RJ 10005 (01/24/96)
Computer Science/Mathematics

IBM Research Report

Developing Tightly-Coupled Applications on IBM DB2/CS Relational Database System: Methodology and Experience

Rakesh Agrawal, Kyuseok Shim

IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties).

Research Division
Almaden · T.J. Watson · Tokyo · Zurich

Developing Tightly-Coupled Applications on IBM DB2/CS Relational Database System: Methodology and Experience

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

ABSTRACT: We present a methodology for tightly coupling applications to the IBM DB2/CS relational database system to build high-performance data-intensive applications, without requiring any change to the DB2/CS software. Our technique utilizes user-defined functions in SQL statements in a novel way to selectively push parts of the application that access data records and perform computations on them into the database systems. We thus avoid one-at-a-time record retrieval from the database address space to the application address space, saving both copying and process context switching costs for each record. Our case study of tightly integrating a data mining application with the DB2/CS system shows that our approach resulted in nearly two-fold improvement in the application performance, while changes to the existing application code were minimal.

1. Introduction

The relational query language SQL lacks computational completeness to express the non-data-manipulation parts of an application. The current paradigm in developing database applications is to use *loosely-coupled* SQL with some general-purpose host programming language. The front-end of the application is implemented in the host programming language and SQL statements are embedded 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 on a database system built on top of the UNIX operating system.

The on-line transaction processing applications in which relational databases have been used with great success typically retrieve a small amount of data. Therefore, the loose-coupling does not cause severe performance problem. Relational databases are now being increasingly used in decision-support applications such as data mining and data warehouse applications [5] [13]. These applications typically examine a large portion of database and perform complex computations to analyze the retrieved data. Poor performance is often the deterrent in using relational databases in these applications.

We present a methodology for *tightly-coupled* integration of application programs with the IBM DB2/CS 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 [8]. A major attraction of our methodology is that it does not require changes to the DB2 software. We validated our methodology by tightly-coupling a data mining application—discovering association rules [1]. We report on this experience and present performance results using real-life data that show 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 server rather than in the applications has manifested in the past in several systems. Works on database programming languages [4], object-oriented database systems [3], and the integration of abstract data types in relational systems (e.g. [9] [12]) have been partially driven by the same motivation. Stored procedures in commercial database products have been designed for the same purpose. For example, Oracle [11] 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 (earlier called Montage DBMS [10]) also provides a facility for user-defined aggregations to be performed within the DBMS.

The contribution of this paper lies in providing a methodology to those interested in building high-performance data-intensive decision-support applications using the DB2/CS database system. We also provide a case study of building a data mining application using this methodology and measurements from this application with quantified performance gains. This application is now being field-deployed. DB2/CS also provides a stored procedure facility [7], but the performance gains using this facility over loose-coupling were not significant.

Paper Organization. The organization of the rest of the paper is as follows. In Section 2, we first review the structure of computation of typical decision-support applications and describe how they are coded in the loosely-coupled integration. Next, we present our methodology for tightly-coupled integration. In Section 3, we give the case study of developing a tightly-coupled data mining application using this methodology. We also present the performance comparison between loosely-coupled and tightly-coupled integration. We conclude with a summary in Section 4.

2. Methodology

We introduce our methodology by giving an example of a simple loosely-coupled application and transforming it into a tightly-coupled one.

2.1. Computational Structure of a Decision-Support Application

Consider the nature of computation in a typical decision-support application. First some local variables are declared and initialized to keep track of the state of computation. Then an embedded SQL statement is used to define the set of desired records. This statement may cause the database system to perform computations such as sorting records in the table, performing group-by operators, computing aggregations, and applying selection predicates. Next, a loop in the application program accesses records in the result set one-by-one. As the records are fetched, some computation is performed using the values in the fields of the records, possibly saving intermediate results in the local variables. After exiting from the loop, the final computation is performed based on the state generated by the previous loop.

2.2. Loosely-Coupled Integration

Figure 1 shows a simple loosely-coupled application. This application retrieves sales records and counts how many customers bought each of the items sold. The schema for the table is *sales (tid, itemno)*. If two items i_1 and i_2 were bought in the customer transaction t_1, the *sales* table will have two records, $< t_1, i_1 >$ and $< t_1, i_2 >$. This application is somewhat contrived — we chose it to simplify the exposition of our methodology. Later in the paper, we will give example of a real application.

```
procedure AlgorithmLC()
begin
1   declare and initialize the array count[MAXSIZE];
2.  exec sql connect to database;
3.  exec sql declare cur cursor for
              select *
              from sales
              for read only;
4.  exec sql open cur;
5.  exec sql fetch cur into :tid, :itemid;
6.  while (sqlcode ≠ endOfRec) do {
7.     count[itemid] := count[itemid] + 1;
8.     exec sql fetch cur into :tid, :itemid;
9.  }
10. exec sql close cur;
11. print count array;
end
```

Figure 1: A Loosely-Coupled Application

The embedded SQL statements are prefixed by `exec sql` and are terminated by a semicolon. They can include references to host variables and such references are prefixed with a colon to distinguish them from column names of the tables. The variables *tid* and *itemid* of line 5 in Figure 1 are such host variables. The status of the execution of a SQL statement is returned to the application program in the so called SQL Communication Area. In particular, a numeric status indicator called `sqlcode` is returned.

Coming to the specifics of the application in Figure 1, the statement in line 1 declares and initializes the array *count* that maintains the count of the number of occurrences of each item in the *sales* table. The SQL statement in line 2 connects the application to the *database* in which the *sales* table is stored.

To access a set of records returned by a `select` statement, an iteration mechanism called *cursor* is provided. The statement in line 3 defines a database iterator called *cur* for the query specified in the `select` statement. The `for read only` clause allows block I/O; if this clause is omitted, the performance degrades severely.

The query is not executed until the cursor is opened by the SQL statement in line 4 of the algorithm. The `fetch` statement in line 5 is used to obtain the next record in the result set of the query. Fields of the retrieved record are copied into host variables specified in the `into` clause. Since the result of the `select` statement is normally a set of records, the `fetch` is executed inside a loop until there are no more records in the result set. Note that each fetch results in copy from the database address space to the application address space and process context switches, causing the performance degradation mentioned in the introduction. After we exit from the loop, the cursor *cur* is closed in line 10.

2.3. User-Defined Functions in DB2/CS

User-defined functions in DB2/CS [7] [8] are completely analogous to the built-in scalar functions except that they are defined and implemented by the users in a general-purpose programming language. Users can register a user-defined function via a `create function` statement, which describes the function, its input arguments, return value, and some other attributes. The executable of a user-defined function is stored at the database system server site so that database system can access and invoke the function whenever the function is referenced in a SQL statement. DB2/CS does not allow SQL statements inside a user-defined function.

The user-defined functions are normally kept in the subdirectory *sqllib/function* of the directory in which DB2/CS has been installed. More than one user-defined function can be kept in a library in this directory and there can be more than one library. Assume we have a user-defined function, called *allocSpace()*, written in C++. If this function is in a library called *mineudf*, we can register the function as follows:

```
create function allocSpace(int)
       returns int
       external name ´sqllib/function/mineudf!allocSpace´
       language c++ parameter style db2sql
       not variant no sql no external action
       not fenced
```

The important clause from our perspective in the above `create` statement is the `not fenced` clause. It allows the function to run in DB2's address space. The `not variant` clause indicates that the function always returns the same result for given argument values. The `no sql` clause specifies that the function does not contain any SQL statement. The `no external action` clause indicates that the function does not take some external action such as sending a message. These clauses are hints to the optimizer.

2.4. Tightly-Coupled Integration

For tightly coupling an application, we would like not to return from the database server after fetching every record. Rather, we will like to return only after all the records have been processed. It also means that the intermediate state of the computation should be saved within the address space of the database server. We illustrate by converting the loosely-coupled application in Figure 1

procedure AlgorithmTC()
begin
1. **exec sql connect to** database;
2. **exec sql select** allocSpace(MAXSIZE) **into** :blob
        from onerecord;
3. **exec sql select** *
        from sales
        where updateCount(:blob, TID, ITEMID) = 1;
4. **exec sql select** getResult(:blob) **into** :resultBlob
        from onerecord;
5. *update the array* count[MAXSIZE] *using* resultBlob;
6. **exec sql select** deallocSpace(:blob)
        from onerecord;
7. *print* count *array*;
end

Figure 2: A Tightly-Coupled Application

into a tightly-one and then give the general methodology. The tightly-coupled version is shown in Figure 2.

We first allocate space in the address space of the database server to maintain intermediate state of the computation. The user-defined function *allocSpace()* allocates work-area for saving state and initializes it. In this case, this function allocates an integer array of MAXSIZE elements to maintain counts, initializes all the elements of this array to zero, and returns the address of the array. This function is executed by the SQL statement in line 2 of Figure 2. We want this function to be executed only once. Therefore, the `select` statement has been defined over a *one-record* table[1]. The result of the statement on line 2 will be that the lone record from this table will be selected and the *allocSpace()* function will be executed for this record as this function appears in the `select` list. Consequently, the space for the array will be allocated and a pointer to this space is returned in the application variable *blob*. Admittedly, the value in variable *blob* has no meaning in the address space of the application program. However, this value is never referenced in the application program, but passed as input argument in the next SQL statement to another user-defined function.

The user-defined function *updateCount()* gets the starting address of work-area and the values of

[1]The *onerecord* table does not need to be a permanent table. Rather, this table can be dynamically created by changing the `from` clause to:

`from` (values(1)) as onerecord

For brevity, we omit this detail in all our examples.

columns in the *sales* table as input arguments, and updates information in the work-area depending on the column values in the input record. Importantly, the function always returns zero. This function is referenced in the `where` clause of the SQL statement on line 3, the selection condition being that the value returned by the function is 1. This condition will always be false. Therefore, this SQL statement never returns a record. However, the user-defined function is applied to each record in the *sales* table and the count array is updated. We thus have been able to push the application program logic into the database system, avoiding copying and context switches. When the result set of `select` statement has at most one record, we do not require cursor mechanism to retrieve the result. Thus, the while-loop in lines 4-8 of Figure 1 of the loosely-coupled algorithm has been transformed into one SQL statement.

The application program cannot access the address space allocated by *allocSpace()* in the database server. After we finish scanning the *sales* table, therefore, we bring the results from the database server through the user-defined function *getResult*. We again reference this function in the `select` list of the SQL statement in line 4 over a one-record temporary table. Finally, we deallocate the work-area allocated in database server with another user-defined function *deallocSpace()* before the application program exits.

2.5. General Methodology

Our methodology for developing tightly-coupled applications on DB2/CS has the following ingredients:

- 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 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. Create this temporary one-record dynamically by using the construct *(value(1))* `as` *onerecord* in the `from` clause.

- To execute a user-defined function once for each selected record without ping-ponging between the database address space and the application address, have the function always 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 last 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 *allocSpace()* 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 (e.g. *updateCount()* and *getResult()*) save and query the state of the application program. The function *deallocSpace()* can be thought of as the destructor for the object.

Note that our approach does not require changes to the database software. We simply utilize user-defined functions in SQL in a novel way. In effect, we are using SQL to orchestrate the execution of the application logic within database system, using SQL also as an inter-process communication mechanism. This use of the user-defined functions is quite different from the earlier usage, such as for applying a complex selection predicate on record-by-record basis with no residue between executions on two records [9] [12], or for integrating specialized data managers with the relational database systems [6].

3. A Case Study

To validate our methodology, we tightly-coupled a data mining application to DB2/CS and measured its performance against the loosely-coupled implementation. In this section, we report our experiences as a case study.

```
procedure AprioriAlg()
begin
L_1 := {frequent 1-itemsets};
for ( k := 2; L_{k-1} ≠ ∅; k++ ) do {
    C_k := apriori-gen(L_{k-1});  // New candidates
    forall transactions t ∈ D do {
        forall candidates c ∈ C_k contained in t do
            c.count++;
    }
    L_k := {c ∈ C_k | c.count ≥ min-support}
}
Answer := ⋃_k L_k;
end
```

Figure 3: Apriori Algorithm

Data mining (also called knowledge discovery in databases) is the efficient discovery of previously unknown patterns in large databases, and is emerging as a major application area for databases [5] [13]. Decision support applications for retail organizations are the major drivers of this technology. Almost all large retail organizations collect and store massive amount of point-of-sales data, referred to as the *basket* data. A record in such a database table consists of a transaction id and an item id. All the items belonging to the same transaction id represent a customer transaction. The input data to the table comes naturally sorted by transaction id.

We consider specifically the problem of mining association rules over basket data, introduced in [1]. An association rule is an expression $X \Longrightarrow Y$, where X and Y are sets of items. The intuitive meaning of such a rule is that transactions which contain X tend to contain Y. An example of such a rule might be that 98% of customers who purchase tires and automobile accessories also get automotive services. We use the Apriori algorithm in [2] for our case study. We give both loosely-coupled and tightly-coupled implementation of this algorithm over DB2/CS and present performance comparisons using several real-life datasets.

3.1. Overview of the Apriori Algorithm

The problem of mining association rules is decomposed into two subproblems [1]: 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 3. 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}, {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. See [1] for details of the algorithm.

3.2. Loosely Coupled Integration

Figure 4 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 3. 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 3. 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.

3.3. Tightly Coupled Integration

We give a tightly-coupled implementation of the Apriori algorithm in Figure 5 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.

```
procedure LoosleyCoupledApriori() :
begin
exec sql connect to database;
exec sql declare cur cursor for
          select TID, ITEMID from sales
          for read only;
exec sql open cur;
notDone := true;
while notDone do {
  exec sql fetch cur into :tid, :itemid;
  if (sqlcode ≠ endOfRec) then
    update counts for each itemid;
  else
    notDone := false
}
exec sql close cur;
L_1 := {frequent 1-itemsets};
for ( k := 2; L_{k-1} ≠ ∅; k++ ) do {
  C_k := apriori-gen(L_{k-1});  // New candidates
  exec sql open cur;
  t := ∅; prevTid := −1; notDone := true;
  while notDone do {
    exec sql fetch cur into :tid, :itemid;
    if (sqlcode ≠ endOfRec then) {
      if (tid ≠ prevTid and t ≠ ∅) then {
        forall candidates c ∈ C_k contained in t do
          c.count++;
        t := ∅; prevTid := tid;
      }
      t := t ∪ itemid
    }
    else
      notDone := false;
  }
  exec sql close cur;
  L_k := {c ∈ C_k | c.count ≥ min-support}
}
Answer := ⋃_k L_k;
end
```

Figure 4: Loosely-coupled Apriori Algorithm

```
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 GenL_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 GenL_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 5: Tightly-coupled Apriori Algorithm

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 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 4.

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.

We could have combined the functionality of some of the above user-defined functions in one function. We did not do that because we wanted to reuse as much as possible the previous implementation of the Apriori algorithm with loosely-coupled integration. The result was that we were able to create the tightly-coupled version with minimal programming effort.

3.4. Performance

To assess the effectiveness of our approach, we empirically compared the performance of tightly-coupled and loosely-coupled implementations of the Apriori algorithm. The machine used for the experiments was a IBM RS/6000 250 workstation with a CPU clock rate of 80 MHz, 128 MB of main memory, and running AIX 3.2.5. The database system used was DB2/CS Version 2.1 that supports user-defined functions. The experiments were performed in a configuration in which the application client and the database server were running on the same machine.

Six real-life customer datasets were used in the experiment. These datasets were obtained from department stores, supermarkets, and mail-order companies. Table 1 summarizes the characteristics of these datasets.

For each data set, we measured the execution times of both loosely-coupled and tightly-coupled implementations of the Apriori algorithm for different support levels. When measuring running

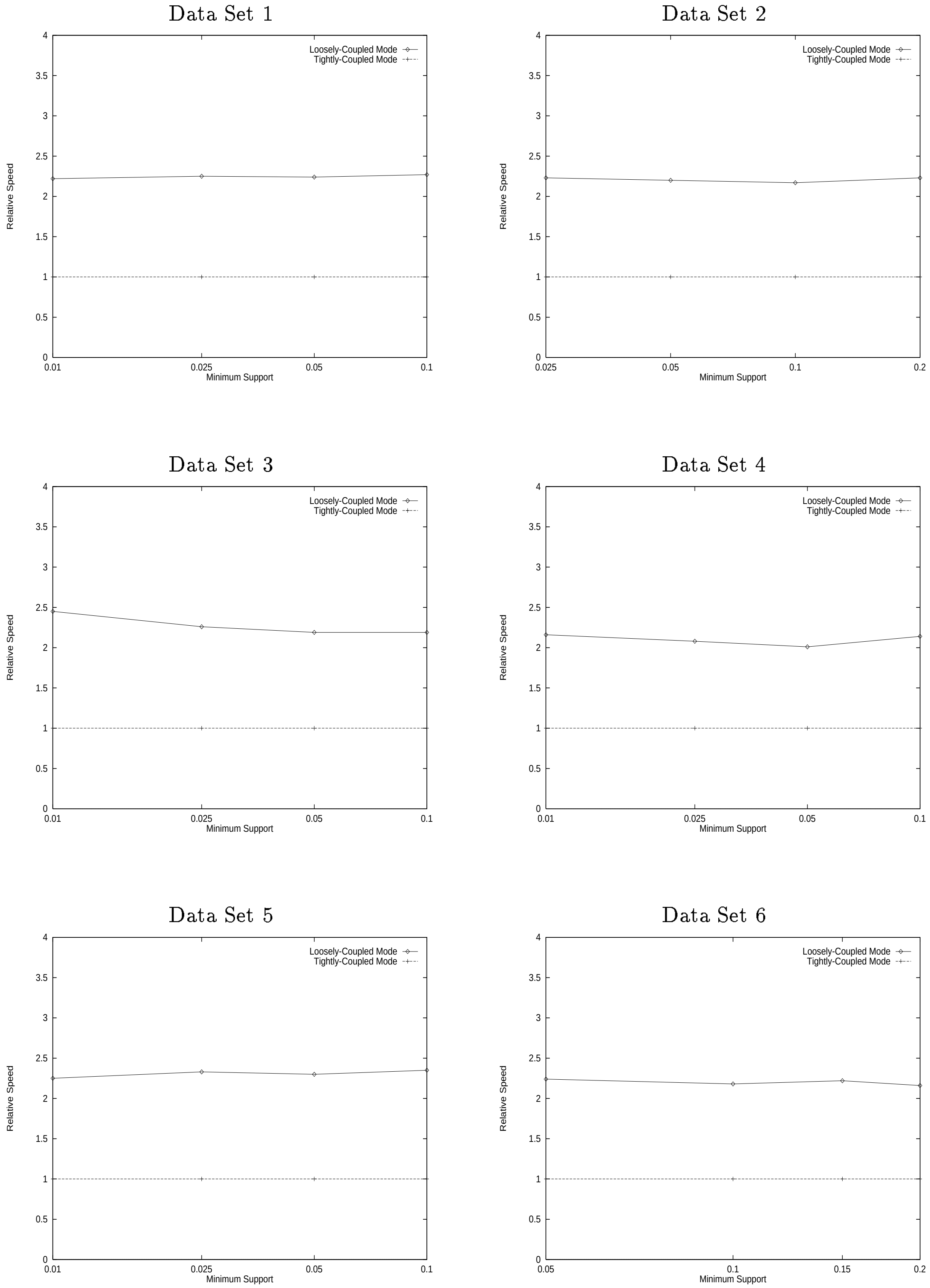

Figure 6: Relative execution speed of the loosely-coupled and tightly-coupled implementations of the Apriori algorithm

Table 1: Characteristics of the Datasets

Name Name	# of Tuples	Size of Tuple	Size of Tid	Size of Itemid	Avg. # of Items per Transaction	# of Items
Data Set 1	115192	8	5	3	2.45	58
Data Set 2	500000	86	30	13	4.84	47838
Data Set 3	662210	64	25	7	2.09	26412
Data Set 4	512596	23	11	12	4.24	51440
Data Set 5	2600000	25	15	5	2.62	15499
Data Set 6	2600000	26	16	10	1.64	56358

times for the loosely-coupled version, we used *row blocking* that allows a group of rows to be returned to an application in response to a `fetch` request [7]. It reduces the overhead of the database manager as a block of rows are retrieved in a single operation.

Figure 6 summarizes the experimental results. The x-axis is the minimum support used for each experiment. The y-axis is the ratio of execution time of the loosely-coupled mode to the execution-time of the tightly-coupled mode. The minimum supports for each data set were chosen to allow a reasonable number of passes and frequent itemsets.

We see that in all cases, tight-coupling gives more than two fold performance advantage over loose-coupling. We would like to mention that work is underway to improve the performance of the implementation of the user-defined functions in DB2. The tightly-coupled implementation would directly benefit from any performance gains from this effort.

4. Summary

We considered the performance degradation due to copying and context switches when integrating a data-intensive application to the IBM DB2/CS relational database system using the conventional loosely-coupled integration paradigm. We proposed a methodology for tightly coupling applications to DB2/CS to build high-performance applications. Our approach does not require changes to the DB2/CS software, but simply utilizes the user-defined functions in a novel way. We validated our technique by converting a loosely-coupled data mining application into a tightly-coupled one. Not only did we empirically observe a nearly two-fold performance improvement, we found that the programming effort was quite minimal and we were able to reuse most the existing code. This application is now being field-deployed.

Our fond hope is that the designers of high-performance, decision-support applications on DB2/CS will benefit from this methodology. The database system designers may also take cues from this methodology while designing native support for emerging data-intensive decision-support applications.

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.

[1] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Mining association rules between sets of items in large databases. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 207–216, Washington, D.C., May 1993.

[2] Rakesh Agrawal and Ramakrishnan Srikant. Fast Algorithms for Mining Association Rules. In *Proc. of the 20th Int'l Conference on Very Large Databases*, Santiago, Chile, September 1994.

[3] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In M. Stonebraker, editor, *Readings in Database Systems*, pages 946–954. Morgan Kaufmann, 1994.

[4] M. P. Atkinson and P. Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2), June 1987.

[5] Gartner Group. Data mining: The next generation of business intelligence? ATG Research Note T-517-246, Gartner Group Inc., Stamform, CT, March 1994.

[6] W. Hasan, M. Heytens, C. Kolovson, M.-A. Neimat, S. Potamianos, and D. Schneider. Papyrus GIS demonstration. In *Proc. of the ACM-SIGMOD Int'l Conference on the Management of Data*, Washington, D.C., June 1993.

[7] IBM. *DB2 Application Programming Guide Version 2*, 1995.

[8] IBM. *DB2 SQL Reference for Common Servers Version 2*, 1995.

[9] G. Lohman, B. Lindsay, H. Pirahesh, and K. B. Schiefer. Extensions to starburst: Objects, types, functions, and rules. *Communications of the ACM*, 34(10), October 1991.

[10] Montage. *Montage User's Guide*, March 1994.

[11] Oracle. *Oracle RDBMS Database Administrator's Guide Volumes I, II (Version 7.0)*, May 1992.

[12] M. Stonebraker and L. A. Rowe. The design of postgres. In *Proc. of the ACM-SIGMOD Int'l Conference on the Management of Data*, Washington, D.C., May 1986.

[13] Business Week. Database marketing, September 1994.