The Gamma Database Machine Project

David DeWitt, Shahram Ghandeharizadeh, Donovan Schcheider,
Allan Bricker, Hui-i Hsiao, and Rick Rasmussen

Slides adopted from those of Deepak Bastakoty,
and Ghandeharizadeh and DeWitt

Presenter: Jianhao Cao

Discussion Leader: Jeffrey Niu

UBC CPSC 504 – 2023.03.07

Outline

- Motivation
- Hardware Architecture
- Software Architecture
- Query Processing
- Transaction and Failure Management
- Performance
- Conclusion

Motivation

- Why parallel databases?
  - Obtain faster response time
  - Increase query throughput
  - Improve robustness to failure
  - Reduce processor workload
  - Enable scalability

Motivation

- DIRECT
  - Early parallel database project
  - Shared memory
  - Centralized control of parallel algorithms

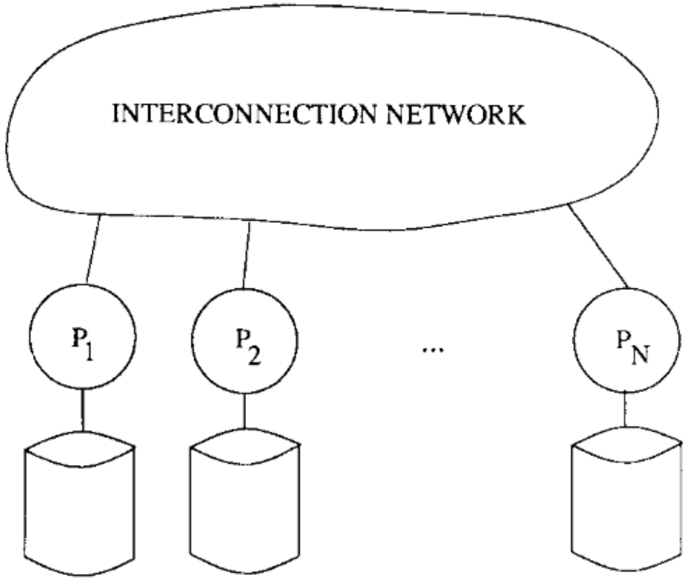
Motivation

- DIRECT
  - Early parallel database project
  - Shared memory
  - Centralized control of parallel algorithms

Impossible to scale the architecture to hundreds of processors!

Motivation

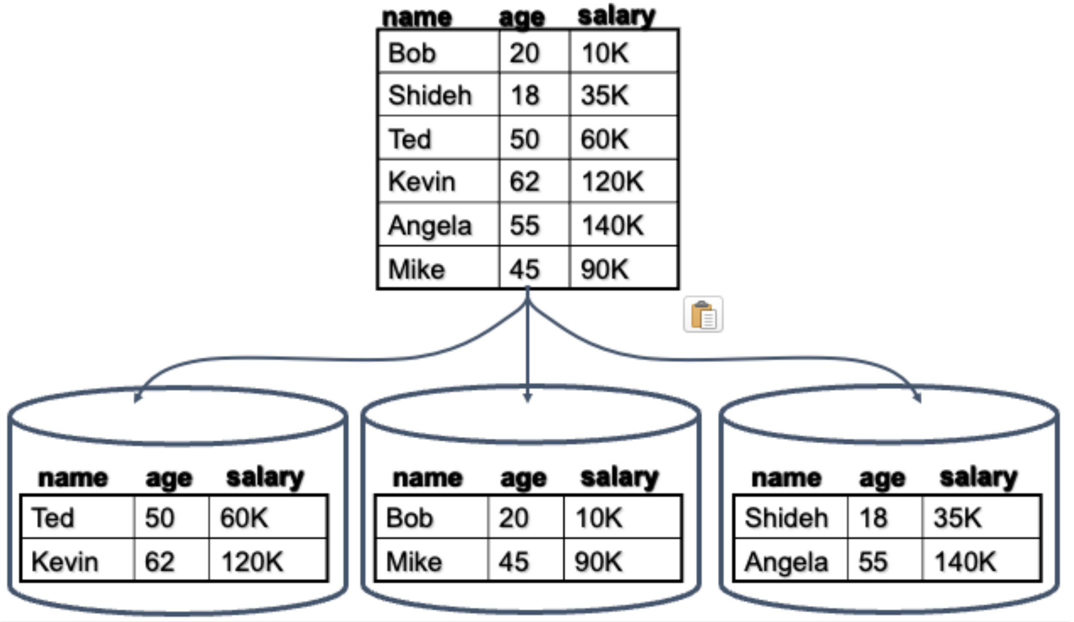
- Share-nothing
  - Each processor has it own memory or disk(s)
- Hash-based parallel algorithms
  - No centralized control



Motivation

❑ Horizontal partitioning (declustering)

- Tuples of a relation distributed over multiple disks.
- Round robin; hashed; range partitioned



Hardware Architecture

❑ GAMMA 1.0

- 17 VAX 11/750 processors, each with 2 MB memory
- Another VAX as the host machine
- An 80 Mb/s token ring to connect processors
- 8 processors attached with 333 MB disk drivers

❑ Problems

- The token ring network packet size is too small (2K bytes)
- The bandwidth mismatch between the token ring and the Unibus on the 11/750
- Insufficient memory for each processor

Hardware Architecture

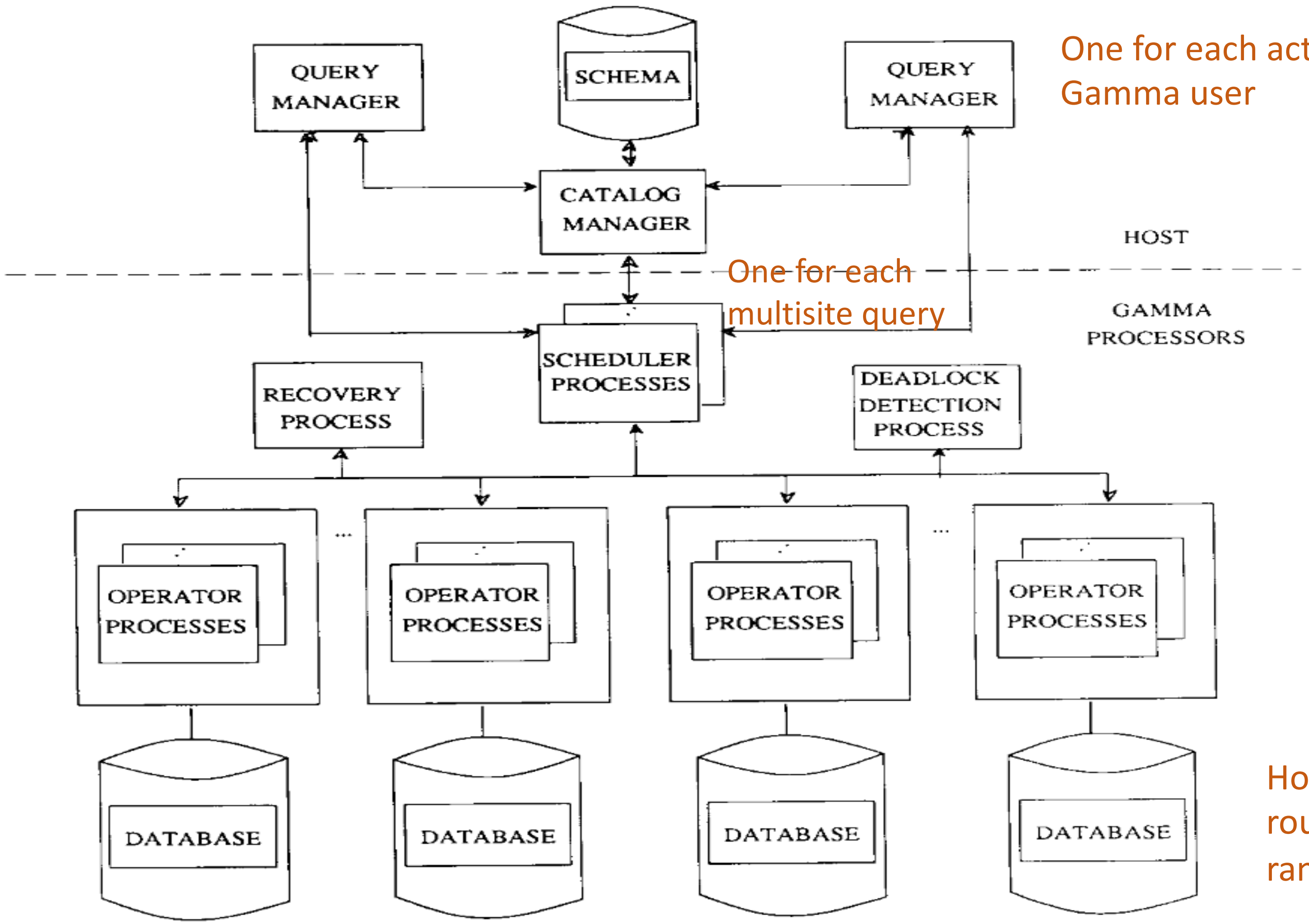
❑ GAMMA 2.0

- 32 processor iPSC/2 hypercube from Intel
- 386 CPU, 8 MB memory
- 330 MB MAXTOR 4380 disk drive with a 45 KB RAM buffer
- Custom VLSI routing modules for network communication
- NOSE (Gamma’s OS) run as a thread package inside a process

Discussion

Did the experience with VAX, iPSC/2 and the bugs they found to strengthen the paper, weaken it, or didn’t impact it? (Sid)

Software Architecture



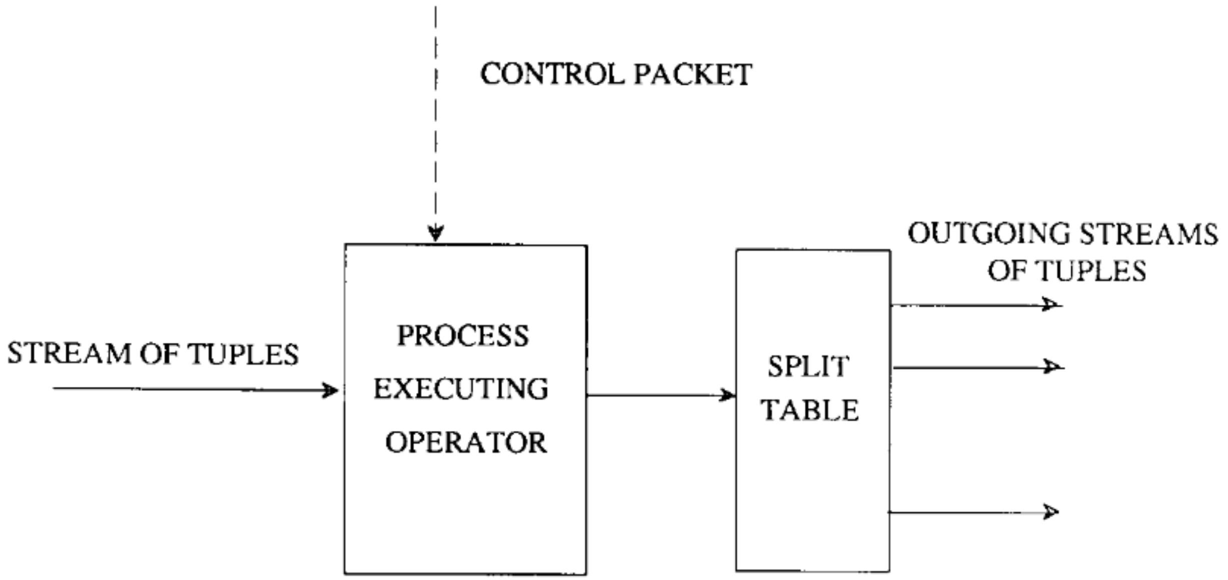
One for each active Gamma user

One for each multisite query

Horizontally partitioned data: round robin; hashed; range partitioned

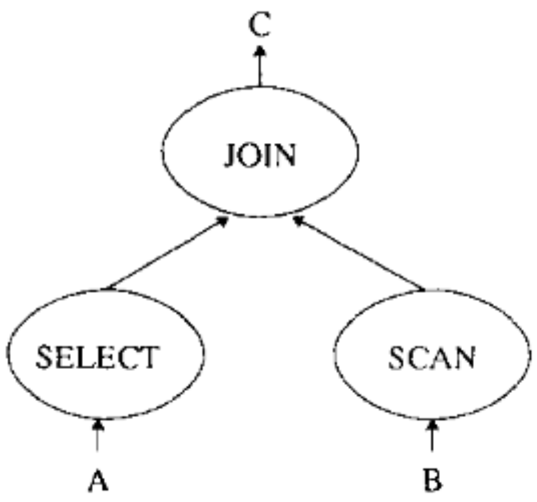
Software Architecture

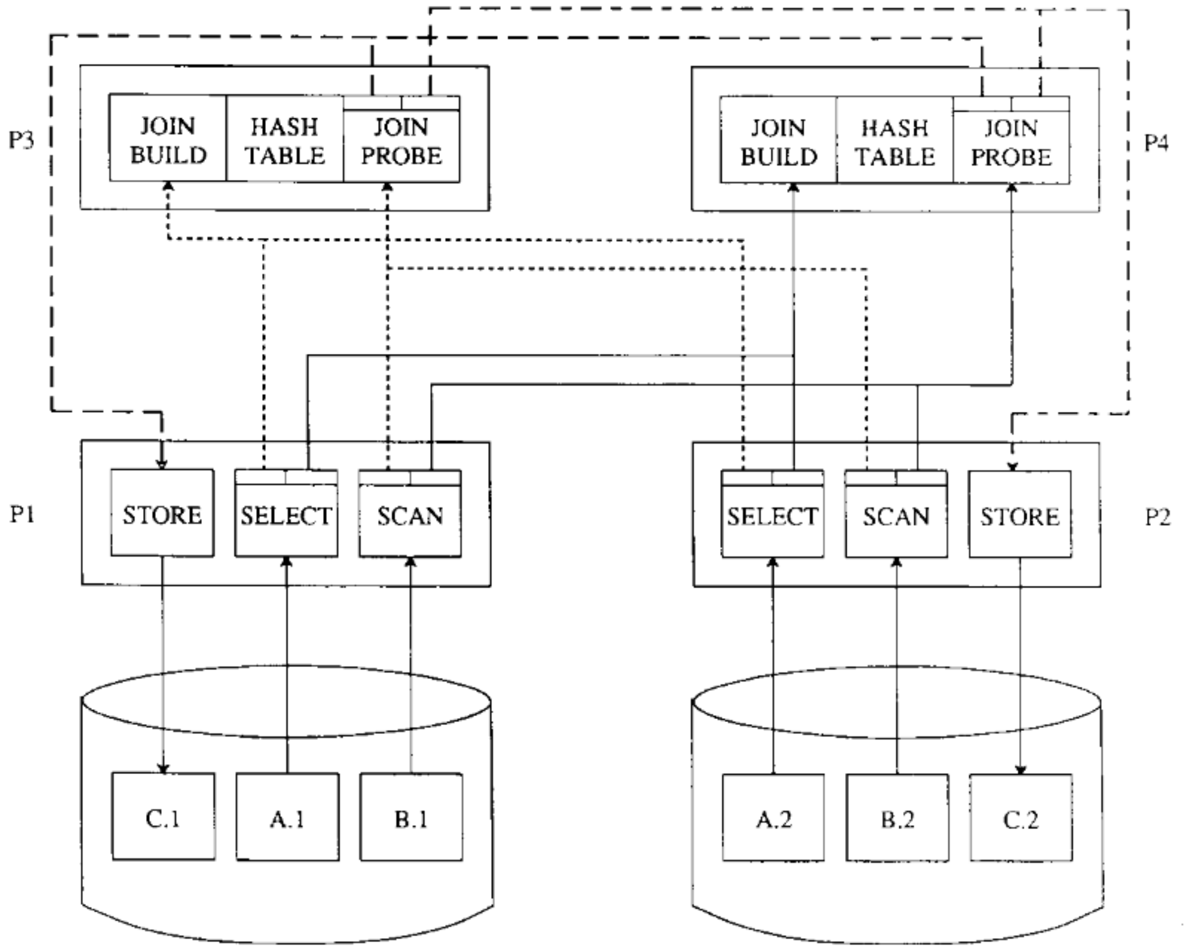
The split table defines a mapping of values to a set of destination processes.



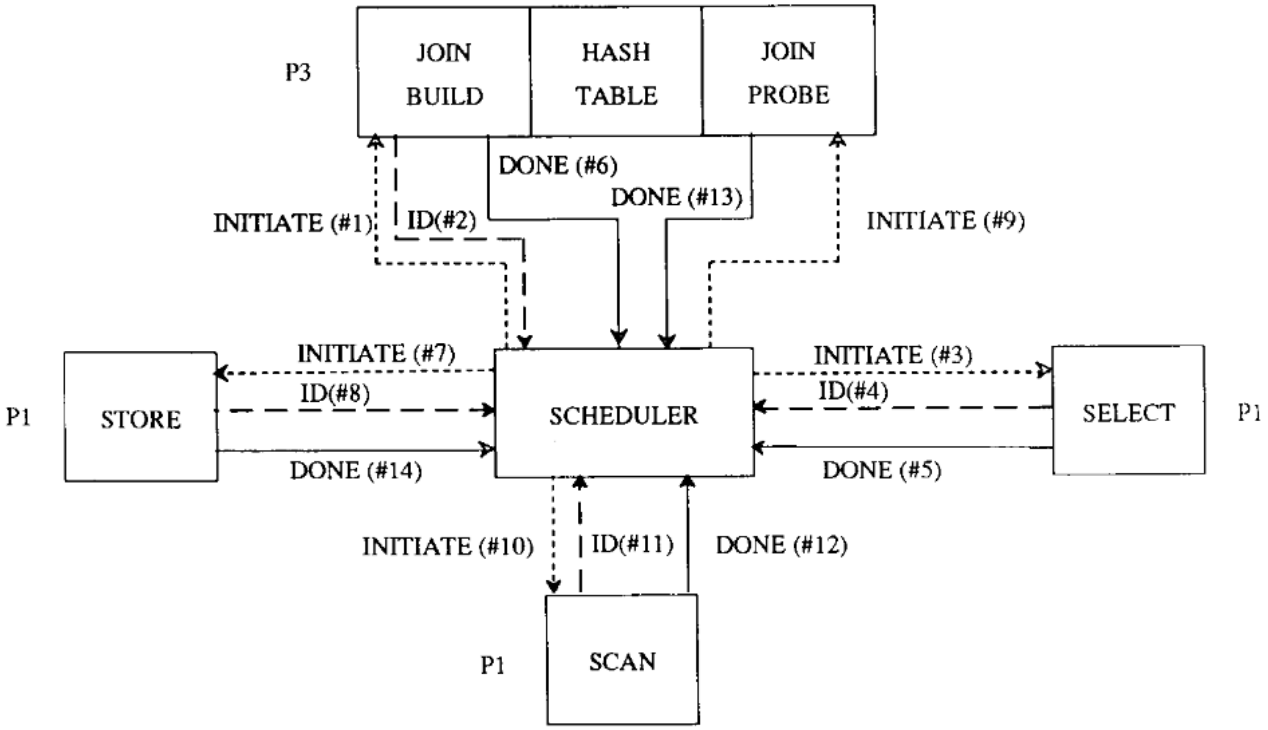
Value	Destination Process
0	(Processor #3, Port #5)
1	(Processor #2, Port #13)
2	(Processor #7, Port #6)
3	(Processor #9, Port #15)

The Parallel Simple Hash Join





Data flow



Control flow

Query Processing

❑ Selection

- Selection on the partitioning attribute
  - Direct the selection to a subset of nodes if hash or range partitioned.
  - Initiate the selection on all nodes if round-robin partitioned.

❑ Join

- Partition relations into disjoint subsets (buckets) by hashing on the join attribute.
- Four types of parallel joins: sort-merge, Grace, Simple, Hybrid.
- The Hybrid hash join almost always provides the best performance.

Query Processing Algorithms

❑ Aggregate functions

- Each processor computes a partial results on its partition.
- The processors redistribute the results on hashing on the “group by” attribute.

❑ Update operators

- Most operators are implemented with standard techniques.
- A replace operator will send a tuple to the partition to which it belongs.

Ideal Parallelism

- **Speedup**

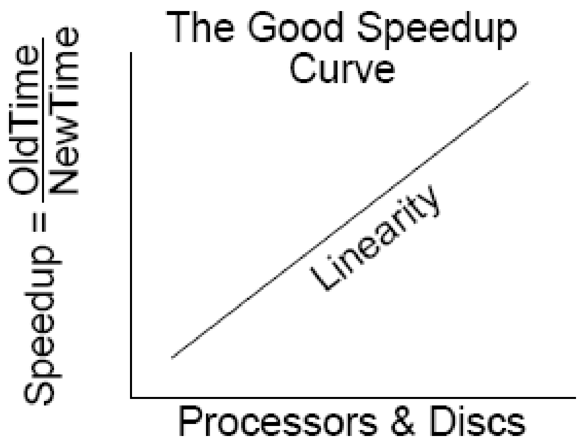
  Given a system with 1 node, does adding *n* nodes speed it up with a factor of *n* ?

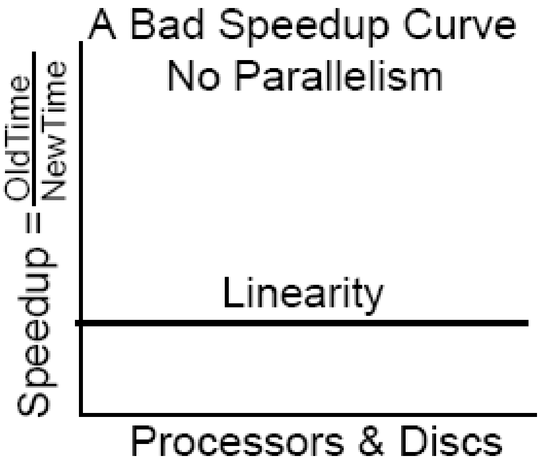
$$Speedup = \frac{small\_system\_elapsed\_time}{big\_system\_elapsed\_time}$$

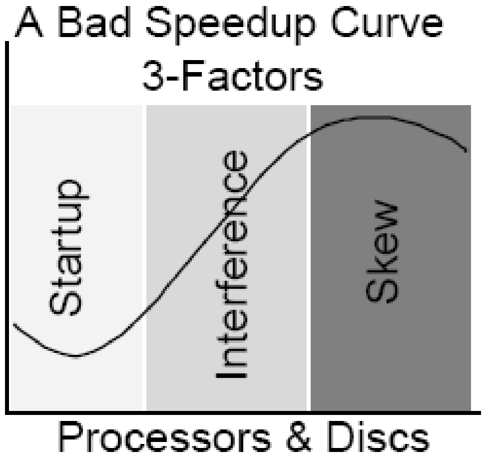
- **Scaleup**

  Given a system with 1 node, does the response time remain the same with *n* nodes ?

$$Scaleup = \frac{small\_system\_elapsed\_time\_on\_small\_problem}{big\_system\_elapsed\_time\_on\_big\_problem}$$







Conclusion

- Three key ideas that enable Gamma to be scaled to hundreds of processors:
  - Horizontally partitioning
  - Extensive use of hash-based parallel algorithms
  - Dataflow scheduling techniques for multioperator queries

Discussion

What are the similarities and differences between parallel databases and data integration?

- Problem setup (motivation, what/where data is available)
- Goals (what does the system aim for?)

MapReduce: Simplified Data Processing on Large Clusters

Jeff Dean, Sanjay Ghemawat

Google, OSDI 2004

Slides based on those by authors and other online sources

Presenter: Jianhao Cao

Discussion Leader: Jeffrey Niu

UBC CPSC 504 – 2023.03.07

Motivation

- Large scale data processing
  - Using hundreds or thousands of machines but without the hassle of management
- MapReduce benefits
  - Automatic parallelization & distribution
  - Fault tolerance
  - I/O scheduling
  - Monitoring & status updates

Programming model

- Input & Output: each a set of key/value pairs
- Programmer specifies two functions:

```
map(in_key, in_value) -> list(out_key, intermediate_value)
```

  - Processes each input key/value pair
  - Produces set of intermediate pairs

```
reduce(out_key, list(intermediate_value)) -> list(out_value)
```

  - Combines all intermediate values for a particular key
  - Produces a set of merged output values (usually just one)
- Inspired by similar primitives in LISP and other functional languages

Example: Count word occurrences

- Input: (URL, content) pairs
- `map(key=URL, value=content)`:
  - for each word w in content, output (w, 1)
- `reduce(key=word, values=uniq_counts_list)`
  - sum all 1’s in uniq_counts_list
  - output(word, sum)

Word count example illustrated

map(key=url, val=content):

For each word *w* in contents, emit (w, “1”)

reduce(key=word, values=uniq_counts_list):

Sum all “1”s in values list

Emit result “(word, sum)”

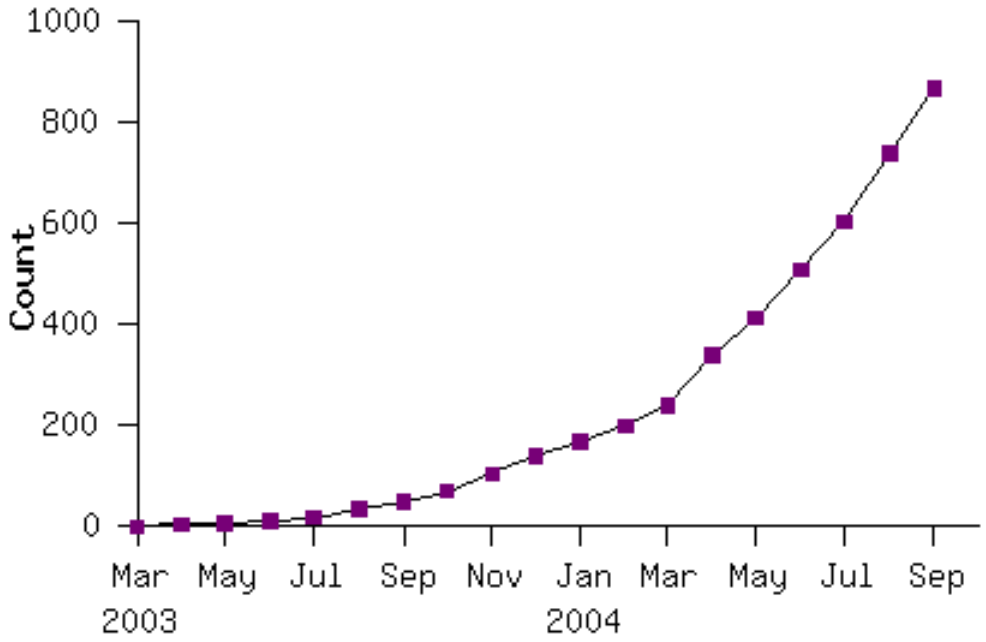
see bob throw
see spot run

see	1
bob	1
run	1
see	1
spot	1
throw	1

bob	1
run	1
see	2
spot	1
throw	1

MapReduce model widely applicable

- MapReduce prog



Examples

distributed grep	distributed sort	web link-graph reversal
term-vector / host	web access log stats	inverted index construction
document clustering	machine learning	statistical machine translation
...	...	...

Implementation overview

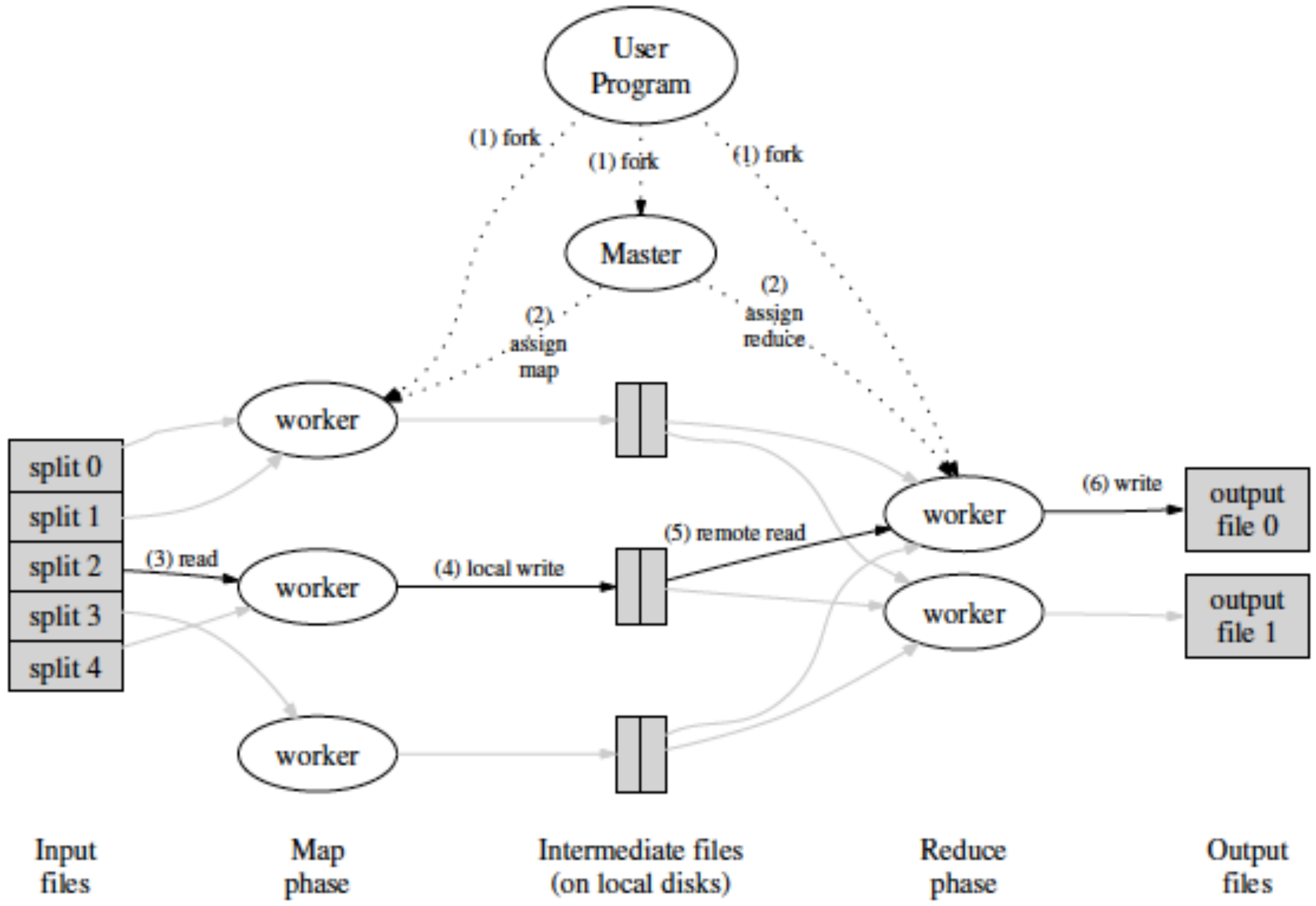
- Typical cluster:
  - 100s/1000s of 2-CPU x86 machines, 2-4 GB of memory
  - Limited bisection bandwidth
  - Storage is on local IDE disks
  - GFS: distributed file system manages data (SOSP'03)
  - Job scheduling system: jobs made up of tasks, scheduler assigns tasks to machines
- Implementation as C++ library linked into user programs

Discussion

The implementation hardware is quite impressive.

- Is it helpful for entities like Google to release papers on projects that are out of scope for most others? (Jason)
- If you were in a less resourceful setting, how would you approach a research topic like this? How would the research be different (e.g. evaluation)? (Michael)

Overall execution workflow



Fault-tolerance via re-execution

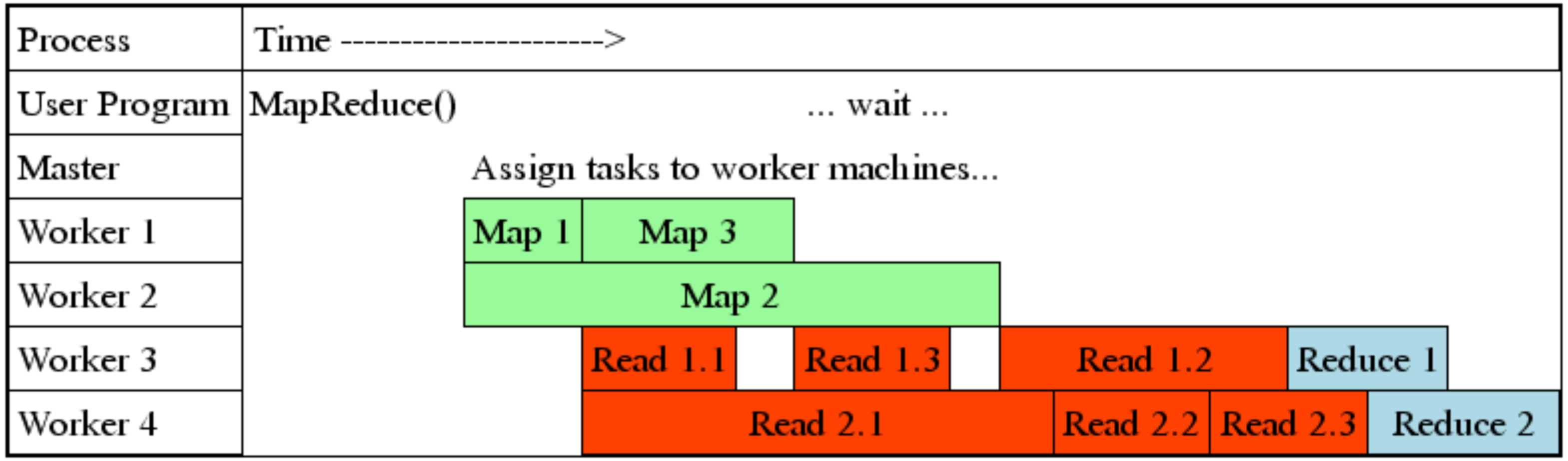
- On worker failure:
  - Detect failure via periodic heartbeats
  - Re-execute completed and in-progress *map* tasks
    - Output stored on the local disk becomes inaccessible
  - Re-execute in progress *reduce* tasks
    - Output stored in a global file system
  - Task completion committed through master
- Master failure:
  - Left unhandled as considered unlikely
  - Abort the MapReduce computation
- Robust: lost 1600 of 1800 machines, but finished fine

Refinement: Locality Optimization

- Master scheduling policy:
  - Asks GFS for locations of replicas of input file blocks
  - Map tasks typically split into 64MB (== GFS block size)
  - Map tasks scheduled so GFS input block replica are on same machine or same rack
- Effect: Thousands of machines read input at local disk speed
  - Without this, rack switches limit read rate

Refinement: Task Granularity

- Fine granularity tasks: map tasks >> machines
  - Minimizes time for fault recovery
  - Can pipeline shuffling with map execution
  - Better dynamic load balancing
- Often use 200K map and 5000 reduce tasks running on 2000 machines



Refinement: Backup Execution

- Slow workers significantly lengthen completion time
  - Other jobs consuming resources on machine
  - Bad disks with soft errors transfer data very slowly
  - Weird things: processor caches disabled (!!)
- Solution: Near end of phase, start backup task copies
  - Whichever one finishes first "wins"
- Benefit: Dramatically shortens job completion time

Refinement: Skipping Bad Records

- Map/Reduce functions sometimes fail for particular inputs
  - Best solution is to debug & fix, but not always possible
- On segmentation fault:
  - Send UDP packet to master from the signal handler
  - Include sequence number of record being processed
- If master sees two failures for the same record:
  - Next worker is told to skip the record
- Effect: Can work around bugs in third-party libraries

Other Refinements

- Sorting guarantees within each reduce partition
- Compression of intermediate data
- Combiner: useful for saving network bandwidth
- Local sequential execution for debugging/testing
- User-defined counters

Google Experience: Rewrite of Production Indexing System

- Rewrote Google's production indexing system using MapReduce
  - New code is simpler, easier to understand
  - MapReduce takes care of failures, slow machines
  - Easy to make indexing faster by adding more machines

Conclusions

- MapReduce has proven to be a useful abstraction.
- Network bandwidth is a scarce resource.
- Redundant execution can reduce the impact of slow machines and machine failures.

Discussion

- In 2008, David DeWitt (author on the Gamma paper) and Michael Stonebraker (author on What Goes Around Comes Around) wrote a scathing review of MapReduce, calling it "a major step backwards".
- In it, they lament that MapReduce ignores lessons from 40 years of database technology and that schools are even teaching MapReduce to first-year students.

Discussion

In the article, they present five criticisms of MapReduce:

1. MapReduce is a step backward in database access
   - MapReduce doesn’t have schemas, data independence, and high-level access languages
   - No different than CODASYL
2. MapReduce is a poor implementation
   - No indices, essentially brute-force sequential search
   - No experimental evaluation to prove it scales
3. MapReduce is not novel
   - Concepts have been introduced 20 years ago
   - MapReduce no different from user-defined aggregate functions

Discussion

4. MapReduce is missing features
   - Indices, updates to change data in database, transactions, integrity constraints
5. MapReduce incompatible with DBMS tools
   - Report writers (prepare reports for human visualization)
   - Data mining (discovery of structure in large datasets)
   - Database design tools (assist user in constructing database)
   - Hard to use MapReduce in end-to-end task without these tools

Discussion

1. MapReduce is a step backward in database access
2. MapReduce is a poor implementation
3. MapReduce is not novel
4. MapReduce is missing features
5. MapReduce incompatible with existing DBMS tools

Are these criticisms valid, invalid, or irrelevant?

✔ = valid

X = invalid

O = irrelevant