ADBMS

UNIT 3

Distributed Data Storage:

Description:

- Distributed databases store data across multiple nodes or servers interconnected within a network.
- Data is partitioned, replicated, or distributed across nodes to improve scalability, fault tolerance, and performance.

Key Concepts:

1. Partitioning: Dividing the database into smaller subsets (partitions) distributed across nodes based on key ranges, hash values, or other criteria.
2. Replication: Maintaining copies of data across multiple nodes to improve availability, fault tolerance, and data locality.
3. Consistency Models: Defining rules and protocols for ensuring data consistency and synchronization across distributed replicas.

ACEDEMIA FORMERLY CODECHAMP

Distributed Transactions:

Description:

- Distributed transactions involve multiple operations that must be executed atomically, consistently, and durably across multiple nodes in a distributed database.

Key Concepts:

1. Atomicity: Ensuring that all operations within a distributed transaction either succeed or fail as a single unit.
2. Consistency: Maintaining data consistency constraints before and after transaction execution.
3. Isolation: Providing a level of isolation between concurrent transactions to prevent interference and ensure transaction correctness.
4. Durability: Guaranteeing that committed transactions persist even in the event of system failures or crashes.

ACEDEMIA
FORMERLY CODECHAMP

Commit Protocol: Two-Phase Commit (2PC)

Description:

- The Two-Phase Commit (2PC) protocol is a distributed algorithm used to ensure atomicity and durability of transactions across multiple nodes.

Phases:

1. Prepare Phase:
    - The coordinator node sends a "prepare" message to all participant nodes, asking them to prepare to commit the transaction.
    - Participant nodes execute the transaction locally and reply with a "ready" message if successful.
2. Commit Phase:
    - If all participants are ready, the coordinator sends a "commit" message to all nodes, instructing them to commit the transaction.
    - Upon receiving the commit message, participant nodes commit the transaction and acknowledge the coordinator.

ACEDEMIA
FORMERLY CODECHAMP

Advantages:

- Guarantees atomicity and durability of distributed transactions.
- Provides fault tolerance and recovery mechanisms in case of node failures or network partitions.

Disadvantages:

- Blocking behavior during the prepare phase can lead to scalability and performance issues.
- Susceptible to coordinator failures, leading to potential transaction termination or indefinite blocking.

Three-Phase Commit (3PC)

Description:

- The Three-Phase Commit (3PC) protocol extends the Two-Phase Commit (2PC) protocol to mitigate some of its limitations, such as blocking behavior and coordinator failures.

Phases:

1. CanCommit Phase:
    - The coordinator sends a "canCommit" message to all participant nodes, asking if they are ready to commit the transaction.
    - Participant nodes reply with either "yes" or "no" based on their local transaction state.
2. PreCommit Phase:
    - If all participants reply "yes," the coordinator sends a "preCommit" message to all nodes, indicating its intention to commit the transaction.
    - Participant nodes perform additional checks and prepare for the commit phase.
3. DoCommit Phase:
    - The coordinator sends a "doCommit" message to all nodes, instructing them to commit the transaction.
    - Participant nodes commit the transaction and acknowledge the coordinator.

ACEDEMIA
FORMERLY CODECHAMP

Advantages:

- Reduces the likelihood of blocking and deadlocks compared to the Two-Phase Commit (2PC) protocol.
- Provides better fault tolerance and recovery mechanisms in case of coordinator failures or network partitions.

Disadvantages:

- Increased complexity and overhead compared to the Two-Phase Commit (2PC) protocol.
- Still susceptible to some issues, such as blocking behavior and uncertainty about transaction outcomes.

Example:

Scenario:
A banking system processes a distributed transaction to transfer funds between two accounts stored in a distributed database.

Two-Phase Commit (2PC) Protocol:

1. Prepare Phase:
    - The transaction coordinator sends a "prepare" message to the participating nodes responsible for the source and destination accounts.
    - Participant nodes validate the transaction and reply with a "ready" message.
2. Commit Phase:
    - If both participants are ready, the coordinator sends a "commit" message to all nodes.
    - Participant nodes commit the transaction and acknowledge the coordinator.

Three-Phase Commit (3PC) Protocol:

1. CanCommit Phase:
    - The coordinator sends a "canCommit" message to the participating nodes.
    - Participant nodes reply with "yes" if they can commit or "no" if they cannot.
2. PreCommit Phase:
    - If all participants reply "yes," the coordinator sends a "preCommit" message.
    - Participant nodes prepare for the commit phase.
3. DoCommit Phase:
    - The coordinator sends a "doCommit" message to all nodes.
    - Participant nodes commit the transaction and acknowledge the coordinator.

ACEDEMIA FORMERLY CODECHAMP

Concurrency control in distributed databases is crucial for maintaining data consistency and isolation in the presence of concurrent transactions executed across multiple nodes. Let's explore two common approaches: the Single Lock-Manager Approach and the Distributed Lock Manager.

Single Lock-Manager Approach:

Description:

- In the Single Lock-Manager Approach, a centralized lock manager is responsible for managing locks on data items across all nodes in the distributed database.

Key Components:

1. Lock Manager:

- Centralized component responsible for granting and managing locks on data items.
- Maintains a global lock table containing information about locked data items and their associated transactions.

2. Lock Types:
    - Read Locks (Shared Locks): Allow multiple transactions to read data concurrently but prevent write operations.
    - Write Locks (Exclusive Locks): Grant exclusive access to a data item, preventing both read and write operations by other transactions.

Concurrency Control Protocol:

- When a transaction requests a lock on a data item, it sends a request to the centralized lock manager.
- The lock manager checks if the requested lock conflicts with existing locks held by other transactions.
- If there is no conflict, the lock manager grants the lock to the requesting transaction; otherwise, it waits until the conflicting lock is released.

Advantages:

1. Simplified implementation, as all lock management operations are centralized.
2. Ensures strict serializability by enforcing a global order of transactions.

Disadvantages:

1. Centralized lock manager can become a performance bottleneck and single point of failure.
2. Increases network traffic and communication overhead, especially in large-scale distributed systems.
3. Susceptible to scalability issues as the number of transactions and data items grows.

Distributed Lock Manager:

Description:

- In the Distributed Lock Manager approach, each node in the distributed database manages locks locally, coordinating with other nodes as needed.

Key Components:

1. Local Lock Managers:
    - Each node has its own local lock manager responsible for granting and managing locks on data items stored locally.
    - Local lock managers communicate with each other to coordinate distributed transactions and resolve lock conflicts.
2. Global Lock Manager (Optional):
    - In some implementations, a global lock manager may exist to handle cross-node transactions and resolve global lock conflicts.

Concurrency Control Protocol:

- When a transaction requests a lock on a data item, it sends a request to the local lock manager of the node containing the data item.
- The local lock manager grants the lock if there are no conflicting locks held locally; otherwise, it communicates with other nodes to resolve conflicts.
- Distributed deadlock detection and resolution mechanisms may be employed to detect and break deadlocks involving locks held by multiple nodes.

ACEDEMIA
FORMERLY CODECHAMP

Advantages:

1. Distributed approach reduces contention and communication overhead compared to a centralized lock manager.
2. Improves scalability and fault tolerance by distributing lock management across multiple nodes.

Disadvantages:

1. Increased complexity in implementation and coordination between local lock managers.
2. Potential for inconsistencies and conflicts in lock management across distributed nodes.
3. Requires efficient communication and coordination mechanisms to ensure data consistency and isolation.

Example:

Scenario:

ACEDEMIA
FORMERLY CODECHAMP

In a distributed banking application, multiple branches (nodes) execute transactions concurrently to update customer account balances stored in a distributed database.

Single Lock-Manager Approach:

- A centralized lock manager coordinates lock requests and grants locks on account balances across all branches.
- Transactions requesting locks on account balances communicate with the centralized lock manager to ensure data consistency and isolation.

Distributed Lock Manager:

- Each branch maintains its own local lock manager to manage locks on account balances stored locally.
- Transactions requesting locks on account balances interact with the local lock manager of the branch containing the account.
- In case of conflicts involving account balances across multiple branches, local lock managers coordinate with each other to resolve conflicts and ensure data consistency.
-

In parallel databases, various techniques are employed to exploit parallelism and enhance performance. These techniques include I/O parallelism, interquery parallelism, and intraquery parallelism.

I/O Parallelism:

Description:

- I/O parallelism involves parallelizing disk I/O operations to improve throughput and reduce latency in accessing data from disk storage.

Key Concepts:

1. Parallel Disk Access:
    - Multiple disk drives or storage devices are accessed concurrently to read or write data in parallel.
    - Parallelism is achieved by distributing data across multiple disks or by striping data across disk arrays (RAID).
2. Parallel File Systems:
    - Specialized file systems such as Parallel Virtual File System (PVFS) or Lustre are designed to support concurrent access to files from multiple nodes in a parallel computing environment.

ACEDEMIA
FORMERLY CODECHAMP

Advantages:

1. Improved throughput by leveraging multiple disk drives or storage devices simultaneously.
2. Reduced I/O latency by distributing I/O requests across parallel channels.
3. Scalability to handle large datasets and high-volume workloads.

Disadvantages:

1. Increased hardware complexity and cost associated with deploying and managing parallel storage systems.
2. Overhead in coordinating parallel I/O operations and managing data consistency across multiple disks.

Interquery Parallelism:

Description:

- Interquery parallelism involves executing multiple queries concurrently to utilize available processing resources efficiently.

Key Concepts:

1. Query Pipelining:
    - Queries are broken down into multiple stages or subtasks, with each stage executed concurrently by different processing units.
    - Output from one stage is fed as input to the next stage, enabling data streaming and pipelined execution.
2. Workload Partitioning:
    - Queries are partitioned or distributed across multiple processing nodes based on workload characteristics and resource availability.
    - Each node executes its assigned portion of the workload independently, enabling parallel query processing.

Advantages:

1. Increased throughput and reduced query response times by executing multiple queries concurrently.

ACEDEMIA
FORMERLY CODECHAMP

2. Efficient utilization of available processing resources by parallelizing query execution across multiple nodes.
3. Scalability to handle complex workloads and fluctuating query loads in parallel databases.

Disadvantages:

1. Potential for resource contention and interference between concurrent queries, leading to performance degradation.
2. Complexity in workload partitioning and query scheduling to ensure optimal resource utilization and load balancing.

Intraquery Parallelism:

Description:

- Intraquery parallelism involves parallelizing the execution of individual queries to exploit parallel processing capabilities within a single query.

Key Concepts:

1. Parallel Query Execution Plans:
    - Queries are optimized and parallelized by generating parallel execution plans that distribute query processing tasks across multiple processing units.
    - Parallelism is achieved at various levels, including parallel scan, join, sort, aggregation, and parallel execution of query operators.
2. Parallel Database Operators:
    - Database operators such as parallel table scans, parallel joins, and parallel sort are designed to leverage multiple processing units for concurrent execution.
    - Data partitioning and distribution techniques are used to divide query processing tasks into parallelizable units.

Advantages:

1. Faster query processing by exploiting parallel processing capabilities within a single query.
2. Efficient utilization of available CPU cores and processing resources to accelerate query execution.
3. Scalability to handle complex analytical queries and large datasets in parallel databases.

ACEDEMIA
FORMERLY CODECHAMP

Disadvantages:

1. Overhead in coordinating parallel query execution and managing data movement between processing units.
2. Complexity in query optimization and parallel execution planning to achieve optimal performance and resource utilization.

Example:

Scenario:
A parallel database system is processing a complex analytical query involving large datasets and multiple processing steps.

I/O Parallelism:

- Multiple disk drives in the storage subsystem are accessed concurrently to read data partitions in parallel.
- Parallel file systems such as Lustre are utilized to support simultaneous access to data files from multiple nodes.

Interquery Parallelism:

- Concurrent queries from different users or applications are executed simultaneously to maximize resource utilization.
- Query workload is partitioned and distributed across multiple processing nodes based on available resources and workload characteristics.

Intraquery Parallelism:

- Within each query, parallel execution plans are generated to parallelize query processing tasks.
- Parallel database operators such as parallel table scans, parallel joins, and parallel aggregations are employed to leverage multiple CPU cores and processing resources.

Intraoperation Parallelism:

Parallel Sort:

Description:

ACEDEMIA
FORMERLY CODECHAMP

- Parallel sort involves distributing sorting tasks across multiple processing units to accelerate sorting operations on large datasets.

Key Concepts:

1. Data Partitioning:
    - Input data is partitioned into smaller chunks or segments distributed across parallel processing units.
    - Each processing unit independently sorts its assigned data segment using a local sorting algorithm.
2. Merge Sorting:
    - Sorted data segments from parallel processing units are merged together using a parallel merge operation to produce the final sorted output.
    - Parallel merge algorithms efficiently merge sorted segments in parallel, leveraging multiple CPU cores.

Advantages:

1. Faster sorting of large datasets by leveraging parallel processing capabilities.
2. Improved scalability to handle increasing data volumes and processing loads.
3. Reduced sorting latency and improved query performance in parallel databases.

Disadvantages:

1. Overhead in coordinating data partitioning, sorting, and merging operations across multiple processing units.
2. Potential for load imbalance and uneven data distribution, leading to suboptimal performance in certain scenarios.

Parallel Join:

Description:

- Parallel join involves parallelizing join operations, such as hash join or sort-merge join, to efficiently combine data from multiple tables or datasets.

Key Concepts:

1. Data Partitioning:

ACEDEMIA
FORMERLY CODECHAMP

- Input tables are partitioned based on join keys or join predicates, with each partition assigned to a parallel processing unit.
- Parallel hash or range partitioning techniques are commonly used to distribute data evenly across processing units.

2. Parallel Join Algorithms:
    - Parallel hash join and parallel sort-merge join are commonly employed to execute join operations in parallel.
    - These algorithms leverage multiple processing units to perform join tasks concurrently and efficiently.

Advantages:

1. Improved join performance and reduced query execution times by parallelizing join operations.
2. Scalability to handle large join operations and complex query workloads in parallel databases.
3. Enhanced resource utilization by leveraging multiple CPU cores and processing units.

Disadvantages:

1. Overhead in data partitioning and distribution across parallel processing units.
2. Complexity in coordinating parallel join operations and managing data movement between processing units.

Interoperation Parallelism:

Pipelined Operations:

Description:

- Pipelined operations involve executing multiple query processing stages concurrently through pipelining, enabling efficient data streaming and processing.

Key Concepts:

1. Query Pipelining:
    - Query processing tasks are divided into multiple stages or operators, each responsible for a specific computation or transformation.
    - Data flows through a sequence of operators, with each operator processing a portion of the data and passing it to the next operator in the pipeline.

ACEDEMIA
FORMERLY CODECHAMP

2. Concurrent Execution:
    - Operators in the pipeline execute concurrently on different processing units, enabling parallel processing of query tasks.
    - Pipelined execution minimizes idle time and maximizes resource utilization by overlapping computation and I/O operations.

Advantages:

1. Increased query throughput and reduced query response times by parallelizing query processing stages.
2. Efficient resource utilization by overlapping computation and I/O operations in pipelined execution.
3. Scalability to handle complex query workloads and large datasets in parallel databases.

Disadvantages:

1. Complexity in designing and optimizing pipelined query execution plans to maximize performance and resource utilization.
2. Potential for data skew and load imbalance in pipelined operations, leading to suboptimal performance in certain scenarios.

ACEDEMIA
FORMERLY CODECHAMP

Example:

Scenario:
A parallel database system is executing a complex query involving sorting, joining, and aggregation operations on large datasets.

Intraoperation Parallelism:

- Parallel sort and parallel join operations are employed to accelerate sorting and joining tasks, leveraging parallel processing capabilities.
- Sorting and joining tasks are parallelized across multiple processing units to improve query performance and reduce execution times.

Interoperation Parallelism:

- Pipelined query processing is utilized to execute multiple query processing stages concurrently.

- Sorting, joining, and aggregation operators are arranged in a pipeline, allowing data to flow through the pipeline and undergo parallel processing.
- Pipelined execution minimizes idle time and maximizes resource utilization, enhancing overall query throughput and efficiency.

Independent parallelism refers to the ability to execute multiple tasks or operations concurrently without any dependencies or data sharing requirements between them. In a parallel computing environment, independent parallelism allows for efficient utilization of processing resources by executing unrelated tasks simultaneously. Let's explore the key concepts and advantages of independent parallelism:

Key Concepts:

1. Task Independence:
    - Independent parallelism involves tasks or operations that can be executed concurrently without any dependencies or interdependencies between them.
    - Each task operates on its own set of data or performs a distinct computation, without relying on the results of other tasks.
2. Parallel Execution:
    - Independent tasks are executed simultaneously on multiple processing units or cores in a parallel computing system.
    - Parallel execution enables efficient utilization of processing resources and improves overall system throughput and performance.
3. No Data Sharing:
    - Independent tasks do not share data or resources with each other during execution.
    - Data isolation ensures that concurrent tasks can execute safely and efficiently without conflicts or contention.

Advantages:

1. Increased Throughput:
    - Independent parallelism allows for the concurrent execution of multiple tasks, leading to improved system throughput and reduced overall execution times.
2. Efficient Resource Utilization:
    - By executing unrelated tasks simultaneously, independent parallelism maximizes the utilization of processing resources, such as CPU cores and memory.
3. Scalability:

ACEDEMIA
FORMERLY CODECHAMP

- Independent parallelism scales effectively with increasing workload sizes and processing demands.
- Additional tasks can be executed concurrently without impacting the performance of existing tasks, leading to linear scalability.

4. Fault Tolerance:
    - Independent tasks are resilient to failures in other tasks, as failures or errors in one task do not affect the execution of unrelated tasks.
    - Fault tolerance is improved by isolating tasks and reducing the impact of failures on overall system operation.

Example:

Scenario:
A data processing system receives a batch of independent data analysis tasks from multiple users, each analyzing a different dataset or performing a distinct computation.

Implementation:

1. Task Partitioning:
    - The system partitions incoming tasks into independent units of work, ensuring that each task operates on a separate dataset or performs a unique computation.
2. Parallel Execution:
    - Independent tasks are distributed across multiple processing units or cores in the system and executed concurrently.
    - Each task operates independently of others, without requiring coordination or communication during execution.
3. Resource Management:
    - Processing resources, such as CPU cores and memory, are allocated dynamically to execute independent tasks efficiently.
    - Resource scheduling algorithms ensure fair allocation of resources and maximize system throughput.

Benefits:

- The system achieves high throughput and efficient resource utilization by executing independent tasks concurrently.
- Users experience reduced wait times and faster response times for their data analysis tasks, improving overall user satisfaction and system performance.

ACEDEMIA
FORMERLY CODECHAMP