ESQL Files: Constraining Your Data to Guarantee Integrity

By Nick Decker

BASIS first introduced ESQL (Exclusive SQL) tables with the release of BBj 6.0. As the name suggests, ESQL tables are only accessible via SQL statements so the traditional verbs like OPEN() and READ() do not apply to these files. Instead, developers can create these files via SQL "Create Table" statements and add or read rows by SQL "Insert" and "Select" statements. Additionally, ESQL also offers true SQL data types such as DECIMAL with a defined precision and scale, DATE, and TIMESTAMP, to name a few. ESQL tables offer a host of other features as well, such as variable length records and dynamic index creation.

Introduction

New for BBj 7.0, BASIS is adding another round of enhancements to ESQL tables. In addition to their support of industry-standard syntax and data types, ESQL tables now boast support for constraints. Simply put, constraints are rules or restrictions used to define what data is acceptable. By imposing constraints on table data, the developer ensures that the data added to the table meets all of the required criteria and assures data integrity. The database is now involved in the process of validating the potential data, so instead of allowing problematic data be saved only to raise errors in the applications at a later time, this validation process can prevent the incorrect data from being saved in the first place. Constraints put the onus of data integrity on the database itself – not the client applications. This centralization of business logic is paramount as data integrity is now controlled and managed in one centralized location - the database itself - regardless of how many data paths exist to the table (applications, utilities, web pages, ODBC/JDBC).

Primary Key Constraint

While constraints may seem a bit complex and daunting at first, it is comforting to know that we have been using them in one form or another for several years. For example, most developers are familiar with the concept of the primary key in a data file. Simply put, the primary key is a field used to identify a record uniquely. The primary key may be a single value such as a customer number or a conglomerate of other fields and sections thereof – whatever is necessary to ensure uniqueness. BBx® MKEYED files have always required a primary key just as the newer VKEYED and ESQL file types. This primary key requirement is actually one of the simpler and more fundamental constraints used when creating database tables. Following is an example of the syntax for the *primary key* constraint:

```
CREATE TABLE Constraints (CustNum integer primary key, Name varchar(50))
```

By specifying the words **`primary key`** after the declaration of the CustNum field and data type, we have applied a constraint on the column. As it turns out, because BBj requires that the first column of an ESQL table is a primary key, this constraint will be applied even it if is not explicitly stated. Therefore, the SQL statement in the example above will execute correctly if the developer removed the **`primary key`** constraint because it is implied.

Not NULL Constraint

The primary key constraint was relatively simple as it always existed and the language applied it even when the developer was not aware of it being an active constraint. The database does not imply other constraints, however, and requires explicit declaration in order to be in effect. *Not NULL* is a good example of this as it is a relatively common and frequently used constraint. By way of explanation, databases have the concept of NULL, which may seem a bit unusual at first. The easiest way to think of a NULL value is that it is simply unknown. Therefore, it is not necessarily a string or a number. Especially important is the fact that NULL is not the same as an empty string because even though an empty string may signify nothing, it is still a known value. When designing a database, the developer usually determines whether it is valid for a field to be empty or NULL. In other words, some fields may be required while others are optional. For example, it is reasonable in a customer table to expect that the customer's name is required and not NULL. Other fields, such as a secondary e-mail address, may not always exist and therefore that field is not required. The Not NULL constraint can translate these requirements when added to the table creation statement. Here is a sample SQL statement to create a table with a Name field that is required, and an e-mail field that is not:

```
CREATE TABLE Constraints (
      CustNum integer primary key,
      Name varchar(50) not null,
      Email varchar(50)
      )
```

Nick Decker
Engineering Supervisor

continued...

Similar to the primary key constraint is the *unique* constraint, which prohibits multiple rows from having the same value for the same column. In other words, it is taking the unique portion of the primary key requirement and making it available as a separate constraint for the other columns. This gets more interesting when allowing a field to be NULL as a particular column value may be NULL, but only if it does not already exist as NULL in another row in the table.

Column Constraints

Constraining the possible values of one or more columns in a table is likely to be the most widely used feature of ESQL tables. As the name implies, *column* constraints give the developer the power to constrain or filter the data that goes into a particular column. Constraints can be as simple as ensuring that a particular field always contains a value such as the Not NULL constraint. Moving up a level in complexity, constraints can also ensure that character or numeric fields match particular criteria. For example, implementing a constraint on a character field will ensure that the length of the field satisfies a predetermined requirement. Likewise, a constraint can also ensure that a particular numeric field is always greater than zero. Constraints can go further still and incorporate scalar string and numeric functions, time and date functions, custom scalar functions, and so on, when defining the rules for the column. That means that it is possible to ensure that strings are of a desired case, numbers fall within preset ranges, character fields must match an existing set, dates must be after a certain point in time, and so on.

Following are a few simple examples of column constraints:

The first example creates a table with a field called QtyOnHand to indicate how many items are in stock. This sample contains a simple *numeric* constraint on the column to ensure that the value never drops below zero.

```
CREATE TABLE Constraints (
      ItemNum integer primary key,
      QtyOnHand integer check( QtyOnHand>=0 )
      )
```

So executing an SQL statement such as:

```
insert into Constraints (ItemNum, QtyOnHand) Values ('1',2)
```

works as expected, but providing a negative quantity results in the following error:

```
insert into Constraints (ItemNum, QtyOnHand) Values ('1',-2)
UNABLE TO EXECUTE SQL STATEMENT: Problem with QTYONHAND: Column does not meet
check constraint: <value> >= 0
```

The second example creates a table with a field called LastName and contains two constraints on this field: 1) it must have at least two characters and 2) it must begin with an uppercase letter.

```
CREATE TABLE Constraints (
   CustNum integer primary key,
   LastName varchar(20)
      check( len(LastName)>1)
      check ( left(LastName,1) = ucase(left(LastName,1)) )
   )
```

Attempting to violate these constraints result in the following errors:

```
insert into Constraints (LastName) values ('P')
UNABLE TO EXECUTE SQL STATEMENT: Problem with LASTNAME: Column does not meet
check constraint: LEN(<value>) > 1

insert into Constraints (LastName) values ('paulson')
UNABLE TO EXECUTE SQL STATEMENT: Problem with LASTNAME: Column does not meet
check constraint: LEFT(<value>,1) = UCASE(LEFT(<value>,1))
```

continued...

The third example creates a table and limits the values of the ShipMethod field to a predefined set, ensuring that the shipping method is valid regardless of whether a record is inserted or updated.

```
CREATE TABLE Constraints (
   OrderNum integer primary key,
   ShipMethod varchar(5)
      check( ShipMethod in (‘FEDEX’,’UPGR’,’UPBLU’,’UPRED’) )
      )
```

Custom Column Definitions

By taking advantage of constraints, developers can create the equivalent of custom data types that are a subset of a standard SQL data type. For example, assume that a particular table has a field that contains percentages. Normally, the developer would define this field as an integer or decimal, depending on whether or not the values can have a fractional part. However, with either of these standard data types, it is possible for a value to be less than zero or greater than 100. Since the field is going to hold percentage values, by very definition, the data cannot be less than zero or greater than 100. In the past, the client application was fully responsible for ensuring this level of compliance. BBx programs would have to incorporate business logic in order to ensure that the data entered into the database was really a percentage. Check constraints relieve the client application of this duty and move it to the database level, consolidating logic and making it easier for developers to provide new front ends and alternative access points to their data, such as interactive Web pages and third party programs. By adding two column constraints to the table, one to ensure that the data is greater than or equal to zero and the other to ensure that the data is less than or equal to 100, the developer can be confident that the percentage values will always be in bounds. These check constraints have done more than ensure data integrity, though. They have worked together to create a new pseudo data type that is specific to the developer’s needs. Here is an example of what such a constraint would look like:

```
CREATE TABLE Constraints (
   ProjectID integer primary key,
   PctComplete integer check( PctComplete>=0 ) check( PctComplete<=100 )
   )
```

ROW Constraints

While the previous examples focused on the validity of a particular column, row constraints evaluate multiple columns in the row in order to determine the validity of the entire row. This means that the check may involve more than one column and may compare columns to one another in order to satisfy the constraint. Row constraints appear at the end of the CREATE TABLE statement and result in Boolean expressions (TRUE or FALSE) that determine whether the row is valid, as shown in the example below:

```
CREATE TABLE Constraints (
      ItemNum integer primary key,
      Description varchar(25),
      OurCost decimal(5,2) not null,
      SalePrice decimal(5,2) not null,
      check ( SalePrice >= OurCost )
      )
```

Notice the standalone check at the end of the statement. It ensures that the company never sells any items at a loss by guaranteeing that the sale price of each item is equal to or greater than the original cost of the item. To test the row constraint, execute the following SQL statement:

```
insert into Constraints (ItemNum, Description, OurCost, SalePrice)
   values (22,’Tortilla Press’,12.50,12.00)
```

The database correctly rejects the potential record with the following error message:

```
UNABLE TO EXECUTE SQL STATEMENT: Record does not meet check constraint:
SALEPRICE >= OURCOST
```

continued...

Looking Beyond Constraints

Computed Columns

Computed columns are another new feature that ESQL tables offer. The developer can define columns in a table to be the result of a computation based on other fields in the same table. In the past, developers usually accomplished computed columns such as these by creating a new VIEW on the table that returns the result of the computation and includes the algorithm for arriving at the answer. While this method still works, it is less desirable as the complexity of the calculation is defined in the view, not the base table itself. This means that whenever customers want access to the computed value, they must reference the view instead of the base table. This leads to added complexity and a potential for creating nested views if a particular subset of the data provided by the new view is required. The only way to accomplish this would be to replicate the view (and all of the complexity of the computed column) or create a second view based off the first view. Recreating the algorithm for the computed column is not desirable, as it would require changing multiple views if the algorithm needs adjustment. Creating views from other views is also subject to failure, as a required modification of the base view may have a domino effect, making the child views invalid.

To eliminate the problems introduced with views, create a computed column in an ESQL table. By defining the algorithm in the table creation, the developer gives the BASIS DBMS all of the information that it needs to provide users with the correct computed result based on the data that exists in each row. The algorithm is stored in a single place and the developer can create multiple views and ad hoc queries on the base table without having to replicate the computational logic. As a final benefit, the computed columns do not take up more space in the database as their values are computed on-the-fly when the data is queried.

Below is a simple example of a computed column based on numeric fields. The table contains fields for the price of the item and the quantity ordered. It also defines a computed column named Total that is the product of the Price and Quantity fields.

```
CREATE TABLE Constraints (
      OrderNum integer primary key,
      Price decimal(5,2),
      Quantity integer,
      Total as (Price * Quantity)
      )
```

To populate the table, execute the following SQL insert statement:

```
insert into Constraints (OrderNum, Price, Quantity) values (1,58.97,2)
```

Executing an SQL select from the table reveals the computed Total value:

OrderNum	Price	Quantity	Total
1	58.97	2	117.94

Computed columns are not limited to numeric fields, though. The next example creates two computed columns: FullName, based on string values in the row and RenewalDate, based on date values.

```
CREATE TABLE Constraints (
      CustNum integer primary key,
      LastName varchar(20),
      FirstName varchar(20),
      FullName as (rtrim(LastName) + ', ' + rtrim(FirstName)),
      RenewalDate as date(TimestampAdd('SQL_TSI_MONTH',3,now()))
      )
```

The FullName computed column serves a popular and often-used function – returning the full name of a customer as a single formatted field. It does so by trimming and combining the LastName and FirstName fields. The RenewalDate computed column utilizes the TimestampAdd scalar function to return a date that resolves to three months from now.

continued...

To test the computed columns, insert a record via the following SQL statement:

```
insert into Constraints (CustNum, LastName, FirstName) values
(1,'Baldrake','Gregory')
```

Executing a SELECT * from the table results in the following:

```
CustNum     LastName    FirstName   FullName          RenewalDate
1           Baldrake    Gregory     Baldrake,Gregory  12/11/06
```

Default Values

ESQL tables offer other features like default values in addition to constraints. Default values are a convenient way to ensure that certain columns populate with default data, even if this data does not exist in the SQL insert statement. Without default values, executing an insert statement that did not provide data for a particular set of columns would result in those columns being empty. However, by adding default values to the column definition of the table, the developer can guarantee that there will be a predetermined default value in the column, even if the original SQL insert statement did not provide data for that field.

To affect a read-only column, combine default values with constraints. For example, consider the following SQL statement:

```
CREATE TABLE Constraints (
      CustNum integer primary key,
      Name varchar(20) not null,
      LastModifiedBy varchar(20) default user() check( LastModifiedBy =
user() ),
      LastModifiedAt varchar(30) default now() check( LastModifiedAt = now()
)
      )
```

This statement creates a table and utilizes default values and check constraints for the last two columns, LastModifiedBy and LastModifiedAt. The purpose of these columns is to indicate which user last modified the particular record and when the modification took place. The default values utilize the system USER() and NOW() functions to set the user name and provide a timestamp for the modification. The example goes a step further and utilizes these same functions in the check constraints. Without these constraints, the two fields would still be set to the appropriate user and timestamp whenever an inserted record did not provide values for those two columns, due to the specified default values. However, it would be possible to execute an SQL insert or update that provided false information for these fields, overriding the default values and providing misleading information. To ensure that the fields contain the correct data, add the check constraints to force the values to the same as the default values, thus effectively making the two fields read-only.

To test the system, issue the following SQL insert statement that should automatically fill out the last two fields with the appropriate values:

```
insert into Constraints (CustNum,Name) values (1,'Betsy Heebink')
```

After the SQL insert, a SELECT * from the table results in:

```
CustNum  Name            LastModifiedBy   LastModifiedAt
1        Betsy Heebink   admin            2006-09-07 14:48:15.687
```

To ensure that the constraints prevent someone from providing incorrect information, issue the following SQL insert:

```
insert into Constraints (CustNum,Name, LastModifiedBy)
      values (2,'Betsy Heebink','Nobody')
```

Since the value of the LastModifiedBy field does not match the check constraint of USER(), the insert fails with the following message:

```
UNABLE TO EXECUTE SQL STATEMENT: Problem with LASTMODIFIEDBY:
Column does not meet check constraint: <value> = USER()
```

continued...

Identity Fields

Identity fields are another addition to ESQL tables. Like their big brother sequences, identity fields are a capability offered by the database and are meant to provide an automatically generated unique numeric value. However, unlike sequences, identity fields are table-specific whereas sequences function at a database level. This reduced scope results in a simpler concept that is easy to implement. Instead of requiring a separate SQL statement to define the sequence, defining identity fields occurs during table creation. Additionally, the database generates the values and automatically populates the table; whereas sequences need to be referenced in the SQL insert statement. Think of identity fields as a “hands-off” mechanism that simplifies the arduous task of determining unique numerical values for a primary key such as a **CustNum** field in the example below.

```
CREATE TABLE Constraints (
      CustNum integer identity(100,10) primary key,
      Name varchar(20) not null
      )
```

The **CustNum** field is not only the primary key, but it is also an identity field with a seed value of 100 and an increment value of 10. That means that the database populates the **CustNum** field automatically with unique values that will start at 100 and increase by 10.

Summary

ESQL tables boast several attractive new additions and enhancements in the upcoming BBj 7.0, which includes Identity Fields, Default Values, and Constraints. These features allow developers to define their database tables to their specifications, ensuring data integrity and improving application reliability. By placing the responsibility of the validity of the data on the database itself, rather than the applications, developers can avert data related issues before committing anything to the database.