D2I

**Integrazione, Warehousing e Mining di sorgenti eterogenee**

*Programma di ricerca (cofinanziato dal MURST, esercizio 2000)*

# Index selection techniques in data warehouse systems

M. GOLFARELLI, S. RIZZI, E. SALTARELLI

**D2.R5** **4 febbraio 2002**

**Sommario**

In the area of decision support systems, a basic role is held by data warehouses. Though designing a data warehouse requires different techniques from those experienced in operational databases, it is still possible to decompose the design process into three distinct phases: conceptual design, logical design and physical design. This paper addresses a basic issue in physical design by proposing a heuristic approach which selects an optimal index set to be built in a data warehouse implemented on a relational DBMS. To achieve this goal we simulate an optimizer which generates query execution plans and define a cost model to evaluate them. The indexes considered belong to two very common categories: tid-list indexes and bitmap indexes. Finally, we outline a greedy algorithm which chooses, from a set of candidate indexes, the most promising ones respecting a constraint on the disk space devoted to indexing.

| | |
|---|---|
| **Tema** | Tema 2: Progettazione e Interrogazione di Data Warehouse |
| **Codice** | D2.R5 |
| **Data** | 4 febbraio 2002 |
| **Tipo di prodotto** | Rapporto tecnico |
| **Numero di pagine** | 19 |
| **Unità responsabile** | BO |
| **Unità coinvolte** | BO |
| **Autore da contattare** | Ettore Saltarelli<br>DEIS - Università di Bologna<br>Viale Risorgimento 2, 40136 Bologna, Italia<br>esaltarelli@deis.unibo.it |

# Index selection techniques in data warehouse systems

M. Golfarelli, S. Rizzi, E. Saltarelli

4 febbraio 2002

**Abstract**

In the area of decision support systems, a basic role is held by data warehouses. Though designing a data warehouse requires different techniques from those experienced in operational databases, it is still possible to decompose the design process into three distinct phases: conceptual design, logical design and physical design. This paper addresses a basic issue in physical design by proposing a heuristic approach which selects an optimal index set to be built in a data warehouse implemented on a relational DBMS. To achieve this goal we simulate an optimizer which generates query execution plans and define a cost model to evaluate them. The indexes considered belong to two very common categories: tid-list indexes and bitmap indexes. Finally, we outline a greedy algorithm which chooses, from a set of candidate indexes, the most promising ones respecting a constraint on the disk space devoted to indexing.

## 1 Introduction

The huge data flow that daily invests companies makes automatic tools crucial to manage information efficiently. Such a need is particularly felt by managers and knowledge workers, who very often require automatic or semi-automatic tools to support their job. For this reason, during the last years the presence of *decision-support systems* in the market has been constantly increasing. The core of many decision support applications is formed by an integrated data repository called *data warehouse* (DW) [13], where data are represented in a multidimensional way. DW design goes through different phases, namely *conceptual design*, *logical design* and *physical design* [4]. In this work we focus on physical design; in particular, we propose a heuristic approach to index selection in relational DWs implemented through star schemes.

Given the DW logical scheme (including materialized views), a workload, the data volume and a constraint defining the disk space devoted to indexes, the goal is to determine the optimal *physical scheme*, that is, an index set that minimizes the workload execution cost respecting the space constraint. For this purpose we define an optimizer model that creates an execution plan for each query and a cost model to compare different solutions. The indexes considered are tid-list and bitmap indexes. The queries express aggregations over the star join between a fact table and a set of dimension tables; selections may be formulated on attributes of dimension tables. A set of potentially useful candidate indexes is preliminarly determined considering the workload. Then, a greedy algorithm progressively chooses, from the set of candidate indexes, the most beneficial ones while the space constraint is met.

The paper is organized as follows: Section 2 summarizes the main issues concerning logical and physical design; Section 3 outlines the functional architecture on which our approach relies; Section 4 analyzes in detail the component responsible of selecting the execution plan to solve each query; Section 5 presents the mathematical model used to evaluate and compare the costs of different execution plans; Section 6 shows the heuristic algorithm which determines the optimal index set to be built; finally, Section 7 draws the conclusions on the work carried out and gives some suggestions for future work.

## 2 From logical design to physical design

When DWs are implemented on relational DBMSs (ROLAP), the multidimensional view of data is achieved by adopting the so-called *star scheme*, or other schemes derived from it [13, 2]. The basic configuration for a star scheme includes:

- a set of *dimensional tables* (DTs), one for each dimension, each characterized by a surrogate primary key and by a set of attributes, representing the dimension of analysis at different aggregation levels. DTs are completely denormalized and consequently they do not represent functional dependencies.
- A *fact table* (FT) whose primary key is obtained by composing the foreign keys referencing the DTs. With reference to the terminology of the relation theory we call *prime* the attributes of the FT that are part of its primary key. Usually the FT also contains other attributes, called *measures*, that describe quantitatively each single instance of the fact.

One of the main requirements of the users of DW tools is to minimize the query response time. For this reason DW design includes, among the others, some phases specifically aimed at reducing the query response time as much as possible. One of the most effective ways to improve DW performance is *view materialization* [8, 1, 23, 2, 5]. A view contains aggregated data obtained from the base FT; the aggregation level characterizing a view is called its *aggregation pattern* and consists of a set of attributes from the DTs.

Because of the exponential dimension of the space of the possible views, materialization techniques must rely on an algorithm to select an optimal subset of views which satisfies an assigned constraint on disk space. Usually such algorithms are driven by a *workload*, consisting of a set of queries that are assumed to be representative of the wider set that will be submitted to the system during operation. Thus, the algorithms determine the set of views that should be materialized to optimize the response to the workload.

View materialization obviously involves a modification of the DW logical schema. Among the different alternatives proposed in the literature, in this paper we adopt the variant of the classic star scheme sketched in Figure 1 for the sale example, in which one separate FT is created for each materialized view and a separate DT is created for each attribute $a_i$ belonging to the aggregation pattern of at least one view. The DT for $a_i$ includes a surrogate key and a field for each attribute functionally dependent on $a_i$, including $a_i$ itself.

It should be noted that, in order to make the presence of materialized views transparent to the user, DBMSs are required to include a component, sometimes called *aggregate navigator*, that carries out a logical pre-optimization selecting for each query the FT that minimizes the access cost (typically, the smallest one among those on which the query could be solved).

Given the logical scheme of the DW, the goal of this work is to determine the physical scheme by defining the set of indexes, to be built on both FTs and DTs, that minimizes the workload execution cost respecting a given space constraint. Unlike previous phases, physical design strongly depends on the features of the specific DBMS: namely the categories of indexes available, the types of execution plans generated, the statistics consulted by the optimizer.

The architecture of DWs, in which updates are executed periodically when the system is off-line, makes the issues related to the updating costs for indexes absolutely marginal. Thus, new classes of indexes such as bitmap indexes [15], join indexes [15] or projection indexes [22, 16], which in operational systems are seldom used due to their high update cost, represent for DW designers an extremely valid instrument to improve performances [21].

Selecting the best indexes to be built is a very hard problem [9, 14]. In particular, the presence of several materialized views that can solve the same query creates an undesired inter-dependence between logical and physical design: in fact, the utility of a materialized view may depend on the set of indexes created on other views. Thus, a perfect algorithm should carry out logical and physical design simul-

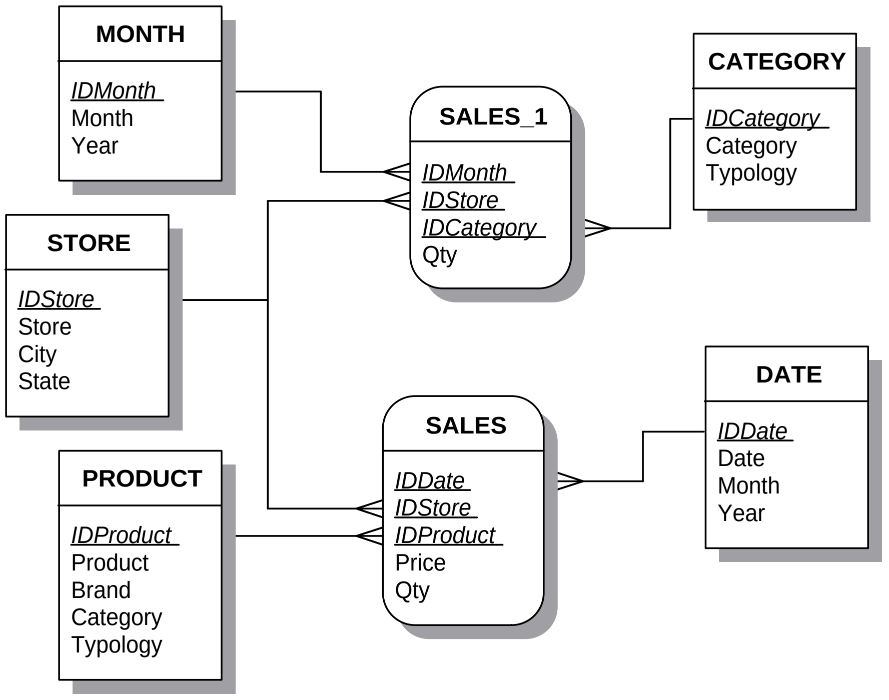

Figure 1: Logical schema consisting of a base FT and one materialized view.

taneously; since this approach is unpracticable in real cases due to its complexity, we will assume for simplicity that the best view available to solve a query is chosen independently of the physical scheme.

## 3 Architectural sketch

In this section we describe the functional architecture on which our approach is based. Our solution is depicted in Figure 2; the elements involved in processing are outlined and, for each of them, the function carried out is explained.

### 3.1 Inputs

In the following we briefly list the information we use for index selection:

- *DW logical scheme*: it describes the table structure and the relationships among them. In particular, it includes information related to both base tables and views. The logical model is the one described in Section 2.
- *Workload*: a set of queries $q_i$ to be executed on the DW, each characterized by a frequency $freq(q_i)$. The queries to be inserted in the workload should be selected according to their impact on the DW performance.
- *Data volume*: in order to evaluate the query execution cost on a given physical scheme, the optimizer requires quantitative information about data, such as the number of distinct values of each attribute.
- *System constraints*: index selection must take the limits imposed by hardware devices into account. The information used are the available disk space reserved to indexing, $S$, and the size of the memory buffer for hybrid hash joins, $hb$. While the first information is used to define an upper bound to the number of indexes, the second is used during the evaluation of the workload cost.

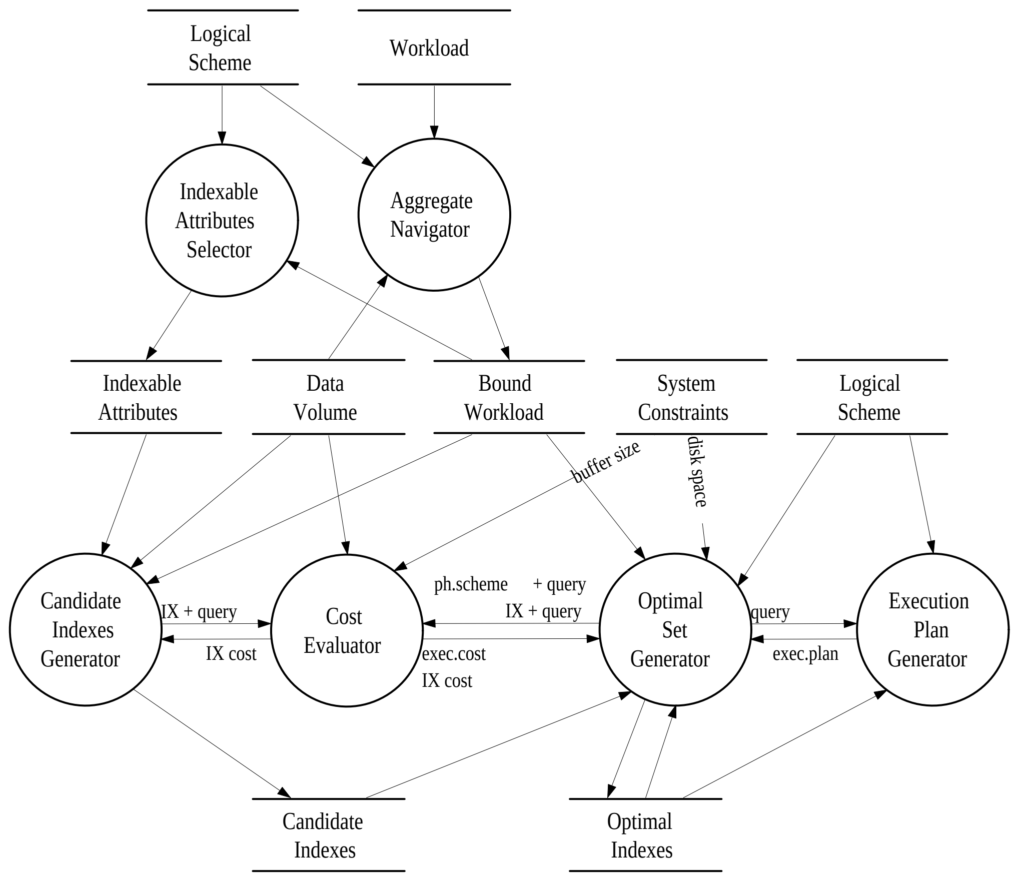

Figure 2: Functional architecture for index selection.

## 3.2 Processing

From a functional point of view, our approach features different components, each responsible for a particular function:

- *Aggregate Navigator*: given a workload and a logical scheme including one or more materialized views, this component is in charge of selecting the best view on which each query should be solved. The aggregate navigator does not usually take indexes into account.
- *Selector of Indexable Attributes*: based on the structure of the queries, this component determines which attributes of DTs could be usefully indexed.
- *Generator of Candidate Indexes*: for each indexable attribute, this component evaluates which type of index is the most convenient. The indexes selected by this component, defined by couples $(attribute, index\ type)$, are called *candidate indexes*.
- *Optimal Set Generator*: this component implements the algorithm which selects the indexes to be created. The optimal index set includes a subset of candidate indexes on DT attributes as well as *all* the indexes built on primary keys of DTs and FTs.
- *Cost Evaluator*: it is necessary to both the Generator of Candidate Indexes and the Optimal Set Generator to evaluate the access cost for each index.
- *Execution Plan Generator*: given a physical scheme, a query and the FT on which it should be solved, it returns the best execution plan which solves the query.

## 3.3 Outputs

Each processing component, given some inputs, returns some outputs which either act as inputs for other component(s) or are returned as the final solution. Such outputs are summarized in the following:

- *Bound Workload*: it stores, for each query, a reference to the FT used to solve the query.
- *Indexable Attributes*: the set of all the attributes that can be usefully indexed to speed up some queries.
- *Candidate Indexes*: it stores, for each indexable attribute, the most convenient index type.
- *Optimal Indexes*: the index set to be built.

## 3.4 Queries

The queries we consider are modeled as *GPSJ* (*Generalized Projection-Selection-Join*) expressions [7]. Briefly, a GPSJ expression is a selection $\sigma_2$ over a generalized projection $\pi$ over a selection $\sigma_1$ over a join $\chi$:

$$\sigma_2\ \pi\ \sigma_1\ \chi$$

where $\chi$ denotes the join among a FT and the related DTs. Selection $\sigma_1$ defines a filter on the tuples at the non-aggregated level, either on the FT measures or on the attributes in the DTs. Generalized projection $\pi$ defines the pattern on which tuples have to be aggregated (i.e. the attributes within the GROUP BY clause in the SQL formulation), as well as the aggregation operators to be used for each measure. Finally, selection $\sigma_2$ expresses a filter on the aggregated measures.

The class of queries we consider restricts GPSJ queries with reference to selections, in particular:

1. $\sigma_1$ only expresses conditions on DT attributes;
2. selections $\sigma_2$ on aggregated measures are not allowed.

Selections on measures are not considered since their presence affects the algorithm which selects the execution plan only if indexes on FT measures are available. Given that measures are usually defined on numeric domains with high cardinality, it is reasonable to build on them only indexes that store value ranges instead of single values. Furthermore, in many DBMSs this kind of indexes are not even implemented.

Thus, the queries we consider have the following form:

$$\pi_{P,M}\ \sigma_{Pred}\ \chi$$

where $P$ is an aggregation pattern, $M$ is a set of aggregated measures and $Pred$ is expressed as a conjunction of disjunctions of range predicates on DT attributes.

## 4 The optimizer

A DW usually contains several materialized views and indexes allowing a query to be executed in many different ways [6]. This section proposes an optimizer model (*Execution Plan Generator*, EPG) that, given a query, returns a possible execution plan. The model is closely based on the optimizer of Informix RedBrick 6.0 [11, 12], whose behavior was determined through a black-box analysis. In the following we report some preliminary considerations on the indexes and the join algorithms used by our EPG.

**Indexes.** EPG considers the following index types:

- *TID-List Index*: for each key value it stores a list of all the tids associated to the tuples assuming that key value.
- *Bitmap Index*: for each key value it stores a binary vector including one bit for each tuple in the table on which the index is built. Each bit is set to 1 if the corresponding tuple assumes that key value, 0 otherwise.

As shown in Figure 3, both index types use a $B^+$-tree as a support structure to reach either the list or the bit vector related to a given key value. Each index node and index leaf is stored in one disk page.

In order to simplify the problem of determining the indexes to be built, we introduce some restrictions: (1) the only indexes built on two or more attributes are those on the FT primary key, all the others involve only one attribute; (2) prime attributes are the only indexable attributes in FTs, while all the attributes in DTs are indexable; (3) at most one index can be built on each attribute. Furthermore, a tid-list index on each primary key of either a FT or a DT is always built in order to allow the DBMS to check integrity constraints.

**Join methods.** As to join methods, EPG considers:

- *Nested Loops Join with index on inner table*: either a FT or a DT can be set as the inner table;
- *Hybrid Hash Join* [20]: usually the smallest table is chosen as the build table. Since we only consider joins between a FT and a DT, we will use the DT as the build table and the FT as the probe table.

### 4.1 Cost-based vs. rule-based

Given the instruments available to EPG, it is possible to define the criteria it will adopt to analyze the search space.

Optimizers can be classified in two categories: *cost-based* and *rule-based*. A cost-based optimizer works by building the set of the feasible execution plans and by computing for each the execution cost based on the statistics about the database; finally, it selects the plan yielding the lowest execution cost.

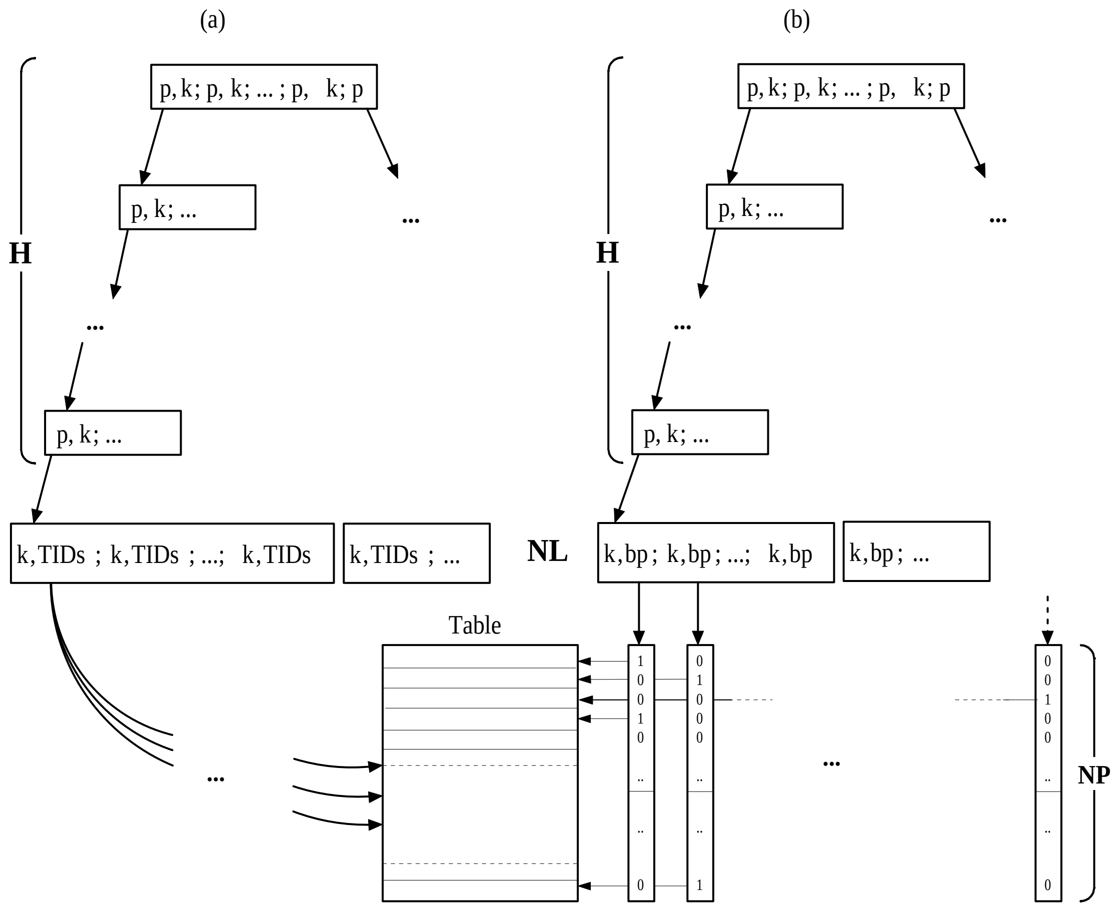
(a)
(b)
H
p, k; p, k; ... ; p, k; p
p, k; ...
...
k, TIDs ; k, TIDs ; ...; k, TIDs
k, TIDs ; ...
Table
NL
k, bp ; k, bp ; ...; k, bp
k, bp ; ...
NP

Figure 3: General structure of (a) a tid-list index and (b) a bitmap index.

On the other hand, a rule-based optimizer uses no statistics about the database and determines the execution plan my means of a set of heuristic rules based on the database structure (e.g. relationships between tables, indexes built on attributes, etc.). Therefore, a rule-based optimizer will produce the same execution plan for two queries that differ only for some constants in the selection predicates, regardless of their selectivity which may relevantly affect the execution cost.

The RedBrick optimizer will be considered to be rule-based since it does not use statistics while evaluating the index types and the join algorithms taken into account here (it uses statistics only when choosing among different *STAR indexes*). An example of a commercial DBMS relying on a cost-based optimizer is Oracle Enterprise 8.1.7 [18, 19, 17], which however can be forced to use a rule-based optimizer too. Moreover, Oracle allows specific commands called *hints* to be issued by the user in order to force a given execution plan.

Identifying the actual execution plans by a black-box analysis is an hard task that is made even harder by the degrees of freedom introduced by hints, thus we adopted for EPG the rule-based model inferred by analyzing RedBrick. Nevertheless, the chosen physical scheme will be valid for cost-based DBMSs too, since the statistics on data are used to evaluate the cost of execution plans.

## 4.2 Basic operators for execution plans

A query execution plan is a sequence of elementary operators applied to the database physical scheme. Each operator models a particular function carried out by the DBMS on either a table or an index, represented as parameters, and is characterized by an input and an output defined by applying set and couple constructors to three elemental types: *tid* (a tuple identifier), *value* (a value of an attribute), *tuple* (a value of a table tuple). Some operators allow to specify a Boolean predicate to filter the output.

We will sequence operators by means of arrows; the sequence $O_1 \rightarrow O_2$ denotes that the output produced by $O_1$ is consumed by $O_2$. Two operators can be sequenced only if the output of the first includes the input of the second. In particular, the sequence $O_1 \rightarrow O_2$ is correct if any combination of the following criteria occurs:

- $O_1$ returns a list and $O_2$ consumes an element belonging to that list.
- $O_1$ returns a tuple and $O_2$ consumes a value of an attribute belonging to the tuple.
- $O_1$ returns a set and $O_2$ consumes one of its elements, denoting that $O_2$ will consume separately and progressively each single element produced by $O_1$.
- $O_1$ returns a set and $O_2$ consumes a set of the same type, denoting that $O_2$ is called only once to consume the whole set of elements produced by $O_1$.

Finally, in some cases the execution of two or more subsequences must be considered to be concurrent; we will denote concurrency by enclosing the subsequences in parentheses, and separating them by commas. Thus, sequence $O_1 \rightarrow (O_{c_1}, \ldots, O_{c_m}) \rightarrow O_n$ is correct if:

- each sequence $O_1 \rightarrow O_{c_i}$ is correct according to the rules above; and
- the input to $O_n$ is either a couple or a set of the outputs of the $O_{c_i}$s.

Depending on their possible positions inside a plan, operators can be classified into three categories as follows; for each operator, the input accepted and the output produced are denoted.

1. *Starters*, all the operators that always appear at the beginning of an execution plan:
   - *Table Scan*,
   $$TS(table)_{pred} \longrightarrow \{tuple\}$$
   which sequentially scans $table$ verifying for each row the condition expressed by $pred$. In output it returns the set of all the tuples that satisfy $pred$.

- *Index Scan*,
$$XS(index)_{pred} \longrightarrow \{tid\}$$
which considers the condition expressed in $pred$ as a disjunction of elementary range conditions, and for each accesses $index$ using the lowest value in the range as the key. Once it reaches the index leaves, it scans them until the highest value in the range is reached. Finally, it returns the set of tids of the rows that satisfy $pred$.

2. *Linkers*, all the operators that always appear in intermediate positions:

- *Table Access*,
$$tid \longrightarrow TA(table)_{pred} \longrightarrow (tid, tuple)$$
which directly accesses $table$ to get the tuple related to the input tid, which is returned only if it satisfies $pred$.
- *Index Access*,
$$value \longrightarrow XA(index) \longrightarrow \{(tid, value)\}$$
which accesses $index$ with a given value for the index key, and returns a set of couples {(*tid*, *value*)} each associating that key value with the tid of a tuple yielding that value.
- *Hash Join*,
$$(\{tuple_1\}, \{tuple_2\}) \longrightarrow HJ(table_1, table_2) \longrightarrow \{(tid, tuple)\}$$
which carries out the natural join between two sets of tuples using the hybrid hash join algorithm. Parameters $table_1$ and $table_2$ are, respectively, the build and the probe tables. The output is a set of couples each combining a tuple returned by the join and the related tid of the probe table. In this context, the FT is always the probe table while the build table is one of the DTs.
- *TID Intersection*,
$$\{\{tid_1\}, \ldots, \{tid_n\}\} \longrightarrow TI(table) \longrightarrow \{tid\}$$
which receives as input at least two sets of tids, builds their intersection and returns it. The parameter is the table the tids refer to.

3. *Terminators*, all the operators that always appear at the end of an execution plan. For the class of queries we consider this set set contains one operator only:

- *Aggregation*,
$$\{tuple\} \longrightarrow AG() \longrightarrow \{tuple\}$$
which concatenates and groups the tuples in input.

## 4.3 Analysis of execution plans

The grammar defining the feasible execution plans for *EPG* can be defined as follows[1]:

[1]Square brackets denote sequences of symbols. Symbols '*' and '+' mean, respectively, that a sequence is repeated 0 o more times and 1 or more times

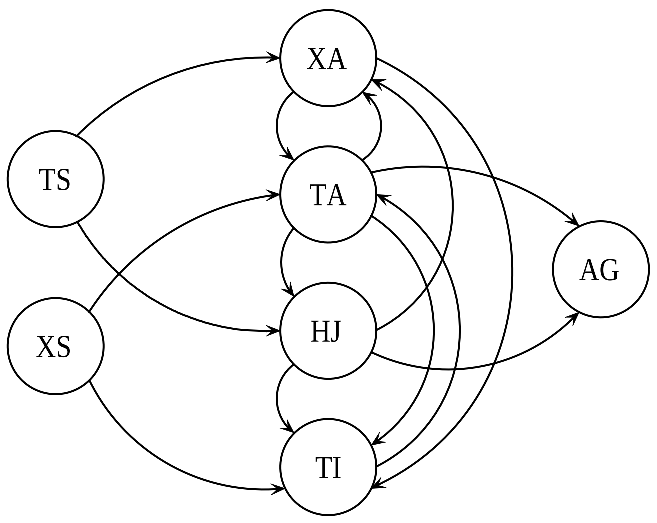

Figure 4: Possible sequencing of operators within an execution plan.

$$
\begin{aligned}
ExecutionPlan &::= FTaccess[\rightarrow DTsAccess] \rightarrow \mathtt{AG}()\\
FTaccess &::= \mathtt{TS}(\mathtt{FT}) \mid DThashFT \mid FTXaccess \rightarrow \mathtt{TA}(\mathtt{FT})\\
DTsAccess &::= (\mathtt{XA}(\mathtt{KDTX}) \rightarrow \mathtt{TA}(\mathtt{DT})[, \mathtt{XA}(\mathtt{KDTX}) \rightarrow \mathtt{TA}(\mathtt{DT})]^{*})\\
DThashFT &::= (DTaccess, \mathtt{TS}(\mathtt{FT})) \rightarrow \mathtt{HJ}(\mathtt{DT}, \mathtt{FT})\\
FTXaccess &::= DTaccess \rightarrow \mathtt{XA}(\mathtt{FTX}) \mid\\
&\quad ([DTaccess \rightarrow \mathtt{XA}(\mathtt{FTX}) \mid DThashFT]\\
&\quad [, DTaccess \rightarrow \mathtt{XA}(\mathtt{FTX}) \mid DThashFT]^{+}) \rightarrow \mathtt{TI}(\mathtt{FT})\\
DTaccess &::= \mathtt{TS}(\mathtt{DT})_{\mathtt{Pred}} \mid DTXaccess \rightarrow \mathtt{TA}(\mathtt{DT})_{\mathtt{Pred}}\\
DTXaccess &::= \mathtt{XS}(\mathtt{DDTX})_{\mathtt{Pred}} \mid\\
&\quad (\mathtt{XS}(\mathtt{DDTX})_{\mathtt{Pred}}[, \mathtt{XS}(\mathtt{DDTX})_{\mathtt{Pred}}]^{+}) \rightarrow \mathtt{TI}(\mathtt{DT})
\end{aligned}
$$

where:

| | |
|---|---|
| FT | is a FT; |
| DT | is a DT; |
| Pred | is a condition expressed on DT attributes; |
| FTX | is an index built either on the FT primary key or on a FT prime attribute; |
| DDTX | is an index built on a DT attribute; |
| KDTX | is an index built on a DT primary key. |

Figure 4 represents the possible sequencing of operators within an execution plan.

**Example.** Let us consider the following string representing a query execution plan:

$$\mathtt{TS}(\mathtt{DT}_1)_{\mathtt{Pred}} \rightarrow \mathtt{XA}(\mathtt{FTX}) \rightarrow \mathtt{TA}(\mathtt{FT}) \rightarrow (\mathtt{XA}(\mathtt{KDTX}_2) \rightarrow \mathtt{TA}(\mathtt{DT}_2), \mathtt{XA}(\mathtt{KDTX}_3) \rightarrow \mathtt{TA}(\mathtt{DT}_3)) \rightarrow \mathtt{AG}()$$

where FTX is an index on FT whose first attribute is the key referencing $\mathtt{DT}_1$; $\mathtt{KDTX}_2$ and $\mathtt{KDTX}_3$ are the indexes on the primary keys of $\mathtt{DT}_2$ and $\mathtt{DT}_3$, respectively. The plan, shown in Figure 5, starts with the sequential scan of $\mathtt{DT}_1$ in order to apply the filter expressed in Pred. Then, for each tuple of $\mathtt{DT}_1$

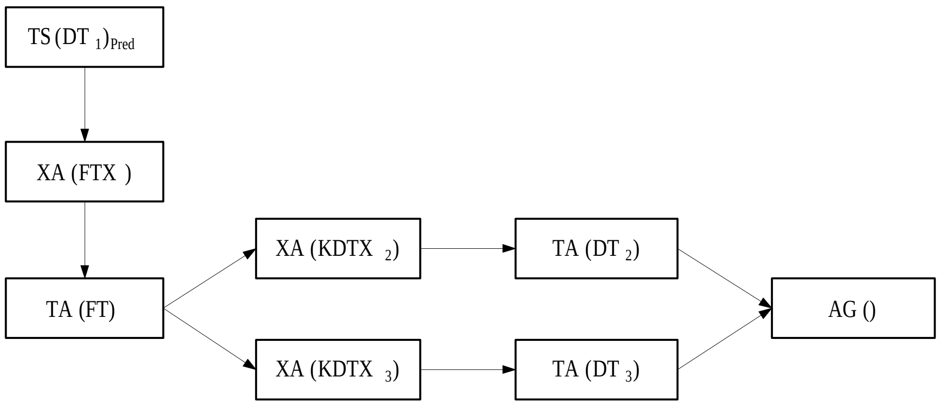

Figure 5: Graphical representation of an execution plan.

satisfying `Pred`, `FTX` is accessed and the resulting tids are used to retrieve the `FT` tuples. For each of the tuples selected in `FT`, $\mathtt{DT}_2$ and $\mathtt{DT}_3$ are joined, accessing the primary key index first, then retrieving the tuple using the tid retrieved from the index. Finally, the joined tuples are grouped.

## 4.4 Selection of an execution plan

Figure 6 sketches the flow-chart describing the algorithm used in EPG to select an execution plan for query $\pi_{P,M}\,\sigma_{Pred}\,(FT \bowtie DT_1 \bowtie \cdots \bowtie DT_n)$. It should be noted that the join between $FT$ and $DT_i$ (when $FT$ is external and $DT_i$ is internal) is always executed by accessing, for each tuple of $FT$ that satisfies the conditions specified in the query, the index built on the $DT_i$ primary key. The final aggregation on $P$ (expressed in SQL by the GROUP BY clause) is common to all the execution plans, thus we will not consider it.

The decision flow is mainly determined by the number of DTs on which at least one condition is expressed, which are called *conditioned* DTs:

- If no conditioned DT are present, $FT$ is sequentially scanned then joined with all the DTs involved in the query.
- If only one conditioned DT is present, $DT_c$, EPG checks if there is an index allowing to access $FT$ from its foreign key referencing $DT_c$ (it may be either a single-attribute index on the foreign key or an index on the primary key of $FT$ where the foreign key is in the first position). If so, for each tuple of $DT_c$ that satisfies the selection predicate $Pred$, EPG accesses this index and then $FT$. Otherwise, an hybrid hash join between $DT_c$ filtered by $Pred$ and $FT$ is executed. In both cases, the result is eventually joined with the other DTs.
- If there are two or more conditioned DTs, for each of them EPG decides how to carry out the join with $FT$ (nested loops if there is an index on the corresponding foreign key in $FT$, hybrid hash otherwise). Each join returns the subset of FT tids whose related DT tuple satisfies $Pred$. The tid sets obtained from the different conditioned DTs are then intersected, and the resulting tuples of $FT$ are accessed.

As to conditioned DT access, the plan changes according to the set of indexes and attributes interested by the query predicate as shown in Figure 7. The number of DT attributes on which a filter is defined and an index is built, $\alpha$, drives the choice of the plan as follows:

1. If $\alpha = 0$, a sequential scan of the DT is executed, applying the filter to each tuple.
2. $\alpha = 1$ means that an index on a conditioned attribute is built. In this case an index scan is executed and, for each tid retrieved, the DT is accessed. The tuples retrieved can be further filtered by applying conditions expressed on other DT attributes to them.

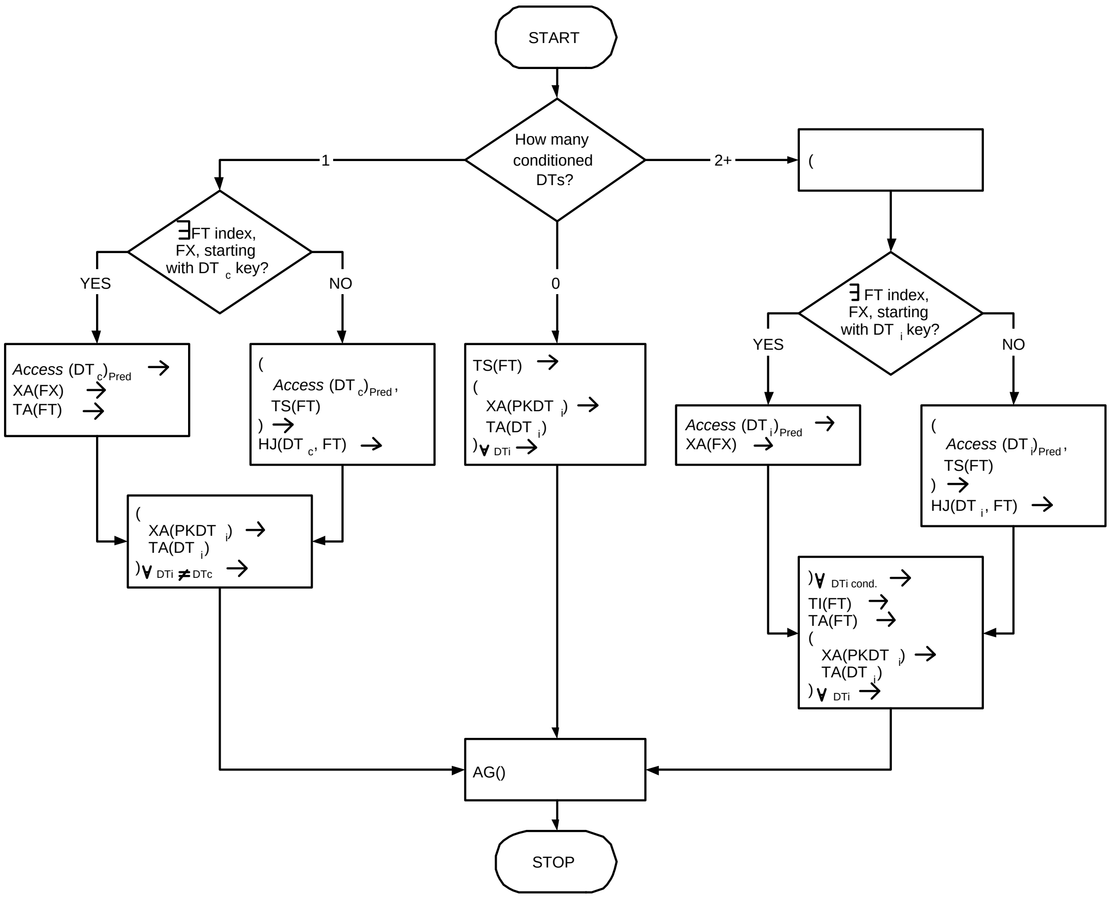
START
How many conditioned DTs?
1
0
2+
(
∃FT index, FX, starting with DT c key?
YES
NO
Access (DT c)Pred →
XA(FX) →
TA(FT) →
( Access (DT c)Pred , TS(FT) ) →
HJ(DT c, FT) →
( XA(PKDT i) → TA(DT i) )∀ DTi ≠ DTc →
TS(FT) →
( XA(PKDT i) → TA(DT i) )∀ DTi →
∃FT index, FX, starting with DT i key?
Access (DT i)Pred →
XA(FX) →
( Access (DT i)Pred , TS(FT) ) →
HJ(DT i, FT) →
)∀ DTi cond. →
TI(FT) →
TA(FT) →
( XA(PKDT i) → TA(DT i) )∀ DTi →
AG()
STOP

Figure 6: Flow-chart for selecting the best execution plan for a query.

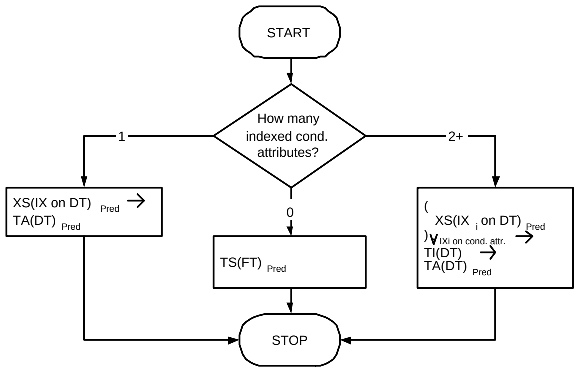
START
How many indexed cond. attributes?
1
0
2+
XS(IX on DT) Pred →
TA(DT) Pred
TS(FT) Pred
( XS(IX i on DT) Pred )∀ IXi on cond. attr. →
TI(DT) →
TA(DT) Pred
STOP

Figure 7: Flow-chart for determining the DT access plan.

| **System information** | |
|---|---|
| $b$ | disk page size in bytes |
| $hb$ | size in bytes of the buffer used for hybrid hash join |
| **Table statistics** | |
| $NT$ | number of tuples |
| $t$ | tuple size in bytes |
| $NP$ | number of disk pages |
| **Index statistics** | |
| $NK$ | number of distinct key values |
| $NT$ | overall number of tids stored |
| $NL$ | number of leaves (see Fig. 3) |
| $H$ | height of the B$^+$-tree (see Fig. 3) |
| $NB$ | number of disk pages to store a bitmap |

Table 1: Statistics used by the cost evaluator.

3. If $\alpha \geq 2$, all indexes on conditioned attributes are accessed, the tid sets obtained are intersected, possibly further filters are applied, and finally the DT is accessed.

## 5 The cost model

In order to compare different physical schemes, a cost model is necessary to evaluate each execution plan. According to the cost model adopted in this paper, the cost of a plan is expressed as the number of logical pages that must be read to execute it [10, 9, 3].

Evaluating the cost of a plan requires a cost function for each single operator appearing in it; this cost may depend on the output of the previous operator, in particular on its cardinality. The cost of the full plan is the sum of costs of all its operators. Table 1 reports the information required by the cost model, divided in two categories: system information and database statistics.

In the following, the cost function and the output cardinality are reported for each operator (except $AG$ which is assumed to have a null cost). Function $sel(Pred)$ returns the portion of tuples satisfying Boolean predicate $Pred$; function $int$ rounds to the nearest integer.

- $TS(table)_{Pred}$

$$cost = NP$$
$$\#output = NT * sel(Pred)$$

- $XS(index)_{Pred}$

$$cost = \begin{cases} \sum_{\forall RangePred}(H + \lceil NL * sel(RangePred) \rceil) & \text{, if } index \text{ is a tid-list;} \\ \sum_{\forall RangePred}(H + \lceil NL * sel(RangePred) \rceil) & \\ +int(NK * sel(RangePred)) * NB & \text{, if } index \text{ is a bitmap} \end{cases}$$
$$\#output = NT * sel(Pred)$$

- $TA(table)_{Pred}$

$$cost = NP * \left(1 - \left(1 - \frac{1}{NP}\right)^{\#input}\right)$$
$$\#output = \#input * sel(Pred)$$

- $XA(index)$

$$cost = \begin{cases} \#input * \left(H + \lceil \frac{NL}{NK} \rceil\right) & \text{, if } index \text{ is a tid-list;} \\ \#input * (H + 1 + NB) & \text{, if } index \text{ is a bitmap} \end{cases}$$
$$\#output = \#input * \frac{NT}{NK}$$

- $(\#input_1, \#input_2) \rightarrow HJ(table_1, table_2)$

$$\#HashPartitions = \left\lceil \frac{\#input_1 * t}{hb} \right\rceil$$
$$cost = \begin{cases} 0 & \text{, if } \#HashPartitions = 1 \\ \left\lceil \frac{\#input_2 * t_2}{b} \right\rceil * \#HashPartitions & \text{, otherwise} \end{cases}$$
$$\#output = \frac{\#input_1}{NT_1} * \#input_2$$

- $(\#input_1, \ldots, \#input_n) \rightarrow TI(table)$

$$cost = 0$$
$$\#output = \frac{\#input_1}{NT} * \ldots * \frac{\#input_n}{NT} * NT = \frac{\prod_i \#input_i}{NT^{n-1}}$$

**Example.** Consider the following execution plan:

$$TS(DT_1)_{Pred} \rightarrow XA(FTX) \rightarrow TA(FT) \rightarrow$$
$$\rightarrow (XA(KDTX_2) \rightarrow TA(DT_2), XA(KDTX_3) \rightarrow TA(DT_3)) \rightarrow AG()$$

executed on a database with the following statistics:

$$NP_{DT_1} = 368,\ NT_{DT_1} = 50000$$
$$H_{FTX} = 2,\ NL_{FTX} = 8000,\ NK_{FTX} = 1000000$$
$$NT_{FT} = 10000000,\ NP_{FT} = 85470$$
$$H_{KDTX_2} = 1,\ NL_{KDTX_2} = 5$$
$$NT_{DT_2} = 4000,\ NP_{DT_2} = 50$$
$$H_{KDTX_3} = 2,\ NL_{KDTX_3} = 1250$$
$$NT_{DT_3} = 1000000,\ NP_{DT_3} = 9804$$
$$sel(Pred) = 0.3$$

For each single operator we have:

$$\begin{aligned}
TS(DT_1)_{Pred}: &\quad \mathit{cost} = 368 \\
&\quad \#\mathit{output} = 50000 * 0.3 = 15000 \\
XA(FTX): &\quad \mathit{cost} = 15000 * \left(2 + \left\lceil \frac{8000}{1000000} \right\rceil \right) = 45000 \\
&\quad \#\mathit{output} = 15000 * \frac{10000000}{1000000} = 150000 \\
TA(FT): &\quad \mathit{cost} = 70692 \\
&\quad \#\mathit{output} = 150000 \\
XA(KDTX_2): &\quad \mathit{cost} = 150000 * \left(1 + \left\lceil \frac{5}{4000} \right\rceil \right) = 300000 \\
&\quad \#\mathit{output} = 150000 * \frac{4000}{4000} = 150000 \\
TA(DT_2): &\quad \mathit{cost} = 50 \\
XA(KDTX_3): &\quad \mathit{cost} = 150000 * \left(2 + \left\lceil \frac{1250}{1000000} \right\rceil \right) = 450000 \\
&\quad \#\mathit{output} = 150000 * \frac{1000000}{1000000} = 150000 \\
TA(DT_3): &\quad \mathit{cost} = 9804
\end{aligned}$$

Thus, the total cost of the plan turns out to be:

$$\mathit{cost} = 368 + 45000 + 70692 + 300000 + 50 + 450000 + 9804 = 875914\ \mathit{pages}$$

## 6 The index selection algorithm

Due to its high complexity, the index selection problem is usually faced heuristically. As already stated, the view used to solve a query is selected during logical design, which is carried out neglecting the issues related to indexing.

An attribute $a \in DT$ is said to be *indexable* if at least one query in the workload expresses a condition on $a$ and is solved on an FT linked to $DT$ [2]. A prime attribute $a \in FT$, $a$ referencing $DT$, is *indexable* if at least one query in the workload expresses a condition on an attribute in $DT$ and requires data stored in $FT$. The queries that make $a$ indexable are called the *support* for $a$.

Indexable attributes and primary keys of tables are the only elements that may be indexed. It should be noted that indexing an indexable attribute does not necessary lead to a performance improvement; on the other hand, if an index on an indexable attribute is built, the execution plans for all the queries in its support will use that index.

It is remarkable that, in the physical scheme, each index is independent of the others. In fact, given an index $IX$ on an indexable attribute whose support contains query $q_i$, EPG will always use $IX$ in the same way and with the same cost regardless of the contemporary presence of other indexes. The contribution of $IX$ to the execution cost of $q_i$, $qCost(IX, q_i)$, depends on the table on which $IX$ is bult. If $IX$ is built on a DT attribute, it is accessed by a scan driven by the selection predicate of $q_i$:

$$qCost(IX, q_i) = \mathit{cost}(XS(IX)_{Pred_i})$$

If $IX$ is built on a prime attribute of the FT, it is accessed once for each of the $ET$ tuples of the DT that satisfy the selection predicate:

$$qCost(IX, q_i) = \mathit{cost}(XA(IX)) \cdot ET_{Pred_i}$$

[2]The same attribute $a$ may appear on several DTs due to view materialization.

Now, it is possibile to define for $IX$ a *total cost* its global contribution to the workload cost, computed as the weighted sum of its contributions to the single queries:

$$tCost(IX) = \sum_{q_i} freq(q_i) \cdot qCost(IX, q_i)$$

and a *weighted cost* as its size in disk pages, $sizeP(IX)$, times its total cost: $sizeP(IX) \cdot tCost(IX)$. The weighted cost is used to compare different types of indexes (tid-list and bitmap) built on the same attribute. Thus, for each indexable attribute $a$, the corresponding candidate index $IX = (a, index\ type)$ is the one whose weighted cost is minimal.

Usually designers reserve a fixed disk space $S$ to store indexes; such space can be partitioned into three parts whose sizes are defined a priori: (1) one part, $S_{KDT}$, for indexes on DT primary keys; (2) one part, $S_{KFT}$, for indexes on FT primary keys; and (3) the remaining part, $S_{free}$, for all the other indexes. We assume that only tid-list indexes are built on primary keys and that primary keys of DTs are surrogated, so that $S_{KDT}$ can be easily calculated. Also the space contribution $S_{KFT}$ can be easily computed a priori since the size of each index on the primary key of a FT only depends on the number of prime attributes, not on their ordering. Finally, the space contribution for other indexes is $S_{free} = S - S_{KDT} - S_{KFT}$.

The pseudo-code for the index selection algorithm is proposed in the following. $C$ and $O$ represent, respectively, the set of candidate indexes and the set of optimal indexes, i.e. those actually built.

```
procedure $BuildOptimalIndexSet()$
{  $C = initializeC()$;    // initialization of the set of candidate indexes
   $O = initializeO()$;    // initialization of the set of optimal indexes
   $S_{free} = S - S_{KFT} - S_{KDT}$;    // expressed in disk pages
   while $(\exists IX \in C : sizeP(IX) \leq S_{free})$ do
   {  $IX_{max} = argmax_{\{IX \in C : sizeP(IX) \leq S_{free}\}}\{benefitPerPage(IX, O)\}$;
      $O\cup = \{IX_{max}\}$;
      $C- = \{IX_{max}\}$;
      $S_{free}- = sizeP(IX_{max})$;
      if $\exists FT : attr(IX_{max}) \in prime(FT)$ and $\forall a_i \in prime(FT) \exists IX \in O : a_i = attr(IX)$
      {  $IX_{min} = argmin_{\{IX \in O : attr(IX) \in prime(FT)\}}\{decayPerPage(IX)\}$;
         $O- = \{IX_{min}\}$;
         $S_{free}+ = sizeP(IX_{min})$;
         $O\cup = \{multInd(attr(IX_{min}))\}$;
      }
   }
   for each $FT$ : $O$ does not contain any index on the primary key of $FT$
     if $\exists IX \in C : attr(IX) \in prime(FT)$
     {  $IX_{min} = argmin_{\{IX \in C : attr(IX) \in prime(FT)\}}\{benefitPerPage(multInd(attr(IX)), O)\}$;
        $C- = \{IX_{min}\}$;
        $O\cup = \{multInd(attr(IX_{min}))\}$;
     }
     else
     {  $a$ = any prime not indexable attribute of $FT$;
        $O\cup = \{multInd(a)\}$;
     }
}
```

where:

- $initializeC()$ returns the set $C$ of candidate indexes for the workload. The set includes, for each indexable attribute, the most useful candidate index selected according to its weighted cost.

- $initializeO()$ initializes the set of the optimal indexes, $O$. For each DT, it inserts in $O$ a tid-list index built on the primary key.
- $sizeP(IX)$ returns the size of $IX$ in disk pages.
- $attr(IX)$ returns the ordered list of the attributes on which $IX$ is built.
- $prime(FT)$ returns the set of prime attributes of $FT$ .
- $multInd(a)$: given a prime attribute $a$ of $FT$, this function returns a (multiple) tid-list index on the primary key of $FT$ whose first attribute is $a$. The order of the other attributes is not interesting.
- $benefitPerPage(IX, O)$ returns the relative benefit of $IX$, estimated as

  $$benefitPerPage(IX, O) = \frac{wklCost(O) - wklCost(O \cup \{IX\})}{sizeP(IX)}$$

  where $wklCost(O)$ is the execution cost, expressed in disk pages, for the whole workload when the indexes in $O$ are built.
- $decayPerPage(IX)$: given index $IX$ on attribute $a$, this function returns the relative performance decay due to transforming $IX$ from single-attribute to multiple-attribute index:

  $$decayPerPage(IX) = \frac{tCost(multInd(a)) - tCost(IX)}{sizeP(IX)}$$

The algorithm can be subdivided into three distinct sections. The first one initializes the sets of candidate and optimal indexes as well as the available space for indexes on non-prime attributes, $S_{free}$.

The second section, delimited by the while loop, carries out a greedy selection of indexes from $C$ based on the benefit per index page. If, after inserting a new index in $O$, it turns out that all the prime attributes of a FT are indexed, one of these indexes can be transformed into a multiple-attribute index on the FT primary key; the choice is driven by the decay per index page related to the transformation. It should be noted that the decay per index page can be computed by comparing the total costs since it is used to decide which single-attribute index on a prime attribute of the FT should be transformed into a multiple index on the FT primary key, and this transformation does not affect execution plans. On the other hand, the benefit per index page must be computed with reference to the whole workload cost since dropping an index from $O$ may radically impact on the execution plans adopted.

Once all indexes have been selected, the third section sets up the primary key indexes for the remaining FTs. If, for a given FT, a non-empty set of candidate indexes still exists, the one whose insertion in $O$ as a multiple-attribute index on the primary key is cheapest is chosen. Otherwise, a non-indexable attribute is randomly chosen to build the multiple index.

## 7 Conclusions

In this paper we proposed a heuristic approach to the problem of index selection in a data warehouse with materialized views. Obviously, the approach devised needs to be implemented and extensively tested to evaluate its benefit, and to be characterized in terms of its computational complexity; these goals will be achieved during the next phase of the project.

Future work on this topic includes considering other types of indexes (e.g. indexes on two or more tables such as join and star indexes) and join algorithms (e.g. sort merge).

## References

[1] E. Baralis, S. Paraboschi, and E. Teniente. Materialized view selection in multidimensional database. In *Proc. 23rd VLDB*, pages 156–165, Athens, Greece, 1997.

[2] G. K. Y. Chan, Q. Li, and L. Feng. Optimized Design of Materialized Views in a Real-Life Data Warehousing Environment. *International Journal of Information Technology*, 7(1), 2001.

[3] A. Datta, B. Moon, K. Ramamritham, H. Thomas, and I. Viguier. ”Have your Data and Index it, too” Efficient Storage and Indexing for Data Warehouses. Technical Report 98-7, Department of Computer Science, The University of Arizona, 1998.

[4] M. Golfarelli and S. Rizzi. Designing the data warehouse: key steps and crucial issues. *Journal of Computer Science and Information Management*, 2(3), 1999.

[5] M. Golfarelli and S. Rizzi. View Materialization for Nested GPSJ Queries. In *Proc. DMDW’2000*, Stockholm, 2000.

[6] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[7] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-Query Processing in Data Warehousing Environments. In *Proc. 21st VLDB*, Zurich, Swizerland, 1995.

[8] H. Gupta. Selection of Views to Materialize in a Data Warehouse. In *ICDT*, pages 98–112, 1997.

[9] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index Selection for OLAP. In *ICDE*, pages 208–219, 1997.

[10] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing Data Cubes Efficiently. In *Proc. ACM Sigmod Conf.*, Montreal, Canada, 1996.

[11] Informix. *Administrator’s Guide Informix Red Brick Decision Server, Version 6.0*, November 1999.

[12] Informix. *SQL Reference Guide Informix Red Brick Decision Server, Version 6.0*, November 1999.

[13] R. Kimball. *The data warehouse toolkit*. John Wiley & Sons, 1996.

[14] W. J. Labio, D. Quass, and B. Adelberg. Physical Database Design for Data Warehouses. In *ICDE*, pages 277–288, 1997.

[15] P. O’Neil. INFORMIX and Indexing Support for Data Warehouses. *Database Programming and Design*, 10(2):38–43, February 1997.

[16] P. O’Neil and D. Quass. Improved Query Performance with Variant Indexes. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 38–49, 1997.

[17] Oracle. *Designing and Tuning for Performance*, December 1999.

[18] Oracle. *Administrator’s Guide for Windows NT*, January 2000.

[19] Oracle. *SQL Reference*, September 2000.

[20] J. M. Patel, M. J. Carey, and M. K. Veron. Accurate Modeling of The Hybrid Hash Join Algorithm. In *Measurement and Modeling of Computer Systems*, pages 56–66, 1994.

[21] S. Sarawagi. Indexing OLAP Data. *Data Engineering Bulletin*, 20(1):36–43, 1997.

[22] I. R. Viguier and A. Datta. Sizing Access Structures for Data Warehouses. Technical report, Dept. of MIS, University of Arizona, Tucson, 1997.

[23] J. Yang, K. Karlaplem, and Q. Li. Algorithms for Materialized View Design in DataWarehousing Environments. In *Proc. 23rd VLDB*, pages 136–145, Athens, Greece, 1997.