Algebra-based Optimization Strategies for Decentralized 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

In previous work [6, 10], we demonstrated the importance of mining of decentralized data (i.e., normalized tables, possibly residing in several repositories, with several schemas, separate administration etc) in contrast to current technology which applies to centrally stored data. Our decentralized approach to mining data computes partial results at the separate tables, and merges them to obtain the final one – thereby avoiding the expensive materialization of joins. We provided approaches to mining multiple tables located in a data warehouse, and indicated how these may be further extended. Our decentralized algorithms for finding frequent itemsets (used in association rules and classification algorithms) across multiple tables were analyzed for their performance characteristics, and these were validated empirically as well.

In this paper, we detail our approach as extended to general database schema designs, including physically distributed tables, and we describe our techniques which are similar to standard query optimization and processing. We present an algebra (more as an explanatory tool than for rigor) over expressions that denote the mined information (frequent itemsets, in our case) to describe how our basic decentralized mining approaches apply to generic database schema designs. To complement our algebra, we provide useful equivalences among the expressions, and this allows expression re-writing to represent different alternatives possible during the mining. Each equivalent expression indicates several processing strategies for decentralized and/or distributed designs. Using cost formulae available from our previous work, we present the choice among the processing strategies as an optimization problem as regards the efficiency in execution. We discuss the parameters needed for estimating the cost of different executions and finally, we describe heuristics to reduce the search within the space of possible execution plans.

1 Introduction

Data mining (DM) is generally performed on data stored in data warehouses, as opposed to transactional/operational data. Such data is not stored in a single table as is assumed by DM algorithms (e.g., see [3, 18]). Typical DM algorithms applied to such decentralized data, require that, first the join of all the tables be computed in order to produce a single table. 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

*This research was supported, in part by NSF Grant 9978510.

well (e.g., see [6, 10]). While DM is often applied to data warehouses, even in a centralized data warehouse design, data is stored in a *star schema* [15], in which information is decentralized into facts and dimension tables. Facts form the relationships among the dimensions which contain the attributes about the entities being modeled. Such schema represent vertical partitioning in a database design.

Even though database design[1] impacts the efficiency of DM, few techniques (e.g., see [6, 10]) for mining general designs in which tables are vertically and/or horizontally partitioned has been examined. Centralized database designs (i.e., with data stored in one central repository, homogeneous with a central administration, and in a single schema such as one table) is not typical for most large enterprises. The 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 an environment, design decisions involve placement of data and programs across the sites in the computer network (e.g., see [12]). Design considerations such as data fragmentation, distributions strategies, and data allocation methods need to be examined. Similar to query processing and optimization ([16]), the decentralization and distribution dictates different ways in which a decentralized DM approach is to be effected for efficiency. Therefore, besides techniques for identifying all the different options for DM, techniques for deciding the best among them is also important.

Our previous work [6, 10] described decentralized algorithms for finding frequent itemsets (used in association rules and classification algorithms) in decentralized schema. We compute partial results at the separate tables, and merge them to obtain the final one. Our approach avoids the expensive materialization of joins. In this paper, we explain how our approach is extended to more general database schema designs, including physically distributed tables, and we develop algebraic techniques to describe the processing and optimization of our approach. Our approach closely resembles the expression re-write techniques in relational database query processing, and as such, may be suitable to include inside a database system. Note that others have also suggested an approach which resembles database query optimization techniques (e.g., see [9]).

The remainder of this paper is organized as follows. In Section 2, we illustrate the available techniques for mining decentralized data with an example. In Section 3, we present our algebra (more as a tool than for mathematical rigor) over expressions and equivalences for expression re-writing. In Section 4, we exemplify cost formulas to use for the different possible techniques. In Section 5, we describe the choice of processing strategies as an optimization problem and present heuristics to reduce the search within the space of possible execution plans. Finally, in Section 6, we discuss relevant related work and present our conclusions.

[1] Relational databases are designed to contain several inter-related tables, often generated by the process of normalization. Goals of normalization include generating an appropriate set of table schemas to reduce redundancy, to check efficiently for certain constraints on the data values (e.g., functional dependencies), and yet to allow for easy and efficient data access (e.g., see [16]). In general, a relation schema (i.e., for a table) with many attributes is decomposed into several schemas (leading to several inter-related tables) with fewer attributes. Each such table could reside at a different site, and even when the data resides at one site, the data is decentralized among different tables.

2 Mining Decentralized Tables: Many Choices

Decentralized data schema may involve tables that are horizontally or vertically partitioned, physically distributed, and even replicated. A simple approach to applying a DM algorithm, such as for finding frequent itemsets, is to reconstruct the data into a single central table, thereby requiring to ship information and perform the joins. Alternatively, the DM algorithms may be modified to run on the individual sites separately, and thereafter, the results "merged" (e.g., see [6, 10] where the approach is detailed). How precisely to merge these results requires careful considerations, and there are efficiency trade-offs between the traditional and the decentralized approaches.

We use the problem of discovering association rules (AR), introduced in [2], to develop our ideas. 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 find associations among the items such that the presence of some items in a row will imply the presence of other items in the same row. Discovering AR was decomposed into: (1) discovering all the frequent itemsets (i.e., which meet a user-defined support threshold); and (2) generating all the AR, based on the support counts found in step 1. In [3], the *Apriori* algorithm improved upon the performance of the original algorithm of [2]. The Apriori algorithm performs the counting of itemsets in an iterative manner, by counting the itemsets that contain k items (k-itemsets) at iteration k. In each iteration, a candidate set of frequent itemsets is identified and the database is scanned to count the number of occurrences of each candidate itemset. Thereafter, the frequent (i.e., with a count above the predetermined threshold) itemsets are found, and re-used to compute the set of candidates for the next iteration. The finding of frequent itemsets is expensive in terms of processing, and the research in AR concentrates on improving its performance as is our focus.

2.1 An Example

We now use an example schema (adapted from [10]) to describe some key problems in decentralized DM.

- $Customer(\underline{ssn}, name, street, area, age, salary)$[2]
- $Demographics(\underline{area}, weather, schoolsystem, avghouseprice)$
- $Product(\underline{pid}, description, color, size, price)$
- $ItemsBought(\underline{xact\#}, ssn, pid, date, qty)$

In Figure 1, we show a relevant projection of the tables, and we assume that the quantitative attributes (i.e., age and monetary amounts) are partitioned using an appropriate algorithm (e.g., see [18]). Also assume that the $ItemsBought$ table is horizontally partitioned (and physically distributed) into two tables: $ItemsBought1$ and $ItemsBought2$. $ItemsBought2$ and $ItemsBought1$ contain the transactions of customers who live in areas z, and x or y, respectively. This illustrates a case where there are retail stores in areas z and x, and area y is nearer to area x than to area z.

For this example, note that traditional approaches to discovering AR work relatively efficiently for finding associations such as: $< size : large > \Rightarrow < price : high >$ within table $Product$. However, the same

[2] Each table's primary key is underlined; entries in $ItemsBought$ for the same (ssn,pid) correspond to different $xact\#$'s.

Demographics		
area	weather	schools
x	hot	good
y	mild	medium
z	cold	excellent

Customer			
ssn	salary	area	age
01	100000	x	20
02	55000	z	25
03	100000	y	20
04	20000	y	30
05	50000	x	31
06	100000	z	35

ItemsBought1	
ssn	pid
01	A
01	A
03	A
03	B
04	B
04	E
05	A
05	A
05	D

ItemsBought2	
ssn	pid
02	A
02	C
02	C
06	C
06	F

Product			
pid	color	size	price
A	white	small	10
B	blue	small	40
C	green	med.	42
D	white	med.	150
E	gray	large	100
F	green	large	110

Figure 1: Relevant projection of the tables.

approach would be inefficient for finding $< weather : hot >=>< color : white >$ (i.e., across *Demographics*, *Customer*, *Product* and *ItemsBought* tables), or $< size : large >=>< age : 30..35 >$ (i.e., across *Product*, *Customer* and *ItemsBought* tables). The reason for the inefficiencies in the traditional techniques arise for rules involving more than one table, where the tables first need to be joined ($Demographics \bowtie Customer \bowtie (ItemsBought1 \cup ItemsBought2) \bowtie Product$). There is significant redundancy in such joined tables. For example, the itemset $\{< age : 30..39 >, < area : x >\}$ that occurs three times in the joined table would be counted three times by traditional algorithm; and yet, it corresponds to just one entry in the *Customer* table, namely the one for primary key $ssn = 05$.

As an alternative to the inefficiencies in traditional DM techniques, we consider available techniques for decentralized DM. These are decentralized DM techniques for horizontally partitioned data (e.g. [5]), and in our previous work, we provided approaches for the basic cases involving vertical partitioning [6, 10]. We describe these approaches below in the context of our example.

2.2 Horizontal Partitioning

Other work (e.g., [5]) has considered horizontal distribution, but not in combination with vertical partitioning such as the case illustrated by our example in Figure 1. That is, the database designs were limited to a single table horizontally partitioned among different sites. In order to use the distributed algorithm in [5], two joins need to be materialized: ($Demographics \bowtie Customer \bowtie ItemsBought1 \bowtie Product$) and ($Demographics \bowtie Customer \bowtie ItemsBought2 \bowtie Product$). Thereafter, the algorithm, at each iteration,

counts the support of candidate sets locally. This requires an exchange of messages with counts after each iteration of the algorithm. Furthermore, the itemsets from tables *Demographics*, *Customer* and *Product* that were central in one table and location, now are across multiple locations. Therefore, as an example, in order to count itemsets that involve only the attributes of table *Customer*, counts need to be exchanged after each iteration, and it is best not to have to incur communication costs in doing so.

2.3 Vertical Partitioning

For vertically partitioned tables, our decentralized approach [6, 10] finds frequent itemsets in separate steps. This works well for finding itemsets in a schema such as *Customer*, *ItemsBought* (not horizontally partitioned) and *Product*. The approach is to find itemsets containing items from table *Customer* only; find itemsets containing items from table *Product* only; and then find itemsets containing items from all three tables. The final results from our algorithm (i.e., the frequent itemsets) are exactly the same as with any traditional itemset counting algorithm run on the joined table $T = Customer \bowtie ItemsBought \bowtie Product$. To ensure these final results, before we count frequent itemsets in the separate tables, we take into account how many times each row of the table is in the final joined table T. For this example (i.e., a single *ItemsBought* table, and no *Demographics* table), our algorithm from [10] proceeds as follows (and resembles a semi-join based technique common in distributed query processing).

- **Phase I:** Count itemsets on tables *Customer* and *Product*

  1. *Compute a projection of table ItemsBought*

  Count the number of occurrences of each value for the foreign keys *ssn* and *pid* in table *ItemsBought*. Now, the number of rows in *ItemsBought* is the same as the number of rows in the joined table T, and we can determine the number of times each row of *Customer* and *Product* will appear in table T. We store this number of occurrences in a *weight vector* for each of *Customer* and *Product*.

  2. *Count frequent itemsets on separate tables*

  Find the frequent itemsets on tables *Customer* and *Product* using any traditional centralized algorithms (e.g., see [3, 4]) with the following modification: when incrementing the support of an itemset present in row i by 1, increment the value of the support by the number of occurrences of $id_t = i$ in *ItemsBought* (given by the weight vector computed above). In this way, for *Customer* and *Product*, we find the sets of frequent itemsets, $l_{Customer}$ and $l_{Product}$ respectively.

- **Phase II:** Count frequent itemsets across the separate tables using the table *ItemsBought*

  This step can be effected in different ways, and we present one of the approaches detailed in [6, 10].

  - *I/O saving merge:* Generate candidates from *Customer* and *Product* using a 2-dimensional array, where one dimension corresponds to $l_{Customer}$ and the other to $l_{Product}$. For each row in table *ItemsBought*, identify all frequent itemsets from $l_{Customer}$ and $l_{Product}$, say $i_{Customer}$, $i_{Product}$ respectively, that are present in the corresponding rows of tables *Customer* and *Product*. This can be done on the fly, i.e., after reading each row in table *ItemsBought*, or can be done by essentially computing the join of the three tables and processing each row of the joined table as generated. In the 2-dimensional array, the support of an element I corresponding to the union

of one element from $i_{Customer}$ and one from $i_{Product}$, indicating that element I appeared in this particular row of $ItemsBought$, is incremented. That is, increment all positions (J, L) such that J is an element of $i_{Customer}$ and L is an element of $i_{Product}$. After table $ItemsBought$ is processed in this manner, the 2-dimensional array contains the support for all the candidate itemsets which can be used to determine which itemsets are frequent. If indices for the entries of *ssn* and *pid* on table $ItemsBought$ are available, instead of processing the entries of $ItemsBought$, for each pair of rows from $Customer$ and $Product$, we can use the indices to determine which entries in $ItemsBought$, the rows appear together (in reality, since in this example there are no other attributes on $ItemsBought$, all you need is the number of times they appear together, regardless of where).

This technique applied to our example in Section 2.1, would need the join $Demographics \bowtie Customer$, and the union $ItemsBought1 \cup ItemsBought2$ to be computed first. However, first obtaining the join of $Demographics$ and $Customer$ loses the advantage of only assessing itemsets local to $Demographics$. Similarly, the union of $ItemsBought1$ and $ItemsBought2$ 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.

2.4 Optimizing among Alternatives

Our description indicates that there are several alternatives to computing the join and/or union first when dealing with decentralized tables. In order to choose among alternatives freely, there is a need to unify the decentralized DM approaches for the horizontal and the vertical partitioning, and below, we exemplify some ways to do so.

We can adapt our algorithm from Section 2.3 for horizontal partitions as follows. Assume each table is at a different site.[3] For Phase I, we compute the weight vectors for $ItemsBought1$ and $ItemsBought2$, and ship them to the sites where $Customer$ and $Product$ are located. At each site, the weight vectors from each of the partitions of the table $ItemsBought$ are combined (by a simple vector addition operation). Thereafter, the second part of Phase I proceeds as in the non-distributed case (i.e., frequent itemsets are counted locally). For Phase II, $Customer$ and $Product$ are shipped to the location of $ItemsBought1$ and $ItemsBought2$ (as would have been needed for joining the tables, if using the approach in [5]), and the information regarding the itemsets found frequent at each one of $Customer$ and $Product$ is also shipped. Thereafter, the Phase II counting of the cross-table itemsets can proceed concurrently at the sites for $ItemsBought1$ and $ItemsBought2$. There is only one exchange of messages needed at the end of the counting(to determine, finally, the frequent itemsets).

Similarly, we can adapt our algorithm to decentralized computation from $Demographics$. After computing the weight vector for $Customer$ w.r.t. $ItemsBought$, we compute a weight vector for $Demographics$ (taking into account $ItemsBought$). In our example, the weight vector for $Customer$ w.r.t. $ItemsBought$ is $(2, 3, 2, 2, 3, 2)$. The first row of $Demographics$ $(area = x)$ occurs 5 times at the joined table: 2 due to the first row of $Customer$ $(ssn = 01)$, and 3 due to the fifth row of $Customer$ $(ssn = 05)$ – these are the two entries in $Customer$ which have $area = x$. Therefore, the weight vector for $Demographics$ w.r.t.

[3] The horizontal partitions are assumed to be disjoint.

ItemsBought is $(2+3, 2+2, 3+2) = (5, 4, 5)$. In this manner, we may determine the number of times each row of *Demographics* would appear in the final table, and therefore, count *Demographics* separately. At the merging phase, we would add another dimension for *Demographics*, or we could first merge $Customer$ with *Demographics*, and then merge this result with *ItemsBought* and $Product$.

Note that for the example in Section 2.1, there are relatively few options to decentralized DM. For more involved designs, however, there could a large number of options. Similar to query processing, the more tables that need to be joined, the more options there are in processing of the joins. In a sense, in Section 2.3, we processed a 3-way join instead of effecting the merging in two steps (which would be another alternative). Each strategy involves its own cost of execution, and in order to choose the best strategy, we need a clear and simple means to enumerate the possibilities, and then to decide which one is the best based on cost metrics. We examine these issues in the bulk of this paper.

3 Expression Equivalences

We provide an "algebra" over expressions, where each expression represents the mined information at a particular stage. When counting frequent itemsets in a particular table, we are primarily interested in itemsets whose items are non-key attributes; the key attributes are used to inter-relate tables. Therefore, frequent itemsets of a table X will refer to itemsets involving non-key attributes of X with a support greater than a pre-specified minimum support threshold. The basic terminology 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.
- $CI(\{X_1, X_2, \cdots, X_n\}, W)$: the "cross-itemsets", which are the set of frequent itemsets from table $X_1 \bowtie X_2 \bowtie \cdots \bowtie X_n \bowtie W$ involving non-key attributes of table $X_1 \bowtie X_2 \bowtie \cdots \bowtie X_n$, such that an itemset contains items from at least two tables.

Above, the term W, the *weight table*, in the $FI(X, W)$ denotes the table from which the weight vector will be computed, and in $CI(\{X_1, X_2, \cdots, X_n\}, W)$, it denotes the table that links the X_i tables.

Let I denote the identity operand for the join operator (i.e., $I = \pi_{keyofT}(T)$ in order that $T \bowtie I = T$). As an example illustrating the use of our algebra, consider the case in Section 2.1 where we were seeking to find $FI(T, I)$, where $T = Demographics \bowtie Customer \bowtie (ItemsBought1 \cup ItemsBought2) \bowtie Product$. Similarly, in Section 2.3, we computed:

- $FI(Customer, ItemsBought)$ and $FI(Product, ItemsBought)$ in Phase I, and
- $CI(\{Customer, Product\}, ItemsBought)$ in Phase II.

3.1 Equivalences

Our approach to exploiting algebraic manipulations starts with an expression such as $FI(T, I)$, where T represents the table arising from joining and/or creating the union of all the decentralized tables. Thereafter, we aim to obtain an expression that involves finding itemsets in separate tables, and then merging the results. As mentioned in Sections 1 and 2, table T could be a table that is not materialized, and the tables that form table T when joined could be decentralized and/or physically distributed. Using the equivalences below, we can provide alternative expressions that are equivalent, and that may be less expensive to realize, than using traditional techniques which first materialize T. We present the following equivalences to help in this process:

$$FI(X \bowtie Y, W) = FI(X, X \bowtie Y \bowtie W) \ \cup \ FI(Y, X \bowtie Y \bowtie W) \ \cup \ CI(\{X, Y\}, X \bowtie Y \bowtie W) \tag{1}$$

$$CI(\{X, Y, Z\}, W) = CI(\{X, Y \bowtie Z\}, W) \ \cup \ CI(\{Y, Z\}, W) \tag{2}$$

We explain why the above equivalences hold. For equivalence (1), the expression on the left represents the set of frequent itemsets for table $X \bowtie Y \bowtie W$. The expression on the right, represents the same set as a union of three subsets: the set of frequent itemsets from $X \bowtie Y \bowtie W$ that involve only items from table X, that involve only items from table Y, and that contain items from both tables 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. Similar explanations apply to the other equivalence. It should be clear that the expressions on the right of both (1) and (2) represent a more decentralized DM approach.

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 $FI(X, W)$. This is also true for CI similarly. Therefore, we have

$$\begin{aligned} FI(X, W) &= FI(X, \pi_{id_x}(W)) \\ CI(\{X, Y\}, W) &= CI(\{X, Y\}, \pi_{id_x, id_y}(W)) \end{aligned} \tag{3}$$

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. Besides the equivalences listed above, the usual equivalences of relational algebra expressions (e.g., the commutativity and associativity of joins) continue to hold. In particular, the join distributes over unions – which is useful in considering horizontal partitions. As an example, we have

$$FI(X \bowtie (Y_1 \cup Y_2) \bowtie Z, W) = FI((X \bowtie Y_1 \bowtie Z) \cup (X \bowtie Y_2 \bowtie Z), W) \tag{4}$$

However, notice that once we distribute a join over a union, we can no longer compute local frequent itemsets locally at tables X and Z. The reason is, as an instance, if we use our decentralized approach for vertically partitioned cases for $X \bowtie Y_1 \bowtie Z$, we would to find itemsets for attributes of X according to the occurrences of X in $X \bowtie Y_1 \bowtie Z$, and not $X \bowtie (Y_1 \cup Y_2) \bowtie Z$ (which is what is needed). By distributing the join over the union, we limit ourselves to using the algorithm for horizontal partitioning for the tables involved. In some cases, this would be the preferred approach, while in other cases, it may be a bad choice – which would be assessed based on the associated costs for evaluating each expression. As an example, in Section 2.2, we wanted to compute $FI(T, I)$, where $T =$

$Demographics \bowtie Customer \bowtie (ItemsBought1 \cup ItemsBought2) \bowtie Product$. Distributing the join over the union, we get: $FI(T_1 \cup T_2, I)$, where $T_1 = Demographics \bowtie Customer \bowtie ItemsBought1 \bowtie Product$ and $T_2 = Demographics \bowtie Customer \bowtie ItemsBought2 \bowtie Product$. We explained in Section 2.2 why distributing the join was not a good idea in this case, and similar arguments could be constructed for other situations as well. In Section 5.2, we re-visit the issue of distributing the join over unions.

3.2 Expression Re-writing

We use the example from Section 2.1 and illustrate the use of our equivalences for expression re-writing. Restated, we aim to find $FI(T, I)$, where $T = Demographics \bowtie Customer \bowtie (ItemsBought1 \cup ItemsBought2) \bowtie Product$, which we shorten to $T = D \bowtie C \bowtie (B1 \cup B2) \bowtie P$. Therefore, the traditional centralized approach gives us the first alternative to the computation

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

Our equivalences help to develop a decentralized processing plan as follows.

By using the commutativity of joins and equivalence 1, we have:

$$\begin{aligned} FI(T, I) \quad = \quad & FI(B1 \cup B2, D \bowtie C \bowtie (B1 \cup B2) \bowtie P) \ \cup \\ & FI(D \bowtie C \bowtie P, D \bowtie C \bowtie (B1 \cup B2) \bowtie P) \ \cup \\ & CI(\{B1 \cup B2, D \bowtie C \bowtie P\}, D \bowtie C \bowtie (B1 \cup B2) \bowtie P) \end{aligned}$$

Tables D and C do not have common attributes with table P, therefore their join would be their cartesian product. While this may seem to be a bad choice, by applying more equivalences, we could arrive at a better expression. Since $B1 \cup B2$ has 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 (irrespective of X and W), since by the definition of CI, the frequent itemsets should have at least one non-key attribute from $B1 \cup B2$. So, we have

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

Now, using equivalences 1 and 3, we get

$$\begin{aligned} FI(T, I) \quad = \quad & FI(D, D \bowtie C \bowtie (B1 \cup B2) \bowtie P) \ \cup \\ & FI(C \bowtie P, D \bowtie C \bowtie (B1 \cup B2) \bowtie P) \ \cup \\ & CI(\{D, C \bowtie P\}, D \bowtie C \bowtie (B1 \cup B2) \bowtie P) \end{aligned} \tag{3}$$

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

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

And so, we arrive at a new expression

$$\begin{aligned} FI(T,I) &= FI(D, \pi_{area}(C \bowtie (B1 \cup B2)) \cup \\ &\quad FI(C \bowtie P, \pi_{ssn,pid}(B1 \cup B2)) \cup \\ &\quad CI(\{D, C \bowtie P\}, \pi_{area,ssn,pid}((B1 \cup B2) \bowtie C)) \end{aligned} \quad (4)$$

Using equivalence 1 in the second term gets us to

$$\begin{aligned} FI(T,I) &= FI(D, \pi_{area}(C \bowtie (B1 \cup B2)) \cup \\ &\quad FI(C, \pi_{ssn}(B1 \cup B2)) \cup FI(P, \pi_{pid}(B1 \cup B2)) \cup CI(\{C, P\}, \pi_{ssn,pid}(B1 \cup B2)) \cup \\ &\quad CI(\{D, C \bowtie P\}, \pi_{area,ssn,pid}((B1 \cup B2) \bowtie C)) \end{aligned} \quad (5)$$

Using equivalence 2 in the last two terms:

$$\begin{aligned} FI(T,I) &= FI(D, \pi_{area}(C \bowtie (B1 \cup B2)) \cup FI(C, \pi_{ssn}(B1 \cup B2)) \cup FI(P, \pi_{pid}(B1 \cup B2)) \cup \\ &\quad CI(\{D, C, P\}, \pi_{area,ssn,pid}((B1 \cup B2) \bowtie C)) \end{aligned} \quad (6)$$

Our algebraic manipulation provides expressions equivalent to the original $FI(T,I)$, and yet, it represents avoiding having to first create the join $D \bowtie C \bowtie (B1 \cup B2) \bowtie P$. In fact, expression 5 significantly decentralizes the computation of frequent itemsets into the individual tables, and avoids the union of the horizontal partitions. By computing each individual term in the final expression, we obtain the set of frequent itemsets for table T.

As noted in [6, 10], in some cases, merging the results from several tables simultaneously (i.e., computing CI) may prove to be expensive (due to memory space requirements). If so, we could merge partial results in several steps (some of which are detailed in [6] – e.g., by multiple passes over weight table, i.e., the table that relates the various tables). This could be done in different ways, such as by using expression 4, or by computing the join of a few tables in advance (note that both the join and union operations are commutative and associative). For example, we could join D and C early, and use

$$\begin{aligned} FI(T,I) &= FI(D \bowtie C, \pi_{ssn}(B1 \cup B2)) \cup FI(P, \pi_{pid}(B1 \cup B2)) \cup \\ &\quad CI(\{D \bowtie C, P\}, \pi_{ssn,pid}(B1 \cup B2) \end{aligned} \quad (7)$$

Another strategy may be to distribute the join over the union from expression 1 as follows

$$FI(T,I) = FI((D \bowtie C \bowtie B1 \bowtie P) \cup (D \bowtie C \bowtie B2 \bowtie P), I) \quad (8)$$

However, we again notice that distributing the join over the union, limits processing to algorithms meant solely for horizontally partitioned cases; and further re-writing is limited as well.

As we can see, even for this small problem involving only four tables, we have many possible strategies to obtaining the frequent itemsets (note that all possibilities were not enumerated). By associating processing costs to each operation, each expression may be associated with a cost, and an optimization problem arises. This problem may be solved to find the best strategy to use in terms of efficiency.

4 Incorporating Processing Costs

In this section, we discuss the costs involved in computing FI, CI and $\cup$ (some basic cases are drawn from details in [5]). We discuss the join operation since several possible alternative strategies involve the joining of certain tables prior to computing FI or CI. Also, we discuss how to estimate parameters that affect the cost of execution of a mining strategy, and provide rough cost estimates for our discussed examples.

4.1 Cost Formulas

Let,

- r_i be the number of rows 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
- $|c_j^i|$ be the number of candidates of length j from table T_i
- $|l_j^i|$ be the number of frequent itemsets of length j from table T_i

When considering a term such as $FI(T_i, T_w)$, T_i and/or T_w may have to be computed ahead of time (e.g., if $T_i = T_{i_1} \bowtie T_{i_2}$ the join needs to be computed, or if $T_i = T_{i_1} \cup T_{i_2}$ the union needs to be computed the union. When discussing the costs of $FI(T_i, T_w)$, we do not include the costs of computing T_i and T_w (i.e., the join and the union mentioned, are discussed separately). In cases where we do not expect to compute the union, for example, $FI(T_{i_1} \cup T_{i_2}, T_w)$, and instead use an algorithm that works on horizontally distributed data (e.g., from [5]), we will leave the union symbol, $\cup$, stated explicitly.

We now describe the individual operation costs. We separate the processing costs into local costs (I/O and CPU), and communication costs. For the CPU costs, the "subset operation" (checking of a row against a set of candidates itemsets) is the most expensive CPU operation, and therefore, we only consider this operation as far as CPU costs are concerned. Note that, in several instances, indices may be used effectively, and we provide a few example cases below.

4.1.1 $FI(T_i, T_w)$

The costs of finding the frequent itemsets of a table depend on the algorithm used. We may use an approach where we compute the weight vector from T_w and then, a centralized algorithm must be chosen (e.g., [3, 4, 14]), and modified slightly (as mentioned in Section 2.3). Therefore, the cost will be that of computing the weight vector, and applying the chosen algorithm to find the frequent itemsets of T_i. As an example, we consider the Apriori algorithm [3] to exemplify the costs (both I/O and CPU cost formulas for the vertical partitioning case, and their derivations for the centralized case, are discussed in detail in [6]). We regard this computation to occur in two steps:

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

- *Local costs:* Scan table T_w and compute the weight vector for T_i. The scan cost (I/O cost) is given by the number of rows, r_w, in T_w, and the weight vector size is given by the number of rows, r_i, in T_i. If the weight vector fits in main memory, the CPU costs are those of projecting the scanned T_w into r_i. If it does not fit in main memory, there will be I/O costs involved in the computation and writing of the weight vector. Therefore, the I/O and CPU costs are given by the parameters r_w and r_i. Now, consider the situation where an index on table T_w is available for each row of T_i. In that case, we may use the index to determine how many entries a particular row of T_i has in T_w (note that we do not need to know the specific entries, just how many), by simply reading the index. In such case, the costs are given by r_i (and not r_w).
- *Communication costs:* The computation of the weight vector from T_w is local to the site where T_w is, and therefore, there are no communication costs involved here.

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

   - *Local costs:* These are the usual costs of computing frequent itemsets for one single table (i.e., multiple scans of the table T_i whose size is $m_i r_i$, and multiple scans of weight vector if the weight vector does not fit into main memory). With the length k_i of the longest candidate itemset, the I/O costs are given by: $k_i m_i r_i + k_i r_i$. With decentralized computation, the smaller tables may fit in main memory, which would reduce I/O costs substantially. The CPU costs for computing the frequent itemsets are the same as for the traditional centralized approach. These costs depend on the size $|c_j^i|$ of the candidate itemset for each iteration of the algorithm, the length m_i of the row that needs to be checked against the candidates, and the choice of data structures used to support the counting process.
   - *Communication costs:* If T_w is not located at the same site as T_i, the communication cost is the cost of shipping the weight vector (of size r_i) to the site where T_i is located. If T_w is at the same site as T_i, there are no communication costs involved.

4.1.2 $FI(T_i, T_{w_1} \cup T_{w_2} \cup \cdots \cup T_{w_l})$

By having a union in the second argument of the FI term, we can still use our decentralized approach of Section 2.3 with the modification explained in Section 2.4. Again, we chose the Apriori algorithm [3] to exemplify the costs, and we regard this computation also to occur in two steps:

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

   - *Local costs:* The local costs are the same as the previous case, except that the load is shared by the different locations in which $T_{w_q}, q = 1..l$ are located. Therefore, the I/O and CPU costs are given by the parameters r_{w_q} and r_i.
   - *Communication costs:* The computation of the weight vector from $T_{w_q}, q = 1..n$ is local to the site where T_{w_q} is, and therefore, there are no communication costs involved here.

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

- *Local costs:* The first step is to compose the total weight vector from the l weight vectors computed from $T_{w_1}, T_{w_2}, \cdots T_{w_l}$. This is a simple vector addition operation, and its cost depends on the number of vectors (i.e., l), and the size of the vector (i.e., r_i). Thereafter, the costs are the same as $opFI$ in the previous case.
- *Communication costs:* Since the horizontal partitions of the weight table is located in many sites, at most one of the T_{w_q} is located at the same site as T_i. Therefore, the communication cost is the cost of shipping the weight vectors (of size r_i) to the site where T_i is located.

4.1.3 $FI(T_{i_1} \cup T_{i_2} \cup \cdots \cup T_{i_n}, T_w)$

A union in the first argument of the FI expression requires us to use a distributed algorithm that synchronizes and exchanges messages at the end of each pass, and we use [5] as an example.

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

   This operation is exactly the same as in 4.1.1.

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

   - *Local costs:* Let $T_i = T_{i_1} \cup T_{i_2} \cup \cdots \cup T_{i_n}$. After each pass, the individual sites synchronize, but the amount of information processed locally (taking all sites into account) is the same as though the union had been computed in advance. The reason is that each site, at each pass, scans the local database, and counts the candidates, but the candidates are the same as the candidates for table T_i. The cost in each local site, depends on the parameters $|c_j^{i_p}|, p = 1..n$, m_{i_p} and k, which are the same for all sites, and is proportional to the number of rows present. The sum of the total number of local rows is the same as the total rows in T_i (i.e., $r_i = \sum_{p=1}^{n} r_{i_p}$), therefore, I/O and CPU costs are the same as in 4.1.1.
   - *Communication costs:* First consider the shipping of the weight vectors. The original weight vector is horizontally partitioned, and the respective vectors are sent to the respective $T_{i_q}, q = 1..n$. If it is impossible to determine in advance which partition of the vector should be sent to a particular site, the entire vector is shipped to all sites involved. Therefore, this cost depends on n, number of sites, and r_i, total number of rows of T_i (where T_i is the union of all $T_{i_p}, p = 1..n$). The communication cost should also account for the cost of shipping the candidate set counts at the end of each pass. Each site ships this information to other sites. Therefore, this part accounts for $\sum_{j=1}^{k}(|c_j^i|) * p$ in communication costs.

4.1.4 $CI(\{T_{i_1}, T_{i_2}, \cdots T_{i_n}\}, T_w)$

The first cost is the computation of the join of the tables: $T_{i_1} \bowtie T_{i_2} \bowtie \cdots \bowtie T_{i_n} \bowtie T_w$ (see join costs below). We choose one of the available algorithms from [6, 10], the I/O saving algorithm explained in Section 2.3, as an example. Let $T_i = T_{i_1} \bowtie T_{i_2} \bowtie \cdots \bowtie T_{i_n} \bowtie T_w$.

- *Local costs:* In the I/O saving approach, we process each row of the joined table as it is created, avoiding the cost of storing and re-scanning the joined table T_i. Therefore, the I/O cost is just the scanning of T_i once. For the CPU costs, for each row s in the joined table, the cost is in determining which frequent itemsets found in the individual tables T_{i_p} are present in s, so that the frequent itemsets for the joined table can be computed. Therefore, for each row in T_i, for each table $T_{i_p}, p = 1..n$, we need to check m_{i_p} against $\sum_{j=1}^{k_{i_p}} |l_j^{i_p}|$. This check is done r_i times. If indices are available, this operation can be done more efficiently, as will be discussed in Section 4.2.

- *Communication costs:* Besides the costs of sending the tables that is already included in the cost of join, the frequent itemsets found for each of the individual tables T_{i_p} need to be sent to the site where $opCI$ will be executed. This is the cost of shipping $\sum_{p=1}^{n}(\sum_{j=1}^{k_{i_p}} |l_j^{i_p}|)$ elements.

4.1.5 $CI(\{T_{i_1}, T_{i_2}, \cdots T_{i_n}\}, (T_{w_1} \cup T_{w_2} \cup \cdots \cup T_{w_l}))$

Again, we choose the I/O saving algorithm from Section 2.3, as an example. The first cost is the computation of the join of the tables: $T_{i_q} = T_{i_1} \bowtie T_{i_2} \bowtie \cdots \bowtie T_{i_n} \bowtie T_{w_q}$, for $q = 1..l$ (see join costs below).

- *Local costs:* The costs are basically the same as in 4.1.4, except that the load is distributed at the different locations. The only extra cost is the cost of adding the partial counters that were computed in each location, i.e., a matrix addition operation. Still, the I/O cost is just the scanning of each of the T_{i_q}'s once. For the CPU costs, for each row s in a joined table, the cost is in determining which frequent itemsets found in the individual tables T_{i_p} are present in s, so that the frequent itemsets for the joined table can be computed. Therefore, for each row in T_{i_q}, for each table $T_{i_p}, p = 1..n$, we need to check m_{i_p} against $\sum_{j=1}^{k_{i_p}} |l_j^{i_p}|$. This check is done r_i times total (since $r_i = \sum_{q=i}^{l}(r_{i_l})$). Finally, the matrix addition operation depends on the size of the matrix (given by $\pi_{p=1}^{n} \sum_{j=1}^{k_{i_p}} |l_j^{i_p}|$) and the number of horizontal partitions, l.

- *Communication costs:* Same communications costs as 4.1.4, except that some communication is needed in order for the final addition of matrices to be performed, i.e., the local matrix counters must be sent to a central location. This again, depends on the size of the matrix.

4.1.6 Union

In our approach, the union operation is required to compose the set of frequent itemsets, i.e., $FI \cup FI \cup \cdots \cup CI$, or to compute the tables of interest, i.e., the tables that compose the arguments of the operations FI and CI, e.g., $FI(T_1 \cup T_2, T_w)$ in case where we chose not to use a horizontal distribution algorithm such as in [5]. We discuss both situations in the following.

1. Composing the set of frequent itemsets.

   - *Local costs:* Usually, the cost of a union operation only involves costs of removal of duplicates. In our case, since the sets that participate in the union do not contain overlapping frequent itemsets (i.e., FI and CI are disjoint), the cost for the union in our case is negligible.

- *Communication costs:* The union operation compose sets of frequent itemsets computed by $opFI$ and $opCI$ operations. Every time an $opCI$ is performed, all frequent itemsets involving items from the tables for which the $opCI$ is being performed, will be sent to the site where $opCI$ is executed, and this communication cost is already accounted for in $opCI$. Furthermore, in every plan, the last operation to be executed before the union is an $opCI$ that computes the frequent itemsets involving items from every table. Therefore, all frequent itemsets are already local to the site where the final $opCI$ is performed, and so, no communication costs are involved.

2. Computing the tables in advance.

   In this case, traditional union costs from query processing and optimization literature, i.e., shipping of tables and removal of duplicates.

4.1.7 Join

Join operations could be required in CI given above. Furthermore, join operations could be required in situations where tables are joined prior to counting frequent itemsets. For example, in Section 3, $FI(T, I)$ requires the computation of table T which is a join of all the original tables. As in traditional query processing, there are many ways to perform a join. Choices are available in the ordering of the joins as well as in the use of particular join strategies; each plan leads to a different cost. Similarly, when the tables to be joined are physically distributed, semi-join strategies may be considered. Therefore, the local costs and communication costs for the join (or semi-join) are the typical costs as available in the query processing literature (e.g., see [12]).

4.1.8 Estimating the Parameters

In order to assess a more precise cost for a particular execution, parameters such as the table size, join selectivity factors, length of longest candidate itemset, and number of frequent itemsets found etc. need to be estimated. Most of the parameters needed are similar to the ones required in query processing and optimization. Also, the techniques used for estimating are similar; the cost and size of a joined table (e.g., by using a join selectivity factor) can be used in our context. While some parameters are easier to determine and maintain (e.g., table size), others may be assessed only by performing part of the DM itself (e.g., number of frequent itemsets found in a table). Another approach would be to use sampling techniques.

A major consideration is to decide whether the tables should be joined early, or be processed separately (i.e., traditional vs. decentralized mining). Also, in decentralized DM, the best strategy to merge results requires an estimate of the number of frequent itemsets. The costs, and the estimate of parameters of the tables, may be used in the same manner as used in cost-based query optimization. Note that this is applicable whether or not DM is effected on the tables as stored in a database, or loaded into a file system.

4.2 Our Example

We now illustrate how to use our cost formulas for our example in Section 2.1. Figure 2 shows some parameters for the tables. Table T represents the final table (i.e., $D \bowtie C \bowtie (B1 \cup B2) \bowtie P$). We also consider partial joins (e.g., $D \bowtie C$), since we want to assess the cost of the various different strategies. Also note that $C \bowtie P$ is essentially a cartesian product since the two tables do not have any attributes in common. However, the cartesian product may have a smaller number of rows than the final table T (this is typically the case in data warehouses, where a central fact table is several orders of magnitude larger than the smaller dimension tables).

Parameters for Tables			
Table X	r_X	m_X	k_X
D	10K	25	7
C	50K	10	4
P	5K	15	6
$B1$	1.5G	2	-
$B2$	1G	2	-
$D \bowtie C$	50K	34	6
$D \bowtie P$	50M	40	5
$C \bowtie P$	250M	25	7
$D \bowtie C \bowtie P$	250M	49	12
T	2.5G	49	12

Figure 2: Parameters for tables.

We restrict our attention, for simplicity, to the I/O costs, but similar assessment could be done for the CPU and communication costs. Also, we only consider only the FI and CI operations, and do not compute explicitly the costs of *join* and *union* which are normally involved in finding the frequent itemsets. We assume that there are bitmap indices for the tables C and P. Such an index would be $log(2.5G) = 32$ bits. Considering that the attributes of each table are 4 bytes long, each row of tables C and P will have an extra byte for holding the bitmap index. For simplicity, we can construct a bitmap index for table D, by scanning table C, and adding the bitmaps for the entries in C that correspond to the same entry in D. The cost of building this index is of the order of the size of C, and therefore, is negligible when compared to other costs. With these bitmap indices available, the operation $opWV$ only needs a scan of the index, unless there is a explicit join on the weight table. The only exception is when the weight table is $C \bowtie (B1 \cup B2)$ for computing D, in which case we will use this precomputed bitmap index for D. The use of bitmap indices, greatly simplifies the computation of the weight vector ($opWV$); for each row of the individual tables (e.g., D, C, or P) the number of "on" bits on the index needs to be counted. The cross-itemsets counting is also simplified, since a join (even though non-materialized) need not be computed. In fact, tables $B1$ and $B2$ do not need to be scanned at all. This is true for our particular example because there are no non-key attributes in tables $B1$ and $B2$.[4] For each pair of rows, take the bitwise *and* of the two bitmap indices to obtain the rows of $B1$ and $B2$ where the pair occurs together. By taking the number of "on" entries in the new index (i.e., after computing the bitwise and), we know how much to increment the support of the candidate itemsets.

[4] In [6], we show how to deal with non-key attributes in the weight table.

We now compute the I/O costs (in some appropriate cost units) for each of the expressions $(1-8)$ found in Section 3.2.

1. $FI(T, I)$

   - $opWV$: since the weight table is I (i.e., there is no weight vector to be computed), there are no I/O costs for $opWV$.
   - $opFI$: $2.5G * 49 * 12 = 1,470G$

   total cost: $1,470G$

2. $FI(D \bowtie C \bowtie P, D \bowtie C \bowtie (B1 \cup B2) \bowtie P)$

   - $opWV$: since the weight table involves joining of tables, we do not use the bitmap indices; instead, the table is scanned to count occurrences.

     $2.5G * 49 = 122.5G$
   - $opFI$: $250M * 49 * 12 = 147G$

   total cost: $269M$

3. Note that expressions 3 and 4 have the same costs as far as I/O. The difference is that the weight table requires computation of the join (which for simplicity, we do not include here). Therefore, we only describe costs for expression 4.

4. (a) $FI(D, (C \bowtie (B1 \cup B2))$

   - $opWV$: $10K$
   - $opFI$: $10K * 25 * 7 = 1.75M$

   (b) $FI(C \bowtie P, (B1 \cup B2))$

   - $opWV$: when joining tables C and P, for each new entry of $C \bowtie P$, a new bitmap is generated by computing the bitwise *and* of the two indices for each corresponding row from C and P. This new bitmap index will be of the size of the new joined table: $250M$.
   - $opFI$: $250M * 25 * 7 = 43.75G$

   (c) $CI(\{D, C \bowtie P\}, ((B1 \cup B2) \bowtie C))$

   using a nested loop for the entries of D and $C \bowtie P$, the cost would be: $(10K * (25 + 1)) + 10K * (250M * (25 + 1)) = 260K + 10K * 6.5G = 65,000G$

   since table D is probably small enough to fit into main memory, the cost is actually: $(10K * (25 + 1)) + (250M * (25 + 1)) = 260K + 6.5G = 6.5G$

   total cost: $50.25G$

5. (a) $FI(D, (C \bowtie (B1 \cup B2))$ from above: $1.82M$

   (b) $FI(C, (B1 \cup B2))$

   - $opWV$: $50K$

- $opFI$: $50K * 10 * 4 = 2M$

(c) $FI(P, (B1 \cup B2))$

- $opWV$: $5K$
- $opFI$: $5K * 15 * 6 = 450K$

(d) $CI(\{C, P\}, (B1 \cup B2))$

if P fits in memory: $(5K * (15 + 1)) + (50K * (10 + 1)) = 80K + 550K = 630K$

(e) $CI(\{D, C \bowtie P\}, ((B1 \cup B2) \bowtie C))$

from above: $6.5G$

total cost: $6.5G$

6. (a) $FI(D, (C \bowtie (B1 \cup B2))$ from above: $1.82M$

(b) $FI(C, (B1 \cup B2))$ from above: $2.05M$

(c) $FI(P, (B1 \cup B2))$ from above: $455K$

(d) $CI(\{D, C, P\}, ((B1 \cup B2) \bowtie C))$

if P and D fit in memory: $(5K*(15+1))+(10K*(25+1))+(50K*(10+1)) = 80K+260K+550K = 890K$

total cost: $5.125M$

7. (a) $FI(D \bowtie C, (B1 \cup B2))$

- $opWV$: $50K$
- $opFI$: $50K * 34 * 6 = 10.2M$

(b) $FI(P, (B1 \cup B2))$ from above: $455K$

(c) $CI(\{D \bowtie C, P\}, (B1 \cup B2)$

if P fits in memory: $(5K * (15 + 1)) + (50K * (34 + 1)) = 80K + 1750K = 1.83M$

Therefore, total cost is: $12.485M$

8. $FI((D \bowtie C \bowtie B1 \bowtie P) \cup (D \bowtie C \bowtie B2 \bowtie P), I)$

- $opWV$: no cost, since the weight table is I.
- $opFI$: $(1.5G * 49 * 12) + (1G * 49 * 12) = 1,470G$

total cost: $1,470G$

In our calculations above, notice that the different techniques differ significantly in (I/O) cost – by several orders of magnitude. This exemplifies the importance of applying optimization strategies in arriving at the best DM plan.

5 Computation Optimization

As in query optimization and processing, there are many equivalent expressions for decentralized DM, and the exhaustive enumeration and cost assessment for all possible execution plans would be prohibitively expensive. Note that the choices for DM become a multiplicative factor on the choices for evaluating the relational algebra expressions. In this section, we discuss the optimization problem and some heuristic approaches to help reduce the execution costs – in a manner similar to those used in typical query processing.

5.1 Optimization Problem

In Section 3, we provided a way of representing the mined information at particular stages. By using the equivalences from Section 3.1, we showed how to rewrite expressions in Section 3.2, and thereby provide alternative equivalent expressions. In Section 4, we detailed cost formulas which help to assess the cost of each of the mining tasks (e.g., FI, CI etc.). In order to find the best strategy, we need to enumerate different ways of mining the desired information, both at the logical level (i.e., the particular FI's and/or CI's) and at the physical level (i.e., which particular mining algorithm to use in each case). The steps involved in optimizing the task of finding frequent itemsets for a table T (where T could be a join and/or union of multiple tables) are summarized as follows:

1. Enumerate expressions equivalent to $FI(T, I)$, and let $e_i, i = 1..n$ denote them.
2. Compute the costs associated with each expression.
3. Find the expression e_i with the minimum execution costs.

Now, enumerating all possible equivalent expressions results in a large search space. Also, computing each expression e_i is an optimization problem in itself: each expression e_i is composed of a union of one or more subexpressions, and each subexpression (e.g., $FI(C, \pi_{ssn}(B1 \cup B2))$ in Section 3.2) would have different costs associated with it, depending on the mining strategy employed (e.g., Apriori [3], DIC [4]). Furthermore, each subexpression could involve a join of tables, in which case the decision of how to join these tables, and which join strategy to use, is also an aspect of optimization. However, since the total cost of expression e_i is a sum of each subexpression, by minimizing the cost of each subexpression, we find the least cost for e_i. As such, each of the subexpressions could be minimized separately, and this "monotonicity" property may be exploited in the same manner as is done in query optimization. Nonetheless, the large search space suggests a need for heuristic approaches.

Of note is the similarity to query processing and optimization. If the itemset counting is made part of the database engine, it could be associated with the usual query optimization strategies (including joins, etc), and may result in highly efficient executions. Alternatively, the DM optimization may be effected outside the database engine as an application-level issue. Even so, having some knowledge regarding the parameters would assist in allowing for higher-level optimization (i.e., the database query optimizers would be complemented by the DM level optimization). This is also the case in a distributed environment, where the distributed database level query optimizers would be augmented with our approach.

5.2 Heuristics

Besides the typical rules available in query optimization, such as performing selects early, we present some heuristics that pertain to our problem of finding frequent itemsets. In most cases, the heuristics are design to provide more efficient DM plans. For example, in some cases, it is clear that the number of frequent itemsets in one table is smaller than the number of frequent itemsets in another table (e.g., $Customer$ as compared to T in Section 4, since the set of frequent itemsets in table T includes the set of frequent itemsets of $Customer$). Such information would help decide which approach is better without knowing the specific values of the parameters.

5.2.1 Weight Table Choice

Consider the equivalence presented in Section 3, $FI(X, W) = FI(X, \pi_{id_x}(W))$. As discussed in Section 4, the computation of the weight vector from the weight table (in this case, either W or $\pi_{id_x}(W)$) might require a scan of the weight table. Therefore, it is advantageous if the weight table were to have fewer attributes. This rule is important in cases such as our example in Section 3.2 where:

$$\begin{aligned} FI(C, D \bowtie C \bowtie (B1 \cup B2) \bowtie P) &= FI(C, \pi_{ssn}(D \bowtie C \bowtie (B1 \cup B2) \bowtie P)) \\ &= FI(C, \pi_{ssn}(B1 \cup B2)) \end{aligned}$$

The additional project operation could incur an undesirable extra cost; however, since we notice that $FI(C, \pi_{ssn}(B1 \cup B2)) = FI(C, (B1 \cup B2))$, we may compute the weight vector directly from $B1$ and $B2$ without having to perform the join and project operations.

Heuristic 5.1 *Use fewest number of tables for the weight table to compute the FI's.*

5.2.2 Table with only Key Attributes

We have the following equivalence:

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

If table Y has only key attributes (i.e., attributes that are foreign keys to other tables), we argue that the expression on the right will be less expensive to evaluate. The reason is that the second and third terms on the expression on the right do not need to be evaluated because Y has only key attributes (i.e., because of the definition of the terms FI and CI, the two terms will evaluate to the empty set). Therefore, we only need to compare $FI(X \bowtie Y, W)$ with $FI(X, X \bowtie Y \bowtie W)$ to identify the less expensive approach. It is very likely that $X \bowtie Y$ will have a greater number of rows than X, in which case, the computation costs will be higher for the expression on the left due to the size of the table that needs to be scanned during the counting of itemsets.

Heuristic 5.2 *When a table that contains only key attributes participates in a join comprising the first argument of an FI, convert the expression using equivalence 2.*

5.2.3 Cartesian Products

Consider the following example:

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

Assume that tables X and Y do not have any attributes in common, and table W contains the foreign keys to both X and Y. In this case, using equivalence 3, the expression is further reduced to:

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

We note that when computing $opWV$ (i.e., computing the weight vector) for $FI(X, W)$, we can also compute the weight vector for $FI(Y, W)$ without additional disk accesses for W. And thereafter, the tables may process $opFI$ separately, which would be cheaper than computing $opFI$ on the cartesian product. A situation where we may need to process a large table would be when computing CI which, as indicated above, could be reduced considerably if indices are available (which is usually the case for a table W that contains only foreign keys).

Heuristic 5.3 *When the first argument of an FI has joins that result in a cross product, use equivalence 3 to convert the expression.*

5.2.4 Merging CI Expressions

Equivalence 2 provided in Section 3.1 is:

$$CI(\{X, Y, Z\}, W) = CI(\{X, Y \bowtie Z\}, W) \ \cup \ CI(\{Y, Z\}, W) \tag{2}$$

Since the expression on the right has two CI's, generally it will be more expensive in terms of I/O than the one on the left, since the table W may potentially be accessed for each CI operation. Only in some cases is the expression on the right to be preferred over the expression on the left – e.g., when the multidimensional array needed for the expression on the left (3 dimensions) is much larger than the 2-dimensional arrays needed for the expression on the right. Therefore, unless there is not enough memory for the higher dimensional array, we the expression on the left is to be preferred.

Heuristic 5.4 *Merge multiple CI expressions using equivalence 2 when there is sufficient memory for the required multidimensional array required for evaluating the merged CI.*

5.2.5 Horizontal Distribution

We re-consider our example in Section 2.1, and the operations manipulations on it by expression rewriting in Section 3.2. By using equivalences 1, 2 and 3, we arrived at expression 6 which had a highly decentralized computation of frequent itemsets. On the other hand, by distributing the join over the union, we arrived at expression 8, where we could no longer decentralize the computation on the individual tables. As we indicated in Section 4, the case with expression 6 is far more efficient.

As another example, if the table $Product$ had been horizontally distributed (i.e., $P = P1 \cup P2$), we may have arrived at

$$\begin{aligned} FI(T,I) &= FI(D \bowtie C \bowtie B \bowtie (P1 \cup P2), I) \\ &= FI(D, C \bowtie B) \ \cup \ FI(C,B) \ \cup \ FI((P1 \cup P2), B) \ \cup CI(\{D, C, (P1 \cup P2)\}, B) \end{aligned}$$

In this case, only table $Product$ enjoys distributed computation. The counting for tables $Customer$ and $Demographics$, and the counting for all tables in Phase II, do not change. Again, if we distribute the join over the union, we would no longer decentralize the computation of the individual tables, and therefore, the CPU and I/O costs would be the same as compared to a situation with the entire table being in a central location (except that the load would be distributed).

Heuristic 5.5 *Avoid distribution of join over unions.*

5.3 Rules Applied

We now illustrate how we could apply our heuristics to the expressions from Section 3.2. Using Heuristic 5.2, we eliminate expression 1 and 8, since $B1$ and $B2$ do not have any non-key attributes. Using Heuristic 5.5, we eliminate expression 8 (note that it is just a coincidence that the table that is horizontally partitioned happens to have only key attributes). Using Heuristic 5.1, we choose expression 4 over 3. Using Heuristic 5.3, we choose expression 5 over 4, since tables C and P do not have any attributes in common, and tables $B1$ and $B2$ have key attributes to both tables. Finally, we notice that for expression 5, the weight tables for both CI operations are essentially the same (with an extra column on the second one), and therefore, if we consider have enough memory for the 3-dimensional array, using Heuristic 5.4 we choose expression 6 over expression 5. Our heuristics do not help in choosing between expressions 6 and 7; therefore, after applying the heuristics, we are left with expressions 6 and 7:

$$\begin{aligned} FI(T,I) &= FI(D, \pi_{area}(C \bowtie (B1 \cup B2)) \ \cup \ FI(C, \pi_{ssn}(B1 \cup B2)) \ \cup \ FI(P, \pi_{pid}(B1 \cup B2)) \ \cup \\ & \quad CI(\{D, C, P\}, \pi_{area,ssn,pid}((B1 \cup B2) \bowtie C)) \end{aligned} \tag{6}$$

$$\begin{aligned} FI(T,I) &= FI(D \bowtie C, \pi_{ssn}(B1 \cup B2)) \ \cup \ FI(P, \pi_{pid}(B1 \cup B2)) \ \cup \\ & \quad CI(\{D \bowtie C, P\}, \pi_{ssn,pid}(B1 \cup B2) \end{aligned} \tag{7}$$

These two expressions happen to be the ones with the lowest I/O costs involved (as shown in Section 4.2). Now, most of our heuristics dealt with approaches for reducing I/O. Heuristic 5.5, 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 will also increase. Finally, notice that, reducing I/O also reduces the size of tables that are scanned (both in the number of rows, as well as in the number of columns). This I/O reduction also reduces CPU costs because the CPU costs are affected by the check required for each row against the candidate set (which depends on the number of columns). Therefore, although more heuristics could be developed to deal specifically with CPU costs reduction, most of our heuristics provide reduction on CPU cost measures as well.

6 Related Work and Conclusions

The problem of AR was introduced by [2]. In [3], the *Apriori* algorithm improved on the performance of the original algorithm. Since then, a number of algorithms based on [3] have been presented (e.g., [13, 14, 4, 1]). Some algorithms have been considered for new kinds of AR (e.g., [17, 8, 18]).

There has been some work on distributing the Apriori algorithm (e.g., [5]) when the database is horizontally but not vertically partitioned. That is, the database design considered is limited to one table that is horizontally partitioned in different sites, and therefore each site has the exactly the same schema. In that sense, the amount of information read and being processed is the same as in a sequential algorithm (other than the message exchanges) except that the load is distributed. However, more general types of distribution are common, and have not been examined in that work. In our previous work [10], we provided an approach that only applies to vertically partitioned tables.

Database design is an area which is likely to impact the task of data mining, considering the decentralized approach discussed in this paper. Important aspects include the way in which the data is stored, partitioned, distributed, how it relates. Work in normalization theory (e.g., see [11, 16]) as well as partitioning/allocation (e.g., see [19]) is relevant to decentralized mining of data.

There is significant work done in query processing that is related to our goal of optimizing decentralized DM. For instance, in distributed query processing, the semi-join algorithms ([12]) greatly benefit our approach when applied to physically distributed tables. In fact, as noted above, our approach of using a weight vector, is similar a semi-join based approach. General sorting and joining algorithms ([16, 7, 20]), available in the literature are also important in the stage of combining the data, either prior, as well as during different stages of the decentralized algorithms. Also, note that query processing techniques have been suggested for DM in earlier work (e.g., see [9]).

To summarize, based on our previous work [10] in basic decentralized approaches to mining data residing in multiple tables, we extended our approach to general database designs. Our approach applies to tables that are vertically and/or horizontally partitioned (as a result of normalization and data allocation strategies), and may reside in physically distributed sites. We provided a simple algebraic approach to represent and manipulate expressions that denote the mined information (which, in our case, are the frequent itemsets). Together with the cost formulae for our basic operations, our algebra provides a means to describe different processing strategies – each with a different processing cost. Therefore, our approach provides an important optimization problem, similar to those encountered in query processing, and based on our algebra, we are able to describe and exemplify heuristics to reduce the overall computation, I/O and communication costs.

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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

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

[9] T. Imielinski and H. Mannila. A database perspective on knowledge discovery. *Communications of the ACM*, 39(11):58–64, November 1996.

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

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

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

[13] 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.

[14] 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.

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

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

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

[18] 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.

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

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