Hypergraphs in the Service of Very Large Scale Query Optimization. Application : Data Warehousing

Ahcène Boukorca

ENSMA : Ecole Nationale Supérieure de Mécanique et d'Aérotechnique
LIAS : Laboratoire d'Informatique et d'Automatique pour les Systèmes

THESE

pour l'obtention du Grade de

DOCTEUR DE L'ÉCOLE NATIONALE SUPÉRIEURE DE MÉCANIQUE ET D'AÉROTECHNIQUE

(Diplôme National — Arrêté du 7 août 2006)

Ecole Doctorale : Science et Ingénierie pour l'Information, Mathématiques
Secteur de Recherche : INFORMATIQUE ET APPLICATIONS

Présentée par :

Ahcène BOUKORCA

****

Hypergraphs in the Service of Very Large Scale Query Optimization
Application: Data Warehousing

Les Hypergraphes au Service de l'Optimisation de Requêtes à très Large Echelle
Application: Entrepôt de Données

****

Directeurs de Thèse : **Ladjel BELLATRECHE**

Soutenue le 12 Décembre 2016
devant la Commission d'Examen

JURY

Rapporteurs :	**Yannis MANOLOPOULOS**	Professor, Aristotle University of Thessaloniki, Greece
	Sofian MAABOUT	Maître de Conférences (HDR), Université de Bordeaux
Examinateurs :	**Omar BOUSSAID**	Professeur, ERIC, Université de Lyon 2
	Arnaud GIACOMMETI	Professeur, Université de Tours
	Ladjel BELLATRECHE	Professeur, ISAE-ENSMA, Poitiers
	Sid-Ahmed BenAli SENOUCI	Dr, Mentors Graphics, Grenoble

ENSMA : **E**cole **N**ationale **S**upérieure de **M**écanique et d'**A**érotechnique
LIAS : **L**aboratoire d'**I**nformatique d'**A**utomatique pour les **S**ystèmes

THESE

pour l'obtention du Grade de

DOCTEUR DE L'ECOLE NATIONALE SUPERIEURE DE MECANIQUE ET D'AEROTECHNIQUE
(Diplôme National – Arrêté du 7 août 2006)

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

Présentée par :

Ahcène BOUKORCA

****

Hypergraphs in the service of very large scale query optimization. Application: data warehouse

Les hypergraphes au service de l'optimisation de requêtes à très large échelle. Application: entrepôts de données

****

Directeurs de Thèse : Ladjel BELATRECHE

****

Soutenue le 12 Décembre 2016
devant la Commission d'Examen

****

JURY

Acknowledgment

I would like to express my sincere gratitude to my supervisor Ladjel BELLATRECHE for his guidance, suggestions, and encouragement to my research. I have learn a lot from him and I'm very thankful to him for all the opportunities which he gave me in the research domain. His availability and his pedagogical and scientific skills were invaluable. This thesis would not have been possible without the help, support and patience of my supervisor.

I wish to express my deep appreciation to Yannis MANOLOPOULOS and Sofian MAABOUT, for having accepted to report this work and for their valuable comments and suggestions. I also thank Omar BOUSSAID, Arnaud GIACOMMETI and Sid-Ahmed Benali SENOUCI for honoring us in my jury members and their interest to my work.

I wish also to express my deep appreciation to Zoe Faget and Sid-Ahmed Benali Snouci for their help and guidance on the progress of this thesis study, and on the validation of our contribution.

I would like to extend my gratitude to the members of LIAS who accompanied me and played a crucial role in the preparation of this thesis, Specially a thanks for Mr Emmanuel GROLLEAU for kindly welcoming me in the laboratory LIAS for supporting my PhD program.

I would also like to thank Selma BOUARAR et Amira KERKAD for helping me with important comments and suggestions. And a thanks for Soumia BENKRID for his intellectual contributions in Chapter 4. Also thanks to all other lab colleagues (Zouhir, Yacine, Baron, Brice, Amine) for their scientific help during the research program.

My sincere thanks also go to all friends Ilyes, Guillaume , Okba, Nadir, Lahcene, Aymen, Thomas, Zahira, Bery, Géraud, Zakaria for their supports and good humour.

I would like to thank everybody who was important to the successful realization of thesis, as well as expressing my apology that I could not mention personally one by one.

Finally, I would express a deep sense of gratitude to my wife, who has always stood by me like a pillar in times of need and to whom I owe my life for her constant love, encouragement, moral support and blessings.

Ahcène Boukorça

Dedication

To my parents MOHAMED and FATIMA,

my wife SOUAD,

my children MOHAMED, WASSIM and WAEL,

my sisters and brothers,

my all those who are dear to me.

Contents

Introduction

I. Context

The success stories of the database technology keep companies continuously demanding more and more efficient services (such as data storage, query processing/optimization, security, transaction processing, recovery management, etc.) to deal with dimensions that Big Data brought. They include: large volumes of high velocity, complex and variable data that require advanced techniques and technologies to enable the capture, storage, distribution and management analysis of the information. These dimensions have been rapidly advocated by the database community, where several thematic conferences and products have been launched. We would like to draw attention to another dimension, which in our opinion, does not get the *same buzz* as the traditional dimensions. It concerns the sharing in the world of database. The sharing may concern several database entities: data, resource, and queries.

From data perspective nowadays, large amounts of data are continuously generated in many domains like *Life Science Research, Natural Resource Exploration, Natural Catastrophe Prevention, Traffic Flow Optimization, Social Networks, Business Competition*, etc. These data need to be *collected*, *stored* and *analyzed* in order to be well exploited by organization's managers and researchers to perform day-to-day company's tasks such as: (i) decisions making, (ii) generation of added-value to be more competitive and (iii) sharing the whole or fragments of their data stored in storage systems with other users and communities. Recently, large and small companies and organizations raise demands for more data sharing. Usually these data are stored in $\mathcal{DBMS}$*s*. For instance, the Competition and Markets Authority (CMA) has ordered Britain's high street banks to adopt *opening banking principles* to increase data-sharing between financial service organisations[1]. In other disciplines such as behavioral Science, the

[1] http://www.experian.co.uk/blogs/latest-thinking/a-new-era-of-data-sharing-how-the-cma-is-shaking-up-retail-banking/

data sharing contributes in facilitating the *reproduction* of computational research [88]. The need of data sharing pushes several countries (such as Australia, Ireland, etc.) to legislate the usage sharing policy (for privacy issues) in several sectors including government sector[2].

From computation resources and storage systems perspective, database storage systems are evolving towards decentralized commodity clusters that can scale in terms of capacity, processing power, and network throughput. The efforts that have been deployed to design such systems *share* simultaneously physical resources and data between applications [106]. Cloud computing largely contributed in augmenting sharing capabilities of these systems thanks to their nice characteristics: *elasticity*, *flexibility*, *on-demand storage* and *computing services*.

Queries which represent one of the most important entities of the database technology are also concerned by the sharing phenomenon. The query sharing got a particular interest in 80's, where Timos Sellis identified a new phenomenon known by **query interaction** and gave rise to a new problem called Multi-Query Optimization (MQO) [223]. Solving this problem aims at optimizing the global performance of a query workload, by augmenting the reuse of intermediate query results. One of the main characteristics of this research problem is that it has been tackled throughout all query languages associated to all database generations: traditional databases (*RDB-SQL*) [223], object oriented databases (*OODB - OQL*) [277], semantic databases (*SDB - Sparql*) [166, 110], distributed databases [153], stream database (*STDB - CQL*) [96], data warehouses (*SQL OLAP*) [273], etc. as shown in Figure 1.1. The advent of *MapReduce* for large-scale data analysis brings the notion of job sharing. Hence, MQO has proved to be a serious candidate to exploit the common jobs and then factorize them [190]. The main idea of this work has been borrowed from multi-query optimizations developed in traditional databases.

To integrate the actual reality, the query sharing has to consider a new dimension representing the *volume of queries* that storage systems have to deal with. This volume is motivated by the explosion of E-commerce Web sites, social media, etc. For instance, in *Amazon*, 100 million queries for various objects can arrive at its databases. The same fact is identified in *Alibaba* the giant Chinese E-commerce site. These queries are complex (require heavy operations such as joins and aggregations [30]) and repetitive. Another aspect that contributes in increasing the number of queries that a particular $\mathcal{DBMS}$ has to handle is the recommendation and the exploration of queries [99]. Faced to this situation, the traditional query sharing has to be revisited to integrate the volume of queries.

Based on this discussion, we *claim that any data storage system has to deal with three joint dimensions: data volume, query volume (we call it big queries) and query sharing issues.* The data volume impacts seriously the performance of queries. Having efficient methods to accelerate data processing becomes more important and urgent than ever, and hence they receive

[2]`http://www.legislation.nsw.gov.au/acts/2015-60.pdf`

special attention from academic and industrial researchers. This is well illustrated by the huge number of proposals, aiming at optimizing queries at the logical level [227, 167, 236, 168, 210, 266, 85, 224], the physical level [279, 79, 14, 33, 73, 81, 154] and the deployment level [36]. The query sharing has largely contributed in optimizing queries [110, 273] and physical design [191, 7]). These efforts ignore the volume of queries.

Thus, *we would like to issue a think tank about sharing among large amount of queries*. This think tank is discussed in the context of relational data warehouses usually modeled by a **star** schema or its variants (ex. snowflake schemas) that provide a query centric view of the data. Another important characteristic of this schema is that it increases the interaction among OLAP (Online Analytical Processing) queries, since they perform joins involving the central fact table(s) and peripheral tables, called dimensions.

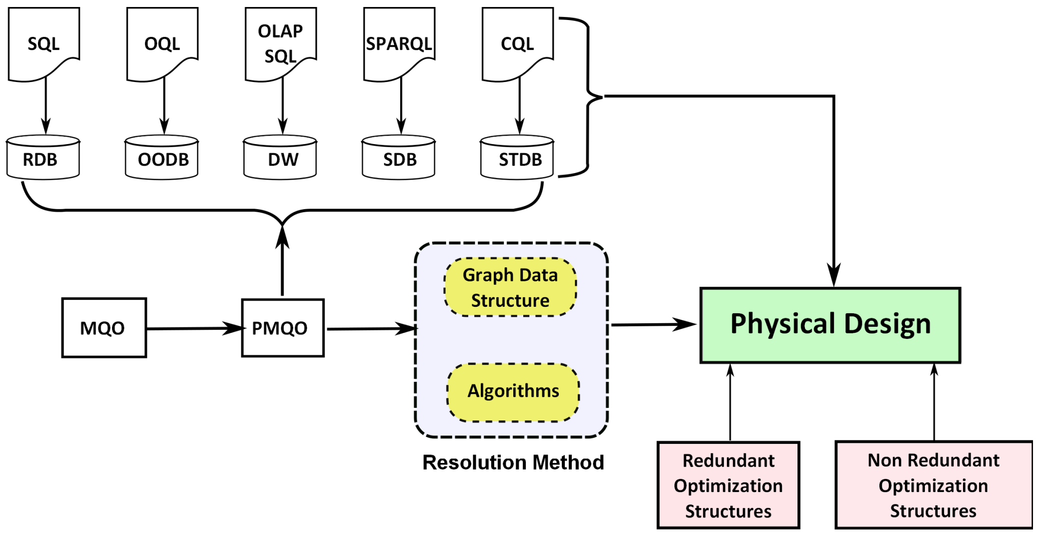


Fig. 1.1 – MQO in database generations

As we said before, **data and queries** are related to the advanced deployment platforms such as parallel, database clusters, etc. [36]. Designing a database in a such platform requires sensitive steps/phases, such as: (1) partition data into many fragments, (2) allocate the fragments in their corresponding processing nodes, and (3) give a query processing strategy over processing nodes, to ensure load balancing of queries. Data partitioning, data allocation and query load balancing problems have, as main inputs: a workload of queries, the characteristics of processing nodes and the database schema.

To deal with the volume of interacted queries, the development of advanced data structures that capture their sharing is widely recommended in defining intelligent algorithms. The graph theory and its data structures largely contribute in solving complex problems involving large scale of search space, such as data partitioning for centralized databases, where traditional graph representations have been used by several researchers: (i) Navathe et al. [186, 184] to define vertical and horizontal data partitioning algorithms for relational databases and (ii) Bellatreche et al. [29] for horizontal partitioning in the context of object oriented databases. Recently, Curino et al. [81] propose a generalization of the graph concept: hypergraphs, to partition and replicate the instances of a particular database in the case of distributed OLTP (Online Transaction Processing), where the processing of distributed transactions is expensive due to consensus protocol, distributed locks and communication cost. In this context, the nodes and the edges of a such graph represent respectively, instances of the database and transactions, where each edge is weighted by a number of transactions. A such the graph can be very huge, since its nodes are associated to the number of tuples of a database. To manipulate this very big graph, the authors are motivated to use hypergraph-partitioning algorithms that can ensure the scalability. In which, Hypergraph partitioning is well studied by the Electronic Design Automation (EDA) to test electronic circuit, besides, a large panoply of partitioning libraries exist such as hMETIS and Zoltan-PHG.

This motivates us to explore hypergraphs in dealing with large volume of interacted queries (since they were used to dealing with large number of instances). As a consequence, we approached, *Mentors Graphics company* located in Grenoble - France, a leader in EDA software to get their expertise in hypergraph usage and elaborate an analogy between our problem and their finding. This collaboration was fruitful, since after several months of collaboration, we succeed to find an analogy between an electronic circuit and an unified query plan, by representing it as a hypergraph. We also adjust the using of their tools to be exploited for our problem. Once this hypergraph is generated, we consider it in logical and physical optimizations and the deployment phase of the life cycle of a data warehouse design. This rich experience will be wholly discussed in this thesis.

II. Thesis objectives and contributions

Due to the large spectrum of topics that we studied in this thesis, covering: logical optimizations of queries; physical optimizations; data warehousing; optimization structures selection; multi-query optimization problem, cost models, deployment phase, hypergraphs, etc. It is necessary to present a synthetic survey on (i) the data warehouse technology, its life cycle that includes several phases (requirement collection, conceptual phase, logical phase, ETL (Extract, Transform, Load), deployment phase, physical phase and exploitation phase), (ii) logical and physical query optimizations, knowing that the former have been largely studied in the context of the traditional databases, whereas the latter have been amplified in the context of the data warehouses, and (iii) the deployment phase on a parallel database architecture.
Since we deal with the phenomenon of big queries and their interactions, we have to show the importance of a scalable data structure that easily captures this interaction, and can be used as a support to define intelligent algorithms with less computation and high quality for

selecting logical and physical optimizations. We have also to consider case studies to deploy our findings. To do so, we consider the problem of selecting two optimizations structures which are the materialized views ($\mathcal{MV}$) and the horizontal partitioning for centralized and parallel environment. We have also developed a tool to assist designers and DBA (Database Administrator) when dealing with the physical design involving the interaction among queries.

II.1. Contributions

This section highlights the contributions of the thesis. First, we discuss the typical design of data warehouse to provide a general understanding of the design choices and deployment architectures. We will focus on The logical and physical optimizations, because they are the most important tasks of query optimizers. Another point concerns the deployment phase of the life cycle of the data warehouse.

From these overviews, we discuss the origin of our data structure to represent the query interaction issued from our collaboration with Mentors Graphics company. We would like to share this experience and the efforts provided to find an analogy between our problem and electronic circuit.

Parallel to the definition of this structure, many other developments have been conducted to understand the algorithms and the tools used by VLSI (*Very-Large-Scale Integration*) community and adapt them to our context. Once, this adaptation is done, we consider two *workload-driven problems*: the selection of materialized views and the selection of horizontal partitioning. Furthermore, we discuss how we can improve the life cycle of the deployment phase by integrating our finding that concerns our data structure and its associated algorithms. Finally, we investigate the scalability of our algorithms and the quality of their solutions, by stressing them in terms of database size, size of queries, etc.

II.1.1. Survey of Logical and Physical Optimizations

We conduct an in-depth analysis of how the state-of-art database manages the interaction among queries and how it is used in logical, physical optimizations and the deployment phase of the life cycle of the data warehouse. Based on this survey, we give new classifications of the existing studies and techniques used in the context of query optimization, and a clarification of the process of selecting optimization structures during the physical design phase, considered as one of the major phase. This material was published in the following paper:

[47] *Ahcène Boukorca, Ladjel Bellatreche, Sid-Ahmed Benali Senouci, Zoé Faget: Coupling Materialized View Selection to Multi Query Optimization: HyperGraph Approach. International Journal of Data Warehousing and Mining (IJDWM), 11(2): 62-84 (2015).*

II.1.2. HyperGraphs as a Data Structure

We present an analogy between electronic circuit and unified query graph, that is usually generated by merging individual query trees. Note that the leaves of a tree are operands — either variables standing for relations or particular, constant relations. Interior nodes are algebraic operators, applied to their child or children. On the other hand, a logical circuit is composed of gates (or standard cells) that perform logical operations and connected by metal wires [196]. The same electrical signal may propagate from one gate to several other gates – such a connection is called a net, and can be conveniently represented by a hyperedge. The hypergraph corresponding to a logic circuit directly maps gates to vertices and nets to hyperedges. Figure 1.2 shows the usage of hypergraphs in representing logical circuit. The processors in today's PCs contain tens of millions of microscopic transistors. This scalability is usually ensured by graph partitioning techniques [196]. We can cite for instance PaToH (Partitioning Tool for Hypergraphs) [66] and hMETIS - Hypergraph & Circuit Partitioning[3]. hMETIS is a set of programs for partitioning hypergraphs such as those corresponding to VLSI circuits.
Making the analogy between our problem and electronic circuit allows us borrowing techniques and tools of the EDA community and adapting them to the generation of an unified query plan from numerous queries. Our finding is then exploited to solve the problem of MQO and to evaluate its efficiency and quality against the most popular study of the state-of-art. This material was published in the following paper:

[50] Ahcène Boukorca, Ladjel Bellatreche, Sid-Ahmed Benali Senouci, Zoé Faget: SONIC: Scalable Multi-query OptimizatioN through Integrated Circuits. 24th International Conference on Database and Expert Systems Applications (DEXA), pp. 278-292, Prague, Czech Republic, August 26-29, 2013.

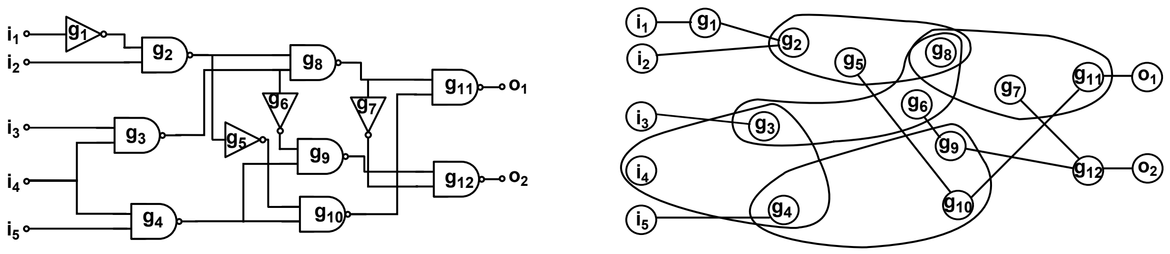


***Fig. 1.2** – From Logical Circuit to a HyperGraph*

II.1.3. What-if Unified Query Plan Generation

The query interaction represented by a unified query plan (UQP) is lying at the intersection of two worlds: the world of multi-query optimization and the world of physical design. An

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

optimal unified query plan is not necessary good for all instances of the physical design problem [273, 274]. Thus, it is needed to choose a good plan as input in order to produce a good set of an optimization structure candidates. Moreover, a good UQP for an optimization structure ($\mathcal{OS}$), is not necessary good for another structure. Choosing the best a UQP requires enumerating all possible plans, what is impossible because of the huge number of possible plans in the case of big workloads (*big-queries*). On the other side, the size of candidates proposed by any UQP in *big-queries* context, may be very big. As a consequence, finding the optimal optimization structure configuration becomes very difficult or even impossible (complexity of combinatorial algorithms).
To overcome these problems, we have proposed a new approach that incorporates the knowledge related to optimization structures when generating UQP, in order to minimize their enumeration cost. Our proposal captures query interaction in an UQP, by dividing the initial problems in several small disjoint sub-problems, which can be executed in parallel. Dividing multi-query optimization search space implies splitting the set of candidate elements in several small subsets, which minimizes the complexity of *combinatorial algorithms* to select the best configuration of an optimization structure. Our approach works around the *hypergraph*: a data structure (DS) used to model query interaction. This DS supports a big number of candidate elements and has already the adequate algorithms to partition the initial configuration into several sub-configurations with minimum loss of its characteristic (query interaction). The approach is inspired from the hypergraph partitioning and its contributions in testing logical circuits. More concretely, we propose a greedy approach that produces an adequate UQP for an $\mathcal{OS}$, called $\mathcal{OS}$-oriented UQP, by injecting optimization structure knowledge in the process of UQP generation.
This proposal was published in the following paper:

[52] A*hcène Boukorca, Zoé Faget, Ladjel Bellatreche: What-if Physical Design for Multiple Query Plan Generation. 25th International Conference on Database and Expert Systems Applications (DEXA), pp.492-506, Munich, Germany, September 1-4, 2014.*

To evaluate the efficiency and effectiveness of our approach, we consider two traditional optimization structures: materialized views and horizontal partitioning – both are based on a workload of queries, but use different constraints. For materialized views, we evaluate our proposal for static and dynamic cases, with taking account or not of the problem of query scheduling.
This material was published in the following paper:

[49] *Ahcène Boukorca, Ladjel Bellatreche, Alfredo Cuzzocrea: SLEMAS: An Approach for Selecting Materialized Views Under Query Scheduling Constraints. 20th International Conference on Management of Data (COMAD), pp. 66-73, Hyderabad, India, December 17-19, 2014.*

[212] *Amine Roukh, Ladjel Bellatreche, Ahcène Boukorca, Selma Bouarar: Eco-DMW: Eco-Design Methodology for Data warehouses. Proceedings of the ACM Eighteenth In-*

ternational Workshop on Data Warehousing and OLAP (DOLAP), pp. 1-10, Melbourne, VIC, Australia, October 19-23 2015.

II.1.4. Query Interaction in the Deployment Phase

Since queries are used by all phases of the deployment steps: data partitioning, data allocation and load balancing, we think that it could be interesting to push our reflection of incorporating our hypergraph structure in all query sensitive problems, in the deployment phase. The promising results obtained in a project in our lab, that aims at proposing a joint approach to deploy a data warehouse in a parallel architecture [36], motivate us to integrate the dimension of the interaction among queries to deploy a data warehouse in a parallel architecture, especially the horizontal data partitioning and data allocation.
This material was published in the following paper:

[48] *Ahcène Boukorca, Ladjel Bellatreche, Soumia Benkrid: HYPAD: Hyper-Graph-Driven Approach for Parallel Data Warehouse Design. 15th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP), pp. 770-783, Zhangjiajie, China, November 18-20, 2015.*

- [141] *Dhouha Jemal, Rim Faiz, Ahcène Boukorca, Ladjel Bellatreche: MapReduce-DBMS: An Integration Model for Big Data Management and Optimization. 26th International Conference on Database and Expert Systems Applications (DEXA), pp. 430-439, Valencia, Spain, September 1-4, 2015*

II.1.5. The development of Advisor Big-queries

Based on our findings, it will be opportune to develop an advisor inspired from the well-known tools developed by commercial editors and academicians to assist both designers and DBAs during their deployment activities and administration tasks when selecting optimization structures. As a consequence, we develop a tool called *Big-Queries* that combines two main functionalities: selecting optimization structures and deploying a $\mathcal{DW}$ in parallel machines. This interface allows DBA visualizing the state of his/her database that concerns three aspects: (i) tables of the target data warehouse (their descriptions, definition and domain of each attribute), (ii) the used workload (their SQL descriptions, access frequency of each query, selectivity factors of selection and join predicates), and (iii) resources required by physical design phase (size of the buffer, page size, the constraints related to our $\mathcal{OS}$), (iv) the interaction among queries visualized as hypergraphs, the different schemes related to our $\mathcal{OS}$ (materialized views and horizontal partitioning), the deployment schema with their respective costs (inputs/outputs, maintenance costs, storage costs). It is doted of gateway connection to several types of $\mathcal{DBMS}$ (Oracle11g, Oracle 12c, PostgreSQL).
Big-queries tool is available in the forge of our laboratory: `http://www.lias-lab.fr/forge/projects/bigqueries`.

III. Thesis Outline

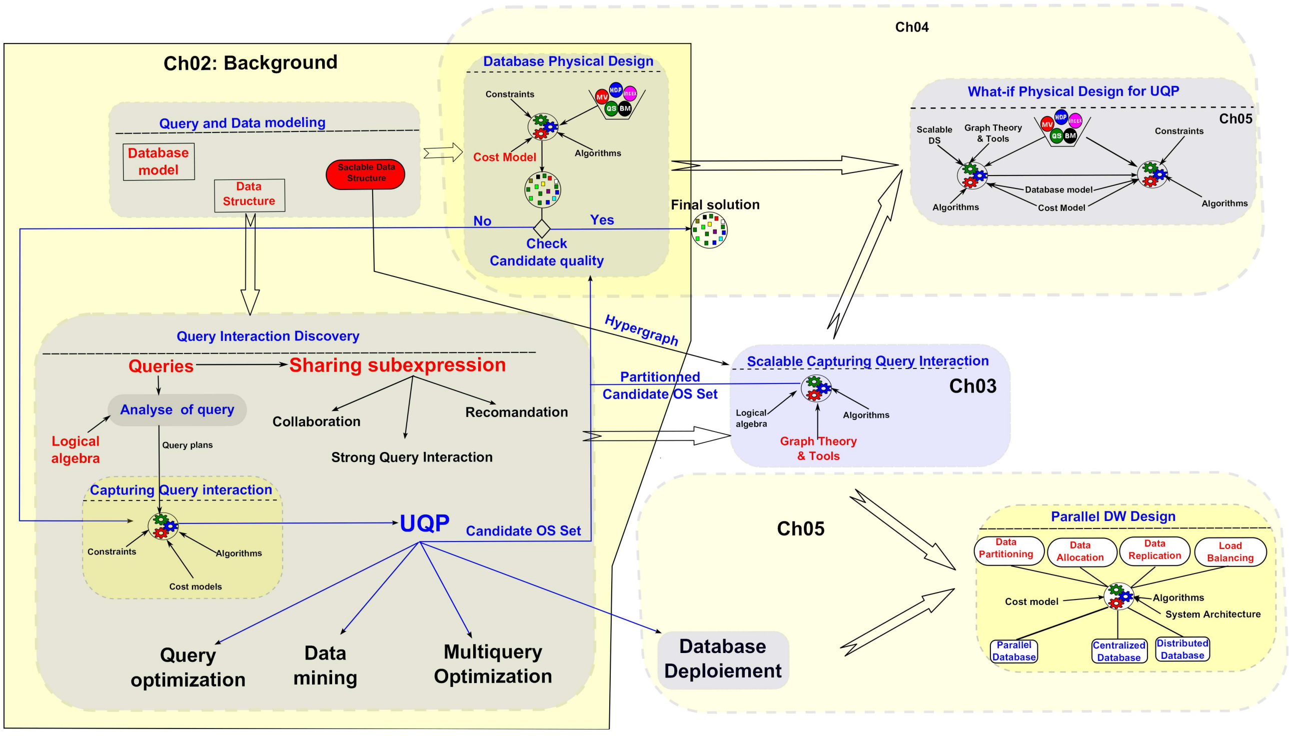


Fig. 1.3 – Repartition of thesis chapters

The remainder of the manuscript is organized as follows:

Chapter 2: provides the necessary background to understand our contributions. In this chapter, we start by zooming on data warehousing technology on which our experiments validation are done. Thereafter, we detail the parameters that affect response query processing time, materialized views, data partitioning and mulit-query optimization problems, and we give overviews of designing parallel data warehouse.

Chapter 3: details our main contribution that consists at using graph theory to overcome the of managing big-queries. The contribution aims at merging the MQO and physical design problems to generate target candidates for a specific optimization technique ($\mathcal{MV}$, index, and data partitioning). This is done by injecting $\mathcal{OS}$ knowledge in the generation of UQP.

Chapter 4: presents the application of our oriented-UQP approach in the selection of different $\mathcal{OS}$. First, the UQP is applied for $\mathcal{MV}$ selection issue with and without constraints, and exploited in dynamic materialization with query scheduling. Secondly, the UQP is used for horizontal data partitioning ($\mathcal{HDP}$).

Chapter 5: details our approach to design a parallel data warehouse taking into account query interaction that represented by unified query plan.

Chapter 6: gives a description of our tool *Big-Queries* used to validate our work.

Conclusion: concludes the thesis by providing a summary and an evaluation of the presented work. This chapter also discusses several opportunities for future work.

The main contributions of this thesis are presented in chapters 3, 4, 5 and 6. Figure 1.3 shows schematically the main axes of this work and their distribution in each chapter.

Part I

Backgrounds

Chapter 2

Background & State of Art

Contents

Abstract

This chapter is divided into two parts: **(i)** the **background** that presents different concepts and definitions related to the context of our study which represents the data warehouse. A special focus on its design life cycle is given. It includes the following phases: **(a)** elicitation of user requirements, **(b)** the conceptual modelling, **(c)** the logical modelling, **(d)** ETL (Extract, Transform, Load), **(e)** the deployment, **(f)** the physical modelling, and **(g)** the exploitation phase. This background aims at facilitating the presentation and the understanding the set of our contributions that *covers three chained phases: logical, deployment and physical.* **(ii)** Due to the complexity of these phases, we propose to present **state of art** related to logical and physical optimizations of queries running on centralized and parallel platforms. In this state of art, we also discuss *the role of interaction between queries* – a phenomenon well present in the context of relational data warehouses.

I. Introduction

Processing and optimization of queries in all database generations (traditional databases, object oriented databases, XML databases, data warehouses, graph databases, factorized databases, etc.) strongly depend on the proprieties of their: *logical models*, *query languages* and *physical models*, which are the *funnel* of all other phases of the database life cycle. This situation offers two main types of query optimizations: **(i)** logical optimizations and **(ii)** physical optimizations. As regards the first type of optimizations, when a query is issued by a user or an application, it is parsed to check whether the query is correctly specified, resolve any names and references, verify consistency, and perform authorization tests. When the query passes these tests, it is converted into an internal representation that can be easily processed by the subsequent phases. Then, the "query rewrite" module transforms the query into an equivalent form by carrying out a number of optimizations offered by the properties of the used query languages (e.g. algebraic properties the relational model) [205]. In the context of very large databases requiring complex queries involving joins and aggregations, the logical optimizations are *not enough* [72]. As a consequence, the physical optimizations (such as materialized views, indexes, partitioning, data compression, etc.), selected during the physical phase ($\mathcal{PYP}$), are necessary. These optimizations are crucial for performance of the final database/data warehouse, since they deal with decisions about selecting physical storage structures and access methods for data files. It should be noticed that the $\mathcal{PYP}$ exploits the properties of the *deployment platform* of the target databases/data warehouses.

The query processing and optimization have been one of the most active research topics in the Core Databases and Information Systems. By *scholar.googling "query processing databases"*, in 2016, we find *11 300 entries*. Queries can be executed either in an isolated way or in a joint manner. In this later, the exploitation of common intermediate results of the queries is usually considered when selecting optimization structures. The problems related to query optimization and physical design are both known as hard tasks. They integrate several parameters belonging to various phases of the life cycle of database/data warehouse design: conceptual, logical and deployment and constraints (e.g. storage, maintenance, energy). Consequently, their search spaces may be very large [222, 78, 201, 220, 68, 24]. To reduce the complexity of these problems, pruning of these search spaces is mandatory [162].
In early researches on databases, the majority of the studies were focused on logical optimizations. The importance of physical optimizations and especially physical design, were amplified as query optimizers became sophisticated to cope with complex decision support queries that *share intermediate results* [72]. To strengthen our finding, we consider the *multi-query optimization*, initially discussed by Professor *Timos Sellis* in 1988 [223], in the context of logical optimizations. It aims at exploiting common sub-expressions to reduce evaluation cost of queries. Modelling this problem has been exploited by the process of physical optimization structures such as *materialized views* [273]. This clearly shows the strong dependency between logical ($\mathcal{LOP}$) and physical ($\mathcal{POP}$) optimizations. This means that if a change in the $\mathcal{LOP}$

often results in a change in that of $\mathcal{POP}$. With the explosion of data volumes, the increasing needs of end users and decision makers in terms of exploring data [143] and recommending queries [9], and the diversity of deployment platforms motivate us to consider the dependency between logical and physical optimizations in the context of *big-queries*. In this chapter, we introduce the foundations on which this work is built, and position our contributions.
This chapter begins with an overview of the data warehouse technology, its architecture, its design life cycle (Section II). In Section III, we present factors that influence the logical and physical optimizations of OLAP queries. Section discusses the most popular data structures used in database context. Finally, Section V concludes the background material.

II. The Data Warehousing Technology

Traditional databases, called operational databases, are designed for building day-to-day applications. They aim at ensuring fast and concurrent accesses to data while guaranteeing consistency. They are based on the On-Line Transaction Processing (OLTP) paradigm. These databases are designed with high normalization degree using functional dependencies and normal forms [101]. Generally, the OLTP transactions access and return few records [101]. On the other hand, analytical requirements imposed by big companies such as *Wal-Mart* [267] need to aggregate a large volume of historical data from many tables (using joins). Unfortunately, operational databases fail to support analytical queries. This is because they were not designed to store the historical data and process them in efficient way. These limitations largely contributed to the birth of a new technology which is the data warehousing that supports On-Line Analytical Processing (OLAP).
A Data Warehouse ($\mathcal{DW}$) aims at supporting decision's making. It *collects* data from various *heterogeneous*, *autonomous*, evolving and *distributed* data sources, *transforms* and *cleans* them and finally loads them into new data structures designed to support OLAP queries. These structures are represented by a *hypercube*, with *dimensions* corresponding to various business perspectives, and their *cells* contain the measures to be analyzed.
In the sequel, we give an overview related to a $\mathcal{DW}$. A brief description of the conventional architecture, query languages and designing steps of a $\mathcal{DW}$ are discussed. A large discussion about the logical and physical optimizations is also given.

II.1. Definitions

Definition 1. *$\mathcal{DW}$ is a repository of integrated data issued from different sources for data analytic purposes [158]. More technically, a $\mathcal{DW}$ is defined as a collection of subject-oriented, integrated, non-volatile, and time-varying data to support management decisions [256].*

Below, we explain the four key characteristics of a $\mathcal{DW}$ [256]:

- **Subject oriented:** means that the $\mathcal{DW}$ focuses on analytical needs which depend on the nature of activities performed by the companies (e.g. inventory management, product sales recommendation, etc.).
- **Integrated:** means that data are extracted from several organizational and external sources have to be integrated after the processes of cleaning, transformation usually performed through an Extraction, Transformation, and Loading Process (ETL).
- **Non-volatile:** means that durability of data is guaranteed by disallowing data modification and removal, thus expanding the scope of the data to long period than operation systems usually offer.
- **Time varying:** indicates the possibility of keeping, for the same information, different values related to its changes (evolution).

Definition 2. ***On-Line Analytical Processing (OLAP)*** *deals with analytical queries that handle a heavy load of data which involves aggregation of values captured by scanning all records in a database [59].*

Definition 3. ***Multidimensional model*** *is a data representation in an n-dimensional space [117]. Usually called* **data cube** *or* **hypercube**. *It is used as a data structure to save data and to facilitate the optimization of OLAP queries.*

A multidimensional model is composed of a set of *dimensions* and set of *facts*. A dimension is composed of either one level or one or more *hierarchies*. A level is analogous to an entity type in Entity-Relationship model [75]. A hierarchy comprises several related levels (e.g. *year* $\longrightarrow$ *semester* $\longrightarrow$ *month* $\longrightarrow$ *day*). A dimension may contain several hierarchies. Levels in a hierarchy permit analyzing data at various granularities or level of details. These levels impact the query processing and optimization (e.g., the case of bitmap join indexes [32]). Figure 2.1 shows an example of a cube representing sales activities. It has three dimensions: *Product*, *Time*, and *Customer*.

Definition 4. ***Data mart:*** *is a specialized data warehouse oriented for a specific business line or a team (department) [114].*

II.2. Data warehouse architecture

In the literature, the general architecture of a $\mathcal{DW}$ is composed of five tiers [158] as shown in Figure 2.2.

1. **Data sources tier:** represents the operational activities that are saved in operational databases, XML files, etc. Data sources can be internal (produced inside the companies) or external (from the Web).

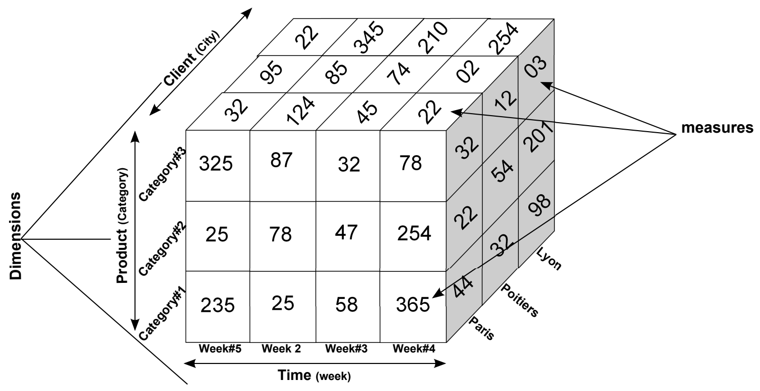


Fig. 2.1 – An example of multidimensional model representation

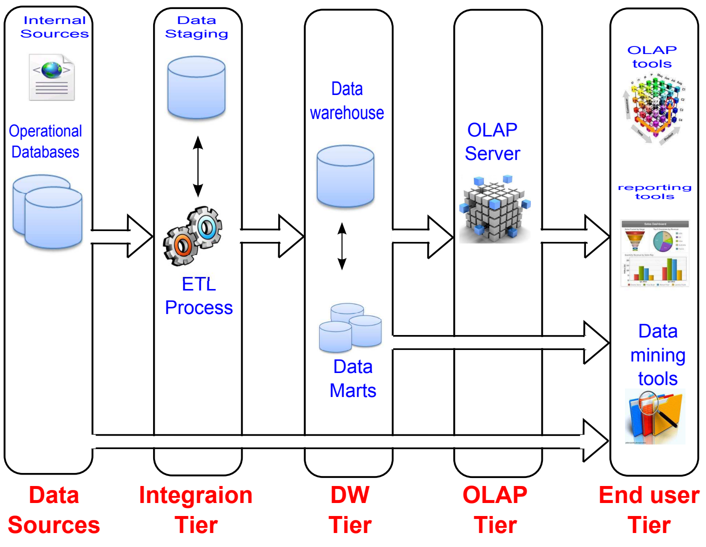


Fig. 2.2 – Data warehouse architecture

2. **The Integration tier:** is composed of ETL tools (Extract, Transform, Load) involved to feed $\mathcal{DW}$ by data from sources (see. Section II.3). The ETL process needs an intermediate temporary database called *data staging area* [258], which is used to temporarily save extracted data to facilitate the integration and transformation processes before their loading into the $\mathcal{DW}$.

3. **The data warehouses tier:** represents the data repository dedicated to store a $\mathcal{DW}$ or several data marts. The $\mathcal{DW}$s and data marts are described by *Meta-data* that defines the semantic of data and organizational rules, policies, and constraints related to data items [114].

4. **The OLAP tier:** is composed of OLAP server usually used for business use of multidi-

mensional data representation.

5. **The end user tier:** contains tools allowing users to exploit the $\mathcal{DW}$ contents. It includes: (1) *tools* to execute OLAP queries. (2) *reporting and statistical tools* to provide dashboards following the decision requirements, and (3) *data mining tools* to discover some valuable knowledge from data currently stored in the $\mathcal{DW}$. To load data, the end user tools can use the OLAP server, $\mathcal{DW}$, or data marts.

II.3. Data warehouse design methods

In this section, we present different phases of the life-cycle of its design inherited from traditional databases. Usually, it includes the following phases: data source analysis, elicitation of requirements, conceptual, logical, deployment, and physical design. ETL is appended as a design phase responsible for analytical processing [156]. Phases depicted in Figure 2.3 have to be follow when designing a $\mathcal{DW}$.

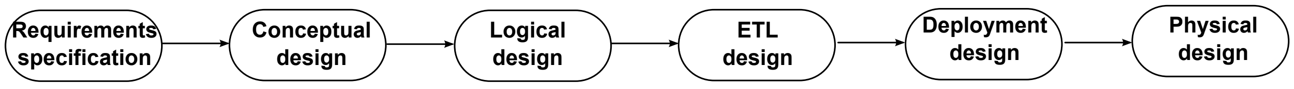


Fig. 2.3 – Phases in data warehouse design

Data Sources. As we said before a $\mathcal{DW}$ is built from *heterogeneous* data sources, with *different data representations*, and *provenance*. The heterogeneity poses serious problems when designing the $\mathcal{DW}$. This is due to the presence of conflicts that may exist among sources. Goh et al. [111] suggest the following taxonomy of conflicts: *naming conflicts*, *scaling conflicts*, *confounding conflicts* and *representation conflicts*. These conflicts may be encountered at *schema level* and *at data level*.

- *Naming conflicts:* occur when naming schemes of concepts differ significantly. The most frequently case is the presence of *synonyms* and *homonyms*. For instance, the status of a person means her familial status or her employment status.

- *Scaling conflicts*: occur when different reference systems are used to measure a value (for example *price* of a product can be given in Dollar or in Euro).

- *Confounding conflicts*: occur when concepts seem to have the same meaning, but differ in reality due to different measuring contexts. For example, the weight of a person depends on the date where it was measured. Among properties describing a data source, we can distinguish two types of properties: *context dependent properties* (e.g. the weight of a person) and *context non-dependent properties* (gender of a person).

- *Representation conflicts* : arise when two source schemas describe the same concept in different ways. For example, in one source, student's name is represented by two elements *FirstName and LastName* and in another one it is represented by only one element *Name*.

The data sources may be grouped into five main categories [206]: (i) *production data*, (ii) *internal data*, (iii) *archived data*, (iv) *external data* and (v) *experimental data*. The production data come from the various operational systems of the enterprise. The internal data include the data stored in spreadsheets, documents, customer profiles, databases which are not connected to the operational systems. External data such as data produced by external agencies, weather forecast services, social network, and recently knowledge bases such as Yago [133], etc. play an crucial role in $\mathcal{DW}$ design of adding new values to the warehouses [39]. Recently, the computational science community such as physics, aeronautic, etc. is building warehouses from the experiment results. We can cite for instance, the example of project AiiDA[1].
The data of sources range from traditional ones, to semantic data, passing by graph databases [132, 102]).

Data Warehouse Requirements. As any product, the development of a $\mathcal{DW}$ application is based on functional and non-functional requirements [114]. Note that functional requirements describe the functionalities, the functioning, and the usage of the $\mathcal{DW}$ applications to satisfy the goals and expectations of decision makers. These requirements are known as *business requirements* [55] that represent high-level objectives of the organization for the $\mathcal{DW}$ application. They identify the primary benefits that the $\mathcal{DW}$ technology brings to the organization and its users. They express business opportunities, business objectives and describe the typical users and organizations requirements and their added-values. Other functional requirements are associated to users of the $\mathcal{DW}$ (*called user requirements*) describe the tasks that the users must be able to accomplish with thanks to the $\mathcal{DW}$ application. User requirements must be collected from people who will actually use and work with this technology. Therefore, these users can describe both the tasks they need to perform with the $\mathcal{DW}$. These requirements are modelled using several formalisms such as: UML use cases, scenario descriptions and goals [46].

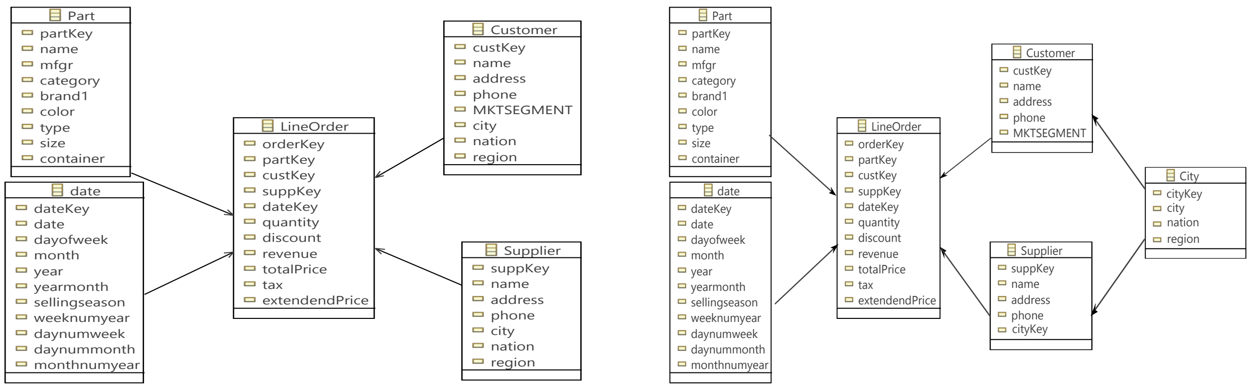
Non-functional requirements, called quality attributes are either optional requirements or needs/constraints [164], they are detailed in system architecture. They describe how the system will do the following objectives: the security, the performance (e.g. response time, refresh time, processing time, data import/export, load time), the capacity (bandwidth transactions per hour, memory storage), the availability, the data integrity, the scalability, the energy, etc. This type of requirements has to be validated over the majority of the phases of the $\mathcal{DW}$ life-cycle.

Conceptual Design. It aims at deriving an implementation-independent and expressive conceptual schema according to the conceptual model. As said in [256] that database community

[1]`http://www.aiida.net/`

agreed for several decades that conceptual models allow better communication between designers in terms of understanding application requirements. However, there is *no well-established* conceptual model for multidimensional data. Several formalisms mainly borrowed from traditional databases exist to design the conceptual model of a $\mathcal{DW}$: *E/R* model [112, 218], object-oriented model [3, 252, 173], and non-graphical representation [200]. The quality of a $\mathcal{DW}$ conceptual model is evaluated using testing methods [247].

Logical Design. There are three main approaches to represent logical model of a $\mathcal{DW}$, depending on how the data cube is stored: *(i) the relational ROLAP* (ROLAP), which stores data cube in relational databases and uses an extension of SQL language to process these data. Two main schemes are offered by ROLAP: the star schema and snowflake schema. Figures 2.4a and 2.4b show, respectively, an example of $\mathcal{DW}$ star and snowflake schemes of *SSB* benchmark [193]. A star schema consists of a one or several large fact table (s) connected to multiple dimension tables via foreign keys. Dimension tables are relatively small compared to the fact table and are rarely updated. They are typically non normalized so that the number of needed join operations is reduced. To avoid redundancy, the dimension tables of a star schema can be normalized. There is a debate on the benefits of having such normalized dimension tables, since it will, in general, slow down query processing, but in some cases it provides a necessary logical separation of data such as in the case of the demographic information [169]. *(ii) multi-*



(a) Star schema **(b)** Snowflake schema

Fig. 2.4 *– Examples SSB schemes*

dimensional OLAP (MOLAP) stores cube data in multidimensional array format. The OLAP operations are easily implemented on this structure. For high dimensionality of data, ROLAP solutions are recommended [256]. (iii) *Hybrid OLAP (HOLAP)* combines both approaches. It gets benefit from the storage capacity of ROLAP and the processing power of MOLAP.

ETL design. ETL processes are responsible for extracting and transforming data from heterogeneous business information sources to finally loading them into the target warehouse. This phase plays a crucial role in the process of the $\mathcal{DW}$ design. The quality of the warehouse strongly depends on ETL (the garbage in garbage out principle) [2]. The different steps of ETL need to understand different schemes (conceptual, logical and physical) of data sources [232, 31, 259, 262, 251, 2, 268, 254]. Several commercial and academic tools of ETL are also available such as: Oracle Warehouse Builder (OWB), SAP Data Services, Talend Studio for Data Integration, Pentaho Data Integration, etc.[2].

The first studies on ETL were concentrated on the physical models of data sources that includes the deployment platform (centralized, parallel, etc.) and the storage models used by the physical models (e.g. tables, files). In [254], a set of algorithms was proposed to optimize the physical ETL design. Alkis et al. [232] propose algorithms for optimizing the efficiency and performance of ETL process. Other non-functional requirements such as freshness, recoverability, and reliability have been also considered [233]. The work of [183] proposes an automated data generation algorithm assuming the existing physical models for ETL to deal with the problem of data growing.

Other initiatives attempted to move backward ETL from physical models to logical models. [262] proposed an ETL work-flow modelled as a graph, where nodes of the graph are activities, record-sets, attributes, and the edges are the relationship between nodes defining ETL transformations. It should be noticed that graphs contributing in modelling ETL activities. In [259], a formal logical ETL model is given using Logical Data Language (LDL) [229] as a formal language for expressing the operational semantics of ETL activities. Such works assume that ETL workflow knows the logical models of data sources and the elements of their schemes (relations, attributes).

Another move backward to conceptual models has been also identified, where approaches based on ad-hoc formalisms [260], standard languages using UML [251], model driven architecture (MDA) [140], Business Process Model and Notation(BPMN) [268, 8] and mapping modelling [62, 174]. However, these approaches are based on semi-formal formalisms and do not allow the representation of semantic operations. They are only concentrated on the graphical design of ETL processes without specifying the operations and transformations needed to overcome the arising structural and semantic conflicts. Some works use ontologies as external resources to facilitate and automate the conceptual design of ETL process. [234] automated the ETL process by constructing an OWL ontology linking schemes of semi-structured and structured (relational) sources to a target data warehouse ($\mathcal{DW}$) schema. Other studies have been concentrated on semantic models of data sources such as the work of [187] that considers data source provided by the semantic Web and annotated by OWL ontologies. However, the ETL process

[2]`https://www.etltool.com/list-of-etl-tools/`

in this work is dependent on the storage model used for instances which is the triples. Note that after each research progress on ETL, its operations have to be specified and rewritten. This situation motivates researchers to make ETL more generic. In [262], the authors propose a generic model of ETL activities that plays the role of a pivot model, where it can define 10 generic ETL operator [235]. The signature of each operator is personalized to the context where target data source contains integrated record-sets related to attributes extracted from sources satisfying constraints:

1. $Retrieve(S, A, R)$: retrieves record-sets R related to attributes A from Source S;
2. $Extract(S, A, R, CS)$: enables selection and extraction of data from source S satisfying constraint CS;
3. $Merge(S, A_1, A_2, R_1, R_2)$: merges record-sets R_1 and R_2 belonging to the same source S;
4. $Union(S_1, S_2, A_1, A_2, R_1, R_2)$: unifies record-sets R_1 and R_2 belonging to different sources S_1 and S_2 respectively;
5. $Filter(S, A, R, CS')$: filters incoming record-sets R, allowing only records with values satisfying constraints CS';
6. $Join(S, A_1, A_2, R_1, R_2)$: joins record-sets R_1 and R_2 having common attributes;
7. $Convert(S, A, R, F_S, F_T)$: converts incoming record-sets R from the format F_S of source S to the format of the target data source F_T;
8. $Aggregate(S, A, R, F)$: aggregates incoming record-set R applying the aggregation function F (count, sum, avg, max) defined in the target source.
9. $DD(R)$: detects and deletes duplicate values on the incoming record-sets R;
10. $Store(T, A, R)$: loads record-sets R related to attributes A in target data source T,

Building the ETL process is potentially one of the biggest tasks of building a warehouse; it is complex, time consuming, and takes the lion's share of design and implementation efforts of any warehouse project. This is because it requires several tasks and competencies in terms of modelling, work-flows, and implementations. Non-functional requirements such as quality and the performance are taken into account when designing ETL [261, 231, 53]. In this thesis, we assume that ETL phase is already performed.

Deployment Phase. This phase consists in choosing the adequate platform in which the target warehouse will be deployed. Several platforms can candidates to store the warehouse: centralized, database clusters, parallel machines, Cloud, etc. The choice of the deployment platform depends on the company budget and the fixed non-functional requirements [36].

Physical design. It is a crucial phase of the $\mathcal{DW}$ life cycle. Note that the majority of non-functional requirements are evaluated during this phase. It uses the inputs of deployment and the logical phases. In this design, optimization structures such as materialized views, indexes, data partitioning, etc. are selected to optimize one or several non-functional requirements such as query performance and energy.
Since, we concentrate on processing and optimizing big queries that take into account the logical, deployment and physical phases, a special focus on deployment and physical phases will be given in next sections.

II.4. Summary

Based on the above discussion, we figure out the presence of diversity that concerns all phases of the life cycle that designers have to consider when designing a $\mathcal{DW}$ application. If we project this diversity on our contributions (related to processing and optimizing big queries), we have identified *four main dimensions* that have to be taken into account when offering adequate optimizations of these queries:

1. the chosen logical model ($\mathcal{LM}$) of the $\mathcal{DW}$ application identified during the logical phase of the life cycle;
2. the used query language ($\mathcal{QL}$) offered by the $\mathcal{DBMS}$;
3. the used deployment platform ($\mathcal{DP}$) including the target $\mathcal{DBMS}$ and hardware;
4. the available logical and physical optimizations ($\mathcal{PO}$) offered by the $\mathcal{DBMS}$ that have to be exploited by DBA either in isolated or joint manners to ensure the performance of her/his workload.

Once these dimensions are chosen, they give the signature of a warehouse application. The particularity of these dimensions is that the first three are *generally supposed as frozen*, and the last one may vary according the DBA expertise and the query optimization requirements.

Example 1. *To illustrate these four dimensions that we considered as the core of our think tank, we give an example of tested results obtained by the Transaction Processing Council (TPC). http://www.tpc.org/tpch/results/tpch_perf_results.asp presents the TPC-H - Top Ten Performance Results. We focus on the 100,000 GB Results (last table of this Web Site). It presents a TPC-H schema (variant of the star schema) (corresponding to our dimension 1) with its queries (corresponding to our dimension 2), executed on EXASOL EXASolution 5.0 database and Dell PowerEdge R720xd using EXASolution 5.0 (dimension 3). The dimension 4 is related to different optimizations offered by the used database.*

In this context, our work consists in getting benefit from the characteristics of the frozen dimensions and exploiting them to optimize very large number of queries. More correctly, our problem of optimizing big queries may be formalized as follows: given:

- A *workload* $\mathcal{W}$ of queries, expressed in:
- a *query language* ($\mathcal{QL}$) related to:
- a *logical model* ($\mathcal{LM}$) translated to:
- a *physical model* ($\mathcal{PM}$) associated to:
- a set of *physical optimizations* ($\mathcal{PO}$) deployed in:
- a *platform* ($\mathcal{P}$).

The objective of our problem is to optimize the workload $\mathcal{W}$ by:
exploiting as much as possible the characteristics of each input.

To taking better advantages of these characteristics, we propose in next sections to review each dimension and their *interaction*. Figure 2.5 presents an UML model illustrating our proposal.

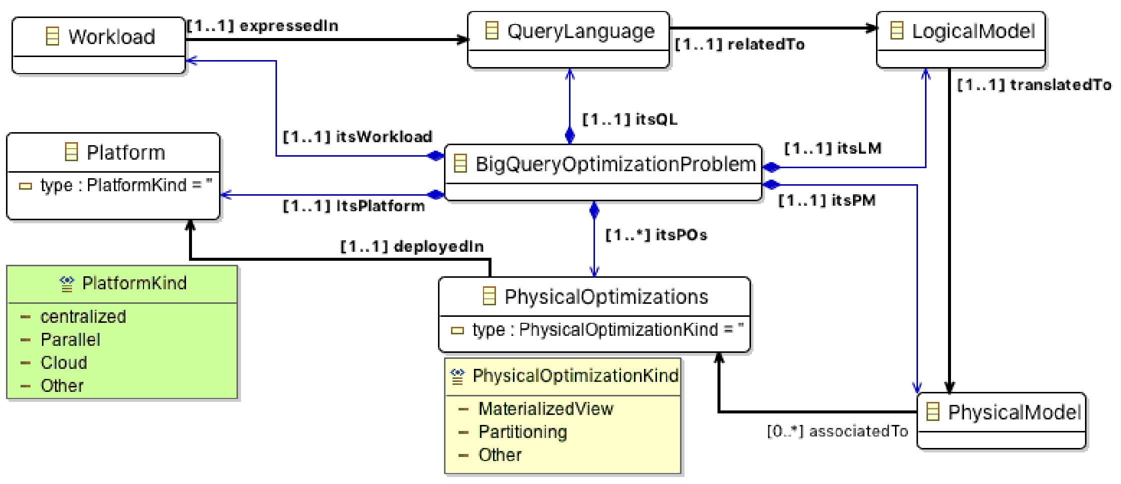


Fig. 2.5 *– UML Model of our Big-Queries Optimization Problem*

III. Factors Impacting Query Processing and Optimization

In this section, we first review each entry, its characteristics and its role in optimizing big queries.

III.1. Dimension 1: The Logical Model of our Target Warehouse

We have already reviewed the major logical schemes of $\mathcal{DW}$. In our study, we concentrate on a $\mathcal{DW}$ modelled by a relational schema such as the star schema or its variants.

III.2. Dimension 2: OLAP Queries

Operations in $\mathcal{DW}$ applications are mostly read ones and are dominated by large and complex queries. Two main classes of query languages exist in the context of $\mathcal{DW}$. They depend on the architecture of the target $\mathcal{DW}$: ROLAP (Relational OLAP) and MOLAP (Multidimensional OLAP). The ROLAP exploits the maturity and standardization of relational database solutions. The typical queries defined on the most popular logical schema which is the star schema are called *star join queries.* They have the following characteristics:

1. a multi-table join among a large fact table and dimension tables,
2. each one of the dimension tables involved in the join operation has multiple selection predicates on its descriptive attributes,
3. there is no join operation between dimension tables.

offers a fast response time for queries, but it suffers from modelling all user requirements. Naturally, ROLAP uses an extension of SQL languages that includes OLAP operations such as cube, roll-up, drill-down, etc. For MOLAP, MDX (multidimensional Expressions) language is mainly used. While SQL operates over tables, attributes, and tupes of the relational schemes (e.g. star schema), MDX works over data cubes, dimensions, hierarchies and members (at the instance level).
The syntax of a typical MDX query is as follows:

```
SELECT <axis specification>
FROM <cube>
[WHERE <slicer specification>]
```

In this thesis, we concentrate on a SQL language for ROLAP $\mathcal{DW}$. This language uses several operators with a rich set of properties on their compositions that impact the deployment and the physical design phases. These properties are described in the Appendix XII.
In the first generation of query processing, the queries were treated in isolation. In 1988, the concept of multi-query optimization has been introduced in [223].

Definition 5. ***Multiple query optimization (MQO)*** *(called also global query optimization) exploits the interaction among queries by the means of intermediate results of a given the workload [223].*

To understand this phenomenon of MQO, let us consider the following example.

Example 2. *Let Q_1 and Q_2 be two queries. Q_1 has joins between the base relations R_1,R_2 and R_3 with a selection on R_1 using the predicate $p_{(att1=val)}$. Q_2 has joins between base relations R_1,R_3 and R_4 and the same selection as Q_1. Their query trees are given in isolated way in Figures 2.7a and 2.7a. Figure 2.7c presents a merging of these two tree by exploiting a common node.*

This phenomenon has been largely exploited by logical and physical optimizations as we will show in next Sections.

III.3. Dimension 3: Logical and Physical Optimizations

As we said before, in this dimension, we distinguish two main types of optimizations: *logical* and *physical.*

III.3.1. Logical Optimizations

The logical optimizations are related to different operations that a query optimizer has to performed when generating the best query plan (for a given query) among numerous plans. This is done by the exploitation of the properties of relational algebra. Note that all plans are equivalent in terms of their final output, but vary in their costs. This cost quantifies a metric of a non functional requirement such as the amount of time that they need to run [137]. One of the main task of a query optimizer is to select the plan with minimum amount of time. *This requires smart algorithms associated with advanced data structures.*

More concretely, the query trees and plans are one the most important inputs of any query optimizer. The process of executing a given query passes through four main steps: **(i)** parsing, **(ii)** rewriting, **(iii)** planning/optimizing, and **(iv)** executing. The *parser* checks the query string for valid syntax using a set of grammar rules, then translates the query into an equivalent relational algebra expression. The output is the *query tree.* The *rewriter* processes and rewrites the tree using a set of rules. The *planner/optimizer* creates an optimal execution plan using a cost model that accounted for CPU costs as well Inputs Outputs (IO) costs. This type of optimization is called cost-based optimizations. Figure 2.6 summarizes the different steps of a query optimizer.

A cost model specifies the arithmetic formulas that are used to estimate the cost corresponding a given non functional requirement metric of execution plans (e.g., execution time, energy consuming, etc.). For every different join implementation, for every different index type access, and in general for every distinct kind of step that can be found in an execution plan, there is a formula that gives its costs [137]. In **Appendix I**, we detail our used cost models.

The discussion that we had above concerns queries executed in isolation way. If all queries are considered in a batch fashion, the problem of MQO arises. It can be defined as follows:

for given a set of queries Q=$\{Q_1, .., Q_n\}$ to be optimized, each query Q_i has a set of possible individual local plans P_i =$\{p_1, .., p_{k_i}\}$; MQO consists in finding a global execution plan obtained by merging individual plans such that the query processing cost of all queries is minimized.

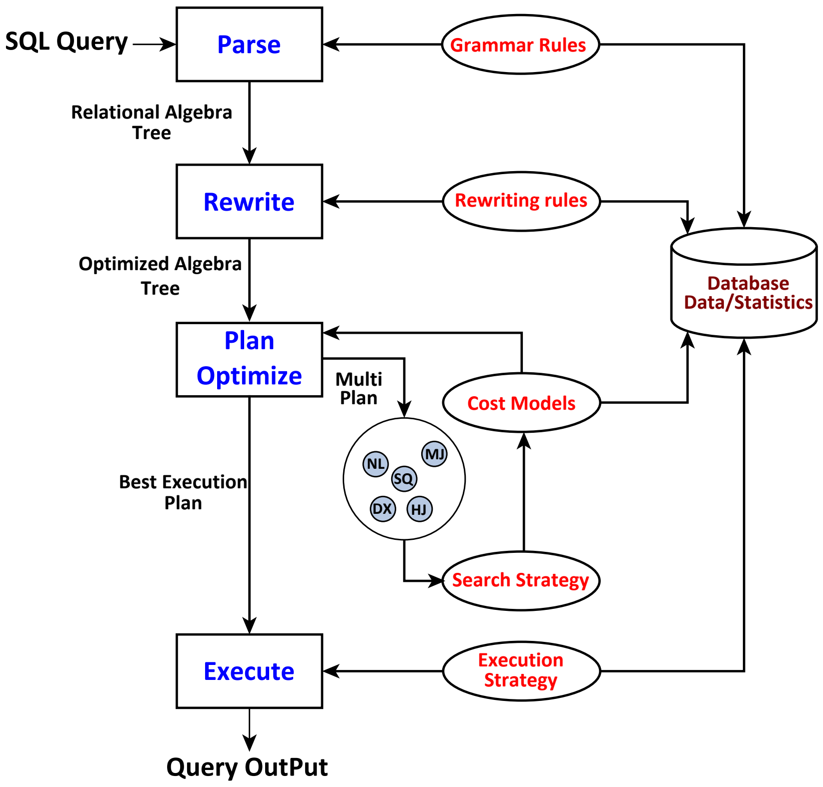
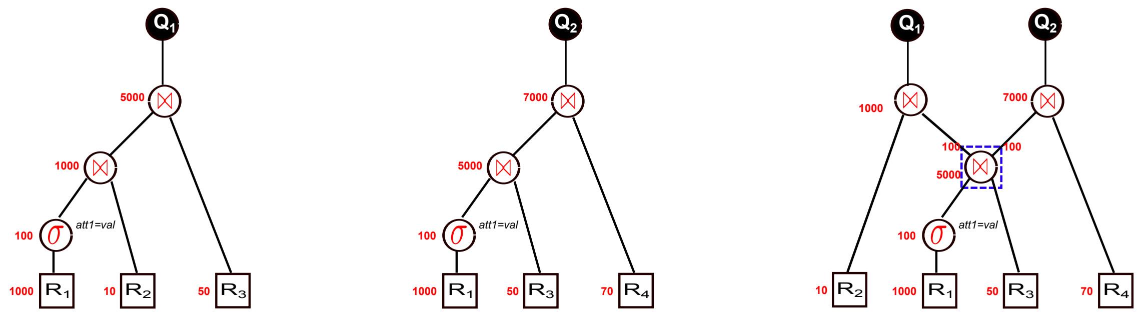


Fig. 2.6 *– Query optimizer steps*

Example 3. *To illustrate the process of selecting the best plan of queries in isolated and joint ways, let us consider the queries discussed in Example 3, where we suppose that the base relations* R_1*,* R_2*,* R_3 *and* R_4 *have respectively a scan cost of 1000, 10, 50 and 70 units and we assume the absence of any optimization structure.*

The query optimizer suggests individual best plans for each query and the $\mathcal{DBMS}$ *uses these plans to execute the queries independently. As shown in Figures 2.7a and 2.7b, the optimal plans of the queries* Q_1 *and* Q_2 *are* $(\sigma_{att_1=val}(R_1) \bowtie R_2) \bowtie R_3$ *and* $(\sigma_{att_1=val}(R_1) \bowtie R_3) \bowtie R_4$ *respectively. The total cost of the two queries is 20 380 units. However, we remark that the intermediate result* $(\sigma_{att_1=val}(R_1) \bowtie R_3)$ *can be shared by both queries. As shown in Figure 2.7c, the global plan has a total cost of 14430 units. So, by using MQO technique, we can easily reduce the execution cost by about 30%.*



(a) Individual query plan (Q_1)

(b) Individual query plan (Q_2)

(c) Multi query plan (Q_1&Q_2)

Fig. 2.7 *– Example of MQO benefit*

In the above example, the process of merging individual plans is straightforward. But, in a large workload, finding the best merging of different query plans is a hard problem. Since 80, it has been widely studied [74, 80, 105, 198, 222, 223, 228, 240, 280]. Sellis [223] proposed an A$*$ algorithm search algorithm to produce a global execution plan based on all possible local plans of each query. Alternatively, the local plan is constructed by either a dynamic programming algorithm [198] or a branch bound algorithm [116]. Subsequently, the local plans are merged into a single global plan such that the results of common sub-expressions can be shared as much as possible. These methods may take considerable memory resource and time to generate an optimized plan. Accordingly, many heuristic algorithms are proposed to improve significantly the generation time. They reduce the size of the search space for A$*$ algorithm, but they produce a sub-optimal global plan.

Logical optimizations have a great impact on physical optimizations, as we will show in the next sections.

III.3.2. Physical Optimizations

Physical optimizations have been amplified by the explosion of data and the necessity to optimize them [72]. These optimizations are ensured by the means of advanced structures such as materialized views, advanced indexing, data partitioning, etc. In this section, we describe the most used database optimization structures ($\mathcal{OS}$). A particular interest to the $\mathcal{OS}$ used by our study. We start by materialized views that is a redundant $\mathcal{OS}$ and is widely used to optimize OLAP queries in the context of the $\mathcal{DW}$. The second structure is horizontal data partitioning that is considered as a non-redundant $\mathcal{OS}$ that can be applied in several entities: a table, a materialized view and an index. Another particularity of the horizontal partitioning is that it can be used in centralized and distributed databases. A brief discussion on indexes is also presented, because they are usually associated to *data structures* (e.g., trees).

Materialized views

Definition 6. ***Virtual Views****: is a definition of a relation constructed logically from tables by an expression much like a query, bu it do not exist physically.*

Definition 7. ***Materialized views ($\mathcal{MV}$)*** *is a definition of a relation constructed periodically from tables by an expression much like a query and they are theirs tuples are stored in the storage devices.*

Materialized views are used to pre-compute either stored aggregated data or joins with/without aggregations. So, materialized views are suitable for queries with expensive joins or aggregations. Once materialized views are selected, all queries will be rewritten using materialized views (this process is known as *query rewriting*). A rewriting of a query Q using views is a query expression Q' referencing to these views. The query rewriting is done *transparently* by the query optimizer. To generate the best rewriting for a given query, a cost-based selection method is used [35]. Figure 2.8 illustrates this process.
Two major problems related to materialized views are: (a) the *view selection problem* and (b) the *view maintenance problem.*

Views selection problem. The database administrator cannot materialize all candidate views, as he/she is constrained by some resources like, disk space, computation time, maintenance overhead and cost required for query rewriting process [125]. Hence, he/she needs to pick an appropriate set of views to materialize under some resource constraints.

Formally, view selection problem ($\mathcal{VSP}$) is defined as follows: given:

1. a set of most frequently used queries $Q = \{Q_1, Q_2, ..., Q_n\}$, where each query Q_i has an access frequency f_i $(1 \leq i \leq n)$;

2. a set of constraints $\mathcal{C}$ (e.g., storage cost, maintenance cost, etc.);

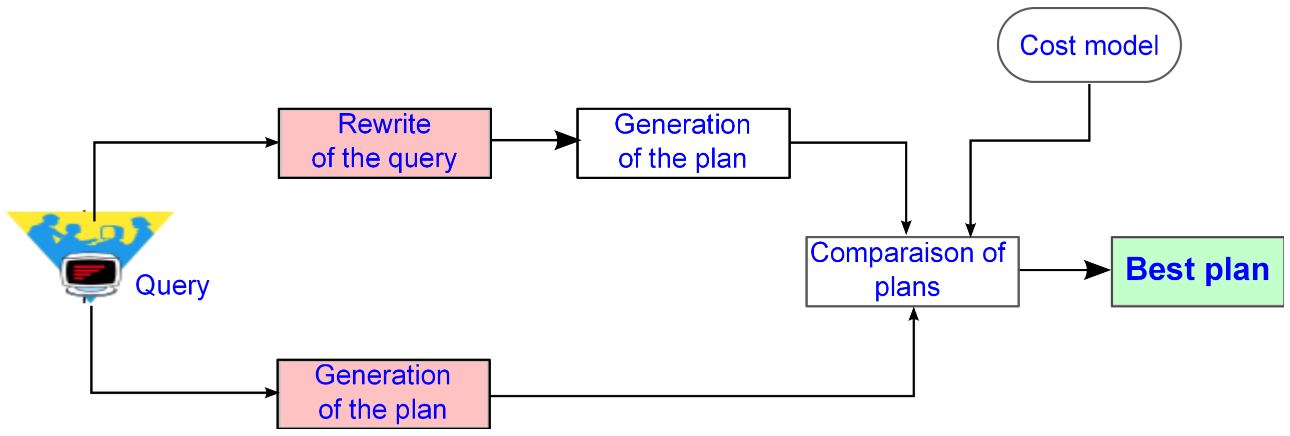


Fig. 2.8 – The query rewriting process

3. a set of non-functional requirements $\mathcal{NFR}$ (e.g., query processing cost, maintenance cost, energy consumption, etc.).

The $\mathcal{VSP}$ consists in selecting a set of materialized views that satisfies the $\mathcal{NFR}$ and respect the constraints $\mathcal{C}$.

From the above generic formalization, several variants of the $\mathcal{VSP}$ can be derived, by instantiating either the set of $\mathcal{NFR}$ or the constraints. The most important variants are:

- minimizing the query processing cost subject to storage size constraint [121];
- minimizing query cost and maintenance cost subject to storage space constraint [165];
- minimizing query cost under a maintenance constraint [119], and
- minimizing query processing and energy consumption under storage constraint [212].

The problems corresponding to these variants known as an NP-hard problem [120]. Several algorithms were proposed to deal with variants. For a complete classification of these algorithms, we recommend the readers to refer to the survey paper of Mami et al. [177] which divides algorithms in the following categories: *deterministic algorithms* [230, 89, 243], *randomized algorithms* [139], *evolutionary algorithms* [212] and *hybrid algorithms* [279].
The analysis of different variants of the $\mathcal{VSP}$ allows us giving a multidimensional representation (called the $\mathcal{VSP}$ cube) of these studies including three dimensions which are: *Non-functional requirements, used algorithms*, and *constraints* (Figure2.9). This cube can instantiate any work dealing with the $\mathcal{VSP}$.

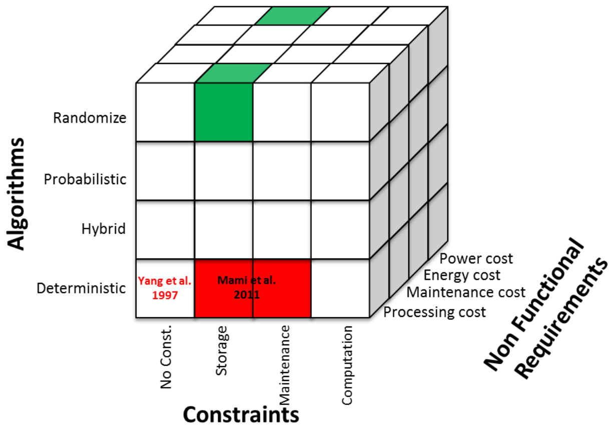


Fig. 2.9 – Views Selection Problem

Data structures & Algorithms for $\mathcal{MV}$ selection The traditional formalization selects views in a static manner, where it assumes that the queries are a priori known [121, 273]. To relax this hypothesis, a dynamic selection has been proposed. Algorithms in this selection may be divided into two categories, based on their incoming workload [86]: algorithms in the first category suppose that the query workload is predefined [203], whereas in the second category, the workload is unknown [161, 219].
Our discussion on $\mathcal{MV}$ selection algorithms is guided by two dimensions: the constraints and the $\mathcal{NFR}$.

- Chronologically, the first studies related to the $\mathcal{VSP}$ assume the absence of the constraints [273, 274]. After that and due to the large size of the stored selected materialized views, the storage space becomes the main constraint that the selection process has to integrated [279]. Then, maintenance cost becomes an important parameter since the selected materialized views need to be updated once the base tables change [120]. Some studies considered both constraints [120].

- $\mathcal{NFR}$ that the selected materialized views have to satisfy span several objectives: the minimization of query response time [211], the satisfaction of the maintenance cost [121]. Recently, we enriched these objectives by considering the energy consumption when executing a workload [212]. These objectives may be combined to give rise a multi-objective formalization of the problem of materialized view section problem such as in [273] (where response time and maintenance cost were considered) and in [212] (where response time and consuming energy are used).

The process of selecting materialized views passes through the identification of view candidates to be materialized. The number of these views can be very large [119]. Several research efforts have been conducted to represent efficiently the search space of the $\mathcal{VSP}$. Many data structures

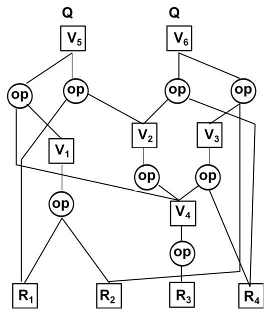


Fig. 2.10 – Example of AND-OR graph

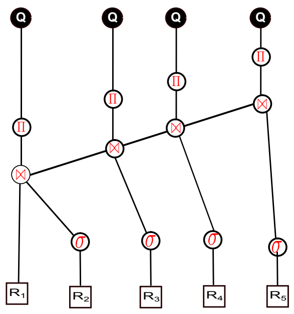


Fig. 2.11 – Example of acyclic graph

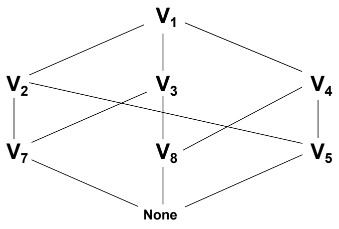


Fig. 2.12 – Example of Lattice Graph

have been used to represent and/or to prune the search space. The most used data structures are:

- **AND/OR view graph:** is a directed acyclic graph (DAG), which can be seen as the union of all possible execution plans of each query. It is composed of two types of nodes: operation and equivalent nodes. Each *operation node* corresponds to an operation in the query plan (selection, join, projection, etc.). Each *equivalence node* corresponds to a set of equivalent logical expressions (i.e., that yield the same result). The operation node have only equivalent nodes as children and equivalent nodes have only operation nodes as children. Many algorithms have been proposed in the literature to exploit this structure [120, 180, 213, 14, 178, 93]. Figure 2.10 gives an example of *AND/OR graph*, for a query.

- **Multi-View Processing Plan (MVPP):** is a directed acyclic graph in which the root nodes represent the queries, the leaf nodes correspond to the relations (base tables), and the intermediate nodes represent different operations used by the queries such as: selection, join, projection, etc. MVPP has been introduced by Yang et al. [273, 273]. Figure 2.11 gives an example of a MVPP.

- **Data Cube Lattice:** is a directed acyclic graph, proposed in the context of multidimensional data warehousing, which the nodes represent the queries or views (the views are characterized by grouping operation), and the edges define the relationship between views in each nodes [125, 144, 275]. Figure 2.12 gives an example of *Latice graph.*

- **Syntactical Analysis:** is a technique used by many algorithms that identify the candidate views directly by analysing synthetically the workload [5, 73].

- **Query plan:** it has been used by some algorithms to identify candidate views and rewrite queries [242, 243].

$\mathcal{VSP}$ may be either combined or not with the MQO problem. The work of Yang et al. [273] is the pioneer in the context of $\mathcal{DW}$ that deals with two interdependent problems: (a) constructing an optimal global plan of queries and (b) use this plan to select views to be materialized.

Construction of the global plan is performed following the bottom up scenario. Initially, the authors select the join local plans of each query (logical plans that have only join operations). These plans are merged in a single plan, called *Multi-Views Processing Plan* MVPP. The plan represented by an acyclic graph. This plan has four levels: at level 0, we find the leave nodes representing the base tables of the $\mathcal{DW}$. At level 1, we find nodes which represent the results of unary algebraic operations such as selection and projection. At level 2, we find nodes representing binary operations such as join, union, etc. The last level represents the results of each query. Each intermediate node of the graph is tagged with the cost of each operation and its maintenance cost. Two algorithms are proposed for selecting the best MVPP which has the minimum cost. The first algorithm, called *A feasible solution*, generates all possible MVPP and the plan with the minimum cost will be chosen. This algorithm is costly in terms of computation. To simplify the previous algorithm, a second algorithm is proposed based on *0-1 integer programming.* The view selection algorithm is performed in two steps: (1) generation of materialized views candidates which have positive benefit between query processing and view maintenance. This benefit corresponds to the sum of query processing using the view minus the maintenance cost of this view, (2) only candidate nodes with positive benefit are selected to be materialized [273, 274].
In [110, 40], a connection between MQO and the process of selecting materialized views for Sparql queries in the context of semantic databases has been established.

Based on this discussion, the traditional classification proposed in [177, 212] can be enriched. In addition to the traditional dimensions (the used constraints, non-functional requirements, type of selection algorithms), we can include other dimensions such as: nature of the selection (static/dynamic), used data structures to represent the search space, involving MQO. Figure 2.13 summarizes this classification.

View maintenance problem. Note that materialized views store data from base tables. In order to keep the views in the data warehouse up to date, it is necessary to maintain them in response to the changes at the base tables. This process of updating views is called *view maintenance* which has generated a great deal of interest in the past years. Views can be either recomputed from scratch, or incrementally maintained by propagating the base data changes onto the views. As re-computing the views can be prohibitively expensive, the incremental maintenance of views is of significant value [120].

Data Partitioning . Data partitioning is an $\mathcal{OS}$ that divides the data of database into distinct small parts called *fragment.* It aims at minimizing the unnecessary accesses to tables. Initially, it is considered as a logical database design technique which facilitates efficient execution of queries by reducing the irrelevant data accessed [29]. Nowadays, it is one of important aspect of physical design [6]. It is a divide-and-conquer approach that improves query performance, operational scalability, and the management of ever-increasing amounts of data. It is

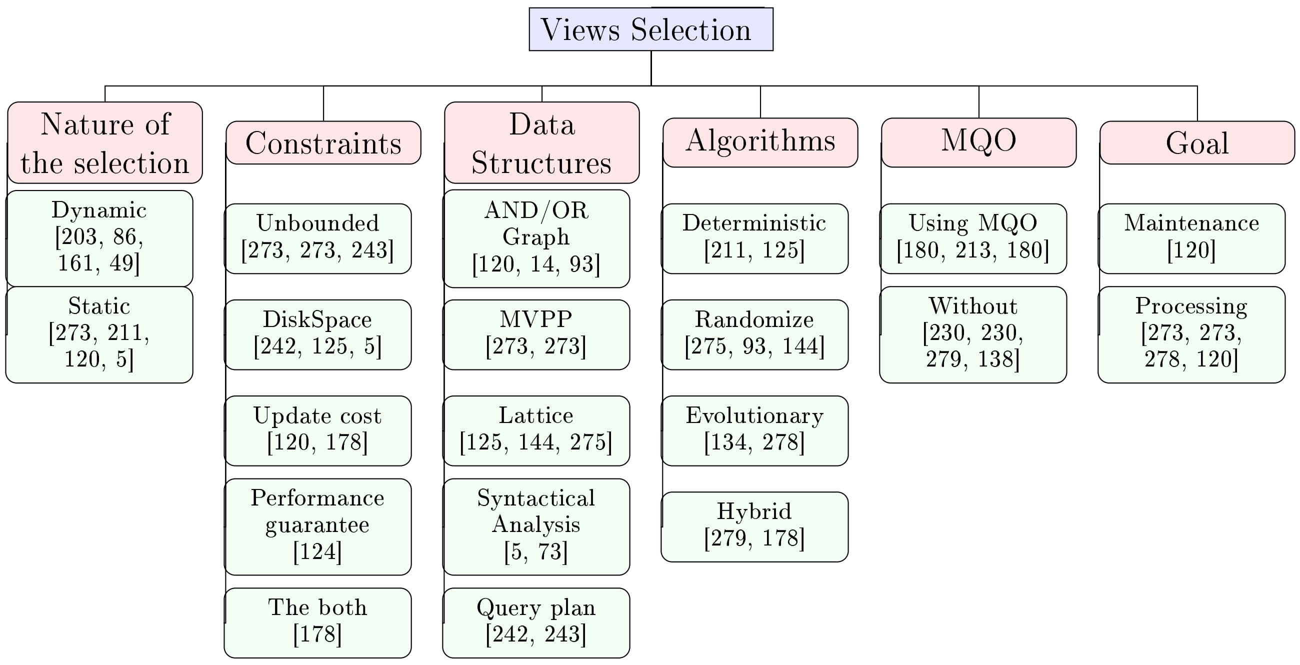


Fig. 2.13 *– Classification of view selection techniques*

also been widely used in the distributed and parallel databases.
Three types of data partitioning exists: horizontal, vertical and mixed.

- Horizontal data partitioning allows tables, indexes and materialized views to be partitioned into *disjoint* sets of rows that are physically stored and accessed separately [6] or in parallel. Horizontal partitioning may have a significant impact on performance of queries and manageability of very large data warehouses. It has the ability to be combined with other optimization structures like indexes, materialized views. Splitting a table, a materialized view or an index into smaller pieces makes all operations on individual pieces much faster. Contrary to materialized views and indexes, data partitioning does not replicate data, thereby reducing space requirements and minimizing update overhead [197].

  A native database definition language support is available for horizontal partitioning, where several fragmentation modes are available in Oracle 11G $\mathcal{DBMS}$: range, list and hash. In the range partitioning, an access path (table, view, and index) is decomposed according to a range of values of a given set of columns. The hash mode decomposes the data according to a hash function (provided by the system) applied to the values of the partitioning columns. The list partitioning splits a table according to the listed values of a column. These methods can be combined to generate *composite partitioning* (List-List, Range-Range, Hash-Hash, Range-List, etc.). Another mode of horizontal partitioning is also available in Oracle11g, called *virtual column based Partitioning.* It is defined by one of the above mentioned partition techniques and the partitioning key is based on a virtual column. Virtual columns are not stored on disk and only exist as meta-data.

Two versions of horizontal partitioning are available [195]: *primary* and *derived* horizontal partitioning. Primary horizontal partitioning of a table is performed using predicates defined on that relation. It can be performed using the different fragmentation modes above cited. Derived horizontal partitioning is the partitioning of a table that results from predicates defined in other table(s). The derived partitioning of a table R according to a fragmentation schema of table S is feasible if and only if there is a join link between R and S.

Example 4. *Using star schema of $\mathcal{DW}$ shown in Figure 2.4, that contains a fact table Lineorder and four dimension tables Customer,Supplier,Product and Dates. If the table Customer is candidate for partitioning, it may be divided using the attribute Region as follows:*

- $Customer_{Europe} = \sigma_{Region='EUROPE'}(Customer)$
- $Customer_{Others} = \sigma_{Region<>'EUROPE'}(Customer)$

The execution of the query that gives the European orders may be rewritten as follows:

```
SELECT Count(*)
FROM Customer PARTITION (Customer_Europe) C, Lineorder L
WHERE L.CID=C.CID;
```

The fragmentation of the *Customer* table can be propagated on the fact table *Lineorder* as follows:

- $Lineorder_{Europe} = Lineorder \ltimes Customer_{Europe}$
- $Lineorder_{Others} = Lineorder \ltimes Customer_{Others}$

Consequently, the search of number of European orders may be optimized as follows:

```
SELECT Count(*)
FROM Lineorder PARTITION(Lineorder_Europe);
```

In the context of relational data warehouse, derived horizontal partitioning is well adapted. In other words, to partition a data warehouse, the best way is to *partition some/all dimension tables using their predicates, and then partition the fact table* based on the fragmentation schemas of dimension tables [24]. This fragmentation takes into consideration star join queries requirements (these queries impose restrictions on the dimension values that are used for selecting specific facts; these facts are further grouped and aggregated according to the user demands). To illustrate this fragmentation, let us suppose a relational warehouse modelled by a star schema with d dimension tables and a fact table F. Among these dimension tables, g tables are fragmented ($g \leq d$). Each dimension table

D_i $(1 \leq i \leq g)$ is partitioned into m_i fragments: $\{D_{i1}, D_{i2}, ..., D_{im_i}\}$, where each fragment D_{ij} is defined as:
$D_{ij} = \sigma_{cl_j^i}(D_i)$, where cl_j^i and σ $(1 \leq i \leq g, 1 \leq j \leq m_i)$ represent a conjunction of simple predicates and the selection operator, respectively. Thus, the fragmentation schema of the fact table F is defined as follows: $F_i = F \ltimes D_{1j} \ltimes D_{2k} \ltimes ... \ltimes D_{gl}$, $(1 \leq i \leq m_i)$, where $\ltimes$ represents the semi join operation.

The number of schemes $\mathcal{N}$ equals to the number of fragments of fact tables and can be as the product of the number of fragments of each dimension tables.

$$\mathcal{N} = \prod_{i=1}^{d} Frag_i \tag{2.1}$$

which $Frag_i$ is the number of fragments of g dimension tables.

Derived horizontal partitioning (or referential partitioning in Oracle $\mathcal{DBMS}$) has two main advantages in relational data warehouses, in addition to classical benefits of data partitioning: (1) pre-computing joins between fact table and dimension tables participating in the fragmentation process of the fact table [22] and (2) optimizing selections defined on dimension tables.

The derived horizontal partitioning of a $\mathcal{DW}$ has been formalized as follows [24]: Given:

- $\mathcal{DW}$ with a set of d dimension tables $\{D_1, D_2, ..., D_d\}$ and a fact table F;
- a workload of queries $\mathcal{Q}$ and
- a maintenance constraint W fixed by DBA that represents the maximal number of fact fragments that he/she can maintain.

The referential horizontal partitioning problem consists in (1) identifying candidate dimension table(s), (2) splitting them using single partitioning mode and (3) using their partitioning schemes to decompose the fact table into N fragments, such that: (a) the cost of evaluating all queries is minimized and (b) $N \leq W$. This problem has been proven as a NP-complete problem [24].

- Vertical data partitioning can be viewed as a redundant structure even if it results in little storage overhead. The vertical partitioning of a table T splits it into two or more tables, called, sub-tables or vertical fragments, each of which contains a subset of the columns in T. Note that the key columns are duplicated in each vertical fragment, to allow "reconstruction" of an original row in T. Since many queries access only a small subset of the columns in a table, vertical partitioning can reduce the amount of data that needs to be scanned to answer the query. Unlike horizontal partitioning, indexes or materialized views, in most of today's commercial database systems there is no native database definition language support for defining vertical partitions of a table [6].

To vertically partition a table with m non primary keys, the number of possible fragments is equal to $B(m)$, which is the m^{th} Bell number [195]. For large values of m, $B(m) \cong m^m$. For example, for m = 10; $B(10) \cong 115\ 975$. These values indicate that it is futile to attempt to obtain optimal solutions to the vertical partitioning problem. Many algorithms were proposed and classified into two categories: grouping and splitting [195]. Grouping starts by assigning each attribute to one fragment, and at each step, joins some fragments until some criteria is satisfied. Splitting starts with a table and decides on beneficial partitioning based on the query frequencies.
In the data warehousing environment, [113] proposed an approach for materializing views in vertical fragments, each including a subset of measures possibly taken from different cubes, aggregated on the same grouping set. This approach may unify two or more views into a single fragment.

- The above partitioning techniques can be mixed. *In this thesis, we mainly focus on horizontal data partitioning.*

Classical methodology of $\mathcal{HDP}$ In $\mathcal{DW}$, horizontal data partitioning follows two main phases [45]:

- **preparation phase:** in this phase selection predicates and candidate tables are identified. Once this identification done, each domain of a fragmentation attribute (belonging to a selection predicate) is divided into sub-domains. Finally, the set of minterms is generated [195]

- **selection phase** that identifies the final partitioning schema using a given algorithm.

Correction rules for $\mathcal{HDP}$ In horizontal data partitioning, the partitioning process is verified using three rules: completeness, disjunction, reconstruction [45]. To understand these rules, we suppose that we have a table T partitioned into k fragments $\{T_1, .., T_k\}$.
Completeness: means that for all tuples in the table T must be existing at list in one fragment T_i, $1 \leq i \leq k$. This rule ensures that no loss of data and all data exist in the partitioned schema.
Disjunction: means that for all instances i in the fragment T_j, no other fragment T_k that contains this instance i. This rules ensures that non redundant of data.
Reconstruction: means that the original table T can be formed using the set of fragments T_i, $1 \leq i \leq k$. This rule ensures that data partitioning is a reversible operation.

A Taxonomy of data partitioning approaches Due to its importance, the data partitioning has been widely studied in the literature. By analysing these studies, we propose to classify using the following criteria: type of partitioning, type of databases, type of algorithms, the platform and the using of data structure (Figure 2.15).

- **Type of partition:** There are three types of data partitions, vertical data partition [185, 186], horizontal data partitioning [18, 82, 241, 281], and hybrid data partitioning [45, 84, 209, 238, 188, 199].

- **Type of database:** Data partitioning has been applied in all database generations: relational databases [186, 115], object-databases [209, 188, 184, 34], deductive database [182], $\mathcal{DW}$ [18, 45, 189], and recently in graph databases [202]. Also data partitioning are applied on some optimization structures like indexes [241].

- **The used Algorithms:** In the first generation of the existing work, the problem is treated without any constraint: based on Minterms [67, 195] or affinity [146]. In the second generation, the maintenance constraint is considered that is defined by the maximum number of fragments, in which cost based approaches have been proposed [29]. In this generation, others techniques based on data mining are proposed [175]. Choosing the best predicates is not a easy task. Another point is the ignorance of the interaction between queries, except the work of Kerkad et al. [155, 154]. Figure 2.14 shows the evolution approaches of $\mathcal{HDP}$.

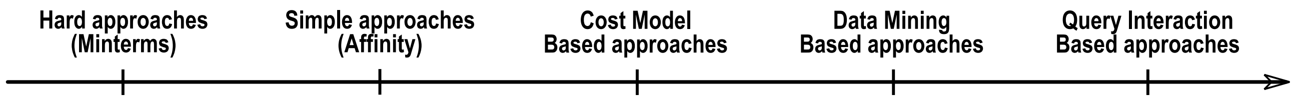

Hard approaches (Minterms)
Simple approaches (Affinity)
Cost Model Based approaches
Data Mining Based approaches
Query Interaction Based approaches

Fig. 2.14 *– The evolution of $\mathcal{HDP}$ approaches*

- **Platforms:** used to allocate the fragments differentiate data partitioning approach, in which it can be centralized platform [45, 18, 82, 186, 185, 115, 76], distributed platform [34, 184, 189] or parallel platform [281, 238, 209, 241, 188, 199, 87].

- **Data structures:** used to represent the candidates or the result of the process of data partitioning, in which, the most used is the array structures like affinity Matrix [185, 186], graphs [271] and hypergraphs [81];

III.3.3. $\mathcal{HDP}$ in Distributed Environments

In the context of distributed databases, the fragmentation/partitioning is tightly coupled with the so-called *fragment allocation problem* [128]. Indeed, in real-life application scenarios, these two processes, even thought inter-related, are tackled in an isolated manner. This evidence has generated two different research contexts. The first one is focused on the data fragmentation problem, whereas the second one on the issue of allocating partition-generated fragments. Also, this dichotomy implies the presence of two different cost models for the fragmentation and allocation phases, respectively. As a consequence, the fragmentation cost model very often does not

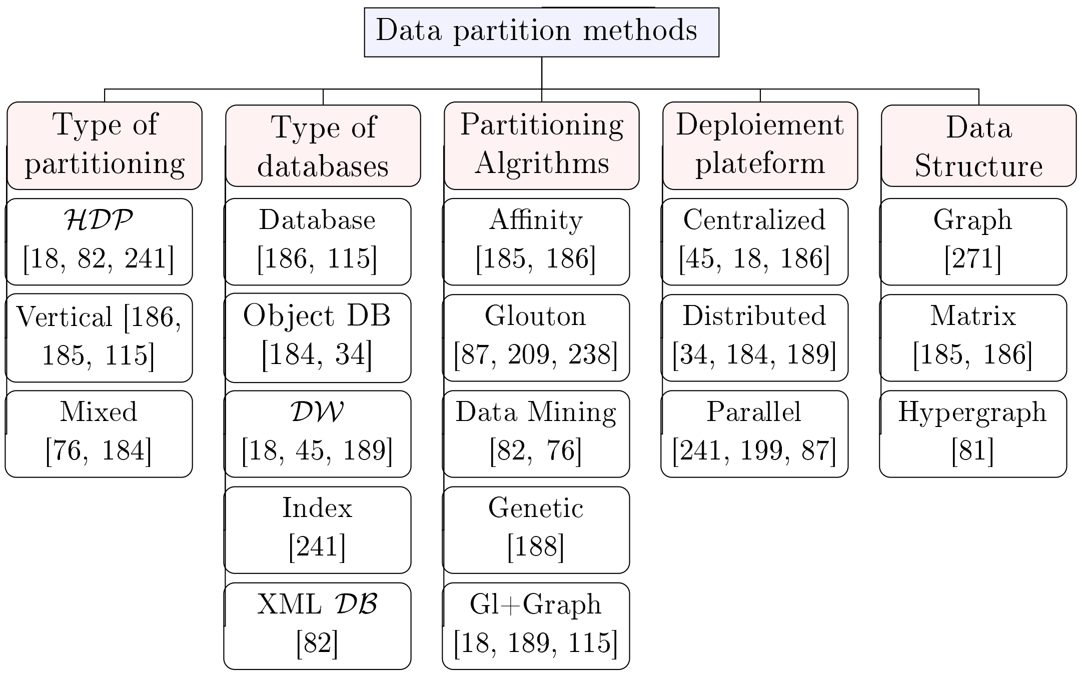


Fig. 2.15 *– Taxonomy of data partitioning methods*

take into account the yet-relevant allocation parameters. To face-off deriving drawbacks, several approaches have been proposed with the goal of integrating the two distinct fragmentation and allocation processes, usually according to a *sequential manner*.

In line with this approach, Furtado [108] discusses partitioning strategies for *node-partitioned data warehouses*. The main suggestion coming from [108] can be synthesized in a "best-practice" recommendation stating to partition the fact table on the basis of the *larger* dimension tables (given a ranking threshold). In more detail, each larger dimension table is first partitioned by means of the *Hash mode* approach via its primary key. Then, the fact table is again partitioned by means of the Hash mode approach via foreign keys referencing the larger dimension tables. Finally, the so-generated fragments are allocated according to two alternative strategies, namely *round robin* and *random*. Smaller dimension tables are instead fully-replicated across the nodes of the target data warehouse. The fragmentation approach [108] does not take into account specific star query requirements, being such queries very often executed against data warehouses, and it does not consider the critical issues of controlling the number of generated fragments, like in [237, 24].

In [171], Lima *et al.* focus the attention on data allocation issues for database clusters. Authors recognize that how to place data/fragments on the different PC of a database cluster in the dependence of a given criterion/goal (e.g., query performance) plays a critical role, hence the following two straightforward approaches can be advocated: (*i*) full replication of the target database on *all* the PC, or (*ii*) meaningful partition of data/fragments across the PC. Starting from this main intuition, authors propose an approach that combines partition and replication for OLAP-style workloads against database clusters. In more detail, the fact table is partitioned and replicated across nodes using the so-called *chained de-clustering*, while dimension tables are fully-replicated across nodes. This comprehensive approach enables the middleware layer to perform load balancing tasks among replicas, with the goal of improving query response time. Furthermore, the usage of chained de-clustering for replicating fact table partitions across nodes allows the designer not to detail the way of selecting the number of replicas to be used during the replication phase. Just like [108], [171] does not control the number of generated fact table fragments.

In [237], Stöhr *et al.* propose an approach for constructing and managing a data warehouse on a *disk-shared parallel machine* having K disks. Here, the data warehouse is modeled via a star schema with one fact table and a number of dimension tables. The fragmentation process is performed as follows: each dimension table is virtually partitioned by means of the *Interval mode* on attributes belonging to the lower levels of dimensional hierarchies, and the fact table is consequentially partitioned on the basis of so-partitioned dimension tables. Dimension tables and ad-hoc B-tree indexing data structures are duplicated over each disk of the parallel machine. To speed-up queries, *bitmap join indexes* are selected over fact table fragments via using attributes of dimension tables belonging to the higher levels of dimensional hierarchies. Note that this approach may generate a number of fragments N_F largely greater than the available number of disks K. To ensure a high parallelism degree and efficient load balancing, a round robin allocation of fact fragments and associated bitmap indexes over the K disks is finally performed, with the goal of placing bitmap indexes that are associated to the same fact fragment onto consecutive disks. In [238], the same authors further extend they research by proposing a data allocation tool, called *Warlock*, for parallel data warehouses. The novelty of *Warlock* consists in taking into account the number of final fact fragments that the designer is interested to. Finally, in [27] an innovative design methodology for data warehouses over *shared-nothing architectures* that initially argues to perform fragmentation and allocation jointly is proposed. Table 2.1 summarizes main characteristics of comparison approaches for designing data warehouses in distributed environments via fragmentation and allocation paradigms.

***Table 2.1** – A Comparison of main approaches*

	Fragmentation & Allocation	*Design*	*Environment*	Control over the Number of Fragments
[83]	Yes	Sequential	Grid	Yes
[108]	Yes	Sequential	DB Cluster	No
[171]	Yes	Sequential	DB Cluster	No
[237]	Yes	Sequential	Disk-Shared	No
[27]	Yes	Combined	Shared-Nothing	Yes

Indexes. *Indexing* has been at the foundation of performance tuning for databases for many years. A database index is a data structure that improves the speed of operations in a table. Indexes can be created using one or more columns. An index can be either clustered or non-clustered. It can be defined on one table, fragment, views or many tables using a join index [192]. The traditional indexing strategies used in database systems do not work well in data warehousing environments since most OLTP queries are point queries. B-trees, which are used in most common relational database systems, are geared towards such point queries. In the data warehouse context, indexing refers to two different things: (a) indexing techniques and (b) index selection problem.

Indexing techniques. A number of indexing strategies have been suggested for data warehouses: Value-List Index, Projection Index, Bitmap Index, Bit-sliced Index, Data Index, Join Index, and Star Join Index. Bitmap index is probably the most important result obtained in the data warehouse physical optimization field. The bitmap index is more suitable for low cardinality attributes, since its size strictly depends on the *number of distinct values of the column on which it is built.* Besides disk space saving (due to their binary representation and potential compression), such index speeds up queries having Boolean operations (such as AND, OR and NOT) and COUNT operations. Bitmap join index is proposed to speed up join operations. In its simplest form, it can be defined as a bitmap index on a table R based on a single column of another table S, where S commonly joins with R in a specific way.

Index selection problem. The task of index selection is to automatically select an appropriate set of indices for a data warehouse (having a fact table and dimension tables) and a workload under resource constraints (storage, maintenance, etc.). It is challenging for the following reasons [68]: The size of a relational data warehouse schema may be large (many tables with several columns), and indices can be defined on a set of columns. Therefore, the search space of indices that are relevant to a workload can be very large [32]. To deal with this problem, most selection approaches use two main phases: (1) *generation of candidate attributes* and (2) *selection of a final configuration.* The first phase prunes the search space of index selection problem, by eliminating *non relevant* attributes. In the second phase, the final indices are selected using greedy algorithms [70], linear programming algorithms [68], etc. The *quality of the final set of indices depends essentially on the pruning phase.* To prune the search space of index candidates, many approaches were proposed [68, 32, 22], that can be classified into two categories: *heuristic enumeration-driven approaches* and *data mining driven approaches.*
In heuristic enumeration-driven approaches, heuristics are used. For instance, in [70], a greedy algorithm is proposed that uses optimizer cost of SQL Server to accept or reject a given configuration of indices. The weakness of this work is that it *imposes the number of generated candidates.* IBM DB2 Advisor is another example belonging to this category [257], where the query parser is used to pick up selection attributes used in workload queries. The generated candidates are obtained by a few *simple combinations* of selection attributes [257].

In data mining-driven approaches, the pruning process is done using data mining techniques, like in [32]. In this approach, the number of index candidates is not a priori known as in the first category. The basic idea is to generate frequent closed itemsets representing groups of attributes that could participate in selecting the final configuration of bitmap join indexes. A data mining based approach has been developed for selecting bitmap join indexes [32].

III.3.4. MQO and physical design problems

In the section, we present the unification of the problem of MQO and the physical design.

MQO and $\mathcal{MV}$ problems MQO contributes largely in different stages of materialized views: in the selection of the best views [13, 120, 273] and in their maintenance [131, 180, 211]. Construction of the UQP is performed following a bottom up scenario. Initially, the authors select the join local plans of each query. These plans are merged in a single UQP, called in [273] *Multi-Views Processing Plan* (MVPP). Each intermediate node of the MVPP is tagged with the cost of the processing data and the maintenance cost in the case if it is materialized. Then, the MVPP is used as an input for selection algorithms.

MQO and data partitioning problems To the best of our knowledge, the work proposed by *Kerkad et al.* [30] is the only work that exploits query interaction for resolving the problem of horizontal data partitioning in the context of relational data warehouses. More correctly, it uses the sub-domains of attributes obtained by applying selection predicates of the queries to prune the search space of the horizontal partitioning problem and identifies the selection predicates candidates for partitioning by considering the most influenced attributes and their sub-domains in terms of reducing the cost of the UQP.

MQO and Buffer management and query scheduling The query ordering has used to improve MQO algorithm [80]. Hence, Kerkad et al., [155], have proposed a new technique of using intermediate results to solve the buffer management and query scheduling problems, in which the intermediate results have been used to define a strategy for managing the buffer (allocation/deallocation) and schedule queries. In [284] a *cooperative scans* have been proposed to improve sharing between queries by dynamic scheduling queries that share the scans of data from base tables.

MQO and caching of query results Another related area is caching of query results. MQO can optimize a batch of queries given together, caching takes a sequence of queries over time, deciding what to materialize and keep in the cache as each query is processed [280, 161, 74].
It should be noticed that the algorithms used in these different selections are workload sensitive. Since, the used queries are too small, these algorithms may suffer from scalability in the context of big queries.

The queries are also one of the main components of the deployment phase that we discuss it in the following section.

III.4. Dimension 4: Deployment Phase

The $\mathcal{DBMS}$ hosting a $\mathcal{DW}$ application is deployed in a hardware platform that offers storage layers. It should be noticed that the logical and physical optimizations strongly depends on the deployment platforms of the target $\mathcal{DW}$. The deployment phase of the life cycle design of $\mathcal{DW}$ uses is one of the consumers of OLAP queries. There exist two fundamental types of architecture suitable for deploying database applications. The most well-known and ubiquitous architectural type is the centralized $\mathcal{DBMS}$ architecture. It is very well defined and mature. The other type is distributed and parallel $\mathcal{DBMS}$.
In this section, we consider an example of parallel $\mathcal{DW}$. This is motivated by the spectacular explosion of the data volume and the queries. The terms "parallel $\mathcal{DW}$ " and “distributed /dw” are very often used *interchangeably*. In practice, however, the distinction between the two has historically been quite significant. Distributed $\mathcal{DW}s$, much like distributed databases, grew out of a need to place processing logic and data in proximity to the users who might be utilizing them. In general, multi-location organizations (motivated by globalization) consist of a few distinct sites, each typically associated with a subset of the information contained in the global data pool. In the $\mathcal{DW}$ context, this has traditionally led to the development of some form of *federated* architecture. In contrast to monolithic, centralized $\mathcal{DW}s$, federated models are usually constructed as a cooperative coalition of departmental or process specific *data marts* [28]. For the most part, design and implementation issues in such environments are similar to those of distributed operational $\mathcal{DBMS}$ [195]. For example, it is important to provide a *single transparent conceptual model* for the distinct sites and to divide data to reduce the effects of *network latency* and *bandwidth limitations*.

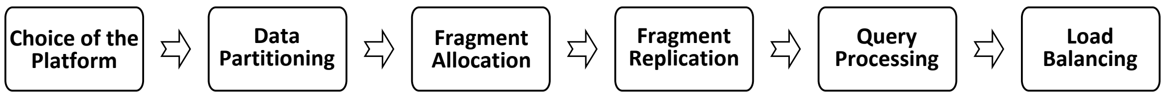
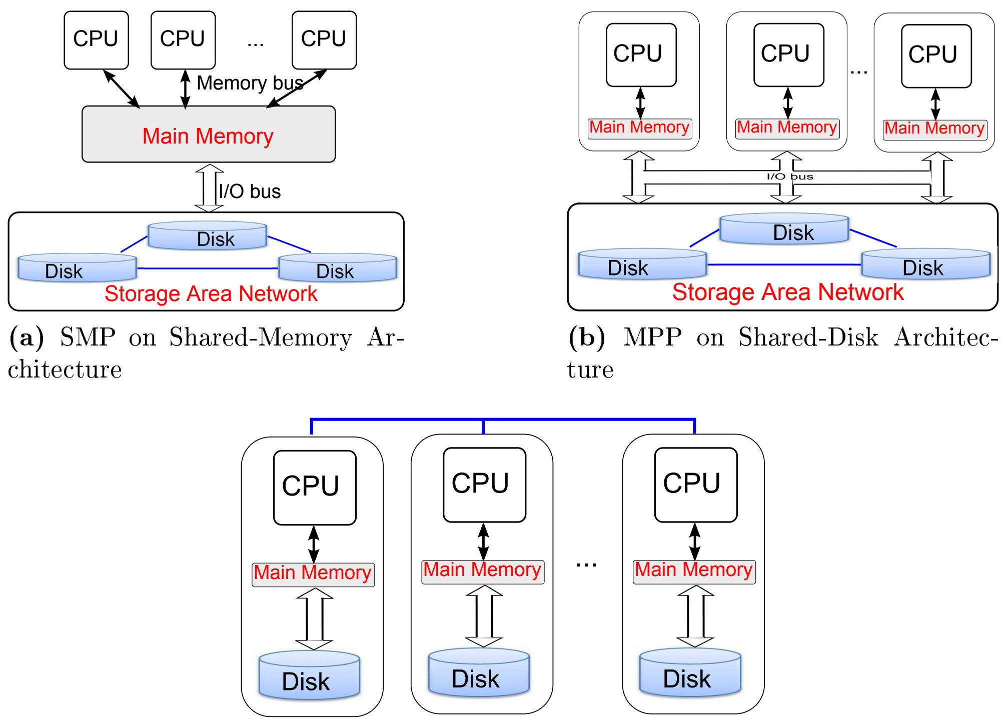


Fig. 2.16 *– Life cycle of the parallel database design*

Designing a parallel $\mathcal{DW}$ goes through a well-identified life cycle including six main steps (Figure 2.16) [27]: **(1)** choosing the hardware architecture, **(2)** partitioning the target $\mathcal{DW}$, **(3)** allocating the so-generated fragments over available nodes, **(4)** replicating fragments for efficiency purposes, **(5)** defining efficient query processing strategies and **(6)** defining efficient load balancing strategies. Each one of these steps will be detailed in the following paragraphs.

1. **The choice of the platform:** The choice of the deployment platform is a pre-condition to achieve the scalability and availability of data. Hardware architectures are classified into five main categories: (i) shared-memory, (ii) shared-disk, (iii) shared-nothing, (iv)

shared-something and (v) shared-everything. As we said before, the **Shared-everything** architecture has been adopted by $\mathcal{DW}$ for parallelization purpose. **Shared memory and shared disk architectures** have the advantage that they are relatively easy to administer as the hardware transparently performs much of the "magic". This architecture has been recommended by the leaders of Gamma project as the reference architecture for supporting high-performance data warehouses modelled in terms of relational star schemes. That being said, shared everything designs also tend to be quite expensive and have limited scalability in terms of both the CPU count and the number of available disk heads. In Petabyte-scale $\mathcal{DW}$ environments, either or both of these constraints might represent a serious performance limitation. As the choice of the hardware architecture is influenced by price, high-performance features, extensibility and data availability, clusters of workstations are very often used as a valid alternative to Shared-Nothing architectures (e.g., [171]). Figure 2.17 summarizes these architectures.



(a) SMP on Shared-Memory Architecture

(b) MPP on Shared-Disk Architecture

(c) Shared-Nothing Architecture

***Fig. 2.17** – Data warehouse system architectures*

2. **Data partitioning.** The partitioning in the context of parallel $\mathcal{DW}$ is based mainly in horizontal partitioning (for more details, see the section III.3.2).

3. **Data allocation** consists in placing generated fragments over nodes of a reference paral-

lel machine. This allocation may be either redundant (with replication) or non-redundant (without replication) [12]. Once fragments are placed, global queries are executed over the processing nodes according to parallel computing paradigms. In more detail, parallel query processing on top of a parallel $\mathcal{DW}$ (a critical task within the family of parallel computing tasks) includes the following phases [36]: (i) rewriting the global query according to the fixed $\mathcal{DW}$ partitioning schema; (ii) scheduling the evaluation of so-generated sub-queries over the parallel machine according to a suitable allocation schema. Generating and evaluating sub-queries such that the query workload is evenly balanced across all the processing nodes is the most difficult task in the parallel processing above.

4. **Replication of fragments.** is related to two major problems: (1) replica creation and (2) maintenance of materialized replicas. Our study focuses on the problem of creating replicas which is strongly influenced by the number of replicas and their placement. Indeed, the reference Replica Placement Problem (RPP) consists in choosing the best replica placement on the distributed system in order to optimize given performance criteria. The optimal replica placement problem has been shown to be an NP-Hard problem [270]. Therefore, it is often solved by means of approximate solutions in a feasible time, and a relevant amount of work has been devoted to this paradigm in the literature [171].

   Three complexes problems related to replication technique: (1) the selection of candidate fragments to be replicated and how much, (2) the placement of replicas and, (3) the updating methods. Replication techniques and updating replicas can be classified into two strategy [118].

5. **Load balancing of queries** Once partitioning, allocation and replication data are done; load balancing of queries occurs to rewrite each query in many sub-queries and affect them over processing nodes. The main challenge is to ensure the most equilibrate work of processing nodes to response a query or workload, and minimize the average processing time, in which minimize the difference of consuming time from node to anther.

   To obtain a good load balancing, it is necessary to determine the best degree of parallelism and choose wisely the processing nodes for executing a query. The load balancing process has four steps [179]:

   - **partitioning of the workload:** The coordinator node handles to partition the workload of queries into a set of sub-queries according to the horizontal partitioning scheme for the database. Indeed, the horizontal partitioning produces sub-queries that can run on independent processing nodes. The size of fragments affect the quality of load balancing, the small fragments ensures more flexibility and reduces the effects of skew. Thus, each sub-query is characterized by its size that corresponds to the access time of loading fragments from disks.
   - **Selection of processing nodes:** For each sub-query, all possible processing nodes are determined by the placement pattern of fragments on the processing nodes and

the cache status of each node.

- **Scheduling of sub-queries:** Sub-queries are assigned to processing nodes, in which all nodes have approximately the same workload. Scheduling sub-queries is to minimize the average response time of tasks and maximize the degree of resource utilization. The determination of the distribution of the final charge is predetermined by the data allocation, system architecture, the cache, and the dependencies between data. The scheduling of sub-queries also defines the order of query execution.
- **Processing of sub-queries :** When sub-queries are affected to processing nodes, they start to verify the existence of fragments on its disks (If the needed fragments are not found, they get a copy from its neighbours). Once all fragments are available, the execution of sub-queries start.

A good load balancing is critical to the performance of a parallel system where the response time corresponds to later processing node. Load balancing can be influenced by several problems: competition control (parallel execution requires simultaneous access to shared resources.), interference and communication, and the skewed distribution[67].

The bad balancing is mainly affected by the skewed distribution of data and / or processing. Walton et al. [265] have classified the effects of a poor distribution of data in a parallel execution as follows:

- **Attribute Value Skew:** it occurs when the attribute values are not distributed uniformly over processing nodes. This means that some attribute values appear with much higher frequencies than others.
- **tuples Placement Skew:** is the consequence when the initial partition and allocation based on attributes with poor distribution of values.
- **Selectivity Skew:** is due to the variation of the selectivity of the selection predicates between processing nodes.
- **Redistribution Skew:** occurs when there is a difference between the distribution of values of join keys.
- **Result Size Skew:** It occurs when there is a big difference between the size of the results obtained by each processing nodes.
- **Capacity Skew:** It represents the workload that each processor is able to process it. This may be due to heterogeneity for the architecture and the operating system such as memory size, version of the operating system, computing power, storage capacity.
- **Processing Skew:** It occurs when there a difference of the execution time of the sub-queries on the processing nodes.

III.4.1. Metrics of deployment quality

To evaluate the quality of deployment two main metrics are used: speed-up and scale up. The speeding factor (Speed-Up) measures the performance gain obtained by increasing the number of processing nodes. If a query Q, needs T_s units (e.g., seconds) to be executed sequentially, and T_p units to be executed in parallel manner on p processing nodes, the *speed-up* is defined as follow:

$$speed - up(p) = \frac{T_s}{T_p} \tag{2.2}$$

The scaling factor (Scale-Up) measures the change of query response time following a proportional increase of database size and the number of processing nodes. The scale-up is ideal if it is still equal to 1 (called also linear scaling).
Let db_1 and db_2 be two databases with the following sizes: $size(db_1)$ and $size(db_2)$. If $size(db_2)$ = N * $size(db_2)$, and T_{db_i,p_i} is query response time of the query Q on the db_i using p_i processing nodes, the scale factor is defined as follows:

$$scale - up = \frac{T_{db_1,p_1}}{T_{db_2,N*p_1}} \tag{2.3}$$

Based on the above discussion, we can give a formalization of the parallel $\mathcal{DW}$ design: Given:

- a logical schema of data warehouse that contains a fact table F, and a set D of d dimension tables D =$\{D_1, .., D_d\}$;
- a workload $\mathcal{Q}$ of k queries $\mathcal{Q}$ =$\{Q_1, .., Q_n\}$, each query i_q has a access frequency f_i, for for $1 \leq i \leq n$;
- a set $\mathcal{N}$ of M processing nodes $\mathcal{N}$ =$\{N_1, .., N_M\}$, in which each N_i has a storage capacity S_i, for $1 \leq i \leq M$;
- a set of non-functional requirements like the maximum query response time ($response_{max}$);
- technical constraints like maximum number of fragments W.

The problem of designing a parallel consists in fragmenting the $\mathcal{DW}$ and then allocating its obtained fragments over the nodes that the overall the cost of executing queries is minimized and the used constraints are satisfied [36].

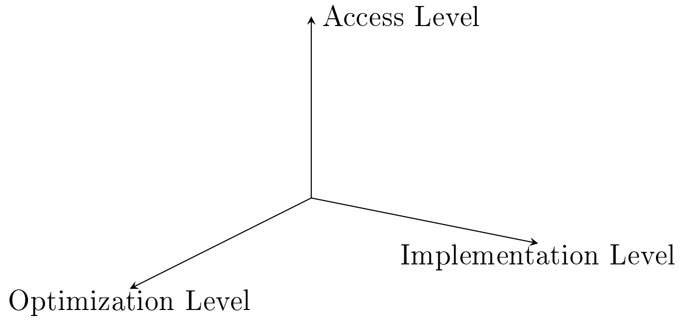


***Fig. 2.18** – Data structures and algorithms space of data processing in databases*

IV. Data structures and access algorithms

In this section, we discuss the different usages of data structures in the context of databases. We can distinguish three main usages: (i) representation and modelling stored data, (ii) supports for algorithms dedicated to all problems that we can find in different phases of the life cycle of the design, and (iii) representation of access methods. So, these data structures span three levels concerning optimizations, accesses and implementations.

The implementation level describes data modelling (logically and physically) and access algorithms to retrieve, add or update data from and to storage devices. The **access level** describes the real implementations of optimizations structures such as $\mathcal{MV}$, indexes, logical query plan, data partitioning, data cube. The **optimization level** contributes in implementing and resolving traditional optimization problems (e.g., physical design, ETL, deployment, etc.).
Figure 2.18 shows the three dimensions describing the roles of DS in the context of databases. In the following, we overview some data structures used in different levels.

IV.1. Data structures in implementation level

In the **implementation level**, the data are stored in secondary storage devices (e.g., hard disks), where DS are used to represent data. In the logical DS, data are represented by many formats like, *file*, unstructured data, key-value, semi-structured data like *XML* file [170], table [221], row-store [1], column-store [163, 1, 239], etc. In the physical DS, data are stored based on specific format of file system, like *HDFS* [3], *NTFS* [4], etc. The file system depends on the type of storage devices (*Hard disk*, *Flash memory*, etc.). For example, data in *HDFS*, the data are chopped up in several blocks of 64 MB. Each DS needs basic operations like reading, adding, updating, deleting or finding a value of data, which they are implemented with many

[3]https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[4]https://technet.microsoft.com/en-us/library/cc781134%28v=ws.10%29.aspx

algorithms, e.g., for reading, the basic implementation of the algorithm is to read page by page. Note that, these DS are used to store data and accessing algorithms, define the efficiency of any $\mathcal{DBMS}$ [172, 91]. So, In physically store, data are allocated generally, on pages, cluster and sectors. And in the logically store, data are stored as many formats like, files, *tables* [75], *binary* [244], *key-values* [90], *triplet*[40].
In this level, the main problems related to data are: *reading*, *adding*, *deleting*, and *updating.* Due to the data volume, having efficient access methods to find quickly a target data in storage devices *is necessary.* As a consequence, many optimization structures like indexes have been proposed and implemented.
Hence, in the implementation level, access methods are involved for several purposes: to represent data, to resolve a set of related problems (read, delete, insert data), and to use a set of access techniques to facilitate and accelerate resolving related problems.

IV.2. Data structures in access level

In the access level, data concerns mainly optimization techniques to speed up queries. The query optimization covers two main aspects: the physical design, identification of the best query plans. To ensure their efficiency, the optimization structures have to contain a summary of all or a part of original data needed by queries to avoid as much as possible costly accesses to original data.

IV.3. Data structures in Optimization level

In the optimization level, the DBA is always working to find the optimal configuration corresponding to one or several $\mathcal{OS}$. Selecting optimal $\mathcal{OS}$ has three main steps:

1. the representation of the search space by one or more data structures;
2. the reduction of the search space in small set of solution candidates to minimize the complexity of selection algorithm;
3. the selection of the optimal configuration based using naive or advanced algorithms (evolutionary algorithms, systematic algorithms, randomize algorithms, etc.).

The elements of search space and the set of candidates of $\mathcal{OS}$ are defined using one or more DS. Often; it is impossible to list all their elements which require defining the search space using Abstract Data Type (ADT), like graph and tree. Applying rules and heuristics on these ADT gives an infinite possible $\mathcal{OS}$, there they can be represented by a DS like *array*, *graph*, *tree*, *linked lists*, etc.
The main related problems in this level are selecting the optimal $\mathcal{OS}$ under constraints, that they often considered as combinatorial problems with *NP-hard* complexity [120, 68, 216]. Accordingly, the researchers have always used different types of heuristics to reduce the complexity of the problem and then identify a near optimal solution.

To summarize, in the optimization level, the elements of possible search space are usually defined using ADT and the elements of candidate $\mathcal{OS}$ are defined using one or more DS. The main problems related to data are the selection of the optimal configuration of one or more $\mathcal{OS}$. Figure **??** gives the modelling space of data in the optimization level.

IV.4. Discussion

In this chapter, we show the role of the interaction among queries in physical design and deployment phases. This interaction has certainly exploited by researchers to propose *large spectrum of algorithms* [177]: deterministic algorithms, randomized algorithms, genetic algorithms, hybrid algorithms. These algorithms explore the nodes of the global plan of the interacted queries to identify the optimization schemes. These algorithms do not highlight their data structures. In the context of big queries, the presence of flexible and scalable data structure is recommended to define advanced and efficient algorithms. Note that the global plan is a *graph*, usually can be partitioned to ensure scalability of algorithms. Graph partitioning is a fundamental problem, with applications to many areas such as parallel computing and VLSI layout. The goal of graph partitioning is to split the nodes in graph into several disjoint parts such that a predefined objective function, for example, ratio-cut, or normalized cut, is minimal. This partitioning saves the characteristics of the initial search space represented by the global plan.
Another point that we realized when analyzing these studies is the absence of a connexion between the problem of physical design and the problem of MQO. More precisely, to select an optimization structure such as materialized views, the majority of the proposals use a sequential methodology meaning that they assume that the problem MQO is already solved and then they use its solution to run their algorithms. Due to the strong interdependency between these two problems, a joint methodology is recommended.

V. Conclusion

This chapter gave an overview, accompanied with analysis and comparison of the different concepts used in our thesis. We started by introducing the data warehousing technology including its design and exploitation by OLAP queries. The performance of the exploitation process is linked to the physical design of the life cycle of the data warehouse design that includes: user requirements, conceptual, logical, deployment and physical phases. Afterwards, we concentrated our discussion on the OLAP queries and their characteristics: they are routinely and share common intermediate results. In the Era of very large database applications, their optimization has become a crucial issue. The characteristics of OLAP queries force us to use multi-query optimization solutions. Finally, based on this overview, we deduced the following: to optimize very large queries defined on extremely large databases, we need a scalable data structures associated with advanced algorithms.

The next chapter, we present in details our scalable data structure based on hypergraphs to represent queries and their intermediate results.

Part II

Contributions

Chapter

3 Modeling Query Interaction using Hypergraphs

Contents

Abstract

In this chapter, we present our first contribution that concerns the proposition of hypergraph driven data structure which is the fruit of our collaboration with Mentor Graphics, Inc. In the EDA domain, we manipulate logical circuits with millions of gates. For simulation and testing purposes, the hypergraph data structures are widely used. This manage this volume, simulation and testing algorithms use the *divide and conquer principle.* As a consequence, several hypergraph partitioning libraries exit. This situation motivates us to find an analogy between the representation of a unified query plan and a logical circuit. Once this analogy established, we adapt EDA hypergraph algorithms and tools to our problem. Intensive experiments are conducted to evaluate the efficiency and the quality of our proposal.

I. Introduction

As we have mentioned in the previous chapters, the generation of unified query plan for a workload with high interacted queries is an NP-hard problem [222]. Several research efforts have been triggered to tackle this problem, where a large panoply of algorithms have been proposed that can be divide into three main categories: (a) dynamic programming algorithms [246, 273], (b) genetic algorithms [17] and (c) randomized algorithms [139]. We would like to mention that these algorithms have been tested for workloads with small sets of queries and can require high computation time. Facing this situation, we have two scenarios: (i) reproduction of the existing algorithms. However, it is not obvious that a particular algorithm Coarsening is reproducible to a sufficient extent with this technique to deal with a large scale search space and (ii) revisit our problem and solve it differently, by integrating the volume and sharing of queries. In this thesis, we adopt the second scenario.

The database technology has a great experience in using advanced data structures such as graphs, hypergraphs, tree (for indexing), etc. These structures are used either to speed up the data access or as a support for algorithms dealing with large scale search space. We can cite the example of the problem of vertical partitioning in relational databases. Shamkant B. Navathe which is one of the pioneer who dealt with this problem, proposed a graphical algorithm with less computational complexity than his first algorithms [185, 185] that generates all meaningful fragments simultaneously [186]. The efforts spent Shamkant B. Navathe were concentrated to the transformation of the vertical partitioning problem to a graph.

A unified query plan has been already represented by traditional graph. The presence of node sharing of OLAP queries makes our problem similar to electronic circuit, where the hypergraph corresponding to a logical circuit directly maps gates to vertices and nets (connection between gates) to hyperedges [147]. Therefore, the hardness of our study is to make the analogy between these two worlds: unified query plan generation and logical circuits. Finally, our problem will be transported in the context of EDA.

By the means of this Chapter, we would like to share our collaboration with Mentor Graphics in terms of exploitation of their rich findings in terms of algorithms and tools.

This chapter begins by the presentation of the hypergraph theory (Section II) and their usages. Section III focuses on the analogy between the electronic design automation (EDA) and multi-query optimization problems. In the section IV, we detail our approach, in which we describe hypergraph representation and partitioning, and we show the mapping between the hypergraph and the unified query plan. The hardness study of our problem is given in Section V. Section VI describes the set of experiments that we conduct to validate our proposal. Finally, Section VII concludes the chapter.

II. Hypergraphs and their usages

Hypergraphs give more expressive than traditional graphs in modeling sets of objects. This characteristic allow the hypergraphs capturing any relationship between a group of objects, whereas graphs can only capture binary relationships [65, 204]. This specificity gives more flexibility in accurately formulating several important problems in combinatorial scientific computing [276]. Since 20 years, the hypergraph theory contributes in modeling and solving several problems in various domains such as location problems [159, 159], combinatorial problems [38] and so on [94, 160, 54, 97, 264].
The major particularity that hypergraphs offers is their ability in dividing the search space of a complex problem into several sub search spaces. Thus, hypergraph partitioning significantly reduces the complexity of the studied problems. The wide applicability of hypergraph theory has motivated the development of *fast partitioning tools*, some are standardized. We can cite the examples of as *hMeTiS*[1] [147, 148, 152], *PaToH*[2] [66], and *Mondriaan* [3][263]. In addition, we find also parallel partitioning tools such as *Parkway* [4] [250] and *Zoltan* [5][94].
This section presents a review of hypergraph theory and their applications to resolve combinatorial problems, with a special focus on database related problems.

II.1. Definitions

In this section, we present some fundamental definitions and underlying notions to understand the hypergraph and its partitioning algorithms.

Definition 8. ***Hypergraph:*** *A hypergraph $\mathcal{H}$ =($\mathcal{V}$,$\mathcal{E}$) is defined as a set $\mathcal{V}$ =$\{v_1, .., v_n\}$, of nodes (vertices) and a set $\mathcal{E}$ =$\{e_1, .., e_m\}$ of hyperedges, where every hyperedge e connects a non-empty subset of nodes.*

We said that the hyperedge $e \in \mathcal{E}$ connects the vertex $v \in \mathcal{V}$ if, and only if, $v \in e$.
A weight and costs can be respectively associated with vertices and hyperedges of a hypergraph. Let $w[v]$ and $c[e]$ denote respectively the weight of vertex v and the cost of hyperedge e.

Definition 9. ***Vertex degree:*** *The degree of vertex $v \in \mathcal{V}$, denoted by $d(v)$, is defined as the number of distinct hyperedges in $\mathcal{E}$ that connect v.*

Definition 10. ***Hyperedge size:*** *The hyperedge size (length) of the edge $e \in \mathcal{E}$ denoted by $|e|$, is defined as it cardinality.*

Definition 11. ***Hypergraph partition:*** *A partition of the hypergraph $\mathcal{H}$ ($\mathcal{V}$,$\mathcal{E}$), denoted by* Π *=$\{\mathcal{V}_1,\mathcal{V}_2,..,\mathcal{V}_k\}$, is a finite collection of subsets of $\mathcal{V}$, called parts, such that :*

[1]http://glaros.dtc.umn.edu/gkhome/metis/hmetis/overview
[2]http://bmi.osu.edu/~umit/software.html
[3]http://www.staff.science.uu.nl/~bisse101/Mondriaan/
[4]http://www.doc.ic.ac.uk/~at701/parkway/
[5]http://www.cs.sandia.gov/Zoltan/ug_html/ug_alg_phg.html

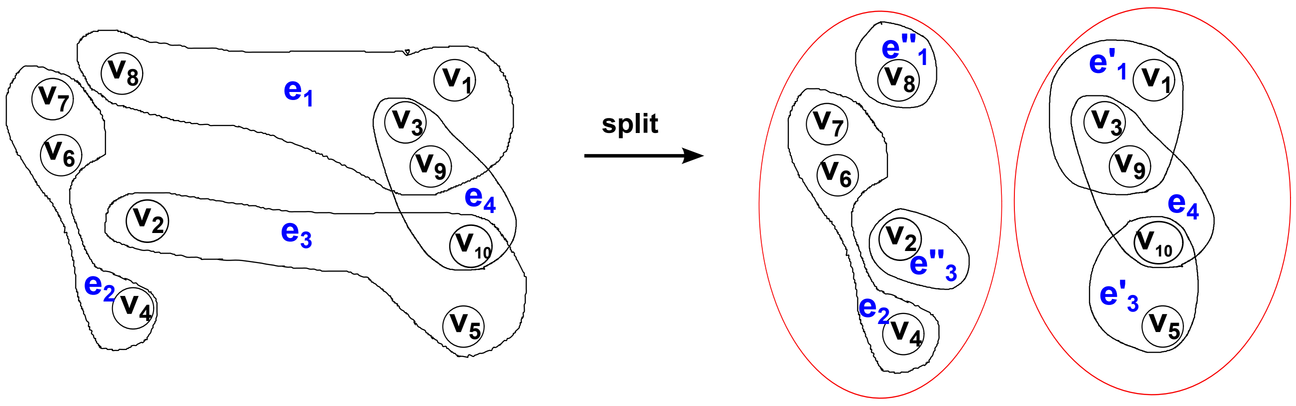


Fig. 3.1 – Example of cutting hyperedge in hypergraph partitioning

- *each subset $\mathcal{V}_i$ is a non-empty subset of vertices, i.e., $\mathcal{V}_i \neq \emptyset$, for $1 \leq i \geq k$;*
- *all subsets are pairwise disjoint, i.e., $\mathcal{V}_i \cap \mathcal{V}_j = \emptyset$, for $1 \leq i < j \geq k$;*
- *the union of all subsets is equal to $\mathcal{V}$. i.e., $\cup_{v_i} = \mathcal{V}$, for $1 \leq i < j \geq k$.*

When the number of subsets $|\Pi| = k$ and $k > 2$, the partition Π is called a k-way or a multi-way partition. Otherwise (when k = 2), we call Π as a hypergraph bi-partitioning (it called also bisections).

Definition 12. ***Connectivity of hyperedge:*** *In a partition Π of $\mathcal{H}$ $(\mathcal{V},\mathcal{E})$, we say that a hyperedge $e \in \mathcal{E}$ connects the part $\mathcal{P}_i$, if e connects at least one node from $\mathcal{V}_i$, and we say that e connect the part $\mathcal{P}_i$ or e connect the subset $\mathcal{V}_i$. We denote by $\mathcal{P}_e$ as the list of parts $\mathcal{V}_i$ connected by the hyperedge e. The connectivity of hyperedge e denoted by $|\mathcal{P}_e|$, is the number of distinct parts connected by e.*

Definition 13. ***Cut hyperedge:*** *A hyperedge $e \in \mathcal{E}$ is cut if it connects more than one part (i.e., $|\mathcal{P}_e| > 1$), and uncut otherwise (i.e., $|\mathcal{P}_e| = 1$).*

As shown in Figure 3.1, the bi-partitioning, has produced two cut of hyperedges (e_1 and e_3), which each one is split into two hyperedges. The first on e'_1 and e''_1, and the second on e'_3 and e''_3.

Definition 14. ***Cutset:*** *A cutset of a partition Π denoted by $C(\Pi)$, is the set of hyperedges that are cut by Π. So, a hyperedge in the $C(\Pi)$ has vertices in at least two distinct parts. The cardinality of the cutset is called the partition* ***cutsize****.*

II.2. Hypergraph partitioning algorithms

Definition 15. ***Hypergraph partitioning:*** *can be defined as the task of dividing a hypergraph into two or more partitions while minimizing the cutsize and maintaining a given balance criterion ε among partitions weights.*

II.2.1. Objectives of Hypergraph partitioning

Hypergraph partitioning is commonly used to model decomposition of problems, where the interconnection within the decomposition (partition) is minimized. The decomposition is evaluated using a cost that represents a quantitative measure of partition quality (integer or real value). This cost expresses the objective of the partitioning, which aims at minimizing or maximizing this cost. Several cost objectives used to partition the hypergraph. Most of them use the number of hyperedge cuts. Therefore, the partitioning aims at finding the partition with a minimum cut. For example, in VLSI placement, reducing hyperedges cut corresponds a minimizing needed wires between gates. In parallel computation the minimum cut corresponds at minimum of communication inter processors.

II.2.2. Formalization of Hypergraph partitioning

Formally, hypergraph partitioning is defined as:

- **Inputs:**
  - a hypergraph $\mathcal{H}$ ($\mathcal{V}$,$\mathcal{E}$);
  - an integer $k > 1$ that represents the suitable partitions;
  - a real value balance criterion $0 < \varepsilon < 1$.

- **Outputs:**
  a partition $\Pi = \{\mathcal{P}_1, .., \mathcal{P}_k\}$, with corresponding part weights f_w $(\mathcal{P}_i)$, $1 \leq i \leq k$.
- **Constraints:**
  f_w $(\mathcal{P}_i)<(1+\varepsilon)*W_{avg}$, for all $1 \leq i \leq k$, where W_{avg} is the average weight.
- **Objective:** minimization of the number of cut of hyperedge *cutsize*. The objective function may use other costs.

This partitioning has to ensure a balance between the weights of different result parts. This balance is considered as partitioning **constraints**. The weight of a given part is calculated as the sum of the weights of its vertices. Let f_w $(\mathcal{P}_i)$ be the weight of a part $\mathcal{P}_i$, which is defined as the sum of the weights of its vertices $\mathcal{V}_i$, as shown in the following equation:

$$f_w(\mathcal{P}_i) = \sum_{v \in \mathcal{V}_i} w[v] \tag{3.1}$$

To ensure the balance constraint, we need the average weight W_{avg} of parts and an allowed predetermined maximum imbalance ratio ε, where $0 < \varepsilon < 1$. The average weights W_{avg}, is defined as:

$$W_{avg} = (\sum_{v \in \mathcal{V}} w[v])/k \tag{3.2}$$

where k is the number of partitions.
A partition is said to be balanced, if each partition $\mathcal{P}_i$ satisfies the following criterion :

$$f_w(\mathcal{P}_i) \leq W_{avg} * (1 + \varepsilon); 1 \leq i \leq k \tag{3.3}$$

The cost of hypergraph partitioning Π can be defined by:

$$cost(\Pi) = \sum_{e \in C(\Pi)} c[e] \tag{3.4}$$

where $C(\Pi)$ and $c[e]$ represent respectively the cut set and the cost of the hyperedge e.

II.2.3. Methods of Hypergraph partitioning

Numerous approaches exist for solving the hypergraph partitioning problem that can be classified into four categories.

- **Exhaustive Enumeration** that produces all possible partitioning. At the end, it selects the partition with optimal cost defined by the designers. This method gives the optimal solution, but it has exponential complexity, since it requires an exploration of the search space [61].
- **Branch and Bound** is proposed to improve the above approach by implementing a depth left tree of partial partitioning. This tree allows having faster a sub-optimal solution by verifying the balance constraints in each node. It prunes the illegal solutions. Despite its performance, it may have an exponential complexity [61].
- **Fiduccia-Mattheyses** is proposed to scale partitioning algorithms [104], which is a linear heuristic that improves a partitioning using iterative passes wherein each vertex is moved exactly once. Passes are generally applied until little or no improvement remains. Initial solutions are often produced using a simple randomized algorithm.
- **Multilevel Fiduccia-Mattheyses Framework** consists of three main components: clustering, top-level partitioning and refinement (these steps will be detailed below).

Other methods exist for solving hypergraph partitioning problem such as *simulated annealing* and *Tabu search* [100], spectral techniques [10], network-flow driven algorithms [272], and incidental flow-solver [135, 136].

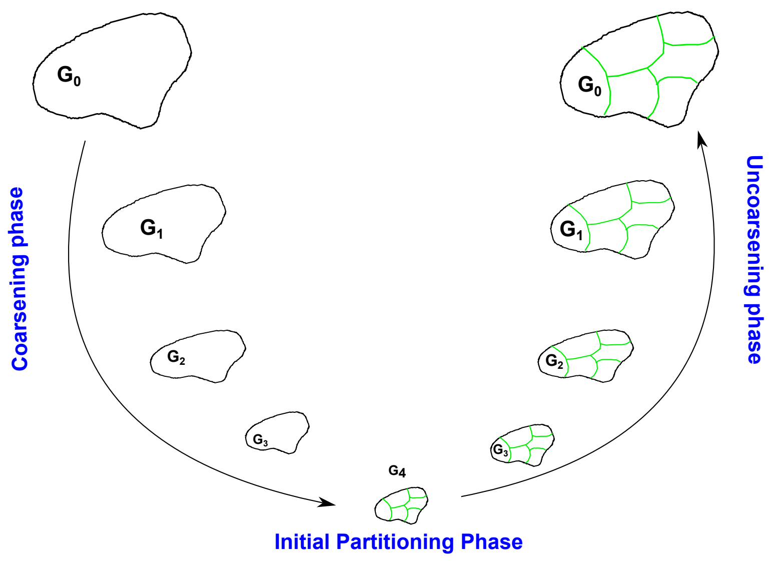


Fig. 3.2 *– Cut hyperedge splitting during recursive bisection*

The Multilevel hypergraph partitioning is the most used technique for hypergraph partitioning. It goes through three phases: *coarsening*, *initial partitioning*, and *uncoarsening*. In the first phase, a bottom-up multilevel clustering is successively applied on the original hypergraph, by adopting various heuristics, until the number of vertices reaches a predetermined threshold value in the coarsened graph. In the second phase, the coarsest graph is bi-partitioned using various bottom-up heuristics. In the third phase, the partition found in the second phase is successively projected back towards the original graph by refining the projected partitions on intermediate level uncoarser graphs using various top-down iterative improvement heuristics. The following sub-sections briefly summarize these three phases illustrated in Figure 3.2.

Coarsening Phase

In this phase, for given hypergraph $\mathcal{H} = (\mathcal{V},\mathcal{E})$, undergo the object of sequence of successive approximations, the hypergraph is coarsened into a sequence of smaller hypergraphs $\mathcal{H}_1 = (\mathcal{V}_1,\mathcal{E}_1)$, $\mathcal{H}_2 = (\mathcal{V}_2,\mathcal{E}_2)$ ... $\mathcal{H}_m = (\mathcal{V}_m,\mathcal{E}_m)$. satisfying $|\mathcal{V}_1| > |\mathcal{V}_2| > ... > |\mathcal{V}_m|$, and the number of vertices in the coarsest approximation $\mathcal{V}_m$ has some fixed upper bound that depends on the solicited number of parts in the partition (k).
In [152], Karypis et al., give some desirable characteristics that a coarsening algorithm should possess:

- a near-optimal partition of the coarsest hypergraph $\mathcal{H}_m$ should project to a near-optimal partition of $\mathcal{H}$.
- the successive coarser hypergraphs should have significantly fewer large hyperedges than the original hypergraph.

- the sum of the hyperedge weights in the successive coarser hypergraphs should decrease as quickly as possible

Coarsening Approaches

The early algorithms for coarsening graph [58, 130] use a purely random edge matching scheme. Unmatched vertices are only allowed to match with other unmatched vertices, resulting in vertex clusters of size two. In [15], *Simon* and *Barnard* coarsen the graph by first computing a maximal independent set of vertices $\mathcal{V}' \subseteq \mathcal{V}_i$, such that for any two vertices u, $v \in \mathcal{V}'$, there is no hyperedge in $\mathcal{E}$ that connects u and v. Then constructing the coarse vertices by matching vertices in the maximal independent set with their neighbors.
Toyonaga et al. [248] sort the hyperedges in non-decreasing order by cardinality and then traverse the sorted list of hyperedges, with vertices of a hyperedge forming a cluster if they have not been matched previously as part of another hyperedge.
Karypis et al. [147, 148] sort the hyperedges in a non-increasing order of weight and then hyperedges of the same weight are sorted in a nondecreasing order of cardinality. The sorted set of hyperedges is then traversed and vertices of a hyperedge form a cluster if they have not been matched previously as part of another hyperedge.

Initial Partitioning Phase

The goal in this phase is to find a bipartition on the coarsest hypergraph $\mathcal{H}_m$, that satisfies the partition balance constraint and optimizes the objective function.
Effectively, any heuristic optimization algorithm that can be applied to hypergraph partitioning may be used to compute the initial partition. For example, Hauck and Borriello [127] evaluate a number of simple methods for generating a starting feasible partition, prior to using a heuristic partitioning algorithm. Random partition creation by randomly chosen vertices is proposed to improve the previous algorithms in terms of the quality of the generated partitions and the runtime. Similar initial partition generation methods are also used in [147, 152].
In [60], the authors have introduced a relaxation balance constraint of random computation by adding deterministic computation. Generally, the quality of these algorithms is sensitive to the choice of the initial random vertex.

Uncoarsening Phase

During this phase, a partition of the coarsest hypergraph is projected through each successive finer hypergraph, where at each level i (for i $= m$, $m-1$,..., 1), bipartition Π_i found on $\mathcal{H}_i$ is projected back to a bipartition Π_{i-1} on $\mathcal{H}_{i-1}$. A simple algorithm for projecting a partition which runs in $\mathcal{O}$ (n) time proceeds as follows:
The vertices of a hypergraph $\mathcal{H}_i$ ($\mathcal{H}_i$,$\mathcal{E}_i$) are traversed, and for each vertex $v \in \mathcal{V}_i$, the part corresponding to the coarse vertex $g(v) \in \mathcal{V}_{i+1}$ is noted from Π_{i+1} and v is assigned to the same

part in Π_i. Typically, iterative improvement algorithms based on the KL/FM framework are used. Although [214] uses a *Tabu search-based* algorithm.

II.3. Applications of hypergraph theory

Recently, the hypergraph theory [148] allowed a hyperedge to connect an arbitrary number of vertices instead of two in regular graphs. Therefore, it provides better understanding of large range of real systems. The hypergraph theory is very useful in many domains like **(1)** in *chemistry* search they are used to giving more convenient description of molecular structures [160]. **(2)** In telecommunication, where they are used to modeling cellular mobile communication systems to define the groups of cells that cannot use simultaneously the same channel under predefine distance [159]. **(3)** In image processing in which they are used to segmenting images [54, 97]. **(4)** Other domains like recommendation [41, 226], analyzing public health [264], etc. In the following, we detail some practical applications of hypergraph theory.

- **Data mining implementation:** hypergraph has been exploited by researchers in data mining, where data are clustered in several groups, such that the intra-cluster similarity is maximized, while inter-cluster similarity is minimized [122, 42, 129]. The set of vertices and hyperedges of the hypergraph corresponds respectively to the set of data items and the set of relations between them.

- **Database schemes modeling:** a database can be viewed as a set of attributes and set of relations between these attributes. This property has motivated some works to introduce hypergraph theory to model relational database schemes [103]. The vertices of hypergraph is the set of attributes, and the set of hyperedges is the set of relations between these attributes. Other applications have been discussed in the previous chapters.

- **VLSI computer-aided design:** the application of hypergraph partitioning within VLSI design has been well-addressed in literature [11, 142, 79]. The hypergraph partitioning problem is used to facilitate the task of designing modern integrated circuits that have a very large number of components. A hypergraph is used to represent the connectivity information from the circuit specification. Each vertex in the hypergraph represents a gate in the circuit and each hyperedge represents a connection inter-gates [11]. Hypergraph partitioning allows dividing a circuit specification into clusters of components, such that the cluster interconnect is minimized. Each component can then be assembled independently, speeding up the design and the integration processes.

- **Social network analysis:** there is a tendency to use hypergraph theory to analyze social networks. In this context, the vertices and hyperedges of a hypergraph represent respectively the type of actors (e.g., peoples, groups, etc.) and the relationships between them. The corresponding hypergraphs are used for many proposes such as: forensics analysis to solve crimes (e.g., identifying the regions of twitter [109]) and for identifying topological features to understand tagged networks [283].

II.4. Discussion

In many applications of hypergraph theory, the size of input grows continually and rapidly. For example, the number of transistors in VLSI design problem continues to grow exponentially in which today tens of millions gates and in the near-future hundreds of millions are needed to design logical circuits. Same situation concerns the social network medias, where the number of worldwide users is expected to reach some 2.95 billion by 2020, around a third of Earth's entire population[6]. The success story of hypergraphs, where they have has proved their motivates us to use them in our context of big queries.

III. Analogy between UQP and EDA

In this section, we present the analogy between MQO problem and electronic design automation (EDA). This analogy is the motivation to brow EDA techniques and tools to be used in our problems.

III.1. Analogy between UQP and Electronic circuit

An UQP can be represented using an directed acyclic graph those nodes (vertices), represent the query operators which are binary and unary. The edges describe the data flow between operators, where loading data start from the leaf nodes and each intermediate node passes its results computing to their connected nodes, until root nodes (that describe the final results). Similarly, a logical circuit can be represented by directed acyclic graph those nodes represent theirs gates (ports) that can be associated to many inputs/outputs. An edge describes the data flux between gates that defines the sense of transforming data from one gate to another. The leaf nodes capture data from external source, whereas intermediate nodes modify the captured data and transform them to the connected nodes. The root nodes contain the results of electronic circuit function. Hence, an UQP can be considered as electronic circuit with only unary and/or binary ports.

Example

Consider the star schema benchmark *SSB* [7] [194]. The $\mathcal{DW}$ contains a fact table *Lineorder* and four dimension tables: *Customer*, *Supplier*, *Part* and *Dates*. This schema is queried by a set of 30 queries of SSB benchmark. Figure 3.3 describes a generated UQP of those 30 queries.

[6]`https://www.statista.com/topics/1164/social-networks/`

[7]`http://www.cs.umb.edu/poneil/StarSchemaB.pdf`

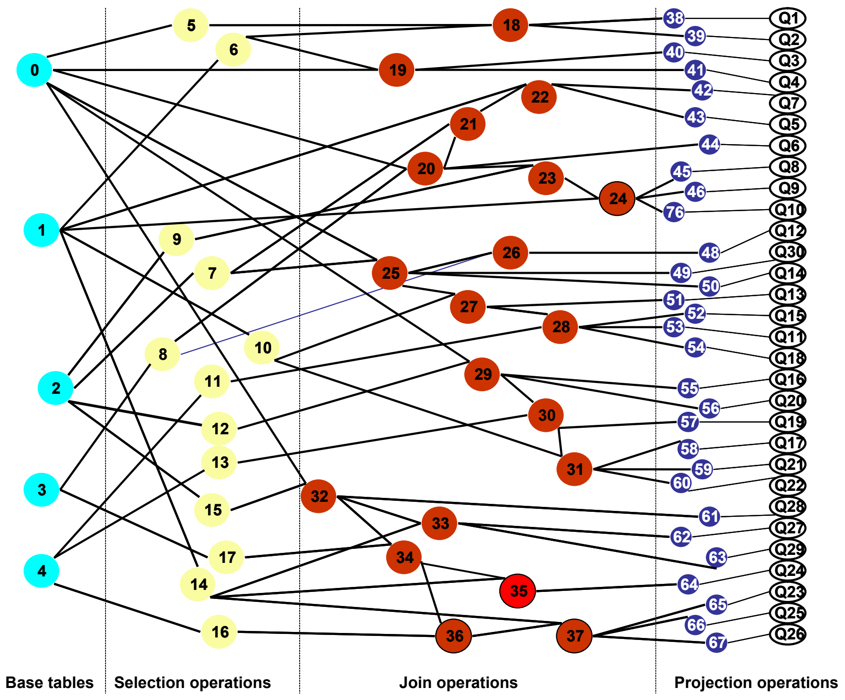


Fig. 3.3 – An example of UQP of 30 queries

Note that the UQP contains four levels of nodes (selection, join, projection and aggregation). Figure 3.4 shows an UQP designed as an electronic circuit by replacing its intermediate results by electronics ports (AND, OR, XOR).

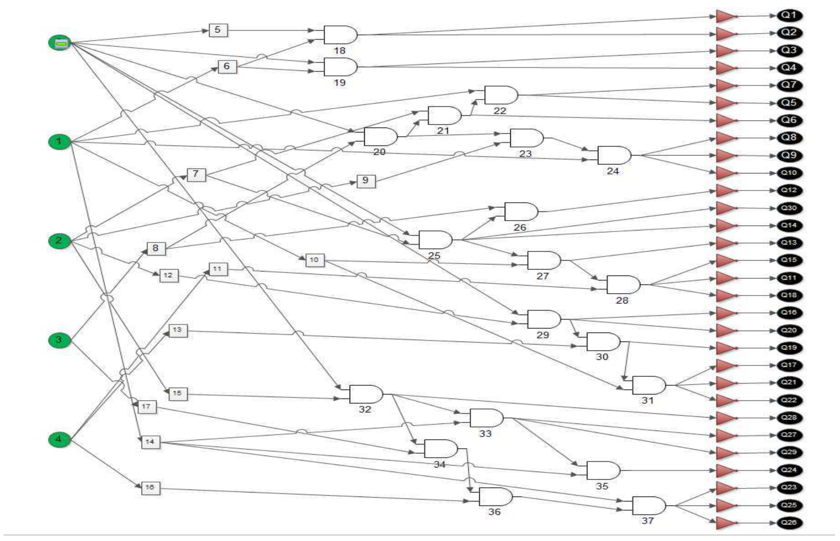
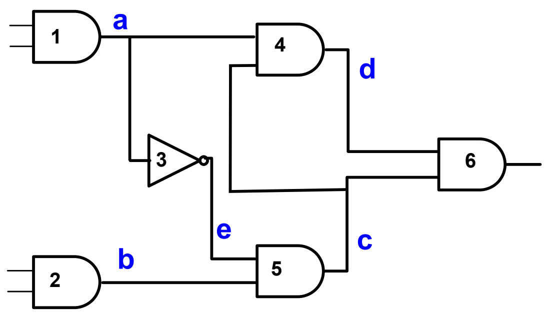


Fig. 3.4 – Electronic circuit corresponding to the UQP

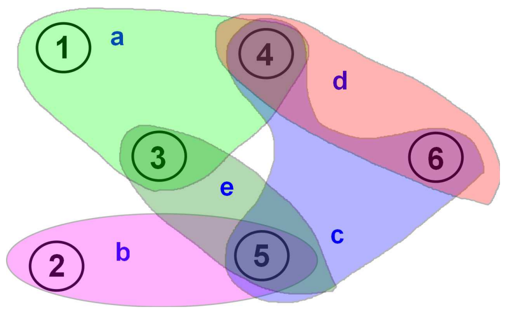
This analogy allows us borrowing optimization techniques and tools defined in electronic circuits' domain to handle very large UQP.

III.2. Hypergraph as DS in VLSI circuits

In very large scale integration (VLSI) process, it is important to be able to split (or partition) the circuit in many clusters, where the interconnection between ports is important inside a cluster and minimal connections between clusters. It has many applications, like logic and physical synthesis, circuit testing, mapping, floor plan, place and root, timing analysis and simulation. Particularly, the placement steps uses partitioning to situate a circuit of hundreds



(a) Circuit



(b) Hypergraph

Fig. 3.5 *– Hypergraph representation of the circuit*

of thousands or millions of gates with respect of the specifications of the blocks (constrained by minimal timing and area). To address this problem, a circuit can be represented by a hypergraph: gates of the circuits are the vertices of the hypergraph and the hyperedges are considered as the connections between ports. Figure 3.5b gives a representation of the circuit shown in the Figure 3.5a. The hypergraph will be partitioned in many clusters, where each cluster will be designed and tested independently. Finally, a global test and merging of all clusters is conducted.
By analogy the representation of query interaction by a hypergraph, allows us to use partitioning algorithms to group queries in many sets, in which queries inside each set share the maximum of intermediate results and have minimum interaction with the others queries.

IV. Hypergraph as a solution of scalability

The successful analogy inspires us to use similar techniques as those used in EDA. More precisely, we model a workload by a hypergraph, and we partition it to facilitate the exploration of the search space by dividing the global hypergraph in several disjoint subsets. These subsets will be transformed in many local UQP (each subset of queries has an UQP), which will be

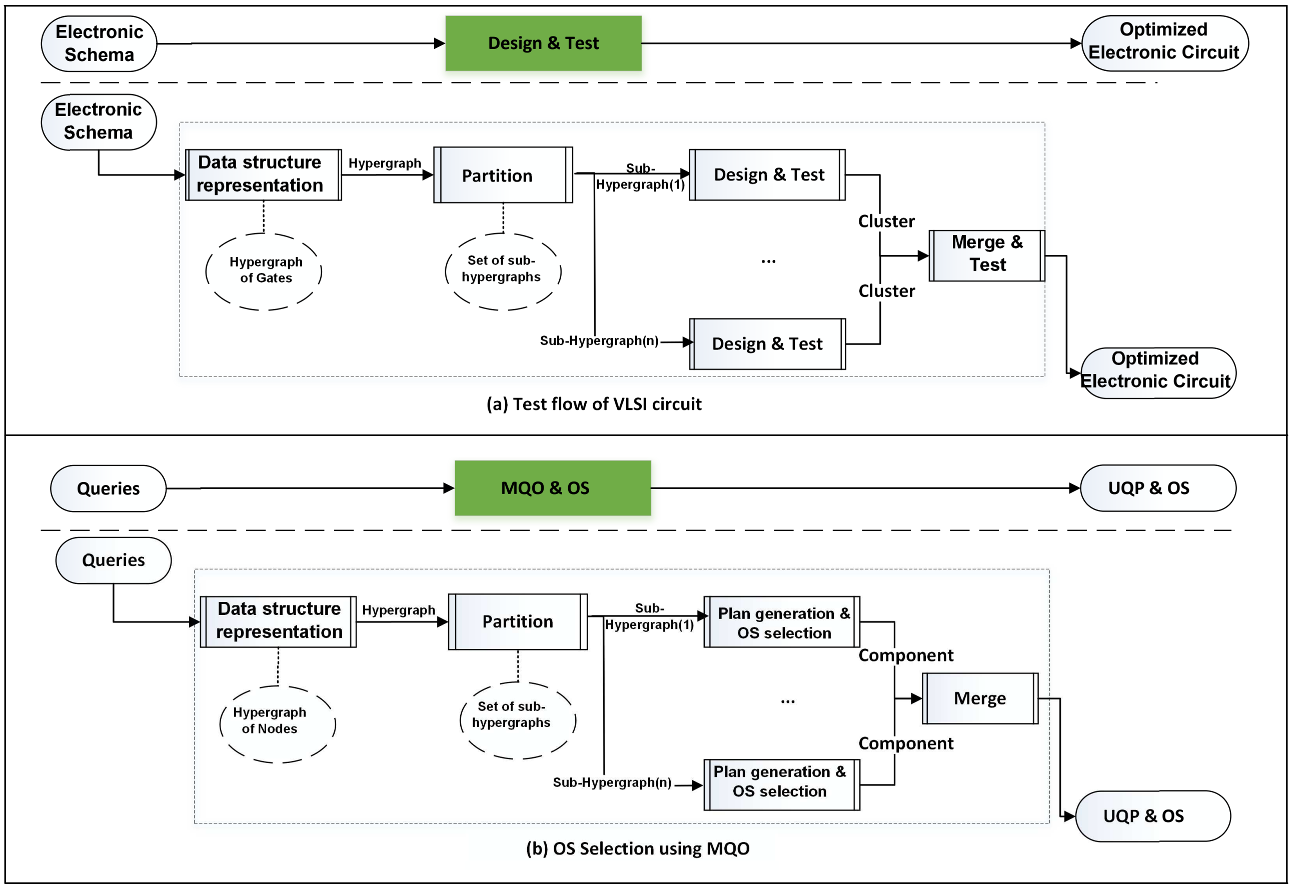


Fig. 3.6 – Analogy between VLSI circuit and MVPP generation approaches

ultimately merged to obtain the global UQP, for all queries. We detail below, the steps of our 5-steps methodology (Figure 3.7).

- **Step 1:** parses each SQL query of our workload to identify the logical operations (nodes). Since, we are assuming SPJ (Select-Project-Join) queries, three types of nodes are distinguished: **(a)** selection nodes are determined by the selection predicate. Each predicate has a selectivity factor; **(b)** join nodes are determined by a join predicate and two selection nodes and **(c)** projection nodes are determined by a predicate which correspond to a set of columns projected by the query.

- **Step 2:** models the join nodes by a hypergraph, where the vertices represents the set of join nodes and the hyperedge set represents the workload of queries[8].

[8]In our work, we focus on only by join nodes that are the costly nodes and the other operations are pushed as far down the logical query tree as possible.

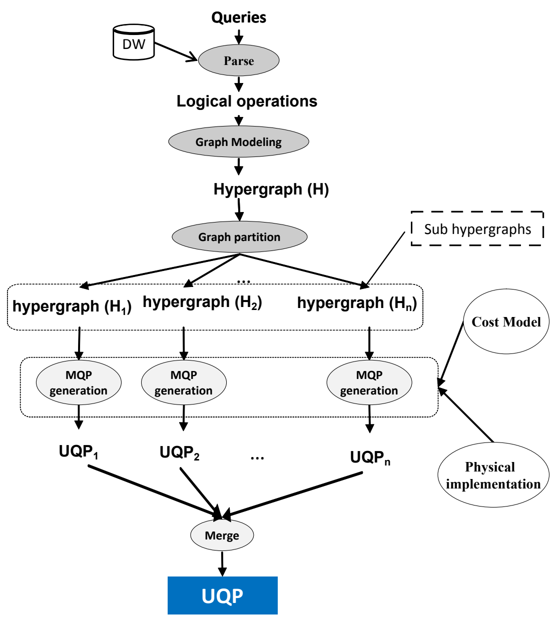


Fig. 3.7 – UQP generation approach

- **Step 3:** generates connected components using hypergraph partitioning algorithms. The result is a set of disjoint components of queries. Each component is represented by a sub-hypergraph (SH_i).
- **Step 4:** transforms each sub-hypergraph into an oriented graph using a mathematical cost model and implementation algorithms to order the nodes. This step is crucial, since we can plug any $\mathcal{OS}$ knowledge consumer of unified graph such as materialized views, horizontal partitioning, etc.
- **Step 5:** merges the graphs resulting from the previous transformation to generate the global unified query processing (UQP).

IV.1. Query workload representation

IV.1.1. Parsing query

The parser takes as input a SQL query and converts it into a parsed tree. This process is done by:

- a syntactic analysis to give the basic grammar elements of the text query in the form of a parse tree;
- a semantic checking of the parsed tree by verifying the used relations and attributes, query clauses, etc.;

- a verification whether views are involved in the current query or not.

Syntax analysis

The syntax analysis allows verifying the consistency of parse tree nodes. The leaves nodes of the parse tree represent the atomic elements. They are the lexical elements of the query such as keywords (e.g., *SELECT*, *WHERE*, *FROM*), names of relations and attributes, constants, logical and arithmetic operators, etc. The other nodes of the parsed tree represent a composite element formed by many atomic elements like *Condition*, other sub query. In Figure 3.9, the composite elements are reported by elements between "<" and ">". Figure 3.9 shows the parsed tree of SQL query presented in the Figure3.8.

```
SELECT studentName
FROM Studients
WHERE idStudient IN (
        SELECT idStud
        FROM FollowCours
        WHERE speciality='CS');
```

***Fig. 3.8** – Example of a text written query*

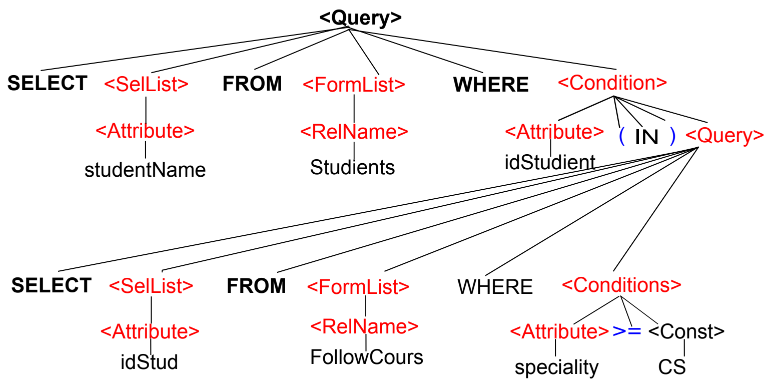


***Fig. 3.9** – The parse tree of the previous query (Figure 3.8)*

Semantic checking of the parsed tree

The semantic parse tree checking is an intermediate step aiming at verifying the syntactical and the semantic rules of the query. The checking process follows these main tasks:

- checking whether every relation, attribute, or view mentioned in the *FROM-clause* belongs to the schema of database or data warehouse (using the meta-model).

- checking whether every attribute mentioned in the *SELECT/WHERE-clause* belongs to the relations appeared in the *FROM-clause*.
- checking whether attributes' types are respected in the *Condition-clause*.

Views checking

In this step, each virtual view in the query parse tree will be replaced by a sub-parse tree that represents the sub-query constructing this view. Note that in the case of materialized views, there is no substitution, because these latter are considered as base tables (relations). Figure 3.10 shows the replacement of a virtual view by its sub-parse tree.

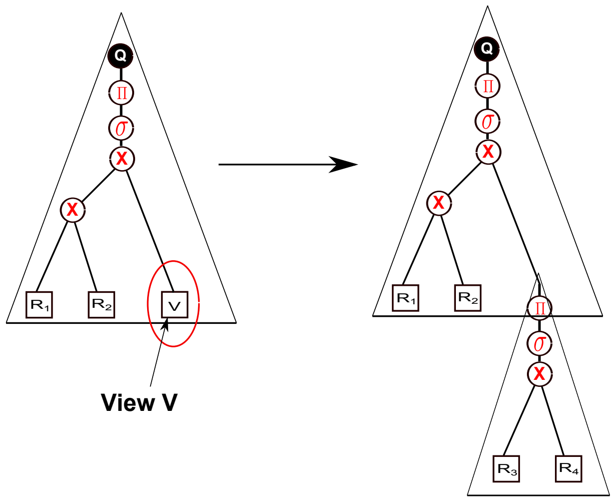


Fig. 3.10 *– Virtual view substitution.*

IV.1.2. Logical query plan selection

The logical query optimization is founded on the algebraic equivalences. In fact, many equivalent expressions can be generated for a query [44] (cf. Chapter I) A query is composed of different operations which follow algebraic laws (commutativity, associativity, and distributivity), and can be applied in both directions: from left to right, and from right to left. So, a huge space of equivalent expressions can be derived for one query. Finding a better plan, during the logical step is not possible, since many implementation details required for the cost estimation may be missed and eventually further added to the plan (during the physical phase).
The generation of the logical plan of a query presented as a parsed tree, goes through two principal steps: (1) transforming this tree to a logical query plan, and (2) improving the logical query plan using algebraic laws, and (3) selecting an appropriate logical plan. Figure 3.11 summarizes the principal steps of the logical query plan generation.

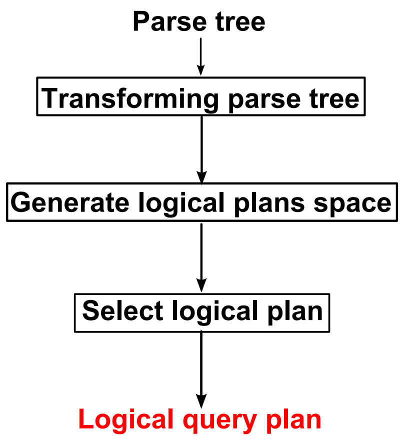


Fig. 3.11 – Logical plan generation steps

Parse tree translation

Transforming the parsed tree into more preferred logical query plans is performed through two steps: **(i)** the nodes of the parsed tree are replaced by one or several operators of relational algebra. **(ii)** the query optimizer applies the algebraic laws on the resulting query expression to generate an expression more appropriate to be physically processed thereafter. The translation applies the following rules:

- Relations are loaded from <*FromList*> of the parsed tree;
- Selections σ_C correspond to every C captured from <*conditions*> of the parsed tree;
- Projections Π_L correspond to each attribute present in the list L captured from <*SelList*>.
- the initial logical plan corresponds to the cross product of the relations, two by two, and second a selection σ_C, and finally a project Π_L. Figure 3.12 gives an example of an initial logical query plan associated to the query in Figure 3.8.
- If the condition represents a sub-query, then it will be replaced by an expression of relation algebra following the previous steps.

IV.2. Hypergraph generation

An hypergraph $\mathcal{H}$ is a set of vertices $\mathcal{V}$ and a set of hyperedges $\mathcal{E}$. In our case, $\mathcal{V}$ represents a set of join nodes, such that for each vertex $v_i \in \mathcal{V}$, corresponds a join node n_j. The same way, $\mathcal{E}$ represents the workload of queries $\mathcal{Q}$, such that for each hyperedge $e_i \in \mathcal{E}$ corresponds a query q_j. A hyperedge e_i connecting a set of vertices corresponds to a join nodes that participate on the execution of the query q_j. As shown in Figure 3.13, an example of a join hypergraph is

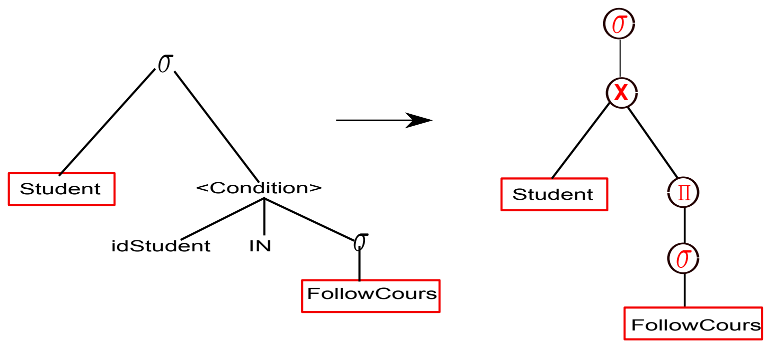


Fig. 3.12 – Example of first translating parse tree

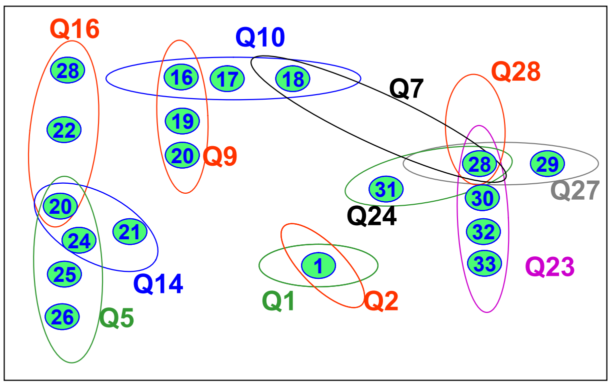


Fig. 3.13 – An example of join hypergraph

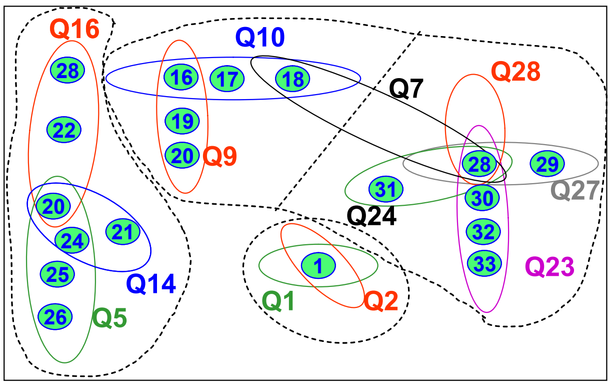


Fig. 3.14 – Result of hypergraph partitioning

given, where the hyperedge e_{23} corresponds to query q_{23} and connects the join nodes: n_{28}, n_{30}, n_{32} and n_{33}. The hypere-dge e_2 corresponds to query q_2 and connects one join node n_1.
The set of join nodes will be partitioned into several disjoint sub-sets, called *connected components*. Each component can be processed independently to generate a local UQP by ordering the nodes.
Table 3.1 summarizes the mapping between the *graph* vision and the *query* vision.

IV.3. Hypergraph partitioning

As we explain in the section III, the hypergraph partitioning allows us to group queries in many sets, in which queries inside each set share the maximum of intermediate results and have minimum interaction with the others queries. To partition our hypergraph, we adapt an existing algorithm derived from graph theory to aggregate the join nodes into small connected components. The partition process is applied on initial hypergraph $\mathcal{H}$ $(\mathcal{V},\mathcal{E})$ and the result of hypergraph partitioning is k sub-hypergraphs that are a hypergraphs $\mathcal{H}_i$ $(\mathcal{V}_i,\mathcal{E}_i)$, where $|\mathcal{E}_i| \leq M$, for all $1 \leq i \leq k$. Our partitioning algorithm has the following steps:

hypergraph Vision	Query vision
$V = \{v_i\}$ set of vertices	$\{n_j\}$ set of join nodes
Hyperedge e_j	Query q_j
Section (sub-hypergraph)	Connect component
Hypergraph HA	Workloads of queries $\mathcal{Q}$
Oriented graph	Processing Plans

***Table 3.1** – Analogy Graph – Query.*

1. Firstly, we adapt *the code of the multilevel hypergraph partitioning* (*hMeTiS*) to split our hypergraph into several partitions such as no cut of hyperedges. *hMeTiS* is a free software library developed by the *Karypis laboratory*. This software allows parallel and serial partitioning. As we said before, it divides a hypergraph into k partitions (new hypergraphs) such that the number of hyperedges cut is minimal [148, 151]. In our context, the exact number of partitions to construct is unknown. In the same time, we want to get all possible disjoint partitions (connected components). To do so, we adapt the original algorithm to our problem by bi-partitions until no partition can be repartitioned without cutting hyperedges. More precisely, the algorithm behaves as follows: (i) the set of vertices will be divided if and only if the number of hyperedges cutting is null. (ii) Each bisection result of the hypergraph partitioning will be divided in the same way until no more divisible hypergraph is found.

2. Secondly, we use *hMeTiS* programs to partition each sub-hypergraph $\mathcal{H}_i$ ($\mathcal{V}_i$,$\mathcal{E}_i$), such as $|\mathcal{E}_i| \geq M$. The sub-hypergraph $\mathcal{H}_i$ will be partitioned into k' partitions such as $k' = (|\mathcal{E}_i| / M) + 1$.

Figure 3.14 presents the join hypergraph partitioning of the hypergraph shown in the Figure 3.13, where three connected components corresponding to three new hypergraphs are obtained.

IV.4. Transforming hypergraph to UQP

After the partitioning process of the hypergraph into several small sub-hypergraps, the generation of UQP becomes a simple transformation of each sub-hypergraph into an oriented graph and finally merging of resulting graphs. In this thesis, we assume that the queries of our workload are represented only left-deep plan, so for a join unified query plan will have many connected components. Each component contains a query plan that *shared at least one join node*. The first join node, called *pivot node*(*pivot*), which is shared by all queries of its component. The pivot has a *direct impact on the other nodes of its component*. So, the quality of component depends to the pivot nodes. Hence, the generation of $\mathcal{OS}$-oriented UQP becomes at finding the *adequate pivot* node that defines the components of the $\mathcal{OS}$-oriented UQP.

To generate the local UQP (for one sub-hypergraph), the join hypergraph must be transformed into oriented join graphs $\mathcal{G}$. Adding an arc to a graph corresponds to establishing an order between two join nodes. We start by the pivot node which is preferred for the $\mathcal{OS}$ that is identified using a function translating the knowledge of $\mathcal{OS}$ (see Chapter 4). The pivot node will be added to the oriented graph ($\mathcal{G}$) and deleted from the hypergraph. This operation is repeated until all nodes are added to the graph.

Algorithm 1: transformHyperGraph(Hypergraph $\mathcal{H}$)

```
1: while $\mathcal{V}$ not empty do
2:   $pivot \leftarrow findPivot$ ($\mathcal{H}$); {Algorithm 2}
3:   if $pivot \in$ all ($e_i \in \mathcal{H}$) then
4:     $addVertexToGraph$ ($pivot$); {add $pivot$ to $\mathcal{G}$ }
5:     $deleteVertex$ ($pivot$); {delete $pivot$ from $\mathcal{V}$ of $\mathcal{H}$ }
6:     for all $e_i \in \mathcal{H}$ do
7:       if $|e_i| = 0$ then
8:         $deleteHyperEdge$ ($e_i$); {delete all hyperedges which have no a vertex}
9:       end if
10:    end for
11:  else
12:    $partitionPivot$ ($pivot$, $H_1$, $H_2$); {Algorithm 3}
13:    $addVertexToGraph$ ($pivot$);
14:    $transformHyperGraph$ ($H_1$); {re-transformation}
15:    $transformHyperGraph$ ($H_2$); {re-transformation}
16:  end if
17: end while
```

Algorithm 1 describes the transformation steps of a hypergraph into an oriented graph.
Transforming a hypergraph into an oriented graph UQP is a two-part process. On one hand, we select a pivot node from the hypergraph to be removed, while on the other hand the corresponding join node is added to the UQP. This process is repeated until there are no more vertices in the hypergraph is found. Hence, the transformation has three steps: **(1)** choosing the *pivot node* (*pivot*), depending on the target $\mathcal{OS}$. **(2)** transforming the *pivot node* from the hypergraph to the oriented graph. **(3)** removing the pivot node from the hypergraph.
Figure 3.15 shows the transformation steps of a hypergraph into an oriented graph. In which, we have as input the hypergraph $\mathcal{H}$ ($\mathcal{V}$,$\mathcal{E}$)=({n_{20}, n_{21}, n_{22}, n_{24}, n_{25}, n_{26}, n_{28} },{Q_5,Q_{14},Q_{16} }). In the first step, the node n_{20} is chosen as the pivot node, which is determined by two selections S_1 and S_2. The transformation starts by inserting the first join node corresponding to n_{20} into the oriented graph (basically in the first step, the oriented graph don't contain any nodes), and deleting the n_{20} from the hypergraph $\mathcal{H}$. In which produces two other sub-hypergraphs : H_1 (V_1,E_1)=({ n_{21}, n_{24}, n_{25}, n_{26} },{Q_5,Q_{14} }) and H_2 (V_2,E_2)=({ n_{22}, n_{28} },{Q_{16} }). In the

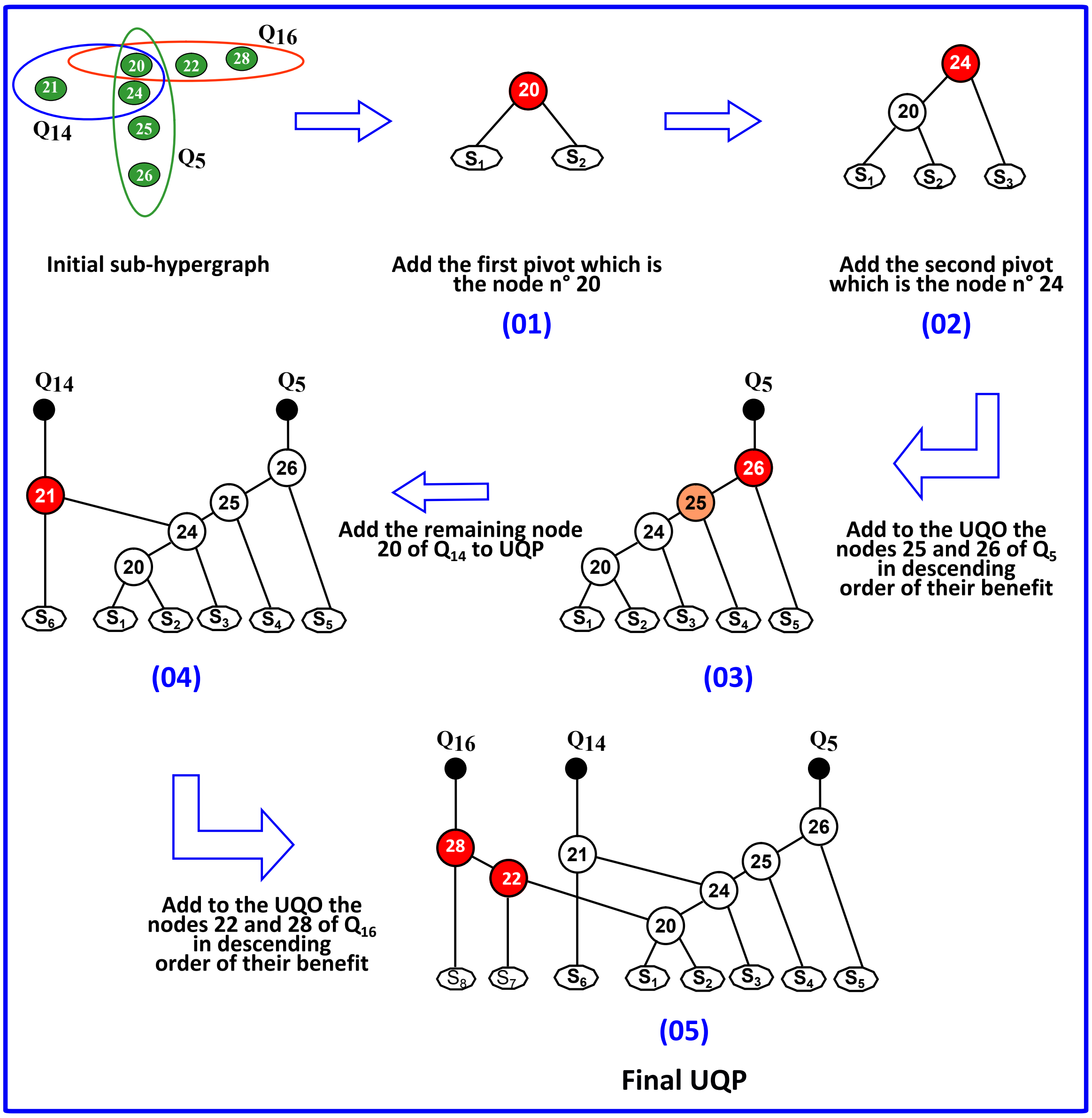

Q16
20
22
28
21
24
Q14
25
Q5
26
Initial sub-hypergraph
Add the first pivot which is the node n° 20
(01)
Add the second pivot which is the node n° 24
(02)
Add to the UQO the nodes 25 and 26 of Q5 in descending order of their benefit
(03)
Add the remaining node 20 of Q14 to UQP
(04)
Add to the UQO the nodes 22 and 28 of Q16 in descending order of their benefit
(05)
Final UQP

Fig. 3.15 – Transformation steps of a join hypergraph to an oriented graph

second step, the n_{24} (determined by two selection S_1 and S_3) is chosen as pivot node. The pivot node will be added into the graph after the node n_{20} (so, the right predecessor becomes n_{20} instead S_1). The removing the n_{24} from H_1 will generate two other sub-hypergraphs H_{1_1} $=(\{n_{25}, n_{26}\},\{Q_5\})$ and $H_{1_2}=(\{n_{21}\},\{Q_{14}\})$. These operations (choose pivot node, inset node into oriented graph and removing node from hypergraph), will be repeated until no nodes into any sub-hypergraph, and as consequence we obtain the final oriented graph (UQP).

IV.4.1. Finding the pivot node

The choice of *pivot node* from the hypergraph at a given point is made through a *benefit* function which takes into account the characteristics of the specific problem. The function computes the benefit relative to the targeted $\mathcal{OS}$ for each remaining node of the section and adds the node with maximum benefit. For more details on the benefit function, see Chapter 4.

Algorithm 2: *findPivot* (Hypergraph $\mathcal{H}$)

```
1: $benefitemax \leftarrow 0$;
2: for all $v_i \in \mathcal{V}$ do
3:    $nbr \leftarrow nbrUse\ (v_i)$; {$nbr$: number of hyperedges that connect $v_i$ }
4:    $benefit \leftarrow getBenefit(v_i)$ {calculate the benefit of $v_i$ following the $\mathcal{OS}$ }
5:    if $benefit > benefitemax$ then
6:       $benefitemax \leftarrow benefit$;
7:       $pivot \leftarrow v_i$;
8:    end if
9: end for
10: return $pivot$
```

Algorithm 2 allows finding the vertex (node) which is the pivot in the hypergraph $\mathcal{H}$. This pivot corresponds to the node which has the best possible benefit of intermediate results reuse. The benefit is the number of *reuse* multiplied by the processing cost minus the cost processing to generate the intermediate result.

IV.4.2. Remove the pivot node from hypergraph

The removing of pivot node impacts on the hypergraph. A hypergraph is the disjoint union of *connected section*. Two hyperedges belong to the same connected section if there is a connected path of hyperedges between them. Let $\mathcal{S}$ be a hypergraph and $v_i \in \mathcal{S}$ be a vertex being removed. When a vertex is removed from a section, the remaining nodes of that section are affected in different ways depending on the following case:
Let vertex v_j and v_j two vertices the same hypergraph (section) the different cases when removing a vertex are:

- **Case 1**: v_i and v_j belongs to the same hyperedge and there is no hyperedge to which v_j belongs and not v_i. The remove of v_i, v_j is unchanged.
- **Case 2**: v_i and v_j belongs to the same hyperedge and there exists an hyperedge to which v_j belongs and not v_i. The remove of v_i, v_j is duplicated, one unchanged and one renamed with v'_j
- **Case 3**: v_i and v_j do not belong to the same hyperedge. The remove of v_i, v_j is unchanged.

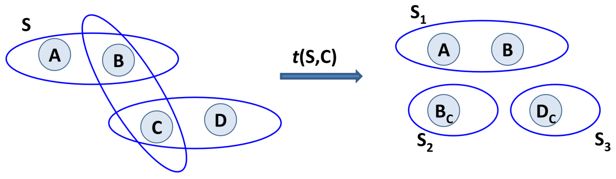


Fig. 3.16 – Example of transformation

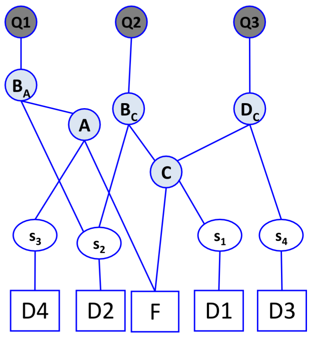


Fig. 3.17 – A resulting UQP

We denote by $t(\mathcal{S},v)$ the transformation that removes the vertex,v from section $\mathcal{S}$. In Figure 3.16, we apply $t(\mathcal{S},v)$, which means we remove the node C from $\mathcal{S}$. If C is the vertex being removed, then D is type 1 (there is an hyperedge to which both C and D belong, but no hyperedge to which D belongs but not C), B is type 2 (there is a hyperedge to which both C and B belong, and there is a hyperedge to which B belongs but not C), and A is type 3 (there is no hyperedge to which A and C belong). Figure 3.16 shows the effect of $t(\mathcal{S},C)$ on section $\mathcal{S}$, the result being three new sections $\mathcal{S}_1$, $\mathcal{S}_2$,$\mathcal{S}_3$. Figure 3.17 shows the result of transformations $t(\mathcal{S},C)$,$t(\mathcal{S}_1,B_A)$,$t(\mathcal{S}_2,B_C$, $t(\mathcal{S}_3,D_C)$ on the example of Figure 3.16.
So, removing the *pivot node* from the hypergraph can generate a partitioning of the hypergraph into two disjoint hypergraphs $\mathcal{H}_1$ and $\mathcal{H}_2$ by Algorithm 3. H_1 includes all hyperedges that including the *pivot* and $\mathcal{H}_2$ has the other hyperedges. Both hypergraphs are transformed the same way into oriented graphs$\mathcal{G}$.

Algorithm 3: *partitionPivot* (Vertex *pivot*, Hypergraph $\mathcal{H}_1$, Hypergraph $\mathcal{H}_1$)

```
1:  for all $e_i \in \mathcal{H}$ do
2:     if $pivot \in e_i$ then
3:        addToGraph ($e_i$, $\mathcal{H}_1$); {add the hyperedge $e_i$ to $\mathcal{H}_1$ }
4:     else
5:        addToGraph ($e_i$, $\mathcal{H}_2$); {add the hyperedge $e_i$ to $\mathcal{H}_2$ }
6:     end if
7:  end for
8:  for all $v_i \in \mathcal{H}_2.\mathcal{V}$ do
9:     if $v_i \in \mathcal{H}_1.\mathcal{V}$ then
10:       putNewID ($v_i$, $\mathcal{H}_2$); {new ID for duplicate node }
11:    end if
12: end for
```

Algorithm 3 allows partitioning a hypergraph into two disjoint hypergraphs using a node as a *pivot*. The first hypergraph contains the hyperedges that use the *pivot*, and the second

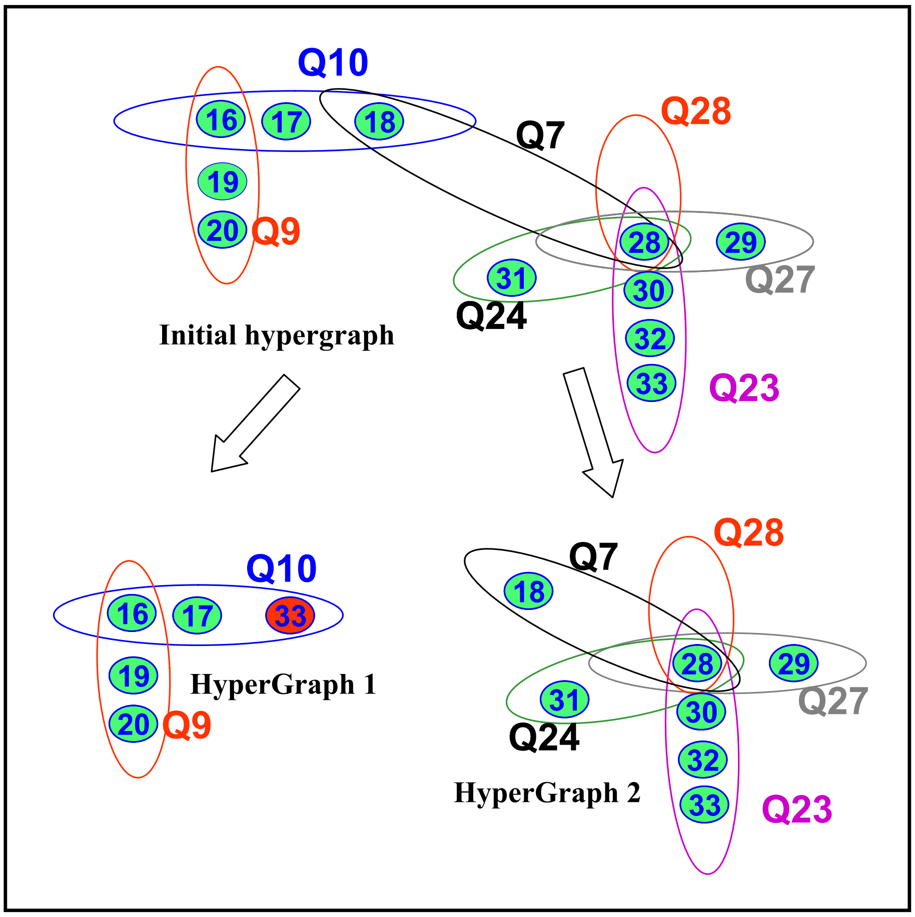


***Fig. 3.18** – An Example of hypergraph partitioning with node as a pivot*

hypergraph contains the other hyperedges.

Figure 3.18 shows an example of the result of hypergraph partitioning with the *pivot* n_{28}. We note that the node n_{18} belong the case 2, which needs to be duplicated into the node n_{33} in the second hypergraph, when removing n_{28}. the nodes n_{29}, n_{30}, n_{31}, n_{32} and n_{33}, belong in the case 1. the nodes n_{17}, n_{16}, n_{19}, and n_{20} belong the case 3.

IV.5. Merging the local UQP

The transformation step allows transforming a set of n hypergraphs $\{\mathcal{H}_1\ (\mathcal{V}_1,\mathcal{E}_1),..,\mathcal{H}_n\ (\mathcal{V}_n,\mathcal{E}_n)\}$, to n oriented graphs $\{\mathcal{G}_1\ (\mathcal{V}'_1,\mathcal{E}'_1),..,\mathcal{G}_n\ (\mathcal{V}'_n,\mathcal{E}'_n)\}$. The n oriented graphs are disjoints and their merging is quite easy which gives the graph $\mathcal{G}$ ($\mathcal{V}$" ,$\mathcal{E}$"), where

$$\mathcal{V}" = \bigcup_{1 \leq i \leq n} (\mathcal{V}') \tag{3.5}$$

$$\mathcal{E}" = \bigcup_{1 \leq i \leq n} (\mathcal{E}') \tag{3.6}$$

V. Complexity of the algorithms

As we explained in the previous sections, our approach has three main sequential steps. The complexity of each step is given separately:

- The first step consists in parsing the queries to identify the involved operations in order to construct the hypergraph. This task has a linear complexity $\mathcal{O}(N * P)$ where N and P represent respectively the number of queries and the maximal number of operations used by a given query. In the context of $\mathcal{DW}$, if we have D dimension tables, we have as maximum $(D + 1)$ selections (by assuming the selection are grouped), D joins, D projections and one aggregation so P =3*D +2.

- The second step concerns the process of hypergraph partitioning performed by *hMeTiS* algorithm. The complexity of *hMeTiS* has been studied by *Han et al.* [123], where it was shown that for a K-way partitioning the complexity is $\mathcal{O}((V + N)log(K))$, where V is the number of vertices (in our case, it has the number of logical operations involved in the workload) and N is the number of edges (number of queries).

- The last step is the transformation of each sub-hypergraph into oriented graphs. During this step, we need to successively identify the pivot node (*pivot*) and then remove it from the sub-hypergraph until all nodes have been removed. Hence, identification of the pivot, the cost of each node of the sub-hypergraph is computed. Therefore, the complexity is $\mathcal{O}(V^2)$, where V is the number of nodes inside the component. Each component has a limited number of queries and can be transformed independently, which allows our approach to scale.

In our approach, the queries and their operations (join nodes) are presented by a hypergraph. A hyperedge that represents a query can represent all possible orders between their nodes. This hyperedge can connect thousands of nodes. So our approach can scale with the number of dimension tables. Note that our transformation algorithm is applied in the context of star schema. If the structure of the schema changes (e.g. snowflake), new transformations are required.

VI. Performance evaluation

VI.1. Experimental setting

In this section, we present an experimental validation of our approach. We developed a simulator tool in Java Environment. This tool is composed by the following modules.

- Two UQP generation modules associated to the two studied optimization structures (materialized views and horizontal data partitioning). Each module contains several roles: (1) the parsing SQL-queries to get all selection, join, projection and aggregation nodes, (2) the hypergraph construction, (3) hypergraph partitioning that adapts *hMeTiS* tool, (4) the transformation of each hypergraph into an UQP using the appropriate benefit function according to the targeted $\mathcal{OS}$ (the benefit function is detailed in the next Chapter), (5) the merging process that assembles all selection, join, projection and aggregation nodes to generate the final UQP and (6) the display functionalities that use *Cytoscape*[9] plug-in to display the global processing plan.

- The $\mathcal{MV}$ selection module that takes an UQP as input and produces candidate views to be materialized (detailed in the next Chapter).

We have implemented another module implementing the of Yang et al.'s approach [273], by its two proposed algorithms: *feasible solution* and *0-1 integer programming* (cf. Chapter 2). To do so, we were obliged to develop the following functions: (a) generation of individual plan tree, (b) MVPP generation, using merging individual plans, (c) *0-1 matrix* representation of using queries and plans (d) selecting materialized views, (e) and an estimation of query processing using materialized views (by the means of query rewriting capabilities).
Regarding the data sets used by our experiments, we consider the *Star Schema Benchmark* (SSB) [194] with different data sizes (01 Gb and 100 Gb).

VI.2. The obtained results

VI.2.1. UQP generation scalability

Size of components

In the first experiments, we evaluate the process of partitioning a workload of queries in several subsets, where each subset contains queries that have high interaction between them. In this case we suppose that no cutting of hyperedges exists. We consider different query workloads randomly generated using *SSB* query generator [194] and a 100 Gb data warehouse.
We run our UQP generation tool and we calculate the number of components and their sizes (the number of queries in a component). As shown in the Table 3.2, the number of components increases when the number of queries increases. Most components are small in terms of

[9]http://www.cytoscape.org

number of queries which reduces considerably the complexity of optimizing the queries of each component. We can see also that the biggest component is still relatively small compared to the initial workload. For example, the biggest component of our 10 000 queries workload only has 357 queries.

Nbr of queries	Nbr of components	Max of queries/Component
100	20	15
200	38	31
400	72	62
500	89	83
800	120	123
1000	140	154
2000	204	178
5000	488	234
8000	703	278
10000	945	357

Table 3.2 *– Component according to workload*

VI.2.2. Quality of hypergraph partitioning

Computing cost

A battery of tests has been conducted to evaluate the UQP generation module against *Yang*'s and *Sellis*'s modules. In each test, we change the number of input queries to monitor the behavior of each algorithm. The set of the results are given in Figure 3.19. It should be noticed that Sellis's algorithm [223] *needs more than five days to generate a UQP for 30 queries.* Yang's algorithm [273] needs 10 hours generate a UQP for 20 00 queries. On the other hand, our algorithm takes about five minutes to generate a UQP for 10 000 queries, for both $\mathcal{OS}$ ($\mathcal{MV}$ and $\mathcal{HDP}$). This proves the scalability of our approach (Figure 3.20).

Quality of the unified query plan

To evaluate the quality of our approach, we consider the process of materialized view selection using the traditional approaches (e.g. Yang's algorithm [273]) in generating UQP and our finding. Therefore, we run consider a workload of 30 queries and a 100 Gb *SSB* data warehouse *SSB* deployed in Oracle 11g $\mathcal{DBMS}$. We use a server of 32 GB of RAM with Intel Xeon Processor E5530 (8M Cache, 2.40 GHz). The quality of each configuration is estimated using a cost model developed in our Lab (see Appendix IV). Figure 3.21 summarizes the obtained results. It shows that our approach outperforms largely Yang et al. 's algorithm. The obtained

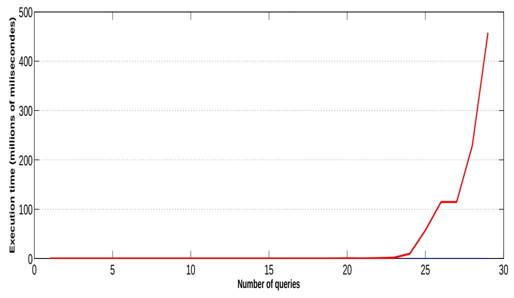


Fig. 3.19 – Execution time to generate an UQP using Sellis's Algorithm [223]

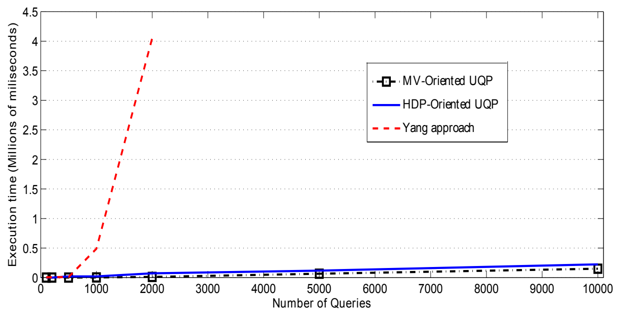


Fig. 3.20 – Execution time to generate an UQP using 3 methods

materialized views by Yang et al.'s and our approaches are then injected in Oracle 11g $\mathcal{DBMS}$ and the initial queries are evaluated by considering these two sets of views. The obtained results indicates the quality in reducing the overall cost of queries when our approach is considered.

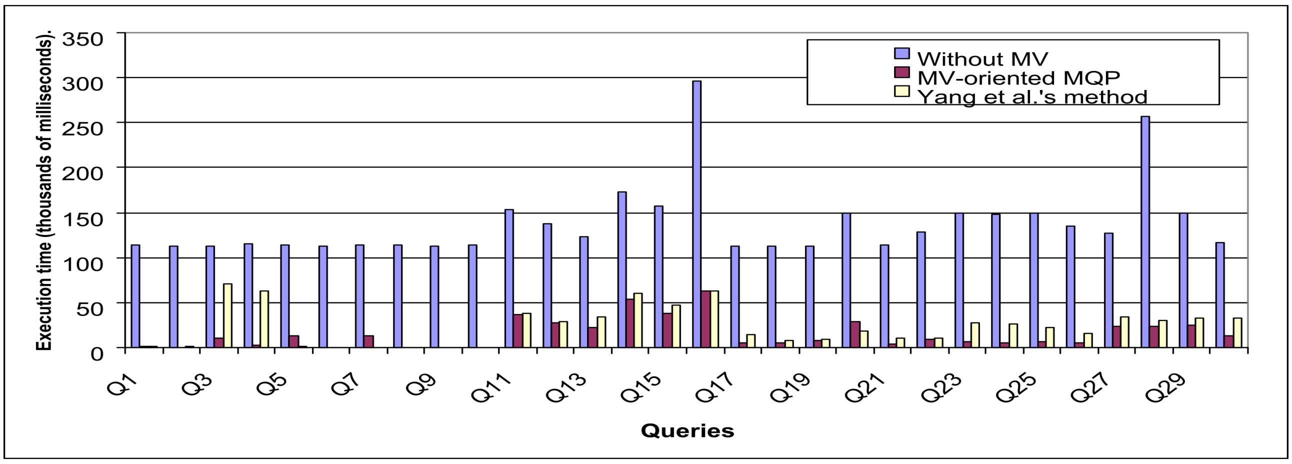


Fig. 3.21 – Individual queries execution time

VII. Conclusion

In this chapter, we first identified the limitations of the existing studies dealing with the problem of generation of an optimal UQP. It should be noticed that this generation is crucial for the process of selecting $\mathcal{OS}$. Secondly, with the fruitful collaboration with Graphics, Inc. realized for a long period to understand each other's, to find a correct analogy and get familiar with their tools. Once the analogy found, we adapt the EDA partitioning algorithms and tools to our problem to generate the best UQP for a given workload. The experimental results prove

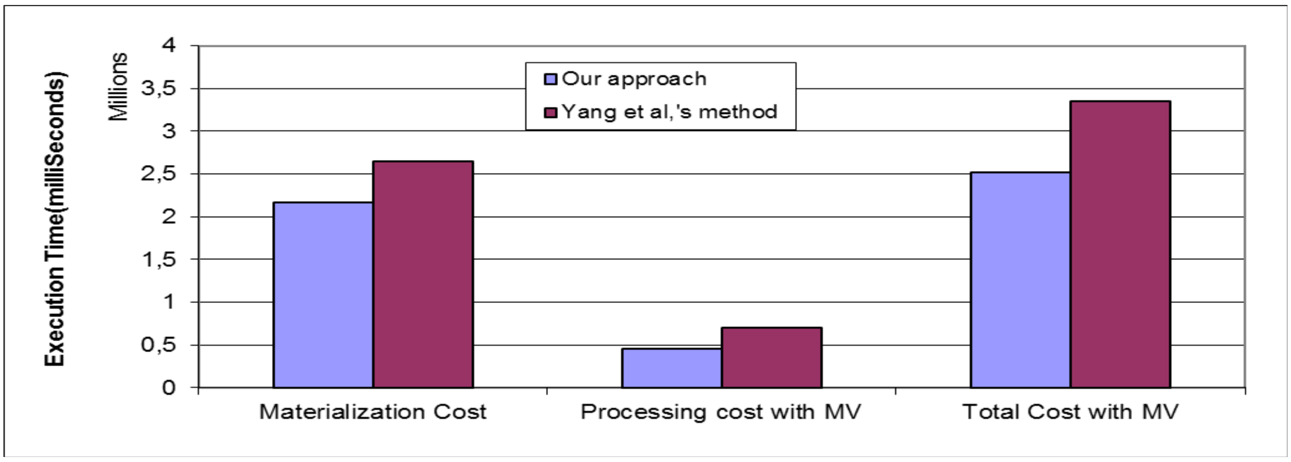


***Fig. 3.22** – Workload execution times*

the effectiveness and the efficiency of our approach when applied to a large workload of queries. We also showed the impact of using our generated UQP for selecting materialized views.

Chapter 4

What-if UQP Analysis

Contents

Abstract

Capturing the sharing of queries through a hypergraph driven scalable and efficient data structure is a good opportunity to evaluate the impact of changes of UQP on the process of selecting optimization structures ($\mathcal{OS}$). As we said in the previous chapters that the majority of algorithms selecting $\mathcal{OS}$ in the context of relational data warehouse consider an a priori generated UQP and then use it to run their selection algorithms. As a consequence, they assume that the two processes are dependent. The aims of this chapter at contradicting this assumption, by showing the strong dependency between UQP and $\mathcal{OS}$. This means that each $\mathcal{OS}$ has its own UQP.

I. Introduction

In the first generation of database, the query interaction phenomenon has mainly been studied to resolve the problem of multi-query optimization (MQO) without considering $\mathcal{OS}$ [223, 222, 198]. Afterwards, and after 10 years of waiting, it has been integrated to the process of selecting $\mathcal{OS}$, especially materialized views [273, 119] in 1999. This is due to the query interaction brought by data warehousing. Recently, the query interaction has been revisited and exploited to deal with the classical database problems problem such as horizontal data partitioning schema selection [154], query scheduling [154, 245], buffer management [154], query caching in the context of distributed data stream [224]. As we said before (cf. Chapter 2), and to the best of our knowledge, the work of Kamalakar Karlapalem group, when he was in Hong Kong University of Science and Technology, was the pioneer that showed the dependency between the generated of UQP and the selection of materialized views [273, 274], but he misses to generalize his findings to consider other $\mathcal{OS}$.
Recall that the generation of the optimal UQP needs the enumeration of all possible UQP, which is not realistic in the context of *big-queries*. Furthermore, an UQP of big-queries can offer an exponential number of $\mathcal{OS}$ candidate. This makes the process of choosing the optimal configuration for $\mathcal{OS}$ hard. So, to facilitate the exploration of these candidates, subdividing their search space is recommended.

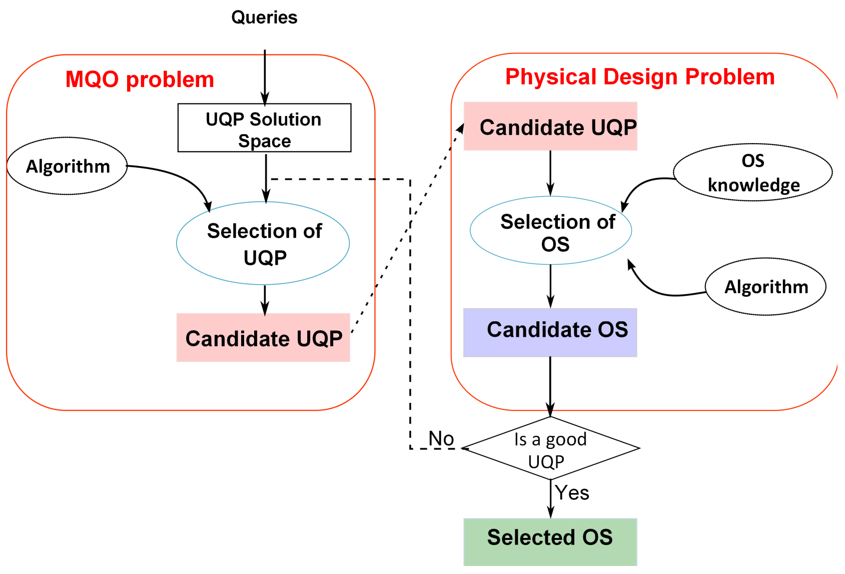


***Fig. 4.1** – Overlap between MQO and physical design problems*

Summing up, the UQP finds itself at the intersection between two worlds: the world of multi-query optimization and the world of physical design. Combining MQO and physical design problems gives raise to three challenging issues: (1) the generation of a UQP for a workload is NP-hard problem [222], (2) the selection of the UQP for specific $\mathcal{OS}$ needs the enumeration of all possible UQP and (3) the identification of the optimal $\mathcal{OS}$ from numerous candidate elements, has a high complexity. Figure 4.1 shows the overlap between two words MQO problems and physical design problem.

In this chapter, we propose a scalable methodology to select the best $\mathcal{OS}$ based on UQP that incorporates $\mathcal{OS}$ knowledge's in the UQP generation process. Our methodology uses the hypergraph theory to generate the UQP (see Chapter 2), in which it captures the query interaction in two ways: (i) dividing the initial problem in a set of several disjoint small sub-problems, which each one concerns a component of queries that will be exploited by the $\mathcal{OS}$ selection algorithm. This may contribute in reducing the overall complexity of the selection process. (ii) Injecting the $\mathcal{OS}$ knowledge when constructing the near-optimal UQP. Figure 4.2 shows the our methodology of constructing an $\mathcal{OS}$-oriented UQP.

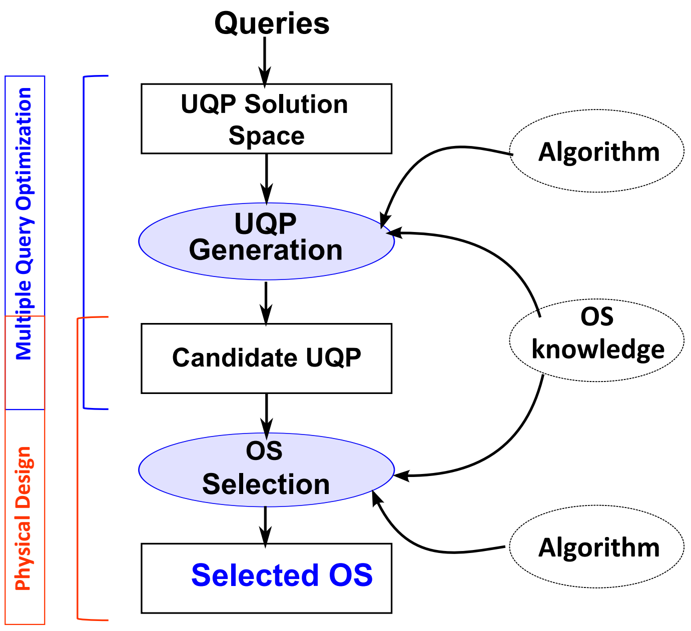


Fig. 4.2 – Injecting $\mathcal{OS}$ knowledge in MQO for physical design problems

To confirm our claim regarding the strong dependencies between the processes of generation of $\mathcal{OS}$ and UQP, we consider two problems: the selection of materialized views (considered as redundant structures) and the selection of horizontal partitioning (considered as a non-redundant structure).

This chapter begins by describing our methodology to generated $\mathcal{OS}$-oriented UQP (Section II). Section III shows the methods used to resolve our two problems by considering several constraints and scenarios. Section IV presents our experiments that confirm our proposal. Finally, Section V concludes this chapter.

II. A new approach for selecting OS

To transform an hypergraph to unified query plan by injecting $\mathcal{OS}$ knowledge, we need to define a benefit function that reflects the non-functional requirements using $\mathcal{OS}$ (like minimizing the processing nodes, energy). This benefit function is used to find the important node (pivot node), that will be deleted from the hypergraph and add it, gradually to the UQP.
As detailed in the previous Chapter 2, the generation of UQP follows five main steps:

- **Step 1:** parses the SQL query to identify the logical operations (nodes).
- **Step 2:** models the join nodes by a hypergraph, where the vertices set represents the set of join nodes and the hyperedge set represents the workload of queries.
- **Step 3:** generates connected components using hypergraph partitioning algorithms.
- **Step 4:** transforms each sub-hypergraph into an oriented graph using a cost model and implementation algorithms to order the nodes, in which the algorithms use $\mathcal{OS}$ knowledge.
- **Step 5:** merges the graphs resulting from previous transformation to generate the global unified query processing (UQP).

To have a specific generation of $\mathcal{OS}$-oriented UQP, the injection $\mathcal{OS}$ knowledge's in the **step 4** is necessary. More concretely, it corresponds to define a benefit function to represent $\mathcal{OS}$ related the considered non-function requirement in the general formalization of the $\mathcal{OS}$ selection problem.

II.1. Generation of MV-oriented UQP

All views that have positive benefit, are considered as potential materialized views. Intuitively, the views with minimum processing cost time have more chance to be selected. Hence, the non-functional requirement (minimizing query processing cost) can be translated to the benefit function of each node n_i, as follows:

$$benefit(n_i) = (nbr\text{-}1) * cost_{process}(n_i) - cost_{mat}(n_i) \tag{4.1}$$

where *nbr* is the number of queries using n_i, given that each node must be generated at least once before their use by queries, which gives the benefit of reuse depends to the number of reusing nodes (nbr-1). $cost_{process}$ (n_i) is the processing cost and $cost_{mat}$ $(n_i)$is the materialization cost of n_i. Vertices of the hypergraph which maximize the benefit function are selected first so that their benefit will propagates towards as much queries as possible.

II.2. Generation of HDP-oriented UQP

Using the selection predicate of a pivot node (that corresponds to the elected query), can efficiently improve the quality of the final partitioning schema by feeding the elected query with a $\mathcal{HDP}$-oriented UQP.
To this end, we push down join operations for which the gain on their selection predicates is maximum. Nodes that have selection predicates with extreme selectivity factors (too high or too low) are ignored. The benefit function of the node n_i, corresponds to the participation of resulting fragments when their selection predicates are chosen. So, the benefit function of the horizontal data partitioning is :

$$benefit(n_i) = (cost_{noFrag}(n_i) - cost_{frag}(n_i)) * nbr \tag{4.2}$$

where:
nbr: is the number of queries using n_i,
$cost_{frag}$ (n_i) is the processing cost of the node if its predicate selected for partitioning,
and $cost_{noFrag}$ (n_i) is the processing cost of the node without data partitioning.
To satisfy the data partitioning constraints (maximum of fragments, and ignoring selection predicates with extremely selectivity), the procedure of identifying the pivot node has be modified, as follows, to be more sensitive to the chosen $\mathcal{OS}$:

Algorithm 4: *getPivotNodeHDP* (Hypergraph $\mathcal{H}$)

```
1:  $benefitemax \leftarrow 0$;
2:  for all $v_i \in \mathcal{V}$ do
3:     $selectPredicate \leftarrow getSelectionPredicat$ $(v_i)$; { get the selection predicate of the
       dimension table of the node}
4:     if $selectivity$ $(selectPredicate) > threshold_{min}$ and $selectivity$ $(selectPredicate)$
       $<threshold_{max}$ then
5:        $candidatePredicate \leftarrow selectPredicate$; {ignore nodes with selectivity too
          high($threshold_{max}$) or too low ($threshold_{min}$) ofits selection predicates }
6:        $nbr \leftarrow nbrUse$ $(v_i)$; {$nbr$: number of hyperedges that connect $v_i$ }
7:        $benefit \leftarrow getBenefit(v_i, nbr)$ {the benefit calculated using the equation 4}
8:        if $benefit > benefitemax$ then
9:           $benefitemax \leftarrow benefit$;
10:          $pivot \leftarrow v_i$;
11:       end if
12:    end if
13: end for
14: return $pivot$
```

III. Application of OS-oriented UQP

In this section, we give two applications of $\mathcal{OS}$-oriented UQP to resolve the problem of selecting materialized views and to define the schema of horizontal data partitioning.

III.1. UQP as input for selecting materialized views

The $\mathcal{MV}$-oriented UQP is used to propose candidate views with and without the storage space constraint.

III.1.1. New Formalization of $\mathcal{MV}$ selection

$\mathcal{MV}$ selection in $\mathcal{DW}$ using $\mathcal{MV}$-oriented UQP is formalized as following:

- **Inputs:**
  - a $\mathcal{DW}$ with fact table F and d dimension tables $\{D_1, .., D_d\}$,
  - a workload $\mathcal{Q}$ of n queries, $\mathcal{Q} = \{Q_1, .., Q_n\}$.
  - a $\mathcal{MV}$-oriented UQP that represents the unified query plan for $\mathcal{Q}$.
- **Outputs:**
  - a set of materialized views to be materialized.
- **Constraints:**
  - a set of constraints S, related to store the selected views.
- **Objectives:**
  - the total query processing is minimized and the set of constraints is satisfied.

Resolving $\mathcal{MV}$ selection problem starts by proposing candidate views that are the join operations in the UQP (join nodes are chosen because they consume the most time to response a query). Only nodes with positive benefit will be selected as candidate, in which nodes are ordered in descending order, following their benefit. The benefit of node is the cost of its processing minus the cost of materializing it.

III.1.2. Description of selection algorithm

We look to materialized views independently in each connected component (CC). In each selection of a view, the algorithm recalculates the benefit of the remaining nodes by taking into account already selected nodes until no node with positive benefit is found. Algorithm 5 details the different steps of our selection of view in each component. To select all views that optimize workload, we repeat the previous algorithm for each component in the UQP (Algorithm 6).

Algorithm 5: Materialized views selection from component: *selectMVC*(*Component C*)

```
1: Input : $C$; {Component}
2: Output : $L_{MV}$ ; {List of materialized views} ;
3: $L_{MV} \leftarrow \phi$; {initialize materialized views set}
4: $L \leftarrow getAllCandidateNodes\ (C)$; {All nodes of C that have positive benefit}
5: $calculateBenefit\ (L[0],L)$; calculate the benefit of each node of $L$
6: $descendingOrder\ (L)$; {Descending order of the candidate nodes set}
7: while $benefit\ (L[0])$ and $L <> \phi$ do
8:    $add\ (L[0],L_{MV})$; {Add the first candidate node to the selected nodes set}
9:    $delete\ (L[0],L)$;
10:   $calculateBenefit\ (L[0],L)$; re-calculate the benefit of each node of $L$
11:   $descendingOrder\ (L)$;
12: end while
```

Algorithm 6: Materialized views selection from UQP: *selectFinalMV*(UQP *p*)

```
1: Input : $p$; {UQP }
2: Output : $MV_{final}$ ; {List of materialized views} ;
3: $MV_{final} \leftarrow \phi$; {initialize materialized views set}
4: $C \leftarrow getComponents$ (UQP); {All connected component}
5: for all $c \in C$ do
6:    $MV \leftarrow selectMVC\ (c)$; {get selected $\mathcal{MV}$, Algorithm 5}
7:    $MV_{final} \leftarrow MV_{final} \cup MV$;
8: end for
```

III.2. UQP as input for dynamic materialization and query scheduling

Traditional selection of views supposes consider a static workload. To relax this hypothesis, we propose a on demand dynamic materialized of views. If a view is no longer efficient, it will be removed from materialized view pool. To do so, we have to integrate query scheduling module in the process of this materialization.

III.2.1. Formalization of the problem

Before formalizing the $\mathcal{MV}$ Problem (MVP) considering the Query Scheduling Problem (QSP, we think it would be wiser to propose a separate formalization of both MVP and QSP.
The $\mathcal{MV}$ problem considering QSP takes (i) A $\mathcal{DW}$ and (ii) a set of queries $\mathcal{Q}$, (iii) a set of intermediate nodes candidates for materialization; a constraint representing the limited storage size. The problem aims at providing: (i) a scheduled set of queries and (ii) $\mathcal{MV}$, minimizing the overall processing cost of and satisfying the storage constraint.
Query scheduling problem (QSP) is formalized as follows:

- **Inputs:**
  - a $\mathcal{DW}$ with fact table F and d dimension tables $\{D_1, .., D_d\}$,
  - a workload $\mathcal{Q}$ of n queries, $\mathcal{Q}$ =$\{Q_1, .., Q_n\}$.
  - set of m candidate views to be materialized V =$\{v_1, .., v_m\}$.
  - a disk allocation policy (e.g., *FIFO*).
- **Outputs:**
  - scheduled queries of the workload into a new ordered set.
- **Constraints:**
  - a set of constraints S, like disk space to materialize views.
- **Objectives:**
  - the workload have the least execution cost.

The MVP considering QSP takes (i) A $\mathcal{DW}$ and (ii) a set of queries, (iii) a set of intermediate nodes candidates for materialization; a constraint representing the limited storage size. The problem aims at providing: (i) a scheduled set of queries and (ii) a set of $\mathcal{MV}$ that minimize the overall processing cost of queries and satisfying the storage constraint.
Consequently, we address the problem of dynamic materialized view selection by considering the query scheduling. A formalization of the problem of view selection considering the re-ordering of numerous queries is given as follows.

- **Inputs:**
  - a $\mathcal{DW}$ with fact table F and d dimension tables $\{D_1, .., D_d\}$,
  - a workload $\mathcal{Q}$ of n queries, $\mathcal{Q}$ =$\{Q_1, .., Q_n\}$.
  - a $\mathcal{MV}$-oriented UQP that represents the unified query plan for $\mathcal{Q}$,
  - a disk allocation policy (e.g., *FIFO*).
- **Outputs:**
  - scheduled queries of the workload into a new ordered set.
  - set of m candidate views to be materialized, V =$\{v_1, .., v_m\}$.
- **Constraints:**
  - a set of constraints S, like disk space to materialize views.
- **Objectives:**
  - the workload have the least execution cost,
  - satisfy all constraints, like the space occupied, in each period, by $\mathcal{MV}$ satisfies the space constraint.

Dynamic materialization with query scheduling has three main modules: the first selects $\mathcal{MV}$ candidate, the second orders the queries and the third manages views materialization. These modules are discussed in the next sections.

III.2.2. $\mathcal{MV}$ Selection algorithm

Note that all nodes of the global plan are candidate for materialization which may represent a huge number. For instance, in our experiments, we consider 1 000 queries involving 1552 join nodes. As a consequence, a pruning mechanism is needed. It shall take into account the benefit of the nodes and their constraints related to their storage and maintenance. To do so, we define some functions:

- $cost_{WO}(q_i, \Phi)$: the processing cost of the query q_i without view(s).
- $cost_{WV}(q_i, V_j)$: the query processing cost of query q_i using the materialized view V_j.
- $cost_{Mat}(V_j)$: the maintenance cost of the view V_j.
- $Size(V_j)$: the cost needed to store the view V_j.

We define the benefit of a given view V_j (denoted by $Benefit(V_j)$) by:

$$benefit(V_j) = cost_{WO}(V_j) - cost_{WV}(V_j) - cost_{Mat}(V_j) \tag{4.3}$$

where $cost_{WO}(V_j)$ and $cost_{WV}(V_j)$ represent respectively the total processing cost of queries without/with the view V_j. Instead of treating the whole search space including all candidates as in the usual approaches for $\mathcal{MV}$, we propose to use a *divide-conquer approach*, where the search space is divided into several sub search spaces, where each one corresponds to a connected component of UQP. Contrary to the existing studies where they allocate the whole storage constraint to all views candidate, our approach allocates this storage to each component to be fair. Then, each component C_k is processed individually, where its nodes are sorted according their benefits and their ability to satisfy the storage constraint. Three selection cases are possible. Let N^{C_k} be the set of nodes of C_k having a greater benefit. The top nodes satisfying the storage constraint are selected.

III.2.3. Query scheduling

To avoid massively view dropping, we schedule queries. This is done by respecting the following principle: *when a view is materialized, it should optimize the maximum of queries before its dropping*. Therefore, we propose the following procedure supported by an example in which we consider a connected component with 14 queries (Figure 4.3-a).

1. The identification of node(s) of each component with maximal benefit (called queen nodes). In our example, four *queen nodes* are selected:
   $\{qn_1, qn_2, qn_3, qn_4\}$ (represented by solid nodes in Figure 4.3-a).
2. Ordering queen nodes: Let N^{C_k} be the number of nodes of the component C_k. Their ordering is based on their benefit. The benefit of the queen nodes are propagated to their

Algorithm 7: Materialized views selection from component: *selectMVC*(*Component* *C*,space *S*)

```
1: Input : $C$; {Component}
2: Input : $S$; {disk space}
3: Output : $L_{MV}$ ; {List of materialized views} ;
4: $L_{MV} \leftarrow \phi$; {initialize materialized views set}
5: $L \leftarrow getAllCandidateNodes\ (C)$; {All nodes of C that have positive benefit}
6: $calculateBenefit\ (L[0],L)$; calculate the benefit of each node of $L$
7: $descendingOrder\ (L)$; {Descending order of the candidate nodes set}
8: $diskSpace \leftarrow 0$;
9: while $benefit\ (L[0])$ and $diskSpace \leq S$ and $L <> \phi$ do
10:    $add\ (L[0],L_{MV})$; {Add the first candidate node to the selected nodes set}
11:    $diskSpace \leftarrow diskSpace + size(L[0])$;
12:    $delete\ (L[0],L)$;
13:    $calculateBenefit\ (L[0],L)$; re-calculate the benefit of each node of $L$
14:    $descendingOrder\ (L)$;
15: end while
```

queries. As a consequence, each query may be assigned to a weight representing the sum of the benefit of its nodes. These weight are then used to schedule the queries based on their overall benefit (Figure 4.3-d).

III.2.4. Manage Views materialization

Till now, materialized views candidate are identified and the order of queries. Based on the different cost models, it can easily decide on materializing or de-materializing views by performing simulation using Algorithm 8.

Algorithm 8: materializeView (*mv*)

```
1: $cost \leftarrow estimateMaintenance(mv)$;{Estimate the maintenance cost of the view }
2: changeStat($mv$, true);{ change the stat of the view as materialized}
3: $diskSpace \leftarrow diskSpace - sizeOf(mv)$;
4: return $cost$;
```

III.3. UQP as input for selecting data partitioning schema

III.3.1. Formalization of the problem

The horizontal data partitioning is formulated as following:

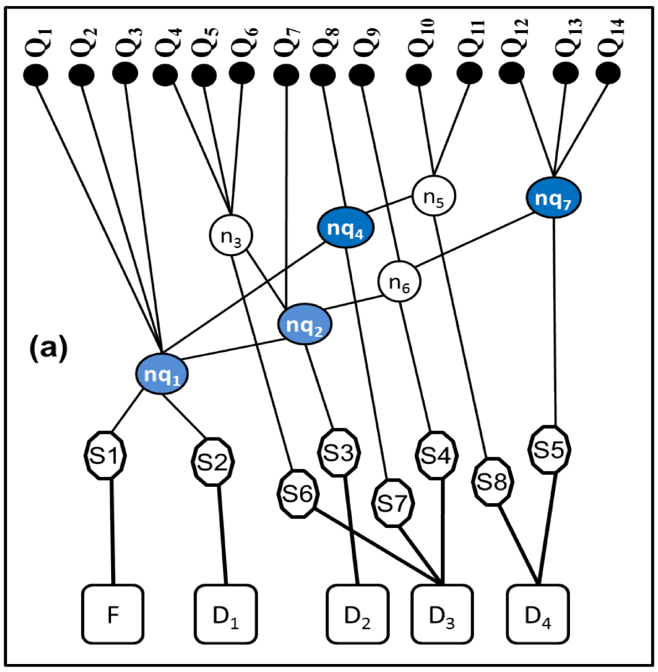
(b)

Node	Benefit
nq_1	400
nq_2	100
nq_4	300
nq_7	200

Node	Benefit
nq_1	400
nq_4	300
nq_7	200
nq_2	100

(c)

Query	Use	Weight
Q_1	nq_1	400
Q_2	nq_1	400
Q_3	nq_1	400
Q_4	nq_1, nq_2	500
Q_5	nq_1, nq_2	500
Q_6	nq_1, nq_2	500
Q_7	nq_1, nq_2	500
Q_8	nq_1, nq_4	700
Q_9	nq_1, nq_2	600
Q_{10}	nq_1, nq_4	700
Q_{11}	nq_1, nq_4	700
Q_{12}	nq_1, nq_2, nq_7	700
Q_{13}	nq_1, nq_2, nq_7	700
Q_{14}	nq_1, nq_2, nq_7	700



(d)

Query order														
	Q_8	Q_{10}	Q_{11}	Q_{12}	Q_{13}	Q_{14}	Q_9	Q_4	Q_5	Q_6	Q_7	Q_1	Q_2	Q_3

Fig. 4.3 – Example of query scheduling method

- **Inputs :**
  - a $\mathcal{DW}$ with fact table F and d dimension tables $\{D_1, .., D_d\}$,
  - a workload $\mathcal{Q}$ of n queries, $\mathcal{Q} = \{Q_1, .., Q_n\}$,
  - a threshold M, that defines the maximum of possible fragments.
- **Outputs :** - a data partitioning schema.
- **Constraints :** - a threshold M that defines the maximum of possible fragments.

III.3.2. Algorithm description

We have used an horizontal data partitioning algorithm, developed in our lab [50]. The algorithm takes in consideration of query interaction (represented in UQP), in selecting $\mathcal{HDP}$ schemes for relational data warehouses. The problem using incremental encoding of any horizontal partitioning schema. The steps of algorithm are:

- Decomposition of an attributes' domain into sub-domains, following selection predicated used in the UQP. The decomposition is systematically done incremental encoding based on the UQP. Note that, all selection predicate discarding selection with high or low selectivity factors.

- $\mathcal{HDP}$ schema is represented by fixed coding, which is juxtaposition of arrays representing attributes sub-domain.
- incremental encoding generation using successive vertical and horizontal split of the initial array representations, which represent the selection predicate.
- pruning the search space using elected query, that correspond to the first query of pivot node.

IV. Experimental Evaluation and Analysis

IV.1. Experimental setting

In this section, we present an experimental validation of our approach. We developed a simulator tool in Java Environment. This tool consists in the following modules.

- Two UQP generation modules, one for each optimization structure (materialized views and horizontal data partitioning). Each module contains several functions : (1) a parsing SQL-queries function to get all selection, join, projection and aggregation nodes, (2) hypergraph generation function to represent the queries by an hypergraph of nodes, (3) hypergraph partitioning function that uses $hMeTiS$ tools to partition the hypergraph into several connected sections, (4) a transformation function, to transform each hypergraph into an UQP using the appropriate benefit function according to the targeted OS, (5) a merging function that assembles all selection, join, projection and aggregation nodes to generate the final UQP, (6) a display function that uses *Cytoscape*[1] plug-in to display the global processing plan.
- a $\mathcal{MV}$ selection module, that takes an UQP as input and produces candidate views to be materialized.
- $\mathcal{HDP}$ module, that takes a UQP as input and generates a Horizontal Partition Schema.

Another module is developed to implement approach of Yang et al., [273] considering their two algorithms: *feasible solution* and *0-1 integer programming*. We have developed the following functions: (a) generation of individual plan tree, (b) MVPP generation, using merging individual plans, (c) *0-1 matrix* representation of using queries and plans (d) selecting materialized views, (e) and an estimation of query processing using materialized views.
Our tests are run in star schema $\mathcal{DW}$ context, which we have chosen a star schema $\mathcal{DW}$. We have used *Star Schema Benchmark* (SSB) [194] with different sizes (01 Gb and 100 Gb) of data.

[1]http://www.cytoscape.org

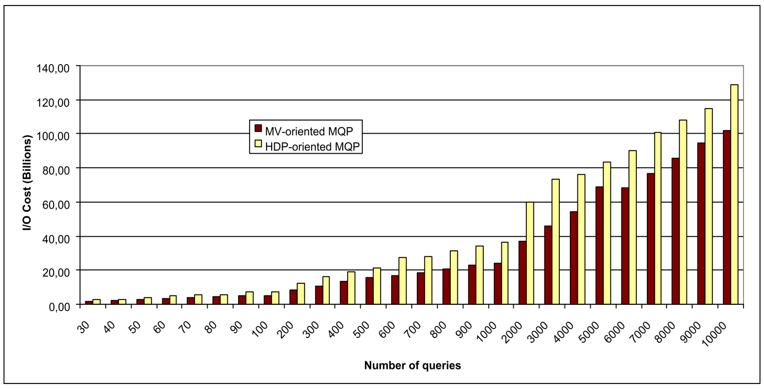


Fig. 4.4 – Total cost of workload using $\mathcal{MV}$ selected using $\mathcal{MV}$-oriented UQP and $\mathcal{HDP}$-oriented UQP

IV.2. OS-Sensitivity of UQP

IV.2.1. Theoretical validation

We now exhibit how the choice of UQP for $\mathcal{OS}$-selection purposes influences the quality of the resulting OS. We use a cost model developed in our lab IV, to estimate query processing and views maintenance cost. This model estimates the number of Inputs/Outputs pages required for executing a given query. To perform this experiment, we developed a simulator tool using Java Environment. The tool can automatically extract the data warehouse's meta-data characteristics. The data warehouse used in our test is SSB (Start Schema Benchmark) [193]. Its size is 100Go, with a facts table *Lineorder* of 600 millions of tuples and four dimension tables: *Part*, *Customer*, *Supplier* and *Dates*. We used a SSB query Generator to generate 10000 queries for *SSB* data warehouse. Our evaluation is conducted using those queries, which cover most types of OLAP queries.
We proceed to generate $\mathcal{MV}$-oriented UQP and $\mathcal{HDP}$-oriented UQP for different workloads of queries varying from 30 to ten thousands (10000) queries. The resulting UQP are then used by two different OS-selection algorithms : the MV-selection algorithm and the $\mathcal{HDP}$-selection algorithm. As shown in Figure 4.4 we see that the $\mathcal{MV}$-oriented *UQP* always produces a better set of candidate views to materialize than the $\mathcal{HDP}$-oriented UQP. Reciprocally, Figure 4.5 shows that the $\mathcal{HDP}$-oriented UQP always leads to a better fragmentation schema than the $\mathcal{MV}$-oriented UQP. The difference in performance gain increases with the number of queries.

IV.2.2. Oracle Validation

Finally, in order to validate our theoretical results, we select materialized views candidates and the fragmentation schema using our $\mathcal{OS}$-oriented UQP generation modules for two workloads of 500 and 3000 queries respectively, and we deploy the results in Oracle 11g DBMS. Figures 4.6 and 4.7 compare maintenance costs, query processing costs and total execution costs using

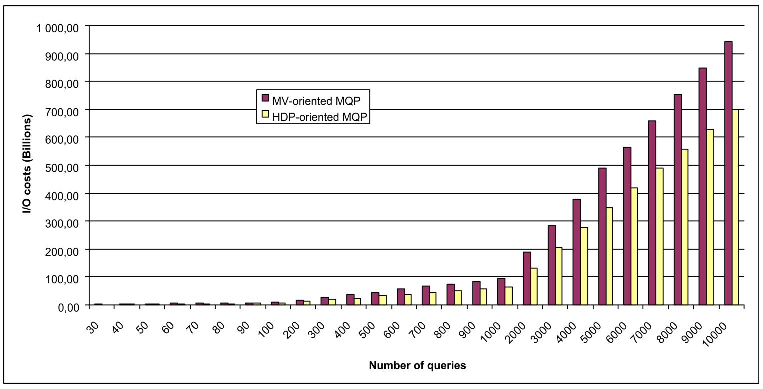


Fig. 4.5 *– Total cost of workload using $\mathcal{HDP}$ selected using $\mathcal{MV}$-oriented UQP and $\mathcal{HDP}$-oriented UQP*

materialized views selected using either a $\mathcal{HDP}$-oriented or a $\mathcal{MV}$-oriented UQP. $\mathcal{MV}$-oriented UQP always give better results.

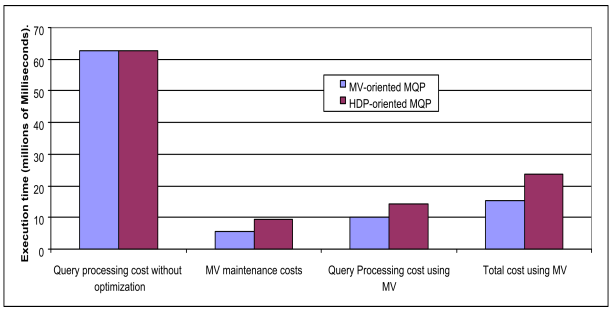


Fig. 4.6 *– Different costs using MV in Oracle for 500 queries*

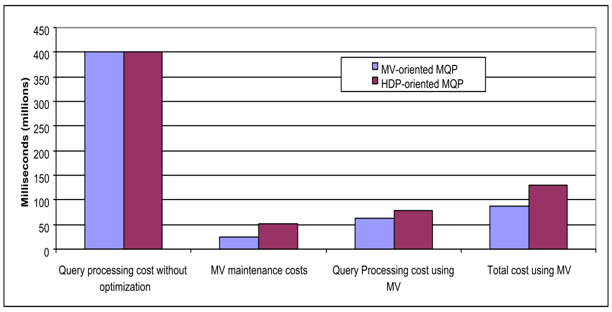


Fig. 4.7 *– Different costs using MV in Oracle for 3000 queries*

Similarly, in Figures 4.8 and 4.9, we compare execution costs using a fragmentation schema obtained by using either a $\mathcal{HDP}$-oriented or a $\mathcal{MV}$-oriented UQP.
The $\mathcal{HDP}$-oriented UQP gives better results. Results are summarized in Figures 4.10 and 4.11. Theoretical results are confirmed by Oracle experiments.

IV.3. The quality of OS-oriented UQP

IV.3.1. Theoretical validation

To evaluate the impact of our obtained $\mathcal{MV}$-oriented UQP on the problem of selecting materialized views. We compare the algorithm detailed above and compare it against Yang et al. [273]. To estimate the query processing and views maintenance costs in our tests, we used a

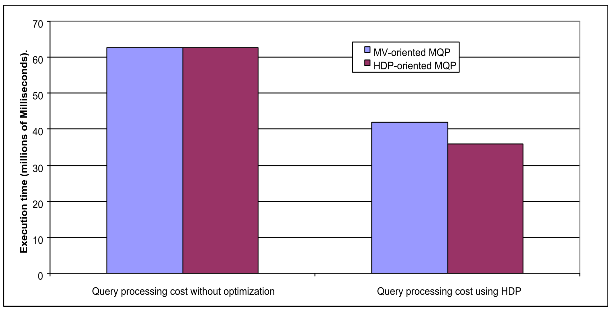


Fig. 4.8 – Execution costs using HDP in Oracle for 500 queries

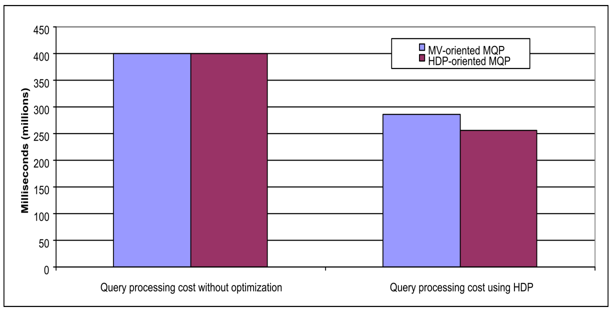


Fig. 4.9 – Execution costs using HDP in Oracle for 3000 queries

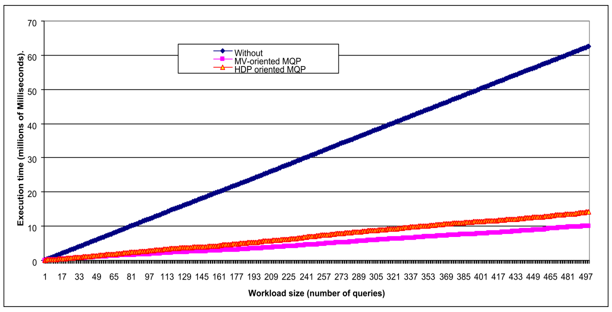


Fig. 4.10 – Validation of MV in Oracle for 3000 queries

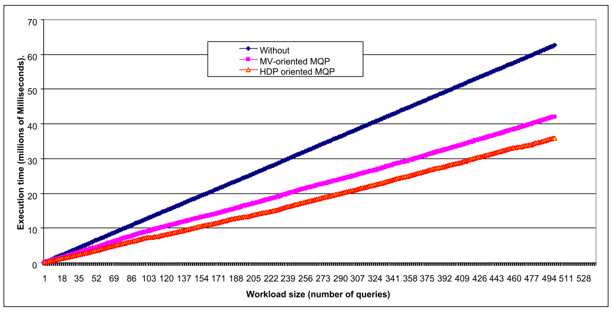
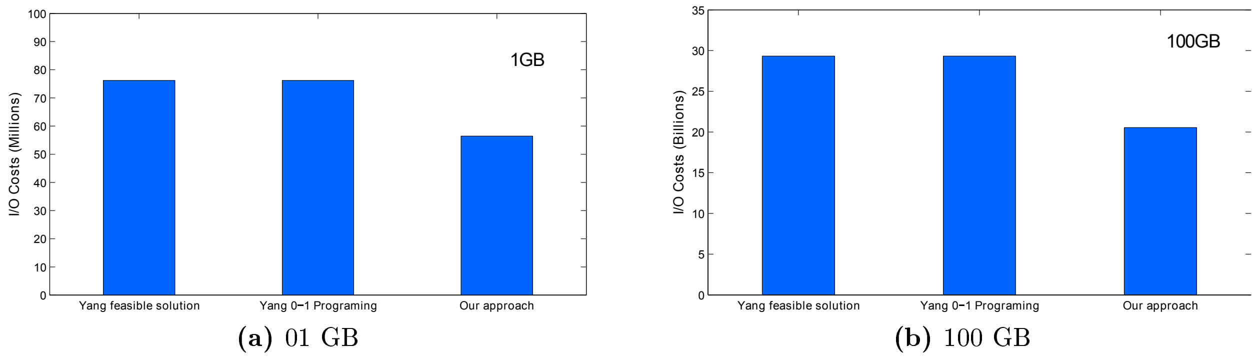


Fig. 4.11 – Validation of HP in Oracle for 3000 queries

cost model developed in our lab (see Annex IV). This model estimates the number of Inputs/Outputs pages required for executing a given query. We use *hash-join* implementation of joins.



(a) 01 GB **(b)** 100 GB

Fig. 4.12 – Query processing cost using MV selected without constraint

To perform this experiment, Ww consider a workload of 30 OLAP queries running on two data

warehouses (1GB and 100GB). The obtained results are described in Figure 4.12. They show that our method is rather slightly better than its competitor.
To test our approach for a large number of queries, we conduct a series of tests with workload of varying sizes (generated randomly using *SSB-query generator*). The different types of queries are: queries 1-joins, 2-joins, 3-joins and 4-joins; with also aggregation and group-by. For each workload, its UQP is generated and $\mathcal{MV}$ selection algorithm is applied. We estimate the cost of the workload with and without using views ($cost_{with}$ and $cost_{without}$ respectively), and we calculate the optimization rate (1-$cost_{with}/cost_{without}$). As shown in Figure 4.13, our approach becomes more interesting as the number of queries increase (90% optimization rate for a workload of 10000 queries).

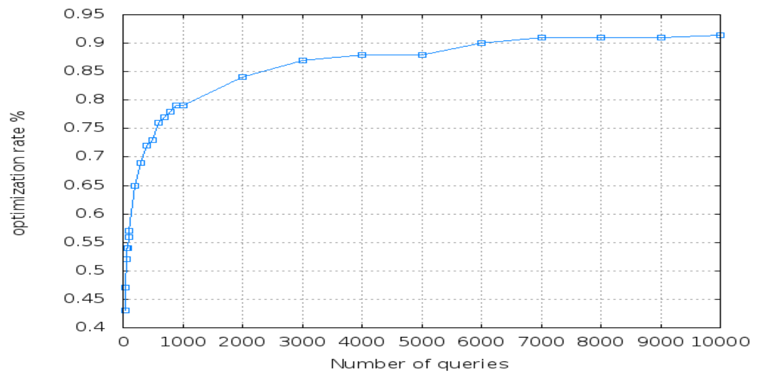


Fig. 4.13 *– Optimization rate of query processing cost using $\mathcal{MV}$*

IV.3.2. Oracle validation

Some of our simulated results are then deployed on Oracle 11g DBMS, running on a Core 2 Duo server with 2.40GHZ CPU and 32 GB of main memory. We use the Star Schema Benchmark (SSB) [194] with 100 GB of data. We consider a workload of 30 queries running on the data warehouse with 100 GB. The obtained results described in Figure 4.14 are quite similar to those obtained by our simulator.

IV.4. Dynamic Materialization with query scheduling

IV.4.1. Theoretical validation

We conduct several experiments to evaluate the efficiency of our dynamic materialization algorithm. we developed a simulation tool using Java that run the three algorithms: (1) our dynamic materialization algorithm; (2) dynamic materialization algorithm proposed by Phan et al.'s [203] and (3) $\mathcal{MV}$ selection algorithm proposed by Yang et al's [273] using a static formalization. To analyze the behaviors of these algorithms, we consider four scenarios: (1)

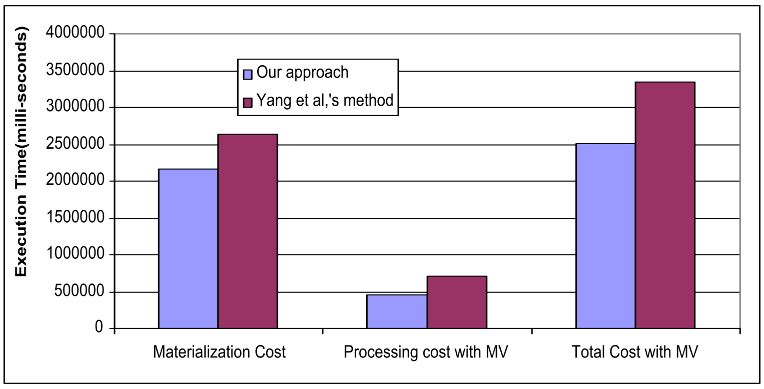


Fig. 4.14 *– Oracle validation of $\mathcal{MV}$ selection without constraint*

using static materialization, (2) dynamic materialization, (3) with considering query scheduling and (4) without query scheduling. These scenarios are tested by varying: (a) the size of data warehouse, (b) consideration of different query workload's randomly generated using *SSB-query generator* and (c) varying the storage space constraint. The simulated results are then deployed on Oracle 11g *DBMS*, running on a Core 2 Duo server with 2.40GHZ CPU and 32 GB of main memory, and the star Schema Benchmark (SSB) with 100 GB.

In the first experiments, we test the interest of dynamic and query scheduling on optimizing queries. To perform our experiments, we consider a data warehouse with different sizes (1Gb, 10Gb, 100Gb and 1 Tb) and a workload of 30 queries, the candidate nodes are selected using our approach. Three scenarios are considered: (i) naive scenario in which nodes are materialized till the saturation of storage space, (ii) materializing without scheduling using our dynamic approach and considering the workload as pre-ordered[2] and (iii) materializing with query scheduling. The overall cost of queries in terms of inputs/outputs is then computed by varying the storage constraint. Figure 4.15 summarizes the obtained results. The main lesson is: the dynamic materialization with query scheduling outperforms the other scenarios whatever the size of the data warehouse. This shows the interest of incorporating the query scheduling in materializing views.

In the second experiments: we test the performance of our approach compared with two existing approaches: Phan et al.'s method [203], and Yang et al.'s method [273]. For Phan method, we have developed the following algorithms: (i) a genetic algorithm to find an optimal query permutation by emulating Darwinian natural selection of 1000 generations; (ii) algorithm to select candidate nodes which are the nodes that have greater benefit have been selected as candidates (in Phan et al.'s, they are used *DB2 advisor* to have those all nodes with greater benefit); (iii) pruning algorithm of candidate views using their benefit; (iv) an evaluation algorithm to estimate the total net benefit of using pruned candidate views set by query workload; (v) algorithm to manage the cache of the views (LRU). For Yang et al. approach's [273], we

[2]The query scheduling module of our approach in this case is obsolete.

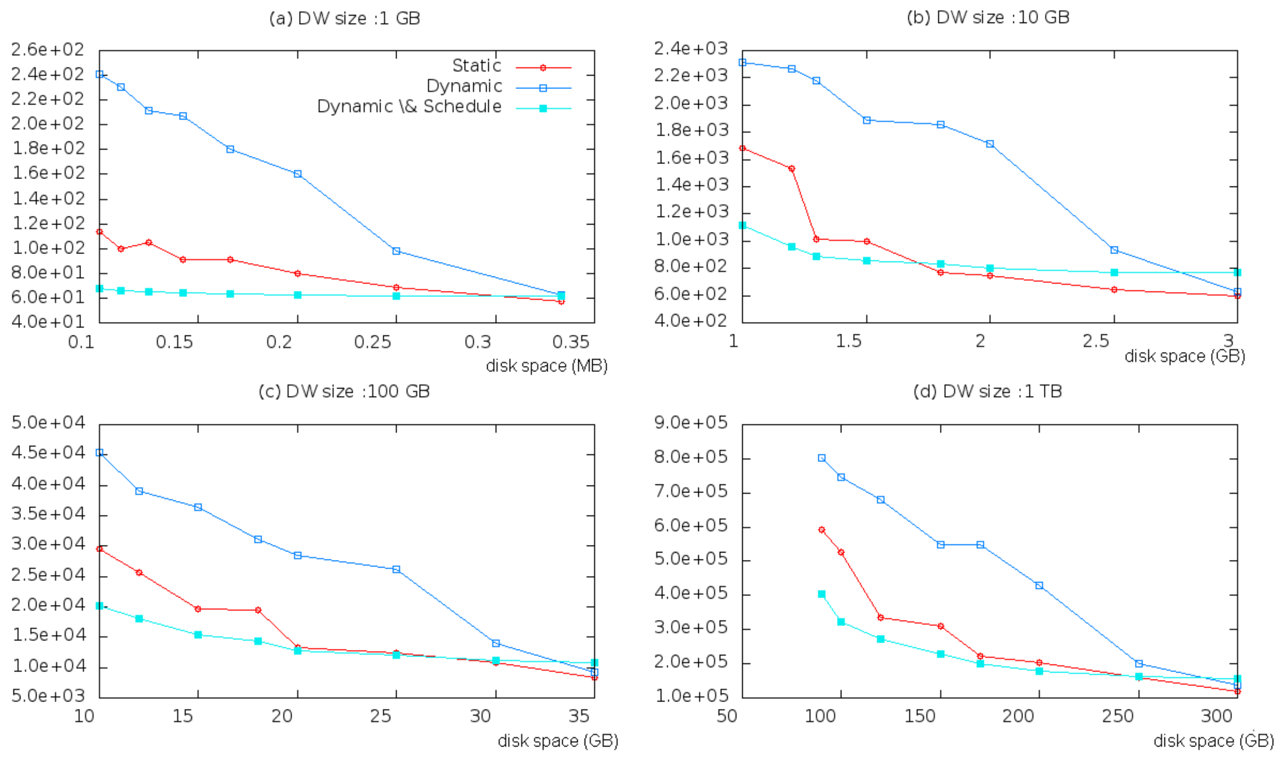


Fig. 4.15 – Advantage of dynamic materialization with query scheduling

have developed the following functions: (i) generation of individual plan tree, (ii) generating MVPP, using merging individual plans (iii) selecting materialized views using MVPP (iv) estimation of query MVPP using views. To show the performance of our approach three scenarios are considered:

1. **Static materialization:** we consider a data warehouse with 1Gb and a workload of 30 queries, the nodes selected by each algorithm are materialized until the saturation of the fixed storage space. As shown in the Figure 4.16-a, there is not a big difference between the three methods. This proves that our approach does not avoid the selection of best views.

2. **Dynamic materialization without query scheduling**, we have used the same configuration as static materialization. As shown in the Figure 4.16-b, Phan et al.'s approach outperforms our approach. This is due to the difference of materialization/dematerialization number (Figure 4.16-c). Phan et al.'s method tries to find best candidate views that optimize all queries of the workload. This minimizes the dropping process of the views. The views in our approach are divided a many sub sets, where each sub-set optimizes some queries, which increases the probability of dropping views if the query not

scheduled.

3. **Dynamic materialization with query scheduling:** we have used data warehouse with different size (1Gb and 100Gb) and a workload of 30 queries. As shown in the Figures 4.16-c and 4.16-e, our approach outperforms Phan method because the number dropping is minimal and each materialized views are used maximally to optimize the queries of the component. As shown in the Figures 4.16-f, our approach outperforms the traditional techniques in the context of big queries (in our tests: 1000 queries).

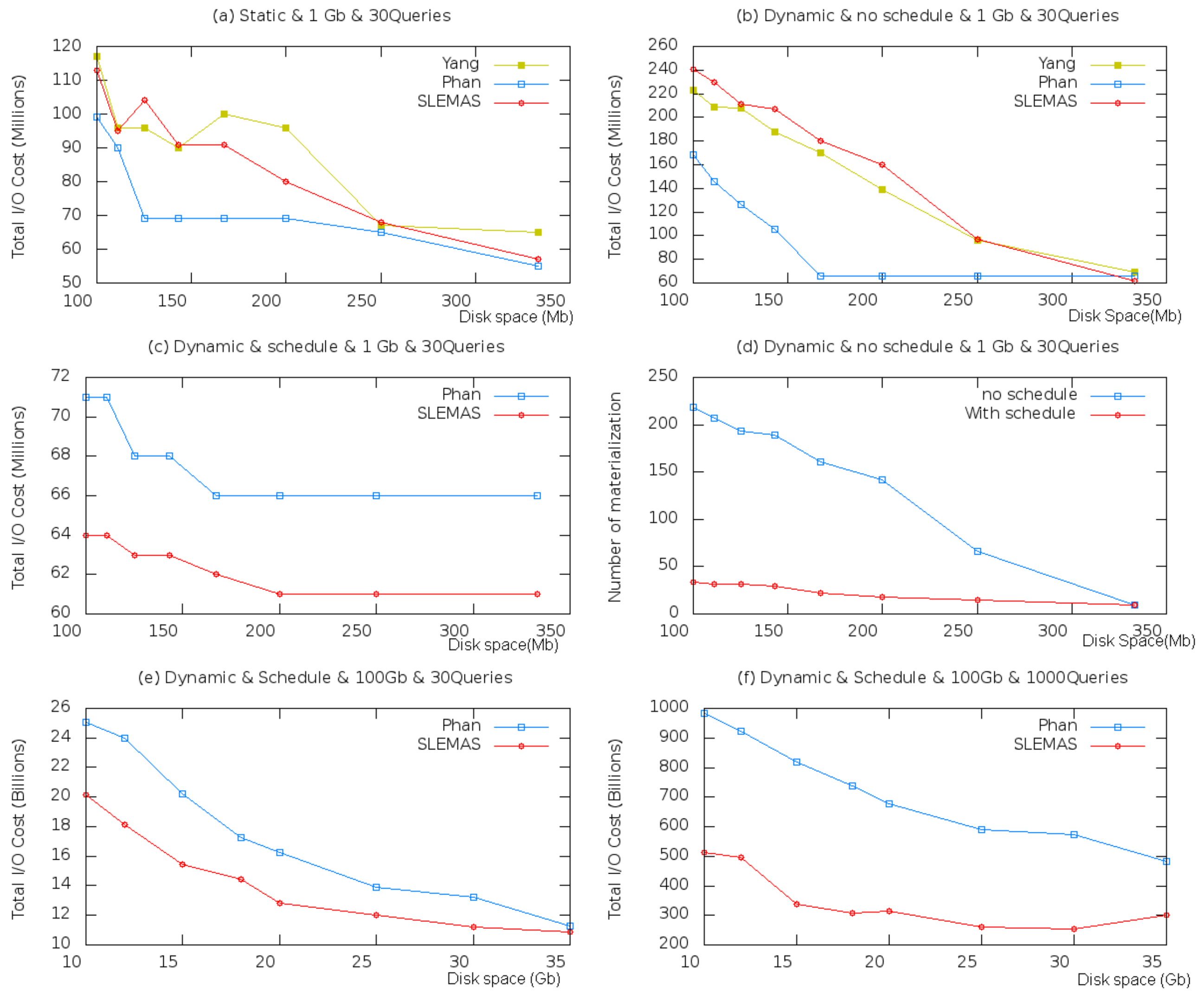
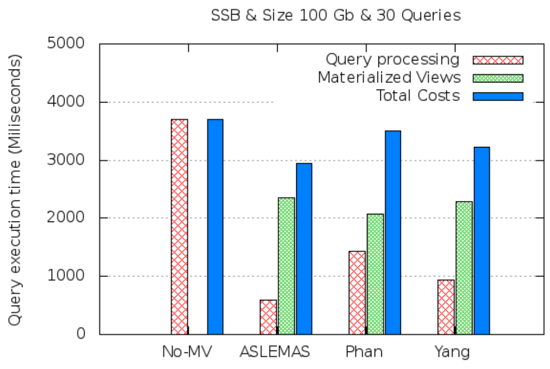


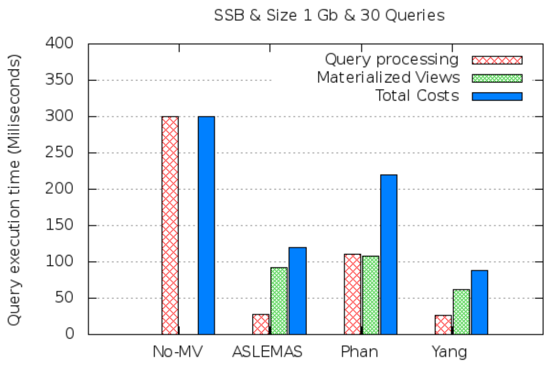
Fig. 4.16 – Performance of dynamic materialization approach

IV.4.2. Validation in Oracle

Due to the complexity and time needed to deploy all theoretical solutions on Oracle DBMS, we propose the following to do our validation: we consider a workload of 30 queries running on two data warehouses (1GB and 100GB). The disk storage is set to 400MB and 30Gb respectively.



(a) 100 Gb



(b) 1 Gb

Fig. 4.17 *– Oracle validation of dynamic materialization*

The obtained results described in the Figures 4.17a and 4.17b which are quite similar to those obtained by our simulator, which shows the quality of our used cost models. The results show that using a hypergraph partitioning algorithms can efficiently give a good solution.

V. Conclusion

In this chapter, we discussed an example-driven approach to select optimization structures (materialized views and horizontal data partitioning) basing on $\mathcal{OS}$-oriented UQP. The issue of finding candidate set of optimization structures dwarfed when compared to the complexity and ambiguity associated with Big-queries. In the next chapter, we will discuss the example-driven of modeling parallel data warehousing.

Chapter

5 Query Interaction Serving the Deployment Phase

Contents

Abstract

In this chapter, we detail our contribution that consists at using $\mathcal{OS}$-oriented UQP to resolve the problem of parallel data warehouse deployment design. This design is complex, since it requires several phases: data partitioning, fragment allocation, load balancing, query processing, etc. The main particularity of these phases is their property to take into account the workload. Therefore, it is appropriate to generalize our finding in unifying UQP and MQO in the context of parallel data warehouse design.

I. Introduction

With the era of Big Data, we are facing a data deluge[1]. Multiple data providers are contributing to this deluge. We can cite three main examples: **(i)** the massive use of sensors (e.g. 10 Terabyte of data are generated by planes every 30 minutes), **(ii)** the massive use of social networks (e.g., 340 million tweets per day), **(iii)** transactions (*Walmart* handles more than 1 million customer transactions every hour, which is imported into databases estimated to contain more than 2.5 Peta-bytes of data). The decision makers need fast response time to their requests in order to predict in *real time* the behaviour of users, so they can offer them services via analyzing large volumes of data. The data warehouse ($\mathcal{DW}$) technology deployed on conventional platforms (e.g. centralized) has become obsolete, even with the spectacular progress in terms of advanced optimization structures (e.g., materialized views, indexes, storage layouts used by DBMS, etc.). Despite this, the sole use of these structures is not sufficient to gain efficiency during the evaluation complex OLAP queries over relational $\mathcal{DW}$. To deal with this data deluge and simultaneously satisfy the requirements of company's decision makers, distributed and parallel platforms have been proposed as a robust and scalable solution to store, process and analyze data, with the layers of modern analytic infrastructures [98]. Editors of $\mathcal{DBMS}$ already propose *turnkey parallel platforms* to companies to adopt these solutions (e.g., *Teradata*). Another alternative is to go to the Cloud (e.g. Amazon Redshift). Note that these solutions may become rapidly expensive when data size grows. For instance, the cost of storing 1 Tera-bytes of data per year in *Amazon Redshift* is about 5 500 USD. Several efforts have been deployed to ensure a balance between low cost and high performance solutions to manage this deluge of data. Several initiatives have been deployed to avoid turnkey parallel platforms, by constructing a series of commodity DBMS systems, "glued together" with a thin partition-aware wrapper layer. Google's search clusters reportedly consist of tens of thousands of shared-nothing nodes, each costing around $700 [95]. Such clusters of PCs are frequently termed *grid computers*. Other initiatives combine the cheapest DBMS and Cloud technology. We can consider the example of *Alibaba* company. In their recent works published in VLDB 2014 [64], they propose a MySQL driven solutions to deal with the data deluge over the Cloud. The most used parallel $\mathcal{DW}$ systems are Oracle Exadata, Teradata, IBM Netezza, Microsoft SQL server Parallel data warehouse, Greenplum.

Regardless of the way in which a parallel $\mathcal{DW}$ is designed, it requires integrating our findings in the discussed phases. These steps are quite sensitive to the workload. We have already shown the strong interaction between the UQP and the horizontal partitioning selection in the context of centralized $\mathcal{DW}$. Since the partitioning is a pre-condition of any parallel database design [27] and its schema is used by all steps of the parallel $\mathcal{DW}$ life design life cycle, it is worthy to integrate the volume and sharing of queries in this design.

In this chapter, we discuss this integration. To do so, we propose and experimentally assess an innovative methodology guided by the *big query interaction* for designing Parallel *RDW*

[1]`http://www.economist.com/node/15579717`

($PRDW$) on shared nothing database cluster architecture. To explore the large number of queries, we use UQP that helps us in visualizing our workload and partitions it, if necessary, into several components to reduce the complexity of the design. Then, we focus on two major problems when designing $PRDW$ which are data partitioning and fragment allocation.
The chapter is organized as follows. Section II discusses the main alternatives in designing parallel relational$\mathcal{DW}$. Section III depicts a motivating example. Section IV contains the details of our $PRDW$ design methodology. Section V provides our experimental results obtained from testing the performance of our approach using dataset of the Star Schema Benchmark (SSB). Finally, Section VI concludes the chapter and discusses some open issues.

II. Parallel Database Design Alternatives

Based on the above discussion, we assert that parallel $\mathcal{DW}$ design can be modelled by the following tuple [37]: $< Arch, DP, DA, DR, LB >$, where: $Arch$ denotes the parallel architecture, DP the data partitioning scheme, DA the data allocation scheme, DR the data replication scheme and LB the load balancing scheme, respectively. Note that the problem corresponding to each component of the above tuple is NP-hard [12, 270].
Two main methodologies exist to deal with the problem of parallel $\mathcal{DW}$ design: *iterative design* and *conjoint design*. Iterative design methodologies have been proposed in the context of traditional distributed and parallel database design [215, 238, 208, 108]. The idea underlying this class of methodologies consists in first fragmenting the relational $\mathcal{DW}$ using any partitioning algorithm, and then allocating the so-generated fragments by means of any allocation algorithm. In the most general case, each partitioning and allocation algorithm has its *own cost model*. The main limitation of iterative design is represented by the fact that they neglect the *inter-dependency* between the data partitioning and the fragment allocation phase, respectively. Figure 5.1 **(a)** summarizes the steps of iterative design methodologies. To overcome these limitations, the combined design methodologies were proposed in [27]. They consist in merging some phases of the life cycle of the distributed/parallel life cycle. The main idea of this merging is that a phase is performed knowing the requirements of the next step. Three main variants of this methodology are distinguished (Figure 5.1 **((b),(c),(d))**. The conjoint design methodologies that have been addressed in Soumia BENKRID thesis, elaborated in our laboratory LIAS [36] and tested in Teradata machine [19].
Figure 5.1 (b) depicts an architecture, where the basic idea consists in first partitioning the $\mathcal{DW}$ using any partitioning algorithm and then determining how fragments are allocated to the nodes by also determining replication of fragments. The main advantage of this architecture is that it takes into account the inter-dependency between allocation and replication, which are closely related [27]. The main limitation is the fact that it neglects the inter-dependency between the data partitioning and fragment allocation. Figure 5.1 (c) shows a second variant of the conjoint design, in which the $\mathcal{DW}$ is first horizontally partitioned into fragments, and then fragments are allocated to nodes within the *same* phase as in [217]. After that, a replication algorithm is

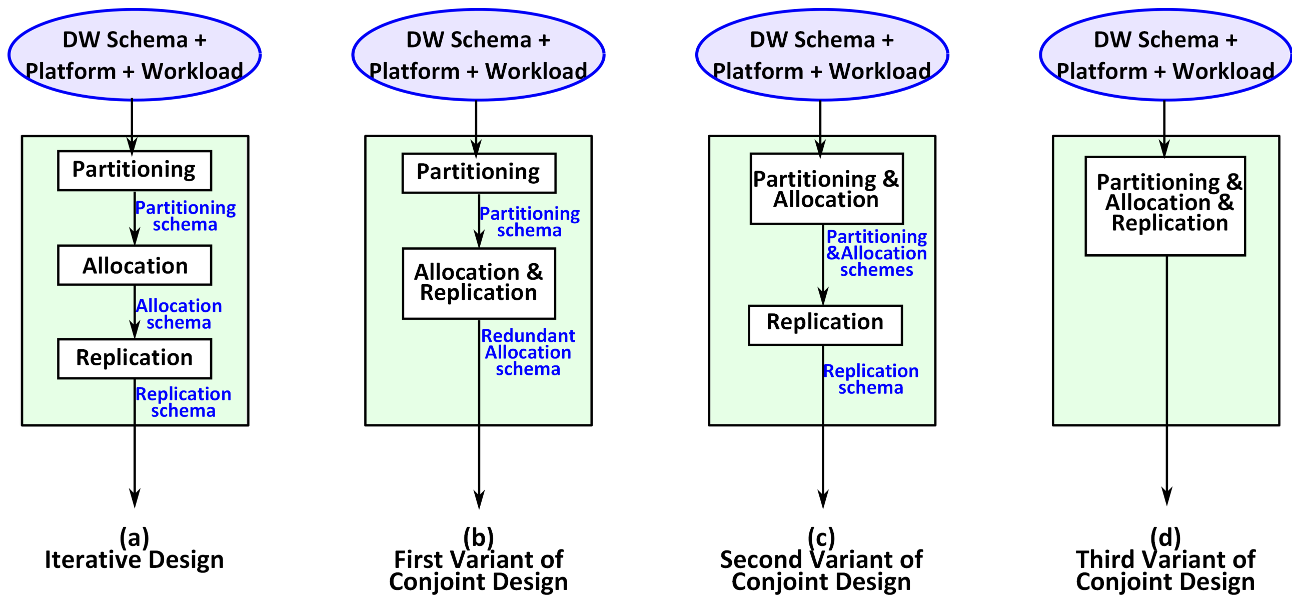

DW Schema + Platform + Workload
Partitioning
Partitioning schema
Allocation
Allocation schema
Replication
Replication schema
(a) Iterative Design
DW Schema + Platform + Workload
Partitioning
Partitioning schema
Allocation & Replication
Redundant Allocation schema
(b) First Variant of Conjoint Design
DW Schema + Platform + Workload
Partitioning & Allocation
Partitioning &Allocation schemes
Replication
Replication schema
(c) Second Variant of Conjoint Design
DW Schema + Platform + Workload
Partitioning & Allocation & Replication
(d) Third Variant of Conjoint Design

Fig. 5.1 – *Iterative and Variants of Conjoint Design Methodologies.*

used in order to determine how to allocate replicated fragments. The advantage of this variant consists in simultaneously performing the allocation phase at the partitioning time. However, the drawback of the architecture is that it neglects the closely-related dependency between the allocation and the replication process. Finally, Figure 5.1 (d) depicts an architecture such that partitioning, allocation and replication are combined into a *unified process.*
The basic difference between the four reference architectures depicted in Figure 5.1 is represented by the selection of partitioning attributes. The iterative approach determines the partitioning attributes using a cost model that neglects the inter-dependency between partitioning, allocation and replication.
Formally, our parallel $\mathcal{DW}$ design problem that considers the different phases of the life cycle can be formalized as the following *Constraint Optimization Problem.* Given:

- **Inputs:**
  -a parallel platform ($\mathcal{PP}$) with $\mathcal{P}$ nodes $\mathcal{N} = \{N_1, N_2, \ldots, N_P\}$;
  - a relational $\mathcal{DW}$ modeled according to a star schema and composed of one fact table $\mathcal{F}$ and d dimension tables $\mathcal{D} = \{D_1, D_2, \ldots, D_d\}$ (similarly to [171], we suppose that all dimension tables are replicated over the nodes of the parallel platform and can fit in main memory);
  -a set of star join queries $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_L\}$ to be executed over $\mathcal{PP}$, each query Q_l characterized by an access frequency f_l.

- **Outputs:**

-data partitioning schema;
-data allocation strategy of fragments over processing nodes;
-query processing strategy over processing nodes.

- **Constraints:**
  -the *maintenance constraint* $\mathcal{W}$ representing the number of fragments that the designer considers relevant for his/her target allocation process (note that this number must be greater than the number of nodes, i.e. $\mathcal{W} \gg P$)
  -the *replication constraint* $\mathcal{R}$, such that $\mathcal{R} \leq \mathcal{P}$, representing the number of fragment copies that the designer considers relevant for his/her parallel query processing;
  -the *attribute skewness constraint* Ω representing the degree of non-uniform value distributions of the attribute sub-domain chosen by the designer for the selection of the partitioning attributes;
  -the *data placement constraint* α representing the degree of data placement skew that the designer allows for the placement of data;
  -the *load balancing constraint* δ representing the data processing skew that the designer considers relevant for his/her target query processing.

The problem of designing a $\mathcal{DW}$ over the parallel platform (PP) consists in *fragmenting the fact table $\mathcal{F}$ based on the partitioning schemes of some/all dimension tables into $\mathcal{N}, N >> P$ fragments and allocating them and the replicated fragments over different nodes of the parallel platform such that the total cost of executing all the queries in $\mathcal{Q}$ can be minimized while all constraints of the problem are satisfied.*
In this thesis, we focus on the variant of $\mathcal{DW}$ parallel design in which the partitioning and allocation phases are combined. This design has been called $\mathcal{F}$ &$\mathcal{A}$ design [27]. In the next section, we show the impact of considering query interaction on the horizontal partitioning.

III. Motivating Example

, To the best of our knowledge, even some studies of *PRDW* design considering the interaction between queries exist, but none considers the *Big Interacted Queries Phenomenon.* In this section, we study the impact of this phenomenon in two phases: data partitioning and fragment allocation. To illustrate this finding, let us consider an example with a *PRDW* with the following configuration:

- A star schema composed of one fact table: *Lineorder* and four dimension tables : *Supplier*, *Dates*, *Customer*, and *Part*.
- On the top of this schema, 10 star queries $\{Q_1, .., Q_{10}\}$. The unified plan is described in Figure 5.2-a. Three types of nodes of this plan are distinguished: a selection operation, denoted by S_i (with the form *Attribute* θ *value*, where *Attribute* $\in$ *Table*, $\theta \in \{=, <, >$

$, ...\}$ and $Value \in Domain(Attribute)$), a join operation, denoted by J_i and a projection operation denoted by π. We note that seven (7) selections and joins are identified.

- The candidate attributes with their respective domains to perform the partitioning are given in Figure 5.2-c.
- A database cluster with four nodes $N = \{N_1, .., N_4\}$.

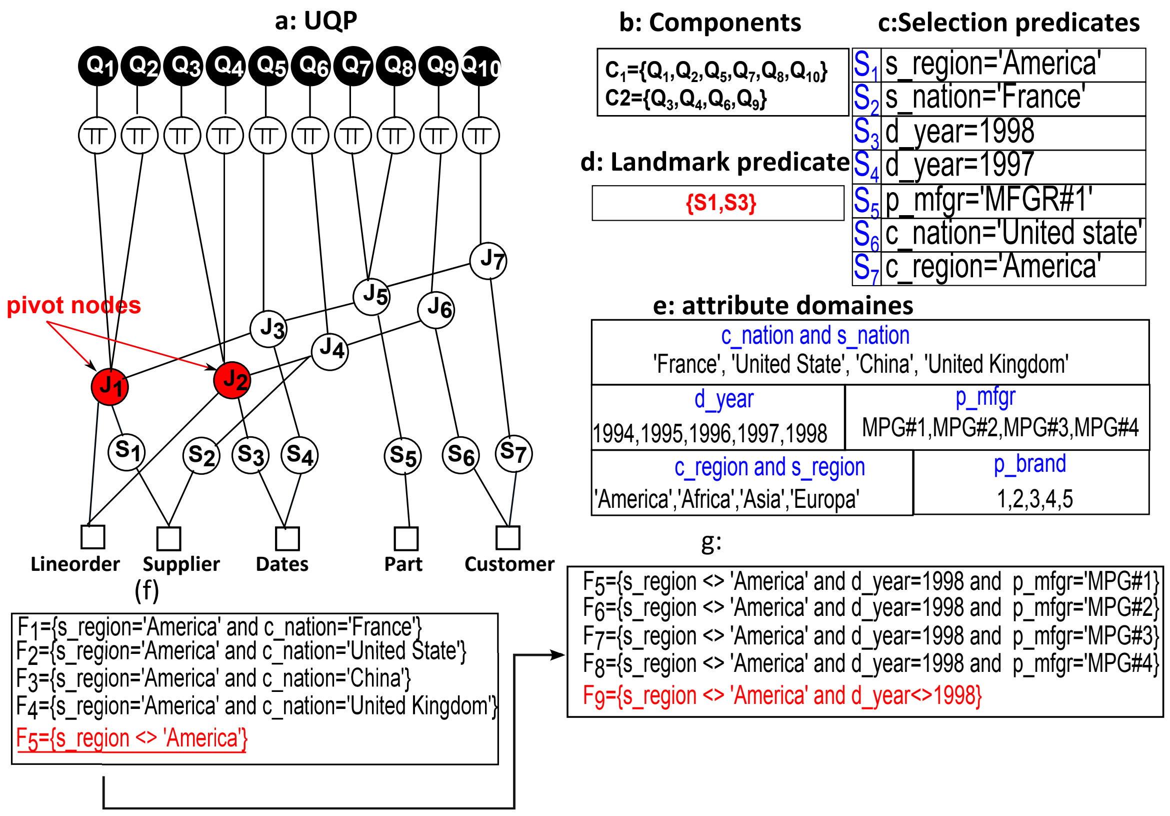

a: UQP
Q1 Q2 Q3 Q4 Q5 Q6 Q7 Q8 Q9 Q10
π
J7
J5
J6
J3
J4
pivot nodes
J1
J2
S1 S2 S3 S4 S5 S6 S7
Lineorder
Supplier
Dates
Part
Customer
b: Components
C1={Q1,Q2,Q5,Q7,Q8,Q10}
C2={Q3,Q4,Q6,Q9}
c:Selection predicates
S1 s_region='America'
S2 s_nation='France'
S3 d_year=1998
S4 d_year=1997
S5 p_mfgr='MFGR#1'
S6 c_nation='United state'
S7 c_region='America'
d: Landmark predicate
{S1,S3}
e: attribute domaines
c_nation and s_nation
'France', 'United State', 'China', 'United Kingdom'
d_year
1994,1995,1996,1997,1998
p_mfgr
MPG#1,MPG#2,MPG#3,MPG#4
c_region and s_region
'America','Africa','Asia','Europa'
p_brand
1,2,3,4,5
(f)
F1={s_region='America' and c_nation='France'}
F2={s_region='America' and c_nation='United State'}
F3={s_region='America' and c_nation='China'}
F4={s_region='America' and c_nation='United Kingdom'}
F5={s_region <> 'America'}
g:
F5={s_region <> 'America' and d_year=1998 and p_mfgr='MPG#1}
F6={s_region <> 'America' and d_year=1998 and p_mfgr='MPG#2}
F7={s_region <> 'America' and d_year=1998 and p_mfgr='MPG#3}
F8={s_region <> 'America' and d_year=1998 and p_mfgr='MPG#4}
F9={s_region <> 'America' and d_year<>1998}

***Fig. 5.2** – UQP Representation example*

By examining the UQP, generated using partitioning hypergraph, we figure out that the queries can be regrouped into two main groups called *components* (Figure 5.2-b): $C_1 = \{Q_1, Q_2, Q_5, Q_7, Q_8, Q_{10}\}$ and $C_2 = \{Q_3, Q_4, Q_6, Q_9\}$. Each component contains a set of queries that share at least one join operation. The first shared join node is called the *pivot node* of the component and the set of selection predicates of its branches called *set of landmark predicates*. In our case, we have two pivot nodes (J_1 for the component C_1 and J_2 for the component C_2) with their sets of landmark predicates are $\{s\_region = "America"\}$ and $\{d\_year = 1998\}$, respectively.
Note that the *pivot node* notion is quite important, since it guides the partitioning process of a component by the means of its *set of landmark predicates*. A landmark predicate partitions

its corresponding table (a leaf node of the plan) into two partitions, one with all instances satisfying the predicate (e.g., s_region = "*America*") and another representing the *ELSE* partition ($s\_region \neq$ "*America*"). Note that the partitioning of dimension tables will be *propagated* to partition the fact table. This partitioning is called *derived partitioning* [27]. This initial partitioning schema of a component will be refined by considering other predicates of the set of landmark predicates and other predicates do not belonging to pivot node (e.g., S_4). In the case, where the obtained fragments of each component are allocated over these nodes in *round robin fashion*, the maximum number of algebraic operations will be executed over all the cluster nodes. Therefore, queries of a given partitioned component (e.g., C_1) will get benefit from this process.
The remaining fragment of the initial component (F_5 in our example) will be concerned by the partitioning process of the component C_2. The above partitioning and allocation reasoning applied to C_2 (Figure 5.2-g). Note that the queries of the component C_2 need fragments of the component C_1.
A partitioning order has to be defined among components. In our proposal, we favorite the component involving most costly queries. The components with an empty landmark predicate set are not considered for the partitioning process.

IV. UQP as a service for designing PRDW

In this section, we consider that data partitioning and allocation are performed iteratively and exploits the query interaction. First of all, we start with the data partitioning phase then the allocation of the obtained fragments.

IV.1. The Data Partitioning Phase

The problem of data partitioning in the context of $PRDW$ may be formalized as follows:

- **Inputs :**
  - a $\mathcal{DW}$ with fact table F and d dimension tables $\{D_1, .., D_d\}$,
  - a workload $\mathcal{Q}$ of n queries, $\mathcal{Q} = \{Q_1, .., Q_n\}$,
  - a DataBase Clusters (DBC) with M processing nodes, $N = \{N_1, .., N_M\}$;
- **Outputs :** - a data partitioning schema.
- **Constraints :** - disk spaces of nodes, and maintenance cost representing the maximum number of generated fragments that the designer want to have (W).
- **Objective:** - minimize the make-span of the workload $\mathcal{Q}$ over DBC and satisfy the cited constraints.

IV.2. Partitioning Algorithm Description

The algorithm aims at ensuring equitable node processing by generating effective data partitioning and fragments allocation schemes using query interaction. The partitioning process is guided by sharing expensive operations (e.g. joins), where data needed to process these operations (determined using landmarks predicate), will be partitioned and distributed over cluster-nodes.
The sharing operations are described using the UQP where they are regrouped in several components. Thus, the partitioning is driven by one operation shared by one or more components (component area). Each component area is becoming the subject of a sub-partition that optimizes a sub-set of queries presented by the component area. In the UQP, the components are disjoint which allows subdividing the partitioning problem into a set of independent partial data partitioning and the partitioning follows an incremental process. It starts by the weighted component area and so on for the other. The flowchart for our proposed *PRDW* design methodology is sketched in Figure 5.3 (called BQ-Design: BigQuery design).

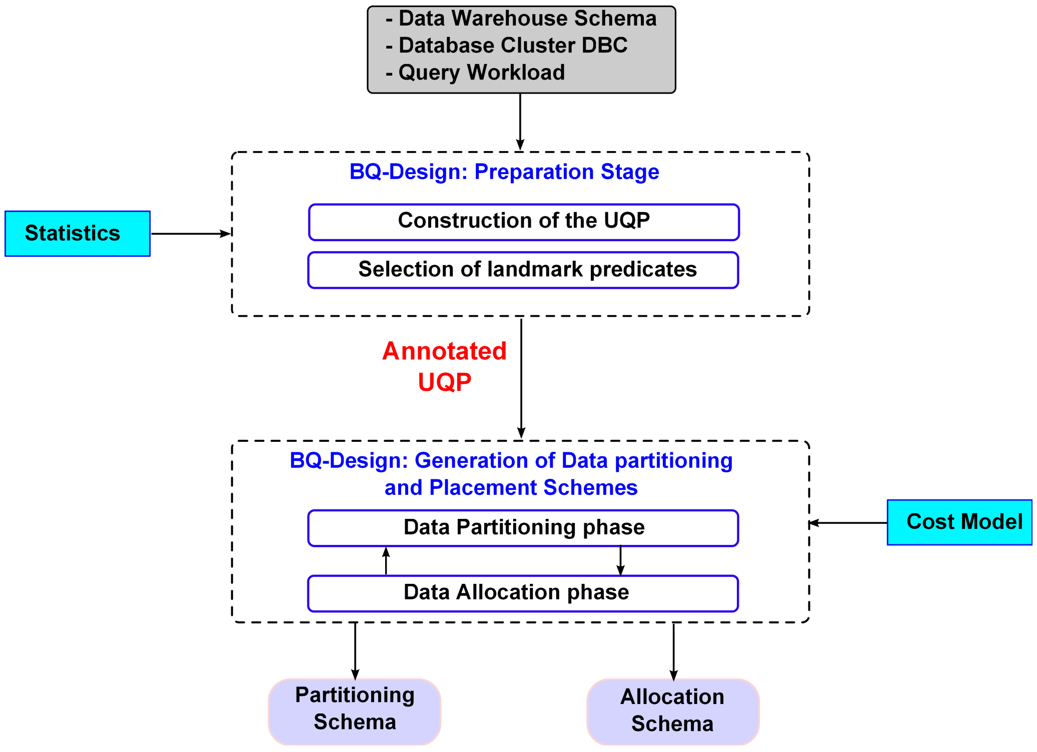


Fig. 5.3 *– Flowchart for PRDW design methodology*

The main steps of *PRDW* design methodology are:

- **Generation UQP:** to generate the UQP for a workload of queries, we use $\mathcal{HDP}$-oriented UQP as explained in the Chapter 3.

- **Identifying of landmark predicates:** an UQP composed of a set of components, in which each component has pivot node that corresponds to the first join operation, and the landmark predicate corresponds to selection predicate of its branch.
- **Annotation of UQP:** is a operation to represent each component of the UQP by a landmark predicate.
- **Generation of partial data partitioning schema:** is an operation to generate a partial partitioning schema of one component (sub-set of queries).
- **Generation of partial fragments allocation:**: is an operation to generate a partial fragments allocation of one component (sub-set of queries).

IV.2.1. Annotation of UQP

Each *Landmark Predicate*, noted by *lp*, represents one or more components but each component is represented by only one landmark predicate. We call *Components Area* of *lp* (CA_{lp}), all components represented by *lp*. Hence, the query workload can be represented by a set of *Landmark Predicates*, called LP.
Each *landmark predicate lp*, is annotated by a weight $w(l_p)$ which equals to the sum of processing cost of all queries of their component area (CA_{lp}). Let *isComponentArea*(Q_i, l_p) a boolean function that returns 1 if Q_i is in the component area of l_p, 0 otherwise. $W(l_p)$ is defined as follows:

$$W(l_p) = \sum_{i=1}^{L} Cost(Q_i) \times isComponentArea(Q_i, l_p) \tag{5.1}$$

The elements of landmark predicate list LP, are sorted in the descending order of their landmark predicates weights $W(l_p)$.

IV.2.2. Generation of Partial Data Partitioning Schema

We use an incremental partitioning, where in each step, we generate a data partition schema for a subset of queries presented by component area. For each component area, we start by a fragment preparation that splits data into two fragments, using a landmark selection predicate of the component area. The first is the initial fragment (noted $Candidate_{Frag}$) that will be used to generate the partial data partition schema and the second is the remaining partition (noted $NoCandidate_{Frag}$), which will be kept the next partitioning step. We note that the first fragment is all data to be partitioned (e.g., fact table). In the following, we give the steps of our *Partial Data Partitioning*:

- **Preparation of attributes partitioning.** We identify all possible attributes which have more than value in their sub-domains in the definition of the fragment object to the partial partition. These attributes are divided into two categories: the first category called *first candidates* (FC) contains attributes not used by the queries in the *component area*, and the remaining attributes are called *second candidates* (SC). The partition starts using first candidates (FC) to split the fragment such that each query can be processed by the maximum of cluster nodes. When there is none attribute in FC, the partition process uses SC to select attribute partition.

- **Iterative fragment splitting.** The partial partitioning is a sequence of splitting operations, where each split is applied on one fragment to produce two fragments. So, the partial partition schema starts with on fragment and it increases by one in each splitting. Each split is applied on the most voluminous fragment. The volume is defined by the *selectivity factor* that equal the multiplication of selectivity factors of all attributes that participate on the definition of the fragment, and the selectivity of attribute is the sum of the selectivity of their sub-domain. The splitting attribute corresponds to the attribute that has the minimum values in its sub-domain, and the splitting operation divides the sub-domain of n values into two sub-domains, each one has $n/2$ values if n is pair else one sub-domain has $(n+1)/2$ values and the other has $(n-1)/2$ values. The splitting process continues until produce m fragments (m is number of cluster nodes) are produced.
- The previous steps are repeated for all *component areas.*

IV.2.3. Generation of Partial Fragment Allocation Schema

To allocate each Partial Data Partitioning Schema generated, we use a *round robin placement*, in which the allocation unit is a fragment. The placement algorithm is simple: we affect the voluminous fragment in the node that has the maximum free space, and so on.
After placement process, the system verifies the maintenance constraint. With the existence of big-queries (big number of components), the number of fragments can be superior to the predefined threshold W. In this case, it will be necessary to merge fragments in each cluster node. The merging starts with the smallest fragments.
Once the data placement schema is generated, we compute the cost of executing the set of queries Q over M nodes in terms of number of inputs-outputs.

V. Experimental Evaluation and Analysis

This section reports the results of an experimental evaluation of our proposed approach. Our simulation conducted on a computer with 3.4GHz Intel(R) Core(TM) i7-3770 equipped with 8GB RAM. Algorithms were carried out in Java programming language. For the hardware architecture, we simulate a homogeneous Database Cluster of 8 to 128 nodes. The dataset from the *Star Schema Benchmark (SSB)* [194], it has different sizes (from 100 GB to 2TB).

Algorithm 9: Partial data partition algorithm (*partialPartition*)

```
1: Input : $lp$; { landmark predicate}
2: Input : $CA_{lp}$; {List of components each one contains a set of queries}
3: Input : $fragment$; {The candidate fragment}
4: Input : $m$; {number of cluster nodes}
5: Output: $PPS$; {partial partition schema}
6: Output: $noncandidate$; {A fragment for the next partial partition }
7: $(candidate, noncandidate) \leftarrow split(fragment, lp)$;{Split the fragment using $lp$}
8: $(FC, SC) \leftarrow preparAttribute(candidate, CA_{lp})$;{select candidate attributes}
9: $nbrFrag \leftarrow 1$;{number of fragment in $PPS$ }
10: $PPS \leftarrow candidate$;{initiate the $PPS$}
11: while $nbrFrag < m$ do
12:    $frag \leftarrow getBigFragment(PPS)$;{get the volumounus fragement in $PPS$}
13:    if $|FC| > 0$ then
14:       $att \leftarrow getSmallAttribut(FC, frag)$;{get the small candidate attribut}
15:    else
16:       $att \leftarrow getSmallAttribut(SC, frag)$;
17:    end if
18:    $(frag_1, frag_2) \leftarrow split(frag, att)$;{Split the fragment using attribut $att$}
19:    $add(frag_1, PPS)$; {add $frag_1$ to $PPS$}
20:    $add(frag_2, PPS)$; {add $frag_2$ to $PPS$}
21:    $delete(frag, PPS)$; {delete $frag$ from $PPS$}
22:    $nbrFrag \leftarrow nbrFrag + 1$;
23: end while
```

Several workloads randomly generated, are used (their sizes vary from 32 to 10 000 queries). In all experiments the query processing are estimated as the number of $\mathcal{I}/\mathcal{O}$. Our tests have two objectives: (i) compare the performance of this approach with a recent good approach of PRDW designing [27] and (ii) check the quality and scalability where workload and databases size change.

V.1. BQ-Design vs. $\mathcal{F}$ &$\mathcal{A}$

As a first experiment, we study the performance of our proposed methodology BQ-Design, compared against $\mathcal{F}$ &$\mathcal{A}$ approach [27], where allocation phase is done at partitioning phase; the fragmentation phase uses a Genetic Algorithm and the allocation phase is based in innovative matrix-based formalism and a related fuzzy k-means clustering. For each PRDW design methodology, we set the fragmentation threshold W to 500 and we measured the query execution time versus the variation of the number of database cluster nodes M over the interval

[8:32]. Figure 5.4 shows the results obtained and confirms to us that the BQ-Design approach outperforms the $\mathcal{F}$ &$\mathcal{A}$ one; since offers 18%-39% savings. The $\mathcal{F}$ &$\mathcal{A}$ approach has proven that is practical and reliable a real-life parallel processing database system, Teradata DBMS [27].

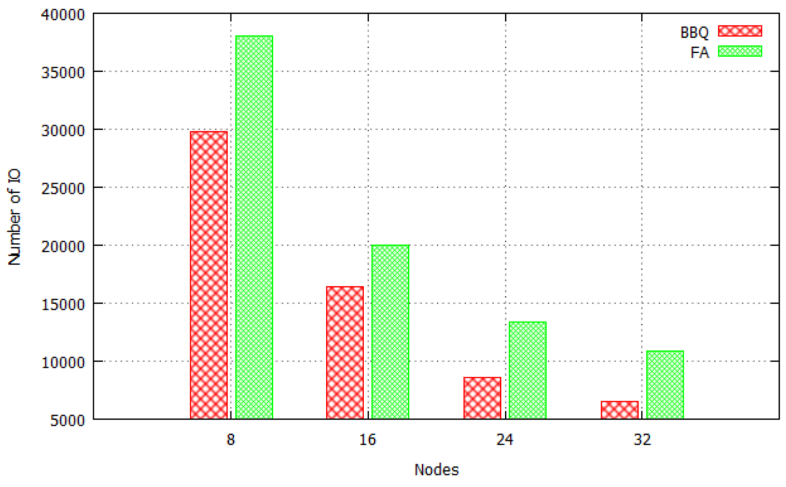


Fig. 5.4 *– Computational Overhead Performance of BQ-Design against $\mathcal{F}$ &$\mathcal{A}$ Design*

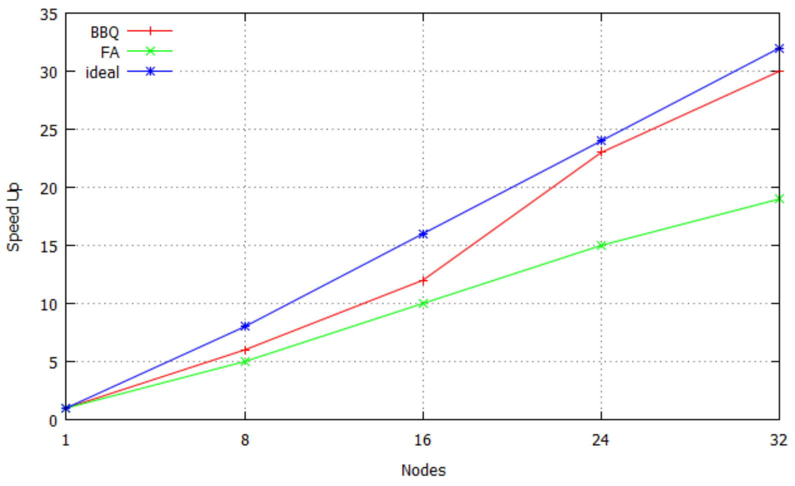


Fig. 5.5 *– Performance of BQ-Design against $\mathcal{F}$ &$\mathcal{A}$ Design Approach*

To check BQ-Design quality when changing the number of cluster, we conduct the same experiment, we calculate the speed up factor for each approach. Figure 5.5, shows that this factor is not linear since BQ-Design does not ensure a load balancing processing. To determine the cause of this imbalance processing, our second experiment is the study of the data placement distribution of BQ-Design. To this end, we fix the number of nodes to M=8 and we check the amount of data stored in each node. As sketched in the figure 5.6, both approaches ($\mathcal{F}$ &$\mathcal{A}$, BQ-Design) suffers by data placement skew with 43% and 48% respectively. This is due to the selectivity skew. Indeed, both approaches are based on the multi-level partitioning that is based on the splitting of the attribute's domain. This type of splitting depends on the nature of the distribution of the attribute's domain.

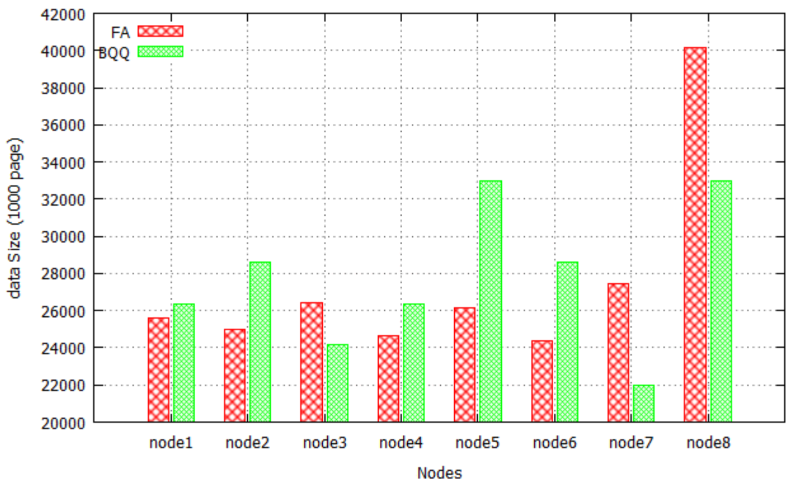


Fig. 5.6 – BQ-Design Data Placement Distribution

In the third experiment, we focused on the effect of the size of the workload on the performance of BQ-Design comparing with $\mathcal{F}$ &$\mathcal{A}$ approach. Here, we fixed the number of nodes to $M = 16$ and we ranged the workload size over the interval [30 :60] in order to study how the BQ-Design query performance varies accordingly. Figure 5.7 gives more detail from the execution of a workload of 30 queries under 16 processing. For BQ-Design, all queries benefit from the partitioning schema but for $\mathcal{F}$ &$\mathcal{A}$ only queries using the fragmentation attributes benefit from the partitioning.

V.2. BQ-Design Scalability

In this step, the experiments show the impact of the volume of queries and the size of the $\mathcal{DW}$ on the quality BQ-Design. We used different workload sizes (from 100 to 10 000 queries), ranged the number of nodes over the interval [8:128], and varied the size of data warehouses (100GB, 1TB, and 2TB). We note that the following tests are applied only by BQ-Design tool because the $\mathcal{F}$ &$\mathcal{A}$ approach does not scale when many queries are used.

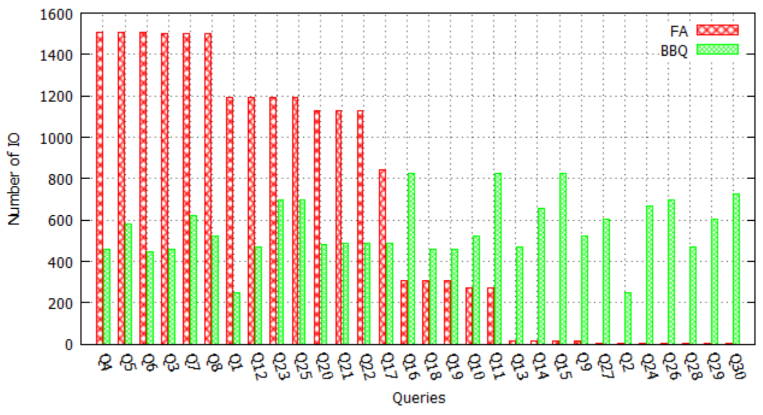


Fig. 5.7 – Queries Makspen

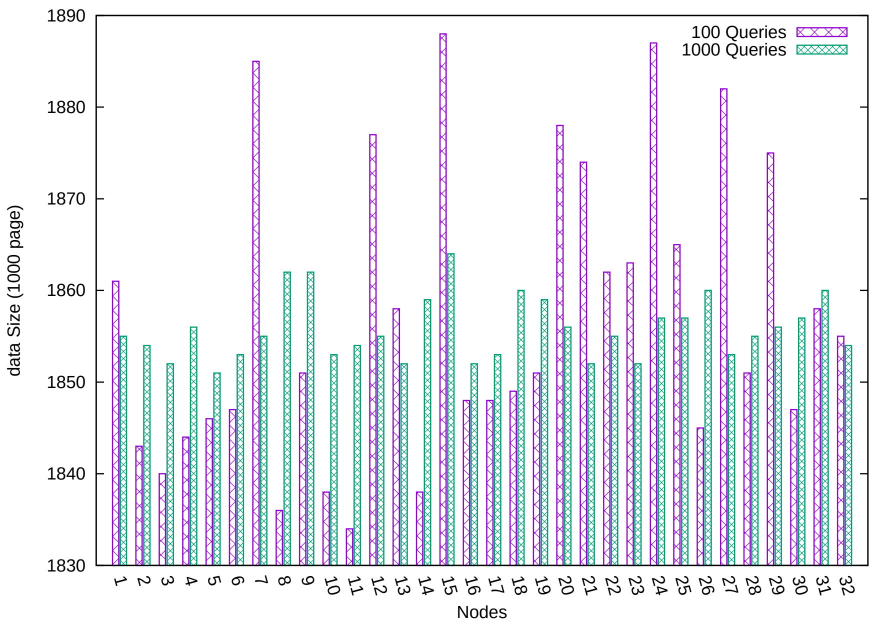


Fig. 5.8 – The impact of workload size on data distribution quality

The first experiments studies the impact of the workload size on the quality of data distribution. To do so, we used 32 cluster nodes, dataset with 2 TB and two workloads with 100 and 1000 queries respectively. By analyzing the obtained results in Figure 5.8, we figure out that the data placement skew of BQ-Design is improved when considering more queries. The second experiments aim at checking the scalability of BQ-Design to meet increasing workload and database sizes. Intensive tests were applied using different configurations by varying the number

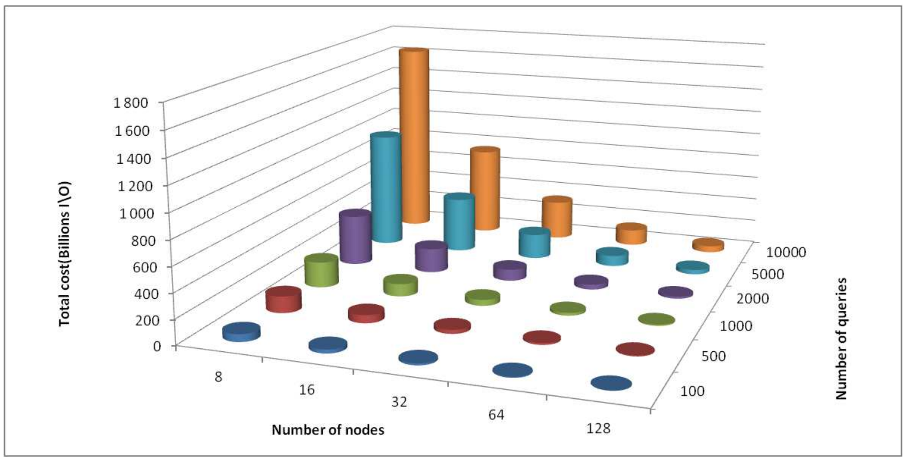


***Fig. 5.9** – Scale-up of BQ-Design when workload size increase*

of cluster nodes([8:128]), number of queries (from 100 to 10000 queries) and the size of dataset (100GB, 1TB and 2TB). In each configuration, the simulator estimates the total cost of query processing. To check the scalability of BQ-Design when database size increases, we have fixed the number of query on 1000 queries. Figure 5.10, shows that BQ-Design can scale-up where data size increase. To check the scalability of BQ-Design when workload size increases, we

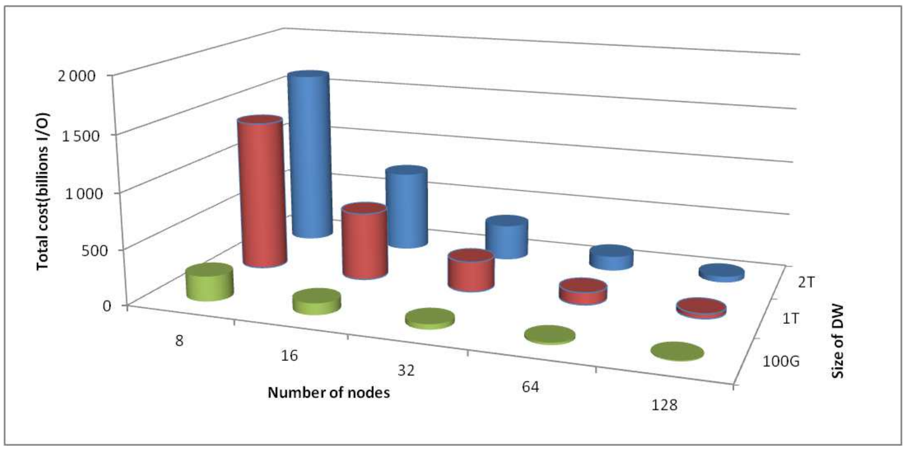


***Fig. 5.10** – Scale-up of BQ-Design when database size increase*

have fixed the size of dataset on 2TB. Figure 5.9, shows that BQ-Design can scale-up where workload size increase.

VI. Conclusion

In this chapter, we motivated the consideration of the query interaction in designing parallel $\mathcal{DW}$ under concurrent analytical queries. We proposed a new scalable $PRDW$ designing approach that can generate effective data partition and data placement schemes. The main steps of our approach are: (i) capturing of interaction among queries ($\mathcal{HDP}$-oriented UQP). (ii) Generation of landmark predicates that performed by using connected components that compose the UQP. (iii) Elaboration of modular data partition and data allocation, guided by the components of UQP.
Our approach has been compared against the most important state of art works and the obtained results show the efficiency and effectiveness of our approach. It has been tested under big size workload (10 000 queries) and shows its scalability.

Chapter 6

Big-Queries framework

Contents

Abstract

In this chapter, we present **Big-Queries**, a database design advisor that assists a DBA during her/his administration tasks when dealing with interaction numerous queries. Specifically, Big-queries provides the following functionalities: (1) scale generation of an unified query plan oriented for a specific optimization structures (materialized views and horizontal data partitioning). The generation of the UQP starts by transforming a workload of queries into an hypergraph and applying different partitioning algorithms with injecting of $\mathcal{OS}$ knowledges. (2) Generation of near-optimal materialized basing on the UQP and (3) generating horizontal data partitioning. (4) Use of query interaction to design parallel relational data warehouse

I. Introduction

To assist administrators in their physical design tasks, several commercial advisor tools were proposed to suggest recommendations to DBA for choosing optimization structures. We can cite, for instance, the Microsoft AutoAdmin, Database Tuning Advisor (which is part of Microsoft SQL Server 2005) [4], the DB2 Design Advisor [257] and the Oracle Advisor [16]. Recently, The Azure SQL Database[1] provides recommendations for creating and dropping indexes, parameterizing queries, and fixing schema issues. The recommendations that are best suited for a query workload are recommended. Other advisors have been also developed by academicians. Parinda ([176], developed by Data-Intensive Applications and Systems Laboratory of the EPFL School, Lausanne, Switzerland, and SimulPh.D. [21, 23], developed in our laboratory LIAS/ISAE-ENSMA, and RITA (An Index-Tuning Advisor for Replicated Databases) [249] – a joint work between Oracle, Google and the University of California, Santa Cruz and EPFL, etc. These advisors are DBMS-dependent and mainly concentrated in developing self-managing systems that can relegate many of the database designer's more mundane and time-consuming tasks [282]. These tools use cost models of their optimizers. Some of these tools focus on one $\mathcal{OS}$ such as RITA elaborated for indexes, whereas the majority deals with several $\mathcal{OS}$. Table Table:advisors recapitulates these advisors in terms of used $\mathcal{OS}$.
These tools intensively use cost models to suggest recommendations. They are not designed to deal with volume of interacted queries. To overcome this limitation, we propose a new tool, called Big-Queries that exploits the query interaction of numerous and complex queries. The query interaction presented by an unified query plan (UQP), that is generated thanks to hypergraph partitioning techniques. Big-Queries has two main objectives: the first objective consists in recommending materialized views and horizontal data partitioning configurations for a given workload. (2) The second objective consists in taking profit from our experience in centralized database in terms of query volume and sharing experience and deploying it in the context of parallel $\mathcal{DW}$ design.

Advisors	**Supported Optimization Structures**
SQL Database Tuning Advisor	Partitioning; Materialized views, Indexes
Oracle SQL Access Advisor	Partitioning; Materialized views; Indexes
DB2 Index Advisor	Partitioning; Materialized Views; Indexes; Clustering
Parinda	Partitioning; Indexes
SimulPh.D.	Partitioning; Materialized views; Indexes

Table 6.1 *– Presentation of well-known administration advisors*

Big-queries plays the role of simulator that uses mathematical cost models (developed in Appendix 1) to quantify the quality of the different recommendations. This chapter gives an overview of **Big-queries** tool, its different components and examples of its usages.

[1] `https://azure.microsoft.com/en-us/documentation/articles/sql-database-advisor/`

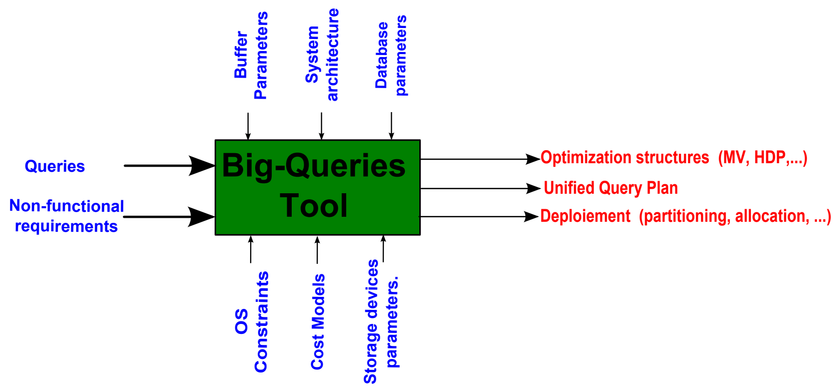

Buffer Parameters
System architecture
Database parameters
Queries
Non-functional requirements
Big-Queries Tool
Optimization structures (MV, HDP,...)
Unified Query Plan
Deploiement (partitioning, allocation, ...)
OS Constraints
Cost Models
Storage devices parameters.

***Fig. 6.1** – Functionalities of Big-Queries Tool*

This chapter is structured as follows. Next section presents the architecture of Big-queries II. In the section III, we describe the main modules of the framework. Section IV gives a technical description of the frameworks. Finally, Section V concludes this chapter.

II. System architecture

In this section, we describe the architecture of Big-Queries. It is composed of four components: *feature extractor*, UQP *generator*, *physical design advisor* and *deployment advisor*. The feature extractor gets all needed parameters like database parameters. The UQP generator transforms query interaction of workload in UQP, using a set of tools and algorithms inspired from hypergraphs theories. The physical design advisor is a set of tools that help the DBA to select $\mathcal{OS}$ following non-functional requirements and it estimates the quality of a schema before their deployment in real system. The deployment advisor is a set of tools that help the DBA to propose a deployment schema (data partitioning and allocation) tacking account many parameters like the processing nodes, parallel execution, etc.

The framework architecture of our demonstration is described in Fig6.2. In the following sections, we will briefly present our underlying models.

II.1. Features Extractor

Features extractor allows to get databases parameters, system parameters (storage device, buffer, processing nodes, etc), related query properties (selectivity, operators, tables), $\mathcal{OS}$ constraints (injected by the DBA). These properties are extracted automatically using databases statistical model, parsing queries and different files.

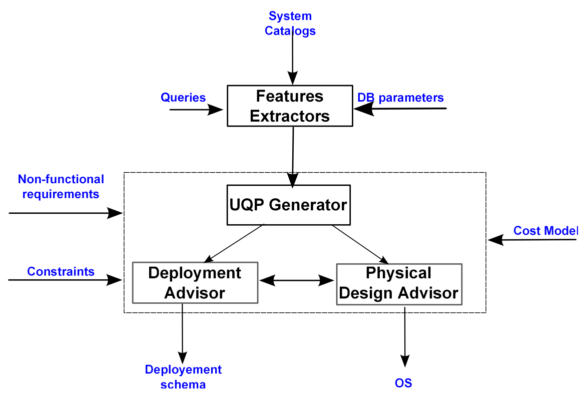


Fig. 6.2 – Big-Queries system architecture

II.2. UQP Generator

The UQP generator is used to transform a set of queries into an unified query plan. It uses a set hypergraphs algorithms (in our case, we have used multi-level hypergraph partitioning algorithms [148]), UQP generator generates an unified query plan oriented for a specific $\mathcal{OS}$ (See Chapter 3). It uses a mathematical model called (cost model), to estimate the cost of query, logical operation, workload of queries or related $\mathcal{OS}$ costs. The cost model uses parameters of queries, system catalog (Buffer, disk space, processor capability, main memory characteristics, system architecture, etc), See Annexe annexe02. Big-queries can generate the UQP in specific format that can be visualized directly Cytoscape Plug-in [2] as show in the Figure 6.3.

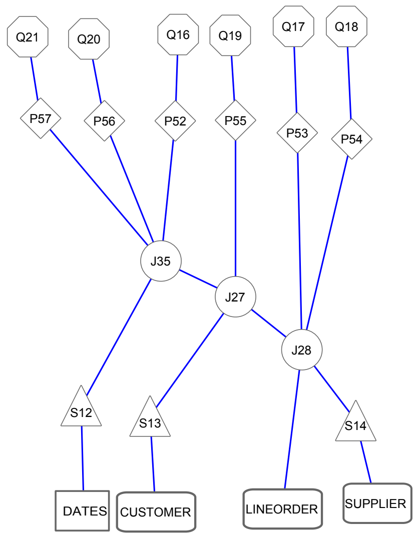


Fig. 6.3 – Example of an UQP of an component of workload

[2]http://www.cytoscape.org

II.3. Physical Design Advisor

The physical design advisor takes the advantages of query interaction represented by an UQP to propose near-optimal of two optimization structures: $\mathcal{MV}$ and $\mathcal{HDP}$, under three scenarios:

- static selection of $\mathcal{MV}$ under constraints;
- dynamic selection of $\mathcal{MV}$ with scheduling of queries;
- static selection of horizontal data partitioning.

For each optimization structure, the physical design advisor can give an estimation of the execution time within that structure. This assists DBA to take a decision about its selecting or not.

II.4. Deployment Advisor

Deployment advisor proposes a data partitioning and data allocation for designing Parallel *RDW* (*PRDW*) on shared nothing database cluster architecture. It allows simulating the quality of parallel $\mathcal{DW}$ before their implementation in real machine.

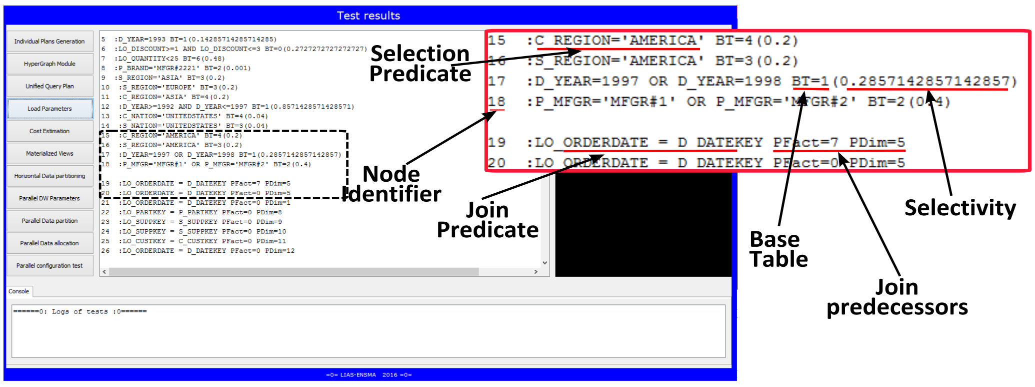
III. System Modules

We develop our Big-queries is composed of a set of independent modules to integrate any algorithm. This means that, the DBA can plug any $\mathcal{OS}$ selection algorithm for the studied structures (UQP, $\mathcal{MV}$, $\mathcal{HDP}$ schema, allocation schema). All these modules will be discussed in the next sections.

III.1. Query parser module

In this module, users can give either a single SQL query or workload to be executed. Queries supported vary from simple transactional operations to more complex reporting operations involving several large size tables. The parser takes as an input a text written in a specific language such as SQL and converts it into a parsed tree. Figure 6.4 shows an example of parse tree. The parsed tree is done by:

- a syntactic analysis to give the basic grammar elements of the text query in the form of a parsed tree; The leaves nodes of the parsed tree represent the atomic elements. They are the lexical elements of the query such as keywords (e.g., *SELECT*, *WHERE*, *FROM*), names of relations and attributes, constants, logical and arithmetics operators, etc. The other nodes of the parsed tree represent a composite element formed by many atomic elements like *Condition*, other sub query.



```
5  :D_YEAR=1993 BT=1(0.14285714285714285)
6  :LO_DISCOUNT>=1 AND LO_DISCOUNT<=3 BT=0(0.2727272727272727)
7  :LO_QUANTITY<25 BT=6(0.48)
8  :P_BRAND='MFGR#2221' BT=2(0.001)
9  :S_REGION='ASIA' BT=3(0.2)
10  :S_REGION='EUROPE' BT=3(0.2)
11  :C_REGION='ASIA' BT=4(0.2)
12  :D_YEAR>=1992 AND D_YEAR<=1997 BT=1(0.8571428571428571)
13  :C_NATION='UNITEDSTATES' BT=4(0.04)
14  :S_NATION='UNITEDSTATES' BT=3(0.04)
15  :C_REGION='AMERICA' BT=4(0.2)
16  :S_REGION='AMERICA' BT=3(0.2)
17  :D_YEAR=1997 OR D_YEAR=1998 BT=1(0.2857142857142857)
18  :P_MFGR='MFGR#1' OR P_MFGR='MFGR#2' BT=2(0.4)

19  :LO_ORDERDATE = D_DATEKEY PFact=7 PDim=5
20  :LO_ORDERDATE = D_DATEKEY PFact=0 PDim=5
21  :LO_ORDERDATE = D_DATEKEY PFact=0 PDim=1
22  :LO_PARTKEY = P_PARTKEY PFact=0 PDim=8
23  :LO_SUPPKEY = S_SUPPKEY PFact=0 PDim=9
24  :LO_SUPPKEY = S_SUPPKEY PFact=0 PDim=10
25  :LO_CUSTKEY = C_CUSTKEY PFact=0 PDim=11
26  :LO_ORDERDATE = D_DATEKEY PFact=0 PDim=12
```

```
======0: Logs of tests :0======
```

```
15  :C_REGION='AMERICA' BT=4(0.2)
16  :S_REGION='AMERICA' BT=3(0.2)
17  :D_YEAR=1997 OR D_YEAR=1998 BT=1(0.2857142857142857)
18  :P_MFGR='MFGR#1' OR P_MFGR='MFGR#2' BT=2(0.4)

19  :LO_ORDERDATE = D_DATEKEY PFact=7 PDim=5
20  :LO_ORDERDATE = D_DATEKEY PFact=0 PDim=5
```

Fig. 6.4 – Example of the result of query parser

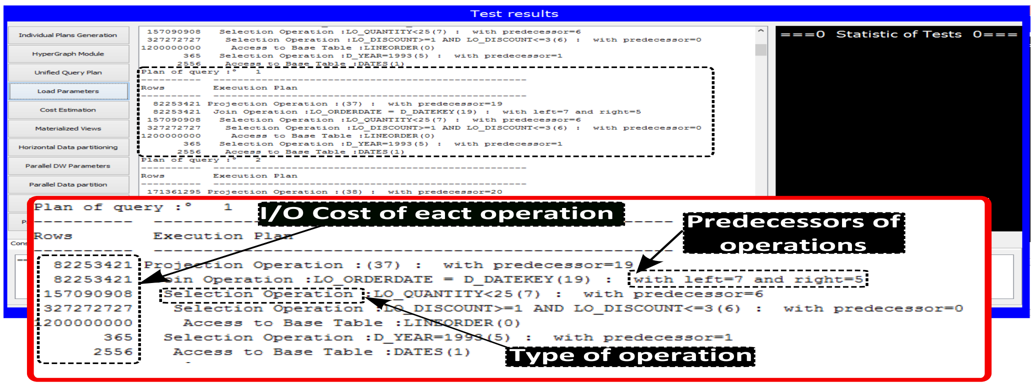
- a semantic checking of the parsed tree by verifying the used relations and attributes, query clauses, etc. The checking process follows these steps:
  - To check whether every relation, attribute, or view mentioned in the *FROM-clause* belongs to the schema of database or data warehouse (using the meta-model).
  - To check whether every attribute mentioned in the *SELECT/WHERE-clause* belongs to the relations appeared in the *FROM-clause*.
  - To check whether attributes' types are respected in the *Condition-clause*.
- a verification whether views are involved in the current query or not. Each virtual view in the query parse tree will be replaced by a sub-parse tree that represents the sub-query constructing this view. Note that in the case of materialized views, there is no substitution, because these latter are considered as base tables (relations).

III.2. Logical query plan generator

The module of selection the logical query is based on the algebraic equivalences. A query is composed of different operations which follow algebraic laws (commutativity, associativity, and distributivity), and can be applied in both directions: from left to right, and from right to left. The generation of the logical plan of a query follows two principal steps: (1) transforming the parsed tree to a logical query plan, and (2) improving the logical query plan using algebraic laws, and (3) selecting an appropriate logical plan using a cost model. Figure 6.5, shows an example of presenting a logical plan of a query.

III.3. Hypergraph module

Hypergraph module is responsible at generating an $\mathcal{OS}$ oriented UQP. It takes a set of logical operations (selection, projection and join), their corresponding queries. To ensure the

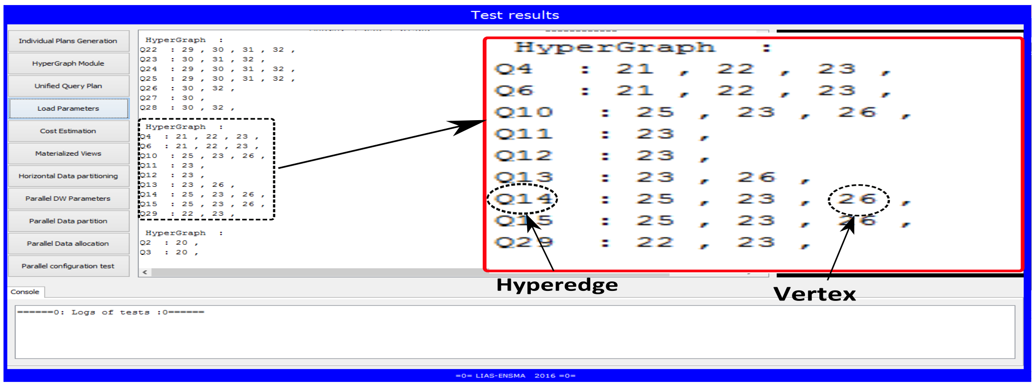


```
 157090908   Selection Operation :LO_QUANTITY<25(7) :  with predecessor=6
 327272727    Selection Operation :LO_DISCOUNT>=1 AND LO_DISCOUNT<=3(6) :  with predecessor=0
1200000000     Access to Base Table :LINEORDER(0)
       365   Selection Operation :D_YEAR=1993(5) :  with predecessor=1
      2556    Access to Base Table :DATES(1)
Plan of query :°   1
----------  ----------------------------------------------------
Rows        Execution Plan
----------  ----------------------------------------------------
  82253421 Projection Operation :(37) :  with predecessor=19
  82253421  Join Operation :LO_ORDERDATE = D_DATEKEY(19) :  with left=7 and right=5
 157090908   Selection Operation :LO_QUANTITY<25(7) :  with predecessor=6
 327272727    Selection Operation :LO_DISCOUNT>=1 AND LO_DISCOUNT<=3(6) :  with predecessor=0
1200000000     Access to Base Table :LINEORDER(0)
       365   Selection Operation :D_YEAR=1993(5) :  with predecessor=1
      2556    Access to Base Table :DATES(1)
Plan of query :°   2
----------  ----------------------------------------------------
Rows        Execution Plan
----------  ----------------------------------------------------
 171361295 Projection Operation :(38) :  with predecessor=20
```

```
Plan of query :°   1
----------  ----------------------------------------------------
Rows        Execution Plan
----------  ----------------------------------------------------
  82253421 Projection Operation :(37) :  with predecessor=19
  82253421  Join Operation :LO_ORDERDATE = D_DATEKEY(19) :  with left=7 and right=5
 157090908   Selection Operation :LO_QUANTITY<25(7) :  with predecessor=6
 327272727    Selection Operation :LO_DISCOUNT>=1 AND LO_DISCOUNT<=3(6) :  with predecessor=0
1200000000     Access to Base Table :LINEORDER(0)
       365   Selection Operation :D_YEAR=1993(5) :  with predecessor=1
      2556    Access to Base Table :DATES(1)
```

Fig. 6.5 – *Example of presenting logical plan of a query*

scalability, at beginning, the workload is represented as an hypergraph and using a set of hypergraph partitioning algorithms, the hypergraph on many sub-hypergraphs. Secondly, each sub-hypergraph (called component of queries), will be transformed separately into a unified query plan. Finally, all unified query plans will be merged into a global UQP. Figures 6.6 and 6.7 show an example of presenting respectively the sub-hypergraph and components of a workload.



```
 HyperGraph  :
Q22  : 29 , 30 , 31 , 32 ,
Q23  : 30 , 31 , 32 ,
Q24  : 29 , 30 , 31 , 32 ,
Q25  : 29 , 30 , 31 , 32 ,
Q26  : 30 , 32 ,
Q27  : 30 ,
Q28  : 30 , 32 ,
 HyperGraph  :
Q4  : 21 , 22 , 23 ,
Q6  : 21 , 22 , 23 ,
Q10  : 25 , 23 , 26 ,
Q11  : 23 ,
Q12  : 23 ,
Q13  : 23 , 26 ,
Q14  : 25 , 23 , 26 ,
Q15  : 25 , 23 , 26 ,
Q29  : 22 , 23 ,
 HyperGraph  :
Q2  : 20 ,
Q3  : 20 ,
```

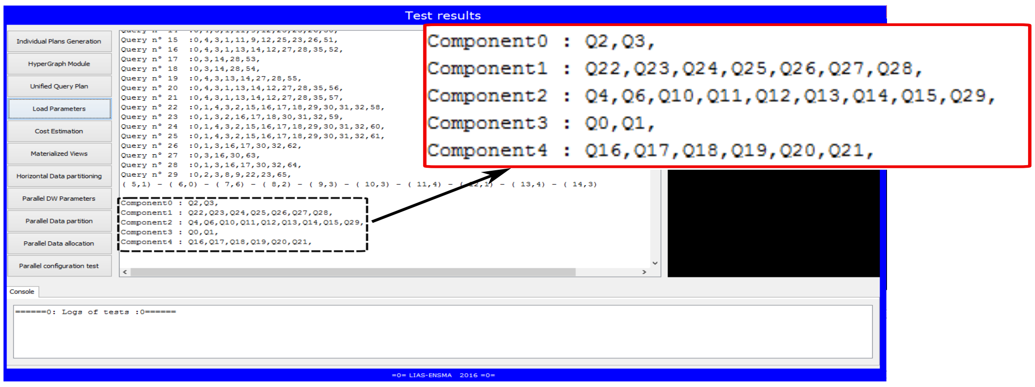
```
 HyperGraph  :
Q4  : 21 , 22 , 23 ,
Q6  : 21 , 22 , 23 ,
Q10  : 25 , 23 , 26 ,
Q11  : 23 ,
Q12  : 23 ,
Q13  : 23 , 26 ,
Q14  : 25 , 23 , 26 ,
Q15  : 25 , 23 , 26 ,
Q29  : 22 , 23 ,
```

```
======0: Logs of tests :0======
```

Fig. 6.6 – *Example of presenting an hypergraph of workload*

III.4. Query processing cost estimator

The module *query processing cost estimator*, allows estimating the cost of query or set of queries taking account or not of optimization structures (materialized views and horizontal data partitioning). To estimate query processing cost, this module needs to rewrite queries following selected optimization structures.
In the relational model, the query is processed using a tree-like physical plan, where each node represents an algebraic operator (selection, join, projection) and each operator is tagged by



```
Query n° 15  :0,4,3,1,11,9,12,25,23,26,51,
Query n° 16  :0,4,3,1,13,14,12,27,28,35,52,
Query n° 17  :0,3,14,28,53,
Query n° 18  :0,3,14,28,54,
Query n° 19  :0,4,3,13,14,27,28,55,
Query n° 20  :0,4,3,1,13,14,12,27,28,35,56,
Query n° 21  :0,4,3,1,13,14,12,27,28,35,57,
Query n° 22  :0,1,4,3,2,15,16,17,18,29,30,31,32,58,
Query n° 23  :0,1,3,2,16,17,18,30,31,32,59,
Query n° 24  :0,1,4,3,2,15,16,17,18,29,30,31,32,60,
Query n° 25  :0,1,4,3,2,15,16,17,18,29,30,31,32,61,
Query n° 26  :0,1,3,16,17,30,32,62,
Query n° 27  :0,3,16,30,63,
Query n° 28  :0,1,3,16,17,30,32,64,
Query n° 29  :0,2,3,8,9,22,23,65,
( 5,1) - ( 6,0) - ( 7,6) - ( 8,2) - ( 9,3) - ( 10,3) - ( 11,4) - ( 12,1) - ( 13,4) - ( 14,3)

Component0 : Q2,Q3,
Component1 : Q22,Q23,Q24,Q25,Q26,Q27,Q28,
Component2 : Q4,Q6,Q10,Q11,Q12,Q13,Q14,Q15,Q29,
Component3 : Q0,Q1,
Component4 : Q16,Q17,Q18,Q19,Q20,Q21,
```

```
======0: Logs of tests :0======
```

Fig. 6.7 – Example of presenting components of workload

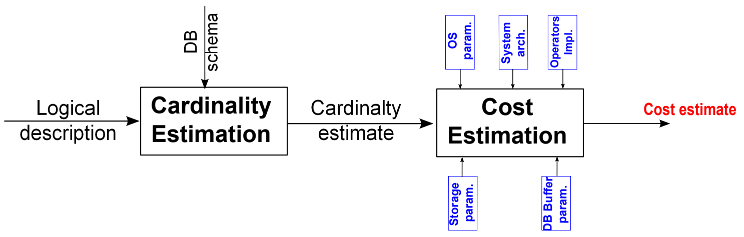


Fig. 6.8 – Overview of cost estimation process

an implementation algorithm. Estimating query processing cost implies the estimation of the execution cost of each operator, which needs also to calculate their cardinalities to estimate the above operators.

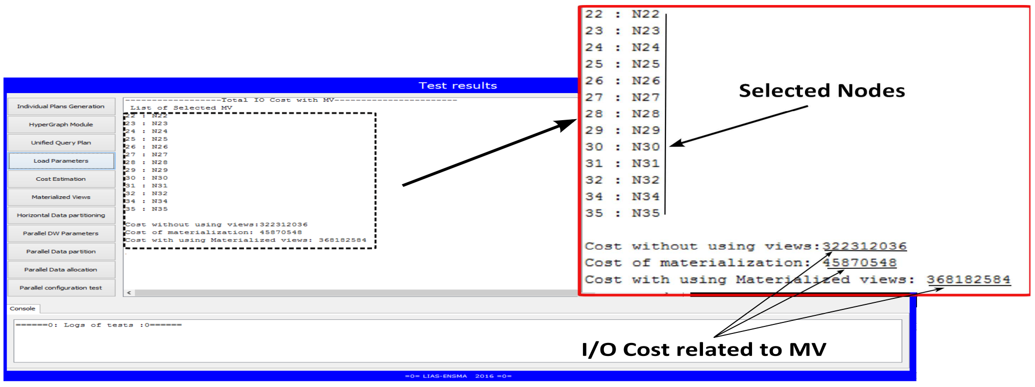
The real cost of query processing is done by time unite (e.g., second), that corresponds to consuming time from the starting query processing until the result is given by the $\mathcal{DBMS}$. This cost depends on two parameters: (1) The input/output cost that is the time of loaded data from storage devices (in our case: *Hard disk*). (2) The $\mathcal{CPU}$ cost ($\mathcal{C}_{\mathcal{CPU}}$) that depends on the physical layout, input/output and the number of arithmetic and logical operations. Figure 6.8, gives an overview and the parameters implied in the processes of estimating query processing cost.

Parallel to using cost models to estimate query processing and to select the optimal plan, other cost models are used in selecting the optimal $\mathcal{OS}$ as $\mathcal{MV}$ and $\mathcal{HDP}$. In addition, each $\mathcal{OS}$ has its own cost functions that estimate their impacts in query execution time and physical parameter like disk space.

The cost model used to develop our estimator module is detailed in the appendix IV.

III.5. Materialized views selection

This module takes a unified query plan to select a set of views under the storage constraint and the maintenance cost. It estimates the impact of using selected $\mathcal{MV}$ on the initial workload. To ensure the scalability of selecting views, the proposed algorithm works on the components of UQP which have small set of candidate views (for more detail, see Chapter 4). Also, selection following components allows proposing a solution of dynamic selection of the views with scheduling of queries. The tools can gives the list of selected $\mathcal{MV}$ and their correspondent I/O costs as shown in the Figure 6.9.



```
------------------Total IO Cost with MV------------------------
 List of Selected MV
22 : N22
23 : N23
24 : N24
25 : N25
26 : N26
27 : N27
28 : N28
29 : N29
30 : N30
31 : N31
32 : N32
34 : N34
35 : N35

Cost without using views:322312036
Cost of materialization: 45870548
Cost with using Materialized views: 368182584
```

```
======0: Logs of tests :0======
```

Fig. 6.9 – A example of the result of selecting materialized views of workload

III.6. Data partitioner

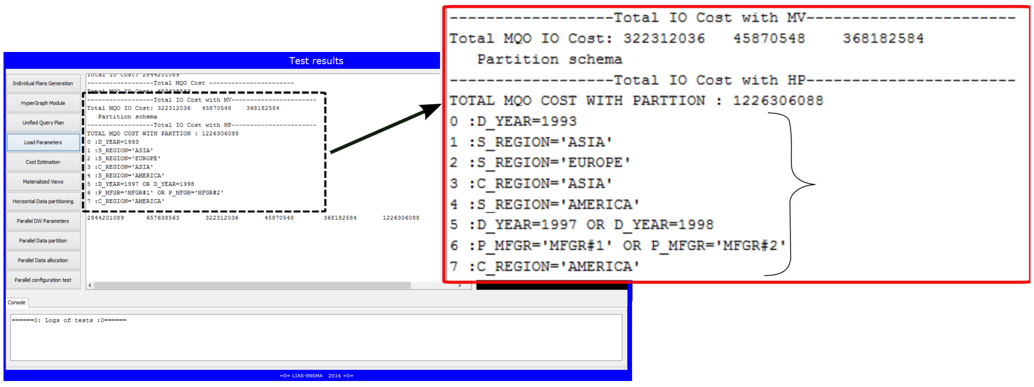
This module uses the query interaction presented as an UQP to select the horizontal data partitioning schema (for more detail, see chapter 4). The module gives the partitioning schema defined by a set of selection predicates, also it gives an estimation of the cost of this schema as shown in the Figure 6.10.

III.7. Deployment designer

Deployment designer allows generating effective data partitioning and fragments allocation schemes by exploiting the query interaction and offering equitable node processing. The partitioning process is guided by sharing expensive operations (e.g. joins). The data needed to process these operations are partitioned and allocated over cluster-nodes.

IV. Implementation

In this section, we present the technical implementation of the framework.



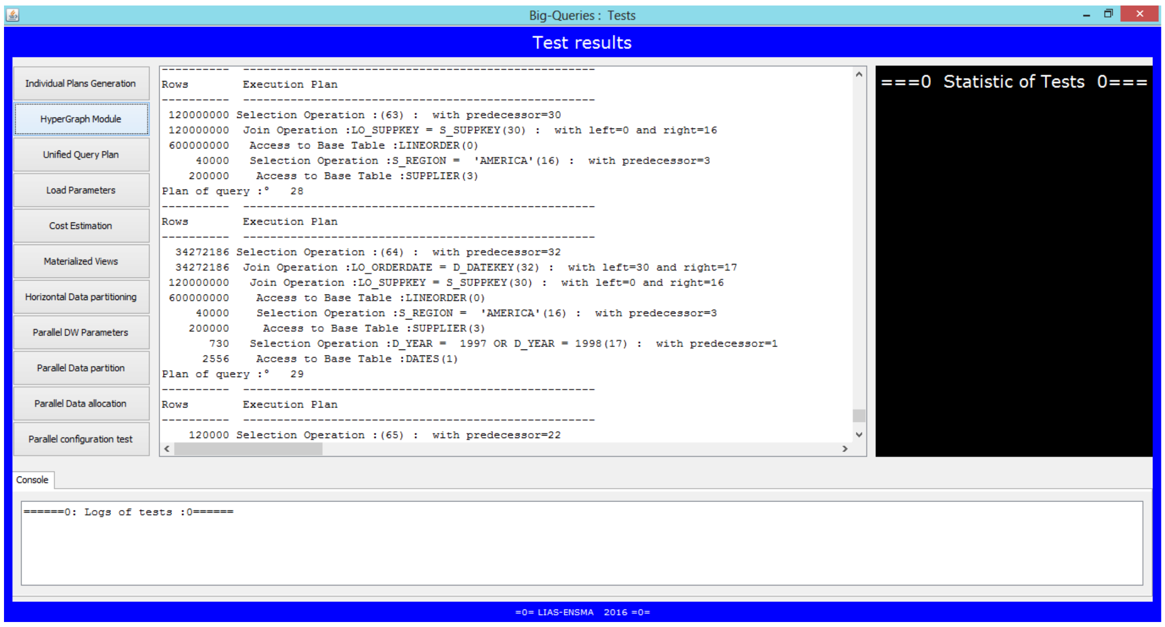
```
Total IO Cost: 2844201089
------------------Total MQO Cost -----------------------
Total MQO IO Cost: 657638563
------------------Total IO Cost with MV-----------------------
Total MQO IO Cost: 322312036   45870548    368182584
   Partition schema
------------------Total IO Cost with HP-----------------------
TOTAL MQO COST WITH PARTION : 1226306088
0 :D_YEAR=1993
1 :S_REGION='ASIA'
2 :S_REGION='EUROPE'
3 :C_REGION='ASIA'
4 :S_REGION='AMERICA'
5 :D_YEAR=1997 OR D_YEAR=1998
6 :P_MFGR='MFGR#1' OR P_MFGR='MFGR#2'
7 :C_REGION='AMERICA'
2844201089     657638563     322312036     45870548     368182584     1226306088
```

```
======0: Logs of tests :0======
```

Fig. 6.10 – A example of the result of data partitioning schema of workload

IV.1. Development environment

The framework has been implemented using the Java programming language. The choice was rather obvious, Java being a solid tool for software development, being platform independent, very robust and reliable and having excellent libraries (including libraries for user interface development). Big-queries framework is developed in windows and Linux environments (Java JDK 1.8. environment). Figure 6.11 shows the main interface of the framework.



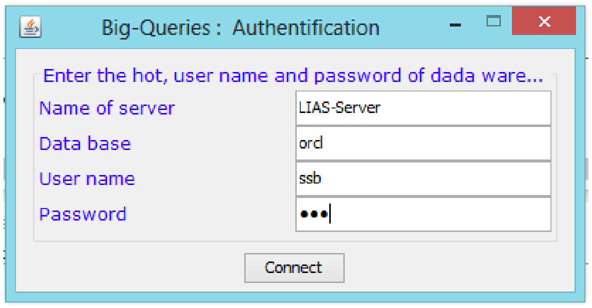
```
----------  ----------------------------------------------------
Rows        Execution Plan
----------  ----------------------------------------------------
 120000000 Selection Operation :(63) :  with predecessor=30
 120000000  Join Operation :LO_SUPPKEY = S_SUPPKEY(30) :  with left=0 and right=16
 600000000   Access to Base Table :LINEORDER(0)
     40000   Selection Operation :S_REGION =  'AMERICA'(16) :  with predecessor=3
    200000    Access to Base Table :SUPPLIER(3)
Plan of query :°   28
----------  ----------------------------------------------------
Rows        Execution Plan
----------  ----------------------------------------------------
  34272186 Selection Operation :(64) :  with predecessor=32
  34272186  Join Operation :LO_ORDERDATE = D_DATEKEY(32) :  with left=30 and right=17
 120000000   Join Operation :LO_SUPPKEY = S_SUPPKEY(30) :  with left=0 and right=16
 600000000    Access to Base Table :LINEORDER(0)
     40000    Selection Operation :S_REGION =  'AMERICA'(16) :  with predecessor=3
    200000     Access to Base Table :SUPPLIER(3)
       730   Selection Operation :D_YEAR =  1997 OR D_YEAR = 1998(17) :  with predecessor=1
      2556    Access to Base Table :DATES(1)
Plan of query :°   29
----------  ----------------------------------------------------
Rows        Execution Plan
----------  ----------------------------------------------------
    120000 Selection Operation :(65) :  with predecessor=22
```

```
===0  Statistic of Tests  0===
```

```
======0: Logs of tests :0======
```

Fig. 6.11 – Main Interface of Big-queries

The framework doted of gateway connection to any $\mathcal{DBMS}$ (Oracle 11g, oracle 12c and PostGress). Figure 6.12, shows the implementation of the gateway connexion.


Big-Queries : Authentification

Enter the hot, user name and password of dada ware...

Name of server	LIAS-Server
Data base	orcl
User name	ssb
Password	•••

***Fig. 6.12** – Gateway connexion of database*

The set of software of our simulator represents applications that give a graphic interface to evaluate a solution without monopolizing of a real design of physical nodes processing. They use a mathematical model to simulate the functionality of the target system with its physical parameters. So, these softwares can give good indicates about query processing, material architecture, system properties in a short amount of time with lower cost. The simulation allows the comparison of many scenarios by measuring the performance of different data.

V. Conclusion

In this chapter, we presented our Big-queries advisor composed of a set of tools to help the DBA to perform the database physical design and deployment. Big-queries takes advantages of the hypergraph theory to exploit in efficient way the volume of interacted queries to select $\mathcal{OS}$ and the deployment platform for a given $\mathcal{DW}$ application. Our tool is considered as the core of the simulation of all processes that we discussed in this thesis thanks to our algorithms and cost models. Currently, we are integrating other $\mathcal{OS}$ such as indexes and replication of data fragments over cluster nodes.

Part III

Conclusion and Perspectives

Wrap-up

In this chapter, we summarize the contributions of our thesis, discuss the results and provide an overview of possible future issues.

I. Summary

Processing large amount of queries issued by advanced applications involving mass data issued from social media, E-commerce Web sites, scientific experiments, etc., as social network, E-commerce, scientific, sensors, etc. has to consider the interaction among queries. This interaction is materialized by the reuse of common intermediate results of these queries. Another important point that motivates our claim is the crucial role that queries play in different phases of the life cycle of database/data warehouse design, especially for physical and deployment phases. It should be noticed that various advisors and tools developed by the most important editors of commercial and academic DBMS either in the context of centralized databases (e.g., DB2 Advsiors [257], Database Tuning Advisor for Microsoft SQL Server [4], [176] for PostgreSQL, etc.) or in distributed databases (e.g. Schism [81]) exist. They aim at recommending to administrators optimization configurations such as indexes and materialized views. *The majority of them are based on workloads. Hence, the question of integrating the volume and interaction of queries is no longer about why doing that, but instead how we can do it in an efficient manner, and for which phases ?*
Therefore, this thesis revisits the multi-query optimization – a classical concept introduced in 80's –, and leverages it by our two dimensions which are query volume and sharing. Our methodology was designed to first define a data structure adapted to our study, and then to evaluate its efficiency and effectiveness regarding two famous problems of the physical design of a relational data warehouses which are materialized view selection and horizontal data partitioning, as well as the problem of parallel data warehouse design.

We would like to mention that our thesis covers large spectrum of topics: multi-query optimizations, data warehouses, physical design, deployment phase, hypergraphs, cost models, etc. To facilitate the presentation of our finding, we fixed some objectives: (i) to conduct a survey of the most important concepts, techniques, algorithms and tools presented in this thesis. This objective forced us to be more precise and to present comparison and analysis of the studied concepts. (ii) to clearly outline our collaboration with Mentor Graphics that yields the analogy between the problem of representing a unified query plan of a workload and an electronic circuit. This collaboration is the key success of our thesis, since it brings us several positive results at research (discovering a new discipline) and personal terms (working, sharing and transmitting our problems and knowledge to the engineers of Mentor Graphics). (iiii) to choose relevant case studies for the deployment of our hypergraph structures, (iv) to develop a tool that capitalizes our finding and assists designers and administrators during their tasks and finally (v) to conduct intensive experiments using our developed simulators within commercial DBMS (Oracle in our case) to evaluate the quality of our findings.

I.1. Surveys of Logical and Physical Optimizations

We have elaborated an in-depth analysis of how state-of-the-art database manage the interaction among queries and how it is exploited in logical, physical optimizations and the deployment phase of the life cycle of the data warehouse. Based on these surveys, we provided new classifications related to logical and physical optimizations that concern multi-query optimization, generic formalization of the problem of the physical design, examples of its instances: materialized views, indexes and horizontal data partitioning, and different used algorithms with their implementation environments (centralized, distributed and parallel architectures).

I.2. Hypergraphs driven Data Structure for Management of Volume and Sharing of Queries

The first contribution of our thesis is the fruit of our collaboration with Mentor Graphics Company to find an analogy between an electronic circuit and a unified query graph. This collaboration gave rise to an approach called SONIC (Scalable Multi-query OptimizatioN through Integrated Circuits). This latter is based on a hypergraph structure that can represent any unified query plan, by ignoring the query order as done in the most traditional approaches [273]. Once it is generated, it is explored by a greedy algorithm that partitions it by the means of hypergraphs partitioning library (HMETIS), and selects the most important node of each partition (called pivot node) that can be candidate for sharing. The pivot node is similar to the principle of queen-bee.
Our approach was validated to evaluate its efficiency in terms of execution time and quality by considering the process of selecting materialized views.
Furthermore, we investigated the scaling behavior of our approach by considering very large

workloads of queries, and identified a high scalability of our generated unified query plan on selecting materialized views.

I.3. What if Unified Query Plan Generation

The second contribution of our thesis concerns the incorporation of the what-if question in the physical design phase of advanced databases such as data warehouses. This incorporation is performed by merging two problems: the multi-query optimization and the physical structure selection, usually performed in a sequential way. As a consequence, we proposed to consider the question which is what-if physical design for multiple query plan generation. A formalization of our problem has been given and instantiated with two optimization structures: materialized views in static and dynamic contexts, and horizontal data partitioning. The hypergraph data structure used for generating the best multiple query plan facilitate the development of algorithms for selecting our optimization structures, where metrics related to each structures are injected into the process of multiple query plan generation. Mathematical cost models estimating the query processing cost are also developed. We conducted intensive experiments using theoretical and a real validation in Oracle 11g. The results approve our what-if issue and the theoretical results are confirmed by Oracle experiments that demonstrate the quality of our cost models.

I.4. Query Interaction in the Deployment Phase

We investigated the role of our hypergraph structures in the process of designing parallel warehouses. Due to the complexity of this phase that includes several steps: data partitioning, data allocation, load balancing, etc. we have only focused on the two first phases; data partitioning and data allocation. We thus presented HYPAD approach (HYper-graph-driven approach for PArallel Data warehouse design) that does the following tasks: (i) capture of interaction among queries. (ii) Generation of landmark predicates using connected components that compose the unified query plan. (iii) Elaboration of modular data partition and data allocation, guided by the components of the unified query plan. Our approach is compared against the most important state of art works and the obtained results show its efficiency and effectiveness. It has been tested under big size workload (10000 queries), that shows its scalability. Note that mathematical cost models have been elaborated and used by our simulator.

I.5. Big-Queries

We presented our tool textitBigQueries, a sort of advisor that takes advantages of the interaction among a large amount of queries. Actually, it gives recommendations for selecting two optimization structures in isolated way: materialized views and horizontal partitioning. It is a modular tool, that can: (1) offer DBA the possibility to select different optimization structures, based on the interaction of queries, (2) give an estimation of query performance, (3) generates

and displays a unified query plan and (4) recommends optimization configurations regarding the deployment phase that suggests partitioning schemes and fragment allocation schema for a given workload. The functional architectures of these two tools have been elaborated based on existing academic and commercial advisors and tools. We would like to mention that BigQueries tool has been used in Graduate Master course "Big Data" given in Poitiers University. Based on the feedbacks of students, it has been incrementally improved in terms of interface and different functionalities such as cost models.
Our tools are open sources and available at the forge of our laboratory LIAS (http://www.lias-lab.fr/forge/projects/bigqueries). Actually, 86 downloads have been done, which represents an asset for students and researchers.

II. Future Work

In this section, we identify open issues related to the set of our contributions. We differentiate between two major classes of issues, namely the hypergraph structure and its usages.

II.1. Hypergraph Structure

In this case of issues, we distinguish three main challenges that have to be addressed in the near future: testing of other hypergraph partitioning tool, dynamic construction of hypergraph and exploratory algorithms.

II.2. Testing Other Partitioning Hypergraph Tools

The popularity of hypergraphs and the increasing demands of IT companies in using them, motivate the research community to develop effective hypergraph partitioning tools: hMeTiS [151], PaToH [66] Mondriaan [263], and parallel tools such Parkway [250], and Zoltan [43]. In this thesis, we only used one tool, which is hMeTis. The exploration of other tools is necessary to evaluate the impact of the partitioning on the obtained results.

II.3. Dynamic Construction of Hypergraphs

In our study, we assume that the workload is known in advance. The construction of our hypergraph is based on this static workload. In many applications of graph algorithms, including communication networks, VLSI design, graphics, and assembly planning, graphs are subject to discrete changes, such as insertions or deletions of vertices or edges. In the last two decades there has been a growing interest in such dynamically changing graphs, and a whole body of algorithmic techniques and data structures has been discovered [92]. Two types of problems concern dynamic graphs : partially dynamic graphs subject only to either insertions, or dele-

tions, but not both at the same time. Fully dynamic Graphs subject to intermixed sequences of insertions and deletions.
These findings have to be integrated in our proposal, by proposing an algebra to construct incremently our hypergraph. The dynamic aspect of graphs complicates the partitioning algorithms. The usage of library managing dynamic graphs such as graph stream is recommended (`http://graphstream-project.org/`).

II.4. Advanced Algorithms to Explore our Hypergraphs

In our thesis, we use simple algorithms (greedy) to explore our hypergraphs to select different pivot nodes. Hence, it would be beneficial to investigate the efficiency of advanced algorithms such as evolutionary ones.

II.5. Hypergraph for Mixed Workload

Today's enterprise systems are partitioned into the so-called Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP). In this thesis, we have mentioned the work performed by Curino et al. [81] about database replication and partitioning considering OLTP transactions that use hypergraphs. In this thesis, we particularity focused on OLAP workload. Merging these two hypergraphs structures is a nice opportunity to consider the problem of mixed workload management. To do so, the development of metrics to measure performance of the mixed workload have to be elaborated in our context.

II.6. Consideration of others problems in the Deployment Phase

Use hypergraph structures to include other complex phases in designing parallel data warehouse, as data replication and load balancing. Also, including other dimensions in designing parallel data warehouses, as optimization structures (materialized views and indexes), in which we will use hypergraph structures to propose a new model of selecting $\mathcal{OS}$ in this new context.

II.7. Other Usages of our Findings

In our proposal, we concentrate our efforts on selecting materialized views and data partitioning. Note that there is a large panoply of optimization structures, where their selection is driven by workloads such as indexes, vertical partitioning, etc. Our finding has to be extended to integrate any optimization structure. To do so, we have to make our tool more generic to allow developers to enrich it by other structures.
Another usage of our finding may concern the query recommendation and relaxation. A recommended or relaxed query is a super or sub query that already exists in the system.

Part IV

Appendices

Cost models

III. Introduction

In the relational model, each query is processed using a tree-like physical plan, where each node represents an algebraic operator (selection, join, projection) and each operator is tagged by an implementation algorithm. Estimating query processing cost implies the estimation of the execution cost of each operator, which needs also to calculate their cardinalities to estimate the above operators.

The real cost of query processing is done by a time unit (e.g., second) that corresponds to consuming time from the starting query processing until the result is given by the $\mathcal{DBMS}$. This cost depends on two main parameters: (1) The inputs/outputs cost ($\mathcal{C}_{\mathcal{I}/\mathcal{O}}$) that gives the time of loaded data from storage devices (in our case: *Hard disk*) to the main memory. (2) The $\mathcal{CPU}$ cost ($\mathcal{C}_{\mathcal{CPU}}$) that depends on the physical layout, input/output and the number of arithmetic and logical operations. Fig.7.1, gives an overview and the parameters implied in the processes of estimating query processing cost.

Parallel to using cost models to estimate query processing and to select the optimal plan, other cost models are widely used in database physical design. They usually used in selecting the optimal $\mathcal{OS}$ as $\mathcal{MV}$, indexes, data partitioning, etc. In addition, each $\mathcal{OS}$ has related cost functions that estimate their impacts in query execution time and physical parameter like disk space.

In this Chapter, we develop the cost models and their parameters used in our work. Firstly, we give cardinality estimation functions of intermediate results in the physical query plan. Secondly, we detail estimating functions of query processing cost representing the number of inputs/outputs.

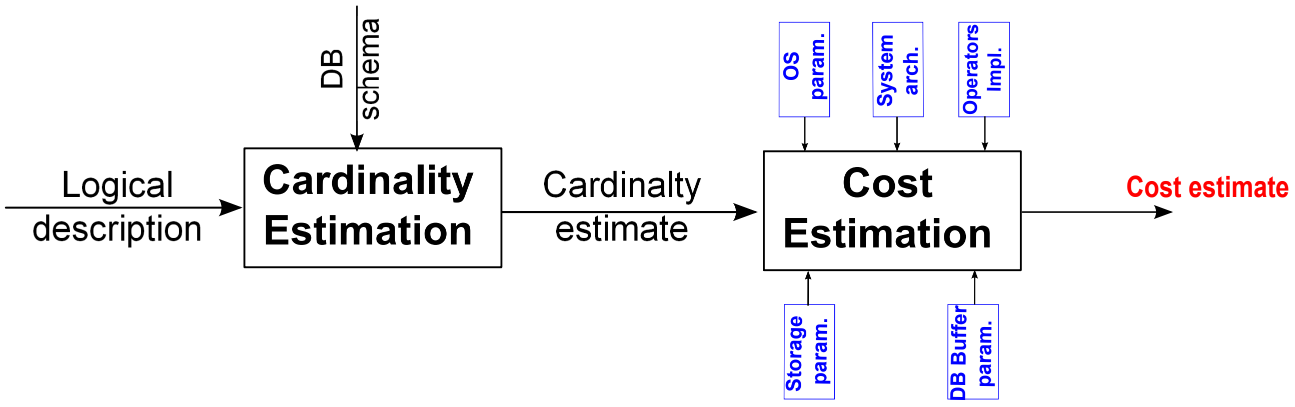


Fig. 7.1 – Overview of cost estimation process

IV. Cardinality estimation

The cardinality of a relation or an intermediate result is the number of its tuples. The cardinality considered as the main input of any estimation functions of query processing cost or selecting $\mathcal{OS}$. Accordingly, the quality of any cost model depends on the quality of cardinality estimation functions. Many works have shown that error estimation of cardinality implies choosing bad plan that can be thousands times slower. [255, 181].
To estimate the cardinality of intermediate results, it is necessary to have statistics about data distribution in base relations. So, to estimate cardinality or intermediate results, any query optimizer uses tools to get and update the statistics of data distribution in databases and then it applies functions on this statistic according query parameters.

IV.1. Statistics computation

The quality of a query optimizer is tightly tied to the quality of cost estimation functions, which are, in turn, based on the cardinality estimation. Accordingly, bad statistics can easily alter the execution plan, and the availability of good statistics can readily improve the quality of the best processing plan[69]. Statistics are periodically re-computed by the $\mathcal{DBMS}$, or right after updating base relations. Today, most of existing $\mathcal{DBMS}$ have implemented statistics catalogs [63, 56]. Although the fact that they do not undergo a radical change in a short period, their computation may be very costly [71, 57]. These statistics can be categorized into three categories:

- distribution of attribute values, like number of tuples, average size of column/tuple, distinct and maximum/minimum values of attributes. To reduce the complexity of statistics computation issue, many $\mathcal{DBMS}$ assumes that there is a uniform distribution of values [209, 107].

Parameters name	**Description**
R_i	a relation that may represents fact or dimension tables
$\|R_i\|$	number of tuples of the relation R_i
$\|\|R_i\|\|$	number of pages on which the relation R_i is stored
$\|A\|$	number of distinct values of the attribute A
max_A	max value for an attribute A in the relation R
min_A	min value for an attribute A in the relation R
$L_a(A_{R_i})$	average length of a value of attribute A of relation R_i (in bytes)
$L_t(R_i)$	average length of a tuple of relation R_i in bytes

***Table 7.1** – Cardinality estimation parameters*

- $\mathcal{DBMS}$ Hardware parameters that include network, hard disk characteristics (e.g. bandwidth and time of reading one page).
- Statistics of using access methods (index, $\mathcal{MV}$, etc.), like spent-time to scan an index.

IV.2. Intermediate results cardinality

Cardinality estimation of the intermediate results depends on the type of the current operator in the physical query plan. Next, we present basic functions used to estimate the cardinality of intermediate results for every operator type (selection, projection, join, etc.). Note that the goal of cardinality estimation is not to give an exact information, but instead an estimation to select a good physical query plan or to select near-optimal $\mathcal{OS}$.
For a given relation or an intermediate result R_i, we denote its cardinality by $|R_i|$. We need some notations to present our functions, as depicted in Table 7.1. Our cost functions based on the work of Selinger et al. [221].

IV.2.1. Cardinality of selection

The selection applied to one relation using one or more predicates. Let p, a selection predicate applied to a relation R. We denote $\sigma_p(R)$ as the result of this selection. To do so, we need an extra parameter called the *selectivity factor* of the predicate p, which is defined as the part of rows satisfying the predicate, and its value is between 0 and 1. In other words, it is the probability to pick each tuple from relations to be used as arguments of operators. Let f_p be the selectivity factor of the predicate p. f_p is defined as :

$$f_p = \frac{|\sigma(R)|}{|R|} \tag{7.1}$$

Therefore, the cardinality of the selection result can be defined as:

$$|\sigma_p(R)| = f_p * |R| \tag{7.2}$$

Predicate p	**Selectivity** f_p
$\neg(p)$	1-f_p
$p_1 \wedge p_2$	$f_{p_1} * f_{p2}$
$p_1 \vee p_2$	$f_{p1} + f_{p_2}$ - $f_{p_1} * f_{p_2}$
$A = val$	$1/|A|$
$A = B$	$1/max(|A|, |B|)$
$A > val$	$(max_A - val)/(max_A - min_A)$
$A < val$	$(val - min_A)/(max_A - min_A)$
$val_1 \leqslant A \leqslant val_2$	$(val_2 - val_1)/(max_A - min_A)$

Table 7.2 – Selectivity estimation of complex predicates

To estimate the cardinality, we suppose that we have uniform values in columns. One predicate p can involve more than one attribute or value. So, the predicate p can be simple when it has one value or complex when it involves *boolean* operators linking several simple predicates (like p_1 and p_2). The selectivity of complex predicates can be easily computed based on their simple predicates. In the Table 7.2, we summarize the selectivity estimation formulas, where A and B denote attributes, val, val_1 and val_2 denote constants, max_A is the maximum value in the column A, $|A|$ is the number of distinct values in the attribute A. Note that, the optimizer can use logical equivalents to detect if the condition equivalent to $FALSE$, and in this case the cardinality is clear that equivalent to 0.

IV.2.2. Cardinality of projection

The cardinality of the result of a projection on the relation R_i is, in principle, the cardinality of R_i itself:

$$|\Pi_A(R)| = |R| \tag{7.3}$$

such that A, is an attribute or a set of attributes.
Sometimes, the query optimizer eliminates duplicate tuples to improve query plan, because when these latter are used by other operators like *join*, the result is the same. In this case, the cardinality is equal to:

$$|\Pi_A(R)| = |A| \tag{7.4}$$

IV.2.3. Cardinality of Cross product

The cardinality of the result of a cross product of two relations is the multiplication of their cardinalities.

$$|R_1 \times R_2| = |R_1| \times |R_2| \tag{7.5}$$

IV.2.4. Cardinality of Join

In our work, we have considered only the natural join. The natural join between two relations involves only the equality of two attributes. So we study the join $R_1(A, b) \bowtie R_2(b, C)$, where b is the attribute of the join predicate, and A, C denote a set of attributes. The cardinality of the join result $|R_1 \bowtie R_2|$ depends on the values of the attribute b. For example:

- If R_1 and R_2 have disjoint sets of b -values, $|R_1 \bowtie R_2|$ =0;
- If b is the key of R_2 and the foreign key of R_1, so each tuple of R_1 joins exactly one tuple in R_2, and $|R_1 \bowtie R_2|$ =$|R_1|$.

In the general case, we suppose that b_{R_1} and b_{R_2} are the b attribute of R_1 and R_2 respectively, and r_1 is a tuple of R_1, and r_2 of R_2. if $|b_{R_1}| > |b_{R_2}|$, then b-value of r_2 is one value that appears in the b-value of R_1. Hence, the probability that r_1 and r_2 share the same b-value is $1/|b_{R_1}|$. Similarly, if $|b_{R_1}| < |b_{R_2}|$, then the probability that r_1 and r_2 share the same b-value is $1/|b_{R_2}|$. In join operation, the possible comparison of pairs r_1 and r_2 is $|R_1|$ *$|R_2|$, So the cardinality of the result of a natural join is:

$$|R_1 \bowtie R_2| = max(|R_1|, |R_2|)/max(|b_{R_1}|, |b_{R_2}|) \tag{7.6}$$

The same goes for multiple join attributes. We suppose that B are a set of the attributes involved in the join, the cardinality becomes:

$$|R_1 \bowtie R_2| = max(|R_1|, |R_2|)/max(|bi_{R_1}|, |bi_{R_2}|), \forall b_i \in B \tag{7.7}$$

If we assume that $J_{R_1R_2}$ is the probability that two rows verify join conditions, the cardinality of the result of that join can be calculated as follows:

$$|R_1 \bowtie R_2| = max(|R_1|, |R_2|) \times J_{R_1R_2} \tag{7.8}$$

IV.2.5. Cardinality of aggregation

The number of tuples generated from an aggregation corresponds to the number of the resulting groups. This latter can be between 1 and $|R|$, so the cardinality of the result of the aggregation equals to:

$$|\varGamma(R)| = {}^{|R|}/_{2} \tag{7.9}$$

IV.2.6. Cardinality of Union

The maximum of tuples number generated from the union of two relations R_1 and R_2 ($R_1 \cup R_2$), is the sum of their cardinalities $|R_1|$ and $|R_1|$, and the minimum is the maximum between $|R_1|$ and $|R_2|$. So, the cardinality of the union can be the average of maximum and minimum values.

$$|R_1 \cup R_2| = max(|R_1|, |R_2|) + {}^{min(|R_1|, |R_2|)}/_{2} \tag{7.10}$$

IV.2.7. Cardinality of intersection

The cardinality of the result of the intersection of two relations R_1 and R_2 ($R_1 \cap R_2$), is in the range of 0 to the minimum cardinality of the two relations. So, the cardinality can be considered as the average value:

$$|R_1 \cap R_2| = {}^{min(|R_1|,|R_2|)}/_2 \tag{7.11}$$

IV.2.8. Cardinality of difference

The maximum of tuples generated from the difference of two relations R_1 and R_2 ($R_1 \setminus R_2$), is the cardinality of R_1 ($|R_1|$) and the minimum is $|R_1|$-$|R_2|$. So, the cardinality of the difference can be considered as the average:

$$|R_1 \setminus R_2| = {}^{(|R_1| - |R_2|)}/_2 \tag{7.12}$$

V. Cost estimation

The execution cost of an operator is the sum of two measures: (1) Input/output ($\mathcal{C}_{\mathcal{I}/\mathcal{O}}$) which is the time of loading/storing data from/to the disk, and (2) $\mathcal{CPU}$ cost ($\mathcal{C}_{\mathcal{CPU}}$) consumed by arithmetic and comparison functions processed by the $\mathcal{CPU}$. Note that, in the context of distributed databases (interconnected sites), the network flow involves additionally a network cost ($\mathcal{C}_{net}$). The $\mathcal{C}_{net}$ of a query is the necessary time to transfer all data and sub-results between sites and finally to produce the result.
Recently, with green computing, another measure has been considered is query cost estimation, which is the power and energy consumption cost. In this case, the challenge is to reduce energy [253] and CO_2 pollution.
The query is executed following its physical plan that shows the data flow between its different operators. The total $\mathcal{I}/\mathcal{O}$ and $\mathcal{CPU}$ costs of a query is the sum of the cost of each operator (op) occurring in the physical execution plan. For a given query the total costs are:

$$\mathcal{C}_{\mathcal{I}/\mathcal{O}} = \sum_{op \in \text{UQP}} \mathcal{C}_{\mathcal{I}/\mathcal{O}}(op) \tag{7.13}$$

$$\mathcal{C}_{\mathcal{CPU}} = \sum_{op \in \text{UQP}} \mathcal{C}_{\mathcal{CPU}}(op) \tag{7.14}$$

The system can execute operators in parallel, which makes the summation of the $\mathcal{I}/\mathcal{O}$ and $\mathcal{CPU}$ costs insignificant. For these reasons, researchers have proposed many techniques to calculate the total cost of the query, as described below.

- **Weighted sum of costs**: The total cost $\mathcal{C}_{\mathcal{Q}}$ is equal to a weighted sum of the $\mathcal{I}/\mathcal{O}$ and $\mathcal{CPU}$ costs [221]:

$$\mathcal{C}_{\mathcal{Q}} = \mathcal{C}_{\mathcal{I}/\mathcal{O}} + w \times \mathcal{C}_{\mathcal{CPU}} \tag{7.15}$$

where w is an adaptive variable between $\mathcal{I}/\mathcal{O}$ and $\mathcal{CPU}$ costs. The weight w increases when $\mathcal{CPU}$ takes more time in executing operators and decreases when $\mathcal{I}/\mathcal{O}$ takes more time to load/store processes.

- **Sum of the cost**: due to the difficulty of defining the relation between $\mathcal{CPU}$ and $\mathcal{I}/\mathcal{O}$ cost by the weight value, some propositions ignore it [126]. Hence, the total cost becomes the sum of $\mathcal{CPU}$ and $\mathcal{I}/\mathcal{O}$ costs:

$$\mathcal{C}_{\mathcal{Q}} = \mathcal{C}_{\mathcal{I}/\mathcal{O}} + \mathcal{C}_{\mathcal{CPU}} \tag{7.16}$$

- **Maximum of the cost**: In the previous approaches, the concurrency between $\mathcal{CPU}$ processing time and $\mathcal{I}/\mathcal{O}$ loading time is totally ignored. The time execution becomes the maximum between $\mathcal{CPU}$ and $\mathcal{I}/\mathcal{O}$ costs [77].

$$\mathcal{C}_{\mathcal{Q}} = \mathrm{Max}(\mathcal{C}_{\mathcal{I}/\mathcal{O}} + \mathcal{C}_{\mathcal{CPU}}) \tag{7.17}$$

V.1. Cost model parameters

As shown in Fig.7.1, estimation function depends on many parameters like buffer size, storage devices, system architectures, etc. [225]. Next, we describe these parameters:

V.1.1. Storage devices

Query processing time is sensitive to the storage access parameters [210]. The complexity of storage systems has been increasing, and so do the cost parameters. The execution of a query can involve various types of storage devices like hard disks, flash disks, and main memory. These storage devices can be classified into three categories according to their storage capacity, seek speed, and reading one-byte cost:

1. **Cache:** is founded using the same chip of the micro-processor, data can be accessed in a few nanoseconds.
2. **Main Memory:** data can be accessed between 10 to 100 nanoseconds range.
3. **Secondary storage:** covers all types of hard/optical disks, flash memory, etc. They access and read data in milliseconds, and most of them are made of magnetic disks, and have the following parameters:
    - **storage layout:** like time of reading one page using direct access or sequential access;
    - **storage device:** like number of cylinder, track, sectors, seek cost, etc.

V.1.2. Database Buffer

A part or all intermediate results can be directly stored in the buffer for future use by parent operators, what minimizes their input cost. Thus, the buffer size and management algorithm become necessary to estimate the $\mathcal{I}/\mathcal{O}$ cost of intermediate results [269].

V.1.3. Access paths

data can be loaded from tables using sequential or random access. Each access mode has its specific cost.

V.1.4. Query parameters

Cost estimating functions take as input the query in the form of a physical plan that contains selection, join, projection, aggregation, and grouping operations. Each operator has a predicate like selection and join predicates. These latter are used to estimate the cardinality of intermediate results (For more details, see Section IV). In our work, we consider a workload of k queries, $\mathcal{W}$ =$\{Q_1, .., Q_k\}$, where each query Q_i has a frequency of use f_i.

V.1.5. Database parameters

The schema of databases is always considered as input for any cost model, used for estimating the cardinality of intermediate results and to identify $\mathcal{OS}$ characteristics. In our work, we have used a star schema of $\mathcal{DW}$, that has one fact table $\mathcal{F}$ and d dimension tables $\mathcal{D}$ =$\{D_1, .., D_d\}$. Two important parameters are associated to each table $\mathcal{F}$ or $\mathcal{D}_i$ ($1 \leq i \leq d$): cardinality denoted by $|\mathcal{F}|$ and $|\mathcal{D}_i|$ respectively, and the real table size generally given in bytes (in our case we have used the *pagesize* unit, PS =8192 bytes), denoted by $||\mathcal{F}||$. Other parameters are needed like tuple and columns average size (for more details, see Table 7.1).

V.1.6. Deployment parameters

$\mathcal{DB}$ or $\mathcal{DW}$ can be deployed according to different architectures (*SMP*,*MPP* and *Shared nothing* (More details, see Chapter 2). In our tests, we have used Shared Nothing Database Cluster, denoted by *SN-DBC*; and centralized $\mathcal{DB}$. The *SN-DBC* is composed of M processing nodes (servers) $\mathcal{N}$ =$\{N_1, .., N_m\}$, where each node N_i has a storage capacity S_i to store a part of data. The nodes are interconnected using high-speed network and their processing are coordinated by one node called *coordinator node*. Fig.7.2, illustrates an example of a $SN - DBC$.
The communication between nodes produces an extra cost called *communication cost* ($\mathcal{C}_{net}$). It can expressed using an $M \times M$ matrix CC, where either columns or lines represent the processing nodes (N_i $1 \leq i \leq M$) and the value $CC[i][j]$ represents the number of pages transferred from the processing node N_i to the processing node N_j in a period of time. Thus, the total

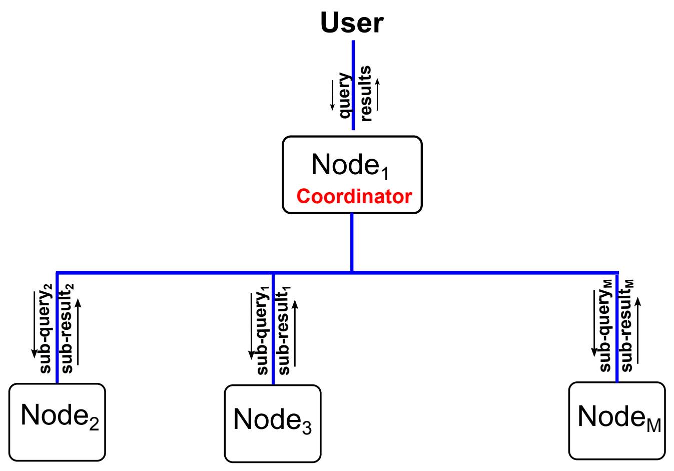


***Fig. 7.2** – Example of $SN - DBC$*

communication cost can be defined as :

$$\mathcal{C}_{net} = \sum_{1 \leq i,j \leq M, i \neq j} CC[i][j] \tag{7.18}$$

The parameters that influences the cost model in $Costmodel!SN - DBC$ system are:

- **Data partitioning parameters :** In a $SN - DBC$ architecture, data are partitioned over many clusters. Let N_F be the number of fragments generated by the process of data partitioning $\mathcal{P}_c$ =$\{F_1, .., F_{N_F}\}$. It is defined as follows:

$$N_F = \prod_{i=1}^{d} F_{d_i} \tag{7.19}$$

  Such as F_{d_i} is the number of fragments of the table D_i, and it is assigned the default value (1) whenever the table does not participate in the partitioning process [45]. To avoid the explosion of fragments number that causes a difficulty to DBA in its maintenance task [20], a threshold W that delimits the maximum number of fragments N_F is therefore required. For any partitioning schema, it is necessary to know which fragment is used by each query. This is done using a matrix called *matrix of use* M_u. The columns and lines of the matrix correspond to k queries and N_F fragments respectively. $M_u[i][j]$ equal to value 1 if the query Q_i use the fragment F_j and 0 otherwise. Table 7.3, illustrates the parameters of fragments.

- **Data allocation parameters :** In $NS - DBC$, the set of fragments are allocated to processing nodes $\mathcal{N}$. If each fragment is allocated to one and only one cluster node, we call it *no-redundancy allocation*, and *redundancy allocation* otherwise. To ease the localization of fragments, the system uses a binary matrix of placement (M_p) [36]. The

Parameters name	**Description**
DBC	cluster of M processing nodes $\mathcal{N}$ $=\{N_1, .., N_M\}$
M	number of processing nodes
S_i	storage capacity of the processing node N_i
N_F	number of fragments
W	threshold of data partitioning
$\lvert F_i \rvert$	cardinality of the fragment F_i
$\lvert\lvert F_i \rvert\rvert$	size of the fragment F_i in pages
M_u	matrix of use of fragments

Table 7.3 – Data partitioning schema parameters

lines and columns of the matrix are the N_F fragment and M processing nodes respectively. $M_p[i][j]$ has a value 1 if the fragment F_i is allocated in the processing node N_j, and a value 0 otherwise. The size of data stored in each processing node N_j will be calculated using the following equation:

$$size(N_j) = \sum_{i=1}^{M} ||F_i|| \times M_p[i][j] \tag{7.20}$$

V.1.7. $\mathcal{OS}$ parameters

- **$\mathcal{MV}$:** update frequency and if they are automatically rewritten or not.
- **Indexes:** Index has three costs to defined, firstly the cost of updating index following updating in the indexed object. Cost of updating the structure of the index like *bitmap index* and the cost of processing queries that equal to the scan index generally in the main memory and the cost of accessing to data.
- **Data partitioning:** the number of partitions, size of each partitions. To estimate the selection cardinality, the selectivity will be calculated for each partition.

V.2. Estimation functions for query processing

As has been previously mentioned, the cost of an operator or the whole query plan can be represented by two costs: $\mathcal{CPU}$ and $\mathcal{I}/\mathcal{O}$.

V.2.1. $\mathcal{CPU}$ cost

$\mathcal{CPU}$ cost is the number of $\mathcal{CPU}$ cycles necessary to compare different tuples. To estimate this number, it is necessary to know the cardinality of the input(s) and the output of each operator in the query execution plan. Once the input cardinality is estimated, it only remains to add

PS	page size
t_{disk}	disk transfer time for one page
t_{net}	network transfer time for one page
$\|\|R\|\|$	size on pages of the relation R
$\|R\|$	cardinality of the relation R (number of tuples)
$L_t(R)$	tuple length for the relation R
$L_a(C_R)$	the average length of the column C of the relation R
$\|R_C\|$	number of distinct value of the column C of the relation R
f_R	selectivity factor for selection f on the relation R
π_R	selectivity factor for projection π on the relation R
$J_{R_1R_2}$	selectivity factor for join J on the relations R_1 and R_2

Table 7.4 *– Cost function parameters*

the $\mathcal{CPU}$ cost of the comparison functions for each $\mathcal{CPU}$ cycle. The number of $\mathcal{CPU}$ cycles is easy to compute for most comparison types (like boolean and number). However, it gets more complicated when comparing string values. In our work, we have ignored this measure, because $\mathcal{C}_{\mathcal{I}/\mathcal{O}}$ is by far the most dominant cost in the context of very large $\mathcal{DW}$.

V.2.2. $\mathcal{I}/\mathcal{O}$ cost

We assume that every operator is responsible for conveying its result to the next operator through main memory.
For our cost model, we consider some parameters, as depicted in Table 7.4:
The processing cost is concerned with measuring execution time, which is directly influenced by the number of $\mathcal{I}/\mathcal{O}$ pages. Thus, we represent the $\mathcal{C}_{\mathcal{I}/\mathcal{O}}$ of any operator (op) either by loaded pages number or time seconds, as follows:

$$\mathcal{C}_{\mathcal{I}/\mathcal{O}}(op) = \begin{cases} P_{op}, & \text{if the cost represented by number of } \mathcal{I}/\mathcal{O} \text{ pages} \\ T_{op}, & \text{if the cost represented by elapsed time} \end{cases} \tag{7.21}$$

where P_{op} is the number of pages read or written during the execution of the operator op, and T_{op} is the elapsed time while running the operator op. P_{op} and T_{op} are defined as:

$$P_{operator} = P_{input} + P_{interm} + P_{output} \tag{7.22}$$

$$T_{operator} = T_{input} + T_{interm} + T_{output} \tag{7.23}$$

where P_{input} and T_{input} are the number of pages and time consumed in reading the input relation(s) respectively. P_{output} and T_{output} are respectively, the number of pages and time consumed in forwarding output relation(s) to the next parent operator (task) or to disk, assuming that the final output stream is written to disk for subsequent usage. P_{interm} and T_{interm} are the number of pages and time consumed in storing intermediate results on disk. For some operators, P_{interm} and T_{interm} are null. $T_{operator}$ can be directly estimated from $P_{operator}$ by using calibrating database parameters to define the average time for loading pages using network or disk. Knowing that unary operators need one relation to be read unlike binary ones, which need two relations; the input cost of any operator is defined as:

$$T_{input} = \begin{cases} P_{input} * t_{disk}, & \text{if unary operators} \\ P_{input_1} * t_{disk} + P_{input_2} * t_{disk}, & \text{if binary operators without using pipeline} \\ max(P_{input_1} * t_{disk}, P_{input_2} * t_{disk}), & \text{if binary operators with using pipeline} \end{cases} \tag{7.24}$$

The output and intermediate costs remain the same whether for unary or binary operators:

$$T_{output} = P_{output} * t_{disk} \tag{7.25}$$

$$T_{interm} = P_{interm} * t_{disk} \tag{7.26}$$

Next, we give the $\mathcal{I}/\mathcal{O}$ cost for each type of operator:

- **Cost of a Selection:** a selection on a relation R reduces the size of R: horizontally by a selectivity factor f_R and vertically by a filtering factor $\phi_{R,q}$ (only q attributes are retained). We define ϕ_R^q as the ratio of the size of q columns from one tuple in the relation R, ϕ_R^q can be defined as:

  $$\phi_R^q = \sum_{c \in q} (L_a(R_c))/L_t(R) \tag{7.27}$$

  The output cost of a selection operation defined as:

  $$P_{output} = f_R * ||R|| * \phi_R^q \tag{7.28}$$

  For the input cost, all tuples of R are scanned, except if R comes sorted by restriction attributes, then only tuples satisfying the selection condition will be scanned. So, the input cost can be calculated as follows:

  $$P_{input} = \begin{cases} ||R|| & \text{, if } R \text{ is not sorted} \\ f_R \times ||R|| & \text{, if } R \text{ is sorted} \end{cases} \tag{7.29}$$

  Selection operator do not produce any intermediate result (P_{interm} =0). Hence, the $\mathcal{I}/\mathcal{O}$ cost defined as:

  $$\mathcal{C}_{\mathcal{I}/\mathcal{O}} = \begin{cases} ||R|| + f_R * ||R|| * \phi_R^q & \text{, if } R \text{ does not come sorted} \\ f_R \times ||R|| + f_R * ||R|| * \phi_R^q & \text{, if } R \text{ comes sorted} \end{cases} \tag{7.30}$$

- **Cost of a Projection:** In projection, there are not redundant attributes to be removed, i.e., the input is exactly the attributes to be forwarded to the parent operator.

$$P_{output} = \phi_R * ||R|| \tag{7.31}$$

If the projection must sort its input, the P_{interm} becomes:

$$P_{interm} = \begin{cases} 0 & \text{, if } P_R \leq M \text{ or no sorting is performed} \\ P_R \times log_{M-1}(P_R) & \text{, otherwise} \end{cases} \tag{7.32}$$

- **Cost of a Join**

  We consider the classic join operator that is applied on two relations R_1 an R_2, and produces an output relation $R_1 \bowtie R_2$ equivalent to:

$$P_{output} = J_{R_1 R_2} \times PS \times \frac{L_t(R_1 \bowtie R_2)}{L_t(R_1) \times L_t(R_2)} \times ||R_1|| \times ||R_2|| \tag{7.33}$$

  P_{input} and $P_{interim}$ depends on the used join algorithm, as follows:

  - **Nested loop algorithm** In the nested loop algorithm, the outer relation should be the smallest one, in order to reduce the number of iterations [157]. However, if only one relation fits in main memory, it is used as the inner relation to avoid repeated disk accesses.

    Let R_1 and R_2 be the relations implied in the join operator, where R_1 is the outer relation. So, $||R_2|| \leq M \leq ||R_1||$ As we have shown previously, P_{input} is defined as:

$$P_{input} = \begin{cases} ||R_1|| + ||R_2|| & \text{, if no pipeline} \\ max(||R_1||, ||R_2||) & \text{, if pipeline} \end{cases} \tag{7.34}$$

    For the intermediate cost, it differs whether inner relation (R_1) can fit in memory or not. If it does, the cost of the join is equal to the CPU-cost. Otherwise, R_1 must be stored on disk and be repeatedly read from it for every tuple of the outer relation (R_2). So, the P_{interm} can be defined as:

$$P_{interm} = \begin{cases} 0 & \text{, if } ||R_1|| \leq (M\text{-}||R_2||) \\ |R_1| \times ||R_2|| & \text{, otherwise} \end{cases} \tag{7.35}$$

    But if the inner relation is already sorted, the comparison will be repeated only for tuples satisfying condition predicate.

- **Merge Join Algorithm** The merge join algorithm is only used when both relations come sorted on the join attribute. Relations are retrieved in parallel. So, P_{interm} is null, and P_{input} is computed as follows:

$$P_{input} = max(||R_1||, ||R_2||) \tag{7.36}$$

- **Hash Join Algorithm** We assume that R_1 is the outer relation. So, the hash table is built for R_2 and $||R_2|| \leq ||R_1||$. The cost of creating the hash table HT is the cost of loading R_2 and writing it to the disk. Then HT and R_1 are joined following a nested loop join such that HT is the inner relation. If we suppose that J_{hash} is the percentage of tuples of R_2 that depends on the join selectivity factor and on the hash function, then the cost of input P_{input} and intermediate P_{interm} will be defined as:

$$||HT|| = J_{hash} \times ||R_2|| \tag{7.37}$$

$$P_{input} = \begin{cases} ||R_1|| + ||R_2|| + ||HT|| & \text{, if no pipeline} \\ max(||R_1||, ||R_2||) + ||HT|| & \text{, if pipeline} \end{cases} \tag{7.38}$$

$$P_{interm} = \begin{cases} 0 & \text{, if } ||HT|| \leq (M\text{-}||R_1||) \\ |R_1| \times ||HT|| & \text{, otherwise} \end{cases} \tag{7.39}$$

SSB-Based Benchmark Query Templates

VI. Introduction

In our work, we have used a *SSB-Query* generator, that randomly generates a workload based on list of template. The generator tool randomizes the values of selection and the number of tables by using the following templates:

VII. List of query template

Q:1.1
```
        select
        sum(lo_extendedprice*lo_discount) as revenue
        from
        lineorder , dates
        where
        lo_orderdate = d_datekey
        and d_year = [Y]
        and lo_discount between [DL] and [DH]            | [DL]= and [DH] and [DH]>[DL]
        and lo_quantity [COMP] 25;
```
Q:1.2
```
        select
            sum(lo_extendedprice*lo_discount) as revenue
        from
            lineorder , dates
        where
            lo_orderdate = d_datekey
            and d_yearmonthnum = [YMN]
            and lo_discount between [DL] and [DH]
            and lo_quantity between [QL] and [QH];
```
Q:1.3
```
        select
            sum(lo_extendedprice*lo_discount) as revenue
        from
            lineorder , dates
        where
```

```
            lo_orderdate = d_datekey
            and d_weeknuminyear = [WNY]
            and d_year = [Y]
            and lo_discount between [DL] and [DH]
            and lo_quantity between [QL] and [QH];
Q:2.1
        select
            sum(lo_revenue), d_year, p_brand
        from
            lineorder, dates, part, supplier
        where
            lo_orderdate = d_datekey
            and lo_partkey = p_partkey
            and lo_suppkey = s_suppkey
            and p_category = [C]
            and s_region = [R]
        group by
            d_year, p_brand
        order by
            d_year, p_brand;
Q:2.2
        select
            sum(lo_revenue), d_year, p_brand
        from
            lineorder, dates, part, supplier
        where
            lo_orderdate = d_datekey
            and lo_partkey = p_partkey
            and lo_suppkey = s_suppkey
            and p_brand between [BL] and [BH]
            and s_region = [R]
        group by
            d_year, p_brand
        order by
            d_year, p_brand;
Q:2.3
        select
            sum(lo_revenue), d_year, p_brand
        from
            lineorder, dates, part, supplier
        where
            lo_orderdate = d_datekey
            and lo_partkey = p_partkey
            and lo_suppkey = s_suppkey
            and p_brand = [B]
            and s_region = [R]
        group by
            d_year, p_brand
        order by
            d_year, p_brand;
Q:3.1
        select
            c_nation, s_nation, d_year,
            sum(lo_revenue) as revenue
        from
            customer, lineorder, supplier, dates
        where
            lo_custkey = c_custkey
            and lo_suppkey = s_suppkey
            and lo_orderdate = d_datekey
            and c_region = [R]
            and s_region = [R]
        and d_year >= [YL] and d_year <= [YH]
    group by
        c_nation, s_nation, d_year
    order by
        d_year asc, revenue desc;
```

Q:3.2

```
    select
        c_city, s_city, d_year, sum(lo_revenue) as revenue
    from
        customer, lineorder, supplier, dates
    where
        lo_custkey = c_custkey
        and lo_suppkey = s_suppkey
        and lo_orderdate = d_datekey
        and c_nation = [N]
        and s_nation = [N]
        and d_year >= [YL] and d_year <= [YH]
    group by
        c_city, s_city, d_year
    order by
        d_year asc, revenue desc;
```

Q:3.3

```
    select
        c_city, s_city, d_year, sum(lo_revenue) as revenue
    from
        customer, lineorder, supplier, dates
    where
        lo_custkey = c_custkey
        and lo_suppkey = s_suppkey
        and lo_orderdate = d_datekey
        and (c_city=[CI1] or c_city=[CI2])
        and (s_city=[CI1] or s_city=[CI2])
        and d_year >= [YL] and d_year <= [YH]
    group by
        c_city, s_city, d_year
    order by
        d_year asc, revenue desc;
```

Q:3.4

```
    select
        c_city, s_city, d_year, sum(lo_revenue) as revenue
    from
        customer, lineorder, supplier, dates
    where
        lo_custkey = c_custkey
        and lo_suppkey = s_suppkey
        and lo_orderdate = d_datekey
        and (c_city=[CI1] or c_city=[CI2])
        and (s_city=[CI1] or s_city=[CI2])
        and d_yearmonth = [YM]
    group by
        c_city, s_city, d_year
    order by
        d_year asc, revenue desc;
```

Q:4.1

```
    select
        d_year, c_nation,
        sum(lo_revenue - lo_supplycost) as profit
    from
        DATES, CUSTOMER, SUPPLIER, PART, LINEORDER
    where
        lo_custkey = c_custkey
        and lo_suppkey = s_suppkey
        and lo_partkey = p_partkey
        and lo_orderdate = d_datekey
        and c_region = [R]
        and s_region = [R]
        and (p_mfgr = [MFGR1] or p_mfgr = [MFGR2])
    group by
        d_year, c_nation
    order by
        d_year, c_nation;
```

Q:4.2

```
    select
        d_year, s_nation, p_category,
        sum(lo_revenue - lo_supplycost) as profit
    from
        DATES, CUSTOMER, SUPPLIER, PART, LINEORDER
    where
        lo_custkey = c_custkey
        and lo_suppkey = s_suppkey
        and lo_partkey = p_partkey
        and lo_orderdate = d_datekey
        and c_region = [R]
        and s_region = [R]
        and (d_year = [Y1] or d_year = [Y2])
        and (p_mfgr = [MFGR1] or p_mfgr = [MFGR2])
    group by
        d_year, s_nation, p_category
    order by
        d_year, s_nation, p_category;
```

Q:4.3

```
    select
        d_year, s_city, p_brand,
        sum(lo_revenue - lo_supplycost) as profit
    from
        DATES, CUSTOMER, SUPPLIER, PART, LINEORDER
    where
        lo_custkey = c_custkey
        and lo_suppkey = s_suppkey
        and lo_partkey = p_partkey
        and lo_orderdate = d_datekey
        and s_nation = [N]
        and (d_year = [Y1] or d_year = [Y2])
        and p_category = [C]
    group by
        d_year, s_city, p_brand
    order by
        d_year, s_city, p_brand;
```

hMetiS: A hypergraph partitioning Package

VIII. Introduction

A hypergraph is a generalization of a graph, where the set of edges is replaced by a set of hyperedges. A hyperedge extends the notion of an edge by allowing more than two vertices to be connected by a hyperedge.
Hypergraph partitioning is an important problem and has extensive applications in many domains like VLSI design [11], transportation management, and data-mining [123], defining of molecular structures in chemistry search [160], supervise of cellular mobile communication [159], image segmentation [54, 97], etc.
Hypergraph partitioning is to partition the vertices of a hypergraph in k-roughly equal parts, such that the number of hyperedges connecting vertices in different parts is minimized.
In this chapter, we detail a hypergraph partitioning package, called *hMeTiS* [151, 148] [1], used in our works.

IX. Overview of *hMeTiS*

IX.1. *hMeTiS* algorithms

hMeTiS is a software package for partitioning large hypergraphs, especially those arising in circuit design. The algorithms in *hMeTiS* are based on multilevel hypergraph partitioning described in [147, 149, 151]. Multilevel partitioning algorithms are illustrated in chapter the third chapter (section II), that reduce the size of the graph (or hypergraph) by collapsing vertices

[1] *hMeTiS* is copyrighted by the regents of the University of Minnesota. This work was supported by IST/B-MDO and by Army High Performance Computing Research Center under the auspices of the Department of the Army

and edges, partition the smaller graph (initial partitioning phase), and then uncoarsen it to construct a partition for the original graph (uncoarsening and refinement phase). The highly tuned algorithms employed on *hMeTiS* allow it to quickly produce high-quality partitions for a large variety of hypergraphs.

IX.2. Advantages of *hMeTiS*

The advantages of *hMeTiS* compared to other similar algorithms are the following:

- **Quality of partitions :** Experiments on a large number of hypergraphs arising in various domains including VLSI, databases, and data mining show that *hMeTiS* produces partitions that are consistently better than those produced by other widely used algorithms [150].

- **Consuming of execution time :** Experiments on a wide range of hypergraphs has shown that *hMeTiS* is one to two orders of magnitude faster than other widely used partitioning algorithms.

X. Using of hmetis program

The program *hMeTiS* is invoked by providing 9 or 10 command line arguments as follows:
hmetis *HGraphFile Nparts UBfactor Nruns CType RType Vcycle Reconst dbglvl*
or
hmetis *HGraphFile FixFile Nparts UBfactor Nruns CType RType Vcycle Reconst dbglvl*
The meaning of the various parameters is as follows:

HGraphFile : is the name of the file that stores the hypergraph.

FixFile is the name of the file that stores information about the pre-assignment of vertices to partitions.

Nparts: is the number of desired partitions. *hMeTiS* can partition a hypergraph into an arbitrary number of partitions, using recursive bisection. That is, for a 4-way partition.

UBfactor: This parameter is used to specify the allowed imbalance between the partitions during recursive bisection. This is an integer number between 1 and 49, and specifies the allowed load imbalance in the following way. Consider a hypergraph with n vertices, each having a unit weight, and let b be the UBfactor. The number of vertices assigned to each one of the two partitions will be between (50 - *UBfactor*)n/100 and (50 + *UBfactor*)n/100, at each bi-sections.

Nruns: This is the number of the different bisections that are performed by *hMeTiS*. It is a number greater or equal to one, and instructs *hMeTiS* to compute *Nruns* different bisections, and select the best as the final solution. A default value of 10 is used.

CType: is the type of vertex grouping schema to use during the coarsening phase. It is an integer parameter and the possible values are:

– 1: means that *hMeTiS* selects the hybrid first-choice schema. This schema is a combination of the first-choice and greedy first-choice schema.

– 2: means that *hMeTiS* selects the first-choice schema. In this schema, vertices are grouped together if they are present in multiple hyperedges. Groups of vertices of arbitrary size are allowed to be collapsed together.

– 3: means that *hMeTiS* selects the greedy first-choice schema. In this schema, vertices are grouped based on the first choice schema, but the grouping is biased in favor of faster reduction in the number of the hyperedges that remain in the coarse hypergraphs.

– 4: means that *hMeTiS* selects the hyperedge schema. In this schemz vertices are grouped together that correspond to entire hyperedges.

– 5: means that *hMeTiS* selects the edge schema. In this schema, pairs of vertices are grouped together if they are connected by multiple hyperedges.

You may have to experiment with this parameter to see which schema works better for the classes of hypergraphs that you are using. *In our experiments we have used the value 1.*

RType: This is the type of refinement policy to use during the uncoarsening phase. It is an integer parameter and the possible values are:

– 1: means that *hMeTiS* selects the Fiduccia-Mattheyses (*FM*) refinement schema used by *hMeTiS* tool.

– 2: means that *hMeTiS* selects the one-way *FM* refinement schema, where, during each iteration of the *FM* algorithm, vertices are allowed to move only in a single direction.

– 3: means that *hMeTiS* selects the early-exit *FM* refinement schema. In this case, the FM iteration is aborted if the quality of the solution does not improve after a relatively small number of vertex moves.

In our case, we have used the value 1 that correspondent to using *Fiduccia-Mattheyses* that is recommended by the developer of *hMeTiS* [150].

Vcycle: This parameter selects the type of V-cycle refinement to be used by the algorithm. It is an integer parameter and the possible values are:

- 0: means that *hMeTiS* does not perform any form of V -cycle refinement.
- means that *hMeTiS* performs V -cycle refinement on the final solution of each bisection step.
- 2: means that *hMeTiS* performs V -cycle refinement on each intermediate solution whose quality is equally good or better than the best found so far.
- 3: means that *hMeTiS* performs V -cycle refinement on each intermediate solution. That is, each one of the Nruns bisections is also refined using V-cycles.

In our experiments , we have used the third choice that best time/quality tradeoffs.

Reconst This parameter is used to select the schema to be used in dealing with hyperedges that are being cut during the recursive bisection. It is an integer parameter and the possible values are:

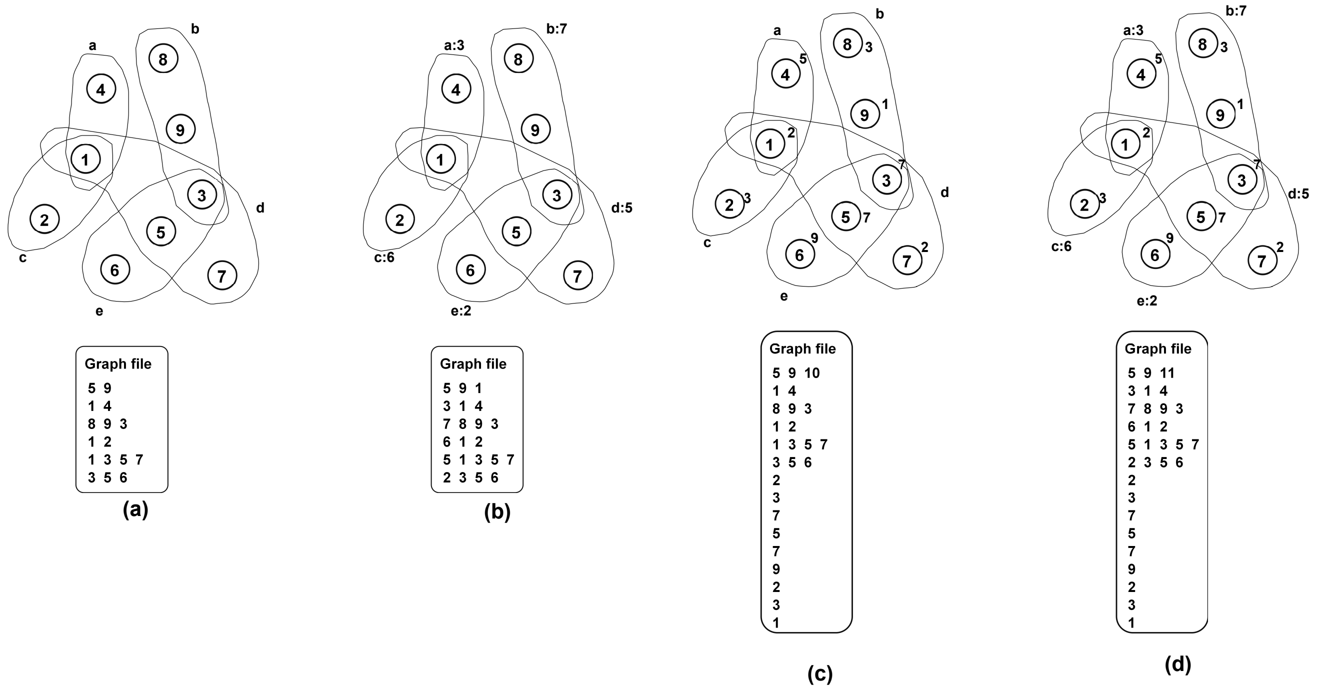
- 0: means that *hMeTiS* removes any hyperedges that were cut while constructing the two smaller hypergraphs in the recursive bisection step.
- 1: means that *hMeTiS* reconstructs the hyperedges that are being cut, so that each of the two partitions retain the portion of the hyperedge that corresponds to its set of vertices.

Reconst with the value 0, gives a good quality of k-way partitioning.

dbglvl This is used to request *hMeTiS* to print debugging information. The value of **dbglvl** is computed as the sum of codes associated with each option of *hMeTiS*. The various options and their values are as follows:

- 0: show no additional information.
- 1: show information about the coarsening phase.
- 2: show information about the initial partitioning phase.
- 4: show information about the refinement phase.
- 8: show information about the multiple runs.
- 16: show additional information about the multiple runs.

For example, if we want to see all information about the multiple runs the value of **dbglvl** should be $8 + 16 = 24$.



Graph file (a)

```
5 9
1 4
8 9 3
1 2
1 3 5 7
3 5 6
```

Graph file (b)

```
5 9 1
3 1 4
7 8 9 3
6 1 2
5 1 3 5 7
2 3 5 6
```

Graph file (c)

```
5 9 10
1 4
8 9 3
1 2
1 3 5 7
3 5 6
2
3
7
5
7
9
2
3
1
```

Graph file (d)

```
5 9 11
3 1 4
7 8 9 3
6 1 2
5 1 3 5 7
2 3 5 6
2
3
7
5
7
9
2
3
1
```

Fig. 7.3 *– Graph File representation for (a) unweighted hypergraph, (b) weighted hyperedges, (c) weighted vertices, and (d) weighted hyperedges and vertices*

X.1. Format of hypergraph file

The primary input of *hMeTiS* is the hypergraph to be partitioned. A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E}$) with $\mathcal{V}$ vertices and $\mathcal{E}$ hyperedges is stored in a plain text file that contains $|\mathcal{E}| + 1$ lines, if there are no weights on the vertices and $|\mathcal{E}| + |\mathcal{V}| + 1$ lines if there are weights on the vertices. The first line contains either two or three integers. The first integer is the number of hyperedges ($|\mathcal{E}|$), the second is the number of vertices ($|\mathcal{V}|$), and the third integer (*fmt*) contains information about the type of the hypergraph. In particular, depending on the value of *fmt*, the hypergraph $|\mathcal{H}|$ can have weights on either the hyperedges (*fmt*=1, shows Figure 7.3-b), the vertices(*fmt*=10, shows Figure 7.3-c), or both (*fmt*=11, shows Figure 7.3)-d. In the case that H is unweighted, *fmt* is omitted (shows Figure 7.3-a).

X.2. Format of the Fix File

The FixFile is used to specify the vertices that are pre-assigned to certain partitions. In general, when computing a k-way partitioning, up to k sets of vertices can be specified, such that each set is pre-assigned to one of the k partitions. For a hypergraph with $|\mathcal{V}|$ vertices, the FixFile consists of $|\mathcal{V}|$ lines with a single number per line. The i th line of the file contains either the partition number to which the i th vertex is pre-assigned to, or -1 if that vertex can be assigned to any partition . Note that the partition numbers start from 0.

X.3. Format of Output File

The output of *hMeTiS* is a partition file. The partition file of a hypergraph with $|\mathcal{V}|$ vertices, consists of $|\mathcal{V}|$ lines with a single number per line. The i^{th} line of the file contains the partition number that the i^{th} vertex belongs to.

X.4. *hMeTiS* Library Interface

The hypergraph partitioning algorithms in *hMeTiS* can also be accessed directly using the stand-alone library *libhmetis.a.* This library provides the routine *HMETIS_PartRecursive().* The calling sequences and the description of the various parameters of this routines are as follows:
HMETIS_PartRecursive (int nvtxs, int nhedges, int *vwgts, int *eptr, int *eind, int *hewgts, int nparts, int ubfactor, int *options, int *part, int *edgecut)

nvtxs, nhedges: The number of vertices and hyperedges in the hypergraph, respectively.

vwgts: An array that stores the weight of the vertices. Specifically, the weight of vertex i is stored at **vwgts**[i]. If the vertices in the hypergraph are unweighted, then **vwgts** can be NULL.

eptr, eind: Two arrays that are used to describe the hyperedges in the graph. The first array, **eptr**, is of size nhedges+1, and it is used to index the second array **eind** that stores the actual hyperedges. Each hyperedge is stored as a sequence of the vertices that it spans, in consecutive locations in **eind**. Specifically, the i^{th} hyperedge is stored starting at location **eind**[**eptr**[i]] up to **eind**[**eptr**[i +1]].

hewgts: An array of size nhedges that stores the weight of the hyperedges. The weight of the i hyperedge is stored at location *[*i]. If the hyperedges in the hypergraph are unweighted, then hewgts can be NULL.

nparts: The number of desired partitions.

ubfactor: This is the relative imbalance factor to be used at each bisection step. Its meaning is identical to the **UBfactor** parameter described above.

options: This is an array of 9 integers that is used to pass parameters for the various phases of the algorithm. If options[0]=0 then default values are used. If options[0]=1, then the remaining elements of options are interpreted as follows:

– options[1]: is the **Nruns** parameter of *hMeTiS*, described above.

- options[2]: is the **CType** parameter of *hMeTiS*, described above.
- options[3]: is the **RType** parameter of *hMeTiS*, described above.
- options[4]: is the **Vcycle** parameter of *hMeTiS*, described above.
- options[5]: is the **Reconst** parameter of *hMeTiS*, described above.
- options[6]: determines whether or not there are sets of vertices that need to be pre-assigned to certain partitions. A value of 0 indicates that no pre-assignment is desired, whereas a value of 1 indicates that there are sets of vertices that need to be pre-assigned.
- options[7]: determines the random seed to be used to initialize the random number generator of *hMeTiS*. A negative value indicates that a randomly generated seed should be used (default behavior).
- options[8]: is the **dbglvl** parameter of *hMeTiS*, described above.

XI. General Guidelines

The *hMeTiS* program allows controlling the multilevel hypergraph bisection paradigm by providing a variety of algorithms for performing the various phases. In particular, it allows checking the following tasks:

1. How the vertices are grouped together during the coarsening phase. This is done by using the *CType* parameter.

2. How the quality of the bisection is refinement during the uncoarsening phase. This is done by using the *RType* parameter.

Depending on the classes of the hypergraphs that are partitioned, these default settings may not necessarily be optimal. You should experiment with these parameters to see which schemes work better for your classes of problems.

XII. System Requirements

hMeTiS has been written in C language and it has been extensively tested on Sun, SGI, Linux, and IBM. Even though, *hMeTiS* contains no known bugs. *hMeTiS* and its updates are made available at: `http://www.cs.umn.edu/~metis`.

Relational Algebra

As the traditional relational queries, ROLAP queries are based on relational algebra which is one of the two formal query languages associated with the relational model [207]. Queries in this algebra can be viewed as a composition of a collection of operators (called a query expression). These operators are classified into two main categories:

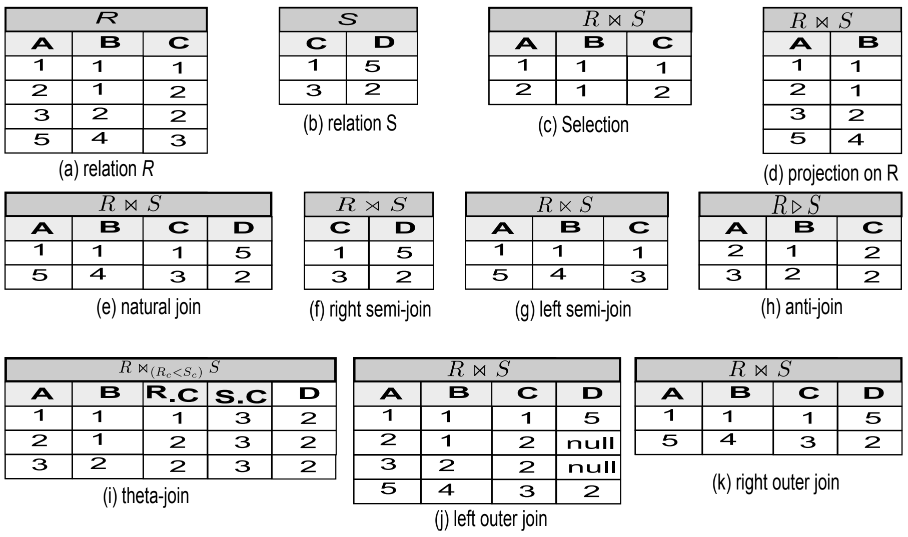
- *unary operations* (e.g., selection(σ), projection ($\prod$)) need only one relation/intermediate result;
- *binary operations* (e.g., join ($\bowtie$), union, intersection, etc.) need two relations/intermediate results.

A *query expression* can be represented by a *tree*, whose leaves and internal nodes represent respectively: base tables and relational algebra operators applied to node's children. The tree is executed from leaves to root. For a given tree, there exists numerous equivalent query trees (due the properties (laws) of the relational algebra). This makes its optimization *undecidable.*

- **Commutative and associative laws:** the most of the operators (Join, Union, Intersection, Cross Product) are commutative and associative.
- **Selection laws:** One of the most important action of a query optimizer (specially in selecting a good logical query plan) is to push down selections in a query in order to minimize the size of the manipulated relations. To perform this operations, some laws exist:
  - **splitting laws:** when the selection involves multiple predicates connected by *AND* or *OR*, the condition will be broken into many sub-conditions as follows:

$$\sigma_{C_1 \text{ AND } C_2}(R) = \sigma_{C_1}(\sigma_{C_2}(R)) \tag{7.40}$$

$$\sigma_{C_1 \text{ OR } C_2}(R) = \sigma_{C_1}(\sigma_{C_2}(R)) \tag{7.41}$$



R

A	B	C
1	1	1
2	1	2
3	2	2
5	4	3

(a) relation *R*

S

C	D
1	5
3	2

(b) relation S

$R \bowtie S$

A	B	C
1	1	1
2	1	2

(c) Selection

$R \bowtie S$

A	B
1	1
2	1
3	2
5	4

(d) projection on R

$R \bowtie S$

A	B	C	D
1	1	1	5
5	4	3	2

(e) natural join

$R \rtimes S$

C	D
1	5
3	2

(f) right semi-join

$R \ltimes S$

A	B	C
1	1	1
5	4	3

(g) left semi-join

$R \triangleright S$

A	B	C
2	1	2
3	2	2

(h) anti-join

$R \bowtie_{(R_c<S_c)} S$

A	B	R.C	S.C	D
1	1	1	3	2
2	1	2	3	2
3	2	2	3	2

(i) theta-join

$R \bowtie S$

A	B	C	D
1	1	1	5
2	1	2	null
3	2	2	null
5	4	3	2

(j) left outer join

$R \bowtie S$

A	B	C	D
1	1	1	5
5	4	3	2

(k) right outer join

Fig. 7.4 *– Overview of different operator in relational model*

$X \cup \emptyset = X$, $X \cup X = X$, $X \cap \emptyset = X$, $X \cap X = X$, $X \setminus \emptyset = X$, $\emptyset \setminus X = \emptyset$, $X \setminus X = \emptyset$,	Basic laws
$X \cup Y = Y \cup X$, $X \cap Y = Y \cap X$, $X \setminus Y \neq Y \setminus X$	Commutativity
$X \cup (Y \cup Z) = (X \cup Y) \cup Z$, $X \cap (Y \cap Z) = (X \cap Y) \cap Z$, $X \setminus (Y \setminus Z) \neq (X \setminus Y) \setminus Z$	Associativity
$X \cup (Y \cap Z) = (X \cap Y) \cup (X \cap Z)$, $X \cap (Y \cup Z) = (X \cup Y) \cap (X \cup Z)$, $(X \cup Y) \setminus Z) = (X \setminus Z) \cup (Y \setminus Z)$,	Distributivity

Table 7.5 *– Overviews of laws for Set operations*

- **ordering laws:** the order of sequence of selections is not important (commutative properties). $\sigma_{c_1}(\sigma_{c_2}(R)) = \sigma_{c_2}(\sigma_{c_1}(R))$
- **pushing laws:** there are three types of pushing selection through binary operations, depending on which arguments the selection is pushed.
  - the selection must be pushed to the both of arguments (like in the union operation);
  - the selection must be pushed to one of argument and the second argument is optional (like in the intersection and difference operations)
  - the selection must be pushed only to one argument (like join cross product operations).

Example 5. *Let us Consider two relations R (a,b) and S (b,c) and the expression* $\sigma_{a=2\ AND\ b<val}(R \bowtie S)$*. The condition (a=2) applies only to R and the condition (b<val) applies only to S. Pushing process needs three steps:*

N	**Algebra properties**
01	$\sigma_{p_1 \wedge p_2}(R) \equiv \sigma_{p_1}(\sigma_{p_2}(R))$
02	$\sigma_{p_1}(\sigma_{p_2}(R)) \equiv \sigma_{p_2}(\sigma_{p_1}(R))$
03	$R \bowtie_p S \equiv S \bowtie_p R$
04	$R \times S \equiv S \times R$
05	$R \times (S \times T) \equiv (R \times S) \times T$
06	$R_1 \bowtie_{p_{1,2}} (R_2 \bowtie_{p_{2,3}} R_3) \equiv (R_1 \bowtie_{p_{1,2}} R_2) \bowtie_{p_{2,3}} R_3$
07	$\sigma_p(R \times S) \equiv \sigma_p(R) \times S$
08	$\sigma_p(R \bowtie_q S) \equiv \sigma_p(R) \bowtie_q S$
09	$\Pi_A(R \bowtie_p S) \equiv \Pi_{A_1}(R) \bowtie_p \Pi_{A_2}(S)$
10	$\Pi_A(R \times S) \equiv \Pi_{A_1}(R) \times \Pi_{A_2}(S)$
11	$\sigma_p(R\theta S) \equiv \sigma_p(R)\ \theta\ \sigma_p(S)$ where $\theta = \{\cup, \cap, \setminus\}$
12	$\sigma_p(R \times S) \equiv R \bowtie_p S$
13	$R \bowtie_C S \equiv \sigma_C(R \times S)$

***Table 7.6** – Relation algebra properties*

- *splitting selection using the logical operator AND, the expression becomes* $\sigma_{a=2}(\sigma_{b<val}(R \bowtie S))$*;*
- *push the selection with the condition* $b<val$ *to* S*, the expression becomes* $\sigma_{a=2}(R \bowtie \sigma_{b<val}(S))$*;*
- *push the first condition to* R*, the expression becomes* $\sigma_{a=2}(R) \bowtie \sigma_{b<val}(S)$*.*

- **Projection laws:** the projection can be pushed down like selection to minimize the size of relation by reducing the size of tuples and then minimizing the cost of query processing. Pushing down projection must verifies some conditions:
  - the projection in the expression must eliminate only the attributes that are not used by all above operators;
  - the projection must add the join attributes if there is above join operators;
  - the projection must add all attributes used to define a condition in above operators (join, selection, etc.).
- **Joins laws** More than the join operators are commutative and associative, they can be defined using selection on one or both relations that verify a join condition. So, $R \bowtie_C S = \sigma_C(R \times S)$

Résumé

Contexte

Aujourd'hui, une grande quantité de données sont générées en continu dans de nombreux de domaines tels que la science de la vie, l'exploitation des ressources naturelles, la prévention des catastrophes naturelles, l'optimisation des flux de circulation, les réseaux sociaux, la concurrence commerciale, etc. Ces données doivent être collectées, stockées et analysées afin d'être bien exploité par les gestionnaires des entreprises et les chercheurs pour effectuer leurs tâches quotidiennes. Une autre réalité, est que ces données souvent nécessitent d'être partagées avec d'autres utilisateurs et communautés. Où le partage de données commence à devenir un besoin vital pour la réussite de plusieurs entreprises. Par exemple, l'Autorité de la concurrence et des marchés (CMA) a ordonner les grandes banques de Grande-Bretagne à adopter l'ouverture des principes bancaires pour augmenter le partage des données entre les organismes de services financiers[2]. Le partage de données entre un nombre important d'utilisateurs impliquera l'apparition du phénomène d'un nombre volumineux de requêtes que les systèmes de stockage et les systèmes de gestion de base de données doivent faire face. Ce nombre est motivé par l'explosion de site web de E-commerce, média sociaux, etc. A titre d'exemple, Amazon, 100 million de requêtes peut arriver à ses bases de données. Le même constat est détecté sur le géant chinois d'E-commerce Alibaba. Ces requêtes sont complexes et répétitives et nécessitent des opérations lourdes comme les jointures et les agrégations. Aussi, la gestion de la recommandation et de l'exploration des contribue fortement à augmenter le nombre de requêtes d'un SGBD particulier [99]. Partant de ressources de calcul et les perspectives des systèmes de stockage, les systèmes de stockage des bases de données optent vers la décentralisation des produits pour être capable d'évoluer en termes de capacité, de puissance de traitement, et de débit de communication. Des efforts ont été déployés pour concevoir ces systèmes pour partager simultanément les res-

[2] http://www.experian.co.uk/blogs/latest-thinking/a-new-era-of-data-sharing-how-the-cma-is-shaking-up-retail-banking/

sources physiques et les données entre les applications [106]. Le Cloud Computing a largement contribué à augmenter les capacités de partage de ces systèmes grâce à leurs caractéristiques : l'élasticité, la souplesse, le stockage et le calcule à la demande. Par conséquent, les données et les requêtes sont devenues liées aux plateformes avancées de déploiement, comme le parallèle, clusters, etc. [36]. Concevoir une base de données dans une telle plateformes exige des phases sensibles telles que : (1) la fragmentation de données on plusieurs fragments ; (2) l'allocation de ces fragments dans leurs nœuds de calcules correspondants, et (3) la stratégie d'exécution des requêtes, pour assurer l'équilibre de charge des requêtes.

En outre, Les requêtes qui représentent l'un des entités les plus importants dans la technologie des bases de données, sont aussi concernés par le phénomène de partage de données. Le partage entre les requêtes connus sous le nom : Interaction de requêtes. Mr Timos Sellis a donné la naissance d'un nouveau problème appelée optimisation multi-requêtes, qui vise à optimiser l'exécution globale d'une charge de requêtes par l'augmentation de la réutilisation des résultats intermédiaires des requêtes [223]. Cette interaction de requêtes a largement contribué à l'optimisation des requêtes [110, 273] et à la conception physique [191, 7]). *Mais ces efforts ont ignorés le volume de requêtes.*

Le volume de données a un impact sérieux les performances des requêtes. Avoir des méthodes efficaces pour accélérer le traitement des données devient plus important et urgent que jamais, et par conséquent, ils reçoivent une attention particulière de chercheurs universitaires et industriels. Ceci est bien illustré par le grand nombre de propositions, vise à optimiser les requêtes au niveau logique [227, 167, 236, 168, 210, 266, 85, 224], au niveau physique [279, 79, 14, 33, 73, 81, 154] et au niveau de déploiement [36].

En se basant sur cette discussion, nous prétendons que tout système de stockage de données doit faire face à trois dimensions communes : volume de données, le volume de requêtes (nous l'appelons big-queries) et les questions de partage de la requête. Face à cette situation, le partage de la requête traditionnelle doit être revu pour intégrer les requêtes nombreuses. Par conséquent, nous aimerions émettre un groupe de réflexions sur le partage entre les requêtes de grande quantité. Ce groupe de réflexion est discuté dans le contexte des entrepôts de données relationnels, généralement modélisées par un schéma en étoile ou ses variantes (ex. schémas de flocon de neige) qui fournissent une vue centrée de requête sur les données. Une autre caractéristique importante est que ce schéma augmente l'interaction entre les requêtes OLAP, car ils effectuent des jointures impliquant une (des) table(s) de fait centrale(s) et tables périphériques, appelés dimensions.

Pour faire face au volume de requêtes interagi, le développement des structures de données évolutif qui capturent le partage est largement recommandé dans la définition des algorithmes intelligents. La théorie des graphes et ses structures de données à contribué largement à résoudre des problèmes complexes impliquant à grande espace de recherche, telles que (1) le fragmentation vertical et horizontale dans les bases de données relationnelles[186, 184], (2) le fragmentation horizontale dans les bases de données orientés objets [29] (3), et récemment, Curino et al. [81] ont proposé la généralisation du concept de graph par un hypergraphe pour

partitionner et répliquer les instances d'une base de données dans le cas distribuée. Où les sommets et les hyperarêtes de l'hypergraphe graphe représentent respectivement les instances de la base de données et les transactions. Un tel graphe peut être très vaste, puisqu'il est associé au nombre de tuples d'une base de données. Cela a rendu son partitionnement nécessaire pour assurer le passage à l'échelle de ses algorithmes.
Le partitionnement de Hypergraphe est bien étudié par le Design Automation électronique (de eda) pour tester le circuit électronique, en outre, une grande panoplie de bibliothèques de partitionnement existent tels que hMETIS et Zoltan-PHG. Cela nous motive à explorer hypergraphes pour faire face le phénomène du grand volume de requêtes. En conséquence, nous avons approché à la société Graphics Mentors situé à Grenoble- France, un leader de l'automatisation de la conception électronique, pour obtenir leur expertise dans l'utilisation de hypergraphe et d'élaborer une analogie entre notre problème et leur conclusion. Cette collaboration a fructueuse, puisque, après plusieurs mois de collaboration, nous réussissons à trouver une analogie entre un circuit électronique et un plan unifiée de requêtes, en le présentant comme un hypergraphe. Nous approprions aussi leurs outils. Une fois que ce paragraphe est généré, nous le considérons dans l'optimisation logique et physique et de la phase de déploiement du cycle de vie d'une conception de l'entrepôt de données. Cette riche expérience sera entièrement discutée dans cette thèse.

Objectifs de la thèse

Les principaux objectifs fixés pour cette thèse sont :

- En raison du large éventail de sujets que nous avons étudiés dans cette thèse qui couvre : l'optimisations logiques de requêtes, l'optimisations physiques, l'entreposage de données, lasélection de structures d'optimisation ; l'optimisation multi-requête, la phase de déploiement, les hypergraphes, etc., il est nécessaire de présenter une étude de synthèse de tous ces aspects.

- Puisque nous traitons le phénomène de big-queries et leurs interactions, nous devons montrer l'importance d'une structure de données évolutive qui capte facilement cette interaction, et utilisable comme un support pour définir des algorithmes intelligents avec moins de calculs et de haute qualité.

- Nous devons également tenir compte des études de cas pour déployer notre approche. Pour ce faire, nous considérons le problème de la sélection de deux optimisations structures qui sont les vues matérialisées et le fragmentation horizontal pour environnement centralisé et parallèle.

- Nous avons également développé un outil pour aider les concepteurs et les administrateurs de base de données pour perfectionner la conception physique en se basant sur l'interaction entre les requêtes.

Solutions proposées

Cette section présente les contributions de la thèse. Tout d'abord, nous discutons la conception typique de l'entrepôt de données pour fournir une compréhension générale des choix de conception et d'architectures de déploiement. Nous nous concentrerons sur l'optimisation logique et physique, parce qu'ils sont les tâches les plus importantes d'optimiseurs de requêtes. Un autre point concerne la phase de déploiement du cycle de vie de l'entrepôt de données.
A partir de ces vues d'ensemble, nous discutons de l'origine de notre structure de données pour représenter l'interaction des requêtes émise à partir de notre collaboration avec la société *Mentors Graphics*. Nous aimerions partager cette expérience et les efforts fournis pour trouver une analogie entre notre problème et le circuit électronique.
Parallèlement à la définition de cette structure, de nombreux autres développements ont été menées pour comprendre les algorithmes et les outils utilisés par la communauté VLSI et de les adapter à notre contexte. Une fois, cette adaptation a été fait, nous considérons deux problèmes dirigé par la charge de requêtes : la sélection des vues matérialisées et la détermination du schéma de la fragmentation horizontal. De plus, nous examinons comment améliorer le cycle de vie de la phase de déploiement en intégrant notre constatation selon laquelle notre structure de données et ses algorithmes sont associés. Enfin, nous étudions le passage à l'échelle de nos algorithmes et la qualité de leurs solutions, en les soulignant en termes de taille de base de données, la taille des requêtes, etc.

- **Enquête sur l'optimisation logiques et physique :** Nous procédons à une analyse en profondeur de la façon dont l'état de l'art des bases de données gère l'interaction entre les requêtes et comment elle est utilisée dans l'optimisation logique et physique, et la phase de déploiement du cycle de vie de l'entrepôt de données. En se basant sur cette enquête, nous donnons de nouvelles classifications des études existantes et les techniques utilisées dans le cadre de l'optimisation des requêtes, et une clarification du processus de sélection des structures d'optimisation lors de la phase de conception physique, qui est considéré comme l'un de la phase majeure. Enfin, nous proposons une méthodologie de référence pour traiter le processus de sélection de ces optimisations en présence de l'interaction de la requête.

- **Hypergraphs comme structure de données :**

  Faire l'analogie entre notre problème de génération d'un plan unifié de requêtes et le problème de conception des circuits électroniques. Cette analogie nous permet d'emprunter des techniques et des outils de la communauté EDA et en les adaptant à la génération d'un plan unifié de requêtes. Où le problème est modélésé en utilisant les hypergraphe qui fournissent support pour définir des algorithmes qui passe à l'échelle. Cette évolutivité est généralement assurée par des techniques de partitionnement de graphes [196]. Plusieurs outils de partitionnement des hypergraphes ont été proposés, comme : PaToH [66] et hMETIS [3]. Notre conclusion est ensuite exploitée pour résoudre le problème d'optimisation multi-requêtes et d'évaluer son efficacité et qualité contre les approches les plus populaires de l'art state-of.

- **What-if pour la génération du plan unifié de requêtes :**

  L'interaction de requêtes représentées par un UQP est située à l'intersection de deux mondes : le mode de l'optimisation multiple de requêtes et le monde de conception physique. L'interaction de requête participe efficacement à résoudre les problèmes de conception physiques par la proposition d'un ensemble de candidats. Mais un UQP n'est pas nécessaire qu'il est bien pour les instances du conception physique [273, 274], d'où la nécessité de choisir un bon plan pour produire de bons candidats de structures d'optimisation. En outre, un bon UQP pour une structure d'optimisation, ne soit pas nécessaire bon pour une autre structure. En plus, la sélection du meilleur UQP exige l'énumération de tous les plans possibles, ce qui est impossible en raison du grand nombre de plans possibles, ce qui peut être infinie dans le cas de Big-queries. De l'autre côté, le nombre de candidats proposés par un UQP dans le contexte de Big-query, peut être très grand. En conséquence, trouver la configuration optimale de la structure d'optimisation devient très difficile, voire impossible (complexité des algorithmes combinatoires).

  Pour surmonter ces problèmes, nous avons proposé une nouvelle approche qui utilise l'hypergraphes et les algorithmes de partitionnement pour intègre les connaissances liées aux structures d'optimisation lors de la génération du UQP orienté structure d'optimisation, afin de minimiser leur coût d'énumération. Notre proposition capture l'interaction de la requête dans un UQP, en divisant les problèmes initiaux dans plusieurs petits sous-problèmes disjoints, qui peuvent être exécutées en parallèle. La division de l'espace de recherche du problème de l'optimisation multiple de requêtes implique la division de l'ensemble des éléments candidats dans plusieurs petits sous-ensembles, ce qui minimise la complexité des algorithmes combinatoires pour sélectionner la meilleure configuration d'une structure d'optimisation.

  Pour évaluer l'efficience et l'efficacité de notre approche, nous considérons deux structures d'optimisation traditionnelles : vues matérialisées et la fragmentation horizontal.

- **L'interaction de requêtes dans la phase de déploiement :** Comme les requêtes sont utilisées par toutes les phases de déploiement : fragmentation de données, allocation de données et équilibrage de charge, nous pensons que cela pourrait être intéressant de pousser notre réflexion d'intégrer notre structure de hypergraphe dans tous les problèmes sensibles de la requête. En complétant le travail de déploiement d'un entrepôt de données dans une architecture parallèle du Soumia BENKRID [36], réalisé dans notre lab, la dimension Interaction de requêtes a été intégrés à ses algorithmes, en particulier l'allocation et la fragmentation horizontale de données.

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

- **Le développement d'un outil d'aide "Big-queries" :**

  Sur la base de nos conclusions, nous avons développé un outil d'aide, inspiré des outils bien connus développés par des éditeurs commerciaux et universitaires pour aider les concepteurs et les DBAs au cours de leurs activités de déploiement et durant les tâches d'administration, comme la sélection des structures d'optimisation. Cette interface permet au DBA de visualiser l'état de sa base de données qui concerne trois aspect : (1) les tables du $\mathcal{DW}$ en question, (2) la charge de requêtes utilisée, (3) les ressources requises pour la conception physique, (4) l'interaction de requêtes visualisée par un hypergraph, le schéma des différent structures d'optimisation ($\mathcal{MV}$ et $\mathcal{HDP}$), le schéma de déploiement et le coût relative. L'outil est doté d'une passerelle de connexion au différent type de SGBD (oracle11g, Oracle12c et PostgresSQl).

  L'outil big queries est disponible dans le forge de notre laboratoire `http://www.lias-lab.fr/forge/projects/bigqueries`.

Plan du Mémoire

La première partie du manuscrit contient le chapitre qui fournit le contexte nécessaire pour comprendre nos contributions. Dans ce chapitre, nous commençons par un zoom sur la technologie d'entreposage de données sur lesquelles nos expériences de validation sont effectuées. Par la suite, nous détaillons les paramètres qui influent sur le temps de réponse de traitement des requêtes : vues matérialisées, fragmentation de données et le problème d'optimisation multiple de requêtes, et nous donnons un aperçu de la conception de l'entrepôt de données parallèle.
La deuxième partie du manuscrit est consacrée à la contribution. Elle commence par le chapitre 3, qui détail notre principale contribution qui consiste à utiliser la théorie des graphes pour surmonter la gestion des grands-requêtes. La contribution vise à la fusion de la MQO et les problèmes de conception physique pour générer des candidats cibles pour une technique d'optimisation spécifique (MV, index, et fragmentation de données). Ceci est réalisé en injectant de la connaissance des structures d'optimisation dans la génération de UQP.
Le chapitre 4 présente l'application de notre UQP orienté structure d'optimisation, où il est appliqué pour la sélection des vues matérialisées avec et sans contraintes, et est exploité dans la matérialisation dynamique avec ordonnancement des requêtes. En plus, est utilisé pour la fragmentation horizontale de données.
Le chapitre 5 détaille notre approche de conception d'un entrepôt de données parallèle en tenant compte de l'interaction entre les requêtes, représentée par un plan unifié de de requêtes.
Le chapitre 6 donne une description de l'outil *Big-queries*, dévelopé dans notre laboratoire pour valider les contribution.
Le chapitre 7 résume et conclut la thèse en discutant les résultats.

Related Publications

Ahcène Boukorca and adjel Bellatreche and Sid-Ahmed Benali Senouci and Zoé Faget. Coupling Materialized View Selection to Multi Query Optimization: HyperGraph Approach. *International Journal of Data Warehousing and Mining (IJDWM)*, 11(2):62-84, 2015. [47]

Boukorca, Ahcène and Faget,Zoé and Bellatreche, Ladjel. What-if Physical Design for Multiple Query Plan Generation. *Proceedings of International Conference on Database and Expert Systems Applications (DEXA)*, pages 492–506, edited by LNCS Springer, 204. [52].

Ahcène Boukorca and Ladjel Bellatreche and Alfredo Cuzzocrea . SLEMAS: an approach for selecting materialized views under query scheduling constraints. *Proceedings of the 20th International Conference on Management of Data*, pages= 66–73, 2014. [49]

Dhouha Jemal, Rim Faiz, Ahcène Boukorca, Ladjel Bellatreche: MapReduce-DBMS: An Integration Model for Big Data Management and Optimization. *Proceedings of the 26th International Conference on Database and Expert Systems Applications (DEXA)*, pages= 430-439, , 2015. [141]

Ahcène Boukorca and Ladjel Bellatreche and Sid-Ahmed Benali Senouci and Zoé Faget. SONIC: Scalable Multi-query OptimizatioN through Integrated Circuits. *Proceedings of International Conference on Database and Expert Systems Applications (DEXA)*, pages 278-292, edited by LNCS Springer, 2013. [50].

Ahcène Boukorca and Ladjel Bellatreche and Sid-Ahmed Benali Senouci and Zoé Faget. Votre Plan d'Exécution de Requêtes est un Circuit Intégré : Changer de Métier. *Actes des 9èmes journées francophones sur les Entrepôts de Données et l'Analyse en ligne*, pages 133–148, edited by RNTI ,2013. [51].

Ladjel Bellatreche and Salmi Cheikh and Sebastian Breß and Amira Kerkad and Ahcène Boukorca and Jalil Boukhobza. How to exploit the device diversity and database interaction to propose a generic cost model?. *17th International Database Engineering & Applications Symposium, IDEAS '13*, pages 142-147, edited by BytePress/ACM and ACM's Digital Library, 2013. [26].

Ladjel Bellatreche and Sebastian Breß and Amira Kerkad and Ahcène Boukorca and Cheikh Salmi . The generalized physical design problem in data warehousing environment: towards a generic cost model. *36th International Convention on Information & Communication Technology Electronics & Microelectronics (MIPRO)*, pages 1131-1137, IEEE, 2013. [25]

Ramin Karimi and Ladjel Bellatreche and Patrick Girard and Ahcène Boukorca and András Hajdu. BINOS4DNA: Bitmap Indexes and NoSQL for Identifying Species with DNA Signatures through Metagenomics Samples. *Proceedings of International conferenceInformation on Technology in Bio- and Medical Informatics (ITBAM)*, pages 1-14, edited by LNCS Springer,2014. [145]

Submitted: other conference/journal articles are under submission.

List of Figures

List of Tables

Bibliography

[1] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 967–980. ACM, 2008.

[2] S. Abdellaoui, L. Bellatreche, and F. Nader. A quality-driven approach for building heterogeneous distributed databases: The case of data warehouses. In *IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 631–638, 2016.

[3] A. Abelló, J. Samos, and F. Saltor. Yam2: a multidimensional conceptual model extending uml. *International Journal on Information Systems*, 31(6):541–567, 2006.

[4] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala. Database tuning advisor for microsoft sql server 2005: Demo. In *ACM SIGMOD*, pages 930–932, 2005.

[5] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *Proceedings of the International Conference on Very Large DataBases (VLDB)*, pages 496–505. Morgan Kaufmann Publishers Inc., 2000.

[6] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 359–370. ACM, 2004.

[7] M. Ahmad, A. Aboulnaga, S. Babu, and K. Munagala. Interaction-aware scheduling of report-generation workloads. *VLDB Journal*, 20(4):589–615, 2011.

[8] Z. Akkaoui, J. Mazón, A. Vaisman, and A. Zimányi. Bpmn-based conceptual modeling of etl processes. In *DaWaK*, pages 1–14, 2012.

[9] J. Aligon, E. Gallinucci, M. Golfarelli, P. Marcel, and S. Rizzi. A collaborative filtering approach for recommending OLAP sessions. *Decision Support Systems*, 69:20–30, 2015.

[10] C. J. Alpert and A. B. Kahng. Multi-way partitioning via spacefilling curves and dynamic programming. In *Proceedings of the Design Automation Conference*, pages 652–657. ACM, 1994.

[11] C. J. Alpert and A. B. Kahng. Recent directions in netlist partitioning: a survey. *Integration, the VLSI journal*, 19(1):1–81, 1995.

[12] P. M. Apers. Data allocation in distributed database systems. *ACM Transactions on Database Systems (TODS)*, 13(3):263–304, 1988.

[13] E. Baralis, S. Baraboschi, and E. Teniente. Materialized view selection in a multidimensional database. In *Proceedings of the International Conference on Very Large DataBases (VLDB)*, pages 156–165, August 1997.

[14] X. Baril and Z. Bellahsene. Selection of materialized views: A cost-based approach. In *Advanced Information Systems Engineering*, pages 665–680. Springer, 2003.

[15] S. T. Barnard and H. D. Simon. Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: Practice and experience*, 6(2):101–117, 1994.

[16] R. Bayer. The universal b-tree for multidimensional indexing: General concepts. In *Worldwide Computing and Its Applications*, pages 198–209. Springer, 1997.

[17] M. Bayir, I. Toroslu, and A. Cosar. Genetic algorithm for the multiple-query optimization problem. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 37:147–153, 2007.

[18] L. Bellatreche. *Utilisation des vues materialisees, des index et de la fragmentation dans la conception logique et physique d'un entrepot de donnees.* PhD thesis, Universite Blaise Pascal Clermont Ferrand, 2000.

[19] L. Bellatreche, S. Benkrid, A. Ghazal, A. Crolotte, and A. Cuzzocrea. Verification of partitioning and allocation techniques on teradata DBMS. In *11th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, pages 158–169, 2011.

[20] L. Bellatreche and K. Boukhalfa. An evolutionary approach to schema partitioning selection in a data warehouse. In *Proceedings of International Conference on Data Warehousing and Knowledge Discovery (DaWaK)*, pages 115–125. Springer, 2005.

[21] L. Bellatreche, K. Boukhalfa, and Z. Alimazighi. Simulph.d.: A physical design simulator tool. In *20th International Conference on Database and Expert Systems Applications DEXA*, pages 263–270, 2009.

[22] L. Bellatreche, K. Boukhalfa, and M. K. Mohania. Pruning search space of physical database design. In *DEXA*, pages 479–488, 2007.

[23] L. Bellatreche, K. Boukhalfa, and P. Richard. Primary and referential horizontal partitioning selection problems: Concepts, algorithms and advisor tool. *Integrations of Data Warehousing, Data Mining and Database Technologies: Innovative Approaches*, 2011.

[24] L. Bellatreche, K. Boukhalfa, P. Richard, and K. Y. Woameno. Referential horizontal partitioning selection problem in data warehouses: Hardness study and selection algorithms. *International Journal of Data Warehousing and Mining (IJDWM)*, 5(4):1–23, 2009.

[25] L. Bellatreche, S. Breß, A. Kerkad, A. Boukorca, and C. Salmi. The generalized physical design problem in data warehousing environment: towards a generic cost model. In *International Convention on Information & Communication Technology Electronics & Microelectronics (MIPRO)*, pages 1131–1137. IEEE, 2013.

[26] L. Bellatreche, S. Cheikh, S. Breß, A. Kerkad, A. Boukorca, and J. Boukhobza. How to exploit the device diversity and database interaction to propose a generic cost model? In *17th International Database Engineering & Applications Symposium, IDEAS '13*, pages 142–147, 2013.

[27] L. Bellatreche, A. Cuzzocrea, and S. Benkrid. Effectively and efficiently designing and querying parallel relational data warehouses on heterogeneous database clusters: The f&a approach. *Journal of Database Management (JDM)*, 23(4):17–51, 2012.

[28] L. Bellatreche, K. Karlapalem, M. K. Mohania, and M. Schneider. What can partitioning do for your data warehouses and data marts? In *IDEAS*, pages 437–446, 2000.

[29] L. Bellatreche, K. Karlapalem, and A. Simonet. Algorithms and support for horizontal class partitioning in object-oriented databases. *Distributed and Parallel Databases*, 8(2):155–179, 2000.

[30] L. Bellatreche, A. Kerkad, S. Bress, and D. Geniet. Roupar: Routinely and mixed query-driven approach for data partitioning. In *OTM Confederated International Conferences On the Move to Meaningful Internet Systems*, volume 8185, pages 309–326. Springer Berlin Heidelberg, 2013.

[31] L. Bellatreche, S. Khouri, and N. Berkani. Semantic data warehouse design: From ETL to deployment à la carte. In *DASFAA*, pages 64–83, 2013.

[32] L. Bellatreche, R. Missaoui, H. Necir, and H. Drias. A data mining approach for selecting bitmap join indices. *JCSE*, 1(2):177–194, 2007.

[33] L. Bellatreche, M. Schneider, H. Lorinquer, and M. Mohania. Bringing together partitioning, materialized views and indexes to optimize performance of relational data warehouses. In *Proceedings of International Conference on Data Warehousing and Knowledge Discovery (DaWaK)*, pages 15–25. Springer, 2004.

[34] L. Bellatreche, A. Simonet, and M. Simonet. Vertical fragmentation in distributed object database systems with complex attributes and methods. In *Proceedings of International Conference on Database and Expert Systems Applications (DEXA)*, pages 15–21, 1996.

[35] R. G. Bello, K. Dias, A. Downing, J. Feenan, J. Finnerty, W. D. Norcott, H. Sun, A. Witkowski, and M. Ziauddin. Materialized views in oracle. In *Proceedings of the International Conference on Very Large DataBases (VLDB)*, volume 98, pages 24–27, 1998.

[36] S. Benkrid. *Le déploiement, une phase à part entière dans le cycle de vie des entrepôts de données : application aux plateformes parallèles.* PhD thesis, ISAE-ENSMA and ESI of Algeria, jun 2014.

[37] S. Benkrid, L. Bellatreche, and A. Cuzzocrea. A global paradigm for designing parallel relational data warehouses in distributed environments. *Trans. Large-Scale Data- and Knowledge-Centered Systems*, 15:64–101, 2014.

[38] C. Berge. *Hypergraphs: combinatorics of finite sets*, volume 45. Elsevier, 1984.

[39] N. Berkani, L. Bellatreche, and B. Benatallah. A value-added approach to design BI applications. In *DAWAK*, pages 361–375, 2016.

[40] M. Bery. *Conception physique des bases de données a base ontologique : le cas des vues matérialisées.* PhD thesis, ISAE-ENSMA, dec 2014.

[41] M. Blattner. B-rank: A top n recommendation algorithm. *arXiv preprint arXiv:0908.2741*, 2009.

[42] D. Boley, M. Gini, R. Gross, E.-H. S. Han, K. Hastings, G. Karypis, V. Kumar, B. Mobasher, and J. Moore. Partitioning-based clustering for web document categorization. *Decision Support Systems*, 27(3):329–341, 1999.

[43] E. G. Boman, Ü. V. Çatalyürek, C. Chevalier, and K. D. Devine. The zoltan and isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering and coloring. *Scientific Programming*, 20(2):129–150, 2012.

[44] W. Bosma, J. Cannon, and C. Playoust. The magma algebra system i: The user language. *Journal of Symbolic Computation*, 24(3):235–265, 1997.

[45] K. Boukhalfa. *De la conception physique aux outils d administration et de tuning des entrepôts de données*. PhD thesis, ISAE-ENSMA, jul 2009.

[46] I. Boukhari. *Intégration et exploitation de besoins en entreprise étendue fondÃ©es sur la sémantique*. PhD thesis, ISAE-ENSMA, jan 2014.

[47] A. Boukorca, adjel Bellatreche, S. B. Senouci, and Z. Faget. Coupling materialized view selection to multi query optimization: Hyper graph approach. *International Journal of Data Warehousing and Mining (IJDWM)*, 11(2):62–84, 2015.

[48] A. Boukorca, L. Bellatreche, and S. Benkrid. HYPAD: hyper-graph-driven approach for parallel data warehouse design. In *15th International on Conference,Algorithms and Architectures for Parallel Processing (ICA3PP)*, pages 770–783, 2015.

[49] A. Boukorca, L. Bellatreche, and A. Cuzzocrea. Slemas: an approach for selecting materialized views under query scheduling constraints. In *Proceedings of International Conference on Management of Data (COMAD)*, pages 66–73. Computer Society of India, 2014.

[50] A. Boukorca, L. Bellatreche, S.-A. B. Senouci, and Z. Faget. Sonic: Scalable multi-query optimization through integrated circuits. In *Proceedings of International Conference on Database and Expert Systems Applications (DEXA)*, pages 278–292, 2013.

[51] A. Boukorca, L. Bellatreche, S.-A. B. Senouci, and Z. Faget. Votre plan d'exécution de requêtes est un circuit intégré : Changer de métier. In *Actes des 9èmes journées francophones sur les Entrepôts de Données et l'Analyse en ligne*, pages 133–148, 2013.

[52] A. Boukorca, Z. Faget, and L. Bellatreche. What-if physical design for multiple query plan generation. In *Proceedings of International Conference on Database and Expert Systems Applications (DEXA)*, pages 492–506, 2014.

[53] J. Bowen. *Getting Started with Talend Open Studio for Data Integration*. Packt Publishing Ltd, 2012.

[54] A. Bretto and L. Gillibert. Hypergraph-based image representation. In *Graph-based Representations in Pattern Recognition*, pages 1–11. Springer, 2005.

[55] A. D. Brucker, I. Hang, G. Lückemeyer, and R. Ruparel. Securebpmn: modeling and enforcing access control requirements in business processes. In *ACM SACMAT*, pages 123–126, 2012.

[56] N. Bruno and S. Chaudhuri. Exploiting statistics on query expressions for optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 263–274. ACM, 2002.

[57] N. Bruno and S. Chaudhuri. Efficient creation of statistics over query expressions. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 201–212, 2003.

[58] T. Bui, C. Heigham, C. Jones, and T. Leighton. Improving the performance of the kernighan-lin and simulated annealing graph bisection algorithms. In *Proceedings of the 26th ACM/IEEE Design Automation Conference*, pages 775–778. ACM, 1989.

[59] L. Cabibbo and R. Torlone. Querying multidimensional databases. In *Database programming languages*, pages 319–335. Springer, 1998.

[60] A. E. Caldwell, A. B. Kahng, and I. L. Markov. Improved algorithms for hypergraph bipartitioning. In *Proceedings of Asia and South Pacific Design Automation Conference*, pages 661–666. ACM, 2000.

[61] A. E. Caldwell, A. B. Kahng, and I. L. Markov. Optimal partitioners and end-case placers for standard-cell layout. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(11):1304–1313, 2000.

[62] D. Calvanese, M. Lenzerini, and D. Nardi. Description logics for conceptual data modeling. In *Logics for Databases and Information Systems*, pages 229–263, 1998.

[63] R. L. Cannon, J. V. Dave, and J. C. Bezdek. Efficient implementation of the fuzzy c-means clustering algorithms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(2):248–255, 1986.

[64] W. Cao, F. Yu, and J. Xie. Realization of the low cost and high performance mysql cloud database. *International Journal on Very Large DataBases*, 7(13), 2014.

[65] Ü. V. Çatalyürek and C. Aykanat. Decomposing irregularly sparse matrices for parallel matrix-vector multiplication. In *Parallel Algorithms for Irregularly Structured Problems*, pages 75–86. Springer, 1996.

[66] U. V. Catalyürek and C. Aykanat. Patoh: a multilevel hypergraph partitioning tool, version 3.0. *Bilkent University, Department of Computer Engineering, Ankara*, 6533, 1999.

[67] S. Ceri, M. Negri, and G. Pelagatti. Horizontal data partitioning in database design. In *SIGMOD Conference*, pages 128–136, 1982.

[68] S. Chaudhuri, M. Datar, and V. Narasayya. Index selection for databases: A hardness study and a principled heuristic solution. *IEEE Transactions on Knowledge and Data Engineering*, 16(11):1313–1323, 2004.

[69] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *ACM Sigmod record*, 26(1):65–74, 1997.

[70] S. Chaudhuri and V. R. Narasayya. An efficient, cost-driven index selection tool for microsoft sql server. In *Proceedings of the International Conference on Very Large DataBases (VLDB)*, volume 97, pages 146–155, 1997.

[71] S. Chaudhuri and V. R. Narasayya. Autoadmin 'what-if' index analysis utility. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 367–378, 1998.

[72] S. Chaudhuri and V. R. Narasayya. Self-tuning database systems: A decade of progress. In *Proceedings of the International Conference on Very Large DataBases (VLDB)*, pages 3–14, 2007.

[73] L. W. F. Chaves, E. Buchmann, F. Hueske, and K. Böhm. Towards materialized view selection for distributed databases. In *Proceedings of International Conference on Extending Database Technology (EDBT)*, pages 1088–1099. ACM, 2009.

[74] C. M. Chen and N. Roussopoulos. The implementation and performance evaluation of the adms query optimizer: Integrating query result caching and matching. In *Proceedings of International Conference on Extending Database Technology (EDBT)*, volume 779, page 323. Springer Science & Business Media, 1994.

[75] P. P. S. Chen. The entity relationship model toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976.

[76] C.-H. Cheng, W.-K. Lee, and K.-F. Wong. A genetic algorithm-based clustering approach for database partitioning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 32(3):215–230, 2002.

[77] J. Cheng, D. Haderle, R. Hedges, B. R. Iyer, T. Messinger, C. Mohan, and Y. Wang. An efficient hybrid join algorithm: A db2 prototype. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 171–180, 1991.

[78] S. Cluet and G. Moerkotte. On the complexity of generating optimal left-deep processing trees with cross products. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 54–67. Springer, 1995.

[79] J. Cong, M. Romesis, and M. Xie. Optimality, scalability and stability study of partitioning and placement algorithms. In *Proceedings of the International Symposium on Physical Design*, pages 88–94. ACM, 2003.

[80] A. Cosar, E.-P. Lim, and J. Srivastava. Multiple query optimization with depth-first branch-and-bound and dynamic query ordering. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 433–438. ACM, 1993.

[81] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *International Journal on Very Large DataBases*, 3(1-2):48–57, 2010.

[82] A. Cuzzocrea, J. Darmont, and H. Mahboubi. Fragmenting very large xml data warehouses via k-means clustering algorithm. *International Journal of Business Intelligence and Data Mining*, 4(3):301–328, 2009.

[83] A. Cuzzocrea, J. Darmont, and H. Mahboubi. Fragmenting very large XML data warehouses via k-means clustering algorithm. *IJBIDM*, 4(3/4):301–328, 2009.

[84] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin. Automatic sql tuning in oracle 10g. In *Proceedings of the International Conference on Very Large DataBases (VLDB)*, pages 1098–1109. VLDB Endowment, 2004.

[85] N. N. Dalvi, S. K. Sanghai, P. Roy, and S. Sudarshan. Pipelining in multi-query optimization. *Journal of Computer and System Sciences*, 66(4):728–762, 2003.

[86] N. Daneshpour and A. A. Barforoush. Dynamic view management system for query prediction to view materialization. *International Journal of Data Warehousing and Mining (IJDWM)*, 7(2):67–96, 2011.

[87] A. Datta, B. Moon, and H. Thomas. A case for parallelism in data warehousing and olap. In *Proceedings of International Conference on Database and Expert Systems Applications (DEXA)*, pages 226–231. IEEE, 1998.

[88] R. de la Sablonnière, E. Auger, M. Sabourin, and G. Newton. Facilitating data sharing in the behavioural sciences. *Data Science Journal*, 11:DS29–DS43, 2012.

[89] M. F. de Souza and M. C. Sampaio. Efficient materialization and use of views in data warehouses. *SIGMOD Record*, 28(1):78–83, 1999.

[90] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[91] B. Debnath, S. Sengupta, and J. Li. Flashstore: high throughput persistent key-value store. *Proceedings of the International Conference on Very Large DataBases (VLDB)*, 3(1-2):1414–1425, 2010.

[92] C. Demetrescu and G. F. Italiano. Trade-offs for dynamic graph problems. In *Encyclopedia of Algorithms*, pages 958–961. Springer, 2008.

[93] R. Derakhshan, B. Stantic, O. Korn, and F. Dehne. Parallel simulated annealing for materialized view selection in data warehousing environments. In *Algorithms and Architectures for Parallel Processing*, pages 121–132. Springer, 2008.

[94] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *20th International Conference on Parallel and Distributed Processing Symposium, IPDPS'06*, pages 10–pp. IEEE, 2006.

[95] D. J. DeWitt, S. Madden, and M. Stonebraker. How to build a high-performance data warehouse, 2006.

[96] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Sketch-based multi-query processing over data streams. In *Data Stream Management*, pages 241–261. Springer, 2016.

[97] A. Ducournau, A. Bretto, S. Rital, and B. Laget. A reductive approach to hypergraph clustering: An application to image segmentation. *Pattern Recognition*, 45(7):2788–2803, 2012.

[98] T. Eavis and R. Sayeed. High performance analytics with the r3-cache. In *Proceedings of International Conference on Data Warehousing and Knowledge Discovery (DaWaK)*, pages 271–286, 2009.

[99] M. Eirinaki, S. Abraham, N. Polyzotis, and N. Shaikh. Querie: Collaborative database exploration. *IEEE Transactions on Knowledge and Data Engineering*, 26(7):1778–1790, 2014.

[100] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli. System level hardware/software partitioning based on simulated annealing and tabu search. *Design automation for embedded systems*, 2(1):5–32, 1997.

[101] R. Elmasri and S. B. Navathe. *Fundamentals of database systems.* Pearson, 2014.

[102] L. Etcheverry and A. A. Vaisman. Enhancing olap analysis with web cubes. In *The Semantic Web: Research and Applications*, pages 469–483. Springer, 2012.

[103] R. Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *Journal of the ACM (JACM)*, 30(3):514–550, 1983.

[104] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Conference on Design Automation*, pages 175–181. IEEE, 1982.

[105] S. Finkelstein. Common expression analysis in database applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 235–245. ACM, 1982.

[106] M. Flouris, R. Lachaize, and A. Bilas. Orchestra: Extensible block-level support for resource and data sharing in networked storage systems. In *14th International Conference on Parallel and Distributed Systems, ICPADS*, pages 237–244, 2008.

[107] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International journal of high performance computing applications*, 15(3):200–222, 2001.

[108] P. Furtado. Experimental evidence on partitioning in parallel data warehouses. In *Proceedings of the 7th ACM international workshop on Data warehousing and OLAP*, pages 23–30. ACM, 2004.

[109] J. Gentry. twitter: R based twitter client. *R package version 0.99*, 19, 2012.

[110] F. Goasdoué, K. Karanasos, J. Leblay, and I. Manolescu. View selection in semantic web databases. *Proceedings of the International Conference on Very Large DataBases (VLDB)*, 5(2):97–108, 2011.

[111] C. H. Goh. Context interchange: New features and formalisms for the intelligent integration of information. *ACM TOIS*, pages 270–293, 1999.

[112] M. Golfarelli, D. Maio, and S. Rizzi. Conceptual design of data warehouses from e/r schemes. In *Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, volume 7, pages 334–343. IEEE, 1998.

[113] M. Golfarelli, V. Maniezzo, and S. Rizzi. Materialization of fragmented views in multidimensional databases. *Data Knowl. Eng.*, 49(3):325–351, 2004.

[114] M. Golfarelli and S. Rizzi. *Data Warehouse Design: Modern Principles and Methodologies.* McGraw-Hill, Inc., 1 edition, 2009.

[115] N. Gorla and P. W. Y. Betty. vertical fragmentation in databases using data-mining. *Strategic Advancements in Utilizing Data Mining and Warehousing Technologies: New Concepts and Developments: New Concepts and Developments*, page 178, 2009.

[116] J. Grant and J. Minker. On optimizing the evaluation of a set of expressions. *International Journal of Computer & Information Sciences*, 11(3):179–191, 1982.

[117] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[118] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 25(2):173–182, 1996.

[119] H. Gupta. Selection and maintenance of views in a data warehouse. Ph.d. thesis, Stanford University, 1999.

[120] H. Gupta and I. S. Mumick. Selection of views to materialize under a maintenance cost constraint. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 453–470. Springer, 1999.

[121] H. Gupta and I. S. Mumick. Selection of views to materialize in a data warehouse. *IEEE Transactions on Knowledge and Data Engineering*, 17(1):24–43, 2005.

[122] L. W. Hagen, D. J. Huang, and A. B. Kahng. On implementation choices for iterative improvement partitioning algorithms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(10):1199–1205, 1997.

[123] E.-H. Han, G. Karypis, V. Kumar, and B. Mobasher. Clustering in a high-dimensional space using hypergraph models. *Proceedings of data mining and knowledge discovery*, 1997.

[124] N. Hanusse, S. Maabout, and R. Tofan. A view selection algorithm with performance guarantee. In *Proceedings of International Conference on Extending Database Technology (EDBT)*, pages 946–957. ACM, 2009.

[125] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. *ACM SIGMOD Record*, 25(2):205–216, 1996.

[126] E. P. Harris and K. Ramamohanarao. Join algorithm costs revisited. *International Journal on Very Large DataBases*, 5(1):064–084, 1996.

[127] S. Hauck and G. Borriello. An evaluation of bipartitioning techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(8):849–866, 1997.

[128] J. Hauglid, K. Nørvåg, and N. Ryeng. Dyfram: dynamic fragmentation and replica management in distributed database systems. *Distributed and Parallel Databases*, 28(2–3):157–185, 2010.

[129] C. Hébert, A. Bretto, and B. Crémilleux. A data mining formalization to improve hypergraph minimal transversal computation. *Fundamenta Informaticae*, 80(4):415–434, 2007.

[130] B. Hendrickson and R. Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM Journal on Scientific Computing*, 16(2):452–469, 1995.

[131] U. Herzog and J. Schlösser. Global optimization and parallelization of integrity constraint checks. In *Proceedings of International Conference on Management of Data (COMAD)*. Citeseer, 1995.

[132] P. Hitzler, M. Krotzsch, and S. Rudolph. *Foundations of semantic web technologies*. CRC Press, 2011.

[133] J. Hoffart, F. M. Suchanek, K. Berberich, E. Lewis-Kelham, G. de Melo, and G. Weikum. YAGO2: exploring and querying world knowledge in time, space, context, and many languages. In *WWW*, pages 229–232, 2011.

[134] J.-T. Horng, Y.-J. Chang, B.-J. Liu, and C.-Y. Kao. Materialized view selection using genetic algorithms in a data warehouse system. In *Proceedings of the Congress on Evolutionary Computation (CEC)*, volume 3. IEEE, 1999.

[135] S.-W. Hur. *Hybrid techniques for standard cell placement*. PhD thesis, University of Illinois at Chicago, 2000.

[136] S.-W. Hur and J. Lillis. Relaxation and clustering in a local search framework: application to linear placement. *VLSI Design*, 14(2):143–154, 2002.

[137] Y. E. Ioannidis. Query optimization. *ACM Comput. Surv.*, 28(1):121–123, 1996.

[138] Y. E. Ioannidis and Y. Kang. Randomized algorithms for optimizing large join queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 312–321. ACM, 1990.

[139] Y. E. Ioannidis and Y. C. Kang. Randomized algorithms for optimizing large join queries. In *ACM SIGMOD*, pages 312–321, 1990.

[140] M. J-N. and J. Trujillo. An mda approach for the development of data warehouses. In *JISBD*, pages 208–208, 2009.

[141] D. Jemal, R. Faiz, A. Boukorca, and L. Bellatreche. Mapreduce-dbms: An integration model for big data management and optimization. In *26th International Conference on Database and Expert Systems Applications (DEXA)*, pages 430–439, 2015.

[142] A. B. Kahng. Futures for partitioning in physical design. In *Proceedings of the International Symposium on Physical Design*, pages 190–193, 1998.

[143] A. Kalinin, U. Çetintemel, and S. B. Zdonik. Searchlight: Enabling integrated search and exploration over large multidimensional data. *PVLDB*, 8(10):1094–1105, 2015.

[144] P. Kalnis, N. Mamoulis, and D. Papadias. View selection using randomized search. *Data & Knowledge Engineering*, 42(1):89–111, 2002.

[145] R. Karimi, L. Bellatreche, P. Girard, A. Boukorca, and A. Hajdu. BINOS4DNA: bitmap indexes and nosql for identifying species with dna signatures through metagenomics samples. In *Proceedings of International conference Information on Technology in Bio- and Medical Informatics (ITBAM)*, pages 1–14, 2014.

[146] K. Karlapalem. *Redesign of distributed relational databases.* PhD thesis, Georgia Institute of Technology, 1992.

[147] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: applications in vlsi domain. In *34th Design and Automation Conference*, pages 526–529, 1997.

[148] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: applications in vlsi domain. *IEEE Transactions on Very Large Scale Integration Systems*, 7(1):69–79, 1999.

[149] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48(1):71–95, 1998.

[150] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48(1):71–95, 1998.

[151] G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning. In *ACM/IEEE Design Automation Conference (DAC)*, pages 343–348. ACM, 1999.

[152] G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning. *VLSI design*, 11(3):285–300, 2000.

[153] A. Kementsietsidis, F. Neven, D. V. de Craen, and S. Vansummeren. Scalable multi-query optimization for exploratory queries over federated scientific databases. *PVLDB*, 1(1):16–27, 2008.

[154] A. Kerkad. *L'interaction au service de l'optimisation à grande èchelle des entrepôts de données relationnels.* PhD thesis, ISAE-ENSMA, dec 2013.

[155] A. Kerkad, L. Bellatreche, and D. Geniet. Queen-bee: query interaction-aware for buffer allocation and scheduling problem. In *Proceedings of International Conference on Data Warehousing and Knowledge Discovery (DaWaK)*, pages 156–167, 2012.

[156] S. Khouri. *Cycle de vie sémantique de conception de systèmes de stockage et de manipulation de données*. PhD thesis, ISAE-ENSMA and ESI of Algeria, oct 2013.

[157] W. Kim. On optimizing an sql-like nested query. *ACM Transactions on Database Systems (TODS)*, 7(3):443–469, 1982.

[158] R. Kimball and M. Ross. *The data warehouse toolkit: the complete guide to dimensional modeling*. John Wiley & Sons, 2011.

[159] S. Klamt, U.-U. Haus, and F. Theis. Hypergraphs and cellular networks. *PLoS Comput Biol*, 5(5):e1000385, 2009.

[160] E. V. Konstantinova and V. A. Skorobogatov. Application of hypergraph theory in chemistry. *Discrete Mathematics*, 235(1):365–383, 2001.

[161] Y. Kotidis and N. Roussopoulos. Dynamat: a dynamic view management system for data warehouses. *ACM Sigmod record*, 28(2):371–382, 1999.

[162] W. Labio, D. Quass, and B. Adelberg. Physical database design for data warehouses. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 277–288, 1997.

[163] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The vertica analytic database: C-store 7 years later. *International Journal on Very Large DataBases*, 5(12):1790–1801, 2012.

[164] S. Lauesen. Task descriptions as functional requirements. *IEEE Software*, 20(2):58–65, Mar. 2003.

[165] M. Lawrence and A. Rau-Chaplin. Dynamic view selection for OLAP. In *8th International Conference on Data Warehousing and Knowledge Discovery*, pages 33–44, 2006.

[166] W. Le, A. Kementsietsidis, S. Duan, and F. Li. Scalable multi-query optimization for sparql. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 666–677. IEEE, 2012.

[167] C. Lee, C.-S. Shih, and Y.-H. Chen. Optimizing large join queries using a graph-based approach. *IEEE Transactions on Knowledge and Data Engineering*, 13(2):298–315, 2001.

[168] A. Lerner and D. Shasha. Aquery: Query language for ordered data, optimization techniques, and experiments. In *Proceedings of the International Conference on Very Large DataBases (VLDB)*, pages 345–356. VLDB Endowment, 2003.

[169] M. Levene and G. Loizou. Why is the snowflake schema a good data warehouse design? *Inf. Syst.*, 28(3):225–240, 2003.

[170] Q. Li, B. Moon, et al. Indexing and querying xml data for regular path expressions. In *Proceedings of the International Conference on Very Large DataBases (VLDB)*, volume 1, pages 361–370, 2001.

[171] A. A. B. Lima, C. Furtado, P. Valduriez, and M. Mattoso. Parallel OLAP query processing in database clusters with data replication. *Distributed and Parallel Databases*, 25(1-2):97–123, 2009.

[172] K.-I. Lin, H. V. Jagadish, and C. Faloutsos. The tv-tree: An index structure for high-dimensional data. *International Journal on Very Large DataBases*, 3(4):517–542, 1994.

[173] S. Luján-Mora, J. Trujillo, and I.-Y. Song. A uml profile for multidimensional modeling in data warehouses. *Data & Knowledge Engineering*, 59(3):725–769, 2006.

[174] S. Luján-Mora, P. Vassiliadis, and J. Trujillo. Data mapping diagrams for data warehouse design with uml. In *ER*, pages 191–204, 2004.

[175] H. Mahboubi and J. Darmont. Enhancing xml data warehouse query performance by fragmentation. In *Proceedings of ACM symposium on Applied Computing*, pages 1555–1562. ACM, 2009.

[176] C. Maier, D. Dash, I. Alagiannis, A. Ailamaki, and T. Heinis. PARINDA: an interactive physical designer for postgresql. In *Proceedings of International Conference on Extending Database Technology (EDBT)*, pages 701–704, 2010.

[177] I. Mami and Z. Bellahsene. A survey of view selection methods. *ACM SIGMOD Record*, 41(1):20–29, 2012.

[178] I. Mami, R. Coletta, and Z. Bellahsene. Modeling view selection as a constraint satisfaction problem. In *Proceedings of International Conference on Database and Expert Systems Applications (DEXA)*, pages 396–410. Springer, 2011.

[179] H. Märtens, E. Rahm, and T. Stöhr. Dynamic query scheduling in parallel data warehouses. *Concurrency and Computation: Practice and Experience*, 15(11-12):1169–1190, 2003.

[180] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *ACM SIGMOD Record*, volume 30, pages 307–318. ACM, 2001.

[181] G. Moerkotte, T. Neumann, and G. Steidl. Preventing bad plans by bounding the impact of cardinality estimation errors. *International Journal on Very Large DataBases*, 2(1):982–993, 2009.

[182] M. K. Mohania and N. L. Sarda. Some issues in design of distributed deductive databases. In *Proceedings of the International Conference on Very Large DataBases (VLDB)*, pages 60–71, 1994.

[183] E. Nakuçi, V. Theodorou, P. Jovanovic, and A. Abelló. Bijoux: Data generator for evaluating ETL process quality. In *ACM DOLAP*, pages 23–32, 2014.

[184] S. Navathe, K. Karlapalem, and M. Ra. A mixed fragmentation methodology for initial distributed database design. *IEEE Transactions on Software Engineering*, 3(4):395–426, 1995.

[185] S. B. Navathe, S. Ceri, G. Wiederhold, and J. Dou. Vertical partitioning algorithms for database design. *ACM Transactions on Database Systems (TODS)*, 9(4):680–710, 1984.

[186] S. B. Navathe and M. Ra. Vertical partitioning for database design: a graphical algorithm. *ACM SIGMOD Record*, 18(2):440–450, 1989.

[187] V. Nebot and R. Berlanga. Building data warehouses with semantic web data. *Decision Support Systems*, 52(4):853–868, 2012.

[188] R. Nehme and N. Bruno. Automated partitioning design in parallel database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1137–1148. ACM, 2011.

[189] A. Y. Noaman and K. Barker. A horizontal fragmentation algorithm for the fact relation in a distributed data warehouse. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)*, pages 154–161. ACM, 1999.

[190] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. Sharing across multiple mapreduce jobs. *ACM Trans. Database Syst.*, 39(2):12, 2014.

[191] K. O'Gorman, D. Agrawal, and A. El Abbadi. Multiple query optimization by cache-aware middleware using query teamwork. In *Proceedings of the International Conference on Data Engineering (ICDE)*, page 274, 2002.

[192] P. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *ACM SIGMOD Record*, 24(3):8–11, 1995.

[193] P. O'Neil, B. O'Neil, and X. Chen. Star schema benchmark, 2009.

[194] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak. The star schema benchmark and augmented fact table indexing. In R. Nambiar and M. Poess, editors, *Performance Evaluation and Benchmarking*, volume 5895 of *Lecture Notes in Computer Science*, pages 237–252. Springer Berlin Heidelberg, 2009.

[195] M. T. Özsu and P. Valduriez. *Principles of distributed database systems.* Springer Science & Business Media, 2011.

[196] D. A. Papa and I. L. Markov. Hypergraph partitioning and clustering. *Approximation algorithms and metaheuristics*, 61:1–19, 2007.

[197] S. Papadomanolakis and A. Ailamaki. Autopart: Automating schema design for large scientific databases using data partitioning. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 383–392, 2004.

[198] J. Park and A. Segev. Using common subexpressions to optimize multiple queries. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 311–319, 1988.

[199] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 61–72. ACM, 2012.

[200] T. B. Pedersen, C. S. Jensen, and C. E. Dyreson. A foundation for capturing and querying complex multidimensional data. *International Journal on Information Systems*, 26(5):383–423, 2001.

[201] A. Pellenkoft, C. A. Galindo-Legaria, and M. Kersten. The complexity of transformation-based join enumeration. In *Proceedings of the International Conference on Very Large DataBases (VLDB)*, pages 306–315, 1997.

[202] P. Peng, L. Zou, L. Chen, and D. Zhao. Query workload-based RDF graph fragmentation and allocation. In *Proceedings of the 19th International Conference on Extending Database Technology, EDBT*, pages 377–388, 2016.

[203] T. Phan and W.-S. Li. Dynamic materialization of query views for data warehouse workloads. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 436–445. IEEE, 2008.

[204] A. Pinar, Ü. V. Çatalyürek, C. Aykanat, and M. Pinar. Decomposing linear programs for parallel solution. In *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science*, pages 473–482. Springer, 1996.

[205] E. Pitoura. Query optimization. In *Encyclopedia of Database Systems*, pages 2272–2273. Springer US, 2009.

[206] P. Ponniah. *Data Warehousing Fundamentals for IT Professionals.* Wiley, 2010.

[207] R. Ramakrishnan and J. Gehrke. *Database Management Systems.* McGraw-Hill, Inc., 3 edition, 2003.

[208] J. Rao, C. Zhang, G. Lohman, and N. Megiddo. Automating physical database design in a parallel database. In *ACM SIGMOD*, pages 558–569, 2002.

[209] J. Rao, C. Zhang, N. Megiddo, and G. Lohman. Automating physical database design in a parallel database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 558–569. ACM, 2002.

[210] F. R. Reiss and T. Kanungo. A characterization of the sensitivity of query optimization to storage access cost parameters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 385–396. ACM, 2003.

[211] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *ACM SIGMOD Record*, volume 25, pages 447–458. ACM, 1996.

[212] A. Roukh, L. Bellatreche, A. Boukorca, and S. Bouarar. Eco-dmw: Eco-design methodology for data warehouses. In *ACM DOLAP*, pages 1–10, 2015.

[213] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 249–260. ACM, 2000.

[214] Y. G. Saab. An effective multilevel algorithm for bisecting graphs and hypergraphs. *IEEE Transactions on Computers*, 53(6):641–652, 2004.

[215] D. Saccà and G. Wiederhold. Database partitioning in a cluster of processors. In *Proceedings of the International Conference on Very Large DataBases (VLDB)*, pages 242–247, 1983.

[216] D. Sacca and G. Wiederhold. Database partitioning in a cluster of processors. *ACM Transactions on Database Systems (TODS)*, 10(1):29–56, 1985.

[217] D. Saccà and G. Wiederhold. Database partitioning in a cluster of processors. *ACM Transactions on Database Systems*, 10(1):29–56, 1985.

[218] C. Sapia, M. Blaschka, G. Höfling, and B. Dinter. Extending the e/r model for the multidimensional paradigm. In *Advances in Database Technologies*, pages 105–116. Springer, 1999.

[219] P. Scheuermann, J. Shim, and R. Vingralek. Watchman: A data warehouse intelligent cache manager. In *Proceedings of the International Conference on Very Large DataBases (VLDB)*, pages 51–62, 1996.

[220] W. Scheufele and G. Moerkotte. On the complexity of generating optimal plans with cross products (extended abstract). In *Proceedings of ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 238–248. ACM, 1997.

[221] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 23–34. ACM, 1979.

[222] T. Sellis and S. Ghosh. On the multiple query optimization problem. *IEEE Transactions on Knowledge and Data Engineering*, pages 262–266, 1990.

[223] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)*, 13(1):23–52, March 1988.

[224] S. Seshadri, V. Kumar, and B. F. Cooper. Optimizing multiple queries in distributed data stream systems. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 25–25. IEEE, 2006.

[225] K. C. Sevcik. Data base system performance prediction using an analytical model. In *Proceedings of the International Conference on Very Large DataBases (VLDB)*, pages 182–198. VLDB Endowment, 1981.

[226] M.-S. Shang, Z.-K. Zhang, T. Zhou, and Y.-C. Zhang. Collaborative filtering with diffusion-based similarity on tripartite graphs. *Physica A: Statistical Mechanics and its Applications*, 389(6):1259–1264, 2010.

[227] L. Shapiro, D. Maier, P. Benninghoff, K. Billings, Y. Fan, K. Hatwal, Q. Wang, Y. Zhang, H.-M. Wu, and B. Vance. Exploiting upper and lower bounds in top-down query optimization. In *International Symposium on Database Engineering and Applications*, pages 20–33. IEEE, 2001.

[228] K. Shim, T. Sellis, and D. Nau. Improvements on a heuristic algorithm for multiple-query optimization. *Data & Knowledge Engineering*, 12(2):197–222, 1994.

[229] O. Shmueli and S. Tsur. Logical diagnosis of ldl programs. *New Generation Computing*, 9(3/4):277–304, 1991.

[230] A. Shukla, P. Deshpande, J. F. Naughton, et al. Materialized view selection for multidimensional datasets. In *Proceedings of the International Conference on Very Large DataBases (VLDB)*, volume 98, pages 488–499, 1998.

[231] A. Simitsis. Mapping conceptual to logical models for etl processes. In *Proceedings of the 8th ACM international workshop on Data warehousing and OLAP*, pages 67–76. ACM, 2005.

[232] A. Simitsis, P. Vassiliadis, and T.-K. Sellis. Optimizing etl processes in data warehouses. In *ICDE*, pages 564–575, 2005.

[233] A. Simitsis, K. Wilkinson, U. Dayal, and M. Castellanos. Optimizing etl workflows for fault-tolerance. In *ICDE*, pages 385–396, 2010.

[234] D. Skoutas and A. Simitsis. Ontology-based conceptual design of etl processes for both structured and semi-structured data. *Int. J. Semantic Web Inf. Syst.*, 3(4):1–24, 2007.

[235] D. Skoutas and A. Simitsis. Ontology-based conceptual design of ETL processes for both structured and semi-structured data. *Int. J. Semantic Web Inf. Syst.*, 3(4):1–24, 2007.

[236] G. Slivinskas, C. S. Jensen, and R. T. Snodgrass. Bringing order to query optimization. *ACM SIGMOD Record*, 31(2):5–14, 2002.

[237] T. Stöhr, H. Märtens, and E. Rahm. Multi-dimensional database allocation for parallel data warehouses. In *Proceedings of the International Conference on Very Large DataBases (VLDB)*, pages 273–284, 2000.

[238] T. Stöhr and E. Rahm. Warlock: A data allocation tool for parallel warehouses. In *Proceedings of the International Conference on Very Large DataBases (VLDB)*, pages 721–722, 2001.

[239] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: a column-oriented dbms. In *Proceedings of the International Conference on Very Large DataBases (VLDB)*, pages 553–564. VLDB Endowment, 2005.

[240] S. N. Subramanian and S. Venkataraman. Cost-based optimization of decision support queries using transient-views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 319–330. ACM, 1998.

[241] D. Taniar and J. W. Rahayu. A taxonomy of indexing schemes for parallel database systems. *Distributed and Parallel Databases*, 12(1):73–106, 2002.

[242] D. Theodoratos, S. Ligoudistianos, and T. Sellis. View selection for designing the global data warehouse. *Data & Knowledge Engineering*, 39(3):219–240, 2001.

[243] D. Theodoratos and T. K. Sellis. Data warehouse configuration. In *Proceedings of the International Conference on Very Large DataBases (VLDB)*, pages 126–135, 1997.

[244] Y. Theoharis, V. Christophides, and G. Karvounarakis. Benchmarking database representations of rdf/s stores. In *The Semantic Web–ISWC 2005*, pages 685–701. Springer, 2005.

[245] D. Thomas, A. A. Diwan, and S. Sudarshan. Scheduling and caching in multiquery optimization. In *Proceedings of International Conference on Management of Data (COMAD)*, pages 150–153, 2006.

[246] I. H. Toroslu and A. Cosar. Dynamic programming solution for multiple query optimization problem. *Information Processing Letters*, 92(3):149–155, 2004.

[247] A. Tort, A. Olivé, and M. Sancho. An approach to test-driven development of conceptual schemas. *Data Knowl. Eng.*, 70(12):1088–1111, 2011.

[248] M. Toyonaga, S.-T. Yang, T. Akino, and I. Shirakawa. A new approach of fractal-dimension based module clustering for vlsi layout. In *IEEE International Symposium on Circuits and Systems. ISCAS'94*, volume 1, pages 185–188. IEEE, 1994.

[249] Q. T. Tran, I. Jimenez, R. Wang, N. Polyzotis, and A. Ailamaki. RITA: an index-tuning advisor for replicated databases. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management SSDBM*, pages 22:1–22:12, 2015.

[250] A. Trifunovic and W. J. Knottenbelt. Parkway 2.0: A parallel multilevel hypergraph partitioning tool. In *Computer and Information Sciences-ISCIS 2004*, pages 789–800. Springer, 2004.

[251] J. Trujillo and S. Luján-Mora. A uml based approach for modeling etl processes in data warehouses. In *ER*, pages 307–320, 2003.

[252] J. Trujillo, M. Palomar, J. Gomez, and I.-Y. Song. Designing data warehouses with oo conceptual models. *IEEE Computer*, 34(12):66–75, 2001.

[253] D. Tsirogiannis, S. Harizopoulos, and M. A. Shah. Analyzing the energy efficiency of a database server. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 231–242. ACM, 2010.

[254] P. Tziovara, P. Vassiliadis, and A. Simitsis. Deciding the physical implementation of etl workflows. In *DOLAP*, pages 49–56, 2007.

[255] K. Tzoumas, A. Deshpande, and C. S. Jensen. Efficiently adapting graphical models for selectivity estimation. *International Journal on Very Large DataBases*, 22(1):3–27, 2013.

[256] A. Vaisman and E. Zimányi. *Data Warehouse Systems: Design and Implementation.* Springer, 2014.

[257] G. Valentin, M. Zuliani, D. C. Zilio, G. Lohman, and A. Skelley. Db2 advisor: An optimizer smart enough to recommend its own indexes. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 101–101. IEEE Computer Society, 2000.

[258] P. Vassiliadis. A survey of extract–transform–load technology. *International Journal of Data Warehousing and Mining (IJDWM)*, 5(3):1–27, 2009.

[259] P. Vassiliadis, A. Simitsis, P. Georgantas, M. Terrovitis, and S. Skiadopoulos. A generic and customizable framework for the design of etl scenarios. *Inf. Syst.*, 30(7):492–525, 2005.

[260] P. Vassiliadis, A. Simitsis, and S. Skiadopoulos. Conceptual modeling for etl processes. In *DOLAP*, pages 14–21, 2002.

[261] P. Vassiliadis, A. Simitsis, and S. Skiadopoulos. Conceptual modeling for etl processes. In *Proceedings of the 5th ACM international workshop on Data Warehousing and OLAP*, pages 14–21. ACM, 2002.

[262] P. Vassiliadis, A. Simitsis, and S. Skiadopoulos. Modeling etl activities as graphs. In *DMDW*, pages 52–61, 2002.

[263] B. Vastenhouw and R. H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM review*, 47(1):67–95, 2005.

[264] A. Vazquez. Population stratification using a statistical model on hypergraphs. *Physical Review E*, 77(6):066–106, 2008.

[265] C. B. Walton, A. G. Dale, and R. M. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *Proceedings of the International Conference on Very Large DataBases (VLDB)*, volume 91, pages 537–548, 1991.

[266] G. Wang and C.-Y. Chan. Multi-query optimization in mapreduce framework. *International Journal on Very Large DataBases*, 7(3):145–156, 2013.

[267] P. Westerman. *Data Warehousing Using the Wal-Mart Model.* Morgan Kaufmann Series, 2001.

[268] K. Wilkinson, A. Simitsis, M. Castellanos, and U. Dayal. Leveraging business process models for etl design. In *ER*, pages 15–30, 2010.

[269] J. L. Wolf, B. R. Iyer, K. R. Pattipati, and J. Turek. Optimal buffer partitioning for the nested block join algorithm. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 510–519. IEEE, 1991.

[270] O. Wolfson and S. Jajodia. Distributed algorithms for dynamic replication of data. In *PODS*, pages 149–163, 1992.

[271] Z. Wu and R. Leahy. An optimal graph theoretic approach to data clustering: Theory and its application to image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(11):1101–1113, 1993.

[272] H. Yang and D. Wong. Efficient network flow based min-cut balanced partitioning. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 50–55, 1994.

[273] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. In *Proceedings of the International Conference on Very Large DataBases (VLDB)*, pages 136–145. Morgan Kaufmann Publishers Inc., 1997.

[274] J. Yang, K. Karlapalem, and Q. Li. A framework for designing materialized views in data warehousing environment. In *Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 458–465, 1997.

[275] J. X. Yu, X. Yao, C.-H. Choi, and G. Gou. Materialized view selection as constrained evolutionary optimization. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 33(4):458–467, 2003.

[276] A. Yzelman and R. H. Bisseling. Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods. *SIAM Journal on Scientific Computing*, 31(4):3128–3154, 2009.

[277] S. B. Zdonik and D. Maier, editors. *Readings in Object-Oriented Database Systems*. Morgan Kaufmann, 1990.

[278] C. Zhang and J. Yang. Genetic algorithm for materialized view selection in data warehouse environments. In *DataWarehousing and Knowledge Discovery*, pages 116–125. Springer, 1999.

[279] C. Zhang, X. Yao, and J. Yang. An evolutionary approach to materialized views selection in a data warehouse environment. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 31(3):282–294, 2001.

[280] Y. Zhao, P. M. Deshpande, J. F. Naughton, and A. Shukla. Simultaneous optimization and evaluation of multiple dimensional queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 271–282. ACM, 1998.

[281] D. C. Zilio, A. Jhingran, and S. Padmanabhan. *Partitioning key selection for a shared-nothing parallel database system.* IBM TJ Watson Research Center, 1994.

[282] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. Db2 design advisor: integrated automatic physical database design. In *Proceedings of the International Conference on Very Large DataBases (VLDB)*, pages 1087–1097. VLDB Endowment, 2004.

[283] V. Zlatić, G. Ghoshal, and G. Caldarelli. Hypergraph topological quantities for tagged social networks. *Physical Review E*, 80(3):036118, 2009.

[284] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Cooperative scans: dynamic bandwidth sharing in a dbms. In *Proceedings of the International Conference on Very Large DataBases (VLDB)*, pages 723–734. VLDB Endowment, 2007.

Résumé

L'apparition du phénomène *Big-Data*, a conduit à l'arrivée de nouvelles besoins croissants et urgents de partage de données qui a engendré un grand nombre de requêtes que les SGBD doivent gérer. Ce problème a été aggravé par d'autres besoins de recommandation et d'exploration des requêtes. Vu que le traitement de données est toujours possible grâce aux solutions liées à l'optimisation de requêtes, la conception physique et l'architecture de déploiement, où ces solutions sont des résultats de problèmes combinatoires basés sur les requêtes, il est indispensable de revoir les méthodes traditionnelles pour répondre aux nouvelles besoins de passage à l'échelle. Cette thèse s'intéresse à ce problème de nombreuses requêtes et propose une approche, implémentée par un Framework appelé *Big-Quereis*, qui passe à l'échelle et basée sur le hypergraph, une structure de données flexible, qui a une grande puissance de modélisation et permet des formulations précises de nombreux problèmes de combinatoire informatique. Cette approche est le fruit de collaboration avec l'entreprise *Mentor Graphics*. Elle vise à capturer l'interaction de requêtes dans un plan unifié de requêtes et utiliser des algorithmes de partitionnement pour assurer le passage à l'échelle et avoir des structures d'optimisation optimales (vues matérialisées et partitionnement de données). Ce plan unifié est utilisé dans la phase de déploiement des entrepôts de données parallèles, par le partitionnement de données en fragments et l'allocation de ces fragments dans les nœuds de calcule correspondants. Une étude expérimentale intensive a montré l'intérêt de notre approche en termes de passage à l'échelle des algorithmes et de réduction de temps de réponse de requêtes.

Mots-clefs : Conception physique, Entrepôt de données, Hypergraphe, Vues matérialisées, fragmentation de données.

Abstract

The emergence of the phenomenon Big-Data conducts to the introduction of new increased and urgent needs to share data between users and communities, which has engender a large number of queries that DBMS must handle. This problem has been compounded by other needs of recommendation and exploration of queries. Since data processing is still possible through solutions of query optimization, physical design and deployment architectures, in which these solutions are the results of combinatorial problems based on queries, it is essential to review traditional methods to respond to new needs of scalability.
This thesis focuses on the problem of numerous queries and proposes a scalable approach implemented on framework called *Big-queries* and based on the hypergraph, a flexible data structure, which has a larger modeling power and may allow accurate formulation of many problems of combinatorial scientific computing. This approach is the result of collaboration with the company Mentor Graphics. It aims to capture the queries interaction in an unified query plan and to use partitioning algorithms to ensure scalability and to optimal optimization structures (materialized views and data partitioning). Also, the unified plan is used in the deployment phase of parallel data warehouses, by allowing data partitioning in fragments and allocating these fragments in the correspond processing nodes. Intensive experimental study showed the interest of our approach in terms of scaling algorithms and minimization of query response time.

Keywords: Physical Design, Data Warehouse, Hypergraph, Materialized Views, Data Partitioning

---

Secteur de recherche : Informatique et applications

LABORATOIRE D'INFORMATIQUE ET D'AUTOMATIQUE POUR LES SYSTEMES
Ecole Nationale Supérieure de Mécanique et d'Aérotechnique
Téléport 2 – 1, avenue Clément Ader – BP 40109 – 86961 Chasseneuil-Futuroscope Cedex
Tél : 05.49.49.80.63 – Fax : 05.49.49.80.64