Sybase IQ In-database MapReduce – Stock Price Variance Use Case and Code Sample

For questions, contact: Courtney Claussen, Product Management, courtney.claussen@sybase.com

Introduction

This document describes the MapReduce processing model, its benefits and weaknesses, and how Sybase IQ 15.4 has evolved into a platform that can execute highly distributed processing in a manner similar to MapReduce. With Sybase IQ in-database MapReduce, the developer writes programmatic "map" and "reduce" functions in C++, and invokes them from SQL. The data to be analyzed is selected and partitioned using SQL, the map and reduce functions execute within the Sybase IQ server process space, and results can be stored back into the database. In-database MapReduce takes advantage of the Sybase IQ PlexQ distributed processing architecture for performance, as well as the data indexing, security and high availability features of the Sybase IQ database. An additional benefit is Sybase IQ's integration with data management, ETL and BI tools.

Of particular interest to the reader, this document presents an example of a programming problem – computation of price variances of a set of stocks - that can be solved using Sybase IQ in-database MapReduce. You will see the source code that implements the solution, understand the logic of the implementation, learn how to build and deploy the executable code, and how to invoke the map and reduce functions from SQL.

First of All, What is MapReduce?

MapReduce is an approach for highly distributed processing of large datasets using a grid of computers. There are two steps:

"Map" step: A master node takes the input data, partitions it into subsets, and distributes them to worker nodes. A worker node might repeat the partitioning process, leading to a multi-level tree structure. The worker nodes process the smaller data subsets, and pass the answers back to the master node.

"Reduce" step: The master node collects the answers from the worker nodes performing the map operations, and combines the results to answer the problem it was originally trying to solve.

The MapReduce framework has been popularized in connection with Hadoop, an open source computing platform that implements MapReduce on clusters of low cost machines. A large number of companies are now using or experimenting with Hadoop clusters for the purposes of storing large amounts of data – today's "big data" - and analyzing it with MapReduce and other tools.

What are the Benefits of MapReduce?

There are several benefits of MapReduce over conventional data processing techniques:

- The model is easy to use, even for programmers without experience with distributed systems. MapReduce hides the details of parallelization, fault tolerance, locality optimization and load balancing.
- MapReduce allows developers to write applications in their language of choice: Java, C#, Python, C++, R, etc.
- A large variety of problems are easily expressible as MapReduce computations: sorting, data mining, machine learning, and many others.
- MapReduce enables scaling of applications across large clusters of machine comprising thousands of nodes, with fault-tolerance built in.

Storing big data within a Hadoop or similar environment has significant advantages over traditional approaches to data warehousing: it is less expensive, caters to non-relational data, scales out easily and is simple to establish and implement.

However, there is a downside. In particular, Hadoop does not have the fault tolerant capabilities that most enterprises expect. For example, both the NameNode and JobTracker within Hadoop represent single points of failure. Secondly, Hadoop has been designed to run batch-based queries and is not generally suitable for real-time query processing. Moreover, it will not perform as well as a purpose-built data warehouse.

Fortunately (!), Sybase IQ now allows the user to execute MapReduce-like processing all within the database, to support big data applications with the added reliability and real-time capabilities of a database system.

How does Sybase IQ Execute MapReduce-like Processing?

Sybase IQ has an extensibility framework called UDF (User Defined Functions). UDFs allow the user to write functions in either C++ or Java, and execute the functions from SQL. UDF functions written in C++ execute within the same process space as the Sybase IQ server. UDF functions written in Java run in a separate process, but still close to the data for performance. There are several different APIs that UDFs can conform to, depending on the structure of data being input to and output from the function: single value, aggregates (multiple values), or tables.

The most sophisticated kind of UDF is called a TPF – Table Parameterized Function. A TPF accepts a table as input, and outputs a table of data. This is ideal for applications that involve bulk data exchanges, such as statistical analysis or MapReduce.

TPFs are invoked from SQL, and a "PARTITION BY" clause can be used in the SQL statement to partition the data sent to the TPF. Each data partition will be processed by a separate invocation of the TPF, to achieve parallel and distributed processing across a Multiplex grid – similar to a MapReduce application running in a Hadoop cluster.

Differences between MapReduce in Hadoop and MapReduce in Sybase IQ

MapReduce in IQ mirrors MapReduce in Hadoop in key ways:

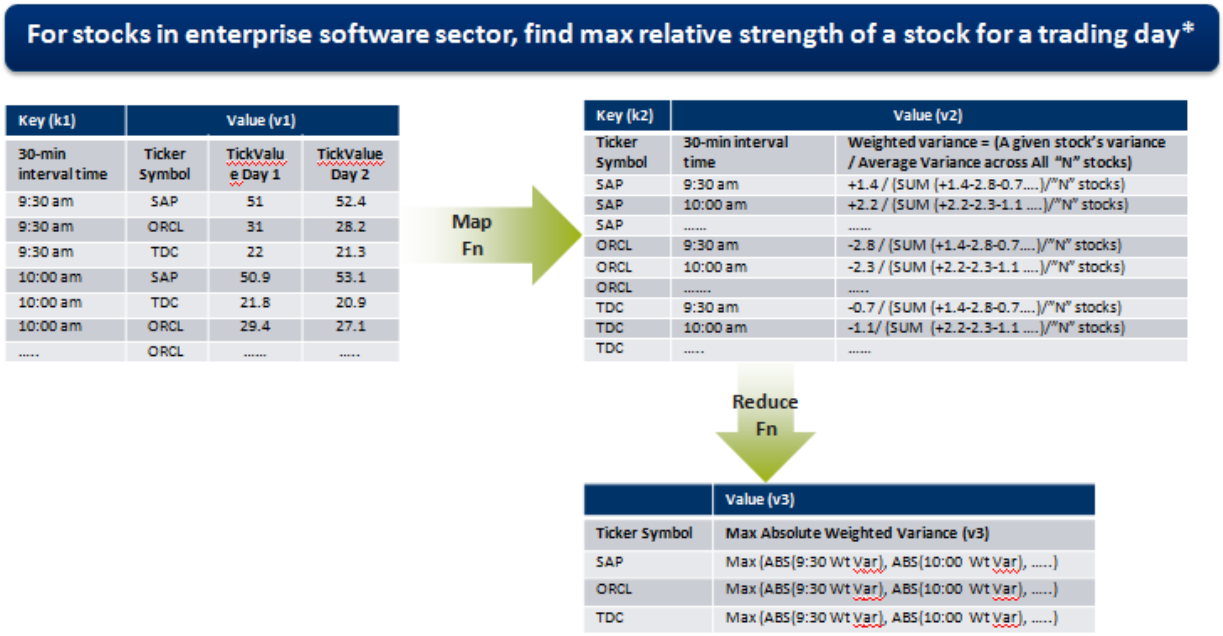
- MapReduce functions can be written in popular programming languages
- MapReduce functions consume and produce data sets in bulk
- MapReduce functions execute as parallel job working on disjoint data sets
- Several levels of nested MapReduce function calls are possible, resulting in multi-level tree execution
- MapReduce processing is fault tolerant, with participating worker units taking over for failed worker units

Along with the strong similarities between MapReduce in Hadoop and MapReduce in Sybase IQ, there are also some important differences:

MapReduce in Hadoop	Sybase IQ Native MapReduce
MapReduce functions are invoked within a completely procedural framework.	MapReduce functions are invoked from declarative SQL.
Data store is a distributed file system, is batch oriented, has little to no security protection on data access, and complex joins are cumbersome.	Data store is a column store DBMS which allows ad-hoc queries, complex joins, and enterprise security protection.
Data is schema-less and requires no ETL.	Data requires a schema and at least some ETL.
Fault tolerant, however there are two single points of failure: NameNode and JobTracker	Sybase IQ has high fault tolerance, and any node can take over for another one to complete processing.
Requires a lot of hardware for performance – shared nothing MPP is mandatory.	Requires less hardware footprint for good performance – SMP or shared everything MPP.
Wider variety of programming language support: C++, Java, PHP, etc.	C++ only.

Description of a MapReduce Problem that can be Processed by Sybase IQ

Over the period of time since Sybase IQ 15.4 was released, the Sybase IQ product management team has been presenting an example of how Sybase IQ in-database MapReduce can be used to solve a programming problem – computing price variations on stock tick data:



Price variations contribute to the perception of the strength of a stock. The relative strength of a stock is a numerical measurement expressed as a percentage. For example, if a stock has a relative strength of 75, then it has outperformed 75% of the stocks over a specified period. Specifically, relative strength is about a stock's price over time. The higher the number, the higher the performance. If you look at a stock for a short period, relative strength may not mean much. However, if a stock holds a strong relative strength number for a longer period, say 6 months or more, then you might be comfortable that the company is legitimately moving forward. In statistics, variance is a measure of how far a set of numbers is spread out from the mean. Variance in stock price over time is a measure of its volatility, and therefore risk level and strength. Weighted variance looks at variance with respect to the variance of other members of a set. So you are looking at relative performance compared to other stocks of interest.

The data points in this example are tick prices at half hour intervals across a trading day for a set of stocks. The tick data is received in random order. The "map" part of the map/reduce operation partitions the data by the 30-minute interval time to calculate the weighted variance of all stocks at a particular time of day. The "reduce" part of the MapReduce operation then aggregates the results of all of the map functions, partitions them by stock symbol, and finds the maximum variance value for each stock –its biggest rise or fall two day trading period.

Logic of TPFs to Implement MapReduce Example

The Sybase IQ server and a TPF form a producer and consumer relationship when exchanging rows of data. The producer produces table rows; the consumer consumes table rows. The server is the producer of the input table to the TPF. The TPF is the consumer of the input table. The TPF also generates an output table. Here, the TPF is the producer of the output table, and the server is the consumer of the output table.

The server delivers input table data to the TPF in a structure called a "row block". The server is responsible for allocating and deallocating this structure. In general, the layout of a row block conceptually matches the row and column format of the table; a row block consists of a number of rows, and each row consists of a number of columns. In the implementation of this TPF, the server allocates the row block, fills it with data, and makes it available to the TPF. The server may write data to the row block many times, until the entire input table is consumed by the TPF.

When developing a TPF, the library must declare what functions are available for the server to call. The code defines a library entry-point function, and specifies how the server will get output row data from the TPF – whether the server should allocate and deallocate the output row block, or whether the TPF will handle that. The TPF also describes metadata about the input and output tables that it expects – number of columns and data types of the columns. And of course, the TPF contains the logic to read the input table, process it, and generate the output table.

At a high level, our "map" function calculates the relative variance of stock prices for a particular stock at a particular time of day. It receives an input table of prices for all stocks partitioned by time of day. Our "reduce" function gets the results of the map function, but partitioned by stock symbol. So it consumes a set of relative price variances for each time of day for a particular stock. The reduce function produces a single row of data: the maximum relative price variance over the course of the day. The top level SELECT statement that invokes the map and reduce functions orders the results in a descending order, so the stock with the highest relative price variance shows up first. This is the stock with the lowest strength, because it had the biggest relative change in stock price between day 1 and day 2.

Logic of the Map TPF:

(The source code for this function is in an appendix at the end of this document.)

The Map TPF receives an input table of stock tick data, partitioned by time of day. So, one invocation of the Map TPF receives all the day 1 and day 2 stock prices for all stocks at a particular time of day – say 9:30AM. The Map TPF returns an output table of data. There is one row in the output table for each row in the input table. The input table contains day 1 and day 2 prices for a partitular stock at a particular time of day. The output table contains the relative variance of the stock price for each stock at the particular time of day. There are several functions that make up the TPF. The functions are called in the following order:

1. mapvar_describe
2. mapvar_evaluate
3. mapvar_open
4. mapvar_fetch_into
5. mapvar_close

mapvar_describe is called only once, no matter how many partitions of input data there are, and how many separate invocations of the TPF there will be during the course of the SQL query. This function describes the metadata of the input and output tables expected by the TPF.

mapvar_evaluate is called only once, no matter how many partitions of input data there are, and how many separate invocations of the TPF there will be during the course of the SQL query. This function tells the server which functions the TPF is implementing.

When the TPF is invoked for a particular partition of input data (there is a partition for each half hour interval of a stock trading day), the server calls mapvar_open. This function opens the input table, allocates memory for a variable to hold some state information, and calculates the average of all the stock price differences between day 1 and day 2. This average value will be needed later, when generating the output table. Then mapvar_open rewinds the input table, because it will have to be traversed again to calculate the weighted variance.

Then the server calls mapvar_fetch_into multiple times until the output table is completely generated. The output table holds a row for each row in the input table. The input row consists of tickTime, stockSymbol, day1Price and day2Price. The output row corresponding to the input row consists of stockSymbol, tickTime, relativeVariance. The relative variance value for a stock is calculated by: (day2Price – day1Price) / AveragePriceDifference). The AveragePriceDifference value was calculated in the call to mapvar_open.

The TPF will tell the server that the output table is complete, and then the server calls the mapvar_close function. This function closes the input table, and frees the memory for the state object.

Logic of the Reduce Function:

(The source code for this function is in an appendix at the end of this document.)

The Reduce TPF receives an input table of relative price variances, partitioned by stock symbol. So, one invocation of the Reduce TPF receives price variance values at each half hour interval in a trading day for a particular stock. The Reduce TPF returns a single row of data for a stock: the maximum of the absolute value of all the price variances, to find the biggest jump during a two day trading period. The output table contains the relative variance of the stock price for each stock at the particular time of day. There are several functions that make up the TPF. The functions are called in the following order:

1. redvar_describe
2. redvar_evaluate
3. redvar_open
4. redvar_fetch_into
5. redvar_close

redvar_describe is called only once, no matter how many partitions of input data there are, and how many separate invocations of the TPF there will be during the course of the SQL query. This function describes the metadata of the input and output tables expected by the TPF.

redvar_evaluate is called only once, no matter how many partitions of input data there are, and how many separate invocations of the TPF there will be during the course of the SQL query. This function tells the server which functions the TPF is implementing.

When the TPF is invoked for a particular partition of input data (there is a partition for each stock symbol), the server calls redvar_open. This function opens the input table, and allocates memory for a variable to hold some state information.

Then the server calls redvar_fetch_into multiple times until the output table is complete. The output table holds a single output row – the maxium price variance for the stock. The input row consists of stockSymbol, tickTime, and relativeVariance. The single output row consists of stockSymbol, tickTime, and maxVariance.

The TPF will tell the server that the output table is complete, and then the server calls the redvar_close function. This function closes the input table, and frees the memory for the state object.

How to Build the TPFs

In the "$IQDIR15/samples/udf" directory of a Sybase IQ 15.4 installation, there are code samples and build scripts for building TPFs. (We are assuming the Linux platform for this example.)

Create files mapvar.cxx (map function) and redvar.cxx (reduce function) from the text in the appendices in this document. Copy these files to $IQDIR15/samples/udf.

Copy $IQDIR15/samples/udf/build.sh to $IQDIR15/samples/udf/buildmr.sh.

Edit $IQDIR15/samples/udf/buildmr.sh with a text editor. Change the following section of the file to create the library "mapreduce.so", and only compile the files mapvar.cxx and redvar.cxx:

```
so_file_name="mapreduce.so"
object_file_location="./build_temp"

udf_scalar_files=""
udf_aggregate_files=""
udf_table_files=""
udf_table_files1=""
udf_tpf_files="mapvar.cxx redvar.cxx"
udf_common_files="udf_utils.cxx"
udf_v3_main="my_main.cxx"
udf_v4_main="udf_main.cxx"
```

Run the "buildmr.sh" command. There will be a new library mapreduce.so created in the build_temp subdirectory. Copy this library to $IQDIR15/lib64 and restart your Multiplex reader node to pick up the new library.

Stored Procedure Creation

After your library is built, you will need to create two stored procedures to invoke the map and reduce functions. Use 'dbisql' to create these in your Sybase IQ database:

```
CREATE OR REPLACE PROCEDURE mapvar_tpf( IN stockTickTable
      TABLE( inTickTime CHAR(6), inStockSymbol CHAR(10),
            inDay1Price DOUBLE, inDay2Price DOUBLE ) )
      RESULT( StockSymbol CHAR(10), TickTime CHAR(6), StockVariance
            DOUBLE )
      EXTERNAL NAME 'mapvar@mapreduce'
;

CREATE OR REPLACE PROCEDURE redvar_tpf( IN stockTickTable
      TABLE( inStockSymbol CHAR(10), inTickTime CHAR(6),
            inVariance DOUBLE ) )
      RESULT( StockSymbol CHAR(10), TickTime CHAR(6),
            MaxStockVariance DOUBLE )
      EXTERNAL NAME 'redvar@mapreduce'
;
```

Create Input Table Definition and Load Data

You will need an table to hold the stock tick data:

```
create table TickTable (
      tickTime          char(6),
      stockSymbol       char(10),
      day1Price         double,
      day2Price         double,
      primary key (tickTime, stockSymbol)
);
```

Note that 'tickTime' is a character data type, not a time. No datetime computations were done in the TPF, and to simplify the example, the time was left as character data.

Here is data for the TickTable table (copy and paste this text to a data file that you can load into IQ):

```
9:30|ACME|51.1|51.9|
10:00|ACME|51.9|70.3|
10:30|ACME|51.8|69.8|
11:00|ACME|52.3|69.6|
11:30|ACME|52.9|69.1|
12:00|ACME|50.5|68.4|
12:30|ACME|50.3|67.6|
1:00|ACME|52.1|63.2|
1:30|ACME|51.8|61.4|
2:00|ACME|52.3|60.7|
2:30|ACME|52.5|59.3|
3:00|ACME|52.7|55.3|
3:30|ACME|51.2|52.6|
```

```
4:00|ACME|51.7|52.5|
9:30|GLOBEX|41.2|40.8|
10:00|GLOBEX|41.3|40.9|
10:30|GLOBEX|40.5|40.2|
11:00|GLOBEX|40.7|40.4|
11:30|GLOBEX|40.9|40.7|
12:00|GLOBEX|41.4|40.6|
12:30|GLOBEX|41.2|40.9|
1:00|GLOBEX|41.0|41.3|
1:30|GLOBEX|41.4|41.4|
2:00|GLOBEX|41.2|41.6|
2:30|GLOBEX|41.3|41.4|
3:00|GLOBEX|39.9|41.2|
3:30|GLOBEX|40.3|40.9|
4:00|GLOBEX|40.7|41.0|
9:30|LEXCORP|62.2|61.3|
10:00|LEXCORP|62.1|61.6|
10:30|LEXCORP|62.4|62.0|
11:00|LEXCORP|62.5|62.3|
11:30|LEXCORP|62.1|62.1|
12:00|LEXCORP|61.8|61.8|
12:30|LEXCORP|61.9|61.9|
1:00|LEXCORP|61.7|62.4|
1:30|LEXCORP|61.2|62.2|
2:00|LEXCORP|61.3|62.1|
2:30|LEXCORP|60.9|61.8|
3:00|LEXCORP|60.8|61.7|
3:30|LEXCORP|61.1|61.8|
4:00|LEXCORP|61.1|63.2|
9:30|VIRTUCON|35.1|36.0|
10:00|VIRTUCON|35.3|36.2|
10:30|VIRTUCON|35.8|36.3|
11:00|VIRTUCON|35.1|36.5|
11:30|VIRTUCON|34.8|36.9|
12:00|VIRTUCON|34.9|36.7|
12:30|VIRTUCON|35.3|36.5|
1:00|VIRTUCON|35.8|36.1|
1:30|VIRTUCON|35.6|36.9|
2:00|VIRTUCON|35.4|36.2|
2:30|VIRTUCON|35.4|35.9|
3:00|VIRTUCON|35.9|35.7|
3:30|VIRTUCON|36.1|35.8|
4:00|VIRTUCON|35.8|35.7|
9:30|ZIFF|27.7|51.9|
10:00|ZIFF|27.8|24.9|
10:30|ZIFF|23.4|25.2|
11:00|ZIFF|23.5|25.1|
11:30|ZIFF|23.8|25.8|
12:00|ZIFF|24.3|25.1|
12:30|ZIFF|24.4|25.8|
1:00|ZIFF|24.2|26.5|
1:30|ZIFF|24.0|28.9|
```

```
2:00|ZIFF|24.1|28.5|
2:30|ZIFF|24.5|29.0|
3:00|ZIFF|24.7|29.1|
3:30|ZIFF|24.8|29.0|
4:00|ZIFF|24.6|25.9|
```

You can load the data with this command (substitute <data_file> with the path to your data file):

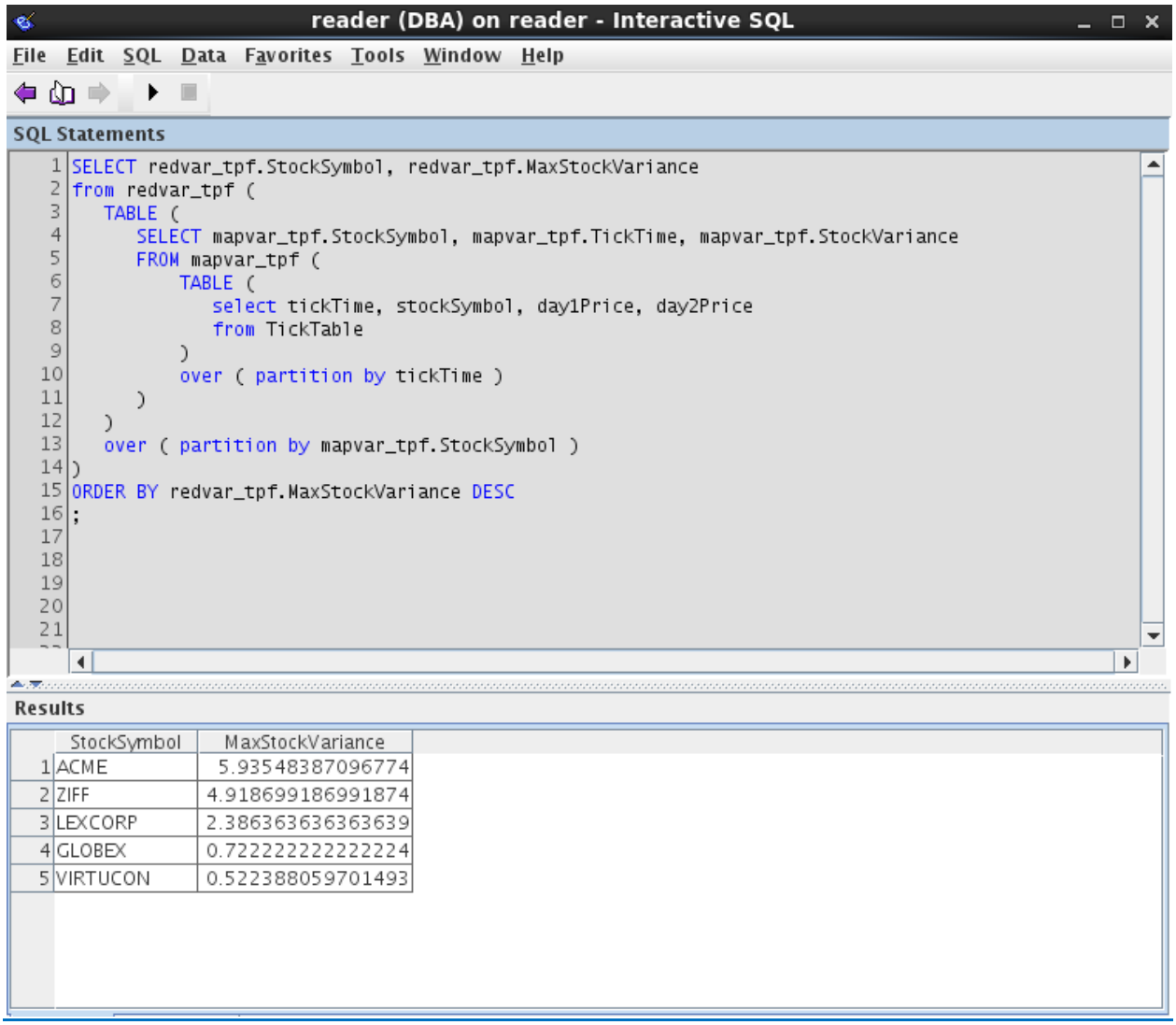
```
load table TickTable (
      tickTime     '|',
      stockSymbol  '|',
      day1Price    '|',
      day2Price    '|'
)
from '<data_file>'
preview on notify 250000 block size 500000
row delimited by '\x0a'
quotes off escapes off
;
```

How to Execute the TPFs to Calculate Maximum Stock Price Variance

You will need to execute the following SQL on a reader node of a Multiplex. Because C++ TPFs execute within the database server process space, for safety reasons, Sybase requires that you not alter the database when running a TPF. To insure this, you must run TPFs only on a reader node that does not have privileges to change the database. You will execute the following SQL:

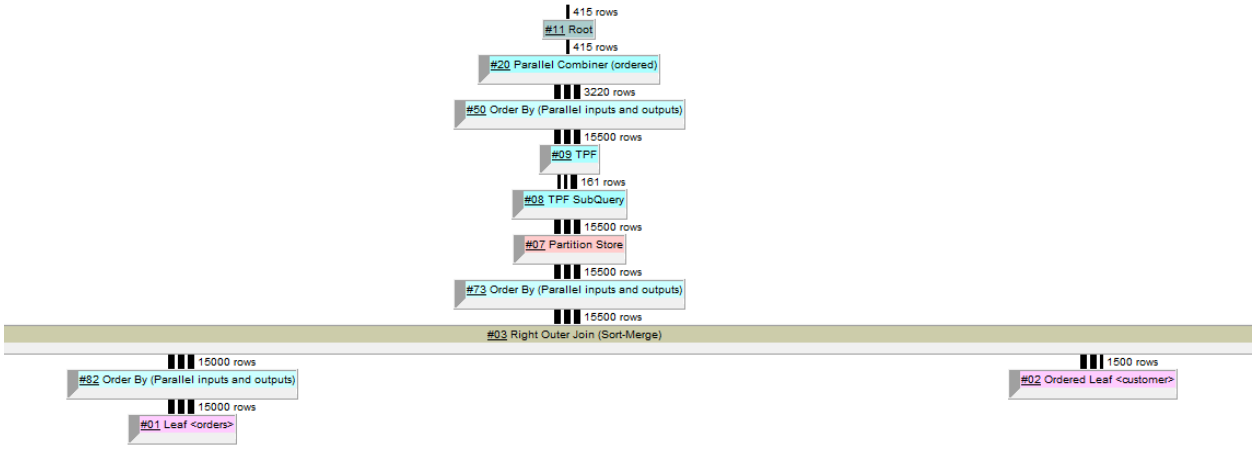
```
SELECT redvar_tpf.StockSymbol, redvar_tpf.MaxStockVariance
from redvar_tpf (
   TABLE (
      SELECT mapvar_tpf.StockSymbol, mapvar_tpf.TickTime,
           mapvar_tpf.StockVariance
      FROM mapvar_tpf (
          TABLE (
             select tickTime, stockSymbol, day1Price, day2Price
             from TickTable
          )
          over ( partition by tickTime )
      )
   )
   over ( partition by mapvar_tpf.StockSymbol )
)
ORDER BY redvar_tpf.MaxStockVariance DESC;
;
```

Here is a picture of running this command from a DBISQL window:



Parallelism of TPFs

When you partition the input table to a TPF, each partition can be executed in parallel, and potentially distributed onto another machine in a Sybase IQ Multiplex. The map TPF receives partitions of data for each half hour of a trading day, so potentially 15 machines (for 15 half hour intervals between 9 and 4) could partitipate in distributing the query. The reduce TPF receives data partitioned by stock symbol, so potentially a number of machines equal to the number of different stock symbols could participate in distributing the query. Here is an example of a query plan for a query that distributes a TPF (this is for a different query than described in this example):



If a part of a query is distributed, you will see a triple black line between nodes that were distributed. When you position a mouse cursor over the row count next to the parallel lines in the display, it will show the number of remote rows (how many were distributed). The width of the rightmost bar is sized depending on the number of remote rows. Look for these artifacts in the query plan when you execute TPFs from a SELECT statement. Remember that TPFs may run only on reader nodes of a Multiplex.

Summary

This document described how Sybase IQ has become a platform for running MapReduce-like distributed processing, and some of the benefits that come from running MapReduce inside a data warehouse environment. You have seen an example of a MapReduce programming problem, and how you can implement it using Sybase IQ's UDF extensibility framework.

Appendix: "Map" code

```
// Copyright (c) 2011 Sybase, Inc.
// All rights reserved. All unpublished rights reserved.
//
// ***************************************************
// This sample code is provided AS IS, without warranty or liability
// of any kind.
//
// You may use, reproduce, modify and distribute this sample code
// without limitation, on the condition that you retain the foregoing
// copyright notice and disclaimer as to the original Sybase, Inc.
code.

#include <stdio.h>
#include <string.h>
#include "extfnapiv4.h"
#include "udf_utils.h"

/*****************************************************************
 * This TPF example implements a function to compute the weighted
price
 * variance of a stock at a particular time of day compared to other
stocks
 * at the same time of day.  The input to the TPF are rows with the
following
 * columns:
 *
 * tickTime  stockSymbol  day1Price  day2Price
 *
 * The TPF receives data partitioned by tickTime.  Here is some
example
 * pipe delimited data for 5 stocks at time 9:30AM:
 *
 * 9:30|ACME|51.1|51.9|
 * 9:30|GLOBEX|41.2|40.8|
 * 9:30|LEXCORP|62.2|61.3|
 * 9:30|VIRTUCON|35.1|36.0|
 * 9:30|ZIFF|27.7|51.9|
 *
 * The logic of this TPF is to first calculate the average
 * price difference: AvgPriceDiff = AVG ( day2Price - day1Price ) of
all the
 * stocks in the partition at the particular time of day.  Then it
calculates
 * the relative price variance for a particular stock:
 *
 *          ( day2Price - day1Price ) / AvgPriceDiff
 *
 * The SQL required to create this procedure is:
 *
```

```
 * CREATE OR REPLACE PROCEDURE mapvar_tpf( IN stockTickTable
 *   TABLE( inTickTime CHAR(6), inStockSymbol CHAR(10), inDay1Price
DOUBLE,
 *       inDay2Price DOUBLE ) )
 *   RESULT( StockSymbol CHAR(10), TickTime CHAR(6), StockVariance
DOUBLE )
 *   EXTERNAL NAME 'mapvar@mapreduce';
 *
 * A SELECT statement to invoke this function looks like this:
 *
 *   select * from mapvar_tpf
 *   (
 *       TABLE
 *       (
 *           select tickTime, stockSymbol, day1Price, day2Price
 *           from TickTable
 *       )
 *       over ( partition by tickTime )
 *   );
 *
 ****************************************************************/


// Forward declarations.
static void UDF_CALLBACK mapvar_describe(
    a_v4_extfn_proc_context *ctx );

static void UDF_CALLBACK mapvar_evaluate(
    a_v4_extfn_proc_context *ctx,
    void *args_handle );

static short UDF_CALLBACK mapvar_open(
    a_v4_extfn_table_context * tctx );

static short UDF_CALLBACK mapvar_fetch_into(
    a_v4_extfn_table_context *tctx,
    a_v4_extfn_row_block *rb);

static short UDF_CALLBACK mapvar_close(
    a_v4_extfn_table_context *tctx);


/*
 * This defines the a_v4_extfn_proc descriptor that is used by the
 * server to determine what functions this TPF is implementing.
 */
static a_v4_extfn_proc mapvar_descriptor =
{
    NULL,        // _start_extfn
    NULL,        // _finish_extfn
    mapvar_evaluate,  // _evaluate_extfn
    mapvar_describe,  // _describe_extfn
```

```
    NULL,        // _leave_state_extfn
    NULL,        // _enter_state_extfn
    NULL,        // Reserved: must be NULL
    NULL         // Reserved: must be NULL
};

/*
 * This defines the main library export function entry point.  This
 * entry point is used when declaring the procedure in the
 * EXTERNAL NAME clause
 */
extern "C"
a_v4_extfn_proc * SQL_CALLBACK mapvar()
/****************************************/
{
    return &mapvar_descriptor;
}

/*
 * This defines the a_v4_extfn_table_func descriptor that is used by
 * the server to determine row data will be retreieved.
 */
static a_v4_extfn_table_func mapvar_table_funcs =
{
    mapvar_open, // _open_extfn
    mapvar_fetch_into,     // _fetch_into_extfn
    NULL,        // _fetch_block_extfn
    NULL,        // _rewind_extfn
    mapvar_close,     // _close_extfn
    NULL,        // Reserved: must be NULL
    NULL         // Reserved: must be NULL
};

static a_v4_extfn_table mapvar_table = {
    &mapvar_table_funcs,   // Table function descriptor
    3                      // number_of_columns (output): stockSymbol,
tickTime, Variance
};

/*
 * This structure will be used to retain state information.  It will
 * be allocated during the _open_extfn method and freed during the
 * _close_extfn method.
 */
struct mapvar_state {
    a_v4_extfn_table_context *   rs;
    a_v4_extfn_row_block *      rb_in;
    unsigned int      rb_current_row;
    double            priceDiffAvg;
    bool                        rs_processed;
};
```

```
/*****************************************************************
 * Implementation of the a_v4_extfn_table_func functions
 *****************************************************************/

/*****************************************************************
 * mapvar_open allocates a state object to hold state information for
 * the duration of the call to the TPF, reads through the input result
set
 * and calculates the average price difference between the day2Price
and the
 * day1Price for all stocks at a particular time of day.  Then the
function
 * rewinds the input result set so that mapvar_fetch_into can
calculate
 * the relative price variance of a particular stock at the particular
time
 * of day.
 *****************************************************************/
static short UDF_CALLBACK mapvar_open(
     a_v4_extfn_table_context * tctx )
/*************************************/
{
    an_extfn_value         value;
    mapvar_state *         state           = NULL;
    a_v4_extfn_table_context *   rs        = NULL;
    char                        s[20];

    LOG_MESSAGE( tctx->proc_context, "Enter mapvar_open" );

    // Allocate memory for the state variable using the
a_v4_extfn_proc_context
    // function alloc.
    state = (mapvar_state *)
      tctx->proc_context->alloc( tctx->proc_context,
                        sizeof( mapvar_state ) );

    // Initialize state variables.
    state->rs = NULL;
    state->rb_in = NULL;
    state->rb_current_row = 0;
    state->priceDiffAvg = 0;
    state->rs_processed = 0;

    // Read in the value of the input parameter and store it away in a
    // state object.  Save the state object in the context.
    if( !tctx->proc_context->get_value( tctx->args_handle,
                           1,
                           &value ) ) {

    // Send an error to the client if we could not get the value.
    tctx->proc_context->set_error(
         tctx->proc_context,
```

```
        17001,
        "Error: Could not get the value of parameter 1" );

      return 0;
    }

    // Open a result set for the input table.
    if( !tctx->proc_context->open_result_set( tctx->proc_context,
                                ( a_v4_extfn_table * )value.data,
                                &rs ) ) {
      // Send an error to the client if we could not open the result
      // set.
      tctx->proc_context->set_error(
          tctx->proc_context,
          17001,
          "Error: Could not open result set on input table." );

      return 0;
    }

    a_v4_extfn_row_block *      rbfb        = NULL;
    a_v4_extfn_row *            rfb         = NULL;
    a_v4_extfn_column_data *    cd_day1Price_in  = NULL;
    a_v4_extfn_column_data *    cd_day2Price_in  = NULL;
    unsigned int      countTicks = 0;
    double            priceDiffAvg = 0;
    double            priceDiff = 0;

    // When using fetch block to read rows from an input table, the
    // server will manage the row block allocation.
    while( rs->fetch_block( rs, &rbfb ) ) {

      // Each sucessful call to fetch will fill rows in the server
      // allocated row block.  The number of rows retrieved is
      // indicated by the num_rows member.
      for( unsigned int i = 0; i < rbfb->num_rows; i++ ) {
          rfb = &(rbfb->row_data[i]);
          cd_day1Price_in = &(rfb->column_data[2]);
          cd_day2Price_in = &(rfb->column_data[3]);

             priceDiff = *( double * )( cd_day2Price_in->data );
             priceDiff = priceDiff - *( double * )( cd_day1Price_in-
>data );
             priceDiffAvg = priceDiffAvg + priceDiff;
             countTicks++;
      }
    }
    priceDiffAvg = priceDiffAvg / countTicks;
    LOG_MESSAGE( tctx->proc_context, "mapvar_open: calculated
priceDiffAfg: ");
    sprintf(s, "%lf", priceDiffAvg);
    LOG_MESSAGE( tctx->proc_context, s);
```

```
    if( !( rs->rewind( rs ) ) ) {
        // Send an error to the client if we could not rewind the
      // result set.
        tctx->proc_context->set_error(
          tctx->proc_context,
          17001,
          "Error: Could not rewind result set on input table." );

        return 0;
    }

    // Save pointer to open result set.  Will need during fetch_into.
    state->rs = rs;

    // Save average price differences for result set.  Will need
    // to calculate variace.
    state->priceDiffAvg = priceDiffAvg;
    state->rs_processed = 0;

    // Save the state on the context
    tctx->user_data = state;

    LOG_MESSAGE( tctx->proc_context, "Exit mapvar_open" );
    return 1;
}

/*****************************************************************
 * mapvar_get_next_row is an iterator that returns each row from the
 * input result set.  It may read through multiple row blocks until
 * the result set is drained.
 *****************************************************************/
static a_v4_extfn_row * mapvar_get_next_row(
     a_v4_extfn_table_context *tctx,
       mapvar_state * state)
/*****************************************/
{
    a_v4_extfn_table_context *   rs         = state->rs;
    a_v4_extfn_row *           r_in       = NULL;

    if ( !( state->rb_in ) || ( state->rb_current_row == state->rb_in-
>num_rows ) ) {
        if( !( rs->fetch_block( rs, &( state->rb_in ) ) ) ) {
            // No more input data.  Done.
            LOG_MESSAGE( tctx->proc_context, "mapvar_get_next_row: no
more input data" );
            return NULL;
        }
        state->rb_current_row = 0;
    }
    r_in = &(state->rb_in->row_data[state->rb_current_row]);
```

```
    state->rb_current_row++;
    return r_in;
}

/*****************************************************************
 * mapvar_fetch_into reads through the input result set rows, and for
 * each input row generates an output row that contains the value of
the
 * relative variance of the difference between the day2Price and the
 * day1Price of the stock at a particular time of day.
 *****************************************************************/
static short UDF_CALLBACK mapvar_fetch_into(
     a_v4_extfn_table_context *tctx,
     a_v4_extfn_row_block *rb_out)
/*****************************************/

{
    mapvar_state *              state = (mapvar_state *)tctx-
>user_data;
    a_v4_extfn_row *            r_in        = NULL;
    a_v4_extfn_column_data *    cd_timestamp_in  = NULL;
    a_v4_extfn_column_data *    cd_stocksym_in   = NULL;
    a_v4_extfn_column_data *    cd_day1Price_in  = NULL;
    a_v4_extfn_column_data *    cd_day2Price_in  = NULL;
    double            variance;

    LOG_MESSAGE( tctx->proc_context, "Enter mapvar_fetch_into" );

    // Because we are implementing fetch_into, the server has provided
    // us with a row block. We need to inform the server how many rows
    // this call to _fetch_into has produced.
    rb_out->num_rows = 0;

    // Read through result set, and for each input row, calculate the
relative
    // variance and set the output row values.

    while ( ( r_in = mapvar_get_next_row( tctx, state ) ) != NULL ) {

        // Gather all the input data from the current input row.
        cd_timestamp_in = &(r_in->column_data[0]);
        cd_stocksym_in = &(r_in->column_data[1]);
        cd_day1Price_in = &(r_in->column_data[2]);
        cd_day2Price_in = &(r_in->column_data[3]);

        // Get the current output row from the output row block data.
      a_v4_extfn_row   &r_out = rb_out->row_data[ rb_out->num_rows ];

        // Get the locations of the column data for the current output
row.
     a_v4_extfn_column_data     &cd_stocksym_out = r_out.column_data[
0 ];
```

```
     a_v4_extfn_column_data     &cd_timestamp_out = r_out.column_data[
1 ];
     a_v4_extfn_column_data     &cd_variance_out = r_out.column_data[
2 ];

        // Set the output column data from the input column data
        memcpy( cd_stocksym_out.data, cd_stocksym_in->data,
cd_stocksym_in->max_piece_len );
        memcpy( cd_timestamp_out.data, cd_timestamp_in->data,
cd_timestamp_in->max_piece_len );

        // Calculate weighted variance
        variance = *( double * )( cd_day2Price_in->data );
        variance = variance - *( double * )( cd_day1Price_in->data );
        variance = variance / state->priceDiffAvg;

        // Set the weighted variance in the output column.
        memcpy( cd_variance_out.data, &variance, sizeof(variance) );

        // Advance the number of output rows.
        rb_out->num_rows++;

        // If out of room in output row block, exit.  Next fetch_into
will
        // resume with a new row block.
        if (rb_out->num_rows >= rb_out->max_rows) {
            LOG_MESSAGE( tctx->proc_context, "mapvar_fetch_into: end
of output row block " );
            break;
        }
    }

    // If we produced any rows, return true.
    LOG_MESSAGE( tctx->proc_context, "Exit mapvar_fetch_into" );
    return( rb_out->num_rows > 0 );
}

/*****************************************************************
 * mapvar_close closes the input result set and frees the state object
 * to clean up at the end of the call to the TPF.
 *****************************************************************/
static short UDF_CALLBACK mapvar_close(
     a_v4_extfn_table_context *tctx)
/***********************************/
{
    mapvar_state *    state = NULL;

    LOG_MESSAGE( tctx->proc_context, "Enter mapvar_close" );

    // Retrieve the state that was saved in user_data
    state = (mapvar_state *)tctx->user_data;
```

```
    // Close the result set.
    if( state->rs && !tctx->proc_context->close_result_set( tctx-
>proc_context, state->rs ) ) {
        // Send an error to the client if we could not close the
        // result set.
      tctx->proc_context->set_error(
      tctx->proc_context,
      17001,
      "Error: Could not close result set on input table." );
    }

    // Free the memory for the state using the a_v4_extfn_proc_context
    // function free.
    tctx->proc_context->free( tctx->proc_context, state );
    tctx->user_data = NULL;

    LOG_MESSAGE( tctx->proc_context, "Exit mapvar_close" );
    return 1;
}


/*****************************************************************
 * mapvar_describe specifies the metadata for the TPF: data types of
 * columns of input and output tables.
 *****************************************************************/
static void UDF_CALLBACK mapvar_describe(
     a_v4_extfn_proc_context *ctx )
/***************************************/
{
    a_sql_int32       desc_rc;

    LOG_MESSAGE( ctx, "Enter mapvar_describe" );

    // The following describes will ensure that the schema defined
    // by the user matches the schema supported by this TPF
    // This is achieved by telling the server what our schema is
    // using describe_xxxx_set methods.
    if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {

      a_sql_data_type       type       = DT_NOTYPE;
      a_sql_uint32          width      = 0;
      a_sql_uint32          num_cols   = 0;
      a_sql_uint32          num_parms  = 0;

      // Inform the server that we support a single input
      // parameter.
      num_parms = 1;
      desc_rc = ctx->describe_udf_set
          ( ctx,
            EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS,
            &num_parms,
            sizeof( num_parms ) );
```

```
      // Checks the return code and sets an error if the
      // describe was unsucessful for any reason.
      UDF_CHECK_DESCRIBE( ctx, desc_rc );

      // Inform the server that the type of input parameter 1 is a
TABLE
      type = DT_EXTFN_TABLE;
      desc_rc = ctx->describe_parameter_set
          ( ctx,
            1,
            EXTFNAPIV4_DESCRIBE_PARM_TYPE,
            &type,
            sizeof( type ) );

      UDF_CHECK_DESCRIBE( ctx, desc_rc );

      // Inform the server that the input table should have 4 columns.
      num_cols = 4;
      desc_rc = ctx->describe_parameter_set
          ( ctx,
            1,
            EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS,
            &num_cols,
            sizeof( num_cols ) );

      UDF_CHECK_DESCRIBE( ctx, desc_rc );

      // Inform the server that the input table column 1 (tickTime) is
a
         // DT_FIXCHAR (should be changed to a TIME at some point).
      type = DT_FIXCHAR;
      desc_rc = ctx->describe_column_set
          ( ctx,
            1,
            1,
            EXTFNAPIV4_DESCRIBE_COL_TYPE,
            &type,
            sizeof( type ) );

      UDF_CHECK_DESCRIBE( ctx, desc_rc );

      // Inform the server that the width of column 1 (tickTime) in our
      // result set is 6.
      width = 6;
      desc_rc = ctx->describe_column_set
          ( ctx,
            1,
            1,
            EXTFNAPIV4_DESCRIBE_COL_WIDTH,
            &width,
            sizeof( width ) );
```

```
      UDF_CHECK_DESCRIBE( ctx, desc_rc );

      // Inform the server that the type of input table column 2
         // (stockSymbol) is a DT_FIXCHAR;
      type = DT_FIXCHAR;
      desc_rc = ctx->describe_column_set
          ( ctx,
            1,
            2,
            EXTFNAPIV4_DESCRIBE_COL_TYPE,
            &type,
            sizeof( type ) );

      UDF_CHECK_DESCRIBE( ctx, desc_rc );

      // Inform the server that the width of column 2 (stockSymbol) in
our
      // result set is 10.
      width = 10;
      desc_rc = ctx->describe_column_set
          ( ctx,
            1,
            2,
            EXTFNAPIV4_DESCRIBE_COL_WIDTH,
            &width,
            sizeof( width ) );

      UDF_CHECK_DESCRIBE( ctx, desc_rc );

      // Inform the server that the input table column 3 (day1Price) is
a
         // DT_DOUBLE
      type = DT_DOUBLE;
      desc_rc = ctx->describe_column_set
          ( ctx,
            1,
            3,
            EXTFNAPIV4_DESCRIBE_COL_TYPE,
            &type,
            sizeof( type ) );

      UDF_CHECK_DESCRIBE( ctx, desc_rc );

      // Inform the server that the input table column 4 (day2Price) is
a
         // DT_DOUBLE
      type = DT_DOUBLE;
      desc_rc = ctx->describe_column_set
          ( ctx,
            1,
            4,
```

```
            EXTFNAPIV4_DESCRIBE_COL_TYPE,
            &type,
            sizeof( type ) );

      UDF_CHECK_DESCRIBE( ctx, desc_rc );

      // Inform the server that the number of columns in our
      // result set is 3.
      num_cols = 3;
      desc_rc = ctx->describe_parameter_set
          ( ctx,
            0,
            EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS,
            &num_cols,
            sizeof( num_cols ) );

      UDF_CHECK_DESCRIBE( ctx, desc_rc );

      // Inform the server that the type of column 1 (stockSymbol) in
our
      // result set is DT_FIXCHAR.
      type = DT_FIXCHAR;
      desc_rc = ctx->describe_column_set
          ( ctx,
            0,
            1,
            EXTFNAPIV4_DESCRIBE_COL_TYPE,
            &type,
            sizeof( type ) );

      UDF_CHECK_DESCRIBE( ctx, desc_rc );

      // Inform the server that the width of column 1 (stockSymbol) in
our
      // result set is 10.
      width = 10;
      desc_rc = ctx->describe_column_set
          ( ctx,
            0,
            1,
            EXTFNAPIV4_DESCRIBE_COL_WIDTH,
            &width,
            sizeof( width ) );

      UDF_CHECK_DESCRIBE( ctx, desc_rc );

      // Inform the server that the type of column 2 (tickTime) in our
      // result set is DT_FIXCHAR.
        // (should be changed to a TIME).
      type = DT_FIXCHAR;
      desc_rc = ctx->describe_column_set
          ( ctx,
```

```
            0,
            2,
            EXTFNAPIV4_DESCRIBE_COL_TYPE,
            &type,
            sizeof( type ) );

      UDF_CHECK_DESCRIBE( ctx, desc_rc );

      // Inform the server that the width of column 2 (tickTime) in our
      // result set is 6.
      width = 6;
      desc_rc = ctx->describe_column_set
          ( ctx,
            0,
            2,
            EXTFNAPIV4_DESCRIBE_COL_WIDTH,
            &width,
            sizeof( width ) );

      UDF_CHECK_DESCRIBE( ctx, desc_rc );

      // Inform the server that the type of column 3 (stockVariance) in
our
      // result set is DT_DOUBLE.
      type = DT_DOUBLE;
      desc_rc = ctx->describe_column_set
          ( ctx,
            0,
            3,
            EXTFNAPIV4_DESCRIBE_COL_TYPE,
            &type,
            sizeof( type ) );

      UDF_CHECK_DESCRIBE( ctx, desc_rc );
    }
    if( ctx->current_state == EXTFNAPIV4_STATE_OPTIMIZATION ) {
      // Inform the server that a rewind will be required on the input
table.
      a_sql_byte      rewind_required = 1;
      desc_rc = ctx->describe_parameter_set
          ( ctx,
            1,
            EXTFNAPIV4_DESCRIBE_PARM_TABLE_REQUEST_REWIND,
            &rewind_required,
            sizeof( rewind_required ) );

      UDF_CHECK_DESCRIBE( ctx, desc_rc );
         LOG_MESSAGE( ctx, "mapvar_describe: requesting rewind" );
    }
}

/*****************************************************************
```

```
 * mapvar_evaluate is executed after mapvar_describe and tells the
server
 * what functions are being implemented.
 ****************************************************************/
static void UDF_CALLBACK mapvar_evaluate(
     a_v4_extfn_proc_context *ctx,
       void *args_handle )
/***************************************/
{
    an_extfn_value    result_table = { &mapvar_table,
                           sizeof( mapvar_table ),
                           sizeof( mapvar_table ),
                           DT_EXTFN_TABLE };

    LOG_MESSAGE( ctx, "Enter mapvar_evaluate" );

    // Tell the server what functions table functions are being
    // implemented and how many columns are in our result set.
    ctx->set_value( args_handle, 0, &result_table );

    LOG_MESSAGE( ctx, "Exit mapvar_evaluate" );
}
```

Appendix: "Reduce" code

```
// Copyright (c) 2011 Sybase, Inc.
// All rights reserved. All unpublished rights reserved.
// ************************************************
// This sample code is provided AS IS, without warranty or liability
// of any kind.
//
// You may use, reproduce, modify and distribute this sample code
// without limitation, on the condition that you retain the foregoing
// copyright notice and disclaimer as to the original Sybase, Inc.
code.

#include <stdio.h>
#include <string.h>
#include "extfnapiv4.h"
#include "udf_utils.h"

/*****************************************************************
 * This TPF example implements a function to compute the largest
 * absolute, weighted price variance of a stock over a set of times.
 * The input to the TPF are rows with the following columns (output
from
 * call to TPF mapvar_tpf - source file: mapvar.cxx):
 *
 * stockSymbol stockSymbol weightedVariance
 *
 * The TPF receives data partitioned by stockSymbol.  Here is some
example
 * pipe delimited data showing the weighted relative price variance
for a
 * stock at each half hour interval of a trading day:
 *
 * ACME|9:30|-0.16260162601626|
 * ACME|10:00|5.93548387096774|
 * ACME|10:30|4.591836734693876|
 * ACME|11:00|-4.36868686868687|
 * ACME|11:30|4.029850746268656|
 * ACME|12:00|4.543147208121826|
 * ACME|12:30|4.413265306122449|
 * ACME|1:00|-3.775510204081633|
 * ACME|1:30|2.857142857142858|
 * ACME|2:00|-2.837837837837838|
 * ACME|2:30|-2.656249999999999|
 * ACME|3:00|1.44444444444444|
 * ACME|3:30|-1.060606060606061|
 * ACME|4:00|0.9090909090909906|
 *
 * The logic of this TPF is to read through the input result set, take
the
```

```
 * absolute value of the weighted variance, find the maximum value,
and output
 * a single row for the particular stock with the absolute maximum
weighted
 * variance.
 *
 *          max ( abs ( weightedVariance ) )
 *
 * The SQL required to create this procedure is:
 *
 * CREATE OR REPLACE PROCEDURE redvar_tpf( IN stockTickTable
 *   TABLE( inStockSymbol CHAR(10), inTickTime CHAR(6), inVariance
DOUBLE ) )
 *   RESULT( StockSymbol CHAR(10), TickTime CHAR(6), MaxStockVariance
DOUBLE )
 *   EXTERNAL NAME 'redvar@mapreduce';
 *
 * A SELECT statement to invoke this function looks like this:
 *
 * select * from redvar_tpf
 * (
 *   TABLE
 *   (
 *        select outStockSymbol, outTickTime, variance
 *        from MyTempTable
 *   )
 *   over ( partition by outStockSymbol )
 * );
 *
 * This TPF is meant to be invoked with input that is the output of
 * another TPF: mapvar_tpf (source file: mapvar.cxx).  That TPF
calculates
 * the weighted price variance of a stock at each half hour interval
of a
 * trading day.
 *
 * A SELECT statement to invoke the redvar_tpf function with input
from
 * the output of mapvar_tpf looks like this:
 *
 * SELECT redvar_tpf.StockSymbol, redvar_tpf.MaxStockVariance
 * from redvar_tpf (
 *  TABLE (
 *     SELECT mapvar_tpf.StockSymbol, mapvar_tpf.TickTime,
mapvar_tpf.StockVariance
 *     FROM mapvar_tpf (
 *         TABLE (
 *            select tickTime, stockSymbol, day1Price, day2Price
 *            from TickTable
 *         )
 *         over ( partition by tickTime )
 *     )
```

```
 *  )
 *  over ( partition by mapvar_tpf.StockSymbol )
 * )
 * ORDER BY redvar_tpf.MaxStockVariance DESC;
 *
 ****************************************************************/

// Forward declarations.
static void UDF_CALLBACK redvar_describe(
    a_v4_extfn_proc_context *ctx );

static void UDF_CALLBACK redvar_evaluate(
    a_v4_extfn_proc_context *ctx,
    void *args_handle );

static short UDF_CALLBACK redvar_open(
    a_v4_extfn_table_context * tctx );

static short UDF_CALLBACK redvar_fetch_into(
    a_v4_extfn_table_context *tctx,
    a_v4_extfn_row_block *rb);

static short UDF_CALLBACK redvar_close(
    a_v4_extfn_table_context *tctx);


/*
 * This defines the a_v4_extfn_proc descriptor that is used by the
 * server to determine what functions this TPF is implementing.
 */
static a_v4_extfn_proc redvar_descriptor =
{
    NULL,        // _start_extfn
    NULL,        // _finish_extfn
    redvar_evaluate,  // _evaluate_extfn
    redvar_describe,  // _describe_extfn
    NULL,        // _leave_state_extfn
    NULL,        // _enter_state_extfn
    NULL,        // Reserved: must be NULL
    NULL         // Reserved: must be NULL
};

/*
 * This defines the main library export function entry point.  This
 * entry point is used when declaring the procedure in the
 * EXTERNAL NAME clause
 */
extern "C"
a_v4_extfn_proc * SQL_CALLBACK redvar()
/***************************************/
{
    return &redvar_descriptor;
```

```
}

/*
 * This defines the a_v4_extfn_table_func descriptor that is used by
 * the server to determine row data will be retreieved.
 */
static a_v4_extfn_table_func redvar_table_funcs =
{
    redvar_open, // _open_extfn
    redvar_fetch_into,     // _fetch_into_extfn
    NULL,        // _fetch_block_extfn
    NULL,        // _rewind_extfn
    redvar_close,     // _close_extfn
    NULL,        // Reserved: must be NULL
    NULL         // Reserved: must be NULL
};

static a_v4_extfn_table redvar_table = {
    &redvar_table_funcs,   // Table function descriptor
    3                      // number_of_columns (output): stockSymbol,
tickTime, Variance
};

/*
 * This structure will be used to retain state information.  It will
 * be allocated during the _open_extfn method and freed during the
 * _close_extfn method.
 */
struct redvar_state {
    a_v4_extfn_table_context *   rs;
    a_v4_extfn_row_block *      rb_in;
    unsigned int      rb_current_row;
    bool                        rs_processed;
};

/*****************************************************************
 * Implementation of the a_v4_extfn_table_func functions
 *****************************************************************/

/*****************************************************************
 * redvar_open allocates a state object to hold state information for
 * the duration of the call to the TPF, and opens the input result
set.
 *****************************************************************/
static short UDF_CALLBACK redvar_open(
     a_v4_extfn_table_context * tctx )
/*************************************/
{
    an_extfn_value         value;
    redvar_state *         state           = NULL;
    a_v4_extfn_table_context *   rs        = NULL;
```

```
    LOG_MESSAGE( tctx->proc_context, "Enter redvar_open" );

    // Allocate memory for the state variable using the
a_v4_extfn_proc_context
    // function alloc.
    state = (redvar_state *)
      tctx->proc_context->alloc( tctx->proc_context,
                         sizeof( redvar_state ) );

    // Initialize state variables.
    state->rs = NULL;
    state->rb_in = NULL;
    state->rb_current_row = 0;
    state->rs_processed = 0;

    // Read in the value of the input parameter and store it away in a
    // state object.  Save the state object in the context.
    if( !tctx->proc_context->get_value( tctx->args_handle,
                           1,
                           &value ) ) {

    // Send an error to the client if we could not get the value.
    tctx->proc_context->set_error(
         tctx->proc_context,
         17001,
         "Error: Could not get the value of parameter 1" );

      return 0;
    }

    // Open a result set for the input table.
    if( !tctx->proc_context->open_result_set( tctx->proc_context,
                                ( a_v4_extfn_table * )value.data,
                                &rs ) ) {
      // Send an error to the client if we could not open the result
      // set.
      tctx->proc_context->set_error(
          tctx->proc_context,
          17001,
          "Error: Could not open result set on input table." );

      return 0;
    }

    // Save pointer to the open result set.  Will need it during
fetch_into.
    state->rs = rs;

    // Save the state on the context
    tctx->user_data = state;

    LOG_MESSAGE( tctx->proc_context, "Exit redvar_open" );
```

```
    return 1;
}

/*****************************************************************
 * redvar_get_next_row is an iterator that returns each row from the
 * input result set.  It may read through multiple row blocks until
 * the result set is drained.
 *****************************************************************/
static a_v4_extfn_row * redvar_get_next_row(
     a_v4_extfn_table_context *tctx,
       redvar_state * state)
/*****************************************/
{
    a_v4_extfn_table_context *   rs         = state->rs;
    a_v4_extfn_row *            r_in        = NULL;

    if ( !( state->rb_in ) || ( state->rb_current_row == state->rb_in-
>num_rows ) ) {
        if( !( rs->fetch_block( rs, &( state->rb_in ) ) ) ) {
            // No more input data.  Done.
            LOG_MESSAGE( tctx->proc_context, "redvar_get_next_row: no
more input data" );
            return NULL;
        }
        state->rb_current_row = 0;
    }
    r_in = &(state->rb_in->row_data[state->rb_current_row]);
    state->rb_current_row++;
    return r_in;
}

/*****************************************************************
 * redvar_fetch_into reads through the input result set rows, takes
 * the absolute value of each weighted variance, keeps track of the
 * maximum value, and outputs a single output row for the stock with
 * the maximum absolute price weighted variance of the price
difference
 * between the day2Price and the day1Price.
 *****************************************************************/
static short UDF_CALLBACK redvar_fetch_into(
     a_v4_extfn_table_context *tctx,
     a_v4_extfn_row_block *rb_out)
/*****************************************/

{
    redvar_state *              state = (redvar_state *)tctx-
>user_data;
    a_v4_extfn_row *            r_in        = NULL;
    a_v4_extfn_column_data *    cd_timestamp_in  = NULL;
    a_v4_extfn_column_data *    cd_stocksym_in   = NULL;
    a_v4_extfn_column_data *    cd_variance_in   = NULL;
    double            variance  = 0;
```

```
    double            maxAbsVariance  = 0;
    char              s[20];

    LOG_MESSAGE( tctx->proc_context, "Enter redvar_fetch_into" );

    // The server provided row block structure contains a max_rows
    // field.  This field is the maximum number of rows that this row
    // block can handle.  This function will generate only one output
row,
    // however.
    rb_out->num_rows = 0;

    // Get the first output row from the output row block data.
    a_v4_extfn_row    &r_out = rb_out->row_data[ 0 ];

    // Get the locations of the column data for the first output row.
    a_v4_extfn_column_data  &cd_stocksym_out = r_out.column_data[ 0 ];
    a_v4_extfn_column_data  &cd_timestamp_out = r_out.column_data[ 1 ];
    a_v4_extfn_column_data  &cd_variance_out = r_out.column_data[ 2 ];

    // Read through result set, and find the maximum of the absolute
value of
    // the variances for the particular stock.

    while ( ( r_in = redvar_get_next_row( tctx, state ) ) != NULL ) {

        // We need to inform the server how many rows this call to
fetch_into
        // has produced.  There is input data, so it will generate
one.
        rb_out->num_rows = 1;

        // Gather all the input data from the current input row.
        cd_stocksym_in = &(r_in->column_data[0]);
        cd_timestamp_in = &(r_in->column_data[1]);
        cd_variance_in = &(r_in->column_data[2]);

        // Set the stock symbol in the output row from the input row.
        memcpy( cd_stocksym_out.data, cd_stocksym_in->data,
cd_stocksym_in->max_piece_len );

        // Get the variance from the input row.
        variance = *( double * )( cd_variance_in->data );

        // Get absolute value of variance.
     if ( variance < 0 ) variance = 0.0 - variance;

        // If absolute variance greater than current max, set current
max to
        // absolute variance.  (Also set the corresponding timestamp
in the
        // output row.)
```

```
        if ( variance > maxAbsVariance )  {
            maxAbsVariance = variance;
            memcpy( cd_timestamp_out.data, cd_timestamp_in->data,
cd_timestamp_in->max_piece_len );
            memcpy( cd_variance_out.data, &maxAbsVariance,
sizeof(maxAbsVariance) );
        }
    }

    // If we produced any rows, return true.
    LOG_MESSAGE( tctx->proc_context, "Exit redvar_fetch_into" );
    return( rb_out->num_rows > 0 );
}

/*****************************************************************
 * redvar_close closes the input result set and frees the state object
 * to clean up at the end of the call to the TPF.
 *****************************************************************/
static short UDF_CALLBACK redvar_close(
     a_v4_extfn_table_context *tctx)
/***********************************/
{
    redvar_state *    state = NULL;

    LOG_MESSAGE( tctx->proc_context, "Enter redvar_close" );

    // Retrieve the state that was saved in user_data
    state = (redvar_state *)tctx->user_data;

    // Close the result set.
    if( state->rs && !tctx->proc_context->close_result_set( tctx-
>proc_context, state->rs ) ) {
        // Send an error to the client if we could not close the
        // result set.
      tctx->proc_context->set_error(
      tctx->proc_context,
      17001,
      "Error: Could not close result set on input table." );
    }

    // Free the memory for the state using the a_v4_extfn_proc_context
    // function free.
    tctx->proc_context->free( tctx->proc_context, state );
    tctx->user_data = NULL;

    LOG_MESSAGE( tctx->proc_context, "Exit redvar_close" );
    return 1;
}


/*****************************************************************
 * redvar_describe specifies the metadata for the TPF: data types of
```

```
 * columns of input and output tables.
 ************************************************************/
static void UDF_CALLBACK redvar_describe(
     a_v4_extfn_proc_context *ctx )
/*************************************/
{
    a_sql_int32       desc_rc;

    LOG_MESSAGE( ctx, "Enter redvar_describe" );

    // The following describes will ensure that the schema defined
    // by the user matches the schema supported by this TPF
    // This is achieved by telling the server what our schema is
    // using describe_xxxx_set methods.
    if( ctx->current_state == EXTFNAPIV4_STATE_ANNOTATION ) {

      a_sql_data_type       type       = DT_NOTYPE;
      a_sql_uint32          width      = 0;
      a_sql_uint32          num_cols   = 0;
      a_sql_uint32          num_parms  = 0;

      // Inform the server that we support a single input
      // parameter.
      num_parms = 1;
      desc_rc = ctx->describe_udf_set
          ( ctx,
            EXTFNAPIV4_DESCRIBE_UDF_NUM_PARMS,
            &num_parms,
            sizeof( num_parms ) );

      // Checks the return code and sets an error if the
      // describe was unsucessful for any reason.
      UDF_CHECK_DESCRIBE( ctx, desc_rc );

      // Inform the server that the type of input parameter 1 is a
TABLE
      type = DT_EXTFN_TABLE;
      desc_rc = ctx->describe_parameter_set
          ( ctx,
            1,
            EXTFNAPIV4_DESCRIBE_PARM_TYPE,
            &type,
            sizeof( type ) );

      UDF_CHECK_DESCRIBE( ctx, desc_rc );

      // Inform the server that the input table should have 3 columns.
      num_cols = 3;
      desc_rc = ctx->describe_parameter_set
          ( ctx,
            1,
            EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS,
```

```
            &num_cols,
            sizeof( num_cols ) );

      UDF_CHECK_DESCRIBE( ctx, desc_rc );

      // Inform the server that the type of input table column 1
         // (stockSymbol) is a DT_FIXCHAR;
      type = DT_FIXCHAR;
      desc_rc = ctx->describe_column_set
          ( ctx,
            1,
            1,
            EXTFNAPIV4_DESCRIBE_COL_TYPE,
            &type,
            sizeof( type ) );

      UDF_CHECK_DESCRIBE( ctx, desc_rc );

      // Inform the server that the width of column 1 (stockSymbol) in
our
      // result set is 10.
      width = 10;
      desc_rc = ctx->describe_column_set
          ( ctx,
            1,
            1,
            EXTFNAPIV4_DESCRIBE_COL_WIDTH,
            &width,
            sizeof( width ) );

      UDF_CHECK_DESCRIBE( ctx, desc_rc );

      // Inform the server that the input table column 2 (tickTime) is
a
         // DT_FIXCHAR (should be changed to a TIME at some point).
      type = DT_FIXCHAR;
      desc_rc = ctx->describe_column_set
          ( ctx,
            1,
            2,
            EXTFNAPIV4_DESCRIBE_COL_TYPE,
            &type,
            sizeof( type ) );

      UDF_CHECK_DESCRIBE( ctx, desc_rc );

      // Inform the server that the width of column 2 (tickTime) in our
      // result set is 6.
      width = 6;
      desc_rc = ctx->describe_column_set
          ( ctx,
            1,
```

```
            2,
            EXTFNAPIV4_DESCRIBE_COL_WIDTH,
            &width,
            sizeof( width ) );

      UDF_CHECK_DESCRIBE( ctx, desc_rc );

      // Inform the server that the input table column 3 (variance) is
a
         // DT_DOUBLE
      type = DT_DOUBLE;
      desc_rc = ctx->describe_column_set
          ( ctx,
            1,
            3,
            EXTFNAPIV4_DESCRIBE_COL_TYPE,
            &type,
            sizeof( type ) );

      UDF_CHECK_DESCRIBE( ctx, desc_rc );

      // Inform the server that the number of columns in our
      // result set is 3.
      num_cols = 3;
      desc_rc = ctx->describe_parameter_set
          ( ctx,
            0,
            EXTFNAPIV4_DESCRIBE_PARM_TABLE_NUM_COLUMNS,
            &num_cols,
            sizeof( num_cols ) );

      UDF_CHECK_DESCRIBE( ctx, desc_rc );

      // Inform the server that the type of column 1 (stockSymbol) in
our
      // result set is DT_FIXCHAR.
      type = DT_FIXCHAR;
      desc_rc = ctx->describe_column_set
          ( ctx,
            0,
            1,
            EXTFNAPIV4_DESCRIBE_COL_TYPE,
            &type,
            sizeof( type ) );

      UDF_CHECK_DESCRIBE( ctx, desc_rc );

      // Inform the server that the width of column 1 (stockSymbol) in
our
      // result set is 10.
      width = 10;
      desc_rc = ctx->describe_column_set
```

```
          ( ctx,
            0,
            1,
            EXTFNAPIV4_DESCRIBE_COL_WIDTH,
            &width,
            sizeof( width ) );

      UDF_CHECK_DESCRIBE( ctx, desc_rc );

      // Inform the server that the type of column 2 (tickTime) in our
      // result set is DT_FIXCHAR.
        // (should be changed to a TIME).
      type = DT_FIXCHAR;
      desc_rc = ctx->describe_column_set
          ( ctx,
            0,
            2,
            EXTFNAPIV4_DESCRIBE_COL_TYPE,
            &type,
            sizeof( type ) );

      UDF_CHECK_DESCRIBE( ctx, desc_rc );

      // Inform the server that the width of column 2 (tickTime) in our
      // result set is 6.
      width = 6;
      desc_rc = ctx->describe_column_set
          ( ctx,
            0,
            2,
            EXTFNAPIV4_DESCRIBE_COL_WIDTH,
            &width,
            sizeof( width ) );

      UDF_CHECK_DESCRIBE( ctx, desc_rc );

      // Inform the server that the type of column 3 (maxVariance) in
our
      // result set is DT_DOUBLE.
      type = DT_DOUBLE;
      desc_rc = ctx->describe_column_set
          ( ctx,
            0,
            3,
            EXTFNAPIV4_DESCRIBE_COL_TYPE,
            &type,
            sizeof( type ) );

      UDF_CHECK_DESCRIBE( ctx, desc_rc );
    }
}
```

```
/*****************************************************************
 * redvar_evaluate is executed after redvar_describe and tells the
server
 * what functions are being implemented.
 *****************************************************************/
static void UDF_CALLBACK redvar_evaluate(
     a_v4_extfn_proc_context *ctx,
       void *args_handle )
/***************************************/
{
    an_extfn_value    result_table = { &redvar_table,
                           sizeof( redvar_table ),
                           sizeof( redvar_table ),
                           DT_EXTFN_TABLE };

    LOG_MESSAGE( ctx, "Enter redvar_evaluate" );

    // Tell the server what functions table functions are being
    // implemented and how many columns are in our result set.
    ctx->set_value( args_handle, 0, &result_table );

    LOG_MESSAGE( ctx, "Exit redvar_evaluate" );
}
```