Integrating Association Rule Mining with Relational Database Systems: Alternatives and Implications

Sunita Sarawagi
sunita@almaden.ibm.com

Shiby Thomas *
sthomas@cise.ufl.edu

Rakesh Agrawal
ragrawal@almaden.ibm.com

IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120

Abstract

Data mining on large data warehouses is becoming increasingly important. In support of this trend, we consider a spectrum of architectural alternatives for coupling mining with database systems. These alternatives include: loose-coupling through a SQL cursor interface; encapsulation of a mining algorithm in a stored procedure; caching the data to a file system on-the-fly and mining; tight-coupling using primarily user-defined functions; and SQL implementations for processing in the DBMS. We comprehensively study the option of expressing the mining algorithm in the form of SQL queries using Association rule mining as a case in point. We consider four options in SQL-92 and six options in SQL enhanced with object-relational extensions (SQL-OR). Our evaluation of the different architectural alternatives shows that from a performance perspective, the Cache-Mine option is superior, although the performance of the SQL-OR option is within a factor of two. Both the Cache-Mine and the SQL-OR approaches incur a higher storage penalty than the loose-coupling approach which performance-wise is a factor of 3 to 4 worse than Cache-Mine. The SQL-92 implementations were too slow to qualify as a competitive option. We also compare these alternatives on the basis of qualitative factors like automatic parallelization, development ease, portability and inter-operability.

1 Introduction

An ever increasing number of organizations are installing large data warehouses using relational database technology. There is a huge demand for mining nuggets of knowledge from these data warehouses.

The initial research on data mining was concentrated on defining new mining operations and developing algorithms for them. Most early mining systems were developed largely on file systems and specialized data structures and buffer management strategies were devised for each algorithm. Coupling with database systems was at best loose, and access to data in a DBMS was provided through an ODBC or SQL cursor interface (e.g. [14, 1, 9, 12]).

Researchers of late have started to focus on issues related to integrating mining with databases. There have been language proposals to extend SQL to support mining operators. For instance, the query language DMQL [9] extends SQL with a collection of operators for mining characteristic rules, discriminant rules, classification rules, association rules, etc. The M-SQL language [13] extends SQL with a special unified operator *Mine* to generate and query a whole set of propositional rules. Another example is the *mine rule* [17] operator for a generalized version of the association rule discovery problem. Query flocks for association rule mining using a generate-and-test model has been proposed in [25].

The issue of tightly coupling a mining algorithm with a relational database system from the systems point of view was addressed in [5]. This proposal makes use of user-defined functions (UDFs) in SQL statements to selectively push parts of the computation into the database system. The objective was to avoid one-at-a-time record retrieval from the database, saving both the copying and process context switching costs. The SETM algorithm [10] for finding association rules was expressed in the form of SQL queries. However, as shown in [3], SETM is not efficient and there are no results reported on running it against a relational DBMS. Recently, the problem of expressing the association rules algorithm in SQL has been explored in [20]. We discuss this work later in the paper.

1.1 Goal

This paper is an attempt to understand implications of various architectural alternatives for coupling data mining with relational database systems. In particular, we are interested in studying how competitive can a mining computation expressed in SQL be compared to a specialized implementation of the same mining operation.

There are several potential advantages of a SQL implementation. One can make use of the database indexing and query processing capabilities thereby leveraging on more than a decade of effort spent in making these systems robust, portable, scalable, and concurrent. One can also exploit the underlying SQL parallelization, particularly in an SMP environment. The DBMS support for checkpointing and space management can be valuable for long-running mining algorithms

The architecture we have in mind is schematically shown in Figure 1. We visualize that the desired mining operation will be expressed in some extension of SQL or a graphi-

* Current affiliation: Dept. of Computer & Information Science & Engineering, University of Florida, Gainesville

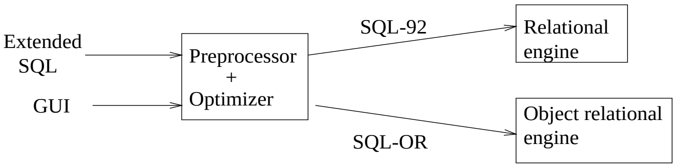


Figure 1: SQL architecture for mining in a DBMS

cal language. A preprocessor will generate appropriate SQL translation for this operation. We consider translations that can be executed on a SQL-92 [16] relational engine, as well as translations that require some of the newer object-relational capabilities being designed for SQL [15]. Specifically, we assume availability of blobs, user-defined functions, and table functions [19].

We compare the performance of the above **SQL architecture** with the following alternatives:

Read directly from DBMS: Data is read tuple by tuple from the DBMS to the mining kernel using a cursor interface. Data is never copied to a file system. We consider two variations of this approach. One is the **loose-coupling** approach where the DBMS runs in a different address space from the mining process. This is the approach followed by most existing mining systems. A potential problem with this approach is the high context switching cost between the DBMS and the mining process [5]. In spite of the block-read optimization present in many systems (e.g. Oracle [18], DB2 [7]) where a block of tuples is read at a time, the performance could suffer. The second is the **stored-procedure** approach where the mining algorithm is encapsulated as a stored procedure [7] that runs in the same address space as the DBMS. The main advantage of both these approaches is greater programming flexibility and no extra storage requirement. The mined results are stored back into the DBMS.

Cache-mine: This option is a variation of the Stored-procedure approach where after reading the entire data once from the DBMS, the mining algorithm temporarily caches the relevant data in a lookaside buffer on a local disk. The cached data could be transformed to a format that enables efficient future accesses. The cached data is discarded when the execution completes. This method has all the advantages of the stored procedure approach plus it promises to have better performance. The disadvantage is that it requires additional disk space for caching. Note that the permanent data continues to be managed by the DBMS.

User-defined function (UDF): The mining algorithm is expressed as a collection of user-defined functions (UDFs) [7] that are appropriately placed in SQL data scan queries. Most of the processing happens in the UDF and the DBMS is used primarily to provide tuples to the UDFs. Little use is made of the query processing capability of the DBMS. The UDFs are run in the unfenced mode (same address space as the database). Such an implementation was presented in [5]. The main attraction of this method over Stored-procedure is performance since passing tuples to a stored procedure is slower than passing it to a UDF. Otherwise, the processing happens in almost the same manner as in the stored procedure case. The main disadvantage is the development cost since the entire mining algorithm has to be written as UDFs involving significant code rewrites [5]. This option can be viewed as an extreme case of the SQL-OR approach where UDFs do all the processing.

1.2 Methodology

We do both quantitative and qualitative comparisons of the architectures stated above with respect to the problem of discovering Association rules [2] against IBM DB2 Universal Server [11].

For the `loose-coupling` and Stored-procedure architectures, we use the implementation of the Apriori algorithm [3] for finding association rules provided with the IBM data mining product, Intelligent Miner [14]. For the Cache-Mine architecture, we used the "space" option provided in Intelligent Miner that caches the data in a binary format after the first pass. For the UDF architecture, we use the UDF implementation of the Apriori algorithm described in [5]. For the SQL-architecture, we consider two classes of implementations: one uses only the features supported in SQL-92 and the other uses object-relational extensions to SQL (henceforth referred to as SQL-OR). We consider four different implementations in the first case and six in the second. These implementations differ in the way they exploit different features of SQL. We compare the performance of these different approaches using four real-life datasets. We also use synthetic datasets at various points to better understand the behavior of different algorithms.

1.3 Paper Layout

The rest of the paper is organized as follows. In Section 2, we cover background material. In Section 3, we present the overview of the SQL implementations. In Sections 4 and 5, we elaborate on different ways of doing the support counting phase of Associations in SQL — Section 4 presents SQL-92 implementations and Section 5 gives implementations in SQL-OR. In Section 6 we present a qualitative and quantitative comparison of the different architectural alternatives. We present conclusions in Section 7. This paper is an abbreviated version of the full paper that appears in [21].

2 Background

2.1 Association Rules

Given a set of transactions, where each transaction is a set of items, an association rule [2] is an expression $X \rightarrow Y$, where X and Y are sets of items. The intuitive meaning of such a rule is that the transactions that contain the items in X tend to also contain the items in Y. An example of such a rule might be that "30% of transactions that contain beer also contain diapers; 2% of all transactions contain both these items". Here 30% is called the *confidence* of the rule, and 2% the *support* of the rule. The problem of mining association rules is to find *all* rules that satisfy a user-specified minimum support and minimum confidence.

The association rule mining problem can be decomposed into two subproblems [2]:

- Find all combinations of items, called *frequent* itemsets, whose support is greater than minimum support.
- Use the frequent itemsets to generate the desired rules. The idea is that if, say, $ABCD$ and AB are frequent, then the rule $AB \rightarrow CD$ holds if the ratio of support$(ABCD)$ to support(AB) is at least as large as the minimum confidence. Note that the rule will have minimum support because $ABCD$ is frequent.

The first part on generation of frequent itemsets is the most time-consuming part and we concentrate on this part in the paper. In [21] we also discuss rule generation.

2.2 Apriori Algorithm

We use the Apriori algorithm [3] as the basis for our presentation. There are recent proposals for improving the Apriori algorithm by reducing the number of data passes [24, 6]. They all have the same basic dataflow structure as the Apriori algorithm. Our goal in this work is to understand how best to integrate this basic structure within a database system. In [21], we discuss how our conclusions extrapolate to these algorithms.

The Apriori algorithm for finding frequent itemsets makes multiple passes over the data. In the kth pass it finds all itemsets having k items called the k-itemsets. Each pass consists of two phases. Let F_k represent the set of frequent k-itemsets, and C_k the set of candidate k-itemsets (potentially frequent itemsets). First, is the **candidate generation** phase where the set of all frequent $(k-1)$-itemsets, F_{k-1}, found in the $(k-1)$th pass, is used to generate the candidate itemsets C_k. The candidate generation procedure ensures that C_k is a superset of the set of all frequent k-itemsets. A specialized in-memory hash-tree data structure is used to store C_k. Then, data is scanned in the **support counting** phase. For each transaction, the candidates in C_k contained in the transaction are determined using the hash-tree data structure and their support count is incremented. At the end of the pass, C_k is examined to determine which of the candidates are frequent, yielding F_k. The algorithm terminates when F_k or C_{k+1} becomes empty.

2.3 Input format

The transaction table T has two column attributes: transaction identifier *(tid)* and item identifier *(item)*. The number of items per tid is variable and unknown during table creation time. Thus, alternatives such as [20], where all items of a tid appear as different columns of a single tuple, may not be practical. Often the number of items per transaction can be more than the maximum number of columns that the DBMS supports. For instance, for one of our real-life datasets the maximum number of items per transaction is 872 and for another it is 700. In contrast, the corresponding average number of items per transaction is only 9.6 and 4.4 respectively.

3 Associations in SQL

In Section 3.1 we present the candidate generation procedure in SQL and in Section 3.2 we present the support counting procedure.

3.1 Candidate generation in SQL

Each pass k of the Apriori algorithm first generates a candidate itemset set C_k from frequent itemsets F_{k-1} of the previous pass.

In the *join* step, a superset of the candidate itemsets C_k is generated by joining F_{k-1} with itself:

$$
\begin{array}{ll}
\textsf{insert into } C_k \textsf{ select} & I_1.item_1, \ldots, I_1.item_{k-1}, I_2.item_{k-1} \\
\textsf{from} & F_{k-1}\ I_1, F_{k-1}\ I_2 \\
\textsf{where} & I_1.item_1 = I_2.item_1 \textsf{ and} \\
 & \vdots \\
 & I_1.item_{k-2} = I_2.item_{k-2} \textsf{ and} \\
 & I_1.item_{k-1} < I_2.item_{k-1}
\end{array}
$$

For example, let F_3 be {{1 2 3}, {1 2 4}, {1 3 4}, {1 3 5}, {2 3 4}}. After the join step, C_4 will be {{1 2 3 4}, {1 3 4 5}}. Next, in the *prune* step, all itemsets $c \in C_k$, where some $(k-1)$-subset of c is not in F_{k-1}, are deleted. Continuing with the example above, the prune step will delete the itemset {1 3 4 5} because the subset {1 4 5} is not in F_3. We will then be left with only {1 2 3 4} in C_4.

We can perform the prune step in the same SQL statement as the join step by writing it as a k-way join as shown in Figure 2. A k-way join is used since for any k-itemset there are k subsets of length $(k-1)$ for which F_{k-1} needs to be checked for membership. The join predicates on I_1 and I_2 remain the same. After the join between I_1 and I_2 we get a k itemset consisting of $(I_1.item_1, \ldots, I_1.item_{k-1}, I_2.item_{k-1})$. For this itemset, two of its $(k-1)$-length subsets are already known to be frequent since it was generated from two itemsets in F_{k-1}. We check the remaining $k-2$ subsets using additional joins. The predicates for these joins are enumerated by skipping one item at a time from the k-itemset as follows: We first skip $item_1$ and check if subset $(I_1.item_2, \ldots, I_1.item_{k-1}, I_2.item_{k-1})$ belongs to F_{k-1} as shown by the join with I_3 in the figure. In general, for a join with I_r, we skip item $r-2$. We construct a primary index on $(item_1, \ldots, item_{k-1})$ of F_{k-1} to efficiently process these k-way joins using index probes.

C_k need not always be materialized before the counting phase. Instead, the candidate generation can be pipelined with the subsequent SQL queries used for support counting.

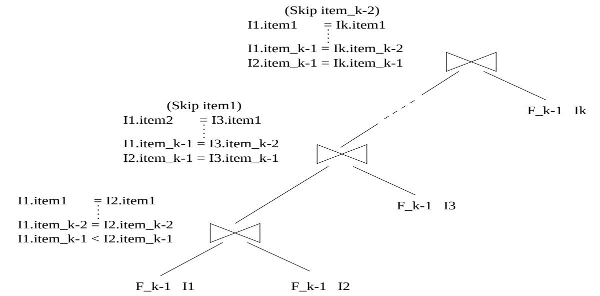


Figure 2: Candidate generation for any k

3.2 Counting support to find frequent itemsets

This is the most time-consuming part of the association rules algorithm. We use the candidate itemsets C_k and the data table T to count the support of the itemsets in C_k. We consider two different categories of SQL implementations:

(A) The first one is based purely on SQL-92. We discuss four approaches in this category in Section 4.

(B) The second utilizes object-relational extensions like UDFs, BLOBs (Binary large objects) and table functions. *Table functions* [19] are virtual tables associated with a user defined function which generate tuples on the fly. They have pre-defined schemas like any other table. The function associated with a table function can be implemented as a UDF. Thus, table functions can be viewed as UDFs that return a collection of tuples instead of scalar values.

We discuss six approaches in this category in Section 5. UDFs in this approach are light weight and do not require extensive memory allocations and coding unlike the UDF architectural option (Section 1.1).

4 Support counting using SQL-92

We studied four approaches in this category – we present the two better ones here. The other two are discussed in [21].

4.1 K-way joins

In each pass k, we join the candidate itemsets C_k with k transaction tables T and follow it up with a group by on the itemsets as shown in Figure 3. The figure 3 also shows a tree diagram of the query. These tree diagrams are not to be confused with the plan trees that could look quite different.

insert into F_k select $item_1, \ldots item_k$, count(*)
from C_k, T t_1, ... T t_k
where t_1.item $= C_k.item_1$ and
$\vdots$
t_k.item $= C_k.item_k$ and
t_1.tid $= t_2$.tid and
$\vdots$
t_{k-1}.tid $= t_k$.tid
group by $item_1, item_2 \ldots item_k$
having count(*) $>$:minsup

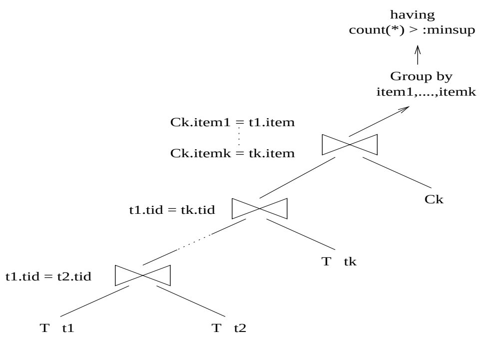


Figure 3: Support Counting by K-way join

This SQL computation, when merged with the candidate generation step, is similar to the one proposed in [25] as a possible mechanism to implement query flocks.

For pass-2 we use a special optimization where instead of materializing C_2, we replace it with the 2-way joins between the F_1s as shown in the candidate generation phase in section 3.1. This saves the cost of materializing C_2 and also provides early filtering of the Ts based on F_1 instead of the larger C_2 which is almost a cartesian product of the F_1s. In contrast, for other passes corresponding to $k > 2$, C_k could be smaller than F_{k-1} because of the prune step.

4.2 Subquery-based

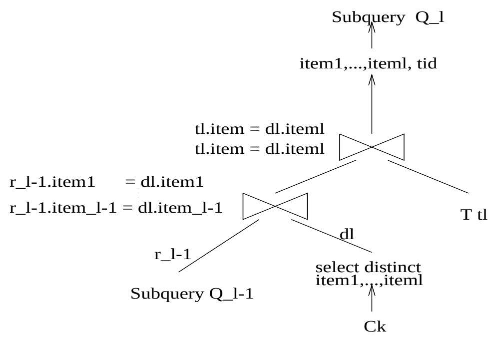
This approach makes use of common prefixes between the itemsets in C_k to reduce the amount of work done during support counting. The support counting phase is split into a cascade of k subqueries. The l-th subquery Q_l (see Figure 4) finds all tids that match the distinct itemsets formed by the first l columns of C_k (call it d_l). The output of Q_l is joined with T and d_{l+1} (the distinct itemsets formed by the first $l+1$ columns of C_k) to get Q_{l+1}. The final output is obtained by a group-by on the k items to count support as above. Note that the final "select distinct" operation on the C_k when $l = k$ is not necessary.

For pass-2 the special optimization of the KwayJoin approach is used.

insert into F_k select $item_1, \ldots, item_k$, count(*)
from (Subquery Q_k) t
group by $item_1, item_2 \ldots item_k$
having count(*) $>$:minsup

Subquery Q_l (for any l between 1 and k):
select $item_1, \ldots item_l$, tid
from T t_l, (Subquery Q_{l-1}) as r_{l-1},
(select distinct $item_1 \ldots item_l$ from C_k) as d_l
where $r_{l-1}.item_1 = d_l.item_1$ and ... and
$r_{l-1}.item_{l-1} = d_l.item_{l-1}$ and
$r_{l-1}.tid = t_l.tid$ and
$t_l.item = d_l.item_l$

Subquery Q_0: No subquery Q_0.



Tree diagram for Subquery Q_l

Figure 4: Support counting using subqueries

4.3 Performance comparison of SQL-92 approaches

We now briefly compare the different SQL-92 approaches; detailed results are available in [21].

Our experiments were performed on Version 5 of IBM DB2 Universal Server installed on a RS/6000 Model 140 with a 200 MHz CPU, 256 MB main memory and a 9 GB disk with a measured transfer rate of 8 MB/sec.

We selected four real-life datasets obtained from mail-order companies and retail stores for the experiments. These datasets differ in the values of parameters like the number of (tid,item) pairs, number of transactions (tids), number of items and the average number of items per transaction. Table 1 summarizes characteristics of these datasets.

Datasets	# Records in millions (R)	# Transactions in millions (T)	# Items in thousands (I)	Avg. #items (R/T)
Dataset-A	2.5	0.57	85	4.4
Dataset-B	7.5	2.5	15.8	2.62
Dataset-C	6.6	0.21	15.8	31
Dataset-D	14	1.44	480	9.62

Table 1: Description of different real-life datasets.

We found that the best SQL-92 approach was the Subquery approach, which was often more than an order of magnitude better than the other three approaches. However, this approach was comparable to the Loose-coupling approach only in some cases whereas for several others it did not complete even after taking ten times more time than the Loose-coupling approach.

The important conclusion we drew from this study, therefore is that implementations based on pure SQL-92 are too slow to be considered an alternative to the existing Loose-coupling approach.

5 Support counting using SQL with object-relational extensions

In this section, we study approaches that use object-relational features in SQL to improve performance. We first consider an approach we call GatherJoin and its three variants in Section 5.1. Next we present a very different approach called Vertical in Section 5.2. We do not discuss the sixth approach called SBF based on SQL-bodied functions because of its inferior performance (see [21]). For each approach, we also outline a cost-based analysis of the execution time to choose between these different approaches. In Section 5.3 we present performance comparisons.

5.1 GatherJoin

The GatherJoin approach (see Figure 5) generates all possible k-item combinations of items contained in a transaction, joins them with the candidate table C_k, and counts the support of the itemsets by grouping the join result. It uses two table functions `Gather` and `Comb-K`. The data table T is scanned in the *(tid, item)* order and passed to the table function `Gather`, which collects all the items of a transaction in memory and outputs a record for each transaction. Each record consists of two attributes: the *tid* and *item-list* which is a collection of all items in a field of type VARCHAR or BLOB. The output of `Gather` is passed to another table function `Comb-K` which returns all k-item combinations formed out of the items of a transaction. A record output by `Comb-K` has k attributes $T\_itm_1, \ldots, T\_itm_k$, which can be used to probe into the C_k table. An index is constructed on all the items of C_k to make the probe efficient.

This approach is analogous to the KwayJoin approach except that we have replaced the k-way self join of T with the table functions `Gather` and `Comb-K`. These table functions are easy to code and do not require a large amount of memory. It is also possible to merge them into a single table function `GatherComb-K`, which is what we did in our implementation. Note that the `Gather` function is not required when the data is already in a horizontal format where each tid is followed by a collection of all its items.

Special pass 2 optimization: For $k = 2$, the 2-candidate set C_2 is simply a join of F_1 with itself. Therefore, we can optimize the pass 2 by replacing the join with C_2 by a join with F_1 *before* the table function (see Figure 6). The table function now gets only frequent items and generates significantly fewer 2-item combinations. We apply this optimization to other passes too. However, unlike pass 2 we still have to do the final join with C_k and therefore the benefit is not as significant.

insert into F_k select $item_1, \ldots, item_k$, count(*)
from C_k,
    (select $t_2.T\_itm_1, \ldots, t_2.T\_itm_k$ from T,
    table (Gather(T.tid, T.item)) as t_1,
    table (Comb-K(t_1.tid, t_1.item-list)) as t_2)
where $t_2.T\_itm_1 = C_k.item_1$ and
    $\vdots$
    $t_2.T\_itm_k = C_k.item_k$
group by $C_k.item_1, \ldots, C_k.item_k$
having count(*) > :minsup

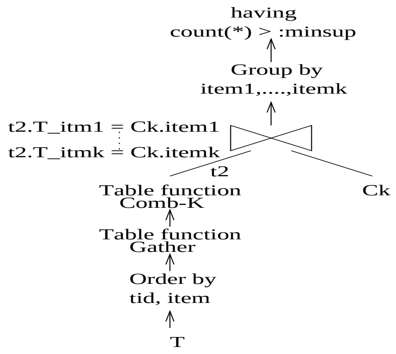


Figure 5: Support Counting by GatherJoin

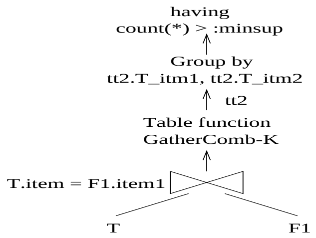


Figure 6: Support Counting by GatherJoin in the second pass

5.1.1 Variations of GatherJoin approach

GatherCount: One variation of the GatherJoin approach for pass two is the **GatherCount** approach where we perform the group-by inside the table function `GatherComb-2`. We will refer to this extended table function as `Gather-Cnt`. The candidate 2-itemset C_2 is represented as a two dimensional array (as suggested in [3]) inside function `Gather-Cnt`. Instead of outputting the 2-item combinations, the function uses the combinations to directly update support counts in memory and outputs only the frequent 2-itemsets, F_2 and their support after the last transaction.

The attraction of this otion is the absence of the outer grouping. The UDF code is small since it only needs to maintain a 2D array. We could apply the same trick for subsequent passes but the coding becomes considerably more complicated because of the need to maintain hash-tables to index the C_ks. The disadvantage of this approach is that it can require a large amount of memory to store C_2. If enough memory is not available, C_2 needs to be partitioned and the process has to be repeated for each partition. Another problem with this approach is that it cannot be automatically parallelized.

GatherPrune: A problem with the **GatherJoin** approach is the high cost of joining the large number of item combinations with C_k. We can push the join with C_k inside the table function and thus reduce the number of such combinations. C_k is converted to a BLOB and passed as an argument to the table function.

The cost of passing the BLOB for every tuple of R can be high. In general, we can reduce the parameter passing cost by using a smaller Blob that only approximates the real C_k. The trade-off is increased cost for other parts notably grouping because not as many combinations are filtered. A problem with this approach is the increased coding complexity of the table function.

Horizontal: This is another variation of GatherJoin that first uses the `Gather` function to transform the data to the horizontal format but is otherwise similar to the GatherJoin approach. Rajamani et al. [20] propose finding associations using a similar approach augmented with some pruning based on a variation of the GatherPrune approach. Their results assume that the data is already in a horizontal format which is often not true in practice. They report that their SQL implementation is two to six times slower than a UDF implementation.

R	number of records in the input transaction table
T	number of transactions
N	avg. number of items per transaction $= \frac{R}{T}$
F_1	number of frequent items
$S(C)$	sum of support of each itemset in set C
R_f	number of records out of R involving frequent items $= S(F_1)$
N_f	average number of frequent items per transaction $= \frac{R_f}{T}$
C_k	number of candidate k-itemsets
$C(N,k)$	number of combinations of size k possible out of a set of size n: $= \frac{n!}{k!(n-k)!}$
s_k	cost of generating a k item combination using table function `Comb-k`
group(n,m)	cost of grouping n records out of which m are distinct
join(n,m,r)	cost of joining two relations of size n and m to get a result of size r
blob(n)	cost of passing a BLOB of size n integers as an argument

Table 2: Notations used for cost analysis of different approaches

5.1.2 Cost analysis of GatherJoin and its variants

The relative performance of these variants depends on a number of data characteristics like the number of items, total number of transactions, average length of a transaction etc. We express the costs in each pass in terms of parameters that are known or can be estimated after the candidate generation step of each pass. The purpose of this analysis is to help us choose between the different options. Therefore, instead of including *all* I/O and CPU costs, we include only those terms that help us distinguish between different options. We use the notations of Table 2 in the cost analysis.

The cost of GatherJoin includes the cost of generating k-item combinations, joining with C_k and grouping to count the support. The number of k-item combinations generated, T_k is $C(N,k)*T$. Join with C_k filters out the non-candidate item combinations. The size of the join result is the sum of the support of all the candidates denoted by $S(C_k)$. The actual value of the support of a candidate itemset will be known only after the support counting phase. However, we approximate it to the minimum of the support of all its $(k-1)$-subsets in F_{k-1}. The total cost of the GatherJoin approach is:

$$T_k * s_k + \text{join}(T_k, C_k, S(C_k)) + \text{group}(S(C_k), C_k),$$

$$\text{where } T_k = C(N,k) * T$$

The above cost formula needs to be modified to reflect the special optimization of joining with F_1 to consider only frequent items. We need a new term join(R, F_1, R_f) and need to change the formula for T_k to include only frequent items N_f instead of N.

For the second pass, we do not need the outer join with C_k. The total cost of GatherJoin in the second pass is:

$$\text{join}(R, F_1, R_f) + T_2 * s_2 + \text{group}(T_2, C_2),$$

$$\text{where } T_2 = C(N_f, 2) * T \approx \frac{N_f^2 * T}{2}$$

Cost of GatherCount in the second pass is similar to that for basic GatherJoin except for the final grouping cost:

$$\text{join}(R, F_1, R_f) + \text{group\_internal}(T_2, C_2) + F_2 * s_2$$

In this formula, "group_internal" denotes the cost of doing the support counting inside the table function.

Cost formulas for the GatherPrune and Horizontal approaches can be derived similarly and appear in [21].

5.2 Vertical

We first transform the data table into a vertical format by creating for each item a BLOB containing all tids that contain that item (Tid-list creation phase) and then count the support of itemsets by merging together these tid-lists (support counting phase). This approach is similar to the approaches in [26]. For creating the Tid-lists we use a table function `Gather`. This is the same as the `Gather` function in **GatherJoin** except that we create the tid-list for each frequent item. The data table T is scanned in the (item,tid) order and passed to the function `Gather`. The function collects the *tids* of all tuples of T with the same item in memory and outputs a (item, tid-list) tuple for items that meet the minimum support criterion. The tid-lists are represented as BLOBs and stored in a new TidTable with attributes (item, tid-list).

In the support counting phase, for each itemset in C_k we want to collect the tid-lists of all k items and use a UDF to count the number of *tids* in the intersection of these k lists. The tids are in the same sorted order in all the tid-lists and therefore the intersection can be done efficiently by a single pass of the k lists. This step can be improved by decomposing the intersect operation to share these operations across itemsets having common prefixes as follows.

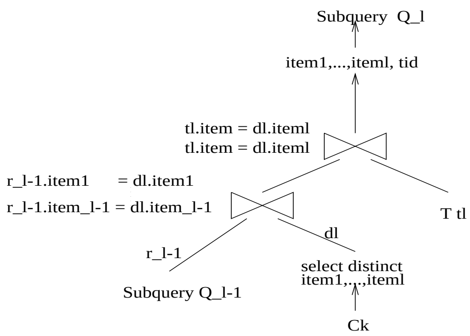
We first select distinct $(item_1, item_2)$ pairs from C_k. For each distinct pair we first perform the intersect operation to get a new result-tidlist, then find distinct triples $(item_1, item_2, item_3)$ from C_k with the same first two items, intersect result-tidlist with tid-list for $item_3$ for each triple and continue with $item_4$ and so on until all k tid-lists per itemset are intersected. This approach is analogous to the Subquery approach presented for SQL-92.

The above sequence of operations can be written as a single SQL query for any k as shown in Figure 7. The final intersect operation can be merged with the count operation to return a count instead of the tid-list — we do not show this optimization in the query of Figure 7 for simplicity.

insert into F_k select $item_1, \ldots, item_k$, count(tid-list) as cnt
  from (**Subquery** Q_k) t where cnt $>$:minsup

Subquery Q_l (for any l between 2 and k):
  select $item_1, \ldots item_l$,
    Intersect(r_{l-1}.tid-list,t_l.tid-list) as tid-list
  from TidTable t_l, (**Subquery** Q_{l-1}) as r_{l-1},
    (select distinct $item_1 \ldots item_l$ from C_k) as d_l
  where $r_{l-1}.item_1 = d_l.item_1$ and ... and
    $r_{l-1}.item_{l-1} = d_l.item_{l-1}$and
    $t_l.item = d_l.item_l$

Subquery Q_1: (select * from TidTable)



Tree diagram for Subquery Q_l

Figure 7: Support counting using UDF

Special pass 2 optimization: For pass 2 we need not generate C_2 and join the TidTables with C_2. Instead, we perform a self-join on the TidTable using predicate $t_1.item < t_2.item$.

insert into F_k select $t_1.item, t_2.item$, cnt
from (select $item_1, item_2$,
    CountIntersect(t_1.tid-list, t_2.tid-list) as cnt
  from TidTable t_1, TidTable t_2
  where $t_1.item < t2.item$) as t
where cnt $>$:minsup

5.2.1 Cost analysis

The cost of the Vertical approach during support counting is dominated by the cost of invoking the UDFs and intersecting the tid-lists. The UDF is first called for each distinct item pair in C_k, then for each distinct item triple and so on. Let d_j^k be the number of distinct j item tuples in C_k Then the number of UDF invocations is $\sum_{j=2}^{k} d_j^k$. In each invocation two BLOBs of tid-list are passed as arguments. The UDF intersects the tid-lists by a merge pass and hence the cost is proportional to 2 * average length of a tid-list. The average length of a tid-list can be approximated to $\frac{R_f}{F_1}$. Note that with each intersect the tid-list keeps shrinking. However, we ignore such effects for simplicity.

The total cost of the Vertical approach is:

$$(\sum_{j=2}^{k} d_j^k) * \left(2 * Blob(\frac{R_f}{F_1}) + \text{Intersect}(\frac{2R_f}{F_1})\right)$$

In the above formula Intersect(n) denotes the cost of intersecting two tid-lists with a combined size of n. We are not including the join costs in this analysis because it accounted for only a small fraction of the total cost.

5.3 Performance comparison of SQL-OR approaches

We studied the performance of six SQL-OR approaches using the datasets summarized in Table 1. Figure 8 shows the results for only four approaches: GatherJoin, GatherCount, GatherPrune and Vertical. For the other two approaches (Horizontal and SBF) the running times were so large that we had to abort the runs in many cases. The reason why the Horizontal approach was significantly worse than the GatherJoin approach was the time to transform the data to the horizontal format.

We first concentrate on the overall comparison between the different approaches. Then we will compare the approaches based on how they perform in each pass of the algorithm.

The Vertical approach has the best overall performance and it is sometimes more than an order of magnitude better than the other three approaches.

The majority of the time of the Vertical approach is spent in transforming the data to the Vertical format in most cases (shown as "prep" in figure 8). The vertical representation is like an index on the *item* attribute. If we think of this time as a one-time activity like index building then performance looks even better. The time to transform the data to the Vertical format was much smaller than the time for the horizontal format although both formats write almost the same amount of data. The reason is the difference in the *number* of records written. The number of frequent items is often two to three orders of magnitude smaller than the number of transactions.

Between GatherJoin and GatherPrune, neither strictly dominates the other. The special pass-2 optimization in GatherJoin had a big impact on performance. With this optimization, for `Dataset-B` with support 0.1%, the running time for pass 2 was reduced from 5.2 hours to 10 minutes.

When we compare these approaches based on time spent in each pass no single approach emerges as "the best" for all passes of the with datasets.

For pass three onwards, Vertical is often two or more orders of magnitude better than the other approaches. For higher passes, the performance degrades dramatically for GatherJoin, because the table function `Gather-Comb-K` generates a large number of combinations. GatherPrune is better than GatherJoin for third and later passes. For pass 2 GatherPrune is worse because the overhead of passing a large object as an argument dominates cost.

The Vertical approach sometimes spends too much time in the second pass. In some of these cases the GatherJoin approach was better in the second pass (for instance for low support values of `Dataset-B`) whereas in other cases (for instance, `Dataset-C` with minimum support 0.25%) GatherCount was the only good option. In the latter case, both GatherPrune and GatherJoin did not complete after more than six hours for pass 2. Further, they caused a storage overflow error because of the large size of the intermediate results to be sorted. We had to divide the dataset into four

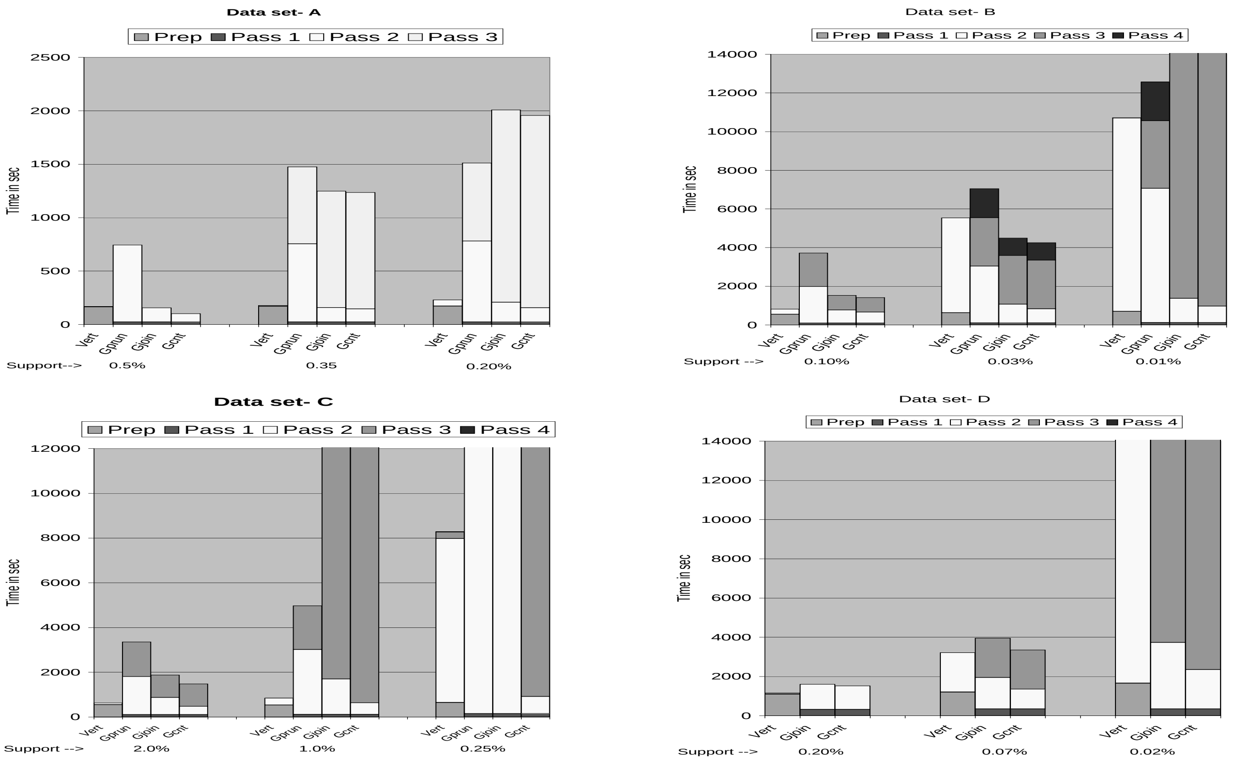


Figure 8: Comparison of four SQL-OR approaches: Vertical, GatherPrune, GatherJoin and GatherCount on four datasets for different support values. The time taken is broken down by each pass and an initial "prep" stage where any one-time data transformation cost is included.

equal parts and ran the second pass independently on each partition to avoid this problem.

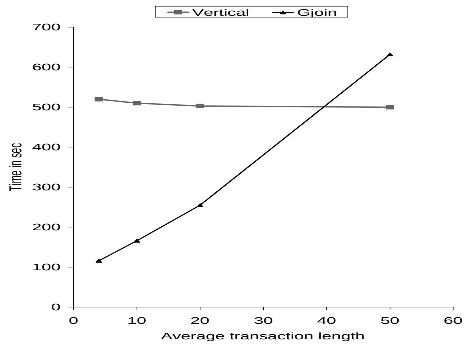


Figure 9: Effect of increasing transaction length

Two factors that affect the choice amongst the Vertical, GatherJoin and GatherCount approaches in different passes and pass 2 in particular are: number of frequent items (F_1) and the average number of frequent items per transaction (N_f). From Figure 8 we notice that as the value of the support is decreased for each dataset causing the size of F_1 to increase, the performance of pass 2 of the Vertical approach degrades rapidly. This trend is also clear from our cost formulae. The cost of the Vertical approach increases quadratically with F_1. GatherJoin depends more critically on the number of frequent items per transaction. For `Dataset-B` even when the size of F_1 increases by a factor of 10, the value of N_f remains close to 2, therefore the time taken by GatherJoin does not increase as much. However, for `Dataset-C` the size of N_f increases from 3.2 to 10 as the support is decreased from 2.0% to 0.25% causing GatherJoin to deteriorate rapidly. From the cost formula for GatherJoin we notice that the total time for pass 2 increases almost quadratically with N_f.

We validated this observation further by running experiments on synthetic datasets for varying values of the number of frequent items per transaction. We used the synthetic dataset generator described in [3] for this purpose. We varied the transaction length, the number of transactions and the support values while keeping the total number of records and the number of frequent items fixed. In Figure 9 we show the total time spent in pass 2 of the Vertical and GatherJoin approaches. As the number of items per transaction (transaction length) increases, the cost of Vertical remains almost unchanged whereas the cost of GatherJoin increases.

5.4 Final hybrid approach

The previous performance section helps us draw the following conclusions: Overall, the Vertical approach is the best option especially for higher passes. When the size of the candidate itemsets is too large, the performance of the Vertical approach could suffer. In such cases, GatherJoin is a good option as long as the number of frequent items per transaction (N_f) is not too large. When N_f *is* large GatherCount may be the only good option even though it may not easily parallelize.

The hybrid scheme chooses the best of the three approaches GatherJoin, GatherCount and Vertical for each pass based on the cost estimates outlined in Sections 5.1.2 and

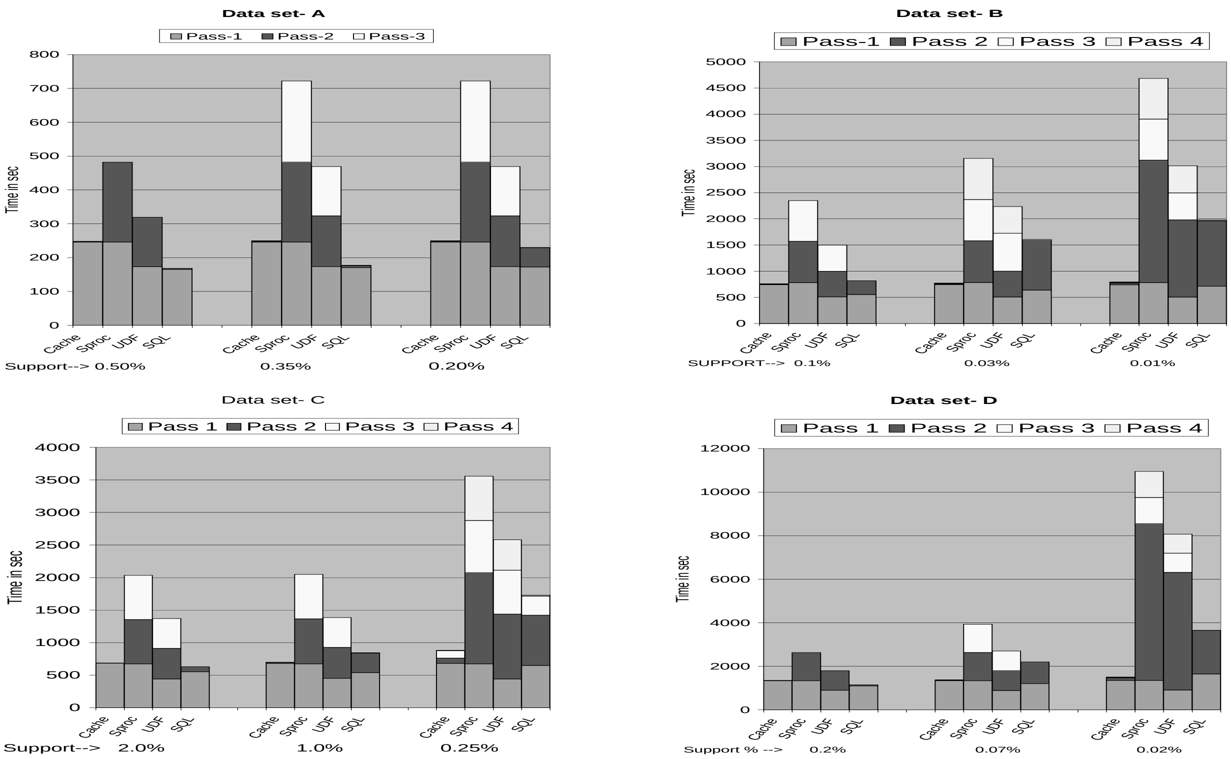


Figure 10: Comparison of four architectures: Cache-Mine, Stored-procedure, UDF and SQL-OR on four real-life datasets. Loose-coupling is similar to Stored-procedure. For each dataset three different support values are used. The total time is broken down by the time spent in each pass.

5.2.1. The parameter values used for the estimation are available at the end of the previous pass. In Section 6 we plot the final running time for the different datasets based on this hybrid approach.

6 Architecture comparisons

In this section our goal is to compare the five alternatives: Loose-coupling, Stored-procedure, Cache-Mine, UDF, and the best SQL implementation.

The Loose-coupling, Stored-procedure and Cache-Mine implementations are derived from IBM's Intelligent Miner [14] code as discussed in Section 1.2. The only difference between Loose-coupling and Stored-procedure approaches is that the former in run in a seperate address space whereas the latter is run in the same address space as the database server. This difference did not impact performance much on DB2, therefore we will often be basing our comparisons on the Stored-procedure approach. For the UDF-architecture, we use the UDF implementation of the Apriori algorithm described in [5]. In this implementation, first a UDF is used to initialize state and allocate memory for candidate itemsets. Next, for each pass a collection of UDFs are used for generating candidates, counting support, and checking for termination. These UDFs access the initially allocated memory, address of which is passed around in BLOBs. Candidate generation creates the in-memory hash-trees of candidates. This happens entirely in the UDF without any involvement of the DBMS. During support counting, the data table is scanned sequentially and for each tuple a UDF is used for updating the counts on the memory resident hashtree.

6.1 Timing comparison

In Figure 10, we show the performance of Cache-Mine, Stored-procedure, UDF and the hybrid SQL-OR implementation for the datasets in Table 1. We do not show the times for the Loose-coupling option because its performance was very close to the Stored-procedure option.

We can make the following observations:

- Cache-Mine has the best or close to the best performance in all cases. 80-90% of its total time is spent in the first pass where data is accessed from the DBMS and cached in the file system. Compared to the SQL approach this approach is a factor of 0.8 to 2 times faster.

- The Stored-procedure approach is the worst. The difference between Cache-Mine and Stored-procedure is directly related to the number of passes. For instance, for `Dataset-A` the number of passes increases from two to three when decreasing support from 0.5% to 0.35% causing the time taken to increase from two to three times. The time spent in each pass for Stored-procedure is the same except when the algorithm makes multiple passes over the data since all candidates could not fit in memory together. This happens for the lowest support values of `Dataset-B`, `Dataset-C` and `Dataset-D`. Time taken by Stored-procedure can be expressed approximately as number of passes *times* time taken by Cache-Mine.

- UDF is similar to Stored-procedure. The only difference is that the time per pass decreases by 30-50% for UDF because of closer coupling with the database.

- The SQL approach comes second in performance after the Cache-Mine approach for low support values and is even somewhat better for high support values. The cost of converting the data to the vertical format for SQL is typically lower than the cost of transforming data to binary format outside the DBMS for Cache-Mine. However, after the initial transformation subsequent passes take negligible time for Cache-Mine. For the second pass SQL takes significantly more time than Cache-Mine particularly when we decrease support. For subsequent passes even the SQL approach does not spend too much time. Therefore, the difference between Cache-Mine and SQL is not very sensitive to the number of passes because both approaches spend negligible time in higher passes.

  The SQL approach is 1.8 to 3 times better than Stored-procedure or Loose-coupling approach. As we decreased the support value so that the number of passes over the dataset increases, the gap widens.

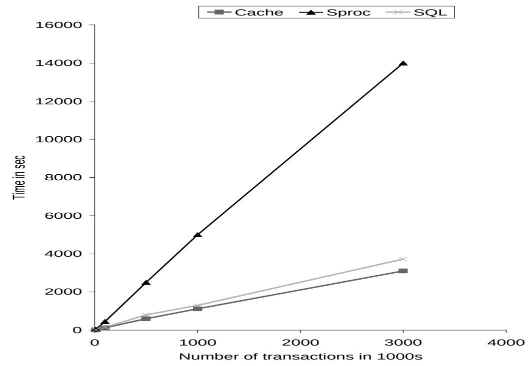


Figure 11: Scale-up with increasing number of transactions

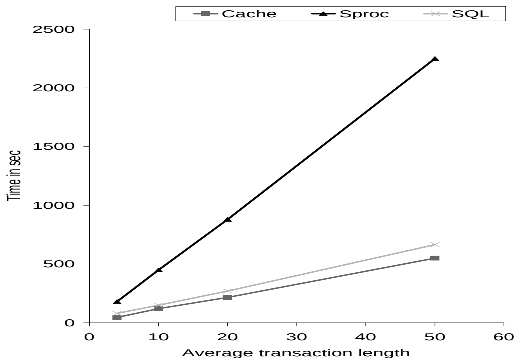


Figure 12: Scale-up with increasing transaction length

6.1.1 Scale-up experiment

Our experiments with the four real-life datasets above has shown the scaling property of the different approaches with decreasing support value and increasing number of frequent itemsets. We experiment with synthetic datasets to study other forms of scaling: increasing number of transactions and increasing average length of transactions. Figure 11 shows how Stored-procedure, Cache-Mine and SQL scale with increasing number of transactions. UDF and Loose-coupling have similar scale-up behavior as Stored-procedure, therefore we do not show these approaches in the figure. We used a dataset with 10 average number of items per transaction, 100 thousand total items and a default pattern length (defined in [3]) of 4. Thus, the size of the dataset is 10 times the number of transactions. As the number of transactions is increased from 10K to 3000K the time taken increases proportionately. The largest frequent itemset was 5 long. This explains the five fold difference in performance between the Stored-procedure and the Cache-Mine approach. Figure 12 shows the scaling when the transaction length changes from 3 to 50 while keeping the number of transactions fixed at 100K. All three approaches scale linearly with increasing transaction length.

6.2 Space overhead of different approaches

We summarize the space required for different options. We assume that the tids and items are integers. The space requirements for UDF and Loose-coupling is the same as that for Stored-procedure which in turn is less than the space needed by the Cache-Mine and SQL approaches. The Cache-Mine and SQL approaches have comparable storage overheads. For Stored-procedure and UDF we do not need any extra storage for caching. However, all three options Cache-Mine, Stored-procedure and UDF require data in each pass to be grouped on the *tid*. In a relational DBMS we cannot assume any order on the physical layout of a table, unlike in a file system. Therefore, we need either an index on the data table or need to sort the table every time to ensure a particular order. Let R denote the total number of (tid,item) pairs in the data table. Either option has a space overhead of $2 \times R$ integers. The Cache-Mine approach caches the data in an alternative binary format where each tid is followed by all the items it contains. Thus, the size of the cached data in Cache-Mine is at most: $R + T$ integers where T is the number of transactions. For SQL we use the hybrid Vertical option. This requires creation of an initial TidTable of size at most $I + R$ where I is the number of items. Note that this is slightly less than the cache required by the Cache-Mine approach. The SQL approach needs to sort data in pass 1 in all cases and pass 2 in some cases where we used the GatherJoin approach instead of the Vertical approach.

In summary, the UDF and Stored-procedure approaches require the least amount of space followed by the Cache-Mine and the SQL approaches which require roughly as much extra storage as the data. When the item-ids or tids are character strings instead of integers, the extra space needed by Cache-Mine and SQL is a much smaller *fraction* of the total data size because before caching we always convert item-ids to their compact integer representation and store in binary format. Details on how to do this conversion for SQL is presented in [21].

6.3 Summary of comparison between different architectures

We present a summary of the pros and cons of the different architectures on each of the following yardsticks: (a) performance (execution time); (b) storage overhead; (c) potential for automatic parallelization; (d) development and maintenance ease; (e) portability (f) inter-operability.

In terms of performance, the Cache-Mine approach is the best option. The SQL approach is a close second — it is always within a factor of two of Cache-Mine for all of our experiments and is sometimes even slightly better. The UDF approach is better than the Stored-procedure approach by

30 to 50%. Between Stored-procedure and Cache-Mine, the performance difference is a function of the number of passes made on the data — if we make four passes of the data the Stored-procedure approach is four times slower than Cache-Mine. Some of the recent proposals [24, 6] that attempt to minimize the number of data passes to 2 or 3 might be useful in reducing the gap between the Cache-Mine and the Stored-procedure approach.

In terms of space requirements, the Cache-Mine and the SQL approach loose to the UDF or the Stored-procedure approach. The Cache-Mine and SQL approaches have similar storage requirements.

The SQL implementation has the potential for automatic parallelization particularly on a SMP machine. Parallelization could come for free for SQL-92 queries. Unfortunately, the SQL-92 option is too slow to be a candidate for parallelization. The stumbling block for automatic parallelization using SQL-OR could be queries involving UDFs that use scratch pads. The only such function in our queries is the `Gather` table function. This function essentially implements a user defined aggregate, and would have been easy to parallelize if the DBMS provided support for user defined aggregates or allowed explicit control from the application about how to partition the data amongst different parallel instances of the function. On a MPP machine, although one could rely on the DBMS to come up with a data partitioning strategy, it might be possible to better tune performance if the application could provide hints about the best partitioning [4]. Further experiments are required to assess how the performance of these automatic parallelizations would compare with algorithm-specific parallelizations (e.g [4]).

The development time and code size using SQL could be shorter if one can get efficient implementations out of expressing the mining algorithms declaratively using a few SQL statements. Thus, one can avoid writing and debugging code for memory management, indexing and space management all of which are already provided in a database system (Note that these same code reuse advantages can be obtained from a well-planned library of mining building blocks). However, there are some detractors to easy development using the SQL alternative. First, any attached UDF code will be harder to debug than stand-alone C++ code due to lack of debugging tools. Second, stand-alone code can be debugged and tested faster when run against flat file data. Running against flat files is typically a factor of five to ten faster compared to running against data stored in DBMS tables. Finally, some mining algorithms (e.g. neural-net based) might be too awkward to express in SQL.

The ease of porting of the SQL alternative depends on the kind of SQL used. Within the same DBMS, porting from one OS platform to another requires porting only the small UDF code and hence is easy. In contrast the Stored-procedure and Cache-Mine alternatives require porting larger lines of code. Porting from one DBMS to another could get hard for SQL approach, if non-standard DBMS-specific features are used. For instance, our preferred SQL implementation relies on the availability of DB2's table functions, for which the interface is still not standardized across other major DBMS vendors. Also, if different features have different performance characteristics on different database systems, considerable tuning would be required. In contrast, the Stored-procedure and Cache-Mine approach are not tied to any DBMS specific features. The UDF implementation has the worst of both worlds — it is large and is tied to a DBMS.

One attraction of SQL implementation is inter-operability and usage flexibility. The adhoc querying support provided by the DBMS enables flexible usage and exposes potential for pipelining the input and output operators of the mining process with other operators in the DBMS. However, to exploit this feature one needs to implement the mining operators inside the DBMS. This would require major rework in existing database systems. The SQL approach presented here is based on embedded SQL and as such cannot provide operator pipelining and inter-operability. Queries on the mined result is possible with all four alternatives as long as the mined results are stored back in the DBMS.

7 Conclusion and future work

We explored various architectural alternatives for integrating mining with a relational database system. As an initial step in that direction we studied the association rules algorithms with the twin goals of finding the trade-offs between architectural options and the extensions needed in a DBMS to efficiently support mining. We experimented with different ways of implementing the association rules mining algorithm in SQL to find if it is at all possible to get competitive performance out of SQL implementations.

We considered two categories of SQL implementations. First, we experimented with four different implementations based purely on SQL-92. Experiments with real-life datasets showed that it is not possible to get good performance out of pure SQL based approaches alone. We next experimented with a collection of approaches that made use of the new object-relational extensions like UDFs, BLOBs, Table functions etc. With this extended SQL we got orders of magnitude improvement over the SQL-92 based-implementations.

We compared the SQL implementation with different architectural alternatives. We concluded that based just on performance the Cache-Mine approach is the winner. A close second is the SQL-OR approach that was sometimes slightly better than Cache-Mine and was never worse than a factor of two on our datasets. Both these approaches require additional storage for caching, however. The Stored-procedure approach does not require any extra space (except possibly for initially sorting the data in the DBMS) and can perhaps be made to be within a factor of two to three of Cache-Mine with the recent algorithms [24, 6]. The UDF approach is a factor of 0.3 to 0.5 faster than Stored-procedure but is significantly harder to code. The SQL approach offers some secondary advantages like easier development and maintenance and potential for automatic parallelization. However, it might not be as portable as the Cache-Mine approach across different database management systems.

The work presented in this paper points to several directions for future research. A natural next step is to experiment with other kinds of mining operations (e.g. clustering and classification [8]) to verify if our conclusions about associations hold for these other cases too. We experimented with generalized association rules [22] and sequential patterns [23] problems and found similar results. In some ways associations is the easiest to integrate as the frequent itemsets can be viewed as generalized group-bys. Another useful direction is to explore what kind of a support is needed for answering short, interactive, adhoc queries involving a mix of mining and relational operations. How much can we leverage from existing relational engines? What data model and language extensions are needed? Some of these questions are orthogonal to whether the bulky mining operations are implemented using SQL or not. Nevertheless, these are important in providing analysts with a well-integrated platform where mining and relational operations can be inter-mixed in flexible ways.

Acknowledgements We wish to thank Cliff Leung, Guy Lohman, Eric Louie, Hamid Pirahesh, Eugene Shekita, Dave Simmens, Amit Somani, Ramakrishnan Srikant, George Wilson and Swati Vora for useful discussions and help with DB2.

References

[1] R. Agrawal, A. Arning, T. Bollinger, M. Mehta, J. Shafer, and R. Srikant. The Quest Data Mining System. In *Proc. of the 2nd Int'l Conference on Knowledge Discovery in Databases and Data Mining*, Portland, Oregon, August 1996.

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 207–216, Washington, D.C., May 1993.

[3] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast Discovery of Association Rules. In U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, chapter 12, pages 307–328. AAAI/MIT Press, 1996.

[4] R. Agrawal and J. Shafer. Parallel mining of association rules. *IEEE Transactions on Knowledge and Data Engineering*, 8(6), December 1996.

[5] R. Agrawal and K. Shim. Developing tightly-coupled data mining applications on a relational database system. In *Proc. of the 2nd Int'l Conference on Knowledge Discovery in Databases and Data Mining*, Portland, Oregon, August 1996.

[6] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proc. of the ACM SIGMOD Conference on Management of Data*, May 1997.

[7] D. Chamberlin. *Using the New DB2: IBM's Object-Relational Database System.* Morgan Kaufmann, 1996.

[8] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining.* AAAI/MIT Press, 1996.

[9] J. Han, Y. Fu, K. Koperski, W. Wang, and O. Zaiane. DMQL: A data mining query language for relational datbases. In *Proc. of the 1996 SIGMOD workshop on research issues on data mining and knowledge discovery, Montreal, Canada*, May 1996.

[10] M. Houtsma and A. Swami. Set-oriented mining of association rules. In *Int'l Conference on Data Engineering*, Taipei, Taiwan, March 1995.

[11] IBM Corporation. *DB2 Universal Database Application programming guide Version 5*, 1997.

[12] T. Imielinski and H. Mannila. A database perspective on knowledge discovery. *Communication of the ACM*, 39(11):58–64, Nov 1996.

[13] T. Imielinski, A. Virmani, and A. Abdulghani. Discovery Board Application Programming Interface and Query Language for Database Mining. In *Proc. of the 2nd Int'l Conference on Knowledge Discovery and Data Mining*, Portland, Oregon, August 1996.

[14] Internationl Business Machines. *IBM Intelligent Miner User's Guide*, Version 1 Release 1, SH12-6213-00 edition, July 1996.

[15] K. Kulkarni. Object oriented extensions in SQL3: a status report. *Sigmod record*, 1994.

[16] J. Melton and A. Simon. *Understanding the new SQL: A complete guide.* Morgan Kauffman, 1992.

[17] R. Meo, G. Psaila, and S. Ceri. A new SQL like operator for mining association rules. In *Proc. of the 22nd Int'l Conference on Very Large Databases*, Bombay, India, Sep 1996.

[18] Oracle. *Oracle RDBMS Database Administrator's Guide Volumes I, II (Version 7.0)*, May 1992.

[19] H. Pirahesh and B. Reinwald. SQL table function open architecture and data access middleware. In *SIGMOD*, 1998.

[20] K. Rajamani, B. Iyer, and A. Chaddha. Using DB/2's object relational extensions for mining associations rules. Technical Report TR 03,690., Santa Teresa Laboratory, IBM Corporation, sept 1997.

[21] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. Research Report RJ 10107 (91923), IBM Almaden Research Center, San Jose, CA 95120, March 1998. Available from `http://www.almaden.ibm.com/cs/quest`.

[22] R. Srikant and R. Agrawal. Mining Generalized Association Rules. In *Proc. of the 21st Int'l Conference on Very Large Databases*, Zurich, Switzerland, September 1995.

[23] R. Srikant and R. Agrawal. Mining Sequential Patterns: Generalizations and Performance Improvements. In *Proc. of the Fifth Int'l Conference on Extending Database Technology (EDBT)*, Avignon, France, March 1996.

[24] H. Toivonen. Sampling large databases for association rules. In *Proc. of the 22nd Int'l Conference on Very Large Databases*, pages 134–145, Mumbai (Bombay), India, September 1996.

[25] D. Tsur, S. Abiteboul, C. Clifton, R. Motwani, and S. Nestorov. Query flocks: A generalization of association rule mining. In *SIGMOD*, 1998. to appear.

[26] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New Algorithms for Fast Discovery of Association Rules. In *Proc. of the 3rd Int'l Conference on Knowledge Discovery and Data Mining*, Newport Beach, California, August 1997.