Lecture #14

Query Execution

Part 2

SPRING 2024  Prof. Jignesh Patel
**ADMINISTRIVIA**

**Project #2** is due **Wed Mar 12, 2024 @ 11:59pm**
**Fri Mar 15, 2024 @ 11:59pm**

**Special OH Thu Mar 14, 2024, 3-5pm GHC 5207**

**Project #3** is due **Sun April 7, 2024 @ 11:59pm**

**Mid-Term**

→ Grades have been posted to Canvas
→ See me during OH for exam viewing
→ You can post a regrade request on Gradescope
In the last class, we discussed composing operators into a plan to execute an arbitrary query.

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

We will now discuss how to execute queries using multiple workers.
WHY CARE ABOUT PARALLEL EXECUTION?

Need to use the (parallel) hardware well
→ Higher Throughput
→ Lower Latency (especially important for human-in-the-loop scenarios)

Potentially lower total cost of ownership (TCO)
→ Fewer machines means less parts / physical footprint / energy consumption.
PARALLEL / DISTRIBUTED

The database is spread across multiple resources to

→ Deal with large data sets that don’t fit on a single machine/node
→ Higher performance
→ Redundancy/Fault-tolerance

Appears as a single logical database instance to the application, regardless of physical organization.

→ SQL query for a single-resource DBMS should generate the same result on a parallel or distributed DBMS.
PARALLEL VS. DISTRIBUTED

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 costs 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 / queries.

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

Approach #1: *Process* per DBMS Worker

Approach #2: *Thread* per DBMS Worker

Approach #3: *Embedded* DBMS
PROCESS PER WORKER

Each worker is a separate OS process.
→ Relies on the OS dispatcher.
→ Use shared-memory for global data structures.
→ A process crash does not take down the entire system.
→ Examples: IBM DB2, Postgres, Oracle
**THREAD PER WORKER**

Single process with multiple worker threads.

→ DBMS (mostly) manages its own scheduling.
→ May or may not use a dispatcher thread.
→ Thread crash (may) kill the entire system.
→ Examples: MSSQL, MySQL, DB2, Oracle (2014)

*Almost every DBMS created in the last 20 years!*

---

Diagram:

- **Application**
  - Connect to **Dispatcher**
  - **Dispatcher** processes SQL Commands and directs them to **Worker Threads**
  - **Worker Threads** execute the commands

- **Dispatcher** is responsible for connecting applications to the database and managing the execution of SQL commands.
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 nearly *always* knows more than the OS.
**SQL SERVER – SQLOS**

SQLOS is a user-level OS layer that runs inside the DBMS and manages provisioned hardware resources.

→ Determines which tasks are scheduled onto which threads.

→ It also manages I/O scheduling and higher-level concepts like logical database locks.

Non-preemptive thread scheduling through instrumented DBMS code.
SQL SERVER – SQLOS

SQLOS is a user-level OS layer that runs inside the DBMS and manages provisioned hardware resources.

→ Determines which tasks are scheduled onto which threads.
→ It also manages I/O scheduling and higher-level concepts like logical database locks.

Non-preemptive thread scheduling through instrumented DBMS code.
SQLOS is a user-level OS layer that runs inside the DBMS and manages provisioned hardware resources.

→ Determines which tasks are scheduled onto which threads.
→ It also manages higher-level concepts like logical database locks.

Non-preemptive thread scheduling through instrumented DBMS code.

How Microsoft brought SQL Server to Linux

Frederic Lardinois 12:00 pm EDT  •  July 17, 2017

Back in 2016, when Microsoft announced that SQL Server would soon run on Linux, the news came as a major surprise to users and pundits alike. Over the course of the last year, Microsoft’s support for Linux (and open source in general), has come into clearer focus and the company’s mission now seems to be all about bringing its tools to wherever its users are.

The company today launched the first release candidate of SQL Server 2017, which will be the first version to run on Windows, Linux and in Docker containers. The Docker container alone has already seen more than 1 million pulls, so there can be no doubt that there is a lot of interest in this new version. And while there are plenty of new features and speed improvements in this new version, the fact that SQL Server 2017 supports Linux remains one of the most interesting aspects of this release.

Ahead of today’s announcement, I talked to Rohan Kumar, the general manager of Microsoft’s Database Systems group, to get a bit more info about the history of this project and how his team managed to bring an extremely complex piece of software like SQL Server to Linux. Kumar, who has been at Microsoft for more than 18 years, noted that his team noticed many enterprises were starting to use SQL Server for their mission-critical workloads. But at the same time, they were also working in mixed environments that included both Windows Server and Linux. For many of these businesses, not being able to run their database of choice on Linux became a friction point.

“Talking to enterprises, it became clear that doing this was necessary,” Kumar said. “We were forcing customers to use Windows as their platform of choice.” In another incarnation of Microsoft, that probably would’ve been seen as something positive, but the company’s strategy today is quite different.
SQL SERVER – SQLOS

SQLOS quantum is 4 ms, but the scheduler cannot enforce that.

DBMS developers must add explicit yield calls in various locations in the source code.

More on a different/modern way to do query/operator scheduling today (bonus material).
EMBEDDED DBMS

DBMS runs inside the same address space as the application. Application is (primarily) responsible for threads and scheduling.

The application may support outside connections.

→ Examples: BerkeleyDB, SQLite, RocksDB, LevelDB
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.

DBMS from the last 15 years use native OS threads unless they are Redis or Postgres forks.
INTER- VS. INTRA-QUERY PARALLELISM

Inter-Query: Execute multiple disparate queries simultaneously.
→ Increases throughput & reduces latency.

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

Improve overall performance by allowing multiple queries to execute simultaneously.

If queries are read-only, then this requires almost no explicit coordination between the queries.
→ Buffer pool can handle most of the sharing if necessary.

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 the organization of operators in terms of a *producer/consumer* paradigm.

There are parallel versions of every operator.
→ 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.

\[
\begin{align*}
R(\text{id, name}) & \quad \text{HT}_R \\
\vdots & \\
\text{max} & \quad \text{HT}_S \\
\quad h_1 & \\
0 & \quad 1 \\
1 & \quad 2 \\
\text{id, value, cdate} & \\
\end{align*}
\]
PARALLEL GRACE HASH JOIN

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.

→ Postgres calls this “gather”
INTRA-OPERATOR PARALLELISM

```
SELECT * FROM A
WHERE A.val > 99
```

\[ \sigma_{value>99} A \]
INTRA-OPERATOR PARALLELISM

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

```
SELECT * FROM A
WHERE A.val > 99
```

![Diagram of INTRA-OPERATOR PARALLELISM]
INTRA-OPERATOR PARALLELISM

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

SELECT * FROM A
WHERE A.val > 99

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

\[ A_1 \quad \sigma \quad A_2 \quad \sigma \quad A_3 \quad \sigma \]

Exchange

1 2 3 4 5

Pages
EXCHANGE OPERATOR

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

```
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
```
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
```
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

\[
\begin{align*}
\pi \\
\downarrow \\
\sigma \\
\downarrow \\
\sigma \\
\downarrow \\
A & \downarrow \sigma & B
\end{align*}
\]

\[
\begin{align*}
\pi \\
\downarrow \\
\sigma \\
\downarrow \\
\sigma \\
\downarrow \\
A_1 & \downarrow \sigma & A_2 & \downarrow \sigma & A_3 & \downarrow \sigma & B_1 & \downarrow \sigma & B_2 & \downarrow \sigma & B_3
\end{align*}
\]

\[
\begin{align*}
\text{Exchange} \\
\downarrow \\
\text{Build HT} \\
\downarrow \\
\pi \\
\downarrow \\
\sigma \\
\downarrow \\
A_1 \\
\downarrow 1 \\
1
\end{align*}
\]

\[
\begin{align*}
\text{Build HT} \\
\downarrow \\
\pi \\
\downarrow \\
\sigma \\
\downarrow \\
A_2 \\
\downarrow 2 \\
2
\end{align*}
\]

\[
\begin{align*}
\text{Build HT} \\
\downarrow \\
\pi \\
\downarrow \\
\sigma \\
\downarrow \\
A_3 \\
\downarrow 3 \\
3
\end{align*}
\]

\[
\begin{align*}
\text{Build HT} \\
\downarrow \\
\pi \\
\downarrow \\
\sigma \\
\downarrow \\
B_1 \\
\downarrow 1 \\
1
\end{align*}
\]

\[
\begin{align*}
\text{Build HT} \\
\downarrow \\
\pi \\
\downarrow \\
\sigma \\
\downarrow \\
B_2 \\
\downarrow 2 \\
2
\end{align*}
\]

\[
\begin{align*}
\text{Build HT} \\
\downarrow \\
\pi \\
\downarrow \\
\sigma \\
\downarrow \\
B_3 \\
\downarrow 3 \\
3
\end{align*}
\]
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
```
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.
→ More common in streaming systems (continuous queries)

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

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

for $r_1 \in \text{outer}$:
    for $r_2 \in \text{inner}$:
        emit($r_1 \Join r_2$)
**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
```
OBSERVATION

Using additional processes/threads to execute queries in parallel won’t help if the disk is always the main bottleneck.

It can sometimes make the DBMS’s performance worse if a worker is accessing different segments of the disk at the same time.
**I/O PARALLELISM**

Split the DBMS across multiple storage devices to improve disk bandwidth latency.

Many different options that have trade-offs:

→ Multiple Disks per Database
→ One Database per Disk
→ One Relation per Disk
→ Split Relation across Multiple Disks

Some DBMSs support this natively. Others require admin to configure outside of DBMS.
MULTI-DISK PARALLELISM

Data on the disk can get corrupted (bit rot), or an entire disk can fail. Get higher performance from a disk array.

Hardware-based: A hardware controller manages multiple devices, e.g. RAID.

Software-based: Use erasure codes at the file/object level. Faster and more flexible.

This is transparent to the DBMS.
MULTI-DISK PARALLELISM

Data on the disk can get corrupted (bit rot), or an entire disk can fail.

Get higher performance from a disk array.

Hardware-based: A hardware controller manages multiple devices, e.g. RAID.
MULTI-DISK PARALLELISM

Data on the disk can get corrupted (bit rot), or an entire disk can fail.

Get higher performance from a disk array.

Hardware-based: A hardware controller manages multiple devices, e.g. RAID.
MULTI-DISK PARALLELISM

Data on the disk can get corrupted (bit rot), or an entire disk can fail.

Get higher performance from a disk array.

Hardware-based: A hardware controller manages multiple devices, e.g. RAID.
MULTI-DISK PARALLELISM

Data on the disk can get corrupted (bit rot), or an entire disk can fail.

Get higher performance from a disk array.

Hardware-based: A hardware controller manages multiple devices, e.g. RAID.

Software-based: Use erasure codes at the file/object level. Faster and more flexible.

This is transparent to the DBMS.
DATABASE PARTITIONING

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 a 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.

*We will cover this further when we talk about distributed databases.*
CONCLUSION

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
Query Optimization
→ Logical vs Physical Plans
→ Search Space of Plans
→ Cost Estimation of Plans
So far, we have largely taken a **data flow** perspective of the query processing model.

The **control flow** was implicit in the processing model. We can make the control flow more explicit with a scheduler.

Query schedulers are often not discussed in database papers. We’ll look at what was done in the Quickstep (academic) project. Based on allowing frequent switches between data flow and control flow.
Clean Separation of Data Flow and Control Flow

**The “traditional” way**

```
SELECT * FROM R, S
WHERE R.b > 10
AND S.c > 100
AND R.a = S.a
```

**The Quickstep way**

```
SELECT * FROM R, S
WHERE R.b > 10
AND S.c > 100
AND R.a = S.a
```
**The Quickstep Scheduler**

**Clean Separation of Data Flow and Control Flow**

Buffer pool: Abstraction to manage “data blocks”/pages using LRU-2.

Data blocks = base data, intermediate results, QP data structures (Hash tables)

*Variable length, but multiples of a base block size. Thus, hash tables can grow (via doubling in size)*

**SELECT * FROM R, S**

**WHERE R.b > 10**

**AND S.c > 100**

**AND R.a = S.a**

---

**Pending work orders**

\[ \sigma (s_1) \sigma (r_2) \sigma (r_1) \]
The Quickstep Scheduler

Clean Separation of Data Flow and Control Flow

SELECT * FROM R, S
WHERE R.b > 10
AND S.c >100
AND R.a = S.a

Pending work orders

Probe Hash (h', s2)  Probe Hash (h', s1)  Build Hash (r')

Buffer Pool

r'  s'  h'

Advantages

+ Cleaner Abstraction
+ Dynamic Optimization
+ In-built query suspension
+ Better p9X
+ Manageability and Debug-ability
Priority scheduling = Elastic behavior
In-built Query Progress Monitoring

Bonus