ARCHITECTURES AND OPTIMIZATIONS FOR INTEGRATING DATA MINING ALGORITHMS WITH DATABASE SYSTEMS

By

SHIBY THOMAS

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1998

To
My parents, sisters and brothers

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to Sharma for his encouragement and support throughout my dissertation and my stay with his group. We have had endless arguments and discussions about various things. Especially during the group meetings, this might have invited the wrath of several other students because, at times, it would have even tested their patience. On the personal side he and his family have been my good friends also. I am greatly indebted to Rakesh Agrawal and Sunita Sarawagi of IBM Almaden Research Center for their help and giving me an opportunity to work with their group. The several discussions I had with them while I was at IBM and later through e-mail were very useful. The work that I did with them contributed to a good part of my dissertation. It was a nice experience working with the Quest data mining group at Almaden and the enthusiasm and hard work of Sunita were contagious. I am also thankful to Sanjay Ranka for his help and suggestions during my initial work on data mining. I thank Professors Eric Hanson, Sartaj Sahni, Stanley Su and Suleyman Tufekci for being on my committee and for their comments and suggestions.

I am grateful to many other people for helping me in several ways. In particular I thank Raja, Sreenath, Mokhtar, Nabeel, Roger, and all my friends at the database center for the chat sessions, lunch sessions, and so on. Many thanks to Sharon Grant for her help with everything. She takes care of everything in the database center and her tireless spirit and attention to even the minute details makes things a whole lot easier. My sincere thanks

to S. Seshadri, my master's thesis advisor. My first exposure to database research was while working with him and he is partially responsible for my pursuing further studies. I owe my graduate studies to my parents, sisters and brothers for their continual reassurance.

This work was supported in part by the research grants of Sharma Chakravarthy from the Office of Naval Research and the SPAWAR System Center – San Diego, Rome Laboratory, DARPA and the NSF grant IRI-9528390. During my first year Theodore Johnson provided support through his research grants until he left the University. His initial support helped me to come to the United States for graduate studies. The CISE department also provided me with teaching assistantships in times of need. I gratefully acknowledge all the support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

GENERAL AUDIENCE ABSTRACT

Dissertation Title	: Architectures and Optimizations for Integrating Data Mining Algorithms with Database Systems
Student's Name	: Shiby Thomas
Phone Number	: (352) 374-2477
Department	: Computer and Information Science and Engineering
Committee Chair	: Dr. Sharma Chakravarthy
Degree	: Ph.D.
Semester of Graduation	: Fall, 1998

Data mining or Knowledge Discovery in Databases (KDD) is the process of discovering useful, understandable and previously unknown information or patterns from data. Typical data mining applications include credit approval, fraud detection, discovering customer trends in mail order data and supermarket data, medical research, and many other business applications. A database is a collection of interrelated data and a database management system (DBMS) provides tools for users to query and manipulate the data stored in a database.

The technological advances have made it possible to collect and store huge volumes of data in several business and scientific domains which are stored and manipulated using customized database management systems that are also called data warehouses. In the coming years, database/data warehouse systems will be called upon to deliver return-of-investment in the form of nuggets of knowledge or better insights about the data they store and manage. According to the vice president of Mellon Bank's advanced technology group, "Data mining is the carrot that justifies the expensive stick of building a data warehouse".

This dissertation presents techniques to apply data mining on data stored in a DBMS or a data warehouse. We also identify enhancements to current database technology to enable more efficient data mining.

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

ARCHITECTURES AND OPTIMIZATIONS FOR INTEGRATING
DATA MINING ALGORITHMS WITH DATABASE SYSTEMS

By

Shiby Thomas

December, 1998

Chairman: Dr. Sharma Chakravarthy
Major Department: Computer and Information Science and Engineering

Data mining on large data warehouses is becoming increasingly important. In support of this trend, we consider a spectrum of architectural alternatives for integrating mining with database systems. These alternatives include loose-coupling through a SQL cursor interface; encapsulation of the mining algorithm in a stored procedure; caching the data to a file system on-the-fly and mining; tight-coupling using primarily user-defined functions; and SQL implementations for processing in the DBMS. First, we comprehensively study the option of expressing the association rule mining algorithm in the form of SQL queries. We consider four options in SQL-92 and six options in SQL enhanced with object-relational extensions (SQL-OR). Our evaluation of the different architectural alternatives shows that from a performance perspective, the `Cache-Mine` option is superior, although the SQL-OR option comes a close second. Both the `Cache-Mine` and the SQL-OR approaches incur a higher storage penalty than the loose-coupling approach which performance-wise

is a factor of 3 to 4 worse than `Cache-Mine`. We also compare these alternatives on the basis of qualitative factors like automatic parallelization, development ease, portability and interoperability.

We further analyze the SQL-92 approaches with the twin goals of studying how best can a DBMS without any object-relational extensions execute these queries and to identify ways of incorporating the semantics of mining into cost-based query optimizers. We develop cost formulae for the mining queries based on the input data parameters and relational operator costs. We also identify certain optimizations which improve the performance. Next, we study generalized association rule and sequential pattern mining and develop SQL formulations for them there by demonstrating that more complex mining operations can be handled in the SQL frame work.

We develop an incremental association rule mining algorithm which does not need to examine the old data if the frequent itemsets do not change. Even otherwise, access to the old database can be limited to just one scan. We categorize the various kinds of constraints on the items that are useful in the context of interactive mining to facilitate goal-oriented mining. We show how the incremental mining technique can be adapted to handle constraints and certain kinds of constraint relaxation. We also show the applicability of the incremental algorithm to other classes of data mining and decision support problems. Finally, we identify certain primitive operators that are useful for a large class of data mining and decision support applications. Supporting them natively in the DBMS could enable these applications to run faster.

CHAPTER 1
INTRODUCTION

The rapid growth in data warehousing technology combined with the significant drop in storage prices has made it possible to collect large volumes of data about customer transactions in retail stores, mail order companies, banks, stock markets, telecommunication companies and so on. For example, AT&T call records are about 1 giga byte per hour [73] and super market chains like WalMart collect tera bytes of data. In order to transform this huge amounts of data into business competitiveness and profits, it is extremely important to be able to mine nuggets of useful and understandable information from these data warehouses. In this chapter, we introduce data warehousing and data mining technologies in Sections 1.1 and 1.2 respectively and in Section 1.3, we motivate the need for coupling the two which is the focus of this dissertation. In Section 1.4, we discuss and list the specific problems addressed in this dissertation and in Section 1.5, we outline the dissertation organization.

1.1 Data Warehousing

A *data warehouse* is simply a single, complete and consistent store of data obtained from a variety of sources and made available to end users in a way they can understand and use. Achieving completeness and consistency of data in today's information systems environment is far from simple. The first problem is to discover how completeness and consistency can be defined. In the business context, this entails understanding the business

strategies and the data required to support and track their achievement. This process—called enterprise modeling—requires substantial involvement of business users and is traditionally a long-term process. A data warehouse consists of several components and tools which are depicted in Figure 1.1 [34].

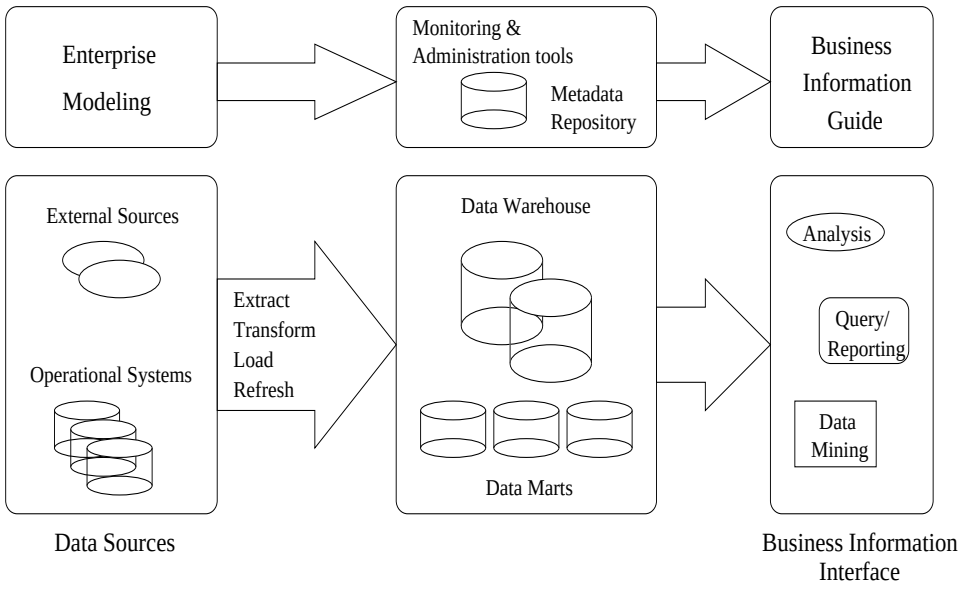


Figure 1.1. Data warehousing architecture

Knowing what data are required is just the first step. These data exist today in various sources on different platforms, and must be copied from these sources for use in the warehouse. They must be combined according to the enterprise model, even though it was not originally designed to support such integration. They must be cleansed of structural and content errors. This step—typically known as data warehouse population—requires tools for extracting data from multiple operational databases and external sources, for cleaning, transforming and integrating these data; and for loading data into the data warehouse. In addition to the main warehouse, there may be several departmental data marts. It also requires tools for periodically refreshing the warehouse to maintain consistency and to purge data from the warehouse, perhaps onto slower archival storage.

In order to understand and profitably use the data in a business context, they must be transformed into information. The warehouse data are stored and managed by one or more warehouse servers, which present multidimensional views of the data to a variety of front end tools: query tools, report writers, analysis tools and data mining tools. There is also a repository which stores metadata derived using the enterprise model, and tools for monitoring and administering the warehousing system.

The warehouse may be distributed for load balancing, scalability and higher availability. In such a distributed architecture, the metadata repository is usually replicated with each fragment of the warehouse, and the entire warehouse is administered centrally. An alternative architecture is a federation of data warehouses or data marts, each with its own repository and decentralized administration.

1.2 Data Mining

Data mining, also referred to as knowledge discovery in databases, is the process of extracting implicit, understandable, previously unknown and potentially useful information from data. In other words, data mining is the act of drilling through huge volumes of data in order to discover relationships, or to answer specific questions that are too broad in nature for traditional query tools. Data mining is also projected as the next step beyond online analytical processing (OLAP) for querying data warehouses. Rather than seek out known relationships, it sifts through data for unknown relationships. For instance, consider the transaction data in a mail-order company stored in the following relations: *sales(customer, widget, state, year), booster(customer, widget, driver), catalog(widget, manufacturer).* The sale information is stored in the *sales* table and the *catalog* table stores the widgets of different manufacturers. The *booster* table stores the driver that influences the particular

sale. In this database, OLAP finds answers to queries of the form "How many widgets were sold in the first quarter of 1998 in California vs. Florida?" However, data mining attempts to answer queries like "What are the drivers that caused people to buy these widgets from our catalog?" Fundamentally, data mining is statistical analysis and has been in practice for a long time. But, until recently, statistical analysis was a time-consuming, manual process which limited the amount of data that could be analyzed and the accuracy depended heavily on the personnel involved in the analysis. Today, with the advent of various sophisticated technologies, tools exist that automate the process, making data mining a practical solution for a wide range of companies. For example, Fingerhut's (a direct-mail catalog company) statistical analysis was limited to taking samples of 10 to 20 percent of its customers. With data mining, it can examine 300 specific characteristics of each of the 10 to 12 million customers in a much more focused way [15].

The initial efforts on data mining research were to cull together techniques from machine learning and statistics [24, 28, 29, 37, 39, 40, 60, 65, 81, 90, 95, 102, 103, 104] to define new mining operations and develop algorithms for them [5, 7, 19, 77]. In the remainder of this section, we briefly introduce the various data mining problems.

1.2.1 Association Rule

Association rule which captures co-occurrence of items or events was introduced in the context of market basket data [7]. An example of such a rule might be that 60% of transactions containing beer also contain diapers and 2% of transactions contain both these items. Here 60% is the *confidence* and 2% is the support of the rule $beer \Longrightarrow diaper$.

Association rule mining is stated formally as follows [13]. Let $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ be a set of literals, called items. Let $\mathcal{D}$ be a set of transactions, where each transaction T is

a set of items such that $T \subseteq \mathcal{I}$. Each transaction has a unique identifier, called its *tid*. An association rule is an implication of the form $X \Longrightarrow Y$, where $X \subset \mathcal{I}$, $Y \subset \mathcal{I}$, and $X \cap Y = \emptyset$. The rule $X \Longrightarrow Y$ holds in the transaction set $\mathcal{D}$ with *confidence* c if $c\%$ of transactions in $\mathcal{D}$ that contain X also contain Y. The rule $X \Longrightarrow Y$ has *support* s in the transaction set $\mathcal{D}$ if $s\%$ of transactions in $\mathcal{D}$ contain $X \cup Y$. Given a set of transactions $\mathcal{D}$, the problem of mining association rules is to generate all association rules that have support and confidence greater than the user-specified minimum support and minimum confidence.

The association rule mining problem has attracted tremendous attention from data mining researchers and several algorithms have been proposed for it [13, 26, 66, 111, 131]. Researchers have also proposed several variants of the basic association rule mining to handle taxonomies over items [63, 118], numeric attributes [76, 92, 119] and constraints on items appearing in rules [97, 121].

1.2.2 Sequential Patterns

Sequential pattern mining was first introduced in Agrawal and Srikant [14] and further generalized in Srikant and Agrawal [120]. Given a set of data-sequences each of which is a list of transactions ordered by the transaction time, the problem of mining sequential patterns is to discover all sequences with a user-specified minimum *support*. Each transaction contains a set of items. A sequential pattern is an ordered list (sequence) of itemsets. The itemsets that are contained in the sequence are called *elements* of the sequence. For example, $\langle (computer, modem)(printer) \rangle$ is a sequence with two elements; $(computer, modem)$ and $(printer)$. The support of a sequential pattern is the percentage of data-sequences that contain the sequence. A sequential pattern can be further qualified by specifying maximum and/or minimum time gaps between adjacent elements and a sliding time window within

which items are considered part of the same sequence element. These time constraints are specified by three parameters, *max-gap, min-gap* and *window-size*. We refer the reader to Srikant and Agrawal [120] for a formal definition of the problem.

1.2.3 Classification

Classification is a well-studied problem [91, 129]. However, only recently has there been focus on algorithms that can handle large datasets [17, 53, 85, 106, 114]. Classification is the process of generating a description or a model for each class of a given dataset, called the *training set*. Each record of the training set consists of several attributes which could be *continuous* (coming from an ordered domain), or *categorical* (coming from an unordered domain). One of the attributes, called the *classifying* attribute, indicates the *class* to which each record belongs. Once a model is built from the given examples, it can be used to determine the class of future unclassified records.

Several classification models based on neural networks, statistical models like linear/quadratic discriminants, decision trees and genetic models have been proposed over the years. Decision trees are particularly suited for data mining since they can be constructed relatively fast compared to other methods and they are simple and easy to understand. Moreover, trees can be easily converted into SQL statements that can be used to access databases efficiently [5].

A decision tree is a class discriminator that recursively partitions the training set until each partition consists entirely or dominantly of examples from one class. Each non-leaf node of the tree contains a *split point* which is a test on one or more attributes and determines how the data is partitioned. Figure 1.2 shows a sample decision-tree classifier and the training set from which it is derived. Each record represents a credit card applicant

and we are interested in building a model that categorizes the applicants into high and low risk classes.

Age	Salary	Married	Risk
21	30,000	No	High
18	28,000	Yes	High
30	50,000	Yes	Low
35	20,000	Yes	High
23	60,000	No	High
47	80,000	Yes	Low

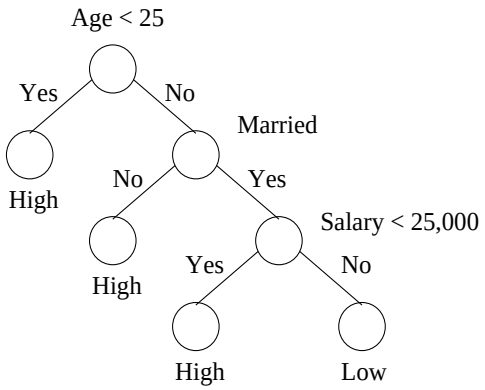


Figure 1.2. Credit card classification example

1.2.4 Clustering

The fundamental clustering problem is that of grouping together (clustering) similar data items and is useful for discovering interesting distributions and patterns in the underlying data. Clustering has been formulated in various ways in the machine learning [48], pattern recognition [51], optimization [112] and statistics literature [20]. The problem of clustering can be defined as follows: given n data points in a d-dimensional metric space, partition the data points into k clusters such that the data points within a cluster are more similar to each other than data points in different clusters. The classic K-Means clustering algorithm starts with estimated initial values for the k cluster centroids. Each of the data points is assigned to the cluster with the nearest centroid. After the assignment the centroids are refined to the mean of the data points in that cluster. This process is repeated several times until an acceptable convergence is reached. There are several research efforts reported in the data mining literature for clustering large databases [23, 42, 57, 132].

Research on time series analysis [10, 43, 75], similarity search [3, 8, 21, 72, 115], high dimensional data analysis [4, 22, 79], and text data management [30, 31] can also be classified under the broad category of data mining.

1.3 Mining Databases

The initial research efforts on data mining were aimed at defining new mining problems and a majority of the algorithms for them were developed for data stored in file systems. Each had its own specialized data structures, buffer management strategies and so on [8, 13, 14, 22, 49, 50, 62, 63, 74, 84, 85, 86, 111, 115, 119, 121]. In cases where the data are stored in a DBMS, data access was provided through an ODBC or SQL cursor interface [2, 64, 68, 71]. Data mining tools are being used in several application domains [16, 18, 27, 38, 41, 46, 52, 70, 82, 96, 101, 108, 130]. Coupling the data mining tools with a growing base of accessible enterprise data—often in the form of a data warehouse—provides business institutions at its disposal a tool with immense implications. According to the vice president of Mellon Bank's advanced technology group, "Data mining is the carrot that justifies the expensive stick of building a data warehouse."

The majority of the *warehouse stores*—systems used for storing warehouse data—are relational databases or their variants. The advantages of using database systems are numerous: SQL was invented for direct query of data, most client/server connectivity is supplied by relational vendors, most replication systems have been designed with relational sources and targets and most of the relational vendors are delivering parallel database solutions. There are important alternatives in this segment, however. The OLAP multidimensional engines offer unique performance characteristics across their problem domain. We might also see traditional file stores providing significantly better performance for some data mining

operations. Differentiators for the relational engines will be overall scalability, availability across a broad spectrum of hardware platforms, "affinity" with legacy data platforms and stores, availability of extensions for image, text and so on to support the next generation of applications, and availability of supporting tools.

The investment in building and managing a data warehouse is enormous. With the advent of business intelligence and decision support systems, it is imperative that the data warehouse support multiple applications. Figure 1.3 illustrates how a data warehouse will typically be used in a business organization. The spectrum of applications include basic querying and reporting tools, OLAP tools, multidimensional analysis tools and data mining tools. There are several excellent and popular query/report writers. There are also tools that support multidimensional analysis directly on relational data stores without the separate OLAP engine. Note that these tools are like tools in a tool box; that is, they can be used in combination to produce the desired result. There is no single class of tools that satisfies the broad range of decision support and business intelligence system requirements. Therefore, it is crucial that the data mining tools integrate with relational data warehouses much like the query/report and OLAP tools.

1.4 Goal

The goal of this dissertation is to explore the various issues of database/data warehouse integration of mining operations. We first study the various architectural alternatives for coupling data mining with relational database systems, primarily from a performance perspective. We develop various SQL formulations for association rules [7], a representative mining problem, and analyze how competitive can the SQL implementation be compared to other specialized implementations of association rule mining. We further focus on the

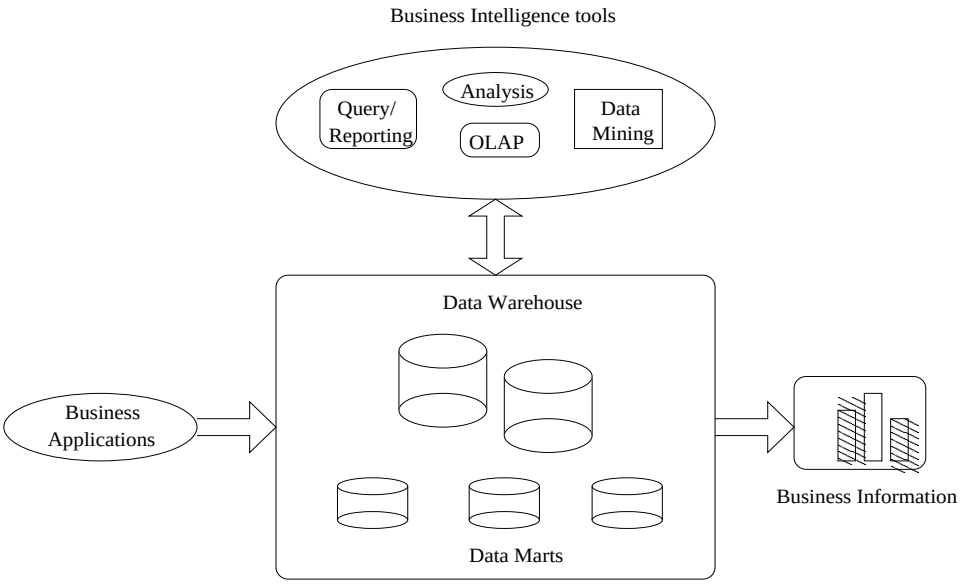


Figure 1.3. Typical data warehouse usage

analysis of various execution plans chosen by relational database systems for executing some of the SQL-based mining queries. We expect that this study will reveal the domain-specific semantic information of the mining algorithms that need to be integrated into next generation query optimizers to handle mining computations efficiently. We also develop SQL formulations for a few other mining operations, namely, generalized association rules [118] and sequential patterns [14, 120]. We also propose a few primitive database operators that are useful for mining and other decision support applications. These operators, if natively supported in a database system, could potentially speed up the execution of mining queries.

Data warehouses on which the mining tools operate typically are populated incrementally. In order to improve the reliability and usefulness of the discovered information, large volumes of data need to be collected and analyzed over a period of time. A naive approach to update the mined information, when new data are added or part of the current data are deleted, is to recompute them from scratch. However, it would be ideal to develop an incremental algorithm so that the computation effort spent on the original data can be

effectively utilized. We develop an incremental algorithm and its SQL formulations for updating the association rules, based on the negative border concept. It is based on the closure property of frequent itemsets, that is, all subsets of a frequent itemset are also frequent. We show that the incremental mining algorithm can be generalized to handle various kinds of constraints. We categorize the constraints based on their usage in the mining process and develop a general framework to process them. We further show that the incremental technique is applicable to other classes of mining and decision support problems such as sequential patterns, correlation rules, query flocks and so on.

We summarize and list the goals of this dissertation below:

- Survey the various data mining problems and algorithms,
- Study the different database integration alternatives for data mining,
- Develop and implement SQL formulations of mining algorithms,
- Analyze the performance profile of current DBMSs to execute the above SQL queries,
- Explore the enhancements to current cost-based optimizers to incorporate the domain-specific semantics of mining,
- Develop an incremental association rule mining algorithm and its SQL formulations,
- Generalize the incremental algorithm for mining constrained associations and show its applicability to other data mining and decision support problems, and
- Explore primitive operators for mining in databases.

This work has a significant impact on the state-of-the-art in data mining system architectures and comes at the appropriate time when the data mining community is looking for

answers to "How to mine data warehouses?" Given the amount of data involved in mining, its potential impact on various business sectors and the fact that OLAP is finding its way into commercial database systems (for example, the *cube* operator), it is only a matter of time before mining becomes an integral part of database systems. We believe that this work is a small but strong step in the right direction. This will also have a significant impact on query optimization and parallel query processing techniques.

1.5 Thesis Organization

The rest of this dissertation is organized as follows: We discuss the various architectural alternatives for integrating mining with database systems/data warehouses in Chapter 2. The various SQL formulations of association rules, their performance profiles and optimizations for improving the performance are detailed in Chapter 3. In Chapter 4, we present the use of object-relational extensions to SQL for improving the performance of SQL-based association rule mining and the performance comparison of the various architectural alternatives. SQL-based mining of generalized association rules and sequential patterns are described in Chapters 5 and 6 respectively. Chapter 7 presents the incremental association rule mining algorithm, performance comparison, SQL formulation and generalizations for mining constrained associations. We conclude in Chapter 8 with a discussion of the proposed database operators and avenues for further research.

CHAPTER 2
FROM FILE MINING TO DATABASE MINING

The "first-generation" KDD systems offer isolated discovery features using tree inducers, neural nets and rule discovery algorithms. Such systems cannot be embedded into a large application and typically offer just one knowledge discovery feature. The current state of data mining systems is very much similar to the days in which database applications were written in COBOL with just *read* and *write* commands as the interface to data stored in large files. The advent of relational database systems which offered SQL for *ad hoc* queries and various relational APIs (application programming interfaces) for application programming made database applications much easier to develop and manage. Data mining has to undergo a similar transition from the current "file mining" to data warehouse mining and a richer set of APIs for developing business intelligence and decision support applications.

In the remainder of this chapter, we survey some of the prior research related to the database integration of mining in Section 2.1. The various architectural alternatives are discussed in Section 2.2.

2.1 Related Work

Researchers have started to focus on various issues related to integrating mining with databases [6, 67, 68]. The research on database integration of mining can be broadly classified into two categories; one which proposes new mining operators and the other which

leverages the query processing capabilities of current relational DBMSs. In the former category, there have been language proposals to extend SQL with specialized mining operators. A few examples are (i) the query language DMQL proposed by Han et al. [64] which extends SQL with a collection of operators for mining characteristic rules, discriminant rules, classification rules, association rules and so on, (ii) The M-SQL language of Imielinski and Virmani [69] which extends SQL with a special unified operator *Mine* to generate and query a whole set of propositional rules, and (iii) the *mine rule* operator proposed by Meo et al. [89] for a generalized version of the association rule discovery problem. However, they do not address the processing techniques for these operators inside a database engine and the interaction of the standard relational operators and the proposed extensions. It is also important to break these operators to a finer level of granularity in order to identify commonalities between them and derive a set of primitive operators that should be supported natively in a database engine.

In the second category, researchers have addressed the issue of exploiting the capabilities of conventional relational systems and their object-relational extensions to execute mining operations. This entails transforming the mining operations into database queries and in some cases developing newer techniques that are more appropriate in the database context. The proposal of Agrawal and Shim [12] for tightly coupling a mining algorithm with a relational database system makes use of user-defined functions (UDFs) in SQL statements to selectively push parts of the application that perform computations on data records into the database system. The objective was to avoid one-at-a-time record retrieval from the database to the application address space, saving both the copying and process context switching costs. In the KESO project [116], the focus is on developing a data mining system which interacts with standard DBMSs. The interaction with the database

is restricted to two-way table queries, a special kind of aggregate query. Two-way tables, which are used in the mining process, have sets of source and target attributes and an associated count. Association rule mining was formulated as SQL queries in the SETM algorithm [66]. However, it does not use the subset property—all subsets of a frequent itemset are frequent—for candidate generation. As a result, SETM counts a large number of candidate itemsets in the support counting phase and hence is not efficient [9]. Query flocks generalizes boolean association rules to mine associations across relational tables. A query flock is a parameterized query with a filter condition to eliminate the values of parameters that are "uninteresting". Tsur et al. [128] present the use of query flocks for mining and emphasizes the need for incorporating the a-priori technique into new generation query optimizers to handle mining queries efficiently.

2.2 Architectural Alternatives

The various architectural alternatives for integrating mining with relational systems, proposed in [109], can be categorized as shown in Figure 2.1. It shows a taxonomy of various alternatives from loose-coupling to an integrated approach. In the remainder of this section, we describe each of the alternatives.

2.2.1 Loose-Coupling

This is an example of integrating mining applications into the client in a client/server architecture or into the application server in a multi-tier architecture. The mining kernel can be considered as the application server. In this approach, data are read tuple by tuple from the DBMS to the mining kernel using a cursor interface. The intermediate and the final results are stored back into the DBMS. The data are never copied on to a file system. Instead, the DBMS is used as a file system. This is the approach followed by most existing

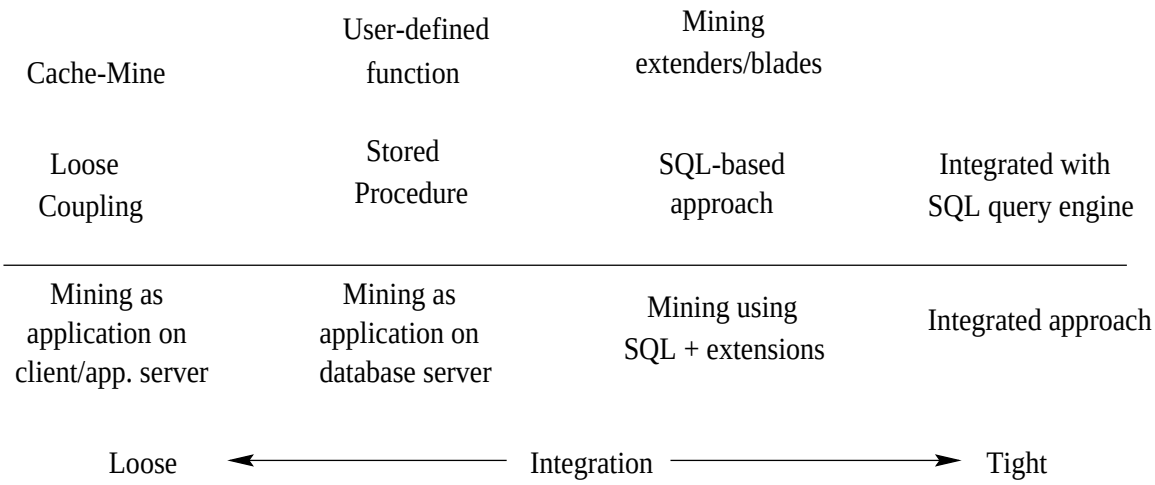


Figure 2.1. Taxonomy of architectural alternatives

mining systems. A potential problem with this approach is the high context switch costs between the DBMS and the mining kernel processes since they run in different address spaces. In spite of the block-read optimization present in many systems (for example, Oracle [98], DB2 [32]) where a block of tuples is read at a time instead of reading a single tuple at a time, the performance could suffer. This architecture is outlined in Figure 2.2

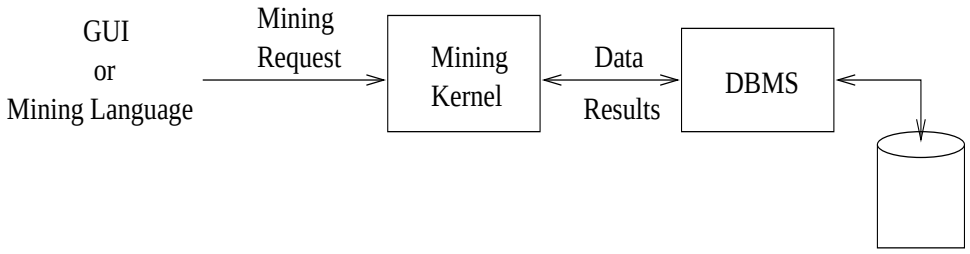


Figure 2.2. Loose-coupling architecture

2.2.2 Cache-Mine

This is a special case of the loose-coupling approach where the mining algorithm reads data from the DBMS only once and caches the relevant data in flat files on local disk for future references. The data could be transformed and cached in a format that enables more efficient access in the future. The mined results, first generated as flat files, are imported

into the DBMS. This method has all the advantages of the stored procedure approach (described below) plus it promises to have better performance. The disadvantage is that it requires additional disk space for caching. This architecture is outlined in Figure 2.3.

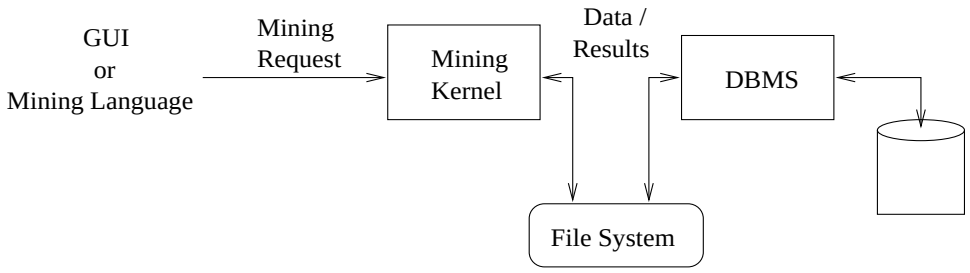


Figure 2.3. Cache-mine architecture

2.2.3 Stored-Procedure

This architecture is representative of the case where the mining logic is embedded as applications on the database server. In this approach, shown in Figure 2.4, the mining algorithm is encapsulated as a stored procedure [32] that is executed in the same address space as the DBMS. The main advantage of this as well as the loose-coupling and cache-mine approach is greater programming flexibility. Also, any existing file system code can be easily transformed to work on data stored in the DBMS.

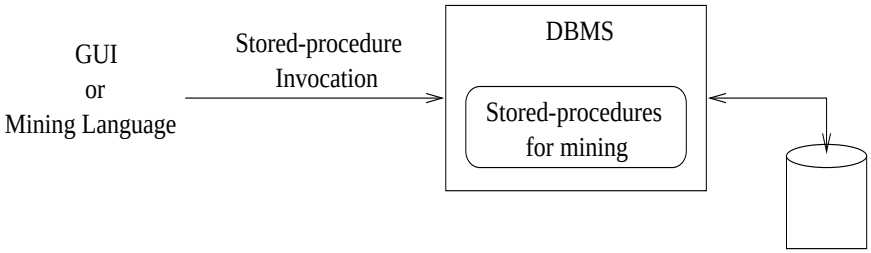


Figure 2.4. Stored-procedure architecture

2.2.4 User-Defined Function

This approach is another variant of embedding mining as an application on the database server if the user-defined functions are run in the unfenced mode (same address space as

the server) [32]. In this case, the entire mining algorithm is encapsulated as a collection of user-defined functions (UDFs) [32] that are appropriately placed in SQL data scan queries. The architecture is represented in Figure 2.5. Most of the processing happens in the UDF and the DBMS is used simply to provide tuples to these UDFs. Little use is made of the query processing capabilities of the DBMS. The UDFs can be run in either fenced (different address space) or unfenced (same address space) mode. The main attraction of this method is performance since when run in the unfenced mode individual tuples never have to cross the DBMS boundary. Otherwise, the processing happens in almost the same manner as in the stored procedure case. The main disadvantage is the development cost [12] since the entire mining algorithm has to be written as UDFs involving significant code rewrites. Further, these are "heavy-weight" UDFs which involve significant processing and memory management.

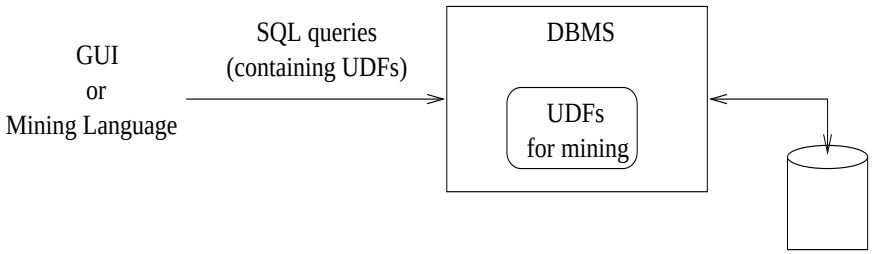


Figure 2.5. UDF-based mining architecture

In order to provide a query interface or application programming interface to the discovered rules, they can be passed through a post-processing step. The rule discovery itself could be done by any of the above alternatives.

2.2.5 SQL-Based Approach

This is the integration architecture explored in this dissertation. In this approach, the mining algorithm is formulated as SQL queries which are executed by the DBMS query

processor. We develop several SQL formulations for a few representative mining operations in order to better understand the performance profile of current database query processors in executing these queries. We believe that it will enable us to identify what portions of these mining operations can be pushed down to the query processing engine of a DBMS.

There are also several potential advantages of a SQL implementation. One can profitably make use of the database indexing and query processing capabilities thereby leveraging on more than two decades of development effort spent in making these systems robust, portable, scalable, and highly concurrent. Rather than devising specialized parallelizations, one can potentially exploit the underlying SQL parallelization, particularly in an SMP environment. The current approach to parallelizing mining algorithms is to develop specialized parallelizations for each of the algorithms [11, 61, 113, 114]. The DBMS support for check-pointing and space management can be especially valuable for long-running mining algorithms on huge volumes of data. The development of new algorithms could be significantly faster if expressed declaratively using a few SQL operations. This approach is also extremely portable across DBMS's since porting becomes trivial if the SQL approaches use only the standard SQL features.

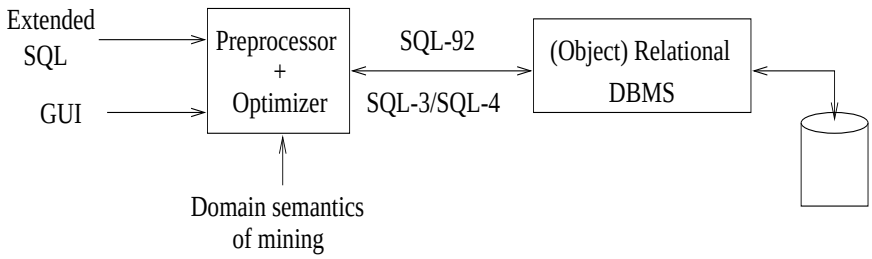


Figure 2.6. SQL architecture for mining in a DBMS

The architecture we have in mind is schematically shown in Figure 2.6. We visualize that the desired mining operation will be expressed in some extension of SQL or a graphical language. A preprocessor will generate appropriate SQL translation for this operation.

This preprocessor will be able to select the right translation taking into account input data distributions. We consider SQL translations that can be executed on a SQL-92 [88] relational engine, as well as translations that require some of the newer object-relational capabilities being designed for SQL [78]. Specifically, we assume availability of blobs, user-defined functions, and table functions [80] in the Object-Relational engine. We do not require any mining specific extension in the underlying execution engine; identification of such extensions is one of the goals of this study.

We do quantitative and qualitative comparisons of some of the architectural alternatives listed here. Our primary focus is on the performance of various architectural alternatives and the identification of possible enhancements to the query optimizer and the query processing engine. The issues of the language constructs required to extend SQL with mining features, and the details of the preprocessing step shown in Figure 2.6 are secondary.

It might also be possible to integrate mining with databases using the newer extension technologies like database extenders, data cartridges or data blades.

2.2.6 Integrated Approach

This is the tightest form of integration where the mining operations are an integral part of the database query engine. In this approach there is no clear boundary between simple querying, OLAP and mining; that is querying and mining are treated to be similar operations. The user's goal is to get information from the data store. He/she should not have to make the distinction as to whether it is the result of querying/OLAP/mining. This entails unbundling the bulky mining operations and identifying common operator primitives with which the mining operations can be composed. We cannot expect to have

a specialized operator for every mining task. It also needs a language in which the required operations can be specified. In order to realize this goal, it requires tremendous amount of research in various aspects like designing language extensions, better query processing and optimization strategies. However, we envision that the query processing engine will eventually be extended with primitive mining operators. When that is accomplished, a mining system architecture will resemble the one shown in Figure 2.7.

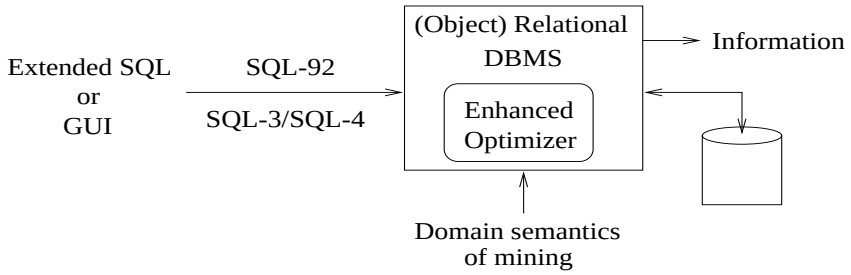


Figure 2.7. Architecture for mining in next-generation DBMSs

2.3 Summary

The first two approaches in the architecture taxonomy in Figure 2.1, namely, mining in the application server or database server, facilitates the move from file mining to database mining rather easily. However, as explained in the loose-coupling, cache-mine, stored-procedure and user-defined function approaches, they do not utilize the query processing functionality provided by the DBMS.

In this dissertation we pursue the third approach in Figure 2.1, which uses SQL and its extensions to implement the mining algorithms. This acts as a pre-cursor to determine the extensions to current query processors and optimizers in order to move towards the last approach which is the truly integrated approach.

Note The architectural alternatives in Section 2.2 were developed primarily by researchers from IBM Almaden Research Center and the author was a contributor.

CHAPTER 3
ASSOCIATION RULES

In this chapter, we discuss the various SQL-92 (SQL with no object-relational extensions) formulations of association rule mining. We start with a review of the apriori algorithm for association rule mining in Section 3.1. A few other algorithms for mining association rules are briefly outlined in Section 3.2. The input-output data formats are described in Section 3.3 and in Section 3.4, we introduce SQL-based association rule mining. The various SQL-92 formulations are presented in Section 3.5. We present experimental results showing the performance of these formulations on some real-life datasets in Section 3.6. In Section 3.7, we develop cost formulae for the cost of executing the above SQL queries on a query processor, based on the input data parameters and relational operator costs. A few performance optimizations to the basic SQL-92 approaches and the corresponding performance gains are presented in Section 3.8. Section 3.9 quantifies the overall performance improvements of the optimizations with experiments on synthetic datasets.

The association rule mining problem outlined in Section 1.2.1 can be decomposed into two subproblems [7].

- Find all combinations of items whose support is greater than minimum support. Call those combinations *frequent* itemsets.
- Use the frequent itemsets to generate the desired rules. The idea is that if, say, $ABCD$ and AB are frequent itemsets, then we can determine if the rule $AB \rightarrow CD$ holds by

computing the ratio $r = \text{support}(ABCD)/\text{support}(AB)$. The rule holds only if $r \geq$ minimum confidence. Note that the rule will have minimum support because $ABCD$ is frequent.

3.1 Apriori Algorithm

We use the Apriori algorithm [9] as the basis for our presentation. There are recent proposals aimed at improving the performance of the Apriori algorithm by reducing the number of data passes [26, 127]. They all have the same basic data-flow structure as the Apriori algorithm. Our goal is to understand how best to integrate this basic structure within a database system.

$F_1 = \{\text{frequent 1-itemsets}\}$
for $(k = 2; F_{k-1} \neq \emptyset; k++)$ do
    $C_k = \text{apriori-gen}(F_{k-1})$; // generate new candidates
    forall transactions $t \in \mathcal{D}$ do
        $C_t = \text{subset}(C_k, t)$; // find all candidates contained in t
        forall candidates $c \in C_t$ do
            c.count++;
        done
    done
    $F_k = \{c \in C_k | c.\text{count} \geq \text{minsup}\}$
done
Answer $= \bigcup_k F_k$;

Figure 3.1. Apriori algorithm

The basic Apriori algorithm shown in Figure 3.1 makes multiple passes over the data. In the k^{th} pass it finds all itemsets with k items having the minimum support, called the frequent k-itemsets. Each pass consists of two phases. Let F_k represent the set of frequent k-itemsets, and C_k the set of candidate k-itemsets (potentially frequent itemsets). First is the **candidate generation** phase where the set of all frequent $(k-1)$-itemsets, F_{k-1}, found in pass $(k-1)$, is used to generate the candidate itemsets C_k. The candidate generation procedure ensures that C_k is a superset of the set of all frequent k-itemsets. The algorithm builds a specialized hash-tree data structure in memory out of C_k. Then is the **support counting** phase where the transaction database is scanned. For each transaction, the algorithm determines which of the candidates in C_k are contained in the transaction using the hash-tree data structure and increments their support count. At the end of the pass, C_k is examined to determine which of the candidates are frequent, yielding F_k. The algorithm terminates when C_{k+1} becomes empty.

3.2 Other Algorithms

There are several other file based algorithms for mining association rules. However many of them follow the basic apriori framework and tries to improve on the computation and I/O requirements.

The partition algorithm [111] reduces the I/O requirements to just two passes over the entire dataset. The reason the database needs to be scanned multiple times is because the number of possible itemsets to be tested for support is exponentially large if it must be done in a single scan of the database. The partition algorithm generates a set of all potentially frequent itemsets in one scan of the database. This set is a superset of all frequent itemsets. In the second scan, the actual support of these itemsets in the whole database is computed.

The algorithm executes in two phases. In the first phase, it divides the database into a number of non-overlapping partitions. The frequent itemsets in each of these partitions are computed separately which will involve multiple passes over the data. However, the partition sizes are chosen such that an entire partition fits in main memory so that it is read only once in each phase. At the end of the first phase, the frequent itemsets from all the partitions are merged together to generate the set of all potentially frequent itemsets. This set is a superset of all the frequent itemsets since all itemsets that are frequent in the whole database have to be frequent at least in one partition. In the second phase, the actual support of these itemsets are counted and the frequent itemsets are identified.

Toivonen proposes a sampling based algorithm [127]. The idea there is to pick a random sample, use it to determine all association rules that probably hold in the whole database, and then to verify the results with the rest of the database. The algorithm thus produces exact association rules in one full pass over the database. In those rare cases where the sampling method does not produce all association rules, the missing rules can be found in a second pass. A superset of the frequent itemsets can be determined efficiently from a random sample by applying any level-wise algorithm on the sample in main memory, and by using a lowered support threshold. In cases where approximate results are sufficient, the sampling approach can significantly reduce the computational and I/O requirements since it works on a much smaller dataset.

A different way of counting support is proposed in Savasere et al. [111] and Zaki et al. [131]. Associated with each item is a *tidlist* which consists of all the transaction identifiers that contain that item. The support for an itemset can be obtained by counting the number of transactions that contain all the items in the itemset. If the tidlists are

kept sorted, this operation can be done by performing a merge-scan of the tidlists of all the items in the itemset.

3.3 Input-Output Formats

Input format The transaction table T normally has two column attributes: transaction identifier *(tid)* and item identifier *(item)*. For a given tid, typically there are multiple rows in the transaction table corresponding to different items that belong to the same transaction. The number of items per transaction is variable and unknown during table creation time. Thus, alternative schemas may not be convenient. In particular, assuming that all items in a transaction appear as different columns of a single tuple [105] is not practical, because often the number of items per transaction can be more than the maximum number of columns that the database supports. For instance, for one of the real-life datasets we experimented with, the maximum number of items per transaction is 872 and for another it is 700. In contrast, the corresponding average number of items per transaction is only 9.6 and 4.4 respectively. Even if the database supports so many columns for a table, there will be lot of space wastage in that scheme.

Output format The output is a collection of rules of varying length. The maximum length of these rules is much smaller than the total number of items and is rarely more than a dozen. Therefore, a rule is represented as a tuple in a fixed-width table where the extra column values are set to NULL to accommodate rules involving smaller itemsets. The schema of a rule is ($item_1, \ldots, item_k$, *len*, *rulem, confidence, support*) where k is the size of the largest frequent itemset. The *len* attribute gives the length of the rule (number of items in the rule) and the *rulem* attribute gives the position of the $\rightarrow$ in the rule. For instance, if $k = 5$, the rule $AB \rightarrow CD$ which has 90% confidence and 30% support is represented by

the tuple $(A,\ B,\ C,\ D,\ NULL,\ 4,\ 2,\ 0.9, 0.3)$. The frequent itemsets are represented the same way as rules but do not have the *rulem* and *confidence* attributes.

3.4 Associations in SQL

In Section 3.4.1 we present the candidate generation process in SQL. In Section 3.4.2 we present the support counting process and in Section 3.4.3 we present the rule generation process.

3.4.1 Candidate Generation in SQL

Recall that the apriori algorithm for finding frequent itemsets proceeds in a level-wise manner. In each pass k of the algorithm we first need to generate a set of candidate itemsets C_k from frequent itemsets F_{k-1} of the previous pass.

Given F_{k-1}, the set of all frequent $(k-1)$-itemsets, the Apriori candidate generation procedure [9] returns a superset of the set of all frequent k-itemsets. We assume that the items in an itemset are lexicographically ordered. Since, all subsets of a frequent itemset are also frequent, we can generate C_k from F_{k-1} as follows.

First, in the *join* step, we generate a superset of the candidate itemsets C_k by joining F_{k-1} with itself as shown below.

insert into C_k select $I_1.item_1, \ldots, I_1.item_{k-1}, I_2.item_{k-1}$

from $F_{k-1}\ I_1, F_{k-1}\ I_2$

where $I_1.item_1 = I_2.item_1$ and

$\vdots$

$I_1.item_{k-2} = I_2.item_{k-2}$ and

$I_1.item_{k-1} < I_2.item_{k-1}$

For example, let F_3 be {{1 2 3}, {1 2 4}, {1 3 4}, {1 3 5}, {2 3 4}}. After the join step, C_4 will be {{1 2 3 4}, {1 3 4 5}}.

Next, in the *prune* step, all itemsets $c \in C_k$, where some $(k-1)$-subset of c is not in F_{k-1}, are deleted. Continuing with the example above, the prune step will delete the itemset {1 3 4 5} because the subset {1 4 5} is not in F_3. We will then be left with only {1 2 3 4} in C_4. We can perform the prune step in the same SQL statement as the join step above by writing it as a k-way join as shown in Figure 3.2. A k-way join is used since for any k-itemset there are k subsets of length $(k-1)$ for which we need to check in F_{k-1} for membership. The join predicates on I_1 and I_2 remain the same. After the join between I_1 and I_2 we get a k-itemset consisting of $(I_1.item_1, \ldots, I_1.item_{k-1}, I_2.item_{k-1})$ as shown above. For this itemset, two of its $(k-1)$-length subsets are already known to be frequent since it was generated from two itemsets in F_{k-1}. We check the remaining $k-2$ subsets using additional joins. The predicates for these joins are enumerated by skipping one item at a time from the k-itemset as follows. We first skip $item_1$ and check if the subset $(I_1.item_2, \ldots, I_1.item_{k-1}, I_2.item_{k-1})$ belongs to F_{k-1} as shown by the join with I_3 in Figure 3.2. In general, for a join with I_r $(3 \leq r \leq k)$, we skip item $r-2$ which gives us join predicates of the form

$$I_1.item_1 = I_r.item1 \text{ and}$$

$$\vdots$$

$$I_1.item_{r-3} = I_r.item_{r-3} \text{ and}$$

$$I_1.item_{r-1} = I_r.item_{r-2} \text{ and}$$

$$\vdots$$

$$I1.item_{k-1} = I_r.item_{k-2} \text{ and}$$

$$I_2.item_{k-1} = I_r.item_{k-1}.$$

Figure 3.3 gives an example for $k = 4$.

We construct a primary index on $(item_1, \ldots, item_{k-1})$ of F_{k-1} to efficiently process these k-way joins using index probes. Note that sometimes it may not be necessary to materialize C_k before the counting phase. Instead, the candidate generation can be pipelined with the subsequent SQL queries used for support counting.

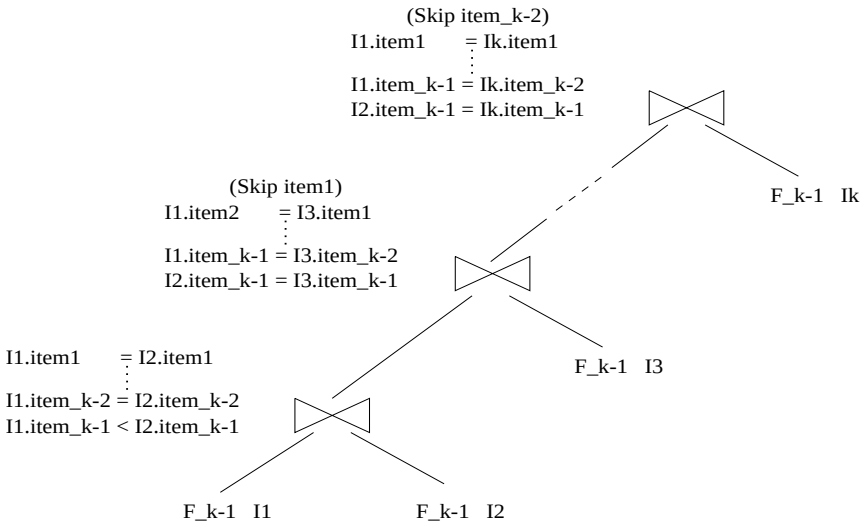


Figure 3.2. Candidate generation for any k

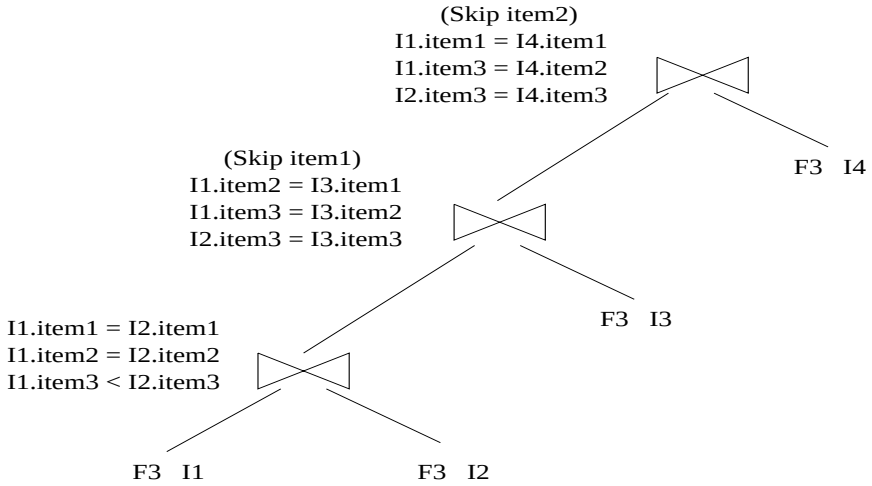


Figure 3.3. Candidate generation for $k = 4$

3.4.2 Counting Support to Find Frequent Itemsets

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

(A) The first one is based purely on SQL-92. We discuss four approaches in this category in Section 3.5.

(B) The second utilizes the new SQL object-relational extensions like UDFs, BLOBs (binary large objects), table functions and so on. *Table functions* [80] are virtual tables associated with a user defined function which generate tuples on the fly. Like normal physical tables they have pre-defined schemas. The function associated with a table function can be implemented as any other UDF. Thus, table functions can be viewed as UDFs that return a collection of tuples instead of scalar values.

We discuss six approaches in this category in Chapter 4. Note that, UDFs in this approach are not heavy weight and do not require extensive memory allocations and coding unlike in a purely UDF-based implementation [12].

3.4.3 Rule Generation

In the second phase of the association rule mining algorithm, we use the frequent itemsets to generate rules with the user specified minimum confidence, *minconf*. For every frequent itemset l, we first find all non-empty proper subsets of l. Then, for each of those subsets m, we find the confidence of the rule $m \rightarrow (l - m)$ and output the rule if it is at least *minconf*.

In the support counting phase, the frequent itemsets of size k are stored in table F_k. Before the rule generation phase, we merge all the frequent itemsets into a single table F.

The schema of F consists of $k+2$ attributes ($item_1, \ldots, item_k$, $support$, len), where k is the size of the largest frequent itemset and len is the length of the itemset as discussed earlier in Section 3.3.

We use the table function *GenRules* to generate all possible rules from a frequent itemset. The input argument to the function is a frequent itemset. For each itemset, it outputs tuples corresponding to rules with all non-empty proper subsets of the itemset in the consequent. The table function outputs tuples with $k+3$ attributes, $T\_item_1, \ldots, T\_item_k$, $T\_support, T\_len, T\_rulem$. The output is joined with F to find the support of the antecedent and the confidence of the rule is calculated by taking the ratio of the support values. The predicates in the where clause match the antecedent of the rule with the frequent itemset corresponding to the antecedent. While checking for this match, we need to check only up to $item_k$ where $k \leq T\_rulem$. The or part ($t_1.T\_rulem < k$) in the predicate accomplishes this. Figure 3.4 illustrates the rule generation query.

We can also do rule generation without using table functions and base it purely on SQL-92. The rules are generated in a level-wise manner where in each level k we generate rules with consequents of size k. Further, we make use of the property that for any frequent itemset, if a rule with consequent c holds, then, so do rules with consequents that are subsets of c as suggested in Agrawal et al. [9]. We can use this property to generate rules in level k using rules with $(k-1)$ long consequents found in the previous level, much like the way we did candidate generation in Section 3.4.1.

The fraction of the total running time spent in rule generation is very small. Therefore, we do not focus much on rule generation algorithms.

insert into R select $T\_item_1, \ldots T\_item_k$,

t_1.support, T_len, T_rulem, t_1.support/f_2.support

from $F\ f_1$, table(GenRules($f_1.item_1,\ldots,f_1.item_k$, f_1.len, f_1.support)) as t_1, $F\ f_2$

where $(t_1.T\_item_1 = f_2.item_1$ or $t_1.T\_rulem < 1)$ and

$\vdots$

$(t_1.T\_item_k = f_2.item_k$ or $t_1.T\_rulem < k)$ and

$t_1.T\_rulem = f_2.len$ and

$t_1.T\_support/f_2$.support > :minconf

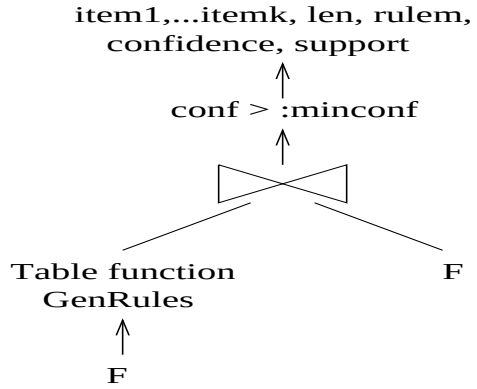


Figure 3.4. Rule generation

3.5 Support Counting Using SQL-92

We present four approaches in this category.

3.5.1 K-way Join

In each pass k, we join the candidate itemsets C_k with k transaction tables T and follow it up with a group by on the itemsets as shown in Figure 3.5.

Figure 3.5 also shows a tree diagram of the query. These tree diagrams are not to be confused with the plan trees which could look quite different.

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

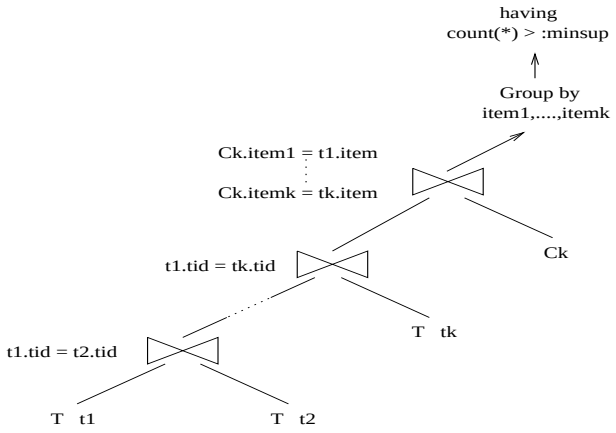


Figure 3.5. Support counting by K-way join

This SQL computation, when merged with the candidate generation step, is similar to the one proposed in Tsur et al. [128] as a possible mechanism to implement query flocks. In Section 3.7, we discuss the different execution plans for this query and the related performance issues.

3.5.2 Three-way Join

The above approach requires $(k+1)$-way joins in the kth pass. We can reduce the cardinality of joins to 3 using the following approach which bears some resemblance to

the AprioriTid algorithm in Agrawal et al. [9]. Each candidate itemset C_k, in addition to attributes $(item_1, \ldots, item_k)$ has three new attributes (oid, id_1, id_2). oid is a unique identifier associated with each itemset and id_1 and id_2 are oids of the two itemsets in F_{k-1} from which the itemset in C_k was generated (as discussed in Section 3.4.1). In addition, in the k^{th} pass we generate a new copy of the data table T_k with attributes (tid, oid) that keeps for each tid the oid of each itemset in C_k that it supported. For support counting, we first generate T_k from T_{k-1} and C_k and then do a group-by on T_k to find F_k as follows.

insert into T_k select t_1.tid, oid

from $C_k, T_{k-1}\ t_1, T_{k-1}\ t_2$

where t_1.oid = $C_k.id_1$ and t_2.oid = $C_k.id_2$ and t_1.tid = t_2.tid

insert into F_k select oid, $item_1, \ldots item_k$, cnt

from C_k,

(select oid as cid, count(*) as cnt from T_k

group by oid having count(*) > :minsup) as temp

where C_k.oid = cid

3.5.3 Two Group-by

Another way to avoid multi-way joins is to first join T and C_k based on whether the "item" of a (tid, item) pair of T is equal to any of the k items of C_k, then do a group by on $(item_1, \ldots, item_k, tid)$ filtering tuples with count equal to k. This gives all (itemset, tid) pairs such that the tid supports the itemset. Finally, as in the previous approaches, do a group-by on the itemset $(item_1, \ldots, item_k)$ filtering tuples that meet the support condition.

insert into F_k select $item_1, \ldots item_k$, count(*)

```
from        (select item_1, ... item_k, count(*)
            from   T, C_k
            where  item = C_k.item_1 or
                          ⋮
                   item = C_k.item_k
            group by item_1, ..., item_k, tid
            having count(*) = k) as temp
group by    item_1, ..., item_k
having      count(*) > :minsup
```

3.5.4 Subquery-Based

This approach makes use of common prefixes between the itemsets in C_k to reduce the amount of work done during support counting. We break up the support counting phase into a cascade of k subqueries. The l-th subquery Q_l finds all tids that match the distinct itemsets formed by the first l columns of C_k (call it d_l). The output of Q_l is joined with T and d_{l+1} (the distinct itemsets formed by the first $l+1$ columns of C_k) to get Q_{l+1}. The final output is obtained by doing a group-by on the k items to count support as above. The queries and the tree diagram for subquery Q_l are given in Figure 3.6.

3.6 Performance Comparison

In this section we compare the performance of the four SQL-92 approaches. All of our experiments were performed on Version 5 of IBM DB2 Universal Server installed on a RS/6000 Model 140 with a 200 MHz CPU, 256 MB main memory and a 9 GB disc with a measured transfer rate of 8 MB/s. These experimental results are also reported in Sarawagi et al. [109].

insert into F_k select $item_1, \ldots, item_k$, count(*)

from (`Subquery` Q_k) t

group by $item_1$,$item_2$ $\ldots item_k$

having count(*) > :minsup

`Subquery` Q_l (for any l between 1 and k):

select $item_1, \ldots item_l$, tid

from T t_l, (`Subquery` Q_{l-1}) as r_{l-1},

(select distinct $item_1 \ldots item_l$ from C_k) as d_l

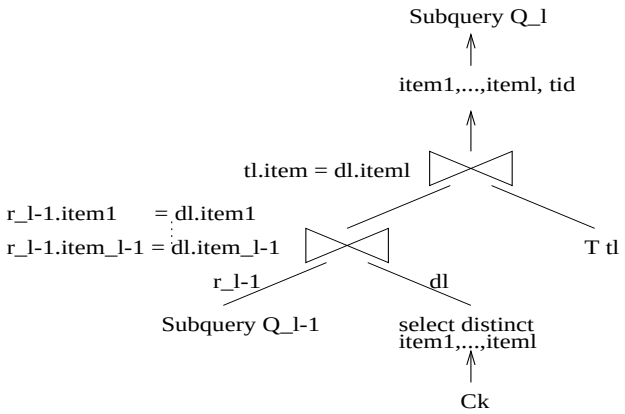
where $r_{l-1}.item_1 = d_l.item_1$ and ... and

$r_{l-1}.item_{l-1} = d_l.item_{l-1}$and

$r_{l-1}.tid = t_l.tid$ and

$t_l.item = d_l.item_l$

`Subquery` Q_0: No subquery Q_0.



Tree diagram for Subquery Q_l

Figure 3.6. Support counting using subqueries

Table 3.1. Description of different real-life datasets

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

We selected a collection of four real-life datasets obtained from various mail-order companies and retail stores for the experiments. These datasets have differing values of parameters like the total number of (tid,item) pairs, the number of transactions (tids), the number of items and the average number of items per transaction. Table 3.1 summarizes these parameters.

In this dissertation, we report the performance with only **Dataset-A**. The overall observation was that mining implementations in pure SQL-92 are too slow to be practical. For these experiments we built a composite index $(item_1 \ldots item_k)$ on C_k, k different indices on each of the k items of C_k and a $(tid, item)$ and a $(item, tid)$ index on the data table. The goal was to let the optimizer choose the best plan possible. We do not include the index building cost in the total time.

In Figure 3.7 we show the total time taken by the four approaches: `KwayJoin`, `3wayJoin`, `Subquery` and `2GroupBy`. For comparison, we also show the time taken by the `Loose-coupling` approach because this is the approach currently used by existing systems. The graph shows the total time split into candidate generation time (Cgen) and the time for each pass. The candidate generation time and the time for the first pass are much smaller compared to the total time. From these set of experiments we can make the following observations.

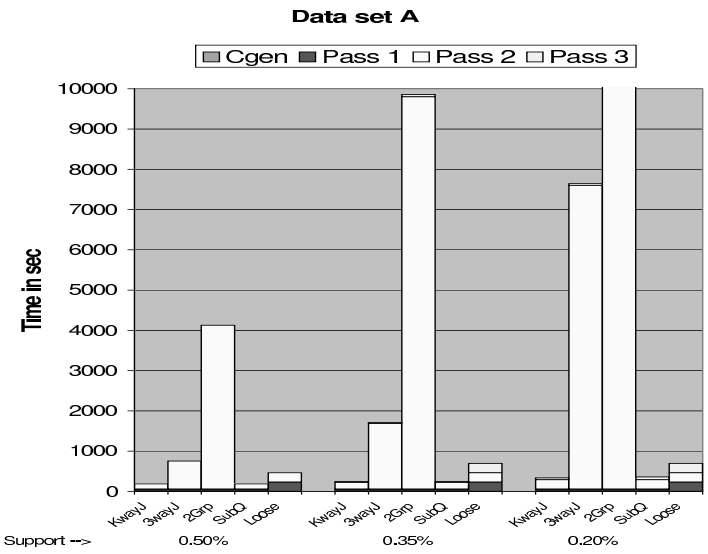


Figure 3.7. Comparison of different SQL-92 approaches

- The best approach in the SQL-92 category is the `Subquery` approach. An important reason for its performing better than the other approaches is exploitation of common prefixes between candidate itemsets. None of the other three approaches uses this optimization. Although the `Subquery` approach is comparable to the `Loose-coupling` approach in some cases, for other cases it did not complete even after taking ten times more time than the `Loose-coupling` approach.

- The `2GroupBy` approach is significantly worse than the other three approaches because it involves an index-ORing operation on k indices for each pass k of the algorithm. In addition, the inner group-by requires sorting a large intermediate relation. The outer group-by is comparatively faster because the sorted result is of size at most C_k which is much smaller than the result size of the inner group-by. The DBMS does aggregation *during* sorting therefore the size of the result is an important factor in the total cost.

- The `3wayJoin` approach is comparable to the `KwayJoin` approach for this dataset because the number of passes is at most three. As shown in Agrawal et al. [9] there might be other datasets especially ones where there is significant reduction in the size of T_k as k increases where `3wayJoin` might perform better than `KwayJoin`. However, one disadvantage of the `3wayJoin` approach is that it requires space to store and log the temporary relations T_k generated in each pass.

3.7 Cost Analysis

In this section, we analyze the cost of the `KwayJoin` and `Subquery` approaches which represent the better ones among the SQL-92 approaches. A relational query engine can execute the `KwayJoin` query in several different ways and the performance depends on the execution plan chosen. We experimented with a number of alternative execution plans for this query. We could force the query processor to choose different plans by creating different indices on T and C_k, and in some cases by disabling certain join methods in our experiments using Postgres. We will elaborate more on this in Section 3.9.

We present two different execution plans—one with C_k as the outermost relation in Section 3.7.1 and another with C_k as the innermost relation in Section 3.7.2—and the cost analysis for them. The effect of subquery optimization on the cost estimates is outlined in Section 3.7.3.

A schematic diagram of the two different execution plans are given in Figures 3.8 and 3.9. In the cost analysis, we use the mining-specific data parameters and knowledge about association rule mining (Apriori algorithm [9] in this case) to estimate the cost of joins and the size of join results. Even though current relational optimizers do not use this mining-specific semantic information, the analysis provides a basis for developing "mining-aware"

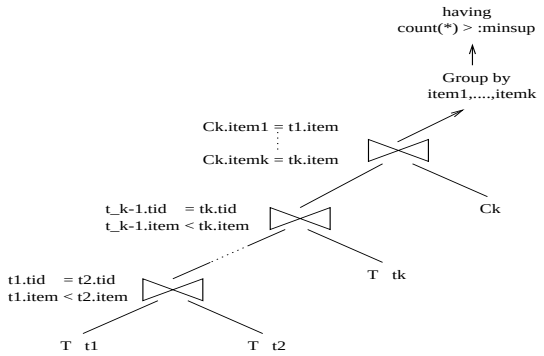


Figure 3.8. K-way join plan with C_k as inner relation

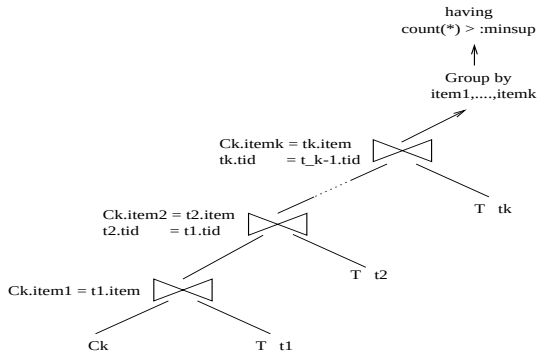


Figure 3.9. K-way join plan with C_k as outer relation

optimizers. The cost formulae are presented in terms of operator costs in order to make them general; for instance join(p, q, r) denotes the cost of joining two relations of size p and q to get a result of size r. The actual cost which is based on the join method used and the system parameters can be derived by substituting the appropriate values in the given formulae. The data parameters and operators used in the analysis are summarized in Table 3.2.

3.7.1 KWayJoin Plan with C_k as Outer Relation

Start with C_k as the outermost relation and perform a series of joins with the k copies of T. The final join result is grouped on the k items to find the support counts. The choice of join methods for each of the intermediate joins depends on factors such as the

Table 3.2. Notations used in cost analysis

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

availability of indices, the size of intermediate results, and the amount of available memory. For instance, the efficient execution of nested loops joins require an index (*item, tid*) on T. If the intermediate join result is large, it could be advantageous to materialize it and perform sort-merge join.

For each candidate itemset in C_k, the join with T produces as many records as the support of its first item. Therefore, the size of the join result can be estimated to be the product of the number of k-candidates and the average support of a frequent item and hence the cost of this join is given by join(C_k, R, $C_k * s_1$). Similarly, the relation obtained after joining C_k with l copies of T contain as many records as the sum of the support counts of the l-item prefixes of the candidate k-itemsets. Hence the cost of the l^{th} join is join($C_k * s_{l-1}$, R, $C_k * s_l$) where $s_0 = 1$. Note that values of the s_i's can be computed from statistics collected in the previous passes. Cost of the last join (with T_k) cannot be estimated using the above formula since the k-item prefix of a k-candidate is not frequent

and we do not know the value of s_k. However, the last join produces $S(C_k)$ records—there will be as many records for each candidate as its support—and therefore, the cost is join($C_k * s_{k-1}$, R, $S(C_k)$). $S(C_k)$ can be estimated by adding the support estimates of all the itemsets in C_k. A good estimate for the support of a candidate itemset is the minimum of the support counts of all its subsets. The overall cost of this plan expressed in terms of operator costs is

$$\{\sum_{l=1}^{k-1} \text{join}(C_k * s_{l-1},\ R,\ C_k * s_l)\} + \text{join}(C_k * s_{k-1},\ R,\ S(C_k)) + \text{group}(S(C_k),\ C_k)$$

3.7.2 KWayJoin Plan with C_k as Inner Relation

In this plan, we join the k copies of T and the resulting k-item combinations are joined with C_k to filter out non-candidate item combinations. The final join result is grouped on the k-items.

The result of joining l copies of T is the set of all possible l-item combinations of transactions and there are $C(N, l) * T$ such combinations. We know that the items in the candidate itemset are lexicographically ordered and hence we can add extra join predicates as shown in Figure 3.8 to limit the join result to l-item combinations (without these extra predicates the join will result in l-item permutations). When C_k is the outermost relation these predicates are not required. A mining-aware optimizer should be able to rewrite the query appropriately. The l^{th} join produces $(l+1)$-item combinations and therefore, its cost is join($C(N, l) * T$, R, $C(N, l+1) * T$). The last join produces $S(C_k)$ records as in the previous case and hence its cost is join($C(N, k) * T$, C_k, $S(C_k)$). The overall cost of this

plan is

$$\{\sum_{l=1}^{k-1} \text{join}(C(N,l)*T,\ R,\ C(N,l+1)*T)\} + \text{join}(C(N,k)*T,\ C_k,\ S(C_k)) + \text{group}(S(C_k),\ C_k)$$

Note that in the above expression $C(N,1) * T = R$.

3.7.3 Effect of Subquery Optimization

The subquery optimization makes use of common prefixes among candidate itemsets. Unfolding all the subqueries will result in a query tree which structurally resembles the `KwayJoin` plan tree shown in Figure 3.9. Subquery Q_l produces $d_k^l * s_l$ records where d_k^j denotes the number of distinct j item prefixes of itemsets in C_k. In contrast, the l^{th} join in the `KwayJoin` plan results in Typically d_k^l is much smaller compared to C_k which explains why the `Subquery` approach performs better than the `KwayJoin` approach. $C_k * s_l$ records. The output of subquery Q_k contains $S(C_k)$ records. The total cost of this approach can be estimated to be

$$\{\sum_{l=1}^{k} \text{trijoin}(R,\ s_{l-1} * d_k^{l-1},\ d_k^l,\ s_l * d_k^l)\} + \text{group}(S(C_k),\ C_k)$$

where $\text{trijoin}(p,q,r,s)$ denotes the cost of joining three relations of size p,q,r respectively producing a result of size s. The value of s_k which is the average support of a frequent k-itemset can be estimated as mentioned in section 3.7.1.

The experimental results presented in Section 3.6 and in Sarawagi et al. [110] shows that the subquery optimization gave much better performance than the basic `KwayJoin` approach (an order of magnitude better in some cases). We observed the same trend in our additional experiments using synthetic datasets. We used synthetic datasets for some of the

experiments because the real-life datasets were not available outside IBM. The synthetic datasets used in our experiments are detailed below. The main reason for the subquery approach performing better is that the number of distinct l-item prefixes is much less compared to the total number of candidate itemsets which results in joins between smaller tables. The number of candidate itemsets and the corresponding distinct item prefixes for various passes in one of our experiments is given in Figure 3.10. These numbers are for the dataset T10.I4.D100K and 0.33% support. Note that d_k^k is not shown since it is the same as C_k. In pass 3, C_3 contains 2453 itemsets where as d_3^1 has only 295 1-item prefixes (almost a factor of 10 less than C_3). This results in correspondingly smaller intermediate tables as shown in the analysis above, which is the key to the performance gain.

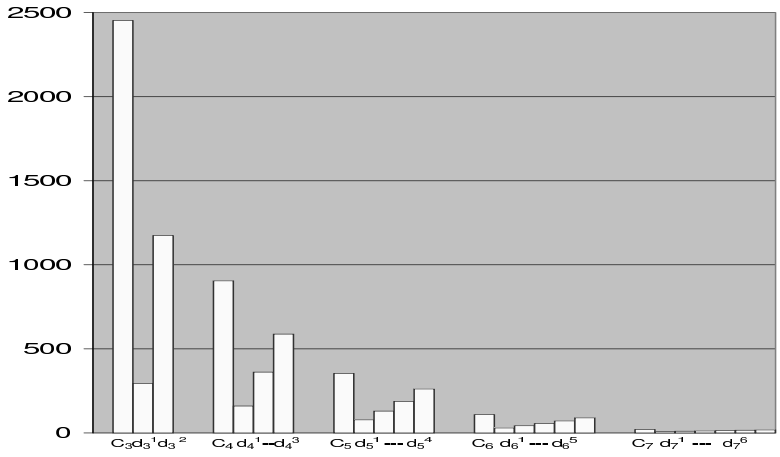


Figure 3.10. Number of candidate itemsets vs distinct item prefixes

Experimental datasets We used synthetic data generated according to the procedure explained in Agrawal and Srikant [13] for some of the experiments. The results reported here are for the datasets T5.I2.D100K and T10.I4.D100K. The first dataset consists of 100 thousand transactions, each containing an average of 5 items. The average size of the

Table 3.3. Description of synthetic datasets

Datasets	# Records	# Transactions	# Items	Avg.#items
T5.I2.D100K	546651	100000	1000	5
T10.I4.D100K	1154995	100000	1000	10

maximal potentially frequent itemsets (denoted as I) is 2. The transaction table corresponding to this dataset had approximately 550 thousand records. The second dataset has 100 thousand transactions, each containing an average of 10 items (total of about 1.1 million records) and the average size of maximal potentially frequent itemsets is 4. The different parameters of the two datasets are summarized in Table 3.3.

The experiments reported in the rest of this chapter were performed on PostgreSQL Version 6.3 [100], a public domain DBMS, installed on a 8 processor Sun Ultra Enterprise 4000/5000 with 248 MHz CPUs and 256 MB main memory per processor, running Solaris 2.6. Note that PostgreSQL is not parallelized. It supports nested loops, hash-based and sort-merge join methods and provides finer control of the optimizer to disable any of the join methods. We have found it to be a useful platform for studying the performance of different join methods and execution plans.

3.8 Performance Optimizations

The cost analysis presented above provides some insight into the different components of the execution time in the different passes and what can be optimized to achieve better performance. In this section, we present three optimizations to the `KwayJoin` approach (other than the subquery optimization) and discuss how they impact the cost. Based on these optimizations, we develop the Set-oriented Apriori approach in Section 3.8.4.

3.8.1 Pruning Non-Frequent Items

The size of the transaction table is a major factor in the cost of joins involving T. It can be reduced by pruning the non-frequent items from the transactions after the first pass. We store the transaction data as (tid, item) tuples in a relational table and hence this pruning means simply dropping the tuples corresponding to non-frequent items. This can be achieved by joining T with the frequent items table F_1 as follows.

```
insert into T_f select t.tid, t.item
from       T t, F_1 f
where      t.item = f.item
```

We insert the pruned transactions into table T_f which has the same schema as that of T. In the subsequent passes, joins with T can be replaced with corresponding joins with T_f. This could result in improved performance especially for higher support values where the frequent item selectivity is low, since we join smaller tables. For some of the synthetic datasets we used in our experiments, this pruning reduced the size of the transaction table to about half its original size. This could be even more useful for real-life datasets which typically contains lots of non-frequent items. For example, some of the real-life datasets used for the experiments reported in Sarawagi et al. [109] contained of the order of 100 thousand items out of which only a few hundreds were frequent. Figure 3.11 shows the reduction in transaction table size due to this optimization for our experimental datasets. The initial size (R) and the size after pruning (R_f) for different support values are shown.

With this optimization, in the cost formulae of section 3.7, R can be replaced with R_f—the number of records in T involving frequent items and N with N_f—the average

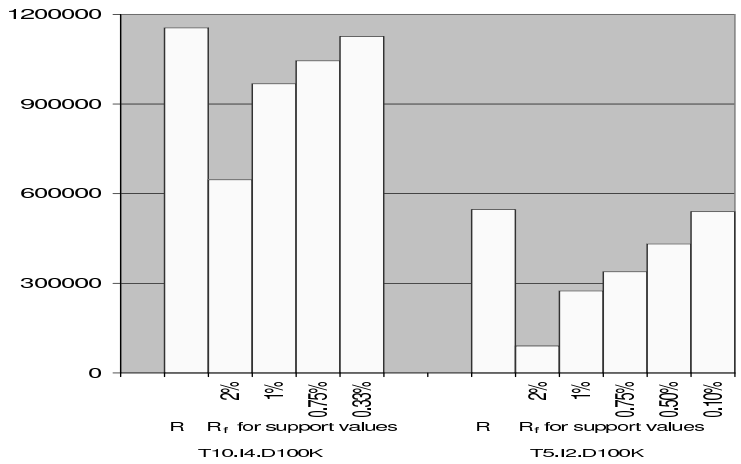


Figure 3.11. Reduction in transaction table size by non-frequent item pruning

number of frequent items per transaction. Note that with the subquery optimization we could achieve significant performance improvements by reducing the relation sizes of joins.

3.8.2 Eliminating Candidate Generation in Second Pass

This optimization aims at reducing the cost of the second pass which is a significant portion of the total cost in some cases. In the second pass, C_2 is almost a cartesian product of the two F_1s used to generate it and hence materializing it and joining with the T's (or T_f's) could be expensive. In order to circumvent the problem of counting the large C_2, most mining algorithms use special techniques in the second pass. A few examples are two-dimensional arrays in IBM's Quest data mining system [2] and hash filters proposed in Park et al. [99] to limit the size of C_2. The generation of C_2 can be completely eliminated by formulating the join query to find F_2 as

```
insert into F2 select p.item, q.item, count(*)
from     Tf p, Tf q
where    p.tid = q.tid and p.item < q.item
group by p.item, q.item
```

having count(*) > :minsup

The cost of second pass with this optimization is

$$\text{join}(R_f,\, R_f,\, C(N_f,\, 2)) + \text{group}(C(N_f,\, 2), C(F_1,\, 2))$$

Even though the grouping cost remains the same, there is a big reduction from the basic `KwayJoin` approach in the join costs. Figure 3.12 compares the running time of the second pass with this optimization to the basic `KwayJoin` approach for the two experimental datasets. For the `KwayJoin` approach, the best execution plan was the one which generates all 2-item combinations, joins them with the candidate set and groups the join result. We can see that this optimization has a significant impact on the running time.

3.8.3 Reusing the Item Combinations from Previous Pass

The SQL formulations of association rule mining is based on generating item combinations in various ways and similar work is performed in all the different passes. Therefore, reusing the item combinations will improve the performance especially in the higher passes. We will explain more about this optimization and the corresponding cost reduction in the next section.

3.8.4 Set-Oriented Apriori

In this section, we develop a set-oriented version of the apriori algorithm combining all the performance optimizations outlined above, which we call Set-oriented Apriori. We store the item combinations generated in each pass and use them in the subsequent passes instead of generating them in every pass from the transaction tables. In the k^{th} pass of the support counting phase, we generate a table T_k which contains all k-item combinations

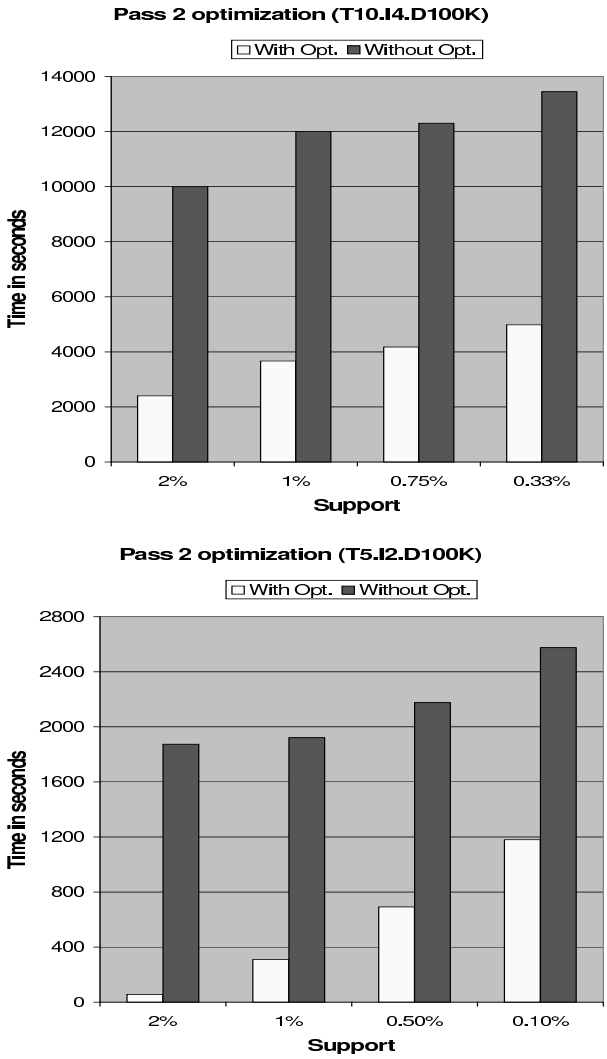


Figure 3.12. Benefit of second pass optimization

that are candidates. T_k has the schema $(tid, item_1, \ldots, item_k)$. We join T_{k-1}, T_f and C_k as shown below to generate T_k. A tree diagram of the query is also given in Figure 3.13. The frequent itemsets F_k is obtained by grouping the tuples of T_k on the k items and applying the minimum support filtering.

We can further prune T_k by filtering out item combinations that turned out to be non-frequent. However, this is not essential since we join it with the candidate set C_{k+1} in

insert into T_k

select p.tid, p.$item_1$, ... p.$item_{k-1}$, q.$item$

from C_k, T_{k-1} p, T_f q

where p.$item_1$ = C_k.$item_1$ and

⋮

p.$item_{k-1}$ = C_k.$item_{k-1}$ and

q.$item$ = C_k.$item_k$ and

p.tid = q.tid

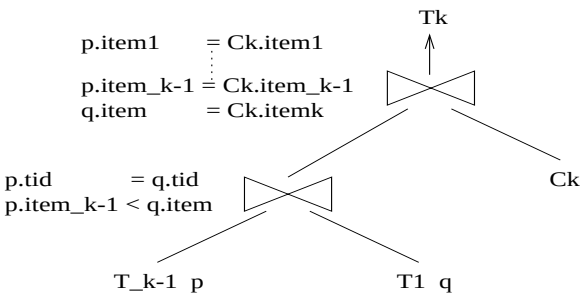


Figure 3.13. Generation of T_k

the next pass to generate T_{k+1}. The only advantage of pruning T_k is that we will have a smaller table to join in the next pass; but at the additional cost of joining T_k with F_k.

We use the optimization discussed above for the second pass and hence do not materialize and store the 2-item combinations T_2. Therefore, we generate T_3 directly by joining T_f with C_3 as

insert into T_3 select p.tid, p.$item$, q.$item$, r.$item$

from T_f p, T_f q, T_f r, C_k

where p.$item$ = C_3.$item_1$ and q.$item$ = C_3.$item_2$ and r.$item$ = C_3.$item_3$ and

p.tid = q.tid and q.tid = r.tid

We can also use the `Subquery` approach to generate T_3 if that is estimated to be less expensive. T_3 will contain exactly the same tuples produced by subquery Q_3.

The Set-oriented Apriori algorithm bears some resemblance with the three-way join approach in Section 3.5.2 and the AprioriTid algorithm in Agrawal and Srikant [13]. In the three-way join approach, the temporary table T_k stores for each transaction, the identifiers of the candidates it supported. T_k is generated by joining two copies of T_{k-1} with C_k. The generation of F_k requires a further join of T_k with C_k. AprioriTid makes use of special data structures which are difficult to maintain in the SQL formulation.

Cost Comparison

In Section 3.7.3, we saw that the cost of the k^{th} pass with the subquery optimization is

$$\{\sum_{l=1}^{k} \text{trijoin}(R,\ s_{l-1} * d_k^{l-1},\ d_k^l,\ s_l * d_k^l)\} + \text{group}(S(C_k),\ C_k)$$

As a result of the materialization and reuse of item combinations, Set-oriented Apriori requires only a single 3-way join in the k^{th} pass [1]. The cost of the k^{th} pass in Set-oriented Apriori is

$$\text{trijoin}(R_f,\ T_{k-1},\ C_k,\ S(C_k)) + \text{group}(S(C_k),\ C_k)$$

where T_{k-1} and C_k denote the cardinality of the corresponding tables. The grouping cost is the same as that of the subquery approach. The table T_{k-1} contains exactly the same tuples as that of subquery Q_{k-1} and hence has a size of $s_{l-1} * d_k^{l-1}$. Also, d_k^k is the same as C_k. Therefore, the k^{th} pass cost of Set-oriented Apriori is the same as the k^{th} term in

[1] Note that this may be executed as two 2-way joins since 3-way joins are not generally supported in current relational systems.

the join cost summation of the subquery approach. This results in significant performance improvements especially in the higher passes.

Figure 3.14 compares the running times of the subquery and Set-oriented Apriori approaches for the dataset T10.I4.D100K for 0.33% support. We show only the times for passes 3 and higher since both the approaches are the same in the first two passes.

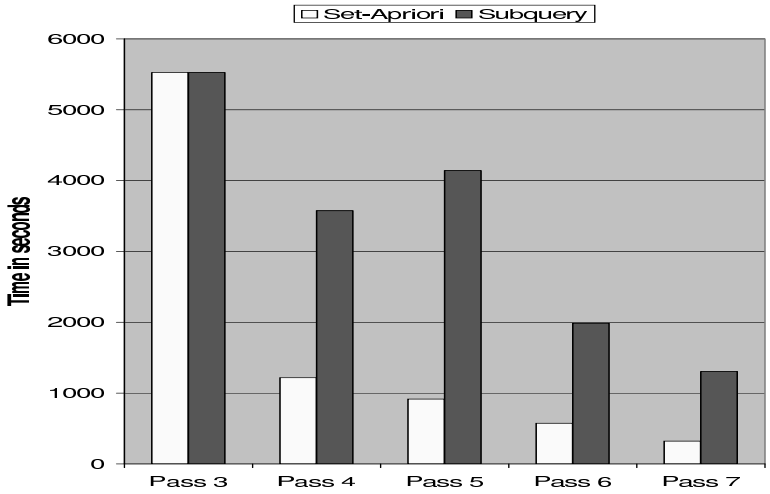


Figure 3.14. Benefit of reusing item combinations

Space Overhead

The Set-oriented Apriori approach requires additional space in order to store the item combinations generated. The size of the table T_k is the same as $S(C_k)$, which is the total support of all the k-item candidates. Assuming that the *tid* and *item* attributes are integers, each tuple in T_k consists of $k+1$ integer attributes. Figure 3.15 shows the space required to store T_k in terms of number of integers, for the dataset T10.I4.D100K for two different support values. The space needed for the input data table T is also shown for comparison. T_2 is not shown in the graph since we do not materialize and store it in the Set-oriented Apriori approach. Note that once T_k is materialized T_{k-1} can be deleted unless it needs to be retained for some other purposes.

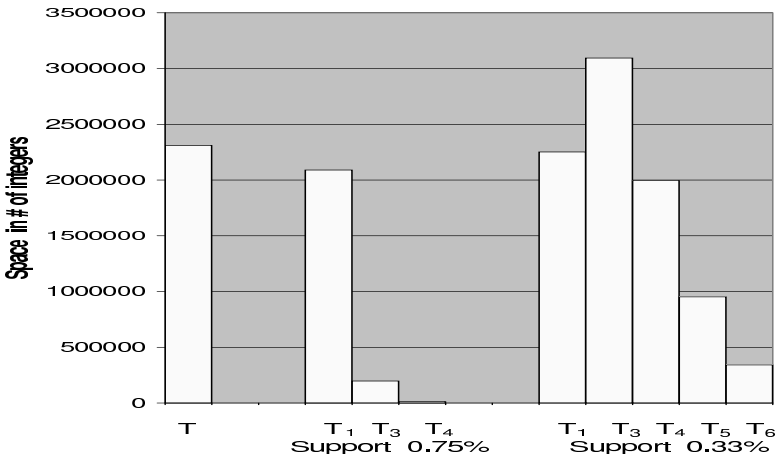


Figure 3.15. Space requirements of the set-oriented apriori approach

In the earlier passes, if the number of item combinations is too large and we do not want to store them, the `Subquery` approach can be used instead. However, the `Subquery` approach will also generate large intermediate tables except in cases where the output of an intermediate join is pipelined to the subsequent operation. The transition to Set-oriented Apriori can be made in any pass by materializing the item combinations. An advantage of the Set-oriented Apriori is that it requires only simple joins and hence it is easier for the optimizer to handle them. With multiple levels of nested subqueries, optimization becomes much harder.

3.9 Performance Experiments with Set-Oriented Apriori

In this section, we compare the total running time of the `Subquery` and Set-oriented Apriori approaches.

We studied the performance of the `Subquery` approach (the best SQL-92 approach in Section 3.6) and the Set-oriented Apriori for a wide range of data parameters and support values. We report the results on two of the datasets—T5.I2.D100K and T10.I4.D100K—which are described in Section 3.7.3.

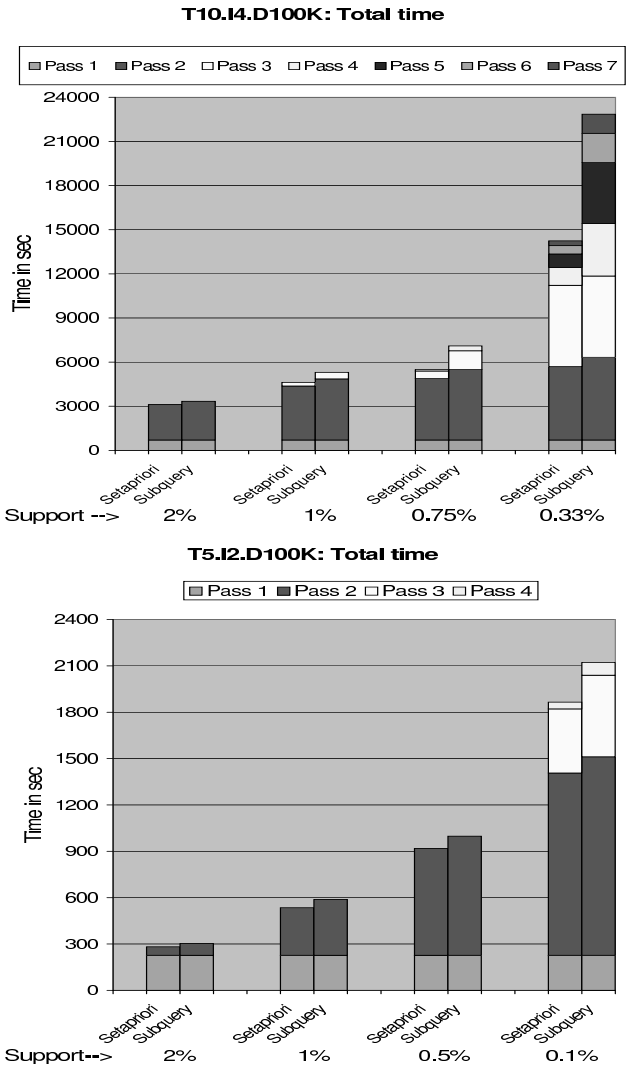


Figure 3.16. Comparison of `Subquery` and Set-oriented Apriori approaches

In Figure 3.16, we show the relative performance of `Subquery` and Set-oriented Apriori approaches for the two datasets. The chart shows the total time taken for each of the different passes. Set-oriented Apriori performs better than `Subquery` for all the support values. The first two passes of both the approaches are similar and they take approximately equal amount of time. The difference between Set-oriented Apriori and `Subquery` widens for higher numbered passes as explained in Section 3.8.4. For T5.I2.D100K, F_2 was empty

for support values higher than 0.3% and therefore we chose lower support values to study the relative performance in higher numbered passes.

In some cases, the optimizer did not choose the best plan. For example, for joins with T (T_f for Set-oriented Apriori), the optimizer chose nested loops plan using (item, tid) index on T in many cases where the corresponding sort-merge plan was faster; an order of magnitude faster in some cases. We were able to experiment with different plans by disabling certain join methods (disabling nested loops join for the above case). We also broke down the multi-way joins in some cases into simpler two-way joins to study the performance implications. The reported times correspond to the best join order and join methods.

Figure 3.17 shows the CPU time and I/O time taken for the dataset T10.I4.D100K. The two approaches show the same relative trend as the total time. However, it should be noted that the I/O time is less than one third of the CPU time. This shows that there is a need to revisit the traditional optimization and parallelization strategies designed to optimize for I/O time, in order to handle the newer decision support and mining queries efficiently.

3.9.1 Scale-up Experiment

We experimented with several synthetic datasets to study the scale-up behavior of Set-oriented Apriori with respect to increasing number of transactions and increasing transaction size. Figure 3.18 shows how Set-oriented Apriori scales up as the number of transactions is increased from 10,000 to 1 million. We used the datasets T5.I2 and T10.I4 for the average sizes of transactions and itemsets respectively. The minimum support level was kept at 1%. The first graph shows the absolute execution times and the second one shows the times

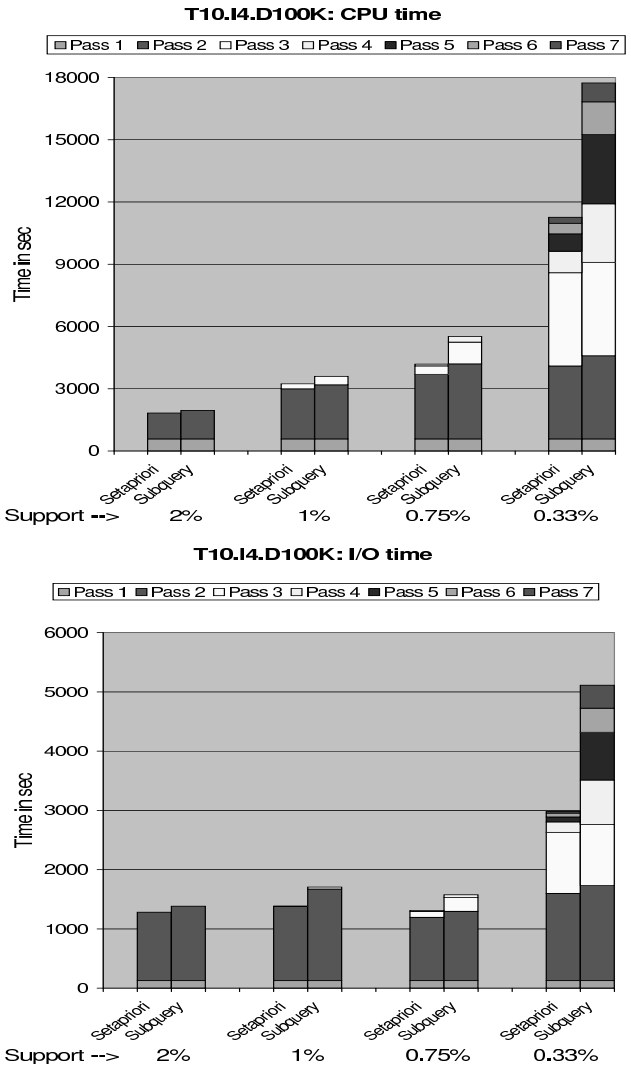


Figure 3.17. Comparison of CPU and I/O times

normalized with respect to the times for the 10,000 transaction datasets. It can be seen that the execution times scale quite linearly and both the datasets exhibit similar scale-up behavior.

The scale-up with increasing transaction size is shown in Figure 3.19. In these experiments we kept the physical size of the database roughly constant by keeping the product of the average transaction size and the number of transactions constant. The number of transactions ranged from 200,000 for the database with an average transaction size of 5 to

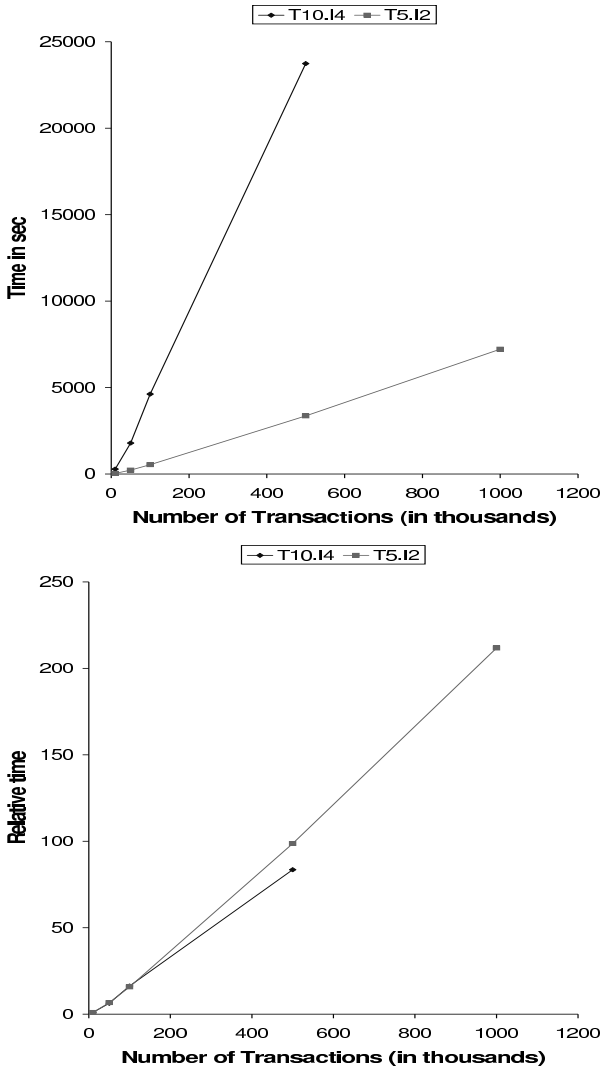


Figure 3.18. Number of transactions scale-up

20,000 for the database with an average transaction size of 50. We fixed the minimum support level in terms of the number of transactions, since fixing it as a percentage would have led to large increases in the number of frequent itemsets as the transaction size increased. The numbers in the legend (for example, 1000) refer to this minimum support. The execution times increase with the transaction size, but only gradually. The main reason for this increase was that the number of item combinations present in a transaction increases with the transaction size.

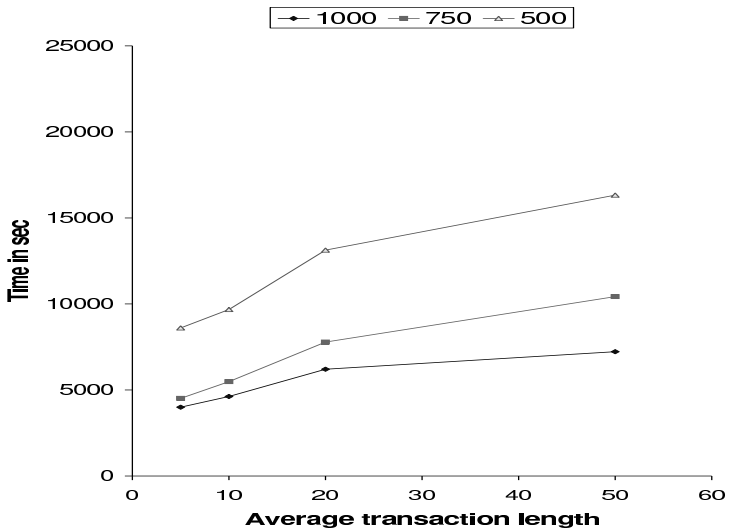


Figure 3.19. Transaction size scale-up

There has been increasing interest in developing scalable data mining techniques [23, 33, 117]. For the scalability of the SQL approaches, we leverage the development effort spent in making the relational query processors scalable.

3.10 Summary

In this chapter, we presented four SQL-92 formulations for association rule mining. We analyzed the best approach and developed cost formulae for the different execution plans of the corresponding SQL queries based on the input data parameters and the relational operator costs. This cost analysis provides a basis for incorporating the semantics of mining algorithms into future query optimizers.

While doing the experiments, it was difficult to force the optimizer to choose certain execution plans and join methods since commercial DBMSs do not provide that level of control. Postgres was relatively better in that respect, since we could control the choice of join methods. However, the Postgres optimizer did not optimize long queries, especially the ones involving nested subqueries, well.

Note A part of the work described in this chapter was primarily done by researchers from IBM Almaden Research Center. Specifically, the SQL-based candidate generation in Section 3.4.1 and the support counting approaches in Section 3.5 were developed by them. They are included in this dissertation for completeness.

CHAPTER 4
SUPPORT COUNTING USING SQL WITH OBJECT-RELATIONAL EXTENSIONS

In this chapter, we study alternative approaches that make use of additional object-relational features in SQL. For each approach, we also outline a cost-based analysis of the execution time to enable one to choose between these different approaches. We present six different approaches, optimizations and their cost estimates in Sections 4.1, 4.2 and 4.3. Experimental results comparing the performance of these approaches are presented in Section 4.4. In Section 4.5, we propose a hybrid approach which combines the best of all approaches. The performance of different architectural alternatives described in Chapter 2 is compared in Section 4.6. In Section 4.7, we summarize qualitative comparisons of these architectures. The applicability of the SQL-based approach to other association rule mining algorithms are briefly discussed in Section 4.8.

4.1 GatherJoin

This approach (see Figure 4.1) is based on the use of table functions described in section 3.4.2. It generates all possible k-item combinations of items contained in a transaction, joins them with the candidate table C_k and counts the support of the itemsets by grouping the join result. It uses two table functions `Gather` and `Comb-K`. The data table T is scanned in the *(tid, item)* order and passed to the table function `Gather`. This table function collects all the items of a transaction (in other words, items of all tuples of T with the same *tid*) in memory and outputs a record for each transaction. Each such record consists of two

attributes, the *tid* and *item-list* which is a collection of all its items in a VARCHAR or a BLOB. The output of `Gather` is passed to another table function `Comb-K` which returns all k-item combinations formed out of the items of a transaction. A record output by `Comb-K` has k attributes $T\_itm_1, \ldots, T\_itm_k$, which can be directly used to probe into the C_k table. An index is constructed on all the items of C_k to make the probe efficient. Figure 4.1 presents SQL queries for this approach.

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

4.1.1 Special Pass 2 Optimization

Note that for $k = 2$, the 2-candidate set C_2 is simply a join of F_1 with itself. Therefore, we can specially optimize the pass 2 by replacing the join with C_2 by a join with F_1 *before* the table function (see Figure 4.2). That way, the table function gets only frequent items and generates significantly fewer 2-item combinations. This optimization can be useful for other passes too but unlike for pass 2 we still have to do the join with C_k.

4.1.2 Variations of GatherJoin Approach

GatherCount

One variation of the `GatherJoin` approach for pass two is the **GatherCount** approach where we push the group-by inside the table function instead of doing it outside. The candidate 2-itemsets (C_2) are represented as a two dimensional array inside the modified

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

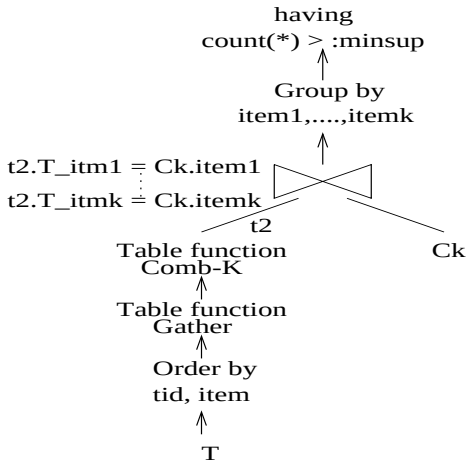


Figure 4.1. Support counting by `GatherJoin`

table function `Gather-Cnt` for doing the support counting. Instead of outputting the 2-item combinations of a transaction, it directly uses it to update support counts in the memory and output only the frequent 2-itemsets, F_2 and their support after the last transaction. Thus, the table function `Gather-Cnt` is an extension of the `GatherComb-2` table function used in `GatherJoin`.

The absence of the outer grouping makes this option rather attractive. The UDF code is also small since it only needs to maintain a 2D array. We could apply the same trick for subsequent passes but the coding becomes considerably more complicated because of the

insert into F_2 select $tt_2.T\_itm_1$, $tt_2.T\_itm_2$, count(*)
from (select * from T, F_1 where $T.item = F_1.item_1$) as tt_1,
table (GatherComb-2(tid,item)) as tt_2)
group by $tt_2.T\_itm_1$, $tt_2.T\_itm_2$
having count(*) > :minsup

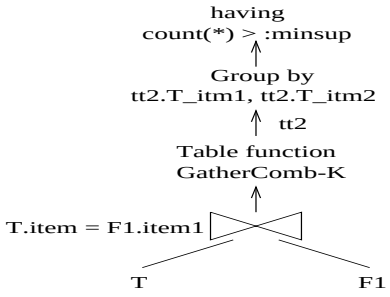


Figure 4.2. Support Counting by `GatherJoin` in the second pass

need to maintain hash-tables to index the C_k's. The disadvantage of this approach is that it can require a large amount of memory to store C_2. If enough memory is not available, C_2 needs to be partitioned and the process has to be repeated for each partition. Another serious problem with this approach is that it cannot be automatically parallelized unlike the other approaches.

GatherPrune

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

The cost of passing the BLOB for every tuple of T could be high. In general, we can reduce the parameter passing cost by using a smaller Blob that only approximates the real

C_k. The trade-off is increased cost for other parts notably grouping because not as many combinations are filtered.

Horizontal

Another variation of `GatherJoin` is the `Horizontal` approach that first uses the `Gather` function to transform the data to the horizontal format but is otherwise similar to the `GatherJoin` approach.

Rajamani et al. [105] propose finding associations using an approach quite similar to this `Horizontal` approach. They assume (rather unrealistically) that the data is already in a horizontal format. However, they do not use the frequent item-set filtering optimization we outlined for pass 2. Without this optimization, the time for pass 2 for most real-life datasets blows up even for relatively high support values. Also, at the time of candidate generation, rather than doing self-join on F_{k-1}, they join F_{k-1} with F_1, thereby generating considerably more combinations than needed. Thus, the approach in Rajamani et al. [105] is likely to perform worse than `Horizontal`.

4.1.3 Cost Analysis of GatherJoin and its Variants

The choice of the best approach depends on a number of data characteristics like the number of items, total number of transactions, average length of a transaction and so on. We express the costs of different approaches in each pass in terms of parameters that are known or can be estimated after the candidate generation step of each pass. We include only the terms that were found to be the dominant part of the total cost in practice. We use the notations of Table 3.2 in the cost analysis.

The cost of `GatherJoin` includes the cost of generating k-item combinations, joining with C_k and grouping to count the support. The number of k-item combinations generated,

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

$$T_k * t_k + \text{join}(T_k, C_k, S(C_k)) + \text{group}(S(C_k), C_k), \text{ where } T_k = C(N,k) * T$$

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

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

$$\text{join}(R, F_1, R_f) + T_2 * t_2 + \text{group}(T_2, C_2), \text{ where } T_2 = C(N_f, 2) * T \approx \frac{N_f^2 * T}{2}$$

Cost of `GatherCount` in the second pass is similar to that for basic `GatherJoin` except for the final grouping cost. In this formula, "group_int" denotes the cost of doing the support counting inside the table function.

$$\text{join}(R, F_1, R_f) + \text{group\_int}(T_2, C_2) + F_2 * t_2$$

For `GatherPrune` the cost equation is

$$R * \text{blob}(k * C_k) + S(C_k) * t_k + \text{group}(S(C_k), C_k).$$

We use blob($k * C_k$) for the BLOB passing cost since each itemset in C_k contains k items.

The cost estimate of `Horizontal` is similar to that of **GatherJoin** except that here the data is materialized in the horizontal format before generating the item combinations.

4.2 Vertical

In this approach, we first transform the data table into a vertical format by creating for each item a BLOB containing all tids that contain that item (Tid-list creation phase) and then count the support of itemsets by merging together these tid-lists (support counting phase). This approach is related to the approaches discussed in Savasere et al. [111] and Zaki et al. [131].

For creating the Tid-lists we use a table function `Gather`. This is the same as the `Gather` function in **GatherJoin** except that here we create the tid-list for each frequent item. The data table T is scanned in the (item, tid) order and passed to the function `Gather`. The function collects the *tids* of all tuples of T with the same item in memory and outputs a (item, tid-list) tuple for items that meet the minimum support criterion. The tid-lists are represented as BLOBs and stored in a new TidTable with attributes (item, tid-list). The SQL query which does the transformation to vertical format is given in Figure 4.3.

```
insert into TidTable select item, tid-list from
(select * from T order by item, tid) as tt1,
    table(Gather(item,tid,:minsup)) as tt2
```

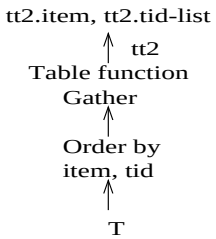


Figure 4.3. Tid-list creation

In the support counting phase, conceptually for each itemset in C_k we want to collect the tid-lists of all k items and use a UDF to count the number of *tids* in the intersection

of these k lists. The tids are in the same sorted order in all the tid-lists and therefore the intersection can be done easily and efficiently by a single pass of the k lists. This conceptual step can be improved further by decomposing the intersect operation so that we can share these operations across itemsets having common prefixes as follows:

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

The above sequence of operations can be written as a single SQL query for any k as shown in Figure 4.4. The final intersect operation can be merged with the count operation to return a count instead of the tid-list. We do not include this optimization in the query of Figure 4.4 for simplicity.

4.2.1 Special Pass 2 Optimization

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

```
insert into F_k select t1.item, t2.item, cnt
from     (select item1, item2, CountIntersect(t1.tid-list, t2.tid-list) as cnt
          from   TidTable t1, TidTable t2
          where  t1.item < t2.item) as t
where    cnt > :minsup
```

insert into F_k select $item_1, \ldots, item_k$, count(tid-list) as cnt

from (`Subquery` Q_k) t where cnt > :minsup

`Subquery` Q_l (for any l between 2 and k)

select $item_1, \ldots item_l$, Intersect(r_{l-1}.tid-list,t_l.tid-list) as tid-list

from TidTable t_l, (`Subquery` Q_{l-1}) as r_{l-1},

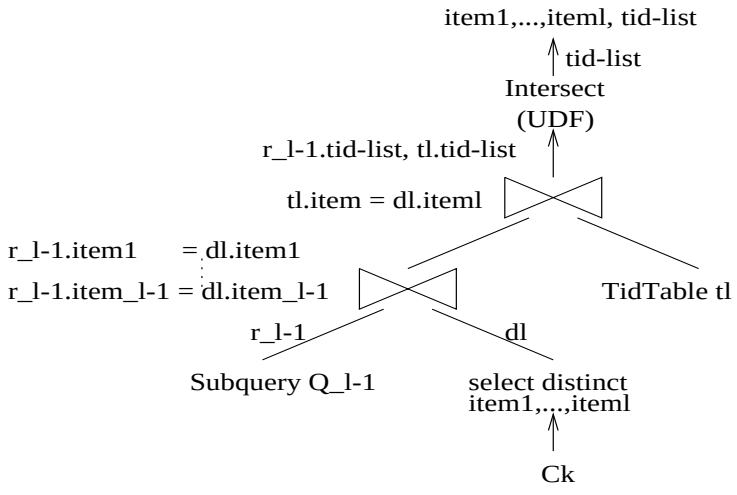
(select distinct $item_1 \ldots item_l$ from C_k) as d_l

where $r_{l-1}.item_1 = d_l.item_1$ and ... and

$r_{l-1}.item_{l-1} = d_l.item_{l-1}$and

$t_l.item = d_l.item_l$

`Subquery` Q_1: (select * from TidTable)



Tree diagram for Subquery Q_l

Figure 4.4. Support counting using UDF

4.2.2 Cost Analysis

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

In addition to the intersect cost it includes the cost of joins in the query also. The join cost of subquery Q_l can be recursively defined as

$$C(Q_l) = \text{trijoin}(F_1, Q_{l-1}, C_k, d_l^k) + C(Q_{l-1})$$

where tri-join(p, q, r, s) denotes the cost of joining three relations of size p, q, r respectively producing a result of size s. The exact cost of the 3-way join will depend on the join order. The cost of subquery Q_1 is the cost of scanning the TidTable which has F_1 tuples. The result size of the subquery Q_l is d_l^k and the result size of Q_1 is F_1. The total cost of the `Vertical` approach is

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

In the formula above Intersect(n) denotes the cost of intersecting two tid-lists with a combined size of n. The total cost is dominated by the intersect cost and join costs account for only a small fraction. Therefore, we can safely ignore the join costs in the above formulae.

The total cost of the second pass is

$$\text{join}(F_1, F_1, \frac{(F_1)^2}{2}) + C_2 * \{2 * Blob(\frac{R_f}{F_1}) + Intersect(\frac{2R_f}{F_1})\}$$

4.3 SQL-Bodied Functions

This approach is based on SQL-bodied functions commonly known as SQL/PSM [87]. SQL/PSMs extend SQL with additional control structures. We make use of one such construct for do .. end.

We use the for construct to scan the transaction table T in the (tid, item) order. Then, for each tuple (tid, item) of T, we update those tuples of C_k that contain one matching item. C_k is extended with 3 extra attributes (prevTid, match, supp). The *prevTid* attribute keeps the *tid* of the previous tuple of T that matched that itemset. The *match* attribute contains the number of items of *prevTid* matched so far and *supp* holds the current support of that itemset. On each column of C_k an index is built to do a searched update.

```
for this as select * from T do
    update C_k set prevTid = tid,
            match = case when tid = prevTid then match+1 else 1 end,
            supp = case when match = k-1 and tid = prevTid then supp+1
                    else supp
    end
```

where item = $item_1$ or

$\vdots$

item = $item_k$

end for

insert into F_k select $item_1, \ldots, item_k$, supp

from C_k where supp > :minsupp

The cost of this approach can be mainly attributed to the cost of updates to the candidate table C_k. For each tuple of the data table T, for all the candidate itemsets in C_k which contains that item, three updates are performed (the attributes prevTid, match, supp of the itemset are updated). If N_k is the average number of k-item candidates containing any given item, the total number of updates is $3 * R * N_k$. The cost due to updates for this approach is $U(3 * R * N_k)$ where $U(n)$ is the cost of n updates. If the updates are logged this cost includes the logging cost also.

4.4 Performance Comparison

We studied the performance of six approaches in this category: `GatherJoin` and its variants `GatherPrune`, `Horizontal` and `GatherCountVertical` and `SBF`. We used the four datasets summarized in Table 3.1. In Figure 4.5 we show the performance of only the four approaches: **GatherJoin**, **GatherCount**, **GatherPrune** and **Vertical**. For the other two approaches the running times were comparatively so large that we had to abort the runs in many cases. The main reason why the `Horizontal` approach was significantly worse than the `GatherJoin` approach was the time to transform the data to the horizontal format. For instance, for **Dataset-C** it was 3.5 hours which is almost 20 times more than the time taken by `Vertical` for 2% support. For **Dataset-B** the process was aborted

after running for 5 hours. After the transformation, compared to `GatherJoin` the time taken by `Horizontal` was also significantly worse when run without the frequent itemset filtering optimization but with the optimization the performance was comparable. The `SBF` approach had significantly worse performance because of the expensive indexing ORing of the k join predicates. Another problem with this approach is the large number of updates to the C_k table. In DB2, all of these updates are logged resulting in severe performance degradation.

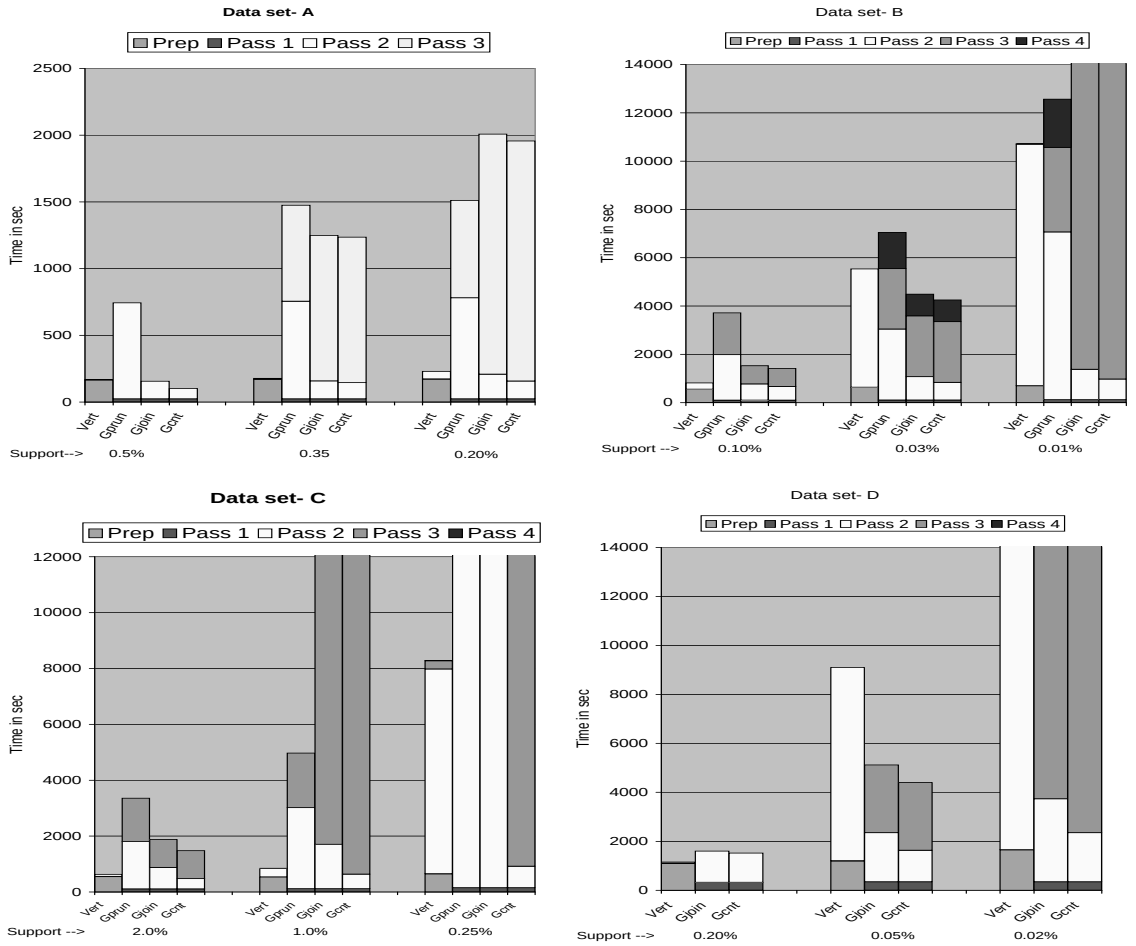


Figure 4.5. Comparison of four SQL-OR approaches: `Vertical`, `GatherPrune`, `GatherJoin` and `GatherCount`

Figure 4.5 shows the total running time of the different approaches. The time taken is broken down by each pass and an initial "prep" stage where any one-time data transformation cost is included. We can make several observations from the experimental results. First, let us concentrate on the overall comparison between the different approaches. Then we will compare the approaches based on how they perform in each pass of the algorithm.

Overall, the `Vertical` approach has the best performance and is sometimes more than an order of magnitude better than the other three approaches.

The majority of the time of the `Vertical` approach is spent in transforming the data to the `Vertical` format in most cases (shown as "prep" in figure 4.5). The vertical representation is like an index on the *item* attribute. If we think of this time as a one-time activity like index building then performance looks even better. Note that the time to transform the data to the `Vertical` format was much smaller than the time for the horizontal format although both formats write almost the same amount of data. The main reason was the difference in the *number* of records written. The number of frequent items is often two to three orders of magnitude smaller than the number of transactions.

Between `GatherJoin` and `GatherPrune`, neither strictly dominates the other. The special optimization in `GatherJoin` of pruning based on F_1 had a big impact on performance. With this optimization, for **Dataset-B** with support 0.1%, the running time for pass 2 alone was reduced from 5.2 hours to 10 minutes.

When we compare these different approaches based on time spent in each pass we observe that no single approach is "the best" for all different passes of the different datasets especially for the second pass.

For pass three onwards, `Vertical` is often two or more orders of magnitude better than the other approaches. Even in cases like **Dataset-B**, support 0.01% where it spends three

hours in the second pass, the total time for next two passes is only 14 seconds whereas it is more than an hour for the other two approaches. For subsequent passes the performance degrades dramatically for `GatherJoin`, because the table function `Gather-Comb-K` generates a large number of combinations. For instance, for pass 3 of **Dataset-C** even for support value of 2% pass 3 did not complete after 5.2 hours whereas for `Vertical` pass 3 finished in 0.2 seconds. `GatherPrune` is better than `GatherJoin` for the third and later passes. For pass 2 `GatherPrune` is worse because the overhead of passing a large object as an argument dominates cost.

The `Vertical` approach sometimes ended up spending too much time in the second pass. In some of these cases the `GatherJoin` approach was better in the second pass (for instance for low support values of **Dataset-B**) whereas in other cases (for instance, **Dataset-C** support 0.25%) `GatherCount` was the only good option. For this case both the `GatherPrune` and `GatherJoin` did not complete after more than six hours even for pass 2. Further, they caused a storage overflow error because of the large size of the intermediate results to be sorted. We had to divide the dataset into four equal parts and run the second pass independently on each partition to avoid this problem.

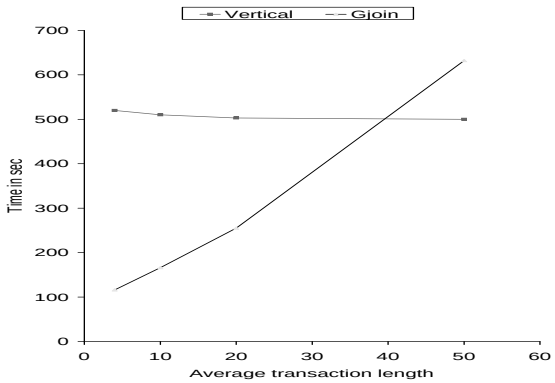


Figure 4.6. Effect of increasing transaction length (average number of items per transaction)

Two factors that affect the choice amongst the `Vertical`, `GatherJoin` and `GatherCount` approaches in different passes and pass 2 in particular are: number of frequent items (F_1) and the average number of frequent items per transaction (N_f). From the graphs in Figure 4.5 we notice that as the value of the support is decreased for each dataset causing the size of F_1 to increase, the performance of pass 2 of the `Vertical` approach degrades rapidly. This trend is also clear from our cost formulae. The cost of the `Vertical` approach increases quadratically with F_1. `GatherJoin` depends more critically on the number of frequent items per transaction. For **Dataset-B** even when the size of F_1 increases by a factor of 10, the value of N_f remains close to 2, therefore the time taken by `GatherJoin` does not increase as much. However, for **Dataset-C** the size of N_f increases from 3.2 to 10 as the support is decreased from 2.0% to 0.25% causing `GatherJoin` to deteriorate rapidly. From the cost formula for `GatherJoin` we notice that the total time for pass 2 increases almost quadratically with N_f.

We validate this observation further by running experiments on synthetic datasets for varying values of the number of frequent items per transaction. We used the synthetic dataset generator described in Agrawal et al. [9] for this purpose. We varied the transaction length, the number of transactions and the support values while keeping the total number of records and the number of frequent items fixed. In Figure 4.6 we show the total time spent in pass 2 of the `Vertical` and `GatherJoin` approaches. As the number of items per transaction (transaction length) increases, the cost of `Vertical` remains almost unchanged whereas the cost of `GatherJoin` increases.

4.5 Final Hybrid Approach

The previous performance section helps us draw the following conclusion: Overall, the `Vertical` approach is the best option especially for higher passes. When the size of the candidate itemsets is too large, the performance of the `Vertical` approach could suffer. In such cases, `GatherJoin` is a good option as long as the number of frequent items per transaction (N_f) is not too large. When that happens `GatherCount` may be the only good option even though it may not easily parallelizable. These qualitative differences are captured by the cost formulae we presented earlier and are used by our final hybrid scheme.

The hybrid scheme chooses the best of the three approaches `GatherJoin`, `GatherCount` and `Vertical` for each pass based on the cost estimation outlined in the previous sections. The parameter values used for the estimation are all available at the end of the previous pass. In Section 4.6 we plot the final running time for the different datasets based on this hybrid approach.

4.6 Architecture Comparisons

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

For `Loose-coupling`, we use the implementation of the Apriori algorithm [9] for finding association rules provided with the IBM data mining product, Intelligent Miner [71]. For `Stored-procedure`, we extracted the Apriori implementation in Intelligent Miner and created a stored procedure out of it. The stored procedure is run in the unfenced mode in the database address space. For `Cache-Mine`, we used an option provided in Intelligent Miner that causes the input data to be cached as a binary file after the first scan of the data

from the DBMS. The data is copied in the horizontal format where each tid is followed by an encoding of all its frequent items.

For the `UDF`-architecture, we use the UDF implementation of the Apriori algorithm described in Agrawal and Shim [12]. In this implementation, first a UDF is used to initialize state and allocate memory for candidate itemsets. Next, for each pass a collection of UDFs are used for generating candidates, counting support, and checking for termination. These UDFs access the initially allocated memory, address of which is passed around in BLOBs. Candidate generation creates the in-memory hash-trees of candidates. This happens entirely in the UDF without any involvement of the DBMS. During support counting, the data table is scanned sequentially and for each tuple a UDF is used for updating the counts on the memory resident hash-tree.

4.6.1 Timing Comparison

In Figure 4.7, we show the performance of the four architectural alternatives: `Cache-Mine`, `Stored-procedure`, `UDF` and our best SQL implementation for the datasets in Table 3.1. We do not show the times for the `Loose-coupling` option because its performance was very close to the `Stored-procedure` option. For each dataset three different support values are used. The total time is broken down by the time spent in each pass.

We can make the following observations.

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

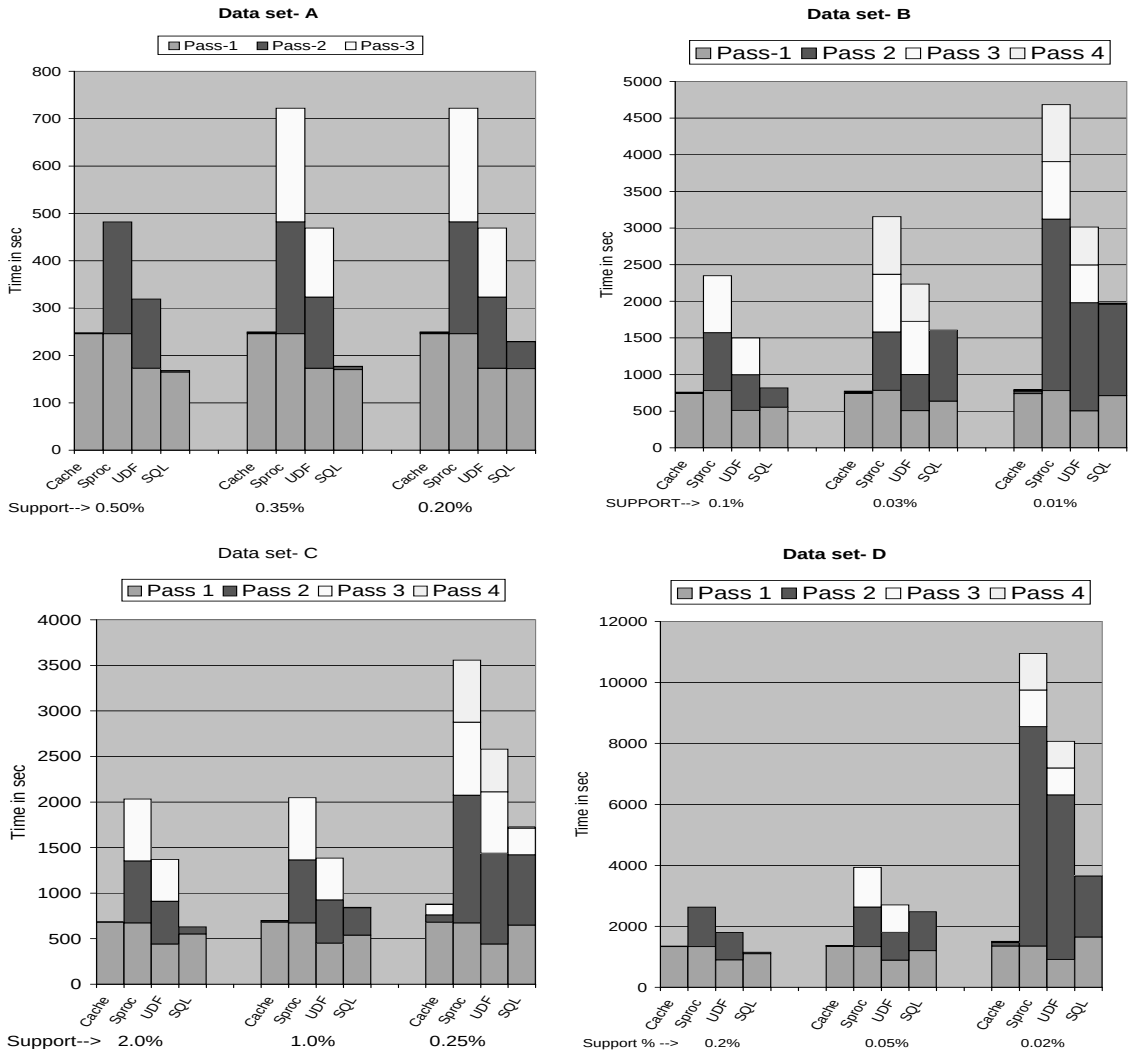


Figure 4.7. Comparison of four architectures

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

Dataset-C and **Dataset-D**. Time taken by `Stored-procedure` can be expressed approximately as number of passes *times* time taken by `Cache-Mine`.

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

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

  The `SQL` approach is 1.8 to 3 times better than `Stored-procedure` or `Loose-coupling` approach. As we decreased the support value so that the number of passes over the dataset increases, the gap widens. Note that we could have implemented a Stored-procedure using the same hybrid algorithm that we used for `SQL` instead of using the IM algorithm. Then, we expect the performance of `Stored-procedure` to improve because the number of passes to the data will decrease. However, we will pay the storage penalty of making additional copy of the data as we did in the `Cache-Mine` approach. The performance of `Stored-procedure` cannot be better than `Cache-Mine`

because as we have observed the most of the time of `Cache-Mine` is spent in the first pass which cannot be changed for `Stored-procedure`.

4.6.2 Scale-up experiment

Our experiments with the four real-life datasets above has shown the scaling property of the different approaches with decreasing support value and increasing number of frequent itemsets. We experiment with synthetic datasets to study other forms of scaling: increasing number of transactions and increasing average length of transactions. In Figure 4.8 we show how `Stored-procedure`, `Cache-Mine` and `SQL` scale with increasing number of transactions. `UDF` and `Loose-coupling` have similar scale-up behavior as `Stored-procedure`, therefore we do not show these approaches in the figure. We used a dataset with 10 average number of items per transaction, 100 thousand total items and a default pattern length (defined in [9]) of 4. Thus, the size of the dataset is 10 times the number of transactions. As the number of transactions is increased from 10K to 3000K the time taken increases proportionately. The largest frequent itemset was 5 long. This explains the five fold difference in performance between the `Stored-procedure` and the `Cache-Mine` approach. Figure 4.9 shows the scaling when the transaction length changes from 3 to 50 while keeping the number of transactions fixed at 100K. All three approaches scale linearly with increasing transaction length.

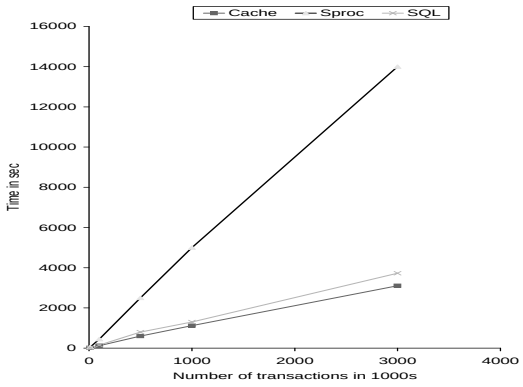


Figure 4.8. Scale-up with increasing number of transactions

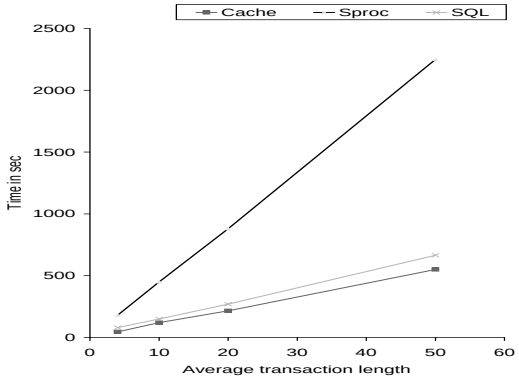


Figure 4.9. Scale-up with increasing transaction length

4.6.3 Impact of longer names

In these experiments we assumed that the tids and item-ids are all integers. Often in practice these are character strings longer than four characters. Longer strings need more storage and cost more during comparisons. This could hurt all four of the alternatives. For the `Stored-procedure`, `UDF` and `Cache-Mine` approach the time taken to transfer data will increase. The Intelligent Miner code [71] maps all character fields to integers using an in-memory hash-table. Therefore, beyond the increase in the data transfer and mapping costs (which accounts for the bulk of the time), we do not expect the processing time of these three alternatives to increase. For the `SQL` approach we cannot assume an in-memory hash-table for doing the mapping therefore we use an alternative approach based on table functions.

For `SQL` approach we discuss the hybrid approach. The two (already expensive) steps that could suffer because of longer names are (1) final group-bys during pass 2 or higher when the `GatherJoin` approach is chosen and (2) tid-list operations when the `Vertical` approach is chosen. For efficient performance, the first step requires a mapping of item-ids and the second one requires us to map tids. We use a table function to map the tids to

unique integers efficiently in one pass and without making extra copies. The input to the table function is the data table in the *tid* order. The table function remembers the previous *tid* and the maintains a counter. Every time the *tid* changes, the counter is incremented. This counter value is the mapping assigned to each *tid*. We need to do the *tid* mapping only once before creating the TidTable in the `Vertical` approach and therefore we can pipeline these two steps. The *item* mapping is done slightly differently. After the first pass, we add a column to F_1 containing a unique integer for each item. We do the same for the TidTable. The `GatherJoin` approach already joins the data table T with F_1 before passing to table function `Gather`. Therefore, we can pass to `Gather` the integer mappings of each item from F_1 instead of its original character representation. After these two transformations, the tid and item fields are integers for all the remaining queries including candidate generation and rule generation. By mapping the fields this way, we expect longer names to have similar performance impact on all of our architectural options.

4.6.4 Space Overhead of Different Approaches

In Figure 4.10 we summarize the space required for different datasets for three options: `Stored-procedure`, `Cache-Mine` and `SQL`. For these experiments we assume that the tids and items are integers. The first part refers to the space used in caching data and the second part refers to any temporary space used by the DBMS for sorting or alternately for constructing indices to be used during sorting. The size of the data is the same as the space utilization of the `Stored-procedure` approach. The space requirements for `UDF` is the same as that for `Stored-procedure`which requires less space than the `Cache-Mine` and `SQL` approaches. The `Cache-Mine` and `SQL` approaches have comparable storage overheads. For `Stored-procedure` and `UDF` we do not need any extra storage for caching. However,

all three options `Cache-Mine`, `Stored-procedure` and `UDF` require data in each pass to be grouped on the *tid*. In a relational DBMS we cannot assume any order on the physical layout of a table, unlike in a file system. Therefore, we need either an index on the data table or need to sort the table every time to ensure a particular order. Let R denote the total number of (tid,item) pairs in the data table. Either option has a space overhead of $2 \times R$ integers. The `Cache-Mine` approach caches the data in an alternative binary format where each tid is followed by all the items it contains. Thus, the size of the cached data in `Cache-Mine` is at most: $R + T$ integers where T is the number of transactions. For `SQL` we use the hybrid `Vertical` option. This requires creation of an initial TidTable of size at most $I + R$ where I is the number of items. Note that this is slightly less than the cache required by the `Cache-Mine` approach. The `SQL` approach needs to sort data in pass 1 in all cases and pass 2 in some cases where we used the `GatherJoin` approach instead of the `Vertical` approach. This explains the large space requirement for **Dataset-B**. However, in practice when the item-ids or tids are character strings instead of integers, the extra space needed by `Cache-Mine` and `SQL` is a much smaller *fraction* of the total data size because before caching we always convert item-ids to their compact integer representation and store in binary format.

4.7 Summary of Comparison Between Different Architectures

In Table 4.1 we present a summary of the pros and cons of the different architectures by ranking them on a scale of 1 (good) to 4 (bad) on each of the following yardsticks: (a) performance (execution time); (b) storage overhead; (c) scope for automatic parallelization; (d) development and maintenance ease; (e) portability (f) inter-operability.

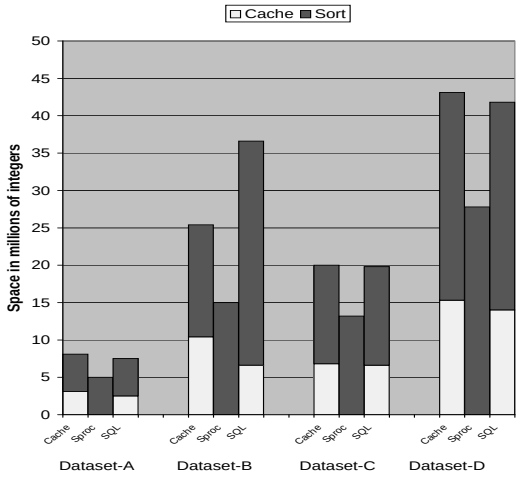


Figure 4.10. Comparison of different architectures on space requirements.

Table 4.1. Pros and cons of different architectural options ranked on a scale of 1(good) to 4(bad)

Metric	`Stored-proc.`	`UDF`	`Cache-Mine`	`SQL`
Performance	4	3	1	2
Storage overhead	1	1	2	2-3
Automatic Parallelism	2	2	2	1(?)
Development and maintenance ease	2	3	2	1-2
Portability	1	3	1	2
Inter-operability	2	2	2	1(?)

In terms of performance, the `Cache-Mine` approach is the best option followed by the `SQL` approach. The `SQL` approach was within a factor of 0.8 to 2 of the `Cache-Mine` approach for all of our experiments. The `UDF` approach is better than the `Stored-procedure` approach in performance by 30 to 50% but it looses on the metrics of development and maintenance costs and portability. In terms of space requirements, the `Cache-Mine` and the `SQL` approach loose to the `UDF` or the `Stored-procedure` approach. Between the `Stored-procedure` and the `Cache-Mine` implementation, the performance difference is exactly a function of the number of passes made on the data; that is, if we make four passes

of the data the `Stored-procedure` approach is four times slower than `Cache-Mine`. Therefore, if one is not willing to pay the penalty of extra storage the best strategy for improving performance is to reduce the number of passes to the data even if it comes at the cost of extra processing. Some of the recent proposals [26, 127] that attempt to minimize the number of data passes to 2 or 3 might be useful in that regard.

The `SQL` approach is not the winner in terms of performance and space requirements but it is competitive. The benefit of the `SQL` approach could arise from other secondary advantages.

The `SQL` implementation has the potential for automatic parallelization. Parallelization could come for free because bulk of our processing is expressed in terms of standard SQL queries. As long as the database supports efficient parallelization of these queries the mining code can be easily parallelized. The problem case is where the UDFs use scratch pads. The only such function in our queries is the `Gather` table function. This function essentially implements a user defined aggregate, and would have been easy to parallelize if the DBMS provided support for user defined aggregates or allowed explicit control from the application about how to partition the data amongst different parallel instances of the function. For MPPs, one could rely on the DBMS to come up with a data partitioning strategy. However, it might be possible to better tune performance if the application could provide hints about the best partitioning [11] to use. Further experiments are required to assess how the performance of these automatic parallelizations would compare with algorithm-specific parallelizations [11].

The development time and code size using `SQL` could be shorter if one can get efficient implementations out of expressing the mining algorithms declaratively using a few SQL statements. Thus, one can avoid writing and debugging code for memory management,

indexing and space management all of which are already provided in a database system. However, there are some detractors to easy development using the `SQL` alternative. First, any attached UDF code will be harder to debug than stand-alone C++ codes due to lack of debugging tools. Second, stand-alone code can be debugged and tested faster when run against flat file data. Running against flat files is typically a factor of five to ten faster compared to running against data stored in DBMS tables. Finally, some mining algorithms (for instance, neural-net based) might be too awkward to express in SQL.

The ease of porting of the `SQL` alternative depends on the kind of SQL used. Within the same DBMS, porting from one OS platform to another requires porting only the small UDF code and hence is easy. In contrast the `Stored-procedure` and `Cache-Mine` alternatives require porting larger lines of code. Porting from one DBMS to another could get hard for `SQL` approach, if non-standard DBMS-specific features are used. Unfortunately, we found SQL-92 implementations (which would have been quite portable) to be unacceptable from the performance viewpoint. Our preferred SQL implementation relies on the availability of DB2's table functions. Table functions, for example, in Oracle 8 do not have the same interface and semantics as DB2. Also, if different features have different performance characteristics on different database systems, considerable tuning would be required. In contrast, the `Stored-procedure` and `Cache-Mine` approach are not tied thus to any DBMS specific features. The `UDF` implementation has the worst of both worlds; it is large and is tied to a DBMS.

The biggest attraction of the SQL implementation is inter-operability and usage flexibility. The *ad hoc* querying support provided by the DBMS enables flexible usage and exposes potential for pipelining the input and output operators of the mining process with

other operators in the DBMS. However, to exploit this feature one either needs to implement the mining operators inside the DBMS or use table functions in novel ways. The first alternative would require major rework in existing database systems. The second alternative requires table functions that can execute SQL statements (a facility not currently available in DB2 UDB). Furthermore, some mining operators generate multiple different kinds of output (for instance, model tree and statistics for decision trees). In such cases, pipelining of mining operators becomes harder to fit in existing relational models. Note that the `SQL` approach presented here is based on embedded SQL and cannot provide operator pipelining and inter-operability. Queries on the mined result is possible with all four alternatives as long as the mined results are stored back in the DBMS.

4.8 Other Associations Algorithms

There are recent proposals [26, 127] that attempt to minimize the number of data passes in the Apriori algorithm to two or three. Currently, the `Stored-procedure` approach makes as many passes as the length of the largest itemset and in each pass, most of the time is spent in reading the data from the DBMS. Therefore, by using these alternative algorithms it is possible to reduce the time spent by `Stored-procedure` by as many times as the number of passes. However, the `Cache-Mine` approach is almost a lower bound on the best performance possible for doing associations outside the DBMS because it spends most of its time in the first pass which is the minimum amount of work any mining algorithm that works outside the DBMS has to spend; read all of the data at least once. Since our `SQL` approach is competitive with the one-pass `Cache-Mine` approach, we expect our conclusions to hold for other algorithms that spend 1.5 to 3 times the time spent by `Cache-Mine`. These alternative algorithms are not likely to help improve the performance of our `SQL`

implementation because they are aimed at reducing the number of passes over the data. We have seen that with our `SQL` implementations hardly any time is spent on passes beyond 2 because of the `Vertical` approach. The time spent in pass 2 is not reduced because they all require counting support of all of C_2 on the entire data anyway.

4.9 Summary

In this chapter, we developed and experimented with a set of approaches that made use of the new object-relational extensions like UDFs, BLOBs and table functions. These approaches performed much better than the SQL-92 approaches in Chapter 3. We developed a hybrid scheme which picks the best approach for each pass based on the cost estimates. We also compare the various architectural alternatives both qualitatively and quantitatively.

Note A part of the work described in this chapter was primarily done by researchers from IBM Almaden Research Center. The author's primary contributions were the optimizations to the approaches, in Sections 4.1.1 and 4.2.1, the hybrid approach in Section 4.5 and the cost analysis of the various support counting approaches. The author has also contributed to the performance experiments which led to the various comparisons.

CHAPTER 5
GENERALIZED ASSOCIATION RULES

In most real-life applications the set of items that appear in transactions can be categorized according to a taxonomy (is-a hierarchy) on the items. The taxonomy shown in Figure 5.1 says that Pepsi *is-a* soft drink *is-a* beverage and so on. In general, a taxonomy can be represented as a directed acyclic graph (DAG). Given a set of transactions T each of which is a set of items, and a taxonomy *Tax*, the problem of mining generalized association rules is to discover all rules of the form $X \rightarrow Y$, with the user-specified minimum support and minimum confidence. X and Y can be sets of items at any level of the taxonomy, such that no item in Y is an ancestor of any item in X [118]. For example, there might be a rule which says that "50% of transactions that contain Soft drinks also contain Snacks; 5% of all transactions contain both these items."

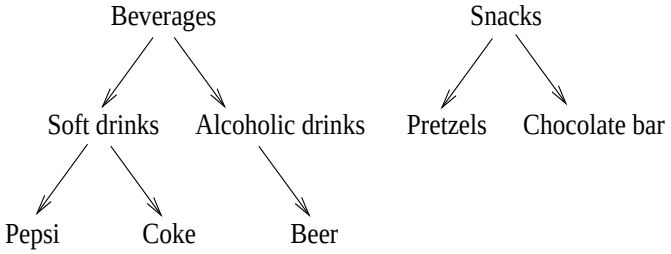


Figure 5.1. Example of a taxonomy

In this chapter, we present several SQL formulations of generalized association rule mining [126]. In Section 5.1, we describe the input-output formats and in Section 5.2, we briefly outline the *Cumulate* algorithm for generalized association rule mining [118].

Table 5.1. An example of the taxonomy table

Parent	Child
Beverages	Soft drinks
Beverages	Alcoholic drinks
Soft drinks	Pepsi
Soft drinks	Coke
Alcoholic drinks	Beer
Snacks	Pretzels
Snacks	Chocolate bar

We present SQL formulations of ancestor pre-computation and candidate generation in Sections 5.3 and 5.4 respectively. Several approaches for support counting based on SQL-92 and SQL-OR are presented in Sections 5.6 and 5.7. In Section 5.8, we discuss some experimental results.

5.1 Input-Output Formats

Input format The transaction table T has the same schema, (tid, item) as in the case of boolean association rules. While transactions normally contain only items that are leaves in the taxonomy, it is not a requirement. The taxonomy table *Tax* has two column attributes, *parent* and *child*. Each record in *Tax* corresponds to an edge in the taxonomy DAG. The taxonomy shown in Figure 5.1 is represented by Table 5.1. The number of children for the interior nodes and the depth of the DAG are unknown at table creation time and therefore alternative schemas may not be convenient.

Output format The output is a collection of all rules that satisfy the user-specified minimum support and minimum confidence. The schema of the rules is the same as in the boolean associations case 3.3.

5.2 Cumulate Algorithm

Srikant and Agrawal [118] present several algorithms: Basic, Cumulate, Stratify and EstMerge. We picked Cumulate for our SQL-formulations since it has the best performance. Stratify and EstMerge are sometimes marginally better but they are far too complicated to merit the additional development cost. Cumulate has the same basic structure as the Apriori algorithm [9] for boolean associations. It makes multiple passes over the data where in the k^{th} pass it finds frequent itemsets of size k. Each pass consists of two phases: In the candidate generation phase, the frequent $(k-1)$-itemsets, F_{k-1}, found in the previous pass is used as the seed set to generate candidate k-itemsets (C_k) that are potentially frequent. In the support counting phase for each itemset $t \in C_k$, the number of extended transactions (transactions augmented with all the ancestors of its items) that contains t is counted. At the end of the pass, the frequent candidates are identified yielding F_k. The algorithm terminates when F_k or C_{k+1} becomes empty.

The above basic structure is augmented with a few optimizations. The important ones are pruning itemsets containing an item and its ancestor and pre-computing the ancestors for each item. We extend the SQL-based boolean association rule framework in Chapters 3 and 4 with these optimizations.

5.3 Pre-Computing Ancestors

In this section, we explain how to compute the set of ancestors of each item using SQL. We call $\hat{x}$ an ancestor of x if there is a directed path from $\hat{x}$ to x in *Tax*. The *Ancestor* table is primarily used for *(i)* pruning candidates containing an item and its ancestor and *(ii)* extending the transactions by adding all the ancestors of its items. The ancestor

computation can be expressed in SQL using a recursive query as shown in Figure 5.2. The result of the query is stored in table *Ancestor* having the schema (*ancestor, descendant*).

```
insert into Ancestor
with      R-Tax (ancestor, descendant) as
          (select parent, child from Tax
                union all
          select  p.ancestor, c.child from R-Tax p, Tax c
          where  p.descendant = c.parent)
select    ancestor, descendant from R-Tax
```

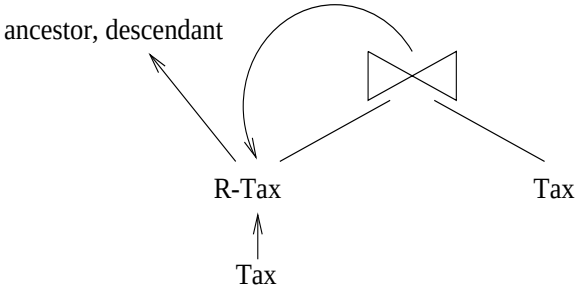


Figure 5.2. Pre-computing ancestors

5.4 Candidate Generation

In the candidate generation phase, we use the frequent itemsets F_{k-1} found in the $(k-1)^{th}$ pass to generate a set of candidate itemsets C_k that contains all k itemsets such that all k of its $(k-1)$-length subsets are in F_{k-1}. Section 3.4.1 shows how to express this operation as a k-way join between the frequent $(k-1)$-itemsets (F_{k-1}'s). We can use the same formulation except that we need to prune from C_k itemsets containing an item and its ancestor. Srikant and Agrawal [118] prove that this pruning needs to be done only in

the second pass (for C_2). In the SQL formulation as shown in Figure 5.3, we prune all (ancestor, descendant) pairs from C_2 which is generated by joining F_1 with itself.

insert into C_2

(select $I_1.item_1, I_2.item_1$ from $F_1 I_1, F_1 I_2$

where $I_1.item_1 < I_2.item_1$)

except

(select ancestor, descendant from Ancestor union

select descendant, ancestor from Ancestor)

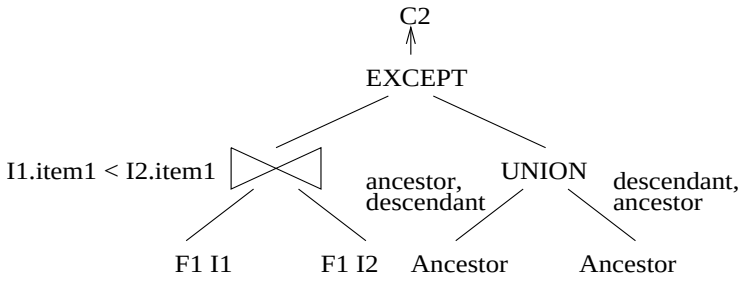


Figure 5.3. Generation of C_2

5.5 Support Counting to Find Frequent Itemsets

We use the candidate itemsets C_k, the data table T and the ancestor table *Ancestor* to count the support of the itemsets in C_k. We consider two categories of SQL implementations based on SQL-92 and SQL-OR in Sections 5.6 and 5.7 respectively. All the SQL approaches developed for boolean associations in Chapters 3 and 4 can be extended to handle taxonomies. However, we present the extensions to only a few representative approaches. In particular, we consider the `KwayJoin` approach from SQL-92 and `Vertical` and `GatherJoin` from SQL-OR.

5.6 Support Counting Using SQL-92

5.6.1 K-way Join

In each pass k, we join the candidate itemsets C_k with k copies of an extended transaction table T^* (defined below), and follow it up with a group by on the itemsets.

The extended transaction table T^* is obtained by augmenting T to include $tid, item$ entries for all ancestors of items appearing in T. This can be formulated as a SQL query as shown in Figure 5.4.

Query to generate T^*

```
select item, tid from T union
select distinct A.ancestor as item, T.tid
from T, Ancestor A
where A.descendant = T.item)
```

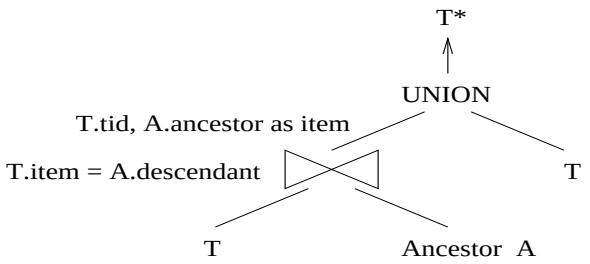


Figure 5.4. Transaction extension subquery

The select distinct clause is used to eliminate duplicate records due to extension of items with a common ancestor. Note that for this approach we do not materialize T^*. Instead we use the SQL support for common subexpressions (with construct) to pipeline the generation of T^* with the join operations. The final SQL query and the corresponding tree diagram are shown in Figure 5.5.

insert into F_k

with T^*(tid, item) as (Query for T^* defined above)

select $item_1, \ldots item_k$, count(*)

from $C_k, T^*\ t_1, \ldots T^*\ t_k$

where t_1.item = $C_k.item_1$ and

$\vdots$

t_k.item = $C_k.item_k$ and

t_1.tid = t_2.tid and

$\vdots$

t_{k-1}.tid = t_k.tid

group by $item_1$,$item_2 \ldots item_k$

having count(*) > :minsup

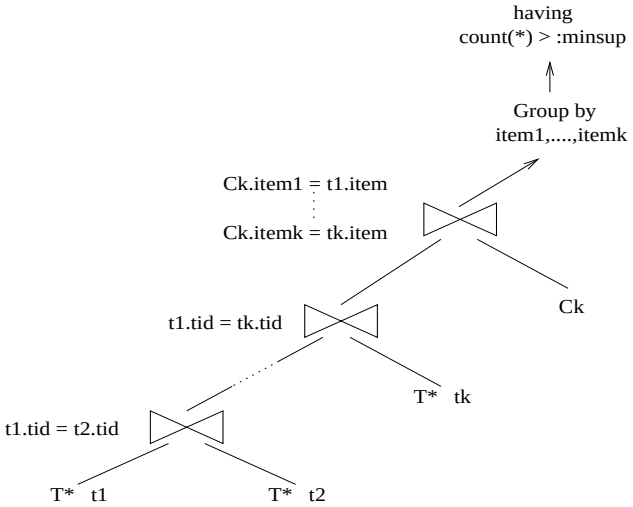


Figure 5.5. Support counting by K-way join

The `3wayJoin` and `2GroupBy` approaches for boolean associations can be extended to handle taxonomies, in a similar way by replacing T with T^* in the corresponding queries.

5.6.2 Subquery Optimization

The basic `KwayJoin` approach can be optimized to make use of common prefixes between the itemsets in C_k by splitting the support counting phase into a cascade of k subqueries. The subqueries in this case are exactly similar to those for boolean associations presented in 3.5.4 except for the use of T^* instead of T.

5.7 Support Counting Using SQL-OR

In this section we present three approaches that make use of the object-relational features of SQL. We also present a cost-based analysis of the execution time of the various approaches.

5.7.1 GatherJoin

The `GatherJoin` approach, which is based on the use of table functions [80], generates all possible k-item combinations of extended transactions, joins them with the candidate table C_k and counts the support by grouping the join result. The extended transactions T^* (defined in Section 5.6.1) are passed to the table function `GatherComb-K` in the (*tid, item*) order. A record output by the table function is a k-item combination supported by a transaction and has k attributes $T\_item_1, \ldots, T\_item_k$. SQL queries for this approach is presented in Figure 5.6.

The special optimization for pass 2 and the variations of the `GatherJoin` approach, namely `GatherCount`, `GatherPrune` and `Horizontal` (refer Section 4.1) are also applicable here.

insert into F_k select $item_1, \ldots, item_k$, count(*)

from C_k,

(select $t_2.T\_itm_1, \ldots, t_2.T\_itm_k$ from T^* t_1,

table (GatherComb-K(t_1.tid, t_1.item)) as t_2)

where $t_2.T\_itm_1 = C_k.item_1$ and

$\vdots$

$t_2.T\_itm_k = C_k.item_k$

group by $C_k.item_1, \ldots, C_k.item_k$

having count(*) > :minsup

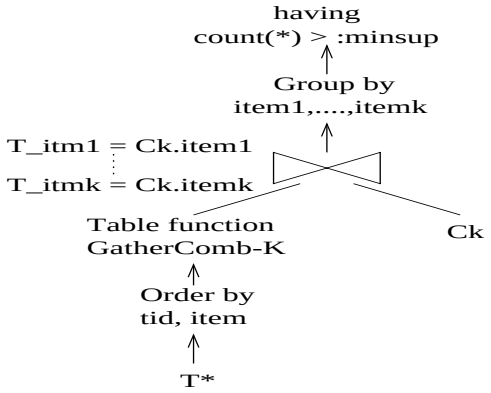


Figure 5.6. Support counting by `GatherJoin`

5.7.2 GatherExtend

This is a variation of the `GatherJoin` approach where we push the transaction extension inside the table function. For each item, we create an ancestor-list (stored as a BLOB) which contains a list of ancestors of that item. This can be accomplished using similar SQL queries as in the tid-list creation phase of the `Vertical` approach for boolean associations (refer Section 4.2). The (item, ancestor-list) pairs are stored in a new AncListTable. Note that ancestor-list needs to be created only for items that appear in the input transactions.

In the support counting phase, we join the transaction table T with the AncListTable and the resulting (tid, item, ancestor-list) tuples are passed to a table function `GExtendComb-K` in tid order. The table function collects all the items and the corresponding ancestor-lists of the same transaction, extends it and outputs k-item combinations of the extended transaction. Figure 5.7 illustrates the SQL queries for this approach. This approach was not implemented because it might not perform better than the Vertical approach explained next.

This approach can be combined with the `GatherPrune` approach (refer Section 4.1 to prune out item combinations that are not candidates. In that case, we can also delete from the extended transactions items that are not present in any of the candidates, before generating the item combinations.

5.7.3 Cost Analysis

In the cost analysis we use the notations of Table 5.2 in addition to the notations introduced for boolean associations in Table 3.2.

Table 5.2. Additional notations used for cost analysis

D	number of records in the input taxonomy table
d	average depth of the taxonomy
l	number of leaf nodes in the taxonomy
A	number of records in the Ancestor table $\approx l * d$
N^*	average number of items in an extended transaction $= N * d$
R^*	total number of records after transaction extension $= R * d$
e_k	cost of generating a k-item combination using `GExtendComb-K`
order(n)	cost of sorting n records

The cost of `GatherJoin` includes the cost of extending the transactions by joining the transaction table with the ancestor table, generating k-item combinations, joining them with C_k and grouping the join result to count the support. Note that the transaction

insert into F_k select $item_1, \ldots, item_k$, count(*)

from C_k, (select $t_2.T\_itm_1, \ldots, t_2.T\_itm_k$

from (select tid, item, ancestor-list

from T, AncListTable A

where T.item = A.item) as t_1,

table (GExtendComb-K(t_1.tid, t_1.item, t_1.ancestor-list)) as t_2)

where $t_2.T\_itm_1 = C_k.item_1$ and

$\vdots$

$t_2.T\_itm_k = C_k.item_k$

group by $C_k.item_1, \ldots, C_k.item_k$

having count(*) > :minsup

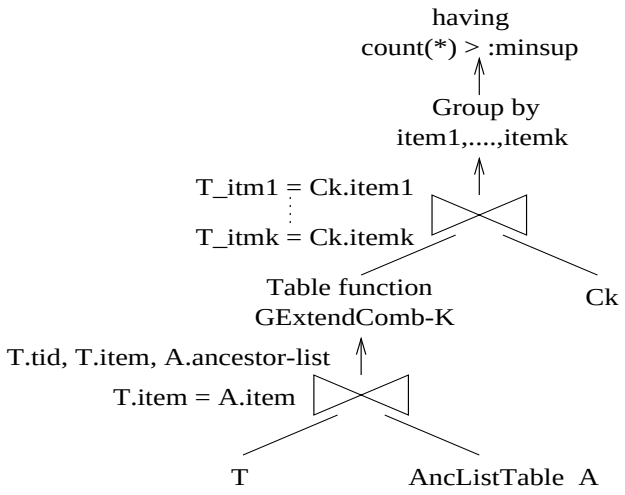


Figure 5.7. Support counting by `GatherExtend`

extension cost can be ignored if T^* is materialized because in that case, it will be a one-time cost. The result size of the join which extends the transactions is R^* and the number of k-item combinations generated, T_k is $C(N^*, k) * T$. Therefore, the total cost of the `GatherJoin` approach is

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

In `GatherExtend`, transaction extension is done inside the table function. The cost of passing the ancestor-list as a BLOB is blob(d) since the average length of the ancestor-list is the same as the depth of the taxonomy. The cost of `GatherExtend` is

$$\text{join}(R, l, R) + R * \text{blob}(d) + T_k * e_k + \text{join}(T_k, C_k, S(C_k)) + \text{group}(S(C_k), C_k)$$

5.7.4 Vertical

In this approach, the transactions are first converted into a vertical format by creating for each item a BLOB (tid-list) containing all tids that contain that item. The support for each itemset is counted by merging the tid-lists of all its items. The tid-list of leaf node items can be created using a table function in the same way as in the boolean associations case. We present two approaches for creating the tid-list of the interior nodes in the taxonomy DAG.

The first approach is based on doing the union of the descendant's tid-lists of an interior node. In the first phase, we create an initial TidTable containing the tid-lists of the leaf nodes. The TidTable is joined with the ancestor table and the join result is passed in the order of the *ancestor* attribute to the table function `TUnion`. For every node x, the

```
insert into TidTable select t2.item, t2.tid-list
from       (select A.ancestor, T.tid-list
            from   TidTable T, Ancestor A
            where  T.item = A.descendant
            order by ancestor) as t1,
            table(TUnion(t1.ancestor, t1.tid-list)) as t2
```

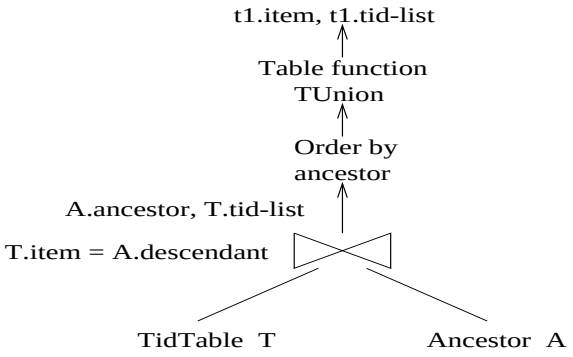


Figure 5.8. Interior nodes' tid-list generation by union

table function collects the tid-lists of all the leaf nodes reachable from x, union them and outputs the tid-list for x. The SQL query for this is illustrated in Figure 5.8. This approach can be further optimized to do the tid-list union of only the children of each interior node. However, in the first phase tid-lists have to be created and materialized for every leaf node item irrespective of its support. This is very expensive especially when the number of items is large.

The second approach is to pass T^* (defined in Section 5.6.1) to the `Gather` table function, as shown in Figure 5.9. The table function outputs tid-lists for all the items in the taxonomy.

The support counting queries are exactly the same as for boolean associations explained in Section 4.2. The cost formula is also the same as in boolean associations except

insert into TidTable select t_2.item, t_2.tid-list

from T^* t_1, table(Gather(t_1.item, t_1.tid, :minsup)) as t_2

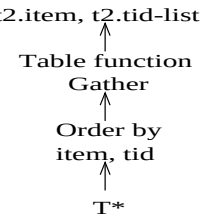


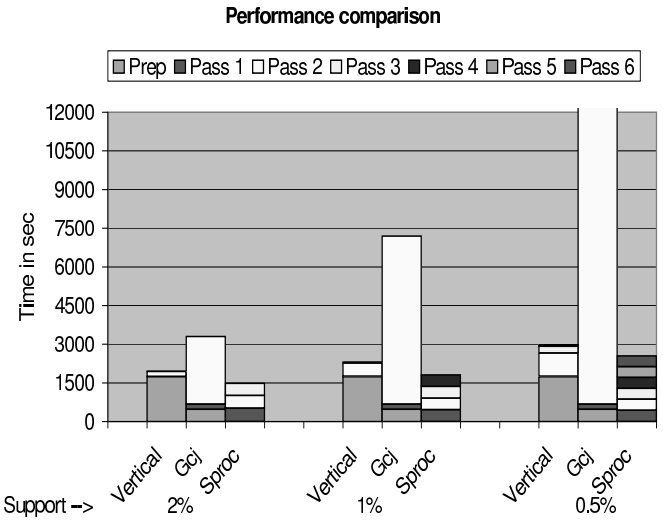
Figure 5.9. Interior nodes' tid-list generation from T^*

that the average length of a tid-list in this case is $\frac{R_f^*}{F_1}$, where R_f^* is the sum of support counts of all the frequent items.

5.8 Performance Results

In this section, we report the results of some of our performance experiments on real-life datasets. We present only one set of results since our emphasis here is on showing that the boolean association rule framework can be easily extended to handle more complex mining computations like generalized association rules.

The experiments reported here were performed on Version 5 of IBM DB2 Universal Server installed on a RS/600 Model 140 with a 200MHz CPU, 256 MB main memory and a 9 GB disk with a measured transfer rate of 8 MB/sec. Figure 5.10 shows the performance of three approaches: Vertical, `GatherJoin` (shown as Gcj) and `Stored-procedure` (shown as Sproc). The chart shows the preprocessing time and the time taken for the different passes. For `Vertical` the preprocessing time includes ancestor pre-computation and tid-list creation times, where as for `GatherJoin` it is just the time for ancestor pre-computation. In the stored procedure approach, the mining algorithm is encapsulated as a stored procedure [32]



Mail order data:

Total number of records = 2.5 million

Number of transactions = 568000

Number of items (leaf nodes in taxonomy DAG) = 85161

Total number of items (including interior nodes) = 228428

Max. depth of the taxonomy = 7

Avg. number of children per node = 1.6

Max. number of parents = 3

Figure 5.10. Comparison of different SQL approaches

which runs in the same address space as the DBMS. For the `Stored-procedure` experiment, we used the generalized association rule implementation provided with the IBM data mining product, Intelligent Miner [71]. For all the support values, the `Vertical` approach performs equally well as the `Stored-procedure` approach. In some of the experiments on other datasets, the `Vertical` approach performed better than the `Stored-procedure` approach. The `GatherJoin` approach is worse mainly due to the large number of item combinations generated. In the `GatherJoin` approach, the extended transactions are passed to the GatherComb table function and hence the effective number of items per transaction

gets multiplied by the average depth of the taxonomy. In the `GatherJoin` approach, we show the performance numbers for only the second pass. Note that just the time for second pass is an order of magnitude more than the total time for all the passes of `Vertical`. For 0.5% support, second pass of `GatherJoin` took over 20,000 seconds while the total time for `Vertical` was only about 3000 seconds. We did not do extensive experimentation here because, based on the SQL formulations and the analysis, we expect similar performance observations as in the case of boolean association rules.

5.9 Summary

In this chapter, we presented various SQL formulations for mining generalized association rules. It shows that the boolean association rule framework can be easily extended for generalized association rules. The major additions were to "extend" the input transaction table (transform T to T^*) and to pre-compute the ancestors.

We have presented here the extensions only for a few representative SQL approaches for boolean association rules in Chapters 3 and 4. The other approaches can be extended in a similar manner. We expect the performance trends from boolean association rules to hold for all these approaches for generalized associations also. There could be better ways of handling the taxonomy like the *EstMerge* and *Stratify* algorithms presented in Srikant and Agrawal [118]. However, with the currently available SQL operators they will be too awkward to program. One possibility is to push some of those optimizations inside user-defined functions and it remains to be seen how they will influence the performance.

CHAPTER 6
SEQUENTIAL PATTERNS

Sequential pattern mining utilizes the time associated with the transaction data to find frequently occurring patterns. A sequential pattern is an ordered list (sequence) of itemsets (refer Section 1.2.2 for a brief description of sequential patterns). Given a set of data-sequences each of which is a list of transactions ordered by the transaction time, the problem of mining sequential patterns is to discover all such patterns with a user-specified minimum *support*.

We present several SQL formulations of sequential pattern mining in this chapter. In Section 6.1, we describe the input-output formats and in Section 6.2, we briefly outline the GSP algorithm for sequential pattern mining [120]. SQL-based sequential pattern candidate generation is presented in Section 6.3. In Sections 6.5 and 6.6, we present several SQL formulations for support counting based on SQL-92 and SQL-OR. In Section 6.7, we briefly mention how to handle taxonomies over items. We concentrate on showing that sequential pattern mining can also be handled within the same SQL-based framework. We expect the performance trends from association rule mining to hold here also.

6.1 Input-Output Formats

6.1.1 Input Format

The input table D of data-sequences has three column attributes: sequence identifier (*sid*), transaction time (*time*) and item identifier (*item*). Every data-sequence typically

contains several transactions each of which is a set of items. The data-sequence table contains multiple rows corresponding to different items that belong to transactions in the data-sequence. The taxonomy on the items is represented in the same way as for the generalized association rules.

6.1.2 Output Format

The output is a collection of frequent sequences. A sequence which is represented as a tuple in a fixed-width table consists of an ordered list of elements where each element is a set of items. The schema of the frequent sequences table is $(item_1, eno_1, \ldots, item_k, eno_k, len)$. The *len* attribute gives the length of the sequence, that is, the total number of items in all the elements of the sequence. The *eno* attributes stores the element number of the corresponding items. For sequences of smaller length, the extra column values are set to NULL. For example, if $k = 4$, the sequence $\langle (computer, modem)(printer) \rangle$ is represented by the tuple *(computer, 1, modem, 1, printer, 2, NULL, NULL, 3).*

6.2 GSP Algorithm

We use the GSP algorithm of Srikant and Agrawal [120] as the basis for our SQL formulations. The basic structure of the GSP algorithm is similar to that of the apriori algorithm for association rule mining [9], although the specific details are quite different. Therefore, the SQL-based association rule framework can be used for sequential patterns also. The algorithm makes multiple passes over the data-sequences where in the k^{th} pass it finds frequent sequences of length k. Each pass consists of two phases: the *candidate generation* phase and the *support counting* phase. In the candidate generation phase, the frequent $(k-1)$-length sequences, F_{k-1}, found in the previous pass is used as the seed set to generate candidate k-length sequences (C_k) that are potentially frequent. The candidate

generation procedure ensures that C_k is a superset of F_k. The counting phase makes a pass over the input data-sequences to find the support for each candidate sequence. In the support counting phase for each candidate sequence $s \in C_k$, the number of distinct data-sequences that contains s is counted. Two techniques to perform this operation efficiently are explained in Srikant and Agrawal [120]: one which uses a hash-tree data structure and another based on transforming the data-sequences into an alternate representation. The algorithm terminates when the set of candidate sequences C_k or the frequent sequences F_k becomes empty.

6.3 Candidate Generation

In each pass k, the candidate k-sequences C_k are generated from the frequent $(k-1)$ sequences F_{k-1}. The schema of C_k is the same as that of frequent sequences explained above in Section 6.1, except that we do not require the *len* attribute since all the tuples in C_k have the same length k.

Candidates are generated in two steps. The *join* step generates a superset of C_k by joining F_{k-1} with itself. A sequence s_1 joins with s_2 if the subsequence obtained by dropping the first item of s_1 is the same as the one obtained by dropping the last item of s_2. This can be expressed in SQL as follows.

insert into C_k select $I_1.item_1$, $I_1.eno_1$, ..., $I_1.item_{k-1}$, $I_1.eno_{k-1}$, $I_2.item_{k-1}$,

$I_1.eno_{k-1} + I_2.eno_{k-1} - I_2.eno_{k-2}$

from $F_{k-1}\ I_1, F_{k-1}\ I_2$

where $I_1.item_2 = I_2.item_1$ and

$\vdots$

$I_1.item_{k-1} = I_2.item_{k-2}$ and

$$I_1.eno_3 - I_1.eno_2 = I_2.eno_2 - I_2.eno_1 \text{ and}$$

$$\vdots$$

$$I_1.eno_{k-1} - I_1.eno_{k-2} = I_2.eno_{k-2} - I_2.eno_{k-3}$$

In the above query, subsequence matching is expressed as join predicates on the attributes of F_{k-1}. Note the special join predicates on the `eno` fields that ensure that not only do the joined sequences contain the same set of items but that these items are grouped in the same manner into elements. The result of the join is the sequence obtained by extending s_1 with the last item of s_2. The added item becomes a separate element if it was a separate element in s_2, and part of the last element of s_1 otherwise. In our representation of the candidate sequence, eno_{k-1} and eno_{k-2} determine if the added item is a separate element.

For example, let F_3 be $\{\langle(1,2)\ (3)\rangle, \langle(1,2)\ (4)\rangle, \langle(1)\ (3,4)\rangle, \langle(1,3)\ (5)\rangle, \langle(2)\ (3,4)\rangle, \langle(2)\ (3)\ (5)\rangle\}$. In the join step, the sequence $\langle(1,2)\ (3)\rangle$ joins with $\langle(2)\ (3,4)\rangle$ to generate $\langle(1,2)\ (3,4)\rangle$ and with $\langle(2)\ (3)\ (5)\rangle$ to generate $\langle(1,2)\ (3)\ (5)\rangle$. There are no other join compatible sequences in F_3.

In the *prune* step, all candidate sequences that have a non-frequent contiguous $(k-1)$-subsequence are deleted. In the above example, the sequence $\langle(1,2)\ (3)\ (5)\rangle$ will be deleted since its contiguous subsequence $\langle(1)\ (3)\ (5)\rangle$ is not frequent.

We perform both the join and prune steps in the same SQL statement by writing the above query as a k-way join as shown in Figure 6.1. For any k-sequence there are *at most* k contiguous subsequences of length $(k-1)$ for which F_{k-1} needs to be checked for membership. Note that all $(k-1)$-subsequences may not be contiguous because of the *max-gap* constraint between consecutive elements. The join predicates on I_1 and I_2 remain the same. The join of I_1 and I_2 generates a k-sequence $(I_1.item_1, \ldots, I_1.item_{k-1}, I_2.item_{k-1})$

where two of its $(k-1)$-subsequences are known to be frequent; it is generated from two such $(k-1)$-subsequences. Membership checks for the other $(k-2)$ subsequences are performed using additional joins. The join predicates for these joins are enumerated by dropping one item at a time from the generated k-sequence. We first drop $item_2$ and if it results in a contiguous subsequence, we check for its membership in F_{k-1} by the join with I_3. For the join with I_r, we drop $item_{r-1}$.

The OR clause in the join predicate is to avoid checking for non-contiguous subsequences that are formed when $eno_{r+1} - eno_{r-1} = 2$ When there is no max-gap constraint, the join predicates will not contain the OR part.

While joining F_1 with itself to get C_2, we need to generate sequences where both the items appear as a single element as well as two separate elements. Also note that the prune step will not delete any candidate sequences. The generation of C_2 is expressed in SQL as

insert into C_2 (select $I_1.item_1$, 1, $I_2.item_1$, 2
from $F_1\ I_1, F_1\ I_2$) union
(select $I_1.item_1$, 1, $I_2.item_1$, 1
from $F_1\ I_1, F_1\ I_2$
where $I_1.item_1 < I_2.item_1$)

For instance, if F_1 contains {(1), (2)}, C_2 will have {⟨(1) (1)⟩, ⟨(1) (2)⟩, ⟨(2) (1)⟩, ⟨(2) (2)⟩, ⟨(1,2)⟩}.

6.4 Support Counting to Find Frequent Sequences

In each pass k, we use the candidate table C_k and the input data-sequences table D to count the support. We consider SQL implementations based on SQL-92 and SQL-OR in Sections 6.5 and 6.6 respectively.

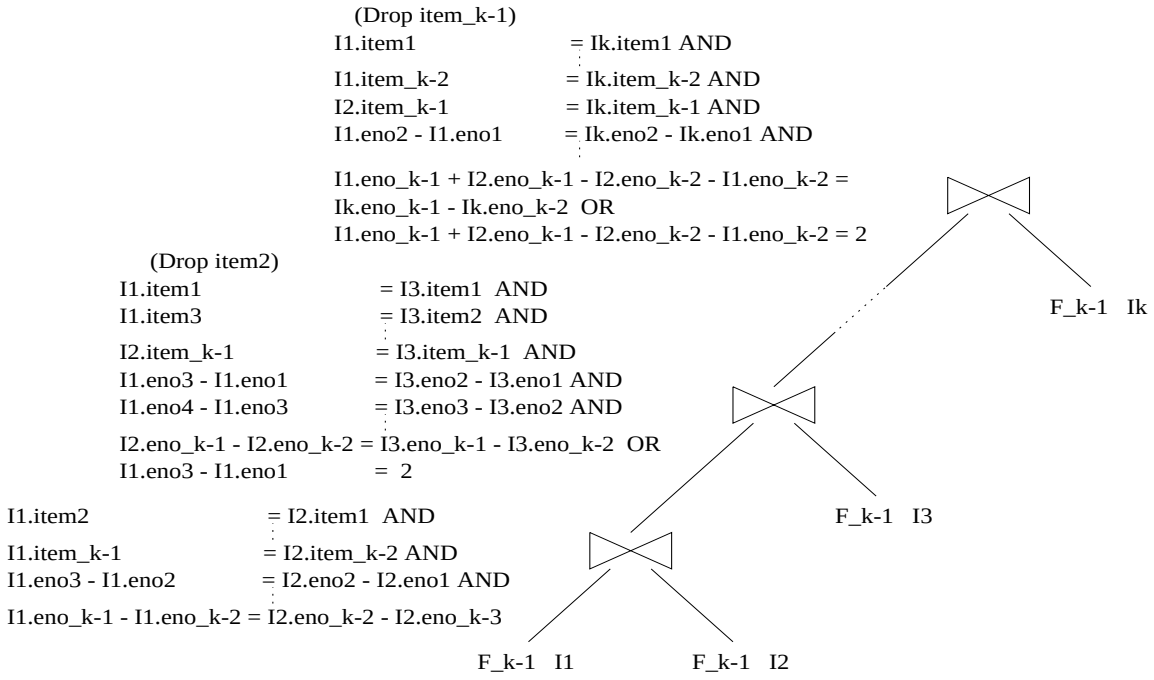


Figure 6.1. Candidate generation for any k

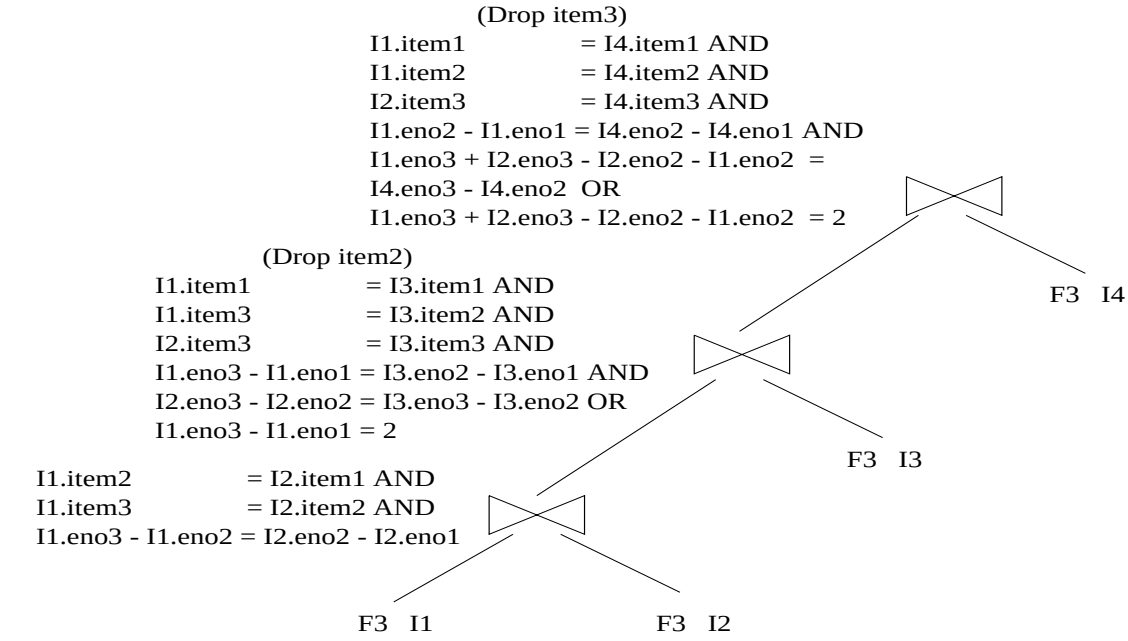


Figure 6.2. Candidate generation for $k = 4$

6.5 Support Counting Using SQL-92

6.5.1 K-way Join

In the k^{th} pass, we join the candidate table C_k with k copies of the data-sequence table D and group the join result on the candidate sequences as shown in Figure 6.3. This approach is very similar to the `KwayJoin` approach for association rules (refer Section 3.5.1) except for the following two key differences.

1. We use select distinct to ensure that only distinct data-sequences are counted.

2. Second, we have additional predicates between sequence numbers that are denoted as PRED(k) in the query for brevity. The predicates PRED(k) is a conjunct (and) of terms $p_{ij}(k)$ corresponding to each pair of items from C_k. $p_{ij}(k)$ is expressed as

$$
\begin{aligned}
&(C_k.eno_j = C_k.eno_i \text{ and } \text{abs}(d_j.time - d_i.time) \leq \text{window-size}) \text{ or} \\
&(C_k.eno_j = C_k.eno_i + 1 \text{ and } d_j.time - d_i.time \leq \text{max-gap and} \\
&\qquad\qquad d_j.time - d_i.time > \text{min-gap}) \qquad \text{or} \\
&(C_k.eno_j > C_k.eno_i + 1)
\end{aligned}
$$

   Intuitively, these predicates check (a) if two items of a candidate sequence belong to the same element, then the difference of their corresponding transaction times is at most *window-size* and (b) if two items belong to adjacent elements, then their transaction times are at most *max-gap* and at least *min-gap* apart.

We compute the frequent 1-sequences by grouping the data-sequences table on the *item* attribute, counting the number of distinct sequences in which the item is present and filtering the non-frequent items. The SQL query for this computation is given below.

insert into F_1 (select item, 1, count(*)

from (select distinct sid, item

from D) as t

group by item

having count(*) > :minsup

6.5.2 Subquery Optimization

The subquery optimization for association rules can be applied for sequential patterns also by splitting the support counting query in pass k into a cascade of k subqueries. The predicates p_{ij} can be applied either on the output of subquery Q_k or sprinkled across the different subqueries (shown as SubQ-PRED(l) in Figure 6.4). The predicates SubQ-PRED(l) is a conjunct of terms $p_{ij}(l)$ corresponding to each pair of items from d_l, the distinct l-item prefixes of C_k. $p_{ij}(l)$ is expressed as

$(d_l.eno_j = d_l.eno_i$ and abs$(time_j - time_i) \leq$ window-size) or

$(d_l.eno_j = d_l.eno_i + 1$ and $time_j - time_i \leq$ max-gap and

$time_j - time_i >$ min-gap) or

$(d_l.eno_j > d_l.eno_i + 1)$

6.6 Support Counting Using SQL-OR

6.6.1 Vertical

This approach is similar to the `Vertical` approach for association rules (refer Section 4.2), where we convert the transaction table into a vertical format. For each item, we create a BLOB (s-list) containing all (*sid, time*) pairs corresponding to that item. We use a table function `Gather` for creating the s-lists. The sequence table D is scanned in the

insert into F_k

select $item_1,\ eno_1, \ldots, item_k,\ eno_k$, count(*)

from (select distinct d_1.sid, $item_1,\ eno_1, \ldots, item_k,\ eno_k$

from C_k, D d_1,...,D d_k

where d_1.item = $C_k.item_1$ and

$\vdots$

d_k.item = $C_k.item_k$ and

d_1.sid = d_2.sid and

$\vdots$

d_1.sid = d_k.sid and

PRED(k)

) as t

group by $item_1,\ eno_1, \ldots, item_k,\ eno_k$

having count(*) > :minsup

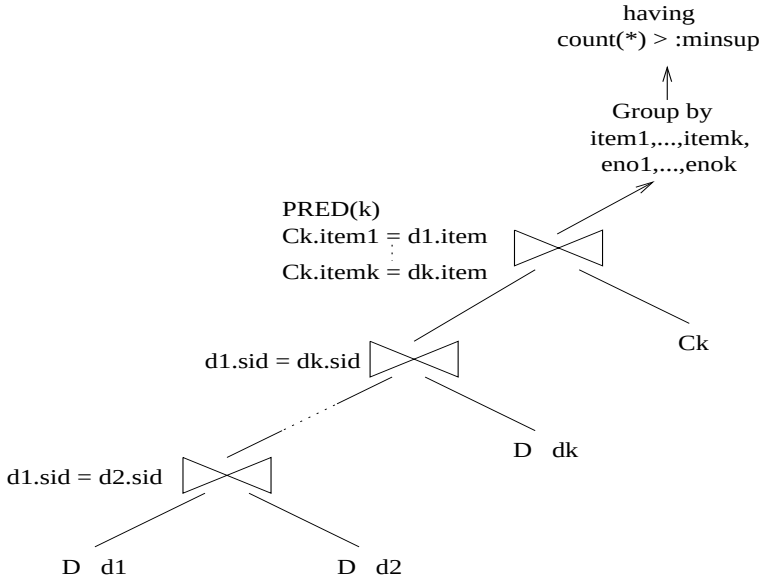


Figure 6.3. Support counting by K-way join

insert into F_k

select $item_1, eno_1, \ldots, item_k, eno_k$, count(*)

from (select distinct $sid, item_1, eno_1, \ldots,$

$item_k, eno_k$ from (`Subquery` Q_k))

group by $item_1, eno_1, \ldots, item_k, eno_k$

having count(*) > :minsup

`Subquery` Q_l (for any l between 1 and k):

select $item_1, eno_1, time_1, \ldots item_l, eno_l, t_l.time$, sid

from D t_l, (`Subquery` Q_{l-1}) as r_{l-1},

(select distinct $item_1, eno_1, \ldots$

$item_l, eno_l$ from C_k) as d_l

where $r_{l-1}.item_1 = d_l.item_1$ and ... and

$r_{l-1}.item_{l-1} = d_l.item_{l-1}$and

$r_{l-1}.sid = t_l.sid$ and

$t_l.item = d_l.item_l$ and SubQ-PRED(l)

`Subquery` Q_0: No subquery Q_0.

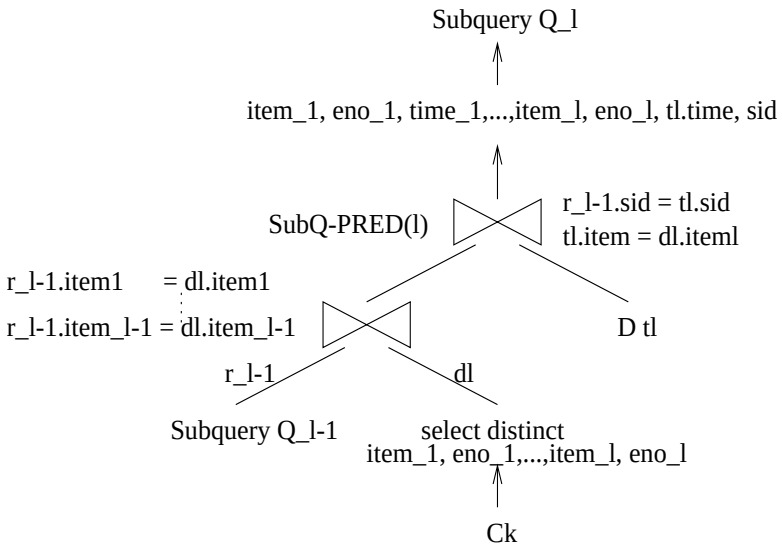


Figure 6.4. Subquery optimization for `KwayJoin` approach

(item, sid, time) order and passed to the function `Gather`, which collects the (sid, time) attribute values of all tuples of D with the same item in memory. For items that meet the minimum support criterion, the function outputs a (item, s-list) pair. The s-lists are maintained sorted using sid as the major key and time as the minor key, and is stored in a new SlistTable with the schema (item, s-list).

In the support counting phase, we collect the s-lists of all the items of a candidate and intersect them to determine the data-sequences containing that sequence. For each candidate sequence in C_k, we select from the SlistTable the s-lists corresponding to the k items in the sequence and pass them to a UDF `CountIntersect-K` along with the *eno* attributes of the candidate, as shown in Figure 6.5. The UDF intersects the s-lists and does the support counting. Eventhough this approach is the same as the one for associations, the `CountIntersect-K` function is quite different.

The UDF first finds the data-sequence containing all the items and marks the boundaries of it in each of the s-lists. Note that the s-lists are maintained sorted on (sid, time). The UDF also groups the s-lists according to the elements of the candidate sequence using the values of the *eno* parameters. For determining whether the candidate is contained in the data-sequence, we use an algorithm similar to the one described in Srikant and Agrawal [120]. The details of this procedure are different due to the vertical representation of data-sequences. If the candidate is contained in the data-sequence the support count is incremented and the same steps are repeated for the subsequent data-sequences.

The intersect operation can be decomposed to share it across sequences having common prefixes similar to the subquery optimization in the `KwayJoin` approach. In this approach we need to use two UDFs, `Intersect` and `Count`. The former intersects the s-lists to filter out the data-sequences that does not contain all the items of the candidate. A new s-list

insert into F_k

select $t.item_1,\ t.eno_1, \ldots, t.item_k,\ t.eno_k$, cnt

from (select $item_1,\ eno_1, \ldots, item_k,\ eno_k$,

CountIntersect-K($C_k.eno_1, \ldots, C_k.eno_k$, d_1.s-list,. . . ,d_k.s-list,

window-size,min-gap,max-gap) as cnt

from C_k, SlistTable d_1,. . . ,SlistTable d_k

where d_1.item = $C_k.item_1$ and

$\vdots$

d_k.item = $C_k.item_k$) as t

where cnt > :minsup

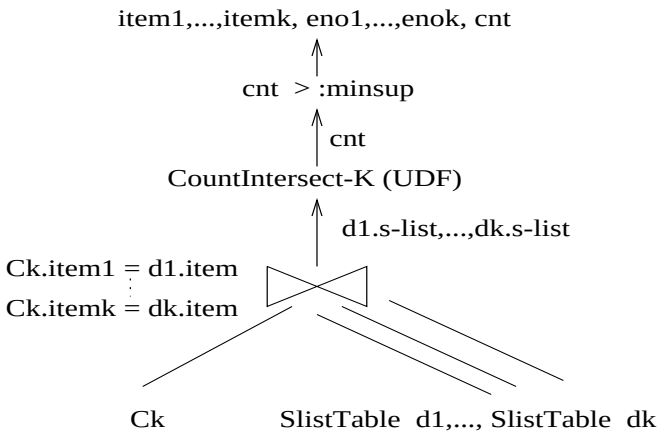


Figure 6.5. Support counting by `Vertical`

corresponding to the intersection is created and is passed to `Count` which counts the number of data-sequences containing the candidate. In the later passes of the GSP algorithm, this approach might be better since several candidates may share common prefixes and also the length of the intersected s-list is likely to be small. However, in the approach using CountIntersect-K, the intersected s-list is not materialized since the UDF interleaves the intersection and counting. Therefore, it is suitable for the earlier passes.

6.6.2 GatherJoin

The `GatherJoin` approach for association rules (refer Section 4.1) can be extended to mine sequential patterns also. In this approach, we first Gather all the $(tid, item)$ pairs corresponding to fixed values of sid. We then generate all possible k-sequences using a table function, join them with C_k and group the join result to count the support of the candidate sequences. The time constraints are checked on the table function output using join predicates PRED(k) as in the `KwayJoin` approach. The number of tuples generated by the table function can be reduced by applying the max-gap, min-gap and window-size constraints inside the table function. However, PRED(k) is still required to check if the sequences in C_k are contained in the generated k-sequences. The items in a candidate sequence are not lexicographically ordered. Therefore, a data-sequence containing n items can potentially support n^k k-sequences. For real-life datasets, the value of n is large and hence this approach will generate too many sequences, thereby making it less attractive in terms of performance. However, an approach similar to `GatherCount` will be feasible where we keep the candidate sequences inside the table function and generate only sequences present in the candidate set. The `GatherJoin` approach was not implemented for sequential patterns.

6.7 Taxonomies

The approaches to handle taxonomies for association rules (refer Chapter 5) are applicable for sequential patterns also. The basic idea is to replace each data-sequence d with an "extended-sequence" d'. Each transaction of d is extended with all the ancestors of its items to get d'. In order to optimize the performance, we pre-compute the ancestors and prune candidates with an element that contains both an item and its ancestor.

6.8 Summary

In this chapter, we presented various SQL formulations for mining sequential patterns. We extended a few representative association rule mining approaches in order to show that sequential pattern mining can also be handled with in the SQL framework. The major changes were in the join predicates for candidate generation and support counting.

We did not do extensive experimentation for the sequential pattern mining approaches. Among the SQL-based approaches, the Vertical approach performed the best in our experiments. The SQL queries for the higher numbered passes becomes too long since there are several join predicates and as a result the optimizer does not generate optimal execution plans. Further, in the Vertical approach, the user-defined functions for support counting are much more complex than their association rule counterparts. We also did not implement the extensions to handle taxonomies in this case.

CHAPTER 7
INCREMENTAL MINING

Data mining discovers information within data warehouses and finds answers to questions about your data that you haven't thought to ask. The rules discovered from a database only reflect the current state of the database. In order to make the discovered rules reliable and useful, large volumes of data need to be collected and analyzed over a period of time. This entails the development of techniques to handle large volumes of data, and to maintain rules over a significantly long period of time. Therefore, efficient algorithms to update, maintain and manage the discovered rules are central to the database mining technology.

Association rule mining is an important data mining problem which finds applications in various business domains. Association rules are discovered from a transaction database and updates to it could potentially invalidate existing rules or introduce new rules. The problem of updating the rules can be reduced to finding the new set of frequent itemsets. Since rule generation is computationally inexpensive, it is not critical to develop incremental rule generation algorithms. A simple solution to the update problem is to re-compute the frequent itemsets from the updated database. This is clearly inefficient because all the computations done initially for finding the old frequent itemsets are wasted. An algorithm, FUP (Fast Update), for updating the frequent itemsets has been developed for the addition of new transactions to the database [36]. It is based on the Apriori algorithm and needs

$O(n)$ passes over the database where n is the size of the maximal frequent itemset. Another incremental algorithm is presented in Feldman et al. [47].

In this chapter, we present an algorithm to find the new frequent itemsets with minimal re-computation when new transactions are added to or deleted from the transaction database [123]. Deletion is important in cases where we want to analyze the data in a sliding time window. The important characteristics of our algorithm are the following.

- Along with the frequent itemsets, we also maintain the *negative border* [127][1] . The algorithm uses negative borders to decide when to scan the whole database and it can be used in conjunction with any level-wise algorithm like Apriori [13] or Partition [111].

- We first compute the frequent itemsets of the increment database. The algorithm requires a full scan of the whole database only if the negative border of the frequent itemsets expands, that is, if an itemset outside the negative border gets added to the frequent itemsets or its negative border. Even in such cases, it requires only one I/O pass over the whole data set.

Constrained association mining is another useful technique for goal-oriented mining. We generalize the incremental mining algorithm to handle cases with contstraints on the items and certain kinds of constraint relaxation.

In Section 7.1, we present the incremental algorithm to update the frequent itemsets. Experimental results quantifying the performance advantages of the incremental algorithm are presented in Section 7.2 and in Section 7.3, we compare it with FUP. In Section 7.4, we explore the database integration alternatives of incremental mining and in Section 7.4.2

[1]Note that the negative border can be maintained while computing the frequent itemsets without any additional computation overhead.

we present the performance results of the SQL-based incremental algorithm [124]. In Section 7.5, we introduce constrained associations and generalize the incremental approach to handle constraints. The incremental algorithm is applicable to a larger class of data mining and decision support problems. Some of them are briefly outlined in Section 7.6.

7.1 Incremental Updating of Frequent Itemsets

In this section, we develop an efficient algorithm for updating the frequent itemsets when the database is updated. In the context of *basket* data, database update effectively means addition of new transactions to the database or deletion of existing transactions. In Section 7.1.1, we describe how to compute the negative border of a set F of frequent itemsets. In Section 7.1.2, we present and prove some lemmas and theorems before describing the algorithm to handle the addition of transactions. Handling the deletion of transactions is outlined in Section 7.1.3.

7.1.1 Computing $\mathcal{NB}d(F)$ from F

The *negative border* ($\mathcal{NB}d(F)$) of a set of frequent itemsets F can be computed by repeating the join and prune steps of the *apriori-gen* function in the apriori algorithm [13]. This computation can be done using only the set of frequent itemsets F and the database need not be scanned.

Definition 1 *The* negative border $\mathcal{NB}d(F)$, *of a collection of itemsets F is defined as follows: Given a collection $F \subseteq \mathcal{P}(R)$ of sets, closed with respect to the set inclusion relation, the negative border $\mathcal{NB}d(F)$ of F consists of the minimal itemsets $X \subseteq R$ not in F [83].*

The *apriori-gen* function (described in Figure 7.1) takes as argument F_{k-1}, the set of all frequent $(k-1)$-itemsets. It returns the set of k-itemsets that are potentially frequent.

---

function apriori-gen(F_{k-1})

for each p and $q \in F_{k-1}$ *do*

if $p.item_1 = q.item_1, \ldots, p.item_{k-2} = q.item_{k-2}$ and $p.item_{k-1} < q.item_{k-1}$

then insert $p.item_1, p.item_2, \ldots, p.item_{k-1}, q.item_{k-1}$ into C_k

for each $c \in C_k$

delete c from C_k if some $(k-1)$-subset of c is not in F_{k-1}

Figure 7.1. A high-level description of the apriori-gen function

The negative border consists of all itemsets that were candidates which did not have the minimum support in the level-wise method. That is, $\mathcal{NB}d(F_k) = C_k - F_k$ where C_k is the set of candidate k-itemsets, F_k is the set of frequent k-itemsets and $\mathcal{NB}d(F_k)$ is the set of k-itemsets in $\mathcal{NB}d(F)$. Therefore, $F_k \cup \mathcal{NB}d(F_k) = C_k$. The *negativeborder-gen* function to compute $\mathcal{NB}d(F)$ is explained in Figure 7.2.

---

function negativeborder-gen(F)

Split F into $F_1, F_2, \ldots, F_n$ where n is the size of the largest itemset in F

forall $k = 1, 2, \ldots, n$ *do*

compute C_{k+1} using *apriori-gen*(F_k)

$F \cup \mathcal{NB}d(F) = \bigcup_{i=2,\ldots,n+1} C_k \cup I_1$ where I_1 is the set of 1-itemsets.

Figure 7.2. A high-level description of the negativeborder-gen function

Lemma 1 *All 1-itemsets should be present in* $F \cup \mathcal{NB}d(F)$.

Proof: The lemma follows directly from the definition of negative border since any 1-itemset not in F is a minimal itemset not in F. □

7.1.2 Addition of New Transactions

When new transactions are added to the database, an old frequent itemset could potentially become infrequent in the updated database. Similarly, an old infrequent itemset could potentially become frequent in the new database.

In order to solve the update problem efficiently, we maintain the frequent itemset and the negative border along with their support count in the database. That is, for every $s \in F \cup \mathcal{NB}d(F)$, we maintain $s.count$. In the rest of this section, DB denotes the original database, db denotes the transactions that are newly added and $DB+$ denotes the updated database. Also F^{DB}, F^{db} and F^{DB+} denotes the frequent itemset and $\mathcal{NB}d(F^{DB})$, $\mathcal{NB}d(F^{db})$ and $\mathcal{NB}d(F^{DB+})$ denotes the negative border of the original database, increment database and the updated database respectively.

Lemma 2 *Let s be any itemset such that $s \notin F^{DB}$. Then $s \in F^{DB+}$ only if $s \in F^{db}$.*

Proof: Assume that there exists an itemset s such that $s \in F^{DB+}$, $s \notin F^{DB}$ and $s \notin F^{db}$. Let $t_{DB}(s)$ and $t_{db}(s)$ be the number of transactions in DB and db respectively containing the itemset s. Also let t_{DB} and t_{db} be the total number of transactions in DB and db respectively. Since $s \notin F^{DB}$ and $s \notin F^{db}$,

$$\frac{t_{DB}(s)}{t_{DB}} < minSupport \ \ and \ \frac{t_{db}(s)}{t_{db}} < minSupport.$$

From these two equations, it can be shown that

$$\frac{t_{DB}(s) + t_{db}(s)}{t_{DB} + t_{db}} < minSupport$$

Therefore, $s \notin F^{DB+}$ which is a contradiction. □

Lemma 3 *Let s be an itemset such that $s \in \mathcal{NB}d(F)$. Then all possible subsets of s must be present in F.*

Proof: For a contradiction, let t be an itemset such that $t \subset s$ and $t \notin F$. By the definition of negative border, $\mathcal{NB}d(F)$ consists of the *minimal* itemsets not in F. Since $t \notin F$, s is not a minimal itemset not in F. Therefore s cannot be in $\mathcal{NB}d(F)$, which is a contradiction. □

Theorem 1 *Let s be an itemset such that $s \notin F^{DB} \cup \mathcal{NB}d(F^{DB})$ and $s \in F^{DB+}$. Then there exists an itemset t such that $t \subset s$, $t \in \mathcal{NB}d(F^{DB})$ and $t \in F^{DB+}$. That is, some subset of s has moved from $\mathcal{NB}d(F^{DB})$ to F^{DB+}.*

Proof: Since $s \in F^{DB+}$, all possible subsets of s should be in F^{DB+}. But all the subsets of s cannot be in F^{DB} because if that was the case, then s should be present in at least $\mathcal{NB}d(F^{DB})$ if not in F^{DB} itself. By our assumption, $s \notin F^{DB} \cup \mathcal{NB}d(F^{DB})$. Therefore, there exists an itemset t such that $t \subset s$ and $t \notin F^{DB}$. Now we have two cases.

Case *i* **:** $t \in \mathcal{NB}d(F^{DB})$.

In this case, $t \in F^{DB+}$ since $s \in F^{DB+}$ and $t \subset s$. Therefore, we have found a subset of s which has moved from $\mathcal{NB}d(F^{DB})$ to F^{DB+}.

Case *ii* **:** $t \notin \mathcal{NB}d(F^{DB})$.

That is, $t \notin F^{DB} \cup \mathcal{NB}d(F^{DB})$. But, we know that $t \in F^{DB+}$ since $s \in F^{DB+}$ and $t \subset s$. Therefore, $t \notin F^{DB} \cup \mathcal{NB}d(F^{DB})$ and $t \in F^{DB+}$ and hence we can apply the theorem recursively on t. Note that the size of t is less than the size of s since $t \subset s$.

When this is applied recursively, there are two possibilities. First is, for some subset of t, case i holds true in which case, there is a subset of t which has moved from $\mathcal{NB}d(F^{DB})$ to F^{DB+}, and hence the theorem is proved. Otherwise, t will finally become a 1-itemset. By Lemma 1, we know that all 1-itemsets are present in $F^{DB} \cup \mathcal{NB}d(F^{DB})$. Since $t \notin F^{DB}$, $t \in \mathcal{NB}d(F^{DB})$ which contradicts the assumption for case ii. That is, case ii is not possible if t is a 1-itemset. □

By Theorem 1, if none of the itemsets move from the negative border to the frequent itemset, we do not need to scan the whole database. Even in cases where some itemsets move from the negative border to the frequent itemset, a complete database scan is required only if the negative border expands because, for all the itemsets in the negative border, we can derive the updated support count without a database scan.

We maintain the support count for all itemsets in the frequent itemset and the negative border. First, we compute the frequent itemset in *db* using a level-wise algorithm like Apriori or Partition. Simultaneously we count the support for all itemsets in $F^{DB} \cup \mathcal{NB}d(F^{DB})$ in *db*. If an itemset $t \in F^{DB}$ does not have minimum support in $DB \cup db$, then t is removed from F^{DB}. This can be easily checked since we know the support count for t in DB and *db*. The change in F^{DB} could potentially change $\mathcal{NB}d(F^{DB})$ also. Therefore, we have to recompute the negative border using the *negativeborder-gen* function explained in subsection 7.1.1.

On the other hand there could be some new itemsets which become frequent in the updated database. Let s be an itemset which gets added to the frequent itemset of the updated database. By Lemma 2, we know that s has to be in F^{db}. We also know by Theorem 1 that some subset of s must move from $\mathcal{NB}d(F^{DB})$ to F^{DB+}. For each itemset $s \in F^{db}$, we check if s gets the minimum support to move from $\mathcal{NB}d(F^{DB})$ to F^{DB+}. If

function Update-Frequent-Itemset(F^{DB}, $\mathcal{NB}d(F^{DB})$, *db*)

//DB and *db* denote the number of transactions in the original database and the increment database respectively.

Compute F^{db}

for each itemset $s \in F^{DB} \cup \mathcal{NB}d(F^{DB})$ *do*

$t_{db}(s)$ = number of transactions in *db* containing s

$F^{DB+} = \phi$

for each itemset $s \in F^{DB}$ *do*

if $(t_{DB}(s) + t_{db}(s)) \geq minsup * (DB + db)$ *then* $F^{DB+} = F^{DB+} \cup s$

for each itemset $s \in F^{db}$ *do*

if $s \notin F^{DB}$ and $s \in \mathcal{NB}d(F^{DB})$ and $(t_{DB}(s) + t_{db}(s)) \geq minsup * (DB + db)$ *then* $F^{DB+} = F^{DB+} \cup s$

if $F^{DB} \neq F^{DB+}$ *then*

$\mathcal{NB}d(F^{DB+})$ = negativeborder-gen(F^{DB+})

else $\mathcal{NB}d(F^{DB+}) = \mathcal{NB}d(F^{DB})$

if $F^{DB} \cup \mathcal{NB}d(F^{DB}) \neq F^{DB+} \cup \mathcal{NB}d(F^{DB+})$ *then*

$S = F^{DB+}$

repeat

compute $S = S \cup \mathcal{NB}d(S)$

until S does not grow

$F^{DB+} = \{x \in S | support(x) \geq minsup\}$

//$support(x)$ is the support count of x in $DB \cup db$

$\mathcal{NB}d(F^{DB+})$ = negativeborder-gen(F^{DB+})

Figure 7.3. A high-level description of the Update-Frequent-Itemset function

none of the itemsets in $\mathcal{NB}d(F^{DB})$ gets the minimum support, no new itemsets will be added to F^{DB+}. If some itemsets in $\mathcal{NB}d(F^{DB})$ gets the minimum support move them to F^{DB+} and recompute the negative border. If $F^{DB+} \cup \mathcal{NB}d(F^{DB+}) \neq F^{DB} \cup \mathcal{NB}d(F^{DB})$, we have to find the negative border closure of F^{DB+} and scan the entire database(DB) once to find the updated frequent itemset and negative border. The negative border closure of F is found by repeatedly finding $F = F \cup \mathcal{NB}d(F)$ until F does not grow.

During the database scan, all the itemsets which are in the negative border closure that were not originally in $F \cup \mathcal{NB}d(F)$ are used as the candidate itemsets and their support count is computed. The candidate set can further be pruned by applying an optimization while finding the negative border closure. It can be observed that an itemset which is not frequent in the increment database (db) cannot get added to the updated set of frequent itemsets. Therefore, such itemsets can be pruned at each step of the negative border closure computation to get the pruned negative border closure. However, the support count of these pruned itemsets should also be found since they may potentially be in the updated negative border.

7.1.3 Deletion of Existing Transactions

Similar to the case where new transactions are added to the database, the frequent itemset and its negative border could potentially change when some existing transactions are deleted from the database. As in the former case, we maintain the frequent itemset and the negative border along with their support count in the database. Let $DB-$ denote the updated database and F^{DB-} and $\mathcal{NB}d(F^{DB-})$ denote its frequent itemset and negative border respectively.

Lemma 4 *Let s be an itemset such that $s \in F^{DB}$. Then $s \notin F^{DB-}$ only if $s \in F^{db}$. That is a frequent itemset s will become infrequent only if $s \in F^{db}$.*

This lemma can be proved in the same way as lemma 2.

The algorithm to compute the frequent itemset and the negative border of $DB-$ is similar to the one in the case where new transactions are added to the database and can be derived easily.

7.2 Experimental Results

We conducted a set of experiments to compare the performance of our incremental algorithm. These experiments were performed on a Sun SPARCstation 4 running SunOS 5.5. In this section, we report on the results of some of those experiments.

The experiments were performed on synthetic data generated using the same technique as in Agrawal and Srikant [13]. The dataset used for the baseline experiment was T10.I4.D100K (Mean size of a transaction = 10, Mean size of maximal potentially large itemsets = 4, Number of transactions = 100 thousand). The increment database is created as follows: We generate 100 thousand transactions, of which $(100 - d)$ thousand is used for the initial computation and d thousand is used as the increment, where d is the fractional size (in percentage) of the increment.

We compare the execution time of the incremental algorithm with respect to running Apriori on the whole data set. Figure 7.4 shows the speed up of the incremental algorithm over Apriori for different *minimum support* thresholds. We report the results for increment sizes of $1\%, 2\%, 5\%$ and 10% (shown in the legend). From the graph, it can be seen that the incremental algorithm achieves speed up of about 3 to 20. The algorithm shows better speed up for medium support threshold than low and high support thresholds. At high

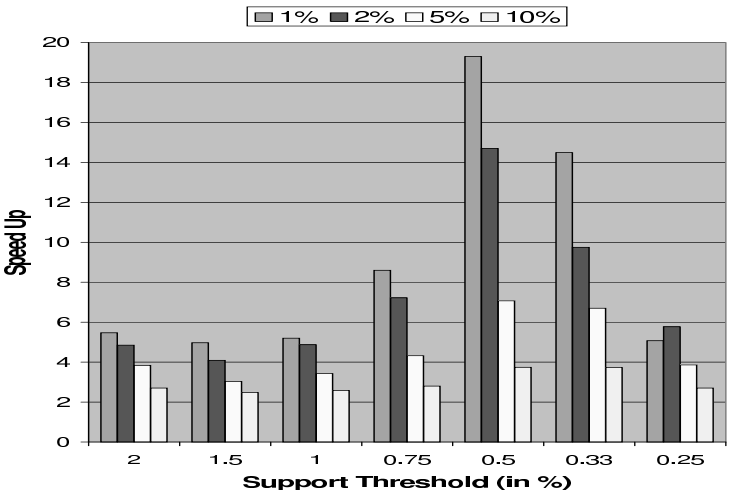


Figure 7.4. Speed-up of the incremental algorithm

support thresholds, the number of frequent itemsets and the number of passes in the level-wise algorithm are less and hence it is less costly to run Apriori on the whole database. At low support thresholds, the probability of the negative border expanding is higher and as a result the incremental algorithm may have to scan the whole database. Also, the speed up is higher for smaller increment sizes since the incremental algorithm needs to process less data.

7.3 Comparison with FUP

The framework of FUP [36] is similar to that of Apriori and contains a number of iterations. Each iteration is associated with a complete scan of the whole database and in iteration k all the frequent k-itemsets are found. The candidate sets for iteration $k+1$ are generated based on the frequent itemsets found in iteration k. FUP uses the following steps to compute the frequent itemsets.

1. At each iteration k, the support of the frequent k-itemsets (L_k) is updated against the increment database db to filter out the itemsets that are no longer frequent in the updated database.

2. The set of candidate sets C_k is generated by applying the *apriori-gen* function on L_{k-1}^{DB+} (frequent (k-1)-itemsets in the updated database). The itemsets in L_k are pruned from C_k because they have already been handled in the above step. C_k is further pruned by removing the itemsets that do not have minimum support in *db* since they cannot be frequent in the updated database.

3. Scan the whole database to count the support of all the itemsets in C_k and the ones which have the minimum support are identified as the new frequent k-itemsets.

The speed up of FUP over Apriori can be mainly attributed to the reduction in the number of candidate itemsets thereby reducing the computation. However there is no reduction in the I/O requirements because, like apriori FUP may also require $O(n)$ passes over the database where n is the size of the maximal frequent itemset.

The steps involved in our incremental algorithm can be summarized as follows.

1. Compute the frequent itemsets of the increment database (L^{db}) using any level-wise algorithm like Apriori or Partition. Simultaneously, count the support in *db* of all the frequent itemsets and the negative border.

2. Using the above support counts and the negativeborder-gen function, determine if the negative border has expanded.

3. In case of a negative border expansion, find the negative border closure and scan the whole database once to count the support of the newly added itemsets. The ones with minimum support are added to the set of frequent itemsets.

The notable feature here is that the whole database is scanned only if required (and that too only once), thereby reducing the I/O requirements drastically. Computing the

negative border closure may increase the size of the candidate set. However, a majority of those itemsets would have been present in the original negative border or frequent itemset. Only those itemsets which were not covered by the negative border need to be checked against the whole database. As a result, the size of the candidate set in the final scan could potentially be much smaller as compared to FUP.

7.4 Database Integration of Incremental Mining

In this section, we discuss the various database integration architectures for incremental mining based on the alternatives presented in Chapter 2. In Section 7.4.1, we present two SQL-based approaches for incremental frequent itemset computation. The applicability of the other architectural alternatives are discussed in Section 7.4.4.

7.4.1 SQL Formulations of Incremental Mining

Two categories of SQL formulations for frequent itemset mining based on SQL-92 and SQL-OR are presented in Chapters 3 and 4 respectively. A cost-based analysis of the SQL-92 approaches and the related performance optimizations are discussed in Thomas and Chakravarthy [125]. We discuss here how these techniques can be adapted for incremental mining. Efficient SQL formulation of the incremental algorithm entails counting support of multiple-sized candidates together in the same pass.

The input transaction data is stored in a relational table T with the schema (tid, item). The increment transaction table δT also has the same schema. The frequent itemsets and negative border of size k are stored in tables with the schema $(item_1, \ldots, item_k, count)$. We discuss the extensions to `Subquery` and `Vertical`; two representative approaches from the SQL-92 and SQL-OR categories. We also outline the SQL-based candidate closure computation to compute negative border closure.

Subquery Approach

In this approach, support counting is done by a set of k nested subqueries where k is the size of the largest itemset. We present here the extensions to the subquery approach in Section 3.5.4 to count candidates of different sizes. Subquery Q_l finds all tids that support the distinct candidate l-itemsets (d_l). d_l comprises of the distinct l-item prefixes of all candidate itemsets. However, it is sufficient to use C_l, the candidate l-itemsets instead of d_l since all l-item prefixes of candidate itemsets with more than l items will be present in C_l. The output of Q_l is grouped on the l items to find the support of the candidate l-itemsets. Q_l is also joined with δT (T while counting the support in the whole database) and C_{l+1} to get Q_{l+1}. The SQL queries and the corresponding tree diagrams for the above computations are given in Figure 7.5. δB_l stores the support counts of all frequent and negative border itemsets in δT.

The output of subquery Q_l needs to be materialized since it is used for counting the support of l-itemsets and for generating Q_{l+1}. If the query processor is augmented to support multiple streams where the output of an operator can be piped into more than one subsequent operators, the materialization of Q_l's can be avoided. In the basic association rule mining, we do not have to count itemsets of different sizes in the same pass since C_{l+1} becomes available only after the frequent l-itemsets are computed.

Tables B_l and δB_l store the frequent and negative border l-itemsets and their support count in T and δT respectively. Support counts of itemsets in the whole database can be computed by joining B_l and δB_l and adding the corresponding support counts. We add another attribute to B_l to keep track of promoted borders (itemsets that moved from the negative border to the frequent set).

insert into δB_l select $item_1, \ldots, item_l$, count(*)

from (**Subquery** Q_l) t

group by $item_1, \ldots, item_l$

Subquery Q_l (for any l between 1 and k):

select $item_1, \ldots item_l$, tid

from δT t_l, (**Subquery** Q_{l-1}) as r_{l-1}, C_l

where $r_{l-1}.item_1 = C_l.item_1$ and ... and

$r_{l-1}.item_{l-1} = C_l.item_{l-1}$and

$r_{l-1}.tid = t_l.tid$ and

$t_l.item = C_l.item_l$

Subquery Q_0: No subquery Q_0.

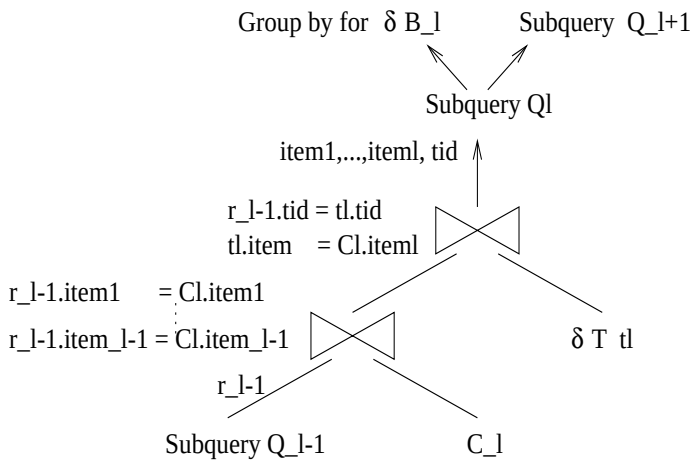


Figure 7.5. Support counting using subqueries

Computing Candidate Closure

In the apriori algorithm candidate itemsets are generated in two steps: the join step and the prune step. In the join step, two sets of $(k-1)$-itemsets called *generators* and *extenders* are joined to get k-itemsets. An itemset s_1 in generators joins with s_2 in extenders if the first $(k-2)$ items of s_1 and s_2 are the same and the last item of s_1 is lexicographically smaller than the last item of s_2. The join results in an itemset that is s_1 extended with the last item of s_2. The result of the join step is subjected to subset pruning which filters out all itemsets with a non frequent $(k-1)$-subset. This can be accomplished by subsequent joins with $(k-2)$ copies of a set of itemsets termed *filters*. While generating C_k from F_{k-1} in the basic apriori algorithm, F_{k-1} acts as generators, extenders and filters.

In the incremental algorithm, we compute the candidate closure to avoid multiple passes while counting the support of the new candidates. It can be seen that all the new candidates will be supersets of promoted borders. Therefore, it is sufficient to use the itemsets that are promoted borders as generators. In order to generate C_k, the candidates of size k, we use $PB_{k-1} \cup C_{k-1}$ as generators and $PB_{k-1} \cup C_{k-1} \cup F_{k-1}$ as extenders and filters. PB_{k-1} and F_{k-1} denote promoted borders and frequent itemsets of size $(k-1)$ respectively. The candidate generation process starts with C_0 as the empty set and terminates when C_k becomes empty. It is straight-forward to derive SQL queries for this process and we do not present them here (refer Section 3.4.1).

Vertical

In the `Subquery` approach, for every transaction that supports an itemset we generate (itemset, tid) tuples resulting in large intermediate tables. The `Vertical` approach avoids this by collecting all tids that support an itemset into a BLOB (binary large object) and

generates (itemset, tid-list) tuples. Initially, tid-lists for individual items are created using a table function. The tid-list for an itemset is obtained by intersecting the tid-lists of its items using a user-defined function (UDF). The SQL queries for support counting are similar in structure to that of the `Subquery` approach except for the use of UDFs to intersect the tid-lists. We refer the reader to Section 4.2 for the details.

The increment transaction table δT is transformed into the vertical format by creating the delta tid-lists of the items. The delta tid-lists are used to count the support of the candidate itemsets in δT which are later merged with the original tid-lists. This can be accomplished by joining the original tid-list table with the delta tid-list table and merging the tid-lists with a UDF. If the incremental algorithm requires a pass over the complete data, the merged tid-lists are used for support counting.

7.4.2 Performance Results

In this section, we report the results of some of our performance experiments to quantify the advantages of the incremental mining algorithm. Note that incremental mining can be considered as a special case of relaxing the frequency constraint. These experiments were performed on Version 5 of IBM DB2 Universal Server installed on a Sun Ultra 5 Model 270 with a 269 MHz UltraSPARC-IIi CPU, 128 MB main memory and a 4 GB disk. We report the results of the SQL formulations of the incremental algorithm based on the Subquery and Vertical approaches.

The experiments were performed on synthetic data generated using the technique in Agrawal and Srikant [13]. The dataset used for the experiment was T10.I4.D100K (Mean size of a transaction = 10, Mean size of maximal potentially large itemsets = 4, Number of transactions = 100 thousand). The increment database is created as follows: We generate

100 thousand transactions, of which $(100 - d)$ thousand is used for the initial computation and d thousand is used as the increment, where d is the fractional size (in percentage) of the increment.

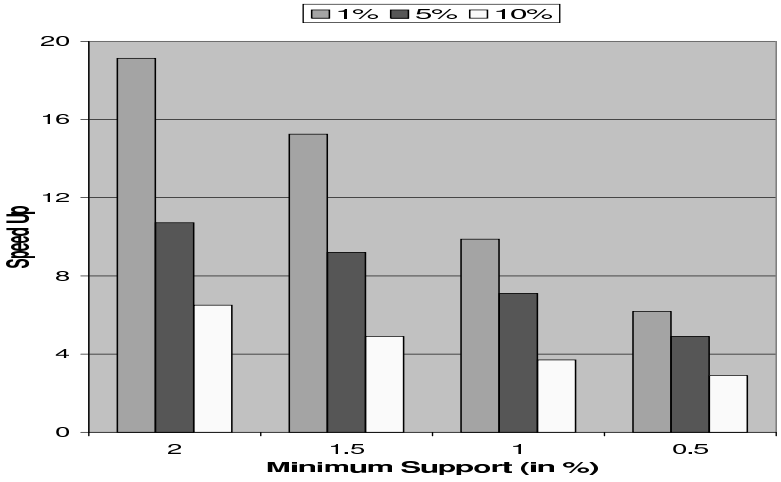


Figure 7.6. Speed up of the incremental algorithm based on the Subquery approach

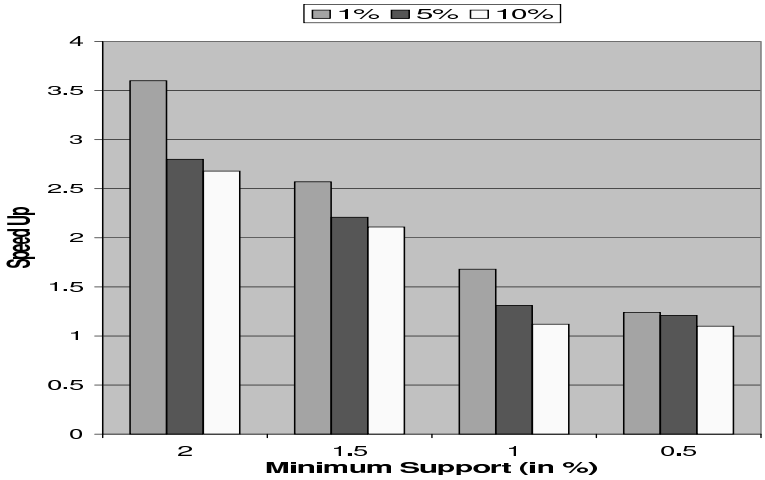


Figure 7.7. Speed up of the incremental algorithm based on the Vertical approach

We compare the execution time of the incremental algorithm with respect to mining the whole dataset. Figures 7.6 and 7.7 shows the corresponding speed-ups of the incremental algorithm based on the Subquery and the Vertical approaches for different minimum

support thresholds. We report the results for increment sizes of $1\%, 5\%$ and 10% (shown in the legend). We can make the following observations from the graphs.

- The incremental algorithm based on the Subquery approach achieves a speed-up of about 3 to 20 as compared to mining the whole dataset. However, the maximum speed-up of the Vertical approach is only about 4. For support counting the Vertical approach uses a user-defined function (UDF) to intersect the tid-lists. The incremental algorithm should also invoke the UDF at least the same number of times since the support of all the itemsets in the frequent set and the negative border needs to be found in the increment database. In cases where the support of new candidates needs to be counted the number of invocations will be even more as compared to mining the whole dataset. The time taken by the Vertical approach in the support counting phase is directly proportional to the number of times the UDF is called. However, the incremental algorithm saves in the tid-list creation phase since the size of the increment dataset is only a fraction of the whole dataset. This explains why the speed-up of the Vertical approach is low. In contrast the Subquery approach achieves higher speed-up since the time taken is proportional to the size of the dataset.

  It is possible to achieve better speed-up for the Vertical approach by allocating a smaller BLOB (binary large object) for computations involving the increment dataset. Note that the tid-lists for the items are stored as BLOBs. In our experiments, we used the same BLOB size for the increment dataset and the initial dataset in order to use the same user-defined function for support counting and the same table function for tid-list creation (refer Section 4.2 for a detailed description of the Vertical approach).

- The speed-up reduces as the minimum support threshold is lowered. At lower support values the chances of the negative border expanding is higher and as a result the incremental algorithm may have to compute the candidate closure and count the support of the new candidates in the whole dataset.

- The speed-up is higher for smaller increment sizes since the incremental algorithm needs to process less data.

- With respect to the absolute execution time, the Subquery and the Vertical approaches followed the same trend as explained in Chapter 4. The Vertical approach was about 3 to 6 times faster than the Subquery approach.

7.4.3 New-Candidate Optimization

In the basic incremental algorithm, we find the frequent itemsets in the increment database *db* along with counting the support of all the itemsets in the frequent set and the negative border. However, the frequent itemsets in *db* is used only to prune the non frequent itemsets in *db* while computing the candidate closure. In the candidate closure computation we assume that the new candidate k-itemsets are frequent while generating the $(k+1)$-itemsets. At this step the new candidate k-itemsets that are infrequent in *db* are known to be infrequent in the whole dataset as well and can be pruned. This is because the new candidate k-itemsets were infrequent in the old dataset (they were not even in the negative border). Therefore, they need to be frequent at least in *db* to have a chance of being frequent in the whole dataset.

With the new-candidate optimization, we count the support of an itemset in *db* only if it is required. In the first phase, while counting the support in *db* of the itemsets in the frequent set and the negative border, we do not find all the frequent itemsets in *db*. During

the candidate closure computation, we count the support in *db* of just the new-candidates for the pruning explained above. This results in better speed-up as compared to the basic incremental algorithm.

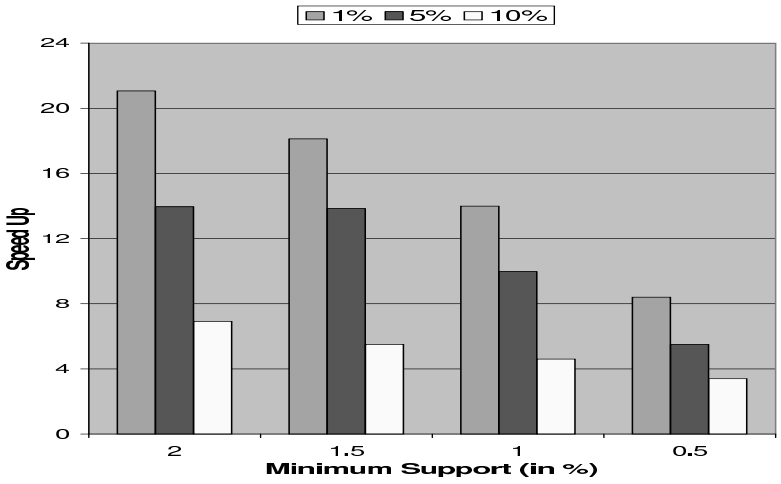


Figure 7.8. Speed up of the incremental algorithm based on the Subquery approach with the new-candidate optimization

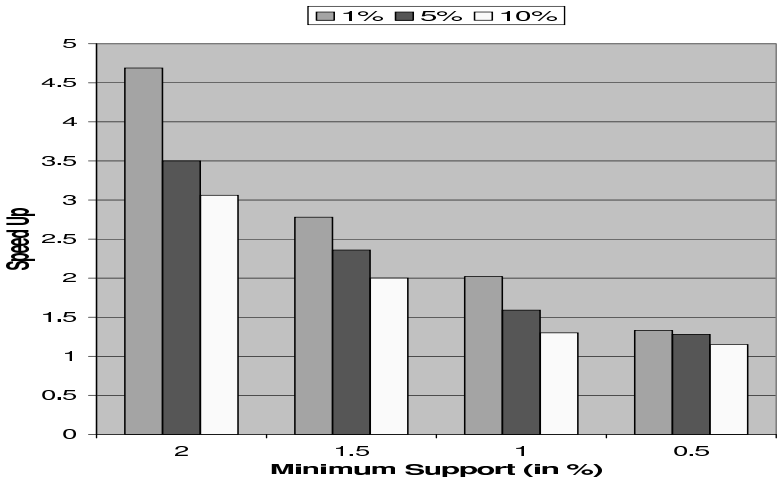


Figure 7.9. Speed up of the incremental algorithm based on the Vertical approach with the new-candidate optimization

Figures 7.8 and 7.9 show the speed-up of the Subquery and Vertical approaches with the new-candidate optimization. We can observe that this optimization achieves speed-ups that are up to 25% better. The improvement is more at smaller increment sizes.

The reason is that, when the increment is smaller we have to use proportionately smaller minimum support values while finding the frequent itemsets in *db*. This could result in counting too many spurious candidates.

7.4.4 Other Approaches

In the `Loose-coupling` approach, the transaction data is read tuple by tuple from the DBMS to the mining kernel using a cursor interface. This architecture can be extended to handle incremental mining just by implementing the incremental algorithm outlined in Section 7.1 in the mining kernel. The DBMS interface does not require any change. In cases where the support of new itemsets need to be counted, limiting the data access to just one scan of the whole database entails counting candidate itemsets of multiple sizes in the same pass. This can be accomplished by passing the transactions through all the candidates of different sizes and updating their support counts.

The `Stored-procedure` approach where the mining algorithm is encapsulated as a stored procedure that runs in the same address space as the DBMS and the `Cache-Mine` approach where the data is cached outside the DBMS, can also be extended for incremental mining. However, the `Cache-Mine` approach might not give better performance than the others since the incremental algorithm requires at most one scan of the entire data. Extending the `UDF` approach for incremental mining is straight-forward and will involve writing UDFs for the different steps of the incremental algorithm.

7.5 Constrained Associations

In this section, we introduce associations with different kinds of constraints on the itemsets or constraints that characterize the dataset from which the associations are derived. Let $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ be a set of literals, called items that are attribute values

of a set of relational tables. A constrained association is defined as a set of itemsets $\{X|X \subseteq \mathcal{I} \,\&\, C(X)\}$, where C denotes one or more boolean constraints. Note that we do not concentrate on generating the association rules in the traditional sense [7]. However, the associations between attribute values that we generate can be used for rule generation.

7.5.1 Categories of Constraints

We divide the constraints into four different categories that are outlined below[2] We illustrate each of them with sample mining computations. The data model used in our examples is that of a point-of-sale (POS) model for a retail chain. When a customer buys a product or series of products at a register, that information is stored in a transactional system, which is likely to hold other information such as who made the purchase and what types of promotions were involved. The data is stored in three relational tables *sales, products_sold* and *product* with respective schemas as shown in Figure 7.10.

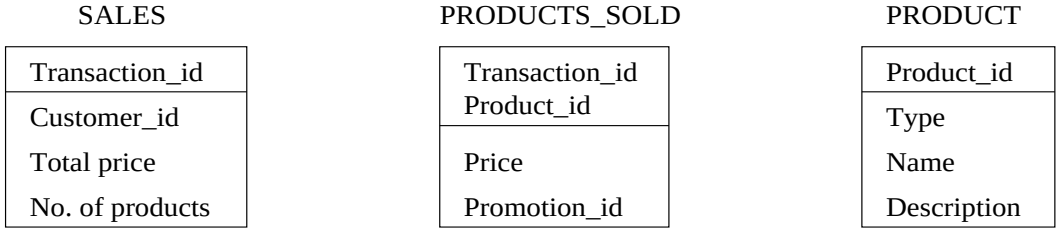


Figure 7.10. Point of sales data model

Frequency Constraint

This is the same as the minimum support threshold in the "support-confidence" framework for association rule mining [13]. An itemset X is said to be frequent if it appears in at least s transactions, where s is the minimum support. In the point-of-sales data model a

[2] We refer the reader to Srikant et al. [121] and Ng et al. [97] for nice discussions of various kinds of constraints. Here we categorize them based on their usage in the mining process which is explained in Section 7.5.2 with an example

transaction correspond to a customer transaction. Note that in other domains, the notion of a transaction and an itemset appearing in a transaction could be different. Frequent itemsets, the ones which satisfy the frequency constraint are defined as $\{X|f(X) \geq s\}$ where $f(X)$ is the frequency of X.

Most of the algorithms for frequent itemset discovery utilizes the downward closure property of itemsets with respect to the frequency constraint; that is, if an itemset is frequent, then so are all its subsets. Downward closure is a pruning property. Level-wise algorithms [7] find all itemsets with a given property among itemsets of size k (k-itemsets) and use this knowledge to explore $(k+1)$-itemsets. They start with the assumption that all $(k+1)$-itemsets are potentially frequent (frequency is just an example for a downward closed property). As the k-itemsets are examined, they prune out some $(k+1)$-itemsets that cannot be frequent. In effect, for pruning, they use the contrapositive of the frequent itemset definition "if any subset of a $(k+1)$-itemset is not frequent, then neither can the $(k+1)$-itemset". After the pruning, they go through the remaining list, checking each $(k+1)$-itemset for its frequency. The downward closure property is similar to the anti-monotonicity property defined in Ng et al. [97].

In the context of the point-of-sale data model, the frequency constraint can be used to discover products bought together frequently.

Item Constraint

In order to discover goal-oriented associations, it is often required to impose constraints on the itemsets. For instance, find only the itemsets containing at least one item from a user-defined subset of items. Let $\mathcal{B}$ be a boolean expression over the set of items $\mathcal{I}$. The problem is to find itemsets that satisfy the constraint $\mathcal{B}$. Typically the item constraint will

be associated with a frequency constraint also, where we want to find frequent itemsets that satisfy $\mathcal{B}$. Three different algorithms for mining associations with item constraints are presented in Srikant et al. [121].

The item constraints enables us to pose mining queries such as "What are the products whose sales are affected by the sale of, say, barbecue sauce?" and "What products are bought together with sodas and snacks?". The 1-variable constraints in Ng et al. [97] are a special case of item constraints.

Aggregation Constraint

These are constraints involving aggregate functions on the items that form the itemset. For instance, in the POS example an aggregation constraint could be of the form $min(products\_sold.price) \geq p$. Here we consider a product as an item. The aggregate function could be *min, max, sum, count, avg* or any other user-defined aggregate function. An aggregation constraint of the form $min(products\_sold.price) \geq p$ can be used to find "expensive products that are bought together". Similarly $max(products\_sold.price) \leq q$ can be used to find "inexpensive products that are bought together". These aggregate functions can be combined in various ways to express a whole range of useful mining computations. For example, the constraint $(min(products\_sold.price) \leq p) \,\&\, (avg(products\_sold.price) \geq q)$ targets the mining process to inexpensive products that are bought together with the expensive ones.

External Constraint

External constraints filter the data used in the mining process. These are constraints on attributes which do not appear in the final result (which we call external attributes). For example, if we want to find "products bought during big purchases where the total

sale price of the transaction is larger than some amount" we can impose a constraint of the form $sales.total\,price \geq P$. These constraints are useful to target the mining process to just the relevant data there by speeding up the process.

7.5.2 Constrained Association Mining

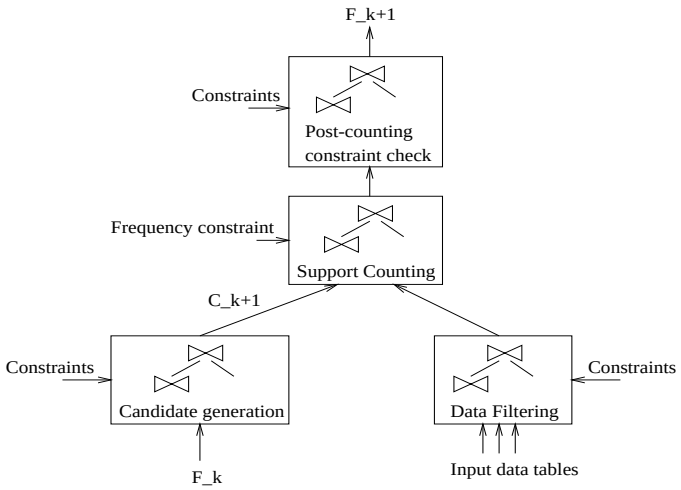


Figure 7.11. Framework for constrained association mining

Figure 7.11 shows the general framework for the k^{th} level in a level-wise approach for mining associations with constraints. Note that we could also use SQL-based approaches for implementing the various operations in the different phases. The different constraints are applied at different steps in the mining process. For example, the item constraints and the aggregation constraints that satisfy the closure property can be used in the candidate generation phase for pruning unwanted candidates. Three different algorithms for generating candidate itemsets in the presence of item constraints are presented in Srikant et al. [121] and we could use them here. The aggregation constraints can be applied on the result of the candidate generation process with item constraints. The external constraints and some of the aggregation and item constraints can be applied at the data filtering stage to reduce the size of the input data to the support counting phase (see Figure 7.12 for a specific

example). The frequency constraint is applied during support counting to filter out the non-frequent itemsets. Finally, any unprocessed constraints are checked as a post-counting operation. The frequent itemsets before the post-counting constraint check are used for the next level candidate generation since these constraints do not possess the closure property.

In the basic frequent itemset mining, only the frequency constraint is present and the candidate generation step uses only the subset pruning strategy [13].

Example: We illustrate the application of the various constraints here with a specific example using the point-of-sales data model in Section 7.5. The example shown in Figure 7.12 finds product combinations containing "barbecue sauce" where all the products cost less than $50 and the average price is more than $25 (some notion of similar priced products). The combinations should appear in at least 100 sales transactions with the total price of the transaction greater than $500. This gives an idea of what other moderately priced products people buy with barbecue sauce in big purchases (perhaps for parties). It could help the shop owner to decide on various promotions.

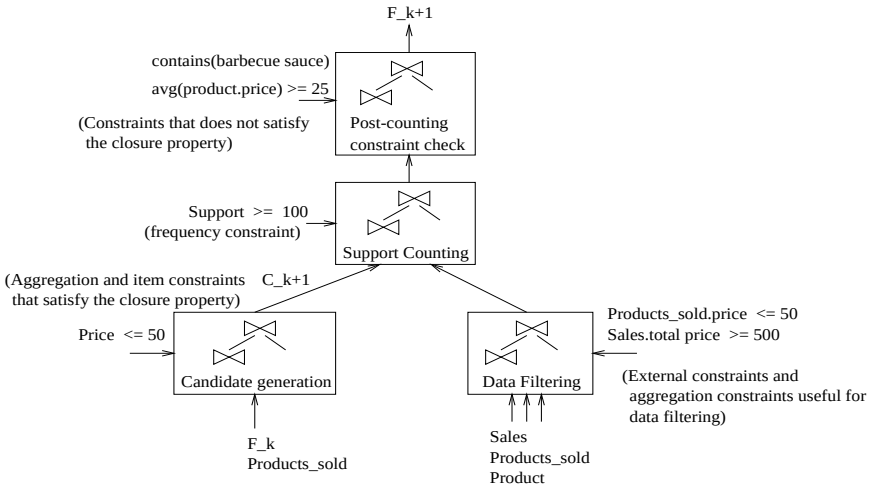


Figure 7.12. Point-of-sales example for constrained association mining

In the above example, the constraint on the total price of the transaction is an external constraint and is applied in the data filtering stage. Since the maximum price of a product in the desired combination is \$50, we can also filter out records that does not satisfy this condition in the data filtering stage. $max(Price) \leq 50$ is an aggregation constraint which satisfies the closure property and can be applied in the candidate generation phase. The constraint to include barbecue sauce in the combination and the constraint on the average price are checked in the post-counting phase since they do not satisfy the closure property.

7.5.3 Incremental Constrained Association Mining

The negative border based incremental mining algorithm is applicable for mining associations with constraints that are closed with respect to the set inclusion property, that is, if an itemset satisfies the constraint then so do all its subsets. For incremental mining of constrained associations also we need to materialize and store all the itemsets in the negative border and their support counts. The reason why this is enough is that when new transaction data is added, only the support count of the itemsets could change. We assume that there is a frequency constraint, which is typically the case in association mining. The definition of the negative border also remains the same, which is the set of minimal itemsets that did not satisfy the frequency constraint. We list the various steps of the incremental algorithm which is very similar to the case with just the frequency constraint except for a few differences.

1. Find the frequent itemsets in the increment database *db*. Simultaneously count the support of all itemsets $X \in FrequentSets \cup NegativeBorder$ in *db*. *For this we use the framework outlined in Section 7.5.2. The different constraints that are present can be handled at the various steps in the mining process as shown in Figure 7.11.*

2. Update the support count of all itemsets in $FrequentSets \cup NegativeBorder$ to include their support in *db*.

3. Find the itemsets in NegativeBorder that became frequent by the addition of *db* (call them Promoted-NegativeBorder, PNb).

4. Find the candidate closure with PNb as the seed. During this computation, we use the candidate generation procedure with constraints as described in Section 7.5.2.

5. If there are no new itemsets in the candidate closure, skip this step. Otherwise, count the support of all new itemsets in the candidate closure against the whole dataset. *The dataset is subjected to the data filtering step before the support counting as shown in Figure 7.11.*

7.5.4 Constraint Relaxation

The negative border idea can be used to handle some cases of constraint relaxation also, especially relaxations to the external constraints and the frequency constraint. For instance, in the example in Section 7.5.2, if we relax the constraint on the sales transaction to $sales.Total\_price \geq 300$, it can be cast as an incremental mining problem. In this case, the increment dataset would be the transactions with *Total_price* between 300 and 500. Note that, for this to work the relaxed constraint should subsume the initial constraint. Relaxations to the frequency constraint involves two cases. In cases where the frequency threshold is increased, it is straight-forward to find the new frequent itemsets by just filtering the itemsets with the new threshold. On the other hand, if the frequency threshold is lowered we need to use an approach similar to the incremental mining algorithm where we compute the promoted negative borders and candidate closure to determine if the support of any new itemset needs to be found.

7.6 Applicability Beyond Association Mining

The incremental algorithm we have proposed makes use of the closure property of frequent itemsets. In this section, we discuss how the incremental approach can be generalized to other data mining and decision support problems. In Section 7.6.1, we discuss the applicability of the incremental algorithm to mine *closed sets*. Generalizations to incremental evaluation of query flocks and certain kinds of materialized views are described in Sections 7.6.2 and 7.6.3 respectively.

7.6.1 Mining Closed Sets

All the efficient algorithms for mining association rules exploits the closure property of frequent itemsets. Minimum support which characterizes frequent itemsets is downward closed: if an itemset has minimum support then all its subsets also have minimum support. The idea of negative border can be used for all incremental mining problems that possess closure properties. If the closure property is incrementally updatable (for instance support) also, it is possible to limit the database access to at most one scan of the whole database as shown in the incremental frequent itemset mining example. A property is incrementally updatable if it is possible to derive the value of it from the corresponding values of different partitions of the input data. A few examples are `COUNT`, `SUM`, `MIN`, `MAX` and so on. We list below a few other data mining problems that have closure properties.

1. Sequential patterns: In sequential pattern mining, the frequent patterns are closed with respect to minimum support, much like the frequent itemsets. The support is incrementally updatable.

2. Correlation rules: Correlation rules [25] are upward closed with respect to *correlation*; that is, if a set of items S is correlated, so is every superset of S. A set of items is

said to be correlated if any of its subsets are dependent. For efficient evaluation of correlation rules, a notion of *support* is introduced in Brin et al. [25], which is downward closed. However, these properties are not incrementally updatable.

3. Consequent part of association rules: For any frequent itemset, if a rule with consequent c holds (that is, it has minimum confidence) then so do rules with consequents that are subsets of c [9]; that is, such rules are downward closed. Note that the confidence is not incrementally updatable.

4. Maximal frequent itemsets: Random walk algorithms [58] which works by generating a series of random walks along the itemset lattice exploits the downward closure property of maximal frequent itemsets. Support which characterizes frequent itemsets in this case is also incrementally updatable.

7.6.2 Query Flocks

The boolean association rules was first defined in the context of market basket data and it has been generalized to mine associations across relational tables expressed as query flocks [128]. A query flock is a parameterized query with a filter condition to eliminate the values of parameters that are "uninteresting". For example, let us assume that we want to evaluate the mining query "What are the interesting drivers that caused customers to buy the widgets from a catalog ?". A driver is deemed "interesting" if it has caused at least 10 customers to buy the widget. Let the data be stored in a set of relational tables, namely, catalog(widget, manufacturer), sale(customer, widget), driver(customer, widget, driver). The above query can be written as a query flock in Datalog as shown below. The filter condition prunes out values which do not have minimum support. In Section 7.6.2,

we discuss how the negative border idea can be used for efficient incremental evaluation of query flocks.

QUERY:

```
answer(C) :-
        sale(C, $W) AND
        driver(C, $W, $D) AND
        catalog($W, manufacturer)
```

FILTER:

```
COUNT(C) > 10
```

Incremental Evaluation of Query Flocks

Applying the apriori technique for evaluating the above query flock will result in the following safe subqueries [128].

1. Q_1: `answer(C) :- sale(C, $W)`
2. Q_2: `answer(C) :- driver(C, $W, $D)`
3. Q_3: `answer(C) :- sale(C, $W) AND driver(C, $W, $D)`
4. Q_4: `answer(C) :- sale(C, $W) AND driver(C, $W, $D) AND catalog($W, manufacturer)`

The query flocks corresponding to the safe subqueries form a lattice with query containment as the partial order and the original query flock as the top element. That is, if Q_1 and Q_2 are elements of the lattice, $Q_1 \preceq Q_2 \Leftrightarrow$ the result of Q_2 is contained in the result of Q_1.

During the execution of the subqueries of the query flock, all records with parameter values which satisfy the filter condition are propagated to the next higher subquery in the

lattice for further evaluation. For example, after Q_1 is evaluated all the records corresponding to widgets that are bought by at least 10 customers will be piped to Q_3 which is immediately above Q_1 in the subquery lattice. The parameter value combinations in this case are analogous to itemsets in boolean association rule mining.

The negative border in this apriori-based query flock evaluation is the set of parameter value combinations that does not pass the filter condition test. These combinations if materialized and stored along with their support counts can be effectively used to update the result of the query flock when the base tables over which it is defined are updated. We first check if any of the records in the negative border pass the filter condition as a result of changes to the base tables. This can be done starting at the bottom element of the lattice and proceeding upwards, propagating the deltas corresponding to the new combinations. If none of the records in the negative border passes the filter condition, we do not have to evaluate the subqueries for the lattice elements above. The base tables may have to be looked up (for example, to be joined with the deltas) depending upon the subquery representing the lattice element. It is also possible to evaluate the filter condition for all the lattice elements' negative borders together. However, this might involve more computations than propagating the deltas through the lattice.

7.6.3 View Maintenance

Incremental mining can also be seen as materialized view maintenance. In boolean association rules, the frequent itemsets and the negative border are in fact aggregate views over the transaction table. In query flocks each element in the subquery lattice can be considered as a view defined on the base tables. Therefore, view maintenance techniques similar to the ones in Mumick et al. [94] can be used for incremental mining. On the other

hand, the negative border based change propagation can also be applied for the maintenance of views involving monotone aggregate functions that satisfy the apriori subset property. For example if the view definition has a filter condition such as the SQL "having" clause, it could be beneficial to also store the records which does not pass the filter condition test (same as the negative border). When the base tables are updated, these records will be useful to maintain the view rather than re-materializing it.

An itemset can also be treated as a point in the data cube [55] defined by the items as dimensions and support as the measure. The data cube points can be arranged as a lattice according to the partial order on the itemsets. In that case, the data cube maintenance algorithms similar to that of Roussopoulos et al. [107] are also applicable here. However, this approach might not be viable in cases consisting of large number of items.

7.7 Summary

In this chapter, we developed an incremental algorithm for mining frequent itemsets. We developed SQL formulations for the incremental approach based on a few representative approaches from Chapters 3 and 4. We also show how the incremental algorithm can be generalized to handle certain kinds of constraints and constraint relaxations and its applicability to other mining problems.

CHAPTER 8
CONCLUSIONS

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

We considered two categories of SQL implementations. We experimented with four different implementations based purely on SQL-92. We picked the best SQL-92 implementation for further analysis. We developed cost formulae for different execution plans based on the input data parameters and the relational operator costs. This cost analysis provides a basis for incorporating the semantics of mining algorithms into future query optimizers. We next experimented with a collection of approaches that made use of the new object-relational extensions like UDFs, BLOBs, and table functions. With this extended SQL we got significant performance improvements over the SQL-92 based implementations.

We compared the SQL implementation with different architectural alternatives. We observed that based just on performance the `Cache-Mine` approach is the winner. A close second is the SQL-OR approach that was sometimes slightly better than `Cache-Mine` and

was never worse than a factor of two on the real-life datasets. Both these approaches require additional space for caching, however. The `Stored-procedure` approach does not require any extra space (except possibly for initially sorting the data in the DBMS) and can perhaps be made to be within a factor of two to three of `Cache-Mine` with the recent algorithms [26, 127]. The `UDF` approach is about a factor of two faster than the `Stored-procedure` approach but is significantly harder to code. The SQL approach offers some secondary advantages like easier development and maintenance and potential for automatic parallelization. However, it might not be as portable as the `Cache-Mine` approach across different database management systems.

Next, we wanted to find out if it is possible to handle other more complex mining tasks within the same SQL framework. We studied generalized association rules and sequential patterns with the goal of showing that the association rule framework can be extended easily for these mining problems as well. We developed several SQL formulations for generalized association rule and sequential pattern mining; some of them by extending the association rule approaches. The major addition for generalized association rule was to "extend" the input transaction table (transform T to T^*). For sequential patterns, the join predicates for candidate generation and support counting were significantly different. We conducted some experiments on real-life datasets and found that the performance observations hold here also.

In order to handle the volume of data stored in present day data warehouses, which keeps streaming in, it is important to develop incremental mining algorithms. We developed an incremental association rule mining algorithm which does not need to examine the old data if the frequent itemsets do not change. Even in cases where new frequent itemsets are added, access to the old database can be limited to just one scan.

In the context of goal-oriented mining, we often have to mine associations with various kinds of constraints. We identify and categorize the different constraints and show how they can be processed in the relational framework. The incremental approach can also be used to handle certain kinds of constraint relaxation. We develop a general framework for the incremental approach to mine associations with constraints having the downward closure property.

The concept of negative border which is the key to the incremental algorithm has other applications also. It can be used for mining association rules with varying support and confidence values. For instance, the negative border can be used to determine the updated frequent itemsets if the support is changed. If the support is increased it is trivial to update the frequent itemsets. However, if the support is lowered the itemsets in the promoted negative border can be used to determine if the support of any new itemset needs to be counted in a similar manner to incremental mining. This could be quite useful in cases where determining the "correct" support value is difficult. Initially the frequent itemsets for an approximate support can be computed which is further refined based on user feedback.

We further show the applicability of the incremental algorithm to certain classes of data mining and decision support problems.

In Section 8.1, we identify certain extensions to the current database management systems that are useful for mining. We enumerate the specific contributions of this dissertation in Section 8.2 and in Section 8.3, we list possibilities for further research.

8.1 Proposed Extensions

One of the goals of our work has been to "unbundle" complex mining operations like associations and classifications into smaller primitives that can be supported efficiently by a general purpose DBMS and be useful for a large class of mining algorithms. Our exercise with some of the mining operations has helped us identify a few such primitives that we believe could be generally useful in a number of other decision support applications.

8.1.1 Richer **Set** Operations:

We expect a richer collection of set operations to be useful for mining. For the mining problems addressed in this dissertation, we used three different versions of the basic **subset** operation. For candidate generation we needed to find all $k-1$ subsets of a set of k elements for doing the pruning. We enumerated the subsets explicitly in the query predicate because the size of the set and the subsets was *fixed* and known in advance. For support counting, we used the Comb-K table function which generated all k subsets of a *variable* length set. For rule generation, we used the GenSubset table function to generate *all* subsets of a variable length set. The subset operation would also be useful in building decision trees on categorical attributes [85].

Another important set operation was the **intersect** operation used in the `Vertical` approach. Most systems already have internal implementations of this operation for doing AND and OR operations on RID (record identifier) lists obtained during multiple index scans [93]. In current OLAP and data warehousing systems this operation is rampant in the popular bit-mapped indices. Our `Vertical` approach is curiously similar to this bit-mapped approach with one important difference. Instead of performing ANDs on RIDs, we perform the ANDs on another attribute, the transaction identifier.

We also used the *set difference* operation for pruning itemsets containing an item and its ancestor, in generalized association rules.

8.1.2 Enhanced Aggregation

Another common operation that we used is the **Gather** operation that can transform two attributes in a data table to a form where for distinct values of one of the attributes (called the grouping attribute) we collect together in a set all values of the other attribute. We can think of `Gather` as a glorified aggregate function. Its reverse operation **scatter** is a special case of the subset operation where the subset size is 1.

We used this operation in four of our SQL approaches: `GatherJoin`, `GatherPrune`, `Vertical` and `Horizontal`. To implement the `Gather` table function in our queries we required data to appear in a particular order as inputs to the table function. This would be trivial to express in SQL if order by clauses were allowed in inner subqueries. In the absence of such a feature, we relied on our knowledge of DB2's internals to force such an order in our experiments. The `Gather` function can also be treated as an aggregate function that simply concatenates its arguments. Systems that have support for user-defined aggregate functions [122] can easily support such a functionality.

There are several other decision support applications where `gather` could be extremely useful. For instance, suppose we are trying to *classify* customers of a credit card company and the data, in addition to static customer attributes like age and salary, consists of detailed transaction data about each purchase activity of a customer. In this case, we would like to *gather* all activities of a customer as one time series and extract features from these time series for classification. In OLAP applications too, data often needs to be converted back and forth from a format where different measures are gathered together as

different attributes of the same row to a format where different measures are scattered as different rows.

8.1.3 Multiple Streams

In current relational DBMSs, there is only a single stream of control along which data flows. Support for multiple streams, where the output of an operation can be piped to more than one subsequent operations will be useful for a large class of mining and decision support operations. In the SQL formulation of incremental mining, we show how it will be useful for counting the support of multiple-sized candidates in one pass. This idea can be used for association rule mining also to reduce the number of passes utilizing some of the newer algorithms [26, 127].

In classification, we have to compute the class distribution corresponding to different split points and attribute value combinations. This will require multiple scans using standard SQL. The *Unpivot* operator proposed in Graefe et al. [54] attempts to circumvent the problem of multiple scans. However, support for multiple streams will make it much simpler for doing it in one scan of the database. In the OLAP world also this will be useful for computing the datacube [56] and simultaneous multidimensional aggregates [1, 133].

8.1.4 Sampling

Sampling is a useful technique for scaling up algorithms that handle large volumes of data. The use of sampling for query size estimation, histogram construction, computing quantiles and so on has been addressed in several research papers [35, 59].

Sampling can be used to get quick and good approximate answers to the mining and decision support queries. The algorithms can be first run on a sample to give the user approximate answers based on which he/she can decide whether to run it on the whole

dataset and also to fine tune input parameters. There are several mining algorithms which use sampling [44, 127].

8.2 Contributions [1]

In this dissertation, we have addressed the following problems.

1. Analyze the different database integration alternatives for data mining.
2. Develop and implement various SQL-based approaches for association rule mining.
3. Study the performance profile of current DBMSs to execute the above SQL queries.
4. Compare the different database integration architectures quantitatively and qualitatively.
5. Develop cost formulae for the SQL approaches based on input data parameters and relational operator costs. These provide some insights into enhancing current cost-based optimizers to incorporate the domain-specific semantics of mining algorithms.
6. Extend the association rule framework for mining generalized association rules and sequential patterns.
7. Develop an incremental association rule mining algorithm.
8. SQL formulations of the incremental algorithm and its generalization to handle constraints.
9. Generalize the incremental algorithm for constrained association mining and demonstrate its applicability to a larger class of data mining and decision support problems.

[1] The work corresponding to the first four items was primarily done by researchers from IBM Almaden Research Center and the author was a contributor. For the remaining items, the author was the primary contributor. The file-based incremental association rule mining algorithm (item 7) was developed primarily by the author as part of the Introduction to Parallel Computing course project.

10. Explore primitive operators to support mining and decision support in databases.

8.3 Future Work

The work presented in this dissertation points to several directions for future research. A natural next step is to experiment with other kinds of mining operations such as classification and clustering [45], to verify if our conclusions hold for these other cases too. It is also important to derive a set of primitive operators with which the different mining and decision support operations can be composed. The operators we have identified provide some headway in that direction. Another useful direction is to explore what kind of a support is needed for answering short, interactive, *ad hoc* queries involving a mix of mining and relational operations. The success of SQL as the most popular data management language can be mainly attributed to its *ad hoc* query support. In the mining context, what is an *ad hoc* mining query? Is it just mining computations expressed with some constraints on the result? How much support can we leverage from existing relational engines for mining? What data model and language extensions are needed? Does the execution of these operators involve selective materialization and incremental evaluation techniques? Some of these questions are orthogonal to whether the bulky mining operations are implemented using SQL or not. Nevertheless, these are important in providing the analysts with a well integrated platform where mining and relational operations can be inter-mixed in useful and flexible ways.

8.4 Closing

In the coming years, database/data warehouse systems will be called upon to deliver return-on-investment in the form of nuggets of knowledge or better insights about the huge volumes of data they store and manage. In order to efficiently support the multiplicity of

data mining and decision support operations on the same data, tighter integration of these operations with database systems will be required.

This dissertation presents strategies aimed at tighter database integration of mining and identifies optimizations and primitive operators to make database systems a better platform for mining and decision support.

REFERENCES

[1] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proc. of the 22nd Int'l Conference on Very Large Databases*, pages 506–521, Mumbai (Bombay), India, September 1996.

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

[3] R. Agrawal, C. Faloutsos, and A. Swami. Efficient similarity search in sequence databases. In *Proc. of the Fourth Int'l Conference on Foundations of Data Organization and Algorithms*, Chicago, October 1993. Also in Lecture Notes in Computer Science 730, Springer Verlag, 1993, 69–84.

[4] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *Proc. of the ACM SIGMOD Conference on Management of Data*, Seattle, Washington, June 1998.

[5] R. Agrawal, S. Ghosh, T. Imielinski, B. Iyer, and A. Swami. An interval classifier for database mining applications. In *Proc. of the VLDB Conference*, pages 560–573, Vancouver, British Columbia, Canada, August 1992.

[6] R. Agrawal, T. Imielinski, and A. Swami. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):914–925, December 1993.

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

[8] R. Agrawal, K. I. Lin, H. S. Sawhney, and K. Shim. Fast similarity search in the presence of noise, scaling, and translation in time-series databases. In *Proc. of the 21st Int'l Conference on Very Large Databases*, pages 490–501, Zurich, Switzerland, September 1995.

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

[10] R. Agrawal, G. Psaila, E. L. Wimmers, and M. Zaït. Querying shapes of histories. In *Proc. of the 21st Int'l Conference on Very Large Databases*, Zurich, Switzerland, September 1995.

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

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

[13] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th Int'l Conference on Very Large Databases*, Santiago, Chile, September 1994.

[14] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. of the 11th Int'l Conference on Data Engineering*, Taipei, Taiwan, March 1995.

[15] S. Alexander. Users find tangible rewards digging into data mines. Enterprise Computing, July 1997. http://www.infoworld.com/cgi-bin/displayArchive.pl?/97/27/e01-27.61.htm.

[16] K. Ali, S. Manganaris, and R. Srikant. Partial classification using association rules. In *Proc. of the 3rd Int'l Conference on Knowledge Discovery in Databases and Data Mining*, Newport Beach, California, August 1997.

[17] K. Alsabti, S. Ranka, and V. Singh. CLOUDS: A decision tree classifier for large datasets. In *Proc. of the 4th Int'l Conference on Knowledge Discovery and Data Mining*, New York, August 1998.

[18] C. Apte and S. J. Hong. Predicting equity returns from securities data with minimal rule generation. In *KDD-94: AAAI Workshop on Knowledge Discovery in Databases*, pages 407–418, Seattle, Washington, July 1994.

[19] C. Apte and S. Weiss. Data mining with decision trees and decision rules. *FGCS Journal, Special Issue on Data Mining*, 1997.

[20] J. Banfield and A. Raftery. Model-based gaussian and non-gaussian clustering. *Biometrics*, 49:15–34, 1993.

[21] S. Berchtold, C. Bohm, B. Braunmuller, D. A. Keim, and H. Kriegel. Fast parallel similarity search in multimedia databases. In *Proc. of the ACM SIGMOD Conference on Management of Data*, May 1997.

[22] S. Berchtold, D. A. Keim, and H. Kriegel. The X-Tree: An index structure for high-dimensional data. In *Proc. of the 22nd Int'l Conference on Very Large Databases*, Bombay, India, September 1996.

[23] P. Bradley, U. Fayyad, and C. Reina. Scaling clustering algorithms to large databases. In *Proc. of the 4th Int'l Conference on Knowledge Discovery and Data Mining*, New York, August 1998.

[24] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees.* Wadsworth, Belmont, 1984.

[25] S. Brin, R. Motwani, and C. Silverstein. Beyond market baskets: Generalizing association rules to correlations. In *Proc. of the ACM SIGMOD Conference on Management of Data*, May 1997.

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

[27] Business Week. *Database marketing*, September 1994.

[28] J. Catlett. *Megainduction: Machine Learning on Very Large Databases*. PhD thesis, University of Sydney, 1991.

[29] J. Catlett. Overpruning large decision trees. In *12th Int'l Joint Conference on AI*, August 1991.

[30] S. Chakrabarti, B. Dom, R. Agrawal, and P. Raghavan. Using taxonomy, discriminants, and signatures for navigating in text databases. In *Proc. of the VLDB Conference*, Athens, Greece, August 1997.

[31] S. Chakrabarti, B. Dom, and P. Indyk. Enhanced hypertext categorization using hyperlinks. In *Proc. of the ACM SIGMOD Conference on Management of Data*, Seattle, Washington, June 1998.

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

[33] S. Chaudhuri. Data mining and database systems: Where is the intersection? *IEEE Data Engineering Bulletin*, 21(1):4–8, March 1998.

[34] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *ACM SIGMOD Record*, 26(1):65–74, March 1997.

[35] S. Chaudhuri, R. Motwani, and V. Narasayya. Random sampling for histogram construction: How much is enough? In *Proc. of the ACM SIGMOD Conference on Management of Data*, Seattle, Washington, June 1998.

[36] D. Cheung, J. Han, V. Ng, and C. Y. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. In *Proc. of 1996 Int'l Conference on Data Engineering*, New Orleans, USA, February 1996.

[37] G. Cooper and E. Herskovits. A Bayesian method for the induction of probabilistic networks from data. *Machine Learning*, 1992.

[38] David Shepard Associates, Business One Irwin, Illinois. *The new direct marketing*, 1990.

[39] D. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 13(2), 1987.

[40] T. G. Dietterich and R. S. Michalski. Discovering patterns in sequences of events. *Artificial Intelligence*, 25:187–232, 1985.

[41] Direct Marketing Association. *Managing database marketing technology for success*, 1992.

[42] M. Ester, H. Kriegel, and X. Xu. A database interface for clustering in large spatial databases. In *Proc. of the 1st Int'l Conference on Knowledge Discovery in Databases and Data Mining*, Montreal, Canada, August 1995.

[43] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *Proc. of the ACM SIGMOD Conference on Management of Data*, May 1994.

[44] U. Fayyad, C. Reina, and P. Bradley. Initialization of iterative refinement clustering algorithms. In *Proc. of the 4th Int'l Conference on Knowledge Discovery and Data Mining*, New York, August 1998.

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

[46] U. Fayyad, N. Weir, and S. G. Djorgovski. Skicat: A machine learning system for automated cataloging of large scale sky surveys. In *10th Int'l Conference on Machine Learning*, June 1993.

[47] R. Feldman, Y. Aumann, A. Amir, and H. Mannila. Efficient algorithms for discovering frequent sets in incremental databases. In *Proceedings of the 1997 SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery, Tucson, Arizona*, May 1997.

[48] D. H. Fisher. Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, 2(2), 1987.

[49] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Data mining using two-dimensional optimized association rules: Scheme, algorithms, and visualization. In *Proc. of the ACM SIGMOD Conference on Management of Data*, Montreal, Canada, June 1996.

[50] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Mining optimized association rules for numeric attributes. In *Proc. of the 15th ACM Symposium on Principles of Database Systems*, Montreal, Canada, June 1996.

[51] K. Fukunaga. *Introduction to Statistical Pattern Recognition*. Academic Press, 1990.

[52] Gartner Group. *Data Mining: The Next Generation of Business Intelligence?*, ATG research Note T-517-246, Gartner Group Inc., Stamform, CT edition, March 1994.

[53] J. Gehrke, R. Ramakrishnan, and V. Ganti. Rainforest - a framework for fast decision tree construction of large datasets. In *Proc. of the VLDB Conference*, New York, August 1998.

[54] G. Graefe, U. Fayyad, and S. Chaudhuri. On the efficient gathering of sufficient statistics for classification from large SQL databases. In *Proc. of the 4th Int'l Conference on Knowledge Discovery and Data Mining*, New York, August 1998.

[55] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *Proc. of 1996 Int'l Conference on Data Engineering*, New Orleans, USA, February 1996.

[56] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[57] S. Guha, R. Rastogi, and K. Shim. CURE: An efficient clustering algorithm for large databases. In *Proc. of the ACM SIGMOD Conference on Management of Data*, Seattle, Washington, June 1998.

[58] D. Gunopulos, H. Mannila, and S. Saluja. Discovering all most specific sentences by randomized algorithms. In *Proc. of the sixth International Conference on Database Theory*, January 1997.

[59] P. J. Haas, J. F. Naughton, S. Seshadri, and L. Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *Proceedings of the Eighth International Conference on Very Large Databases (VLDB)*, pages 311–22, Zurich, Switzerland, September 1995.

[60] T. Hagerup and C. Rüb. A guided tour of Chernoff bounds. *Information Processing Letters*, 33:305–308, 1989/90.

[61] E. H. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. In *Proc. of the ACM SIGMOD Conference on Management of Data*, Tucson, Arizona, May 1997.

[62] J. Han, Y. Cai, and N. Cercone. Knowledge discovery in databases: An attribute oriented approach. In *Proc. of the VLDB Conference*, pages 547–559, Vancouver, British Columbia, Canada, 1992.

[63] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *Proc. of the 21st Int'l Conference on Very Large Databases*, Zurich, Switzerland, September 1995.

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

[65] S. J. Hong. R-MINI: A heuristic algorithm for generating minimal rules from examples. In *3rd Pacific Rim Int'l Conference on AI*, August 1994.

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

[67] T. Imielinski. From file mining to database mining. In *Proc. of the 1996 SIGMOD workshop on research issues on data mining and knowledge discovery*, pages 35–39, May 1996.

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

[69] T. Imielinski, A. Virmani, and A. Abdulghani. Discovery board application programming interface and query language for database mining. In *Proc. of the 2nd Int'l Conference on Knowledge Discovery and Data Mining*, Portland, Oregon, August 1996.

[70] International Business Machines. *White paper on data mining*, Technical report, 1995.

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

[72] H. V. Jagadish. A retrieval technique for similar shapes. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 208–217, Denver, May 1994.

[73] H. V. Jagadish and C. Faloutsos. Data reduction – a tutorial. Fourth international conference on Knowledge Discovery and Data Mining, tutorial, August 1998.

[74] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A. I. Verkamo. Finding interesting rules from large sets of discovered association rules. In *Third Int'l Conference on Information and Knowledge Management*, pages 401–407, Gaithersburg, Maryland, November 1994. ACM Press.

[75] F. Korn, H. V. Jagadish, and C. Faloutsos. Efficiently supporting ad hoc queries in large datasets of time sequences. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 289–300, 1997.

[76] F. Korn, A. Labrinidis, Y. Kotidis, and C. Faloutsos. Ratio rules: A new paradigm for fast, quantifiable data mining. In *Proc. of the VLDB Conference*, New York, August 1998.

[77] R. Krishnamurthy and T. Imielinski. Practitioner problems in need of database research: Research directions in knowledge discovery. *SIGMOD RECORD*, 20(3):76–78, September 1991.

[78] K. Kulkarni. Object oriented extensions in SQL3: a status report. *SIGMOD RECORD*, 1994.

[79] K. I. Lin, H. V. Jagadish, and C. Faloutsos. The TV-Tree: An index structure for high-dimensional data. *VLDB Journal*, 3(4):517–542, 1994.

[80] G. Lohman, B. Lindsay, H. Pirahesh, and K. B. Schiefer. Extensions to starburst: Objects, types, functions, and rules. *Communications of the ACM*, 34(10), October 1991.

[81] D. J. Lubinsky. Discovery from databases: A review of AI and statistical techniques. In *IJCAI-89 Workshop on Knowledge Discovery in Databases*, pages 204–218, Detroit, August 1989.

[82] J. A. Major and C. Feng. EFD: A hybrid knowledge/statistical-based system for the detection of fraud. *Int'l Journal of Intelligent Systems*, 7(7), 1992.

[83] H. Mannila and H. Toivonen. On an algorithm for finding all interesting sentences. In *Cybernetics and Systems, Volume II, The 13th European Meeting o n Cybernetics and Systems Research, Vienna, Austria*, April 1996.

[84] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovering frequent episodes in sequences. In *Proc. of the 1st Int'l Conference on Knowledge Discovery in Databases and Data Mining*, Montreal, Canada, August 1995.

[85] M. Mehta, R. Agrawal, and J. Rissanen. SLIQ: A fast scalable classifier for data mining. In *Proc. of the Fifth Int'l Conference on Extending Database Technology (EDBT)*, Avignon, France, March 1996.

[86] M. Mehta, J. Rissanen, and R. Agrawal. MDL-based decision tree pruning. In *Proc. of the 1st Int'l Conference on Knowledge Discovery in Databases and Data Mining*, Montreal, Canada, August 1995.

[87] J. Melton and N. Mattos. SQL3—a tutorial. Twenty-second international conference on Very large data bases, tutorial, September 1996.

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

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

[90] R. S. Michalski. A theory and methodology of inductive learning. In Michalski et al., editors, *Machine Learning, An Artificial Intelligence Approach, Vol. 1.* Morgan Kaufmann, 1983.

[91] D. Michie, D. J. Spiegelhalter, and C. C. Taylor. *Machine Learning, Neural and Statistical Classification.* Ellis Horwood, 1994.

[92] R. J. Miller and Y. Yang. Association rules over interval data. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 452–461, May 1997.

[93] C. Mohan, D. Haderle, Y. Wang, and J. Cheng. Single table access using multiple indexes: optimization, execution, and concurrency control techniques. In *Proc. International Conference on Extending Database Technology*, pages 29–43, 1990.

[94] I. Mumick, D. Quass, and B. Mumick. Maintenance of data cubes and summary tables in a warehouse. In *Proc. of the ACM SIGMOD Conference on Management of Data*, Tucson, Arizona, May 1997.

[95] R. Musick, J. Catlett, and S. Russell. Decision-theoretic subsampling for induction on large datsets. In *9th Int'l Conference on Machine Learning*, 1993.

[96] J. P. Nearhos, M. J. Rothman, and M. S. Viveros. Applying data mining techniques to a health insurance information system. In *Proc. of the 22nd Int'l Conference on Very Large Databases*, Mumbai (Bombay), India, September 1996.

[97] R. T. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained association rules. In *Proc. of the ACM SIGMOD Conference on Management of Data*, Seattle, Washington, June 1998.

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

[99] J. S. Park, M. S. Chen, and P. S. Yu. An effective hash based algorithm for mining association rules. In *Proc. of the ACM-SIGMOD Conference on Management of Data*, San Jose, California, May 1995.

[100] PostgreSQL Organization. *PostgreSQL 6.3 User Manual*, February 1998. `http://www.postgresql.org`.

[101] Quest: IBM develops market basket analysis system. *Stores*, May 1994.

[102] J. R. Quinlan. Induction over large databases. Technical Report STAN-CS-739, Stanford University, 1979.

[103] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufman, 1993.

[104] J. R. Quinlan and R. L. Rivest. Inferring decision trees using minimum description length principle. *Information and Computation*, 1989.

[105] K. Rajamani, B. Iyer, and A. Chaddha. Using DB2's object-relational extensions for mining association rules. Technical Report TR 03-690, Santa Teresa Laboratory, IBM Corporation, September 1997.

[106] R. Rastogi and K. Shim. PUBLIC: A decision tree classifier that integrates building and pruning. In *Proc. of the VLDB Conference*, New York, August 1998.

[107] N. Roussopoulos, Y. Kotidis, and M. Roussopoulos. Cube tree: Organization of and bulk incremental updates on the data cube. In *Proc. of the ACM SIGMOD Conference on Management of Data*, Tucson, Arizona, May 1997.

[108] M. A. Roytberg. A search for common patterns in many sequences. *Computer Applications in the Biosciences*, 8(1):57–64, 1992.

[109] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. In *Proc. of the ACM SIGMOD Conference on Management of Data*, Seattle, Washington, June 1998.

[110] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. Research Report RJ 10107 (91923), IBM Almaden Research Center, San Jose, California, March 1998. (Longer version of the SIGMOD 98 paper).

[111] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. of the VLDB Conference*, Zurich, Switzerland, September 1995.

[112] S. Z. Selim and M. A. Ismail. K-means-type algorithms: A generalized convergence theorem and characterization of local optimality. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-6(1), 1984.

[113] J. Shafer and R. Agrawal. Parallel algorithms for high-dimensional similarity joins for data mining applications. In *Proc. of the VLDB Conference*, Athens, Greece, August 1997.

[114] J. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *Proc. of the 22nd Int'l Conference on Very Large Databases*, Bombay, India, September 1996.

[115] K. Shim, R. Srikant, and R. Agrawal. High-dimensional similarity joins. In *Proc. of the 13th Int'l Conference on Data Engineering*, Birmingham, U.K., April 1997.

[116] A. Siebes and M. L. Kersten. KESO: Minimizing database interaction. In *Proc. of the 3rd Int'l Conference on Knowledge Discovery and Data Mining*, Newport Beach, California, August 1997.

[117] C. Silverstein, S. Brin, R. Motwani, and J. Ullman. Scalable techniques for mining causal structures. In *Proc. of the VLDB Conference*, New York, August 1998.

[118] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proc. of the 21st Int'l Conference on Very Large Databases*, Zurich, Switzerland, September 1995.

[119] R. Srikant and R. Agrawal. Mining quantitative association rules in large relational tables. In *Proc. of the ACM SIGMOD Conference on Management of Data*, Montreal, Canada, June 1996.

[120] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proc. of the Fifth Int'l Conference on Extending Database Technology (EDBT)*, Avignon, France, March 1996.

[121] R. Srikant, Q. Vu, and R. Agrawal. Mining association rules with item constraints. In *Proc. of the 3rd Int'l Conference on Knowledge Discovery in Databases and Data Mining*, Newport Beach, California, August 1997.

[122] M. R. Stonebraker and G. Kemnitz. The POSTGRES next generation database management system. *Communications of the ACM*, 34(10), 1991.

[123] S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka. An efficient algorithm for the incremental updation of association rules in large databases. In *Proc. of the 3rd Int'l Conference on Knowledge Discovery and Data Mining*, Newport Beach, California, August 1997.

[124] S. Thomas and S. Chakravarthy. Incremental mining of constrained associations. Technical Report TR 98-018, University of Florida, Gainesville, Florida, October 1998.

[125] S. Thomas and S. Chakravarthy. Performance evaluation and optimization of join queries for association rule mining. Technical Report TR 98-017, University of Florida, Gainesville, Florida, October 1998.

[126] S. Thomas and S. Sarawagi. Mining generalized association rules and sequential patterns using sql queries. In *Proc. of the 4th Int'l Conference on Knowledge Discovery and Data Mining*, New York, August 1998.

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

[128] D. Tsur, J. Ullman, S. Abiteboul, C. Clifton, R. Motwani, S. Nestorov, and A. Rosenthal. Query Flocks: A generalization of association rule mining. In *Proc. of the ACM SIGMOD Conference on Management of Data*, Seattle, Washington, June 1998.

[129] S. M. Weiss and C. A. Kulikowski. *Computer Systems that Learn: Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems*. Morgan Kaufman, 1991.

[130] R. B. Yates and G. H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, October 1992.

[131] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *Proc. of the 3rd Int'l Conference on Knowledge Discovery and Data Mining*, Newport Beach, California, August 1997.

[132] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: An efficient data clustering method for very large databases. In *Proc. of the ACM SIGMOD Conference on Management of Data*, Montreal, Canada, June 1996.

[133] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proc. of the ACM SIGMOD Conference on Management of Data*, Tucson, Arizona, May 1997.

BIOGRAPHICAL SKETCH

Shiby Thomas was born on May 30, 1971, in Kothamangalam, Kerala, India. He received his Bachelor of Technology degree in computer engineering from Regional Engineering College, Calicut, India, in May 1992. After his graduation he worked for a year as a systems engineer at Wipro Systems, Bangalore, India. He received his Master of Technology degree in computer science and engineering from Indian Institute of Technology, Bombay, India, in January 1995.

He joined the Department of Computer and Information Science and Engineering at the University of Florida in fall, 1995. He has worked as a teaching assistant in the Computer and Information Science and Engineering Department of the University and as a research assistant in the Database Systems Research and Development Center of the department. In Summer 1997, he worked as a summer intern in the Quest data mining group at the IBM Almaden Research Center, San Jose, California. He will receive his Doctor of Philosophy degree in December 1998.

His research interests include data mining and decision support technologies, real-time databases and transaction processing.