Module 3

Module 3 Study Guide and Deliverables

Readings: Required Reading:

- CB6 chapter 22 - Transaction Management, pages 619-661 (through 22.3)
- CB6 chapter 24 - Distributed DB concepts, pages 734-778 (please see below for sections which are secondary)
- CB6 chapter 25 - Advanced Distributed DB concepts, Distributed Deadlock management, Failures of distributed environment, 2 phase commit, pages 789-800
- CB6 chapter 26 - Replication and mobile databases, pages 827-840, Replication intro and Mobile databases, pages 267-868

Recommend Reading:

- CB6 chapter 22 - Transaction Management, pages 661-674 (especially section on backup and recovery; 22.5.4 might be relevant)
- CB6 chapter 24.2 - Overview of netowrking (if you are familiar with it - might be worth a scan)
- CB6 chapter 25 - Review 3 phase commit specifically
- CB6 chapter 26 - scan through issues with mobile databases; and 26.5 Oracle replication is a

	good case study of Replication in Oracle specifically • Looney chapter 25 - disucsses database links for distributed databases and examples of location transparency
Assignments:	• Assignments 3.0 and 3.1 due Wednesday, July 29 at 5:00 PM ET • Programming Part 3 due Sunday, August 2 at 6:00 AM ET
Assessments:	Quiz 3 due Wednesday, July 29 at 5:00 PM ET
Term Project Note:	Term Project Update #3, is due Sunday, August 2 at 6:00 AM ET. This term project deliverable may include an update of your project plan or any other portions of your term project. The purpose of this deliverable is to provide your instructor with an opportunity to guide you midway in your term project.
Live Classrooms:	Supplementary Live Session, Wednesday, July 22, 8:00 PM- 10:00 PM ET Current week's assignment review and examples, Thursday, July 23, 8:00 PM - 9:00 PM ET Live office help, Saturday, July 25, 11:00 AM - 12:00 PM ET Live office help, Monday, July 26, 8:00 PM - 8:45 PM ET

Introduction

metcs779_wk3 video cannot be displayed here

Distributed database management systems (DDBMS) are database systems where several (two or more) **database instances** work together to form one logical database. DDBMS are widely used in large geographically distributed organizations, for a number of reasons:

- By distributing the data close to the locations where it is used most, performance is improved, network traffic is reduced, and the risks associated with network outage are reduced.
- DDBMS can be very fault-tolerant. For example, DDBMS can be designed so that critical data is not lost even if a site is destroyed and conventional recovery for that site fails.
- DDBMS can be designed so that they scale well. For example, with a distributed enterprise database design, bringing up a new regional office may only involve installing a new regional database server in the office and making it a part of the enterprise DDBMS.

Definition of a Database Instance

A computer, DBMS software, and associated durable storage that implement a database.

Learning Outcomes

By reading this lesson, participating in discussions, and completing the assignments, you will be able to:

- Describe what a distributed database management system (DDBMS) is, what its components are, and how it works.
- Describe how transactions are managed in a DDBMS.
- Describe two-phase commit, how it works, and why it is important.
- Describe how data are partitioned, replicated, and allocated in distributed database design.
- Describe how distributed database performance, fault tolerance, and network traffic are affected by different design choices.
- Understand distributed database designs that you encounter.
- Design efficient reliable, scalable Distributed Database Management Systems.
- Describe the different levels of transparency in a DDBMS and explain why transparency is important.
- Describe federated and tiered database designs and why they are used.

Lecture 3A - Transaction Management

Overview

This lecture introduces transactions, transaction management, and database concurrency control, including serializability, locking methods, and deadlocks. We then cover the basics of recovering a database, including the need for transaction logging. Finally, we discuss the basic transaction and recovery features that are available in Oracle.

Transaction Management is introduced in Chapter 22 of the sixth edition of Connolly and Begg. You are responsible for the material in this lecture and in Connolly and Begg. You are not responsible for the Oracle-specific techniques, but you may find some of them useful for your term project.

Learning Objectives

By reading this lecture, participating in discussions, and completing the assignments, you will be able to:

- Explain and use the ACIDS properties of transactions.
- Explain why database concurrency control is important and choose appropriate concurrency control for applications.
- Explain the different types of locks and how they are used in databases.
- Describe deadlocks—what causes them, how they affect database applications, and how to prevent deadlock problems.
- Describe how transaction logs are produced and their roles in database recovery.

Properties of Transactions

One of the most important aspects of a transactional database is how the database manages its transactions. Both concurrency control and recovery are required to support high transaction rates and protect the database from inconsistencies and data loss. If a database doesn't have concurrency control it can't support concurrent transactions, greatly reducing database transaction throughput. Without database recovery, a database can't be restored to a consistent state after a transaction is aborted or the database is corrupted by software error, human error, or hardware failure.

A transaction is any action or series of actions carried out by a single user or application program that reads or updates the contents of the database. Transactions sound simple, but they are rather complex and have several properties that you must understand. A transaction can have two different outcomes—committed or aborted. If the transaction is committed by the user or application using the COMMIT command, then the transaction is said to be *committed* and the database is in a consistent state where all database constraints are satisfied. If the transaction does not execute successfully the transaction is *aborted*, and then the database must be restored to the consistent state that it was in before the transaction was started. A transaction can be explicitly aborted by the user or application using the SQL ROLLBACK command. A transaction can also be aborted by the DBMS for reasons such as the unavailability of a lock needed by the transaction. When the DBMS aborts a transaction it normally automatically restarts the transaction later. This restarting of an aborted transaction is normally implemented within the database kernel where it is invisible to the user.

Below is an example of a simple transaction:

```
UPDATE CUSTOMER SET CUSTOMER_FNAME='JIM'
WHERE CUSTOMER_ID=12232
COMMIT;
```

In most DBMS, and in the ANSI and ISO SQL standards, a transaction begins implicitly when a connection is established to the database and each subsequent COMMIT or ROLLBACK begins a new transaction. In other DBMS such as Microsoft SQL Server there would be the line BEGIN TRANSACTION at the top of this example. The COMMIT statement above makes the update of the name permanent. If the last line had been ROLLBACK then the transaction would have been aborted and the database would not have been changed.

In a fully transactional database transactions should possess the five basic **ACIDS** properties:

Atomicity requires that all modifications must follow an “all or nothing” rule. This means that when part of a transaction fails, then the whole transaction fails so that the user doesn’t have to worry about incomplete transactions or inconsistent data.

Consistency guarantees that a transaction will transform the database from one consistent state to another consistent state. This means that the database will enforce consistency through constraints according to the database schema.

Isolation ensures that each transaction operates independently from other transactions. This is possible through locking and concurrency control in the DBMS.

Durability requires that committed transactions are permanently recorded in the database and won’t be lost due to failures such as system crashes.

Serializability makes sure that it appears as if the requests in transactions are executed in the order that the user or application presents them to the DBMS, with no intervening transactions, even though the requests may actually be executed in a different order with intervening transactions.

Concurrency Control

Imagine many people trying to access and update a spreadsheet at once. Even if everyone can obtain a copy of the spreadsheet from a shared file system, when the different users save their changes it will wipe out changes made by other concurrent users. How can you make sure people aren’t updating the same thing at the same time? How can you make sure that the data the person is reading is up-to-date? Concurrency control is a name for the functionality that permits multiple users to read and update the same set of data while ensuring that everyone sees a consistent view of the data and that no one's updates are lost. A spreadsheet does not have concurrency control, but a database does and can assure that numerous users can access and manipulate the same set of data. Concurrency control is obviously very useful. It is also a little complicated.

The major advantage of using a database to manage your data is the ability to have many users access shared data concurrently. With concurrency control the database assures that the data is always consistent. The basic definition of *concurrency control* is the process of managing simultaneous operations on the database without having them interfere with one another. Concurrency control is necessary to prevent problems that can cause incorrect functioning of applications, including:

Lost update problem—a lost update is where a completed update operation by one user can be overwritten by another user’s update operation.

Uncommitted dependency problem—this occurs when one transaction is allowed to see the intermediate results of another transaction before it has committed.

Inconsistent analysis problem—this occurs when a transaction reads several values from the database but a second transaction updates some of them during the execution of the first.

Serializability

The main objective of serializability is to find nonserial schedules that allow transactions to execute concurrently without interfering with one another and thereby produce a database state that could be produced by a serial execution.

Serializability is possible with the help of the concurrency control's scheduling, where transactions are executed by a particular schedule. Here are some definitions:

Schedule—a schedule is a sequence of operations by a set of concurrent transactions. The goal of a schedule is that it functions in the same way for all transactions as a serial schedule.

Serial schedule—a serial schedule is a schedule where the operations of each transaction are executed consecutively without any interleaved operations from other transactions. A serial schedule is the definition of functionally correct concurrency.

Nonserial schedule—a nonserial schedule is a schedule where the operations from a set of concurrent transactions are interleaved. A nonserial schedule may or may not be functionally equivalent to a serial schedule.

There are two main concurrency control mechanisms that allow a DBMS to perform transactions safely in parallel—locking and timestamping.

Test Yourself

Serializability finds non-consecutive ways to execute transactions to produce a database state that could be produced serially. Why would it be a problem to execute transactions on a serial schedule? (Select all that apply.)

With a serial schedule, each transaction would wait for the previous one to end before starting. Waiting for each transaction to complete would have a performance impact.

This is true. With a serial schedule transactions proceed in a given order from start to finish.

With a serial schedule, concurrent transactions would not update the database in the correct order.

This is false. Transactions update the database correctly, since one transaction doesn't start until the previous one finishes.

With a serial schedule, there could be too many concurrent transactions that would overload the database.

This is true. Transactions appear to progress consecutively, which is the best for accuracy, but too many concurrent transactions could cause a serious performance hit.

Locking Methods

Locking is one of the main practices for concurrency control. Locking ensures data integrity. When one transaction is accessing the database, a lock may deny access to other transactions to ensure correct results. For example, look at the query below:

```
BEGIN TRANSACTION
UPDATE CUSTOMER SET CUSTOMER_NAME='JIM'
WHERE CUSTOMER_ID=12232
--COMMIT/ROLLBACK
```

Since we have not committed the transaction yet and since this is an update statement, there will be an exclusive lock on the row where CUSTOMER_ID=12232. If an application or user tried to access the row they would not be able to, because of the lock. If a transaction has an *exclusive lock* then it can read or update the item, and no other transaction can access it.

Once this query is committed another transaction can occur on the item For example: let's assume that the previous query was committed and the query below is executed.

```
SELECT * FROM CUSTOMER
WHERE CUSTOMER_ID=12232
```

The query will have a shared lock on the item because it is only reading the data. If a transaction has a *shared lock* on an item, it can only read the item. Once an item has a shared lock, it can only be read. An exclusive lock cannot be obtained on a shared lock to update the data because of the shared lock. The locking ensures the data integrity so that users don't read incorrect data.

Test Yourself

If user A owns an exclusive lock on the CUSTOMER table, and user B wants to read that table, which kind of lock (exclusive or shared) would user B be able to obtain for his read of the CUSTOMER table?

Exclusive lock

This is false. User B would not be able to get an exclusive lock since user A owns an exclusive lock on the CUSTOMER table.

Shared lock

This is false. User B would not be able to get a shared lock on CUSTOMER since user A owns an exclusive lock on that table.

No lock

This is true. User B would not be able to obtain any locks on the CUSTOMER table until user A released the exclusive lock.

Deadlocks and Timestamps

Deadlocks occur when two or more transactions are each waiting for locks to be released that are held by the other. For example, suppose that Session A holds lock X and is trying to procure lock Y. Suppose that session B holds lock Y and is trying to procure lock X. Neither can proceed. They are deadlocked until something breaks the deadlock. That something is the DBMS. One simple way that DBMS detect deadlocks is when a query waits too long for a lock. A DBA can specify the query timeout dependent upon the type of query.

Once the deadlock is identified the DBA must decide which transaction to abort. Usually there will be one transaction causing a block of numerous transactions so it is obvious that the transaction causing the block should be aborted. But at other times it is hard to tell which transaction to stop, so the DBA will need to do some research in order to find which transaction to kill.

There are numerous things to take into consideration: How long has the transaction been running? How has the deadlock affected the performance of the database? How much data is the transaction going to update? How far do you have to roll back the transaction without having an effect on the database? Also, it is important to look at trends. If a particular query is causing numerous deadlocks, maybe the query needs to be changed or altered.

Test Yourself

If user A owns a shared lock on the CUSTOMER table, and user B wants to update that table, which kind of lock (exclusive or shared) would user B be able to obtain?

Exclusive lock

This is false. User B would not be able to get an exclusive lock since user A owns a shared lock on the CUSTOMER table.

Shared lock

This is true. The only lock available to user B would be a shared lock. Once user A released the shared lock, user B would be able to obtain an exclusive lock.

No lock

This is false. User B could get a shared lock to read the same data on the same table as User A.

Timestamps

Timestamping is a concurrency control mechanism that orders transactions in such a way that older transactions (transactions with smaller timestamps) get priority in the event of conflict. Timestamping is sufficient to assure correct concurrency control, but timestamping is not used without locks, because concurrency control based on timestamps alone results in a very slow database with much less concurrency than if locks were also used.

Recovering Databases

Knowing how to recover a database from an inconsistent state is very important. Sometimes a business will be inoperable if a database is down. For example, airlines can’t fly airplanes and banks can’t conduct business without databases. Every minute the database is offline, Amazon will be losing money. It is important to know how to recover the database and have a strategy or plan to get the database back online as soon as possible. Without the right configuration of database backups, there might be the possibility of data loss.

Transactions play a critical role in recovering a database. The transaction log contains all the database changes since the last backup. The content of a transaction log is different between different DBMS, but a transaction log generally contains the following information:

- A transaction identifier

- A before-image of the data item
- An after-image of the data item
- Type of log record
- Checkpoint information

The transaction log file is used for database recovery, performance, monitoring, and auditing. A *checkpoint* in the log file is the point of synchronization between the database and the log file.

Oracle Concurrency and Recovery

Oracle uses optimistic concurrency control with an advanced multi-version read consistency protocol that guarantees a session sees a consistent view of the data, even though the data has been changed by another session since the transaction began. Oracle does this by maintaining "rollback segments," which store the changes to the database. Before a transaction starts, Oracle writes the before-image to a rollback segment. In that way, rollback segments preserve the original data. This permits Oracle transactions to actually change the data in the database during the transaction, before the transaction is committed. At commit time Oracle checks if the transaction violated any constraints such as locks, and if so, rolls back the transaction.

Isolation Levels

Oracle, like other DBMS, permits DBAs some control of the more expensive aspects of concurrency control. One of these controls is the ability to specify the isolation levels. Isolation levels are DBMS settings that control how transactions perform. They describe how a transaction will be isolated from other transactions.

READ COMMITTED—serialization is forced at the statement level, where each statement within a transaction sees only data that was committed before the statement started. This is the default isolation level for Oracle.

SERIALIZABLE—serialization is forced at the transaction level, so each statement within a transaction sees data with only the changes committed.

READ ONLY—read only is where transactions only see data that was committed before the transaction started.

There are several other isolation levels that are available and they are different in every DBMS, but their intentions are to control the isolation and concurrency of transactions. The isolation level in Oracle is set using the “SET TRANSACTION” or “ALTER SESSION” commands.

Test Yourself

Isolation is the transaction property that ensures that the transaction will be independent of other transactions. Select all that are true about isolation levels.

Because isolation is a transaction property, isolation levels only affect how the DBMS behaves when transactions are committed.

This is false. Isolation levels influence DBMS behavior during transactions. Examples include having SQL read uncommitted changes in the midst of a transaction for less stringent serializability settings. This may result in the same SELECT returning different results when executed at different times in a transaction, because another transaction changed data that the SELECT depended upon.

If the isolation level is set to the typical commercial DBMS default (READ COMMITTED in Oracle), data might change between repeated executions of the same statement in a transaction.

This is true, since READ COMMITTED occurs at the statement level and not at the transaction level. READ COMMITTED is the default isolation level for Oracle.

The SERIALIZABLE level will provide the most accurate data.

This is true. The SERIALIZABLE level makes it appear that transactions are run one at a time. However, while this is the best for accuracy, it is often not the best for performance.

Backup and Recovery

For backup and recovery, Oracle provides the Oracle Recovery MANager (RMAN). The recovery manager maintains backup information and has the ability to perform complete backups, incremental backups, and other types of backups, and also assists with restoring or recovering a database. RMAN logs all backup operations to the control files of the database that is being backed up. Since the DBMS has all of the information needed to determine how to recover, recovering a database can be as simple as one command:

```
RMAN> RESTORE DATABASE;
```

The backup information can also be queried and reported.

Summary

Transaction management and concurrency control are critical in database management systems; they maintain database integrity and correct operation when concurrent transactions are accessing the same data. Concurrency control makes it possible for many users to access the same database, with each user's transactions executing as if that user were the only user of the database, with no lost updates or other concurrency problems. The best concurrency control implementations permit thousands of transactions to execute concurrently, greatly improving database performance and scalability. Concurrency control implementations on modern DBMS all use timestamps and locks, but the details differ between DBMS.

Lecture 3B - Distributed Database Management Systems

DDBMS Basics

The technical definition of a DDBMS is a DBMS that supports distributed transactions. A DDBMS need not be geographically distributed. One common use of DDBMS technology is high reliability systems with two or more database servers in a fault-tolerant configuration connected by high speed local area networks.

Centralized database designs require that data be stored in a single central site, but this is not very flexible and has performance, scalability, and other limitations. The centralized database's shortcomings spawned a demand for applications based on data access from different sources at multiple locations and distributed database technology was developed to meet those needs.

DDBMS Advantages and Disadvantages—The following table summarizes the main advantages and corresponding disadvantages of DDBMS compared to centralized (single instance) DBMS.

Advantages	Disadvantages
Locating data near where they are needed	Multiple database servers and instances to maintain
Faster data access for local data, so better performance	Less central visibility and control over database accesses
Easier to manage growth and scale up and down	Need to configure distribution
Reduced risk of single point of failure with replication	Need for additional storage
Reduced ongoing operating costs	Higher initial training costs
Reduced risk of data loss	Increased risk of data theft
Faster recovery with a good design	Increased complexity of backup and recovery

Test Yourself

What are some of the advantages of a DDBMS compared to a centralized DBMS? (Choose all that are true.)

A DDBMS is easier to scale up as additional capability is needed.

This is true. If the database is well designed, it is easier to scale up by adding more database servers to provide increased capacity.

With a DDBMS, it is easier to train designers and administrators.

This is false. Additional training is needed so that designers and administrators understand how the data is distributed and replicated on different machines.

With a DDBMS, it is easier to recover data.

This is false. It is faster to recover data sometimes, but since fragments can be located on different machines, and only one fragment might fail, planning for complete recovery can become very complex.

Test Yourself

What are some of the advantages of a centralized DBMS compared to a DDBMS? (Choose all that are true.)

A centralized DBMS is easier to design than a DDBMS.

This is true. Fragmentation, allocation and replication are not factors to consider in a centralized design.

A centralized DBMS requires less maintenance than a DDBMS.

This is true. Since there is only one database on one machine a centralized database is easier to maintain.

A centralized DBMS will always have lower operating costs over the long run.

This is false. A DDBMS will have higher initial training costs, but may have lower operating costs over time, because less expensive hosts may be needed and because functionality intrinsic to DDBMS may be easier to manage and have lower costs than corresponding centralized database technology. Network costs can also be lower with a DDBMS, and the costs associated with slower user response may make centralized system costs higher.

A DDBMS must perform all of the functions of a centralized DBMS, plus all functions imposed by the distribution of the data and processing. Ideally, the DDBMS should perform these additional functions *transparently* to the end user, meaning that the DDBMS looks to the user and the application SQL as if it were a single-instance DBMS.

DDBMS are distributed systems, so they have the usual network infrastructure of distributed systems. Sometimes user workstations are associated with a DDBMS, though this isn't necessary.

Transaction processors and data processors are functionality that all DDBMS must have. The term *transaction processor* (TP) refers to the functionality that supervises distributed transactions. The term *data processor* (DP) refers to the functionality that retrieves, updates, and stores the data on the database instances that are part of the DDBMS. The data processor is usually a DBMS, often interfaced via a database driver such as ODBC, JDBC, or ADO.net. The transaction processor functionality may be implemented in a DBMS, or it may be different software, such as BEA's Tuxedo or the distributed transaction management in Microsoft's COM+.

The XA/Open Distributed Transaction Processing standard is the leading standard for the management of distributed transactions, particularly heterogeneous distributed transactions. XA distributed transaction processing is supported by Oracle and other leading DBMS, and also by J2EE and .net. The application programs may communicate using the Java Transaction API (JTA) in Java Enterprise, which is the Java implementation of the XA standard distributed transaction API.

The JTA, or Java Transaction API, is Oracle/Sun Microsystems' Java Enterprise application programming interface for distributed transactions. JTA is based on the standard XA API for distributed transactions. Java application developers can use JTA to communicate with a distributed transaction processor and use that TP service to coordinate distributed transactions on multiple DPs.

You can obtain additional information on the X/Open distributed transaction standards at http://docs.openlinksw.com/mt/xamt.html and other websites.

DDBMS Basics Continued

Distinctions between distributed processing and distributed databases—Distributed databases require distributed processing, but most distributed processing systems do not include distributed databases. The key distinguishing characteristic of distributed databases is distributed transactions. The following table summarizes one of the standard ways of classifying distributed processing and distributed database systems.

Distributed System Class	Characteristics
Single-Site Processing, Single-Site Data (SPSD)	DBMS is located on the host computer, which is accessed by terminals or terminal emulators connected to it. SPSD is mainly found today in legacy systems such as BU's UIS, and in ERP systems, such as BUWorks.
Multiple-Site Processing, Single-Site Data (MPSD)	Multiple processes run on different computers sharing a single database instance. Many two-tier applications, with PCs and other clients directly connected to a database server, are in this class, as are most n-tier application systems with a single DBMS in the database tier.
Federated systems	Multiple-site processing, with multiple-site data, but without distributed transactions. These architectures are fairly common in large data warehouses, where distributed transactions are not required.
True DDBMS	Fully distributed database management systems with support for transaction processing, with multiple data processors and distributed transactions.

Homogenous and heterogeneous DDBMS - DDBMS can be classified as either homogeneous or heterogeneous. Homogeneous DDBMS integrate only one type of DBMS such as Oracle or DB2. Heterogeneous DDBMS include DBMS from different vendors.

DDBMS transparency is important in simplifying the user's model of the DDBMS and in improving maintainability of the SQL code that accesses the DDBMS. The following table summarizes the different transparency features.

Transparency Feature	What this Feature Hides and Does
Distribution transparency	The fact that the data are distributed
Transaction transparency	That distributed transactions are involved; also ensures that database transactions will maintain database integrity
Failure transparency	That an instance or other distributed database component has failed
Performance transparency	That the DDBMS does not perform as fast as a single-instance DBMS
Heterogeneity transparency	That the DDBMS is heterogeneous

Test Yourself

With heterogeneity transparency, a user could be connected to an Oracle DBMS instance which conducts distributed transactions with a DB2 instance and it would seem to the user that they were using the same type of database. True or False?

This is true, and is the definition of heterogeneity transparency.

Distribution transparency has three distinct levels, which are summarized in the following table:

Level of Distribution Transparency	Comments	Example SQL
Fragmentation transparency (highest level)	Hides both data fragmentation and data location. Neither fragments nor locations are required in SQL. Strongly preferred for maintainability of SQL, because SQL does not change when data are refragmented or reallocated.	`select * from customer`
Location transparency (medium level)	Hides the instances (locations) to which the data is allocated, but does not hide the fragmentation. SQL changes may be required when data is refragmented.	`Select * from BOS_CUS UNION select * from NYC_CUS`
Local mapping transparency (lowest level)	Both fragment name and location are required in the SQL, so SQL changes are required when data is refragmented or relocated. Synonyms (or views) and links alone can provide location transparency, so this level should not normally be used.	`Select * from BOS_Link.BOS_CUS UNION select * from NYC_Link.NYC_CUS`

As a general rule you should use the highest level of distribution transparency supported by the TPs and DPs that you are using. The main problem with using less than full distribution transparency is that then your SQL depends on the details of the distribution design, which can change over time. For example, if your implementation has only location transparency then adding a new instance to the distributed database would require changing all of the enterprise-level selects to add unions for the additional logical tables.

Test Yourself

Select all that are true of distribution transparency:

Fragmentation transparency eliminates SQL changes due to data location changes.

This is true. Fragmentation transparency is the highest transparency level and it the hides both the data fragmentation and the data location.

With local mapping transparency, the location of the data is transparent.

This is false. Local mapping transparency is the lowest level of distribution transparency, and both the location and fragment name must be provided in the SQL command.

With location transparency the fragment name must be specified, but the data instance is hidden.

This is true. Location transparency needs more than fragmentation transparency, because it requires fragment names, but it requires less than local mapping transparency because the locations of the fragments do not need to be specified.

Remote and Distributed Requests and Transactions

Remote requests and remote transactions and distributed requests and distributed transactions are all subtly different. Remote requests do not require DDBMS capability, but can be implemented by applications software without a TP. Distributed transactions require distributed ACIDS transactions and TP capabilities.

Remote requests let single SQL statements read, write, or update data on a single DP. Remote requests require a DBMS with database link capability, but do not require DDBMS capability or a TP. From the DBMS' perspective it does not matter whether there is a LAN or WAN between the client and the DBMS, so remote requests are really no different than any other requests, except that the term *remote request* connotes that the DBMS is in a different location than the computer that generates the remote request. Remote requests do not even require that the DP support transactions, so remote requests can be used with DBMS such as MySQL with the default MyISAM back end, which does not support transactions.

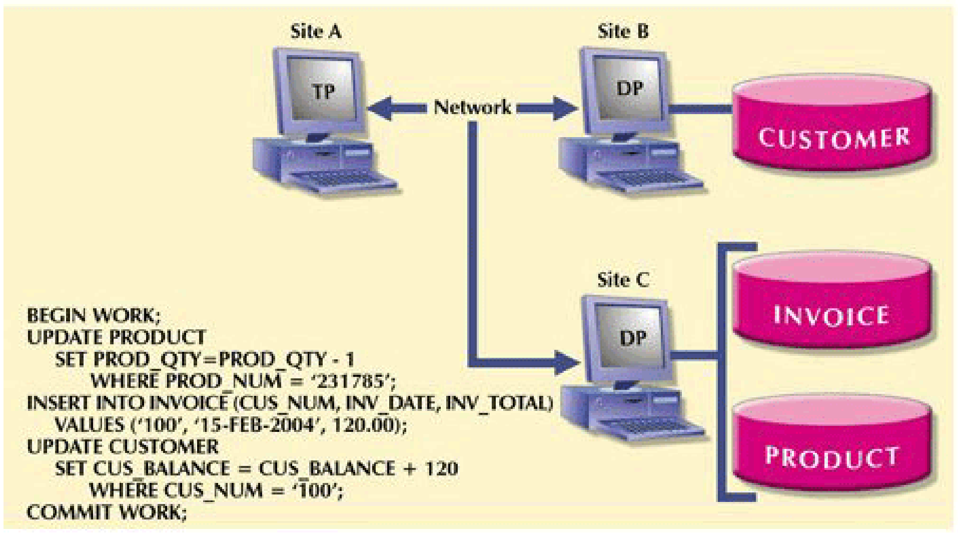
Remote transactions are no different than remote requests, except that several requests (e.g. SQL statements) may be involved. The remote DP is responsible for transactional integrity of the remote requests. All that is required for remote transactions is that the remote DP support transactions.

Transactions

Recall that transactions satisfy the ACIDS properties: Atomicity, Concurrency, Isolation, Durability, and Serializability.

Distributed transactions are transactions that require one or more DPs and a TP that together have distributed transaction capability. Support for distributed transactions is what defines a DDBMS configuration, so without distributed transaction support a distributed system is not a distributed database system. The following figure from our MET CS669 text illustrates a distributed transaction. Note that the distributed transaction involves updating the customer, invoice, and product tables. Requests are part of the atomic transaction, even though the tables are allocated to different DPs.

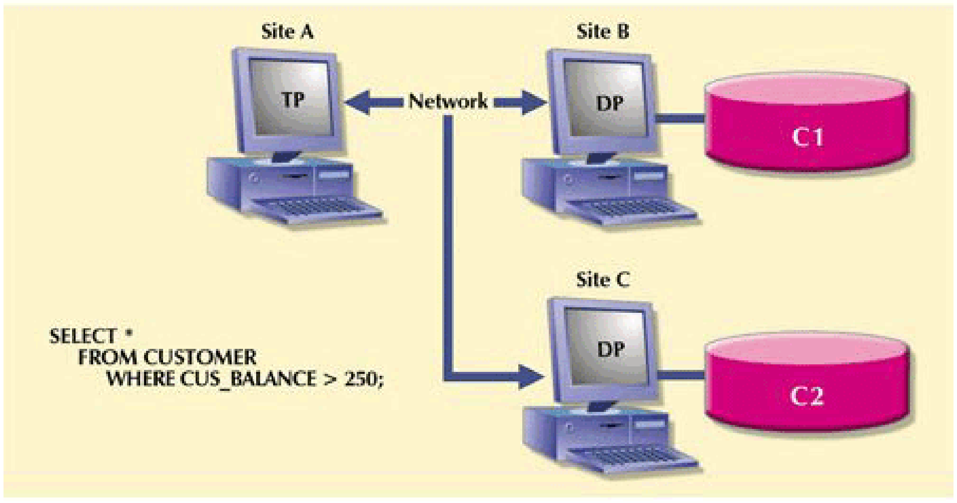
A Distributed Transaction



```
BEGIN WORK;
UPDATE PRODUCT
    SET PROD_QTY=PROD_QTY - 1
        WHERE PROD_NUM = '231785';
INSERT INTO INVOICE (CUS_NUM, INV_DATE, INV_TOTAL)
    VALUES ('100', '15-FEB-2004', 120.00);
UPDATE CUSTOMER
    SET CUS_BALANCE = CUS_BALANCE + 120
        WHERE CUS_NUM = '100';
COMMIT WORK;
```

In the example above, the database tables are allocated to different DPs, but the tables are not fragmented. Consider the example below, where the customer table has been horizontally fragmented into tables C1 and C2 and those tables have been allocated to different instances.

Another Distributed Request



```
SELECT *
    FROM CUSTOMER
        WHERE CUS_BALANCE > 250;
```

In this example full fragmentation transparency has allowed us to refer to C1 and C2 using the logical table name CUSTOMER.

Test Yourself

Select all that are true of remote and distributed requests and transactions:

A remote request requires a transaction processor so that the local DBMS can communicate with the remote DBMS.

This is false. A remote request requires a data processor, but not a transaction processor.

Remote requests and remote transactions are similar, the difference being that remote requests process one SQL command at a time, and remote transactions process blocks of multiple SQL statements that are part of the same transaction.

This is true.

Distributed transactions require at least one data processor and multiple transaction processors.

This is false. A distributed transaction requires at least one transaction processor and can have any number of data processors.

Why do DDBMS need distributed requests and distributed transactions? We might think that we could just use two SELECTs in the application, one to access C1 and another to access C2, and union the result sets, without needing DDBMS capability. Unfortunately this approach might result in inconsistencies, because the states of C1 and C2 may correspond to the final states of different transactions. For example, suppose that our application queries C1, then queries C2, and suppose another session is transferring a customer from C1 to C2 by deleting the customer from C1 and then adding that same customer to C2. If the customer is deleted from C1 before the SELECT against C1 and the customer has not yet been added to C2 before the SELECT against C2, then the customer will not appear in the result set of the two application-level SELECTs, even though the customer should have been in the result set. This simple example points out why it is essential that all transactions against distributed databases (both the two selects and the moving of the customer) be atomic distributed transactions. In other words, distributed requests are similar to distributed transactions, in the sense that they correspond to one transactional state of the entire distributed database. The difference between distributed requests and distributed transactions is that distributed requests leave the distributed database in the same transactional state, while distributed transactions change that distributed transactional state.

Distributed Concurrency Control

In distributed database systems concurrency control is more complex than in single-instance systems and lock contention is more important. This section describes the techniques used for concurrency control in DDBMS.

The distributed concurrency problem—Suppose that a TP or application sends parts of a distributed transaction to two instances, which we call A and B. Suppose that instance A successfully commits the transaction. Suppose further that instance B cannot commit the transaction—for example, because the data violates an integrity constraint in B. Then the TP or application has a database integrity problem that it cannot solve, because to solve it would require either rolling back the committed transaction at A or somehow forcing a commit at B. What is needed is some way to roll back the committed transaction A and/or some way to retry the failed transaction at B. What is needed is a more general implementation of transactions that handles the situation when not all DPs are able to commit. This generalization is provided in two-phase commit protocols.

Three Phase Commit

Two-phase commit protocols can block, for example if a DP times out after sending the commit to the TP but before receiving the final commit from the TP. Blocking is rare enough that most systems use two-phase commit. There is a non-blocking generalization of two-phase commit which introduces a pre-committed state in both the TP and DP. You are not responsible for three-phase commit in this class other than being aware that it exists. The interested reader can consult an excellent treatment in Connolly and Begg pp. 797–800.

Two-phase commit protocols permit any part of a distributed transaction to be undone and retried in the event that not all parts of the distributed transaction can be successfully committed.

In order to support the ability to roll back committed transactions DPs in distributed database systems must write their transaction logs to durable storage before updating the durable representations of the tables, so that both the old and new versions are in durable storage. Two-phase commit protocols also support efficient retrying of portions of the transaction and other recovery mechanisms.

Coordinator and subordinates—Two-phase commit protocols involve communications between the transaction *coordinator* or TP, and one or more *subordinates*, or DPs. In these protocols the coordinator is the client and the subordinates are the servers. The coordinator has the overall responsibility for managing and finally committing the transaction. The final commit cannot be issued until all subordinates have completed their parts of the transaction. The selection of the coordinator may depend on where the transaction is initiated; for example, an application may request a distributed transaction be implemented by a particular TP. The particular subordinates involved in a transaction are the DPs that have been allocated fragments involved in the transaction.

Test Yourself

In the two-phase commit protocol, the transaction processor acts as the server, and the data processors act as clients. True or False?

This is false. The transaction processor acts as the client, and the data processors act as servers.

DO-UNDO_REDO protocols—In two-phase commit, the TP communicates with the DP in terms of DO-UNDO-REDO protocols. The following table summarizes the three parts of this protocol.

Action/Name	What the DP does when the TP requests the action	When it is done
DO	The DP writes the before and after values to the transaction log and then performs the requested operation.	By each involved DP at the beginning of a distributed transaction
UNDO	The DP uses the transaction log entries to rollback the operation.	After a distributed transaction failure, to restore the previous consistent state
REDO	The DP reads the transaction log and retries the operation.	To recover by retrying the transaction

The Preparation and Final Commit Phases—As the name suggests, the two-phase commit protocol is divided into two distinct phases. The first phase is the preparation phase and the second the commit phase. The two phases are described in the following steps:

Two-phase Commit Protocol	

Phase One: The Preparation Phase	Phase Two: The Final Commit
1. The coordinator sends a PREPARE TO COMMIT message to all subordinates. 2. The subordinates respond with an indication signifying whether or not they're prepared to commit. 3. If the coordinator does not receive a Yes from all subordinates, it aborts or retries the transaction, using the DO-UNDO-REDO protocol. 4. If the coordinator receives a Yes from all subordinates it proceeds to phase two. 5. If the coordinator does not receive a Yes from all subordinates after retrying it returns a distributed transaction failure error to the application.	1. The coordinator sends a COMMIT message to all subordinates. 2. Each subordinate commits the changes using the DO protocol. 3. Subordinates reply to the coordinator with a COMMITTED or NOT COMMITTED message. 4. If any subordinates are not able to commit, the coordinator sends an ABORT message to all subordinates, forcing them to UNDO all changes. The coordinator may then retry the transaction or return an error to the caller.

Distributed Query Optimization

Distributed query optimization can be complex, because the performance of a distributed query depends on network performance, replication, data statistics, and other factors. The distributed query optimizer identifies the fragments to be used in the query, selects the optimum execution order, selects the sites to be accessed to minimize communication costs, and selects the replicas to use in the query. The output of the query optimizer is the *execution plan* for the query.

The objective of query optimization is to minimize the total cost associated with the execution of a request. Costs associated with a request are a function of the:

- Total elapsed time to complete the request
- Total elapsed time to get the first parts of the result set to the user
- I/O cost
- Communication (network) cost
- CPU time cost

Replica transparency is the ability of the DDBMS to hide replication in the distributed database. The distributed query optimization must provide replica transparency. This means that the DDBMS must hide the resources required to update replicas. It also means that the DDBMS must generate distributed query plans that are functionally equivalent to plans without replication.

Distributed query optimization techniques. There are many techniques for optimizing distributed queries. These techniques can be classified along the following dimensions:

- Whether they are manual or automatic
- Whether they are static or dynamic

- Whether they are statistics-based or rule-based

Manual versus automatic query optimization techniques—Manual query optimization involves having the SQL programmer provide hints and other directives to the query optimizer to help it determine the best query plan. Manual query optimization has the disadvantage of requiring the manual query plan to be updated each time the distribution design changes or the database statistics change significantly. Automatic optimization is implemented entirely within the DDBMS. As processors become faster the cost of automatic optimization is becoming almost insignificant compared to the cost of executing distributed requests, the speed of which can be constrained by network bandwidth and delays.

Static versus dynamic query optimization techniques—Static query optimization takes place when the request is submitted to the DDBMS. The static query optimization is used for stored procedures, triggers, and methods. With dynamic query optimization the query is optimized each time it is executed. Dynamic query optimization is used when a SQL statement is sent to the DDBMS for processing, optimization and immediate execution, such as when a user or application sends a SQL statement to the DDBMS. Dynamic query optimization is not normally used with stored procedures, triggers, or methods, because the optimal query plan will change only when the database schema or statistics change. You can obtain the advantages of dynamic optimization without the overheads by encapsulating the SQL in durable objects such as stored procedures or triggers and recompiling these durable SQL statements each time the database statistics are recomputed, which is typically scheduled nightly or weekly for stable production systems.

Rule-based versus statistics-based optimization techniques—Rule-based optimizers use a hierarchy of rules to determine the query pattern to use. These rules are very general, but the rules do not take into account important details such as the size of tables and the selectivity of WHERE clauses, because these details depend on the actual data in the database. Because they do not use statistics, rule-based optimizers will occasionally produce plans that take more than an order of magnitude longer than the best plans. Statistics-based optimization algorithms require that statistics be collected on the tables and indexes. The statistics are used to predict the cost of the different alternative query plans, so these optimizers are also called *cost-based optimizers*. Statistics-based optimization generally outperforms rule-based optimization, and cost-based optimization is more robust, meaning that it does not produce substantially suboptimal plans the way that rule-based optimizers occasionally do.

Test Yourself

Select all that are true about distributed query optimization.

Manual distributed query optimization can rapidly become outdated if there are multiple changes to the database or changes to network performance.

This is true. With automatic query optimization, changes that affect the statistics or distributed design will trigger a new optimized query plan.

The goal of distributed query optimization is to provide the same level of accuracy as a local transaction.

This is false. The goal is to provide the same level of service for a distributed transaction as for a local one.

A stored procedure that selects customer records would invoke the dynamic query optimizer.

This is false. A stored procedure normally uses static query optimization. Dynamic query optimization would be used if the stored procedure is recompiled every time new statistics are available.

Lecture 3C - How to Design Distributed Databases

Data Fragmentation

In this section you will learn how to partition the database into fragments, how to determine which fragments to replicate, and how to determine where to locate those fragments and replicas.

Fragmentation breaks objects into two or more fragments stored on different instances. Information about data fragmentation is stored in the distributed data catalog (DDC), from which it is accessed by the TP to process user requests.

There are three data fragmentation strategies—horizontal fragmentation, vertical fragmentation, and mixed fragmentation, which is just a combination of horizontal and vertical fragmentation. *Horizontal fragmentation* (also called *row fragmentation*) is the division of a relation (table) into multiple subsets of tuples (rows). *Vertical fragmentation* is the division of a relation into multiple relations, often sharing some of the attributes (columns), particularly primary key columns. *Mixed fragmentation* is the division of a relation (table) into multiple relations (columns) and the division of each of those relations into multiple relations, each containing a subset of the tuples (rows). Horizontal fragmentation does not normally allow any duplication of the tuples (rows) between the fragments, while vertical fragmentation usually duplicates the primary key in each of the fragments.

Advanced Technical Detail

You may notice that horizontal fragmentation is similar to row partitioning in databases such as Oracle and DB2/UDB. Both use keys to partition a table by row. In both the partitions are separate tables that are combined into one logical table at the SQL level. The main difference is that in the fragmentation of a table in a distributed database the fragments are located on different instances, while in a row-partitioned table the partitions are located on the same instance.

Horizontal fragmentation requires a *fragmentation key* that determines which rows go in each fragment. The fragmentation key may be the primary key or any ordered set of columns. The fragmentation key does not have to be a candidate key for the table and it does not have to be unique. The columns in the fragmentation key should usually not be nullable. For example, in a distributed database design that is horizontally fragmented by geographic region the fragmentation key may be the geographic region identifier.

Fragmentation Examples

The following four figures from *Database Systems: Design, Implementation & Management* (Rob, Coronel, & Morris; Cengage Publishing) illustrate, respectively, a customer table; the customer table horizontally fragmented with the CUS_STATE as the fragmentation key; the customer table vertically fragmented into service-oriented and collection fragments, with CUS_NUM as a common column; and the customer table with mixed fragmentation horizontally by state and vertically as previously.

A Sample Customer Table

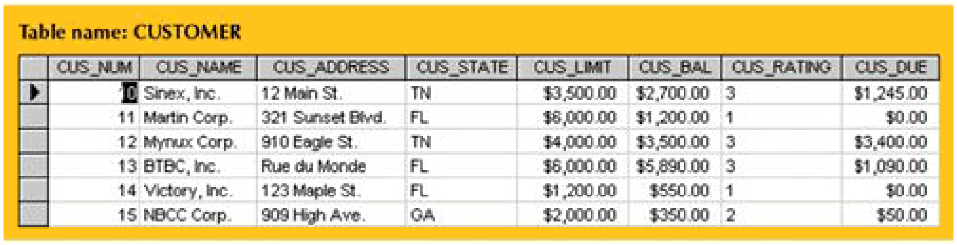


Table name: CUSTOMER

CUS_NUM	CUS_NAME	CUS_ADDRESS	CUS_STATE	CUS_LIMIT	CUS_BAL	CUS_RATING	CUS_DUE
10	Sinex, Inc.	12 Main St.	TN	$3,500.00	$2,700.00	3	$1,245.00
11	Martin Corp.	321 Sunset Blvd.	FL	$6,000.00	$1,200.00	1	$0.00
12	Mynux Corp.	910 Eagle St.	TN	$4,000.00	$3,500.00	3	$3,400.00
13	BTBC, Inc.	Rue du Monde	FL	$6,000.00	$5,890.00	3	$1,090.00
14	Victory, Inc.	123 Maple St.	FL	$1,200.00	$550.00	1	$0.00
15	NBCC Corp.	909 High Ave.	GA	$2,000.00	$350.00	2	$50.00

Horizontally Fragmented Table Contents

Table Fragments in Three Locations

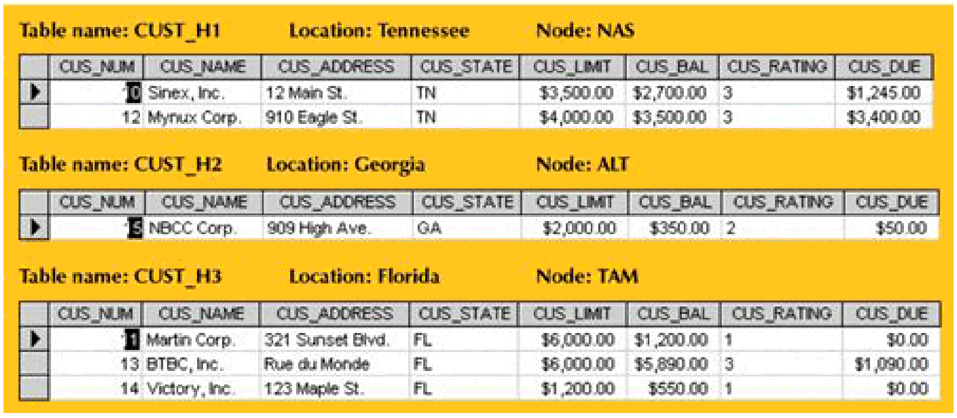


Table name: CUST_H1 Location: Tennessee Node: NAS

CUS_NUM	CUS_NAME	CUS_ADDRESS	CUS_STATE	CUS_LIMIT	CUS_BAL	CUS_RATING	CUS_DUE
10	Sinex, Inc.	12 Main St.	TN	$3,500.00	$2,700.00	3	$1,245.00
12	Mynux Corp.	910 Eagle St.	TN	$4,000.00	$3,500.00	3	$3,400.00

Table name: CUST_H2 Location: Georgia Node: ALT

CUS_NUM	CUS_NAME	CUS_ADDRESS	CUS_STATE	CUS_LIMIT	CUS_BAL	CUS_RATING	CUS_DUE
15	NBCC Corp.	909 High Ave.	GA	$2,000.00	$350.00	2	$50.00

Table name: CUST_H3 Location: Florida Node: TAM

CUS_NUM	CUS_NAME	CUS_ADDRESS	CUS_STATE	CUS_LIMIT	CUS_BAL	CUS_RATING	CUS_DUE
11	Martin Corp.	321 Sunset Blvd.	FL	$6,000.00	$1,200.00	1	$0.00
13	BTBC, Inc.	Rue du Monde	FL	$6,000.00	$5,890.00	3	$1,090.00
14	Victory, Inc.	123 Maple St.	FL	$1,200.00	$550.00	1	$0.00

Vertically Fragmented Table Contents

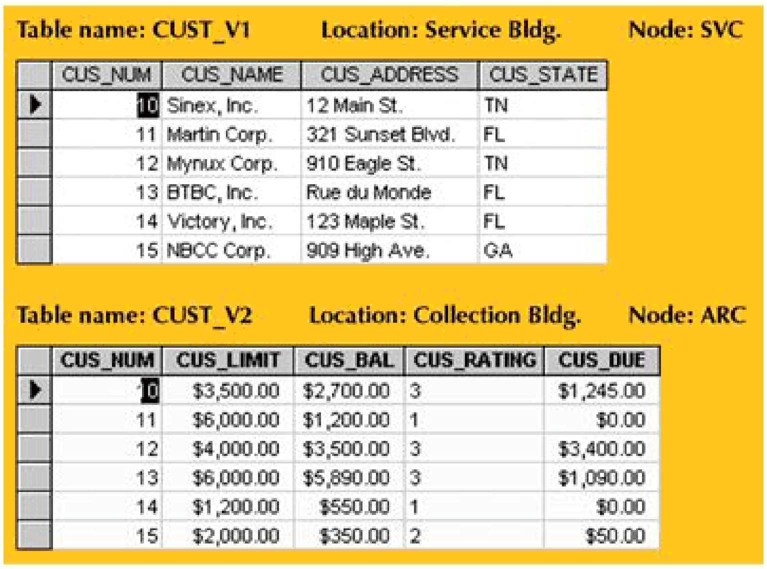


Table name: CUST_V1 Location: Service Bldg. Node: SVC

CUS_NUM	CUS_NAME	CUS_ADDRESS	CUS_STATE
10	Sinex, Inc.	12 Main St.	TN
11	Martin Corp.	321 Sunset Blvd.	FL
12	Mynux Corp.	910 Eagle St.	TN
13	BTBC, Inc.	Rue du Monde	FL
14	Victory, Inc.	123 Maple St.	FL
15	NBCC Corp.	909 High Ave.	GA

Table name: CUST_V2 Location: Collection Bldg. Node: ARC

CUS_NUM	CUS_LIMIT	CUS_BAL	CUS_RATING	CUS_DUE
10	$3,500.00	$2,700.00	3	$1,245.00
11	$6,000.00	$1,200.00	1	$0.00
12	$4,000.00	$3,500.00	3	$3,400.00
13	$6,000.00	$5,890.00	3	$1,090.00
14	$1,200.00	$550.00	1	$0.00
15	$2,000.00	$350.00	2	$50.00

Table Contents After the Mixed Fragmentation Process

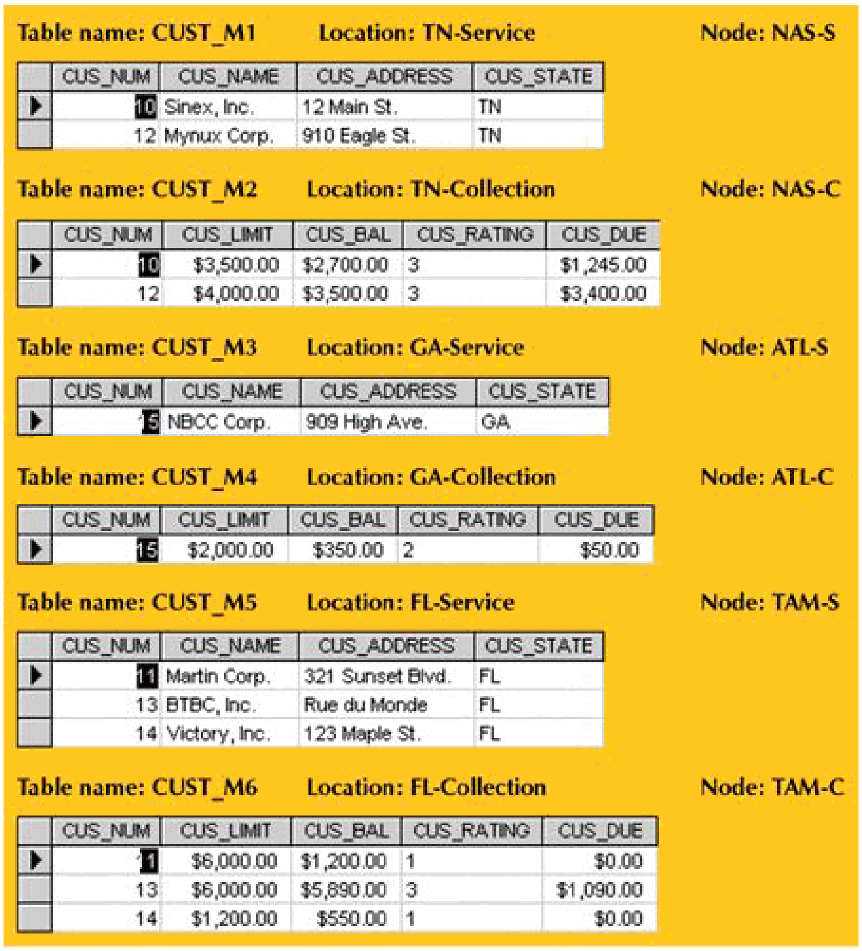


Table name: CUST_M1 Location: TN-Service Node: NAS-S

CUS_NUM	CUS_NAME	CUS_ADDRESS	CUS_STATE
10	Sinex, Inc.	12 Main St.	TN
12	Mynux Corp.	910 Eagle St.	TN

Table name: CUST_M2 Location: TN-Collection Node: NAS-C

CUS_NUM	CUS_LIMIT	CUS_BAL	CUS_RATING	CUS_DUE
10	$3,500.00	$2,700.00	3	$1,245.00
12	$4,000.00	$3,500.00	3	$3,400.00

Table name: CUST_M3 Location: GA-Service Node: ATL-S

CUS_NUM	CUS_NAME	CUS_ADDRESS	CUS_STATE
15	NBCC Corp.	909 High Ave.	GA

Table name: CUST_M4 Location: GA-Collection Node: ATL-C

CUS_NUM	CUS_LIMIT	CUS_BAL	CUS_RATING	CUS_DUE
15	$2,000.00	$350.00	2	$50.00

Table name: CUST_M5 Location: FL-Service Node: TAM-S

CUS_NUM	CUS_NAME	CUS_ADDRESS	CUS_STATE
11	Martin Corp.	321 Sunset Blvd.	FL
13	BTBC, Inc.	Rue du Monde	FL
14	Victory, Inc.	123 Maple St.	FL

Table name: CUST_M6 Location: FL-Collection Node: TAM-C

CUS_NUM	CUS_LIMIT	CUS_BAL	CUS_RATING	CUS_DUE
11	$6,000.00	$1,200.00	1	$0.00
13	$6,000.00	$5,890.00	3	$1,090.00
14	$1,200.00	$550.00	1	$0.00

Note that in this example if the database has customers from all fifty states that there would be 100 fragments. While this is not enough fragments to be a concern, the fragments would be of very different sizes, so it is not likely a good fragmentation design. A better fragmentation key would be something such as the geographic region of the enterprise, or a range of state keys for each fragment. With vertical fragmentation by state, the designer would have to address what happens when data for a new state is inserted into the database and assure that the required tables are automatically identified and created.

Test Yourself

Select all that are true about fragmentation:

A fragmentation key should not include nullable columns.

This is true. If a fragmentation key included nullable columns there would be no basis for fragmenting rows with nulls in those columns.

Mixed fragmentation involves fragmenting different tables on common fields.

This is false. Mixed fragmentation involves fragmenting the rows and columns of the same table.

The fragmentation key needs to be a non-nullable unique key so that rows or columns can be fragmented efficiently.

This is false. The key has to be non-nullable, but it does not have to be unique. For example, a state code column may be the fragmentation key for a sales order table, with many sales orders for the same states.

Replication

Data replication in a distributed database is the storage of the same data on multiple instances in the distributed database. Usually entire tables are replicated. Data replication makes the data available locally on instances, which is faster than having the data fetched over the network, so replication usually speeds queries and improves response time. Replication can also reduce communications traffic. A common use of replication in critical databases is to protect against data loss.

Replication is also commonly used with big data models, such as Hadoop, that are highly optimized for read transactions. The assumption is that hardware fails often, and with the commodity hardware that is the basis for the big data platforms, hardware failure is to be expected. Data is copied to several machines so that if one machine fails, the job can continue executing on one of the replicas. Replication also aids in performance with Hadoop and other big data model frameworks, because reads can be performed in a parallel fashion on the same data on different machines.

Fully replicated databases—Sometimes entire databases are replicated as hot backups for critical high availability systems. For example, repository banks, which keep records of securities for mutual fund companies, sometimes keep their mutual fund data in triply replicated databases. When designing fully replicated databases, consideration must be given to the amount of network traffic required to maintain the replicas and the network delays and DDBMS processing delays that this engenders. If the update rates are very high at all, it is best to connect the replicated instances via a fiber optic or other high speed, low latency network, usually a local or campus-scale network. One common full replication scenario in a geographically partitioned database is to replicate all of the data at headquarters. This approach protects against data loss and also permits the headquarters instance to efficiently analyze the data at enterprise scope.

Partially replicated databases—This is a common design for distributed databases. In partially replicated databases, some fragments may be replicated at sites where they are frequently needed. For example, infrequently updated tables such as those that represent states and countries may be replicated to all sites where they are joined. Sometimes tables that hold critical data such as accounts may be replicated to protect against data loss. When replicating frequently changing fragments be sure to estimate and measure the network traffic and performance degradation to maintain the replicas.

Unreplicated distributed databases—Replication is often not required. For example, a horizontally fragmented database may have all of the data in each instance for the geographic region that it supports, and adequate backup and recovery to protect against data loss. Distributed requests can be run against the unreplicated tables to gather data for enterprise-scale analysis or a separate data warehouse can be developed that gathers and organizes the data appropriately for analysis at the enterprise level.

Test Yourself

Select all that are true of replicated databases.

A frequently updated table in an application with many transactions is a good candidate for replication.

This is false. There is a cost associated with keeping the replicas synchronized, so frequent updates would mean all replicas must be updated as part of the same transaction. This would cause performance issues.

Replication is an essential part of a distributed database.

This is false. While it could be important for failover, or for other reasons, it is not necessary to include replication in a distributed database.

Performance always improves if the data is replicated.

This is false. Performance may improve if the data is mostly used in read-only transactions and jobs can be run in parallel on the same data, but if the replicas are updated frequently, performance may suffer.

Data Allocation

Data allocation is deciding where to locate each fragment of a distributed database. Data allocation design is closely coupled with fragmentation design, because the fragmentation design is often driven by the desire to locate data on the instances where it is used the most, to improve performance. Data allocation design is also closely coupled with replication design, because the reasons for replicating (improving performance and preventing data loss) can only be realized if the replicas are allocated appropriately.

Data allocation strategies—The simplest data allocation strategy is to allocate all of the fragments at one site or instance. This is the allocation strategy for a centralized database. Distributed database allocations are driven by and drive the fragmentation design and replication design. The general guidelines for data allocation are:

- Allocate an unreplicated fragment on the instance where it is most frequently used or where its presence will result in the best overall performance, compared to allocating it elsewhere. To perform this analysis you will need to have a good idea of the distribution of query patterns. If a fragment is frequently accessed by more than one instance separated by a wide area network consider replicating the fragment.
- Consider security implications in the data allocation decision. Avoid locating sensitive data on instances where it could be accessed by individuals or programs that should not have access to it. Collections of sensitive data such as enterprise financial data or sensitive personnel data should often be allocated to instances associated with the corresponding functional parts of the organization such as HR or finance.
- If replication is in the design to prevent loss of critical data allocate the replicas in places that are unlikely to be impacted by the same event. For example, allocate replicas in different buildings, and connect the instances by a fiber optic network.

Network Latency Slows Distributed Databases

Signal transmission latency in the network is a significant factor in the performance of distributed databases, so fiber optic connections are preferred over copper or other wire connections. The signals travel in fiber at nearly the speed of light, while in CAT5 and most other wire-based network implementations the speed of the signals is about one third of the speed of light.

Test Yourself

Select all that are true about data allocation:

Size of the data and performance would be important factors to consider for data allocation strategies for mobile users.

This is true. For mobile users, the data should be used mostly for reads, and the size should be kept to a minimum.

An excellent allocation strategy involves allocating data in different data centers in the same building.

This is false. There are many failure modes that affect multiple data centers in the same building, including power, cooling, security, and structural failures. For backup and recovery, and in the light of disasters like 9/11 or Hurricane Sandy, the best strategy may be to move the data to different buildings or different cities.

If multiple users access a single fragment in multiple locations, it is preferable to allocate the data centrally to reduce network traffic.

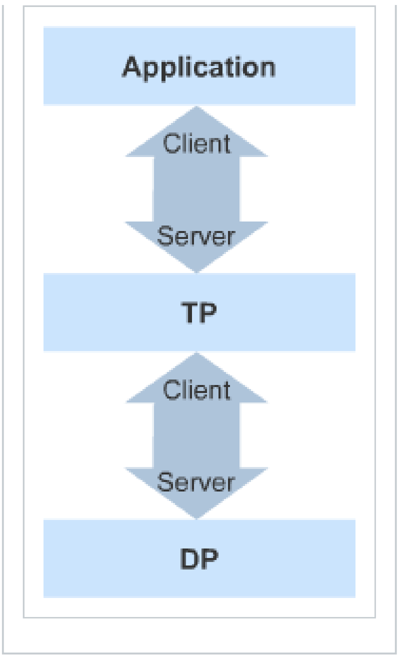
This is false. Although this could be helpful, a better allocation design may be to replicate the data fragment, and allocate it to all machines so they can use it locally. Replication would also likely reduce network traffic compared to centralization.

Lecture 3D - Client/Server relationships in DDBMS

Client/Server Relationships in DDBMS

The way that a TP controls the DPs involved in a distributed transaction is similar to (though more complex than) the way a client controls its servers in traditional client/server architectures. The service provided by the TP is management of the distributed transaction. The TP's client is the application. The DP provides the data processing services to the TP, which is the client of the DP. Thus there are two client/server relationships involving the application, TP and DP. This is illustrated in the following figure.

Two-Phase Commit Protocol



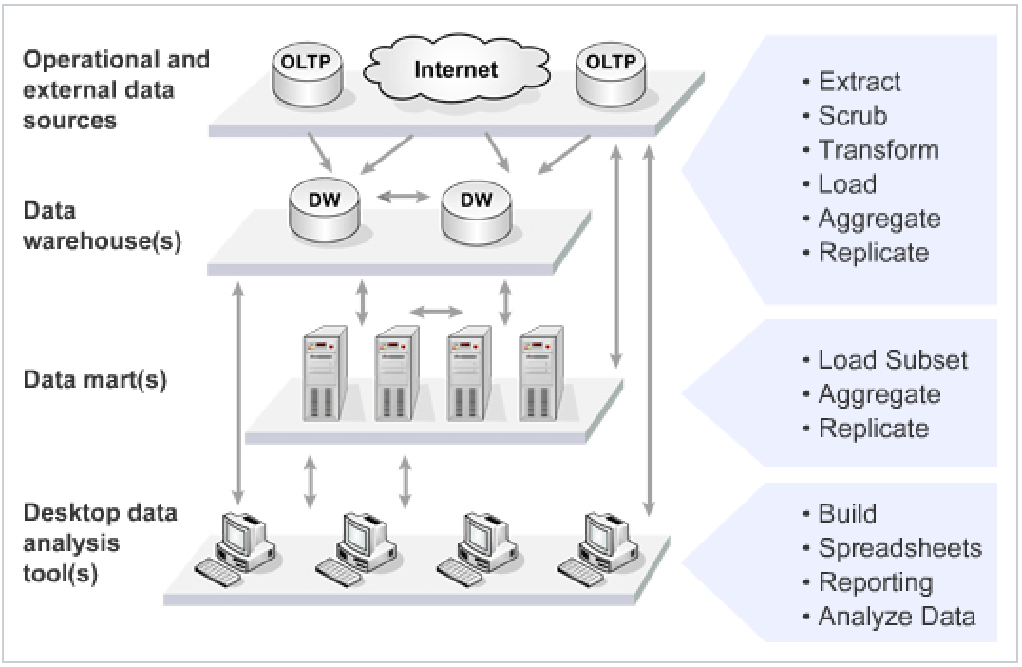
Federated Databases

Federated databases are distributed systems in which applications implement communications between databases. Federated databases do not of themselves support distributed transactions. Federated databases are occasionally used for large distributed data warehouses and other database applications that do not require distributed transactions. Some of the largest data warehouses use federated databases.

Tiered Databases

Tiered databases are hierarchically organized federated databases. Tiered database architectures are often used in larger enterprises. For example, the following figure illustrates a common tiered database architecture where data warehouse databases are tiered with data mart databases and the data marts databases extract their data from the data warehouses.

Data Warehouse Components



Summary

A distributed database stores logically related data in two or more instances (usually located in different sites) connected by computer networks. Distributed database systems support distributed transactions, while federated and tiered database systems may be linked by applications without true distributed transactions.

A distributed database is divided into fragments. The main components of a DDBMS are the transaction processor and the data processor, which may be combined. DDBMS can be described by the extent to which they support different transparencies, which means the degree to which the DDBMS behaves like a single-instance DBMS. A transaction is formed by one or more database requests. All or portions of a database can be replicated over multiple instances connected by a computer network.

Client/server architecture refers to execution control between two computers interacting over a computer network to form a system where the client initiates requests to the server and the server responds, but the server does not initiate requests to the client. In a DDBMS the application is the client of the TP, and the TP is the client of the DPs. The X/Open distributed transaction standards facilitate heterogeneous distributed database systems using a variety of application languages, TPs, and DPs.

Test Yourself

Select all that are true about distributed databases.

A database is a distributed database if it has fragments of the database located at different sites which are connected by a network and the database has transactions that involve data at multiple sites.

This is true, and is the definition of a distributed database.

A federated database processes distributed transactions on multiple computers in the same way as a distributed database.

This is false. A federated database does not process distributed transactions. In a federated database system distributed transactions are implemented externally to the federated databases via a transaction processor if they are used at all.

DDBMS transparencies help to convince the user that they’re operating on a single, centralized database instance.

This is true. This is the goal of the different kinds of transparencies that are covered in this lecture.

Lecture 3E - Replication and Mobile Databases

Overview

This lecture introduces data replication technology for mobile databases and Oracle replication. The goal of this lecture is to help you understand and be able to employ data replication and mobile databases.

We begin this lecture by introducing the basic concepts of data replication. We then move to the different types of replication and how they are used. Next, we describe the replication server, which is the software system that manages the data replication. Then, we introduce mobile databases, including the increasing demands of mobile computing, the advantages and issues. Finally, we discuss how Oracle implements replication.

Replication and mobile databases are introduced in Chapter 26 of the sixth edition of Connolly and Begg. You are responsible for the material in this lecture and in Connolly and Begg. You are not responsible for the Oracle-specific techniques, though you may find them useful for your term project.

Learning Objectives

By reading this lesson, participating in discussions, and completing the assignments, you will be able to:

- Describe the different types of replication.
- Describe replication server functions.
- Explain the demand for mobile databases.
- Describe how Oracle supports data replication.

Introduction to Data Replication

Data replication is duplicating data on one or more database instances. Data replication can serve many purposes, including:

- ***Database Availability***—A business may require a database to be operating 24/7. For example, Amazon.com or another online retailer will lose sales if they are unavailable to customers. If database replication is implemented and there is a problem on the current database, it is fairly straightforward to fail over to the replicated database, with little or no user downtime.
- ***Database Backup and Recovery***—Most businesses can't afford to lose transaction data. If a database were to become corrupted, the DBA would have to restore the database from backup, including replaying the transaction logs since the last backup. Restoring a database from a backup can take many hours or even many days, during which time the database is not available. A good way to maintain availability in spite of database instance failure or corruption is to fully replicate the database and to fail over to the replica if the primary database becomes unavailable or corrupted. Fail over to a replica can be implanted within seconds. Full replication is the most common replication design, largely because of its ability to support fail over.
- ***Scalability and Load Balancing***—Applications such as busy web sites must support very high database request rates. One possible solution is to replicate the data to several databases and have requests balanced on each database. For example, there could be three database servers replicating data to each other (3 database servers all with the same data), so that the system could allocate database requests to each server to spread the request workload. This approach works best with database workload that is mainly SELECTs, with few database changes, because of the overhead of maintaining the replicas when the data changes.

Often a database can be factored into different databases, with some data replicated and some data different for the different databases. For example, a retail web site database could have product data replicated to many servers, while the customer data could be localized to individual servers, each of which supports transactions for a subset of the customers. This replication architecture minimizes replication overhead, because the replicated data changes slowly, while supporting local joins of dynamic data, such as customer and shopping cart data, with the more static replicated data. This architecture can also be very scalable through the addition of many database instances, and it can support very high request rates. This architecture requires a means of routing requests to the appropriate servers, based on customer ID, for example.

Test Yourself

Data replication can provide better availability and faster fail over, but it may not provide easier maintenance. True or False?

This is true. While availability and faster fail over can be easily accomplished with data replication, the replicas incur more storage and can be more complex to back up and recover, so maintenance can be more complex.

Types of Data Replication

Synchronous replication—The replicated data is updated immediately when the source data is updated.

Asynchronous replication—The target database is updated after the source data has been modified.

Test Yourself

An example of synchronous replication would be running a series of batch updates at night to update a replica. True or False?

This is false. This is actually an example of asynchronous replication, since the batch updates are not performed immediately as part of transactions.

Replication Server Functionality

Replication is supported by every major database vendor. There are numerous solutions that vendors offer.

Although the main purpose of replication is to distribute the data to multiple locations, there are many other functions that need to be provided.

Scalability—The replication service should be able to handle all types of data replication. For example, a transaction may consist of adding 10 records or it may consist of updating 1,000 records.

Mapping and transformation—The replication service should be able to replicate data to different server types. For example, an Oracle database server can replicate data to a SQL Server database.

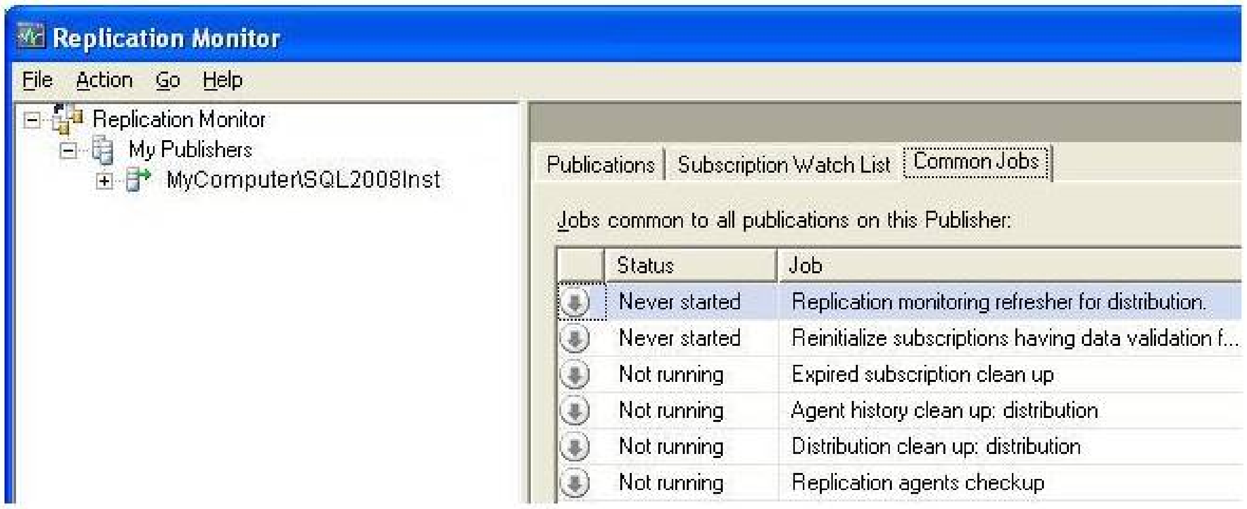
Object replication—For example, if a new table or stored procedure is created, the system should replicate it.

Specification of replication schema—The replication service should allow a user to select specific objects and/or data that is to be replicated. For example, a company may only want to replicate its purchase transactions.

Subscription mechanism—For example, the replication service should allow the setup of one or multiple servers to receive replicated data.

Easy administration—The replication service should have tools that are easy for the DBA to use to check the status and monitor the system. An example of the Microsoft SQL Server Replication Monitor tool is below.

Replication Monitor



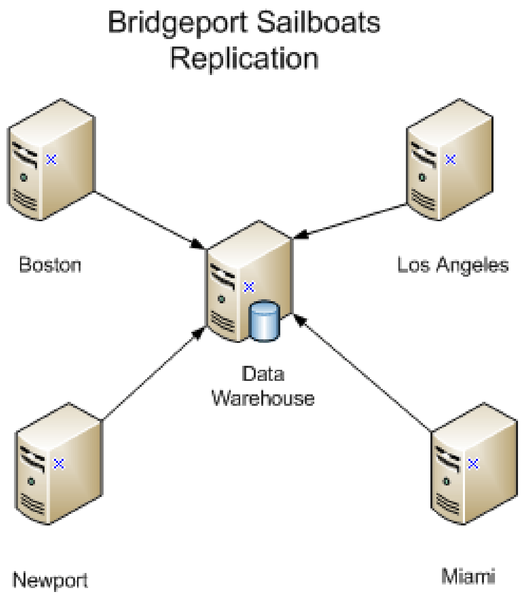
Jobs common to all publications on this Publisher:

	Status	Job
	Never started	Replication monitoring refresher for distribution.
	Never started	Reinitialize subscriptions having data validation f...
	Not running	Expired subscription clean up
	Not running	Agent history clean up: distribution
	Not running	Distribution clean up: distribution
	Not running	Replication agents checkup

Ownership

Ownership deals with what server has privileges to update which data. There are three main types of ownership.

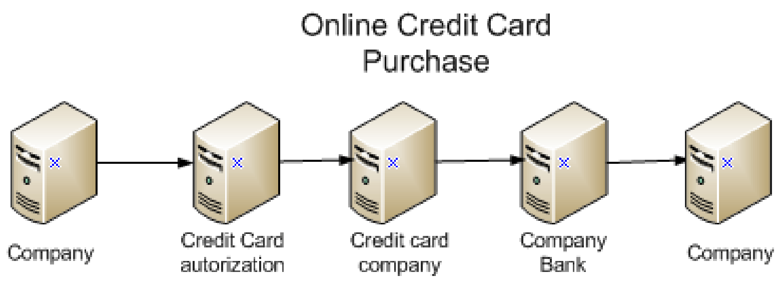
Easy administration—The replication service should have tools that are easy for the DBA to use to check the status and monitor the system. An example of the Microsoft SQL Server Replication Monitor tool is below.



Bridgeport Sailboats has four sales locations and at each location there is one database to record sales transactions. They use publisher/subscriber replication (historically referred to as master/slave replication) to write all of their sales transactions to the data warehouse. The data warehouse is located at the corporate office where upper management uses the data warehouse to perform analysis with decision support tools. Each sales location acts as a primary site (publisher) and then replicates the sales transactions to its subscriber, which is the data warehouse. The data warehouse cannot write or update data to the primary sites. Each sales location can only write or update data on its own server and replicate that data to the data warehouse.

Workflow ownership—Like the publisher/subscriber ownership, all updates and writes are made on the primary site, but once the data is replicated on to the next site, that site is also allowed to update the data. An easy way to grasp the workflow ownership is to look at an order processing system where the processing of orders follows a series of steps.

The following diagram is a simplification of an online credit card purchase transaction where workflow replication might be applied:

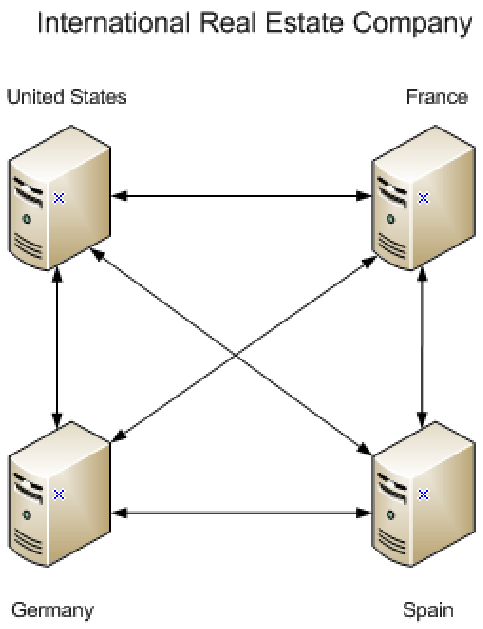


The steps of workflow ownership replication for an online credit card purchase could be:

1. Company sends credit card info for authorization.
2. Once authorized, the data is updated and sent to the credit card company for processing.
3. The credit card company then credits the amount to the company bank account.
4. Finally, the company is notified that the transaction was completed and what funds were credited.

At each of these steps the data was replicated to the next subscriber and then updated by that subscriber.

Update-anywhere ownership—This is like a peer-to-peer network, where all sites can update the replicated data on any site. This a shared ownership environment that allows local sites to function autonomously even when other sites are not available. One simple example of update-anywhere ownership is a company in which each location owns its own information and the locations also need access to data at other locations. For example, view the diagram below of an international real estate company.



A person may visit the United States office to put up a property for sale in Germany. Since the property is in Germany, the sales agent in the United States must access to the server in Germany to put the property for sale. When the data for the property is updated in the database in Germany the data would be replicated to each of the other sites.

Test Yourself

Select all that are true of the different forms of ownership of replicated data.

Update-anywhere ownership can lead to problems with updates and consistency since any server can update any other server.

This is true. With update-anywhere ownership, each server operates somewhat independently, so updates to one server may conflict with updates to a different server.

The principle of workflow ownership gives update rights to each server in sequence.

This is true, but only one server can update the data at any time.

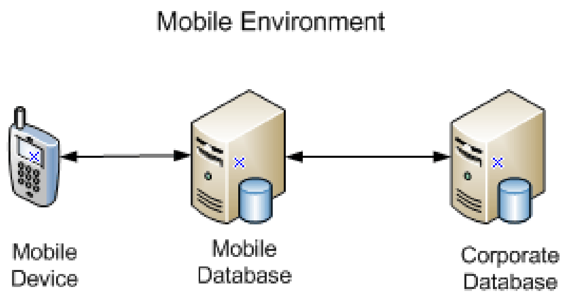
In a publisher-subscriber ownership structure, all updates to database replicas happen at the publisher and are propagated to each of the replicas.

This is true. The replicas can only read the data. All updates are processed by the publisher.

Introduction to Mobile Databases

The mobile industry is rapidly expanding and countless Internet of Things (IoT) devices are changing the way that people conduct business, with web access almost anywhere anytime. Of course there are limitations due to cost and security, but mobile databases offer solutions for some of these limitations.

A *mobile database* is a database that is portable and physically separate from the corporate database server but is capable of communicating with that server from remote sites and sharing corporate data.



An example of a company that uses mobile databases is FedEx. Each delivery truck is equipped with a mobile device that is capable of scanning a package to update the status of the delivery. Then, as soon as the mobile database is updated and communication is available, the corporate database is updated and a customer can go online and view the status of the delivery.

Issues with Mobile Databases

There are several issues with mobile databases:

- While wireless communication is improving, wireless bandwidth is variable and generally lower than copper or fiber.
- There are many areas where wireless connectivity is not available.
- It can be expensive to transfer large amounts of data over wireless networks.
- Security raises many concerns. The transferring of data is more vulnerable, the mobile device could be stolen, and information could be stolen.
- When many people are using mobile devices, the network congestion can degrade communications performance.

On top of these issues, there have been proposals for new transaction models because the classical transaction models may not be appropriate for a mobile environment.

Events that may occur while using a mobile device that need to be taken into consideration:

- A mobile device may disconnect in the middle of a transaction.
- The originating site of the transaction may be different from the ending site.
- Transactions may be lost if a mobile device breaks or is lost.

The *Kangaroo Transaction model* is based on the concepts of open-nested transactions and split transactions, which support mobility and disconnections. This is very important because in a mobile environment a device can move around and disconnect unexpectedly at anytime. Also, query processing must deal with location-aware queries and location-dependent queries, as well as moving object database queries, spatio-temporal queries, and continuous queries. The costs for query optimization are much more difficult to estimate.

Test Yourself

Given the kinds of issues that affect mobile databases, like variable wireless bandwidth, continuous queries, etc., select the best strategy for distributed query optimization:

Static query optimization

The goal of any query optimization is to reduce the total cost of a request. Static query optimization removes the runtime overhead of calculating a new plan each time the query is run, but the plan it generates may not be optimal by the time the query is actually executed on a mobile database.

Dynamic query optimization

Dynamic query optimization is preferable for a mobile database since the conditions are constantly changing.

Oracle Replication

Oracle advanced replication supports both synchronous and asynchronous replication. In the standard edition of Oracle there can only be one master site, but with the enterprise edition there may be multiple master sites. Oracle manages replication objects using replication groups, which are created to organize schema objects that are required by a particular application. An Oracle replication environment supports two types of sites: master sites and materialized view sites. There are four main types of replication in Oracle:

- materialized view replication,
- single master replication,
- multimaster replication, and
- hybrid replication.

Summary

Replication is the process of distributing multiple copies of data to one or more sites. It is important because it enables businesses to provide users with access to current data where and when they need it. With replication there is improved availability, reliability, performance and support for more users. Also, replication can be a means of disaster recovery, with a second database server available if the master server fails. The replication server is the software system that manages the replication and the typical mechanisms are database snapshots and database triggers.

A mobile database is a database that is portable and physically separate from the corporate database server. The needs of mobile databases are robust because of the increasing demand for applications on mobile devices. Since this is a fairly new developing technology, there are still a number of issues that need to be addressed—for example, security and transaction handling. Mobile devices mainly use the Kangaroo transaction model.

Oracle Advanced Replication is available in the standard edition and the enterprise edition of Oracle. There are four types of replication available: materialized view replication, single master replication, multimaster replication, and hybrid replication.