Index selection for data warehousing [*]

Matteo Golfarelli, Stefano Rizzi, Ettore Saltarelli
DEIS, University of Bologna, Italy

Abstract

This paper addresses a basic issue in physical design of data warehouses by proposing a heuristic approach which selects an optimal set of indexes to be created on a ROLAP implementation. To achieve this goal we simulate a rule-based optimizer which generates query execution plans that are evaluated according a specific cost model. The indexes considered belong to two very common categories: tid-list indexes and bitmap indexes, both accessed via a B^+-tree. 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. The validity of the approach is evaluated with reference to some experimental tests, part of which are aimed at comparing the relative benefits arising from view materialization and indexing.

1 Introduction

During the design of a data warehouse (DW), the phases aimed at improving the system performance are logical and physical design. On relational (*ROLAP*) implementations, the multidimensional view of data at the logical level is achieved by adopting the so-called *star scheme*, composed by a set of *dimensional tables*, one for each dimension of analysis, and a *fact table* whose primary key is obtained by composing the foreign keys referencing the dimension tables.

A basic requirements of DW users is to obtain quick answers for their queries. One of the most effective ways to achieve this goal during logical design is *view materialization* [11]. A view contains aggregated data obtained from the base fact table containing elemental data; the aggregation level characterizing a view is called its *pattern* and consists of a set of attributes from the dimension tables. Also materialized views are modeled according to the star scheme. Though the impact of materialized views on overall performance is very strong, using indexing techniques is still fundamental. Indexing, together with all the issues related to implementing the DW on a specific DBMS, are considered by physical design.

Given the logical scheme of the DW, defining its physical scheme requires to determine the set of indexes to be built on both fact tables and dimension tables. Typically, the set that minimizes the workload execution cost respecting a given space constraint is sought. From the computational point of view, this is a very hard problem; 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 simultaneously; since this approach is unfeasible in real cases due to its complexity, in practical cases the best view available to solve a query is chosen *independently* of the physical scheme.

Indexing strongly depends on the features of the specific DBMS: first of all on the categories of indexes available, but also on the types of execution plans generated and on the

[*]This work has been partially supported by the D2I MURST project.

statistics consulted by the optimizer. The peculiar features of DW applications allow several types of indexes, which in operational systems are seldom used due to their high update cost, to be considered [10]: for instance bitmap indexes, join indexes, and projection indexes [7, 8].

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), an OLAP 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. To this purpose we define a rule-based optimizer model capable of determining an execution plan for each query. The indexes considered in this work are tid-list and bitmap indexes, both accessed via a B$^+$-tree. The queries express aggregations over selections over the star join between a fact table and a set 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 satisfying the space constraint.

Despite the high number of indexing techniques devised, only a few works in the literature focus on the selection of indexes for DWs. In [6] the authors propose both an optimal algorithm and a set of thumb rules that should be adopted when the problem size is intractable. Rules, that are justified by the adoption of appropriate cost functions, state that indexes should be created on keys and on attributes involved in joins, as well as when their size fits into main memory. In [3] the problem of simultaneously choosing views and B$^+$-tree indexes is investigated; the linear cost function adopted is very simple, and no specific optimizer model is considered.

The paper is organized as follows: Section 2 briefly describes the functional architecture on which the approach is based; Section 3 analyzes in detail the component responsible of selecting the execution plan to solve each query; Section 4 describes the cost model adopted; Section 5 outlines the heuristic algorithm which determines the optimal index set to be built; finally, Section 6 reports some experimental tests, draws the conclusions on the work carried out and gives some suggestions for future work.

2 Architectural sketch

In this section we briefly describe the functional architecture on which our approach is based, sketched in Figure 1, whose components are briefly described in the following:

- The *logical scheme* is the relational scheme for fact tables (including both base tables and aggregate views) and dimension tables.
- The *workload* is a set of queries to be executed on the DW. The queries we consider are modeled as *GPSJ* (*Generalized Projection-Selection-Join*) expressions [2] in the shape $\pi_{P,M}\ \sigma_{Pred}\ \chi$ where χ denotes the star join among a fact table and the related dimension tables, $Pred$ is a conjunction of simple range predicates on different dimension table attributes, P is a pattern, and M is a set of aggregated measures. Generalized projection $\pi_{P,M}$ defines the pattern P on which tuples have to be aggregated as well as the aggregation operators to be used for each measure.
- The *data volume* contains quantitative information about data, such as the size of tables and the domain cardinality of attributes.
- *System constraints* include the available disk space reserved to indexing, S, and the size of the memory buffer for hybrid hash joins, hb.

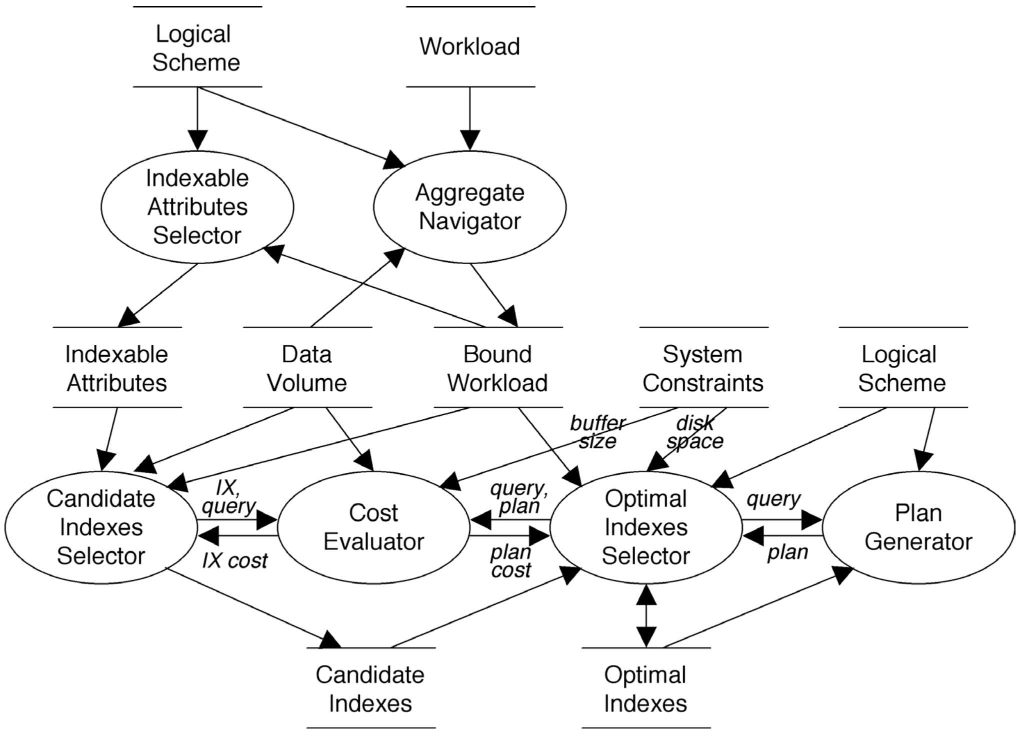

Figure 1: Functional architecture for index selection.

- The *bound workload* couples each query in the workload with a reference to the fact table used to solve it.
- The attributes that can be usefully indexed to speed up some queries are called *indexable attributes*. The set of *candidate indexes* couples each indexable attribute with the most convenient index type for it. In this approach, indexes are always built on a single attribute except for indexes on the fact table primary key. The set of *optimal indexes* to be built includes a subset of candidate indexes as well as *all* the indexes built on primary keys of dimension tables and fact tables.

The processing elements which use these objects to accomplish index selection are:

- the *aggregate navigator* which, given a workload and a logical scheme including one or more materialized views, selects the best view on which each query should be solved without taking indexes into account;
- the *indexable attributes selector* which, based on the structure of the queries, determines which attributes of dimension tables can be usefully indexed;
- the *candidate indexes selector* which, for each indexable attribute, evaluates which type of index is the most convenient;
- the *optimal indexes selector* which selects the indexes to be created out of the set of candidate indexes;
- the *cost evaluator* used to evaluate both the cost of each index and that of each execution plan;

- the *plan generator* which given a physical scheme, a query q and the view v on which q should be solved, returns the best execution plan which solves q on v.

The aggregate navigator and the plan generator, together with the cost evaluator, implement the query optimizer. Decoupling the choice of the view to execute the query and the choice of the plan to access that view allows for reducing the complexity of optimization, and reflects the approach adopted in most DBMSs for data warehousing when the presence of materialized views is meant to be transparent to the user.

3 Plan generation

This section proposes a rule-based model for the plan generator which, given a query and the view (i.e. the fact table) on which it will be solved, returns the execution plan estimated to be the "best". The model is strictly based on the optimizer of Informix Red Brick 6.0 [5], whose behavior was determined through a black-box analysis. After describing in Section 3.1 the elemental operators appearing in execution plans, in Section 3.2 we explain how the plan for a query is chosen.

3.1 Operators for execution plans

A query execution plan is a sequence of elementary operators applied to the physical scheme. Each operator models a function carried out by the DBMS on either tables or indexes and is characterized by an input and an output; some operators allow a local predicate to be specified in order to filter the output.

The operators are briefly described below:

- *Table scan* sequentially scans a table and returns the set of all the tuples that satisfy a given selection predicate.
- *Index scan* accesses an index and retrieves the tids of the tuples that satisfy a given selection predicate.
- *Table access* accesses a table to get the tuple related to a given tid, which is returned only if it satisfies a given predicate.
- *Index access* accesses an index to retrieve the set of tids of the tuples yielding a given value for the index key.
- *Hash join* carries out the natural join between two sets of tuples using the hybrid hash join algorithm.
- *Tid intersection* returns the intersection between two sets of tids.

While table scan and index scan always appear at the beginning of a plan, all the others appear in intermediate positions.[1]

Example 1 *Let us consider the star scheme derived from the TPC-H [9]:*

PART($\underline{\text{PartId}}$, Part, Brand, MFGR, Type, Container, Size)
SUPPLIER($\underline{\text{SupplierId}}$, Supplier, SNation, SRegion)
ORDER($\underline{\text{OrderId}}$, Order, ODate, OMonth, OYear, Customer)
LINEITEM($\underline{\text{PartId}}$, $\underline{\text{SupplierId}}$, $\underline{\text{OrderId}}$, $\underline{\text{ShipDate}}$, Qty, ExtPrice, Discount, DiscPrice, UnitPrice, Tax)

and the following GPSJ query on it:

$$\pi_{\text{Type},\text{SNation},\text{OYear},SUM(\text{ExtPrice}),AVG(\text{UnitPrice})}\ \sigma_{\text{SRegion}='\text{West}'}\ (\text{PART} \bowtie \text{SUPPLIER} \bowtie \text{ORDER} \bowtie \text{LINEITEM})$$

A feasible plan for this query is depicted in Figure 2. □

[1] Actually, every plan is terminated by an aggregation operator which executes the generalized projection

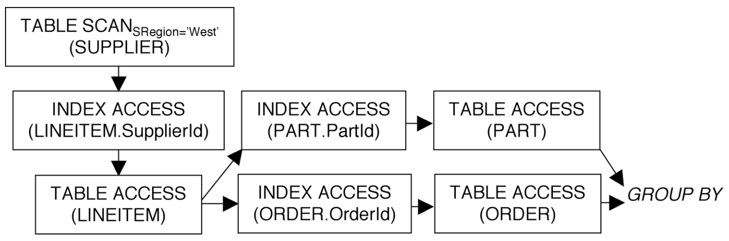
TABLE SCAN SRegion='West' (SUPPLIER)
INDEX ACCESS (LINEITEM.SupplierId)
TABLE ACCESS (LINEITEM)
INDEX ACCESS (PART.PartId)
TABLE ACCESS (PART)
INDEX ACCESS (ORDER.OrderId)
TABLE ACCESS (ORDER)
GROUP BY

Figure 2: Graphical representation of an execution plan.

3.2 Selection of an execution plan

The decision flow for selecting an execution plan for query q is mainly determined by the number of *conditioned* dimension tables, i.e. tables on which a predicate is expressed in q:

- If no conditioned dimension tables are present, the fact table is sequentially scanned (*table scan*) then joined with all the dimension tables involved in q through a nested-loop on their primary key indexes (*index access+table access*).
- If exactly one conditioned dimension table is present, DT_c with condition $Pred$, the plan selection algorithm checks if there is an index allowing to access the fact table 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 the fact table where the foreign key is in the first position). If so, for each tuple of DT_c that satisfies $Pred$, this index and then the fact table are accessed. Otherwise, a hybrid-hash join between DT_c filtered by $Pred$ and the fact table is executed. In both cases, the result is eventually joined with the other dimension tables requested in output.
- If there are two or more conditioned dimension tables, for each of them the algorithm decides how to carry out the join with the fact table (nested-loop if there is an index on the corresponding foreign key in the fact table, hybrid hash otherwise). The join with each dimension table returns the subset of fact table tids whose related dimension table tuples satisfy the local predicates. The tid sets obtained from the different conditioned dimension tables are then intersected, and the resulting tuples of the fact table are accessed.

As to accessing a conditioned dimension table, if no attribute on which a predicate is defined and an index is built exists, then a table scan of the dimension table is executed. Otherwise, all indexes on conditioned attributes are accessed, the tid sets obtained are intersected, possibly further filters on non-indexed attributes are applied, finally the dimension table is accessed.

4 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 [1, 3, 4].

Evaluating the cost of a plan requires a cost function for each single operator appearing in it; the cost of an operator may depend on the cardinality of the output of the previous operator. The cost of the full plan is the sum of the costs of all its operators. Table 1 reports

on the query pattern. This operator is assumed to take no advantage from indexing, thus, it is not considered here.

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
H	height of the B$^+$-tree
NB	number of disk pages to store a bitmap

Table 1: Statistics used by the cost evaluator.

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. Function $sel(Pred)$ returns the fraction of tuples satisfying Boolean predicate $Pred$.

- Table scan on predicate $Pred$:

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

- Index scan on range predicate $Pred$:

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

- Table access on predicate $Pred$:

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

- Index access:

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

- Hash join between $table_1$ and $table_2$:

$$\#HashPartitions = \left\lceil \frac{\#input_1 * t_1}{hb} \right\rceil$$

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

$$\#output = \frac{\#input_1}{NT_1} * \#input_2$$

- Tid intersection between n sets of tids:

$$cost = 0$$

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

5 The index selection algorithm

An attribute $a \in DT$ is said to be *indexable* if at least one query in the workload expresses a condition on a. A prime[2] 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 is executed on FT.

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, the plan generator 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 query q, $QCost(IX, q)$, depends on the table on which IX is built. If IX is built on a dimension table attribute, it is accessed by a scan driven by the selection predicate of q; thus, $QCost(IX, q)$ is the cost of an index scan operation. If IX is built on a prime attribute of the fact table, it is accessed once for each of the ET tuples of the dimension table that satisfy the selection predicate; thus, $QCost(IX, q)$ is equal to ET times the cost of index access. Now, it is possibile to define for IX a *total cost* as its global contribution to the workload cost, computed as the sum of its contributions to the single queries weighted on the query frequencies:

$$TCost(IX) = \sum_q freq(q) \cdot QCost(IX, q)$$

and a *weighted cost* as its size in disk pages 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_{PD}, for indexes on dimension table primary keys; (2) one part, S_{PF}, for indexes on fact table primary keys; and (3) the remaining part for all the other indexes. We assume that only tid-list indexes are built on primary keys and that primary keys of dimension tables are surrogated, so that S_{PD} can be easily calculated. Also the space contribution S_{PF} can be easily computed a priori since the size of each index on the primary key of a fact table only depends on the number of prime attributes, not on their ordering. Finally, the space contribution for other indexes is $S - S_{PD} - S_{PF}$.

The pseudo-code for the index selection algorithm is as follows:

[2]That is, belonging to the primary key.

```
procedure $BuildOptimalIndexSet()$
{   $C = initializeC()$;   // $C$ is the set of candidate indexes
    $O = initializeO()$;   // $O$ is the set of optimal indexes
    $S_{free} = S - S_{PF} - S_{PD}$;   // expressed in disk pages
    while $(\exists IX \in C : sizeP(IX) \leq S_{free})$ do
    {   $IX_{max} = argmax_{\{IX \in C : sizeP(IX) \leq S_{free}\}}\{benefitPerPage(IX, O)\}$;
        $O = O \cup \{IX_{max}\}$;
        $C = C - \{IX_{max}\}$;
        $S_{free} = S_{free} - sizeP(IX_{max})$;   // $sizeP(IX)$ returns the size of $IX$ in disk pages
        if $\exists FT : attr(IX_{max}) \in prime(FT)$ and $\forall a_i \in prime(FT) \exists IX \in O : a_i = attr(IX)$
        // $attr(IX)$ returns the ordered list of the attributes on which $IX$ is built
        // $prime(FT)$ returns the set of prime attributes of $FT$
        {   $IX_{min} = argmin_{\{IX \in O : attr(IX) \in prime(FT)\}}\{decayPerPage(IX)\}$;
            $O = O - \{IX_{min}\}$;
            $S_{free} = S_{free} + sizeP(IX_{min})$;
            $O = O \cup \{multInd(attr(IX_{min}))\}$;
            // given a prime attribute $a \in FT$, $multInd(a)$ returns a (multiple) tid-list index on the
            // primary key of $FT$ whose first attribute is $a$
        }
    }
    for each $FT : O$ does not contain any index on the primary key of $FT$ do
      if $\exists IX \in C : attr(IX) \in prime(FT)$
      {   $IX_{min} = argmin_{\{IX \in C : attr(IX) \in prime(FT)\}}\{benefitPerPage(multInd(attr(IX)), O)\}$;
          $C = C - \{IX_{min}\}$;
          $O = O \cup \{multInd(attr(IX_{min}))\}$;
      }
      else
      {   $a$ = any prime not indexable attribute of $FT$;
          $O = O \cup \{multInd(a)\}$;
      }
}
```

Function *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. Function *initializeO*() initializes the set of the optimal indexes, O: for each dimension table, it inserts in O a tid-list index built on the primary key. Function $benefitPerPage(IX, O)$ returns the relative benefit of IX, estimated as:

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

where $WCost(O)$ is the execution cost, expressed in disk pages, for the whole workload when the indexes in O are built. Given index IX on attribute a, function $decayPerPage(IX)$ 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 attributes others than primary keys, 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 fact table are indexed, one of these indexes must be transformed into a multiple-attribute index on the fact table 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 fact table should be transformed into a multiple index on the fact table primary key, and

VS	S_P	sel = 0.1%	sel = 2%	sel = 10%
100MB	190MB	43.99% (313MB)	1.43% (63MB)	0.01% (63MB)
300MB	198MB	41.06% (319MB)	1.52% (70MB)	0.01% (63MB)
500MB	226MB	41.01% (344MB)	1.40% (70MB)	0.01% (63MB)

Table 2: Relative cost reduction from primary to full indexing in function of the average query selectivity and of the space VS devoted to views. $S_P = S_{PF} + S_{PD}$ is the space used for basic indexing; in parentheses, the space difference between full and primary indexing.

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 fact tables. If, for a given fact table, 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.

6 Experimental tests and conclusions

In this paper we proposed a heuristic approach to the index selection problem starting from a set of star schemes modeling, at the logical level, both primary data and aggregated views. The approach has been tested on the TPC-H benchmark; 20 GPSJ queries inspired to those in the benchmark have been executed varying both their selectivity and the space available for materialized views and indexes. Tests have been executed both in simulation and using the Red Brick 6.0 DBMS. The basic results we obtained are summarized in the following:

- The cost model we adopted turned out to be realistic since it determines costs (expressed as disk pages read) that are constantly about 15% higher than those measured on the DBMS. Such overhead, that does not invalidate the results, can be ascribed to the buffering system that is not modeled in the simulations.
- Indexing may considerably reduce the workload execution cost. Table 2 reports the relative cost reduction, measured as the difference between the cost when only indexes on primary keys are created (*primary indexing*) and the cost when *all* the beneficial indexes are created (*full indexing*), in function of the constraint VS on the disk space available for view materialization, and of the average selectivity of the queries in the workload. This saving, that is up to 44%, strongly depends on the workload selectivity; already for 10% selectivity, it becomes negligible. As to the space for full indexing, Table 2 shows how it reduces for low selectivities since the average utility of indexes decreases.
- The plans produced are sound, in fact (1) the indexes created are always used by the DBMS, and (2) each index created actually reduces the overall execution cost.

An interesting consideration concerns the correlation between the workload selectivity and the best trade-off between the space used for views and that used for indexing. Figure 3 evaluates the workload cost, for a given global space constraint (700 MB), when the ratio between the space constraint on views, VS, and that on indexes, S, is varied. It is apparent that the best trade-off changes significatively depending on the workload selectivity: high selectivities definitely encourage indexing, while at low selectivities view materialization is more convenient.

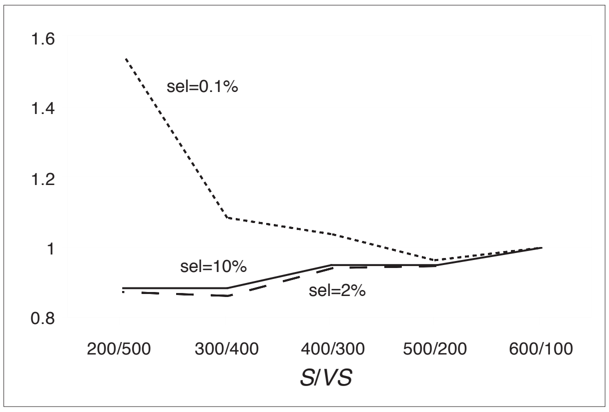

Figure 3: Normalized workload cost with a constant global space constraint.

References

[1] 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, Dept. of CS, University of Arizona, 1998.

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

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

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

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

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

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

[8] 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.

[9] M. Poess and C. Floyd. New tpc benchmarks for decision support and web commerce. *ACM SIGMOD Record*, 29(4), 2000.

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

[11] D. Theodoratos and M. Bouzeghoub. A general framework for the view selection problem for data warehouse design and evolution. In *Proc. DOLAP*, McLean, 2000.