

SELECT[®] Online

The Journal of the International Oracle Users Group

Select Journal
Home

Regular Columns

Past Issues

General Information

IOUG Home



Select Magazine - October 2000

Volume 7, No. 1

Managing Database Links in a Distributed Environment

By Paul Makkar

When you migrate to a multiple-instance environment and start linking databases, you can introduce chaos and complexity - unless you know a few key techniques for making links simple and secure.

Today's complex information technology issues are causing many companies to move from single-instance databases to multiple-instance, distributed-database environments _ environments that challenge DBAs to learn some tricky database-linking maneuvers.

Often these migrations result from the need for Y2K and Euro compliance in a company's ERP (Enterprise Resource Planning) applications. Trading in-house applications for off-the-shelf products relieves the company's IS staff of the daunting burden of these compliance issues. However, when each of the new products requires its own database instance (generally because of its unique operational requirements or database-creation parameters), the DBAs responsible for setting up these new applications can face unanticipated difficulties. In particular, if you're in this situation, you're likely to find that the database links you create (so that the applications can exchange information) can create security leaks and major programming hassles.

Fortunately, there is a way you can create database links without compromising database security and code comprehensibility. As a DBA at the London School of Economics, I have developed some simple techniques that offer substantial advantages compared with the standard method of creating database links. This article explains these advantages and provides step-by-step instructions to help you use my techniques in creating your own database links.

A Close Look at Linking

The use of database links in multiple-instance scenarios has to do with inter-application-processes _ by which I mean the ability to execute transactions or look-ups across applications. As an example, within my own organization, the personnel system needs to be able to commit pertinent data to the payroll system, so that payroll information about new staff members doesn't need to be entered twice. Under a single instance, this data transfer is a relatively simple task, achieved by assigning grants. In a distributed environment, however, this exchange generally requires the use of database links. (While it's possible to use the alternative approach of extracting and loading batch data from application to application by using SQL*Loader, doing so introduces a time delay.) The use of database links has a significant impact on the ways in which you reference and control access to distributed objects.

Taking the communication between a personnel system and a payroll system as our example, let's look at the complications introduced when you move to a distributed environment - and how you can re-simplify the situation by using my database-linking techniques. We'll start by considering the most common type of object access, which is object access for tables and views. Then, we'll take a brief look at how to use and modify the techniques we've discussed for accessing sequences, packages, procedures and functions.

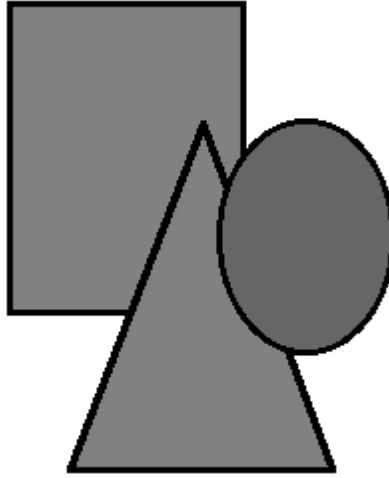


Figure 1. A simple GRANT statement is all it takes to grant privileges on the PAYTAB table in the PAYROLL schema to another user/schema on the same instance.

A Single-Instance Scenario

If you work in an environment where your applications run on one all-encompassing instance, you're no doubt familiar with inter-application processes. Let's assume here that the EMP table in the PERSONNEL schema holds all employee information and the PAYTAB table in the PAYROLL schema contains the payroll details of all employees. In order to avoid double entry of personnel information into both systems, user PAYROLL grants insert, update and delete privileges on table PAYTAB to user PERSONNEL. It is then possible to write triggers to update the payroll system when the personnel system is updated (using SCHEMA.OBJECT_NAME as a naming convention to avoid ambiguity with respect to object ownership).

The commands for granting access rights in this case are quite straightforward, as follows:

```
connect payroll/pwd
grant insert, update, delete on PAYTAB to personnel
connect personnel/pwd
select * from payroll.PAYTAB
```

Now, let's look at how we grant these access rights in a distributed environment.

The Standard Database-Linking Approach

When the PERSONNEL and PAYROLL schemas reside in different instances, the most obvious and straightforward way to allow PERSONNEL to access PAYROLL would be to create a simple database link. In this case, the following commands would create the link and provide access to all available PAYROLL tables from the PERSONNEL schema (where "pays" and "pers" are the system identifiers under which the payroll and personnel systems operate, respectively):

```
connect personnel/pwd@pers
create database link pays.world connect to payroll identified by pwd
using 'pays.world';
select * from tab@pays.world;
<all tables returned>
```

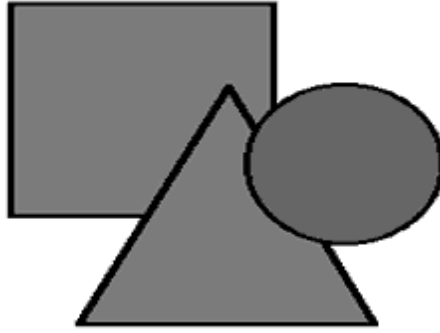


Figure 2: When the PAYROLL schema and the PERSONNEL schema reside on different instances, a simple database link between these schemas gives PERSONNEL access to all PAYROLL tables, with no grants needed. This type of link is an easy way to provide access, but it's also a potential security nightmare and a hassle for programmers (because global naming syntax is now required).

Note that the two references to PAYS.WORLD in the "create database link" statement are not equivalent. The first is the link name, for which the Oracle documentation recommends that you use the global name of the destination instance, as defined in the system view GLOBAL_NAME (see the "What's in a Global Name?" section for more information on global names). The second is the service name, as defined in your tnsnames.ora file.

While this linking strategy is nice and simple, it creates the following undesirable effects:

- It compromises security by providing the PERSONNEL user with full access to all tables in the PAYROLL schema.
- It complicates programming by requiring programmers to use the object-reference syntax OBJECT_NAME@GLOBAL_NAME, instead of the familiar SCHEMA.OBJECT_NAME syntax.
- It restricts the number of database links possible between schemas on the two instances. If you are using global names (switched on by setting the init.ora parameter GLOBAL_NAMES equal to "true"), you are required to use a link name that is equal to the global name of the destination instance, regardless of which schema you connect to. Since database links must be uniquely named, the same user cannot create any further links to the same destination instance (for example, to other schemas). While you might be tempted to bypass this limitation by not using global names, the Oracle documentation strongly recommends that you use them _ and you must use them if there is any possibility that replication will be used in your organization.

Fortunately, you can avoid these problems in accessing remote tables by using my alternate linking strategy, which is based on the concept of the surrogate user.

The Surrogate-User Technique

Suppose that, instead of creating a simple, not-so-secure link between PERSONNEL and PAYROLL, we introduce a third schema in between these two schemas - one that we'll think of as a "surrogate user" for the PAYROLL schema, residing on the PERS instance. (Note that, while Oracle documentation does discuss the "surrogate user" concept in relation to database replication, it does not discuss using surrogate users for the purposes described here.) Now, we can create a specific database link between the two PAYROLL users, the one on the PAYS instance and the surrogate user on the PERS instance, using the following steps (given in general form in the "Creating Your Own Database Links" section):

1. On instance PERS, create a user called PAYROLL, which is then the surrogate user of user PAYROLL on PAYS.
2. On instance PERS, connect as user PAYROLL and create a database link to user PAYROLL on instance PAYS, as follows:

```
create database link 'PAYS.WORLD' connect to payroll identified by payroll
using 'PAYS.WORLD'
```

3. On instance PERS, still connected as PAYROLL, create a view for the table PAYTAB on instance PAYS, as follows:

```
create view PAYTAB as select * from PAYTAB@ PAYS.WORLD
```

4. On instance PERS, still connected as PAYROLL, grant insert, update and delete privileges on PAYTAB to PERSONNEL, as follows:

```
grant insert, update, delete on PAYTAB to PERSONNEL
```

As a result of following these steps, we regain the security and simplicity of access that we had under the single-instance scenario, with links that are easier to manage than those of the standard database-linking approach.

Specifically, we gain the following three advantages over the standard approach:

1. Security is as good as under the single-instance scenario, since the surrogate PAYROLL user can grant a limited set of access rights to PERSONNEL. In our scenario, PERSONNEL has access to all tables in PAYROLL. In the other, PERSONNEL can manipulate data only in table PAYTAB.
2. Object access is as simple as under the single-instance scenario, since objects can be referenced with the familiar SCHEMA.OBJECT_NAME syntax. Once again, the PERSONNEL user can access the PAYTAB table simply by referring to it as "PAYROLL.PAYTAB." The database links are thus transparent to programmers, relieving DBAs of a potentially major burden that looms when references must be converted to include database link syntax.
3. Links are easier to manage without imposed limitations. Database links are created only between schemas and their surrogate schemas _ not between schemas that serve different purposes. And a single schema-to-surrogate-schema link can manage access from multiple other schemas on the remote instance to multiple tables within the target schema. For example, access to any PAYROLL table (on PAYS) from any remote schema on PERS (not just the PERSONNEL schema shown here) is managed through the surrogate PAYROLL schema. So, if there were multiple schemas on PERS needing access to PAYROLL tables, you wouldn't need a separate database link from each schema. Note also that, if there were other instances that required access to PAYROLL tables, each instance could have its own surrogate PAYROLL schema.

In addition to being flexible, the surrogate-user technique is reversible _ for example, you can create a surrogate PERSONNEL user on PAYS to provide access to PERSONNEL tables from the PAYS instance. However, the steps we've detailed do not work for providing access to sequences on remote instances. To access sequences, you'll need to use a modified version of the surrogate-user technique.

The Surrogate-User Technique for Sequences

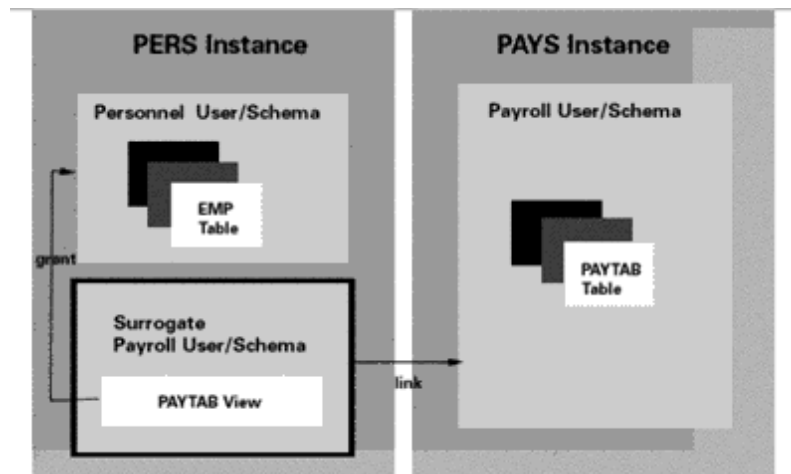


Figure 3: By creating a surrogate PAYROLL user/schema on the PERS instance and linking it to PAYROLL on the PAYS instance, you can grant PERSONNEL controlled access to a view of the PAYTAB table-which PERSONNEL can now reference with the familiar "PAYROLL.PAYTAB" syntax.

The technique we've just demonstrated might seem, at first, as though it ought to work for accessing sequences, as well as tables _ until you remember that you can't create views for sequences. We could try using a synonym instead of a view; however, we'd run into problems trying to make a grant on a synonym. With synonyms (unlike with views), privilege information is not stored in the local data dictionary. Our grant information would therefore need to be passed to the remote dictionary; but remote database definition language (DDL) calls are not allowed. So, our grant attempt would fail.

Since we need to grant access to our sequence locally instead of remotely, the obvious modification to our surrogate-user technique is as follows: To create a surrogate user on the same instance as the sequence (instead of on the instance from which we require remote access to the sequence), so that our surrogate user will be local to the sequence.

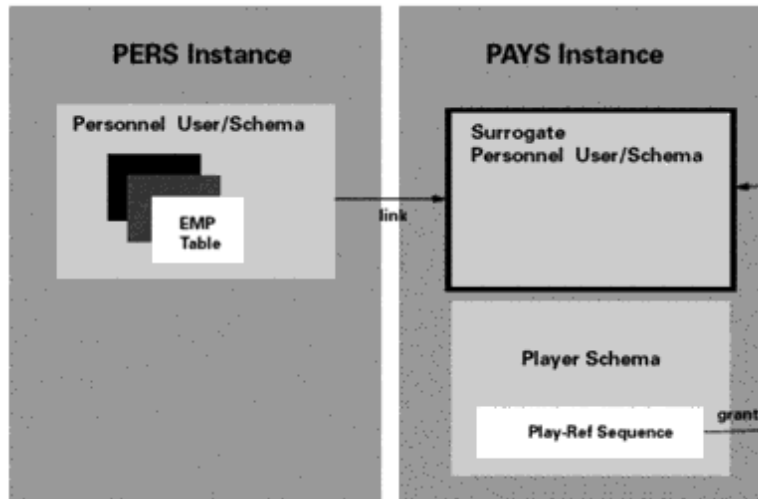


Figure 4: The technique shown in Figure 3 won't work for providing PERSONNEL on the PERS instance with access to the PLAY_REF sequence on the PLAY instance, since views don't work for accessing sequences. Instead, you can create a surrogate PERSONNEL user/schema on the PLAY instance (creating a link from PERSONNEL on PERS to this surrogate) and grant the surrogate controlled access to PLAY_REF.

Putting all this in terms of our example, let's say we want to generate a unique number for each person or business related to our organization. The schema that owns the unique-number data structure is called PLAYER and runs on an instance called PLAY. When we create a new employee, the PERSONNEL user on the PERS instance needs to access the sequence generating this unique number, which we'll call PLAY_REF. To provide this access, we'll create a surrogate PERSONNEL user on the PLAY instance (see Figure 4) and link it to the PERSONNEL user on PERS. PLAYER can then grant the surrogate PERSONNEL user access to the PLAY_REF sequence, and the PERSONNEL user on PERS can access PLAY_REF by way of the database link, using global naming syntax.

Here are the specific steps required to grant PERSONNEL access to the remote sequence PLAY_REF owned by the PLAYER schema on the PLAY instance (steps for the general case of any two schemas are given in the "Creating Your Own Database Links" sidebar):

1. On instance PLAY, create user PERSONNEL, to be the surrogate of the PERSONNEL user on instance PERS.
2. On instance PLAY, connect as PLAYER and grant select privileges on PLAY_REF to the surrogate PERSONNEL user, as follows:

```
grant select on play_ref to personnel; identified by password using 'B.WORLD';
```

3. On instance PERS connected as PERSONNEL, create a database link to PERSONNEL on instance PLAY, as follows:

```
create database link play.world connect to personnel identified by pwd using 'PLAY.WORLD';
```

- From the PERSONNEL schema on the PERS instance, access the PLAY_REF sequence on instance PLAY using global naming syntax, as follows:

```
Select player.play_ref.nextval@play.world from sys.dual; Create database
link B.WORLD connect to userB
```

Although the simplicity of SCHEMA.OBJECT_NAME now needs to be augmented by @GLOBAL_NAME, this change should cause relatively little upheaval for programmers, since sequences are referenced far less than tables and views in program code. And this modified version of the surrogate-user technique still provides security as good as that in the single-instance scenario, as well as easy and flexible link management.

What's in a Global Name?

When you move to a distributed environment, one of the immediate questions you're faced with is whether to enforce the use of Global Names (by setting the init.ora parameter GLOBAL_NAMES equal to "TRUE"). When the Global Names condition is in effect, all database link names must be equal to the GLOBAL_NAME of the remote instance to which the names are being linked. For example, we named our database link "PAYS.WORLD," which is the GLOBAL_NAME of the PAYS instance. The global name of an instance is defined at database creation time to be DB_NAME.DB_DOMAIN, where the DB_NAME parameter equals the system identifier (SID), and the DB_DOMAIN parameter has a default value of "WORLD" if not set. (To alter this global name after the database is created, you simply change the GLOBAL_NAME parameter, rather than altering DB_NAME or DB_DOMAIN.)

In my experience, there are three main benefits of using Global Names in creating database links:

- With Global Names enforced, the number of database link names is kept to a minimum, giving you fewer names that you need to remember. For an environment with n instances, there can be only n different database link names _ a limitation that can be restricting with the standard method of database linking, but not with the surrogate-user method.
- Some forms of database replication, such as updateable snapshots, are possible only if the use of Global Names is enforced. If you don't enforce the use of Global Names and then find that replication becomes a requirement, you'll need to change a lot of code.
- You get better help from Oracle Support Services and Oracle manuals when you're using Global Names, since you're following the procedure that Oracle recommends.

Dealing with Packages, Procedures and Functions

While we don't have room in this article to discuss all of the issues involved in dealing with packages, procedures and functions in a distributed environment, a good general rule is to handle them in the same way that you handle sequences. That is, on the instance containing the package (or procedure or function), create surrogate users for schemas on other instances and grant these surrogates appropriate access to the package (or procedure or function); then, from the appropriate schemas on other instances, create links to these surrogate users.

One point of caution: if your remote package, procedure or function contains a commit statement, it will fail. Commit statements for remote packages, procedures or functions must be executed from the local database, in a process known as a two-phase commit. For more information about the two-phase commit process, consult Oracle8 Server Distributed Database Systems (part no. A54653-01).

Linking it All Together

Moving from a single instance to a distributed environment can be a daunting prospect, especially when considering inter-application processes. The most obvious method of implementing database links is not the easiest _ not when you factor in the security hassles and programming disruption it's likely to cause. Fortunately, the surrogate-user technique for database linking lets you avoid these undesirable effects. By following the steps detailed in the "Creating Your Own Database Links" section, you create database links that are secure, manageable, and transparent to your programming staff. As a DBA, what more can you ask?

Creating Your Own Database Links

While it should be relatively easy to modify our examples of the surrogate-user technique for creating your own database links, it's always been helpful to have general instructions to work from as well. Just substitute the names of your own instances and database objects and follow the appropriate set of steps, either for accessing a table on a remote instance, or for accessing a sequence. To access a remote package, procedure or function, you can follow essentially the same process as for accessing a remote sequence (making sure to comply with the two-phase commit process, as noted elsewhere in this article, if commit statements are involved).

Accessing a Table

Assume on instance A you have userA, and on instance B you have userB. Assume userB has a table called tableB and userA requires select access. Follow these steps to grant access:

```
Create database link B.WORLD connect to userB
identified by password using 'B.WORLD';
```

1. On instance A, create a user called userB.

```
Create view tableB as select * from tableB@B_WORLD;
```

2. On instance A, connect as userB and create a database link to userB on instance B, as follows:

```
Grant select on tableB to userA;
```

3. On instance A, still connected as userB, create a view for tableB as follows:

4. On instance A, still connected as userB, grant select privileges on tableB to userA, as follows:

Now, UserA can refer to tableB in the familiar form userB.tableB.

Accessing a Sequence

Assume on instance A you have userA, and on instance B you have userB. UserA needs access to the sequence, sequenceB, owned by userB on instanceB. Follow these steps to grant access:

1. On instance B, create a user called userA.
2. On instance B, connect as userB and grant select privileges on sequenceB to userA, as follows:

```
Grant select on sequenceB to userA;
```

3. On instance A as userA, create a database link to userA on instanceB.

```
Create database link B.WORLD connect to userA identified by password using
'B.WORLD';
```

Now, UserA can access sequenceB with the following command:

```
Select userB.sequenceB.nextval@B.world from sys.dual;
```

In addition to being flexible, the surrogate-user technique is reversible _ for example, you can create a surrogate PERSONNEL user on PAYS to provide access to PERSONNEL tables from the PAYS instance.

About the Author

Paul Makkar (pmakkar@yahoo.com) is a Senior DBA at the London School of Economics and Political Science. He has more than five years of experience as an Oracle DBA on UNIX and NT.

[Download Acrobat Reader](#)

Copyright 2003 by the International Oracle Users Group