

Supporting Efficient Parametric Search of E-Commerce Data: a Loosely-Coupled Solution

Min Wang, Yuan-Chi Chang, and Sriram Padmanabhan

IBM T. J. Watson Research Center, Hawthorne, NY 10532
{min, yuanchi, srp}@us.ibm.com

Abstract. Electronic commerce is emerging as a major application area for database systems. A large number of e-commerce sites provide electronic product catalogs that allow users to search products of interest. Due to the constant evolution and the high sparsity of e-commerce data, most commercial e-commerce systems use the so-called vertical schema for data storage. However, query processing for data stored using vertical schema is extremely slow because current RDBMS, especially its cost-based query optimizer, is designed to deal with traditional horizontal schema efficiently.

Most e-commerce systems would like to offer advanced parametric search capabilities to their users. However, most searches are expected to be online which means that the query execution should be very fast. RDBMSs require new capabilities and enhancements before they can satisfy the search performance criteria against vertical schema. The tightly-coupled enhancements and additions to a DBMS require considerable amount of work and may take a long time to be accomplished. In this paper, we describe an alternative approach called *SAL*, a Search Assistant Layer that can be implemented outside a database engine to accommodate the urgent need for efficient parametric search on e-commerce data. Our experimental results show that dramatic performance improvement is provided by *SAL* for search queries.

1 Introduction

Electronic commerce is emerging as a major application area for database systems. A large number of e-commerce sites provide electronic product catalogs that allow buyers, sellers, and brokers to search products of interest.

Imagine we are running a marketplace for a big retail chain such as Sears. The e-catalog of this marketplace may contain tens of thousands of products. Each product has its own set of attributes. For example, a “T-shirt” product in woman’s apparel category may be associated with the attribute set $\{size, style, color, price\}$. Another product “TV set” in the electronics category may have a quite different attribute set $\{brand, view_type, signal_type, screen_size, price\}$.

A natural technique for storage of the product information in a relational database system is the *horizontal schema*. Each product is represented as a row in a table, and the columns are the union of the attribute sets across all products. However, this natural approach is not practical due to the following reasons [3]:

- The total number of attributes across all the products can be huge, but current commercial DBMSs do not permit a large number of columns in a table (e.g., both DB2 and Oracle permit only up to 1012 columns in a table).
- Even if a DBMS were to allow huge number of columns in a table, we would have a lot of nulls in most of the fields (e.g., any T-shirt product has a null value in *view_type* column). The large number of null values creates a big storage overhead and increases the size of indexes on these columns.
- Due to the large number of columns and null values, query performance would be very poor since the data records are very wide and only a few columns are used in a query.
- In an electronic marketplace, products traded and sold vary from day to day. We would need frequent altering of the table to accommodate new products. Maintenance and processing of altered tables can be quite expensive in RDBMSs.

An alternative is to use *binary schema* [6, 10]. We create one table for each attribute. Each table contains two columns, one is an attribute column and the other is an *OID* column that ties different fields of a tuple across tables. While there are no null values when using a binary schema, the number of join operations is usually large even when processing simple queries. Hence query performance is a big issue for the binary schema solution. An enhancement is to define a table per product set. This becomes unwieldy since the requirement is for dynamically adding and deleting new products.

Many commercial e-commerce systems use the so-called *vertical schema* design for their e-catalogs. Like the horizontal method, only one table is used. However, this table only has three main attributes: *OID*, *Attr* (for attribute name), and *Value* (for attribute value).¹ Each product is represented as a number of tuples in this table, one for each attribute-value combination and multiple attributes of a product are tied together using the same *OID* across multiple tuples in the table.

Table 1 is an example product table in horizontal representation. The table describes products that consist of one or more attributes in the set $\{A_1, A_2, A_3, A_4, A_5\}$. In the table, null values are denoted by empty spaces. The vertical and binary representations of Table 1 are shown in Figure 1.

The vertical schema has the following advantages when compared to the others:

- *High flexibility*: The schema can handle any number of products and attributes.
- *Ease of schema evolution*: When products are added or deleted, alter or create table operations are not needed. We only need to add/delete tuples that correspond to the attributes of the added/deleted products.
- *Low storage overhead*: In contrast to horizontal schema, the vertical table does not contain null values.

¹ In reality, the *Value* column needs to be extended to accommodate values of different data types. Figure 4 in Section 4 shows one way of doing it.

<i>OID</i>	<i>A</i> ₁	<i>A</i> ₂	<i>A</i> ₃	<i>A</i> ₄	<i>A</i> ₅
1	<i>v</i> ₁	<i>v</i> ₂	<i>v</i> ₃		<i>v</i> ₄
2	<i>v</i> ₅	<i>v</i> ₆	<i>v</i> ₇		<i>v</i> ₈
3	<i>v</i> ₉	<i>v</i> ₁₀			<i>v</i> ₁₁
4			<i>v</i> ₁₂	<i>v</i> ₁₃	<i>v</i> ₁₄
5			<i>v</i> ₁₅		<i>v</i> ₁₆
6				<i>v</i> ₁₇	<i>v</i> ₁₈

Table 1. Horizontal representation.

Due to the above reasons, many commercial e-commerce systems (e.g., IBM Websphere Commerce Server, Ariba Marketplace, I2 Technology) utilize the vertical schema design for their e-catalogs.

Unfortunately, the vertical schema’s flexibility introduces performance challenges for one important activity on e-catalogs, namely, *parametric search queries*. A parametric search query² is a lookup of the e-catalog using specific constraints. Since a product usually contains many attributes, a search query in general is likely to impose several constraints that could be combined in a logical expression using AND/OR or even more sophisticated operators. Writing SQL queries against vertical schema could be cumbersome and error-prone. To solve this problem, Agrawal et al. proposed a method of creating a logical horizontal view on top of a vertical schema [3]. They also presented a set of query rewrite algorithms to convert relational algebra operators against the horizontal view to that against the vertical table. However, the performance of the converted query remains an issue. They show that queries against vertical schema performs no better than against binary schemas in most cases. The main reason for poor performance is that the query optimization and planning techniques in traditional RDBMSs are more suitable for horizontal schema containing well defined and meaningful attributes. Section 2 provides the details.

To achieve good query performance within an RDBMS when using vertical schema, several components need to be enhanced and new capabilities need to be added [3]. The tightly-coupled enhancements and additions to a DBMS require considerable amount of work and may take a long time to be accomplished. In this paper, we present a loosely-coupled solution that can be implemented by building a *Search Assistant Layer* (SAL) outside the database engine. We show that the SAL approach is a viable solution for supporting efficient parametric search against vertical data since SAL effectively provides vertical schema specific query planning and execution capability. It intercepts user search queries and determines the constraints using a *workload learner* component. This component identifies important logical attributes of the vertical schema and their relationships. A self-tuning *histogram builder* is employed to generate single and multi-dimensional histograms for the most important constraint attributes. SAL’s *query planner* consults the histograms before deciding from a number of

² See Section 2 for examples.

<i>OID</i>	Attr	Value
1	A_1	v_1
1	A_2	v_2
1	A_3	v_3
1	A_5	v_4
2	A_1	v_5
2	A_2	v_6
2	A_3	v_7
2	A_5	v_8
3	A_1	v_9
3	A_2	v_{10}
3	A_5	v_{11}
4	A_3	v_{12}
4	A_4	v_{13}
4	A_5	v_{14}
5	A_3	v_{15}
5	A_5	v_{16}
6	A_4	v_{17}
6	A_5	v_{18}

<i>OID</i>	A_1
1	v_1
2	v_5
3	v_9

<i>OID</i>	A_2
1	v_2
2	v_6
3	v_{10}

<i>OID</i>	A_3
1	v_3
2	v_7
4	v_{12}
5	v_{15}

<i>OID</i>	A_4
4	v_{13}
6	v_{17}

<i>OID</i>	A_5
1	v_4
2	v_8
3	v_{11}
4	v_{14}
5	v_{16}
6	v_{18}

(a) Vertical representation (b) Binary representation

Fig. 1. Vertical and binary representations.

query execution choices. Finally, a simple *query processor* layer module is used to combine results of sub-queries as well as to perform post processing. We have evaluated SAL for search queries against realistic e-catalogs. Our performance results validate that SAL is required to provide good performance for parametric search queries against vertical schema.

The rest of the paper is organized as follows. In the next section, we analyze the reasons for poor query performance when data are stored using vertical schema. In Section 3, we describe the SAL approach in more detail. We present our experimental results in Section 4 and draw conclusions in Section 5.

2 Why Are Search Queries Against Vertical Schema So Slow?

In e-commerce applications, users usually search for desired products by providing bounds (constraints) on attribute values. For example, a user may be interested in finding all the T-shirts that satisfy the following constraints: *size* = 'M', *color* = 'Purple', and $\$45 \leq \textit{price} \leq \50 . If we store all T-shirt products in a table T-shirt(*OID*, *size*, *style*, *color*, *price*), such a query is expressed easily against the horizontal schema as shown below.

Q_1 : `SELECT OID`

```

FROM T-shirt
WHERE size = 'M' AND color = 'Purple' AND $45 ≤ price ≤ $50

```

A generic search query against a horizontal schema H has the following format:

```

SELECT  $OID$ 
FROM  $H$ 
WHERE ( $A_{i_1}$  not null) AND (bound on  $A_{i_1}$ ) AND
      ( $A_{i_2}$  not null) AND (bound on  $A_{i_2}$ ) AND
      ...
      ( $A_{i_k}$  not null) AND (bound on  $A_{i_k}$ )

```

In this paper, we only consider the above type of queries.

Most RDBMSs transform the logical query plan represented by the query into a *physical query plan* which represents the operations, the method of performing the operations, and the order of processing the different operations [7]. In generating a physical query plan, the RDBMS uses a query optimizer module which considers many different physical plans that are derived from the logical plan, and evaluates or estimates their costs. After this evaluation, also called as *cost-based optimization*, it picks the physical query plan with the least estimated cost and passes that to the query-execution engine. The efficiency of the query execution depends mainly upon the accuracy of the cost estimation.

Usually a query optimizer estimates cost of a query plan based on statistics collected from the base data. The query optimizer builds a *histogram* for each attribute during an offline phase. The histogram contains the statistical information about the distribution of the corresponding attribute and is stored in a database system catalog [16, 13, 15, 11].

Subsequently, when Q_1 (the T-shirt query) is issued, the query optimizer can quickly estimate the (approximate) selectivity³ of the three predicates in the WHERE clause based on the three histograms in the catalog. For example, if the (estimated) selectivity of the predicate $color = 'Purple'$ is 0.01%, and is much lower than those for the other two predicates, it is very likely that the query optimizer will choose an index scan using the index on the $color$ column (assuming the existence of such index) before applying the other two predicates.

Suppose all products are stored in a vertical table V . Query Q_1 corresponds to the following query which assumes a single *Value* column. (As mentioned earlier, the *Value* column needs to be extended to accommodate values of different data types. Figure 4 in Section 4 shows one approach.)

```

 $Q_2$ :  SELECT  $OID$ 
        FROM  $V$ 
        WHERE  $Attr = 'size'$  AND  $Value = 'M'$ 
        INTERSECT
        SELECT  $OID$ 
        FROM  $V$ 

```

³ The selectivity of a query predicate is the number of rows in the table that satisfy this predicate divided by the table size.

```

WHERE Attr = 'color' AND Value = 'Purple'
INTERSECT
SELECT OID
FROM V
WHERE Attr = 'price' AND $45 ≤ Value ≤ $50

```

Note that the three logical constraints in Q_1 are transformed into six constraints in Q_2 . Among these six constraints, three of them are selecting specific attribute names (i.e., *size*, *color*, and *price*) while the other three specify values (i.e., 'M', 'Purple', and between 45 and 50). When we use the vertical schema, the *Value* column contains the values for *all attributes across all product categories*. Similarly, the *Attr* column contains the string names of all product attributes. The query optimizer still collects the statistical information about the data distribution on these two columns and constructs histograms. However, the information in the histograms is a poor approximation of the distribution of the all attribute names and values for all products. Since the total number of such logical attributes is huge, the histograms become very misleading when we use it to estimate the selectivity of a constraint (e.g., *color* = 'Purple') on a specific product category (e.g., T-shirt). Additionally, the optimizer must choose methods to combine the results of the individual sets of constraints in order to generate the final result. Usually, predicate selectivities are combined assuming *independence*. However, in parametric search queries, the constraints are usually mutually dependent as is the case in our example above. Thus, the optimizer generates poor cardinality estimates and this translates to poor choices on access methods as well as combining operations. In our study, we found that a traditional RDBMS usually generates a physical plan that involves several poorly chosen access methods and join algorithms on the vertical table for a query that is similar to Q_2 . Since the vertical table is large, these query plans result in very long execution time. Essentially, the optimizer is handicapped and can not choose a better plan for search queries against vertical schema.

In an online search environment, any long query execution time is not acceptable. This is the familiar conundrum facing many e-commerce systems today. They would like to provide advanced parametric search capability but are stymied by the inadequate response time for such queries. Users want both the power of advanced search as well as the immediate online response time!

3 SAL: The Search Assistant Layer

We identified the main cause of poor query performance as the inadequacy of the cardinality estimation techniques and statistics of RDBMSs. The cost estimation inside RDBMS is misled by vertical schema due to aggregated statistics of heterogeneous attributes (columns) from different products. There are two possible approaches to solve the performance problem. The first approach is to modify the indexing and query optimization components of the RDBMS to account for vertical schema. This tightly integrated solution requires database kernel change. While it is the likely long term solution, it does not solve the

immediate problem at hand. In this paper, we take an alternative approach of building a Search Assistant Layer (SAL) outside the RDBMS. SAL collects the statistics of the vertical table in a more intelligent way and uses it to patrol and remedy the queries that might lead to suboptimal performance. For example, even though the *Value* column is a single column in the vertical table, we should be aware of the fact that this column is actually a combination (union) of many attributes and these attributes should be treated separately. Moreover, if certain attribute names (e.g., *view_type*, *signal_type*) are associated with certain product category (e.g., TV) and are usually queried together, the optimizer should choose to build a multi-dimensional histogram to capture the joint distribution of the multiple attributes. Our loosely integrated solution does not require any change to the database engine and provides a good short-term solution to the vertical schema search problem.

SAL is composed of four components:

- A *workload learner* that learns from the query workload and decides attributes sets on which histograms should be built.
- A *histogram builder* that builds the selected external histograms outside the RDBMS and uses query feedback to refine the histograms in an online manner.
- A *query planner* that estimates the selectivity of the predicates in a query based on the external histograms and chooses the most efficient physical plan. The query planner uses a rule-based optimizer approach driven by a deep understanding of the vertical schema parameters.
- A *query processor* that executes the SAL query plan and retrieves the query results from the underlying tables in an RDBMS.

Figure 2 shows SAL’s architecture.

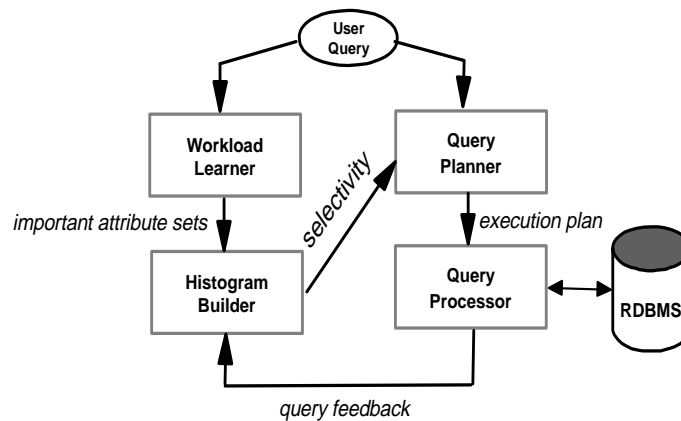


Fig. 2. SAL’s architecture

Next, we elaborate on the four components of SAL.

3.1 Workload Learner

Query optimizer in an RDBMS has traditionally built single-attribute histograms to compute the selectivity of queries [16, 13, 15]. For queries that involve multiple attributes, most RDBMSs assume *attribute value independence*, which usually leads to significant inaccuracy in selectivity estimation [14]. A better alternative is to construct *multidimensional histogram* over multiple attributes [12, 14, 11]. A lot of work has been reported in the literature on designing different types of histograms [16, 13–15, 12, 11].

In practice, an RDBMS can only allocate very limited amount of space to store the histograms. In our approach, we build histograms in SAL. The storage limitation applies to SAL as well even though it is less strict compared to RDBMS histograms. The main reason here is not to conserve memory but rather to conserve query optimization time. The histograms must be small enough to be processed efficiently in a short period of time during query optimization phase.

A typical vertical schema usually contains a large number of logical attribute names and values. Given the vertical schema, we face the problem of *identifying logical attributes that are most important for query processing so that we can effectively use the limited amount of storage*. This requirement can only be addressed by analyzing the search query workload.

Even though this is a fundamental problem in histogram construction and query optimization, there is only very limited amount of past work in this area. Most previous work has focused on identifying the optimal histogram for a *given* attribute or attribute set using fixed amount of storage space. The most relevant work is on global optimization of histograms [9]. In [9], Jadagish et al. presented the idea of global optimization of histograms, i.e., single-attribute histograms for a set of attributes are optimized collectively so as to minimize the overall error in using the histograms. However, the more important problem of choosing the attributes (attribute sets) for constructing histograms remains unsolved.

A naive approach is to build histograms for all possible attribute sets. Since the total number of attributes across all product categories is usual huge, this naive approach is not practical at all. Moreover, a typical search query usually only involves attributes that correspond to a specific product or category. For example, it is very likely that a query contains constraints on attributes *view_type* and *signal_type* at the same time, but it is almost impossible that a query involves both *view_type* (of a TV set) and *style* (of a T-shirt). Hence building histograms for attributes that belong to different product categories becomes a waste of resource. Even for attributes that correspond to the same product category, some attributes are more frequently queried than others. Obviously, building histograms for attributes that frequently appear in queries is more valuable. Also, if a set of attributes is frequently being queried together, we should construct a multidimensional histogram in order to provide better cardinality estimates.

In our method, we propose to build a workload learner to accomplish the task of identifying the important attribute sets on which histograms should be built. The main features of the workload learner module are:

1. When a query is issued, SAL will parse and record the attributes involved in the query constraints as a *transaction entry* in a *query log*. For example, query Q_1 in Section 2 corresponds to transaction $\{size, color, price\}$ in the query log.
2. When there are more than N (a tunable parameter) transactions in the query log, we process it to discover all the *maximally important attribute sets*.
3. The maximally important attribute sets are passed to the histogram builder.

The critical step is to discover all the important attribute sets. Intuitively, an attribute set is important if and only if it appears in a lot of queries. We use the association rule discovery [2, 4, 17, 8, 19] to guide the important attribute set discovery.

Let Q be the query log and A be a set of attributes. The *support* (or occurrence frequency) of A is the number of transactions in Q that A appears ⁴ divided by the total number of transactions in Q . An attribute set A is called an *important attribute set* if its support is no less than a predefined *minimum support threshold*. A *maximally important attribute set* is an important attribute set such that none of its supersets is an important attribute set.

Table 2 shows an example query log Q of size 6. For example, the support of attribute set $\{A_1, A_2, A_3\}$ is $1/2$ and the support of attribute set $\{A_4, A_5\}$ is $1/3$. If the minimum support threshold is $1/2$, the important attribute sets are $\{A_1\}$, $\{A_2\}$, $\{A_3\}$, $\{A_4\}$, $\{A_1, A_2\}$, $\{A_1, A_3\}$, $\{A_2, A_3\}$, and $\{A_1, A_2, A_3\}$. Among them, attribute sets $\{A_1, A_2, A_3\}$ and $\{A_4\}$ are maximally important attribute sets.

QID	Transaction
Q_1	$\{A_1, A_2, A_3, A_4\}$
Q_2	$\{A_1, A_2, A_3\}$
Q_3	$\{A_1, A_2, A_3\}$
Q_4	$\{A_4, A_5, A_6\}$
Q_5	$\{A_4, A_5\}$
Q_6	$\{A_4, A_6\}$

Table 2. Query Log.

The above definition of (maximally) important attribute set directly corresponds to that of large (frequent) item set in association rule mining studied in the literature [2, 4, 17, 8, 19]. We can hence apply any standard association mining algorithm to the query log to find all the maximally important attribute sets.

After the important attribute set discovery is done, the query log may keep growing when more queries are issued, and we need to update the important

⁴ An attribute set A appears in a transaction t if A is a subset of t (i.e., $A \subseteq t$).

attribute sets to capture the distribution of the changed workload. This can be done in an incremental fashion as suggested in [5, 18].

3.2 Histogram Builder

Suppose we obtain n maximally important attribute sets through the workload learner while the amount of storage allocated for external histograms is M bytes. The histogram builder constructs n (multidimensional) histograms under the storage constraint.

The first question for the histogram builder is *how to allocate the storage amongst the histograms?* In our method, we use a simple heuristic based on two intuitions:

- Even though all the n attribute sets are important, it is possible to rank them based on their support. The ones with higher support should be allocated more space during histogram construction.
- An attribute set containing more attributes should be allocated more space because histograms with higher dimensionality usually need more space to achieve reasonable accuracy.

More precisely, suppose the dimensionality of the i th ($1 \leq i \leq n$) maximally important attribute set is d_i and its support is s_i , we use the following formula to compute the amount of storage that should be allocated to its corresponding histogram:

$$M_i = M \times \frac{\alpha s_i + \beta d_i}{\sum_{1 \leq j \leq n} (\alpha s_j + \beta d_j)},$$

where α and β are both positive constants.

The second set of problems faced by the histogram builder is:

- What type of histograms to construct?
- How to dynamically maintain the histograms as the underlying base data changes (i.e., new products are inserted and old products are deleted) over time?

In our current working prototype, we built multidimensional equi-depth histograms using the self-tuning technique presented in [1]. Even though similar in structure to traditional equi-depth histogram, the self-tuning histogram infers data distribution not by examining the data but by using query feedback from the query execution engine. The feedback is used to progressively refine the histogram. Besides low building cost, the self-tuning histogram also has the nice property of low maintenance cost since it is refined through query feedback when the underlying data distribution changes. For details about self-tuning histogram, please refer to [1].

3.3 Query Planner

The query planner is the crucial module of SAL that is responsible for choosing an efficient processing strategy for the common parametric search queries. Its effectiveness is a result of its intimate knowledge (gained from histograms described previously) of the distribution statistics of the logical attributes and attribute sets in the vertical schema. The query planner generates a higher level logical query processing plan for the original query. Such a logical plan might comprise of one or more logical subqueries that are submitted to the underlying RDBMS and then combined in the SAL query processor module.

By examining and experimenting with a large number of queries and data distributions, we concluded that queries against product categories with hundreds and thousands of products in a large catalog (relative to buffer pool size) can create suboptimal query plans if processed directly by an RDBMS engine. Product category is a natural way to group similar products and conduct parametric searches. Our conclusion is expected since estimation error appears in the statistical deviation on the distribution of subsets of records in the database. These queries need to be intercepted and rewritten following one or more strategies described below. On the other hand, we have also observed that there is no need to alter queries against either very small product categories (tens of products) or extremely large categories (compared to catalog size). The former case is driven by filtering the category and the latter has a smaller estimation error due to its large relative size.

These observations result in three rules for determining a SAL query plan. These options are *Direct Search* (DS), *Nested Invocation* (NI), and *Supervised N-way Join* (SN).

Direct Search The first option is the Direct Search (DS) approach that leaves the query optimization and execution to the database engine. As mentioned above, SAL will choose this strategy for certain types of queries and certain product categories.

Nested Invocation The second option is referred as the Nested Invocation (NI) method. An opportunity to apply this method arises when the query planner knows that one or more constraints in the query has very small selectivity that can quickly reduce the number of *OIDs* in consideration. It is then best to retrieve the much smaller group of *OIDs* and verify if the other constraints are satisfied by them. For example, in query Q_2 , suppose *color* = 'Purple' is the most selective constraint. SAL can issue a simple query to retrieve *OIDs* satisfying *Attr* = '*color*' AND *Value* = 'Purple'. Suppose the resulting *OIDs* are 1011, 1022, and 1033. The resulting *OIDs* are then used in an IN clause to drive the following query:

```
SELECT OID
FROM V
WHERE Attr = 'size' AND Value = 'M'
```

```

INTERSECT
SELECT OID
FROM V
WHERE Attr = 'price' AND $45 ≤ Value ≤ $50
INTERSECT
SELECT OID
FROM V
WHERE OID IN (1011, 1022, and 1033).

```

Supervised N-way Join The third option is the Supervised N-way join (SN) method. When the NI method is not applicable, it is often faster to retrieve *OIDs* satisfying individual constraints and then find their intersection set in memory. For example, to process query Q_2 , three separate queries are issued and their *OIDs* are retrieved to an in-memory data structure. Figure 3 shows the separate queries. The *OIDs* are joined in-memory inside SAL and a followup query is issued to retrieve other select list items. While the SN method suffers from the cost of reading data from database, the method often delivers surprisingly good performance with fast CPU and large memory in modern computers.

SELECT <i>OID</i>	SELECT <i>OID</i>	SELECT <i>OID</i>
FROM <i>V</i>	FROM <i>V</i>	FROM <i>V</i>
WHERE <i>Attr</i> = 'size'	WHERE <i>Attr</i> ='color'	WHERE <i>Attr</i> ='price'
AND <i>Value</i> ='M'	AND <i>Value</i> ='Purple'	AND 45<= <i>Value</i> <= 50

Fig. 3. Independent Subqueries in Supervised N-Way Join

Besides the three options discussed above, there are other ways to improve query performance. The choice is a balance between database execution time and in-memory processing time. In-memory processing time can be significant if a very large number of *OIDs* are read out of database but are discarded in later steps.

Our loosely-integrated solution relies on additional information to spot “troubled” queries and makes a decision among DS, NI and SN methods. The query planner needs to know the number of products in the queried category, the selectivity of each query constraint, and the expected number of products satisfying all query constraints. A less selective constraint on a large category or large result set is always a concern as estimation error may aggravate search performance.

Since it is expensive to collect the needed information from database, an external histogram becomes the choice of our solution. The histogram builder and its associated workload learner determine the sets of product attributes in a category to be selected for histogram learning. Details are specified in Section 3.1 and Section 3.2

The query planner makes a series of decisions to detect “troubled” queries and remedy potential performance problems. It is observed that in the vertical schema, the cost (query execution time) of the complete database query is highly nonlinear and difficult to characterize analytically. Hence, SAL’s query planner

is designed to be rule-based. The current implementation follows three simple rules, each of which corresponds to one option:

1. DS: For categories with very small number of products (e.g., a category containing no more than 100 *OIDs*), do not change the query and let database execute the complete query.
2. NI: For a query with one or more highly selective constraints (e.g., result is no more than 1000 *OIDs*), execute the most selective constraint(s) first and retrieve the qualified *OIDs*. Formulate the query with the rest of the constraints and place the retrieved *OIDs* in an IN clause to drive the join.
3. SN: For a query with less selective constraints (e.g., each constraint generates more than 1000 *OIDs*), break up the query into several queries by applying one logical constraint in each and execute them one by one. The resulting *OID* sets are joined in memory inside SAL.

3.4 Query Processor

With the query plan assigned, SAL's query processor executes the plan and retrieves results requested. Since the query processor may need to find the intersection set of multiple returned results, we implemented the intersection method using the Java utility *hashset*. Hashset is a set class with *hashtable* function associated with the key. Using hashset allows the method to quickly decide whether an *OID* exists in the set or not.

In addition, the query processor is commissioned to retrieve attribute descriptions, values, and other properties associated with an *OID* in the final result.

4 Experimental Results

In this section, we compare the performance of SAL with that of executing parametric searches on vertical schema using a traditional RDBMS directly. We refer the latter as DS method.

All our experiments were run on a 933MHz single processor Intel Pentium machine with 1GB DRAM. The operation system was Windows 2000 and the database system used was DB2 UDB 7.1. The machine had a single 17GB SCSI drive. The buffer pool size was set to 168MB and the prefetch size was set to 256KB.

The schemas used in the experiments are typically seen in electronic catalog datastore. First, a table CATE_PROD is used to store the relationship between product categories (CATEGORY_ID) and product definitions (CATENTRY_ID). Under each product definition (CATENTRY_ID), multiple attributes may be registered in the ATTRIBUTE table. The ATTRIBUTE table maps the attribute name (NAME) and product definition (CATENTRY_ID) to attribute identifier (ATTRIBUTE_ID), which are used in the vertical table. The vertical table ATTRVALUE exhibits the 3-ary form of (OID, ATTRIBUTE_ID, ATTRIBUTE_VALUE). An OID is the reference identifier of a product that can be

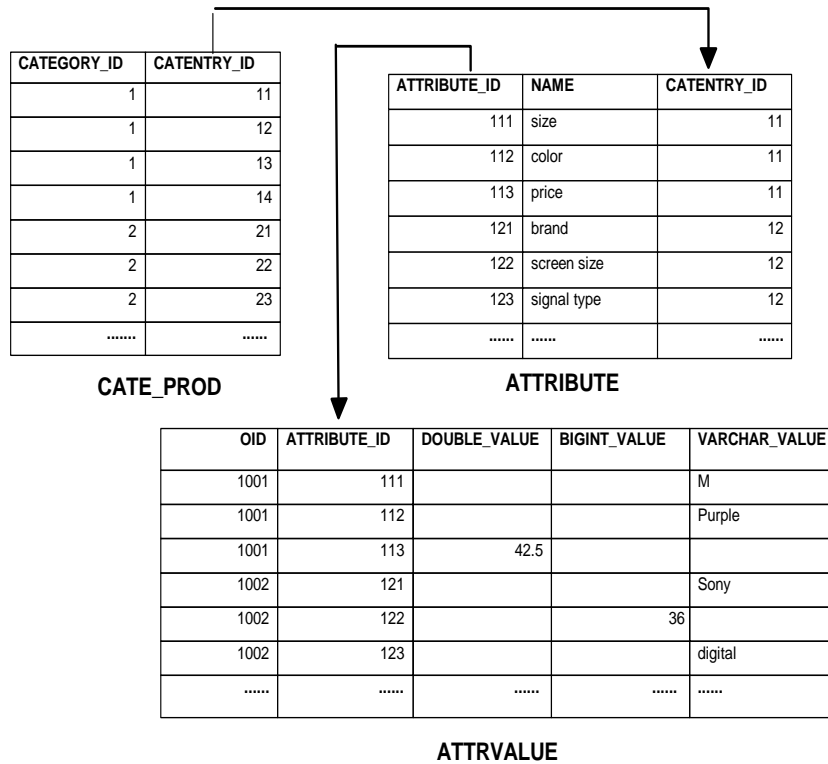


Fig. 4. An example e-catalog

bought, which must be associated with a product definition through the set of ATTRIBUTE_IDS. The attribute values are represented in the database native format such as VARCHAR, BIGINT, and DOUBLE to facilitate search, and the three data types are represented as three columns in the vertical table ATTRVALUE. The relations among CATE_PROD, ATTRIBUTE, and ATTRVALUE are described using an example shown in Figure 4

We populated 300,000 products and their information were put into the catalog using a Java tool which simulates real data set. The products are evenly divided among 60 product definitions, four of which are associated to the search category. In other words, the category contains 20,000 products. Each product has 10 attributes of VARCHAR, BIGINT, or DOUBLE data types. Attribute values are discrete and were generated randomly and independently. The cardinality of the ATTRVALUE is on the order of 3 million rows. Multi-attribute indices on ATTRVALUE were built on the 3-ary form and detailed distribution statistics was collected by the DBMS before we ran any query. An index on CATE_PROD was also created to link categories and product definitions.

Queries were generated randomly by a Java tool which independently decides value ranges of query constraints. A query consists of 2–6 logical constraints. (Each logical constraint corresponds to two constraints in the query against the

vertical schema.) We summarized the main results with benchmark numbers of the following two methods for a set of representative queries only.

- DS: Queries are executed by DBMS directly.
- SAL: Queries are executed using SAL.

We measured the end-to-end execution time for each methods. For SAL, the execution time included execution time within DBMS and in-memory processing time. Queries are issued through JDBC driver offered by the database vendor.

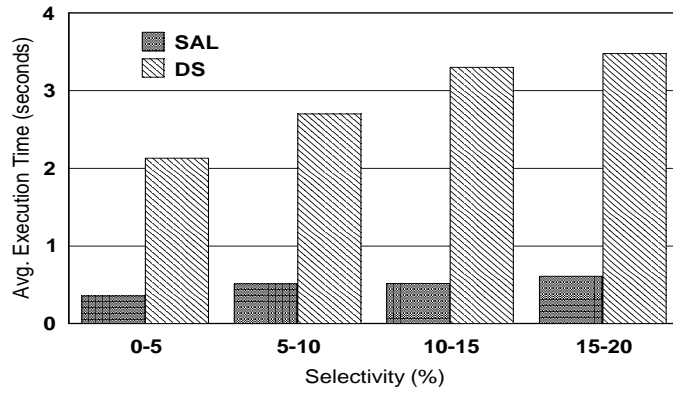
Figure 5 shows the performance of SAL versus that of DS for three typical query sets. Each query set contains 1000 queries with two, three, or four logical constraints. The selectivities of the queries vary from 0% to 20%. We measured the average execution time for queries in different selectivity ranges. We can see from the figures that for any query that contains 2–4 logical constraints, the average query execution time for SAL was always in the range of 0.3 second to 1.5 seconds while that of DS was in the range of 2.2 seconds to 116 seconds.

We also observed that when the number of constraints in the query increases, the gap between the two methods became huge. The execution time for SAL only increased slightly while DS took at least several minutes to execute any query containing more than three logical constraints. Note that in online e-catalog search applications, any query respond time beyond several seconds usually becomes unacceptable for customers.

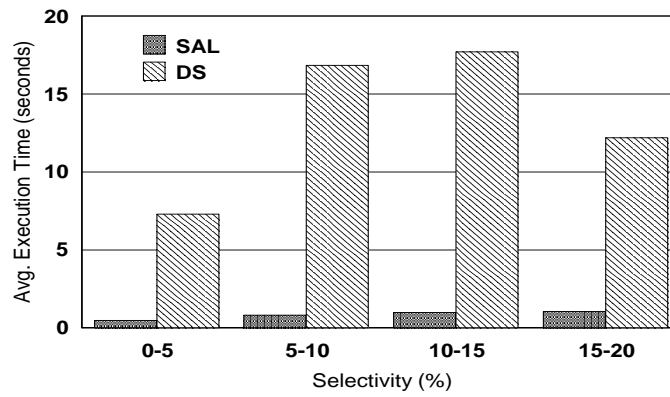
5 Conclusions

In this paper, we have proposed and designed a loosely-coupled Search Assistant Layer , SAL, for enabling parametric search queries against electronic catalogs stored using vertical schema. In the past, e-commerce systems did not support ad-hoc parametric search against such catalogs because the response time was usually too long. SAL maintains histogram based distribution statistics for the logical attributes appearing in search constraints. Multidimensional histograms are constructed for logical attributes that appear jointly. SAL’s query planner consults these histograms and uses a number of rules to decide on a logical query execution strategy. The query planner could choose Direct Search, Nested Invocation, or Supervised N-way Join methods. We have implemented and evaluated SAL in IBM’s Websphere Commerce Suite product. Our results show that SAL produces significant performance improvement for ad-hoc parametric searches and facilitates online searches against e-commerce catalogs greatly.

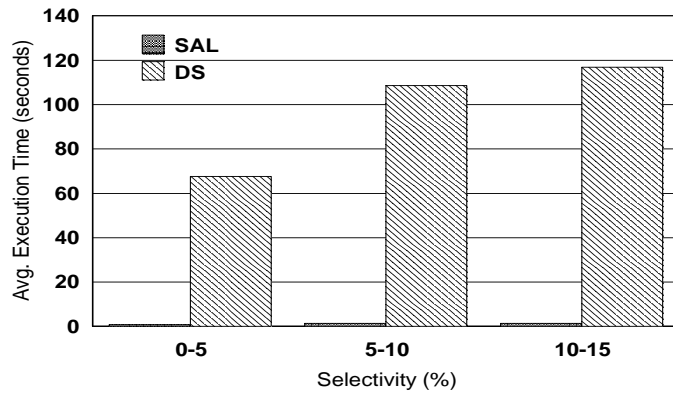
At present, SAL has several limitations. The query planner could use a more rigorous formula to estimate cost of a query plan. However, this will not be easy even if cost estimation from the database optimizer can be accessed. We have observed that the cost estimates reported are quite often far off the real execution time. Secondly, the current implementation only addresses constraints that are combined using AND operators (conjunction). A general logic expression with hierarchies of ANDs and ORs is more difficult to optimize. Finally, the query



(a) For queries with two logical constraints



(b) For queries with three logical constraints



(c) For queries with four logical constraints

Fig. 5. Average query execution time

planner does not address queries involving both vertical and horizontal schema. This multi-model search problem is our next focus.

References

1. A. Abounaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, pages 181–192. ACM Press, 1999.
2. R. Agrawal, T. Imielinsk, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, 1993.
3. R. Agrawal, A. Somani, and Y. Xu. Storage and querying of e-commerce data. In *VLDB'01, Proceedings of 27rd International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*. Morgan Kaufmann, 2001.
4. R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 1994 International Conference on Very Large Databases*, pages 487–499, 1994.
5. D. W.-L. Cheung, J. Han, V. Ng, and C. Y. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. In S. Y. W. Su, editor, *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana, USA*, pages 106–114. IEEE Computer Society, 1996.
6. G. P. Copeland and S. Khoshafian. A decomposition storage model. In S. B. Navathe, editor, *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, Austin, Texas, May 28-31, 1985*, pages 268–279. ACM Press, 1985.
7. H. Garcia-Molina, J. Ullman, and J. Widom. *Database System Implementation*. Prentice-Hall, 2000.
8. J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In W. Chen, J. F. Naughton, and P. A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, volume 29, pages 1–12. ACM, 2000.
9. H. V. Jagadish, H. Jin, B. C. Ooi, and K.-L. Tan. Global optimization of histograms. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, May 21-24, 2001, Santa Barbara, CA, USA*, pages 223–234, 2001.
10. S. Khoshafian, G. P. Copeland, T. Jagodis, H. Boral, and P. Valduriez. A query processing strategy for the decomposed storage model. In *Proceedings of the Third International Conference on Data Engineering, February 3-5, 1987, Los Angeles, California, USA*, pages 636–643. IEEE Computer Society, 1987.
11. Y. Matias, J. S. Vitter, and M. Wang. Wavelet-based histograms for selectivity estimation. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 448–459, Seattle, WA, June 1998.
12. M. Muralikrishna and D. J. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 28–36, 1988.

13. G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 256–276, 1984.
14. V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proceedings of the 1997 International Conference on Very Large Databases*, Athens, Greece, August 1997.
15. V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. Shekita. Improved histograms for selectivity estimation of range predicates. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, Montreal, Canada, May 1996.
16. P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, pages 23–34, 1979.
17. R. Srikant and R. Agrawal. Mining quantitative association rules in large relational tables. In H. V. Jagadish and I. S. Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 1–12. ACM Press, 1996.
18. S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka. An efficient algorithm for the incremental updation of association rules in large databases. In *KDD 1997, Proceedings of the Third International Conference on Knowledge Discovery and Data Mining, August 14-17, 1997, Newport Beach, California, USA*, pages 263–266. AAAI Press, 1997.
19. K. Wang, Y. He, and J. Han. Mining frequent itemsets using support constraints. In A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 43–52. Morgan Kaufmann, 2000.