PERFORMANCE EVALUATION AND ANALYSIS OF

SQL BASED APPROACHES FOR

ASSOCIATION RULE MINING

The members of the Committee approve the masters thesis of Pratyush Mishra

Sharma Chakravarthy _____
Supervising Professor

Diane Cook _____

Alp Aslandogan _____

PERFORMANCE EVALUATION AND ANALYSIS OF SQL BASED APPROACHES FOR ASSOCIATION RULE MINING

by

PRATYUSH MISHRA

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

Dec 2002

To My Parents, Family and Friends

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Dr. Sharma Chakravarthy, for giving me an opportunity to work on this challenging topic and providing me ample guidance and support through the course of this research.

I would like to thank Dr. Diane Cook and Dr. Alp Aslandogan for serving on my committee.

I am grateful to Raman Adaikkalavan, Sreekant Thirunagari, Nishanth Vontela, Nellainayagam Subramaniam, and Arvind Mysore for their invaluable help and advice during the implementation of this work. I would like to thank all my friends in the ITLAB for their support and encouragement.

I would like to acknowledge the support by the Office of Naval Research, the SPAWAR System Center-San Diego & by the Rome Laboratory (grant F30602-01-0543), and the NSF (grants IIS-012370 and IIS-0097517) for this research work.

I would also like to thank my family members for their endless love and constant support throughout my academic career.

Nov 04, 2002

ABSTRACT

PERFORMANCE EVALUATION AND ANALYSIS OF

SQL BASED APPROACHES FOR

ASSOCIATION RULE MINING

Publication No.____

Pratyush Mishra, M.S.

The University of Texas at Arlington, 2002

Supervising Professor: Sharma Chakravarthy

Data mining aims at discovering important and previously unknown patterns from the datasets. Database mining performs mining directly on data stored in Data Base Management Systems. Several SQL based approaches for mining have been studied in the literature.

The main focus in this thesis is on the performance evaluation of these approaches. We study several additional optimizations for the K-way join approach and SQL-OR based approaches and evaluate them using IBM DB2/UDB and Oracle RDBMSs. We experimentally evaluate these approaches and their optimizations and compare their performance on large data sets. We also present analytical evaluation for the K-way join approach and its optimizations. 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 mining optimizer that chooses the specific strategy for mining the input dataset based on its characteristics.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

The rapid improvement in the size of the storage technology with associated drop in the storage cost, and increase in the computing power has made it feasible for organizations to store unprecedented amounts of organizational data and process it. These organizations, though having a gold mine of data, have not yet been able to fully capitalize on its value. Typically, the data captures the business trends over a period of time. However, the nuggets of useful knowledge hidden are not so easy to discern. To compete effectively in today's market, decision makers need to identify and utilize this information buried in the collected data and take advantage of the high return opportunities in a timely fashion.

Given a database of sufficient size and quality, data mining technology can generate new business opportunities by providing a better insight to the business, based on the collected information. The key here is the generation of previously unknown knowledge from huge datasets. The process of mining is driven by the outcome requirements. Based on what we want, a specific data mining technique is employed. The different data mining techniques and their outcomes are briefly discussed below [1]:

Classification: This is a process of grouping items based on a classifying attribute. A model is then built based on the values of other attributes to classify each item to a particular class. A training dataset is typically used for validating and tuning the model. The classification technique may be used, for example, to identify the most probable consumers for a product, based on their spending patterns.

Clustering: The process of clustering tries to group the data set in such a way that the data points in one cluster are more similar to one another while the data points in different

clusters are more dissimilar. A similarity measure needs to be defined and the quality of the outcome, to a large extent, depends on the appropriateness of the similarity measure for the data set or the domain of application. The technique of clustering, for example, can be used to divide the market into distinct groups, so that each group can be targeted with a different strategy.

The basic difference between classification and clustering is that in classification, the classifying class is known previously (also known as supervised), while clustering does not assume any knowledge of clusters (unsupervised).

Prediction: The technique of prediction is based on some continuous valued attributes. Previous history of the attributes is used to build the model. This technique is very commonly used for the prediction of sales of a product.

Deviation analysis: This technique compares current data with previously defined normal values to detect anomalies. Deviation analysis tools may be useful in security systems, where it may warn the authorities if there is any sharp deviation in the usage of resources by a particular user.

Association Rules: It is the process of identifying the dependency of one item(s) with respect to the occurrence of other item(s). 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 a product “A” with another product “B” to an extent that it can be said that whenever “A” is bought, “B” is also bought with high confidence (the number of times B occurs when A occurs).

The work in the field of data mining has resulted in a wide range of architectural alternatives for integrating mining process with the DBMS. These alternatives are depicted in Figure 1.1[2] and are described below.

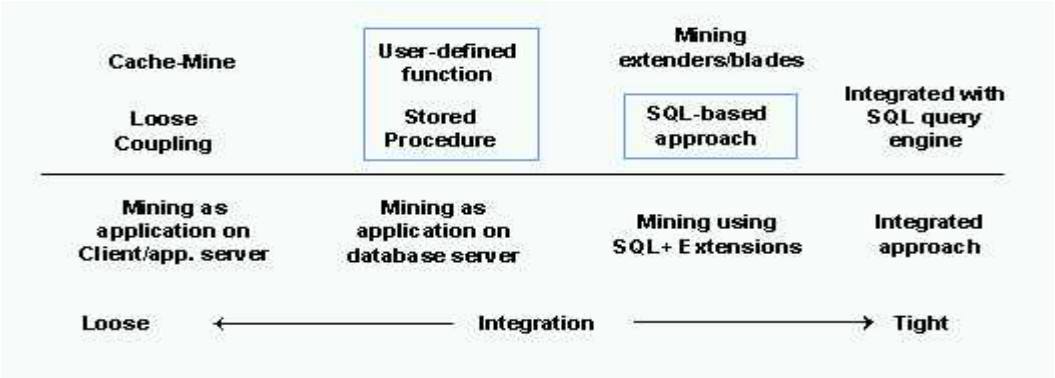

Figure 1.1 Architectural Alternatives

Loose Coupling or Cache based Mining: It's an example of the client/server architecture. The mining kernel can be considered as the application server. Here the data is first fetched from the database and fed to the mining-kernel, which mines and pushes the results back to the database.

Stored procedures and user defined functions: Here, mining logic is embedded as an application on the database server. The applications are executed in the same address space as the DBMS. The flexibility in programming the stored procedure out-weighs their development cost.

SQL based approach: Here, for mining, queries are written in SQL. A mining-aware optimizer may be used to optimize these complex, long running queries based on the mining semantics.

Integrated Approach: This is the tightest form of integration that has no boundary between querying, OLAP, or mining. Mining operators or SQL extended for mining is optimized by the underlying system without any hints from the user. The long-term goal is to extend the current query optimizers to cover OLAP and mining along with SQL queries.

The scope of this thesis lies within the realm of association rule mining over relational database management systems. Here two architectural alternatives of data mining have been explored and compared. These are – approaches using stored procedures and user defined functions (SQL-OR) and approaches based on pure SQL (SQL-92 standard). A more detailed explanation of these alternatives is given in later chapters.

1.1 Background

The work on association rule mining started with the development of the AIS algorithm [3], and was further modified and extended in [4]. Since then, there have been several attempts in improving the performance of these algorithms. The partition algorithm [5] improves the overall performance by reducing the number of passes needed over the complete database to at most two. The turbo-charging algorithm [6] incorporates the concept of data compression to boost the performance of the mining algorithm. The FP-Tree algorithm [7] builds a special tree structure in main memory to avoid multiple passes over database. However, most of these algorithms are applicable to data stored in flat files. The basic characteristics of these algorithms are that they are main memory algorithms, where the data is either read directly from flat files or is first extracted from the DBMS and then processed in 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 better suited to that algorithm. There have been very few attempts, until now, to build database-based mining approaches. In this approach we assume that the data is already stored in tables in a underlying DBMS and we use the SQL provided by the RDBMS for mining to produce interesting rules. SETM [8], 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 overload the SQL with certain mining operators. The Data Mining Query Language DMQL [9] proposed a collection of such operators for classification rules, characteristics rule, association rules, discriminant rules, etc. Meo et al [10] proposed the *MineRule* operator for generating general/clustered/ordered association rules. Agrawal and Shim [11] presents a methodology for tightly-coupled integration of data mining applications with a relational database system. Sarawagi et al [12], 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 earlier 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. Sarawagi et al [12] and Thomas [2] 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 possible combinations for optimizations were not explored. Also the optimizations to SQL-OR based approaches have not received much attention. In this thesis, we will analyze these optimizations in detail both analytically and experimentally. We analyze 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 [13]. To make the implementation operating system independent, we have used Java and JDBC API's. For the purpose of our evaluation, we have run the experiments on both Oracle 8i and IBM DB2/UDB.

1.2 Focus of This Thesis

With increase in the 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. Main memory always imposes a limitation on the size of data that can be processed. However using RDBMSs provides us the benefits of using their buffer management systems specifically developed for freeing the user/applications from the size considerations of the data. Building mining algorithms to work on RDBMSs 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 are those that work on data outside the database. They generally have an upper limit on 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 thesis is on the use of SQL and some of the Object Relational constructs provided by these RDBMSs for association rule mining. We have tried both: SQL-OR implementations (using user defined functions for DB2 and Java stored procedures for Oracle) and implementation using SQL-92 standards for association rule mining.

The goal of this thesis is to study these approaches for association rule mining and explore additional performance optimizations to these approaches. Based on the performance

evaluation of various approaches, we plan on generating heuristics that will help us select an approach that is better suited from among the approaches available. The other goal of our work is to consolidate the heuristics as metadata that can be used by a mining optimizer. Most of the relational query optimizers are not designed to optimize queries that are typically used for mining. Also, 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 approach and optimizations for 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 (as they are only based on cardinality, support and underlying RDBMS, not on the semantics of the data set) in understanding the approaches and can certainly be converted into meta-data for its use by the mining application.

Figure 1.2 shows the proposed architecture. At the heart of the architecture is the mining optimizer. This optimizer uses metadata, which is inferred from analytical evaluation and the results obtained from a series of experiments on the various SQL formulations of the apriori algorithm [12], [2], [13] and its optimizations.

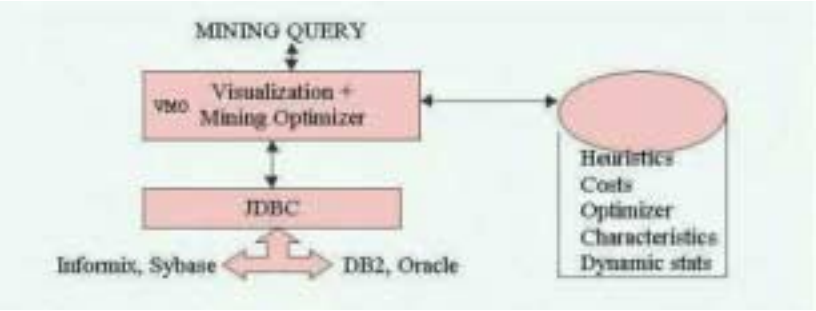

Figure 1.2 Proposed Architecture

The rest of this thesis is organized as follows. Chapter 2 introduces the apriori algorithm and different SQL formulations of it for association rule mining. Chapter 3 covers SQL-92 based approaches for support counting. It also covers in detail the basic k-way join method for support counting and its basic optimizations along with their performance analysis. SQL-OR based approaches and their possible optimizations are covered in Chapter 4. Chapter 5 includes extensions done in building this mining tool. Chapter 6 concludes the thesis with emphasis on the future work.

CHAPTER 2

ASSOCIATION RULE MINING

Association models examine the extent to which values of one field depend on, or are predicted by the values of another field. The rules discover items that "go together". The rules have a user-stipulated support, confidence, and length.

The problem of association rule mining was formally defined by Srikant and Agrawal [4]. 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). There are two terms associated with association rules. These are: “Support” and “Confidence”. If the support of itemset {AB} is 30%, it means “30% of all the transactions contain both the itemsets – itemset A and itemset B”.

$$\text{Support of itemset } \{AB\} = \frac{\text{Number of times itemsets A and B where bought together}}{\text{Total Number of Transactions}}$$

And if the confidence of the rule A $\Rightarrow$ B is 70%, it means “70% of all the transactions that contain itemset A also contain itemset B”.

$$\text{Confidence of the rule A} \Rightarrow \text{B} = \frac{Support(\{AB\})}{Support(\{A\})}$$

In this chapter, we discuss SQL-92 and SQL-OR formulation [2], [13] for the generic apriori algorithm. An association rule-mining problem is broken down into two sub-problems. 1) generate all the item combinations (itemsets) whose support is greater than the user specified minimum support. Such sets are called the frequent itemset and 2) use the identified frequent itemsets to generate the rules that satisfy a user specified confidence.

2.1 Apriori Algorithm

The apriori algorithm is based on the above-mentioned steps of frequent itemset and rule generation phase. Frequent itemsets are generated in two steps. In the first step all the possible combination of items, called the candidate itemset (C_k) is generated. In the second step, support of each candidate itemsets is counted and those itemsets that have support values greater than the user specified minimum support form the frequent itemset (F_k). The algorithm is depicted below.

```
F1 = {frequent 1-itemsets}
for (k = 2; Fk-1 ≠ 0; k++) do
     Ck = generate(Fk-1)
     for all transactions t ∈ D do
          Ct = subset(Ck, t)
          for all candidates, c ∈ Ct do
               c.count++
          end for
     end for
     Fk = { c ∈ Ck | c.count ≥ minsup}
end for
Answer = ∪k{Fk}
```

2.2 Candidate Generation

For SQL formulation, the database is represented as a relation with 2 attributes: Tid and item. Multiple tuples of this transaction relation represent the items associated with a single transaction. Candidate and frequent itemsets are represented as relations containing a set of attributes, each representing an item. In the k^{th} pass, the set of candidate itemsets C_k is generated from the frequent itemsets F_{k-1} (generated in the $(k-1)^{th}$ pass) as shown below:

```
Insert    into Ck
Select    I1.item1, … ,I1.itemk-1, I2.itemk-1
```

```
From      F(k-1) I1, F(k-1) I2
Where     I1.item1 = I2.item1 and
                  :
          I1.item (k-2) = I2.item(k-2) and
          I1.item (k-1) < I2.item(k-1)
```

The number of candidate itemsets generated in each pass, by the above step is reduced by pruning out all itemsets $c \in C_k$ where some (k-1)-subsets (itemsets of length k-1) of c are not in F_{k-1}. This is based on the subset property that in order for an itemset to be a frequent item, all subsets of that itemset have to be frequent. In the generated itemset of length k, two of its itemsets of length k-1 are frequent itemsets since the itemset of length k was generated from these two itemsets. The remaining subsets (itemsets) of length k-1 are validated for memberships. This is done by additional join predicates, which skip one item at a time from the k-itemset. The tree diagram for this process is shown below.

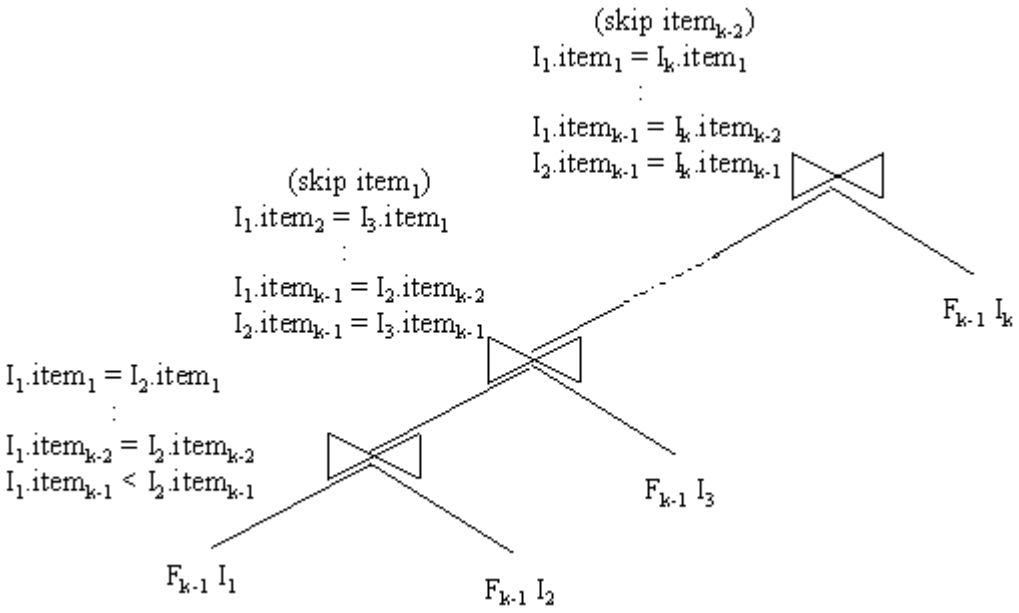

Figure 2.1 Candidate Generations for Any k

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}}. In the prune step, all itemsets $c \in C_k$, where some (k-1) subset of c is not in F_{k-1} as mentioned before are deleted. Thus the prune step will delete the itemset {1 3 4 5}. The itemset {1 3 4 5} is known to be generated from the subsets {1 3 4} and {1 3 5}. However, the subset {3 4 5} is not in F_3. Hence it is deleted and C_4 contains only {1 2 3 4}.

2.3 Support Counting

This is an important and most time-consuming part of the mining process. This step is needed to identify all the frequent itemsets from the set of candidate itemsets. In the following section we present 3 methods using the SQL-92 standard for support counting and 3 methods using SQL-OR constructs for support counting.

2.3.1 Support Counting Using Simple SQL

This consists of writing SQL (conforming to SQL-92 standards) for counting support of the candidate itemsets. The approaches based on SQL-92 standard are discussed in the ensuing sections.

2.3.1.1 K-way Join (Kwj)

The basic approach for support counting is that for 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 is needed to identify all itemsets whose count is > user specified threshold value, as frequent itemsets, which are then considered potential itemsets for the rule generation phase. The SQL

statement and the tree diagram for support counting with k-way join approach are shown below.

```
Insert    into Fk
Select    item1, … , itemk, count(*)
From      Ck, T t1, … , T tk
Where     t1.item = Ck.item1 and
                  :
          tk.item = Ck.itemk and
          t1.tid = t2.tid and
                  :
          tk-1.tid = tk.tid
Group by  item1,  item2, … ,itemk
Having    count(*) > minsup
```

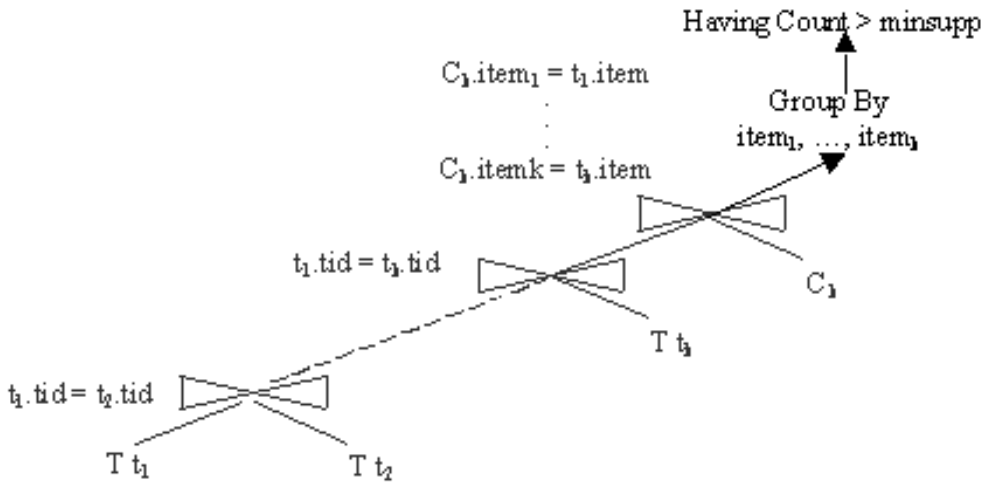

Figure 2.2 Support Counting by K-way Join Approach

2.3.1.2 2-Group By (Tgb)

This approach avoids the multi-way joins of the k-way join approach. Here we join the candidate itemset C_k and the input table T once. The join condition checks whether the

item present in the input table T is same as any of the k items in the candidate itemset C_k. If so, then group all such items, i.e., do a group by on the ($item_1$, $item_2$, … , $item_k$, tid) with a filtering condition that the count of such items is equal to k. The result is a set of items and their tid such that this tid supports the itemset in C_k. Once all such itemsets are identified, a group by on each item ($item_1$, $item_2$, …, $item_k$) of these itemsets is done. Those itemsets whose count > user specified threshold value, are put in the frequent item list. The SQL statement used in this approach is shown below.

```
Insert    into Fk
Select    item1, item2, … , itemk, count(*)
From      (Select   item1, item2, … , itemk, count(*)
           From     T, Ck
           Where    item = Ck.item1 or
                         :
                    item = Ck.itemk
           Group by item1, item2, … , itemk, tid
           Having count(*) = k
           ) As temp
Group by  item1, item2, … , itemk
Having    count(*) > minsup
```

In this approach, for support counting, the candidate itemset relation is joined only once with the input relation and hence the number of joins in any pass k are comparatively lesser than the k-way join. But then this approach uses the “OR” operator for comparing the itemsets along with two group by and having clauses (once during the grouping of the items in the input table and the second time for the actual support counting) for identifying frequent itemsets of length k. Since the “OR” operator does not lend itself for optimization, using the “OR” operator along with two group by clauses turns out to be very time consuming.

2.3.1.3 Query Sub Query (Qsq)

This approach makes use of the common prefixes for support counting. There are lots of intermediate sub-queries generated in this approach. We will denote sub-queries by Q_i, meaning that it is the i^{th} sub-query. Note that there is no sub-query Q_0. Subquery Q_i will select items from sub-query Q_{i-1} and relations C_i and input table T. The condition being that the items in Q_{i-1} match with those in C_i and tid in T matches with that in Q_{i-1}. Thus in any pass, say m, (m-1) items ($item_1$, $item_2$, … , $item_{m-1}$) in Q_{m-1} are matched with (m-1) items in C_m. Also tid in T and Q_{m-1} are matched. If all of them match, and there support > user specified threshold value, then all such items are inserted into F_k.

The tree diagram for the Sub-Query Q_i and the corresponding SQL statements are shown below.

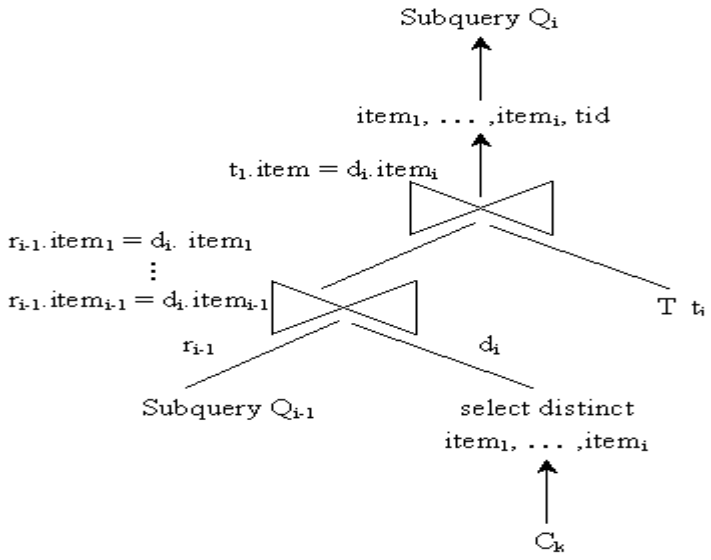

Figure 2.3 Tree Diagram for Sub-Query Q_i

```
Insert    into Fk
Select    item1, item2, … , itemk, count(*)
From      (Subquery Qk) t
Group by  item1, item2, … , itemk
Having    count(*) > minsup

Subquery Qi (1 ≤ i ≤ k):
Select    item1, item2, … , itemi, tid
From      T ti, (Subquery Qi-1) as ri-1,
          (Select distinct item1, item2, … , itemi
           From Ck) As di
           Where    ri-1.item1 = di.item1 and
                              :
                    ri-1.itemi-1 = di.itemi-1 and
                    ri-1.tid = ti.tid and
                    ti.item = di.itemI
```

Figures below compare the performance of these approaches on Oracle and IBM DB2/UDB. The methodology of performing these experiments is detailed in section 3.2.2.

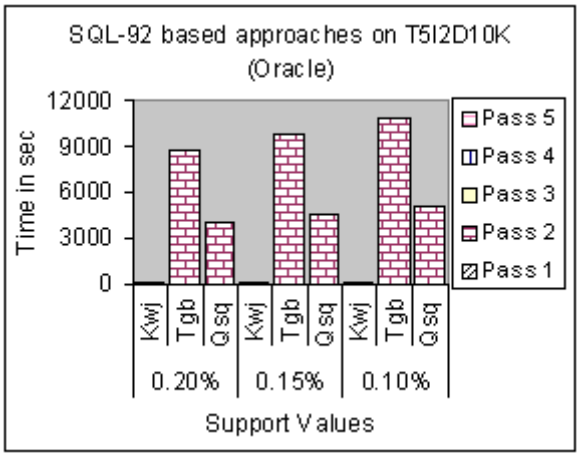

Figure 2.4 SQL-92 Based Approaches (Oracle)

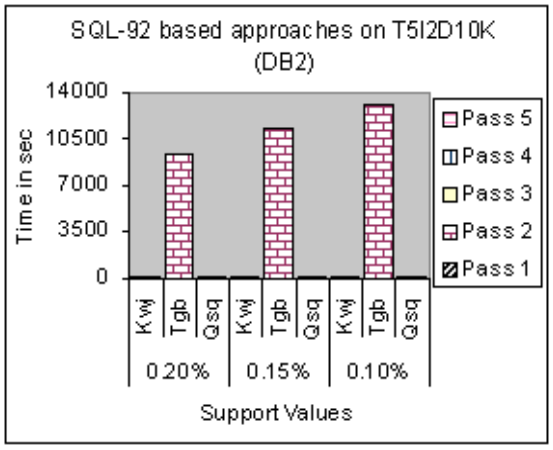

Figure 2.5 SQL-92 Based Approaches (DB2)

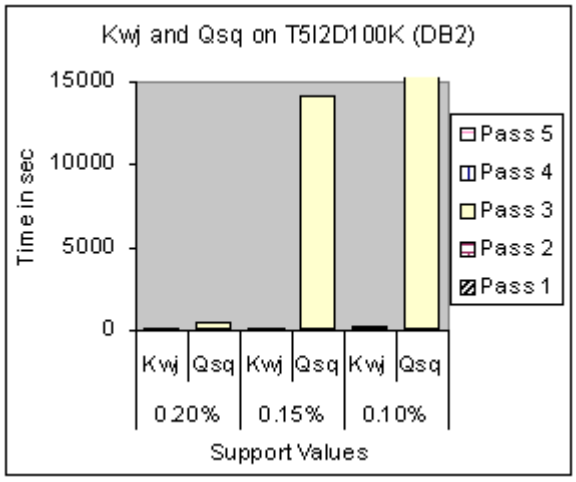

Figure 2.6 Kwj and Qsq on T5I2D100K (DB2)

Figure 2.4 compares the overall time required for mining dataset T5I2D10K using Kwj, Tgb and Qsq with varying support values on Oracle. Figure 2.5 shows the same for DB2. From Figure 2.4 it is clear that Kwj is the best on Oracle. On DB2, for dataset T5I2D10K, the time taken by Kwj and Qsq is nearly same while the time taken by Tgb is very high hence the same cant be said from the Figure 2.5. Figure 2.6 compares Kwj and Qsq for dataset T5I2D100K on DB2. Here for support value of 0.10%, Qsq didn't complete even after running for 9 hrs. From this figure, it is clear that, Kwj is much better than the Qsq. Similar trend has been found on both – Oracle and DB2 for other datasets.

2.3.2 Frequent Itemset Generation Using Stored Procedures and User Defined Functions

This section describes queries, in which stored procedures and user defined functions are written to enhance the candidate itemset and support counting phase. Here, at times, some SQL-OR constructs are used (such as CLOBs) for better representation of input data. For Oracle, all stored procedures have been implemented as a Java stored procedures and for

IBM DB2/UDB, the same has been implemented as user defined functions (or UDFs) using Java. In this section the word "*procedure*" is used in general for both – user defined functions for IBM DB2/UDB and stored procedures for Oracle, unless otherwise stated.

2.3.2.1 VerticalTid (Vtid)

The SQL-92 based approaches assumed that the input table (the one to be mined) has following attributes: (Tid, Item). In the VerticalTid approach, the representation of input data is changed and the transactions are inserted in a different relation (TidListTable) having following attributes: (Item, TidList). For every unique item id in the input dataset, the TidListTable has only one tuple. This tuple represents the item id and the list of all the transactions in which that item was bought. Each list of transactions is represented as a CLOB and stored in the TidList column of the TidListTable.

For the purpose of support counting, procedures are used to read these CLOBs and for each item combination (itemset), count the number of same transaction ids that are present in the TidList of each item id in that itemset. If the count > the user specified minimum support value, then the itemset is considered as frequent and is used to generate item combinations of length one more than itself in the subsequent pass. The SQL for generation of frequent itemsets is given below.

```
Insert    into Fk
Select    item1, item2, …, itemk
From      (Select   item1, item2,…, itemk,
                    CountAndK(I1.TidList,I2.TidList, … ,
                    Ik.TidList) as cnt
          From      Ck, TidListTable I1, TidListTable I2,…,
                    TidListTable Ik,
          Where     Ck.item1 = I1.item And
                    Ck.item2 = I2.item And
```

```
                                    :
                                    :
                          Ck.itemk = Ik.item) as temp
Where     cnt > minsup.
```

Here ***CountAndK*** is a procedure that in pass k, accepts k TidLists and returns the count of transactions that are common in each of them.

2.3.2.2 Gather Join (Gjn)

This method differs in the way candidate itemsets are generated and the way in which support counting of these candidate itemsets is done. This approach also uses a different representation for the input table. Here the input table is read and for each unique transaction all the items bought in that particular transaction are collected together. For implementation on DB2, we use CLOBs to represent the list of items bought in a transaction and use a different table (ItemListTable) having the attributes (Tid, ItemList) to materialize it. For Oracle, this is implemented as a stored procedure, which makes a single pass over the input table and collects all the items bought in a given transaction in a vector. The vector is then used directly for generation of candidate itemsets. The ItemListTable is not materialized. In each pass k, procedures are used to read this collection of items (ItemList column of the ItemListTable if it is materialized, else the vector of items) and generate item combinations of length k, which are then inserted in the candidate itemsets table C_k. For support counting, since these candidate itemsets of length k, are generated from the items bought in any transaction, there is no need to join C_k with k copies of input table (as is done in SQL-92-based approaches), rather a simple “Group by” on the k-items of the candidate itemset is sufficient to identify those itemsets that have count > user specified minimum support. The JDBC calls and the SQL used for generation of candidate itemsets in Oracle and DB2 is shown below:

```
try
{
     Connectin con;
     CallableStatement cStmt;
     String qs = "{Call CombinationK(?)}";
     cStmt = con.prepareCall(qs);
     cStmt.setString(1,”InputTable”);
     cStmt.execute ();
     cStmt.close();
}
catch (Exception e)
{
     e.printStackTrace();
}
```

The above JDBC calls are for Oracle. Here a call to ***CombinationK*** stored procedure is made to generate all item combinations of length k.

For DB2 CombinationK procedure has been implemented as an udf. The call to udf is made as shown below:

```
Insert    into Ck
Select    item1, item2,…, itemk
From      (select *
           From ItemListTable) as T1,
               table(CombinationK(tid, ItemList)) as T2;
```

Here the keyword “***table***” means that the CombinationK udf is a table udf that returns a table. So here the CombinationK udf returns a table consisting of candidate itemsets of length k. The SQL for generating frequent itemsets from the above generated candidate itemsets is given below.

```
Insert    into Fk
Select    item1, item2,…, itemk, count(*)
From      Ck
Group by  item1, item2,…, itemk
Having    count > minsup;
```

2.3.2.3 Gather Count (Gcnt)

This approach is very similar to the Gather join approach. This approach also uses the same stored procedures for generating candidate itemsets of length k. The only departure from the Gather join approach is in the second pass. In the second pass, instead of directly inserting the generated candidate itemsets into table C_2, a two-dimensional array, representing the count of all 2-item combinations is built. At the end of the pass, the two-dimensional array is read and only those item combinations, whose count is greater than the user specified minimum support value, are inserted in the frequent itemsets table (F_2). This has been implemented for Oracle, by modifying the Combination2 stored procedure (called GatherCount2 stored procedure). Because of memory constraints, the same could not be implemented for pass 3 and higher passes (we will see this in details in Chapter 4). So this approach, uses the same stored procedures as used by the Gather join approach for pass 3 and onwards. Similar modification could not be done in the udf's for DB2. The JDBC code below shows the calls made to different stored procedure for support counting.

```
try
{
     CallableStatement cStmt;
     Connection con;
     String qs= “”;
     if(k==2) // call the GatherCount2 stored procedure
     {
          qs = "{Call GatherCount2(?,?)}";
          cStmt = con.prepareCall(qs);
```

```
            cStmt.setString(1,”InputTable”);
            cStmt.setInt(2,minsup);
      }
      else // call the CombinationK stored procedure
      {
            qs = "{Call CombinationK(?)}";
            cStmt = con.prepareCall(qs);
            cStmt.setString(1,"InputTable");
      }
      // call the stored procedure
      cStmt.execute();
      cStmt.close();
}
catch (Exception e)
{
      e.printStackTrace();
}
```

CHAPTER 3

SQL-92 BASED APPROACHES

In the previous chapter, we described different approaches to generate candidate itemsets and count their support. Each of these approaches has their own advantages and disadvantages. In this chapter we will revisit the SQL-92 based approaches in more detail. The outline of this chapter shall be as follows: Section 3.1 will bring out the differences, advantages and disadvantages of different approaches for support counting. Section 3.2 will enumerate various optimizations to the k-way join approach for support counting and evaluate them analytically and experimentally. In Section 3.3 we will observe the effects of combining these individual optimizations. We conclude this chapter in section 3.4, with a summary of performance gained due to these optimizations.

3.1 Support Counting Revisited

The simplest way of support counting is the **k-way join** approach. Here in pass k, k– copies of input table are joined for counting the support of all the candidate itemsets of length k. For datasets of small sizes, or for low values of k, this approach works fine, but for large datasets or for low support values resulting in substantial number of passes, self-join of the input dataset need to be performed k times and is very time consuming. The **2-Group By** approach tries to overcome this problem. But with this approach also, there are certain advantages and disadvantages. This approach gets around the problem of multi-way joins and the number of joins are comparatively less than the k-way join, but this approach suffers from the overhead involved in the comparisons made using the “OR” operator and executions of group by and having clauses. In fact, the group by and the having clauses have to be executed twice – once during the grouping of the items in the input table and the second

time for the actual support counting. Experiments on input tables with different characteristics have shown that the execution of the group by clause twice, is more time consuming than the k-way join of the input table.

The **Query Sub Query** approach for support counting makes use of common prefixes for support counting. In any pass k, a sub-query (Q_{k-1}) is generated to find out the frequent items of length k-1. The sub-query Q_{k-1}, in turn calls subquery Q_{k-2}. This continues, until subquery Q_1 is executed which returns all frequent 1-itemsets. The result obtained from the subquery Q_1, is streamed to subquery Q_2 for the generation of frequent itemsets of length 2. This process continues upwards till the last subquery Q_{k-1} is executed. Since the results from any subquery are not materialized, in any pass k, k-1 sub queries have to be executed. The work by S. Thomas [2] has reported that of all the SQL-92 based approaches this approach for support counting has the best performance. But from the various experiments that we have done – both on IBM DB2/UDB and Oracle, we have found that the performance of k-way join is the best. Figure 2.4, Figure 2.5 and Figure 2.6 shows this.

3.2 Analysis of K-way Join Approach And its Optimizations

Of all the SQL-92 based approaches, k-way join has been found to be the best. So this section will cover the analysis of the possible optimizations to the basic k-way join and their implications. The purpose of these optimizations and their analysis (along with performance evaluation) is to understand the impact of various optimizations on datasets of different characteristics (size, average transaction length, support, confidence etc.). The purpose of this analysis is to obtain heuristics that relate various optimization techniques and their effect on the dataset characteristics. Though not all the optimizations produce better timings, our conjecture is that the study of these optimizations can give us a better insight to the metadata

that can be used for making a mining-aware optimizer. In addition to providing analytical support for these optimizations, we will present the results obtained when datasets with different characteristics were mined using them.

3.2.1 Notations Used for Cost Analysis

The cost analysis of some of the approaches was done in Thomas [12]. We use similar notations for our study of these optimizations. These notations are described in the table given below.

Table 3.1 Notations Used for Cost Analysis of Different Approaches

R	Number of records in the input transaction table
T	Number of transactions
N	Avg. number of items per transaction = R/T
F_1	Number of frequent 1-itemsets
S(C)	Sum of support for each itemset in C
s_k	Average support of a frequent k-itemset = $S(F_k)/(F_k)$
R_f	Number of records out of R involving frequent items = $S(F_1)$
N_f	Average number of frequent items per transaction = 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: (n!)/(k!(n-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

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/UDB). 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.

3.2.2 Methodology for Experimental Evaluation

The performance results presented in this thesis 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 and IBM DB2 / UDB V7.2 (installed on Windows 2000 server with 512MB of RAM). Each experiment has been performed 4 times. The values from the first run are ignored so as to avoid the effect of the previous experiments and other database setups. The average of the next 3 runs 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. For most of the experiments, we have found that the percentage difference of each run with respect to the average is less than one percent. For SQL-92 based approaches, 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 [13]). 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 thesis, for most of the optimizations we have shown the results only for three datasets – T5I2D500K, T5I2D1000K and T10I4D100K. 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.

3.2.3 Cost Analysis of the Basic K-way Join Approach (Kwj)

For support counting, in any pass k, k copies of input table are joined with C_k (Figure 2-2). The total number of items produced in a join of C_k with T is equal to the number of records in C_k * average support of first item in C_k. Using the join notations, the join cost can be represented as join (C_k, R, C_k*s_1). Similarly, the join of m-copies of T with C_k will result in a table, containing the sum of support count for first m-items of an itemset in C_k. In terms of join notation this can be represented as join (C_k*s_{m-1}, R, C_k*s_m). The cost of last join, cannot be calculated from the above formula as the s_k 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 many tuples as the sum of support for each itemset in set C_k. Using the join notation this can be represented as join (C_k*s_{k-1}, R, $S(C_k)$). Hence the cost of any pass, for this approach can be given as:

$$\sum_{m=1}^{k-1} join(C_k * s_{m-1}, R, C_k * s_m) + join(C_k * s_{k-1}, R, S(C_k)) + group(S(C_k), C_k) \quad (3.1)$$

Figure 3.1 compares the time required for mining the relation T5I2D1000K on DB2, with break-up for each pass for support values of 0.20%, 0.15% and 0.10%, while Figure 3.2 shows the same on Oracle. (On DB2, for support value of 0.10%, the experiment did not complete even after running it for over 9 hrs). The analysis of time required for each pass shows that, of all the passes, second pass is the most time consuming. This is true as in

second pass ${}^{n}C_2$ (all combinations of two elements from frequent 1-itemsets) candidate itemsets are generated, where n is the cardinality of frequent-1 itemset.

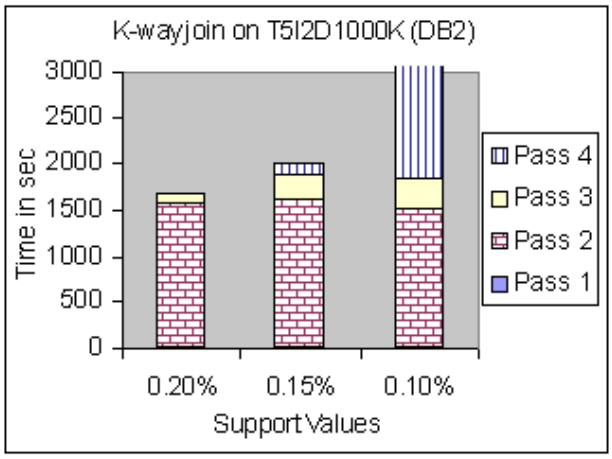

Figure 3.1 K-way Join on T5I2D1000K (DB2)

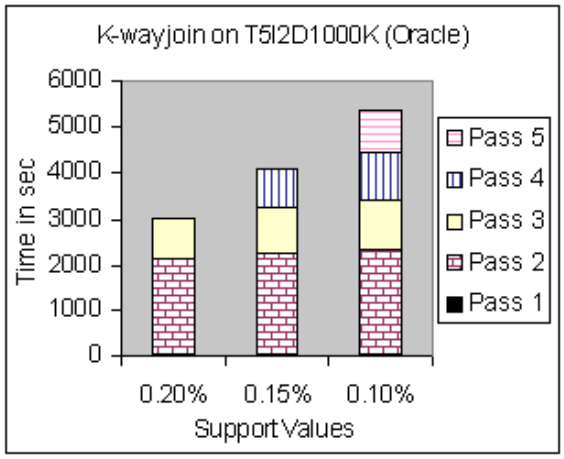

Figure 3.2 K-way Join on T5I2D1000K (Oracle)

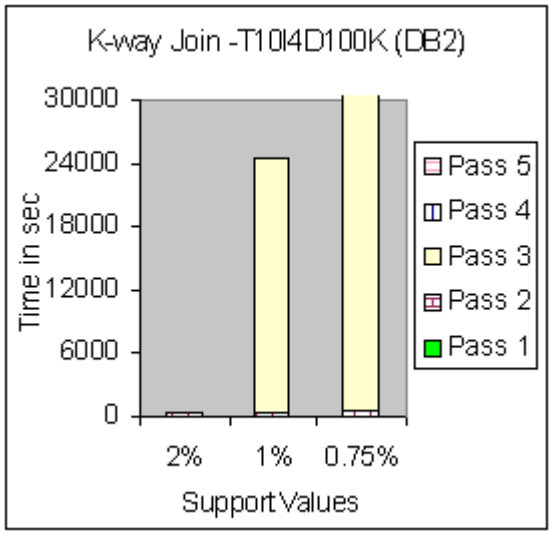

Figure 3.3 K-way Join on T10I4D100K (DB2)

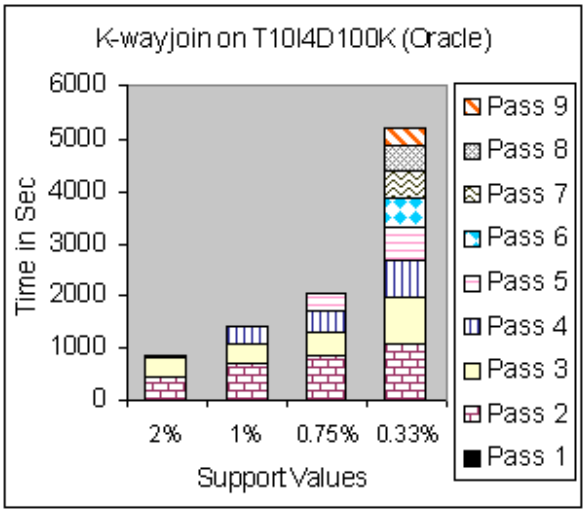

Figure 3.4 K-way Join on T10I4D100K (Oracle)

Figure 3.3 shows the time required for mining relation T10I4D100K for different support values on DB2, Figure 3.4 shows the same for Oracle and Table 3.2 shows the number of candidate itemsets generated in respective passes, when different tables were mined with different support values. (Here we have chosen the dataset T10I4D100K, because for this dataset, the experiment runs for 9 passes and we wanted to see how k-way join performs with the increase in the number of passes.) For DB2, for support value of 0.75% the experiment didn't complete even after running for 12 hours.

Table 3.2 Number of Candidate Itemsets in Different Passes

	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9
T5I2D500K. Sup = 0.10%	307720	126	7	0	--	--	--	--
T5I2D1000K. Sup = 0.10%	309291	127	61	0	--	--	--	--
T10I4D100K. Sup = 0.75%	12470	65	3	0	--	--	--	--
T10I4D100K. Sup = 0.33%	216153	2453	905	354	109	20	2	0

The analysis of theses figures shows that for mining configuration, where the length of the largest frequent itemset 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 datasets with long frequent itemsets, 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. In equation 3.1 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 3.2.4 discusses 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 3.2.5 discusses this second pass optimization in more detail.

Let us compare the SQL tree for support counting (Figure 2.2) for two successive passes, say pass 4 and pass 5. In the 4^{th} pass C_4 is joined with 4 copies of input table, to identify all frequent itemsets of length 4. In the 5^{th} 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 itemsets, 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 3.2.6 discusses about this optimization and its effects. Let us now compare the cost of the basic k-way join approach given by equation 3.1 with each of these optimizations, and then with their combinations. The ensuing section deals with this.

3.2.4 Pruning the Input Table (Pi)

As indicated earlier, reducing the size of the input table and using it in all the passes for support counting can improve the time for support counting. Eliminating the records of those single itemsets from it, whose support is lower than the user specified minimum value, can reduce the size of the input table. Instead of deleting these records, a new relation, T_f, having following attributes: (tid, item), is created to contain tuples of only frequent itemsets of length 1. This is done by generating F_1, as before. Then F_1 is joined with input dataset T

on the “item” column, and records of only those items whose support > user defined support value are inserted in the relation T_f. The SQL for creating the pruned input table is given below:

```
Insert    into Tf select t.tid, t.item
From      T t, F1 f
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:

$$join(R, F_1, R_f) + \sum_{m=1}^{k-1} join(C_k * s_{m-1}, R_f, C_k * s_m) + join(C_k * s_{k-1}, R_f, S(C_k)) + group(S(C_k), C_k) \quad (3.2)$$

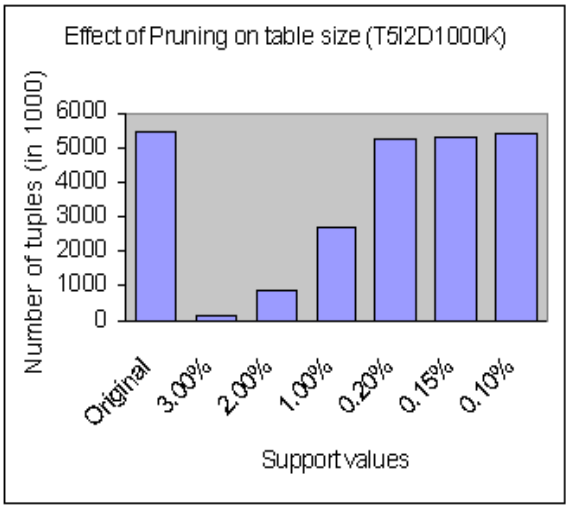

Figure 3.5 Reduction in Table Size Due to Pruning

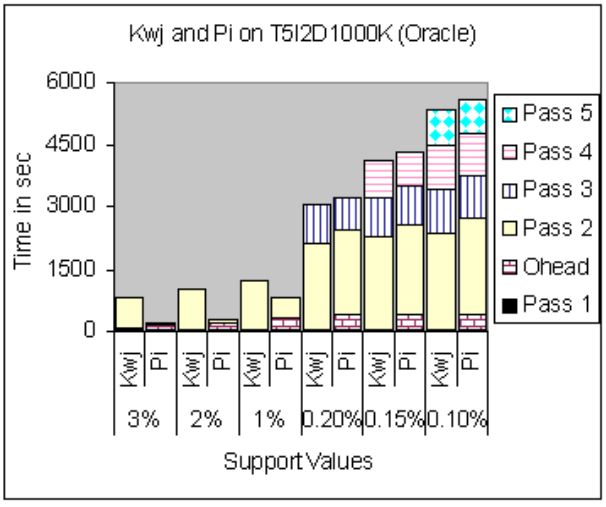

Figure 3.6 K-way Join and Pruned Input

The difference between equations 3.1 (Kwj) and 3.2 is that, in equation 3.1, an additional cost for materializing the pruned relation is involved. And then this pruned relation is used instead of the original dataset 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 3.5 shows the reduction in size of input table T5I2D1000K for different support values. It is evident from the figure that the reduction in the size of the table is very marked for higher support values. But, pruning might not always end up in giving better performance. Figure 3.6 compares the total cost of mining the relation T5I2D1000K on oracle, using the pruned relation (time for pruning also considered) with basic Kwj, for different support values. Its evident from this figure that for higher support values, (3.0%, 2.0% and 1.0%), the total time taken is less when pruned relation was used, but the reverse is true for lower support values. To explain this let us see the time taken by each pass.

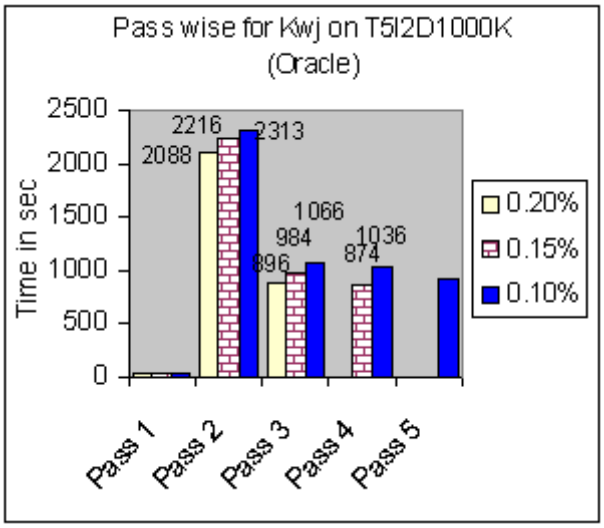

Figure 3.7 Pass Wise for K-way Join

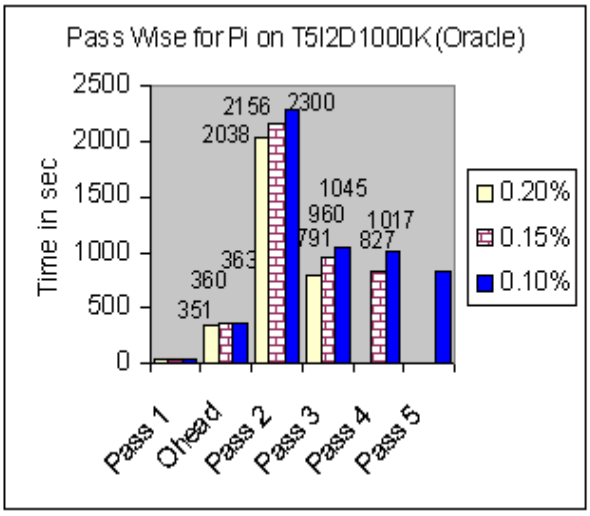

Figure 3.8 Pass Wise for Pruned Input

Figure 3.7 (for Kwj) and Figure 3.8 (for Pi) shows the cost of various passes for table T5I2D1000K for support values of 0.2%, 0.15% and 0.10%. From these figures, we see that the use of pruned relation hardly has any effect on the running time of any pass; rather, there is an additional cost involved in pruning (given by the SQL above, denoted as "Ohead" on the X-axis of the Figure 3.8). Because of this overhead, at low support values, the overall time of using pruned relation comes out more than the simple Kwj.

3.2.5 Second Pass Optimization (Spo)

As explained earlier and is also apparent from the figures shown above, that, of all the passes, second pass is the most time consuming. In general, 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. Also there are no rules associated with F_1. Hence the process of generating F_1 then C_2 followed by its support counting phase can be replaced by directly generating F_2. This is done by joining two copies of the input table, such that the item from first copy < item from the second copy and that both items belong to same transaction. The SQL for the same is as follows:

```
Insert    into F2 select t1.item, t2.item, count(*)
From      InputTable T1, InputTable T2
Where     T1.tid = T2.tid and T1.item < T2.item
Group by  T1.item, T2.item.
Having    count(*) > minsup
```

The cost of second pass is thus:

$$join(R, R, C(N_f, 2)) + group(C(N_f, 2), C(F_1, 2)) \quad (3.3)$$

Figure 3.9, compares the overall time required for mining table T5I2D500K using Kwj and Spo. We can see that though the cost of grouping is same as that in the Kwj, yet skipping of the support counting phase during the second pass results in a big reduction in the overall cost. For table T5I2D500K, the overall time required for mining it is reduced by 3 to 4 times when the second pass is optimized.

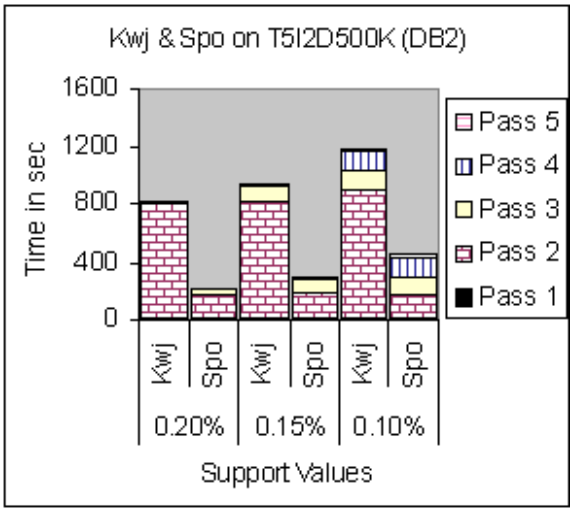

Figure 3.9 Kwj and Spo

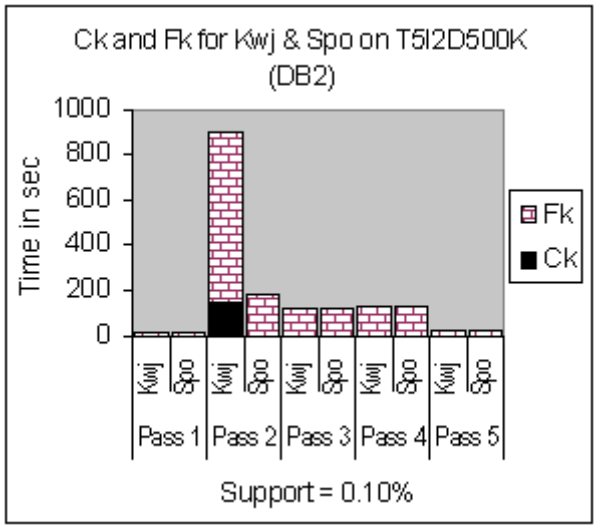

Figure 3.10 Ck and Fk for Kwj and Spo

Figure 3.10 compares the time taken for the candidate generation phase (C_k) and support counting phase (F_k) for the Kwj and the Spo. The values in Pass-2 of this figure shows that the improvement in performance is due to savings on the join cost at two stages. The first is by totally bypassing the generation of candidate itemset C_2 and the second is during the generation of the frequent itemset F_2. The reason for the improvement in the

second stage will become clearer by analyzing the costs of the SQL statements executed for the second pass. The cost of the second pass for the Spo is:

$join(R, R, C(N_f, 2)) + group(C(N_f, 2), C(F_1, 2))$

and for the k-way join is:

$join(C_2, R, C_2 * s_1) + join(C_2 * s_1, R, S(C_2)) + group(S(C_2), C_2)$.

Thus in Spo, for generation of the frequent itemset F_2, input relation (T) is directly joined with itself, instead of joining three relations - C_2 with 2 copies of input relation (as is done in Pass-2 of Kwj), which results in decrease in the computation time of F_2.

3.2.6 Reuse of Item Combinations (Ric)

This optimization aims to reduce the cost of support counting, in any pass k, 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, and using it for support counting in the k^{th} pass. This saves from redoing the same sequence 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. So in k^{th} 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. The SQL for this is given below:

```
Insert    into Comb_k
Select    T_1.tid, T_1.item_1, T_1.item_2,…, T_1.item_k-1,T_2.item
From      C_k, Comb_k-1 T_1, T T_2
Where     T_1.item_1 = C_k.item_1 and
                  :
                  :
          T_1.item_k-1 = C_k.item_k-1 and
          T_2.item = C_k.item_k and
          T_1.tid = T_2.tid
```

F_k is then generated from $Comb_k$ by grouping on k items ($item_1$, $item_2$, …, $item_k$) and selecting those that satisfy the minimum support criteria. The SQL for this given below:

```
Insert    into Fk
Select    item1, item2, …, itemk
From      Combk
Group by  item1, item2, …, itemk
Having    count(*) > minsup
```

Let us analyze the cost of this optimization. Instead of joining the input table k times, in 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 notion of joins of two tables is extended to three tables as trijoin (p, q, r, s). Which means that relations having p, q, and 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:

$$trijoin\ (Comb_{k-1}, R, C_k, S(C_k)) + group\ (S(C_k), C_k)). \qquad (3.4)$$

Figure 3.11 compares the total time taken for mining the relation T10I4D100K using Kwj and Ric for different support values on Oracle. Figure 3.12 shows the same for DB2. For higher support values, the experiments runs for less number of passes and hence the cost of support counting using the Kwj, without materializing the intermediate results seems to do better. But for low support value, 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.

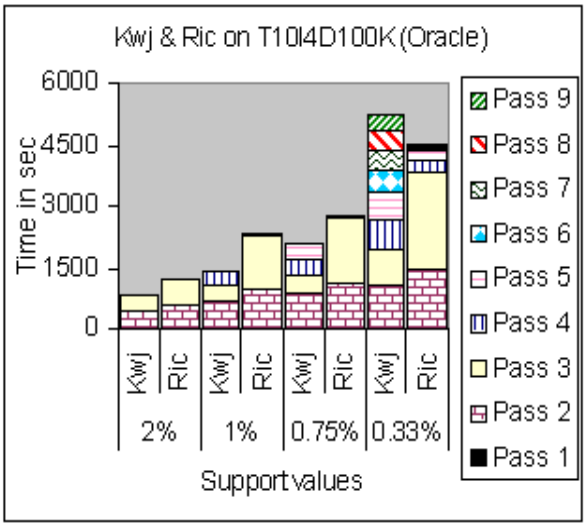

Figure 3.11 Reuse of Item Combination (Oracle)

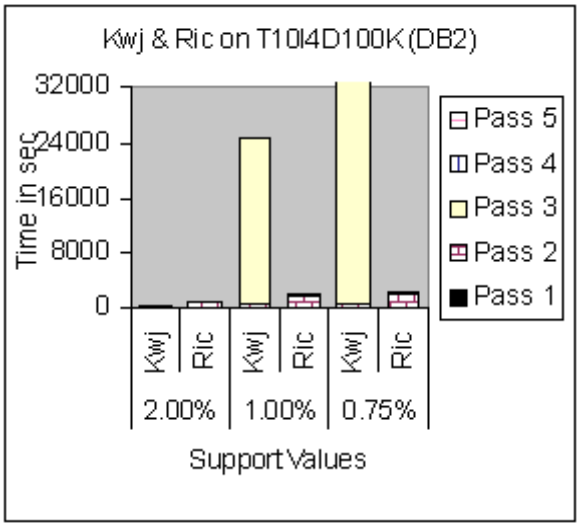

Figure 3.12 Reuse of Item Combination (DB2)

In Figure 3.12, for support value of 0.75%, Kwj does not complete, while Ric does. This can be explained more clearly by comparing equation 3.1 (for Kwj) and equation 3.4 (for Ric). In equation 3.1, for support counting of the 3rd pass, 4 relations are joined - 3 copies of the input table and C_3, while in equation 3.4 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, doesn't seem to complete (we ran the experiment on DB2 for 9 hrs.)

3.3 Combinations of Basic Optimizations

Sections 3.2.4, 3.2.5 and 3.2.6 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 additional optimizations obtained by combining these individual optimizations.

3.3.1 Second Pass Optimization on Pruned Input (SpoPi)

This combination of optimization uses pruned input along with second pass optimization. Although the Spo does result in some performance gain under all situations, the same is not true with Pi. Hence the overall performance obtained from this combination is limited by the overhead of pruning. Figure 3.13 compares the time taken for mining the table T5I2D500K using the Kwj, Pi, Spo and by SpoPi, for different support values on Oracle. For low support values, the overhead of building pruned input outweighs any performance gained due to optimization of the second pass. Hence at a low support value, SpoPi does better than when only pruned input is used, but its performance is worse than when only Spo is used. In all the experiments performed, the performance of this combination has not been very impressive.

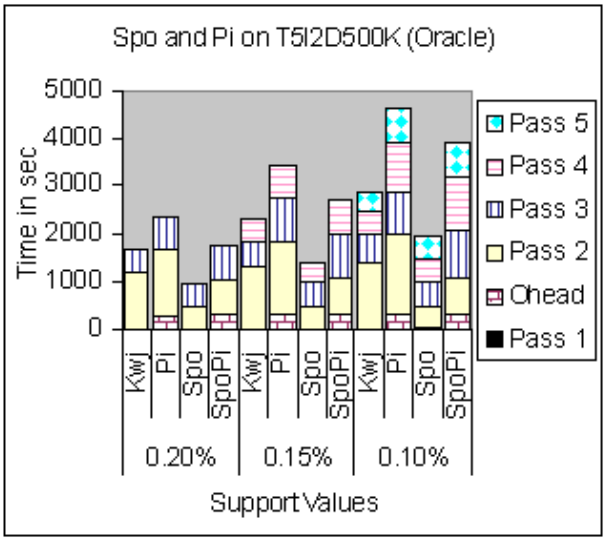

Figure 3.13 Spo on Pi

The overall cost of this optimization is thus same as the use of pruned input except for the second pass, where the cost of second pass is similar to Spo. This is given below:

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

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

$$\sum_{m=1}^{k-1} join(C_k * s_{m-1}, R_f, C_k * s_m) + join(C_k * s_{m-1}, R_f, S(C_k)) + group(S(C_k), C_k) \quad (3.5)$$

3.3.2 Reuse of Item Combinations on Pruned Input (RicPi)

This optimization is similar to the one discussed in the section 3.2.6, except that instead of using the input table as it is, non-frequent itemsets of length one are pruned out and then this pruned input table is used in all passes for joining with $Comb_{k-1}$ to produce $Comb_k$. The cardinality of $Comb_{k-1}$ is $S(F_{k-1})$. Thus the cost of generating frequent itemset for any pass is given as:

$$trijoin(Comb_{k-1}, R_f, C_k, S(C_k)) + group(S(C_k), C_k)) \quad (3.6)$$

The analysis of this combination shows that for most of the experiments at low support values, it **did not** produce the added performance of: (1) reusing the frequent itemsets generated in the previous pass and (2) using pruned relation. The overall performance for this combination is dominated either by the cost of building the pruned relation at low support values or by the cost of materializing the $Comb_2$ 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 in most cases, we do not 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 3.14 shows this for table T5I2D1000K on DB2.

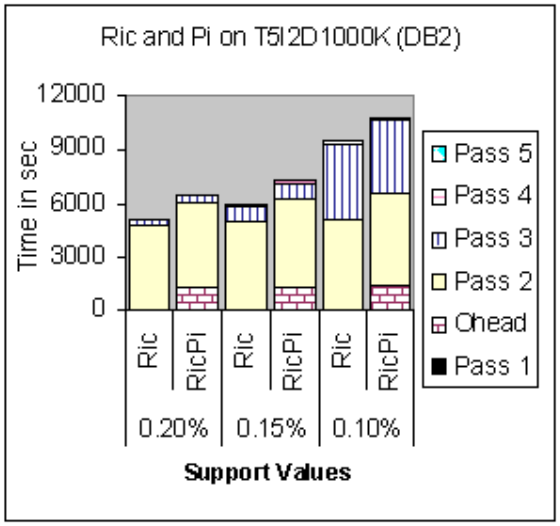

Figure 3.14 Ric on Pi

3.3.3 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 3.2.5 for Spo, first pass and candidate itemset generation in second pass is skipped. Since in Spo, C_2 is not generated, in RicSpo, instead of generating $Comb_2$, input table is joined thrice with C_3 to produce $Comb_3$ 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 for Ric in section 3.2.6. The SQL for generating $Comb_3$ directly is shown below:

```
Insert    into Comb3
Select    T1.tid, t1.item, t2.item, t3.item
From      InputTable T1, InputTable T2, InputTable T3, C3
Where     T1.item = C3.item1 and
          T2.item = C3.item2 and
          T3.item = C3.item3 and
          T1.tid = T2.tid and
          T2.tid = T3.tid
```

The overall cost for this optimization is given below. Here 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 $Comb_3$ is $S(F_3)$.

Second pass: $join(R, R, C(N_f, 2)) + group(C(N_f, 2), C(F_1, 2)) +$

Third pass: $quadjoin(R, R, R, C_3, C(N_f, 3)) + group(C(N_f, 3), C(F_2, 3)) +$

For (k >3): $trijoin(Comb_{k-1}, R, C_k, S(C_k)) + group(S(C_k), C_k))$ (3.7)

Figure 3.15 compares second pass optimization and reuse of frequent itemsets of the previous pass with their combination for table T5I2D1000K on Oracle. Figure 3.16 shows the same on DB2. As seen from these figures, in Ric, Pass-2 and Pass-3 take most of the time. This is to materialize $Comb_2$ in Pass-2, which is very huge and using this $Comb_2$ in Pass-3. Table 3.3 shows the number of tuples in $Comb_k$ for 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 $Comb_2$. For most of the datasets, this combination of optimization has come out as one of the best optimization.

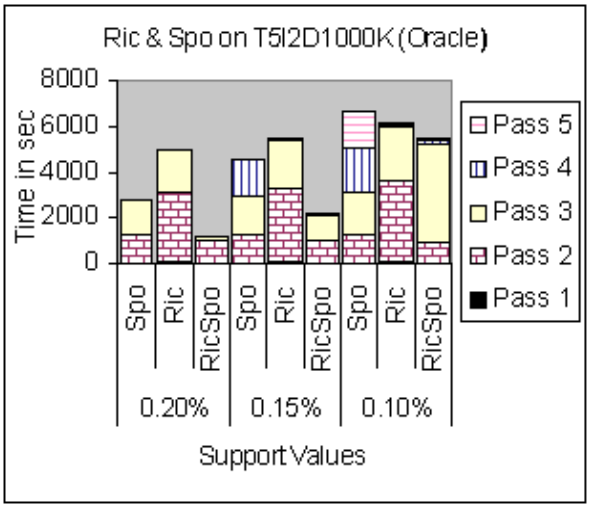

Figure 3.15 Ric and Spo (Oracle)

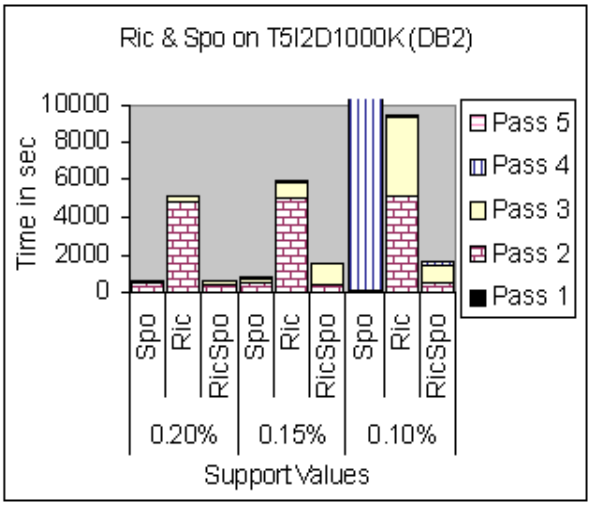

Figure 3.16 Ric and Spo (DB2)

Table 3.3 Number of Records in $Comb_k$ in 1000's

Table Characteristics	Number of Tuples in 1000's		
	$Comb_2$	$Comb_3$	$Comb_4$
T5I2D1000K. Support = 0.20%	13267	0	0
T5I2D1000K. Support = 0.20%	13756	22	0
T5I2D1000K. Support = 0.20%	14165	111	6

3.3.4 Combination of All Optimizations (All)

This is the last optimization, which is basically the combination of all the three individual optimizations discussed in sections 3.2.4, 3.2.5 and 3.2.6. The SQL for this approach is similar to the one discussed in section 3.3.3, except that instead of using the input relation as such, we first prune out all the non-frequent itemsets and then in place of input dataset use this pruned relation for support counting. So the overall cost for this optimization

would be similar to the equation 3.7 with the addition of one time cost of pruning + cost of using the pruned input in all the passes. This is given below:

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

Second pass: $join(R_f, R_f, C(N_f, 2)) + group(C(N_f, 2), C(F_1, 2)) +$

Third pass: $quadjoin(R_f, R_f, R_f, C(N_f, 3)) + group(C(N_f, 3), C(F_2, 3)) +$

For (k>3): $trijoin(Comb_{k-1}, R_f, C_k, S(C_k)) + group(S(C_k), C_k))$ (3.8)

Figure 3.17 and Figure 3.18 show the effect of this combination of optimizations on T5I2D1000K with different support values for IBM DB2/UDB and Oracle respectively. As seen from these figures, optimizing the second pass does save some time in almost all cases and also the combination of second 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 hardly ever 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, for lower support values, the total mining time for this combination of all individual optimizations is dominated by the former sub-combination, which eclipses any performance gained by the second sub-combination. Hence for most of the experiments, this combination of all the optimizations has shown better performance than others, but has been worse than the RicSpo optimization.

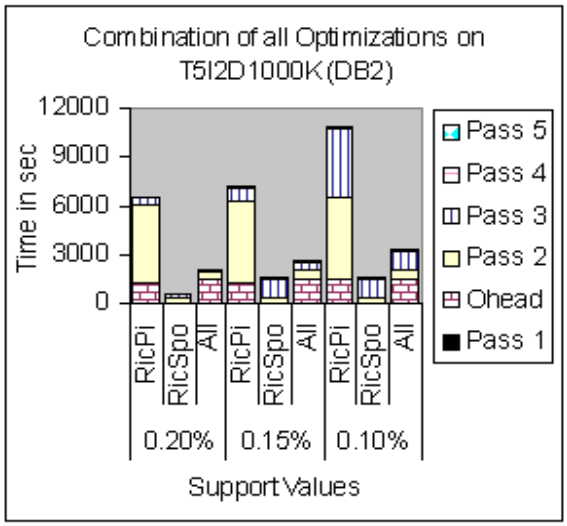

Figure 3.17 All Optimizations Combined (DB2)

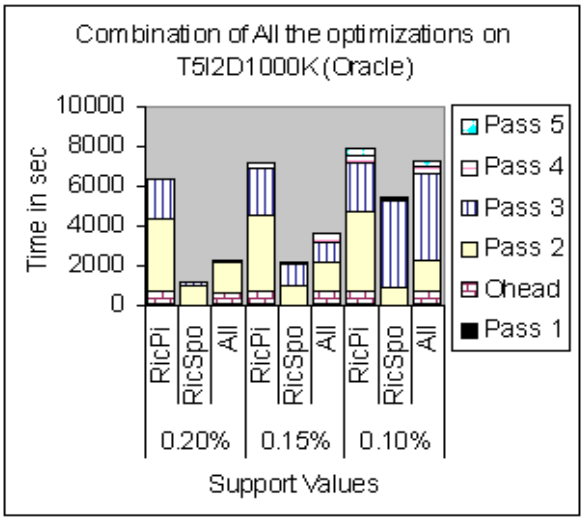

Figure 3.18 All Optimizations Combined (Oracle)

3.4 Conclusion

Based on the experiments that have been performed using SQL-92 based approaches we can summarize the effect of various optimizations as follows:

Out of the 3 approaches (k-way join, 2-Group By and Query Sub Query) for support counting, k-way join has the best performance.

In all the optimizations for k-way join method for support counting, pruning the input dataset and then using it, is very effective for higher support values while the reuse of item combinations is the best if the length of the largest frequent itemset is large. The second pass optimization has shown to reduce the mining time in almost all the cases. When combinations of these individual optimizations are considered, reuse of item combinations with second pass optimization is found to be the best among all the optimizations.

CHAPTER 4

SQL-OR BASED APPROACHES

In chapter 2 we discussed how stored procedures and udfs can be used for the purpose of candidate itemset generation and their support counting. In this chapter we shall revisit them to see their applicability in detail. Sections 4.1, 4.2 and 4.3 deal with the working of VerticalTid, Gather Join and Gather Count approaches respectively. In section 4.4 we will present further optimizations to these approaches. Finally in section 4.5, we will draw the conclusions, after comparing the experimental results of these approaches. Note that for Oracle we have implemented these as stored procedures (sp) and for IBM DB2/UDB the same has been implemented as user defined functions (udf). So, in this chapter we will use the word procedure to represent both sp for Oracle and udf for DB2, unless indicated otherwise.

4.1 VerticalTid approach (Vtid)

This approach makes uses of two procedures – SaveTid and CountAndK. The SaveTid procedure is called once to create CLOBs for representing a list of transactions. This procedure scans the input table once and for every unique item id, generates a CLOB containing the list of transactions in which that item occurs (TidList). These item ids, along with there corresponding TidList is then inserted in the TidListTable relation, which has the following schema (Item: number, TidList: CLOB). Table 4.1 shows a sample input table. In this table for the item id 2, there are 3 distinct values in the corresponding Tid column. This means that the item denoted by item id 2 occurs in those 3 transactions. The SaveTid procedure reads this item id value and generates a CLOB containing the 3 transaction ids in

which it was bought. Table 4.2 shows the corresponding values in the TidListTable produced by the SaveTid procedure.

Table 4.1 Input Table

Item	Tid
1	1
2	2
2	3
2	4
3	1
3	2
3	3
4	1
4	3
5	2
5	3
5	4

Table 4.2 TidListTable

Item	TidList
1	“1”
2	“2,3,4”
3	“1,2,3”
4	“1,3”
5	“2,3,4”

Once the TidListTable is generated, then this relation is used for support counting in all the passes. The way support counting is done is that for each itemset, the number of common transactions in the TidList of all the items constituting that itemset is found. In pass k, for support counting, the procedure CountAndK is invoked for every candidate itemset. This procedure accepts k CLOBs as input, each representing the TidList of the corresponding item constituting that itemset. The procedure then intersects these TidLists to find out those transactions, which are present in all the TidLists. The count of the common tids, which represents the support for that itemset, is then returned. Hence in pass 2 the CountAnd2 procedure will receive 2 TidLists each corresponding to the item constituting the candidate itemset of length 2 and will intersect them for finding the support count. Similarly in pass 3 the CountAnd3 procedure will receive 3 TidLists and intersect them to find the support count of candidate itemset of length 3. Table 4.3, explains the working of the CountAnd3

procedure. For example if C_3 contains following itemsets {2,3,4}, {2,3,5} and {3,4,5} where Table 4.2 shows the TidList for each itemset of length 1, then for support counting, calls made to CountAnd3 procedure and the values returned are:

Table 4.3 Counting Support Using CountAndK Procedure

Stored Procedure called	Value Returned
CountAnd3 (“2,3,4”, “1,2,3”, “1,3”)	1 (only transaction 3 is present in all the TidLists)
CountAnd3 (“2,3,4”, “1,2,3”, “2,3,4”)	2 (Transactions 2 and 3 are present in all the TidLists)
CountAnd3 (“1,2,3”, “1,3”, “2,3,4”)	1 (only transaction 3 is present in all the TidLists)

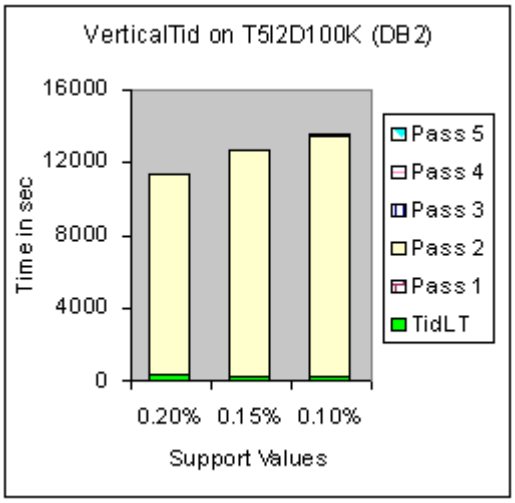

Figure 4.1 VerticalTid on T5I2D100K (DB2)

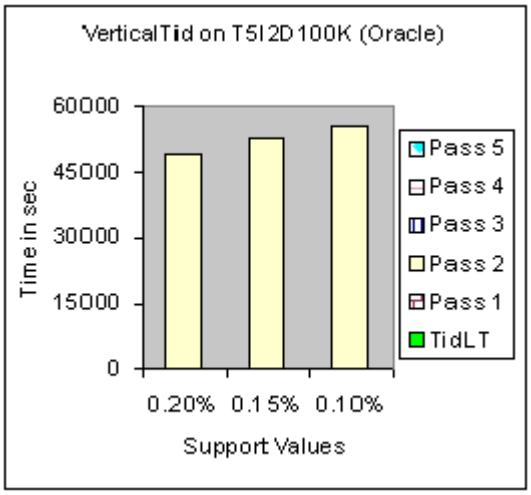

Figure 4.2 VerticalTid on T5I2D100K (Oracle)

Figure 4.1 shows the time for mining the relation T5I2D100K with different support values on DB2. Figure 4.2 shows the same for Oracle. A pass-wise analysis of these figures

shows that second pass is consuming most of the time. This is where the TidList of items constituting the 2-itemsets are compared for finding the common transactions in them. Though the counting process seems to be very straightforward but the process of reading and intersecting these CLOBs is time consuming. As number of 2-candidate itemsets is very large, the total time taken for support counting in pass 2 is very high.

The source codes for all these procedures are listed in the Appendix, at the end of thesis.

Figure 4.3 and Figure 4.4 shows how this approach scales up as size of datasets increase for support values of 0.20%, 0.15% and 0.10% on DB2 and Oracle respectively. From these figures it is clear that Vertical Tid does not do well as size of the datasets increases. Also for size of 500K and beyond, this approach did not complete even after running for over 24 hours.

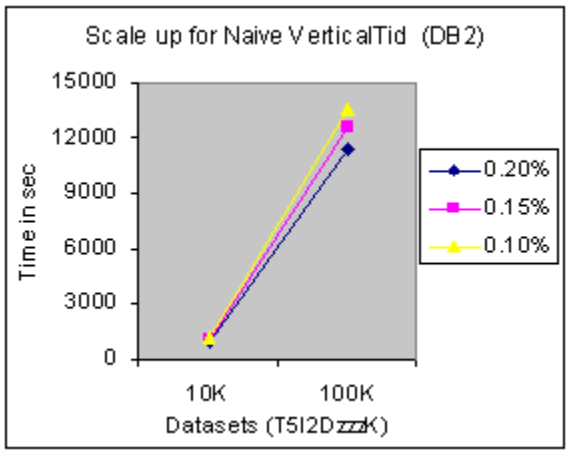

Figure 4.3 Vertical Tid scale up (DB2)

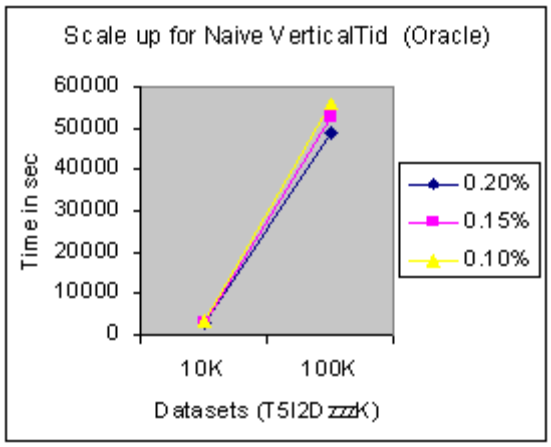

Figure 4.4 Vertical Tid scale up (Oracle)

4.2 Gather Join Approach (Gjn)

In this approach for candidate itemset generation, Thomas [2], Dudgikar [13], and our implementation for DB2 uses the SaveItem procedure. This procedure is similar to the SaveTid procedure. The only difference being that here a CLOB object represents a list of item ids. The SaveItem procedure scans the input dataset and for every unique transaction, generates a CLOB object to represent the list of items bought in that transaction (called ItemList). The transaction along with its corresponding ItemList is then inserted into the ItemListTable relation, which has the following schema: (Tid: number, ItemList: CLOB). The ItemList column is then read in every pass for generation of k-candidate itemset. In our implementation, for Oracle, we skip the generation of ItemListTable and the CombinationK stored procedure has been modified. The CombinationK udf for DB2 uses the ItemList column from the ItemListTable to generate k-candidate itemsets while in Oracle, in any pas k, this stored procedure reads the input dataset ordered by “Tid” column and inserts all item ids, corresponding to a particular transaction in to a vector. This vector is then used to generate all the possible k-candidate itemsets. This is done to avoid the usage of CLOBs as working on CLOBs in Oracle has been found to be very time consuming and also the implementation in Oracle had to be done as stored procedure, which does not necessarily needs the inputs as CLOBs.

Table 4.4 ItemListTable

Tid	ItemList
1	“1,3,4”
2	“2,3,5”
3	“2,3,4,5”
4	“2,5”

Table 4.4 shows the corresponding ItemListTable produced by the SaveItem procedure for the input table shown in Table 4.1.

Table 4.5 Input Table

Tid	Item
1	1
1	3
1	4
2	2
2	3
2	4
3	2
3	3
3	4
3	5
4	2
4	5

Table 4.6 C_2 Table

Item1	Item2
1	3
1	4
3	4
2	3
2	4
3	4
2	3
2	4
2	5
3	4
3	5
4	5
2	5

Table 4.7 C_3 Table

Item1	Item2	Item3
1	3	4
2	3	4
2	3	4
2	3	5
2	4	5
3	4	5

Table 4.5, Table 4.6, Table 4.7 show the working of this approach as implemented in Oracle. In pass 2 and pass 3, Combination2 and Combination3 stored procedures read the input dataset (Table 4.5) and generate candidate itemsets of length 2 (Table 4.6) and length 3 (Table 4.7) respectively. For DB2 the process of candidate itemset generation is as follows: In any pass k, for each tuple of ItemListTable, the CombinationK udf is invoked. This udf receives the ItemList as input and returns all k-item combinations. Figure 4.5 and Figure 4.6 show the time taken for mining the dataset T5I2D100K with different support values, using this approach on Oracle and DB2 respectively. The legend “ItemLT” corresponds to the time taken in building the ItemListTable. Since building of ItemListTable is skipped for our Oracle implementation, the time taken for building ItemListTable for Oracle is zero.

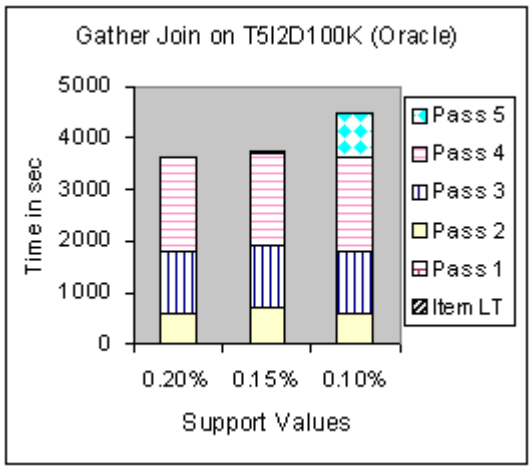

Figure 4.5 Gather Join on T5I2D100K (Oracle)

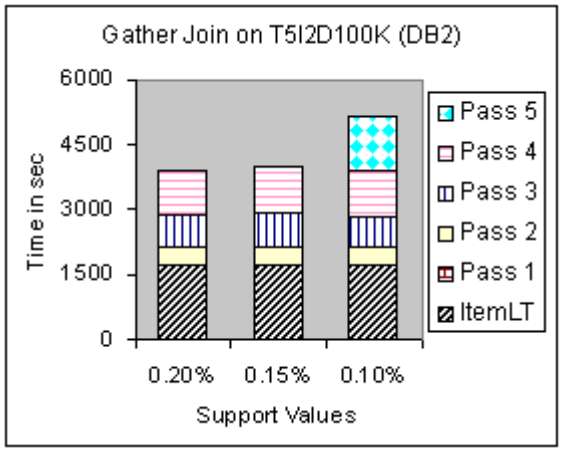

Figure 4.6 Gather Join on T5I2D100K (DB2)

4.3 Gather Count Approach (Gcnt)

This approach has been implemented for Oracle only. This is a slight modification to the Gather Join approach. Here an attempt is done to count the support of candidate itemsets directly in the memory, so as to save the time spent in materializing the candidate itemsets and then counting their support. So in pass 2, Gcnt uses GatherCount2 procedure, which is a modification to the Combination2 procedure. In second pass, instead of simply generating all the candidate itemsets of length 2 (as is done in the Combination2 procedure in Gjn), the GatherCount2 procedure uses a 2 dimensional array to count the occurrence of each itemset and then only those itemsets that have support count > the user specified minimum support value are inserted in frequent itemsets table. This reduces the time taken for generating frequent itemsets of length 2 as it skips the materialization of C_2 relation. The way it is done is that in pass 2 a 2-D array of dimensions [# of items] * [# of items] is build. All the cells of this array are initialized to zero. The GatherCount2 procedure generates all 2-item combinations (similar to the way it was done in Combination2 procedure of Gjn) and increments the count of the itemset in the array. Thus if an itemset {2,3} is generated, the

value in the cell [$Item_2$][$Item_3$] is incremented by 1. As the itemsets are generated in such a way that item in position 1 < the item in position 2, hence half of the cells in the 2-D array will always be zero.

Table 4.8 2-D Array for Support Counting

	$Item_1$	$Item_2$	$Item_3$	$Item_4$	$Item_5$
$Item_1$	0	0	1	1	0
$Item_2$	0	0	2	2	2
$Item_3$	0	0	0	3	1
$Item_4$	0	0	0	0	1
$Item_5$	0	0	0	0	0

Table 4.9 Relation F_2

$Item_1$	$Item_2$	Count
2	3	2
2	4	2
2	5	2
3	4	3

For the input dataset shown in Table 4.5, Table 4.8 shows the 2-Dimensional array generated by the GatherCount2 procedure. Once the entries in this array are filled, a pass is made over it to find out the number of times an item combination has been generated. If an item combination has a count > user specified support value, then it is inserted in the frequent itemsets' relation. For the dataset shown Table 4.5 the F_2 relation will contain only the itemsets shown in Table 4.9.

However this method of support counting cannot be used for higher passes, because building a 3 or more dimensional array would cost a whole lot of memory. Also creating an in-memory tree like data structure for representing the itemsets of length k and then trying to resolve keys at k-levels (during support counting) will not only consume a whole lot of memory but also shall be very time consuming. Hence for all passes greater than 2, Gcnt uses the same CombinationK procedure that are used by the Gjn approach.

4.4 Analysis and Optimizations to the SQL-OR based approaches

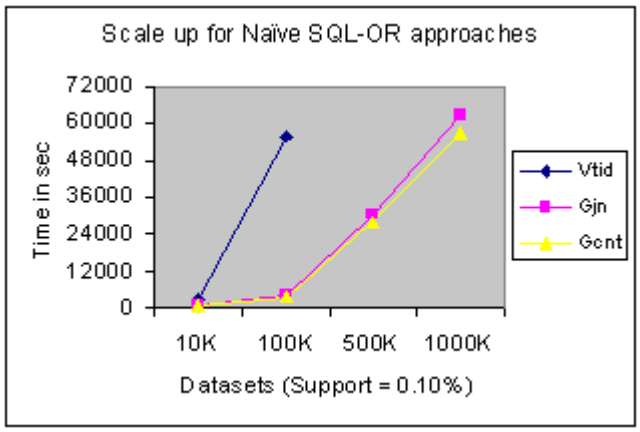

Figure 4.7 Naïve SQL-OR Based Approaches (Oracle)

Figure 4.7 compares the time taken for mining by the naïve SQL-OR based approaches for support values of 0.10% on datasets T5I2D10K, T5I2D100K, T5I2D500K and T5I2D1000K on Oracle. From this figure it is very clear that of the 3 approaches, Vertical Tid has the worst performance (VerticalTid didn't complete for datasets T5I2D500K and T5I2D1000K even after running for 24 hours). This is because Vtid blows up at the second pass, where the overall time taken in support counting of all the 2-itemsets by intersecting their TidLists is very large. So the optimization to Vtid would be to reduce the

number of TidLists processed by the CountAndK procedure in each pass. This optimization is explained in more detail in section 4.4.1.

For the other two approaches, though they complete for large datasets but they take a whole lot of time. The difference in the candidate itemset generation process, as is done in these approaches and the way it is done for any SQL-92 based approach is that here in any pass k, all the items bought in a transaction (the complete ItemList) are used for generation of candidate itemsets. Whereas in the SQL-92 based approaches, in k^{th} pass, only frequent itemsets of length k-1, were extended. The significance of this is in the number of candidate itemsets that are generated at each pass and the way support counting is done. In SQL-92 based approaches, frequent itemsets of length k-1 are used to generate candidate itemsets of length k and then additional joins are done to consider only those candidate itemsets, whose subsets of length k-1 are also frequent (because of the subset property). This reduces the number of candidate itemsets that are generated at each pass significantly. But then for support counting input dataset had to be joined k-times with an additional join condition to identify that these items (constituting an itemset) where coming from same transaction. In Gjn and Gcnt, since the candidate itemsets are generated from the complete ItemList of a transaction, there is no need to join the input dataset. Just a single group by on the items constituting an itemset, with a having clause is sufficient to identify all those candidate itemsets that are frequent. However, in any pass k, there is no easy way to identify the frequent itemsets of length k-1 and use them selectively to generate candidate itemsets of length k; rather the entire ItemList is used for generation of k-candidate itemsets. This generates a huge number of unwanted candidate itemsets and hence an equivalent increase in the time for support counting.

Table 4.10 Number of Candidate Itemsets

Relation	Number of Candidate Itemsets
C_2	1450790
C_3	2521176
C_4	3246582
C_5	3312486
C_6	0

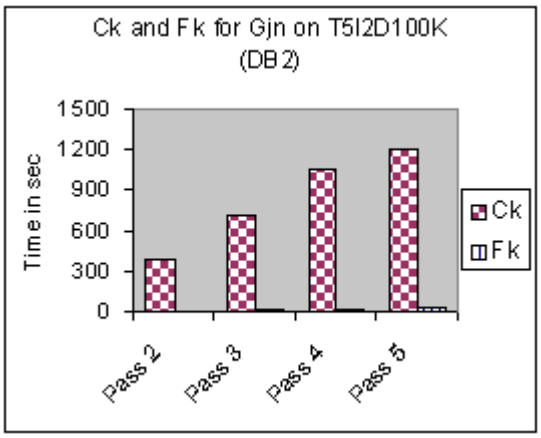

Figure 4.8 Ck and Fk for Gjn (DB2)

Table 4.10 shows the number of candidate itemsets generated and Figure 4.8 compares the time taken in generation of these candidate itemsets and their support counting for each pass for dataset T5I2D100K, for support value of 0.10% on DB2. These figures suggest that most of the time taken is in the generation of large number of candidate itemsets. So a way to optimize it would be to reduce the number of candidate itemsets generated. This optimization is explained in detail in the section 4.4.2 and 4.4.3.

4.4.1 Improved VerticalTid Approach (IM_Vtid)

In Vtid approach for support counting, in any pass k, the TidList of each item constituting an itemset is passed to the CountAndK procedure. As the length of the itemsets increases, the number of TidLists passed as parameter to the CountAndK procedure also increases (in pass k, CountAndK procedure receives k TidLists). So to enhance the process of support counting, this optimization does the following: In pass 2, frequent itemsets of length two are generated directly by performing a self-join of input dataset. The join condition being that the item from the first copy < the item from second copy and that both the items belong to the same Tid. This is same as is done in the Second pass optimization for SQL-92 based

approaches [2]. For pass 3 onwards, for those itemsets, whose count > minimum support value, the CountAndK procedure builds again a list of transactions (as a CLOB) that have been found common in all the TidLists to represent that itemset as a whole. (We have implemented this for Oracle only and have modified the CountAndK stored procedure to reflect the above change, hence for this optimization CountAndK procedure is used only in the reference of implementation for Oracle.) In pass k, the itemset along with its TidList is materialized in an intermediate relation. In the next pass (pass k+1), during the support counting of the candidate itemsets (which are one extension to the frequent itemsets of length k, that have been materialized in pass k), there is no need to pass the TidLists of all the items constituting this itemset. Instead, just two TidLists – one representing the k-itemset and other representing the item, extending this itemset are passed. This saves a whole lot of time, in searching the list of common transactions in the TidLists received by the CountAndK procedure.

Table 4.11 Counting Support Using CountAnd3 Procedure

Stored Procedure called	Intermediate Relation				
	$Item_1$	$Item_2$	$Item_3$	TidList	Count
CountAndK (“2,3,4”, “1,2,3”, “1,3”)	2	3	4	“3”	1
CountAndK (“2,3,4”, “1,2,3”, “2,3,4”)	2	3	5	“2,3”	2
CountAndK (“1,2,3”, “1,3”, “2,3,4”)	3	4	5	“3”	1

For example if C_3 contains following itemsets {2,3,4}, {2,3,5} and {3,4,5} where Table 4.2 shows the TidList for each itemset of length 1, then for support counting, calls made to CountAnd3 procedure and the values inserted in the intermediate relation is shown in Table 4.11. For itemset {2,3,4} the list of transactions common in the TidList of items {2},

{3} and {4} along with the count is inserted in the intermediate relation. In pass 4 if the C_4 is {2,3,4,5} then the procedure CountAnd4 is called as: CountAnd4 ("3", "2,3,4"), where the first TidList represents the itemset {2,3,4} and the second TidList represents the itemset {5}.

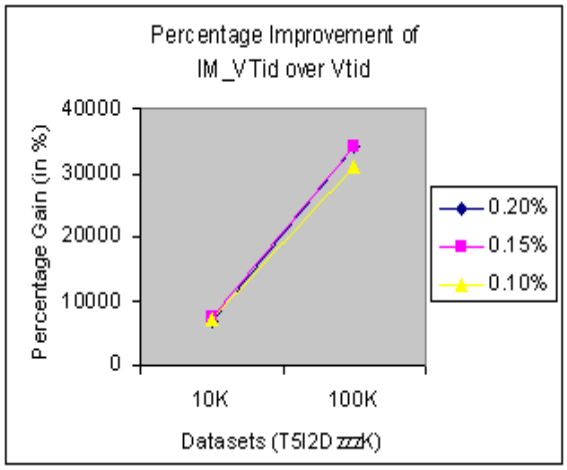

Figure 4.9 Percentage Gain of Im_Vtid over Vtid

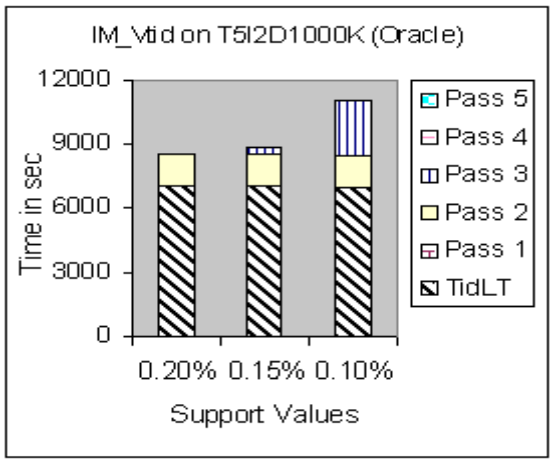

Figure 4.10 IM_Vtid on T5I2D1000K

Figure 4.9 shows the performance gained (in percentages) by using Im_Vtid over Vtid for datasets T5I2D10K and T5I2D100K for support values of 0.20%, 0.15% and 0.10% (for other datasets Vtid didn't complete). Figure 4.10 shows the overall time taken for mining the relation T5I2D1000K with IM_Vtid approach for different support values on Oracle. The legend TidLT represents the time taken in building the TidListTable from the input dataset (T5I2D1000K). This phase basically represents the time taken in building the TidList (a CLOB object) for each item id. From Figure 4.10 it is clear that time taken in building the TidListTable is a huge overhead. It accounts for nearly 60 to 80 percent of the total time spent for mining. Though this optimization is very effective but still the time taken for building the TidListTable shows that the efficiency of RDBMS in manipulating CLOBs is a bottleneck.

4.4.2 Improved Gather Join Approach (IM_Gjn)

In Gjn approach, in any pass k, all the items that occur in a transaction are used for the generation of candidate itemsets of length k. In subsequent passes, the items, which did not participate in the generation of frequent itemsets, are not eliminated from the list of items for that transaction. (We know through the subset property that, if an itemset, "A" is not frequent, then any extension of itemset "A" will not be frequent). There is no easy way of scanning and eliminating all those items from the ItemList of a transaction that did not participate in the formation of frequent itemsets in any pass. As there is no pruning of the items, a huge number of unwanted candidate itemsets are generated in every pass, which is very time consuming. One possible way to optimize this would be that in any pass k, use the tuples of only those transactions, (instead of the entire input table) which have contributed to the generation of frequent itemsets in pass k-1. For this we use an intermediate relation FComb. FComb has the following schema: (Tid: number, $item_1$: number, $item_2$: number, … $item_{k-1}$: number). In any pass k, this relation contains the tuples of only those transactions whose items have contributed in the formation of frequent itemsets in pass k-1. This is done by joining the candidate itemsets table (C_{k-1}) with the frequent itemsets table (F_{k-1}). But for identifying the candidate itemsets that belong to same transaction, the CombinationK stored procedure has been modified (for Oracle) to insert the transaction id along with the item combinations that were generated from the ItemList of that transaction, in the C_k relation. In any pass k, FComb table is thus generated which is then used by the CombinationK stored procedure (instead of the input dataset) to generate candidate itemsets of length k. This optimization has been implemented for Oracle only, as this does not need the ItemList column to be materialized as a CLOB object; rather it can generate it in every pass using the FComb relation. The same wont be very effective with DB2, as the CombinationK udfs need

the input to be in the form of CLOB and creation of CLOB for every ItemList at each pass would be very time consuming.

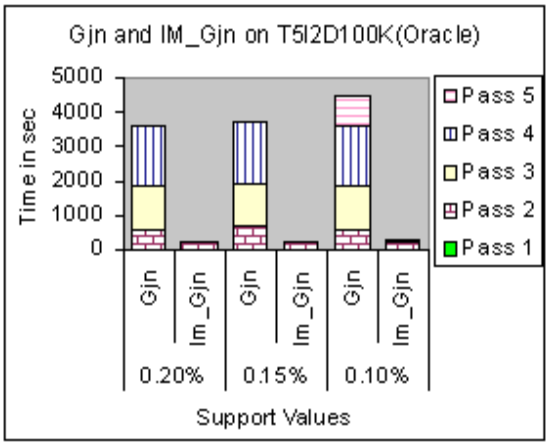

Figure 4.11 Gjn & IM_Gjn on T5I2D100K (Oracle)

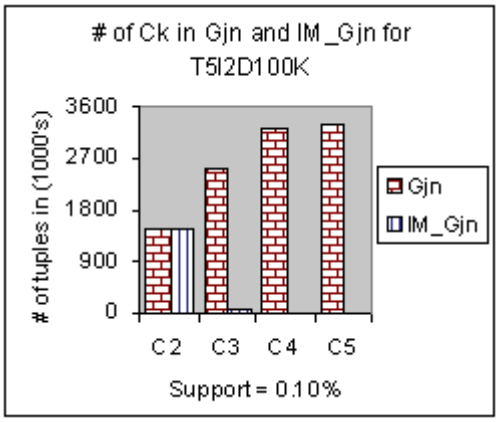

Figure 4.12 Size of Ck (Gjn & IM_Gjn)

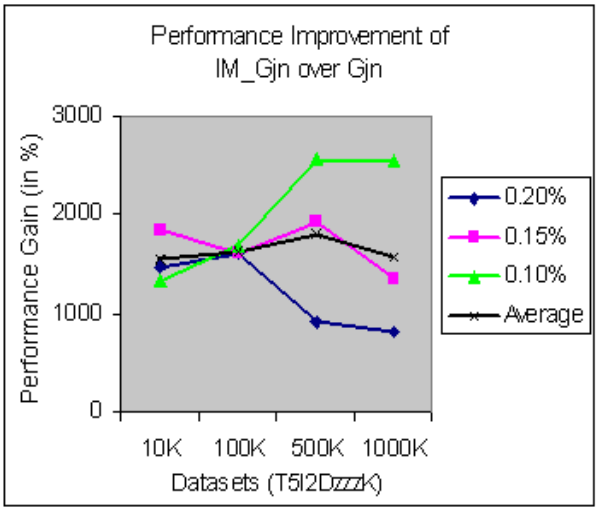

Figure 4.13 Performance Gain for IM_Gjn

Figure 4.11 compares the time required for mining the dataset T5I2D100K on Oracle for different support values, when the FComb table is materialized (IM_Gjn) and used for the generation of candidate itemsets and when input table is used as it is (Gjn) for generation of candidate itemset. We see that the total mining time by using FComb relation is quite less than the total mining time using the input dataset as it is. Also in Gjn, for different support values (0.20%, 0.15% and 0.10%) the time taken in each pass is nearly same. This is because in Gjn there is no pruning of candidate itemsets and then irrespective of the user specified support values the entire ItemList is used for generating all the candidate itemsets of length k. Figure 4.12 compares the number of candidate itemsets generated for relation T5I2D100K when input relation and when FComb relation are used by the CombinationK stored procedure for support value of 0.10%. From this figure, we see that in higher passes, when input relation is used then the number of candidate itemsets are significantly larger than when FComb relation is used which accounts for the difference in the total time taken for mining by these two methods. Figure 4.13 shows the performance gained (in percentages) by using IM_Gjn over Gjn on datasets T5I2D10K, T5I2D100K, T5I2D500K and T5I2D1000K for different support value. From this figure we see that on the different datasets, on an average the gain is 1500%. The SQL for generation of FComb table is shown below:

```
Insert    into FComb
Select    I1.Tid, I2.item1, I2.item2,…, I2.itemk-1
From      Ck-1 I1, Fk-1 I2
Where     I1.item1 = I2.item1 And
          I1.item2 = I2.item2 And
                   :
          I1.itemk-1 = I2.itemk-1
```

4.4.3 Improved Gather Count Approach (IM_Gcnt)

As Gcnt approach is a slight modification of the Gjn approach, the optimization suggested for the Gjn can be used for Gcnt approach also. In Gcnt approach, the second pass uses a 2-Dimensioanl array to count the occurrence of all item combinations of length 2 and those item combinations whose count > user specified support value are directly inserted in the frequent itemsets' relation (F_2). As materialization of the candidate itemsets of length 2 (C_2) at this step is skipped hence in the third pass, F_2 is joined with two copies of input dataset to generate FComb, which is then used by the modified Combination3 stored procedure. For subsequent passes, materialization of FComb relation is done in the same manner as is done for the IM_Gjn approach.

Figure 4.14 compares the mining time for tables T5I2D1000K on Oracle using IM_Gcnt approach for different support values and also compares it with the IM_Gjn approach. This figure shows that, of both the approaches, IM_Gcnt performs better than IM_Gjn. This is because of the time saved in the second pass of the IM_Gcnt approach. For the rest of the passes, the time taken by both of them is almost same as both of them use the same modified CombinationK stored procedure for generation of candidate itemsets. Thus if memory is available for building the 2-D array then performance can be improved by counting the support in memory. Remember that the size of the array needed would be of the order of n^2 where n is the number of distinct items in the dataset. Figure 4.15 shows the performance gained (in percentages) by using IM_Gcnt over Gcnt on datasets T5I2D10K, T5I2D100K, T5I2D500K and T5I2D1000K for different support values. From this figure we see that on an average the gain for different support values is 2500% on the different datasets.

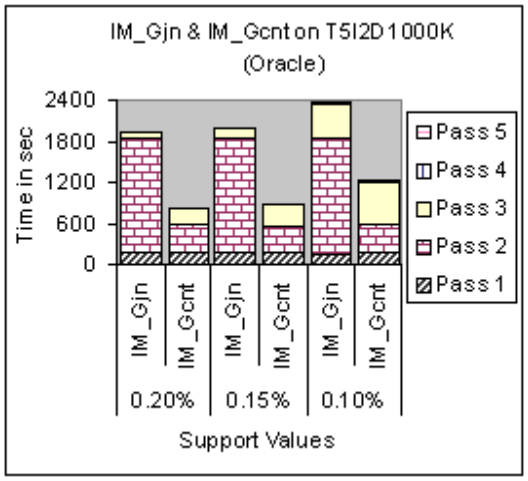

Figure 4.14 Gjn and Gcnt for T5I2D1000K (Oracle)

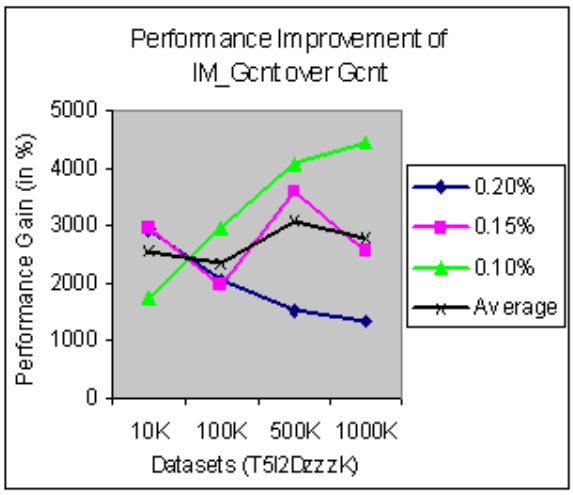

Figure 4.15 Performance Gain for IM_Gcnt

4.5 Conclusion

The SQL-OR based approaches seems to be quite simple in the way candidate itemsets are generated and there support is counted. But when compared with SQL-92 based approaches, they do not even come close. The time taken by the naïve SQL-OR based approaches, using stored procedures and udfs, is much more than the basic k-way join approach for support counting. This is because the data representation in SQL-92 based approaches is very simple which reduces their processing time. In the SQL-OR based approaches, the use of complex data structure though makes the process of mining so simple, also makes it quite inefficient, as time taken to process these data structures is very large. Among the naïve SQL-OR based approaches, we found that the Gather Count approach is the best while the VerticalTid approach has the worst performance. Figure 4.16 shows this for dataset T5I2D100K on Oracle and Figure 4.7 compares the total time taken by these approaches for different datasets. The Gather count outperforms the Gather join approach because in the second pass it uses main memory to do the support counting and hence skips

the generation of candidate itemsets of length 2. The performance of VerticalTid approach is worst. Here for each candidate itemset produced, CountAndK procedure has to be called, which intersects all the TidLists received as parameter to do the support counting. As processing CLOBs is costly, hence the overall time required by the VerticalTid approach is much more than other approaches.

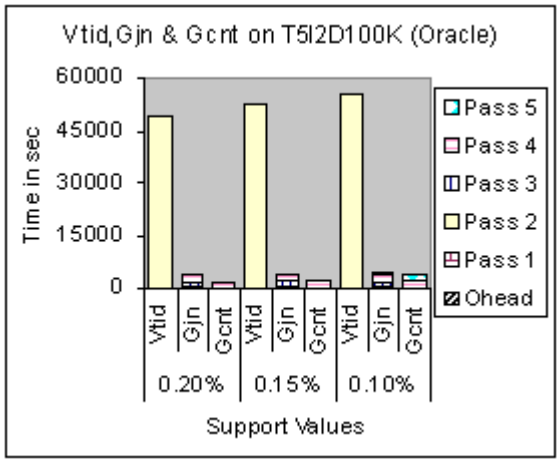

Figure 4.16 Vtid, Gjn and Gcnt on T5I2D100K (Oracle)

The optimizations to these approaches, thus tries to mix the advantages of both: SQL-92 and SQL-OR based approaches. The IM_Vtid approach uses second-pass optimization for pass two (simple SQL query), and then onwards uses CLOBs for support counting. Also the optimization reduces the number of TidLists (materialized as CLOB objects) that the procedure CountAndK has to process for each candidate itemset to two, by building the TidList for itemsets found frequent in pass k and using it for the purpose of support counting in the next pass.

Similarly the other optimizations (IM_Gjn and IM_Gcnt), as implemented in Oracle, avoid the usage of CLOB objects and hence these improved versions seem to be very

promising. The Gather Count approach, which makes use of system memory in second pass for support counting, is an improvement over the optimization for the Gather Join approach. Figure 4.10 and Figure 4.14 shows the performance of IM_Vtid, IM_Gjn and IM_Gcnt for dataset T5I2D1000K for different support values. From these figures it is clear that IM_Gcnt is the best of the three SQL-OR approaches and their optimizations discussed in this chapter.

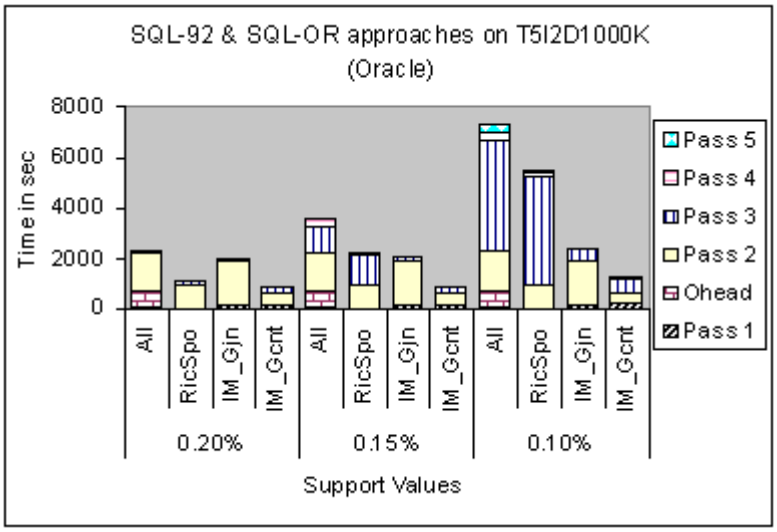

Figure 4.17 Best Optimizations on T5I2D1000K (Oracle)

Figure 4.17 compares the performance of the two best optimizations for SQL-OR based approaches (IM_Gjn & IM_Gcnt) with the two best optimizations for the SQL-92 based approach (combination of all the optimization and Reuse of item combinations with Second pass optimization) on dataset T5I2D1000K for Oracle. This figure shows that, the optimized SQL-OR based approaches are very efficient and the IM_Gcnt approach is the best of all the optimizations seen so far.

CHAPTER 5

OTHER CONTRIBUTIONS

This chapter consists of other contributions that are needed for the generation of association rules and for performing large number of experiments. Section 5.1 deals with the process of generation of subsets of the frequent itemsets, needed during the rule generation phase. Section 5.2 explains the Configuration file that can be used in running this tool in a batch mode. Section 5.3 discusses the Log files that are generated during the mining process and how these logs can be formatted to give us a better understanding of the results.

5.1 Subsets Generation

The generation of rules subsumes the generation of subsets of all the itemsets identified as frequent itemsets after the end of the support counting phase. The frequent itemsets generated in pass k, are stored in relation F_k. For rule generation, all the frequent itemsets are collected in one single relation FrequentTable. The FrequentTable has the following attributes ($item_1$, $item_2$,…,$item_K$, Count and NullMarker). Here "$_K$"is the user specified value (Stop Level), which represents the upper bound on the length of the largest frequent itemset that can be found during mining a given dataset. It can certainly be greater than the actual length of the largest frequent itemset found during mining that dataset. Since the frequent itemsets are of varying length, the NullMarker is used. It identifies the length of the frequent itemset. The count column in FrequentTable maintains the count of each frequent itemset found during support counting. This is later on used for calculating the confidence value of the rules associated with that frequent itemset. An example below shows the representation of the frequent itemsets in the relation FrequentTable.

Table 5.1 FrequentTable

Item1	Item2	Item3	Item4	Item5	Item6	Item7	Item8	Count	NullMarker
2	4	5	0	0	0	0	0	9	4

Table 5.1 shows that FrequentTable has 10 columns – 8 ($Item_1$, $Item_2$,…, $Item_8$) for storing the items constituting a frequent itemset and the 9^{th} for storing its count value and 10^{th} for its length. Here it is assumed that the maximum length of a frequent itemset will not be more than 8. Hence the FrequentTable has 8 columns for storing the item ids. The tuple shown in the FrequentTable represents the frequent itemset {2,4,5} and the NullMarker says that column 4 onwards, the values should be ignored for this frequent itemset.

For rule generation, each tuple (representing a frequent itemset) from the FrequentTable is retrieved and all the subsets of that frequent itemset are generated to form the rules. The generation of subset and rules is as follows:

Table 5.2 Rule Generation

Position	Pos1	Pos2	Pos3	Rule		Count
ItemSet	2	4	5	Antecedent	Consequence	9
Sequence	0	0	1	2,4	5	9
	0	1	0	2,5	4	9
	0	1	1	2	4,5	9
	1	0	0	4,5	2	9
	1	0	1	4	2,5	9
	1	1	0	5	2,4	9

For a frequent itemset of length n, we generate 2^n –2 sequences of length n. These sequences are the binary representation of all the numbers from 1 to 2^n – 1. Now if any sequence has a *"0"*, in position *m* that means the item id in position *m* (in the frequent

itemset) is in the antecedent of the rule. Similarly, if there is *"1"* at position *m*, then the item id at position *m*, is in the consequence of the rule. All the rules generated from a frequent itemsets inherit its support count (number of times that frequent itemset occurs in the entire dataset) Table 5.2 shows all the rules generated this way for frequent itemset {2,4,5}. So if the itemset {2,4} is in the antecedent and {5} in the consequence, the association rule is depicted as $\{2,4\} \Rightarrow \{5\}$.

The rules are then inserted in the RulesTable relation and only those rules are finally displayed whose confidence satisfies the user specified value. The process of calculation of confidence is same as described by Dudgikar [13] but this process of rule generation has been found to be more efficient than using joins and complex SQL queries, as was done by Dudgikar [13] for generating subsets.

5.2 Configuration File

There are two ways for using this mining tool. The first is using the GUI and other is using the configuration file. Running this mining tool using GUI has been described in [13]. The GUI is useful for a non-expert (or a novice), but needs some human intervention to provide the configuration needed for mining. The configuration file is useful for automating the mining process. It consists of a number of parameters, which once specified correctly, can be used for mining in an unattended mode. It can also be used for mining several datasets with varying mining configurations without any user intervention. The variables defined in the configuration file are:

RDBMS Name: The RDBMS name (Oracle or DB2) where the input relation is present.

Database Name: The database that contains your input relation.

UserId: The user who has access over the input relation.

Password: The password associated with the UserId – needed to connect to the database.

Log File: The name of the Result Log file to generate.

Approach Number: The approach number to be used for mining. It is an integer value. All the approaches and their optimizations are given a unique integer value to identify them.

Table Name: The name of the input relation.

Support: Minimum support value to be used for mining. This is in percentage.

Confidence: The confidence value to be used for rule generation. It is an integer value (as percentage)

Stop Level: The maximum number of passes to go before stopping.

Debug: If true, then prints the debug statements.

Skip Rules: If true, the program stops after the generation of frequent itemset. Rule generation is skipped.

Reverse Mapping: If true, the results (item ids) are mapped back to their original names.

Log Results to file: If true, trace values will be written to the Log File.

For each experiment, the values of all these variables are written in a single line in the order of the variables shown above and are demarcated by a "$" sign. Thus if the configuration file contains several such lines, the mining algorithms will be invoked that many time. To skip a line, the line should start with the word "REM". Below is an example of some mining configurations.

REM Experiment on DB2. Approach -Kwj

DB2$Sample$ntmining$ntmining$D_A10_T5I2D500K.txt10T5I2D500K$3.00%$50$8$false$true$false$true

Here the first line is ignored as it starts from the word "REM". For second line values are used as follows:

RDBMS to use: DB2

Database Name: Sample

UserID: ntminig

Password: ntmining

Log File: D_A10_T5I2D500K.txt

Approach Name: 10 (for K-way Join)

Input Table: T5I2D500K.

Support: 3.00%

Confidence: 50 (percent)

Stop Level: 8

Debug: False (don’t print debug statements)

Skip Rules: True (skip rule generation)

Reverse Mapping: False (don’t do reverse mapping)

Log result to File: True (write the log file).

5.3 Writing Log File

Data mining is a time-consuming process and at times it happens that for certain mining configurations, mining a given dataset may take 10 to 15 hrs or even more. Since we have to compare the performances of these approaches with others, after a given time limit, if the approach does not complete, the mining process has to be killed. Also for the purpose of studying these algorithms, we need to know about their progress during mining a data set. Hence it is very important to note down the time at each step of the algorithm and produce a log file containing enough information. This log file can then be processed to generate the useful information such as the number of passes completed, time taken for each pass, intermediate relations generated and cardinality of each of them, even if the mining process is killed before it completes. For this purpose, we generate two log files. One is the time log, which is written at the end of materialization of any relation generated during the mining

process. This log (TimeLog) contains the time stamp of when a particular pass of the approach started and if any intermediate relations where generated, what is there cardinality. Below is a sample content of these logging files.

Contents of the TimeLog file:

Start - Approach 50. Table = FinalInput Support =0 Wed Sep 11 17:47:11 CDT 2002

// This indicates the start of VerticalTid approach on input relation FinalInput.

// The support value, in terms of row count is 0.

TIDT Wed Sep 11 17:47:11 CDT 2002

F1 Wed Sep 11 17:47:11 CDT 2002

C2=10 Wed Sep 11 17:47:11 CDT 2002

F2 Wed Sep 11 17:47:12 CDT 2002

C3=10 Wed Sep 11 17:47:12 CDT 2002

F3 Wed Sep 11 17:47:12 CDT 2002

C4=5 Wed Sep 11 17:47:12 CDT 2002

F4 Wed Sep 11 17:47:13 CDT 2002

C5=1 Wed Sep 11 17:47:13 CDT 2002

F5 Wed Sep 11 17:47:13 CDT 2002

C6=0 Wed Sep 11 17:47:14 CDT 2002

F6 Wed Sep 11 17:47:14 CDT 2002

C7 Wed Sep 11 17:47:14 CDT 2002

F7 Wed Sep 11 17:47:14 CDT 2002

C8 Wed Sep 11 17:47:14 CDT 2002

F8 Wed Sep 11 17:47:14 CDT 2002

Subsets Wed Sep 11 17:47:14 CDT 2002

Rules Wed Sep 11 17:47:14 CDT 2002

The second column, in each row is the timestamp when all the tuples where inserted in that that particular relation. The first column contains the relation name and their cardinality. For those relation names, which do not have "=" character in them, they are either the relations for Frequent itemsets (F_k) or were not generated but are there as the variable *Stop Level,* in the configuration file, specifies that the experiment should run until that pass number. (We do so just to maintain a consistency in the output that is generated. The cardinalities of frequent itemsets relations are calculated at the end during writing the ResultLog.) The other log (ResultLog) is written only when a given approach completes successfully. This log contains the same information as the TimeLog but it is formatted in a different way. A Java program (LogProcess.java) is used over the ResultLog file for processing it and extracting only the needed information. It also formats the output so that it can be directly loaded in an Excel spreadsheet for the purpose of analysis. A sample content of the formatted output produced by the LogProcess.java file is shown below:

Output of the LogProcess.java file:

// The second column shows the cardinality of the corresponding relation.

F1	786
C2	308505
F2	930
C3	281
F3	130
C4	14
F4	14
C5	0
F5	0
C6	0

F6 0

C7 0

F7 0

C8 0

F8 0

Subsets 2836

Rules 589

// The second column after this shows the time (in milliseconds) in generating the // corresponding relation.

F1 511

C2 160301

F2 17765

C3 781

F3 2294

C4 621

F4 2012

C5 1052

F5 1232

C6 10

F6 0

C7 0

F7 10

C8 0

F8 0

Total Count 186599

Rule Gen 41049

For each row from top, till the first column containing the value “Rules”, the second column shows the cardinality of the corresponding relation. The second column then onwards shows the time taken to generate that relation. The last two rows show the total time taken in generation of all the frequent itemsets and the time taken in the generation of rules, respectively. Time here is measured in milliseconds.

CHAPTER 6

CONCLUSION AND FUTURE WORK

We have compiled the results obtained from mining different relations into a tabular format. This can be converted into metadata and made available to the mining-optimizer so that it 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 some of the characteristics of the input table. As for SQL-92 based approaches, since Kwj was found to be the best, results reported here are only for Kwj and its optimizations. For SQL-OR based approaches, as the optimizations were implemented on Oracle only, the summary table (Table 6.4) for optimizations to SQL-OR based approaches report results only for Oracle.

Table 6.1 and Table 6.2, summarizes the ranking of the basic Kwj and its 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, the summary table contains 3 rows. The first two rows specify the two best optimizations and the last row lists the worst optimization for that dataset. The format is the same for both – Oracle and IBM DB2/UDB. For the purpose of building the meta-data Table 6.3 and Table 6.4 provides a summary of results obtained from mining various other datasets on both IBM DB2/UDB and Oracle. 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, specified support, and if there is enough additional space to materialize the intermediate results.

Table 6.1 Trends in Oracle

Table Name	Ranking	Supp = 0.20%	Supp = 0.15%	Supp = 0.10%	
T5I2D1000K	First	**RicSpo**	**RicSpo**	*Kwj*	
	Second	*All*	*All*	*RicSpo*	
	Last	RicPi	RicPi	RicPi	
T5I2D500K	First	**RicSpo**	**RicSpo**	Spo	
	Second	Spo	Spo	**RicSpo**	
	Last	RicPi	RicPi	RicPi	
	Ranking	Supp = 2.00%	Supp = 1.00%	Supp = 0.75%	Supp=0.33%
T10I4D100K	First	*All*	**RicSpo**	**RicSpo**	Ric
	Second	Pi	*All*	*All*	Spo
	Last	Ric	RicPi	RicPi	**RicSpo**

Table 6.2 Trends in IBM DB2/UDB

Table Name	Ranking	Supp = 0.20%	Supp = 0.15%	Supp = 0.10%
T5I2D1000K	First	**RicSpo**	Spo	**RicSpo**
	Second	Spo	**RicSpo**	*All*
	Last	RicPi	SpoPi	SpoPi
T5I2D500K	First	Spo	Spo	Spo
	Second	**RicSpo**	**RicSpo**	**RicSpo**
	Last	SpoPi	SpoPi	SpoPi
	Ranking	Supp = 2.00%	Supp = 1.00%	Supp = 0.75%
T10I4D100K	First	Spo	**RicSpo**	**RicSpo**
	Second	**RicSpo**	*All*	*All*
	Last	Ric	*Kwj*	*Kwj*

Table 6.3 Meta–data Table for SQL-92 Based Approaches

Table Size T5I2DzzzK	DB2		Oracle		Support Value
	Extra Space	No Extra Space	Extra Space	No Extra Space	
10K	RicSpo	Spo	RicSpo	Spo	S = 0.20%
	RicSpo	Spo	RicSpo	Spo	S = 0.15%
	RicSpo	Spo	Spo	Spo	S = 0.10%
50K	RicSpo	Spo	RicSpo	Spo	S = 0.20%
	Spo	Spo	RicSpo	Spo	S = 0.15%
	Spo	Spo	Spo	Spo	S = 0.10%
100K	RicSpo	Spo	RicSpo	Spo	S = 0.20%
	Spo	Spo	RicSpo	Spo	S = 0.15%
	Spo	Spo	Spo	Spo	S = 0.10%
500K	Spo	Spo	RicSpo	Spo	S = 0.20%
	Spo	Spo	RicSpo	Spo	S = 0.15%
	Spo	Spo	Spo	Spo	S = 0.10%
1000K	RicSpo	Spo	RicSpo	Spo	S = 0.20%
	Spo	Spo	RicSpo	Spo	S = 0.15%
	Spo	Spo	Kwj	Kwj	S = 0.10%

Table 6.4 Meta–data Table for SQL-OR Based Approaches

Table Size T5I2DzzzK	DB2		Oracle		Support Value
	Extra Space	No Extra Space	Extra Space	No Extra Space	
10K	-NA-	Gjn	IM_Gcnt	Gcnt	S = 0.20%
	-NA-	Gjn	IM_Gcnt	Gcnt	S = 0.15%
	-NA-	Gjn	IM_Gcnt	Gcnt	S = 0.10%
50K	-NA-	Gjn	IM_Gcnt	Gcnt	S = 0.20%
	-NA-	Gjn	IM_Gcnt	Gcnt	S = 0.15%
	-NA-	Gjn	IM_Gcnt	Gcnt	S = 0.10%
100K	-NA-	Gjn	IM_Gcnt	Gcnt	S = 0.20%
	-NA-	Gjn	IM_Gcnt	Gcnt	S = 0.15%
	-NA-	Gjn	IM_Gcnt	Gcnt	S = 0.10%
500K	-NA-	Gjn	IM_Gcnt	Gcnt	S = 0.20%
	-NA-	Gjn	IM_Gcnt	Gcnt	S = 0.15%
	-NA-	Gjn	IM_Gcnt	Gcnt	S = 0.10%
1000K	-NA-	Gjn	IM_Gcnt	Gcnt	S = 0.20%
	-NA-	Gjn	IM_Gcnt	Gcnt	S = 0.15%
	-NA-	Gjn	IM_Gcnt	Gcnt	S = 0.10%

6.1 Conclusion and Future Work

In this thesis, we have explored the various formulations (both SQL-92 and SQL-OR based), their combinations and possible optimizations for association rule mining. We have analytically and experimentally compared these approaches and their optimizations in an attempt to provide better insight to their effect 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 available storage, 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 for SQL-92 based approaches is the reuse of item combinations or reuse of item combinations combined with the second pass optimization when we have enough space (on disk) 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.

In SQL-OR based approaches, if we have enough memory to build a 2 dimensional array for counting support in the second pass, then Gather count approach has been found to be the best of all the naïve SQL-OR based approaches. If building an in memory 2-dimensional array is a problem, then Gather join is a better alternative. The same implies when we have enough space to materialize intermediate relations (on disk). Hence when the optimizations to the SQL-OR based approaches is considered; the optimized Gather count approach (IM_Gcnt) is the best in all the optimizations. Also in most of the cases IM_Gcnt has been found to be the best of all the all the approaches and their optimizations (including those for SQL-92 based approaches).

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. 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 more efficiently. A natural extension to this work would be trying to mix these optimizations at different passes i.e. a mixed optimization may consider using some SQL-92 based approach for support counting for first couple of passes and then switch to some SQL-OR approach for support counting.

We can then also try evaluating these mixed approaches with the current SQL-92 based implementations; if the results are comparable, it can be included in the meta-data.

We have gathered some interesting observations on how the DBMSs generate plans for various queries that are used in mining. Many a times, the query plan chosen does not seem to be the right one as the estimated cost does not seem to match the actual cost. This has provided several insights into the working of the current optimizers and how they need to be modified. If it were possible to suggest hints for certain optimizations, that could have been exploited in goading the optimizer to generate more efficient plans.

The long-term goal is to build a single optimizer for SQL queries that may include OLAP and Mining components. Most of the results reported in this thesis will be useful for that purpose.

APPENDIX A

WRITING STORED PROCEDURES FOR ORACLE

CHAPTER 7

WRITING STORED PROCEDURES FOR ORACLE

7.1 Writing Java Stored Procedures for Oracle

Writing Java Stored Procedures for Oracle consists of following steps:

1. Writing Java File
2. Loading Java Class File in Oracle Server
3. Publishing the Java Methods for calling them.
4. Calling the stored Procedure.

1. Writing Java File (say TestManager.java)

Implement all database updations that have to be done (similar / related things) as methods of this file. This is very similar to any java file. The stored procedures are generally tested by first writing them as any general java file and only after verification, we load it in the Oracle Server. The only thing that needs to be changed while loading the file, is that all statements, where "Connection" object are created, are changed to as shown below:

```
Connection conn = DriverManager.getConnection("jdbc:default:connection:");
```

2. Loading Java File in Oracle Server

Once the Java file to be loaded has been implemented and tested, then loading can be done from any client. We use the "loadjava" command for this. An example is shown below:

```
loadjava -u scott/tiger@paris:1521:orcl -v -r -t TestManager.java
```

Similarly, if this file has to be dropped from the Oracle Server, use the "dropjava" command. Example below shows the syntax for dropjava:

```
dropjava -u scott/tiger@paris:1521:orcl -v  -t TestManager.java
```

3. Publishing the Java Method for calling them.

After loading the Java stored procedures, you must publish the methods of your java program that you want to use as stored procedures, in the Oracle data dictionary. To do that, you write call specs, which map Java method names, parameter types, and return types to their SQL counterparts.

The methods in the Java class TestManager.class are logically related, so you can group their call specs in a PL/SQL package. First, you create the package spec, as follows:

```
CREATE OR REPLACE PACKAGE t_mgr AS
  PROCEDURE Some_method1 (param1 type1, param2  type2,…, paramN  typeN);
  PROCEDURE Some_method2 (param1 type1, param2  type2,…, paramN  typeN);
                                    :
  PROCEDURE Some_methodN (param1 type1, param2  type2,…, paramN  typeN);
End t_mgr;
```

Where Some_method1, Some_method2,…, Some_methodN are the method names you want to use for registering your Java methods (They can be different than the method names used in your TestManager.Java file). These methods names will be used to invoke your stored procedures. param1, param2,…, paramN are the parameters passed to these methods. type1, type2,…, typeN are the corresponding data types for these parameters. These are PL/SQL data types that are equivalent of the Java data types. For example for “int” data type the equivalent PL/SQL data type is NUMBER and for String it is VARCHAR2.

Then, you create the package body as shown below, by writing call specs for the Java methods:

```
CREATE OR REPLACE PACKAGE BODY t_mgr AS
    PROCEDURE Some_method1 (param1 type1, param2  type2,…, paramN  typeN)
    AS   LANGUAGE JAVA
    NAME 'TestManager.method1(Type1, Type2,…, TypeN)';

    PROCEDURE Some_method2 (param1 type1, param2  type2,…, paramN  typeN)
    AS   LANGUAGE JAVA
    NAME 'TestManager.method1(Type1, Type2,…, TypeN)';
```

PROCEDURE Some_methodN (param1 type1, param2 type2,…, paramN typeN)
AS LANGUAGE JAVA
NAME 'TestManager.methodN(Type1, Type2,…, TypeN)';

END t_mgr;

Here Type1, Typ2, TypeN are the Java data types (same as the ones used in corresponding methods of your Java Implementation File.

4. Calling the Stored Procedures

Now, you can call your Java stored procedures from the top level and from database triggers, SQL DML statements, and PL/SQL blocks. To reference the stored procedures in the package *t_mgr*, you must use dot notation.

Example: t_mgr.Some_method1(2010, 'camshaft', 245.00);

7.2 Inserting CLOBS in an Oracle Table or Updating Inside a Result Set.

Inserting new CLOBS in Oracle Table isn't very straightforward. Following describes the steps to follow for writing CLOBS.

1. Insert Empty CLOBS in the table

If suppose the table used is MyTable and it has following schema (item: int, cnt: int and TidList: ClOBS) then following SQL shows how to do this:

```
INSERT    INTO MyTable VALUES (item1, 0, EMPTY_CLOB())
```

(here item1 is a variable of type int)

2. Lock the tuple in which you want to write the CLOB

The following SQL locks the rows for updation:

```
ResultSet rset =    SELECT cnt, TidList
                    FROM myTable
                    WHERE item = pitem
                    FOR UPDATE;
```

Here *pitem* is a variable containing the value of the item used to uniquely identify a single tuple in the relation MyTable. The “For Update” clause locks the particular tuple identified by the “Select” clause.

3. Get the handler to the CLOB object in the tuple and update it.

To update the column having CLOB values, first we need to get the handler to that CLOB and then update it using that handler. The following JDBC code is used for the same.

```
try
{
     oracle.sql.CLOB c = null;
     while(rset.next())
          c = ((CLOB)(rset.getObject(2)));
     java.io.Writer w = c.getCharacterOutputStream();
     w.write(tidString);
     w.close();
}
catch (SQLException sqlex)
{
     sqlex.printStackTrace();
}
```

4. Update any other column and return the Update Lock

For updating any other column of the MyTable relation, use the “Update” SQL. Once the row is updated, the Lock on it is released by the RDBMS. An example of the same is shown below:

```
UPDATE MyTable SET cnt = count where item = pitem
```

This updates the *cnt* column and releases the lock from the tuple uniquely identified by the *item* column.

7.3 Code for Different Stored Procedures

This section lists the Java code used in implanting the stored procedures in Oracle.

7.3.1 CountAndK Stored Procedure

This stored procedure, receives k TidLists (CLOB objects) and returns the count of common transactions in them. This method inserts all the inputs received in a Vector and passes it to the CountK method, which basically does the counting of the common transactions. The Java code below shows this for CountAnd2 procedure, which is mapped to Count2 method of the Java Class file.

```
public static int Count2(oracle.sql.CLOB c1, oracle.sql.CLOB c2)
{
   Vector input = new Vector();
   input.addElement(c1);
   input.addElement(c2);
   int count = CountK(input);
   return count;
}


private static int CountK(Vector in)
{
   int K = in.size();
   StringBuffer tidBuffer = new StringBuffer();
   String tidString[] = new String[K];
   Vector[] tidVector = new Vector[K];
   int count =0;
   try
   {
    for(int i =0; i< K; i++)
    {
     Reader r = ((CLOB)in.elementAt(i)).getCharacterStream();
     int ch =0;
     while((ch = r.read()) > 0)
         tidBuffer.append((char)ch);

     tidString[i] = tidBuffer.toString();
     r.close();
     tidBuffer.delete(0,tidBuffer.length());

     StringTokenizer st1 = new StringTokenizer(tidString[i], ",");
     tidVector[i] = new Vector();

     while(st1.hasMoreTokens())
        tidVector[i].addElement(st1.nextToken());
    }
   }
```

```
    catch (IOException ioex)
    {
      ioex.printStackTrace();
    }
    catch(SQLException sqlex)
    {
      sqlex.printStackTrace();
    }
    boolean isPresent;

    for(int i =0; i < tidVector[0].size(); i++)
    {
      String tid = tidVector[0].elementAt(i).toString();
      isPresent = true;
      int j = 1;
      while((j < in.size()) &&(isPresent))
      {
        if(! (tidVector[j].contains(tid)))
          isPresent = false;
        j++;
      }
      if(isPresent)
        count++;
      else
        isPresent = true;
    }
    return count;
}
```

7.3.2 CombinationK Stored Procedure

The CombinationK stored procedure receives the name of the input dataset as a parameter. It reads the input dataset ordered by Tid and collects all the item ids belonging to a transaction in a Vector and generates items combinations of length K from it. This is shown by the Java code below for Combination2 procedure.

```
public static void Combination2(String tabName, String type)
{
  String sql = "";
  ResultSet rSet = null;
  Statement stmt = null, stmt2 = null;
  Connection conn = null;
  PreparedStatement pstmt = null;
  int v_tid = 0, v_item = 0, v_pTid= -1;

  itemList.removeAllElements();
  try
  {
    conn = DriverManager.getConnection(url2);
    sql = "Select * from " + tabName + " order by tid";
    String sql2 = "Insert into C values (?, ?, ?)";
    stmt = conn.createStatement();
```

```
    pstmt = conn.prepareStatement(sql2);
   }
   catch(SQLException sqlex1)
   {
     sqlex1.printStackTrace();
   }
   try
   {
    rSet = stmt.executeQuery(sql);
    rSet.next();
    v_tid = rSet.getInt(1);
    v_item = rSet.getInt(2);
    v_pTid = v_tid;
    itemList.addElement(new Integer(v_item));

    while(rSet.next())
    {
     v_tid = rSet.getInt(1);
     v_item = rSet.getInt(2);
     if(v_pTid == v_tid)
       insertIntoVector(v_item);
     else
     {
      for(int i =0; i < itemList.size()-1; i++)
        for(int j = i+1; j < itemList.size(); j++)
        {
          pstmt.setInt(1, v_pTid);
          pstmt.setInt(2, ((Integer)(itemList.elementAt(i))).intValue());
          pstmt.setInt(3, ((Integer)(itemList.elementAt(j))).intValue());
          pstmt.executeUpdate();
        }
      itemList.removeAllElements();
      itemList.addElement(new Integer(v_item));
      v_pTid = v_tid;
     }
    }

    for(int i =0; i < itemList.size()-1; i++)
      for(int j = i+1; j < itemList.size(); j++)
      {
       pstmt.setInt(1, v_pTid);
       pstmt.setInt(2, ((Integer)(itemList.elementAt(i))).intValue());
       pstmt.setInt(3, ((Integer)(itemList.elementAt(j))).intValue());
       pstmt.executeUpdate();
      }
    itemList.removeAllElements();

    rSet.close();
    stmt.close();
    pstmt.close();
    conn.close();
   }
   catch(SQLException e)
   {
    System.err.println(e.getMessage());
    e.printStackTrace();
   }
  }
```

APPENDIX B

WRITING USER DEFINED FUNCTIONS FOR IBM DB2/UDB

CHAPTER 8

WRITING USER DEFINED FUNCTIONS FOR IBM DB2/UDB

DB2 udfs can be written as a Java file. A basic difference with any Java file and the one that implements udfs is that the Java class has to extend from COM.ibm.db2.app.UDF class. There are two types of user defined functions in DB2 – column udfs and table udfs. Following section covers writing DB2 udfs.

8.1 Writing column udfs

A column udf in DB2 is the one that can take some input parameters and return a single value or a column. The value that is returned from a column udf is also mentioned in the parameter list of the column udf along with the input parameters. It is implemented as a method of a Java file that extends from COM.ibm.db2.app.UDF class, which provides with setter methods to set the value of the return parameter. The Java code below shows the CountAnd2 udf. This is implemented as CountAnd2 method of the Java class file and like the Oracle CountAndK stored procedure, calls CountK method to do the actually counting.

```
public void CountAnd2(Clob tids1, Clob tids2, int count) throws Exception
  {
    vArray.addElement(tids1);
    vArray.addElement(tids2);
    count = CountK(vArray);
    if(needToSet(3)) set(3,count); // to set the value of the third (output) parameter
    vArray.removeAllElements();
  }
```

8.2 Writing table udfs

Table udfs are used to return multiple values that correspond to the columns of an intermediate relation. This intermediate relation can be used in any SQL as any other base relation. The way table udfs are called internally by the system is very different from the way it is done for column udfs. Table udfs have 5 distinct call types. These are explained below.

SQLUDF_TF_FIRST: The table udf is invoked with this call type, when it is called for the first time by DB2. This is the best place to initialize variables that are independent to the type of call.

Once the udf is called with the *SQLUDF_TF_FIRST* call type, then for every input tuple a call to udf is made. It is called by DB2 with the following 3 call types in the given order:

SQLUDF_TF_OPEN: This section of udf is place for initializing variables that are useful to that particular call.

SQLUDF_TF_FETCH: Until the udf is called for the next input tuple, the DB2 keeps invoking the udf, with this call type. The significance of this is that, the table udfs can return more than one tuple for every one input tuple. For returning a tuple, once the output variables are set, the state of the udf is set to *"00000"*. This indicates a successful return of a tuple. Once the condition to read the next tuple is set, the state of the udf is set to *"02000"*. If the DB2 finds that the state of the udf is set to *"02000"*, it calls the udf for the next input tuple.

SQLUDF_TF_CLOSE: Before invoking the udf again for the next input tuple, the udf is called with this call type. The code in this section of the udf is used to reset the values of any initialization variables.

Finally when the udf has been invoked for all the input values, then before returning, it calls the udf last time with the call type set to *SQLUDF_TF_FINAL*.

SQLUDF_TF_FINAL: The code in this section is executed to finally clear all the variables initialized for this udf.

In table udfs, like the column udfs, the column values to be returned are listed in the parameter list along with the input parameter. The Java code below shows the implementation for SaveTid udf. This udf receives 3 inputs: item, tid and the total number of tuples for which it will be invoked. For each input tuple the udf returns three values – the item id, list of common transactions (TidList) and their count.

```
public void SaveTid (int item, int tid, int tableSize, int T_item, int T_cnt,
                COM.ibm.db2.app.Clob TidList) throws Exception
{
   switch (getCallType())
   {
    case SQLUDF_TF_FIRST:
       iter = 0; // this for the first time
       tupleNum = 0;
       prevItem = -1;
       break;
    case SQLUDF_TF_OPEN:
        iter = 0; // so that the FETCH is invoked only once for every input tuple
        tupleNum++;
       break;
    case SQLUDF_TF_FETCH:
        if(iter == 1)
          setSQLstate("02000"); // read next tuple from the input
        else
        {
         if(tupleNum == 1) // if this is the first tuple from the input table
         {
          v1.addElement(new Integer(tid));
          prevItem = item;
          iter++;
         }
         else
         if(prevItem == item)
         {
          if(tupleNum == tableSize)
          {// if this is the last tuple of the input table
          // add the last element to the vector and write it to the CLOB
            v1.addElement(new Integer(tid));
            tidString = v1.elementAt(0).toString();
            for (int i = 1; i < v1.size(); i ++)
              tidString = tidString + "," + v1.elementAt(i).toString();
            returnClob = COM.ibm.db2.app.Lob.newClob();
            resultWriter = returnClob.getWriter();
            resultWriter.write(tidString.toCharArray());
            if(needToSet(4)) set(4,prevItem);
            if(needToSet(5)) set(5,v1.size());
            if(needToSet(5)) set(6,returnClob);
            setSQLstate("00000"); // write a tuple to the output.
```

```
            iter++;
           }
           else
           {
            v1.addElement(new Integer(tid));
            iter++;
           }
          }
          else // prevItem != item
          {
           tidString = v1.elementAt(0).toString();
           for (int i = 1; i < v1.size(); i ++)
            tidString = tidString + "," + v1.elementAt(i).toString();
           returnClob = COM.ibm.db2.app.Lob.newClob();
           resultWriter = returnClob.getWriter();
           resultWriter.write(tidString.toCharArray());
           if(needToSet(4)) set(4,prevItem);
           if(needToSet(5)) set(5,v1.size());
           if(needToSet(5)) set(6,returnClob);
           setSQLstate("00000"); // write a tuple to the output.
           v1.removeAllElements();
           v1.addElement(new Integer(tid));
           prevItem = item;
           if(tupleNum == tableSize)
           {
            tidString = v1.elementAt(0).toString();
            for (int i = 1; i < v1.size(); i ++)
             tidString = tidString + "," + v1.elementAt(i).toString();
            returnClob = COM.ibm.db2.app.Lob.newClob();
            resultWriter = returnClob.getWriter();
            resultWriter.write(tidString.toCharArray());
            if(needToSet(4)) set(4,prevItem);
            if(needToSet(5)) set(5,v1.size());
            if(needToSet(5)) set(6,returnClob);
            setSQLstate("00000"); // write a tuple to the output.
           }
           iter++;
          }
         }
         break;
    case SQLUDF_TF_CLOSE:
         break;
    case SQLUDF_TF_FINAL:
          v1.removeAllElements();
         break;
   }
} // end of saveTid Method
```

REFERENCES

1. Thuraisingham, B., *A Primer for Understanding and Applying Data Mining.* IEEE, 2000. Vol. 2, No.1: p. 28-31.

2. Thomas, S., *Architectures and optimizations for integrating Data Mining algorithms with Database Systems,* in *CSE*. 1998, University of Florida: Gainesville.

3. Agrawal, R., T. Imielinski, and A. Swami. *Mining Association Rules between sets of items in large databases*. in *ACM SIGMOD International Conference on the Management of Data*. 1993. Washington, D.C.

4. Agrawal, R. and R. Srikant. *Fast Algorithms for mining association rules*. in *20th Int'l Conference on Very Large Databases (VLDB)*. 1994.

5. Sarasere, A., E. Omiecinsky, and S. Navathe. *An efficient algorithm for mining association rules in large databases*. in *21st Int'l Cong. on Very Large Databases (VLDB)*. 1995. Zurich, Switzerland.

6. Shenoy, P., et al. *Turbo-charging Vertical Mining of Large Databases*. in *ACM SIGMOD Int'l Conference on Management of Data*. 2000. Dallas.

7. Han, J., J. Pei, and Y. Yin. *Mining Frequent Patterns wihtout Candidate Generation*. in *ACM SIGMOD Int'l Conference on Management of Data*. 2000. Dallas.

8. Houtsma, M. and A. Swami. *Set-Oriented Mining for Association Rules in Relational Databases*. in *11th International Conference on Data Engineering (ICDE)*. 1995.

9. Han, J., et al. *DMQL: A data mining query language for relational database*. in *ACM SIGMOD workshop on research issues on data mining and knowledge discovery*. 1996. Montreal.

10. Meo, R., G. Psaila, and S. Ceri. *A New SQL-like Operator for Mining Association Rules*. in *Proceedings of the 22nd VLDB Conference*. 1996. Mumbai, India.

11. Agrawal, R. and K. Shim, *Developing tightly-coupled Data Mining Applications on a Relational Database System*. 1995, IBM Almaden Research Center: San Jose, California.

12. Sarawagi, S., S. Thomas, and R. Agrawal. *Integrating Association Rule Mining with Rekational Database System: Alternatives and Implications*. in *ACM SIGMOD Int'l Conference on Management of Data*. 1998. Seattle, Washington.

13. Dudgikar, M., *A Layered Optimizer or Mining Association Rules over RDBMS*, in *CSE Department*. 2000, University of Florida: Gainesville.

BIOGRAPHICAL INFORMATION

Pratyush Mishra was born November 25, 1976 in Mumbai, India. He received his Bachelor of Engineering degree in Computer Science and Engineering from Amravati University, Maharashtra, India in September 1998. In the Fall of 2000, he started his graduate studies in Computer Science and Engineering at The University of Texas, Arlington. He received his Master of Science in Computer Science and Engineering from The University of Texas at Arlington, in December 2002. His research interests include Decision support systems, Data mining and Business Intelligence.