Distributed OLAP Databases
Homework #5: Will be released today. It is due Thursday Dec 2\textsuperscript{nd} @ 11:59pm.

Project #4: Due Sunday Dec 5\textsuperscript{th} @ 11:59pm.

Guest Lecture from Google BigQuery: Monday Nov 29\textsuperscript{th} @ 3:05pm. Attendance required.

Final Exam: Friday Dec 10\textsuperscript{th} @ 8:30am at Doherty Hall 2210. Bring pencil and rubber.
UPCOMING DATABASE TALK

Query Optimization and Acceleration at Dremio
→ Mon Nov 22\textsuperscript{ed} @ 4:30pm ET

Research Talk on Google BigQuery
→ Tue Nov 30\textsuperscript{th} @ 12:00pm ET
LAST CLASS

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

OLTP Databases

OLAP Database
BIFURCATED ENVIRONMENT

OLTP Databases

- Extract
- Transform
- Load

OLAP Database
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

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

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

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

**PRODUCT_DIM**
- CATEGORY_NAME
- CATEGORY_DESC
- PRODUCT_CODE
- PRODUCT_NAME
- PRODUCT_DESC

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

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

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

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

**CATEGORY_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
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.
Application Server

Partitions

P1

P2

P3

P4
PROBLEM SETUP

Application Server

Partitions

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

```
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.
PUSH QUERY TO DATA

Node

Application Server

Node

P1→R.id:1-100
P1→S.id:1-100

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

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

R ⨝ S
IDs [101, 200]

Node

Application Server

Node

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

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]

Result: R ⨝ S
PULL DATA TO QUERY

**Application Server**

**Storage**

**Node**

- **P1** → **ID: 1-100**
- **P2** → **ID: 101-200**
PULL DATA TO QUERY

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

Node

<table>
<thead>
<tr>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1→ID:1-100</td>
</tr>
<tr>
<td>Node</td>
</tr>
<tr>
<td>P2→ID:101-200</td>
</tr>
</tbody>
</table>

Application Server

Storage
**PULL DATA TO QUERY**

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

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

Node

Page ABC

Page XYZ

Application Server

Storage

R ⨝ S
IDs [101,200]

Node

P1→ID:1-100

P2→ID:101-200
SELECT * FROM R JOIN S ON R.id = S.id

Storage

Node

P1→ID: 1-100

R \Join S
IDs [101,200]

Result: R \Join S

Node

P2→ID: 101-200

Application Server
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?
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
SELECT * FROM R JOIN S ON R.id = S.id
SELECT * FROM R JOIN S ON R.id = S.id
SEL\text{EC}T \ast \ FRO\text{M} R \ JO\text{IN} S \n\O\text{N} R.\text{id} = S.\text{id}

\text{Result: } R \bowtie S
SELECT * FROM R JOIN S ON R.id = S.id
**QUERY FAULT TOLERANCE**

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

Result: $R \bowtie S$
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 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 that I know of that use this approach.
QUERY PLAN FRAGMENTS

```
SELECT * FROM R JOIN S
  ON R.id = S.id
```
Union the output of each join to produce the 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 the data is at the node, the DBMS 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 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
```

Diagram:
- P1: $R \bowtie S$
  - Id: 1-100
  - Replicated

- P2: $R \bowtie S$
  - Id: 101-200
  - Replicated
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 #2

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

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

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

```sql
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.

\[
\text{SELECT} \quad \ast \quad \text{FROM} \quad R \quad \text{JOIN} \quad S \\
\text{ON} \quad R.\text{id} = S.\text{id}
\]
SCENARIO #4

Both tables are *not* partitioned on the join key. The DBMS copies the tables by "shuffling" 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 "shuffling" 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 "shuffling" 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 "shuffling" them across nodes.

```sql
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 "shuffling" 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 "shuffling" 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 "shuffling" 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 "shuffling" them across nodes.

```
SELECT * FROM R JOIN S
ON 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**.
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 JOIN S ON R.id = S.id WHERE R.id IS NOT NULL
S E M I - J O I N

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.
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**.

```sql
SELECT R.id FROM R JOIN S ON R.id = S.id WHERE R.id IS NOT NULL
```
S E M I - J O I N

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 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**.
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`.

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

```sql
SELECT R.id FROM
  R
WHERE EXISTS (
  SELECT 1 FROM S
  WHERE R.id = S.id)
```
S E M I - J O I N

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  
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)
```
S E M I - J O I N

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**.
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
→ 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
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.
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.
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.
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.
UNIVERSAL 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.
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
- Teradata
NEXT CLASS

Google Guest Speaker