2009 Mexican International Conference on Computer Science

Query optimization using case-based reasoning in ubiquitous environments

Lourdes Angélica Martínez-Medina †
Research Center in Information and Automation Technologies CENTA
UDLA-P, Puebla, México
Lourdes-Angelica.Martinez-Medina@imag.fr

Christophe Bibineau ‡
Grenoble Informatics Laboratory
Grenoble Institute of Technology INP
Grenoble, France
Christophe.Bobineau@imag.fr

Jose Luis Zechinelli-Martini §
Research Center in Information and Automation Technologies CENTIA
UDLA-P, Puebla, México
Joseluis.Zechinelli@udlap.mx

***Abstract*—Query optimization is a widely studied problem, a variety of query optimization techniques have been suggested. These approaches are presented in the framework of classical query evaluation procedures that rely upon cost models heavily dependent on metadata (e.g. statistics and cardinality estimates) and that typically are restricted to execution time estimation. There are computational environments where metadata acquisition and support is very expensive. Additionally, execution time is not the only optimization objective of interest. A ubiquitous computing environment is an appropriate example where classical query optimization techniques are not useful any more. In order to solve this problem, this article presents a query optimization technique based on learning, particularly on case-based reasoning. Given a query, the knowledge acquired from previous experiences is exploited in order to propose reasonable solutions. It is possible to learn from each new experience in order to suggest better solutions to solve future queries.**

***Keywords*-classical query optimization techniques, ubiquitous computing environment, metadata, learning, case-based reasoning, similarity function**

I. INTRODUCTION

A wide variety of query optimization techniques as semantic, parametric, and query optimization via probing queries have been suggested. Even though these approaches allow the efficient query processing they are presented in the framework of classical query evaluation procedures that rely upon cost models heavily dependent of metadata (e.g. statistics and cardinality estimates). There exist different computational environments where no metadata are available. This characteristic hampers the efficient information access. An example of this type of environments is ubiquitous computing environment that integrates information from autonomous, heterogeneous, and dynamic computational tools and applications (software applications, web services, calculation tool, database systems, etc.) as well as electronic devices (sensor networks, PDAs, cell phones, etc.) located in a distributed fashion [1] [2]. Efficient methods of query processing are crucial for accessing information at any time, from any place and by any resource. Query optimization is essential to improve the performance of query processing. Challenges for query optimization in ubiquitous environments are discussed in this paper.

Resources in ubiquitous environments are heterogeneous, they are also dynamic and autonomous. Additionally, these resources are distributed in different locations. Another characteristic is that they present physical limitations that constrain their appropriate operation, e.g. processing and storage capability, energy consumption, location, among others. Finally, one of the most important characteristics that we addressed in this work is the lack of metadata. This characteristic results from the conjunction the others (mobility, dynamicity, autonomy, etc.). Typically, systems related to data management and data retrieval require many types of metadata (e.g. global schema), this information is essential for classical query optimization techniques as we will explain later. Below is a brief description for each feature [3]:

1) **Heterogeneity.** Ubiquitous computing environments integrate an extensive range of computational resources and electronic devices such as portable computers, PDAs, cell phones, censor networks, embedded systems, among others. These devices present important differences related to their physical and logical characteristics.
2) **Dynamicity.** Resources in ubiquitous environments change continuously, many of these changes are due to their mobility, e.g. when a resource is moved, its location and availability change. Also, the communication network properties and the resources with which it interacts vary.
3) **Distribution.** Resources in ubiquitous environments are distributed in different locations within a physical space where users interact to carry out different activities. The information used to complete these activities is stored or produced by these resources. The distributed access to this information must be performed in order to make it available at any time and location.
4) **Autonomy.** Resources integrated in ubiquitous environments are autonomous; this means that they can change their availability status at any time (they can be available to interact with other resources within the environment or not) e.g every type of resource

(mobile or stationary) can be disconnected during its interaction with other resources and there is no guarantee that it will be available again.

5) **Physical constrains.** Resources present physical limitations that constrain their appropriate operation, e.g. processing and storage capability, energy consumption, location, among others. A device or a process is capable to complete a task only if it counts with the appropriate computational resources. It is convenient to optimize the task performance based on specific critical resource.
6) **Metadata lack.** The resource characteristics previously mentioned make difficult the acquisition and maintenance of metadata like cardinality and statistics associated to data values. There is not a global schema in ubiquitous computational environments, its maintenance is very expensive due to the constant changes in the environment.

In ubiquitous computational environments metadata acquisition and maintenance is very expensive. Ubiquitous computing environments must provide a set of methods to retrieve information from available resources. The properties that characterize resources in ubiquitous environment represent new challenges for query processing. Some of them refer to lack of metadata required for estimating the evaluation cost of execution plans (possible execution plans to compute the results of a query) as a result of a highly variable execution context. Others are related to physical constrains of e electronic devices and computational tools. The need to optimize a query according to different parameters (user requirements) and not just according to execution time represents another significant challenge.

In this work we propose a query optimization technique capable to deal with the challenges of a context where metadata acquisition and maintenance is very costly and difficult. Also we address the necessity for optimize a query according to different parameters of interest. Our work suggests a query optimization technique that does not require metadata and that is capable to estimate the cost of an execution plan according to user requirements.

The remaining of this paper is organized as follows. Section 2 presents classical query optimization process. Section 3 presents the impact of ubiquitous environments on query optimization. Section 4 presents our approach exploiting machine learning techniques, more specifically case reasoning. Section 5 discusses the similarity notion among queries. Section 5 introduces the software architecture of a query optimizer. Section 7 presents related works. Section 8 shows first experimental performance evaluation results. Finally, Section 9 presents the conclusions of our work.

II. Classical query optimziation

Evaluation cost models used for most of classical query optimization techniques are tightly tied to metadata use. To support this affirmation, we describe the classical query optimization process. In a distributed environment, this process is composed by three phases: logical, global, and physical. Logical and physical optimization phases are related to centralized environments. Global optimization is required in a distributed environment. Figure 1 illustrates the optimization phases of the typical optimization process.

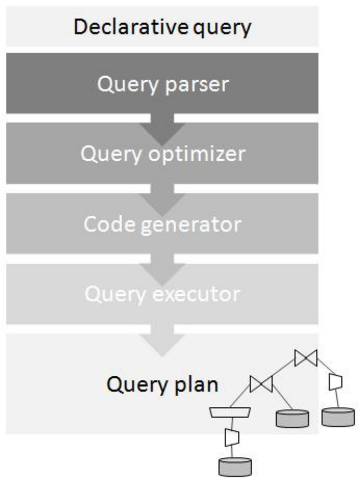

Figure 1. Phases of the optimization process

Each phase require different metadata types and has different optimization objectives. It is important to remark that in ubiquitous environments is interesting to achieve the same objectives. However, it is impossible to apply the same techniques due to environment characteristics. For this reason, it is needed to find other alternatives for optimizing a query in accordance to specific parameters (e.g. memory, CPU, and energy).

A. Logical optimization

Logical query optimization aims to reduce the number of tuples combined as intermediate results. It is nedded to decide the appropriate order for applying selection, projection, and join operators. There are heuristics in charge to achieve this goal. In case of selection and projection, the heuristics indicate that these operations must be applied as soon as possible in order to reduce the number of data that will be combined. Furthermore, metadata is required to decide the order in which join operations must be performed (e.g. relation cardinality and data values distribution). The result of this phase is an algebraic query tree that optimizes the order in which operators must be applied.

Figure 2 illustrates two trees composed by algebraic operators. The tree (a) is not the best plan due to the selection operation is applied after a join. On the other hand, tree (b) is the optimal algebraic plan due to all selection and projection operations are applied as soon as possible.

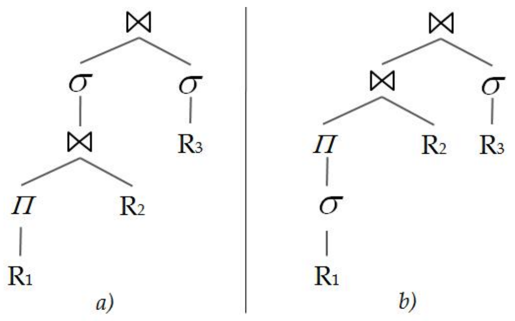

Figure 2. Algebraic query trees

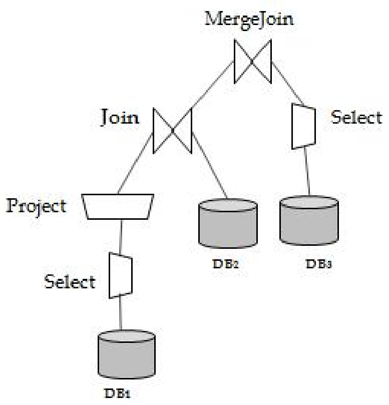

Figure 3. Query execution plan

B. Global optimization

Global optimization aims to minimize communication cost related to interactions among resources and a set of views (e.g. global view). In ubiquitous computational environments there is not a global view due to its expensive maintenance and constant changes in the environment. The global optimizer divides the execution tree that was produced by the logical optimizer and decides the place to perfor each part . The new execution tree includes communication operators e.g. work of activities performed for a set of resources.

C. Physical optimization

Physical optimization techniques aim to reduce disk access for retrieving requested data as well as to minimize execution time for evaluating query execution plans. An execution plan is formed by algorithms that implement each operator of the algebraic tree. Different implementations for each algebraic operator have been proposed. Given an algebraic tree, this optimization phase produces all corresponding execution plans that specify the implementation of each algebraic operator. To calculate the execution time of each algorithm and to minimize the disk access, metadata related to execution context is required, e.g. the algorithms to perform a join operation are nested-loop join, merge join, and hash join.

Figure 3 provides an example of a query execution plan that corresponds to the algebraic tree that was produced in the local optimization phase.

III. Ubiquitous computing impact in query optimization

Query optimization is essential to improve query evaluation. Classical query optimization techniques are not capable to deal with the challenges that ubiquitous environment characteristics imply. We are particularity interested in the optimization problems related to lack of metadata and optimization objective customization. Classical query optimization techniques are based cost models (functions for estimating the cost of execution plans) on statistics and metadata that is not available [4] [5]. Moreover, classical query optimization techniques typically generate query execution plans that are optimized according to a single dimension, query execution time [6].

It is needed to deal with metadata absence; useful knowledge must be obtained from previously executed queries and be managed and exploited by means of automatic learning techniques. The goal is to improve or acquire new capabilities from experience related to some specific tasks. We propose an approach for query optimization in ubiquitous computing environments based on machine learning in order to address the problem related to lacking of metadata [3]. Moreover, the query evaluation time is no longer the main optimization objective in ubiquitous computing environments, where computational tools and electronic devices have physical limitations (e.g. energy), in addition they are distributed, dynamic, and autonomous. Our work proposes a query optimization technique that allows the user customize the optimization objective according to his requirements.

IV. Query optimization based on lerning

The general idea of this approach is to learn from past experiences. An experience is the knowledge gained from a problem resolution. Learning is defined as the acquisition of knowledge in order to improve the behavior or to acquire new capabilities from previous experiences. Within computational domain, the subdiscipline concerned with learning is called machine learning [5]. It is an artificial intelligence subdiscipline in charge of designing and developing methods that allow computers to automatically learn in order to improve or create specific capabilities. There are different automatic learning methods, from which, we are interested in case-base reasoning [7].

In our approach, the knowledge gained from an experience is composed by the execution plan that solves the query and the computational resources that were consumed during the execution of the plan. Given a new query Q, an existent query plan is retrieved if it can be adapted to Q. Also it is required to verify if it is possible to accomplish its execution with the computational resources available at the moment of the query execution (e.g. memory, CPU, and energy).

A. *Case-based reasoning*

Case-based reasoning is a machine learning approach that proposes a reasoning process that aims to solve new problems using the experience gained when similar problems are solved. A case is the minimum unit of reasoning. It consists of (i) a problem description, (ii) its correspondent solution, and, (iii) a set of annotations that describe how the solution was derived. Case based reasoning has been formalized as a four-step process: retrieve, reuse, review and retain [7].

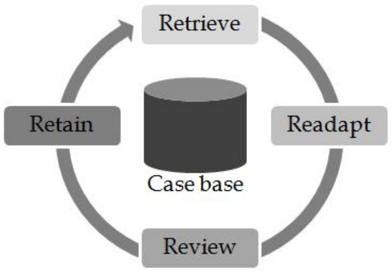

Figure 4. Case-based reasoning process

Figure 4 shows the four step case-based reasoning process. First, relevant cases must be retrieved to solve a new problem. Then, at the reusing step, an adjustment between the solutions offered by the relevant case and the problem requirements must be done. At the reviewing step, the new solution must be verified in the real world (or simulation). Finally, after the solution has been successfully adapted to the target problem and pertinently verified, the new experience must be stored as a new case in memory; this is accomplished in the retaining step. In order to propose a strategy based on machine learning, this work adapts case based reasoning to query optimization, which is also appropriate when dealing with the lack of metadata characterized by ubiquitous environments.

B. *Case-based reasoning adaptation to query optimization*

We propose query optimization strategy that adapts case based reasoning in order to provide optimal execution plans to solve new queries. This strategy recovers, adapts or generates execution plans using the knowledge acquired from the experience to optimize and execute similar queries [8].

According to this approach, a case corresponds to the representation of the knowledge acquired from the experiences obtained from the optimization and evaluation of previous queries. A problem corresponds to a new query submitted in the ubiquitous computing environment. The solution is represented by the corresponding execution plan.

1) Query: A query associated to a problem is defined by three clauses *selectClause*, *fromClause*, and *whereClause*. The *selectClause* specifies the attributes that must be projected as query result. The *fromClause* specifies the data sets that contain the information requested in the query. The *whereClause* specifies the set of conditions (for data selection and data combination or join) that must be verified by the data to form part of the query result.

Figure 5 illustrates the model that we propose for representing a query. In a query, selection and join operations are the most important and most frequent.

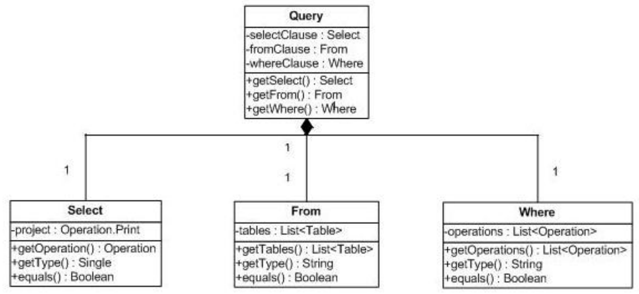

Figure 5. Query representation (UML diagram)

Each operation is defined by a type (e.g. Select or Join), a set of attributes, and a specific condition. Selection operation is expressed by a condition of the form *condition(attrexp, cnstexp)*, where *attrexp* represents the selection attribute and *cnstexp* a suitable constant value. Join operation is expressed by a condition of the form *condition(attrexp.a, attrexp.b)*, where *attrexp.a* and *attrexp.b* represent the join attributes.

The condition expresses a comparison (e.g. equal, different, lower, greater, etc.) between the data values of a table *attrexp* and a constant value or another attribute. An attribute is defined by an *id*, its *name*, and the *table* to which it corresponds. The following example allows us to appreciate the *whereClause* representation of a query Q as a set of its selection and join operations.

Query 1

SELECT Rest.nom FROM Resto,Ville,Region
WHERE Region.nom='RA' AND Resto.spec='It' AND
Resto.vil=Ville.nom AND Ville.numDep=Region.numDep

The query Q includes two selection operations *Region.nom=RA* (o_1) and *Resto.spec = 'It'* (o_2), as well as two join operations *Resto.vil=Ville.nom* (o_3) and *Ville.numDep=Region.numDep* (o_4). They apply the *Equal* condition that express the comparison operator of equality. The operation set related to Q is denoted by $Q_s = \{o_1, o_2, o_3\}$. It is possible to apply different conditions to the same attribute(s), i.e. o_x = *Greater(R.a$_n$,value)* and o_y = *GreaterOrEqual(R.a$_n$, value)*, select the values that are greater/greater or equal to some specific attribute that pertains to the relation *R*.

The attributes that are involved in operations of *Q* are *numReg (a_1)* and *nom (a_3)* that pertain to *Ville (R_1)* relation, *spec (a_2)* and *vil (a_4)* that pertain to *Resto ((R_2))* relation and *nom (a_5)* and *num (a_6)* that pertains to *(Region R_3)* relation.

We propose the concept of operation family in order to group operations that include the same condition applied to the same attributes and for this reason, the same relations. Two operations o_x and o_y pertain to the same operation family if they are of the same type (selection or join) and involve the same attributes (each of them must pertains to the same data source respectively). An operation family is represented as follows:

(1) $\Im_{R.an} = \{o_n \mid o_n = \textit{condition(R.a}_n\textit{,value)}\}$

The operation family $\Im_{R.an}$ is composed by the operations set o$_n$ with a condition of the form *condition(R.a$_n$, value)*, where a$_n$ is an attribute that pertains to the relation *R* and the condition denotes the set of all possible comparison operators: Equal, EqualOrLower, Lower, GreaterOrEqual, Greater and Different. All the queries are associated to a set of operation families. The *whereClause* of a query *Q* is defined by an operations set. These operations are members of different operation families: $\Im_{R1.a1}$, $\Im_{R2.a2}$ and $\Im_{R1.a3,R2.a4}$. Equation (2) shows the operation families Q$\Im$ that are associated to each operations in *Q*.

(2) $Q\Im = \{\Im_{R1.a1},\Im_{R2.a2},\Im_{R1.a3,R2.a4},\Im_{R2.a4,R3.a5}\}$

Each different combination of operation families $\Im_{R.an}$ conforms a class description, i.e. the class C_n defined by the operation families in (3). The queries are classified in a set of classes.

(3) $C_n = \{\Im_{Rn.an},\Im_{Rm.am},\Im_{Rn.ap,Rm.aq},\Im_{R2.a4,R3.a5}\}$

The class C_n is composed by all queries Q$_n$ that contain at least one operation that pertains to each of the specified families as defined in (4). This means, a query Q_n pertains to the class C_n if and only if for all operation family $\Im$ that describes C_n, exists an operation o_n in Q_n such as this operation is of the form of the operation family $\Im$.

(4) $Q_n \in C_n$ *iff* $(\forall\ \Im_{\ Rn.an} \in C_n)\ \exists\ ((o_n \in Q_n) \wedge (o_n \in \Im_{\ Rn.an}))$

According to the query *Q* presented above, the selection operation o$_1$ pertains to operation family $\Im_{\ R3.a5}$, the selection operation o$_2$ pertains to operation family $\Im_{\ R2.a2}$, the join operation o$_3$ pertains to operation family $\Im_{\ R2.a4,R1.a3}$, and the join operation o$_4$ that pertains to the operation family $\Im_{\ R1.a1,R3.a6}$. The operator and the attribute value are not important to determine the operation family to which a specific operation pertains, the important knowledge is related to the operation type and the attribute(s) included in the operation. The operation families described before make up a class a). Any query that is composed by operations that pertain to the families described before pertains to the same class b).

a) $C = \{\Im_{R3.a5},\Im_{R2.a2},\Im_{R2.a4,R1.a3}\ \textit{and}\ \Im_{\ R1.a1,R3.a6}$
b) $q \in C$ *iff* $(\forall\Im_{Rn.an} \in C_n)\exists((o_n{\in}q)\wedge(o_n{\in}\Im_{Rn.an}))$

A query is the medullar part of knowledge in the definition of a problem and a case. The problem specifies a optimized query, the optimization parameters, (e.g. memory, energy, and CPU) and measures related to computational resources available for query execution. The case specifies the optimized query, the solution to solve the query (query plan), and the measures related to computational resources that were consumed by the query execution. It is clear that the problem is not only the query by itself because it also includes the optimization objective and the circumstances under the query must be solved, it varies according to user requirements.

The query is the piece of knowledge that links a problem with the existent cases, i.e., the cases that contain a query similar to the query included in the problem can be useful to solve the new query; however this is the only characteristic of interest for us. It is also necessary to pay attention on the computational resources consumed by the query and those that are available at the moment that the new query will be executed as well as in the optimization objective that can changes each time the query is executed.

2) Problem: A problem is composed by a query, a context execution representation, and an optimization objective. Figure 6 illustrates the components of a problem.

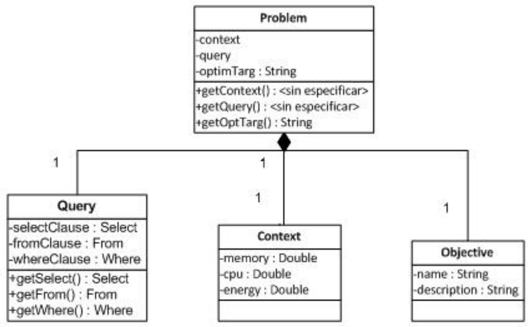

Figure 6. Problem representation (UML diagram)

In this representation, a query is the optimization target that must be solved. The context is described as a set of couples of the form *<attribute, value>* that represents measure of the computational resources available when the query is executed. These attributes may include CPU charge, available memory, and remaining energy, among others. Finally, the optimization objective indicates the resource or set of resources that will be optimized, e.g. minimize energy consumption. Figure 7 shows an example.

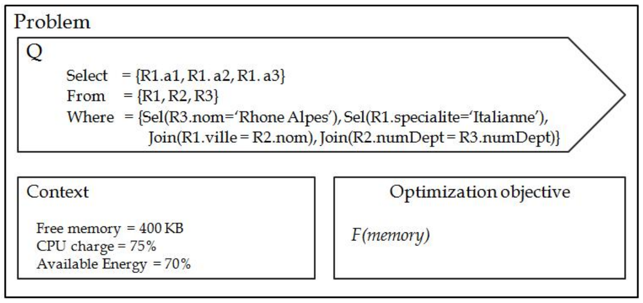
Problem
Q
Select = {R1.a1, R1. a2, R1. a3}
From = {R1, R2, R3}
Where = {Sel(R3.nom='Rhone Alpes'), Sel(R1.specialite='Italianne'), Join(R1.ville = R2.nom), Join(R2.numDept = R3.numDept)}
Context
Free memory = 400 KB
CPU charge = 75%
Available Energy = 70%
Optimization objective
F(memory)

Figure 7. An example of a problem

The set of touples that represent the instance of context depicted on Figure 7 is: *Context* = $\{$*<memory, 400>, <CPU, 75>, <energy, 70>*$\}$. Finally, the optimization objective indicates the resource or resources from which their consumption must be optimized. Typically, optimization means minimize the utilization of these resources. According to our example, the optimization objective is minimize the memory consumption specified by *F(memory)*.

3) Case: A case is composed of a query, a solution (query plan) and a set of evaluation measures used to express the optimization objective of a query. Figure 8 illustrates the components of a case.

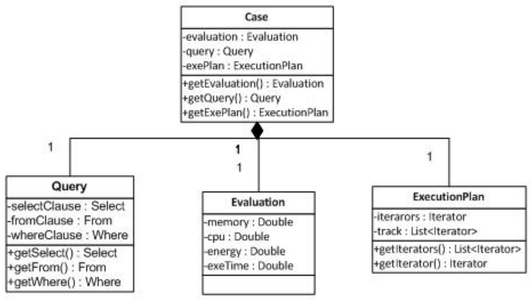

Figure 8. Case representation (UML diagram)

In a case, a query corresponds to an optimization target that has been evaluated and solved. The solution corresponds to the physical execution plan that solves the query; typically, a tree where nodes represent algorithms (implementations of query operators) and links represent data flows among operators. Finally, the evaluation corresponds to a set of measures collected during the query execution. These measures are represented as couples of the form *<attribute, value>* and express the computational resources (e.g. memory, CPU, or energy) consumed by the query execution. Figure 9 shows an example.

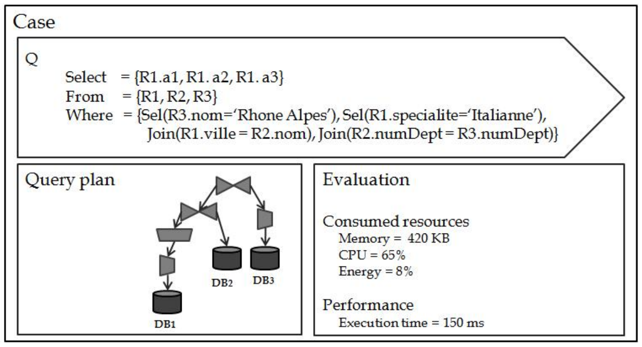
Case
Q
Select = {R1.a1, R1. a2, R1. a3}
From = {R1, R2, R3}
Where = {Sel(R3.nom='Rhone Alpes'), Sel(R1.specialite='Italianne'), Join(R1.ville = R2.nom), Join(R2.numDept = R3.numDept)}
Query plan
DB1
DB2
DB3
Evaluation
Consumed resources
Memory = 420 KB
CPU = 65%
Energy = 8%
Performance
Execution time = 150 ms

Figure 9. An example of a case

Figure 9 presents a simple instance of this model. Such instance solves the query Q by means of the query problem which is an ordered and pertinent sequence of selection, projection, sort, and join operations for accessing a set of data sources. The set of touples representing the resources consumed during the query evaluation applying the proposed query plan are described as follows: *Evaluation* = $\{$*<memory, 420>, <CPU, 65>, <energy, 8>, <execution time, 150>*$\}$.

4) Reasoning process: The reasoning process that must be accomplished given a new problem is an adaptation of the four-step case-based reasoning process. The recovering step is based on a similarity function in order to perform a smart search retrieving the most relevant cases to solve the query of the problem.

Among these relevant cases, the one that minimizes the cost function of the problem is selected. The reusing step is related to the adaptation process of the execution plan included in the relevant case to the new query. The reviewing step consists in verifying the query by its execution, where resource consumption measures are done. Finally, at the retaining step, the problem and its solution are stored in the case base in form of a new case [9].

1) *Retrieval*. Relevant cases are determined by a similarity function applied directly over the queries, in order to know which queries are similar and useful to solve the new problem (query, optimization objective and context measures). Given a problem, those cases containing a similar already solved query must be retrieved. If there is no relevant case in the case base, a new query plan must be (pseudo-)randomly generated to be able to increase the query optimizer knowledge.
2) *Reuse*. It is possible to reuse the retrieved solutions by means of their adaptation to the new situation. This

adaptation process consist in executeing the pertinent changes to the query plan included in the retrieved case (relevant) to satisfy the query problem requirements, e.g. modifying the projection attributes or the comparison values.

3) *Review*. Here the execution plan is verified. We analyze and adopt a specific model, the iterator model, in order to support the execution of each operation involved in a query according to the proposed algorithms. Iterators are responsible for applying specific algorithms implementing query operators and taking some resource consumption measures.
4) *Retention*. The query class, query plan and consumption measures are stored in form of a case within the case base.

V. SIMILARITY FUNCTION

Similarity notion is useful in different steps of the case-based reasoning process. At the retrieving step, a relevant case is identified by applying a similarity function. Furthermore, at adaptation step, the matching process depends on how much similar is the relevant case to the query problem. Finally, at the retaining step, a case is stored in the base of cases according to a defined classification. It is possible to know to which class pertains a case determining the similarity between the class of the query problem and the class in the case base.

We propose a similarity function that is performed in two steps. We define the membership of a query as a first step achieved by an inter-class similarity function. Then, the most relevant case within the class must be retrieved by means of an intra-class similarity function [10][11]. When the most relevant case is retrieved, a detailed comparison between the clauses of the new query and the relevant query (the query included by the relevant case) is carried out. This determines a similarity level between the two queries.

These functions are based on the contrast model of similarity proposed by Tversky [12] that allow us to determine the similarity between two objects by means of a feature-matching function. Similarity increases as most common features and decreases as most distinctive features [13]. The formalization of the original definition is expressed as follows [12]:

$$(5)\ S(a, b) = \theta f(A \cap B) - \alpha f(A - B) - \beta f(B - A)$$

Similarity between a and b, is defined in terms of the features common to a and b, $A \cap B$, the features that pertain to a but no to b, $A - B$, and those that pertain to b but no to a, $B - A$. The variables θ, α, and β are non-negative valued free parameters that determine the relative weight of these three components of similarity. Such variables provide the flexibility when modifying the importance of similarities or differences that in conjunction determine the similarity between two elements according to the area of application. The function f measures the salience of a particular set of features (also can be a single feature)[12].

A. Inter-class similarity

Adapting this model to our work, inter-class similarity is defined as an increasing function of common operation families and as a decreasing function of distinctive families, in other words, families that pertain to one query but not the other. The function can be applied to both classes, each one defined by a set of operation families, or applied to a query and a class. In this case, it is necessary to determine the operation families related to the involved operations. The formalization of this definition in terms of the similarity between a query and a class is expressed as follows:

$$(6)\ S(C_1,Q) = \theta\Im(C_1 \cap Q) - \alpha\Im(C_1\text{-}Q) - \beta\Im(Q\text{-}C_1)$$

Similarity between C_1 and Q, is defined in terms of operation families common to C_1 and Q, $C_1 \cap Q$, the features that pertain to C_1 but no to Q, C_1 - Q, and those that pertain to Q but no to C_1, Q - C_1. The function f refers particularly to operation families $\Im$. According to the purpose of our work, these are the features that must be compared.

For practical purposes, suppose that we know the class of the query q and the definition of the classes c_1 and c_2.

$$\text{q} = \{o_1, o_2, o_3\}\ \text{c} = \{\Im_{R.a1}, \in \Im_{R.a2}, \Im_{R.a3,R.a4}\}$$
$$c_1 = \{\Im_{R.a1}, \Im_{R.a2}, \Im_{R.a3,R.a4}\}$$
$$c_2 = \{\Im_{R.a1}, \Im_{R.a2}, \Im_{x}\}$$

From the intersections between the query class c that describes the query q and the classes c_1 and c_2, it is possible to state that the query class c is similar to c_1.

$$S(c_1,q) = \Im(C_1 \cap Q) = \{\Im_{R.a1}, \Im_{R.a2}, \Im_{R.a3,R.a4}\}$$
$$S(c_2,q) = \Im(C_2 \cap Q) = \{\Im_{R.a1}, \Im_{R.a2}\}$$

B. Intra-class similarity

Intra-class similarity function aims to find the most similar queries with respect to a new query, which is desired to be optimized, within the same class. In this step, all the compared queries are defined exactly by the same operations (operation type and involved attributes), the difference is related to the comparison operators, as well as the attribute values. Similarity between two queries Q_1 and Q_2 is defined as an increasing function of common operations (identical operations in terms of its type, attributes and operators). The formalization of this definition is as follows:

$$(7)\ S(Q_1, Q_2) = \theta o(Q_1 \cap Q_2) - \alpha o(Q_1 - Q_2) - \beta o(Q_1 - Q_2)$$

Similarity between Q_1 and Q_2 is defined in terms of the operations that are common to Q_1 and the features that pertain to Q_1 but no to Q_2. It is possible to establish a mapping one to one for each of the operations included in Q_1 and Q_3 according to the comparison operator that each of them apply. Q_1 and Q_2 have two operations in common; they differ in the operator applied by the join operation. According to this simple analysis, Q_3 is the most similar query with respect to Q_1 because it contains the maximum number of operation mappings.

For practical purposes, suppose that q_1, q_2, q_3, q_4 and q_5 pertain to the same class. The definition of each query, as well as its operations, is presented in the following example.

o_1 = Equal(a_1,value), o_2 = Equal(a_2,value),
o_3 = Equal(a_3,a_4), o_4 = Lower(a_3, a_4),
o_5 = LowerOrEqual(a_3, a_4) and o_6 = Different(a_2,value)

$q_1 = \{o_1, o_2, o_3\}$
$q_2 = \{o_1, o_2, o_4\}$
$q_3 = \{o_1, o_3, o_2\}$
$q_4 = \{o_1, o_2, o_5\}$
$q_5 = \{o_1, o_6, o_4\}$

It is possible to establish a mapping one to one for each of the operations included in q_1 and q_3 according to the comparison operator that each of them apply. q_1 and q_2 have just two operations in common, they differ in the operator applied by the join operation. Also, q_1 and q_2 have two operations in common, they differ in the operator applied by the join operation. Finally, q_1 and q_5 have only one operation in common. According to this analysis, q_2 is the most similar query with respect to q_1 because contains the maximum number of operation mappings. q_5 is the most different query respect to q_1 because it contains the minimum number of operation mappings. On the other hand, q_1 has exactly the same number of mappings with q_3 and q_4. How can we know which of these two queries is the most similar to q_1.

C. Similarity levels

The similarity level between two queries indicates which clauses of the relevant query must be adapted. This adaptation can be performed just over Select and Where clauses. For the Select clause, interesting attributes to be projected can vary. For the Where clause, comparison operators or some values related to the variables can be modified. On the other hand, the From clause can not be changed because the tables to be queried can not be changed. Table I illustrates the diverse similarity levels. Here, *selectClause* is expresed as *SC*, *fromClause* as *FC* and *whereClause* as *WC*.

The adaptation must be performed for the similarity levels (3), (2) and (1). If the similarity level is (3) the From and Where query clauses are equal, the adaptation must be performed on the select clause, which means that the projected attributes must be actualized according to the new requirements. If the similarity level is (2) the From and select query clauses are equal; therefore, the adaptation must be performed on the Where clause. In a similar way, some operation conditions are different but applied over the same attributes and tables- For this case, the condition type (e.g. equal, different and greater) and/or the constant value, to which the attribute is compared must be adjusted. Finally, if the similarity level is (1) an adaptation of the Select and Where clauses must be done.

Sim. Level	Equivalent clauses	Different clauses
4	SC, FC and WC	—
3	FC and WC	SC
2	SC and FC	WC
1	FC	SC and WC
0	—	SC, FC and WC

Table I
SIMILARITY LEVELS BETWEEN TWO QUERIES

VI. QUERY OPTIMIZER ARCHITECTURE

In order to solve a query, we propose a query optimizer that provides two alternatives. The first one reutilizes the solutions related to queries that have been solved; the second one generates new solutions. The optimizer is composed by two main modules, the case-reasoner and the execution plan generator. The case-base reasoner is in charge of adapting the solutions of similar queries to the new situation. The execution plan generator is in charge of generating new query plans in a pseudo-aleatory way. The case-base reasoner is the most complex of the two modules but the smartest, on the other hand, the execution plan generator is simpler and probably faster; however it does not apply machine learning techniques. Figure 10 illustrates the optimizer architecture.

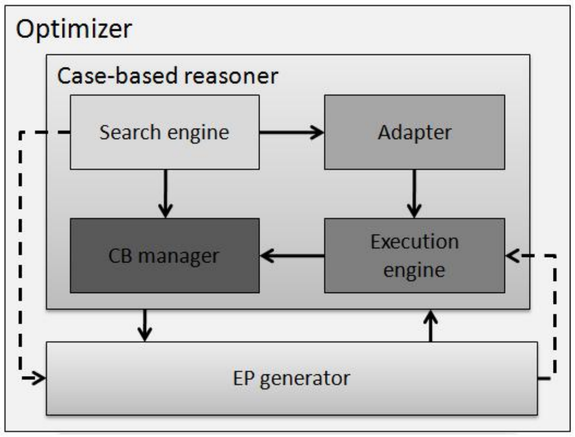

Figure 10. Optimizer architecture

A. *Case-based reasnoner*

The case-based reasoner implements the four case-base reasoning steps. It is composed by four submodules, one module for each reasoning step. Modules that comprise it are the smart search engine, the adapter, the execution engine and the case-base manager. This module interacts with a case base that allows managing the useful knowledge acquired from experiences (queries that have been solved).

1) Smart search engine: It accomplishes the first step of reasoning process retrieving relevant cases. This module performs a smart search by applying the similarity function (inter-class similarity and intra-class similarity) in order to recover those cases that contain the most similar query with respect to the query target to be optimized. Additionally, this module selects among those retrieved cases the one that minimizes the optimization parameters (e.g. energy consumption).

2) Adapter: It adapts the query plan included in the relevant case (retrieved in the previous step) to the query problem specifications. The relevant case is the most convenient (similar case) to facilitate and minimize the cost of the adaptation process; however, exist different similarity levels between the query related to the relevant case and the query problem.

3) Execution engine: It tests the new query execution plan created by the adaptation module. When executing operations, it implements the basic query operator: select, join, print and scan. Our implementation is based on the iterator model. Here, it is also needed to take some measures related to the performance of the new query plan. The stored measures are the execution time, the memory consumed, and the CPU charge.

4) Case base manager: Finally, this module carries out the last step in reasoning process; it allows to retain the new knowledge in form of a case. To store a new case we determine if it pertains to some existing class or if a new one must be constructed. The similarity function is also here because it determines the similarity between two classes. To perform this comparison, the query class of the new query is calculated (the operations families vector) and compared to each of the classes of the case-base. If it is similar to one of these classes, it must be stored as an instance of the class. Otherwise the new class must be added to the case-base and then, the case must be stored as an instance of the new class.

B. *Execution plan generator*

It might be possible that there is not any (or not enough) relevant case to solve a specific problem; therefore a totally new query execution plan must be built. We propose a pseudo-random mechanism for the generation of execution plans that consists of a single phase. In contrast to typical optimization strategies, this mechanism does not include the construction of algebraic trees; it constructs directly trees composed by physical operators. We implement these operators according to the iterator model. An execution plan is reasonable if it does not include redundant and useless operations that represent a waste of computational resources.

The construction of reasonable execution plans is based on a set of rules that define the physical operator (algorithm based on the iterator model) pertinent for applying the execution of each operation. The set of rules implements classical algebraic heuristics (selections and projections first) and identifies reasonable choices that have to be randomly done (e.g. join execution order or reasonable algorithms). These rules are applied in each level of the execution plan every time that an operator is added to the tree.

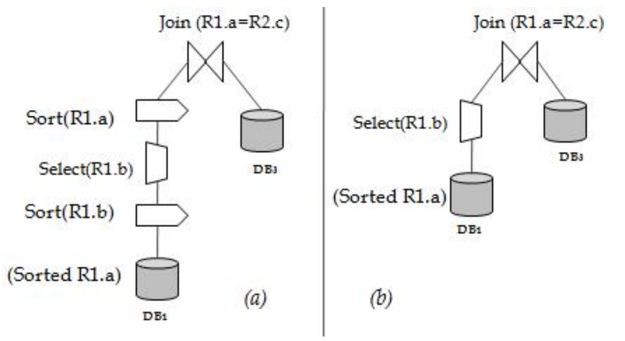

Figure 11. Reasonable (a) and unreasonable (b) execution plans

An execution plan is reasonable if it does not include redundant and useless operations that represent a waste of computational resources. Figure 11.a illustrates an unreasonable execution plan. Figure 11.b illustrates a reasonable execution plan.

The input data of the plan (a) is a sorted relation R according to the attribute a, as a first step, the relation is sorted according to the attribute b in order to improve the execution of the next selection operation. Then, the result is sorted again according to attribute a, finally a join operation is carried out with another relation S.

Plan (b) is the most simple, first step performs the selection and then the join operation that take advantage of the ordering of the relation R according to attribute a. It is evident that the first plan consumes more computational resources than the second one, i.e. disk space, memory and cpu for each sort operation. An execution plan must be selected in an aleatory way. The selection is performed over a set of reasonable execution plans; this is why the mechanism that we propose is not totally aleatory.

VII. Related work

Related work use machine learning techniques in order to address the query optimization problem. However, they are still based on metadata. Moreover, they perform some special treatment over metadata in order to improve the optimization calculus, e.g. the LEarning Optimizer LEO, that allows to repair incorrect statistics and cardinality, also estimates a query execution plan[14].

Optimizers based on plan caching i.e. SQL Server also use machine learning techniques. Particularly, SQL Server stores execution plans and data buffers. Nevertheless, it extends the general query optimizer architecture and is based on classical query optimization techniques, e.g. it is still narrowly tied up to the use of metadata[15].

VIII. First experiments

This section presents the first experimental results of our prototype. The objective is twofold: (i) observe the behavior of the case-based reasoning optimizer and (ii) study the variations on evaluation time and memory consumption when the case-base is preloaded with different amounts of knowledge. This knowledge is a threshold under which execution plans are systematically generated instead of extracted from the case base. The experiment that we propose consists in executing a set of queries related to five different query classes and registrating the measures related to the computational resources that were consumed during their execution.

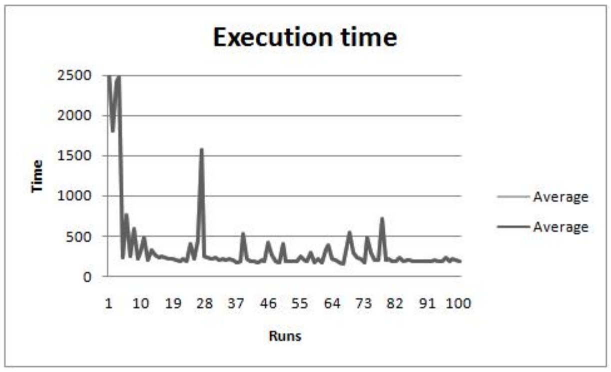

Figure 12. Evaluation time results (a) threshold = 10

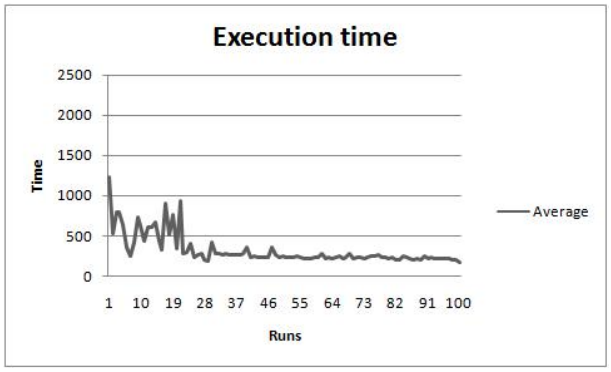

Figure 13. Evaluation time results (b) threshold = 20

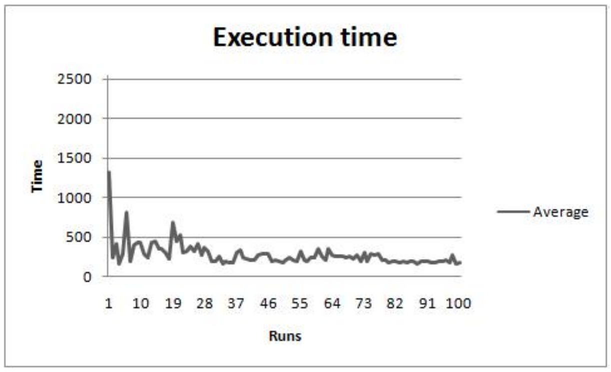

Figure 14. Evaluation time results (c) threshold = 30

Each figure is composed of 100 rounds of query evaluations where queries are taken from five different query classes. The executed query is randomly selected. The X axis represents each round. The Y axis represent the consumed ocmputational resources.

Figures 12, 13 and 14 show the query evaluation time w.r.t. different thresholds. Here, the optimization objective is minimizing the total evaluation time. Something we observe is that this evolution is decomposed in two phases: (i) a learning phase where evaluation time est very cahotic; (ii) a stabilized phase where the query optimizer reuse known query plans and updates the execution measures.

We can also observe that low values of threshold lead to poor performance stability and longer learning phase (Figure 12 while highest values reduce learning phase but do not improve stability (comparing threshold of 20 and 30 on figures 13 and 14). The gain in evaluation time is a factor from two to five.

Figures 15, 16, and 17 show memory consumption measures under the same experimental conditions, except optimization objective that is minimizing memory consumption. In a similar way to the first series of experiments, we observe that the memory consumption is divided in two phases: chaotic initial learning phase, and stabilized exploitation phase. One major point is that with a vert low threshold value, the gained optimization is only a two factor (figure 15) while we obtain a factor four with greater values (figures 16 and 17). Once again a threshold of 30 is not clearly better than a threshold of 20.

This graphic shows the average of the computational resources consumed during the experiments. It reflex the reduction of the consumed computational resources as more knowledge within in the case base. This experimental evaluation will be improved by dynamically varying other parameters, such as number of query families, database volume and evaluation rounds.

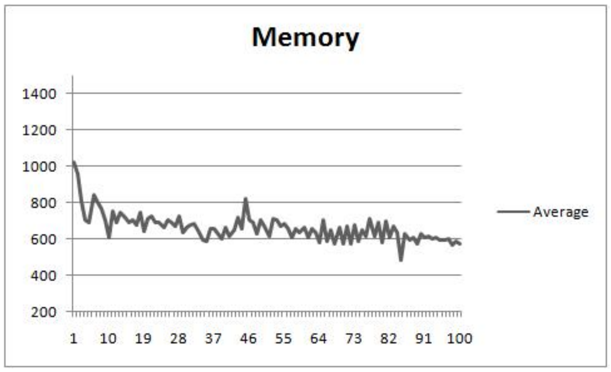

Figure 15. Memory consumption results (a) threshold = 10

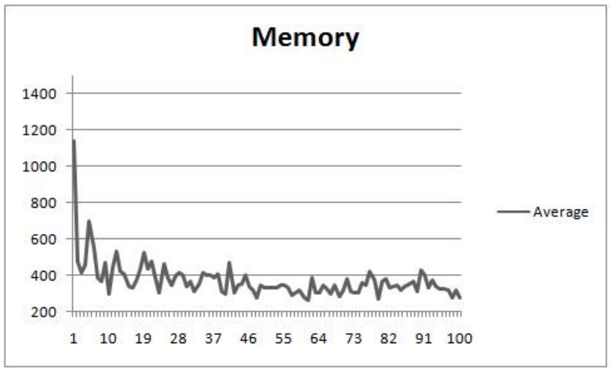

Figure 16. Memory consumption results (b) threshold = 20

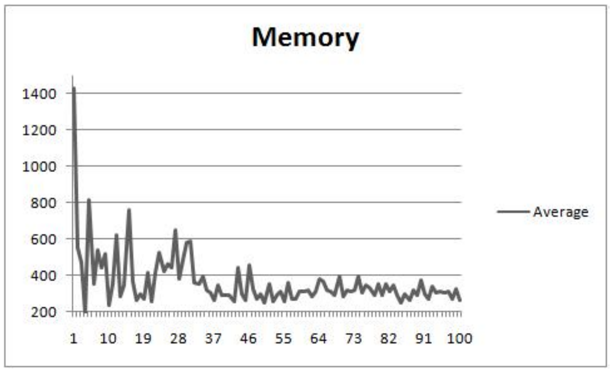

Figure 17. Memory consumption results (c) threshold = 30

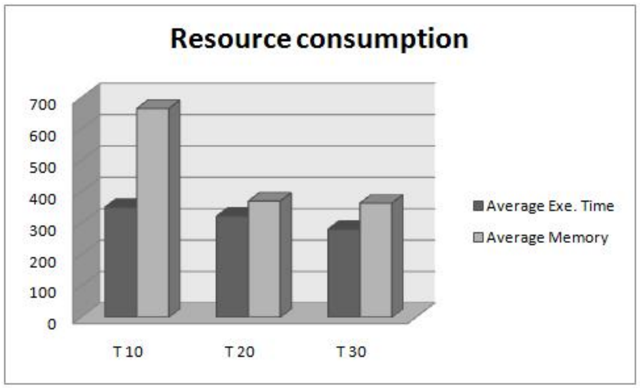

Figure 18. Resource consumption results

IX. Conclusions and future work

We propose a new query optimization technique that exploits case-based reasoning in order to improve query optimization in ubiquitous environments. Our approach deals with the challenge that the lack of metadata implies in this execution context. We propose a technique that is based on the useful knowledge (resource consumption measures) obtained from previous query executions. In addition, we propose a technique that allows the configuration of the optimization objective according to the users and application requirements, even for each single query.

The most important contributions of our work are centered in the reasoning steps related to retrieval and re-adaptation of the useful knowledge. These steps retrieve the most relevant cases and adapt a previous solution to the new situation. We propose a model to represent a query, a problem and a case. Moreover we define a similarity function to determine which query, related to cases stored in the case base, is similar and useful to optimize a new one, which is the problem to be solved. The adaptation depends on the similarity level that is determined between the two queries. Basically, the adaptation process consists in evaluating the operations of the query with variations in their conditions. Finally, we propose a pseudo-random query plan generator able to evaluate queries even if there is no corresponding case in the case base.

Currently we are working in a prototype that implements our solution in order to perform an experimental evaluation of our approach. It is very interesting to apply this solution to different application domains where possible solutions can be enumerated, performance criteria are known, but where important information is not available when deciding the best solution (statistics in our case). For example, automatic service composition generated on demand via a declarative language seems a good target. Dynamicity management is also an important point to work on. However, the system should be able to detect those cases in its case base no longer relevant and thus delete them. This is a problem of knowledge refreshment. We are also working on improving the knowledge acquisition and exploitation, for example by sharing case bases between nodes in a network or by reducing the granularity of cases to handle sub queries instead of full queries.

References

[1] E. Bardram, “Activity-based computing: support for mobility and collaboration in ubiquitous computing,” *Personal Ubiq-*

uitous Comput., vol. 9, no. 5, pp. 312–322, 2005.

[2] F. Perich, S. Avancha, and et al, “Profile driven data management for pervasive environments,” in *In: Proceedings of the 13th International Conference on Database and Expert Systems Applications (DEXA 2002)*. London, UK: Springer-Verlag, 2002, pp. 361–370.

[3] M. Franklin, “Challenges in ubiquitous data management,” in *Informatics - 10 Years Back. 10 Years Ahead.*, R. Wilhelm, Ed. Springer-Verlag, 2001, pp. 24–33.

[4] D. Subramanian and K. Subramanian, “Query optimization in multidatabase systems,” *Distrib. Parallel Databases*, vol. 6, no. 2, pp. 183–210, 1998.

[5] G. A. A. Deyand, P. Brown, and et al, “Towards a better understanding of context and context-awareness,” in *In: Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing (HUC 1999)*, September 1999.

[6] Y. Ioannidis, “Query optimization,” *ACM Comput. Surv.*, vol. 28, no. 1, pp. 121–123, 1996.

[7] A. Aamodt and E. Plaza, “Case-based reasoning: Foundational issues, methodological variations, and system approaches,” *AI Communications*, vol. 7, no. 1, pp. 39–59, 1994.

[8] M. Othman-Abdallah, “Optimisation de requtes par apprentissage,” Ph.D. dissertation, Institut National Polytechnique de Grenoble.

[9] L. D. Mantaras, R. McSherry, and et al, “Retrieval, reuse, revision and retention in case-based reasoning,” *Knowl. Eng. Rev.*, vol. 20, no. 3, pp. 215–240, 2005.

[10] R. Bergmann and A. Stahl, “Similarity measures for object-oriented case representations,” in *In: Proceedings of the 4th European Workshop on Advances in Case-Based Reasoning B*, B. Smyth and P. Cunningham, Eds. Springer Verlag, 1998.

[11] M. Gu, X. Tong, and A. Aamodt, “Comparing similarity calculation methods in conversational cbr,” in *In: Proceedings of the 2005 IEEE International Conference on Information Reuse and Integration*, 2005, pp. 427–432.

[12] A. Tversky and I. Gati, “Studies of similarity,” 1978.

[13] “Features of similarity and category-based induction,” Edinbumtgh, Scotland, 1997.

[14] M. Stillger, G. Lohman, V. Markl, and M. Kandil, “Leo - db2’s learning optimizer,” in *In: Proceedings of the 27th International Conference on Very Large Data Bases (VLDB 2001)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 19–28.

[15] Microsoft, “Excution plan caching and reuse,” 2008, http://technet.microsoft.com/en-us/library/bb545450.aspx. [Online]. Available: http://technet.microsoft.com/en-us/library/ms181055.aspx