Oracle Rdb™ for OpenVMS

Guide to Distributed Transactions

Release 7.2 for OpenVMS Industry Standard 64 for Integrity Servers and OpenVMS Alpha systems.

January 2006

ORACLE®

Oracle Rdb for OpenVMS Guide to Distributed Transactions

Release 7.2 for OpenVMS Industry Standard 64 for Integrity Servers and OpenVMS Alpha

Contents

Index

Examples

Figures

Tables

Send Us Your Comments

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this document. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most?

If you find any errors or have any other suggestions for improvement, please indicate the document title, chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail:nedc-doc_us@oracle.com
- FAX — 603-897-3825 Attn: Oracle Rdb Documentation
- Postal service:
  Oracle Corporation
  Oracle Rdb Documentation
  One Oracle Drive
  Nashua, NH 03062-2804
  USA

If you would like a reply, please give your name, address, telephone number, and (optionally) electronic mail address.

If you have problems with the software, please contact your local Oracle Support Services.

Preface

Oracle Rdb software is a general-purpose database management system based on the relational model. Oracle Rdb, using the two-phase commit protocol, coordinates the activity of transaction components across multiple databases to ensure the integrity of transactions and to maintain logical consistency among distributed transactions.

This manual describes the two-phase commit protocol, how to start and complete distributed transactions using SQL and how to recover databases from unresolved transactions using Oracle RMU commands.

Intended Audience

This manual is intended for experienced programmers who write and maintain database applications that access more than one database in a transaction or that attach to a database more than once in a transaction.

To get the most out of this manual, you should understand database concepts and terminology, particularly how to start and end transactions. In addition, you should be experienced in programming in a high-level language and be familiar with the SQL interface.

How This Manual Is Organized

This book contains the following chapters and appendixes:

Chapter 1	Introduces the two-phase commit protocol and distributed transactions, and explains when to use distributed transactions.
Chapter 2	Explains how distributed transactions work.
Chapter 3	Describes considerations in designing databases and applications for use with distributed transactions.

Chapter 4	Describes how to use distributed transactions with the SQL module language and precompiled SQL.
Chapter 5	Describes how to complete unresolved transactions using Oracle RMU commands.
Appendix A	Summarizes problems that are unique to distributed transactions and suggests solutions to those problems.
Appendix B	Describes how to use the DECdtm XA Gateway to allow an Oracle Rdb database to participate in a distributed transaction managed by XA.

Related Manuals

For more information on Oracle Rdb, see the other manuals in this documentation set, especially the following:

- *Oracle Rdb Guide to SQL Programming*
- *Oracle Rdb Guide to Database Design and Definition*
- *Oracle Rdb Guide to Database Maintenance*
- *Oracle Rdb SQL Reference Manual*
- *Oracle RMU Reference Manual*

Conventions

OpenVMS I64 refers to OpenVMS Industry Standard 64 for Integrity Servers.

Oracle Rdb refers to Oracle Rdb for OpenVMS Alpha and Oracle Rdb for OpenVMS I64 software.

The OpenVMS I64 operating system and the OpenVMS Alpha operating system are referred to as OpenVMS.

Oracle Rdb refers to Oracle Rdb for OpenVMS Alpha software.

The SQL interface to Oracle Rdb is referred to as SQL. This interface is the Oracle Rdb implementation of the SQL standard ANSI X3.135-1992, ISO 9075:1992, commonly referred to as the ANSI/ISO SQL standard or SQL92.

XA refers to the XA Distributed Transaction Model as specified in the XA Specification published by The Open Group.

In examples, an implied carriage return occurs at the end of each line, unless otherwise noted. You must press the Return key at the end of a line of input.

The following conventions are also used in this manual:

. . .	Vertical ellipsis points in an example mean that information not directly related to the example has been omitted.
. . .	Horizontal ellipsis points in statements or commands mean that parts of the statement or command not directly related to the example have been omitted.
e, f, t	Index entries in the printed manual may have a lowercase e, f, or t following the page number; the e, f, or t is a reference to the example, figure, or table, respectively, on that page.
< >	Angle brackets enclose user-supplied information.
$	The dollar sign represents the DIGITAL Command Language prompt. This symbol indicates that the DCL interpreter is ready for input.

1
Introduction to Distributed Transactions

The task of maintaining database integrity and consistency can be difficult if an application uses more than one database or attachment to a database during a single transaction. To make this task easier, Oracle Rdb uses the two-phase commit protocol provided by DECdtm services. DECdtm services are part of the OpenVMS operating system. (This manual often refers to DECdtm services as DECdtm.)

The **two-phase commit protocol** coordinates the activity of participants in a transaction to ensure that every required operation is completed before a transaction is made permanent, even if the transaction is a distributed transaction.

A **distributed transaction** groups more than one database or more than one database attachment together into one transaction, even if the databases are located on different nodes. If one operation in a transaction cannot be completed, none of the operations is completed. This "all or nothing" approach guarantees that databases involved in a distributed transaction remain logically consistent with one another.

The major benefit of using the two-phase commit protocol is that you can create applications that access several different databases without compromising the integrity or consistency of your data.

While Oracle Rdb does not provide direct support for the XA two-phase commit protocol, the DECdtm XA Gateway software allows an Oracle Rdb database to be integrated into an environment using XA transactions.

This chapter describes:

- When to use the two-phase commit protocol and distributed transactions
- What the two-phase commit protocol and distributed transactions are
- What other products you can use with Oracle Rdb in a distributed transaction

- How a distributed transaction terminates
- How to complete unresolved transactions

1.1 Deciding When to Use Distributed Transactions

Use the two-phase commit protocol when your application starts a distributed transaction. As stated previously, a **distributed transaction** is a transaction that groups more than one database or more than one database attachment together into one transaction. The following are distributed transactions:

- A transaction that attaches more than once to an Oracle Rdb database
- A transaction that attaches to more than one Oracle Rdb database
- A transaction that attaches to more than one data management system or resource manager that supports DECdtm services

Often, a transaction attaches to more than one database, and the databases are separated geographically. For example, a bank in Switzerland transfers $1,000,000 from a local account to a bank account in the United States. First, the application debits $1,000,000 from the Swiss bank, then it credits $1,000,000 to the U.S. account. Because the transaction involves two different databases, without the two-phase commit protocol this transaction must be committed sequentially. That is, the debit is committed, then the credit is committed. However, if the network connection fails after the application debits the funds from the Swiss account but before it credits the funds to the U.S. account, the bank could lose $1,000,000. With the two-phase commit protocol, if the network connection fails before it credits the funds to the U.S. account, the entire transaction is rolled back.

Without the two-phase commit protocol, keeping remote databases consistent with one another can be difficult. The two-phase commit protocol coordinates the transaction components and ensures consistency among the databases. With the two-phase commit protocol, if the transaction does not complete successfully, the changes to both databases are rolled back.

In another example, an engineering company has a central Oracle CODASYL DBMS database, and also Oracle Rdb databases for each division—the East division and the West division. The central database contains information about employees from both divisions. The Oracle Rdb database for each division contains more detailed information about its employees, but only employees in that division. When an employee joins the company, an application adds the employee to the central database and to either the East or West database. The application performs both operations in one distributed transaction to ensure the information in the central Oracle CODASYL DBMS

database remains logically consistent with the information in each division's Oracle Rdb database.

1.2 Using Distributed Transactions with Other Products

In addition to using the two-phase commit protocol with more than one attachment to an Oracle Rdb database, you can use the two-phase commit protocol when you have information in more than one type of data management system. The two-phase commit protocol does not limit you to using only Oracle Rdb databases. An application can start a distributed transaction using an Oracle Rdb database and any resource that implements the two-phase commit protocol using DECdtm services. Products that support the two-phase commit protocol include the following:

- Oracle CODASYL DBMS

  For information about using Oracle CODASYL DBMS with distributed transactions, see the Oracle CODASYL DBMS documentation.

- RMS Journaling

  For information about using RMS Journaling with distributed transactions, see the RMS Journaling documentation.

Although products other than database products support DECdtm services' implementation of the two-phase commit protocol, for simplicity, this manual usually describes distributed transactions as operating on one or more databases.

Note

To use the two-phase commit protocol, the DECdtm software must be running on all nodes involved in the transaction.

Additionally, the DECdtm XA Gateway provides a method for Oracle Rdb databases to participate in distributed transactions being managed using the XA two-phase commit protocol. In this mode, Oracle Rdb uses the DECdtm two-phase protocol as usual and the DECdtm XA Gateway provides the interface to convert between the XA and DECdtm two-phase commit protocols. The DECdtm XA Gateway can be used Oracle Rdb release 7.1 and later. The DECdtm XA Gateway is available in OpenVMS Version 7.3-1 and later.

Note

In order to use the DECdtm XA Gateway, an XA Transaction Manager must be running on each node where an Oracle Rdb database is to participate in the transaction.

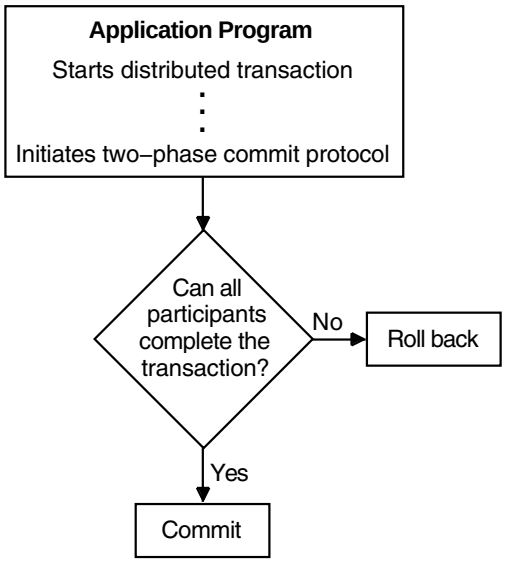
See Appendix B for information on using the DECdtm XA Gateway to allow Oracle Rdb databases to participate in an XA-managed distributed transaction. For information on the DECdtm XA Gateway, see the OpenVMS documentation set.

1.3 Understanding the Two-Phase Commit Protocol and Distributed Transactions

Oracle Rdb uses the two-phase commit protocol to coordinate a distributed transaction and to ensure consistency among databases involved in the transaction. The two-phase commit protocol guarantees that either all changes in the distributed transaction are made, or else none of them is made.

Figure 1–1 illustrates the logic of the two-phase commit protocol.

Figure 1–1 Executing the Two-Phase Commit Protocol



ZK–1772A–GE

The two-phase commit protocol operates in a distributed transaction environment that includes the following participants:

- Resource manager

  The **resource manager** is a database management system, such as Oracle Rdb. The resource manager is responsible for maintaining and recovering its own resources. From the perspective of the application, the resource manager is a single attachment to an Oracle Rdb database. In addition, other products that support the two-phase commit protocol as implemented by DECdtm services can be resource managers.

- Transaction manager

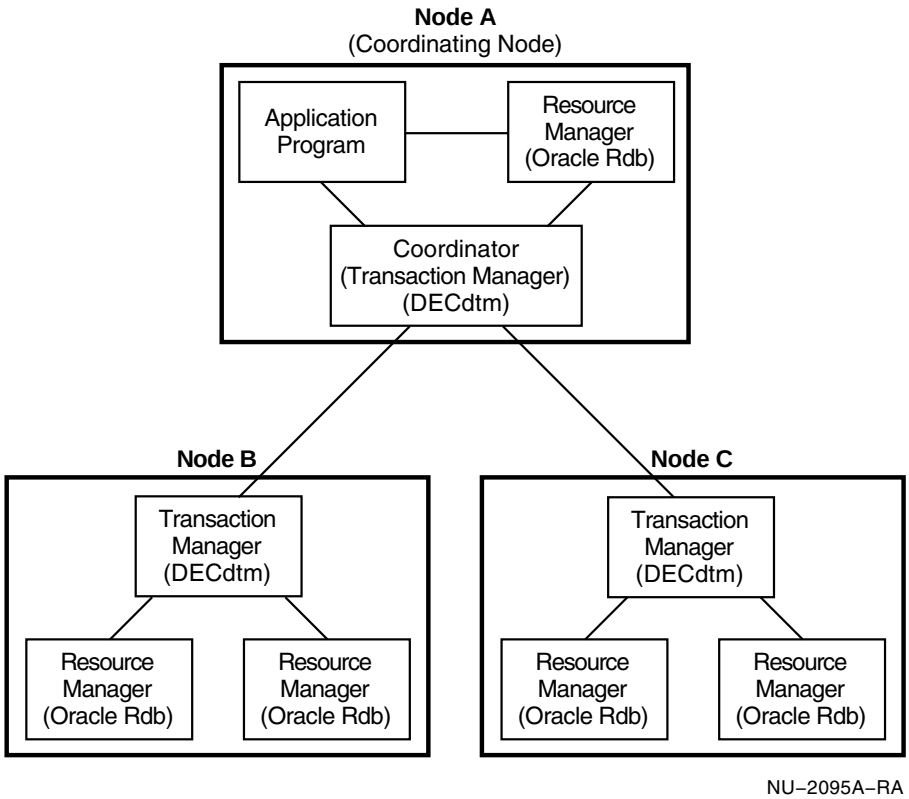
  The **transaction manager** coordinates the actions of the resource managers that are located on the same node (local resource managers) as the transaction manager. DECdtm provides the transaction manager. (A transaction manager may also act as the coordinator under specific circumstances.)

- Coordinator

  The **coordinator** is the transaction manager on the node where the application started the transaction. The coordinator orchestrates the distributed transaction by communicating with transaction managers on other nodes (remote transaction managers) and with resource managers on the same node (local resource managers). DECdtm provides the transaction manager that acts as coordinator.

Figure 1–2 shows the participants in a distributed transaction that involves databases on three nodes. Oracle Rdb and DECdtm software are installed on all three nodes.

Figure 1–2 Participants in a Distributed Transaction



When your application starts a distributed transaction, the coordinator checks that DECdtm software is running on all the nodes participating in the transaction. (If the DECdtm software is not running, Oracle Rdb returns an error and does not start the distributed transaction.) Then, the coordinator generates a distributed transaction identifier and associates the identifier with all the participants in that particular transaction. Section 2.1 describes this process in more detail.

When the application is ready to commit all the changes to the databases involved in the distributed transaction, all the nodes in the transaction must execute the two phases of the two-phase commit protocol, the *prepare* phase and the *commit* phase.

During the **prepare** phase, the coordinator asks each resource manager participating in the transaction whether or not it is prepared to commit the transaction. If the coordinator receives yes responses (or votes) from *all* the resource managers, the coordinator instructs the participants in the transaction to enter the commit phase.

During the **commit** phase, the coordinator instructs the resource managers to make permanent all changes to the database, that is, to commit the changes. Then, the resource managers commit the changes and the transaction is completed.

The next section describes the circumstances under which distributed transactions are committed or rolled back.

1.3.1 Terminating Distributed Transactions

The two-phase commit protocol ensures that if one database involved in a distributed transaction cannot commit the changes, none of the databases involved in the transaction commits the changes.

Table 1–1 summarizes the circumstances under which distributed transactions are committed or rolled back.

Table 1–1 Committing or Rolling Back Distributed Transactions

When This Happens:	This Is the Result:
Application instructs the transaction to roll back.	Transaction rolls back.
Process or image failure occurs before all participants vote.	Transaction rolls back.

(continued on next page)

Table 1–1 (Cont.) Committing or Rolling Back Distributed Transactions

When This Happens:	This Is the Result:
Any participant votes no.	Transaction rolls back.
All participants vote yes.	Transaction commits.
Process or image failure occurs after all participants have voted and the coordinator has received all yes votes.	Transaction commits.
Process or image failure occurs after all participants have voted and the coordinator has received one or more no votes.	Transaction rolls back.

The two-phase commit protocol coordinates the actions of the individual transaction participants so that either the entire transaction is committed or the entire transaction is rolled back, even if there is a subsequent process or image failure. See Chapter 2 for more information about completing transactions and handling unplanned terminations of transactions.

1.3.2 Completing Unresolved Transactions

On rare occasions, an unexpected failure occurs that causes the coordinator to be unreachable for an extended period of time. For example, the network connection may be lost or the node itself may suffer some irreparable damage. If these situations occur after the participants vote, but before transactions are completed on all nodes, the distributed transaction is **unresolved**. If you cannot wait for the connection to the coordinator to be restored or if the damage cannot be repaired, you can force the distributed transaction to commit or roll back.

To recover from an unresolved distributed transaction, you use commands provided by the resource manager. For Oracle Rdb databases, you use commands provided by Oracle RMU, the Oracle Rdb management utility. Section 2.4 describes unresolved transactions and Chapter 5 explains in detail how to recover databases from unresolved transactions.

This chapter provided an introduction to the two-phase commit protocol. It described the two-phase commit protocol, when to use it, and how it works. Chapter 2 describes in more detail how the two-phase commit protocol and distributed transactions work.

2
How Distributed Transactions Work

As Section 1.1 explains, the two-phase commit protocol applies only to distributed transactions. Distributed transactions group more than one database or more than one database attachment together into one transaction.

This chapter discusses the following topics:

- Starting distributed transactions
- Completing successful distributed transactions
- Understanding how the database recovery (DBR) process works with distributed transactions
- Completing unresolved distributed transactions

Note

You cannot use the two-phase commit protocol with batch-update transactions. For the two-phase commit protocol to work, you must be able to roll back transactions. However, because batch-update transactions do not write to recovery-unit journal files, these transactions cannot be rolled back.

2.1 Starting Distributed Transactions

To execute a distributed transaction, your application must invoke, either implicitly or explicitly, system services that DECdtm provides. Invoking the system services **implicitly** means that Oracle Rdb calls them on behalf of your application; invoking the system services **explicitly** means that the application directly calls the DECdtm system services.

With Oracle Rdb, you use the following DECdtm services:

- SYS$START_TRANS to start a distributed transaction

- SYS$END_TRANS to end a distributed transaction

  The DECdtm service SYS$END_TRANS commits a distributed transaction if all the participants vote yes, or rolls back the transaction if any of the participants votes no.

- SYS$ABORT_TRANS to roll back a distributed transaction

If a transaction involves only Oracle Rdb databases, the application can call the DECdtm system services either explicitly or implicitly. If the transaction involves Oracle Rdb databases and other resource managers (such as Oracle CODASYL DBMS or RMS Journaling) that use DECdtm services' implementation of the two-phase commit protocol, your application must explicitly call the DECdtm system services.

Chapter 4 describes in more detail when your application must call the DECdtm system services explicitly and when it can call them implicitly.

As Section 1.3 explains, a distributed transaction environment includes the following participants:

- Resource manager

  A database management system such as Oracle Rdb. From the perspective of the application, the resource manager is a single attachment to an Oracle Rdb database. In addition, other database products that support the two-phase commit protocol can be resource managers.

- Transaction manager

  DECdtm software on each node involved in the transaction. A transaction manager may act as the coordinator under specific circumstances.

- Coordinator

  The transaction manager on the node on which the application started.

In its role as coordinator, the DECdtm transaction manager maintains a list of resource managers on the local node and transaction managers on remote nodes that participate in the execution of a transaction. The coordinator generates a distributed transaction identifier (TID), which it uses to keep track of the transaction and of the local resource managers and remote transaction managers that are involved in that transaction.

Each participating transaction manager maintains a list that includes the distributed TID that identifies the transaction and the local resource managers that are involved in the distributed transaction. The transaction manager writes transaction information to a transaction log file, which contains a record of transaction states. By having access to a transaction log, a transaction

manager can resume the execution of the distributed transaction after a node recovers from a system failure.

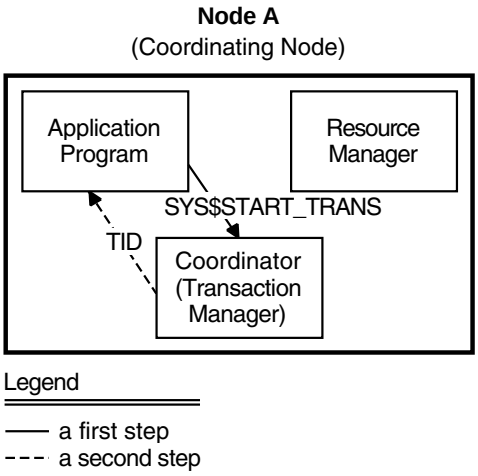
Each resource manager logs enough information to its recovery-unit journal (.ruj) file and after-image journal (.aij) file to allow it to undo or redo operations it performed on behalf of a transaction. The resource manager communicates with the transaction manager by using the DECdtm system services.

The following steps describe how to start a distributed transaction and what actions the coordinator, the transaction managers, and the resource managers take:

1. The application starts a distributed transaction by calling, either explicitly or implicitly, the DECdtm system service SYS$START_TRANS.
2. The coordinator, DECdtm, generates a unique distributed TID so that it can keep track of the transaction and the participants in the transaction. The coordinator returns the distributed TID to the application if the application called SYS$START_TRANS explicitly, or to the resource manager if the resource manager called SYS$START_TRANS on behalf of the application.

   Figure 2–1 shows the application program explicitly calling SYS$START_ TRANS and the coordinator returning the distributed TID to the application.

Figure 2–1 Starting a Distributed Transaction



NU–2096A–RA

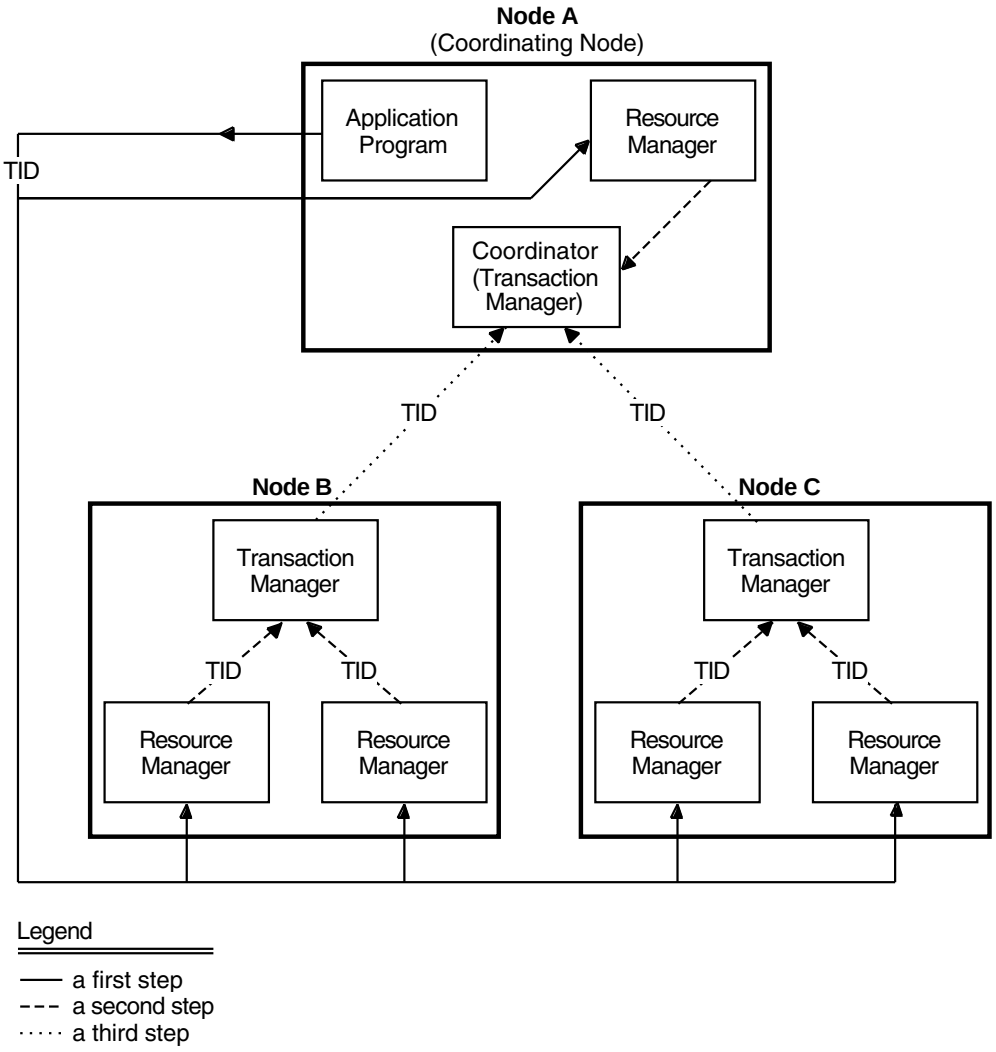
The coordinator and transaction managers use the distributed TID to keep track of the transaction and to identify all the participants in the transaction. The participants include the resource managers and transaction managers.

3. The application passes the distributed TID to the resource managers and each resource manager joins the distributed transaction by notifying the transaction manager on its node that it is part of the distributed transaction distinguished by a particular distributed TID.

   Note that these actions are invisible to the application; the application only needs to invoke SYS$START_TRANS explicitly or implicitly.

   Figure 2–2 shows how the transaction participants pass the distributed TID to inform each other that they are part of that particular distributed transaction.

Figure 2–2 Passing the Distributed Transaction Identifier (TID)



4. If the SYS$START_TRANS system service does not return an error, the application starts the transaction.

Note

If DECdtm software is not running on all the nodes involved in a distributed transaction, Oracle Rdb returns an error and does not start the distributed transaction.

Section 2.2 describes the prepare and commit phases of the two-phase commit protocol and how distributed transactions are committed or rolled back.

2.2 Completing Successful Distributed Transactions

This section describes the prepare and commit phases of the two-phase commit protocol and how a distributed transaction commits or rolls back all changes to the databases.

When an application is ready to end a distributed transaction, it can either roll back or commit the changes made during that transaction.

To roll back the changes, the application can use the SYS$ABORT_TRANS system service or the ROLLBACK statement.

- If the application starts the transaction by explicitly calling the SYS$START_TRANS system service, it must explicitly use the system service SYS$ABORT_TRANS to roll back the changes.
- If the application starts the transaction implicitly (that is, the resource manager calls the SYS$START_TRANS system service on behalf of the application), it must use the ROLLBACK statement to roll back the changes. In this case, the resource manager calls SYS$ABORT_TRANS on behalf of the application.

When the application uses either of these methods, the coordinator simply instructs the participants to roll back the transaction. The coordinator does not use the two-phase commit protocol.

To commit the changes, the application can use the SYS$END_TRANS system service or the COMMIT statement.

- If the application starts the transaction by explicitly calling the SYS$START_TRANS system service, it must explicitly use the SYS$END_TRANS system service to commit the changes.
- If the application starts the transaction implicitly (that is, the resource manager calls the SYS$START_TRANS system service on behalf of the application), it must use the COMMIT statement to commit the changes. In this case, the resource manager calls SYS$END_TRANS on behalf of the application.

When the application calls the SYS$END_TRANS system service, either implicitly or explicitly, the coordinator instructs the participants in the distributed transaction to enter the prepare phase.

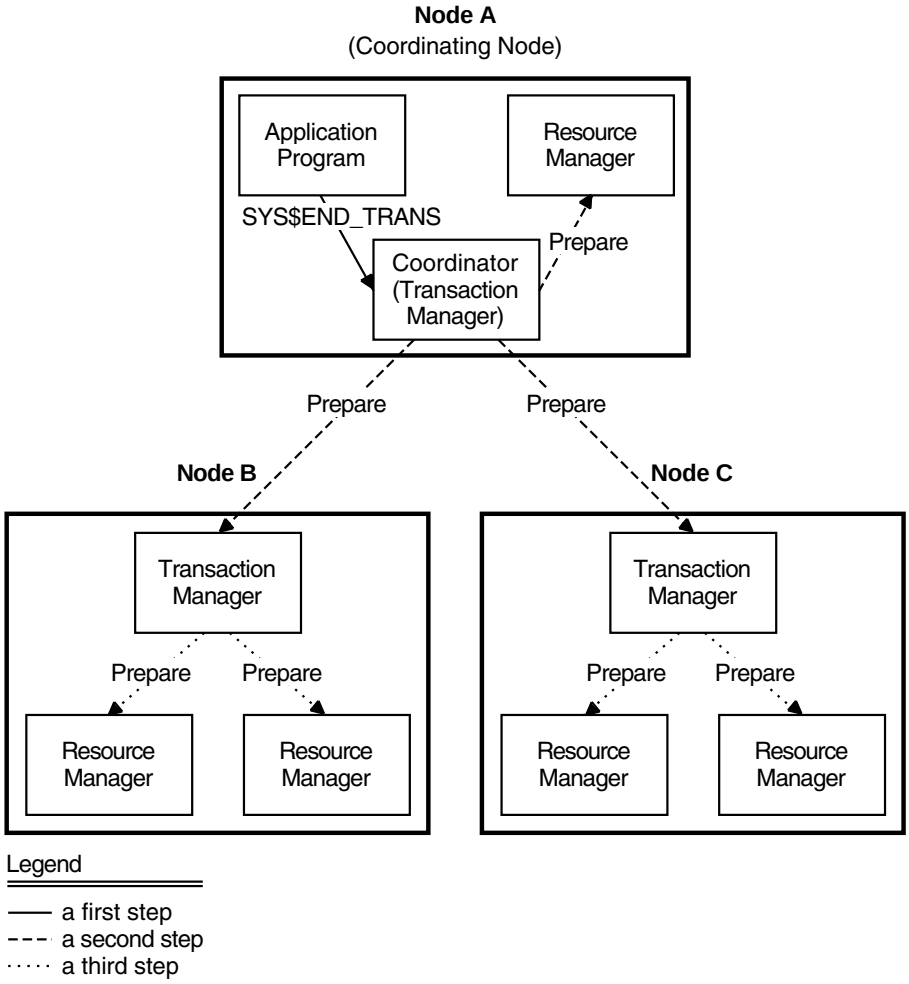
2.2.1 Executing the Prepare Phase

The prepare phase is the first phase of the two-phase commit protocol. The following steps describe the prepare phase and what actions the coordinator, the transaction manager, and the resource manager take:

1. The coordinator, DECdtm, receives the SYS$END_TRANS call and instructs the local resource managers and remote transaction managers to enter the prepare phase of the transaction.
2. The remote transaction managers instruct their local resource managers to enter the prepare phase of the transaction.

   Figure 2–3 shows an application program initiating the process to end a distributed transaction by explicitly calling the SYS$END_TRANS system service and the resulting instructions from the coordinator and transaction managers to enter the prepare phase.

Figure 2–3 Initiating the Prepare Phase



ZK–1774A–GE

3. When the resource managers receive instructions to prepare to commit, they write enough information to disk to commit or roll back any changes.

   The resource manager (Oracle Rdb) writes the following:

   - The before-images (original values) of the rows that have been modified and a prepare record to the recovery-unit journal file (file type .ruj)

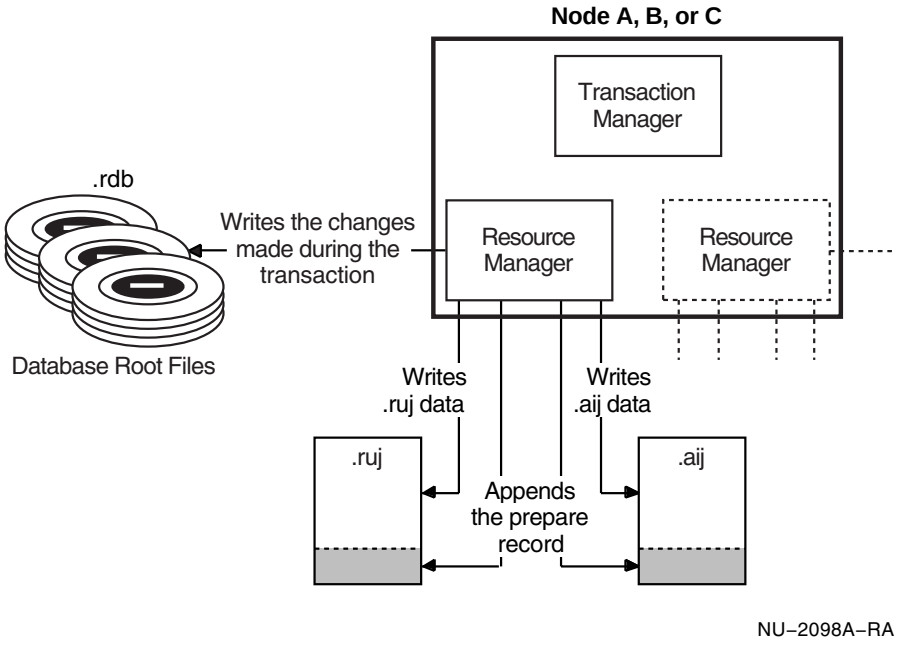
The resource manager writes the before-images to the .ruj file so that it can roll back the transaction if the coordinator instructs it to do so. In addition, the resource manager writes a prepare record to the .ruj file. If a process or image failure occurs, the prepare record indicates to the database recovery (DBR) process that the database is part of a distributed transaction and that the DBR process must consult with the coordinator to decide whether to commit or roll back the changes.

- All changes made during the transaction to the database root file (file type .rdb) and storage area files (file type .rda)
- All changes made during the transaction and a prepare record to the after-image journal file (file type .aij)

  The resource manager writes the changes to the .aij file so that it can commit the transaction if the coordinator instructs it to do so. In addition, the resource manager writes a prepare record to the .aij file. If a process or image failure occurs, the prepare record indicates that the database is part of a particular distributed transaction and makes it possible to complete an unresolved transaction when a database has become corrupted.

Figure 2–4 shows the information that the resource managers write to disk during the prepare phase.

Figure 2–4 Resource Managers Execute the Prepare Phase

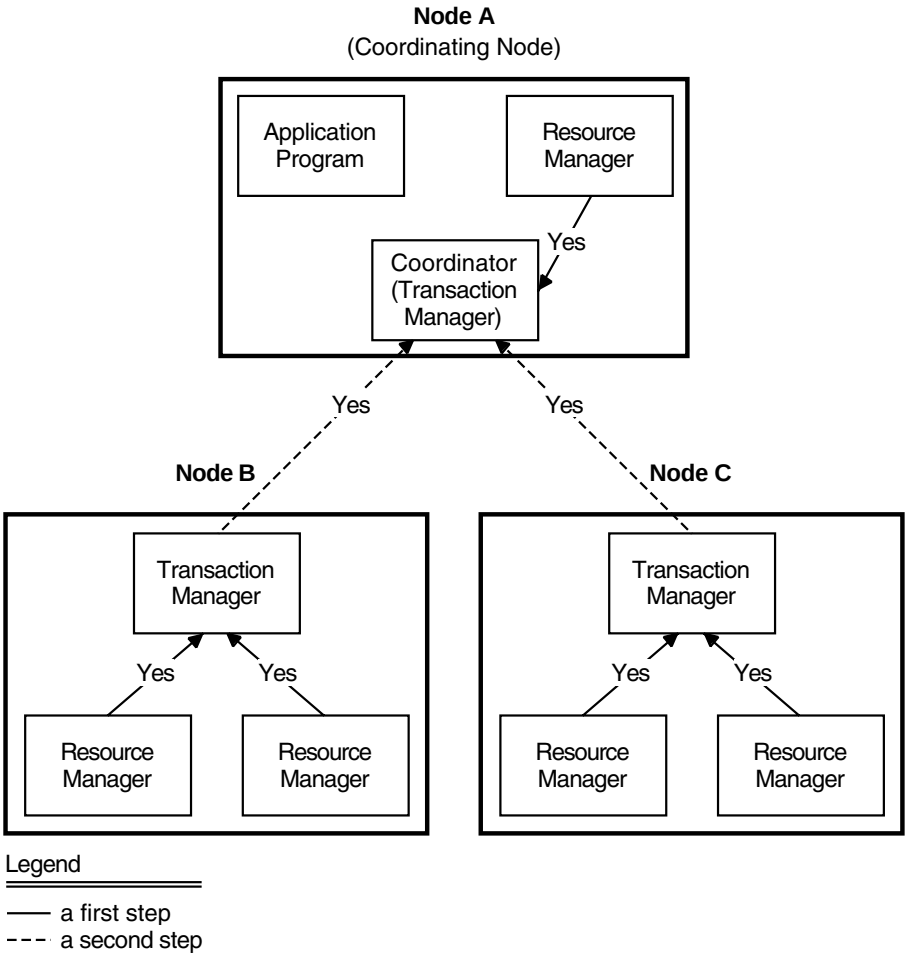


4. When the information is written to disk, the resource manager is *prepared* and returns a "Yes, prepared and willing to commit" vote to its local transaction manager.

5. Each transaction manager evaluates the votes from its local resource managers. If all the votes are yes, the transaction manager returns a yes vote to the coordinator. If any one of the votes is no, the transaction manager returns a no vote to the coordinator. (Remember that on the node that started the application, the coordinator is also the transaction manager. In this case, the coordinator, in its role as transaction manager, evaluates the votes from its local resource managers.)

   Figure 2–5 shows the participants voting that they are prepared and willing to commit.

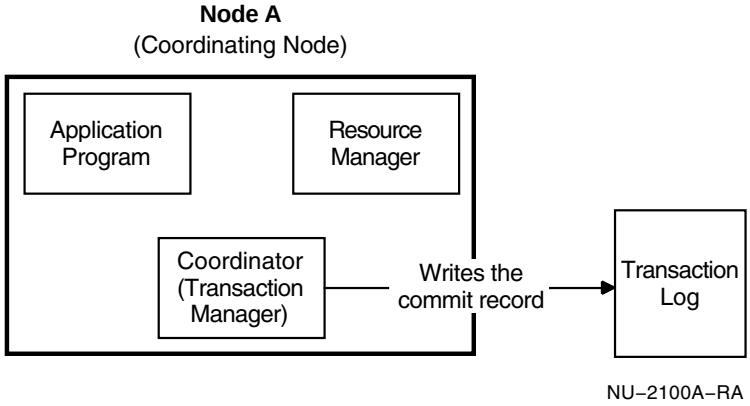
Figure 2–5 Voting by Participants in a Distributed Transaction



6. The coordinator evaluates the votes it receives from its local resource managers and the remote transaction managers.
   - If all the votes are yes, the coordinator writes the commit record to the transaction log. When the commit record is successfully written, the transaction enters the commit phase. At this point, the distributed transaction is guaranteed; no matter what happens, the transaction can commit the changes to the databases.

Figure 2–6 shows the coordinator writing the commit record to the transaction log.

Figure 2–6 Writing the Commit Record



- If any one of the votes is no, the coordinator instructs the resource managers to roll back the transaction. A participant might vote no if a process or image failure prevents it from committing the changes made during the transaction. If a process or image failure occurs before a resource manager votes, the coordinator assumes a no vote. Section 2.3 describes how the participants roll back the transaction if an unplanned process or image termination occurs.

2.2.2 Executing the Commit Phase

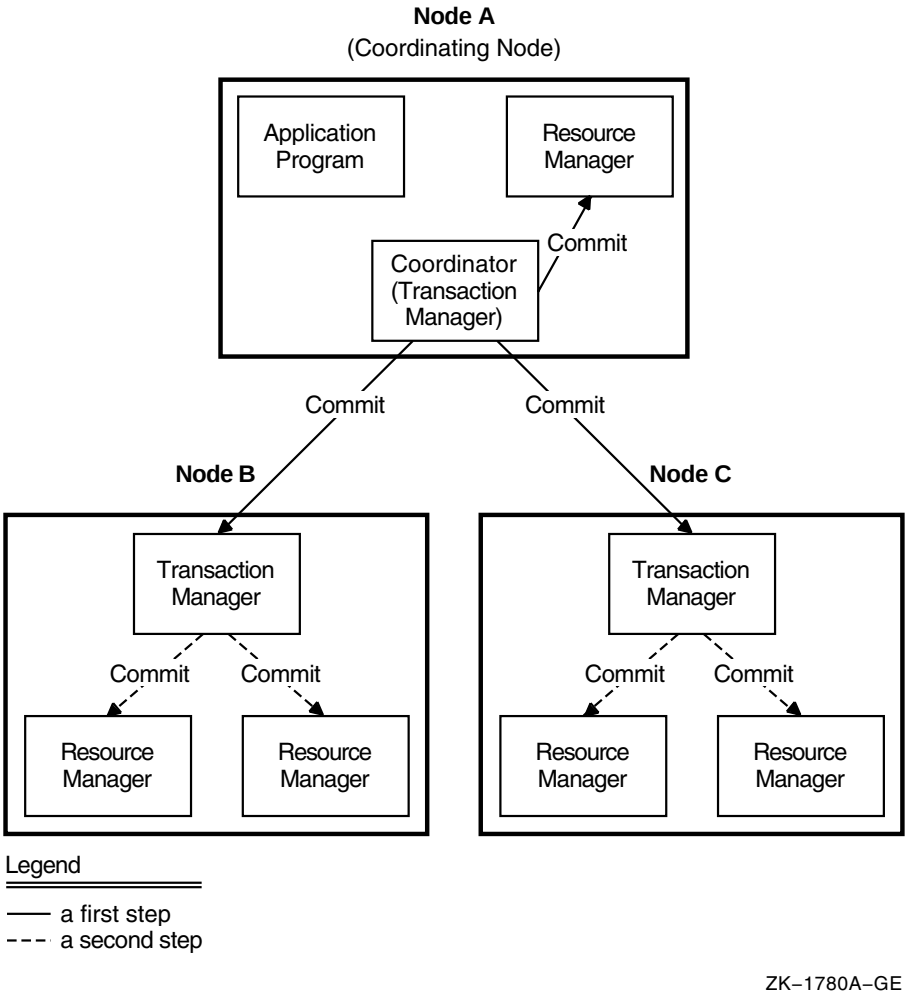
When all the participants in a distributed transaction vote yes, signifying a willingness to commit the changes to the database, the distributed transaction enters the commit phase of the two-phase commit protocol.

The following steps describe the commit phase and the actions the coordinator, transaction managers, and resource managers take:

1. The coordinator writes a commit record for this transaction to its transaction log file. As stated in Section 2.2.1, when the commit record is successfully written, the transaction is guaranteed to commit.
2. The coordinator instructs the local resource managers and the remote transaction managers to finish processing the commit phase and to make the transaction permanent. Each remote transaction manager instructs the resource managers on its node to finish processing the commit phase and to make the transaction permanent.

Figure 2–7 shows the coordinator and transaction managers initiating the commit phase.

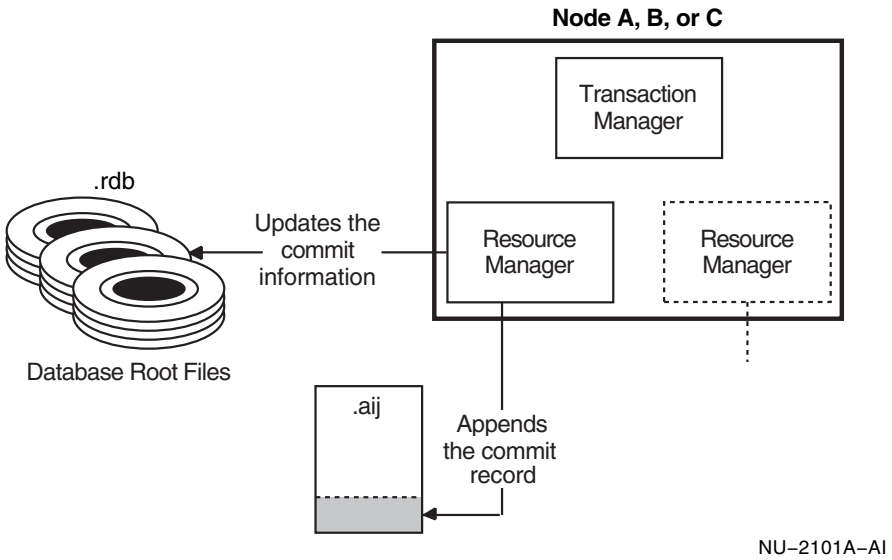
Figure 2–7 Initiating the Commit Phase



3. Each resource manager commits the transaction.

   As Figure 2–8 shows, the resource manager updates the commit information in the database root file (file type .rdb) and writes a commit record to the .aij file. At this point, entries in the .ruj file associated with this transaction are forgotten because before-image values are no longer needed.

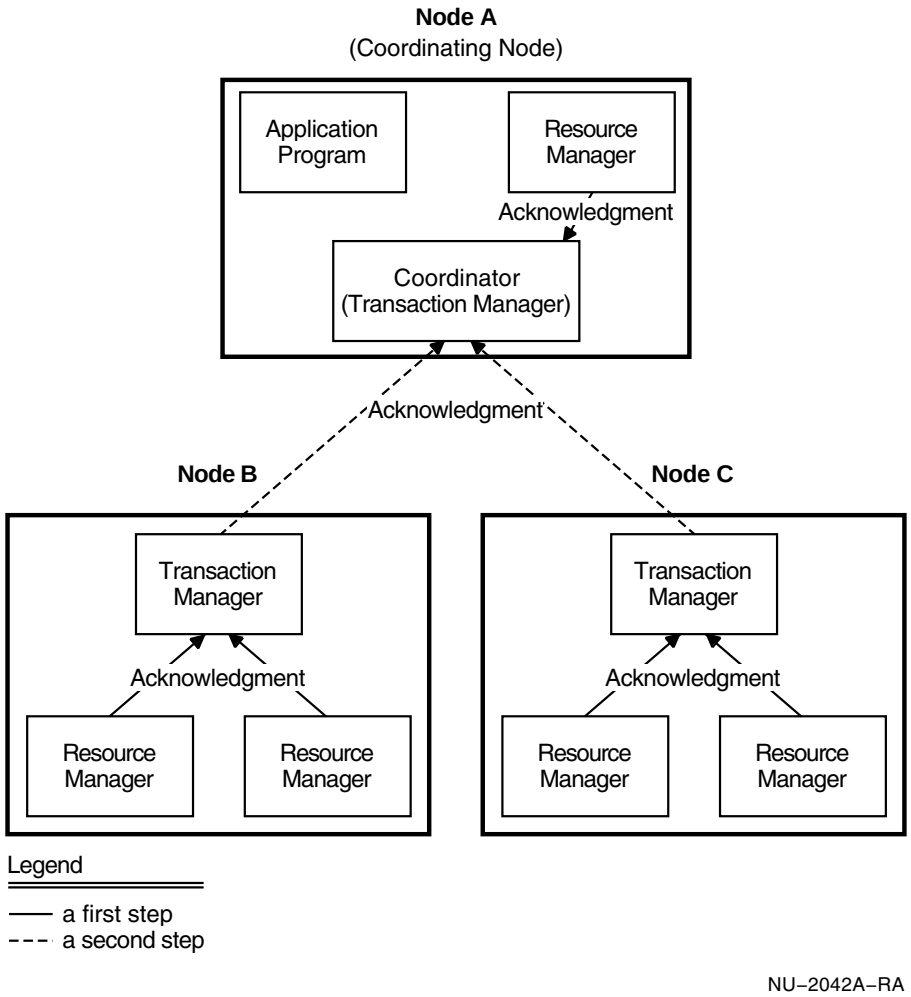
Figure 2–8 Resource Managers Commit the Transaction



4. Each resource manager sends an acknowledgment to its local transaction manager that it has committed the transaction. Then, each transaction manager sends an acknowledgment to the coordinator.

   Figure 2–9 shows the resource managers and transaction managers acknowledging that the transaction has been committed.

Figure 2–9 Acknowledging That the Transaction Has Been Committed



2.3 Understanding the Database Recovery Process

The two-phase commit protocol ensures the integrity and consistency of your application even if a process or image failure occurs. For example, even if a remote node fails, the two-phase commit protocol ensures that either all the databases in the transaction commit or none commits.

If a process or image failure occurs *before* a participant votes, the transaction cannot be committed. The following describes the actions the coordinator, transaction managers, and resource managers take when this kind of failure occurs:

1. The transaction manager on the node on which the failure occurs immediately tells the coordinator that it will not be able to commit the transaction. That is, the transaction manager votes no.
2. The coordinator instructs each local resource manager and the remote transaction managers to roll back any changes to the database. Each remote transaction manager instructs the resource managers on its node to roll back any changes made to the database.
3. Each resource manager performs the following tasks:
   - Uses the .ruj file to roll back changes
   - Writes a rollback record into the .aij file

If a process or image fails *after* a participant votes but before all the changes to all the databases involved in the transaction have been committed, the following occurs:

1. The Oracle Rdb monitor creates a database recovery (DBR) process for each database attachment affected by the process or image failure.
2. The DBR process searches the .ruj file for that database attachment and finds a prepare record.
3. Because it finds a prepare record, the DBR process knows that this database is involved in a distributed transaction. The DBR process asks the local transaction manager what to do. Then, the transaction manager asks the coordinator what to do.
4. The coordinator tells the DBR process to:
   - Roll back, if the coordinator has received any no votes
   - Commit, if the coordinator has received all yes votes
5. The DBR process rolls back or commits the changes to the database as directed by the coordinator.

Without the two-phase commit protocol, if a process or image failure occurs, the DBR process always rolls back the transaction. The two-phase commit protocol coordinates the actions of individual transaction participants so that either all the changes made during the transaction are committed or none of the changes is committed even if a process or image fails.

2.4 Completing Unresolved Transactions

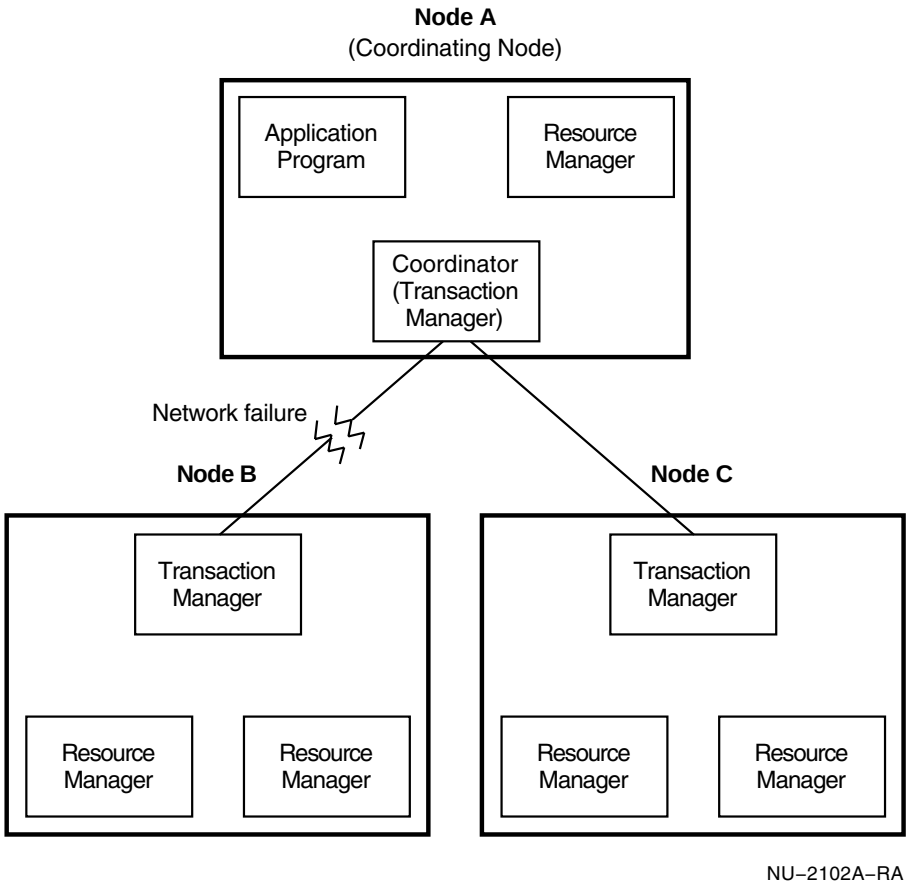
As described in Section 2.3, the “all-or-none” attribute of transactions is guaranteed even when image or process failures occur. On rare occasions, an unexpected failure can occur that causes the coordinator to be unreachable for an extended period of time. For example, the network connection may be lost or the node itself may have suffered some damage. If these situations occur *after* the participants vote, but *before* a transaction completes on all nodes, the transaction is unresolved.

The DBR process cannot complete the transaction until the coordinator is reachable. While the coordinator is unreachable, the DBR process does not let users access databases affected by unresolved transactions. For cases when you cannot wait for the coordinator to become reachable, Oracle Rdb provides a manual way to complete unresolved transactions by letting the DBR process proceed without instructions from the coordinator.

If you choose to circumvent the two-phase commit process by manually completing transactions, you should complete the transaction in the same manner for all of the databases participating in the distributed transaction. To be able to manually complete unresolved transactions in a timely fashion, you should identify all databases that are involved in distributed transactions before you run an application.

Figure 2–10 shows a distributed transaction that is unresolved because the network connection from Node B to the coordinator has failed.

Figure 2–10 Unresolved Distributed Transaction



In Figure 2–10, users cannot access the databases involved in the distributed transaction until the transaction completes. To resolve this transaction without waiting for the network connection to be restored, use Oracle RMU, the Oracle Rdb management utility. Using the RMU commands described below, you find out if a database is involved in an unresolved transaction and whether the other databases in the transaction (identified by the distributed TID) have committed or rolled back the changes. Then, you complete the transaction for all databases in the same manner. That is, if one database committed its changes, all the databases in the transaction should commit their changes.

To manually complete unresolved transactions, take the following steps:

1. Use the RMU Dump command with the Users and State=Blocked qualifiers to generate a list of unresolved transactions. Depending upon your application, you might need to generate the list for more than one database.
2. Examine the list to identify transactions that need to be resolved. Note the transaction sequence numbers (TSNs) associated with these transactions.
3. Refer to your application to determine which databases are affected by these transactions.
4. Consult with the database administrators for all databases affected by the transactions to determine the state to which to alter an unresolved transaction. Collaboration is necessary because the transaction might have completed on one node before the coordinator became unreachable to the other nodes. When this occurs, the transaction must be altered to the same state on all nodes affected by the transaction.
5. Use the RMU Resolve command to complete unresolved transactions on each database.

Sometimes, not only does an unexpected failure occur that causes the coordinator to be unreachable for an extended period of time, but your database may also be corrupted. If this happens, you should take the following steps:

1. Use the RMU Dump After_Journal command with the State=Prepared qualifier to generate a list of records associated with unresolved transactions in the .aij file. Depending upon your application, you might need to generate the list for more than one .aij file.
2. Examine the list to identify transactions that need to be resolved. Note the transaction sequence numbers (TSNs) associated with these transactions.
3. Refer to your application to determine which databases are affected by these transactions.
4. Consult with the database administrators for all databases affected by the transactions to determine the state to which to alter an unresolved transaction. Collaboration is necessary because the transaction might have completed on one node before the coordinator became unreachable to the other nodes. When this occurs, the transaction must be altered to the same state on all nodes affected by the transaction.
5. Use the RMU Restore command to restore the database from the latest backup file (file type .rbf).

6. Use the RMU Recover Resolve command to restore the database and alter these transaction records in the .aij file.

Chapter 5 explains in detail how to complete unresolved transactions.

This chapter explained how the two-phase commit protocol and distributed transactions work. Chapter 3 discusses considerations in designing databases and applications for use with distributed transactions.

3

Designing Databases and Applications for Distributed Transactions

Most of the principles of designing databases and application programs are the same for distributed transactions as for non-distributed transactions. However, because the two-phase commit protocol ensures the integrity of data even if a transaction involves more than one database, you might consider modifying the design of your databases. Moreover, using distributed transactions can have implications for how you design applications.

This chapter describes:

- Strategies for designing databases for use with distributed transactions
- Considerations in using distributed transactions in a VMScluster environment
- Privileges needed to use databases with distributed transactions
- Methods to avoid deadlock in distributed transactions
- Controlling when an application attaches to a database
- Controlling when an SQL application detaches from a database

3.1 Designing Databases for Distributed Transactions

Because Oracle Rdb uses the two-phase commit protocol with distributed transactions, it allows you greater flexibility in designing your databases. Now, instead of designing one large database, you can design two or more smaller databases and still preserve the integrity of the data. The two-phase commit protocol lets you group several databases with a single distributed transaction because the two-phase commit protocol ensures that the transaction commits changes to a database only if it can commit the changes to *all* the databases involved in the transaction.

When you use distributed transactions, you can design several small databases, rather than one large database. There are four basic strategies for designing databases for distributed transactions:

- Partitioning by business function
- Partitioning geographically
- Replicating data
- Combining any of the preceding strategies

3.1.1 Partitioning Databases by Business Function

Partitioning databases by business function means that you divide one large database into several smaller databases based on the functions of the different organizations of the business.

Often a database grows as a business grows. You can divide a large database so that data used by a particular department is contained in one database and is separate from data used by other departments. If you divide the database in this way, most of the transactions will use only one database. However, when you need the data from more than one department, you can start a distributed transaction and be assured that the two-phase commit protocol will guarantee that the databases remain logically consistent with one another.

For example, you can divide a database so that the warehouse inventory is in one database and the accounts receivable data is in another database. In many applications, the transactions involve only one of the databases. However, an order-entry application can start a distributed transaction that involves both databases without compromising the integrity of the data. In a single distributed transaction, Oracle Rdb can decrease the warehouse inventory by the items ordered and update the accounts receivable information.

Even if all of the database users are on the same cluster of nodes, you may achieve some performance gains by dividing your database into smaller databases. If you logically divide one large database into several smaller ones, most of your transactions will attach to only one database and will be non-distributed transactions. In addition, if all applications that attach to a particular database start on the same node, the nodes will spend fewer resources managing database locks and communicating with other nodes in the cluster.

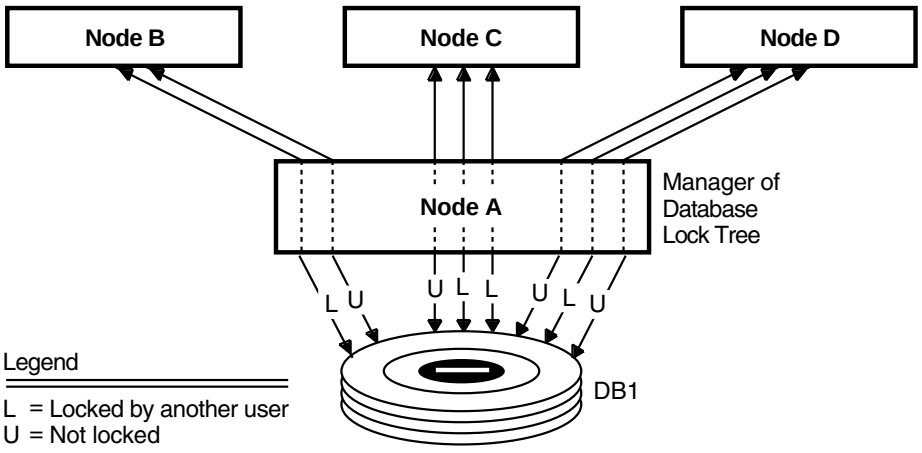
For example, assume you have a cluster with four nodes, A, B, C, and D, and one large database, DB1. If the first application to attach to DB1 starts on node A, node A is responsible for managing the database lock tree. When an application starts on node B and attaches to DB1, node B must first communicate with a clusterwide directory to find out which node manages the

database lock tree for DB1. Then, node B must communicate with node A to perform remote locking because node A manages the locking for DB1.

If applications that access DB1 start on nodes C and D, those nodes must also perform remote locking because the lock tree is managed by node A. Each request from these three nodes requires a message to and from node A. As requests increase, contention builds up and as a result, node A spends more and more of its time locking resources and communicating with the other nodes and has less time available for application tasks.

Figure 3–1 shows the communication that occurs between node A and the other nodes when node A manages the lock tree for a database all four nodes are using.

Figure 3–1 Communication Among Nodes in a Cluster When One Node Manages the Database Lock Tree



NU–2103A–RA

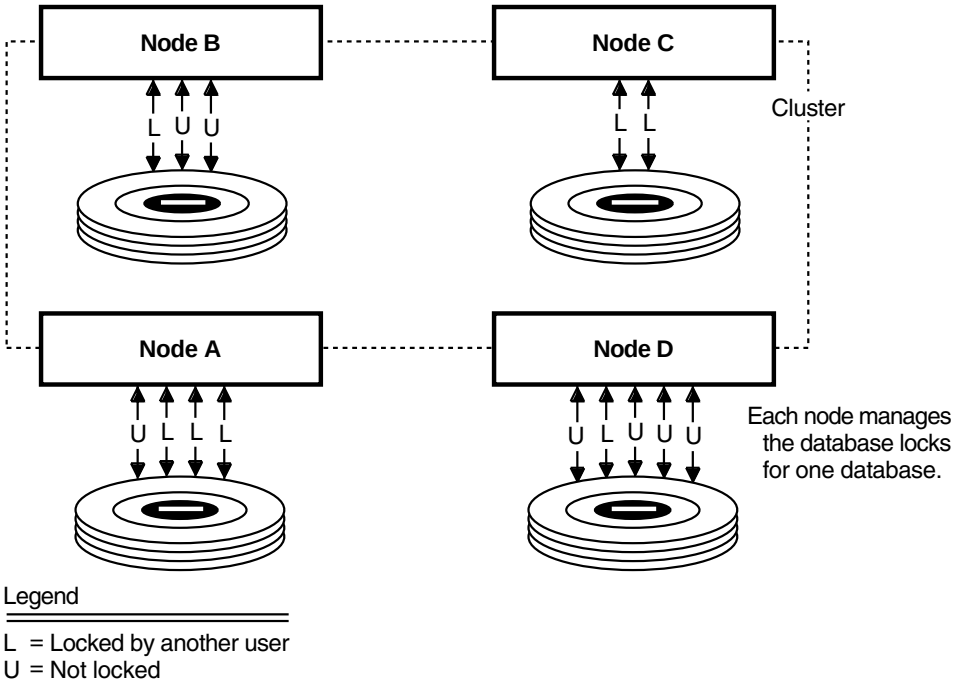
If you divide your database into several smaller databases and decide that applications that access each database should start on a specific node, you can reduce the amount of CPU time spent communicating among the nodes. If you have several databases, you can decide that all processes that attach to one database should start on one node and that all processes that attach to another database should start on another node.

For example, assume you have a cluster with four nodes (A, B, C, and D) and that you have divided one large database into four smaller ones (DB1, DB2, DB3, and DB4). If you start all applications that access only DB1 on node A, all applications that access only DB2 on node B, all applications that access only DB3 on node C, and all applications that access only DB4 on node D, each node manages the database lock tree for only one database. Thus, the nodes consume fewer system resources for the following reasons:

- Because there are multiple lock trees and fewer distributed transactions, the nodes perform less remote locking and communicating with one another, and have more time available for application tasks.
- Because you divide the databases logically, most of the transactions access only one local database and are non-distributed. The transaction path length remains short. Shorter transactions utilize resources better, including CPU time.
- Only distributed transactions span more than one node. If an application does use a distributed transaction, the tasks of the transaction are distributed over more than one node.

Figure 3–2 shows each node managing the lock tree for one database.

Figure 3–2 Communication Among Nodes in a Cluster When Each Node Manages One Database Lock Tree



By logically dividing your databases and controlling where applications start, you may improve performance without changing the hardware configuration of your cluster.

3.1.2 Partitioning Databases Geographically

Partitioning databases geographically means that you divide one large database into several smaller databases based on which site uses which data most frequently. You can divide a database so that the data used by one site is contained in a database located at that site, and data used by a second site is contained in a database located at the second site. By storing data close to where it is used most frequently, you increase the efficiency of database retrievals and updates. In addition, security may be enhanced because data that is not relevant to a particular site is less readily available.

For example, if a company has several warehouses, each warehouse has its own database to keep track of inventory. However, because the company wants its customer information in one place and wants its customers to be able to order from any of its warehouses, it has only one database for customer accounts. When a customer places an order, an application begins a distributed transaction that attaches to the customer database and to the local warehouse database. If the local warehouse does not have a particular item, the distributed transaction can attach to another warehouse's database so that it can fill the customer's order.

3.1.3 Replicating Data

Replicating data means that you store a copy of a database on more than one node and the nodes can be located at different sites. By storing separate copies of the database in different locations, each site can process queries quickly. In addition, data replication guarantees that you can still use the database even if the database is unavailable at one of the sites. Replication is particularly useful if most transactions are read-only and updates are infrequent. On those occasions when you update the databases, you can use a distributed transaction to ensure that any changes made to one copy of the database are made to all the copies. The primary disadvantage to data replication is the storage requirement: you must maintain full copies of the database at each site.

3.1.4 Combining Database Design Strategies

Combining any of the strategies means that you design databases that use more than one of the preceding strategies. For example, you might divide a database geographically, but decide that some of your data must be replicated at each site so that you can process queries quickly.

In summary, when you design databases for distributed transactions, you should consider storing the data where it will be accessed most frequently. In addition, you should consider database security and cost in your design. The two-phase commit protocol lets you design databases for use in distributed transactions without worrying that transactions that involve more than one database will compromise the integrity of your data.

3.2 Using Distributed Transactions in a VMScluster Environment

If possible, when you start an application that uses a distributed transaction, you should start the application on a node in a VMScluster environment. If one node fails, the coordinator is still available on another node in the cluster and is able to take the appropriate action to complete the transaction.

3.3 Required Privileges for Distributed Transactions

If you want to start a distributed transaction involving a particular database, you must have the DISTRIBTRAN privilege for that database.

Because SQL uses the two-phase commit protocol by default when you compile your application with the SQL module processor or the SQL precompiler, it is especially important that you have the DISTRIBTRAN privilege for all databases involved in distributed transactions.

If you do not have the DISTRIBTRAN privilege on a particular database and you try to start a distributed transaction involving that database, Oracle Rdb returns an error and does not start the transaction.

As with all database privileges, the owner of the database is automatically granted the DISTRIBTRAN privilege.

The following privileges override the DISTRIBTRAN privilege:

- The SQL privilege DBADM
- The OpenVMS privilege SYSPRV
- The OpenVMS privilege BYPASS

When you start a distributed transaction that attaches to a database on a remote node, Oracle Rdb checks that the account on the remote node has the DISTRIBTRAN privilege. For example, if you use a proxy account on the remote node, the proxy account must have the DISTRIBTRAN privilege on that database. The *Oracle Rdb Installation and Configuration Guide* discusses your options for attaching to remote databases, including using proxy accounts.

For more information about granting privileges, see the *Oracle Rdb Guide to Database Design and Definition* and the *Oracle Rdb SQL Reference Manual*.

3.4 Avoiding Deadlock with Distributed Transactions

When you use distributed transactions to access databases on remote systems, you run a risk of encountering undetected deadlock. **Deadlock** occurs when two users are locking resources that each needs and neither user can continue until the other user ends a transaction. When deadlock occurs on the same node or the same cluster, the OpenVMS lock manager detects the deadlock and issues the deadlock error condition to one user. However, when a transaction accesses databases on remote systems, the OpenVMS lock manager cannot detect the deadlock.

To help avoid distributed deadlock, Oracle Rdb provides the following methods to set the amount of time a transaction waits for locks to be released:

- The logical name RDM$BIND_LOCK_TIMEOUT_INTERVAL

  You can specify a default wait interval by defining the logical name in the process, group, or system logical name table. The following example shows how to define the wait interval for the system as 15 seconds:

  ```
  $ DEFINE/SYSTEM RDM$BIND_LOCK_TIMEOUT_INTERVAL 15
  ```

- The WAIT <interval> clause of the SQL SET TRANSACTION or DECLARE TRANSACTION statements

  Use this clause to specify a wait interval for an application. The following example shows a SET TRANSACTION statement with a wait interval of 15 seconds:

  ```
  SET TRANSACTION READ WRITE WAIT 15;
  ```

- The LOCK TIMEOUT INTERVAL clause of the SQL statements CREATE DATABASE and ALTER DATABASE

  Use this clause to specify a database-wide wait interval. The following example shows an ALTER DATABASE statement that sets the wait interval for the database to 25 seconds:

  ```
  ALTER DATABASE FILENAME PERSONNEL
        LOCK TIMEOUT INTERVAL IS 25 SECONDS;
  ```

The amount of time you specify for the wait interval depends upon your application. However, as a general guideline, use a value greater than the value specified in the SYSGEN parameter DEADLOCK_WAIT.

When you use a wait interval, you should be aware of the following rules:

- When you define the RDM$BIND_LOCK_TIMEOUT_INTERVAL logical name in the system logical name table, you should be aware that the wait interval does not pertain to all the processes on the cluster. It pertains to only to processes on the local node.
- If your application specifies NOWAIT, Oracle Rdb does not use a wait interval even if the logical name or database parameter specifies one.
- When you specify a wait interval in more than one way—any combination of logical name, application wait interval, and database parameter— Oracle Rdb uses the lowest wait interval. The following describes in more detail which wait interval takes precedence:
  - If you specify an application wait interval, that interval overrides any wait interval specified by the logical name that has a *higher* value.

- If you specify a database-wide wait interval, that interval overrides any other wait interval that has a *higher* value.

No matter which method you use to specify the wait interval, Oracle Rdb returns a lock conflict error if the resource is still locked after the transaction waits the specified interval.

Note that the wait interval is expressed in seconds, but that the time period is approximate.

3.5 Controlling When Applications Attach to Databases

In certain circumstances, you may want to consider explicitly calling DECdtm services even though your application only uses Oracle Rdb databases and can implicitly call DECdtm services. Specifically, if your application starts a transaction using one database and performs some tasks, and then, based on the outcome of an operation, adds one of several databases to the transaction, then explicitly calling DECdtm services lets you control when your application attaches to the databases.

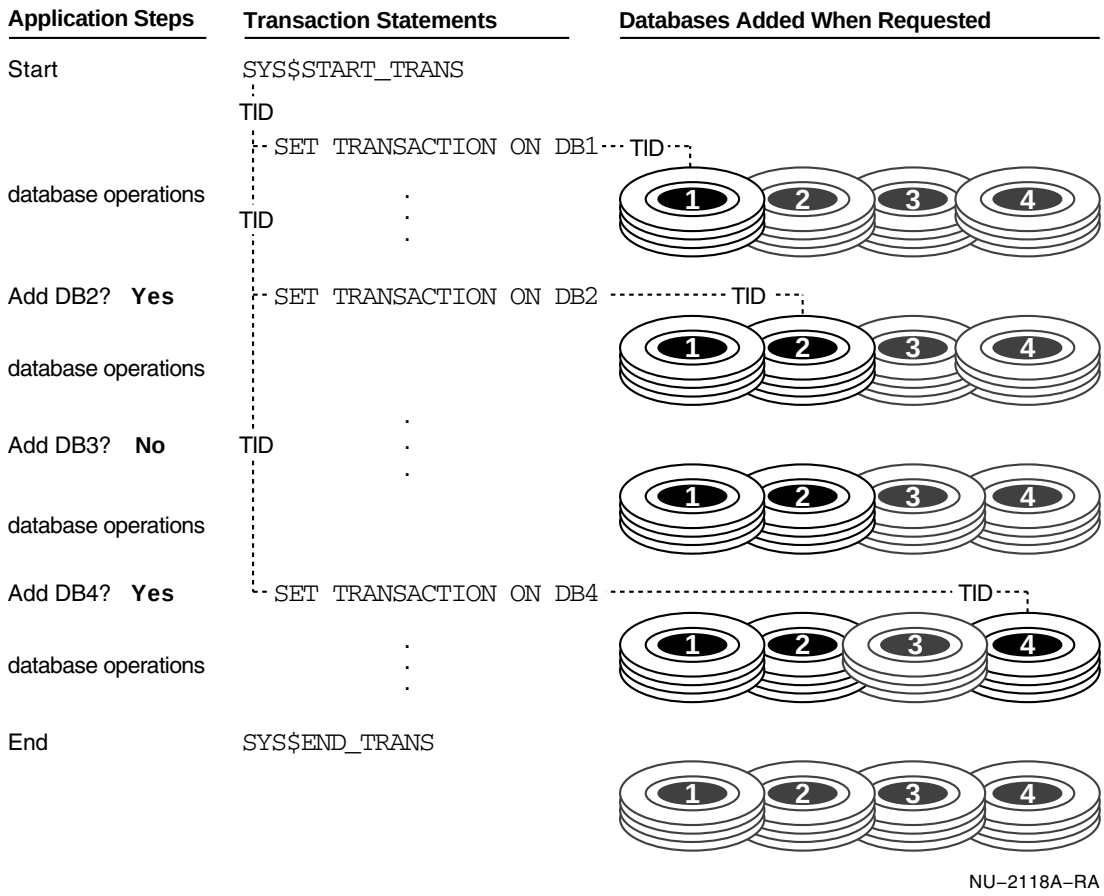
When you call the SYS$START_TRANS system service explicitly, you can start a transaction using one database. Then, after the transaction has performed some tasks, your application can decide to add one or more of several databases to the distributed transaction. You can add another database to that distributed transaction by passing the distributed TID to the database. You eliminate the overhead of including unneeded databases in a transaction.

For example, you may have an application that starts a transaction using one database, DB1. Then, based on certain criteria, the application determines whether to add DB2, DB3, or DB4 to the distributed transaction.

If you explicitly call the SYS$START_TRANS system service, your application can pass the distributed TID to the first database, DB1, by using the SQL statement SET TRANSACTION. After your application updates DB1, it can determine whether DB2, DB3, or DB4 should be added to the existing distributed transaction. Your application can add DB2, DB3, or DB4 to the transaction by using another SET TRANSACTION statement to pass the distributed TID to the next database.

Figure 3–3 illustrates how you can add databases to an already started transaction by explicitly calling the DECdtm system services.

Figure 3–3 Using DECdtm Services to Add Databases to Existing Transactions



By explicitly starting a distributed transaction with the SYS$START_TRANS system service, you do not have to start the transaction using all three databases. You can eliminate the overhead of including in a transaction databases that the application determines it does not need.

See Section 4.4 for information about how to pass the distributed TID to the SET TRANSACTION statement.

3.6 Controlling When SQL Applications Detach from Databases

You can use SQL connections and explicit calls to DECdtm services to control when you attach and detach from specific databases. For example, you may have an application that starts a transaction using two databases, DB1 and DB2. After that transaction completes, the application starts another transaction using the databases DB2 and DB3. The application no longer needs to be attached to the database DB1. In fact, you want the application to detach from the database so that the resources can be used by other transactions and applications. To detach from one database while remaining attached to other databases, you must start transactions by explicitly calling DECdtm system services *and* you must associate each database with an SQL connection.

First, you set up multiple connections, one for each database or set of databases. Then, your application explicitly calls the SYS$START_TRANS system service and, for each database or set of databases, sets the connection and adds the databases in the connection to the transaction by passing the distributed TID to the SET TRANSACTION statement executed within that connection.

When the transaction completes, you can disconnect from unneeded connections by using the DISCONNECT statement and specifying only the unneeded connection.

For example, your application explicitly calls the SYS$START_TRANS system service. Then, it sets up the connection CON1 for the database DB1 and adds the database to the transaction; it sets up the connection CON2 for DB2 and adds the DB2 database to the transaction; and finally it sets up the connection CON3 for DB3 and adds the DB3 database to the transaction. When the distributed transaction completes, your application detaches from DB1 by disconnecting from the connection CON1. Now your application can start a transaction using DB2 and DB3 in connections CON2 and CON3.

See Section 3.5 for information about how to add databases to a transaction. See the *Oracle Rdb Guide to SQL Programming* for information on how to set up connections.

This chapter discussed how to design databases for use with distributed transactions and the privileges you need to use databases with distributed transactions. In addition, it discussed considerations in designing applications that use distributed transactions and when to explicitly call DECdtm services to control when your application attaches and detaches from databases. Chapter 4 describes how to write SQL applications that use distributed transactions.

3.7 Avoiding Excess Transaction Errors

The DECdtm two-phase commit protocol allows multiple, concurrent transaction contexts to be active in the same process. Oracle Rdb is restricted to a single active transaction context at any given time. If Oracle Rdb detects that a transaction is being started while another one is active, it returns an RDB$_EXCESS_TRANS error and does not start the transaction. When your application receives an RDB$_EXCESS_TRANS error, it normally indicates an application error.

However, because of a DECdtm optimization, an application might receive an RDB$_EXCESS_TRANS error even though multiple transactions should not be active. In particular, DECdtm will return from a SYS$END_TRANS(W) call as soon as all participating Resource Managers (RMs) vote "yes" during the prepare phase without waiting for the RMs to complete the commit processing. This behavior occurs even if you use the synchronous version of the system service, SYS$END_TRANSW, or if you use SYS$SYNCH with SYS$END_TRANS. The result can be a race condition where the application issues another SYS$START_TRANS call before the participating RMs (including Oracle Rdb) have completed the commit processing. The race condition may last for a significant time if one of the participating Oracle Rdb databases is on a remote node. Even if the application is using implicit distributed transactions, a new SYS$START_TRANSW call may be issued by Oracle Rdb in response to a SQL statement submitted by the application. Regardless of how the SYS$START_TRANS(W) process gets started, Oracle Rdb receives a new call to its transaction AST while the previous transaction is still not completely finished. In order to avoid an RDB$_EXCESS_TRANS error when a new transaction is started, Oracle Rdb tests to see if a previous transaction is in the process of committing. If so, it will wait while periodically retesting to see if the previous transaction has completed. At the end of this transaction start waiting period Oracle Rdb returns an RDB$_EXCESS_TRANS error if the previous transaction has not completed.

The default amount of time for the transaction start waiting period is three seconds. If your application has remote database participants and sometimes receives an RDB$_EXCESS_TRANS error even though you have not started multiple transactions, it may be necessary to adjust the length of the waiting period. It can be adjusted for the remote databases by using the SQL_TRANS_START_WAIT configuration file parameter. See the *Oracle Rdb Installation and Configuration Guide* for instructions on how to use configuration files.

4

Using SQL with Distributed Transactions

Before you read this chapter, you should read Chapter 1 and Chapter 2. These chapters introduce the two-phase commit protocol and explain how distributed transactions work. In addition, you should read Chapter 3, which explains considerations in designing databases and applications for distributed transactions.

You can use the two-phase commit protocol with application programs that use the SQL module processor or the SQL precompiler. In addition, you can use the two-phase commit protocol with dynamic SQL. You use the two-phase commit protocol in dynamic SQL by following the discussions in this chapter for using the two-phase commit protocol for the SQL module processor or the SQL precompiler.

To take advantage of the two-phase commit protocol, your application must invoke, either implicitly or explicitly, the DECdtm system services.

This chapter describes:

- The different ways you can use the two-phase commit protocol with SQL
- How to disable the two-phase commit protocol
- How to use the two-phase commit protocol so that applications implicitly call the DECdtm system services
- How to write applications that explicitly call DECdtm services
- How to use the sample application

4.1 Using the Two-Phase Commit Protocol with SQL

When you compile an application program using the SQL module processor or the SQL precompiler, transactions use the two-phase commit protocol by default.

Applications can use the two-phase commit protocol by calling the DECdtm system services implicitly or explicitly. SQL provides the following ways for application programs to use the two-phase commit protocol:

- By implicitly calling the DECdtm system services

  You can use the two-phase commit protocol simply by compiling your programs. When you do, Oracle Rdb invokes the DECdtm system services for your application.

  See Section 4.3 for more information about using SQL to call the DECdtm system services implicitly.

- By explicitly calling the DECdtm system services and passing the value of the distributed transaction identifier (TID) to the application

  If your application starts a distributed transaction that includes Oracle Rdb and other resource managers, your application *must* explicitly invoke the DECdtm system services. For example, if your application starts a distributed transaction using Oracle Rdb and Oracle CODASYL DBMS, your application must explicitly call the DECdtm system services SYS$START_TRANS and SYS$END_TRANS.

When you explicitly call the DECdtm system services, you can pass the value of the distributed TID to the application in one of the following ways:

- Using a context structure and variables to pass the value of the distributed transaction identifier (TID)

  The **context structure** is a host language structure that contains the distributed TID as one of its elements. You must write new application programs or modify existing programs to declare the context structure and pass the value of the distributed TID to SQL statements.

- Using the default transaction support in distributed transactions

  You can specify whether or not SQL automatically uses the default distributed transaction identifier (TID) established by the DECdtm system service SYS$START_TRANS. To specify that SQL automatically use the default distributed TID, your application must explicitly call the DECdtm system services and you must compile your application using the TRANSACTION_DEFAULT=DISTRIBUTED qualifiers.

  These qualifiers eliminate the need to declare context structures in SQL precompiled programs and host language programs and to pass the context structure to embedded SQL statements or SQL module procedures.

See Section 4.4.4 for more information about using SQL module language to start distributed transactions explicitly.

See Section 4.4.5 for more information about using precompiled SQL to start distributed transactions explicitly.

4.2 Disabling the Two-Phase Commit Protocol

You can disable the two-phase commit protocol by defining the logical name SQL$DISABLE_CONTEXT to be "True," as shown in the following example:

```
$ DEFINE SQL$DISABLE_CONTEXT TRUE
```

You can override the definition of the logical name by defining it in a lower-level logical name table. For example, if the logical name is defined as "True" in the system logical name table, you can override it by defining it as "False" in the process logical name table. If you define the logical name as "True" and your application attempts to start an explicit distributed transaction, SQL displays a warning message reminding the user that the two-phase commit protocol will not be used for that program or SQL module.

Keep in mind that it is at compile time that SQL evaluates whether the two-phase commit protocol is enabled or disabled. For example, if the SQL$DISABLE_CONTEXT logical name is defined as "True" when you precompile your SQL program, the program cannot take advantage of the two-phase commit protocol. This is true even if the program uses dynamic SQL and the logical name is defined as "False" or is not defined when the user executes the program. If you compile an SQL module while the SQL$DISABLE_ CONTEXT is defined as "True," dynamic SQL will not use the two-phase commit protocol in that particular module.

Because you cannot use batch-update transactions with distributed transactions, you should define the SQL$DISABLE_CONTEXT logical name as "True" before you start a batch-update transaction. (Distributed transactions require that you are able to roll back transactions. Because batch-update transactions do not write to recovery-unit journal files, batch-update transactions cannot be rolled back.)

If you attempt to start a distributed transaction using a batch-update transaction, what happens depends upon whether you call the DECdtm system services implicitly or explicitly and which SQL statement you use to start the transaction:

- If you start a batch-update transaction and explicitly call the DECdtm system services, SQL returns an error at compile time.

- If you start a batch-update transaction and implicitly call the DECdtm system services, SQL takes the following actions:
  - If you use a SET TRANSACTION statement with the BATCH UPDATE clause, SQL starts a non-distributed transaction.
  - If you use a DECLARE TRANSACTION statement with the BATCH UPDATE clause, SQL returns an error at compile time.

4.3 Using the Two-Phase Commit Protocol Implicitly

If your application attaches only to Oracle Rdb databases, you can take advantage of the two-phase commit protocol simply by compiling your application program. When you do this, your application does not have to explicitly invoke the DECdtm system services; Oracle Rdb invokes them on behalf of the application.

When you compile your application programs, Oracle Rdb treats most transactions that attach to more than one Oracle Rdb database as distributed transactions. However, you should note the following:

- If the logical name SQL$DISABLE_CONTEXT was defined as "True" at compile time, SQL does not start a distributed transaction.
- You cannot use a batch-update transaction with distributed transactions. Section 4.2 explains what actions SQL takes if you attempt to do so.
- If the transaction involves only one database attachment, Oracle Rdb does not use the two-phase commit protocol.
- If DECdtm software is not running on all nodes involved in the distributed transaction, Oracle Rdb returns an error and the transaction does not start.
- If a transaction involves Oracle Rdb and a resource manager that requires that your application call the DECdtm services explicitly, and your application calls the DECdtm system services implicitly, Oracle Rdb starts a transaction that involves only the Oracle Rdb databases.
- If a transaction involves a database management product that does not support DECdtm services, Oracle Rdb returns an error and does not start the transaction.

4.3.1 Handling Errors in Implicit Distributed Transactions

Although you can take advantage of the two-phase commit protocol by merely compiling your application program, consider adding some error handling code. In particular, you should check the RDB$LU_STATUS field of the message vector for the following errors:

- RDB$_DISTABORT

  Oracle Rdb returns this error when the transaction has been rolled back for any reason, such as a constraint violation or network failure. Often, as in the case of a constraint violation, Oracle Rdb returns a secondary error. Your application should test for the secondary error in the RDB$_LU_ARGUMENTS[1] field and take action based on that error.

  When Oracle Rdb returns the RDB$_DISTABORT error with no secondary error, you should detach from the databases using the SQL DISCONNECT statement. Then, you can start the transaction again.

- RDB$_IO_ERROR

  Oracle Rdb returns this error when the network link fails.

  Roll back the transaction and detach from the databases using the SQL DISCONNECT statement. Then, you can start the transaction again.

Example 4–1 is an excerpt from a C language program that shows how you can test for these errors. In the example, a C program is calling procedures in a SQL Module Language program (for example, `start_trans`). As an alternative to the `globalvalue` language extension, the C header file `rdbmsgs.h`, which contains `#define` statements for the error codes, can be included. If you use `rdbmsgs.h`, the `$` in the error code is replaced by an underscore. For example, `RDB$DISTABORT` would become `RDB_DISTABORT`.

Example 4–1 Trapping Errors in Implicit Distributed Transactions

```
   .
   .
   .
/* Declare the message vector. */

extern struct {
    int RDB$LU_NUM_ARGUMENTS;
    int RDB$LU_STATUS;
    int RDB$LU_ARGUMENTS[18]; } RDB$MESSAGE_VECTOR;

/* Declare the error codes. */
```

(continued on next page)

Example 4–1 (Cont.) Trapping Errors in Implicit Distributed Transactions

```
globalvalue long RDB$_DISTABORT;
globalvalue long RDB$_NOSECERR;
globalvalue long RDB$_IO_ERROR;

/* Define a macro for error handling.  If the program returns the RDB$_DISTABORT
error, the macro checks to see if there is a secondary error.  If there is not,
the macro detaches from the databases and starts the program again.  If there is
a secondary error, the macro writes the error message and detaches from the
databases.  If the program returns the RDB$_IO_ERROR, the macro rolls back the
transaction, disconnects the databases, and starts the program again. */

#define check_status { \
                      if (RDB$MESSAGE_VECTOR.RDB$LU_STATUS == RDB$_DISTABORT) \
                      { \
                        if (RDB$MESSAGE_VECTOR.RDB$LU_ARGUMENTS [1] == RDB$_NOSECERR) \
                        { \
                          printf ("No secondary error detected.\n"); \
                          disconnect_databases (&sqlcode); \
                          goto start_prg; \
                        } \
                        else \
                         { \
                            short int errlen; \
                            char errbuf[256]; \
                            struct dsc$descriptor_s errbuf_dsc; \
                            printf ("Secondary error detected."); \
                            errbuf_dsc.dsc$b_class = DSC$K_CLASS_S; \
                            errbuf_dsc.dsc$b_dtype = DSC$K_DTYPE_T; \
                            errbuf_dsc.dsc$w_length = 255; \
                            errbuf_dsc.dsc$a_pointer = &errbuf; \
                            sql$get_error_text(&errbuf_dsc, &errlen); \
                            errbuf[errlen] = 0; \
                            printf("\nerror text: \n%s", errbuf); \
                            disconnect_databases (&sqlcode); \
                            exit (1); \
                         } ; \
                      } \
                      else if (RDB$MESSAGE_VECTOR.RDB$LU_STATUS == RDB$_IO_ERROR) \
                      { \
                         printf ("I/O error detected.\n"); \
                         rollback_trans (&sqlcode); \
                         disconnect_databases (&sqlcode); \
                         goto start_prg; \
                      } \
                      else if (sqlcode < 0) \
                      { \
                         sql_signal (); \
                      } \
                    }
   .
   .
   .
```

(continued on next page)

Example 4–1 (Cont.) Trapping Errors in Implicit Distributed Transactions

```
start_prg:
/* Start a distributed transaction. */
        start_trans( &sqlcode, &context_struc);
        check_status;
   .
   .
   .
```

4.4 Using the Two-Phase Commit Protocol Explicitly

You can write applications that explicitly call the DECdtm system services to start and end distributed transactions. Note that if your application starts a distributed transaction that includes resource managers such as Oracle CODASYL DBMS, your application *must* explicitly call the DECdtm system services.

DECdtm provides the following system services for distributed transactions:

- SYS$START_TRANS or SYS$START_TRANSW

  The Start Transaction system service starts a transaction and allocates a unique distributed transaction identifier (TID). SYS$START_TRANS executes asynchronously; SYS$START_TRANSW executes synchronously.

- SYS$END_TRANS or SYS$END_TRANSW

  The End Transaction system service executes the prepare and commit phases of the distributed transaction. SYS$END_TRANS executes asynchronously; SYS$END_TRANSW executes synchronously.

- SYS$ABORT_TRANS or SYS$ABORT_TRANSW

  The Abort Transaction system service rolls back any changes made during a distributed transaction. SYS$ABORT_TRANS executes asynchronously; SYS$ABORT_TRANSW executes synchronously.

Note

Although most of the information in this chapter pertains to both the asynchronous and synchronous versions of the system services, the text usually names only the asynchronous versions. However, differences between the two versions are explained where appropriate.

The following example in the C language shows how to call the SYS$START_TRANS system service to start a distributed transaction:

```
status = SYS$START_TRANS(
         0,    /* efn    */
         0,    /* flags  */
         iosb, /* iosb   */
         0,    /* astadr */
         0,    /* astprm */
         (context_struc.distributed_tid) /* tid */
         );
```

If you explicitly call the SYS$START_TRANS system services, you must explicitly call SYS$END_TRANS or SYS$ABORT_TRANS. You cannot use the SQL statements COMMIT or ROLLBACK. If you do, SQL returns the error SQL$_NO_TXN with an SQLCODE value of –1005.

In addition, note that you cannot use the SQL DISCONNECT statement if an explicitly started transaction is still active. If you do, Oracle Rdb returns an error saying that the detach failed because the transaction is active. After the transaction completes, you can detach from the databases using the SQL DISCONNECT statement. The DISCONNECT statement lets you detach from those databases in your current connection (database environment), from databases in a specified connection, or from all attached databases. (See the *Oracle Rdb Guide to SQL Programming* and the *Oracle Rdb SQL Reference Manual* for more information about the SQL DISCONNECT statement and connections.)

For a detailed description of the DECdtm system services, the arguments, and the condition values returned in the I/O status block, see the OpenVMS documentation about DECdtm services.

4.4.1 Handling Errors in Explicit Distributed Transactions

When your application explicitly starts a distributed transaction, you should be aware of the following:

- If a commit-time constraint is violated, the transaction aborts, but you may not know why it aborted.
- You should test that each system service completes successfully.

If your application evaluates constraints at commit time and a constraint is violated, the distributed transaction aborts. Unfortunately, you may not be able to find out why the transaction aborted or what constraint was violated, because the DECdtm system service returns only one error. To make it easier to find out that a constraint has been violated, override the evaluation time of commit-time constraints. You can override the evaluation time by using the

SQL SET ALL CONSTRAINTS ON statement, command line options for the SQL precompiler and module processor, or the EVALUATING clause of the SET TRANSACTION or DECLARE TRANSACTION statement.

For example, if you include the SET ALL CONSTRAINTS ON statement before other SQL statements in your program, any commit-time constraints are evaluated at the end of each statement. As another example, you can use the EVALUATING AT VERB TIME clause of the SET TRANSACTION statement when you want row-by-row constraint evaluation. See the *Oracle Rdb Guide to SQL Programming* for more information about controlling when constraints are evaluated.

When your application explicitly starts a distributed transaction, you should test that each system service completes successfully. You can do this by checking that the service returns a value of SS$_NORMAL. In addition, you should check the value returned to the I/O status block (IOSB) from calls to the DECdtm system services. The following Fortran example calls the SYS$START_TRANSW system service and checks that it completes successfully:

```
        STATUS = SYS$START_TRANSW (
1                       %VAL(0),
2                       %VAL(1),
3                       IOSB,
4                       %VAL(0),
5                       %VAL(0),
6                       DIST_TID)
        IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))

        IF (STATUS .EQ. SS$_NORMAL) THEN
              IF (STATUS) STATUS = IOSB(1)
        IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))
        END IF
```

When you use an asynchronous system service, you should use the SYS$SYNCH system service to ensure that each system service completes successfully. To do this, you pass to SYS$SYNCH the event flag and the I/O status block that you passed to the asynchronous system service; SYS$SYNCH waits for the event flag to be set, then ensures that the system service (rather than some other program) sets the event flag by checking the I/O status block. If the I/O status block is zero, SYS$SYNCH waits until the I/O status block is not zero.

The following Fortran example calls the SYS$START_TRANS system service and checks that it completes successfully:

```
        STATUS = SYS$START_TRANS (
1                       %VAL(0),
2                       %VAL(1),
3                       IOSB,
4                       %VAL(0),
5                       %VAL(0),
6                       DIST_TID)
        IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))
        IF (STATUS .EQ. SS$_NORMAL) THEN
                STATUS = SYS$SYNCH (
1                               %VAL(0),
2                               IOSB)
        IF (STATUS) STATUS = IOSB(1)
        IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))
        END IF
```

For a detailed description of the DECdtm system services, the arguments, and the condition values returned in the I/O status block, see the OpenVMS documentation about DECdtm services. For information about how to test for successful completion after a system service call, see the OpenVMS documentation about system services.

For information about troubleshooting problems with distributed transactions, see Appendix A.

4.4.2 Using Context Structures with SQL Statements

When you declare a context structure in an application, you must associate it with most *executable* SQL statements. This is true whether you use SQL module language or precompiled SQL, although the method you use to associate the context structure with SQL statements differs depending upon which compiler you choose.

However, you cannot associate a context structure with the following categories of executable statements:

- Statements that you cannot execute if a transaction has already started. These statements are as follows:

  ALTER DATABASE
  CREATE DATABASE
  DROP DATABASE
  DROP PATHNAME

- Statements that do not execute within the scope of a transaction and that are independent of any transaction context. These statements are as follows:

  CONNECT
  DESCRIBE

DISCONNECT
Extended dynamic DECLARE CURSOR
RELEASE
SET CATALOG
SET CHARACTER LENGTH
SET CONNECT
SET DEFAULT CHARACTER SET
SET DEFAULT DATE FORMAT
SET DIALECT
SET IDENTIFIER CHARACTER SET
SET KEYWORD RULES
SET LITERAL CHARACTER SET
SET NAMES
SET NATIONAL CHARACTER SET
SET OPTIMIZATION LEVEL
SET QUOTING RULES
SET SCHEMA
SET VIEW UPDATE RULES

- Statements that you cannot use in transactions that have been started by explicit calls to DECdtm system services. These statements are as follows:

  COMMIT
  ROLLBACK

  (The DISCONNECT statement can be considered in this category as well as the previous category.)

Remember that you cannot associate a context structure with nonexecutable SQL statements. Moreover, you cannot pass a context structure to a multistatement procedure if that procedure contains a SET TRANSACTION, COMMIT, or ROLLBACK statement.

Using context structures to pass the value of the distributed TID allows you greater control in using or not using distributed transactions within a program.

For example, if your program uses distributed transactions in most cases, but you want to use a non-distributed transaction occasionally, you can pass a distributed TID containing zeros. If your program does not pass a context structure or passes a distributed TID containing all zeros, SQL starts a non-distributed transaction.

For information about using context structures in SQL module language, see Section 4.4.4.1; for information about using context structures in precompiled SQL, see Section 4.4.5.1.

4.4.3 Using Cursors with Distributed Transactions

When you use cursors with distributed transactions, note the following:

- In precompiled SQL, if one cursor statement contains the USING CONTEXT clause, all statements that refer to that cursor must contain the USING CONTEXT clause.
- You should explicitly close any cursors, unless you use the default transaction support as described in Section 4.4.4.2 and Section 4.4.5.3.

In any one module in a precompiled SQL program, if one statement that refers to a cursor contains the USING CONTEXT clause, all statements that refer to that cursor must contain the USING CONTEXT clause. The only exception is the DECLARE CURSOR statement, which does not take a USING CONTEXT clause.

For example, if an OPEN cursor statement contains a USING CONTEXT clause, all other statements that operate on that same cursor in the same module must contain a USING CONTEXT clause. Because the second OPEN statement in the following example is illegal, SQL returns an error:

```
main ()
  {
  exec sql declare a_cursor cursor for select * from employees;
  exec sql using context :ctx open a_cursor;
  .
  .
  .
  exec sql open a_cursor;
  }
```

Because the SYS$END_TRANS and SYS$ABORT_TRANS system services do not close cursors, you should explicitly close any open cursors before you end the transaction. You can use the SQL CLOSE statement or the sql_close_cursors( ) routine. The SQL CLOSE statement closes cursors individually; the sql_close_cursors( ) routine closes all open cursors. This routine takes no arguments. The following example shows an excerpt of an SQL precompiled program that uses the sql_close_cursors( ) routine to close two cursors:

```
   .
   .
   .
/* Fetch records from two cursors.  The program has already declared them and
   opened them. */
EXEC SQL USING CONTEXT :CONTEXT_STRUC FETCH CURSOR_A;
EXEC SQL USING CONTEXT :CONTEXT_STRUC FETCH CURSOR_B;
   .
   .
   .
/* Close both cursors.*/

  sql_close_cursors();
   .
   .
   .
```

If you use default transaction support, you do not need to close any cursors, because default transaction support closes all cursors for you.

For more information about the sql_close_cursors( ) routine, see the *Oracle Rdb SQL Reference Manual*.

4.4.4 Using SQL Module Language with Distributed Transactions

When you use the SQL module language, you can explicitly use the two-phase commit protocol by explicitly calling the DECdtm system services in the host language program, as described in Section 4.4. In addition, you must take *one* of the following actions:

- Declare and use a context structure to pass the distributed TID to SQL statements. See Section 4.4.4.1 for more information.
- Use the default transaction support for distributed transactions. See Section 4.4.4.2 for more information.

4.4.4.1 Using Context Structures with SQL Module Language

To use context structures to pass the value of the distributed TID to SQL statements, you must take the following actions:

- Declare a context structure in the host language program.
- Pass the address of the distributed TID to *most* SQL module procedures.
- Process the SQL source files using the CONTEXT qualifier in the SQL module processor command line.

The context structure contains the value of the distributed TID as one of its elements. Your application must declare and initialize the context structure in the host language program. When you declare the context structure, you must declare the distributed TID to be a data type that is 16 bytes. For example, if your host language program is written in the C language, you declare the context structure and initialize it by using the following code:

```
struct
    {
     long version;
     long type;
     long length;
     long distributed_tid[4];
     long end;
    } context_struc = {1,1,16,{0,0,0,0},0};
```

Your application must use the context structure to pass the address of the distributed TID from the host language program to procedures in the SQL module that are involved in the distributed transaction. You pass the context structure to procedures that contain *executable* SQL statements, except for statements that you cannot execute because a transaction has already started, statements that you cannot use because you have explicitly called the DECdtm system services, or statements that do not execute within the scope of a transaction. Section 4.4.2 lists the executable statements that do not take a context structure.

For example, to pass the context structure to an SQL module procedure called UPDATE_PERS, a host language program written in the C language would use code similar to the following:

```
update_pers( &sqlcode, &context_struc);
```

In addition to modifying the host language program, you must process the SQL module using the CONTEXT qualifier in the SQL module processor command line. The CONTEXT qualifier tells SQL that it should execute module language procedures in the context of a particular distributed transaction. When you use this qualifier, SQL generates an additional parameter for the procedures and places the parameter as the last parameter declared in the procedure. You should not modify the SQL module to add parameters for the context structure; the SQL module processor does this for you when you use the CONTEXT qualifier.

The format of the command line qualifier is:

```
/CONTEXT =  <value>
```

You can specify the following for the value:

- ALL

  If you specify ALL, SQL generates an additional parameter, the context parameter, for all the procedures in the module and declares this parameter in all the procedures. The procedures accept the context parameter passed from the host language program. The two-phase commit protocol is used in all the transactions in the module.

- NONE

  If you specify NONE, SQL does not generate an additional parameter for any of the procedures in the module. The procedures do not accept a context parameter if the host language program passes one to them. When you specify NONE, the application calls the DECdtm system services implicitly. That is, Oracle Rdb makes the calls on behalf of your application.

- The names of SQL module procedures

  If you specify the names of particular procedures, SQL generates an additional parameter, the context parameter, for the specified procedures and declares this parameter in those procedures. Only the specified procedures accept the context parameter specified by the host language program. Therefore, the two-phase commit protocol is used only in transactions in the specified procedures.

For more information about the CONTEXT command line qualifier, see the *Oracle Rdb SQL Reference Manual*.

Example 4–2 shows a simple C host language program that calls SQL module procedures.

Example 4–2 Writing Host Language Programs That Use Distributed Transactions to Modify Databases

```
/* This program, update_dist.c, calls procedures in the SQL module
*  update_dist_mod.sqlmod.  It modifies the department name of one
*  department.
*/

#include <ssdef.h>
#include <stdio.h>
#include <descrip.h>
#include <startlet.h>
```

(continued on next page)

Example 4–2 (Cont.) Writing Host Language Programs That Use Distributed Transactions to Modify Databases

```
main()
{
/* Declare and initialize the context structure.*/

struct
    {
     long version;
     long type;
     long length;
     long distributed_tid[4];
     long end;
    } context_struc = {1,1,16,{0,0,0,0},0};

long status;
long iosb[4];
void start_trans();
void update_pers();
void update_mfpers();

/* Declare sqlcode.*/

long sqlcode;

/* Call SYS$START_TRANS to start the transaction. */

        status = sys$start_transw(
            0, /* efn */
            0, /* flags */
            iosb, /* iosb */
            0, /* astadr */
            0, /* astprm */
            (context_struc.distributed_tid) /* tid */
            );

        start_trans( &sqlcode, &context_struc);

/* Call the procedure to update a row in the personnel database and
*  pass, by reference, the context structure.
*/

        update_pers( &sqlcode, &context_struc);

/* Call the procedure to update a row in the mf_personnel database and
*  pass, by reference, the context structure.
*/

        update_mfpers( &sqlcode, &context_struc);
```

(continued on next page)

Example 4–2 (Cont.) Writing Host Language Programs That Use Distributed Transactions to Modify Databases

```
/* Call SYS$END_TRANS to end the transaction.*/

        status = sys$end_transw(
            0, /* efn */
            0, /* flags */
            iosb, /* iosb */
            0, /* astadr */
            0, /* astprm */
            (context_struc.distributed_tid) /* tid */
            );

}
```

Example 4–3 provides the SQL procedures needed by the host language program update_dist.c.

Example 4–3 Writing SQL Modules That Use Distributed Transactions to Modify Databases

```
-- This SQL module, update_dist_mod.sqlmod, provides the SQL procedures
-- needed by the programs update_dist.c and update_dist_txn_def.c.
--
-- If you link this module with update_dist.c, use the CONTEXT qualifier
-- on the SQL module command line.
--
-- If you link this module with update_dist_txn_def.c, use the qualifier
-- TRANSACTION_DEFAULT=DISTRIBUTED on the SQL module command line.
--
-- This program modifies the department name of one department.
-------------------------------------------------------------------------
-- Header Information Section
-------------------------------------------------------------------------
MODULE           UPDATE_DIST_MOD       -- Module name
LANGUAGE         C                     -- Language of calling program
AUTHORIZATION    SAMPLE_USER           -- Authorization ID
ALIAS            RDB$DBHANDLE          -- Default alias
PARAMETER COLONS                       -- Parameters are prefixed by colons
-------------------------------------------------------------------
-- Declare Statement Section
-------------------------------------------------------------------
DECLARE PERS ALIAS FOR FILENAME personnel
DECLARE MF_PERS ALIAS FOR FILENAME mf_personnel
```

(continued on next page)

Example 4–3 (Cont.) Writing SQL Modules That Use Distributed Transactions to Modify Databases

```
------------------------------------------------------------------------
-- Procedure section
------------------------------------------------------------------------

-- Start a distributed transaction using two databases.

PROCEDURE START_TRANS
     SQLCODE;

     SET TRANSACTION ON PERS USING (READ WRITE)
                 AND ON MF_PERS USING (READ WRITE);

-- Modify the personnel database.  Change a department name from
-- "Personnel Hiring" to "Personnel Recruiting."

PROCEDURE UPDATE_PERS
     SQLCODE;

     UPDATE PERS.DEPARTMENTS
            SET DEPARTMENT_NAME ='Personnel Recruiting'
            WHERE DEPARTMENT_CODE ='PHRN';

-- Modify the mf_personnel database.  Change a department name from
-- "Personnel Hiring" to "Personnel Recruiting."

PROCEDURE UPDATE_MFPERS
     SQLCODE;

     UPDATE MF_PERS.DEPARTMENTS
            SET DEPARTMENT_NAME ='Personnel Recruiting'
            WHERE DEPARTMENT_CODE ='PHRN';
```

When you run this application, the participants in the distributed transaction take the following actions:

1. The application calls the SYS$START_TRANS system service.
2. The coordinator (DECdtm services) generates a unique distributed TID so that it can keep track of the transaction and its participants. The coordinator returns the distributed TID to the application.
3. The application starts the transaction using the SET TRANSACTION statement, which specifies the databases that are part of the distributed transaction. The host language program passes the context structure by reference to the SQL module procedure.

4. The coordinator makes sure that each transaction participant knows about certain other transaction participants and communicates with those participants.
5. The application ends the transaction by calling the SYS$END_TRANS system service. However, the application could end the transaction by calling the SYS$ABORT_TRANS system service.
   - If the application calls SYS$END_TRANS, the coordinator initiates the prepare phase. If all participants vote yes, the transaction enters the commit phase and the resource managers commit the transaction. If any of the participants votes no, the coordinator instructs the participants to roll back all changes made to the databases during the distributed transaction.
   - If the application calls SYS$ABORT_TRANS, the coordinator instructs the participants to roll back all changes made to the databases during the distributed transaction.

4.4.4.2 Using Default Transaction Support with SQL Module Language

You can specify whether or not SQL automatically uses the default distributed transaction identifier (TID) established by the DECdtm system service SYS$START_TRANS. To specify that SQL automatically use the default distributed TID, your application must explicitly call the DECdtm system services and you must compile your application using the SQL module language command line qualifier TRANSACTION_DEFAULT=DISTRIBUTED.

This qualifier eliminates the need to declare context structures in host language programs and to pass the context structure to SQL module procedures. Also, it closes all cursors in the program, eliminating the need to call the sql_close_cursors( ) routine.

Example 4–4 shows how to use default transaction support to coordinate distributed transactions. It is a C host language program that calls the SQL module in Example 4–3.

Example 4–4 Writing Host Language Programs That Use Default Distributed Transactions

```
/* This program, update_dist_txn_def.c, calls procedures in the SQL module
*  update_dist_mod.sqlmod.  It takes advantage of the default distributed
*  transaction feature, which means that you do not need to declare the
*  context structure or pass the context structure to the SQL module
*  procedures.
```

(continued on next page)

Example 4–4 (Cont.) Writing Host Language Programs That Use Default Distributed Transactions

```
*  You must compile the SQL module update_dist_mod.sqlmod with the
*  TRANSACTION_DEFAULT=DISTRIBUTED qualifier.
*
*  This program modifies the department name of one department.
*/

#include <ssdef.h>
#include <stdio.h>
#include <descrip.h>
#include <starlet.h>

main()
{

long status;
long iosb[4];
static long  tid[4];
void start_trans();
void update_pers();
void update_mfpers();

/* Declare sqlcode.*/

long sqlcode;

/* Call SYS$START_TRANS to start the transaction. */

        status = sys$start_transw(
            0, /* efn */
            0, /* flags */
            iosb, /* iosb */
            0, /* astadr */
            0, /* astprm */
            tid /* tid */
            );

        start_trans( &sqlcode);

/* Call the procedure to update a row in the personnel database. */

        update_pers( &sqlcode);

/* Call the procedure to update a row in the mf_personnel database */

        update_mfpers( &sqlcode);
```

(continued on next page)

Example 4–4 (Cont.) Writing Host Language Programs That Use Default Distributed Transactions

```
/* Call SYS$END_TRANS to end the transaction.*/

        status = sys$end_transw(
            0, /* efn */
            0, /* flags */
            iosb, /* iosb */
            0, /* astadr */
            0, /* astprm */
            tid /* tid */
            );

}
```

Compare this program to the program in Example 4–2 to see the changes made to use default distributed transactions.

The C program update_dist_txn_def.c calls the SQL module shown in Example 4–3. You must process the SQL module using the TRANSACTION_ DEFAULT=DISTRIBUTED qualifier on the SQL module command line.

4.4.5 Using Precompiled SQL with Distributed Transactions

When you use precompiled SQL, you can explicitly use the two-phase commit protocol by explicitly calling the DECdtm system services in the program as described in Section 4.4.

In addition, you must take *one* of the following actions:

- Declare and use a context structure to pass the distributed TID to SQL statements. See Section 4.4.5.1 for more information.
- Use the default transaction support for distributed transactions. See Section 4.4.5.3 for more information.

4.4.5.1 Using Context Structures with Precompiled SQL

To use context structures to pass the value of the distributed TID to SQL statements, you must take the following actions:

- Declare a context structure in the program.
- Add the USING CONTEXT clause to most *executable* SQL statements that are involved in the distributed transaction.

The context structure contains the value of the distributed TID as one of its elements. Your application must declare and initialize the context structure. When you declare the context structure, you must declare the distributed TID to be a data type that is 16 bytes. For example, if your application is written in the C language, you declare the context structure and initialize it by using the following code:

```
struct
    {
     long version;
     long type;
     long length;
     long distributed_tid[4];
     long end;
    } context_struc = {1,1,16,{0,0,0,0},0};
```

When you write applications for the Ada precompiler, you declare a context structure differently than you do in other languages. Section 4.4.5.2 describes how to declare and initialize a context structure in Ada.

In addition, your application must include a USING CONTEXT clause in most *executable* SQL statements that are involved in the distributed transaction. The USING CONTEXT clause tells SQL that the statement is part of a particular distributed transaction, which is distinguished from other transactions by its distributed TID. Section 4.4.2 lists the executable statements that do not take a context structure.

The following code shows the format of an executable SQL statement used in a distributed transaction:

```
EXEC SQL USING CONTEXT :<variable>   <embedded_SQL_statement>
```

For example, the following embedded SQL statement opens a cursor as part of a distributed transaction:

```
EXEC SQL USING CONTEXT :CONTEXT_STRUC   OPEN CURSOR1
```

See the *Oracle Rdb SQL Reference Manual* for a syntax diagram of embedded SQL statements.

Example 4–5, an excerpt from the sample Fortran program sql_dist_trans.for, shows how to declare the context structure and how to call the DECdtm system services.

Example 4–5 Declaring and Calling the DECdtm System Services in an SQL Precompiled Fortran Program

```
   .
   .
   .
        IMPLICIT NONE

        INCLUDE '($SSDEF)'
        EXTERNAL LIB$STOP,SYS$START_TRANSW,SYS$END_TRANSW
        CHARACTER main_choice*1
        LOGICAL main_exit

C Declare the variables needed for the DECdtm system services.

        INTEGER*2 IOSB(4)
        INTEGER*4 STATUS,SYS$START_TRANSW,SYS$END_TRANSW
        INTEGER*4 DIST_TID(4)

C Display the main menu and accept the user's choice.

        main_exit = .FALSE.
        DO WHILE (.NOT. main_exit)
            TYPE 1000
            ACCEPT 1001, main_choice

        IF (main_choice .EQ. '5') THEN
                main_exit = .TRUE.
        ELSE

C Invoke the SYS$START_TRANSW system service and check the value of
C the I/O status block (IOSB).

                PRINT *, 'Starting distributed transaction'
                IOSB (1) = 0
                STATUS = SYS$START_TRANSW (
        1                       %VAL(0),
        2                       %VAL(0),
        3                       IOSB,
        4                       %VAL(0),
        5                       %VAL(0),
        6                       DIST_TID)
                IF (.NOT. STATUS) THEN
                        CALL LIB$STOP (%VAL(STATUS))
                ELSE
                        STATUS = IOSB(1)
                END IF
                IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))

            IF      (main_choice .EQ. '1') THEN
                CALL add_employee(dist_tid)
            ELSE IF (main_choice .EQ. '2') THEN
                CALL transfer_west(dist_tid)
```

(continued on next page)

Example 4–5 (Cont.) Declaring and Calling the DECdtm System Services in an SQL Precompiled Fortran Program

```
            ELSE IF (main_choice .EQ. '3') THEN
                CALL transfer_east(dist_tid)
            ELSE IF (main_choice .EQ. '4') THEN
                CALL del_employee(dist_tid)
            ELSE
                CONTINUE
            END IF

C Invoke the SYS$END_TRANSW system service to end the distributed
C transaction.

                PRINT *, 'Ending distributed transaction'
                IOSB (1) = 0
                STATUS = SYS$END_TRANSW (
        1                       %VAL(0),
        2                       %VAL(0),
        3                       IOSB,
        4                       %VAL(0),
        5                       %VAL(0),
        6                       DIST_TID)
                IF (.NOT. STATUS) THEN
                        CALL LIB$STOP (%VAL(STATUS))
                ELSE
                        STATUS = IOSB(1)
                END IF
                IF (.NOT. STATUS) CALL LIB$STOP (%VAL(STATUS))

        END IF
        END DO
    .
    .
    .
```

The program sql_dist_trans.for generates a menu and calls one of several subroutines, depending upon which menu item the user selects. If the user selects menu choice 2, the program calls the precompiled SQL program transfer_west.sfo. Example 4–6 shows this program, which illustrates how to declare and use context structures in precompiled SQL.

Example 4–6 Using Context Structures in an SQL Precompiled Fortran Program

```
C This subroutine is called by the program sql_dist_trans.for.  It
C transfers an employee record from the EAST database to the WEST
C database.  Both the WEST and the EAST databases are Oracle Rdb databases.

        SUBROUTINE transfer_west(in_tid)

        IMPLICIT NONE
        EXEC SQL INCLUDE SQLCA
        CHARACTER employee_id*5,last_name*14,first_name*10,
        1       middle_initial*1,address_data_1*25,address_data_2*20,
        2       city*20,state*2,postal_code*5,ascii_date*23,
        3       sex*1,status_code*1,birthday*8

C Declare the distributed TID.

        INTEGER*4 in_tid(4)

C Declare the context structure.

        STRUCTURE /CONTXT_STRUCT/
                INTEGER*4 VERSION
                INTEGER*4 TYPE
                INTEGER*4 LENGTH
                INTEGER*4 tid(4)
                INTEGER*4 END
        END STRUCTURE
        RECORD /CONTXT_STRUCT/ context

C Initialize the context structure.

        context.version = 1
        context.type = 1
        context.length = 16
        context.tid(1) = in_tid(1)
        context.tid(2) = in_tid(2)
        context.tid(3) = in_tid(3)
        context.tid(4) = in_tid(4)
        context.end = 0

C Declare the databases using aliases.

        EXEC SQL DECLARE east ALIAS FILENAME 2pceast
                IF (SQLCOD .LT. 0) CALL sql_dist_trans_error(in_tid,sqlcod)
        EXEC SQL DECLARE west ALIAS FILENAME 2pcwest
                IF (SQLCOD .LT. 0) CALL sql_dist_trans_error(in_tid,sqlcod)
```

(continued on next page)

Example 4–6 (Cont.) Using Context Structures in an SQL Precompiled Fortran Program

```
C Prompt the user for input and accept the input.

        PRINT *, '  '
        TYPE 100
100     FORMAT ('$',' Please enter the ID of the employee:  ')
        ACCEPT 120, employee_id
120     FORMAT (A)

C Declare the distributed transaction.

        EXEC SQL DECLARE TRANSACTION ON east USING (READ WRITE
        1             RESERVING east.EMPLOYEES, east.JOB_HISTORY,
        2                       east.SALARY_HISTORY, east.DEGREES
        3             FOR PROTECTED WRITE)
        4           AND ON west USING (READ WRITE NOWAIT
        5             RESERVING west.EMPLOYEES FOR SHARED WRITE)
                IF (SQLCOD .LT. 0) CALL sql_dist_trans_error(in_tid,sqlcod)

C Select the employee record from the EAST database. The USING CONTEXT clause
C tells SQL that the statement is part of a particular distributed
C transaction.

        PRINT *, '  Fetching the row in 2pceast database'
        EXEC SQL USING CONTEXT :context
        1        SELECT EMPLOYEE_ID, LAST_NAME, FIRST_NAME, SEX,
        2               MIDDLE_INITIAL, ADDRESS_DATA_1, ADDRESS_DATA_2,
        3               CITY, STATE, POSTAL_CODE, BIRTHDAY, STATUS_CODE
        4        INTO   :employee_id, :last_name, :first_name, :sex,
        5               :middle_initial, :address_data_1,
        6               :address_data_2, :state, :city, :postal_code,
        7               :birthday, :status_code
        8        FROM east.EMPLOYEES
        9        WHERE east.EMPLOYEES.EMPLOYEE_ID = :employee_id
                IF (SQLCOD .LT. 0) CALL sql_dist_trans_error(in_tid,sqlcod)
                IF (SQLCOD .EQ. 100) CALL sql_dist_trans_error(in_tid,sqlcod)

C Delete the employee record from the EAST database.  The USING CONTEXT
C clause tells SQL that the statement is part of a particular distributed
C transaction.

        PRINT *, '  Deleting the row in 2pceast database'
        EXEC SQL USING CONTEXT :context
        1       DELETE FROM east.EMPLOYEES E WHERE
        2       E.EMPLOYEE_ID = :employee_id
                IF (SQLCOD .LT. 0) CALL sql_dist_trans_error(in_tid,sqlcod)

C Store the employee record in the WEST database.  The USING CONTEXT clause
C tells SQL that the statement is part of a particular distributed
C transaction.
```

(continued on next page)

Example 4–6 (Cont.) Using Context Structures in an SQL Precompiled Fortran Program

```
PRINT *, '  Storing the row in 2pcwest database'
EXEC SQL USING CONTEXT :context
1       INSERT INTO west.EMPLOYEES
2       (EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL,
3       ADDRESS_DATA_1, ADDRESS_DATA_2, CITY, STATE, POSTAL_CODE,
4       SEX, BIRTHDAY, STATUS_CODE)
5       VALUES
6       (:employee_id, :last_name, :first_name, :middle_initial,
7       :address_data_1, :address_data_2, :city, :state,
8       :postal_code, :sex, :birthday, :status_code)
        IF (SQLCOD .LT. 0) CALL sql_dist_trans_error(in_tid,sqlcod)

RETURN
END
```

When you run the sql_dist_trans.for application, the participants in the distributed transaction take the following actions:

1. The application calls the SYS$START_TRANSW system service.
2. The coordinator (DECdtm services) generates a unique distributed TID so that it can keep track of the transaction and its participants. The coordinator returns the distributed TID to the application.
3. The application declares the transaction using the statement DECLARE TRANSACTION, which specifies the databases that are part of the distributed transaction. Note that because the DECLARE TRANSACTION statement is not an executable SQL statement, you do not specify the USING CONTEXT clause with it.
4. The application starts the transaction by issuing an executable SQL statement, in this case, a SELECT statement. SQL knows that the statement is part of a particular distributed transaction because the statement includes the USING CONTEXT clause.
5. The coordinator makes sure that each transaction participant knows about certain other transaction participants and communicates with those participants.

6. The application ends the transaction by calling the SYS$END_TRANSW system service. However, the application could end the transaction by calling the SYS$ABORT_TRANS system service.
   - If the application calls SYS$END_TRANS, the coordinator initiates the prepare phase. If all participants vote yes, the transaction enters the commit phase and the resource managers commit the transaction. If any of the participants votes no, the coordinator instructs the participants to roll back all changes made to the databases during the distributed transaction.
   - If the application calls SYS$ABORT_TRANS, the coordinator instructs the participants to roll back all changes made to the databases during the distributed transaction.

The programs sql_dist_trans.for and transfer_west.sfo are part of a sample application that demonstrates the use of distributed transactions. The source files for the programs are provided in the sample directory.

4.4.5.2 Declaring the Context Structure in Ada

When you write applications for the Ada precompiler, you should declare a context structure by declaring a variable of data type SQLCONTEXT_REC, instead of declaring a structure. When you declare a variable with the data type SQLCONTEXT_REC, the Ada precompiler generates a context structure for you. For example, you declare the variable using the following code:

```
context_struc  sqlcontext_rec;
```

The following example shows how you initialize the context structure created by the Ada precompiler:

```
context_struc.sqlcontext_ver := 1;
context_struc.sqlcontext_tid.sqlcontext_tid_type := 1;
context_struc.sqlcontext_tid.sqlcontext_tid_len := 16;
context_struc.sqlcontext_tid.sqlcontext_tid_value(1) := 0;
context_struc.sqlcontext_tid.sqlcontext_tid_value(2) := 0;
context_struc.sqlcontext_tid.sqlcontext_tid_value(3) := 0;
context_struc.sqlcontext_tid.sqlcontext_tid_value(4) := 0;
context_struc.sqlcontext_end := 0;
```

4.4.5.3 Using Default Transaction Support with Precompiled SQL

You can specify whether or not SQL automatically uses the default distributed transaction identifier (TID) established by the DECdtm system service SYS$START_TRANS. To specify that SQL automatically use the default distributed TID, your application must explicitly call the DECdtm system services and you must compile your application using the SQL precompiler command line qualifier SQLOPTIONS=(TRANSACTION_DEFAULT=DISTRIBUTED).

This qualifier eliminates the need to declare context structures in SQL precompiled programs and to pass the context structure to embedded SQL statements. Also, it closes all cursors in the program, eliminating the need to call the sql_close_cursors( ) routine.

Example 4–7 shows how to use default transaction support to coordinate distributed transactions.

Example 4–7 Writing SQL Precompiled Programs That Use Default Distributed Transactions

```
/* This program, update_dist_txn_def.sc, takes advantage of the default
*  distributed transaction feature, which means that you do not need to
*  declare the context structure or pass the context structure to embedded
*  SQL statements.
*
*  You must use the SQLOPTIONS = (TRANSACTION_DEFAULT=DISTRIBUTED) qualifier
*  when you precompile this program.
*
*  This program modifies the department name of one department.
*/

#include <ssdef.h>
#include <stdio.h>
#include <descrip.h>
#include <starlet.h>

main()
{

long status;
long iosb[4];
static long  tid[4];
void start_trans();
void update_pers();
void update_mfpers();
```

(continued on next page)

Example 4–7 (Cont.) Writing SQL Precompiled Programs That Use Default Distributed Transactions

```
/* Declare sqlcode.*/

long SQLCODE;

/* Declare the databases. */

EXEC SQL DECLARE PERS ALIAS FOR FILENAME personnel;
EXEC SQL DECLARE MF_PERS ALIAS FOR FILENAME mf_personnel;

/* Call SYS$START_TRANS to start the transaction. */

        status = sys$start_transw(
            0, /* efn */
            0, /* flags */
            iosb, /* iosb */
            0, /* astadr */
            0, /* astprm */
            tid /* tid */
            );

/*Start a distributed transaction using two databases. */

EXEC SQL SET TRANSACTION ON PERS USING (READ WRITE)
                AND ON MF_PERS USING (READ WRITE);

/* Update a row in the personnel database. */

EXEC SQL UPDATE PERS.DEPARTMENTS
               SET DEPARTMENT_NAME ='Personnel Recruiting'
               WHERE DEPARTMENT_CODE ='PHRN';

/* Update a row in the mf_personnel database */

EXEC SQL UPDATE MF_PERS.DEPARTMENTS
              SET DEPARTMENT_NAME ='Personnel Recruiting'
              WHERE DEPARTMENT_CODE ='PHRN';

/* Call SYS$END_TRANS to end the transaction.*/

        status = sys$end_transw(
            0, /* efn */
            0, /* flags */
            iosb, /* iosb */
            0, /* astadr */
            0, /* astprm */
            tid /* tid */
            );
}
```

4.5 Compiling, Linking, and Running the Sample Distributed Transaction Application

The sample application, sql_dist_trans, located in the sample directory, uses distributed transactions to update a company's databases. This application updates the following databases: two Oracle Rdb databases, 2PCEAST and 2PCWEST, which contain information about the employees in each of the company's divisions, and an Oracle CODASYL DBMS database, 2PCDBMS, which contains information about all employees in the company.

The program sql_dist_trans is the driver program that displays a menu and calls other routines to update a company's database. The sql_dist_trans source code lists the other files used in the sample application.

To use this application, Oracle Rdb, DECdtm software, and Oracle CODASYL DBMS V4.2 or higher must be running on your system. To compile, link, and run the Fortran version of the sample application, take the following steps:

1. Create the sample personnel database. Assign it the logical name 2PCEAST.
2. Create a copy of the personnel database in a different directory or on a different node. Assign it the logical name 2PCWEST.
3. Create the Oracle CODASYL DBMS sample PARTS database and assign it the logical name 2PCDBMS. (See the Oracle CODASYL DBMS documentation for information about how to create the sample PARTS database.)
4. Compile the Fortran programs (file type .for) with the Fortran compiler.
5. Compile the embedded SQL programs (file type .sfo) with the SQL precompiler. For example, use the following command to precompile transfer_west.sfo:

```
$ SQLPRE  :== $SQL$PRE
$ SQLPRE
INPUT FILE> transfer_west /FORTRAN
```

6. Compile the SQL module, sql_dist_trans_for.sqlmod, with the SQL module processor. Use the CONTEXT qualifier as the following example shows:

```
$ SQLMOD :== $SQL$MOD
$ SQLMOD
INPUT FILE> sql_dist_trans_for /CONTEXT=ALL
```

7. Link all the programs.

8. Run the application:

```
$ RUN sql_dist_trans
```

This application displays a menu and calls subroutines to add or delete employee records from the databases or to transfer employee records from one division to another.

This chapter described how to use distributed transactions with the SQL module language and precompiled SQL. Chapter 5 describes how to complete unresolved transactions using Oracle RMU commands.

5

Completing Unresolved Transactions

Before you read this chapter, you should read Chapter 1 and Chapter 2. These chapters introduce the two-phase commit protocol and explain how distributed transactions work.

The two-phase commit protocol ensures that all databases involved in a distributed transaction will commit or that none of them will commit. If a process or image failure occurs *before* all of the participants vote, the coordinator instructs each participant to roll back all changes made to the databases. However, if an unexpected failure occurs *after* all the participants vote, as soon as it is able to, the coordinator instructs the participants to roll back the changes if any participant voted no or to commit the changes if all participants voted yes.

On rare occasions, an unexpected failure may occur that causes the coordinator to be unreachable for an extended period of time. For example, the network connection may be lost. While the coordinator is unreachable, the distributed transaction cannot complete.

As Section 2.3 explains, when a failure occurs, the database recovery (DBR) process tries to resolve the transaction. When the DBR process finds a prepare record in the .ruj file, it knows that it must receive instructions from the coordinator. The DBR process must wait until it can reach the coordinator before it can complete the transaction. If the transaction has not completed, users cannot access any databases affected by the incomplete transaction. If you cannot wait for the DBR process to be able to reach the coordinator, Oracle Rdb provides a way to manually force the transaction to commit or roll back by letting the DBR process proceed without instructions from the coordinator.

If you choose to circumvent the two-phase commit process by manually completing the transactions, you must complete the transaction in the same manner for all of the databases involved in the distributed transaction. That is, you must either commit all changes or you must roll back all changes. To be able to manually complete unresolved transactions in a timely fashion, you should identify all databases that are involved in distributed transactions

before you run an application. Early planning and preparation are necessary to ensure that the transactions are resolved consistently across all databases.

Note that if the coordinating node fails so that its databases cannot be accessed, manually completing the transaction on the other nodes could lead to inconsistent results because you cannot know whether the databases on the coordinating node committed or rolled back the transaction.

This chapter explains:

- The RMU commands you use to manually complete unresolved transactions
- How to complete unresolved distributed transactions
- How to complete unresolved distributed transactions when a database becomes corrupted

5.1 Using Oracle RMU to Complete Unresolved Transactions

Oracle RMU, the Oracle Rdb management utility, provides the following commands to help you resolve distributed transactions:

- RMU Dump Users State=Blocked

  Lists unresolved transactions for a database.

- RMU Resolve

  Completes an unresolved transaction by altering the state of the transaction. You can specify the State qualifier with one of the following options:

  - Commit

    Commits any unresolved transactions on a particular database.

  - Abort

    Aborts any unresolved transactions on a particular database.

  If you do not specify the State qualifier, Oracle RMU prompts you to enter an action—Commit, Abort, or Ignore—for each unresolved transaction on that database. If you enter the Ignore option, Oracle RMU does nothing. The transaction remains unresolved until the coordinator again becomes available and instructs the DBR process to complete the transaction or you manually complete the transaction by using the RMU Resolve command again.

- RMU Dump After_Journal State=Prepared

  Lists unresolved transactions for a database by reading the .aij file.

- RMU Recover Resolve

  Recovers a database and completes an unresolved transaction by altering the state of the transaction. You can specify the State qualifier with one of the following options:

  - Commit

    Recovers the database and commits any unresolved transactions on that database.

  - Abort

    Recovers the database and aborts any unresolved transactions on that database.

  - Ignore

    Does not resolve the transaction. When you use this option, Oracle RMU attempts to contact the coordinator to resolve the transaction. The transaction remains unresolved until the coordinator again becomes available and instructs the DBR process to complete the transaction or you manually complete the transaction by using the RMU Recover Resolve command again.

  If you do not specify the State qualifier, Oracle RMU prompts you to enter an action for each unresolved transaction on that database.

For more information about these RMU commands, see the *Oracle RMU Reference Manual*.

5.2 Manually Completing Unresolved Transactions

To manually complete an unresolved transaction, take the following steps:

1. Use the State=Blocked qualifier of the RMU Dump Users command. This command generates a list of unresolved transactions for a particular database. Depending upon your application, you might need to generate the list for more than one database.

   The following example generates a list of all unresolved transactions for the personnel database:

```
$ RMU/DUMP/USERS /STATE=BLOCKED  personnel
Blocked user with process ID 000000C5
    Stream ID is 1
    Monitor ID is 1
    Transaction ID is 1
    Recovery journal filename is "DISK1:[USERA]PERSONNEL$0094FF5D19B7E2A0.RUJ;1"
    Transaction sequence number is 104
    DECdtm TID is  0019F490 0019F494 0019F498 0019F49C
    Failure occurred on node DBTWO
    Parent node is DBONE
```

2. Examine the list to identify transactions that need to be resolved. (Remember that a database can be involved in more than one unresolved transaction.) Note the transaction sequence number associated with these transactions.

   In the example shown in Step 1, the personnel database is in a **blocked** state; that is, changes have neither committed nor rolled back and the database is involved in an unresolved distributed transaction. The transaction sequence number is 104. If the database is involved in more than one unresolved transaction, the command would generate output for each unresolved transaction.

3. Refer to your application to determine which databases are affected by the unresolved transactions.

4. Consult with the database administrators for all of the databases affected by the unresolved transactions to determine how to resolve them. Collaboration is necessary because the transaction might have completed on one node before the coordinator became unreachable to the other nodes. When this occurs, the transaction must be altered to the same state on all nodes affected by the transaction.

5. Use the RMU Resolve command on the database to complete the unresolved transactions.

   For example, to complete the unresolved transactions for the personnel database, and confirm and log your action, enter the following command:

```
$ RMU/RESOLVE/LOG/CONFIRM  personnel
Blocked user with process ID 000000C5
    Stream ID is 1
    Monitor ID is 1
    Transaction ID is 1
    Recovery journal filename is "DISK1:[USERA]PERSONNEL$0094FF5D19B7E2A0.RUJ;1"
    Transaction sequence number is 104
    DECdtm TID is  0019F028 0019F02C 0019F030 0019F034
    Failure occurred on node DBTWO
    Parent node is DBONE
```

```
Do you wish to COMMIT/ABORT/IGNORE this transaction: COMMIT
Do you really want to COMMIT this transaction? [N]: Y
%RMU-I-LOGRESOLVE, blocked transaction with TSN 104 committed
```

If the database is involved in more than one unresolved transactions, Oracle RMU asks what action you want to take for each transaction.

If you want to commit or abort *all* unresolved transactions for the database, you can specify the action in the command line by using the State qualifier. For example, to commit all the unresolved transactions involving the personnel database, use the following command:

```
$ RMU/RESOLVE/CONFIRM/LOG /STATE=COMMIT  personnel
```

6. Check to see that all the transactions are resolved:

```
$ RMU/DUMP/USERS/STATE=BLOCKED personnel
No blocked users
```

Section 5.3 describes the actions you can take if a distributed transaction is unresolved and your databases have become corrupted.

5.3 Recovering Corrupted Databases and Completing Unresolved Transactions

If your database is corrupted and contains unresolved distributed transactions, you must recover the database and complete the unresolved transactions. Oracle RMU provides commands to complete these unresolved transactions by modifying their transaction records in the .aij file.

To manually recover a corrupted database and complete an unresolved distributed transaction, take the following steps:

1. Use the State=Prepared qualifier of the RMU Dump After_Journal command to generate a list of records associated with unresolved transactions in the .aij file. Depending upon your application, you might need to generate the list for more than one database.

   The following command generates this list:

```
$ RMU/DUMP/AFTER/STATE=PREPARED personnel.aij
*-------------------------------------------------------------------------------
* Oracle Rdb V6.0-00                                    11-OCT-1995 09:26:37.25
*
* Dump of After Image Journal
*     Filename: DISK1:[USERA]PERSONNEL.AIJ;1
*
*-------------------------------------------------------------------------------

1/1             TYPE=O, LENGTH=510, TAD=11-OCT-1995 08:32:17.59
    Database DISK1:[USERA]PERSONNEL.RDB;3
    Database timestamp is 19-AUG-1995 20:29:14.01
    Facility is "RDMSAIJ ", Version is 601.0
```

```
    AIJ Sequence Number is 0
    Last Commit TSN is 80
    Synchronization TSN is 0
    Type is Normal (unoptimized)
    Open mode is Initial
    Backup type is Active
    I/O format is Record

4/8 1     TYPE=V, LENGTH=229, TAD=11-OCT-1995 09:20:21.98
   2  TSN=104

000000007E47E4D100973DA0B8681680        TID: '..h¸.=..ÑäG~....'
000000007F850E9C009652D836ECCAA0        TM LOG_ID: '.Êì6ØR.........'
00000000000000000000000120400964        RM LOG_ID: 'd.@ ............'
00305359535F5453455544244520200DD        RM_NAME: 'Ý..RDBTEST_SYS0.'
000000000001000100000080A180000         RM_NAME: '................'
                      4555474F52        NODE NAME: 'DBTWO'
                    414D47495337        PARENT NODE NAME: 'DBONE'
```

2. Examine the list to identify transaction records that need to be resolved.

   1 TYPE=V tells you that the resource manager voted and is prepared to commit the changes to the database.

   2 The transaction sequence number is 104.

3. Refer to your application to determine which databases are involved in these transactions.

4. Consult with the database administrators for all of the databases affected by the unresolved transactions to determine how to resolve them. Collaboration is necessary because the transaction might have completed on one node before the coordinator became unreachable to the other nodes. When this occurs, the transaction must be altered to the same state on all nodes affected by the transaction.

5. Restore the database using the latest backup file (file type .rbf). Use the Noaij_Options qualifier to retain the current journaling configuration. Moreover, you should use the Norecovery qualifier to prevent RMU Restore from using automatic recovery mode. Automatic recovery mode does not allow you to specify specific State options when you subsequently use the RMU Recover Resolve command.

   For example, to restore the personnel database, use the following command:

```
$ RMU/RESTORE /DIR=DISK1:[USERA] /NOAIJ_OPTIONS /NORECOVERY -
_$ /NOCDD /NEW_VERSION /LOG personnel
%RMU-I-RESTXT_04, Thread 1 uses devices DISK1:
%RMU-I-AIJRSTBEG, Restoring after-image journal "state" information
%RMU-W-PREVACL, Restoring the root ACL over a pre-existing ACL.
       This is a normal condition if you are using the CDO utility.
```

```
   .
   .
   .
%RMU-I-AIJRECFUL, Recovery of the entire database starts with AIJ file sequence 0
%RMU-I-AIJRECBEG, recovering after-image journal "state" information
%RMU-I-AIJRSTAVL, 0 after-image journals available for use
%RMU-I-LOGMODFLG,     disabled after-image journaling
%RMU-I-AIJRECEND, after-image journal "state" recovery complete
```

If you have not deleted the database, you must use the New_Version qualifier as shown in the previous example, and you must specify the database root file in the RMU Recover Resolve command.

6. Use the RMU Dump After_Journal End=1 command to match the .aij file sequence number as displayed by the RMU Restore output with an .aij file, as the following example shows:

```
$ RMU/DUMP/AFTER_JOURNAL/END=1 personnel.aij
*-------------------------------------------------------------------------------
* Oracle Rdb V6.0-00                                    11-OCT-1995 09:38:30.70
*
* Dump of After Image Journal
*     Filename: DISK1:[USERA]PERSONNEL.AIJ;1
*
*-------------------------------------------------------------------------------
1/1             TYPE=0, LENGTH=510, TAD=11-OCT-1995 08:32:17.59
    Database DISK1:[USERA]PERSONNEL.RDB;1
    Database timestamp is 19-AUG-1995 20:29:14.01
    Facility is "RDMSAIJ ", Version is 601.0
    AIJ Sequence Number is 0
   .
   .
   .
```

The preceding example shows that the .aij file sequence number of 0 corresponds to the personnel.aij;1 file.

7. Use the RMU Recover Resolve command to roll forward from the correct .aij file. For example, to recover the personnel database and resolve the distributed transaction, enter the following command:

```
$ RMU/RECOVER/RESOLVE/LOG personnel.aij;1 /ROOT=personnel.rdb;
%RMU-I-LOGRECDB, recovering database file DISK1:[USERA]PERSONNEL.RDB;4
%RMU-I-LOGOPNAIJ, opened journal file DISK1:[USERA]PERSONNEL.AIJ;1
%RMU-I-AIJONEDONE, AIJ file sequence 0 roll-forward operations completed
%RMU-I-LOGRECOVR, 4 transactions committed
%RMU-I-LOGRECOVR, 7 transactions rolled back
%RMU-I-LOGRECOVR, 0 transactions ignored
%RMU-I-AIJACTIVE, 1 active transaction not yet committed or aborted
%RMU-I-LOGRECSTAT, transaction with TSN 104 is active
%RMU-I-AIJPREPARE, 1 of the active transaction prepared but not yet committed
or aborted
%RMU-I-AIJSUCCES, database recovery completed successfully
%RMU-I-AIJNXTSEQ, to continue this AIJ file recovery, the sequence number
needed will be 1
```

```
_AIJ_file: Return
    TSN=104

000000007E47E4D100973DA0B8681680        TID: '..h¸.=..ÑäG~....'
000000007F850E9C009652D836ECCAA0        TM LOG_ID: '.Êì6ØR.........'
00000000000000000000000120400964        RM LOG_ID: 'd.@ ............'
00305359535F5453455442445202000DD       RM_NAME: 'Ý..RDBTEST_SYS0.'
00000000000100010000000080A180000       RM_NAME: '................'
                      4555474F52        NODE NAME: 'DBTWO'
                    414D47495337        PARENT NODE NAME: 'DBONE'

Do you wish to COMMIT/ABORT/IGNORE this transaction: COMMIT
%RMU-I-AIJALLDONE, AIJ roll-forward operations completed
%RMU-I-LOGSUMMARY, total 1 transaction committed
%RMU-I-LOGSUMMARY, total 0 transactions rolled back
%RMU-I-LOGSUMMARY, total 0 transactions ignored
%RMU-I-AIJSUCCES, database recovery completed successfully
%RMU-I-AIJFNLSEQ, to start another AIJ file recovery, the sequence number
needed will be 1
```

In the previous example, because there are no other .aij files to roll forward, the user presses the Return key at the .aij file name prompt (_AIJ_file:). If there are more .aij files to roll forward, enter the file name, including the version number for that .aij file. For more information about rolling forward and determining transaction sequence numbers for .aij files, see the *Oracle Rdb Guide to Database Maintenance*.

If the database is involved in more than one unresolved transaction, Oracle RMU asks you what action you want to take on the other transactions.

If you want to commit, abort or ignore *all* transactions involving this database, you can specify the action with the State qualifier in the command line. For example, to commit all unresolved transactions involving the personnel database, use the following command:

```
$ RMU/RECOVER/RESOLVE /LOG /STATE=COMMIT personnel.aij;1
```

For more information about recovering corrupted databases, see the *Oracle Rdb Guide to Database Maintenance*.

This chapter explained how to complete unresolved distributed transactions and how to complete unresolved transactions that involve databases that have become corrupted.

A

Troubleshooting Distributed Transactions

When you use distributed transactions, you must ensure that DECdtm software, Oracle Rdb, and your OpenVMS system are set up correctly. If your distributed transactions access databases on remote nodes, you must set up Oracle Rdb and the OpenVMS systems to allow remote access. The *Oracle Rdb Installation and Configuration Guide* describes how to set up the RDB$REMOTE account, the RDBSERVER network object, and proxy accounts. In addition, it describes possible problems in using remote databases and solutions to those problems.

This appendix summarizes problems that are unique to distributed transactions and suggests solutions to those problems. Table A–1 shows some of the error messages you may encounter when using distributed transactions. It also describes possible reasons for the error and actions you can take to solve the problem.

Table A–1 Troubleshooting Distributed Transactions

Error	Problem	Solution
RDB$_SYS_REQUEST_CAL and RDB$_DECDTMERR	DECdtm system service error. DECdtm may not be running.	Make sure DECdtm is running. Make sure the DECdtm log file is started. Deassign the logical name SYS$DECDTM_INHIBIT. Deassign the logical name SQL$DISABLE_CONTEXT. (See Section 4.2.)
RDB$_NODISTBATU or SQL$_NOBATCHUPDATE	You cannot use a distributed transaction with batch-update mode.	Use a non-distributed transaction. Define the logical name SQL$DISABLE_CONTEXT as "True." (See Section 4.2.)

(continued on next page)

Table A–1 (Cont.) Troubleshooting Distributed Transactions

Error	Problem	Solution
RDB$_STRANSNAL	You cannot use a distributed transaction with a product that does not support DECdtm services.	Use a non-distributed transaction.
RDB$_DISTABORT	Transaction rolled back.	Test for secondary error. See Section 4.3.1.
RDB$_IO_ERROR	Network link fails.	Roll back transaction and disconnect from databases. See Section 4.3.1.
RDB$_NO_PRIV	User has no privilege to access the database using a distributed transaction.	Grant the DISTRIBTRAN privilege to the user for all databases involved in the distributed transaction.
SQL$_NO_TXN	You cannot use COMMIT or ROLLBACK statements when you explicitly call SYS$START_TRANS.	Use SYS$END_TRANS or SYS$ABORT_TRANS.
RDB$_EXCESS_TRANS	Normally an application error.	Make sure the application ends each transaction prior to starting another.
	Can be caused by long commit processing in a remote database.	Adjust the transaction start wait time. (See Section 3.7.)

It is important to understand that, under certain circumstances, Oracle Rdb starts a non-distributed transaction even though your application tried to start a distributed transaction. Oracle Rdb starts a non-distributed transaction under the following circumstances:

- The SQL$DISABLE_CONTEXT logical name is defined as "True" at compile time and you try to start an implicit distributed transaction.

  To start a distributed transaction, deassign the logical name SQL$DISABLE_CONTEXT and compile the program again. For more information about the logical name, see Section 4.2.

- You try to start an implicit distributed transaction using the SET TRANSACTION statement and specifying batch-update mode.

  You cannot use a distributed transaction with batch-update mode. Define the logical name SQL$DISABLE_CONTEXT as "True." For more information about the logical name and batch-update transactions, see Section 4.2.

- You try to start an implicit distributed transaction with an Oracle Rdb database and another database management product that requires that you explicitly call DECdtm services.

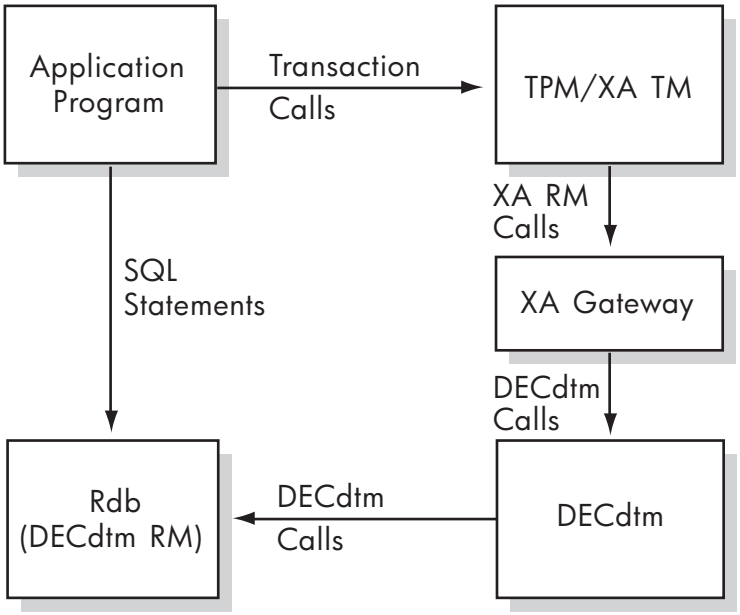
  Oracle Rdb starts a non-distributed transaction involving only the Oracle Rdb database. Use an explicit distributed transaction in this situation. See Section 4.4 for information about explicit distributed transactions.

B

Using Oracle Rdb with the DECdtm XA Gateway

The DECdtm XA Gateway is a component of DECdtm Version 2.0 and later which provides an interface between an XA Transaction Manager (TM) and a Resource Manager (RM) managed by DECdtm. It was introduced in OpenVMS 7.3-1 and can be used with Oracle Rdb release 7.1 and later as described in this appendix. The XA Gateway allows an Oracle Rdb database to participate in a transaction managed by XA. It appears as an XA resource manager to the XA TM and as a DECdtm Coordinator to the node's DECdtm TM. This is shown graphically in Figure B–1.

Figure B–1 Oracle Rdb Database in a Transaction Managed by XA



The XA TM would typically be provided by an Application Server or Teleprocessing Monitor (TPM) which provides an interface to an application to call an XA RM. The DECdtm XA Gateway is configured to be visible to the TPM's XA TM as the desired XA resource. The XA Gateway translates between the XA protocol and the DECdtm protocol and makes DECdtm-related system calls to the node's DECdtm service. DECdtm, in turn, calls the Oracle Rdb RM AST entry points using the default DECdtm transaction. The application communicates via SQL statements to Oracle Rdb using any of the normal interfaces.

In order to develop and run an application in this environment, use the following techniques:

- Precompile your application with SQL$PRE or SQL$MOD using the SQLOPTION of TRANSACTION_DEFAULT=DISTRIBUTED. This will cause Oracle Rdb to automatically use distributed transactions and to join the default DECdtm transaction for the process. See Section 4.4.4.2 and Section 4.4.5.3 for more information on this SQLOPTION and implicit distributed transactions. The DECdtm XA Gateway manages transactions using the DECdtm default transaction for the application process.

- Develop your application using the interface provided by your TPM or Application Server to control the transaction boundaries. Do not submit transaction-related SQL statements such as ROLLBACK or COMMIT to Oracle Rdb. It is possible to use DECLARE TRANSACTION to establish the characteristics of a transaction such as READ WRITE or to specify locking attributes. You must not use the BATCH UPDATE clause in the DECLARE TRANSACTION statement. See Appendix A for tips on troubleshooting distributed transactions.

  Because your application will be using implicit distributed transactions, it must not call DECdtm system services (e.g., SYS$START_TRANS) explicitly and must not pass a context structure with any SQL statements.

  If your Application Server or TPM requires you to use a precompiler instead of providing a direct call interface, you may want to package your SQL interface into SQL Module Language modules in order to prevent conflicts between precompilers. This allows you to precompile the main portion of your application with the Application Server of TPM precompiler and interact with Oracle Rdb via a procedural call interface.

- Make sure that the DECdtm XA Gateway Server (DDTM$XG_SS.EXE) is installed and use the XA Gateway Control Program (XGCP) to create an XA domain log and start up the XA Gateway Server. See the *OpenVMS System Management Utilities Reference Manual* for information on how to use the XA Gateway Control Program. See the *OpenVMS System Managers Manual* for more information on managing distributed transactions.

- Configure your Application Server or TPM to recognize the XA domain associated with the XA Gateway according to the vendor's directions.

Index

D

E

I

L

M

T

U

V

W