Principles of Data Management

Lecture #15
(Parallel DB Systems)

Instructor: Mike Carey
mjcarey@ics.uci.edu

Today’s Notable News

- Project dates
  - Part 3 (B+ trees): today (3/3)
  - Part 4 (Query processing): 2 weeks out (3/17)
- Upcoming lectures
  - This week: Parallel databases & MapReduce
  - Following week: Wrap-up & review, endterm (!)
- Big Data Projects class
  - Prerequisite directions on Piazza course page
Why Parallel Access To Data?

At 10 MB/s
1.2 days to scan

1,000 x parallel
1.5 minute to scan.

Parallelism:
Divide a big problem into many smaller ones to be solved in parallel!

Parallel DBMS: Intro

- Parallelism is natural to DBMS processing
  - Pipelined parallelism: many machines each doing one step in a multi-step process.
  - Partitioned parallelism: many machines doing the same thing to different pieces of data.
  - Both are natural in DBMS!

Pipeline

Partition

outputs split N ways, inputs merge M ways
**DBMS: The Success Story**

- For a long time, DBMSs were the most (only?) successful/commercial application of parallelism.
  - Teradata, Tandem vs. Thinking Machines, KSR.
  - Every major DBMS vendor has some server.
  - (Of course we also have Web search engines now. 😊)

- Reasons for success:
  - Set-oriented processing (partitioned -ism).
  - Natural pipelining (relational operators/trees).
  - Inexpensive hardware can do the trick!
  - Users/app-programmers don’t need to think in -ism.

---

**Some Terminology**

- **Speed-Up**
  - Adding more resources results in proportionally less running time for a fixed amount of data.

- **Scale-Up**
  - If resources are increased in proportion to an increase in data/problem size, the overall time should remain constant.
Architecture Issue: Shared What?

Shared Memory (SMP)
- Easy to program
- Expensive to build
- Difficult to scale
- Sequent, SGI, Sun

Shared Disk
- Hard to program
- Cheap to build
- Easy to scale
- VMSccluster, Sysplex

Shared Nothing (network)
- (Use affinity routing to approximate SN-like non-contention)
- Tandem, Teradata, SP2

What Systems Work This Way

(as of 9/1995)

Shared Nothing
- Teradata: 400 nodes
- Tandem: 110 nodes
- IBM / SP2 / DB2: 128 nodes
- Informix/SP2: 48 nodes
- ATT & Sybase: ? nodes

Shared Disk
- Oracle: 170 nodes
- DEC Rdb: 24 nodes

Shared Memory
- Informix: 9 nodes
- RedBrick: ? nodes
Different Types of DBMS | -ism

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

- Inter-operator parallelism
  - each operator may run concurrently on a different site (exploits pipelining)

- Inter-query parallelism
  - Completely different queries run on different sites

- We’ll focus mainly on intra-operator -ism

Automatic Data Partitioning

Partitioning a table:

Range
- Good for equijoins, exact-match queries, and range queries
- Shared disk, shared memory less sensitive to partitioning.

Hash
- Good for equijoins, exact match queries

Round Robin
- Good to spread load
- Shared nothing benefits from "good" partitioning.
Parallel Scans/Selects

- Scan in parallel and merge (a.k.a. union all).
- Selection may not require all sites for range or hash partitioning, but always does for RR.
- Indexes can be constructed on each partition.
  - Indexes useful for local accesses, as expected.
  - Per-partition index lookups will occur in parallel.
  - However, what about unique indexes...?
  - (May not always want primary key partitioning!)

Secondary Indexes

- Secondary indexes become somewhat troublesome in the face of partitioning...
- Can partition them via base table key.
  - Inserts are then local (unless unique??).
  - Lookups, however, go to ALL indexes.
  - Many systems do this (including AsterixDB).
- Can partition by secondary key ranges.
  - Inserts then hit 2 nodes (base, index).
  - Ditto for index lookups (index, base).
  - Uniqueness is easy, however, and also performance for small queries is good.
- Teradata’s index partitioning solution:
  - Partition non-unique indices by base table key.
  - Partition unique indices by secondary key.
**Grace Hash Join**

- In Phase 1 in the **parallel** case, partitions’ data will be (re)distributed (as needed) to different sites:
  - A good hash function *automatically* distributes work evenly! (Different hash function for partitioning, BTW.)
- Do Phase 2 (actual joining) at each site in parallel.
- Almost always the winner for (ad hoc) equi-joins.

**Dataflow Network for || Joins**

- Use of split/merge makes it easier to build parallel versions of sequential join code.
**Broadcast Join**

- One problem with Grace Hash Join is that it scans both relations – always!
  - Talked about this in the single-system case – there are times when Index NL Join is better! (Q: When?)
  - Need to adapt Index NL Join to the parallel world!
- Solution is to “broadcast”, i.e., replicate, the outer relation.
  - Send the tuples of the outer relation to all sites with a partition of the inner relation.
  - In parallel, join the outer relation with each partition of the inner relation using Index NJ Join.

**Parallel Sorting**

- Basic idea:
  - Scan in parallel, range-partition as you go.
  - As tuples arrive, perform “local” sorting.
  - Resulting data is sorted and range-partitioned (i.e., spread across system in known way).
  - Just one potential problem: **Skew!**
  - Solution: “sample” the data at the outset to determine good range partitioning points.
**Parallel Aggregation**

- For each aggregate function, need a decomposition:
  - $\text{count}(S) = \sum \text{count}(s(i))$, ditto for $\text{sum}()$
  - $\text{avg}(S) = (\sum \text{sum}(s(i))) / \sum \text{count}(s(i))$
  - and so on...

- For groups:
  - Sub-aggregate groups close to the source.
  - Pass each sub-aggregate to its group’s partition site.
  - Remember this for next time (≈ combiners in MapReduce).

**Complex Parallel Query Plans**

- Complex Queries: Inter-Operator parallelism
  - Pipelining between operators:
    - note that sort or phase 1 of hash-join block the pipeline!
  - Bushy Trees
Observations

- It’s relatively easy to build a fast parallel query executor.
  - S.M.O.P., well understood today.
- It’s hard to write a robust and world-class parallel query optimizer.
  - There are many tricks.
  - One quickly hits the complexity barrier.
  - Many resources to consider simultaneously (CPU, disk, memory, network).

Parallel Query Optimization

- Common approach: 2 phases
  - Pick best sequential plan (System R algorithm)
  - Then pick degree of parallelism based on current system parameters.
- “Bind” operators to processors
  - Take query tree, “decorate” it with site assignments as in previous picture.
What’s Wrong With That?

- Best serial plan != Best || plan! Why?
- Trivial counter-example:
  - Table partitioned with local secondary index at two nodes
  - Range query: all of node 1 and 1% of node 2.
  - Node 1 should do a scan of its partition.
  - Node 2 should use secondary index.
- Ex: SELECT *
  FROM telephone_book
  WHERE name < “NoGood”;

Parallel DBMS Summary

- ||-ism natural to query processing!
  - Both pipeline and partition ||-ism are natural.
- Shared-Nothing vs. Shared-Memory
  - Shared-disk too, but less “standard”. (But the cloud may bring this architecture back?)
  - Shared-memory easy, costly. Limited scaleup.
  - Shared-nothing cheap, scales well, harder to implement, but nice fault isolation.
- Intra-op, Inter-op, and Inter-query ||-ism are all possible.
Data layout choices are important!
- In practice, will not N-way partition every table.
- And, N will need to stop being “all nodes”.

Most DB operations can be done partition- | | .
- Select, sort-merge join, hash-join.
- Sorting, aggregation, ...

Complex plans.
- Allow for pipeline- | | ism, but sorts and hashes necessarily block the pipeline.
- Partitioned | | -ism achieved via bushy trees.

Hardest part of the equation: optimization.
- 2-phase optimization simplest, but can be ineffective.
- More complex schemes still at the research stage.

We haven’t said anything about xacts, logging, etc.
- Easy in shared-memory architecture.
- Takes a bit more care in shared-nothing architecture.
- One reason local secondary indexes can be attractive.
Parallel query optimization.
Physical database design (incl. placement).
Mixing batch & OLTP activities.
  • Resource management and concurrency challenges for DSS queries versus OLTP queries/updates.
  • Also online, incremental, parallel, and recoverable utilities for load, dump, and various DB reorg ops.
Application program parallelism.
  • MapReduce, anyone...?
  • (Some new-ish companies looked at this some, e.g., GreenPlum, AsterData, …) – all later acquired.