Distributed Databases

Concept: A distributed database is a database system that stores data across multiple computers or sites on a network. This differs from a traditional database where all data resides on a single server.

Benefits:

- **Scalability:** Easily add more servers to handle growing data volumes or user access.
- **Availability:** If one site fails, others can still function (higher uptime).
- **Performance:** Distribute workload across multiple servers for faster queries.
- **Geographical Distribution:** Data can be located closer to users in different regions.
- **Improved fault tolerance:** Replication mechanisms ensure data consistency even in case of failures.

Components:

- **Sites:** Individual computers or servers storing database fragments.
- **Nodes:** Processing units within a site that manage data.
- **Distributed Database Management System (DDBMS):** Software that manages the entire distributed database and provides a unified view to users.

Distribution Schemes:

- **Fragmentation:** Dividing the database logically into smaller units for storage across sites.
    - Example: A customer database could be fragmented by region (North America, Europe, Asia).

Challenges:

- **Data Consistency:** Maintaining consistency of data across all sites after updates. (Ensuring everyone sees the same data)
- **Complexity:** Managing data across multiple locations requires more complex software and administration.
- **Network Reliability:** Reliant on a reliable network for communication between sites.

Use Cases:

- Large organizations with geographically dispersed operations (e.g., retail chains)
- Applications requiring high availability and scalability (e.g., e-commerce platforms)

- Data management for geographically distributed sensor networks (e.g., weather monitoring)

Examples of Distributed Databases: Apache Cassandra, MongoDB, CockroachDB ,Amazon DynamoDB (cloud-based), Google Spanner (cloud-based)

Distributed Data Storage

Distributed data storage is a method of storing information across multiple physical servers, often spread across different locations. Unlike traditional centralized storage, it breaks down data into chunks and distributes them for redundancy and improved performance.

Benefits:

- **Scalability:** Easily add more storage capacity by adding new servers to the network.
- **Reliability:** If one server fails, the data remains accessible on other servers.
- **Performance:** Distributed storage can improve access speeds by distributing workloads.
- **Availability:** Data is always accessible even during maintenance or upgrades.

Types of Distributed Storage:

- **Distributed File System (DFS):** Makes data appear as a single large drive even though it's spread across servers. (Example: Apache Hadoop Distributed File System (HDFS))
- **Distributed Block Storage:** Stores data in fixed-size blocks, offering high performance for large data transfers. (Example: Storage Area Network (SAN))
- **Distributed Object Storage:** Stores data in self-contained objects with unique identifiers, ideal for unstructured data. (Example: Amazon S3)

Applications:

- Cloud storage services
- Backup and disaster recovery etc.

Overall, distributed data storage offers a robust and scalable solution for managing massive amounts of data.

Distributed Transactions

Concept: A set of database operations executed across multiple interconnected databases (nodes) as a single logical unit.

ACID Properties: Ensures Atomicity, Consistency, Isolation, and Durability even across distributed data.

Why Use:

- Update related data spread across databases (e.g., updating customer info in multiple systems).
- Improve scalability and performance by distributing workload.

Example: Transferring funds between accounts in different branches (requires updating both accounts).

Process (often uses Two-Phase Commit):

1. **Global Coordinator:** Initiates the transaction and coordinates participating nodes.
2. **Preparation Phase:**
   - Each node involved prepares to commit changes (e.g., logs updates).
   - If a node fails during preparation, the transaction is aborted.
3. **Commit Phase:**
   - Coordinator instructs all nodes to commit (apply changes).
   - If a node fails during commit, others rollback (undo changes).

Challenges:

- **Network failures:** Can disrupt communication and require complex rollback procedures.
- **Data consistency:** Ensuring all nodes reflect the same changes after a transaction.

Commit Protocols

Commit protocols ensure data integrity and consistency in DBMS by coordinating transactions across multiple sites.They guarantee that a transaction either fully succeeds (commits) on all participating sites, or completely fails (aborts) on all sites.

1. One-Phase Commit (Local): Used in single-site DBMS. Transaction updates are applied, and the commit is final.

2. Two-Phase Commit (2PC): Common protocol for distributed transactions.

- **Phase 1: Preparing to Commit**
  - Coordinator (central site) asks participants (other sites) if they are ready to commit (based on local checks).

    - Participants perform pre-commit actions (e.g., write redo logs) and send "Ready" if successful.
- **Phase 2: Committing or Aborting**
    - Coordinator decides to commit or abort based on participant responses.
    - If commit: Coordinator sends "Commit" message, participants apply changes and update logs.
    - If abort: Coordinator sends "Abort" message, participants undo pre-commit actions.

3. Three-Phase Commit (3PC): Extension of 2PC for improved fault tolerance.

- **Phase 1: Pre-commit** (Similar to 2PC)
    - Coordinator asks participants if they can commit.
    - Participants check locally and send "Prepare" if ready.
- **Phase 2: Waiting for Agreement**
    - Coordinator waits for all "Prepare" or any "Abort" message.
- **Phase 3: Commit or Abort** (Similar to 2PC)
    - Based on Phase 2, coordinator sends final "Commit" or "Abort" message.

Example: Consider a bank transfer between accounts on two servers (Site A and Site B). A 2PC ensures both accounts are updated successfully or neither is.

- Phase 1: Coordinator checks balances and sends "Prepare" messages to both sites.
- Phase 2: Sites deduct from one account and send "Ready" messages. Coordinator sends "Commit" messages, and both sites update balances permanently.
- 3PC adds a phase to handle coordinator failures. It's more complex but avoids blocking if the coordinator fails.

Choosing a Commit Protocol:

- 2PC offers simplicity but has a single point of failure (coordinator) and blocking issues.
- 3PC improves fault tolerance but adds complexity and message overhead.
- The choice depends on specific requirements for performance, reliability, and availability.

Concurrency Control in Distributed Databases

- **Ensures data consistency** when multiple transactions access the same data concurrently across distributed databases.
- Maintains **ACID properties** (Atomicity, Consistency, Isolation, Durability) even in a distributed environment.

- Achieves **serializability**, meaning transactions are executed as if they happened one after another, preventing conflicts.

Single Lock-Manager Approach

- **Centralized coordinator** manages all locks for the distributed database.
- Transactions requesting a lock send a message to the lock manager.
- **Pros:** Simple to implement, good for small systems.
- **Cons:** Bottleneck at the lock manager, single point of failure.

Example: Imagine a library management system with a single database server. The lock manager controls access to borrower and book records, ensuring only one transaction updates a record at a time.

Distributed Lock Manager

- **Locks are managed locally** at each database site.
- Transactions coordinate locking across sites using protocols.
- **Protocols:**
  - Two-phase commit (2PC): Ensures all sites commit or abort a transaction together.
  - Timestamp ordering: Transactions are assigned timestamps, ensuring they commit in timestamp order.
- **Pros:** Scalable, avoids single point of failure.
- **Cons:** More complex to implement, higher overhead due to message passing.

Example: An e-commerce website with distributed databases for product and customer information. Each database manages its own locks, and a 2PC protocol ensures both databases commit or abort an order transaction together.

Parallel Databases

- **Concept:** A database system that leverages multiple processors (CPUs) and disks to improve performance through parallel processing of tasks.
- **Goal:** Handle large datasets and complex queries faster compared to traditional single-processor systems.

Benefits:

- **Increased Throughput:** Handle a larger volume of transactions per second.

- **Faster Queries:** Complex queries can be broken down and executed simultaneously on multiple CPUs.
- **Improved Scalability:** Add more processing power and storage as data grows.
- **High Availability:** System remains operational even if a single node fails (certain architectures).

Architectures:

- **Shared-Memory Architecture:** Multiple CPUs share a common memory space, each with a private disk. (Ex: Suitable for smaller deployments)
- **Shared-Disk Architecture:** Multiple CPUs have private memory but share disk storage. (Ex: More complex to manage than shared-memory)
- **Shared-Nothing Architecture:** Each node (computer) has its own CPU, memory, and disk. (Ex: Highly scalable but requires careful data distribution)

Use Cases:

- Large data warehouses for analytics
- Financial transaction processing systems
- Scientific computing applications
- Big data processing

I/O Parallelism

Concept: Speeds up data retrieval from disk by dividing relations (tables) across multiple disks.

Process:

- Data is partitioned based on a chosen attribute.
- Each partition is stored on a separate disk.
- Queries can then access data from multiple disks simultaneously.
- Results from each partition are combined after processing.

Benefits:

- Significantly reduces I/O wait time, especially for large datasets.
- Improves overall query performance.

Example: Imagine a table storing sales data. You can partition the table by year (yearly sales data on separate disks). A query filtering data for 2023 can then access the relevant disk directly, improving retrieval speed.

Implementation:

- Requires a database management system (DBMS) that supports parallel processing.
- Data partitioning strategy needs to be defined (e.g., by hash function, round-robin).

Limitations:

- Increased complexity in query optimization.
- Overhead associated with managing multiple partitions.
- Not always effective for all queries (e.g., those requiring joins across multiple partitions).

Inter-Query Parallelism

What it is: The ability of a DBMS to execute multiple queries from various applications simultaneously.

Key Points:

- **Focuses on throughput:** Aims to handle a higher volume of transactions, not necessarily speeding up individual queries.
- **Independent execution:** Each query runs separately but concurrently with others.
- **Benefits:**
  - Increased transaction processing power for busy systems.
  - Better resource utilization, especially CPU.

Example: Imagine a web application with many users running reports. Interquery parallelism allows the database to process these reports concurrently, improving overall responsiveness for all users.

Intra-Query Parallelism

Concept: A technique to speed up complex queries in a Database Management System (DBMS) by breaking them down into smaller, independent tasks that can be executed concurrently on multiple processors.

Benefits:

- **Faster Query Execution:** Reduces overall query runtime by distributing workload.
- **Improved Scalability:** Enables handling larger datasets and higher user concurrency.
- **Efficient Resource Utilization:** Utilizes available processing power more effectively.

Implementation:

- **Decomposing the Query:** The DBMS identifies independent operations within the query, such as table scans, joins, aggregations, or sorts.
- **Parallelization:** These operations are assigned to different processors for simultaneous execution.
- **Combining Results:** The DBMS gathers and combines the partial results from each processor to generate the final query output.

Intraoperation Parallelism

Intraoperation parallelism focuses on parallelizing individual operations within a single SQL query. This approach leverages multiple processors and disks to significantly improve query performance for large datasets. Here's a breakdown of two key examples:

1. Parallel Sort:

- Traditional sorting algorithms work sequentially, meaning they process data one element at a time.
- Parallel sort divides the data into smaller chunks and sorts them concurrently on multiple processors.
  - **Example:** Sorting a large customer table by name. The DBMS can split the table into partitions based on the first letter of the last name. Each partition can be sorted simultaneously on different processors, followed by merging the sorted partitions into the final result.

2. Parallel Join:

- Joins combine data from multiple tables based on a specific condition. Sequential joins process each row from one table against all rows in the other, leading to slowdowns for massive datasets.
- Parallel join partitions the tables and performs the join operation on each partition concurrently.
  - **Example:** Joining a customer table with an order table based on customer ID. The DBMS can partition both tables by customer ID and perform the join on each partition independently. Finally, the results are combined to form the final joined table.

Benefits:

- Significant performance improvement for complex queries involving large datasets.
- Efficient utilization of multi-core processors and storage systems.

Challenges:

- Requires careful query optimization to ensure effective partitioning and balanced workload distribution.
- Overhead associated with managing parallel execution and synchronization can negate benefits for smaller datasets.

Interoperation Parallelism

Interoperation parallelism deals with executing different operations within a single query expression in parallel. It utilizes two main techniques:

1. Pipelined parallelism: This approach focuses on overlapping the execution of subsequent operations in a query. Here's how it works:

- The output of one operation is fed into the next operation even before the first operation finishes processing all its data.
- Imagine an assembly line. One station finishes a task and sends the partially completed product to the next station, which can begin its work even before the first station finishes entirely.

2. Independent parallelism: This technique concentrates on executing independent operations within a query expression simultaneously.

- Independent operations have no data dependency between them, meaning the output of one doesn't affect the input of the other.
    - Example: A query that filters two separate tables (A and B) based on independent criteria can leverage independent parallelism. Processors can filter A and B concurrently as the filtering process on each table is independent.

Benefits of Interoperation Parallelism:

- Significantly reduces overall query execution time by overlapping operations.
- Makes efficient use of multiple processors in a system.

Considerations: Not all operations within a query can be parallelized due to dependencies. It requires careful query optimization to identify opportunities for parallelism.