Learn based Optimisation of Distributed Queries

Lourdes Angélica Martínez-Medina

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 Août 2006

Présentée par

«Lourdes Angelica MARTINEZ MEDINA»

Thèse dirigée par **«Mme. Christine COLLET»**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
dans **l'École Doctorale de Mathématiques, Sciences et Technologies de l'Information et de l'informatique**

Optimisation des requêtes distribuées par apprentissage

Thèse soutenue publiquement le **«07-01-2014»**,
devant le jury composé de:

Mme. Esther PACITTI
Professeur à l'université à l'université Montpelier 2, Rapporteur

M. Bruno DEFUDE

Professeur à l'université TELECOM SudParis, Rapporteur

M. Andrzej DUDA

Professeur à l'Institut polytechnique de Grenoble, Examinateur

M. Stéphane GRUMBACH

Directeur de recherche INRIA, Examinateur

Mme. Christine COLLET

Professeur à l'Institut polytechnique de Grenoble, Directeur

M. Christophe BOBINEAU

Professeur à l'Institut polytechnique de Grenoble, Co-encadrant

Remerciements

Je tiens à remercier sincèrement à ma directrice de thèse Christine Collet pour m'avoir accueillie au sein de son équipe, pour avoir dirigé ce travail, pour son aide, ses encouragements et ses précieux conseils. Je remercie également à mon coordinateur Christophe Bobineau pour sa disponibilité, son écoute, ses commentaires, et sa collaboration dans la réalisation de ce travail. Egalement je remercie à ceux qui m'ont fait l'honneur d'être les membres de mon jury. Je remercie Ether Pacitti et Bruno Defude pour avoir évalué et reconnu mon travail, ainsi que Andrzej Duda et Stéphane Grumbach pour l'avoir examiné et jugé.

Je remercie vivement tous les membres de l'équipe Hadas pour leur soutien, les discussions, les conseils, la convivialité et la bonne ambiance qu'ils font régner. Je souhaite exprimer ma sincère gratitude à Genoveva Vargas-Solar et à Hector Manuel Trujillo-Ojeda pour m'avoir offert l'opportunité de venir en France. Je remercie l'ensemble de mes amis en France pour leur aide et leur soutien ainsi que pour tous les bons moments que nous avons passés ensemble. Je tiens à remercier profondément mon frère Juan Omar Martinez-Medina et mes parents Juan Omar Martinez-Lopez et Nora Angélica Medina-Del-Angel pour leur immense tendresse et leur soutien inconditionnel.

Enfin, un très grand merci à Gabriel Pedraza-Ferreira pour avoir supporté mes nombreuses angoisses, pour tout son amour, son soutien, et sa bonne humeur, ainsi que pour tous les moments que nous avons passés ensemble.

Grenoble, 09 Janvier 2014
Lourdes Angélica Martinez-Medina

Abstract

Distributed data systems are becoming increasingly complex. They interconnect devices (e.g. smartphones, tablets, etc.) that are heterogeneous, autonomous, either static or mobile, and with physical limitations. Such devices run applications (e.g. virtual games, social networks, etc.) for the online interaction of users producing / consuming data on demand or continuously. The characteristics of these systems add new dimensions to the query optimization problem, such as multi-optimization criteria, scarce information on data, lack of global system view, among others.

Traditional query optimization techniques focus on semi (or not at all) autonomous systems. They rely on information about data and make strong assumptions about the system behavior. Moreover, most of these techniques are centered on the optimization of execution time only. The difficulty for evaluating queries efficiently on nowadays applications motivates this work to revisit traditional query optimization techniques.

This thesis faces the previous challenges by adapting the Case Based Reasoning (CBR) paradigm to query optimization process. This adaptation, associated to a pseudo-random exploration of the search of solutions provides a way to optimize queries when there is no prior knowledge of data. This approach focuses on the optimization of queries using cases generated from the evaluation of similar past queries. A query case comprises: (i) the query (the problem), (ii) the query plan (the solution) and (iii) the measures of computational resources consumed during the query plan execution (the evaluation of the solution). This thesis also concerns the way the CBR process interacts with the query plan generation process, allowing the exploration of the space of solutions. This process uses classical query optimization heuristics and makes decisions randomly when information on data is not available (e.g. for ordering joins, selecting algorithms or choosing message exchange protocols). This process also exploits the CBR principle for generating plans for subqueries, thus accelerating the learning of new cases. The propositions of this thesis have been validated with the CoBRa optimizer developed in the context of the UBIQUEST project[1].

[1] The CoBRa optimizer was developed in the context of the UBIQUEST ANR-09-BLAN-0131-01 project.

Résumé

Les systèmes de gestion de données distribuées deviennent de plus en plus complexes. Ils interagissent avec des réseaux de dispositifs fixes et/ou mobiles, tels que des smartphones ou des tablettes, dispositifs hétérogènes, autonomes et possédant des limitations physiques. Ces dispositifs exécutent des applications permettant l'interaction des usagers (i.e. jeux virtuels, réseaux sociaux). Ces applications produisent et consomment des données à tout moment voire même en continu. Les caractéristiques de ces systèmes ajoutent des dimensions au problème de l'optimisation de requêtes, telles que la variabilité des objectifs d'optimisation, l'absence d'information sur les données (métadonnées) ou le manque d'une vision globale du système.

Les techniques traditionnelles d'optimisation des requêtes n'abordent pas (ou très peu) les systèmes autonomes. Elles se basent sur les métadonnées et font des hypothèses très fortes sur le comportement du système. En plus, la majorité de ces techniques d'optimisation ciblent uniquement l'optimisation du temps d'exécution. La difficulté d'évaluation des requêtes dans les applications modernes incite à revisiter les techniques traditionnelles d'optimisation.

Cette thèse fait face aux défis décrits précédemment par l'adaptation du paradigme du Raisonnement à partir de cas (CBR pour *Case-Based Reasoning*) au problème de l'optimisation des requêtes. Cette adaptation, associée à une exploration pseudo-aléatoire de l'espace de solutions fournit un moyen pour optimiser des requêtes dans les contextes possédant très peu voire aucune information sur les données. Cette approche se concentre sur l'optimisation de requêtes en utilisant les cas générés précédemment dans l'évaluation de requêtes similaires. Un cas de requête et composé par : (i) la requête (le problème), (ii) le plan d'exécution (la solution) et (iii) les mesures de ressources utilisés par l'exécution du plan (l'évaluation de la solution). Cette thèse aborde également la façon que le processus CBR interagit avec le processus de génération de plan d'exécution de la requête qui doit permettre d'explorer l'espace des solutions. Ce processus utilise les heuristiques classiques et prennent des décisions de façon aléatoire lorsque les métadonnées viennent à manquer (e.g. pour l'ordre des jointures, la sélection des algorithmes, voire même le choix des protocoles d'acheminement de messages). Ce processus exploite également le CBR pour générer des plans pour des sous-requêtes, accélérant ainsi l'apprentissage de nouveaux cas. Les propositions de cette thèse ont été validées à l'aide du prototype CoBRA développé dans le contexte du projet UBIQUEST.[1].

Table of content

List of tables

List of figures

1. INTRODUCTION

This chapter introduces the characteristics of nowadays distributed data systems and their impact on distributed query optimization. It recalls the basics of query processing for better understanding the responsibilities of the query optimization process. For motivating this work, it discusses the importance of efficient data querying for some applications (e.g. user interaction by data sharing), and the necessity to revisit traditional query optimization techniques to improve the functionality of applications development on complex distributed data systems. It presents the objective of this thesis, and our proposed approach for achieving such goal. Finally, it specifies the contributions of our work and the order in which they are presented in the remainder of this document.

1.1 CONTEXT AND MOTIVATION

The rapid evolution of information technologies drives us to an increasingly digitalized world, where applications support users cooperation producing/consuming data on demand or continuously. Distributed data systems are currently the technology for modeling, storing, managing and efficiently querying these large amounts of data. Nowadays distributed data systems are becoming increasingly complex. They face difficult challenges in the attempt to handle data of applications deployed on large number of distributed and heterogeneous computing devices (i.e. system nodes); which are also autonomous, either static or mobile, and that present physical limitations (e.g. energy, memory and computing capability). The reliability of distributed data systems depends critically on querying data efficiently.

Distributed data systems enable users accessing and managing data transparently (i.e. separation of the higher-level semantics of a system from lower-level implementation issues) using a declarative query language (e.g. SQL [ElSh11] , OQL [Banc89], XQuery [Wadl03]) that the system traduces into the implementation of a plan for query execution. Given a declarative query there are several alternative execution plans for producing the query results. However, even if these plans produce the same final output, they may drastically differ in their execution time and resources consumption. The query optimization process selects the query execution plan that minimizes plans cost according to an optimization objective (e.g. time). The complexity of distributed data systems add new dimensions to the query optimization problem, such as multi-optimization criteria, unpredictable

system behavior, scarce information on data, lack of a global view of the system, little (or any) control over the execution of queries, among others.

The literature has proposed seminal heuristics-based and cost-based optimization techniques. The former relies on data information (e.g. data distribution, data value statistics, etc.) for estimating plans cost, which due to the environment characteristics there is no guarantee to be available. This lack of information leads to heuristic-based approaches, which comply with avoiding the worst plans, but do not achieve to find the optimal one. Optimization techniques based on query feedback also have been suggested to overcome both, the lack of information and query plan improvement. They have been applied on distributed data systems that trend to centralize the optimization process, and where its components lack of autonomy. Moreover, they still rely on the usage of data information difficult to count on. Most of these techniques focus on the optimization of execution time only.

The difficulty for evaluating queries efficiently on nowadays applications motivates this work to revisit traditional query optimization techniques. This section presents the characteristics of distributed data systems, the query processing but particularly query optimization to analyze the challenges that must be faced. Finally, it addresses the particular query optimization problems that we tackle and that motivate this research.

1.1.1 Distributed data systems

A distributed data system comprises a number of autonomous and heterogeneous processing elements interconnected by a computer network that cooperate to perform some assigned tasks. The processing element referred to in this definition is a computing device that can execute a program on its own [OzVa11]. This section presents characteristics of distributed data systems, and highlights how such characteristics potentially difficult the optimization of queries. We focus on three main characteristics: (i) distribution, (ii) autonomy, and (iii) heterogeneity.

The distribution dimensions deals with data; thus it is considered the distribution of data among multiple nodes (e.g. some few specific sites or all nodes of the system). Autonomy refers to the distribution of control among system nodes for executing a query. An autonomous node may join or leave the system at any time without restrictions, and can decide about the availability of its resources (e.g. data). Heterogeneity concerns, from a logical point of view, to the representation of data, and from a physical point of view, to the characteristics of computing devices and network technologies. There are variations in the distribution, autonomy and heterogeneity of system components. Figure 1.1 positions representative distributed data systems, such as client-server [Scou95], multi-database [Ali09], and peer-to-peer [AkPV07], according to the previous characteristics.

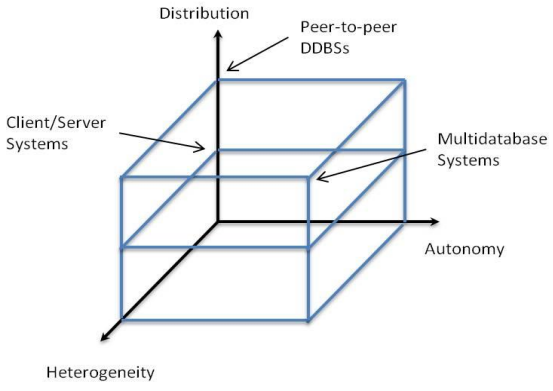

Figure 1.1 Characterization of distributed data systems [OzVa11]

During this time many innovations and extensions have been proposed to enhance distributed data systems in power, and spectrum of applications. For example cloud computing and grid technologies [TBBC07][WLGP08][JuXi09]. In its simplest form, a distributed data system comprises a single server that is accessed by several clients. In the client-server interaction, the client passes declarative queries to the server; the server does the most of work (i.e. data storage, query processing and optimization) and returns the result to the client. The client provides the user with an application interface; it also manages some specific data that is catch to the client. Thus, data is distributed in some few machines; the server has a global control of query execution. This approach has proved effective for applications that can benefit from centralized control and full-fledge data handle capabilities. However, it is not effective for large number of users.

Multi-database systems and parallel database systems have extended the previous approach with the objective to manage huge amounts of data and to support much more client requests. They proposed, in some or another way, to distribute data management among several machines to enable parallel execution of queries: (i) Inter-query parallelism for executing of several queries at a time, and (ii) Intra-query parallelism for executing several operators within a query at a time. The queries and operators are executed by many processors/machines each one working on a subset of data. System components may present different capabilities and behavior; also data and query languages may be heterogeneous, which represent an additional difficulty for query optimization. However, such systems still rely on information of data and made strong assumptions about the availability of resources (e.g. data, communication and processing).

In contrast P2P systems adopt a completely decentralized approach to data sharing by distributing data storage and processing across autonomous system nodes. Many domain specific P2P system have already been deployed [DKNW04], e.g. Gnutella, Kazaa, Napster [GBFR03], among others. However these P2P systems support simple functionality (e.g. file sharing) and the execution of straightforward queries (e.g. keyword search). In this systems, a lot of effort has been put into refining topologies and query routing functionalities e.g. CAN [RFHK01] and CHORD [SMKK01]; extending the query functionalities offered by such systems has been left little aside.

Query optimization takes a central stage in data systems, nowadays environment characteristics make the task of enabling efficient querying even more difficult. We can appreciate that the complexity of queries to be treated represents another complications for the query optimization issues. Next section focuses on query processing steps to expose the process that are in tour to query optimization.

1.1.2 Distributed query processing

A user expresses a query (i.e. global query) in a high-level language and in terms of virtual global data sources. A global source is physically distributed across different sites by fragmenting and replicating the data. The distributed query processing traduces a global query into an equivalent lower-level query implementing an execution strategy.

Processing a global query involves accessing data from several distributed and heterogeneous sources, as well as a complex process of data computation and exchange. Figure 1.2 shows a generic layering schema for distributed query processing scheme, it comprises four layers [OzVa11]: (i) query decomposition, (ii) data localization, (iii) global optimization, and (iv) distributed execution. Typically, this process is achieved in two phases, a compilation phases that includes from queries decomposition till code-generation steps, and an execution phase.

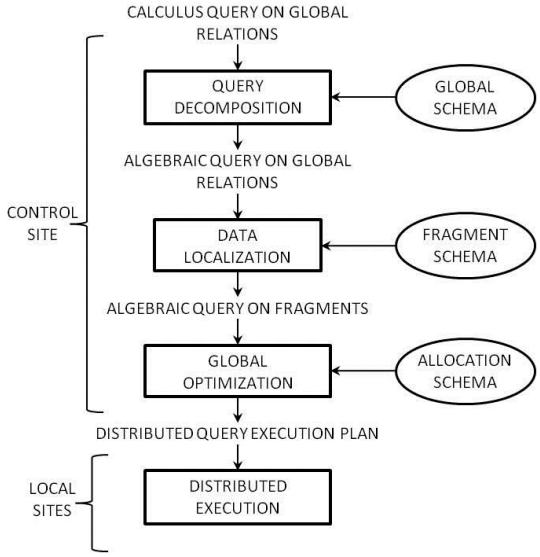

Figure 1.2. Generic layering schema for distributed query processing [OzVa11]

Query decomposition

The query decomposition process transforms a global query into an *internal representation* of the system, but still on global sources. The global query is expressed in a high-level declarative language, as mentioned before. The internal system representation typically corresponds to a query expression based in some algebra (e.g. relational algebra). Thus, the resulting query is represented as an algebraic tree where nodes correspond to operators for access and computation (e.g. filtering, projection, join, etc.) of data; the edges represent the relations between operators. The communication operations are not included yet since the query considers global sources.

Data localization

The data localization process takes as input the decomposed query on global relations and applies data distribution information to the query in order to localize the concerning data. So far in this chapter we elucidate that sources of data are actually split in subsets of data stored at different nodes of, each of these subsets is called *fragment*. Therefore, data localization determines the *fragments* of data that a global query includes, and transforms such query into a query on fragments.

Global optimization

The global optimization process aims to find an efficient plan for the distributed execution of a given query. The *query plan* comprises optimization decisions such as: the operations for executing the query, the order of such operators, the algorithms for executing the operations, and the system node where each operation is to be computed.

Different decisions may lead to many equivalent query plans in the sense that all of them achieve the same query result; however such plans present different performance. The query optimization process enumerates such alternative plans and estimates their cost by applying a cost function. Typically, such cost function estimates the execution time. The (close to) optimal query plan is that one that minimizes the cost function; finding the optimal plan is computationally intractable

[IbKa84]. Important inputs for estimating the cost function are statistics about data fragments and formulas for estimating the cardinalities of intermediate operation results.

Distributed execution

For executing a query, the optimal query plan is transformed in executable code that is computed in distributed way at several system nodes (e.g. the nodes having fragments involved in the query, all the nodes). Distributed execution techniques are integrated to query plans for improving queries execution. Such techniques mostly concern the exchange of queries/data trough the network; for example query shipping, data shipping and hybrid shipping [Bowm01]. Also they concern the computation of joining data stored at different nodes; semi-join [BGWR81], hash-join [WiAp93], horizontal join [EpSW78] and pointer-based joins [EiGK95] are some representative examples. Naturally, such techniques ought to be considered in the query optimization decisions to use them favorably.

1.1.3 Motivation example

This section presents two examples of social applications to highlight the motivation of this work. Both examples concern the problem of estimating plans cost due to the lack of information required by traditional query optimization techniques. The first one evokes data sharing e-science applications, which involve intensive data processing on highly distributed environments. The second one concerns the deployment of social networking application using a high-level declarative approach; applications functionality relies on efficient distributed query processing.

Example 1

E-science can be understood as the application of computing technology to the undertaking of modern scientific investigation [Tayl00]. It concerns large scale science that is carried out through scientists' collaboration in highly distributed network environments. Such applications involve sharing and processing massive collections of data in different formats. Moreover scientific also share their own programs for specialized data processing.

The CARMEN project is an interesting example of e-science application [WLGP08]. This work designed a system to allow neuroscientists to share, integrate and analyze brain data. Such a data have multiple formats and are expensive to collect; thus they are rarely shared. Moreover the analysis tools built at each research laboratory have their own specifications.

Thus, accessing and querying data in such distributed data system is quite challenging because of the nature, amount and distribution of data, scale of the network and the autonomy of resources. Most optimization techniques statically exploit data and network information; they do no longer accomplish the efficient evaluation of queries in this kind of complex environments. Moreover data processing includes a wide range of programs and services, with unknown properties and behavior. Most of highly distributed data systems support applications with simple well known functionalities that typically include the basic data management operators (e.g. algebraic operators [ElSh11]) [VaPa05].

Example 2

In modern networking environments, more and more applications rely on interaction between users that know each other, more precisely between the data stored on the devices the users own. Such social

links build wide logical networks of devices on which many applications are deployed. A way to facilitate the development of such applications is using a high-level declarative approach [ABCD12a]. For example, imagine a developer that wants develop virtual world game where user have own avatars that can interact if they are located in the same area. Avatars position in the virtual world can be represented as a single distributed table and the avatar actions can be expressed as queries and updates.

Data of the application is stored on remote servers and on user devices. Maintaining a global view of the participant devices is a difficult task due to their autonomy to (dis)connection at any time. Maintaining a global view of data distribution is even more difficult because devices (or users) may not want to (or cannot) share metadata and statistics, e.g. for heterogeneity reason. Classical distributed query evaluation techniques necessitate in-deep knowledge on the whole distributed system (network and data). This lack of information on data, essential for traditional query optimization techniques is the main motivation of our research.

1.2 OBJECTIVE

The general objective of this thesis is to provide a technique for optimizing global queries with incomplete information on data/system, and according to multiple-customizable optimization objectives. The distribution of data and the autonomy of devices hinder the availability of information essential for typical query optimization techniques. Applications and devices have different performance requirements (e.g. minimize response time, save energy consumption), this fact demands multiple and customizable optimization objectives. The general objective of this thesis comprises the following sub-objectives:

- Provide a query optimization technique independent of the information on data used by traditional query optimization techniques. Also, such technique must go one step further than heuristic-based query optimization techniques, which only avoid the worst query plans. The aim is to determine close to optimal query execution plans.

- Provide a query optimization technique that facilitates the customization of optimization objectives according to the needs of applications and the physical limitations of devices. Optimizing queries usually concerns the minimization of execution time. Although, nowadays applications may have different priorities, these may include fees to access information, quality of the data, number of sources to access, etc.; computing devices may present different physical limitations (e.g. sensors have restrain energy consumption).

- Provide a query optimization technique that progressively evolves with eventual changes of the environment. Availability and behavior of system components can change with most or less frequency over the time. The query optimization should avoid the deterioration of plans performance due to these variations. We consider environments where the frequency of changes allows correcting some optimization decisions to improve further query processing.

1.3 APPROACH

In this thesis we propose an approach for optimizing global queries based on feedback gathered from the execution of past queries. Such feedback is exploited to learn suitable query execution plans according to different (customizable) objectives. The optimization of queries is a complicated problem when complete information on data/system is not always available. We rely in the use of this query

feedback to face such problem that, as far we know, has not been fully addressed yet, particularly in distributed data systems with the characteristics that we exposed in the context of this work (Section 1.1).

Our optimization approach integrates the learning Case Based Reasoning (CBR) paradigm in the query processing [MCBD12]. CBR is a learning principle for problem solving within the artificial intelligence (AI) domain [LMBL05]. It proposes the exploitation of knowledge from solved past problems to improve further problems solving. The unit of knowledge is named *case*; the cases are stored in a repository called *casebase*. There is an expert in charge to feed the casebase with new cases and to evaluate the efficiency of proposed solutions.

Much of the inspiration for the study of CBR came from cognitive science research on human memory, thus reflecting human use of remembered problems and solutions to improve further problems solving. Just as CBR provides a way for people to generate solutions; it also provides a way for a computer program to propose solutions efficiently when previous similar problem solutions have been encountered. It has been shown that such principle is useful for problem solving when knowledge is incomplete and or evidence is sparse [Kolo92].

Our approach focuses on optimizing queries using cases generated from the evaluation of similar past queries; where a problem corresponds to a query, the problem solution corresponds to a query plan. A case also comprises global measures of time and computational resources (e.g. memory, energy, CPU) consumed during the execution of query plans. A global measure includes the total of computational resources consumed by all system nodes that participate in the evaluation of a query.

The learning-based optimization process concerns the way the CBR reasoning process interacts with the query plan generation process. Such process uses classical heuristics and makes decisions randomly (e.g. when statistics on data are not available). It also (re)uses cases (existing query plans) in the generation of plans for similar queries.

The general principle of our learning-based query optimization approach was presented in [MCBD12]. The CoBRa optimizer was developed in the context UBIQUEST project[2] [ABCD12b]; it allowed carrying out the experimental phase for validating our approach. This proposal remains prospective to some degree due to the need of incorporating additional aspects out of the scope of our work. These include new cost models independent of metadata, an inspired example is in [ShKM00]. Also, the definition of alternative query similarity functions.

1.4 CONTRIBUTIONS

We summarize the contributions of this thesis as follow:

- We adapt the Case Based Reasoning (CBR) paradigm to query processing, providing a way to optimize queries when there is no prior knowledge of data. Every query execution produces case(s) that includes the internal representation of the query, its plan, and measures of computational resources consumed during query plan execution. The optimization process then uses such cases to generate optimal execution plans for queries. Several steps are involved: (i) Query case retrieving, (ii) Query plan adaptation or generation, (iii) Query plan execution and monitoring and (iv) Query case storage.

[2] The CoBRa optimizer was developed in the context of the UBIQUEST ANR-09-BLAN-0131-01 project.

- We revisit the query plan generation process. This process uses classical heuristics and makes decisions randomly (e.g. when there is no statistics for join ordering and selection of algorithms, routing protocols). The query plan generation process also exploits the case base (existing query plans) for generating query plan parts, improving the query optimization and evaluation efficiency.

- We implement the CoBRa optimizer for validating our approach. Such optimizer prototype was implemented in the context of the ANR UBIQUEST project. The UBIQUEST approach provides a high level programming abstraction for developing networking applications [ABCD12b][ABCD12a]. It abstracts the network as a large distributed database that gives a unified view of "objects" handled by both networks and applications. The applications interact through declarative queries. Thus, such queries are posed in environments consisting of applications, servers and devices which can be heterogeneous, dynamic and autonomous. These elements are distributed in different locations and present physical limitations such as processing and storage capacity or energy consumption. The elements interact between them producing/consuming data on demand or continuously. In such environments there is no prior knowledge on data (sources) and certainly no related metadata such as data statistics. An instance of CoBRa was embedded at each node in network system for optimizing the evaluation of queries.

1.5 DOCUMENT ORGANIZATION

The remainder of this document is organized as follows:

- **Chapter 2** recalls the principle and representative techniques of distributed query optimization. It focuses on the study of query optimization approaches basing on query feedback for addressing the problem of incomplete information on data.

- **Chapter 3** introduces our learning-based distributed query optimization approach. It specifies the characteristics of data, queries, and the query language that we consider. It presents an overview of the Case-Based Reasoning principle, and concentrates on the adaptation of each of its components to our query optimization process.

- **Chapter 4** exposes first our representation of query case; afterward, our prospective definition of query similarity is described. This chapter also includes the organization of query cases within the casebase. Finally, it presents the mechanisms for retrieving and managing (i.e. insert, delete and update operations on the casebase) query case.

- **Chapter 5** is consecrated to present the query plan generation process and its interaction with the CBR paradigm: (i) the generation of query plans when no useful query cases exist within the casebase for solving a query, (ii) the exploitation of query cases for generating a query plan, and finally, (iii) an improvement strategy to avoid the overload and minimize the consumption of computing resources during this process.

- **Chapter 6** presents the design and implementation of the CoBRa optimizer. It comprises an overview of the optimizer architecture, followed by the specification of data and storage

structures corresponding to cases and the casebase. Then, it exposes the implementation details of our query optimization process. Finally, it explains the validation of our approach in the context of the UBIQUEST project and presents our experimental results.

- **Chapter 7** concludes the document. It summarizes the developed work, and discuses the challenges that remain open for extending our work.

2. QUERY OPTIMIZATION IN DISTRIBUTED DATA SYSTEMS

Query optimization has received considerable attention in the context of both, centralized and distributed data systems. However, the query optimization problem is considerably more difficult in distributed environments due to the larger number of aspects that affect the evaluation of queries; in particular, because of the inherent necessity to move data from one place to another. The reliability of such systems depends critically on the efficient querying of data.

The optimization process aims to find efficient strategies for the distributed execution of queries. The physical distribution of data in the system, the distribution of control for data access and query processing, and the heterogeneity in computing devices and applications requirements constitute the most important aspects that affect the distributed query optimization.

This chapter evokes query optimization foundations and focuses on relevant distributed query optimization techniques. Such techniques must answer some questions such as how to generate efficient query execution strategies and select the appropriate moment and location to carry out the optimization process. Our study is centered structured data querying (e.g. relational or object-oriented databases), and queries expressed on declarative languages as SQL-like and OQL-like. Nevertheless, some optimization approaches presented in this chapter can also be applied to other data models (e.g. semi-structured or unstructured).

This study highlights the importance of information on data (i.e. metadata) for traditional query optimization techniques. It reviews the existent proposals to deal with the challenge of query optimization with incomplete metadata. These works are mostly bases on query feedback (i.e. supplementary information of data gathered during the execution of queries). This chapter compiles different proposals of query optimization based on query feedback, and discuses about their advantages and their drawbacks for accomplishing the problems that motivate this dissertation.

The remainder of this chapter is structured as follows: Section 2.1 presents the basic principles of distributed query optimization. Section 2.2 reviews the query optimization techniques concerning: (i) the selection of efficient query evaluation strategies, (ii) the period of time for optimizing a query, and (iii) the element(s) in the system responsible to carry out the optimization

process. Section 2.3 studies in depth the traditional metadata used by mostly of classical optimization techniques. Section 2.4 focuses on relevant works that in some way or another have been based on query feedback to tackle the optimization of queries with incomplete (or unreliable) information on data. Section 2.5 concludes this chapter.

2.1 BASIC PRINCIPLES

The distributed query optimization aims to find efficient strategies, called *query plans*, for the distributed execution of queries. A *query plan* is a compilation of query optimization decisions that include: (i) the operators for executing the query, (ii) the ordering for executing such operators, (iii) the execution algorithms for computing each operator, (iv) the operators scheduling for their execution, and (v) the elements (i.e. nodes) of the system where each operation is to be computed.

A query plan is typically represented as a tree. The nodes of a plan are operators and every operator carries out one particular operation (e.g. join, sort, scan, etc.). The edges of a plan represent consumer-producer relationships of operators.

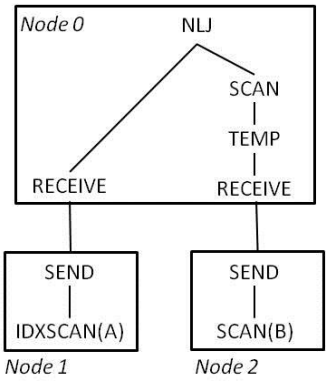

Figure 2.1 Example of a query plan

Figure 2.1 shows an example of plan for a query that involves sources A and B. The plan specifies that source A is accessed at the node N_1 using the index-scan (A) algorithm (uses a data index), B is accessed at node N_2 using the scan (B) algorithm (without index). The sources A and B are shipped to N_3 where are joined using a nested-loop-join algorithm. The send and receive operators encapsulate all the communication activity so that all other operators (e.g. nested-loop-join or scan) can be implemented and used in the same way as in a centralized data system.

Query optimization includes three components [Ioan96]: a search space, a cost model, and a search strategy. The search space is the set of alternative execution plans for evaluating a query. Different decisions may lead to many equivalent query plans, in the sense that all of them achieve the same query result, but have different performance. The search space is generated by applying transformation rules (i.e. operation equivalences in such a way that the query result is not altered). For instance, permuting the order of operations (e.g. joins and unions) for combining data fragments within a query may vary the performance of the query plan in orders of magnitude.

The query optimization process enumerates alternative query plans and estimates their cost by applying a cost function. The enumeration algorithm, well known as search strategy, defines which plans are examined and in which order. The search strategy explores the search space and selects the plan that optimizes the cost function. Typically, such cost function estimates the execution time. Important inputs for estimating the cost function are statistics about data fragments and formulas for

estimating the cardinalities of intermediate operation results. The characteristics of the environment (e.g. centralized versus distributed) are captured by the search space and the cost model. Next subsections briefly describe these three query optimization components.

2.1.1 Search space

The search space, or solution space, is the set of all equivalent query plans that the system considers for evaluating a given query. Such query plans correspond to different query evaluation strategies that naturally achieve the same result but usually present significant variations in their execution cost. A point within the search space corresponds to a specific query plan.

The search space is divided in the logical space and the physical space. The logical space query plans composed of logical algebraic operators. The physical space comprises plans composed of physical operators, this is, an execution technique is assigned to each logical operator. For a complex query, the number of equivalent query plans can be very high. For instance, the number of alternative query plans that can be produced for joining n sources is O(n!), this because of the commutative and associative properties of the join operator [StMK97].

The search space importantly increases when considering query plans for evaluating global queries. First, because the global query is expressed in terms of sources, when actually several source fragments are involved. A global source can be materialized by joining and/or unifying its fragments; this includes binary operator (i.e. join and union), both of them with associative and commutative properties. Thus a query that joins n global sources becomes in a query that combines a larger number of source fragments. Moreover, other distributed execution techniques (e.g. semi-join) and execution sites must be considered. Efficient strategies for searching the space of alternative query plans are required.

2.1.2 Search strategy

The search strategy, or enumeration algorithm, refers the way to explore the search space for finding the (close to) optimal query plan. The search space is explored by generating query plans according to the search space restrictions (e.g. heuristics and shape), in different order, and by applying different construction strategies. The cost of such plans is estimated by applying a cost function, the optimal plan is that one that minimizes the function result. In most of cases, search strategies focus on the generation of join plan trees (i.e. ordering and implementation), since join is the most frequent and costly operation.

Searching a very large space may turn the query optimization process prohibitive in terms of resources consumption, sometimes much more than the resources consumption during query execution. Therefore, search strategies restrict the size of the search space by applying heuristics (e.g. selection and projection first) upon the order of certain plan operators, and constraining the query plan shapes.

For constraining the shape of plans is useful to concentrate on *join tree* plans. A join tree is a binary tree that comprises sources of data as lefts and joins as inner nodes. Linear and bushy trees are two kinds of distinguished join trees [AlAO05]. A linear tree comprises joins where at least one operand of each join is a source of data. A bushy tree may have operators with no base relations as operands. By considering only linear trees, the size of the search space is reduced to O(2N) [Chau98]. However, in a distributed environment, bushy trees are useful in exhibiting parallelism for executing a

query. Other shape of query plan called zigzag tree are mainly of interest in distributed and parallel database systems.The query plan generation techniques for searching the space of alternative plans are presented in Section 2.2.2.

2.1.3 Distributed cost model

The selection of the (close to) optimal query plan requires the estimation of the execution cost of the alternative candidate plans. A cost model specifies arithmetic formulas, so-called *cost functions*, for estimating the cost of executing such plans. A cost function is often defined in terms of time units, and is calculated taking into consideration the resources consumed during the plan execution. In a distributed environment, such execution cost is typically defined as a weighted combination of I/O, CPU and communication cost; and it is expressed with respect to either, the total time or the response time. The total time is the sum of all time (also referred to as cost) components, while the response time is the elapsed time from the initiation to the completion of the query. A general formula for determining the total time can be specified as follows [LMHD85]:

$$Total\_time = T_{CPU} * \#insts + T_{I/O} * \#I/Os + T_{MSG} * \#msgs + T_{TR} * \#bytes$$

The two first components measure the local processing time, where T_{CPU} is the time of a *CPU* instruction and $T_{I/O}$ is the time of a disk *I/O*. The communication time is depicted by the two last components. T_{MSG} is the fixed time of initiating and receiving a message, while T_{TR} is the time it takes to transmit a data unit from one site to another. The data unit is given here in terms of bytes (*#bytes* is the sum of the sizes of all messages), but could be in different units (e.g. blocks). Such cost model however does not consider intra-query parallelism. When the response time of the query is the objective function of the optimizer, parallel local processing and parallel communications must also be considered [BaBu90]. A general formula for response time is presented below; where *seq #x*, in which *x* can be instructions *(insts)*, *I/O*, *messages (msgs)* or *bytes*, is the maximum number of *x* which must be done sequentially for the execution of the query. Thus any processing and communication done in parallel is ignored.

$$Response\_time = T_{CPU} * seq\_\#insts + T_{I/O} * seq\_\#I/Os + T_{MSG} * seq\_\#msgs + T_{TR} * seq\_\#bytes$$

The main factor affecting the performance of an execution strategy is the size of the intermediate results that are produced as the output of each operator execution. In distributed environments when the operations must be computed at different nodes, intermediate results must be transmitted over the network. The time to transmit units of data (e.g. bytes, blocks) from one node to another is one of the major concerns for calculating the cost of a query plan. It is of prime interest to estimate the size of intermediate results. Such estimation is based on statistical information about source fragments and formulas to predict the size of operation results. For each of basic algebraic operators (selection, projection, Cartesian product, join, semi-join, union, and difference) a formula is defined for estimating the cardinalities the operator result. Such formula depends on the cardinality of the involved sources and on the operator selectivity factor, this is, the amount of data items (e.g tuples) that hold the operator predicate (e.g. a condition <*attribute*, *comparison operator*, *value*>).

Given the complexity for estimating the cost of query plans, most of cost functions are based on simple approximations of what the system actually does. Also, such functions consider traditional assumptions, like uniform distribution of values and independence of attributes (i.e. the value of an attribute does not affect the value of any other attribute), and a uniform distribution of data among

nodes and not duplicates of data at the different nodes. These assumptions are often wrong in practice, but they make the problem tractable.

In conclusion, global query optimization highly depends on the allocation and available information on source fragments. Some of most important information concern sources indexes, sources cardinality, statistics such as distribution of attributes-values, and operators selectivity. There is a direct trade-off between the precision of the statistics and the cost of managing them, the more precise statistics being the more costly [PiCo84]

2.2 TECHNIQUES

This section reviews the query optimization techniques for: (i) exploring the search space, (ii) deciding the site and (iii) deciding the moment for achieving the optimization process. This analysis aids the better understanding of query optimization techniques based on query feedback (Section 2.4). Optimization techniques using query feedback are variations, or extensions of the generic optimization techniques studied in this section.

This section focuses on the query optimization part related to the search of alternative plans. The specification of the search space and the definition of a cost model are also essential parts of query optimization; however go deeper in these subjects is out of the scope of this thesis. Another important aspect that this section analyzes corresponds to the optimization site; in distributed environments the query optimization responsibility may be assigned to a single node, to some specific nodes or each node in the system is in charge of optimizing the queries that it receives by their own means. Finally, this investigation also concerns the optimization timing; this is, the moment when the optimization process is achieved. This aspect is fundamental for optimizing queries in environments where data, availability of resources and network topology change over time.

2.2.1 Searching the optimal plan

Traditional query optimization strategies have been classified in three main categories [OuBo04]:

- *Heuristic-based*. Heuristic rules are used to *re-arrange* the different operations in a query execution plan. For example, to minimize the size of intermediate results.
- *Cost-based*. The costs of different strategies are estimated and the best one is selected in order to minimize the objective cost function. For example, the number of I/Os.
- *Hybrid*. Heuristic rules and cost estimates are combined together.

This section particularly focuses on the optimization part related to plan generation for searching the space of alternative plans. The other optimization issues i.e. search space and cost model specifications are out of the scope of this thesis. A large number of alternative search strategies have been proposed; all of them based on heuristics and/or cost estimation. The most representative strategies are the deterministic strategies [SACL79][KoSt00] and the randomized strategies [IoWo87][NaSS86][SwGu88][Swam89]. Genetic algorithms [OwKS05] correspond to an artificial intelligence technique that even if seldom used; we include it in our studies because of its attractiveness to improve the performance of information retrieval systems [KuSV00].

Deterministic strategies solve complex problems by splitting them in simpler parts (sub-problems), solving such sub-problems separately and then combining the partial solutions to reach the overall solution. Randomized strategies concentrate on searching for the optimal solution around some particular points within the search space. They do not guarantee that the best solution is obtained, but

avoid the high cost of the optimization process, in terms of memory and time consumption. Genetic algorithms mimic the biological evolution in their search for the optimal solution [Mitc99]. The main idea is to starting from some initial set (i.e. population) of solutions, to generate offspring by random crossover and mutation. The best individuals in the population survive through different generations (until e.g. customizable number of generations, homogeneous population above some cost threshold). Next sections detail these strategies and seminal variations.

Deterministic strategies

Deterministic strategies perform some sort of deterministic search of the solution space, either through exhaustive search, or by applying some heuristics pruning the space. The algorithms based on the application of heuristics only typically have polynomial time and space complexity, but they produce plans that are often orders of magnitude more expensive than a plan generated by exhaustive search algorithm. All published exhaustive search algorithms have exponential time and space complexity, however they guarantee to find the optimal query plan according to a given cost model.

The most popular deterministic strategy is *dynamic programming* proposed by P. Selinger et al. [SACL79] in the System R context. Such strategy suggests a bottom-up plan generation building more complex plans from those that have been previously constructed until achieving a complete plan. It comprises three phases.

- phase 1: In the first phase, the dynamic strategy builds a partial plan for accessing each source in the query. Typically there are several different access plans for a source, thus all the possible plans per source are enumerated; the optimal plan for each plan is selected and retained for the next phase.
- phase 2: In the second phase, the strategy enumerates all possible two-way join plans using the access plans from the previous phase as building blocks. Again, the algorithm would enumerate alternative join plans and select the optimal one. The algorithm continues iterating over the join operations in the query until it has enumerated all n-way join plans.
- phase 3: In the third phase the selected query plan is completed attaching operators e.g. projection, sort or group-by if necessary.

Dynamic programming is almost exhaustive building all possible plans (breadth-first) before it chooses the optimal plan. This property assures finding the optimal plan from all the plans that the search space includes. To reduce the optimization cost, partial plans that are not likely to lead to the optimal plan are discarded as soon as possible. It incurs an acceptable optimization cost (in terms of time and space) for optimizing a query that includes a reduced number of sources. While this algorithm produces good optimization results (i.e. optimal plans), its high complexity can be prohibitive for optimizing complex queries.

Another appreciated property of dynamic programming is that it can easily be extended, for example for optimizing queries in distributed data systems, where in addition of deciding which access paths, join ordering and join methods use, is necessary to indicate the nodes of the system that must compute each operation. The *dynamic programming for distributed data systems* extends the basic dynamic programming phases as follows:

- phase 1: If a source is distributed, different plans to access the fragments of such source at each site must be generated.
- phase 2:The operators for combining such plans must be annotated by specifying at which site the join must be carried out (e.g. at the site where the outer source is produced, at the site where the inner source is produced, or at other interesting site).

- phase 3: A data shipping operator must be attached if the top-level operator of a plan is not executed at the site at which the result of the query must be returned (i.e. client site).

Greedy algorithms have been proposed as another deterministic strategy [SwGu88][ShYT93][LVZC91]. Just like dynamic programming, this algorithm has three phases and constructs plans in a bottom-up way. It uses the same partial access plans and join plans as bricks for generating a complete plan. However, during the second phase and for each join iteration this algorithm applies a rigorous evaluation plan function that dictates joins ordering (i.e. the next best join). Such evaluation plan functions can estimate the minimum cost, the minimum cardinality or the minimum selectivity. Another option is a function that peeks into the future for evaluating a subplan by generating a complete plan from a given subplan and uses the cost of this complete plan as a metric for evaluating the subplan. Greedy algorithms are faster than dynamic programming since avoids the enumeration of all possible join ordering for each n-way plan, but they typically produce sub-optimal query plans [StMK97].

Kossmann and Stocker present another deterministic strategy called *Iterative Dynamic programming* (IDP) [KoSt00]. Such algorithm applies a greedy algorithm to find the set of sources that should be joined early. Then, the dynamic programming algorithm generates a good plan for joining the sources selected in the previous step. The algorithm continues to optimize the processing of the temporally source generated by the subplan generated previously, and all the other sources of the query by iteratively applying the greedy algorithm and dynamic programming until a complete plan for the query is achieved. It has reasonable complexity (i.e. polynomial) and produces in most situations very good plans. Experiments have shown that IDP produces better plans than other algorithms (e.g. random algorithms) in situations in which dynamic programming is not viable because of its high (exponential) complexity.

Randomized

Randomized strategies are desirable for complex queries, where dynamic programming becomes too expensive in terms of resources consumption. Such strategies do not guarantee to find the optimal query plan, but avoid the worst plans, as well as high cost of optimization in terms of memory and time consumption. These algorithms concentrate on searching for the optimal solution around some particular points in a solution space and connect these points by edges that are defined by a set of transformations.

A randomized algorithm first generates one or more start plans by a greedy strategy. Then, random transformations are progressively applied with the objective to enumerate equivalent plans that improve the performance of the start plan. An example of typical transformation consists in exchanging two randomly chosen operand sources. The applied transformations depend on the solution restrictions.

These kinds of algorithms were initially proposed in the context of extensible SGBD [ElSh11]. In this approach, the heuristics consist in the ordering of the applied transformations. The most used heuristics are “selection first”, “avoiding Cartesian product”, “diminishing of constituents” (i.e. apply the projections as soon as possible). Such heuristics reduce the size of intermediate results. Other algorithms utilize heuristics for exploring the search space with a randomly path (random walks). [BrGJ10] is a recent relevant work.

Iterative improvement and *Simulated annealing* [StMK97] and are seminal examples of randomized algorithms. In the iterative improvement algorithm, once selecting a random starting point, the algorithm searches a minimum cost point using a strategy similar to hill-climbing. Beginning at the starting point, a random neighbor (i.e. a point that can be reached by exactly one

transformation) is selected. If the cost associated with the neighboring point is lower than the cost of the current point, the transformation is carried out and a new neighbor with the lower cost is sought. Simulated annealing is a variant on iterative improvement [IoWo87] [SwGu88]. In contrast to iterative improvement, it carries out the transformation of a plan even if the cost of the neighboring point is higher, but with certain probability that eventually the minimal cost will be reached.

Two-phase optimization is another variation of the basic randomized strategy. First, for a number of randomly selected starting points, the neighbor with the minimal cost is sought by applying iterative improvement. Then, from the lowest of these local minima, the simulated annealing algorithm is started in order to search the neighborhood for better solutions. It has been shown experimentally that randomized strategies provide better performance than deterministic strategies as soon as the query involves more than several relations [IoKa90].

Genetic algorithms

Genetic algorithms are designed to simulate the natural evolution process. They have been applied for join order optimization [BeFI91]. We provide a brief overview of genetic algorithms. One of the most important characteristics of genetic algorithms is that they do not work on a single solution, but on a set of solutions, the *population*.

A genetic algorithm first generates a population comprising solutions randomly created. Such population corresponds to the "zero" generation of solutions. Then, each next generation is determined as follows:

1. The solutions of the population minimizing a given cost function are propagated into the next generation (*selection*).
2. Some solutions propagated in the previous step are combined (*crossover*).
3. Some solutions generated in the previous step (not necessarily those minimizing the cost function) are altered randomly (*mutation*).

This loop is iterated until the best solution in the population has reached the desired quality, certain predetermined number of generations has been produced or no improvement has been observed for a certain number of generations. Seminal works are presented in [ViPa11][OwKS05] [IbSS09].

2.2.2 Optimization timing

An important aspect to be considered in query optimization (centralized and distributed) is the moment when the optimization process takes place. A query may be optimized statically at compilation time, dynamically at execution time, or using a hybrid approach that generates parts of the query plan at compilation and at execution time. This decision depends on the available information about the state of the system [OzVa11].

The traditional approach is to optimize a query at compilation time. Such approach is called *static-timing* query optimization. In this approach, the query execution cannot adapt to changes, for example shifts in the load of sites. Therefore, it may lead to plans of poor performance in some situations (e.g. data systems with dynamic and autonomous nodes).

To deal with this problem, an approach that optimizes queries at execution time has been proposed. Such approach is called *dynamic-timing* query optimization. The idea is to start executing a partial plan (the first operators); the selection of the next operators is based on information obtained from executing the precedent operators, thus taking into consideration last changes in the execution

environment, and minimizing the probability of choosing a bad plan. A hybrid approach has been proposed in order to taking the advantages of static and dynamic techniques.

Static-timing query optimization

The static query optimization clearly separates the generation of the query plan, which is done at compilation time as mentioned before, and its execution. The input of the optimization process is an algebraic tree resulting from the preliminary query optimization steps (e.g. query decomposition and data localization in distributed data systems). The output is a query plan that implements the optimal strategy for executing a given query.

The general optimization algorithm consists of two major steps. First, it selects the best access methods (e.g. partial plan, data access algorithm) for each source involved in the query. This choice is mostly based on the selectivity factor of the select predicates over a source. Second, it examines all possible permutations of joins ordering by applying commutative and associative rules; it estimates the cost of each alternative plan by applying a cost function, then it selects the plan that minimizes the cost function. To reduce the cost of the optimization process the search space is reduce by applying dynamic programming.

Static query optimization is done once; in this sense static techniques amortize the cost of the optimization process. However, the cost of query plans is estimated basing on information, for example the size of intermediate results, which is known until run time. Errors in these estimates can lead to the selection of suboptimal query plans. Thus, an accurate cost model is especially important to predict the cost of alternative query plans; in consequence a proper maintenance of the information catalog is critical. This technique is mostly used by exhaustive enumeration algorithms. The most popular static query optimization algorithm is that of System R [AMPT76], one of the first relational database systems.

Dynamic-timing query optimization

In dynamic-timing query optimization the query plan generation process interleaves the optimization process with the execution process. A part of the plan is generated; it is executed while the generation of another part of the plan is carried out. In this technique there is no need for a cost model, since during the optimization process is possible to count on with some measures (e.g. the real size of intermediate results) gathered during partial plan execution.

The general algorithm recursively decomposes a query expressed in some high-level declarative language (i.e. SQL query) into a sequence of subqueries having a single source in common. The condition for executing a resulting subquery is that it must be defined considering a single source. This decomposition uses two basic techniques: detachment and substitution.

The detachment technique splits a query Q (comprising sources A, B, C) into Q1 (comprising source A) and Q2 (comprising sources B and C). The evaluation of Q2 produces the temporal result set D (D comprises the data from A and B that holds certain conditions). The detachment technique reduces the size of the source on which Q is defined (sources A, B, C versus A, D). Detachment extracts the select operations, which are usually the most selective ones. This can have adverse effects on performance if the selection has bad selectivity.

There are queries comprising more than one source that cannot be reduced by applying the detachment technique (if its query graph is a chain with two nodes or a cycle with k nodes, where k>2

[WoYo76]). The substitution technique is applied in this case. This technique substitutes a source by some of its data items (e.g. tuple). For each item obtained the subquery is recursively processed by substitution. Finally, the subqueries (mono-source) are processed by selecting the best method for accessing the source (e.g. index, sequential scan). Then, the algorithm tries to minimize the sizes of intermediate results in ordering binary operations.

Dynamic-timing query optimization can be done several times, at any point during the execution of a query. The selection of the best next operator to be added to the query plan is based on more accurate information from the result of the operators executed previously; thus minimizing the probability of choosing a bad plan. In contrast, it may be an expensive task because of its several occurrences. In dynamic query optimization, even if data statistics are not needed to estimate the size of intermediate results as static query optimization, they are still necessary for selecting the first operators of the query plan. This approach is best for ad-hoc queries. The most popular dynamic query optimization algorithm is that of INGRES [Ston86].

Hybrid-timing query optimization

Hybrid query optimization attempts to provide the advantages of static query optimization while avoiding inaccurate estimates of plans cost. The approach is basically static, but further optimization decisions may take place at run time during a re-optimization phase [CWYA81]. Thus, plans that have become infeasible (e.g., because indices have been dropped) or suboptimal (e.g. because of changes in relation sizes) are re-optimized. However, detecting suboptimal plans is hard and this approach tends to perform much more re-optimization than necessary.

Some optimization decisions are made at runtime. Thus, plans are produced at compilation time using any static algorithm, but some of their parts (i.e. subplans) comprise partial order of operators. Partial order refers that the optimizer envisages alternative subplans that can be better or worst according to different execution parameters, but that their cost cannot be estimated at compilation time due to the lack of information. In such situation the cost of alternative plans is *incomparable*. The information required for costs estimation is retrieved at execution time.

This approach was pioneered in System R by adding a conditional runtime re-optimization phase for execution plans statically optimized [CWYA81][Grae94]. Experimentation with the Volcano query optimizer [Grae94] has shown that this hybrid query optimization outperforms both dynamic and static query optimization. In particular, the overhead of dynamic query execution plan evaluation at startup time is significantly less than that of dynamic optimization, and the reduced execution time of dynamic query execution plans relative to static query execution plans more than offsets the startup time overhead.

2.2.3 Optimization site

In distributed data systems, a single or several nodes may be responsible of the query optimization process [OzVa11]. This subject is tightly related to the autonomy of control of a system, on the one hand we distinguishes systems that totally centralizes the control of query processing, thus query optimization in a single system element. On the other hand, there are systems where once a query is submitted; control over its processing is not longer possible. Thus, in such systems each of its elements is responsible for optimizing the queries that it receive.

The centralized approach is simpler but requires knowledge of the entire distributed data, most distribute data systems centralizes the optimization responsibility to a single system entity. For

example, client/server architecture systems, where the server can have a global view of the system, a centralized control over queries execution, and full-fledge database capabilities. Other data systems partially distribute query optimization assigning major optimization decisions to some specific nodes of the system. However this technique still relies on strong assumptions about the state of the system. Multi-database systems are clear examples of this partial distribution of the optimization process.

Database systems have evolved toward a higher degree of distribution. While it has been a long way from central databases to truly distributed databases, we currently see first explorations toward true peer-to-peer data management infrastructures which will have all characteristics of P2P systems. Such systems require complete decentralized approaches, where each node has local control of data, dynamic addition and removal of peers, only local knowledge of available data and schemas and self -organization and -optimization. Few efforts have been put into extending dedicated query optimization functionalities. Next sections summarize the distribution of query optimization responsibilities for the centralized, partially distributed, and full distributed techniques.

Centralized

In the centralized query optimization, some preliminary optimization processes are carried out at the client where the query is posed, for example, query decomposition. The server is responsible for optimizing and executing the query. This approach makes sense because operations such as query analysis and query rewrite can very well be executed at the clients so that they do not disturb the server, whereas steps such as query optimization require a good knowledge of the current state of the system i.e. the load on the server, and should therefore, be carried out by the server.

Partially distributed

In the partially distributed optimization, one site makes the major decisions and other sites can make local decisions are also frequent. For example, System R* [RDDP82] uses a hybrid approach.

Distributed

In systems with many servers, no single server has complete knowledge of the whole system. In such systems one server needs to carry out query optimization; e.g. the server located closest to the client. This server needs to either guess the state of the network and other servers based on statistics of the past or try to discover the load of other servers by asking them for their current load. While asking is obviously better than guessing, in terms of generating good plans, asking involves at least two extra messages for every server that is potentially involved in a query.

2.3 METADATA FOR QUERY OPTIMIZATION

Query optimization highly relies on information about data; such information is typically called metadata and is stored in a catalog [Koss00]. It maintains the schema of data (i.e. definition of sources, views of such sources, user-defined types and functions, etc.), information about the fragmentation of global sources, as well as the replications of such fragments and instruction for materializing the global sources. In addition, the catalog includes the necessary information for the computation of the cost function.

The catalog can be seen as to another data source, for instance, in relational database systems the catalog is stored like all other data tables. The way to storage and manage such catalog are important issues to consider in a data system. In a distributed system also must be considered the location for placing the catalogs.

2.3.1 Statistics and Histograms

Statistical information about the data sources is fundamental for the computation of a cost function. In particular, they are used to estimate the cardinality of intermediate results product of the plan operators. For a source S defined over the attributes $A = \{ a_1, a_2, \ldots, a_n\}$, and fragmented as $S = \{s_1, s_2, \ldots, s_n\}$, the statistical data typically are the following [ElSh11]:

- For each attribute a_i, its length (in number of bytes), denoted by $length(a_i)$, and for each attribute a_i of each fragment s_j, the number of distinct values of a_i, with the cardinality of the projection of fragment s_j on a_i, denoted by $card(\Pi a_i (s_j))$.
- For the domain of each attribute a_i, which is defined on a set of values that can be ordered (e.g., integers or reals), the minimum and maximum possible values, denoted by $min(a_i)$ and $max(a_i)$.
- For the domain of each attribute a_i, the cardinality of the domain of a_i, denoted by $card(dom[a_i])$. This value gives the number of unique values in the $dom[a_i]$.
- The number of tuples in each fragment s_j, denoted by $card(s_j)$.

For estimating the cardinalities of intermediate results of queries rely on the strong assumption that the distribution of attribute values in a relation is uniform. The advantage of this assumption is that the cost of managing the statistics is minimal since only the number of distinct attribute values is needed. However, this assumption is not practical. In case of skewed data distributions, it can result in fairly inaccurate estimations and query execution plans which are far from the optimal. An effective solution to accurately capture data distributions is to use histograms.

A *histogram* on attribute *a* from the source *S* is a set of buckets. Each bucket *bi* describes a range of values of *a*, denoted by $range_i$, with its associated frequency f_i and number of distinct values d_i. f_i gives the amount of units of data (e.g. tuples) of S where $S:a \in range_i$. d_i gives the number of distinct values of a where $S:a \in range_i$.

This representation of a source's attribute can capture non-uniform distributions of values, with the buckets adapted to the different ranges. However, within a bucket, the distribution of attribute values is assumed to be uniform. Histograms can be used to accurately estimate the selectivity of selection operations. They can also be used for more complex queries including selection, projection and join. However, the precise estimation of join selectivity remains difficult and depends on the type of the histogram [PHIS96].

2.3.2 Catalog management approaches

Efficient catalog management in (centralized and distributed) data systems is critical to ensure satisfactory performance in the execution of queries. Read and update the catalog information are the most common task involved in the catalog management. Three popular approaches have been proposed: centralized catalog, fully replicated catalogs, and partially replicated catalogs. The choice of the approach depends on the characteristics of the data system, as well as those of the applications turning over the system [Koss00][OzVa11].

Centralized Catalogs

In the centralized approach, the entire catalog is stored in one single site. Owing to its central nature, it is easy to implement. However, a centralized catalog can quickly become a bottleneck; it is not a good option for the scalability of the system. The regular read and update task in this approach are as

follow: For read operations from non-central sites, the requested catalog data is locked at the central site and is then sent to the requesting site. On completion of the read operation, an acknowledgement is sent to the central site, which in turn unlocks this data. All update operations must be processed through the central site.

Sensor networks are the kind of systems where this approach is typically applied. In TinyDB [MFHH05], for instance, metadata correspond to statistics describing important physical information such as position, density and connectivity of sensors. It also comprises system information, such as system workload and network scalability. Moreover, it includes workload distribution of generated values, events and user functions handled at each node.

Actually, due to the amount of required metadata, and the dynamicity of sensor networks, collect metadata in a central node seams not to be the best option. During the query optimization information about the status of the sensor network is required. The query optimization is centralized in the best station. The catalog is periodically updated by collecting metadata from the sensor nodes. This involves the transmission of important amounts of data that can provoke the congestion of the network.

Fully Replicated Catalogs

In the fully replicated approach, identical copies of the complete catalog are present at each site. This scheme facilitates faster reads by allowing them to be answered locally. However, all updates must be broadcast to all sites. Updates are treated as transactions and a centralized two-phase commit scheme is employed to ensure catalog consistency. As with the centralized scheme, write-intensive applications may cause increased network traffic due to the broadcast associated with the writes.

Partially Replicated Catalogs

The centralized and fully replicated schemes restrict site autonomy since they must ensure a consistent global view of the catalog. Under the partially replicated scheme, each site maintains complete catalog information on data stored locally at that site. Each site is also permitted to cache entries retrieved from remote sites. However, there are no guarantees that these cached copies will be the most recent and updated. The system tracks catalog entries for sites where the object was created and for sites that contain copies of this object. Any changes to copies are propagated immediately to the original (birth) site.

Retrieving updated copies to replace stale data may be delayed until an access to this data occurs. In general, fragments of relations across sites should be uniquely accessible. Also, to ensure data distribution transparency, users should be allowed to create synonyms for remote objects and use these synonyms for subsequent referrals.

Such catalogs can be implemented in a hierarchical way as described in [EiKK97]. The main idea behind the distributed catalogs is that for many P2P applications, the distribution of the underlying data among servers is not random. It is often the case that data are stored, grouped, replicated and queried according to one or more categorization hierarchies that are natural for the application.

2.4 OPTIMIZATION USING QUERY FEEDBACK

In this section we review seminal optimization proposals based on feedback obtained from the execution of queries. This feedback serves for estimating plans cost when there is incomplete information on data at compilation time (Section 2.3 addresses the optimization timing techniques).

These approaches have considered different kinds of query feedback, such as statistical information (data production rates, number and frequency of data values, cardinality of sources, availability indices, etc.), measures of resources availability, resources consumption and plans for queries evaluation.

In the literature, the optimization approaches that exploit feedback on-the-fly are known as adaptive techniques. They are inspired on dynamic or hybrid (static-dynamic) optimization timing principles. We classify the optimization approaches that store feedback for its latter exploitation as evaluation, and use it to improve the processing of further queries. Such learning approaches involve new aspects to consider to accomplish a successful optimization process e.g. the selection of suitable storage structures, as well as mechanisms for feedback exploitation (i.e. retrieval and reuse) and management (i.e. advantageous organization, insertion, update and deletion). [KaDe98][INSS97] [UrFA98][VaKi00][AbCh99a][SLMK01] are some representative pioneer works. More recent learning-based optimization techniques have been proposed in [ChLH12] [BaBr10][AÇRU12], just to mention a few.

For analyzing such works we divide this chapter in three major parts. The first part exposes adaptive query optimization techniques (Section 2.4.1). The second part focuses on optimization based on the storage and reutilization of query plans. This kind of query optimization techniques are well known as plan caching techniques (Section 2.4.2). Finally, the third part addresses optimization approaches centered on collecting, maintaining and repairing statistics and metrics about resources consumption with the objective to improve the accuracy of plans cost estimation (2.4.3).

2.4.1 Adaptive query optimization

Traditionally, there is a clear gap between query optimization and query execution: query plans are optimized at compilation time and sent to the execution engines for evaluation until all query results are completely computed [DeIR06]. Adaptive optimization takes decisions at compilation and at execution time. It is inspired on the dynamic and static-dynamic optimization techniques. This approach faces the lack of information and occurrence of unpredictable data characteristics and environment events during query execution [OuBo04].

The adaptive query optimization process comprises five generic stages [Liu00]: (i) Plan optimization (generating an initial plan for a query), (ii) Plan Monitoring (monitoring the plan status, system performance, as well as data characteristics), (iii) Plan Analysis (analyzing how well the current plan functions and deciding whether an adaptation is needed), (vi) Plan Re-optimization (finding a new plan that is better than the current plan), and (v) Plan Migration (migrating the current plan to the new plan). These stages form a loop and are continuously executed until complete query result is computed. The literature exposes these steps from an adaptive query process, and not specifically from the query optimization. However, the adaptation is based on optimization decisions (only); the current query plan (optimization output) is modified to improve the query evaluation (optimization objective). In summary, any adaptation actions are responsibility of the optimization process.

Research on adaptive query optimization follows two main directions. The first approach responds to changes in the evaluation environment by modifying the execution plan at runtime (e.g. by changing the operators used or the order in which they are evaluated). The other approach involves the development of operators that deal more flexibly with unpredictable conditions and adapt their behavior by collecting and taking into consideration information that becomes available at query

runtime about how query evaluation is proceeding and about changes in the wider execution environment. Next sections present seminal works basing on this approach.

An interesting characterization of adaptive query optimization process is given in [OuBo04][Liu00]. It states that query optimization is adaptive if:

- it receives information from its environment
- it uses that information to determine its behavior, and
- this process iterates over time, generating a feedback loop between environment and behavior

I would add to the previous characteristics that, query optimization is an adaptive process if the gathered feedback is keep for a short period of time, only while it is useful for taking some optimization decisions at execution time. This study refer to seminal works based on the adaptive query optimization principle, such as *re-optimization* foundations [KaDe98], *parametric query optimization* [INSS97] [AlDB12], *dynamic query scrambling* [UrFA98] because unavailable memory and dynamic *query scheduling* [BFMV00a] due to unexpected delays. It also survey *ECA rules* to face undesirable environment behavior [IFFL99], and *Eddies* [AvHe00][TiDe03] for the adaptive routing of bursty data-flows on execution time.

In spite of their different adaptation mechanisms, all these works aim the improvement of the global evaluation of queries. Such approaches recourse to different kinds of feedback, for example system parameters, statistics and consumption of resources. Also, each of them is consecrated to the optimization of different objective e.g. minimizing time, saving memory. Adaptive query optimization includes several challenges, such as the detection of plans with poorer performance than the expected; identify the appropriate moment to re-optimize a query plan, and the temporary maintenance of query feedback.

The remainder of this section presents a summary of such techniques: re-optimization principle and main contributions such as, parametric query optimization, query scrambling, dynamic query scheduling, Tukwila [IFFL99] and Telegraph [HFCD00] projects. Finally, we present a comparative synthesis to show the strengths and weaknesses.

Re-optimization [KaDe98]

This work describes a dynamic re-optimization algorithm that detects sub-optimality of a query execution plan during query execution in order to re-optimize and improve its performance. The basic idea is to collect statistics (Section 2.4 exposes classical statistics and histograms) at key points during the execution of complex queries. Thus, during the query optimization, the produced plan is annotated with the various estimates and statistics used by the optimizer.

It is assume that a conventional query optimizer exists for producing an execution plan at compilation time for a given query. Current statistics are collected at-query execution time. Such statistics are compared with the annotated in the query plan. The difference between these statistics is taken as an indicator of whether the query execution-plan is suboptimal. The new statistics (much more accurate than the initial those estimated by the optimizer) are used to improve the execution of the remainder of the query. Such improvements correspond: (i) to re-allocate shared resources (e.g. memory) to the various operators of the query, (ii) to detect whether the remainder of the execution plan must be re-optimized.

The collection of statistics at query execution time may result in a significant overhead. To prevent this problem the collected statistics, and the most effective points to collect them are

determined and statistic collection operators (i.e. *StatisticCollector*) are inserted into the query execution plan (see Figure 2.2).

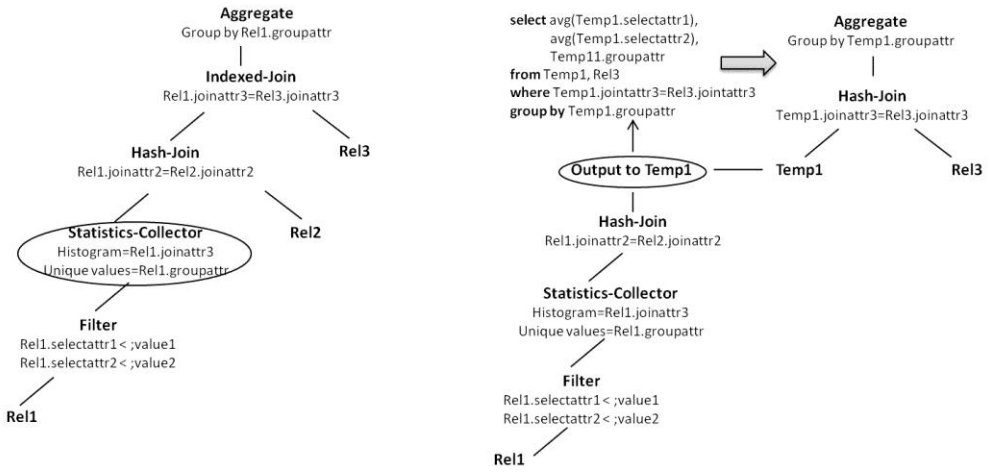

Figure 2.2 Query plan including the *StatisticsCollector* operator

There is a filter operation that applies selection predicates to the Rel1 relation. Just after the filter operation, a statistics collector operator is inserted into the query execution plan. As the tuples are being produced by the filter operator, they can be examined by a statistics collection routine, and the required statistics can be gathered without interrupting the normal execution of the query. Thus, for example, the cardinality of the result of the filter operation can be computed by keeping a running count of the number of tuples that stream past the statistics collection routine, and the average tuple size can be computed by keeping a running average.

The statistics-collectors insertion algorithm decides what statistics to collect, and where insert within the query plan *StatisticsCollector* operators. Such operators are inserted at compilation time after a conventional optimizer has produced a first query execution plan. To make this decisions, the algorithm generates a list of the potentially useful statistics, for instance, a histogram on a particular attribute is potentially useful if that attribute is part of a join or a selection predicate.

Given this list, the algorithm determines those statistics that should to be discarded and those that should be retained. The retained statistics are those whose computation time is less than a parameter (μ) that indicates a maximum acceptable overhead; and that are the most effective in determining the sub-optimality of the plan. Thus, it is required to estimate the time for computing each statistic, this is done using the optimizer estimates of size of intermediate results. Also, it is required to measure the effectiveness of statistics to detect sub-optimality of a plan. This is detected by two key factors: (i) the probability that the corresponding optimizer estimates are inaccurate (if the probability that the first estimates are accurate there is no much reason to gather such statistics); and the fraction of the query execution plan that might be affected by a statistic (if the part is minimal it is not required neither to collect such statistics). This approach was validated in the context of the Paradise Database System. The experiments report significant improvement in the performance of complex queries.

Parametric query optimization [GrWa89][INSS97][CoGr94]

Parametric query optimization proposes optimizing queries considering some parameter, important for estimating query plans cost, whose values can change between compilation-time and execution time. Such parameters may concern data (e.g. cardinality, indexes) and system environment (e.g. availability of resources like memory, disk, processing power, etc.). This approach attempts to identify at compile

time several execution plans, each one of which is optimal for a subset of all possible values of certain run-time parameters. At run time, when the actual parameter values are known, the appropriate plan should be identifiable with essentially no overhead [INSS97].

The objective of this approach is twofold: (i) *objectively* explore the search space, and (ii) reduce the re-optimization occurrences (waist of computing resources). For the search space exploration certain assumptions about data are made (e.g. value distribution); however these assumptions may be violated at runtime (e.g. the database content change). This fact may lead to selecting query plans with poor performance, otherwise to re-optimize the plan. The parametric query optimization principle consider that a search space exploration by using run-time parameters (i.e. system, data and/or query parameters) for selecting the optimal query plan may avoid, or at least reduce the coincidences of the previous scenario.

Figure 2.3 shows the global view of the parametric query optimization process. The choice of an optimal plan is carried out in two steps: (i) the generation of query plans (*Parametric query optimizer*) APi associated to possible parameter values (Pi); and (ii) the selection (*Chooser*) of the optimal query plan, the cost of query plans is estimated according to the parameter values reveled during the execution.

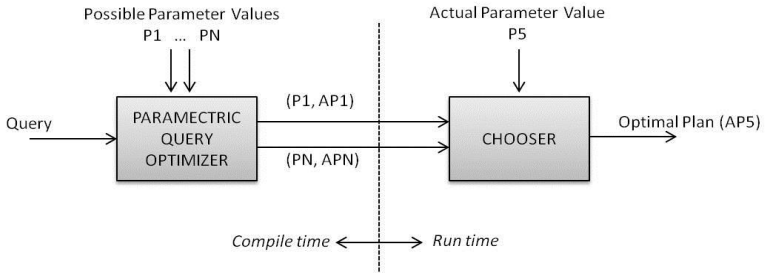

Figure 2.3 Overall architecture of parametric query optimization

The earliest significant work in this area is by Graefe and Ward [GrWa89]. They discuss the implementation of dynamic query plans in the Volcano optimizer generator [GrMc93]. These are plans that include a choose-plan operator, which chooses among multiple available conventional plans given the values of certain run-time parameters. The proposal is for choose-plan operators to be introduced in all places of a plan where the choice of subplans underneath is sensitive to the values of these parameters. This work includes many important concepts related to parametric query optimization but does not include a complete search strategy to identify the dynamic plans and the positions where the choose-plan operators should be place.

To illustrate the benefits of dynamic plans over traditional, static plans consider a hash join of relations R and S. The size of S is predictable, while the join input from R can be very small or very large depending on a selection of R based on a user variable, Since hash joins perform much better if the smaller of the two inputs is used as the build input [Grae93], two join plans should be included in a dynamic plan for this query. A suitable dynamic plan for this query is shown in Figure 2.4.

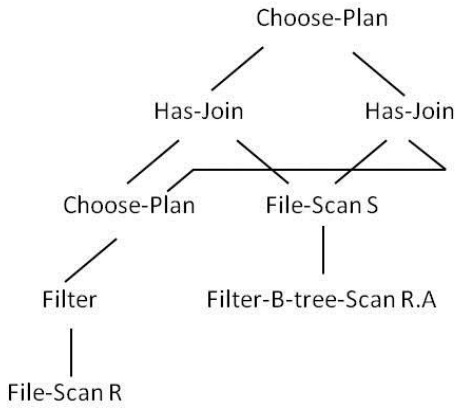

Figure 2.4 Example of dynamic query plan

In principle, the optimal plan generated by parametric query optimization may be different from each distinct combination of all the possible run-time parameter values. Naturally, the total cost of generating all these plans is prohibitive. Ioannidis Y. et al. present an approach to produce distinct plans for values of a selected subset of run-time parameters (i.e. buffer size parameter) [INSS97]. For the exploration of the search space they adopt randomized algorithms, and enhance them with a *sideways information passing* feature that increases their effectiveness in the new task. Such randomized algorithms correspond to: Simulated annealing (SA), [Kirk84][IoWo87], *iterative improvement* (II) [NSS86, SG88], and *two-phase optimization* (2PO) [IoKa90][IoKa91] for parametric query optimization of select-project-join queries; briefly explained in Section 2.3.1.2.

Experimental results of these enhanced algorithms show that they optimize queries for large numbers of buffer sizes in the same time needed by their conventional versions for a single buffer size, without much sacrifice in the output quality and with essentially zero run-time overhead. Another method based on [GrWa89] was subsequently developed by Cole and Graefe [CoGr94]. Once again, the bulk of the optimization effort is done at compilation-time, and during delays carefully selected certain optimization decisions are made at execution-time. It focuses in the generation of dynamic query plans exploring the search space with a search strategy based on dynamic programming.

One of the major proposals of this work is the notion of incomparability of plans cost at compilation time. Missing runtime binding may render impossible to calculate the cost of query plans. If so, alternative plans are only partially ordered by cost (instead of being totally ordered as in traditional query optimization). Thus, the choice of the optimal plan is delayed until start-up-time, when all the parameter values are known cost estimation and the comparison for selecting the optimal plan is feasible.

If two or more alternative plans are incomparable at compilation-time, they are both included in the query plan and linked together by a choose-plan operator, thus creating a dynamic plan as defined in [GrWa89]. The choose-plan operator allows postponement of the choice among two or more equivalent, alternative plans until start-up-time, when the decision can be based on up-to-date knowledge, e.g., the bindings of user variables unbound at compile-time.

In summary, this approach states to resolve the ambiguity in selectivity and cost estimation. The three most important ambiguity problems correspond to: errors in selectivity estimation [IoCh91], unknown run-time bindings for host variables in embedded queries, and unpredictable availability of resources at run-time. In order to validate the approach, they extend the Volcano optimizer generator [GrMc93]. While the prototype is based on the relational data model, the problem of uncertain cost-

model parameters and its solution using incomparable costs are also applicable to other data models that require query optimization based on estimation of uncertain parameters.

Query scrambling [UrFA98][ATFU96]

A *query scrambling* approach is proposed in [ATFU96]. The goal is to react to delays by modifying the query execution plan *on-the-fly*. Unexpected delays are "hidden" by performing other useful works. Scrambling has a two-pronged action: *rescheduling* (scheduling other operators for execution) and *operator synthesis* (new operators are created when there is no other operator to execute). These two techniques are repeated as necessary to modify the query execution plan.

Scrambling policies differ by the degree of parallelism they introduce or the aggressiveness with which scrambling changes the existing query plan. Three important trades-offs must be considered in *re-scheduling*: First, the number of operators to reschedule concurrently. This concerns the benefits of overlapping multiple delays and the cost of materializations used to achieve this overlapping. Second, scheduling individual operators or entire sub-trees. Finally, choice of specific operator(s) to reschedule.

For *operator synthesis*, a significant amount of additional work may be added since the operations were not originally chosen by the optimizer. To avoid this problem a simple heuristic of avoiding Cartesian products to prevent the creation of overly expensive joins is used. However, the performance of this heuristic is highly sensitive to the cardinality of the new operators created.

Dynamic query scheduling [BFMV00a][BFMV00b]

Another dynamic scheduling strategy that deals also with memory limitation has been proposed in [*BFMV00a*][*BFMV00a*]. It is based on monitoring arrival rates at the information sources and memory availability. In the case of significant changes, the execution plan is revised. This means that planning and execution phases are interleaved. The query execution plan is represented by an operator tree with two particular edges: *blocking* and *pipelinable*. In a *blocking edge*, the consumption of data cannot start before it is entirely produced.

In a *pipelinable edge*, data can be consumed one tuple at a time meaning that consumption can start as soon as one tuple is available. It is then possible to characterize the query execution plan by pipelinable chains which represent the maximal set of physical operators linked by pipelinable edges. The query engine will have to concurrently select, schedule, and execute several query fragments (pipelinable chains and partial materializations) while minimizing the response time.

The query engine's main components are the *dynamic query optimizer*, *dynamic query scheduler*, *dynamic query processor*, and *communication manager*. The dynamic query optimizer uses dynamic re-optimization techniques to generate an annotated query execution plan. Those annotations relate to blocking and pipelinable edges, memory requirements, and estimates of results' sizes. The dynamic query scheduler builds a scheduling plan at each scheduling phase triggered by events from the query processor. Scheduling is based on some heuristics, the current execution status, and information about the benefits of materialization of pipelinable chains.

The dynamic query processor concurrently processes query fragments while maximizing the processor use based on priorities defined in the scheduled plan. The execution may be interrupted in case there is no data arriving from sources, a query fragment has ended, or delivery rates have significantly changed. This is reported to the query scheduler and eventually to the query optimizer for scheduling or optimization changes. The communication manager receives data from the different

wrappers for the rest of the system. It also estimates delivery rates at the sources and reports significant changes to the query processor.

Tukwila: ECA rules [IFFL99][ILWF00]

Tukwila is another system addressing adaptiveness in data integration environment [IFFL99]. Adaptiveness is introduced at two levels: (1) between the optimizer and the execution engine, and (2) within the execution engine. In the first level, adaptiveness is deployed by annotating initial query plans by (event-condition-action) *ECA rules*. These rules check some conditions when certain events occur and subsequently trigger the execution of some actions.

Examples of events include operator failure, time-out, and out of memory exceptions. Conditions include the comparison of actual cardinalities known at run-time and those estimated. Finally, actions include rescheduling of the query operator tree, re-optimization of the plan, and alteration of memory allocation. A plan is organized into a partially ordered set of fragments and a set of corresponding rules. Fragments are pipelined units of operators. When a fragment terminates, results are materialized and the rest of the plan can be re-optimized or rescheduled. In addition to data manipulation, operators perform two actions: statistics gathering for the optimizer, and event handler invocation in case a significant event occurs.

The operator tree execution follows the top-down Iterator model described in [Grae93]. For the second level of adaptiveness, two operators are used: *dynamic collectors* and the *double pipelined hash join* operator. The collector operator dynamically chooses relevant sources when a union involves data from possibly overlapping or redundant sources. The optimizer specifies the order to access sources and alternative sources in case of unavailability or slow delivery. A collector will then include a set of children (wrapper calls or table-scans) and a policy for contacting them.

The policy is expressed as a set of event-condition-action (ECA) rules. The double pipelined hash join is a symmetric and incremental join. It aims at producing tuples quickly and masking slow data sources transfer rates. This requires maintaining hash tables for in memory relations. The original double pipelined join has been implemented after a few adaptations. The first adaption was to retrofit the data-driven bottom-up execution model of that operator with the Tukwila's query processing top-down Iterator-based scheme. The second relates to the problem of memory overflow. Two strategies based on swapping are used.

Adaptive join operators for minimizing partial response time [HaHe99][WiAp93][UrFr00][IFFL99]

Join operators for accelerate the production of partial results have been proposed: (i) The Ripple join that is a physical pipelining join operators that maximize the flow of statistical information during processing [HaHe99]; (ii) the XJoin that is a variant of Ripple joins [UrFr00]; the symmetric hash join (SHJ) [WiAp93], and the double pipeline hash joins (DPHJ) that is a family of memory adaptive hash joins [IFFL99].

The Ripple joins generalize block nested loops (in the sense that the roles of inner and outer relation are continually interchanged during processing) and hash joins. Ripple joins adapt their behavior during query evaluation according to gathered data statistics, the user preferences about the accuracy of the partial result, and the time between updates of the result aggregates. Given these user preferences, they adaptively set the rate by which they retrieve tuples from each input of the ripple join.

XJoin is a variation of the Ripple joins, but with lower memory requirements. Its execution comprises three steps. In the first step it builds two hash tables, one for each source. In the first stage, a

tuple, which may reside on disk or memory, is inserted into the hash table for that input upon arrival, and then is immediately used to probe the hash table of the other input. A result tuple will be produced as soon as a match is found. The second step begins when the first step blocks and it is used for producing tuples during delays. Tuples from the disk are then used to produce some part of the result, if the expected amount of tuples generated is above a certain activation threshold. The last step is a clean-up stage as the first two stages may only partially produce the final result. In order to prevent the creation of duplicates, special lists storing specific time-stamps are used. Apart from producing initial results quickly, XJoin is also optimized to hide intermittent delays in data arrival from slow and bursty remote sources by reactively scheduling background processing.

The DPHJs initially splits source relations and holds them in memory. When memory is insufficient, one partition held in memory flushes its hash table to disk and deallocates all but one of its buffer pages. The most efficient variant of DPHJs for utilizing additional memory is when partitions of the inner relation are fetched in memory while the outer relation is being scanned and partitioned. This method reduces I/O and, consequently, the total response time of the query.

Telegraph: Eddies on the River [AvHe00][TiDe03]

The Telegraph project [CCDF03] aims to build a query engine over Web information sources based on an adaptive *data-flow paradigm*. The objective is to adaptively route unpredictable and bursty data-flows through computing resources. The query processor continuously reorders applications of pipelined operators in a query plan at run-time on a tuple-by-tuple basis [AvHe00]. It uses the concept of *eddy*, defined as a *n*-ary tuple router interposed between *n* data sources and a set of query processing operators.

An *eddy* encapsulates the ordering of operators by dynamically routing tuples through them. Figure 2.5 shows an example of an eddy operator. The idea is that there are times during the processing of a binary operator (e.g., join, union) when it is possible to modify the order of the inputs without modifying any state in the operator. Such times are called *moments of symmetry*. They occur at the so called *synchronization barriers*. For the case of *merge join*, this corresponds to one table-scan waiting until the other table-scan produces values larger than any one seen before. Most of the reported work for *eddies* is on the join operator due to its impact on query performance.

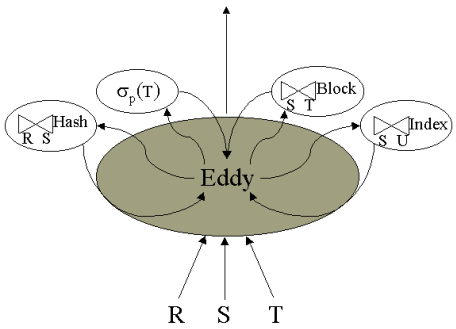

Figure 2.5 An eddy operator in a pipeline

The focus in Telegraph is on join algorithms with frequent times of symmetry, adaptive or non-existent barriers, and minimal ordering constraints. Efficiency of the system depends tightly on the routing policy used in the *eddies*. Different routing policies need to be used under different circumstances. They depend on operator selectivity, operator consumption and production rate, join implementation, and initial delays of input relations. Eddies have been implemented in the context of a shared-nothing parallel query processing framework called River [AATC99].

Discussion

This section exposes the basics of adaptive query optimization, as well as, the most representative works in this domain. Also, it identifies the opportunities that such approach offers to highly dynamic environment by modifying query plans on the fly. There is always a trade-off between the potential benefits from adapting to current conditions on the fly with very high frequency, and the risk of incurring large overheads.

The techniques that have more extensive capabilities in terms of the modifications they can induce in the running plan are more expensive and risky than the others. On the other hand, less complex adaptive strategies (e.g. those that do not affect the logical plan of the query) are generally intended for single processor environments, where it is more common that satisfactorily accurate statistics can be obtained at compile-time.

We distinguish that all these works focus on query evaluation improvement according to an unchanging and unique optimization objective. Such objective typically corresponds to the response time (i.e. providing first partial query results to the user) or the execution time (i.e. providing the complete query result to the user), and few of them on memory consumption, as [WiAp93]. Little attention has been paid to changes in the pool of available processors or may be to energy consumption. In any case, until our knowledge, adaptive query optimization techniques do not address the necessity nowadays applications of optimizing queries according to multiple and customizable objectives.

These techniques are mostly applicable to parallel and distributed settings, taking the adaptation decisions at a global level and relying heavily on the materialization (temporary at execution time) of intermediate results. This happens because in parallel and distributed settings, network costs and resource availability need to be taken into consideration and not adapting to changes in data transfer rates or resources may have detrimental effects. Furthermore, they are applied to distributed data systems that centralized the query optimization process in a single server. They monitor and gather feedback in a system where there is an entity having a global view of the system elements. Thus, they do not consider the distribution of optimization decision to different autonomous system nodes.

Techniques may collect and act on feedback between the executions of different operators in the query plan (i.e. inter-operator). Collection can be triggered after certain operators have been evaluated or after special events, like the arrival of partial results from local data sources in multi-database environments. Other techniques collect feedback during the evaluation of physical operators (i.e. intra-operator). Check-points are added to the operator execution for this purpose. In general, feedback is collected after a block of tuples has been processed. In the limit, this block consists of a single tuple, resulting in a potentially different plan for each tuple. Such feedback is exploited as soon as it is acquired, or it is preserved for its further usage.

The previous analysis considers the common aspects of adaptive optimization techniques; however, they differ in several aspects: (i) the particular problem that they address, (ii) the objective they focus on, (iii) the nature of feedback they collect, and (iv) the frequency at which they can adapt, and (v) their way to carried out adaptation. Table 2.1 summarizes the optimization techniques presented so far in this section according to the evoked dimensions.

Reference	Problem	OpObj	Feedback	Frequency	Realization
Re-optimization [KaDe98]	inaccurate plans cost	execution time	statistics	inter-operator	plan generation algorithm and

	estimation				operator (statistics-collector)
Parametric query optimization [GrWa98, CoGr94, INSS97, IoCh91]	reduce re-optimization occurrences	execution time	statistics and system parameters.	intra-operator	plan generation algorithm
Query scrambling [UFA98, AFTU96]	minimize data arrival rates	response time	delays	inter-operator	algorithm of adaptation
Dynamic query scheduling [BFMV00a, BFMV00b]	minimize response time	response time	delays and memory	inter-operator	optimization algorithm (parallelism)
Tukwila: ECA rules [IFFL99, ILWF00]	adapt to environment changes	response and exe. time	events e.g. time-out, exe. failures	inter-operator	system and operator algorithm
RippleJoin [HeHe99]	satisfy user preferences	response time	delays, results size	intra-operator	operator algorithm
XJoin [UrFr00]	minimize data arrival rates	response time	delays, memory	intra-operator	operator algorithm
Eddies [AvHe00, TiDe03]	adapt to environment changes	response time	statistics	intra-operator	operator and data routing operator

Table 2.1 Adaptive query optimization approaches

2.4.2 Plan caching

The storage and reuse of plans is the basis of plan caching, also known as plan memorization. In this approached the feedback from the evaluation of queries naturally corresponds to query plans (in some representation). Such feedback is typically complemented with supplementary information e.g. statistics, query description, etc.

Some of the main concerns of plan caching are to select feedback representation and structures that favor its reuse. A plan can be reused for a single query or for several queries; the *coverage* of a plan concern the amount of queries that it is useful to solve. The reuse of plans is typically based on comparative functions between queries e.g. equivalence, equality, similarity, etc.; this turns in a queries classification problem, the definition of this function is fundamental. Another problem to consider is the adaptation of plans to fit to the specifications of the new query.

Plan caching is useful to alleviate the optimization overhead for solving complex queries and the systematical necessity to scan the search space for query processing. Former, a query is optimized using typical techniques; the generated optimal plan P is stored in a cache and is reused for solving some further queries without the need to scan the search space. The access and retrieval of reusable plans must be efficient to minimize the overhead of the optimization process; appropriate storage structures are essential (e.g. hash tables, lattices, clusters).

In the eager to find the truly optimal plan (from an exhaustive exploration of the search space), some optimization approaches are very hard and time costly. Such approaches realize the optimization process during a preparation phase (previous to start the execution of queries) or during a backward processes. This is an appropriate approach for optimizing very complex queries or frequent queries, since it does not easy evolve with to changes in a very dynamic environment (as adaptive

query optimization does). Next sections summarize representative query optimization works basing on plan caching.

Plan selection based on query clustering [GPSH02][SeHa03]

This approach is based on plan caching for amortize the query optimization overhead by the fact that a new query is typically optimized afresh. It aims to improve the utility of the plan cache by identifying clusters of queries, in such a way that the queries that belong to a cluster have a common optimal *plan template*. They analyze the characteristics that queries shared when they fit (can be efficiently evaluated) with the same plan template. Thus, a query is represented as a feature vector that includes structural attributes such as the number of tables and joins in the query, as well as statistical quantities such as the sizes of the tables participating in the query.

Using a distance function defined on these feature vectors, queries are grouped into clusters. Each cluster has a representative plan template of the query plan, initially generated by a classical optimizer. This plan template is used to execute all future queries assigned to the cluster. In short, this approach recycles plan templates based on the expectation that its clustering mechanism is likely to assign an execution plan that is identical to what the optimizer would have produced on the same query.

Then, using this similarity definition, *query clusters* are dynamically formed in an incremental manner, with the distance threshold determining the maximum stretch of the cluster. Each cluster has a *representative* for whom the execution plan, as determined by the optimizer, is persistently stored. This plan is used to execute all future queries that are assigned to the cluster. Finally, when a sufficient number of clusters have been formed, a classifier is constructed on the clusters to support efficient identification of the cluster to which a new query may belong, thereby also determining its execution plan.

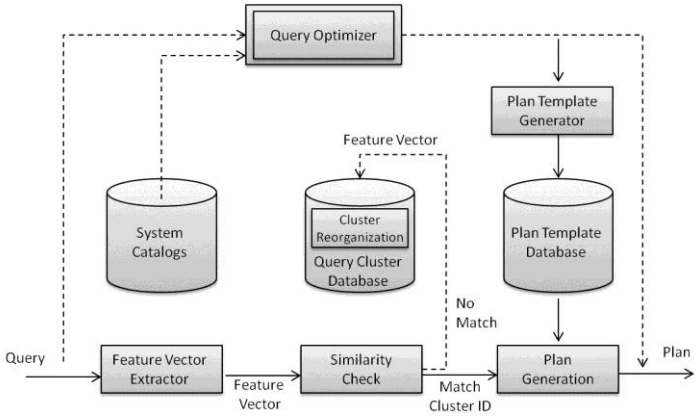

Figure 2.6 The PLASTIC architecture

PLASTIC [SeHa03] is the prototypation of this approach, a *leader* algorithm [Hart75] is used to determine cluster representatives, and a *decision-tree* [SaLa91] is constructed for classification purposes. A block-level diagram of such system components is shown in Figure 2.6 (the solid lines show the sequence of operations in the situation where a matching cluster is found for the new query, while the dashed lines represent the converse situation where no match is available and a fresh cluster is created). The reuse of plans basing on query similarity is the basis of plan caching. The contribution of this approach is the incorporation of a host of new features to the query similarity definition. The problem is the scarcity of such new features in highly distributed systems.

Accurate query optimization by sub-plan memorization [AbCh99a] [AlDB12] [SRRF08] [HeBa10]

[AbCh99a] is a pioneer work proposes to provide the query optimizer with *exact values* for the result size of operators and operator trees (i.e. *sub-plans*), and for the number of distinct values in the output of these subplans. In this approach the query optimizer optimizes the query and records all the sub-plans for which result size or distinct value estimates are required in a data structure, i.e. the *sub-plan memo*. Other works based on a similar plan caching principle are However, other interesting works can be found in *[AlDB12] [SRRF08] [HeBa10]*.

Query optimization in *[AbCh99a]* is done in *phases*. In each phase, the query optimizer fully optimizes the query and produces a query execution plan. In the first phase, the optimizer optimizes the query using its traditional techniques for result size and distinct value estimation. During this optimization, the optimizer records all the sub-plans (operators or operator trees) for which result size or distinct value estimates are required in the *sub-plan memo*. After the optimization is completed, the sub-plans in the sub-plan memo are executed and their actual result sizes and the actual number of distinct values in their outputs are determined and recorded in the sub-plan memo.

In the second phase, the query optimizer re-optimizes the query, but whenever it needs result size or distinct value estimates for a sub-plan for cost estimation, it looks for this sub-plan in the sub-plan memo. If the sub-plan is found, the optimizer uses the accurate result size and distinct value information in the sub-plan memo.

The algorithm used by the query optimizer for searching the plan space when optimizing the query in the second phase is the same algorithm used in the first phase. Thus, most of the sub-plans that the optimizer encounters in the second phase will be ones that were already encountered in the first phase, so they will be found in the sub-plan memo. However, since the second phase uses the more accurate result size and distinct value information found in the sub-plan memo, the optimizer may search parts of the plan space not searched in the first phase and encounter new sub-plans that are not in the subplan memo.

If the optimizer encounters sub-plans that are not in the sub-plan memo, their cost is estimated using traditional techniques, and they are added to the sub-plan memo. At the end of the second phase, all these newly encountered sub-plans are executed and their actual result sizes and the number of distinct values in their outputs are recorded in the sub-plan memo. This process is repeated until the optimizer goes through a phase in which it does not encounter any new sub-plans. The output query execution plan is the one chosen by this last phase. This plan is chosen using completely accurate result size and distinct value information obtained from the sub-plan memo.

The obvious drawback of query optimization by subplan memorization is that to optimize a single query, it is necessary to execute multiple sub-plans to determine their cost, thus found the truly "optimal" one, since this method incurs in a progressive and extensive exploration of the search space of alternative plans for executing a given query. Thus, query optimization will take a long time, and potentially much longer than the execution time of the query being optimized. This makes query optimization by sub-plan memorization too expensive for many queries.

Query optimization by sub-plan memorization is a suitable approach only for embedded queries that are optimized once and executed many times over. For this important class of queries, the potential for choosing more efficient query execution plans by using accurate result size and distinct value information makes the long query optimization times acceptable.

Subquery plan reuse [VaKi00]

This work propose an approach based on plan caching for minimizing the amount of memory consumed by the Dynamic Programming (DP) search strategy. DP is of the most popular search strategies (Section 2.2.1 explains in detail this algorithm), the inconvenient is that for queries with a large number of data sources is infeasible as the search space easily runs out of memory. Instead of fully generating the exponential search space this approach proposes to generate a part of the search space and reusing it for the remaining fraction, thus bringing about computational and memory savings, and getting a high quality query plan close to optimality.

DP generates a query plan in a bottom–up manner. First, it creates the unary operators for accessing each source. Second it generates the alternative plans for joining two sources and selects the optimal one; then it generates a plan for joining another source with the sub-plan generated in the precedent step. This approach proposes to store in a lattice the sub-plans generated at each step, such lattice is called DP lattice. Thus, the first level of the lattice comprises the operators for accessing the sources, in the second level the sub-plan joining two sources, in the third one the sub-plan joining three sources and so on. Thus, the level of the lattice corresponds to the number of sources that have been joined.

The central idea is to reduce the size of the set of sub plans *Plansi* for each level "i" in the DP lattice through sub plan reuse. For that is required to identify similar (sub) queries. A graph query representation is proposed, where relations being nodes and predicates being the edges between nodes. Hence, the problem is converted to a graph problem where the goal is to discover sub graph isomorphism internally, i.e. within a large graph. They define similar subgraphs *fS; S'g as a pair of sibgraphs having* the same structure and the same features i.e. each vertex, v in S should have a corresponding vertex v' in S' such that differences between table sizes and selectivity of the containing edges lie within the corresponding error bounds. The idea is to generate "sets" of similar sub-graphs so that the query plan generated for one representative subquery corresponding to the sub-graph can be re-used by all other subqueries indicated by the remaining sub-graphs in the similar sΣet.

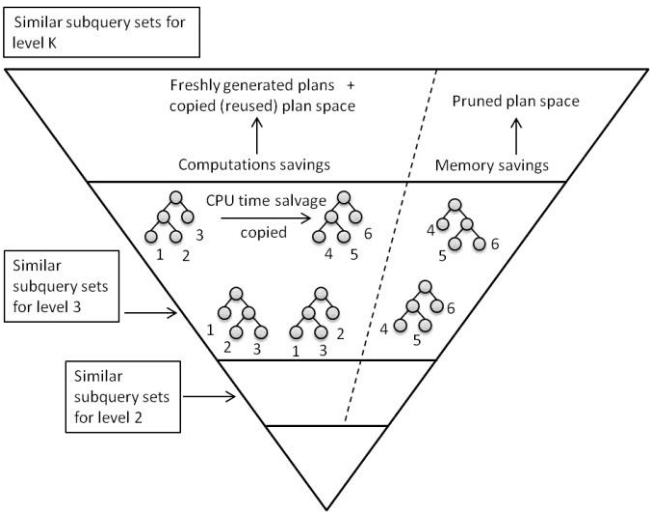

Figure 2.7 Search space generation in DP lattice

Figure 2.7 gives a pictorial representation of our scheme after the identification of similar subqueries. Similar subquery sets are fed to the DP lattice at each level. In the figure, during plan generation for level 3, the optimizer identifies from the similar subquery set that (1, 2, 3) is similar to (4, 5, 6) and hence the least cost plan of (1, 2, 3) is reused for (4, 5, 6). The plan for (4, 5, 6) is still constructed but in a light weight manner by imitating the join order, join methods and indexing

decisions at join node and scan node respectively, thus bringing upon computation savings by avoiding the conventional method of plan generation. Memory savings are brought about since the plans for the various join orders of (4,5,6) are not being generated. So this approach benefits from a mixture of CPU and memory savings.

The collection of sets of similar sub-graphs from all levels in the DP lattice is termed as the cover set of similar sub-graphs. Once the cover set of sub-graphs is generated, construction of query plans for each level in the DP lattice begins and because of exhaustive re-use of sub query plans among the similar subqueries identified by similar sub-graphs present in the cover set, memory savings can be achieved after constructing the query graph from the join predicates participating in the query.

The cover set of sub-graphs can be expressed as *n lev*=2 *Setslev* where *Setslev* =Σ*total i*=1 *Subgraphseti*. Here "total" indicates the total number of similar sub-graph sets at level "lev". *Subgraphseti* indicates the *ith* similar sub-graph set. The summation or total collection of all such sub-graph sets at level "lev" is represented by *Setslev*. The total collection of all such sub-graph sets over all levels gives the cover set of sub-graphs.

Towards empirical driven query optimization [VaKH09]

They propose a framework that collects statistics from the plans returned by the query optimizers and uses a distance function to select the least cost plan for a new query. In this work, the authors executed a large number of queries on a commercial DBMS and analyzed the structure (the order in which various operations are executed) and the cost of the optimizer plans. From their experiments they found that a large number of queries share the same join order. For instance, for a dataset, 8% of the join trees cover about 60% of the queries. They experiment with select-project-join queries randomly generated. For each query Q 2 Q, its optimizer plan, the optimizer join tree and the optimizer join order template were determined.

They characterize the optimal plans of a huge number of queries in form of join trees and join ordering templates. Notice that the *optimal plan* is different to the *optimizer plan*. They use this characterization, and the collection of statistics associated to those plans and templates for discovering the optimal join tree of a test query. A Join tree of a query plan contains the join operations as the non-leaf nodes and the base relations of the query as the leaf nodes (after applying the select conditions on the relations). On the other hand, a join order template of a join tree is obtained by replacing all its leaf nodes with arbitrary relations by traversing the tree in a particular order (for instance pre-order or post-order).

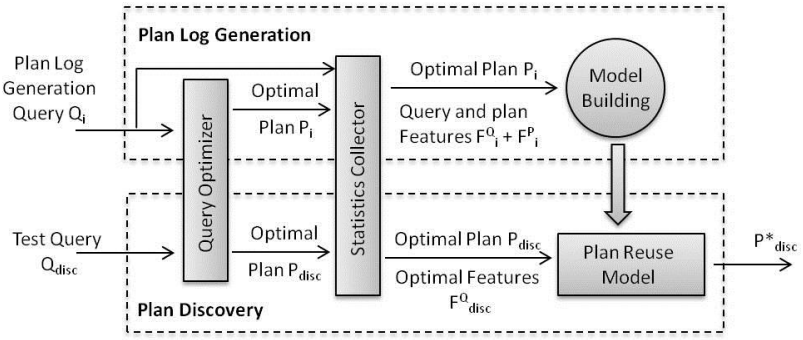

Figure 2.8 Empirically driven framework for QO

Figure 2.8 shows the framework for the empirically driven query optimizer. It can be divided into two phases: the plan logs generation phase and the plan discovery phase. During the plan logs

generation phase, each query Qi is submitted to the query optimizer of the DBMS to obtain its optimizer plan Pi. By invoking the statistics collector which takes the query Qi and the optimizer plan Pi as input, the query related statistics FQi and the optimizer plan related statistics F Pi are collected.

The query Qi, its optimizer Pi, the query features FQi and the plan features FPi constitute the plan reuse model L. Given a query to discover the optimal plan, Qdisc, the statistics collector module of the database can be used to collect the query related features FQdisc. Qdisc and FQdisc are given as input to the plan discovery algorithm which determines the optimal plan for Qdisc, P*disc. In order to compare the goodness of the discovered plan P*disc, Qdisc is also submitted to the query optimizer to get its actual optimizer plan Pdisc. The costs of Pdisc and P*disc can then be used to test the accuracy of the framework.

Discussion

In this section we have study query optimization techniques based on plan caching; we also have evoked their main strengths. Now, let us analyze why this solutions partially solve the motivation problem of this dissertation. Former, all these works relay on traditional cost-based optimization technique for the generation of plans that they latter reuse for the evaluation of further queries; some of these works propose techniques for the progressive improvement of plans. Traditional techniques for costs estimation are usually erroneous since query optimizer cost models are highly sensitive to data information errors (e.g. result size and distinct value estimates); or even works, such estimation it is not possible when the required data information is not available. In general, such cost functions estimate the execution time only.

Some of these works aims to find the truly optimal plan. They argue that classical query optimizer use fast but possibly inaccurate estimation techniques since they lead to choosing acceptable query execution plans, even though these plans may not be truly optimal. They execute sub-optimal plans generated by traditional optimizers (cost estimation based on inaccurate information) for gathering accurate information for cost estimation. For example, result size and distinct value estimates play a very important role in cost estimation. More accurate information about these two quantities typically results in more accurate cost estimates, which helps the optimizer choose more efficient query execution plans.

The problem is that discovering the truly optimal plan can be a very hard and time-consuming task. It is the correct approach for queries embedded in application programs, which comprise a large portion of the workloads handled by database systems. These queries are often optimized off-line to produce compiled query execution plans that are then used whenever the queries are executed. Optimization does not necessarily have to be fast since it is an offline process. Furthermore, these queries are typically executed frequently since the applications that they are part of are typically executed frequently. The in a static contexts where the truly optimal plan is unchangeable, otherwise this process takes a lot of time, what happens if the environment change, they never achieve to use it. They do not support even partially dynamic environments.

Table 2.2 summarizes the characteristics of the studied works considering the following dimensions: (i) the plans granularity since some works take as feedback complete reusable query plans, while others make use of sub-plans too (even isolated operators). (ii) coverage of plans / sub-plans, (iii) supplementary feedback apart from the query plan, (iv) the feedback representation structures, (v) feedback storage structure, (vi) the particular proposed techniques, and (vii) the moment when the optimization is executed.

Reference	Granularity	Cover	Feedback	Representation	Storage	Realization	Timing

		age					
PLASTIC [GPSH02, SeHa03]	Plan	high	Query	vector	clusters	queries classification	off-line/ on-line
Sub-plan memo [AbCh99]	sub-plan / operator	low	query / result size	SQL statements	hash table	truly optimal plan	off-line
Sub-plan reuse [VaKi00]	sub-plan	mediu m	query plan	graph	lattice	graph isomorphism	on-line
Empirical QO [VaKH09]	plan (join tree)	custo m	plan / statistics	plan logs as tuples	relational DBMS	SPJ queries	

Table 2.2 Query optimization approaches based on plan caching

2.4.3 Caching statistics and metrics

This section exposes representative works based on the collection, maintenance and reparation of query feedback. Some examples are statistics for the accurate estimation of query plans cost, and metrics of the amount of computational resources consumed during queries execution. These statistics and metrics are permanently materialized, thus, their representation and management are aspects that are considered in the related works, for example in novel optimization approaches as [GaJu12][TKKP09][ChRo94].

Some information correspond to cost function parameters (e.g. statistics); while another represent the real execution cost of query plans (e.g. measures of resources consumption), thus cost estimation formulas are not required. In general, approaches relying on statistics are useful for query optimization in distributed databases and multi-database systems. The autonomy in P2P systems can makes prohibitive the availability of such information, the monitoring of measures, and their usage to determine plans cost may be pertinent for these environments.

Some works associate the feedback exploitation with a learning process. A seminal approach is LEO, a learning-based optimizer; it monitors the execution of queries and compares the optimizer' estimates with the real once [SLMK01]. According to this comparison it computes adjustments to cost estimates and statistics allowing the optimizer to learn from its past mistakes. Thus, LEO proposes reparation of histograms; and other works propose caching techniques for learning sources response time, unknown operators' performance. All of them propose flexible and simple representation of the gathered information, and also highlight their concerning for its inexpensive storage and management (e.g. summarization of tables for minimizing memory usage and accelerating information exploitation). Details of such works are presented in next section. Finally a discussion about the differences and particular contributions these works concludes this section.

Repairing statistics [SLMK01]

The Learning Optimizer proposes as a comprehensive way to repair incorrect statistics and cardinality estimates of a query execution plan. By monitoring previously executed queries, LEO compares the optimizer's estimates with the real once at each step in a query plan, and computes adjustments to cost estimates and statistics that may be used during future query optimizations. This analysis can be done either on-line or off-line on a separate system and either incrementally or in batches. In this way, LEO introduces a feedback loop to query optimization that enhances the available information on the database where the most queries have occurred, allowing the optimizer to actually learn from its past mistakes.

Over time, LEO amasses experiential information that augments and adjusts the database statistics for the part of the database that enjoys the most user activity. Not only does this information enhance the quality of the optimizer's estimates, but it also can suggest where statistics gathering should be concentrated or even can supplant the need for statistics collection. This technique is general and can be applied to any operation in a QEP, including joins; derived results after several predicates have been applied, and even to DISTINCT and GROUP-BY operators. As shown by performance measurements on a 10 GB TPCH data set, the runtime overhead of LEO's monitoring is insignificant, whereas the potential benefit to response time from more accurate cardinality and cost estimates can be orders of magnitude.

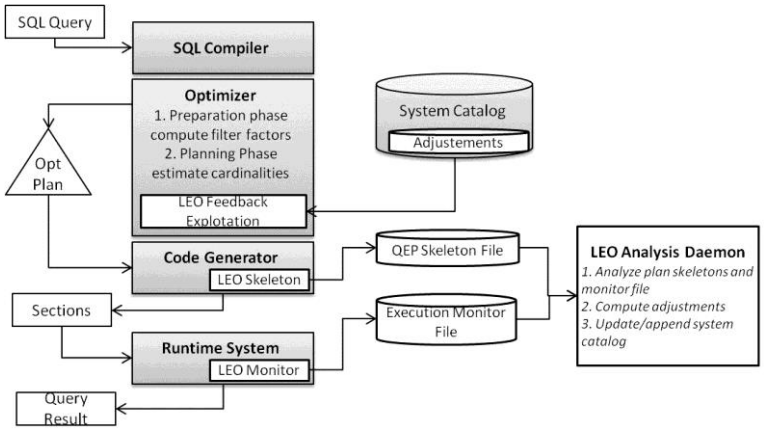

Figure 2.9 LEO architecture

Figure 2.9 shows how LEO is integrated into the architecture of DB2 [SLMK01].The left part of the figure shows the usual query processing flow with query compilation, QEP generation and optimization, code generation, and code execution. The gray shaded boxes show the changes made to regular query processing to enable LEO's feedback loop.

LEO is comprised of four components: a component to save the optimizer's plan, a monitoring component, an analysis component, and a feedback exploitation component. The analysis component is a standalone process that may be run separately from the DB2 server, and even on another system. The remaining three components are modifications to the DB2 server: plans are captured at compile time by an addition to the code generator, monitoring is part of the run-time system, and feedback exploitation is integrated into the optimizer.

The four components can operate independently, but form a consecutive sequence that constitutes a continuous learning mechanism by incrementally capturing plans, monitoring their execution, analyzing the monitor output, and computing adjustments to be used for future query compilations.

During the learning mechanism the LEO components interact as follows: for any query, the code generator dumps essential information about the chosen QEP (a plan "skeleton") into a special file that is later used by the LEO analysis daemon. In the same way, the runtime system provides monitored information about cardinalities for each operator in the QEP. Analyzing the plan skeletons and the runtime monitoring information, the LEO analysis daemon computes adjustments that are stored in the system catalog. The exploitation component closes the feedback loop by using the adjustments in the system catalog to provide adjustments to the query optimizer's cardinality estimates.

Learning response time for data sources [GRZZ00]

This work presents a Multi-Dimensional Table (MDT) that is supplied with feedback from query execution to learn the response time (delay) of a particular web source. Such knowledge (delay of a web source) is used by a scrambling algorithm; the objective is to predict the delay moments when accessing a source and use them as critical points to scramble (modify) a query plan. The scrambling algorithm also aims to hide the expected delay by computing some other part or the query plan unaffected by such delay.

The structure of the MDT is determined by (i) a set of dimensions, (ii) the ordering of such dimensions, and (iii) the ranges / scales of the dimensions. The MDT proposed in [GRZZ00], has three dimensions:

The *Day* of the week: This dimension has a range of seven days, and the minimum scale for this dimension was chosen as one day

- The *Time* of the day: This dimensions has a range of 24 hours, and the minimum scale for this dimension was chosen as one hour
- The *Quantity* of data that is transferred: This dimension does not have fixed range. Based on our experimental data, the minimum scale that we chose was multiples of 100 kilobytes, and the range was from [0 to 800 kilobytes]

The dimensions of the MDT are significant for estimating the response time of a particular source. Those dimensions were selected by applying statistical tests to experiences that consisted in collecting data from different web sources. The ordering of dimensions is critical as is explained in the precedent of the section.

The ordering of dimensions and scale of the dimensions are used to tune the MDT. Three other features are also used: (i) the allowed deviation of the error in response time, the value is specified for each dimension; (ii) the precision for each dimension; the smaller range of the individual cell, the greater is the precision; and (iii) the confidence associated to each dimension, the confidence range is [0.0 – 1.0].

The initial MDT consists of one cell, and the range of its dimensions corresponds to the range described above. During the learning process, each query feedback *qfb* is represented by a value for Time, Day, Quantity, and response time *QryRT* obtained after executing a query. Let us say that several queries accessing the source sw are executed, the obtained query feedback is used to predict the response time *PredRT* of sw. To the *PredRT* of each cell corresponds a degree of confidence *predConf*.

The MDT is tuned to achieve *PredRT* as accurate as possible. For tuning the MDT, first the cell whose dimensions match with the *qfb* is identified. Then, the learning process consists in deciding, based on the query response time *QryRT* of *qfb*, if either, split the cell into two or more cells, or to adjust the concerning *PredRT* and *predConf*.

For taking such decision, the error of the current *qfb* is compared with the allowed deviation for each dimension. Such comparison is carried out according to the order of dimensions in the MDT. If the error is greater than the deviation, the cell is split on that dimension (for simplicity the split is in two cells and the cells has equal range). Only one of the splits cells now matches the dimensions of the *qfb*. The new *PredRT* for the new cell is set to *QryRT*, and the new *predConf* is 0; the respective values of the other cell remain unchanged. Then, the learning algorithm is called recursively for each subsequent dimension. If the error is lower than the deviation the *PredRT* and *predConf* of the cell are

adjusted to reflect the *qfb*. We refer to the reader to [GRZZ00] for details in the mathematic formulate for the adjustments of predicted response time.

The minimum scale of each of the dimensions determines the (final) MDT structure, when it can no longer split into more cells or any dimension. For example, when the Time dimension has been split into 24 cells, each with a one hour range, then no further splitting on that dimension is possible. An example of a MDT structure, at some intermediate point, is in Figure 2.10.

Monday - Friday								Saturday
8am – 2pm				2pm – 8pm			8am–8pm	12am–12am
<200K	200K-400K	400k-600k	> 600K	< 200K	200K-400K	> 400K	0 MAX	0 MAX

Figure 2.10 Example of an intermediate MDT

The MDT approach has some advantages over other learning based techniques, as regression techniques or neural networks. One advantage is the simplicity of the MDT prediction model. The second one is the flexibility provided to manipulate a number of parameters that control learning. The third advantage is that while the chosen dimensions may reflect the effects of sources and network usage, a (lack of) confidence reflects the unpredictable nature of predictions for web sources.

Self-tuning histograms [AbCh99b]

In [AbCh99b] the authors present a query feedback loop, in which actual cardinalities gleaned from executing a query are used to correct histograms. Although similar in structure to traditional histograms, these histograms infer data distributions not by examining the data or a sample thereof, but by using feedback from the query execution engine about the actual selectivity of range selection operators to progressively refine the histogram. These histograms are called *self-tuning histograms* or *ST-histograms* for short.

Since the cost of building and maintaining *ST-histograms* is independent of the data size, such histograms provide a remarkably inexpensive way to construct histograms for large data sets with little up-front costs. *ST-histograms* are particularly attractive as an alternative to multi-dimensional traditional histograms that capture dependencies between attributes but are prohibitively expensive to build and maintain.

The construction of a *ST-histogram* starts with an initial histogram built with whatever information we have about the distribution of the histogram attribute(s). For example, it is constructed an initial bi-dimensional histogram from two existing mono-dimensional histograms assuming independence of the attributes. As queries are issued on the database, the query optimizer uses the histogram to estimate selectivity in the process of choosing query execution plans.

Whenever a plan is executed, the query execution engine can count the number of tuples produced by each operator. The core of the proposed approach is to use this “free” feedback information to refine the histogram. Whenever a query uses the histogram, we compare the estimated selectivity to the actual selectivity and refine the histogram based on the selectivity estimation error. This incremental refinement progressively reduces estimation errors and leads to a histogram that is accurate for similar workloads. A ST-histogram can be refined *on-line* or *off-line*.

Catching statistics [ACPS96]

One of the main contributions of this work is a cost-based optimization technique based on statistics caching. The cached statistics correspond to actual calls to data sources; the cost of query plans is

estimated basing on these statistics. [ACPS96] investigates the design of statistics cache and presents a mechanisms for its effective use. The interest of this proposal is to deal with the difficulty, in some cases such as in HERMES project [SABE94], to obtain accurate cost estimates of query plans because the involving of operators with unknown behavior and performance.

In HERMES, external programs are referred as domains and they are viewed as black boxes that allow certain operations to be performed by outside sources. These operations are executed via domain calls of the form `d:f`, where `d` is the name of the domain and f is the name of a function corresponding to a predefined operation that can be performed in this domain. Domain calls are expressed uniformly in HERMES with the help of a special predicate of the form in `(X,d:f(Args))`, which is read as: "Execute the domain function d:f on arguments `Args` and return the set of results in variable `X`".

The statistics cache is stored and managed by the mediator. It is a database that records cost information about domain calls as they get executed by the mediator. In the simplest version for each domain call it contains a triple of the form (`domain_call`, `cost_vector`, `recird_time`), where `record_time` is the actual time that a call was recorded in the database. Hence, the cost database consists of tables for different domain call, where the columns correspond to the time to compute the first answer, time to compute all the answers, the cardinality of answer and the arguments to which these values correspond to. Figure 2.11 shows some tables by example.

(T16) d1:pb_f (A)

A	Card	T_A
a	4	2.00
a	5	2.20
c	8	2.80
c	8	2.84

(T17) d1:p_bb (A,B)

A	B	Card	T_A
a	g	0	2.50
a	d	1	2.70
c	g	1	2.68
c	d	0	2.65

(T18) d2:q_bf (B)

B	Card	T_A
g	40	50.0
g	41	51.0
g	39	49.0
d	30	48.0
d	35	42.0

(T19) d2:q_ff ()

Card	T_A
100	50.0
95	48.0
105	52.0

Figure 2.11 Tables in the cost vector database

As an example, consider a mediator (M1), a query (Q7) and two candidate plans (P1) and (P2). In order to estimate the cost of the two plans it is precise to estimate the cost of the domain calls `d1:p_bf`, `d1:p_bb`, `d2:q_bf` and `d2:q_ff` that appear in the two plans. It is assumed that tables of the previous figure describe the total execution time and the cardinalities of the listed queries. The same value for an argument may appear more than once in the tables corresponding to different calls. Then, it is possible to estimate the cost of a domain call e.g., `d1:p_bd (a)`, for the execution time to all the answers, by taking the average of the two entities in the table (T16, namely 2.00 and 2.20 to get 2.10. Also it is possible to estimate the cost of a domain call here one or more parameters are unknown. For example, the average i.e. (2.00 + 2.20 + 2.80 + 2.84) /4.

Though tables in Figure 2.12 have the necessary information, there are two important problems regarding their use and maintenance. First, full detailed statistics information represents a heavily bounden on storage. Second, expensive aggregation functions reputedly applied, like the average function of the previous example, thus the time for calculating the cost may be prohibitively long. For solving such problems they propose on-line summarizations of the statistic information stored in the cost vector database. Summarization has a dual purpose: reduce the storage space needed

for statistics and accelerate their exploitation. The challenge is summarize such tables without losing any information, thus they call this summarization loss-less.

(T20)

d1:p_bf (A)			
A	Card	T_A	l
a	4.5	2.10	2
c	8.0	2.82	2

(T21)

d2:q_ff ()		
Card	T_A	l
100	50.0	3

Figure 2.12 Table of summarization of statistics

For example the summarization of table (T16) is in table (T20) in Figure 2.11. In this case the tuples `A='a'` (`or A='c'`) have been aggregated into a single tuple. The 1 attributes indicate the number of original table tuples that correspond to the summarized table tuples. In general such summarization process consists in: (i) split the attributes of every statistics table into a set of dimensions that consists of all attributes of the corresponding call, and the set of metrics that reflects the response time of the call, and the cardinality of the results. (ii) For all tuples that have identical values `d1`, `d2`,...,`dn` on the dimension attributes, aggregate the metrics attributes into a single pair average response time and average cardinality and create a single tuple, where 1 is the number of original table tuples that have been aggregated into the specific tuple. Supplementary summarization actions correspond to drop the attributed that will never been instantiated, and the identification of access patterns for the tables and decide which ones are need very frequently and drop those that are not accessed very often.

Discussion

Several techniques for estimating result sizes and distinct values have been proposed in the literature. One technique for estimating result sizes is sampling the data at query optimization time [LiNS90] [GaJu12]. The main disadvantage of sampling is the overhead it adds to query optimization. Furthermore, sampling cannot be used to accurately estimate the number of distinct values of an attribute [ChMN98]. Sampling is more useful for other applications such as building histograms or approximate query processing.

Another technique for estimating result sizes is histograms. Histograms for database systems were introduced in [Koo80], and most commercial database systems now use them for result size estimation. Although one-dimensional equi-depth histograms are used in most systems, more accurate histograms have been proposed in [PHIS96]. In [PoIo97], the techniques of [PHIS96]are extended to multiple dimensions. A novel approach for building histograms based on wavelets is presented in [MaVW98]. Efficient algorithms for constructing optimal histograms using dynamic programming, and for approximating these optimal histograms using heuristics, are presented in [JPKS00] [ChRo94] [SHMK06], they are particularly based on query feedback for histograms construction and/or histograms self-tuning. Histograms, by their nature, only capture an *approximation* of the data distribution, and they incur varying degrees of estimation errors.

Table 2.1 summarizes the studied works based on caching of statistics and measures considering: (i) the problem, (ii) the feedback, (iii) when such monitoring is achieved, and (iv) at which moment. Moreover, it also includes (v) the monitored subject, (vi) the storage structure, and (vii) the issues that the process aims to learn.

References	Subject	feedback	when	Where	Monitoring	Storage structure	Learning
LEO [SLMK01]	statistics repairing	statistics	on-line / off-line	partially coupled	any operator of QEP		cost estimation error
MDT [GRZZ00]	query scrambling	Delay	on-line	independent	sources access	self-tuning multi-dimensional tables	sources response time
Histogram ST [AbCh99b]	histograms correction	attribute values	on-line / off - line	independent	result size and selectivity factor		infer data distribution
Caching statistics [ACPS96]	performance of unknown operators	response time / exe time / cardinality /		independent	calls to unknown operators	relational table	performance of unknown operators

Table 2.3 Query optimization approaches based on statistics / metrics caching

2.5 CONCLUSIONS

In this chapter, we presented different query optimization techniques in distributed data systems. Former, we addressed the basics of distributed query optimization, the generic techniques and the characteristics and importance of metadata for traditional query optimization. We focus query optimization approaches using query feedback to face the scarcity of metadata (lack of a global view of existent data sources, thus unavailable information on data) in distributed data systems. We analyze the different manner to use query feedback optimizing a query. According to this analysis, we propose to main classifications: adaptive approaches and learning approaches.

On the one hand, works based on adaptive approaches collect and utilizes the feedback as soon as a given query Q is executed. Such feedback is temporary retained, only while it is required for the re-optimization of Q; it is not used for the optimization of other queries. Adaptive approaches monitor statistic information, availability and consumption of computational resources. On the other hand, works based on learning store query feedback permanently (some of them include reparation mechanisms for feedback improvement). Their main concerns are: feedback representation, storage structures, and management and exploitation mechanisms. We divide such works on those caching statistics and measures, and those caching plans.

The study of such approaches confirms that most of proposed works address the optimization problem in tightly integrated systems, which the control of data access and query processing is centralized in a single system element (e.g. distributed databases) or partially distributed (multi-data base systems). Not quite an effort has been done for applying such optimization techniques (or variations of) to fully autonomous environments (e.g. P2P like systems) where there each entity of the system is responsible for optimizing the queries that it receives. These works mainly concern the optimization of execution time and response time only.

Adaptive approaches consecrate their efforts on the optimization of queries in highly dynamic systems, where only real measures serve for an accurate prediction of (close to) optimal plans. The main disadvantage is that multiple re-optimizations of a query plan may take place during the query execution, the resources consumption and time employed for this re-optimization may cause important optimization overhead. Another disadvantage is the low coverage of query feedback, since it is useful for the optimization of a single query.

The learning approach: (i) collect and repair statistics, or (ii) take sub-optimal plans and try to progressively improve them. Statistics serve for estimation of plans cost using traditional optimization

techniques. Plans are generated using traditional cost-based optimization techniques. In contrast to the adaptive approach, they consider that the execution environment does not change very often, or it does not change at all. Such approach is suitable for the evaluation of frequent queries aiming to retain feedback useful for the evaluation of several further queries.

From this study we highlighted useful aspects to consider in our proposal:

- Avoid unnecessary processes e.g. extensive exploration of search space at one-time when it is not possible to have accurate estimation of plans cost. Instead it is possible to apply other techniques, e.g. randomized or genetic algorithms, which do not obtain the optimal, but avoid the worst plans. These techniques are based on heuristics and random decisions. A first batch of plans may be obtained using these techniques, and then be progressively improved.
- Use query feedback that, even if it is not valid for environments that change very frequently, it is useful for systems where the frequency of queries is reasonably higher than the environmental changes in such a way that there is time for the optimizer to learn and evolve with the environment.
- Exploit different query feedback (other than classical metadata).
- Allow independence of optimization responsibility due to the lack of a global view of the system.
- Consider representation and storage structures that facilitate exploitation and management of feedback.
- Envisage exploitation and management mechanisms simple as possible for minimizing the consumption of computational resources.

3. CBR FOR QUERY OPTIMIZATION

Our optimization approach integrates the learning Case Based Reasoning (CBR) paradigm in the query processing. This chapter presents the general principle of our approach using an illustrative application scenario in Section 3.1. Also, it details the data model, and the distribution of data that we consider in Section 3.2. We propose an approach for the optimization of declarative global queries, for explaining our approach we consider queries expressed in a SQL-like query language that includes clauses for specifying the location of data to be queried; Section 3.3 describes the characteristics of such a language.

Section 3.4 describes a scheduling to supply the queries to be optimized. Section 3.5 exposes in deep our CBR query optimization principle; it gives an overview of the CBR foundations (i.e. concepts and reasoning mechanisms), and details the adaptation of such elements to the query optimization process. Section 3.6 discuses the main differences between the classical query optimization principle and our learning-based query optimization proposal. Section 3.7 presents the conclusions of this chapter.

3.1 GENERAL PRINCIPLE

To illustrate the general principle of our learning-based query optimization approach, let us consider a virtual game application where players are owners of one or more avatars. The objective of the game may be social interaction or team fighting; this does not affect the understanding of our proposal. Such application runs on a distributed environment that interconnects –trough wireless technologies– devices (i.e. system nodes) e.g. smartphones, tablets, etc. that can be heterogeneous, autonomous, either static or mobile and that present physical limitations (e.g. energy, memory and processing).

Avatars interact in a virtual world that is divided in areas; an avatar is located within a single area at a time. Every node in the system has information on its own avatars and their neighbors (avatars located in the same area). The application data are stored as *Itemset* data structures of the form: *POSITIONS (Avatar avatar{key}, Int area, NodeID owner)*, as shown in Figure 3.1. For example, the Itemset at node *D* has two items i_1: *<Grey, 2, D>* and i_2: *<Blue, 2, I>*. Thus, the node is the owner of the *Grey* avatar which is in the area 2; also the *Blue* avatar is in the same area but owned by node *I* in this example. Such a *POSITIONS* Itemset is actually a fragment of a virtual global Itemset that maintains all avatars information.

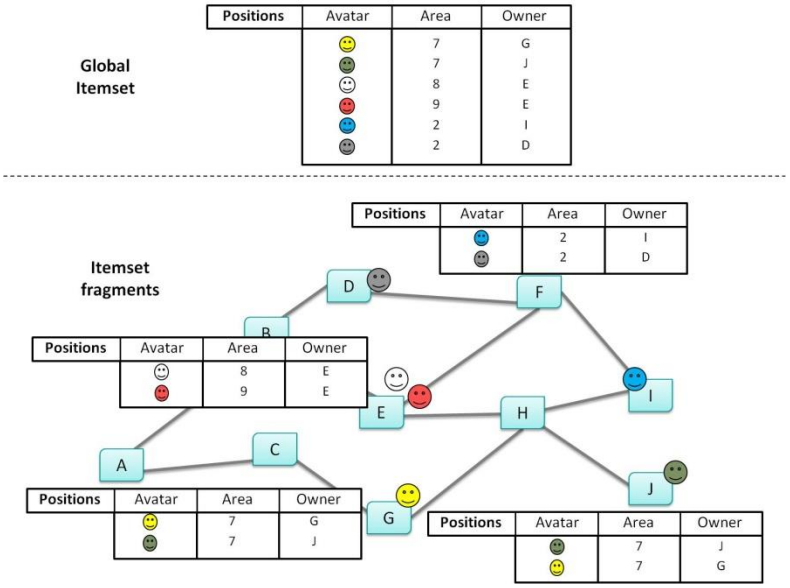

Figure 3.1 Virtual world game

In this example, the global Itemset is horizontally partitioned. Actually, we consider the optimization of queries on distributed data systems that hold a global schema, and as a consequence, where data distribution is according to a horizontal partitioning. This means that a global *Itemset* is physically distributed over several (maybe all) nodes, and that every node stores a (local) *Itemset* fragment that respects a global schema. Itemset fragments may also be stored (i.e. replicated) in one or more nodes. In contrast, we do not consider the existence of a global data allocation schema, which is used in database systems for maintaining information about data storage (e.g. distribution of data values among the system nodes, size of fragments at every node, among other statistics).

Avatar actions are carried out trough declarative global queries. A global query is expressed in terms of global Itemsets (physically stored in distributed fashion among system nodes as mentioned before). A global query can be posed at any node and the system will globally execute it, this means that data from local / remote nodes is accessed, exchanged and computed to construct a complete query result. An example of global query to select all avatars in the virtual world is shown below. It is expressed in a declarative SQL-like language. Such query is global since it considers the global Itemset *POSITIONS*.

Example 3.1 - `SELECT Avatar FROM Positions;`

Alternatively, the application may also consider queries / updates over Itemset fragments at specific nodes. For such purpose we consider queries expressed in the Declarative Location Aware Query Language (DLAQL) [ABCD12b]. DLAQL is a SQL-like language that includes clauses to specify the nodes where data have to be selected, inserted, deleted or updated. These nodes are specified using the node identifiers, subqueries that return a list of node identifiers or keywords provided by the language. For instance, a DLAQL query for selecting the avatars located at the same area that the avatar of some user is shown in the Example 3.2.

Example 3.2 - `SELECT Avatar FROM Positions SCOPE IS SELF;`

For evaluating the previous query is enough to consider the local Itemset fragment only, since the user posed the query at the node (e.g. personal tablet) that comprises information of its own avatar(s) and its avatar's neighbors. Thus, the query should be executed at the current node. In the

previous example, the clause SCOPE IS is preceded by the node(s) that must be queried; the keyword SELF specifies that the query considers the fragment stored in the current node only. The SCOPE IS cause is not mandatory; by default it is GLOBAL as in Example 3.1. The DLAQL language also provides clauses to specify the nodes to carrying out update operations. Thus, data distribution is application-driven since the application supports data location functionality; the system executes insert, delete and update operations at the nodes specified in the update DLAQL expressions. Section 3.3 details the DLAQL language and presents illustrative examples.

A node receives a query from the application or from another node. The first processing step consists in validate the query syntax and semantic. If the query comprises subqueries, the next step is scheduling such subqueries to be optimized independently and in a specific order. Basically, subqueries are optimized and executed first; then they are replaced by their resulting values to continue the outer query processing (we refer as outer query to the query including subqueries).

An optimal query plan has to be generated for evaluating each (sub) query; this is the goal of the query optimization process. To be evaluated, incoming global queries are to be rewritten on subqueries expressed in terms of local/distant Itemset fragments. For example a global query Q may be rewritten as $Q = (SQ_L \cup SQ_D)$, where SQ_L is q subquery executed locally, and SQ_D a subquery executed at some remote node(s); the partial results of such subqueries compose the final query result. We consider that a query plan is a tree of physical operators for the manipulation and communication of data, but also for the exchange of (sub) queries. Chapter 4 details the representation of a query plan and exposes the full suit of data operators that a query plan may include.

Thus, during the optimization process is essential to decide: (i) the parts of the query (i.e. subqueries) to be executed locally and those to be sent through the network to be executed at some remote node(s), (ii) the operators to be executed locally for combining the partial subquery results, and (iii) the execution strategies for computing the operators.

We propose an optimization approach based on feedback gathered from the execution of queries. The final goal is to learn suitable plans for the evaluation of global queries optimizing different (customizable) objectives. The query feedback includes the optimization decisions made for executing the query, and the computational resources (e.g. memory, energy, CPU) consumed during its execution. Our optimization approach gathers, maintains and exploit such a query feedback by following the Case-Based Reasoning (CBR) principle, which states that problem solving can be improved by learning the quality of solutions used to solve past similar problems [MCBD12].

We assume that the nodes in the system are instrumented for monitoring the consumption of computational resources during the evaluation of queries. Let us recall that several nodes participate for the evaluation of a global query. Every node measures the computational resources that it consumes by executing the part of the query (i.e. subqueries) that it concerns. For having a global view of the consumed resources, each remote node sends its own measures together with the query partial results to the requesting node. This allows integrating local and distant measures, thus, query-cases store global measures and not only those concerning to a single (local) node.

To illustrate the collection of query feedback we use a more interesting query example (i.e. join query). For this purpose we extend the representation of our virtual world; in this case, information about each area is managed by a server. A client sends a query to the server that manages the area where a given avatar is located. Each client stores information concerning its own avatars. Therefore, the global schema is as follows: *POSITIONS(Avatar avatar, Coordinate position, NodeID*

ownerID), SERVERS(Int area, String serverID). The query Example 3.3 selects the avatars located in the areas 2 and 7.

Example 3.3 -
```
SELECT Avatar
FROM POSITIONS as p, SERVERS as s
WHERE s.Area IN (7, 2) AND p.Position = s.Area;
```

Figure 3.2 shows a (possible) plan P generated in a node, let's say the node G, for executing the query Example 3.3. Such a plan includes an operator for accessing data from a distant server, and operators for accessing local and distant fragments of the Itemset *POSITINS*. The operators for accessing distant data fragments correspond to messages that comprise subqueries (e.g. the subquery SQ_3) to be executed at remote nodes.

In our example, we measure memory consumption; however other measures can be considered according to the system instrumentation for monitoring the execution of queries. Moreover, such measures may be used as parameters of more complex cost functions. The node G sends the subqueries to its neighbors, which in turn continue the propagation of subqueries. Then it receives the partial results from its neighbors, for example from node H and the aggregated measures of resources consumed by other remote nodes (e.g. node I that retrieved the avatars at the area 2).

The amount of memory consumed for executing the global query corresponds to the aggregation of memory consumed at each participant note (e.g. node I - 80kB, node H - 25kB and node G – 105kB). The query-case at node G from executing the global query comprises a description of the query, the query execution plan and the global measure of memory (i.e. 180 kB) consumed for evaluating such a query using the proposed plan.

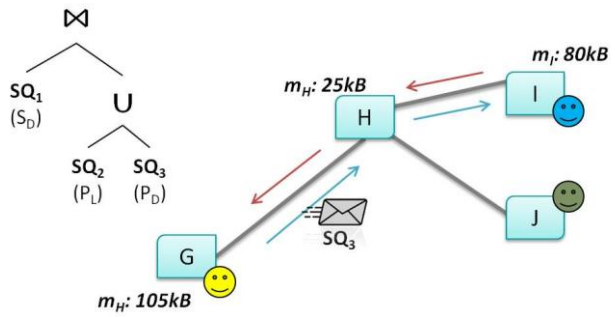

Figure 3.2 Collection of feedback from evaluating a global query

A node comprises a local repository (i.e. casebase) for storing and managing the query-cases. The optimization process exploits the query-cases that it has learned from the evaluation of similar previous queries. While there are not enough query-cases to be exploited, the optimization process generates query plans considering classical optimization heuristics and taking some random decision with the objective to try new query plans and learn about their global cost. The efficient management (e.g. delete of useless query-cases) of query-cases is fundamental in order to preserve within the casebase useful cases only.

In our approach, the optimization of queries is de-centralized, each node in the system is responsible for optimizing the (sub) queries that it receives. Since each node participates for executing different (sub) queries, its learning is different and independent from each other. For example, node G received the original user query and learns about its execution. However, node H also participates for

the global execution of such a query by propagating some subqueries through the network, and node I learns from the subquery that it process.

3.2 DATA MODEL AND DISTRIBUTION

This section presents the assumptions we made concerning the representation of data and its distribution through the system. We consider generic Itemset data structures, which are described in Section 3.2.1. The considered distribution of data is discussed in Section 3.2.2.

3.2.1 Data model

Our data representation is based on the *Itemset* data model, which is a generic and minimalist data model originally proposed by G. Grafe in the Volcano System project [GrMc93]. In such model, the unit of data manipulation is an *item*. An item is composed of a sequence of data *values* that belong to predefined data types (e.g. integer, float, data, etc.).

An *Itemset* is defined by its name, and a description (i.e. schema). Such description is a sequence of attributes of the form <*name, data type*>. A subset of the *attributes* that composes the Itemset description is discriminatory (i.e. key attributes). Thus, two items in the same Itemset have different *values* while corresponding to key attributes. An example of *Itemset* is the Positions table in our application scenario (see Figure 3.2).

3.2.2 Data distribution

We suppose that each application is aware of the set of Itemset definitions that it manipulates. The nodes of the system that wish to cooperate for caring out a task, e.g. running a common application, agree a common schema description for data sharing. Thus, we made the strong assumption that the system counts on with a general description of data (i.e. global schema) for a specific application. Each application can add attributes to the shared Itemsets for its internal needs. Therefore, queries of a specific application are expressed according to the corresponding common schema.

Our approach considers that every node in the system stores data corresponding to a subset of the global Itemsets that a specific application concerns. Such a global Itemset is physically stored in fragments (and probably replicated) distributed in several nodes. We suppose that there is horizontal fragmentation only. Thus, if a node stores part of a global Itemset, actually it stores a subset of items such that, an item comprises a value for each attribute that the Itemset definition includes.

We make no assumptions about where Itemset fragments are distributed over the system nodes. Thus, in general data distribution is *application-driven*: applications decide on which node(s) items have to be stored. There is one exception to this rule: if a key attribute corresponds to the node identifier each node stores the items having the node identifier that it concerns. For example, let us say that the global *Itemset* that comprise the physical communication links between nodes is described as follows: *Link(LocalID{key}, NodeID neighbor {key}).* In this case, each node stores the list of its own neighbors.

3.3 DATA LOCATION AWARE QUERY LANGUAGE

This section presents the Data Location Aware Query Language (DLAQL) for expressing declarative queries and updates over global schemas. The DLAQL language was specified in the context of the UBIQUEST project [ABCD12a]. It is extends part of the well-known SQL2 data manipulation language with clauses and keywords to indicate the nodes in the system where data has to be selected, inserted, deleted or updated.

It includes the expression of classical SELECT-FROM-WHERE, INSERT-INTO, UPDATE-SET-WHERE, DELETE-FROM-WHERE queries, using nested sub-queries, aggregation functions, arithmetic expressions, and selection, join and union operations. However, DLAQL extends only a subset of SQL2 functions since it does not include group by/having clauses, nested queries except in the WHERE clause, synchronous sub-queries. Also, it does not consider EXISTS and UNIQUE condition operators.

In the DLAQL language, the nodes for selecting/updating data are specified using the node identifiers or keywords provided by the language (i.e. extensional specification), or by subqueries that return node identifiers (i.e. intentional specification). Next subsections detail the clauses for queries and updates, introduce the data location clauses, and present some examples of extensional and intentional specifications of data location. Thanks to this language facility, it is not required to known the distribution of data (i.e. count on with a data distribution schema); data location is thus application driven. The syntax of the DLAQL is presented in Annex A.

3.3.1 Scope of a query

A DLAQL expression is defined considering a global schema of Itemsets. It is evaluated considering global Itemsets (union of itemset fragments stored at different system nodes). DLAQL offers the possibility to restrict the set of nodes for selecting data using the SCOPE IS clause in query expressions of the form SELECT-FROM-WHERE.

The SCOPE IS clause may be specified for any global itemset included in the FROM clause of the query. It takes as argument a list of NodeID values, the keyword SELF (meaning the locally stored itemset fragment), or subqueries returning NodeID values. The SCOPE IS clause is not mandatory. In that case, the scope is GLOBAL for the dataset and the fragments from all nodes are selected when evaluating the query. Thus the SCOPE IS expressions can be as follows:

- SCOPE IS SELF: only local dataset fragment is used.
- SCOPE IS ⟨*expr*⟩: the expression ⟨*expr*⟩ may be a list of NodeID values or a query expression returning a list of NodeID values.

When the SCOPE IS clause is followed by the keyword SELF or by an expression ⟨*expr*⟩ corresponding to a list of nodes the scope is specified by extension, as shown in the query **Example 3.2** of the precedent section. In contrast, the query **Example 3.5** presents a query where the scope of the source corresponds to the node identifier resulting from the evaluation of a subquery; thus, the scope is specified by intention.

3.3.2 Location of data

The STORE ON clause is used in INSERT expressions to indicate where a new item has to be inserted. The STORED ON clause is used DELETE and UPDATE expressions to indicate the nodes where data must be eliminated or updated. Such nodes may be specified by intention or by extension, as for the selection queries in the president section.

For instance, let us now assume that the *Yellow* avatar owned by node G, is moved from area 7 (where avatar *Green* owned by node J is localized) to area 8 where the Red avatar (node E) is localized. The movement is coded by several updates executed at node G (owner of *Yellow*) for cleaning area 7, changing the Area attribute of the avatar and finally for storing the new area exploration.

The query in **Example 3.4.a** deletes the Yellow avatar from the local POSITIONS fragments. In this example the clause STORED ON is specified by extension using the keyword SELF, which indicates that the delete operation is local, it must delete items from the local Itemset fragment only.

Example 3.4.a -
```
DELETE FROM Positions
WHERE Area = (SELECT Area
              FROM Positions SCOPE IS SELF
              WHERE Avatar = 'Yellow')
AND Area NOT IN (SELECT Area
              FROM Positions SCOPE IS SELF
              WHERE Avatar <> 'Yellow'
              AND Owner = SELF)
STORED ON SELF;
```

A second operation to accomplish the movement of the Yellow avatar is shown in query as the query **Example 3.4.b**. It inserts the new item ('*MyAvatar*', 8, SELF) in the Positions *Itemset* stored at the local node SELF (the node that owns the Yellow avatar), and in the Positions *Itemset* of any node owning an avatar at area 8. In other words, such operation adds the Yellow avatar as a neighbor of any other avatar that is located at area 8.

Example 3.4.b -
```
INSERT INTO Positions
VALUES ( 'MyAvatar', 8, SELF)
STORE ON SELF, (
              SELECT Owner
              FROM Positions
              WHERE Area = 8);
```

When the STORE ON clause is not specified the semantic of the query depends on the type of data manipulation operator. If it is:

- An INSERT: the new items are inserted in the local fragment of the Itemset.
- A DELETE: items are deleted from the global Itemset (i.e. from all fragments).
- An UPDATE: items are updated in place (i.e. on the node where items to be updated are stored).

3.4 QUERIES SCHEDULING

A query may comprise one or more subqueries. Such subqueries may be explicitly specified by the user, or may result from some pre optimization process. Previous to the optimization process, a (user) global query is parsed, validated and possibly rewritten to simplify the latter processing steps (i.e. optimization and evaluation). These pre optimization processes are not the main concern of this work, but, the optimization of subqueries resulting from such processes. A query comprising subqueries is not optimized all at once; the optimization process starts by optimizing its subqueries according to a specific order. It is important to clarifying the scheduling for dispatching queries to the optimization process.

According to the DLAQL query language, subqueries may be found within the WHERE clause comparing an attribute with a set of values (e.g. Area IN <subquery>) or with a specific value e.g. (Area = <subquery>) resulting from evaluating a subquery as shown in the query below. The clause STORE ON and STORED ON may also comprise subqueries as the queries in Example 3.4 and Example Z respectively. Finally, the INSERT INTO statement may concern explicit values, but also subquery result values.

Example 3.5 -
```
SELECT *
FROM Positions SCOPE IS SELF
WHERE Area = (SELECT P.Area
              FROM Positions P
              WHERE P.Avatar =  'MyAvatar);
```

Each of such queries is optimized independently. Our approach includes a scheduling process for optimizing the most imbricated subqueries first. For example, in the query Example 3.5 the user asks for the avatars that are located in the same area than its avatar. For answering that query (Q), first is precise to process the subquery for obtaining the area where the avatar of the user is located. Thus, the subquery (SQ) it is optimized and evaluated. Once SQ has been executed, it is replaced by its resulting value in at the WHERE clause in Q; then the query Q is ready to be optimized and so on.

3.5 QUERY OPTIMIZATION

Our learning-based approach optimizes global queries on large distributed data systems. It focuses on the difficulty of optimizing queries when there is no complete information on data because of the distribution and autonomy system resources. It also addresses the necessity of customizable optimization objectives according to the requirements of different applications / devices.

The literature has proposed cost based and heuristic based seminal optimization approaches. The former one relies on data information (e.g. resources availability, data distribution, data value statistics, network topology, etc.) difficult to count on in large distributed data systems. Such lack of information on data leads to approaches based on heuristics only, which avoid the worst query execution plans, but still risk to be really far from the optimal plan. Typically, such approaches consider minimizing the execution time as the only optimization objective.

Optimization approaches have proposed the use of query feedback to overcome the lack of information on data. This, they collect different kind of feedback (e.g. data statistics, system parameters, query plans) from queries execution, and propose mechanisms for their management and exploitation. However, they have been mostly applied on distributed data systems that trend to

centralize the optimization process. Moreover, most of them still use classical metadata for plans cost computation, as discussed in Chapter 2.

Our query optimization approach relies on query feedback (i.e. measures of consumed resources during query plan execution) gathered from the execution of global queries. Such feedback serves to learn the quality of query plans according to different parameters (e.g. some measured resource or a cost function that uses such measures as parameters). We based the learning process on the Case-Base Reasoning (CBR) paradigm. The CBR principle is to use knowledge (e.g. query feedback) from past problem solving experiences to improve the solving of further similar problems [AgPl94].

The work also concerns the way the CBR process interacts with the query plan generation process. Such process uses classical heuristics and makes decisions randomly (e.g. when there is no statistics for join ordering and selection of algorithms, routing protocols); this technique is a pseudo-random manner to explore the space of possible query plans. The optimization process learns the best query plans according to different optimization objectives during a learning phase. Then, it exploits such knowledge by reusing suitable query plans for the efficient evaluation of similar queries. This section presents and overview of the case based reasoning principle. Also, it discusses how we adapt the CBR principle to our learning-based query optimization approach.

3.5.1 Case-Based Reasoning overview

The Case-Based Reasoning (CBR) is a machine learning [Domi12] principle for problem solving. CBR learns from experiences (i.e. cases) that retain in a repository of knowledge (i.e. casebase). Such experiences correspond to previous problems and their solutions. The learning process consists in reuse previous experiences for solving further similar problems. It aims to take advantage of success problem solutions and to identify failed solutions to avoid the same future mistakes [Kolo92].

Still, effective learning process in CBR requires a well worked out set of methods in order to extract relevant knowledge from past experiences and exploit it for problems solving. Figure 3.3 illustrates the CBR reasoning principle proposed by Aamodt & Plaza's (1994) [AgPl94]. The input corresponds to the problem specification, and the output to the suggested solution. The CBR components include: (i) the cases, (ii) the similarity function, (iii) the casebase comprising past cases, and (iii) the suit of reasoning mechanisms: retrieve, reuse, review and retain steps. Next sections detail each of these components.

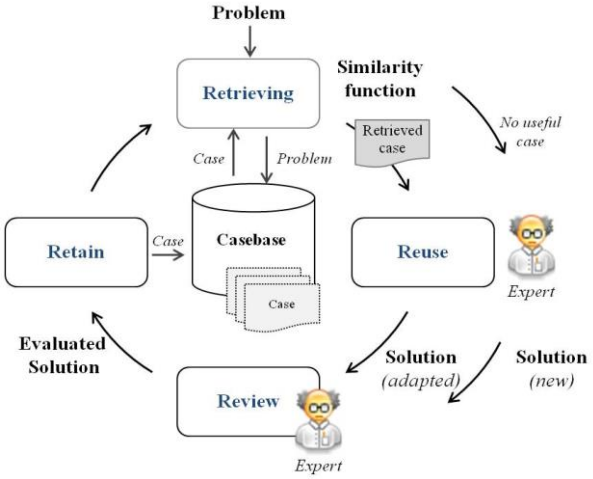

Figure 3.3. The CBR mechanisms (inspired from Aamodt and Plaza [AgPl94])

For a better understanding let us see an example that compares the human reasoning with case-based reasoning. To diagnose/repair cars malfunctions, an automotive engineering is trainer to use facts and knowledge to recognize the common signs and combinations of them. However, he cannot be trained to recognize all possible combination. When a car with unknown signs arrives, the engineering turn to its basic knowledge, but in this case, probably the process for generating a plausible diagnosis will be harder and time consuming. The advantage is that he has learned to recognize a novel combination of malfunctions, and to develop a solution that could be useful to solve a hard similar problem in the future.

Just as CBR provides a way for people to generate solutions; it also provides a way for a computer program to propose solutions efficiently when previous similar problem solutions have been encountered. So, the previous example shows that CBR is useful for people and machines that know a lot about a task and domain because it gives them a way to reusing hard reasoning they have done in the past. However, it is equally useful to those who know little about a task or domain in since it promotes the adaptability of known solutions to new problems.

Case based reasoning is also useful when knowledge is incomplete and or evidence is sparse. Logical systems have trouble dealing with this because they are strictly based on well known facts. In contrast, a case based reasoner makes assumptions to fill the incomplete knowledge based on what experiences suggest, going on from there. Solutions generated in this way won't always be optimal; this depends on the learning gained from evaluating the proposed answers.

Much of the inspiration for the study of CBR came from cognitive science research on human memory. However, the resulting methodology has been shown to be useful in a wide range of applications. Unlike most problem solving methodologies in artificial intelligence (AI), CBR is memory based, thus reflecting human use of remembered problems and solutions as a starting point for new problem solving. An observation on which problem solving is based in CBR, namely that similar problems have similar solutions, has been shown to hold in expectation for simple scenarios [LMBL05], and is empirically validated in many real-world domains. It has enjoyed considerable success in a wide variety of problem solving tasks and domains.

Case

A case is the unit of knowledge in the CBR principle. Kolodner defines in [Kolo92] a case as a "contextualized piece of knowledge representing an experience that teaches a lesson fundamental to achieving the goals of the reasoner". Thus, a problem is posed by a user, a program or a system. The knowledge gained from solving the problem is encapsulated within a case; it is supposed that such knowledge must be helpful for solving further similar problems. The question is which should be this useful knowledge?

Different kind of knowledge can be retained in a case. Very often it is only subdivided into a problem and a solution description, but additional knowledge might be necessary depending on the kind of intended reuse. Figure 3.4 shows an example of a case for our car workshop context. Let us say that the engineering receives a car whose front light doesn't work. According to the problem description, the resulting diagnosis is that the front light is fused and must be replaced.

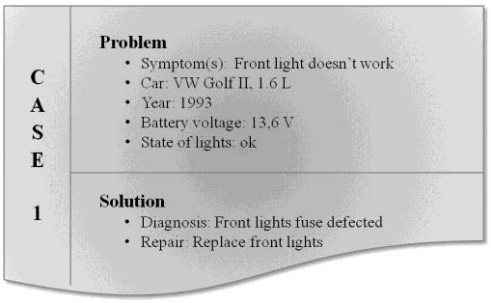

Figure 3.4 Example of a case

[Kolo92] focuses on the role cases can play in helping people make decisions and on the kinds of things that should be represented in a case so that it can be productively used for reasoning. Thus, Kolodner proposes a comprehensive case structure consisting of the following five parts: (i) problem description, (ii) the solution that was proposed, and (iii) the outcome.

The problem description may include the specific conditions placed on the problem and the goal(s) to be achieved, and if necessary/possible the current state of the context where the problem takes place. The solution includes the steps for solving the problem; it may also include an explanation of how / why such solution was derived. Finally, the outcome comprises information about the efficiency of the solution for solving the problem (e.g. the state of the context after the solution was carried out, how close was to what was expected, and lessons that can be learned from the experience).

Similarity of problems

The search of past similar problems for reusing their solutions requires a similarity function. The objective of such similarity function is to establish the resemblances between the problems; for example, car malfunctions in an automotive context, health disorders in a medical context, queries in the query optimization context. The similarity between problems is measured trough a similarity operator, or its inverse, a distance function. Two entities are as similar as higher is the similarity operator result, or inversely, as lower is the distance function result [Rich95][OsBr97].

There are many ways of measuring similarity and different approaches are appropriate for different representations of cases, particularly representations of problems within cases. For example, problems represented as feature vectors are typically compared through a vectors' distance function, the comparison of problems based on a graph representation turns to a graph isomorphism problem. The final goal of determining similarity among entities is the accurate reuse of known solutions.

Casebase

The casebase is a repository for storing cases; often it is also referred as case memory. In general, it is created with a small amount of cases (e.g. first experiences); more cases (e.g. new experiences) are included progressively. It is an expert in charge to feed the casebase with cases related to expected problems, and provides new cases when new problems arrive.

There are three general areas that have to be considered when creating a casebase [BeKP06]: (i) the representation of cases, (ii) the organization of the casebase, and (iii) the selection of indices. The objective of such elements is either to facilitate accuracy for searching useful cases, speed for searching cases, or both.

- The representation of cases. Cases in a casebase can represent knowledge in many different formats. This representation is greatly influenced by the type of knowledge that is stored. It may range from unstructured (e.g. binary files), to structured (e.g. vector of features, graphs), or ad-hoc representations where each component of case has a different format [BeKP06].
- The organization of the casebase. The collection of cases itself has to be structured in some way to facilitate the retrieval of the appropriate case when queried. Numerous approaches have been proposed (e.g. flat [KrBa93] and hierarchical [Hamm89] are common structures).
- The selection of indices. Casebase organization tends to be strongly linked to indexing; case indexing refers to assigning indices to cases to improve future retrieval and comparisons (e.g. similarity). Thus, such indices reflect the aspects of cases (e.g. attributes that influenced the outcome of the case) that determine in which context cases will be retrieved in future. [Kolo92] and [BiCo89] present representative examples of indices. Assigning indexes is typically a manual process relying on human experts, however various attempts of using automated methods were proposed in the literature.

CBR reasoning mechanisms

For solving a problem, the first step is to *retrieve* a useful case (i.e. that comprise a similar problem and provides a good solution). The solution from the retrieved case is *reuse*d for solving the new problem. Reusing such solution may involve some adjustments; account for differences between the case problem and the new problem. The solution is then evaluated and *review*ed typically by an expert. The description of the new problem and its solution can then be *retained* as a new case, acquiring knowledge for solving a new problem.

3.5.2 Adapting Case-Based Reasoning to query optimization

This section presents how our approach adapts the concepts and mechanisms of the CBR principle to the query optimization process. Figure 3.5 illustrates our CBR-based query optimization approach; it includes the fundamental concepts: (i) query-case, (ii) query similarity function and (iii) casebase structure. Also, it includes the steps of the reasoning mechanism, which correspond to the four-step CBR basic mechanism. It is important to remark that our approach does not consider the interaction with an expert, as most of CBR-based systems.

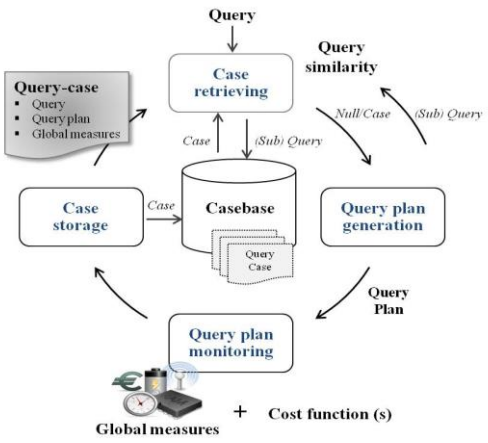

Figure 3.5 CBR based query optimization approach

CBR-based query optimization

In our approach, the problem corresponds to a query and the problem solution to a query execution plan. For evaluating the quality of plan solutions we use global measures of the consumed computational resources; these measures are obtained as feedback from the execution of global queries. We encapsulate this information within a query-case, the unit of knowledge of our learning-based optimization process. Thus, the first adaptation of CBR in our approach is the representation of query feedback as a case.

Given a query Q, the *Query case retrieving* phase selects a case of a query similar to Q and that comprises the query plan that minimizes a given cost function. The *Query plan generation* process reuses the query plan P from the retrieved case by adjusting them according to the specifications of Q (such a plan was used for evaluating a query similar to Q, but not exactly the same). The generated query plan is then executed; the global measures of consumed computational resources are gathered during the *Query plan monitoring* phase. Finally, during the *Case storage* phase, a new case is created and insert into the casebase, or an already existent case is updated. When no useful case is retrieved for solving the query Q, the process decomposes Q into subqueries and tries to retrieve useful query-cases for solve them.

Well understood, the definition of similarity between queries is another fundamental aspect for our CBR-based optimization process, in particular for the query case retrieving step. We propose a qualitative query similarity function based on query features. It is important to consider that such a function should guaranty that retrieved cases are useful to improve query solving. Thus, it is expected that if a query plan had a low cost (e.g. memory consumption) for executing a query Q, it will remain low cost for evaluating a similar query Q'. The challenge is to identify the query features that provide accuracy for useful case retrieving, but that also extend cases usability. A very restrictive similarity function (e.g. equal queries) has a high accuracy (e.g. precise measures of resources consumption); however it makes difficult cases exploitation (e.g. a case serves for solving a single query).

For the design of the casebase, we propose an ad-hoc structured representation of cases (a particular representation of each of its components i.e. query, query plan and measures). We take advantage of our query similarity function grouping cases in clusters of cases with similar queries. We refer to such a cluster as query *family*. This casebase organization accelerates the case retrieving process. We remain simple for the casebase structure and we use indices structures to still improving its access and management.

Our CBR-based query optimization process comprises a learning phase and an exploitation phase. The learning phase supplies the casebase with new query cases. Typically, the CBR principle assumes that there is an expert in charge to feed the casebase with solutions for the expected problems. We propose a query plan generation technique that uses classical heuristics (e.g. selection and projection first) and that makes some random decisions (when classical query optimization techniques uses metadata). The plans cost is learned from real measures of resources consumed during the plan execution (instead of using some kind of metadata for plans cost estimation). Thus, our evaluation of plan solutions relies on learned measures, while in CBR it is the expert also in charge to evaluate the quality of solutions. The learning phase finishes when a threshold of knowledge (i.e. number of query-cases) is reached.

During the exploitation phase the query plans are generated by reusing and setting plans from learned query cases. The optimization process selects and reuses the plan that minimizes a given cost

function, the advantage is that such function is based on real measures of consumed computational resources. When no useful cases exist within the casebase for solving a query Q, our optimization process pursues to reuse cases for solving subqueries from decomposing Q. A complex plan setting may represent an overhead for the query optimization process; however we propose a simple setting process minimizing the consumption of resources.

For ensuring continuous improvement of the execution of queries the following actions are taken:

- When a casebase does not comprises enough cases (customizable threshold) about a query, query plans are generated by pseudo-random to try and learn.
- Once concluded the learning phase, the search space exploration continues by periodically generating the query plans by pseudo-random. The frequency is indicated by an adjustable variable (*fv*). For instance, *fv* = 10% means that from 100 query plans generated during the exploitation phase, 10 will be generated by pseudo-random.
- The learned query plans may be executed several times (i.e. the optimizer has decided that is a suitable plan for executing a query), thus the measures of computational resources are systematically updated.

Chapter 4 is consecrated to the representation and management of cases. Thus, it presents in detail the representation of case and its components. It also exposes the organization and indexes of the casebase. Chapter 5 focuses on the plans generation process.

Distributed query optimization

In our approach, global queries are optimized in a des-centralized manner. We assume that the nodes in the system provide functionalities for data management and processing. Each node in the system is responsible for optimizing the (sub) queries that it receives; naturally by applying the optimization process explained before.

The optimizer at each node takes its own decisions according to its knowledge i.e. query-cases. The optimizer in the node that receives Q determines the subquery(ies) to execute locally and those to be sent through the network. Also, it chooses the nodes for sending the subqueries (i.e. next-hop), and the operators to be executed locally for combining the partial subquery results.

In summary, the optimization of a query is the result of a cooperative optimization process among the nodes that participate in the execution of a query. Each node has a vision of its *mini-world*, it knows its neighbors, and maintains and exploits the query-cases from the execution of queries in which it has participated. Therefore, the type and amount query-cases vary from one node to another.

The knowledge of queries varies at each node. In the best case, all nodes involved in the execution of a given query plan have enough cases to make appropriate choices. On the contrary, if some node(s) is (are) in learning phase, the corresponding sub-query plan most often is sub-optimal impacting the cost of the global query execution.

For example, let's say that a query Q is posed at node N_1 that selects a plan P. Such plan decomposes the execution of Q in the execution of the subqueries SQ_1 and SQ_2 bound by a binary operator (i.e. union). Let's say that the first subquery must be executed locally and the second one must be executed in a remote node N_2. Let's say now that N_1 has reached the exploitation phase concerning SQ_1, since it has already tried-learned several plans for executing such query. In contrast, N_2 is still in learning phase concerning SQ_2, thus the generated sub-query plan is sub-optimal affecting the global execution of Q.

We must consider that the consumed computational resources associated to the execution of P take into account those consumed by the execution of each subquery. Therefore, even if the execution of SQ_1 is optimal, the execution of SQ_2 may consume a lot of resources. In this case the optimizer learns that P is a sub-optimal plan with the risk of never reuses it. However, P can turn in an optimal plan when N_2 moves to the exploitation phase for SQ_2 generating an optimal sub-plan. In short, the optimizer makes hasty conclusions about the efficiency of the query plan P resulting in wrong knowledge. The pseudo-random generation of query plans once in a while after the learning phase reinforces this problem if occurs in some of the involved nodes.

In order to solve this problem a boolean *reliability* variable is associated to query-cases. If all involved nodes are in exploitation phase, thus have enough knowledge to choose an efficient sub-query plan, the *reliability* for the selected query-case is *true*; otherwise it is *false*. This mechanism greatly improves the global optimization because: (i) accurate measures about the cost of query plans i.e. when P is associated to a poor-performance because measures of computational resources are very high are not reliable since a sub-plan was selected without enough knowledge, (ii) exploration of the search space avoids to discard a part of the plan that can pursue to a optimal plan.

3.6 CLASSICAL QUERY OPTIMIZATION VS LEARNING-BASED QUERY OPTIMIZATION

For a better understanding of our approach, we consider interesting to analyze in detail the main aspects of classical query optimization techniques and discuss the main similarities and differences with our learning based query optimization approach. We address this analysis in three parts that correspond to the components of classical query optimization: (i) the space of possible plans for solving a query (i.e. search space), (ii) the model for estimating plans cost, and (iii) the search strategy for exploring the search space.

3.5.1 Search space

The search space comprises the possible execution query plans for solving a query. In classical query optimization, given a query a temporal search space is generated, once determined the optimal plan, the search space is discarded. If the same query is posed latter, the search space must be created from scratch again. In our approach, given a query we create physical query plans that are tested, analyzed and encapsulated within a query case. Such query case is strategically stored in the casebase, and remains it there according to the efficiency that the corresponding query plan as show through the learning query optimization process.

Please consider that there are several plans for solving a query, but according to our approach, also several queries sharing some fundamental characteristics (i.e. similar queries) can be solved using the same plan with some few settings. Since another point of view, for us a search space is not only a set of possible plans for solving a single query, but a set of possible plans for solving a set of similar queries, where a family of cases corresponds to a part of that search space already explored for a set of similar queries. The management and exploitation of query cases aims the exploration of the search space to keep in the casebase only the *best* learned plans.

3.5.2 Plans cost

A cost model comprises mathematic functions for estimating the plans cost; such functions reflect the optimization objective(s). These cost functions are based on information –typically called *metadata*- about data sources (e.g. cardinality and data statistics). In distributed query optimization, also is required information about the communication media (e.g. network topology) for calculating the data communication cost (i.e. cost of sending data from one node to another).

The challenge of distributed query optimization with incomplete information on data is one of the main concerns of this thesis. If metadata is not available, we propose to gather/generate another type of knowledge (i.e. query cases) suitable for estimating the query execution cost.

The query cases contain the measures of computational resources consumed during the execution of plans to measures their quality according to determine optimization objective. It is precise to define new cost models where computational measures are the parameters of the cost function, instead of the conventional metadata. The definition of a new cost model is not the goal of this thesis, however in this document we discussed about a very admittedly simplistic cost function to exemplify the usability of collected measures for determining the query execution cost.

3.5.3 Search strategy

A search strategy selects an optimal execution plan for a given query by exploring the search space. In classical query optimization techniques, given a query a search space is generated and explored until finding the optimal plan. In contrast to classical query optimization, in our approach the exploration of the search space is progressive. Given a query, a single physical plan is generated and evaluated; the experience is stored as a case within the casebase.

During the learning phase several plans are learned for solving queries from different query families. For example, let us say that during the learning phase 1000 queries from 10 different families were run. Let us say that a family QF1 was feed with 100 cases, each case comprises a possible plan for solving queries similar to query cases of that family. Thus, part of the search space was generated and evaluated during the learning phase. The selection of a suitable query plan from our search space takes place during the query case retrieving phase. Given a query, this phase happen once if it is found a query case useful to solve the whole query, or several times if is required to look for query cases that concern the subqueries of the original query.

3.7 CONCLUSIONS

This chapter introduces the characteristics that we consider for the distributed query optimization context. It details our assumptions related to structures and distribution of data, and exposes the characteristics of queries that we address in this thesis. In a second part, it presents an overview of the case-based reasoning principle and highlights the adaptation of CBR to our query optimization approach. Finally, it discusses the differences between classical query optimization techniques and our learning-based query optimization technique.

The objective of adapting CBR to query optimization is to keep knowledge useful (different of classical metadata) to find suitable plans, going further than simply using optimization heuristics. A query case highlights the optimizer decisions and their consequences (query plan and measures of query plan execution respectively). For example, a query case may contain knowledge about the most pertinent decomposition of a query (set of subqueries), the operators for composing the final result,

and a suitable order and execution techniques for computing such operators. Reasoning over this knowledge pursues to explore the space of possible plans progressively keeping useful solutions (i.e. efficient query plans) only.

A query case can be used several times for solving similar queries instead of systematically generating plans from scratch (for each query that arrives) using only heuristics. During the optimization process it is learn that a given query plan is efficient according to certain optimization objective. Some CBR based works have proposed to use cases for solving sub-problems, when there are no cases for solving the whole problem. In our approach, we exploit cases for solving parts (i.e. sub-queries) from a given query when no cases exist for solving the whole query. Finally, measures (computational resources consumed by a query plan) within cases serves to learn the real cost of plans.

The CBR process has been widely studied. While lot of techniques have been proposed for addressing some steps of this reasoning cycle, such as retrieve and reuse. Others have relay in the intervention of experts, for instance the generation of new solutions assuming that it is the expert that provides the casebase with new cases and that reviews the quality of the proposed solutions for solving a problem.

On one hand, we inspire our proposal on the CBR principle by adapting its main foundations (i.e. knowledge representation and reasoning process) to the query optimization process. On the other hand, we take advantage of some query optimization techniques that partially solve the query optimization problem (i.e. query optimization with incomplete metadata) that we face on this thesis. For example, the use of well known optimization heuristics, the consideration of random optimization decisions to improve progressively (from randomized search strategies [IoKa90][StMK97] and genetic algorithms [BeFI91] [ViPa11]), and the use of query feedback (from adaptive query optimization [GPFS02][DeIR06] and plan caching [GPSH02][VaKi00][AbCh99c] techniques).

It is important to mention that our optimization process is independent of human interaction, in contrast to the generic CBR principle where it is assumed the intervention of experts for: (i) supplying the casebase with new cases, and (ii) for evaluating the quality of proposed solutions for problem solving. In our approach, the supplying of new solutions corresponds to the generation of new query plans. As one of the main contributions of this work, we propose a query plan generation technique that combines the *casebase* exploitation for taking advantage of learning, with classical query optimization heuristics (e.g. operator ordering) and random decisions (e.g. join ordering, operators' algorithms) for a pseudo-random exploration of the space of possible solutions.

Another role of the expert is to evaluate adapted solutions, as mentioned before. In our query optimization process, we evaluate the quality of solutions (i.e. query plans) by gathering measures about the computational resources consumed during queries execution. Such measures are exploited the query case retrieving phase; where the cases that optimize (i.e. minimize the consumption of resources according, for example memory, energy or time) the evaluation of similar queries are selected.

Chapter 4 and Chapter 5 present in detail our learning-based distributed query optimization approach. Chapter 4 is consecrated to expose the query case representation, our definition of queries similarity, and the organization and management of cases within the casebase. Chapter 5 focuses on the generation of query plans in both, during the learning phase (plan generation from scratch) and during the exploitation phase (plan generation using cases).

4. REPRESENTATION AND MANAGEMENT OF QUERY CASES

Case representation is the most fundamental issue in case-based reasoning. Ascertaining the correct case features was viewed by Barletta [Barl91] as the most important task in building CBR systems. The steps of the CBR reasoning process depend on the case representation. Section 4.1 exposes our query case representation.

The query similarity is the second foundation for the success of our CBR based approaches. It is the basis for retrieving useful cases for solving a query. Through the query similarity definition it is possible to determine the connections between features of the new query with those of previously solved queries. We take advantage of query similarity for the organization of the casebase by grouping cases in clusters that we term families. A family comprises cases that have similar queries. The definition of query similarity is presented in Section 4.2, and the clustering of query-cases according to queries similarity is defined in Section 4.3.

Section 4.4 presents the casebase storage structure and organization of cases, followed by the operations for managing the casebase in Section 4.5. Such operations comprise the algorithms for case retrieving, storage (insert and update) and deletion.

4.1 QUERY-CASE REPRESENTATION

The representation of cases is the main foundation of the reasoning process. The efficiency of all steps of such process depends on it. Issues such as case components, contents of such components, and case structure all come into the picture. The first aspect to consider in our query case representation is that the content of the query case must be useful for the generation, evaluation and improvement of plans for evaluating queries efficiently. The second objective is to provide case structures that lead to an inexpensive optimization process, thus minimizing the consumption of resources for: storing, managing and exploiting cases.

Our query case has the form QC:<Q,P,M>, thus it is composed of three following parts:

- The query Q solved in the past; we define an abstract representation of query, which independent of a specific high-level declarative query language. Section 4.1.1 details our abstract representation of query.

- The plan P for solving a query; it is represented by a tree of physical operators. Section 4.1.2 exposes the representation of a plan in a query case.

- The evaluation of the plan in form of a set of global measures M. A global measure correspond to the total amount of some computational resource (e.g. CPU, memory, energy, etc.) consumed by the nodes that participate for the execution of the plan P. Also, such global measures may include response time (i.e. time for obtaining the first query result) and execution time (i.e. time for obtaining the complete query result).

In the query case components we include the representation of a query since the similarity between queries is assessed during useful case retrieving. Useful query cases comprise plans for the efficient evaluation of queries. The global measures serve to identify the plan that minimizes the consumption of resources or, a given cost function (using the global measures as function parameters). The accuracy of useful case retrieving depends on the similarity between queries and the evaluation of plans.

The physical representation of cases is another important issue to consider, it varies from structured, semi-structured and unstructured representations. Also, there are specialized (e.g. hybrid, application ad-hoc) representations that combine the previous approaches for representing the different parts of a case. Major systems used case representation approaches extremely specific to the application. We must consider an approach flexible enough for representing our query case content and that economize the resources required by the optimization process, this is: (i) compact storage, and (ii) straightforward exploitation and management processes.

For our query-case representation, we focused on a structural representation, appropriate for complex applications and that allows accurate retrieving of useful cases [Kowa91]. The unstructured (i.e. textual approach) and semi-structured case representation are hardly appropriate for the representation of our kind of problems (i.e. selection-projection-join queries) and solutions (i.e. query-plans). Another disadvantage of such approaches is that the accuracy of retrieving useful cases is not very high [Berg02].

There are different structural approaches; attribute-value, graph-oriented, database-oriented logic of predicates and object-oriented are of the most popular. In [Berg02], Bergmann details the advantages and inconvenient of these representations. However case representation for case-based planning are sometimes more specialized due to the specific structure problems (i.e. queries) and solutions (i.e. query plans). We propose ad-hoc structures for the different case components aiming to favor the processing steps to which they serve for, and to minimize the amount of memory required for cases storage.

For example, case retrieving is based on query similarity; where the comparison of queries is fundamental. We propose an abstract representation of query using a structure that pursues to go from a problem of complex queries comparison, to the comparison of specific elements of the query only (relevant according to the query similarity definition). This modularization of query representation is easy to adapt to different similarity functions. The comparison of specific query elements (and not all the complex query) minimizes the time and CPU consumption. Next subsection details the representation of our query case components.

4.1.1 Query

As major area of CBR applications is problem solving, it is unsurprising that cases contain a problem representation, where information describing the problem is stored. So far in this thesis we have exposed that in our approach queries correspond to the problems to solve. The rationale in our CBR optimization approach is to compare a new query to past queries using some matching method. It is the query representation which is used in remembering a past case, and in determining its applicability to a new situation.

Our abstract query representation is based on the specifications of the declarative DLAQL [ABCD12a][ABCD12a] query language, which extends SQL2 data manipulation language with clauses for specifying fragments of data in a query expression (the specifications of DLAQL is detailed in Section 3.3.2). However, it can be easily adapted to the specifications of other SQL-like languages. This thesis puts special attention to the optimization of Select-Project-Join queries, since are some of the most common queries and the order of binary operators (e.g. join) represent an important problem for the query optimization domain. Thus, this section focuses on the representation of queries only, even if the DLAQL language also includes updates.

Given a user query (in some high-declarative language), it is translated to our abstract query representation, which comprises the following elements: (i) *Projections*, (ii) *Sources*, (iii) *Joins* and (iv) *Subqueries*. Such representation corresponds to a compact structure, easy to compare (computation of query similarity), and also easy to extent for including different query features. Detailing the query translation process is not the main concern of this section, basically it extracts information from the query (i.e. expressions, sources and conditions) to fulfill the elements in our abstract representation. The abstract representation of the query Q for selecting the avatars located in the area 7 (DLAQL query in Chapter 3 - Example 3.1) is as follows:

Example 4.1 -

```
Q: Query = <
       Projections = {Avatar},
       Sources = {SP= <Positions P, {<Area, =, 7>}, {global}>},
       Joins = {}
       Subqueries = {}
   >
```

The execution of such query only consider specific fragments of the global Positions Itemset, those stored at the node where Q is posed (local fragment), and those stored at nodes J and D (remote fragments). In the abstract representation, the *Projection-expressions* corresponds to a singleton that comprises attribute Avatar of the Itemset Positions. *Sources* corresponds to a singleton as well that comprises the source S_P. Such source is specified by the Itemset Positions, the restriction condition <Area, =, 7> for filtering the data of avatars in area set only, and the scope {local, J, D} that specifies the data fragments (i.e. identifiers of nodes that store such data) to consider for answering the query. Since is a query with a single source, *Joins* is an empty set. The query does not comprises subqueries, thus the set *Subqueries* is also empty.

The specification of our abstract query representation is defined in a context-free grammar because of its precision and clarity. The used grammar has the following notation:

- The 〈 〉 symbols represent a concept, for example 〈query〉

- The := symbol is used as the assignment operator to separate a concept and its definition i.e. concept components or concept specializations
- The symbol coma (,) separates the concept components
- The | symbol separates the concept specializations
- The [ ] symbols enclose optional concepts
- The ( ) symbols group concepts
- The * symbol indicates that a concept can be repeated several times
- Constant keywords and symbols are represented in **bolt font**

This section former exposes the representation of mono-source queries, which comprise a single source of data and may be some conditions/expressions for data filtering. Second, it presents multi-source queries, since they comprise two or more data sources, conditions for joining such sources, and may be some conditions/expressions for data filtering. Finally, it represents the union of mono-source and multi-source queries. For easy to reading, these queries are addressed in three subsections respectively: (i) Select query, (ii) Join query, and (iii) Union query.

```
<Query> := <Select-query> | <Join-query> | <Union-query>
```

Select query

A select query selects data from a single source only. Thus, it includes just unary operators i.e. projection and rename operators, a source and a list of subqueries.

```
<Select-query> := (<Projections>, <Sources>, <Subqueries>)
```

The abstract representation of the query Q for selecting the avatars located in the same area as the avatar named '*MyAvatar*' (DLAQL query in Chapter 3 - Example 3.5) is shown in Example 4.2:

Example 4.2 -

```
Q: Select-query = <
            Projections = {*},
            Sources = {SP= <Positions P, {<Area, =, Q1>}, local>},
            Subqueries = {
               Q1: Select-query = <
                      Projections = {Area},
                      Sources = {SP1=<Positions P, {<Avatar,=, 'MyAvatar' >},
                                 global >} >
               }
      >
```

In the query Q, the set of *Sources* is a singleton that comprises the source S_P. Such source is specified by the Itemset Positions, the restriction condition <Area, =, Q_1>, where Q_1 is a subquery for selecting the location of '*MyAvatar*'. Thus, the restriction condition of S_P filters data from Positions related to those avatars at the same area than '*MyAvatar*'. The source S_P also specifies a local scope, which indicates that the query Q only considers the fragments of Positions that is stored at the node where Q was posed. Finally, the *Projections* comprise all the source-field of S_P. The subquery Q_1 also comprises a single source S_{P1}. Such source is defined over the Itemset Positions; it has a restriction condition that looks for the Avatar named '*MyAvatar*'. This time, the source has the scope defined by

default (i.e. global), which means that may be required to access local and distant source fragments for finding the answer. The subquery projects the source-field Area only.

Projections. The data to project are specified in a projection expression that includes values, source fields, functions (i.e. aggregation and arithmetic) and/or another projection expression.

```
<Projections> :=    <value>*
                    | [<source-name> | <alias>].<source-field>*
                    | <function> | <projection>
```

In the projection expression, a <value> belongs to different typed values (e.g. integer, strings, boolean, date, etc.) respecting some compatibility rules (e.g. the values of type integer are a subset of the values of type float). The <function> concerns aggregate or arithmetic operations.

```
<function> :=    <arithmetic>  | <aggregation>
```

An arithmetic function comprises operands and classical arithmetic operators. An operand may be a <value>, a <source-field> and/or other <function>.

```
<arithmetic> := (<operand>, <operator>, <operand>)
<operand> := <value> | <source-field> | <function>
<operator> : = + | - | * | /
```

An aggregation function receives as input parameter a set of <source-field> instances and returns a single <value> of the $\mathbb{R}$ domain. It includes the common aggregation operations such as sum, max, min, avg and count.

```
<aggregate> :=   min ( ([<source> | <alias>].) <source-field>)
                | max ( ( [<source> | <alias>.] ) <source-field>)
                | avg ( ( [<source> | <alias>.] ) <source-field>)
                | count ( ( [<source> | <alias>.] ) ( <source-field> | *) )
```

Source. A source comprises a supply of data that may correspond to an Itemset or to a query resultset, an optional set of restriction conditions over such data, and a scope. The scope specifies the fragments of data (e.g. data stored in the local node, and in node J) that the source includes to select data; a source has a global scope (whole fragments) by default. A source is described by a schema that is a sequence of source-fields. A source-field corresponds to an attribute of Itemset attribute or to a column of query resultset.

```
<Sources> := <source>*
<source> := ( (<dataset> | <sub-query>) [,<restriction-condition>] [,scope])
<source-schema> := (<source-field>*)
```

A restriction condition is a propositional formula pf that consists of atoms and logical operators; it selects the data which holds pf. An atom is composed of a <source-field>, a <cmp-operator> and a <value>. An atomic restriction condition comprises a single atom; a conjunctive

restriction condition comprises a set of atoms or other conjunctive restrictions connected with the conjunctive operator (AND).

```
<restriction-condition> := <atomic-rc> | <conjunctive-rc>
<atomic-rc> := [<source> | <alias>].<source-field>, <operator>, <value>|<sub-query>
<conjounctive-rc> := (<atomic-cr>, AND, (<atomic-rc> | conjunctive-rc>))
<cmp-operator> := < | ≤ | ≥ | > | = | ≠
```

The scope of a <source> specifies the nodes that store the fragments of data to select. The specification of nodes is explicit as a list of <value>(s) of nodes identifiers, or implicit as a <sub-query> that returns a list of nodes identifiers.

```
<scope> :=      (<value> | <sub-query>)*
```

Subquery. A <sub-query> is a query (select, join or union) nested inside another query. Also it can be referred as inner-query and the query that includes it as outer-query. According to the so far defined query components, a subquery can correspond to an implicit dataset of a source, an implicit value in an atomic restriction condition, or an implicit value in a source scope.

Join query

A join query is a select-project-join query as the DLAQL query in Chapter 3 - Example 3.3. It adds a set of join-conditions to the characteristics of the select-query. This condition allows defining a criterion for selecting data from two or more sources (e.g. S_P and S_S).

```
<Join-query> := <Projections>, <Sources>, <Joins>, <Subqueries>
```

A join condition comprises a source-field of each source (e.g. S_P*.Area and* S_S*.Area*), and a comparison operator. A join query can comprise several join conditions connected with a conjunction operator (∧).

```
<join-condition> :=   [<source> | <alias>.]<source-field>,
                      <cmp-operator>, [<source> | <alias>.]<source-field>
<Joins> := <join-condition> , [ AND, <join-condition> | Joins>]
```

An example of the representation of a given join query is presented below (Example 4.3).

Example 4.3 -

```
Q:  Join-query = <
        Projections = {Avatar},
        Sources = {
                SP= <Positions P, {<Area, =, 7>}, {local, J, D}>,
                SS= <Servers S, {}, {global}>}
        Joins = {<SP.Area, =SS.Area}
        Subqueries = {} >
```

Union query

The union query combines the result-set of two or more select queries connected with the union operator. Notice that the <Projections> of each <subquery> must have the same number of source-

fields. The source-fields also must refer values pertaining to the same domain. Also, the source-fields must have the same order.

```
<Union-query> :=  [<Projections>] , <subquery>,<subquery>
```

The following example of union-query is the result from the query rewriting for expressing the original query Q in terms of the fragments of the decomposable source S_P i.e. local fragments (subquery Q_G, where Q was posed at node G) and distant fragments (subqueries sent to remote nodes e.g. Q_J and Q_D).

Example 4.4 -

```
Q :    union-query = <
       Projections = {*}
       QG: selection-query = <
              Projections = {*}
              Sources = {Positions PG, {P.Area=7}, local}
              Joins = {}
              Subqueries = {}>,
       QJ: selection-query = <
              Projection = {*}
              Sources = { Positions PJ,D, {P.Area=7}, J, D}
              Joins = {}
              Subqueries = {} >,
       >
```

4.1.2 Query plan

In CBR system, where a problem is to be solved is store in each case a solution component (i.e. query plan) which describes the solution for that particular case. In our approach it is essential to include plans within query cases. These are reused for solving similar queries. Figure 4.1 depicts an example of query plan.

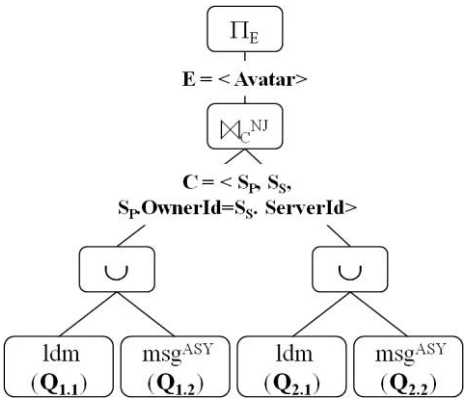

Figure 4.1. Example of a query plan

We represent a query plan as a tree structure where nodes correspond to physical operators (e.g. data manipulation operators) and the edges to the flow of data between nodes. Within the plan structure, the leaves of the tree serve for accessing local data or for exchange queries/data through the network. The inner nodes correspond to data manipulation operators and the root node prepares the

final or partial result to return to the user or to an intermediate node respectively. A plan may also include specialized operators (e.g. invocation of programs and functions) as inner or leaf nodes.

The sample query plan comprises local data access (ldm) operators for the local execution of (sub) queries, and message (msg) operators to send (sub) queries wrapped within messages to be executed at remote nodes. The inner nodes are join ($\bowtie^{ALGO}$) and union ($\cup$) operators to compose a complete query result from local and distant partial result-sets. The root node is the print operator to return the final result to the user. The operators included by a query plan are described in detail below.

- Leaf nodes
  - data-access (ldm). submits queries in a local data management system to retrieve partial data result-sets from the local node.
  - message (msg). corresponds to a send-receive operator for queries and data transfer. It sends queries to be executed at remote nodes, or sends partial data result-sets to requesting nodes. Once a query Q has been sent within a message, it is expected to receive the partial data result-sets from queries executed at remote nodes. A message operator may also comprise measures of computational resources consumed at the remote node during the execution of Q. A message can be sent using different strategies, for example synchronous (msg^{SY}) or in asynchronous fashion (msg^{ASY}).
  - specialized operators, for example programs implementing ad-hoc strategies for executing frequent queries.

- Inner nodes
  - filter (σ_{RC}). determines the data values that hold with a restriction condition over a source.
  - sort. orders the data according to certain source-field.
  - join ($\bowtie_{JC}$). combines two sources according to a given join condition. There are multiple algorithms for its computation i.e. nested-join ($\bowtie^{NJ}$), merge-join ($\bowtie^{MJ}$), hash-join ($\bowtie^{HJ}$) and semi-join ($\bowtie^{SJ}$).
  - union ($\cup$). puts together the data from two or more sources.
  - specialized operators, for example distributed join algorithms ad-hoc to a specific application, user defined functions for data treatment, etc.

- Root nodes
  - print. there are two types of print operator: one in charge to return the complete query result to the final user; and (ii) another that prepares a partial result-set from the up to now execution of a query Q, and the measures of computational resources currently consumed to be returned to a node (data and measures to further be send within a message).
  - project (π_E). determines the source-fields that hold with a projection expression

4.1.3 Global measures

A case with only a query and a query plan gives no indication of whether or not a plan is suitable for evaluating a query efficiently. CBR suggests including within a case some additional information, called *outcome*, with the purpose to evaluate the efficiency of a given solution. In our approach we use global measures (e.g. computational resources, response time and execution time) of computational resources consumed during the execution of query plans to select the plan that minimizes a given cost function. We consider that such cost function uses global measures as parameters.

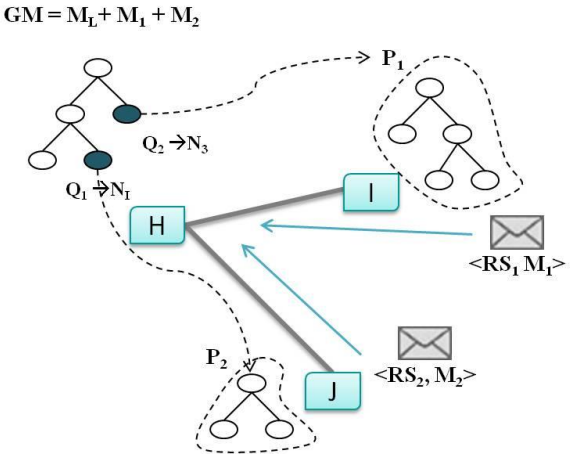

Figure 4.2 Example of global measures

A global measure is the total amount of some computational resource consumed during the execution of a query. Thus, it comprises the resources consumed at each node that participates for executing the query. Figure 4.2 depicts an example, where a query is posed at node H, which generates a plan that includes some subqueries (i.e. Q_1 and Q_2) that are sent through the network to be executed at some remote nodes (i.e. node I and node J). Each remote node is in charge of processing the subquery that it receives, and send the partial results-sets (RS_n) and measures of consumed resources to the requesting node. The node H computes the global measures including the resources that it consumed locally M_L, and the measure M_1 and M_2 that it receives from nodes I and J respectively.

A measure is represented as a triplet of the form m_n:<N, V, U> that comprises the resource name, the total amount of the consumed resource, and the unit of measure. The *name* (N) of the measure belongs to the $\mathbb{S}$ domain of strings. The *value* (V) corresponds to the amount of the consumed resource belongs to the domain $\mathbb{R}$ of real numbers (e.g. run-time, memory, CPU and energy), or the domain of $\mathbb{N}$ natural numbers e.g. messages and cardinality of result-set. The *unit* (U) of measure clearly depends on what is measured, e.g. milliseconds, kilobytes, hertz, IPS, messages transfer rate, among others).

For instance, to process a query in a sensor network some measures to gather during the execution of the query plan could be the following: M={*m_1:<memory, 200, kB>*, *m_2:<energy, 10.4, mA>*, *m_3:< time, 150, ms>*, *m_4:<messages, 2, integer>*}. There are different possible measures to be hold within a case, for example, the computational resources that were consumed during the last execution of the query plan, or the average of computational resources consumed at the several plan executions. Global measures may also be represented in more complex forms such as histograms that register the computational resources consumed by a plan at each execution.

4.2 QUERY SIMILARITY

The definition of query similarity is essential to retrieve promising candidate plans for evaluating a query. Its objective is to identify those queries that can be evaluated using the same (close to optimal) plans. Thus, given a query Q it can be evaluated reusing a plan P that was used for evaluating a previous similar query Q', and that minimizes a cost function to accomplish some optimization objective.

The literature is full of examples of similarity measures; in general, each running CBR approach comprises necessarily at least one way for measuring similarity. Even, sometimes the measure is not fixed and can be improved by a learning process, or customized by an expert. In particular, database literature has proposed to define query similarity basing on its syntactic properties and / or on its semantic properties.

A straightforward query similarity definition is to consider that two queries are similar if they can be solved using the same plan shape [GPSH02]. Thus, such plans comprise the same operators, and the same operators order and implementation algorithms, but some values of operators' parameters are replaced by variables. The problem is that plans having the same shape may serve for evaluating quite different queries. Thus, let us say that Q and Q' are similar according to the previous similarity notion, thus a plan P that minimizes the cost function for evaluating Q but not for Q'. To alleviate this problem, our query similarity definition specifies some features that should be common to queries to denote them as similar. Thus, the query plans have the same shapes, but specific values for some operators' parameters.

Some works go further by including specific properties of different query features; for example, statistics (i.e. cardinality of sources, distribution of data values, etc), or indexing structures over attributes in join and selection conditions [ZhLa00][SeHa03][BaHM07][GuGO12]. The problem is that we cannot guarantee to account with such information due to the characteristics (e.g. autonomy of components and highly distribution of data) of the query execution environments that we consider.

Let S be the set of all queries that can be evaluated using a plan P. In one extreme case, all values of operators in P are replaced by variables. P will not accomplish the optimization objective (minimize the cost) for evaluating any query Q pertaining to S. This is because is difficult to capture the performance of so many different queries using the same plan. In the other extreme, if each query in S was using a different plan, we could get a very accurate knowledge about the plan(s) that optimize the objective. However maintaining a separate plan for every possible query makes hard the exploitation of cases, a huge number of plans would be required. It is really difficult to determine the query features that dictate query similarity while maintaining a suitable trade-off between the margin of error and the exploitation of knowledge.

Our query similarity definition is focused on the comparison of the query features. Section 4.2.1 presents the definition of our similarity query definition. Section 4.2.2 formalizes our query similarity definition.

4.2.1 Definition

Our definition of query similarity is based on the absolute similarity approach, which classifies two objects as being similar or not similar [OsBr97]. E.g. Spanish beach holidays and Greek beach holidays are similar, but Spanish beach holidays and Alpine walking holidays are not. Given two or more measures of similarity of this kind, e.g., for different attributes of an object, it is easy to combine

them to give new similarities with obvious intuitive meanings; for example conjunction (*x* is similar to *y* only if it is similar on both attributes).

A more complex definition of similarity assigns a numeric value, often normalized to an interval of values {0, 1}[Grif97]. Using this approach implies to determine the range of similarity for which a plan is still useful for solving similar queries. However assigning a numeric value to similarity is often arbitrary, and hard to justify in intuitive way. For example, let us say that a Greek beach holiday might be more similar to beach holiday than to alpine holiday, but it is hard to say the adequate number that characterizes this similarity. The assignation of numbers may associate arbitrary meaning to the definition of similarity [OsBr97].

Given a query Q_n, our similarity definition considers the query features Q_{ni} specified in our abstract query representation: (i) projections, (ii) sources, (iii) joins, and (iv) subqueries. The similarity of two queries is the conjunction of comparative functions over different query features. Each query feature is compared independently, and this comparison (i.e. equality, equivalence, similarity) may be different for each element.

In the remainder of this section we expose a prospective definition of query similarity according to our previous explanation. However, different definitions of query similarity can be proposed basing on the introduced approach. For example, let us say that queries are similar if they have (i) similar sources, and (ii) the same join conditions. Two sources are similar if the restriction conditions over the sources include the same attributes, but not necessarily the same values. As mentioned before, we do not consider different similarity degrees, thus queries are similar or not. An example of two similar queries is presented below.

The similar queries Q_1 and Q_2 have some variations; however, all of them acceptable sustain the similarity of queries according to the previous query similarity definition. Q_1 and Q_2 have similar sources, since they refer the same itemsets (i.e. Positions and Servers), the same fragments of each itemset, and have restriction conditions over the same attributes. However, the projection expressions change; Q_1 is interested in the name of Avatars, and Q_2 request all the information all about the Avatars (not only the name). Also, the condition values change, from information of avatars at Area 7 to the information of those at Area 3.

```
Q1:  join-query = <
       Projection-expression = {Avatar},
       Sources = {
         SP=<Positions P, {<Area, =, 7>},
             {local, J, D}>,
          SS=<Servers S, {}, {global}>}
       Joins = {<SP.Area, =SS.Area> }
       Subqueries = {}
     >
```

```
Q2:  join-query = <
       Projection-expression = {*},
       Sources = {
         SP=<Positions P, {<Area, =, 3>},
             {local, J, D}>,
          SS= <Servers S, {}, {global}>}
       Joins = {<SP.Area, =SS.Area> }
       Subqueries = {}
     >
```

The query Q_3 presented below is non-similar with respect to Q_1 and Q_2. Q_3 changes the scope of the source *Positions*, since it is not interested in the data fragment of the node D, in contrast of the other both queries. Q3 does not accomplish our definition of similarity since one of its sources does not *match*.

```
Q3:  join-query = <
            Projections= {*},
            Sources = {
               SP= <Positions P, {<Area, =, 7>}, {local, J}>,
               SS= <Servers S, {}, {global}>}
            Joins = {<SP.Area, =SS.Area> }
            Subqueries = {}
>
```

The fact of considering only these query features remains some flexibility that can lead to cost estimation errors. For instance, the communication cost can drastically change in function of the size of query results, which is directly bound to the values of the condition restrictions. However, a very rigid similarity function leads to knowledge very difficult to exploit, since the gathered plan and measures will be useful for very few further queries.

We try to compensate this margin of error at the query-case retrieving process that we explain in Section 4.5.1. As we mention in the introduction of our approach, a query case is exploited taking in consideration the similarity of queries in first instance, but also the cost estimation of preselected plans looking for that one that minimize the specified optimization objective.

4.2.2 Formalization

The previous prospective definition of query similarity specifies that **two queries Q_1 and Q_2 are similar** if they have similar Sources S_1 and S_2, and identical join conditions J_1 and J_2. The other query elements (i.e. projections) are not taken into consideration. If Q_1 and Q_2 comprise subqueries Sq_1 and Sq_2 respectively, the similarity function is applied recursively over such queries. Our similarity function (μ) is a conjunction of comparative functions over the query features, specifically its sources and its join conditions. It is specified below, where

$$\mu(Q_1, Q_2)=F_S(S_1, S_2) \wedge F_J(J_1, J_2)$$

The comparative function of similarity between sources is denoted by F_S, the comparative function of equality between joins is denoted by F_J. Next subsections define and formalize the similarity of sources, and the equality of joins.

The literature states that the absolute similarity model establishes a binary relation, i.e. $\approx: \alpha \rightarrow \alpha \rightarrow Bool$; where the similarity function ($\approx$) takes as parameters two elements of type α and returns a boolean value. This relation is reflexive ($x \approx x$ – or equivalently $x \approx x = true$), since any object is equivalent to itself, and symmetric ($x \approx y \Leftrightarrow y \approx x$), since if an object is similar to another, the reverse holds as well. In our approach, this relation is also transitive($x \approx y$ and $y \approx z$, thus $x \approx z$), thus if a query Q is similar to two queries Q_1 and Q_1, thus these two queries are similar to.

Similar Sources

For determining the similarity of sources we analyze their components. According our abstract representation of query, such components are: an Itemset (I), (ii) a scope (Sc), and (iii) a set of restriction conditions (Rc). **Two sources s_1 and s_2 are similar** if they concern the same fragments of the same Itemset(s), and the restriction conditions over such Itemset(s) are similar. We formalize our definition of query similarity using the set theory.The similarity between two *elements* is denoted by the symbol (≈) i.e. mathematical symbol of similarity.

$$s_1 \approx s_2 \text{ iff } I_1 = I_2 \text{ and } Sc_1 = Sc_2 \text{ and } Rc_1 \approx Rc_2$$

Where I_1, Sc_1 and Rc_1 correspond to the components of the source s_1; and I_2, Sc_2 and Rc_2 correspond to the components of the source s_2. **Two Itemsets are the equal** if they concern the same Itemset identifier. The fragments of an Itemset are equal if s_1 and s_2 have the same scope. A scope Sc is a set that comprises a set of node identifiers $Sc = \{n_1, n_2, .. , n_i\}$. **Two scopes Sc_1 and Sc_2 are equal** if both comprise exactly the same node identifiers: $Sc_1 - Sc_2 = \varnothing$ and $Sc_2 - Sc_1 = \varnothing$.

Two sets of restriction conditions Rc_1 and Rc_2 are similar if for any restriction condition r_i in Rc_1, exists a similar restriction condition r_j in Rc_2, and vice verse. The variables *i* and *j* belong to the domain $\mathbb{N}$ i.e. natural numbers.

$$Rc_1 \approx Rc_2 \quad \text{iff} \quad \forall \; r_i \in Rc_1 \;\; \exists \;\; r_j \in Rc_2 \;\; | \; r_i \approx r_j \quad \text{and,}$$
$$\forall \;\; r_j \in Rc_2 \; \exists \; r_i \in Rc_1 \;\; | \;\; r_j \approx r_i$$

Our definition of query similarity remains flexible when comparing restriction conditions allowing that two queries are similar if their restriction conditions are similar as well, but not necessarily equal. A restriction condition comprises: an attribute, a comparison operator, and a value or subquery, as explained in our abstract query representation. **Two restriction conditions r_1 and r_2 are similar** if they concern the same attribute (identifier) and the same comparison operator. $r_1 \approx r_2$ iff $attr_1 = attr_2$ and $co_1 = co_2$; where $attr_1$ and co_1 belong to r_1 and $attr_2$ and co_2 belong to r_2.

Finally, **two sets of sources S_1 and S_2 are similar** if for any source s_i in S_1 exist an equivalent (similar) source s_j in S_2, and vice versa. The variables *i* and *j* belong to the domain $\mathbb{N}$ i.e. natural numbers.

$$S_1 \approx S_2 \quad \text{iff} \quad \forall \;\; s_i \in S_1 \;\; \exists \;\; s_j \in S_2 \;\; | \;\; s_i \approx s_j \quad \text{and,}$$
$$\forall \;\; s_j \in S_2 \;\; \exists \;\; s_i \in S_1 \;\; | \;\; s_j \approx s_i$$

Equal Joins

If the set of sources of Q_1 and Q_2 are similar, then the set of join conditions binding such sources are analyzed. A join condition comprises a source-field from each of the both sources (left operand and right operand) to join, and the comparison operator, as previously defined in the abstract query representation. **Two join conditions j_1 and j_2 are equal** if they concern the same source-fields (identifier) and the same comparison operator:

$$j_1 = j_2 \quad \text{iff } sf_{1[left]} = sf_{2[left]} \text{ and } sf_{1[right]} = sf_{2[right]} \text{ and } co_1 = co_2$$

Where $sf_{1[left]}$ and $sf_{1[right]}$ and co_1 are the components of the join condition j_1; and $sf_{2[left]}$ and $sf_{2[right]}$ and co_2 are the components of the join condition j_2.

Finally, **two sets of Joins J_1 and J_2 are equal** if for any join condition j_i in J_1, there exist an equal join condition j_j in J_2, and vice versa. The variables *i* and *j* belong to the domain $\mathbb{N}$ i.e. natural numbers.

$$J_1 = J_2 \quad \text{iff} \quad \forall \;\; j_i \in J_1 \;\; \exists \;\; j_j \in J_2 \, | \, j_i = j_j \quad \text{and,}$$

$$\forall \quad j_j \in J_2 \quad \exists \quad j_i \in J_1 \mid j_j = j_i$$

4.3 CASE FOR A QUERY FAMILY

We take advantage of the query similarity function for clustering cases in families. A family comprises cases of similar queries. Thus, two query cases QC_i: <Q_i, P_i, M_i> and QC_j: <Q_j, P_j, M_j> belong to the same query family QF if Q_i is similar to Q_j, where $i \neq j$. The queries within cases from QF have some common features that we represent as a query template. The plan from any case within QF serves, with some settings, to evaluate any query that fits with such template. Such plan settings basically consist in replacing the parameter values of some plan operators, not worth to keep those parameter values for case storage; instead, the plan comprises empty parameters. Figure 4.3 depicts a query family.

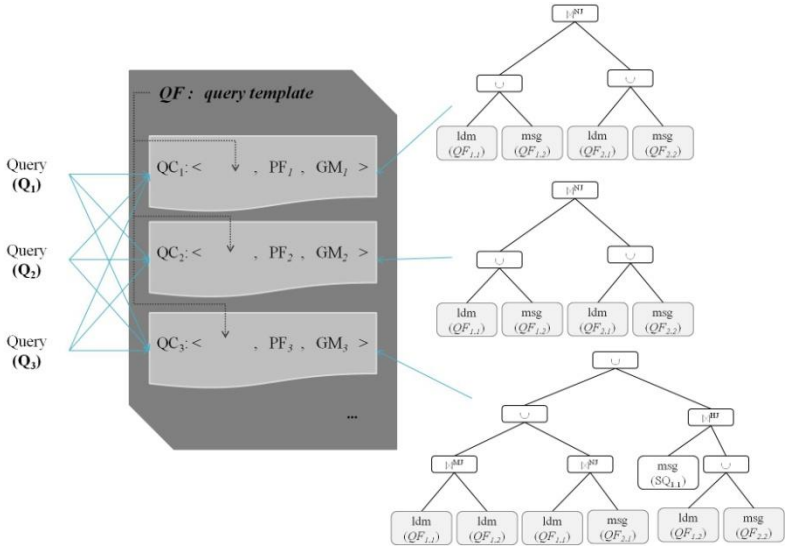

Figure 4.3. Alternative plan templates for solving a query family

The previous remarks bring us to review our original definition/representation of query case, since a case comprises query feedback to be exploited to improve the evaluation of a family of similar queries, and not only for a particular query. Thus, a slight variation of the case components is as follows: QC_i : <QF, PF, MG>; where QF is the query template representative of a *query family*. PF is a template of a physical plan (since some of its operators have empty parameters) that is used for evaluating any query Q that fits with QF. Finally, MG corresponds to the global measures of resources consumed during the evaluation of Q using the plan template PF. Section 4.1.3 explains the content and representation of global measures, where we have already considered that such measures represent the resources consumed by a plan executed for evaluating several similar queries.

We conclude that a query family can be seen as an explored part of the space of possible plans (search space) for evaluating a family of queries; in contrast with the classical query optimization, where the search space is the set of possible plans for evaluating a single query. Two query-cases that belong to the same query family never comprise the same plan (no duplicates of solutions for executing queries). The size of a query family is specified by a customizable threshold that indicates the number of cases within the family. Next sections define the query template and the plan template for a query family.

4.3.1 Query template

A query family is represented by a query template based in our abstract query representation. An example of query template for representing a query-family QF is presented below (Example 4.5):

Example 4.5 -

QF: query-template = <
Projections = {var_1: Expression}
Sources = {
S_P= <Positions P, {<Area, =, var_2:Value>}, {local, J, D}>,
S_S= <Servers S, {}, {global}>}
Joins = {<S_P. Area, = S_S. Area }
>

Such template has specific and variable query features. It specifies the query features that are considered in the query similarity definition. According to our prospective definition of query similarity, this query template specifies: (i) the itemset (or query resultset) and scope of the source (e.g. the Itemset Position, specifically the fragments stored at the local node, and at nodes J and D); (ii) the attribute and operator of the restriction condition(s) e.g. <Area,=,>, and (iii) the join condition(s). The variables of the template correspond to: (i) the projection expressions e.g. var_1, and (ii) the values of restriction conditions e.g. var_2. Specific and variable features change for different definitions of query similarity. Similar queries fit with this template; they have different values for the template variables.

4.3.2 Plan template

A case is exploited for evaluating a family of queries (not a single query only). Thus, any query that fits with the template of certain query family QF can be evaluated using the plan from any case QC_n that belongs to such family. Thus, this plan is not necessarily ad-hoc for solving a specific query Q; actually it serves for solving ay query similar to Q.

We introduce the notion of plan template PF to designate a plan for evaluating similar queries. A plan template has identical structure and the same operators than a query execution plan; the only difference is that some of its operators have empty parameters. Such parameters correspond to the query properties that are not considered by the similarity definition. According to our prospective similarity definition, these operators are the print operators, the data access operators, and the message operators. In P, the parameters of a print operator correspond to specific projection expressions. In a query plan, the print operator has empty parameters that are adjusted according to the projection expressions of different queries.

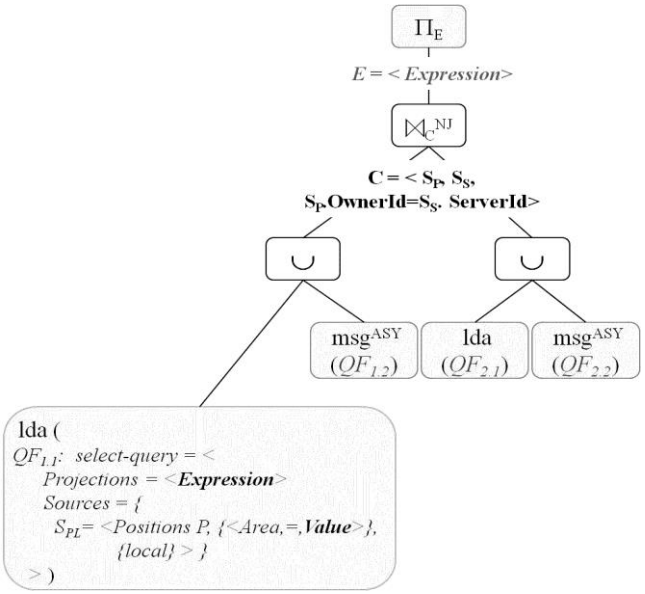

Figure 4.4. Example of plan template

The parameters of data access and message operators of a query execution plan correspond to subqueries of a given query. In a plan template, these operators receive templates representative of such subqueries. Figure 4.4 shows a local data access operator that receives as parameter the query template of a subquery, since some of its expressions and value conditions correspond to variables instead to specific values.

4.4 CASEBASE

In the CBR principle, the repository for storing cases is known as casebase or memory of knowledge. This section presents the structure of our casebase, and the organization and management operation that we propose for its maintenance and exploration.

4.4.1 Indexing

In this section we discuss the organization of the casebase for an efficient search and retrieval of cases. The organization of the casebase that favors the reasoning process depends on the application purposes. Some common approaches for organizing the casebase are:

- Flat lists. It is inefficient for big casebases because of it requires exhaustive search of cases. The implementation of such structure is quiet simple.

- Databases. This offers several advantages such as data security, data independence, data standardization and data integrity. However, this forces to represent cases as flat record of n-ary relations [ScBe00], in addition case retrieval is done through declarative queries e.g. SQL-like leading to our initial query optimization problem.

- Nets. The net is composed by case-nodes and entities-of-information-nodes (IE). Cases are composed by an undefined number of IEs. In this approach the casebase and case structure are bound, which compromises the representation of cases to a particular structure.

- Index structures. Indexes are used to quickly and efficiently provide the exact location of data without exhaustive search. B-trees and Hash indexes are well known structures. The index of case may be seen as a case summarization and must encode the main case characteristics; it is say that the index represents the part of the problem which plays the direct role in the generation of the solution [Wort97][Bobe10].

The organization of our cases repository is based on the use of indexes. We create an (primary) index access structure based on hashing; where cases are indexed according to families. The index entries are of type where *<QC, Pr>*, where Pr is a pointer to the case containing the case QC. Such structure defines categories, also known as *buckets* (i.e. query families), and category samples (i.e. index entries). Thus, our index structure comprises buckets that correspond to query-families; a bucket groups entries according to the query-family to which the case belongs. Figure 4.5 depicts a casebase, and the casebase index structure.

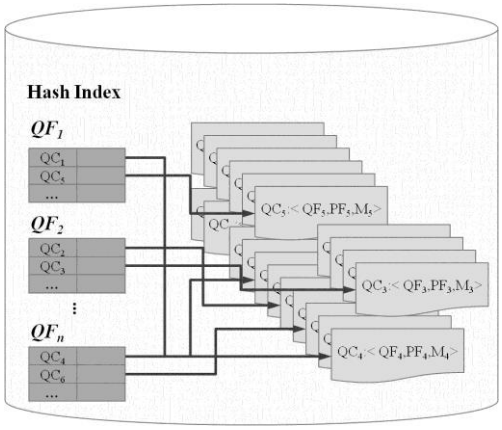

Figure 4.5. Casebase indexing

The idea behind hashing is to provide a function *h*, called hash function, to determine the bucket to which an entry belongs. In our proposal, this function correspond to the definition of query similarity (a bucket comprises cases of similar queries). The advantage of this approach is that there is possible to maintain different indexes, each of them grouping entries according to different similarity functions, and without the need to modify the storage of cases, since this it does not matter, but only the organization of its pointers. The cases may be organized in a simple flat list (e.g. array structure), facilitating the implementation of this part.

4.4.2 Management operations

The management operations of a casebase include storage, retrieving and deletion of cases. The former inserts new cases to the casebase or updates some of the already existent cases within the casebase; this depends of the kind query feedback that is obtained during the query execution. The second selects relevant cases for evaluating a given query efficiently. The latter deletes useless cases to improve knowledge accuracy and to avoid the casebase overflow.

Storage

For each query execution either, new cases are generated and inserted into the casebase, or some of the query-cases that already exist in the casebase are updated. This depends on the acquired feedback from the query execution:

- A *new query*. The casebase does not comprise cases of queries similar to a given query. In this situation, the casebase is extended adding a new query family QF. A new query case is generated and inserted in QF.

- A *new query plan.* The casebase comprises cases of queries similar to a given query Q, however a new plan is proposed for executing the given query. A new case is created and inserted into the corresponding query family.

- A set of *new global measures.* A case from the casebase is used for executing the posed query. Therefore, the measures of such case are updated taken into consideration the computational resources consumed during this last query execution using the plan template that it comprises.

The pseudo-algorithm presented below (Algorithm 4.1) exposes in straightforward manner the mechanisms for storing the different kinds of query feedback. It is required to insert a query case in two situations: (i) when a new query is evaluated, or (ii) when a new query plan is executed. In the first situation, inserting a case requires the preparation of the casebase to hold knowledge of a new type of queries. This preparation consists in generate a new query family QF_n; then the *bucket* corresponding to QF_n is added to the index. The case is stored and the pointer to such case is inserted into the corresponding QF_n *bucket*.

```
Inputs:
       String feedback: The type of query feedback
       QueryCase case: The case to insert or to update
       CaseBase casebase: The casebase
       Query query: the submitted query

Procedure:
QueryCaseStorage(String feedback, QueryCase case, CaseBase casebase, Query
query)
BEGIN
1      If (feedback = 'query')
2            QueryFamily family = generateQueryFamily
3            casebase.insert(family)
4            family.insert(case)
5      If (feedback = 'plan')
6            QueryTemplate queryTemplate = query.getTemplate()/*template of
7                                                        submitted  query*/
7            family = casebase.get(queryTemplate)/*searching within the
8                        casebase the family to which the case should belong*/
9            family.insert(case)
10     If (feedback = 'measures')
11           case.update(feedback)/*the used case temporally retains this
12                       feedback to recalculate its global measures*/
END
```

Algorithm 4.1. Storage (i.e. insertion or update) of a case within the casebase

For inserting a case generated from the execution of a new query plan, the first step is to explore the index for identifying the query family to which the case should belong. To identify such

family we use a straightforward algorithm that analyzes the features of the posed query. The output of the algorithm is a string of characters that corresponds to the template of the query. Such string must correspond to the identifier (i.e. template of query family) of an index bucket. Once the query-family has been identified, the insertion process finishes by adding the new case within the casebase array structure, and inserting the corresponding case pointer within the casebase index structure.

Updating a case specifically consists in updating its global measures. Thus, after a case has been used, the new global measures are temporally retained while recalculating the current global measures. This update method depends on the type of measures that have been decided to retain within cases. For example, global measures may correspond to computational resources consumed at the last plan, thus measures update consists in replacing the current case measures by the new ones. Another example of measures may be aggregated measures (e.g. the average); measures update consists in aggregating the new measures to the up to now measures within the case. Finally, if the measures are a little more complex, such as histograms, the new global measures are just added as new histogram values.

Retrieving

An important step of the CBR based optimization process is the retrieval of cases. Effective retrieval is the matter to find the query plan template that optimizes the execution of a given query. Our retrieving process is founded on the query similarity, and on the query plan *optimality* assessments. The pseudo algorithm of our retrieving process is presented below (Algorithm 4.2).

```
Inputs:
Query query: The new query to solve
CaseBase casebase: The casebase

Outputs:
QueryCase optimalCase: The query case that have the optimal plan template


Procedure:
QueryCaseRetrieving(Query query, CaseBase casebase)
BEGIN
1  /* Retrieving cases with similar queries */
2      QueryFamily family = casebase.get(query)
3      if(family.size() ≥ thresoldOfKnowledge)
4  /* Retrieving the case with the optimal query plan */
5      Double optimalCost = max-value
6      while (family.hasNext())
7             QueryCase case = family.next()
9             Double queryPlanCost = computingCost(case)/*using a given cost
10                                                       function*/
11            if(queryPlanCost < optimalCost)
12                   optimalCost = queryPlanCost
13                   optimalCase = queryCase
14  return optimalCase
END
```

Algorithm 4.2. Exploration of the casebase and retrieving of cases

This approach takes advantage of the organization of cases in the casebase, and on the index structure for accelerating the access to cases. The case retrieving process is composed of two steps: (i) a pre-selection step that retrieve those cases that comprise queries similar to Q, and (ii) a selection step that retrieves the among the pre-selected cases the case having the query plan that minimizes a given cost function. The second step is relevant only if there are enough cases containing similar queries.

Retrieving cases with similar queries. This step consists in finding the cases that comprise queries similar to the query to be evaluated. This step is favored by the organization of the casebase, since it is enough to determine the query family to which the sought query cases belong. Recall that, all cases having similar queries are stored in a single query family.

This straightforward process is an efficient way for detecting the cases that comprise all the learned query plans for executing the posed query, but not all of them are necessarily optimal according to a specific optimization objective. Nevertheless, the search space within the casebase has been reduced to a specific area. In the next retrieving step the searching for the optimal query plan will consider such reduced area only.

Retrieving the case with the optimal query-plan. This step consists in finding among the pre-selected cases the one that minimize the result of a given cost function. Such cost function is a mathematic formula that reflects an optimization objective, and that receives as parameters the measures of consumed computational resources (e.g. time, CPU, memory, energy, communication). The cost function may comprise a single measure, or a combination of them. For instance, the admittedly simplistic linear weight function presented below:

$$\text{OverallCost} = \langle\text{ExecutionTimeCost}\rangle + \langle\text{MemmoryCost}\rangle + \langle\text{CommCost}\rangle$$

Where $\langle\text{ExecutionTimeCost}\rangle = \alpha * t$, $\langle\text{MemoryCost}\rangle = \beta * cpu$, and $\langle\text{CommCost}\rangle = \gamma * msg$. The coefficients: α, β and γ correspond to the weights associated to the parameter measures. Examples of such measures are: $t{:}\langle\ time,\ 150,\ ms\rangle$, $m{:}\langle memory,\ 200,\ kB\rangle$, and $msg{:}\langle messages,\ 2,\ integer\rangle$ saying that in this example the communication cost is measured in function of the number of exchange messages. In this function as more weight is associated to a parameter as more important is the optimization of the corresponding resources. The optimization objective is customized simply modifying the coefficients of weight in the cost function. We assume that exist other more elaborated cost functions that define different optimization objectives. The definition of a cost model is not one of the major aspects to address in this thesis.

For better understanding the use of global measures to select the case having the query plan that minimizes a cost function let us analyze an example. For this example we suppose that the global measures of cases correspond to the average of computational resources consumed at several executions of their plans. We consider that measures average reflect the general behavior of a query plan; could be logic to think that as lower/higher are the average measures as less/more resources were consumed during the multiple executions of a plan. However, this is not always true as is shown next.

Given a query Q that comprises the condition c : <Area, = , *'value'*> where '*value'*=7, and the two cases QC_1 and QC_2 that belongs to the set of cases retrieved in the pre-selection step; let us consider that both query-cases have been already used for executing queries assigning three different values to *c*, as shown in the table below.

Case / value	v_1=5	v_2=7	v_3=3
QC_1= 131,6 ms	128 *ms*	115 *ms*	152 *ms*
QC_2= 124,6 ms	98 *ms*	147 *ms*	129 *ms*

Figure 4.6 Global query execution time

In this table, the columns denote the values of the query condition, and the rows denote the case used for evaluating the query. The intersection of columns correspond to the time for executing a query with the condition value v_i, and using the case QCi. Let us use the cost function presented before assigning the coefficient values $\alpha=1$, $\gamma=0$ and $\beta=0$. Thus, the cost function is simplified to `OverallCost = <ExecutionTimeCost>`. The result of such cost function is equal to the average (i.e. the global measure within cases) of the execution time.

Thus, if we take into consideration the average of measures for evaluating the plans cost, the retrieved case should be QC_2. However, while QC_2 has the plan that has minimized the execution time for the most of queries and that in consequence has the minimal time average; the case QC_1 is the one that minimizes the time for the particular value of Q. This fact may lead to select cases with sub-optimal plans. Other forms of global measures may also been used; for example histograms, which register the consumption of computational resources at each execution of a plan (instead of aggregating them in a single value).

Deletion

The query-case deletion process eliminates useless cases from the casebase. The final goal is to reduce the size of the casebase; this avoids the overflow of the casebase and accelerates the case retrieving process. Deletion policies in CBR correspond to selective retention filters. For instance elimination of redundancy and of knowledge producing unsuccessfully problem solving results. Such policies define when and how trigger the deletion action.

```
Inputs:
QueryFamily family: The query family that has reached its maximal size
Integer threshold: A minimum number of case usages

Procedure:
QueryCaseDeletion(QueryFamily family, Integer threshold)
BEGIN
1
2      while (family.hasNextCase){
3             QueryCase case = family.NextCase()
4             Integer caseUsages = case.getCounter();
5             /*If the case does not reach the minimum number
6             of usages specified by the threshold*/
7             If (caseUsages < threshold)
8                    family.delete(case.getIndex())
9             }
10     }
END
```

Algorithm 4.3. Deletion of useless cases

Our policy triggers the deletion of cases when a query family has reached a maximal size (i.e. threshold of knowledge). When this occurs, a set of cases within such a query family is deleted. The maximal number of cases may be different for each family (e.g. the size of families with cases of mono-source queries should be smaller than the size of families with cases of multi-source complex queries.

The deleted cases are those that have been less used up to now for queries evaluation. This principle is based on the competence principle [Bobe10], that evaluate the utility of individual cases for problem solving. A case is annotated with a counter of the times that it has been used for queries

evaluation. The rare use of a case is due to it plan has reported high measures of consumed computational resources.

The amount of cases to delete at each family may be specified in several ways, for example using a percentage of (e.g. 30%) in order to maintain a proportion between the number of deleted cases and the size of each query family. Another option is to delete the cases that do not reach a minimum number of usages; a prospective algorithm based on this approach is presented before. Whatever the threshold that indicates the cases that must be deleted, for deleting a case the casebase is explored exhaustively to find and delete the less used cases. This action stops when the casebase has been reduced in the number of cases specified in the percentage threshold.

4.5 CONCLUSIONS

In this chapter we specified the representation of a query case, and the representation of its components: (i) a query, (ii) a query plan, and (iii) global measures of resources consume during the plan execution. We based our query representation in the DLAQL query language; however our proposed representation can be easily adapted to SQL-like query languages. Such representation aims to simplify the comparison process to determine the similarity between queries. Also, we recalled the representation of a query plan as a tree structure, and exposed the content and representation of global measures (different forms of queries may be used).

We defined a similarity function between queries. Such similarity function allows us to generalize our case representation, as a case with feedback for solving a family of similar queries, instead to a particular query only. The casebase indexes cases in groups of families, this improve the operations for casebase management. In particular, the case retrieving process benefits from this casebase organization that accelerates the access to cases. Thus, the case retrieving process is decomposed in two steps, a pre-selection step that retrieves the cases with similar queries (family), and the selection step that computes the cost of plans using a given cost function. Such cost function is based on the global measures, which are essential to improve the accuracy of retrieving useful cases for solving a given query.

The management of the casebase is fundamental to avoid its overflow and maintain up to date query cases. The management operations include storage and deletion of cases. The storage operation comprises insertion and update of cases, this depends of the feedback (i.e. new query, new query plan, or new global measures) gained from the execution of a query. The deletion operation eliminates useless cases. The cases are annotated with a counter of the times that they have been used. A customizable threshold specifies the amount of cases that must be deleted (e.g. specific number of cases, a percentage over the total number of cases within a family, the cases with a counter lower than some number of usages, etc.).

So far we have presented the concepts and management mechanisms of our query optimization process. However, the mechanisms for the generation of plans and exploitation of query cases have not been addressed yet. Chapter 5 is consecrated to detail the query plan generation process.

5. QUERY PLAN GENERATION

This chapter presents our query plan generation process. Section 5.1 presents the general principle of such process that generates a query plan by retrieving a query-case and setting it according to the new query specifications. If no useful case is retrieved, we propose query plan generation process by applying well known query optimization heuristics and taking some random decisions (when classical query optimization techniques used metadata for estimating cost functions), for example the order of some operators, as well as their implementation algorithms. Section 5.2 details our pseudo-random top-down query plan generation, and Section 5.3 exposes our query plan setting process.

Finally, Section 5.4 proposes a query plan summarization-regeneration process where a physical query plan is compressed to minimize the amount of memory required for storing a query-case. The compact representation of a query plan is termed plan signature. The inverse process (i.e. plan regeneration) is to generate a physical query plan from a plan signature. Section 5.5 concludes the chapter.

5.1 PRINCIPLE

Figure 5.1 depicts our query plan generation process through a flowchart. The input of the query plan generation process is a query expressed according to our query representation (see Chapter 4) and in terms of global sources. The generation of query plans for nested queries is independent and according to a specific scheduling (see Chapter 3). Basically, plans for low-level nested queries are generated first; such queries are evaluated and replaced by their resulting values before the generation of the top-level queries.

The query Example 5.1 presents a query for obtaining information of all the avatars located at the area 7, such query can be posed at any node and the system must globally executed it. This query will be used to illustrate our query plan generation process in the remainder of this chapter.

Example 5.1 - $Q_{EXAMPLE}$:join-query = <
Projections = {*}
Sources ={S_P= <Positions P, {P.Area= 7},{local,J, D}>
S_S = <Servers S, {}, {global}>}
Joins ={S_P.OwnerID = S_S.ServerID}> >

Our query plan generation process comprises the following steps: (i) query-case retrieving, (ii) query-plan pseudo-random top-down generation and (iii) query-plan setting. Given a query Q, the

casebase is explored for retrieving a useful case (i.e. *Query-case retrieving*). When the process retrieves a case QC, the query plan generation process consists in setting plan template PF from the retrieved case according to the specifications of Q (i.e. *Query-plan setting*), the output of the process is the query execution plan P.

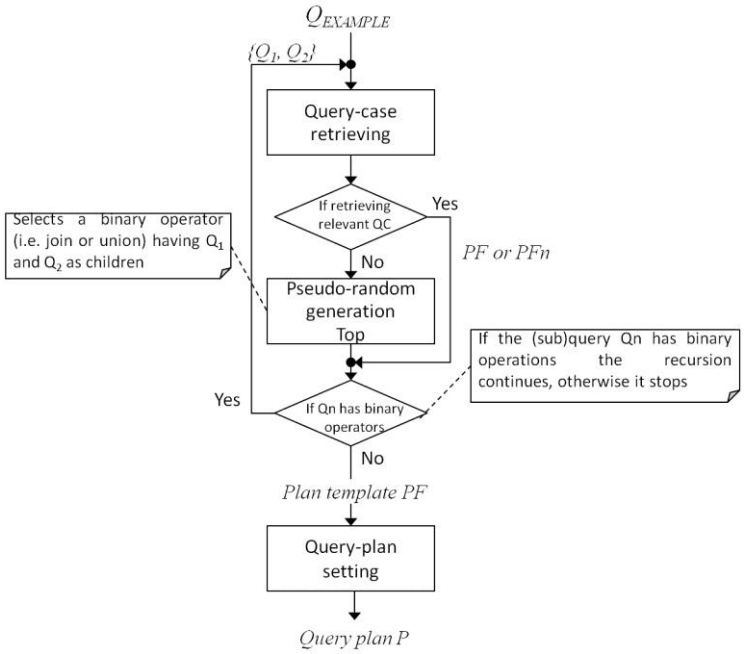

Figure 5.1. Query plan generation process flowchart

Please recall that cases are grouped in families; a family comprises cases generated from the execution of similar queries. Thus, a case is a triplet of the form QC:<QF, PF, GM>, where QF is a query template (comprises the common features of similar queries) representative of the query family to which the case belongs, PF is a template of query plan that can be set to any query that fits with the query template QF. PF has physical operators with some empty parameters that are adjusted during the plan setting process for generating the query execution plan P. Figure 5.2 depicts an example of a query execution plan for evaluating our query example.

When no relevant case is retrieved, a recursive process starts interleaving the *Pseudo-random top-down generation* and *Query-case retrieving* steps. The output of this recursive process is a plan PF. In this step, a binary operator (i.e. join and union) is selected by pseudo-random; and Q is decomposed in two subqueries Q_1 and Q_2. The selected binary operator is added to PF as a physical operator, the subqueries correspond to its operands (e.g. <subquery> $\triangleright\triangleleft_c^{NJ}$ <subquery>). Section 5.3.3 presents the query decomposition rules. Please notice that these subqueries are produced internally to the optimization process, thus are different from the nested queries (i.e. subqueries) at the original query expression (i.e. user query), which are independently optimized according to a specific schedule (see Chapter 3).

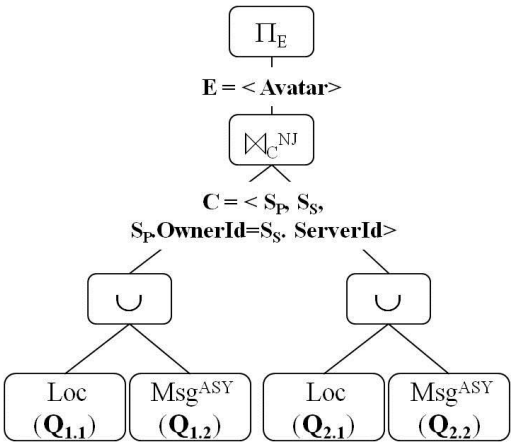

Figure 5.2 Alternative query execution plan

The query-case retrieving process now searches relevant cases for Q_1 and Q_2, and so on. Let us consider our query sample Q, which comprises a join operation and two global sources (i.e. Positions and Servers). A global source corresponds to a union operation between subqueries for accessing local and distant source fragments. Thus, the optimizer may choose between the join operator, the union operator concerning to the global source Positions, or the union operator concerning to the global source Servers. Let's say that the optimizer selects the join operator, the resulting partial plan template is shown in Figure 5.3.a.

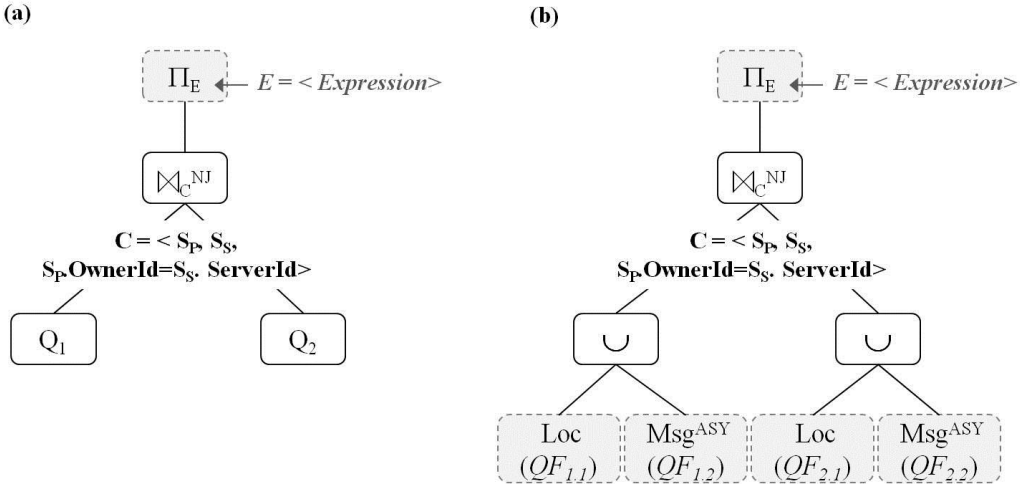

Figure 5.3. A plan template in different generation steps

While Q_n has at least one binary operator (i.e. join or union) the recursion (some branch) continues; generating part of the plan template by pseudo random and searching cases for the resulting subqueries. When a case for some subquery Q_n is retrieved, the plan template PF_n from such case (that solves the subquery Q_n) is a construction block. In other words, it is a subplan that is reused as part of the complete plan template for solving Q. Figure 5.4 depicts an example of a plan template that were generated by reusing a subplan PF_n.

The generation of the query plan template is accomplished when there is no more subqueries with binary operators, as shown in Figure 5.3.b. Thus, all the leaf nodes of the tree are unary data-access (local, distant or program invocation) operators. Once the plan template is completed, it is set

according to the specifications of the query during the plan setting process; the output is the plan for executing the query. The corresponding plan template is kept within a new case to be reused for further queries.

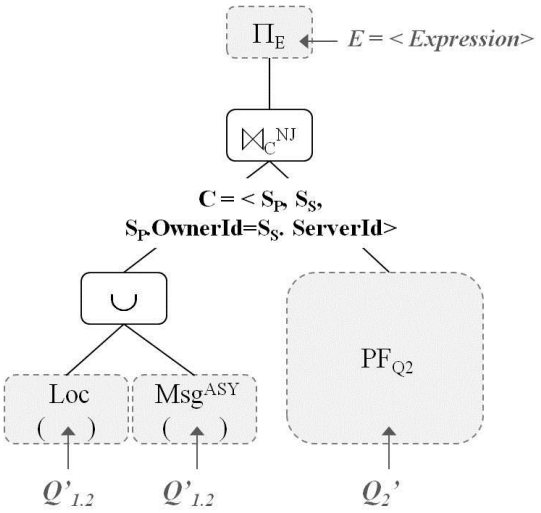

Figure 5.4. A plan template that reuses a query case

Our query plan generation process may generate or update several query-cases; one that correspond to the whole query Q, and others that correspond to the subqueries resulting from query-plan generation process. A new query-case is generated when a new plan template is generated by applying the pseudo-random process. A query-case is updated when a plan is reused.

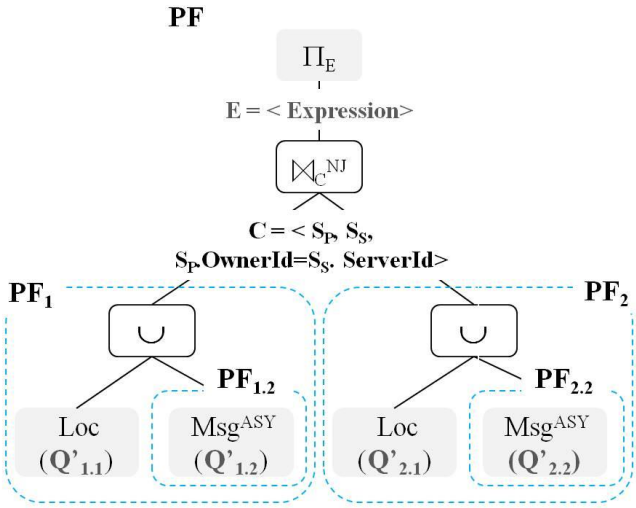

Figure 5.5 A plan template and its subplans

Thus, the generation of the plan template in Figure 5.4 triggers the generation / insertion of the new case QC:<QF, PF, M> within the casebase; but also of the query cases QC_1:<QF_1_id, PF_1, M_1> for the subquery Q1. Also, the already existent case QC_2:< QF_2_id, PF_2, M_2> (used as subplan for the construction of PF) must be updated. Figure 5.5 shows a query plan template and its subplans. Chapter 4 presents the algorithms for storing (i.e. insert or update) cases within the casebase. The general pseudo-algorithm of our query plan generation process is presented below (Algorithm 5.1).

```
Input(s):
Query query: A given query
Output:
QueryExecutionPlan P: A query executable plan
Procedure 1:
Plan_generation(Query query)
BEGIN
1      PlanTemplate PF /*Plan template*/
2      PF = generating_PF(query, PF)
3      QueryExecutionPlan P = setting(PF, query)/*Pseudo algorithm in Section
4                                                5.4*/
5      return P
END

Input(s):
Query query: A given query
PlanTemplate PF: Plan for a family of queries
Output:
PlanTemplate PF: Plan for a family of queries
Procedure 2:
generating_PF(Query query, PlanTemplate PF)
1  QueryFamily family = query.getQueryFamily()
2  QueryCase case = retrieving(family)/*Reuse of PF from retrieved case
3                                  detailed algorithm in Section 4.4, Chapter 4*/
4  If (retrieving)
5      return PF
6  else
7      PF = pseudo_random(Q, PF)/*Partial generation of PF by pseudo-random,
8                                  detailed algorithm in Section 5.3*/
9      Query Q1 = PF.getLeftQuery
10     Query Q2 = PF.getRightQuery
11     If(Q1.hasBinaryOp)
12            PF = generating_PF (Q1, PF)
13     else
14            return PF
15     If(Q2.hasBinaryOp)
16            PF = generating_PF (Q2, PF)
17     else
18            return PF
19     return PF
END
```

Algorithm 5.1. Query plan generation process

In summary, a query execution plan is generated during a recursive process that integrates a pseudo-random exploration of the search space (i.e. pseudo-random top-down generation technique) and the exploitation of query cases (i.e. query-case retrieving and plan setting). The query case retrieving process has been addressed in Chapter 4. This chapter presents in detail the pseudo-random plan generation and plan setting processes in Section 5.2 and Section 5.3 respectively.

5.2 PSEUDO-RANDOM TOP-DOWN PLAN (TEMPLATE) GENERATION

This section addresses the second step of our plan generation process, the pseudo random-random generation. It comprises two steps: (i) selects a binary operation of a given query Q, (ii) decompose Q in two subqueries Q_1 and Q_2 by applying rewriting rules; such subqueries are the operands of the selected operator. Section 5.3.3 presents the rules for rewriting a query from the selection of a join or a union operator. The application of such steps generates a partial plan template. These steps are recursively applied for a top-down generation of a full plan template as shown in Figure 5.3; the last recursion occurs when all the left nodes of the plan are unary operators.

This generation process considers classical optimization heuristics for reducing the size of intermediate results. Such heuristics include: (i) applying selection conditions first, (ii) applying projections as soon as possible, and (iii) avoiding Cartesian products. When classical optimization uses metadata (e.g. selectivity factor for ordering join operators) this process does random decisions, specifically for the selection of binary operators (thus the ordering of joins and unions), and for the selection of execution techniques to implement the plan operators. The pseudo-algorithm of our pseudo-random process for generating a plan template is presented below (Algorithm 5.2).

```
Input(s):
Query query: A given (sub) query
PlanTemplate PF: (partial) Plan for a family of queries

Output:
PlanTemplate PF: (partial) Plan for a family of queries

Procedure:
pseudo_random(query, PF)
BEGIN
1    Operator binaryOp
2    binaryOp = query.getBinaryOp() /*random selection of a binary operation*/
3    Operator o = generatePhyisicalOp(binaryOp)
4    Vector <Query> subqueries = query.decompose(binaryOp) /*decomposition of Q
5                                 according to rules in subsection 5.3*/
6    Operator Q1 = subqueries.get(0)
7    Operator Q2 = subqueries.get(1)
8    o.leftChild(Q1)
9    o.rightChild(Q2)
10   PF.add(o) /*add to the current PF the block: binary operator,
11             Q1, Q2*/
12   Return PF
END
```

Algorithm 5.2. Pseudo-random top-down generation process

A plan template is generated by pseudo-random in the three following situations: (i) when the casebase does not comprise relevant cases for a given query Q (nor for subqueries of Q), (ii) during the learning process, when the casebase is supplied with query-cases until reach a customizable threshold of knowledge (i.e. number of cases by query-family), and (iii) to continue learning; even if the threshold of knowledge for a query family has been reached, a low percentage (e.g. 10%) of the plans are still generated by pseudo-random to try other alternative plans and this continue the search space.

5.2.1 Heuristics and random decisions

Applying heuristics

Classical query optimization techniques apply the mentioned heuristics by pushing-down selection conditions and distributing projection expressions over the leaf-nodes (i.e. data-access operators) of an algebraic query tree [ElSh11]. The leaf nodes of our plan are data-access operators or communication operators. The access-operator retrieves local data by executing (at the current node) a subquery resulting from the top-down decomposition of the query Q. The communication operator sends through the network some of such subqueries to be executed at some remote node.

According to our query representation, these subqueries comprise: (i) a set of sources, (ii) a set of join conditions and (iii) a set of projection expressions. Each source comprises the selection conditions that it concerns as follows: <source>:= (<fragment>, [<select-condition>*], <scope>). The selection conditions are applied at the moment that a fragment of data is accessed. To fulfill the second heuristic, our query decomposition distributes the projection expressions according to the sources that a query comprises (our query representation is detailed in Section 4.1.1, Chapter 4).

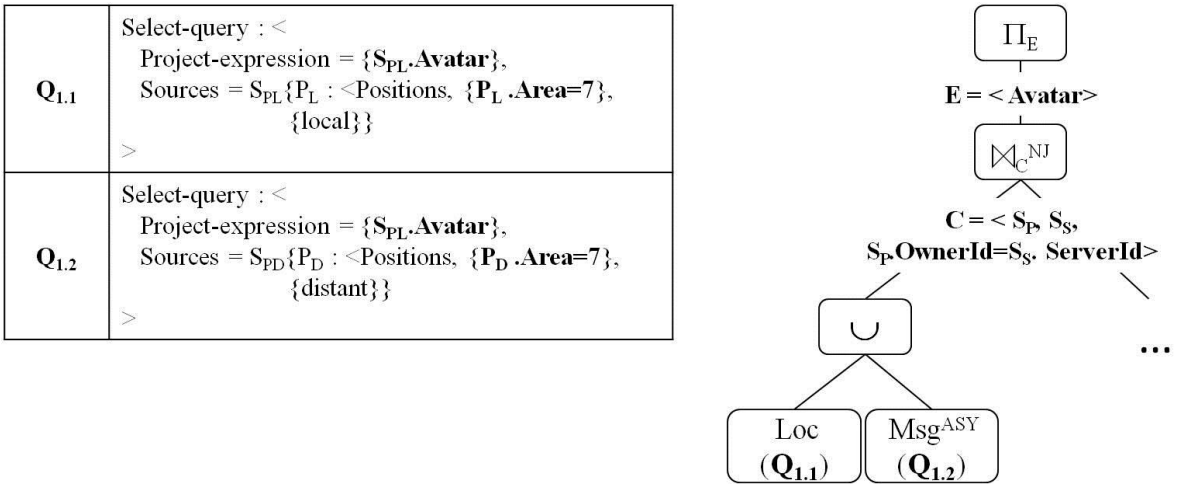

Figure 5.6. Example of query-plan and subqueries

Figure 5.6 depicts part of the query execution plan for Q (right side), and the subqueries (left side), parameters of its data-access and communication operators. For instance, $Q_{1.1}$ is a query executed at the local node, the access to the source is delegated to the local data management system; $Q_{1.2}$ is wrapped in a message and sent through the network to be distantly executed at some remote node(s). Such subqueries filter sources data ipso facto at the moment and at the node where they are executed, achieving the minimization of partial results (and naturally, data exchanging through the network).

Selecting binary operators

The order of binary operators (i.e. join and union) is one of the main concerns in the optimization of a query. From their ordering depends the set of subqueries resulting from decomposing a given query. Naturally this also affects the necessary computation and exchange of partial results. Select the optimal order for these operators is a hard problem. Join and union operators can be ordered in several ways due to their commutative and associative properties [ElSh11][Liu00].

Our pseudo-random plan generation process selects binary operators by random, as exposed before. The rational of this random selection is due to the lack of information on data for the prior costs estimation, as used to be in classical query optimization techniques. Remain that the pseudo-random plan generation is a process for learning the computational resources that a query plan demands during its execution, and reuse this knowledge for optimizing further queries (i.e. progressively discard query-plans with poor performance and/or detect promising query-plans).

This idea approximates the basic principle of genetic algorithms [OwKS05], where a preliminary set of query-plans is randomly created, such set corresponds to the "zero" generation of solutions. Then, each next generation is determined by propagating the solutions that minimize a given cost function for the next generation, combining some query-plans propagated in the previous step, and randomly altering some others. Our query optimization process shares some of these ideals when exploiting query-cases from the learning process.

Chapter 2 exposes a wide range of proposed search strategies; all of them put special attention on the order of binary operators (joins particularly). Seminal search strategies, such as deterministic strategies [SACL79] [KoSt00] [SwGu88][ShYT93][LVZC91], and randomized strategies [IoKa90] [StMK97] [IoWo87] [SwGu88] fully depend on specific information on data (e.g. location and cardinality of data sources, data values distribution, etc.). As discussed so far in this thesis, there is no guarantee that such information is available in a highly distributed data systems. Therefore, some distributed query optimization approaches simply assume that the required metadata exists, examples of such works are presented in [OuBo04]; others use feedback from queries execution for updating and tuning some metadata [SLMK01][AbCh99b]. More dynamic optimization techniques proposed to operators re-ordering during query execution [KaDe98][INSS97][IFFL99][AvHe00] [UrFA98]; they consider real-time parameter values pursuing to a query re-optimization process; however they use them and then forget them (instead of learning from them).

Selecting execution techniques

Query execution plans are typically abstracted by means of algebraic operator trees, which define the order in which the operations are executed. They are enriched with additional information, such as the best execution techniques for each operation. Our approach does not include this abstraction, our query plan generation algorithm selects the operators ordering (i.e. pseudo-random selection of binary operators), and the execution technique for its implementation (Figure 5.6 depicts a nested-join $\bowtie^{NJ}$) at one stroke.

Selecting the execution techniques for implementing the operators in the query plan is mostly made by random; but when possible, we consider the advantages and disadvantages of distributed query execution techniques that the database literature has compiled, especially those of distributed join algorithms [Koss00]. For example, the parallel execution of join operands often leads to a faster execution; however a sequential execution can minimize the number of exchanged messages. Experimental work indicates that the semi-join is not the best option for standard (e.g. relational) distributed databases because of the additional computational overhead is usually higher that the savings in communication cost. However there are applications that involve tables with very large tuples where the semi-join technique can indeed be very attractive [SKBK00]. The advantages of the hash-join [WiAp93] algorithm (and its variants [IFFL99]) is the delivery of query partial result as early as possible and the fully exploit of parallel pipeline reducing the overall response. It is an unfavorable technique for optimizing the memory usage.

The optimization process ought to consider this information, in such a way to apply the most promising execution techniques according to the optimization objectives. It is hard to know which is the best combination; there are also many other variants that affect. In our query optimization approach, we learn how the different strategies work, and eventually consider the obtained knowledge for further optimization decisions. The exploitation of query knowledge (i.e. query-case) is presented in detail in Section 5.4.

5.2.2 Localization

For query plan generation we need to consider the localization of data to be queried. In our approach, data localization is determined by analyzing the scope of source(s) comprised by a given query. Such scope indicates the nodes storing the fragments of sources that must be queried (see our representation of queries in Chapter 4).

The scope of a source may be: (i) local (i.e. fragment of S stored at the local node only), (ii) distant (i.e. fragments of S stored at remote nodes but not at the local node), or (iii) global (i.e. fragment of S stored the local node, but also fragments of S stored at remote nodes). The distant fragments may correspond to those stored in all the nodes of the system, or fragments from specific nodes indicated in a list of the corresponding node identifiers.

A query may be evaluated locally, distantly or globally depending on the scope of its sources. Therefore, according to the localization of data to be queried, there are three situations to consider in our pseudo-random top-down plan generation process:

1. If the source(s) of Q has (have) local scope only, the query must be locally evaluated. The query Q corresponds to the parameter of a local data access operator i.e. Loc(Q). Such operator is attached to the current plan. This also implies that that branch of recursion has been finished (part of the plan). Let us suppose that the query in our Example 5.2 has sources with local scope only; the plan comprises an operator for querying the local data, and a print operator for returning the query result.

2. If the source(s) of Q has (have) distant scope only, the query must be remotely evaluated. The query Q correspond to the parameter of a message operator i.e. Msg(Q). A message operator is added for sending the query through the network to the pertinent node(s). This also implies that that branch of recursion has been finished (part of the plan). If the query in Example 5.2 should be evaluated distantly only, the generated query plan should comprise massage operator as leaf, and a print operator for returning to the user the query result. The optimizer selects the dissemination algorithms for the message propagation.

3. If some source(s) of Q (at least one) has (have) global scope, the query must be globally evaluated. In this case, another recursion of the plan generation is carried out, by randomly selecting a top binary operator (and is execution technique) and decomposing the query once more, and so on. Let us recall that a source with a global scope corresponds to the union of subqueries for accessing local and distant source fragments.

5.2.3 Query decomposition

The selection of a binary operator during the query plan generation involves the decomposition of the original query in two subqueries. Each subquery corresponds to an operand of the selected binary operator. This section explains the query decomposition rules from the selection of union and join operators. The query in Example 5.2 is used in the remainder of this section to illustrate our explanation of the decomposition of queries.

Example 5.2 - Q:

```
join-query = <
  Projection-expression = {Avatar},
  Sources = {
     SP= <Positions P, {<Area, =, 7>}, {local, J, D}>,
     SS= <Servers S, {}, {global}>}
  Joins = {<SP.Area, =SS.Area>}
>
```

Decomposition rule for union operation

Until now we have talk about the selection of binary operators i.e. join and union. While the join operation is explicitly included in the specification of the query; this is not the case for the union operator. The union operator derives from decomposing a given query in terms of the fragments of sources to be queried. Thus a query that comprises global source(s) is decomposed in subqueries expressed on source fragments and operators (typically unions) for materializing a global source from the subquery partial results.

Decomposing a global query in terms of fragments used to be a pre-optimization process; in our approach it is included within in the query plan generation. Thus, the pseudo-random process carries out the localization of the query (if required) at one stroke while selecting the when selecting the operators ordering. When a binary operator is selected, actually the optimizer decides between a global source and a join operator. The advantage of this approach is that once an operator is selected, its position is preserved in the remainder of the plan generation. Classical query optimization techniques receive as input a localized query, where the lefts of the algebraic query tree correspond to the operators for global sources materialization. During the query plan generation process some heuristics (i.e. pushing up union operators) are applied to re-order the operators of the received tentative plan [OzVa11].

Thus selecting a global source implies to add a union operator to the query-plan. Such operator unites partial results from local and distant data fragments. Thus, the first step is to decompose the concerned source in its local/distant fragments. Considering the sample query Q, the result of decomposing the source S_P is as follows:

```
SPL= <Positions P, {<Area, =, 7>}, {local}>
SPD= <Positions P, {<Area, =, 7>}, {J, D}>
```

Rule 1. The query Q is duplicated in Q_1 and Q_2. The source S_P in Q_1 is replaced by the source S_{PL} (the local fragment of S_P). The source S_P in is replaced by the source S_{PD} (the distant fragment of S_P). The resulting queries Q_1 and Q_1 correspond to the operands of the union operator.

Q_1:

```
join-query = <
      Projection-expression = {Avatar},
```

```
        Sources = {
                $S_{PL}$= <Positions P, {<Area, =, 7>}, {local}>,
                $S_S$= <Servers S, {}, {global}>}
        Joins = {<$S_P$.Area, =$S_S$.Area }
        Subqueries = {}
>

Q2:  join-query = <
        Projection-expression = {Avatar},
        Sources = {
                $S_{PD}$= <Positions P, {<Area, =, 7>}, {J, D}>,
                $S_S$= <Servers S, {}, {global}>}
        Joins = {<$S_P$.Area, =$S_S$.Area }
        Subqueries = {}
>
```

Figure 5.7 presents the plan generated at this step. This is an incomplete plan, since Q_1 and Q_2 comprises binary operators, which means that (at least) another recursion of the plan generation process must be carried out. Let us recall that the process stops when subqueries concern to a single source, thus comprising unary operators only.

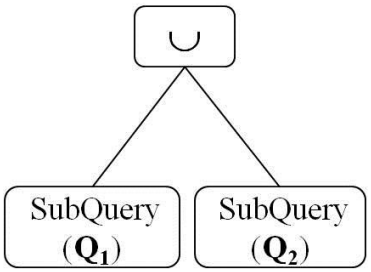

Figure 5.7 Query plan after query decomposition by union selection

Decomposing rule for join operation

Let' say that the optimizer selects the join of Q instead of the source with global scope. We add some components to Q to illustrate the example.

```
Q = join-query = <
        Projection-expression = {Avatar},
        Sources = {
                $S_P$= <Positions P, {<Area, =, 7>}, {local, J, D}>,
                $S_S$= <Servers S, {}, {global}>
                $S_C$= <C, {...}, global>
                $S_D$= <D, {...}, local>
                $S_E$= <E, {...}, local}>
        }
        Joins = {
                $J_{PS}$=<$S_P$.Area, =$S_S$.Area>
                $J_{SC}$ = <S, C, $jcond_{BC}$>
                $J_{CD}$ = <C, D, $jcond_{CD}$>
                $J_{DE}$ = <D, E, $jcond_{DE}$>
```

```
        }
        Subqueries = {}
>
```

Rule 2. The components of the query Q are split in two subsets Q_1 and Q_2, in such a way that Q_1 comprises the source S_1 and any other component of Q directly or indirectly related to S_1. In the same way, Q_2 comprises the source S_2, and any other component of Q directly or indirectly related to S_2. No component of Q belongs to the two generated queries at the same time $Q_1 \cap Q_2 = \{\}$. The resulting queries Q_1 and Q_1 correspond to the operands of the union operator.

An indirect relation between query components is defined by transitivity, such that $X \rightarrow Z$ because of $X \rightarrow Y$ and $Y \rightarrow Z$. For instance, let's say that the optimizer selects the join J_{CD}. In our query example, the source S_C is directly related to J_{CD} as a join operand. In the other sense, it is also true that J_{CD} is directly related with the source S_D as a computing operator over its data. Therefore we can say that $S_C \rightarrow S_D$, because of $S_C \rightarrow J_{CD}$ and $J_{CD} \rightarrow S_D$.

The components of each subquery result from an extensive transition including all the query components. The transitions for generating Q_1 are: $S_C \rightarrow J_{SC} \rightarrow S_S \rightarrow J_{PS} \rightarrow S_P$. The transitions for generating Q_2 are: $S_D \rightarrow J_{DE} \rightarrow S_E$. This rewriting generates the following subqueries.

```
Q1: join-query = <
        Projection-expression = {Avatar},
        Sources = {
                SC= <C, {...}, global>
                SP= <Positions P, {<Area, =, 7>}, {local }>,
                SS= <Servers S, {}, {global}>}
        Joins = {
                JSC = <S, C, jcondBC>
                JPS=<SP.Area, =SS.Area>
        }
>

Q2: join-query = <
        Projection-expression = {Avatar},
        Sources = {
                SD= <D, {...}, local>
                SE= <E, {...}, local}>
        }
        Joins = {
                JDE = <D, E, jcondDE>
        }
>
```

Figure 5.8 depicts the plan generated at this step. This is partial plan not ready yet to be executed.

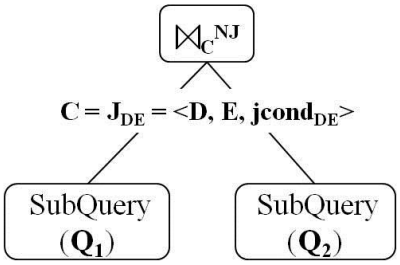

Figure 5.8 Query plan after query decomposition by join selection

5.3 PLAN SETTING

Plan setting is the third and last step of our CBR-based query plan generation process. It receives as input a plan template, and produces the (close to optimal) query execution plan. The plan setting process adjusts a plan template to the specifications of a given query. It succeeds the query-case retrieving (reusing the plan template from such case) or the top-down pseudo-random generation of a new plan template. This process fulfills the empty parameters of the operators in the plan template. Its output is an executable query plan. The plan setting does not modify the shape of the plan template (order of operators).

Setting the print operator (root of the plan tree) of the plan template is a straightforward step; such operator receives as parameters the projection expression of the query to be evaluated. A projection expression may include attributes, aggregation/arithmetic functions, and values (see Chapter 4). Then, there is a recursive process for propagating projections among the remainder of the plan operators. The objective is to minimize the size (i.e. number of attributes) of intermediate results.

The unit of data manipulation is an item, as explained in Chapter 3. Thus, any inner or leaf operator must output items comprising values of a subset of attributes included in the query projection expression. Such items also must include values of attributes (e.g. attributes in join conditions) required for the execution of operators (e.g. joins) at higher levels. Operators are set to the appropriate projection expressions. On the other hand, data-access and communication operators also must be set according to the selection conditions of the new query. These conditions are extracted from the original query. Figure 5.9 shows an example of a plan operator setting.

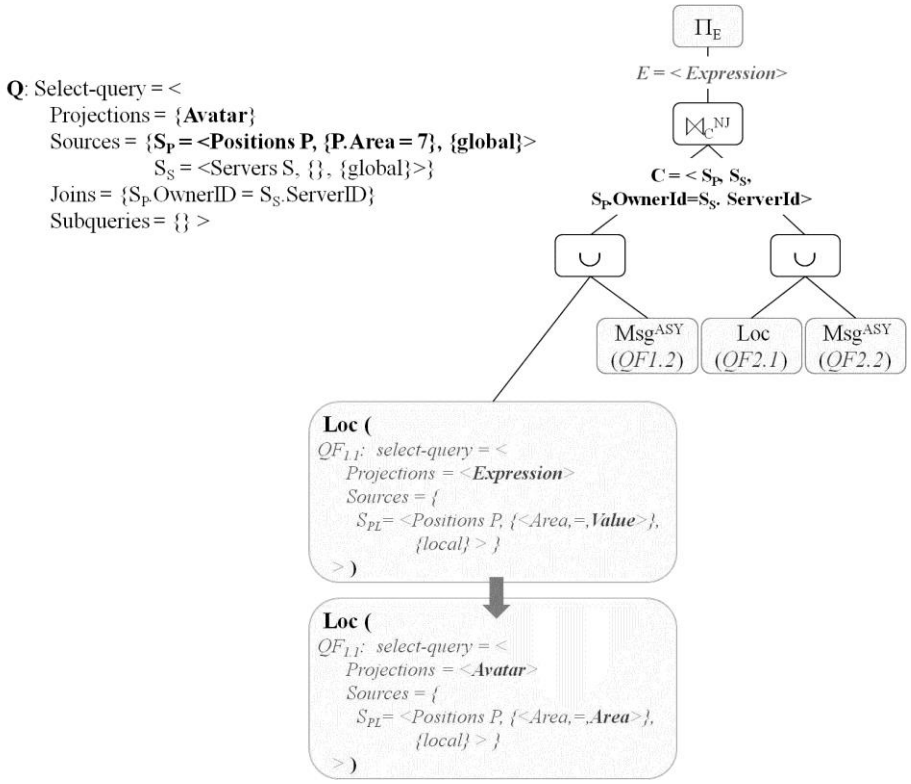

Figure 5.9 Setting a plan template

Thus, the set of data-access or message operators is achieved in three steps: (i) identify the source(s) that the operator concerns, (ii) extract from the given query the projection expressions and selection conditions of such source(s), and the projection expressions required by the parent operators, (iii) finally, assign such values to the query template. Let us recall that in the plan template the parameters of these operators correspond to query templates, the setting process fulfill the missing parts of these templates.

For example, the local data-access operator that has a query template that comprises an empty projection expression, and the source S_{PL} = <Positions P, {<Area, =, *Value*>}, {local}>. To fulfill this template, we extract from the query example the selection condition <Area, =, 7> that concern to the global source S_P (are applied for local S_{PL} and distant S_{PD} data). According the schema *POSITIONS (Avatar avatar{key}, Int area, nodeID owner)*, the projection expression <Avatar> in the query also concerns to the source S_{PL}. The general pseudo algorithm of our plan setting process is presented below (Algorithm 5.3).

```
Input(s):
PlanTemplate PF: Template of a plan for a query family
Query query: A given query
Output:
QueryExecutionPlan P: PF set to the specifications of Q
Procedure 1:
Plan_setting(PlanTemplate PF, Query query)
BEGIN
1      Operator print = PF.getTopOperator(); /*set projections for print
2                                            operator*/
3      Vector <ProjectionExpression> exp = query.getProjectionExpression();
4      operator.set(exp);
5      exp = tacke_off_values(exp);
6      propagation(PF.getSubPlan(), query, exp)/*heuristic – propagation of
7                                        projections and selection conditions*/
8      return (QueryEXecutionPlan)PF;
END
```

Algorithm 5.3. Plan setting process

The setting process pursues to straightforward adjustment of the original plan template. It aims to make the most of the acquired query knowledge i.e. the query plan with the lowest execution cost for a given optimization objective. Complex adjustments of the plan template, like adding and/or removing operators, may drastically change (i.e. increase) the consummation of resources, defeating the purpose of the query optimization approach proposed in this thesis.

It is important to evoke that the definition of query similarity plays an important role in a plan setting. Plans used for evaluating past queries are reused and set for evaluating further similar queries, how complex is such plan setting is determined by the query similarity definition. A very flexible definition of similarity yields to complex plan setting, since two queries are similar even if they have quite differences, for example different join operations. In contrast, if the similarity function is more restrictive, a simplistic plan setting is required as in our case.

In some applications it is wise to use complex plan settings methods, while in others, the simple adjustments are the best option. Complex settings can yield to infeasible plans, then requiring recursive methods for verifying the plan feasibility and for re-adjusting the plan when required until achieving a viable plan (it is not always possible). In the worst case, plan setting can be harder than generating the plan from scratch. The benefit of a complex setting process is the diminishing of the complexity of other steps in the case-based reasoning process. For example, the more complex adaptation, the less cases are needed in the casebase; thus the coverage of each case increases. This may accelerates the query-case retrieving process and facilitates the casebase maintenance.

A consequence of our admittedly straightforward plans setting method is the decrease of cases coverage and thus, the number of cases in the casebase augments to. Therefore, it is very important to accelerate the learning process and refine the strategies for maintaining the casebase. Chapter 4 exposes our casebase management process. The pseudo algorithm for the recursive propagation of projection expressions is presented below.

```
Input(s):
PlanTemplate PF: Template of a plan for a query family
Query Q: A given query
ProjectionExpression exp: …

Output:
Plan P: PF set to the specifications of Q

Procedure:
propagation(PlanTemplate PF, Query query, ProjectionExpression exp)
/*propagation of projection expressions through plan operators*/
BEGIN
1 Operator operator = PF.getTopOperator();
2 ProjectionExpression proj;
3 If (operator is biary)
3      PF_L = PF.getLeftSubtree();
4      PF_R = PF.getRightSubtree();
5      If (operator is union)/*set projections for union operators*/
6            proj = match (propagation(PF_L, query, exp)
7                          propagation(PF_R, query, exp)
8            operator.set(proj)
9      If (operator is join) /*set projections for join operators*/
10           exp_L = exp.add(operator.getLeftAttr())
11           exp_R = exp.add(operator.getRightAttr())
12           proj = intersect(exp, merge(propagation(PF_L, query, exp_L),
13                                       propagation(PF_R, query, exp_R)))
14           operator.set(proj)
15 else
16     If(operator is local-data-access or message)
17           Query subquery = operator.getQuery()
18           Source source = subquery.getSource()
19           proj = intersect(exp, source.getAttributes())
20           RestrictionCondition cond= query.getConditions(source)
21           operator.set(proj, conds) /*set projections and conditions for
22                                     data access operators*/
23           return proj
END
```

Algorithm 5.3. Plan setting process

5.4 PLAN SUMMARIZATION AND REGENERATION

The plan summarization and regeneration processes are supplementary steps that we include to the query plan generation process for minimizing the amount of memory used for the storage of query-cases in the casebase. So far in this chapter, we have discussed the generation of several query-cases (and not a single one) from the execution of a query with the objective to learn more and faster. Storing such high amount of cases demands high memory consumption.

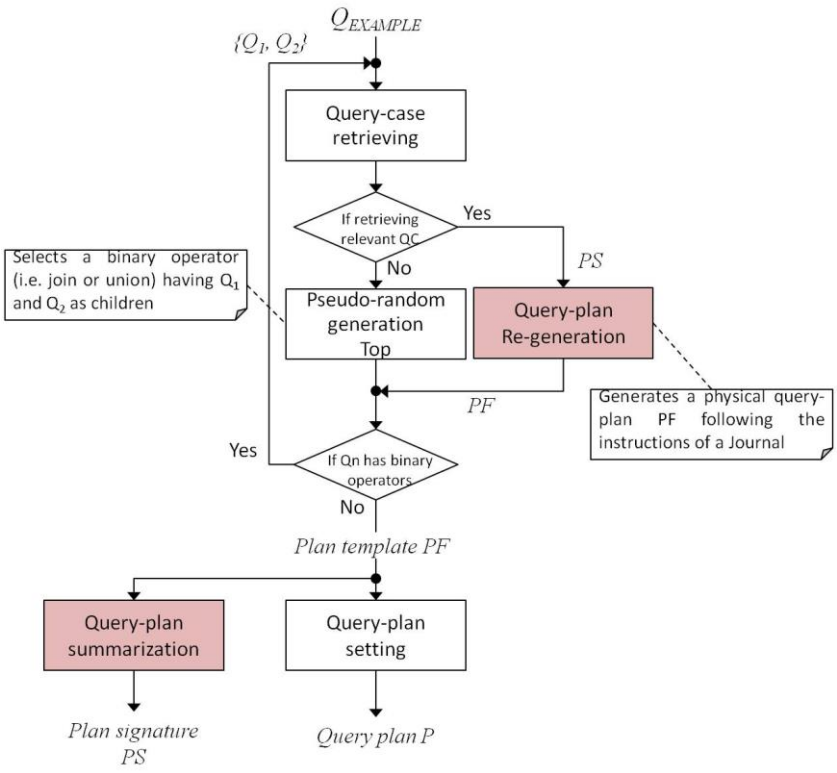

Figure 5.10 Expanded query plan generation process

To alleviate the casebase overflow we proposed techniques for detecting and deleting useless query-cases. However, to minimize as much as possible the consumption of memory, we go one step further by proposing a compact way for representing and storing the plans in the query-cases. Figure 5.10 illustrates the query-plan generation process including these supplementary steps.

Currently, a query-case keeps: (i) the identifier of the query family to which it belongs, (ii) a list of values corresponding to the amount of consumed computational resources, and (iii) a physical query plan with some empty parameters. The memory for storing the two first elements is inexpensive, but storing a complex query plan (e.g. join of several sources) may require significant memory space. Thus, the summarization process generates a compact representation of a given physical plan PF, which we call *plan signature*. The generate query-case keeps such plan signature instead of the whole physical plan. The regeneration process makes the opposite to the summarization, it generates the plan PF from a given *plan signature*.

- *Query-plan summarization*: Given a query Q from the query-family QF, a plan signature PS is generated by registering a summary of the optimizer decisions during the generation of the plan PF.

- *Query-plan regeneration*. Given a query Q, from the query family QF, the plan template PF is generated by exploiting a query case QC. The process receives as input the plan signature PS extracted from QC. The plan template PF is generated by following the instructions in PS.

A plan signature registers the decisions that the optimizer took for generating the plan template. Such decisions comprise the ordering of operators and the execution techniques for the operators' implementation. For regenerating a plan template, the plan signature is interpreted as a set

of instructions to follow to build each operator of the plan and to bind the operators among them (parent-children relation) within the plan tree. The re-generation of the plan template is deterministic, the decisions for the first generation of the plan template are registered in the plan signature; the same decisions are made (followed) when such plan must be reused.

Adding this supplementary steps represent a slight variation over the general query plan generation process presented in Section 5.2. The query plan generation process is carried out just after the query-case retrieving process. The plan signature is extracted from the retrieved case, a plan template is generated by following the instruction of the plan signature; the remainder of the plan generation process remains the same until the generation of the query execution plan. The summarization is a process triggered as background of the query plan generation process.

We propose a summarization process for generating a compact representation of our query plan template that we call plan signature. The benefit is the minimization of the amount of memory requited for its storage. Such process is complemented with a regeneration process to generate a physical plan template from its compressed version. Next sections detail the integration of these two steps to our query plan generation process presented so far. Followed by a description of the plan signature, also we explain our plan summarization and plan regeneration processes.

5.4.1 Plan signature

A plan signature PS is a compact representation of a plan template PF. It registers the optimization decisions for generating PF. Such decisions concern the order of operators, and the algorithm for computing each operator. A plan signature is used to minimize the space of memory used for storing a case and for re-generating learned plans for evaluating further similar queries.

A plan signature comprises a sequence of operators represented as tuples. Such tuples are listed in top-down order according to the levels of the plan template, i.e. the first tuple in the plan signature corresponds to the root operator. The plan signature registers the operators that were added to the query plan because of optimization decisions (some operators are added to the query plan just as a consequence of such decisions). Thus, the memory occupied is minimized by storing the essential knowledge only. For representing our plan signature, we use the context-free grammar that for queries representation in Chapter 4. A plan signature is represented as follows:

<PlanSignature> := <(<operator>)*>
<operator> := (<join > | <union> | <message>)

A plan signature comprises join, union and message operators. It includes such operators since they involve inherent optimization decisions during the plan generation process, while other operators (i.e. local data-access operator and print) do not. For example, the optimization process must decide the order of binary operators (i.e. join and union). It also decides the execution techniques of operators for which a catalog of alternative algorithms exists (e.g. nested-join, semi-join algorithms for join operators, and synchronous our asynchronous send of messages).

In our representation of plan signature, the operator identifier is a string of characters (i.e. J1 for a join operator). The operators' algorithms correspond to identifiers of programs; in our representation we will use string of characters (e.g. nested-join). A join operator comprises an operator identifier, a join condition and the execution technique for its implementation. The join condition comprises two source fields and a comparison operator (as the join condition of a query detailed in

Chapter 4). A union operator comprises an identifier and its execution technique. Finally, a message has an identifier, the protocol for its dissemination, and sometimes a next-hop (i.e. next destination node).

<join>:= (<operator_Id> , <join_condition>, <join_algorithm>)
<union> :=(<operator_Id>,<union_algorithm>)
<message>:= (<operator_Id>, <QF_Id>,<dissemination_protocol>, [<next-hop>])

Some techniques considered for executing the join operator may be nested-join [ElSh11], semi-join [SKBK00], and pipelined hash-join [WiAp93]. The union operator is mostly parallel (i.e. invokes its operands at the same time), but also some times is sequential (i.e. invokes its operands in a certain order). A message can be send through the network by utilizing different data dissemination and query routing techniques [AlHM10][SuEt00], a simple example may be to send messages through the network in synchronous or asynchronous manner; also a message can follow different dissemination routes i.e. next-hop that corresponds to the identifier of the next destination node (intermediate or final). Finally, the query plan generation process ought to consider specialized algorithms i.e. programs for the distributed execution of frequent queries.

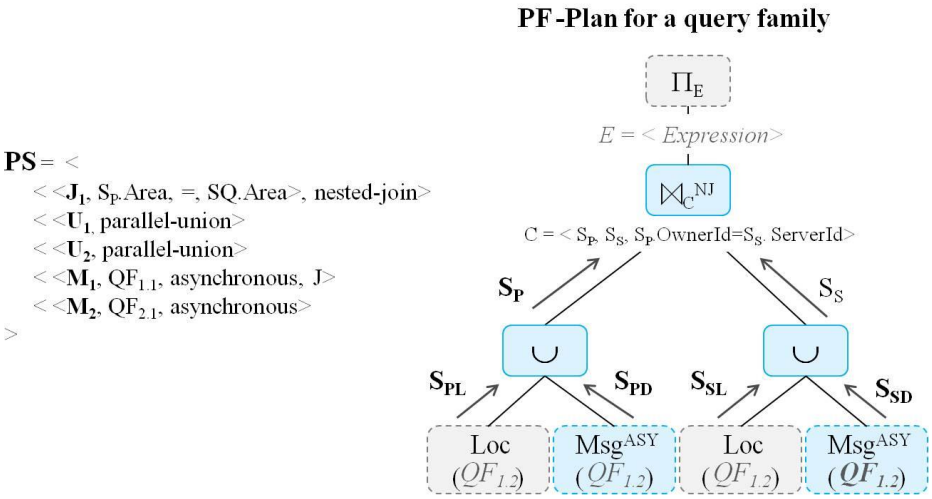

Figure 5.11 Signature of a query plan template

Figure 5.11 presents the plan signature (left side) of a plan template (right side). The join operator is the top of the plan template, within the plan signature such operator is identified as $\mathbf{J_1}$; it joins the global source S_P (i.e. Positions) with the global source S_S (i.e. Servers). The join condition is <S_P.Area, =, SQ.Area>, and it will be computing using the nested-join execution technique. The operands of the join operator are the union operators U_1 and U_2. The union $\mathbf{U_1}$ is applied over the fragments of the source Position that are stored at the local node, and at nodes J and I, and that hold the condition Area=7. The output of U_1 is the global source S_P; such union operator invokes its operands in parallel.

The union $\mathbf{U_2}$ is applied over the fragments of the source Servers that are stored at the local and at other remote nodes (not explicitly specified in the query); its output is the global source S_S. The messages $\mathbf{M_1}$ and $\mathbf{M_2}$ are operands of U_1 and U_2 respectively. They send subqueries through the network to remote nodes for retrieving distant fragments of sources. The query in M_1 fits with the query-family QF_1, such message is send to node J, its next hope, using a synchronous dissemination protocol.

Some operators of the plan template are not specified in the corresponding plan signature, for example, the operators for accessing local data fragments. The plan signature keeps the essential information for the further regeneration of the plan template. The parameters of data access operators (local or distant) are subqueries that can be deduced from the order of join and union operators specified in the signature. Given a query Q, and an order of operators (i.e. joins and unions) for decomposing Q, the resulting subqueries will be systematically the same, thus the corresponding data-access operators.

Moreover, the order of some operators completely depends on heuristics or of the operator functionality itself. For instance, data access operators, and communication operators are the lefts of the tree (i.e. selection first, projection distribution, etc.).Thus, the subqueries within the message (i.e. distant data access) operators can also automatically be deduce from the order of binary operators, however the message operators are included in the plan signature because of it is required to specify the execution technique that was used for its implementation.

The previous operators and algorithms are those that we consider according to our definition of query plan in Chapter 3. However the specification of such operators can be modified for example by including other execution techniques; different operators can be included too.

5.4.2 Summarization

Summarization is the process for generating a plan signature from a plan template. Thus, the plan tree is traversed in breadth-first mode. Each reached operator (i.e. joins, unions and messages) is translated to our tuple representation and listed within the plan signature. Actually, the plan template is summarized, but also its subplans PF_n. The corresponding plan signatures PS_n are kept by the cases for the subqueries Q_n. Please recall that our approach generates a case for the global query Q, but also for its subqueries (from the decomposition of Q during the plan generation process) while must be evaluated in distributed fashion.

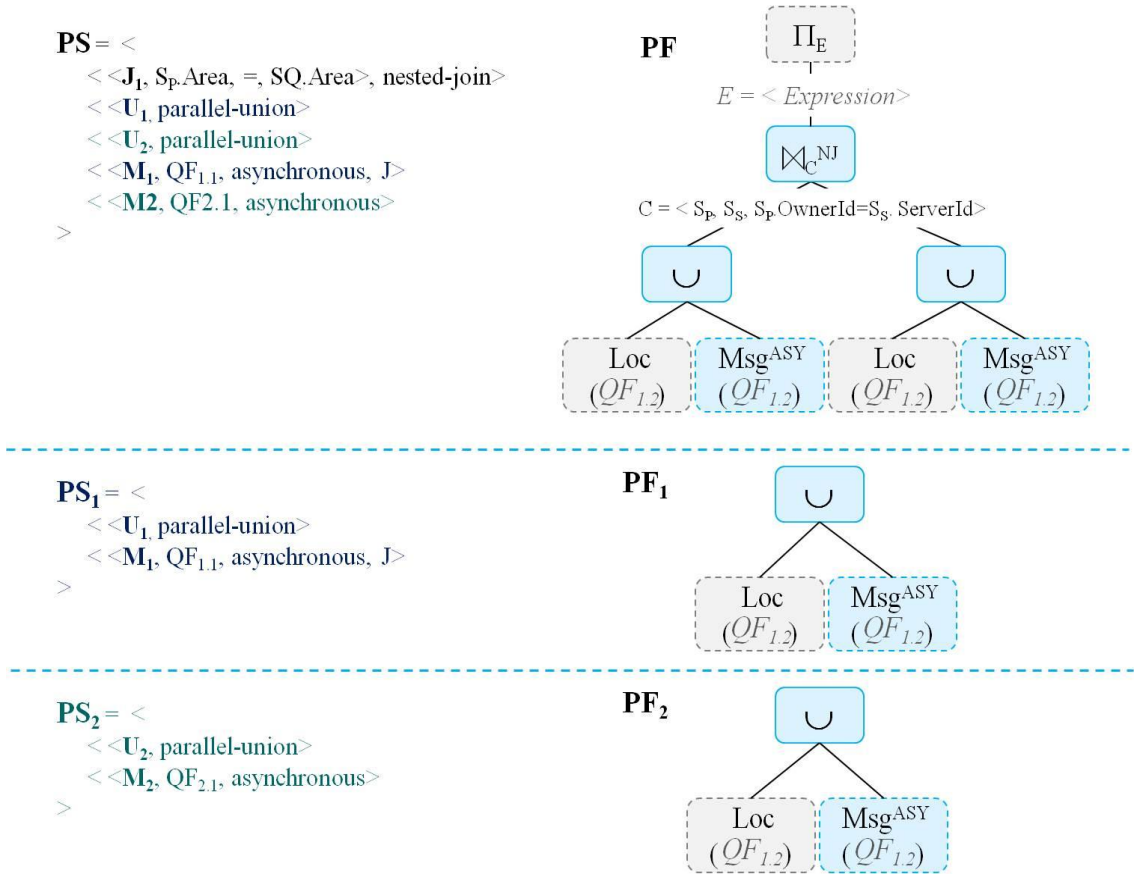

Figure 5.12 Plan signatures PSn resulting from the query optimization process

Figure 5.12 shows an example of a plan template, its subplans PF_n, and the corresponding signatures. The plan template comprises the subplans PF_1 and PF_2, thus the plan signature (i.e. signature of PF) involves the signatures PS_1 (i.e. signature of PF_1) and PS_2 (signature of PF_2).

The algorithm traverse the plan template (i.e. plan tree) in top-down fashion, it translates the operator that it is visiting into a tuple (plan signature representation). Such tuple is annotated with the identifier(s) of plan(s) to which it pertains. For example, the join operator belongs to the plan PF, the union operator belongs to the plan template PF, but also to PF_1, thus it is annotated with the set {PF, PF_1}. The algorithm verifies if the operator that it is visiting is a *top* operator, the root of a subplan PF_n, for example, the union operator a top operator since is the root of PF_1.

In general, the top operators are joins or unions. The rationale is that decomposition of a query Q_n is triggered after selecting one of its binary operators; thus such operator is the root of the plan for evaluating Q_n. The messages are also considered as top operators, since they also correspond to strategies for the distributed evaluation of queries. The output of the algorithm is the complete plan signature PS. The signatures PS_n concerning the sub-plans PF_n are generated by copying the tuples from PS that correspond to the subplan.

```
Inputs:
PlanTemplate PF: Plan for a family of queries
PlanSignature PS: Plan signature
PlanID planIDs: Identifiers of subplans

Outputs:
   PlanSignature PS: Plan signature

Procedure:
summarize(PlanTemplate PF, PlanSignature PS, PlanID planIDs)
BEGIN
1     planIDs = PF.getID()
2     Operator o = PF.NextOp /*First operator of the current PFn*/
3     Tuple t = translate (o)
4     t.set(plansIDs)
5     If(o.isTop())
6          PS.add(t)
7          PF1 = PF.getPFleft()
8          PF2 = PF.getPFright()
9          If(PF1 != null)
10              PS.add(summarize (PF1, PS, planIDs)
11         If(PF2 != null)
12              PS.add(summarize (PF2, PS, planIDs)
13    else
14         return t
15    Return PS
END
```

Algorithm 5.4. Summarization process

PF may comprise new subplans, but also subplans from reusing query-cases; our process summarizes the new plans only. The subplans from reused query-cases are annotated to avoid its summarization (it is not worth since its PS_n is already stored within the reused query-case). Thus, PF_2 is regenerated using PS_2, the top operator of PF_2 is annotated as an already tested plan. PS_n duplicates parts of PS; this represents a waist of memory. The advantage of this approach is that the query-cases are independent from each other. For example, the query-case QC is independent from QC_1 and QC_2, if some of these cases are deleted there is no impact for reusing QC.

Another summarization approach consists in maintaining *links* to the signatures of subplans. Actually, such *links* correspond to the identifiers of query-cases that comprise the subplan signatures used for regenerating parts of a plan PF. A *link* is represented as a string of characters <QueryCase_ID>. The operators of the subplans are replaced by links to corresponding query-cases.

For example, PS maintains references to the query-cases QC_1:<QF_1_id, PS_1,M_1> and to QC_2:<QF_2_id, PS_2,M_2>. The advantage is the minimization of memory usage. In this approach, the query-cases are strongly bonded to each other, since the knowledge required for regenerating a plan PF may be spread in several query-cases. For example, the knowledge for regenerating our PF sample is spread among QC, QC_1 and QC_2 as shown in Figure 5.13.

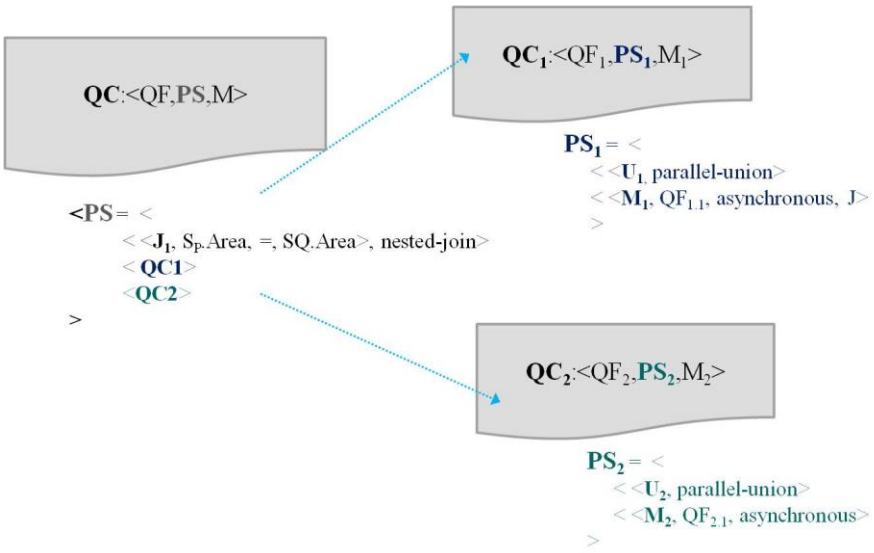

Figure 5.13 Related plan signatures

Thus, the difficulty is to handle the interdependence among query-cases. Maintaining the links among query-cases makes more complex the management of the casebase, particularly the query-case deletion process. A first attempt is delete query-cases in cascade. This is, if a query-case QC is deleted, all query-cases with which it maintains a link must be deleted too. The problem is that a query-case may be linked to more than one query-case, for instance a query-case QC_2, which in our example is bound to QC, may also be bound to another query-case (e.g. QC'). If it is deleted in cascade when QC is deleted, the knowledge for regenerating the plan PF' from QC' will remain incomplete. Moreover, may be QC_2 comprises the signature for regenerating a plan that minimizes some optimization objective; if QC_2 is deleted, valuable knowledge will be lost.

The proposed solution is to establish a bidirectional link, for example between QC and QC_2. The objective is to avoid deleting those query-cases that are still referenced main query-cases. Thus, the plan signature of a query-case comprises a list of the query-cases that utilize it *<used-by>:= (<QueryCase_ID>)**. If some of these query-cases are deleted, then the corresponding query-case identifier is deleted from the plan signature. A query-case can be deleted only if the *used-by* list is empty.

5.4.3 Regeneration

This process is based on the derivational analogy approach, where the goal is to replay previous solutions using a guide for the future reconstruction of the solution for subsequent similar problems [AuMN02]. There is an extensive research focused on different application techniques for solving planning problem; however as far as we know, we are the first to apply the derivational analogy approach in the query optimization domain.

In derivational analogy, cases comprise *derivational traces*, the sequence of decisions made to obtain a plan, instead of the physical plan itself. It is appropriate when the plan setting is one of the main tasks of system and the cost of saving the traces is relatively low [MuCo08]. It can be seen as a set of instructions for *"reproducing"* a query plan template, remaining flexible for replacing certain operator parameters, those bounded by the similarity function.

The input of our regeneration process is a plan signature. The plan template is regenerated following the instructions in the plan signature. As exposed before, the plan signature does not comprise explicit instructions for generating all the operators that should be in the plan signature, but the required for deducing them. We explain the generation process with an example using the plan signature presented below. For clarity reasons, we consider that all the instructions are in a single plan signature (and not split if several PS_n of different query-cases QC_n).

The top operator is the union U_1, the first operator listed in the plan signature. The union is a binary operator, and naturally has two operands. Such operands correspond to the next two operators in the plan signature, thus the joins J_1 and J_2. Because of the join condition it is possible to deduce that the left operand is a data access operator for accessing the local fragment of the source S_P, and the right operand is a union operator for unifying the local and distant fragments of the global source S_S. The operands of J_2 are deduced in the same way, thus the left operand is the message M_1 for retrieving distant fragments of the source S_P, and the left operator is the union U_3.

5.5 CONCLUSIONS

The generation of plans is a recursive process that combines heuristic-based [StMK97][Swam89] and randomized query optimization techniques [StMK97] [IoWo87] [SwGu88]. We go one step further including a learning process based on the CBR principle for reusing learned optimal plans for the evaluation of further similar queries. Thus, given a query, the query plan is generated by retrieving and adapting a query plan from a query-case; we propose a straightforward query plan adaptation algorithm.

When no useful cases are retrieved, we propose a pseudo-random bottom-up technique for the partial generation of a query plan. This technique applies classical query optimization heuristics, and does some random decisions while classical techniques rely on a cost function based on metadata. It decomposes de original query in some subqueries; then, it purses to the exploration of the casebase for retrieving cases useful such subqueries.

It is important to accelerate the query plan generation process (i.e. query-case structure and casebase structure presented in Chapter 4), as well as to minimize the consumption of computational resources. We propose a compact representation of a query plan template with the objective to minimize the memory used for maintaining the casebase. This involve summarization and regeneration processes. The former one summarizes the query plan generating a plan signature, the second one generate a physical plan from a plan signature. Adding these two steps to the query plan generation process involve additional time for carrying out plans generation, but may be a suitable option for devices with limited memory capability.

6. CoBRa for optimizing global queries

This chapter presents the CoBRa query optimizer, which implements and validates our CBR-based query optimization approach. First, Section 6.1 presents an overview and the architecture of our query optimizer. Next, Section 6.2 shows the implementation of the data structures handled by CoBRa. The interaction of the CoBRa modules for generating query-plans is presented in Section 6.3. Afterwards, Section 6.4 describes the UBIQUEST project as the context for validating our approach, and discusses the experimental results. Finally, Section 5.5 concludes the chapter.

6.1 Overview and architecture

CoBRa implements a fully-functional query optimizer bringing together the various aspects of query optimization developed so far in this thesis. Figure 6.1 presents the architecture of the CoBRa optimizer. The optimizer was developed in the context of the UBIQUEST [ABCD12b][ABCD12a].

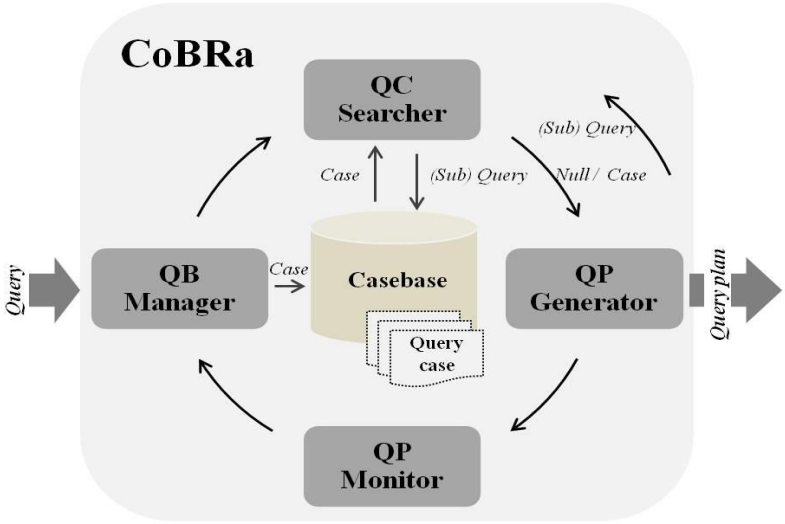

Figure 6.1 Architecture of the CoBRa optimizer

UBIQUEST proposes a high level programming abstraction (i.e. declarative queries) for the rapid prototyping of networking applications. Such applications run in a distributed environment that interconnects –typically through wireless technologies– computational devices (i.e. system node). A UBIQUEST Virtual Machine (VM) is embedded at each node to support query processing. runs an instance of CoBRa for the optimization of global queries. The optimizer interacts with the *Scheduler* and the *ExeEngine* modules within the UBIQUEST VM. The *Scheduler* dispatchs queries to the optimizer with a specific order (see Chapter 3). The *ExeEngine* executes the query execution plans that the optimizer outputs. Section 6.4 presents the UBIQUEST project.

The input of the CoBRa optimizer is a query expressed according to the internal system representation. Such a query results from the preliminary query optimization processes i.e. query parsing, validation and scheduling (Chapter 3 details the query scheduling process for dispatching the queries to be optimized with a specific order). The optimizer outputs the query execution plan. Our optimizer was developed using the Java platform. Object-oriented libraries were used to monitor the consumption of computational resources (e.g. execution time, memory consumption and CPU utilization) during the execution of queries.

The modules of CoBRa correspond to the four steps of the query optimization process presented in Chapter 3. Such modules are: (i) the *QCSearcher* (*Query case retrieving* step), (ii) the *QPGenerator* (*Query plan generation* step), (iii) the *QPMonitor* (*Query plan monitoring* step), and (iv) the *CBManage*r (*Query case management* step –insert, delete, update of cases–).

6.2 DATA STRUCTURES

The CoBRa prototype optimizes global queries expressed in the Data Location Aware Query Language (DLAQL) detailed in Chapter 3. This section presents the implementation of DLAQL queries and of the query execution plans. Also, it exposes the data structures that CoBRa handles for the implementation of cases and the casebase repository.

6.2.1 DLAQL queries

The UML diagram in Figure 6.2 illustrates the implementation of DLAQL queries – Query class–. The following query specializations derive from such class: (i) selection queries –*QSelect* class–, (ii) join queries –*QJoin* class– and (iii) union queries –*QUnion* class –. The *QSelect* comprises a *ProjectionExpression, SelectionConditions,* a list of data *Sources*. It may also include a list of *Subqueries*. A subquery can be in turn any of the three query types mentioned before.

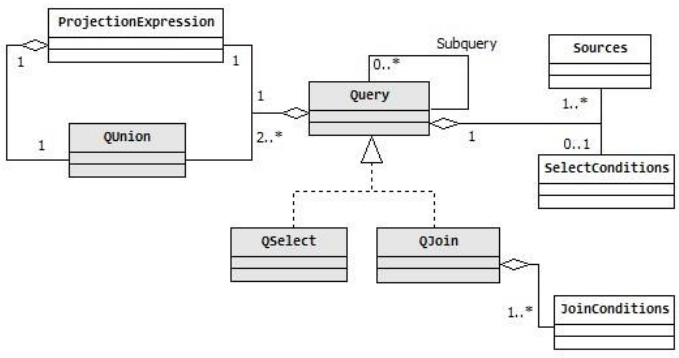

Figure 6.2 UML class diagram of queries

A *QJoin* extends a selection query by including one or more *JoinConditions*. Since the join condition is a binary operator (i.e. includes two sources as operands) the list of sources must have at least two elements. The *QUnion* includes a list of two or more queries (of any of the three types), and its own *ProjectionExpression*.

The classes for data management are the following: *QInsert*, *QDelete*, *QUpdate*; Figure 6.3 show the corresponding UML class diagrams. These data management operations are applied over a single source. Particularly, *QDelete* and *QUpdate* include selection conditions to specify the data to be deleted or updated respectively. *QInsert* and *QDelete* have particular kind of *Expressions*.

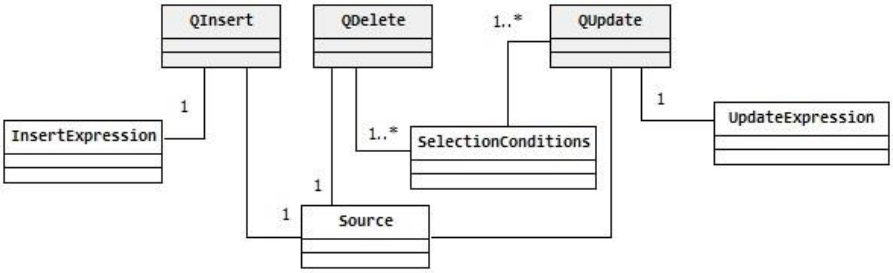

Figure 6.3 UML class diagram of updates

We can distinguish that the main components of queries and updates correspond to: (i) *Expressions*, (ii) *Sources* and (iii) some *Conditions*. There are different expressions for queries and updates as shows the UML class diagram in Figure 6.4.

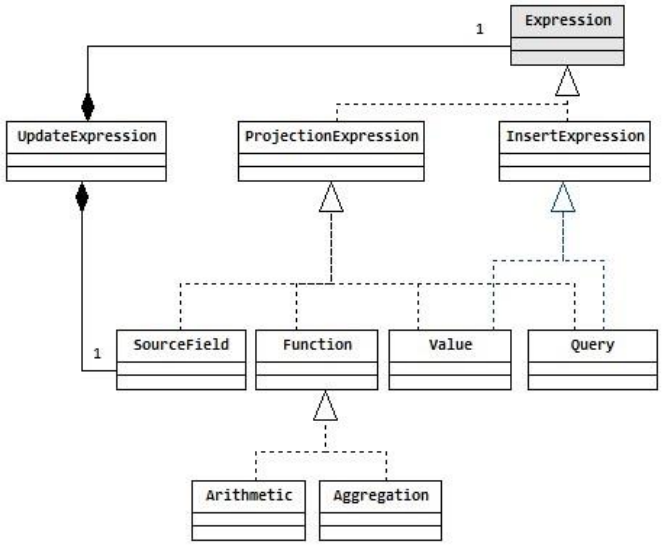

Figure 6.4 UML class diagram of expressions

A *ProjectionExpression* is specific for queries; it may correspond to a value, to a source field, to some function (i.e. arithmetic or aggregation), or to a query. The *UpdateExpression* is composed of a *SourceField* that corresponds to the name of the column to modify on the data source and an *Expression* that corresponds to the new value. The *InsertExpression* it can be a list of values or a *Query* that returns the list of values to insert.

A *Source* of data corresponds to an itemset *ItemsetSource* or to a subquery result *SubuerySource* (the Itemset data model is presented in Chapter 3). It comprises the subset of selection conditions in the query that it concerns, and a *Scope* that indicates the fragments of the source that should be queried/updated. The *Scope* is specified through the SCOPE IS or STORE ON clauses of the DLAQL query language. It also comprises a vector of *SourceFields*. A *SourceField* indicates the properties (i.e. name and data type) of source's columns. Figure 6.5 depicts the UML class diagram of source components.

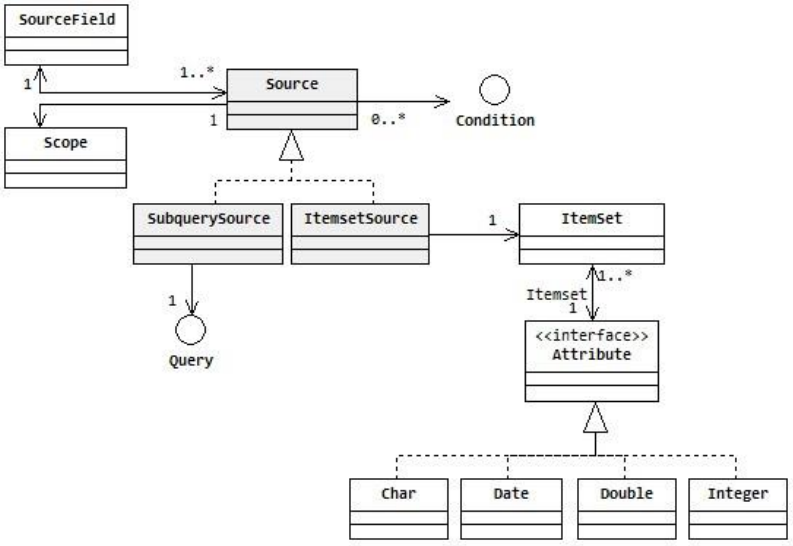

Figure 6.5 UML class diagram of a source

Finally, *Conditions* are classified in *Boolean* conditions (i.e. AND, OR) and *Comparison* conditions. A comparison condition can be a *SelectCondition* or a *JoinCondition*. Figure 6.6 shows the condition representation with an UML diagram.

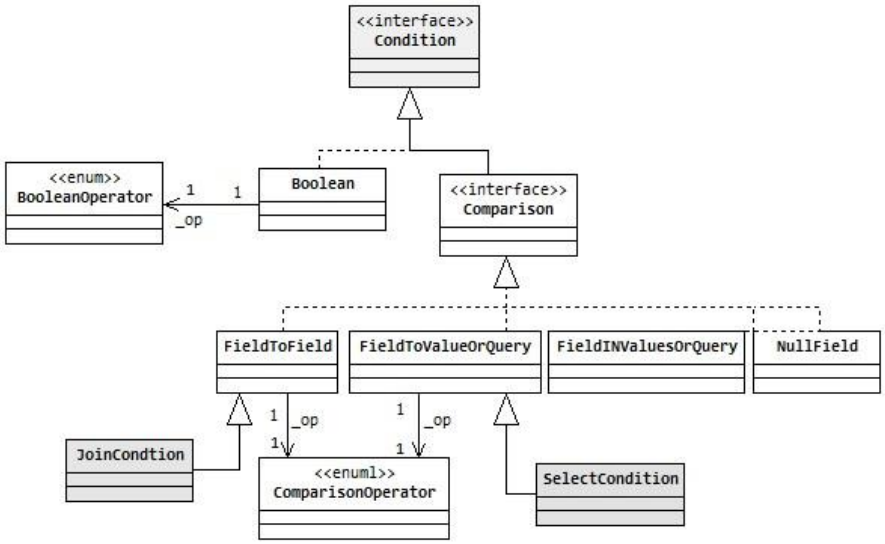

Figure 6.6 UML class diagram of a condition

6.2.2 Query plan

A physical query plan is implemented according to the well-known Iterator model [Grae93]. Therefore, it is a tree structure, where the nodes correspond to *Iterators*. An *Iterator* is a code object that receives as input a vector of children (i.e. other Iterators); such parent-children relation denotes the edges of the tree, thus the flow of data during the execution of the query plan.

The function of an *Iterator* is to iterate over a granule of data (i.e. Item) for applying an operation and producing a partial query result. We define an Item as granule of data; Chapter 3 introduces the Item definition. An *Iterator* separates the implementation of an operator in three processes: (i) the operator is prepared for producing Items (e.g. establishes a connection with the source of data), (ii) the operator demands a new Item, it processes such Item and produces a result Item (e.g. if the Item does not hold a condition the operator produces a *null* result). Finally, (iii) the operator performs a final house-keeping (e.g. close the connection with the source of data). Such processes are called *open*, *next* and *close* and are the main operations of the *Iterator* interface.

For executing a query plan the *Iterators* schedule each other within a single process. While an Iterator needs a new Item, it propagates a *next-call* through its child(ren) until reach the leafs of the tree. The leaf Iterators perform the first operations over the Items (i.e. access to data sources), and send the results to their parents. An *Iterator* is wrapped by a more complex object called *QPNode*. A *QPNode* comprises: (i) an Iterator, (ii) a buffer of Items, and (iii) a collection of measures. Figure 6.7 depicts the diagram of a query plan node.

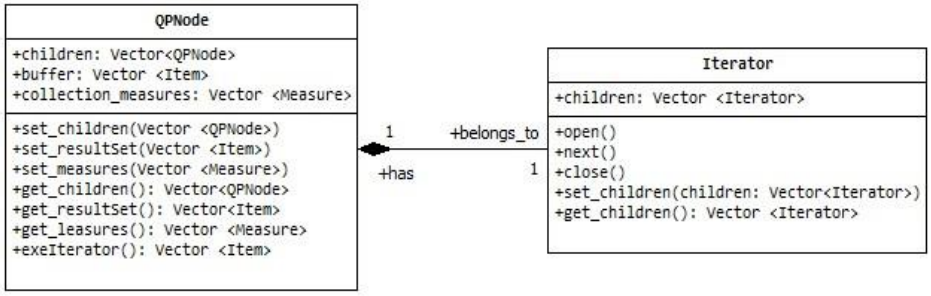

Figure 6.7 UML class diagram of a query plan node

The Buffer of data serves for temporally storing partial results of a query. The Collection of measures stores the measures of resources consumed by the execution of the corresponding Iterator. Such measures serve for calculating the global measures of computational resources consumed during the execution of a query plan P, but also to isolate the measures of the subplans of P.

Consider that an Iterator may correspond to a complete or a partial query plan for executing a query Q. The computational resources consumed by an Iterator are the result of those consumed by its children Iterators. The Collections of measures provide the measures of resources consumed by partial plans executed locally or distantly. From such partial measures is possible to calculate the global measure. Also, the partial plans may correspond to subqueries of Q resulting from the query plan generation process. Therefore, the measures of the execution of such partial plans correspond to the measures that must be kept within the query cases of the subqueries.

The diagram in Figure 6.8 depicts the operators considered in the implementation of our query plan. Such operators correspond to implementation Interfaces, their codification correspond to specific

algorithms for their execution. For example, for a join operator may be implemented with a nested-join, merge-join [ElSh11] or hash-join [UrFr00] [IFFL99].algorithms.

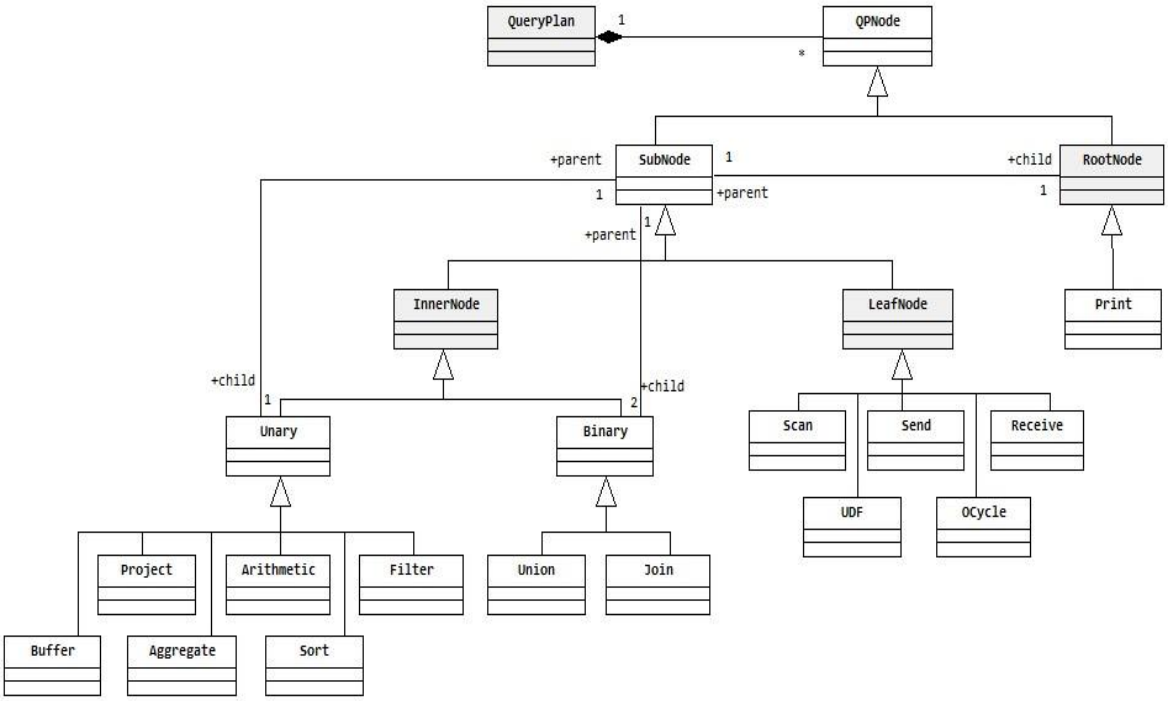

Figure 6.8 UML class diagram of a query plan

The bounds between the Iterators indicate the relation among the operators of the query-plan. They connect the operators maintaining a top-down parent-child(ren) relation and denote the flow of data among operators (e.g. item production-consumption). The operators accept one (unary) or more inputs (binary), and produce a single output.

The position of the operators in the plan tree is specified by hierarchy of classes: (i) leaf operators descend from the class *LeafNode*: *DMS*, *Message*, *Program* and *Subquery* algorithms, (ii) intermediate operators descend from the class *InnerNode*: *Union, Join, Select* and *Aggregate* algorithms. Finally, (iii) the root operators descend from the class *rootNode*: *Print*, *Delete* and *Insert* algorithms. The inner nodes are classified according to the number of operator inputs (i.e. unary and binary). The functionalities of the implemented operators are explained in Table 6.1 below:

Operator	Description
Leaf Iterators	
DMS Iterator	It translates a query specified in the system representation to a language (i.e. SQL) used by some Local Data Management System (i.e. Oracle). *Open* - establishes a connection with the LDMS, poses the query and obtains a partial local data result-set. *Next* - iterates over the result-set, at each iteration it generates an item and sends it to the Iterator that demands its execution. *Close* - interrupt the connection with the LDMS.
Message Iterator	It sends a query through the network for the interaction between distributed nodes. *Open* - Creates a message, sends the message through the network according to a selected routing protocol and obtains distant data results-set. *Next* - iterates over the result-set (this time, is a set of items since the translation was made for the distant

Program Iterator	nodes). It invokes a rule-based program that solves a query. Rule-based programs are defined explicitly for solving queries that are frequently posed. They are permanently stored in the casebase as special query-cases. *Open* - demands the execution of a rule-based program (such task can be delegated to a specialized engine). It obtains a partial local and/or distant result-set. The program is executed locally; however its execution can trigger queries that are sent through the network and solved by distant nodes. *Next* - iterates over the result-set.
Inner nodes	
Aggregation Iterator	It executes aggregation operations (MIN, MAX, AVG, COUNT and SUM) over the item's attributes specified in aggregation projection expressions. *Next* - Iterates over items generated by its *Child Iterator* and executes progressively the aggregation operation. It generates a single item with a unique attribute specified in the aggregation projection expression.
Arithmetic Iterator	It executes arithmetic operations (+, -, /, *) between item attributes. Next - Iterates over items generated by its *Child Iterator* and executes at each of them the specified arithmetic operation. At each iteration it generates an item with the previous attributes, replacing those attributes involved in the arithmetic expression by the operation result.
Join Iterator	It joins items obtained from the request to *leftChild* and *rightChild Iterators*. It joins these items according to a specified join condition. *Open* – Iterates over *leftChild* items. *Next* – Iterates over *rightChild* and checks the join condition comparing all items with the first *leftChild* item. If the condition is satisfied it generates a new item. When it finishes the *rightChild* iteration, requests for the next *leftChild* item and starts again the comparison. It repeats the process until there are no more *leftChild* items.
Union Iterator	It unifies several result-sets. It maintains a list of *Child Iterators* from which it verifies that are defined according to equivalent schemas (type of attributes and order i.e. item from result-set A = <Int: 1, Char: 'a' >, item from result-set B= <Int: 2, Char: 'b' >). *Next* - It starts by the first *Child Iterator* on the list, it iterates over its items, and then it performs the same operation until the last Iterator in the list
Root nodes	
Project Iterator	It prints the item's attributes specified in projection expressions. *Next* - It iterates over items generated by its *Child Iterator* and generates an item that contains only the attributes specified in the projection expressions.
Delete/Insert Iterators	Their functionality is similar to *DMS Iterator*, but changing the type of queries that they support. They translate a query specified in the QOL query representation to a language used for some Local Data Management System. *Open* - They establish a connection to the local database. *Next* - They insert/delete an item. *Close* - They interrupt the database connection.

Table 6.1 Query plan Iterator nodes

6.2.3 Query case

Figure 5 shows a UML class diagram of the implementation of a query case –QueryCase class–. Such class comprises objects that correspond to the query case components (see Chapter 4): (i) a query template, (ii) a plan template, and (iii) a set of global measures.

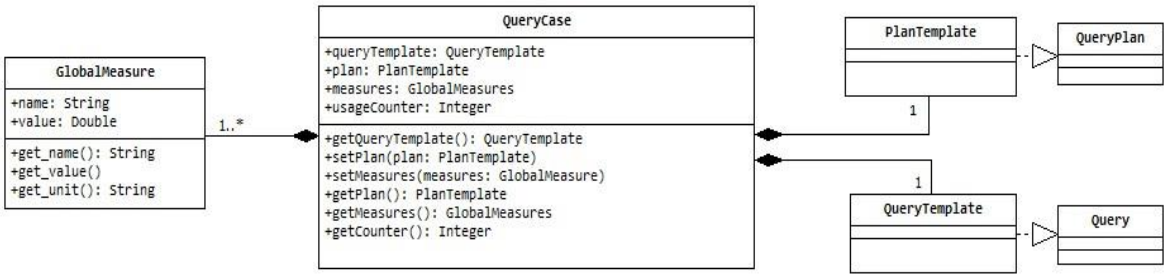

Figure 6.9UML class diagram of a query case

For a brief remainder, in Chapter 4 we present a possible definition of query similarity based on the query features. Also, we explained that we grouped cases comprising similar queries in clusters that we termed query families. A query family QF is represented by a query template –*QueryTemplate* class– that has the same components of a query –*Query* class–; but with the only difference that some of its feature parameters are empty (those that are neglected to consider that two queries are similar e.g. specific projection expression). A query template –*QueryTemplate* class– serves for evaluating a family of similar queries. Thus, it has the same structure than a query plan –*QueryPlan* class–, but with some empty parameters as explained in Chapter 4.

The *GlobalMeasures* class implements the global measures of computational resources consumed during the query plan execution. A global measure comprises its name (i.e. type of measure), the value and the unit of metric. The types of measure are predefined using the *Enum* data structure [Orac11] from the Java API. In the current implementation they include: EXECUTION_TIME, NETWORK_ MSG_COUNT and NETWORK_HOP_COUNT. However, this measures can be easily modified or extent. Each Type such types is associated to a unit of measure denoted with the String: “seconds”, “messages” or “hops”, respectively.

6.2.4 Casebase

The *Casebase* class extends the *HashMap<K, V>* Java class [Orac13]. Such class implements a hash table, which maps keys to values. Any non-`null` object can be used as a key or as a value. It provides methods to insert and delete operators on the table. A *Map* is an object that maps keys to values. This class is based on the implementation of the *Map<K,V>* interface. The *HashMap* class receives as parameters: *K* that denotes the type of keys maintained by this map, and *V* that denotes the type of mapped values. A key can map at most one value.

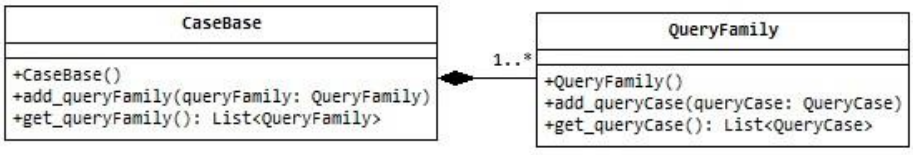

Figure 6.10. UML class diagram of casebase

This implementation provides constant-time performance for the basic operations (`get` and `put`), assuming the hash function disperses the elements properly among the buckets. Iteration over collection views requires time proportional to the "capacity" of the *HashMap* instance (the number of buckets) plus its size (the number of key-value mappings). A bucket corresponds to a query family. In the implementation of our casebase, a key k_i corresponds to the identifier of the query family, a value v_i corresponds to the query-family.

A query family is a repository of cases with similar queries. It is implemented by extending the *TreeSet <E>* Java class. As implied by its name, this class implements the mathematical set abstraction, but providing a total ordering on its elements (i.e. sorted set) by using a Red-Black tree structure. A Red-Black tree is a type of self-balancing binary search tree (BST) [Pfaf04]. Such tree automatically keeps its height (maximal number of levels below the root) small when arbitrary insertions and deletions of *elements*.

The organization of maps in this tree structure is transparent when the objects used as keys implements the Comparable interface responsible of control the order of the tree structure. The sorting methods of this class guaranty that the tree structure does not contain duplicate keys. Moreover, it provides methods for the navigation of such structure. The parameter E corresponds to the type of elements maintained by this set. Such elements correspond to the query cases. The implementation of a red-black tree offers access, insert and delete operations guaranting log(n) time cost. Such algorithms are adaptations of those in Cormen, Leiserson, and Rivest's Introduction to Algorithms [Orac13][Pfaf04].

The self-balancing BSTs have a number of advantages and disadvantages over their main competitors, hash tables. One advantage is that they allow fast enumeration of the items in key order, which hash tables do not provide. One disadvantage is that their lookup algorithms get more complicated when there may be multiple elements with the same key; this is not our case, since each query-family is related to a single key (i.e. query family identifier). Self-balancing BSTs have better worst-case lookup performance than hash tables (i.e. O(log n) compared to O(n)), but have worse average-case performance (i.e. O(log n) compared to O(1)) [Orac13][Pfaf04].

Thus, for our experimentation we use hash tables that allows the easy implementation of clusters of cases (i.e. query families), and a BST for organizing cases within query families, since they do not require to have a specific order. If required, such structure facilitates the organization of its elements according to a specific parameter; we can exploit this property by organizing cases according to a specific measure.

6.3 QUERY OPTIMIZATION PROCESS

This section discus the interaction of the optimizer modules for the generation of query execution plans. We present three different interactions: (i) the learning phase, where the query plan is entirely generated by pseudo-random process since no useful cases for solving the query were retrieved from the casebase, (ii) the exploitation phase when a relevant case was retrieved for solving the posed query, thus the query plan was entirely generated by setting, and (iii) the exploitation phase, where useful cases for solving some subqueries (but not the entire subquery) where retrieved, thus the query plan is generated combining the pseudo-random and the plan setting processes.

6.3.1 Learning phase

Figure 6.11 Interaction of CoBRa modules during the learning phasedepicts the interaction of the CoBRa modules for generating a query-plan during the learning phase. The *Optimizer* receives a query from the *Scheduler* module that, given a query with subqueries it is in charge to dispatch such (sub) queries to be optimized in a specific order (i.e. in bottom-up order according to the imbrication level of queries). Then, the *Optimizer* sends the query to the *QPGenerator* module that generates the query plan by applying our pseudo-random generation technique (see Chapter 5).

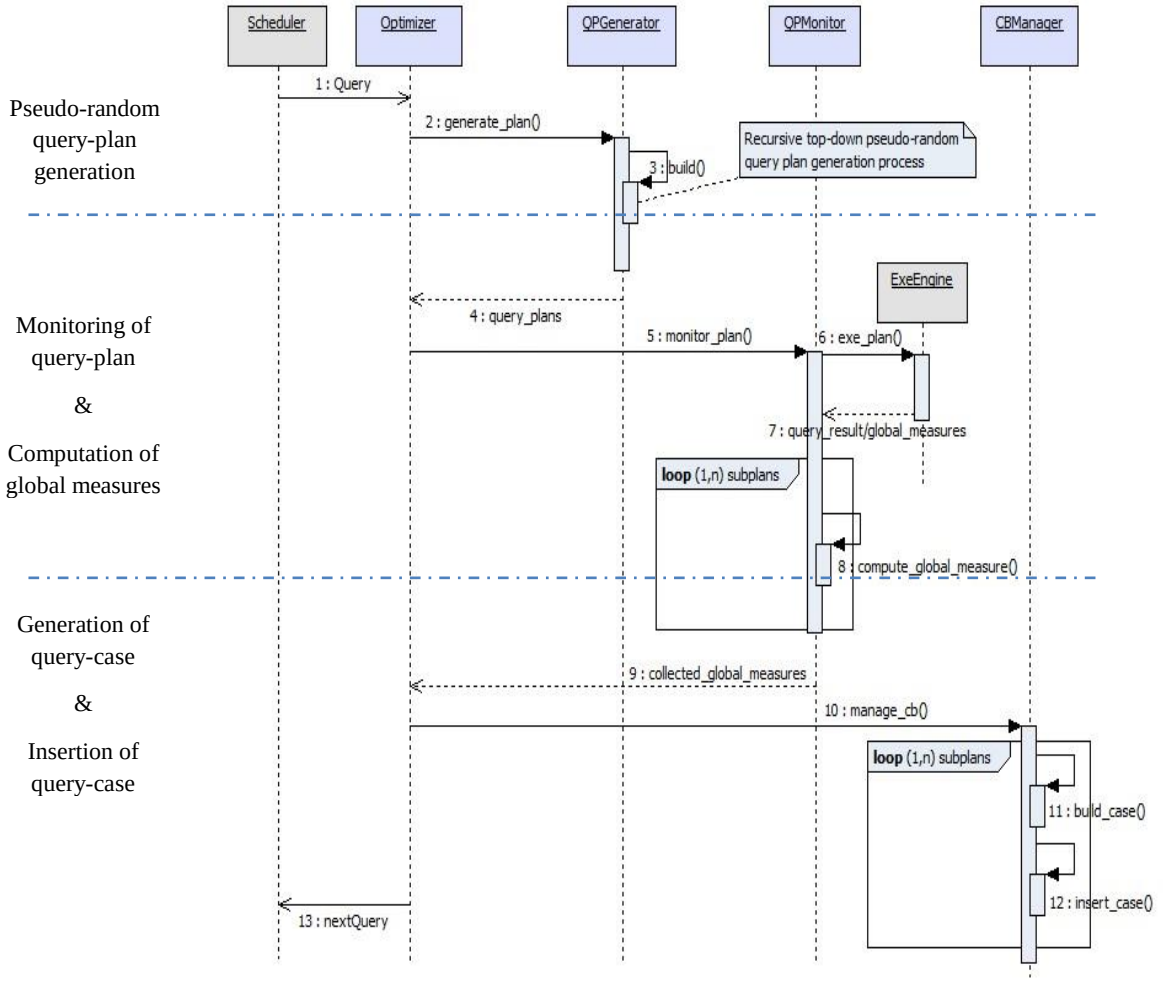

Figure 6.11 Interaction of CoBRa modules during the learning phase

The *QPGenerator* communicates with the *QPMonitor* demanding to monitor the computational resources consumed while executing the query plan. The Optimizer sends to the *ExeEngine* the generated query plan to be executed. When the execution of the query plan has finished, the gathered global measures –*GlobalMeasures* class–, the template of the generated plan, and a description of the executed query are the inputs of the *CBManager* that builds a new *QueryCase* and inserts it into the *Casebase*, as explains in Chapter 4.

Thus, let us suppose that the used summit a query Q that comprises two subqueries Q_1 and Q_2. Let us consider that both queries have the same imbrication level, thus the *Scheduler* dispatch them to the *Optimizer* in indistinct order. Let us say that the optimizer receives Q_2, which during the optimization

process is decomposed on subqueries $Q_{2.1}$ and $Q_{2.2}$ on local and distant data fragments. Our optimization approach states the execution of a query generates may generate several cases, one that correspond to the query and some others that correspond to the subqueries from the query decomposition.

Once the subqueries are optimized and executed, and then the resulting values are replaced in Q to continue its processing. The cases for Q_1 and Q_2 are generated including the pertinent information, and are stored into the casebase. However, it is required to maintain such information while the processing of Q finishes, the global measures consumed by the subqueries must be included in the global measures of resources consumed during the full execution of Q.

shows the data structures handled for scheduling and optimize query (e.g. our query example Q). The *QueriesSchedule* object comprises the query that is posed by the user, followed by the subqueries that it comprises. The *MeasuresCollection* object is a list of global measures corresponding to the computational resources consumed during the evaluation of each (sub) query within the *QueriesSchedule*. The image also illustrates the generated query cases resulting from the whole optimization of Q, which apart from the global measures, also includes the query template QF_n and the corresponding plan template PF_n.

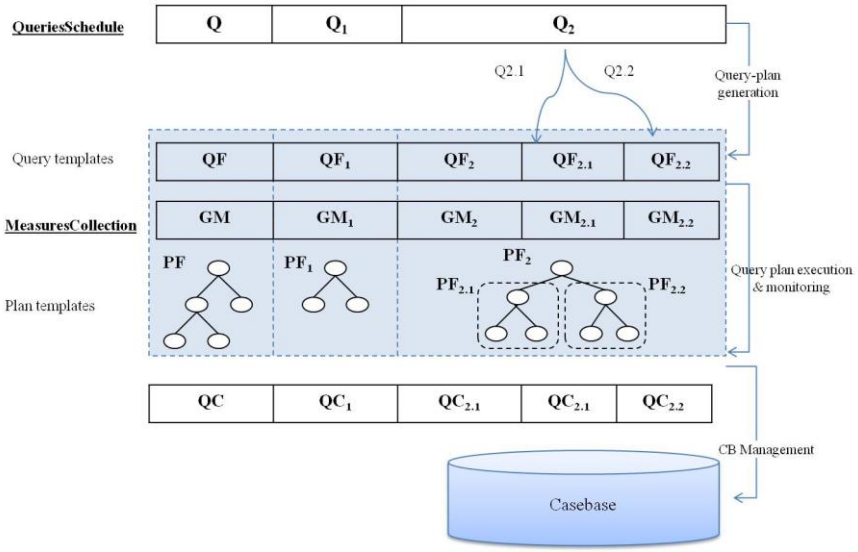

Figure 6.12. Data structures for modules exchange

6.3.2 Exploitation phase

Figure 6.13 depicts the sequence diagram of the plan setting process, showing the interaction among the modules of CoBRa for generating a query plan QP by retrieving and setting a query case. Once again the *Optimizer* receives a (sub) query from the *Scheduler* module. This time, it communicates with the *QCSearcher* for exploring the casebase looking for a case useful for soling the given query. Such module implements a method to determine the query family comprising cases for queries similar to the given query; also it implements a *CostFunction* interface for the specification of different cost functions.

The *QCSearcher* returns to the *Optimizer* the selected query case. Then, the *Optimizer* extracts the plan template and sends it to the *QPGenerator*. It applies the plan setting technique for generating the query plan. We proposed two different ways for storing query cases; the first one stores the

complete plan template PF within a single case, the second one stores part of the plan template within a case and maintains references to other cases that store subplans of PF (Chapter 5 details the both storage approaches).

When the entire plan is within a single case, the *QPGenerator* sets the retrieved plan template, and sends the generated query execution plan to the *ExeEngine*. Otherwise, the *QPGenerator* generates part of the query execution plan, and sends to the *QCSearcher* the references to the required cases. The interaction between these modules continues until the plan for executing the query is accomplished. The Optimizer sends the plan to the execution module and triggers the plan monitoring. Finally, the *CBManager* update the exploited query case(s) aggregating the new gathered global measures.

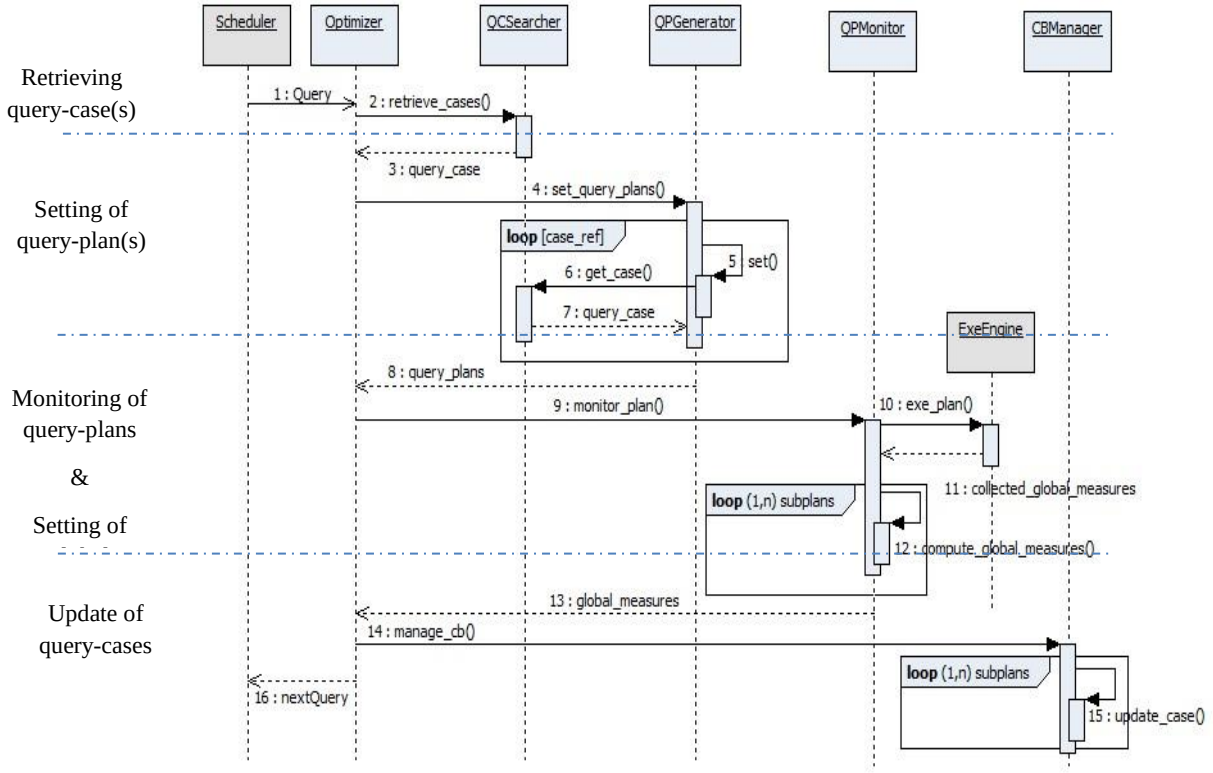

Figure 6.13. Interaction of CoBRa modules during the exploitation phase -plan setting-

6.3.3 Hybrid phase

Figure 6.14 depicts a sequence diagram that shows the interaction of the CoBRa modules optimizing a query by combining the pseudo-random plan generation and the plan setting processes. This situation takes place when the casebase does not comprise a query-plan for solving a query, but for solving some of the subqueries generated from decomposing the given query during the query plan generation process.

The *Optimizer* triggers the case retrieving process by communicating with the *QCSearcher*. Let us suppose that there is not a useful case within the casebase for solving the query. The *QPGenerator* selects a binary operator of the query randomly and decomposes the query in two subqueries (pseudo-random process detailed in Chapter 5). Then, it sends such subqueries to the

QCSearcher. This process is repeated until a useful case for some subquery is retrieved, or until it is not possible to decompose the query (i.e. the query does not comprise more binary operators).

If the *QCSearcher* retrieves a case for some subquery, the *QPGenerator* sets the plan template from the retrieved case according to the specifications of the subquery. When the query plan is completed it is executed and monitored. This process produces some new cases to be inserted to the casebase; the cases that were exploited (by reusing their plans) must be updated.

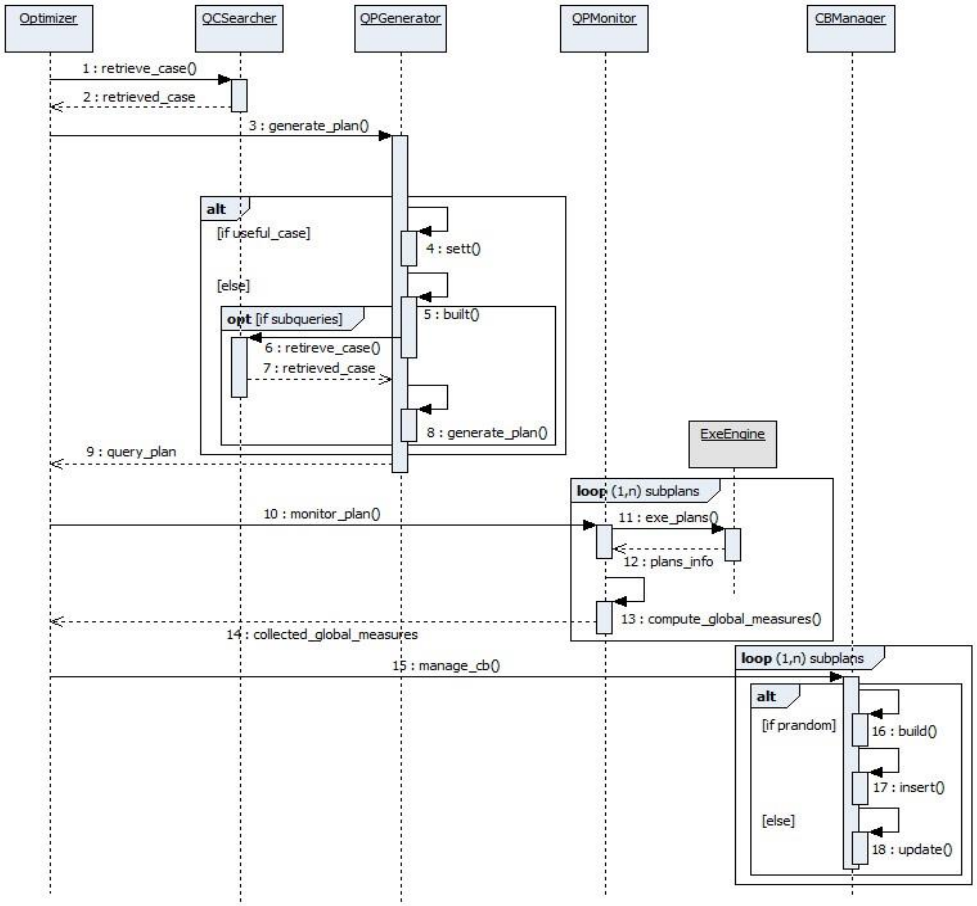

Figure 6.14 Interaction of CoBRa modules combining learning and exploitation

6.4 OPTIMIZING GLOBAL QUERIES IN THE UBIQUEST SYSTEM

6.4.1 UBIQUEST overview

The UBIQUEST project proposes a high level programming abstraction for the rapid prototyping of networking applications. It merges the strengths of two areas: (i) databases, and (ii) declarative networking by abstracting the network as a large distributed database that stores information about the network characteristics and its configuration.

Networking applications produces/consumed data on demand or continuously. They run over heterogeneous devices interconnected between them – typically through wireless network-. These devices are autonomous, either, static or mobile, and present constraints such as energy or communication capabilities. A UBIQUEST node is a device that embeds a Virtual Machine (VM) in charge of data management, processing queries (data selection and updates) and messages propagation for the exchange of queries and data. Also, it comprises a device wrapper for the device/VM interaction.

UBIQUEST nodes interact through declarative global queries i.e. DLAQL queries that must be optimized, but also through declarative networking programs i.e. rule-based programs. Such programs correspond for example to the implementation of algorithms for the total (i.e. specific program mapped to a query) or the partial (i.e. distributed join and message distribution protocols) distributed execution of queries.

6.4.2 UBIQUEST VM

Figure 6.15 shows a simplified architecture of the UBIQUEST Virtual Machine (VM) in a node within a UBIQUEST system. A detailed explanation of such system is not the major goal of this chapter, but positioning CoBRa within this architecture and explains its functionality and how it interacts within the VM modules during a global query processing. A detailed explanation of the UBIQUEST system can be found in.

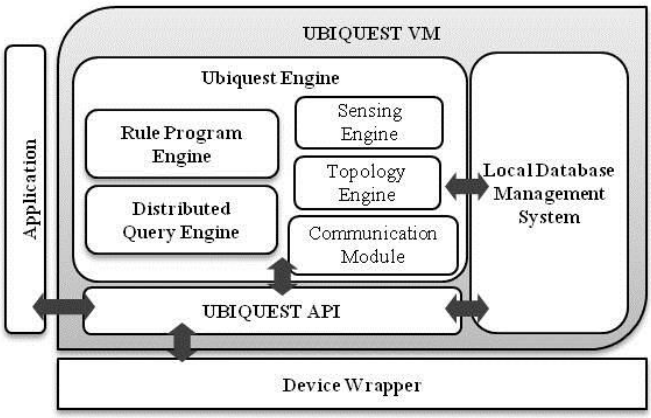

Figure 6.15. UBIQUEST VM architecture

The UBIQUEST VM is composed of the following modules:

- Local Data Management System (DMS). It stores and manages application data (e.g. sensed data, user data) and network data (e.g. table of routes, table of neighbor nodes).
- UBIQUEST Engine. It is composed of specialized sub-engines for: evaluating global queries (i.e. Distributed Query Engine) optimized by CoBRa, executing rule-based programs (i.e. Rule Program Engine), maintaining sensed data (i.e. Sensing Engine), and maintaining the list of neighbor nodes (i.e. Topology engine).
- UBIQUEST API. It manages the interactions of the internal modules (i.e. sub-engines) of the UBIQUEST Engine. Also, it handles the interactions of the UBIQUEST Engine with the rest of the world (i.e. local applications, device wrapper and sensors, and other Ubiquest VMs).

The UBIQUEST VM comprises different engines as explained before. The Distributed Query Engine (DQE) is the module of the UBIQUEST VM responsible of executing global (DLAQL)

queries. The CoBRa optimizer is embedded into the DQE for optimizing the global execution of such queries. CoBRa generates query-plans that may include specialized programs as operators. For executing such operators the DQE may interact with other engines of the VM, specifically the Rule Program Engine (RPE).

We focus specifically on the functionality of the DQE that is composed of: (i) a *Query Scheduler*, (ii) the CoBRa *Query Optimizer* and (iii) an *Execution Engine*. **Error! Reference source not found.Error! Reference source not found.**shows the architecture of the Distributed Query Engine. The functionality of the modules that compose the Distributed Query Engine is presented below:

- The *Query Scheduler* rewrites a global query into a set of sub-queries and schedules their evaluation (e.g. a global *UPDATE* query is decomposed into a sequence of *SELECT*, *DELETE* and *INSERT* sub-queries to read the old value, delete it and insert the new value).

- The *CoBRa Query Optimizer* generates query-plans for executing such queries efficiently (according to a given cost function).

- The *Execution Engine* is responsible of executing the query-plans that CoBRa generates. It delegates the execution of local data access operators to the Local DMS, the execution of rule-based programs to the Programs engine, and the execution of distant data access operators (send of messages) to the Communication engine that manages the *message propagation programs*, and to the UBIQUEST API that deals with the send/receive of messages with others UBIQUEST nodes. The execution of such operators produces partial results that the Execution Engine combines by executing the inner query-plan operators (i.e. joins and unions).

6.4.3 Testbed platform

For demonstrating the CoBRa behavior we used a platform for simulating networks of nodes over which we executed our queries. A screenshot of the simulation platform is depicted in Figure 14. The platform allows to create a network of nodes (add, remove or move nodes) running networking applications developed basing on the UBIQUEST paradigm.

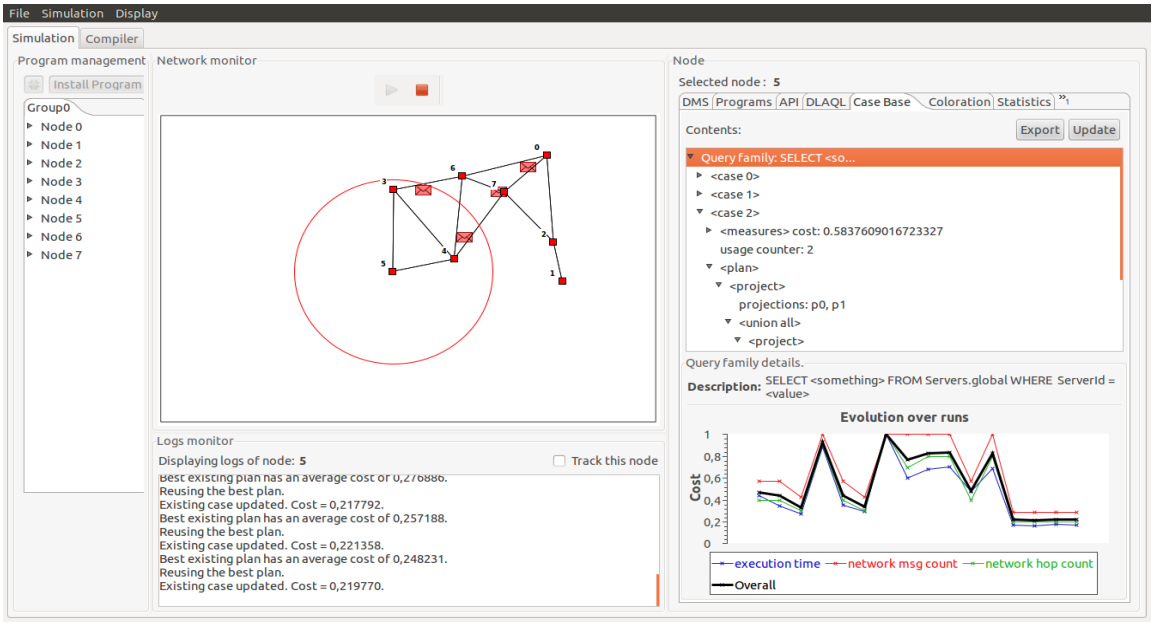

Figure 6.16. Simulation platform

Our simulation platform facilitates the development and monitoring of UBIQUEST applications. A network node corresponds to an UBIQUEST node. Such nodes are instrumented for the monitoring of computational resources consumed during the execution of queries. A DLAQL query can be posed at any node and the system globally optimizes it and executes it. Naturally, an instance of CoBRa runs within each node for optimizing such queries. Partial results are sent as messages to the origin node. Such messages also include the measures of resources consumed by the remote nodes allowing to have global measures of resources consumed during the entire execution of Q.

This platform offers tools for editing and compiling rule-based programs, allows the simulation of UBIQUEST nodes. The main components of the platform are the follow:

- The *Network Parameters Window*, which allows to build and simulate a network with various UBIQUEST nodes
- The *Network Graphical Window*, which allows to visualize and interact with the network at run time
- The *Log Window*, which allows to display the log trace of a given node
- The *Node Settings Tabs*, which allows to interact with a given node and monitor its activity

The *Network Parameters Window* allows creating groups of nodes, displaying the status of the nodes in each group and installing rule programs on them. They can have different colours, radio range, and characteristics, such as mobile or fixed. The system creates the groups and displays the nodes on the left part of the screen. Each node is listed and for each node one can see its identifier, address, position and radio range.

The *Network Graphical Window* offers the view of the groups of nodes, represented by different shapes and colours, the connections between nodes (if two nodes are close enough w.r.t. their radio range), the messages exchanged (network protocol messages are red, DLAQL messages are blue). A node has a unique identifier. The *Network Graphical Window* also allows to interact with the network, and to modify its configuration before starting or during the simulation, by moving nodes, changing their radio range, or deleting edges or nodes for instance.

The *Log Window* displays the log trace of a given node. It helps understanding the system behavior. The *Node Settings Tabs* is displayed on the right part of the screen and exhibits information about the node selected by the user. It contains the following tabs: *DMS*, *API*, *Programs*, *DLAQL*, *Casebase*, *Coloration*, *Statistics* and *Messages*. We are particularly interested on the terminal to summit DLAQL queries and on the tab to display the content within the casebase.

- The *DLAQL* tab allows to type and to submit DLAQL queries. It also displays the concerning query results.
- The *Case Base* tab displays the content of the local *casebase* and information about the query optimization by learning process. It lists the query families processed at the selected node including details about the performance of each *query case* (i.e. query plan). This data can be exported as a spreadsheet file for further analysis. The *query plan* is also represented graphically, as a tree of operators. When a *query family* is selected, an SQL-like description of the *family* is provided. This tab also displays a chart showing the evolution of the query execution cost in terms of different metrics and customizable cost functions. The cost values are normalized to the unit (values from 0 to 1) to use them as parameters of cost functions and obtain the *overall* cost.

6.4.4 Experimental results

6.5 CONCLUSIONS

7. CONCLUSIONS AND PERSPECTIVES

The characteristics of nowadays distributed data systems have change; they include powerful and sophisticated information technologies and hold exorbitant amounts of data. Most of applications running on such environments rely on the efficient querying and processing of such data. In this thesis we focus on the optimization of queries in distributed data systems where, due to their characteristics (e.g. highly distribution of data and autonomy of system resources), there is no guaranty to count on with the information on data used by classical query optimization techniques.

The study in this query optimization domain presented in Chapter 2 highlight the necessity to revisit existent query optimization proposals. Section 7.1 presents the main results and contributions of this thesis. Section 7.2 exposes the perspectives of our feature work.

7.1 MAIN RESULTS AND CONTRIBUTIONS

In this thesis we have presented our approach for optimizing global queries with incomplete complete information on data. This entails declarative querying over highly distributed data systems, taking advantage of query feedback (i.e. measures of consumed resources) to learn the cost of query execution plans. Concretely, in our solution we adapt the CBR principle to the query optimization process.

We study the main foundations of CBR: (i) case content and representation, (ii) problem similarity, (iii) casebase organization, and (iv) reasoning mechanisms. From each of these aspects we select the approaches and techniques appropriate to the needs of our system: (i) minimizing the resources consumed and execution time of the optimization process, and (ii) maximize the accuracy of retrieving useful query cases for the optimization of queries, in consequence that allows efficiently learning optimal plans according to different (customizable) objectives. Although many challenges remain, our solution demonstrated to be efficient and adequate. Next we present our conclusions in regard to the specific contributions of this thesis and discuss about the remaining challenges:

Representation and management of query-cases

One of our main contributions is the adaptation of CBR to query optimization. The addressed aspects for this adaptation are detailed below:

- **Specifying the content of a query case.** The feedback comprises a representation of the query Q, a representation of the query plan P, and global measures M of the computational resources consumed during the evaluation of Q using P. Such measures are exploited to learn the cost of query plans. The cost may correspond to the gathered measures, or to a cost function that uses such measures as parameters. Also, this work suggests a straight forward manner for gathering global measures.

- **Representing a query case.** We propose a structured representation ad-hoc to facilitate the rest of the query optimization process. In particular, we suggest a compact representation of a physical query plan, that we called plan signature, in order to minimize the memory consumption. Also, we include a couple of mechanisms (i.e. summarization and generation) to transform a physical plan template in a plan signature, as well as the inverse process.

- **Defining a query similarity function**. We propose a boolean query similarity function based on query features. A wide variety of query similarity functions have been proposed, most of them relying on very information e.g. characteristics of sources, indexes over attributes, with which it is not easy to count on autonomous and highly distributed data systems. Our similarity function is based on the observation that even queries which differ in projection expressions and selection conditions may still have identical *plan templates*, that is, they share a common database operator tree. By identifying such similarities in the plan space, we can materially improve the utility of plan caching. While current commercial query optimizers do provide facilities for reusing execution plans generated for earlier queries (e.g. "stored outlines" in Oracle 9i), the query matching is extremely restrictive – only if the incoming query has a close textual resemblance with one of the stored queries is the associated plan re-used to execute the new query.

- **Management of query cases**. The case base is organised in groups of cases that comprises similar queries, we called these groups query families. The management operations over the casebase include case retrieving, and storage and deletion of cases.
    - The query *case retrieving* process explores the casebase to select a case useful for evaluating a given query. A useful case comprises a query similar to the posed query and a plan that minimizes a given cost function. Thus, this process involves the evaluation of queries similarity, and the computation of cost functions based on the global measures of query plans.
    - The evaluation of a query triggers *case(s) storage*, which involves either, the generation / insertion of a case(s) into the casebase, or the update of some existing case(s). This depends on the new gathered knowledge: (i) the evaluation of a new query for which no similar queries have been evaluated before, (ii) the execution of a new query plan that has not been already explored within the search space, (iii) or the collection of new global measures from executing a reused plan.

- The *deletion of cases* is triggered when a threshold of knowledge (number of cases within a query family) is reached. Basically, the deletion action consists in delete half cases that have been least used within the family. The objective is to avoid casebase overflow and to continue the exploration of the search while keeping useful cases.

Query plan generation

It is based on the pseudo-random exploration of the search space and the exploitation of cases from query plans evaluated in the past. Our query plan generation algorithm delimits the exploration of the search space by using classical query optimization heuristics. It makes some random decisions when metadata (e.g. data statistics), used by traditional query optimization is not available. It pursues the exploitation of cases to reuse the learned query plans that minimize a given cost function. During the CBR optimization process the quality of these plans is learned by monitoring their execution and gathering their global measures, and the best plans are identified.

Given a query Q, this process tries to retrieve a case of a query similar to Q and that comprises a plan that minimizes a given cost function. If it succeeds, the process reused the plan from the retrieved case by setting it to the query specifications. Otherwise, it generates part of the plan by pseudo-random generating some subqueries. The generation process is recursive; thus it receives such subqueries as input of the recursion. This process is extended by the summarization-regeneration processes that aim to produce a compact representation (i.e. plan signature) of a physical plan to minimize the amount of memory needed for storing cases.

- **Pseudo-random top-down plan generation.** If no useful case is retrieved for the evaluation of a given query, the plan generation process selects a binary operator (i.e. join, union) and its implementation algorithm randomly. The operands of the selected operator are subqueries resulting from the decomposition of the original query (this work specifies rules for query decomposition by join/union selection). The retrieving process is repeated, this time looking for cases useful for evaluating the subqueries.

- **Plan setting.** If a useful query case is retrieved, the query plan generation process extracts the plan from the retrieved case, and sett such plan according to the query specifications. In the setting process the structure of the plan tree remains unchanged; instead, some operator parameters (i.e. projection expressions and selection conditions) are adjusted.

- **Summarization and regeneration.** The summarization process receives as input a physical plan template and outputs a compact representation of the plan. Such a representation is termed (i.e. plan signature; it records the optimization decisions for generating a plan template as instructions. Thus, a case includes a plan signature (shipper in terms of memory consumption) instead of the physical plan template. The regeneration process follows the retrieving process; it is based on the instructions of the plan signature to generate the corresponding physical plan template.

Implementation and validation

We implement the CoBRa optimizer that allows efficient evaluation of global queries without having complete knowledge on data. The CoBRa prototype was developed in the context of the UBIQUEST

project [ABCD12a] that proposes a new high level programming abstraction for the rapid prototyping of networking applications. This optimizer advantageously applies the Case Based Reasoning paradigm to the query optimization process. The CoBRa optimizer acquires performance knowledge (other than classical metadata) while evaluating queries and exploits this knowledge for generating new execution plans for similar queries.

We validate the efficiency of our proposal using a realistic example application, while showing the internal structures of the prototype (case base, case, query execution plans). We carried out our experiments on our testbed platform that supports network-oriented applications. Every node in the system runs an instance of the CoBRa optimizer that is responsible for optimizing the (sub) queries that it receives.

Remaining challenges

We demonstrated that our approach is adequate for data distributed environments with particular characteristics, thus many important challenges remain for the optimization of global queries facing the wide range of difficulties for efficient data querying in nowadays applications.

- **Static of partially dynamic environments.** We present a query optimization approach that is sensible to environmental changes. Thus, our solution will be hardy applicable in highly dynamic and mobile environments where learning about the quality of query plans may easily becomes obsolete. Our approach may be useful only if the frequency of posed queries is considerably higher than the frequency of environmental changes, in such a way that is worth to learn and it is possible to evolve with the environment.

- **Global schema.** We made strong assumptions about data representation, assuming that all nodes running a common application represent data according to a specific global schema. This assumption also restrains the way in which data is partitioned and distributed through the system nodes. Thus, we suppose that data is horizontally partitioned only.

- **Margin of error for computing plans cost.** Our approach states that the cost of plans is learned by monitoring plans execution and gathering global measures of consumed computational resources. It learns the plan that minimize a given cost function (using global measures as parameters), and reuses it for the evaluation of similar queries. The process to select the (close to optimal) plan has two error incomes: (i) similarity function between queries, and (ii) global measures. It is difficult to determine a similarity function that guarantees that reused plans, that have shown to be efficient for a query Q, will be efficient for any query similar to Q. Another difficulty is to determine the kind of measures that reflect the quality of a plan in accurate manner. Different measures may be proposed like, the average, the last measures, histograms, statistics about resources consumed after several runs of a query plan.

- **Simulated applications.** The experiences are carried out in a simulation environment; experiences in a real execution environment should be realized to consolidate the validation of our approach.

7.2 PERSPECTIVES

The work carried out in this thesis constitutes the first step through the definition of a flexible and accurate learning-based query optimization approach. The flexibility refers to the easy customization of cost functions according to different applications needs. The accuracy concerns to minimize the margin of error in the computation of plans cost.

Our query optimization approach is based on try-and-learn experiences to distinguish the query plans that minimize a given (customizable) cost function. To consolidate the validity of our approach should be required to conduct experiments on nowadays data applications and real execution environments. We evoke the perspectives of this work according to these dimensions.

Development of our approach on real environments

The experiences presented in this work were carried out in a simulation environment. The deployment and experimentation of our approach on a real execution environment should be interesting to consolidate our proposal. The data grid, sensor network, the web and scalable storage systems are some possible environments to conduct this work.

For example, several research groups have focused on query processing and optimization in the domain of sensor networks. Typically, the optimization objective is to reduce the energy consumed by the sensors. Some of the main contributions of such works concern to routing and scheduling protocols to minimize energy consumption and communication cost (i.e. amount of data or number of exchanged message between sensors). TinyDB is a seminal project that proposed an acquisitional approach to query processing in sensor networks [MFHH05].The USC/ISI group proposes an energy-efficient aggregation tree using data-centric reinforcement strategies [HeYW00].Cougar is another representative research project that proposed a communication wave-based scheduling approach for communication among sensors [DGRT03].

However, all these works centralize the query optimization process in a single *base station* that maintains a registers of information about all sensors in the system. Could be interesting to distribute the query optimization responsibility; for example using a super-peer architecture where a supper-peer may be a smart-box (or some other device with more computation capabilities than sensors). The sensors act as sources of data. A super peer maintains information of a subset of the sensors in the system. A sensor communicates with its corresponding super peer only. The supper peers are in charge of monitoring the execution of queries to try and learn about the best execution strategies for the different queries that can be submitted in this kind of applications.

Similarity function and cost function

In our work is essential to determine the similarity between queries and to compute plans cost. We presented a prospective boolean query similarity function used to illustrate our optimization approach. However, there is a wide variety of similarity functions that may also be applied (e.g. distance functions) [OsBr97].

The variation of the similarity function may lead to important differences in the experimental results. It has a direct impact in the case retrieving process, where it determines the cases that are relevant for solving a problem. In our approach, the query similarity is critical for selecting cases comprising plans that minimize a given cost function during the evaluation of similar queries.

On the one hand, it determines the coverage of cases. As higher is the coverage of a case as larger is the amount of problems that the solution of case may solve. When cases have high coverage the number of cases within the casebase diminishes. A flexible similarity function increases the coverage of cases; however it does not guaranty that retrieved solutions will evaluate the different problems efficiently. Moreover, the similarity function is the basis for the organization of cases within the casebase in some CBR based systems (as in our approach). The organization of the casebase is critical for our optimization process; a bad organization (e.g. a flat list of huge amount of cases) may lead to a time consuming query optimization process.

Also, in this thesis we used a straightforward cost function (i.e. linear weight function), which uses global measures as parameters and associates a weight according to the importance of the computational resource that the measure parameter represents. The deployment and formalization of a cost model based on global measures remains an open research topic within the query optimization domain.

Dataflow optimization

We think that the utility of our approach may be demonstrated on data intensive applications, such as social networks [Wass94], e-science [VeBW04][WLGP08], *mashups* [SoIS10]. Such applications have a growing demand for massive data sharing, analysis and processing. In data-intensive computing systems applications are expressed in terms of high-level operations on data, and the runtime system transparently controls the scheduling, execution, load balancing, communications, and movement of programs and data across the distributed computing cluster. The programming abstraction and language tools allow the processing to be expressed in terms of complex *dataflows*.

Such dataflows are not restricted to classical arithmetic operators for query processing, they may also comprise arbitrarily operators on data with unknown semantics, algebraic properties and performance characteristics, for example the invocation of services or specialized programs. Traditional query optimization techniques cannot be applied due to the lack of information about the nature of operations on data. Thus, this is an important characteristic of data-intensive computing systems for which we envisage that our approach may be appropriate for this domain.

BIBLIOGRAPHY

[AATC99] Arpaci-Dusseau, Remzi H. ; Anderson, Eric ; Treuhaft, Noah ; Culler, David E. ; Hellerstein, Joseph M. ; Patterson, David ; Yelick, Kathy: Cluster I/O with River: making the fast case common. In: : ACM Press, 1999 — ISBN 1581131232, S. 10–22

[ABCD12a] Ahmad-Kassem, Ahmad ; Bobineau, Christophe ; Collet, Christine ; Dublé, Etienne ; Grumbach, Stéphane ; Ma, Fuda ; Martinez, Lourdes ; Ubéda, Stéphane: UBIQUEST, for rapid prototyping of networking applications. In: : ACM Press, 2012 — ISBN 9781450312349, S. 187–192

[ABCD12b] Ahmad-Kassem, Ahmad ; Bobineau, Christophe ; Collet, Christine ; Dublé, Etienne ; Grumbach, S. ; Ma, F. ; Martinez, Lourdes ; Ubéda, S.: UBIQUEST, A data-centric approach networking applications.pdf. In: *DATA, Rome, Italy* (2012)

[AbCh99a] Aboulnaga, A. ; Chaudhuri, S.: Accurate Query Optimization by Sub-plan Memoization (1999)

[AbCh99b] Aboulnaga, Ashraf ; Chaudhuri, Surajit: Self-tuning histograms: building histograms without looking at data. In: *ACM SIGMOD Record* Bd. 28 (1999), Nr. 2, S. 181–192

[AbCh99c] Aboulnaga, Ashraf ; Chaudhuri, Surajit: Accurate query optimization by sub-plan memoization (1999)

[ACPS96] Adali, S. ; Candan, K. S. ; Papakonstantinou, Y. ; Subrahmanian, V. S.: Query caching and optimization in distributed mediator systems. In: *ACM SIGMOD Record* Bd. 25 (1996), Nr. 2, S. 137–146

[AgPl94] Agnar, Aamodt ; Plaza, Enric: Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. In: : DTIC Document, 1994

[AkPV07] Akbarinia, Reza ; Pacitti, Esther ; Valduriez, Patrick: Query processing in P2P systems (2007)

[AlAO05] Aljanaby, A. ; Abuelrub, E. ; Odeh, M.: A survey of distributed query optimization. In: *The International Arab Journal of Information Technology* Bd. 2 (2005), Nr. 1, S. 48–57

[AlHM10] Al King, Raddad ; Hameurlain, Abdelkader ; Morvan, Franck: Query routing and processing in peer-to-peer data sharing systems. In: *International Journal of Database Management Systems* Bd. 2 (2010), Nr. 2, S. 116–139

[Ali09] Ali, Mohammad Ghulam: A Multidatabase System as 4-Tiered Client-Server Distributed Heterogeneous Database System. In: *arXiv preprint arXiv:0912.0579* (2009)

[AMPT76] Astrahan, M. M. ; Mehl, J. W. ; Putzolu, G. R. ; Traiger, I. L. ; Wade, B. W. ; Watson, V. ; Blasgen, M. W. ; Chamberlin, D. D. ; u. a.: System R: relational approach to database management. In: *ACM Transactions on Database Systems* Bd. 1 (1976), Nr. 2, S. 97–137

[ATFU96] Amsaleg, Laurent ; Tomasic, Anthony ; Franklin, Michael J. ; Urhan, Tolga: Scrambling query plans to cope with unexpected delays (1996)

[AuMN02] Au, T. C. ; Muñoz-Avila, H. ; Nau, D.: On the complexity of plan adaptation by derivational analogy in a universal classical planning framework. In: *Advances in Case-Based Reasoning* (2002), S. 199–206

[AvHe00] Avnur, Ron ; Hellerstein, Joseph M.: Eddies: Continuously adaptive query processing. In: *ACM sigmod record*. Bd. 29, 2000, S. 261–272

[BaBu90] Bancilhon, François ; Buneman, Peter: *Advances in database programming languages*, *ACM Press frontier series*. New York, N.Y. : Reading, Mass : ACM Press ; Addison-Wesley Pub. Co, 1990 — ISBN 0201502577

[Banc89] Bancilhon, Fran\ccois: Query languages for object-oriented database systems: Analysis and a proposal. In: *Datenbanksysteme in Büro, Technik und Wissenschaft*, 1989, S. 1–18

[Barl91] Barletta, R.: An introduction to case-based reasoning. In: *AI Expert* Bd. Vol. 6 (1991), Nr. No. 8

[BeFI91] Bennett, Kristin P. ; Ferris, Michael C. ; Ioannidis, Yannis E.: *A genetic algorithm for database query optimization* : Computer Sciences Department, University of Wisconsin, Center for Parallel Optimization, 1991

[BeKP06] Bergmann, Ralph ; Kolodner, Janet ; Plaza, Enric: Representation in case-based reasoning. In: *The Knowledge Engineering Review* Bd. 20 (2006), Nr. 03, S. 209

[Berg02] Bergmann, R.: *Experience management: foundations, development methodology, and Internet-based applications*, *Lecture notes in computer science*, *Lecture notes in artificial intelligence*. Berlin ; New York : Springer, 2002 — ISBN 3540441913

[BFMV00a] Bouganim, Luc ; Fabret, Fran\ccoise ; Mohan, Chandrasekaran ; Valduriez, Patrick: Dynamic query scheduling in data integration systems. In: *Data Engineering, 2000. Proceedings. 16th International Conference on*, 2000, S. 425–434

[BFMV00b] Bouganim, Luc ; Fabret, Fran\ccoise ; Mohan, C. ; Valduriez, Patrick: A dynamic query processing architecture for data integration systems. In: *IEEE Data Eng. Bull.* Bd. 23 (2000), Nr. 2, S. 42–48

[BGWR81] Bernstein, Philip A. ; Goodman, Nathan ; Wong, Eugene ; Reeve, Christopher L. ; Rothnie Jr, James B.: Query processing in a system for distributed databases (SDD-1). In: *ACM Transactions on Database Systems (TODS)* Bd. 6 (1981), Nr. 4, S. 602–625

[BiCo89] Birnbaum ; Collins: Remindings and engineering design themes: a case study in indexing vocabulary. In: *Proceedings: Case-Based Reasoning Workshop*, *Proceedings: Case-Based Reasoning Workshop*. (1989)

[Bobe10] Boben, Rainu: *A tutorial on case-based reasoning*, COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY, 2010

[Bowm01] Bowman, Ivan T.: Hybrid Shipping Architectures: A Survey. In: *University of Waterloo February* (2001)

[CCDF03] Chandrasekaran, S. ; Cooper, O. ; Deshpande, A. ; Franklin, M. J. ; Hellerstein, J. M. ; Hong, W. ; Krishnamurthy, S. ; Madden, S. R. ; u. a.: TelegraphCQ: continuous dataflow processing. In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003, S. 668–668

[Chau98] Chaudhuri, Surajit: An overview of query optimization in relational systems. In: *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, 1998, S. 34–43

[ChMN98] Chaudhuri, Surajit ; Motwani, Rajeev ; Narasayya, Vivek: Random sampling for histogram construction: how much is enough? In: *ACM SIGMOD Record* Bd. 27 (1998), Nr. 2, S. 436–447

[CoGr94] Cole, Richard L. ; Graefe, Goetz: Optimization of dynamic query evaluation plans. In: *ACM SIGMOD Record* Bd. 23 (1994), Nr. 2, S. 150–160

[CWYA81] Chamberlin, D. D. ; Wade, B. W. ; Yost, R. A. ; Astrahan, M. M. ; King, W. F. ; Lorie, R. A. ; Mehl, J. W. ; Price, T. G. ; u. a.: Support for repetitive transactions and ad hoc queries in System R. In: *ACM Transactions on Database Systems* Bd. 6 (1981), Nr. 1, S. 70–94

[DeIR06] Deshpande, Amol ; Ives, Zachary ; Raman, Vijayshankar: Adaptive query processing. In: *Foundations and Trends® in* Bd. 1 (2006), Nr. 1, S. 1–140

[DGRT03] Demers, A. ; Gehrke, J. ; Rajaraman, R. ; Trigoni, N. ; Yao, Y.: The Cougar Project: a work-in-progress report. In: *Acm Sigmod Record* Bd. 32 (2003), Nr. 4, S. 53–59

[DKNW04] Dhraief, H. ; Kemper, A. ; Nejdl, W. ; Wiesner, C.: Distributed queries and query optimization in schema-based P2P-systems. In: *Proceedings of 13th World Wide Web Conference (WWW 2004)*, 2004

[Domi12] Domingos, Pedro: A few useful things to know about machine learning. In: *Communications of the ACM* Bd. 55 (2012), Nr. 10, S. 78–87

[EiGK95] Eickler, André ; Gerlhof, Carsten Andreas ; Kossmann, Donald: *A performance evaluation of oid mapping techniques* : Fakultät für Mathematik und Informatik, Universität Passau, 1995

[EiKK97] Eickler, André ; Kemper, Alfons ; Kossmann, Donald: Finding data in the neighborhood. In: *VLDB*, 1997, S. 336–345

[ElSh11] Elmasri, Ramez ; Shamkant, Navathe: *Fundamentals of database systems* : Michael Hirsch, 2011

[EpSW78] Epstein, Robert ; Stonebraker, Michael ; Wong, Eugene: Distributed query processing in a relational data base system. In: *Proceedings of the 1978 ACM SIGMOD international conference on management of data*, 1978, S. 169–180

[GBFR03] Garcés-Erice, L. ; Biersack, E. W. ; Felber, P. A. ; Ross, K. W. ; Urvoy-Keller, G.: Hierarchical Peer-to-Peer Systems. In: Kosch, H. ; Böszörményi, L. ; Hellwagner, H.

(Hrsg.) ; Goos, G. ; Hartmanis, J. ; Leeuwen, J. (Hrsg.): *Euro-Par 2003 Parallel Processing*. Bd. 2790. Berlin, Heidelberg : Springer Berlin Heidelberg, 2003 — ISBN 978-3-540-40788-1, 978-3-540-45209-6, S. 1230–1239

[GPFS02] Gounaris, Anastasios ; Paton, Norman W. ; Fernandes, Alvaro AA ; Sakellariou, Rizos: Adaptive query processing: A survey. In: *Advances in Databases* : Springer, 2002, S. 11–25

[GPSH02] Ghosh, A. ; Parikh, J. ; Sengar, V. S. ; Haritsa, J. R.: Plan selection based on query clustering. In: *Proceedings of the 28th international conference on Very Large Data Bases*, 2002, S. 179–190

[Grae93] Graefe, Goetz: Query evaluation techniques for large databases. In: *ACM Computing Surveys* Bd. 25 (1993), Nr. 2, S. 73–169

[Grae94] Graefe, G.: Volcano-an extensible and parallel query evaluation system. In: *Knowledge and Data Engineering, IEEE Transactions on* Bd. 6 (1994), Nr. 1, S. 120–135

[Grif97] Griffiths, A. D.: *Inductive generalisation in case-based reasoning systems*, Technical Report YCST 97/02, University of York Department of Computer Science, 1997

[GrMc93] Graefe, Goetz ; McKenna, William J.: The Volcano optimizer generator: Extensibility and efficient search. In: *Data Engineering, 1993. Proceedings. Ninth International Conference on*, 1993, S. 209–218

[GrWa89] Graefe, G. ; Ward, K.: Dynamic query evaluation plans. In: *ACM SIGMOD Record* Bd. 18 (1989), Nr. 2, S. 358–366

[GRZZ00] Gruser, J. R. ; Raschid, L. ; Zadorozhny, V. ; Zhan, T.: Learning response time for websources using query feedback and application in query optimization. In: *The VLDB Journal—The International Journal on Very Large Data Bases* Bd. 9 (2000), Nr. 1, S. 18–37

[HaHe99] Haas, Peter J. ; Hellerstein, Joseph M.: Ripple joins for online aggregation. In: *ACM SIGMOD Record* Bd. 28 (1999), Nr. 2, S. 287–298

[Hamm89] Hammond, Kristian J.: *Case-based planning: viewing planning as a memory task*, *Perspectives in artificial intelligence*. Boston : Academic Press, 1989 — ISBN 0123220602

[Hart75] Hartigan, John A.: *Clustering algorithms*, *Wiley series in probability and mathematical statistics*. New York : Wiley, 1975 — ISBN 047135645X

[HeYW00] Heidemann, John ; Ye, Wei ; Wills, Jack: Underwater networking research at USC/ISI

[HFCD00] Hellerstein, Joseph M. ; Franklin, Michael J. ; Chandrasekaran, Sirish ; Deshpande, Amol ; Hildrum, Kris ; Madden, Samuel ; Raman, Vijayshankar ; Shah, Mehul A.: Adaptive query processing: Technology in evolution. In: *IEEE Data Eng. Bull.* Bd. 23 (2000), Nr. 2, S. 7–18

[IbKa84] Ibaraki, Toshihide ; Kameda, Tiko: On the optimal nesting order for computing N-relational joins. In: *ACM Transactions on Database Systems* Bd. 9 (1984), Nr. 3, S. 482–502

[IFFL99] Ives, Zachary G. ; Florescu, Daniela ; Friedman, Marc ; Levy, Alon ; Weld, Daniel S.: An adaptive query execution system for data integration. In: : ACM Press, 1999 — ISBN 1581130848, S. 299–310

[ILWF00] Ives, Zachary G. ; Levy, Alon Y. ; Weld, Daniel S. ; Florescu, Daniela ; Friedman, Marc: Adaptive query processing for internet applications. In: *Departmental Papers (CIS)* (2000), S. 130

[INSS97] Ioannidis, Yannis E. ; Ng, Raymond T. ; Shim, Kyuseok ; Sellis, Timos K.: Parametric query optimization. In: *The VLDB Journal—The International Journal on Very Large Data Bases* Bd. 6 (1997), Nr. 2, S. 132–151

[Ioan96] Ioannidis, Yannis E.: Query optimization. In: *ACM Computing Surveys* Bd. 28 (1996), Nr. 1, S. 121–123

[IoCh91] Ioannidis, Yannis E. ; Christodoulakis, Stavros: On the propagation of errors in the size of join results. In: *ACM SIGMOD Record* Bd. 20 (1991), Nr. 2, S. 268–277

[IoKa90] Ioannidis, Y. E. ; Kang, Younkyung: Randomized algorithms for optimizing large join queries. In: : ACM Press, 1990 — ISBN 0897913655, S. 312–321

[IoKa91] Ioannidis, Y. E. ; Kang, Y. C.: Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In: *ACM SIGMOD Record*. Bd. 20, 1991, S. 168–177

[IoWo87] Ioannidis, Yannis E. ; Wong, Eugene: Query optimization by simulated annealing. In: : ACM Press, 1987 — ISBN 0897912365, S. 9–22

[JPKS00] Jagadish, H. V. ; Poosala, Viswanath ; Koudas, Nick ; Sevcik, Ken ; Muthukrishnan, S. ; Suel, Torsten: Optimal histograms with quality guarantees

[KaDe98] Kabra, Navin ; DeWitt, David J.: Efficient mid-query re-optimization of sub-optimal query execution plans. In: *ACM SIGMOD Record* Bd. 27 (1998), Nr. 2, S. 106–117

[Kirk84] Kirkpatrick, Scott: Optimization by simulated annealing: Quantitative studies. In: *Journal of Statistical Physics* Bd. 34 (1984), Nr. 5-6, S. 975–986

[Kolo92] Kolodner, Janet L.: An introduction to case-based reasoning. In: *Artificial Intelligence Review* Bd. 6 (1992), Nr. 1, S. 3–34

[Koo80] Koo, R. P.: *The optimization of queries in relational databases*, Case Western Reserve University, 1980

[Koss00] Kossmann, Donald: State of the art in distributed query processing

[KoSt00] Kossmann, Donald ; Stocker, Konrad: Iterative dynamic programming: a new class of query optimization algorithms. In: *ACM Transactions on Database Systems (TODS)* Bd. 25 (2000), Nr. 1, S. 43–82

[Kowa91] Kowalski, Andrzej: Case-based reasoning and the deep structure approach to knowledge representation. In: *Proceedings of the 3rd international conference on Artificial intelligence and law*, 1991, S. 21–30

[KrBa93] Kriegsman, M. ; Barletta, R.: Building a case-based help desk application. In: *IEEE Expert* Bd. 8 (1993), Nr. 6, S. 18–26

[KuSV00] Kumar, T. V. V. ; Singh, V. ; Verma, A. K.: Distributed query processing plans generation using genetic algorithm

[LiNS90] Lipton, Richard J. ; Naughton, Jeffrey F. ; Schneider, Donovan A.: Practical selectivity estimation through adaptive sampling. In: *ACM SIGMOD Record* Bd. 19 (1990), Nr. 2, S. 1–11

[Liu00] Liu, Mengmeng: An overview of adaptive query processing systems

[LMBL05] Lopez De Mantaras, R. ; McSherry, D. ; Bridge, D. ; Leake, D. ; Smyth, B. ; Craw, S. ; Faltings, B. ; Maher, M. L. ; u. a.: Retrieval, reuse, revision and retention in case-based reasoning. In: *The Knowledge Engineering Review* Bd. 20 (2005), Nr. 03, S. 215–240

[LMHD85] Lohman, Guy M. ; Mohan, C. ; Haas, Laura M. ; Daniels, Dean ; Lindsay, Bruce G. ; Selinger, Patricia G. ; Wilms, Paul F.: Query Processing in R*. In: Kim, W. ; Reiner, D. S. ; Batory, D. S. (Hrsg.) ; Brodie, M. L. ; Mylopoulos, J. ; Schmidt, J. W. (Hrsg.): *Query Processing in Database Systems*. Berlin, Heidelberg : Springer Berlin Heidelberg, 1985 — ISBN 978-3-642-82377-0, 978-3-642-82375-6, S. 31–47

[LVZC91] Lanzelotte, Rosana S. G. ; Valduriez, Patrick ; Ziane, Mikal ; Cheiney, Jean-Pierre: Optimization of nonrecursive queries in OODBs. In: Delobel, C. ; Kifer, M. ; Masunaga, Y. (Hrsg.) ; Goods, G. ; Hartmanis, J. (Hrsg.): *Deductive and Object-Oriented Databases*. Bd. 566. Berlin, Heidelberg : Springer Berlin Heidelberg, 1991 — ISBN 978-3-540-55015-0, 978-3-540-46646-8, S. 1–21

[MaVW98] Matias, Yossi ; Vitter, Jeffrey Scott ; Wang, Min: Wavelet-based histograms for selectivity estimation. In: *ACM SIGMOD Record* Bd. 27 (1998), Nr. 2, S. 448–459

[MCBD12] Martinez, Lourdes ; Collet, Christine ; Bobineau, Christophe ; Dublé, Etienne: The QOL approach for optimizing distributed queries without complete knowledge. In: : ACM Press, 2012 — ISBN 9781450312349, S. 91–99

[MFHH05] Madden, S. R. ; Franklin, M. J. ; Hellerstein, J. M. ; Hong, W.: TinyDB: An acquisitional query processing system for sensor networks. In: *ACM Transactions on Database Systems (TODS)* Bd. 30 (2005), Nr. 1, S. 122–173

[Mitc99] Mitchell, Melanie: An introduction to genetic algorithms. In: *Cambridge, Massachusetts London, England, Fifth printing* Bd. 3 (1999)

[MuCo08] Munoz-Avila, H. ; Cox, M.T.: Case-based plan adaptation: an analysis and review. In: *IEEE Intelligent Systems* Bd. 23 (2008), Nr. 4, S. 75–81

[NaSS86] Nahar, S. ; Sahni, S. ; Shragowitz, E.: Simulated Annealing and Combinatorial Optimization. In: : IEEE, 1986 — ISBN 0-8186-0702-5, S. 293–299

[Orac11] Oracle: *Class Enum from the Java API*. URL http://docs.oracle.com/javase/6/docs/api/java/lang/Enum.html. — Java Platform Standard Ed. 6

[Orac13] Oracle: *Java™ platform, standard edition 7 API specification*. URL http://docs.oracle.com/javase/7/docs/api/

[OsBr97] Osborne, Hugh ; Bridge, Derek: Models of similarity for case-based reasoning. In: *Procs. of the Interdisciplinary Workshop on Similarity and Categorisation*, 1997, S. 173–179

[OuBo04] Ouzzani, M. ; Bouguettaya, A.: Query processing and optimization on the web. In: *Distributed and Parallel Databases* Bd. 15 (2004), Nr. 3, S. 187–218

[OwKS05] Owais, Suhail SJ ; Krömer, Pavel ; Snaˇsel, Vaclav: Query optimization by Genetic Algorithms. In: *DATESO*. Bd. 129, 2005, S. 125–137

[OzVa11] Özsu, M. Tamer ; Valduriez, Patrick: *Principles of distributed database systems*. 3rd ed. Aufl. New York : Springer Science+Business Media, 2011 — ISBN 9781441988331

[Pfaf04] Pfaff, Ben: *An introduction to binary search Ttrees and balanced trees*. Bd. 0.1, 2004

[PHIS96] Poosala, Viswanath ; Haas, Peter J. ; Ioannidis, Yannis E. ; Shekita, Eugene J.: Improved histograms for selectivity estimation of range predicates. In: *ACM SIGMOD Record* Bd. 25 (1996), Nr. 2, S. 294–305

[PiCo84] Piatetsky-Shapiro, Gregory ; Connell, Charles: Accurate estimation of the number of tuples satisfying a condition. In: : ACM Press, 1984 — ISBN 0897911288, S. 256

[PoIo97] Poosala, Viswanath ; Ioannidis, Yannis E.: Selectivity estimation without the attribute value independence assumption. In: *VLDB*. Bd. 97, 1997, S. 486–495

[RDDP82] R., Williams ; D., Daniels ; D., Hass ; P., Walker: R*: An overview of the architecture. In: , *In Proc. 2nd Int. Conf. on Databases*. (1982), S. 1–28

[RFHK01] Ratnasamy, Sylvia ; Francis, Paul ; Handley, Mark ; Karp, Richard ; Shenker, Scott: *A scalable content-addressable network*. Bd. 31 : ACM, 2001

[Rich95] Richter, Michael M.: On the notion of similarity in case-based reasoning. In: *Mathematical and Statistical Methods in Artificial Intelligence* (1995), S. 171–184

[SABE94] Subrahmanian, V. S. ; Adah, S. ; Brink, A. ; Emery, J. ; Lu, A.: HERMES: a heterogeneous reasoning and mediator system paper, 1994

[SACL79] Selinger, P. Griffiths ; Astrahan, M. M. ; Chamberlin, D. D. ; Lorie, R. A. ; Price, T. G.: Access path selection in a relational database management system. In: : ACM Press, 1979 — ISBN 089791001X, S. 23

[SaLa91] Safavian, S. Rasoul ; Landgrebe, David: A survey of decision tree classifier methodology. In: *Systems, Man and Cybernetics, IEEE Transactions on* Bd. 21 (1991), Nr. 3, S. 660–674

[ScBe00] Schumacher, Jürgen ; Bergmann, Ralph: An Efficient Approach to Similarity-Based Retrieval on Top of Relational Databases. In: Blanzieri, E. ; Portinale, L. (Hrsg.): *Advances in Case-Based Reasoning*. Bd. 1898. Berlin, Heidelberg : Springer Berlin Heidelberg — ISBN 978-3-540-67933-2, S. 273–285

[Scou95] Scourias, John: Aspects of Client/Server Database Systems (1995)

[SeHa03] Sengar, V. S. ; Haritsa, J. R.: PLASTIC: reducing query optimization overheads through plan recycling. In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003, S. 676–676

[ShKM00] Shahabi, Cyrus ; Khan, Latifur ; McLeod, Dennis: A probe-based technique to optimize join queries in distributed internet databases. In: *Knowledge and information systems* Bd. 2 (2000), Nr. 3, S. 373–385

[ShYT93] Shekita, Eugene J. ; Young, Honesty C. ; Tan, Kian-Lee: Multi-join optimization for symmetric multiprocessors. In: *VLDB*. Bd. 93, 1993, S. 479–492

[SKBK00] Stocker, K. ; Kossmann, D. ; Braumandi, R. ; Kemper, A.: Integrating semi-join-reducers into state-of-the-art query processors. In: : IEEE Comput. Soc — ISBN 0-7695-1001-9, S. 575–584

[SLMK01] Stillger, M. ; Lohman, G. ; Markl, V. ; Kandil, M.: Leo-db2's learning optimizer. In: *Proceedings of the International Conference on Very Large Data Bases*, 2001, S. 19–28

[SMKK01] Stoica, Ion ; Morris, Robert ; Karger, David ; Kaashoek, M. Frans ; Balakrishnan, Hari: Chord: A scalable peer-to-peer lookup service for internet applications. In: *ACM SIGCOMM Computer Communication Review*. Bd. 31, 2001, S. 149–160

[SoIS10] Soliman, Mohamed A. ; Ilyas, Ihab F. ; Saleeb, Mina: Building ranked mashups of unstructured sources with uncertain information. In: *Proceedings of the VLDB Endowment* Bd. 3 (2010), Nr. 1-2, S. 826–837

[StMK97] Steinbrunn, Michael ; Moerkotte, Guido ; Kemper, Alfons: Heuristic and randomized optimization for the join ordering problem. In: *The VLDB Journal The International Journal on Very Large Data Bases* Bd. 6 (1997), Nr. 3, S. 191–208

[Ston86] Stonebraker, Michael: The case for shared nothing. In: *IEEE Database Eng. Bull.* Bd. 9 (1986), Nr. 1, S. 4–9

[Swam89] Swami, A.: Optimization of large join queries: combining heuristics and combinatorial techniques. In: : ACM Press, 1989 — ISBN 0897913175, S. 367–376

[SwGu88] Swami, Arun ; Gupta, Anoop: Optimization of large join queries. In: : ACM Press, 1988 — ISBN 0897912683, S. 8–17

[Tayl00] Taylor, John: *UK national e-Science center*. URL http://www.nesc.ac.uk/nesc/define.html

[TiDe03] Tian, Feng ; DeWitt, David J.: Tuple routing strategies for distributed eddies. In: *Proceedings of the 29th international conference on Very large data bases-Volume 29*, 2003, S. 333–344

[UrFA98] Urhan, Tolga ; Franklin, Michael J. ; Amsaleg, Laurent: Cost-based query scrambling for initial delays. In: *ACM SIGMOD Record* Bd. 27 (1998), Nr. 2, S. 130–141

[UrFr00] Urhan, Tolga ; Franklin, Michael J.: XJoin: A reactively-scheduled pipelined join operator. In: *Bulletin of the IEEE Computer Society Technical Comittee OnData Engineering* (2000)

[VaKH09] Valluri, S. ; Karlapalem, K. ; Hulgeri, A.: *Towards Empirically Driven Query Optimization* : Technical Report, IIIT Hyderabad, http://web2py. iiit. ac. in/publications/default/view_publication/techreport/55, 2009

[VaKi00] Vamsikrishna, M. V. ; Kian, T.: Subquery plan reuse based query optimization

[VaPa05] Valduriez, Patrick ; Pacitti, Esther: Data management in large-scale P2P systems. In: *High Performance Computing for Computational Science-VECPAR 2004* : Springer, 2005, S. 104–118

[VeBW04] Venugopal, Srikumar ; Buyya, Rajkumar ; Winton, Lyle: A grid service broker for scheduling distributed data-oriented applications on global grids. In: : ACM Press, 2004 — ISBN 1581139500, S. 75–80

[ViPa11] Vijay Kumar, T. V. ; Panicker, Shina: Generating query plans for distributed query processing using genetic algorithm. In: Liu, B. ; Chai, C. (Hrsg.) ; Hutchison, D. ; Kanade, T. ; Kittler, J. ; Kleinberg, J. M. ; Mattern, F. ; Mitchell, J. C. ; Naor, M. ; Nierstrasz, O. ; u. a. (Hrsg.): *Information Computing and Applications*. Bd. 7030. Berlin, Heidelberg : Springer Berlin Heidelberg, 2011 — ISBN 978-3-642-25254-9, 978-3-642-25255-6, S. 765–772

[Wadl03] Wadler, Philip: XQuery: a typed functional language for querying XML. In: *Advanced Functional Programming* : Springer, 2003, S. 188–212

[Wass94] Wasserman, Stanley: *Social network analysis: methods and applications*, *Structural analysis in the social sciences*. Cambridge ; New York : Cambridge University Press, 1994 — ISBN 0521382696

[WiAp93] Wilschut, Annita N. ; Apers, Peter MG: Dataflow query execution in a parallel main-memory environment. In: *Distributed and Parallel Databases* Bd. 1 (1993), Nr. 1, S. 103–128

[WLGP08] Watson, Paul ; Lord, Phillip ; Gibson, Frank ; Periorellis, Panayiotis ; Pitsilis, Georgios: Cloud Computing for e-Science with CARMEN. In: *2nd Iberian Grid Infrastructure Conference Proceedings*, 2008, S. 3–14

[Wort97] Wortmann, Karl L.: *On case representation and indexing in a case-based reasoning system·for waste management*, Department of Computer Science and Information Systems University of Natal, 1997

[WoYo76] Wong, Eugene ; Youssefi, Karel: Decomposition—a strategy for query processing. In: *ACM Transactions on Database Systems (TODS)* Bd. 1 (1976), Nr. 3, S. 223–241

[ZhLa00] Zhu, Q. ; Larson, P.: Classifying local queries for global query optimization in multidatabase systems.pdf. In: *International Journal of Cooperative Information Systems* Bd. Vol. 9 (2000), Nr. No. 3, S. 315–355

ANNEX A
DATA LOCATION AWARE QUERY LANGUAGE

The Data Location Aware Query Language (DLAQL) is a declarative data manipulation language for the UBIQUEST system. It is used to express declarative queries and updates over the global schema. Its syntax is based on SQL syntax with some specific operators and limitations. DLAQL queries/updates are sent to a node directly by applications or are automatically generated during query execution process (subqueries).

With DLAQL, one can specify the scope of each individual query/update (whether local data of the node executing the query/update or all data available over the network). The node(s) on which updates are stored can also be specified on a per update basis. These storing nodes can be explicitly specified (i.e. node identifiers) or implicitly using a subquery returning node identifiers. DLAQL permits to express selection, projection, join and aggregate computation. Imbricated queries can also be explored.

A.1 DLAQL SYNTAX

This section presents the notations used for describing the syntax of the DLAQL language for querying data, inserting data, updating data and deleting data.

A.1.1 Notations

We will use some specific notations in the syntax descriptions. These ones are the following:

- Concepts are represented like this: <concept>.
- [This is an optional part].
- Parts like that * can be repeated several times. A coma (,) is used to separate them. The coma is part of the syntax.
- Keywords and characters that are part of the syntax are represented in **bold font**.

A.1.2 Syntax for querying data

The general syntax for a query is the following:

```
<query> :=
SELECT <itemset> [SCOPE IS <scope>][DISTINCT] (*| (<itemset> | <alias>).* |
<select expression>)*
FROM (<itemset> [<alias>])*
[WHERE <condition> ]
[UNION [ALL] <query> ] ;

<scope>:= (<value>)* | <query>

<select expression> :=
<expression>
| MIN( [(<itemset> | <alias>).]<attribute> )
| MAX( [(<itemset> | <alias>).]<attribute> )
| AVG( [(<itemset> | <alias>).]<attribute> )
| COUNT( [(<itemset> | <alias>).](<attribute> | *) )
| ( <select expression> )
| <select expression> <operator> <select expression>

<expression> :=
[(<itemset> | <alias>).]<attribute>
| <value>
| ( <expression> )
| <expression> <operator> <expression>
<operator> :=
+
| -
| *
| =

<condition> :=
[(<itemset> | <alias>).]<attribute> <comparison operator>
<expression>
| [(<itemset> | <alias>).]<attribute> IN ( (<expression>)————————————————————— )
| [(<itemset> | <alias>).]<attribute> IN ( <query> )
| EXISTS ( <query> )
| ( <condition> )
| <condition> AND <condition>
| <condition> OR <condition>
| NOT <condition>

<Comparison operator> :=
<
| <=
```

| =
| >=
| >
| <>

The interpretation of a query is the following:

1. The FROM clause computes a cartesian product of specified itemsets. Each itemset can be referred to with an optional alias.

2. If specified, the WHERE clause apply a filter to the result of the Cartesian product. Selected items will participate in the remaining of the computation. Filtering conditions can include comparison between an item attribute and an expression, testing if an attribute belongs to a list of value (IN) – the list being extensionally or intentionally expressed –, an existence test which is verified when the subquery does not return an empty result, or any logical combination of conditions.

3. The SELECT clause builds the resulting itemset for the query. Each item is composed by the specified list of select expressions computed on each selected item. Select expression can combine classical expression and aggregate function computation on attributes. The optional argument DISTINCT is used to eliminate duplicates in the result of the query. The SCOPE IS is an optional argument defines the scope of the sources that the query includes. The scope may correspond to one or more values of node identifiers, or a query that return such values. The keywords LOCAL and SELF indicate that only local data of the node where the query is evaluated will be considered. By default all data available over the network are queried (i.e. the scope is global).

4. If specified, the set operators UNION (with duplicates if the ALL option is set) is applied on the result of two queries. Of course both resulting itemsets must have the same schema.

A.1.3 Syntax for inserting data

The syntax for inserting data is the following:

<insert> :=
INSERT INTO <itemset> (
VALUES (<value>*) | (<query>))
[STORE ON <location>*] ;
<location> :=
SELF
| <node identifier>
| (<query>)

The interpretation of an insertion is the following:

1. The INSERT INTO clause specifies in which global itemset new items will be inserted.

2. The items that are to be inserted can be specified extensionally (using VALUES and a list of values) or intentionally via a subquery.

3. The optional clause STORE ON allows the user to specify on which node(s) data will be inserted. The parameter of this clause is a list of node identifiers that can be expressed extensionally - using

node identifiers or SELF that corresponds to the identifier of the node where the query is evaluated) - or intentionally with a subquery returning node identifiers.

A.1.4 Syntax for updating data

The syntax for updating data is the following:

```
<update> :=
UPDATE <itemset>
SET (<attribute> = <expression>)*
[WHERE <condition>]
[STORE ON <location>];
```

The interpretation of an update is the following:

1. The UPDATE clause specifies the itemset that is to be updated.

2. The SET clause specifies the new value of attributes. The expression can exploit the old value of the attribute.

3. The optional WHERE clause selects the items that are to be updated.

4. The optional STORE ON clause specifies the location of updated items. Items can be moved from nodes to other nodes during an update.

A.1.5 Syntax for deleting data

The syntax for deleting data is the following:

```
<delete> :=
[LOCAL] DELETE FROM <itemset>
[WHERE <condition>] ;
```

The interpretation of a delete is the following:
1. The DELETE FROM clause specifies the itemset from which items will be deleted. The LOCAL optional argument specifies that only local data of the node where the delete is evaluated will be removed.

2. The optional WHERE clause selects the items that are to be deleted.

A.2 EXAMPLES

This section presents some examples of DLAQL queries and updates.

A.2.1 On demand routing

A node is asking for the next hop to reach a destination D. Only local data are queried.

Global schema:

- **Route**(NodeId, NextHop, Dest, HopNumber)

Query:

```
SELECT SCOPE IS LOCAL NextHop
FROM Route
WHERE Dest = D;
```

This query is executed by the node demanding the route (NextHop).

A.2.2 Adds distribution over vehicular networks

The objective is to send the address of merchants that are located in a city to vehicles that will pass by this city. The path followed by vehicle is known (e.g. GPS).

Global schema:

- **Merchants**(MerchantName, City)
- **Ads**(Ad, MerchantName)
- **PATH**(NodeId, Order, City)
- **ReceivedAds**(NodeId, Ad)

Query:

```
INSERT INTO ReceivedAds (
       SELECT P.NodeId, A.Ad
       FROM Ads A, Merchants M, Path P
       WHERE A.MerchantName = M.MerchantName
       AND M.City = P.City;
       )
STORE ON P.NodeId;
```

This query is sent to all nodes containing Merchant data.

A.2.3 Virtual world gaming

The virtual world is divided in several zones. Participants (nodes) put their avatars into zones. All avatars that are located are neighbors.

Centralized server

There is only one server for all participants – its identifier is *ServerId*–. The server stores the position (i.e. a zone identifier) and the identifier of the owner for all avatars. All updates are done on the server. Clients send queries to the server to

view the environment (i.e. avatars that are in the same zone) of their avatars or to

update data (the result is not stored on the clients). Clients only store information

about their own avatars.

Global schema:

- **Positions**(Avatar, Zone, OwnerId)

Queries:

- Viewing avatars in the same zone than the avatar named *MyAvatar*

```
SELECT *
FROM SCOPE IS LOCAL Positions
WHERE Zone = (SELECT P.Zone FROM Positions P WHERE P.Avatar = MyAvatar);
```

This query is executed on the server only.

- Entry of a new avatar –named *MyAvatar*– owned by *MyId* in the zone *MyZone*

```
INSERT INTO Positions
VALUES (MyAvatar, MyZone, SELF)
STORE ON ServerId, SELF;
```

This global query is initiated by the client.

- Movement of the avatar named *MyAvatar* to the position *NewZone*

```
UPDATE Positions
SET Zone = NewZone
WHERE Avatar = MyAvatar;
```

Updates are done in place; in this case: on the client initiating the query and on the server.

- The avatar named *MyAvatar* exits the game

```
DELETE FROM Positions
WHERE Avatar = MyAvatar;
```

Peer-to-Peer environment

We decide to store on one node all information about avatars that are in the same zone than their own avatars. Queries concerning a given avatar are issued by nodes that own it and are evaluated in a distributed fashion (except if specified in the query).

Global Schema:

- **Positions**(Avatar, PosX, PosY, NodeId)

Queries:

- Viewing avatars that are in the same zone as the avatar named *MyAvatar*

```
SELECT DISTINCT *
FROM SCOPE IS LOCAL Positions
WHERE Zone = (
       SELECT P.Zone
       FROM SCOPE IS LOCAL Positions P
       WHERE P.Avatar = MyAvatar );
```

- Entry of a new avatar named *MyAvatar*, owned by the node emitting the query, in the zone *MyZone*

-- Test if the zone corresponding to the new position is already known

```
SELECT Avatar
FROM SCOPE IS LOCAL Positions
WHERE Zone = MyZone;
```

-- If unknown zone (previous query returned no result)

```
INSERT INTO Positions (
SELECT DISTINCT *
FROM Positions
WHERE Zone = MyZone
)
STORE ON SELF;
```

-- Insert the new avatar

```
INSERT INTO Positions

VALUES (MyAvatar, MyZone, SELF)
STORE ON SELF,(
SELECT DISTINCT P.OwnerId
FROM Positions P
WHERE P.Zone = MyZone
);
```

- Movement of the avatar named *MyAvatar* to the position *NewZone*

-- Test if the new zone is already known

```
SELECT Avatar
FROM SCOPE IS LOCAL Positions
WHERE Zone = NewZone;
```

-- If unknown zone

```
INSERT INTO Positions (
SELECT DISTINCT *
FROM Positions
WHERE Zone = NewZone
)
STORE ON SELF;
```

-- Update is done by delete/insert

```
DELETE FROM Positions
WHERE Avatar = MyAvatar;
```

```
INSERT INTO Positions
VALUES (MyAvatar, NewZone, SELF)
STORE ON SELF, (
SELECT DISTINCT P.OwnerId
FROM Positions P
WHERE P.Zone = NewZone
);
-- Cleaning of locally useless zones
LOCAL DELETE FROM Positions
WHERE Zone NOT IN (
LOCAL SELECT P.Zone
FROM Positions P
WHERE P.OwnerId = SELF
);
```

- The avatar named *MyAvatar* exits the game

```
DELETE FROM Positions
WHERE Avatar = MyAvatar;
```

Hybrid approach

In this case, there are some node per zone acting as servers in the virtual environment. Each client sends its queries to the servers where the avatar is. Each client stores information concerning its own avatars.

Global Schema:

- **Positions**(Avatar, PosX, PosY, NodeId)

- **Servers**(Zone, ServerId)

Queries:

- Viewing avatars that are in the same zone than the avatar named *MyAvatar*

```
SELECT DISTINCT *
FROM Positions
WHERE Zone = (
SELECT P.Zone
FROM Positions P
WHERE P.Avatar = MyAvatar
);
```

- Entry of a new avatar named *MyAvatar* owned by the query emitter in the zone *MyZone*

```
INSERT INTO Postitions
VALUES (MyAvatar, MyZone, SELF)
STORE ON SELF, (
SELECT ServerId
```

```
FROM Servers
WHERE Zone = MyZone
);
```

- Movement of the avatar named *MyAvatar* to the zone *NewZone*

```
-- Update is done by delete/insert
DELETE FROM Positions
WHERE Avatar = MyAvatar;
INSERT INTO Positions
VALUES (MyAvatar, NewZone, SELF)
STORE ON SELF, (
        SELECT ServerId
        FROM Servers
        WHERE Zone = NewZone
);
```

- The avatar named *MyAvatar* exits the game

```
DELETE FROM Positions
WHERE Avatar = MyAvatar;
```