Chapter 3

Query optimization using case-based reasoning

This chapter presents a query optimization technique that adapts case-base reasoning in order to provide each time better execution plans to solve new queries. This strategy recovers, adapts or generates execution plans using the knowledge acquired from previous experiences to optimize and execute similar queries [24]. The rest of this chapter is organized as follows.

Section 3.1 introduces the general query optimization technique using case-based reasoning. Section 3.2 details the problem definition proposed in this work with particular attention in the query concept. Sections 3.3 to 3.6 present the processes that compose the query optimization technique (Retrieval, Adaptation, Evaluation, and Retention) according to the classical case-based reasoning approach. When this technique cannot be applied due to the absence of useful knowledge to solve a query (problem), other solutions must be provided, Section 3.7 presents a pseudo-random query plan generation technique in charge to accomplish this work. Finally, Section 3.8 concludes this chapter.

3.1 Overall process

The query optimization technique that we propose is an adaptation of the general case-based reasoning process. This technique aims to solve the problem that lack of metadata means for classical query optimization techniques. This technique is feasible to be applied in different execution environments that have in common the hardly and expensive acquisition and maintenance of metadata. Ubiquitous computing environments are some representative examples. Lack of metadata of ubiquitous environments is a consequence of its other characteristics [2] [13].

Since case and problem are the main units of knowledge in this learning approach, we select useful knowledge for query optimization in order to instantiate both concepts. According to our approach, a case represents the knowledge related to the experience gained from the optimization and evaluation of a query. A problem represents a new query, that we call *query problem*, which is submitted in some application pertaining to the ubiquitous environment and that must be optimized according to some particular objective (e.g. CPU, memory, execution time, etc). We provide a more detailed description of each of these concepts in sections 3.2 and 3.3 respectively.

The reasoning process that must be accomplished to optimize a *query problem* is the following. Given a *query problem* an adaptation to query optimization of the four-step case-based reasoning process must be carried out[21].

1. **Retrieval**. This step is based on a similarity function in order to perform a *smart search* to retrieve the most relevant cases to solve the *query problem*. Among these relevant cases, the one that minimizes the cost function of the problem is selected.

2. **Reuse**. Reusing step is related to the adaptation process of the execution plan involved by the case that resulted relevant to solve the new query.

3. **Review**. Reviewing step consist in verify the query by means of its execution. During this step measures about performance as well as computational resources consumption are taken.

4. **Retention**. Finally, in the retaining step, the problem and its solution are stored in the case base in form of a new case.

Since this approach is based in a try and learn principle, when a relevant case to solve a *query problem* is not founded in the case base, is necessary to propose a new solution. We propose a pseudo-randomly query plan generation strategy that is detailed in section 4.7. This general process is repeated each time that a new *query problem* is submitted. It is possible to propose a new solution each time that a query is submitted despite that it is already solved, this with the purpose of perform some comparisons between different query plans related to the same query. These comparisons allow to select each time a better solution until an stabilization is achieved. This stabilization is achieved when the measure related to the optimization objective is relatively constant due to after a pertinent number of attempts, it presents a minimal variation.

3.2 Problem

A problem is composed by a *query problem*, the specification of the *execution context* by means of a set of measures that express the availability of different computational resources, and finally, the *optimization objective* that can be a single resource. Also could be interesting optimize different resources (e.g. CPU, memory, execution time) together. Figure 3.1 presents a UML diagram that illustrates the problem model that we propose.

According to our problem definition, a *query problem* is the target that must be

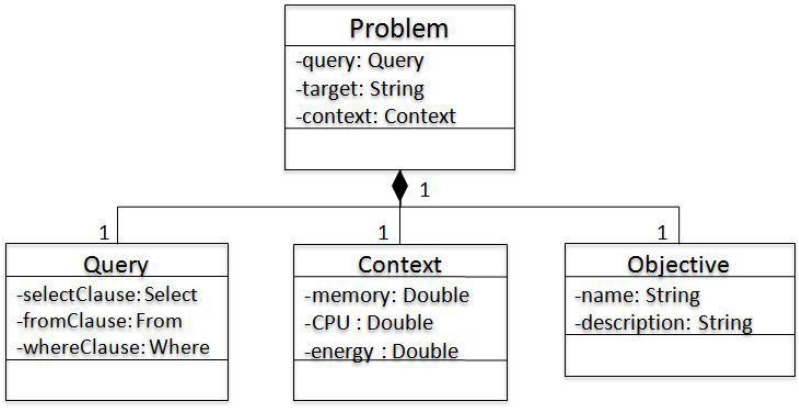

Figure 3.1: Problem model (UML diagram)

solved, of course as optimal as possible. The *execution context* express the characteristics of the execution environment at the moment that the query is evaluated. It is specified as a set of tuples of the form <*resource, value*>. Where *resource* makes reference to some computational resource that is required for computing the query and *value* is a measure that express the resource availability. These resources may include CPU charge, available memory, remaining energy, among others. Figure 3.2 presents a simple instance of this model.

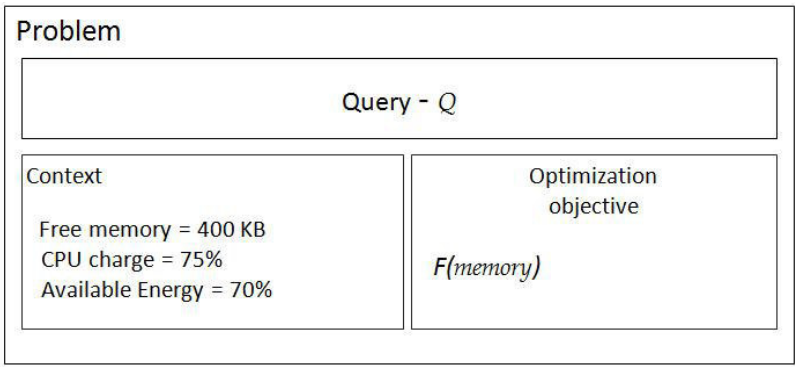

Figure 3.2: Instance of a problem

The set of tuples that represent the instance of context that Figure 3.2 illustrates

is as follows: *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)*.

3.2.1 Query

A query associated to a problem is defined by three clauses *Select* clause, *From* clause and *Where* clause. The *Select* clause specifies the attributes that must be projected as query results. The *From* clause specifies the *Data sources* that contain the information requested by the query. The *Where* clause is composed by a set of *Operations* that indicate the conditions that must be verified by the data to form part of the query result. Figure 3.3 illustrates the query model that we propose.

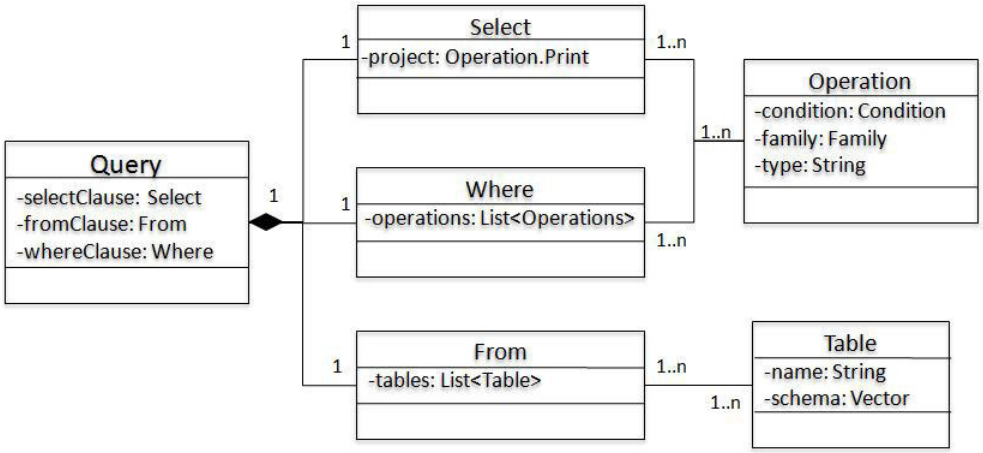

Figure 3.3: Query model (UML diagram)

According to our proposed solution, query operations are key elements for defining two functions that are essential for query optimization, the comparison between queries, as well as for its classification as we well see in Section 4.3. For this reason,

we provide a more detailed explanation about them. Each *Operation* is defined by a specific *Condition*, an operation *Type* and operation *Family*. Figure 3.4 illustrates the operation model that we propose.

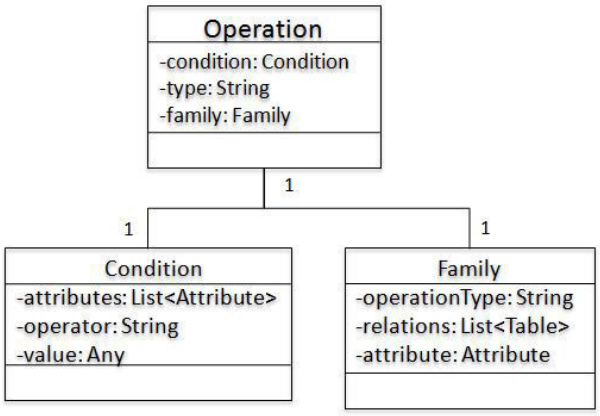

Figure 3.4: Operation model (UML diagram)

A *Condition* express a comparison, i.e. equal, different, lower, greater, etc. between a set of data values, that are associated to some attribute, and a specific constant value or another attribute. An attribute is defined by an identifier, a name and the data source to which it pertains. The form of the condition depends of the operation type to which it is associated. Selection and Join are the most important and most frequent operation types that are involved by the *Where* clause of a query. These operations are in charge to filter and combine the data in coherent and pertinent way. Selection operation is expressed by means of a condition of the form *Condition(satt1, sval1)*, where *satt1* represents the selection attribute and *sval1* a suitable constant value. Join operation is expressed by means of a condition of the form *condition(satt1, satt2)*, where *satt1* and *satt2* represent the join attributes.

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*). This means, select the

values that corresponds to attribute $R.a$ and, that are greater/greater or equal to some constant value. We propose the concept of operation family in order to grouping the operations with these characteristics. 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) $\Im_{R.an} = \{o_n \mid o_n = condition(R.a_n, value)\}$

The operation family $\Im_{R.an}$ is composed by the set of operations o_n, of the form *condition($R.a_n$, value)* that, according to the definition provided before, denotes an operation of Selection. In this condition, a_n is an attribute that pertains to the relation R. The *condition* by itself express all the possible comparison operators: Equal, EqualOrLower, Lower, GreaterOrEqual, Greater and Different. Any query is associated to a set of operation families, each family corresponds to one operation involved by the query.

In order to facilitate the comprehension of the definition of a query, as well as all the concepts that this definition involves, we provide the following example. Figure 3.5 shows a declarative query Q and the schema that describes the data sources that contain the requested information.

In our example, the query Q includes two selection operations o_1 = *EqualNomRegion,Rhone Alpes* and $_2$ = *Equalspecialite, Italienne*. It also includes two join operation $_3$ = *Equal(Restaurant.ville, Ville.nom)* and $_4$ = *Equal(Ville.numDept, Region.numDept)*. All these operations apply the comparison operator of equality *Equal*. The operations set of Q is expressed as follows $Q_{op} = \{o_1, o_2, o_3\}$. Table 3.6 presents the operations involved by Q. The operation families related to the operations in Q are expressed by some abbreviations corresponding to the pertinent data sources and attributes. The set of operation families that are associated to Q is expressed as follows $Q\Im = \{\Im_{R3.a8}, \Im$

Figure 3.5: Query example

$_{R1.a4}$, $\Im_{R1.a3,R2.a5}\}$ *and* $\Im_{R2.a6,R3.a7}\}$ (2).

Id	Type	Condition	Family
O_1	Selection	Equal(Region.nom = 'Rhone Alpes')	$\Im$(R3.a8)
O_2	Selection	Equal(Restaurant.specialite = 'Italianne')	$\Im$(R1.a4)
O_3	Join	Equal(Restaurant.ville = Ville.nom)	$\Im$ (R1.a3, R2.a5)
O_4	Join	Equal(Ville.numDept = Region.numDept)	$\Im$ (R2.a6, R3.a7)

Figure 3.6: Query operations

We can conclude that a query is the medullar part of knowledge for the definition of a problem, but also for a case. It is clear that the problem is not only the query by itself, as we seen before it also includes the circumstances under the query must be solved, this means, the measures related to the computational resources that are available for query execution. Finally, it also includes the optimization objective (e.g. memory, energy and CPU) which varies according to user requirements.

The query is the piece of knowledge that links a problem with the existent cases,

is to say, the cases that contain a query that is similar to the query included by the problem can be useful to solve the new query, but this is not all that is important for us. Also is necessary to put 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 change each time that the query is executed. We address these optimization aspects in the following sections.

3.3 Retrieval process

The retrieval process corresponds to the first step within the case-based reasoning process. This process is in charge of retrieve all the knowledge that could be useful to solve the *query problem* as optimal as possible. The cases that contain this knowledge are called relevant cases. The first step that must be accomplished for determining the relevant cases is the application of a similarity function that receives as parameters the *query problem* and the query of each case in the case base that is evaluated. This evaluation process aims to determine which queries are similar and useful to solve the new one.

However, this is not enough for determining the relevant case, since our solution allows the optimization according to different parameters, it is necessary taking in to account the measures about optimization resource (or set of resources). This knowledge is included in each case. Of course, it will be chosen the case that proposes a solution that consumes fewer resources than the others. The last condition that must be accomplished is that the available resources must be enough to compute the proposed query plan.

The relevant case will be that one that involves a similar query with respect to the query problem and that register the minimal consumption of the computational

resource that must be optimized. 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 as we explain in section 3.7.

3.3.1 Case

A case is composed by an already solved *query*, a proposed query plan to carry out the evaluation of that query, and finally, a set of evaluation measures about the different computational resources (e.g. CPU, memory, execution time) that were consumed during the query evaluation. Figure 3.7 presents an UML diagram that illustrates the case model that we propose.

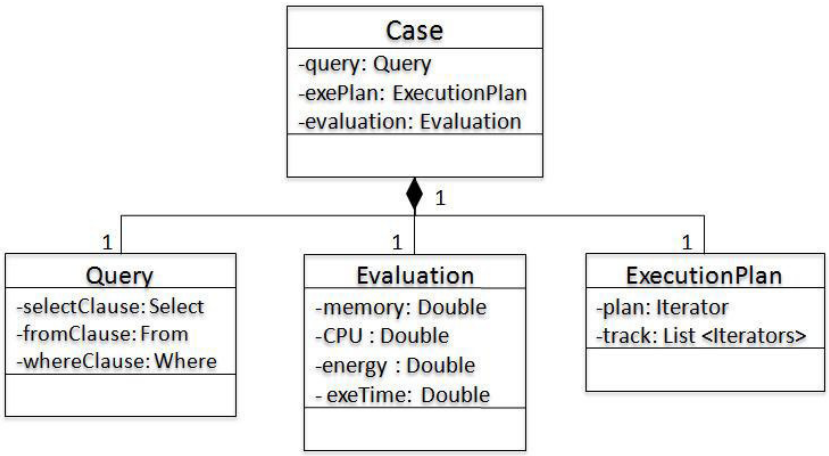

Figure 3.7: Case model (UML diagram)

According to our definition of a case, a *query* corresponds to an optimization target that has been evaluated and solved. The solution corresponds to the physical *query plan* that solves the query, typically, a tree where nodes represent algorithms (implementations of query operators) and links represent data flows among operators. In this case, the algorithms that the query tree involves are based in the iterator model that we well explain later in Section 4.5.

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. These resources may include CPU charge, available memory, remaining energy, among others. Figure 3.8 presents a simple instance of this model.

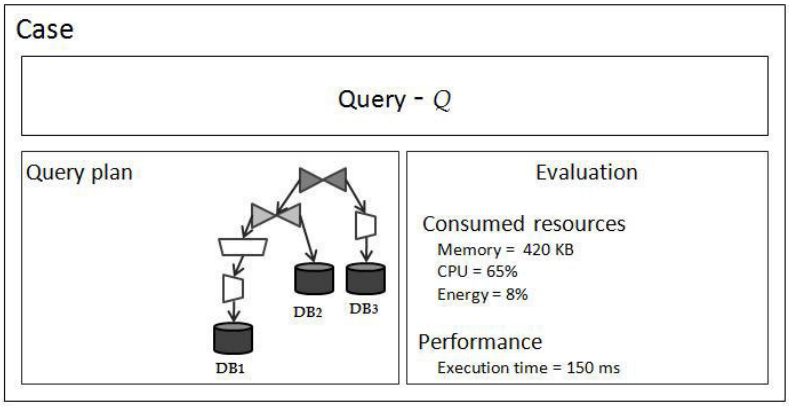

Figure 3.8: Instance of a case

The instance solves the query Q by means of the query problem that is an ordered and pertinent sequence of selection, projection, sort and join operations for accessing a set of data sources. The set of tuples that represent the resources that were consumed during the query evaluation applying the proposed query plan is described as follows: *Evaluation* = {<*memory, 420*>, <*CPU, 65*>, <*energy, 8*>, <*execution time, 150*>}.

3.3.2 Classification

Cases are classified within the case base; this classification is based on the characteristics involved by the query associated to the case. Queries must be classified according to the operation families to which pertain the operations that the query includes. Each different combination of $\Im_{R.an}$ conforms a class description, i.e. the class C_n that is

defined by the following operation families in (2).

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

The class C_n is composed by all queries Q_n that contains at least one operation that pertains to each of the specified families as is defined in (3). 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 on in Q_n such as this operation is of the form of the operation family $\Im$.

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

In order to clarify the previous equations we present an illustrative example. Given two queries Q_1 and Q_2, an evaluation of their Where clauses must be carried out since they contain the operations from which the query classification depends. Figure fig:classExample shows the operations that compose Q_1 and Q_2, as well as the set of operation families that describes the query class C.

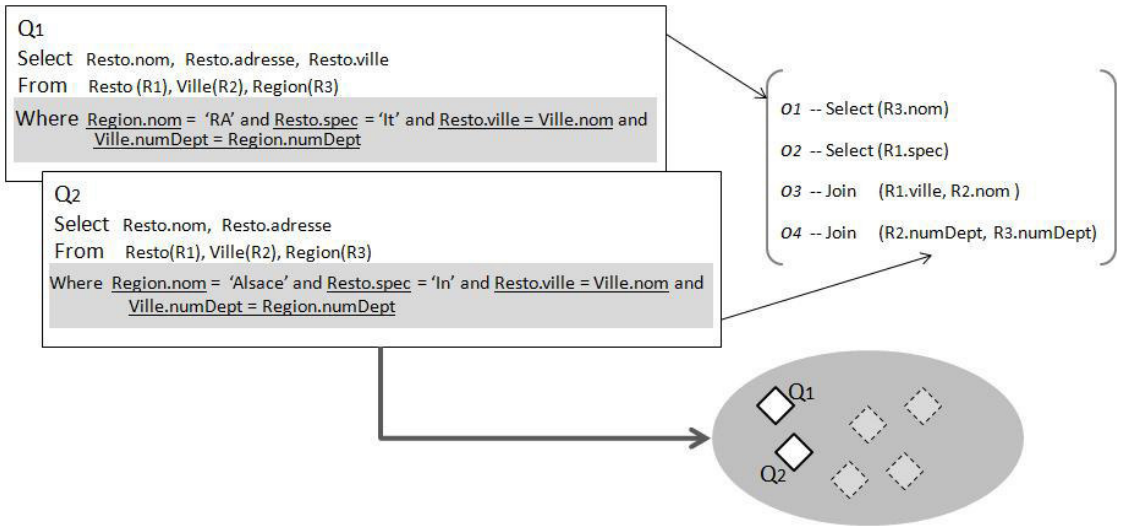

Figure 3.9: Query class example

Referring the query example Q_1, the selection operations *Region.nom=RA* and *Resto.spec=It* pertain to the operation families $\Im_{R3.nom}$ and $\Im_{R1.spec}$ respectively. The

join operation *Resto.ville = Ville.nom* pertains to the operation family $\Im_{R1.ville,R2.nom}$ and *Ville.numDept = Region.numDept* to $\Im_{R2.numDept,R3.numDept}$. On the other hand, Q_2 also has one operation that is possible to map with each operation family.

(a) $C = \{\Im_{R3.nom}, \Im_{R1.spec}, \Im_{R1.ville,R2.nom}$ *and* $\Im_{R2.nDept,R3.nDept}\}$

(b) $Q_1 \in C$ *iff* $(\forall \Im_{Rn.an} \in C) \exists ((o_n \in Q_1) \wedge (o_n \in \Im_{Rn.an}))$

It is important to remember that this is possible due to the operator and the attribute value are not factors to determine de operation family to which an 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 pertains to the families described before pertains to the same class (b).

3.3.3 Similarity

At retrieving step, the medullar part consists in perform a search within the case base in order to find the case that contains the best solution for solving a specific *query problem*. A case that contains a query equal to the submitted *query problem* is a good candidate. However, is not always possible to find cases with this characteristic, there are others that also contain useful knowledge to find feasible solutions. Cases which contain queries that involve the *query problem* (the query problem is subquery of the query in the case) and cases that contain a query that is involved by the *query problem* (the query in the case is subquery of the query problem) offer useful solutions. There are also cases that contain queries that share some conditions with the *query problem*, they could be useful too, but this depends of how many conditions they have in common.

We suggest that the solution provided by a case is more useful as more similar is the query that it involves with respect to *query problem*. We propose a similarity function that is performed in two steps. At the first step, we take advantage of the

organization that classification imposes within the case base, we define an inter-class similarity function for determining which is the more similar class with respect to the class associate to the *query problem* [7]. Then, at the second step, the most relevant case within the class must be retrieved by means of an intra-class similarity function.

These functions are based on the contrast model of similarity proposed by Tversky [30] 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 [1]. The formalization of the original definition is expressed as follows [30]:

(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, $B - A$.

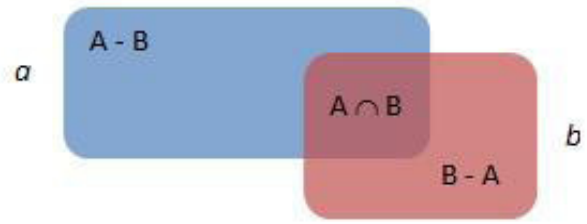

Figure 3.10: Similarity relation between two feature sets

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)[30]. Figure 3.10 illustrates a graphical similarity notion according to Tversky model.

Adapting this model to our work, intra-class similarity function aims to find the most similar queries classes within the case base with respect to the class associated to the *query problem* [15]. It is defined as an increasing function of common operation families and as a decreasing function of distinctive families (operation families that pertain to one query but not to the other). Figure 3.11 illustrates the inter-class search process in order to find the most similar class.

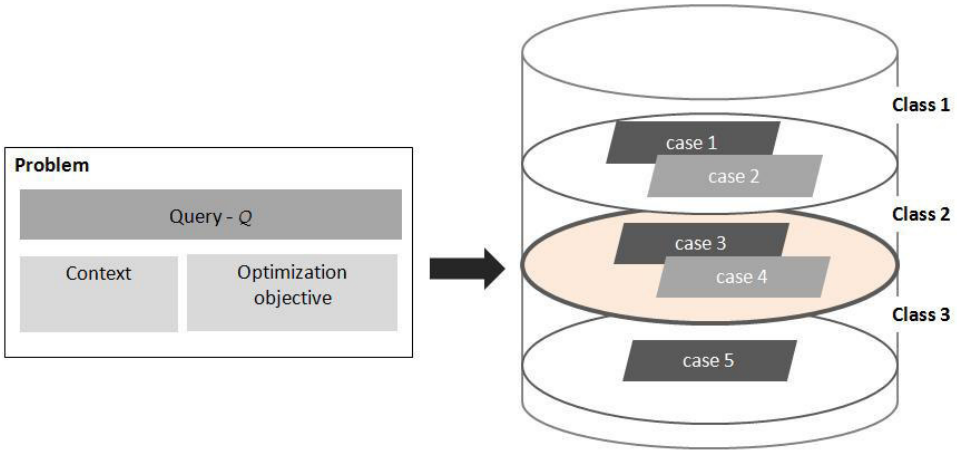

Figure 3.11: Inter-class similarity

The formalization of this definition in terms of the similarity between a query and a class is expressed in (6). According to this function, similarity between C_{Qm} and C_{Qn} is defined in terms of operation families that they have in common $C_{Qm} \cap C_{Qn}$, the operation families that pertain to C_{Qm} but no to C_{Qn}, C_{Qm} - C_{Qn}, and those that pertain to C_{Qn} but no to C_{Qm}, C_{Qn} - C_{Qm}. The function f refers particularly to operation families $\Im$.

(6) $S(C_{Qm}, C_{Qn}) = \theta\ \Im(C_{Qm} \cap C_{Qn})$ - $\alpha\ \Im(C_{Qm}$ - $C_{Qn})$ - $\beta\ \Im(C_{Qm}$ - $C_{Qn})$

On the assumption that C_{Qm} is the query class associated to the *query problem* and C_{Qn} is the query class associated to a query involved by a case. The query class C_{Qm} is equal to C_{Qn} if the operation families that define each class are exactly the same. The query class C_{Qn} involves the operation families that define C_{Qm} if their intersection contains all the operation families that pertain to C_{Qm} but not toC_{Qn} (this knowledge indicates that the query problem is subquery of the query involved by the case).

On the other hand, The query class C_{Qm} involves the operation families that define C_{Qn} if their intersection contains all the operation families that pertain to C_{Qn} but not toC_{Qm} (this knowledge indicates that the query involved by the case is subquery of the problem). Finally, C_{Qm} and C_{Qn} share just an operation families subset if their intersection is different from the empty set, but it does not contain all the operation families from any of them.

For practical purposes, suppose we know that the class of the query $Q = \{o_1, o_2, o_3\}$, corresponds to the query class $C_Q = \{\Im_{R.a1}, \Im_{R.a2}, \Im_{R.a3,R.a4}\}$ and the definition of the classes C_{Q1} and C_{Q2} to $C_{Q1} = \{\Im_{R.a1}, \Im_{R.a2}, \Im_{R.a3,R.a4}\}$ and $C_{Q2} = \{\Im_{R.a1}, \Im_{R.a2}, \Im_x\}$ respectively. From the intersections between the query class C_Q that describes the query Q and the classes C_{Q1} and C_{Q2}, it is possible to state that the query class C_{Q1} is the most similar to C_Q. In this example, the queries are equal. Equations (a) and (b) presents the intersection results of the query class C_Q associated to the *query problem* and the classes C_{Q1} and C_{Q2} respectively.

(a) $S(C_Q, C_{Q1}) = \Im(C_Q \cap C_{Q1}) = \{\Im_{R.a1}, \Im_{R.a2}, \Im_{R.a3,R.a4}\}$

(b) $S(C_Q, C_{Q2}) = \Im(C_Q \cap C_{Q2}) = \{\Im_{R.a1}, \Im_{R.a2}\}$

Intra-class similarity function aims to find the most similar queries (contained by the selected query class) with respect to the *query problem* [4]. It is defined as an increasing function of common operation and as a decreasing function of distinctive operations (operations that pertain to one query but not to the other). Figure 3.12

illustrates the intra-class search process in order to find the most useful case.

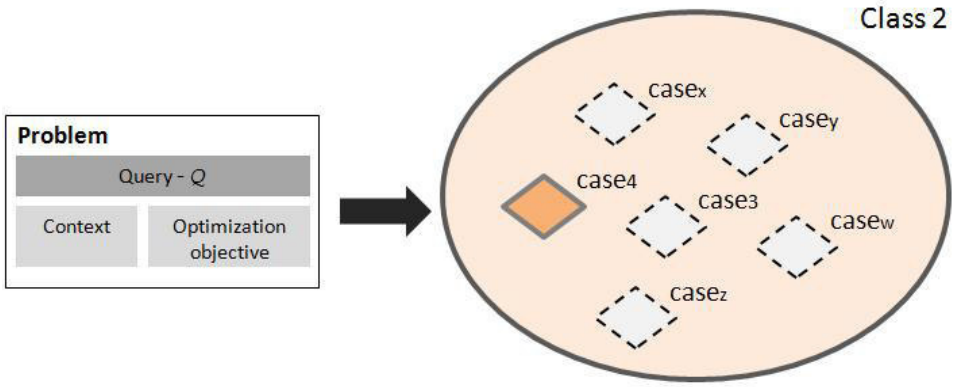

Figure 3.12: Intra-class similarity

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 in terms of the similarity between a query and a class is expressed in (7). According to this function, similarity between Q_m and Q_n is defined in terms of the operations that they have in common $Q_m \cap Q_n$, the operation that pertain to Q_m but no to Q_n, Q_m - Q_n, and those that pertain to Q_n but no to Q_m, Q_n - Q_n. In this case, function f refers particularly to the operations involved by each query.

$$(7)\ S\ (Q_m,\ Q_n) = \theta o(Q_m \cap Q_n) - \alpha o(Q_m - Q_n) - \beta o(Q_m - Q_n)$$

On the assumption that Q_m is the query that denotes the *query problem* and Q_n is the query involved by a case. The query Q_m is equal to Q_m if the operation included by each query are exactly the same. The query Q_n involves Q_m if their intersection contains all the operations that pertain to Q_m but not toQ_n (this knowledge indicates that the query problem is subquery of the query involved by the case).

On the other hand, The query Q_m involves the operation families that define Q_n if their intersection contains all the operations that pertain to Q_n but not to Q_m (this knowledge indicates that the query involved by the case is subquery of the problem). Finally, Q_m and Q_n share just an operations subset if their intersection is different from the empty set, but it does not contain all the operation from any of them.

For practical purposes, suppose that 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=$\{o_1, o_2, o_5\}$ and Q_5=$\{o_1, o_6, o_4\}$ pertain to the same class. The operations that compose the queries are the following: 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*).

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.

3.4 Adaptation process

The adaptation process corresponds to the reusing step in the case-based reasoning process. Once has been retrieved a relevant case for solving a new *query problem*, it is possible to reuse the solution (query plan) that it offers by means of an adaptation process that consists in execute the pertinent changes to the query plan in order to

satisfy the *query problem* requirements e.g. modifying the projection attributes or the comparison values. We propose a simple adaptation process that can be performed just over Select and Where clauses.

In despite of the relevant case offers the better solution to facilitate and minimize the cost of the adaptation process, when it is retrieved, a detailed comparison between the clauses that compose the *query problem* and the query included by the relevant case must be carried out. This comparison determines a similarity level between the two queries that indicates which clauses of the relevant query must be adapted.

3.4.1 Similarity level between queries

The similarity level between two queries indicates which clauses of the relevant query must be adapted. As we said before, this adaptation can be performed just over Select and Where clauses. The adaptation can include a single clause or some combinations.

The Select clause adaptation consists in a change of the attributes of interest that must be projected. The Where clause adaptation consist in a change of the comparison operator or some values related to the variables. On the other hand, the From clause can not be adapted because the data sources involved by the query can not be different. Table 3.13 presents in a simple way, the query clauses that must be adapted according the similarity levels that can be determined between the *query problem* and the query in the relevant case.

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, this means that the attributes to project must be actualized according to the new requirements. If the similarity level is (2) the from and select query clauses are equal, the adaptation must be performed on the where clause,

Similarity level	Equivalent clauses	Different clauses
4	Select, From and Where	-----
3	From and Where	Select
2	Select and From	Where
1	From	Select and Where
0	-----	Select, From and Where

Figure 3.13: Similarity levels

this means that some operation conditions are different but are applied over the same attributes and the same tables, in 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 made. When the level is equal to (4) or (0) it is not needed to perform an adaptation, in the first case because the execution plan is useful to solve the new query, in the second case because it is no possible to carry out an adaptation according to our approach.

As a conclusion, the similarity level reflect the necessary adaptations or adjustments that must be carried out and consequently, it allows us to determine how easy is to adapt an existent query solution to the new query specifications. This represents a direct relation between a problem and the solutions.

3.4.2 Select clause

The Select clause adaptation consists in modify the attributes that will be provided to the user as query results. In order to facilitate the understanding of this adaptation process we provide an example. Given the query problem Q and the relevant Q_1 in Figure 3.14, is nedded to analyze if the attributes that are included by the Select clause

of each of them are exactly the same. The set $Q_s = \{R_1.a_1,\ R_1.a_2,\ R_1.a_3\}$ denotes the projection attributes of Q. The set $Q_{1s} = \{R_1.a_1,\ R_1.a_2\}$ denotes the projection attributes of Q_1.

Query – Q

```
Select   Restaurant.nom, Restaurant.adresse,Restaurant.ville
From     Restaurant, Ville, Region
Where    Region.nom = 'Rhone Alpes' and Restaurant.specialite = 'Italianne' and
         Restaurant.ville = Ville.nom and Ville.numDept = Region.numDept
```

Query – $Q1$

```
Select   Restaurant.nom, Restaurant.adresse
From     Restaurant, Ville, Region
Where    Region.nom = 'Rhone Alpes' and Restaurant.specialite = 'Italianne' and
         Restaurant.ville = Ville.nom and Ville.numDept = Region.numDept
```

Figure 3.14: Select clause adaptation

An adaptation of the Select clause must be carried out if the set that results from the union of the differences between Q_s and Q_{1s} is different from the empty set. The result of this definition is expressed as follows $\{Q_s - Q_{1s}\} \cap \{Q_{1s} - Q_s\} = R_1.a_3$. In this case, we need to modify the set of attributes of Q_1 adding the attribute $R_1.a_3$ in order to satisfy Q.

3.4.3 Where clause

The Where clause adaptation is carry out in two steps. In the first step, a mapping between the operations of each query must be carried out. In the second step a modification of the constant values involved by some Selection operation is performed. In order to facilitate the understanding of this adaptation process we provide an example. Given the query problem Q and the relevant Q_2 in Figure 3.15, is necessary to perform a mapping between the operations involved by the Where clause of each query. If the

mapped operations compare the involved attribute with a different constant value, this must be change.

Figure 3.15: Where clause adaptation

One of the mapping between Q and Q_2 is expressed as follows $R_1.a_4$ = 'Italianne' $\rightarrow$ $R_1.a_4$ = 'Mexicaine'. The constant value for the operation condition in Q_2 must be changed for the value specified in Q.

3.5 Evaluation process

At this step, the query plan included by the relevant case must be evaluated. This means, the best execution plan to solve the *query problem* (according to some specific optimization objective) was determined and the computational resources that are available at the moment of the query execution are enough for evaluating the query plan. During these evaluations, a number is assigned to each of the steps within the plan, representing the computational resources consumed by the operation execution at this step. This shows what is called the evaluation cost for that step. The accumulation of costs for each step is the cost for the execution plan itself.

3.5.1 Query plan representation

A query plan is a set of steps used to access or modify the information in a data source. A query plan typically results from the attempt of the query optimizer for determining the most efficient way to solve the *query problem*. According to our solution, the execution plan involved by a relevant case expresses how a query was executed (or how a query will be executed, when they are created for the first time).

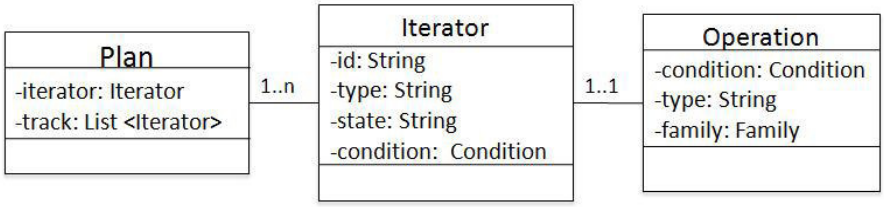

Figure 3.16: Query plan model (UML diagram)

Figure 3.16 presents an UML diagram that illustrates the query plan model. For the evaluation of the query plan we analyze and adopt a specific model, the *Iterator model*, that propose a set of algorithms for implementing each *operation* that the query plan involves. The query plan can be expressed in many different ways. To enable cross-platform portability, most of DBMS compiles queries into some kind of interpretable data structures e.g. trees.

The tree-structured query evaluation plan is used to execute queries by demand-driven dataflow. There exist some packages feature tools which will generate a graphical representation of a query plan, others allow a special mode to generate textual or XML descriptions. Figure 3.17 presents a graphical representation of a query plan that solves the query Q presented in subsection 4.5.1.

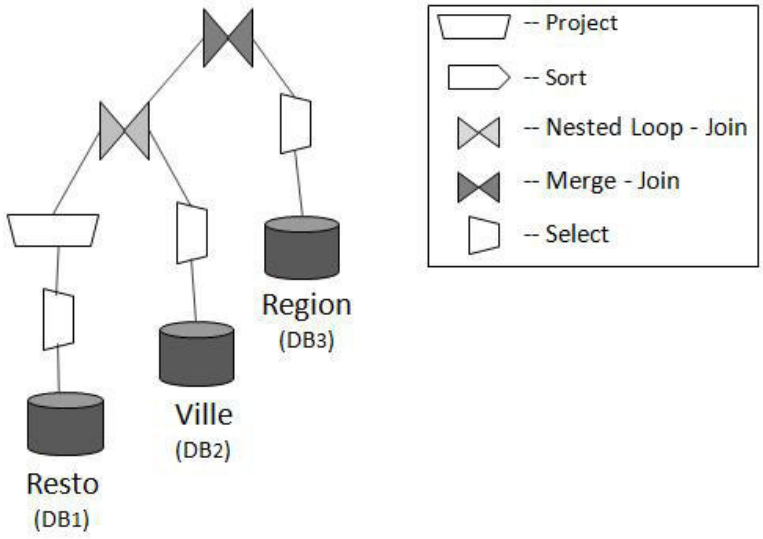

Figure 3.17: Query plan example for solving the query Q

3.5.2 Execution model

Multiple execution models have been proposed for implementing the operators involved by a query plan [14]. As we explain in the previous section, a query plan is typically represented as a tree which its nodes express operators that encapsulate base-table access query execution algorithms. Due to its characteristics, the execution model that we adopt for implementing these operators is the Iterator model.

The general idea is to specify an item, typically a single record, and to iterate over all items for generating an intermediate query result. When an operator requires another item, it throws a call to its input operator, which is responsible for producing one, likewise, that input operator might require an item from its own input to produce another one; in that case, it calls its own input [14]. A collection of algorithms are required for implementing each operator (e.g. Sort, Select and Join). Operators implemented in this model are called Iterators [14]. Figure 3.18 presents an UML diagram that illustrates the general iterator model.

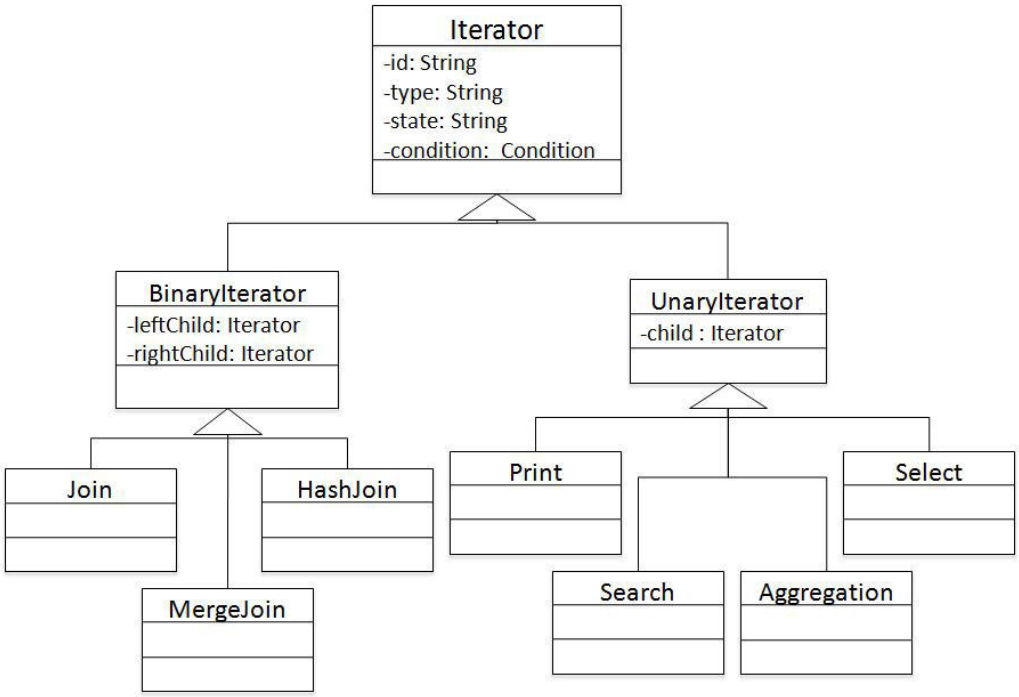

Figure 3.18: Iterator model (UML diagram)

According to the general model, an iterator is specified by an *Id*. They can be from different types, they are divided in *unary* and *binary* iterators, that at the same time, are divided in all the iterators that implement the operations in a query tree (e.g. Select, Sort and Join), each of them involve a specific *condition* (e.g. comparison or combination). Associated with each iterator is a *state* record that keeps the arguments for their corresponding algorithms, e.g. comparisons and hashing are performed by support functions which are given in the state record.

In more detail, unary iterators are those that operate on a single data source e.g. sorting (Sort) and selection (Selection) algorithms, there exist others like different aggregation algorithms as well as the print (Print) algorithm in charge to project the attributes of interest for the user in the final result. Binary iterators are those that implement different algorithms to combine two data sources (e.g. Join, MergeJoin and

HashJoin).

Some important features of operators implemented in this way, and that we took into account to select this model for basing out work are:

1. **Single process computation.** They can interact (e.g. execute and schedule) with each other through simple procedure calls within a single process that are cheaper than inter-process communication because it does not involve interaction with the operating system [14, 18].

2. **Independent logic.** They can be combined into arbitrary query plans since their logic is independent of its parents and its children within the query tree [14]. This means, any operator can be the input of any other. On the other hand, the operator does not need to know anything about the operator that produces its input. Code for particular combinations of Iterators is not required [18, 16].

3. **Multiple structures processing.** Their operation functions can be applied on any data structure without knowing the details of how that data structure is navigated; e.g., the same function could process the nodes in a document, a document sub-tree, or a node list [26].

4. **Demand-driven pull model implementation.** They couple dataflow with control flow [14, 26]. Whenever a node requires a tuple, it invokes a next() function. Each child node returns the produced item to its parent node when it is called [19]. Hence a tuple is returned to a parent in the graph exactly when control is returned [26]. Because of this property, items never wait in a temporary file or buffer between operators because they are never produced before they are needed e.g. a join operator cannot return a tuple unless it gets a tuple both from the outer and the inner relation and the tuples satisfy the join predicate [19].

5. **Efficient execution.** Iterators are simple but powerful abstractions that allows combine any number of operators to evaluate a complex query. It represents the most efficient execution model in terms of time (overhead for synchronizing operators) and space (number of records that must reside in memory at any point of time) for single process query evaluation [16]. Implementing this execution model is simple.

3.5.3 Measures collection

There exist a wide variety of query optimization techniques, (i.e. semantic, parametric, etc.) but all of them presented in a framework of classical query optimization procedures heavily dependent of metadata e.g. statistics and cardinality estimates. There exist environments, such as ubiquitous environments, where metadata acquisition and maintenance is very expensive. In some cases no metadata is available.

One of the main problems addressed by this thesis project is the difficulty for obtaining metadata, since it is indispensable for classical query optimization techniques to estimate the cost of query plans. If it is not possible to count with this metadata, we propose the collection of another type of very useful knowledge for the learning process due to it reflects how good is an experience (a query execution) in terms of efficiency, this means, the minimum consumption of resources, specially, the optimization objective e.g. energy.

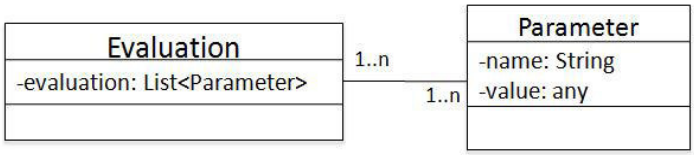

Figure 3.19: Evaluation model (UML diagram)

This knowledge is constituted by a set of measures of the computational resources that were consumed during a query execution. This measures must be taken each time that a query is executed, whit the objective of taking the first measures associated to a specific *query problem* (a query that is solved by the first time), or for actualizing the existent ones. Figure 3.19 presents an UML diagram that illustrates the model for this *evaluation*, that is expressed by means of a list of *parameters* of the form *¡name, value¿* e.g. ¡Energy, 8¿.

3.6 Retention process

In the retention step, the case that results from the new experience (the execution of the *query problem*) must be stored in the case base. The case base is not just a chaotic super set conformed for all the generated cases, it has a specific organization. The case base is organized in groups defined by the class concept. As we already explain in detail in previous section, the class of a query Q is determined by the families to which the operations included by Q pertain. The family to which an operation pertains depends on the operation type and the attributes that it involves (independently from its comparison operator and the comparison value). Many operations can pertain to the same family, and a class can be defined by different combination of families. Figure 3.20 presents an example of a case base.

When a new case must be inserted in the case base, the first task that must be carried out is determined the class associated to the query that it contains. The, a search within the case base must be performed in order to know if the class is already defined within the case base. If the class already exists, the case must be inserted in this group. In other case, the class must be defined and then, the case must be inserted in the new class.

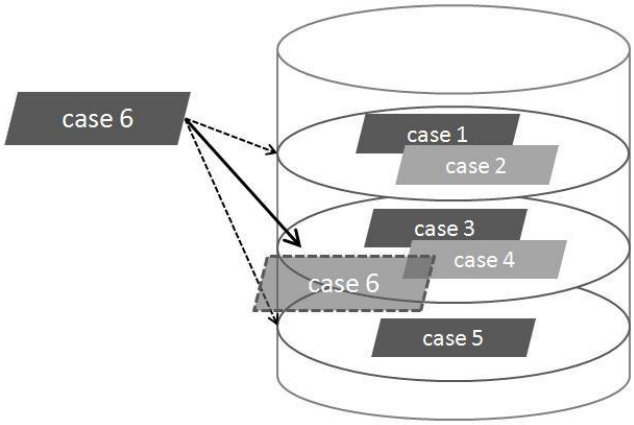

Figure 3.20: Retention of a case (case base example)

3.7 Pseudo-random query plan generation

The optimization strategy that we propose provides two alternatives for the generation of a query plan. The first one concerns to the recovery and adaptation of a query plan that was previously created in order to answer a query similar to the new one (optimization objective). These query plans are stored in a case base, where a case is compose by a query with its respective execution plan and some measures about its execution. The second one is detailed in this section, it concerns to the generation of a new query plan if the case base does not contains any query similar to the new one and; therefore, any useful query plan for adapting it to the new situation.

Typically, the core of the process for generating query plans is divided in two main phases. In the first phase, the order of query operators is defined and expressed by a tree structure; where left nodes correspond to relations, intermediate nodes correspond to query operations over the relations and finally, the root node represents the solution of the query. Each query operator can be executed in different ways; in the second phase of the process, an specific algorithm (or combinations) is associated to each operator for its execution.

We propose a pseudo-random mechanism for the generation of pertinent query plans that, in contrast with typical optimization strategies, consists of a single phase. This process does not include the construction of algebraic trees; it constructs, in a single step, trees structures composed by a set of physical operators. We base the implementation of these operators in the iterator model.

3.7.1 Pertinent query plans

A query plan is pertinent if it does not include redundant and useless operations that represent a waste of computational resources (memory, CPU, energy, etc.). Pertinent query plans define the physical operators (algorithms based on the Iterator model) that are convenient to apply for executing the operations involved by a query, as well as the order in which these operators must be applied. Figure 3.21.a illustrates a non-pertinent query plan. Figure 3.21.b illustrates a pertinent query plan.

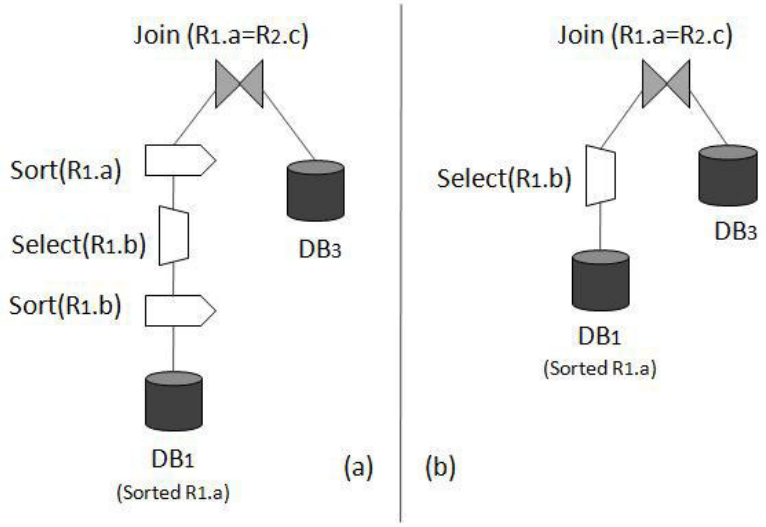

Figure 3.21: Pertinent (a) and not-pertinent (b) query plans

The input data of the query plan (a) is the relation R, that is ordered according

to the attribute *a*. The first step of the query plan shows that this relation is ordered according to the attribute *b* with the objective of improving the execution of the next operation, a selection operation. Then, the result from this selection is sorted again according to attribute *a*. Finally a Join operation that involves another relation S is carried out.

The query plan (b) is most simple, in the first step, it applies 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. The construction of pertinent query plans is based on a set of rules that indicate the physical operators (algorithm based on the iterator model) that are pertinent for executing the different operations. These rules are called pertinence rules.

3.7.2 Pertinence rules

Pertinence rules aim to avoid the execution of redundancy operations and the unnecessary waste of computational resources. An operation is redundant if the input and output data is the same before and after an operation is applied. Execute an operation is useless if it does not perform any change in the data from which another operation can take advantage. Pertinence rules can benefit (performance improvement) just some specific operations or all the operation in the query plan. Pertinence rules also specify the algorithms that can be associated to operations that must not to be benefited. Each rule is composed by a set of conditions that indicate the pertinence rules that can be applied in order to benefit or not an operation, and a set of actions that indicate the algorithm(s) that must be applied to execute the operation.

A pertinent rule is composed by a set of conditions that determine the actions that

can be applied in order to benefit or not an operation, these actions indicate the algorithm(s) that must be applied to execute the operation. The set of rules implements classical algebraic heuristics (selections and projections first) and identify 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, is to say, each time that an operators is added to the tree.

The conditions of a pertinent rule are determined by the operation type, current operation conditions, status of the input relation and future operations. The types of query operations that we implement are print, sort, selection and join (nested-loop and merge join) operations. The conditions of a pertinent rule are determined by the operation type, current operation conditions, status of the input relation and future operations. The types of query operations that we implement are print, sort, selection and join (nested-loop and merge join) operations.

The operations are applied over relations with a defined status, it includes the relation features that result from operations previously applied; particularly, sort operations in order to determine if the relation was ordered in terms of a specific attribute of if it lacks of order. If the relations are ordered according to the attributes included in the operation conditions, the performance of the operations is improved. Pertinence rules related to each situation are presented next.

Sort

A sort operation can be applied over a relation if and only if the relation was not previously ordered in terms of the attribute specified in the operation condition. This operation can be beneficiated only if the operation list related to the relation contains a selection operation that includes the attribute of sort in its condition. Sort a relation that has been filtered by a selection operation has some advantages, due to the decrease

in the amount of data. Is to say, given a sort operation sort(R.a), where a is the ordering attribute, it is viable to execute a selection operation select(R.a) over the same attribute.

As an example, the pertinence rule (3) indicates that it is required to carry out a sort operation with respect to the attribute a, over the relation R. This operation can be beneficed if and only if exist a selection operation that includes the ordering attribute. The selection must be executed previously to sort operation.

Select

A selection operation can be applied if and only if it was not previously executed. There are two forms to proceed when a selection operation must be carried out. The first one involves the performance of the selection without taking important considerations in order to favor the selection. The second one involves order the data previously to the selection to favor the operation. In bout cases the heuristic related to selection operators redistribution (heuristics that specify the cases in which are feasible to apply selection operations before a join) is respected.

If is required to favor a selection operation, the input data features must be considerate (know if input data is ordered according to some attribute or if it lacks of order). Also the operation conditions must be analyzed in order to carry out a convenient order, is to say, in accordance to the attribute of selection. Select data over an ordered relation can involve less effort than execute the same operation over a raw relation since it is possible to avoid the exhaustive analysis of all records. It is possible to stop when the fist value that does not accomplish the condition is read. In order to benefit the selection operation select(C), where C is a selection condition over the attribute a, a sort operation(R.a) must be previously executed.

As an example, the pertinence rule (3) indicates that it is required to carry out a selection operation with respect to the attribute an over the relation R. In contrast to

sort operation, the selection can be favored even if there is not any sort operation in terms of the selection attribute in the operations list that corresponds to R. In this case, the sort operation must be created (specify the operation condition) and applied previously to the selection.

Join

A join operation can be applied if and only if it was not previously executed. There are two forms to proceed when a join operation must be carried out. The first one involves perform the combination of data without taking important considerations in order to favor it. The second one involves order the data that correspond to relations (only one or bout) previously to the selection to favor the operation. In bout cases the heuristic related to selection operators redistribution (heuristics that specify the cases in which are feasible to apply selection operations before a join) is respected. Combine data after a selection operation involves less effort since the amount of data that is combined decrease.

Nested-loop join and merge join algorithms can be executed in order to combine data of relations. The first algorithms can be used indistinctly; nevertheless, if is accomplished that attribute a, in accordance to which the relation is ordered, is included in the join condition R.a , the performance of the algorithm is improved for the reasons that have been explained above. If any relation or only one is conveniently ordered the pertinent operation is nested-loop-join(R.a', S). On the other hand, the second algorithm can be applied if and only if the attributes in accordance to which bout relations are ordered are included in the join condition, in this case the pertinent operation is merge-join(R.a', S), where must be accomplished that a = a'.

As an example, the pertinence rule (4) indicates that it is required to carry out a join operation to combine the data of the unordered relations R and S. In the same way

than selection operation, the join can be favored even if there is not any sort operation in terms of the selection attribute in the operations lists that correspond to R and S. In this case, the sort operation must be created (specify the operation condition) and applied previously to one of the relations. Then, the join operation can be performed and favored. In rule (7), the order is performed in the two relations; in this case, a merge join can be applied.

3.8 Conclusions

This chapter presented a query optimization strategy to generate solutions for data retrieving in ubiquitous computing environments. The proposed solutions is based on machine learning approaches, particulary, case-based reasoning in order to take advantage of knowledge acquired from previous experiences. We define a model for knowledge representation, a similarity function to identify the useful knowledge to solve a problem. Also, a method to retrieve, adapt, manage and share the knowledge. And finally, a method for the generation of new knowledge.