Indexing and Data Access Methods for Database Mining

Ganesh Ramesh[*], William A. Maniatty [†]
Dept. of Computer Science
University at Albany
Albany, NY 12222
http://www.cs.albany.edu/(∼ganesh,∼maniatty)
{ganesh,maniatty}@cs.albany.edu

Mohammed J. Zaki [‡]
Computer Science Dept.
Rensselaer Polytechnic Institute
Troy, NY 12180
http://www.cs.rpi.edu/∼zaki
zaki@cs.rpi.edu

Abstract

Most of today's techniques for data mining and association rule mining (ARM) in particular, can be aptly termed "flat file mining", since the database is typically transformed to a flat file that is input to the mining software. Previous research in the integration of ARM with databases looked largely 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

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, little interaction occurs between the mining engine and the database. Techniques are needed to bridge this gap. The ultimate goal would be to support ad hoc data mining queries, focusing on increasing programmer productivity for mining applications [10]. Previous research in integrating mining and databases has mainly looked at the language support. DMQL [6] 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 [11] 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 [15] extends the semantics of association rules, allowing more generalized queries to be performed. Query flocks [20] 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 [19], 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 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.

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 and for analysis. 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

[*]This work was in part supported by a research assistantship provided by the Department of Computer Science at the University at Albany.

[†]This work was supported in part by a startup grant from the Department of Computer Science at the University at Albany.

[‡]This work was supported in part by NSF CAREER Award IIS−0092978, and NSF Next Generation Software Program grant EIA−0103708.

analysis has been done for ARM methods. One such work by Dunkel and Soparkar [5] 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.

Association Rule Mining and related terminology are described 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. Subsets of an itemset, of length k are called k-subsets. An itemset is frequent or large if its support is more than a user-specified minimum support value (*min_sup*). 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*).

To explore the impact of algorithm selection and data layout, we compared and contrasted *Apriori* [1], a bottom up breadth first counting based approach on horizontal data layouts and *Eclat* [21], a depth first intersection based approach on vertical data layouts. For *Apriori* we measure the impact of pruning approaches [17].

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 we created the vertical format file offline. However, the conversion of candidate itemsets in the *Apriori* phase to a list of prefix-based equivalence classes in the *Eclat* phase, imposes an unavoidable overhead in the hybrid strategy. Hybrid strategies were also studied in [19] and [9]. While the former strategy only involved data representations, the latter also involved a change in search strategy (DFS as opposed to BFS). To avoid premature switching, our hybrid approach uses a heuristic which switches from *Apriori* to *Eclat* when successive iterations of *Apriori* , experiences a decrease in the number of candidates.

2 Data Access Methods and Middleware Design

Consider the impact of the interaction between data layout 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. To explore the trade-off between the level of granularity and indexing overhead, we use 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 $(tid, item)$ for horizontal formats and $(itemsetID, tid)$ 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.

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 [8], 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.

2.1 Low Level Data Access Methods and A Linked Block Structure

Various indexed data organizations support the functionality for the middleware as described in Section 2, two of which were studied. First, we used B^+-trees with different levels of granularity to store the data. Secondly, the nature of data access in *Apriori* , motivated us to develop

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

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* .

Consider a pruning version of *Apriori* [17] 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 2, any new data organization must provide functionality to store data in that format. The most common operations performed during pruning in *Apriori* are, reading of transactions and deletion. Using B^+ trees [4, 12, 13] 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. Reading a transaction consists of fetching the next block on demand if needed, and doing a 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. 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 to support merging. This Linked Block Structure will henceforth be referred to as BFS (stands for Block File structure) layout. For more details see [18].

3 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) [7] and Sleepycat's Berkeley DB [16].

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]. 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.

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.

The experiments used a 500MHz dual Intel Celeron processor 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: (i) observe the effect of data organization on execution time, (ii) explore the storage efficiency/overhead for various data organizations and (iii) 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 APR-BFS, and so on. As compile time binding of data access methods was used, the BFS layout described in Section 2.1 could not be used for *Eclat* and Hybrid approaches, as long tidlists would need to span blocks (which is currently not supported by the BFS layout). 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.

3.1 Effect of Data Organization on Run Time

Run time is sensitive to the algorithm used, the data layout, minimum support value and the file access methods.

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

Figure 1 presents execution times for the synthetic and real databases described in Section 3, measuring both sensitivity to support and scalability with database size. Due to space constraints, we present only a subset of all the plots and refer the reader to [18] for a more comprehensive version of the same.

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 1(d), for *min_sup* $\in \{0.3, 0.4\}$, both methods required 13 passes). BFS with pruning tended to outperform BFS without pruning, although there were some exceptions, e.g. the timings in Figure 1(a) and (g), due to BFS's support for physical deletion. The use of an indexing tool that does not support physical deletion (GiST) does not lead to a performance improvement due to reduced scanning load in later passes over the dataset and hence may reflect the better performance of non-pruning indexed methods over their pruning counterparts.

For flat files, coarse grained indexing and BFS approaches, *Eclat* versions performed well and were scalable relative to *Apriori* versions, as seen in Figure 1. Both *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 as described in section 1 to utilize the best cases of both methods. 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 for the hybrid approach and may switch at different passes than *Apriori* .

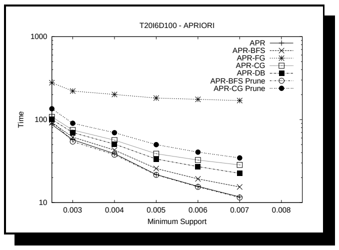
Fine grained structures reflect the layout used in supporting highly normalized data representations in relational database systems. Fine grained approaches using GiST were strongly outperformed by their coarse grained equivalents. This implies that the overhead of the extra data, indexing and the fragmentation and reassembly of transactions/tidlists dominated the computation. 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 with increase in *min_sup*, 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 low values of *min_sup*, since that promoted aggressive pruning in early passes. 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.

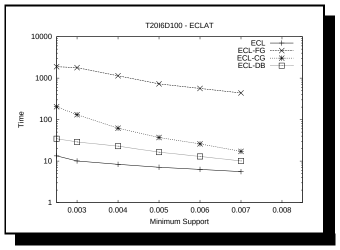
The cost of pruning in early passes requires that later passes have sufficiently reduced volumes of data to scan to offset the additional processing required. Figure 2 shows the iteration-wise execution time split for APR, APR-CG and APR-BFS with and without pruning. The cost of pruning in the initial few iterations was a lot, due to active pruning in the initial passes. 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.

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

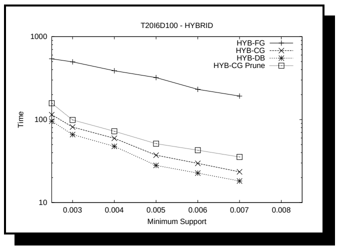
In order to study the effect of data organization on where algorithms spend their time, profiling of CPU time was done for the different algorithms and data formats using the GNU profiling tool, **gprof**. Profiling statistics were measured at the function level (which tends to be less in-



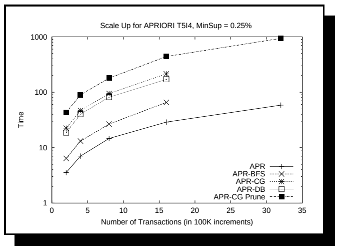
(a) T20I6D100K *Apriori* Timings



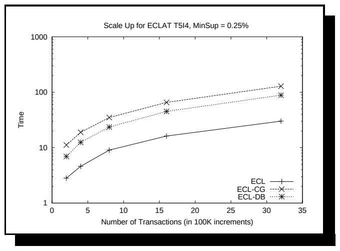
(b) T20I6D100K *Eclat* Timings



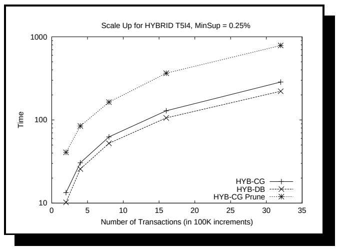
(c) T20I6D100K Hybrid Timings



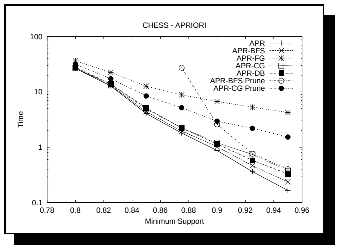
(d) Scaleup *Apriori* Timings



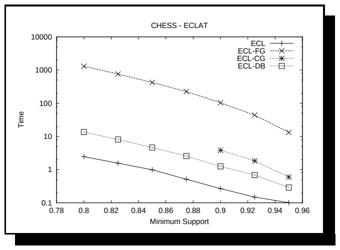
(e) Scaleup *Eclat* Timings



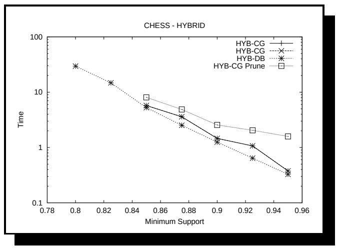
(f) Scaleup Hybrid Timings



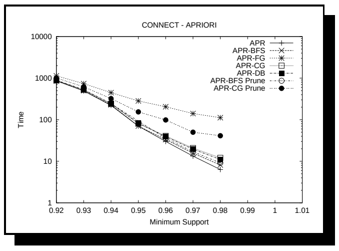
(g) Chess *Apriori* Timings



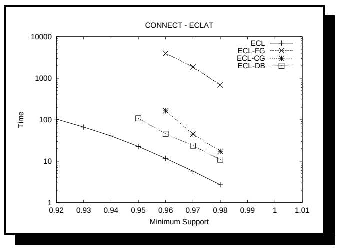
(h) Chess *Eclat* Timings



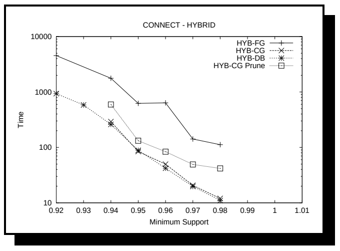
(i) Chess Hybrid Timings



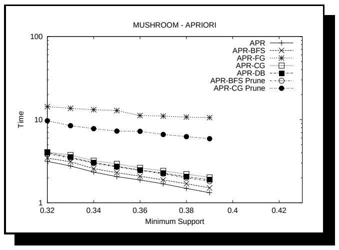
(j) Connect *Apriori* Timings



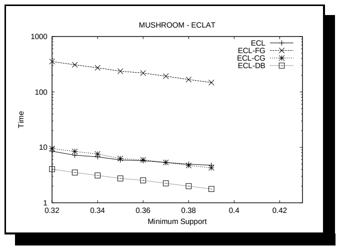
(k) Connect *Eclat* Timings



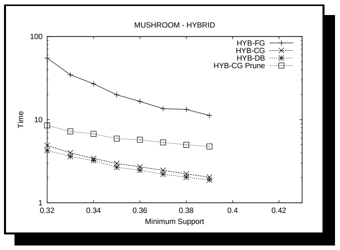
(l) Connect Hybrid Timings



(m) Mushroom *Apriori* Timings

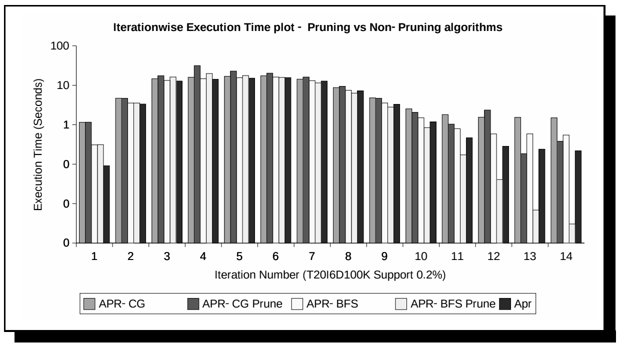


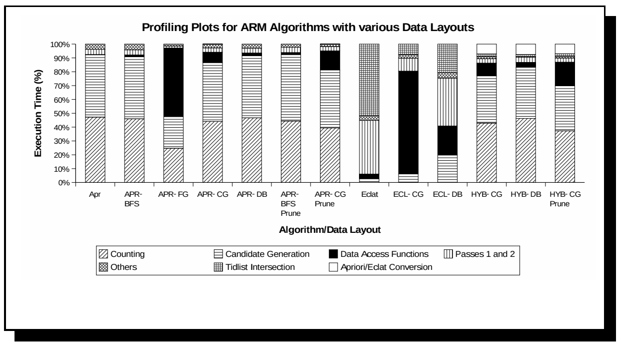
(n) Mushroom *Eclat* Timings



(o) Mushroom Hybrid Timings

Figure 1: Real and Synthetic Database Timings





(a) Iteration-wise Execution time for Pruning and Non-Pruning algorithms

(b) Profiles of *Apriori* According to Data Layout

Figure 2: Itemized Profiling and Performance Measures

trusive) 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 2 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 that counting and candidate generation tend to dominate run time in coarse grained horizontal data format algorithms. In contrast, Figure 2 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 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 that 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* (APR). Hence the expected performance of APR-BFS was close to APR. The low overhead of the cursor management in APR-DB resulted in processor time utilization comparable to APR-BFS.

4 Summary and Conclusion

We explored integrating KDD tools with database management systems with an eye toward improving the user's mining experience, eventually providing seamless systems for DBMS and KDD [14]. Our integrated schemes 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 focus on systems software 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: (i) horizontal and vertical partitioning based algorithms, *Apriori* and *Eclat* , (ii) traditional flat files, a novel BFS file which supports pruning, and indexed structures. (iii) granularity of data access (corresponding to level of normalization in a DBMS) and (iv) 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 BLOB support for native B^+-trees in Sleepycat Berkeley DB is more efficient than the B^+-tree emulation in GiST, 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 often 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 from the issues encountered during our analysis.

1. The memory requirements (for storing candidates, for example) of various approaches, at times, approached or exceeded the memory capacity of the machines. Exploring *out of core* approaches for storing candidates might not only provide a means for increasing our capacity to mine databases, but also allow ARM users to mine at lower values of *min_sup*.

2. Using existing profiling technology to diagnose where the software spends its time, provides a limited amount of information. Profiling technology, however, should permit aggregation of functions together for timing statistics, and allow the measurement of wall clock and idle time, as well. This opens up an exploration of developing profilers for data mining measurements.

3. For highly data dependent tools like KDD tools, it makes sense to automate the collection of profiling data and statistics to a form that is *mineable*. This permits queries on the collected data to correlate performance across a range of inputs. Mining of profiled data might provide feedback that allows improvements in both the compiler and KDD tool performance.

4. ARM is, but one task in KDD. It will be interesting to see how various indexing techniques and data access methods will help in performing other KDD tasks.

5 Acknowledgements

The authors would like to thank the anonymous referees for their valuable suggestions and feedback.

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

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

[6] 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.

[7] 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.

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

[9] Jochen Hipp, Ulrich Güntzer, and Gholamreza Nakhaeizadeh. Mining association rules: Deriving a superior algorithm by analysing today's approaches. In *Proceedings of the 4th European Symposium on Principles of Data Mining and Knowledge Discovery (PKDD '00)*, Lyon, France, September 13-16 2000.

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

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

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

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

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

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

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

[17] 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.

[18] G. Ramesh, W. A. Maniatty, and M. J. Zaki. Indexing and data access methods for database mining. Technical Report 01-01, Dept. of Computer Science, University at Albany, Albany, NY USA, June 2001.

[19] 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.

[20] 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.

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