Performance Evaluation and Analysis of K-way join variants for Association Rule Mining
P. Mishra and S. Chakravarthy
Information and Technology Lab @ CSE.UTA.EDU
The University of Texas at Arlington, Arlington, TX 76019
Email: {pmishra,sharma}@cse.uta.edu

Abstract

Data mining aims at discovering important and previously unknown patterns from the dataset in the underlying database. Database mining performs mining directly on data stored in relational database management systems (RDBMSs). The type of underlying database can vary and should not be a constraint on the mining process. Irrespective of the database in which data is stored, we should be able to mine the data. Several SQL92 approaches (such as K-way join, Query/Subquery, and Two-group by) have been studied in the literature.

In this paper, we focus on the K-way join approach. We study several additional optimizations for the K-way join approach and evaluate them using DB2 and Oracle RDBMSs. We evaluate the approaches analytically and compare their performance on large data sets. Finally, we summarize the results and indicate the conditions for which the individual optimizations are useful. The larger goal of this work is to feed these results into a layered optimizer that chooses specific strategies based on the input dataset characteristics.

1. Introduction

The rapid improvement in the storage technology with steep drop in the storage cost and increase in the computing power has made it feasible for organizations to store huge amounts of data and process it. To compete effectively in today’s market, decision makers need to identify and utilize the information hidden in the collected data and take advantage of the high return opportunities in a timely fashion. Data Mining is the process of inferring knowledge from such data repositories, and database mining incorporates the ability to directly access data stored in a database.

Association rule mining is a process of identifying the co-occurrences of one or more items that satisfy user-defined frequency (specified as support and confidence). These models are often referred to as Market Basket analysis when they are applied to retail industries to study the buying patterns of their customers. Here an attempt is made to identify whether a customer buys item “B” also, whenever he/she buys item “A”. If so, then how many customers buy item “B” along with item “A” is of interest. This is stated as an association rule of the form A \(\Rightarrow\) B. Where A is called the antecedent of the rule and B the consequence. In a more generalized form of the rule, antecedent and consequence can have more than one item (are sets).

The work on association rule mining started with the development of the AIS algorithm [1] and then some of its modifications as discussed in [2]. Since then, there have been continuous attempts in improving the performance of these algorithms. The partition algorithm [3] improves the overall performance by reducing the number of passes needed over the complete database to two. The turbo-charging algorithm [4] incorporates the concept of data compression to boost the performance of the mining algorithm. [5] builds a special tree structure in main memory to avoid multiple passes over database. However, most of these algorithms are applicable to data present in flat files. The basic characteristics of these algorithms are that they are main memory algorithms, where the data is either read directly from the flat files or is first extracted from the DBMS and then processed in the main memory. These algorithms implement their own buffer management strategies. The performance of these algorithms is due to their capability of building specialized data-structures, which is more suited to the specific algorithm. There have been very few attempts until now to build database based mining models. Here we assume that the data is already present in the form of tables in the underlying DBMS and we use the SQL capabilities provided by the RDBMS to churn it and to produce so far unseen and interesting
rules. SETM [6], showed how the data stored in RDBMS can be mined using SQL and the corresponding performance gain achieved by optimizing these queries.

Recent research in the field of database-based mining has been in integrating the mining functions with the database. Various extensions to the SQL have been proposed. These proposals are to load the SQL with certain mining operators. The Data Mining Query Language DMQL [7] proposed a collection of such operators for classification rules, characteristics rule, association rules, discriminant rules, etc. [8] proposed the MineRule operator for generating general/clustered/ordered association rules. [9] presents a methodology for tightly-coupled integration of data mining applications with a relational database system. In [10] the authors have tried to highlight the implications of various architectural alternatives for coupling data mining with relational database systems. They have also compared the performance of the SQL-92 based architecture with SQL-OR based architecture and when mining is done outside the database address space.

Some of the research has focused on the development of SQL-based formulations for association rule mining. Most of these algorithms use the apriori algorithm directly or indirectly with some modifications to it. [10] and [11] deal with the SQL implementation of the apriori algorithm and have compared some of the optimizations to the basic k-way join algorithm for association rule mining but the relative performances and all possible combinations for optimizations were not explored. In this paper, we will analyze the characteristics of these optimizations in detail both analytically and experimentally. We conclude why certain optimizations are always useful and why some perceived optimizations do not seem to work as intended.

There are many commercial mining tools available today in the market, viz., the IBM’s Intelligent Miner, DBMiner, etc., which use the capabilities provided by the underlying database management system for mining. Though these mining tools are quite efficient, they are developed for a particular RDBMS. Hence, they cannot be used if the relevant database is not used. To overcome this limitation, our approach uses a database independent architecture introduced in [12]. To make the implementation operating system independent, we have used java and along with it, we use JDBC API’s to make our implementation independent of the underlying database. For the purpose of our evaluation, we have run the experiments on both Oracle 8i and IBM DB2/UDB 6.1.

1.1 Focus of this paper

With more and more use of RDBMS to store and manipulate data, mining directly on RDBMSs gives us the advantage of using the fruits of decades of research done in this field. As the main memory always imposes a limitation on the size of data that can be processed, using RDBMSs provides us the benefits of using a sophisticated buffer management systems implemented in them. Building mining algorithms to work on RDBMSs directly also gives us the advantage of mining over very large datasets as RDBMSs have been built to manage such large volumes of data. File based mining algorithms or those that work on data outside the database, generally have an upper limit to the number of transaction that can be mined. For example, the DBMiner has an upper limit of 64K on the number of unique transactions that it can process for mining. With the user having a choice of RDBMS to use for his application, the mining algorithms should be developed using such accepted standards so that the underlying system is not a limitation and should be portable on other RDBMSs. Keeping this in mind, our focus in this paper is on the use of SQL-92 features for association rule mining over RDBMSs. Also. We have tried some of the SQL-OR implementations (using UDF or user defined functions and table functions for DB2 and Java / PL/SQL stored procedures for Oracle) for association rule mining. The results of the mining done using the SQL-OR implementations show that these algorithms have poor performance when compared to the SQL-92 implementations. Hence this makes it necessary for us to study the SQL-92 based algorithms and their optimizations.

The goal of this paper is to study the basic k-way join approach for association rule mining and then explore additional performance optimizations to the k-way join. The paper elaborates more on this problem. The other goal of our work is to be able to use the results obtained from mining various relations
to make the mining optimizer mining-aware. Most of the relational query optimizers were not designed to optimize queries that are typically used in mining. Furthermore, current optimizers cannot be given any external input in guiding them towards generating a specific query plan. Hence, the results collected from the performance evaluations of these algorithms are critical for developing a knowledge base that can be used for selecting appropriate approaches as well as optimizations with in a given approach. Due to lack of availability of real datasets, we use synthetic datasets (generated by the program developed at IBM Almaden) for performance evaluation. Nevertheless, the results are useful in understanding the approaches and can certainly be converted into meta-data for use by mining application.

Figure 1 shows the proposed architecture. At the heart of the architecture is the mining optimizer. This optimizer uses metadata, which is inferred by the analytical evaluation and the results obtained from a series of experiments on the various formulations of basic K-way join based apriori algorithm [10], [11], [12] and its optimizations.

The rest of the paper is organized as follows. Section 2 introduces the association rules and the apriori algorithm. Section 3 covers in detail the k-way join method for support counting and its basic optimizations along with their performance analysis. Section 4 considers the combinations of the optimizations discussed in section 3 and reports some of the results due to space limitations. The details can be found in [13] available on the web. In section 5 we have compiled the summary of results obtained from mining various datasets. We conclude and present the future work in section 6.

2. Association Rules

The problem of association rule mining was formally defined in [2]. In short, it can be stated as: Let $I$ be the collection of all the items and $D$ be the set of transactions. Let $T$ be a single transaction involving some of the items from the set $I$. The association rule is of the form $A \Rightarrow B$ (where $A$ and $B$ are sets). If the support of itemset $AB$ is 30%, it means “30% of all the transactions contain both the itemsets – itemset $A$ and itemset $B$”. And if the confidence of the rule $A \Rightarrow B$ is 70%, it means “70% of all the transactions that contain itemset $A$ also contains itemset $B$”. In this section, we discuss SQL92 formulation for the generic apriori algorithm that serves as the basis for our analysis. An association rule mining problem is broken down into two sub-problems: 1) select all the item combinations (itemsets) whose support is greater than the user specified minimum support. Such itemsets are called the frequent itemsets and 2) use the identified frequent itemsets to generate all rules that satisfy user specified confidence.

2.1 Apriori Algorithm

The apriori algorithm is based on the above-mentioned two steps: generate frequent itemsets and generate corresponding rules. Frequent itemsets are generated in two steps. In the first step all the possible combinations of items, called the candidate itemsets ($C_k$) are generated. In the second step, support of each candidate itemset is counted and all those candidate itemsets, which have support values greater than
the user specified minimum support value, form the set of frequent itemsets \( (F_k) \). The algorithm is depicted below.

\[
F_1 = \{ \text{frequent itemsets of length 1} \}
\]

\[
\text{for } (k = 2; F_{k-1} \neq 0; k++) \text{ do}
\]

\[
C_k = \text{generate}(F_{k-1})
\]

\[
\text{for all transactions } t \in D \text{ do}
\]

\[
C_t = \text{subset}(C_k, t)
\]

\[
\text{for all candidates, } c \in C_t \text{ do}
\]

\[
c.\text{count++}
\]

\[
\text{end for}
\]

\[
\text{end for}
\]

\[
F_k = \{ c \in C_k \mid c.\text{count} \geq \text{minsup} \}
\]

\[
\text{end for}
\]

\[
\text{Answer} = \bigcup_k \{F_k\}
\]

The generate function uses frequent itemsets of length \( k-1 \) to generate candidate itemsets of length \( k \), while the subset function produces only that subset of candidate itemsets from \( C_k \) which can be generated from the items bought in any transaction \( t \).

### 2.1.1 Candidate Generation (function generate\((F_k)\))

In the \( k^{th} \) pass, the set of candidate itemsets \( C_k \) is generated from the frequent itemset \( F_{k-1} \) as shown below. \( F_{k-1} \) is generated in the \((k-1)^{th}\) pass. Relations \( C_k \) and \( F_k \) have following attributes \( (\text{item1, item2,..., itemk}) \)

\[
\text{Insert into } C_k
\]

\[
\text{Select } I_1.\text{item}_1, \ldots, I_1.\text{item}_{k-1}, I_2.\text{item}_{k-1}
\]

\[
\text{From } F_{k-1} I_1, F_{k-1} I_2
\]

\[
\text{Where } I_1.\text{item}_1 = I_2.\text{item}_1 \text{ and }
\]

\[
\text{...}
\]

\[
I_1.\text{item}_{k-2} = I_2.\text{item}_{k-2} \text{ and }
\]

\[
I_1.\text{item}_{k-1} < I_2.\text{item}_{k-1}
\]

The number of candidate itemsets generated by the above step can be reduced by deleting all itemsets \( c \in C_k \) where some subset of \( c \) of length \( k-1 \) is not in \( F_{k-1} \). This has been introduced in [2].

### 2.1.2 Support Counting

This is an important and most time-consuming part of the mining process. This step is needed to identify those candidate itemsets that are frequent. The basic approach for support counting is that in any pass \( k \), \( k \) copies of the input table are joined with the candidate itemsets \( C_k \) followed by a group by on the itemsets. The \( k \) copies of the input table are needed to compare the \( k \) items in the candidate itemset \( C_k \) with one item from each of the \( k \)-copies of the input table. The group by clause on the \( k \) items is done to identify all itemsets whose count is \( \geq \) support value, as frequent itemsets. These frequent itemsets are then used in the rule generation phase. The tree diagram for support counting using \( k \)-way join approach is shown below [10], [11], [12].
3. Analysis of basic K-way join method and its optimizations

In this section we analyze the k-way join approach more closely. This section, also discusses a number of basic optimization and their combinations. The purpose of these optimizations and their analysis (along with performance evaluation) is to understand the impact of various optimizations on datasets having different characteristics (size, average transaction length, support, confidence, number of passes needed etc.) and to obtain heuristics that relate various optimization techniques and their effect on the characteristics of the input dataset. Though, not all optimizations always produce better timings, our belief is that the study of these optimizations can give us a better insight in formulating the metadata that can be used for making a mining-aware mining optimizer. In addition to the analytical evaluation for these optimizations, we also present the performance evaluation and the results obtained when datasets with different characteristics were mined using them. The cost analysis for the basic k-way join approach is done in [10] and in [11]. We use similar notations for our study of these optimizations. These notations are described in the table given below.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>Number of records in the input transaction table</td>
</tr>
<tr>
<td>T</td>
<td>Number of transactions</td>
</tr>
<tr>
<td>N</td>
<td>Avg. number of items per transaction = R/T</td>
</tr>
<tr>
<td>F&lt;sub&gt;i&lt;/sub&gt;</td>
<td>Number of frequent items</td>
</tr>
<tr>
<td>S(C)</td>
<td>Sum of support for each itemset in C</td>
</tr>
<tr>
<td>s&lt;sub&gt;k&lt;/sub&gt;</td>
<td>Average support of a frequent k-itemset = S(F&lt;sub&gt;k&lt;/sub&gt;)/(F&lt;sub&gt;k&lt;/sub&gt;)</td>
</tr>
<tr>
<td>R&lt;sub&gt;f&lt;/sub&gt;</td>
<td>Number of records out of R involving frequent items = S(F&lt;sub&gt;i&lt;/sub&gt;)</td>
</tr>
<tr>
<td>N&lt;sub&gt;f&lt;/sub&gt;</td>
<td>Average number of frequent items per transaction = R&lt;sub&gt;f&lt;/sub&gt;/T</td>
</tr>
<tr>
<td>C&lt;sub&gt;k&lt;/sub&gt;</td>
<td>Number of candidate k-itemsets</td>
</tr>
<tr>
<td>C (N,K)</td>
<td>Number of combinations of size k possible out of a set of size n: (n!)/(k!(n-k)!)</td>
</tr>
<tr>
<td>group (n,m)</td>
<td>Cost of grouping n records out of which m are distinct</td>
</tr>
<tr>
<td>join (n,m,r)</td>
<td>Cost of joining two relations of size n and m to get a result of size r</td>
</tr>
</tbody>
</table>

Table 1 Notions used for cost analysis of different approaches

The methodology used for cost analysis is a very general approach to estimate the cost of each optimization. The notations used, do not tell us how the underlying optimizer manages to compute the
SQL query, as that might differ from vendor to vendor (this is very clearly evident from our experimental
results over Oracle and IBM DB2). The formulae does not compare the CPU or the I/O required for
computing the query, but they are powerful enough to provide a guiding cue to be used for choosing the
appropriate optimization for association rule mining for a given dataset.

3.1 Methodology for experimental evaluation

The results shown here are on datasets generated synthetically using the IBM’s data-generator. The
nomenclature of these datasets is of the form TxxIyyDzzzK. Where xx denotes the average number of
items present per transaction. yy denotes the average support of each item in the dataset and zzzK denotes
the total number of transactions in K (1000’s). The experiments have been performed on Oracle 8i
(installed on a Solaris machine with 384MB of RAM) and IBM DB2/UDB (over Windows NT with
256MB of RAM). Each experiment has been performed 4 times. The values from the first results are
ignored so as to avoid the effect of the previous experiments and other database setups. The average of
the next 3 results is taken and used for analysis. This is done so as to avoid any false reporting of time due
to system overload or any other factors. Before feeding the input to the mining algorithm, if it is not in the
(tid, item) format, it is converted to that format (by using the algorithm and the approach presented in
[12]). On completion of the mining, the results are remapped to their original values. Since the time taken
for mapping, rule generation and re-mapping the results to their original descriptions is not very
significant, they are not reported.

For the purpose of reporting the experimental results in this paper, for most of the optimizations we
have shown the results only for three datasets – T512D500K, T512D1000K and T1014D100K. Wherever
there is a marked difference between the results for Oracle and IBM DB2/UDB they are also shown;
otherwise the result from anyone of the RDBMSs have been included.

3.2 Cost Analysis of the basic K-way join (Kwj) approach

For support counting, in any pass k, k copies of input table are joined with Ck (Figure 2). The total
number of items produced in a join of Ck with T is equal to the number of records in Ck * average support
of first item in Ck. Using the join notations, the join cost can be represented as join (Ck, R, Ck*s1).
Similarly, the join of m-copies of T with Ck will result in a table, containing sum of support count for first
m-items of an itemset in Ck. In terms of join notation this can be represented as join (Ck*sm-1, R, Ck*sm).
The cost of last join, can’t be calculated from the above formula as the sk value for the itemset of length k
is not known since the candidate itemsets of length k produced, is not frequent. But the relation obtained
from the last join will have as tuples records as the sum of support for each itemset in set Ck. Using the
join notation this can be represented as join (Ck*sk-1, R, S(Ck)). Hence the cost of any pass, for this
approach can be given as:

\[
\sum_{m=1}^{k-1} \text{join}(C_k * s_{m-1}, R, C_k * s_m) + \text{join}(C_k * s_{k-1}, R, S(C_k)) \oplus \text{group}(S(C_k), C_k).
\]

Equation 1 Cost of basic K-way Join

Figure 3 compares the time required for mining the relation T512D1000K on DB2, with break-up for
each pass for support values of 0.20%, 0.15% and 0.10%, while Figure 4 shows the same on Oracle. (On
DB2, for support value of 0.10%, the experiment didn’t complete even after running it for over 9 hrs).
The analysis for time required for each pass shows that, of all the passes, second pass is most time
consuming. This is true as in second pass "C2 (all combinations of two elements from frequent 1-itemsets)
candidate itemsets are generated, where n is the cardinality of frequent-1 itemset.
Figure 5 shows the time required for mining relation T10I4D100K for different support values on Oracle and Table 2 shows the number of candidate itemsets generated in respective passes, when different tables were mined with different support values. The analysis of the theses figures shows that for mining configuration, where the length of the largest frequent itemsets is small, the time required for support counting at higher passes is not very significant. This is because there is a great reduction in the size of the candidate itemset (C_k). However, for relations with long frequent itemsets, even though the cardinality of the C_k decreases with the increase in the number of passes, even then joining k-copies of input table for support counting at higher passes is quite significant.

<table>
<thead>
<tr>
<th>Relation</th>
<th>C_2</th>
<th>C_3</th>
<th>C_4</th>
<th>C_5</th>
<th>C_6</th>
<th>C_7</th>
<th>C_8</th>
<th>C_9</th>
</tr>
</thead>
<tbody>
<tr>
<td>T5I2D500K. Support = 0.10%</td>
<td>307720</td>
<td>126</td>
<td>7</td>
<td>0</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>T5I2D1000K. Support = 0.10%</td>
<td>309291</td>
<td>127</td>
<td>61</td>
<td>0</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>T10I4D100K. Support = 0.75%</td>
<td>12470</td>
<td>65</td>
<td>6</td>
<td>0</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>T10I4D100K. Support = 0.33%</td>
<td>216153</td>
<td>2453</td>
<td>905</td>
<td>354</td>
<td>109</td>
<td>20</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2 Number of Candidate Itemsets in different passes

In Equation 1 for support counting of any pass k, the input table is joined k-times. Hence an obvious way to optimize this would be by reducing the cardinality of the input table. Section 3.3 discusses about this in more detail. Once again, if we analyze the first and the second pass, frequent itemsets of length 1 (F_1) are generated in pass 1. F_1 is then used to generate C_2, which is followed by support counting of C_2. An efficient way to get around with this time consuming process would be generating the frequent itemsets of length 2 (F_2) directly, by joining the input table with itself with the group-by on the items of the input table that have the same tid. This way Pass 1 can be skipped all together (as there are no rules on
F_1) and also there is no need for candidate generation for pass 2 as F_2 is generated directly by the above
step. Section 3.4 discusses this second pass optimization in more detail.

Let us compare the SQL tree for support counting (Figure 2) for two successive passes, say pass 4 and
pass 5. In the 4th pass C_4 is joined with 4 copies of input table, to identify all frequent itemsets of length 4.
In the 5th pass, again input table is joined 4 times for determining the frequent itemsets of length 4 and
then the support of 1-extensions of these frequent items, present as the fifth item in C_5, are counted by
joining one more copy of the input table with C_5. Thus if all the frequent itemsets contained in any
transaction is saved at the end of the pass 4, they can be used for support counting in pass 5, as frequent
itemsets of length 5 are 1-extensions of these frequent itemsets of length 4. Section 3.5 discusses about
this optimization and its effects.

3.3 Pruning the Input Table (Pi)

Smaller the size of the input table, the faster should be the join computation. Eliminating the records
of those single itemsets from the input table, whose support is lower than the user specified minimum
value, can reduce the size of the input table. Instead of deleting these tuples, a new relation, say, pruned
input (T_f) is created to contain the tuples (transactions) of only frequent itemsets of length 1. This is done
by generating F_1, as before. Then F_1 is joined with T on the “item” column, and tuples of only those items
whose support \( \geq \) user defined support value are inserted in T_f. The SQL for creating the pruned input table
is given below:

\[
\text{Insert into } T_f \text{ select t.tid, t.item } \\
\text{From } T t, F1 f \\
\text{Where } t.item = f.item
\]

Thus the overall cost of this optimization includes the cost of producing the pruned input table T_f +
cost of support counting in every pass. This can be given as the one time cost of pruning:

\[
\text{cost of support counting in every pass as: }
\]

\[
\sum_{m=1}^{k-1} \text{join}(C_k \ast s_{m-1}, R_f, s_m) + \text{join}(C_k \ast s_{k-1}, R_f, S(C_k)) \Rightarrow \text{group}(S(C_k), C_k).
\]

Equation 2 Cost estimation of K-way with pruned input.

The difference between Equation 1 (basic Kwj) and Equation 2 (Pi) is that, in Equation 2, an
additional cost for materializing the pruned input table is involved. And then this pruned input table is
used instead of the original input table in the joins for the support counting of every pass. The pruning of
non-frequent 1-itemset is more effective with higher support values or for relations with a very large
number of distinct items, which results in pruning out a large number of non-frequent 1-itemsets. Figure 6
shows the reduction in size of input table T512D1000K for different support values. The reduction in the
size of the input table is very marked for higher support values. But, pruning might not always end up in
giving better performance. Figure 7 compares the total cost of mining the relation T512D1000K on oracle,
using the pruned input (time for pruning also considered and is shown as “Ohead” in the figure) with the
basic Kwj approach, for different support values. For higher support values, (3.0%, 2.0% and 1.0%), the
total time taken is less when pruned input was used, but the reverse is true for lower support values. This
is because for low support values, the reduction in the size of the input table is almost negligible; hence
use of pruned input hardly has any effect on the running time of any pass, rather, there is an additional
cost involved in generation of the pruned input. Because of this overhead, the overall time of using
pruned input comes out more than the basic Kwj.
From the above analysis, if the characteristics of the input table are:

- \# of distinct items = \( d \) (the probability of each item being in a transaction is same)
- \# of transactions = \( T \)
- support (in percent) = \( s\% \)
- and if \( s \cdot T / 100 \leq T / d \),

then with high probability, the overhead of pruning will be more than the performance obtained by the use of the pruned table. We can keep this information in the metadata, which the optimizer can use. The results obtained from several experiments done on tables with varying characteristics also validates this.

### 3.4 Second Pass Optimization (Spo)

As indicated earlier and is apparent as well from the figures shown above that of all the passes, second pass is the most time consuming. In most of the cases, because of the immense size of \( C_2 \), the cost of support counting for \( C_2 \) is very high. In addition, for candidate sets of length 2, as all the subsets of length 1 are known to be frequent, there is no gain from pruning during the candidate generation. The process of generating \( F_1 \) then \( C_2 \) followed by its support counting phase can be replaced by directly generating \( F_2 \) by joining two copies of the input table. The SQL for this is as follows:

```
Insert into F_2 select t_1.item, t_2.item, count(*)
From T t1, T t2
Where t1.tid = t2.tid and t1.item < t2.item
Group by t1.item, t2.item.
Having count(*) > support
```

The cost of second pass is: \( \text{join}(R, R, C(N, 2)) \oplus \text{group}(C(N, 2), C(F_1, 2)) \) and the cost of other passes \( (k > 2) \) will be:

\[
\sum_{m=1}^{k-1} \text{join}(C_k \ast s_{m-1}, R, C_k \ast s_m) + \text{join}(C_k \ast s_{k-1}, R, S(C_k)) \oplus \text{group}(S(C_k), C_k).
\]

**Equation 3 Cost of Second Pass Optimization**
Figure 8 K-way join and Second Pass Optimization

Figure 8 compares the overall time required for mining table T5I2D500K using Kwj and Spo. For table T5I2D500K, the overall time required for mining is reduced by 3 to 4 times with this optimization alone. Figure 9 compares the time taken for the candidate generation phase (C_k) and support counting phase (F_k) in each pass for the Kwj and the Spo. The values in Pass-2 of this figure show that the improvement in performance for Spo is due to savings on the join cost at two stages – during generation of C_2 and during generation of F_2. The saving in the former case is due to totally bypassing the generation of candidate itemsets C_2. The reason for the improvement in the second stage will become clear by analyzing the costs of the SQL statements executed for the second pass for Kwj (Equation 1) and Spo (Equation 3). In Spo, for generation of the frequent itemset F_2, input table (T) is directly joined with itself, instead of joining three tables - C_2 and 2 copies of input table (as is done in second pass of Kwj), which results in decrease in the computation time for F_2.

3.5 Reuse of Item Combinations (Ric)

This optimization aims to reduce the cost of support counting by avoiding the join of k copies of input table with C_k. This is done by materializing the frequent itemsets obtained from a particular transaction in pass k-1 (F_{k-1}), and using it for support counting in the kth pass. This saves from redoing the same series of joins that were done in the previous pass, which proves to be very effective for cases where the length of the frequent itemset is large by avoiding large number of joins. So in kth pass for support counting, a relation Comb_k having the following attributes: tid, item_1, item_2, ..., item_k is created. The tuples in Comb_k is the result of the join between Comb_{k-1}, T and C_k to select all those transactions in T which contains 1-extensions to the frequent itemsets of length k-1 (F_{k-1}). Then F_k is generated from Comb_k by grouping on k items (item_1, item_2, ..., item_k) and filtering those that don’t satisfy the minimum support criteria. Due to space constraints the SQL formulation is not given here. Please refer to [13] for details.

The analysis of this optimization shows that instead of joining the input table k times, in any pass k, only 3 relations – C_k, T, and Comb_{k-1} are joined. However, the downside of this approach is that, Comb_{k-1} has to be materialized so that it can be used in the next pass. To evaluate the cost of this optimization, the notation for join of two tables is extended to three tables as trijoin (p, q, r, s); which means, relations having p, q, r tuples respectively are joined to produce a relation with s number of tuples. Using this notation the cost of any pass k in this optimization is given as:

\[ \text{trijoin}(\text{Comb}_{k-1}, R, C_k, S(C_k)) + \text{group}(S(C_k), C_k) \]

Where the cardinality of Comb_{k-1} will be S(C_{k-1}).

Equation 4 Cost for Reuse of Item Combinations.
Figure 10 Reuse of Item Combination (Oracle)  
Figure 11 Reuse of Item Combination (DB2)  

Figure 10 compares the total time taken to mine table T10I4D100K using Kwj and Ric for different support values on Oracle. Figure 11 shows the same for DB2. For higher support values, the experiments run for less number of passes and hence the cost of support counting using the Kwj, does better than when Comb_k is materialized for using it in the next pass. But for low support values, at higher passes, the cost of joining input table k-times with C_k turns out to be more costly than materializing Comb_k-1 and using it for support counting. In Figure 11, for support value of 0.75%, Kwj didn’t complete (even after running for 9 hrs.), while the reuse of item combinations did. This can be explained more clearly by comparing Equation 1 (for Kwj) and Equation 4 (for Ric). In Equation 1, for support counting of the 3rd pass, 4 relations are joined - 3 copies of the input table and C_3, while in Equation 4 just 3 relations are joined - Comb_2, T and C_2 to get Comb_3 and then group by on Comb_3 is done for F_3. Because of the immense size of these tables and more number of joins, the experiments (on DB2) in the former case, didn’t complete.

4. Combinations of Basic Optimizations

Sections 3.3, 3.4 and 3.5 discussed, respectively, the use of pruned input, optimization of the second pass and reusing the item combinations generated in the previous pass. In this section we will discuss the optimization obtained by combining these optimizations.

4.1 Second Pass Optimization on Pruned Input (SpoPi)

Sections 3.3 and 3.4 discusses about using pruned input and optimizing the second pass. In this section we will discuss the optimization obtained by combining these two optimizations. As Spo does results in some performance gain under all situations but the same is not true for using the pruned input table, hence the overall performance obtained from this combination is limited by the overhead of pruning. The cost of this is optimization is given as:

Pruning overhead: $join(R, F_1, R_f)$

Cost of Second Pass: $join(R_f, R_f, C(N_f,2)) + group(C(N_f,2), C(F_1,2)) +$

Cost of each pass k (k>2):

\[
\sum_{m=1}^{k-1} join(C_k * s_{m-1}, R_f, C_k * s_m) + join(C_k * s_{m-1}, R_f, S(C_k)) + group(S(C_k), C_k).
\]

Equation 5 Cost for Second Pass Optimization on Pruned Input.
Figure 12 compares the time taken for mining the table T5I2D500K using the basic Kwj, using the pruned input, by optimizing the second pass and by combining the second pass optimization with the use of pruned input, for different support values on Oracle. In this experiment use of Spo along with pruning turns out to be better than the use of pruning alone but is costlier than the basic Kwj and the Spo. This is because, the overhead of building pruned input outcomes any performance gain in using it for support counting. Thus this optimization should be used where it is sure that the use of pruned input will result in overall decrease in the mining time.

4.2 Reuse of Item Combinations on Pruned Input (RicPi)

This optimization is similar to the one discussed in the section 4.5, except that instead of using the input table as it is, non-frequent itemsets of length-1 are pruned out and then this pruned input table is used in all passes for joining with the Comb_{k-1} to produce Comb_k. The overall cost of this plan is given as: Cost of Pruning + cost of generating frequent itemset for each pass. Where the cost of generating frequent itemset for any pass is given as:

\[ \text{tijon}(\text{Comb}_{k-1}, R_f, C_k, S(C_k)) + \text{group}(S(C_k), C_k) \]

and cost of pruning is: \( \text{join}(R, F_1, R_f) \). The cardinality of Comb_{k-1} will be \( S(F_{k-1}) \).

Equation 6 Cost of Reuse of Item Combinations on Pruned Input.

The analysis of this combination for all the different tables mined, shows that almost no where this combination of optimizations produced the added performance of both - (1) reusing the frequent itemsets generated in the previous pass and (2) use of pruned input for support counting. The overall performance for this combination is dominated either by the cost of building the pruned input at low support values or by the cost of materializing the Comb_2 (for using it in the next pass for support count) at high support values. This seems to be quite logical because, as seen earlier the effect of pruning dominates only for high support values and reuse of item combinations is effective for cases where the maximum length of the frequent itemset is large. But since for large support values, the maximum length of the frequent itemset is quite small, hence we never obtain the benefits of materializing the transactions with frequent itemsets of the previous pass. Similarly for low support values, where there is hardly any effect of pruning on the input table size, the overhead of pruning eclipses any time saved by reusing the item combinations. Figure 13 shows this for table T5I2D500K on DB2.
4.3 Reuse of Item Combinations and Second Pass Optimization (RicSpo)

This section describes the effect of combining Spo with the optimization where frequent itemsets generated in the previous pass are materialized and used for support counting. As described in section 3.4 for Spo, first pass and candidate generation in second pass is skipped. Since in Spo, \( C_2 \) is not generated hence in RicSpo, instead of generating \( \text{Comb}_2 \), input table is joined thrice with \( C_3 \) to produce \( \text{Comb}_3 \) directly (\( C_3 \) is produced in the same way as is done in the Spo). And then for subsequent passes, the query is similar to one discussed in section 3.5. The SQL for generating \( \text{Comb}_3 \) directly is shown below:

\[
\begin{align*}
\text{Insert into} & \quad \text{Comb}_3 \select \ t_1.\text{tid}, \ t_1.\text{item}, \ t_2.\text{item}, \ t_3.\text{item} \\
\text{From} & \quad \ T \ t_1, \ T \ t_2, \ T \ t_3, \ C_3 \\
\text{Where} & \quad t_1.\text{item} = C_3.\text{item}_1 \text{ and} \\
& t_2.\text{item} = C_3.\text{item}_2 \text{ and} \\
& t_3.\text{item} = C_3.\text{item}_3 \text{ and} \\
& t_1.\text{tid} = t_2.\text{tid} \text{ and} \\
& t_2.\text{tid} = t_3.\text{tid}
\end{align*}
\]

So the cost for this optimization can be expressed as:

Cost of Second Pass: \( \text{join}(R, R, C(N_f,2)) + \text{group}(C(N_f,2), C(F_1,2)) + \)

Cost of Third Pass: \( \text{quadjoin}(R, R, R, C_3, C(N_f,3)) + \text{group}(C(N_f,3), C(F_2,3)) + \)

Cost of subsequent pass \( k (k>3): \text{trijoin}(\text{Comb}_{k-1}, R, C_k, S(C_k)) + \text{group}(S(C_k), C_k)\).

Where \( \text{quadjoin}(p,q,r,s,t) \) means the cost of joining four relations having \( p, q, r \) and \( s \) tuples respectively to produce a relation with \( t \) number of tuples and the cardinality of \( \text{Comb}_3 \) is \( S(F_3) \).

Equation 7 Cost of Reuse of Item Combinations and Second Pass Optimization.

Figure 14 compares second pass optimization and reuse of frequent itemsets of the previous pass with their combination for table TS12D1000K on Oracle. Figure 15 shows the same on DB2. As seen from these figures, in reuse of item combination, second pass takes most of the time. This is to materialize \( \text{Comb}_2 \), which is very expensive (Table 3 shows the number of tuples in \( \text{Comb}_k \) for any pass-\( k \) for different support values). Hence the combined optimization does better than just the reuse of item combination as it skips the generation of \( C_2 \) and \( \text{Comb}_2 \). Thus for most of the experiments, this combination of optimization has resulted as one of the best optimization.
4.4 Reuse of Item Combinations on Pruned Input with Second Pass Optimization (All)

This is the last optimization, which is basically the combination of all the three individual optimizations discussed in sections 3.3, 3.4 and 3.5. The SQL for this plan is similar to the one discussed in section 4.3, except that instead of using the input table as such, we first prune out all the non-frequent itemsets and then in place of input table use this pruned table for support counting. So the cost for this optimization would be:

Pruning overhead: $\text{join}(R, F_1, R_f)$ +

Cost of Second Pass: $\text{join}(R_f, R_f, C(N_f,2)) + \text{group}(C(N_f,2), C(F_1,2)) +$

Cost of Third Pass: $\text{quadjoin}(R_f, R_f, R_f, C(N_f,3)) + \text{group}(C(N_f,3), C(F_2,3)) +$

Cost of each pass k (k>3): $\text{trijoin}(\text{Comb}_k-1, R_f, C_k, S(C_k)) + \text{group}(S(C_k), C_k))$.

Equation 8 Cost of all the optimization combined.

<table>
<thead>
<tr>
<th>Table Characteristics</th>
<th>Number of tuples in 1000’s</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Comb2</td>
</tr>
<tr>
<td>T512D1000K. Support = 0.20%</td>
<td>13267</td>
</tr>
<tr>
<td>T512D1000K. Support = 0.15%</td>
<td>13756</td>
</tr>
<tr>
<td>T512D1000K. Support = 0.10%</td>
<td>14165</td>
</tr>
</tbody>
</table>

Table 3 Number of records in Comb_k (in 1000’s)

As seen above, optimizing the second pass does saves some time in almost all cases and also the combination of 2 Pass optimization with reuse of item combination has shown to be one of the most effective combination of optimizations, but at the same time use of pruned input with reuse of item combination has never given added performance. Out of the two sets of extreme sub-combinations - (1) reuse of item combination with pruned input and (2) reuse of item combination with second pass optimization, this combination of all individual optimizations is dominated by the former sub-combination, which eclipses any performance gained by the second sub-combination. Reuse of item combinations with pruned input might be very effective for very high support values, where pruning out the non-frequent 1 itemsets might greatly reduce the size of the input table, outperforming any other optimization, while the latter sub-combination dominates when the support value is low and the experiment runs for large number of passes, resulting in performance by joining only 3 tables in any pass.
for support counting instead of joining k-copies of input table along with optimizing the second pass. Figure 16 and Figure 17 compares the above two sub-combination with when all the optimizations are combined together for table T5I2D1000K on DB2 and Oracle respectively.

5. Summary of Experimental Results

We have compiled the results obtained from mining different relations into a tabular form. This can also be made available to the mining optimizer that can use these values as a cue for choosing a particular optimization for mining a given input relation. Here it is assumed that we can easily figure out the characteristics of the input table. Table 4 and Table 5 below, summarizes the ranking of various optimizations based on their performance and also the trend seen in the performance of these optimizations in mining three relations (T5I2D1000K, T5I2D500K and T10I4D100K) with different support values on Oracle and IBM DB2/UDB respectively. For each of these relations, and for different support values, the summary table contains 3 columns. The first two columns specify the two best optimizations and the last column lists the worst optimization for that relation. The format is same for both – Oracle and IBM DB2/UDB. For the purpose of understanding how the meta-data might look, in Table 6 below, we provide a summary of results obtained from mining various other tables on both IBM DB2/UDB and Oracle. Because of the space constraint, their details are skipped. The focus of this summary table is to aid the optimizer in picking up the proper optimization based on a couple of easily determinable constraints. These constraints are: RDBMS to use (if there is a choice), the cardinality of the input table and if there is enough additional space to materialize the intermediate results.

### RDBMS: ORACLE

<table>
<thead>
<tr>
<th>Table Name</th>
<th>Ranking</th>
<th>Supp = 0.20%</th>
<th>Supp = 0.15%</th>
<th>Supp = 0.10%</th>
</tr>
</thead>
<tbody>
<tr>
<td>T5I2D1000K</td>
<td>First</td>
<td>RICSPO</td>
<td>RICSPO</td>
<td>RICSPO</td>
</tr>
<tr>
<td></td>
<td>Second</td>
<td>ALL</td>
<td>ALL</td>
<td>RICSPO</td>
</tr>
<tr>
<td></td>
<td>Last</td>
<td>RICPI</td>
<td>RICPI</td>
<td>RICPI</td>
</tr>
<tr>
<td>T5I2D500K</td>
<td>First</td>
<td>RICSPO</td>
<td>RICSPO</td>
<td>SPO</td>
</tr>
<tr>
<td></td>
<td>Second</td>
<td>SPO</td>
<td>SPO</td>
<td>RICSPO</td>
</tr>
<tr>
<td></td>
<td>Last</td>
<td>RICPI</td>
<td>RICPI</td>
<td>RICPI</td>
</tr>
<tr>
<td>T10I4D100K</td>
<td>First</td>
<td>ALL</td>
<td>RICSPO</td>
<td>RICSPO</td>
</tr>
<tr>
<td></td>
<td>Second</td>
<td>PI</td>
<td>ALL</td>
<td>ALL</td>
</tr>
<tr>
<td></td>
<td>Last</td>
<td>RC</td>
<td>RICPI</td>
<td>RICPI</td>
</tr>
</tbody>
</table>

Table 4 Trends in Oracle

### RDBMS: DB2

<table>
<thead>
<tr>
<th>Table Name</th>
<th>Ranking</th>
<th>Supp = 0.20%</th>
<th>Supp = 0.15%</th>
<th>Supp = 0.10%</th>
</tr>
</thead>
<tbody>
<tr>
<td>T5I2D1000K</td>
<td>First</td>
<td>RICSPO</td>
<td>SPO</td>
<td>RICSPO</td>
</tr>
<tr>
<td></td>
<td>Second</td>
<td>SPO</td>
<td>RICSPO</td>
<td>ALL</td>
</tr>
<tr>
<td></td>
<td>Last</td>
<td>RICPI</td>
<td>SPOI</td>
<td>SPOI</td>
</tr>
<tr>
<td>T5I2D500K</td>
<td>First</td>
<td>SPO</td>
<td>SPO</td>
<td>SPO</td>
</tr>
<tr>
<td></td>
<td>Second</td>
<td>RICSPO</td>
<td>RICSPO</td>
<td>RICSPO</td>
</tr>
<tr>
<td></td>
<td>Last</td>
<td>SPOI</td>
<td>SPOI</td>
<td>SPOI</td>
</tr>
<tr>
<td>T10I4D100K</td>
<td>First</td>
<td>SPO</td>
<td>RICSPO</td>
<td>RICSPO</td>
</tr>
<tr>
<td></td>
<td>Second</td>
<td>RICSPO</td>
<td>ALL</td>
<td>ALL</td>
</tr>
<tr>
<td></td>
<td>Last</td>
<td>RC</td>
<td>KWJ</td>
<td>KWJ</td>
</tr>
</tbody>
</table>

Table 5 Trends in IBM DB2/UDB
6. Conclusion and Future Work

In this paper, we have explored the various optimizations and their combinations for the SQL–92 implementation of the basic Kwj for support counting phase of the association rule mining. We have analytically and experimentally compared these optimizations in an attempt to provide better insight to the effect of these optimizations on the total mining time for relations with varying characteristics and changing support values. Although combination of individual optimizations makes sense intuitively, our analysis and performance evaluation clearly indicates that it is not a given. Also, depending upon the storage available different choices of optimization may have to be used by the mining optimizer.

From most of these experimental results it seems that the best optimization is the reuse of item combinations or reuse of item combinations when combined with second pass optimization when we have enough space for materializing the intermediate relations (Comb_k). But when additional space is the issue then second pass optimization is the best approach. On the other hand for low support values, use of pruned input along with reuse of item combinations was found to be the worst combination for most of the input tables.

The work presented here tries to build this metadata by considering the different optimizations to the basic k-way join approach to association rule mining. A natural extension to this work would be trying to mix these optimizations at different passes. The other possibility would be to use the SQL-OR features provided by these commercial RDBMS’s and develop association rule mining algorithms to use them efficiently. We can then try evaluating these mixed approaches and the SQL-OR based optimizations with the current SQL92 based implementations and if they are comparable, we can include them in the metadata.

7. References


5. Han, J., J. Pei, and Y. Yin. *Mining Frequent Patterns without Candidate Generation.* in *ACM SIGMOD Int'l Conference on Management of Data.* 2000. Dallas.


