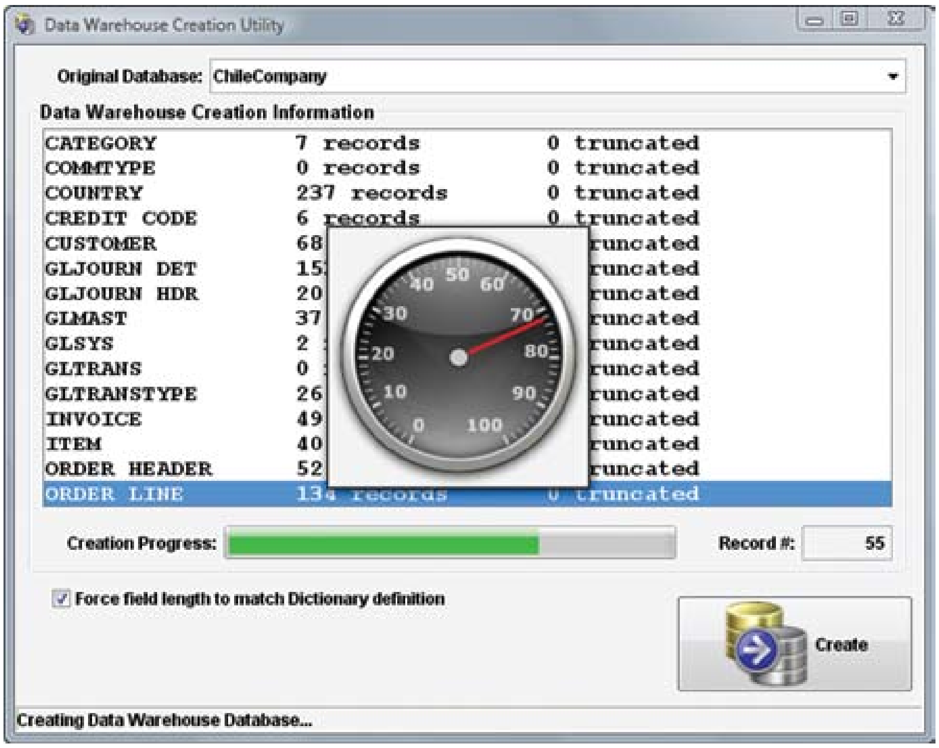
Solving the Data Warehousing Dilemma with BBj's Newest DBMS Features Part II

By Nick Decker

A previous *Advantage* article, Solving the Data Warehousing Dilemma with BBj's Newest DBMS Features, explored the concept of data warehousing, ranging from simple definitions to use cases and the steps necessary to implement such a solution. The last step of the solution involved real-time updates to the data warehouse version of the corporate database. The BASIS DBMS's trigger capability is instrumental in this process, as it provides developers with the ability to configure the system to modify both the operational database and the data warehouse in concert. While the previous article discussed the roles of the triggers and gave a high-level overview, this article delves into the actual code employed to keep both sets of data in sync at all times and provides a walk-through of the entire process via the data warehouse demo programs.

Laying the Groundwork – Creating the Data Warehouse Database

Before digging into the actual trigger code, we will run a couple of the demo programs that set up the data warehouse environment. The first program, the **Data Warehouse Creation Utility** demo†, simplifies the creation of the offline copy of the original database. The utility allows the selection of any BBj-defined legacy database and makes a new ESQL-based data warehouse version of it. **Figure 1** shows the program in action, making a data warehouse version of the BASIS ChileCompany database.


Data Warehouse Creation Utility
Original Database: ChileCompany
Data Warehouse Creation Information

```
CATEGORY          7 records        0 truncated
COMMTYPE          0 records        0 truncated
COUNTRY           237 records      0 truncated
CREDIT CODE       6 records        0 truncated
CUSTOMER          68                truncated
GLJOURN DET       15                truncated
GLJOURN HDR       20                truncated
GLMAST            37                truncated
GLSYS             2                 truncated
GLTRANS           0                 truncated
GLTRANSTYPE       26                truncated
INVOICE           49                truncated
ITEM              40                truncated
ORDER HEADER      52                truncated
ORDER LINE        134 records      0 truncated
```

Creation Progress:
Record #: 55
Force field length to match Dictionary definition
Create
Creating Data Warehouse Database...

Figure 1. The Data Warehouse Creation Utility

The utility program starts off by determining where the original database resides using the GET TABLE INFO command to locate the first table in the database. After determining the location, it creates the new database at the same directory level as the original, appending the __DataWarehouse string to the target directory name to clearly differentiate the original database from the data warehouse version.

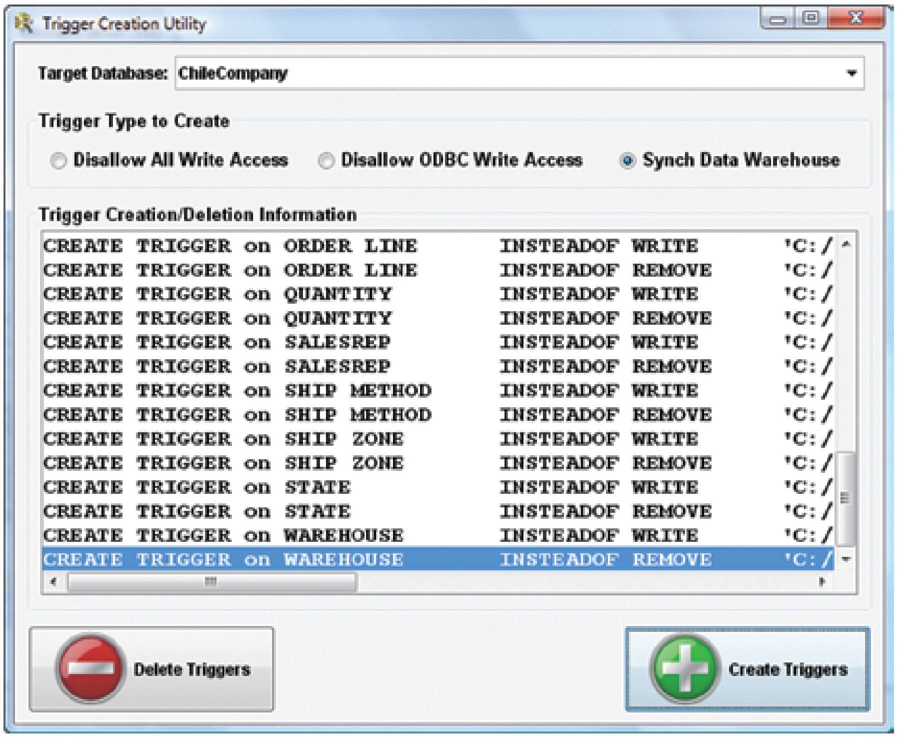
After creating the new database, the utility program systematically makes a copy of every table that exists in the original database. As part of the process, it also iterates through each table's records and makes a copy in the data warehouse database. After the process is complete, a new ESQL-based database named `ChileCompany_DataWarehouse` results that is a record-for-record copy of the original ChileCompany database.

continued...

Nick Decker
Engineering Supervisor

Keeping the Two Databases in Sync

The next demo program, **Create Triggers for Data Warehouse demo**†, simplifies the process of defining the triggers for the ChileCompany database. Creating the triggers is a relatively simple task for a couple of reasons. First, the triggers used to keep the databases in sync are written generically, allowing the same trigger to be used for every table in the database. Second, the program only has to define two triggers for each table – an INSTEADOF_WRITE trigger and an INSTEADOF_REMOVE trigger. After all, one only needs to keep track of write access to the original database in order to keep the two in sync. In other words, READ operations are not significant because they do not alter the table's contents. But WRITE and REMOVE operations both change the table's data so triggers are necessary to alter those types of operations. **Figure 2** shows the result of running the utility program and details the SQL CREATE TRIGGER command that it used for each table.



Target Database: ChileCompany

Trigger Type to Create

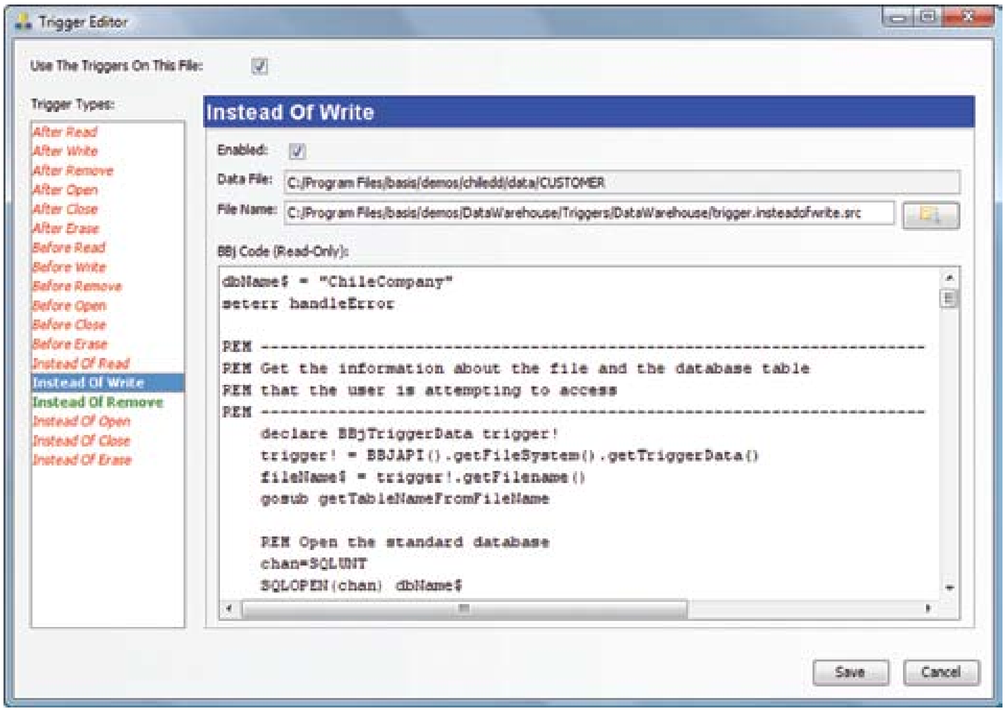
Disallow All Write Access [ ] Disallow ODBC Write Access [ ] [✓ Synch Data Warehouse]

Trigger Creation/Deletion Information

```
CREATE TRIGGER on ORDER LINE        INSTEADOF WRITE      'C:/
CREATE TRIGGER on ORDER LINE        INSTEADOF REMOVE     'C:/
CREATE TRIGGER on QUANTITY          INSTEADOF WRITE      'C:/
CREATE TRIGGER on QUANTITY          INSTEADOF REMOVE     'C:/
CREATE TRIGGER on SALESREP          INSTEADOF WRITE      'C:/
CREATE TRIGGER on SALESREP          INSTEADOF REMOVE     'C:/
CREATE TRIGGER on SHIP METHOD       INSTEADOF WRITE      'C:/
CREATE TRIGGER on SHIP METHOD       INSTEADOF REMOVE     'C:/
CREATE TRIGGER on SHIP ZONE         INSTEADOF WRITE      'C:/
CREATE TRIGGER on SHIP ZONE         INSTEADOF REMOVE     'C:/
CREATE TRIGGER on STATE             INSTEADOF WRITE      'C:/
CREATE TRIGGER on STATE             INSTEADOF REMOVE     'C:/
CREATE TRIGGER on WAREHOUSE         INSTEADOF WRITE      'C:/
CREATE TRIGGER on WAREHOUSE         INSTEADOF REMOVE     'C:/
```

Figure 2. The Trigger Creation Utility

The program iterates through the table list for the database and issues the appropriate CREATE TRIGGER command. The command instructs the SQL engine to create the appropriate type of trigger (e.g. an INSTEADOF_WRITE trigger) for each table, referencing a pre-defined BBj program that runs when the trigger fires. A quick check in the BBj Enterprise Manager confirms that the program created the triggers correctly and even shows a preview of the associated trigger code, as shown in **Figure 3**.



Use The Triggers On This File: ✓

Trigger Types:

After Read
After Write
After Remove
After Open
After Close
After Erase
Before Read
Before Write
Before Remove
Before Open
Before Close
Before Erase
Instead Of Read
Instead Of Write
Instead Of Remove
Instead Of Open
Instead Of Close
Instead Of Erase

Instead Of Write

Enabled: ✓

Data File: C:/Program Files/basis/demos/chiledd/data/CUSTOMER

File Name: C:/Program Files/basis/demos/DataWarehouse/Triggers/DataWarehouse/trigger.insteadofwrite.src

BBj Code (Read-Only):

```
dbName$ = "ChileCompany"
seterr handleError

REM -------------------------------------------------------------------
REM Get the information about the file and the database table
REM that the user is attempting to access
REM -------------------------------------------------------------------
    declare BBjTriggerData trigger!
    trigger! = BBJAPI().getFileSystem().getTriggerData()
    fileName$ = trigger!.getFilename()
    gosub getTableNameFromFileName

    REM Open the standard database
    chan=SQLUNT
    SQLOPEN(chan) dbName$
```

Figure 3. BBj Enterprise Manager's Trigger Editor screen

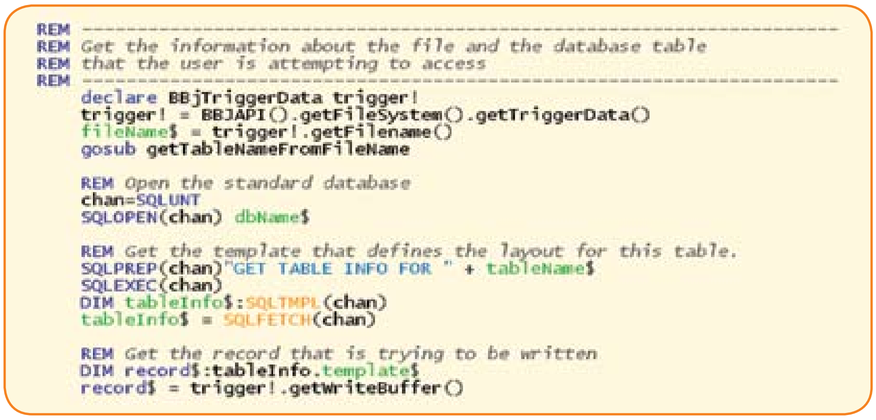
The font color and styling used for each of the trigger types in the left-hand list makes it easy to see at a glance which triggers are enabled for the table. The newly created triggers show up in bold green, whereas the other triggers are italicized and show up in red to indicate that they are not currently active.

A Closer Look at the Trigger Code

Before taking the new triggers out for a spin, take a look under the hood and see how they accomplish the task of keeping the two databases in sync. Because the two trigger programs share most of the same code

continued...

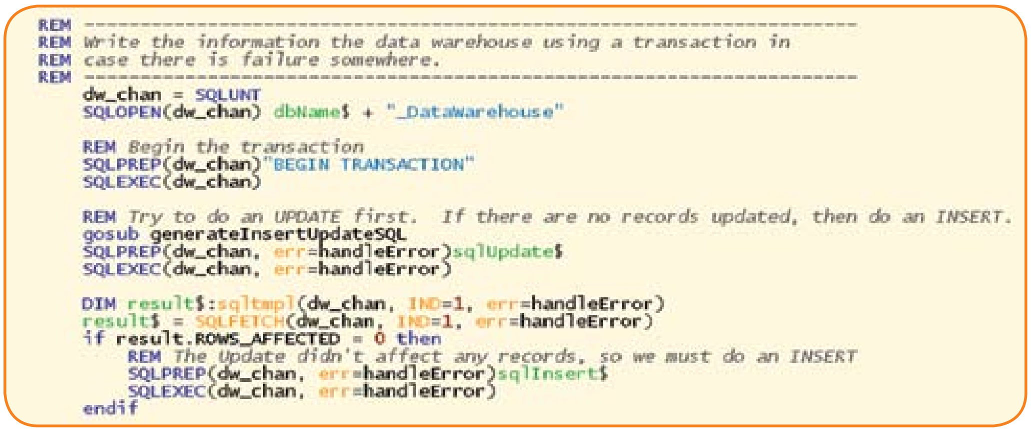
and operate similarly at a high level, we will only look at the INSTEADOF_WRITE trigger program. The first part of the program, shown in **Figure 4**, takes care of the essentials: it gets the BBjTriggerData object from the API, retrieves the filename for the table being written to, locates the physical MKEYED data file associated with the table name, and gets the record that is scheduled to be written to the file.



```
REM ---------------------------------------------------------------------
REM Get the information about the file and the database table
REM that the user is attempting to access
REM ---------------------------------------------------------------------
    declare BBjTriggerData trigger!
    trigger! = BBJAPI().getFileSystem().getTriggerData()
    fileName$ = trigger!.getFilename()
    gosub getTableNameFromFileName

    REM Open the standard database
    chan=SQLUNT
    SQLOPEN(chan) dbName$

    REM Get the template that defines the layout for this table.
    SQLPREP(chan)"GET TABLE INFO FOR " + tableName$
    SQLEXEC(chan)
    DIM tableInfo$:SQLTMPL(chan)
    tableInfo$ = SQLFETCH(chan)

    REM Get the record that is trying to be written
    DIM record$:tableInfo.template$
    record$ = trigger!.getWriteBuffer()
```

Figure 4. The first part of the INSTEADOF_WRITE trigger

With just a few lines of code, the trigger program is now prepared to update both databases as it knows about the original MKEYED data file, the data warehouse database, and the proposed modification to the designated table. The next step in the process is to modify the data warehouse table. Because the data warehouse version of the database utilizes relational ESQL tables, it supports the concept of database transactions. This piece of the puzzle is critical as it ensures that both copies of the database are updated in the same manner. For example, it would not make sense to write the new data to the original database, only to later run into an error attempting to make the corresponding change in the data warehouse version. In order for the two databases to match, all changes must be made to both of them; a change cannot be applied to only one of them. Transactions are perfect for this scenario, as we can use them to guarantee that all changes are applied. If there is a problem applying the change to either database, neither one of them will be modified – resulting in completely synchronized databases. The code that updates the data warehouse is shown in **Figure 5**.

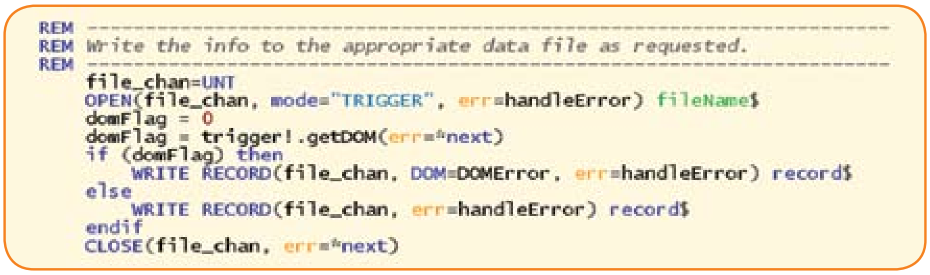


```
REM ---------------------------------------------------------------------
REM Write the information the data warehouse using a transaction in
REM case there is failure somewhere.
REM ---------------------------------------------------------------------
    dw_chan = SQLUNT
    SQLOPEN(dw_chan) dbName$ + "_DataWarehouse"

    REM Begin the transaction
    SQLPREP(dw_chan)"BEGIN TRANSACTION"
    SQLEXEC(dw_chan)

    REM Try to do an UPDATE first.  If there are no records updated, then do an INSERT.
    gosub generateInsertUpdateSQL
    SQLPREP(dw_chan, err=handleError)sqlUpdate$
    SQLEXEC(dw_chan, err=handleError)

    DIM result$:sqltmpl(dw_chan, IND=1, err=handleError)
    result$ = SQLFETCH(dw_chan, IND=1, err=handleError)
    if result.ROWS_AFFECTED = 0 then
        REM The Update didn't affect any records, so we must do an INSERT
        SQLPREP(dw_chan, err=handleError)sqlInsert$
        SQLEXEC(dw_chan, err=handleError)
    endif
```

Figure 5. The code responsible for writing the new data to the data warehouse database

The code begins by initiating a transaction before attempting to modify the database. The next step involves writing the new record to the database. Since it is using SQL to make the modification, it begins by issuing an SQL UPDATE statement. This will be appropriate if the record already exists in the table, but will not have any impact if the record does not exist in the table. To adjust for this, the program then queries the database to see if any records were affected by the UPDATE. If no rows were affected, that indicates that the record does not exist and the program reverts to an SQL INSERT statement to add the new record to the table. The actual UPDATE and INSERT statements were constructed in a subroutine that walks through the trigger's write buffer, creating the appropriate SQL statements based on the fields and their values.

If the data warehouse has been written to without incident, the program continues and attempts to make the same changes to the original database, as shown in **Figure 6**.



```
REM ---------------------------------------------------------------------
REM Write the info to the appropriate data file as requested.
REM ---------------------------------------------------------------------
    file_chan=UNT
    OPEN(file_chan, mode="TRIGGER", err=handleError) fileName$
    domFlag = 0
    domFlag = trigger!.getDOM(err=*next)
    if (domFlag) then
        WRITE RECORD(file_chan, DOM=DOMError, err=handleError) record$
    else
        WRITE RECORD(file_chan, err=handleError) record$
    endif
    CLOSE(file_chan, err=*next)
```

Figure 6. The code responsible for writing the new data to the original database

continued...

The program knows where the original data file is located and OPENs it with the special TRIGGER mode. This mode is only valid when used inside a trigger program and tells BBj to open a channel to the same open file handle that fired the trigger. The program then issues the appropriate WRITE RECORD statement to update the file, depending on whether or not the program originally responsible for causing the trigger to fire included a DOM (duplicate or missing key) mode.

Now that both databases have been updated, the trigger program can begin tying up the loose ends as shown in **Figure 7**.

```
REM -----------------------------------------------------------------------
REM If we got here, we were successful in writing to the original file,
REM and we were successful in updating the data warehouse.  All that's left
REM is to commit the change to the data warehouse.
REM -----------------------------------------------------------------------
    SQLPREP(dw_chan, err=handleError)"COMMIT"
    SQLEXEC(dw_chan, err=handleError)

    SQLCLOSE(dw_chan, err=*next)
    SQLCLOSE(chan, err=*next)
    CLOSE(file_chan, err=*next)
    END
```

Figure 7. Committing the changes to the data warehouse

As the comments indicate, if the trigger program gets to this section of the code it means that it has successfully modified both the data warehouse and the original database. The final step involves committing the transaction for the data warehouse table update. The databases are now synchronized and the trigger program ends.

```
REM -----------------------------------------------------------------------
REM Error routine - rollback the SQL operation, log the error
REM to the BBjServices.out file and THROW a custom error
REM -----------------------------------------------------------------------
handleError:

    REM Log the error to the BBjServices.err file
    msg$ = "Error #" + str(err) + " occured in " + pgm(-1) + " at line " + str(tcb(5))
    if err = 77 then msg$ = msg$ + $0d0a$ + "SQL Err: " + sqlerr(dw_chan)
    java.lang.System.err.println(msg$)

    REM Don't attempt to perform this part if the database has not been opened!
    if (dw_chan>0) then
        REM Rollback the data warehouse transaction
        SQLPREP(dw_chan)"ROLLBACK"
        SQLEXEC(dw_chan)
        SQLCLOSE(dw_chan, err=*next)
        SQLCLOSE(chan, err=*next)
        CLOSE(file_chan, err=*next)
    endif

    REM Throw a custom error
    THROW "Data could not be written to the table and replicated table", 0
```

Figure 8. The program's error handling routine

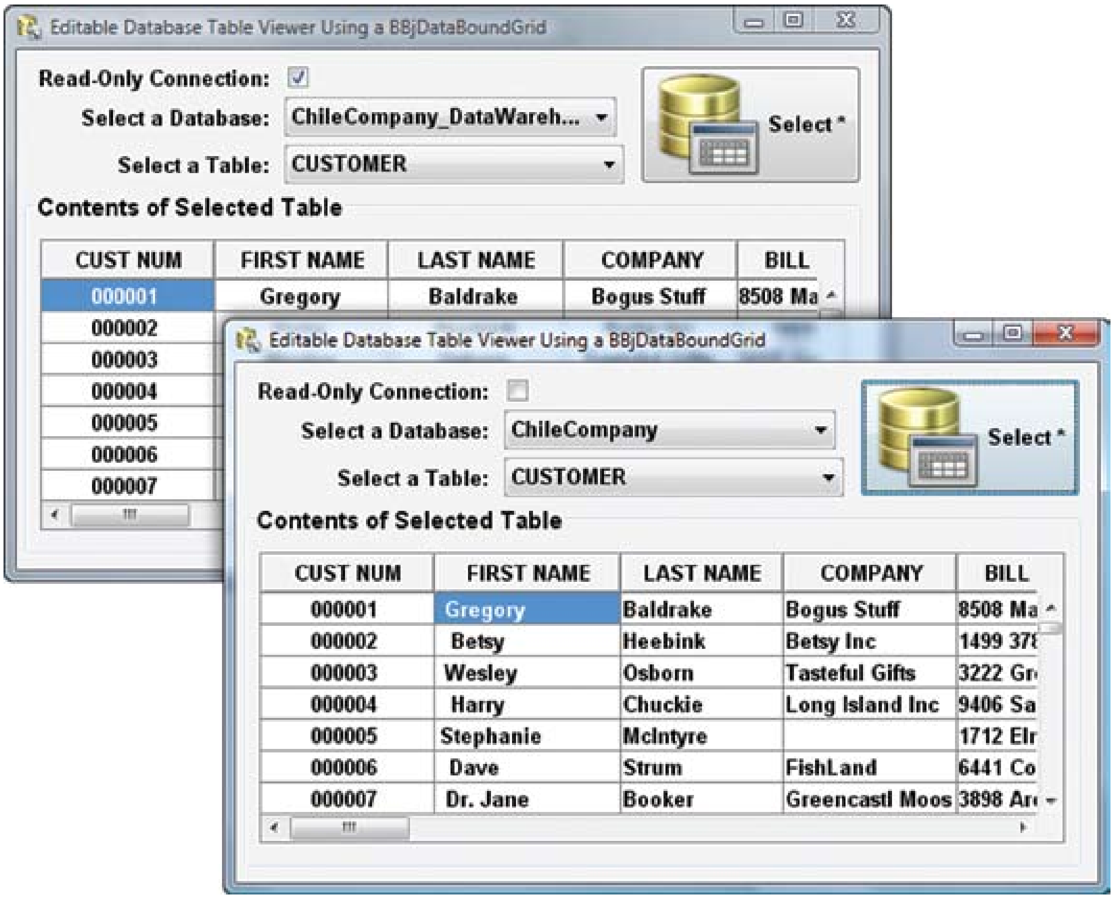
The program is diligent about checking for errors, as a failure to update the data warehouse means that it should not continue and update the original database. Likewise, a failure to update the original database presents a problem as the changes that have already been made to the data warehouse need to be reverted. If an error like this occurs, the program logs the error number, message, and program line number to the `BBjServices.out` log file. It then rolls the transaction back, effectively cancelling any changes made to the data warehouse. Finally, it uses BBj's THROW verb to cause the originating application's WRITE/UPDATE to result in an error, as shown in **Figure 8**.

Putting the Triggers to the Test

The next demo program, **Databound Grid Table Viewer** demo †, enables us to test the triggers and ensure that the two databases are kept in sync. To put the triggers to the test, run two copies of the demo program – one that is connected to the original ChileCompany database and one that is connected to the ChileCompany_DataWarehouse database shown in **Figure 9** on the next page.

Connect the first copy of the demo program to the `ChileCompany_DataWarehouse` database by selecting that database name from the drop-down list box. Ensure that the "Read-Only Connection" checkbox is selected to stop the demo from locking the database and preventing the trigger from making modifications to it. This makes sense since developers should only use the data warehouse database for querying and reporting.

continued...



Editable Database Table Viewer Using a BBjDataBoundGrid

Read-Only Connection: ☑
Select a Database: ChileCompany_DataWareh...
Select a Table: CUSTOMER
Select *

Contents of Selected Table

CUST NUM	FIRST NAME	LAST NAME	COMPANY	BILL
000001	Gregory	Baldrake	Bogus Stuff	8508 Ma
000002				
000003				
000004				
000005				
000006				
000007				

Editable Database Table Viewer Using a BBjDataBoundGrid

Read-Only Connection: ☐
Select a Database: ChileCompany
Select a Table: CUSTOMER
Select *

Contents of Selected Table

CUST NUM	FIRST NAME	LAST NAME	COMPANY	BILL
000001	Gregory	Baldrake	Bogus Stuff	8508 Ma
000002	Betsy	Heebink	Betsy Inc	1499 378
000003	Wesley	Osborn	Tasteful Gifts	3222 Gr
000004	Harry	Chuckie	Long Island Inc	9406 Sa
000005	Stephanie	McIntyre		1712 Elr
000006	Dave	Strum	FishLand	6441 Co
000007	Dr. Jane	Booker	Greencastl Moos	3898 Ar

Figure 9. Running two copies of the Databound Grid Table Viewer

Next, connect the second copy of the program to the original ChileCompany database. Uncheck the “Read-Only Connection” as this is the database that we will be making our changes to. Now we are free to make modifications to the table data. Double-clicking in the desired grid cell will put it into edit mode, allowing changes in the field data. Moving to another row in the grid updates the underlying table and the triggers take care of making the corresponding changes to the data warehouse table. To verify that the changes were propagated to the data warehouse, click the [Select *] button on the first copy of the program – it will refresh the grid, displaying the current contents of the table, which match the recently-modified table from the ChileCompany database.

Summary

By combining the BASIS DBMS’s powerful trigger functionality with its relational ESQL tables, developers can implement an accelerated and powerful data warehousing solution. Triggers ensure that the various tables remain synchronized with one another in real time and completely eliminate the need for bulk copies that tie the system up for hours overnight. Relational ESQL tables ensure that analysts can sift through a plethora of information in the quickest way possible and guarantee data integrity via SQL transactions. By combining both of these new technologies, BASIS developers can finally solve the data warehousing dilemma!

Read more about data warehousing and triggers in earlier issues of the *BASIS International Advantage*

Solving the Data Warehousing Dilemma with BBj’s Newest DBMS Features Part I
www.basis.com/advantage/mag-v11n1/DataWarehouse01adv07.pdf

Using Triggers to Maintain Database Integrity
www.basis.com/advantage/mag-v10n1/triggers.html

† Download with BBj from www.basis.com/products/bbj/download.html and select "Demos" in the Optional File section. After completing the installation, select **BBj > Demos > LaunchDock** from the BASIS folder to run the following demos:

Data Warehouse Creation Utility

Create Triggers for Data Warehouse

Databound Grid Table Viewer