IE494: BIG DATA PROCESSING

Study Report

Group Members : **Prof.** PM Jat

Jayswal Krishil – 202203040

Tenzing: An SQL Implementation on the MapReduce Framework

Abstract
In the era of big data, organizations face the challenge of processing and analyzing vast amounts of structured and unstructured data efficiently. Tenzing, an SQL implementation developed by Google, addresses this challenge by leveraging the power of the MapReduce framework. Designed as an internal tool, Tenzing provides a high-level declarative SQL interface, simplifying complex data processing tasks for engineers and analysts.
Tenzing abstracts the underlying complexities of MapReduce, allowing users to express data queries in SQL while benefiting from the fault tolerance, scalability, and parallelism inherent to the MapReduce paradigm. By translating SQL queries into optimized MapReduce jobs, Tenzing enables efficient execution of analytical workloads over massive datasets distributed across Google's infrastructure.

1 - Introduction

The exponential growth of data in the modern world has led to the emergence of big data technologies designed to handle large-scale data storage and processing. Traditional data processing systems struggle to efficiently manage datasets that span terabytes or petabytes, creating a need for scalable, distributed solutions. MapReduce, introduced by Google, emerged as a revolutionary framework for processing such massive datasets by distributing computations across a large number of machines. While MapReduce is powerful, its low-level programming model requires significant expertise, making it challenging for non-specialist users to develop complex data processing pipelines. Recognizing this limitation, Google developed **Tenzing**, a SQL implementation on top of the MapReduce framework. Tenzing provides a high-level declarative SQL interface that simplifies data processing for engineers and analysts, enabling them to focus on the logic of their queries rather than the mechanics of distributed computing.
Tenzing bridges the gap between the simplicity of SQL and the power of MapReduce. By translating SQL queries into MapReduce jobs, it allows users to perform complex analytics on massive datasets without directly engaging with the intricacies of MapReduce. This approach democratizes big data processing, making it accessible to a wider audience with minimal learning overhead.

2 - History and Motivation

At the end of 2008, Google faced significant challenges with its existing data warehouse for Google Ads, which was implemented on a proprietary third-party database appliance (referred to as **DBMS-X**). While DBMS-X was functional, it could not keep pace with the demands of Google's rapidly growing data and analytical needs. Several critical issues prompted a complete redesign of the platform, eventually leading to the development of **Tenzing**, a system built on Google's internal infrastructure and the MapReduce framework.

2.1 - Challenges with DBMS-X

1. **Increased Cost of Scalability:**
    - Scaling DBMS-X to handle petabytes of data was prohibitively expensive. Google needed a cost-effective solution capable of scaling horizontally to meet its growing data needs.
2. **Rapidly Increasing Data Loading Times:**
    - Importing new data into DBMS-X took hours daily, with additional delays when incorporating new data sources.
    - These data import jobs competed for resources with user queries, degrading query performance and user experience.
3. **Limitations on Analyst Creativity:**
    - Analysts were constrained by DBMS-X's SQL capabilities, which lacked advanced functionality for complex analyses.
    - Analysts frequently resorted to writing custom code (e.g., using Sawzall against logs) to perform non-trivial analyses, leading to inefficiencies and fragmented workflows.

2.2 - The Decision to Transition

Recognizing these challenges, Google decided to transition its data warehouse and analytics platform to its own infrastructure, specifically leveraging the **MapReduce framework**. This transition aimed to address the shortcomings of DBMS-X while capitalizing on Google's expertise in distributed systems.

2.3 - Requirements for the New Platform

The new system, which would eventually become Tenzing, was designed with the following goals:

1. **Scalability:**
    - Handle thousands of cores, hundreds of concurrent users, and petabytes of data seamlessly.
2. **Reliability on Commodity Hardware:**
    - Operate reliably on inexpensive, off-the-shelf hardware, accommodating potential hardware failures without affecting the system's overall reliability.
3. **Performance:**

    - Match or exceed the performance of DBMS-X for common scenarios, ensuring a smooth transition for users.
4. **Integration with Google Infrastructure:**
    - Minimize expensive ETL (Extract, Transform, Load) processes by allowing direct analysis of data stored on Google's distributed systems.
5. **Comprehensive SQL Support:**
    - Offer full SQL functionality to reduce the learning curve for analysts and improve adoption rates.
    - Support advanced features, including complex user-defined functions, predictive analytics, and data mining.

2.4 - Development and Success

The development of Tenzing was completed over 18 months. Following its deployment, all users were successfully migrated from DBMS-X to Tenzing. The new platform provided:

- **Significant data expansion** capabilities, allowing access to more comprehensive datasets.
- **Enhanced analytical power**, enabling complex queries and custom analyses.
- **Improved scalability** without compromising performance for common scenarios.

Tenzing not only solved the challenges of DBMS-X but also offered a transformative shift in how data was analyzed at Google. Its design and architecture became a precursor to many modern SQL-on-distributed-framework systems, setting a new benchmark for scalability and usability in big data analytics.

3 - Implementation Overview

The implementation of **Tenzing** is built upon a distributed and modular architecture designed to handle large-scale data processing efficiently. The system integrates the **MapReduce framework** at its core while providing high-level components to manage and streamline query execution. This section explores the architecture and its core terminologies.

3.1 - Architecture Overview

The **Tenzing architecture** comprises of four major components:

1. **Worker Pool:**
    - A distributed execution system responsible for executing the MapReduce jobs derived from SQL queries.
    - To minimize query latency, the worker pool employs a model where worker processes are kept running persistently instead of being spawned on demand. This approach significantly reduces query execution latency.
    - The worker pool includes:
        - **Masters**: Manage and assign tasks to worker nodes.
        - **Workers**: Execute tasks by manipulating the data corresponding to tables in the metadata layer.

        - **Master Watcher**: Acts as the overall gatekeeper, ensuring proper coordination among masters and workers.
    - The backend storage is heterogeneous, supporting multiple data stores such as **ColumnIO**, **Bigtable**, **GFS**, and even traditional databases like **MySQL**.
2. **Query Server:**
    - Serves as the primary gateway between the user/client and the worker pool.
    - Responsibilities include:
        - Parsing user queries into an **intermediate parse tree**.
        - Fetching table metadata and schemas from the metadata server.
        - Applying **query optimization** using rule-based and cost-based techniques to generate an optimized execution plan.
        - Sending the execution plan to the worker pool via the master.
    - The query server ensures efficient query execution and interacts with clients through various interfaces.
3. **Metadata Server:**
    - Stores and provides access to table metadata, schemas, and security-related information (e.g., Access Control Lists or ACLs).
    - Backed by **Bigtable** for persistent storage.
    - Ensures efficient metadata retrieval and contributes to system scalability.
4. **Client Interfaces:**
    - Multiple interfaces cater to different types of users:
        - **Command-Line Interface (CLI):** Offers powerful scripting capabilities for advanced users.
        - **Web UI:** Designed for novice and intermediate users, with features like syntax highlighting and query browsers.
        - **API:** Allows direct execution of queries for programmatic access.
        - **Standalone Binary:** A lightweight alternative for executing queries without server-side components by directly launching MapReduce jobs.

3.2 - Core Terminologies

1. **Workers:**
   Distributed processes that execute tasks from MapReduce jobs. Workers operate on partitioned data shards and perform both map and reduce tasks.
2. **Masters and Master Watcher:**
   The master nodes manage the distribution of tasks among workers, while the master watcher ensures coordination and resource management across the system.
3. **MapReduce Framework:**
    - A programming model central to Tenzing's data processing.
    - Queries are divided into **MapReduce jobs**, with each job comprising two main stages:

        - **Map Stage:** Processes and partitions the input data into key-value pairs.
        - **Reduce Stage:** Aggregates the mapped data to produce final results.
    - Tenzing extends this framework by embedding SQL query semantics and optimizations.
4. **Metadata Server:**
   Manages metadata about tables, schemas, and data locations. It plays a crucial role in fetching required information for query execution and ensures secure access through ACLs.

3.3 - Life of a Query

The following steps illustrate how a typical query is processed within Tenzing:

1. **Query Submission:**
    - A user submits the query via the Web UI, CLI, or API.
2. **Query Parsing:**
    - The query server parses the query into an intermediate parse tree.
3. **Metadata Retrieval:**
    - The query server retrieves metadata (e.g., table schemas) from the metadata server to create a more detailed intermediate format.
4. **Query Optimization:**
    - The optimizer applies rule-based and cost-based optimizations to create an efficient execution plan.
5. **Job Partitioning:**
    - The execution plan is divided into multiple **MapReduce jobs**, each corresponding to a shard of the query's workload.
    - The query server assigns jobs to available masters via the master watcher.
6. **Task Execution:**
    - Idle workers poll the masters for tasks, execute the assigned work (map/reduce), and write intermediate results to storage.
7. **Result Aggregation:**
    - The query server monitors intermediate results and gathers them as they are produced.
    - Results are streamed back to the client.

4 - SQL Implementation on MapReduce

Tenzing is a comprehensive SQL engine that supports most SQL92 constructs while incorporating specific enhancements designed for advanced analytics. These enhancements are carefully optimized for the underlying **MapReduce framework**, ensuring scalability and parallelism.

4.1 Projection and Filtering

Tenzing provides robust support for standard SQL operators and functions while extending functionality through integration with the **Sawzall** language. Key features include:

1. **Standard SQL Operators:**
    - Tenzing supports common operators like **arithmetic operations**, **IN**, **LIKE**, **BETWEEN**, and **CASE**.
2. **Advanced Functions with Sawzall:**
    - Built-in **Sawzall functions** are embedded in the execution engine.
    - Users can write custom functions in Sawzall and seamlessly invoke them within Tenzing queries.
3. **Query Optimization for Filtering:**
   The Tenzing compiler incorporates multiple optimizations to improve query performance, particularly in projection and filtering:
    - **Constant Expression Optimization:**
        - Expressions evaluating to constants are simplified at compile time.
        - Example: A query with SELECT 2 + 2 is transformed to SELECT 4.
    - **Predicate Push-Down for Indexed Sources:**
        - When predicates involve indexed data sources (e.g., **Bigtable**), conditions like BETWEEN are pushed down for efficient **index range scans**.
        - Example: Scanning date ranges on fact tables.
    - **Filter Push-Down to Databases:**
        - For predicates that do not use complex functions (e.g., **Sawzall functions**) and involve relational databases (e.g., **MySQL**), the filters are directly pushed down to the database query execution layer.

4.2 Aggregation

Tenzing provides extensive support for standard and advanced aggregation capabilities, making it a versatile SQL engine for analytical workloads. The system is designed to efficiently perform aggregate computations within the **MapReduce framework** while incorporating optimizations to enhance performance and scalability.

Standard Aggregate Functions

Tenzing supports the following standard SQL aggregate functions:

- **SUM**: Calculates the sum of values in a column.
- **COUNT**: Counts the number of rows or non-null values.
- **MIN/MAX**: Determines the smallest or largest value in a column.
- **DISTINCT Variants**: Operations like **COUNT DISTINCT** to count unique values.

Statistical Aggregate Functions

In addition to standard aggregates, Tenzing includes a suite of statistical functions for advanced analysis:

- **CORR (Correlation)**: Measures the relationship between two variables.
- **COVAR (Covariance)**: Assesses the degree to which two variables vary together.
- **STDDEV (Standard Deviation)**: Computes the spread of data points from the mean.

Tenzing enhances the standard aggregation process with a **hash table-based aggregation** mechanism:

- **Hash-Based Aggregation:**
    - During the **Map phase**, data is grouped using an in-memory hash table, which stores aggregate values for each group.
    - This method reduces the need for intermediate data shuffling, thereby improving efficiency.
- **Benefits of Hash-Based Aggregation:**
    - Minimizes intermediate storage and network I/O.
    - Accelerates query performance by reducing overhead in the **Reduce phase**.
    - Ensures better utilization of memory and CPU resources.

4.2.1 Hash-Based Aggregation

Hash table-based aggregation is widely used in **Relational Database Management Systems (RDBMS)** to enhance query performance by grouping data efficiently. However, implementing this method within the **MapReduce framework** introduces challenges due to its inherent sorting mechanism during the reduce phase.

Challenges in Standard MapReduce

In the default MapReduce framework, the **reducer stage** always sorts data by key, even when sorting is unnecessary. This behavior makes it inefficient to directly implement hash table-based aggregation since the sorting step adds computational overhead without providing tangible benefits for aggregation tasks.

Enhancements in Tenzing

To address this limitation, Tenzing introduces a modified MapReduce framework:

- **Relaxed Sorting Requirement:**
    - Values associated with the same key are assigned to the same reducer shard without requiring a global sort.
    - This bypasses the mandatory sorting step and enables the reducer to use hash tables directly.
- **Pure Hash Table Aggregation:**
    - Hash-based aggregation avoids the unnecessary sorting step, resulting in significant performance improvements for queries with high cardinality or large group-by keys.

Usage

Users must explicitly request hash-based aggregation due to **optimizer limitations** by specifying query hints (e.g., /*+ HASH */).

4.3 Joins

Joins are a fundamental feature in SQL, enabling users to combine data from multiple sources. In Tenzing, significant effort has been dedicated to implementing and optimizing various types of joins for compatibility with the **MapReduce framework**. This ensures efficient performance while supporting diverse use cases and data sources.

4.3.1 Broadcast Joins

Broadcast joins are an optimization technique used when one of the tables involved in the join is small enough to fit entirely in memory. In Tenzing, this capability allows the smaller "secondary table" to be distributed across all workers for efficient in-memory lookups, resulting in significantly faster join operations.

Working of Broadcast Joins

1. **Cost-Based Optimization:**
   - The Tenzing optimizer determines if the secondary table is small enough to fit in memory.
   - If the specified table order in the query is suboptimal, the optimizer can reorder tables using a combination of rule-based and cost-based heuristics, ensuring that the larger table acts as the driving table.
2. **Data Distribution:**
   - The smaller table is broadcasted (copied) into the memory of each mapper/reducer process.
   - Each worker performs in-memory lookups, which are significantly faster than disk-based joins.
3. **Handling Larger Tables:**
   - For larger tables that cannot fit in memory, Tenzing employs a **sorted disk-based serialized implementation** to conserve memory.

Optimizations in Broadcast Joins

Tenzing applies several case-specific optimizations to enhance broadcast join performance:

1. **Dynamic Data Structures:**
   - The data structure for storing lookup data is determined during execution based on the join key type:
     - **Limited Range Integer Keys:** Use an integer array.
     - **Wide Range Integer Keys:** Use a sparse integer map.
     - **Other Key Types:** Use type-specific hash tables.
2. **Selective Data Loading:**

    - Filters are applied while loading join data to minimize the size of the in-memory structure.
    - Only the columns required for the query are loaded into memory.
3. **Retaining Data for Shards:**
    - Once the secondary dataset is loaded into memory, it is retained for the entire duration of the query.
    - This avoids repeated data copies for each map shard, which is beneficial when processing many shards with limited workers.
4. **Local Disk Caching:**
    - Static and frequently used tables are cached permanently on the worker's local disk.
    - The first use of the table triggers a read into the worker; subsequent uses are served from the cached copy, avoiding remote reads.
5. **Lookup Caching:**
    - Results from the last record are cached since input data is often naturally ordered on join attributes.
    - This saves one lookup access for consecutive records with the same join key.

Performance Benefits of Broadcast Joins

- **Reduced Network Traffic:** Broadcasted data is shared locally among workers.
- **Faster Lookups:** In-memory operations drastically reduce query execution time.
- **Resource Efficiency:** Optimizations like shared memory and selective data loading minimize resource consumption.
- **Scalability:** Suitable for small secondary tables in large-scale distributed systems.

Broadcast joins in Tenzing exemplify its adaptability and optimization capabilities within the **MapReduce framework**, ensuring efficient SQL execution across distributed systems.

4.3.2 Remote Lookup Joins

Remote lookup joins are a strategy employed by Tenzing when the secondary table is too large to fit in memory, making broadcast joins impractical. This approach retrieves only the required data from the secondary table over the network during the join operation. By combining asynchronous batch lookups with caching mechanisms, Tenzing ensures efficient and scalable joins with large datasets.

Key Aspects of Remote Lookup Joins

1. **On-Demand Access:**
    - Instead of loading the entire secondary table into memory, Tenzing retrieves only the necessary rows during the join process.
    - This selective retrieval minimizes both memory usage and network overhead, optimizing resource utilization.
2. **Optimized for Large Tables:**

    - Remote lookup joins are designed for scenarios where the secondary table's size makes broadcasting infeasible.
    - By retrieving data as needed, this method prevents memory overflow and accommodates large-scale datasets.
3. **Asynchronous Batch Lookups:**
    - Tenzing employs batch lookups for efficiency, reducing the number of network requests compared to single-row lookups.
    - These lookups are performed asynchronously to minimize latency and maximize throughput.
4. **Caching and Reuse:**
    - A **local least-recently-used (LRU) cache** is maintained for recent lookup results.
    - As input data is often sequentially ordered on join keys, this cache minimizes repetitive remote requests for similar keys, enhancing performance.
5. **Network and Latency Considerations:**
    - While remote lookups can introduce network latency, Tenzing's cost-based optimizer ensures this join type is selected only when more efficient alternatives, like broadcast joins, are not viable.
    - The optimizer evaluates memory constraints and network costs, striking a balance between the two.

Advantages of Remote Lookup Joins

- **Scalability:** Handles joins with massive secondary tables without requiring extensive memory.
- **Efficiency:** Minimizes data transfer by fetching only required records.
- **Cost Optimization:** Balances network costs with memory constraints.
- **Dynamic Caching:** Improves performance for sequentially ordered input data through reuse of cached results.

Challenges and Mitigations

1. **Network Latency:**
    - Mitigated through asynchronous batch processing and LRU caching.
2. **Dependency on Secondary Table Performance:**
    - Requires the secondary table to support efficient remote lookups (e.g., indexed access in Bigtable).

Use Case

Remote lookup joins are ideal for scenarios where:

- The secondary table is significantly larger than the available memory.
- Indexed access to the secondary table is possible.
- The input data contains sequentially ordered join keys, making caching highly effective.

By integrating remote lookup joins, Tenzing provides a robust mechanism for performing scalable and efficient joins on large datasets, catering to the needs of distributed SQL engines.

4.3.3 Distributed Sort-Merge Joins

Distributed sort-merge joins are a staple in MapReduce-based systems and are particularly effective when the two tables being joined:

1. Are of comparable size.
2. Lack an index on the join key.

This join strategy leverages distributed sorting and merging to perform efficient joins on large datasets.

Overview of Sort-Merge Joins

Sort-merge joins operate in two phases:

1. **Sorting Phase:**
    - Both tables are sorted on the join key in a distributed manner.
    - Sorting ensures that matching rows are adjacent, making the merging phase efficient.
2. **Merging Phase:**
    - The sorted tables are scanned simultaneously, and matching rows are combined.
    - The operation is performed shard-wise across distributed worker nodes.

Key Characteristics

1. **Scalability:**
    - Distributed sorting ensures that even massive datasets can be handled efficiently.
2. **No Index Requirement:**
    - Unlike lookup-based joins, sort-merge joins do not rely on indices, making them suitable for unindexed datasets.
3. **Balanced Performance:**
    - Best suited for cases where the sizes of the tables are similar, preventing bottlenecks caused by skewed data distribution.

Optimizations in Tenzing

1. **Parallel Sorting:**
    - The sorting phase is parallelized across the distributed worker pool, reducing latency.
2. **Shard Alignment:**
    - Data is partitioned across shards based on the join key, ensuring that matching keys are co-located for efficient merging.
3. **Memory Management:**

    - Tenzing ensures efficient memory usage by processing data in chunks during the merging phase.
4. **Adaptive Degree of Parallelism:**
    - The optimizer determines the optimal number of workers to balance load and resource usage.

Advantages

- **Handles Large Datasets:** Can process massive tables efficiently due to distributed sorting.
- **No Index Dependency:** Suitable for datasets without pre-existing indices on the join key.
- **Robust Performance:** Performs well with balanced data distributions

Limitations

1. **Sorting Overhead:**
    - Sorting is computationally expensive, especially for very large datasets.
    - Optimizations like parallelization help mitigate this cost.
2. **Skewed Data Distributions:**
    - Performance may degrade if one table is significantly larger or if data is unevenly distributed across shards.

Use Cases

Distributed sort-merge joins are ideal when:

- Both tables are large and lack indices on the join key.
- The join operation is not feasible using hash-based or broadcast methods due to data size or memory constraints.

By incorporating distributed sort-merge joins, Tenzing provides a flexible and scalable solution for handling large-scale joins in diverse scenarios.

4.3.4 Distributed Hash Joins

Distributed hash joins are a key optimization in Tenzing's join processing, particularly effective when certain conditions arise, making them more efficient than other join methods. This join technique is especially advantageous under the following circumstances:

1. **When neither table fits completely in memory**: In cases where the tables involved are too large to be loaded entirely into memory, distributed hash joins allow the system to partition the data into manageable chunks that can be processed in parallel across multiple nodes in the cluster.
2. **When one table is significantly larger than the other**: Distributed hash joins are particularly beneficial when there is a disparity in the size of the tables. In such cases, the smaller table can be used to build a hash table, while the larger table is scanned for matching entries. This method efficiently handles the large table by reducing the amount of data that needs to be retained in memory at any one time.
3. **When neither table has an efficient index on the join key**: If the tables involved in the join lack indexes on the join key, a hash join becomes an effective

alternative. Rather than relying on sorting or indexing, a hash table is created for the smaller table, and the larger table is scanned for matching entries.

These conditions are common in Online Analytical Processing (OLAP) queries, especially when performing **star joins** with large dimension tables. In OLAP systems, star joins are often used to relate large fact tables to smaller dimension tables. Distributed hash joins in Tenzing optimize this process by partitioning the data across multiple nodes, allowing for parallelized computation and reducing the overall execution time for large-scale joins.

In summary, distributed hash joins in Tenzing provide a scalable and efficient approach for joining large and unevenly sized datasets, particularly when the data cannot be fully loaded into memory and when efficient indexing is not available. This method is ideal for complex analytical queries, making Tenzing highly suitable for OLAP workloads.

4.4 Analytic Functions

Tenzing offers a variety of powerful analytic functions, which are essential for advanced data analysis. These functions follow the syntax of well-known relational databases like PostgreSQL and Oracle, allowing users to easily transition to Tenzing if they are familiar with those systems. Below are some key features and capabilities of Tenzing's analytic functions:

4.4.1 Supported Analytic Functions

Tenzing supports a range of analytic functions that are popular in the analytical community. Some of the key functions include:

- **RANK**: Assigns a rank to each row within a partition, with ties receiving the same rank.
- **SUM**: Calculates the cumulative sum of a specified column over a partition.
- **MIN/MAX**: Retrieves the minimum or maximum value within a partition.
- **LEAD/LAG**: Accesses the next or previous row's value in the result set.
- **NTILE**: Distributes rows into a specified number of groups and assigns each row a unique tile number.

These functions are essential for reporting and data analysis tasks, allowing users to compute complex aggregates and analyze data over partitions or windows of rows.

4.4.2 Combining Aggregation and Analytic Functions

Tenzing also supports the combination of **aggregation** and **analytic functions** in the same query. When both are used, Tenzing's compiler rewrites the query into multiple subqueries. First, the aggregation functions are applied, and then the analytic functions are computed based on the results of the aggregation. This approach allows users to perform both aggregation and advanced analytics within a single query, making it a powerful tool for complex data analysis.

4.5 OLAP Extensions

Tenzing enhances SQL capabilities with support for two important OLAP (Online Analytical Processing) extensions: **ROLLUP()** and **CUBE()**. These extensions allow users to perform advanced aggregation operations, which are essential for multidimensional analysis.

4.6 Set Operations

Tenzing supports all standard SQL set operations, which are essential for performing operations on multiple result sets. Each of these set operations has its own purpose and use case:

- **UNION** combines results from two queries and removes duplicate rows.
- **UNION ALL** also combines results from two queries but retains all rows, including duplicates.
- **MINUS** returns rows that are present in the first query but not in the second.
- **MINUS ALL** returns rows from the first query that do not exist in the second, but keeps duplicates.

4.6.1 Implementation

Tenzing implements set operations primarily in the **reduce phase** of the MapReduce framework. During the **Map()** phase, the mapper emits records with the whole record as the key. This ensures that similar keys (or records) are grouped together and processed in the same **reduce()** call, enabling efficient set operation execution.
For **UNION ALL**, Tenzing optimizes the operation by using **round robin partitioning** and disabling **reducer-side sorting**. This strategy improves efficiency by ensuring that records from both queries are distributed evenly across reducers without the overhead of sorting, which is unnecessary for **UNION ALL** since duplicates are allowed.
This approach to implementing set operations allows Tenzing to efficiently handle large-scale queries, minimizing the computational overhead typically associated with sorting and merging during the set operation process.

4.7 Nested Queries and Subqueries

Nested queries, or subqueries, are queries embedded within another query and can appear in various clauses such as **WHERE**, **FROM**, or **SELECT**. Tenzing optimizes the execution of these queries by leveraging the parallelism offered by the underlying MapReduce framework, which enables efficient distributed processing.

Execution of Nested Queries

Tenzing processes nested queries by treating each subquery as a separate MapReduce job. The main goal is to optimize query execution by reducing redundant operations and efficiently managing intermediate results.

1. **Subquery Execution in the WHERE Clause**:
    - In the case of a **WHERE** clause subquery, Tenzing first executes the subquery to filter records. The result of this subquery is then used as a

condition in the parent query to restrict the data returned. This approach ensures that only relevant records are considered in the outer query.
    - For instance, a query that checks if an ID exists in a subquery result will first execute the subquery and then filter the main dataset based on that condition.
2. **Subquery Execution in the FROM Clause**:
    - When subqueries appear in the **FROM** clause, Tenzing treats them as temporary tables. These subqueries are executed as separate MapReduce jobs, and their results are used as intermediate datasets for further processing in the outer query.
    - This method allows for efficient handling of joins and aggregations on the results of the subquery, improving the overall performance of the query. By handling subqueries as independent jobs, Tenzing can execute complex operations while minimizing memory usage and computational overhead.
3. **Correlated Subqueries**:
    - A **correlated subquery** references columns from the outer query, making its execution more complex. In such cases, Tenzing ensures that the subquery is executed for each row of the outer query, thereby avoiding redundant calculations and minimizing unnecessary data shuffling.
    - Tenzing optimizes the execution of correlated subqueries by efficiently managing intermediate results in memory. This reduces the amount of data shuffled between nodes in the MapReduce cluster and ensures that subqueries are processed efficiently in a distributed environment.

Challenges and Optimizations

While Tenzing provides efficient handling of nested queries, certain challenges remain:

- **Deeply Nested Queries**: Complex, deeply nested queries can increase the computational overhead, as they require multiple MapReduce jobs. The deep nesting may also lead to performance bottlenecks due to the increased number of operations involved.
- **Multiple Correlated Subqueries**: Queries with multiple correlated subqueries are harder to optimize, as each subquery relies on columns from the outer query. Managing the intermediate results for these types of queries can create performance challenges, especially in distributed systems where data shuffling and memory usage need to be carefully managed.

To address these challenges, Tenzing continues to improve its query optimization techniques for nested queries. The system is evolving to handle deeply nested and multiple correlated subqueries more effectively by refining query rewriting strategies and enhancing memory management. These ongoing improvements are aimed at reducing performance bottlenecks and optimizing resource usage across the MapReduce cluster.

4.8 Handling Structured Data

Tenzing offers read-only support for structured data, including nested and repeated formats like protocol buffers. However, its query engine handles only flat relational data, requiring hierarchical structures to be flattened during processing. This approach introduces limitations compared to systems like Dremel, which natively support nested fields.

Key Aspects of Structured Data Handling in Tenzing

1. **Read-Only Support for Nested Data:**
   Tenzing can query complex nested and repeated data formats but only in a read-only capacity, limiting users from modifying nested structures directly.
2. **Flattened Data Processing:**
   Hierarchical data is flattened into multiple flat records for processing, aligning with Tenzing's relational data model. While simplifying queries, this approach adds computational overhead and lacks the efficiency of systems like Dremel, which natively handle nested fields.
3. **Restrictions on Mixed Repetition Levels:**
   Tenzing does not allow querying fields from multiple repetition levels in a single query. For example, selecting fields from both Employee and Location repeating groups is invalid, restricting flexibility when working with independently nested fields.

In summary, Tenzing supports structured data but faces inefficiencies due to flattening and limitations in handling complex nested queries.

7.9 Views in Tenzing

Views in Tenzing are logical constructs that represent reusable named queries, functioning like tables in queries without physical data duplication. They offer enhanced security, modularity, and data abstraction, but come with potential performance trade-offs.

7.9.1 Key Aspects of Views

1. **Definition and Usage:**
   Views are defined using the CREATE VIEW statement, encapsulating complex queries into reusable components. Users can query views like tables, abstracting underlying complexities such as filters, joins, or aggregations.
2. **Data Security:**
   Views enforce row- and column-level security by restricting access to specific data subsets.
   - **Row-Level Security:** Filters rows based on conditions like department ID, ensuring users access only relevant data.
   - **Column-Level Security:** Excludes sensitive fields, such as salary, from views, limiting data exposure.
3. **Inline Expansion and Optimization:**
   Views are expanded inline during query compilation, incorporating the view's logic directly into the main query. While this avoids redundant data storage, it

can increase computational and memory overhead for complex views or large datasets.

4. **Limitations:**
   Complex views involving multiple joins or subqueries may lead to performance overhead. Proper design is essential to balance benefits like security and modularity against potential query execution costs.

7.9.2 Advantages of Views

- **Improved Security:** Fine-grained access control for both rows and columns.
- **Simplified Querying:** Abstracts repetitive logic into reusable views, reducing complexity.
- **Data Abstraction:** Shields users from schema changes, ensuring stability in querying.

In summary, views in Tenzing enhance security, simplify query management, and offer data abstraction. However, they require careful consideration of performance impacts for large-scale or complex queries.

4.10 DML (Data Manipulation Language) in Tenzing:

Tenzing provides basic DML support for batch processing of large datasets through INSERT, UPDATE, and DELETE operations. While efficient for bulk modifications, it is not fully ACID-compliant due to the lack of isolation, which can result in inconsistencies during concurrent transactions.

Key DML Operations:

1. **INSERT:**
   - Designed for large-scale, batch-style insertions. New datasets are appended, but real-time or transactional inserts are not supported.
   - Foreign key constraints can be defined in metadata but are not enforced, leaving data integrity management to users.
2. **UPDATE:**
   - Performed in batch mode by creating a new dataset with modified records, replacing the original dataset.
   - Does not support joins, restricting updates to simple cases.
   - Lack of isolation may cause inconsistent reads during concurrent updates.
3. **DELETE:**
   - Operates in batch mode, creating a new dataset excluding deleted records and replacing the original dataset.
   - Joins are not supported, and concurrent operations may lead to inconsistent states due to the absence of isolation.

ACID Compliance and Limitations:

- Tenzing ensures atomicity, consistency, and durability but lacks isolation, making it prone to issues like dirty or non-repeatable reads during concurrent operations.

- The absence of join support in UPDATE and DELETE limits flexibility for complex data modifications.
- Best suited for bulk, periodic updates or deletions, not real-time transactional systems.

Tenzing's DML operations prioritize batch efficiency over transactional precision, offering basic functionality for large-scale data modifications. However, its limitations, including lack of isolation and join support, necessitate careful consideration for use cases requiring real-time or complex updates.

4.11 DDL (Data Definition Language) in Tenzing

Tenzing supports a wide range of DDL operations to define, manage, and control dataset structures and metadata, following standard SQL conventions. These capabilities enable efficient metadata management, seamless integration with external data, and enhanced access control.

Key DDL Operations:

1. **CREATE [OR REPLACE] [EXTERNAL] [TEMPORARY] TABLE:**
   - Creates tables with options for replacing existing ones, querying external data, or defining session-specific temporary tables.
2. **DROP TABLE [IF EXISTS]:**
   - Deletes tables and metadata safely, avoiding errors if the table does not exist.
3. **RENAME TABLE:**
   - Renames tables for easier restructuring without altering data.
4. **GENERATE STATISTICS:**
   - Computes table statistics (e.g., row counts) to optimize query execution.
5. **GRANT and REVOKE:**
   - Manages fine-grained access control, supporting specific privileges like SELECT or INSERT.

Metadata Discovery:

- **From MySQL:** Automatically imports MySQL schema for direct querying in Tenzing.
- **From Protocol Buffers:** Discovers and imports schemas from hierarchical data formats (e.g., Bigtable, RecordIO), simplifying queries on nested data.

Benefits:

- **Efficient Metadata Management:** Simplifies tasks like table creation and renaming.
- **Seamless Integration:** Automates schema discovery for external systems.
- **Access Control:** Enables row- and column-level security via GRANT/REVOKE.
- **Optimized Queries:** Uses statistics to improve query performance.
- **Flexibility:** Supports temporary and external tables for specific use cases.

Limitations:

- **No Constraint Enforcement:** Primary and foreign keys are defined but not enforced.
- **Limited Transformations:** Focuses on metadata; data transformations require separate processes.

Tenzing's robust DDL capabilities streamline data management, offering powerful tools for metadata handling and external dataset integration. While flexible and automation-focused, users must manage data integrity due to the lack of enforced constraints.

4.12 Table-Valued Functions in Tenzing

Tenzing supports table-valued functions (TVFs), a type of user-defined function (UDF) that operates on entire datasets, enabling custom computations and transformations.

Key Features:

- **Definition**: TVFs take tables as input and return tables as output, making them ideal for complex data processing tasks like transformations and aggregations.
- **Implementation**:
    - **Sawzall Integration**: Embedded for large-scale data processing, supporting tasks like normalization, filtering, and group computations.
    - **Expandable Framework**: Designed to integrate future languages like Lua (simplicity) and R (statistical analysis).

Applications:

- **Data Normalization**: Scaling and standardizing datasets for analysis.
- **Group Computations**: Advanced aggregations and custom calculations like weighted averages.
- **Custom Transformations**: Tasks like pivoting or flattening nested structures.

Advantages:

1. **Customizability**: Define tailored logic.
2. **Scalability**: Leverages distributed processing for large datasets.
3. **Expandability**: Planned integrations with Lua and R enhance analytics.
4. **Ease of Use**: Sawzall simplifies large-scale transformations.

Challenges:

- **Language Dependence**: Current reliance on Sawzall may require learning it.
- **Performance Overhead**: Improperly optimized functions can slow processing.
- **Limited Language Support**: Awaiting Lua and R integration for broader use.

Tenzing's TVFs offer powerful tools for advanced data processing and transformation. With Sawzall and planned enhancements, it is a versatile platform for large-scale analytics.

4.13 Data Formats in Tenzing

Tenzing supports various data formats, enabling efficient querying, loading, and exporting of diverse datasets to suit different use cases and storage systems.

Key Capabilities

1. **General Handling:**

    - Direct querying of supported formats.
    - Seamless data ingestion with schema validation.
    - Exporting data with customizable options.
2. **Customizable I/O Options:**
    - **Delimited text formats:** Adjustable delimiters, encoding, quoting, escaping, and headers for compatibility.

Advantages

- **Flexibility:** Works across columnar, key-value, and relational systems.
- **Data Validation:** Ensures integrity during ingestion.
- **Analytical Optimization:** Formats like ColumnIO and Bigtable minimize read and shuffle costs.
- **Interoperability:** Compatible with existing pipelines using formats like MySQL and Protocol Buffers.

Challenges

- **Configuration Complexity:** Requires expertise in format-specific options.
- **Performance Variations:** Formats like delimited text may lack optimization for large-scale analytics.
- **Metadata Management:** Embedded metadata may be unsuitable for dynamic, large datasets.

Tenzing's versatile data format support, combined with its customizable I/O options, ensures compatibility with diverse data ecosystems while optimizing performance for modern and legacy pipelines.

5. Performance

Tenzing is designed to deliver performance comparable to leading Massively Parallel Processing (MPP) database systems like Teradata, Netezza, and Vertica. To achieve this, the Tenzing team implemented numerous optimizations across its execution framework, particularly in its integration with MapReduce. Below is a detailed explanation of Tenzing's performance-enhancing mechanisms.

5.1 MapReduce Enhancements in Tenzing

Tenzing has made several optimizations to the Google MapReduce framework, aiming to reduce latency, increase throughput, and improve the efficiency of SQL operators.

5.1.1. Workerpool Optimization

A key challenge was reducing latency. Tenzing implemented a workerpool to avoid spawning new binaries for each query:

- **Master Watcher**: Manages work requests, resource allocation, and overall pool health.
- **Master Pool**: Coordinates query execution, distributing tasks to workers.
- **Worker Pool**: Thousands of workers process tasks in a FIFO order. A priority queue is planned for task prioritization.

This approach reduced query execution latency to around **7 seconds**, with typical latencies ranging between **10-20 seconds**. Further improvements are expected to bring this under **5 seconds**.

5.1.2. Streaming & In-memory Chaining

Tenzing initially serialized intermediate data to GFS, which led to poor performance for complex queries. Enhancements included:

- **Streaming**: Upstream and downstream MapReduce jobs now communicate via network, using GFS only for backup.
- **Memory Chaining**: Co-located upstream reducers and downstream mappers improve performance further.

These changes significantly boosted the performance of multi-MapReduce queries, such as hash joins and nested sub-selects.

5.1.3. Sort Avoidance

For operations like hash join and aggregation that require shuffling but not sorting:

- The MapReduce API was modified to skip the sorting step.
- Mappers directly pass data to reducers, bypassing intermediate sorting and improving efficiency.

5.1.4. Block Shuffle

MapReduce traditionally uses row-based encoding for shuffling, but this is inefficient for operations that don’t require sorting:

- **Block-based shuffle** combines small rows into compressed 1MB blocks, treating the entire block as one row.
- This reduces the overhead of row serialization and deserialization, improving shuffle performance by **3X** compared to row-based shuffling with sorting.

5.1.5. Local Execution

For small datasets (under **128 MB**), Tenzing runs queries locally, bypassing the pool and reducing latency to about **2 seconds**.

5.2 Scalability

Tenzing showcases strong scalability, leveraging the MapReduce framework. The production deployment spans two data centers with **2000 cores** each, equipped with **6 GB of RAM** and **24 GB of local disk** per core, supporting sort buffers and local caching. Persistent data storage is handled by **GFS** and **Bigtable**.
Scalability test results:

- **Throughput** (rows processed per second per worker) remains consistent as the number of workers increases.
- This demonstrates Tenzing's ability to maintain high performance even with significant resource scaling.

5.3 System Benchmarks

Benchmark Overview

Tenzing's performance was evaluated against **DBMS-X**, a leading **Massively Parallel Processing (MPP)** database appliance with row-major storage. The benchmark used four commonly used analytical queries (detailed in Appendix A).

Benchmark Setup

- **Data Storage**: Tenzing stored data in **ColumnIO format** on **GFS**.
- **Tenzing Configuration**: 1000 processes, each with:
    - **1 CPU**
    - **2 GB RAM**
    - **8 GB local disk**
- **DBMS-X Configuration**: A high-performance MPP appliance optimized for **row-major storage**.

Results vs. DBMS-X

Query	DBMS-X(s)	Tenzing(s)	Change
#2	129	93	39% Faster
#4	70	69	1.4% Faster
#1	155	213	38% Slower
#3	9	28	3.1x Slower

Key Observations

1. **Query #2 and #4**: Tenzing **outperformed DBMS-X**, with notable improvements in **Query #2** due to its **efficient use of the MapReduce framework** and optimized SQL operator handling.
2. **Query #1**: Tenzing was slower due to its **distributed execution model**, which introduces additional overhead.
3. **Query #3**: The **performance lag** was primarily caused by **high startup time** in Tenzing’s current production setup.

Explanation of Results

- The **production version of Tenzing** was used for the benchmarks.
- Tenzing's performance could have been **significantly faster** if the **experimental LLVM engine** had been available at the time of the benchmark.

5.4 Experimental LLVM Query Engine

Overview of Execution Engine Iterations

Tenzing's execution engine has evolved through multiple iterations to improve **single-node efficiency**, aiming to reach the performance levels of commercial DBMS. The progression included:

1. **First Iteration**: SQL expressions were translated to **Sawzall code**, compiled using Sawzall’s **JIT compiler**. However, this approach was inefficient due to serialization and deserialization costs when converting between Sawzall and Tenzing’s native types.

2. **Second Iteration**: A more efficient approach using **Dremel's SQL expression evaluation engine**, which directly evaluated SQL expression parse trees. While more efficient than the Sawzall implementation, it was still slow due to its **interpreter-like nature** and **row-based processing**.

Third Iteration: LLVM-Based Native Code Generation
The third iteration focused on **LLVM-based native code generation** and **column-major vector-based processing** with **columnar intermediate storage**. Benchmarking results showed that:

- **LLVM-based engine** showed better performance for real-life queries with lower selectivity.
- **Vector-based processing** performed better for **pure select-project queries** with high selectivity.
- **Per-worker throughput** for vector processing was about **three times higher**, while LLVM-based processing was **six to twelve times faster** compared to the production evaluation engine.

Data Storage and Performance Insights

- The **LLVM engine** stores intermediate results by **rows**, even when input and output are in columnar format, offering advantages and disadvantages compared to vector-based processing.
- **Hashtable-based operations** (like aggregation and join) benefit from row-based storage due to **cache locality** and more efficient conflict resolution using pointers.
- **Vector engines** always materialize intermediate results in memory, whereas the LLVM engine can store results on the **stack** or in **registers**, improving **data locality**.
- For queries with **low selectivity**, **vector engines** load less data into cache, making them faster, whereas LLVM requires reading more data due to row-based storage.

Benchmarks: LLVM vs. Vector Engine
Following Table presents the comparison of performance between the **LLVM engine** and the **Vector engine** for typical aggregation queries. The results are measured in terms of throughput (millions of rows processed per second per worker), and the selectivity of the WHERE clause:

Query	Vector Throughput	LLVM Throughput	Ratio (LLVM/Vector)
#1	0.66	2.0	3
#2	23	61	2.6
#3	7.4	15	2
#4	12	13	1.1

The majority of Tenzing's workload consists of **analytic queries with aggregation and joins**, which are **performance bottlenecks**. The **LLVM-based query engine** handles these operations more efficiently, making it a **promising approach** for optimizing query processing.

6. Conclusion

- Tenzing shows strong performance in SQL query processing using MapReduce and optimizations like worker pooling and in-memory chaining.
- Scalability is impressive, with high throughput even as resources scale.
- Some challenges remain, such as startup time and distributed execution overhead.
- The LLVM-based query engine promises further performance improvements.
- Future optimizations will continue to enhance Tenzing's efficiency in large-scale analytical workloads.

7. Future Scope

1. **MapReduce Optimization**: Improve task distribution and reduce latency by enhancing priority queueing and metadata management.
2. **LLVM Query Engine Integration**: Full integration of the LLVM engine will boost performance for aggregation-heavy queries, potentially increasing throughput by 6-12 times.
3. **Columnar Processing Improvements**: Further research into columnar storage and hash table optimizations to improve performance for large scans and complex queries.
4. **Scalability for Large Datasets**: Continue optimizing for large-scale deployments, enabling Tenzing to handle petabytes of data across multiple data centers.
5. **Query Optimization**: Improve SQL operator optimizations, cost-based query planning, and memory management for better performance.
6. **Machine Learning for Query Optimization**: Integrate ML models to automatically tune execution plans and optimize resource usage.
7. **Mixed Workload Support**: Add support for both OLAP and OLTP workloads, allowing Tenzing to handle diverse use cases efficiently.
8. **Tooling Enhancements**: Develop advanced debugging tools for the LLVM engine and system profiling to speed up issue resolution.

These improvements aim to enhance Tenzing's performance, scalability, and reliability, positioning it as a powerful tool for handling larger and more complex datasets.

7. References

[1] Chattopadhyay, Biswapesh, et al. "Tenzing a sql implementation on the MapReduce framework." Proceedings of the VLDB Endowment 4.12 (2011): 1318-1327.
[2] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads.
[3] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. Bigtable: A distributed storage system for structured data.
[4] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall.
[5] https://stephenholiday.com/notes/tenzing/