Lecture #24: Distributed OLAP Databases

15-445/645 Database Systems (Spring 2025)

`https://15445.courses.cs.cmu.edu/spring2025/`
Carnegie Mellon University
Jignesh Patel

1 Decision Support Systems

A common enterprise workload involves gathering data and then doing analysis on it. Therefore, it makes sense to have multiple instances of OLTP databases (e.g. dealing with product/order information) that feed data into a back-end OLAP database, sometimes called a *data warehouse*. There is an intermediate step called *ETL*, or **E**xtract, **T**ransform, and **L**oad, which combines the OLTP databases into a universal schema for the data warehouse.

A current, modern trend (rather than ETL) is *ELT*, or **E**xtract, **L**oad, and **T**ransform. Once the raw data is loaded into the OLAP database, the transform is done on the OLAP database itself.

Decision support systems (DSS) are applications that serve the management, operations, and planning levels of an organization to help people make decisions about future issues and problems by analyzing historical data.

The two approaches for modeling an analytical database are *star schemas* and *snowflake schemas*.

Star Schema

Star schemas contain two types of tables: *fact tables* and *dimension tables*. The fact table contains multiple "events" that occur in the application. It will contain the minimal unique information per event, and then the rest of the attributes will be foreign key references to outer dimension tables. The dimension tables contain redundant information that is reused across multiple events. In a star schema, there can only be one dimension-level out from the fact table. Since the data can only have one level of dimension tables, it can have redundant information. Denormalized data models may incur integrity and consistency violations, so replication must be handled accordingly. Queries on star schemas will (usually) be faster than a snowflake schema because there are fewer joins. An example of a star schema is shown in Figure 1.

Snowflake Schema

Snowflake schemas are similar to star schemas except that they allow for more than one dimension out from the fact table and sometimes contain multiple fact tables. As a result of less redundancy, they take up less storage space, but they require more joins to get the data needed for a query. For this reason, queries on star schemas are usually faster. An example of a snowflake schema is shown in Figure 2.

2 Execution Models

Executing an OLAP query in a distributed DBMS is roughly the same as on a single-node DBMS. However, in the distributed setting, the DBMS considers which node the input is coming from and where to send the output.

There are two types of data in distributed databases, persistent data and intermediate data. Persistent data consists of records in the database that should be saved. Modern systems treat these files as immutable,

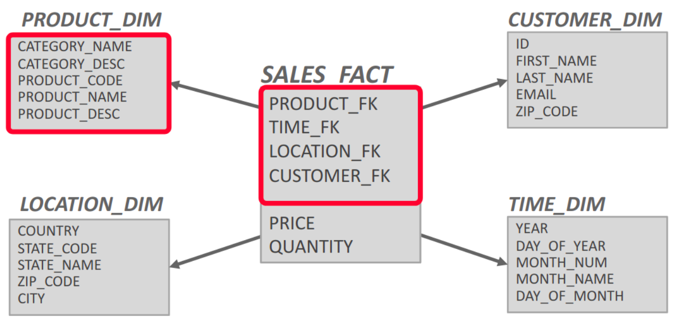

Figure 1: Star Schema – The center of the schema is the SALES fact table that contains key references to outer dimension tables. Because star schemas are only one-dimensional, the outer dimensional tables cannot point to other dimension tables.

but can support updates by rewriting them. Intermediate data is short-lived and is produced as an output of query operators, and will be consumed by other operators.

A common operation in distributed DBMSs is the shuffle operation, which partitions data among a bunch of worker nodes. It's similar to hash partitioning in hash joins, but sends data over the network. This will be discussed more later.

A distributed DBMS's execution model specifies how it will communicate between nodes during query execution. Two approaches to executing a query are *pushing* and *pulling*.

Pushing a Query to Data

For the first approach, the DBMS sends the query (or a portion of it) to the node that contains the data. It then performs as much filtering and processing as possible where data resides before transmitting over network, in order to minimize costly data transmission. The result is then sent back to where the query is being executed, which uses local data and the data it receives to complete the query. This is more common in a shared nothing system.

Pulling Data to Query

For the second approach, the DBMS brings the data to the node that is executing a query that needs it for processing. In other words, nodes detect which partitions of the data they can do computation on and pull from storage accordingly. Then, the local operations are propagated to one node, which does the operation on all the intermediary results. This is normally what a shared disk system would do. The problem with this is that the size of the data relative to the size of the query could be very different. A filter can also be sent to only retrieve the data needed from disk.

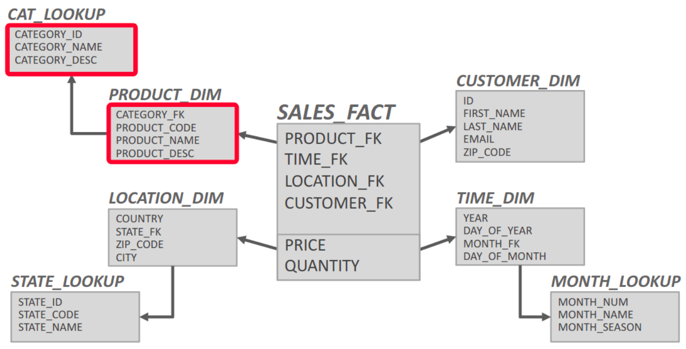

Figure 2: Snowflake Schema – The category information in the product dimension table can be broken out in the snowflake table.

Query Fault Tolerance

The data that a node receives from remote sources are cached in the buffer pool. This allows the DBMS to support intermediate results that are larger than the amount of memory available. Ephemeral pages, however, are not persisted after a restart. Therefore, a distributed DBMS must consider what happens to a long-running OLAP query if a node crashes during execution.

Most shared-nothing distributed OLAP DBMSs are designed to assume that nodes do not fail during query execution. If one node fails during query execution, then the whole query fails, which forces the entire query to re-execute from the start. This can be expensive, as some OLAP queries can take days to execute.

The DBMS could take a snapshot of the intermediate results for a query during execution to allow it to recover if nodes fail. This operation is expensive, however, because writing data to disk is slow. In the modern shared-disk cloud, many DBMSs provide fault tolerance by simply writing data to the shared store (e.g. Amazon S3) after expensive operations.

3 Query Planning

All the optimizations that we talked about before are still applicable in a distributed environment, including predicate pushdown, early projections, and optimal join orderings. Distributed query optimization is even harder because it must consider the physical location of data in the cluster and data movement costs.

Query Plan Fragments

One approach is to generate a single global query plan and then distribute *physical operators* to nodes, breaking it up into partition-specific fragments. Most systems implement this approach.

Another approach is to take the *SQL* query and rewrite the original query into partition-specific queries. This allows for local optimization at each node. SingleStore and Vitess are examples of systems that use this approach.

4 Distributed Join Algorithms

For analytical workloads, the majority of time is spent computing joins and reading from disk, so optimizing joins is important. The efficiency of a distributed join depends on the target tables' partitioning schemes.

One approach is to put entire tables on a single node and then perform the join there. However, the DBMS loses the parallelism of a distributed DBMS, defeating the purpose of making it distributed. This option also entails costly data transfer over the network.

To join tables R and S, the DBMS needs to get the proper tuples on the same node. Once there, it then executes the same join algorithms discussed earlier in the semester. One should always send the minimal amount needed to compute the join.

There are four scenarios for distributed join algorithms.

Scenario 1

One of the tables is replicated at every node and the other table is partitioned across nodes. Each node joins its local data in parallel and then sends their results to a coordinating node.

Scenario 2

Both tables are partitioned on the join attribute, with IDs matching on each node. Each node performs the join on local data and then sends to a node for coalescing.

Scenario 3

Both tables are partitioned on different keys. If one of the tables is small, then the DBMS broadcasts that table to all nodes. This takes us back to Scenario 1. Local joins are computed and then those joins are sent to a common node to operate the final join. This is known as a *broadcast join.*

Scenario 4

This is the worst case scenario. Both tables are not partitioned on the join key. The DBMS partitions the tables across nodes via the shuffle operator such that nodes have data matching on the join ID. This recovers Scenario 2. Local joins are computed and then the results are sent to a common node for the final join. If there isn't enough disk space, a failure is unavoidable. This is called a *shuffle join.*

Semi-Join Optimization

Before pulling data from another node, a *semi-join-filter* can be used to reduce data movement. This filter blocks rows from the data that will be discarded in the result anyway due to a predicate. An approximate filter like a Bloom filter can be used here.

Shuffle Operation

In addition to being useful in joins, a shuffle operation is generally useful. It can be used to rebalance data based on observed characteristics, and adjust the capacity allocated to a subcomputation. The shuffle operator is basically the repartition type of exchange operator discussed previously in lecture.

5 Cloud Systems

Vendors provide *database-as-a-service* (DBaaS) offerings that are managed DBMS environments.

Newer systems are starting to blur the lines between shared-nothing and shared-disk. For example, **Amazon S3** allows for simple filtering before copying data to compute nodes. There are two types of cloud systems, managed or cloud-native DBMSs.

Managed DBMSs

In a managed DBMS, there is no significant modification to the DBMS to make it ”aware” that it is running in a cloud environment. It provides a way to abstract away all the backup and recovery for the client. This approach is deployed in most vendors.

Cloud-Native DBMS

A cloud-native system is designed explicitly to run in a cloud environment. This is usually based on a shared-disk architecture. This approach is used in **Snowflake**, **Google BigQuery**, **Amazon Redshift**, and **Microsoft SQL Azure**.

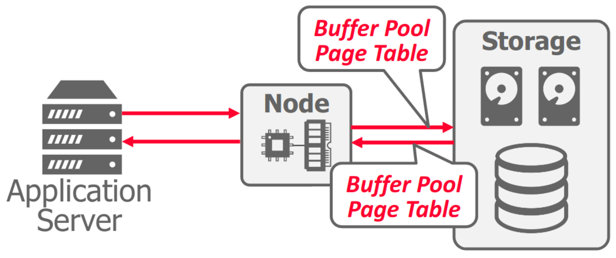

Figure 3: Serverless Database – When the application server becomes idle, the user must pay for resources in the node that are not being used. In a serverless database, when the application server stops, the DBMS takes a snapshot of pages in the buffer pool and writes it out to shared disk so that the computation can be stopped. When the application server returns, the buffer pool page table restores the previous state in the node.

Serverless Databases

Rather than always maintaining compute resources for each customer, a *serverless DBMS* evicts tenants when they become idle, checkpointing the current progress in the system to disk. Now, a user is only paying for storage when not actively querying. A diagram of this is shown in Figure 3.

Data Lakes

A *Data Lake* is a centralized repository for storing large amounts of structured, semi-structured, and unstructured data without having to define a schema or ingest the data into proprietary internal formats. Data lakes are usually faster at ingesting data, as they do not require transformation right away. They do require the user to write their own transformation piplines.

6 OLAP Commoditization

One recent trend of the last decade is the breakout of OLAP engine sub-systems into standalone open-source components. This is typically done by organizations not in the business of selling DBMS software. These components are generally system catalogs, query optimizers, file format/access libraries, and executions engines.

System Catalogs

A DBMS tracks a database's schema (table, columns) and data files in its catalog. If the DBMS is on the data ingestion path, then it can maintain the catalog incrementally. If an external process adds data files, then it also needs to update the catalog so that the DBMS is aware of them. Notable examples include HCatalog, Google Data Catalog, and Amazon Glue Data Catalog.

Query Optimizers

An extendible search engine framework for heuristic-based and cost-based query optimization. The DBMS provides transformation rules and cost estimates. The framework returns either logical or physical query plan. This is the hardest part to build in any DBMS. Notable examples include Greenplum Orca and Apache Calcite.

Data File Formats

Most DBMSs use a proprietary on-disk binary file format for their databases. The only way to share data between systems is to convert data into a common text-based format, including CSV, JSON, and XML. There are new open-source binary file formats, which cloud vendors and distributed database systems support, that make it easier to access data across systems. Writing a custom file format would give way to better compression and performance, but this gives way to better interoperability.

Notable examples include:

- **Apache Parquet:** Compressed columnar storage from Cloudera/Twitter.
- **Apache ORC:** Compressed columnar storage from **Apache Hive**.
- **Apache CarbonData:** Compressed columnar storage with indexes from Huawei.
- **Apache Iceberg:** Flexible data format that supports schema evolution from Netflix.
- **HDF5:** Multi-dimensional arrays for scientific workloads.
- **Apache Arrow:** In-memory compressed columnar storage from Pandas/Dremio.

Execution Engines

Standalone libraries for executing vectorized query operators on columnar data. The input is a directed, acyclic graph of physical operators. They require external scheduling and orchestration. Notable examples include Velox, DataFusion, and Intel OAP.