CPS352 Lecture: Distributed Database Systems last revised 4/20/2017

Materials: Projectable showing horizontal and vertical partitioning

I. What is a Distributed Database System?

A. In a distributed database system, the database is stored on several computers located at multiple physical sites.

1. This distinguishes it from a centralized system, in which the database is stored at a single location.

2. This distinguishes it from a parallel/cluster system, in which the database is stored on multiple computers at the same physical site.

3. This distinguishes it from a client-server system, in which the database is only stored at one site, though it may be accessed from multiple sites. (Of course, the server in a client-server system may itself be a distributed database!)

B. There are two major variants of distributed systems

1. A distributed system can be HOMOGENOUS. All sites run the same brand of DBMS software (and generally the same general kind of hardware and operating system.)

2. A distributed system can be HETEROGENOUS. Different sites run different DBMS software, perhaps on different kinds of hardware running different operating systems as well. The databases at the different sites may have different schemes, and the individual sites may not even be aware of each other.

3. Naturally, coordination of activities between sites is much easier in the homogenous case - and most actual implementations are of this type. This is the type we will focus on.

II. What are the Advantages and Disadvantages of a Distributed System?

A. We will basically consider the advantages and disadvantages of a remotely distributed system as over against a large, centralized system.

1. One of the major advantages of a distributed system is sharing of data generated at the different sites, without requiring that all the data be moved to a single central site.

2. Another advantage is the possibility of LOCAL CONTROL and AUTONOMY. Within boundaries established by the need for sharing, each site can control its own data, determine what is stored how etc.

3. The possibility of mproved response times to queries. As over-against a centralized system, a distributed system that stores data at the site(s) that use it the most allows them to access the data more quickly than they would if they had to get the data from a central site via a communication link.

4. With the rise of reliance on the internet and ecommerce, a fourth motivation that has always been there have become especially prominent: Reliability and availability.

   In a centralized system, the failure of the main system - or some sort of disaster at the place where it is located - shuts down all processing activity until the problem is fixed. But in a distributed system, the failure of a site may reduce performance and/or make some data unavailable. But processing of most kinds of requests can continue. Indeed, if data is replicated at multiple sites, it is possible for most processing to proceed - albeit perhaps with reduced performance.

   We say that the system has improved availability.

5. Much of the work on distributed systems was done before the rise of ecommerce and cloud computing, but the issues that were discovered and solutions to them arose in this context are also relevant to the kinds of systems needed for these contexts as well. Some of the issues have helped motivate movement toward NoSQL database models and away from ACID transactions. But that's for

another lecture.

Our focus in this lecture will be on distribured computing in the context of traditional relational databases with ACID transactions.

B. Disadvantages

1. One major disadvantage of a distributed system is the cost and time required for communication between sites.

a. This is not necessarily a disadvantage, if the alternatives are a centralized system where ALL queries require communication vs a distributed system where SOME queries can be processed locally at the requesting site.

b. But operations requiring access to data at multiple sites will almost always involve more communication between sites than would be required if all the data involved were at one location.

c. The performance impact of communication depends a great deal on what kind of communication links are used. In particular, note that the performance of a network is determined by the slowest link. If the Internet is used, this is often the "last mile" connection between a DBMS site and the ISP.

d. The time cost of any given message is given by:

Access delay + (message length) / (data rate)

i. Access delay is the overhead time needed to set up for a message between sites. This varies greatly from system to system, but will tend to be a constant that is independent of message length.

ii. For short messages, access delay may be the dominant cost.

iii. For longer messages, (message length) / (data rate) may be the dominant cost.

e. Depending on the configuration, communication cost may dominate disk access cost, in which case a distributed systems might need to be optimized to minimize the number and volume of messages, rather than disk accesses.

2. A second disadvantage is increased complexity. As we shall see, choosing a query processing strategy, performing updates, dealing with crashes, and concurrency control are all much more complex in a distributed system.

3. A third disadvantage related to the second is that distributed systems are much harder to debug. In fact, the algorithms used must be bug-proof; discovering the cause of a problem that arises only under certain circumstances of operation timing is not possible using conventional debugging techniques.

III. Partitioning and Replication of Data

A. At the heart of the idea of a distributed system is the distribution of data over multiple sites.

1. The conceptually simplest distribution scheme is to distribute at the table level: any given table is stored in its entirety at some site.

2. However, there are situations which call for splitting a table up between sites. This is called PARTITIONING or FRAGMENTATION. (Fragmentation is the term used in the book, but partitioning seems to be the more common term now.)

3. Oftentimes, an organization will have branch offices at different physical locations, each of which is a network site. If different tuples in a table are associated with different sites, it may make sense to partition the table HORIZONTALLY - i.e. by rows.

   a. Example: the book used the example of a deposit relation for a bank, where each branch stores the rows pertaining to its customers.

   b. Example: a customer relation for a sales organization may be stored so that each customer's data is stored by the branch office that normally services him/her/it.

   c. When a table is partitioned horizontally, the entire table can be reconstructed by doing the UNION of the partitions.

   d. The impact of horizontal partitioning on performance involves both gains and losses:

      i. Operations by a local office on relevant rows of the table incur no communications cost.

      ii. Operations on the entire table (e.g. complete printouts, searches for rows meeting some criterion other than the one used for partitioning, summaries, etc) may incur massive communications cost - more than if the entire unpartitioned relation were stored at one site.

4. In other cases, it will make sense to partition a table VERTICALLY - by columns.

   a. One motivation for this is security.

      i. Most DBMS's provide mechanisms whereby certain users may be given access to some columns of a table but not others. (In a relational scheme, this is done by using views. The restricted users are given access to the view, but not directly to the underlying table.)

ii. In a distributed system, there is increased risk of a breach of the security mechanisms, since each site has some degree of autonomy. Thus, it may be decided that certain sensitive columns should only be stored at the site having primary responsibility for them.

iii. For example, the salary column of a personnel relation may be stored only at the site containing the payroll department. Other sites may store personnel data contained in other columns.

b. Another motivation for vertical partitioning is selective replication of a table. It may be decided that a copy of the entire table should be kept at one central site, but that other sites might store copies of the most frequently used columns. In this way, most queries on the table can be done without communication cost; but some queries (those involving the infrequently-used columns) will have to be referred to the central site.)

c. A vertically-partitioned table can be reconstructed using a natural join if one of the following criteria is met:

i. Each partition participating in the join includes the primary key.

or

ii. Each partition includes a system-generated tuple identifier that is unique for each row of the table. (In essence, this is a system-generated primary key for the table; but it is never actually seen by users.)

d. Note that joining partitions to reconstruct the table is only necessary if there is NO site that has a copy of the whole table. This would be a fairly unusual situation.

5. If reasons for both kinds of partitioning pertain to a given table, then it can be partitioned both ways.

PROJECT Partitioning Example

a. Note that personnel data is distributed horizontally by regional office.

b. In two of the regions, salary data is partitioned off vertically and stored at corporate headquarters. This is not done in the central region - perhaps the division office there shares a system with corporate payroll.

c. Job history data is partitioned off vertically but is not partitioned horizontally. Presumably, an employees past job history may encompass several regions and so cannot be associated with any one. This partition is presumably used less often than other parts of the table, and so may be stored at a central archival site (corporate headquarters).

6. A partitioning scheme can be specified by giving a relational algebra or SQL-like expression specifying exactly what goes into each partition.

a. In a horizontal partitioning, each partition will be specified by a selection operation for each partition that specifies what tuples belong in it.

b. In a vertical partitioning, each partition will be specified by a projection operation for each partition that specifies what columns belong in it.

c. In a combined partitioning, both operations will be present.

B. Two of the major motivations for distributing a database are improved availability and improved performance through making it possible for frequent queries to be processed locally at the originating site. Both of these advantages typically require that some data be REPLICATED - i.e. that the same information be stored at more than one site.

1. Replicated data may still be available in the face of the failure of one site, providing that another site has a copy.

2. Local access to a private replica of commonly-used data is more efficient than having to go to some other site to get at the only copy.

3. Replication can be combined with partitioning: often only the most commonly-used partitions of a given table is replicated widely.

4. Of course, replication is a form of redundancy, and creates its share of problems.

   a. Replicating data uses extra storage space.

   b. Updating of replicated data is a significant problem, to which there are no easy solutions.

      i. Whatever update strategy is used must ensure that all copies of the data agree with one another; but it is inevitable that there be a time during the update process when some copies have been updated while others have not. The update strategy must prevent transactions from producing inconsistent results through reading the wrong version of an item being updated at the same time.

      ii. It is always possible that a replica of data being updated exists on a site that is currently not functioning - either crashed or cut off from the rest of the network by a communication failure. The update strategy must assure that an inconsistent replica is not used in a computation.

   c. When data is replicated, the advantages of improved availability and access must be weighed against these disadvantages to decide whether replicating a particular portion of the database is worth it.

d. In general, replication improves performance for read-only transactions, while requiring more effort (and hence potentially hurting performance) for transactions involving updating data.

C. One author suggested the following broad guidelines for deciding whether to partition, replicate, or centralize a given relation:

1. If a relation used at many sites is small, then replicate it.

2. If a relation used at many sites is large, but there is a way to partition it so that most queries will only access the local part, then partition it.

3. If a relation used at many sites is large, and cannot be partitioned in such a way as to allow most queries to only access the local part, then centralize it. (In this case, a partitioned scheme would require more communications overhead since many queries would need access to data stored at two or more sites.)

Example: suppose we were to take the example library database we have been using throughout the course and distribute it. (We assume that the library has several branches.)

- The category relation might well be replicated at each site, since it is small and referred to frequently.

- The relation tying together a book's call_number, copy_number and barcode might well be partitioned horizontally, with each branch getting the portion pertaining to the books that belong there.

- The relation tying together a book and its authors might well be stored centrally, since it is large, cannot be partitioned in a site-specific way, and is likely accessed infrequently.

D. A highly desirable property of a distributed system is LOCATION TRANSPARENCY: a user should not have to know WHERE an object is stored in order to access it.

1. Location transparency makes it possible for data to be moved to another site if appropriate, without requiring users to alter their queries.

2. Location transparency makes it possible for the query strategy planner to select the most inexpensive site to get data from in the case of replicated data.

3. Location transparency allows a query to access replicated data from another site if the normal site from which it is obtained is down.

E. An important property is that top-level items (databases or perhaps tables within a database) have unique names. This can be handled in a number of ways:

1. A central name server that "registers" all names - but clearly this can be become a performance bottleneck and makes the whole system vulnerable to the failure of a single site.

2. Incorporating the id of a site (e.g. its ip number) into the names of items residing on the site - but this flies in the face of location transparency.

3. Combining the above with the use of aliases at this site. For example, this is the approach db2 uses

   a. You have been accessing the database through one of the workstations, though in fact the database resides on the server, joshua.

   b. Each of the workstations actually runs a copy of db2, which accesses a client instance (not the instance in which the data is actually stored.) The client instance, however, records aliases to items residing on the server.

c. When you issue a command like connect to sample ..., what actually occurs is that the dbms software you are running checks the client database and discovers that "sample" is an alias for a database on joshua, and connects to it. (It happens to the case that this database is also called sample, but that is not necessary)

IV. Querying Distributed Data

A. When data is distributed, we have to deal with the possibility that a given transaction will involve processing at more than one site.

1. We associate a given transaction with the site where it originates; this is the place where the results must finally be sent.

2. A transaction is said to be LOCAL if it only accesses data stored at the site if originates at.

3. A transaction is said to be GLOBAL if it must access data at a site other than its point of origin. From the standpoint of a given site doing some of the work on a global transaction, the transaction may be either:

a. A transaction originating locally, but needing data from elsewhere.

b. A transaction originating elsewhere, but needed local data at the current site.

B. An important goal of a distributed system is FRAGMENTATION TRANSPARENCY. In the case of partitioned or replicated tables, the user should be able to formulate a query without knowing how the table is partitioned and/or replicated across multiple sites - i.e. from the user's standpoint, it should appear as if the the entire table exists at one site. (And when this is combined with location transparency, the appearance is given that this site is the user's own site.)

C. The strategy selection component of a query processor has a challenging job when dealing with a global query. Because of the time costs associated with communication of data, different plausible strategies may differ in time needed by orders of magnitude. C.J. Date gives the following illustration of this point.

1. Assume an organization has three tables, as follows:

   a. Schemes:

      Scheme S(S#, CITY) - relating suppliers to the city where they are located. (Primary key S#)

      Scheme P(P#, COLOR) - relating parts to their colors. (Primary key P#)

      Scheme SP(S#, P#) - specifying which suppliers supply which parts

   b. Cardinalities of the tables (number of tuples):

      n(S) = 10,000
      n(P) = 100,000
      n(SP) = 1,000,000

   c. Each tuple is 200 bits long.

   d. Tables S and SP are stored at site A, and P at site B.

2. Suppose that the data communication system can transmit 50,000 bits per second, with an 0.1 second access delay for each message. (This is an old example. While modern networks offer much higher data rates, the basic relationships between times for various strategies we shall see still hold.)

3. Now consider the following query (originating at some site A): Give the supplier numbers of all suppliers located in London who supply red parts. In SQL:

```
SELECT S.S#
  FROM S NATURAL JOIN SP NATURAL JOIN P
  WHERE CITY = 'LONDON' AND COLOR = 'RED';
```

Suppose that statistical information leads to the following estimates of sizes:

a. From the given n values, we can estimate that each part is supplied by 10 suppliers (n(SP) / n(P)), and that each supplier supplies an average of 100 parts (n(SP) / n(S))

b. Suppose we also can learn that the number of red parts is 10, and the number of SP tuples pertaining to London suppliers is 100,000.

4. Now consider various plausible strategies. In each case, we consider only communication time. The cost for doing the necessary joins is comparable for each strategy.

a. Copy relation P to site A and do all processing there.

T = 0.1 + (100,000 tuples * 200 bits/tuple) / (50,000 bits/second)
= 0.1 + 400 seconds = 400.1 seconds = 6.67 minutes

(The entire relation P is transmitted as a single long message)

b. Copy relations S and SP to site B and do all processing there.

T = 0.1 + (10,000 tuples * 200 bits/tuple) / (50,000 bits/second) +
0.1 + (1,000,000 tuples * 200 bits/tuple) / (50,000 bits/second)
+ 0.1 seconds to send final answer

= 40.1 seconds + 4000.1 seconds + .1 seconds = 4040.3 seconds = 1.12 hours

c. Join relations S and SP at site A, then select out the London tuples. For each, query site B to see if the part involved is red.

T = 2 * 0.1 * 100,000 seconds = 20,000 seconds = 5.56 hours

- There are 100,000 SP tuples pertaining to London suppliers

- Each gives rise to a query message and a response (2 messages).

- Since the messages will presumably be small, the dominant cost will be the access delay of 0.1 second for each. (Even if the message were the full 200 bit tuple, transmission time would only be 0.004 seconds.)

d. Select tuples from P at site B for which the color is red, and send a message to site A requesting a list of London suppliers that supply that particular part. Then send final answer back to A.

T = 10 * (0.1 + 0.1 + 200 / 50,000) + 0.1 second = 2.05 seconds

- There are 10 P tuples for red parts

- Each gives rise to two messages - a query and a response. The query message time is essentially just the access delay.

- The statistical data leads us to expect that each part is supplied by 10 suppliers. Since 1/10 of the suppliers are in London, the reply message will typically contain one supplier.

e. Join S and SP at site A, select tuples from the join for which the city is London, and project the result over (S# P#). Move the projection to site B and complete the processing there. Then send final answer back to A.

T = 0.1 + (100,000*200)/50,000 + 0.1 = 400.2 seconds = 6.67 minutes

f. Select tuples from P at site B for which the color is red. Send these to site A and complete the processing there

T = 0.1 + (10 * 200) / 50,000 = approx 0.1 second

g. Note, then, that these strategies (all of which are plausible) have communication costs ranging from 0.1 second to the better part of a day!

D. One observation that can help to narrow down the number of options to be considered is to recognize that operations on data are of two types: data reducing and data expanding.

1. A data reducing operation produces a result that is smaller in size than what it started with.

a. Select and project are almost always data reducing.

b. Natural join may be data reducing if the number of tuples which successfully join is a fraction of the total number of tuples in the relations involved.

c. Theta join may also be data reducing if the selection condition eliminates most of the tuples produced by the join.

d. Intersection and division are also data reducing.

e. Union is data reducing if some tuples appear in both relations being combined.

f. Summarizing functions (count, sum, average etc.) are data reducing

2. A data expanding operation produces a result that is larger in size than what it started with.

   a. Cartesian join is always data expanding.

   b. Natural join and theta join may be data expanding.

3. As a general rule, data reducing operations should be performed at the site where data is stored before sending it to another site, and data expanding operations should be performed at the site where data is needed after it has been sent there.

E. One data reducing operator that can be used in some cases is the SEMIJOIN.

1. Example:

   a. Suppose we distribute our library database so that checkout information is stored locally, but full catalog information on a book (callno, title etc.) is stored centrally.

   b. Suppose we need to print out overdue notices for a group of checkouts pertaining to books that have just become overdue. We will do something like:

```
select callno, copyno, title, borrower_id
    from checkout natural join bookinfo
    where datedue = yesterday;
```

   c. One option for accomplishing this is to copy the entire bookinfo table from the central site to the local branch, and then do the join locally. This is an obviously bad idea, since the book relation is presumably quite large, and only a few rows are relevant to our query.

d. A better approach would be to send to the local branch only those tuples of the book relation which are relevant to the query - i.e. those whose callno matches the callno of one of the overdue books. This is computed by taking the semijoin:

bookinfo |X checkout

which is defined as

project bookinfo |X| checkout
bookinfo scheme

e. The complete strategy might then be as follows:

i. Compute project callno select datedue = yesterday checkout

ii. Send this to the central site.

iii. Compute the semijoin of bookinfo with this, project out callno and title, and send the result back to the local office.

iv. Join the information returned from the central site with checkout and project the desired fields.

f. It is obvious that, in this case, the semijoin is far superior to sending the entire bookinfo relation to the local site. Even though tuples flow both ways, the total volume of information transmitted is much less.

g. We might also compare this semijoin-based strategy to one other option involving only regular joins: Send the relevant checkout tuples to the central site, do the join with bookinfo there, and send the result back. The semijoin-based strategy involves transmitting the same number of tuples in each direction; but

since the tuples are projected down to smaller size first in each case, the semijoin-based strategy has an edge in terms of total data volume.

2. Formally, given relations r1 on scheme R1 and r2 on scheme R2, r1 |X r2 is defined to be

$$\text{project}_{R1}\ (r1\ |X|\ r2)$$

3. Observe that r2 |X| (r1 |X r2) = r2 |X| r1. This is one fact we relied on in the above example.

4. Further, r1 |X r2 = r1 |X $\text{project}_{R1 \text{ intersect } R2}$ r2. This is what let us do a projection locally before sending checkout tuples to the central site to be semijoined with bookinfo there.

V. Updating Distributed Data

A. Most of the complexities associated with distributed databases arise in conjunction with the updating of data in the database. The two major sources of complexity are these:

1. If a transaction updates data stored at more than one site, we must ensure that either all the updates commit or none of the updates commit. We must avoid the possibility that an update at one site commits while another aborts.

2. When data is replicated, we must make sure that ALL replicas are updated consistently. Since it is impossible to ensure that all updates to replicas of an item occur at exactly the same moment of time, we must also ensure that inconsistencies do not result from a transaction reading a replica that is not yet updated.

3. In addressing each of these problems, we must bear in mind that a distributed system is vulnerable to partial failure - some sites may be up while others are down.

a. An individual site may suffer a hardware failure or software crash at any time - including in the middle of an update operation in which it is one of the participating sites.

b. A communication link between sites can fail at any time - including in the middle of an update operation. The consequences of link failure range from causing a site to be out of touch with the rest of the network to a partitioning of the network into two disconnected subnetworks. When link failure makes communication between certain sites impossible, we say the network has been partitioned. (We do not worry about a link failure that still leaves a site connected to the rest of the network through an alternate path. We assume that lower levels of network software handle this transparently for us.)

c. In addition to the total failure of a site or link, we must also consider the possible garbling or loss of a message in transit.

i. If a message is garbled, lower layers of the software will request a repeat transmission.

ii. However, if a message is lost, appropriate action might need to be taken at a higher level.

B. We consider first the matter of updating data stored at more than one site.

1. To ensure that either all updates commit or no updates commit, we can use a protocol called the TWO-PHASE COMMIT PROTOCOL.

a. In this protocol, one site acts as the coordinator. Normally, this is the site where the transaction originated.

b. As each site completes its work on the transaction and becomes partially-committed, it sends a message to the coordinator.

i. Once the coordinator receives completion messages from all participating sites, it can begin the commit protocol.

ii. However, if it receives a failure message from any site, then it must instruct all sites to abort the transaction.

iii. The coordinator must also abort the transaction if it fails to receive a completion message from some site within a reasonable time. This is necessary to deal with the possibility that a participating site (or its link) might fail in mid transaction.

c. In phase 1 of the protocol, the following events occur:

i. The coordinator adds a <prepare T> entry to its log and forces all log records to stable storage.

ii. The coordinator sends a prepare-to-commit message to all participating sites.

iii. Each site normally adds a <ready T> entry to its log and forces all log entries to stable storage. It then sends a ready message to the coordinator.

iv. However, if some site wishes to abort the transaction, it may still do so by adding a <no T> entry to its log, forcing the log entries to stable storage, and then sending an abort message to the coordinator.

v. Once a site sends a ready message to the coordinator, it gives up the right to abort the transaction. It must go ahead with the commit if the coordinator tells it to do so.

d. In phase 2 of the protocol, the coordinator waits for replies from its prepare-to-commit message to come back.

i. If any comes back negative, or fails to come back in a reasonable time, then the coordinator must write an <abort T> entry in its log and send an abort message to all sites.

ii. If all replies come back positive, the coordinator writes a <commit T> entry in its log and sends a commit message to all sites.

iii. At this point, the decision of the coordinator is final. Whatever was decided, the protocol will now operate to ensure that it occurs - even if a site or link should now fail.

iv. Each site, when it receives the message from the coordinator, either commits or aborts the transaction, makes an appropriate entry in its log, and then sends an acknowledge message to the coordinator. (The sending of an acknowledgement is not strictly necessary - some implementations require this while others do not.)

v. When the coordinator receives acknowledge messages from all sites, it writes a <complete T> entry to its log.

vi. Should any site fail to respond within reasonable time, the coordinator may resend the message to it. This resending can continue at regular intervals until the site response comes back. (This is not strictly necessary, however. Once the coordinator has decided the transaction's fate, the site will assume responsibility for finding it out anyway.)

2. The two-phase commit protocol deals with site or link failures (other than failure of the coordinator) as follows:

a. If a site or link fails before sending its ready message, then the transaction will abort.

i. When the site comes back up or is reconnected, it may try to send a ready message, but if the coordinator has decided to abort the transaction the ready message will now be ignored.

ii. The coordinator will continue sending periodic abort messages to the site until it acknowledges; thus the protocol will eventually finish in a consistent state.

b. If a site or link fails after the site sends its ready message, and the failure results in the ready message being lost, then the result is the same as above.

c. If a site or link fails after the coordinator receives the site's ready message but before the site receives the final decision from the coordinator, then the site's log will contain a <ready T> entry. During recovery, the site must determine the fate of T by either receiving a message from the coordinator or consulting some other site that participated in the transaction.

d. If a site or link fails after it receives the coordinator's decision and records <abort T> or <commit T> in its log - but before the site sends an acknowledge message - then the site knows what to do about the transaction when it comes back up. It will send an acknowledge message in response to a resend of the coordinator's verdict. (The same situation holds if the site's acknowledge message is lost in transmission.)

3. Coordinator failure is handled as follows:

a. If the coordinator fails before it sends a final decision as to the transaction's fate to at least one site, then a site that

has already sent a ready message to the coordinator must wait until the coordinator recovers or is reconnected before deciding what to do about the transaction. (But a site that has not sent a ready message can timeout and decide to abort the transaction, of course; and if another site can find out it has done so, it can abort too.)

b. If the coordinator fails after sending a final decision to at least one site, other sites may be able to obtain the decision from that site so as not to have to wait.

c. In any case, when the coordinator recovers or is reconnected, it will inspect its log and send some decision to each site.

i. If it finds <start T> but no <prepare T>, it will presumably abort the transaction.

ii. If it finds <prepare T> but no <commit T> or <abort T>, it may try again to find out the status of the participating sites, or it may just abort the transaction.

iii. If it finds <abort T> or <commit T> but no <complete T>, it can start the process of sending abort/commit messages and waiting for acknowledgements over again.

iv. Of course, if it finds <complete T> then nothing need be done.

d. Under some circumstances, coordinator failure can result in BLOCKING. If a site has to wait for the recovery of the coordinator before it knows whether or not a given transaction committed, then certain data items may be locked for a long time. This is, of course, highly undesirable. (The book discusses some alternatives for dealing with situations like this, such as 3-phase commit.)

C. Now we consider the matter of updating replicated data.

1. Ultimately, all replicas of a given data item must be updated. However, the system should exhibit REPLICATION TRANSPARENCY; any update transaction should be able to execute without knowing how many replicas of a given item exist. (I.e. a transaction's write to one copy of an item should be converted by the DBMS into writes to all of them.)

2. One way to do this is to perform simultaneous updates at the transaction level - i.e. transform each update transaction into a set of updates at each site.

   a. This flies in the face of replication transparency

   b. This does ensure that all replicas of a given item are consistent, if each replica is locked as part of the update process.

   c. Moreover, it can make response time for update transactions poor.

   d. Finally, one or more replicas may be located at sites that are either down or out of contact due to failed links.

      i. If we made a transaction wait if ANY replica of an updated item is not available, then replication works against availability rather than for it. (We would be better off, from an availability for update standpoint, if data were not replicated.)

      ii. For this reason, some provision must be made for postponing an update to an inaccessible site. Some site must be responsible for informing it of updates that were missed once it comes back.

      iii. This can get complicated, because we must decide who is responsible for informing the unavailable site. What if the responsible site is offline when the other site comes back?

3. An alternate that is often used is to designate one copy of the data as the PRIMARY (or master) copy.

   a. Transactions that wish to read an item may read any replica, but all updates are done first to the primary copy.

   b. The site containing the primary copy is responsible for sending updates to sites containing replicas.

      i. This may be done after every update transaction.

      ii. This may involve resending the updates periodically if a site is down.

      iii. Or, if updates are infrequent and currency of data is not vital, a fresh copy of the entire file may be shipped from the primary site to secondary sites on a periodic basis.

   c. We accept the possibility that data at secondary sites may be a bit out of date. Thus, any transaction for which it is critical that we have the most recent data (e.g. a funds transfer) must read the primary copy. Secondary copies can be read only in cases where inconsistency could not result from stale data.

   d. Of course, we now must face the possibility that the site containing the primary copy of a data item should be offline or disconnected.

      i. In this case, we may declare the data item unavailable for update until the primary copy site comes back online.

      ii. Or, we may temporarily designate a secondary copy as the primary copy, and then report changes back to the primary site when it comes back online.

iii. However, designating a secondary site as the primary copy could lead to inconsistencies (requiring human intervention to straighten out) if partitioning of the network leads to two primary copies being declared - one in each partition.

VI. Concurrency Control for Distributed Data

A. So far, our discussion has dealt with problems that arise if just one transaction attempts to access or update a data item. We now consider how our concurrency control schemes must be modified to cope with distributed data.

B. First, we consider the lock-based schemes.

1. As always, we must lock a data item before we access it. But now we face the possibility of having to lock items stored at several different sites.

2. One approach to locking is to use a CENTRALIZED LOCK MANAGER located at one of the sites. All locks are obtained by means of a message to this site.

a. Compared to other schemes, this has two advantages.

i. A site that needs to lock several replicas of the same item can get all the locks it needs with a single message.

ii. As we shall see, this makes deadlock detection much easier.

b. However, centralized lock management has the usual disadvantages of centralizing any function in a distributed system.

i. Even a local transaction will now involve communication overhead if locking is needed.

ii. The lock manager is a potential bottleneck.

iii. Failure of the lock manager can cripple the system. (There is a way to deal with this by using an election scheme to choose a new lock manager in such cases. The book discusses this.)

3. An alternative to centralized locking is distributed locking, with each site managing the locks on items stored there.

a. The advantages are the opposites of the disadvantages of the centralized scheme. In particular, local transactions stay local.

b. There are some disadvantages, though.

i. The message overhead is increased significantly. For each item to be locked, at least three messages are needed, IN ADDITION TO THE MESSAGES NEEDED TO ACCESS THE DATA:

- A lock request message to the site where the item is stored.

- A lock granted message back to the requestor.

- An unlock message to finally release the item.

(Actually, this last message might be combined with the messages involved in the two-phase commit protocol.)

ii. Deadlock detection now becomes much harder. No one site has all the information needed to determine whether the system is deadlocked.

Example: Transaction T1 locks item Q1 at its own site, site S1.

Transaction T2 (at site S2) locks item Q3 at another site, site S3.

Transaction T1 now sends a message to site S3, requesting a lock on Q3. Transaction T1 is forced to wait.

Transaction T2 now sends a message to site S1, requesting a lock on Q1. Transaction T2 is forced to wait.

Transactions T1 and T2 are now deadlocked, but neither site knows it!

- Site S1 knows that its transaction, T1, is waiting for a lock at site S3, and that transaction T2 from S2 is waiting for a local lock held by T1. This can be represented by the graph

  (T2) --> (T1)

- Site S2 knows that its transaction, T2, is waiting for a lock at site S1.

- Site S3 knows that transaction T1 from site S1 is waiting for a local lock held by T2. This can be represented by the graph

  (T1) --> (T2)

- No one knows about the circular wait!

c. Distributed lock management can further complicate dealing with updates to replicated data

i. If the updating transaction is responsible for updating all copies of an item, then it would appear that no update can take place without first obtaining a lock on ALL the copies. This requires a great deal of message overhead (three messages per replica), and would prevent doing an update if one site were down. Furthermore, deadlock could occur if two transactions were trying to lock the same item but started with different copies. (Each would be holding locks on some copies awaiting the others.)

ii. These problems can be reduced by using a MAJORITY PROTOCOL. An updating transaction is only required to obtain locks on a majority of the replicas in order to proceed with the update. This cuts message traffic in half, allows updates even if one site is down, and reduces (but does not eliminate) the chance of deadlock.

Note: in order for this to work correctly, a transaction that needs to read and then update an item must obtain an exclusive lock originally, rather than obtaining a shared lock which is subsequently upgraded to exclusive.

iii. It is also possible to use a BIASED PROTOCOL for updates, whereby read-only operations are only required to lock one copy of an item, while update transactions must lock all copies.

d. Many of the difficulties in updated replicated items can be reduced if the primary copy method of updating is used. In this case, we need only require the updating transaction to lock the primary copy of the item.

C. Another issue that must be dealt with is the management of locks in conjunction with recovery. Any locks that were held by a transaction whose fate is in doubt during recovery must be held until the fate is determined - potentially resulting in an item being tied up for a long time. (The book discusses an alternative to two phase commit called three phase commit which addresses this issue.)

D. Because of the complexity of implementing locking in a distributed system, timestamp-based methods of concurrency control become an attractive alternative.

1. Some of the key ideas involved in adapting the timestamp algorithms for distributed use are discussed in the book.

2. One important point in any distributed scheme based on timestamps is ensuring consistency and uniqueness of timestamps across the system.

   a. Since each site generates its own timestamps, it is quite possible for two transactions originating at different sites to be given the same timestamp locally. To prevent this, we form a timestamp by concatenating two fields: the locally-generated timestamp and a unique site id, like this:

      [locally-generated timestamp] [site id]

      Note that the site id must become the low order part of the overall timestamp, so that it is significant only when resolving a situation where two transactions from different sites have the same locally-generated portion.

   b. We also want to ensure that if some transaction T1 receives a timestamp TS(T1) at some moment of time, some other transaction T2 does not receive a smaller timestamp at some later moment of time.

      i. This could theoretically be guaranteed if we used clocks to generate timestamps at each site, and if we ensured that the clocks were always synchronized. However, it is hard to ensure this in practice.

ii. The following is sufficient: if any site receives a request from a transaction originating elsewhere whose timestamp is greater than the current reading of its own timestamp clock, it must advance its timestamp clock to be one greater than the value of the timestamp of this transaction.

iii. This latter method will work even if timestamps are generated by a local counter, rather than a local clock.

VII. The CAP Theorem

A. We noted at the outset that one advantage of a distributed system over a centralized system is the potential for improved availability of data (by replication).

B. At the same time, we also noted that when data is replicated, consistency requires preventing a site from accessing an old version of a data item that is updated. But what if the network is partitioned at the time an update occurs?

1. If the primary copy method of ensuring consistency is used - i.e. a site that needs to be sure it has the latest copy of an item gets it from the primary site - and the network is partitioned in such a way that the site containing the primary copy is not accessible, then this works against availability.

2. The same sort of thing can happen with some sort of majority protocol, if the partitioning of the network makes it impossible to reach a majority of the replicas.

C. An important theorem known as the CAP theorem says that you can design protocols to achieve any two of Consistency, Availability, and Partition Tolerance - but you can't have all three!

Note that CAP defines Availability in a special way: "Every request received by a non-failing node in the system must result in a response" [Lynch and Gilbert cited by Sandalage and Fowler]. That is, if an update request is received by a node that ordinarily could fulfill it, but cannot because a network partition has made it impossible - then the data item is said to be non-available.

1. In a distributed system that uses something like the primary copy or majority protocol, a network partition could make some data unavailable at some nodes.

2. Availability could be preserved if we were willing to relax expectations regarding consistency.

D. The practical ramification of this is to explore alternative DBMS structures in which relaxation of the ACID requirements leads to trading off strong consistency in favor of better availability. We explore this next in the context of NoSQL database models.

   An alternative to strict ACID consistency in such systems is sometimes called BASE - Basically Available, Soft State, Eventually Consistent.