An Oracle White Paper
Updated November 2009

Best Practices for Conflict Detection and Resolution in Active-Active Database Configurations Using Oracle GoldenGate

ORACLE

Executive Overview 1
Introduction 1
Key Requirements for Active-Active Configurations 3
Real-Time, Low-Impact Data Movement 3
Conflict Detection and Resolution 3
Heterogeneous Environment Support 4
Conflict Detection 4
Understanding Conflicts and Complex Resolutions 4
Simple Conflict Resolution Methodologies 5
Hybrid Methodologies for Conflict Detection 8
Complex Conflict Resolution 9
Quantitative Resolution 9
Transaction Consistency 11
Conflict Notification and Tracking 13
Exceptions 13
Resolution Notification 15
Minimizing Conflicts 16
Application Segregation 17
Primary Key Generation 17
Conflict Avoidance 17
Oracle GoldenGate for Active-Active Databases 18
Oracle GoldenGate Data Definition Language Replication 20
Conclusion 21

Executive Overview

One of the most effective ways to enable increased availability and performance for database infrastructures is to establish an active-active environment, which distributes database transactions across multiple databases. Beyond disaster recovery and tolerance, active-active configurations facilitate continuous operations, offer additional raw computing capacity, and provide the flexibility to optimize workload management. However, implementing an active-active solution is not trivial. The key to success lies in real-time data movement, conflict detection and resolution, and support for heterogeneous environments. Of the three, conflict detection and resolution introduces the most complexity. This white paper provides best practices for conflict detection and resolution and highlights how Oracle GoldenGate addresses these challenges.

Introduction

A key objective for any IT organization is to create software applications and a database infrastructure that can scale to meet growing and changing business needs. With business processes increasingly migrating to digital transactions, there is a growing organizational reliance and dependence on the IT group's ability to handle larger volumes of data and users, with less system downtime. Active-active configurations provide significant performance and scalability benefits; deliver exceptional high-availability; and enable continuous operations for not only unplanned interruptions but also planned outages such as migrations, upgrades, and systems maintenance.

In most cases, active-active configurations are considered to be part of a continuous availability—not a disaster recovery—plan. At the high end of traditional disaster recovery plans, there are solutions that offer an active-passive configuration where the active system assumes all the workload, but when it fails, the passive system becomes active and assumes the full workload. Under normal operating conditions, the secondary

(passive) system doesn't contribute to handling the data processing load; it is twice the investment to provide the same amount of processing power as a single system. By comparison, an active-active configuration not only facilitates very high levels of recovery point and recovery time objectives, but it also returns value on the investment by adding capacity, flexibility, and higher performance to the operational data infrastructure.

Implementing an effective active-active configuration requires a thorough consideration of technologies available for enabling the data movement and sharing between the database instances. Before moving forward, an organization must understand the different use cases for active-active configurations and the challenges and benefits of each configuration. They must also understand the different methods for detecting data conflicts that occur and how to effectively resolve those conflicts.

Key Requirements for Active-Active Configurations

In an active-active database configuration—also referred to as a master-master, dual-master, multimaster, or peer-to-peer configuration—multiple database systems concurrently process data transactions. Any changes that persist on one system are reflected in the other systems. The key benefit of this type of configuration is the ability to balance the transaction workload across multiple systems. Each additional system in the active-active configuration increases the overall capacity, resulting in improved response times and enhanced system performance.

Active-active configurations enable workload partitioning based on multiple attributes. For example, different applications can be routed to different systems in the configuration, or users in a specific region can be serviced by a local database server. Thus, active-active configurations not only offer additional capacity, but they also offer the flexibility to optimize workload management.

Despite all the availability and performance benefits that can be reaped by the business, it is critical to point out that implementing an active-active solution is not trivial. The key ingredients include

- Real-time bidirectional data movement
- Conflict detection and resolution
- Heterogeneous environment support

Real-Time, Low-Impact Data Movement

To load balance users across multiple databases, all users must have access to the same data. In practice, this requires more than just moving data from one system to another. The ideal solution should impose minimal latency and very low overhead—without introducing interprocess dependencies. Although a synchronous approach using a two-phased commit would provide zero latency, it would also lead to high overhead and dependencies across multiple systems. In active-active configurations, the data movement has to be asynchronous yet provide "synchronouslike" behavior.

Conflict Detection and Resolution

In an active-active configuration, data collisions are inevitable. When two resources simultaneously update the same record on two separate systems, the ensuing conflict must be detected and resolved. To support a wide variety of business rules, an effective active-active solution must facilitate different conflict detection and resolution mechanisms.

Heterogeneous Environment Support

Systems in an active-active configuration might have different hardware setups, operating systems, service packs, database versions, and patch levels. To ensure continuous operations during upgrades and maintenance operations, and to provide flexibility for optimal resource allocation, the active-active solution must provide heterogeneous infrastructure support.

Conflict Detection

Although the requirements in the previous section are needed to enable active-active configurations, the majority of the complexity is introduced by conflict detection and resolution. Understanding the processes behind this prerequisite and how they are affected by organizational business rules is critical to establishing a successful active-active deployment.

Understanding Conflicts and Complex Resolutions

Different types of conflicts require different resolutions. In certain instances, the conflicts are simple and the rules to resolve them are equally straightforward. For example, if a BlackBerry calendar and a Microsoft Outlook calendar were at odds, a simple resolution rule could be created to overwrite the conflicting BlackBerry calendar with the content contained in Outlook. In most enterprise applications, however, business rules are a lot more complex.

Take the case of two users, John and Adam, trying to obtain the same seat on the same flight. Each is using a different system to book the flight, and both reserve seat 11C. In the absence of conflict detection and resolution processes, they could both successfully book the same seat and not know of the problem until they checked in at the gate.

An active-active configuration can, and must, detect these types of data collisions. During the operation, both the prechange data and the changed data need to be captured. When delivering the data, the conflict detection process should match the prechange version of the data from the originating system with the preupdate version of the record on the target system. Matching a primary key or unique key is not sufficient to detect and resolve conflicts. Data lookups, transformations, and custom business logic could also come into play, and the active-active solution needs to facilitate these variations.

In the example, the solution would query the data before the seat reservation row is inserted. It could also be resolved by placing additional unique constraints on the objects or by invoking custom business logic.

Here's another example of a conflict involving inventory data on an item. Two users purchase a specific item from two retail Websites. In each case, the inventory is going to go from X to X -1. The current images of the records in each database are going to be different (expecting to see X), but the result is the same: one less item for sale. The detection mechanisms need to understand that this is more than just a numeric value—it's a quantity. The actual result in the record should

be X -2 (two units have been sold). The solution must have the ability to match the nonkey columns and obtain the before and after images of the records.

Simple Conflict Resolution Methodologies

In an active-passive environment, a conflict is considered an out-of-sync record and is handled individually and manually. Such discrepancies need to be immediately identified and handled, with as much automation as possible. It is also important to use the same resolution procedures on all the systems in the active-active environment, so that the same conflict receives the same resolution across the board.

The two most common conflict resolution methodologies are time stamp and trusted source. As an implementation practice, it is commonplace to have a database procedure for each operation type—one for inserts, one for updates, and one for deletes—that can handle 80 percent of the objects and their data transactions.

Time Stamp

With the time stamp methodology, in most cases, the record that was modified first (though in some cases, last) always wins. For this method to work, each record must contain a time stamp column that contains the date and time the record was inserted or updated. The easiest way to accomplish this, if it is not present in the data, is through a database trigger or by modifying the application code to place the time stamp in the column. A critical component to this method is to ensure that the clocks on all databases are identical and issues such as time zones and daylight savings are taken in to account.

To detect a conflict in a time stamp–based environment, there are two simple rules to follow. First, attempt to apply the row making sure that the preupdate time stamp from the source system is equal to the current time stamp in the target system. If the operation succeeds, there is no conflict. If it fails, then the second rule is to compare the time stamp of the current record in the target database to the after image of the time stamp from the source database. The row that has the oldest time stamp value wins.

Example #1: Reserving a Plane Seat Using Time Stamp Resolution

Returning to the same seat, same flight example, suppose that Adam (in Chicago) modifies his flight plan, and chooses seat 11C on the 12 noon (EST) flight. John (in New York) modifies his flight plan, and reserves seat 11C just a couple of seconds after Adam does.

This is what the transactions would look like

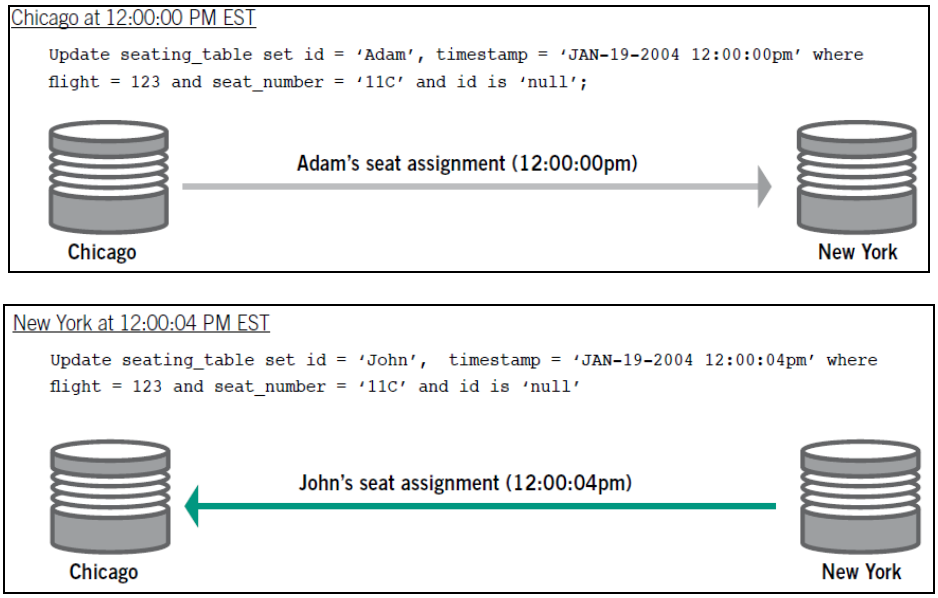


Figure 1. Comparison of the transactions by two users reserving the same seat on the same flight at nearly the same time.

To make this example a little less complex, assume that there is a unique constraint on the flight and seat number, and the lag at this point is six seconds. In the case of the record from Chicago, it is going to move to the New York server and be applied. It's going to fail due to a unique constraint on the flight and seat number columns. The conflict detection succeeded. Now, the time stamp–based conflict resolution is put into effect. Checking the time stamp from the current record (12:00:04pm – New York) in the database to the after image (12:00:00pm – Chicago), you'll notice that they are different. The record from Chicago occurred first, so the conflict resolution is going to modify the record already in the database, and change the user from John to Adam, and the time stamp to JAN-19-2004 12:00:00pm.

Here is what the resolution routine would look like

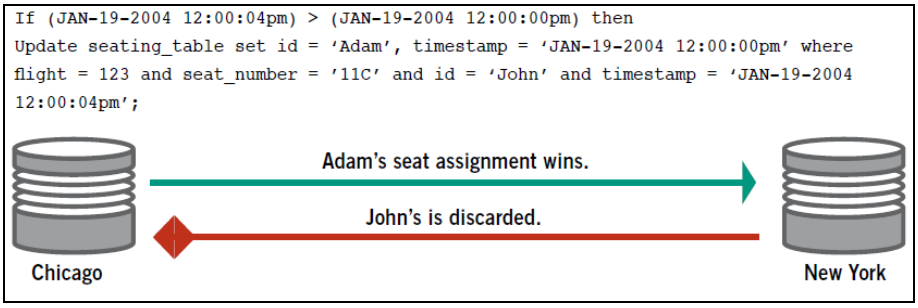


Figure 2. Resolving the conflict created by two users reserving the same seat on the same flight using a time stamp conflict resolution.

Now run through the same conflict detection and conflict resolution routines on the record from New York going to Chicago. Once again, the constraint violation occurs and detects the conflict. The resolution procedure checks the current record time stamp (12:00:00pm) and compares it to the after image from the New York record (12:00:04pm). They are different; but in this case, the record already in the Chicago database occurred first, and the record coming over from New York is discarded.

```
If (JAN-19-2004 12:00:00pm) > (JAN-19-2004 12:00:04pm) then
          Update…
```

Figure 3. With time stamp conflict resolution, most often, the earliest time stamp wins, but it can be configured for either one to win based on business rule requirements.

In the example, the if-then statement fails, so there is no update to process.

The final test of the resolution is to validate that records on the database servers in New York and Chicago are identical after the conflict was resolved. The original row from New York was overwritten with the information from Chicago, and the data in the Chicago database was not changed, so the data in both systems is identical.

Trusted Source

Another common conflict resolution approach is called trusted source. In these resolution routines, there is a single trusted source that is considered to always contain the correct data. This could be as simple as a server location, or as complex as a database user hierarchy. The implementation of this approach is straightforward—the decided trusted source wins.

Example #2: Out-of-Stock Resolution Using Trusted Source

In this scenario, a retail company is used as an example. The data is stored in two locations: Denver contains the online server and Portland houses the server for phone orders. For the server location, the online server is the trusted source. That means no matter what operation or change caused the conflict, Denver's transactions will always win. In this example, two people, one on the phone and one online, are going to order the very last widget in stock (item ABC) at the same time. The person online completes his order, fills in the shipping information and billing information, and then clicks Submit. Simultaneously, the person on the phone does the same, with the customer service representative clicking Submit on her system. The two transactions are subsequently verified for credit card payment. At this point, the inventory at both sites is reduced by one.

```
From Denver (Online)
Update inventory set quantity = quantity -1 where item = ABC;

From Portland (Phone Orders)
Update inventory set quantity = quantity -1 where item = ABC;
```

Figure 4. Two requests from two sources attempt to execute the same transaction at nearly the same time.

Now, the transactions at the top layer look the same, but what is actually read from the transaction logs is different. The data management solution needs to be smart enough to notice those changes and incorporate them into the query it is going to execute against the databases.

```
From Denver (Online)
Update inventory set quantity = 0 where item = ABC and quantity = 1;

From Portland (Phone Orders)
Update inventory set quantity = 0 where item = ABC and quantity = 1;
```

Figure 5. Both transactions believe the quantity to be 1 prior to the transactions occurring and both attempt to change it to zero.

In each case, the WHERE clause is appended with the before image of the value quantity, because that is the value changing. When the active-active solution attempts to apply the row, it will detect the conflict and try to resolve it.

When the conflict occurs, the resolution routines are instructed to take the data from the Denver location as the trusted source. The update to the inventory from the Portland database is discarded, and both sides have the same exact value in the inventory: zero of item ABC.

This resolution is not yet complete because the customer is on the line and needs to know that his order has been changed. The section titled "Complex Conflict Resolution" will address this further.

Hybrid Methodologies for Conflict Detection

Sometimes it is not enough to have just time stamp or trusted source approach for resolving conflicts; it might be necessary to combine them. If the resolution is based on time stamps, what happens if the conflict happens at the exact same time? Who wins? In this case, it is necessary to have a backup plan. If the two time stamps are exactly the same, then it goes into trusted source resolution routines. There are going to be times when one method does not work in every single conflict that could occur in your environment. The more you minimize the cases that resolution routines will fail, the less manual intervention will be required.

Up to this point, only the basics of active-active needs and conflict detection and resolution have been covered. You have seen examples of time stamp and trusted source resolutions. These routines are the foundation of most of the conflicts that will be encountered.

To truly succeed in creating an active-active database environment, it's necessary to delve deeper into the complex transactions that occur in real-world applications. Go beyond the single row and into the actual data and transactions that flow around them. The rest of this white paper explores more-complex implementations, focusing on how to avoid conflicts and how to deal with even the most complex transactions your application provides.

Complex Conflict Resolution

The methods previously discussed are fine for most of the tables normally involved in database transactions. However, there are times when more-complex routines are needed to handle the different issues that can occur. A number of different problems were alluded to in the first two examples where the conflicts were relatively simple. This section addresses more-complex conflicts. There are many different types of environments, transactions, and applications that might require more than simple conflict resolution approaches. Only an industrial-strength data management platform will provide you with the tools to handle each type of conflict.

Two common classes of complex resolutions will be discussed: quantitative values and transaction consistency.

Quantitative Resolution

Quantitative values include tangible values such as inventory, account balances, and sales information—anything that has its value incremented or decremented by a set amount. The out-of-stock example used earlier typifies a quantitative resolution.

Now assume that the inventory amount of item ABC from the time stamp example is two, instead of one. In this case, if both the online user and phone user only purchase a single item, then both should succeed—even if there was a conflict and the before image of the quantity column was different than what was expected. The final amount in the inventory would then be zero.

Example #3: Inventory Resolution Using Quantitative Resolution

In this example, the same database tables from the out-of-stock example will be used and applied to a different item—bowls—which have a quantity of 10 and an item code of 'Bowl.' Given this inventory amount, if two people order the bowl at the same time, they both should succeed in purchasing the item. The transaction in Figure 6 illustrates an online user in Denver trying to purchase three bowls, and a phone customer in Portland attempting to purchase five bowls.

```
From Denver (Online)
Update inventory set quantity = quantity -3 where item = 'Bowl';

From Portland (Phone Orders)
Update inventory set quantity = quantity -5 where item = 'Bowl';
```

Figure 6. Two sources attempt to order eight bowls from an inventory of ten bowls so both should succeed.

Now the transactions that the solution reads should look like this

```
From Denver (Online)
Update inventory set quantity = 7 where item = 'Bowl' and quantity = 10;

From Portland (Phone Orders)
Update inventory set quantity = 5 where item = 'Bowl' and quantity = 10;
```

Figure 7. Because both transactions occur in close proximity, the original inventory of ten has not been updated to reflect any changes. As a result, both transactions believe they are deducting the number of bowls ordered from an inventory of ten.

Look at the Denver order. It's going to be applied to the Portland database, but fail. Why? Because the before image of the quantity is expected to be 10, but instead it is 5. And the message from Portland is going to fail as well, because it expects a quantity of 10, and it sees 7. There is enough inventory for both people to complete their purchase, so neither time stamp nor trusted source is a suitable solution. In this case, these items need to be handled using a quantitative resolution. The resolution needs to look at the actual change in inventory at the source system and apply that to the target system, rather than using the actual numbers.

This is what the conflict resolution routines should apply to the database

```
Formula
Update inventory set quantity = quantity - (before image – after image)
where item = 'Bowls';

Apply to Portland from Denver (Online)
Update inventory set quantity = 5 - (10 - 7) where item = 'Bowl';

Apply to Denver from Portland (Phone Orders)
Update inventory set quantity = 7 - (10 -5) where item = 'Bowl';
```

Figure 8. When resolving conflicts using quantitative resolution, the resolution routine should evaluate both transactions to determine if either should fail.

After both of those statements succeed, the quantity of bowls remaining is two, both in Denver and Portland databases. The resolution routines produced the same result, the rows are still in sync, and there are two happy customers.

Transaction Consistency

Transaction consistency is one of the most challenging topics because different applications and different databases handle transactions in various ways. For the purposes of this white paper, certain platform-specific details are omitted. This section focuses on the problems that arise due to discarding a single operation within a completed transaction.

Taking examples two and three further into a realistic order scenario, the customer has purchased multiple items, but one of those items has a conflict. What are the business rules for handling this type of situation? The final order information will contain all items—including the one that was in conflict. Depending on how the application is structured, and the type of transactions that are generated, you could effectively handle this many different ways. The next example discusses one way that this can be handled.

Example #4: Discard Tables

Discard tables can be used to easily store information about a discarded record. When the conflict resolution routine executes, it does not throw away the changed data; it can place specific values into a discard table.

In this example, the application of the line items is done first, and after the line items are complete, another record with the total bill comes through, followed by a commit to the entire order. If one of the line items is discarded for any reason, the order number, item number, quantity, and item price are added into the discard table. When the total bill record is to be applied, the data management solution first queries the discard table to see if any of the line items had to be discarded. If they were discarded, then the same logic that created the total bill record is used to tabulate the adjusted amount.

Figure 9 is an example flow for code that can be used to create the resolution routine.

```
Apply line items as normally
If a line item conflict is detected and the line item is discarded then
Add a line to the discard file containing the pertinent info
Before applying bill total, check discard file
If the discard file contains records for this order number then
Recalculate bill amount
Apply corrected values to the database
Commit the entire transaction
```

Figure 9. When trascations cannot all be completed successfully, the rejected transactions should be sent to a discard table for review and processing.

To test the conflict resolution routine for this example, some real values need to be applied. The online user is going to purchase three bowls and one widget. The phone order user is going to purchase five bowls and one widget. This example is fairly complex, because each line item will have a conflict: there is only one widget available for purchase.

```
From Denver (Online)

Update inventory set quantity = quantity - 3 where item = 'Bowl';

Insert line_item (order#, item#, quantity_purchased, item_price) values
(1,XYZ,3,$10);

Update inventory set quantity = quantity - 1 where item = 'Widget';

Insert line_item (order#, item#, quantity_purchased, item_price) values
(1,ABC,1,$15);

Insert total_bill (order#, bill_amount) values (1, $45);

From Portland (Phone Orders)

Update inventory set quantity = quantity - 5 where item = 'Bowl';

Insert line_item (order#, item#, quantity_purchased, item_price) values
(2,XYZ,3,$10);

Update inventory set quantity = quantity - 1 where item = 'Widget';

Insert line_item (order#, item#, quantity_purchased, item_price) values
(2,ABC,1,$15);

Insert total_bill (order#, bill_amount) values (2, $65);
```

Figure 10. There is only one widget for sale, so one transaction will fail the conflict resolution routines will determine which to reject.

Going through the Denver order reveals the following:

- An inventory resolution routine is executed on the bowls (item 'Bowl'), and it is resolved just fine. No records were actually discarded, just changed, so there is no need to put a record into the discard table.
- The next line item is for a widget, and there is a problem: there is only one widget. As seen in example two, the widget order from Denver is the winner. Once again, no records are discarded, so you can continue on to the next item.
- The next is the total bill record. The discard table is queried to see if any records in this order were discarded, and the result is zero. Apply the total bill record as normal. The result is exactly what the online user would expect.

Going through the Portland order reveals similar results.

- An inventory resolution routine is executed on the bowls (item 'Bowl'), and it is resolved as well. Once again, no records were discarded, so skip the discard table phase.

- The widget line item (item 'Widget') has caused a problem. There are no more widgets available for the Portland user and the line item is discarded. A record is inserted into the discard table with the information from the line item order.
- The next operation is the total bill record, but you must query the discard table before it is applied. In this case, there is a record in the discard table for this order. The total bill record is modified and the bill amount is adjusted to $50, to account for the missing widget. You certainly don't want to charge someone for an item that was never shipped to them.

As this clearly shows, conflict detection and resolution can become very complex. In this example, the process isn't complete yet because the two databases are not identical. The quantity values are correct, as seen from examples two and three; however, the order entries are different. If you query the Denver database for order# 2, then you will see that it contains a single line item, and a total bill amount of $50. If you query the Portland database for order# 2, you will see that it contains two line items and a total bill amount of $65. The record (the quantity) that caused the conflict caused another record (the line item) to be discarded. This information needs to be sent back to the Portland database. That process will be covered next.

Conflict Notification and Tracking

Once the conflict has been detected, it is extremely important that any changes made to the database are logged for future reference, automated corrections, human intervention, and also to provide an audit trail for the resolution operations.

Exceptions

It helps to have an exceptions or a recovery table that contains the changes that were made by the automated resolution routines. Logging these changes makes it easy to find out what conflicts occurred, how they were handled, and what resolution was taken. In many cases, these tables can also assist in troubleshooting complex environments. As a best practice, as much information should be logged in to these tables as possible. Below is a sample table for capturing this data.

EXCEPTIONS

FIELD	DATATYPE	PRIMARY KEY
EXC_NUM	Number	Yes
OP_TYPE	char(1)	
TABLE_NAME	varchar(500)	
ROW_PK_VALUE	Number	
APPLIED_IMAGE	varchar(500)	
BEFORE_IMAGE	varchar(500)	
OVERWRITTEN_IMAGE	varchar(500)	
EXC_TIMESTAMP	Date	
APPLIED_TIMESTAMP	Date	
OVERWRITTEN_TIMESTAMP	Date	
DISCARDED_RECORD	char(1)	
NOTIFICATION_RQD	char(1)	
NOTIFICATION_SENT	char(1)	

- OP_TYPE contains the operation type, I for insert, U for update, D for delete.
- TABLE_NAME contains the name of the table that the exception occurred on.
- ROW_PK_VALUE contains the primary key value for the row that was being modified. This is used to look up the row again if you needed to verify any data.
- APPLIED_IMAGE holds all the data that was applied to the target row.
- BEFORE_IMAGE contains the preimage of the record that was sent across to the target. This is useful to compare to the OVERWRITTEN_IMAGE to see if the conflict resolution routine was correct in its resolution.
- OVERWRITTEN_IMAGE contains the data that was in the record on the target side that is being overwritten. This is very helpful for notification and resolving any discrepancies with data that might not have been solved by the resolution routines.
- EXC_TIMESTAMP is used to store the time that the conflict was resolved.
- APPLIED_TIMESTAMP is used for time stamp resolution. It is important to log the applied time stamp of the winning row.
- OVERWRITTEN_TIMESTAMP is when using time stamp resolution. You should log the time stamp that was in the original row that was being overwritten. Ensuring that this column is newer than the APPLIED_TIMESTAMP column provides additional proof that the time stamp–based routine was successful.
- DISCARDED_RECORD contains a single value (Y/N) stating that a row was discarded to the discard table.
- NOTIFICATION_RQD and NOTIFICATION_SENT contain a Y/N value to indicate if the user should be/has been notified.

The structure of the exceptions table can be changed depending on the application needs and the resolution routines that are used.

For example, if you are doing trusted source–based conflict detection and resolution, you could remove the time stamp column and replace them with host name columns. Or in a hybrid environment, you might want to have both the host name and time stamp. Once you are confident that your routines are working, the amount of data that is logged can be reduced to the bare minimum. This will reduce the overhead of the resolution routines and improve your load-balancing benefits. However, if a single exceptions table is not enough to handle your applications needs, then you could have a separate exceptions table for each module, or even for each table. Sometimes the automated routines just can't handle every type of conflict, and manual intervention is needed.

Resolution Notification

To this point, the conflict has been detected, resolved, and logged. In some cases, the conflict and its resolution should be communicated to the appropriate parties to inform them that their expected operation has been changed.

In most active-active database environments, this is not necessary; most quantitative resolutions do not require any notification. It is not necessary to tell a user that the quantity of items had a conflict, especially if each order was successfully filled to completion.

However, in the case of the same seat, same flight example, the conflict was handled correctly in the database, but both the New York user and the Chicago user think they are going to be in seat 11C. How do you notify the New York user that he no longer has a seat, or that he needs to choose another one?

The active-active configuration should facilitate resolution notification. In this example, the business rules can dictate how to handle communication of the resolution. The user can be assigned another empty seat, or could simply be told that he needs to get his seat assignment at the gate. Because most businesses would like to automate such a routine task, the active-active solution should provide the necessary framework.

Example #5: Automated Resolution Notification

This example explores the procedure for setting up an automated resolution notification. In this method, the last two fields of the exceptions table example are used: NOTIFICATION_RQD and NOTIFICATION_SENT. Returning to the same seat, same flight example, the user from New York needs to be notified that his seat is no longer available. The easiest way to handle these types of issues is to create a batch load that periodically updates data in the exceptions table. This batch job will handle the notifications.

The batch job to send notifications would follow the procedure outlined in Figure 11.

```
For each row in the exceptions table that contains NOTIFICATION_RQD = 'Y', and
NOTIFICATION_SENT ='N' do

If TABLE_NAME = 'SEATING_INFO' and OP_TYPE = 'I' then

Obtain a new vacant seat for the user

Construct email to customer informing them of seating change

Send email

Update exception record change NOTIFICATION_SENT='Y'

If TABLE_NAME = 'SEATING_INFO' and OP_TYPE ='U' then

If …

If …
```

Figure 11. Batch processes can be used to notify users when data conflicts are identified.

The seating problem that arose from the New York user would have been flagged as NOTIFICATION_RQD = 'Y' and NOTIFICATION_SENT = 'N' as soon as the conflict was resolved. The batch job would then process this record, select an empty seat, and send the user their new seating assignment.

The same process that handles notifications in this example can be used for a number of different operations, such as providing information to the database administration team on the number of exceptions, types of exceptions, and details about those exceptions. Resolving these issues with the least impact to your business is critical, and minimizing conflicts reduces the complexity significantly.

The following section discusses the different ways to reduce—and ideally avoid—conflicts.

Minimizing Conflicts

Even with the best conflict resolution routines, there are still going to be issues that are not easy to handle. A key goal in building active-active environments is minimizing the amount of conflicts that happen. If you can avoid conflicts on 99 percent of the tables, or even on a single type of operation (such as a delete), then you can save an enormous amount of time implementing and maintaining your environment. This section is going to discuss several ways that conflicts can actually be avoided or reduced. Any way to reduce the amount of conflicts will provide a better experience for all stakeholders.

Application Segregation

Segregating application users is one way to avoid conflicts. Both inventory- and quantity-related conflicts can easily be addressed this way. Each server that is balancing the user load contains the primary source for a certain type of product or service.

Stock trades are one type of product that cannot have conflicts, but can still benefit from an active-active environment. By moving the trades on companies that begin with the letters A–M to one server and the letters N–Z to another, you can avoid any conflicts of trading the same stock at the same time. Removing the conflicts in such critical situations can really allow this type of configuration to succeed, but also provide phenomenal results. Through the use of an application server, this can be made even easier by having a pool of connections to issue the trades, rather than having the users log on to both systems.

Another way to do this is with user names. If people are frequently changing account information, or even account balances, you can move an equal number of people to each server and avoid nearly all conflicts. Because the changed data from all servers is going to be propagated to all the other nodes, in the event that the primary node for a group of accounts goes down, they can be routed to a secondary node until everything is back up and running smoothly. Some companies have gone as far as designating primary and secondary load distribution methods to ensure that a single server is not overly burdened by an outage elsewhere in the environment. The more servers that are involved, the more it will help to have secondary application segregation and load-balancing strategies.

Primary Key Generation

Many insert conflicts can be avoided just by handling primary key issues. This is probably the easiest method of reducing the number of conflicts that can arise. Simply alternate the primary key generation sequences or routines. For a two-server environment, have one generate even primary keys, the other odd. For an n-server environment, have each generate keys starting at a different value (1, 2, 3, 4, 5,…n) and have their sequences increment by the number of servers in the environment. For a three-server environment, server one starts at 1 and increments by three (1, 4, 7, 10, 13), server two starts at 2 and increments by three (2, 5, 8, 11, 14), and server three starts at 3 and increments by three (3, 6, 9, 12, 15).

However, even though it is extremely easy to implement, this method might not be available to all applications.

Conflict Avoidance

This goes against much of what has been discussed so far; however, there are going to be certain yet rarer cases where conflicts can just be ignored. One case could be in deleting information. If an item is going to be discontinued and the store manager deletes it out of the inventory system, and the database administrator does it at the same time, it really doesn't matter if someone is

from a trusted source, or someone committed their delete just a split second before the other user. A delete is a delete, and the result will be a discontinued product. Conflicts that arise in these situations can just be ignored. At first, you might want to keep track of them in the exceptions table just to see how often they occur, but once you are confident that there is no harm being done, they can usually be skipped.

Oracle GoldenGate for Active-Active Databases

To effectively identify and respond to data conflicts, organizations need control over the movement of data across the enterprise. When delivering active-active database configurations, Oracle GoldenGate delivers the infrastructure necessary to streamline data movement to ensure seamless operations.

Oracle GoldenGate provides real-time, logical data replication capabilities to move date across heterogeneous IT environments with subsecond speed. The application platform consists of decoupled modules that can be combined across systems to provide maximum flexibility, modularity, and performance. It is an asynchronous solution with synchronouslike behavior.

This architecture facilitates the movement and management of transactional data in four simple, yet powerful steps.

- **Capture.** Oracle GoldenGate's change data capture technology identifies and replicates data changes from database log files in real time using a nonintrusive, high-performance, low-overhead approach. Oracle GoldenGate can capture data from any number of databases, including Oracle, DB2 UDB, DB2 OS/390, as well as those running on HP NonStop/Enscribe, SQL/MP, SQL/MX, and Sybase. All data changes are captured through direct access to native database transaction logs—redo logs where applicable—to minimize any impact on system performance.
- **Route.** Once captured, changed data transactions are placed in queue files (called Trail Files) and can be delivered to any data target including message queues. There are no geographic distance constraints or impacts. Oracle GoldenGate uses a variety of transport protocols as well as compression and encryption techniques prior to routing changed data.
- **Enhance.** To optimize performance and data management capabilities, at any point prior to delivering changed data from the host to the target system, Oracle GoldenGate can execute a number of built-in functions, such as filtering and transformation.
- **Apply.** Oracle GoldenGate can apply changed data to multiple targets with subsecond latency to ensure transaction integrity with features for conflict detection and resolution.

Key technical features that are intrinsic to Oracle GoldenGate's support for active-active configurations include the following:

- **Flexible topology support and bidirectional configurations.** Using a decoupled, modular design, Oracle GoldenGate can support a wide variety of topologies, including one-to-one, one-to-many, many-to-one, and many-to-many—for both unidirectional and bidirectional configurations. For additional scalability, cascading topologies can be created to eliminate any potential bottlenecks. By staging specific sets of database changes on the source or target system, different data replication requirements can be met through a single pass on the datasource. Each set of staged data can contain unique or overlapping sets of data.
- **Conflict detection and resolution.** When two systems are processing data transactions and the activity is shared across both systems, detecting and addressing conflicts across them becomes an essential requirement for any active-active configuration. Oracle GoldenGate provides a wide variety of conflict detection and resolution options to provide the necessary flexibility and adaptability for a range of requirements. Conflict detection and resolution options can be implemented globally, object by object, based on data values and complex filters, and through event-driven criteria including database error messages.
- **Heterogeneity.** Oracle GoldenGate decouples the datasource and target, which enables the application to easily facilitate heterogeneity. In addition, changed data is staged between the systems in a universal data format (Trail Files) to facilitate portability. This provides flexibility in the choice of hardware, operating system, and databases for sources and targets, and can accommodate unplanned outages as well as system, database, and application maintenance activities—without interruption. Unlike architectures that implement a tight "process-to-process" coupling, this decoupled architecture provides each module the ability to perform its tasks independently of other modules or components.
- **Subsecond latency.** Oracle GoldenGate's capture, enhance, route, and delivery processes can move thousands of committed data transactions between systems with subsecond speed. There is very minimal impact on the source system and infrastructure, thus ensuring high performance with high data volumes.

Whether you are using any mix of Oracle, Sybase, SQL Server, DB2, or even HP NonStop or Teradata, Oracle GoldenGate is an excellent solution for improving the performance, accessibility, and availability of your data across the enterprise.

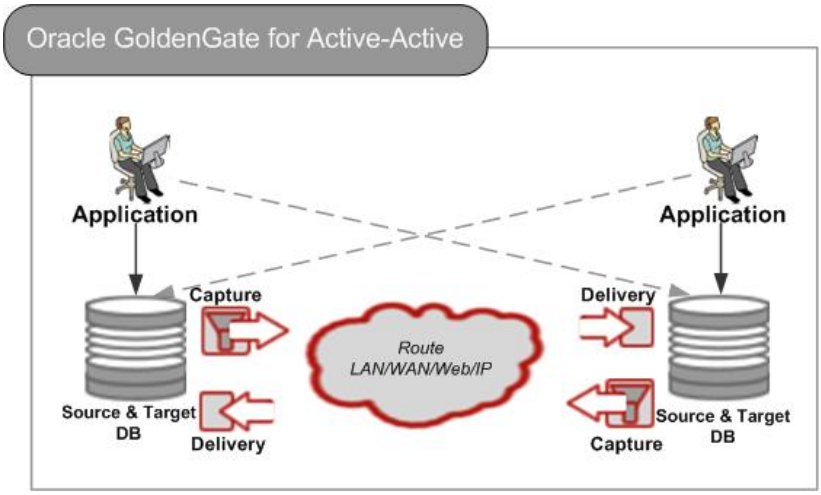


Figure 12. In an active-active configuration, Oracle GoldenGate ensures continuous system access and availability.

Oracle GoldenGate Data Definition Language Replication

Beginning with Release 9.5, Oracle GoldenGate provides Data Definition Language (DDL) replication for Oracle Database to allow a user to capture and propagate DDL changes from one system to another. To use Oracle GoldenGate's DDL replication capabilities in an active-active configuration, select a single system to which to apply the DDL changes. Configure the DDL replication from a single system to apply those changes to all the other environments in the active-active solution. The system that the DDL is manually applied to is the "host" system, and the systems that the DDL is being applied to are the "subscriber" systems. Even though DML will flow bidirectionally between all systems, DDL replication is only one way. This helps to eliminate problems arising from multiple people trying to make a DDL change to the same stored procedure or another object. Because it is not possible to perform conflict detection or resolution with DDL commands, this approach helps create a conflict-free environment.

Depending on the changes made to the objects involved in replication, the conflict detection and resolution rules could also change. Prior to applying these DDL changes to a production environment, a new set of conflict detection and resolution rules, as well as Oracle GoldenGate parameter files, should be tested in a stable environment.

If the changes that are being applied to the host system do not modify the table the way an update, delete, or insert statement would, then the DDL can be run without stopping activity on the subscriber databases. Such changes would include indexes, constraints, adding new tables, or stored procedure changes. In that situation, the conflict resolution and detection procedures do not need to change, and the DDL can be applied to the host system at any time.

If the changes include DDL operations that drop columns, rename tables, or modify objects in such a way that the DML statement, or Oracle GoldenGate parameter files, would also change,

then the subscriber systems need to be quiesced. For example, if a column is dropped on the host, and at the same time, someone inserts a value into that column on a subscriber, that could cause conflicts that are not easily handled by conflict resolution. This would also require changes to conflict detection and resolution procedures or parameter files. Once the subscriber systems are quiesced, ensure that the Oracle GoldenGate Capture and Delivery processes on all systems have completed their activities. If necessary, replace the Oracle GoldenGate parameter files at this time to account for any changes to replicated objects, as well as conflict detections and resolutions. Start the Oracle GoldenGate Capture module on the host system and the associated Oracle GoldenGate Delivery modules on the subscriber systems, and apply the DDL operations on the host system. Once they have been captured and applied on the subscriber systems, restart any remaining Capture and Delivery modules. At this point, users can access the subscriber servers again, and normal operations can continue.

Conclusion

Active-active implementations can seem like a daunting task, especially when building some of the more-complex conflict detection and resolution routines. This should not discourage anyone from pursuing such a solution for their business. There are tremendous benefits to improve database performance, response times, and availability and to achieve significant scalability gains by implementing an active-active database environment.

Databases and servers will fail—it is inevitable. To be ready for this, companies have invested hundreds of millions of dollars on disaster recovery centers and 24/7 operations. Active-active configurations deliver excellent business continuity results in a cost-effective way. Oracle GoldenGate is uniquely designed to support robust conflict detection and resolution processes to optimize data accuracy and integrity in active-active configurations.



Best Practices for Conflict Detection and Resolution in Active-Active Database Configurations Using Oracle GoldenGate
Updated November 2009

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com



Oracle is committed to developing practices and products that help protect the environment

Copyright © 2008, 2009, Oracle and/or its affiliates. All rights reserved. This document is provided for information purposes only and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

0109