CS 133: Databases

Fall 2019
Lec 24 – 12/05
Parallel and Distributed DBMSs
Prof. Beth Trushkowsky

Warm-up Exercise
(See exercise sheet. You can start before class.)

Performance, availability, more storage space to fit the data

Goals for Today

• Discuss distributing data and workload for increased performance in a DBMS

• Reason about DBMS concepts in parallel and distributed setting
  – How to achieve distributed ACID transactions
  – Data consistency across copies

• Understand why some of the disadvantages of distributed databases have led to some relaxation of consistency

Some Parallelism Terminology

• Throughput
  – Amount of work done per unit time

• Latency (response time)
  – Time to complete one unit work

• Speed-Up
  – More resources means less time for a given unit of work
  ➔ do more units of work in same time

• Scale-Up
  – If resources increased in proportion to increase in units of work, time per unit is constant.

Resource contention impacts scale-up/speed-up

Resource contention impacts scale-up/speed-up
Parallel DBMS Architectures

- Multiple processors (CPUs) can do work in parallel
  - How do they communicate about what work to do?
  - Three main parallel DB architectures:

  - **Shared Memory (aka shared everything)**
  - **Shared Disk**
  - **Shared Nothing**

Data Partitioning

- **Partitioning (aka sharding aka horizontally fragmenting) a table:**
  - **Range**
  - **Hash**
  - **Round Robin**

Different Types of DBMS Parallelism

- **Inter-query** parallelism: different queries run on different nodes
- **Intra-operator “partitioned” parallelism**
  - Multiple machines working together to execute an operator (e.g., scan, sort, join)
  - Machines work on disjoint partition of the data
- Parallelizing a relational operator: merge and split
  - Merge streams of output to serve as input to an operator
  - Split output of operator to be processed in parallel

Different Types of DBMS Parallelism

- **Inter-operator “pipelined” parallelism**
  - Each relational operator may run concurrently on a different machine
  - Output of first operator consumed on-the-fly as input to second operator

Could be limited by blocking operators such as sort or aggregation
Exercise 2

- (a) inter-query
- (b) intra-operator “partitioned” parallelism

Distributed DBMS (Shared Nothing)

- Data is stored at several sites (geo-distributed), each managed by a DBMS that can run independently
  - Distributed Data Independence:
    Users should not have to know where data is located
    — Note: catalog needs to keep track of where data is
  - Distributed Transaction Atomicity:
    Users should be able to write Xacts accessing multiple sites just like local Xacts

Distributed Query Processing: Joins

<table>
<thead>
<tr>
<th>Site</th>
<th>Table</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>LONDON</td>
<td>Sailors</td>
<td>500</td>
</tr>
<tr>
<td></td>
<td>Reserves</td>
<td>1000</td>
</tr>
</tbody>
</table>

- **Approach 1 -- Fetch as Needed**: Page NLJ, Sailors as outer (query submitted at London):
  - **D** is cost to read/write page; **S** is cost to ship page
  - **Cost**: 500 D + 500 * 1000 (D+S) = 500,500 D + 500,000 S
  - If query not submitted at London, must add cost of shipping result to query site

- **Approach 2 -- Ship to One Site**: Ship Reserves to London
  - **Cost**: 1000 (D+S) + 500 D + 500*1000 D = 501,500 D + 1000 S

Could also consider other single-site join methods

Semijoin

- **Why ship all of Reserves to London?**
  - Some of these tuples may not even end up being joined, so wasted communication cost
  - **Idea**: only ship the Reserves tuples that will match Sailors tuples

<table>
<thead>
<tr>
<th>Site</th>
<th>Table</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>LONDON</td>
<td>Sailors</td>
<td>500</td>
</tr>
<tr>
<td></td>
<td>Reserves</td>
<td>1000</td>
</tr>
</tbody>
</table>

- **Bottom line**: Tradeoff the cost of computing and shipping projection for cost of shipping full Reserves relation.
- **Note**: Especially useful if there is a selection on Sailors, and then join selectivity is also high
An aside: Bloom Filter

- A **bloom filter** is a bit vector used to quickly determine whether an element belongs to a set
- Hash elements to one of $k$ buckets

Refinement: Bloomjoin

- Idea: rather than shipping the join column, ship a more compact data structure that captures (almost) the same info
  - Bloom filter bit vector
  - Bit-vector cheaper to ship, almost as effective (false positives possible)
- Hash Sailors.sid values into range $[0,k-1]$
  - If tuple hashes to slot $i$, set bit $i$ to 1
- Hash Reserves tuples into same range $[0,k-1]$
  - Discard tuples that hash to 0 in Sailors bit vector

Query Optimization

- New considerations for cost-based approach
  - Communication costs
  - New distributed join methods
- Also optimizing for query response time?
  - Might want to consider a larger space of plans

Exercise 3

- Concerns: correctness, deadlock, performance
- Some options:
  - All lock requests go through a central location
  - Lock requests distributed; request is sent to the site that holds the data that is desired
Distributed Transactions

- With data at distributed sites, a transaction may operate on data at multiple sites
  - xact broken down into sub-xacts that execute at each site

- Example: read and update inventory at four sites with horizontal partitions of the data
  - T: `R(A), R(B), R(C), R(D), W(A), W(B), W(C), W(D)`, commit

- How do we guarantee atomicity??
  - Need a commit protocol (type of consensus protocol)
    - A log is maintained at each site, as in a centralized DBMS, and commit protocol actions are additionally logged

Two-Phase Commit (2PC)

- Site at which xact originates is coordinator; other sites at which it executes are subordinates

- Suppose xact wants to commit (normal execution):

  1. PREPARE!
  2. Log: no or ready
  3. All YES? Log: commit
     - Else Log: abort
  4. Subordinates force-write abort/commit log rec based on msg they get, then send ack msg to coordinator.
  5. Coordinator writes end log rec after getting all acks.

2PC: Steps

- When a Xact wants to commit:
  1. Coordinator sends prepare msg to each subordinate.
  2. Subordinate force-writes a no or ready log record and then sends a no or yes msg to coordinator.

  3. If coordinator gets unanimous yes votes, force-writes a commit log record and sends commit msg to all subs.
     - Else, force-writes abort log rec, and sends abort msg.
  4. Subordinates force-write abort/commit log rec based on msg they get, then send ack msg to coordinator.
  5. Coordinator writes end log rec after getting all acks.

Site and Link Failures 😞

- If coordinator detects a subordinate failed, e.g., after timeout

  Exercise 4: when the failed subordinate site wakes up, can it tell if a global transaction committed or aborted?

- If coordinator for Xact T fails, subordinates who have voted yes cannot decide whether to commit or abort T until coordinator recovers
  - Xact T is blocked

A B C D
Exercise 4

• Site that wakes up should interpret its log
  – (a) T committed! it should REDO actions for T
  – (b) T aborted! it should UNDO actions for T
  – (c) Can’t tell... Needs to consult coordinator about what coordinator decided
  – (d) T aborted! it should write abort, and UNDO
    • Since it never even wrote ready, coordinator couldn’t possibly have decided to commit (since it would have waited for the site to say yes)

Data Replication

• Replication: keep copies of data at different sites

• Benefits
  – Gives increased availability
  – Faster query evaluation

• Flavors
  – Synchronous (eager) vs. Asynchronous (lazy)
    • Vary in how current copies are (i.e., how consistent they are)

• Can be used in addition to data partitioning
  – Full replication: copy of all data at every site (vs. partial)

Updating Distributed Data

• Synchronous (Eager) Replication: set of copies of a modified relation must be updated **before the modifying xact commits**
  – Exclusive locks on all the copies that are modified
  – Users/apps do not need to know data location(s)

• Asynchronous (Lazy) Replication: Copies of a modified relation are only periodically updated
  – Different copies may get out of sync in the meantime
  – Users/applications must be aware of data location(s)

Synchronous Replication: Majority

• Majority technique can guarantee data consistency:
  – Xact must **write a majority** of copies to modify an object
  – Each copy has version number for object
  – Xact must **read enough copies** to be sure of seeing at least one most recent copy

• Example: 6 copies of data

Impact on performance of READ queries??

Could use Read-one-write-all (ROWA) policy instead

Written: 4 nodes
Read: 3 nodes

Not necessarily all copies!