Distributed Databases
It is now common to have databases distributed over many sites.
- Different table may be stored at different sites.
- **Horizontal decomposition** – A single table may have some of its rows stored at one site, other rows at other sites.
- **Vertical decomposition** – A single table may have some of its columns at one site, other columns at other sites. All must have the same primary key.
- Both kinds of decomposition may be used on a single table.
- The part of such a table stored at one site is a fragment.

Potential Problems
- Now the costliest factor is the time needed to transmit data from one site to another.
- When a transaction makes a change to a distributed table, if the transaction process on one site fails, how can we make it fail at all the sites involved?
- How can we assure serializability of transactions when they operate on tables that are distributed?
- How can we assure serializability of transactions when they are initiated at different sites?

Replication Problems
- It is useful to **replicate** data – put copies of the same data at different sites.
  - If one site fails, the data is still available at other sites
  - Queries can be answered faster by having a local copy of the needed data.
- Data copies can cause problems.
  - How do we keep all the copies identical?
  - How many copies do we need and where should they be put?
  - After communication failures between sites, how can identity of the copies be restored?

Computing Distributed Joins
For simplicity, we’ll assume that each table is stored at only one site but different tables needed for the join are at different sites.
We could send the whole table at one site to the other site and have their join computed there. The main cost of the operation is the cost of transmitting that whole file over the network.
Usually we can do better than that. We can avoid transmitting dangling tuples (rows that will not be used to form any of the the rows in the output of the join operation).

Semijoin Reductions
Suppose \( R(X,Y) \bowtie S(Y,Z) \) needs to be computed, with \( R \) stored at site A and \( S \) stored at site B. To avoid transmitting dangling tuples,
- Send \( R \) from site A to site B
- At site B, compute \( S' = \text{semijoin } S \bowtie R \). This is the set of rows in \( S \) that will join with at least one row in \( R \) because their \( Y \) value is on the list that was sent from site A.
- Transmit \( S' \) from site B back to site A, where it is joined with \( R \). \( R \bowtie S' \) (this is the same as \( R \bowtie S \)).

Join of Three or More Relations
If three or more relations are to be joined, it may not be possible to avoid transmitting dangling tuples. The book gives an example where \( R(A,B) \) has rows of the form \( n,n \) and \( S(B,C) \) has rows of the form \( n,n \) but \( T(C,A) \) has rows of form \( n,n+1 \).
No matter which order we use to form the joins, almost all dangling tuples are transmitted, yet the join of these three relations is empty.
How can we know when all dangling tuples can be avoided, and how do we determine the steps in the process for doing so?
A **hypergraph** is a set of nodes and a set of hyperedges. A **hyperedge** is a set of nodes. It can have one, two, three or more nodes in the set. (In a regular graph, an edge is a set of one or two nodes.)

Given a set of relation schemas, the corresponding hypergraph is the hypergraph where the attributes appearing in the schemas are the nodes, and the sets of attributes in specified by the schemas are the hyperedges.

A hyperedge $H$ is an **ear** if there is another hyperedge $G$ in the hypergraph such that any node in $H$ that is not in $G$ is only in $H$. We say that $G$ **consumes** $H$.

A hypergraph is **acyclic** if, by pulling its ears off one at a time, it can be reduced to just one hyperedge.

Example: consider $R(A,B)$, $S(B,C)$, $T(C,D)$. In their hypergraph, the first and third hyperedges are ears.

---

**Full Reducers**

If the hypergraph of a set of relations is acyclic, it is possible to avoid transmission of dangling tuples. (For simplicity, we'll assume that each relation is at a different site whose name is the same as the relation's name. Note also that the relation name also names the corresponding hyperedge in the hypergraph.)

A plan for temporarily modifying the relations so that they contain no dangling tuples is called a **full reducer**. This plan consists of steps of sending projections from one site to another and computing semijoins there. When the full reducer plan is executed, the semijoins can be sent to one site for final assembly of the join of all the relations.

---

**Computing a Full Reducer**

```python
def full-reducer-generator(hypergraph X):
    if X has more than one hyperedge:
        H = one of the ears of X
        G = a hyperedge that consumes H
        Y = the nodes that are common to G and H
        output “send $\sigma_Y(H)$ to site G and set table G = G \times H”
        full-reducer-generator(X – H) # hyperedge H removed
        output “send $\sigma_Y(G)$ to site H and set table H = H \times G”
```

---

**Notes**

- Since a hypergraph usually has more than one ear, pulling the ears of in different orders results in different full reducers.
- If a step in the full reducer has a projection sent to a site $R$ and no subsequent step in the plan involves sending a projection from $R$ to another site and $R$ is the site where the final assembly of the join is going to take place, the step can be dropped. (The same is true for any other relation that is located at that same site.)
- Proof that full-reducer-generator works is by induction on the number of hyperedges in the hypergraph. (Every tuple sent to the site where the join is assembled appears in the join.)

---

**Distributed Commit**

Maintaining atomicity of transactions is harder in distributed systems. Consider a bank headquarters that must transfer money from an account at one branch to an account at another branch. Headquarters must read the amounts in each branch and then send update commands to both branches. Lots can go wrong.

- A program error might occur at one branch and the other branch may not know about it.
- Contact with one branch may be lost; the branch and headquarters will not know whether that branch did its part of the transaction or not.
Two-phase Commit

Assume several sites are involved in the execution of transaction $T$. One of the sites is designated the coordinator. It is the site that initiates $T$ and decides whether $T$ has successfully completed.

After the coordinator has sent out commands to the other sites and is ready to decide whether the transaction should be committed or aborted, it goes through a two-phase process of getting all the other sites to make the same decision.

- Phase 1 – Find out whether the other sites are ready to commit.
- Phase 2 – Make the decision and inform the others.

Phase One

1. Coordinator writes $<\text{Prepare } T>$ to the log on disc at its site.
2. When write is finished, coordinator sends a $\text{prepare } T$ message to each of the other sites.
3. When a site receives the $\text{prepare } T$ message, it (eventually) decides whether its part of $T$ was successful.
   - If successful, the site ensures that the work it has done will be recoverable if it should fail in the near future and then writes $<\text{Ready } T>$ to the log on its disc and then sends $\text{ready } T$ to the coordinator.
   - If not successful, it writes $<\text{Don't commit } T>$ to its log on disc and then sends $\text{don't commit } T$ to the coordinator.

Phase Two

1. If the coordinator receives a $\text{ready } T$ message from all the other sites, it writes $<\text{Commit } T>$ to the log on its disc and then sends $\text{commit } T$ to all the other sites.
2. If the coordinator receives a $\text{don't commit } T$ message or if a timeout occurs without receiving messages from all the other sites, it writes $<\text{Abort } T>$ to the log on its disc and then sends $\text{abort } T$ to all the other sites.
3. If a site receives $\text{commit } T$, it finalizes the commit to what it has done for $T$ and then writes $<\text{Commit } T>$ on its log on disc.
4. If a site receives $\text{abort } T$, it aborts the work it has done and writes $<\text{Abort } T>$ to its log on disc.

If a Site Fails

When the site comes back up, it checks the last record in the log that was saved on disc.
1. If record was $<\text{Commit } T>$, all is fine.
2. If record was $<\text{Abort } T>$, all is fine. It completes the rollback if necessary.
3. If record was $<\text{Don't commit } T>$, rollback its part of $T$.
4. If record was $<\text{Ready } T>$, contact the coordinator or one of the other sites to find out what the commitment decision was and act accordingly.
5. If none of the above, it assumes the worst and aborts its part of the transactions.

If Coordinator Fails

When it comes back up, it asks all sites for their last log entry regarding $T$. If one of them is $<\text{Commit } T>$ or $<\text{Abort } T>$, it commits or rollbacks accordingly (if it hadn't done so already).

If all the entries were $<\text{Ready } T>$, it continues with step 1 of Phase two. Otherwise it continues at step 2 of Phase two ($\text{aborts } T$).

It might find $<\text{Commit } T>$ or $<\text{Abort } T>$ in another site's log because the other sites might have mutinied when they discovered that the coordinator was down and chose another site to be the coordinator.

How did the mutinous sites decide a new coordinator?

Leader Election

After a timeout period, each site sends its id number to all the other sites. After another timeout period, each site sends to all the other sites the lowest id number it has received. After another timeout period, if the same number of messages is received as before and they all agree on the same id number, the site with that number becomes the new coordinator.

If this process fails (not all the messages in the second round show the same lowest id number, or there are fewer messages because a site went down during the election process), the process repeats.
Resolution of Transaction T

1. The new leader asks all the other sites to report what their last log entry for T was.
2. If a site reports <Commit T>, the leader sends commit T to all sites.
3. If a site reports <Abort T>, the leader sends abort T to all sites.
4. If all reports are <Ready T>, nothing can be done; must wait for original coordinator to come back online. (In practice, a human might intervene, but this is risky.)
5. If not all reports are <Ready T> and none are <Commit T> or <Abort T>, the leader sends abort T to all sites.

Locks

Locks are often used to control access to tables so that transactions will not interfere with each other. There are two kinds.

- **Shared** – used when a transaction only needs to read a table. It allows other transactions to read the table too.
- **Exclusive** – used when a transaction needs to make changes in the table (write to it). No other transaction can be allowed access to the table, not even to read it.

A table can have one exclusive lock or many shared locks at the same time – no mixing of the two.

Distributed Locks

Implementing locks in a distributed system presents new problems. Locking every copy of a table (at different sites) becomes difficult, as each copy would have its own lock and they would have to be synchronized.

One solution is to have one site designated to handle all locking requests for the whole system. This is called centralized locking. For simple situations, this will work, but usually a system where the locks are distributed will be better, faster, more robust.

Cost Model

The main cost in a distributed locking system is the time it takes to send messages over the connecting network. A minimum of three messages must be transmitted for each use of a lock.

- Request that a table at a site be locked (shared or exclusive)
- Acknowledge that the table has been locked as requested
- Request that the lock be released

Primary Copy locking

In the primary copy method, one copy of a table is selected to be the one that is actually locked. Since the primary copy of different tables can be on different sites, this is not the same as centralized locking.

Global Locking

In many schemes, any copy of a table might be locked, and not all copies need to be locked. Suppose that there are n copies of a table. Let s be the minimum number of copies that have to be locked to have the table locked in shared mode. Let x be the number of copies that have to be locked to have the table locked in exclusive mode.

- If \( x + s > n \) and \( x > n/2 \), then locking s copies is enough to lock the table in shared mode.
- If \( x + s > n \) and \( x > n/2 \), then locking x copies is enough to lock the table in exclusive mode.
**Example Policies**

Read-Locks-One; Write-Locks-All – In this policy, $s = 1$ and $x = n$.

Majority Locking – In this policy, $s = x = \lceil (n+1)/s \rceil$.

---

**Peer-to-Peer Networks**

A peer-to-peer network is a collection of sites that
- can join or leave the network at any time.
- are connected over a general-purpose network such as the Internet.
- have no sites that control other sites.
- share resources.

---

**Distributed-Hashing Problem**

Suppose a database table is distributed over an entire peer-to-peer network, and there is no central index of where everything is. How can the cooperating sites make it possible to efficiently retrieve a record (or row of the table) given its key? Think of the network as a big, sparsely populated hash table where each site in the network is one non-empty bucket. Starting from any node in the network, how can we find the node that holds the record we want even though we don’t know where all the nodes in the network are?

---

**Chord Circles**

Each node is assigned a number by applying a hash function to its ID. The number will be interpreted as the index number of a bucket in a virtual hash table.

Each node knows its predecessor and successor in the network based on the ascending order of their assigned numbers. The node with the highest number and the node with the lowest number know each other as if the nodes were connected in a circle.

Using a hash function that maps keys into the given index range, the records that normally be stored in virtual bucket i are actually stored at node j where j is the smallest node number that is greater than or equal to i. (If no such j exists, they are at the node with the lowest number.)

---

**Finger Tables**

In addition to knowing its predecessor and successor, each node also knows the location of a handful of other nodes; these nodes are identified in a finger table. For node i, the j’th entry in its finger table is node k, where k is the smallest node number that is greater than or equal to $i + 2^{j-1}$.

To find records with hash key n, starting from node i, if i < n then go to node j where node j is the successor to node i, go to node j. Otherwise, consult node i’s finger table to find node j that is furthest down in the table that is less than n, go to node j.

[The nodes wrap around the circle; think of each node j as also node m + j, where m is the largest index number allowed.] [modular arithmetic]

---

**Crashing the Party**

If a site wants to join the network, it must know where one of the nodes in the network is. Suppose that the hash function applied to the site’s ID gives a hash key of $i$ and the node it knows about is node $j$. The site asks node $j$ which node holds the records with hash key $i$. Suppose that the answer is node $k$. Then node $k$ will tentatively be the site’s successor (if $k \neq i$). The site will be node $i$.

(If $k = i$, the site contacts the other node $i$ and they keep in touch, sharing the responsibility of being node $i$.)
Joining the In Crowd

Node j found node k by the same process it would use to find the location of records with hash key i.

The new node i makes node k its successor and informs node k. Node k reports back its predecessor, node h. If i < h < k, node i makes node h its successor instead of node k and informs node h. This repeats until a node p is found where p < i. Node p becomes node i's predecessor, and informs node p. Records with hash key i are transferred from node i's successor to node i. The finger tables are revised.

This process of establishing a node's predecessor and successor (stabilization) has to be repeated periodically by all nodes to keep the network up-to-date about any changes in the network.

Dropping Out

If a node decides to leave the network, it informs its predecessor and successor and transfers its records to its successor.

Redundancy is used to protect against data loss due to nodes failing. One strategy is to store each record at the correct node and at that node's predecessor and successor. Another strategy is to have nodes agree to form groups of three or more to share records so that if one goes down, the others can cover for it.