Indexing and Data Access Methods for Database Mining

Ganesh Ramesh, William A. Maniatty*
Dept. of Computer Science
University at Albany
Albany, NY 12222
{ganesh,maniatty}@cs.albany.edu

Mohammed J. Zaki
Computer Science Dept.
Rensselaer Polytechnic Institute
Troy, NY 12180
zaki@cs.rpi.edu

Abstract

Most of today's techniques for data mining and association rule mining (ARM) in particular, are really "flat file mining", since the database is typically dumped to an intermediate flat file that is input to the mining software. Previous research in integrating ARM with databases mainly looked at exploiting language (SQL) as a tool for implementing mining algorithms. In this paper we explore an alternative approach, using various data access methods and systems programming techniques to study the efficiency of mining data.

We present a systematic comparison of the performance of horizontal (*Apriori*) and vertical (*Eclat*) ARM approaches utilizing flat-file and a range of indexed database approaches. Measurements of run time as a function of database and minimum support threshold are analyzed. Experimental profiling measures of the frequency and cost of various operations are discussed. This analysis motivated both the use of adaptive ARM techniques and the development of a simple yet novel linked block structure to support efficient transaction pruning. We also explore techniques for determining what kinds of data benefit from pruning, and when pruning is likely to help.

Keywords: Indexing, Data Access, Performance Analysis, Association Rules

*Please addresss all correspondence to Maniatty, the contact author.

1 Introduction

Even after almost a decade of data mining, most of today's techniques can be more appropriately termed as "file mining", since typically, there is little interaction between the mining engine and the database. Most existing mining "systems" are either stand-alone or loosely-coupled with the database management system, the typical interface being through cursors to extract the data from the database. There are a number of problems with decoupled or loosely-coupled approaches: 1) In a decoupled system a user would have to spend a lot of time to extract and format the data in a form that can be mined; different mining tools may have different input format. 2) Exploratory querying can be extremely hard, since the onus of managing, naming, and storing multiple relevant target subsets of the database rest squarely on the analyst. 3) The choice of mining and other pre/post-processing operations are limited. The mining output in terms of rules or more complex models are not able to be queried, since they are stored outside the database (in flat-files).

Techniques are clearly needed to bridge this gap between "file-mining" and "database-mining". Database research, over the last 30 years, has produced constructs and indexing structures that can and should be leveraged in solving data mining tasks, which can really be considered as more advanced forms of database queries. The ultimate goal would be to support ad hoc data mining queries, focusing on increasing programmer productivity for mining applications [15]. There are two complementary approaches to integrating mining with database systems. One is to enhance or propose new language constructs that allow efficient mining query specification and optimization. The other approach is to adapt or develop new indexing and data access methods for efficient query execution.

Previous research in integrating mining and databases has mainly looked at the language support. DMQL [12] is a mining query language designed to support the wide spectrum of common mining tasks. It consists of specifications of four main primitives, which include the subset of data relevant to the mining query, the type of task to be performed, the background knowledge, and constraints or "interestingness" measures. MSQL [16] is an extension of SQL to generate and selectively retrieve sets of rules from a large database. Data and rules are treated uniformly, allowing various optimizations to be performed; one could manipulate generated rules or one could perform selective, query-based rule generation. The MINE RULE SQL operator [20] extends the semantics of association rules, allowing more generalized queries to be performed. Query flocks [27] uses a generate-and-test model of data mining; it extends the *Apriori* [1] technique of association mining to solve more general mining queries. In [2], a tight-coupling of association mining with the database was studied. It uses user-defined functions to push parts of the computation inside the database system.

A comprehensive study of several architectural alternatives for database and mining integration were studied in [24], in the context of association mining; these alternatives include: 1) loose-coupling through a SQL cursor, 2) encapsulating the mining algorithm in a stored-procedure, 3) caching the data on a file-system on-the-fly and then mining it, 4) using user-defined functions for mining, and 5) SQL implementations. They studied four approaches using SQL-92 and another six in SQL-OR (SQL with object-relational extensions). They concluded experimentally that the Cache-Mine approach, which is an enhanced version of the flat-file *Apriori* method, is clearly superior, while SQL-OR approaches come within a factor of two. The SQL-92 approaches were not competitive with the alternatives.

Besides associations, there has been limited work in integrating other mining tasks with databases. A middleware for classification was proposed in [6]; it decomposes and schedules classification primitives over a back-end SQL database. Two generic SQL operations called count-by-group (for class histograms) and compute-tuple-distances (for point distances) were identified in [10] for classification and clustering tasks, respectively.

In this paper, we study the other, almost neglected, axis in mining and database system integration, i.e., efficient indexing and data access support to realize efficient query execution. We consider association rule mining (ARM) as a concrete example to illustrate our techniques. Understanding the design trade-offs of the required structures is impossible without a thorough performance analysis of existing ARM methods. Although making informed design decisions is difficult due to complex trade-offs between data organization, choice of algorithm, and data access methods, it is somewhat surprising that little to no performance analysis has been done for ARM methods. One such work by Dunkel and Soparkar [8] studied the performance and I/O cost of the traditional *row-wise* (or horizontal) implementation of *Apriori* versus a *column-wise* (or vertical) implementation. They found via simulations that the column-wise approach significantly reduces the number of disk accesses.

A systematic comparison of horizontal (*Apriori* [1]) and vertical (*Eclat* [28, 29]) approaches utilizing flat-file and a range of indexed database approaches is presented. This analysis motivates a low-overhead novel linked block file structure to support pruning of transactions and items for horizontal approaches, as seen in Section 4.3.2. One important design goal was to provide the implemented algorithms flexibility in the selection of their data access methods. This goal was satisfied by designing and implementing the middleware layer discussed in Section 4.2. An analysis of the impact of each scheme on storage costs (see Section 5.3) and performance (see Section 5.4) was done by measuring the sensitivity of these parameters to data distribution for two synthetic databases and three real databases. In order to more fully understand the results, performance of various stages of ARM computation was measured using profilers, with results presented in Section 5.6. By this study, we help identify the requirements of indexed data layouts for data mining, and provide guidance for making informed design decisions.

2 ARM: Problem Statement

Association mining works as follows. Let $\mathcal{I}$ be a set of items, and $\mathcal{T}$ a database of transactions, where each transaction has a unique identifier (*tid*) and contains a set of items called an itemset. An itemset with k items is called a k-itemset. The support of an itemset X, denoted $\sigma(X)$, is the number of transactions in which that itemset occurs as a subset. A k-subset is a k-length subset of an itemset. An itemset is frequent or large if its support is more than a user-specified minimum support (*min_sup*) value. F_k is the set of frequent k-itemsets. A frequent itemset is maximal if it is not a subset of any other frequent itemset.

An association rule is an expression $X \xrightarrow{s,c} Y$, where X and Y are itemsets. The rule's support s is the joint probability of a transaction containing both X and Y, and is given as $s = \sigma(XY)$. The confidence c of the rule is the conditional probability that a transaction contains Y, given that it contains X, and is given as $c = \sigma(XY)/\sigma(Y)$. A rule is frequent if its support is greater than *min_sup*, and strong if its confidence is more than a user-specified minimum confidence (*min_conf*).

Association Rule mining involves generating all association rules in the database that have a support greater than *min_sup* (the rules are frequent) and that have a confidence greater than *min_conf* (the rules are strong). The main step in this process is to find all frequent itemsets having minimum support. The search space for enumeration of all frequent itemsets is 2^m, which is exponential in $m = |\mathcal{I}|$, the number of items. However, if we assume the transaction length has a bound, we can show that ARM is essentially linear in database size [28].

ORIGINAL DATABASE

Transaction	Items
1	A B D E
2	B C E
3	A B D E
4	A B C E
5	A B C D E
6	B C D

VERTICAL DATABASE

A	B	C	D	E
1	1	2	1	1
3	2	4	3	2
4	3	5	5	3
5	4	6	6	4
	5			5
	6			

ALL FREQUENT ITEMSETS

MINIMUM SUPPORT = 50%

Support	Itemsets
100% (6)	B
83% (5)	E, BE
67% (4)	A, C, D, AB, AE BC, BD, ABE
50% (3)	AD, CE, DE, ABD, ADE BCE, BDE, ABDE

Figure 1: Mining Frequent Itemsets

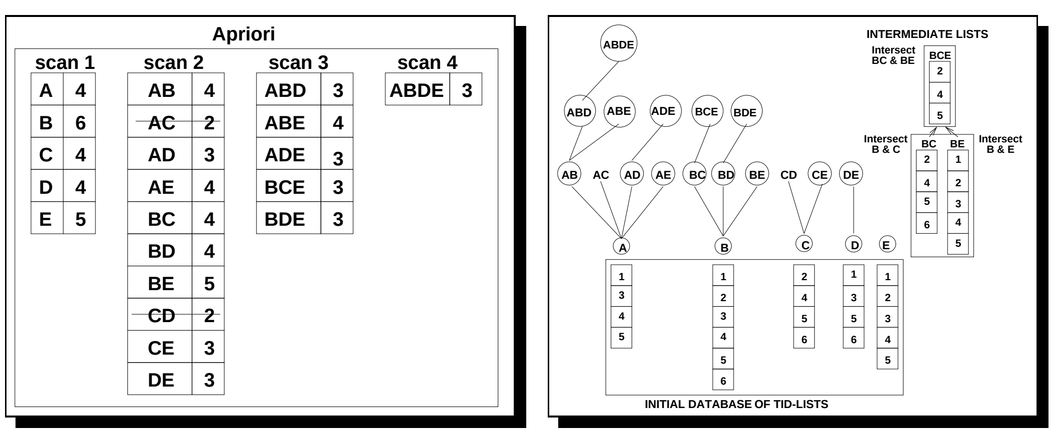
Since rule generation is relatively easy, and less I/O intensive than frequent itemset generation, we will focus only on the first step in the rest of this paper. For example, consider the database shown in Figure 1. There are five different items. The database comprises six transactions; the figure shows all the frequent itemsets contained in at least three of them, i.e., *min_sup*= 50%. The item B appears in all tids; $ABDE$ and BCE are the maximal frequent itemsets.

Horizontal vs. Vertical Format Figure 1 also illustrates the difference between the traditional horizontal and the vertical database formats. In the horizontal approach, each transaction has a tid along with a set of items contained in the transaction. In contrast, the vertical format maintains for each of the items, a list of all transactions in which it is present; this list is called the item's *tidlist*

3 ARM: Algorithms

Apriori The *Apriori* [1] algorithm was the first scalable and efficient method for ARM; it has continued to be the core approach on top of which many other new algorithms have been built. *Apriori* uses a complete, bottom-up search with a horizontal format and enumerates all frequent itemsets. An iterative algorithm, *Apriori* counts itemsets of a specific length in a given database pass. The process starts by scanning all transactions in the database and computing the frequent items. Next, a set of potentially frequent candidate 2-itemsets is formed from the frequent items. Another database scan obtains their supports. The frequent 2-itemsets are retained for the next pass, and the process is repeated until all frequent itemsets have been enumerated (i.e., when the candidate set becomes empty). The algorithm has three main steps (see Figure 2 for a sample execution of *Apriori* on our example database with *min_sup*=50%): 1) *Candidate Generation* Generate candidates of length k, denoted C_k, by a self-join on F_{k-1}, the frequent $(k-1)$ length itemsets. For example, for $F_2 = \{AB, AD, AE, BC, BD, BE, CE, DE\}$, we get $C_3 = \{ABD, ABE, ADE, BCD,$- $BCE, BDE\}$. 2) *Candidate Pruning* Prune any candidate that has at least one infrequent subset. For example, BCD will be pruned because CD is not frequent. 3) *Support Counting* Scan all transactions to obtain candidate supports. *Apriori* stores the candidates in a hash-tree [1] for fast support counting.

The pruning strategy described in [23] was used in *Apriori* versions with data access methods supporting deletion. The pruning strategy tracks the number of candidates each item occurs in during the current pass. For each transaction, if an item in the transaction did not occur in the required number of candidates, it was pruned from the transaction. If the length of a transaction is less than the current pass number, the entire transaction is pruned.



(a) An *Apriori* Example Execution (b) An *Eclat* Example Execution

Figure 2: *Apriori* and *Eclat* Algorithms Applied to Example Data

Eclat A completely different design characterizes the equivalence class-based algorithms proposed in [28], of which *Eclat* can be considered a representative. *Eclat* uses a vertical database format and a complete bottom-up search to generate all frequent itemsets. The main advantage of using a vertical format is that one can determine the support of any k-itemset by simply intersecting the tidlists of the lexicographically first two $k-1$ length subsets that share a common prefix (the generating itemsets). *Eclat* breaks the large search space into small, independent, manageable chunks, i.e., equivalence classes of itemsets sharing the same prefix. These classes can be processed entirely in memory; each class is independent in that it has complete information for generating all frequent itemsets that share the same prefix.

Figure 2 shows the *Eclat* works on our example database. The first equivalence class is the one with the empty prefix, i.e., the set of single frequent items F_1. The main step is to consider all pairwise intersections of the tidlists (denoted $\mathcal{L}$) of elements of an equivalence class, producing new equivalence classes, which are processed recursively. Thus *Eclat* combines candidate generation and counting in a single step. For example, the support of BC is computed by intersecting $\mathcal{L}(B)$ and $\mathcal{L}(C)$ and checking the cardinality of the resulting tidlist against *min_sup*. To see if BCE is frequent, *Eclat* intersects $\mathcal{L}(BC)$ and $\mathcal{L}(BE)$, the two generating sequences of BCE, and so on.

We chose *Eclat* for our base study since it is the representative of the vertical mining methods, it generates all frequent itemsets, and performs bottom-up search. It is thus comparable to *Apriori* , with the major difference being the data format (vertical vs. horizontal), and the counting procedure (intersection of tidlists vs. hash-tree based counting). Figure 2 contrasts how *Apriori* and *Eclat* generate the frequent itemsets from our example database. Thus *Apriori* and *Eclat* serve as the two base approaches exemplifying the trade-offs inherent in the design of efficient ARM methods in terms of the database layout (external) and internal data structures. The optimizations proposed in other ARM algorithms [5, 22, 25, 26] are complementary in nature, since the trade-offs of our study will be applicable to any Apriori-like or Eclat-like algorithm.

Hybrid Algorithms We conjectured that *Apriori* is more efficient than *Eclat* in finding large itemsets in the early passes, when the itemset cardinality is small, but inefficient in later passes of the algorithm, when the frequent itemsets have high length but decrease in number. But *Eclat* on the other hand, has better performance during these passes as it uses tidlist intersections, and the tidlists shrink with increase in the size of itemsets. This motivated a study of an adaptive *hybrid* strategy which switches to *Eclat* in higher passes of the algorithm. Since it is typical to mine the same data many times at different levels of support, we create the vertical format file once (off-line). However, an overhead that arises due to the conversion of candidate itemsets in the *Apriori* phase to a list of prefix-based equivalence classes in the *Eclat* phase, is still unavoidable in the hybrid strategy. The conversion process is optimized so that the tidlist intersections occur only once for the prefix of an equivalence class. The prefix tidlist is then intersected with the tidlist of the items in the equivalence class to form the tidlist of a candidate itemset. Studying the number of candidates that are generated during each pass gives us an idea of how the number of candidates vary with the execution of the algorithm. It is typical for the candidates to increase, reach a maximum and then reduce during the further passes of the algorithm. This information was used to develop a switch logic for the hybrid algorithm. The algorithm maintains the number of candidates generated in the previous two passes of the algorithm. The switch heuristic uses two successive decreases in the number of candidates to switch from *Apriori* to *Eclat* . Two successive decreases are used to eliminate premature switches (a decrease that may be misleading) which may degrade the performance.

4 Data Access Methods and Middleware Design

Apriori uses sequential access to the data. Transactions are accessed in sequential order during each pass. *Eclat* on the other hand, accesses the tidlists based on the items which belong to the equivalence class that is being processed. While *Apriori* uses a method to prune candidates or transactions, implicit pruning is achieved in *Eclat* during the tidlist intersection. Hence, indexed structures that store the data, are required to support fast sequential access to transactions while allowing fast tidlist retrieval based on the items. In addition, the structures should also support pruning of irrelevant items and transactions during the sequential access. B-trees [4, 7, 13, 21] are among the most popular persistent indexing structures and are widely used in databases [9, 11] and in file systems. A B-tree is effectively a tree of index nodes mapping to associated data fields. Many flavors of B-trees support duplicate keys, and some support empty data fields (so that all information is contained in the index). The B^+-tree is a particularly popular variant of B-trees in which each key is stored at the leaf level in a linked list [7, 11, 17]. The *granularity* of data access describes how data is expressed in the index structure, and how the data is partitioned between the index and the associated data field. Our experiences suggested that a more efficient structure using linked lists of blocks might be effective for pruning in horizontal formats (as described in Section 4.3.2).

4.1 Granularity of Data Access

Consider the impact of the interaction between layout (as shown in Figure 1) and our indexing strategy. Horizontal formats group data by transaction, storing the tid and the itemset as a length delimited vector. Vertical formats group data by itemsets, storing the itemset id and the tidlist. Typically tidlists tend to be longer than itemsets, frequently by several orders of magnitude. There is a trade-off between the level of granularity and indexing overhead, giving rise the following design space taxonomy:

- coarse-grained — index on the tid or the itemset id as the key treating the corresponding itemset or tidlist as a variable length data field.
- fine-grained — index ordered pairs using either (tid,item) for horizontal formats and (itemset id,tid) for vertical formats. This level of granularity does not use the data fields associated with the index.
- hybrid granularity — fragment tidlists and/or itemsets into blocks when writing to disk and reassemble while reading into memory. This can be useful if a coarse-grained approach would be desirable but the storage mechanism cannot handle large variable length data items.

In practice hybrid granularity mechanisms would most likely be needed in vertical format approaches due to long tidlists that tend to be encountered.

4.2 Middleware layer design

Different organizations store data in various different layouts/formats. Algorithms requiring data in specialized formats entail a data preprocessing phase. This phase can often be time consuming and expensive. The main motivation for isolating and designing a middleware is to eliminate the dependency of algorithms on data layouts and formats. Thus, various data organizations need to support only a key set of functionalities/access methods in order to support algorithms to be run without preprocessing.

Consider our long term goal of supporting ad hoc queries which drives the middleware layer's design. We used the data access patterns exhibited by *Apriori* and *Eclat* as representative templates of access patterns expected in ad hoc queries (since the queries are likely to invoke the algorithms based on these methods). This structural similarity (with regards to data access) to *Apriori* and *Eclat* allowed us to precisely define the interface functionality. Another important constraint for large data management in general [14], and specifically for general purpose ad hoc mining query tools is that the interface must maximize independence between the algorithm and the underlying data organization while minimizing the amount of efficiency sacrificed. Our selection criteria motivated a design to support efficient access that is transparent with respect to data organization. The interface supporting this access incorporates various high level data access methods as a middleware layer. The high level data access methods used in mining horizontal and vertical data formats are given in Table 1.

A good middleware design must support these access methods for the data organizations supported by the underlying DBMS file access tools. Our middleware layer provides modularity in our code design,

Data Format	Operations Needed
Any	Open Database Close Database Populate Database
Horizontal	Get Next Transaction Reset cursor to start of transaction stream Delete item (if pruning enabled) Delete transaction (if pruning enabled)
Vertical	Get transaction ids associated with itemset insert itemset and associated tidlist

Table 1: Data access requirements according to data format in Association Rule Mining

since changes to the implementation of an algorithm should be restricted to the middleware layer when a new data organization is adopted. For applications requiring run time support for accessing multiple data organizations, a run time binding of access methods to data organization needs to be done. This was not the case for our implementation, and we were able to exploit compile time binding for run-time efficiency.

4.3 Data Access Methods

Various indexed data organizations were studied which provided the functionality for the middleware as described in Section 4.2. Two types of storage were studied. First, we stored the data in B^+-trees and used different types of granularity in storage. Secondly, the nature of data access in *Apriori* , motivated us to develop a less sophisticated but more optimized structure that permitted all the functionality required by *Apriori* , including deletions, and yet whose overhead was not too much from the raw flat file format. This structure was motivated by a minimalist approach of extending flat files to permit deletion of items/transactions. Even though we used this specialized indexing scheme that was applicable only for *Apriori* , it provides a baseline for comparing the indexed data organizations that support pruning in *Apriori* .

4.3.1 Indexed Access using B^+-trees

Two types of B^+-trees were used to store data for the various algorithms. One data organization used a fine grain B^+-tree where the keys were (tid,item) for the transactions (horizontal data format) and (itemset id/item, tid) for the tidlist (vertical data format). This level of granularity enabled a deletion operation to be performed on individual items in a transaction or individual tids in a tidlist. Extensive pruning could be done at the finest level using this type of B^+-trees. This level of granularity also eliminated the need to assemble or update transactions or tidlists during pruning. The second data organization that was used, employed a coarse grain B^+-tree. In this type of data organization, the tid was used as a key for the horizontal data format (transactions) and the itemset that represented the transaction was stored as variable length data. Likewise, the item or itemset id was used as a key in the vertical data format with the tidlist being stored as variable length data. This format needed updates to the data field during pruning while fine grain organization needed only indexed deletions. We wanted to compare the overhead in storage and time for the two organizations.

The issue of how deletions of keys and data fields in B^+-trees and DBMS systems are handled in general is a fundamental attribute of the package used. Physical deletions restructure the index and leaf nodes of the tree, reclaiming the space freed up during deletion. Logical deletion does not do this, but rather marks the keys/data as being deleted. This can result in inefficient memory usage, since a page could have relatively few undeleted keys or data fields on it. However logical deletions have an advantage, that is, they do not corrupt the data but only set deletion flags in the meta data; thus we can roll back to the original state. This rollback could be done using a single post processing pass marking all deleted information as undeleted to restore the B^+-tree to its original state after pruning has been done by the ARM algorithm.

Pruning in Indexed Data Access Methods Vertical format algorithms (e.g. *Eclat*) tend not to benefit from explicit pruning, since tidlist intersection does implicit pruning. Hence, pruning was not done for the vertical data formats in coarse grained layouts. For Fine Grained Hybrid algorithms, however, pruning the vertical data format, which is stored in what we call the *inverted* B^+*-Tree* could potentially reduce the tidlist size during the conversion process. Hence, pruning a (tid,item) pair from the horizontal B^+-Tree also pruned the corresponding (item,tid) pair from the vertical B^+-Tree.

Remarks about *Eclat* and Hybrid algorithms using indexed data access methods In order to isolate the data access methods from the algorithm and develop hybrid strategies, we used the horizontal format to compute the large items and large 2-itemsets. *Eclat* then switches to the vertical data organization and forms tidlists for each of the large 2-itemsets. Hybrid, similarly, uses *Apriori* in the initial passes until it switches to *Eclat* . The main difference being that the switch may be in a pass that is higher than 2. Thus *Eclat* that uses the middleware is essentially the hybrid approach with the switch to *Eclat* occurring at pass 2.

4.3.2 Data Access Using A Linked Block File Structure

Consider a pruning version of *Apriori* [23] that repeatedly traverses the data one transaction at a time. Each time a transaction is visited, the counts of the relevant candidates are updated and then part or the entire transaction are pruned. In addition to the horizontal database operations described in Section 4.2, any new data organization must provide functionality to store data in that format. A good design must both use a space efficient storage format and provide these operations in $O(1)$ time (as measured in number of disk and memory accesses) with low constant factors. Suppose we have N transactions, and where the kth transaction has itemset I_k. Then the mean itemset cardinality is $E[I] = \frac{1}{N}\sum_{k=1}^{N} |I_k|$ and the maximum cardinality (size) is $S_{\max} = \max_{1\leq k\leq N} |I_k|$, by definition. The flat file storage of a horizontal format in a coarse grained representation is $N(E[I] + 2)$ integers, since each transaction records both a transaction id (TID) and the cardinality of the itemset. Thus the storage requirement of the horizontal format is less than the corresponding fine grained representation overhead of $2NE[I]$ integers for the common case where $E[I] > 2$. The most common operations performed during pruning in *Apriori* are, reading of transactions (in the worst case there are $NS_{\max}$ items read) and deletion (up to N transactions deleted and up to $\sum_{k=1}^{N} |I_k|$ items deleted, since an object can be deleted at most once). Using B^+ trees [7, 17, 18] permits $O(1)$ disk and memory operations for reads (same as a flat file), however the deletion overhead is $O(\log_m N)$ for a tree with N elements and m keys per index node due to index maintenance. However, since *Apriori* always positions the data cursor at the transaction where the pruning is going to occur, it is possible to avoid the index traversal, in fact a linked list structure (reminiscent of the leaf node structure in B^+trees) could reduce the delete overhead to $O(1)$ without increasing the reading overhead. We exploit the fact that most databases have a short average transaction length relative to the disk sector size (as reflected in kernel level buffering) and aggregate consecutive transactions to the same file block (desirable to permit efficient prefetching of transactions during traversal) and keep the disk transaction size sufficiently large, motivating a linked block structure as shown in Figure 3. For ease of implementation, we currently do not support single transactions that span blocks, so there may be some unused space in the blocks (called *padding*). We assume that the structure is initially empty and populate it by appending transactions, with a new block being appended to the file and linked into the structure in the event that the transaction does not fit into the remaining space in the block. Let B denote the block size (amount of space in the block available for user data), the wasted space overhead induced by padding[1] is $\frac{B-S_{\max}}{B}$, which is typically small since we choose B such that $B \gg S_{\max}$. Reading a transaction consists of fetching the next block on demand if needed, and doing a

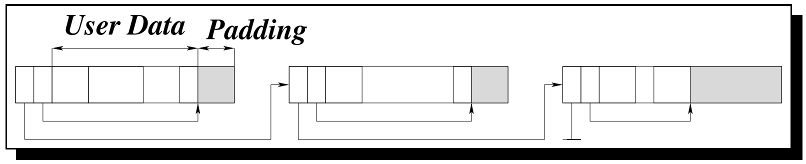


Figure 3: An Example of Linked Block File Structure

pointer offset calculation into the block. For efficiency, transactions are scanned in using a zero copy read that returns a pointer into the memory used to cache the block. Lazy physical deletion is employed by marking items and tids with the value -1 within the block to indicate deletion, and setting a dirty flag for the block. Compaction is done when the block is selected for replacement, in a single pass over the contents of the block. Double buffering is used when the block is completely scanned, so that both the previously read block (if it exists) and the current block are cached. If the the total amount of the data of the two blocks can fit in a single block they are merged into the predecessor block, and the current block is deleted from the linked list. When either the last block is read or it is not possible to merge the current and previously read blocks, then the previously read block is flushed. It can be shown that on average, no more than $\frac{B}{2}$ padding overhead is used, since consecutive pairs of blocks are merged whenever the data contained is less than B. This Linked Block Structure will henceforth be referred to as BFS (stands for Block File structure) layout.

5 Empirical Study

Our use of indexing packages was motivated by a desire to be able to integrate mining techniques more tightly with existing DBMS systems. In keeping with our requirements for efficient indexed and sequential access we selected two freely available B^{+}-tree packages, supporting B^{+}-tree style access,the *generalized indexed search tree* (GiST) [13] and Sleepycat's Berkeley DB [21]. Both *Apriori* and *Eclat* algorithms were developed using indexed data access via a middleware layer as described in Section 4.2. We used both synthetic and real data sets as described in Section 5.1 to analyze the storage overhead of different data representations (see Section 5.3) and to benchmark the performance overhead of various data representations, as seen in Section 5.4.

5.1 Databases Used

The algorithms were benchmarked using a popular set of synthetic databases for many ARM implementations. We used the synthetic benchmark data generation approach described in [1]. The generator is designed to mimic the transactions seen in a retail environment, where people tend to buy sets of items together. The cardinality of the maximal itemsets is clustered about a mean, with a few longer itemsets. A given transaction may contain one or more such frequent sets. The transactions size is also clustered about a mean, however, few of them contain many items.

Two synthetic benchmark databases were chosen to demonstrate the experimental results. Let D denote the number of transactions, T the average transaction size, I the size of a maximal potentially frequent itemset, L the number of maximal potentially frequent itemsets, and N the number of items. In all our experiments, we use $N = 1000$ and $L = 2000$. Experiments are conducted with different values of D, T and I.

[1]The last block is an exception, since it may in worst case contain a single small transaction. This overhead can be treated as negligible since we choose a small B relative to the amount of user data, that is $B \ll 2N + \sum_{k=1}^{N} |I_k|$.

For real databases, we selected three examples from the UCI database repository [3]: CHESS, MUSHROOM and CONNECT, for our experiments. We used our own procedures for database format conversion to allow running the respective algorithms on the databases.

5.2 Experiment Design and Results

The experiments used a 500MHz dual Intel Celeron processor[2] machine with 256MB of ECC RAM and 1GB of swap space running FreeBSD 4.2. The disk controller uses UDMA 33 technology. and the drive is a single IBM Deskstar 34GXP 20.5 GB drive with 7200 RPM rotation speed and 9.0 msec mean access time.

The objectives of the experiment were categorized as follows.

- Study of the effect of data organization on execution time
- Study of the storage efficiency/overhead for various data organizations
- Study the effect of data organization on where the methods spend their time

We explored these issues across the range of algorithms and data access methods presented in Table 2. For example, *Apriori* with flat-files is labeled APR, while *Apriori* with BFS is labeled ARP-BFS, and so on. Because we used compile time binding of data access methods, the BFS layout described in Section 4.3.2 cannot be used for *Eclat* and Hybrid approaches as the long tidlists would need to span blocks in the BFS layout (which is currently not supported). The systematic testing of each algorithm and its available data access methods allowed us to explore a range of interactions and isolate the impact of the data access method on the algorithms performance.

Algorithm	Flat File	Fine Grain GiST	Coarse Grain GiST	BFS	Sleepycat
Apriori	APR	APR-FG	APR-CG	APR-BFS	APR-DB
Eclat	ECL	ECL-FG	ECL-CG		ECL-DB
Hybrid		HYB-FG	HYB-CG		HYB-DB

Table 2: Notational Conventions for ARM Algorithms/Data Access Method Combinations Used

5.3 Effect of Data Organization on Database size

The storage costs of mining are influenced by data layout, indexing and granularity on storage overhead, as shown in Figure 4. Flat files do not contain metadata and thus were expected to have the lowest storage overhead. However, a small amount of data omitted in the other files (a customer ID for each transaction, which is not used in ARM) was inserted by the data generator. This small amount of data occasionally exceeded the padding and metadata overheads used in BFS. We define the *relative storage overhead of a over b* between two file layouts as the ratio of the file sizes of a and b. The relative storage overheads of BFS over flat files, $\frac{\text{BFS file size}}{\text{Flat file size}}$, ranged from a low of 0.87% for T5I4D1600K to a high of 1.03% for chess. For indexed data, the dedicated B^+-tree implementation of Sleepycat was uniformly more storage efficient than the emulated B^+-tree of GiST. For coarse grained horizontal formats GiST's relative storage overhead over Berkeley DB was in the range $2.03 \leq \frac{\text{CG file size}}{\text{DB file size}} \leq 2.33$. For vertical format coarse grained layouts GiST appeared somewhat more efficient, with a relative storage overhead over Sleepycat Berkeley DB of $1.12 \leq \frac{\text{CG-Vertical file size}}{\text{DB-Vertical file size}} \leq 1.98$. Fine grained layouts induced what appears to be disproportionate file size increases in GiST, and making the file $8.72 \leq \frac{\text{FG file size}}{\text{Flat file size}} \leq 13.15$. Recall, as per analysis in Section 4.3.2, there is no more than twice as much data stored in a fine grained format than in the corresponding coarse grained flat file representation of the same information. However, GiST's metadata overhead is evident as the number of keys stored increases, as is evidenced by the fact that fine grained GiST files are $3.27 \leq$

[2] Although overclocking of such machines is popular, ours was running at its base clock speed.

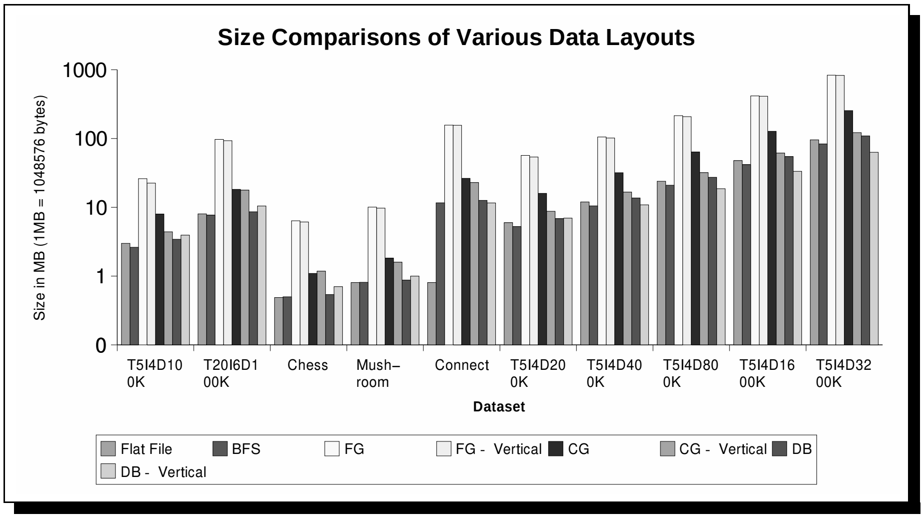


Figure 4: Size Comparison for Data Layouts

$\frac{\text{FG file size}}{\text{CG file size}} \leq 5.93$. times larger than their equivalent coarse grained data representations. The fine grained vertical representations also tended to have increased relative overhead over coarse grained vertical formats, with $5.1 \leq \frac{\text{FG-Vertical file size}}{\text{CG-Vertical file size}} \leq 6.8$.

5.4 Effect of Data Organization on Run Time

Run time is sensitive to the algorithm used, the data layout, minimum support and which file access method used. Figure 5 presents execution times for the synthetic databases described in Section 5.1, measuring both sensitivity to support and scalability with database size. Figure 6 presents execution times for the real databases described in Section 5.1.

We sought to measure how run times of indexing and BFS methods without pruning compared against flat file run times. The APR flat file algorithms tended to slightly outperform the BFS algorithms, the speedup of flat files over BFS fell within the range $1.05 \leq \frac{\text{APR-BFS Run Time}}{\text{APR Run Time}} \leq 2.2$, with the largest speedup in the T5I4D100K database, and the smallest speedup in the T20I6D100K database. Most databases and supports experiencing a speedups in the range of 1.0 and 1.3.

The pruning variants of *Apriori* use either BFS or indexing, in order to trade off the extra processing of pruning in an attempt to reduce scanning in later passes. The overhead of pruning was frequently higher. The minimalist design of the BFS layer came close to matching flat file performance, only occasionally outperforming flat files (as seen in the T20I6D100K timings in Figure 5(d), for $min\_sup \in \{0.3, 0.4\}$, both methods require 13 passes). BFS with pruning tended to outperform BFS without pruning, although there were some exceptions, e.g. the timings in Figure 6(a) and (g), due to BFS's support for physical deletion. The fact that non-pruning indexed methods outperformed indexed pruning methods may reflect the use of an indexing tool that does not support physical deletion (GiST) and thus we did not get a performance improvement due to reduced scanning load in later passes.

For flat files, coarse grained indexing and BFS approaches, *Eclat* versions performed well and were scalable relative to *Apriori* versions, as seen in Figures 5 and 6. Because *Apriori* and *Eclat* use the indexed structures differently, only *Eclat* uses insertion, only pruning variants of *Apriori* use deletion, while other approaches use only retrieval. We expected *Apriori* to perform well in the early passes, and that later passes would benefit from *Eclat* 's depth first approach. We employed a hybrid approach where the algorithm postponed the switch from *Apriori* to *Eclat* until a pass where *Apriori* uncovered fewer new candidates. However, our hybrid approach was consistently outperformed by *Eclat* , in fact we discovered that forcing the switch to *Eclat* at the beginning (after second pass) worked best. Other horizontal mining algorithms may have different performance characteristics and have different switch-over points than *Apriori* .

Fine grained structures reflect the layout used in supporting highly normalized data representations in relational database systems. Fine grained approaches using GiST were strongly outperformed by their coarse grained equivalents. This implies that the overhead of the extra data, indexing and the fragmentation and reassembly of transactions/tidlists dominated. The amount of speedup gained by going to a coarse grained representation appears to be a function of the mean itemset or tidlist length. The largest speedup was $42 = \frac{\text{ECL-FG run time}}{\text{ECL-CG run time}}$ for the connect database. The *Eclat* speedup obtained by going from coarse grained to fine grained representations was sensitive to *min_sup*, for example the T5I4D100K database had a monotonically decreasing speed up as *min_sup* increased, with a speedup of 10.3 for *min_sup* $= 0.002$, and a speedup of 4.7 for *min_sup* $= 0.006$. The conversion from horizontal to vertical format fine-grained layout was a major factor contributing to the speedup of coarse grained over fine grained *Eclat* . The competing hybrid, and *Apriori* approaches in particular had much smaller speedups of coarse grained over fine grained approaches, and were much less sensitive to *min_sup*. The pure *Apriori* approaches enjoyed speedups of about 5 for most databases, with the largest speedup being for the largest values of support; the hybrid approaches had a larger speedup (as high as 11 for Mushroom).

Combined pruning and B^+-tree style indexing was more effective than the corresponding non-pruning variant for large values of *min_sup*, since that promoted the most aggressive early pruning. The fine-grained indexed pruning approaches with frequent itemsets longer than 3 tended to be relatively insensitive to *min_sup*. Much of the pruning occurs during iteration 4, as seen in Section 5.5.

5.5 Study of Pruning

The cost of pruning in early passes requires that later passes have sufficiently reduced volumes of data to scan to offset the additional processing required. Figure 7 shows the iteration-wise execution time split for APR and for APR-CG and APR-BFS with and without pruning. The initial few iterations cost the pruning algorithms a lot as active pruning is done in the initial passes of the algorithm. As the iterations proceed, the effect of reduction in database size plays a role in speeding up the algorithms. To note, pruning on indexed data layouts, speeds up the algorithm in later passes to make it faster than flat file data layout. The timings confirm the hypothesis that pruning can actually help in databases where the algorithm runs into a fairly large number of iterations. Also, the physical deletion in BFS leads to more effective pruning, as seen in the drastic reduction in run time for later passes.

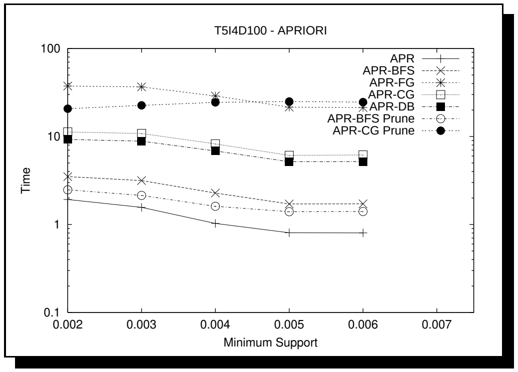
5.6 Effect of data organization on where the methods spend their time

In order to study the effect of data organization on where algorithms spend their time, profiling of CPU time was done for the different algorithms and data formats using the GNU profiling tool, **gprof**. Profiling statistics were measured at the function level (which tends to be less intrusive) with additional runs performed to obtain more detailed profiles with line numbering information. To give a basis of comparison, we aggregated timings based on what sorts of jobs the functions did. We classified each function as performing one of the following tasks: first pass, second pass, candidate generation, counting, data access method operations, and other operations.

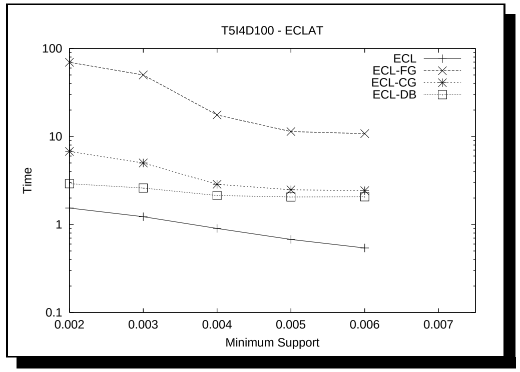
Figure 8 shows the profiling plots for various algorithms. The results are shown for a sample profile run on T20I6D100k with a support threshold of 0.25%. The majority of time for almost all data layouts is spent on functions for support counting or candidate generation. The overhead of fine grained data representation is reflected in the fine grained *Apriori* , which spends almost half the time in data access functions.

Our profiling measurements confirmed the belief that counting and candidate generation tend to dominate run time in coarse grained horizontal data format algorithms. In contrast, Figure 8 shows that the fine grained B^+-tree algorithms' run times were dominated by executing data access methods.

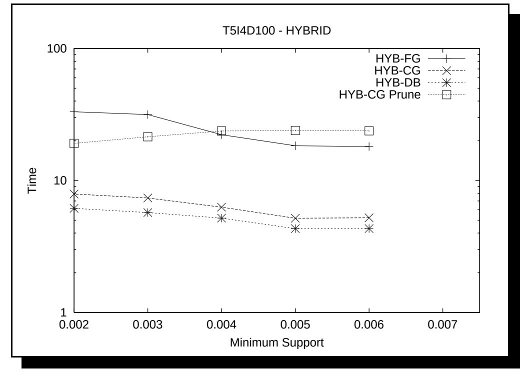
When comparing fine grain and coarse grain methods, we expected the time spent during data access methods to decrease as the transactions and itemset tidlists are grouped together. The time spent in data



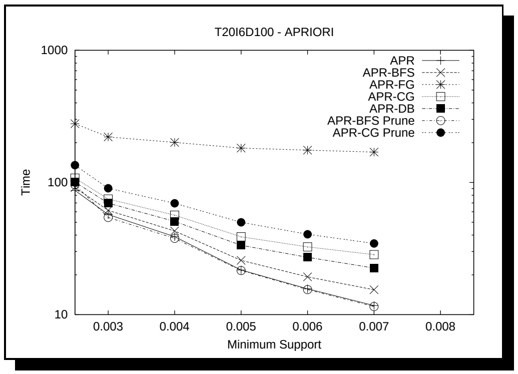
(a) T5I4D100K *Apriori* Timings



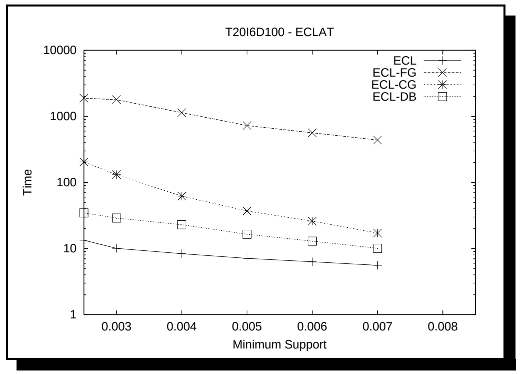
(b) T5I4D100K *Eclat* Timings



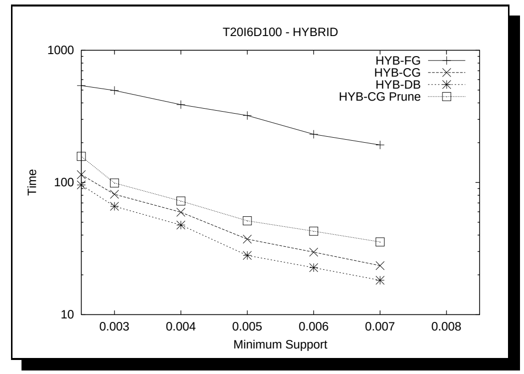
(c) T5I4D100K Hybrid Timings



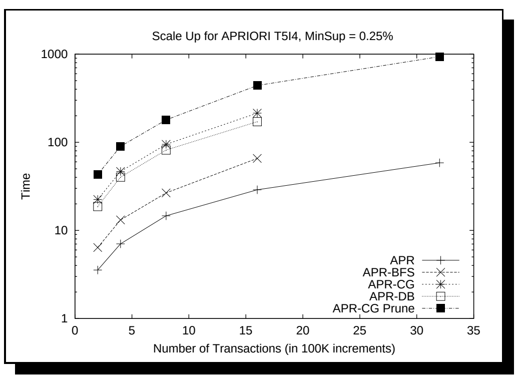
(d) T20I6D100K *Apriori* Timings



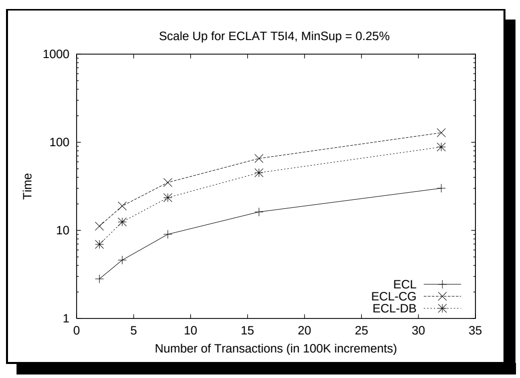
(e) T20I6D100K *Eclat* Timings



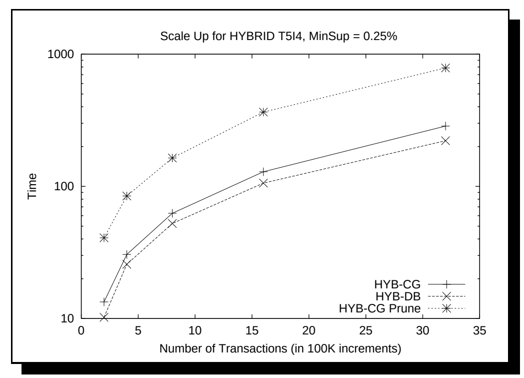
(f) T20I6D100K Hybrid Timings



(g) Scaleup *Apriori* Timings

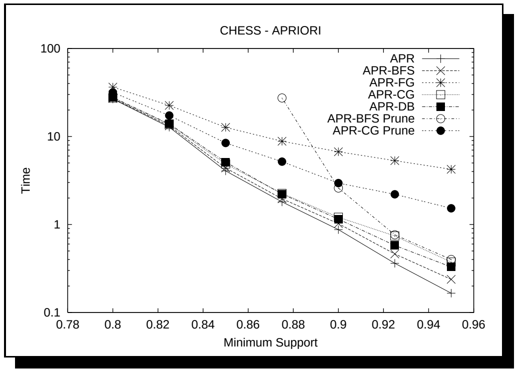


(h) Scaleup *Eclat* Timings

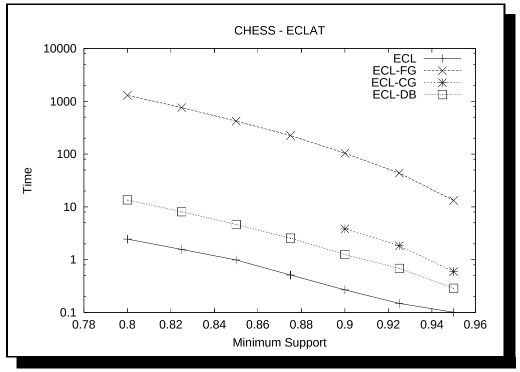


(i) Scaleup Hybrid Timings

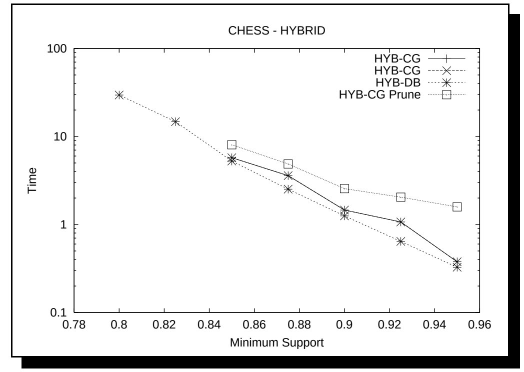
Figure 5: Synthetic and Scaleup Database Timings



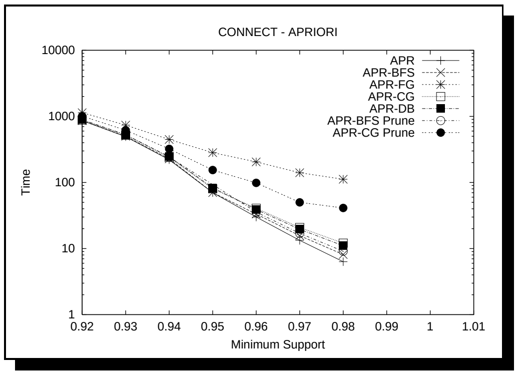
(a) Chess *Apriori* Timings



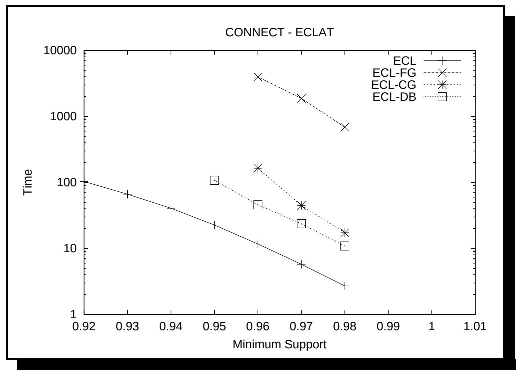
(b) Chess *Eclat* Timings



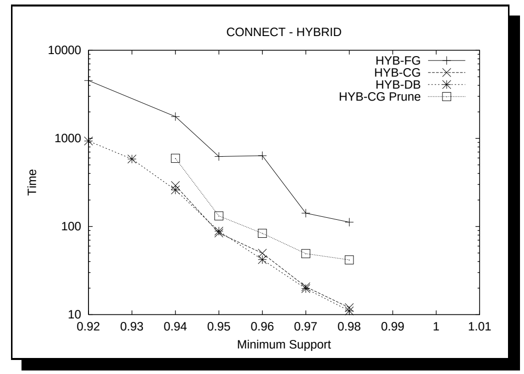
(c) Chess Hybrid Timings



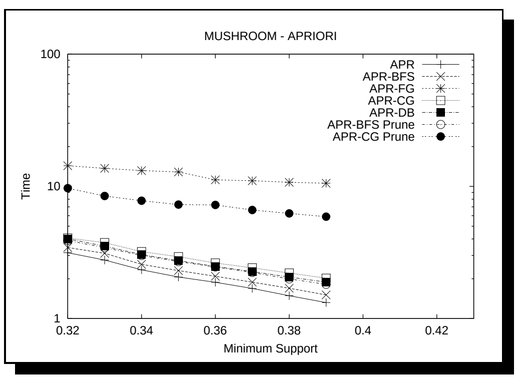
(d) Connect *Apriori* Timings



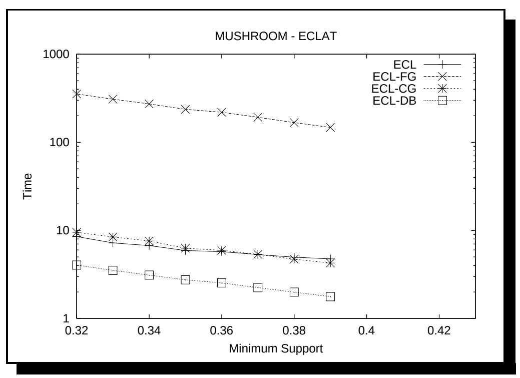
(e) Connect *Eclat* Timings



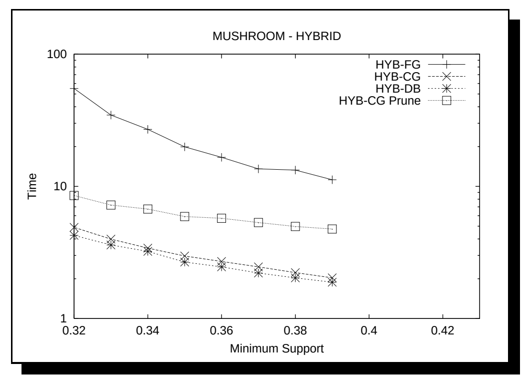
(f) Connect Hybrid Timings



(g) Mushroom *Apriori* Timings



(h) Mushroom *Eclat* Timings



(i) Mushroom Hybrid Timings

Figure 6: Real Database Timings

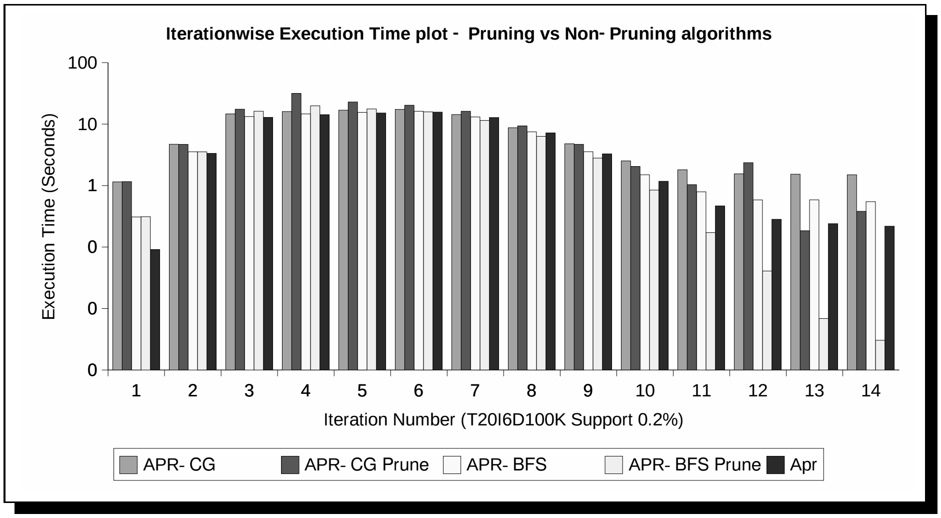


Figure 7: Iteration-wise Execution time for Pruning and Non-Pruning algorithms

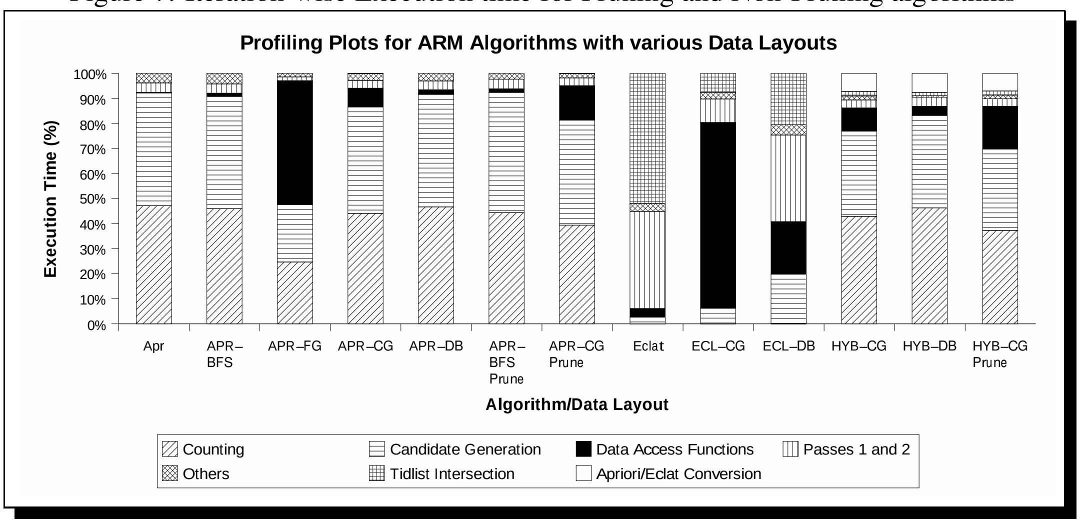


Figure 8: Profiles of *Apriori* Execution with various data layouts

access methods in APR-FG was 49.5% and this reduced to 7.5% in APR-CG and 1.9% in APR-DB.

Flat files and the linked block file structure appeared to be the fastest data access methods. We expected that BFS would be fast, as it uses a minimalist implementation, designed to have low overhead. The measurements confirmed our intuition, as the amount of time spent by a linked block file structure in data access methods was low, approaching the negligible time of the flat file version of *Apriori* . Hence the expected performance of APR-BFS was close to Flat file *Apriori* . The low overhead of the cursor management in APR-DB resulted in processor time utilization comparable to APR-BFS.

6 Summary and Conclusion

We believe that integrating KDD tools with database management systems will improve the mining experience of users, eventually providing seamless systems for DBMS and KDD [19]. Currently, most approaches use flat files, and thus require exporting the data out of the DBMS. In contrast, integrated schemes could avoid file system imposed file size limitations and redundant storage overhead. While others have explored high level approaches to express mining operations using high level query languages, we are investigating how to do low level support for mining and the impact of file structures on mining algorithms and run time support issues, so that informed design decisions can be made. This systematic investigation consisted of determining the individual and collective impact on storage and run time of the following orthogonal design elements:

- horizontal and vertical partitioning based algorithms, *Apriori* and *Eclat* .
- traditional flat files, a novel BFS file which supports pruning, and indexed structures.
- granularity of data access (corresponding to level of normalization in a DBMS).
- the impact of pruning approaches on horizontal ARM (*Apriori*) run time.

Flat file access tended to be faster than coarse grained indexing, often getting a speedup of 5 or more above coarse grained indexed methods for *Apriori* based methods. Our efficient linked block file structure when used without pruning had performance approaching that of flat file access and showed that access methods supporting physical deletion can get significant performance improvement in later passes. Structures using logical deletion experience lesser performance improvements. However, the overhead of determining when to prune was sufficiently large that it tended to offset (and often overwhelms) the benefit of reduced scanning in later passes. It should be noted that logical deletion is preferred to physical deletion in cases where it is desirable to roll back to the original data after mining. We would like to underscore that run time is not the only concern. The fact is that almost 80-90% of the time in KDD is spent in pre-/post-processing. Thus, tight integration of mining with a database makes practical sense when one considers the entire KDD process. Databases facilitate ad-hoc mining and post-mining analysis of results.

For vertical data formats, support for large data items associated with the key, binary large objects (BLOBS), appears to be critical. The Sleepycat Berkeley DB B^+-tree's BLOB support is more efficient than the corresponding GiST B^+-tree which appears to have implicit limitations on the size of data fields associated with the keys. For *Eclat* , coarse grained implementations using Sleepycat Berkeley DB were able to come within a factor of 5 of flat file performance. However, if the data is highly normalized (e.g. in fine-grained layout), the performance of these methods declines dramatically due to transaction reassembly costs, with slowdowns as high as 600 for the chess database. We explored an adaptive approach to test the conjecture that *Apriori* tends to be more efficient in the earlier passes than *Eclat* .

The hybrid method tended to be uniformly less efficient than *Eclat* for the databases but more efficient than *Apriori* , as the overhead of candidate generation and counting of *Apriori* proved more expensive than the intersection methods used in *Eclat* , even in early iterations

The creation of the middleware layer is an important step in developing a flexible and unified software approach to implementing mining tools. With such a layer, it is possible to interchange components allowing many variants of a mining approach. Additionally, a well crafted middleware layer can facilitate experimentation and analysis of various design trade-offs.

Several future directions present themselves, due to issues encountered during our analysis. The memory requirements (for storing candidates, for example) of various approaches at times approached or exceeded the machines capacity. Work on out of core approaches might provide a means for increasing our capacity to mine databases, and allow ARM users to use smaller values of *min_sup*. We also felt that using existing profiling technology to diagnose where the software is spending its time does not provide the level of information we would like, in particular, improved profiling technology should permit aggregation of functions together for timing statistics, and allow the measurement of wall clock and idle time, as well. For highly data dependent tools like KDD tools, it may make sense to automate exporting of the profilers data to a DBMS so that queries can be made on the data to correlate performance across a range of inputs. Mining of profiled data might provide a sort of feedback loop that allows us to improve both compiler and KDD tool performance.

References

[1] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Inkeri Verkamo. Fast discovery of association rules. In U. Fayyad and et al, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI Press, Menlo Park, CA, 1996.

[2] R. Agrawal and K. Shim. Developing tightly-coupled data mining applications on a relational database system. In *2nd Intl. Conf. on Knowledge Discovery in Databases and Data Mining*, August 1996.

[3] S. Bay. *The UCI KDD Archive (kdd.ics.uci.edu)*. University of California, Irvine. Department of Information and Computer Science.

[4] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

[5] S. Brin, R. Motwani, J. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *ACM SIGMOD Conf. Management of Data*, May 1997.

[6] S. Chaudhri, U. Fayyad, and J. Bernhardt. Scalable classification over SQL databases. In *15th IEEE Intl. Conf. on Data Engineering*, March 1999.

[7] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[8] B. Dunkel and N. Soparkar. Data organization and access for efficient data mining. In *15th IEEE Intl. Conf. on Data Engineering*, March 1999.

[9] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Addison Wesley, Reading, Massachusetts, USA, 3rd edition, 2000.

[10] A. Freitas and S. Lavington. *Mining very large databases with parallel processing*. Kluwer Academic Pub., Boston, MA, 1998.

[11] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database System Implementation*. Prentice Hall, 1999.

[12] J. Han, Y. Fu, W. Wang, K. Koperski, and O. Zaiane. DMQL: A data mining query language for relational databases. In *1st ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, June 1996.

[13] J. Hellerstein, J. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *Proc. 21st International Conference on Very Large Data Bases*, Zurich, September 1995. VLDB Endowment, Morgan Kaufmann. Part of the SIGMOD Anthology CDROM series.

[14] J. Hellerstein, M. Stonebraker, and R. Caccia. Independent, open enterprise data integration. *IEEE Data Engineering Bulletin*, 22(1):43–49, March 1999.

[15] T. Imielinski and H. Mannila. A database perspective on knowledge discovery. *Communications of the ACM*, 39(11), November 1996.

[16] T. Imielinski and A. Virmani. MSQL: A query language for database mining. *Data Mining and Knowledge Discovery: An International Journal*, 3:373–408, 1999.

[17] J. Jannink. Implementing deletion in B^+-trees. *ACM SIGMOD Record*, 24(1):33–38, 1995.

[18] R. Maelbrancke and H. Olivie. Optimizing Jan Jannink's implementation of B+-tree deletion. *SIGMOD Record*, 24(3):5–7, 1995.

[19] W. A. Maniatty and M. J. Zaki. Systems support for scalable data mining. *SIGKDD Explorations*, 2(2):56–65, January 2001.

[20] R. Meo, G. Psaila, and S. Ceri. A new SQL-like operator for mining association rules. In *22nd Intl. Conf. Very Large Databases*, 1996.

[21] M. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the 1999 Summer Usenix Technical Conference*, Monterey, California, June 1999.

[22] J. S. Park, M. Chen, and P. S. Yu. An effective hash based algorithm for mining association rules. In *ACM SIGMOD Intl. Conf. Management of Data*, May 1995.

[23] J. S. Park, M.-S. Chen, and P. S. Yu. Using a hash-based method with transaction trimming for mining association rules. *IEEE Transactions on Knowledge and Data Engineering*, 9(5):813–825, September/October 1997.

[24] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with databases: alternatives and implications. In *ACM SIGMOD Intl. Conf. Management of Data*, June 1998.

[25] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *21st VLDB Conf.*, 1995.

[26] P. Shenoy, J.R. Haritsa, S. Sudarshan, G. Bhalotia, M. Bawa, and D. Shah. Turbo-charging vertical mining of large databases. In *ACM SIGMOD Intl. Conf. Management of Data*, May 2000.

[27] D. Tsur, J.D. Ullman, S. Abitboul, C. Clifton, R. Motwani, and S. Nestorov. Query flocks: A generalization of association rule mining. In *ACM SIGMOD Intl. Conf. Management of Data*, June 1998.

[28] M. J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372–390, May/June 2000.

[29] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. Parallel algorithms for discovery of association rules. *Data Mining and Knowledge Discovery: An International Journal*, pages 343–373, 1997. special issue on Scalable High-Performance Computing for KDD.