Big Data Technologies

Agenda

- Apache Hive

Apache Hive

- https://cwiki.apache.org/confluence/display/Hive/LanguageManual

Communication protocols

Communication protocols like **Apache Thrift**, **Protocol Buffers (Protobuf)**, **Avro**, and others are widely used in big data ecosystems for efficient serialization, data transport, and RPC (Remote Procedure Call) communication. They enable systems to exchange structured data efficiently, often across heterogeneous platforms. Here's an overview of these protocols and their roles in big data:

1. Apache Thrift

Description:

- Thrift is an open-source software framework for building cross-language services. It includes a code-generation tool and a serialization format.
- Originally developed by Facebook.

Key Features:

- **Cross-Language Support**: Supports multiple programming languages, including Java, Python, C++, and more.
- **RPC Support**: Built-in support for defining and implementing RPC services.
- **Compact Serialization**: Binary format for efficient data exchange.
- **IDL (Interface Definition Language)**: Used to define service interfaces and data types.

Advantages:

- Highly efficient and compact.

- Integrated RPC capabilities.
- Good for inter-service communication in big data systems.

Limitations:

- Slightly complex to set up and maintain.
- Less flexible schema evolution compared to Avro.

Use Cases:

- Cross-language service communication in distributed systems like Hadoop or Cassandra.

2. Protocol Buffers (Protobuf)

Description:

- Protobuf is a high-performance serialization library developed by Google. It converts structured data into a compact binary format.

Key Features:

- **Compact and Fast**: Smaller and faster than JSON or XML.
- **Cross-Language Support**: Compatible with many languages (Java, Python, Go, etc.).
- **IDL**: Defines data schemas using `.proto` files.

Advantages:

- Lightweight and efficient for serialization.
- Strong backward and forward compatibility.
- Ideal for applications requiring high throughput.

Limitations:

- Slightly harder to debug due to its binary format.
- No built-in RPC support (requires gRPC for RPC).

Use Cases:

- Streaming data in real-time processing systems (e.g., Apache Kafka or Flink).
- Communication between microservices.

3. Avro

Description:

- Avro is a row-oriented serialization format designed for Hadoop ecosystems. It stores data along with its schema in a compact binary format.

Key Features:

- **Schema Evolution**: Supports adding, removing, or modifying fields without breaking compatibility.
- **Compact Storage**: Efficient binary format for storage and transfer.
- **Integration**: Designed for seamless integration with Hadoop and related tools.

Advantages:

- Self-describing format (schema is embedded with the data).
- Highly interoperable in Hadoop-based workflows.
- Supports dynamic and complex data types.

Limitations:

- Row-based, so less optimized for analytical workloads compared to columnar formats.

Use Cases:

- ETL pipelines and data interchange in Hadoop-based systems.
- Real-time data exchange in big data applications.

4. JSON/REST

Description:

- JSON is a human-readable format commonly used for APIs and lightweight data exchange.

Key Features:

- Text-based, self-describing, and language-independent.
- Widely supported in almost all systems.

Advantages:

- Easy to debug and read.
- Universally compatible.

Limitations:

- Larger file sizes compared to binary formats like Protobuf or Avro.
- Slower to parse for large datasets.

Use Cases:

- Lightweight communication in web-based big data systems.
- REST APIs in big data architectures and web applications.

When to Use Each

- **Thrift**: For RPC-based service communication in a polyglot environment (e.g., connecting different services in Hadoop or Cassandra).
- **Protobuf**: For lightweight, efficient serialization in real-time applications, especially with gRPC.
- **Avro**: For big data pipelines and storage in Hadoop ecosystems, where schema evolution is crucial.

Hadoop History Server

- History server can show information about MR job execution.
- History server is started with command (on terminal):

```
terminal> $HADOOP_HOME/bin/mapred --daemon start historyserver
```

- HistoryServer runs on port 19888.
- Browser: http://localhost:19888/
- To stop the HistoryServer give command (on terminal):

```
terminal> $HADOOP_HOME/bin/mapred --daemon stop historyserver
```

Vectorization

- Vectorization is a performance optimization technique in Apache Hive designed to process data in batches instead of one row at a time. It operates on a set of rows (typically 1024 rows per batch) in memory, improving query execution speed by leveraging modern CPU features like SIMD (Single Instruction, Multiple Data).
- To perform math operations on primitive types, hive uses vectorization.
- Typically Hive data is stored in ORC/Parquetformat, that is further divided into the blocks and arranged in columns.
- This helps speeding up math operations on primitive types (block by block).
- Advantages
  - Improved Performance: Reduces the overhead of processing rows individually.
  - Better CPU Utilization: Exploits CPU features for parallelism.
  - Reduced Function Calls: Minimizes the overhead of invoking functions for each row.
  - Efficient Memory Access: Operates on columnar data, leading to better caching and memory locality.
- Example:

```
SET hive.vectorized.execution.enabled=false;

SELECT job, SUM(sal) FROM emp GROUP BY job;

SET hive.vectorized.execution.enabled=true;
SET hive.vectorized.execution.reduce.enabled = true;

SELECT job, SUM(sal) FROM emp GROUP BY job;
```

Hive scripts

- Set of hive commands.
- Example 1: Build Movie Recommendation System.

```sql
-- movie-recommendation.hql
SET mapreduce.reduce.memory.mb = 4096;
SET mapreduce.reduce.java.opts = -Xmx4096m;
SET hive.input.format=org.apache.hadoop.hive.ql.io.HiveInputFormat;

CREATE VIEW IF NOT EXISTS cor_movies
AS
SELECT t1.movieId m1, t2.movieId m2, t1.rating r1, t2.rating r2 FROM ratings t1 INNER JOIN ratings t2 ON
t1.userId = t2.userId
WHERE t1.movieId < t2.movieId;

CREATE MATERIALIZED VIEW IF NOT EXISTS cor_table
STORED AS ORC
TBLPROPERTIES('transactional'='true')
AS
SELECT m1, m2, COUNT(m1) cnt, CORR(r1,r2) cor FROM cor_movies
GROUP BY m1, m2
HAVING CORR(r1,r2) IS NOT NULL;

CREATE VIEW IF NOT EXISTS recommend_movies AS
SELECT c.m1, c.m2, m.id, m.title FROM movies m
INNER JOIN cor_table c ON (m.id = m1 OR m.id = m2)
WHERE cor > 0.7 AND cnt > 30;
```

- Example 2: Rebuild correlation table

```sql
-- rebuild-correlation.hql
SET mapreduce.reduce.memory.mb = 4096;
SET mapreduce.reduce.java.opts = -Xmx4096m;
SET hive.input.format=org.apache.hadoop.hive.ql.io.HiveInputFormat;

INSERT INTO movies
SELECT * FROM movies_staging WHERE condition_to_get_new_records_only;

INSERT INTO ratings
SELECT * FROM ratings_staging WHERE condition_to_get_new_records_only;

ALTER MATERIALIZED VIEW cor_table REBUILD;
```

- To run the script:
  - beeline> !run /path/of/script.hql
  - OR
  - terminal> beeline -u jdbc:hive2://localhost:10000/classwork -n $USER -f /path/of/script.hql

Partitioning

- Data pruning is the process of excluding irrelevant data when processing database queries in order to minimize the amount of data read, processed, and transferred.
- In Hive Partitioning, data is divided into multiple sub-directories under HDFS (table location) based on value of one or more columns.
- When query is executed for given value of column, only respective sub-directories data will be processed. This significantly improves performance.
- Examples:
  - Emp partitioned dept-wise
  - Emp partitioned job-wise
  - Emp partitioned dept-job-wise
- Setting the property "hive.mapred.mode=strict" disables 3 types of queries.
  - Queries on partitioned tables are not permitted unless they include a partition filter in the WHERE clause.
  - ORDER BY operation without a LIMIT clause (since it uses a single reducer which can choke your processing if not handled properly).
  - Cartesian product

Static partitioning

- Data is ingested partition-wise.
- Very fast operation.

Dynamic partitioning

- Data is ingested in staging table.
- Data is loaded partition-wise into main table using mapreduce job.
- The partitions are auto-created (if not present) as per partition column value.

Bucketing

- Data in bucketed tables is divided into multiple files.
    - col.hashCode() % bucketcount = bucket number
- When data is processed using MR job (for DML operations), number of reducers will be same as number of buckets.
- To insert data into bucketed table, it must be uploaded via staging table.
- Usually buckets are created on unique column(s) to uniformly divide data across multiple reducers.
- It provides better sampling and speed-up map side joins.
- It is mandatory for DML operations in Hive 2.x.

Hive Joins

- Hive joins are similar to RDBMS joins
    - CROSS JOIN
    - INNER JOIN
    - LEFT OUTER JOIN
    - RIGHT OUTER JOIN
    - FULL OUTER JOIN
- Salient points to consider when writing join queries:
    - LEFT, RIGHT, and FULL OUTER joins supported in order to provide more control over ON clauses for which there is no match.
    - Complex join expressions are allowed.

```sql
SELECT d.dname, e.ename FROM dept d
LEFT JOIN emp e ON e.deptno = d.deptno AND d.deptno = 40;
```

- More than 2 tables can be joined in the same query.

```sql
SELECT e.ename, m.topic FROM emp e
INNER JOIN emp_meetings em ON e.empno = em.empno
INNER JOIN meetings m ON m.meetingno = em.meetingno;
```

Assignments

1. Create a transactional ORC table "fire_data" with appropriate data types partitioned by city and buckted by call number into 4 buckets. Load data from staging table into this table.
2. Execute following queries on fire dataset.
    1. How many distinct types of calls were made to the fire department?
    2. What are distinct types of calls made to the fire department?
    3. Find out all responses for delayed times greater than 5 mins?
    4. What were the most common call types?
    5. What zip codes accounted for the most common calls?
    6. What San Francisco neighborhoods are in the zip codes 94102 and 94103?
    7. What was the sum of all calls, average, min, and max of the call response times?
    8. How many distinct years of data are in the CSV file?
    9. What week of the year in 2018 had the most fire calls?
    10. What neighborhoods in San Francisco had the worst response time in 2018?