UPCOMING DATABASE EVENTS

**Swarm64 Tech Talk**
→ Thursday November 29th @ 12pm
→ GHC 8102 ← Different Location!

**VoltDB Research Talk**
→ Monday December 3rd @ 4:30pm
→ GHC 8102
OLTP VS. OLAP

On-line Transaction Processing (OLTP):
→ Short-lived read/write txns.
→ Small footprint.
→ Repetitive operations.

On-line Analytical Processing (OLAP):
→ Long-running, read-only queries.
→ Complex joins.
→ Exploratory queries.
BIFURCATED ENVIRONMENT

Extract
Transform
Load

OLTP Databases

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

**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
SNOWFLAKE SCHEMA

PRODUCT_DIM
- CATEGORY_FK
- PRODUCT_CODE
- PRODUCT_NAME
- PRODUCT_DESC

LOCATION_DIM
- COUNTRY
- STATE_FK
- ZIP_CODE
- CITY

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

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

CATEGORY_LOOKUP
- CATEGORY_ID
- CATEGORY_NAME
- CATEGORY_DESC

STATE_LOOKUP
- STATE_ID
- STATE_CODE
- STATE_NAME
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**

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

Application Server

Node

P1 → ID: 1-100

Node

P2 → ID: 101-200

R ⊙ S
IDs [101,200]

Result: R ⊙ S

CMU 15-445/645 (Fall 2018)
PULL DATA TO QUERY

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

Application
Server

Storage

Node

Page ABC

Page XYZ

Node

R ⊗ S
IDs [101,200]

P1→ID: 1-100

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

Node

R ⊙ S
IDs [101,200]

Page ABC

Page XYZ

Node

P1→ID: 1-100

Application Server

Storage

P2→ID: 101-200

CMU 15-445/645 (Fall 2018)
PULL DATA TO QUERY

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

Result: R ⨝ S
Traditional distributed OLAP DBMSs were designed to assume that nodes will not fail during query execution.

→ If the DBMS 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 after a crash.
QUERY PLANNING

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

But now the DBMS must also consider the location of data at each partition when optimizing
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.
→ MemSQL is the only system that I know that does this.
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
Id:1-100
```

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

```
SELECT * FROM R JOIN S
ON R.id = S.id
WHERE R.id BETWEEN 201 AND 300
Id:201-300
```
Union the output of each join together 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.

```
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 #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.

```
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 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
```
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
```
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
```
RELATIONAL ALGEBRA: SEMI-JOIN

Like a natural join except that the attributes that are not used to compute the join are restricted.

Syntax: \((R \bowtie S)\)

Distributed DBMSs use semi-join to minimize the amount of data sent during joins. This is the same as a projection pushdown.
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.
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
UNIVERSAL FORMATS

Traditional DBMSs store data in proprietary binary file formats that are incompatible. One can use text formats (XML/JSON/CSV) to share data across different systems.

There are now standardized file formats.
UNIVERSAL FORMATS

Apache Parquet
→ Compressed columnar storage from Cloudera/Twitter

Apache ORC
→ Compressed columnar storage from Apache Hive.

HDF5
→ Multi-dimensional arrays for scientific workloads.

Apache Arrow
→ In-memory compressed columnar storage from Pandas/Dremio
CONCLUSION

Again, efficient distributed OLAP systems are difficult to implement.

More data, more problems...
NEXT CLASS

VoltDB Guest Speaker