Developing Tightly-Coupled Data Mining Applications on a Relational Database System

Rakesh Agrawal and **Kyuseok Shim**
IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120

Abstract

We present a methodology for tightly coupling data mining applications to database systems to build high-performance applications, without requiring any change to the database software.

Introduction

Most of the current data mining applications have a loose connection with databases. A majority of them treat database simply as a container from which data is extracted to populate main memory data structures before the main execution begins. This approach limits the amount of data the application can handle effectively.

The more database-aware applications use *loosely-coupled* SQL to fetch data records as needed by the mining algorithm. The front-end of the application is implemented in a host programming language, with embedded SQL statements in it. The application uses a SQL `select` statement to retrieve the set of records of interest from the database. A loop in the application program copies records in the result set one-by-one from the database address space to the application address space, where computation is performed on them. This approach has two performance problems: i) copying of records from the database address space to the application address space, and ii) process context switching for each record retrieved, which is costly in a database system built on top of an operating system such as UNIX. The resultant poor performance is often the deterrent in using databases in these applications.

We present a methodology for *tightly-coupled* integration of data mining applications with a relational database system. Instead of bringing the records of database into the application program, we selectively push parts of the application program that perform computation on retrieved records into the database system, thus avoiding the performance degradation cited above. Our approach is based on a novel way of using the user-defined functions in SQL statements. A major attraction of our methodology is that it does not require changes to the database software. We validated our methodology by tightly-coupling the problem of mining association rules (Agrawal, Imielinski, & Swami 1993) on IBM DB2/CS relational database system (Chamberlin 1996). Empirical evaluation using real-life data shows nearly two-fold performance advantage for tight-coupling over loose-coupling. The programming effort in converting the loosely-coupled application to a tightly-coupled one was minimal.

Related Work The idea of realizing performance gains by executing user-specified computations within the database system rather than in the applications has manifested in several systems. Research in database programming languages, object-oriented database systems, and the integration of abstract data types in relational systems has been partially driven by the same motivation. Stored procedures in commercial database products have been designed for the same purpose. For example, Oracle provides a facility to create and store procedures written in PL/SQL as named objects in the database to reduce the amount of information sent over a network. Alternatively, an application can send an unnamed PL/SQL block to the server, which in turn complies the block and executes it. The Illustra DBMS also provides a facility for user-defined aggregations to be performed within the DBMS.

Methodology

Our methodology for developing tightly-coupled applications has the following components:

- Employ two classes of user-defined functions:
  - those that are executed a few times (usually once) independent of the number of records in the table;
  - those that are executed once for each selected record.

The former are used for allocating and deallocating work-area in the address space of the database system and copying results from the database address to the application address space. The latter do computations on the selected records in the database address space, using the work-area allocated earlier.

- To execute a user-defined function once, reference it in the `select` list of a SQL `select` statement over a one-record table. In DB2/CS, create this temporary one-record dynamically by using the construct *(value(1))* `as` *onerecord* in the `from` clause.
- To execute a user-defined function *udf()* once for each selected record without ping-ponging between the database and application address spaces, have the function return 0. Define the SQL `select` statement over the table whose records are to be processed, and add a condition of the form *udf() = 1* in the `where` clause. If there are other conditions in the `where` clause, those conditions must be evaluated first because the user-defined function must be applied only on the selected records.
- If a computation involves using user-defined functions in multiple SQL select statements, they share data-structures by creating handles in the work-area initially created.

Specifically, our approach consists of the following steps:

- Allocate work-area in the database address space utilizing a user-defined function in a SQL `select` statement over a one-record table. A handle to this work-area is returned in the application address space using the `into` clause.
- Setup iteration over the table containing data records and reference the user-defined function encapsulating the desired computation in the `where` clause of the `select` statement as discussed above. Pass the handle to the work-area as an input argument to this user-defined function. If the computation requires more than one user-defined function (and hence multiple `select` statements), have the previous one leave a handle to the desired data structures in the work-area.
- Copy the results from the work-area in the database address space into the application address space using another user-defined function in a SQL `select` statement over a one-record table.
- Use another user-defined function over a one-record table in a SQL `select` statement to deallocate the work-area.

We can cast our approach in the object-oriented programming paradigm. We can think of the function to allocate space as a constructor for an object whose data members store the state of the application program in the address space of database system. A collection of member functions generate, save, and query the state of the application program. The function to deallocate space can be thought of as the destructor for the object.

A Case Study

To validate our methodology, we tightly-coupled the data mining application of discovering association rules to IBM DB2/CS. Given a set of transactions, where each transaction is a set of items, an association rule is an expression of the from $X \Longrightarrow Y$, where X and Y are sets of items. An example of an association rule is: "30% of transactions that contain beer also contain diapers; 2% of all transactions contain both of these items". Here 30% is called the *confidence* of the rule, and 2% the *support* of the rule. The problem is to find all association rules that satisfy user-specified minimum support and minimum confidence constraints.

A transaction is represented as a set of consecutive records in the database. A record consists of a transaction id and an item id; all the items belonging to the same transaction id represent a transaction. The input data comes naturally sorted by transaction id.

Overview of the Apriori Algorithm

Our case study uses the Apriori algorithm for mining association rules (Agrawal & Srikant 1994). The problem of mining association rules is decomposed into two subproblems: i) find all *frequent* itemsets that occur in a specified minimum number of transaction, called *min-support*; ii) use the frequent itemsets to generate the desired rules. We only consider the first subproblem as the database is only accessed during this phase.

The Apriori algorithm for finding all frequent itemsets is given in Figure 1. It makes multiple passes over the database. In the first pass, the algorithm simply counts item occurrences to determine the frequent 1-itemsets (itemsets with 1 item). A subsequent pass, say pass k, consists of two phases. First, the frequent itemsets L_{k-1} (the set of all frequent $(k-1)$-itemsets) found in the $(k-1)$th pass are used to generate the candidate itemsets C_k, using the *apriori-gen()* function. This function first joins L_{k-1} with L_{k-1}, the joining condition being that the lexicographically ordered first $k-2$ items are the same. Next, it deletes all those itemsets from the join result who have some $(k-1)$-subset that is not in L_{k-1}, yielding C_k. For example, let L_3 be {{1 2 3}, {1 2 4}, {1 3 4}, {1 3 5},

```
procedure AprioriAlg()
begin
1.  $L_1$ := {frequent 1-itemsets};
2.  for ( $k$ := 2; $L_{k-1} \neq \emptyset$; $k$++ ) do {
3.     $C_k$ := apriori-gen($L_{k-1}$);  // New candidates
4.     forall transactions $t \in \mathcal{D}$ do {
5.        forall candidates $c \in C_k$ contained in $t$ do
6.           $c$.count++;
7.     }
8.     $L_k$ := $\{c \in C_k \mid c.\text{count} \geq \text{min-support}\}$
9.  }
10. Answer := $\bigcup_k L_k$;
end
```

Figure 1: Apriori Algorithm

{2 3 4}}. After the join step, C_4 will be {{1 2 3 4}, {1 3 4 5} }. The prune step will delete the itemset {1 3 4 5} because the itemset {1 4 5} is not in L_3. We will then be left with only {1 2 3 4} in C_4.

The algorithm now scans the database. For each transaction, it determines which of the candidates in C_k are contained in the transaction using a hash-tree data structure and increments their count. At the end of the pass, C_k is examined to determine which of the candidates are frequent, yielding L_k. The algorithm terminates when L_k becomes empty.

Loosely Coupled Integration

Figure 2 shows the sketch of a loosely-coupled implementation of the Apriori algorithm. Lines 4 through 13 determine the frequent 1-itemsets corresponding to line 1 in Figure 1. We open a cursor over the *sales* table, fetch one record at a time from the database to the application program, and increment count for items found in each record. The count array is maintained in the application program. Note that there is one context switch for every record in the *sales* table. At the end of the loop, the count array is scanned to determine the frequent 1-itemsets.

Lines 14 through 33 contain processing for subsequent passes. These lines correspond to lines 2 through 9 in Figure 1. In line 15, we generate candidates in the application program. The database is now scanned to determine the count for each of the candidates. We open a cursor over the *sales* table and fetch one record at a time from the database process to the application process. After all the records corresponding to a transaction have been retrieved, we determine which of the candidates are contained in the transaction and increment their counts. Finally, we determine in the application which of the candidates are frequent.

```
procedure LoosleyCoupledApriori() :
begin
1.  exec sql connect to database;
2.  exec sql declare cur cursor for
          select TID, ITEMID from sales
          for read only;
3.  exec sql open cur;
4.  notDone := true;
5.  while notDone do {
6.     exec sql fetch cur into :tid, :itemid;
7.     if (sqlcode $\neq$ endOfRec) then
8.        update counts for each itemid;
9.     else
10.       notDone := false
11. }
12. exec sql close cur;
13. $L_1$ := {frequent 1-itemsets};
14. for ( $k$ := 2; $L_{k-1} \neq \emptyset$; $k$++ ) do {
15.    $C_k$ := apriori-gen($L_{k-1}$);  // New candidates
16.    exec sql open cur;
17.    $t := \emptyset$; prevTid := $-1$; notDone := true;
18.    while notDone do {
19.       exec sql fetch cur into :tid, :itemid;
20        if (sqlcode $\neq$ endOfRec then) {
21.          if (tid $\neq$ prevTid and $t \neq \emptyset$) then {
22.             forall candidates $c \in C_k$ contained in $t$ do
23.                $c$.count++;
24.             $t := \emptyset$; prevTid := tid;
25.          }
26.          $t := t \cup$ itemid
27.       }
28.       else
29.          notDone := false;
30.    }
31.    exec sql close cur;
32.    $L_k$ := $\{c \in C_k \mid c.\text{count} \geq \text{min-support}\}$
33. }
34. Answer := $\bigcup_k L_k$;
end
```

Figure 2: Loosely-coupled Apriori Algorithm

Tightly Coupled Integration

We give a tightly-coupled implementation of the Apriori algorithm in Figure 3 using our methodology. The statement in line 2 creates work-area in the database address space for intermediate results. The handle to this work-area is returned in the host variable *blob*. The statement in line 3 iterates over all the records in the database. However, by making the user-defined function *GenL*$_1$*()* always return 0, we force the function *GenL*$_1$*()* to be executed in the database process for every record, avoiding copying and context switching. Line 3 corresponds to the first pass of the algorithm in which frequency of each item is counted and 1-frequent itemsets are determined. *GenL*$_1$*()* receives the handle for the work-area as an input argument and it saves a handle to the 1-frequent itemsets in the work-area

```
Procedure TightlyCoupledApriori() :
begin
1.  exec sql connect to database;
2.  exec sql select allocSpace() into :blob
        from onerecord;
3.  exec sql select *
        from sales
        where Gen$L_1$(:blob, TID, ITEMID) = 1;
4.  notDone := true;
5.  while notDone do {
6.      exec sql select aprioriGen(:blob) into :blob
            from onerecord;
7.      exec sql select *
            from sales
            where itemCount(:blob, TID, ITEMID) = 1;
8.      exec sql select Gen$L_k$(:blob) into :notDone
            from onerecord;
9.  }
10. exec sql select getResult(:blob) into :resultBlob
        from onerecord;
11. exec sql select deallocSpace(:blob)
        from onerecord;
12. Compute Answer using resultBlob;
end
```

Figure 3: Tightly-coupled Apriori Algorithm

before it returns.

Lines 4 through 9 correspond to subsequent passes. First the candidates are generated in the address space of the database process by the the user-defined function *aprioriGen()*. We accomplish this by referencing this function in the `select` list of the SQL statement over *onerecord* table (hence ensuring that it is executed once) and providing the handle to the frequent itemsets needed for generating candidates as input argument to the function. The handle to candidates generated is saved in the work-area.

Statement on line 7 iterates over the database. Again, by making the function *itemCount()* return 0, we ensure that this function is applied to each record, but within the database process. Handle to the candidates is available in the work-area provided as input argument to *itemCount()* and this function counts the the support of candidates. This statement corresponds to the statements in line 16-31 in Figure 2.

Next, the function *GenL_k()* is invoked in the address space of the database process by referencing it in the SQL statement in line 9 over *onerecord* table. In the kth pass, this function generates frequent itemsets with k items and returns a boolean to indicate whether the size of current L_k is empty or not. This value is copied into the host variable *notDone* to determine loop termination in the application program. After the loop exits, the function *getResult()* copies out the result from the database process into the host variable *resultBlob* in the application process. Finally, the function *deallocSpace()* frees up the work-area in the database address space.

Performance

To assess the effectiveness of our approach, we empirically compared the performance of tightly-coupled and loosely-coupled implementations of the Apriori algorithm. Six real-life customer datasets were used in the experiment. These datasets were obtained from department stores, supermarkets, and mail-order companies. We observed that in all cases, tight-coupling gives more than two fold performance advantage over loose-coupling. See (Agrawal & Shim 1995) for details of the performance experiments and results. We would like to mention that work is underway to improve the performance of the implementation of the user-defined functions in DB2/CS. The tightly-coupled implementation would directly benefit from any performance gains from this effort.

Acknowledgments Experiments by Andreas Arning, Toni Bollinger, and Ramakrishnan Srikant brought to our attention the performance penalty of loosely-coupled integration. John McPherson, Pat Selinger, and Don Haderle pointed us to the user-defined functions as a possible way of attacking the performance problem. Don Chamberlin, Guy Lohman, Hamid Pirahesh, Berthold Reinwald, Amit Somani, and Geroge Wilson explained several subtleties of the user-defined functions and stored procedures in DB2/CS. Bob Yost helped us in obtaining the latest versions of DB2/CS. Finally, the generous help and suggestions of Ramakrishnan Srikant were invaluable.

References

Agrawal, R., and Shim, K. 1995. Developing tightly-coupled applications on IBM DB2/CS relational database system: Methodology and experience. Research Report RJ 10005 (89094), IBM Almaden Research Center, San Jose, California. Available from `http://www.almaden.ibm.com/cs/quest`.

Agrawal, R., and Srikant, R. 1994. Fast Algorithms for Mining Association Rules. In *Proc. of the 20th Int'l Conference on Very Large Databases.*

Agrawal, R.; Imielinski, T.; and Swami, A. 1993. Mining association rules between sets of items in large databases. In *Proc. of the ACM SIGMOD Conference on Management of Data*, 207–216.

Chamberlin, D. 1996. *Using the New DB2: IBM's Object-Relational Database System.* Morgan Kaufmann.