On the Optimality of Association-rule Mining Algorithms

Vikram Pudi Jayant R. Haritsa *

Database Systems Lab, SERC

Indian Institute of Science

Bangalore 560012, India

Abstract

Since its introduction close to a decade ago, the problem of efficient mining of association rules on market-basket data has attracted tremendous attention. Numerous algorithms have been proposed, each one in turn claiming to outperform its predecessors on a representative set of databases. In this paper, we first focus our attention on the question of how much space remains for performance improvement over current association rule mining algorithms. Our strategy is to compare their performance against an "Oracle algorithm" that knows in advance the identities of all frequent itemsets in the database and only needs to gather their actual supports to complete the mining process. Our experimental results show that there is a substantial gap between the Oracle's performance and that of the current mining algorithms. Second, we present a new mining algorithm, called ARMOR, that is constructed by making minimal changes to the Oracle algorithm. ARMOR typically performs within a factor of two of the Oracle over both real and synthetic databases over practical ranges of support specifications.

*Contact Author: haritsa@dsl.serc.iisc.ernet.in

1 Introduction

The problem of efficiently mining "association rules" from large historical "market-basket" databases was introduced almost a decade ago, in [4]. Since then, a whole host of algorithms for addressing this problem have been proposed [4, 6, 20, 17, 11, 13, 12, 21, 3]. The latest include FP-growth [12], which utilizes a prefix-tree structure for compactly representing and processing pattern information, and VIPER [21], which organizes and processes the database on a vertical (column) basis as opposed to the more traditional horizontal (row) basis.

While the above efforts have certainly resulted in a variety of novel algorithms, each in turn claiming to outperform its predecessors on a representative set of databases, no logical end appears to be in sight. Therefore, in this paper, we focus our attention on the question of how much space remains for performance improvement over current association rule mining algorithms. Our approach is to compare their performance against an **"Oracle algorithm"** that knows *in advance* the identities of all frequent itemsets in the database and only needs to gather the actual supports of these itemsets to complete the mining process. Clearly, *any* practical algorithm will have to do at least this much work in order to generate mining rules. This "Oracle approach" permits us to clearly demarcate the maximal space available for performance improvement over the currently available algorithms. Further, it enables us to construct new mining algorithms from a completely different perspective, namely, as *minimally-altered derivatives* of the Oracle.

The environment we consider, similar to the majority of the prior art in the field, is one where the data mining system has a single processor and the pattern lengths in the database are small relative to the number of items in the database. That is, we restrict our attention to the class of *sequential bottom-up* mining algorithms.

Within the above framework, we make the following contributions:

Firstly, we show that while the notion of the Oracle is conceptually simple, its *construction* is not equally straightforward. In particular, it is critically dependent on the choice of data structures and database organizations used during the counting process. We present a carefully engineered implementation of Oracle that makes the best choices for these design parameters at each stage of the counting process. Our experimental results show that there is a considerable gap in the performance between the Oracle and existing mining algorithms.

Secondly, we present a new mining algorithm, called **ARMOR** (Association Rule Mining based on ORacle), whose structure is derived by making minimal changes to the Oracle, and is guaranteed to complete in two passes over the database. ARMOR incorporates techniques from a variety of previous algorithms such as PARTITION [20], CARMA [13], AS-CPA [15], VIPER [21] and DELTA [19]. Our empirical study shows

that ARMOR performs within a factor of two of the Oracle, over a variety of databases and practical ranges of support specifications.

Finally, an important feature of our experiments is that they include workloads where the database is large enough that the working set of the database cannot be completely stored in memory. This situation may be expected to frequently arise in data mining applications since they are typically executed on huge historical databases. However, previous performance studies have been largely conducted on databases that completely *fit in main memory*. For example, the standard experiment is one that has only 100K tuples with an average tuple width of 50 bytes – this fits easily in current memories that are typically in the hundreds of megabytes. Therefore, the ability of these algorithms to scale with database size, an important requirement for mining applications, has not been conclusively shown. In our previous work [21], we had demonstrated that this was an important issue and that algorithms that worked very well for memory-resident databases did not necessarily perform as well in disk-resident databases. Consistent with that observation, here too we conduct experiments with such large disk-resident databases.

1.1 Organization

The remainder of this paper is organized as follows: In Section 2 we present a formal description of the problem of mining association rules and the scope of our work in terms of the database and system characteristics considered in our study. The design of the Oracle algorithm is described in Section 3 and is used to evaluate the performance of current algorithms in Section 4. Our new ARMOR algorithm is presented in Section 5. The details of candidate generation in ARMOR are discussed in Section 6, while its main memory requirements are discussed in Section 7. The performance of ARMOR is evaluated in Section 8. Related work on association rule mining is reviewed in Section 9. Finally, in Section 10, we summarize the conclusions of our study and outline future avenues to explore.

2 Problem Formulation and Scope

The problem of mining market-basket databases for boolean association rules was first formulated in [4] and since then has attracted considerable attention. Since the problem is well-known, we quickly review the problem formulation here. We then move on to describe the database, system and pattern characteristics considered in our study. Our choices are such that they match those selected in the majority of the previous studies.

2.1 Basket Mining

The inputs to this model are $\mathcal{I}$, a set of items sold by the store, and $\mathcal{D}$, a database of customer purchase transactions. In this context, an *association rule* is a (statistical) implication of the form $X \Longrightarrow Y$ where $X, Y \subset \mathcal{I}$ and $X \cap Y = \phi$. The problem then, is to find all association rules for which $P(X \cup Y)/P(X) \geq minconf$ and $P(X \cup Y) \geq minsup$ where $minconf$ and $minsup$ are user-specified parameters. It has been observed in [4] that this problem is effectively reducible to finding all sets of items (also called *frequent itemsets*), X, for which $P(X) \geq minsup$. $P(X)$ is also referred to as the *support* of X.

2.2 Database Characteristics

Conceptually, a market-basket database is a two-dimensional matrix where the rows represent individual customer purchase transactions and the columns represent the items on sale. This matrix can be implemented in the following four different ways [21], which are pictorially shown in Figure 1:

Item-vector (IV): The database is organized as a set of rows with each row storing a transaction identifier (TID) and a bit-vector of 1's and 0's to represent for each of the items on sale, its presence or absence, respectively, in the transaction.

Item-list (IL): This is similar to IV, except that each row stores an ordered list of item-identifiers (IID), representing only the items *actually* purchased in the transaction.

Tid-vector (TV): The database is organized as a set of columns with each column storing an IID and a bit-vector of 1's and 0's to represent the presence or absence, respectively, of the item in the set of customer transactions.

Tid-list (TL): This is similar to TV, except that each column stores an ordered list of only the TIDs of the transactions in which the item was purchased.

While a mining algorithm is free to dynamically change the database layout during the mining process, we assume that the *initial* database is always provided in the horizontal item-list (IL) format.

2.3 System Characteristics

While there has been significant work in designing algorithms for the parallel mining of association rules [5, 11, 29, 18], in this study we focus on single processor environments. We also assume that the database is much larger than the available main memory.

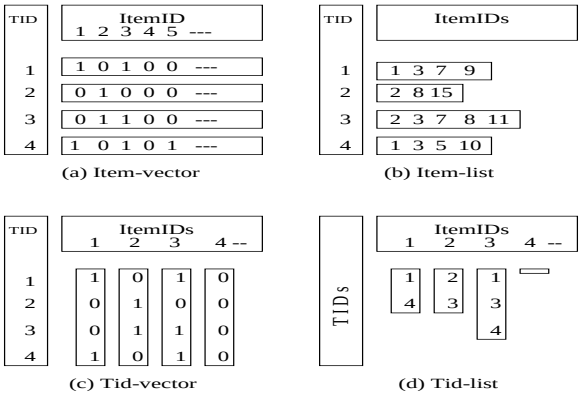

Figure 1: **Comparison of Data Layouts**

2.4 Pattern Characteristics

Boolean Association Rules We restrict our attention to the problem of generating *boolean* association rules where the only relevant information in each database transaction is the presence or absence of an item. Previous works on generating hierarchical [22], quantitative and categorical rules [23] have shown that albeit requiring some preprocessing, these problems are finally reducible to the problem of generating boolean association rules.

Short Patterns The environment we consider is one where the pattern lengths in the database are small relative to the number of items in the database. That is, we restrict our attention to the class of algorithms that take a *bottom-up* approach to enumerate the solution space consisting of the lattice of all possible itemsets. The problem of mining long patterns has been addressed in [7, 14, 2] and solution techniques for such patterns require a combination of top-down and bottom-up methods to be viable.

2.5 Mining Algorithms Input/Output

All *online* mining algorithms in our study take as input the database $\mathcal{D}$ in item-list (IL) format and the minimum support threshold $minsup$ and produce as output the set of frequent itemsets F and its negative border N [25] along with their corresponding supports. The negative border N of a set of itemsets F is defined as follows: An itemset X belongs to N iff $X \notin F$ but all subsets of X are in F.

The Oracle algorithm, on the other hand, takes as input the database $\mathcal{D}$ in item-list (IL) format, the set of frequent itemsets F and its negative border N, and produces as output the supports of itemsets in $F \cup N$. We include the negative border supports as a required output of Oracle due to the following reasons:

- It has been shown in [16] that in certain restricted models of computation all the itemsets in the negative

border have to be examined. In particular, it was shown that:

Theorem 2.1 *Any algorithm that computes the set of frequent itemsets and accesses the data using only queries of the following form: "Is itemset X frequent?" must use at least $|N|$ such queries.*

- The negative border information has been found to be especially useful in the design of *incremental* mining algorithms [19, 24, 10]. These algorithms are designed to efficiently derive the current mining output by utilizing previous mining results when a database has been updated with an increment.

For ease of exposition, we will use the notation shown in Table 1 in the remainder of this paper.

Mining Algorithms Input/Output	
$\mathcal{I}$	Set of items in the database
$\mathcal{D}$	Database of customer purchase transactions
$minsup$	User-specified minimum rule support
F	Set of frequent itemsets in $\mathcal{D}$
N	Set of itemsets in the negative border of F
For Oracle and ARMOR Algorithms	
$P_1, P_2, ..., P_n$	Set of n disjoint partitions of $\mathcal{D}$
d	No of transactions in partitions scanned so far during algorithm execution *excluding* the current partition
d^+	No of transactions in partitions scanned so far during algorithm execution *including* the current partition
$\mathcal{G}$	DAG structure to store candidates during algorithm execution

Table 1: **Notation**

3 The Oracle Algorithm

In this section we present the Oracle algorithm which, as mentioned in the Introduction, "magically" knows in advance the identities of all frequent itemsets in the database and only needs to gather the actual supports of these itemsets. Clearly, *any* practical algorithm will have to do at least this much work in order to generate mining rules. Oracle takes as input the database, $\mathcal{D}$, the set of frequent itemsets, F, and its corresponding negative border, N, and outputs the supports of these itemsets by making *one scan* over the database. While the initial database layout is in the item-list (IL) format, the Oracle algorithm uses different formats during the course of its execution for efficient processing. We first describe the mechanics of the Oracle algorithm below and then move on to discuss the rationale behind its design choices in Section 3.2.

3.1 The Mechanics of Oracle

For ease of exposition, we first present the manner in which Oracle computes the supports of 1-itemsets and 2-itemsets and then move on to longer itemsets. Note, however, that the algorithm actually performs all these computations *concurrently* in one scan over the database.

3.1.1 Counting Singletons and Pairs

```
ArrayCount (T, A1, A2)
Input: Transaction T, Array for 1-itemsets A1, Array for 2-itemsets A2
Output: Arrays A1 and A2 with their counts updated over T
1.      Itemset T^f = null; // to store frequent items from T in Item-List format
2.      for each item i in transaction T
3.              A1[i.id].count++;
4.              if A1[i.id].index ≠ null
5.                      append i to T^f
6.      for j = 1 to |T^f| // enumerate 2-itemsets
7.              for k = j + 1 to |T^f|
8.                      index1 = A1[T^f[j].id].index // row index
9.                      index2 = A1[T^f[k].id].index // column index
10.                     A2[index1, index2]++;
```

Figure 2: **Counting Singletons and Pairs in Oracle**

Data-Structure Description The counters of singletons (1-itemsets) are maintained in a 1-dimensional lookup array, $\mathcal{A}_1$, and that of pairs (2-itemsets), in a lower triangular 2-dimensional lookup array, $\mathcal{A}_2$. (Similar arrays are also used in Apriori [6, 22] for its first two passes.) The kth entry in the array $\mathcal{A}_1$ contains two fields: (1) $count$, the counter for the itemset X corresponding to the kth item, and (2) $index$, the number of frequent itemsets prior to X in $\mathcal{A}_1$, if X is frequent; **null**, otherwise.

Algorithm Description The ArrayCount function shown in Figure 2 takes as inputs, a transaction T along with $\mathcal{A}_1$ and $\mathcal{A}_2$, and updates the counters of these arrays over T. In the ArrayCount function, the individual items in the transaction T are enumerated (lines 2–5) and for each item, its corresponding count in $\mathcal{A}_1$ is incremented (line 3). During this process, the frequent items in T are stored in a separate itemset T^f (line 5). We then enumerate all pairs of items contained in T^f (lines 6–10) and increment the counters of the corresponding 2-itemsets in $\mathcal{A}_2$ (lines 8–10).

3.1.2 Counting k-itemsets, $k > 2$

Data-Structure Description Itemsets in $F \cup N$ of length greater than 2 and their related information (counters, etc.) are stored in a DAG structure $\mathcal{G}$, which is pictorially shown in Figure 3 for a database with items $\{$A, B, C, D$\}$. Although singletons and pairs are stored in lookup arrays, as mentioned before, for expository ease, we assume that they too are stored in $\mathcal{G}$ in the remainder of this discussion.

Each itemset is stored in a separate node of $\mathcal{G}$ and is linked to the first two (in a lexicographic ordering) of its subsets. We use the terms "mother" and "father" of an itemset to refer to the (lexicographically) smaller subset and the (lexicographically) larger subset, respectively. E.g., $\{$A, B$\}$ and $\{$A, C$\}$ are the mother and father respectively of $\{$A, B, C$\}$. For each itemset X in $\mathcal{G}$, we also store with it links to those supersets of X for which X is a mother. We call this list of links as *childset*.

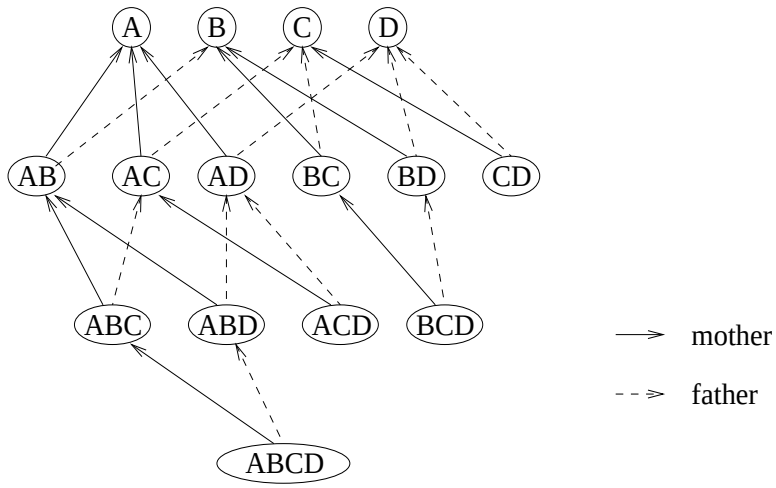

Figure 3: **DAG Structure Containing Power Set of $\{$A,B,C,D$\}$**

Since each itemset is stored in a separate node in the DAG, we use the terms "itemset" and "node" interchangeably in the remainder of this discussion. Also, we use $\mathcal{G}$ to denote the set of itemsets that are stored in the DAG structure $\mathcal{G}$.

Algorithm Description We use a *partitioning scheme* [20] wherein the database is logically divided into n disjoint horizontal partitions $P_1, P_2, ..., P_n$. In this scheme, itemsets being counted are enumerated only at the *end of each partition* and not after every tuple. Each partition is as large as can fit in available main memory. For ease of exposition, we assume that the partitions are equi-sized. However, the technique is easily extendible to arbitrary partition sizes.

The pseudo-code of Oracle is shown in Figure 4 and operates as follows: The ReadNextPartition function (line 3) reads tuples from the next partition and simultaneously creates tid-lists (within that partition) of

```
Oracle (D, G)
Input: Database D, Itemsets to be Counted G = F ∪ N
Output: Itemsets in G with Supports
1.      n = Number of Partitions
2.      for i = 1 to n
3.              ReadNextPartition(P_i, G);
4.              for each singleton X in G
5.                      Update(X);
```

Figure 4: **The Oracle Algorithm**

```
Update (M)
Input: DAG Node M
Output: M and its Descendents with Counts Updated
1.      B = convert M.tidlist to Tid-vector format // B is statically allocated
2.      for each node X in M.childset
3.              X.tidlist = Intersect(B, X.father.tidlist);
4.              X.count += |X.tidlist|
5.      for each node X in M.childset
6.              Update(X);
```

Figure 5: **Updating Counts**

```
Intersect (B, T)
Input: Tid-vector B, Tid-list T
Output: B ∩ T
1.      Tid-list result = φ
2.      for each tid in T
3.              offset = tid + 1− (tid of first transaction in current partition)
4.              if B[offset] = 1 then
5.                      result = result ∪ tid
6.      return result
```

Figure 6: **Intersection**

singleton itemsets in $\mathcal{G}$. The Update function (line 5) is then applied on each singleton in $\mathcal{G}$. This function takes a node M in $\mathcal{G}$ as input and updates the counts of all descendants of M to reflect their counts over the current partition. The count of any itemset within a partition is equal to the length of its corresponding tidlist (within that partition). The tidlist of an itemset can be obtained as the intersection of the tidlists of its mother and father and this process is started off using the tidlists of frequent 1-itemsets. The exact details of tidlist computation are discussed later.

We now describe the manner in which the itemsets in $\mathcal{G}$ are enumerated after reading in a new partition. The set of links, $\bigcup_{M \in \mathcal{G}} M.childset$, induce a spanning tree of $\mathcal{G}$ (e.g. consider only the solid edges in Fig-

ure 3). We perform a *depth first search* on this spanning tree to enumerate all its itemsets. When a node in the tree is visited, we compute the tidlists of all its children. This ensures that when an itemset is visited, the tidlists of its mother and father have already been computed.

The above processing is captured in the function Update whose pseudo-code is shown in Figure 5. Here, the tidlist of a given node M is first converted to the tid-vector (TV) format (line 1) discussed in Section 2.2. Then, tidlists of all children of M are computed (lines 2–4) after which the same children are visited in a depth first search (lines 5–6).

The mechanics of tidlist computation, as promised earlier, are given in Figure 6. The Intersect function shown here takes as input a tid-vector B and a tid-list T. Each tid in T is added to the result if $B[offset]$ is 1 (lines 2–5) where $offset$ is defined in line 3 and represents the position of the transaction T relative to the current partition.

3.2 Rationale for the Oracle Design

Having described the mechanics of the Oracle design, we now move on to providing the rationale for its construction. We show that it is optimal in two respects: (1) It enumerates only those itemsets in $\mathcal{G}$ that need to be enumerated, and (2) The enumeration is performed in the most efficient way possible. The following theorem shows that there is *no wasted enumeration* of itemsets in Oracle in typical mining scenarios.

Theorem 3.1 *If the size of each partition is large enough that every itemset in $F \cup N$ of length greater than 2 is present at least once in it, then the only itemsets being enumerated in the Oracle algorithm are those whose counts need to be incremented in that partition.*

Proof: The first observation is that *all* 1-itemsets must be in either F or N. Hence every occurance of a 1-itemset in the entire database needs to be accounted for in the final output. Oracle does no more than this as it enumerates each singleton in every transaction only once (lines 2–5 in Figure 2).

The 2-itemsets that are enumerated (lines 6–10 in Figure 2) are all guaranteed to be either in F or in N since only combinations of frequent 1-itemsets are considered. Hence there is no wasted work in enumerating them.

If each partition is large enough that every itemset in $F \cup N$ of length greater than 2 is present at least once in it, then it is *necessary* to increment the counts of all these itemsets over that partition. This is precisely what is done in Oracle. Also, note that by the definition of depth first search, each node in the DAG is visited *only once*. Hence, it follows that there is no wasted enumeration of itemsets in Oracle. □

The assumption in Theorem 3.1 that every itemset in $F \cup N$ of length greater than 2 is present *at least once* in each partition would typically hold on large partitions. Even if this does not strictly hold, the Oracle algorithm degrades gracefully in that: If there are m itemsets that are not present in some partition, then the amount of wasted enumeration is only m.

We now move on to the second part of our proof, namely, to show that the data-structures used in the Oracle algorithm are the most efficient for the range of operations required in Oracle.

Theorem 3.2 *The cost of enumerating each itemset in Oracle is* $\Theta(1)$.

Proof: Since the counts of singletons and pairs are stored in direct lookup arrays, the cost of finding the counters of an arbitrary singleton or pair is $\Theta(1)$.

For an itemset X such that $|X| \geq 2$, the cost of enumerating its children is $\Theta(|X.childset|)$ since links to all nodes in $X.childset$ are available in the node containing X. Amortizing this cost over all the children results in $\Theta(1)$ cost per child. Also, X has direct pointers to its mother and father. Hence the cost of finding them in order to compute the tid-list of X is $\Theta(1)$.

Since the only operations done in Oracle in each visit to a node during the depth first search are to compute the tidlists of each of its children, the amortized cost incurred for enumerating each node is $\Theta(1)$. $\Box$

We assume that the underlying computing model is a unit cost RAM [9]. In this model, operations such as accessing an arbitrary element in an array and following a pointer have unit cost and cannot therefore be improved upon. Since the costs involved in the above proof are of array lookups and following pointers, the constant factor involved in the $\Theta(1)$ expression is *tight*.

While Oracle is optimal in most respects as described above, we note that there may remain some scope for improvement in the details of *tidlist computation*. That is, the Intersect function (Figure 6) which computes the intersection of a tid-vector B and a tid-list T requires $\Theta(|T|)$ operations. B itself was originally constructed from a tid-list, although this cost is amortized over many calls to the Intersect function. We plan to investigate in our future work whether the intersection of two sets can, in general, be computed more efficiently – for example, using **diffsets**, a novel and interesting approach suggested in [27]. The diffset of an itemset X is the set-difference of the tid-list of X from that of its mother. Diffsets can be easily incorporated in Oracle – only the Update function in Figure 5 of Section 3 is to be changed to compute diffsets instead of tidlists by following the techniques suggested in [27].

3.2.1 Advantages of Partitioning Schemes

Oracle, as discussed above, uses a partitioning scheme. An alternative commonly used in current association rule mining algorithms, especially in hashtree [6] based schemes, is to use a tuple-by-tuple approach. A problem with the tuple-by-tuple approach, however, is that there is considerable wasted enumeration of itemsets. The core operation in these algorithms is to determine all candidates that are subsets of the current transaction. Given that a frequent itemset X is present in the current transaction, we need to determine all candidates that are immediate supersets of X and are also present in the current transaction. In order to achieve this, it is often necessary to enumerate and check for the presence of many more candidates than those that are actually present in the current transaction.

4 Performance Study

In the previous section, we have described the Oracle algorithm. In order to assess the performance of current mining algorithms with respect to the Oracle algorithm, we have chosen VIPER [21] and FP-growth [12], among the latest in the suite of online mining algorithms. For completeness and as a reference point, we have also included the classical Apriori in our evaluation suite.

Our experiments cover a range of database and mining workloads, and include the typical and extreme cases considered in previous studies – the only difference is that we also consider database sizes that are *significantly larger* than the available main memory. The performance metric in all the experiments is the *total execution time* taken by the mining operation.

The databases used in our experiments were synthetically generated using the technique described in [6] and attempt to mimic the customer purchase behavior seen in retailing environments. The parameters used in the synthetic generator and their default values are described in Table 2. In particular, we consider databases with parameters T10.I4, T20.I12 and T40.I8 with 10 million tuples in each of them.

Parameter	Meaning	Default Values
N	Number of items	1000
T	Mean transaction length	10, 20, 40
L	Number of potentially frequent itemsets	2000
I	Mean length of potentially frequent itemsets	4, 8, 12
D	Number of transactions in the database	10M

Table 2: **Parameter Table**

We set the rule support threshold values to as low as was feasible with the available main memory. At

these low support values the number of frequent itemsets exceeded twenty five thousand! Beyond this, we felt that the number of rules generated would be enormous and the purpose of mining – to find interesting patterns – would not be served. In particular, we set the rule support threshold values for the T10.I4, T20.I12 and T40.I8 databases to the ranges (0.1%–2%), (0.4%–2%) and (1.15%–5%), respectively.

Our experiments were conducted on a 700-MHz Pentium III workstation running Red Hat Linux 6.2, configured with a 512 MB main memory and a local 18 GB SCSI 10000 rpm disk. For the T10.I4, T20.I12 and T40.I12 databases, the associated database sizes were approximately 500MB, 900MB and 1.7 GB, respectively. All the algorithms in our evaluation suite are written in C++. We implemented a basic version of the FP-growth algorithm wherein we assume that the entire FP-tree data structure fits in main memory. Finally, the partition size in Oracle was fixed to be 20K tuples.

4.1 Experimental Results for Current Mining Algorithms

We now report on our experimental results. We conducted two experiments to evaluate the performance of current mining algorithms with respect to the Oracle. Our first experiment was run on large (10M tuples) databases, while our second experiment was run on small (100K tuples) databases.

4.1.1 Experiment 1: Performance of Current Algorithms

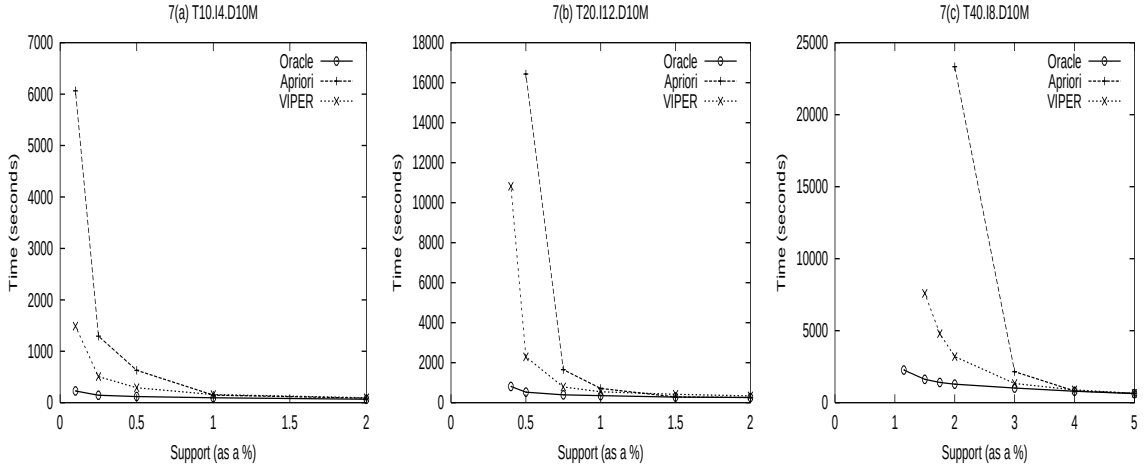

Figure 7: **Performance of Current Algorithms (Large Databases)**

In our first experiment, we evaluated the performance of Apriori, VIPER and Oracle algorithms for the T10.I4, T20.I12 and T40.I8 databases each containing 10M transactions and these results are shown in Figures 7a–c. The x-axis in these graphs represent the support threshold values while the y-axis represents the response times of the algorithms being evaluated.

In these graphs, we see that the response times of all algorithms increase exponentially as the support threshold is reduced. This is only to be expected since the number of itemsets in the output, $F \cup N$, increases exponentially with decrease in the support threshold.

We also see that there is a considerable gap in the performance of both Apriori and VIPER with respect to Oracle. For example, in Figure 7a, at a support threshold of 0.1%, the response time of VIPER is more than 6 times that of Oracle whereas the response time of Apriori is more than 26 times!

In this experiment, we could not evaluate the performance of FP-growth because it did not complete in any of our runs on large databases due to its heavy and database size dependent utilization of main memory. The reason for this is that FP-growth stores the database itself in a condensed representation in a data structure called FP-tree. In [12], the authors briefly discuss the issue of constructing disk-resident FP-trees. We however, did not take this into account in our implementation. We return to this issue later in Section 4.1.2.

4.1.2 Experiment 2: Small Databases

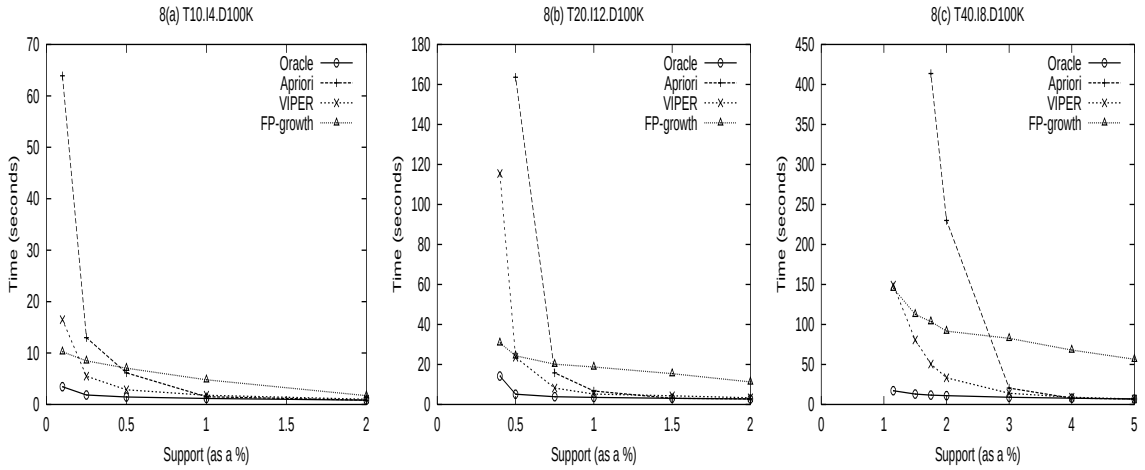

Figure 8: **Performance of Current Algorithms (Small Databases)**

Since, as mentioned above, it was not possible for us to evaluate the performance of FP-growth on large databases due to its heavy utilization of main memory, we evaluated the performance of FP-growth and other current algorithms on small databases consisting of 100K transactions. The results of this experiment are shown in Figures 8a–c, which correspond to the T10.I4, T20.I12 and T40.I8 databases, respectively.

In these graphs, we first see there continues to be a considerable gap in the performance of current mining algorithms with respect to Oracle. For example, for the T40.I8 database, the response time of FP-growth is more than 8 times that of Oracle for the entire support threshold range.

Second, although FP-growth does well at low supports, its performance is worse than Apriori for high

supports. These results are inconsistent with those shown in [12] where it was shown that FP-growth consistently performs better than Apriori for the entire support range. While this could possibly be due to differences between our respective implementations of FP-growth and/or Apriori, we feel that there are logical reasons for this behaviour as explained below.

At high supports Apriori typically performs only two passes over the data since with these supports there are usually no frequent itemsets of length greater than two. In these cases, the first pass of Apriori is *identical* to the preprocessing pass in FP-growth in which all frequent singletons are obtained. The second pass of Apriori is quite efficient since the counts of candidate 2-itemsets are maintained in a 2-dimensional lookup array. FP-growth, on the other hand, constructs an FP-tree during the second pass. The FP-tree is updated on a tuple-by-tuple basis. Each node in the FP-tree contains an $item\text{-}name$ field. A critical operation during FP-tree construction is to find the child of a node given a key $item\text{-}name$. If these keys are stored in lookup-arrays, the memory requirements of FP-tree would be still worse. The alternative is to use an indexing data structure such as a red-black tree or a skip-list that requires $\mathcal{O}(\log n)$ time to perform the find operation, but this would make the FP-tree construction slow. Even assuming that the cost of FP-tree construction is equal to the second pass of Apriori, FP-growth still needs to mine the FP-tree. Hence FP-growth finally loses out at high supports.

5 The ARMOR Algorithm

In the previous section, our experimental results have shown that there is a considerable gap in the performance between the Oracle and existing mining algorithms. We now move on to describe our new mining algorithm, ARMOR (Association Rule Mining based on ORacle). In this section, we overview the main features and the flow of execution of ARMOR – the details of candidate generation are deferred to the following section.

The guiding principle in our design of the ARMOR algorithm is that we consciously make an attempt to determine the *minimal amount of change* to Oracle required to result in an online algorithm. This is in marked contrast to the earlier approaches which designed new algorithms by trying to address the limitations of *previous* online algorithms. That is, we approach the association rule mining problem from a completely different perspective.

In ARMOR, as in Oracle, the database is conceptually partitioned into n disjoint blocks $P_1, P_2, ..., P_n$. At most *two* passes are made over the database. In the first pass we form a set of candidate itemsets, $\mathcal{G}$, that is guaranteed to be a superset of the set of frequent itemsets. During the first pass, the counts of candidates

```
ARMOR (D, I, minsup)
Input: Database D, Set of Items I, Minimum Support minsup
Output: F ∪ N with Supports
1.      n = Number of Partitions

        //—— First Pass ——
2.      G = I      // candidate set (in a DAG)
3.      for i = 1 to n
4.              ReadNextPartition(P_i, G);
5.              for each singleton X in G
6.                      X.count += |X.tidlist|
7.                      Update1(X, minsup);

        //—— Second Pass ——
8.      RemoveSmall(G, minsup);
9.      OutputFinished(G, minsup);
10.     for i = 1 to n
11.             if (all candidates in G have been output)
12.                     exit
13.             ReadNextPartition(P_i, G);
14.             for each singleton X in G
15.                     Update2(X, minsup);
```

Figure 9: **The ARMOR Algorithm**

in $\mathcal{G}$ are determined over each partition in exactly the same way as in Oracle by maintaining the candidates in a DAG structure. The 1-itemsets and 2-itemsets are stored in lookup arrays as in Oracle. But unlike in Oracle, candidates are inserted and removed from $\mathcal{G}$ at the end of each partition. Generation and removal of candidates is done *simultaneously* while computing counts. The details of candidate generation and removal during the first pass are described in Section 6. For ease of exposition we assume in the remainder of this section that all candidates (including 1-itemsets and 2-itemsets) are stored in the DAG.

Along with each candidate X, we also store the following three integers as in the CARMA algorithm [13]: (1) $X.count$: the number of occurrences of X *since* X was last inserted in $\mathcal{G}$. (2) $X.firstPartition$: the index of the partition *at* which X was inserted in $\mathcal{G}$. (3) $X.maxMissed$: upper bound on the number of occurrences of X *before* X was inserted in $\mathcal{G}$.

While the CARMA algorithm works on a tuple-by-tuple basis, we have adapted the semantics of these fields to suit the partitioning approach. If the database scanned so far is d (refer Table 1), then the support of any candidate X in $\mathcal{G}$ will lie in the range $[X.count/|d|, (X.maxMissed + X.count)/|d|]$ [13]. These bounds are denoted by minSupport(X) and maxSupport(X), respectively. We define an itemset X to be d-frequent if minSupport$(X) \geq minsup$. Unlike in the CARMA algorithm where only d-frequent itemsets are stored at any stage, the DAG structure in ARMOR contains other candidates, including the *negative*

border of the d-frequent itemsets, to ensure efficient candidate generation. The details are given in Section 6.

At the end of the first pass, the candidate set $\mathcal{G}$ is pruned to include only d-frequent itemsets and their negative border. The counts of itemsets in $\mathcal{G}$ over the entire database are determined during the second pass. The counting process is again identical to that of Oracle. No new candidates are generated during the second pass. However, candidates may be removed. The details of candidate removal in the second pass is deferred to Section 6.1.

The pseudo-code of ARMOR is shown in Figure 9 and is explained below.

5.1 First Pass

At the beginning of the first pass, the set of candidate itemsets $\mathcal{G}$ is initialized to the set of singleton itemsets (line 2). The ReadNextPartition function (line 4) reads tuples from the next partition and simultaneously creates tid-lists of singleton itemsets in $\mathcal{G}$.

After reading in the entire partition, the Update1 function (details in Section 6) is applied on each singleton in $\mathcal{G}$ (lines 5–7). It increments the counts of existing candidates by their corresponding counts in the current partition. It is also responsible for generation and removal of candidates.

At the end of the first pass, $\mathcal{G}$ contains a superset of the set of frequent itemsets. For a candidate in $\mathcal{G}$ that has been inserted at partition P_j, its count over the partitions $P_j, ..., P_n$ will be available.

5.2 Second Pass

At the beginning of the second pass, candidates in $\mathcal{G}$ that are neither d-frequent nor part of the current negative border are removed from $\mathcal{G}$ (line 8). For candidates that have been inserted in $\mathcal{G}$ at the first partition, their counts over the entire database will be available. These itemsets with their counts are output (line 9). The OutputFinished function also performs the following task: If it outputs an itemset X and X has no supersets left in $\mathcal{G}$, X is removed from $\mathcal{G}$.

During the second pass, the ReadNextPartition function (line 13) reads tuples from the next partition and creates tid-lists of singleton itemsets in $\mathcal{G}$. After reading in the entire partition, the Update2 function (details in Section 6.1) is applied on each singleton in $\mathcal{G}$ (lines 14–15). Finally, before reading in the next partition we check to see if there are any more candidates. If not, the mining process terminates.

6 Candidate Generation in ARMOR

ARMOR utilizes a technique from *incremental* mining algorithms [19, 24, 10] in order to generate candidates efficiently. These algorithms are designed to efficiently derive the current mining output by utilizing previous mining results when a database has been updated with an increment. ARMOR treats the database scanned so far, d, as the "original database" and the current partition being processed as the "increment". Let d^+ denote the portion of the database scanned so far *including* the current partition being processed (see Table 1 in Section 2). Let F^d and F^{d^+} be the sets of frequent itemsets over d and d^+, respectively, and N^d and N^{d^+} be their corresponding negative borders. In this context, it is shown in [24] that:

Theorem 6.1 *If X is an itemset that is not in F^d but is in F^{d^+}, then there must be some subset x of X which was in N^d and is now in F^{d^+}.*

The itemsets that move from N^d to F^{d^+} are called *promoted borders*. The above Theorem then means that the only candidates that need to be generated are those that are supersets of the promoted borders. We use the term *expanding* a promoted border P to denote the process of generating the required supersets of P.

We present now a technique for efficiently expanding a promoted border. Our technique is captured in the Expand function presented in Figure 10, the inputs to which are P, the promoted border to be expanded and $\mathcal{G}$, the current set of candidates. The Expand function is similar to the AprioriGen function [6] since the siblings of P are exactly those itemsets in $\mathcal{G}$ that differ from P in the last item. However, the Expand function and its usage differs from AprioriGen in that: (1) It is applied dynamically whenever a candidate that was in the negative border becomes d-frequent; (2) It is applied to individual itemsets, whereas the AprioriGen function is applied to sets of itemsets; (3) It performs a parent based pruning optimization unlike AprioriGen which enumerates all immediate subsets of a candidate inorder to prune it.

At first glance, it may appear surprising that we do not consider the same pruning strategy as of AprioriGen in Expand. The reason we do not do so is because it results in significant overheads due to the dynamic and incremental manner in which candidate generation occurs in Expand. We illustrate this with the following example: Consider the situation in which the itemsets $\{U, V\}$ and $\{U, W\}$ are d-frequent but $\{V, W\}$ is not. Then $\{U, V, W\}$ will not be in $\mathcal{G}$ if Apriori-type pruning were incorporated in Expand. If $\{V, W\}$ also becomes d-frequent, then $\{U, V, W\}$ will need to be added to $\mathcal{G}$. But $\{V, W\}$ cannot be combined with another itemset that differs only in the *last* item to produce $\{U, V, W\}$. This means that if we incorporate Apriori-type pruning, the Expand function needs to combine $\{V, W\}$ with itemsets that differ from it in *any one* item.

From the above discussion, it is clear that incorporating Apriori-type pruning in the Expand function, results in significant cost for two reasons: (1) It requires a separate traversal of the DAG structure to find all itemsets that differ from a given itemset in *any one* item. (2) All immediate subsets of a given itemset need to be searched for in the DAG.

Without Apriori-type pruning, in the above example, $\{U, V, W\}$ would have already been in $\mathcal{G}$ regardless of whether $\{V, W\}$ is d-frequent or not since it would not have been pruned due to the absence of $\{V, W\}$. Therefore, when $\{V, W\}$ becomes d-frequent, it is not necessary to regenerate $\{U, V, W\}$.

Due to the above reasons we do not incorporate Apriori-type pruning in ARMOR. Instead, a candidate is automatically pruned if one of its parents is not d-frequent since it would not even be generated in the first place. Our experiments (Section 8) showed that the number of additional candidates generated in ARMOR compared to Apriori's $|F \cup N|$ candidates was marginal – the worst case being about *ten percent* more.

The Expand function is incorporated into ARMOR by calling it from the Update1 function that is invoked for each partition scanned during the first pass. The Update1 function is presented in Figure 11 and is explained below.

```
Expand (P, G)
Input: Promoted Border P, DAG G
        for each sibling X of P in G
                if (X is d-frequent) then
                        S = P ∪ X        // new candidate
                        Insert S into G as a child of P
```

Figure 10: **Expanding a Promoted Border**

```
Update1 (M, minsup)
Input: DAG Node M, Minimum Support minsup
Output: M and its Descendents Updated
// Lines 1–4 of Update function shown in Figure 5 and explained in Section 3
1.      for each node X in M.childset
2.              if maxSupport(X) ≤ minsup then
3.                      if |X.childset| > 0      // already expanded
4.                              remove all supersets of X reachable from X in the DAG
5.              else
6.                      if |X.childset| = 0      // not yet expanded
7.                              Expand(X);
// Lines 5–6 of Update function shown in Figure 5 and explained in Section 3
```

Figure 11: **Updating Counts**

The manner in which the counts of candidates are computed in Update1 is exactly the same as that in Update (described in Section 3). The extra processing in Update1 is only to generate and remove candidates dynamically. This is done in one enumeration of all children of a given node M (lines 1–7). For each child X that is enumerated, if it has supersets but is not d-frequent, then we remove all supersets of X that are reachable from X in the DAG (lines 2–4). Note that X itself is not removed since it could be part of the current negative border. On the other hand, if X is d-frequent and has not yet been expanded, then it is now expanded by calling the Expand function (lines 6–7).

6.1 Candidate Removal During Second Pass

A candidate X is removed during the second pass whenever the following two conditions are satisfied: (1) The count of X over the entire database is available, which becomes true when $X.firstPartition$ is the next partition to be processed; and (2) X has no supersets in $\mathcal{G}$.

We now describe the Update2 function (called from ARMOR in Figure 9), which is responsible for removing candidates as described above. The Update2 function increments the counts of existing candidates by their corresponding counts in the current partition in a manner identical to that of the Update function of Oracle (described in Section 3). It differs from Update only in that it also outputs candidates whose counts over the entire database are known. If it outputs an itemset X and X has no supersets left in $\mathcal{G}$, X is removed from $\mathcal{G}$.

7 Memory Utilization in ARMOR

In the design and implementation of ARMOR, we have opted for speed in most decisions that involve a space-speed tradeoff. Therefore, the main memory utilization in ARMOR is certainly more as compared to algorithms such as Apriori. However, in the following discussion, we show that the memory usage of ARMOR is well within the reaches of current machine configurations. This is also experimentally confirmed in the next section.

The main memory consumption of ARMOR comes from the following sources: (1) The 1-d and 2-d arrays for storing counters of singletons and pairs, respectively; (2) The DAG structure for storing counters of longer itemsets, including tidlists of those itemsets, and (3) The current partition.

The total number of entries in the 1-d and 2-d arrays and in the DAG structure corresponds to the number of candidates in ARMOR, which as we have discussed in Section 6, is only marginally more than $|F \cup N|$. For the moment, if we disregard the space occupied by tidlists of itemsets, then the amortized amount of

space taken by each candidate is a small constant (about 10 integers for the dag and 1 integer for the arrays). E.g., if there are 1 million candidates in the dag and 10 million in the array, the space required is about 80MB. Since the environment we consider is one where the pattern lengths are small, the number of candidates will typically be comparable to or well within the available main memory. [26] discusses alternative approaches when this assumption does not hold.

Regarding the space occupied by tidlists of itemsets, note that ARMOR only needs to store tidlists of d-frequent itemsets. The number of d-frequent itemsets is of the same order as the number of frequent itemsets, $|F|$. The total space occupied by tidlists while processing partition P_i is then bounded by $|F| \times |P_i|$ integers. E.g., if $|F| = 5K$ and $|P_i| = 20K$, then the space occupied by tidlists is bounded by about 400MB. We assume $|F|$ to be in the range of a few thousands at most because otherwise the total number of rules generated would be enormous and the purpose of mining would not be served. Note that the above bound is very pessimistic. Typically, the lengths of tidlists are much smaller than the partition size, especially as the itemset length increases.

Main memory consumed by the current partition is small compared to the above two factors. E.g., If each transaction occupies 1KB, a partition of size 20K would require only 20MB of memory. Even in these extreme examples, the total memory consumption of ARMOR is 500MB, which is acceptable on current machines.

Therefore, *in general we do not expect memory to be an issue* for mining market-basket databases using ARMOR. Further, even if it does happen to be an issue, it is easy to modify ARMOR to free space allocated to tidlists at the expense of time: $M.tidlist$ can be freed after line 3 in the Update function shown in Figure 5.

A final observation to be made from the above discussion is that the main memory consumption of ARMOR is proportional to the size of the *output* and does not "explode" as the input problem size increases.

8 Experimental Results for ARMOR

We evaluated the performance of ARMOR with respect to Oracle on a variety of databases and support characteristics. We now report on our experimental results for the same performance model described in Section 4.

8.1 Experiment 3: Performance of ARMOR

In this experiment, we evaluated the response time performance of the ARMOR and Oracle algorithms for the T10.I4, T20.I12 and T40.I8 databases each containing 10M transactions and these results are shown in

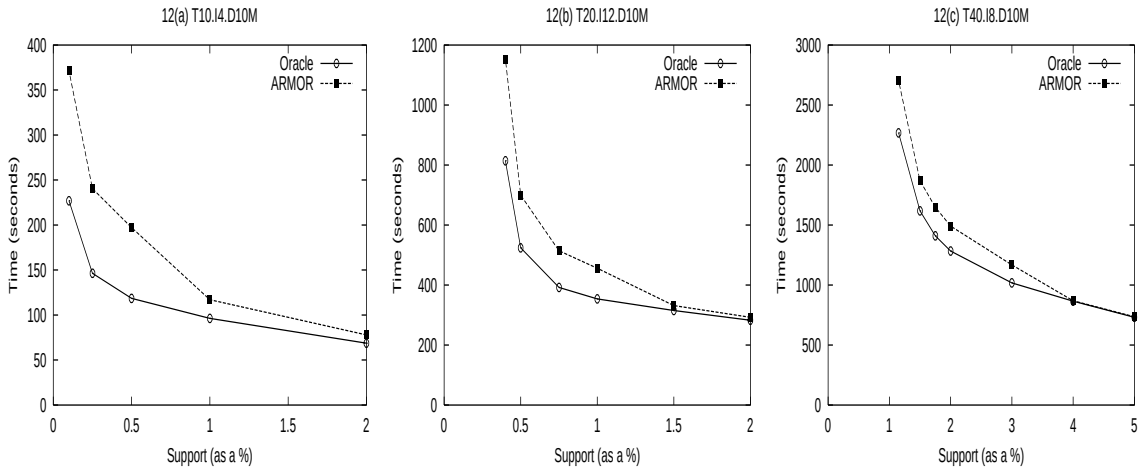

Figure 12: **Performance of ARMOR (Synthetic Datasets)**

Figures 12a–c.

In these graphs, we first see that ARMOR's performance is close to that of Oracle for high supports. This is because of the following reasons: The density of the frequent itemset distribution is sparse at high supports resulting in only a few frequent itemsets with supports "close" to $minsup$. Hence, frequent itemsets are likely to be locally frequent within most partitions. Even if they are not locally frequent in a few partitions, it is very likely that they are still d-frequent over these partitions. Hence, their counters are updated even over these partitions. Therefore, the complete counts of most candidates would be available at the end of the first pass resulting in a "light and short" second pass. Hence, it is expected that the performance of ARMOR will be close to that of Oracle for high supports.

Since the frequent itemset distribution becomes dense at low supports, the above argument does not hold in this support region. Hence we see that ARMOR's performance relative to Oracle decreases at low supports. But, what is far more important is that ARMOR consistently performs within a *factor of two* of Oracle. This is highlighted in Table 3 where we show the ratios of the performance of ARMOR to that of Oracle for the lowest support values considered for each of the databases.

Database	$minsup$(%)	ARMOR (seconds)	Oracle (seconds)	ARMOR/Oracle
T10.I4.D10M	0.1	371.44	226.99	1.63
T20.I12.D10M	0.4	1153.42	814.01	1.41
T40.I8.D10M	1.15	2703.64	2267.26	1.19

Table 3: **Worst-case Efficiency of ARMOR w.r.t Oracle**

8.2 Experiment 4: Memory Utilization in ARMOR

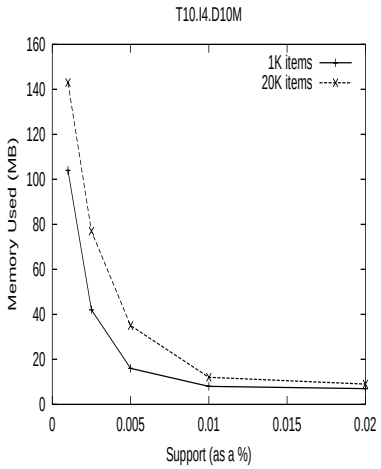

Figure 13: **Memory Utilization in ARMOR**

The previous experiments were conducted with the total number of items, N, being set to 1K. In this experiment we set the value of N to 20K items for the T10.I4 database – this environment represents an extremely stressful situation for ARMOR with regard to memory utilization due to the very large number of items. Figure 13 shows the memory utilization of ARMOR as a function of support for the N = 1K and N = 20K cases. We see that the main memory utilization of ARMOR scales well with the number of items. For example, at the 0.1% support threshold, the memory consumption of ARMOR for N = 1K items was 104MB while for N = 20K items, it was 143MB – an increase in less than 38% for a 20 times increase in the number of items! The reason for this is that the main memory utilization of ARMOR does not depend directly on the number of items, but only on the size of the output, $F \cup N$, as discussed in Section 7.

8.3 Experiment 5: Real Datasets

Despite repeated efforts, we were unable to obtain large real datasets that conform to the sparse nature of market basket data since such data is not publicly available due to proprietary reasons. The datasets in the UC Irvine public domain repository [8] which are commonly used in data mining studies were not suitable for our purpose since they are dense and have long patterns. We could however obtain two datasets – BMS-WebView-1, a clickstream data from Blue Martini Software [30] and EachMovie, a movie database from Compaq Equipment Corporation [1], which we transformed to the format of boolean market basket data. The resulting databases had 59,602 and 61,202 transactions respectively with 870 and 1648 distinct items.

We set the rule support threshold values for the BMS-WebView-1 and EachMovie databases to the ranges

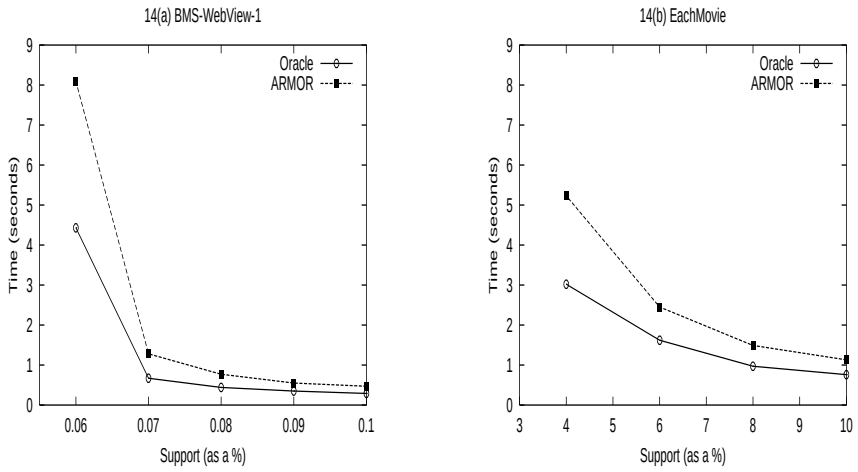

Figure 14: **Performance of Armor (Real Datasets)**

(0.06%–0.1%) and (3%–10%), respectively. The results of these experiments are shown in Figures 14a–b. We see in these graphs that the performance of ARMOR continues to be within twice that of Oracle. The ratio of ARMOR's performance to that of Oracle at the lowest support value of 0.06% for the BMS-WebView-1 database was 1.83 whereas at the lowest support value of 3% for the EachMovie database it was 1.73.

8.4 Discussion of Experimental Results

We now explain the reasons as to why ARMOR should typically perform within a factor of two of Oracle. First, we notice that the only difference between the single pass of Oracle and the first pass of ARMOR is that ARMOR continuously generates and removes candidates. Since the generation and removal of candidates in ARMOR is dynamic and efficient, this does not result in a significant additional cost for ARMOR.

Since candidates in ARMOR that are neither d-frequent nor part of the current negative border are continuously removed, any itemset that is locally frequent within a partition, but not globally frequent in the entire database is likely to be removed from G during the course of the first pass (unless it belongs to the current negative border). Hence the resulting candidate set in ARMOR is a good approximation of the required mining output. In fact, in our experiments, we found that in the worst case, the number of candidates counted in ARMOR was only about *ten percent* more than the required mining output.

The above two reasons indicate that the cost of the first pass of ARMOR is only slightly more than that of (the single pass in) Oracle.

Next, we notice that the only difference between the second pass of ARMOR and (the single pass in) Oracle is that in ARMOR, candidates are continuously removed. Hence the number of itemsets being counted in ARMOR during the second pass quickly reduces to much less than that of Oracle. Moreover, ARMOR

does not necessarily perform a complete scan over the database during the second pass since the second pass ends when there are no more candidates. Due to these reasons, we would expect that the cost of the second pass of ARMOR is usually less than that of (the single pass in) Oracle.

Since the cost of the first pass of ARMOR is usually only slightly more than that of (the single pass in) Oracle and that of the second pass is usually less than that of (the single pass in) Oracle, it follows that ARMOR will typically perform within a factor of two of Oracle.

9 Related Work

In this section, we briefly review a representative set of the major association rule mining algorithms proposed in the literature. The very first algorithm was AIS [4] which was followed by the Apriori algorithm [6], a multi-pass algorithm that incorporates the optimization of frequency based pruning of candidates. The 2-pass CARMA algorithm proposed in [13] was a novel approach in that it performs candidate generation and removal on a tuple-by-tuple basis.

All the above algorithms do candidate counting on a tuple-by-tuple basis and hence suffer from the drawbacks mentioned in Section 3. An alternative approach was suggested in [20] based on a partitioning scheme in which the database is logically divided into disjoint partitions, allowing for further pruning of candidates. A variation of Partition was proposed in [15] that makes use of the cumulative count of each candidate to achieve an illusion of a "large partition". The FP-growth algorithm proposed in [12] is based on a different approach. It constructs a condensed representation of the database called an FP-tree and then performs mining over the FP-tree.

While the above algorithms were primarily horizontal (tuple) based approaches, the MaxClique [28] and VIPER [21] algorithms were designed to efficiently mine databases that are available in a vertical layout.

All the above-mentioned studies (except VIPER, as discussed below) have focussed on evaluating the performance of mining algorithms with respect to their predecessors. In particular, most of them compare against the classical Apriori online mining algorithm.

With regard to evaluating the performance of mining algorithms with respect to idealized, offline algorithms, a preliminary step was taken in our previous work [21], where we compared the vertical VIPER against the oracle version of *Apriori*. This oracle, which we will refer to as Apriori-Oracle differs very much from the Oracle algorithm discussed in this paper in the following significant aspects: (1) The Apriori-Oracle primarily used the hashtree data structure [6] whereas Oracle primarily uses the DAG structure (as defined in Section 3). (2) The Apriori-Oracle does counting with a tuple-by-tuple approach, while Oracle follows a

partitioning approach. (3) The Apriori-Oracle was implemented only to count itemsets in F whereas Oracle counts itemsets in both F and N. (4) Finally, no proofs of optimality were associated with the Apriori-Oracle.

10 Conclusions

A variety of novel algorithms have been proposed in the recent past for the efficient mining of association rules, each in turn claiming to outperform its predecessors on a set of standard databases. In this paper, our approach was to quantify the algorithmic performance of association rule mining algorithms with regard to an idealized, but practically infeasible, "Oracle". The Oracle algorithm utilizes a partitioning strategy to determine the supports of itemsets in the required output. It uses direct lookup arrays for counting singletons and pairs and a DAG data-structure for counting longer itemsets. We have shown that these choices are optimal in that only required itemsets are enumerated and that the cost of enumerating each itemset is $\Theta(1)$. Our experimental results showed that there was a substantial gap between the performance of current mining algorithms and that of the Oracle.

We also presented a new online mining algorithm called ARMOR (Association Rule Mining based on ORacle), that was constructed with minimal changes to Oracle to result in an online algorithm. ARMOR utilizes a new method of candidate generation that is dynamic and incremental and is guaranteed to complete in two passes over the database. Our experimental results demonstrate that ARMOR performs within a *factor of two* of Oracle.

In our future work, we propose to extend the Oracle approach to the development of *parallel* and *top-down* algorithms.

References

[1] Eachmovie collaborative filtering data set. http://www.research.compaq.com/SRC/eachmovie/, 1997.

[2] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. Depth first generation of long patterns. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, June 1998.

[3] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent itemsets. *Parallel and Distributed Computing*, March 2001.

[4] R. Agrawal, T. Imielinski, and A. Swamy. Mining association rules between sets of items in large databases. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 1993.

[5] R. Agrawal and J. C. Shafer. Parallel mining of association rules. *IEEE Transactions on Knowledge and Data Eng.*, December 1996.

[6] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of Intl. Conf. on Very Large Databases (VLDB)*, September 1994.

[7] R. J. Bayardo. Efficiently mining long patterns from databases. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, June 1998.

[8] C.L. Blake and C.J. Merz. UCI repository of machine learning databases. http://www.ics.uci.edu/∼mlearn/MLRepository. html, 1998.

[9] S. Cook and R. Reckhow. Time bounded random access machines. *Computer and System Sciences*, 7, 1973.

[10] R. Feldman et al. Efficient algorithms for discovering frequent sets in incremental databases. In *Proc. of SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, May 1997.

[11] E-H. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 1997.

[12] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 2000.

[13] C. Hidber. Online association rule mining. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, June 1999.

[14] D. Lin and Z. M. Kedem. Pincer-search: A new algorithm for discovering the maximum frequent set. In *Intl. Conf. on Extending Database Technology*, March 1998.

[15] J. Lin and M. H. Dunham. Mining association rules: Anti-skew algorithms. In *Intl. Conf. on Data Engineering*, 1998.

[16] H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. Technical report, University of Helsinki, 1997.

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

[18] S. Parthasarathy, M. J. Zaki, M. Ogihara, and W. Li. Parallel data mining for association rules on shared-memory systems. *Knowledge and Information Systems*, February 2001.

[19] V. Pudi and J. Haritsa. Quantifying the utility of the past in mining large databases. *Information Systems*, July 2000.

[20] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. of Intl. Conf. on Very Large Databases (VLDB)*, 1995.

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

[22] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proc. of Intl. Conf. on Very Large Databases (VLDB)*, September 1995.

[23] R. Srikant and R. Agrawal. Mining quantitative association rules in large relational tables. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, June 1996.

[24] S. Thomas et al. An efficient algorithm for the incremental updation of association rules in large databases. In *Intl. Conf. on Knowledge Discovery and Data Mining*, August 1997.

[25] H. Toivonen. Sampling large databases for association rules. In *Proc. of Intl. Conf. on Very Large Databases (VLDB)*, 1996.

[26] Y. Xiao and M. H. Dunham. Considering main memory in mining association rules. In *Intl. Conf. on Data Warehousing and Knowledge Discovery*, 1999.

[27] M. J. Zaki and K. Gouda. Fast vertical mining using diffsets. Technical report, Rensselaer Polytechnic Institute, 2001.

[28] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *Intl. Conf. on Knowledge Discovery and Data Mining*, August 1997.

[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*, December 1997.

[30] Z. Zheng, R. Kohavi, and L. Mason. Real world performance of association rule algorithms. In *Intl. Conf. on Knowledge Discovery and Data Mining*, August 2001.