An Introduction to Row Level Security

March 2002

Table of Contents

3 **Introduction to Row Level Security**
3 **The Row Level Security Paradigm**
4 **Row Level Security in Practice**
4 **Brio Intelligence and Row Level Security**
5 **Publishing in a Secure Environment**
5 **Securing the Security Information**
6 **Row Level Security Tables**
6 The BRIOSECO Table
7 The BRIOSECG Table
8 The BRIOSECR Table
9 **Row Level Security Examples**
9 Preface to the Examples
10 Defining the Users and Groups
11 Dealing with "The Rest of the Users"
12 Overriding Constraints
12 Cascading Restrictions
13 **Other Important Facts**
13 Custom SQL
13 Limits
14 Naming
15 **Table and Column Names**
15 **Alias Names**

Introduction to Row Level Security

"A little knowledge is a dangerous thing."

This oft-quoted sentence (no known attribution and probably a misquote) is, as is just as often stated, misunderstood. Instead of suggesting that having knowledge is apt to lead to trouble and so one is better off not knowing, it really acknowledges that actions based on incomplete ("little" instead of "total") knowledge is where trouble lies. Business Intelligence is predicated on the ability to build the foundation of knowledge necessary to make informed decisions.

Still, organizations possess data that to members of that organization can serve as a distraction to their main functions and duties. A classic example is payroll information. Knowing what coworkers are paid does not necessarily serve to improve an individual's ability to perform the role for which he or she was hired. Instead, it could well serve to steer the individual into a misguided and inappropriate understanding of their place in the organization and consequently affect their performance. Knowing what their salary is compared to coworkers does not explain how it came to be that way.

Another but less obvious example is sales data. It is useful for most everyone to know how the company as a whole is performing so that objectives can be adjusted as needed. However, accessing sales performance information specific to a particular salesman or region can color an individual salesperson's perception of their relative worth to the organization, or it can reveal compensation information for individuals in a goal-oriented organization, which is just as sensitive as payroll data.

Imparting appropriately complete knowledge is the goal of the Brio Intelligence feature called "Row Level Security". Properly implemented, it gives individuals in the organization the information necessary to make informed decisions, but it restricts access to information the organization considers sensitive. For example, managers need payroll information on their direct reports, but those direct reports do not need that information for their peers. Managers do not need to know the details of payroll information for other departments at their same level within the organization, but they might find summary payroll information across the organization useful as a means to recruit and retain qualified employees.

The Row Level Security Paradigm

Most database administrators understand the concept of row level security. Returning to the payroll data example, all detailed compensation data on the employees of an organization is stored in the same table(s) within the database. To do otherwise would greatly complicate the implementation of payroll applications when it comes time to produce the paychecks. Typically, in a normalized database, some column within this table can be used as a "limit", either directly or by a join to another table with its own set of limits, to restrict access to the data within the table based on the identification of the user accessing the data. Following the payroll example, an employee ID often identifies the sensitive compensation data. A join to a separate employee information table, which contains non-compensation related information such as home address and title, would include a department number. A manager would be limited to details on the employees for her/his particular department.

Row level security, implemented at the database level, is often done by means of a view. To an application, accessing a view is no different than accessing a table. However, the view is instantiated based on the appropriate limits, effectively hiding data from "prying eyes". Coupled with the GRANT and corresponding REVOKE data definition statements available with the prevalent Relational Database Management Systems (RDBMS), the base tables can be made inaccessible to most users, and the views on that data, filtered based on user identification, made accessible instead. Multiple views may sometimes be required to fully implement a security scheme, depending on how the tables are defined and how the information contained therein must be shared. For instance, a different view for managers versus those in human resources might be required.

Row Level Security in Practice

The prevalent RDBMS offer a great deal of flexibility in providing a secure yet appropriately deployed access to information important to the organization. Along with the aforementioned views and GRANT/REVOKE capabilities, some offer the means to group users by their "role" within the organization, and then facilitate granting/revoking access to these roles, which then filters down to its members. Moreover, the database itself is the most appropriate place to put security controls, at the source and commonly applied to any access from any sort of software.

However, the extent to which these capabilities are utilized is dependent on many factors. Often, it is a matter of resources. Effectively implementing and maintaining an appropriate security system is potentially costly, especially in the absence of an ability to define roles. Sometimes, as organizations come to depend on standardized software solutions to their business needs instead of always building these solutions in-house, such security measures stand in the way of successfully deploying the application, especially if multiple packages need access to the same data with varying tolerances for these measures. Other times, these systems attempt to impose their own built-in solutions to the security issue, ignoring or conflicting with the controls in the database.

The net effect of these issues, and there are many other similar issues, is to take information out of the hands of the many, leaving the critical day-to-day decisions to a select few.

Brio Intelligence and Row Level Security

For a business to succeed, decisions must be made on accurate, timely, and appropriately complete information. Brio Intelligence has helped a great many organizations be successful. It works well in cases where the security controls were placed in the database, and in many cases could implement access based on user identification even when the database did not enforce it directly with views. However, a more flexible approach to the issue of data security was needed. Starting with Brio Intelligence version 6.6, organizations can build stronger and more robust security measures into their Business Intelligence reporting solutions, not only within their organization, but also beyond, as they begin to share information on the "extranet" with vendors and customers alike.

The Brio approach to data security is server based. Both the Broadcast Server and especially the OnDemand Server are designed to fully implement a secure data access platform. The non-server based clients do not participate in this security mechanism; server-based solutions are deployed far more commonly, and for the Designer client in particular, its users need access beyond that of most users to effectively create the Executive Information Systems (dashboards), and analytic reports required by the majority of the data consumers. In addition, the security information can be placed in a centralized location for the servers (the repository). For the desktop clients, it would in some cases need to be dispersed to multiple databases and maintained separately.

To effectively control access, the servers key off the user's identification when connecting to it. For the OnDemand Server, this is the logon user name. For the Broadcast Server, this is the name of the user who scheduled the job. When an OnDemand Server client schedules a job, these names will be the same. When a desktop client schedules a job, it is the database identifier used to connect to the server's repository database.

Beyond this user name, the servers make no assumptions about the user's place within the organization. A security system can be built entirely independently of any existing grouping of users and new groupings can be defined in lieu of existing ones. This is especially important for existing deployments of Brio Intelligence servers, where the current user groups were often more arbitrary and not rigorously defined in any data-related way. Where those groups were defined with data security as a primary goal, the row level security can take full advantage of the existing structure. In many cases, it is flexible enough to work within existing database role definitions, third-party software security systems, and the like.

The system is designed in such a way as to not impose any significant performance penalty. The security information is collected at the time the user opens a document from the server's repository, and only then if the server knows the security controls are enabled. The cost of this is the execution of a SELECT statement returning a single-row, single-column result. For locally saved documents that originated from the OnDemand Server, the security information is recollected when reconnecting to the server in case it has changed.

Publishing in a Secure Environment

A powerful feature of Brio Intelligence is the ability to take data "on the road". Once data has been extracted from the database, which is where the row level security restrictions are enforced, that data can be saved with the Brio document for offline analysis and reporting. Users who publish should be aware of the implications of their audience when publishing data and reports.

If the publication of the data is difficult to control in the current configuration of users and groups known to the server, consider the following options:

1) Publish without the detailed results of the queries, leaving only the summary charts and Pivots for the "general" audience. If they need to drill into the summary data, they will need to rerun the queries, at which time their particular security restrictions will be applied. (Even some charts and Pivots can reveal too much, so there is still a need for prudence when publishing these documents.)
2) Create the documents with OnStartup scripts to reprocess queries as the document is opened. This will always give the user only the data to which they are entitled.

All users should take similar precautions when sharing information generated from Brio Intelligence software. This includes exchanging the report files (BQY extensions) themselves by email or shared network directories, exporting the data as HTML files and publishing them to a web site, posting the data on FTP servers through the broadcast server, and creating PDF files from the reports.

Securing the Security Information

Before getting into the details of implementation, take note that the row level security feature is implemented by means of tables within the server's repository. The servers alone read this data and never update it. Brio Software recommends these tables actually be defined elsewhere in the database, and that read-only access to views on these tables be made available in the server repository. The same owner would own these views as the other repository tables. However, many users can publish documents, requiring permission to write to the rest of the repository. Only a select few should be able to update the security information. The views should select all the available columns in each of the tables needed (or convert the columns of existing tables into the columns defined for the Brio defined implementation of row level security if feasible).

As an additional protection, a WHERE clause can be added to each view definition so that only the server's user identification, by which it connects to the repository, can read the content, if the database supports it. For example, if the repository connection is made as user 'brioserver':

Database	Sample Where Clause on CREATE VIEW
DB2	WHERE USER = 'BRIOSERVER'
Oracle	WHERE USER = 'BRIOSERVER'
SQL Server (Microsoft and Sybase)	WHERE USER = 'brioserver'
	*Be aware of case sensitivity with the user name and allow that, for SQL Server, the user might be 'dbo'.

Row Level Security Tables

A total of three tables implement the row level security features of Brio Intelligence servers. As previously described, they co-reside (same owner) with the server's repository tables. For the OnDemand Server (ODS), the main table is BRIOCAT2; for the Broadcast Server, the main table is BRIOJOBS. For simplicity, the tables for both servers should also be coresident. This is a requirement if OnDemand Server users are going to be scheduling jobs or if the Broadcast Server will be registering documents to the ODS.

Because the tables can be populated in a manner appropriate to the site's requirements, no predefined user interface to maintain the tables is provided as part of the Brio Intelligence product. However, in the samples directory, a Brio document, "Row level security.bqy", can be modified to suit an ad hoc implementation of the row level security feature, and as a tutorial or test tool when setting up a production system. The only requirement is that the basic set of column definitions be retained. The sample document can be used in all cases as a reporting tool for the row level security data as the servers see it.

Implementing a secure data access environment using Brio's row level security features requires an understanding of SQL. First, knowing how the database relationships are defined is critical. Second, specifying the restrictions is directly translated into the SQL ultimately processed at the database.

The BRIOSECO Table

The BRIOSECO table is used to tell the server that the row level security feature is enabled. It consists of a single column, named BENABLE, defined as a single character (CHAR(1)). The server reads only the first row. If the column contains a "Y", the server enables row level security features; if it contains an "N", the server will not enforce these restrictions or even read them.

The BRIOSECG Table

The BRIOSECG table defines the users and groups that are subject to row level security restrictions. There are two columns, BUSER and BGROUP, both of varying character length (VARCHAR(n)). The maximum length, n, is not fixed by the server; set it to a practical value.

A user name is defined as the server authentication name (ODSUsername is the property of the ActiveDocument object in the Brio Object Model) when implemented in the OnDemand Server. For the Broadcast Server, it is the user who scheduled the job. If a job is scheduled by an ODS client, these will be one and the same value. Group names are arbitrary. The data security administrator is free to define these as required. When both columns of a row are populated with non-null values, the user name defined in the BUSER column is a member of the group name defined in BGROUP.

As maintained by the sample BQY file, "Row level security.bqy", when a user is added, a row is added to the table with a NULL value in the BGROUP column. When a group is added, a NULL value is stored in the BUSER column. This is a device used by the sample document to maintain the table, but it is not a requirement for operation of the server.

This table is theoretically optional. Without it, however, all users exist as single individuals; they cannot be grouped to apply a single set of restrictions to all members. For example, if VIDHYA and CHI are members of the PAYROLL group, but this relationship if not defined in BRIOSECG, then any restrictions that apply to VIDHYA that should also apply to CHI would have to be defined twice. By defining the PAYROLL group and its members, VIDHYA and CHI, the restrictions can be defined only once as applying to PAYROLL.

A group name cannot be used in BUSER; that is, groups cannot be members of other groups. Users, of course, can be members of multiple groups, and this can effectively set up a group/subgroup hierarchy. For example, a PAYROLL group might contain users Sally, Michael, Kathy, David, Bill, Paul, and Dan. Sally, Dan, and Michael are managers, and so they can be made members of a PAYROLL MANAGER group. Certain restrictions on the PAYROLL group can be "overridden" by the PAYROLL MANAGER group, and Dan, to whom Sally and Michael report, can have specific overrides to those restrictions placed explicitly on the PAYROLL MANAGER group.

Where the database supports it, and if the ODS user names correspond, as they would when the ODS uses database authentication for example, this table can be a view created from the roles this user has in the database. For example, in Oracle:

```
CREATE VIEW BRIOSECG (BGROUP, BUSER) AS
        SELECT GRANTED_ROLE, GRANTEE FROM DBA_ROLE_PRIVS
```

Note that DBA_ROLE_PRIVS is a restricted table. At a minimum, the database logon used by the server to access its repository tables needs permission to read from the table (SELECT ANY TABLE permission). Since the server reads the table under its own user name in the database, it would not be appropriate to use USER_ROLE_PRIVS instead of DBA_ROLE_PRIVS, because that view will reflect only the server's roles, not the user on whose behalf the server is operating. Again, this is an Oracle example; other RDBMS may or may not provide a similar mechanism. In some cases, depending on the database, a stored procedure could collect the role information for the users and populate a BRIOSECG table if a simple SELECT is inadequate to collect the information. This would require some means to invoke the procedure each time role definitions were changed.

If the groups presently defined in the OnDemand Server are sufficient to meet the requirements for adequate data protection, consider using the following view:

```
CREATE VIEW BRIOSECG (BGROUP, BUSER) AS
        SELECT GROUPNAME, USERNAME FROM BRIOGRP2
```

When using the database's catalog, the ODS repository, or some other means to "populate" BRIOSECG, the sample document, "Row level security.bqy", cannot be used to maintain user and group information.

A special group, PUBLIC, exists. It does not need to be explicitly defined in BRIOSECG, however. All users are members of the PUBLIC group. Any data access restriction defined against the PUBLIC group applies to every user unless explicitly overridden, as described later.

All users can be made part of a group at once by inserting a row where BUSER is 'PUBLIC' and BGROUP is that group name. While this may seem redundant, given the existence of a PUBLIC group, it offers some benefits:

1) It allows the database catalog technique described above to work. For example, in Oracle, a role can be granted to PUBLIC.
2) It allows restrictions for a group other than PUBLIC to quickly be applied to or removed from everyone in an instant.
3) It provides more flexibility when using "override" specifications as described later.

Note that restrictions are never applied against a user named PUBLIC, but only the group PUBLIC. For this reason, do not use PUBLIC as a user name. Similarly, to avoid problems, do not name a group the same as a user name.

The BRIOSECR Table

The BRIOSECR table is the heart of the row level security feature. It defines the specific restrictions to be applied to users and the groups (including PUBLIC) to which they belong. These restrictions take the form of join operations (a user cannot access a column in the employee salary table unless it is joined to the employee table), and limits (WHERE clause expressions) to be applied to either the source table (SALARY) or table(s) (EMPLOYEE) to which it is joined.

The BRIOSECR table contains the following columns:

Column Name	Column Type	Functional Use
UNIQUE_ID	INT	This column contains an arbitrary numeric value. It should be unique, and it is useful for maintaining the table by whatever means the customer chooses. The servers do not rely upon this column, and the servers never access this column. To that extent, it is an optional column but recommended. (It is required when using the sample file, "Row level security.bqy".) When the RDBMS supports it, a unique constraint or unique index should be applied to the table on this column.
USER_GRP	VARCHAR	The name of the user or the name of a group to which a user belongs. If "PUBLIC", the restrictions are applied to all queries.
SRCDB	VARCHAR, can be null	Used to identify a topic in the Data Model. (In BrioQuery, a "topic" typically corresponds to a table in the database, but it could be a view in the database.) If the physical name property of the topic is of the form name1.name2.name3, this represents name1. Most often, this represents the database in which the topic exists. This field is optional unless required by the connection in use. The most likely circumstance in which to encounter this requirement will be with Sybase or Microsoft SQL Servers where the OCE (the connection definition file) is set for access to multiple databases.
SRCOWNER	VARCHAR, can be null	Used to identify the owner/schema of the topic in the Data Model. This would be 'name2' in the three-part naming scheme shown above. If the topic property "physical name" contains an owner, then it must be used here as well.
SRCTBL	VARCHAR	Used to identify the table/relation identified by the topic in the Data Model. This is 'name3' in the three-part naming scheme.
SRCCOL	VARCHAR	Used to identify a column in a table. This is a "topic item" in Data Model terminology, and is an item that might appear on the Request line in a query built from the Data Model. In the context of the security implementation, the item named here, qualified by the preceding SRC* columns, is the object of the restrictions being defined by this row of the security table BRIOSECR. If this column contains an *, all columns in SRCTBL are restricted.
JOINDB	VARCHAR, can be null	If present, defines the database name qualifier of a table/relation that must be joined to SRCTBL.
JOINOWNR	VARCHAR, can be null	If present, defines the schema/owner name qualifier of a table/relation that must be joined to SRCTBL.
JOINTBL	VARCHAR, can be nul	If present, names the table/relation that must be joined to SRCTBL.
JOINCOLS	VARCHAR, can be null	If present, names the column name from SRCTBL to be joined to a column from JOINTBL.
JOINCOLJ	VARCHAR, can be null	If present, names the column name in JOINTBL that will be joined (always an equal join) to the column named in JOINCOLS.

Column Name	Column Type	Functional Use
CONSTRTT	CHAR(1), can be null	If present, identifies a table/relation to be used for applying a constraint (limit). This is a coded value. If the value in this column is "S", the column to be limited is in SRCTBL. If "J", a column in JOINTBL is to be limited. If the value in this column is "O", it indicates that for the current user/group, the restriction on the source column for the group/user named in column OVRRIDEG is lifted, rendering it ineffective. If this value is NULL, then no additional restriction is defined. If this value is NULL, then a join table must be specified.
CONSTRTC	VARCHAR, can be null	The column in the table/relation identified by CONSTRTT to which a limit is applied.
CONSTRTO	VARCHAR, can be null	The constraint operator, such as =, <> (not equal), etc. BETWEEN and IN are valid operators. Basically, any valid operator for the database can be supplied.
CONSTRTV	VARCHAR, can be null	The value(s) to be used as a limit. The value(s) properly form a condition that together with the content of CONSTRTC and CONSTRTO columns create valid SQL syntax for a "condition" in a WHERE clause. Subquery expressions, therefore, are allowed. Literal values should be enclosed in single quotes or whatever delimiter is needed by the database for the type of literal being defined. If the operator is BETWEEN, the "AND" keyword would separate values, etc. If :USER is used in the value, then the user name is the limit value. If :GROUP is used, all groups of which the user is a member are used as the limiting values. Both :USER and :GROUP can be specified, separated by commas. The "public" group must be named explicitly; it is not supplied by reference to :GROUP.
OVRRIDEG	VARCHAR, can be null	The name of a group or user. Used when CONSTRTT is set to "O". If the group named in OVRRIDEG has a restriction on the source element, then this restriction is effectively ignored for the user/group named in USER_GRP. SRCDB, SRCOWNER, SRCTBL, and SRCCOL as a collection must be equal between the row specifying the override and the row specifying the conditions to be overridden. (See examples.)

As suggested earlier, existing security definitions can sometimes be "translated" into the format described here. A view, stored procedure, or programmatic mechanism can be used to translate and/or populate the information needed in BRIOSECR by the Brio Intelligence servers. When these methods are used, the Brio servers require the column names and data types defined above. And again, do not attempt to use the sample document, "Row level security.bqy", to manage this information.

If a join table is specified and it does not already exist in the data model the user accesses, it will still be added to the final SQL generated to ensure the security restrictions are enforced. This process is iterative. When a table is added and the present user, either directly or by group membership, has restricted access to that added table, those restrictions will also be applied, which may mean additional tables will be added, and those restrictions will also be checked, and so on. Circular references will result in an error if they are defined.

Row Level Security Examples

Preface to the Examples

Maybe this all seems fairly complicated. To get past this rhetoric, a few examples are provided here in attempt to clarify some of these concepts. The examples are based on the sample Access database, "Brio 6.0 Sample Advanced.mdb", provided by option when installing Brio Software on a Windows platform, using the connection file

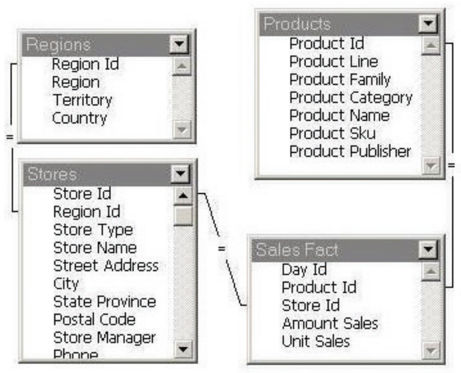
"Brio 6.0 Sample 2 – Advanced.oce". Either use a freshly installed server with its sample repository to replicate these examples or add this processing connection to an existing OnDemand Server test platform to begin learning how to use this powerful data security feature. If all else fails, use a test server with whatever tables are available as a learning tool.

The sample repository installed with the ODS comes with a predefined set of users. For these examples, the users BRIO and VIEW&PROCESS require access to data that is denied to the rest of the users. These two users both belong to the group "AMERICAS", which corresponds to a region of the same name. However, the user BRIO is a corporate officer who should be able to see all data.

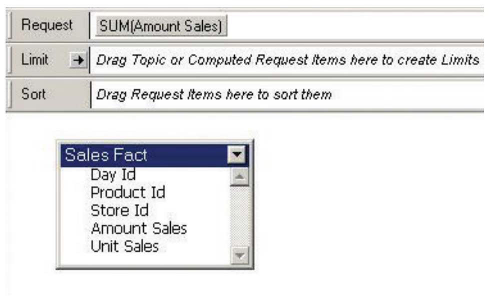
Only one piece of data will be accessed in the course of these examples: the "amount sales" column from the "sales fact" table. The examples are more far-reaching than this might seem however.

Screenshots for these examples come from the Brio Server Administrator, the document to which processing restrictions are applied, and from the sample document, "Row level security.bqy", mentioned earlier. For the screen shots from the sample document, note that the columns in the BRIOSECR table follow in a top down, left to right manner for the most part with the fields on the screen. Deviations from this will be noted where possible. In particular, though, note that the UNIQUE_ID column is not shown in this sequence of fields, appropriate to its optional role to the functionality of the software, although it is used "behind the scenes" by the sample document.

Here is the layout of the data in the database, to illustrate the possible joins as intended by the database designer:



The data model in the actual document published to the ODS appears as follows:

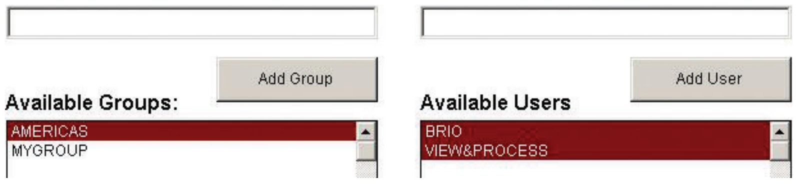

Request
SUM(Amount Sales)
Limit
Drag Topic or Computed Request Items here to create Limits
Sort
Drag Request Items here to sort them
Sales Fact
Day Id
Product Id
Store Id
Amount Sales
Unit Sales

Defining the Users and Groups

The groups defined in the sample ODS repository do not include an "AMERICAS" group. To add this group for purposes of row level security, insert a minimum of two rows into the BRIOSECG table, as follows:

BGROUP	BUSER
AMERICAS	BRIO
AMERICAS	VIEW&PROCESS

In the sample document for maintaining Row Level Security information, once the information has been added, it would look something like this when the AMERICAS group is selected:

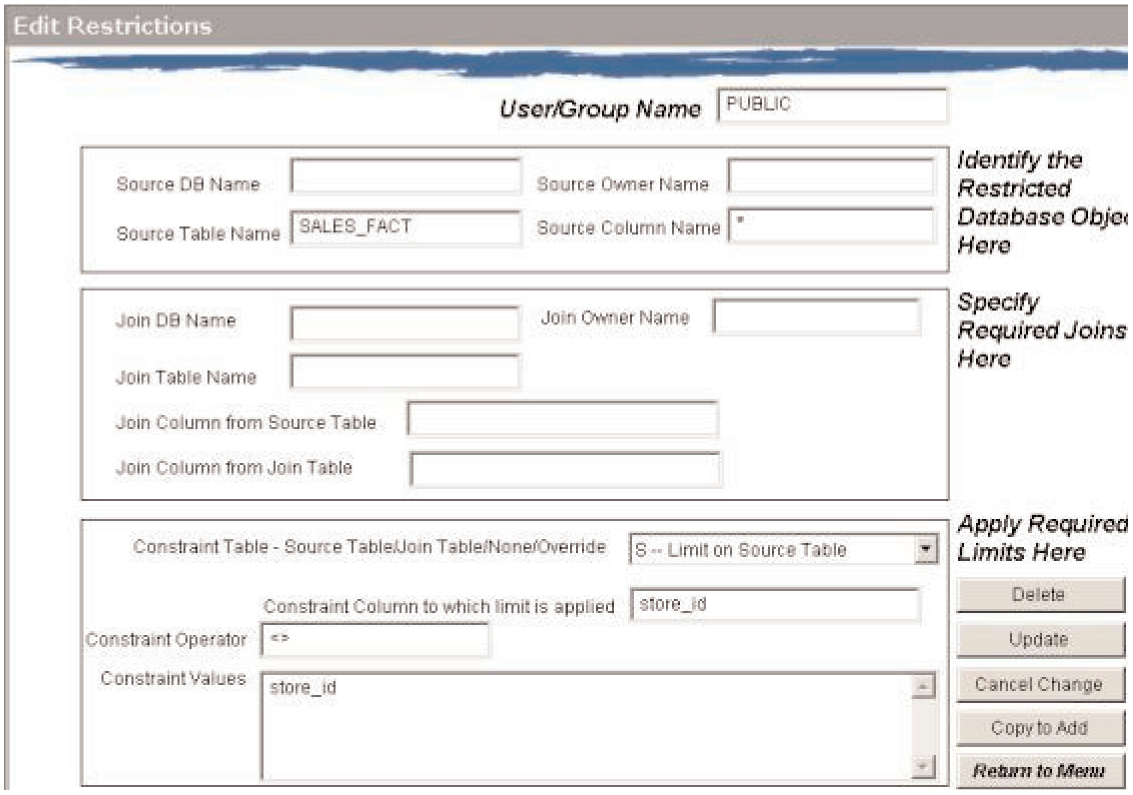


Available Groups:
AMERICAS
MYGROUP

Available Users
BRIO
VIEW&PROCESS

Dealing with "The Rest of the Users"
The requirement here was that users who were not part of the AMERICAS group should have no access to this information. There are several ways to do this, and in part, what the best way is depends on who the "rest of the users" are.

If there are what are commonly called extranet users, this probably literally means "no access"; users outside of the corporate network should not get sales data, even summary data, as this might be considered proprietary and certainly not for any potential competitors. Using the PUBIC group, restrict the entire SALES_FACT table from accessing this information by using the asterisk to reflect "all columns":


Edit Restrictions
Identify the Restricted Database Objec Here
Specify Required Joins Here
Apply Required Limits Here
Delete
Update
Cancel Change
Copy to Add
Return to Menu

User/Group Name: PUBLIC

Source DB Name:
Source Owner Name:
Source Table Name: SALES_FACT
Source Column Name: *

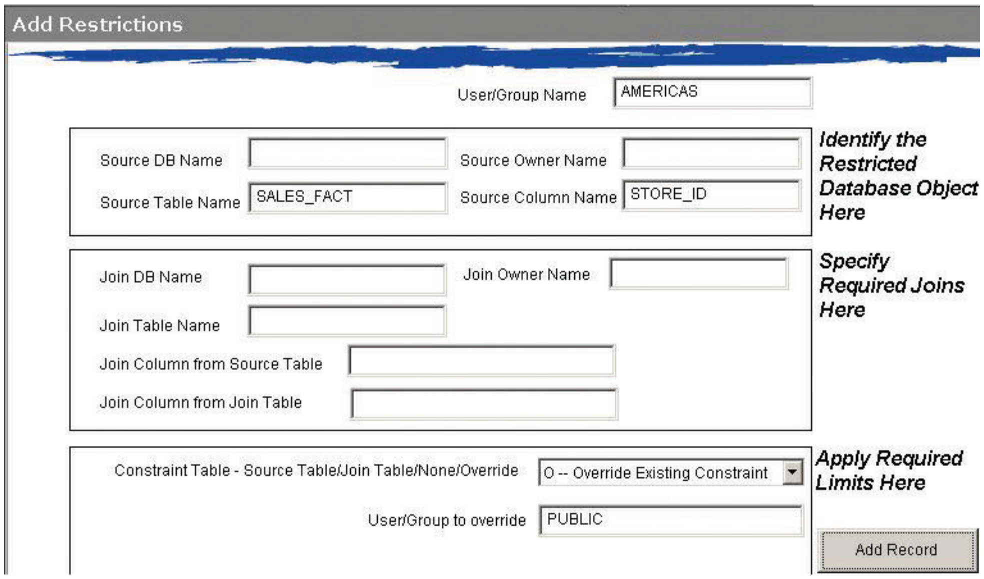
Join DB Name:
Join Owner Name:
Join Table Name:
Join Column from Source Table:
Join Column from Join Table:

Constraint Table - Source Table/Join Table/None/Override: S -- Limit on Source Table
Constraint Column to which limit is applied: store_id
Constraint Operator: <>
Constraint Values: store_id

Where there are no extranet concerns, and as stated earlier, it might be appropriate for all the employees to know how their company is doing overall, so such a blanket restriction is not recommended. Instead, restrict the use the STORE_ID column, the only means by which the sales information can be tied back to any particular store, country, region, etc. This will look identical to the case above except that STORE_ID is specified instead of an asterisk for the "Source Column Name".

Overriding Constraints

Obviously, members of the AMERICAS group are also members of PUBLIC. So, regardless of the way the PUBLIC group was restricted, those restrictions are not to be applied to the AMERICAS group for the sales information. That group might be restricted in different ways, or not at all, and the same mechanism ensures that happens while PUBLIC restrictions are in place. This is shown as follows when using the sample document, "Row level security.bqy":



Add Restrictions

User/Group Name	AMERICAS
Source DB Name	
Source Owner Name	
Source Table Name	SALES_FACT
Source Column Name	STORE_ID
Join DB Name	
Join Owner Name	
Join Table Name	
Join Column from Source Table	
Join Column from Join Table	
Constraint Table - Source Table/Join Table/None/Override	O -- Override Existing Constraint
User/Group to override	PUBLIC

Identify the Restricted Database Object Here
Specify Required Joins Here
Apply Required Limits Here
Add Record

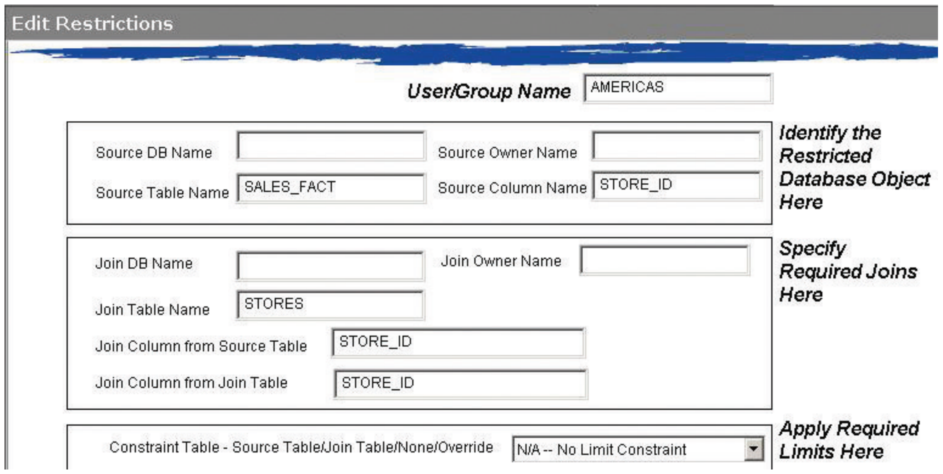
It is important to note that this only overrides PUBLIC constraints for this particular column. Restrictions on PUBLIC against other columns are still enforced against members of the AMERICAS group as well.

Cascading Restrictions

In order to give the members of the AMERICAS group access to the sales information required for the appropriate region, and only that, the query will have to include references to columns in other tables, not necessarily part of the existing data model. It is OK if those tables are already present; the row level security will function the same in either case.

As seen in the table relationships pictured above, the region information is "bridged" to the sales information by the store table. So, to implement a constraint that only sales information is available for a particular region requires two entries in the BRIOSECR table, one to join sales to stores, and one to join stores to regions. This latter case also requires a limit value for the region name. (A limit on region_id could also accomplish the same goal but isn't as readable, especially in an example. See the discussion to follow about subqueries for another perspective on limits on "ID" type columns.)

The first restriction required for this example is on the store_id column. In order to use that column, a join must be made back to the stores table. This would be specified as follows:

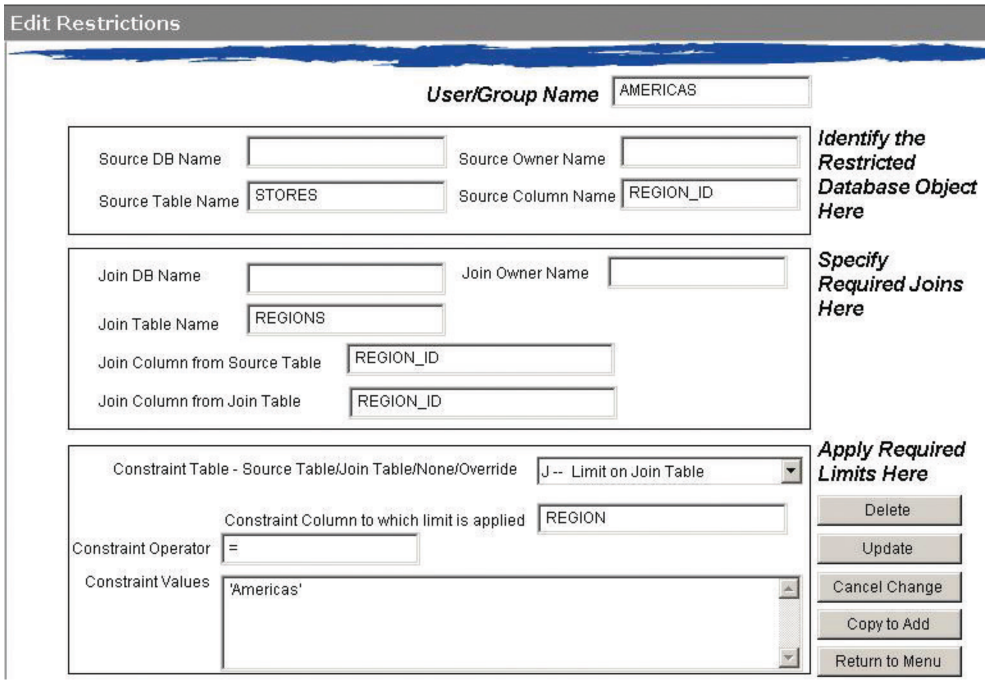


Edit Restrictions

User/Group Name	AMERICAS
Source DB Name	
Source Owner Name	
Source Table Name	SALES_FACT
Source Column Name	STORE_ID
Join DB Name	
Join Owner Name	
Join Table Name	STORES
Join Column from Source Table	STORE_ID
Join Column from Join Table	STORE_ID
Constraint Table - Source Table/Join Table/None/Override	N/A -- No Limit Constraint

Identify the Restricted Database Object Here
Specify Required Joins Here
Apply Required Limits Here

Now, the join to the Regions table is added, and most importantly, with the appropriate constraining value:


Edit Restrictions

User/Group Name: AMERICAS

Source DB Name		Source Owner Name	
Source Table Name	STORES	Source Column Name	REGION_ID

Identify the Restricted Database Object Here

Join DB Name		Join Owner Name	
Join Table Name	REGIONS		
Join Column from Source Table	REGION_ID		
Join Column from Join Table	REGION_ID		

Specify Required Joins Here

Constraint Table - Source Table/Join Table/None/Override	J -- Limit on Join Table
Constraint Column to which limit is applied	REGION
Constraint Operator	=
Constraint Values	'Americas'

Apply Required Limits Here

The only remaining part of the example is letting user BRIO, also a member of the AMERICAS group, see the data in an unrestricted way. Handling this case is left as an exercise for the reader.

Other Important Facts

Custom SQL

Custom SQL is often used to provide special SQL syntax that the software will not otherwise generate on its own. In the absence of the row level security feature, given the proper other permissions on the document, an ODS user can modify this custom SQL on their own to do some totally ad hoc things.

When Row Level Security is in place, Custom SQL is affected in two ways:

1) If the published document contains an open Custom SQL window, it is used "as is" when the user processes a query. No restrictions are applied to the SQL. However, the user cannot modify the SQL. While this can be a handy feature, care should be taken when publishing documents that require custom SQL that they don't compromise the security requirements.
2) If the user clicks the "reset" button on the Custom SQL window, the SQL shown includes the data restrictions, and the original intent of the Custom SQL is lost and the user will not be able to get it back except by requesting the document from the server again.

Similar issues apply to the use of imported SQL.

Limits

The Row Level Security feature affects limits in three ways.

First, if a user is restricted from accessing the content of certain columns, and the user attempts a 'show values' when setting a limit on the restricted column, the restrictions will be applied to the SQL used to get the 'show values' list. That way, the user cannot see and specify a value they would not otherwise be permitted to access.

Second, setting limits can result in some perhaps unexpected behavior when coupled with row level security restrictions. This is best explained by example. In order to read the amount of sales, the user is restricted to a join on the store_id column back to the stores table and in addition, the user can only see information for the store_id when the state is Ohio. This user tries to set a limit on the unrestricted column STATE, and chooses something other than Ohio, thinking this a way to subvert the data restrictions. Unfortunately for that user, no sales amount information will be returned at all in this case. The SQL will specify "where state = ‘user selected value’ AND state = ‘OH’". Obviously, the state cannot be two different values at the same time, so no data will be returned.

Of course, a user may try to set a limit on the city column instead of the state column, thinking the city name might exist in multiple states. As long as the need exists to access the amount of sales column in the sales table with identifying store information, though, the state limit will still be applied, and no data the user should not be able to see will be accessible to that user. It just will not prevent a user from getting a list of stores when sales data is not part of that list. Generally speaking, restricting access to "facts" based on the foreign key in the fact table(s) works best. If it is necessary to restrict the user’s access to a list of stores, these "dimension" restrictions work best when applied to all columns in the dimension table with a limit on the source table.

For example, using the requirements described above to restrict the amount of sales information in Ohio only, with the same restriction on the dimension-only queries, do not apply any limit on access of the amount sales information except that it must be joined back to the stores table on store_id. Then, add a restriction for all columns in the stores table, limiting it to only stores in Ohio. This limits access to both fact and dimension data.

Third, when setting a limit using "Show Values", it has already been noted that any restrictions on the column to be limited are applied to the SQL that generates the "show values" list. For example, using the restrictions described in the previous paragraph, attempting to show the values list for the city column would be constrained to those cities in Ohio. Now, consider the following scenario.

The sales fact table also has a transaction date and product_id column. The transaction date column is tied back to a Periods table, where dates are broken down into quarters, fiscal years, months, etc. In this somewhat contrived example, a restriction is placed on the Periods table, where values there are joined back to the sales transaction table and restricted by product_id values in a certain range. The user sets a limit on fiscal year in the Periods table and invokes "show values" in the Limit dialog to pick the range. Because of the restrictions in place, only one fiscal year is available, and the user picks it. Now, the user builds a query that does not request the fiscal year column itself but does reference the product_id field and processes it. This query returns, for the sake of argument, 100 rows. Now the user decides there is a need to see the fiscal year value and adds it to the Request line. Reprocessing the query only returns 50 rows.

Why? In the first case, product_id values outside of the range allowed when querying the fiscal year column will appear in the results. In the second case, the query will cause the restriction on product_id range to be included. Restrictions are only applied when a user requests to see data. There was no request to see the fiscal year column in the first case, except while setting the limit. There is no restriction on seeing product_id values. This example is contrived because restricting access to a "dimension" based on data in a fact table would be extremely unusual. Nevertheless, it illustrates a behavior that should be kept in mind when implementing restrictions.

Naming

Another way to set the restrictions described above is by a subquery. Instead of directly setting the limit on the STATE column, limit the values in the store_id column in the stores table. The constraint operator would be IN, and the constraint values field might look something like this:

```
(SELECT S.STORE_ID FROM STORES S WHERE S.STATE = 'OH')
```

Now, no matter what limit the user sets in the STORES table, they will always be constrained to the set of store IDs that are allowed based on their group memberships and their own user name. Even if a city outside of the allowed state is chosen, such as a city that exists in more than one state, any stores that other city has will not show up in the results.

Using a subquery can be useful when incorporating existing security systems into the Row Level Security feature of Brio Intelligence. When constructing constraints of this type, it is especially important to know SQL. For example, to specify a subquery, it helps to know that a subquery is always enclosed in parentheses. It is also important to know how Brio software generates SQL and to follow its naming conventions to make sure the syntax generated is appropriate.

Table and Column Names

For the most part, simple security constraints reference directly the actual object names in the database. Case sensitivity in names should be observed when and where required. For subqueries and other SQL constructs used to specify the "constraint values", it is sometimes useful to refer to objects already used by the software's SQL generation process. To do this:

1) For table references in the FROM clause, use From.tablename, where 'tablename' is the display name seen in the Brio document's data model as the display name. If the display name contains a space, use the underscore to represent the space.
2) For column names, use tablename.columnname, following the same rule as above, except that the 'From.' should NOT be used.

Alias Names

By default, when processing user queries, table references in the SQL are always given alias names. Alias names are convenient shorthand for long table references, and they are required when trying to build correlated subqueries. These alias names take the form "ALn", when the 'n' is replaced by an arbitrary number. These numbers are usually based on the topic priority properties of the data model and can easily change based on several factors. For example, a user with the proper permissions can rearrange the topics, thus giving them different priorities. Because these numbers are dynamic, constraint specifications should never rely on them. Instead, by using the naming scheme above, the appropriate alias will be added to the constraints. So, if the requirement is a correlated subquery, the appropriate name will be given to the column in the outer query when referenced by the correlated subquery.

In the example above, using a subquery to restrict store_id values to those in a specific state, it was neither necessary nor desirable to use the Brio naming conventions. There, the set of values was to be derived in a subquery that operated independently of the main query. Consequently, the 'From.' was not used in the FROM clause of the subquery, and the alias names were given in a way to not conflict with the alias names generated automatically by the software.

To use a correlated subquery, then, consider syntax like the following:
…FROM STORES S WHERE S.STORE_ID = Stores.Store_Id
The reference to the right of the equal sign will pick up the alias name from the outer query and thus provide the correct correlation requirements.

A more thorough discussion of the naming conventions and using subqueries with Brio can be found by acquiring a copy of the presentation on subqueries with Brio 5.x version software, given at Insight 99, the Brio Users' conference. With the advent of version 6.x, Brio handles subquery requirements quite differently, but the techniques for naming required by 5.x are still valid when applied to the row level security feature described here.