ORACLE 11g
FUSION MIDDLEWARE
GOLDENGATE

An Oracle White Paper
Updated December 2012

Best Practices for Conflict Detection and Resolution in Active-Active Replication Environments Using Oracle GoldenGate

ORACLE

Contents

Executive Overview

One of the most effective ways to enable increased availability and performance for database infrastructures is to establish an active-active replication environment, which distributes database transactions across multiple databases. Beyond disaster recovery and fault tolerance, active-active database replication configurations facilitate continuous operations, offer additional raw computing capacity, and provide the flexibility to optimize workload management. However, implementing an active-active replication 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 11*g* Release 2 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 replication 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 replication 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 replication 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 replication 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.

Oracle GoldenGate for Active-Active Replication

To effectively identify and respond to data conflicts, organizations need control over the movement of data across the enterprise. When offering active-active database replication configurations, Oracle GoldenGate 11*g*R2 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 11*g*R2 can capture data from any number of databases, including Oracle, DB2 for Linux/Unix/Windows, DB2 for i-Series, DB2 for z/OS, 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 replication configurations include the following:

- **Flexible topology support and bidirectional configurations.** Using a decoupled, modular design, Oracle GoldenGate can support a wide variety of replication 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 data source. 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 replication configuration. Oracle GoldenGate 11*g* Release 2 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 data source 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 Database, 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.

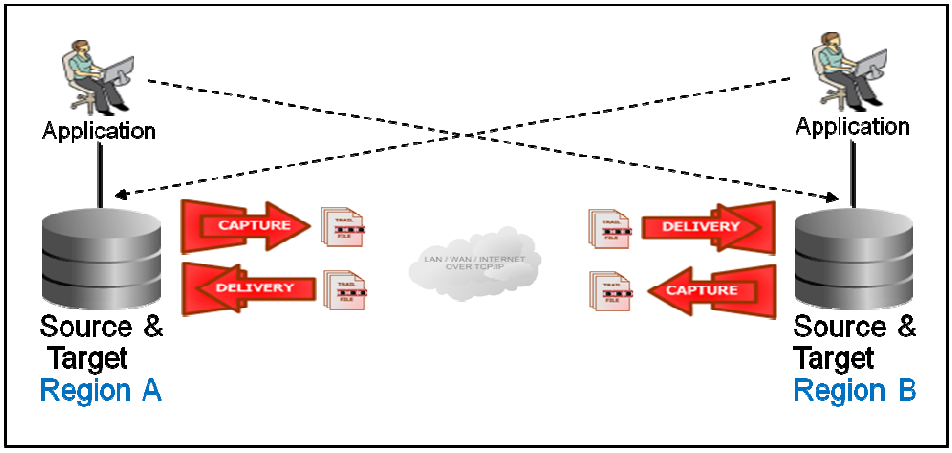

Figure 1. In an active-active replication configuration, Oracle GoldenGate delivers continuous system availability.

Key Requirements for Active-Active Replication Configurations

In an active-active replication configuration—also referred to as a master-master, dual-master, multi-master, 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 replication configuration increases the overall capacity, resulting in improved response times and enhanced system performance.

Active-active replication 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 replication 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 replication 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 replication configurations, the data movement has to be asynchronous yet provide "synchronouslike" behavior.

Conflict Detection and Resolution

In an active-active replication 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 replication solution must facilitate different conflict detection and resolution mechanisms.

Heterogeneous Environment Support

Systems in an active-active replication 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 replication solution must provide heterogeneous infrastructure support.

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 replication 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 replication 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

It is critical that table primary keys and unique indexes are unique for each database participating in an active-active replication environment. The table primary keys and unique indexes for each database in the active-active replication environment must contain information that identifies which database the operation occurred on. Insert conflicts can be avoided and almost eliminated by implementing database specific primary keys and unique indexes and is probably the easiest method of reducing the number of conflicts that can arise.

For Oracle databases that are configured to use sequences for table primary keys, 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.

Allowable Conflicts

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 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 just to see how often they occur, but once you are confident that there is no harm being done, they can usually be skipped.

Conflict Detection

Even through application segregation and primary key isolation techniques can significantly reduce conflicts in an active-active replication configuration, there may still be conflict scenarios that can't be avoided and need to be addressed through conflict detection and resolution. Oracle GoldenGate 11*g* Release 2 provides built-in logic to reduce the complexity of configuring conflict detection and resolution for successful active-active replication deployments.

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.

Take the case of an airline reservation system that has an active-active replication configuration with one database located in the United States used for online transactions and the other database in Europe used for over the phone transactions. Two customers, Joe and Kevin, are reserving their seats for flight 123 from Barcelona to New York City at approximately the same time. Joe is making his seat reservation over the phone, so when the operator submits his seat reservation for 2A it is made to the European Database. Kevin is making his seat reservation online at approximately the same time, so when he submits his seat reservation for 2A it is made to the U.S. Database. In the absence of conflict detection and resolution processes, Joe and Kevin would have both reserved seat 2A successfully.

An active-active replication configuration can, and must, detect these types of data collisions. During the operation, both the pre-change data and the changed data need to be captured. When delivering the data, the conflict detection process should match the pre-change version of the data from the originating system with the pre-update 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 replication solution needs to facilitate these variations.

In the example, the solution must have the ability to match the non-key columns and obtain the before and after image of the records. It could also be resolved by placing additional unique constraints on the objects or by invoking custom business logic.

Simple Conflict Resolution Methodologies

In an active-passive replication 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 replication environment, so that the same conflict receives the same resolution across the board.

The two most preferred 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 timestamp 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 timestamp in a table column.

It is critical to ensure that the clocks on all databases are identical to one another and it's recommended that all database servers are configured to maintain accurate time through a time server using the network time protocol (NTP). Even in environments where databases span different time zones, all database clocks must be set to the same time zone or Coordinated Universal Time (UTC) must be used to maintain accurate time. Failure to maintain accurate and synchronized time across the databases in an active-active replication environment will result in data integrity issues.

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

Example #1: Reserving an Airline Seat Using Time Stamp Resolution

An airline reservation system uses the SEAT_RESV table to store flight and seat reservation information for a passenger. The SEAT_RESV table is pre-populated with the airline's flight and seat information while the PASSENGER column is null until a passenger reserves a seat and the table is updated. For example, before a passenger reserves seat 2A on flight 123, the table would look like below.

(SEAT_RESV table)

ID	PASSENGER	SEAT	FLIGHT	LAST_UPDATE
1		2A	123	10-10-2012 8:00:00

Returning to the previous example of Joe and Kevin reserving a seat, suppose that Joe (by phone) reserved seat 2A at 10:30:00am, while Kevin (online) reserved seat 2A one second after Joe did at 10:30:01am.

```
Update to European Database (SEAT_RESV table) Over the Phone
Update EURO.SEAT_RESV set PASSENGER = 'Joe', LAST_UPDATE = (timestamp '2012-10-15 10:30:00'
where ID = 1;

Update to U.S. Database (SEAT_RESV table) Online
 Update US.SEAT_RESV set PASSENGER = 'Kevin', LAST_UPDATE = (timestamp '2012-10-15 10:30:01'
where ID = 1;
```

Before Oracle GoldenGate replicates the seat reservation update transactions to the European and U.S. Databases, the seat reservation tables would look like the following.

European Database (SEAT_RESV table)

ID	PASSENGER	SEAT	FLIGHT	LAST_UPDATE
1	Joe	2A	123	10-15-2012 10:30:00

U.S. Database (SEAT_RESV table)

ID	PASSENGER	SEAT	FLIGHT	LAST_UPDATE
1	Kevin	2A	123	10-15-2012 10:30:01

How would Oracle GoldenGate 11*g* Release 2 be configured to detect this conflict and resolve it using a timestamp?

Setup Before Image Capture

As mentioned above, the pre-update or before image of the timestamp column needs to be captured from the source system, so that it can be compared against the target timestamp column to determine if a conflict has occurred. In order for the timestamp resolution method to be successful, the application must always value the timestamp column when updating the table. In this example, it is required that the airline reservation system always values the timestamp column LAST_UPDATE when updating the SEAT_RESV table. If the airline reservation system cannot be configured to always value the timestamp column LAST_UPDATE, then the timestamp resolution method cannot be used.

The following line would be added to the Oracle GoldenGate Capture parameter file for the European and U.S Databases. The parameter instructs the Capture process to capture the before values in addition to the after values for the key column ID and the LAST_UPDATE column when an update occurs on the SEAT_RESV table.

```
European Database Capture

        TABLE euro.seat_resv, GETBEFORECOLS(ON UPDATE KEYINCLUDING(last_update));

U.S. Database Capture

        TABLE us.seat_resv, GETBEFORECOLS(ON UPDATE KEYINCLUDING(last_update));
```

Configure Oracle GoldenGate Delivery

Now that the Oracle GoldenGate Capture processes have been configured to capture the before image data required to detect a conflict, the Oracle GoldenGate Delivery processes need to be configured to detect the conflict and resolve it.

The following lines would be added to the Oracle GoldenGate Delivery parameter file for the European and U.S. Databases.

```
European Database (Delivery)

        MAP us.seat_resv, TARGET euro.seat_resv,
        COMPARECOLS (ON UPDATE KEYINCLUDING (last_update)),
        RESOLVECONFLICT (UPDATEROWEXISTS, (DEFAULT, USEMIN (last_update)));

U.S. Database (Delivery)

        MAP euro.seat_resv, TARGET us.seat_resv,
        COMPARECOLS (ON UPDATE KEYINCLUDING (last_update)),
        RESOLVECONFLICT (UPDATEROWEXISTS, (DEFAULT, USEMIN (last_update)));
```

The parameter COMPARECOLS instructs the Delivery process to compare the before image values of the LAST_UPDATE column with the current LAST_UPDATE column value when an update occurs to the SEAT_RESV table. The KEYINCLUDING parameter instructs the Delivery process to use the primary key ID for uniqueness when comparing rows. If the LAST_UPDATE column values match, the update transaction will complete normally, as no conflict has occurred. If the LAST_UPDATE column values don't match, and in this example the LAST_UPDATE values are different, a conflict has occurred and the RESOLVECONFLICT parameter would fire. The RESOLVECONFLICT parameter, in this example, instructs the Delivery process to update the PASSENGER and LAST_UPDATE columns when the LAST_UPDATE value is older than the current value through USEMIN. In this example, Kevin's seat reservation would be overwritten by Joe's seat reservation in the U.S Database, as Joe's LAST_UPDATE value is 1 second older than Kevin's. Kevin's seat reservation transaction would be ignored in the European Database, as Joe's LAST_UPDATE value is 1 second older than Kevin's. The databases remain in sync as seat 2A is now reserved by Joe in both databases.

Confirm Conflict Detection Resolution Succeeded

Issue the following command in GGSCI on the European and U.S. Database servers to display the current CDR statistics for the Delivery (also called Replicat) process. This is a quick method to determine if conflicts are being resolved successfully by the Delivery process.

```
GGSCI> STATS REPLICAT <group>, REPORTCDR

Total inserts                          0.00
Total updates                          1.00
Total deletes                          0.00
Total discards                         0.00
Total operations                       1.00
Total CDR conflicts                    1.00
CDR resolutions succeeded              1.00
CDR UPDATEROWEXISTS conflicts          1.00
```

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 always wins.

Example #2: Reserving an Airline Seat Using Trusted Source Resolution

The airline reservation system example will be used again, but in the example the LAST_UPDATE column does not exist on the SEAT_RESV table. To handle conflicts, the airline established the rule that the U.S. Database, used for online reservations, will be the trusted source. This means no matter what operation or change caused the conflict on the SEAT_RESV table, the U.S. Database transactions will always win. In this example, Joe and Kevin are making their seat reservations for flight 123 from Barcelona, Spain to New York City. Joe is making his seat reservation over the phone, while Kevin is making his online. The operator assisting Joe completes his seat reservation for 2A over the phone and clicks submit. Simultaneously, Kevin has also completed selecting seat 2A online and clicks submit.

Update to European Database (SEAT_RESV table) Over the Phone

```
Update EURO.SEAT_RESV set PASSENGER = 'Joe' where ID = 1;
```

Update to U.S. Database (SEAT_RESV table) Online

```
Update US.SEAT_RESV set PASSENGER = 'Kevin' where ID = 1;
```

Before Oracle GoldenGate replicates the seat reservation update transactions to the European and U.S. Databases, the seat reservation tables would look like the following.

European Database (SEAT_RESV table)

ID	PASSENGER	SEAT	FLIGHT
1	Joe	2A	123

U.S. Database (SEAT_RESV table)

ID	PASSENGER	SEAT	FLIGHT
1	Kevin	2A	123

How would Oracle GoldenGate 11g Release 2 be configured to detect this conflict and resolve it using a trusted source?

Setup Before Image Capture

When using the trusted source method, pre-update or before image values for all columns in the table must be captured from the source system, so that they can be compared against all columns in the target database to determine if a conflict has occurred. To accomplish this, all columns

must be added to the SEAT_RESV table supplemental log group and the Oracle GoldenGate Capture processes need to be configured to always capture all column before image values.

Issue the following command to add the SEAT_RESV table supplemental log group using all the table columns ID, PASSENGER, SEAT, and FLIGHT. The SEAT_RESV supplemental log group needs to be created in both the European and U.S. Databases.

```
GGSCI> ADD TRANDATA SEAT_RESV, COLS (PASSENGER, SEAT, FLIGHT)
```

The following line would be added to the Oracle GoldenGate Capture parameter file for the European and U.S Databases. The parameter instructs the Capture process to capture the before values in addition to the after values for all columns when an update occurs on the SEAT_RESV table.

```
European Database Capture

        TABLE euro.seat_resv, GETBEFORECOLS(ON UPDATE ALL);

U.S. Database Capture

        TABLE us.seat_resv, GETBEFORECOLS(ON UPDATE ALL);
```

Configure Oracle GoldenGate Delivery

Because the U.S. Database is the trusted source in this configuration, the GoldenGate Delivery process applying transactions to the European Database would always overwrite the values in the SEAT_RESV table when a conflict occurs. The GoldenGate Delivery process applying transactions to the SEAT_RESV table in the U.S. Database would always ignore the transactions that came from the European Database when a conflict occurs.

By default the GoldenGate Delivery process will write to the target database unless the map statement is configured to explicitly ignore it. Because of this, the COMPARECOLS and RESOLVECONFLICT parameters would only need to be added to the U.S. Database Delivery parameter file. There is no reason to check for conflicts of transactions coming from the U.S. Database, because the U.S. Database is the trusted source and always wins. The European Database parameter file would just contain the standard map statement, while the U.S. Database parameter file map statement would be configured to ignore conflicts.

```
European Database (Delivery - Overwrite)

        MAP us.seat_resv, TARGET euro.seat_resv;

U.S. Database (Delivery - Ignore)

        MAP euro.seat_resv, TARGET us.seat_resv,
        COMPARECOLS (ON UPDATE ALL),
        RESOLVECONFLICT (UPDATEROWEXISTS, (DEFAULT, IGNORE));
```

The parameter COMPARECOLS instructs the Delivery process to compare the before image values of columns when an update occurs to the SEAT_RESV table. The ALL parameter instructs the Delivery process to compare all table columns. If all column values match, the update transaction will complete normally, as no conflict has occurred. If any of the column

values don't match, and in this example the PASSENGER values are different, a conflict has occurred and the RESOLVECONFLICT parameter would fire. The RESOLVECONFLICT parameter instructs the Delivery process to IGNORE the transaction if it came from the European Database. In this example, Kevin's seat reservation would overwrite Joe's seat reservation in the European Database, because Kevin's transaction occurred in the trusted source U.S. Database. Joe's seat reservation transaction would be ignored by the GoldenGate Delivery process applying to the U.S. Database, because the transaction came from the European Database. The databases remain in sync as seat 2A is now reserved by Kevin in both databases.

Confirm Conflict Detection Resolution Succeeded

Issue the following command in GGSCI on the European and U.S. Database servers to display the current CDR statistics for the Delivery (Replicat) process. This is a quick method to determine if conflicts are being resolved successfully by the Delivery (Replicat) process.

```
GGSCI> STATS REPLICAT <group>, REPORTCDR

Total inserts                          0.00
Total updates                          1.00
Total deletes                          0.00
Total discards                         0.00
Total operations                       1.00
Total CDR conflicts                    1.00
CDR resolutions succeeded              1.00
CDR UPDATEROWEXISTS conflicts          1.00
```

Quantitative 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 the more-complex quantitative conflict resolution method.

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.

Example #3: Flight Seats Available Using Quantitative Resolution

Prior to Joe and Kevin reserving their seats for flight 123, they had to purchase their initial tickets. For this example, flight 123 has 10 seats available and Joe is purchasing 3 tickets for his family over the phone which utilizes the European Database, while simultaneously Kevin is purchasing 4 tickets for his family online which utilizes the U.S. Database.

```
Update to European Database (FLIGHT_INV table) Over the Phone
Update EURO.FLIGHT_INV set SEATS_AVAIL = seats_avail – 3 where FLIGHT = 123;
```

```
Update to U.S. Database (FLIGHT_INV table) Online

Update US.FLIGHT_INV set SEATS_AVAIL = seats_avail – 4 where FLIGHT = 123;
```

Before Oracle GoldenGate replicates the flight inventory update transactions to the European and U.S. Databases, the flight inventory tables would look like the following.

```
European Database (FLIGHT_INV table)
FLIGHT    SEATS_AVAIL
123              7
U.S. Database (FLIGHT_INV table)
FLIGHT    SEATS_AVAIL
123              6
```

When Joe's transaction is replicated to the U.S. Database it will fail. Why? Because the before image value of SEATS_AVAIL is expected to be 10, but instead it is 6 from Kevin's purchase of 4 tickets. Kevin's transaction will also fail for the same reason when it is replicated to the European Database, because it's also expecting the before image value of SEATS_AVAIL to be 10. There are enough seats available for both Joe and Kevin to complete their purchase, so neither timestamp nor trusted source is a suitable solution. In this case, SEATS_AVAIL needs to be handled using a quantitative resolution. The resolution needs to look at the actual change in SEATS_AVAIL at the source system and apply that to the target system, rather than using the actual numbers.

How would Oracle GoldenGate 11*g* Release 2 be configured to detect this conflict and resolve it using a quantitative resolution?

Setup Before Image Capture

The preupdate or before image of the SEATS_AVAIL column value needs to be captured from the source system, so that it can be compared against the target SEATS_AVAIL column value to determine if a conflict has occurred. To accomplish this, the SEATS_AVAIL column needs to be added to the FLIGHT_INV table supplemental log group and the Oracle GoldenGate Capture processes need to be configured to always capture the SEATS_AVAIL column's before image values.

Issue the following command to add the FLIGHT_INV table supplemental log group using the primary key column FLIGHT and the inventory column SEATS_AVAIL. The FLIGHT_INV table supplemental log group needs to be created in both the European and U.S. Databases.

```
GGSCI> ADD TRANDATA FLIGHT_INV, COLS (SEATS_AVAIL)
```

The following line would be added to the Oracle GoldenGate Capture parameter file for the European and U.S Databases. The parameter instructs the Capture process to capture the before values in addition to the after values for key column ID and the SEATS_AVAIL column when an update occurs on the FLIGHT_INV table.

```
European Database Capture

        TABLE euro.flight_inv, GETBEFORECOLS(ON UPDATE KEYINCLUDING (seats_avail));

U.S. Database Capture

        TABLE us.flight_inv, GETBEFORECOLS(ON UPDATE KEYINCLUDING (seats_avail));
```

Configure GoldenGate Delivery

Now that the Oracle GoldenGate Capture processes have been configured to capture the before image data required to detect a conflict, the Oracle GoldenGate Delivery processes need to be configured to detect the conflict and resolve it.

The following lines would be added to Oracle GoldenGate Delivery parameter file for the European and U.S. Databases.

```
European Database (Delivery)

        MAP us.flight_inv, TARGET euro.flight_inv,
        COMPARECOLS (ON UPDATE KEYINCLUDING (seats_avail)),
        RESOLVECONFLICT (UPDATEROWEXISTS, (delta_resolution_method, USEDELTA, COLS (seats_avail)),
        (DEFAULT, OVERWRITE));

U.S. Database (Delivery)

        MAP euro.flight_inv, TARGET us.flight_inv,
        COMPARECOLS (ON UPDATE KEYINCLUDING (seats_avail)),
        RESOLVECONFLICT (UPDATEROWEXISTS, (delta_resolution_method, USEDELTA, COLS (seats_avail)),
        (DEFAULT, OVERWRITE));
```

The parameter COMPARECOLS instructs the Delivery process to compare the before image values of the SEATS_AVAIL column with the current SEATS_AVAIL column value when an update occurs to the FLIGHT_INV table. The KEYINCLUDING parameter instructs the Delivery process to use the primary key FLIGHT for uniqueness when comparing rows. If the SEATS_AVAIL column values match, the update transaction will complete normally, as no conflict has occurred. If the SEATS_AVAIL column values don't match, and in this example the SEATS_AVAIL values are different, a conflict has occurred and the RESOLVECONFLICT parameter would fire. The RESOLVECONFLICT parameter, in this example, instructs the Delivery process to update the SEATS_AVAIL column by subtracting the before image value of the column from the after image value of the column and then adding that value to the current value of the column.

Database	After Value		Before Value		Current Value		Final Value
European	6	-	10	+	7	=	3
U.S.	7	-	10	+	6	=	3

The COLS parameter instructs the Delivery process which columns to update on resolution and the USEDELTA parameter instructs the Delivery process to calculate the final value. In this example, Joe ordered 3 tickets, so the current value of the SEATS_AVAIL column in the European Database would be 7, while Kevin ordered 4 tickets, so the current value of the SEATS_AVAIL column would be 6. When Joe's order is applied to the U.S. Database, the original inventory of 10 tickets would be subtracted from his after value of 7 and then the current U.S. Database value of 6 would be added for a final value of 3. When Kevin's order is applied to the European Database, the original inventory of 10 tickets would be subtracted from his after value of 6 and then the current European Database value of 7 would be added for a final value of 3. The quantitative resolution routines produced the same result of 3 for SEATS_AVAIL in both databases and the rows are in sync.

Confirm Conflict Detection Resolution Succeeded

Issue the following command in GGSCI on the European and U.S. Database servers to display the current CDR statistics for the Delivery (Replicat) process. This is a quick method to determine if conflicts are being resolved successfully by the Delivery process.

```
GGSCI> STATS REPLICAT <group>, REPORTCDR

Total inserts                          0.00
Total updates                          1.00
Total deletes                          0.00
Total discards                         0.00
Total operations                       1.00
Total CDR conflicts                    1.00
CDR resolutions succeeded              1.00
CDR UPDATEROWEXISTS conflicts          1.00
```

Conflict Notification and Tracking

When configuring an active-active replication environment for conflict detection and resolution, it is a best practice to track conflicts through an exceptions table. The exceptions table 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. Exception tables assist in troubleshooting, auditing, and notification purposes in complex environments.

Example #4: Configure an Exceptions Table for Auditing and Notification

In the seat reservation example, timestamp was used to resolve the conflict of Joe and Kevin making the seat reservation at approximately the same time. The conflict was handled correctly in the database, but Joe and Kevin both believe they have reserved seat 2A. Joe made his seat reservation for 2A one second before Kevin and overwrote Kevin's seat reservation in the U.S. Database. As far as Kevin knows, he has reserved seat 2A. How would the airline know to notify Kevin that his seat reservation was not successful and that he needed to reserve a new seat? The airline would use the exceptions table to identify Kevin's rejected seat reservations and notify him to reserve a new seat.

How would Oracle GoldenGate 11*g* Release 2 be configured to write conflicts to an exceptions table for seat reservations and notify Kevin to reserve a new seat?

Create Exceptions Table

An exceptions table is recommended to be created in both the European and U.S. Databases and a best practice is to write as much information as possible to the exceptions table about the conflict. The exceptions table should include all columns in the parent table as well as additional identifying information about the transaction. Further, it is recommended to create the exceptions table without any primary key or unique indexes to avoid unique constraint violations. For example the SEAT_RESV exceptions table would be named SEAT_RESV_EXCEPTIONS and might look like the following.

Name	Null?	Type
TOTAL_CDR_CONFLICTS		NUMBER
CDR_FAILED		NUMBER
CDR_SUCCESSFUL		NUMBER
RESOLUTION_DATE		DATE
OPTYPE		VARCHAR2(10)
DBERRNUM		NUMBER
DBERRMSG		VARCHAR2(25)
TABLE_NAME		VARCHAR2(20)
PASSENGER_AFTER		VARCHAR2(10)
LAST_UPDT_AFTER		TIMESTAMP(6)
PASSENGER_BEFORE		VARCHAR2(10)
LAST_UPDT_BEFORE		TIMESTAMP(6)
PASSENGER_CURRENT		VARCHAR2(10)
LAST_UPDT_CURRENT		TIMESTAMP(6)
ID		NUMBER
SEAT		VARCHAR2(3)
FLIGHT		NUMBER

Configure Oracle GoldenGate Delivery

Once the exceptions table has been created in the European and U.S. Databases, the Oracle GoldenGate Delivery processes need to be configured to write conflict and resolution exceptions to the SEAT_RESV_EXCEPTIONS table.

The following lines would be added to Oracle GoldenGate Delivery parameter file for the European and U.S. Databases. The map statement for SEAT_RESV_EXCEPTIONS needs to appear immediately following the SEAT_RESV map statement in the parameter file.

European Database (Delivery)

```
MAP us.seat_resv, TARGET euro.seat_resv,
COMPARECOLS (ON UPDATE KEYINCLUDING (last_update)),
RESOLVECONFLICT (UPDATEROWEXISTS, (min_resolution_method, USEMIN (last_update), COLS (passenger,
last_update)), (DEFAULT, OVERWRITE));

MAP us.seat_resv, TARGET euro.seat_resv_exceptions,EXCEPTIONSONLY, INSERTALLRECORDS,
SQLEXEC (id qry,
QUERY "select passenger,last_updt from euro.seat_resv where id = :p1",
PARAMS (p1 = id)),
 COLMAP ( USEDEFAULTS,
 total_cdr_conflicts = @GETENV(“DELTASTATS”, “TABLE”, “euro.seat_resv”, “CDR_CONFLICTS”),
 cdr_failed = @GETENV(“DELTASTATS”, “TABLE”, “euro.seat_resv”, “CDR_RESOLUTIONS_FAILED”),
 cdr_successful = @GETENV(“DELTASTATS”, “TABLE”, “euro.seat_resv”, “CDR_RESOLUTIONS_SUCCEEDED”),
 resolution_date = @DATENOW(),
 optype = @GETENV("LASTERR", "OPTYPE"),
 dberrnum = @GETENV("LASTERR", "DBERRNUM"),
 dberrmsg = @GETENV("LASTERR", "DBERRMSG"),
 table_name = @GETENV("GGHEADER", "TABLENAME"),
 passenger_after = passenger,
 last_updt_after = last_updt,
 passenger_before = before.passenger,
 last_updt_before = before.last_updt,
 passenger_current = qry.passenger,
 last_updt_current = qry.last_updt,
);
```

U.S. Database (Delivery)

```
MAP euro.seat_resv, TARGET us.seat_resv,
COMPARECOLS (ON UPDATE KEYINCLUDING (last_update)),
RESOLVECONFLICT (UPDATEROWEXISTS, (min_resolution_method, USEMIN (last_update), COLS (passenger,
last_update)), (DEFAULT, OVERWRITE));

MAP euro.seat_resv, TARGET us.seat_resv_exceptions,EXCEPTIONSONLY, INSERTALLRECORDS,
SQLEXEC (id qry,
QUERY "select passenger,last_updt from us.seat_resv where id = :p1",
PARAMS (p1 = id)),
 COLMAP ( USEDEFAULTS,
 total_cdr_conflicts = @GETENV(“DELTASTATS”, “TABLE”, “us.seat_resv”, “CDR_CONFLICTS”),
 cdr_failed = @GETENV(“DELTASTATS”, “TABLE”, “us.seat_resv”, “CDR_RESOLUTIONS_FAILED”),
 cdr_successful = @GETENV(“DELTASTATS”, “TABLE”, “us..seat_resv”, “CDR_RESOLUTIONS_SUCCEEDED”),
 resolution_date = @DATENOW(),
 optype = @GETENV("LASTERR", "OPTYPE"),
 dberrnum = @GETENV("LASTERR", "DBERRNUM"),
 dberrmsg = @GETENV("LASTERR", "DBERRMSG"),
 table_name = @GETENV("GGHEADER", "TABLENAME"),
 passenger_after = passenger,
 last_updt_after = last_updt,
 passenger_before = before.passenger,
 last_updt_before = before.last_updt,
 passenger_current = qry.passenger,
 last_updt_current = qry.last_updt,
);
```

The parameter EXCEPTIONSONLY instructs the Delivery process to handle errors in the previous SEAT_RESV table map statement if they occur. The parameter INSERTALLRECORDS instructs the Delivery process to change all transactions to insert statements when writing to the SEAT_RESV_EXCEPTIONS table. COLMAP explicitly maps source columns to target columns and USEDEFAULTS automatically maps columns with the same name. The TOTAL_CDR_CONFLICTS column stores the total number of conflicts that have occurred with the DELTASTATS parameter keeping a running count from the previous time statistics were collected. The CDR_FAILED column stores the number of conflicts that failed to resolve with the DELTASTATS parameter keeping a running count from the previous time statistics were collected. The CDR_SUCCESSFUL column stores the number of conflicts that were resolved successfully with the DELTASTATS parameter keeping a running count from the previous time statistics were collected. The RESOLUTION_DATE column will store the date/time the conflict occurred. The OPTYPE column will store the operation type of insert, update, or delete. The DBERRNUM column will store the database error code. The DBERRMSG column will store the database error message. The TABLE_NAME column will store the name of the table that had the conflict. The PASSENGER_AFTER column will store the after image of the PASSENGER value from the trail file. The LAST_UPDT_AFTER column will store the after image of the LAST_UPDT value from the trail file. The PASSENGER_BEFORE column will store the before image of the PASSENGER value from the trail file. The LAST_UPDT_BEFORE column will store the before image of the LAST_UPDT value from the trail file. The PASSENGER_CURRENT column will store the PASSENGER value that is currently in the SEAT_RESV table. The LAST_UPDT_CURRENT column will store the LAST_UPDT value that is currently in the SEAT_RESV table.

Notification

In this example, Kevin's seat reservation was overwritten by Joe's seat reservation in the U.S Database and Joe's transaction would be inserted into the SEAT_RESV_EXCEPTIONS table. The SEAT_RESV_EXCEPTIONS table would contain the following row in the U.S. Database.

U.S. Database (SEAT_RESV_EXCEPTIONS table)

RESOLUTION_DATE	OPTYPE	DBERRNUM	DBERRMSG	TABLE_NAME
15-OCT-12	SQL COMPUP	1403		EURO.SEAT_RESV

PASSENGER_AFTER	LAST_UPDT_AFTER	PASSENGER_BEFORE	LAST_UPDT_BEFORE
JOE	15-OCT-12 10:30:00 AM		01-OCT-12 8:15:45 AM

PASSENGER_CURRENT	LAST_UPDT_CURRENT	ID	SEAT	FLIGHT
JOE	15-OCT-12 10:30:00 AM	1	2A	123

TOTAL_CDR_CONFLICTS	CDR_FAILED	CDR_SUCCESSFUL
1	0	1

Kevin's seat reservation transaction would be ignored and inserted into the SEAT_RESV_EXCEPTIONS table in the European Database, as Joe's seat reservation is older than Kevin's. The SEAT_RESV_EXCEPTIONS table would contain the following row in the European Database.

European Database (SEAT_RESV_EXCEPTIONS table)

RESOLUTION_DATE	OPTYPE	DBERRNUM	DBERRMSG	TABLE_NAME
15-OCT-12	SQL COMPUP	1403		US.SEAT_RESV

PASSENGER_AFTER	LAST_UPDT_AFTER	PASSENGER_BEFORE	LAST_UPDT_BEFORE
KEVIN	15-OCT-12 10:30:01 AM		01-OCT-12 8:15:45 AM

PASSENGER_CURRENT	LAST_UPDT_CURRENT	ID	SEAT	FLIGHT
JOE	15-OCT-12 10:30:00 AM	1	2A	123

TOTAL_CDR_CONFLICTS	CDR_FAILED	CDR_SUCCESSFUL
1	0	1

The information stored in the SEAT_RESV_EXCEPTIONS tables would be used to identify customers that need to re-reserve their seats. A best practice would be to have a batch process periodically check the exception tables where the after image is not equal to the current image. In this example, the PASSENGER_AFTER value is "Kevin" and the PASSENGER_CURRENT value is "Joe" in the European Database's SEAT_RESV_EXCEPTIONS table. From this information, the airline identifies Kevin's seat reservation was overwritten by Joe's and notifies Kevin that he needs to re-reserve his seat.

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.

Oracle GoldenGate Data Definition Language Replication

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. However, special considerations need to be taken into account if DDL replication is to be used in an active-active replication configuration.

DDL operations that alter or change the existing table definitions should not be issued in a production active-active replication environment. These statements would 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 be required to change. For

example, if a column is dropped in one database and at the same time someone inserts a value into that column on a different database, this would cause problems that could not be handled by conflict resolution. In order to manage these types of DDL operations in an active-active replication environment, one database would be quiesced until only one database was active and the configuration was now running in an active-passive replication mode. DDL operations could then be executed and parameter files updated on the passive database without interfering with in-flight transactions. After the passive database has been updated, the roles would be reversed for the remaining database to be updated as well. Once both databases have been updated, including GoldenGate parameter files, the environment configuration would return to active-active replication mode.

DDL operations that don't modify existing table definitions can safely be executed in an active-active replication environment, but should only be executed against one database. These statements would include DDL operations that create new tables, create new tablespaces, or create users.

Conclusion

Active-active database replication 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 replication 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 replication 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 replication configurations.

Best Practices for Conflict Detection and Resolution in Active-Active Replication Environments Using Oracle GoldenGate
Updated December 2012

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

0109