7/8. Distributed Databases

Slides adapted from Pearson Ed.

Definitions

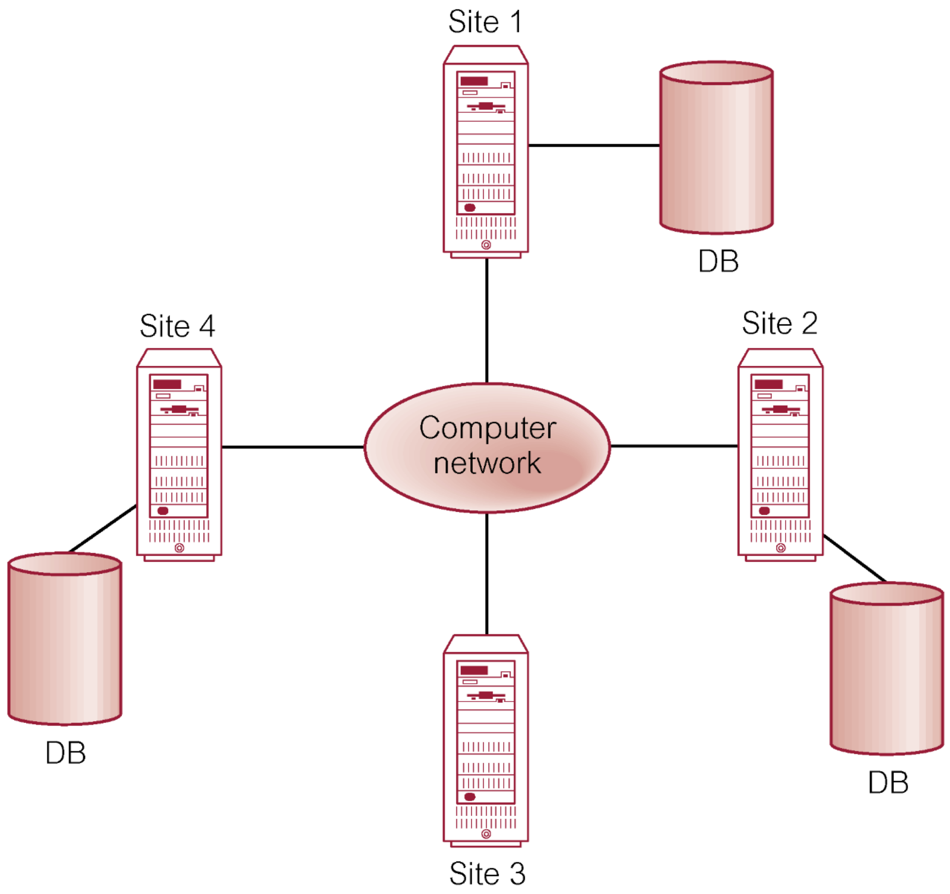
- **Distributed database**: a database (logically interrelated collection of shared data and description of said data) that is physically split across a computer network.
- **Distributed DBMS (DDBMS)**: software that facilitates the management of a distributed database, including making aspects of the distribution *transparent* (hidden from the user).

DDBMS structure

- Data is physically stored in multiple sites in **fragments**.
- Each site is managed by a DBMS.
- DBMSs communicate over the network with other sites / DBMSs.

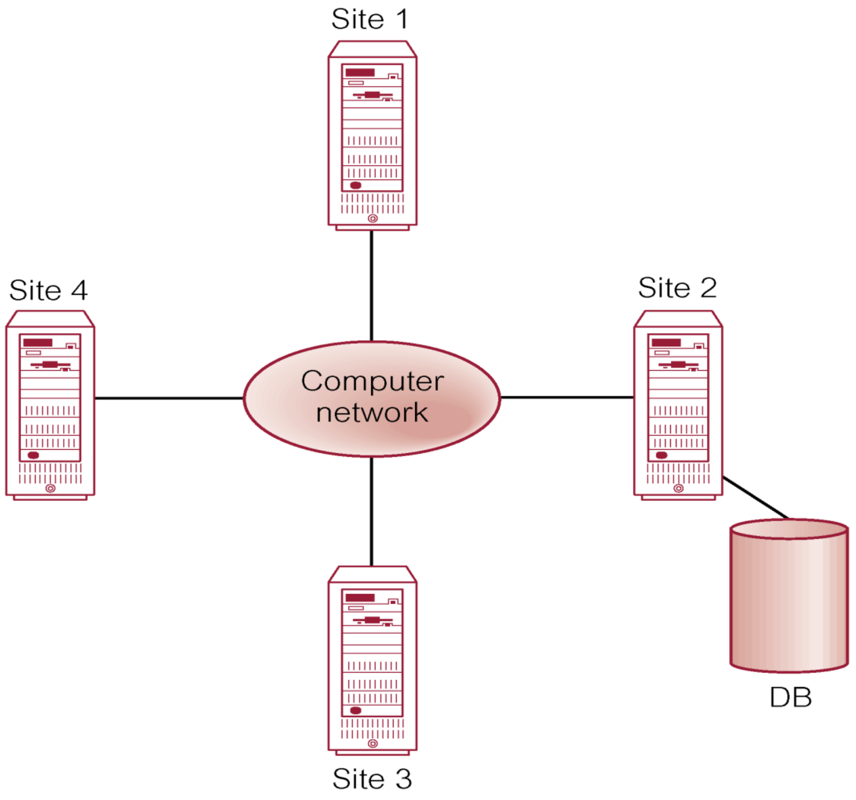
Example DDBMS

- Fragments of a database split across multiple sites.
- Not all sites necessarily contain database data.



Contrast with distributed processing

- Data is still centralized at a single site.



Main advantages

- Many sites hosting a (part of a) database makes the system more reliable and responsive and easy to set up.
  - Scalable growth.
  - Horizontal growth: Use inexpensive hardware to manage a large amount of data and usage.
- DDBMSs can take advantage of heterogeneous DB architecture and geographically specialized access.
- Re-decentralization!

Main disadvantages

- More complicated / difficult to implement equivalent features of a regular DBMS such as transaction management / ACID, integrity checks, data model management, etc.
- Costs of network traffic (time / money).

Heterogeneous DDBMS

- Particular type of DDBMS featuring multiple DBMS products.
  - As opposed to homogeneous DDBMS.
- Presents special difficulties in terms of translating queries, schema, etc.
- Database Interoperability Consortium (DBIOP) – attempt to define ways for different DBMSs to interact.
- Multidatabase systems (MDBS): a DBMS added on top of independent DBMSs that presents a unified interface to user.

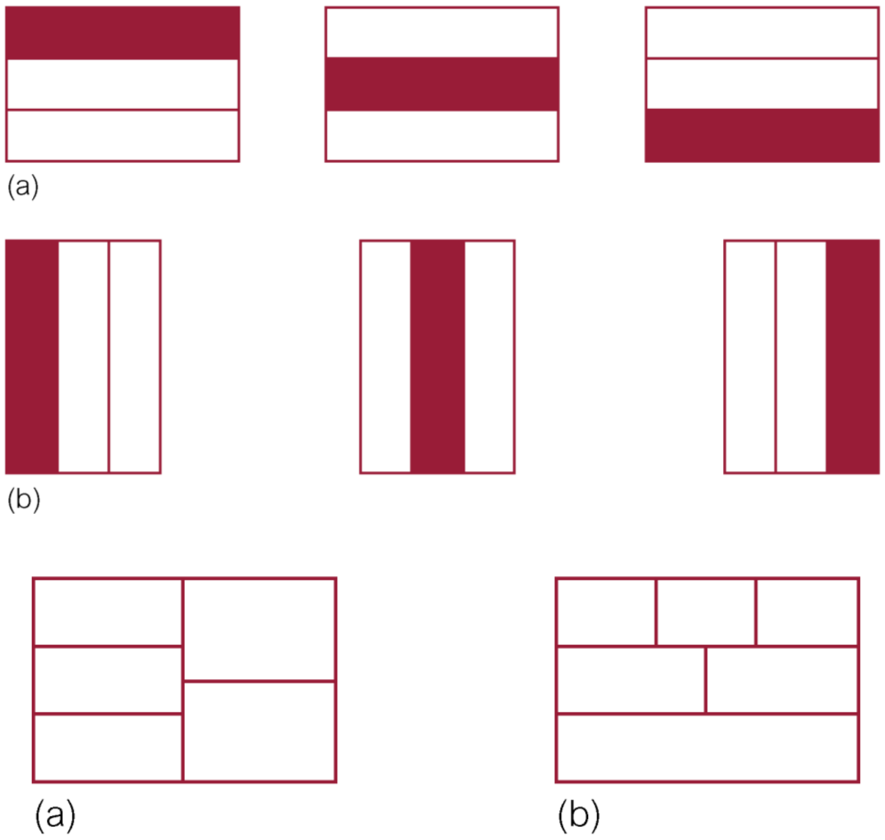
Design considerations for distributed DBs

Distributed DBs: design

- Additional design decisions needed for distributed databases:
  - Fragmentation: how to divide up database into smaller chunks.
  - Site allocation of fragments: where to store each fragment.
  - Fragment replication: whether to store multiple copies of a fragment or fragments.

Types of fragmentation

- Four types of fragmentation:
  - Horizontal
  - Vertical
  - Mixed
  - Derived
- Fragments break down data into disjoint sets that cover the complete relations.
- Must be able to reconstruct original relation from fragments.



Horizontal fragmentation

- Fragments = subsets of tuples
- Defined by: selection (σ) op
- Recombine fragments using: set union (∪) op
- Need a set of selection conditions that cover the whole set of tuples.
- Example:
  - $P1 = \sigma_{type='House'}(PropertyForRent)$
  - $P2 = \sigma_{type='Flat'}(PropertyForRent)$

Vertical fragmentation

- Fragments = subsets of a table's attributes
- Defined by: projection (Π) op
- Recombine fragments using: natural join (⋈)
  - Need to keep primary key in each vertical fragment to do this.
  - Perform natural join on the primary key.
- Example:
  - $S1 = \Pi_{\text{staffNo, position, sex, DOB, salary}}(\text{Staff})$
  - $S2 = \Pi_{\text{staffNo, fName, lName, branchNo}}(\text{Staff})$

Mixed fragmentation

- Can mix both types of fragmentation (have a vertical fragment that is then horizontally fragmented or vice versa).
- Sequence of selections and projections.
- Reversed using sequence of joins and unions.
- Example:
  $S_1 = \Pi_{\text{staffNo, position, sex, DOB, salary}}(\text{Staff})$
  $S_2 = \Pi_{\text{staffNo, fName, lName, branchNo}}(\text{Staff})$

  $S_{21} = \sigma_{\text{branchNo ='B003'}}(S_2)$
  $S_{22} = \sigma_{\text{branchNo ='B005'}}(S_2)$
  $S_{23} = \sigma_{\text{branchNo ='B007'}}(S_2)$

Derived horizontal fragmentation

- Horizontally (tuple-based) fragment one table based on foreign key or characteristics of parent table.
- Defined by: semijoin (▷) op
- Recombine using: set union (∪) op
- Example: Segment PropertyForRent table based on branch of employee managing it:
  P_i = PropertyForRent $\triangleright_{branchNo=i}$ Staff

Fragmentation considerations

- Fragmentation and allocation should be done in a way that maximizes locality and minimizes communication needs.
- Need to analyse/understand how a database is typically used.
  - How is the data typically queried?
  - What data is requested at the same time as other data?

Data allocation strategies

- Possible distribution schemes:
  - **Centralized**: database and DBMS at one site that handles requests from all users.
  - **Partitioned**: database split into disjoint fragments located at various sites, hopefully closest to requesting applications. No replication.
  - **Complete replication**: multiple sites each containing a complete database, identical to each other.
  - **Selective replication**: database containing some partitioning, replication, and centralization as needed. Flexible and popular.

Considerations for database systems

- **Locality of reference**: data is located close to users/applications that frequently request it.
- **Reliability and availability**: data is frequently available and able to be accessed.
- **Performance**: many requests are able to be handled concurrently with low response times.
- **Storage costs**: less storage space needed.
- **Communication costs**: less network traffic needed to transfer data.

Comparison

Scheme	Locality of reference	Reliability & availability	Performance	Storage costs	Communication costs
Centralized	Low	Lowest	Can be insufficient	Lowest	Highest
Fragmented	Can be high	Improvement for system but not for individual items	Potential improvement	Lowest	Potential improvement
Complete replication	Highest	Highest	Best for read	Highest	Expensive update, good reads
Selective replication	Can be high	Improvement for system and replicated items	Potential improvement	Average	Potential improvement

Features of distributed DBMSs

DDBMS features

- Compared to a DBMS managing a centralized database, a DDBMS faces extra challenges:
  - Ability to communicate effectively over networks.
  - Ability to manage multiple data schemas, possibly collated into a single global schema.
  - Processing distributed queries.
  - Managing distributed transactions.
    - Concurrency control,
    - Recovery.

Transparency

- **Transparency** refers to when implementation details are hidden (invisible) to the user.
  - Misleading name.
  - But often a desired feature from the user's point of view.
- Extra work/complexity for DBMS software to abstract out these details.

Transparencies in DDBMS

- DDBMSs may or may not implement the following transparencies:
  - Distribution transparency: hiding signs of data distribution from the user.
    - Multiple types/levels: Fragmentation, location, replication, local mapping, and naming transparencies.
  - Transaction transparency: automatically handling the implementation of performing transactions on distributed systems.
    - Concurrency transparency; failure transparency.
  - Performance transparency: DDBMS is successfully able to prevent degradations in performance (throughput, response time) detectable to the user.
  - DBMS transparency: DDBMS successfully hides the usage of multiple different DBMSs from user.

Distribution transparency

- Distributed database looks like a single entity to the user.
- Different levels of transparency:
  - Fragmentation transparency: user does not know data is fragmented.
    - Queries look e.g. like normal SQL queries for RDBMS.
  - Location transparency: user knows data is fragmented but does not know where/how they are stored.
    - Queries look e.g. like SQL queries with fragment names used instead of relations.
  - Local mapping transparency: no abstraction (rare).
    - Queries must reference fragment names and storage locations.

Distribution transparency (cont.)

- Other dimensions of transparency relevant at various levels:
  - Replication transparency: user does not know that replicas of database (fragments) exist and must also be updated.
  - Naming transparency: user doesn’t manually have to ensure names don’t conflict in global schema.

Transaction transparency

- All distributed transactions (transactions accessing data at multiple locations) maintain database integrity/consistency.
- Additional complexities involved in implementing distributed transactions:
  - Distributed transaction accesses data stored at more than one location.
  - Therefore, transactions must be divided into subtransactions, one for each site that has to be accessed.
  - DDBMS must coordinate actions/commit of subtransactions to ensure the atomicity of global transactions.

Example: distributed transaction

- Transaction T wants to print names of all staff. Staff data containing staff names has been split into three fragments by branchNo, and stored at sites 3, 5, and 7.
- Need to split T into three transactions sent to each site.
- Work is naturally parallelized:

Time	T_{s_3}	T_{s_5}	T_{s_7}
t_1	begin_transaction	begin_transaction	begin_transaction
t_2	read(fName, lName)	read(fName, lName)	read(fName, lName)
t_3	print(fName, lName)	print(fName, lName)	print(fName, lName)
t_4	end_transaction	end_transaction	end_transaction

Concurrency transparency

- DDBMS guarantees concurrent transactions do not introduce inconsistency.
- Similar concepts of serializability, except multiple sites must coordinate to converge on a *single* serial schedule.
- Similar techniques of locking and timestamping, extended to work with distributed systems i.e. preferably in a decentralized manner.
- Existence of replicas creates problems for consistency when not all replicas are available.
- More on this later.

Failure transparency

- DDBMS is robust to failures.
- With distributed databases, failures include additional scenarios related to network communication issues.
- Ensuring recoverability: Like with centralized databases, need to carefully control commit procedure.
  - Need to extend commit protocol to work with multiple subtransactions.
  - DDBMS must ensure that all subtransactions have completed successfully before recording a final COMMIT for global transaction – **two-phase commit (2PC)** protocol is most common.

Performance transparency

- DDBMS must not suffer performance degradation compared to a centralized database.
- Must perform query processing to optimize query strategy, taking into account **communication costs**, which may dominate other costs (disk access costs, processing/computation costs).
- Depending on decisions about which site to process data and what data needs to be sent over the network, dramatic differences in query completion times.

Preview: response time depends on strategy

STRATEGY	TIME
(1) Move Client relation to London and process query there	16.7 minutes
(2) Move Property and Viewing relations to Glasgow and process query there	28 hours
(3) Join Property and Viewing relations at London, select tuples for Aberdeen properties and, for each of these in turn, check at Glasgow to determine if associated maxPrice > £200,000	2.3 days
(4) Select clients with maxPrice > £200,000 at Glasgow and, for each one found, check at London for a viewing involving that client and an Aberdeen property	20 seconds
(5) Join Property and Viewing relations at London, select Aberdeen properties, project result over propertyNo and clientNo, and move this result to Glasgow for matching with maxPrice > £200,000	16.7 minutes
(6) Select clients with maxPrice > £200,000 at Glasgow and move the result to London for matching with Aberdeen properties	1 second

DBMS transparency

- DDBMS abstracts out details for working with different types of DBMSs (heterogeneous databases).

Note on transparencies

- Transparencies not universally agreed on as desirable.
- Transparencies provide a simpler experience for users.
- But may make database administration difficult and may make each site less autonomous/independent.

Date’s 12 rules for DDBMS

Overarching principle: To user, a distributed DB should look exactly like a non-distributed DB.

1. Sites are autonomous. Data stored at a site is managed at that site and no network communication is needed to query that data.
2. No reliance on a central site (for e.g. transaction management, deadlock detection, query optimization, maintaining global system catalogue, etc.)
3. Continuous operation. No downtime needed.

Date’s 12 rules (cont.)

4. Location independence (transparency).
5. Fragmentation independence (transparency).
6. Replication independence (transparency).
7. Distributed query processing implemented.
8. Distributed transaction processing implemented.

9-12. Hardware / OS / network / DBMS independence. DDBMS works seamlessly across heterogeneous hardware, operating systems, networks, and types of databases.

Distributed query optimization

Distributed query example

Three tables at two sites:

- Property(propNo, city) 10,000 records in London
- Client(clientNo,maxPrice) 100,000 records in Glasgow
- Viewing(propNo, clientNo) 1,000,000 records in London

Query (find Aberdeen properties viewed by clients interested in properties worth at least £200,000):

- ```
  SELECT p.propNo
  FROM Property p INNER JOIN
          (Client c INNER JOIN Viewing v ON c.clientNo = v.clientNo)
  ON p.propNo = v.propNo
  WHERE p.city=‘Aberdeen’ AND c.maxPrice > 200000;
  ```
  ```

Example: assumptions

- Assume:
  - Each tuple in each relation is 100 characters long.
  - 10 renters with maximum price greater than £200,000.
  - 100,000 viewings for properties in Aberdeen.
  - Computation time negligible compared to communication time.

Example – cost of various strategies

STRATEGY	TIME
(1) Move Client relation to London and process query there	16.7 minutes
(2) Move Property and Viewing relations to Glasgow and process query there	28 hours
(3) Join Property and Viewing relations at London, select tuples for Aberdeen properties and, for each of these in turn, check at Glasgow to determine if associated maxPrice > £200,000	2.3 days
(4) Select clients with maxPrice > £200,000 at Glasgow and, for each one found, check at London for a viewing involving that client and an Aberdeen property	20 seconds
(5) Join Property and Viewing relations at London, select Aberdeen properties, project result over propertyNo and clientNo, and move this result to Glasgow for matching with maxPrice > £200,000	16.7 minutes
(6) Select clients with maxPrice > £200,000 at Glasgow and move the result to London for matching with Aberdeen properties	1 second

Query optimization considerations

- Extra considerations for distributed databases:
  - Multiple sites, so multiple options for data transfer.
  - Which sites should do which processing and when?

Distributed query optimization for RDBMSs

- Given a SQL query, can convert to a relational algebra tree and optimize it as normal.
- Then replace simple leaf nodes with subtrees reflecting how data was fragmented and should be recombined.
  - Massage this RAT to eliminate redundant/useless work.
- Use database statistics (if available) to decide on a detailed strategy.
- Distribute operations to remote sites where they can optimize locally.

Distributed query optimization - example

Query:

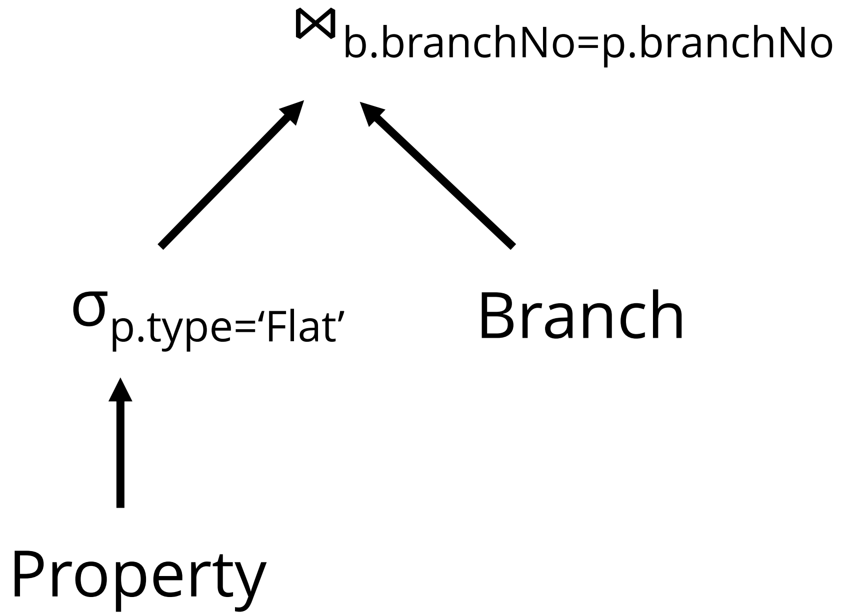
- SELECT * FROM Branch b, PropertyForRent p
  WHERE b.branchNo = p.branchNo AND p.type = ‘Flat’;

Fragmentation:

- P1: $\sigma_{\text{branchNo = ‘B003’ ^ type = ‘House’}}$ (PropertyForRent)
- P2: $\sigma_{\text{branchNo = ‘B003’ ^ type = ‘Flat’}}$ (PropertyForRent)
- P3: $\sigma_{\text{branchNo != ‘B003’}}$ (PropertyForRent)
- B1: $\sigma_{\text{branchNo = ‘B003’}}$ (Branch)
- B2: $\sigma_{\text{branchNo != ‘B003’}}$ (Branch)

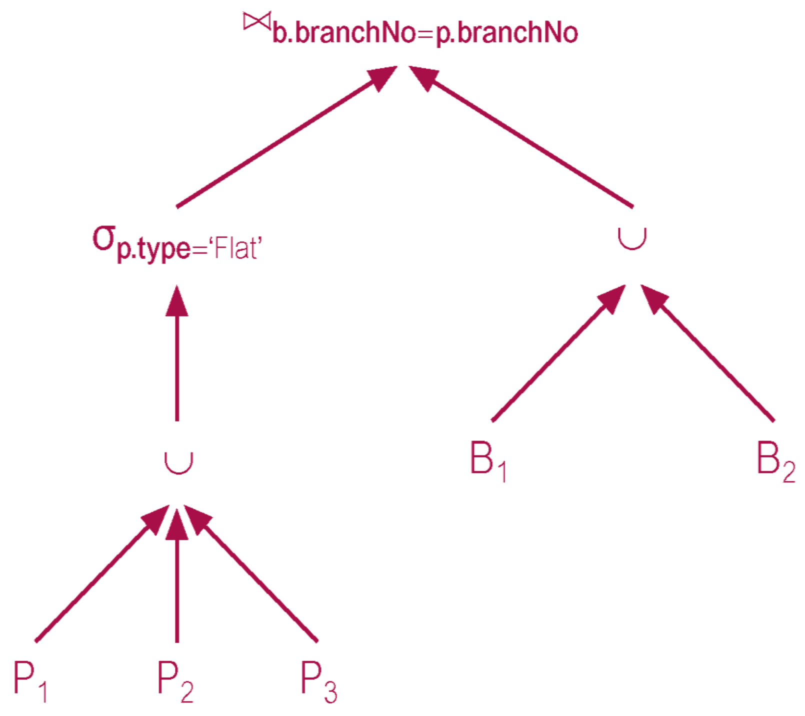
Example – simple relational algebra tree

- After optimization.



Example – replacing leaves with fragments

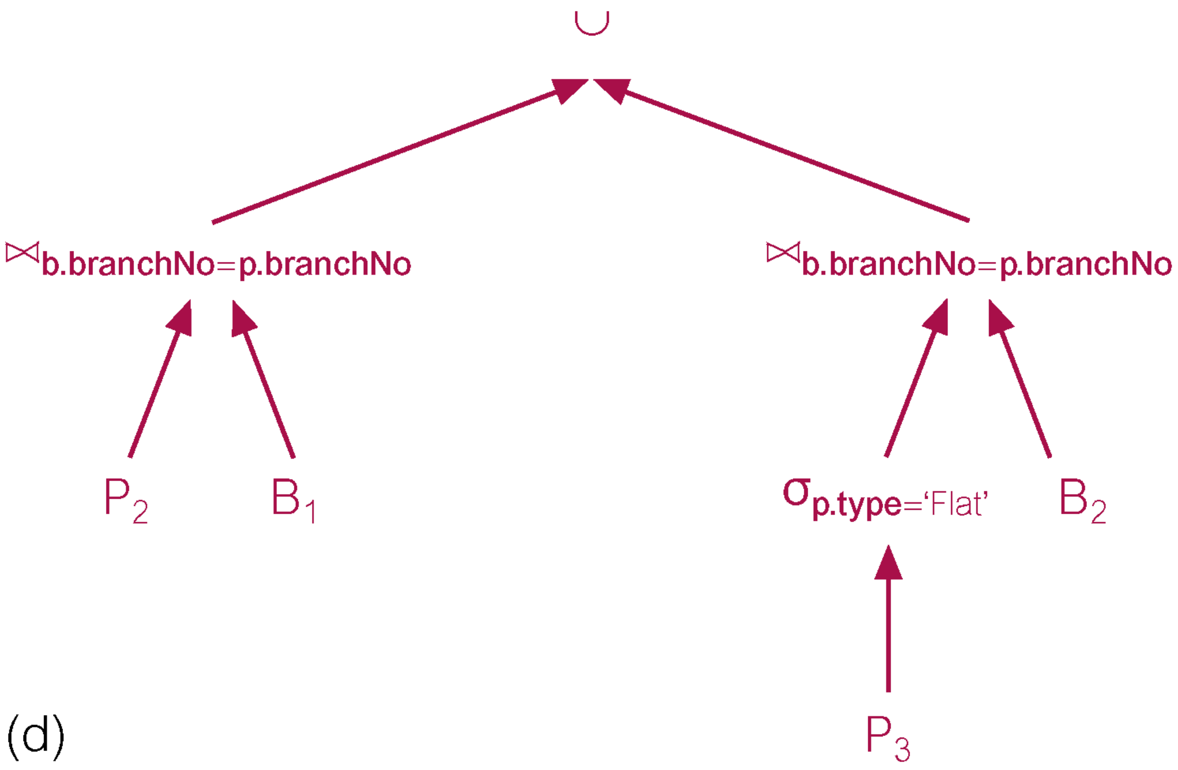
- Horizontal fragments (derived and otherwise) are combined using union.
- Vertical fragments are combined using natural join.



(a)

Example – further massaging

- By definition, P1 doesn't contain any Flat type properties.
- By definition, P1/P2 can only match to B1 table (tuple).
- By definition, P3 can only match to B2 table.



Example – the rest

- Still need to decide on more detailed execution strategy.
- Use statistics to estimate data transfer time for various strategies.
- Once plan is decided, send to individual sites to execute their part.

Distributed query optimization: summary

- Important for performance transparency to have a good strategy for executing distributed queries.
- Different parts:
  - Convert SQL query to fragments.
  - Eliminate redundant/useless work.
  - Use statistics to pick strategy.
  - Optimize locally.

Distributed transactions

Review: Transaction transparency

- Transaction transparency = distributed database maintains consistency (ACID properties) in the presence of transactions.
- Two parts: concurrency transparency and failure transparency.
- Need to be careful about timing of commits in distributed transactions.

Concurrency in distributed databases

- Similar concepts of serializability, extended to make sure schedules distributed across multiple sites are compatible with a single serial schedule (i.e. executions at individual sites must all be equivalent to the *same* serial ordering).
- Similar techniques of locking and timestamping, extended to work with distributed systems i.e. preferably in a decentralized manner.
  - Especially important for locking and deadlock detection.

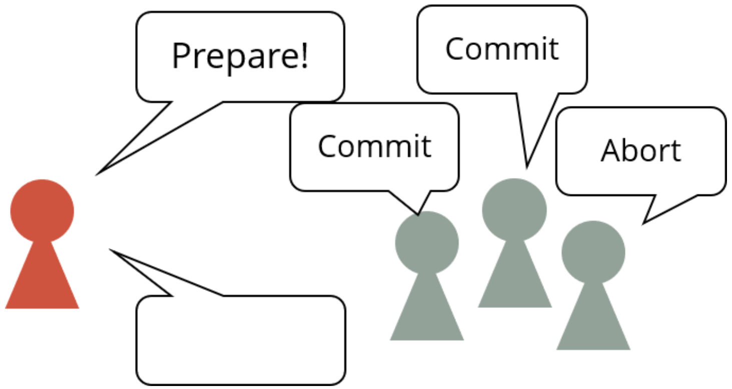
Concurrency: challenges with replication

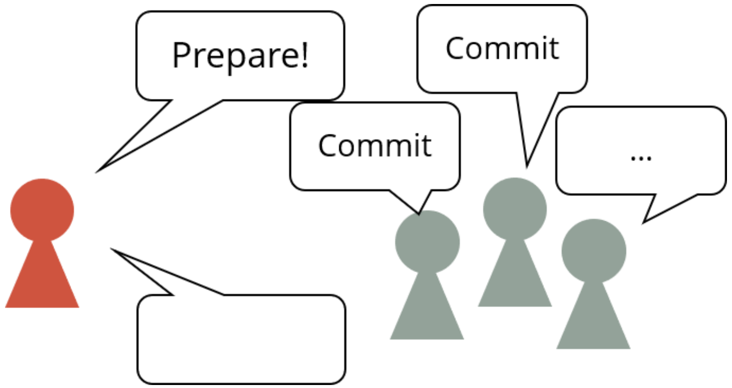
- Updating multiple replicas is challenging as they may not all be available at one time (busy, down, etc.).
- Two options:
  - Wait to complete update/transaction until all sites are reachable (could take a while).
  - Update sites that are available, or allow updates to happen asynchronously. **Eventual consistency**: system may be in an inconsistent state from anywhere from a few seconds to hours, but will eventually become consistent.

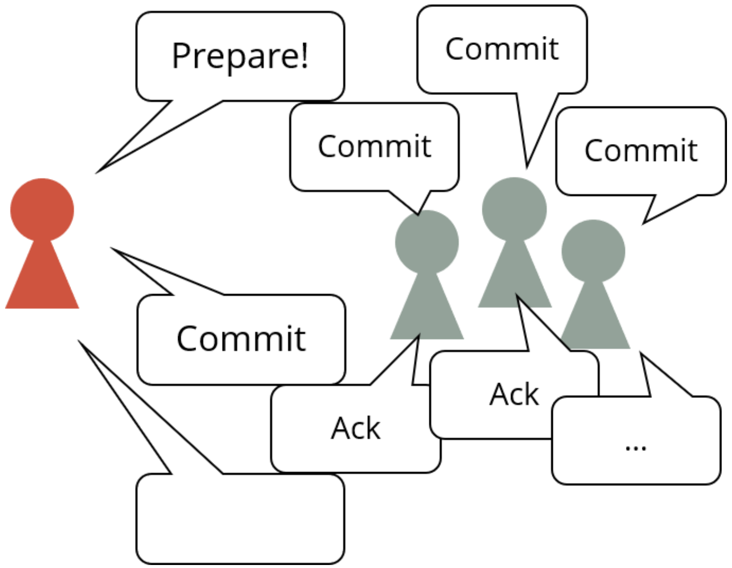
Two-phase commit (2PC)

- Coordinator sends out a PREPARE message and waits for other participants to respond with a COMMIT/ABORT decision (voting phase).
- Based on responses, coordinator will send out a final instruction to either COMMIT or ABORT and waits for participants to ACKNOWLEDGE (decision phase).

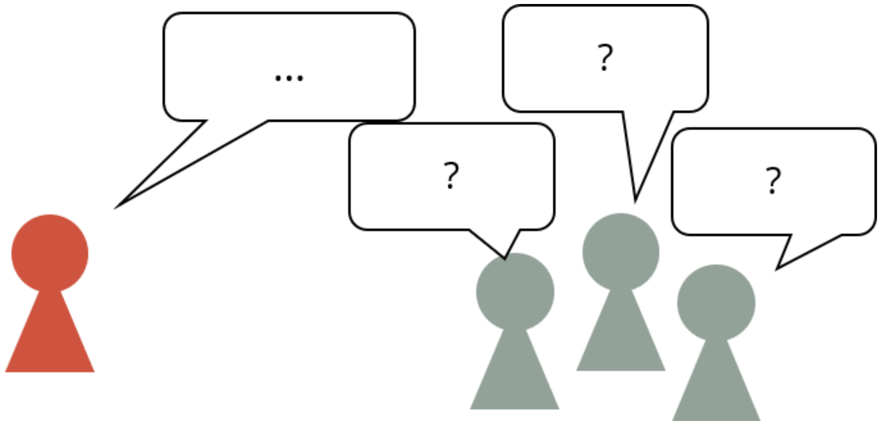
Examples – coordinator behavior

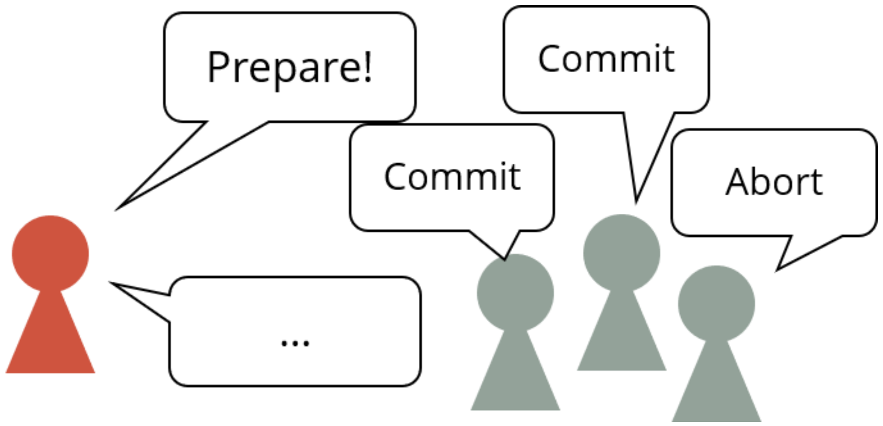






Examples – participant behavior





2PC is a blocking protocol

- Under certain conditions, some sites may be unable to proceed when a node goes down.
- Can get around this with more complicated protocols (e.g. 3-phase commit (3PC)).

Recovery

- Machines can fail in the middle of the commit procedure.
- Need to be able to recover to a consistent state upon restart.
- Same as with simple databases, logging is the tool needed to ensure this.

Log entries

- Coordinator writes:
  - Begin Commit
  - Commit Decision OR Abort Decision (depending on vote)
  - End of Transaction (committed transactions)

- Participants write:
  - Ready Commit OR Abort (depending on transaction status)
  - Final Commit OR Abort (vote result)

In case of failure: coordinator

Contents of log	Action to take
Transaction started, but no commit log entries exist	
Begin Commit only	
Abort/Commit Decision entered	
End of Transaction entered	

In case of failure: coordinator

Contents of log	Action to take
Transaction started, but no commit log entries exist	Begin commit procedure with PREPARE.
Begin Commit only	Resend PREPARE and collect votes.
Abort/Commit Decision entered	Request acknowledgement of decision.
End of Transaction entered	Make sure results of transaction are in database.

In case of failure: participant

Contents of log	Action to take
Transaction started, but no commit log entries exist	
Ready Commit/Abort entered	
Final Commit/Abort entered	

In case of failure: participant

Contents of log	Action to take
Transaction started, but no commit log entries exist	Abort.
Ready Commit/Abort entered	Request final decision and block on hearing back, or proceed with abort.
Final Commit/Abort entered	Make sure results of transaction are/are not in database.

Distributed transactions summary

- Still possible to guarantee ACID properties with distributed databases.
- But procedures and coordination get more complicated.
- Availability of database is affected.
- We will talk about this more later.

Case study: DDBMS functionality in Oracle

Fragmentation

- Oracle doesn’t support fragmentation, but does support setting up and accessing remote databases (which can be seen as a “fragment” of a larger database).
  - Database administrator (DBA) must manually design and define these “fragments”.
  - No fragmentation transparency, but location transparency.

Linking remote databases

URL consisting of database name and domain name

```
CREATE PUBLIC DATABASE LINK
    RENTALS.GLASGOW.NORTH.COM;
SELECT * FROM Staff@RENTALS.GLASGOW.NORTH.COM;
UPDATE Staff@RENTALS.GLASGOW.NORTH.COM
    SET salary=salary*1.05;
```

Use @ to refer to tables

- Link acts as a remote login to remote database.
- Oracle is capable of establishing a link to both Oracle and non-Oracle databases.

Types of transactions in a distributed database

- Remote SQL statements: query data that fully exists at a remote site.
- Distributed SQL statements: a query that requires data located at multiple sites.
- Remote transactions: a transaction that can be fully run on a single remote site and must be atomic.
- Distributed transactions: a transaction that requires multiple sites to update their data, and must be atomic.
- Oracle supports multisite queries and transactions using 2PC.

Integrity constraints

- Some constraints can be maintained across databases using triggers.
- Otherwise, ability to define constraints across multiple databases is not as powerful as with a centralized database.

Heterogeneous distributed databases

- Communication and translation of schema and queries is accomplished through gateways installed on Oracle site or non-Oracle site.