Parallel Data Management

Parallel DBMS

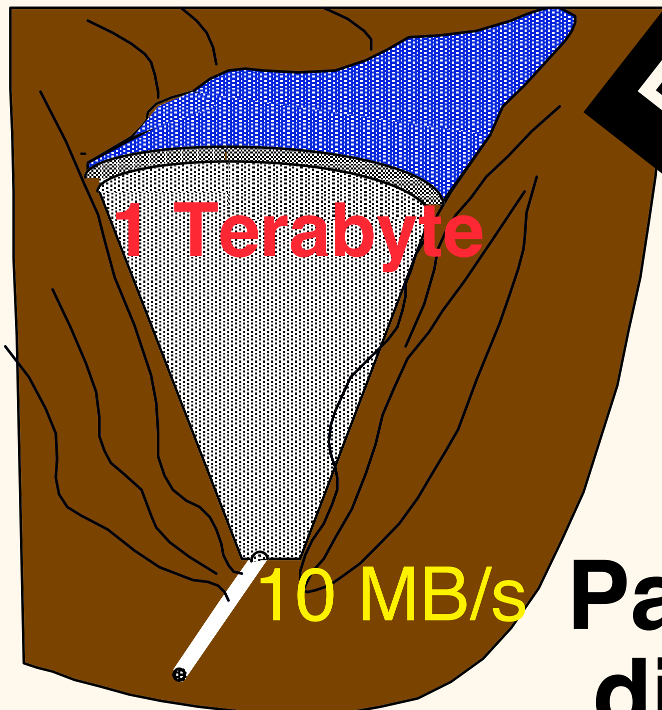
Module 9, Lecture 1

Slides by Joe Hellerstein, UCB, with some material from Jim Gray, Microsoft Research. See also:

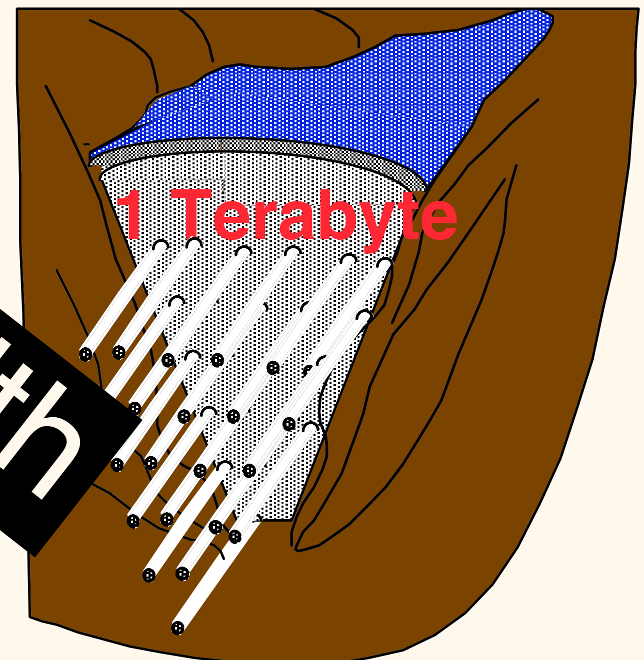
http://www.research.microsoft.com/research/BARC/Gray/PDB95.ppt

Why Parallel Access To Data?

At 10 MB/s
1.2 days to scan



1,000 x parallel
1.5 minute to scan.

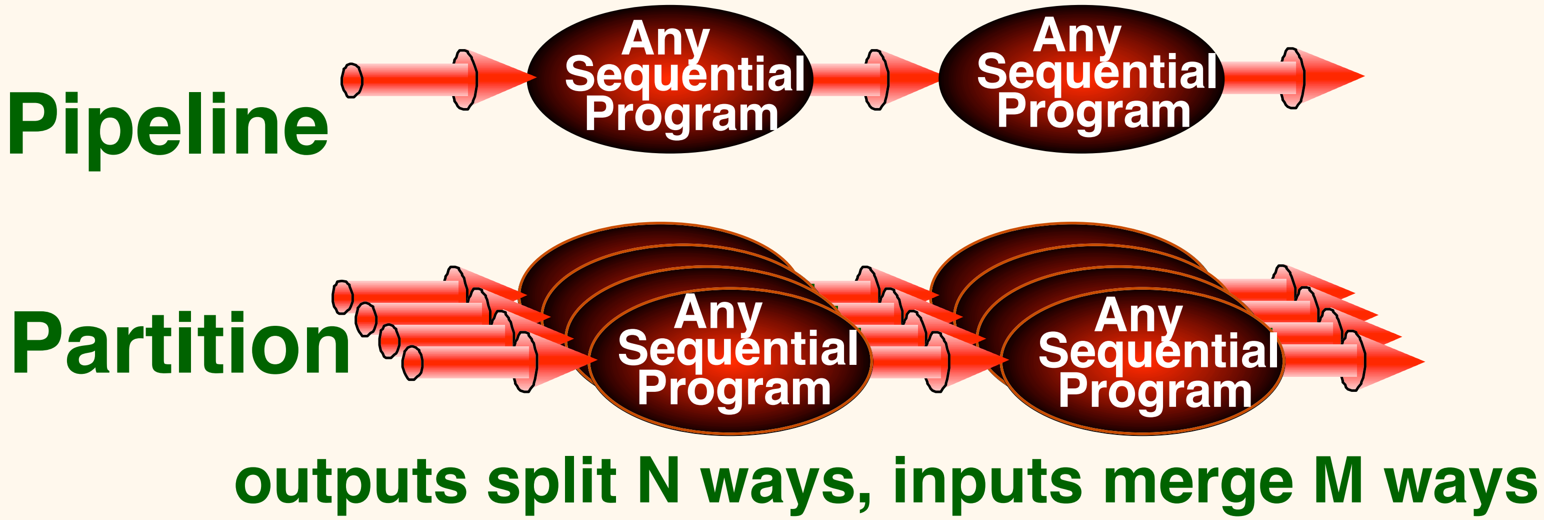


Bandwidth

Parallelism:
divide a big problem
into many smaller ones
to be solved in parallel.

Parallel DBMS: Intro

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

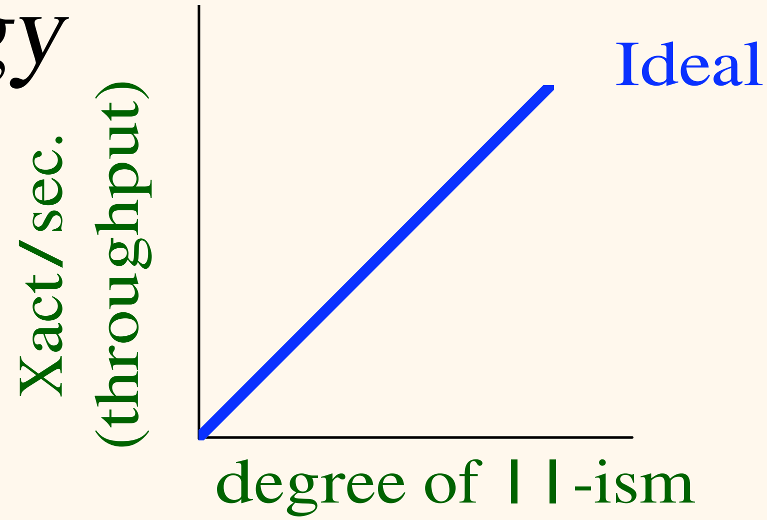


DBMS: The || Success Story

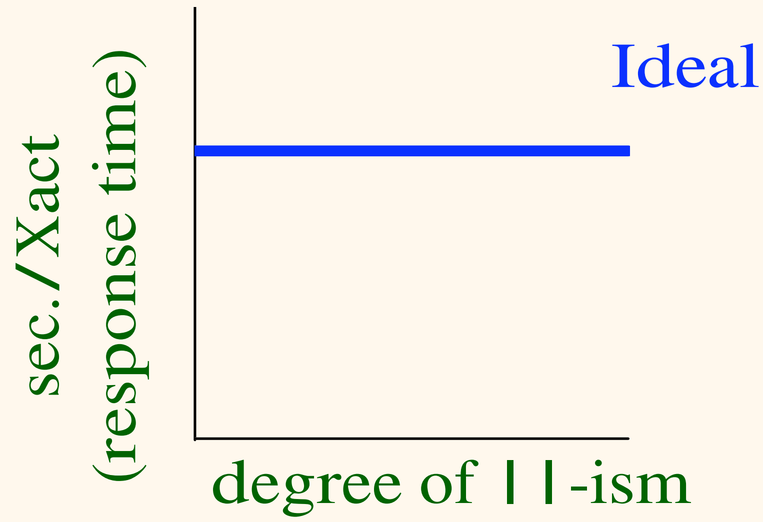
- DBMSs are the most (only?) successful application of parallelism.
  - Teradata, Tandem vs. Thinking Machines, KSR..
  - Every major DBMS vendor has some || server
  - Workstation manufacturers now depend on || DB server sales.
- Reasons for success:
  - Bulk-processing (= partition ||-ism).
  - Natural pipelining.
  - Inexpensive hardware can do the trick!
  - Users/app-programmers don't need to think in ||

Some || Terminology

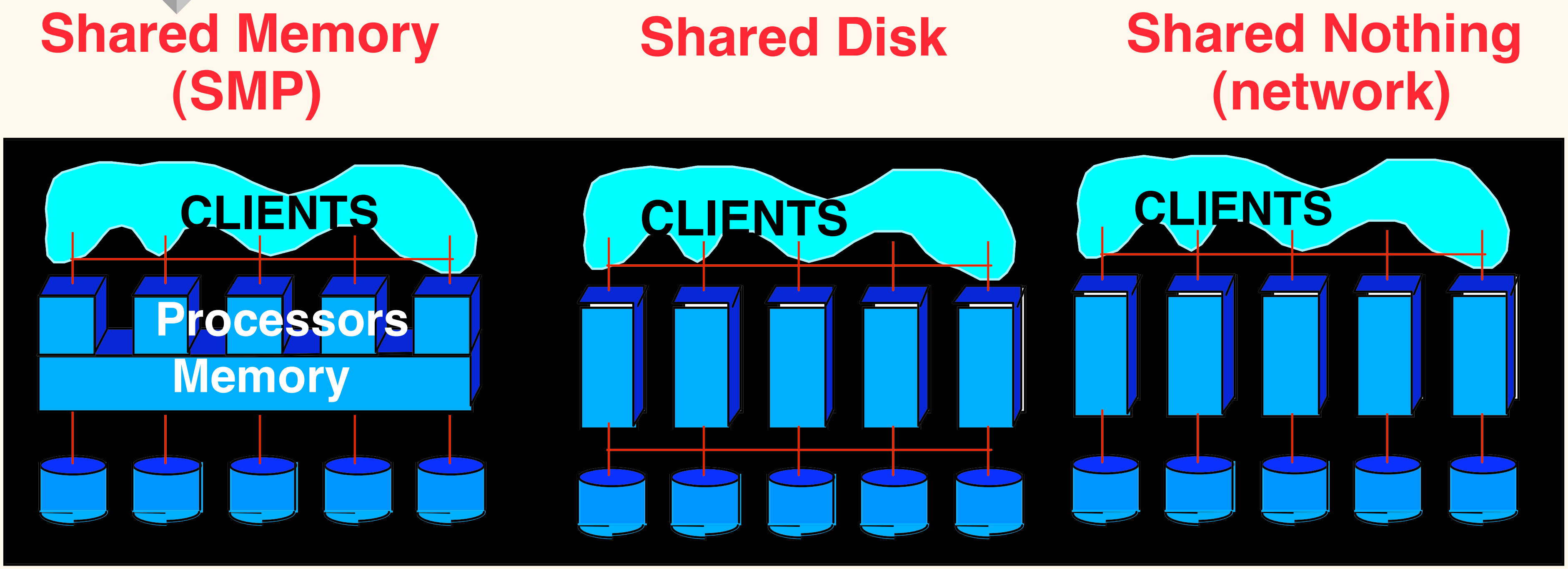
- Speed-Up
  - More resources means proportionally less time for given amount of data.



- Scale-Up
  - If resources increased in proportion to increase in data size, time is constant.



Architecture Issue: Shared What?



Easy to program
Expensive to build
Difficult to scaleup
Sequent, SGI, Sun

VMScluster, Sysplex

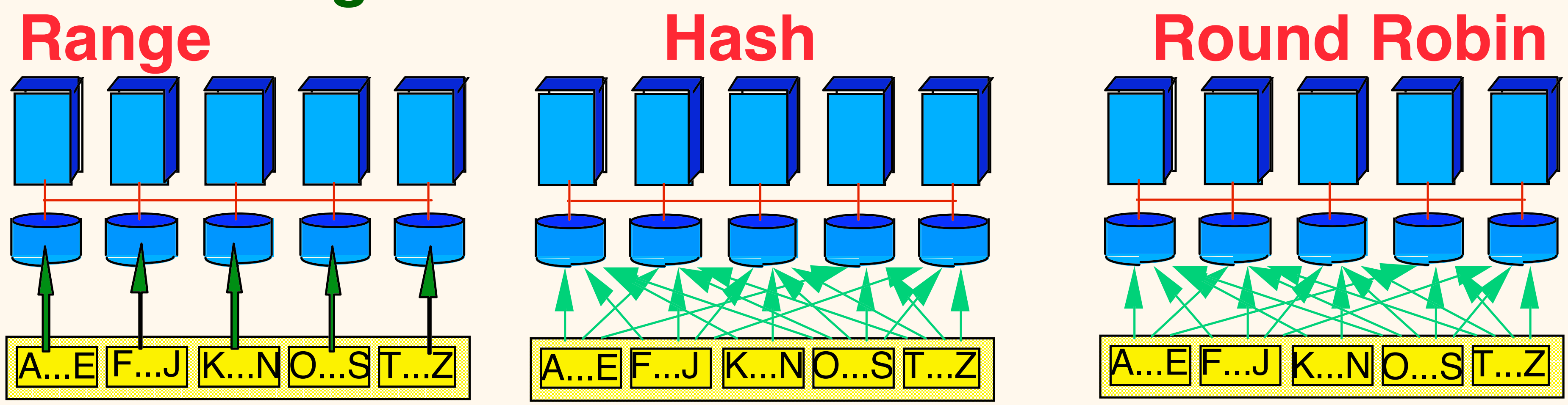
Hard to program
Cheap to build
Easy to scaleup
Tandem, Teradata, SP2

Different Types of DBMS || -ism

- Intra-operator parallelism
  - get all machines working to compute a given operation (scan, sort, join)
- Inter-operator parallelism
  - each operator may run concurrently on a different site (exploits pipelining)
- Inter-query parallelism
  - different queries run on different sites
- We'll focus on intra-operator || -ism

Automatic Data Partitioning

Partitioning a table:



Good for equijoins, range queries group-by

Good for equijoins

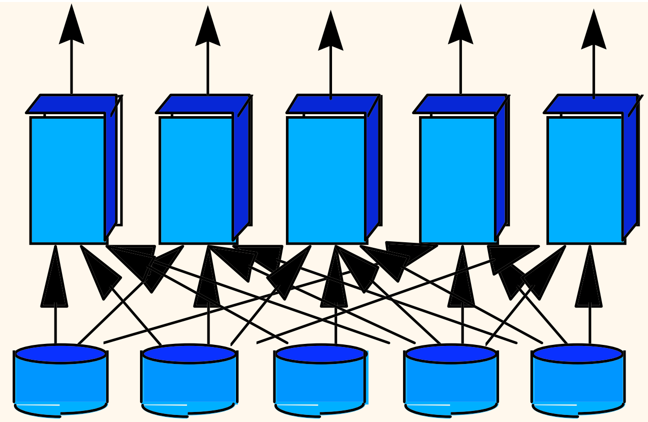
Good to spread load

Shared disk and memory less sensitive to partitioning,
Shared nothing benefits from "good" partitioning

Parallel Scans

- Scan in parallel, and merge.
- Selection may not require all sites for range or hash partitioning.
- Indexes can be built at each partition.
- Question: How do indexes differ in the different schemes?
  - Think about both lookups and inserts!
  - What about unique indexes?

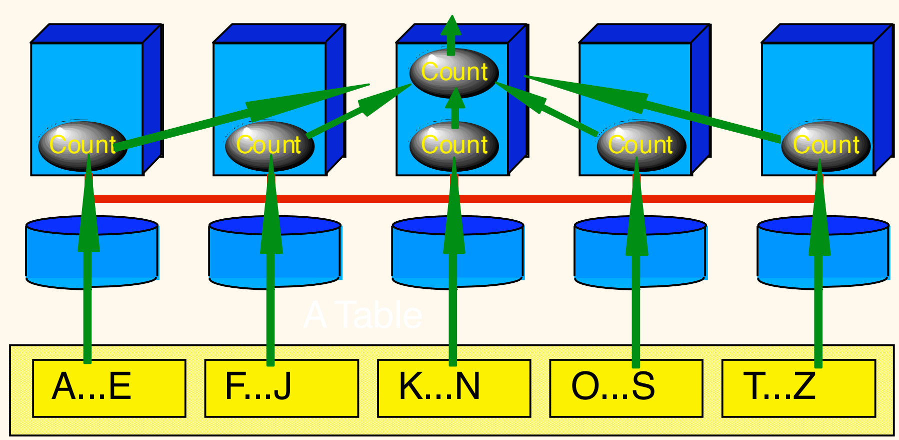
Parallel Sorting



- Current records:
  - 8.5 Gb/minute, shared-nothing; Datamation benchmark in 2.41 secs (UCB students! http://now.cs.berkeley.edu/NowSort/)
- Idea:
  - Scan in parallel, and range-partition as you go.
  - As tuples come in, begin “local” sorting on each
  - Resulting data is sorted, and range-partitioned.
  - Problem: *skew!*
  - Solution: “sample” the data at start to determine partition points.

Parallel Aggregates

- For each aggregate function, need a decomposition:
  - count(S) = Σ count(s(i)), ditto for sum()
  - avg(S) = (Σ sum(s(i))) / Σ count(s(i))
  - and so on...
- For groups:
  - Sub-aggregate groups close to the source.
  - Pass each sub-aggregate to its group's site.
    - Chosen via a hash fn.

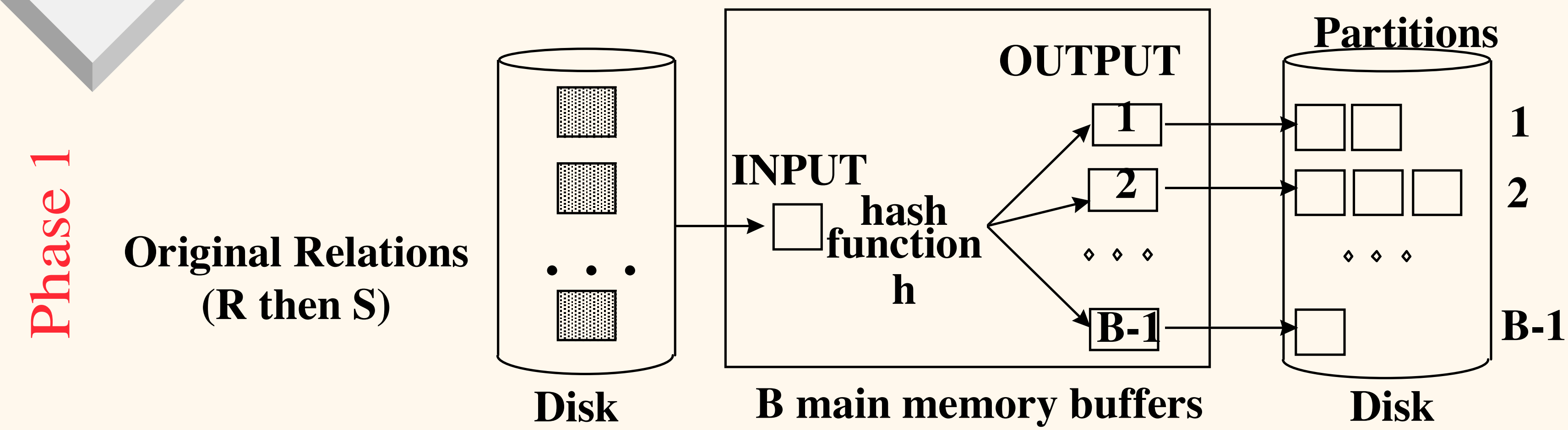


EXAMPLE

Parallel Joins

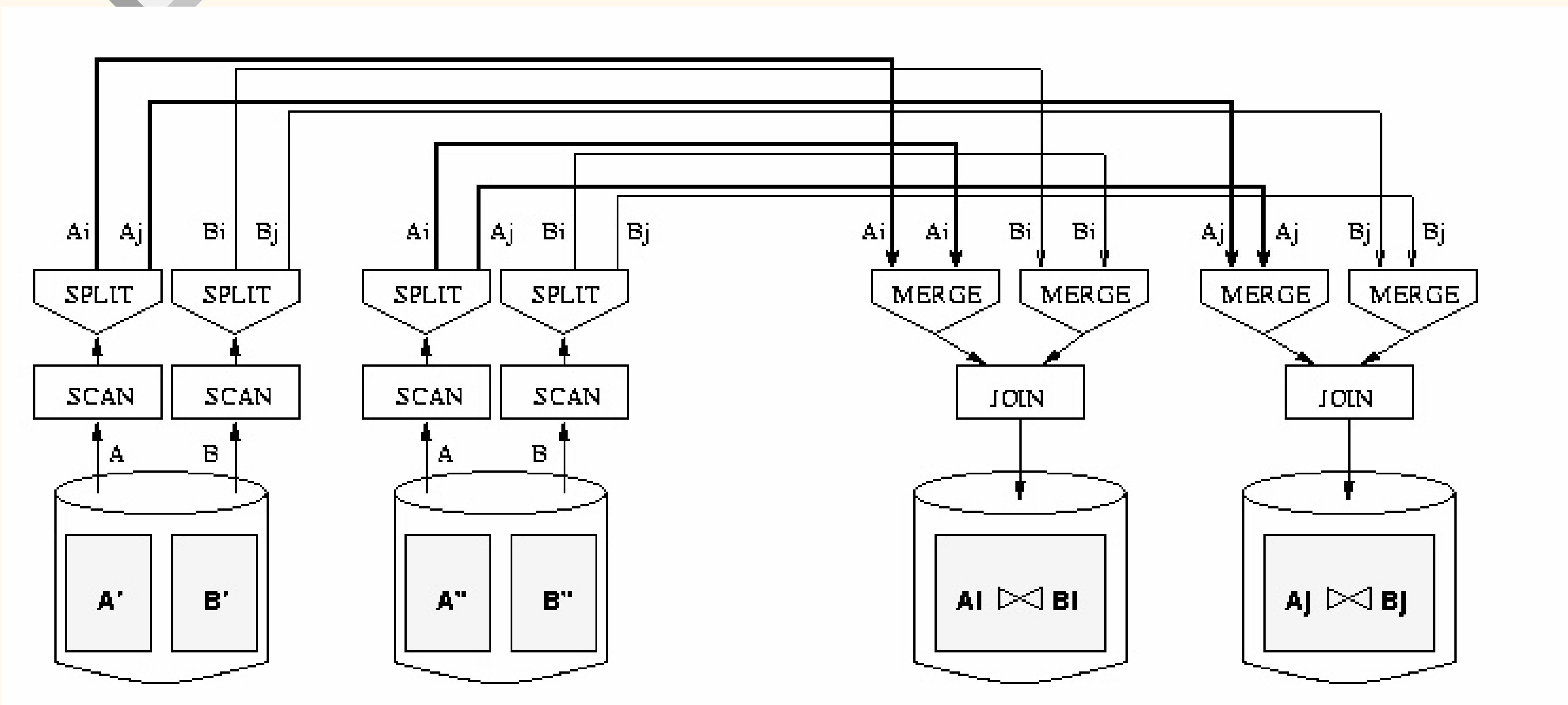
- Nested loop:
  - Each outer tuple must be compared with each inner tuple that might join.
  - Easy for range partitioning on join cols, hard otherwise!
- Sort-Merge (or plain Merge-Join):
  - Sorting gives range-partitioning.
    - But what about handling 2 skews?
  - Merging partitioned tables is local.

Parallel Hash Join



- In first phase, partitions get distributed to different sites:
  - A good hash function *automatically* distributes work evenly!
- Do second phase at each site.
- Almost always the winner for equi-join.

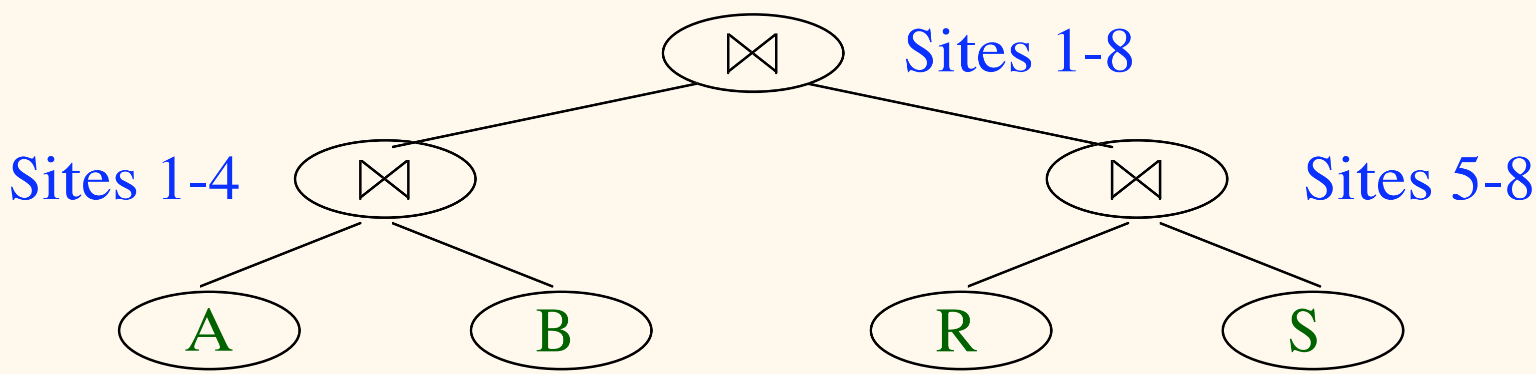
Dataflow Network for | | Join



- ❖ Good use of split/merge makes it easier to build parallel versions of sequential join code.

Complex Parallel Query Plans

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



Observations

- It is relatively easy to build a fast parallel query executor
  - S.M.O.P.
- It is hard to write a robust and world-class parallel query optimizer.
  - There are many tricks.
  - One quickly hits the complexity barrier.
  - Still open research!

Parallel Query Optimization

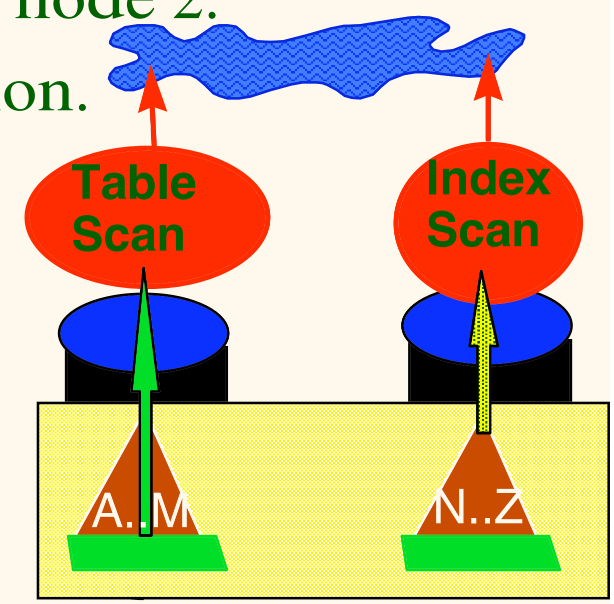
- Common approach: 2 phases
  - Pick best sequential plan (System R algorithm)
  - Pick degree of parallelism based on current system parameters.
- "Bind" operators to processors
  - Take query tree, "decorate" as in previous picture.

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.
- SELECT *

  FROM telephone_book

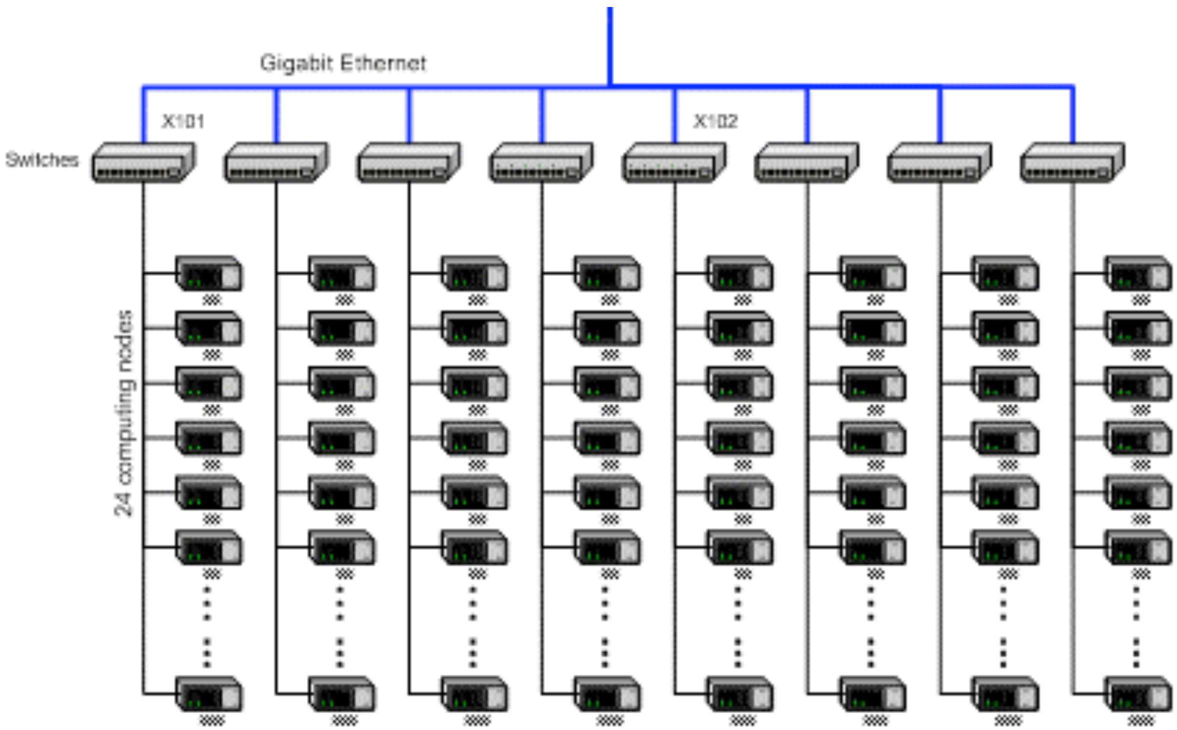
  WHERE name < "NoGood";



Google Approach to Systems Engineering

Prof. Christof Fetzer, Ph.D.
Heinz-Nixdorf Endowed Chair for Systems Engineering
TU Dresden

Localize: Network Architecture



© Nasa

Approach [2]

- Goal:
  - Hide the complexity of parallelism, data distribution and fault-tolerance
- Approach: MapReduce
  - Simplify programming by hiding these issues in a library
  - The programmer focuses on the problem at hand (e.g., counting URL access frequency)
  - Two phase approach:
    - Map: generates a list of intermediate results
    - Reduce: generates list of final results

Map

- Produces a list of intermediate results
- Name comes from map function in LISP
  - `(map 'list #'+ '(1 2 3) '(1 2 3)) => (2 4 6)`
- Example:
  - Count the number of words over a collection of documents
  - Input: list(document, content)
  - Output: list(word, total_count)

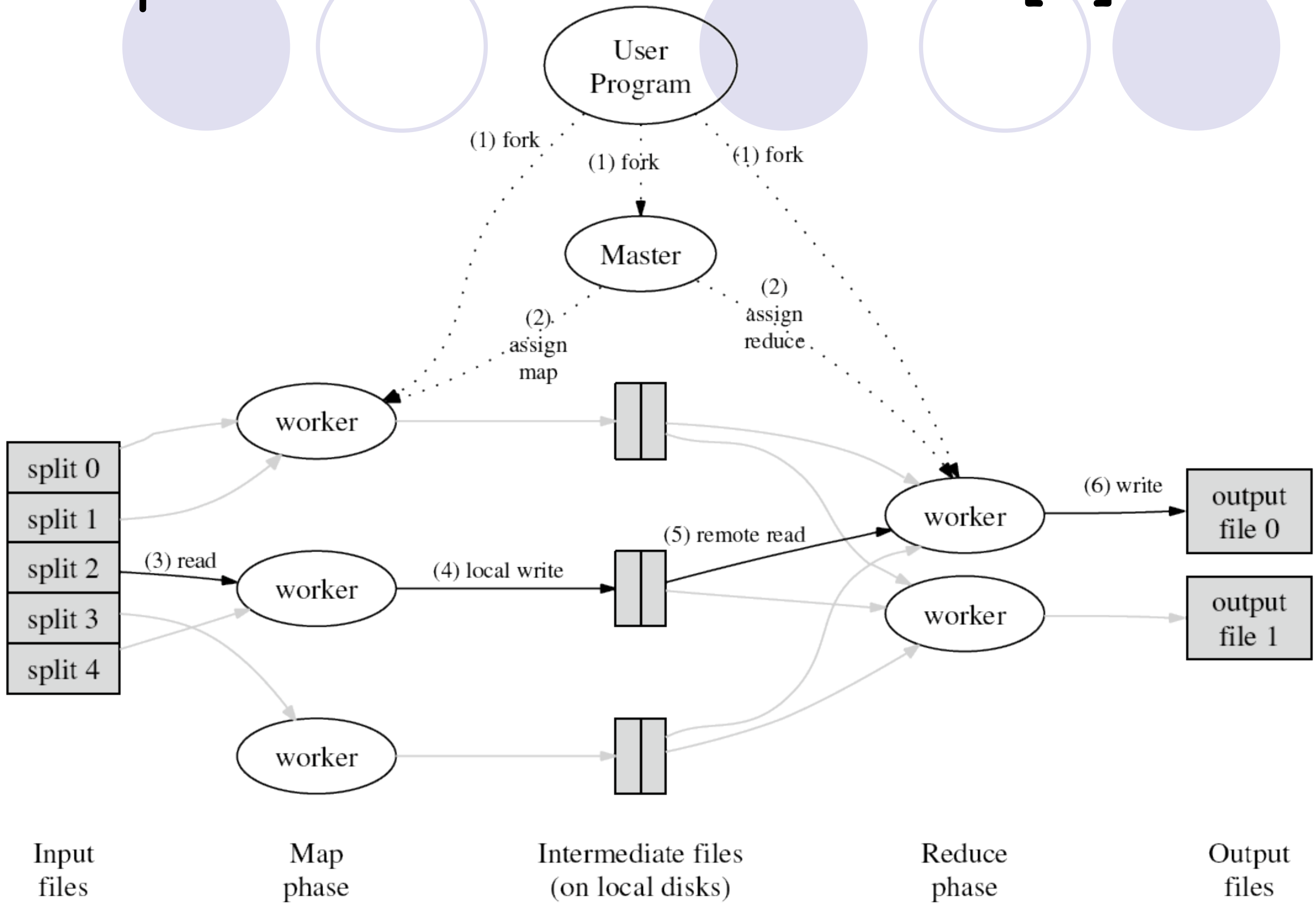
```
map(document, content) {
      for each word in content
            emit(word, "1")
}
```

Reduce

- Reduce combines intermediate results
- Name comes from reduce function in LISP
  - `(reduce #'+ '(1 2 3 4 5)) =>  15`
- Example:
  - Intermediate result: list(word, list(value))

```
reduce(word, values) {
     result = 0;
     for each value in values
          result += value
     emitString(w, result)
}
```

Implementation Architecture [2]



Combiner Function

- Problem:
  - intermediate results can be quite verbose
  - e.g., ("the", 1) could occur many times in previous example
- Approach:
  - perform a local reduction before writing intermediate results
  - typically, combiner same function as reduce func
- This will reduce the run-time because less writing to disk and across the network

Problem: Stragglers

- Often some machines are late in their replies
  - slow disk, overloaded, etc
- Approach:
  - when only few tasks left to execute, start backup tasks
  - a task completes when either primary or backup completes task
- Performance:
  - without backup, sort (->) takes 44% longer

Machine Uptime (1999 data, NT)

Item	Machine Uptime Statistics	Machine Downtime Statistics
Number of entries	616	682
Maximum	85.2 days	15.76 days
Minimum	1 hour	1 second
Average	11.82 days	1.97 hours
Median	5.54 days	11.43 minutes
Standard Deviation	15.656 days	15.86 hours

FROM:

Failure data analysis of a LAN of Windows NT based computers
Kalyanakrishnam, M.; Kalbarczyk, Z.; Iyer, R.;
Reliable Distributed Systems, 1999. Proceedings of the 18th IEEE Symposium on19-22 Oct. 1999
Page(s):178 - 187

Implications

- Probability that a given machine fails might be sufficiently low for some jobs
  - Probability that no machine fails is typically not acceptable for large jobs (many machines and/or long runtime)
- Software needs to be able to cope with failures!

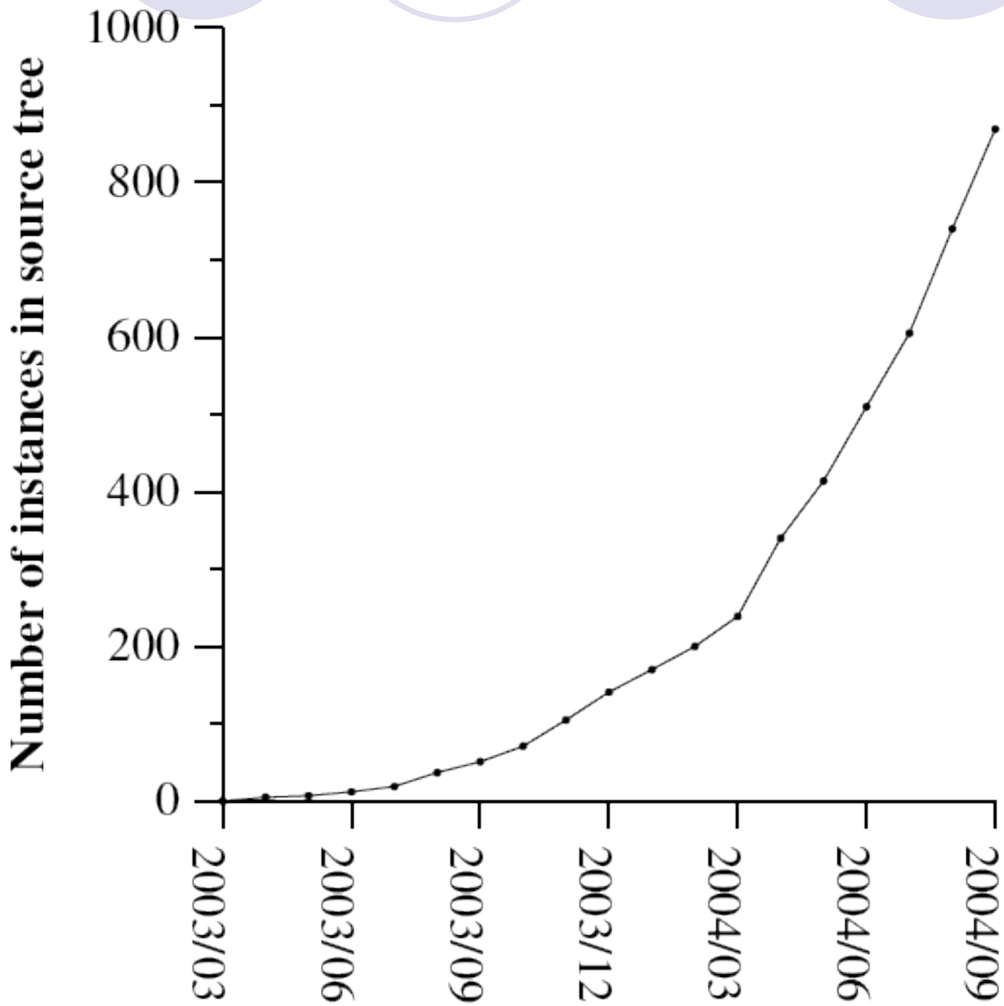
Fault Tolerance

- Crash of worker
  - all - even finished - tasks are redone
- Crash of leader
  - crash of leader process
    -> restart process with checkpoint
  - crash of leader machine
    -> unlikely - restart computation
  - redo computation

Software Fault Tolerance

- map and reduce might crash for certain records
  - often it is not possible to fix all bugs -> need to live with the bugs
  - deterministic crashes prevent termination
  - when function crashes, it sends msg to master saying that it has crashed on certain record
  - master will give up to retry after crashing multiple times on some record

Usage of MapReduce @ Google [2]



Workload (August 2004) [2]

Number of jobs	29,423
Average job completion time	634 secs
Machine days used	79,186 days
Input data read	3,288 TB
Intermediate data produced	758 TB
Output data written	193 TB
Average worker machines per job	157
Average worker deaths per job	1.2
Average map tasks per job	3,351
Average reduce tasks per job	55
Unique *map* implementations	395
Unique *reduce* implementations	269
Unique *map/reduce* combinations	426