Scaling Databases

Distribution, Parallelism, Virtualization

COSC 404 – Database System Implementation

Scaling Database Systems

Scaling a database system involves handling:

- larger data sets and queries that involve more data
- larger number of users/transactions/queries
- handling failures and concurrency issues when supporting more users and servers

Scaling is achieved by adding more servers (i.e. cluster) and replicating/distributing/partitioning data across those servers that handle the data and query load.

There are a variety of architectures and approaches.

★ Performance Measures for Parallel and Distributed Systems

Throughput - the number of tasks that can be completed in a given time interval.

Response time - the amount of time it takes to complete a single task from the time it is submitted.

Speedup - how much faster a fixed-sized problem can be executed on hardware that is *N*-times faster.

- *speedup = time on basic system / time on N-times faster system*
- Speedup is *linear* if equation equals N.

Scaleup - is the ability of a system *N* times larger to perform a job *N* times larger, in the same time as the original system.

- *scaleup* $= \dfrac{\textit{time to execute small problem on small system}}{\textit{time to execute large problem on large system}}$
- Scale up is *linear* if equation equals 1.

Factors Limiting Speedup and Scaleup

Speedup and scaleup are often sublinear due to:

- **Startup costs**: Cost of starting up multiple processes may dominate computation time if the degree of parallelism is high.
- **Interference**: Processes accessing shared resources (e.g. system bus, disks, or locks) compete with each other and spend time waiting on other processes, rather than performing work.
- **Skew**: Increasing the degree of parallelism increases the variance in service times of parallel executing tasks. Overall execution time determined by ***slowest*** of executing tasks.

Parallel Performance Measures

Question: How many of the following statements are true?

- i) Response time is how long it takes to complete a given task from the time it is submitted.
- ii) Throughput is the rate at which tasks can be completed.
- iii) Interference is one factor that can limit scaleup.
- iv) When a company wants to grow its database, scaleup is an important factor.

A) 0

B) 1

C) 2

D) 3

E) 4

Parallel Database Systems

A ***parallel database system*** consist of multiple processors and storage connected by a fast interconnection network.

Parallel database systems are used for:

- storing large volumes of data
- processing time-consuming decision-support queries
- providing high throughput for transaction processing

Parallel execution occurs *within* a system in the form of exploiting parallelism available in CPUs and graphics cards.

Distributed Database System

A ***distributed database system*** (DDBS) is a database system distributed across several network nodes that appears to the user as a single system.

A DDBS processes complex queries by coordinating among the individual nodes. Processing may be done at a site other than the location of query submission. This requires cooperation on transaction management, concurrency control, and query optimization.

Parallel and distributed databases have many features in common and the line between them is not always clear. One main difference is that a distributed system is designed to be ***physically/geographically distributed*** where a parallel DBMS may be in a single server/data center.

Parallel and Distributed Databases Advantages and Disadvantages

Advantages:

- **PERFORMANCE, availability, reliability**
- Local autonomy
- Reflects organization structure
- Economics (smaller systems)
- Less network traffic compared to centralized

Disadvantages:

- Complexity
- Lack of control
- Cost
- Security
- More complex database design

Parallel/Distributed Architectures

Shared memory - processors share a common memory.

- Memory shared using a bus allowing fast communication between processors. Good for small parallel systems.
- Architecture is not scalable since the bus is a bottleneck.

Shared disk - processors share a common disk.

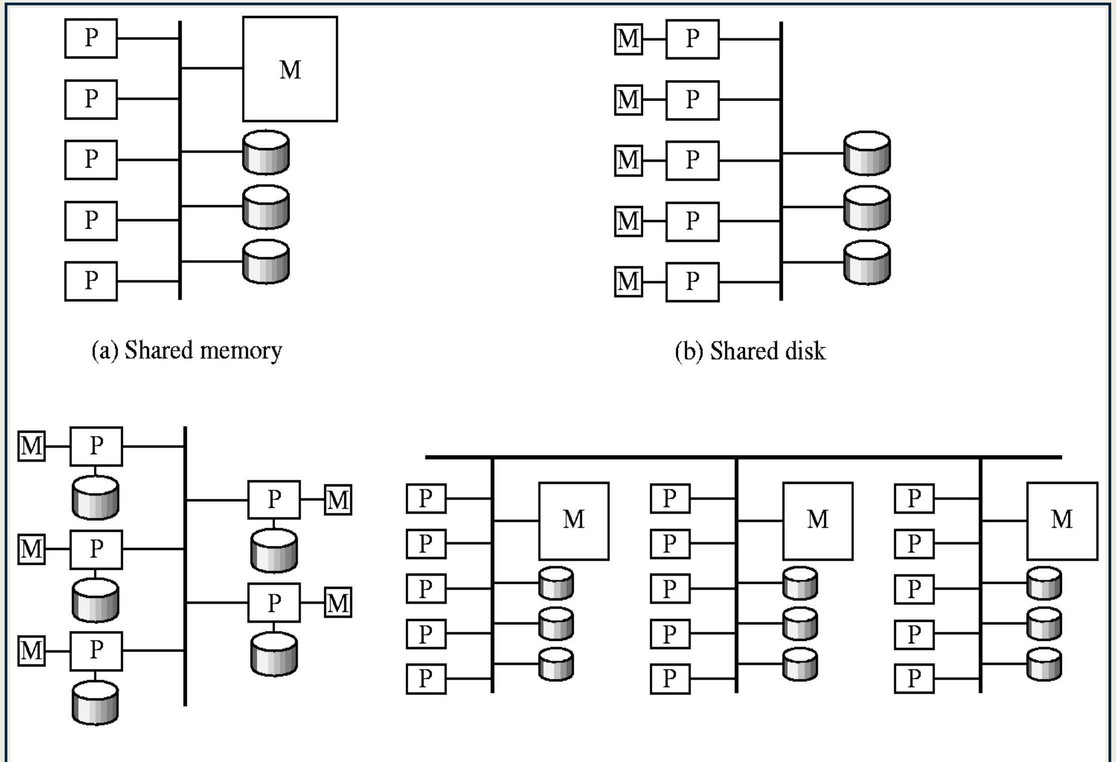
- Processors shared data on disk but have private memories.
- Bottleneck at disk system instead of bus. Slower data sharing.

Shared nothing - processors share no memory or disks.

- A node consists of a processor, memory, and one or more disks. Nodes communicate over the network.
- Can be scaled up to thousands of processors.

Hierarchical - hybrid combination of the above architectures.

Parallel/Distributed Architectures



Parallel/Distributed Architectures

Question: How many of the following statements are true?

- i) Shared memory is used when servers are in different locations.
- ii) Shared nothing is the architecture that is the least popular.
- iii) MongoDB assumes/uses a shared disk architecture.
- iv) The shared disk architecture is the hardest to implement.

A) 0

B) 1

C) 2

D) 3

E) 4

Types of Database Parallelism

A database can exploit a parallel hardware system by:

- **Partitioning** - dividing the data across hardware to allow for parallel I/O and query processing.
- **Interquery Parallelism -** executing multiple queries concurrently using the parallel hardware resources.
- **Intraquery Parallelism** - executing operators of a query plan in parallel or parallelizing individual operators.

Distributed Data Storage
Replication and Partitioning

A key decision in a parallel/distributed database is how to allocate the data across nodes.

This allocation involves both ***replication*** and ***partitioning***:

- **Replication -** system maintains multiple copies of data stored at different sites for faster retrieval and fault tolerance.
- **Partitioning -** relation is partitioned into several fragments/partitions stored in distinct sites.
- **Replication and partitioning** - relation is partitioned into several partitions and system maintains several identical replicas of each partition.

Data Replication Discussion

Replication is good for reads and bad for writes!

Advantages of Replication:

- **Availability** - failure of a site containing a relation does not result in unavailability if replicas exist.
- **Parallelism** - queries on a relation may be processed by several nodes in parallel.
- **Reduced data transfer -** relation is available locally at each site containing a replica of it.

Disadvantages of Replication:

- **Increased update cost** - each replica must be updated.
- **Increased complexity of concurrency control** - concurrent updates to distinct replicas may lead to inconsistent data.
- **Increased space requirements** - more storage is needed.

Maintaining Consistency with Replication: CAP Theorem

The ***CAP Theorem*** (Brewer 2000) proves that a distributed system can have only two of these three properties: consistency, availability, and partition-tolerance.

In a large system, partitions cannot be prevented, so must sacrifice either availability or consistency.

Many new NoSQL databases select availability over consistency which means that the replicas are not always consistent in time, which is called weak or ***eventual consistency***.

- Strong consistency – all replicas same value at end of update
- Weak consistency – may take some time for all replicas to become consistent

BASE Properties – Not ACID

In eventually consistent systems, the ACID properties do not hold. We may consider these systems to have BASE properties:

Basically **A**vailable

- If server is accessible, can do reads and updates (even if have network partition). Availability at the cost of consistency.

Soft state

- Each replica may have different values (due to partitioning or time delay in update propagation).

Eventually consistent

- Replicas are not consistent at instance of update but will become consistent eventually as updates are propagated and conflicts resolved.
- Merging inconsistent updates is still a challenge.

Primary/Secondary Configuration for Handling Replication and Ensuring Consistency

In a ***primary/secondary configuration***, one primary server is responsible for updates to each partition and sends the updates to the secondary servers that contain copies of the partition.

- ***Primary copy ownership*** – one site owns data, perform updates on that site, and updates are sent out to subscribers who update their replicas. These updates may be sent out by shipping the log to the secondary sites.
- The primary node is read/write. The secondary nodes are read only. This requires a way to specify a read-only transaction (e.g. set at connection or statement level before executing query) so that it can be processed by a secondary node.

Primary/Primary Configuration for Handling Replication and Consistency

In a ***primary/primary configuration***, more than one server is able to perform updates on a given data item. This requires co-ordination by the primary servers.

Techniques:

- Any update must be "approved" by all (or a majority) of the primary servers. This approval may be done before commit (online) using a distributed algorithm (e.g. two phase commit).
- Updates may be allowed on multiple servers simultaneously, but there must be some system or user-configured resolution mechanism to handle conflicts.

Data Partitioning

Partitioning is the process of dividing a relation *r* into partitions r_1, r_2, …, r_n that can be combined to reconstruct *r*.

- **Horizontal partitioning** - each tuple of *r* is assigned to one or more partitions.
  - Partition can be defined using selection from *r*.
  - Reconstruct *r* from partitions by performing union.
- **Vertical partitioning** - the schema for relation *r* is split into several smaller schemas.
  - Partitions are defined using projection on *r*.
  - Reconstruct *r* by joining partitions.
  - All schemas must contain a common candidate key (or superkey) to ensure lossless join property.
  - A special attribute, such as a tuple id attribute may be added to each schema to serve as a candidate key.
- Vertical and horizontal partitioning can be mixed.

Horizontal Data Partitioning

Horizontal partitioning – tuples are divided among many servers such that each tuple resides on one server.

- Partitioning techniques (assuming *n* servers):
  - **Round-robin**: Send the *i*th tuple in the relation to server *i* mod *n*.
  - **Hash partitioning**: Use a hash function *h*(*x*) on partitioning attributes *x* that maps each tuple to one of the *n* servers.
  - **Range partitioning**: Chose a partitioning attribute V and divide the domain of V using a partitioning vector [v_o, v_1, ..., v_{n-2}]. For each tuple with value v, if $v \leq v_i$ then tuple goes on server *i*. If $v \geq v_{n-2}$ go to server *n*-1.

Question: How does each partitioning technique perform for these different types of queries?

- 1) Scanning the entire relation
- 2) Lookup queries (on the partition attribute)
- 3) Range queries (on the partition attribute)
- 4) Lookup or range queries not on the partition attribute

Horizontal Partitioning Example

branch-name	*account-number*	*balance*
Hillside	A-305	500
Hillside	A-226	336
Hillside	A-155	62

account$_1$

branch-name	*account-number*	*balance*
Valleyview	A-177	205
Valleyview	A-402	10000
Valleyview	A-408	1123
Valleyview	A-639	750

account$_2$

Partitioned Account relation on branch-name attribute.

Vertical Partitioning Example

deposit$_1$

branch-name	*customer-name*	*tuple-id*
Hillside	Lowman	1
Hillside	Camp	2
Valleyview	Camp	3
Valleyview	Kahn	4
Hillside	Kahn	5
Valleyview	Kahn	6
Valleyview	Green	7

deposit$_2$

account number	*balance*	*tuple-id*
A-305	500	1
A-226	336	2
A-177	205	3
A-402	10000	4
A-155	62	5
A-408	1123	6
A-639	750	7

Partitioned Deposit relation.

Advantages of Partitioning

Horizontal:

- allows parallel processing on a relation or data item collection
- allows a relation to be split so that items are located where they are most frequently accessed

Vertical:

- allows for further decomposition from what can be achieved with normalization
- tuple id attribute allows efficient joining of vertical fragments
- allows parallel processing on a relation
- allows tuples to be split so that each part of the tuple is stored where it is most frequently accessed

Skew

Skew is when the distribution of data is not uniform. Skew reduces the performance of algorithms such as partitioning that intend for the data to be uniformly distributed across hardware.

Types of skew:

- **Attribute-value skew**
  - Some values appear in the partitioning attributes of many tuples. All the tuples with the same value for the partitioning attribute end up in the same partition.
    - E.g. Ages of students are skewed between 18-25.
  - Can occur with range-partitioning and hash-partitioning.
- **Partition skew**
  - With range-partitioning, badly chosen partition vector may assign too many tuples to some partitions and too few to others.
  - Less likely with hash-partitioning if a good hash-function is chosen.

Partitioning

Question: How many of the following statements are true?

- i) Schema partitioning is another name for horizontal partitioning.
- ii) Vertical partitioning divides a relation by its attributes.
- iii) Skew is beneficial when performing partitioning.
- iv) Replication and partitioning can be used together.

A) 0

B) 1

C) 2

D) 3

E) 4

Interquery Parallelism

Interquery parallelism is when queries execute in parallel.

- Increases throughput but does not improve response time.
- Easiest form of parallelism to support particularly in a shared memory database.

More complicated to implement on shared disk or shared nothing architectures when dealing with updates:

- Locking and logging must be coordinated by passing messages between processors if system guarantees consistency.
  - *Cache coherency* has to be maintained as reads and writes of data in buffer must find latest version of data.
- Partitioning can often help as data within a partition is only located on one server. (Replication will be an issue though).

Intraquery Parallelism

Intraquery parallelism is the execution of a single query in parallel.

- Reduces response time (especially for long-running queries).

Two forms of intraquery parallelism:

- **Intraoperation Parallelism** – parallelize the execution of each individual operation in the query.
- **Interoperation Parallelism** – execute the different operations in a query expression in parallel.

Intraoperation parallelism scales better because the number of tuples processed by each operator is typically more than the number of query operators.

Parallel Processing of Relational Operations

Our discussion assumes a shared-nothing architecture of *n* processors, $P_0, \ldots, P_{n-1}$ and *n* disks $D_0, \ldots, D_{n-1}$, where disk D_i is associated with processor P_i.

For all algorithms, we will assume that we have already partitioned relation *R* across the *n* processors uniformly using either range or hash partitioning.

Implementing parallel selection and projection:

- Each processor performs local selection (projection) on its partition. Result is sent to client.
- This also works for duplicate elimination and aggregation.

Parallel Sorting

Parallel External Sort-Merge

- Assume the relation is partitioned among disks D_0, ..., $D_{n\text{-}1}$.
- Each processor P_i locally sorts the data on disk D_i.
- The sorted runs on each processor are then merged to get the final sorted output.

Optimizations:

- The merge is trivial if the relation was range partitioned on the sort attribute.
- Note that range partitioning can be used after the local sort to parallelize the merge as well (less of a benefit).

Parallel Join

Parallel join algorithms partition the relations across the processors such that two tuples will join if and only if they are in the same partition at a single processor.

- Range or hash partitioning can be used on the *join attributes*.
- Each processor computes its local join and the final result is the union of the results of all local joins.

Interoperation Parallelism

There are two types of interoperation parallelism:

- **Pipelined parallelism** - output tuples of one operation are consumed as input by another operation. (Iterators)
  - Avoids writing intermediate results to disk.
  - With parallel systems, operations can be performed at different processors. Output of one processor is input for another processor.
  - Useful for sequences of joins but limited parallelism scaling.
- **Independent parallelism** - operations in a query that do not depend on each other can be performed in parallel.
  - Different branches of operator tree.
  - E.g. Join of four relations can be computed as join of two temporary joins of relations r_1 and r_2 and relations r_3 and r_4.

Distributed Query Optimization

Distributed query optimization is even more complex than with a centralized system.

Issues:

- Query cost estimation – must consider processing capabilities of each node as well as location of data and transfer cost
- Query decomposition – how to divide query across nodes
- Data localization – ***goal is to move query to data***
- Global optimization – optimize query overall
- Local optimization – optimize part of query on particular node
- Distributed operations – parallelizing and distributing work of joins/sorts over multiple nodes

Semijoin

The ***semijoin*** of r_1 with r_2, is denoted by $r_1 \ltimes r_2$.

Semijoin is computed by:

$$\Pi_{r1}\,(r_1 \bowtie r_2)$$

$r_1 \ltimes r_2$ selects tuples of r_1 that are present in the join of r_1 and r_2.

The semijoin operation is used to reduce the number of tuples in a relation before transferring it to another site.

- The basic idea is that one site sends all the values of the join key to the other site which then knows which tuples will participate in the join (and will only send those tuples back).

Semijoin Example

Let *Emp(ssn, name, deptName)* be at site S_1 and *Dept (name, mgrssn)* be at S_2. Compute Emp $\bowtie_{ssn=mgrssn}$ Dept.

Algorithm:

- Compute $temp_1 \leftarrow \Pi_{mgrssn}(Dept)$ at S_2. Send $temp_1$ to S_1.
- At S_1 compute $temp_2 \leftarrow Emp \bowtie temp_1$ and send back to S_2.
- Compute $Dept \bowtie temp_2$ at S_2. This is the result of $Emp \bowtie Dept$.
- In this operation sequence, temp$_2$ = $Emp \ltimes Dept$.

Performance question:

- T(*Emp*)=100,000 and T(*Dept*)=500. Size of *ssn* and *mgrssn* = 9 bytes. The size of *name* and *deptName* is 50 bytes.
- Compute the network cost of this algorithm.

Parallel Operators

Question: How many of the following statements are true?

- i) A parallel sort can perform sorting on each node and then send the sorted sublists to a single node to be merged.
- ii) A semijoin gets its efficiency by only sending tuples that participate in the join.
- iii) The #1 rule for optimization is move the data to the query.
- iv) Intraquery parallelism is the execution of a single query in parallel.

A) 0

B) 1

C) 2

D) 3

E) 4

Distributed Transaction Model

Features of a distributed transaction model:

- Transactions may access data at several sites.
    - A **local transaction** accesses data in the *single* site at which the transaction was initiated.
    - A **global transaction** either accesses data in a site different from the one at which the transaction was initiated or accesses data in several sites.
- Each site has a local **transaction manager** responsible for:
    - Maintaining a log for recovery purposes.
    - Participating in coordinating the concurrent execution of the transactions executing at that site.
- Each site has a **transaction coordinator** responsible for:
    - Starting the execution of transactions that originate at the site.
    - Distributing subtransactions at appropriate sites for execution.
    - Coordinating the termination of each transaction that originates at the site, which may result in the transaction being committed or aborted at all sites.

Distributed Concurrency Control

Concurrency control protocols must be modified to handle distributed databases.

- Locking protocols may have to determine how to share lock information.
- Propagating updates may be eager (immediate) or lazy (delayed).
- Deadlock detection using wait-for graphs must handle detecting deadlocks across multiple servers.

Commit Protocols

Commit protocols are used to ensure atomicity across all sites:

- A transaction which executes at multiple sites must either be committed at all the sites or aborted at all the sites.
- It is not acceptable to have a transaction committed at one site and aborted at another.

The ***two-phase commit*** (**2PC**) protocol is widely used.

The ***three-phase commit*** (**3PC**) allows for faster recovery than 2PC as no site must wait. However, the protocol is more complicated/costly and does not handle network partitioning.

Two-Phase Commit Protocol (2PC)

The ***two-phase commit*** (***2PC***) protocol is widely used to ensure atomicity across all sites.

The two-phase commit protocol assumes a *fail-stop* model.

- Failed sites simply stop working and do not cause any other harm, such as sending incorrect messages to other sites.

Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached.

The protocol involves all the local sites at which the transaction executed.

Phase 1: Obtaining a Decision

After all processing of a transaction is complete, the coordinator asks all participants to ***prepare to commit*** transaction *T*:

- **"Prepare" Request:** Coordinator sends (**prepare** *T*) messages to all sites at which *T* executed.
  - Coordinator adds the record <**prepare** *T*> to the log and forces log to stable storage. Will wait for response with a timeout.

Upon receiving "Prepare" message, transaction manager at site determines if it can commit the transaction.

- **"Abort" Response:** send (**abort** *T*) message to coordinator
  - Write <**abort** *T*> to the log and send (**abort** *T*) message to coordinator
- **"Ready" Response:** send (**ready** *T*) message to coordinator if the transaction can be committed.
  - Write <**ready** *T*> to the log
  - force *all records* for *T* to stable storage
  - send (**ready** *T*) message to coordinator

Phase 2: Recording the Decision

T can be committed if coordinator received a (**ready** *T*) message from all the participating sites, otherwise *T* is aborted.

Coordinator adds a decision record <**commit** *T*> or <a**bort** *T*> to the log and forces record onto stable storage.

Coordinator sends a message to each participant informing it of the decision (commit or abort).

Participants take appropriate action locally.

Two-Phase Commit (2PC) Protocol

Question: How many of the following statements are true?

- i) The protocol uses two phases of message passing.
- ii) The first phase sends a prepare to commit message to each site involved in the transaction.
- iii) A site can respond to the prepare to commit message by sending either "ready" or "abort".
- iv) If all sites respond with "ready" the coordinator, sends out a "commit" message to all participating sites.

A) 0

B) 1

C) 2

D) 3

E) 4

Handling Failures during 2PC

There are various possible failures during 2PC such as *site failure*, *coordinator failure*, and *network partitioning*.

Handling Site Failure:

- When site S_i recovers after failure, it examines its log to determine the fate of transactions active at the time of failure.
- If log contains <**commit** *T*> record, site executes **redo**(*T*).
- If log contains <**abort** *T*> record, site executes **undo**(*T*).
- If log contains <**ready** *T*> record, site must consult coordinator to determine the fate of *T*:
  - If *T* committed, **redo** (*T*) otherwise if *T* aborted, **undo** (*T*).
- If the log contains no control records concerning *T* means that site failed before responding to the <**prepare** *T*> message.
  - Since the failure of the site precludes the sending of such a response to the coordinator, site must abort *T* and executes **undo** (*T*).

Handling Failures during 2PC (2)

Handling Coordinator Failure:

- If coordinator fails while the commit protocol for *T* is executing then participating sites must decide on *T*'s fate.
  - If an active site contains a <**commit** *T*> record in its log, then *T* must be committed.
  - If an active site contains an <**abort** *T*> record in its log, then *T* must be aborted.
  - If some active site does not contain a <**ready** *T*> record in its log, then the failed coordinator cannot have decided to commit *T*. Therefore abort *T*.
  - If none of the above cases holds, then all active sites must have a <**ready** *T*> record in their logs, but no additional control records (such as <abort T> or <commit T>). In this case active sites must wait for coordinator to recover, to find decision.

Blocking problem: Active sites may have to wait for failed coordinator to recover.

Handling Failures during 2PC (3)

Handling Network Partitioning:

- If the coordinator and all its participants remain in one partition, the failure has no effect on the commit protocol.
- If the coordinator and its participants belong to several partitions:
  - Sites that are not in the partition containing the coordinator think the coordinator has failed, and execute the protocol to deal with failure of the coordinator.
    - No harm results, but sites may still have to wait for decision from coordinator.
- The coordinator and the sites are in the same partition as the coordinator think that the sites in the other partition have failed, and follow the usual commit protocol.
  - Again, no harm results

Recovery and Concurrency Control

Recovery system must handle *in-doubt* transactions.

- Transactions that have a <**ready** *T*>, but neither a <**commit** *T*> nor an <**abort** *T*> log record.
- The recovering site must determine the commit-abort status of such transactions by contacting other sites.
    - This can be slow and potentially block recovery.

Thus, recovery algorithms note lock info in the log:

- Instead of <**ready** *T*>, write out <**ready** *T*, *L*> where *L* = list of write locks held by *T* when the log is written.
- For every in-doubt transaction *T*, all the locks noted in the <**ready** *T*, *L*> log record are reacquired.

After re-acquiring locks, processing can resume.

- The commit/abort of in-doubt transactions is performed concurrently with execution of new transactions.
    - Note that new transactions may still have to wait on locks.

Handling Failures with 2PC

Question: How many of the following statements are true?

- i) If a site fails in 2PC, the transaction is always aborted.
- ii) If a coordinator fails in 2PC, the transaction is always aborted.
- iii) If a site fails and in recovery sees a "commit" entry in its log for a transaction, it performs "redo" as transaction is committed.
- iv) If a site is in a different network partition than the transaction coordinator, it always must wait for communication to coordinator to be fixed.

A) 0

B) 1

C) 2

D) 3

E) 4

2PC Question

Assume that a transaction T executes at 3 sites (S1,S2,S3) and was started at S2. The transaction completed its execution and the controller at S2 sent out prepare to commit message to all sites.

What happens if?

- 1) Site S3 replies with (abort T) message?
- 2) All sites reply with (ready T) messages but the coordinator S2 fails before it can make a decision?
- 3) All sites reply with (ready T) messages, the coordinator locally commits T and sends out commit messages but S1 fails before it gets the commit message.

Two-Phase Commit (2PC) Exercise

In groups of at least 3, act out the possible failure modes and how they are handled:

- 1) Failure of a site
- 2) Failure of coordinator
  - One site has <commit> in log
  - One site has <abort> in log
  - All sites have <ready> in log but no <commit> or <abort>
- 3) Network partitioned
  - All participants in same partition
  - Coordinator and one participant in a partition and another participant in the other partition

What is Integration/Virtualization?

Database integration and virtualization is combining the data in more than one database to have a consistent, global view.

- Typically, databases were developed independently and organization needs to combine data for reporting/analysis.
- Alternative to data warehousing which would involving moving data into a new system.

Database integration/virtualization systems must handle different operating systems, database systems, database schema designs, and query languages.

Other integration challenges:

- data model differences, naming conflicts, different database capabilities, no control over systems (autonomous)

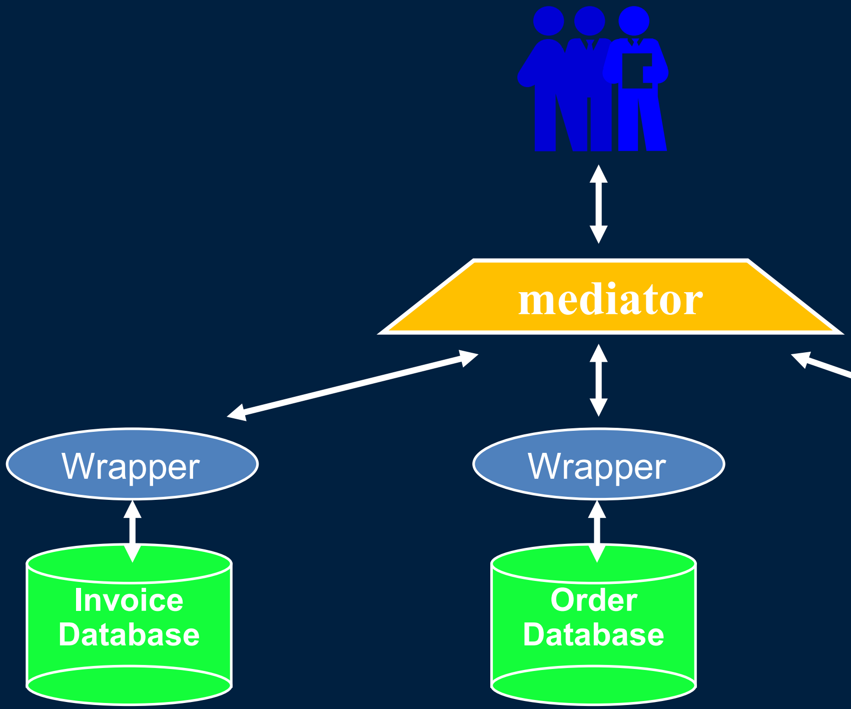
Integration/Virtualization using Mediators/Wrappers

Unlike integration using a data warehouse, integration architectures that use wrappers and mediators provide online access to operational systems.

Wrappers are software that converts global level queries into queries that the local database can handle. A ***mediator*** is global-level software that receives global queries and divides them into subqueries for execution by wrappers.

Unlike data warehouses, these systems are not suitable for very large decision-support queries because the data must be dynamically extracted from operational systems. They are useful for integrating operational systems without creating a single, unified database.

Query-Driven Dynamic Approach



Features:
- view dynamically built
- data is extracted at query-time
- still typically read-only

Database Integration/Virtualization vs. Distributed Database Systems

Integrated database systems are similar to distributed database systems as they consist of a set of databases distributed over the network.

The major difference is that all databases in an integrated database system are ***autonomous***.

- They have their own unique schema, database administrator, transaction protocols, structures, and unique function.

This autonomy introduces complexities in determining an integrated view of the data, processing local and global transactions and concurrency control, and handling database system and model heterogeneity.

Key point: Nodes in a distributed database system work together while those in a multidatabase (virtualized) system do not.

Integration/Virtualization Challenges

Database integration is an active area of research. Common problems include:

- 1) **Schema matching and merging** - How can we create a single, global schema for users to query? Can this be done automatically?
- 2) **Global Query Optimization** - How do we optimize the execution of queries over independent data sources?
- 3) **Global Transactions and Updates** - Is it possible to efficiently support transactions over autonomous databases?

Using a Global View

Once a global view has been constructed, it can be used to query the entire system:

- A user writes a query on the global view.
- The mediator converts the query into queries on the local sources (views).
- The queries are executed on the local sources and the answers integrated at the mediator before presentation to the user.

Schema Matching and Model Management

One challenging research problem is how do you automatically construct the global view?

Bernstein *et al.* have proposed model management and schema matching algorithms for this problem.

The schema matching problem takes as input two schemas and uses the names and types to determine matches between them.

- A very challenging problem involving semantics, linguistics, and ontologies.

Transaction Management

Transaction management is somewhat similar to distributed databases with the existence of local and global transactions.

However, global transactions and local transactions are managed differently:

- Local transactions are executed by each local DBMS, outside of the global system control. (*autonomy*)
- Global transactions are executed under global system control and appear as regular local transactions at each local database system.

Transaction Management (2)

Respecting *local autonomy* requires that each LDBS cannot communicate directly to synchronize global transaction execution and the MDBS has no control over local transaction execution.

Thus, the global level mediation software must guarantee *global serializability* since each LDBS only guarantees local serializability.

- Local concurrency control scheme needed to ensure that DBMS's schedule is serializable and must be able to guard against local deadlocks.

A schedule is globally serializable if there exists an ordering of committing global transactions such that all subtransactions of the global transactions are committed in the same order at all sites.

Approaches to MultiDatabase Transaction Management

Transaction management in a multidatabase has proceeded in 3 general directions:

- Weakening autonomy of local databases
- Enforcing serializability by using local conflicts
- Relaxing serializability constraints by defining alternative notions of correctness

Still a great potential to make a contribution in this area!

Global Serializability using Tickets

Architecture:

- Each site S_i has a special data item called a *ticket.*
- Transaction T_j that runs at site S_i writes the ticket at site S_i.
- Before a global transaction is allowed to commit, verify that there are no cycles based on tickets (optimistic protocol).
- Pessimistic protocol allows global transaction manager to decide serial ordering of global transactions by controlling order in which tickets are accessed.

Ensures global transactions are serialized at each site, regardless of local concurrency control method, so long as the method guarantees local serializability.

Problems include hot spot at ticket and frequent aborts under heavy transaction loads (optimistic version).

Global Serialization Graph

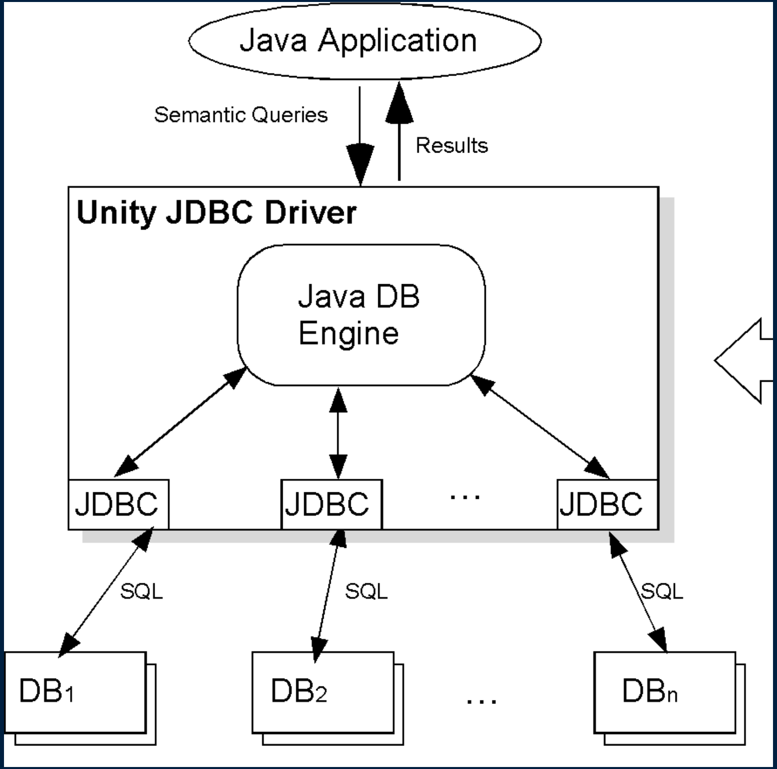
A global serialization graph (GSG) is used to determine if a global transaction can be committed using the tickets.

- The nodes of a GSG are “recently” committed transactions.
- An edge G_i -> G_j exists if at least one of the subtransactions of G_i preceded (had a smaller ticket that) one of G_j at any site.
- Initially the GSG contains no cycles.
- Add a node for the global transaction G to be committed and the appropriate edges.
- If a cycle exists abort G otherwise commit G.

My Research

My integration research built a JDBC driver called ***UnityJDBC*** that can query multiple databases at the same time.

- The system is based on the virtualization, mediator architecture.
- Contains a query parser, optimizer, and execution engine.
- Allows cross-database joins (executed client-side).
- Previous students have worked on schema matching, high-level query languages, and optimization techniques.
- Still opportunities for further work.
- Driver is used as basis for MongoDB JDBC driver that allows querying MongoDB with SQL.



Summary

Parallel and distributed databases allow scalability by using more hardware for data storage and query processing.

- Goal is for increased performance, reliability, and availability.
- Data may be distributed, partitioned, and replicated.
- Queries are distributed across nodes.
- Specialized parallel algorithms and 2PC for transactions.

Database integration/virtualization combines data from multiple databases into a single virtual system.

- The ***global view*** may be *materialized* as in ***data warehouses*** or *virtual* as in ***mediator/wrapper systems***.
- Integrated databases must handle issues in concurrency control and recovery, global view generation and maintenance, and query execution and optimization.

Major Objectives

The "One Things":

- Explain the two phase commit (2PC) protocol and how sites recover after failure.

Major Theme:

- Distributed/parallel databases allow for increased performance but complicate concurrency control and recovery.

Objectives:

- Define replication and partitioning (horizontal and vertical).
- List advantages/disadvantages of partitioning.
- Explain how semijoins are used in distributed query processing.
- Use the 4 metrics for parallel systems.
- List some factors limiting speedup and scaleup.
- Define and give an example of skew.

Objectives (2)

Objectives:

- Be able to explain some challenges in constructing an integrated database system.
- Compare/contrast integrated databases and DDBS.
- Discuss and draw the mediator architecture.
- Give an example of naming and structural conflicts.
- Define the schema matching problem.
- Define globally serializable.
- Explain the ticket protocol.

THE UNIVERSITY OF BRITISH COLUMBIA