Introduction to Data Management
Parallel Processing

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle
Announcements

- HW 7 out this weekend
  - Topic is Spark/MapReduce, next week
  - Uses AWS, you will sign up for trial credits
- Today is introduction to parallel processing in general
Course Context

- Phase 1: Core RDBMS
  - SQL and RA
  - Logical and Physical Database Design
  - Transactions

- Phase 2: Parallel Data Management
  - Distributed Relational Databases
  - Spark query language

- Phase 3: NoSQL
Humans have a tendency to tackle problems that are too big to compute

- Breaking the enigma code (WWII)
  - Using automation (the bombe)
- Computing rocket trajectories (Space Race)
  - Using programming languages (FORTRAN)
- Now: Data driven applications
  - Protein folding
  - Internet of things
  - Financial forecasting
  - Weather prediction
  - Social media platforms
  - ...
More Data, More Problems

- The rates at which we generate and use information have **outpaced the capabilities of a single computer**
- Problems:
  - Need more speed
  - Need more scale
Big Data

Volume is not an issue

- Databases do parallelize easily; techniques available from the 80’s
  - Data partitioning
  - Parallel query processing

- SQL is embarrassingly parallel

New workloads are an issue

- More complex Machine Learning, e.g. click prediction, topic modeling, SVM, k-means
- Requires innovation – Active research area
Great for one application (maybe more) and one user.
DBMS Deployment: Client/Server

Great for many apps and many users

Data files

DB Server

connection
(ODBC, JDBC)

Applications
DBMS Deployment: 3 Tiers

Great for Web-based Applications

Data files → DB Server → Web Server & App Server → Browser

Connection (e.g., JDBC)

HTTP/SSL
All infrastructure is hosted on virtual machines
How to Scale?

DB Server

Connection (e.g., JDBC)

http multiplex

HTTP/SSL

Browser

Use many Web servers: Easy!
How to Scale?

Many DBMS instances: HARD

Web Server Farm

Connection (e.g., JDBC)

http multiplex

HTTP/SSL

Browser
How to Scale?

- We can easily replicate the web servers and the application servers.

- We cannot so easily replicate the database servers, because the database is unique.

- We need to design ways to scale up the DBMS.
How to Scale a DBMS?

Scale out

(one machine)

A more powerful server

More servers, one database
Speed Up

**Speed up:**

same data, more nodes $\rightarrow$ higher speed

![Graph showing ideal linear speedup](image-url)
Scale up:
more data, more nodes $\rightarrow$ same speed

![Graph showing query speed vs. larger data and number of computing nodes]

Ideal-linear scaleup
Parallel computing is not a magic bullet

Common reasons for sublinear performance:
- **Overhead cost**
  - Starting and coordinating operations on many nodes
- **Interference/Contention**
  - Shared resources are not perfectly split
- **Skew**
  - Process is only as fast as the slowest node
Implementations for Database Parallelism

- **Architecture Parallelism**
  - Shared Memory
  - Shared Nothing*

- **Query Parallelism**
  - Inter-Query Parallelism
    "parallelism between queries”
  - Inter-Operator Parallelism*
    "parallelism within queries”

(*) The version we will learn to use
Implementations for Database Parallelism

- **Architecture Parallelism**
  - Shared Memory
  - Shared Nothing*

- **Query Parallelism**
  - Inter-Query Parallelism
    "parallelism between queries"
  - Intra-Operator Parallelism*
    "parallelism within queries"

(*) The version we will learn to use
Shared-Memory Architecture

- Shared main memory and disks
- Your laptop or desktop uses this architecture
- Expensive to scale
- Easiest to implement on

**Diagram:**

- **P** (Processor)
- **D** (Disk)
- **Interconnection Network** (Motherboard)
- **Global Memory**

**Software:**
- Microsoft SQL Server
- PostgreSQL
- SQLite
- MySQL
Shared-Nothing Architecture*

- Uses cheap, commodity hardware
- No contention for memory and high availability
- Theoretically can scale infinitely
- Hardest to implement on

Interconnection Network (TCP)

- P
- M
- D

Parallel Processing
Implementations for Database Parallelism

- Architecture Parallelism
  - Shared Memory
  - Shared Nothing*

- Query Parallelism
  - Inter-Query Parallelism
    "parallelism between queries"
  - Intra-Operator Parallelism*
    "parallelism within queries"

(*) The version we will learn to use
Inter-Query Parallelism

- Each transaction is processed on a separate node
- Scales very well for **lots of simple transactions**
Intra-Query Parallelism*

- Specifically intra-operator parallelism
- Each operator is processed by multiple nodes
- Scales well in general

$P \bowtie \{ Q \mid Q \subseteq R \wedge P.CID = R.CID \}$

Payroll $P$  \quad Regist $R$  \quad Car $C$
From here, we will assume a system that consists of multiple commodity machines on a common network where nodes may carry out specified relational operations.

New problem: Where does the data go?
Unpartitioned Table

- Entire table on just one node in the system
- Simplest choice if data can fit on a single node
- Might result in being a bottleneck
Block Partitioning

Tuples are horizontally partitioned by raw size (no ordering considered)

\[ B(R) = K \]

\[ B(R_1) = K/N \]

\[ B(R_2) = K/N \]

\[ B(R_N) = K/N \]

N nodes
Hash Partitioning

Node contains tuples with chosen attribute hashes

\[ R_1, 1 = h(A) \% N \]
\[ R_2, 2 = h(A) \% N \]
\[ R_N, 0 = h(A) \% N \]
Range Partitioning

Node contains tuples in chosen attribute ranges

\[ R_1, -\infty < A \leq v_1 \]
\[ R_2, v_1 < A \leq v_2 \]
\[ R_N, v_N < A < \infty \]
The Justin Bieber Effect

- Hashing data to nodes is very good when the attribute chosen better approximates a uniform distribution

- Keep in mind: Certain nodes will become bottlenecks if a poorly chosen attribute is hashed
Partitioned Aggregation

1. Hash shuffle tuples
2. Local aggregation

Assume:
R is block partitioned

```
SELECT * 
FROM R 
GROUP BY R.A 
```
Partitioned Aggregation

1. Hash shuffle tuples
2. Local aggregation

Assume:
R is block partitioned

```
SELECT * 
FROM R 
GROUP BY R.A
```
Partitioned Aggregation

1. Hash shuffle tuples
2. Local aggregation

Assume:
R is block partitioned
SELECT *
FROM R
GROUP BY R.A

\[ \gamma_{R.A} \]

\[ \gamma_{R.A} \]

\[ \gamma_{R.A} \]
Partitioned Aggregation

1. Hash shuffle tuples
2. Local aggregation

Assume:
R is block partitioned

\[
\begin{align*}
\text{SELECT} &\quad \text{*} \\
\text{FROM} &\quad R \\
\text{GROUP BY} &\quad R.A \\
\end{align*}
\]
**Partitioned Aggregation**

1. Hash shuffle tuples
2. Local aggregation

Assume:
- \( R \) is block partitioned
- \[
\text{SELECT * FROM } R \text{ GROUP BY } R.A
\]
Partitioned Aggregation

1. Hash shuffle tuples
2. Local aggregation

Would I need to shuffle if R was hash or range partitioned?
Implicit Union

Parallel query plans implicitly union at the end
1. Hash shuffle tuples on join attributes
2. Local join

Assume:
R and S are block partitioned

```sql
SELECT *
FROM R, S
WHERE R.A = S.A
```

Node 1 Node 2 Node 3
Partitioned Hash Equijoin Algorithm

1. Hash shuffle tuples on join attributes
2. Local join

Assume:
R and S are block partitioned

```
SELECT *
FROM R, S
WHERE R.A = S.A
```
Next Time

- MapReduce
- Programming with the Java Spark API