Lecture 16: Parallel and Distributed Databases

Architectural Capriccio with Jephthah and his Daughter, Dirck van Dalen, 1633
Recovery Recap

• What happens during crash:
  ▪ Memory is reset
  ▪ State on disk persists

• After a crash, recovery ensures:
  ▪ **Atomicity**: partially finished transactions are rolled back
  ▪ **Durability**: committed transactions are on stable storage (disk)

• Brings database into a transaction consistent state, where committed transactions are fully reflected, and uncommitted transactions are completely undone
ARIES/Logging Recap

• NO FORCE, STEAL logging
• Use write ahead logging protocol
• Must FORCE log on COMMIT
• Periodically take (lightweight) checkpoints
• Asynchronously flush disk pages (without logging)
Parallel & Distributed DBs Overview

- Parallel DBs: how to get multiple processors/machines to execute different parts of a SQL query
  - Especially relevant for big, slow running queries

- Distributed DBs: what happens when these machines are physically disjoint / fail independently
  - Especially relevant for transaction processing
Parallel DB Goal

• SQL, but faster by running on multiple processors

• What do we mean by faster?

\[
\text{speed up} = \frac{old \ time}{new \ time} \quad \text{on same problem, with N times more hardware}
\]

\[
\text{scale up} = \frac{1x \ larger \ problem \ on \ 1x \ hardware}{Nx \ larger \ problem \ on \ Nx \ hardware}
\]

• Not necessarily the same: smaller problem may be harder to parallelize
DB Specific Metrics

• **Transaction speedup**: fixed set of txns, with 1 vs N machines
• **Batch speedup**: fixed sized DB, with 1 vs N machines

• **Transaction scaleup**: N times as many txns for N machines
• **Batch scaleup**: N times as big a query for N machines
**Speedup Goal**

- Linear?
Barriers to Linear Scaling

- Startup times
  - e.g., may take time to launch each parallel executor
- Interference
  - processors depend on some shared resource
  - E.g., input or output queue, or other data item
- Skew
  - workload not of equal size on each processor

- Almost all workloads will stop scaling at some point!

- What are some barriers in analytics and transactional workloads?
Properties of Parallelizable Workloads

• Provide linear speedup
• Usually can be decomposed into small units that can be executed independently
  ▪ "embarrassingly parallel"
• As we will see, relational model generally provides this
Parallel Architectures

• Several different ways we might parallelize databases
• Multiple cores?
• Multiple machines?
Types of Parallelism – Shared Everything

- Conventional multicore computer
- Multiple threads for execution
- Each core can access any record
- Difficult to scale beyond a few cores
- Not fault tolerant
Types of Parallelism – Shared Disk

- Several machines
- Each can access any record on disk
- Requires complex disk-oriented coherency protocols
- Relies on reliable disk array for fault tolerance
- Popularized by Oracle, not common otherwise
Types of Parallelism – Shared Nothing

- Several machines
- Data partitioned across machines
  - Each machine responsible for processing & modifying its data
- Scales very well
  - Easy to add new machines & partitions
- Fault tolerance via replication

High speed interconnect (e.g., 10GB Ethernet, Infiniband, ...)

Diagram:
- CPUs with cores
- Memory
- Disks
- High speed interconnect
Types of Parallelism – Shared Nothing on Distributed File System

- Decouples scaling of storage from scaling of processing
- Storage layer implements its own fault tolerance
- Logically data is still partitioned and operated on by different processors
- Has become common with rise of cloud computing
  - E.g., SnowFlake, MapReduce, …
## Tradeoffs Between Partitioning Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared Memory</td>
<td>Easy to build</td>
<td>Performance / scalability</td>
</tr>
<tr>
<td></td>
<td>No changes to concurrency control / recovery</td>
<td>Poor fault tolerance</td>
</tr>
<tr>
<td>Shared Disk</td>
<td>Better scalability</td>
<td>Complex cache coherency</td>
</tr>
<tr>
<td></td>
<td>Better fault tolerance</td>
<td>Poor scalability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Relies on expensive disk array</td>
</tr>
<tr>
<td>Shared Nothing (partitioned data)</td>
<td>Cost</td>
<td>New concurrency control/recovery</td>
</tr>
<tr>
<td></td>
<td>Scalability</td>
<td>New executor</td>
</tr>
<tr>
<td></td>
<td>Fault tolerance</td>
<td></td>
</tr>
</tbody>
</table>
Parallel Query Processing

• Three main ways to parallelize
  1. Run multiple queries, each on a different thread
  2. Run operators in different threads (“pipeline”)
  3. Partition data, process each partition in a different processor

- Diagram:
  - Processor 1: A1 -> filter -> sort
  - Processor 2: A2 -> filter -> sort -> merge
  - Runs on 1 of the processors
Pipelined Parallelism

• Only works when each pipeline stage is about the same speed
• Limited parallelism as most pipelines are short
• Inputs to stage $i+1$ depend on stage $i$
• If stage $i$ blocks (i.e., sorts), breaks pipeline

• As a result, partitioned parallelism is the primary way database systems scale
Partitioning Strategies

• Random / Round Robin
  ▪ Evenly distributes data (no skew)
  ▪ Requires us to repartition for joins

• Range partitioning
  ▪ Allows us to perform joins without repartitioning, when tables are partitioned on join attributes
  ▪ Subject to skew

• Hash partitioning
  ▪ Allows us to perform joins without repartitioning, when tables are partitioned on join attributes
  ▪ Only subject to skew when there are many duplicate values
Round Robin Partitioning

Advantages:
Each partition has the same number of records

Disadvantage:
No ability to push down predicates to filter out some partitions
Range Partitioning

Advantages:
Easy to push down predicates (on partitioning attribute)

Disadvantage:
Difficult to ensure equal sized partitions, particularly in the face of inserts and skewed data
Hash Partitioning

$H(T.A)$ is a hash function mapping from each record in $T$ to its partition, based on value of attribute $A$.

Advantages:

Each partition has about the same number of records, unless one value is very frequent

Possible to push down equality predicates on partitioning attribute

Disadvantages:

Can’t push down range predicates
Parallel Operations in a Partitioned DB

• SELECT
  ▪ Trivial to “push down” to each worker
  ▪ Depending on partitioning attribute, may be able to skip some partitions

• PROJECT
  ▪ Assuming all columns are on each node, nothing to be done

• JOIN
  ▪ Depending on data partitioning, may be able to process partitions individually and then merge, or may need to repartition

• AGGREGATE
  ▪ Partially aggregate data at each node, merge final result
Join Strategies

• If tables are partitioned on same attribute, just run local joins
  ▪ Also, if one table is replicated, no need to join

• Otherwise, several options:
  1. Collect all tables at one node
     o Inferior except in extreme cases, i.e., very small tables
  2. Re-partition one or both tables – “shuffle join”
     o Depending on initial partitioning
  3. Replicate (smaller) table on all nodes
Table Pre-Partitioned on Join Attribute

• Suppose we have hashed A on a, using hash function F to get F(A.a) \( \rightarrow 1..n \) (n = # machines)

• Also hash B on b using same F

• Query: SELECT * FROM A,B WHERE A.a = B.b
Repartitioning Example – “Shuffle Join”

- Suppose A pre-partitioned on a, but B needs to be repartitioned
Repartitioning Example

- Suppose A pre-partitioned on a, but B needs to be repartitioned

Generalizes to the case of repartitioning both tables
Study break!
How many bytes are sent and received from each machine?

What about when we repartition both tables?

Repartitioning Example

• Suppose A pre-partitioned on a, but B needs to be repartitioned
Repartitioning Example

- Suppose A pre-partitioned on a, but B needs to be repartitioned

Each node sends and receives \((|B|/n) / n \times (n-1)\) bytes

- Each partition is \(|B|/n\) records
- Repartitioned splits it into \(n\) new chunks, sends \(n-1\) of them
Repartitioning Both Tables

• Suppose both tables, A and B, need to be repartitioned
• Each node sends and receives
  \[
  \frac{|A|}{n} \frac{n}{n} \times (n-1) + \frac{|B|}{n} \frac{n}{n} \times (n-1) \text{ bytes}
  \]
Replication Example

• Suppose we replicate B to all nodes

$$\frac{|B|}{n} \times (n-1) \text{ bytes sent & received by each node}$$
Replication vs Repartitioning

- Replication requires each node to send smaller table to all other nodes
  - \((|T| / n) \times (n-1)\) bytes sent by each node
  - vs \(((|T| / n) / n) \times (n-1)\) to repartition one table

- When would replication be preferred over repartitioning for joins?
  - If size of smaller table < data sent to repartition one or both tables
  - Should also account for cost of join: will be higher with replicated table

- Example: \(|B| = 1\) MB, \(|A| = 100\) MB, \(n=3\)
- Need to repartition A (B distributed on join attr)
  - Data to repartition A is \(|A|/3 \times 3 \times 2 = 22.2\) MB per node
    - Join 0.33 MB to 33 MB
  - Data to broadcast B is \(|B| = 1/3 \times 2 = .66\) MB
    - Join 1 MB to 33 MB
Study Break 2

- Suppose we have two tables A and B, partitioned across 3 nodes
- |A| = 9 MB
- |B| = 90 MB
- Join is $A.a = B.b$
  - B is hash partitioned on b, A is not partitioned on a
- How much data does each node send if we:
  1. Repartition A  
     \[
     \frac{9}{3} \times 2 = 6 \text{ MB}
     \]
  2. Replicate A  
     \[
     \frac{9}{3} \times 2 = 6 \text{ MB}
     \]
Additional Options for Joins

• Pre-replicated small tables
  - If space permits, can be a good option

• “Semi-join”
  - send list of join attribute values in each partition of B to A,
  - then send list of matching tuples from A to B,
  - then compute join at B

• Good for selective joins of wide tables
  - Pre-filters A with join values that actually occur in B, rather than sending all of B
Semi-join Example

Node 1

Node 2

Total cost:

Each node sends & receives

\[
\frac{\text{ljoin col}}{n} \div n \times (n-1) \\
+ \\
\frac{f \times |A|}{n} \div n \times (n-1)
\]

Where \( f \) is join selectivity
Aggregation

In general, each node will have data for the same groups

So merge will need to combine groups, e.g.:

- MAX (MAX1, MAX2)
- SUM (SUM1, SUM2)

What about average?
  Maintain SUMs and COUNTs, combine in merge step
Generalized Parallel Aggregates

• Express aggregates as 3 functions:
  - **INIT** – create partial aggregate value
  - **MERGE** – combine 2 partial aggregates
  - **FINAL** – compute final aggregate

• E.g., AVG:
  - INIT(tuple) → (SUM=tuple.value, COUNT=1)
  - MERGE (a1, a2) → (SUM=a1.SUM + a2.SUM, COUNT=a1.count+a2.count)
  - FINAL(a) → a.SUM/a.COUNT
What does MERGE do?

• For aggregate queries, receives partial aggregates from each processor, MERGEs and FINALizes them

• For non-aggregates, just UNIONs results
DB Parallel Processing vs General Parallelism

• Shared nothing partitioned parallelism is the dominant approach
• Hooray for the relational model!
  ▪ Apps don't change when you parallelize system (physical data independence!).
  ▪ Can tune, scale system without changing applications!
  ▪ Can partition records arbitrarily, w/o synchronization
• Essentially no synchronization except setup & teardown
  ▪ No barriers, cache coherence, etc.
  ▪ DB transactions work fine in parallel
• Data updated in place, with 2-phase locking transactions
  ▪ Replicas managed only at EOT via 2-phase commit (next lecture)
  ▪ Coarser grain, higher overhead than cache coherency on processors
• Bandwidth much more important than latency (in analytics at least)
  ▪ Often pump 1-1/n % of a table through the network
    ○ Aggregate net BW should match aggregate disk BW