Mining Decentralized Data Repositories*

Viviane M. Crestana Nandit Soparkar

Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109-2122
{viviane,soparkar}@eecs.umich.edu

Abstract

Current technology for mining data typically applies to data stored centrally (i.e., in one single repository, homogeneous, with central administration, and a single schema). For instance, association rules algorithms assume that the data of interest is available at one location in a single table, or that prior to mining, the data will be warehoused centrally. However, it is important to study the mining of decentralized data (i.e., consisting of several tables, perhaps obtained via normalization or partitioning and allocation, stored in several repositories with possibly separate administration and schema). Real-life database design, management, and performance aspects suggest these considerations. While there have been a few extensions to mining algorithms for such data, the algorithms developed have largely been aimed at parallel processing (as opposed to the specific issues relating to decentralized data).

In this paper, we examine the issues associated with mining decentralized data. We motivate the need for developing decentralized techniques, and identify the problems that must be addressed. We describe the mining of association rules for the case of a star schema wherein several smaller dimension tables are associated by foreign keys to a central fact table, and present an algorithm that adapts standard algorithms to work efficiently with such data. In contrast to approaches where the data would have been joined first to form a single table, we exploit the foreign key relationships to develop decentralized algorithms that execute concurrently on the separate tables, and thereafter merge the results. We provide analyses to assess our techniques, and we present empirical studies for particular cases to validate the performance.

*This work was initiated when the authors were at the IBM T.J. Watson Research Center.

1 Introduction

Today, a vast amount of stored business and scientific data is available to organizations which may analyze it for profitable use. For instance, retail chains may assess the market potential of products for different age groups, or insurance agencies may gauge health risk factors for different demographic sets. Consequently, a new area of research has emerged: data mining (often called knowledge discovery in databases, or KDD) – the efficient discovery of previously unknown patterns in large databases [11]. As noted in [8], running data mining algorithms turns out to be a small part in the entire process of extracting useful information from raw data. The data must be accessed, selected, cleaned, prepared and transformed into a set that can be mined, and thereafter, the gleaned results must be correctly interpreted by experts and the knowledge gained interactively applied for further mining. Besides the technical matters, human factors often tend to play a crucial role. In particular, the separately generated and maintained data sets even within a given organization may prove to be a significant challenge to current *data mining* (DM) techniques. Centralized data (i.e., data that is stored in one central repository, is homogeneous with a central administration, and in a single schema such as a table) is not typical for most large enterprises. The information may be distributed among different tables, and in some cases, the tables may reside in different physical locations. In fact, in large enterprises, data is often distributed and administered separately even within the same organization.

DM itself is generally performed on data stored in data warehouses, as opposed to transactional/operational data. Even so, the data may not be stored as a flat table as is assumed by data mining algorithms (e.g., see [4, 20]). A popular schema design used in data warehouses is the star schema [17], in which information is classified into two groups: facts and dimensions. Facts are the core data being analyzed, and dimensions are the attributes about the facts. The fact table is often much larger than the dimension tables, and would thereby be a consideration with regards to performance in DM. If the typical *association rules* (AR) algorithms (e.g., see [3]) were applied to data stored in a star schema, the first step would be to compute the join of the fact table with its dimension tables. Since the cost of actually computing the join gets overshadowed by the cost of running DM, it may appear acceptable to compute the join. However, a join results in a table with many more columns and rows (which normalization had been used to avoid), and this would significantly increase the cost of DM as well. Given the large datasets, and combinatorial explosion issues in DM processing, performance issues take on a new urgency for DM. In fact, as part of our preliminary study for performance, we examined some DM algorithms for centralized data (see [1]). Referring to our experience and analysis as presented in this paper, we argue that there is a significant need and benefit in developing DM (specifically here for AR) for decentralized tables, and we believe this is an important issue.

The commonly used relational databases are designed to contain several inter-related tables, often generated by the process of normalization. The goal of normalization is to generate an appropriate set of table schemas to reduce redundancy, to check efficiently for certain constraints on the data values (e.g., functional dependencies), and yet to allow for easy and efficient data access (e.g., see [18]). In general, a relation schema (i.e., a table) that has many attributes is decomposed into several schemas (tables) with fewer attributes.

The design of a distributed system involves making decisions on the placement of data and programs across the sites of a computer network ([14]). Enterprises have data stored in a distributed format, with design considerations such as data fragmentation, distributions strategies, and data allocation methods. These factors affect the operational usage of such data (e.g., maximize performance for updates, load balance, etc.), as opposed to facilitate data mining. Therefore, in most cases, data mining is run on data that was gathered for different operational purposes.

The distributed nature of the data leads to several different processing architectures [8]. There may be a "disconnected" approach wherein data is sent on physical media to large central mainframes where all the warehousing is done. Alternatively, a "transaction server" approach allows for data to be downloaded across a network, and processed by servers. Finally, for administrative reasons, and to some extent performance considerations, a "data server" approach seems very appropriate: the processing is effected at the sites where the data resides.

Another important aspect is determining how database design considerations affects data mining. Considering that the data is already distributed (due to database design), it would be ideal to take advantage

of this distribution and distribute the data mining task, which would provide load balancing as far as I/O, CPU and even communication costs, considering that we would only need to ship part of the data – the one that after some pre-processing proves itself to be relevant to the mining.

In this paper, we restrict our attention to one specific type of decentralized data, namely, star schema tables which represent a case of tables associated by foreign key relationships. Motivated by pragmatic considerations stated above, we present an algorithm that adapts and extends the state-of-the-art AR algorithms to work efficiently with such decentralized (though homogeneous) tables. We exploit the decentralization by executing AR algorithms concurrently on the separate tables, and thereafter we merge the results. This approach requires some re-design of available algorithms, and new factors that affect performance must be considered. We present our analytical and empirical evaluation for particular cases to illustrate the performance benefits of our approach.

This paper is organized as follows. In Section 2, we provide the most relevant related work. In Section 3, we provide an overview for DM of decentralized tables with an example. In Section 4, we present our decentralized approach to discovering AR, and in Section 5, we present our cost analysis comparing our and the standard approach. In Section 6, we present our empirical validation of some aspects of our decentralized approach. Finally, Section 7 concludes the paper.

2 Related Work

The KDD process is interactive and iterative, involving numerous steps such as understanding the application domain, selecting the target data set, cleaning the data as well as transforming the data to a suitable format, applying the data mining algorithms and finally interpreting and evaluating results [10]. We are focused, for the most part, on the DM algorithms themselves. In order to do so, especially for large quantities of data, there is a need for studying how the base data is stored (e.g., see [7]). Among other aspects, system-level support is necessary for data mining algorithms, heterogeneity and distributed environments (for a survey on these issues, see [8]).

Among the DM algorithms of interest to us is the discovery of association rules. The problem of discovering association rules was introduced by [3]. Given a set of items I, and a set of transactions T, where each transaction t_j is composed of a subset i_j of the set of items I, the problem is to find associations among the items such that the presence of some items in a transaction will imply the presence of other items in the same transaction. In [4], the *Apriori* algorithm improved upon the performance of the original algorithm. Since then, a number of algorithms based on [4] have been presented (e.g., [15, 16, 5, 1]). Some algorithms have been considered for new kinds of association rules (e.g., [19, 12, 20]). In particular, the work in [20] considered the problem of applying the Apriori algorithm to a more general class of attributes which could be either quantitative (e.g. age, income) or categorical (e.g. zip code, make of car).

There has been some work on distributing the Apriori algorithm (e.g., [6]) in which the database is horizontally partitioned. In that sense, the amount of information read and being processed is the same as in a sequential algorithm (other than the message exchanges) except that the load is distributed. In contrast, we provided an approach that applies to vertically partitioned tables, perhaps also physically distributed, by which the cost of processing is reduced.

Database design is an area which is likely to impact the task of data mining. Important aspects include the way in which the data is stored, partitioned, distributed, how it relates. In particular, there is significant work in normalization theory (e.g., see [13, 18]) as well as partitioning/allocation (e.g., see [21]) that is relevant to decentralized mining of data.

There is significant work done in distributed query processing that is related to the general goal of distributed data mining. For instance, the semi-join algorithms ([14]) could greatly benefit our approach when applied to physically distributed tables. General sorting and joining algorithms ([18, 9, 22]), available in the literature are also important in the stage of combining the data, either prior, as well as during different stages of the decentralized algorithms.

Finally, when dealing with distributed data sets, one important issue is the heterogeneity of the data. There has been considerable work in the database field dealing with heterogeneous databases ([2]), and while we recognize this as an important topic, we are not looking into it directly.

3 Mining Decentralized Tables

In this section, we illustrate some of the problems in data mining for decentralized data in the discovery of association rules.

3.1 Association Rules

We consider the problem of discovering association rules introduced in [3]. Given a set of items I, and a set of transactions T, where each transaction t_j is composed of a subset i_j of the set of items I, we want to find associations among the items such that the presence of some items in a transaction will imply the presence of other items in the same transaction. An association rule, defined as an implication of the form $X \Rightarrow Y$, where X and Y are subsets of a set of items I, is said to have a *confidence* of $0 \leq c \leq 1$ iff at least $c\%$ of the transactions in the database which contain the items in X also contain the items in Y. The *support* for such a rule is defined to be the fraction of records in the database which contain all of the items in $X \cup Y$. The problem is to find rules that meet user-specified minimum confidence and support.

In [4], the problem of discovering association rules is decomposed into the following two steps:

1. Discovering all the large (i.e., frequent) itemsets (i.e., which meet a user-defined support threshold).
2. Generating all the association rules, based on the support counts found in step 1.

1) $L_1 = \{\text{large 1-itemsets}\}$;
2) **for** $(k = 2; L_{k-1} \neq \emptyset; k++)$ **do begin**
3) generate C_k from (L_{k-1});
4) **forall** transactions $t \in D$
5) **forall** $c \in t$ such that $c \in C_k$
6) increment count of c (i.e., c.count++)
7) $L_k = \{c \in C_k | c.\text{count} \geq \text{minsup}\}$
8) **end**
9) Answer $= \bigcup_k L_k$;

Figure 1: Apriori algorithm sketch.

In [4], the Apriori algorithm was proposed for the first step of the association rules discovery (i.e., discovering all frequent itemsets). Figure 1 provides a sketch of the algorithm. The Apriori algorithm performs the counting of itemsets in an iterative manner, by counting the itemsets that contain k items (k-itemsets) at iteration k. In each iteration, a candidate set of frequent itemsets is constructed and the database is scanned to count the number of occurrences of each candidate itemset. After counting all the candidate itemsets, the algorithm determines which ones were found to be frequent, i.e., have a count above the predetermined threshold. The frequent itemsets found at the end of the scan are used to compute the set of candidates for the next iteration. For details on the Apriori algorithm, please refer to [4].

The first step is by far the most expensive step, and most of the work in discovering association rules concentrates on improving the performance of this step (e.g., see [15, 16, 5, 1]).

3.2 Example Schema

The following is an example of a schema composed of more than one table that may arise in a bank schema:

- $Customer(\underline{acct\#}, name, balance, street, zipcode, age)$ [1]
- $ATM(\underline{atm\#}, type, street, zipcode, limit)$
- $ATMactivity(\underline{xact\#}, acct\#, atm\#, time, amount)$

On Figure 2, we show a relevant projection of the tables. [2] Assume that the quantitative attributes (of age and monetary amounts) were partitioned [3] using the algorithm presented in [20] and are presented in Figure 3.

Customer Table			
acct#	balance	zipcode	age
01	1000	x	20
02	1500	z	25
03	1000	y	20
04	2000	y	30
05	5000	x	31
06	1000	z	35

ATMactivity Table		
acct#	atm#	amount
01	A	20
01	A	15
02	A	20
02	C	50
02	C	50
03	A	20
03	B	20
04	B	50
04	E	500
05	A	25
05	A	25
05	D	50
06	C	50
06	F	700

ATM Table			
atm#	type	zipcode	limit
A	drive	x	7000
B	out	y	4000
C	out	z	4200
D	in	x	15000
E	in	y	10000
F	in	z	11000

Figure 2: Relevant projection of the three tables.

Customer Table			
acct#	balance	zipcode	age
01	1000..1999	x	20..29
02	1000..1999	z	20..29
03	1000..1999	y	20..29
04	2000..5000	y	30..39
05	2000..5000	x	30..39
06	1000..1999	z	30..39

ATM Table			
atm#	type	zipcode	limit
A	drive	x	0..9999
B	out	y	0..9999
C	out	z	0..9999
D	in	x	10000..19999
E	in	y	10000..19999
F	in	z	10000..19999

Figure 3: Tables Customer and ATM after partitioning of attributes.

Due to accesses to single tables, the traditional approach to discovering association rules will work well for finding associations such as:

- $< age : 20..29 > \Rightarrow < balance : 1000..1999 >$ with confidence 100%, support 50%, for table *Customer*

[1] The underlined attribute is the primary key for the table.

[2] There are repeated entries in $ATMactivity$, which correspond to different $xact\#$'s (i.e., the projection here does not eliminate duplicates).

[3] In this paper we are not concerned with the manner in which quantitative attributes are partitioned.

- $< type = in > \Rightarrow < limit : 10000..19999 >$ for table ATM
- $< type = out > \Rightarrow < limit : 1000..9999 >$ for table ATM which, together with the previous rule, indicates the type of ATM likely to influence the amount of money withdrawn daily.

However, suppose we want to find associations w.r.t. activities on the ATM. For example, we want:

- $< age : 30..39 > AND < type = drive > \Rightarrow < Customer.zipcode = x >$ which reflects that people between the age of 30 and 39 that go to a drive-thru ATM, tend to live in area x. (In our example, area x is the only one that has a drive-thru ATM.) This could mean that customers of this age are unlikely to drive out of their area just to find an ATM that is drive-thru.

For this latter type of rules, (i.e., ones involving more than one table), if we were to use the algorithm described in [4, 20], we would first have to join of the tables ($Customer \bowtie ATMactivity \bowtie ATM$ as shown in Figure 4), and then run the algorithm. There is significant redundancy in this joined table, e.g., the itemset $< age : 30..39 > AND < area = x >$ that happens 3 times in the final table, and therefore would be counted 3 times, corresponds to one entry in the customer table, namely the one for primary key $acct\# = 05$.

Alternatively, we suggest that the association rules algorithm could be modified to run on the individual tables of $Customer$ and ATM separately. Thereafter, using the $ATMactivity$ table, the results could be "merged." Of course, how precisely to merge these results requires careful considerations. In case the tables are physically distributed, our approach would help with load balancing, since our algorithm could be run on the tables $Customer$ and ATM individually and concurrently.

acct#	balance	zipcode	age	amount	atm#	type	zipcode	limit
01	1000..1999	x	20..29	15..25	A	drive	x	0..9999
01	1000..1999	x	20..29	15..25	A	drive	x	0..9999
02	1000..1999	z	20..29	15..25	A	drive	x	0..9999
02	1000..1999	z	20..29	50..100	C	out	z	0..9999
02	1000..1999	z	20..29	50..100	C	out	z	0..9999
03	1000..1999	y	20..29	15..25	A	drive	x	0..9999
03	1000..1999	y	20..29	15..25	B	out	y	0..9999
04	2000..5000	y	30..39	50..100	B	out	y	0..9999
04	2000..5000	y	30..39	500..1000	E	in	y	10000..19999
05	2000..5000	x	30..39	15..25	A	drive	x	0..9999
05	2000..5000	x	30..39	15..25	A	drive	x	0..9999
05	2000..5000	x	30..39	50..100	D	in	x	10000..19999
06	1000..1999	z	30..39	50..100	C	out	z	0..9999
06	1000..1999	z	30..39	500..1000	F	in	z	10000..19999

Figure 4: Resulting table after the three tables were joined.

4 A Decentralized Approach

We now describe our technique for decentralized DM of tables described in Section 3. Let a *primary table* be a relational table with one primary key and no foreign keys; the table may have other fields that are categorical, numerical or boolean. Also, let a *relationship table* be a relational table which contains foreign keys to other *primary tables*. A *relationship table* is a *n-relationship table* when it has n foreign keys to n primary tables. Typically, primary tables refer to entities, and relationship tables correspond to relationships in an ER diagram ([18]).

Considering the example presented in Section 3.2, the $Customer$ and ATM tables are primary tables, and $ATMactivity$ is a relationship table. Our goal is to find association rules in the final joined table T, where $T = Customer \bowtie ATMactivity \bowtie ATM$.

The final results we get by running a decentralized algorithm (i.e., the discovered association rules), are exactly the same as with the original algorithm run on the joined table T. That is, when computing the support of itemsets in an individual primary table, the support should reflect the final support that this itemset would have in table T. Therefore, before we run the algorithm in the primary table, we have to take into account the number of times each row of the primary table is in the final joined table T. After that, we can proceed, and run a modified algorithm in the primary tables.

We present a decentralized version of the Apriori algorithm applied to star schemas in which there are n primary tables – the *dimension tables* – and one central relationship table – the *fact table*.

4.1 A Decentralized Algorithm

Let $T_1, T_2, \cdots, T_n$ be the dimension tables, and T_{1n} be the fact table, described as follows.

- $T_t(id_t, a_{t1}, a_{t2}, \cdots, a_{tm_t})$ with id_t as primary key and $a_{t1} \cdots a_{tm_t}$ as categorical/numerical attributes, $\forall t, t = 1..n$
- $T_{1n}(id_1, id_2, \cdots, id_n)$ with id_t as foreign key to table T_t.

Our algorithm is as follows.

1. *Computing a projection of the relationship table*

   Read relationship table T_{1n} and count the number of occurrences of each value for the foreign keys $id_1, id_2, \cdots, id_n$.

2. *Counting frequent itemsets on primary tables*

   Run the counting step of the association rules algorithm on the primary tables as described in [4] for all the attributes, except for the primary key, with the following modification: on dimension table T_t, when incrementing the support of an itemset present on row i by 1, increment the value of the support by the number of occurrences of $id_t = i$ on table T_{1n} (computed in the previous step). We refer to this step of our algorithm as the "modified Apriori" in the remainder of this document. We find n sets of frequent (or large) itemsets, namely $l_1, l_2, \cdots, l_n$, for each dimension table, $T_1, T_2, \cdots, T_n$ respectively.

3. *Count itemsets across the primary tables using the relationship table*

   Here we describe 2 different algorithms to count itemsets that contain items from more than one primary table. The naming for each algorithm will be explained later.

   (a) *I/O saving:* Generate candidates from the n primary tables $T_1, T_2, \cdots, T_n$ by having an n-dimensional array, where dimension t corresponds to elements of the set l_t plus the empty set (we need the empty set in order to allow for candidates from more than one and less than n tables). Next, we compute the joined table T, but process each row as the table is created, avoiding the storage costs. For each row in T (say $\sigma_e(T)$), for each t, we take the attributes that came from table T_t (i.e., $\pi_{T_t}(\sigma_e(T))$), and identify all itemsets from l_t contained in $\pi_{T_t}(\sigma_e(T))$, say i_t (i_t includes the empty set also). After computing all i_t's, $t = 1..n$, add one to each position of the n-dimensional array for the elements of $i_1 \times i_2 \times \cdots \times i_n$ (these are the itemsets contained in table T, whose items belong to more than one table).

   (b) *Memory saving:* Build the join of the $n + 1$ tables, creating table T. Run the counting step of the association rules algorithm on table T, but generating only the candidates that contain items from more than one table. In the pruning step, use the frequent itemsets $l_1, l_2, \cdots, l_n$, as well as the frequent itemsets computed in this step of the algorithm.

4. *Generate rules using the support of frequent itemsets found*

   Finally, we generate the association rules the same way as in [4, 20].

We argue that our algorithms, running on the decentralized tables will find the same set of association rules that the Apriori algorithm does running on the joined table.

For the first algorithm, we see that I/O costs are saved during the multiple passes on the smaller tables, as compared to the big table T. Also, we save on processing time since a given frequent itemset consisting of items from only one primary table will be counted fewer times than if we counted on the table T. This may be verified by simple counting arguments. A disadvantage is that, as far as itemsets that contain items from more than one table, there is no pruning from pass to pass because all itemsets are counted in one scan and therefore, we consider some candidates that would not have been considered if we could perform pruning at every step (to the remainder of this document, called "false candidates"). Therefore, if the sets $l_1, l_2, \cdots, l_n$ are too large, this step may require considerable memory space. For the second algorithm, a disadvantage is that we have to store the joined table T, and the I/O costs could be higher since we may have to scan table T as many times as done in the Apriori originally, and we still have to scan tables $T_1, T_2, \cdots, T_n$ more than once (although the sizes of $T_1, T_2, \cdots, T_n$ are very small when compared to the size of T, and the number of passes in table T may be smaller on the third step of our algorithm than on the traditional Apriori algorithm due to data distribution). On the other hand, we save on the counting of the itemsets whose items belong to only one table (i.e., avoid counting redundant items), and enable the counting of the itemsets whose items belong to multiple tables to be faster since the number of candidates to check is much smaller.

Our general approach of distributing the mining algorithms potentially provides many performance improvements. In the case the primary tables reside in different locations, we can apply ideas from semi-join algorithms [14] by sending the information gathered in step 1 to the primary table sites, so they can perform the step 2 locally (and therefore also distributing the load among the different locations). In case of the first algorithm, to perform step 3, we would just send to the more appropriate site (depending on distribution, according to the semi-join strategy) the information of which frequent itemsets are found in each row of the primary tables, as opposed to shipping in the entire primary table, which would have been necessary in the traditional approach to discovering association rules.

4.2 Extensions

We now discuss a couple of extensions to our algorithm, with some of which we experimented, and reported on Section 6.

The first extension is with regards to categorical and/or numerical attributes in the fact table (as shown in the example on Section 3.2). One solution is to treat each attribute of the fact table as coming from a different table and run the algorithms as normal. When scanning the fact table for the first time in order to count occurrences (step 1 of our algorithm), we could verify the frequent itemsets of size one for the attributes of the fact table. On Step 3, in the case of the Memory saving algorithm, having extra attributes do not affect the general algorithm, they are just considered like any other attribute (except that they would not participate in much pruning since they were not combined with other attributes during the second step of the algorithm). In case of the I/O saving algorithm, we could add a dimension for each new attribute. If there are many categorical and/or numerical attributes in the fact table, this solution could be infeasible due to the number of dimensions required. Another approach is to vertically partition the fact table into two tables: one containing all the categorical and/or numerical attributes and a newly generated primary key id_{1n}; and one containing id_{1n} together with all other foreign keys originally present in the fact table. We have now the same scheme as before, with one extra table, and our algorithm can run in the same manner.

Another extension is with regards to a compromise between I/O and Memory saving. In the I/O saving algorithm presented, table T is computed and processed once, so there is no need to store the table, at the expense of having to count all of the itemsets across tables in one step, therefore not only needing a lot of memory space to store the n-dimensional array, as well as not being able to do a better pruning of itemsets. If the n-dimensional array does not fit in memory, we have to resort to the Memory saving algorithm, where

we do all the necessary pruning at the expense of having to store table T and scan it k_{1n} times. Instead, we suggest two hybrid approaches:

1. Build table T and perform the Memory saving approach by scanning the database a few times up to a point where we can fit the entire remaining subsets of the sets $l_1, l_2, \cdots, l_n$ in memory, then switch to I/O saving approach. We implemented a limited version of this algorithm, and it proved to work well in our experiments (see Section 6).

2. Before putting all the tables together, process them pair-wise, e.g., $T_1 \bowtie T_{1n} \bowtie T_2$, with the I/O saving algorithm considering the sets l_1 and l_2 as candidates. This way, we can increase the pruning when considering, say, T_1, T_2 and T_3 because we already know which false candidates involving attributes of T_1 and T_2 not to consider. In this case, we do not need to materialize table T, but the number of joins that we have to compute (first pair-wise, then all tables) is a trade-off and the best solution may be to materialize table T. In the situation where we do not have enough space to materialize T, this algorithm may be a good option.

5 Our Cost Analysis

We now compare the costs associated with running the original Apriori algorithm in the joined table with the cost of running the algorithm we proposed in Section 4.1, considering the two options for step 3, i.e., the I/O saving and Memory saving approaches. We assume that the primary tables are ordered in their primary keys.

5.1 Nomenclature

Suppose we have n primary tables $T_1, T_2, \cdots, T_n$ and a relationship table T_{1n} described as in Section 4.1. Furthermore, the tables T_t, $t = 1..n$ have r_t records, and T_{1n} has r_{1n} records. We also assume that $r_t << r_{1n}, \forall t, t = 1..n$. As described previously, in order to run the standard association rules algorithm [4, 20], the first step is to build the join of the $n+1$ tables: $T = T_1 \bowtie T_{1n} \bowtie T_2 \bowtie \ldots \bowtie T_n$. The table T has attributes $(id_1, a_{11}, \cdots, a_{1m_1}, id_2, a_{21}, \cdots, a_{2m_2}, \cdots, id_n, a_{n1}, \cdots, a_{nm_n})$, and has r_{1n} records.

Let,

- k be the length of the longest candidate itemset when the Apriori algorithm is run on T
- c_{ij} be the i^{th} candidate of length j when the Apriori algorithm is run on T
- l_{ij} be the i^{th} frequent itemset of length j when the Apriori algorithm is run on T
- $|c_{ij}|$ be the number of candidates of length j when the Apriori algorithm is run on T
- $|l_{ij}|$ be the number of frequent itemsets of length j when the Apriori algorithm is run on T
- s_{ij} be the support (number of occurrences) of candidate c_{ij} at table T
- k_t be the length of the longest candidate itemset when the modified Apriori algorithm is run on T_t
- k_{1n} be the length of the longest candidate itemset when the Apriori algorithm is run on T, such that the items belong to more than one primary table ($k_{1n} \leq k$)
- $|c_{ij}^t|$ be the number of candidates of length j where all the items belong to table T_t, $t = 1..n$, when the Apriori algorithm is run on T
- $|c_{ij}^{1n}|$ be the number of candidates of length j where the items belong to more than one primary table, when the Apriori algorithm is run on T

- $|l_{ij}^t|$ be the number of frequent itemsets of length j where all the items belong to table T_t, when the Apriori algorithm is run on T
- $|l_{ij}^{1n}|$ be the number of frequent itemsets of length j where the items belong to more than one primary table, when the Apriori algorithm is run on T
- s_{ij}^t be the support (number of occurrences) of candidate c_{ij} at table T_t. If not all the items in c_{ij} belong to table T_t, $s_{ij}^t = 0$

We divide up the processing costs into I/O and CPU, and we examine each in the following.

5.2 I/O Cost

The I/O cost concerns the cost of accessing the data. In the Association Rules algorithm, this cost is directly related to the size of the database and the number of scans performed.

- Apriori on table T:

$$cost\ of\ join + r_{1n} \sum_{t=1}^{n}(m_t) + k * r_{1n} \sum_{t=1}^{n}(m_t)$$

  where the first term is the cost necessary to compute the join of the tables, the second term is the cost of writing the computed join (so that the Apriori algorithm can run), and the last term is the cost of scanning the database when the Apriori algorithm is run (k scans of the database [4]).

- I/O saving:

$$n * r_{1n} + \sum_{t=1}^{n} r_t + \sum_{t=1}^{n} k_t(m_t + 1)r_t + cost\ of\ join$$

  where the first two terms correspond to step 1 of the algorithm presented in Section 4.1, first counting occurrences and second storing the occurrences to be used in the next step. The third term corresponds to step 2 of the algorithm, i.e., running the modified Apriori algorithm on the primary tables, and the last term corresponds to step 3: we need to compute the join in order to count itemsets across the primary tables, without having to save the result of the join though.

- Memory saving:

$$n * r_{1n} + \sum_{t=1}^{n} r_t + \sum_{t=1}^{n} k_t(m_t + 1)r_t + cost\ of\ join + r_{1n} \sum_{t=1}^{n}(m_t) + k_{1n} * r_{1n} \sum_{t=1}^{n}(m_t)$$

  when comparing to I/O saving, we added the costs of saving the join and scanning table T k_{1n} times.

We do not explicitly compute the cost of the join, since our aim is to compare the approaches, and the cost of the join is the same in either. For the I/O saving approach, we see that the dominant summation is the scanning of the primary tables (k_t times for each table T_t), and for the Apriori on table T approach, the dominant term is the scanning of table T k times. Given that $k_t \leq k, t = 1..n$, and $r_t \ll r_{1n}$, we see that as far as I/O cost, our first approach offers a big saving as compared to first computing the join and then running the Apriori algorithm.

For the Memory saving approach, multiple scans of table T are necessary (last summation in cost formula), and therefore savings in I/O costs are not so evident, and, as mentioned in Section 4.1, our approach may even be more expensive. It will depend on how k_{1n} compares to k. In practice, k_{1n} can be much smaller than k, since more related items tend to be stored together.

[4] In some cases, the algorithm performs one extra scan since the longest candidate itemset could be longer than the longest frequent itemset, but we ignore this case in our cost formulas since this extra scan is not so significant when compared to the rest of the cost. The same is true for our decentralized approach.

5.3 CPU Cost

For easy of understanding, we divide up the CPU costs into four:

- subset: given a row and a set of candidate itemsets, determining the candidates present at the given row.
- actual counting: after determining the candidates present in a row, incrementing the associated counters.
- determining frequent itemsets: at the end of each iteration of the Apriori algorithm (i.e., after each scan) all counters are examined to determine the frequent candidates.
- generating candidates: generating the candidates for the next iteration.

5.3.1 Subset

The subset function is executed for each row of the database, at each scan. This function can be implemented simply, for example, generating all possible itemsets from the given row, and comparing with the possible candidates. This approach turns out to be expensive, and there has been considerable effort put into actually improving this function [4, 1]. It is difficult to assess the cost for this function, since it is highly dependent on the data set distribution. However, the cost is dependent on the length of the row (e.g., m for table T), and on the number of candidates ($|c_{ij}|$ for table T). Let $f(p, q)$ be the cost of the subset function for a row of length p and a candidate set of size q.

The costs are:

- Apriori on table T:

$$\sum_{j=1}^{k} r_{1n} * f(m, |c_{ij}|)$$

  for each iteration j, we perform the subset function on each row read (r_{1n} rows), with parameters m, the length of the row, and $|c_{ij}|$, the number of candidates at iteration j.

- I/O saving:

$$\sum_{t=1}^{n} \sum_{j=1}^{k_t} r_t * f(m_t, |c_{ij}^t|) + \sum_{j=1}^{k} r_{1n} * f(m_t, |l_{ij}^t|)$$

  where the first summation corresponds to the modified Apriori cost at the primary tables, and the second corresponds to step 3 of our algorithm, i.e., when we count the itemsets that have items across tables. For each entry in the joined table (although not materialized), we have to perform subset functions to determine which frequent itemsets are present in a specific row.

- Memory saving:

$$\sum_{t=1}^{n} \sum_{j=1}^{k_t} r_t * f(m_t, |c_{ij}^t|) + \sum_{j=1}^{k_{1n}} r_{1n} * f(m, |c_{ij}^{1n}|)$$

  where $m = \sum_{t=1}^{n} m_t$ and $|c_{ij}^{1n}| = |c_{ij}| - \sum_{t=1}^{n} |c_{ij}^t|$. The first summation corresponds to the modified Apriori cost at the primary tables, and the second summation indicates the counting of itemsets that have items across tables, the same way it is done on Apriori (except with a smaller set of candidates).

The comparison of our first approach (I/O saving) with Apriori is difficult without a formula for the cost $f(p,q)$. Comparing the first summation of the equation for our approach with the equation for the Apriori on table T, wee see that the former terms are much smaller, given that $m_t < m$, $|c_{ij}^t| < |c_{ij}|$, $k_t < k$ and $r_t \ll r_{1n}$, for $t = 1..n$. The terms for the second summation are harder to compare. However, it is always the case that $|l_{ij}| \leq |c_{ij}|$, and in many cases, $|l_{ij}| \ll |c_{ij}|$, in particular, $|c_{i2}|$ is in general much larger than the entire set of frequent itemsets, i.e., $|c_{i2}| \gg \sum_{j=1}^{k} |l_{ij}|$. Also, the terms are multiplied by r_{1n} and in fact, they can be multiplied by less than r_{1n} depending on the distribution of the data (e.g., in case two entries in T_{1n} that have the same value for id_1 are close enough so that the subset function does not need to be recomputed), and in the case of the Apriori on table T approach, we cannot take advantage of that. With that in mind, it is not hard to convince ourselves that our approach will be cheaper than the Apriori on table T approach.

As far as our second approach, Memory saving, again, it is difficult to compare the costs, although we notice that the only term that is multiplied by r_{1n} has a considerable smaller set of candidates in our approach than in the original Apriori.

Of course, in order to assess the real difference in cost of the two approaches, we need a more rigorous analysis or empirical evaluation.

5.3.2 Actual Counting

For each row of the database, after we perform the subset function, we execute the actual counting. The cost here depends on the number of times we have to increment the counters, which is nothing less than the support of the various candidate itemsets.

The costs are:

- Apriori on table T:

$$\sum_{j=1}^{k} \sum_{\forall i} s_{ij}$$

- I/O saving:

$$\sum_{t=1}^{n} \sum_{j=1}^{k_t} \sum_{\forall i} s_{ij}^t + \sum_{j=1}^{k} \sum_{\forall i, c_{ij} \notin T_t, t=1..n} s_{ij} + \sum_{j=1}^{\sum_{t=1}^{n} k_t} \sum_{\forall f} s_{fj}^{'}$$

  where the first summation corresponds to counting during the modified Apriori at the primary tables, the second summation corresponds to counting on the third step (for table T), and the last summation corresponds to counting of "false candidates", i.e., the candidates which would not have been considered if we were running the Apriori on table T, due to pruning at each step.

- Memory saving:

$$\sum_{t=1}^{n} \sum_{j=1}^{k_t} \sum_{\forall i} s_{ij}^t + \sum_{j=1}^{k} \sum_{\forall i, c_{ij} \notin T_t, t=1..n} s_{ij}$$

  where the first and second summations are the same as the first and second summations of the I/O saving approach.

We now compare our approaches with the Apriori. I/O saving: for a given c_{ij}, we have that: if $c_{ij} \in T_t$, for some $t = 1..n$, it will be counted s_{ij}^t times ($s_{ij}^t \ll s_{ij}$); o.w., it will be counted s_{ij} times, which is the same as on the latter approach. So, comparing the first two summations of the I/O saving algorithm with the Apriori on table T, we can see that our approach could be much cheaper depending on the percentage of itemsets belonging to individual tables, and in the worse case, the same. The last summation, where we are counting the "false candidates", is the cost that is extra, i.e., does not appear in the approach where Apriori is run on table T. We analyze two cases:

- *a large percentage of the frequent itemsets belong to the primary tables.* In this case, there will be a lot of savings by counting these itemsets in the primary tables, as opposed to the table T. It will then, most likely pay off having to count the extra "false candidates" indicated in the last term of our approach.
- *a large percentage of the frequent itemsets do not belong to the individual primary tables.* In this case, we will not be saving so much by counting in the primary tables, but on the other hand, since the number of frequent itemsets computed on step 2 is not so large as compared to the "true candidates", we will not be counting so many "false candidates".

Also, it is the case that "false candidates" do not have a high support (otherwise they would have been frequent), so it can turn out not to be a big penalty. Finally, we again notice that in order to assess the real costs, we need a more rigorous analysis or empirical evaluation.

Memory saving: Since our second approach does not count false candidates, this approach, in terms of actual counting could be much better than the Apriori on table T approach, depending on the percentage of itemsets belonging to individual tables, and in the worst case, the same.

5.3.3 Determining Frequent Itemsets

The lasts two cost components (determining frequent itemsets and generating candidates) contribute the least to the final cost of the algorithms. For completeness, we include them in this analysis. Frequent itemsets are determined once for each iteration, and the function consists of visiting all the counters and determining which ones were found to be frequent.

- Apriori on table T:

$$\sum_{j=1}^{k} \sum_{\forall c_{ij}} 1$$

- I/O saving:

$$\sum_{j=1}^{k} \sum_{\forall c_{ij}} 1 + \sum_{j=1}^{\sum_{t=1}^{n} k_t} \sum_{\forall c'_{ij}} 1$$

- Memory saving:

$$\sum_{j=1}^{k} \sum_{\forall c_{ij}} 1$$

This is the only cost where the Apriori on table T approach wins our first approach. This is because in our approach we incur the cost of counting the "false candidates" that are not considered in the traditional Apriori algorithm. Our second approach has the same cost as the traditional Apriori algorithm approach. However, as it was mentioned before, this component of the final cost in negligible as compared to the other components.

5.3.4 Generating Candidates

This function is executed once for each iteration, after we determine the itemsets that were found to be frequent. The cost is dependent on the number of frequent itemsets found. Let the cost be $g(S)$, where S is the set of frequent itemsets.

- Apriori on table T:

$$\sum_{j=1}^{k} g(|l_{ij}|)$$

- I/O saving:

$$\sum_{t=1}^{n} \sum_{j=1}^{k_t} g(|l_{ij}^t|)$$

- Memory saving:

$$\sum_{t=1}^{n} \sum_{j=1}^{k_t} g(|l_{ij}^t|) + \sum_{j=1}^{k_{1n}} g(|l_{ij}^{1n}|)$$

For the I/O saving approach, we only generate candidates when running the modified Apriori on the primary tables. On the Memory saving approach, we also generate candidates when discovering association rules across multiple tables. However, given that $|l_{ij}^t| \subset |l_{ij}|$ and $|l_{ij}^{1n}| \subset |l_{ij}|$ we would expect that the cost of generating candidates is cheaper in both our approaches. So, our approaches presents some savings with regards to generation of candidates, although as we mentioned in the previous section, this cost is negligible when compared to the other components.

6 Empirical Validation

We identify situations where one can benefit from our approach, less in the context of time savings, but more to consider situations where applying the Apriori algorithm is infeasible (e.g., when there is no space available to store the join of the tables). Our primary goal is to study the problem of mining decentralized data (and not just improving the efficiency of existing algorithms). Nevertheless, we would like our approach to be no worse than the traditional approach to the discovery of association rules, and as our results indicate, it performed better in all cases studied. We restricted our attention to the case with three decentralized tables: two primary tables and one relationship table.

6.1 Experimental Setup

In order to evaluate our approach, we ran experiments on synthetic data. We used the data generator provided in [4]; such data is frequently used for evaluation of association rules algorithms. Since our study is based on decentralized data, and the data generated in [4] is centralized (i.e., one table), we used the synthetic data generator to generate the primary tables T_1 and T_2. The parameters for the generator were: N, the number of items; $|D|$, the number of transactions; $|T|$, the average length of transactions; $|I|$ the average length of the maximal potentially frequent itemsets.

In order to generate T_{12}, we wrote our own generator which takes as arguments: $|D|$, the number of rows (or transactions) in the final table T (where $T = T_1 \bowtie T_{12} \bowtie T_2$), and R, the average number of repetitions of entries in T_{12} (remember: $r_{12} \leq r$). For each line generated for table T_{12}, we randomly pick one transaction from table T_1, one from table T_2, and the number of repetitions for this entry was selected from a Poisson distribution (with mean $= R$). We then computed the join of the three tables, and stored table T, in order to compare with running the original Apriori algorithm. After computing the join, we determined the average

length of transactions for table T. The cost of the join was not included in the results, since all the algorithms have to compute the join at one point (i.e., without necessarily materializing the table).

We implemented the I/O saving approach, the Memory saving approach, and a limited version of the Hybrid approach (in which we switch to the I/O saving approach after the first pass). Our experiments were performed using a 200 MHz Pentium Pro machine with 256 Mbytes of RAM, running Linux. We ran extensive evaluations, among which we selected four representative experiments. The parameters for the four experiments, together with the resulting average length of transactions for the resulting table, are listed in Figure 5.

Parameters for testing					
	Parameters	Test 1	Test 2	Test 3	Test 4
T_1	N	500	500	500	500
	$\|D\|$	1000	1000	100000	100000
	$\|T\|$	10	20	6	6
	$\|I\|$	4	6	4	4
T_2	N	550	550	550	550
	$\|D\|$	1200	1200	100000	100000
	$\|T\|$	10	20	7	7
	$\|I\|$	4	6	4	4
T_{12}	$\|D\|$	100000	100000	10000000	10000000
	R	50	50	50	300
T	$\|T\|$	20	38	13	13

Figure 5: Parameters for testing.

6.2 Experimental Results

We ran experiments for various support values, where the hash tree for the Apriori algorithm (please refer to [4]) always fit completely in memory. In case the hash tree does not fit in memory, the savings provided by our approach would be even greater, given that at each pass, the number of candidates that our approach has to check is smaller than the original Apriori algorithm.

For clarity and easy of comparison, we present the results in the following way: first, we plot the time taken by the Apriori algorithm when run against table T, and second, the time taken by our approach divided by the time taken by the Apriori approach, what we refer to as normalized results. The results are presented in Figures 6 to 9.

For the first two experiments (Test 1 and Test 2), the files are reasonably small (Table T with only 100000 transactions), and our goal was to verify whether our approach could provide some CPU savings, considering that the entire table fit in main memory. We verified that for both experiments, our approach performed better, with the exception of the I/O saving algorithm. The reason for I/O saving approach performing poorly in this case, is that, as we explained in Section 4.1, when there are too many candidates from the primary tables, the number of false candidates grows exponentially. In such a situation, the I/O saving approach becomes infeasible. For lower values of support, the savings provided are not so significant, but our algorithm still performs no worse than the Apriori.

For Test 3 and Test 4, the files were larger, and we were able to assess the savings provided by scanning the big table T fewer times. For Test 4 specifically, we see that the I/O saving algorithm performs well for all support values, which is an advantage with our approach since the I/O saving algorithm need never materialize the computed join, and therefore, there are no excessive space requirements.

Our algorithms were run serially, i.e., first we ran modified Apriori on table T_1, then we ran modified Apriori on table T_2, and then step 3 of our algorithm on table T to compute frequent itemsets across tables

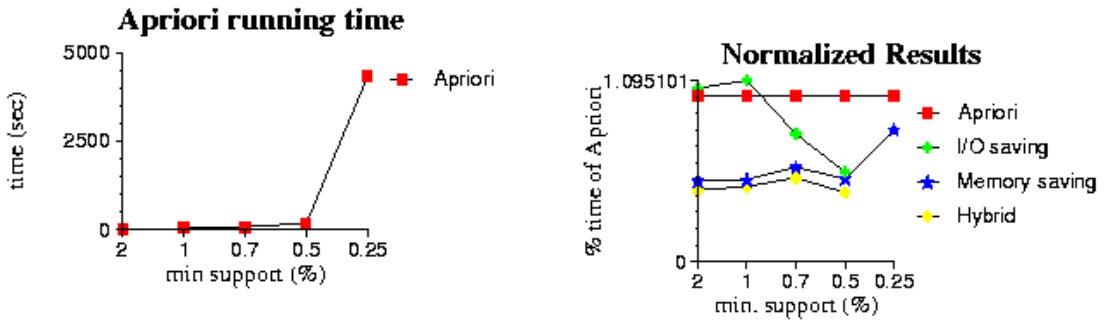

Figure 6: Results for Test 1.

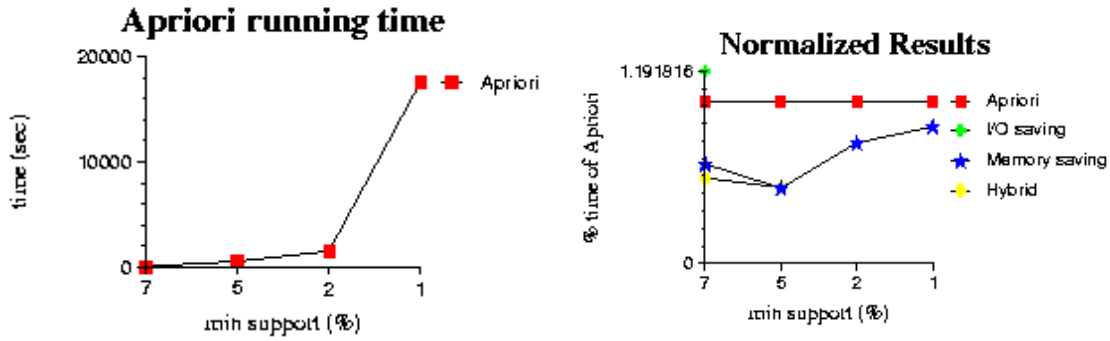

Figure 7: Results for Test 2.

(I/O saving, Memory saving, or Hybrid). Therefore, the times shown, account for the running time on the primary tables as well. If the original tables were physically distributed, this part of the load would be divided, i.e., the modified Apriori would run concurrently on the primary tables, and the actual running times for our approach would be much lower (due to parallelism). The savings are likely to be even more significant when there are more than two primary tables involved, and we are currently extending our implementation to run with an arbitrary number of tables.

6.3 Comparison with Cost Analysis

In this section we briefly discuss our cost model in retrospect. We modified our code to keep track of the number of times each operation was performed. For example, for computing the subset, a counter was incremented each time a node in the hash tree was accessed, and each time an itemset contained in a leaf was checked against a transaction (i.e., a row in the database).

For experiments Test 1 and Test 2, the process was CPU intensive (the percentage of the total time that was CPU time ranged from 99% to 100%). We now compare the numbers we gathered by counting the operations with the running time reported in Section 6.2. For Test 1, with minimum support = 0.5%, 179M operations were performed for the subset computation (i.e., accessing the tree as well as checking itemsets at the leaves) for the Memory saving approach, and 290M for the Apriori approach. For the counting function, there were 5.2M operations for the Memory saving approach while for the Apriori approach the number was 13M. The costs for the candidate generation and determining frequent itemsets were 1M each for the

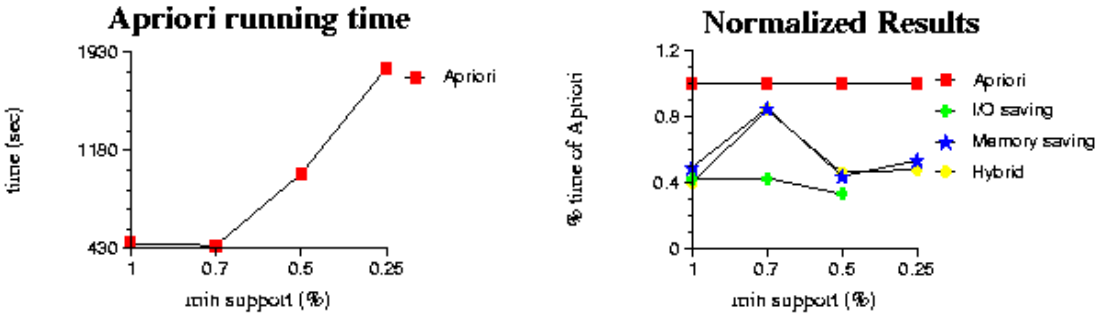

Figure 8: Results for Test 3.

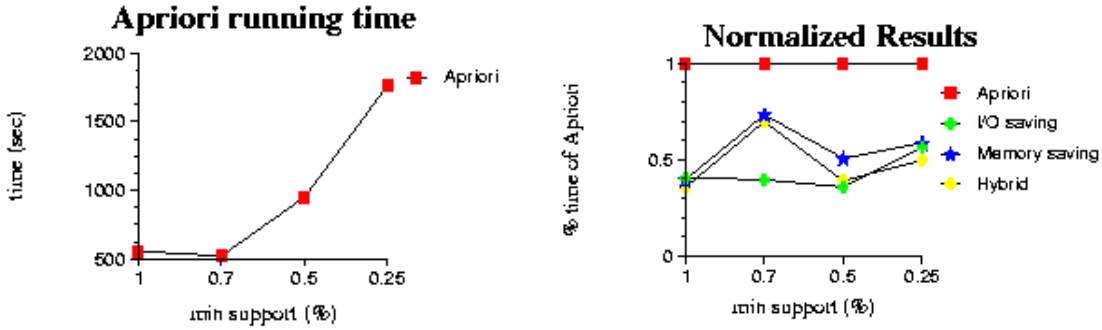

Figure 9: Results for Test 4.

Memory saving approach, and less than 1M each for the Apriori approach. We see that the subset component accounted for 96% of the total cost for the Memory saving approach, and 95% of the total cost of the Apriori approach. We also see that the ratio of the number of operations of Memory saving/Apriori is 62% considering only the subset component, and 61% considering all four components. By our empirical evaluation, the ratio found for the total running time was 58% (see Figure 6), which is close within experimental errors.

Component by component, we note that the ratio was less than one (and therefore a win for our approach) for both the subset and counting components. For the other two components, Apriori was more efficient. As mentioned in Section 5, these components are negligible, and the ratio was close to one (in the candidate generation component, the ratio was 1.02). The reason is that, for the original Apriori algorithm, the frequent itemsets used for generating the candidates for the next pass are already in the hash tree, whereas in our case, we read in the frequent itemsets generated from applying the modified Apriori to the primary tables. This leads us to consider improvements in the the Memory saving algorithm. In case all tables are at the same site, we could execute steps 2 and 3 of our algorithm (see Section 4.1) concurrently so that the frequent itemsets from the primary tables are readily available. Of course, there are other considerations, e.g., if we were to execute concurrently, a considerable amount of memory would be needed to store all three hash trees at the same time.

For experiments Test 3 and Test 4, the I/O component was more significant than for the previous experiments, given that the percentage of CPU time was considerably lower, e.g., the percentage of CPU time for Test 4, Apriori approach, was 36%. Again, for Test 4, with minimum support = 0.5%, the time spent by I/O (i.e., total time – CPU time) was 606.11 seconds for the Apriori approach, and 143.93 seconds for the Hybrid approach. Therefore the Hybrid/Apriori ratio was 0.24%. When we compare the size of

the data files for each one of the approaches, we have that the Apriori algorithm performed 5 passes on a 671Mbyte file, and the Hybrid approach performed 3 passes on a 3.4Mbyte file, 5 passes on a 3.8Mbyte file and 2 passes on a 671Mbyte file. By taking the Hybrid/Apriori ratio, we get 0.23%, which is again very close.

We made similar analyses for other experiments, with other support values, and the findings for both CPU and I/O were similar. The results demonstrate that our cost analysis provides us with an accurate estimation of the total cost of the algorithm.

7 Conclusions

In this paper, we examined the issues associated with mining decentralized data. We motivated the need for developing decentralized techniques, and identified problems to be addressed. We described the mining of association rules for the case of a star schema wherein several smaller dimension tables are associated by foreign keys to a central fact table, and presented an algorithm that adapts standard algorithms to work efficiently with such data. In contrast to approaches where the data would have been joined first to form a single table, we exploited the foreign key relationships to develop decentralized algorithms that execute concurrently on the separate tables, and thereafter merge the results. We provided analyses to assess our techniques, and we presented empirical evaluation for particular cases to validate the performance. Our preliminary study suggests that the adapted techniques are viable alternatives to first joining the decentralized tables into a single warehoused table prior to mining.

We restricted our attention to one specific organization of decentralized data – the star schema tables – which represent a case of tables associated by foreign key relationships. There are several ways in which our work needs to be extended. Although we believe that it is feasible to generalize our approach, there are many possible formats of database design (not only the star schema), and many factors that could determine how our approach could be applied (e.g., the memory constraints which would limit the feasible solutions). Also, we considered decentralized data residing in one location, whereas a set of decentralized though related tables may be allocated to reside at several sites in a distributed environment. In such cases, not only are the I/O and processing costs of importance, but also, the communication costs among the sites or processors. Finally, our approach was validated empirically for only a specific type of star schema (i.e., for two dimension tables and one fact table), and further experimentation is necessary to extend the study to realistic cases.. We expect that our research in mining decentralized (perhaps non-warehoused) data will become increasingly important as focus shift toward scaling up to real-life, distributed datasets.

Acknowledgements. The authors would like to thank Ramesh C. Agarwal and Anant Jhingran from the IBM T.J. Watson Research Center for discussions and valuable suggestions in the early stages of this work.

References

[1] R. C. Agarwal, C. C. Aggarwal, V. V. V. Prasad, and V. Crestana. A tree projection algorithm for generation of large itemsets for association rules. IBM Research Report: RJ 21246. IBM Research Division, T.J. Watson Research Center, Yorktown Heights, New York, 1998.

[2] D. Agrawal and A. El Abaddi. Special issue on heterogeneous databases. *ACM Computing Surveys*, 22(3), September 1990.

[3] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of ACM-SIGMOD Int'l Conference on Management of Data*, 1993.

[4] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th Int'l Conference on Very Large Data Bases*, 1994.

[5] S. Brin, R. Motwani, J. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of ACM-SIGMOD Int'l Conference on Management of Data*, 1997.

[6] D. Cheung, V. Ng, A. Fu, and Y. Fu. Efficient mining of association rules in distributed databases. *IEEE Transactions on Knowledge and Data Engineering*, 1996.

[7] B. Dunkel and N. Soparkar. Data organization and access for efficient data mining. In *Proceedings of the 15th IEEE Int'l Conference on Data Engineering*, 1999. To appear.

[8] B. Dunkel, N. Soparkar, J. Szaro, and R. Uthurusamy. Systems for KDD: From concepts to practice. *Future Generation Computer Systems*, pages 231–242, January 1997.

[9] R. Elmasri and S. Navathe. *Fundamentals of Database Systems.* The Benjamin/Cummings Publishing Company, Inc., 1989.

[10] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. From data mining to knowledge discovery: An overview. *Advances in Knowledge Discovery and Data Mining*, 1996.

[11] U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining.* AAAI Press, Menlo Park, CA, 1996.

[12] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *Proceedings of the 21th Int'l Conference on Very Large Data Bases*, 1995.

[13] D. Maier. *The Theory of Relational Databases.* Computer Science Press, 1983.

[14] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, 1991.

[15] J. S. Park, M-S Chen, and P. S. Yu. An effective hash-based algorithm for mining association rules. In *Proceedings of ACM-SIGMOD Int'l Conference on Management of Data*, 1995.

[16] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of the 21th Int'l Conference on Very Large Data Bases*, 1995.

[17] Star Schemas and Starjoin Technology. A Red Brick systems white paper. 1995.

[18] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database Systems Concepts.* Mc Graw Hill, third edition, 1996.

[19] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proceedings of the 21th Int'l Conference on Very Large Data Bases*, 1995.

[20] R. Srikant and R. Agrawal. Mining quantitative association rules in large relational tables. In *Proceedings of ACM-SIGMOD Int'l Conference on Management of Data*, 1996.

[21] T. Teorey. *Database Modeling & Design.* Morgan Kaufmann Publishers, Inc., third edition, 1998.

[22] J. Ullman. *Principle of Database Systems and Knowledge-Based Systems*, volume I. Computer Science Press, 1988.