Chapter 15. Distributed Database Systems

Table of contents

- Objectives
- Introduction
- Context
- Client-server databases
  - The 2-tier model
    - The client
    - The server
    - Query processing in 2-tier client-server systems
    - Advantages of the client-server approach
    - Disadvantages of the client-server approach
  - Variants of the 2-tier model
    - Business (application) logic
    - Business logic implemented as stored procedures
  - The 3-tier architecture
- Distributed database systems
  - Background to distributed systems
  - Motivation for distributed database systems
- Fragmentation independence
- Replication independence
- Update strategies for replicated and non-replicated data
  - Eager (synchronous) replication
    - Eager replication and distributed reliability protocols
    - The two-phase commit (2PC) protocol
    - Read-once / write-all protocol
  - Lazy or asynchronous replication
    - Lazy group replication
    - Lazy master replication
- Reference architecture of a distributed DBMS
- Discussion topics

Objectives

At the end of this chapter you should be able to:

- Understand what is meant by client-server database systems, and describe variations of the client-server approach.
- Describe the essential characteristics of distributed database systems.
- Distinguish between client-server databases and distributed databases.
- Describe mechanisms to support distributed transaction processing.

- Compare strategies for performing updates in distributed database systems.
- Describe the reference architecture of distributed DBMSs.

Introduction

In parallel with this chapter, you should read Chapter 22 and Chapter 23 of Thomas Connolly and Carolyn Begg, "Database Systems A Practical Approach to Design, Implementation, and Management", (5th edn.).

Distributed databases have become an integral part of business computing in the past years. The ability to maintain the integrity of data and provide accurate and timely processing of database queries and updates across multiple sites has been an important factor in enabling businesses to utilise data in a range of different locations, sometimes on a global scale. Standardisation of query languages, and of the Relational and Object models, has assisted the integration of different database systems to form networks of integrated data services. The difficulties of ensuring the integrity of data, that updates are timely, and that users receive a uniform rate of response no matter where on the network they are situated, remain, in many circumstances, major challenges to database vendors and users. In this chapter we shall introduce the topic of distributed database systems. We shall examine a range of approaches to distributing data across networks, and examine a range of strategies for ensuring the integrity and timeliness of the data concerned. We shall look at mechanisms for enabling transactions to be performed across different machines, and the various update strategies that can be applied when data is distributed across different sites.

Context

Many of the issues considered in other chapters of this module require a degree of further consideration when translated into a distributed context. When it becomes a requirement to distribute data across a network, the processes of transaction processing, concurrency control, recovery, security and integrity control and update propagation become significantly more involved. In this chapter we shall introduce a number of extensions to mechanisms which we have previously considered for non-distributed systems.

Client-server databases

For years, serious business databases were monolithic systems running only on one large machine, accessed by dumb terminals. In the late 1980s, databases evolved so that an application, called a 'client', on one machine could run against a database, called a 'server', on another machine. At first, this client-server

database architecture was only used on mini and mainframe computers. Between 1987 and 1988, vendors like Oracle Corp and Gupta first moved the client function and then the database server down to microcomputers and local area networks (LANs).

Today, the client-server concept has evolved to cover a range of approaches to distributing the processing of an application in a variety of ways between different machines.

An example of the client-server approach is the SQLserver system, from Microsoft. The SQLserver system is run on a server machine, which is usually a fairly powerful PC. A client program is run, usually on a separate machine, and makes requests for data to SQLserver via a local area network (LAN). The application program would typically be written in a language such as Visual Basic or Java. This approach allows multiple client machines on the network to request the same records from the database on the server. SQLserver will ensure that only one user at a time modifies any specific record.

The 2-tier model

Client-server architecture involves multiple computers connected in a network. Some of the computers (clients) process application programs and some computers (servers) perform database processing.

This approach is known as the 2-tier model of client-server computing, as it is made up of the two types of component, clients and servers. It is also possible that a machine that acts as a server to some clients, may itself act as a client to another server. This arrangement also falls under the 2-tier model, as it still only comprises the two types of machine within the network.

The client

This is the front-end of the client-server system. It handles all aspects of the user interface — it is the front-end because the client presents the system to the user. It can also be used to provide PC-based application development tools used to enter, display, query and manipulate data on the central server, and to build applications. The client operating system is usually Windows, MACOS, Linux or Unix.

The server

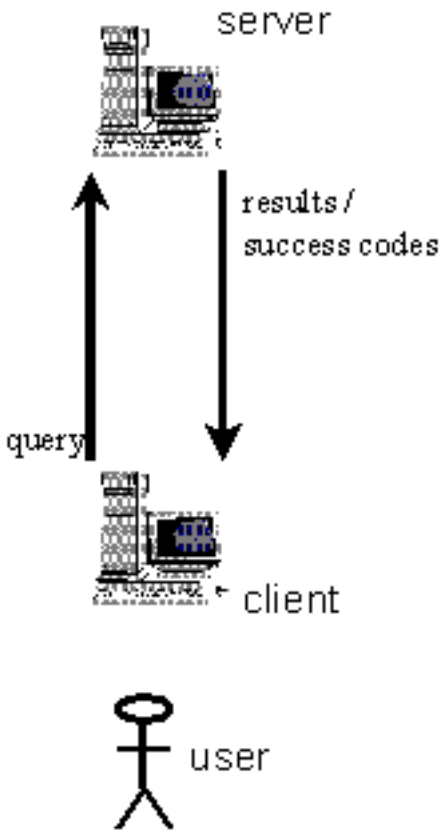
Servers perform functions such as database storage, integrity checking and data dictionary maintenance, and provide concurrent access control. Moreover, they also perform recovery and optimise query processing. The server controls access to the data by enforcing locking rules to ensure data integrity during transactions. The server can be a PC, mini or mainframe computer, and usually

employs a multi-tasking operating system such as Ubuntu Server and Windows Server OS.

Query processing in 2-tier client-server systems

Typically, in a client-server environment, the user will interact with the client machine through a series of menus, forms and other interface components. Supposing the user completes a form to issue a query against a customer database. This query may be transformed into an SQL SELECT statement by code running on the client machine. The client will then transmit the SQL query over the network to the server. The server receives the command, verifies the syntax, checks the existence and availability of the referenced objects, verifies that the user has SELECT privileges, and finally executes the query. The resulting data is formatted and sent to the application, along with return codes (used to identify whether the query was successful, or if not, which error occurred). On receipt of the data, the client might carry out further formatting - for example, creating a graph of the data - before displaying it to the user.

The following diagram illustrates how a user interacts with a client system, which transmits queries to the database server. The server processes the queries and returns the results of the query, or a code indicating failure of the query for some reason.



Advantages of the client-server approach

- Centralized storage: Users do not have to retain copies of corporate data on their own PCs, which would become quickly out of date. They can be assured that they are always working with the current data, which is stored on the server machine.
- Improved performance: Processing can be carried out on the machine most appropriate to the task. Data-intensive processes can be carried out on the server, whereas data-entry validation and presentation logic can be executed on the client machine. This reduces unnecessary network traffic and improves overall performance. It also gives the possibility of optimising the hardware on each machine to the particular tasks required of that machine.
- Scalability: If the number of users of an application grows, extra client

machines can be added (up to a limit determined by the capacity of the network or server) without significant changes to the server.

Disadvantages of the client-server approach

- Complexity: Operating database systems over a LAN or wide area network (WAN) brings extra complexities of developing and maintaining the network. The interfaces between the programs running on the client and server machines must be well understood. This usually becomes increasingly complex when the applications and/or DBMS software come from different vendors.
- Security: A major consideration, as preventative measures must be in place to protect against data theft or corruption on the client and server machines, and during transmission over the network.

Review question 1

- Explain what is meant by the 2-tier model of client-server computing.
- Why is it sometimes said that client-server computing improves the scalability of applications?
- What additional security issues are involved in the use of a client-server system, compared with a traditional mainframe database accessed via dumb terminals?

Variants of the 2-tier model

Business (application) logic

The relative workload on the client and server machines is greatly affected by the way in which the application code is distributed. The more application logic that is placed on client machines, the more of the work these machines will have to do. Furthermore, more data will need to be transmitted from servers to client machines because the servers do not contain the application logic that might have been used to eliminate some of the data prior to transmission. We can avoid this by transferring some of the application logic to the server. This helps to reduce the load on both the clients and the network. This is known as the split logic model of client-server computing. We can take this a stage further, and leave the client with only the logic needed to handle the presentation of data to users, placing all of the functional logic on servers; this is known as the remote presentation model.

Business logic implemented as stored procedures

The main mechanism for placing business logic on a server is known as 'stored procedures'. Stored procedures are collections of code that usually include SQL

for accessing the database. Stored procedures are invoked from client programs, and use parameters to pass data between the procedure and invoking program. If we choose to place all the business logic in stored procedures on the server, we reduce network traffic, as intermediate SQL results do not have to be returned to client machines. Stored procedures are compiled, in contrast with uncompiled SQL sent from the client. A further performance gain is that stored procedures are usually cached, therefore subsequent calls to them do not require additional disk access.

2-tier client-server architectures in which a significant amount of application logic resides on client machines suffer from the following further disadvantages:

- Upgrades and bug fixes must be made on every client machine. This problem is compounded by the fact that the client machines may vary in type and configuration.
- The procedural languages used commonly to implement stored procedures are not portable between machines. Therefore, if we have servers of different types, for example Oracle and Microsoft, the stored procedures will have to be coded differently on the different server machines. In addition, the programming environments for these languages do not provide comprehensive language support as is found in normal programming languages such as C++ or Java, and the testing/debugging facilities are limited.

Though the use of stored procedures improves performance (by reducing the load on the clients and network), taking this too far will limit the number of users that can be accommodated by the server — i.e. there will reach a point at which the server becomes a bottleneck because it is overloaded with the processing of application transactions.

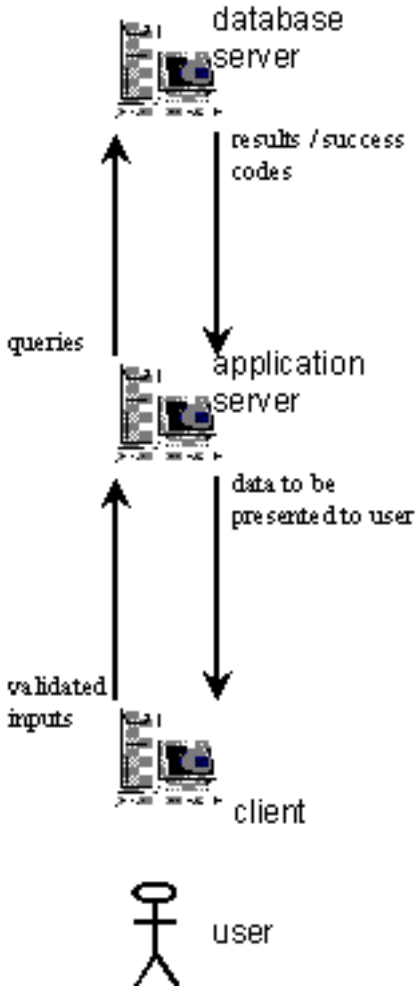
The 3-tier architecture

The 3-tier architecture introduces an applications server, which is a computer that fits in the middle between the clients and server. With this configuration, the client machines are freed up to focus on the validation of data entered by users, and the formatting and presentation of results. The server is also freed to concentrate on data-intensive operations, i.e. the retrieval and update of data, query optimisation, processing of declarative integrity constraints and database triggers, etc. The new middle tier is dedicated to the efficient execution of the application/business logic.

The application server can perform transaction management and, if required, ensure distributed database integrity. It centralises the application logic for easier administration and change.

The 3-tier model of client-server computing is best suited to larger installations. This is true because smaller installations simply do not have the volume of transactions to warrant an intermediate machine dedicated to application logic.

The following diagram illustrates how a user interacts with a client system, which deals with the validation of user input and the presentation of query results. The client system communicates with a middle tier system, the applications server, which formulates queries from the validated input data and sends queries to the database server. The database server processes the queries and sends to the applications server the results of the query, or a code indicating failure of the query for some reason. The application server then processes these results (or code) and sends data to the client system to be formatted and presented to the user.



Activity 1 – Investigating stored procedures

Read the documentation of the DBMS of your choice and investigate stored procedures as implemented in the environment. A visit to the software's website

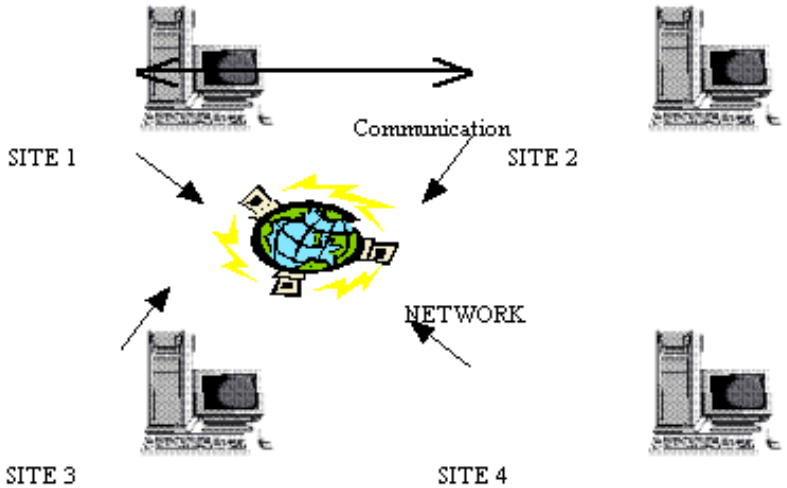
maybe also be useful. Identify the types of situations in which stored procedures are used, and by looking at a number of examples of stored procedures, develop an overall understanding for the structure of procedures and how they are called from a PL/SQL program.

Distributed database systems

Background to distributed systems

A distributed database system consists of several machines, the database itself being stored on these machines, which communicate with one another usually via high-speed networks or telephone lines. It is not uncommon for these different machines to vary in both technical specification and function, depending upon their importance and position in the system as a whole. Note that the generally understood description of a distributed database system given here is rather different from the client-server systems we examined earlier in the chapter. In client-server systems, the data is not itself distributed; it is stored on a server machine, and accessed remotely from client machines. In a distributed database system, on the other hand, the data is itself distributed among a number of different machines. Decisions therefore need to be made about the way in which the data is to be distributed and also about how updates are to be propagated over multiple sites.

The distribution of data by an organisation throughout its various sites and departments, allows data to be stored where it was generated, or indeed is most used, while still being easily accessible from elsewhere. The general structure of a distributed system is shown below:



A distributed system

In effect, each separate site in the example is really a database system site in its own right, each location having its own databases, as well as its own DBMS and transaction management software. It is commonly assumed when discussion arises concerning distributed database systems, that the many databases involved are widely spread from each other in geographical terms. Importantly, it should be made clear that geographical distances will have little or no effect upon the overall operation of the distributed system; the same technical problems arise whether the databases are simply logically separated or if separated by a great distance.

Motivation for distributed database systems

So why have distributed databases become so desirable? There are a number of reasons that promote the use of a distributed database system; these can include such factors as:

- The sharing of data
- Local autonomy
- Data availability
- Improving performance
- Improving system reliability

Furthermore, it is likely that the organisation that has chosen to implement the system will itself be distributed. By this, we mean that there are almost always several departments and divisions within the company structure. An illustrative example is useful here in clarifying the benefits that can be gained by the use of distributed database systems.

Scenario banking system

Imagine a banking system that operates over a number of separate sites; for the sake of this example, let us consider two offices, one in Manchester and another in Birmingham. Account data for Manchester accounts is stored in Manchester, while Birmingham's account data is stored in Birmingham. We can now see that two major benefits are afforded by the use of this system: efficiency of processing is increased due to the data being stored where it is most frequently accessed, while an increase in the accessibility of account data is also gained.

The use of distributed database systems is not without its drawbacks, however. The main disadvantage is the added complexity that is involved in ensuring that proper co-ordination between the various sites is possible. This increase in complexity can take a variety of forms:

- **Greater potential for bugs:** With a number of databases operating concurrently, ensuring that algorithms for the operation of the system are correct becomes an area of great difficulty. The potential is there for the existence of extremely subtle bugs.
- **Increased processing overhead:** The additional computation required in order to achieve inter-site co-ordination is a considerable overhead not present in centralised systems.

Date (1999) gives the 'fundamental principle' behind a truly distributed database:

"to the user; a distributed system should look exactly like a NONdistributed system.""

In order to accomplish this fundamental principle, twelve subsidiary rules have been established. These twelve objectives are listed below:

1. Local autonomy
2. No reliance on a central site
3. Continuous operation
4. Location independence
5. Fragmentation independence
6. Replication independence
7. Distributed query processing
8. Distributed transaction management
9. Hardware independence
10. Operating system independence
11. Network independence
12. DBMS independence

These twelve objectives are not all independent of one another. Furthermore, they are not necessarily exhaustive and, moreover, they are not all equally significant.

Probably the major issue to be handled in distributed database systems is the way in which updates are propagated throughout the system. Two key concepts play a major role in this process:

- **Data fragmentation:** The splitting up of parts of the overall database across different sites.
- **Data replication:** The process of maintaining updates to data across different sites.

Fragmentation independence

A system can support data fragmentation if a given stored relation can be divided up into pieces, or 'fragments', for physical storage purposes. Fragmentation is desirable for performance reasons: data can be stored at the location where it is most frequently used, so that most operations are purely local and network traffic is reduced.

A fragment can be any arbitrary sub-relation that is derivable from the original relation via restriction (horizontal fragmentation – subset of columns) and projection (vertical fragmentation – subset of tuples) operations. Fragmentation independence (also known as fragmentation transparency), therefore, allows users to behave, at least from a logical standpoint, as if the data were not fragmented at all. This implies that users will be presented with a view of the data in which the fragments are logically combined together by suitable joins and unions. It is the responsibility of the system optimiser to determine which fragment needs to be physically accessed in order to satisfy any given user request.

Replication independence

A system supports data replication if a given stored relation - or, more generally, a given fragment - can be represented by many distinct copies or replicas, stored at many distinct sites.

Replication is desirable for at least two reasons. First, it can mean better performance (applications can operate on local copies instead of having to communicate with remote sites); second, it can also mean better availability (a given replicated object - fragment or whole relation - remains available for processing so long as at least one copy remains available, at least for retrieval purposes).

What problems are associated with data replication and fragmentation?

Both data replication and fragmentation have their related problems in implementation. However, distributed non-replicated data only has problems when the relations are fragmented.

The problem of supporting operations, such as updating, on fragmented relations has certain points in common with the problem of supporting operations on join and union views. It follows too that updating a given tuple might cause that tuple to migrate from one fragment to another, if the updated tuple no longer satisfies the relation predicate for the fragment it previously belonged to (Date, 1999).

Replication also has its associated problems. The major disadvantage of replication is that, when a given replicated object is updated, all copies of that object must be updated — the update propagation problem. Therefore, in addition to transaction, system and media failures that can occur in a centralised DBMS,

a distributed database system (DDBMS) must also deal with communication failures. Communication failures can result in a site that holds a copy of the object being unavailable at the time of the update.

Furthermore, the existence of both system and communication failures poses complications because it is not always possible to differentiate between the two (Ozsu and Valduriez, 1996).

Update strategies for replicated and non-replicated data

There are many update strategies for replicated and fragmented data. This section will explore these strategies and will illustrate them with examples from two of the major vendors.

Eager (synchronous) replication

Gray et al (1996) states that eager replication keeps all replicas exactly synchronised at all nodes (sites) by updating all the replicas as part of one atomic transaction. Eager replication gives serialisable execution, therefore there are no concurrency anomalies. But eager replication reduces update performance and increases transaction times, because extra updates and messages are added to the transaction. Eager replication typically uses a locking scheme to detect and regulate concurrent execution.

With eager replication, reads at connected nodes give current data. Reads at disconnected nodes may give stale (out-of-date) data. Simple eager replication systems prohibit updates if any node is disconnected. For high availability, eager replication systems allow updates among members of the quorum or cluster. When a node joins the quorum, the quorum sends the node all replica updates since the node was disconnected.

Eager replication and distributed reliability protocols

Distributed reliability protocols (DRPs) are implementation examples of eager replication. DRPs are synchronous in nature (ORACLE, 1993), and often use remote procedure calls (RPCs).

DRPs enforce atomicity (the all-or-nothing property) of transactions by implementing atomic commitment protocols such as the two-phase commit (2PC) (Gray, 1979). Although a 2PC is required in any environment in which a single transaction can interact with several autonomous resource managers, it is particularly important in a distributed system (Ozsu and Valduriez, 1996). 2PC extends the effects of local atomic commit actions to distributed transactions, by insisting that all sites involved in the execution of a distributed transaction

agree to commit the transaction before its effects are made permanent. Therefore, 2PC is an example of one copy equivalence, which asserts that the values of all physical copies of a logical data item should be identical when the transaction that updates it terminates.

The inverse of termination is recovery. Distributed recovery protocols deal with the problem of recovering the database at a failed site to a consistent state when that site recovers from failure (Ozsu and Valduriez, 1996). The 2PC protocol also incorporates recovery into its remit.

Exercise 1

What is meant by the terms 'atomic commitment protocol' and 'one copy equivalence'?

The two-phase commit (2PC) protocol

The 2PC protocol works in the following way (adapted from Date, 1999): COMMIT or ROLLBACK is handled by a system component called the Co-ordinator, whose task it is to guarantee that all resource managers commit or rollback the updates they are responsible for in unison - and furthermore, to provide that guarantee even if the system fails in the middle of the process.

Assume that the transaction has completed its database processing successfully, so that the system-wide operation it issues is COMMIT, not ROLLBACK. On receiving the COMMIT request, the Co-ordinator goes through the following two-phase process:

1. First, the Co-ordinator instructs all resource managers to get ready either to commit or rollback the current transaction. In practice, this means that each participant in the process must force-write all log entries for local resources used by the transaction out to its own physical log. Assuming the force-write is successful, the resource manager now replies 'OK' to the Co-ordinator, otherwise it replies 'Not OK'.

2. When the Co-ordinator has received replies from all participants, it force-writes an entry to its own physical log, recording its decision regarding the transaction. If all replies were 'OK', that decision is COMMIT; if any replies were 'Not OK', the decision is ROLLBACK. Either way, the Co-ordinator then informs each participant of its decision, or each participant must then COMMIT or ROLLBACK the transaction locally, as instructed.

If the system or network fails at some point during the overall process, the restart procedure will look for the decision record in the Co-ordinator's log. If it finds it, then the 2PC process can pick up where it left off. If it does not find it, then it assumes that the decision was ROLLBACK, and again the process can complete appropriately. However, in a distributed system, a failure on the part of the Co-ordinator might keep some participants waiting for the Co-ordinator's

decision. Therefore, as long as the participant is waiting, any updates made by the transaction via that participant are kept locked.

Review question 2

- Explain the role of an application server in a 3-tier client-server network.
- Distinguish between the terms 'fragmentation' and 'replication' in a distributed database environment.
- Describe the main advantage and disadvantage of eager replication.
- During the processing of the two-phase commit protocol, what does the Co-ordinator do if it is informed by a local resource manager process that it was unable to force-write all log entries for the local resources used by the transaction out to its own physical log?

Read-once / write-all protocol

A further replica-control protocol that enforces one-copy serialisability is known as read-once / write-all (ROWA) protocol. ROWA protocol is simple, but it requires that all copies of a logical data item be updated before the transaction can terminate.

Failure of one site may block a transaction, reducing database availability.

A number of alternative algorithms have been proposed that reduce the requirement that all copies of a logical data item be updated before the transaction can terminate. They relax ROWA by mapping each write to only a subset of the physical copies. One well-known approach is quorum-based voting, where copies are assigned votes, and read and write operations have to collect votes and achieve a quorum to read/write data.

Three-phase commit is a non-blocking protocol which prevents the 2PC blocking problem from occurring by removing the uncertainty for participants after their votes have been placed. This is done through the inclusion of a pre-commit phase that relays information to participants, advising them that a commit will occur in the near future.

Lazy or asynchronous replication

Eager replication update strategies, as identified above, are synchronous, in the sense that they require the atomic updating of some number of copies. Lazy group replication and lazy master replication both operate asynchronously.

If the users of distributed database systems are willing to pay the price of some inconsistency in exchange for the freedom to do asynchronous updates, they will insist that:

1. the degree of inconsistency be bounded precisely, and that

2. the system guarantees convergence to standard notions of 'correctness'.

Without such properties, the system in effect becomes partitioned as the replicas diverge more and more from one another (Davidson et al, 1985).

Lazy group replication

Lazy group replication allows any node to update any local data. When the transaction commits, a transaction is sent to every other node to apply the root transaction's updates to the replicas at the destination node. It is possible for two nodes to update the same object and race each other to install their updates at other nodes. The replication mechanism must detect this and reconcile the two transactions so that their updates are not lost (Gray et al, 1996).

Timestamps are commonly used to detect and reconcile lazy-group transactional updates. Each object carries the timestamp of its most recent update. Each replica update carries the new value and is tagged with the old object timestamp. Each node detects incoming replica updates that would overwrite earlier committed updates. The node tests if the local replica's timestamp and the update's old timestamp are equal. If so, the update is safe. The local replica's timestamp advances to the new transaction's timestamp and the object value is updated. If the current timestamp of the local replica does not match the old timestamp seen by the root transaction, then the update may be 'dangerous'. In such cases, the node rejects the incoming transaction and submits it for reconciliation. The reconciliation process is then responsible for applying all waiting update transactions in their correct time sequence.

Transactions that would wait in an eager replication system face reconciliation in a lazy group replication system. Waits are much more frequent than deadlocks because it takes two waits to make a deadlock.

Lazy master replication

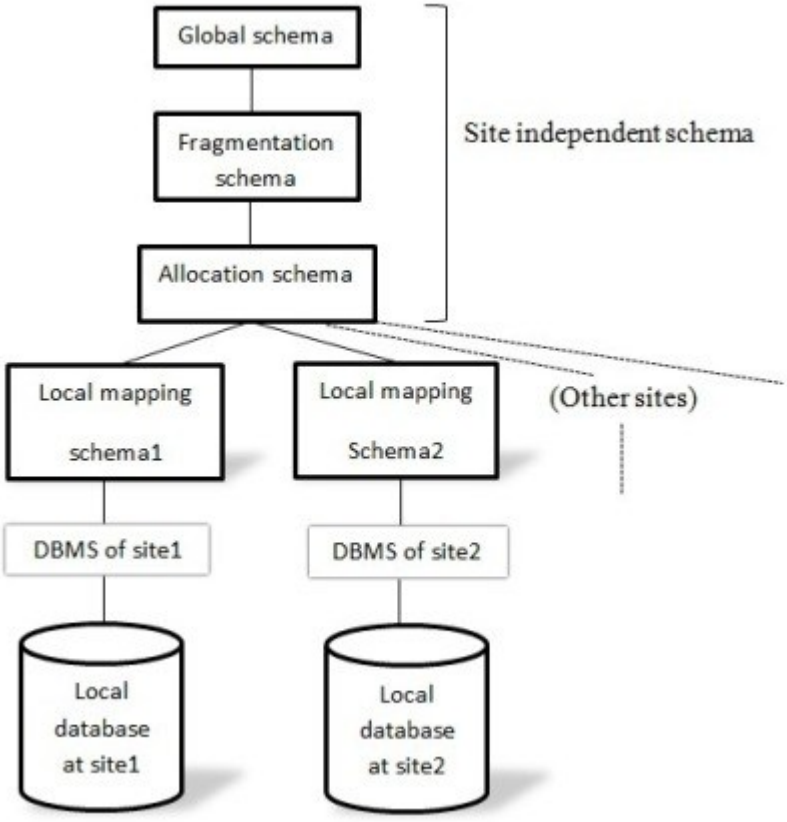
Another alternative to eager replication is lazy master replication. Gray et al (1996) states that this replication method assigns an owner to each object. The owner stores the object's correct value. Updates are first done by the owner and then propagated to other replicas. When a transaction wants to update an object, it sends a remote procedure call (RPC) to the node owning the object. To achieve serialisability, a read action should send read-lock RPCs to the masters of any objects it reads. Therefore, the node originating the transaction broadcasts the replica updates to all the slave replicas after the master transaction commits. The originating node sends one slave transaction to each slave node. Slave updates are timestamped to assure that all the replicas converge to the same final state. If the record timestamp is newer than a replica update timestamp, the update is 'stale' and can be ignored. Alternatively, each master node sends replica updates to slaves in sequential commit order.

Review question 3

When an asynchronous update strategy is being used, if two copies of a data item are stored at different sites, what mechanism can be used to combine the effect of two separate updates being applied to these different copies?

Reference architecture of a distributed DBMS

In chapter 1 we looked at the ANSI_SPARC three-level architecture of a DBMS. The architecture reference shows how different schemas of the DBMS can be organised. This architecture cannot be applied directly to distributed environments because of the diversity and complexity of distributed DBMSs. The diagram below shows how the schemas of a distributed database system can be organised. The diagram is adopted from Hirendra Sisodiya (2011).



Reference architecture for distributed database

1. **Global schema**

   The global schema contains two parts, a global external schema and a global conceptual schema. The global schema gives access to the entire system. It provides applications with access to the entire distributed database system, and logical description of the whole database as if it was not distributed.

2. **Fragmentation schema**

   The fragmentation schema gives the description of how the data is partitioned.

3. **Allocation schema**

   Gives a description of where the partitions are located.

4. **Local mapping**

   The local mapping contains the local conceptual and local internal schema. The local conceptual schema provides the description of the local data. The local internal schema gives the description of how the data is physically stored on the disk.

Review question 4

List the characteristics of applications that can benefit most from:

- synchronous replication
- asynchronous replication

Discussion topics

1. We have covered the client-server and true distributed database approaches in this chapter. Client-server systems distribute the processing, whereas distributed systems distribute both the processing and the data. Discuss the proposition that most commercial applications are adequately supported by a clientserver approach, and do not require the additional features of a truly distributed database system.

2. Discuss the proposition that, in those situations where a distributed database solution is required, most applications are adequately provided for by a lazy or asynchronous replication strategy, and do not require the sophistication of an eager or synchronous replication system. Discuss the implications for end users of synchronous and asynchronous updating.