Multi-Master Replication Conflict Avoidance and Resolution

By John Garmany and Robert Freeman (not pictured)

The following is the final chapter from Oracle Replication, *by John Garmany and Robert Freeman, published by Rampant TechPress.*

Introduction

Congratulations if you have gotten this far and your tables are happily replicating data back and forth. But you are not quite finished yet. If you are using advanced replication, you need to take steps to avoid data conflicts and implement conflict resolution.

No matter how well you planned your replication environment, there are many ways that conflicts with table keys can show up. Any time data is loaded or changed outside the replication environment, such as offline instantiation or data loading, you run the risk of creating key conflicts.

Before placing a replication environment into production, you must implement conflict resolution to protect data integrity. In this chapter, we are going to discuss what conflicts are, how to avoid them, and how to implement an automated way to resolve them. Finally, we will discuss how to determine when your data is out of synchronization between two tables, as well as the methods to resynchronize them.

What are Conflicts?

The function of a primary key on a single table is easy to understand. The primary key can be used to uniquely identify every row in the table. If you try to insert a row into a table with a primary key and that key already exists in the table, Oracle will reject the new row. In the PUBS database, authors are listed in the *author* table that contains an author key called *auth_key.* This key uniquely identifies each author. This is how you differentiate authors with the same name, like Robert Smith.

Oracle's replication uses this primary key to identify unique rows at all sites containing that table. What about tables without primary keys? All tables in the replication environment contain a key that replication uses to uniquely identify rows across the separate master sites.

Remember, when you created the master group at the master definition site, all tables that did not have a primary key had to have a column or columns identified that uniquely identified rows. Using the SET COLUMNS procedure, you created a key for Oracle to use in replication. Conflicts happen when data is propagated that violates the key.

To greatly increase the efficiency of replication, Oracle propagates transactions on a schedule. All sites that allow updates, such as updatable Mviews and master sites, will insure that the keys used for replication are not violated. When a row is inserted into a table, the local key is checked, and if the data violates the key, the insert is refused. If the data does not violate the key, it is inserted into the table and the change passes to the deferred transaction queue.

During a push, the data is propagated to other master site deferred transaction queues where the receiver applies them. If the receiver detects that the data violates the key on the local site, it will refuse the transaction and place it in the deferred error queue. This is an example of a conflict. The data is applied at one master site but cannot be applied at another site. At this point, the replicated tables no longer contain the same data.

Key conflicts are just one possible type of replication conflict. When a receiver applies a transaction from the deferred transaction queue, it checks the "before" image to insure that the table data is still the same. What happens when one row of data is updated at two remote sites at the same time? The first update propagates normally and is applied, the second update follows and, because it expects the before image to be the original data, it fails to be applied. The before image is changed by the first update. This is called an update conflict.

A delete conflict is similar; one site updates a row while another site deletes the row. If the update propagates first, the delete fails because the before image does not match. If the delete propagates first, the row no longer exists when the update is applied. As you can see, getting your replication environment to function is just the beginning. To keep it operating, you need to plan for conflict avoidance and apply a method of conflict resolution.

Conflict Avoidance

Conflict avoidance is the first step to insure transactions don't end up in the deferred error queues. In planning the replication environment, you must insure that each master site has the ability to generate unique keys. In the last chapter, we discussed two common methods to generate unique keys: assigning blocks of numbers via sequences and attaching a site prefix to a sequence number.

Assigning Blocks of Numbers Via Sequences

If a sequence is used to establish a primary key, you can assign each site a block of numbers to be used for generating a primary key. This is accomplished by setting the sequence at each site to start at a different number.

continued on page 10

```
connect pubs/pubs@navdb.world
create sequence pubs.seq_pk_author
increment by 1
start with 1
maxvalue 99999999
minvalue 1 nocycle
cache 10;


connect pubs/pubs@mydb.world
create sequence pubs.seq_pk_author
increment by 1
start with 100000000
maxvalue 199999999
minvalue 1 nocycle
cache 10;

connect pubs/pubs@newsite.world
create sequence pubs.seq_pk_author
increment by 1
start with 200000000
maxvalue 1.0E28
minvalue 1 nocycle
cache 10;
```

The code above establishes a sequence at three master sites that start at 100,000,000 intervals. With a max value for a sequence equal to 1.0E28, you have a very large pool to divide between the master sites.

There are two key points to remember. First, set up an automated check of the sequences to warn you if any site begins to approach maxvalue. Second, don't forget the future possibility of adding additional master sites. If you divide the available number equally between all master sites, adding one master site cuts the available number for one site in half to make room for the new site. This issue alone makes this method of generating unique keys less scalable.

Remember to check *dbms_reputil.from_remote=FALSE* inside the trigger to insure that the trigger does not fire on a transaction from a remote site.

Assigning a Prefix Based on Site

A more flexible (and scalable) approach is to have all sites start their sequence at 1 and add a site prefix to the sequence number to generate the key. At NAVDB.WORLD, the first three rows created would have the keys navdb1, navdb2, and navdb3. At the master site MYDB.WORLD, the first three would be mydb1, mydb2, and mydb3. As these rows are propagated none of the keys will collide.

In real life, keys are not always created on sequences. In cases such as the *author_key* column in the PUBS schema, you may need keys generated from a central location. All master sites obtain their keys for that table from the central repository. You must either require that only that site creates the records or create a procedure on the remote site that will fetch the next key from the depository. For efficiency, you could create a trigger that pre-fetches a number of keys and places them in a table and provides them as needed on the local site, fetching additional keys as needed.

It is always a much better solution to use the natural key of the table, if one exists, rather than a derived key. A natural key can avoid a great deal of trouble.

As you can see, conflict avoidance must be part of your replication planning. Once the replication environment has been created and is operating, it is much harder to avoid conflicts. Also, insure that your conflict avoidance method is flexible enough to handle adding and removing master sites.

However, no matter how diligent you are at planning conflict avoidance, you will eventually have a conflict, and unless you implement conflict resolution, your replication environment will grind to a halt until you manually intervene.

Conflict Resolution

Establishing conflict resolution is a process of defining rules for Oracle to apply when a conflict is detected. If the conflict is resolved, the transaction does not end up in the deferred error column group that the resolution method applies to.

You can have multiple column groups on a table and define a priority for each group. A column group can consist of all the columns in the table or any subset of those columns. Once you have created a column group, you can assign one or more conflict resolution methods to it. Oracle has defined a number of standard conflict resolution methods that will usually meet your needs. The following table provides a list of the different conflict resolution methods Oracle makes available:

METHOD	DESCRIPTION
Latest Timestamp Value	With the latest timestamp value method, you define a column that contains a date data type to use in comparing multiple transactions. When a transaction fails because the before image has changed, the column timestamps of the transactions are compared, and if the second transaction is later than the one changing the before image, the second transaction is applied and overlays the current data (which contains the first transaction).
Earliest Timestamp Value	This is the opposite of the above method. The method is available but rarely used.
Minimum Value, Maximum Value	When a column group has a conflict between two transactions, the minimum value method evaluates the current and new values of a defined column and selects the lower (or higher for maximum) values to determine if the new columns are applied.
Group Priority Value	In this case, column groups are assigned a priority and conflicts are adjudicated to the highest priority group.
Site Priority Value	In this instance, sites are assigned a priority. When two transactions conflict, the transaction from the master site with the highest priority will be applied. This is actually a special case of the Group Priority Value method above.

Conflict Resolution never performs a rollback of a transaction. Since all master sites will contain the same resolution methods, each site should apply the same transactions. If transaction A overwrites transaction B, the site that originally creates transaction B will eventually overwrite it with transaction A. Since all transactions are committed at the originating site, rolling back the transaction is not possible.

What if Oracle is not able to determine which transaction should be applied after using a conflict resolution method? You can define more than one conflict resolution method. If Oracle is still unable to resolve a conflict after using all defined conflict resolution methods, the transaction ends up in the deferred error queue and someone must manually resolve the conflict.

Examples of Defining Conflict Resolutions

Let's look at creating some of the more common resolution methods in our PUBS schema. First, we will look at an example of the latest timestamp conflict resolution. Then we will look at using the site priority conflict resolution method.

Defining Latest Timestamp

Conflict resolution methods are defined using the *dbms_repcat.add_update_resolution* procedure. In the following example, we will configure Oracle to use the latest timestamp method of resolution to settle conflicts with the *pubs.sales* table's ORDER_DATE column. In this example we will:

1. Define a column group for use by the resolution method.

2. Define the resolution method.

3. Add additional support as required.

Let's look at these steps in more detail then.

Define the Column Group

The first step is to define a column group on the master definition site that uses this resolution method.

```
execute dbms_repcat.make_column_group(
 'PUBS','SALES','SALES_COLGP','*');
```

In this example, we are defining a column group called SALES_COLGP on the *sales* table in the PUBS schema. This column group will include every column in the table, since the '*' is being used.

It is not necessary to include all columns in the column group. Since the ORDER_DATE column can act as a primary key, we could have only defined ORDER_DATE in our column group, which would have caused Oracle to resolve only conflicts on that column. By defining all columns in the column group, any conflict within the column group will use the conflict resolution method.

Define the Conflict Resolution Method

Next, we define the conflict resolution method for that column group.

```
execute dbms_repcat.add_update_resolution(
 sname => 'PUBS',
 oname => 'SALES',
 column_group => 'SALES_COLGP',
 sequence => 1,
 method => 'LATEST TIMESTAMP',
 parameter_column_name => 'ORDER_DATE');
```

Now, when a conflict occurs on the column group, the first (and only, for the moment) method of resolution will be the latest date method (defined via the *method* parameter). The date within the ORDER_DATE column will be used to resolve the conflict.

If we wanted to define a second method, we could use the same command, but replace the *method* parameters and change the *sequence* to equal two. Oracle will apply the two conflict resolution methods in the sequence order.

You might have noticed that the above example has limited usefulness, since the order date is likely to remain the same for all transactions. It would be more useful to compare the transaction execution time, rather than the ORDER_DATE.

So let's alter our replicated object using the procedure *dbms_repace.alter_master_repobject,* which we call from the master definition site. With this procedure, we will add an additional column to the *sales* table that we'll call SALES_TS of type DATE.

```
BEGIN
DBMS_REPCAT.SUSPEND_MASTER_ACTIVITY('REP_GROUP2');
 DBMS_REPCAT.ALTER_MASTER_REPOBJECT(
  sname => '"PUBS"',
  oname => '"SALES"',
  type => 'TABLE',
  ddl_text => -
'alter table PUBS.SALES add (SALES_TS DATE)');
DBMS_REPCAT.RESUME_MASTER_ACTIVITY('REP_GROUP2');
END;
/
```

Now that we have added our column, we want it populated each time some action takes place on the table. To perform this action we write the following database trigger:

```
create trigger SALES_TS_TG
before insert or update
on PUBS.SALES
for each row
begin
 if dbms_reputil.from_remote = FALSE
 then :NEW.SALES_TS := SYSDATE;
 end if;
end;
/
```

Now we need to replicate the trigger to the other master sites using the *create_master_repobject* procedure. This will ensure the trigger is replicated to all of the master replication sites:

```
BEGIN
DBMS_REPCAT.CREATE_MASTER_REPOBJECT (
 gname => 'REP_GROUP2',
 type => 'TRIGGER',
 oname => 'SALES',
 sname => 'PUBS',
 ddl_text => ' create trigger SALES_TS_TG
  before insert or update
  on PUBS.SALES
  for each row
  begin
  if dbms_reputil.from_remote = FALSE
   then :NEW.SALES_TS := SYSDATE;
  end if;
 end;');
END;
/
```

Then, we need to redefine the column group and add the conflict resolution method as shown here:

```
execute dbms_repcat.add_update_resolution(
 sname => 'PUBS',
 oname => 'SALES',
 column_group => 'SALES_COLGP',
 sequence => 1,
 method => 'LATEST TIMESTAMP',
 parameter_column_name => 'SALES_TS');
```

Now our replication will function more like a stand-alone database. If two

continued on page 12

users modify the same data, the data will be applied in the order of commits. In our replication, the two transactions will be applied in the order of the timestamp. This is the most common conflict resolution method.

There is one caveat to consider with this method. All replication database servers must be operating on one time standard. If you have two servers in your replication scheme, one in Virginia and the other in California, the Virginia server time will be four hours ahead of the California server.

Normally databases used in a replication environment use a standard time, either GMT or local time at the company headquarters. It doesn't matter as long as they are the same. You can also create the trigger to compensate for the time differences, but you will not be able to replicate the trigger. Instead, you will need to create the trigger at each site, adjusting for the time. Since I live in Colorado I use Mountain Time, to convert to GMT, I would change the trigger to convert the time.

```
:NEW.LAST_UPDATE := NEW_TIME(SYSDATE,'MDT','GMT');
```

If Oracle was unable to resolve the conflict using the latest timestamp, we could provide an additional resolution method to use. It's called the Site Priority method.

Defining Site Priority

The site priority conflict resolution method uses a column value to determine replication conflict priority. Before we start, we need to stop replication activities for the group (this is known as a quiesce of the replication group).

We define another column in the *pubs.sales* table that will contain an identifier for the site that created the transaction. Next, we add a trigger to insert a site identifier (usually the database global name). We then need to define the site priorities, using the stored procedure *dbms_repcat.define_site_priority.* Finally, create the column group and define the conflict resolution method.

First, we need to stop replication as seen here:

```
BEGIN
DBMS_REPCAT.SUSPEND_MASTER_ACTIVITY('REP_GROUP2');
END;
/
```

For this example, we added the SALES.SALES_SP column (varchar2(30)) to use for the site data (an example of adding a column to a replicated table was shown earlier in this chapter). Next we need to add the trigger.

```
create trigger SALES_SP_TG
before insert or update
on PUBS.SALES
for each row
declare
 site_name varchar2(30) := -
   dbms_reputil.global_name;
begin
 if dbms_reputil.from_remote = FALSE
 then :NEW.SALES_SP := site_name;
 end if;
end;
/
```

Remember to add the trigger to the replication group so that it is replicated to all master sites (an example of adding triggers to the replication group was provided earlier).

Now, every update or insert transaction will contain the site's global name in the SALES_SP column. We need to define a priority to each site in our replication scheme. The *define_site_priority* is the procedure we will use:

```
execute dbms_repcat.define_site_priority(
 'REP_GROUP2','SITE_PRI');

execute dbms_repcat.add_site_priority_site('
 REP_GROUP2','SITE_PRI','MYDB.WORLD',10);

execute dbms_repcat.add_site_priority_site('
 REP_GROUP2','SITE_PRI','NAVDB.WORLD',100);

execute dbms_repcat.add_site_priority_site('
 REP_GROUP2','SITE_PRI','NEWSITE.WORLD',5);
```

Here, we define the priority for each site in our replication scheme. In this example, the master definition site NAVDB.WORLD has the highest priority (the higher the number, the higher the priority). Notice that we did not have to create them in order, nor do you have to increment the priority number by one (which comes in handy if you add a master site later down the line). At this point, you are actually assigning priorities to a field in a column, not actual master sites thus, you can predefine a priority for a master site that is not yet created.

Now that we have defined our master sites priorities, we need to define the conflict resolution method.

```
execute dbms_repcat.add_update_resolution(
 sname => 'PUBS',
 oname => 'SALES',
 column_group => 'SALES_COLGP',
 sequence => 2,
 method => 'SITE PRIORITY',
 parameter_column_name => 'SALES_SP',
 priority_group => 'SITE_PRI');
```

Before resuming replication, we need to regenerate replication support for the *sales* table.

```
execute dbms_repcat.generate_replication_support(
 'PUBS','SALES','TABLE');
```

And finally, resume replication activity to propagate the changes to all other master sites.

```
execute dbms_repcat.resume_master_activity(
 'REP_GROUP2');
```

There are now two methods of conflict resolution defined on the *pubs.sales* table. Oracle will first try to resolve any conflict using the latest timestamp method. If that fails, it will try using the site priority method. If the site priority method fails, the transaction will be placed in the deferred error queue.

Monitoring Conflict Resolution

To monitor conflict resolution, Oracle provides a number of views. First, we need to register the object that Oracle is going to maintain statistics on.

```
execute dbms_repcat.register_statistics(
 'PUBS','SALES');
```

You can then get information on resolved conflicts from the view *dba_represolution_statistics.*

SELECT * FROM dba_represolution_statistics;

dba_represolution_statistics contains the following fields.

```
Name     Null? Type
 ---------------------------- ----------
 SNAME    NOT NULL VARCHAR2(30)
 ONAME    NOT NULL VARCHAR2(30)
 CONFLICT_TYPE    VARCHAR2(10)
 REFERENCE_NAME  NOT NULL VARCHAR2(30)
 METHOD_NAME  NOT NULL VARCHAR2(80)
 FUNCTION_NAME    VARCHAR2(92)
 PRIORITY_GROUP    VARCHAR2(30)
 RESOLVED_DATE  NOT NULL DATE
 PRIMARY_KEY_VALUE NOT NULL VARCHAR2(2000)
```

As conflicts are resolved, rows are added to the view. To remove the statistics, you need to purge them.

```
execute dbms_repcat.purge_statistics(
 'PUBS','SALES');
```

To stop gathering resolution statics on the *sales* table:

```
execute dbms_repcat.cancel_statistics(
 'PUBS','SALES');
```

Of course, data on conflicts that are not resolved is in the deferred error queue and can be found in the deferror view.

select * from deferror;

Using OEM to Define Conflict Resolution Methods

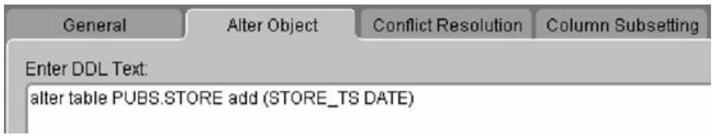
Oracle Enterprise Manager can not only create column groups and define conflict resolution, but it presents the information in a fairly comprehensive way. While we wholeheartedly recommend OEM for monitoring multi-master replication, we do not recommend it for creating multi-master replication and conflict resolution.

Because of the complex nature and multiple layers of conflict resolution, we recommend that you create scripts that document methods, column groups, etc. OEM will create conflict resolution without allowing you to document what and how it was created.

That being said, let's look at how OEM creates the latest timestamp conflict resolution method on the *pubs.sales* table.

Log on to OEM as REPADMIN and navigate to the replication group created in the last chapter, REP_GROUP2. Submit a stop request to quiesce replication activity. Expand the REP_GOUP2 folder and select the *pubs.store* table.

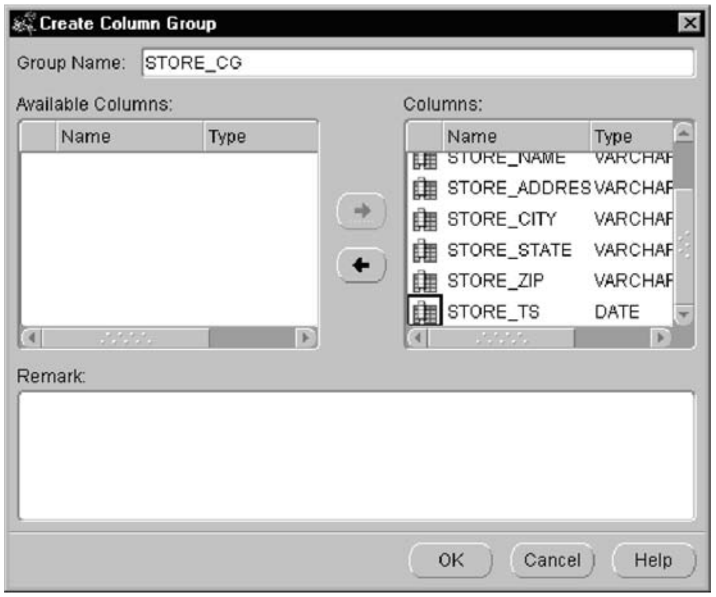
First, we need to add a column to the *store* table called STORE_TS. Select the Alter Object tab and enter the DDL to add the column and select apply. Now, create the trigger and add it to the replication master group.



Enter DDL Text:
alter table PUBS.STORE add (STORE_TS DATE)

Figure 1. OEM Adding a Column to the store table

Select Column Subsetting to define the column group. The only column group defined is the Shadow Group, which is an internal group that contains all columns in the table. Once you define a column group, the shadow group will disappear. Select the Create button to open the Create Column Group window. Name the column group STORE_CG. Select all the columns and move them to the right hand pane with the right arrow button.



Group Name: STORE_CG

Columns:

Name	Type
STORE_NAME	VARCHAR
STORE_ADDRES	VARCHAR
STORE_CITY	VARCHAR
STORE_STATE	VARCHAR
STORE_ZIP	VARCHAR
STORE_TS	DATE

Figure 2. OEM Create Column Group

Select OK to create the STORE_CG column group. Once the group is created, highlight it and select the Add button in the Column Group Resolution Methods text box to bring up the Edit Resolutions Methods Window. Select the Latest Timestamp and STORE_TS from the combo boxes and select OK to create the method.

continued on page 14

Multi-Master Replication Conflict Avoidance and Resolution continued from page 13

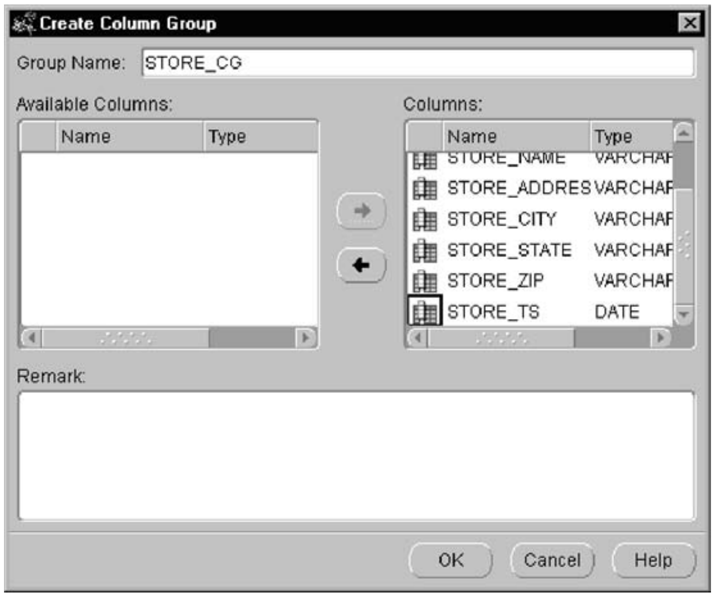


Figure 3. OEM Update Resolution Methods

The final task is to again generate replication support for the *store* table and then resume replication activity.

Setting up conflict resolution using OEM is relatively painless, however, there is no documentation trail, such as a script, to allow you to recreate the scheme if needed.

As you can see, advanced replication can and does become very complicated, with multiple conflict resolution methods defined on column groups, on tables, and in master groups. Documentation becomes very important when multiple DBAs support the replication environment. As the replication scheme grows in complexity, scripts to recreate and document it become an important asset.

So now we have our replication scheme functional with conflict resolution in place. The last area to cover is what happens when tables no longer contain the same data. This is called Data Mis-convergence.

Data Mis-convergence

Data Mis-convergence happens when the data in the tables does not match. This should never happen, but it will, so there needs to be a plan to resynchronize the data. If the same number of rows exist in the replicated tables, then the data mis-convergence was caused by updates performed while replication was turned off using *dbms_reputil.replication_off.*

There are a number of ways to resynchronize the data in the tables. When the data is out of sync, any update will generate a conflict, because the before image is not what is expected. A last timestamp conflict resolution method will automatically resolve the conflict and re-sync the data with the latest update.

If knowledgeable about the replication scheme, a DBA may update all the timestamp fields on a table to force all the rows to propagate the data from the correct table, re-synchronizing the remote tables.

Using dbms_rectifier_diff

One way to determine if your data is out of synch is to use the *dbms_rectifier_diff* package that comes with Oracle9*i*. This package compares the data at a master site (not required to be the master definition site) with the data at a remote master site. It loads any discrepancies it finds into a user created table and can then use that data to synchronize the two tables.

In this example we will compare the *author* table at NAVDB.WORLD to the author table at MYDB.WORLD. First, we will create a couple of tables to hold the data found by the *differences* procedure.

```
connect repadmin/repadmin@navdb.world

create table PUBS.MISSING_ROWS_AUTHOR (
 AUTHOR_KEY   VARCHAR2(11),
 AUTHOR_LAST_NAME VARCHAR2(40),
 AUTHOR_FIRST_NAME VARCHAR2(20),
 AUTHOR_PHONE  VARCHAR2(12),
 AUTHOR_STREET  VARCHAR2(40),
 AUTHOR_CITY   VARCHAR2(20),
 AUTHOR_STATE  VARCHAR2(2),
 AUTHOR_ZIP   VARCHAR2(5),
 AUTHOR_CONTRACT_NBR NUMBER(5));

 create table PUBS.MISSING_LOCATION_AUTHOR (
 present     VARCHAR2(128),
 absent      VARCHAR2(128),
 r_id                    ROWID);
```

The first table matches the definition of the *author* table so that it can hold a complete row of data. The second table must be defined as shown. Next, we need to suspend replication activity.

```
dbms_repcat.suspend_master_activity(
 'REP_GROUP2');
```

Now execute the stored procedure *dbms_rectifier_diff.differences* to determine if rows are not in sync between the two tables.

```
execute dbms_rectifier_diff.differenced(
 sname1                           => 'PUBS',
 oname1                           => 'AUTHOR',
 reference_site                   => 'NAVDB.WORLD',
 sname2                           => 'PUBS',
 oname2                           => 'AUTHOR',
 comparison_site                  => 'MYDB.WORLD',
 where_clause                     => NULL,
 column_list            => NULL,
 missing_rows_sname     => 'PUBS',
 missing_rows_oname1    => 'MISSING_ROWS_AUTHOR',
 missing_rows_oname2    => 'MISSING_LOCATION_AUTHOR',
 missing_rows_site      => 'NAVDB.WORLD',
 max_missing            => 500,
 comit_rows             => 100);
```

Using a NULL in the *column_list* parameter will cause all columns to be used. The rows that are out of sync between the two tables are contained in the *missing_rows_author* table. You can use this information to manually fix the data or you can have Oracle re-sync the two tables.

To have Oracle resynchronize the two tables you next run the *dbms_rectifier_diff.rectify* procedure. Oracle will resynchronize the remote table with the local table. The local table's data will not be changed. Oracle will insert the missing rows and delete the rows that are not contained in the local table.

Because of *rectify's* resynchronization methodology, you may not want to use it. It is important that you run these procedures from the location that has the correct data. You must run *differences* before running *rectify.*

```
dbms_rectifier_diff.rectify(
  sname1               => 'PUBS',
  oname1               => 'AUTHOR',
  reference_site       => 'NAVDB.WORLD',
  sname2               => 'PUBS',
  oname2               => 'AUTHOR',
  comparison_site      => 'MYDB.WORLD',
  column_list          => NULL,
  missing_rows_sname => 'PUBS',
  missing_rows_oname1 => 'MISSING_ROWS_AUTHOR',
  missing_rows_oname2 => 'MISSING_LOCATION_AUTHOR' ,
  missing_rows_site    => 'NAVDB.WORLD',
  comit_rows           => 100);
```

The last point about the *dbms_rectifier_diff* package is that both procedures may take an extremely long time to execute. You might want to just re-instantiate the remote table using transportable tablespaces or an export of the local table.

Conclusion

This chapter contains a lot of complex procedures to insure the integrity of your data in the replication environment. The key discussions were on Conflict Avoidance, Conflict Resolution, and Data Mis-convergence.

- **Conflict Avoidance –** Plan the replication environment from the start to insure conflicts are rare. Do not rely on only one site updating and inserting data. Insure your plan is flexible enough to allow for adding or remove master sites.
- **Conflict Resolution –** Here we introduced a few of the most common methods of automatic conflict resolution and the steps required to implement them. Remember that conflict resolution allows the database to determine which transactions are applied when conflicts arise. If Oracle is unable to make that determination, the transaction is placed in the local deferred error queue and replication begins to fail.
- **Data Mis-convergence –** Re-synchronizing tables in a replication environment can be a daunting task. Oracle9*i* provides the *dbms_rectifier_diff* package to help you determine if your tables are no longer synchronized. However, for large data sets, it may be advantageous to reconstruct the replication table due to the time it will take the *rectify* procedure to run.

That's it. We have implemented both basic and advanced replication, monitoring scripts, and conflict resolution. You should now be able to plan and create your replication environment.

This article has not been an exhaustive examination of Oracle replication, nor was it meant to be. Our goal was to provide the basics, identify some of the pitfalls, and give working examples that allow you to implement replication in your environment.

Remember to start in a test environment. Plan out the types of replication needed and don't implement multi-master replication unless your requirements dictate. Also, document each step. Remember, someone else may have to support or rebuild the replication scheme.

About the Authors

John Garmany is a Senior Oracle DBA with Burleson Consulting. A graduate of West Point, an Airborne Ranger and a retired Lt. Colonel with 20+ years of IT experience, he also holds a master's degree in information systems, a graduate certificate in software engineering, and a BS degree in electrical engineering from West Point. A respected Oracle expert and author, Garmany writes for Oracle Internals, SearchOracle and DBAZine. He is the author of four bestselling Oracle books published by Rampant TechPress, Oracle Press, and CRC Press and hosts a popular Oracle Application Server Newsletter.

Robert Freeman is an Oracle expert and author of five popular Oracle books, including *Oracle9i RMAN Backup & Recovery* and the bestselling *Oracle Replication.* A master of martial arts and a black belt in karate, Freeman is certified in Oracle7 and Oracle8 with more than a decade of experience. An exciting and dynamic speaker, Robert Freeman has taught extensively and is a popular speaker at Oracle conferences.