Optimization and Evaluation Strategies for Decentralized Database Mining*

Viviane Crestana Jensen Nandit Soparkar

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

Abstract

Data repositories, including warehouses, usually store information in multiple inter-related tables for reasons of efficiency in storage space, access time, constraint checks, and even administration. We examine the relatively unexplored mining of such decentralized data: we compute partial results at the separate tables, and merge the results - thereby avoiding expensive join materialization prior to mining. In our previous work [18], our algorithms for finding frequent itemsets for simple Star schema in data warehouses were analyzed and validated experimentally.

In this paper, we apply our techniques to general normalized schema designs, including horizontal and/or vertical partitions, distributed repositories etc. In doing so, the choices available for mining are very large, and we describe our techniques to enumerate, optimize and evaluate decentralized frequent itemset counting - in ways similar to query processing in database systems. Our work has less to do with developing specific new mining algorithms; instead, we provide a framework to apply available mining algorithms and tools efficiently for decentralized data.

Our approach includes an "algebra" over expressions (more for exposition than rigor) to denote the mined information in decentralized datasets. We provide rules for establishing equivalences among expressions, and this allows for expression re-writing to represent alternative mining strategies - each expression reflects several processing techniques. Our framework also provides approaches to optimization which enumerate different ways, at logical and physical levels, to effect the decentralized mining. We describe a cost-based optimization problem and solution strategies for the choices in mining. As in database query optimization, we provide heuristics to optimize among the available mining strategies when the estimates of parameters that affect execution costs are unavailable. We perform empirical validation of our decentralized algorithms, and of our heuristic optimization approach when applied to the TPC-D Benchmark data schema. Our results establish our problem formulation and the validity of our heuristic optimization techniques. Finally, we consider a few extensions of our approach such as the use of indices.

*This paper is a longer, extended version of [17].

1 Introduction

Even though database design impacts the efficiency of *data mining* (DM), few techniques (*e.g.*, see [8, 18]) for mining general designs in which tables are partitioned and decentralized have been examined. Centralized database designs (*i.e.*, with data stored in one central repository, homogeneous with a central administration, and in a single schema) are not typical for most large enterprises: information may be placed among different tables, and in some cases, the tables may reside in different physical locations. In large enterprises, data is often distributed and administered separately even within the same organization. For such environments, design decisions involve data fragmentation, distributions strategies, and data allocation considerations. Even for data warehouses and related data marts, the data is stored in decentralized tables. Similar to query processing and optimization (*e.g.*, see [25]), efficiency reasons suggest that decentralized DM must be studied carefully, and there is need for techniques to choose the best strategy among the numerous possibilities.

For example, consider the current approach of mining data stored in data warehouses. Such data is is typically stored in a *Star schema* [27], in which information is decentralized into facts and dimension tables; such schema represent vertical partitioning due to normalization. Facts form the relationships among the dimensions which contain the attributes about the modeled entities. Typical DM algorithms applied to such decentralized data require that the join of all the tables be materialized before any mining is effected. Since the cost of computing the join is overshadowed by that of DM, this approach may appear acceptable. However, a join results in a table with many more columns and rows (which normalization is used to avoid), and this significantly increases the cost of DM as well (*e.g.*, see [9, 18]).

Our previous work [18] describes decentralized algorithms for finding frequent itemsets (used in association rules and classification algorithms) for Star schema. We compute partial results at the separate tables, and merge them to obtain the final results – thereby avoiding the expensive materialization of joins. In this paper, we describe how available centralized mining techniques may be effectively decentralized to apply to general database schemas (including physically distributed tables) in a similar manner.

We develop a framework consisting of several aspects. It includes an "algebra" (more for exposition than rigor) over expressions that denote the mined information in decentralized datasets. We provide equivalences among expressions, and these are useful in expression re-writing to represent alternative mining strategies – each expression suggests several appropriate ways of processing. Our framework also provides optimization techniques which enumerate different ways, at logical and physical levels, to process decentralized mining. Our approach resembles the re-write techniques for relational query processing (as has been suggested by others – *e.g.*, see [16]). We describe a cost-based optimization problem and solution strategies for the choices in mining, and as in database query optimization, we provide heuristics to optimize among the available mining strategies when the estimates of parameters that affect execution costs are unavailable. Also, we perform empirical validation of our heuristic optimization approach when applied to the TPC-D Benchmark data schema, thereby establishing our problem formulation and the validity of our heuristic optimization techniques. We *emphasize* that our approach is *designed* to be similar to query processing in order to be able to integrate, in the future, to similar techniques; this is a strength of our approach.

2 Motivation

Decentralized schemas may involve tables that are horizontally or vertically partitioned, physically distributed, or even replicated. A simple approach to applying a DM algorithm is to reconstruct the information into a single central table, thereby requiring shipping of information, and performing joins. Alternatively, the DM algorithms may be modified to run at individual sites separately, and thereafter, the results "merged." How precisely to merge these results requires careful consideration, and there are efficiency trade-offs between the traditional and the decentralized approaches. We assume that data is *not* replicated; replication is to be handled outside our framework (*e.g.*, by using query processing techniques).

2.1 A Running Example

We use an example schema[1] to illustrate decentralized DM. The schema contains the following tables:

- *Customer*(*cid*, *name*, *street*, *area*, *age*, *salary*)
- *Demographics*(*area*, *population*, *weather*, *schoolsystem*)
- *Product*(*pid*, *description*, *color*, *size*, *price*)
- *ItemsBought*(*xact#*, *cid*, *pid*, *date*, *qty*)

Figure 1 shows a relevant projection of the tables, and we assume that the quantitative attributes (*i.e.*, age, and monetary amounts) are discretized using an appropriate algorithm (*e.g.*, see [26]). Assume that the *ItemsBought* table is horizontally partitioned (and physically distributed) into two tables, *ItemsBought*1 and *ItemsBought*2, which contain the transactions of customers living in areas x or y, and z respectively. This may illustrate a case where there are retail stores in areas z and x, and area y is nearer to area x than to area z. Our example illustrates how any inter-relationships based on foreign keys to link the tables (*i.e.*, most common database designs) is amenable to our techniques.

Site 1

ItemsBought1		
xactid	cid	pid
1	100	A
2	100	A
3	300	A
4	300	B
5	400	B
6	400	E
7	500	A
8	500	C
9	500	D
10	500	D

Site 3

Demographics			
area	population	weather	schools
x	75K	hot	good
y	100K	mild	medium
z	50K	cold	excellent

Site 5

Product			
pid	color	size	price
A	white	small	30
B	blue	small	50
C	white	med.	45
D	blue	med.	60
E	white	large	55
F	blue	large	70

Site 4

Customer			
cid	salary	area	age
100	100000	x	20
200	55000	z	25
300	100000	y	20
400	20000	y	30
500	50000	x	31
600	100000	z	35

Site 2

ItemsBought2		
xactid	cid	pid
11	200	A
12	200	C
13	200	C
14	600	F

Figure 1: Tables horizontally partitioned and located at different sites.

[1]Primary keys are underlined; entries in *ItemsBought* for the same (*cid*,*pid*) correspond to different *xact#*'s.

3 Background and Related Work

We use the problem of frequent itemset counting – used for of *association rules* (AR) discovery (*e.g.*, see [3]) to develop our approach. In this section, we first introduce the problem of frequent itemset counting and then review available approaches applied to decentralized data. Decentralized DM techniques serve as alternatives to handle the inefficiencies in traditional DM techniques for decentralized datasets. Among the available techniques for decentralized DM, are those for horizontally partitioned data (*e.g.*, [8]), and in our previous work involving simple cases of vertical partitioning [9, 18]. We discuss these approaches in the context of our example given in Section 2.1.

3.1 Frequent Itemset Counting

Given a set of items I, and a set of records (*i.e.*, rows) T, where each row t_j is composed of a subset i_j of the set of items I, the problem is to count all the frequent itemsets (*i.e.*, which meet a user-defined level of frequency of occurrence – a *support* threshold). The datasets effectively have categorical attributes with only the presence or absence of an item being indicated in the tables; such sets constitute market-basket data. The problem of AR introduced in [3] was decomposed into: (1) finding the large (*i.e.*, frequent) itemsets (*i.e.*, which meet a user-defined support threshold); and (2) generating the AR, based on the support counts found in step 1. In [4], the *Apriori* algorithm, improving on [3], performs the counting of itemsets in an iterative manner: in each iteration, a "candidate" set of frequent itemsets is identified, and the table is scanned to count the number of occurrences of each candidate itemset. A set of candidates are computed for the next iteration by using frequent itemsets of size k (k-itemsets) at iteration k. The counting of frequent itemsets is expensive, and research efforts concentrates on this step (*e.g.*, [4, 22, 24, 6, 1, 15]).

In [26], an algorithm was proposed to run on quantitative data; after adequate discretization, an algorithm similar to Apriori counts the frequent itemsets. Most algorithms for finding frequent itemsets in market-basket data can be adapted to apply to quantitative attributes (*e.g.*, see [6]). Therefore, although our domain data may also have quantitative attributes (*e.g.*, in data warehouses), we discuss the available algorithms proposed for market-basket data. Our techniques are easily modified to apply to general attribute types.

3.2 Traditional Centralized Approach

For our example from Section 2.1, traditional approaches to AR discovery using frequent itemset counts work relatively efficiently for finding associations such as (size=large) ⇒ (price=high) within table *Product*. However, the same approach would be inefficient for finding ARs (weather=hot) ⇒ (color=white), or (size=large) ⇒ (age=30..35) across multiple tables. The inefficiencies arise due to the need for the joined table ($Demographics \bowtie Customer \bowtie (ItemsBought1 \cup ItemsBought2) \bowtie Product$) in which there is significant redundancy. For example, the itemset {(age=30..39),(area=x)} that occurs four times in the joined table would be counted four times by traditional algorithms; and yet, it corresponds to just one entry (for primary key $cid = 05$) in the *Customer* table.

3.3 Horizontal Partitioning Approach

There has been some work on parallel/distributed algorithms (*e.g.*, see [8, 33]) for horizontally partitioned databases. The designs considered are limited to one table horizontally partitioned into different processors/sites, and therefore, with each partition having the same schema. In such cases, the amount of information read and being processed is the same as in centralized algorithms (except for message exchanges), but the load is shared. Work on handling vertical partitions of a single table for load sharing chooses partitions of columns of a single table across multiple processors – such that individual processors count the frequent itemsets pertaining to a particular partition.

To apply these algorithms to our example, two joins would need to be materialized: $T_1 = (Demographics \bowtie Customer \bowtie ItemsBought1 \bowtie Product)$ and $T_2 = (Demographics \bowtie Customer \bowtie ItemsBought2 \bowtie Product)$. Thereafter, the algorithm would count the frequent itemsets for table $T = T_1 \cup T_2$ by counting, at each iteration, the candidate sets supports locally at T_1 and T_2 – with an exchange of messages with counts at the end of the iteration. The itemsets from tables *Demographics*, *Customer* and *Product*, that were centrally located, would now need to be counted across multiple locations. Furthermore, although the load is distributed, the amount of information read and processed is the same as if the counting was performed against table T in one location. Therefore, although algorithms such as in [8] is an improvement over the traditional centralized approach, for cases such as our example, where there are both horizontal and vertical partitioned tables, the available distributed algorithms that deal only with horizontal partitions do not take full advantage of the decentralization of tables.

3.4 Decentralized Star Schema Approach

For tables in a Star schema, our decentralized approach [9, 18] finds frequent itemsets efficiently – *e.g.*, across tables *Customer*, *ItemsBought* (not horizontally partitioned) and *Product*. Our approach finds itemsets containing items only from *Customer* and only from *Product*, and then the itemsets for items across all three tables. The final frequent itemsets from our algorithm are exactly the same as those found with any traditional algorithm run on the joined table $T = Customer \bowtie ItemsBought \bowtie Product$.

Our technique in [18], as applied to our complete example with all tables considered, needs the join $Demographics \bowtie Customer$, and the union $ItemsBought1 \cup ItemsBought2$ to be computed in advance. However, first obtaining the join of *Demographics* and *Customer* loses the advantage of only assessing itemsets local to *Demographics*. Similarly, the union of *ItemsBought*1 and *ItemsBought*2 loses out in load distribution, and increases communication by having to ship large tables. Furthermore, if each table is in a separate location, all but one of them may have to be shipped to a central location.

4 Decentralized Approach for General Schemas

In this section, we review our basic decentralized approach and present our decentralized approach as extended to more general schemas, and not only the Star schema as described in [18]. We also discuss how mining tables that are physically distributed can take advantage of our decentralized approach.

4.1 Problem Setting

Let a *primary table* be a relational table containing some non-key attributes; the attributes could be categorical, numerical or boolean. Also, let a *relationship table* be a relational table which does not contain non-key attributes. A relationship table, besides its primary key, 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 entity-relationship diagram [28]. Primary tables can also contain foreign keys to other primary tables. In this case, the table is also referred to as a *primary-relationship table.* In the example presented in Section 2.1, the *Demographics* and *Product* tables are primary tables, *Customer* is a primary-relationship table, and *ItemsBought* is a relationship table.

4.2 Basic Decentralized Computation

Our decentralized solution [18] finds frequent itemsets in each primary (or primary-relationship) table individually, and then proceeds to find frequent itemsets that span multiple tables. We call *Phase I* the process of finding the frequent itemsets that contain items from individual tables, and *Phase II* the process of finding itemsets containing items that span more than one table. We explain our approach for a schema containing two primary tables and one relationship table.

4.2.1 Phase I

The final results of a decentralized algorithm should be exactly the same as the original algorithm run on the joined table T. As such, the support of itemsets in an individual table should reflect the final support that this itemset would have in table T. We achieve this as follows:

1. *Compute weight vectors for primary tables.* Count the number of occurrences of each record of the individual tables in the final joined table. This number is given by the number of times each value for the foreign keys is present in the relationship table. If indices are available, this number of occurrences can be determined by scanning the indices,[2] otherwise, the relationship table needs to be scanned once. The number of occurrences is stored in a *weight vector* for each primary table.

2. *Count frequent itemsets on primary tables.* Find the frequent itemsets for each of the primary tables using any traditional centralized algorithm (*e.g.*, see [26, 6]) with the following modification: instead of incrementing the support of an itemset present in record r by 1, increment by the number of occurrences of r in the relationship table (given by the weight vector computed above).

4.2.2 Phase II

We consider two strategies for counting the itemsets that belong to both primary tables.

- *Memory Saving*

  In the traditional approach, T is formed by joining all three tables, and then an Apriori-like algorithm is run on the joined table. In our approach, all of the itemsets belonging to each individual table have already been counted in Phase I. In Phase II, we then count the itemsets against table T as in the traditional centralized approach, except with a smaller set of candidates (only those which contains items from both tables).

- *I/O Saving*

  Generate candidates from each primary table using a 2-dimensional candidate array, where one dimension corresponds to the set of frequent itemsets found in each of the primary tables during Phase I. For each record in the relationship table, identify all frequent itemsets from each primary table that are present in the corresponding records of the primary tables. These frequent itemsets per record (FIPR) can be computed "on the fly", when a record of the relationship table is read, or can be pre-computed for each record of the primary tables. Finally, increment 1 in position on the candidate array which corresponds to all pairs of itemsets: one from each of the FIPRs computed. After the relationship table is processed in this manner, the candidate array contains the support for all the candidate itemsets for the joined table T that span more than one table.

[2]The use of indices is examined in detail later in Section 9.

Discussion

Both merging strategies save processing time since a frequent itemset consisting of items from only one primary table are counted fewer times than if counted from join table T. This may be verified by simple counting arguments. A disadvantage of the Memory saving strategy is that the joined table T must be stored, and the I/O costs could be higher since extra scans may be necessary. Notice however, that the join can be computed "on the fly" for each iteration of our Memory saving approach. This way, instead of materializing table T and scanning it several times, we only scan the relationship table and build the joined records on the fly. Besides avoiding the materialization of the join, this also reduces the cost of each scan since they are performed on a smaller table (the relationship table is considerably smaller than the joined table).

For the I/O saving strategy I/O costs are saved since multiple passes are done on the smaller tables and not the large table T. A disadvantage is that itemsets spanning more than one table, are not pruned from pass to pass because all itemsets are counted in one scan. This results in the counting of candidates that would not be considered if pruning were done at every step. In the remainder of this paper these are referred to as *false candidates.* If the FIPRs are large, this step may require considerable memory space due to the false candidates. This is especially true for more complex schemas involving when more than 2 primary tables.

4.3 Star Schemas

We consider a schema in which there are n primary tables, the *dimension tables*, and one central relationship table, the *fact table.* Let $T_1, T_2, \cdots, T_n$ be the dimension tables, and T_{1n} be the fact table.

When the fact table has no extra attributes, other than the foreign keys to the primary tables, the extension of our basic algorithm is as follows. For Phase I, we compute weight vectors for the n primary tables, and then count frequent itemsets locally – for each of the n primary tables. For Phase II, the Memory saving strategy is the same as described above. For the I/O saving strategy, we generate an n-dimensional array, and compute FIPRs for each of the n tables. The extension of our basic algorithm to such schema is detailed in [18]. In this section, we examine other situations that arise when mining on a Star schema.

4.3.1 Attributes in the Fact Table

In typical data warehouses, the fact table may contain non-key attributes (*i.e.*, categorical and numerical attributes). In our example, suppose there are extra attributes in the table *ItemsBought*, such as *date* and *qty* (indicating how many products were purchased). When it is desired to mine information across primary tables only, the approach described above suffices. In situations where the mined information includes attributes from the fact table (*e.g.*, finding the association between *salary* and *qty*), we adapt our approach as follows.

1. *Treat each attribute of the fact table as though it were from from a different table.*

   On the first scan of the fact table to compute the weight vectors (step 1 of Phase I), verify the frequent itemsets of size one for the attributes of the fact table. For Phase II, the fact table attributes do not affect the Memory saving strategy, they are considered like any other attribute. For the I/O saving strategy, we add a dimension to the counting array for each new attribute. A large number of categorical or numerical attributes in the fact table, could render this solution infeasible due to the number of dimensions required.

2. *Vertically partition the fact table.*

   The fact table is vertically partitioned into two tables: one containing all the categorical or numerical attributes with a new primary key id_{1n}, and one containing id_{1n} together with all other foreign keys originally present in the fact table. This results in the same schema as before, with one extra table, allowing our algorithm to run in the same manner.

An example schema that resembles a Star schema with non-key attributes in the fact table is given in Section 2.1 where the table *Customer* may be considered as a fact table with respect to table *Demographics*.

4.3.2 Pairwise I/O saving

Considering a Star schema with many dimension tables, it may not be feasible to complete an I/O saving merge in one scan of the fact table. The I/O saving strategy avoids storing the joined table at the expense requiring the memory space to store the n-dimensional array used to count all of the itemsets across tables in one step, as well as sacrificing some ability to prune itemsets. If the n-dimensional array does not fit in memory resorting to the Memory saving algorithm, though allowing the desired pruning, leads to the expense of storing table T and scanning it multiple times. Instead, we present a compromise between I/O and Memory saving, referred to as *Pairwise I/O saving*.

Phase I requires no changes since each table is processed separately. In Phase II, instead of merging all tables in one step, process them pairwise. For example, considering tables T_1, T_2, T_3, and T_{13}, process $T_1 \bowtie T_{13} \bowtie T_2$ using I/O saving merge considering the sets F^1 and F^2 as candidates (and likewise for T_1, T_3, and T_2, T_3). This way, pruning is increased considering that the false candidates involving attributes of two tables will be known. Although there are multiple scans of the fact table, the join need not be materialized, and the number of candidates is significantly reduced, thereby increasing the efficiency of the merge step. In Section 8, we show this technique to be efficient in some cases.

4.4 Snow Flake Schemas

We now consider an example involving a primary-relationship table. We revisit our schema shown in Section 2.1. that contains two primary tables, one primary-relationship and one relationship table. Our technique, described thus far requires the join *Demographics* $\bowtie$ *Customer* to be computed first. However, this loses the advantage of only assessing itemsets local to *Demographics*. Instead, we show how to adapt our algorithm to count the items from a table that is not directly related to the relationship table (in this case, the *Demographics* table), without resorting to pre-computing its join with a table that does (in this case, the *Customer* table).

First, let us consider Phase I. To count itemsets in *Demographics* separately, a weight vector for table *Demographics* with respect to *ItemsBought* must be computed. This weight vector (wv_D) is computed using the table containing the foreign key to *Demographics* (table *Customer*) and its weight vector (wv_C). We explain with our example. The weight vector for *Customer* was $(2,3,2,2,4,1)$. The first record of *Demographics* ($area = x$) occurs 6 times in the joined table: 2 due to the first record of *Customer* ($cid = 100$), and 4 due to the fifth record of *Customer* ($cid = 500$). Thus, the value in wv_D for ($area = x$) is the sum of the values in the wv_C for ($cid = 100$) and ($cid = 500$). The final weight vector for *Demographics* is: $(2+4, 2+2, 3+1) = (6,4,4)$. In this manner, it is possible to determine the number of times each record of *Demographics* appears in the final table, and therefore, to effect counting in *Demographics* separately.

Now, consider Phase II. For Memory saving, there are no changes other than considering more than two primary tables (as in the case of Star schema describe previously) – in this example, three. For an I/O saving merge, there are more possibilities, and we discuss each of them. The first option is to proceed as in the Star schema case adding one dimension to the array to represent table *Demographics*. Since *area* (the primary key for *Demographics*) is not in *ItemsBought*, we use table *Customer* to get the corresponding record of table *Demographics* for each entry in *ItemsBought*. Another option is to take advantage of the fact that *Customer* is directly related to *Demographics*. We call this approach *Combined I/O saving*, since we combine tables *Customer* and *Demographics* without, however, joining them. In this case, we perform the merge in the following two steps:

1. Compute the itemsets from *Customer* and *Demographics* up-front. This counting is performed without obtaining any information from table *ItemsBought* (other than the weight vectors). Create a 2-dimensional array with frequent itemsets from tables *Customer* and *Demographics*. Then, for each record r of table *Customer* we identify the FIPR for *Customer*, I^C, and the FIPR for the record of *Demographics* related by the foreign key, I^D. Then identify all the pairs from I^C and I^D and add to the corresponding position in the 2-dimensional array the value in wv_C for record r. The reason for adding this value, instead of one (as in the typical I/O saving merge), is that this weight vector entry reflects how many times these two records would have been merged if we were computing the merge by reading *ItemsBought* instead.

2. Process table *ItemsBought*. Merge tables *Product* and *Customer* scanning table *ItemsBought* as before, with the difference that, for each record of table *Customer*, the FIPRs include the itemsets from both tables *Customer* and *Demographics* (merged in step 1 above).

In a general Snow Flake schema, there could be more than one table that does not relate to the fact table directly. Each of these tables would be processed as with table *Demographics* above.

4.5 Schemas with no Central Relationship Table

We call *foreign key based* decentralization when the tables are to be joined based on the foreign key relationships, *i.e.*, the join is a *natural join*. In general, if two tables, say T_i and T_j, are related by a foreign key (say id_i is a primary key in table T_i and a foreign key in table T_j) it is possible to directly apply our approach of decentralized counting (as shown in Section 4.4). As an example of a less trivial foreign key based decentralized schema, consider a series of binary relationships (as opposed to the one n-ary relationship exhibited in the Star schema). That is, consider n primary tables $(T_1, T_2, \cdots, T_n)$ and $n-1$ relationship tables $(T_{12}, T_{23}, \cdots, T_{(n-1)n})$.

Again, the goal is to find association rules in the final joined table $T = T_1 \bowtie T_{12} \bowtie T_2 \bowtie \cdots \bowtie T_{n-1} \bowtie T_{(n-1)n} \bowtie T_n$. It may seem intuitive, but would be erroneous, in general, to perform a straight-forward combination application of our approach (*i.e.*, $T_t \bowtie T_{t(t+1)} \bowtie T_{t+1}$). This is apparent when the number of occurrences of records in the individual relationship tables is different as compared to the number of occurrences in table T.

To count the occurrences in table T of the records in the primary tables, one approach is to first join $T_{12} \bowtie T_{23} \bowtie \cdots \bowtie T_{(n-1)n}$. This joined table is equivalent to the fact table in the Star schema, allowing the same procedure as for the Star schema. Another option is to take advantage of the smaller binary relationship tables and run our algorithm in combination, similar to the approach described in Section 4.4. In order to run the algorithm in combination, count the occurrences of the records from the relationship tables, T_{ij} in table T in the same manner that occurrences of the records from the dimension tables were counted in the fact table in Section 4.2. Then, when computing a weight vector from a given relationship table (*e.g.*, T_{12}), take into account, in addition to the occurrences of the foreign keys (*e.g.*, id_1 and id_2), the number of times a record in the relationship table appears in T. Thereafter, run the modified Apriori on the primary tables, and then in combination (*e.g.*, finding association rules for $T_1 \bowtie T_{12} \bowtie T_2$). After all the combined counting is performed, find the frequent itemsets for all the tables using table T (or re-calculate T in case the joined table T was not stored in the first instance). The above exemplifies our possible approaches to more general foreign key based decentralization.

4.6 Distribution Factors

In all examples above, we considered decentralized data residing in one location. The Star schema tables, and indeed, any set of decentralized though related tables, may be allocated to reside at several sites in a distributed environment (as illustrated in Section 2.1).

Our general approach of distributing the mining algorithms potentially provides many performance improvements. When the primary tables reside in different locations, ideas from semi-join algorithms[21] may be applied to effect Phase I by sending the information gathered in step 1 to the primary table sites so step 2 may be performed locally (thereby distributing the load among the different locations). For the I/O saving merge strategy, the FIPRs would be gathered at an appropriate site (depending on distribution and according to the semi-join strategy), as opposed to shipping the entire primary table – necessary in the traditional approach to frequent itemset counting.

Our techniques, as described thus far, applied to our example shown in Section 2.1, require the the union $ItemsBought1 \cup ItemsBought2$ to be computed first. However, doing so decreases load distribution and increases communication by having to ship large tables. Furthermore, if each table is in a separate location, all but one of them may have to be shipped to a central location. We adapt our algorithm for disjoint horizontal partitions as follows (where we assume each table is at a different site). For Phase I, compute the weight vectors from $ItemsBought1$ and $ItemsBought2$, and send them to the sites with $Customer$ and $Product$. At each site, the weight vectors from the two $ItemsBought$ partitions are combined by a simple vector addition. The second part of Phase I proceeds as previously described, *i.e.*, frequent itemsets are counted locally. For Phase II, FIPRs for $Customer$ and $Product$ (or the tables themselves as would be needed if using the traditional horizontally distributed approach [8]) are sent to each of the locations of $ItemsBought1$ and $ItemsBought2$. Then, the Phase II counting may proceed concurrently at the sites for $ItemsBought1$ and $ItemsBought2$. Thereafter, messages are exchanged to determine the final frequent itemsets.

5 Algebra Expressions and Equivalences

In [18], we analyzed the costs of a few implementation strategies for the Star schema. However, there are other options for decentralization that were not considered, such as the ones mentioned in Section 4: Pairwise and Combined I/O saving. In order to assess which general approach is better, centralized or decentralized, we need to analyze all of the possible decentralized alternatives. The choices available for mining are numerous, and we describe our techniques to enumerate, optimize and evaluate decentralized frequent itemset counting – similar to what is done for query processing.

5.1 Decentralization: Many Alternatives

Decentralized DM techniques serve as alternatives to handle the inefficiencies in traditional DM techniques for decentralized datasets. Among the available techniques for decentralized DM are those for horizontally distributed data (*e.g.*, see [8]), and for normalized (and even physically distributed or not) data as presented in Section 4.

There are several alternatives to materializing the joins or unions when dealing with such decentralized tables. For instance, some subset of tables could be joined or unioned before starting decentralized mining, thereby providing at least as many alternatives for DM as there are for joins or unions. Furthermore, there are additional possibilities for decentralized DM, such as the merge strategy used. Although our example has relatively few options for decentralized DM, more complex designs could have a large number of options – over and above those available for join processing of distributed queries.

To choose among alternatives effectively, there is a need to unify the decentralized DM approaches for horizontally and vertically partitioned tables, and we describe ways to do so. Each strategy involves its own cost of execution, and in order to choose the best strategy, we need a simple means to enumerate the possibilities and choose the best alternatives based on cost metrics. Our approach can easily be used in conjunction with available mining tools and techniques. Furthermore, the optimization and evaluation techniques exemplified below can be applied at a level above the actual mining algorithms: for the most part, they manipulate the tables that are then provided to the basic algorithms.

We provide an "algebra" over expressions which represent the mined information at a particular stage. In the expressions, we are primarily interested in itemsets whose items are non-key attributes; the key attributes inter-relate the tables. Our algebra facilitates enumerating and choosing among alternatives similar to the way relational algebra is used for standard query processing.

5.2 Algebra Expressions

The basic notation for our algebra is as follows; the purpose of the term W is clarified below.

- $FI(X, W)$: set of frequent itemsets from table $X \bowtie W$ involving only non-key attributes of table X.
- $CI_m(\{X_1, X_2, \cdots, X_n\}, W)$: set of *cross-itemsets*, which are the frequent itemsets from table $X_1 \bowtie X_2 \bowtie \cdots \bowtie X_n \bowtie W$ involving non-key attributes of table $X_1 \bowtie X_2 \bowtie \cdots \bowtie X_n$, and where an itemset contains items from at least m tables (note: when $m > n$, CI_m is the empty set). For simplicity, when $m = 2$ it is omitted from the notation.

In the above, the term W is for the *weight table*; in $FI(X, W)$, it denotes the table from which the weight vector is computed, and in $CI_m(\{X_1, X_2, \cdots, X_n\}, W)$, it denotes the table that links the X_i tables.

Let I denote the identity operand for the join operator (*i.e.*, $I = \pi_{keyofT}(T)$ such that $T \bowtie I = T$). To illustrate the use of our algebra, consider the case in Section 3.2 where $FI(T, I)$ was to be computed, where $T = Demographics \bowtie Customer \bowtie (ItemsBought1 \cup ItemsBought2) \bowtie Product$. In Section 4.2, we computed $FI(Customer, ItemsBought)$ and $FI(Product, ItemsBought)$ in Phase I, and $CI(\{Customer, Product\}, ItemsBought)$ in Phase II.

5.3 Expression Equivalences

Our enumeration approach begins with an expression such as $FI(T, I)$, where T represents the table arising from joining (or creating the union) of all the decentralized tables. Thereafter, we aim to obtain expressions that represent decentralized DM that finds frequent itemsets in separate tables, and merges the partial results. Using the equivalences below, we obtain alternative expressions that are equivalent, and represent implementations that are often less expensive than using traditional techniques which materialize T.

Equiv 1

$$FI(X \bowtie Y, W) \;=\; FI(X, X \bowtie Y \bowtie W) \,\cup\, FI(Y, X \bowtie Y \bowtie W) \,\cup\, CI(\{X, Y\}, X \bowtie Y \bowtie W)$$

Proof Sketch: The left expression represents the set of frequent itemsets for table $X \bowtie Y \bowtie W$. The right expression represents the same set as a union of three subsets: the set of frequent itemsets from $X \bowtie Y \bowtie W$ involving only items from X, only items from Y, and from both X and Y. Each term on the right represents a (usually proper) subset of the expression on the left, and their union evaluates to it as well.

Equiv 2

$$CI_m(\{X, Y, Z\}, W) \;\; = \;\; CI_m(\{X, Y \bowtie Z\}, W) \; \cup \; CI_m(\{Y, Z\}, W)$$

Proof Sketch: Similar arguments as Equiv 1.

Equiv 3

$$\begin{aligned} CI_m(\{X_1, \ldots, X_n\}, W) \;\; = \;\; & CI_m(\{X_1, \ldots, X_{n-1}\}, W) \; \cup \; \cdots \; \cup \\ & CI_m(\{X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n\}, W) \; \cup \; \cdots \; \cup \\ & CI_m(\{X_2, \ldots, X_n\}, W) \; \cup \; CI_n(\{X_1, \ldots, X_n\}, W) \end{aligned}$$

Proof Sketch: All CI terms are with respect to the joined table $X_1 \bowtie X_2 \bowtie \cdots \bowtie X_n \bowtie W$. The CI expression on the left of the equality represents all frequent itemsets where an itemset contains items from at least m tables. The last CI term (CI_n) represents all frequent itemsets where an itemset contains items from all n tables. The difference between this term and the one on the left are the frequent itemsets spanning from m up to $n-1$ tables. The remaining term on the right (*i.e.*, all but the last) represent exactly these itemsets: including items from m thru $n-1$ tables. Some frequent itemsets could be double counted, but the expression is still correct since we are taking the union. However, no redundancy occurs when $m = n-1$.

Equiv 4

$$FI(X, W) = FI(X, \pi_{id_x}(W))$$

where id_x is the primary key of X, and π is a modified project operation from relational algebra, such that duplicates are not removed.

Proof Sketch: Since $FI(X, W)$ is the set of frequent itemsets involving only non-key attributes of table X, all attributes of W other than the primary key of X (*i.e.*, the attribute that inter-relates the two tables) are not relevant to the computation of $FI(X, W)$.

Equiv 5

$$CI_m(\{X, Y\}, W) = CI_m(\{X, Y\}, \pi_{id_x, id_y}(W))$$

Proof Sketch: Similar arguments as Equiv 4.

The usual equivalences in relational algebra (*e.g.*, commutativity and associativity of joins) continue to hold. For example, we have

Equiv 6

$$FI(X \bowtie (Y_1 \cup Y_2) \bowtie Z, W) = FI((X \bowtie Y_1 \bowtie Z) \cup (X \bowtie Y_2 \bowtie Z), W)$$

Proof Sketch: From relational algebra equivalences – a join distributes over unions.

Equiv 1 and Equiv 2 indicate how increasing degrees of decentralized DM may be introduced into an expression: the right expressions are more decentralized as compared to the left. The expression in Equiv 3 corresponds to merging tables in a Pairwise manner (Section 4.3). Equiv 4 and Equiv 5 are mainly useful for the algebraic manipulations, and for indicating how to compute the weight vectors. Equiv 6 is useful in considering horizontal partitions. Note that, once a join is distributed over a union, local frequent itemsets for tables X and Z can no longer be computed locally, which may prove to be counter productive. We re-visit the issue of distributing joins over unions later in Section 7.

5.4 Expression Re-writing

We illustrate the use of our equivalences for our example from Section 2.1 with horizontal partitions (Figure 1). Again, we aim to find $FI(T, I)$, where $T = \mathit{Demographics} \bowtie \mathit{Customer} \bowtie (\mathit{ItemsBought}1 \cup \mathit{ItemsBought}2) \bowtie \mathit{Product}$, which we shorten to $T = D \bowtie C \bowtie (B1 \cup B2) \bowtie P$.

The traditional centralized approach provides a first alternative to the computation of $FI(T, I)$:

$$FI(D \bowtie C \bowtie (B1 \cup B2) \bowtie P, I) \tag{1}$$

Our equivalences help develop a decentralized processing plan. By distributing the join over the union we get

$$FI((D \bowtie C \bowtie B1 \bowtie P) \cup (D \bowtie C \bowtie B2 \bowtie P), I) \tag{2}$$

The expression (2) limits processing to algorithms meant solely for horizontally partitioned cases. Instead, by using the commutativity of joins and Equiv 1 to expression (1) , we have

$$\begin{gathered} FI(B1 \cup B2, D \bowtie C \bowtie (B1 \cup B2) \bowtie P) \;\cup\; FI(D \bowtie C \bowtie P, D \bowtie C \bowtie (B1 \cup B2) \bowtie P) \;\cup \\ CI(\{B1 \cup B2, D \bowtie C \bowtie P\}, D \bowtie C \bowtie (B1 \cup B2) \bowtie P) \end{gathered}$$

Since $B1 \cup B2$ contains only key attributes, $FI(B1 \cup B2, W)$ is empty – irrespective of the elements of W. Also, $CI(\{B1 \cup B2, X\}, W)$ is empty since the frequent itemsets in CI should have at least one non-key attribute from $B1 \cup B2$. So, the above expression reduces to

$$FI(D \bowtie C \bowtie P, D \bowtie C \bowtie (B1 \cup B2) \bowtie P) \tag{3}$$

Tables D and C do not have common attributes with P, therefore, their join is a Cartesian product. While this may seem a bad choice, applying more equivalences lead to better expressions. In particular, using Equiv 1 on the above expression leads to

$$\begin{gathered} FI(D, D \bowtie C \bowtie (B1 \cup B2) \bowtie P) \;\cup\; FI(C \bowtie P, D \bowtie C \bowtie (B1 \cup B2) \bowtie P) \;\cup \\ CI(\{D, C \bowtie P\}, D \bowtie C \bowtie (B1 \cup B2) \bowtie P) \end{gathered} \tag{4}$$

Since T has the same number of records as $B1 \cup B2$, which has foreign keys to all tables (except D), we have

$$\begin{aligned} \pi_{cid}(D \bowtie C \bowtie (B1 \cup B2) \bowtie P) &= \pi_{cid}(B1 \cup B2) \\ \pi_{pid}(D \bowtie C \bowtie (B1 \cup B2) \bowtie P) &= \pi_{pid}(B1 \cup B2) \\ \pi_{area}(D \bowtie C \bowtie (B1 \cup B2) \bowtie P) &= \pi_{area}(C \bowtie (B1 \cup B2)) \end{aligned}$$

And so, using Equiv 4, Equiv 5, and the above equalities we arrive at a new expression

$$FI(D, (C \bowtie (B1 \cup B2)) \;\cup\; FI(C \bowtie P, (B1 \cup B2)) \;\cup\; CI(\{D, C \bowtie P\}, ((B1 \cup B2) \bowtie C)) \tag{5}$$

Using Equiv 1 in the second term leads to

$$\begin{gathered} FI(D, (C \bowtie (B1 \cup B2)) \;\cup\; FI(C, (B1 \cup B2)) \;\cup\; FI(P, (B1 \cup B2)) \;\cup \\ CI(\{C, P\}, (B1 \cup B2)) \;\cup\; CI(\{D, C \bowtie P\}, ((B1 \cup B2) \bowtie C)) \end{gathered} \tag{6}$$

Using Equiv 2 in the last two terms leads to

$$\begin{gathered} FI(D, (C \bowtie (B1 \cup B2)) \;\cup\; FI(C, (B1 \cup B2)) \;\cup\; FI(P, (B1 \cup B2)) \;\cup \\ CI(\{D, C, P\}, ((B1 \cup B2) \bowtie C)) \end{gathered} \tag{7}$$

The various expressions, equivalent to $FI(T, I)$, represent ways of mining that avoid the initial materialization of the join $D \bowtie C \bowtie (B1 \cup B2) \bowtie P$. In fact, expression (7) significantly decentralizes the computation, and avoids the union of the horizontal partitions.

As noted in Section 4, merging the results from many tables simultaneously (*i.e.*, computing CI) may prove expensive due to memory space requirements, or due to counting of extra "false candidates" in the I/O saving merge. However, results may be merged over several steps using multiple passes over the weight table. One way to reduce the number of false candidates is to process the tables pairwise. Applying Equiv 3 in the last term of (7) results in a Pairwise strategy

$$\begin{gathered} FI(D, (C \bowtie (B1 \cup B2)) \ \cup \ FI(C, (B1 \cup B2)) \ \cup \ FI(P, (B1 \cup B2)) \ \cup \\ CI(\{D, C\}, ((B1 \cup B2) \bowtie C) \ \cup \ CI(\{C, P\}, (B1 \cup B2)) \ \cup \\ CI(\{D, P\}, ((B1 \cup B2) \bowtie C) \ \cup \ CI_3(\{D, C, P\}, ((B1 \cup B2) \bowtie C)) \end{gathered} \tag{8}$$

There are other ways to merge in steps, such as using expression (6), or by materializing the join of a few tables in advance. For example, we could join D and C early (*i.e.*, by applying Equiv 1 to expression (3)) to get

$$FI(D \bowtie C, (B1 \cup B2)) \ \cup \ FI(P, (B1 \cup B2)) \ \cup \ CI(\{D \bowtie C, P\}, (B1 \cup B2)) \tag{9}$$

Then, applying Equiv 1 to the first term leads to

$$\begin{gathered} FI(D, (C \bowtie (B1 \cup B2)) \ \cup \ FI(C, (B1 \cup B2)) \ \cup \ FI(P, (B1 \cup B2)) \ \cup \\ CI(\{D, C\}, ((B1 \cup B2) \bowtie C)) \ \cup \ CI(\{D \bowtie C, P\}, (B1 \cup B2)) \end{gathered} \tag{10}$$

Figure 2 depicts the various equivalent expressions that we obtain. We do not explicitly show the weight table for clarity[3]. Note that all expressions, except for expression (8) can be depicted as a tree. (expression (8), Pairwise merging, is similar to (7) in the sense that there are 3 FIs and one 3-way merge – the 2-way merges are used as pruning steps for the 3-way merge, denoted in the picture by dotted lines).

Our discussion shows that, even for a small problem involving only four tables, there are many possible strategies to obtain the frequent itemsets. In fact, all possibilities were not enumerated (*e.g.*, we did not consider the prejoin $D \bowtie P$). By assigning costs to each operation, each expression may be associated with a processing cost, and an optimization problem arises.

6 Incorporating Cost Estimates

We briefly review costs involved in computing FI, CI and $\cup$. The join operation is also discussed since several strategies involve materializing joins prior to computing FI or CI. Detailed cost estimates are not discussed since they depend on the particular algorithm employed, and our goal is to provide a framework where such costs can be utilized if needed. An example of detailed cost estimates was provided in [18] for a n-way merge approach to a Star schema. Also, we discuss how to estimate parameters that affect the mining, and provide cost estimates for our discussed examples.

For our description of costs, we use the following definitions and assumptions. Let r_i be the number of records in table T_i; m_i be the number of attributes in table T_i; k_i be the length of the longest candidate itemset on table T_i; $|c^i_j|$ be the number of candidates of length j from table T_i; and $|l^i_j|$ be the number of frequent itemsets of length j from table T_i. For a term such as $FI(T_i, T_w)$, T_i or T_w may have to be computed in advance (*e.g.*, if $T_i = T_{i_1} \bowtie T_{i_2}$, then the join needs to be computed; or if $T_i = T_{i_1} \cup T_{i_2}$ the union needs to

[3]In many cases (especially data warehouses), indices are available, so the size of the weight table is less of an issue. The use of indices is discussed in Section 9.

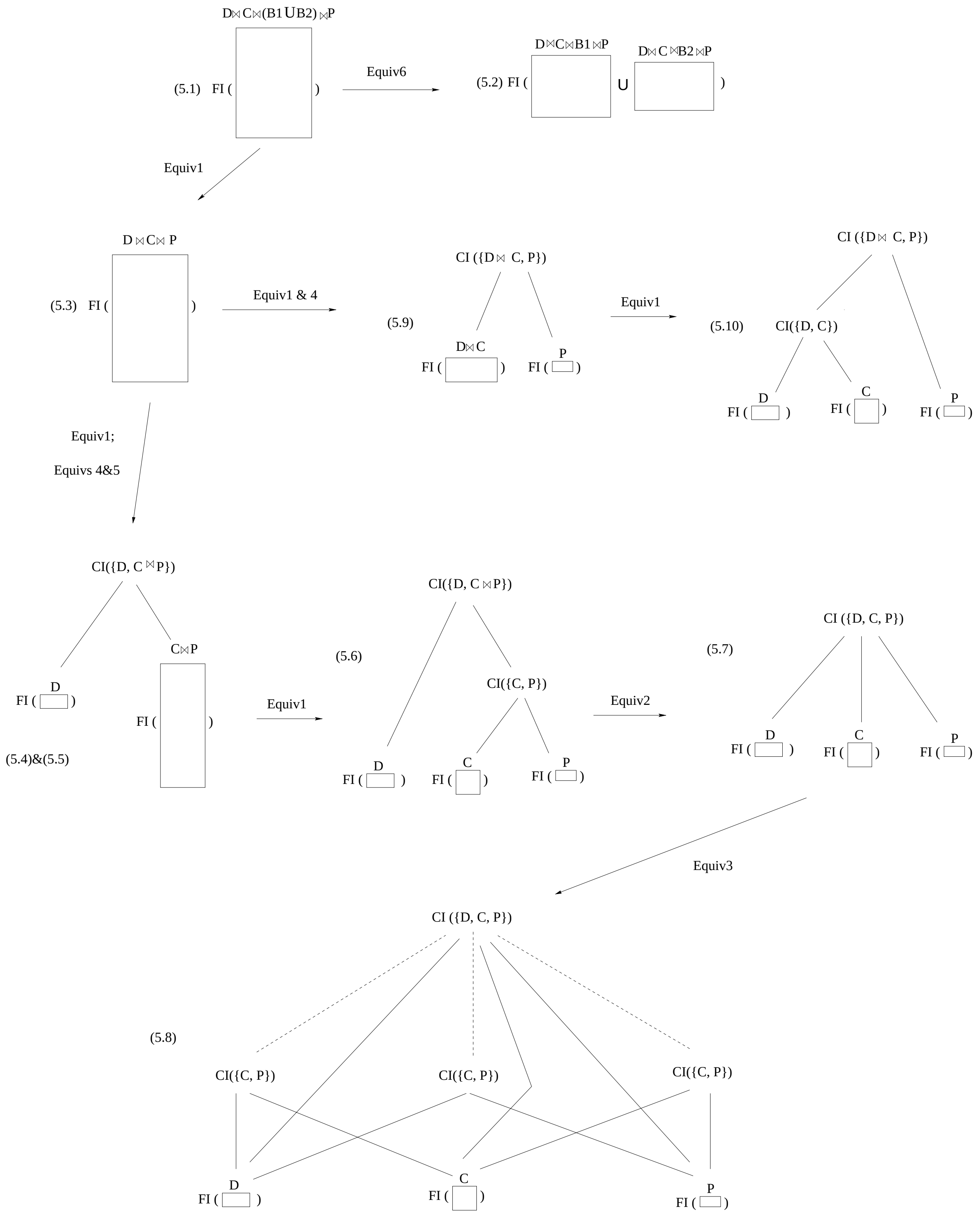


Figure 2: Different expressions that arise.

be computed). We do not include computing costs for T_i and T_w – which are discussed separately. Also, for cases where the union would not be computed (*e.g.*, $FI(T_{i_1} \cup T_{i_2}, T_w)$), and instead, an algorithm would be used that works on horizontally distributed data (*e.g.*, from [8]), we leave the union symbol stated explicitly.

We separate the individual operation processing costs into local costs (I/O and CPU), and communication costs. In several cases, indices may be used effectively, and though we briefly touch on examples with indices, we leave the discussion of how indices affect the algorithms and costs to Section 9.

6.1 Phase I Costs

6.1.1 $FI(T_i, T_w)$

The costs depend on the algorithm used. We may compute the weight vector from T_w, and then, apply a centralized algorithm (*e.g.*, see [4, 6, 24]) with modifications as mentioned in Section 4.2. Therefore, the costs include computing the weight vector ($opWV$), and counting of the frequent itemsets of T_i ($opFI$). For simplicity, I/O costs are associated with the size of data moved from disk (and not number of disk blocks).

1. $opWV$: *Computing the weight vector from* T_w

   - *Local costs:* If an index is not available, the weight vector needs to be computed from T_w, so the cost is given by $r_w m_w$. If an index on T_w is available for each record of T_i, we only need to read the index; in such cases, the costs are given by the number of entries in the index, r_w.
   - *Communication costs:* None, since the weight vector is computed locally (it either scans the index, or scans T_w).

2. $opFI$: *Computing the frequent itemsets in* T_i

   - *Local costs:* These include costs for multiple scans of T_i whose size is $m_i r_i$, and of the weight vector if it does not fit in main memory. For the Apriori algorithm [4], for example, the number of scans is given by k_i, so the I/O costs are: $k_i m_i r_i + k_i r_i$. With decentralized DM, the smaller tables may fit in main memory, which would reduce I/O substantially. The CPU costs, as in the traditional centralized approach, depend on $|c_j^i|$ for each iteration, the length m_i of the record that needs to be checked against the candidates, and the data structures used in the counting. Costs associated with the subset operation will dominate the CPU costs.
   - *Communication costs:* None, if T_w and T_i are co-located. Else, if T_w is remote from T_i, there is communication cost in shipping the weight vector of size r_i to T_i.

6.1.2 $FI(T_i, T_{w_1} \cup T_{w_2} \cup \cdots \cup T_{w_l})$

The union in this term allows using our decentralized approach of Section 4.2 with the modification of Section 4.6.

1. $opWV$: *Computing the weight vector from* $T_{w_1} \cup T_{w_2} \cup \cdots \cup T_{w_l}$

   - *Local costs:* These are as for $FI(T_i, T_w)$, except that the load is shared by the different locations of $T_{w_q}, q = 1..l$; the I/O and CPU costs are given by r_{w_q}.
   - *Communication costs:* None, since the weight vectors are computed locally.

2. $opFI$: *Computing the frequent itemsets in* T_i

- *Local costs:* Composing the total weight vector from those arising from $T_{w_1}, T_{w_2}, \cdots T_{w_l}$ involves a simple vector addition, and its cost depends on the number of horizontal partitions, l, and the total number of records, r_i. Thereafter, the costs are as in *opFI* for $FI(T_i, T_w)$.
- *Communication costs:* Since the horizontal partitions are distributed, at most one T_{w_q} is co-located with T_i. Therefore, there is communication cost in shipping the l (or $l-1$) weight vectors of size r_i to T_i.

6.1.3 $FI(T_{i_1} \cup T_{i_2} \cup \cdots \cup T_{i_n}, T_w)$

The union in this term requires using a distributed algorithm that synchronizes with message exchanges after each iteration, and we use the algorithm from Cheung *et al.* [8] as an example.

1. *opWV*: *Computing the weight vector from* T_w

   As above in Section 6.1.1.

2. *opFI*: *Computing the frequent itemsets in* $(T_{i_1} \cup T_{i_2} \cup \cdots \cup T_{i_n})$

   - *Local costs:* Let $T_i = T_{i_1} \cup T_{i_2} \cup \cdots \cup T_{i_n}$. At each pass the amount of local processing across all sites is approximately the same with T_i computed in advance (since at each site the candidates are the same as for table T_i). The local costs which are related to the number of records present, depend on $|c_j^{i_p}|, p = 1..n$, m_{i_p} and k, which are the same for all sites. The sum of the records present across all sites is the same as in T_i, and therefore, the local costs are basically the same as in Section 6.1.1.
   - *Communication costs:* The computed weight vector is horizontally partitioned, and the appropriate vectors are sent to $T_{i_q}, q = 1..n$. Therefore, this communication cost depends on n and r_i. The communication cost should also include shipping the candidate set counts after each pass, and each site must ship this information to every other site – leading to $\sum_{j=1}^{k}(|c_j^i|) * p$ in communication costs.

6.2 Phase II Costs

6.2.1 $CI(\{T_{i_1}, T_{i_2}, \cdots T_{i_n}\}, T_w)$

These costs depend on the algorithm used on Phase II. In the following discussion, we consider the I/O saving approach as explained in Section 4.3 as an example.

- *Local costs:* In the I/O saving approach, each record of T_w is processed once, and if indices are available (see Section 9), or smaller tables fit into main memory, the I/O costs are the scans of each of the tables. For the CPU costs, for each record s in T_w, the cost is in determining which frequent itemsets found in the T_{i_p} are present in s. If FIPRs are pre-computed, the subset operation is only applied to smaller tables, and therefore, negligible: for each record in table $T_{i_p}, p = 1..n$, we check m_{i_p} against $\sum_{j=1}^{k_{i_p}} |l_j^{i_p}|$, and this check is effected r_p times. The main CPU costs are for counting the frequent itemsets, which depends on the sizes of the FIPRs.

- *Communication costs:* These costs include sending the T_{i_p} (accounted for in effecting the join) as well as the frequent itemsets found for each T_{i_p} – to the site where *opCI* is executed. The latter requires shipping $\sum_{p=1}^{n}(\sum_{j=1}^{k_{i_p}} |l_j^{i_p}|)$ elements which is the latter's communication costs.

6.2.2 $CI(\{T_{i_1}, T_{i_2}, \cdots T_{i_n}\}, (T_{w_1} \cup T_{w_2} \cup \cdots \cup T_{w_l}))$

Again, we choose the I/O saving approach to illustrate.

- *Local costs:* The costs are as in Section 6.2.1, except that the load is distributed. The only extra cost is for adding the partial counters computed at each location (*i.e.*, a matrix addition operation). The I/O cost at each partition is the one scan of T_{w_q} and the scan of the smaller tables. For the CPU costs, again counting dominates (assuming FIPRs are pre-computed). The matrix addition operation depends on l and the matrix size $\pi_{p=1}^{n} \sum_{j=1}^{k_{i_p}} |l_j^{i_p}|$.
- *Communication costs:* As in Section 6.2.1, except that additional communication is needed for the final addition of matrices (*i.e.*, the local matrix counters must be sent to a central location). Again, this depends on the size of the matrix.

6.3 Union Costs

We discuss the following situations involving unions which arise in our approach.

1. Composing the set of frequent itemsets (*i.e.*, union of FI's and CI's)
    - *Local costs:* Since we assume that FI and CI are disjoint, these costs are negligible.
    - *Communication costs:* The unions compose sets generated by *opFI* and *opCI* operations. For *opCI*, all generated frequent itemsets from *opFI* are sent to the site where *opCI* is executed, and this communication cost is account for in *opCI*.
2. Computing the union of tables in advance.

   In this case, costs from traditional query processing apply (*i.e.*, shipping the tables).

6.4 Join Costs

As in traditional query processing, there are many ways to perform joins required in decentralized DM. Choices available in join ordering and strategies lead to different costs. Therefore, the local and communication costs for the join are the traditional ones from query processing (*e.g.*, see [21]).

6.5 Choosing among Alternatives

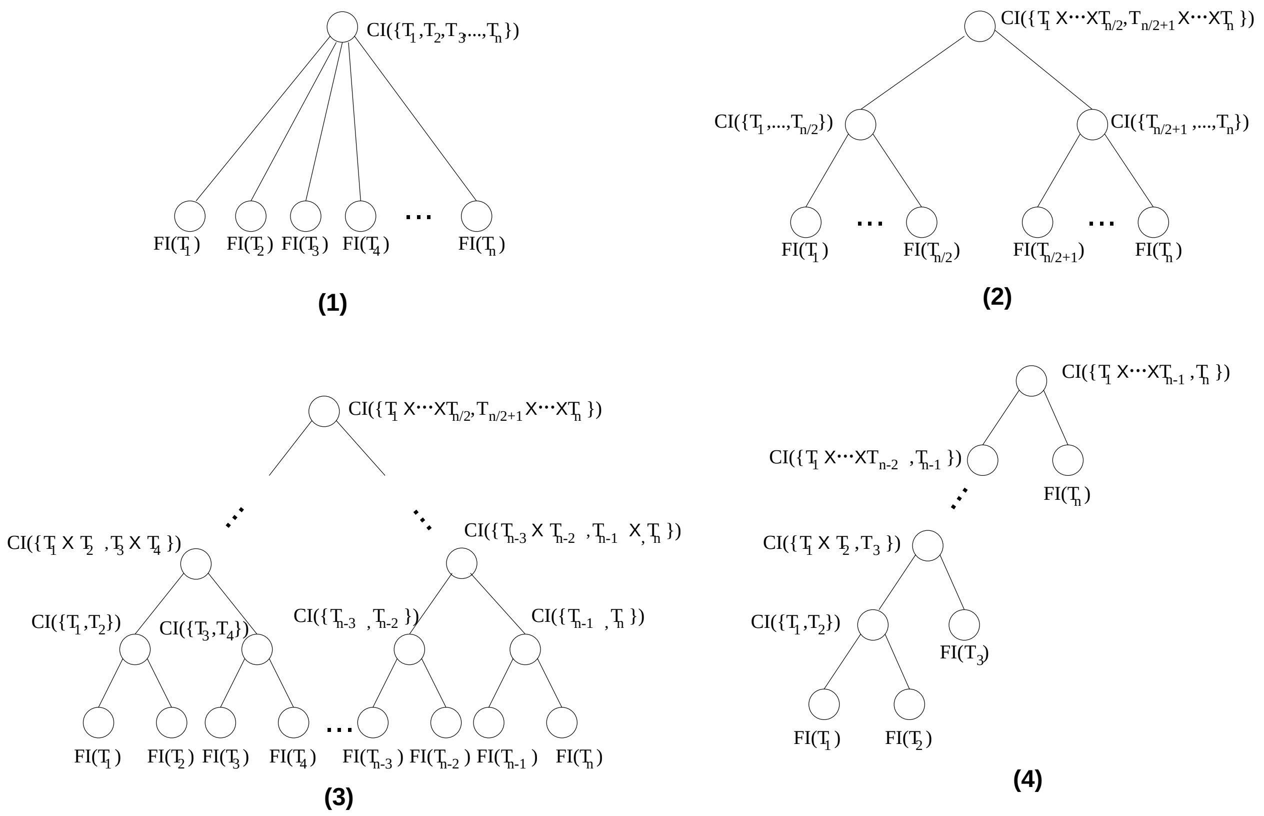
Above, we presented cost estimates for our decentralized approach. In this section, we show how to use cost estimates in optimizing decentralized DM. We consider enumerating expressions in different ways, both at the logical (*i.e.*, the particular FI's and/or CI's) and at the physical (*i.e.*, which algorithm to use) levels. The steps are summarized as follows.

(a) Enumerate expressions $e_i, i = 1..n$ equivalent to $FI(T, I)$;

(b) Compute costs associated with each e_i; and

(c) Find the e_i with minimal costs.

We show how to enumerate expressions, and to estimate how many unique expressions exist. We then show an efficient way of searching for the minimal cost plan.



CI({T1,T2,T3...,Tn})
FI(T1) FI(T2) FI(T3) FI(T4) FI(Tn)
(1)
CI({T1 X···XTn/2,Tn/2+1 X···XTn })
CI({T1,...,Tn/2})
CI({Tn/2+1 ,...,Tn})
FI(T1) FI(Tn/2) FI(Tn/2+1) FI(Tn)
(2)
CI({T1 X···XTn/2,Tn/2+1 X···XTn })
CI({T1 X T2 ,T3 X T4 })
CI({Tn-3 X Tn-2 ,Tn-1 X Tn })
CI({T1,T2})
CI({T3,T4})
CI({Tn-3 ,Tn-2 })
CI({Tn-1 ,Tn })
FI(T1) FI(T2) FI(T3) FI(T4) FI(Tn-3) FI(Tn-2) FI(Tn-1) FI(Tn)
(3)
CI({T1 X···XTn-1 ,Tn })
CI({T1 X···XTn-2 ,Tn-1 })
FI(Tn)
CI({T1 X T2 ,T3 })
CI({T1,T2})
FI(T3)
FI(T1) FI(T2)
(4)

Figure 3: Different mining trees.

6.5.1 Enumerating Expressions

Here, we discuss how to enumerate all expressions at the FI and CI levels, without considering which particular algorithm will be applied for a CI or FI term, and which particular order the join will be computed – if the first term in FI is a join of tables.

As in query optimization [25], the enumeration process is as follows. Begin with the top level expression (*i.e.*, $FI(T, I)$), where T is the table obtained by joining all tables in our schema. Given an expression, if any sub-expression matches a side of one of the equivalences listed in Section 5.3, the equivalence is applied and a new expression generated. This process is repeated until no new expressions can be generated.

For a schema with n tables, the number of FIs could be between 1 and n. One FI corresponds to our original expression: $FI(T, I)$. If there is only one FI, there will be no CI term since there is nothing to be merged. Therefore, there is only one expression with one FI term. There can be no more than n since there is one FI per table in the most decentralized computation. If there are n FIs, there are many ways in which the FIs could be merged.

Unless there is a Pairwise merge, the expression can be depicted as a tree as shown in Section 5.4, Figure 2 – which we refer to as the *mining tree*. We use the mining tree to show the different logical expressions – in the same way that a query tree maps to different logical query evaluation plans.[4] Each leaf of the mining tree corresponds to an FI expression, and each internal node to a CI expression that merges all of its children. In the case of n FIs, any tree containing n leaves could represent a different merging strategy at the logical level. For simplicity, in what follows we assume $n = 2^k$ for integer k. A few examples are listed here, and depicted in Figure 3.

[4]For simplicity, we do not consider Pairwise merges here.

1. n-way merge – two possible alternatives for merging (I/O or Memory saving).
2. two $n/2$-way merges and a final 2-way combination merge – with a total of $\binom{n}{n/2}$ possible alternatives.
3. $n/2 - 1$ 2-way merges forming a balanced binary tree – with a total of $n!$ possible alternatives.
4. a left-deep tree containing 2-way combination merges where a new FI is merged at each step – with a total of $n!$ possible alternatives.

We notice that the last example in the list resembles a left-deep join tree. In fact, there are as many logical plans containing n FIs and only 2-way merges as there are possible join orderings for joining n tables. To see that, consider each FI expression to be a table, and each CI to be a join operation. Each mining tree corresponds to a different merging strategy of the n FIs, in the same way as each query tree corresponds to a different join ordering of the n tables. The number of possible join orderings of n tables is $(2(n-1))!/(n-1)!$ (see Silberschatz *et al.* [25]). Considering that we could have 3-way merges, 4-way merges etc., the number of possible mining trees increases, and is considerably larger than the join ordering problem. As in the case of the join ordering problem, heuristics are best used (in a dynamic setting) to find good execution plans.

6.5.2 Obtaining Optimal Plan

Even though the number of join orderings (and therefore, possible 2-way merges) is very large even for small n (*e.g.*, for $n = 10$, the number exceeds 176 billion), we do not need to compute the cost for every possible plans. Since the total cost for an an expression e_i is a sum of each sub-expression of e_i, by minimizing the cost of each sub-expression, we find the minimal cost for evaluating a particular e_i. This "monotonicity" property may be explored in the same manner as is done in distributed or centralized query optimization by using dynamic programming. Each sub-expression corresponds to a sub-tree in the mining tree. For example, for $n = 6$ the root of the tree (*i.e.*, the last merge step) could be: $CI(\{T_1 \bowtie T_2 \bowtie T_3, T_4 \bowtie T_5 \bowtie T_6\})$. The number of ways to merge the tables T_1, T_2, and T_3 considering only 2-way merges is 12. The number of ways to merge T_4, T_5, and T_6 is also 12. However, instead of considering all possible 144 (*i.e.*, $12 * 12$) plans, we can limit ourselves to computing 24 (*i.e.*, 12+12) and then pick the most efficient sub-plan from each sub-tree. This is the same strategy as used in query processing.

7 Heuristic Optimization

To assess precise costs, we must estimate parameters such as table size, join selectivity factors, length of longest candidate itemset, number of frequent itemsets found etc. These parameters are similar to the ones required in query optimization, and some are easier to determine and maintain (*e.g.*, table size), whereas others may be assessed only by performing part of the DM itself (*e.g.*, the number of frequent itemsets). When using cost models, most database systems utilize heuristics to further reduce the number of choices for which cost estimates are computed. Likewise, we develop heuristics so that the number of choices to be examined can be reduced. Furthermore, in situations where cost estimates are not available, heuristics can help decide which plan to use for DM in order inefficient plans are avoided.

7.1 Heuristics for Decentralized Mining

We consider heuristic approaches to finding frequent itemsets in decentralized DM, which can find sub-optimal, but efficient, plans without requiring precise cost estimates.

Weight Table Choice

Computing the weight vector may requires a scan of the weight table. In this case, it is advantageous if the weight table were to have few attributes. By using Equiv 4, we can reduce the number of attributes from the weight table as in $FI(C,T) = FI(C,\pi_{cid}(T)) = FI(C,\pi_{cid}(B1 \cup B2))$. The project operation could incur an undesirable cost; however, since $FI(C,\pi_{cid}(B1 \cup B2)) = FI(C,(B1 \cup B2))$, we may compute the weight vector directly from $B1$ and $B2$. Furthermore, by reducing the weight table to one table (or horizontal partitions of one table), a check can be done to see if indices are available, which would further reduce costs.

Heuristic 1 *Use Equiv 4 to reduce the number of tables for the weight table.*

Table with only Key Attributes

Consider Equiv 1 to get: $FI(C \bowtie B1, I) = FI(C, C \bowtie B1) \cup FI(B1, C \bowtie B1) \cup CI(\{C, B1\}, C \bowtie B1)$. Table $B1$ has only key attributes, therefore the expression on the right is less expensive to evaluate. The reason is that the second and third terms on the right will be empty. Therefore, we only compare $FI(C \bowtie B1, I)$ with $FI(C, C \bowtie B1)$, and since it is likely that $C \bowtie B1$ has more records than C the scan for the left expression will be more expensive.

Heuristic 2 *When a table that contains only key attributes participates in a join comprising the first argument of an FI, convert the expression using Equiv 1.*

Cartesian Products

Consider $FI(C \bowtie P, B1) = FI(C, C \bowtie P \bowtie B1) \cup FI(P, C \bowtie P \bowtie B1) \cup CI(\{C, P\}, C \bowtie P \bowtie B1)$. Since C and P do not have common attributes and $B1$ contains foreign keys to both C and P, Equiv 4 and Equiv 5 allows $FI(C \bowtie P, B1) = FI(C, B1) \cup FI(P, B1) \cup CI(\{C, P\}, B1)$. This way, the counting may be separately processed – which would be cheaper than counting the itemsets on the Cartesian product.

Heuristic 3 *When the first argument of an FI has joins that result in a cross product, use Equiv 1 to convert the expression.*

Large FI Term

Consider a table *State* with a primary key $state_{id}$. Suppose table *Demographics* has another attribute, namely a foreign key to table *State*. Consider now, $FI(S \bowtie D \bowtie C, B1)$. Since the table $S \bowtie D \bowtie C$ is not a Cartesian product (even though S and C do not have any common attributes), the expression on the right is probably less expensive to realize, considering that there will be a lot of redundancy of the records of *State* in the joined table $S \bowtie D \bowtie C$. In this case, it is better to break the 3-table join by using Equiv 1. Notice that we could either arrive at:
(a) $FI(S \bowtie D, C \bowtie B1) \ \cup \ FI(C, B1) \ \cup \ CI(\{S \bowtie D, C\}, C \bowtie B1)$, or
(b) $FI(S, D \bowtie C \bowtie B1) \ \cup \ FI(D \bowtie C, B1) \ \cup \ CI(\{S, D \bowtie C\}, D \bowtie C \bowtie B1)$

Since the size of tables $S \bowtie D$ and C is likely to be smaller when compared to S and $D \bowtie C$, we choose expression (a).

Heuristic 4 *When the first argument of an FI contains at least 2 tables that are not directly related, use Equiv 1 to convert the expression, such that the tables that are not directly related are in different FI's, and the table that links the two is in the same FI as the smaller table.*

Merging CI Expressions

Consider $CI(\{D, C, P\}, B1) = CI(\{D, C \bowtie P\}, B1) \cup CI(\{C, P\}, B1)$. The expression on the right has two CI's. As such, the two scans of $B1$ on the right may be more expensive in terms of I/O. Two full scans of $B1$ are necessary because tables C and P do not have any direct relationship, and they are involved in both CI's. On the other hand, in some cases the expression on the right is preferable – *e.g.*, when the multidimensional array needed on the left (3 dimensions) is too large. Therefore, unless there is insufficient memory, the left expression is preferred.

Heuristic 5 *When tables do not have a direct relationship, merge multiple CI expressions using Equiv 2 when there is sufficient memory for the required multidimensional array required for evaluating the merged CI.*

Pre-merging Combinations

One exception to the previous heuristic is when one of the CI's involve tables that are directly related. For example, consider $CI(\{D, C, P\}, B1) = CI(\{D \bowtie C, P\}, B1) \cup CI(\{D, C\}, B1)$. Although $B1$ is the weight table for all the CI's, for the right most CI involving tables D and C, $B1$ does not need to be scanned. This is because tables D and C have a direct relationship, and therefore, we can use the weight vector for C and find the cross table itemsets for D and C without scanning $B1$ (as shown in Section 4, Section 4.4). In this case, we prefer the expression on the right, since we only have 2-way merges (less expensive than 3-way), and $B1$ is still scanned only once.

Heuristic 6 *When tables have a direct relationship, merge them before merging with other tables in the dataset.*

Pairwise Merging

Consider Equiv 3: $CI(\{D, C, P\}, B1) = CI(\{D, C\}, B1) \cup CI(\{D, P\}, B1) \cup CI(\{C, P\}, B1) \cup CI_3((\{D, C, P\}, B1)$. While the expression on the right contains multiple CI's, the only 3-way CI (the last term) is considerably cheaper then the 3-way CI on the left. As shown in Section 4, Section 4.3.2, the Pairwise technique not only reduces the number of candidates for the 3-way CI (since we only consider candidates involving attributes of all three tables), but also removes a considerable amount of false candidates, which is inherent in the I/O saving merge strategy. The larger the number of dimensions, the more false candidates are incurred. So, when tables are not related so that prejoining or combination is not a good option (above heuristic), we choose to process them pairwise.

Heuristic 7 *When there are three or more unrelated tables in a CI expression, convert the expression to a Pairwise merge using Equiv 3.*

Horizontal Distribution

For our example in Section 2.1, the use of Equivs 1, 2 and 4, provided (7) which is highly decentralized. On the other hand, we could also use (2), where we limit ourselves to using the algorithm for horizontal partitioning for the tables involved, and no further decentralization (*e.g.*, computing frequent itemsets locally at C) is possible. In this case, the weight table is horizontally partitioned (*i.e.*, $B = B1 \cup B2$), whereas the other tables are not.

A different situation occurs, where *Product* is horizontally partitioned (*i.e.*, $P = P1 \cup P2$), whereas the weight table is not. Here, by using Equivs 1, 2 and 4 we obtain $FI(D \bowtie C \bowtie B \bowtie (P1 \cup P2), I) = FI(D, C \bowtie B) \cup FI(C, B) \cup FI((P1 \cup P2), B) \cup CI(\{D, C, (P1 \cup P2)\}, B)$. In this case, only *Product* has horizontal distribution in the computation (*i.e.*, we would use the algorithm in [8] to compute $FI((P1 \cup P2), B)$); the counting for *Customer* and *Demographics*, and for all tables in Phase II, remain the same. Again, if we distribute the join over the union, we would no longer decentralize the computation, and the CPU and I/O costs would be the same as a centralized situation (except that the load would be shared).

Heuristic 8 *Avoid distribution of join over unions, when it prevents further decentralization.*

Use of Memsaving

All of the above heuristics deal with rewriting at the logical level. For any of the expressions, we can choose an I/O saving or Memory saving strategy. Because the I/O saving scans the large table only once, we should expect that it would perform better than the Memory saving approach. As pointed out in [18], the Memory saving approach is always cheaper than the centralized approach, while the I/O saving approach is not guaranteed to be better – especially under situations that cause many false candidates, such as merging many tables in one step. So, while the I/O saving can provide better savings in certain circumstances, the Memory saving is a safe bet when we do not know the percentage of false candidates.

Heuristic 9 *When no estimates are available, and there are more than 3 primary tables, use n-way Memory saving merge, instead of I/O saving.*

7.2 Developing a Heuristic Plan

The above Heuristics deal mainly with options at the logical plan. Exceptions are Heuristics 1 and 9. One way of selecting the best heuristic strategy is to enumerate all possible plans (as discussed in Section 6.5.1), and to use our heuristic rules above to choose in the space of plans. We showed in Section 6.5.1 how the number of possible plans is large, and therefore, the complete enumeration should be avoided. Instead, we indicate a way to develop a plan based on our heuristic rules such that the total time to create the plan is on the order of the number of tables. Our algorithm constructs a *mining tree* that efficiently merges only related tables. We make the simplifying assumption that each primary key is a foreign key in only one table.[5]

First, we create a tree, called *schema tree*, with n nodes, where each node corresponds to one table, and $n-1$ edges, where each edge corresponds to a foreign key relationship. Usually, one of these nodes corresponds to a table whose primary key is not a foreign key in any table (*e.g.*, the fact table in a Star schema); let this node be the root node.[6] We associate two labels to each node of the schema tree:

- *table*: the table in the schema that the node represents, and
- *expression*: the join expression that joins all tables in the sub-tree.

We associate two labels to each node of the mining tree:

- *expression*: the algebra expression generated, and
- *link*: the sub-tree in the schema tree which corresponds to the expression.

The mining tree is initialized with a root node with expression $FI(T, I)$s – *i.e.*, the FI expression that applies to the join of all tables in the database, and link referencing the root node of the schema tree. Then, applying Equivalences 1 and 2, we decentralize the computation by breaking the FI expression into smaller FIs and one CI (the CI merges all the sub-expression in the FIs). We use the information in the schema tree to decentralize the FI expression such that each sub-tree in the schema tree corresponds to one FI. Then, we recursively decentralize each of the FIs generated; the recursive function **Decentralize** is depicted in Figure 7.2.

[5]If a primary key for table T_i is a foreign key to more than one table, say $T_{j_1}, \ldots T_{j_n}$, and if more than one T_{j_k} is in the natural join, there will be more than one instance of T_i in the join – one instance for each T_{j_k} in the join. Our algorithm would treat each instance of T_i as a separate table.

[6]When such a table for the root node is not available, we consider the table to be the one where the number of records corresponds to the number of records in the final joined table T. See Section 4.5 for details on schema with such characteristics.

```
**Decentralize** (node $t$)

01) if $t.link$ is null
02)     return;
03) if $t.link$ is a leaf node
04)     return;

05) create $c$ a child of $t$;
06) $c.expression = FI(t.link.table)$;
07) $c.link = null$;
08) $t.expression = CI(t.link.table)$;

09) for each $i$ child of $t.link$ do
10)     create $d$ child of $t$;
11)     $d.link = i$;
12)     $d.expression = FI(i.expression)$;
13)     $d.table = i.table$;
14)     add $d.expression$ to argument of $CI$ in $t.expression$;
15) endfor

16) for each $i$ child of $t$ do
17)     Decentralize(i);
18) endfor

19) return;
```

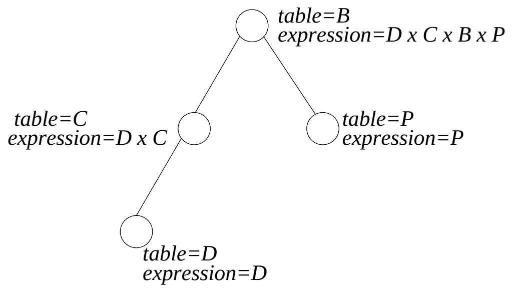
Figure 4: Decentralizing computation in the *mining tree*.

The above algorithm creates a decentralized plan with n FIs: one for each table (or the union of horizontal partitions of a table). This way, distributing the joins over unions (Heuristic 8), and a large FI term (Heuristics 4) are avoided. Because no tables are prejoined, Cartesian products (Heuristic 3), and joins containing a table with only key attributes (Heuristic 2) are also avoided. Only tables that are directly related by foreign keys are prejoined (Heuristic 6). The *mining tree* can be further pruned by removing FI terms in which the first argument is a table with only key attributes, since such FIs evaluate to the empty set; also, such tables should be removed from any CI terms.

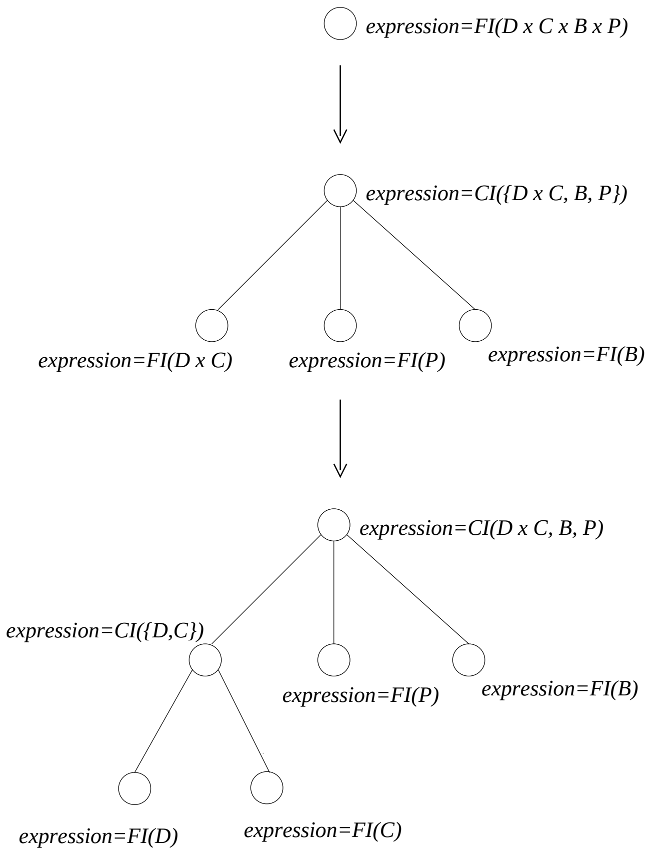
The next step is to consider particular implementations at the physical level. As mentioned previously, the Pairwise merge of n tables can be seen as the same logical plan of an n-way merge, except that extra pruning is done. By using Heuristic 7, we convert all of the CI terms with more than 2 tables to Pairwise merges. The other particulars, such as computations of weight vectors (using Heuristic 1) and choice of I/O saving *vs.* Memory saving (using Heuristic 9) can be relegated to the phase of mapping the *mining tree* to particular algorithms. Furthermore, if cost estimates are available, a given sub-tree could be substituted by an FI if the join of the leaves in this sub-tree was pre-computed and materialized, if a centralized approach to this sub-tree was less expensive than the decentralized approach.

7.3 Heuristics Applied

Figure 5(a) shows the schema tree, and Figure 5(b) shows the construction of the mining tree by applying our heuristic algorithm to our example from Section 2.1.



(a)



(b)

Figure 5: Creation of the mining tree.

Since $ItemsBought$ contains only key attributes, we prune the mining tree from Figure 5, and arrive at the final mining tree depicted in Figure 6(a). Notice that we arrive at expression (10) since our transformation algorithm does not consider any prejoins. As mentioned previously, if a cost analysis suggests that the prejoin is more efficient, the sub-tree rooted by $CI(\{D \bowtie C\})$ could be changed to one node, $FI(D \bowtie C)$, as shown in Figure 6(b).

Our heuristics deal mainly with approaches for reducing I/O (except for Heuristic 9). Heuristic 8 also deals with communication costs, since by distributing joins over unions, an increase in synchronization needs (*e.g.*, after every iteration) is likely to occur, and therefore, communication costs increase. Notice that our heuristics reduce the size of the scanned tables (both in the number of records and columns), therefore reducing CPU costs associated with the subset operation (which depends on the number of columns). Therefore, although heuristics could be developed to deal with CPU cost reduction specifically, our heuristics are already helpful in this regard.

8 Empirical Validation

Our primary goal is to provide algorithms that run against decentralized data, thereby avoiding the sometimes infeasible (or expensive) computation of joins and their mining thereafter. Our intent is *not* in merely improving the efficiency of existing algorithms. Nevertheless, it is desirable for our decentralized approach to be no worse than the traditional approach to the frequent itemset counting. Benefits from our approach

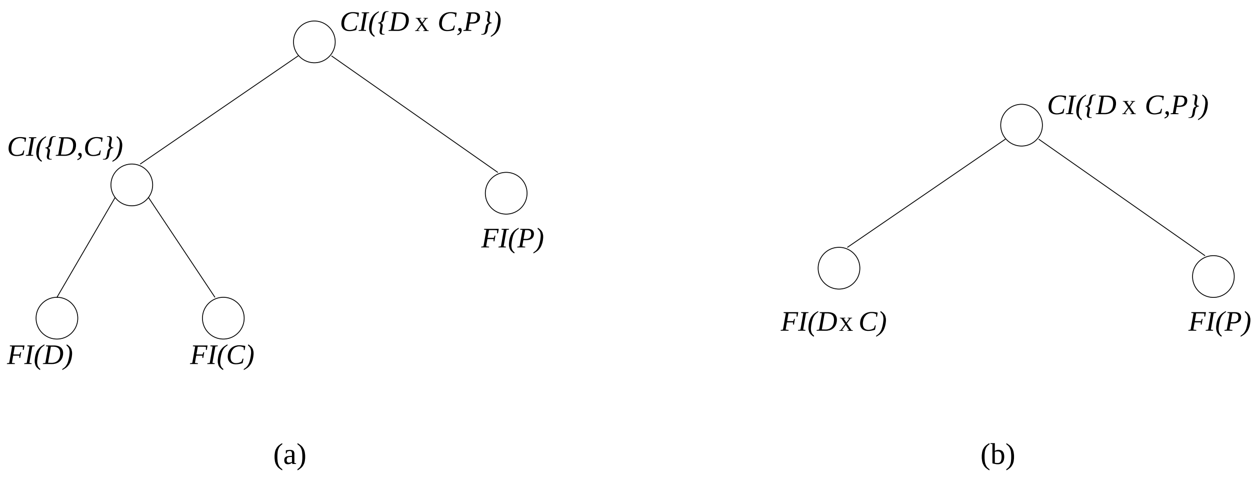


Figure 6: Final mining trees.

include not only time savings, but also in cases where applying the centralized approach is infeasible (*e.g.*, when there is insufficient space available to store the joined tables). Since the choices for decentralized mining are numerous as explained in Section 5, we also assess our heuristic approaches to choosing better performing plans.

In this section, we evaluate our decentralized approach as compared to a typical centralized approach to finding frequent itemsets. Among the centralized algorithms developed for frequent itemset counting, the Apriori algorithm with its variations have been the most successful and widely used. We chose to compare empirically our decentralized approach with the typical centralized Apriori approach. We recognize that other centralized algorithms have been suggested, and in certain situations, they outperform the Apriori. Therefore, we include a discussion on how several other centralized algorithms compare with our decentralized approach.

8.1 Experimental Setup

Association rules algorithms for categorical and numerical data (*e.g.*, [26, 6]) have been evaluated using data such as census data. This data is centralized in one table since most of the algorithms for counting frequent itemsets deal only with one central table, and therefore is not suited for testing our approach. Instead, we based our dataset on the TPC-D benchmark [30] which reflects the decentralized schemas typical in a data warehouse. The schema in the TPC-D benchmark[7] which is is depicted in Figure 7. A line connecting two tables indicates a foreign key relationship, with the arrow indicating the direction of the many to one relationship.

To incorporate customer buying patterns we discarded the *LineItem* table provided in the benchmark, and replaced it with the transactions generated by the IBM Almaden synthetic generator [5]. This generator produces transactions that mimic those in the retailing environment, and assumes that people tend to buy sets of items together. Transactions sizes, as well as length of frequent itemsets, are clustered around a mean – with a few transactions and frequent itemsets having a larger number of items. We performed several experiments, and we describe the generation of the relationship table in each of the different scenarios we considered.

[7]Tables *Nation* and *Region* are considerably smaller than the other tables, so we only considered table *Nation* and left out *Region*.

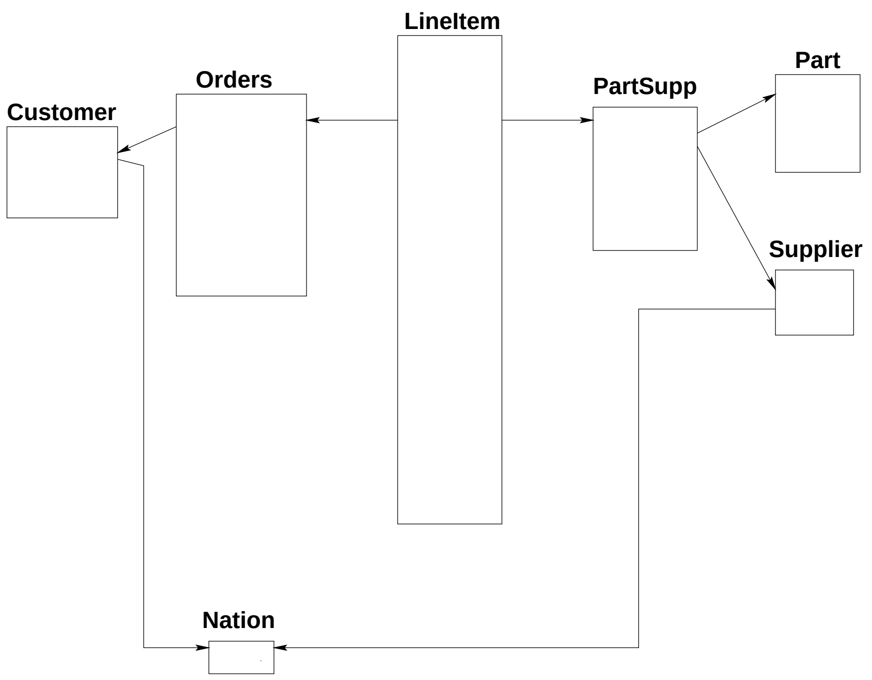


Figure 7: TPC-D benchmark schema.

Also, consider the data generated for the non-key attributes in the TPC-D Benchmark. For the table *Customer*, for example, the original attributes[8] were:

- *customer id*: primary key of table *Customer*
- *customer name*: string containing `Customer` and the customer id.
- *address*: string with 10 to 40 characters.
- *nation key*: foreign key to table Nation.
- *phone*: 10 digits number.
- *account balance*: real number.
- *market segment*: categorical value: 1, 2, 3, 4, or 5.
- *comment*: string with at most 59 words.

To generate more attributes on which to perform the mining, we created new attributes based on those generated from the TPC-D Benchmark. As an example, the attribute *comment* from the *Customer* table was divided into six attributes. Below, we describe how we generated a few of these attributes, and how they corresponded to the attributes of the *Customer* table.

- *education level*: Given by the ASCII value (modulo 5) of the first character in the comment field. As a result, the field *education level* has five possible values.
- *age*: Given by 15 added to the number of words in the comment field to obtain customers' age range from 15 to 74.

[8]For details on how each of these attributes were generated, refer to the TPC-D Benchmark description document [30].

- *gender*: Given by the ASCII value (modulo 10) of the last character in the comment field; the value is split into two categories: 0..3 corresponding to male, and 5..9 corresponding to female.

To run the frequent itemsets counting algorithms, the attributes were discretized (*i.e.*, transformed into items). For example, the *gender* attribute of the *Customer* table corresponded to 2 items: one representing "male", and the other representing "female". Quantitative attributes, such as *total price* in table Orders, were less trivial to discretize. Note, when dealing with real data, such discretization is done as a first pass of the algorithm (*e.g.*, as described in [26]) without incurring much extra time. Since the discretization of attributes is not the focus of our work, we pre-discretized all attributes to have the data in an *item* format when running our tests.

In Figure 8 we describe the characteristics of the tables after the generation and discretization of all the necessary attributes. Henceforth, references to tables in the TPC-D Benchmark in this document imply these final tables.

Table	key attrs	non-key attrs	items	records	size (in Mb)
Customer	2	13	61	150,000	9,000
Nation	2	6	22	25	0.8
Orders	2	13	51	1,500,000	90,000
Part	1	20	94	200,000	16,800
Partsupp	3	8	32	800,000	35,200
Supplier	2	12	56	10,000	560

Figure 8: Characteristics of resulting tables.

Our experiments were performed using a 200 MHz Pentium Pro machine with 256 Mbytes of RAM, running Linux. We implemented the basic I/O saving and Memory saving approaches, and many of the variations described in Section 4. We ran extensive evaluations, among which we selected a few representative experiments. The experiments are described in two sections. The first set of experiments show the feasibility of our decentralized algorithms, and the runtime improvement achieved. The second set of experiments evaluates our general approach of enumeration, and the selection of best strategies. Thereafter, we discuss comparisons of our decentralized techniques to mining algorithms other than the Apriori.

8.2 Decentralized Algorithms

We describe here the results of the tests performed on three different schemas. The schemas are in increasing order of complexity. We also include scalability results for increasing the number of records in the relationship table.

8.2.1 Two Primary Tables

The data schema (depicted in Figure 9) used in this experiment was composed of three tables:

- *Customer*: "Customer" table from the TPC-D Benchmark.
- *Product*: "Part" table from the TPC-D Benchmark.
- *ItemsBought*: Table containing two attributes: *c_id* (foreign key to *Customer* table) and *p_id* (foreign key to *Product* table).

The *ItemsBought* table was generated using the Almaden synthetic generator with the following parameters:

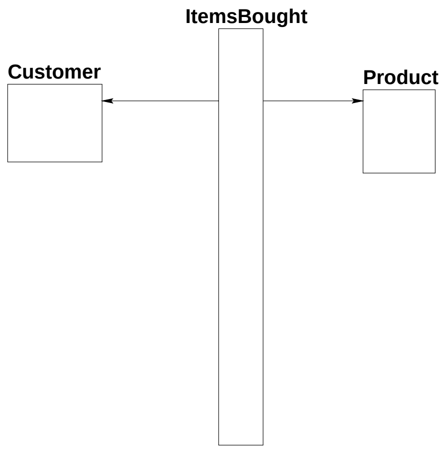


Figure 9: Two primary tables schema.

- number of transactions: 150,000 (corresponding to 150,000 customers)
- transaction length: 40 (each customer purchases, on average, 40 products)
- number of items: 200,000 (corresponding to 200,000 products)

The sizes of the tables *Customer* and *Product* were as shown in Figure 8. The table *ItemsBought* contained 5,949,147 records and was approximately 40 Mbytes in size. The table obtained by joining all three tables, therefore, also had 5,949,147 records with 33 non-key attributes, and was approximately 833 Mbytes in size.

We now use our algebra presented in Section 5 to enumerate different processing alternatives. For ease of exposition, henceforth we use C to refer to table *Customer*, P to refer to table *Product*, and B to refer to table *ItemsBought*. We omit the project notation for the second argument of FI and CI. Our goal is to find the frequent itemsets in the joined table $T = C \bowtie B \bowtie P$.

Processing alternatives at the logical level:

1. $FI(T, I)$
2. $FI(C, B) \cup FI(P, B) \cup CI(\{C, P\}, B)$

For alternative (1) there is only one option at the physical level: compute the joined table T and run Apriori on T. For alternative (2), we first computed the weight vectors and ran modified Apriori on separate tables *Customer* and *Product*. However, there are many ways in which we could implement the merge physically (see Section 4 for details regarding different merging plans).

For I/O saving with support value 20%, there was sufficient memory to read in both primary tables, and to pre-compute all frequent itemsets per record (FIPR) for each primary table. Furthermore, since *ItemsBought* is ordered by *c_id*, it was possible to read in the *Customer* table in blocks, thereby saving memory without significant overhead in the computation of the FIPR for table *Customer*. Given that we wanted to experiment with situations where there is insufficient memory to pre-compute all FIPRs, or enough disk space to save the pre-computed FIPRs, we simulated different scenarios. We tested the following implementations for I/O saving:

- io: *Customer* read in blocks; *Product* FIPRs stored in memory.
- io-n: *Customer* FIPRs and *Product* FIPRs stored in memory.
- io-dC: *Customer* read on demand; *Product* FIPRs stored in memory.
- io-dP: *Customer* read in blocks; *Product* read on demand (no pre-computation of FIPRs).
- io-dCP: both tables read on demand (no pre-computation of FIPRs).
- io-fC: *Customer* FIPR computed and read on demand; *Product* FIPRs stored in memory.
- io-fP: *Customer* read in blocks; *Product* FIPRs computed and read on demand.
- io-fCP: both tables' FIPRs computed and read on demand.

We tested the same parameters for Memory Saving, resulting in the following tests: mem, mem-n, mem-dC, mem-dP, mem-dCP (note that there are no FIPR in the Memory Saving merge). Furthermore, for Memory Saving, we also considered the situation where the table T is prejoined: mem-j.

In Figure 10, we show the runtime for support level 20%. In the graph, we show separate shades for the various stages. "Join" is the time taken to pre-compute the join (needed for approach mem-j and apriori). "Apriori" is the time taken to run the Apriori algorithm on the joined table. "Phase I" is the time to compute the weight vectors and run the modified Apriori on tables *Customer* and *Product*. "Phase II" corresponds to the time taken for the I/O or Memory saving algorithms.

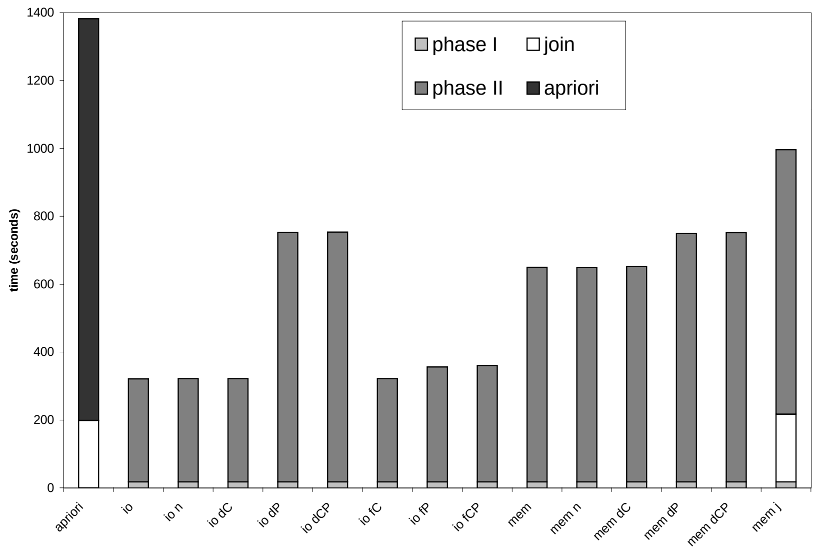


Figure 10: Time performance for $FI(C \bowtie B \bowtie P, I)$ for support level 20%.

While all decentralized strategies outperformed the centralized approach, the best results were achieved by the I/O saving strategy. Among the various implementations for the I/O saving strategy, those with FIPRs computed multiple times for records of table *Product* (io-dP and io-dCP) presented the worst performance. While storing pre-computed FIPRs in memory were the best strategies (io, io-n, and io-dC), storing them on disk and reading on demand (io-fC, io-fP, and io-fCP) did incur a significant penalty in performance. Similarly, reading records on demand did not have a large effect on the performance of the Memory saving strategy.

We now examine results for different support values: 30, 20, 15, 10%. Some costs remain the same regardless of the support: computation of the weight vectors and materialization of the join. In Figure 11, we compare the runtime for selected scenarios: io, mem, mem-j and apriori. The plot shows the runtime for each scenario divided by the runtime taken by the Apriori for the same support value. The scenarios we plot are: io, mem, mem-j and apriori.

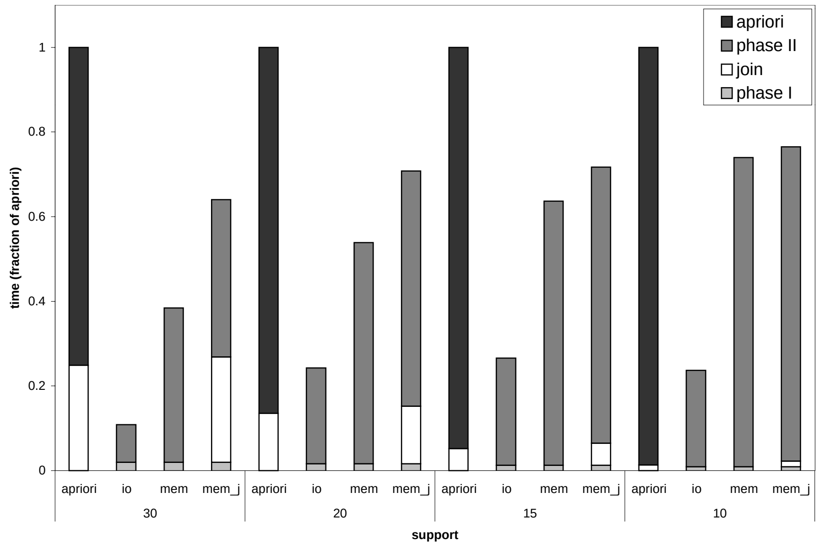


Figure 11: Time performance for $FI(C \bowtie B \bowtie P, I)$ for different support levels.

The times for Phase I and the computation of the join decreased as a percentage of the total time for lower support values. While the relative performance of the Memory saving approach is worse for lower support values, the I/O saving strategy continued to run at approximately 20% of the time taken by the centralized approach. The best relative improvement was achieved for support 30%, with the I/O saving running an order of magnitude faster than the centralized approach. The reason is the small average size of the FIPRs, since there were fewer frequent itemsets found in Phase I, resulting in fewer false candidates. For the Memory saving, lower support values resulted in a larger fraction of frequent itemsets that span across tables, which decreased the relative savings.

8.2.2 Star Schema

The data schema (depicted in Figure 12) used in this experiment was composed of four tables:

- *Customer*: "Customer" table from the TPC-D Benchmark.
- *Product*: "Part" table from the TPC-D Benchmark.
- *Store*: "Supplier" table from the TPC-D Benchmark.

- *ItemsBought*: Table containing three attributes: *c_id* (foreign key to *Customer* table), *p_id* (foreign key to *Product* table), and *s_id* (foreign key to *Store* table).

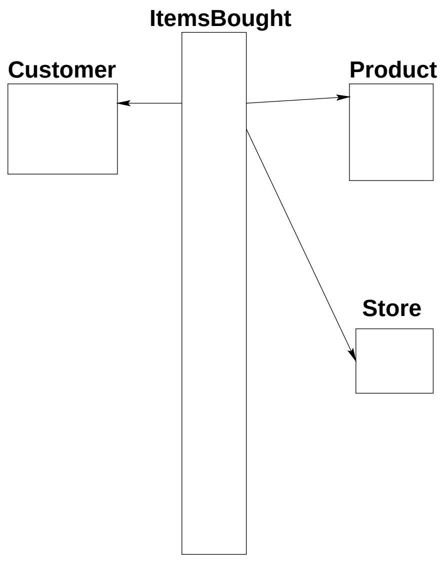


Figure 12: Star schema.

The first two attributes of the table *ItemsBought* were generated using the Almaden synthetic generator with the same parameters used in Section 8.2.1. An *s_id* was randomly assigned to each (c_id, p_id) pair. The table obtained by joining all three tables had 5,949,147 records, 45 non-key attributes, and was approximately 1.1 Gbytes in size.

Again, we use our algebra presented in Section 5 to enumerate different processing alternatives. For ease of exposition, we use S, C, P, and B to refer to tables *Store*, *Customer*, *Product* and *ItemsBought*. Our goal is to find the frequent itemsets in the joined table $T = S \bowtie C \bowtie B \bowtie P$.

Here, we considered the Apriori, 3-way merge, and Pairwise merge strategies:

1. $FI(T, I)$
2. $FI(S, B) \cup FI(C, B) \cup FI(P, B) \cup CI(\{S, C, P\}, B)$
3. $FI(S, B) \cup FI(C, B) \cup FI(P, B) \cup CI(\{S, C\}, B) \cup CI(\{S, P\}, B) \cup CI(\{C, P\}, B) \cup CI_3(\{S, C, P\}, B)$

In Figure 13 we show the runtime for each of the strategies for support value 30%. The time shown in the graph for each plan is as follows:

1. *Apriori.* The total time is divided into: computing join $T = S \bowtie C \bowtie B \bowtie P$, and Apriori (running original centralized algorithm on T).
2. *3-way merge.* The total time is divided into: Phase I (computing the weight vectors and running modified Apriori on C, P, and S), and Phase II (merging step). We considered the following merge strategies: I/O saving, Memory saving, and Memory saving on the joined table T.

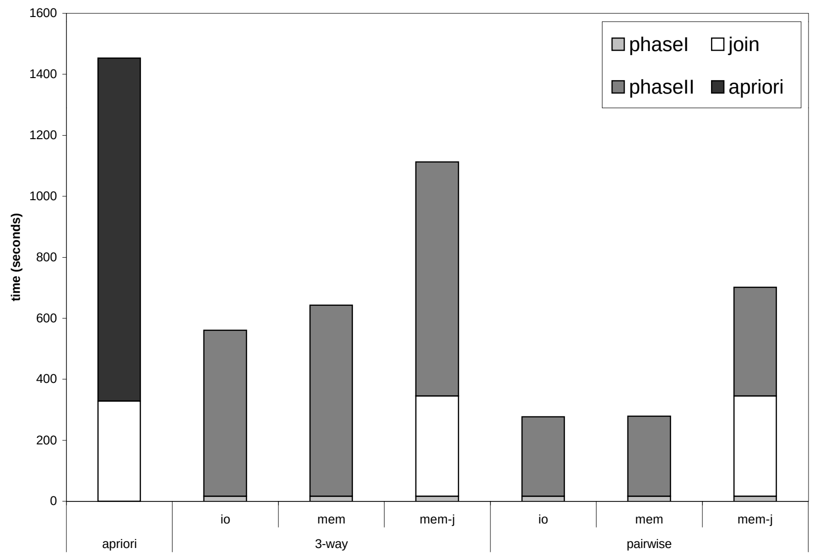


Figure 13: Time performance for Star schema for support level 30%.

3. *Pairwise merge.* The total time is divided into: Phase I (computing the weight vectors and running modified Apriori on C, P, and S), and Phase II (merging step). Phase II in this case is divided into four parts: three 2-way merges: one for each pair of tables, and one 3-way merge of the results. We considered only the I/O saving strategy for the 2 table merges (since by the previous example, I/O saving works better for two tables), and all three of the merge strategies for the 3 table merge.

All decentralized strategies outperform the centralized approach. In fact, the Pairwise I/O and Memory saving strategies perform in less time than the computation and materialization of the join that precedes the centralized Apriori approach.

In Figure 14, we show the runtime for support value 20%. Most decentralized strategies continue to outperform the centralized approach. The only decentralized strategy that is worse is the 3-way I/O saving merge. The reason is, for this support value, the number of FIPRs is high resulting in an expensive counting step (when we count the Cartesian product of FIPRs). Clearly, the aim would be to avoid choosing such an approach to effect the mining, and accurate data statistics together with our heuristic approach would help do so (see Section 8.3).

In Figure 15, we show the time taken by each strategy (except 3-way I/O saving) relative to the time taken by the centralized Apriori approach for support values 30, 20, and 15%. As the support value, decreases, the length of the FIPRs increases, resulting in a more expensive counting step. The size of the FIPRs affect the relative performance of any I/O saving merge (Pairwise or 3-way). We also notice that, for this data set, like in the previous section, lower support values increase the percentage of cross-table itemsets, which decreases the relative performance of the 3-way Memory saving approach. For the Pairwise Memory saving, reducing the support does not negatively affect the relative performance of the Pairwise Memory saving since the percentage of itemsets across all three tables does not significantly increase.

Finally, we point out the main difference between then Memory saving and Memory saving with the table joined up front: the former composes a join "on the fly". While Memory saving is an acceptable strategy for this data set, it might not be for complex, expensive joins. However, even for complex, expensive joins,

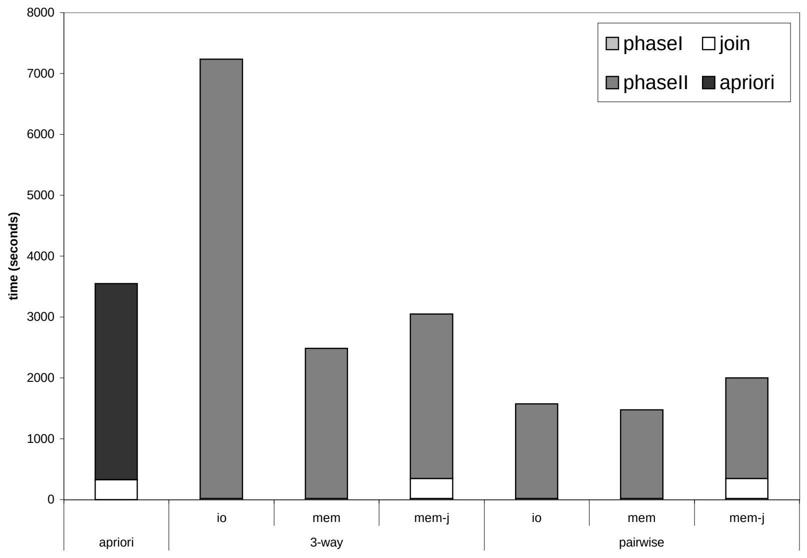


Figure 14: Time performance for Star schema for support level 20%.

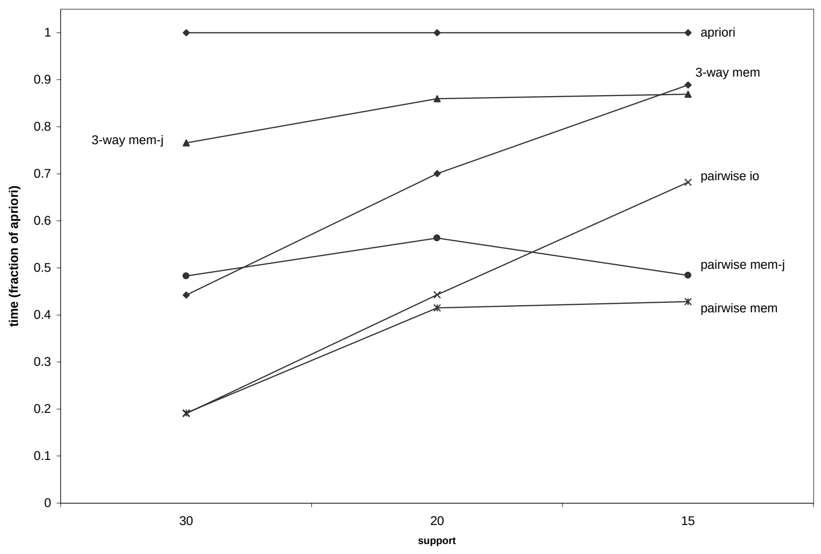


Figure 15: Time performance for Star schema for different support levels.

we expect the relative performance of the I/O saving and Memory saving joined to be similar to the one exhibited for this data set.

8.2.3 Snow Flake Schema

The data schema (depicted in Figure 16) used in this experiment was composed of four tables:

- *Customer*: "Customer" table from the TPC-D Benchmark, with an added attribute: *s_id* (foreign key to table *Store*).
- *Product*: "Part" table from the TPC-D Benchmark.
- *Store*: "Supplier" table from the TPC-Benchmark.
- *ItemsBought*: Table containing two attributes: *c_id* (foreign key to *Customer* table) and *p_id* (foreign key to *Product* table).

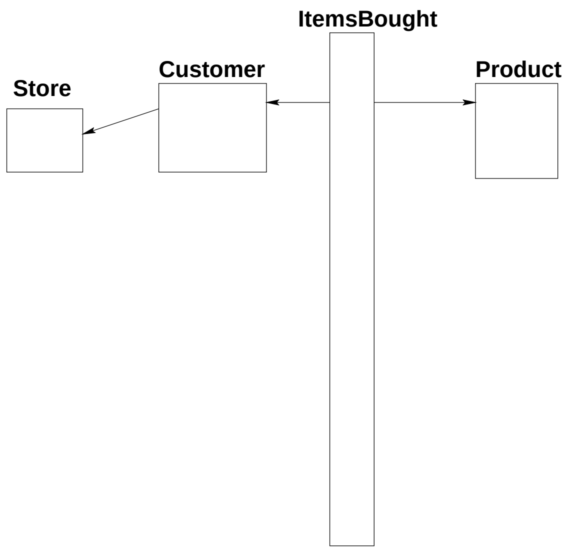


Figure 16: Snow Flake schema.

The table *ItemsBought* is the same as used in Section 8.2.1. We assumed that all products are sold in every store, and that each customer shops at one store only. A store was randomly assigned to each customer (given by the attribute *s_id*). The table obtained by joining all three tables had 5,949,147 records, 45 non-key attributes, and was approximately 1.1 Gbytes in size.

The main difference between this data schema and the previous one in Section 8.2.2, is the direct relationship between the primary tables *Customer* and *Store*. In such a scenario, prejoining some primary tables in a decentralized approach might be a good idea. This schema is similar to the one given in Section 2.1. The possible decentralized alternatives were listed in Section 5.4, the only differences are the table *Store* instead of table *Demographics*, and the table *ItemsBought* is not horizontally partitioned. Therefore, we have at least nine different alternatives at the logical level (we do not consider the one where we distributed the join over the unions, since *ItemsBought* is not horizontally partitioned).

Processing alternatives at the logical level we examined were:

1. $FI(T, I)$
2. $FI(S, C \bowtie B) \; \cup \; FI(C, B) \; \cup \; FI(P, B) \; \cup \; CI(\{S, C, P\}, B)$
3. $FI(S, B) \cup FI(C, B) \cup FI(P, B) \cup CI(\{S, C\}, B) \cup \; CI(\{S, P\}, B) \cup CI(\{C, P\}, B) \cup CI_3(\{S, C, P\}, B)$
4. $FI(S \bowtie C, B) \; \cup \; FI(P, B) \; \cup \; CI(\{S \bowtie C, P\}, B)$
5. $FI(S, C \bowtie B) \; \cup \; FI(C, B) \; \cup \; CI(\{S, C\}, B) \; \cup \; FI(P, B) \; \cup \; CI(\{S \bowtie C, P\}, B)$

The only prejoin that we consider, plan (4), is the prejoin of tables *Customer* and *Store* since the tables are directly related. Also, the only combination (combined merge strategy), plan (5), combines *Customer* and *Store*, again for the same reason. Prejoins of tables that are not directly related is discussed in the next section. When computing $CI(\{S, C\}, B)$ in plan (5) we make use of the direct relationship between tables *Store* and *Customer*, and use the weight vector computed for *Customer*.

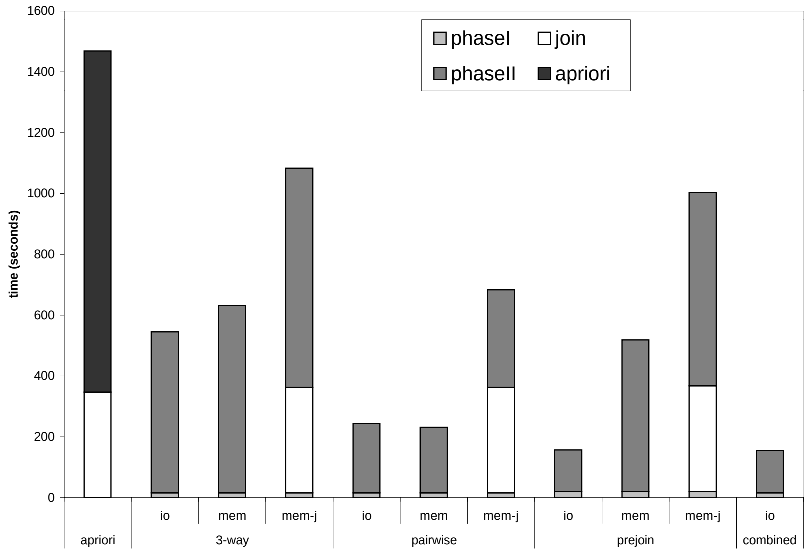


Figure 17: Time performance for Snow Flake schema for support level 30%.

In Figures 17 and 18 we show the runtime for each of the strategies for support value 30 and 20%, respectively. We refer to each one of the plans as: (1) Apriori; (2) 3-way merge; (3) Pairwise merge; (4) Prejoin merge; and (5) Combined merge. The time shown in the graphs for the first three plans is in the same format as for the previous section. The time for the other two plans is as follows:

4. *Prejoin merge.* The total time is divided into: computing join $S \bowtie C$, Phase I (computing the weight vectors and running modified Apriori on $S \bowtie C$ and P), and Phase II (merging step). Again, we consider all three merge strategies.

5. *Combined merge.* The total time is divided into: Phase I (computing the weight vectors and running modified Apriori on C, P, and S), and Phase II (merging step). We consider only the I/O saving strategy (since Memsaving is basically the same as in plan (4)).

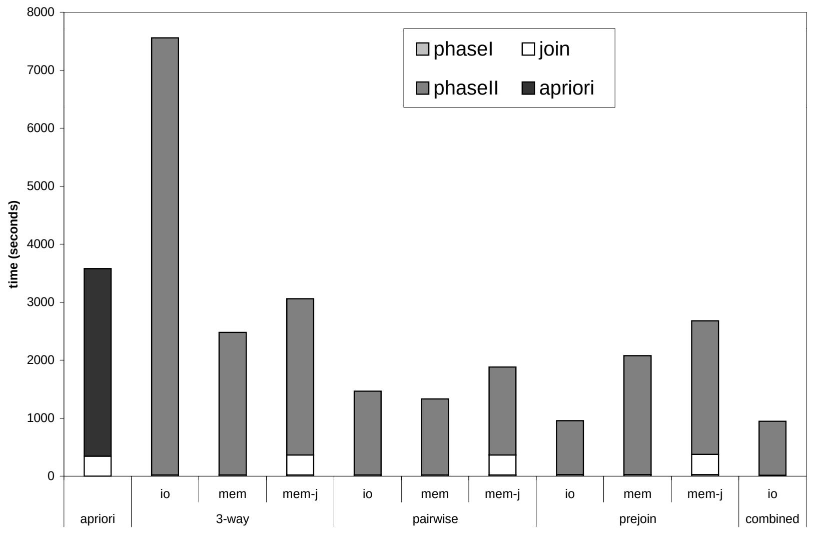


Figure 18: Time performance for Snow Flake schema for support level 20%.

As shown in both graphs, the best times are achieved by either prejoining or combining tables *Customer* and *Store*. As in the star schema, the 3-way I/O saving merge does not perform well when the size of FIPRs increase.

Relative results of selected scenarios with different support values is shown in Figure 19. The relative performance of the two best strategies, Prejoined and Combined I/O saving (overlapping in the graph), scales well with the increase in size of the FIPRs (*i.e.*, with lowering of support values). In fact, it presents a similar scale-up as the case of two primary tables (see Figure 11), which was expected since either prejoining or combining two tables up-front results in a final 2-way merge, as opposed to a 3-way merge.

8.2.4 Results for Scalability with Database Sizes

To determine how the decentralized strategies scale with the size of the database, we ran tests for the schemas in Section 8.2.1 and Section 8.2.2 for different sizes of *ItemsBought*. Figures 20 and 21 show the time taken by the I/O saving, Memory saving, and centralized Apriori strategies for the schema with two primary tables (Section 8.2.1) for support levels of 30% and 20%, respectively. The Apriori algorithm was already known to scale linearly with the table size [4]. The graphs show that our decentralized algorithms also scale linearly, and as indicated by the slopes in the graphs, our decentralized algorithms scale better.

Figures 22 and 23 show the time taken by the 3-way I/O saving merge, 3-way Memory saving merge, Pairwise I/O saving merge, and centralized Apriori strategies for the schema with 3 primary tables (Section 8.2.2) for support levels of 30% and 20%, respectively. Again, our graphs show that our decentralized algorithms scale linearly with the number of records. For support value 20%, the I/O saving algorithm takes more time than the Apriori algorithm. This was expected since we observed that, for higher dimensions (in this case, three), the algorithm is sensitive to the number of candidates, which is inversely related to minimum support.

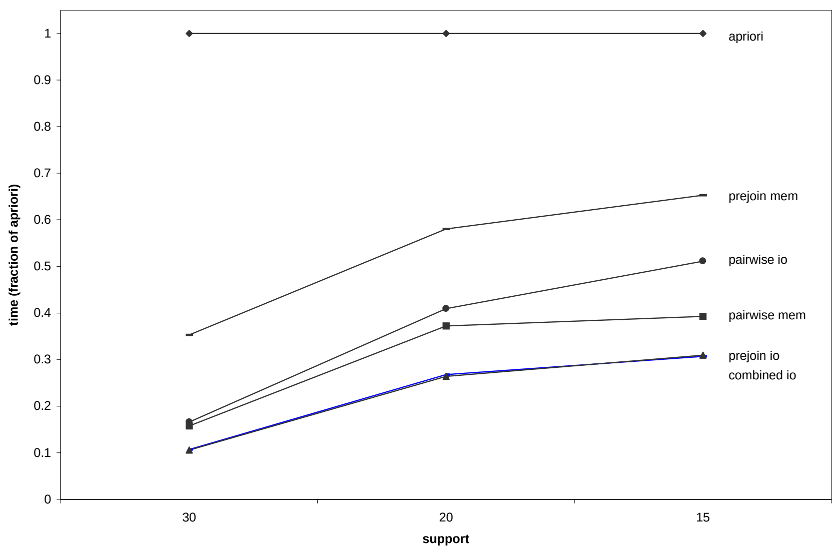


Figure 19: Time performance for Snow Flake schema for different support levels.

8.3 Choosing among Alternatives

In this section, we evaluate our method of enumerating the possibilities and choosing the best alternative. We look at two examples: the Customer-Product-Store schema discussed in Sections 8.2.2 and 8.2.3, and the schema based on the TPC-D benchmark shown in Figure 7.

8.3.1 Star and Snow Flake Schemas

For both the Star and Snow Flake schemas given in Sections 8.2.2 and 8.2.3, respectively, the centralized strategy was $FI(T, I)$, where $T = S \bowtie C \bowtie B \bowtie P$. When developing decentralized plans in this section, we refer to the Star schema, but note that the same plans also apply to the Snow Flake schema.

In Section 8.2.2, we used three different plans for the Star schema given: centralized Apriori, decentralized 3-way merging, and decentralized Pairwise merging. As discussed in Section 5, by applying the re-write rules to the original expression $FI(T, I)$, we could arrive at several different decentralized expressions (more than the those given in Section 8.2.2). We could have 1, 2, or 3 FIs in our expression, corresponding to Apriori, Prejoined or Combined merge, and 3-way merge. Therefore, the total number of possibilities, at the logical level is at least:

$$1 + 2 \times \left(\begin{array}{c} 3 \\ 2 \end{array} \right) + \left(\begin{array}{c} 3 \\ 3 \end{array} \right) = 8$$

Furthermore, using equivalence 3 from Section 5, we also have the possibility of a Pairwise merge. At the logical level nine processing alternatives we arrive at are:

1. $FI(T, I)$
2. $FI(S, B) \cup FI(C, B) \cup FI(P, B) \cup CI(\{S, C, P\}, B)$
3. $FI(S, B) \cup FI(C, B) \cup FI(P, B) \cup CI(\{S, C\}, B) \cup CI(\{S, P\}, B) \cup CI(\{C, P\}, B) \cup CI_3(\{S, C, P\}, B)$
4. $FI(S \bowtie C, B) \cup FI(P, B) \cup CI(\{S \bowtie C, P\}, B)$

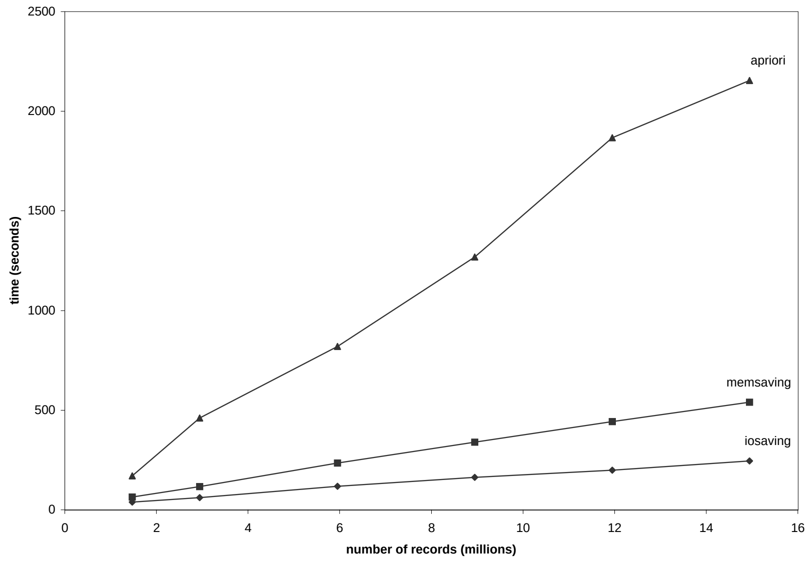


Figure 20: Scalability experiment using two primary tables schema (of Figure 9), and with support 30%.

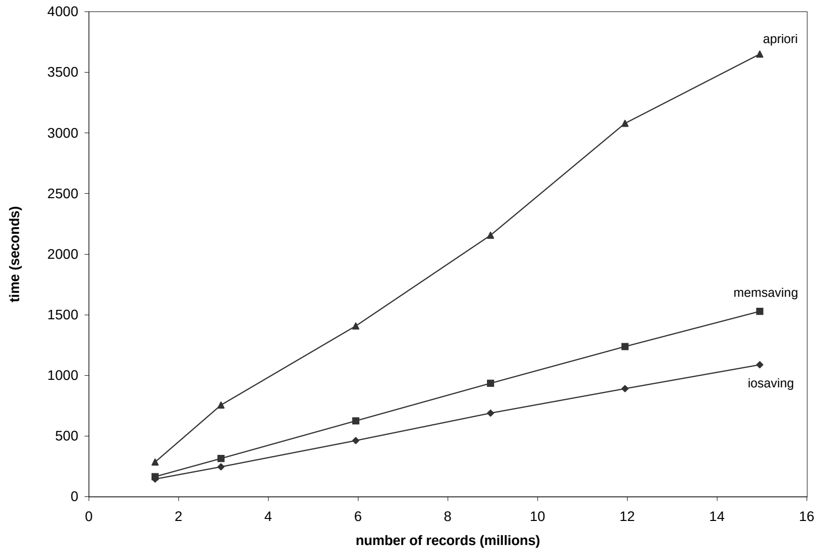


Figure 21: Scalability experiment using two primary tables schema (of Figure 9), and with support 20%.

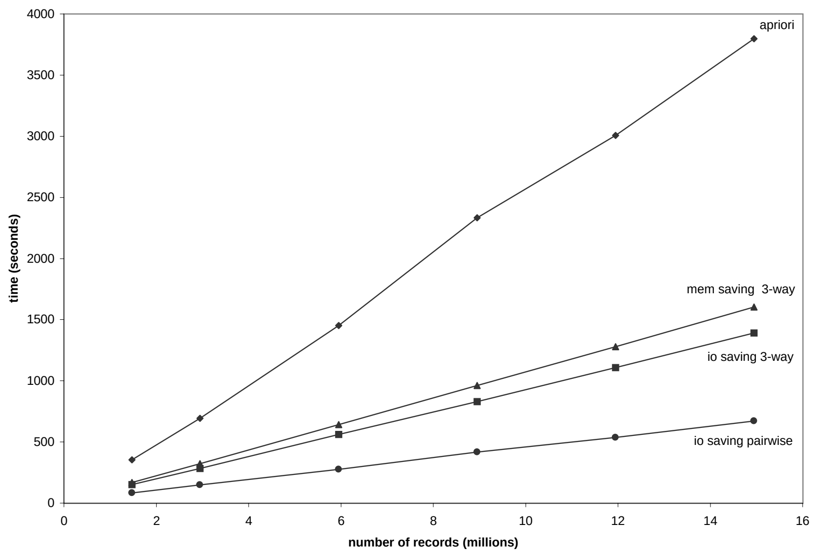


Figure 22: Scalability experiment using Star schema (of Figure 12), and with support 30%.

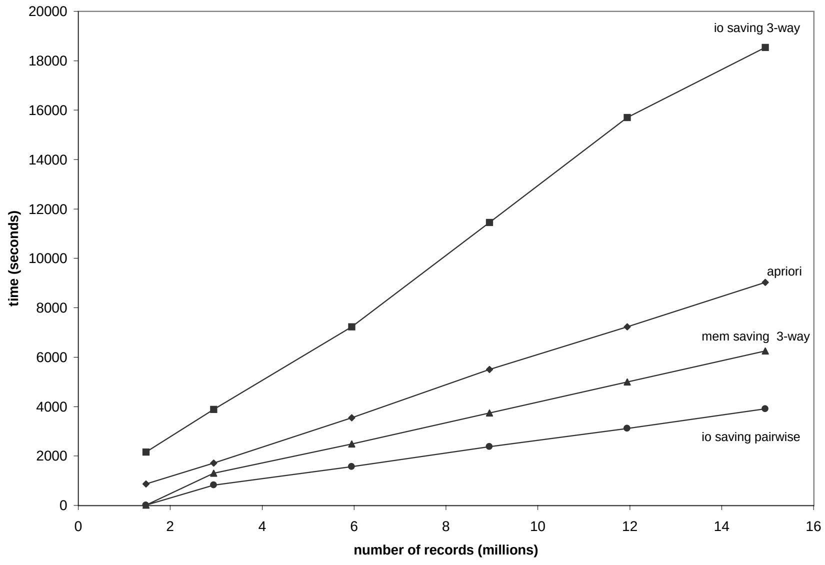


Figure 23: Scalability experiment using Star schema (of Figure 12), and with support 20%.

5. $FI(S \bowtie P, B) \cup FI(C, B) \cup CI(\{S \bowtie P, C\}, B)$

6. $FI(C \bowtie P, B) \cup FI(S, B) \cup CI(\{S, C \bowtie P\}, B)$

7. $FI(S, B) \cup FI(C, B) \cup CI(\{S, C\}, B) \cup FI(P, B) \cup CI(\{S \bowtie C, P\}, B)$

8. $FI(S, B) \cup FI(P, B) \cup CI(\{S, P\}, B) \cup FI(C, B) \cup CI(\{S \bowtie P, C\}, B)$

9. $FI(C, B) \cup FI(P, B) \cup CI(\{C, P\}, B) \cup FI(S, B) \cup CI(\{C \bowtie P, S\}, B)$

For the physical level, we consider two merging strategies for plans 2 and 3 which are (a) I/O saving, and (b) Memory saving. For the merge of two tables (plans 4-9, and a few terms in plan 3), we considered only the I/O saving strategy (else there would arise too many plans: 16 implementations for plan 3 alone).

For the support values used, there was enough memory to store the FIPRs of the inner tables (*i.e.*, tables whose keys are not ordered in the relationship file) for I/O saving, as well as to store the inner tables in the case of Memory saving. Also, when computing the prejoin of two tables (plans 4, 5, and 6), we compute the join with respect to table *ItemsBought*, since the Cartesian product of each pair of primary tables would be larger than the join with respect to *ItemsBought*.

Figure 24 shows the time taken by each of the plans for support levels 30% and 20% for each of the schemas. For plans 2 and 3, the times for both the I/O saving and Memory saving merge are shown. Note that the graph version of results are in Sections 8.2.2 and 8.2.3.

Plan	Support 30		Support 20	
	Star	Snow Flake	Star	Snow Flake
1	1453	1468	3548	3575
2(a)	560	544	7235	7559
2(b)	642	631	2486	2479
3(a)	276	243	1571	1464
3(b)	278	231	1473	1332
4	817	157	2058	958
5	1235	1255	3227	3270
6	1274	1306	3709	3797
7	257	154	1288	943
8	332	355	1845	1890
9	405	410	1869	1880

Figure 24: Run time (in seconds) for Star schema and Snow Flake schema.

The time taken by different plans varies significantly, and although most (all for support value 30%) decentralized strategies outperform the centralized Apriori, we would like to choose the optimal plan. For both schemas, plan 7 was the most efficient (I/O saving combination with *Customer* and *Store* being combined), with a pair following close: plan 4 (prejoined *Customer* and *Store*) for the Snow Flake schema, and plan 3 (Pairwise strategy) for the Star schema. Now we compare these results to the plans chosen by our heuristic algorithm presented in Section 7.2.

First, consider the Snow Flake schema that, as pointed out in Section 8.2.3, is similar to the one given in Section 2.1, only with table *Store* instead of table *Demographics*. In Section 7.3, we applied our heuristic rules to the schema from Section 2.1 resulting in two plans: (9) computing the prejoin $D \bowtie C$, and (10) merging in two steps by first merging D and C. The prejoin and combination of S (in place of D) and C translate to plans 4 and 7 above, respectively. These two plans had approximately the same runtime and turned out to be the best strategies. The plan chosen based on our heuristic rules turned out to be the best choice.

Now, consider the Star schema were our heuristic algorithm would produce a 3-way merge instead (since all dimension tables would be a direct child of the fact table in the schema tree). Our heuristic algorithm would avoid plans 4-6, therefore avoiding any prejoin (these plans are among the most expensive decentralized plans), and plans 7, 8, and 9. Finally, using Heuristic 7, we modify the final mining tree by choosing plan 3 over plan 2. We notice that the final plan chosen by our heuristic rules leads us to the second best plan.

For both schemas, our heuristic approach chose efficient plans – if fact, the best one for the Snow Flake schema. For the Star schema, although a second best plan was chosen, it was only about 14% more expensive than the best plan for support 20%, and only 7% more expensive for support 30%. It is important to note that, even without any cost estimations, the more expensive plans were avoided: plan 1 based on Heuristic 2, plan 2(a) based on Heuristic 9, and plans 5 and 6 based on either Heuristic 3, or Heuristic 5.

8.3.2 TPC-D Schema

We now examine a more complex schema, the schema based on the TPC-D benchmark shown in Figure 7, with a total of seven tables: *Orders*, *Customer*, *PartSupp*, *Part*, *Supplier*, and *Nation* from the TPC-D Benchmark, and a relationship table that we generate for the purpose of this test, *MyLineItem*, containing two attributes: *o_id* (foreign key to *Orders* table) and *ps_id* (foreign key to *PartSupp* table). For ease of exposition, we refer to the tables by their first initial (with Ps for *PartSupplier*).

The *MyLineItem* table was generated using the Almaden synthetic generator with the following parameters:

- number of transactions: 1,500,000 – corresponding to 1,500,000 orders
- transaction length: 10 – each order has on average, 10 products
- number of items: 800,000 – corresponding to 800,000 (product,supplier) pairs

The size of the tables (except for *MyLineItem*) were given in Figure 8, and the table *MyLineItem* contained about 14 million records, and was approximately 120 Mbytes in size. The table obtained by joining all tables had the same number of records, had 92 non-key attributes, and was approximately 5.3 Gbytes in size. Table *Nation*, which is referenced by both tables *Customer* and *Supplier*, has its attributes repeated in the final joined table T (not necessarily with the same values). In a sense, it is as though there were two tables for *Nation* since, in order to establish an association rule such as *C.nation_name* $\Rightarrow$ *S.nation_name*, the items referring to *C.nation_name* should be distinguished from the items referring to *S.nation_name*. In what follows, we refer to table *Nation* as either *Nation_c* (Nc) or *Nation_s* (Ns) depending on whether it is referenced by table *Customer* or *Supplier*, respectively.

In the previous example, with three primary tables there were 9 alternatives at the logical level. In this case, with 7 tables there are many more possible alternatives and we examine a few representative ones. In the following, we omit the second argument of the FI and CI notations (the weight table) for clarity. We start by considering the centralized Apriori and a few decentralized plans resulting from applying Equivalences 1, 2, and 3 to $FI(T, I)$:

1. $FI(T)$
2. $FI(O) \cup FI(C) \cup FI(Nc) \cup FI(Ps) \cup FI(P) \cup FI(S) \cup FI(Ns) \cup CI(\{O, C, Nc, Ps, P, S, Ns\})$
3. $FI(O \bowtie C \bowtie Nc) \cup FI(Ps \bowtie P \bowtie S \bowtie Ns) \cup CI(\{O \bowtie C \bowtie Nc, Ps \bowtie P \bowtie S \bowtie Ns\})$

4. $FI(O \bowtie C \bowtie Nc) \cup FI(Ps \bowtie P) \cup FI(S \bowtie Ns) \cup CI(\{O \bowtie C \bowtie Nc, Ps \bowtie P\}) \cup CI(\{O \bowtie C \bowtie Nc, S \bowtie Ns\}) \cup CI(\{Ps \bowtie P, S \bowtie Ns\}) \cup CI_3(\{O \bowtie C \bowtie Nc, Ps \bowtie P, S \bowtie Ns\})$

5. $FI(C \bowtie Nc) \cup FI(O \bowtie L \bowtie Ps) \cup FI(P) \cup FI(S \bowtie Ns)\ CI(\{C \bowtie Nc, O \bowtie L \bowtie Ps, P, S \bowtie Ns\})$

Many other plans could be generated, but for simplicity, we only consider the above five. We examine possible implementations with their respective runtime for each of the plans.

Plan 1

The only option is to join the tables and execute the centralized algorithm. The runtime is shown in Figure 25.

Plan	Support 30	Support 20
1	14024	108776

Figure 25: Run time (in seconds) for plan 1.

Plan 2

For the 7-way merge, the I/O saving algorithm was impractical due to the size of the 7-dimensional array, so we chose only the Memory saving approach, which had the runtime shown in Figure 26.

Plan	Support 30	Support 20
2	11853	103232

Figure 26: Run time (in seconds) for plan 2.

Plan 3

There are many ways in which we could further split Plan 3, and we examine a few options. For the first term, $FI(O \bowtie C \bowtie Nc)$, we considered the following sub-plans:

(a) $FI(O \bowtie C \bowtie Nc)$

(b) $FI(O) \cup FI(C) \cup FI(Nc) \cup CI(\{O, C, Nc\})$

(c) $FI(O) \cup FI(C \bowtie Nc) \cup CI(\{O, C \bowtie Nc\})$

(d) $FI(O) \cup FI(C) \cup FI(Nc) \cup CI(\{C, Nc\}) \cup CI(\{O, (C \bowtie Nc)\})$

Similarly, for the second term, $FI(Ps \bowtie P \bowtie S \bowtie Ns, L)$, we considered the following sub-plans:

(a) $FI(Ps \bowtie P \bowtie S \bowtie Ns)$

(b) $FI(Ps) \cup FI(P) \cup FI(S) \cup FI(Ns) \cup CI(\{Ps, P, S, Ns\})$

(c) $FI(Ps) \cup FI(P) \cup FI(S \bowtie Ns) \cup CI(\{Ps, P, S \bowtie Ns\})$

(d) $FI(Ps) \cup FI(P) \cup FI(S) \cup FI(Ns) \cup CI(\{S, Ns\}) \cup CI(\{Ps, P, S \bowtie Ns\})$

(e) $FI(Ps) \cup FI(P) \cup FI(S \bowtie Ns) \cup CI(\{Ps, P\}) \cup CI(\{Ps, S \bowtie Ns\}) \cup CI(\{P, S \bowtie Ns\}) \cup CI_3(\{Ps, P, S \bowtie Ns\})$

(f) $FI(Ps) \cup FI(P) \cup FI(S) \cup FI(Ns) \cup CI(\{S, Ns\}) \cup CI(\{Ps, P\}) \cup CI(\{Ps, S \bowtie Ns\}) \cup CI(\{P, S \bowtie Ns\}) \cup CI_3(\{Ps, P, S \bowtie Ns\})$

The third term, $CI(\{O \bowtie C \bowtie Nc, Ps \bowtie P \bowtie S \bowtie Ns\})$, can be implemented in different ways, among which we considered the following:

(a) I/O saving merge of 2 prejoined tables $O \bowtie C \bowtie Nc$ and $Ps \bowtie P \bowtie S \bowtie Ns$

(b) Combined I/O saving merge, considering two prejoins: $C \bowtie Nc$ and $S \bowtie Ns$

(c) Combined I/O saving merge with no prejoins.

The runtime for each of the above sub-plans is summarized in Figure 27. For the (b) plans of the first two terms we considered both I/O and Memory saving implementations. For the others, we considered only I/O saving.

	$FI(O \bowtie C \bowtie Nc)$	
Plan	Support 30	Support 20
(a)	213	463
(b).io	169	1208
(b).mem	152	377
(c)	80	211
(d)	82	211

	$FI(Ps \bowtie P \bowtie S \bowtie Ns)$	
Plan	Support 30	Support 20
(a)	204	801
(b).io	795	17279
(b).mem	232	810
(c)	160	2670
(d)	161	2297
(e)	85	379
(f)	88	382

	$CI(\{O \bowtie C \bowtie Nc, Ps \bowtie P \bowtie S \bowtie Ns\})$	
Plan	Support 30	Support 20
(a)	4554	43010
(b)	2935	46626
(c)	2933	43148

Figure 27: Run time (in seconds) for sub-plans of plan 3.

We call plan 3_{ijk} the plan corresponding to sub-expression (i) of $FI(O \bowtie C \bowtie Nc)$, sub-expression (j) of $FI(Ps \bowtie P \bowtie S \bowtie Ns)$, and sub-expression (k) of $CI(\{O \bowtie C \bowtie Nc, Ps \bowtie P \bowtie S \bowtie Ns\})$. For simplicity, assume that the particular sub-plans dictate the implementation of the final CI. For example, when both FI's correspond to fully prejoined tables, CI will be a merge of two prejoined tables, but the final CI for the plan composed of (d) and (f) will be a combined merge (since no tables were prejoined). By combining all the sub-plans listed, we have a total of 24 (=4*6) alternatives for plan 3. Instead of showing the time to all 24 possible logical combinations, we list a few in Figure 28.

Plan	Support 30	Support 20
3_{aaa}	4971	44274
3_{bbc}.io	3897	61635
3_{bbc}.mem	3317	44335
3_{ccb}	3175	49507
3_{ceb}	3100	47216
3_{ddc}	3175	45655
3_{dfc}	3103	43741

Figure 28: Run time (in seconds) for plan 3.

Plan 4

This plan contains 3 FI's merged pairwise. We considered a few strategies for implementing the first three terms, the FI's. The first term is the same as the first term in Plan 3, so we can have any of the 4 sub-plans listed above for the first FI in Plan 3. For the second and third terms, we considered: (a) prejoin the tables, and (b) perform I/O saving merge. For the CI terms, we only considered the I/O saving merge. By combining all sub-plans, we have a total of 16 (=4*2*2) logical plans, and we list a few of them in Figure 29.

Plan	Support 30	Support 20
4_{aaa}	3474	79069
4_{bbb}	3382	78917
4_{cba}	3310	78750
4_{dbb}	3312	78750

Figure 29: Run time (in seconds) for plan 4.

Plan 5

For the FI terms in this plan, we considered only two possibilities: (a) tables in each FI are prejoined, and (b) tables in each FI are merged with the I/O saving strategy (excluding the third FI which has only one table). For the CI term, we considered:

(a) 4-way I/O saving merge strategy, and

(b) 3-way I/O saving merge with the combination of tables P and $S \bowtie Ns$, *i.e.*, $CI(\{P, S \bowtie Ns\}) \cup CI(\{C \bowtie Nc, O \bowtie L \bowtie Ps, P \bowtie S \bowtie Ns\})$.

A selection of possible plans for Plan 5 with their runtime is listed in Figure 30 (for the 4-way merge we only list the runtime for support 30%).

Plan	Support 30	Support 20
5_{aa}	267635	-
5_{ab}	25881	948886
5_{ba}	266624	-
5_{bb}	24870	947270

Figure 30: Run time (in seconds) for plan 5.

This is by far the less efficient of all the plans presented. The main reasons for this are:

1. High dimensionality: 4-way merge (the most expensive), and
2. Computing $FI(O \bowtie L \bowtie Ps)$ is expensive because O and Ps do not have common attributes other than through the fact table L, resulting in an expensive FI term, and a CI term that merges very long tables (since $O \bowtie L \bowtie Ps$ has as many records as the final joined table T).

Heuristic algorithm

We now revisit our heuristic algorithm presented in Section 7.2. Figure 31(a) shows the schema tree for the TPC-D schema and Figure 31(b) shows the resulting mining tree after applying the algorithm. For the 3-way merge, the expression is converted using heuristic 7 arriving at a Pairwise merge. In the absence of particular heuristics to decide the implementation of each of the merges, the way in which the FI's were computed dictate the merging process. In this case, the plan did not have any prejoins, therefore, we used a combined merge.

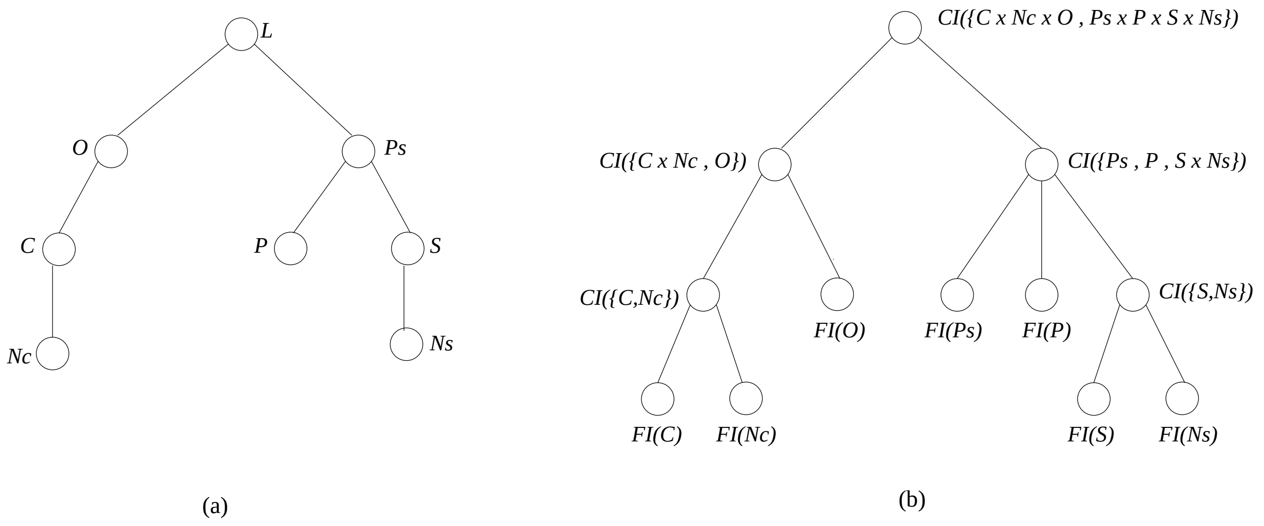


Figure 31: Schema and mining tree for TPC-D schema.

The heuristic plan corresponds to plan 3_{dfc} which was the best plan for support 20%, and the second best plan (within 1%) for support 30%. If cost estimates are available, they can be used in places where our heuristics do not help: *e.g.*, deciding whether to prejoin any of the sub-trees. Notice that the optimal plan for support 30% has 2 prejoins: $C \bowtie Nc$ and $S \bowtie Ns$.

8.4 Other Centralized Algorithms

Many suggested algorithms for frequent itemset counting use the Apriori idea (*e.g.*, see [22, 24, 6, 35]). The improvements fall into two categories: *I/O related*, where the number of scans can be reduced depending on the distribution of data values, or *computation related*, where a different internal data structure is suggested to improve the checking of candidates. In either case, the improvement of the algorithm corresponds to the reduction in the number of candidates.

Note that our Memory saving merge strategy can use the same algorithm, or data structure, as used in any Apriori-like centralized approach – with the added advantage accrued from decentralized counting, whereby the number of candidates are further reduced. Therefore, any Apriori-like centralized algorithm has an equivalent Memory saving merge strategy, with expected savings comparable to the ones shown in this section.

While the Apriori algorithm (and its many variations) has been very successful and performs well for various datasets, a few algorithms that are not simple extensions on the Apriori approach have been proposed. Below, discuss a few of these algorithms, and how they compare to a decentralized approach in the following.

8.4.1 Column-wise

A column-wise approach for finding frequent itemsets [10, 35] has been examined, which is similar to the Apriori approach in that the candidate set is generated and then checked against the database. However, it differs significantly in the way the data is accessed and processed since the database is processed one column at a time, instead of one record at a time. Dunkel and Soparkar [10] have shown this approach to work significantly better than a record-wise approach when the data has a considerably large number of columns as compared to the number of records. In this case, even if the data is originally stored in a record-wise format, it could be better to transform the data into a column-wise format and then run the column-wise algorithm.

When the number of records is significantly greater than the number of columns, the advantage of a column-wise algorithm no longer holds. While it is typical to have a significantly larger number of records than columns in a data warehouse, if the data is already in a column-wise format (for reasons other than data mining, for example), it could still be advantageous to perform column-wise mining.

8.4.2 FP-growth

The FP-growth approach proposed by Han *et al.* [15] avoids the general Apriori-like candidate set generation-and-test approach. The FP-growth approach was shown to outperform the Apriori algorithm for very low support values. The main idea behind this approach is a highly compressed data structure which holds the database in memory: the FP-tree. For all the datasets tested, the FP-tree structure fits into main memory: while an acceptable assumption for some datasets, it is not realistic for typical data warehouses with large decentralized tables since the entire joined table would need to be stored in an FP-tree structure. Therefore, multiple scans could be necessary, and the advantage of the FP-growth approach would no longer hold. Although discussions on how to implement a disk-resident FP-tree when the entire dataset does not fit into main memory are presented, an in-depth study and experimental validation is needed for large datasets. Where the FP-tree fits in memory, we suggest a decentralized approach for FP-tree construction that could provide savings when compared to a centralized FP-tree approach.

In the centralized FP-growth approach, the joined table T is scanned twice: first to determine the frequent items, and second to construct the FP-tree. While the first scan could be combined with the creation of T, the joined table will still need to be scanned a second time to construct the FP-tree. The first scan, *i.e.*, determining the frequent items, can easily be done in a decentralized way: individually at each table, as it is done in our decentralized approach. The actual FP-tree construction takes $O(r * m)$, where r is the number of records, and m, the number of attributes [15]. If an FP-tree is constructed against a joined table T, both the number of records and number of attributes can be quite large. Instead, we can construct smaller FP-trees for each individual table and then merge them, saving time for the construction of the FP-tree. As an example, let us consider our schema from Section 2.1. Suppose we want to construct an FP-tree for $Customer \bowtie ItemsBought \bowtie Product$. An FP-tree would be inexpensive to construct for the *Customer* table since the number of records in *Customer* would typically be much less than the number of records in a joined table. The records of *Customer* are inserted in the FP-tree the same way it is done in the centralized case, except that every time a record is inserted, the end position in the tree is stored for that particular record. After the FP-tree for *Customer* is built, the FP-growth algorithm is used to determine all frequent itemsets that contain items from table *Customer* only. Next, the table *ItemsBought* is processed similarly to our I/O saving merge strategy: one record of *ItemsBought* at a time. For each record in *ItemsBought*, the record of *Product* is inserted at the position in the tree where the corresponding record of *Customer* ends (the extra book-keeping that added to our FP-tree construction). By building the tree this way, the cost of insertion is about half when compared to the centralized approach. I/O costs are also reduced since the scan of the joined table can be much more expensive than scanning the *ItemsBought* and *Product* table (especially if *Product* fits into memory). Furthermore, when running the FP-growth algorithm to determine the frequent itemsets, itemsets local to table *Customer* need not be considered since they have already been counted.

8.4.3 Sampling Techniques

Sampling techniques have been examined for the problem of computing frequent itemsets [29, 34] where the database is considered to be one table, and a sample of the records in the table is selected and mined. Typically, after the frequent itemsets have been computed for the sample, a scan of the entire dataset is performed to validate the results. The sample is chosen to fit in main memory, an therefore, the algorithms run efficiently on the sample. It is advantageous to have a large enough sample to provide accurate results, and yet small enough to fit in main memory. As shown by our experiments, counting itemsets in the in-memory resident smaller dimension tables is fast when compared to counting itemsets against the large joined table. It has been noted [34] that a small sample size may generate many false frequent itemsets (and rules), and thus degrading the performance. Therefore, it would be advantageous to have a large enough sample to provide accurate results, and yet small enough to fit into main memory.

Storing samples in a decentralized manner, as opposed to storing sample records of the joined table, provides for a more compact representation of the same information. Consider a Star schema example. Instead of sampling on the join of the fact table with the dimension tables (and therefore causing expensive computation and storage requirements), a sample could be taken directly from the fact table. Depending on the records in the sample, the corresponding records of the dimension tables would be retrieved, and also stored in memory. By storing the tables decentralized in memory, we are able to store more records of the fact table than records of the joined table. We can then apply our decentralized algorithms that do not require the join to be computed. In this manner, we are able to increase the sample size, without penalizing performance, to improve the accuracy of the final results. After the sample is mined, a scan of the entire dataset is performed. As shown in our experiments, scanning a decentralized dataset provides for further performance improvement.

In the case of decentralized data, sampling can also be used as a tool to estimate costs of different merging strategies. The merging strategy that performed best in the sample data can be used against the entire dataset.

9 Decentralized Mining Extensions

There are several possible database designs, and other factors, that influence our basic decentralized approach. We described various schema in which data was stored in typical tabular formats, and the tables were joined via foreign key relationships (*i.e.*, all joins considered were natural joins). In this section, we examine different join conditions (*i.e.*, not natural joins). Furthermore, we consider use of indices (when primary indices are unavailable), and also, utilizing pre-computed data cubes (summary data) in data warehouses. In our discussions below, we refer back to our example from Section 2.1.

9.1 Condition Based Joins

In contrast to natural joins, we now examine joins based on condition on attribute values in the tables; the joining condition could be arbitrary (*i.e.*, a θ-join). There are important situations where the mining needs to be effected on such joins (as exemplified below). When the join condition involves only a few attributes from each table, we can apply algorithms that are similar to our semi-join based algorithms.

9.1.1 An Example θ-Join

As an example of condition based joins, consider a table created by the following SQL query:

```
SELECT P1.* P2.*
FROM Product P1, Product P2
WHERE P1.price > 2*P2.price
```

Each record in the above table was formed by two records of the table *Product* such that the price of one of the products is more than twice the price of the other. Suppose we want to determine the frequent itemsets in this table so that we can establish associations between attributes of products where the above condition holds (*i.e.*, one is twice the price of the other). Since there is no relationship table in this case, materializing the join and applying the centralized Apriori algorithm may appear to be the best choice. On the other hand, decentralized algorithms could be modified to handle such joins. Note that when the join result is small, a centralized approach may well be better for reasons of storage, etc. In the following discussion, we assume that the join produced is large, thereby suggesting the use of decentralized approaches.

In computing the above join, available indices will be used. Assume that the table *Product* is ordered by *p_id*, and that there is a B-tree index on the attribute *price*. At the leaves of the B-tree index, we would have the record_id, *p_id*.[9] When materializing the join, whenever a pair of records match (*i.e.*, two *p_id*s satisfy the join condition), both records in their entirety are read into memory, and the joined record is saved.

9.1.2 Decentralized Computation on Condition Joins

Instead, when a pair of records match, the approach could be just to save the matched pair of *p_id*s. By doing so, we essentially generate the relationship table, and thereafter, we could apply our decentralized approach directly. In fact, weight vectors could be computed during this creation of the relationship table (*i.e.*, step 1 of Phase I). The computation and materialization of this relationship table is less expensive than the computation and materialization of the join, and savings are also achieved by executing a decentralized computation of frequent itemsets. In case indices are not available, we could take the relevant projections of the tables (*i.e.*, *p_id* and *price*) in order to create the relationship table, similar to semi-join strategies [21] – whereas in the centralized approach, entire records need to be accessed.

When many attributes are involved in an arbitrary join condition, it could be difficult to apply an algorithm based on semi-joins, due to the number of attributes involved (and the fact that indices might not be available). In such cases, the savings in the computation of the relationship table might not be much greater than computing the entire join. However, we still save in materialization costs, and this would provide the option of running decentralized algorithms.

For both the cases (few or many attributes involved), other options may be considered. For example, instead of generating a relationship table, to keep a list (for each record of a given table) which identifies matching records of the other table. This is a more compact form for the relationship table. This would help in storage costs.

[9]In reality, the position of the record on disk could be stored at the leaves, but we assume in this case that the *p_id* is stored, and a further mapping from *p_id* to physical location is available.

9.2 Indices

Availability of indices does not especially influence the choice between centralized and decentralized DM approaches. One advantage that indices offer for centralized approaches is in the computation of the joins [25]. Whenever an index helps in the computation of a join, the same index will help in the computation of a relationship table, as shown in Section 9.1. In this section, we explore possibilities for uses of indices in the decentralized approach when a relationship table is available (*i.e.*, foreign key based decentralization). We briefly review some available indices, and explain how to take advantage of them.

9.2.1 Using Indices

We discuss the use of indices in three situations: (1) indices on the primary key of the primary table; (2) indices on the foreign keys of the relationship table; and (3) indices on non-key attributes. In the following discussions, we use the data schema presented in Section 2.1 as an example.

The savings achieved during the computation of joins in the centralized approach through the use of available indices on primary or foreign keys, are usually also achieved in a decentralized approach. In some cases, a decentralized approach can use the indices in multiple steps of the computation, providing additional opportunity for savings. Our discussion is by no means comprehensive, and where warranted, we could resort to computing the join table (therefore accruing the savings as in a centralized case) and use our Memory saving strategy with the table prejoined.

9.2.2 Index on a Primary Key

Assume there is an index on the primary key values of a primary table, such as an index on *p_id* for table *Product*. This helps with joins, and as an example, an index-join [25] could be computed where, for each record of *ItemsBought*, the corresponding records of tables *Customer* and *Product* are fetched. If both tables *Customer* and *ItemsBought* were ordered by *c_id*, then only one index for *p_id* is needed since the join of *ItemsBought* with *Customer* could be done by a merge-sort-join technique. As in the example of Section 9.1, such an index is helpful when building the joined table for the centralized approach.

In our decentralized approach, indices are utilized at several points. In Section 9.1 we used the index in the decentralized approach to compute the relationship table. We can also use an index in Phase II of our approach where, when processing a record of the relationship table we need to access the corresponding records of the primary tables. In a sense, we build a join of the tables without materializing it.

For the I/O saving merging technique, in some cases we pre-compute the frequent itemsets per record (FIPR) for the primary tables (as pointed out in Section 4). In such cases, when processing a record of the relationship table, we need to access the corresponding pre-computed FIPR, and not the record itself. When pre-computing the FIPRs, we essentially build an index such that, given a record id, we can efficiently retrieve the corresponding FIPR. In this case, we use an index which has a similar effect as the originally available index. Situations in which computing the FIPR index is impractical (*e.g.*, when the primary table is large, or we do not have enough space to store the index), we would resort to computing FIPRs on the fly, using the available index on the primary key.

9.2.3 Index on a Foreign Key

An index on foreign keys for a relationship table, such as on table *ItemsBought* for the search keys *c_id* and*p_id* can also be used to speed up join computation. Such an index can be used both in the centralized case, and in Phase II of the decentralized case. Here, we show how an index on the foreign keys can be used in Phase I of the decentralized approach.

Computation of weight vectors, as explained in Section 4, requires a scan of the relationship table in order to count the occurrences of each foreign key value. With a B-tree secondary index, say, on foreign keys, the number of pointers (or record ids) for a given foreign key value equals the number of occurrences of this foreign key value in the relationship table. Therefore, the weight vector for the primary table can be computed by reading the information in this index instead of scanning the relationship table. This may be especially useful in computing weight vectors for foreign keys which are not the ones on which a relationship table is ordered. For a foreign key that is not the primary ordering for the relationship table, the computation may access any position of the weight vector leading to blocks of the weight vector being brought in and out of memory many times. In this situation, computing the weight vectors based on the index avoids such I/O overhead.

The availability of indices on the relationship table can also provide further savings during Phase II. For the I/O saving merge strategy, instead of processing each record of the relationship table, we could use the indices on the relationship table to compute the cross table itemsets as follows. Suppose we are counting frequent itemsets across tables *Customer* and *Product*. For each pair of (*c_id*,*p_id*), we retrieve the corresponding index entries to determine the number of records of the relationship table where the pair (*c_id*,*p_id*) occurs. This can be effected by computing the size of the intersection of the two sets of pointers (or record ids). In a sense, we are computing a Cartesian product of the primary tables, and therefore, this technique will work for data sets where the relationship table is substantially larger than the primary tables.

9.2.4 Non-key Attribute Indices

Various indices are used on non-key attributes to speed up computation of queries such as select queries. Suppose, in our example, that we have an index on *salary* for table *Customer*, with two entries: one for $salary \leq 6000$ and one for $salary > 6000$. When computing the frequent itemset on table *Customer* (step 2 of Phase I), we can determine the count of the items corresponding to the attribute *salary* by counting the number of entries in each of the intervals (assuming that these intervals were found by a discretizer algorithm). Note that each entire record still needs to be retrieved in order to count the other items in the table. Therefore, indices may help if they are available for all attributes of interest: we are essentially looking at the data "vertically" (*i.e.*, for each item, we find a list of record ids where the item is present). This is similar to a column-wise approach to mining [10]. In cases where we could benefit for a column-wise approach, indices on non-key attributes (assuming the data is in a record-wise format originally) would be useful to transform the data into a column-wise format.

9.3 Data Warehouses

A data warehouse environment has data that is not updated frequently (although data may be periodically added to the repository), and most of the activity relates to the computation of complex queries. It is possible to use complex indices to speed up complex queries, and a few traditional index schemes have been suitably adapted, as well as new index schemes have been proposed (*e.g.*, see [20, 14, 32, 23, 7, 19]). In this section, we review a few of these approaches and discuss how to utilize them in our decentralized computation.

9.3.1 Value-List Index

A Value-List index for an attribute has, for each value of the attribute, a list of records (either pointers or record ids) that contain the given attribute value. The index could be a B-tree or different; these are covered in Section 9.2.

9.3.2 Bitmap Index

A Bitmap index is similar to the Value-List index, except that for each attribute value, there is a bit string length equal to the number of records. For each record that has the specific value for which the index was built, the corresponding bit in the bit string is set to 1. Bitmap indices perform well when the range of values for the attribute is small (*i.e.*, search key with low cardinality). A separate bit string is required for each attribute value, and in the case of natural joins, where the attribute of interest is the primary key (typically with high cardinality), a straight bitmap index would be prohibitively expensive. In such cases, other types of bitmap indices have been proposed such as encoded bitmap indices and bit-sliced indices [20]. Bit-sliced indexing was shown to perform well for a certain class of queries, such as summation. We are interested in count queries (in Phase I) and identifying a list of record ids for each attribute value (in Phase II), and therefore, encoded bitmaps and bit-sliced indexing have only limited use in our decentralized computation: they could simply be used in the same way as a value-list index.

9.3.3 Projection Index

A projection index [20] on an attribute of a table is the vertical partition of this attribute from the table (with no duplicate removal). If a projection index is available for a foreign key in the relationship table, weight vectors in Phase I could be easily computed by scanning each of the projection indices – each scan of an index leading to one weight vector. If projection indices are available for non-key attributes of primary tables, count of attributes could be performed directly by accessing the index. This is similar to the column-wise approach [10].

9.3.4 Join Index

Join indices provide an index on one table for a quantity that involves a column value of a different table for commonly encountered joins [31]. For example, a Star Join index on the foreign key values contains, for each value, a list of records (*e.g.*, by record id) in each of the attribute tables which contain the value. In fact, the relationship table in a Star schema is, effectively, a Star join index. Our approach as described in Section 4 utilizes such an "index" (or foreign keys in the relationship table) during the merge phase.

9.3.5 Hierarchical Index

An indexing method, such as the hierarchically split cube forests [19], has an index tree built on one dimension (say *Product*), and summary data stored at the *Product* level. Each *Product* value (*i.e.*, *p_id* in our case) contains a separate index for the *Customer* dimension, and summaries at the $Product - Customer$ level are stored. The context for this index is summary data computation, and the summary data is typically the summation of the values of an attribute. In our data schema, if we want summary information on *qty* (quantity) of table *ItemsBought*, a query for "the total quantity of white shirts purchased by customers living in area x" is efficiently answered by an index look-up. This index is very similar to a pre-computed data cube (in fact, it is an implementation of a data cube), and is discussed below.

9.3.6 Data Cubes

OLAP applications rely on pre-computed aggregates across many dimensions (*e.g.*, see [2]). The CUBE operator [13] supports such multiple aggregates in data warehouses in order that OLAP queries are answered efficiently. The operator computes group-bys corresponding to all possible combinations of a list of attributes. An example data cube for our data schema from Section 2.1 is as follows:

```
SELECT c_id, p_id, count(*)
FROM ItemsBought
CUBE-BY c_id, p_id
```

The above query results in the computation of three group-bys. Let us now examine what each of these group-bys mean in our example. A group-by on *c_id* indicates the number of entries in the table *ItemsBought* for each specific value of *c_id*. Referring to Section 4.2, observe that this is the definition of a weight vector for table *Customer*. Similarly, the group-by on *p_id* for table *Product* describes a weight vector. A group-by on *c_id*, *p_id* indicates the number of entries in table *ItemsBought* for each specific combination of values *c_id*, *p_id*.

Now, in our decentralized approach, Phase II processes each record of table *ItemsBought*. Considering the first two entries of table *ItemsBougth* in Figure 1, both are seen to have $c\_id = 100$ and $p\_id = A$. By having the described cube pre-computed, we would know that the pair $(100, A)$ has a count of two. For both merging strategies (I/O saving and Memory saving), we can exploit this information by counting each candidate corresponding to the pair $(100, A)$ once by an increment of two, instead of counting twice. This is similar to the adapted merging scheme used for Snow Flake schemas presented in Section 4.4. Also, assuming that the average count value for this particular group-by on *c_id*, *p_id* is 10, we have the CPU processing costs reduced by a factor of 10 as compared to the merging strategy used to process the entire table *ItemsBought*. For I/O costs, similar savings accrue since, instead of scanning the entire table *ItemsBought*, we only need to scan the computed cube. The cost savings using a data cube are even better than our basic decentralized approach.

For a data schema with more than two foreign keys in the relationship table, a similar approach can be used. As an example, let our original data schema from Section 2.1 have one additional attribute on the *ItemsBought* table as follows: *area* (foreign key to table *Demographics*). In such a situation, the following data cube could be computed:

```
SELECT c_id, p_id, area, count(*)
FROM ItemsBought
CUBE-BY c_id, p_id, area
```

The above query would result in the computation of 7 (*i.e.*, $2^3 - 1$) group-bys corresponding to all possible non-empty combinations of the three attributes.

In the above case, for Phase I, we would use the group-bys on individual attributes for the computation of weight vectors. During Phase II, we could use the pre-computed group-bys depending on the execution plan chosen. For 3-way merging, we would use the group-by on *c_id*, *p_id*, *area* in the same manner that we computed the 2-way merging above. For a Pairwise merge, for each pair considered (*e.g.*, *Product* and *Demographics*), we would use the relevant group-by (*e.g.*, group-by on *p_id*, *area*). The savings we achieve by using such pre-computed information depend on the average count values.

10 Conclusions and Future Work

In this paper, we examined the problem of mining decentralized datasets, and used the frequent itemset counting as an example. We provided simple algebraic means to represent and manipulate expressions that represent the mined information (which, in our case, are frequent itemsets). Our “algebra” allows enumerating the many different decentralized mining strategies – each with different processing costs. When cost estimates are available for the basic operations involved in our approach, there is an opportunity to optimize for the best strategy in a manner similar to query processing. As such, our approach may be suitably integrated with available algorithms for large-scale decentralized data mining. We describe and exemplify heuristics to reduce the overall computation, I/O and communication costs, when cost estimates are not available. Our empirical results establish the efficiency of our decentralized techniques and the validity of our heuristic optimization.

Other mining techniques also consider the data of interest to be local to one table. Both the RainForest approach [12] to classification, as well as the CACTUS system [11] used for clustering, compute summary data against a single large table. While these techniques provide significant improvements over previous ones, if the dataset contains multiple tables, our decentralized counting techniques could be used to avoid scans of the full datasets (by replacing them with scans of smaller tables). As an example, for classification, the class label distribution could be counted in a decentralized way. Consider a Star schema where class labels are associated with each record in the fact table. In order to compute a class label distribution for a particular attribute value, the occurrences of records in the fact table with respect to each specific class need to be known. Therefore, weight vectors need to be computed for each class label (for each dimension table). By using such weight vectors, the distribution of a class label for attributes of a dimension table can be computed locally at the dimension table. This approach avoids the expensive computation and materialization of joins, as well as provides savings by scanning smaller tables during computations. When entire database partitions need to be written out, we could write out just the partition of the fact table.

Note that as focus shifts from improving the performance of basic algorithms for small centralized tables to decentralized, real-life datasets, we believe that approaches such as ours will become increasingly important.

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] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proceedings of the 22th Int'l Conference on Very Large Data Bases*, 1996.

[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] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. IBM Research Report: RJ 9839. IBM Research Division, Almaden Research Center, San Jose, CA 95120-6099., 1994.

[6] 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.

[7] C.-Y. Chan and Y. Ioannidis. Bitmap index design and evaluation. In *Proceedings of ACM SIGMOD Int'l Conference on Management of Data*, 1998.

[8] 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.

[9] V. Crestana and N. Soparkar. Mining decentralized data repositories. Technical Report: CSE-TR-385-99. The University of Michigan, EECS Dept. Ann Arbor, USA. February 1999.

[10] 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.

[11] V. Ganti, J.Gehrke, and R. Ramakrishnan. CACTUS – clustering categorical data using summaries. In *Proceedings of the 1999 SIGKDD Conference*, 1999.

[12] J. Gehrke, R. Ramakrishnan, and V. Ganti. Rainforest – a framework for fast decision tree construction of large databases. In *Proceedings of the 24th Int'l Conference on Very Large Data Bases*, 1998.

[13] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *Proceedings of the 12th IEEE Int'l Conference on Data Engineering*, 1996.

[14] H. Gupta, V. Harinarayan, A. Rajaraman, and J. Ullman. Index selection for olap. In *Proceedings of the 13th IEEE Int'l Conference on Data Engineering*, 1997.

[15] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings of ACM SIGMOD Int'l Conference on Management of Data*, 2000.

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

[17] V. Crestana Jensen and N. Soparkar. Algebra based optimization strategies for decentralized mining. Technical Report: CSE-TR-437-00. The University of Michigan, EECS Dept. Ann Arbor, USA., 2000.

[18] V. Crestana Jensen and N. Soparkar. Frequent itemset counting across multiple tables. In *Proceedings of the 4th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 2000.

[19] T. Johnson and D. Shasha. Some approaches to index design for cube forests. *IEEE Data Engineering Bulletin*, 20(1), March 1997.

[20] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *Proceedings of ACM SIGMOD Int'l Conference on Management of Data*, 1997.

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

[22] 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.

[23] S. Sarawagi. Indexing OLAP data. *IEEE Data Engineering Bulletin*, 20(1), March 1997.

[24] 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.

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

[26] 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.

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

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

[29] H. Toivonen. Sampling large databases for association rules. In *Proceedings of the 22th Int'l Conference on Very Large Data Bases*, 1996.

[30] Transaction Processing Performance Council. http://www.tpc.org., May 1995.

[31] P. Valduriez. Join indices. *ACM Transactions on Database Systems*, 12(2), June 1987.

[32] M.-C. Wu and A. Buchmann. Encoded bitmap indexing for data warehouses. In *Proceedings of the 14th IEEE Int'l Conference on Data Engineering*, 1998.

[33] M. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, 7(4), December 1999. Special issue on Parallel Mechanisms for Data Mining.

[34] M. Zaki, S. Parthasarathy, Wei Li, and M. Ogihara. Evaluation of sampling for data mining of association rules. In *Proceedings of the 7th Int'l Workshop on Research Issues in Data Engineering (RIDE)*, 1997.

[35] M. Zaki, S. Parthasarathy, and M. Ogihara. New algorithms for fast discovery of association rules. In *Proceedings of the 3rd Int'l Conference on Knowledge Discovery and Data Mining*, 1997.