Touray-DSP567-Homework4

Sheikh-Sedat Touray

October 2024

1 Support offered by operating systems to a Distributed Database Management System and Benefits.

Operating systems provide several essential supports to Distributed Database Management Systems (DDBMS). These supports are critical for the efficient operation, management, and coordination of distributed databases across multiple locations. Below are key supports offered by operating systems to DDBMS, along with the benefits of each support:

1.0.1 1. Concurrency Control

- **Support**: The operating system ensures that multiple users or processes can access and modify the database concurrently without conflicts. It provides mechanisms such as semaphores, locks, and monitors to manage access to shared resources.
- **Benefit**: Helps maintain data consistency and integrity by preventing race conditions and ensuring that transactions are properly managed in a distributed environment.

1.0.2 2. Process Synchronization

- **Support**: Operating systems provide process synchronization techniques, such as message passing, shared memory, and inter-process communication (IPC), which allow multiple processes on different nodes to coordinate and collaborate.
- **Benefit**: Enables smooth operation of distributed transactions, ensuring that all nodes in the DDBMS are in sync during transaction execution.

1.0.3 . Data Storage and File Management

- **Support: Operating systems manage file systems and storage devices, ensuring efficient data storage and retrieval across distributed environments. Distributed file systems like NFS or**

Hadoop Distributed File System (HDFS) provide seamless access to data across nodes.

- **Benefit: Simplifies the management of distributed data storage, ensuring high availability, fault tolerance, and efficient data retrieval in a distributed environment.**

1.0.4 Security and Authentication

- **Support**: Operating systems provide support for user authentication, access control mechanisms, and encryption to protect data and ensure secure access to distributed database systems.
- **Benefit**: Helps safeguard sensitive data in a distributed environment by ensuring that only authorized users can access or modify data across multiple nodes.

1.0.5 Load Balancing

- **Support**: The operating system can help distribute workloads across multiple nodes in a distributed system. Load balancing techniques ensure that no single node is overwhelmed by too many requests, thus maintaining system performance.
- **Benefit**: Improves the scalability and performance of the DDBMS by evenly distributing query processing and transaction loads across nodes.

2 Database Architechtures.

2.0.1 Alternative 1: Keep All Databases As They Are (Federated Architecture)

In this scenario, the architecture will keep all databases (hierarchical, network, and relational) as they currently exist without converting them. A federated database system will act as a global application layer to integrate the geographically distributed databases. The global application will communicate with each database through appropriate gateways or middleware. The federated system will translate queries from the global application into the format required by each legacy system.

Diagram for Alternative 1: Federated Architecture

Global Application Layer		
Legacy Hierarchical DB (e.g., IMS)	**Legacy Network DB** (e.g., IDMS)	**Relational DB** (e.g., MySQL, Oracle)
Local Schema 1	**Local Schema 2**	**Local Schema 3**

Steps in this Architecture:

- **Global Application Layer**: Interfaces with users and handles requests.
- **Legacy Hierarchical DB, Legacy Network DB, Relational DB**: The existing databases remain in place.
- **Local Schemas**: Define how the data is structured and stored in each database type.
- **Gateways/Middleware**: Translate the global application's queries into the database-specific formats.

2.0.2 Alternative 2: Convert All Databases to Relational Model (Distributed Integrated Relational Database)

In this alternative, all legacy databases (hierarchical, network) are first converted into relational databases. This creates a unified relational database schema that supports global applications. The relational databases will then be geographically distributed but can be queried and managed as a single integrated system using a Distributed Database Management System (DDBMS).

Diagram for Alternative 2: Distributed Integrated Relational Database

Global Application Layer		
Global Schema		
Relational DB 1 (converted from hier.)	**Relational DB 2** (converted from net.)	**Relational DB 3** (existing rel. DB)
Local Schema 1	**Local Schema 2**	**Local Schema 3**

Steps in this Architecture:

- **Global Application Layer**: Interfaces with users and handles requests for the distributed database system.
- **Global Schema**: A unified relational schema that integrates all previously existing databases.
- **Relational DB 1, 2, 3**: The legacy hierarchical and network databases are converted into relational models.
- **Local Schemas**: The relational schema for each converted database that maps to the global schema.

DDBMS: A Distributed Database Management System that manages data consistency and query processing across all locations.

Runtime Considerations for the Two Alternatives:

Table 1: Comparison between Federated Architecture and Distributed Integrated Relational Database

Criteria	**Alternative 1: Federated Architecture**	**Alternative 2: Distributed Integrated Relational Database**
Data Model	Heterogeneous (mix of hierarchical, network, and relational)	Homogeneous (all converted to relational)
Query Execution	Requires query translation for different database types	Direct SQL execution with optimized relational queries
Performance	Lower performance due to translation overhead	Higher performance through optimized query execution
Concurrency Control	Varies by database type; complex management	Uniform concurrency control across the system
Data Access Speed	Varies depending on database type (e.g., tree navigation, indexing)	Consistent and optimized for relational access
Network Latency	Increased due to multiple database types and geographical distribution	Reduced through data locality and efficient query planning
Data Distribution	Not optimized; may lead to inefficient access patterns	Optimized data distribution and replication strategies
Scalability	Limited by integration complexity of multiple systems	Easier to scale with uniform relational schema
Complexity of Queries	More complex due to variations in query structures across databases	Simpler queries due to unified relational format
Transaction Management	Complex, with differing methods across systems	Consistent and efficient transaction management
Maintenance	More challenging due to multiple systems and models	Easier with a single database model and management system
Cost of Integration	Potentially high due to maintaining multiple database systems	Lower, as databases are integrated into a single relational system

3 For which types of applications were NoSQL systems developed?

NoSQL systems were developed primarily to meet the demands of modern applications that require scalability, flexibility, and high performance, especially in scenarios involving large volumes of diverse data, real-time processing, and dynamic data structures. Including but not limited to **Big Data Applications**, **Real-time Web Applications**, **Social Networking Sites, IoT Applications, Geospatial Applications, E-Commerce and Online Retail etc.**

4 Install a MongoDB server on your local machine.

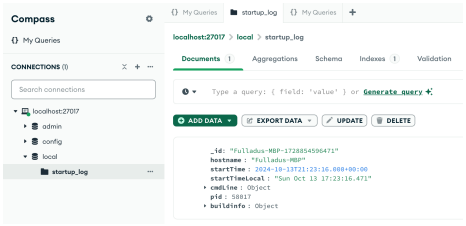

Figure 1: MongoDB Compass

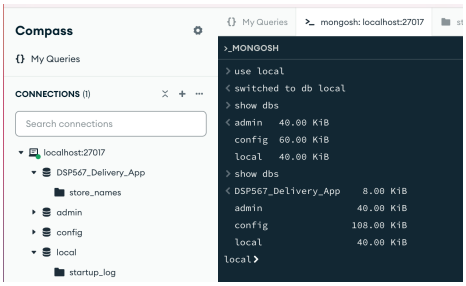

Figure 2: Query Showing Databases

5 Query language for Neo4j

The query language for Neo4j is **Cypher**. Cypher is a declarative query language specifically designed for querying and manipulating graph data in Neo4j. Some key features and characteristics of Cypher includes the fact that it; it enables users to query the graph by matching patterns of nodes and relationships, it can be used for both querying (READ) and updating **(CREATE, DELETE, SET)** the graph data, supports various functions and aggregations, allows users to perform calculations and transformations on their data, allows users to filter results using conditions and sort the results based on specific attributes and, also, allows queries to be executed within the context of transactions, ensuring data integrity and consistency. For example:

MATCH (p:Person)-[r]− >(m)
RETURN p, r, m;
This query:

- **MATCH** specifies the pattern to look for.
- **(p:Person)** represents nodes labeled as **Person**.
- $-[r]- >$ represents the relationships from **p** to another node **m**.
- **RETURN** specifies the data to be returned.

6 Services YARN can offer beyond MapReduce.

YARN (Yet Another Resource Negotiator) serves as a resource management layer within the Hadoop ecosystem, enhancing its capabilities beyond traditional MapReduce. One of the primary functions of YARN is resource management, which allocates cluster resources—such as **CPU, memory, and disk space**—to various applications, allowing multiple applications to run concurrently on the same cluster. This multi-tenancy support enables various data processing frameworks, including **Apache Spark, Apache Flink, and Apache Storm**, to operate within the same Hadoop environment, showcasing YARN's versatility.

YARN also **supports diverse processing models**, encompassing batch, interactive, and streaming processing. Its advanced job scheduling capabilities allow for dynamic resource allocation based on current workloads and application priorities, enhancing resource utilization. Additionally, YARN **provides application monitoring and management features**, enabling administrators to track the status of running applications, **monitor resource usage**, and **access application logs**. The system is designed for scalability, effectively handling thousands of nodes and applications, while offering improved fault tolerance and better isolation between applications, ensuring that one application's resource demands do not negatively impact others.

Moreover, YARN is extensible, allowing developers to build and integrate custom applications and processing frameworks that leverage its **resource management capabilities**. The combination of these features makes YARN a powerful tool in the Hadoop ecosystem, transforming it into a more versatile platform suitable for a wide range of data processing tasks beyond traditional MapReduce, ultimately catering to diverse use cases in big data analytics.

7 The different releases of Hadoop.

Hadoop has undergone several major releases since its inception, each introducing new features, improvements, and bug fixes. Below is a summary of the different major releases of Hadoop:

7.0.1 Hadoop 1.x

Initial Release (2011): Hadoop 1.0 was primarily focused on MapReduce and HDFS (Hadoop Distributed File System). This version included the core components like the JobTracker and TaskTracker for resource management and job scheduling.

7.0.2 Hadoop 2.x

- **Hadoop 2.0 (2013)**: This release introduced YARN (Yet Another Resource Negotiator), significantly enhancing resource management and allowing Hadoop to run multiple processing frameworks beyond MapRe-

duce, such as Apache Spark and Apache Tez. This version also improved scalability and provided support for high availability.

- **Hadoop 2.2 (2013)**: Introduced the concept of "Application Master" for better application management and improved performance.
- **Hadoop 2.5 (2014)**: Included support for additional security features, including Kerberos authentication.

7.0.3 Hadoop 3.x

- **Hadoop 3.0 (2017)**: This major release brought significant enhancements, including:
  - Support for containerized applications through Docker.
  - Support for erasure coding, which reduces storage overhead.
  - Improved support for cloud storage.
  - New APIs for improved developer experience.
- **Hadoop 3.1 (2018)**: Added features like Hadoop REST APIs and improvements to YARN's resource management.
- **Hadoop 3.2 (2019)**: Introduced features for enhanced compatibility with cloud services and improvements in security and performance.

7.0.4 Hadoop 3.3

Hadoop 3.3 (2020): Further improved scalability, resource management, and introduced new features for better integration with cloud storage solutions like Amazon S3.

7.0.5 Hadoop 3.4

Hadoop 3.4 (2023): This release included optimizations for running on Kubernetes, enhancements for storage efficiency, and additional performance improvements.

8 The different types of joins that can be optimized using MapReduce.

In the MapReduce framework, various types of joins can be optimized to efficiently process large datasets. The **inner join** returns records with matching values in both tables, and optimizations like partitioning and hash joins can minimize data shuffling between mappers and reducers. Similarly, the **left outer join** retrieves all records from the left table and matched records from the right table, ensuring that unmatched records from the left dataset are retained during

the shuffle phase. The **right outer join** operates in the same way but focuses on retaining all records from the right table while pulling in matched records from the left.

The **full outer join** combines the results of both left and right outer joins, including all records from both tables, even if they don't match, leading to the inclusion of NULL values for unmatched records. Optimization techniques for full outer joins include ensuring all unmatched records are accounted for during processing. The **cross join**, which produces a Cartesian product of two datasets, can be resource-intensive, but can be optimized through early filtering in the map phase or by utilizing sampling techniques.

Additionally, **self joins** involve comparing rows within the same table, with similar optimization strategies as inner joins. Overall, key optimization techniques for these joins include partitioning data based on join keys to reduce data shuffling, utilizing bloom filters for quick membership tests, and implementing combiner functions to decrease the data passed to reducers. For small datasets, **map-side joins** can be particularly effective, as they allow one dataset to be loaded into memory for faster joins during the map phase. These optimizations collectively enhance MapReduce's ability to handle joins on large datasets, improving overall performance and resource utilization.

9 Apache Pig and Pig latin.

Apache Pig is a high-level platform designed for processing and analyzing large datasets in the Apache Hadoop ecosystem. It provides an abstraction over the complexity of writing MapReduce programs, allowing users to express data transformations in a simpler, more intuitive way. Pig is designed for iterative data processing tasks and can handle both batch processing and data analysis efficiently.

Pig Latin is the scripting language used by Apache Pig. It resembles SQL in its declarative style but is tailored for processing data flows in a distributed environment. Pig Latin scripts consist of a series of data transformations, and they can handle complex data types, making it easier to work with nested data structures.

9.0.1 Example of a Query in Pig Latin

Here's a simple example of a Pig Latin script that processes a dataset of student records. Suppose we have a dataset called **Students** that contains student names and their grades, and we want to filter out students with grades less than 70.

```
-- Load the dataset
students = LOAD 'students.csv' USING PigStorage(',') AS (name:chararray, grade:int);

-- Filter students with grades less than 70
passing_students = FILTER students BY grade >= 70;
```

```
-- Group by name
grouped_students = GROUP passing_students BY name;

-- Count the number of passing students
count_passing = FOREACH grouped_students GENERATE group, COUNT(passing_students);

-- Store the results
STORE count_passing INTO 'passing_students_count' USING PigStorage(',');
```

In this query;

- **LOAD**: Loads the dataset from a CSV file and defines the schema with data types.
- **FILTER**: Filters out students who have grades below 70.
- **GROUP**: Groups the passing students by their names.
- **FOREACH** ... **GENERATE**: Counts the number of passing students for each name.
- **STORE**: Saves the results into an output file.

10 The main features of Apache Hive.

Apache Hive is a data warehousing solution built on top of Hadoop that enables users to perform large-scale data analysis using a SQL-like query language called **HiveQL**. Hive simplifies working with Hadoop's MapReduce framework by allowing users to write queries similar to SQL, making it accessible for those familiar with relational databases. Its **schema on read** feature allows flexibility in working with various data formats such as text, ORC, Parquet, and Avro, without requiring data to be preloaded into a predefined structure.

Hive is optimized for large-scale data processing, handling petabytes of data in a distributed manner. Features like **data partitioning** and **bucketing** help improve query performance by minimizing the amount of data scanned during operations. It also supports **indexing** to further enhance query efficiency. The system stores metadata in a **metastore**, which helps manage information about tables, columns, and partitions and contributes to query optimization.

Additionally, Hive integrates well with the Hadoop ecosystem, working seamlessly with **HDFS**, **MapReduce**, and tools like **Apache HBase**, **Apache Spark**, and **Apache Tez**. It supports **optimized storage formats** like ORC and Parquet, which improve performance through better compression and faster read times. Hive also provides extensibility through user-defined functions (UDFs) and has security features like **Kerberos authentication** and integration with **Apache Ranger** for access control.

Although Hive is mainly used for **batch processing** and is optimized for long-running queries on large datasets, it remains a powerful tool for data warehousing, big data analytics, and large-scale data processing in the Hadoop ecosystem.