Distributed OLAP Databases
ADMINISTRIVIA

Homework #5: Monday Dec 3\textsuperscript{rd} @ 11:59pm

Project #4: Monday Dec 10\textsuperscript{th} @ 11:59pm

Extra Credit: Wednesday Dec 10\textsuperscript{th} @ 11:59pm

Final Exam: Monday Dec 9\textsuperscript{th} @ 5:30pm
ADMINISTRIVIA

Monday Dec 2\textsuperscript{nd} – Oracle Lecture
→ Shasank Chavan (VP In-Memory Databases)

Wednesday Dec 4\textsuperscript{th} – Potpourri + Review
→ Vote for what system you want me to talk about.
→ https://cmudb.io/f19-systems
LAST CLASS

Atomic Commit Protocols
Replication
Consistency Issues (CAP)
Federated Databases
BIFURCATED ENVIRONMENT

OLTP Databases

Extract
Transform
Load

OLAP Database
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

**LOCATION_DIM**
- COUNTRY
- STATE_CODE
- STATE_NAME
- ZIP_CODE
- CITY

**TIME_DIM**
- YEAR
- DAY_OF_YEAR
- MONTH_NUM
- MONTH_NAME
- DAY_OF_MONTH

**CUSTOMER_DIM**
- ID
- FIRST_NAME
- LAST_NAME
- EMAIL
- ZIP_CODE

**SALES_FACT**
- PRODUCT_FK
- TIME_FK
- LOCATION_FK
- CUSTOMER_FK
- PRICE
- QUANTITY
SNOWFLAKE SCHEMA

**PRODUCT_DIM**
- CATEGORY_FK
- PRODUCT_CODE
- PRODUCT_NAME
- PRODUCT_DESC

**LOCATION_DIM**
- COUNTRY
- STATE_FK
- ZIP_CODE
- CITY

**CATEGORY_DIM**
- CATEGORY_ID
- CATEGORY_NAME
- CATEGORY_DESC

**CUSTOMER_DIM**
- ID
- FIRST_NAME
- LAST_NAME
- EMAIL
- ZIP_CODE

**TIME_DIM**
- YEAR
- DAY_OF_YEAR
- MONTH_FK
- DAY_OF_MONTH

**MONTH_LOOKUP**
- MONTH_NUM
- MONTH_NAME
- MONTH_SEASON

**STATE_LOOKUP**
- STATE_ID
- STATE_CODE
- STATE_NAME

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

Partitions

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

Application Server

Partitions

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

Execution Models
Query Planning
Distributed Join Algorithms
Cloud Systems
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.
SELECT * FROM R JOIN S
ON R.id = S.id

Application Server

Node

P1\(\rightarrow\)ID: 1-100

\[101, 200\]

Result: \(R \bowtie S\)

P2\(\rightarrow\)ID: 101-200
**PULL DATA TO QUERY**

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

![Diagram showing the pull data to query process with nodes, application server, and storage.](image)
SELECT * FROM R JOIN S ON R.id = S.id

P1→ID:1-100

Storage

Application Server

Page ABC

Page XYZ

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

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

Application Server

Storage

Node

P1→ID: 1-100

Node

P2→ID: 101-200

R ⊲⊳ S
IDs [101,200]

Result: R ⊲⊳ S

CMU 15-445/645 (Fall 2019)
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.
**QUERY FAULT TOLERANCE**

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

Result: $R \Join S$
**Query Fault Tolerance**

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

Result: \( R \Join S \)
All the optimizations that we talked about before are still applicable in a distributed environment.
→ Predicate Pushdown
→ Early Projections
→ Optimal Join Orderings

Distributed query optimization is even harder because it must consider the location of data in the cluster and data movement costs.
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.
→ MemSQL is the only system that I know that does this.
**QUERY PLAN FRAGMENTS**

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

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

- **Id:101-200**
  ```
  SELECT * FROM R JOIN S
  ON R.id = S.id
  WHERE R.id BETWEEN 101 AND 200
  ```

- **Id:201-300**
  ```
  SELECT * FROM R JOIN S
  ON R.id = S.id
  WHERE R.id BETWEEN 201 AND 300
  ```
Union the output of each join to produce final result.
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.
DISTRIBUTED JOIN ALGORITHMS

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 that we discussed earlier in the semester.
SCENARIO #1

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

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

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

P1: \(R \bowtie S\)

\[
\begin{array}{c}
\text{Id: 1-100} \\
\text{Id: 1-100}
\end{array}
\]

\[
\begin{array}{c}
R\{\text{Id}\} \\
S\{\text{Id}\}
\end{array}
\]

P2: \(R \bowtie S\)

\[
\begin{array}{c}
\text{Id: 101-200} \\
\text{Id: 101-200}
\end{array}
\]

\[
\begin{array}{c}
R\{\text{Id}\} \\
S\{\text{Id}\}
\end{array}
\]

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

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

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

\[
\text{SELECT } * \text{ FROM R JOIN S ON R.id = S.id}
\]
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
```
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 the tables by **reshuffling** them across nodes.

```sql
SELECT * FROM R JOIN S ON R.id = S.id
```
Both tables are not partitioned on the join key. The DBMS copies the tables by **reshuffling** them across 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 the tables by **reshuffling** them across 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 the tables by **reshuffling** them across nodes.

```
SELECT * FROM R JOIN S
ON R.id = S.id
```
Both tables are not partitioned on the join key. The DBMS copies the tables by **reshuffling** them across 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 the tables by **reshuffling** them across nodes.

```
SELECT * FROM R JOIN S
ON R.id = S.id
```
Both tables are not partitioned on the join key. The DBMS copies the tables by reshuffling them across nodes.

\[
\text{SELECT} \hspace{5pt} * \hspace{5pt} \text{FROM} \hspace{5pt} R \hspace{5pt} \text{JOIN} \hspace{5pt} S \hspace{5pt} \text{ON} \hspace{5pt} R.\text{id} = S.\text{id}
\]
SEMI-JOIN

Join operator where the result only contains columns from the left table. Distributed DBMSs use semi-join to minimize the amount of data sent during joins.
→ This is like a projection pushdown.

Some DBMSs support **SEMI JOIN** SQL syntax. Otherwise you fake it with **EXISTS**.

```
SELECT R.id FROM R
  LEFT OUTER JOIN S
  ON R.id = S.id
WHERE R.id IS NOT NULL
```
SEMI-JOIN

Join operator where the result only contains columns from the left table. Distributed DBMSs use semi-join to minimize the amount of data sent during joins. → This is like a projection pushdown.

Some DBMSs support SEMI JOIN SQL syntax. Otherwise you fake it with EXISTS.

```
SELECT R.id FROM R 
LEFT OUTER JOIN S 
ON R.id = S.id 
WHERE R.id IS NOT NULL
```
SEMI-JOIN

Join operator where the result only contains columns from the left table. Distributed DBMSs use semi-join to minimize the amount of data sent during joins.

→ This is like a projection pushdown.

Some DBMSs support **SEMI JOIN** SQL syntax. Otherwise you fake it with **EXISTS**.

```
SELECT R.id FROM R
  LEFT OUTER JOIN S
  ON R.id = S.id
WHERE R.id IS NOT NULL
```

```
SELECT R.id FROM R
WHERE EXISTS (  
  SELECT 1 FROM S  
  WHERE R.id = S.id
)
```
Semi-join

Join operator where the result only contains columns from the left table. Distributed DBMSs use semi-join to minimize the amount of data sent during joins.

→ This is like a projection pushdown.

Some DBMSs support **SEMI JOIN** SQL syntax. Otherwise you fake it with **EXISTS**.
Like a natural join except that the attributes on the right table that are not used to compute the join are restricted.

Syntax: \((R \bowtie S)\)
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.
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
→ The system is designed explicitly to run in a cloud environment.
→ Usually based on a shared-disk architecture.
→ Examples: Snowflake, Google BigQuery, Amazon Redshift, Microsoft SQL Azure
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.
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.
DISAGGREGATED COMPONENTS

System Catalogs
→ HCatalog, Google Data Catalog, Amazon Glue Data Catalog

Node Management
→ Kubernetes, Apache YARN, Cloud Vendor Tools

Query Optimizers
→ Greenplum Orca, Apache Calcite
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.
**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.
CONCLUSION

More money, more data, more problems...

Cloud OLAP Vendors:
- Amazon Redshift
- Oracle
- Snowflake
- Microsoft SQL Azure

On-Premise OLAP Systems:
- ClickHouse
- Splice Machine
- Vertica
- Oracle Exadata
- Presto
- Greenplum
- Teradata
NEXT CLASS

Oracle Guest Speaker