CHAPTER 19

Distributed Databases

Practice Exercises

19.1 How might a distributed database designed for a local-area network differ from one designed for a wide-area network?

Answer: Data transfer on a local-area network (LAN) is much faster than on a wide-area network (WAN). Thus replication and fragmentation will not increase throughput and speed-up on a LAN, as much as in a WAN. But even in a LAN, replication has its uses in increasing reliability and availability.

19.2 To build a highly available distributed system, you must know what kinds of failures can occur.

a. List possible types of failure in a distributed system.

b. Which items in your list from part a are also applicable to a centralized system?

Answer:

a. The types of failure that can occur in a distributed system include
   i. Site failure.
   ii. Disk failure.
   iii. Communication failure, leading to disconnection of one or more sites from the network.

b. The first two failure types can also occur on centralized systems.

19.3 Consider a failure that occurs during 2PC for a transaction. For each possible failure that you listed in Practice Exercise 19.2a, explain how 2PC ensures transaction atomicity despite the failure.

Answer: A proof that 2PC guarantees atomic commits/aborts inspite of site and link failures, follows. The main idea is that after all sites reply with a <**ready** T> message, only the co-ordinator of a transaction can make a commit or abort decision. Any subsequent commit or abort by a

site can happen only after it ascertains the co-ordinator's decision, either directly from the co-ordinator, or indirectly from some other site. Let us enumerate the cases for a site aborting, and then for a site committing.

a. A site can abort a transaction T (by writing an <**abort** T> log record) only under the following circumstances:

   i. It has not yet written a <**ready** T> log-record. In this case, the co-ordinator could not have got, and will not get a <**ready** T> or <**commit** T> message from this site. Therefore only an abort decision can be made by the co-ordinator.

   ii. It has written the <**ready** T> log record, but on inquiry it found out that some other site has an <**abort** T> log record. In this case it is correct for it to abort, because that other site would have ascertained the co-ordinator's decision (either directly or indirectly) before actually aborting.

   iii. It is itself the co-ordinator. In this case also no site could have committed, or will commit in the future, because commit decisions can be made only by the co-ordinator.

b. A site can commit a transaction T (by writing an <**commit** T> log record) only under the following circumstances:

   i. It has written the <**ready** T> log record, and on inquiry it found out that some other site has a <**commit** T> log record. In this case it is correct for it to commit, because that other site would have ascertained the co-ordinator's decision (either directly or indirectly) before actually committing.

   ii. It is itself the co-ordinator. In this case no other participating site can abort/ would have aborted, because abort decisions are made only by the co-ordinator.

19.4 Consider a distributed system with two sites, A and B. Can site A distinguish among the following?

- B goes down.
- The link between A and B goes down.
- B is extremely overloaded and response time is 100 times longer than normal.

What implications does your answer have for recovery in distributed systems?

Answer:

Site A cannot distinguish between the three cases until communication has resumed with site B. The action which it performs while B is inaccessible must be correct irrespective of which of these situations has actually

occurred, and must be such that B can re-integrate consistently into the distributed system once communication is restored.

19.5 The persistent messaging scheme described in this chapter depends on timestamps combined with discarding of received messages if they are too old. Suggest an alternative scheme based on sequence numbers instead of timestamps.

Answer: We can have a scheme based on sequence numbers similar to the scheme based on timestamps. We tag each message with a sequence number that is unique for the (sending site, receiving site) pair. The number is increased by 1 for each new message sent from the sending site to the receiving site.

The receiving site stores and acknowledges a received message only if it has received all lower numbered messages also; the message is stored in the *received-messages* relation.

The sending site retransmits a message until it has received an ack from the receiving site containing the sequence number of the transmitted message, or a higher sequence number. Once the acknowledgment is received, it can delete the message from its send queue.

The receiving site discards all messages it receives that have a lower sequence number than the latest stored message from the sending site. The receiving site discards from *received-messages* all but the (number of the) most recent message from each sending site (message can be discarded only after being processed locally).

Note that this scheme requires a fixed (and small) overhead at the receiving site for each sending site, regardless of the number of messages received. In contrast the timestamp scheme requires extra space for every message. The timestamp scheme would have lower storage overhead if the number of messages received within the timeout interval is small compared to the number of sites, whereas the sequence number scheme would have lower overhead otherwise.

19.6 Give an example where the read one, write all available approach leads to an erroneous state.

Answer: Consider the balance in an account, replicated at N sites. Let the current balance be \$100 – consistent across all sites. Consider two transactions T_1 and T_2 each depositing \$10 in the account. Thus the balance would be \$120 after both these transactions are executed. Let the transactions execute in sequence: T_1 first and then T_2. Let one of the sites, say s, be down when T_1 is executed and transaction t_2 reads the balance from site s. One can see that the balance at the primary site would be \$110 at the end.

19.7 Explain the difference between data replication in a distributed system and the maintenance of a remote backup site.

Answer: In remote backup systems all transactions are performed at the primary site and the data is replicated at the remote backup site. The

remote backup site is kept synchronized with the updates at the primary site by sending all log records. Whenever the primary site fails, the remote backup site takes over processing.
The distributed systems offer greater availability by having multiple copies of the data at different sites whereas the remote backup systems offer lesser availability at lower cost and execution overhead.
In a distributed system, transaction code runs at all the sites whereas in a remote backup system it runs only at the primary site. The distributed system transactions follow two-phase commit to have the data in consistent state whereas a remote backup system does not follow two-phase commit and avoids related overhead.

19.8 Give an example where lazy replication can lead to an inconsistent database state even when updates get an exclusive lock on the primary (master) copy.

Answer: Consider the balance in an account, replicated at N sites. Let the current balance be \$100 – consistent across all sites. Consider two transactions T_1 and T_2 each depositing \$10 in the account. Thus the balance would be \$120 after both these transactions are executed. Let the transactions execute in sequence: T_1 first and then T_2. Suppose the copy of the balance at one of the sites, say s, is not consistent – due to lazy replication strategy – with the primary copy after transaction T_1 is executed and let transaction T_2 read this copy of the balance. One can see that the balance at the primary site would be \$110 at the end.

19.9 Consider the following deadlock-detection algorithm. When transaction T_i, at site S_1, requests a resource from T_j, at site S_3, a request message with timestamp n is sent. The edge (T_i, T_j, n) is inserted in the local wait-for graph of S_1. The edge (T_i, T_j, n) is inserted in the local wait-for graph of S_3 only if T_j has received the request message and cannot immediately grant the requested resource. A request from T_i to T_j in the same site is handled in the usual manner; no timestamps are associated with the edge (T_i, T_j). A central coordinator invokes the detection algorithm by sending an initiating message to each site in the system.

On receiving this message, a site sends its local wait-for graph to the coordinator. Note that such a graph contains all the local information that the site has about the state of the real graph. The wait-for graph reflects an instantaneous state of the site, but it is not synchronized with respect to any other site.

When the controller has received a reply from each site, it constructs a graph as follows:

- The graph contains a vertex for every transaction in the system.
- The graph has an edge (T_i, T_j) if and only if:
  - There is an edge (T_i, T_j) in one of the wait-for graphs.

- An edge (T_i, T_j, n) (for some n) appears in more than one wait-for graph.

Show that, if there is a cycle in the constructed graph, then the system is in a deadlock state, and that, if there is no cycle in the constructed graph, then the system was not in a deadlock state when the execution of the algorithm began.

Answer: Let us say a cycle $T_i \rightarrow T_j \rightarrow \cdots \rightarrow T_m \rightarrow T_i$ exists in the graph built by the controller. The edges in the graph will either be local edges of the from (T_k, T_l) or distributed edges of the form (T_k, T_l, n). Each local edge (T_k, T_l) definitely implies that T_k is waiting for T_l. Since a distributed edge (T_k, T_l, n) is inserted into the graph only if T_k's request has reached T_l and T_l cannot immediately release the lock, T_k is indeed waiting for T_l. Therefore every edge in the cycle indeed represents a transaction waiting for another. For a detailed proof that this imlies a deadlock refer to Stuart et al. [1984].

We now prove the converse implication. As soon as it is discovered that T_k is waiting for T_l:

a. a local edge (T_k, T_l) is added if both are on the same site.

b. The edge (T_k, T_l, n) is added in both the sites, if T_k and T_l are on different sites.

Therefore, if the algorithm were able to collect all the local wait-for graphs at the same instant, it would definitely discover a cycle in the constructed graph, in case there is a circular wait at that instant. If there is a circular wait at the instant when the algorithm began execution, none of the edges participating in that cycle can disappear until the algorithm finishes. Therefore, even though the algorithm cannot collect all the local graphs at the same instant, any cycle which existed just before it started will anyway be detected.

19.10 Consider a relation that is fragmented horizontally by *plant_number*:

employee (*name, address, salary, plant_number*)

Assume that each fragment has two replicas: one stored at the New York site and one stored locally at the plant site. Describe a good processing strategy for the following queries entered at the San Jose site.

a. Find all employees at the Boca plant.

b. Find the average salary of all employees.

c. Find the highest-paid employee at each of the following sites: Toronto, Edmonton, Vancouver, Montreal.

d. Find the lowest-paid employee in the company.

Answer:

a. i. Send the query $\Pi_{name}(employee)$ to the Boca plant.
   ii. Have the Boca location send back the answer.

b. i. Compute average at New York.
   ii. Send answer to San Jose.

c. i. Send the query to find the highest salaried employee to Toronto, Edmonton, Vancouver, and Montreal.
   ii. Compute the queries at those sites.
   iii. Return answers to San Jose.

d. i. Send the query to find the lowest salaried employee to New York.
   ii. Compute the query at New York.
   iii. Send answer to San Jose.

19.11 Compute $r \ltimes s$ for the relations of Figure 19.9.
Answer: The result is as follows.

$r \ltimes s =$

A	B	C
1	2	3
5	3	2

19.12 Give an example of an application ideally suited for the cloud and another that would be hard to implement successfully in the cloud. Explain your answer.
Answer: Any application that is easy to partition, and does not need strong guarantees of consistency across partitions, is ideally suited to the cloud. For example, Web-based document storage systems (like Google docs), and Web based email systems (like Hotmail, Yahoo! mail or GMail), are ideally suited to the cloud. The cloud is also ideally suited to certain kinds of data analysis tasks where the data is already on the cloud; for example, the Google Map-Reduce framework, and Yahoo! Hadoop are widely used for data analysis of Web logs such as logs of URLs clicked by users.
Any database application that needs transactional consistency would be hard to implement successfully in the cloud; examples include bank records, academic records of students, and many other types of organizational records.

19.13 Given that the LDAP functionality can be implemented on top of a database system, what is the need for the LDAP standard?
Answer: The reasons are:

a. Directory access protocols are simplified protocols that cater to a limited type of access to data.

b. Directory systems provide a simple mechanism to name objects in a hierarchical fashion which can be used in a distributed directory system to specify what information is stored in each of the directory servers. The directory system can be set up to automatically forward queries made at one site to the other site, without user intervention.

19.14 Consider a multidatabase system in which it is guaranteed that at most one global transaction is active at any time, and every local site ensures local serializability.

a. Suggest ways in which the multidatabase system can ensure that there is at most one active global transaction at any time.

b. Show by example that it is possible for a nonserializable global schedule to result despite the assumptions.

Answer:

a. We can have a special data item at some site on which a lock will have to be obtained before starting a global transaction. The lock should be released after the transaction completes. This ensures the single active global transaction requirement. To reduce dependency on that particular site being up, we can generalize the solution by having an election scheme to choose one of the currently up sites to be the co-ordinator, and requiring that the lock be requested on the data item which resides on the currently elected co-ordinator.

b. The following schedule involves two sites and four transactions. T_1 and T_2 are local transactions, running at site 1 and site 2 respectively. T_{G1} and T_{G2} are global transactions running at both sites. X_1, Y_1 are data items at site 1, and X_2, Y_2 are at site 2.

T_1	T_2	T_{G1}	T_{G2}
write(Y_1)			
		read(Y_1)	
		write(X_2)	
	read(X_2)		
	write(Y_2)		
			read(Y_2)
			write(X_1)
read(X_1)			

In this schedule, T_{G2} starts only after T_{G1} finishes. Within each site, there is local serializability. In site 1, $T_{G2} \rightarrow T_1 \rightarrow T_{G1}$ is a serializability order. In site 2, $T_{G1} \rightarrow T_2 \rightarrow T_{G2}$ is a serializability order. Yet the global schedule schedule is non-serializable.

19.15 Consider a multidatabase system in which every local site ensures local serializability, and all global transactions are read only.

a. Show by example that nonserializable executions may result in such a system.

b. Show how you could use a ticket scheme to ensure global serializability.

Answer:

a. The same system as in the answer to Exercise 19.14 is assumed, except that now both the global transactions are read-only. Consider the schedule given below.

T_1	T_2	T_{G1}	T_{G2}
			read(X_1)
write(X_1)			
		read(X_1)	
		read(X_2)	
	write(X_2)		
			read(X_2)

Though there is local serializability in both sites, the global schedule is not serializable.

b. Since local serializability is guaranteed, any cycle in the system wide precedence graph must involve at least two different sites, and two different global transactions. The ticket scheme ensures that whenever two global transactions access data at a site, they conflict on a data item (the ticket) at that site. The global transaction manager controls ticket access in such a manner that the global transactions execute with the same serializability order in all the sites. Thus the chance of their participating in a cycle in the system wide precedence graph is eliminated.