Lecture #24

Distributed OLAP Databases
BIFURCATED ENVIRONMENT

OLTP Databases

Extract
Transform
Load

OLAP Database
BIFURCATED ENVIRONMENT

OLTP Databases

Extract
Transform
Load

OLAP Database
BIFURCATED ENVIRONMENT

OLTP Databases

Informatica
Fivetran
talend
Qlik

Extract
Transform
Load

dbt
Airbyte

OLAP Database
BIFURCATED ENVIRONMENT

OLTP Databases

Extract
Transform
Load

Informatica
Fivetran
Talend
Qlik

dbt
Airbyte

OLAP Database
DECISION SUPPORT SYSTEMS

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.

Star Schema vs. Snowflake Schema
STAR SCHEMA

PRODUCT_DIM
- CATEGORY_NAME
- CATEGORY_DESC
- PRODUCT_CODE
- PRODUCT_NAME
- PRODUCT_DESC

SALES_FACT
- PRODUCT_FK
- TIME_FK
- LOCATION_FK
- CUSTOMER_FK
- PRICE
- QUANTITY

CUSTOMER_DIM
- ID
- FIRST_NAME
- LAST_NAME
- EMAIL
- ZIP_CODE

LOCATION_DIM
- COUNTRY
- STATE_CODE
- STATE_NAME
- ZIP_CODE
- CITY

TIME_DIM
- YEAR
- DAY_OF_YEAR
- MONTH_NUM
- MONTH_NAME
- DAY_OF_MONTH
SNOWFLAKE SCHEMA

CAT_LOOKUP
- CATEGORY_ID
- CATEGORY_NAME
- CATEGORY_DESC

PRODUCT_DIM
- CATEGORY_FK
- PRODUCT_CODE
- PRODUCT_NAME
- PRODUCT_DESC

SALES_FACT
- PRODUCT_FK
- TIME_FK
- LOCATION_FK
- CUSTOMER_FK
- PRICE
- QUANTITY

CUSTOMER_DIM
- ID
- FIRST_NAME
- LAST_NAME
- EMAIL
- ZIP_CODE

LOCATION_DIM
- COUNTRY
- STATE_FK
- ZIP_CODE
- CITY

TIME_DIM
- YEAR
- DAY_OF_YEAR
- MONTH_FK
- DAY_OF_MONTH

STATE_LOOKUP
- STATE_ID
- STATE_CODE
- STATE_NAME

MONTH_LOOKUP
- MONTH_NUM
- MONTH_NAME
- MONTH_SEASON
STAR VS. SNOWFLAKE SCHEMA

Issue #1: Normalization
→ Snowflake schemas take up less storage space.
→ Denormalized data models may incur integrity and consistency violations.

Issue #2: Query Complexity
→ Snowflake schemas require more joins to get the data needed for a query.
→ Queries on star schemas will (usually) be faster.
PROBLEM SETUP

Application Server

SELECT * FROM R JOIN S
ON R.id = S.id

Partitions

P1
P2
P3
P4
PROBLEM SETUP

SELECT * FROM R JOIN S
ON R.id = S.id

Application Server

Partitions
PROBLEM SETUP

SELECT * FROM R JOIN S
ON R.id = S.id
TODAY'S AGENDA

Execution Models
Query Planning
Distributed Join Algorithms
Cloud Systems
DISTRIBUTED QUERY EXECUTION

Executing an OLAP query in a distributed DBMS is roughly the same as on a single-node DBMS.
→ Query plan is a DAG of physical operators.

For each operator, the DBMS considers where input is coming from and where to send output.
→ Table Scans
→ Joins
→ Aggregations
→ Sorting
DISTRIBUTED SYSTEM ARCHITECTURE

A distributed DBMS's system architecture specifies the location of the database's data files. This affects how nodes coordinate with each other and where they retrieve/store objects in the database.

Two approaches (not mutually exclusive):
→ Push Query to Data
→ Pull Data to Query
PUSH VS. PULL

Approach #1: Push Query to Data
→ Send the query (or a portion of it) to the node that contains the data.
→ Perform as much filtering and processing as possible where data resides before transmitting over network.

Approach #2: Pull Data to Query
→ Bring the data to the node that is executing a query that needs it for processing.
→ This is necessary when there is no compute resources available where database files are located.
Approach #1: Push Query to Data
→ Send the query (or a portion of it) to the node that contains the data.
→ Perform as much filtering and processing as possible where data resides before transmitting over network.

Approach #2: Pull Data to Query
→ Bring the data to the node that needs it for processing.
→ This is necessary when there is no compute resources available where database files are located.
PUSH VS. PULL

Approach #1: Push Query to Data
→ Send the query (or a portion of it) to the node that contains the data.
→ Perform as much filtering and processing as possible where data resides before transmitting over network.

Approach #2: Pull Data to Query
→ Bring the data to the node that is executing a query that needs it for processing.
→ This is necessary when there is no compute resources available where database files are located.
PUSH QUERY TO DATA

P1 → R.id: 1-100
P1 → S.id: 1-100
P2 → R.id: 101-200
P2 → S.id: 101-200
PUSH QUERY TO DATA

```
SELECT * FROM R JOIN S
  ON R.id = S.id
```
PUSH QUERY TO DATA

```
SELECT * FROM R JOIN S
ON R.id = S.id
```

Application Server

Node

```
P1→R.id:1-100
P1→S.id:1-100
```

```
P2→R.id:101-200
P2→S.id:101-200
```

R ⨝ S
IDs [101,200]
SELECT * FROM R JOIN S ON R.id = S.id

Result: R ⨝ S
PULL DATA TO QUERY

SELECT * FROM R JOIN S
ON R.id = S.id

Node

P1→ID:1-100

Node

P2→ID:101-200

Application Server

Storage
PULL DATA TO QUERY

```
SELECT * FROM R JOIN S
ON R.id = S.id
```

Application Server

Node

```
R ⧶ S
IDs [101,200]
```

Storage

Node

```
P1→ID: 1-100
```

```
P2→ID: 101-200
```
PULL DATA TO QUERY

SELECT * FROM R JOIN S ON R.id = S.id

Application Server

Node

Page ABC

Page XYZ

Storage

Node

Node

P1→ID: 1-100

P2→ID: 101-200

R ∙ S
IDs [101,200]
**PULL DATA TO QUERY**

### SQL Query

```
SELECT * FROM R JOIN S
ON R.id = S.id
```

### Diagram

- **Application Server**
  - Node
    - R $\bowtie S$
      - IDs [101, 200]
    - Page ABC
    - P1→ID: 1-100
  - Node
    - P2→ID: 101-200
- **Storage**
**PULL DATA TO QUERY**

**Application Server**

```
SELECT * FROM R JOIN S
ON R.id = S.id
```

**Node**

```
R ⨝ S
IDs [101,200]
```

**Storage**

**Result: R ⨝ S**

```
P1→ID: 1-100
P2→ID: 101-200
```
OBSERVATION

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 large than the amount of memory available.
→ Ephemeral pages are not persisted after a restart.

What happens to a long-running OLAP query if a node crashes during execution?
QUERY FAULT TOLERANCE

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.

The DBMS could take a snapshot of the intermediate results for a query during execution to allow it to recover if nodes fail.
SELECT * FROM R JOIN S ON R.id = S.id

Application Server

Node

Storage
QUERY FAULT TOLERANCE

SELECT * FROM R JOIN S
ON R.id = S.id
**QUERY FAULT TOLERANCE**

```sql
SELECT * FROM R JOIN S
ON R.id = S.id
```

Application Server

Node

Storage

Result: $R \bowtie S$

$R \bowtie S$
QUERY FAULT TOLERANCE

```
SELECT * FROM R JOIN S ON R.id = S.id
```

Application Server

Node

Storage

![Diagram showing query fault tolerance in a database system](image-url)
**QUERY FAULT TOLERANCE**

```
SELECT * FROM R JOIN S
ON R.id = S.id
```

Node

Result: \( R \Join S \)
QUERY PLANNING

All the optimizations that we talked about before are still applicable in a distributed environment.
→ Predicate Pushdown
→ Projection Pushdown
→ Optimal Join Orderings

Distributed query optimization is even harder because it must consider the physical location of data and network transfer costs.
QUERY PLAN FRAGMENTS

Approach #1: Physical Operators
→ Generate a single query plan and then break it up into partition-specific fragments.
→ Most systems implement this approach.

Approach #2: SQL
→ Rewrite original query into partition-specific queries.
→ Allows for local optimization at each node.
→ SingleStore + Vitess are the only systems we know that use this approach.
QUERY PLAN FRAGMENTS

SELECT * FROM R JOIN S
ON R.id = S.id
QUERY PLAN FRAGMENTS

SELECT * FROM R JOIN S
ON R.id = S.id

SELECT * FROM R JOIN S
ON R.id = S.id
WHERE R.id BETWEEN 1 AND 100

SELECT * FROM R JOIN S
ON R.id = S.id
WHERE R.id BETWEEN 101 AND 200

SELECT * FROM R JOIN S
ON R.id = S.id
WHERE R.id BETWEEN 201 AND 300

id:1-100

id:101-200

id:201-300
Union the output of each join to produce final result.

```
SELECT * FROM R JOIN S
ON R.id = S.id
WHERE R.id BETWEEN 1 AND 100

SELECT * FROM R JOIN S
ON R.id = S.id
WHERE R.id BETWEEN 101 AND 200

SELECT * FROM R JOIN S
ON R.id = S.id
WHERE R.id BETWEEN 201 AND 300
```
OBSERVATION

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.
→ You lose the parallelism of a distributed DBMS.
→ 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 the data is at the node, the DBMS then executes the same join algorithms that we discussed earlier in the semester.

→ Need to produce the correct answer as if all the data is located in a single node system.
SCENARIO #1

One table is replicated at every node. Each node joins its local data in parallel and then sends their results to a coordinating node.

```
SELECT * FROM R JOIN S
ON R.id = S.id
```
SCENARIO #1

One table is replicated at every node. Each node joins its local data in parallel and then sends their results to a coordinating node.

```
SELECT * FROM R JOIN S
ON R.id = S.id
```
SCENARIO #1

One table is replicated at every node. Each node joins its local data in parallel and then sends their results to a coordinating node.

SELECT * FROM R JOIN S ON R.id = S.id
SCENARIO #1

One table is replicated at every node. Each node joins its local data in parallel and then sends their results to a coordinating node.

\[
\text{SELECT } * \text{ FROM } R \text{ JOIN } S \text{ ON } R.id = S.id
\]
SCENARIO #2

Tables are partitioned on the join attribute. Each node performs the join on local data and then sends to a coordinator node for coalescing.

```
SELECT * FROM R JOIN S ON R.id = S.id
```
SCENARIO #2

Tables are partitioned on the join attribute. Each node performs the join on local data and then sends to a coordinator node for coalescing.

```
SELECT * FROM R JOIN S
ON R.id = S.id
```
SCENARIO #2

Tables are partitioned on the join attribute. Each node performs the join on local data and then sends to a coordinator node for coalescing.

```
SELECT * FROM R JOIN S 
ON R.id = S.id
```
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.

```
SELECT * FROM R JOIN S
ON R.id = S.id
```
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.

```
SELECT * FROM R JOIN S
ON R.id = S.id
```
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.

```
SELECT * FROM R JOIN S
ON R.id = S.id
```
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.

```
SELECT * FROM R JOIN S 
ON R.id = S.id
```
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.

```
SELECT * FROM R JOIN S
ON R.id = S.id
```
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.

SELECT * FROM R JOIN S
ON R.id = S.id
SCENARIO #4

Both tables are not partitioned on the join key. The DBMS copies/re-partitions the tables on-the-fly across nodes.

→ This repartitioned data is generally deleted when the query is done.

SELECT * FROM R JOIN S
ON R.id = S.id
SCENARIO #4

Both tables are not partitioned on the join key. The DBMS copies/re-partitions the tables on-the-fly across nodes.

→ This repartitioned data is generally deleted when the query is done.
SCENARIO #4

Both tables are not partitioned on the join key. The DBMS copies/re-partitions the tables on-the-fly across nodes.

→ This repartitioned data is generally deleted when the query is done.

```
SELECT * FROM R JOIN S
ON R.id = S.id
```
SCENARIO #4

Both tables are **not** partitioned on the join key. The DBMS copies/re-partitions the tables on-the-fly across nodes.

→ This repartitioned data is generally deleted when the query is done.

```
SELECT * FROM R JOIN S
ON R.id = S.id
```
SCENARIO #4

Both tables are **not** partitioned on the join key. The DBMS copies/re-partitions the tables on-the-fly across nodes.

→ This repartitioned data is generally deleted when the query is done.

```
SELECT * FROM R JOIN S
ON R.id = S.id
```
**SCENARIO #4**

Both tables are **not** partitioned on the join key. The DBMS copies/re-partitions the tables on-the-fly across nodes.

→ This repartitioned data is generally deleted when the query is done.

```
SELECT * FROM R JOIN S
ON R.id = S.id
```
SCENARIO #4

Both tables are **not** partitioned on the join key. The DBMS copies/re-partitions the tables on-the-fly across nodes.

→ This repartitioned data is generally deleted when the query is done.

```
SELECT * FROM R JOIN S ON R.id = S.id
```
SCENARIO #4

Both tables are not partitioned on the join key. The DBMS copies/re-partitions the tables on-the-fly across nodes.
→ This repartitioned data is generally deleted when the query is done.

```
SELECT * FROM R JOIN S
ON R.id = S.id
```
SEMI-JOIN: REDUCE DATA MOVEMENT

Can use this technique to reduce data movement
→ Before pulling data from another node, send a semi-join filter to reduce data movement.

```
SELECT Fact.price, Dim.*
FROM Fact JOIN Dim
ON Fact.id = Dim.id
WHERE Dim.zip = 15213
```
Can use this technique to reduce data movement
→ Before pulling data from another node, send a semi-join filter to reduce data movement.

```
SELECT Fact.price, Dim.*
FROM Fact JOIN Dim
ON Fact.id = Dim.id
WHERE Dim.zip = 15213
```

\[ D_{semi} = \Pi_{id} (\sigma_{zip = 15213} Dim) \]
Can use this technique to reduce data movement

→ Before pulling data from another node, send a semi-join filter to reduce data movement.

```
SELECT Fact.price, Dim.*
FROM Fact JOIN Dim
ON Fact.id = Dim.id
WHERE Dim.zip = 15213
```

\[ D\text{-semi} = \Pi_{id} (\sigma_{zip = 15213} \text{Dim}) \]
**SEMI-JOIN: REDUCE DATA MOVEMENT**

Can use this technique to reduce data movement

→ Before pulling data from another node, send a semi-join filter to reduce data movement.

$$F\text{-small} = \text{Fact} \bowtie \text{D-semi}$$

$$\text{D-semi} = \Pi_{\text{id}} (\sigma_{\text{zip} = 15213} \text{Dim})$$

```
SELECT Fact.price, Dim.*
FROM Fact JOIN Dim
ON Fact.id = Dim.id
WHERE Dim.zip = 15213
```
Can use this technique to reduce data movement

→ Before pulling data from another node, send a semi-join filter to reduce data movement.

\[
\text{F-small} = \text{Fact} \bowtie \text{D-semi}
\]

\[
\text{D-semi} = \Pi_{id} (\sigma_{\text{zip} = 15213} \text{Dim})
\]

**SEMI-JOIN: REDUCE DATA MOVEMENT**

```
SELECT Fact.price, Dim.*
FROM Fact JOIN Dim
ON Fact.id = Dim.id
WHERE Dim.zip = 15213
```
Can use this technique to reduce data movement

→ Before pulling data from another node, send a semi-join filter to reduce data movement.

```
SELECT Fact.price, Dim.*
FROM Fact JOIN Dim
ON Fact.id = Dim.id
WHERE Dim.zip = 15213
```
Can use this technique to reduce data movement
→ Before pulling data from another node, send a semi-join filter to reduce data movement.

```
SELECT Fact.price, Dim.*
FROM Fact JOIN Dim
ON Fact.id = Dim.id
WHERE Dim.zip = 15213
```

Result = \( \Pi_{\text{price}} (\text{Dim} \bowtie F\text{-small}) \)
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.

→ Example: You can do simple filtering on Amazon S3 before copying data to compute nodes.
CLOUD SYSTEMS

Approach #1: Managed DBMSs
→ No significant modification to the DBMS to be "aware" that it is running in a cloud environment.
→ Examples: Most vendors

Approach #2: Cloud-Native DBMS
→ System designed explicitly to run in a cloud environment.
→ Usually based on a shared-disk architecture.
→ Examples: Snowflake, Google BigQuery
SERVERLESS DATABASES

Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.
SERVERLESS DATABASES

Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.
SERVERLESS DATABASES

Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.
Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.
SERVERLESS DATABASES

Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.
SERVERLESS DATABASES

Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.
SERVERLESS DATABASES

Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.
Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.
SERVERLESS DATABASES

Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.
SERVERLESS DATABASES

Rather than always maintaining compute resources for each customer, a "serverless" DBMS evicts tenants when they become idle.
DATA LAKES

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

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

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.

CREATE TABLE foo (...);

INSERT INTO foo VALUES (...);
DATA LAKES

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

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

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.

```
SELECT * FROM foo
```
DATA LAKES

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.

SELECT * FROM foo
DATA LAKES

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

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.
OLAP COMMODITIZATION

One recent trend of the last decade is the breakout OLAP engine sub-systems into standalone open-source components.

→ This is typically done by organizations not in the business of selling DBMS software.

**Examples:**

→ System Catalogs
→ Query Optimizers
→ File Format / Access Libraries
→ Execution Engines
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 implementations:

→ HCatalog
→ Google Data Catalog
→ Amazon Glue Data Catalog
QUERY OPTIMIZERS

Extendible search engine framework for heuristic- and cost-based query optimization.
→ DBMS provides transformation rules and cost estimates.
→ Framework returns either a logical or physical query plan.

This is the hardest part to build in any DBMS.

Notable implementations:
→ Greenplum Orca
→ Apache Calcite
DATA FILE FORMATS

Most DBMSs use a proprietary on-disk binary file format for their databases.
→ Think of the BusTub page types…

The only way to share data between systems is to convert data into a common text-based format
→ Examples: CSV, JSON, XML

There are new open-source binary file formats that make it easier to access data across systems.
**DATA FILE FORMATS**

**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.
→ Input is a DAG of physical operators.
→ Require external scheduling and orchestration.

Notable implementations:
→ Velox
→ DataFusion
→ Intel OAP
CONCLUSION

The cloud has made the distributed OLAP DBMS market flourish. Lots of vendors. Lots of money.

But more money, more data, more problems...
NEXT CLASS

Review: Come to class. No recording.