Lecture 5 - Wisconsin Papers

Setup

Three papers for today

- The Gamma Database Machine Project - De Witt and Co, 1990
- Parallel Database Systems - De Witt and Jim Gray, 1992
- Transaction Management in R* - Mohan, Lindsay and co, 1986

Goals for today:

- Introduce parallel (first two papers) and distributed databases (last paper)
- Discuss the power of parallelism in data processing - and challenges

Some history

- Soon after the various teams "solved" single-node data processing (i.e., the success of R and Ingres), they moved on to multi-node data processing
- Bifurcated into two major types of efforts
  - Distributed databases
    - essentially multiple "parties/sites" trying to perform data processing across their sites
    - key driver: autonomy and availability
    - key examples: R*, Distributed INGRES, SDD-1, MULTIBASE
  - Parallel databases
    - essentially databases being actively divided up into smaller portions to take advantage of parallelism or reliability or both
    - key examples: Gamma, Bubba, Teradata, Tandem NonStop SQL
  - Some efforts touched both
    - Mariposa/Cohera (Berkeley)

Distributed Databases

- Also known as a "federated database", with the data being distributed also known as data federation
- Two well-known efforts: R* from IBM and Distributed INGRES from Berkeley - also SDD-1, MULTIBASE
  - Earlier efforts: Mariposa from Stonebraker, then a startup Cohera
    - Didn't succeed - the challenge was not in building a faster backend backend but in "data integration" - a gnarly human-centric problem
      - ETL to data warehouses with unified schema sufficed
- Modern efforts:
  - Presto/Ahana (built on Presto - acquired by IBM), Trino/Starburst Data (built on Trino), Dremio
  - Google BQ Omni (querying over many clouds), AWS Athena

Setup:

- Independent, autonomous databases connected via a wide-area (slow) network
  - From a user standpoint, goal is to have location transparency - they don't know where the data is coming from, can query any "site"
  - Site autonomy - no dependencies on central controller/coordinator. No:

    - Central catalog
        - though SDD and Ingres did have a central catalog
    - Central scheduler
    - Central deadlock detector/fixer
    - Central access control (done per site)
- Tables stay entirely in one site
    - So no "sharing" - what the parallel database papers call *declustering/partitioning*
- A big challenge is naming
    - Bruce@San_Jose . T@San_Jose

Key Contributions

- 2 Phase Commit with WAL and recovery, including Presumed Commit/Abort optimizations. (Mohan paper)
    - 2PC Intuition:
        - Prepare phase:
            - Coordinator sends all participants a **PREPARE**
            - Each site:
                - forces log to disk
                - replies **YES**/**NO**
        - Commit phase:
            - If all **YES**. then coordinator decides to commit, else abort
            - Coordinator:
                - Writes decision to storage
                - Sends **Commit/Abort** to participants
            - Participants
                - Execute decision
                - Release locks
                - Acknowledge with **ACK
        - WAL allows for replaying a site's decision
    - Messages $4(N-1)$ -> Prepare, Response, Commit, Ack
    - Presumed commit / abort allows for reducing communication
        - to $3(N-1)$
        - Basically can drop the ACK if it is "implied"
        - No need to use extensive logging for both abort or commit; can skip steps depending on which is more common
- Distributed deadlock detection
    - Waits-for graphs may span machines
    - Periodic exchange of information to detect cycles
    - One option: ship entire graph per site to centralized detector
        - Or: ship paths
- Practical use of semi-joins and Bloom joins (bloom filter-based semijoin) to minimize network bottleneck for joining across a network
    - Joining `R(A)` at site 1 with `S(A)` at site 2 over WAN is expensive
        - Naive: Ship all of `R` or `S` across network
        - Semi-join:
            - Send projection of join keys
            - Filter remote table
            - Send back only matching tuples
        - Bloom Join:

        - Further optimization - compress into Bloom filter
        - Q: What is a bloom filter?
            - set membership data structure that admits false positives but no false negatives
- Query optimization that takes NW cost into account (SIGMOD 86)
    - Generalizes join optimization to also consider location of join result as another "interesting property"
    - When joining two relations, there are various options, suppose A@M (outer) and B@N (inner) (this is from the R* Architecture paper)
        - Send B to M, join at M -> result at M
        - Send one tuple of A at a time to N, join at N -> result at N
        - Send join attributes of A to N (via bloom/semi join), find selected tuples, send it back to M -> result at M
        - Send A and B to another site -> result at other site

Parallel Databases

- Let's talk about the Gamma paper
    - 5 year into the project (1984-1989)
    - Previous attempt: DIRECT
        - Unsuccessful (from the paper):
            - too much emphasis on shared memory (rather than shared nothing)
            - and centralized control
    - Paper is very clear about it being mostly an engineering exercise
        - but impressive as a PoC, especially in academia
- Key ideas
    - Shared nothing - network of commodity machines
        - Much like MapReduce of past, or even today's data warehouses (Redshift, Snowflake, etc. etc.)
    - Extensive use of hash-based parallel implementations
    - Horizontal partitioning/declustering
- Other efforts:
    - Bubba (Wisconsin)
    - Teradata
    - Tandem Non-Stop SQL (Jim Gray moved here)
    - All use horizontal partitioning and shared nothing
    - Teradata / Tandem use hashing for partitioning, but conventional SMJ algorithms per site

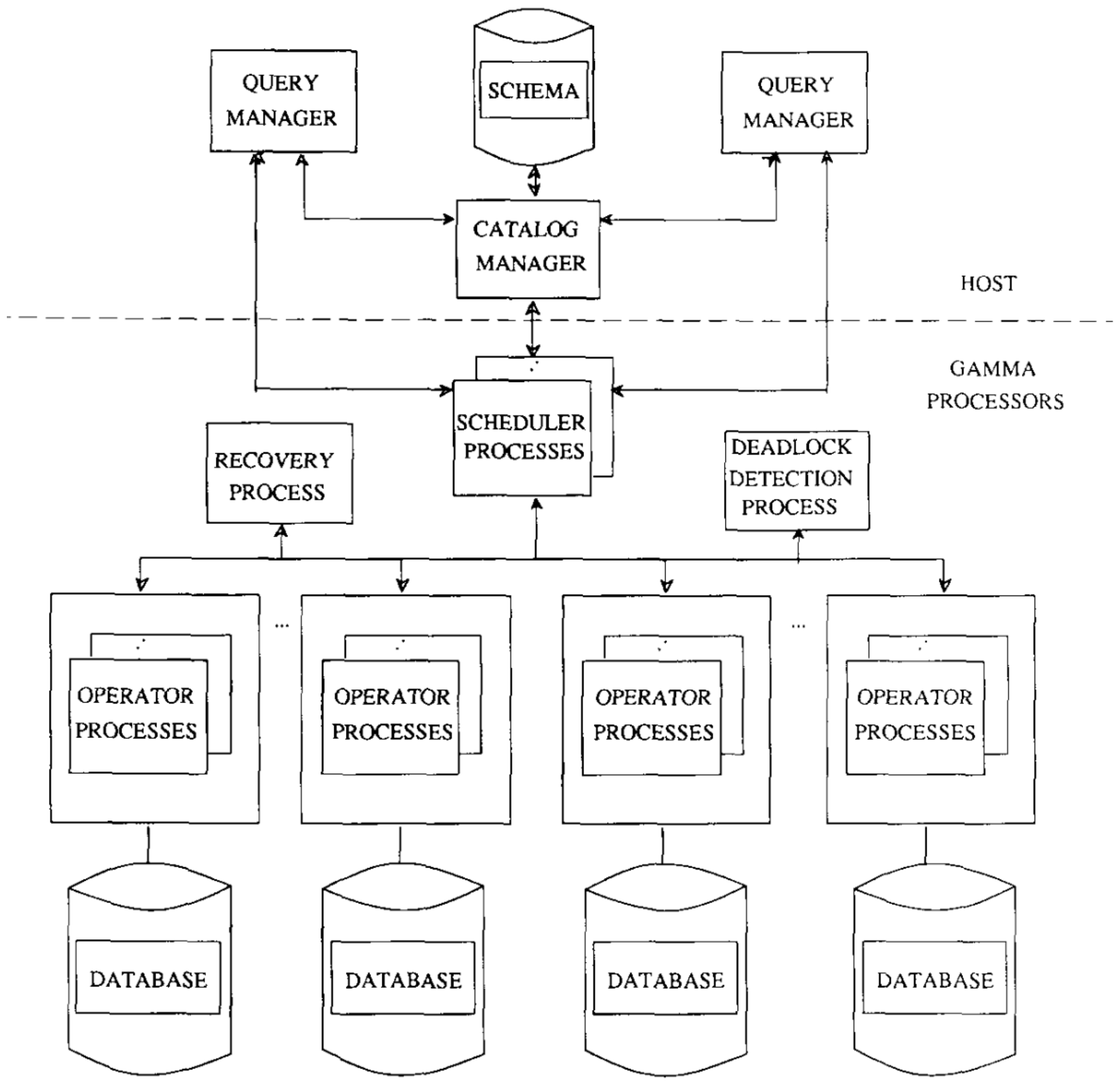
Hardware architecture

- Shared nothing
    - Each processor has its own disk
    - I/O bandwidth was pretty low
    - Can significantly increase overall I/O bandwidth by multiplication
        - Contrast to shared memory, where the I/O bus and processor interconnects become a bottleneck
    - Essentially "pushes CPUs to data", same lesson as Postgres!
- Paper describes long (and painful) history of challenges involved in connecting machines to each other and dealing with h/w and n/w issues
- DeWitt had a background in h/w - this was right up his alley

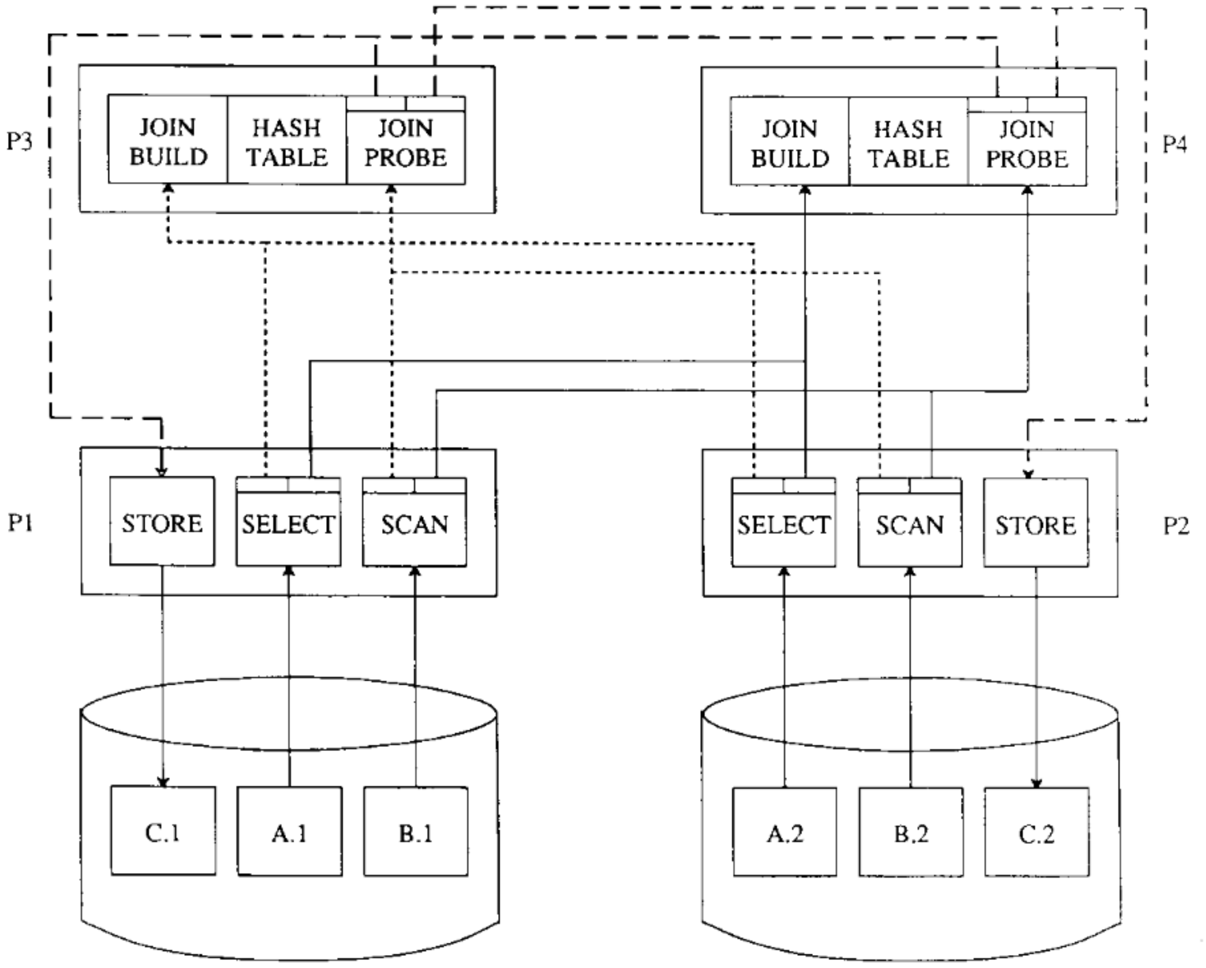
Software architecture

- Storage:

    - Default partitioning: round-robin, but user can override to hash/range-based
    - All stored in catalog - as is typical
    - Indexes are constructed on all partitions, global index = union
    - Partitioning is simply light-weight indexing;
        - so if partitioning is on attribute A via hash or range, and there is a predicate on A, can skip most partitions
    - Refreshing to see caveats - partitioned all relations, but possibly better to partition some
- Process structure:
    - One query manager process per user, responsible for parsing, optimization etc.
    - Communicates with scheduler process (overall scheduling) and catalog manager
    - Operator processes for each op in the query tree - for each participating processor



-
- Query language
    - QUEL variant - DeWitt and Stonebraker had a history
    - Simplified optimization:
        - Only hash-based joins
        - Left-deep

        - Rationale for no bushy/right-deep - not enough memory
    - Using left-deep and hashing = no more than two joins were active at a time
- Operator implementations
  - Each op is unaware that it is one of many sub-ops working on a partition
  - Reads streams of tuples, sends out streams - but batches into a group at a time
  - Outgoing stream is **split** on some value (e.g., something that signifies the destination).- so called a *split table*
    - for example hash on join attribute reveals split value
- Example where select (A) ⋈ B -> C
- 

Query Processing Algorithms

- Simple, Grace, Hybrid hash-join, sort-merge
- Hybrid hash join
  - Centralized variant:
    - partition inner relation R into N buckets
    - 1st bucket used to construct hash table, remaining N-1 for temp files
    - similarly partition S into N buckets
    - as before N-1 for temp files, while those in first bucket are used immediately to probe

Failure management

Node	0	1	2	3	4	5	6	7
Primary Copy	R0	R1	R2	R3	R4	R5	R6	R7
Backup Copy	r7	r0	r1	r2	r3	r4	r5	r6

Fig. 9. Chained declustering (relation cluster size = 8).

	Cluster 0				Cluster 1			
Node	0	1	2	3	4	5	6	7
Primary Copy	R0	R1	R2	R3	R4	R5	R6	R7
Backup Copy		r0.0	r0.1	r0.2		r4.0	r4.1	r4.2
	r1.2		r1.0	r1.1	r5.2		r5.0	r5.1
	r2.1	r2.2		r2.0	r6.1	r6.2		r6.0
	r3.0	r3.1	r3.2		r7.0	r7.1	r7.2	

Fig. 10. Interleaved declustering (cluster size = 4).

Node	0	1	2	3	4	5	6	7
Primary Copy	R0	---	$\frac{1}{7}$R2	$\frac{2}{7}$R3	$\frac{3}{7}$R4	$\frac{4}{7}$R5	$\frac{5}{7}$R6	$\frac{6}{7}$R7
Backup Copy	$\frac{1}{7}$r7	---	r1	$\frac{6}{7}$r2	$\frac{5}{7}$r3	$\frac{4}{7}$r4	$\frac{3}{7}$r5	$\frac{2}{7}$r6

Fig. 11. Fragment utilization with chained declustering after the failure of node 1 (relation cluster size = 8).

- Uses chained declustering instead of interleaved
  - Does a better job of work distribution and higher degree of availability than interleaved
- Chained:
  - if ith partition is stored in i mod M location; backup is stored in (i+1) mod M
- Interleaved:
  - subfragments per partition, evenly distributed across other partitions
- Both interleaved and chained can sustain one failure
- Single node failure:
  - load on each node to increase by 1/7
  - In this case, easy enough
- Downside of interleaved relative to chained:
  - availability.

- similar for one failure, but if there are two failures, then interleaved has *definitely* lost some data, but chained may not have