Scanning once a large distributed database to mine global association rules by growing a prefix tree for each local transaction

F. Wang & N. Helian
Department of Computing, London Metropolitan University, UK

Abstract

Most of the popular data mining algorithms are designed to work for centralized data and they often do not pay attention to the resource constraints of distributed and mobile environments. In support of the third generation of data mining systems on distributed and massive data, we proposed an efficient distributed and mobile algorithm for global association rule mining, which does not need to ship all of local data to one site thereby not causing excessive network communication cost. In this algorithm the contribution from each transaction is comprehensively taken into account by growing a prefix tree for each transaction and enumerating all subsets of the transaction itemset. There is no need at all to store and re-scan the previously-scanned transactions, which will be discarded after a single pass. In each site, attributing to the integrity of the data structure designed in this algorithm, the locally enumerated itemset counters can be stored in a pair of local relational tables. The power of generating ad hoc queries in SQL ensures fast access to any desired counter. The local absolute support count for each enumerated itemset is first found. The global absolute support count for each itemset can then be determined by summing up, for each enumerated itemset, the local support of that itemset in all the distributed sites over data grid. The experiments show that this algorithm implemented in SQL beats classic Apriori algorithm for large problem sizes, by factors ranging from 2 to more than 20, and this gap grows wider when the volume of transactions further grows up.

1 Introduction and related work

Knowledge discovery and data mining deal with the problem of extracting interesting associations, classifiers, clusters, and other patterns from data.
Among the best known algorithms is the Apriori algorithm. An ever increasing number of organizations are installing large data warehouse using relational database technology. With the explosion of the commodity internet and the emergence of wide area high performance networks, mining distributed data is becoming recognized as a fundamental scientific challenge. The Internet, corporate intranets, sensor networks, and even scientific computing domains support this observation [1]. We are also witnessing increasing demands on mobile computing to provide the types of support required by a growing number of mobile workers. Such individuals require working as if in the office but in reality they are working from remote location including homes, clients’ premises, or simply while en route to remote locations. The “office” may accompany a remote worker in the form of laptops, palmtops, handhelds, embedded systems, and wearable computers, or other Internet access device. With the rapid expansion of cellular, wireless, and satellite communications, it will soon be possible for mobile users to access any data, anywhere, at any time. Advanced analysis of distributed data for extracting useful knowledge is the next natural step in the increasingly connected world of ubiquitous, distributed and mobile computing.

Concretely, data mining may be viewed as extracting a learning set from one or more (distributed) data warehouses and applying a data mining algorithm to produce a predictive model or rule set [2]. Distributed data mining can choose to move data, to move intermediate results, to move predictive models, or to move the final results of a data mining algorithm. Different strategies are possible, depending upon the data, its distribution, the resources available, and the accuracy required: 1. MD (Move Data): Ship raw data D across the network to another node for processing; 2. MM (Move Models): Process the data locally and ship the predictive model M to another node for further processing [3]; 3. MR (Move Results): Process the data locally until a result R is obtained and ship the result to another node for further processing [4].

As a motivating example, consider a problem of mining global association rules. Suppose that a large supermarket chain has a (virtual) transaction data base that is distributed among several locations. Transactions in each component database have the same format (horizontal fragmentation), namely T_j: \{i_1, ..., i_m\}, where T_j is a transaction identifier, and i_k (1≤k≤m) is the identifier of an item purchased in the transaction. The goal of this data mining query might be to understand whether bread is associated with milk (i.e., frequently purchased together by the same customer). Notice that this is much different than a traditional search which would simply return all transactions containing the keyword milk or bread. In the recent past, these types of queries would have been impossible without first moving all of the data to a central location and then analyzing it within the memory of a single workstation. Next generation broadband networks might create the possibility of moving large amounts of data.

In support of the current practice that restricts the communication cost but, if possible, without missing important rules, we proposed an efficient distributed and mobile algorithm for global association rule mining while leaving the data in
place. This provides a fundamentally new technology, since, in fact, most data is
distributed. In this article, we will first describe architecture of this distributed
and mobile data mining system. Secondly we will discuss some issues about
Oracle’s distributed DBMS functionality and PL/SQL programming
environment. Thirdly we design a new method of distributedly organizing sets of
itemset counters in relational tables. Fourthly, the proposed distributed algorithm
is implemented in Oracle PL/SQL. Fifthly, we present a performance
comparison of the Distributed ScanOnce with the classical algorithms against a
synthetic database. We will finally conclude the paper by giving four features of
this algorithm to explain why it is experimentally an order of magnitude faster
than Apriori in very large distributed databases.

![Figure 1: The architecture for a distributed and mobile data mining system
described in this article. In this system, the data mining (DM)
algorithms have been encapsulated into SQL Server stored
procedures. The intermediate results are stored back into the local
relational database and then transferred to one site for final
processing to obtain global association rules.](image)

2 Distributed and mobile data mining system

A distributed data mining system consists of a single logical database that is split
into a number of fragments, as shown in Figure 1. Each fragment is stored on
one or more computers under the control of a separate DBMS, with the
computers connected by a communication network (wiredly and wirelessly).
Each site is capable of independently processing user requests that require access
to local data (that is, each site has some degree of local autonomy) and is also capable of processing data stored on other computers in the network.

As shown in Figure 1, with mobile database, users have access to corporate data on their laptop, PDA, or other Internet access device that is required for applications at remote sites. Depending on the particular requirements of mobile applications, in some cases the user of a mobile device may log on to a corporate database server and work with data there, while in others the user may download data and work with it on a mobile device or upload data captured at the remote site to the corporate database.

3 Oracle’s distributed DBMS functionality

Like many commercial Distributed DBMSs, Oracle does not support of fragmentation mechanism, although it does support location transparency. We provide an overview of Oracle’s distributed functionality, covering:

3.1 Connectivity

Net8 is the data access application Oracle supplies to support communication between clients and servers. Net8 enables both client-server and server-server communications across any network, supporting both distributed processing and distributed DBMS capability. Net8 establishes a connection by passing connection request to the Transparent Network Substrate (TNS), which determines which server should handle the request and sends the request using the appropriate network protocol (for example, TCP/IP or SPX/IPX). The Oracle Names product stores information about the databases in a distributed environment in a single location. When an application issues a connection request, the Oracle Names repository is consulted to determine the location of the database server (Figure 1).

3.2 Global database names

Each distributed database is given a name, called the global database name, which is distinct from all databases in the system. Oracle forms a global database name by prefixing the database’s network domain name with the local database name.

3.3 Database links

The purpose of database links is to make remote data available for request and updates, in essence acting as a type of stored login to the remote database. A database link should be given the same name as the global name of the remote database it refers to, in which case database links are in essence transparent to users of a distributed database. For example, the following statement creates a database link in the local database to the remote database at London:

CREATE public database link transaction.london.co.uk
Once a database link has been created, it can be used to refer to tables and views on the remote database by appending @databaselink to the table or view name used in an SQL statement. A remote table or view can be queried or accessed with the SELECT, INSERT, UPDATE, and DELETE statements.

3.4 Heterogeneous distributed databases

In an Oracle heterogeneous distributed DBMS at least one of the DBMS is a non-Oracle system. Using Heterogeneous Services and a non-Oracle system-specific Heterogeneous Services agent, Oracle can hide the distribution and heterogeneity from the users. The Heterogeneous Services agent communicates with the non-Oracle system, and with the Heterogeneous Services component in the Oracle server. On behalf of the Oracle server, the agent executes SQL, procedure, and transactional requests at the non-Oracle system.

4 Data transformation and distributed management of the itemset counter sets

PL/SQL is Oracle’s procedural extension to SQL. SQL is good in defining the structure of the database and generating ad hoc queries. However to build applications, the power of a full-fledged high-level programming language is needed. PL/SQL provides such an environment to develop application programs. It supplements SQL with several high-level programming language features such as object-oriented features [5]. Thus, PL/SQL can be used to build sophisticated database applications, like the implementation of our proposed algorithm in this article.

4.1 Data transformation

Since we are implementing our Distributed ScanOnce algorithm in PL/SQL, a Boolean database D (Table 1) needs to be transformed to relational representation. Table 2 shows the common relational representation of Boolean transaction data in Table 1. Each transaction and each item is uniquely identified by an integer (for ease of demonstration, we use alphabetic letters to identify different items in the article). The two column representation is implied by the set characteristic of the transactions. That is, the number of items typically contained in a transaction may vary largely. Moreover, a maximal transaction size may not be determined in advance.

4.2 Local database access using cursors and records

PL/SQL provides cursors for processing a query resulting in more than one row. A PL/SQL cursor allows the program to fetch and process information from the database into the PL/SQL program. The relational transaction table (Table 2) should be loaded into a predefined cursor, as shown in Figure 2, one transaction (consisting of a number of rows) at a time. Cursor variables provide a pointer to the cursor work area and thereby increase the flexibility of the use of cursors.
Once a cursor has been declared, it can be processed using the open, fetch, and close statements. PL/SQL provides for loop to be used with cursors. This loop is very useful in our situation where all rows of the cursor (one transaction) are to be processed.

Table 1: Boolean transaction database D.

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Item a</th>
<th>Item b</th>
<th>Item c</th>
<th>Item d</th>
<th>Item e</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2: Relational representation of D.

<table>
<thead>
<tr>
<th>TID</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
</tr>
<tr>
<td>1</td>
<td>e</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td>2</td>
<td>e</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>6</td>
<td>b</td>
</tr>
<tr>
<td>6</td>
<td>c</td>
</tr>
<tr>
<td>6</td>
<td>d</td>
</tr>
</tbody>
</table>

A PL/SQL record is a composite data structure, similar to a structure in a high-level programming language. In Figure 2, a record is used, whose structure is based on the select-list of a cursor. The record into which the cursor rows are to be fetched must be compatible with the cursor row type. The below coding list (Figure 3) declares a cursor-based record and then processes the cursor using the record. Within the body of the loop, the individual fields of the record can be
accessed. The loop terminates automatically when all rows of the cursor, i.e., the current transaction, have been scanned.

```
declare
cursor ta_line is --cursor definition
    select item
    from transaction ta
    where tid=i; --i is a loop variable for scanning
    --the entire transaction database
ta_rec ta_line%rowtype; --record definition
begin
    open ta_line;
    loop
        fetch ta_line into ta_rec;
        exit when ta_line%notfound;
        Distributed ScanOnce Algorithm in PL/SQL (see Figure 6)
    end loop;
    close ta_line;
end;
```

Figure 3: Declaration of a cursor-based record and then processing of the cursor using the record.

### 4.3 Distributed management of the itemset counter sets

After the whole local database is scanned (See Section 5 for details of Distributed ScanOnce Algorithm), the counters for the all enumerated itemsets, organized in a group of itemset counter sets, is obtained, as shown in the left part of Figure 4, which is the contribution from an exemplified transaction 'abde' consisting of four singleton items ‘a’, ‘b’, ‘d’ and ‘e’. Each entry in the set is in the form (IS, f), where IS represents an itemset and f represents the exact frequency count of IS. Organizing the counters in such sets not only allows us to
store them efficiently (using little memory), but also supports generating the rules. The infrequent itemsets will be eventually pruned after summing up the support across all sites since they do not have minimum support.

<table>
<thead>
<tr>
<th>C_no</th>
<th>C_sub</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>1</td>
<td>c</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
</tr>
</tbody>
</table>

Support_count table for 4-itemset

<table>
<thead>
<tr>
<th>C_no</th>
<th>C_Sup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C_no</th>
<th>C_sub</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
</tr>
</tbody>
</table>

Support_count table for 3-itemset

<table>
<thead>
<tr>
<th>C_no</th>
<th>C_Sup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C_no</th>
<th>C_sub</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>d</td>
</tr>
</tbody>
</table>

Support_count table for 2-itemset

<table>
<thead>
<tr>
<th>C_no</th>
<th>C_Sup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C_no</th>
<th>C_sub</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
</tr>
</tbody>
</table>

Support_count table for 1-itemset

<table>
<thead>
<tr>
<th>C_no</th>
<th>C_Sup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 4: The relational representation of the sets of the itemset counters in each local site.

How to store these sets efficiently? Different from traditional linked-list tree representation [6], we use very simple relational tables to store these sets. As shown in Figure 4, we push the counter set into a relational table “Counter”, which records the composition of each itemset counter. Each counter is uniquely identified by an integer C_no. Such a relational representation of an itemset counter is flexible as the number of item identifiers in a counter may vary largely. Moreover, a maximal number of the counters may not be determined in advance. Another relational table “Support_Count” is also locally maintained for storing support count for each counter. The counter identifier (C_no) links “Counter” and “Support_Count” tables.
To locate a desired counter becomes another important issue. Not only in the first step of itemset appearance counting in each local site, but also in the second step of generating association rules from globally frequent itemsets, there are frequent accesses to those counters. The power of generating ad hoc queries in PL/SQL ensures fast access to any desired counter. The below coding list (Figure 5) shows an example of locating a desired counter corresponding to 3-itemset ‘abd’. An intermediate variable “pass”, declared as support.count%type, is used to transfer the previous count number from the “Support_Count” table, which will be updated by a new count number.

\[
\text{Declaration}
\]

\[
\text{... (see Figure 3 for details)}
\]

\[
\begin{align*}
\text{select s.count} \\
\text{into pass} & \quad //\text{pass is a predefined intermediate variable} \\
\text{from support s} \\
\text{where s.c_no=} & \quad (\text{select c1.}}c \text{c_no} \\
\text{select c1.}}c \text{c_sub='a' and c2.}}c \text{c_sub='b' and c3.}}c \text{c_sub='d'} \\
\text{and c1.}}c \text{c_no=}}c \text{2.}}c \text{no and c2.}}c \text{no=}}c \text{3.}}c \text{no} \\
\text{and c1.}}c \text{no in} & \quad (\text{select distinct c.}}c \text{no} \\
\text{from counter c} \\
\text{group by c.}}c \text{no} \\
\text{having count(}=3)); & \quad //\text{assume 3 is minimum support}
\end{align*}
\]

Figure 5: An example of locating a desired counter corresponding to 3-itemset ‘abd’.

5 Distributed ScanOnce algorithm in PL/SQL

The distributed data mining algorithms are encapsulated into SQL Server stored procedures. The algorithm is outlined as below. In each site, the local absolute support count for each enumerated itemset is found. These intermediate results are stored back into the local relational database and then transferred to one site for final processing. The global absolute support count for each itemset in the union CF of all of the local itemsets across all the distributed sites can be determined by summing up, for each enumerated itemset, the local support of that itemset in all the distributed sites (Move Result via network). Doing this for each itemset in CF will give us their global supports. Itemsets whose global supports pass the support threshold are globally frequent itemsets. Finally, strong association rules are derived from the globally frequent itemsets.

In each site, the Distributed ScanOnce algorithm is described in Figure 6. This is a purely sequential (rather than recursive like Apriori) counting procedure which is well compatible with the relational representation. To count a certain
transaction (represented by a PL/SQL cursor), we merely start at the first row (item) in the cursor and then sequentially traverse the cursor by following the pointer as indicated in Figure 2. Our algorithm is described in the pseudo-code in Figure 6, where \( N \) denotes the number of transactions in the database and \( T \) the transaction being currently scanned. Our data structure \( A \) is a set of entries of the form \((IS, f)\), where \( IS \) is an itemset enumerated from the current transaction and \( f \) is an integer representing its frequency. Initially, \( A \) is empty. The contribution from each transaction is comprehensively taken into account by growing a prefix tree for each transaction and enumerating all subsets of the transaction itemset, as shown in Figure 7.

... 

1. \( A \leftarrow \emptyset \) //\( A \): The set of all counters
2. \( T \leftarrow \) next transaction //\( T \): Transaction
   //\( \{\text{item-IDs}\} \)
3. Grow subset tree for \( T \) and enumerate all subsets of the current transaction \( T \) (Figure 2)
4. \( IS \leftarrow \) each subset
5. if \((IS, f)\) exists in \( A \) do //\( f \): frequency
6. \( f \leftarrow f+1 \)
7. else do
8. insert \((IS, 1)\) to \( A \)
9. endif
10. Goto 2
11.
12. scan \( A \) and prune infrequent itemsets
13. if \( f \geq \min \_sup \times N \) //\( \min \_sup \): minimum //support, \( N \): Number of transactions
14. output \((IS, f)\)
15. endif
16.
17. Generate rules from frequent itemsets \( IS \) satisfying minimum confidence \( c \)
   specified by the user

... 

Figure 6: Pseudo-code for our Distributed ScanOnce algorithm described in this article.

Figure 7 shows the enumerating procedure of all subsets of an exemplified transaction "abde" (Note missing 'c') by growing a prefix tree in a local site. In order to find the frequent itemsets, we have to count the transactions they are contained in. Our implementation is based on the idea to organize the counters for the itemsets in a special kind of prefix tree for enumeration. The structure of a itemset tree as our implementation uses it is shown in Figure 6. Each \( IS \) denotes a counter for an itemset \( IS \). A node in the tree represents an itemset consisting of Item-IDs in that node and all its ancestors, as underlined in the figure. The itemsets enclosed in a dashed rectangle share the same ancestor. For
ease of illustration, symbolic letters are used here to represent the items. In practice they should be 4-byte integers. Since the common part would be a prefix if we were dealing with sequences instead of sets, such a data structure is commonly called a prefix tree. That we are dealing with sets, not sequences, is the reason, why this tree structure is unbalanced: ‘abd’, for instance, is the same as ‘bda’ and therefore only one of these counters is needed. This full prefix tree is created level by level. That is, the root node is created first. Then the second tree level is created—the children of the root and so on. Of course, in doing so, some branches of the tree can be pruned eventually after the whole database has been scanned, because by simply applying a user-specified threshold we can find out whether a branch can contain frequent itemsets or not.

Whenever a new itemset $IS$ arrives, we first lookup $A$, to see whether an entry for $IS$ already exists or not. If the lookup succeeds, we update the entry by incrementing its frequency $f$ by one. Otherwise, we create a new entry of the form $(IS, f)$. For an entry $(IS, f)$, $f$ represents the exact frequency count of $IS$ ever since this entry was inserted into $A$.

The enumerated $k$-itemsets will be written into their corresponding $k$-itemset counter set, together with their support count in the form of $(IS, f)$ [not shown in the figure]. As mentioned in Section 4, we push the counter set into a relational table “Counter”, which records the composition of each itemset counter. Another relational table “Support_Count” is also locally maintained for storing support count for each counter. The power of generating ad hoc queries in PL/SQL ensures fast access to any desired counter.
In summary, we process the input data stream transaction by transaction in each local site. This is 100% sequential counting procedure and therefore there is no need at all to store and re-scan the previously-scanned transactions, which will be discarded after a single pass. In [7], they try to fill available main memory with as many transactions as possible, and then process recursively such a batch of transactions together. This is where our algorithm differs from that one. The amount of main memory available can be devoted to itemset counters. Their compact data structure ensures fast access to any counter in the set.

The above is so-called first step of association rule mining, in which the frequent itemsets are determined in each local site. The second step of generating global association rules from the frequent itemsets from the globally frequent itemsets after summing up the support across all sites is straightforward. Note that there is no need to re-scan the original transaction database (re-connect the network) any longer as the counters organized in relational tables have retained sufficient information for rule generating. In other words these tables are equivalent to the provided transaction database in terms of finding the frequent itemsets. In contrast, the classic Apriori algorithm requires repeated scans of the databases[8-11] thereby resulting in unrealistically heavy network accesses particularly when considering large candidate sets.

6 Experiment results and further analysis

The distributed ScanOnce association rule mining algorithm in PL/SQL described in this article is designed to economize mining efficiency and communication overhead, and we must show that this overriding concern for speed is compatible with a reasonable utilization of computer network. Our experiment with association rule mining algorithm is based on a simulation program coded in Oracle PL/SQL. The program runs on a variety of platforms, as illustrated in Figure 1.

Three desktop computers, one laptop computer and one pocket computer were used for the experiments. The desktop computers were Intel Pentium machine with Windows 2000 Advanced Server operating system and Oracle9i enterprise edition. The CPU frequency was 1.7 GHz. The desktop computers were connected by the 10 Mbps MAN Network. The laptop computer and pocket computer were used as mobile database platforms (see Figure 1 for reference). Both of these computers were equipped with Wireless LAN cards (11 Mbps). The laptop (1.1 MHz CPU, 512MB RAM) was loaded with Oracle9i enterprise edition and Microsoft Access XP whereas the pocket computer (Compaq iPAQ Pocket PC H3970, Windows CE 2.0, 400 MHz CPU, 64 MB RAM, 288MB Flash ROM) was loaded with Wireless Database 4.0 (KelBran Software) [12]. Wireless Database supports a subset of the standard database SQL language. This feature allows us to access large databases (up to 1GB data) remotely and create our own query for distributed data mining with the SQL query wizard.

The experiments were conducted on a synthetic database, generated using the procedure described in [9-10]. In this data set, the maximum transaction size and average transaction size are set to 15 and 10, respectively. The number of
transactions in a single site ranges from 200,000 to 5 millions, which occupies up to 450 MB space. The experiments compared total mining time of distributed mining (Move Result) versus centralized mining (Move Data). For completeness we included the mining time on the mobile platform (see Figure 1 for reference). The total mining time is the time it takes to transfer any data (communication), build/transfer models, and score the validation set back at the control workstation. As shown in Figure 8, our preliminary results indicate that our PL/SQL implementation of our distributed ScanOnce algorithm is much faster than Apriori mining. Furthermore, distributed ScanOnce scales much better than Apriori. This is mainly because the wasteful operations of unnecessarily rescanning those previously-scanned subsets have been avoided by this new algorithm. It must be cautioned that error rates for distributed mining might be highly dependant on the organization of the data set and/or other factors. More extensive tests are currently under way.

An objective of a distributed data mining systems is to minimize both the volume of data transmitted over the network and the number of network transmissions. The time taken to send a message depends upon the length of the message and the type of network being used. It can be estimated using the formula:

\[ \text{Communication Time} = C_0 + \left( \frac{\text{no\_of\_bits\_in\_message}}{\text{transmission\_rate}} \right) \]

where \( C_0 \) is a fixed cost of initiating a message from one site to another, known as the access delay. In our instance, using an access delay of 1 second and a transmission rate of 10Mbps, we can calculate the time to send 1,000,000 transactions, each consisting of 720 bits (in average) if stored in a relational table (Table 2), as: Communication Time = 1 + (10^9 x 720/10^7) = 73 seconds. Consider a simplified schema consisting of the following three transaction relations: 5,000,000 transactions stored in Site 1, 5,000,000 transactions stored in Site 2,
5,000,000 transactions stored in Site 3. Assume that the data mining computation time is negligible compared with the communication time. We give two possible strategies for association rule mining: the classic Apriori algorithm and the distributed ScanOnce algorithm proposed in this article. We calculate the response times for these two strategies as follows:

Apriori: As mentioned early, Apriori algorithm requires repeated scans of the databases and so the realistic way to handle the situation is to ship all of the data to one site. Move the transaction relation from Site 2 and Site 3 to Site 1, respectively, and then process mining there:

\[
\text{Time} = 1 + 2 \times (5 \times 10^6 \times 720/10^7) = 721 \text{ seconds} \quad (1)
\]

Distributed ScanOnce: The local absolute support count for each itemset is found in each site respectively. These intermediate results are stored back into the support count table (Figure 4) in local relational database and then transferred to London for final processing to obtain global frequent itemsets.

\[
\text{Time} = 1 + 2 \times (1 \times 10^6 \times 112/10^7) = 23.4 \text{ seconds} \quad (2)
\]

where each counter record in support count table (Figure 4) occupies 112 bits.

The estimated response times vary across a wide range, in agreement with the experimental results considering the actual mining time, yet each strategy is a legitimate way to mine the data. Clearly, the communication time is significantly longer transferring the transaction databases themselves because of their big size. If the wrong strategy is chosen, then the effect can be devastating on system performance.

7 Conclusions

Most of the popular data mining algorithms are designed to work for centralized data [13-17] and they often do not pay attention to the resource constraints of distributed and mobile environments. In support of the third generation of data mining systems on distributed and massive data, we proposed an efficient distributed and mobile algorithm for global association rule mining, which does not need to ship all of data to one site thereby not causing excessive network communication cost. Our preliminary results indicate that our PL/SQL implementation of our distributed ScanOnce algorithm is much faster than Apriori mining.

However, it is not without drawback with this new algorithm. In the first step of finding frequent itemsets, we even count those itemset which are not globally frequent although they will be pruned eventually. Based on the observation that if any given set of attributes S is not adequately supported, any superset of S will also not be adequately supported and consequently any effort to calculate the support for such supersets is wasted. However, considering the advantage of performance improvement brought by avoiding shipping all of the data to one site, this new algorithm presents us with a broad range of trade-offs based on
speed requirement and storage requirement, particularly in dealing with a huge distributed database of short transactions. It is worth mentioning that fewer items will be purchased at the same time in real life [18]. The experiments show that this Distributed ScanOnce algorithm in PL/SQL beats classic Apriori algorithm, which requires repeated scans of the databases thereby shipping all of the data to one site and consequently causing excessive network communication overhead, for large problem sizes, by factors ranging from 2 to more than 20. As the volume of transactions (preferably the depth rather than the width) grows up further, the difference between the two methods becomes larger and larger.

References


