ARMOR: Association Rule Mining based on ORacle

Vikram Pudi
Intl. Inst. of Information Technology
Hyderabad, India
vikram@iiit.net

Jayant R. Haritsa
Database Systems Lab, SERC
Indian Institute of Science
Bangalore, India
haritsa@dsl.serc.iisc.ernet.in

Abstract

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 current mining algorithms do not perform uniformly well with respect to the Oracle for all database characteristics and support thresholds. In many cases 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 consistently performs within a factor of two of the Oracle on both real and synthetic datasets over practical ranges of support specifications.

1 Introduction

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.

First, 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 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.

Second, 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. Although ARMOR is derived from the Oracle, it may be seen to share the positive features of a variety of previous algorithms such as PARTITION [9], CARMA [5], AS-CPA [6], VIPER [10] and DELTA [7]. Our empirical study shows that ARMOR consistently performs within a factor of two of the Oracle, over both real (BMS-WebView-1 [14] from Blue Martini Software) and synthetic databases (from the IBM Almaden generator [2]) over practical ranges of support specifications.

The environment we consider is one where the pattern lengths in the database are small enough that the size of mining results is comparable to the available main memory. This holds when the mined data conforms to the sparse nature of market basket data for which association rule mining was originally intended. It is perhaps inappropriate to apply the problem of mining *all* frequent itemsets on dense datasets.

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

Organization The remainder of this paper is organized as follows: The design of the Oracle algorithm is described in Section 2 and is used to evaluate the performance of current algorithms in Section 3. Our new ARMOR algorithm is presented in Section 4 and its main memory requirements are discussed in Section 5. The performance of ARMOR is evaluated in Section 6. Finally, in Section 7, we summarize the conclusions of our study.

$\mathcal{D}$	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
$P_1, ..., P_n$	Set of n disjoint partitions of $\mathcal{D}$
d	Transactions in partitions scanned so far during algorithm execution *excluding* the current partition
d^+	Transactions in partitions scanned so far during algorithm execution *including* the current partition
$\mathcal{G}$	DAG structure to store candidates

Table 1. Notation

2 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, $\mathcal{D}$ in item-list format (which is organized as a set of rows with each row storing an ordered list of item-identifiers (IID), representing the items purchased in the transaction), 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. We first describe the mechanics of the Oracle algorithm below and then move on to discuss the rationale behind its design choices in Section 2.2.

2.1 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.

2.1.1 Counting Singletons and Pairs

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 [2, 11] for its first two passes.) The k^{th} entry in the array $\mathcal{A}_1$ contains two fields: (1) $count$, the counter for the itemset X corresponding to the k^{th} item, and (2) $index$, the number of frequent itemsets prior to X in $\mathcal{A}_1$, if X is frequent; **null**, otherwise.

ArrayCount $(T, \mathcal{A}_1, \mathcal{A}_2)$
Input: Transaction T, 1-itemsets Array $\mathcal{A}_1$, 2-itemsets Array $\mathcal{A}_2$
Output: Arrays $\mathcal{A}_1$ and $\mathcal{A}_2$ 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.      A_1[i.id].count++;
4.      if A_1[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.          index_1 = A_1[T^f[j].id].index // row index
9.          index_2 = A_1[T^f[k].id].index // column index
10.         A_2[index_1, index_2]++;
```

Figure 1. Counting Singletons and Pairs in Oracle

Algorithm Description The ArrayCount function shown in Figure 1 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).

2.1.2 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 2 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*.

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 $\mathcal{G}$.

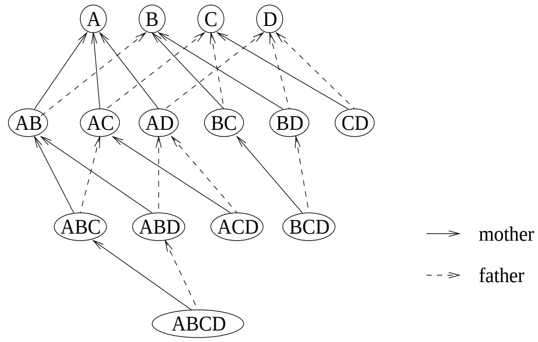

Figure 2. DAG Structure Containing Power Set of $\{$A,B,C,D$\}$

Algorithm Description We use a *partitioning* [9] scheme 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, we hasten to add that the technique is easily extendible to arbitrary partition sizes.

The pseudo-code of Oracle is shown in Figure 3 and operates as follows: The ReadNextPartition function (line 3) reads tuples from the next partition and *simultaneously* creates tid-lists[1] (within that partition) of singleton itemsets in $\mathcal{G}$. Note that this conversion of the database to the tid-list (TL) format is an *on-the-fly* operation and does not change the complexity of Oracle by more than a (small) constant factor. Next, the Update function (line 5) is 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 Figure 2). 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.

[1] A tid-list of an itemset X is an ordered list of TIDs of transactions that contain X.

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

Figure 3. 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 4. Updating Counts

Intersect (B, T)
Input: Tid-vector B, Tid-list T
Output: $B \cap T$
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. $result$ = $result \cup tid$
6. **return** $result$

Figure 5. Intersection

The above processing is captured in the function Update whose pseudo-code is shown in Figure 4. Here, the tidlist of a given node M is first converted to the tid-vector (TV) format[2] (line 1). 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 5. 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.

[2] A tid-vector of an itemset X is a bit-vector of 1's and 0's to represent the presence or absence respectively, of X in the set of customer transactions.

2.2 Rationale for the Oracle Design

We show that Oracle 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. These results are based on the following two theorems. Due to lack of space we have deferred the proofs of theorems to [8].

Theorem 2.1 *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.*

Theorem 2.2 *The cost of enumerating each itemset in Oracle is $\Theta(1)$ with a tight constant factor.*

While Oracle is optimal in these respects, we note that there may remain some scope for improvement in the details of *tidlist computation*. That is, the Intersect function (Figure 5) 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 [13]. 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 4 of Section 2 is to be changed to compute diffsets instead of tidlists by following the techniques suggested in [13].

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 [2] 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.

3 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 [10] and FP-growth [4], 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 [2] 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.I8 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[3] 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.

3.1 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.

3.1.1 Experiment 1: Performance of Current Algorithms

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 6a–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 6a, 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 [4], the authors briefly discuss the issue of constructing disk-resident FP-trees. We however, did not take this into account in our implementation.

3.1.2 Experiment 2: 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 7a–c, which correspond to the T10.I4, T20.I12 and T40.I8 databases, respectively.

In these graphs, we 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.

[3] The original implementation by Han et al. was not available.

4 The ARMOR Algorithm

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

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

  //—— Second Pass ——
    RemoveSmall(G, minsup);
    OutputFinished(G, minsup);
    for i = 1 to n
        if (all candidates in G have been output)
            exit
        ReadNextPartition(P_i, G);
        for each singleton X in G
            Update2(X, minsup);
```

Figure 8. 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 [8] due to lack of space.

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.

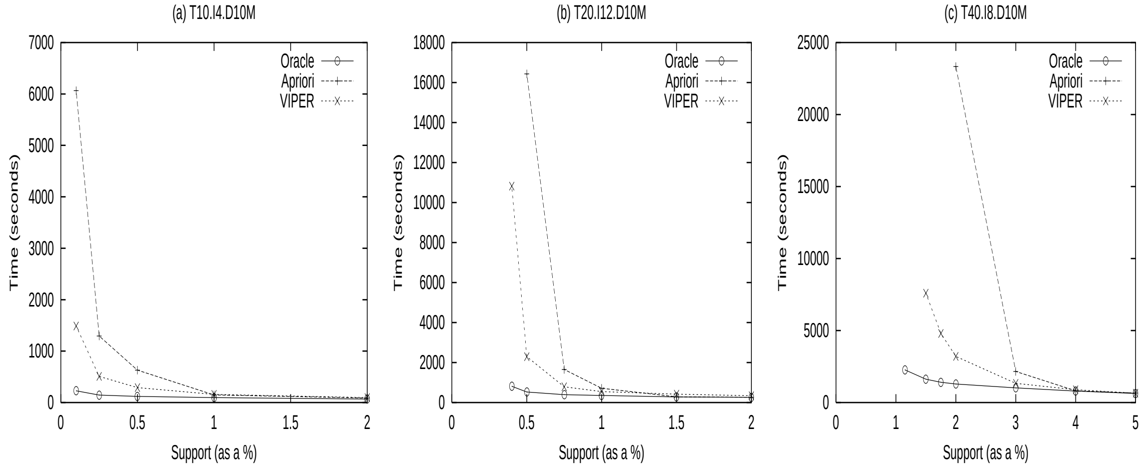

Figure 6. **Performance of Current Algorithms (Large Databases)**

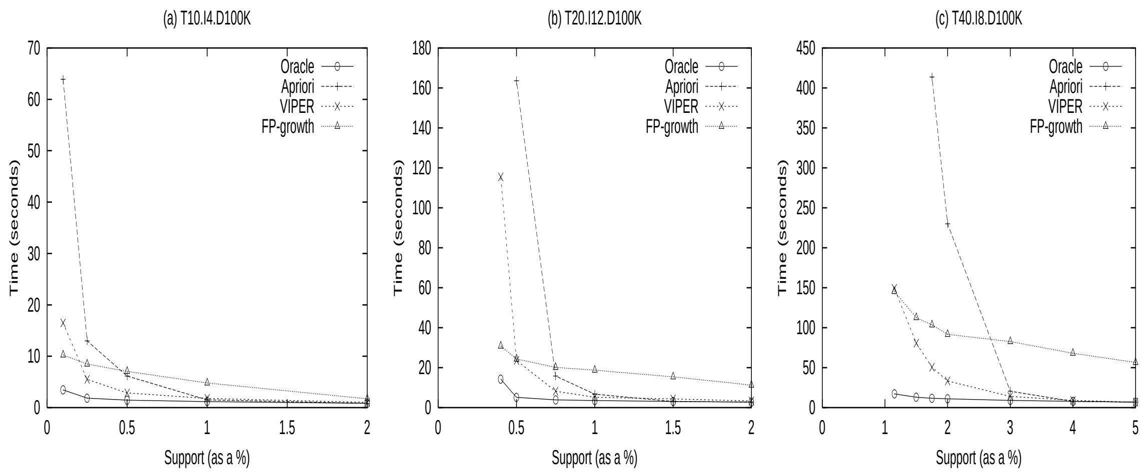

Figure 7. **Performance of Current Algorithms (Small Databases)**

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 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 [8] due to lack of space. 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 [5]: (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 $\mathcal{G}$. $X.firstPartition$ and $X.maxMissed$ are computed when X is inserted into $\mathcal{G}$ in a manner identical to CARMA.

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 $[X.count/|d|, (X.maxMissed + X.count)/|d|]$ [5]. These bounds are denoted by

$\mathsf{minSupport}(X)$ and $\mathsf{maxSupport}(X)$, respectively. We define an itemset X to be d-frequent if $\mathsf{minSupport}(X) \geq minsup$. 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 [8].

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 [8].

The pseudo-code of ARMOR is shown in Figure 8 and is explained below.

4.1 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 [8]) 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.

4.2 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 [8]) 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.

5 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 (and tidlists) of longer 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 [8], 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 well within the available main memory. [12] 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 4.

A final observation 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.

6 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 3. Since Apriori, FP-growth and VIPER have already been compared against Oracle in Section 3.1, we do not repeat those observations here, but focus on the performance of ARMOR. This lends to the visual clarity of the graphs. We hasten to add that ARMOR does outperform the other algorithms.

6.1 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 Figures 9a–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.

6.2 Experiment 4: 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 10 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 5.

Database (|$\mathcal{D}$|=10M)	$minsup$ (%)	ARMOR (seconds)	Oracle (seconds)	ARMOR / Oracle
T10.I4	0.1	371.44	226.99	1.63
T20.I12	0.4	1153.42	814.01	1.41
T40.I8	1.15	2703.64	2267.26	1.19

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

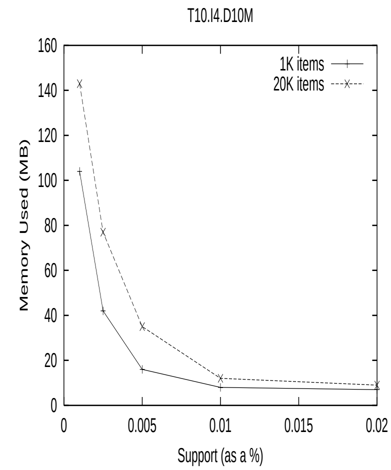

Figure 10. Memory Utilization in ARMOR

6.3 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 [3] which are commonly used in data mining studies were not suitable for our purpose since they

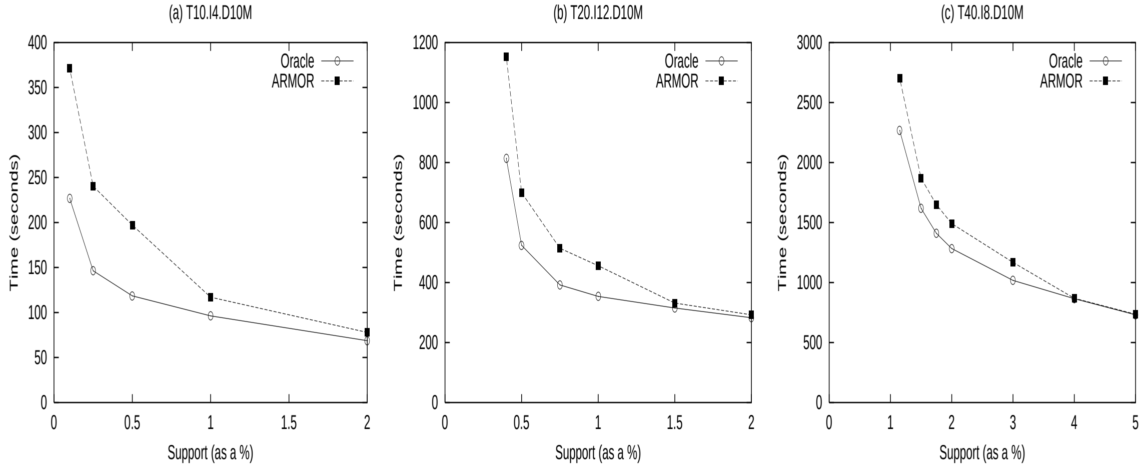

Figure 9. **Performance of ARMOR (Synthetic Datasets)**

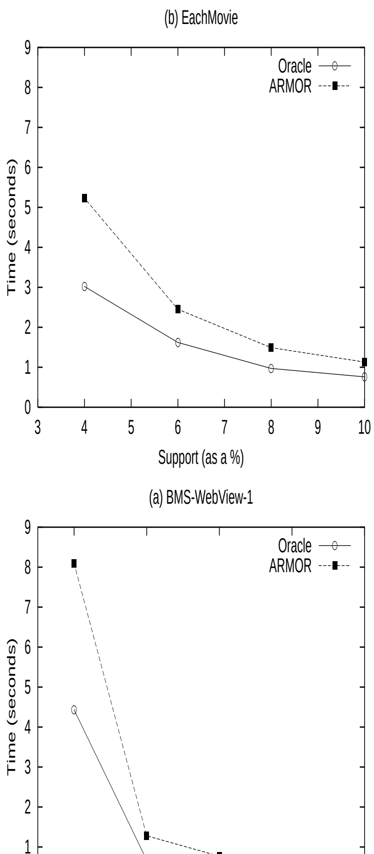

Figure 11. **Performance of Armor (Real Datasets)**

are dense and have long patterns. We could however obtain two datasets – BMS-WebView-1, a clickstream data from Blue Martini Software [14] 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 (0.06%–0.1%) and (3%–10%), respectively. The results of these experiments are shown in Figures 11a–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.

6.4 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.

In summary, due to the above reasons, it appears unlikely for it to be possible to design algorithms that substantially reduce either the number of database passes or the number of candidates counted. These represent the primary bottlenecks in association rule mining. Further, since ARMOR utilizes the same itemset counting technique of Oracle, further overall improvement without domain knowledge seems extremely difficult. Finally, even though we have not proved optimality of Oracle with respect to tidlist intersection, we note that any smart intersection techniques that may be implemented in Oracle can also be used in ARMOR.

7 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.

Acknowledgments This work was partially supported by a Swarnajayanti Fellowship from the Dept. of Science and Technology, Govt. of India.

References

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

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

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

[4] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 2000.

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

[6] J. Lin and M. H. Dunham. Mining association rules: Anti-skew algorithms. In *Proc. of Intl. Conf. on Data Engineering (ICDE)*, 1998.

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

[8] V. Pudi and J. Haritsa. On the optimality of association-rule mining algorithms. Technical Report TR-2001-01, DSL, Indian Institute of Science, 2001.

[9] 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.

[10] 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.

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

[12] Y. Xiao and M. H. Dunham. Considering main memory in mining association rules. In *Proc. of Intl. Conf. on Data Warehousing and Knowledge Discovery (DAWAK)*, 1999.

[13] M. J. Zaki and K. Gouda. Fast vertical mining using diffsets. Technical Report 01-1, Rensselaer Polytechnic Institute, 2001.

[14] Z. Zheng, R. Kohavi, and L. Mason. Real world performance of association rule algorithms. In *Proc. of Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*, Aug. 2001.