CS122D: Beyond SQL Data Management
— Lecture #4 —

Mike Carey
UC Irvine
mjcarey@ics.uci.edu
Announcements

• Be sure to keep watching the wiki and Piazza pages
  • http://www.ics.uci.edu/~cs122d
  • http://piazza.com/uci/spring2021/cs122d/home
  • Also, don’t forget to do the readings!

• Hopefully your RDBMS cobwebs have been cleared?
  • If not, you should go back over your CS122A notes and perhaps re-watch some of your instructor’s lectures...!

• HW #1 is now available (see the course wiki page)
  • ShopALot.com will be the theme for the quarter
  • Start by getting PostgreSQL up and running!

• Today: Parallel (Relational) Database Systems
What If My Problem Size Grows?

“Scaling Up” vs. “Scaling Out”

Parallelism to the rescue...!
Just How “Big” Is “Big Data”?


Bigger than the system’s aggregate memory!
Parallel Performance Objectives

![Graph showing linear speed-up and linear scale-up](image)

(a) Linear speed-up
(b) Linear scale-up

Fig. 8.1: Extensibility Metrics

Performance Objectives (cont.)

• Linear speed-up: Adding more resources results in a proportionally faster execution time for a fixed amount of data

• Batch scale-up: Adding more resources can handle a proportionally bigger problem in the same amount of time

• Transaction scale-up: Adding more resources allows the handling of proportionally more concurrent requests
Alternative Parallel DB Architectures

**Shared Everything (SE)**

- Processor
- Interconnection Network
- Global Shared Memory
- Disk

**Shared Disk (SD)**

- Processor
- Memory
- Interconnection Network
- Disk

**Shared Nothing (SN)**

- Processor
- Memory
- Disk

Notable research systems
- Gamma, Grace, Bubba, …

Many commercial examples
- Teradata
- IBM DB2 Parallel
- Microsoft PDW
- Vertica
- …

Over 30 years of R&D!

But...?
Teradata: The Big Gorilla (😊)

A 1TB Teradata system on its way to K-Mart!
(Thanks to Todd Walter, a retired Teradata CTO)
Two Forms of Intra-Query Parallelism

- **Partitioned parallelism**: Many machines each running the same step on a different subset (partition) of the data
- **Pipelined parallelism**: Many machines each working on a different step of a multi-step process

• For many years, relational DBMSs were the most (only?) successful/commercial application of parallelism
  • Early years: Teradata, Tandem, Thinking Machines, KSR, ...
  • Now every major RDBMS vendor has some || offering
    • (Now we have parallel Web search and ML too)

• Reasons for parallel RDBMS success include:
  • Set-oriented processing (partitioned parallelism)
  • Natural pipelining (plan trees)
  • Applicability of inexpensive hardware to the problem
  • Users/application developers don’t have to think about ||

Notes:
• Storage manager and runtime executor per node
• Upper layers orchestrate them
  • One way in/out: via the SQL door

SQL’s programming model is scale-independent!

DBMS: The || Success Story
Parallelism in RDBMS Terms

- **Intra-operator** parallelism:
  - Get all machines working together to concurrently compute a given operation (e.g., a scan, sort, or join operator)

- **Inter-operator** parallelism:
  - Get different operations (query operators) running concurrently on different machines within the system

- **Inter-query** parallelism:
  - Admit multiple queries into the system at once and allow them to run concurrently (possibly on different machines)
Automatic Data Partitioning

Partitioning a SQL table’s data

Range

Good for equijoins, exact-match queries, and range queries

Hash

Good for equijoins and exact match queries

Round Robin

Good to spread load (e.g., full scans)

Notes:
- Shared disk (SD) and shared memory (SM) less sensitive to data partitioning
- Shared nothing (SN) benefits from "good" partitioning (to avoid skew)
Parallel Scan/Select Queries

• Scan in parallel and merge (a.k.a. union all)

• Note: Selections may not require all sites for range or hash partitioning, but always do for round robin

• Secondary indexes can be constructed on each partition
  • Indexes useful for local lookups, as expected
  • Per-partition index lookups will occur in parallel
  • However, what about unique indexes...?
    (May not always want primary key partitioning...)

(M.M. Carey, Spring 2021: CS122D)
Secondary Indexes

- **Secondary indexes become “interesting” in the face of partitioning...**

- **Local secondary indexes**: Partition secondary indexes by the base table’s primary key
  - Inserts are then local (unless unique index, if allowed)
  - Lookups, however, then go to **ALL** indexes
  - Query-oriented systems do this (including AsterixDB 😊)

- **Global secondary indexes**: Partition secondary indexes by the secondary key range
  - Inserts must then hit 2 nodes (base table + index)
  - Ditto for index lookups (index + base table)
  - Uniqueness is easy to enforce
  - Performance for short queries/updates is very good

- Teradata’s index partitioning solution (a hybrid):
  - Partition non-unique indices by base table key
  - Partition unique indices by secondary key
**Local Joins: Hybrid Hash Join**

**SELECT** *
**FROM** Item i, Manufacturer m
**WHERE** i.mid = m.id;

**Rules:**
1) In each Join, Probe is blocked by Build
2) Hash table (on *item.mid*) should be in memory
3) One output produced (concurrent with Probe)
4) Spills to disk in case of insufficient memory

(Thanks to UCI Ph.D. student Shiva Jahangiri!)
Parallel Hash Join

Hash-distribute the rows from each table (R and S) to the join processing nodes based on join key (jk) values (if/as needed)
Parallel Hash Join (cont.)

\[ R \Join S = \text{union} (R_{ij} \Join S_{ij}) \]

\(T_{ij} \equiv \text{Local partition j of global partition i of input table T}\)
Multiple Hash Join Queries

- **Left Deep Tree**
  - Low parallelism (Sequential)
  - Low memory usage (2 relations at a time)

- **Bushy Tree**
  - Independent parallelism
  - Plan-dependent memory usage

- **Right Deep Tree**
  - High parallelism
  - High memory usage

(Thanks to UCI Ph.D. student Shiva Jahangiri!)
Some Parallel Join Methods

• **Parallel hash join** (we just saw this one)
  - Each dataset (R, S) is **repartitioned** (if **needed**) by hashing their records on the join key
  - A local hash join is performed between each of the resulting partition pairs

• **Nested loops index join**
  - The “outer” dataset R is **replicated** to **all** partitions of the “inner” **indexed** dataset S
  - Arriving R tuples are used (as they arrive – pipelining!) to probe the S index for matches

• **Broadcast join**
  - The build dataset R is **broadcast** (replicated) to **all** partitions of the probe dataset and arriving tuples are treated as the build tuples in a local hash join
  - The probe dataset S is then used to probe the build dataset
Parallel Grouped Aggregation

• Hash-based partitioned parallelism also works for grouping!
  • In this case, hash partitioning is based on the grouping attribute(s)

```sql
SELECT br.brewery_id, COUNT(*) AS num_beers
FROM beers br
GROUP BY br.brewery_id;
```
Parallel Sorting w/Global Merge

- Partitioned parallelism also works for sorting!
- The O(N \lg N) step is parallelized, followed by a global merge

```sql
SELECT br.brewery_id, br.name
FROM beers br
ORDER BY br.name;
```
Range-Based Parallel Sorting

• Also possible to eliminate a global merge
  • Scan (sample), range partition, and sort ranges

```
SELECT br.brewery_id, br. name
FROM beers br
ORDER BY br.name;
```
Parallel RDBMS Summary

• Pipeline-ism occurs naturally in relational query processing
  • Both pipelined and partitioned pipeline-ism arise
  • Intra-operator, inter-operator, and inter-query pipeline-ism

• Shared Nothing is the dominant choice for scaling out
  • Shared Disk is used too (e.g., Oracle), but is less common
    • The cloud is bringing SD back, however...!
  • Shared Memory approach is actually easiest but costly
    • And there are limits to scaling up – and then what?
  • Shared Nothing is cheap, scales well, supports incremental growth and has nice fault isolation – but harder to implement

• Expect a “deja vu” sense for the platforms ahead...
Parallel RDBMS Summary (cont.)

• Data layout choices are important!
  • Hash, range, and round-robin partitioning options
  • Local vs. global secondary index layout

• Most DB operations can be done partition-parallel
  • Select, project, join, ...
  • Sorting, grouping/aggregation, ...

• Complex queries are supportable
  • Involve a mix of pipelining and blocking steps
  • Note: Query optimization can be challenging!

• Transactions and consistency are “interesting” for SN
  • Need two-phase commit if transactions span nodes
So Where Are We...?

• We’re done setting the “Beyond SQL” stage
  • Looked at the history of “Beyond SQL” attempts
  • Refreshed our knowledge of SQL (incl. HW #1)
  • Refreshed our knowledge of data modeling
  • Learned how SQL was parallelized in the ‘80’s & 90’s

• Let our “Beyond SQL” tour begin...!
  • First stop (next Monday): Key-value stores
  • Reading reminder:

<table>
<thead>
<tr>
<th>Date</th>
<th>Topic</th>
<th>Relevant Material</th>
</tr>
</thead>
<tbody>
<tr>
<td>M 3/29</td>
<td>Post-relational “escape attempts”</td>
<td>Ch. 1-2 NoSQL Distilled, Introducing JSON website</td>
</tr>
<tr>
<td>W 3/31</td>
<td>Relational databases and SQL (and “beyond”)</td>
<td>SQL chapters of any DB textbook, Advanced Aggregation excerpt</td>
</tr>
<tr>
<td>M 4/05</td>
<td>Logical DB design and E-R modeling</td>
<td>E-R chapter of any DB textbook</td>
</tr>
<tr>
<td>W 4/07</td>
<td>Scaling RDBMSs through parallelism</td>
<td>Parallel RDBMS paper</td>
</tr>
<tr>
<td>M 4/12</td>
<td>Key-value stores: Architecture &amp; consistency</td>
<td>Baseball paper, Ch. 4-6, 8 NoSQL Distilled, Abadi paper</td>
</tr>
</tbody>
</table>
Questions...?