R on Netezza's TwinFin Appliance

by Łukasz Bartnik, Netezza
lbartnik@netezza.com

Netezza & TwinFin

TwinFin, called also a "Data Warehouse Appliance" or "Analytic Appliance", is a *massively parallel processing* (MPP) *database management system* (DBMS). What distinguishes it among other DBMSes, is the fact that it was designed from scratch, including the special hardware using *field programmable gate arrays* (FPGAs), in order to attain the best performance and avoid the pitfalls of parallel data processing.

Unlike *TwinFin*, other database platforms were designed initially to work on single computers – at most multiprocessor, and bringing the parallel processing to them causes many problems, arising from the different software architecture, etc. This is obviously the case always when a big and complicated computer system is to be redesigned and adapted to a new hardware platform.

The main concepts behind *TwinFin* are:

- *On-Stream Analytics™* – which can be expressed as "bring algorithms to data" instead of bringing data to algorithms; this means that *TwinFin* comes with a set of mechanisms and interfaces which support designing and writing parallel versions of algorithms utilizing the specific capabilities of this platform
- no indices supporting SQL queries – all data processing is done on-the-fly and takes a huge advantage of the FPGAs to filter out and extract only the desired data
- intelligent query optimizer – which prepares an execution plan with minimal effort put to move data between the cluster nodes; the level of expertise required to implement such optimizer is substantial, and this is one of the biggest advantages of the whole platform

Being a cluster, *TwinFin* consists of a number of nodes. One, called *Host*, is the *master* node user connects to from his client computer; SQL interpreter & query optimizer work on the *Host*, also the SQL stored procedures are executed on this node. *Slave* nodes, called SPUs (*Snippet Processing Units*), are capable of *executing* SQL queries compiled on *Host,* UDXs (*User Defined Functions* and *Aggregates* implemented in C++), they also are the actual place where the data is stored.

TwinFin & R

The *TwinFin*-R approach is quite straightforward. Since the data processing is handled in parallel, it is obvious that also R has to comply with this paradigm. We have implemented a set of either entirely new or new versions of previously existing mechanisms:

- *apply* & *tapply* – which are used to run a user-defined function on each row or each group of rows given a grouping column, respectively
- *nzrun*, *nzfilter*, etc. – which are different, specialized (or generalized) versions of the two above (more information below)

These functions are used with *nz.data.frame* – a Netezza's version of a well-known R *data.frame* class. Basically, *nz.data.frame* is a pointer to a table which resides on the *TwinFin*. It implements a number of methods for extracting a subset of its data, gathering meta-info, similar to the *data.frame* ones, and working with parallel data-processing algorithms.

Sample code:

```
 > library(nzr)
 > nzconnect("user", "password", "host", "database")
 > library(rpart)
 > data(kyphosis)
 # this creates a table out of kyphosis data.frame
 # and sends its data to TwinFin
 > invisible(as.nz.data.frame(kyphosis))
 > nzQuery("SELECT * FROM kyphosis")
    KYPHOSIS AGE NUMBER START
 1    absent  71      3     5
 2    absent 158      3    14
 3   present 128      4     5
[ cut ]
 # now create a nz.data.frame
 > k <- nz.data.frame("kyphosis")
 > as.data.frame(k)
    KYPHOSIS AGE NUMBER START
 1    absent  71      3     5
 2    absent 158      3    14
 3   present 128      4     5
[ cut ]
 > nzQuery("SELECT * FROM kyphosis")
   COUNT
 1    81
```

R on *TwinFin* – internals

The *"R on TwinFin"* package is built on top of RODBC: each of its functions/methods invoked in R client is translated into SQL query and run directly on *TwinFin*. If a user-defined function has to be applied on a *TwinFin* table, it is serialized and sent to *TwinFin* as a string literal.

There is a low-level API available on *TwinFin*, which is hidden by function like *nzapply*, *nzfilter*, etc. When a user-defined code is run in parallel on *TwinFin*, on each node (SPU) a new R session is started, which then proceeds with the code execution. Each such sessions has to communicate with the DBMS process using the aforementioned low-level API to fetch & send data, report error, etc. Below, are some of its functions:

- *getNext()* – fetches new row from the appliance
- *getInputColumn(index)* – return the value of the column specified by its *index*
- *setOutput(index,value)* – sets the output column *value*
- *outputResult()* – sends the current output row to the appliance
- *columnCount()* - returns the number of input columns
- *userError()* – reports an error

These functions might be used directly when *nzrun* is used to run a user-defined code/function, or indirectly, through a high-level API, when *nzapply* or *nztapply* is used. The two following examples show the same task approached in both ways.

```
 > ndf <- nz.data.frame("kyphosis")
 > fun <- function(x) {
 >   while(getNext()) {
 >     v <- inputColumn(0)
 >     setOutput(0, v ^ 2)
 >     outputResult()
 >   }
 > }
 > nzrun(ndf[,2], fun)
```

Example 1: low-level API, nzrun

```
 > ndf <- nz.data.frame("kyphosis")
 > nzapply(ndf[,2], NULL, function(x) x[[1]]^2)
```

Example 2: high-level API, nzapply

R on *TwinFin* – a bit more sophisticated

Given is a *TwinFin* table with transactions data; its columns are *shop_id, prod_1, prod_2, prod_3*.The first column determined the shop where the transaction took place, the other three store the total amount spent on three different product types. The goal is, for each shop, to build a linear price model of the form:

```
prod_1 ~ prod_2 + prod_3
```

This can be accomplished with *nztapply*, where the grouping column will be *shop_id*:

```
> transactions <- nz.data.frame("transactions")
> prodlm <- function(x) {
>   return(lm(prod_1 ~ prod_2 + prod_3, data=x)$coefficients)
> }
> nztapply(transactions, shop_id, prodlm)
```

This performs the linear-model-building procedure for each group determined by the *shop_id* column. Groups are processed independently, and if they reside on different nodes (SPUs) due to the data distribution specified for the *TwinFin* table, then the computations can take place in parallel.

The *transactions* table can be generated with the function given in Appendix A.

Decision Trees – interfacing *TwinFin* algorithms in R

Since R implementation of computation-intensive algorithms might not be efficient enough, there is a number of algorithms whose design and implementation (in SQL and C++) takes into consideration properties specific to *TwinFin*.

One of them is the *Decision Trees* algorithm. It requires three datasets: training, testing, and pruning. Then, a R wrapper *dectree2* is called, which starts the computations on *TwinFin*. Object of class *dectree2* is returned as the result.

```
> adultTrain <- nz.data.frame("adult_train")
> adultTest  <- nz.data.frame("adult_test")
> adultPrune <- nz.data.frame("adult_prune")
> tr <- dectree2(income~., data=adultTrain, valtable=adultPrune,
                 minsplit=1000, id="id", qmeasure="wAcc")
```

This can be then printed (Example 3) or plotted (see Fig. 1):

```
 > print(tr)
node), split, n, deviance, yval, (yprob)
      * denotes terminal node

  16) root 0 0 small ( 0.5 0.5 ) *
           8) education_num < 10 0 0 small ( 0.5 0.5 )
         4) capital_gain < 5013 0 0 small ( 0.5 0.5 )
       1) NANA 0 0 small ( 0.5 0.5 )
     2) marital_status=Married-civ-spouse 0 0 small ( 0.5 0.5 )
          17) education_num > 7 0 0 small ( 0.5 0.5 )
            34) age < 35 0 0 small ( 0.5 0.5 ) *
               140) hours_per_week < 34 0 0 small ( 0.5 0.5 ) *
            35) age > 35 0 0 small ( 0.5 0.5 )
[ cut ]
```

Example 3: printing a TwinFin decision tree

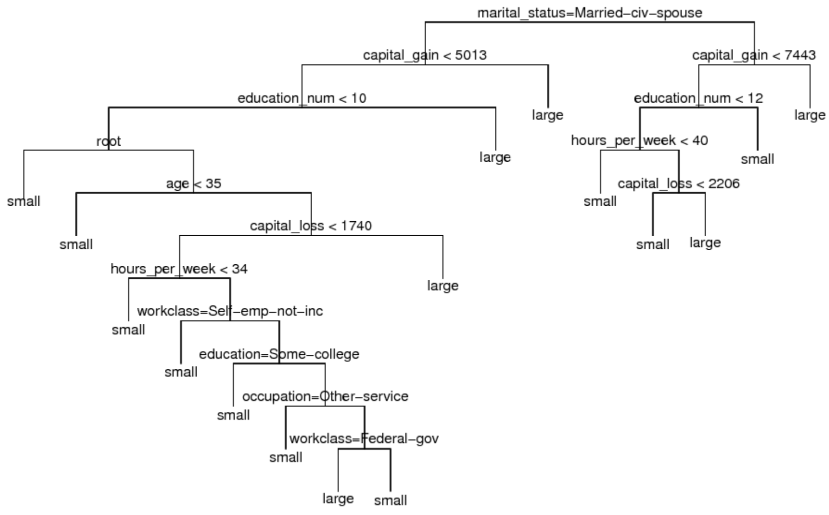

Figure 1: Example plot of an `dectree2` *object*

Appendix A

Generating data for the *nztapply* example in Section 4.

```
gendata <- function(prodno = 3, shopsno = 4, tno = 300) {
    trans <- data.frame(matrix(NA, shopsno*tno, 4))
    names(trans) <- c("shop_id", "prod_1", "prod_2", "prod_3")
    r <- 1
    for (s in seq(shopsno)) {
        sigma <- matrix(NA, prodno, prodno)
        for (t in seq(prodno)) {
            x <- seq(prodno-t+1)+t-1
            sigma[t,x] <- sigma[x,t] <- runif(length(x),max=0.75)
        }
        diag(sigma) <- 1
        chsigma <- t(chol(sigma))
        for (r in seq(tno)+tno*(s-1))
            trans[r,] <- c(s,abs(chsigma %*% rnorm(prodno)))
    }
    return(trans)
}
```

Bibliography

1. *http://www.netezza.com/data-warehouse-appliance-products/twinfin.aspx*
2. *http://www.netezza.com/documents/whitepapers/streaming_analytic_white_paper.pdf*
3. *http://www.netezza.com/analystReports/2006/bloor_100906.pdf*