of our novel parallel algorithms for building the weak, strong, typed weak, and typed strong summaries. First, we analyzed the total execution time. We observe that computation through Spark can come with a significant overhead with respect to centralized implementation. This is in part due to network and synchronization costs inherent to the MapReduce-like systems. Since Spark is not designed as a graph-oriented store, we must perform the expensive data preprocessing such as integer encoding. Never-

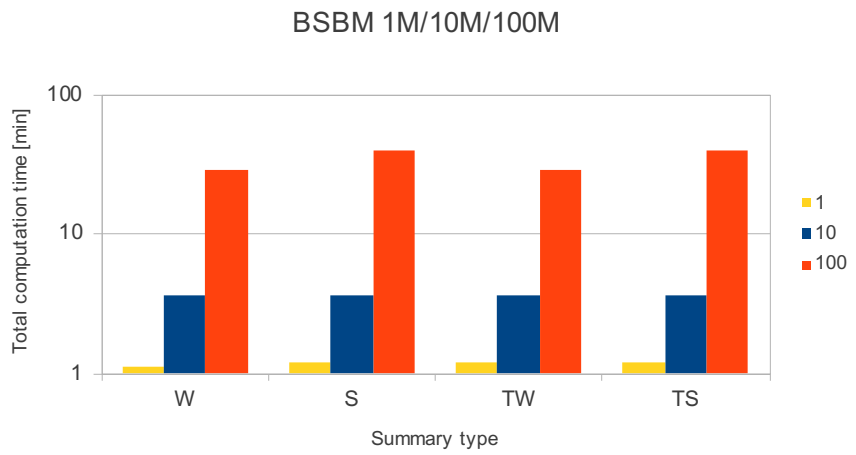


Figure 27: Total run time in the parallel setting for datasets of different size.

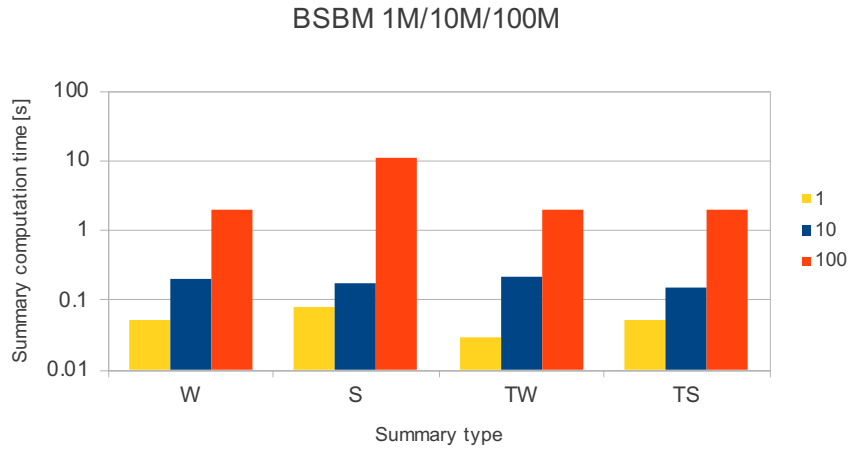


Figure 28: Summarization time in the parallel setting for datasets of different size.

theless, we showed that our summarization algorithms benefit from parallelism. In particular, we summarization time alone is small and lets us both scale up with the dataset size and scale out the run time linearly as the number of executors grows. In our experiments, we observed the best performance for the weak summary with high number of executor nodes (36 or 45). We recommend to use the centralized algorithms whenever the memory constraints of a centralized settings permit that. However, in the settings where the graph size exceeds the memory available for a single machine, our parallel algorithms are the only means to carry out large summarization.

3.10 NON-QUOTIENT RDF GRAPH SUMMARIZATION

As discussed in the introduction and in Section 3.2, summarization is a well-studied notion for general graphs [23] and RDF ones in particular [14]. Many non-quotient RDF graph summaries exist, see, e.g., [14]. Our work pertains to the family of quotient summaries of RDF graphs, and most directly compares to bisimilarity-based quotients, as well as the more RDF-specific ones.

[39] builds answer-preserving summaries for reachability and graph pattern queries. However, these summaries do not preserve query structure (i.e., joins), which quotient summaries do preserve.

Other graph summaries compress graphs with bounded "error" (number of edges to be added as "corrections" after decompression, to retrieve the original graph) [52], [53]. Nodes and edges are summarized according to their frequencies and/or based on ontology patterns in [54], [55]. Summaries where nodes are grouped by graph clustering [56], user-defined aggregation rules [57], mining [58], and identification of frequent subtrees [59] do not reflect the complete structure, and/or require user input. With different objectives, these summaries may omit part of the graph structure, or be much too large for visualization.

Work on fitting XML data into relational stores [60], [61] also aimed at finding "homogeneous" node groups; "inlining" there meant storing in the relation of node n, its properties which occur at most once. Our inlining (Section 3.6) pushes leaf nodes within their parents, regardless of their number of occurrences.

Quality metrics for RDF graph summaries have been proposed in [62]. Along these metrics, our summaries score high for preserving all classes and properties from $\mathtt{G}$; the price they pay for compactness is to sometimes show instances that do not exist in $\mathtt{G}$ (as our precision loss experiments show).

3.11 CONCLUSION

This chapter has described my thesis research on quotient summarization of RDF graphs, that is, graph summaries derived from a notion of equivalence among RDF graph nodes. We made the following contributions:

1. We presented four novel RDF graph summaries: weak, strong, typed weak, and typed strong, which are often small and easy-to-comprehend, in the style of ER diagrams;

2. We showed RDFQuotient, a framework built upon our novel efficient and scalable (amortized linear-time) algorithms for computing our summaries either from scratch, or incrementally, reflecting additions to the graph;

3. We provided new parallel algorithms that surpass limitations of memory-constrained environments, and we implemented them in the Spark framework; they enable further scaleout of our RDFQuotient summarization methods;

4. We experimented with datasets of size different by orders of magnitude; we assessed the performance and showed benefits of the global, incremental, and parallel algorithms (while varying the degree of parallelism);

5. We presented the first formal study of the interplay between RDF graph saturation in the presence of an RDFS ontology, and summarization. We formulated a sufficient condition for a highly efficient shortcut method to build the quotient summary of a graph without saturating it;

6. We stated formal results establishing the shortcut conditions for some of our summaries and others from the literature;

7. We demonstrated experimental validations of our claims based on our RDFQuotient tool available online.

Moreover, we showed how our new summaries compare with prior work. Even though more complex summaries, such as `1fb` and `ioat`, are better suited for indexing, they do not serve a purpose of a first interface to show to users. In contrast, our summaries and, among them, the strong summary, which we generally find the best, has the advantages of being:

- Compact and easy to understand for domain-specific graphs,
- Efficiently computed in linear-time, and
- Benefiting from the original shortcut procedure we introduced.

In addition, none of our summaries has the drawback of collapsing all leaves, all roots, or all untyped nodes.

RDFQuotient has been used or built upon in several research works. Among the ones we know in our team:

1. It has been used to prune empty-answers RDF queries in [63].

2. It is currently being used in the PhD thesis of Nelly Barret to help abstract heterogeneous data graphs [64], created out of different types of data.

3. My own thesis work has built upon it to help identify interesting aggregates in an RDF graph, as we describe in the next chapter.

4

DISCOVERING INTERESTING AGGREGATES IN RDF GRAPHS

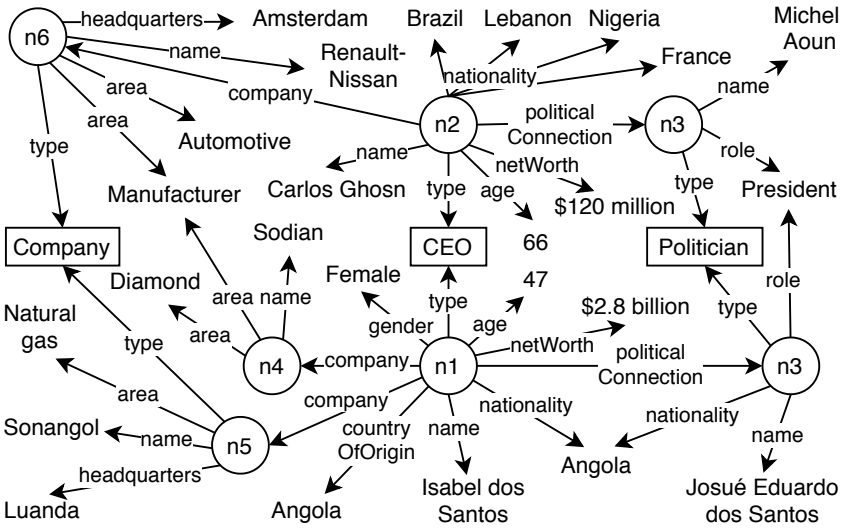
In this chapter, we study the problem of discovering interesting aggregates in RDF graphs, automatically. We start by formulating our research problem. We then outline the existing aggregation techniques based on the CUBE operator known from relational data warehouses. We show and explain reasons why the existing algorithms are not suitable for RDF graphs as they incur errors in the results once run on RDF graphs. Next, we introduce our novel RDF graph aggregation techniques that are correct and efficient. We describe in detail a new end-to-end system that finds the interesting aggregates in RDF graphs. We demonstrate how we can further speedup the aggregate evaluation using new early-stop techniques. Finally, we validate the performance of our algorithms and our system as a whole with experiments.

4.1 MOTIVATION

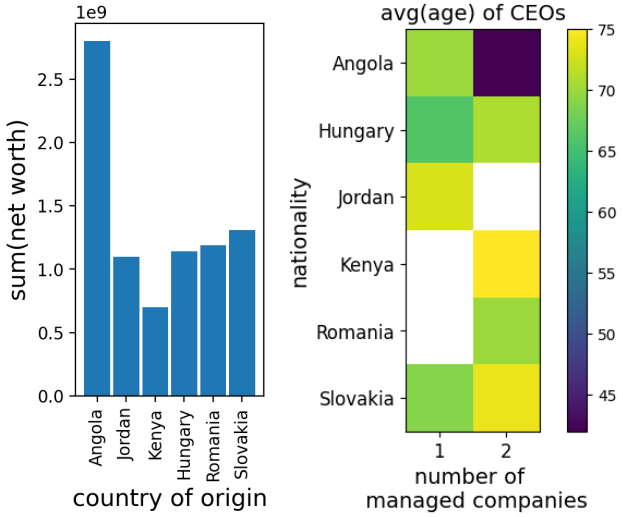
RDF graphs are increasingly being published and shared as part of the Linked Open Data (LOD) movement. Given the size, heterogeneity, and complexity of these graphs, their information content is hard to grasp, in particular for non-expert users. In this chapter, we explore automatic insight extraction from RDF graphs [6], [7], [65]. Given a graph and an integer k, we seek to automatically identify the k most *interesting insights* in the graph. An insight is an *RDF analytical (aggregate) query* that results in aggregated *measures* over the data, grouped by a set of *dimensions* [10], [11]. The query can be expressed in a language such as SPARQL 1.1, the W3C's standard RDF query language [9], and evaluated by any RDF query engine. The *interestingness* of an insight is assessed based on a statistical measure of the query result.

Recall from Chapter 1 our motivating application, the Computational Lead Finding. Having addressed graph data summarization needs in Chapter 3, we now provide a solution for the targeted journalist-oriented lead-finding scenario. Below, we outline our approach to RDF insight extraction using examples from statistical lead discovery.

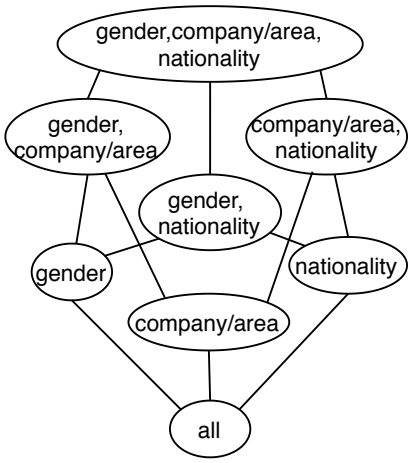
Running examples. Consider an RDF graph comprising politicians, CEOs, and connections between them. We can extract such a graph, for



(a) RDF data.



(b) Examples of MDAs.



(c) Lattice for Example 3.

Figure 29: Running examples using the CEOs dataset.

instance, from the WikiData open-source RDF repository. Figure 29(a) shows an example RDF graph where CEOs are linked with politicians, e.g., Isabel dos Santos, a wealthy Angolan CEO (at the heart of the Luanda Leaks scandal), is the daughter of a former president of Angola. Starting from the graph, we aim to *automatically identify* a small set of aggregate queries that are *statistically interesting*. Here, interestingness is a statistical measure that indicates deviation from the prior knowledge of the journalists. For example, the interesting aggregate results may deviate from a uniform distribution of values over different aggregate groups, or a normal distribution over numeric dimensions such as *age*.

Table 8 shows three example aggregates, whose dimensions and measures are either properties in the RDF graph or properties that we derive to enrich the scope of the analysis. In Example 1, *CEO*s, *politicalConnection*s, *countryOfOrigin*, and *netWorth* are either types or properties in the RDF graph in Figure 29(a). Example 2 analyzes the CEOs along *the number of managed companies*, which is not a property in the graph: we *derive* it by *counting* the properties of each CEO. This enables us to discover, e.g., that the average age of Angolan CEOs that manage two companies is low compared to other nationalities. Example 3 analyzes CEOs by *areas of companies*; we derive this from the graph by following a path from the CEOs to the companies they manage, then to their areas. Similar path examples include *company/headquarters*, *politicalConnection/role*; longer paths produce a larger number of novel angles for the analysis.

Among all possible aggregate queries that we can generate, the above three examples are selected because their results show significant deviation from uniform values (having outliers). For Examples 1 and 2, Figure 29(b) shows, respectively, a histogram that exhibits an outlier in *sum*(*netWorth*) for Angola, and a heat map where the dark color reflects a low value of *avg*(*age*) of CEOs, both due to Dos Santos. We can show to the user such interesting insights as:

- Histograms (if one-dimensional),
- Heat maps (if two-dimensional), or
- Tables (for high-dimensional aggregates).

Our goal to discover the k most interesting aggregates from an RDF graph poses two unique challenges:

Challenge C1 - Aggregate identification. Automatic extraction of interesting aggregates is one among many existing techniques for data exploration and visualization recommendation. Yet, most prior works assume a fixed relational schema [66], [67]. In contrast, in RDF graphs, facts, dimensions, and measures are not specified but must be identified therein. To address this challenge, given an RDF graph, we provide a variety of strategies to create new dimensions and measures,

Example 1	Sum of the net worth of CEOs with political connections grouped by country of origin.
Example 2	Average age of CEOs grouped by nationality and number of managed companies.
Example 3	Number of CEOs grouped by nationality, gender, and area of the companies they manage.

Table 8: Examples of interesting aggregates.

which enable us to examine a rich space of candidate aggregates and to discover the most interesting ones. We further develop a modular framework for aggregate identification, which can be extended or customized as needs arise.

Challenge C2 - Efficient and correct aggregate evaluation. Since we define interestingness on an aggregate *result*, we must evaluate candidate aggregates to determine if they are among the k most interesting ones. A key feature of our work is that we look for *multidimensional aggregates* (MDAs), such as Examples 2 and 3. A set of N dimensions, among which we enumerate candidate aggregates, leads to a lattice [68] of 2^N nodes, each of which is an MDA (see Figure 29(c)). We may have many such lattices to consider at once, and efficiently evaluating them all poses a salient challenge.

To address the challenge, first, we revisit a classical framework for *lattice-based MDA computation* in *relational data warehouses* (DWs). Efficient algorithms, such as ArrayCube [22], compute an aggregate in the lattice from the result of one of its parents, and compute all aggregates in the lattice in a single pass over the data. However, a crucial observation we make in this work is that the classical one-pass approach to lattice computation is *incorrect* for RDF data, due to a phenomenon called *multi-valued dimensions*, that is, *an RDF node (fact) may have multiple values along a given dimension*. To tackle the issue while still retaining the benefits of one-pass algorithms, we provide a theoretical analysis of how the classical approach produces errors. Furthermore, we develop a new RDF-compatible one-pass algorithm that:

1. Correctly and efficiently handles lattice-based MDA computation where the aggregates use multi-valued dimensions;

2. For each node in the lattice (with a given set of dimensions), simultaneously handles many aggregates that differ in the measure (among many possible ones) and the aggregate function in use;

3. Saves computation cost by sharing measures across all lattices that analyze the same set of facts.

Second, to further improve efficiency, we develop a new technique to stop the evaluation of an MDA as soon as we can determine (with high probability) that it will not be among the top k. Our technique builds on the work in [69], which provides *confidence-interval* (CI) bounds on an *approximate* aggregate result. Our problem is harder because we want to approximate *the interestingness score computed over the aggregate result*, which amounts to estimating the result of a *nested* aggregate query, whereas the prior work does not support such nested queries. Using advanced statistical tools, we construct CIs for the interestingness function including *variance*, *skewness*, and *kurtosis* over estimated results of candidate aggregates, enabling early pruning of uninteresting aggregates.

In summary, the contributions we make in this work include:

1. We formalize our problem of finding the top-k most interesting aggregates in an RDF graph (Section 4.2);
2. Spade, a new *RDF-oriented end-to-end framework* that automatically identifies, enumerates, and efficiently evaluates RDF MDAs to determine the most interesting ones (Section 4.3);
3. MVDCube, the first correct and efficient algorithm for *one-pass lattice-based computation of RDF MDAs* (Section 4.4);
4. A novel *early-stop technique* that stops the evaluation of MDAs that, with a high probability, will not be in the top-k list (Section 4.5);
5. We further detail the system architecture and provide an insight into the Spade implementation choices (Section 4.6);
6. *Experimental results* (Section 4.7) validating:
   - The ability of Spade to extract insights from a large space of candidate aggregates;
   - The frequent, and potentially high errors that existing algorithms introduce on real-life, heterogeneous RDF graphs;
   - The efficiency of our one-pass algorithm, which is faster than PostgreSQL's GROUP BY CUBE implementation by 20% to 80%;
   - The extra speedup of 10% to 43% achieved by our early-stop technique, and
   - The scalability of Spade as the size and complexity of the graphs increase.

4.2 PROBLEM STATEMENT AND NOTATION

We consider well-formed RDF graphs that we saturate prior to our analysis (recall Chapter 2).

A **candidate fact set** (CFS) is a set of RDF nodes that we build an interesting aggregate on; we call a member of the set a **candidate fact** (CF).

An **attribute** is either a (direct) **property** (P) of a CF in the original RDF data, or a **derived property** (DP), which we create from the data and attach to a CF to enrich the analysis. For instance, one may attach to each CEO the number of companies they manage (the full set of derivation strategies is discussed in Section 4.3). An attribute can be used as a **dimension**, to group CFs by value, or as a **measure**, to be aggregated within each group of CFs.

We employ an **aggregate function**, f, that ranges over the common set $\Omega = \{count, min, max, sum, avg\}$.

A **multidimensional aggregate** (MDA), $A = \langle CFS, \mathcal{D}, M, f \rangle$, is determined by: a CFS, a set $\mathcal{D} = \{D_1, D_2, \ldots, D_N\}$ of dimensions (which are attributes), a measure M (also an attribute), and an aggregate function f. The semantics of A is that of a SPARQL 1.1 aggregate query [9] (recall Section 2.2 in preliminaries), which also agrees with that of the RDF analytical queries introduced in [11], [10]. The result of A on an RDF graph $\mathtt{G}$, denoted $A(\mathtt{G})$, is the set of tuples, one per each distinct combination of dimension values (aggregate group) in the data:

$$A(\mathtt{G}) = \{(d_1, d_2, \ldots, d_N, f\{m_j \mid \exists\, CF_i \in CFS, CF_i.D_1 = d_1, \\ CF_i.D_2 = d_2, \ldots, CF_i.D_N = d_N, CF_i.M = m_j\})\}$$

where CF_i has (at least) the values $d_1, d_2, \ldots, d_N$ along the dimensions $D_1, D_2, \ldots, D_N$, and m_j iterates over the set of values of the measure M on CF_i. Finally, $f\{\cdot\}$ is the result of running the aggregate function f over the measure values from a given set.

Our semantics, unlike that of relational DWs, does account for *heterogeneity in RDF data*:

1. Some CFs may miss dimensions and/or measures, and thus they do not contribute to the result. For the graph in Figure 29, the result for Example 1 is {(Angola, \$2.8B)}, due to n_1, whereas n_2 does not contribute to the result as it lacks the *countryOfOrigin* dimension;

2. A CF may contribute to multiple groups in A (if it has multiple values for a dimension), and/or multiple times to the aggregated value (if it has several values for the measure). The result for Example 2 is {(Nigeria, 1, 65), (France, 1, 65), (Lebanon, 1, 65), (Brazil, 1, 65)}, all obtained from n_2 given its four distinct values of *nationality*. Although n_1 has both dimensions, it does not contribute to the result as it misses the *age* measure.

An **interestingness function**, h, is applied over the result of an aggregate A. Let W be the number of tuples in $A(\mathtt{G})$ and, for each

tuple $t_i \in A(\mathtt{G})$, let $t_i.v$ be the aggregated value computed by f. Then h takes the set $\{t_1.v, t_2.v, \ldots, t_W.v\}$ and returns a score, i.e., a positive real number, reflecting a measure of interestingness of A. The user chooses the function to be used during the analysis.

Finally, the problem we address is stated as follows:

Problem 1 *Given an RDF graph* G*, a positive integer k, and an interestingness function h of choice, find the aggregates* $A_1(\mathtt{G}), \ldots, A_k(\mathtt{G})$ *whose interestingness on* G *is the highest.*

4.3 OVERVIEW OF THE APPROACH

In this section, we describe the system design of Spade, a new *RDF-oriented end-to-end framework* that automatically identifies, enumerates, and efficiently evaluates MDAs to determine the most interesting ones. Figure 30 shows Spade's analytics pipeline; it comprises an *offline* phase, where an RDF graph is loaded and preprocessed, and an *online* phase, where user-specific analysis is performed.

OFFLINE PROCESSING Upon loading an RDF graph, we first build a structural summary thereof, using the open-source RDFQuotient tool [44]. The summary captures all the properties occurring in the graph and proposes a set of RDF node groups such that the RDF nodes in each group are considered equivalent. Spade uses the summary to expedite several steps of the analysis, e.g., the enumeration of RDF types and properties, as described below.

Next, we perform **Offline Attribute Analysis** with three main purposes:

1. To gather a set of *statistics* for each property in the graph,
2. To determine if derivations should be generated for a given property, and
3. To decide if preaggregated values of some properties should be computed and stored in the database.

Derived properties are the key to a rich search space and to effectively addressing challenge **C1**. With this aim, we compute statistics including the type of property values (e.g., String, Integer, Date) and, if they are multi-valued, their number of distinct values, the lowest and highest values, etc. Based on these results, **Derived Property Enumeration** generates:

1. *Property counts* for multi-valued properties, e.g., how many companies a CEO manages;
2. *Keywords occurring in property values*, e.g., if a company's description is "Sonangol oversees petroleum production", we attach to

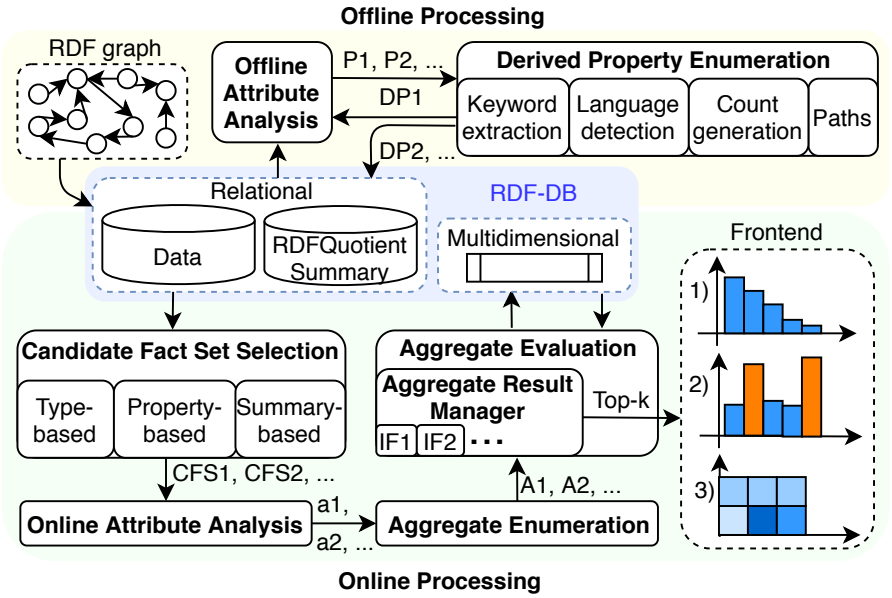


Figure 30: The architecture of Spade.

the company the multi-valued attribute *kwInDescription* with the values "Petroleum" and "Production";

3. The *language* of a text property, e.g., a company may gain the attribute *langOfDescription* with the value "English";

4. *Paths*, e.g., a CEO politically connected to a "President" gains the attribute *politicalConnection/role* with the value "President".

Finally, each derived property is also analyzed and stored along with its statistics in the database.

In addition, for each multi-valued attribute, we create a table in the database storing its values, preaggregated on the RDF nodes that have it. More specifically, for each RDF node, we compute and store the aggregated value for each (attribute, aggregate function) pair, e.g., the sum of a_1, the count of a_1, the minimum of a_2. This allows Spade to account for facts with multiple measure values and improve Aggregate Evaluation during Online Processing.

ONLINE PROCESSING The analysis of RDF graphs suits the specific needs of users and proceeds in the following steps. **Step 1** is **Candidate Fact Set Selection**. To address challenge **C1**, Spade identifies CFSs in three ways:

1. *Type-based*: for each type T in the graph, the set of RDF nodes of type T;

2. *Property-based*: for a (user-specified) set of properties, all the RDF nodes having those outgoing properties;

3. *Summary-based*: each set of RDF nodes identified as equivalent by the RDFQuotient summary; RDF nodes in the same equivalence class tend to have many common properties, making them interesting candidates to be analyzed together.

Step 2 is **Online Attribute Analysis**. In this step, for each CFS, we first enumerate all direct and derived properties. Then, we enrich the offline-analysis results by adding CFS-dependent statistics, e.g., the support of an attribute among all the facts in the CFS, the number of CFs that have such an attribute more than once, and the number of distinct values. Spade exploits the gathered statistics in different steps, e.g., to guide the choice of dimensions, measures, and aggregate functions and to improve Aggregate Evaluation.

Step 3 is **Aggregate Enumeration**. Spade uses the pool of analyzed attributes to generate candidate MDAs. To address challenge **C1**, we generate a rich space of candidate aggregates while applying rule-based pruning to avoid meaningless candidates.

1. *Identifying dimensions and measures from (derived) properties*: We first enumerate all the (derived) properties and consider them for dimensions or measures, subject to the following rules:
   - Dimensions and measures must be *frequent*, i.e., having a support greater than a defined threshold;
   - Dimensions should *not have too many distinct values* when compared to the number of facts to examine (e.g., we do not consider counting the number of CEOs by their birthday as there are too many distinct values for the birthday).

2. *Identifying the dimension set of each lattice*: We compute the Maximal Frequent Sets of attributes [70] in the CFS. Each of the found sets is the root of one lattice. We further filter them so that each lattice:
   - Has at most N attributes, and
   - Does not contain attributes that are derived one from the other, e.g., *nationality* and *numOfNationalities* are not allowed as dimensions of the same lattice.

   Although we aim to offer a general approach, we also note that the readability of MDAs by human users is maximized at levels of relatively low dimensionality, i.e., $N \in \{1, 2, 3, 4\}$.

3. *Identifying the measures in each lattice*: Once a lattice acquires dimensions $\mathcal{D}_i$, we assign it a measure set $\mathcal{M}_i$ that comprises all the analyzed attributes of the CFS except those in $\mathcal{D}_i$, and those that are derived from a dimension in $\mathcal{D}_i$, e.g., *numOfNationalities* cannot be a measure in an aggregate whose dimension is *nationality*.

Several lattices may be found for a CFS, e.g., for CEOs, we have three: {*countryOfOrigin*}, {*nationality*, *numOfCompanies*}, and {*nationality*, *gender*, *company/area*} (Examples 1-3). They might partially overlap in dimensions and/or measures; e.g., Examples 2 and 3 share *nationality*. Spade ensures that the results of evaluated MDAs are reused (not recomputed) in the other lattices where they appear.

Step 4 is **Aggregate Evaluation**. This step triggers the actual evaluation of the enumerated MDAs. To address challenge **C2**, we combine:

- Our novel early-stop technique to quickly prune the unpromising MDAs, and
- Our MVDCube algorithm to efficiently compute the remaining MDAs in a single pass. The final results are produced in an incremental fashion and handled by the **Aggregate Result Manager** (ARM). The ARM stores them and incrementally updates statistics such as minimum and maximum values, as we explain in Section 4.4. These are used to determine the interestingness of the computed MDAs (by applying h) in one pass over their results.

Step 5 finally performs **Top-k Computation**. Once the evaluation is complete, the ARM retrieves all the evaluated MDAs, computes their *interestingness* score by applying h, and returns the k best aggregates. Spade natively supports three interestingness functions, from which the user can choose to suit their preferences:

- Variance,
- Skewness, and
- Kurtosis,

where *variance* can detect deviation from uniform aggregate values, whereas the latter two can detect deviation from a normal distribution of aggregated values over numeric dimensions.

For example, Figure 29(b) shows a histogram that exhibits a peak (outlier) in *sum*(*netWorth*) for Angola, due to Dos Santos, and a heat map where the dark color reflects a particularly low value of *avg*(*age*) of CEOs, again due to Dos Santos. The two aggregates are interesting because of their high variance scores. More sophisticated interestingness functions for insight detection can be applied on the Step 4 results via the ARM; we discuss early-stop extensions in Section 4.5.2.

4.4 LATTICE-BASED COMPUTATION

Recall from Chapter 2, ArrayCube, a classical optimized method of computing all aggregates in a lattice. In this section, we explain its limitations and the errors it makes in our setting. Finally, we present our new algorithm to compute lattices of RDF aggregates correctly and efficiently.

4.4.1 *Incorrectness in the RDF setting*

Results computed by ArrayCube may be *incorrect in the presence of multi-valued dimensions*. Consider our running examples that show CEOs with *various* nationalities and *at most one* gender who manage companies in *several* areas. In a relational DW, each such CEO would be stored as a tuple in the fact table, and their multiple nationalities (respectively, company areas) would be modeled as a dimension table associating them with each of their nationalities (company areas). We could then find the result for Example 3 with a query q that joins all the relations, groups the data by the dimensions, and finally aggregates the measure. To evaluate all MDAs in the lattice determined by the dimensions in Example 3, ArrayCube would use the MMST in Figure 4(b) and compute the aggregate A_1 by means of q, using its result to compute the rest of the lattice.

Figure 5 shows the result of A_1 when applied to the two CEOs in Figure 29. The tuples t_1 to t_3 are derived from Dos Santos (the RDF node n_1), whereas t_4 to t_{11} are due to Carlos Ghosn (the RDF node n_2). Since n_2 lacks gender information, the tuples t_4 to t_{11} have *gender*=`null`. We need to keep them to compute the rest of the lattice correctly. Since n_2 has valid values for *nationality* and *company/area*, we must count this CEO when computing aggregates over one or both of these dimensions, e.g., A_4 in Figure 5. We obtain the result of A_2 by aggregating A_1's result to project away the *nationality* dimension. For instance, the tuples t_4, t_6, t_8, and t_{10}, which are all associated with n_2, collapse into the tuple t_4 in A_2 where now this CEO counts as *four*. Then, A_2 is further aggregated by projecting away *company/area* to compute A_3 and separately *gender* to compute A_4. The cardinality "bug" introduced in A_2 *propagates* down the lattice. In A_4's result, we find *five* CEOs managing Manufacturer companies, whereas there are only *two*. A similar error occurs in A_3 where we count *three* female CEOs (which is wrong) because the tuples t_1 to t_3 of A_2 are aggregated into the same tuple and are, thus, counted three times (although they all represent n_1). Since A_3 will not be further aggregated, we can remove the tuple having *gender*=`null`.

The above example shows that multiple values for a dimension may lead to errors *when an aggregate is computed from one of its parents*. To correctly compute the whole lattice from the root aggregate, naïve solutions may:

1. Require that each CEO fact be represented by at most one tuple, e.g., in our example, by ignoring all but one of Ghosn's nationalities (company areas); this would clearly miss an interesting part of the data;

2. Compute each of the 2^N aggregates in the lattice separately, missing the benefits of efficient one-pass algorithms; this would entail a high run time overhead.

However, computing each lattice aggregate from the base data is very inefficient.

Interestingly, if we alter the query q to *count distinct CEOs in each group* (in lieu of $count(*)$), no errors occur in the result of Example 3. By design, ArrayCube cannot compute aggregates including *distinct*: instead, it computes all aggregates from the result of the lattice root, where information about individual facts is no longer present. Other one-pass algorithms for lattice-based aggregate computation, such as PostgreSQL's GROUP BY CUBE implementation [71] (PGCube), do support the counting of distinct values and can thus be used to obtain the correct result for Example 3. Thus, one could get correct result for Example 3 by asking *count*(distinct *CEO*) rather than $count(*)$. However, in the presence of multi-valued dimensions, computing aggregates from the result of one of their parents in the lattice *may still lead to wrong results*, as illustrated in the following variations of Example 3.

Variation 1. Consider the aggregate *"sum of the net worth of CEOs by nationality, gender, and area of the companies they manage"*. We first augment the data in the root aggregate A_1 with the sum of *netWorth* (*NW*). The tuples t_1 to t_3 contain the *NW* of Dos Santos: \$2.8 billion. The tuples t_4 to t_{11} contain the *NW* of Ghosn: \$120 million. We then compute the sum of *NW* by *company/area*. The tuples t_2, t_5, t_7, t_9, and t_{11} (all having *company/area*=M) sum up into one tuple, and result in the sums of \$2.8B of Dos Santos (from t_2), and $4 \times$ \$120M of Ghosn (from the other tuples), whereas both CEOs should have contributed exactly once. Moreover, we cannot solve this issue with the *sum*(distinct *NW*) aggregate. If both CEOs had the same *NW*, a *sum*(distinct) would sum *NW* once, instead of (correctly) summing it twice (once per each of them).

Similarly, the following variation illustrates another scenario leading to wrong results.

Variation 2. Consider the aggregate *"average age of CEOs by nationality, gender, and area of the companies they manage"*. We obtain it as *sum*(*age*)/*count*(*age*), i.e., the sum in Variation 1 is divided by 5. Instead, the correct value is sum of ages of Dos Santos and Ghosn divided by 2. As in Variation 1, we cannot solve this issue by using *avg*(distinct *age*).

As our experiments show (Section 4.7.3), the number of incorrectly computed aggregates, and the magnitude of the error itself, can be quite significant. This is because of the flexible RDF model, which allows multi-valued dimensions. Conversely, in a relational DW, once a fact table is joined with dimension tables, ArrayCube assumes that each fact has exactly one value for a dimension (for instance, due to a functional dependency). Below, we formally characterize the situations when ArrayCube introduces errors on RDF data.

ANALYSIS OF ARRAYCUBE ERRORS ON RDF Consider an RDF graph $\mathtt{G}$ and a lattice of N dimensions (2^N nodes) on $\mathtt{G}$. Whether a lattice node can be computed correctly from one of its parents, depends on the presence of *multi-valued dimensions* in the lattice:

Lemma 4 *Let $\mathtt{G}$ be an RDF graph. Let $P = \langle CFS, \mathcal{D}_P, M, f\rangle$, $C = \langle CFS, \mathcal{D}_C, M, f\rangle$ be two aggregates in a lattice on $\mathtt{G}$ such that P is a parent of C, $\mathcal{D}_P = \mathcal{D}_C \cup \{D\}$ where D is a dimension, $f \in \{count(*), count(M), sum(M), avg(M)\}$, and there exists a fact $n \in CFS$ with more than one value along the dimension D. Then, computing $C(\mathtt{G})$ from the result of $P(\mathtt{G})$ may lead to wrong results.*

Proof *Let the fact $n \in CFS$ have the values $n.D = \{a, b\}$ and, for each $D_j \in \mathcal{D}_P$, $D_j \neq D$, $n.D_j = d_j$ and d_j is not $\mathtt{null}$. By definition of P, there exist tuples $t_1, t_2 \in P(\mathtt{G})$ such that $t_1 = (d_1, \ldots, a, \ldots, d_N, v_1)$ and $t_2 = (d_1, \ldots, b, \ldots, d_N, v_2)$, to both of which n contributes. Hence, there exists a tuple $t_3 \in C(\mathtt{G})$ such that $t_3 = (d_1, \ldots, d_N, v_3)$, in which the dimension D does not appear.*

When computing $C(\mathtt{G})$ from $P(\mathtt{G})$, the aggregated value v_3 is obtained from $t_1.v_1$ and $t_2.v_2$ based on the function f. For instance, if f is $count(*)$, the fact n will be counted twice, instead of just once. If f is $sum(M)$, the M value(s) of n will be summed twice, which falsifies the result (except for the particular case where their sum is 0). Computing the avg may similarly lead to wrong results.

How does Lemma 4 impact the one-pass lattice-based computation for a given graph $\mathtt{G}$? We show the following result:

Theorem 8 *Given an RDF graph $\mathtt{G}$ and a lattice on $\mathtt{G}$, let $\mathcal{MD} \subseteq \mathcal{D}$ be the set of all the dimensions for which some fact(s) $n \in CFS$ have more than one value, and let $K > 0$ be the size of $\mathcal{MD}$. Then:*

1. *A one-pass algorithm* cannot *compute correctly all the lattice aggregates.*
2. *The maximum number of MDAs (lattice nodes) that can be computed correctly (depending on the choice of the MMST) is 2^{N-K}.*

Proof *We prove 1. and 2.:*

1. Among the $N \cdot 2^{N-1}$ lattice edges, $K \cdot 2^{N-1}$ are labeled with a dimension from $\mathcal{MD}$, meaning that the dimension is projected away when computation follows this edge. As Lemma 4 shows, if the MMST contains one such edge, the result of the child node of that edge may contain errors. However, no spanning tree, thus, no MMST, can avoid all *edges labeled with a dimension in $\mathcal{MD}$. This is because to go from the root, whose dimensions are $\mathcal{D}$, to a node lacking one dimension $D \in \mathcal{MD}$, by the construction of the lattice, the MMST* must *traverse an edge labeled D.*

2. The lattice nodes that can be computed correctly in one pass (starting from the root's result) are exactly those having all the $\mathcal{MD}$ dimensions: a

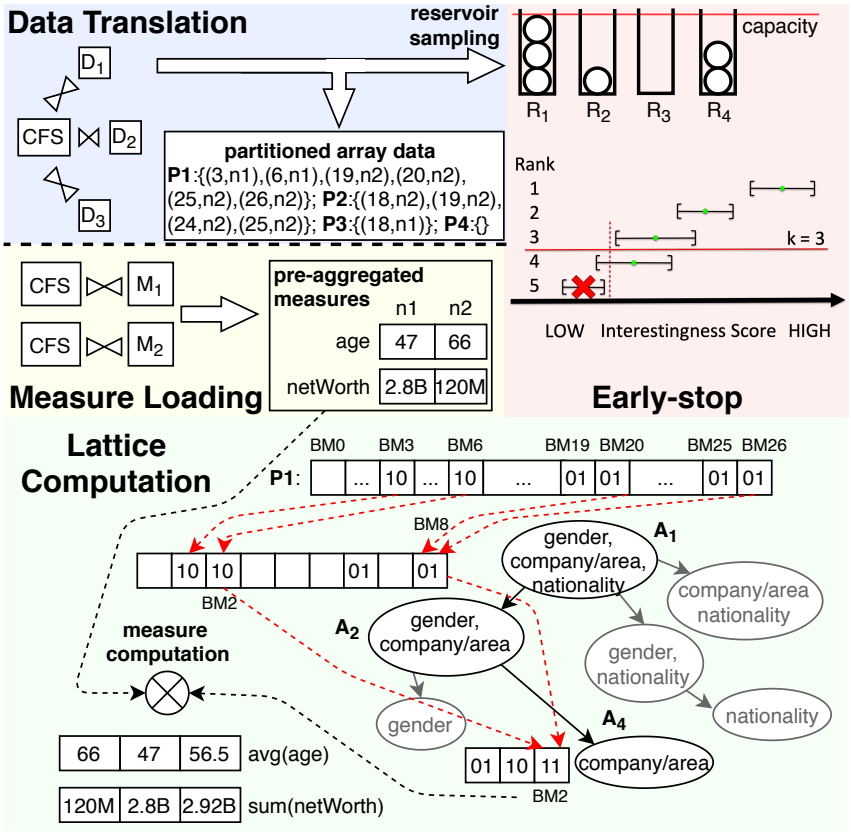


Figure 31: Aggregate evaluation using MVDCube and early-stop.

node lacking one such dimension would be obtained by aggregating a parent's result along that dimension, and thus, by Lemma 4, be wrongly computed. The lattice has 2^{N-K} *such nodes.* Fewer *nodes may be computed correctly if the MMST picks a "wrong" edge, even if it could have avoided doing so.*

4.4.2 *MVDCube algorithm*

We now present **Multi-Valued Data Cube** (MVDCube), our new one-pass MDA evaluation method. Going beyond existing algorithms [22], MVDCube:

1. *Produces correct results* even in the presence of missing or multi-valued dimensions and/or measures,
2. Computes *several aggregate functions* over *a large set of measures* in the same lattice, and
3. Saves computation cost by sharing measures across all lattices from a given CFS.

Before we move forward, we clarify that our RDF database uses the following storage: a CFS is represented by a single-column table

storing the identifiers (IDs) of the facts; for each attribute a, a table t_a stores (s, o) pairs for each (s, a, o) triple in the RDF graph.

Figure 31 depicts the main features of MVDCube. Our MVDCube evaluation method proceeds in the following steps, with the pseudo-code of its core functions shown in Algorithm 6.

BUILDING MMSTS Given a CFS and a set of lattices (identified in Step 3 of Spade's pipeline), each with the dimensions $\mathcal{D}_i$ and the measures $\mathcal{M}_i$, we construct one MMST per lattice as in [22].

DATA TRANSLATION For each lattice, we process the root node by sending a join query to the database to obtain all the CFs that have a value for at least one of the dimensions in $\mathcal{D}_i$. We then translate the join result to lay the data in a *partitioned array representation* of cells. A partition is a set of pairs (cell index, CF). We assign each RDF node a cell index based on its dimensions' values; in the case of multiple values for a dimension, we assign indexes of all corresponding cells. We add the special value `null` in the domain of each dimension to account for missing values. Therefore, each cell is associated with *the set of RDF nodes that correspond to the combination of dimension values that this cell represents*. Like ArrayCube, we take an initial pass over the data to bring it into the array representation, where the (conceptual) multidimensional array is stored as a serialized one-dimensional array. If the data does not fit into the available memory, we partition it, store to disk, and later read back, one partition at a time; otherwise, MVDCube accesses the array directly from the main memory, in a single pass, in subsequent steps.

MEASURE LOADING The measure loading step is performed in parallel to Data Translation. For each measure M in $\mathcal{M}_i$, we query the database to retrieve, for each CF, the preaggregated values of M (which were computed and stored offline). We load the values *ordered by the IDs of the CFs*, and share them among all MMSTs in a given CFS. As they are stored at the granularity of a CF, they can be used to compute aggregate results for all cells, as we describe below.

LATTICE COMPUTATION We then carry out the Lattice Computation step in one pass over the data using the MMST. MVDCube associates an MMST node with a (large) set of aggregates; we denote such a node as $A_i = \langle CFS, \mathcal{D}_j \rangle$. Each node then *represents all the MDAs that have dimensions* $\mathcal{D}_j$ (but might differ in their measure and aggregate function). Suppose that we want to compute the lattice with $\mathcal{D}$={*gender*, *company/area*, *nationality*}, $\mathcal{M}$={*age*, *netWorth*} and that *age* is associated with *avg*, and *netWorth* is associated with *sum*. Node A_2 in Figure 31 represents the two MDAs:

1. Average *age* of CEOs, and

2. Sum of *netWorth* of CEOs, both grouped by *gender* and *company/area*.

In the MMST, we allocate, for each node, the needed memory. We load partitions successively into the root. In Figure 31, we assume that each partition contains 3 distinct values of each dimension, hence 27 cells. For compactness, we encode each set of RDF nodes in a cell using a Roaring Bitmap [72] (also adopted in Spark because of the strong compression and lookup performance). In Figure 31, each cell stores a set of CEOs (a subset of the facts n_1 and n_2). The *bitmaps follow the same ordering of the CFs applied during Measure Loading*. The cell of index 3 in A_1 contains a bitmap of size 2, $\text{BM}_3 = 10$, representing that n_1 is in the set, whereas n_2 is not.

(a) Projection and bitmap propagation. We scan the bitmaps in the cells of the root node and immediately propagate them to the child nodes in the MMST as dimensions are projected away (line 4 in Algorithm 6). We union (OR) the bitmap in each cell in a child node with each bitmap received from the parent (line 9): this models *the contribution of all facts in a parent node to the corresponding cell in the child node*. In particular, as we project away a multi-valued dimension from a parent node to a child node, if a fact has multiple values of the dimension, it belongs to different cells in the parent node, but will be consolidated in the same cell in the child node.

Red arrows in Figure 31 show propagations. For example, the bitmap of cell 2 in node A_4, BM_2, is initially empty (i.e., 00). Then it is updated to 01 when BM_8 from A_2 is propagated, and later to 11 when BM_2 from A_2 is also propagated.

Once a partition is evaluated, we apply the ArrayCube check (Section 2.3) in the nodes to learn if it is time to write results to disk (line 10). If so, we first propagate their memory content to their child nodes (line 11), and then we compute the values of the aggregated measures and store them (line 12).

(b) Measure computation (denoted as $\otimes$). When a node is ready to write to disk, we scan its memory one cell at a time. For each cell:

- We identify the preaggregated measures of each RDF node in the cell's bitmap, and
- We apply the relevant aggregate functions to them.

Note that measure computation is very fast as both the bitmaps and the preaggregated measures are ordered by the fact ID, and can aggregate different measures simultaneously.

Revisit A_4 in Figure 31. Once P1 and P2 are evaluated, A_4 is ready to write current results to disk. We scan the three cell bitmaps, and for each bitmap:

- Identify the age and the net worth of each CEO in the bitmap by accessing the preaggregated measures,

- Aggregate the respective measures by applying *avg* on the age and *sum* on the net worth.

For example, for BM_2, we identify the ages (respectively, net worth) of n_1: 47 ($2.8B) and n_2: 66 ($120M) because they are both present in the bitmap, then compute their average (respectively, sum). The Aggregate Result Manager (line 19) receives the computed measures, and the values of the dimensions obtained from the cell index. For BM_2, the ARM will receive 57.5 and 2.92B, and *company/area*=M. Finally, we empty A_4's memory in the MMST and reuse it to evaluate the aggregate on the next partition of data (line 13).

Algorithm 6: MVDCube(root, partitions)

```
1  Function Main(root, partitions):
2      foreach P ∈ partitions do
3          root.loadPartition(P);
4          root.updateSubtree();
5          root.computeAndStoreAggregatedMeasures();
6  Function updateSubtree():
7      foreach child ∈ children do
8          foreach pair (partition, offset) ∈ memory do
9              child.updateBitmap(partition, offset);
10             if timeToStoreToDisk() then
11                 child.updateSubtree();
12                 child.computeAndStoreAggregatedMeasures();
13                 child.emptyMemory();
14 Function computeAndStoreAggregatedMeasures():
15     foreach pair (partition, offset) ∈ memory do
16         currentBitmap = getBitmap(partition, offset);
17         foreach pair (measure, aggFunction) do
18             aggregatedMeasure = currentBitmap ⊗
                preAggregatedMeasure(measure, aggFunction);
19             resultManager.add(partition, offset, aggregatedMea-
                sure);
```

MEMORY USAGE Our memory analysis builds on the corresponding ArrayCube study [22]. Assuming N dimensions with d distinct values each and c distinct values per partition, the MMST uses at most $M_T = c^N + (d + 1 + c)^{N-1}$ array cells to compute *one* aggregated measure. In MVDCube, the memory for an MMST is also upper bounded by M_T cells. However, cells have a variable size as each of them contains a Roaring Bitmap (RB). For this reason, we provide a

worst-case estimation of MVDCube's memory needs for the MMST and the preaggregated measures:

- The size of an RB used to store Z integers in the interval $[0, u)$ is bound in [72] to $M_{RB} = 2 \cdot Z + 9 \cdot (u/65535 + 1) + 8$, that is, beyond a fixed overhead for u, the universe size, RBs never use more than 2 bytes per integer. In the worst case, we could have $|CFS|$ facts in each cell, occupying a total of $M_T \cdot M_{RB}$ bytes.
- For m measures, MVDCube needs $|CFS| \cdot \sum_{i=1}^{m} |S_{M_i}|$ float numbers in the worst case, where M_i refers to each measure and S_{M_i} is the set of aggregate functions assigned to the measure. As an optimization, we detect, offline, the *numeric* properties having *at most one value* for all their RDF nodes, e.g., the age of CEOs. To save memory, we allocate a single float number for all preaggregated results (*min*, *max*, and *sum*) for such properties.

4.5 EARLY-STOP AGGREGATE PRUNING

To reduce the effort required to compute lattices of aggregates, we have developed a novel technique called **early-stop** (ES).

4.5.1 *Early-stop principle*

Given an aggregate $A = \langle CFS, \mathcal{D}, M, f \rangle$ and an interestingness function h, finding, how interesting A is, amounts to evaluating a query of the form:

SELECT h(aggregated) **FROM**
(**SELECT** $D_1, D_2, \ldots D_N, f(M)$ **AS** aggregated
FROM $CFS^{\mathcal{D},M}$ **GROUP BY** $D_1, D_2, \ldots, D_N$) **AS** inner;

where $CFS^{\mathcal{D},M}$ is CFS joined with dimensions $\mathcal{D}$ and the (preaggregated) measure M. Note that we only need to present the result of the inner query to the user, if A ends up in the top-k. This leads to the following idea: we could reduce the effort to compute some aggregates *if we can determine (with high probability) that they will not be among the k most interesting ones*.

The literature [69], [73] introduced conservative and large-sample confidence intervals as means of estimating *the result* of a query such as inner but not the result of the full nested query, i.e., the *interestingness* score that we aim to obtain. Recent work on visualization recommendation [67] shows how to stop the evaluation of low-utility one-dimensional aggregates early on relational data. In doing so, it relies on a *worst-case* (*conservative*) confidence-interval-based pruning. In contrast, we extend the line of research on aggregate pruning by constructing a *large-sample confidence interval* around the interesting-

ness score estimator. We provide our novel approach and formalize its probabilistic guarantees below.

To enable early-stop pruning, we estimate the interestingness of the aggregate A using an estimator $\widehat{H}_r$, and bound this approximate score within our large-sample confidence interval. (We derive the formula for the interval in Section 4.5.2.) We draw from each aggregate group a sample containing the same number of facts. For the sake of efficiency, our sampling procedure proceeds in batches of a given size. After scanning a batch, we update the estimate of the aggregate's interestingness based on the (preaggregated) measure values of the facts in the batch. To prune some aggregates, if we find that the *upper-bound on the estimate of A's interestingness is lower than the current lower-bound of the k-th best aggregate*, we can give up evaluating A, and thus obtain the top-k aggregates more quickly. The central part of Figure 31 illustrates this with five aggregates and $k = 3$: the fifth aggregate can be stopped after the current batch, whereas the estimation of the fourth aggregate will continue in the next batch. This procedure terminates once the sample is exhausted or no aggregates have been pruned in a given number of batches.

4.5.2 *Estimating the interestingness score*

NOTATION RECALL A **simple random sample** of size r is a vector $(v_1, \ldots, v_r)^\intercal$ of values drawn uniformly without replacement from a population V of size R; the sample is modeled by a set of independent, identically distributed (i.i.d.) random variables $X_1, \ldots, X_r$.

An **estimator** is a random variable equal to a linear or nonlinear combination of $X_1, \ldots, X_r$ (typically modeling a simple random sample). Evaluating the estimator on a vector $(v_1, \ldots, v_r)^\intercal$ of concrete values taken by these random variables yields an **estimation**.

Let S be a statistic of V, $\widehat{S}_r$ be an estimator of the true value of S based on a sample of size r, and $(1-\alpha)$ be a confidence level for $0 \leq \alpha \leq 1$. Then, a $(1-\alpha)$**-confidence interval** (CI) is a random interval such that for each $1 \leq r \leq R$, $P(\widehat{S}_r - \underline{\varepsilon}_r \leq S \leq \widehat{S}_r + \bar{\varepsilon}_r) = 1 - \alpha$. One interval is derived deterministically from one sample; the probability is taken over all such intervals. We denote $L_r = \widehat{S}_r - \underline{\varepsilon}_r$ and $U_r = \widehat{S}_r + \bar{\varepsilon}_r$, respectively, the lower and the upper bounds at $(1-\alpha)$ confidence level on $\widehat{S}_r$. As in [69], the *large-sample confidence interval* contains the true value with the probability approximately equal to $1 - \alpha$.

CONSTRUCTING THE ESTIMATOR We begin by developing formulas for the point estimator $\widehat{H}_r$ of the query's result when the aggregate function (f) in use is *count*, *sum*, or *avg* and the interestingness function (h) is *variance*, *skewness*, or *kurtosis*. We first detail this for *avg*, and variance and then discuss extensions to other functions.

Let $g_1, g_2, \ldots, g_G$ be the aggregate groups of A and $\boldsymbol{\mu} = (\mu_1, \mu_2, \ldots, \mu_G)^\intercal$ be the true result of A, that is, the vector containing, for each group, the average of the preaggregated values of M for facts from that group. Further, for each group g_i, let $\bar{\boldsymbol{Y}}_i = \frac{1}{r}\sum_{j=1}^{r} X_j$ be the sample mean estimator, where the variable X_j has mean μ_i and variance σ_i^2 and models the (preaggregated) measure value of the j-th fact of the sample of size r, drawn from the facts in g_i. Note that, from the Central Limit Theorem (Theorem 5.5.14 in [74]), each $\bar{\boldsymbol{Y}}_i \sim \mathcal{N}(\mu_i, \frac{\sigma_i^2}{r})$ as $r \to \infty$, where $\mathcal{N}(\mu_i, \sigma_i^2)$ is the normal distribution centered in μ_i with standard error σ_i.

We estimate $\widehat{H}_r(\boldsymbol{\mu})$ with $\widehat{H}_r(\bar{\boldsymbol{Y}})$, where $\bar{\boldsymbol{Y}} = (\bar{Y}_1, \bar{Y}_2, \ldots, \bar{Y}_G)^\intercal$ is the vector of all the group estimators. We thus obtain the (unbiased) estimator of the variance of a vector $\boldsymbol{y} = (y_1, y_2, \ldots, y_G)^\intercal$:

$$\widehat{H}_r(\boldsymbol{y}) = \frac{1}{G-1}\sum_{i=1}^{G}\left(y_i - \frac{1}{G}\sum_{j=1}^{G} y_j\right)^2 \tag{1}$$

DERIVING CI BOUNDS We aim at providing a large-sample confidence interval around $\widehat{H}_r(\bar{\boldsymbol{Y}})$. Our formal result is as follows:

Theorem 9 *Let $\widehat{H}_r$ be the estimator of variance. There exists an error $\varepsilon_r > 0$ such that $\widehat{H}_r(\boldsymbol{\mu}) \in [\widehat{H}_r(\bar{\boldsymbol{Y}}) - \varepsilon_r, \widehat{H}_r(\bar{\boldsymbol{Y}}) + \varepsilon_r]$ with the probability approximately equal to $1 - \alpha$.*

Proof *We prove Theorem 9 constructively, thus exhibiting a concrete formula for ε_r. To derive the confidence interval, first, we approximate $\widehat{H}_r(\bar{\boldsymbol{Y}})$ around $\boldsymbol{\mu}$ using the first two terms of its Taylor series expansion: $\widehat{H}_r(\bar{\boldsymbol{Y}}) \approx \widehat{H}_r(\boldsymbol{\mu}) + \nabla\widehat{H}_r(\boldsymbol{\mu}) \cdot (\bar{\boldsymbol{Y}} - \boldsymbol{\mu})$. Then, we apply the Multivariate Delta Method (Theorem 5.5.28 in [74]) to state that*

$$\sqrt{r}\left[\widehat{H}_r(\bar{\boldsymbol{Y}}) - \widehat{H}_r(\boldsymbol{\mu})\right] \xrightarrow{D} \mathcal{N}(0, \tau^2) \tag{2}$$

where $\xrightarrow{D}$ denotes convergence in distribution, $\tau^2 = \sum_{s=1}^{G}\sum_{t=1}^{G}\sigma_{s,t}\frac{\partial\widehat{H}_r(\boldsymbol{\mu})}{\partial y_s}\frac{\partial\widehat{H}_r(\boldsymbol{\mu})}{\partial y_t}$, $\sigma_{s,t} = \mathrm{Cov}(\bar{\boldsymbol{Y}}_s, \bar{\boldsymbol{Y}}_t)$ for $1 \le s, t \le G$. In other words, the difference between the correct value of interestingness, $\widehat{H}_r(\boldsymbol{\mu})$, and that on the estimator, $\widehat{H}_r(\bar{\boldsymbol{Y}})$, converges in distribution to a 0-centered normal distribution.

*To apply this theorem, we must show that (**1**) $\widehat{H}_r$ has continuous first partial derivatives and that (**2**) $\tau^2 > 0$. Condition (**1**) can be easily shown by applying basic calculus on Eq. 1. For (**2**), we assume that $\bar{\boldsymbol{Y}}_1, \bar{\boldsymbol{Y}}_2, \ldots, \bar{\boldsymbol{Y}}_G$ are independent random variables. Hence, for $1 \le s, t \le G$, if $s \neq t$, then $\mathrm{Cov}(\bar{\boldsymbol{Y}}_s, \bar{\boldsymbol{Y}}_t) = 0$, else $\mathrm{Cov}(\bar{\boldsymbol{Y}}_s, \bar{\boldsymbol{Y}}_t) = \mathrm{Var}(\bar{\boldsymbol{Y}}_s) = \frac{\sigma_s^2}{r}$, and $\tau^2 = \sum_{s=1}^{G}\frac{\sigma_s^2}{r}\left(\frac{2}{G-1}\left(\boldsymbol{\mu}_s - \frac{1}{G}\sum_{i=1}^{G}\boldsymbol{\mu}_i\right)\right)^2$ is positive.*

We now move toward a formula for the confidence interval based on the samples in the groups. We derive it by "standardizing" the distribution of

the difference obtained in Eq. 2, and taking quantiles of the standard normal distribution, $\mathcal{N}(0,1)$, as the interval's ends.

Let $\widehat{\tau^2} = \sum_{s=1}^{G} \frac{\widehat{\sigma_s^2}}{r} \left(\frac{2}{G-1} \left(\bar{\mathbf{Y}}_s - \frac{1}{G} \sum_{i=1}^{G} \bar{\mathbf{Y}}_i \right) \right)^2$, *where* $\widehat{\sigma_s^2}$ *are (unbiased) estimators of variances in all the G groups. From the Strong Law of Large Numbers (Theorem 5.5.9 in [74]), we have that* $\lim_{r\to\infty} \widehat{\tau^2} = \tau^2$ *almost surely. Then, applying Slutsky's theorem (Theorem 5.5.17 in [74]), we get* $\sqrt{r}\left[\widehat{H}_r(\bar{\mathbf{Y}}) - \widehat{H}_r(\boldsymbol{\mu})\right] / \sqrt{\widehat{\tau^2}} \xrightarrow{D} \mathcal{N}(0,1)$. *In turn, for large r, we obtain:* $P\left(\left|\widehat{H}_r(\bar{\mathbf{Y}}) - \widehat{H}_r(\boldsymbol{\mu})\right| \leq \varepsilon_r\right) = P\left(\frac{\sqrt{r}\left|\widehat{H}_r(\bar{\mathbf{Y}}) - \widehat{H}_r(\boldsymbol{\mu})\right|}{\sqrt{\widehat{\tau^2}}} \leq \frac{\varepsilon_r\sqrt{r}}{\sqrt{\widehat{\tau^2}}}\right) \approx 2\Phi\left(\frac{\varepsilon_r\sqrt{r}}{\sqrt{\widehat{\tau^2}}}\right) - 1$, *where* Φ *denotes the cumulative distribution function of a normally distributed variable.*

Let z_p *be the* $\frac{p+1}{2}$ *quantile of* Φ. *Solving* $z_p = \frac{\varepsilon_r\sqrt{r}}{\sqrt{\widehat{\tau^2}}}$ *for* ε_r, *gives us* $\varepsilon_r = \sqrt{\frac{z_p^2\widehat{\tau^2}}{r}}$. *Finally, choosing* $z_p = z_{1-\alpha}$ *we obtain the approximation at the desired confidence level:*

$$P\left(\left|\widehat{H}_r(\bar{\mathbf{Y}}) - \widehat{H}_r(\boldsymbol{\mu})\right| \leq \sqrt{\frac{z_{1-\alpha}^2\widehat{\tau^2}}{r}}\right) \approx (1-\alpha)$$

4.5.3 *Other interestingness functions*

To derive confidence intervals for skewness and kurtosis, we follow similar derivations by replacing the definition of $\widehat{H}_r$ (Eq. 1) with their respective formulas. We derive the CIs based on the Delta Method – both cases exhibit continuous first partial derivatives.

SKEWNESS AND KURTOSIS AS INTERESTINGNESS FUNCTIONS In case of skewness, $\widehat{I}_r(\boldsymbol{y}) = \left[\frac{1}{G}\sum_{i=1}^{G}\left(y_i - \frac{1}{G}\sum_{j=1}^{G} y_j\right)^3\right] \cdot \left[\widehat{H}_r(\boldsymbol{y})\right]^{\frac{2}{3}}$. First, we derive $\frac{\partial \widehat{I}_r(\boldsymbol{y})}{\partial y_s}$:

$$\begin{aligned}
\frac{\partial \widehat{I}_r(\boldsymbol{y})}{\partial y_s} &= \frac{\partial}{\partial y_s}\left\{\left[\frac{1}{G}\sum_{i=1}^{G}\left(y_i - \frac{1}{G}\sum_{j=1}^{G} y_j\right)^3\right] \cdot \left[\widehat{H}_r(\boldsymbol{y})\right]^{\frac{2}{3}}\right\} \\
&= \left\{\frac{\partial}{\partial y_s}\left[\frac{1}{G}\sum_{i=1}^{G}\left(y_i - \frac{1}{G}\sum_{j=1}^{G} y_j\right)^3\right]\right\} \cdot \left[\widehat{H}_r(\boldsymbol{y})\right]^{\frac{2}{3}} \\
&+ \left[\frac{1}{G}\sum_{i=1}^{G}\left(y_i - \frac{1}{G}\sum_{j=1}^{G} y_j\right)^3\right] \cdot \left\{\frac{\partial}{\partial y_s}\left[\widehat{H}_r(\boldsymbol{y})\right]^{\frac{2}{3}}\right\}
\end{aligned}$$

Second, we derive the sub-expressions

$$\frac{\partial}{\partial y_s}\left[\frac{1}{G}\sum_{i=1}^{G}\left(y_i-\frac{1}{G}\sum_{j=1}^{G}y_j\right)^3\right]=$$
$$=\frac{3}{G}\left[y_s^2-\frac{1}{G}\left(\sum_{i=1}^{G}\left(y_i^2-\frac{2y_i}{G}\sum_{j=1}^{G}y_j\right)+2y_s\sum_{j=1}^{G}y_j\right)\right]$$

and

$$\frac{\partial}{\partial y_s}\left[\widehat{H}_r(\boldsymbol{y})\right]^{\frac{2}{3}}=\frac{2}{3}\left[\widehat{H}_r(\boldsymbol{y})\right]^{-\frac{1}{3}}\cdot\frac{\partial\widehat{H}_r(\boldsymbol{y})}{\partial y_s}$$

Then, coming back to the original equation, we have that

$$\frac{\partial\widehat{I}_r(\boldsymbol{y})}{\partial y_s}=$$
$$=\frac{3}{G}\left[y_s^2-\frac{1}{G}\left(\sum_{i=1}^{G}\left(y_i^2-\frac{2y_i}{G}\sum_{j=1}^{G}y_j\right)+2y_s\sum_{j=1}^{G}y_j\right)\right]\cdot\left[\widehat{H}_r(\boldsymbol{y})\right]^{\frac{2}{3}}$$
$$+\frac{2}{3}\left[\frac{1}{G}\sum_{i=1}^{G}\left(y_i-\frac{1}{G}\sum_{j=1}^{G}y_j\right)^3\right]\cdot\left[\widehat{H}_r(\boldsymbol{y})\right]^{-\frac{1}{3}}\cdot\frac{\partial\widehat{H}_r(\boldsymbol{y})}{\partial y_s}$$

Therefore, $\frac{\partial\widehat{I}_r(\boldsymbol{y})}{\partial y_s}$, as a combination of:

1. $\widehat{H}_r(\boldsymbol{y})$, which is itself a combination of elementary (thus continuous) functions,
2. $\frac{\partial\widehat{H}_r(\boldsymbol{y})}{\partial y_s}$, which we showed previously to be continuous, and
3. Other elementary (thus continuous) functions

is also continuous.

In case of kurtosis, $\widehat{J}_r(\boldsymbol{y}) = \left[\frac{1}{G}\sum_{i=1}^{G}\left(y_i-\frac{1}{G}\sum_{j=1}^{G}y_j\right)^4\right]\cdot\left[\frac{G-1}{G}\widehat{H}_r(\boldsymbol{y})\right]^{-2}-3$. First, we derive $\frac{\partial\widehat{J}_r(\boldsymbol{y})}{\partial y_s}$:

$$\frac{\partial\widehat{J}_r(\boldsymbol{y})}{\partial y_s}=$$
$$=\frac{\partial}{\partial y_s}\left\{\left[\frac{1}{G}\sum_{i=1}^{G}\left(y_i-\frac{1}{G}\sum_{j=1}^{G}y_j\right)^4\right]\cdot\left[\frac{G-1}{G}\widehat{H}_r(\boldsymbol{y})\right]^{-2}-3\right\}$$
$$=\left\{\frac{\partial}{\partial y_s}\left[\frac{1}{G}\sum_{i=1}^{G}\left(y_i-\frac{1}{G}\sum_{j=1}^{G}y_j\right)^4\right]\right\}\cdot\left[\frac{G-1}{G}\widehat{H}_r(\boldsymbol{y})\right]^{-2}$$
$$+\left[\frac{1}{G}\sum_{i=1}^{G}\left(y_i-\frac{1}{G}\sum_{j=1}^{G}y_j\right)^4\right]\cdot\left\{\frac{\partial}{\partial y_s}\left[\frac{G-1}{G}\widehat{H}_r(\boldsymbol{y})\right]^{-2}\right\}$$

Second, we derive the sub-expressions

$$\frac{\partial}{\partial y_s}\left[\frac{1}{G}\sum_{i=1}^{G}\left(y_i-\frac{1}{G}\sum_{j=1}^{G}y_j\right)^4\right]=$$
$$=\frac{4}{G}\left[y_s^3-\frac{1}{G}\left(\sum_{i=1}^{G}\left(y_i^3-\frac{3y_i^2}{G}\sum_{j=1}^{G}y_j+\frac{3y_i}{G^2}\left(\sum_{j=1}^{G}y_j\right)^2\right)\right.\right.$$
$$\left.\left.+3y_s^2\sum_{j=1}^{G}y_j+\frac{3y_s}{G}\left(\sum_{j=1}^{G}y_j\right)^2\right)\right]$$

and

$$\frac{\partial}{\partial y_s}\left[\frac{G-1}{G}\widehat{H}_r(\boldsymbol{y})\right]^{-2}=\frac{2(G-1)}{G[\widehat{H}_r(\boldsymbol{y})]^3}\cdot\frac{G-1}{G}\frac{\partial\widehat{H}_r(\boldsymbol{y})}{\partial y_s}$$
$$=\frac{2(G-1)^2}{G^2[\widehat{H}_r(\boldsymbol{y})]^3}\cdot\frac{\partial\widehat{H}_r(\boldsymbol{y})}{\partial y_s}$$

Then, coming back to the original equation, we have that

$$\frac{\partial\widehat{J}_r(\boldsymbol{y})}{\partial y_s}=\frac{4}{G}\left[y_s^3-\frac{1}{G}\left(\sum_{i=1}^{G}\left(y_i^3-\frac{3y_i^2}{G}\sum_{j=1}^{G}y_j+\frac{3y_i}{G^2}\left(\sum_{j=1}^{G}y_j\right)^2\right)\right.\right.$$
$$\left.\left.+3y_s^2\sum_{j=1}^{G}y_j+\frac{3y_s}{G}\left(\sum_{j=1}^{G}y_j\right)^2\right)\right]\cdot\left[\frac{G-1}{G}\widehat{H}_r(\boldsymbol{y})\right]^{-2}$$
$$+\frac{2(G-1)^2}{G^2[\widehat{H}_r(\boldsymbol{y})]^3}\left[\frac{1}{G}\sum_{i=1}^{G}\left(y_i-\frac{1}{G}\sum_{j=1}^{G}y_j\right)^4\right]\cdot\frac{\partial\widehat{H}_r(\boldsymbol{y})}{\partial y_s}$$

Therefore, $\frac{\partial\widehat{J}_r(\boldsymbol{y})}{\partial y_s}$, as a combination of:

1. $\widehat{H}_r(\boldsymbol{y})$, which is itself a combination of elementary (thus continuous) functions,
2. $\frac{\partial\widehat{H}_r(\boldsymbol{y})}{\partial y_s}$, which we showed previously to be continuous, and
3. Other elementary (thus continuous) functions

is also continuous.

In general, one can derive similar formulas for any interestingness function that meets conditions (**1**) and (**2**) examined in the proof.

OTHER AGGREGATE FUNCTIONS For *sum*, we estimate the group sizes while sampling and compute the estimate as a product of the *avg* and *count* estimates. For *min* and *max*, we use the sample min and the sample max, respectively, as point estimates; we apply Popoviciu's and

Szőkefalvi-Nagy's inequalities [75] for the upper and lower bounds, respectively.

To obtain the *sum* estimate, we compute the product of the size of the i-th aggregate group c_i and the sample mean. We estimate c_i while sampling during Data Translation: the count in the root node of the lattice is always correct, whereas in the other lattice nodes, depending on the presence of multi-valued dimensions, it may be overestimated. Recall from Section 4.5.2 the sample mean estimator $\bar{\boldsymbol{Y}}_i = \frac{1}{r}\sum_{j=1}^{r} X_j$, and that $\bar{\boldsymbol{Y}}_i \sim \mathcal{N}(\mu_i, \frac{\sigma_i^2}{r})$ as $r \to \infty$. We now construct a new estimator $\boldsymbol{S}_i = \frac{c_i}{r}\sum_{j=1}^{r} X_j = c_i\bar{\boldsymbol{Y}}_i$. As a consequence, we have that $\boldsymbol{S}_i \sim \mathcal{N}(c\mu_i, \frac{c_i^2\sigma_i^2}{r})$ as $r \to \infty$. This leads to the correct *sum* estimate thanks to the estimator mean equal to $c_i\mu_i$. While deriving the CI bounds in the proof, we account for the different variance of the estimator by applying $\mathrm{Var}(\boldsymbol{S}_s) = \frac{c_s^2\sigma_s^2}{r}$ to obtain τ^2.

Finally, the CI bounds are scaled by the constant factor of c_s^2 for each aggregate group w.r.t. the case of the average estimate: the impact of the scaling is hidden within $\widehat{\tau^2}$, the estimator of τ^2. We thus obtain the formula for our *sum*-estimate confidence interval:

$$P\left(\left|\widehat{H}_r(\boldsymbol{S}) - \widehat{H}_r(\boldsymbol{c\mu})\right| \leq \sqrt{\frac{z_{1-\alpha}^2\,\widehat{\tau^2}}{r}}\right) \approx (1-\alpha)$$

where $\boldsymbol{S} = (\boldsymbol{S}_1, \boldsymbol{S}_2, \ldots, \boldsymbol{S}_G)^\intercal$ and $\boldsymbol{c} = (c_1, c_2, \ldots, c_G)^\intercal$ (the correct aggregate group sizes).

Point estimates for *min*, and *max* are sample min, respectively, max: the function applied over the sample, i.e., $\widehat{Z}_r(\boldsymbol{x}) = \min_r(\boldsymbol{x})$ or $\widehat{Z}_r(\boldsymbol{x}) = \max_r(\boldsymbol{x})$. We then bound $\widehat{H}_r(\boldsymbol{y})$, the variance of $\boldsymbol{y} = \widehat{Z}_r(\boldsymbol{x})$, with Popoviciu's inequality for the upper bound: $\widehat{H}_r(\boldsymbol{y}) \leq \frac{1}{4}(\widehat{Z}_r(\boldsymbol{x}) - b)^2$, where b is the lower bound on min (respectively the upper bound on max).

Analogically, we apply Szőkefalvi-Nagy's inequality for the lower bound: $\widehat{H}_r(\boldsymbol{y}) \leq \frac{(\widehat{Z}_r(\boldsymbol{x})-b)^2}{2r}$. We obtain the global statistics for b for each attribute during Online Attribute Analysis step (Section 4.3).

4.5.4 *Plugging early-stop into MVDCube*

We integrate early-stop into MVDCube to speed up Aggregate Evaluation, and thus address challenge **C2**. The evaluation of an MMST begins with the Data Translation step, run *in parallel* with Measure Loading (recall Section 4.4.2). We exploit the data translation to create a stratified sample of facts for the early-stop pruning. Given the MMST, each address in the multidimensional space in the root corresponds to a unique group of facts. We allocate empty reservoirs $R_1, R_2, \ldots, R_G$, one per aggregate group, each with a capacity equal to the sample

size: this way we ensure stratification. While reading each tuple, we determine its group, hence also the reservoir, and either put the fact in or not with some probability. If the reservoir is full, we discard one of the previously inserted facts. This strategy is known as reservoir sampling and guarantees a choice of a simple random sample [76]. Figure 31 shows an on-going sampling process with four reservoirs R_1 to R_4, each of size 3; should R_1 accept a new fact, it will overwrite one of its three facts to avoid an overflow.

The sample thus obtained is used by early-stop as follows. Once the translation is finished, we propagate the facts sampled from the MMST's root down the tree using Roaring Bitmaps as in MVDCube (see bitmap operations in Figure 31): each node in the MMST receives its own sample. Then, we perform the early-stop pruning based on these samples. All the aggregates that have not been pruned (deemed sufficiently interesting) by early-stop are subsequently evaluated by MVDCube.

4.6 IMPLEMENTATION DETAILS

We now provide the implementation details of the Spade system. We describe the external tools and the internal architecture we optimized for run-time efficiency and moderate memory consumption. We show how to leverage a PostgreSQL database to store RDF data and to perform some operations on them. Spade leverages two existing tools developed in the team, namely OntoSQL [48] and RDFQuotient (Chapter 3) which also builds on OntoSQL: both these tools are backed by PostgreSQL. We discuss the offline and online processing in Spade and their internal implementation. Then, we come back to explain external components; we elaborate on the functioning of the Aggregate Result Manager (ARM, recall Section 4.3) and its various backends: raw-CSV-files output, PostgreSQL, Redis, and BerkeleyDB storage. Finally, we comment on the Top-k Computation step.

4.6.1 *Offline processing*

Recall the phases and steps of our approach we laid out in Section 4.3. We commence the offline processing by loading an RDF graph using OntoSQL into a PostgreSQL relational database. OntoSQL adopts the common practice of encoding space-consuming URIs and literals into compact integers, together with a dictionary table that allows going from one to the other. Recall from Section 4.4.2 that, for a given RDF node n, OntoSQL stores all triples of the form x *rdf:type* n in a single-column table t_n holding the subjects x; for each property p other than *rdf:type*, a table t_p stores (`s`, `o`) pairs for each `s p o` triple in the RDF graph. We also saturate the graph upon loading. This

property-oriented layout benefits Spade, unlike RDFQuotient where the single triples table layout was sufficient (compare with Section 3.8).

Further in the offline phase, once the RDF graph is loaded, we compute its typed strong summary (Section 3.4.2) using our RDFQuotient tool and store it in the database. To speed up successive processing, each table in the database is also indexed, with one index per permutation of columns (our tables have one, two, or three columns, either only `s`, `s` and `o` or `s`, `p`, and `o`) and subsequently clustered using the first index.

OFFLINE ATTRIBUTE ANALYSIS Next, we perform the Offline Attribute Analysis. OntoSQL stores integer encodings of the graph labels in a dictionary table, which works like a hash map from the integer key to the string value. At this point, we extend the dictionary table by adding a "clean value" column. The clean-value column cells are equal to the value column cells for URIs, but, for each literal value, we only keep the content between the quotation marks. We use the value for the property data profiling, whereas we use the clean value in the output aggregates.

PROPERTY ENUMERATION Subsequently, we query the summary to find all the direct properties (the properties present in the input graph). For each direct property, we analyze it with the help of the data profiler and SQL queries. In the offline phase, we don't know the CFS yet. Therefore, for each direct property, we collect and set its non-CFS-specific statistics, i.e., statistics on the values that these properties have in the graph independently of a chosen CFS (the usage of the statistics will be explained gradually below):

1. A boolean `isNumeric` flag (we exploit here the data type discovered by the data profiler);
2. The minimum and maximum value (`null` for non-numeric attributes);
3. A boolean `hasOnlyLiteralValues` flag;
4. The mean value length (the average length of the clean-value string);
5. The number of distinct subjects (used to find the almost-one value below);
6. The almost-one ratio: the number of nodes having more than one value of the attribute divided by the number of distinct subjects.

We then store the statistics, for each analyzed direct property, in a dedicated metadata table.

PROPERTY DERIVATION After that, for each direct property, we try to derive new properties from it. We begin the derivation at level 0. We choose the derivation depth (the number of levels) as a parameter in the configuration. Spade natively supports four derivation mechanisms (recall Section 4.3); we can enable or disable them individually. For the path-derived properties, we enumerate all the paths in the summary. We can set the path derivation length to a value greater than one to explore longer paths (multiple hops). In our experiments, we default to paths of length 1 and the derivation depth of 1 as these settings, empirically, proved to lead to the most meaningful results. We only allow count derivation on the properties that are not at-most-one, i.e., their almost-one ratio is not zero. The values of count derived properties are numeric so, to spare fruitless encoding, we store the integers directly in the `o` column. We restrict keyword extraction and language detection derivations to the properties that only have literal values: we don't apply them on URIs or numeric values. Finally, as for direct properties, we analyze each derived property, as well. By the end of this process, we analyze all the attributes (both the direct and the derived ones), and the metadata table contains their statistics.

ATTRIBUTE PREAGGREGATION Based on the statistics gathered as explained above, we can enumerate all the possible aggregate functions that we can apply on an attribute; we store this information in the `AnalyzedAttribute` object. At this stage, we are ready to preaggregate all the attributes based on their assigned aggregate functions: *count*, *sum*, *min*, or *max*; we don't store the average as it is non-distributive. It remains algebraic, though, and we can and do reconstruct it based on the *sum* and *count* totals.

RDF measures chosen during the (online) Aggregate Enumeration step may have multiple values (even though it was rare in our experiments). By preaggregating their values, we can store, for each fact and each aggregate function, one measure value: we exploit this during the (online) Measure Loading step. If the attribute has multi-valued subjects (i.e., it is not at-most-one), is numeric, and we run with early-stop enabled, apart from the regular preaggregated measure values, we also have to store the raw-values list. The reason for this is due to the lower and upper bounds in early-stop that use variance and, therefore, need to compute it over the collection of raw values.

We terminate the offline phase by indexing and clustering all the newly created tables that haven't been so optimized during the graph loading and summarization.

4.6.2 *Online processing*

Given the offline preprocessing, we begin an online phase with the Candidate Fact Set Selection step. Depending on the choice of the

selection mechanism (see Section 4.3), we either can leverage the typed strong summary of the graph to find types or summary nodes or ask a query against the main triple table in the database to find the set of nodes with the properties given by the user. In each case, if a table containing all candidate facts does not exist, we create it; in the type-based selection, we exploit the OntoSQL layout, which provides us a single-column table per each type. Once we select the candidate fact set (CFS), we add a `subjectID` column with an index: this ID is consecutive (unlike the encoding) and helps us perform Data Translation and Measure Loading, where we benefit from the reduced range of fact IDs (keys in hash maps or raw Java array indexes).

ONLINE ATTRIBUTE ANALYSIS Upon the beginning of the online run, we fetch the attribute statistics we computed offline. Based on the fixed CFS, in the Online Attribute Analysis step, we compute the CFS-specific attribute statistics:

- The number of subjects in the CFS;
- The number of distinct subjects in the CFS;
- The number of distinct objects in the CFS;
- The distinct objects to subjects ratio;
- The support of an attribute: the number of distinct subjects in the CFS divided by the size of the CFS;
- The almost-one ratio for the CFS (as it might differ from the almost-one ratio for the whole attribute because some subjects may not be in the CFS).

AGGREGATE LATTICES CONSTRUCTION In the Aggregate Enumeration step, we apply a user-specified threshold on the support (Section 4.3): in our case, we deem a dimension to be frequent if its support is greater than 0.6. Also, we typically set thresholds for the good dimensions to have at least 2 and at most 100 distinct values (objects), and whose ratio of the distinct objects to subjects is at most 0.4. The good measure is numeric or not almost-one. To enumerate multidimensional aggregate lattices, we draw a sample of facts in a CFS. We run a frequent itemset mining algorithm to find sets of dimensions co-occurring on the facts in the CFS to form the lattices. The mining uses the state-of-the-art DFIN algorithm [77].

DATA TRANSLATION We begin the Data Translation step by computing a join between the CFS and all the dimensions of the aggregate lattice's root. As we described in Section 4.4.2, we need to compute the left joins since we need to preserve `null`s. During the translation into the partitioned array representation of cells, we perform the reservoir

sampling to gather the sample for early-stop (if enabled) as explained in Section 4.5.4. One reservoir corresponds to one aggregate group of the aggregate lattice root. Since we examine each tuple from the dimensions join once, the sampling overhead is minimal.

MEASURE LOADING In parallel to Data Translation, we run the Measure Loading step. Here, we load the preaggregated measure tables into raw Java arrays: we create one array per measure-aggregate function combination. To load the measure values specific to the CFS, we join each measure table with the CFS table[1] : we use the `subjectID`s of facts as indexes in the raw Java arrays. All the loaded measure arrays are shared between lattices of the same CFS. As an additional optimization, we use attribute statistics to decide which arrays need to be loaded depending on the measure-aggregate function combination. For instance, if we have to load the *count* preaggregated measure, but we have the *sum* preaggregated measure, and the measure is at-most-one, we can reconstruct *count* from *sum*. Therefore, we only load *count* if we can't reconstruct it from *sum*. As discussed in the offline phase, we may need to load the raw-value list (as a "preaggregated" array) if we need it for early-stop.

EARLY-STOP The moment the longer of the two parallel steps (threads) terminate, we start the Aggregate Evaluation step. Before the MVDCube evaluation, if we enable early-stop, we run an aggregate pruning procedure. In our experiments, Data Translation typically takes much less time than Measure Loading; thus, the early-stop sampling is cost-free. Nevertheless, the pruning itself is a blocking operation, as it depends on the sampling performed in the Data Translation thread and the measure values loaded in the other thread. It must wait for both threads to terminate; therefore, we have to implement it frugally. For early-stop to benefit the Spade run time, the pruning overhead must be lower than the time cost for the regular aggregate evaluation with the MVDCube algorithm. We show in our experiments that this is frequently the case.

EARLY-STOP PRUNING We implement our pruning procedure from Section 4.5.4. We share reservoir samples among all the aggregates in an aggregate lattice, i.e., we share:

- We propagate the samples across the lattice from a parent node to children nodes using bitmaps, similarly as the MVDCube algorithm (compare with Algorithm 6);

1 Due to our bad experience with the PostgreSQL optimizer, for these queries, we manually disable the hash-join, which helped us speed up the simple CFS-measure joins.

- Within the lattice node, each batch of facts drawn from the reservoir sample between aggregates with the same dimension set but different measures.

Additionally, between *avg* and *sum* we share the measure values for the respective facts in the sample. The sample propagation happens once, whereas the batch draws induce new rounds of pruning. We prune aggregates based on the pruning criterion detailed in Section 4.5.1. We terminate the pruning if the sample (usually of the size of 30 facts for us) is exhausted or there have been no aggregates pruned in the specified number of rounds (2 for us). All operations performed during the early-stop pruning are sped up via raw Java arrays to store the sample facts and efficient lookups to the loaded measure arrays. We minimize the object creation in this critical part of the Spade pipeline.

MVDCUBE We back the implementation of our MVDCube evaluation algorithm with PostgreSQL, from where we examine the result of the join in Data Translation. We perform the following algorithmic steps in Java, using efficient RoaringBitmaps to propagate the facts from a parent to children nodes in an aggregate lattice. From Section 4.4.2, recall that during the lattice computation, we fetch data partitions from the disk (in our implementation, they are stored in a PostgreSQL database). However, if we have enough memory to store all the data, we set the partition sizes equal to the number of distinct values of their respective dimensions; this way, we only fetch one partition.

AGGREGATE RESULT MANAGER While evaluating an aggregate lattice, we need to store partially computed aggregate results. For this task, we employ the Aggregate Result Manager (ARM). We have experimented with four backends for the ARM module:

1. Raw-CSV-file storage;
2. PostgreSQL;
3. Redis; and
4. BerkleyDB.

Each backend stores aggregate results at the granularity of aggregate groups. We may write the results to disk into raw CSV files. Although this is the most straightforward choice, it suffers from I/O overheads (even using buffered writers), but - most importantly - from OS limitations: there is a limit on the number of open files that the OS can handle[2] . PostgreSQL is another obvious choice that solves the

2 The limit can be raised, but overall this approach scales poorly, and we avoid it for datasets with a large number of aggregates.

too-many-open-files problem. However, compared to the in-memory databases such as Redis or BerkeleyDB, we found the latter two to perform better. Finally, we chose Redis early as our default backend mainly for its ease of use and good performance. However, to fully justify this choice, the decisive argument for it has to do with another feature of the ARM: the incremental update of the interestingness score. Redis is natively a key-value store, which can handle complex objects (called hashes in Redis terminology). Moreover, Redis offers a Lua programming language scripting API. On the course of Aggregate Evaluation, we store aggregate results group by group. We take advantage of the API to keep track of the incremental ARM statistics over the aggregate result; we update, on-the-fly, the *sum*, *count*, *min*, and *max* aggregated values to later use them to compute the variance score (and normalize it), as we explain below.

TOP-k COMPUTATION We can compute the interestingness score based on the ARM statistics; this can lead to noticeable performance improvements if there are many aggregate groups. For instance, to compute the variance interestingness score (recall Eq. 1), we need to know the mean value of the aggregated measure across all groups: it requires one linear pass over all the groups, and then we still need to do another linear pass to find the sum of the squared distances from the mean and divide it by the number of groups (decreased by 1)[3] .

INTERESTINGNESS SCORE Finally, in RDF graphs, we encounter very diverse measures with different value distributions. To make interestingness scores comparable across all aggregates, we normalize their scores (a single number per aggregate) by scaling their aggregated values (as many as there are aggregate groups). We apply the normalization for the variance score. On the contrary, the skewness and kurtosis are statistical moments: they are themselves defined to inspect the aspects of the data distribution such as skew (for skewness) or "tailedness" (for kurtosis). For these reasons, we don't normalize them.

Let $V = [v_1, \ldots, v_n]$ be a set of values. We call $V' = [v'_1, \ldots, v'_n]$ the set of normalized values if we obtain it through one of the following transformations.

- Through the feature scaling: $v'_i = \frac{v_i - v_{min}}{v_{max} - v_{min}}$;
- Through division by sum: $v'_i = \frac{v_i}{\sum_{j=1}^{n} v_j}$; or
- Through division by mean: $v'_i = \frac{v_i}{\frac{1}{n}\sum_{j=1}^{n} v_j}$.

3 This optimization has an exclusively practical interest as we only optimize the constant, and the variance score remains $O(n)$ anyways. Yet, from our experience, applying it is worthwhile.

Dataset	#triples	#CFSs	#P	#A	#DP				#A
				woD	kw	lang	count	path	wD
Airline [78]	56M	1	30	5,923	0	0	0	0	5,923
CEOs [79]	85k	237	61	159	1	1	37	462	27,860
DBLP [80]	33M	1	21	1	5	3	8	19	961
Foodista [81]	1M	5	13	0	1	1	6	38	14
NASA [82]	99k	10	37	19	3	15	3	87	1,449
Nobel [83]	87k	15	39	58	3	3	18	87	30,658

Table 9: Real datasets used for testing.

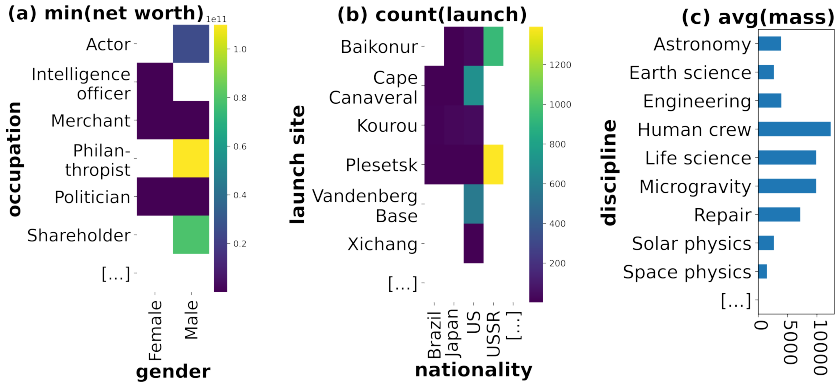


Figure 32: Examples of interesting aggregates found by Spade.

For instance, the normalized variance score remains indifferent to the units of values and the cardinality of the set itself. Variance remains the same even if the mean has been shifted (by some constant) in the process of normalization. At the same time, we need to preserve the shape of the distribution of the values because we can only then truly capture the variance. Normalization by dividing the values by their mean fulfills these requirements, and that's why we chose to use it. (The normalization through division by sum leads to a variance score that is n^2 times lower than the score obtained using values dived by mean.) Also the score obtained from values normalized using the division by mean is equal to $\frac{1}{\bar{v}^2}\,\mathrm{Var}(V)$, where $\bar{v} = \frac{1}{n}\sum_{i=1}^{n} v_i$. Values normalized using sum or mean of the values are well-defined only for the sum different from 0: most measure values are positive numbers, though.

4.7 EXPERIMENTAL EVALUATION

COMPUTATIONAL ENVIRONMENT We ran all experiments on an Intel Xeon CPU E5-2640 v4 @ 2.40GHz, 40 cores (2 sockets with 10 physical cores each, hyper-threading enabled), running CentOS 7

with 90GB for JVM (OpenJDK 1.8) and 30GB of shared buffers for PostgreSQL 12 (with 640 MB working memory).

SYSTEMS We implemented Spade in Java 1.8 (18k lines of code); it relies on OntoSQL 1.0.12, an efficient RDF storage and query answering platform on top of an RDBMS [46], [47], [48] (PostgreSQL in our case). We compare the performance of our *aggregate evaluation* method against the best-effort **baseline**, which uses PostgreSQL's GROUP BY CUBE implementation, since 2016 based on an efficient one-pass computation of all aggregates in a lattice [84], that supports additional features such as *count*(distinct), which were not available in ArrayCube [22]. We denote this by PGCube. As discussed in Section 4.4.1, PGCube may fail to compute correct results in the presence of multi-valued dimensions. However, the support for counting of distinct values may help PGCube correct some wrong results. Thus, we consider two variants:

1. PGCube computing counts using $count(*)$, denoted PGCube*, and
2. PGCube computing counts using *count*(distinct), denoted PGCube^d.

In both cases, *our Java code is at a disadvantage against a C/C++ engine*.

REAL-WORLD GRAPHS Our experiments involve a set of real-application RDF graphs, for which Table 9 shows: the number of triples, the number of CFSs, the number of (direct) properties and derived properties (#P and #DP, respectively) in the graph, and the number of aggregates without and with derivations (#A$_{woD}$ and #A$_{wD}$, respectively). The graph sizes in this work are similar to the real-world dataset sizes used in comparable relational works, e.g., 20k tuples in [66], and up to 60M tuples in [67]. Airline was originally a relational dataset on flight delays used in prior work [67]; we converted it into RDF (each tuple becomes a CF with a fixed set of properties), whereas the others are natively RDF. We discuss differences between this and the other graphs shortly.

4.7.1 *Analysis of example results*

We begin by showing, in Figure 32, example interesting aggregates found by Spade when using variance as an interestingness score:

1. *"Minimum net worth of CEOs by gender and occupation"*:
   - There are two outliers, male philanthropists and male shareholders: their minimum net worth is much higher than others';

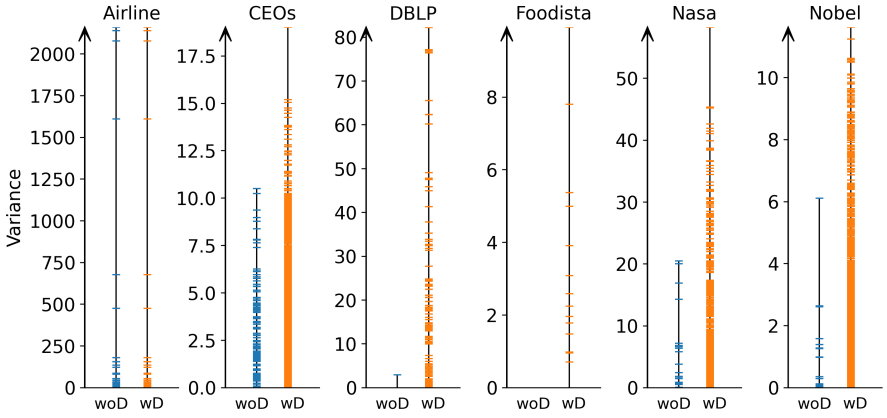


Figure 33: Interestingness of MDAs due to derivations.

- The net worth value is known for all but one occupation for male CEOs, but only in a half of them for female CEOs (e.g., there are no female philanthropists);
- The minimum net worth of female CEOs is nearly the same across occupations.

2. *"Number of launches by launch site and nationality"* in the NASA graph:
   - Very high values for USSR spacecrafts launched from Plesetsk and Bajkonur;
   - The two most used USA launch sites are Cape Canaveral and Vandenberg Base.
3. *"Average mass of spacecrafts by discipline"*:
   - Here, 4 disciplines, i.e., Human crew, Microgravity, Life sciences and Repair stand out with the average spacecraft mass significantly higher than others'.

Nonetheless, many candidate MDAs are uninteresting: Figure 34 shows the aggregate *"minimum number of occupations of CEOs by gender and number of companies"* in the CEOs dataset, where all aggregated values are uniformly equal to 1; or *"average number of launched vehicles by launch site"* in the NASA dataset, where most values are equal to 1, and only 8 out of 35 bars are slightly higher but still less than 1.05. These aggregates don't exhibit any significant outliers and were therefore ranked low by Spade.

This confirms the need for using early-stop to prune such MDAs.

It could have been in principle envisioned to compare the interestingness of the aggregates found by our system with that of some manually chosen aggregates. However, doing so is hampered by the lack of feasible selection methods available to human users. For this reason, the starting point of this work is precisely the observation that

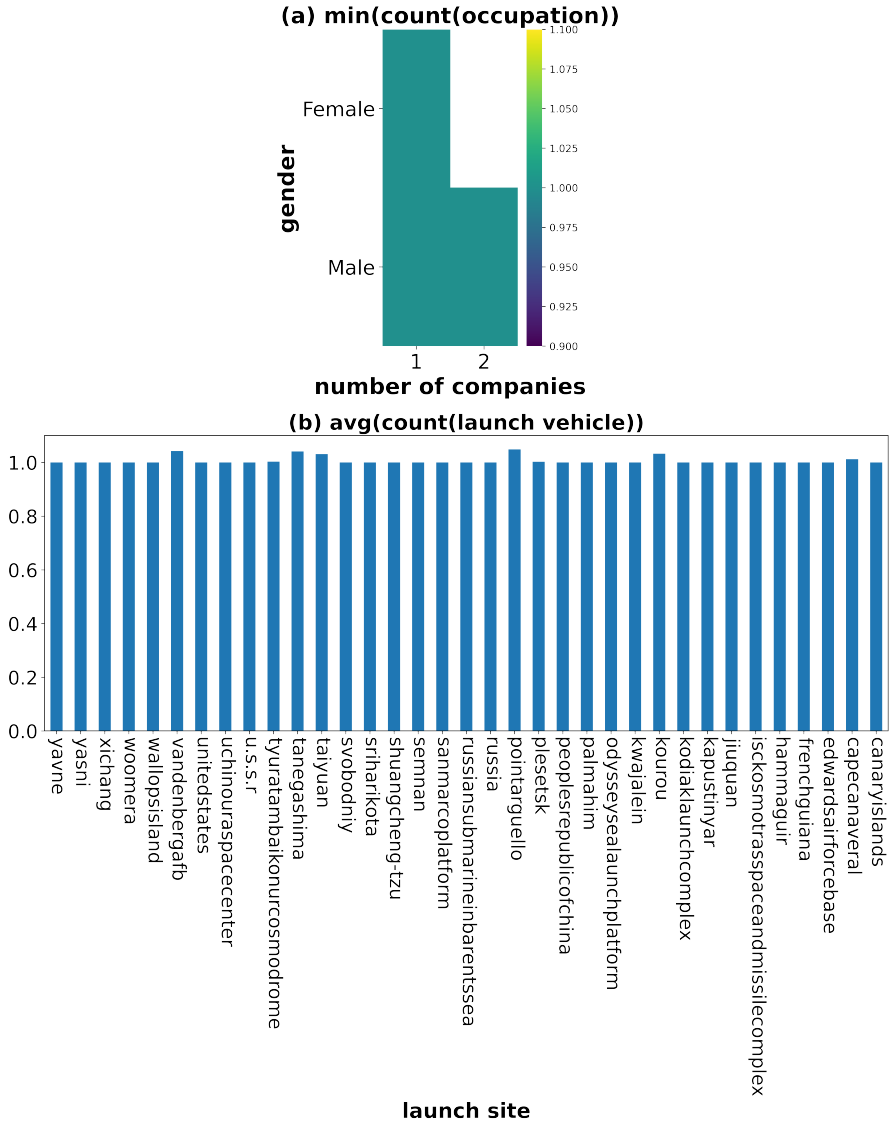


Figure 34: Example uninteresting aggregates found by Spade.

it is very hard to select aggregates manually. There are several reasons for this:

1. The sheer size of the graph impedes human understanding, and it is hard to induce human users to attempt solving such a computationally expensive task at all. Even if they did try to solve it, typically, such users would use a simple SPARQL engine that can evaluate aggregates, and hence they would have to formulate the queries themselves, which requires expertise in writing such complex aggregate queries.

2. Even if we reduce a graph to a modest size, e.g., through summarization [85], [14] or sampling, the reduced graph may not reflect:

a) all possible combinations of facts, dimensions, and measures in the original data;

b) the graph values, e.g., the frequent values and value distributions; or (*c*) any derived properties.

Even under strong (unrealistic) assumptions, e.g., 2a and 2b are both known for a simple, regular RDF graph, users would still not know which aggregates are interesting (e.g., deviating from a uniform distribution) before enumerating and evaluating them all at least partially.

3. Supporting interactions with the system leads users inevitably to inject some information about their preferences in the aggregate selection process. For example, in the NASA dataset, the users may prefer to investigate launches grouped by the launch site rather than the discipline of the spacecraft staff. In contrast, Spade is a fully automated approach to discovering *statistically interesting* aggregates, with no user input required. It defines and enumerates a large set of candidate aggregates by applying heuristics to generate potentially interesting dimensions and measures and evaluates them efficiently.

As examples in Figure 32 show, our highly-ranked results returned from the six real datasets reveal interesting insights. Due to the automatic nature of Spade, in some datasets, there may be a small fraction of aggregates that, despite being statistically sound, are unlikely to be chosen by the user. For example, the aggregate *minimum net worth of CEOs by nationality/image* uses a derived property, *nationality/image*, which is statistically similar to other meaningful dimensions, e.g., *nationality/label*, but the user is unlikely to choose it. This indicates that a "human-in-the-loop" approach can further improve the effectiveness of our automated approach. While for the above example, the user can simply add *nationality/image* to a stop list for dimensions, a full design of "human-in-the-loop" data exploration will be a focus of our future research.

4.7.2 *Benefits of derived properties*

We begin our evaluation by validating the benefits of Derived Property Enumeration (Section 4.3). This step is crucial to address challenge **C1**. We show that it allows us to increase the pool of attributes and to generate a large and rich space of interesting aggregates.

Experiment 1. We compare the results of our analytical strategy when:

1. Only RDF graph properties were used for the analysis (*woD*), and

2. Derived properties were also considered (*wD*).

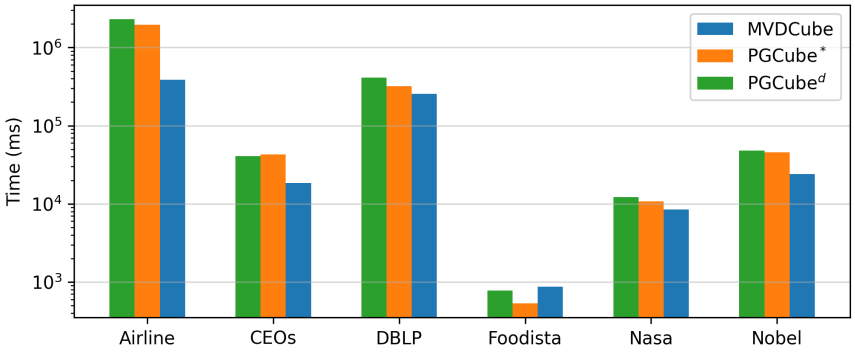


Figure 35: Run times (on log scale) of MVDCube and PGCube.

As Table 9 shows, the Airline dataset (originally relational) leads to no derivations: tuples are not linked to each other, and thus no paths can be derived; it lacks multi-valued attributes, thus no count derivation applies; the data is mostly numeric, so keyword or language attributes are not derived. The other (native RDF) graphs differ drastically: they feature several CFSs, multi-valued properties, links among RDF nodes leading to many path derivations (Table 9 shows counts of path derivations of length 1, as they are the most numerous); textual attributes are also quite frequent. Figure 33 further shows, for each graph, the interestingness of its MDAs (measured with variance) in woD and wD settings (left and right lines, respectively); a horizontal tick in a line depicts an MDA.

Our first main observation, denoted as **remark (R1)**, is that

- *Derivations increase the total number of enumerated MDAs*: for instance, on Foodista, no MDA exists without derivations, whereas we find several by deriving the recipe language, the count of ingredients, etc.; on DBLP, only *year* is a good dimension, whereas through derivations we obtain, e.g., $keyword(title)$;
- *Derivations increase the interestingness of the best aggregates*.

Henceforth, we enable derivations in our experimental analysis.

4.7.3 *Analysis of MVDCube against PGCube*

Our next set of experiments focuses on Aggregate Evaluation, the last step of our online pipeline, where most computation takes place. Since PGCube is not able to prune unpromising aggregates, for fairness, in this section, early-stop is disabled.

Experiment 2. We compare MVDCube with PGCube in *run time* and *quality* (*correctness*). Recall that PGCube's results may be erroneous (Section 4.4.1). We use the six real graphs with derivations.

Regarding the *run time*, Figure 35 shows MVDCube against PGCube* and PGCube^d on our real datasets. We observe that *MVDCube achieves a time gain of 20% to 80% over PGCube* and of 30% to 83%*

Dataset	PGCube*	PGCube^d
	#wrong aggs	#wrong aggs
Airline	0	0
CEOs	4,723	3,998
DBLP	102	87
Foodista	2	0
NASA	378	312
Nobel	4,154	3,821

Table 10: PGCube* and PGCube^d errors on real-graph aggregates.

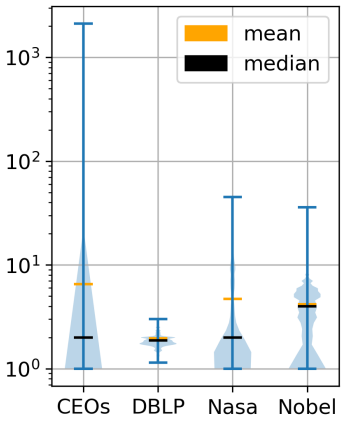


Figure 36: Distribution of PGCube^d errors.

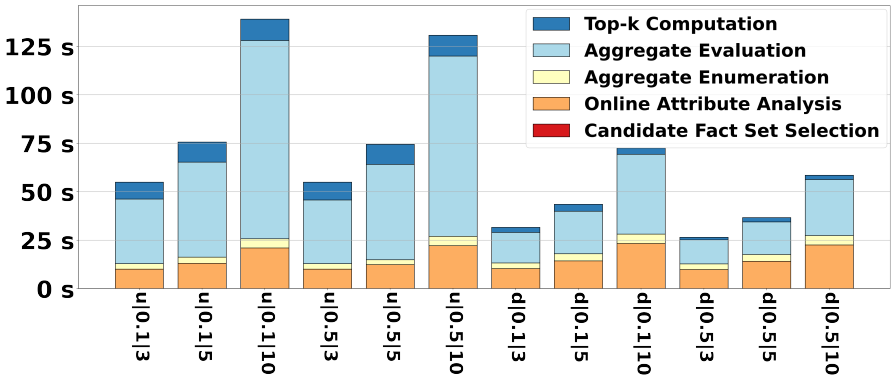


Figure 37: Run times of the steps in Spade's online pipeline.

over PGCube^d on most datasets (**R2**). Specifically, *MVDCube outperforms PGCube when there are many (more than 15) aggregates to evaluate* **(R3)**. This is because MVDCube:

1. Shares measures across all the aggregates from the same CFS, and
2. Computes each aggregate only once, even if it appears in several lattices.

In contrast, PGCube evaluates each lattice in a separate query, each of which joins the facts with the measures. Except for the Foodista dataset, which has a small number of aggregates and both methods run under a second, MVDCube shows significant gains on CEOs, NASA and Nobel Prizes graphs, where many MDAs are evaluated, MVDCube gains 40% over PGCube. Similarly, Airline leads to almost 6k MDAs, the dataset is rather large (6M facts), and the repeated joins are expensive: PGCube* takes 5 times MVDCube's time.

Regarding the *errors*, Table 10 shows, for each graph, the number of aggregates with incorrect results (#wrong aggs) for PGCube* and

PGCube^d. We observe that PGCube* and PGCube^d produce errors in, respectively, 14% and 12% of all computed aggregates **(R4)**. PGCube^d, PGCube's best effort to generate correct results, still produces errors in 9% to 21% of the computed aggregates across different datasets. As shown in Section 4.4.1, errors are related to multi-valued attributes in the data. Indeed, CEOs, NASA, and Nobel Prizes datasets have the greatest number of multi-valued attributes and the highest error, ranging from 12% to 21%.

Experiment 3. We now *quantify the error* in those aggregates that are computed wrongly by PGCube^d. Given an aggregate A, we denote m_j^A the value of the aggregated measure of the j-th group in A, as computed by MVDCube. We denote by p_j^A the value that PGCube^d computes for the same group. As p_j^A can only be *higher than or equal to* the correct value m_j^A, ideally, this ratio should be 1. When an aggregate is shared by two lattices, it can be computed from either lattice, leading to different error ratios. When this happens, we record the maximum error, to measure the "worst-case risk" incurred by evaluating the lattice through PGCube. Each aggregate thus leads to a set of error ratios, one per group. Figure 36 shows their distribution, for *count* and *sum* aggregates, for the four datasets from Table 10 where errors were detected. We note that *errors can easily exceed one order of magnitude* **(R5)**: in 3 out of 4 cases, PGCube^d produces at least 1 tuple whose value is more than 30 times the true value. In CEOs, one group records an error ratio greater than 10^3; it comes from a three-dimensional lattice where all dimensions were multi-valued. Such incorrect values would severely falsify the selection of the k most interesting aggregates.

4.7.4 *Impact of early-stop on MVDCube*

Experiment 4. We next study the effectiveness of our early-stop technique (ES). For our real graphs, Table 11 shows:

1. The evaluation time taken by MVDCube alone,
2. The time with ES enabled, as described in Section 4.5.4,
3. The time gain due to ES,
4. The fraction of aggregates pruned, and
5. The accuracy of ES.

Following [67], if $T_k^{w/o}$ and T_k^w are the sets of the top-k aggregates returned by MVDCube without and with ES, the accuracy is computed as the fraction of true positives in T_k^w: $|T_k^{w/o} \cap T_k^w| / |T_k^{w/o}|$. We show this for $k \in \{3, 5, 10\}$, in keeping with comparable works in a relational DW setting [67] and using a sample size of 60 with 2 batches, a configuration we found empirically to work well. Table 11

leads to two observations. First, *ES can bring significant evaluation time gains*, from 10% to 43% in our experiments; and it *aggressively prunes uninteresting aggregates*, frequently as much as 70% **(R6)**. ES is especially beneficial on graphs with more than 100 aggregates, except for DBLP, where translating the data into an array representation is much more expensive than evaluation, and thus, the saved evaluation effort appears small. In some cases, the impact of ES was negative (and very small), due to a sampling overhead. Second, *MVDCube with ES is often quite accurate* **(R7)**: 100% accuracy is attained in the majority of cases, except for Nobel Prizes, where, e.g., the true top-10 contains aggregates with interestingness score greater than 10.49, whereas ES returns those greater than 9.45.

4.7.5 *Scalability study*

We finally analyze the scalability of our approach and compare it with PGCube, when varying different data characteristics. To be able to fully control them, we designed a **synthetic benchmark** (a set of graphs) with fixed numbers of facts $|CFS|$, N dimensions and M measures. All property values are numeric. We ensure that a single CFS is found and that each dimension D_i, $1 \leq i \leq N$, takes at most 100 values (so that they are considered good dimensions, recall Step 2 in Section 4.3). We denote each graph by $|D_1| : |D_2| : \ldots : |D_N|$, the maximum number of distinct values along each dimension. To obtain realistic distributions of the facts in this multidimensional space, we randomly assign dimension values as in [86], controlled by a sparsity parameter $s \in [0, 1]$. To ensure PGCube correctness, each fact has only one value for each dimension.

Experiment 5. We analyze the performance of *the entire online pipeline of Spade* on benchmark datasets. We use 12 configurations, each having $|CFS|$=1M, 3 dimensions, and 3, 5, or 10 measures. We also use two different combinations of distinct values for dimensions, 100:100:100 (uniform) and 100:5:2 (decreasing), and two different sparsity coefficients, 0.1 and 0.5. In Figure 37, each bar represents one configuration ("u" or "d" for value distribution | sparsity coefficient | number of measures) and reports the total execution time of Spade using MVDCube without early-stop. Each segment of a bar covers one computation step (recall Figure 30). In the pipeline order of steps, we observe that:

1. Candidate Fact Set Selection is too fast to be visible; although there is only one CFS here, *in all our experiments with real graphs*, it was 5-10 ms.

2. Online Attribute Analysis's time is noticeable, between 15% and 37% of the total time, and increases with the number of

	TOP 3					TOP 5					TOP 10				
dataset	MVD	MVD+ES	gain%	pruned%	acc%	MVD	MVD+ES	gain%	pruned%	acc%	MVD	MVD+ES	gain%	pruned%	acc%
Airline	381,710	316,168	**17.17**	**96.13**	**100.00**	369,369	316,885	**14.21**	**93.52**	**100.00**	373,660	330,467	**11.56**	**88.10**	90.00
CEOs	18,114	14,624	**19.27**	**79.21**	33.33	18,685	14,108	**24.50**	**72.86**	**100.00**	18,047	15,108	**16.29**	66.86	**100.00**
DBLP	256,832	255,918	0.36	**88.03**	**100.00**	250,916	248,982	0.77	**85.33**	**100.00**	249,463	256,325	-2.75	**80.85**	**100.00**
Foodista	855	917	-7.25	0.00	**100.00**	1,173	893	23.87	0.00	**100.00**	886	920	-3.84	0.00	**100.00**
NASA	8,633	7,366	**14.68**	**82.40**	**100.00**	8,581	7,750	9.68	**76.54**	80.00	8,458	8,151	3.63	**59.01**	**100.00**
Nobel	24,633	13,897	**43.58**	**95.94**	0.00	24,453	13,829	**43.45**	**95.59**	20.00	23,848	14,267	**40.18**	**94.70**	30.00

Table 11: Early-stop effectiveness on real datasets. All times in ms; in bold: gain% > 10%, pruned% > 70%, and acc% = 100%.

measures: Spade must analyze them before deciding that they are not suitable dimensions.

3. Aggregate Evaluation dominates the processing time; it increases with the number of distinct groups and the number of measures as each measure leads to a different aggregate.

4. The time to select the best aggregates (evaluate their interestingness and pick the top-k) is also noticeable and grows as expected with the number of aggregates.

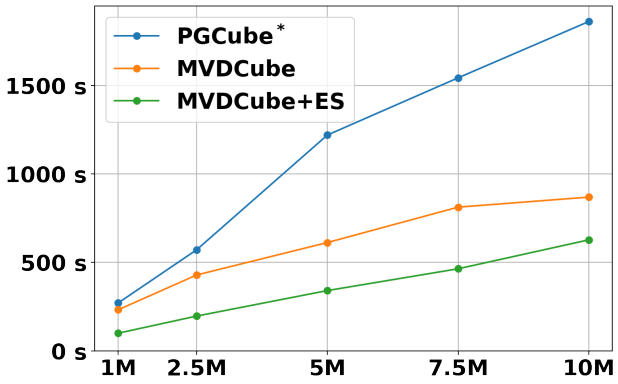
5. Sparsity has a moderate impact.

From these results, we conclude that *for a fixed CFS, Aggregate Evaluation dominates Spade's execution, increasing with the number of distinct groups and the number of measures; Online Attribute Analysis has the second-highest cost, growing with the number of attributes* **(R8)**.

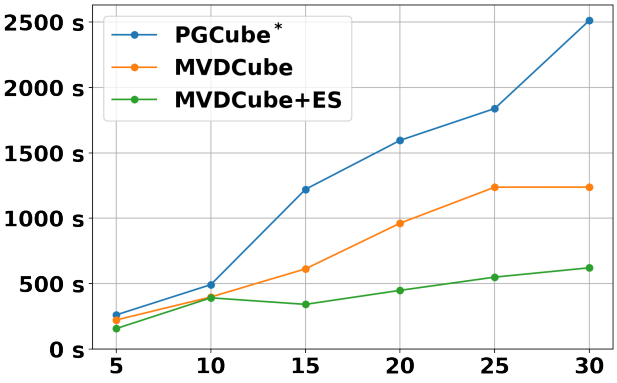
Experiment 6. We now study the impact of $|CFS|$, N, and M on the performance of Spade. As a base configuration, we fixed the synthetic graph with $|CFS| = 5$M, 3 dimensions, and 15 measures (generated as above). For each dimension, we set the uniform value distribution (as above) and sparsity 0.1, as Experiment 5 proved this configuration to be the most difficult. Figures 38a, 38b, 38c show the total execution time of Spade's online pipeline when we vary $|CFS| \in \{1\text{M}, 2.5\text{M}, 5\text{M}, 7.5\text{M}, 10\text{M}\}$, $M \in \{5, 10, 15, 20, 25, 30\}$, and $N \in \{1, 2, 3, 4\}$, respectively; the Aggregate Evaluation step was executed through PGCube*, MVDCube, and MVDCube with early-stop as evaluation modules. We chose PGCube* as on these graphs it is correct, and it is faster than PGCube^d. The figures show that *MVDCube scales linearly when $|CFS|$ and M grow, and its run time increases more with N*; the latter is expected given the high number of lattices that are enabled by more dimensions. Further, Spade using MVDCube *is consistently faster* than using PGCube* by up to 2.9×; it also scales better as $|CFS|$, N and M grow, and *MVDCube with early-stop is consistently the fastest* **(R9)**. Note that in Figure 38b, MVDCube with early-stop took slightly longer for M=10 than for M=15: in these cases, the random samples drawn by early-stop (Section 4.5) were less helpful for M=10 than for M=15.

4.7.6 *Experimental conclusions*

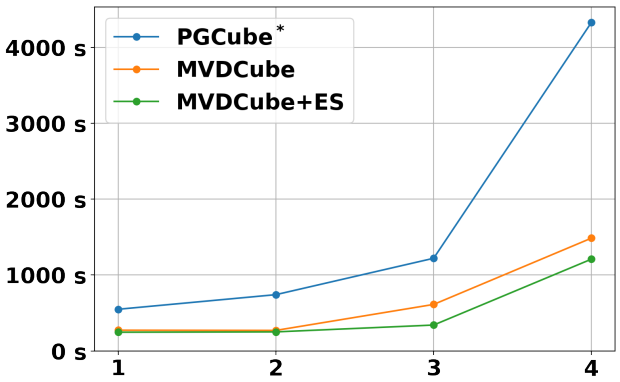
Our experimental results established, first, the need for a novel framework for finding interesting aggregates in RDF graphs: in heterogeneous graphs lacking well-defined facts, dimensions, and measures, Property Derivation increases significantly the space of interesting aggregates **(R1)**. Due to multi-valued dimensions, relational aggregate evaluation algorithms often introduce errors **(R4)**, which can be very significant **(R5)**. On real-world graphs, our algorithm, MVDCube, not



(a) Varying the number of facts $|CFS|$.



(b) Varying the number of measures M.



(c) Varying the number of dimensions N.

Figure 38: Scalability of Spade in the number of facts, measures, and dimensions.

only produces correct results but is also faster (by 20% to 80%) than the best comparable (PostgreSQL) baseline **(R2), (R3)**. Our novel early-stop technique reduces MVDCube's run time by 10% to 43% in many cases **(R6)**, while remaining accurate **(R7)**. In the entire online pipeline of Spade, the most time-consuming steps are Aggregate Evaluation, followed by Online Attribute Analysis **(R8)**. MVDCube consistently outperforms PGCube while scaling in the number of facts, measures, and dimensions; early-stop further improves the performance **(R9)**.

4.8 RELATED WORK

GRAPH EXPLORATION By providing visually meaningful, user-friendly, interactive interfaces, RDF graph visualization [87] allows casual users to access the data in RDF graphs. Based on the graph structure, content, and/or semantics, RDF summarization [14] computes a synopsis (summary) of the data, encapsulating the essential information of the graph from a given perspective. Example-based graph exploration, such as in [88], helps users discover data based on examples they specify. Our work is complementary to these approaches.

INSIGHT EXTRACTION FROM MULTIDIMENSIONAL DATA Another common technique for data exploration is insight extraction from multidimensional data. Research conducted in [66] and [89] provides automatic extraction of the top-k insights from multidimensional *relational* data. An insight is an observation derived from aggregation in multiple steps; it is considered interesting when it is remarkably different from others, or it exhibits a rising or falling trend. Multi-structural databases [90] distribute data across a set of dimensions, compare two sets of data along given dimensions, and separate the data into cohesive groups. A smart drill-down operator [91] is proposed for interactively exploring a relational table to discover groups of tuples that are frequent, specific, and diverse. Works in this area assume a fixed relational schema; more recently, they consider graphs as in [92], but, unlike Spade, they require them to have a very regular and simple structure.

VISUALIZATION RECOMMENDATION SeeDB [67] identifies, in relational data, the one-dimensional aggregates that exhibit the largest deviation between a target dataset and a reference dataset. A study in [93] lays out a recommendation scheme for top-k aggregate visualizations from relational data using a multi-objective utility function to prune as many low-utility views as possible. Recent work [94] shows how to automatically discover the utility function to match the user intentions. DeepEye [95] finds and ranks visualizations by combining a binary classifier, supervised learning, and expert rules.

QAGView [96], [97] provides summaries of high-valued aggregate query answers that ensure properties including coverage, diversity, and relevance, customized based on user preferences. LensXPlain [98] helps users understand answers to aggregate queries by providing the top-k explanations.

In contrast to these works, Spade applies on a schemaless RDF graph, and hence must automatically derive those dimensions and measures that are good candidates to produce some insights.

CUBE COMPUTATION The cube computation lies at the heart of multidimensional data analysis and has been intensely studied [21]. To limit the number of scans of the data and to share computation as much as possible, many algorithms compute the aggregates in the lattice from one of their parents [86], [22], [99]. ArrayCube [22] is a widely accepted algorithm in this category proposing a one-pass solution that simultaneously aggregates along multiple dimensions. Analytical queries have been studied for property graphs in [100]. This ad-hoc OLAP system, similarly to Spade, provides ways of composing aggregates consisting of facts, dimensions, and measures. Conceptual data warehouse designs [101] may account for the double-counting problem in non-strict attribute hierarchies upon a roll-up operation, similar to aggregation of aggregate lattices with multi-valued dimensions discussed in Section 4.4.

4.9 CONCLUSION

SUMMARY Discovering interesting insights from RDF graphs requires automatic, expressive, and efficient methods. We presented Spade, an extensible framework that enumerates a large and rich space of insights in the form of RDF aggregate queries and produces top-k results that maximize a given interestingness function. To the best of our knowledge, Spade is the first system that explores complex RDF data without a clear schema, with many ways to choose facts, dimensions, and measures.

To efficiently explore the large space of candidates aggregates, Spade introduces:

1. MVDCube, an efficient algorithm for evaluating many aggregates in a single pass over the data, 20% to 80% faster than the best comparable method implemented in PostgreSQL, and

2. A novel probabilistic technique that prunes uninteresting aggregates early.

Moreover, Spade scales well with the data size and the number of measures.

FUTURE WORK We plan to study more insight extraction methods to support numeric trends [66], time series, and geo-referenced data.

Another research direction is "human-in-the-loop" data exploration that allows the user to work synergistically with the system to broaden the set of insights discovered from large graphs.

Finally, we envision exploring slicing and dicing operations applied on an aggregate lattice similar to sibling groups proposed in [66]. If an RDF graph contains ontology, we could exploit it to find more candidate fact sets based on the type hierarchies. As in relational data warehouses, we may explore different data granularity levels for aggregates with drill down and roll up. We could use the RDF subclasses and superclasses to enable the respective operations in the RDF setting.

5

CONCLUSION AND PERSPECTIVES

5.1 THESIS CONCLUSION

In this thesis, we provided novel data exploration techniques for RDF graphs. We first showed how we could get non-expert users acquainted with the data through visualizations. We provided RDFQuotient, a tool that enables first-sight structure discovery on RDF graphs. Our compact, meaningful, and informative summary drawings, which we obtain by applying leaf inlining that makes them similar to ER diagrams, provide an overview of the dataset. With this starting point, we further described our new approach for discovering insights in RDF data. As we learned from the RDF graph summarization, performing large-scale processing is hard and requires automatic, expressive, and efficient techniques. To this end, we devised our Spade system, an end-to-end framework that finds the top-k most interesting aggregates in an RDF graph.

RDFQuotient and Spade are new-generation tools for data analysis in the realm of Web data. They leverage the flexible standard of RDF and enable analytics over large, complex, and heterogeneous graphs. We ensure compliance with RDF by taking into account the semantics of the data; this is why our algorithms for summarization and aggregation yield correct results. These new state-of-the-art frameworks are also designed for efficient data processing: our experiments (Sections 3.8, 3.9, 4.7) prove their good performance and scalability.

SUMMARY OF CONTRIBUTIONS We can divide the main contributions we make in this thesis as follows.

In the area of the RDF graph summarization:

1. We developed new quotient RDF graph summaries: weak, strong, typed weak, and typed strong;

2. When we summarize an RDF graph, we showed how to account for the graph's semantics, including its ontology (if present). We further described efficient methods for combining the graph summarization with the saturation: our shortcut procedure speeds up such semantically correct computation.

3. Within our RDFQuotient tool, we implemented our novel summarization algorithms for the centralized setting, in global (two-pass) and incremental (single-pass) variants;

4. We provided new parallel algorithms for memory-constrained environments dedicated for MapReduce-like frameworks that we implemented on the Spark platform; and

5. In the centralized setting, we explained how to build our four summaries efficiently in linear time, while in the parallel setting, we showed that the summarization time scales out linearly as the number of executors grows.

In the area of the data exploration and insight discovery in RDF graphs:

1. We described and implemented our Spade system that explores a large space of candidate RDF aggregates to choose those among them whose interestingness score is the highest;

2. We designed Spade, our new system that discovers facts, dimensions, and measures in complex RDF data without a clear schema;

3. We provided MVDCube, our novel efficient algorithm for evaluating RDF aggregate lattices in one pass over the data;

4. We presented early-stop, our new probabilistic technique for pruning uninteresting aggregates that allows us to reduce the aggregate evaluation effort; and

5. Through experimental study, we demonstrated the scalability of Spade in the number of facts, measures, and dimensions, as well as benefits of early-stop run on top of MVDCube.

5.2 FUTURE WORK PERSPECTIVES

We envision extending our RDFQuotient and Spade line of research by adding interactions with the users. In our recent and ongoing collaboration with Matteo Lissandrini and the Daisy team from Aalborg University, Denmark, we target "human-in-the-loop" data exploration. The starting point of this work is a limitation shared by all fully automated tools meant to help RDF graph data exploration, including RDFQuotient and Spade. Despite universal answers, which our systems provide to their respective problems, they do not take user preferences into account. With this modified objective, we are currently designing a new framework.

Our goal is to reverse-engineer a SPARQL aggregate query on an RDF graph based on a user input table. We begin by presenting an

empty table to the user; then, we ask them to fill in some initial values for the cells and/or headers. Next, we propose some hints to the user to guide further exploration and find the target query. We require our future system to allow interactivity and leave as much flexibility to the user as possible.

BIBLIOGRAPHY

[1] Šejla Čebirić, François Goasdoué, and Ioana Manolescu. "Query-Oriented Summarization of RDF Graphs." In: *BICOD*. 2015.

[2] Šejla Čebirić, François Goasdoué, and Ioana Manolescu. "Query-Oriented Summarization of RDF Graphs (demonstration)." In: *PVLDB* 8.12 (2015). URL: http://www.vldb.org/pvldb/vol8/p2012-cebiric.pdf.

[3] Šejla Čebirić, François Goasdoué, and Ioana Manolescu. "A Framework for Efficient Representative Summarization of RDF Graphs." In: *ISWC (poster)*. 2017.

[4] Šejla Čebirić, François Goasdoué, and Ioana Manolescu. *Query-Oriented Summarization of RDF Graphs*. Research Report RR-8920. https://hal.inria.fr/hal-01325900/file/RR.pdf. INRIA, June 2017. URL: https://hal.inria.fr/hal-01325900v5.

[5] Paweł Guzewicz and Ioana Manolescu. "Quotient RDF Summaries Based on Type Hierarchies." In: *DESWeb Workshop*. Paris, France, Apr. 2018. URL: https://hal.inria.fr/hal-01721163.

[6] Yanlei Diao, Ioana Manolescu, and Shu Shang. "Dagger: Digging for Interesting Aggregates in RDF Graphs." In: *ISWC Posters & Demonstrations and Industry Tracks*. CEUR-WS.org, 2017.

[7] Ioana Manolescu and Mirjana Mazuran. "Speeding up RDF Aggregate Discovery through Sampling." In: *EDBT Workshops*. CEUR-WS.org, 2019.

[8] W3C. *Resource Description Framework*. http://www.w3.org/RDF/.

[9] W3C. *SPARQL 1.1 Query Language*. http://www.w3.org/TR/sparql11-query/. Mar. 2013. URL: https://www.w3.org/TR/sparql11-query/ (visited on 01/28/2021).

[10] Dario Colazzo, François Goasdoué, Ioana Manolescu, and Alexandra Roatiş. "RDF Analytics: Lenses over Semantic Graphs." In: *WWW*. Seoul, South Korea: Association for Computing Machinery, 2014. DOI: 10.1145/2566486.2567982. URL: https://hal.inria.fr/hal-00960609.

[11] Elham Akbari Azirani, François Goasdoué, Ioana Manolescu, and Alexandra Roatis. "Efficient OLAP operations for RDF analytics." In: *ICDE Workshops*. Seoul, South Korea: IEEE, Apr. 2015, pp. 71–76. DOI: 10.1109/ICDEW.2015.7129548. URL: https://hal.inria.fr/hal-01187448.

[12] Adnene Belfodil, Sylvie Cazalens, Philippe Lamarre, and Marc Plantevit. "Identifying exceptional (dis)agreement between groups." In: *Data Mining and Knowledge Discovery* 34.2 (2020), pp. 394–442. DOI: 10.1007/s10618-019-00665-9. URL: https://hal.archives-ouvertes.fr/hal-02383776.

[13] You Wu, Pankaj K. Agarwal, Chengkai Li, Jun Yang, and Cong Yu. "Computational Fact Checking through Query Perturbations." In: *ACM Transactions on Database Systems* 42.1 (2017). ISSN: 0362-5915. DOI: 10.1145/2996453. URL: https://doi.org/10.1145/2996453.

[14] Sejla Cebiric, François Goasdoué, Haridimos Kondylakis, Dimitris Kotzinos, Ioana Manolescu, Georgia Troullinou, and Mussab Zneika. "Summarizing Semantic Graphs: A Survey." In: *The VLDB Journal* 28.3 (2018). URL: https://hal.inria.fr/hal-01925496.

[15] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. ISBN: 0-201-53771-0.

[16] François Goasdoué, Ioana Manolescu, and Alexandra Roatiş. "Efficient query answering against dynamic RDF databases." In: *EDBT*. 2013.

[17] François Goasdoué, Ioana Manolescu, and Alexandra Roatis. "Efficient Query Answering against Dynamic RDF Databases." In: *EDBT*. Genoa, Italy: Association for Computing Machinery, 2013. DOI: 10.1145/2452376.2452412. URL: https://hal.inria.fr/hal-00804503.

[18] Christian Jensen, Torben Bach Pedersen, and Christian Thomsen. 2010, p. 111. DOI: https://doi.org/10.2200/S00299ED1V01Y201009DTM009.

[19] Torben Bach Pedersen, Christian S. Jensen, and Curtis E. Dyreson. "Extending Practical Pre-Aggregation in On-Line Analytical Processing." In: *Proceedings of the 25th International Conference on Very Large Data Bases*. VLDB '99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, pp. 663–674. ISBN: 1558606157.

[20] Carlos A. Hurtado, Claudio Gutierrez, and Alberto O. Mendelzon. "Capturing Summarizability with Integrity Constraints in OLAP." In: *ACM Trans. Database Syst.* 30.3 (Sept. 2005), pp. 854–886. ISSN: 0362-5915. DOI: 10.1145/1093382.1093388. URL: https://doi.org/10.1145/1093382.1093388.

[21] Konstantinos Morfonios, Stratis Konakas, Yannis E. Ioannidis, and Nikolaos Kotsis. "ROLAP implementations of the data cube." In: *ACM Computing Surveys* 39.4 (2007), p. 12.

[22] Yihong Zhao, Prasad Deshpande, and Jeffrey F. Naughton. "An Array-Based Algorithm for Simultaneous Multidimensional Aggregates." In: *SIGMOD*. Tucson, Arizona, USA: Association for Computing Machinery, 1997, pp. 159–170.

[23] Yike Liu, Tara Safavi, Abhilash Dighe, and Danai Koutra. "Graph Summarization Methods and Applications: A Survey." In: *ACM Comput. Surv.* 51.3 (June 2018). ISSN: 0360-0300. DOI: 10.1145/3186727. URL: http://doi.acm.org/10.1145/3186727.

[24] Monika Rauch Henzinger, Thomas A. Henzinger, and Peter W. Kopke. "Computing Simulations on Finite and Infinite Graphs." In: *FOCS*. 1995.

[25] Tova Milo and Dan Suciu. "Index Structures for Path Expressions." In: *ICDT*. 1999. DOI: 10.1007/3-540-49257-7_18. URL: http://dx.doi.org/10.1007/3-540-49257-7%5C%5F18.

[26] Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, and Henry F. Korth. "Covering indexes for branching path queries." In: *SIGMOD*. 2002. DOI: 10.1145/564691.564707. URL: http://doi.acm.org/10.1145/564691.564707.

[27] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon, and Ehud Gudes. "Exploiting Local Similarity for Indexing Paths in Graph-Structured Data." In: *ICDE*. 2002. DOI: 10.1109/ICDE.2002.994703.

[28] Qun Chen, Andrew Lim, and Kian Win Ong. "$D(K)$-index: An Adaptive Structural Summary for Graph-structured Data." In: *SIGMOD*. 2003.

[29] Mariano P. Consens, Renée J. Miller, Flavio Rizzolo, and Alejandro A. Vaisman. "Exploring XML web collections with DescribeX." In: *TWEB* 4.3 (2010). DOI: 10.1145/1806916.1806920. URL: http://doi.acm.org/10.1145/1806916.1806920.

[30] Thanh Tran, Günter Ladwig, and Sebastian Rudolph. "Managing Structured and Semistructured RDF Data Using Structure Indexes." In: *IEEE TKDE* 25.9 (2013). DOI: 10.1109/TKDE.2012.134. URL: http://doi.ieeecomputersociety.org/10.1109/TKDE.2012.134.

[31] Alexander Schätzle, Antony Neu, Georg Lausen, and Martin Przyjaciel-Zablocki. "Large-scale Bisimulation of RDF Graphs." In: *SWIM Workshop*. 2013. ISBN: 978-1-4503-2194-5. DOI: 10.1145/2484712.2484713. URL: http://doi.acm.org/10.1145/2484712.2484713.

[32] Stéphane Campinas, Renaud Delbru, and Giovanni Tummarello. "Efficiency and precision trade-offs in graph summary algorithms." In: *IDEAS*. 2013.

[33] Paweł Guzewicz and Ioana Manolescu. *The project website of the RDFQuotient project*. 2018. URL: https://rdfquotient.inria.fr/ (visited on 07/02/2021).

[34] Günes Aluç, Olaf Hartig, M. Tamer Özsu, and Khuzaima Daudjee. "Diversified Stress Testing of RDF Data Management Systems." In: *ISWC*. 2014, pp. 197–212.

[35] François Goasdoué, Pawel Guzewicz, and Ioana Manolescu. "Incremental structural summarization of RDF graphs." In: *EDBT*. Lisbon, Portugal, Mar. 2019. URL: https://hal.inria.fr/hal-01978784.

[36] Roy Goldman and Jennifer Widom. "DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases." In: *VLDB*. 1997. URL: http://www.vldb.org/conf/1997/P436.PDF.

[37] Yuanyuan Tian, Richard A Hankins, and Jignesh M Patel. "Efficient aggregation for graph summarization." In: *SIGMOD*. ACM. 2008.

[38] Shahan Khatchadourian and Mariano P. Consens. "ExpLOD: Summary-Based Exploration of Interlinking and RDF Usage in the Linked Open Data Cloud." In: *ESWC*. 2010.

[39] Wenfei Fan, Jianzhong Li, Xin Wang, and Yinghui Wu. "Query preserving graph compression." In: *SIGMOD*. 2012. DOI: 10.1145/2213836.2213855. URL: http://doi.acm.org/10.1145/2213836.2213855.

[40] Mariano P. Consens, Valeria Fionda, Shahan Khatchadourian, and Giuseppe Pirrò. "S+EPPs: Construct and Explore Bisimulation Summaries + Optimize Navigational Queries; all on Existing SPARQL Systems (demonstration)." In: *PVLDB* 8.12 (2015). URL: http://www.vldb.org/pvldb/vol8/p2028-consens.pdf.

[41] Andrey Gubichev and Thomas Neumann. "Exploiting the query structure for efficient join ordering in SPARQL queries." In: *EDBT*. 2014, pp. 439–450.

[42] Thomas Neumann and Guido Moerkotte. "Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins." In: *ICDE*. 2011.

[43] Wangchao Le, Feifei Li, Anastasios Kementsietsidis, and Songyun Duan. "Scalable Keyword Search on Large RDF Data." In: *IEEE TKDE* 26.11 (2014). DOI: 10.1109/TKDE.2014.2302294. URL: http://dx.doi.org/10.1109/TKDE.2014.2302294.

[44] François Goasdoué, Paweł Guzewicz, and Ioana Manolescu. "RDF graph summarization for first-sight structure discovery." In: *The VLDB Journal* 29.5 (2020), pp. 1191–1218. DOI: 10.1007/s00778-020-00611-y. URL: https://hal.inria.fr/hal-02530206.

[45] Zvi Galil and Giuseppe F. Italiano. "Data Structures and Algorithms for Disjoint Set Union Problems." In: *ACM Comput. Surv.* 23.3 (1991), pp. 319–344.

[46] Damian Bursztyn, François Goasdoué, and Ioana Manolescu. "Teaching an RDBMS about ontological constraints." In: *PVLDB* 9.12 (2016), pp. 1161–1172.

[47] Maxime Buron, François Goasdoué, Ioana Manolescu, and Marie-Laure Mugnier. "Reformulation-based query answering for RDF graphs with RDFS ontologies." In: *ESWC*. Portorož, Slovenia: Association for Computing Machinery, 2019. URL: https://hal.archives-ouvertes.fr/hal-02051413.

[48] Maxime Buron, François Goasdoué, Ioana Manolescu, and Marie-Laure Mugnier. "Ontology-Based RDF Integration of Heterogeneous Data." In: *EDBT*. Copenhagen, Denmark: Association for Computing Machinery, 2020. URL: https://hal.inria.fr/hal-02446427.

[49] Christian Bizer and Andreas Schultz. "The Berlin SPARQL Benchmark." In: *Int. J. Semantic Web Inf. Syst.* 5.2 (2009). DOI: 10.4018/jswis.2009040101. URL: http://dx.doi.org/10.4018/jswis.2009040101.

[50] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. "LUBM: A benchmark for OWL knowledge base systems." In: *J. Web Sem.* 3.2-3 (2005).

[51] Shahan Khatchadourian and Mariano P. Consens. "Constructing Bisimulation Summaries on a Multi-Core Graph Processing Framework." In: *GRADES Workshop*. 2015. DOI: 10.1145/2764947.2764955. URL: http://doi.acm.org/10.1145/2764947.2764955.

[52] Saket Navlakha, Rajeev Rastogi, and Nisheeth Shrivastava. "Graph summarization with bounded error." In: *SIGMOD*. 2008. DOI: 10.1145/1376616.1376661. URL: http://doi.acm.org/10.1145/1376616.1376661.

[53] Kifayat-Ullah Khan, Waqas Nawaz, and Young-Koo Lee. "Set-based approximate approach for lossless graph summarization." In: *Computing* 97.12 (2015), pp. 1185–1207. DOI: 10.1007/s00607-015-0454-9. URL: http://dx.doi.org/10.1007/s00607-015-0454-9.

[54] Kristen LeFevre and Evimaria Terzi. "GraSS: Graph Structure Summarization." In: *SDM*. 2010. DOI: 10.1137/1.9781611972801.40. URL: http://dx.doi.org/10.1137/1.9781611972801.40.

[55] Renzo Arturo Alva Principe, Blerina Spahiu, Matteo Palmonari, Anisa Rula, Flavio De Paoli, and Andrea Maurino. "ABSTAT 1.0: Compute, Manage and Share Semantic Profiles of RDF Knowledge Graphs." In: *ESWC*. 2018.

[56] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. "Using Graph Summarization for Join-Ahead Pruning in a Distributed RDF Engine." In: *SWIM Workshop*. 2014. DOI: 10.1145/2630602.2630610. URL: http://doi.acm.org/10.1145/2630602.2630610.

[57] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. "SynopSys: large graph analytics in the SAP HANA database through summarization." In: *GRADES*. 2013. URL: http://event.cwi.nl/grades2013/16-Rudolf.pdf.

[58] Chen Chen, Cindy Xide Lin, Matt Fredrikson, Mihai Christodorescu, Xifeng Yan, and Jiawei Han. "Mining Graph Patterns Efficiently via Randomized Summaries." In: *PVLDB* 2.1 (2009). URL: http://www.vldb.org/pvldb/2/vldb09-80.pdf.

[59] Peixiang Zhao, Jeffrey Xu Yu, and Philip S. Yu. "Graph Indexing: Tree + Delta >= Graph." In: *VLDB*. 2007. URL: http://www.vldb.org/conf/2007/papers/research/p938-zhao.pdf.

[60] Alin Deutsch, Mary F. Fernández, and Dan Suciu. "Storing Semistructured Data with STORED." In: *SIGMOD*. 1999.

[61] Philip Bohannon, Juliana Freire, Prasan Roy, and Jérôme Siméon. "From XML Schema to Relations: A Cost-Based Approach to XML Storage." In: *ICDE*. 2002.

[62] Mussab Zneika, Dan Vodislav, and Dimitris Kotzinos. "Quality Metrics For RDF Graph Summarization." In: *Semantic Web* (2018).

[63] Maxime Buron, François Goasdoué, Ioana Manolescu, Tayeb Merabti, and Marie-Laure Mugnier. "Revisiting RDF storage layouts for efficient query answering." In: *Proceedings of the 12th International Workshop on Scalable Semantic Web Knowledge Base Systems co-located with 19th International Semantic Web Conference (ISWC 2020), Athens, Greece, November 2, 2020*. Ed. by Thorsten Liebig, Achille Fokoue, and Zhe Wu. Vol. 2757. CEUR Workshop Proceedings. CEUR-WS.org, 2020, pp. 17–32. URL: http://ceur-ws.org/Vol-2757/SSWS2020%5C%5Fpaper2.pdf.

[64] Nelly Barret, Ioana Manolescu, and Prajna Upadhyay. "Toward Generic Abstractions for Data of Any Model." In: *BDA 2021 - Informal publication only*. Paris, France, Oct. 2021. URL: https://hal.inria.fr/hal-03344041.

[65] Yanlei Diao, Paweł Guzewicz, Ioana Manolescu, and Mirjana Mazuran. "Spade: A Modular Framework for Analytical Exploration of RDF Graphs (demonstration)." In: *PVLDB*. Los Angeles, California, USA: VLDB Endowment, 2019, pp. 1926–1929. DOI: 10.14778/3352063.3352101. URL: https://hal.inria.fr/hal-02152844.

[66] Bo Tang, Shi Han, Man Lung Yiu, Rui Ding, and Dongmei Zhang. "Extracting Top-K Insights from Multi-dimensional Data." In: *SIGMOD*. Chicago, Illinois, Unirted States: Association for Computing Machinery, 2017, pp. 1509–1524.

[67] Manasi Vartak, Sajjadur Rahman, Samuel Madden, Aditya Parameswaran, and Neoklis Polyzotis. "SeeDB: Efficient Data-Driven Visualization Recommendations to Support Visual Analytics." In: *PVLDB* 8.13 (2015), pp. 2182–2193. ISSN: 2150-8097. DOI: 10.14778/2831360.2831371. URL: https://doi.org/10.14778/2831360.2831371.

[68] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. "Implementing Data Cubes Efficiently." In: *SIGMOD*. Montreal, Quebec, Canada: Association for Computing Machinery, 1996, pp. 205–216.

[69] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. "Online Aggregation." In: *SIGMOD*. Tucson, Arizona, USA: Association for Computing Machinery, 1997, pp. 171–182.

[70] Karam Gouda and Mohammed Javeed Zaki. "Efficiently Mining Maximal Frequent Itemsets." In: *ICDM*. San Jose, California, USA: IEEE, 2001, pp. 163–170.

[71] PostgreSQL Global Development Group. *PostgreSQL 12 CUBE*. 2020. URL: https://www.postgresql.org/docs/12/cube.html (visited on 01/28/2021).

[72] Daniel Lemire, Gregory Ssi Yan Kai, and Owen Kaser. "Consistently faster and smaller compressed bitmaps with Roaring." In: *Software: Practice and Experience* 46.11 (2016), pp. 1547–1569. DOI: 10.1002/spe.2402.

[73] P. J. Haas. "Large-sample and deterministic confidence intervals for online aggregation." In: *SSDBM*. Olympia, Washington, USA: Association for Computing Machinery, 1997, pp. 51–62.

[74] George Casella and Roger Berger. *Statistical Inference*. Pacific Grove, California, USA: Duxbury Resource Center, 2001. ISBN: 0534243126.

[75] T. Popoviciu and Béla Szőkefalvi-Nagy. *Popoviciu's and Szőkefalvi-Nagy's inequalities on variances*. 1935. URL: https://en.wikipedia.org/wiki/Popoviciu%5C%27s%5C%5Finequality%5C%5Fon%5C%5Fvariances (visited on 01/28/2021).

[76] Jeffrey S. Vitter. "Random Sampling with a Reservoir." In: *ACM Transactions on Mathematical Software* 11.1 (1985), pp. 37–57. ISSN: 0098-3500. DOI: 10.1145/3147.3165. URL: https://doi.org/10.1145/3147.3165.

[77] Zhi-Hong Deng. "DiffNodesets: An Efficient Structure for Fast Mining Frequent Itemsets." In: *CoRR* abs/1507.01345 (2015). arXiv: 1507.01345. URL: http://arxiv.org/abs/1507.01345.

[78] Giovanni Gonzalez. *Airline delays causes dataset*. 2016. URL: https://www.kaggle.com/giovamata/airlinedelaycauses (visited on 01/28/2021).

[79] Mirjana Mazuran. *CEOs dataset*. 2020. URL: https://www.dropbox.com/s/af8kzjwesz1vs2y/CEOsWithAllTheirData%5C%5FPlus2hops.nt (visited on 01/28/2021).

[80] Linked Data Fragments. *DBLP dataset*. Additional resources: https://www.rdfhdt.org/datasets/ and https://linkeddatafragments.org/. 2017. URL: http://downloads.linkeddatafragments.org/hdt/dblp-20170124.hdt (visited on 01/28/2021).

[81] Leigh Dodds. *Foodista dataset*. 2011. URL: https://old.datahub.io/dataset/foodista (visited on 01/28/2021).

[82] Leigh Dodds. *NASA dataset*. Additional resources: https://data.nasa.gov/. 2010. URL: https://old.datahub.io/dataset/data-incubator-nasa (visited on 01/28/2021).

[83] The Nobel Prize Committee. *Nobel Prizes dataset*. 2020. URL: http://data.nobelprize.org/dump.nt (visited on 01/06/2020).

[84] PostgreSQL Global Development Group. *PostgreSQL support for GROUPING SETS, CUBE and ROLLUP in one pass over the input*. 2015. URL: https://git.postgresql.org/gitweb/?p=postgresql.git;a=commitdiff;h=f3d3118532175541a9a96ed78881a3b04a057128 (visited on 01/28/2021).

[85] Yike Liu, Tara Safavi, Abhilash Dighe, and Danai Koutra. "Graph Summarization Methods and Applications: A Survey." In: *ACM Computing Surveys* 51.3 (2018). ISSN: 0360-0300. DOI: 10.1145/3186727. URL: https://doi.org/10.1145/3186727.

[86] Sameet Agarwal, Rakesh Agrawal, Prasad Deshpande, Ashish Gupta, Jeffrey F. Naughton, Raghu Ramakrishnan, and Sunita Sarawagi. "On the Computation of Multidimensional Aggregates." In: *VLDB*. Mumbai (Bombay), India: VLDB Endowment, 1996, pp. 506–521.

[87] Laura Po, Nikos Bikakis, Federico Desimoni, and George Papastefanatos. *Linked Data Visualization: Techniques, Tools, and Big Data*. Vol. 10. Morgan & Claypool Publishers, 2020, pp. 1–157. ISBN: 9781681737256. DOI: 10.2200/S00967ED1V01Y201911WBE019.

[88] Matteo Lissandrini, Davide Mottin, Themis Palpanas, and Yannis Velegrakis. "Multi-Example Search in Rich Information Graphs." In: *ICDE*. Paris, France: IEEE, 2018, pp. 809–820. DOI: 10.1109/ICDE.2018.00078. URL: https://doi.org/10.1109/ICDE.2018.00078.

[89] Rui Ding, Shi Han, Yong Xu, Haidong Zhang, and Dongmei Zhang. "QuickInsights: Quick and Automatic Discovery of Insights from Multi-Dimensional Data." In: *SIGMOD*. Amsterdam, The Netherlands: Association for Computing Machinery, 2019, pp. 317–332.

[90] Ronald Fagin, R. Guha, Ravi Kumar, Jasmine Novak, D. Sivakumar, and Andrew Tomkins. "Multi-Structural Databases." In: *PODS*. Baltimore, Maryland: Association for Computing Machinery, 2005, pp. 184–195. ISBN: 1595930620. DOI: 10.1145/1065167.1065191. URL: https://doi.org/10.1145/1065167.1065191.

[91] Manas Joglekar, Hector Garcia-Molina, and Aditya G. Parameswaran. "Interactive Data Exploration with Smart Drill-Down." In: *IEEE Transactions on Knowledge and Data Engineering* 31.1 (2019), pp. 46–60. DOI: 10.1109/TKDE.2017.2685998. URL: https://doi.org/10.1109/TKDE.2017.2685998.

[92] Dritan Bleco and Yannis Kotidis. "Using entropy metrics for pruning very large graph cubes." In: *Information Systems* 81 (2019), pp. 49–62.

[93] Humaira Ehsan, Mohamed A. Sharaf, and Panos K. Chrysanthis. "Efficient Recommendation of Aggregate Data Visualizations." In: *IEEE Transactions on Knowledge and Data Engineering* 30.2 (2018), pp. 263–277. DOI: 10.1109/TKDE.2017.2765634. URL: https://doi.org/10.1109/TKDE.2017.2765634.

[94] Xiaozhong Zhang, Xiaoyu Ge, and Panos K. Chrysanthis. "Interactive View Recommendation with a Utility Function of a General Form." In: *SIGMOD Workshops*. Portland, Oregon, USA: Association for Computing Machinery, 2020.

[95] Yuyu Luo, Xuedi Qin, Nan Tang, Guoliang Li, and Xinran Wang. "DeepEye: Creating Good Data Visualizations by Keyword Search." In: *SIGMOD*. Houston, Texas, USA: Association for Computing Machinery, 2018, pp. 1733–1736.

[96] Yuhao Wen, Xiaodan Zhu, Sudeepa Roy, and Jun Yang. "QAGView: Interactively Summarizing High-Valued Aggregate Query Answers." In: *SIGMOD*. Houston, Texas, USA: Association for Computing Machinery, 2018, pp. 1709–1712.

[97] Yuhao Wen, Xiaodan Zhu, Sudeepa Roy, and Jun Yang. "Interactive Summarization and Exploration of Top Aggregate Query Answers." In: *PVLDB* 11.13 (2018), pp. 2196–2208. DOI: 10.14778/3275366.3275369. URL: http://www.vldb.org/pvldb/vol11/p2196-wen.pdf.

[98] Zhengjie Miao, Andrew Lee, and Sudeepa Roy. "LensXPlain: Visualizing and Explaining Contributing Subsets for Aggregate Query Answers." In: *PVLDB* 12.12 (2019), pp. 1898–1901. DOI: 10.14778/3352063.3352094. URL: http://www.vldb.org/pvldb/vol12/p1898-miao.pdf.

[99] Zhimin Chen and Vivek R. Narasayya. "Efficient Computation of Multiple Group By Queries." In: *SIGMOD*. Baltimore, Maryland, USA: Association for Computing Machinery, 2005, pp. 263–274.

[100] Michael Rudolf, Hannes Voigt, Christof Bornhövd, and Wolfgang Lehner. *Ad-hoc analytical query of graph data*. Apr. 2021.

[101] Alejandro Vaisman and Esteban" Zimányi. "Conceptual Data Warehouse Design." In: *Data Warehouse Systems: Design and Implementation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 89–119. ISBN: 978-3-642-54655-6. DOI: 10.1007/978-3-642-54655-6_4. URL: https://doi.org/10.1007/978-3-642-54655-6_4.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

ECOLE
DOCTORALE

Titre : ExpRalytics: analyse expressive et efficace de graphes RDF

Mots clés : RDF, requêtes exploratoires, Web Sémantique, analyse des données

Résumé : Les données ouvertes sont souvent partagés sous la forme de graphes RDF, qui sont une incarnation du principe Linked Open Data (données ouvertes liées). De telles données n'ont toutefois pas atteint leur entier potentiel d'utilisation et de partage. L'obstacle pour ce faire réside principalement au niveau de la capacité des utilisateurs à explorer, découvrir et saisir le contenu et des graphes RDF; cette tâche est complexe car les graphes sont naturellement hétérogènes, et peuvent être à la fois volumineux et complexes.

Nous proposons de nouvelles méthodes pour résumer de grands graphes de données, avec un accent particulier sur les graphes RDF. À cette fin, nous avons proposé une nouvelle approché pour la construction de résumés structurels de graphes RDF, à savoir RDFQuotient.

Nous considérons aussi le problème d'identifier automatiquement les requêtes d'agrégation les plus intéressantes qui peuvent être évaluées sur un graphe RDF.

Title : ExpRalytics: Expressive and Efficient Analytics for RDF Graphs

Keywords : RDF, exploratory queries, Semantic Web, data analysis

Abstract : Large (Linked) Open Data are increasingly shared as RDF graphs today. However, such data does not yet reach its full potential in terms of sharing and reuse.

We provide new methods to meaningfully summarize data graphs, with a particular focus on RDF graphs. One class of tools for this task are structural RDF graph summaries, which allow users to grasp the different connections between RDF graph nodes. To this end, we introduce our novel RDFQuotient tool that finds compact yet informative RDF graph summaries that can serve as first-sight visualizations of an RDF graph's structure.

We also consider the problem of automatically identifying the k most interesting aggregate queries that can be evaluated on an RDF graph, given an integer k and a user-specified interestingness function. Aggregate queries are routinely used to learn insights from relational data warehouses, and some prior research has addressed the problem of automatically recommending interesting aggregate queries.

Institut Polytechnique de Paris
91120 Palaiseau, France