CHAPTER 23

Parallel and Distributed Transaction Processing

Practice Exercises

23.1 What are the key differences between a local-area network and a wide-area network, that affect the design of a distributed database?

Answer:

Data transfer is much faster, and communication latency is much lower on a local-area network (LAN) than on a wide-area network (WAN). Protocols that require multiple rounds of communication maybe acceptable in a local area network, but distributed databases designed for wide-area networks try to minimize the number of such rounds of communication.

Replication to a local node for reducing latency is quite important in a wide-area network, but less so in a local area network.

Network link failure and network partition are also more likely in a wide-area network than in a local area network, where systems can be designed with more redundancy to deal with failures. Protocols designed for wide-area networks should handle such failures without creating any inconsistencies in the database.

23.2 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.

23.3 Consider a failure that occurs during 2PC for a transaction. For each possible failure that you listed in Exercise 23.2a, explain how 2PC ensures transaction atomicity despite the failure.

Answer:
A proof that 2PC guarantees atomic commits/aborts in spite of site and link failures follows. The main idea is that after all sites reply with a <**ready** *T*> message, only the coordinator 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 coordinator's decision, either directly from the coordinator 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 coordinator 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 coordinator.

   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 coordinator's decision (either directly or indirectly) before actually aborting.

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

b. A site can commit a transaction *T* (by writing a <**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 coordinator's decision (either directly or indirectly) before actually committing.

ii. It is itself the coordinator. In this case no other participating site can abort or would have aborted because abort decisions are made only by the coordinator.

23.4 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 it must be such that *B* can re-integrate consistently into the distributed system once communication is restored.

23.5 The persistent messaging scheme described in this chapter depends on timestamps. A drawback is that they can discard received messages only if they are too old, and may need to keep track of a large number of received messages. Suggest an alternative scheme based on sequence numbers instead of timestamps, that can discard messages more rapidly.

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.

23.6 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 entire database 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. Different data items may be replicated at different nodes.
In a distributed system, transaction code can run at all the sites, whereas in a remote backup system it runs only at the primary site. The distributed system transactions needs to follow two-phase commit or other consensus protocols to keep the data in consistent state, whereas a remote backup system does not follow two-phase commit and avoids related overhead.

23.7 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 if data were read from a node other than the master.

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.

23.8 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 edgem (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 implies 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 be detected.

23.9 Consider the chain-replication protocol, described in Section 23.4.3.2, which is a variant of the primary-copy protocol.

a. If locking is used for concurrency control, what is the earliest point when a process can release an exclusive lock after updating a data item?

b. While each data item could have its own chain, give two reasons it would be preferable to have a chain defined at a higher level, such as for each partition or tablet.

c. How can consensus protocols be used to ensure that the chain is uniquely determined at any point in time?

Answer:

a. The lock can be released only after the update has been recorded at the tail of the chain, since further reads will read the tail. Two phase locking may also have to be respected.

b. The overhead of recording chains per data item would be high. Even more so, in case of failures, chains have to be updated, which would have an even greater overhead if done per item.

c. All nodes in the chain have to agree on the chain membership and order. Consensus can be used to ensure that updates to the chain are done in a fault-tolerant manner. A fault-tolerant coordination service such as ZooKeeper or Chubby could be used to ensure this consensus, by updating metadata that is replicated using consensus; the coordination service hides the details of consensus, and allows storage and update of (a limited amount of) metadata.

23.10 If the primary copy scheme is used for replication, and the primary gets disconnected from the rest of the system, a new node may get elected as primary. But the old primary may not realize it has got disconnected, and may get reconnected subsequently without realizing that there is a new primary.

a. What problems can arise if the old primary does not realize that a new one has taken over?

b. How can leases be used to avoid these problems?

c. Would such a situation, where a participant node gets disconnected and then reconnected without realizing it was disconnected, cause any problem with the majority or quorum protocols?

Answer:

a. The old primary may receive read requests and reply to them, serving old data that is missing subsequent updates.

b. Leases can be used so that at the end of the lease, the primary knows that it if it did not successfuly renew the lease, it should stop serving requests. If it is disconnected, it would be unable to renew the lease.

c. This situation would not cause a problem with the majority protocol since the write set (or write quorum) and the read set (read quorum) must have at least one node in common, which would serve the latest value.

23.11 Consider a federated database 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 federated database 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 coordinator and requiring that the lock be requested on the data item which resides on the currently elected coordinator.

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 nonserializable.

23.12 Consider a federated database 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 23.11 is assumed, except that now both the global transactions are read-only. Consider the following schedule:

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 systemwide 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 systemwide precedence graph is eliminated.

23.13 Suppose you have a large relation $r(A, B, C)$ and a materialized view $v = {}_{A}\gamma_{\text{sum}(B)}(r)$. View maintenance can be performed as part of each transaction that updates r, on a parallel/distributed storage system that supports transactions across multiple nodes. Suppose the system uses two-phase commit along with a consensus protocol such as Paxos, across geographically distributed data centers.

a. Explain why it is not a good idea to perform view maintenance as part of the update transaction, if some values of attribute A are "hot" at certain points in time, that is, many updates pertain to those values of A.

b. Explain how operation locking (if supported) could solve this problem.

c. Explain the tradeoffs of using asynchronous view maintenance in this context.

Answer:

This is a very bad idea from the viewpoint of throughput. Most transactions would now update a few aggregate records, and updates would get serialized on the lock. The problem that due to Paxos delays plus 2PC delays, commit processing will take a long time (hundreds of milliseconds) and there would be very high contention on the lock. Transaction throughput would decrease to tens of transactions per second, even if transactions do not conflict on any other items.

If the storage system supported operation locking, that could be an alternative to improve concurrency, since view maintenance can be done using operation locks that do not conflict with each other. Transaction throughput would be greatly increased.

Asynchronous view maintenance would avoid the bottleneck and lead to much better throughput, but at the risk of reads of the view seeing stale data.