Query Execution – Part II
Project #2 is due Sunday, Oct 17th @ 11:59pm

Homework #3 is due Sunday, Oct 24th @ 11:59pm

Midterm Exam is Wednesday, Oct 13th
→ During regular class time @ 3:05-4:25pm
→ Open book / open notes
→ Will include all material covered before midterm
→ See Piazza post for more details
We discussed in the last class how to compose operators together into a plan to execute an arbitrary query.

We assumed that the queries execute with a single worker (e.g., a thread).

We will now discuss how to execute queries using multiple workers.

```
SELECT R.id, S.cdate
FROM R
JOIN S
ON R.id = S.id
WHERE S.value > 100
```
WHY CARE ABOUT PARALLEL EXECUTION?
WHY CARE ABOUT PARALLEL EXECUTION?

Increased performance
→ Throughput
→ Latency
WHY CARE ABOUT PARALLEL EXECUTION?

Increased performance
→ Throughput
→ Latency

Increased responsiveness and availability
WHY CARE ABOUT PARALLEL EXECUTION?

Increased performance
→ Throughput
→ Latency

Increased responsiveness and availability

Potentially lower *total cost of ownership* (TCO)
Database is spread out across multiple **resources** to improve different aspects of the DBMS.

Appears as a single logical database instance to the application, regardless of physical organization. → SQL query for a single-resource DBMS should generate same result on a parallel or distributed DBMS.
Parallel DBMSs
→ Resources are physically close to each other.
→ Resources communicate over high-speed interconnect.
→ Communication is assumed to be cheap and reliable.

Distributed DBMSs
→ Resources can be far from each other.
→ Resources communicate using slow(er) interconnect.
→ Communication cost and problems cannot be ignored.
TODAY'S AGENDA

Process Models
Execution Parallelism
I/O Parallelism
A DBMS’s **process model** defines how the system is architected to support concurrent requests from a multi-user application.

A **worker** is the DBMS component that is responsible for executing tasks on behalf of the client and returning the results.
PROCESS MODEL

Approach #1: Process per DBMS Worker

Approach #2: Process Pool

Approach #3: Thread per DBMS Worker
Each worker is a separate OS process.
→ Relies on OS scheduler.
→ Use shared-memory for global data structures.
→ A process crash doesn’t take down entire system.
→ Examples: IBM DB2, Postgres, Oracle
Each worker is a separate OS process.
→ Relies on OS scheduler.
→ Use shared-memory for global data structures.
→ A process crash doesn’t take down entire system.
→ Examples: IBM DB2, Postgres, Oracle
Each worker is a separate OS process.
→ Relies on OS scheduler.
→ Use shared-memory for global data structures.
→ A process crash doesn’t take down entire system.
→ Examples: IBM DB2, Postgres, Oracle
Each worker is a separate OS process.
→ Relies on OS scheduler.
→ Use shared-memory for global data structures.
→ A process crash doesn’t take down entire system.
→ Examples: IBM DB2, Postgres, Oracle
Each worker is a separate OS process.
→ Relies on OS scheduler.
→ Use shared-memory for global data structures.
→ A process crash doesn’t take down entire system.
→ Examples: IBM DB2, Postgres, Oracle
A worker uses any free process from the pool.
→ Still relies on OS scheduler and shared memory.
→ Bad for CPU cache locality.
→ Examples: IBM DB2, Postgres (2015)
A worker uses any free process from the pool.
→ Still relies on OS scheduler and shared memory.
→ Bad for CPU cache locality.
→ Examples: IBM DB2, Postgres (2015)
Thread per Worker

Single process with multiple worker threads.
→ DBMS manages its own scheduling.
→ May or may not use a dispatcher thread.
→ Thread crash (may) kill the entire system.
→ Examples: IBM DB2, MSSQL, MySQL, Oracle (2014)
Advantages of a multi-threaded architecture:
→ Less overhead per context switch.
→ Do not have to manage shared memory.

The thread per worker model does not mean that the DBMS supports intra-query parallelism.
SCHEDULING

For each query plan, the DBMS decides where, when, and how to execute it.
→ How many tasks should it use?
→ How many CPU cores should it use?
→ What CPU core should the tasks execute on?
→ Where should a task store its output?

The DBMS *always* knows more than the OS.
Inter-Query: Different queries are executed concurrently.  
→ Increases throughput & reduces latency.

Intra-Query: Execute the operations of a single query in parallel.  
→ Decreases latency for long-running queries.
INTER-QUERY PARALLELISM

Improve overall performance by allowing multiple queries to execute simultaneously.

If queries are read-only, then this requires little coordination between queries.

If multiple queries are updating the database at the same time, then this is hard to do correctly...
INTER-QUERY PARALLELISM

Improve overall performance by allowing multiple queries to execute simultaneously.

If queries are read-only, then this requires little coordination between queries.

If multiple queries are updating the database at the same time, then this is hard to do correctly...
INTRA-QUERY PARALLELISM

Improve the performance of a single query by executing its operators in parallel.

Think of organization of operators in terms of a \textit{producer/consumer} paradigm.

There are parallel versions of every operator.
\rightarrow Can either have multiple threads access centralized data structures or use partitioning to divide work up.
PARALLEL GRACE HASH JOIN

Use a separate worker to perform the join for each level of buckets for $R$ and $S$ after partitioning.
PARALLEL GRACE HASH JOIN

Use a separate worker to perform the join for each level of buckets for R and S after partitioning.
Use a separate worker to perform the join for each level of buckets for \( R \) and \( S \) after partitioning.
INTRA-QUERY PARALLELISM

Approach #1: Intra-Operator (Horizontal)

Approach #2: Inter-Operator (Vertical)

Approach #3: Bushy
INTRA-OPERATOR PARALLELISM

Approach #1: Intra-Operator (Horizontal)
→ Decompose operators into independent fragments that perform the same function on different subsets of data.

The DBMS inserts an exchange operator into the query plan to coalesce/split results from multiple children/parent operators.
INTRA-OPERATOR PARALLELISM

SELECT * FROM A
WHERE A.value > 99
INTRA-OPERATOR PARALLELISM

SELECT * FROM A
WHERE A.value > 99

\[ \sigma_{\text{value}>99} \]

\[ \sigma \]

A

\[ \sigma \]

\[ \sigma \]

Exchange

\[ A_1 \]

\[ A_2 \]

\[ A_3 \]

1

2

3

4

5

Pages
INTRA-OPERATOR PARALLELISM

SELECT * FROM A
WHERE A.value > 99
SELECT * FROM A
WHERE A.value > 99

INTRA-OPERATOR PARALLELISM

\[\sigma\text{value}>99\]
INTRA-OPERATOR PARALLELISM

```
SELECT * FROM A
WHERE A.value > 99
```
INTRA-OPERATOR PARALLELISM

SELECT * FROM A
WHERE A.value > 99

σ

σ

σ

Exchange

A

A

A

value>99

1 2 3

4 5

Pages

Next

Next
INTRA-OPERATOR PARALLELISM

SELECT * FROM A
WHERE A.value > 99

A

value>99

A

Exchange

A

σ

A

σ

A

σ

A

Pages

1
2
3
4
5
**SELECT * FROM A
WHERE A.value > 99**
INTRA-OPERATOR PARALLELISM

SELECT * FROM A
WHERE A.value > 99

\[ \sigma_{value>99} A \]

\[ \sigma_1 A_1 \]
\[ \sigma_2 A_2 \]
\[ \sigma_3 A_3 \]

Exchange

Pages

1 2 3 4 5
INTRA-OPERATOR PARALLELISM

SELECT * FROM A
WHERE A.value > 99
Exchange Type #1 – Gather
→ Combine the results from multiple workers into a single output stream.

Exchange Type #2 – Distribute
→ Split a single input stream into multiple output streams.

Exchange Type #3 – Repartition
→ Shuffle multiple input streams across multiple output streams.

Source: Craig Freedman
INTRA-OPERATOR PARALLELISM

SELECT A.id, B.value
FROM A JOIN B
ON A.id = B.id
WHERE A.value < 99
AND B.value > 100
INTRA-OPERATOR PARALLELISM

```
SELECT A.id, B.value
FROM A JOIN B
    ON A.id = B.id
WHERE A.value < 99
    AND B.value > 100
```
INTRA-OPERATOR PARALLELISM

```sql
SELECT A.id, B.value
FROM A JOIN B
    ON A.id = B.id
WHERE A.value < 99
AND B.value > 100
```
SELECT A.id, B.value
FROM A JOIN B
ON A.id = B.id
WHERE A.value < 99
AND B.value > 100
INTRA-OPERATOR PARALLELISM

```
SELECT A.id, B.value
FROM A JOIN B
ON A.id = B.id
WHERE A.value < 99
AND B.value > 100
```
SELECT A.id, B.value
FROM A JOIN B
ON A.id = B.id
WHERE A.value < 99
AND B.value > 100
INTRA-OPERATOR PARALLELISM

```sql
SELECT A.id, B.value
FROM A JOIN B
ON A.id = B.id
WHERE A.value < 99
AND B.value > 100
```
**INTRA-OPERATOR PARALLELISM**

```sql
SELECT A.id, B.value
FROM A JOIN B
ON A.id = B.id
WHERE A.value < 99
AND B.value > 100
```
INTRA-OPERATOR PARALLELISM

```
SELECT A.id, B.value
FROM A JOIN B
ON A.id = B.id
WHERE A.value < 99
AND B.value > 100
```
INTRA-OPERATOR PARALLELISM

```
SELECT A.id, B.value
FROM A JOIN B
ON A.id = B.id
WHERE A.value < 99
AND B.value > 100
```
SELECT A.id, B.value
FROM A JOIN B
ON A.id = B.id
WHERE A.value < 99
AND B.value > 100
SELECT A.id, B.value
FROM A JOIN B
    ON A.id = B.id
WHERE A.value < 99
    AND B.value > 100
SELECT A.id, B.value
FROM A JOIN B
ON A.id = B.id
WHERE A.value < 99
AND B.value > 100
**INTRA-OPERATOR PARALLELISM**

```sql
SELECT A.id, B.value
FROM A JOIN B
ON A.id = B.id
WHERE A.value < 99
AND B.value > 100
```
INTRA-OPERATOR PARALLELISM

```sql
SELECT A.id, B.value
FROM A JOIN B
  ON A.id = B.id
WHERE A.value < 99
  AND B.value > 100
```
**INTRA-OPERATOR PARALLELISM**

SELECT A.id, B.value
FROM A JOIN B
ON A.id = B.id
WHERE A.value < 99
AND B.value > 100
INTER-OPERATOR PARALLELISM

Approach #2: Inter-Operator (Vertical)

→ Operations are overlapped in order to pipeline data from one stage to the next without materialization.
→ Workers execute operators from different segments of a query plan at the same time.

Also called pipeline parallelism.
INTER-OPERATOR PARALLELISM

```
SELECT A.id, B.value
FROM A JOIN B
ON A.id = B.id
WHERE A.value < 99
AND B.value > 100
```
SELECT A.id, B.value
FROM A JOIN B
ON A.id = B.id
WHERE A.value < 99
AND B.value > 100
\[
\begin{align*}
\text{SELECT } & \ A.id, B.value \\
\text{FROM } & \ A \ JOIN \ B \\
\text{ON } & \ A.id = B.id \\
\text{WHERE } & \ A.value < 99 \\
\text{AND } & \ B.value > 100 \\
\end{align*}
\]

\[
\begin{align*}
\text{for } r_1 \in \text{outer:} \\
\text{for } r_2 \in \text{inner:} \\
\text{emit}(r_1 \Join r_2)
\end{align*}
\]
SELECT A.id, B.value
FROM A JOIN B
ON A.id = B.id
WHERE A.value < 99
AND B.value > 100

\[
\pi \\
\sigma \\
\sigma \\
A \\
\bowtie \\
B
\]

\[
\pi \\
\sigma \\
\sigma \\
A \\
\bowtie \\
B
\]

for \( r \in \text{incoming} \):
\[
\text{emit}(\pi r)
\]

for \( r_1 \in \text{outer} \):
\[
\text{for } r_2 \in \text{inner}:
\text{emit}(r_1 \bowtie r_2)
\]
INTER-OPERATOR PARALLELISM

SELECT A.id, B.value
FROM A JOIN B
ON A.id = B.id
WHERE A.value < 99
AND B.value > 100

for \( r_1 \in \text{outer} \):
for \( r_2 \in \text{inner} \):
emit \((r_1 \Join r_2)\)

for \( r \in \text{incoming} \):
emit(\(\pi r\))
Approach #3: Bushy Parallelism

→ Hybrid of intra- and inter-operator parallelism where workers execute multiple operators from different segments of a query plan at the same time.

→ Still need exchange operators to combine intermediate results from segments.
**Bushy Parallelism**

**Approach #3: Bushy Parallelism**

→ Hybrid of intra- and inter-operator parallelism where workers execute multiple operators from different segments of a query plan at the same time.

→ Still need exchange operators to combine intermediate results from segments.

```
SELECT * 
FROM A JOIN B JOIN C JOIN D
```
Approach #3: Bushy Parallelism

→ Hybrid of intra- and inter-operator parallelism where workers execute multiple operators from different segments of a query plan at the same time.

→ Still need exchange operators to combine intermediate results from segments.

\[
\text{SELECT } * \\
\text{FROM A JOIN B JOIN C JOIN D}
\]
Using additional processes/threads to execute queries in parallel won't help if the disk is always the main bottleneck.

→ In fact, it can make things worse if each worker is working on different segments of the disk.
I/O PARALLELISM

Split the DBMS across multiple storage devices.
→ Multiple Disks per Database
→ One Database per Disk
→ One Relation per Disk
→ Split Relation across Multiple Disks
→ ...
MULTI-DISK PARALLELISM

Configure OS/hardware to store the DBMS's files across multiple storage devices.
→ Storage Appliances
→ RAID Configuration

This is **transparent** to the DBMS.
MULTI-DISK PARALLELISM

Configure OS/hardware to store the DBMS's files across multiple storage devices.
→ Storage Appliances
→ RAID Configuration

This is **transparent** to the DBMS.

RAID 0 (Striping)
Configure OS/hardware to store the DBMS's files across multiple storage devices.
→ Storage Appliances
→ RAID Configuration

This is **transparent** to the DBMS.

**MULTI-DISK PARALLELISM**

RAID 1 (Mirroring)
Some DBMSs allow you to specify the disk location of each individual database.
→ The buffer pool manager maps a page to a disk location.

This is also easy to do at the filesystem level if the DBMS stores each database in a separate directory.
→ The DBMS recovery log file might still be shared if transactions can update multiple databases.
PARTITIONING

Split single logical table into disjoint physical segments that are stored/managed separately.

Partitioning should (ideally) be transparent to the application.
→ The application should only access logical tables and not have to worry about how things are physically stored.
VERTICAL PARTITIONING

Store a table’s attributes in a separate location (e.g., file, disk volume).
Must store tuple information to reconstruct the original record.

```
CREATE TABLE foo (  
    attr1 INT,  
    attr2 INT,  
    attr3 INT,  
    attr4 TEXT  
);
```
VERTICAL PARTITIONING

Store a table’s attributes in a separate location (e.g., file, disk volume).
Must store tuple information to reconstruct the original record.

```
CREATE TABLE foo (
    attr1 INT,
    attr2 INT,
    attr3 INT,
    attr4 TEXT
);
```
Store a table’s attributes in a separate location (e.g., file, disk volume).
Must store tuple information to reconstruct the original record.

**Partition #1**

<table>
<thead>
<tr>
<th>Tuple#1</th>
<th>attr1</th>
<th>attr2</th>
<th>attr3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuple#2</td>
<td>attr1</td>
<td>attr2</td>
<td>attr3</td>
</tr>
<tr>
<td>Tuple#3</td>
<td>attr1</td>
<td>attr2</td>
<td>attr3</td>
</tr>
<tr>
<td>Tuple#4</td>
<td>attr1</td>
<td>attr2</td>
<td>attr3</td>
</tr>
</tbody>
</table>

**Partition #2**

<table>
<thead>
<tr>
<th>Tuple#1</th>
<th>attr4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuple#2</td>
<td>attr4</td>
</tr>
<tr>
<td>Tuple#3</td>
<td>attr4</td>
</tr>
<tr>
<td>Tuple#4</td>
<td>attr4</td>
</tr>
</tbody>
</table>

```sql
CREATE TABLE foo ( 
    attr1 INT, 
    attr2 INT, 
    attr3 INT, 
    attr4 TEXT 
);
```
HORIZONTAL PARTITIONING

Divide table into disjoint segments based on some partitioning key.
→ Hash Partitioning
→ Range Partitioning
→ Predicate Partitioning

CREATE TABLE foo ( attr1 INT, attr2 INT, attr3 INT, attr4 TEXT );

<table>
<thead>
<tr>
<th>Tuple#1</th>
<th>attr1</th>
<th>attr2</th>
<th>attr3</th>
<th>attr4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>attr1</td>
<td>attr2</td>
<td>attr3</td>
<td>attr4</td>
</tr>
<tr>
<td>Tuple#2</td>
<td>attr1</td>
<td>attr2</td>
<td>attr3</td>
<td>attr4</td>
</tr>
<tr>
<td>Tuple#3</td>
<td>attr1</td>
<td>attr2</td>
<td>attr3</td>
<td>attr4</td>
</tr>
<tr>
<td>Tuple#4</td>
<td>attr1</td>
<td>attr2</td>
<td>attr3</td>
<td>attr4</td>
</tr>
</tbody>
</table>
HORIZONTAL PARTITIONING

Divide table into disjoint segments based on some partitioning key.
→ Hash Partitioning
→ Range Partitioning
→ Predicate Partitioning

CREATE TABLE foo ( attr1 INT, attr2 INT, attr3 INT, attr4 TEXT );

<table>
<thead>
<tr>
<th>Tuple#1</th>
<th>attr1</th>
<th>attr2</th>
<th>attr3</th>
<th>attr4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>attr1</td>
<td>attr2</td>
<td>attr3</td>
<td>attr4</td>
</tr>
<tr>
<td>Tuple#2</td>
<td>attr1</td>
<td>attr2</td>
<td>attr3</td>
<td>attr4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Partition #1</th>
<th>attr1</th>
<th>attr2</th>
<th>attr3</th>
<th>attr4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuple#1</td>
<td>attr1</td>
<td>attr2</td>
<td>attr3</td>
<td>attr4</td>
</tr>
<tr>
<td>Tuple#2</td>
<td>attr1</td>
<td>attr2</td>
<td>attr3</td>
<td>attr4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Partition #2</th>
<th>attr1</th>
<th>attr2</th>
<th>attr3</th>
<th>attr4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuple#3</td>
<td>attr1</td>
<td>attr2</td>
<td>attr3</td>
<td>attr4</td>
</tr>
<tr>
<td>Tuple#4</td>
<td>attr1</td>
<td>attr2</td>
<td>attr3</td>
<td>attr4</td>
</tr>
</tbody>
</table>
Parallel execution is important, which is why (almost) every major DBMS supports it.

However, it is hard to get right.
→ Coordination Overhead
→ Scheduling
→ Concurrency Issues
→ Resource Contention
MIDTERM EXAM

Who: You
What: Midterm Exam
Where: Here (McConomy Auditorium)
When: Wednesday, Oct 13th @ 3:05-4:25pm
Why: [Link](https://youtu.be/EDRsQQ6Onnw)

https://15445.courses.cs.cmu.edu/fall2021/midterm-guide.html
Exam will cover all lecture material up to and including today (Lecture #12).

Open book / open notes / calculator

What to bring:
→ CMU ID
→ Calculator
→ Pen or pencil (pencils recommended)