Parallel Databases

- Increase performance by performing operations in parallel

Parallel Architectures

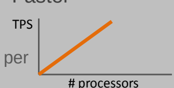
- Shared memory
- Shared disk
- Shared nothing

closely coupled

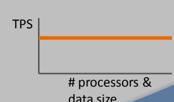
loosely coupled

Parallelism Terminology

- Speedup: More processors ➔ Faster
  If you increase the processors, how much faster is the system (typically measured in transactions per second)?



- Scaleup: More processors ➔ Process more data
  If you increase proportionally the number of processors and data, what is the performance of the system?



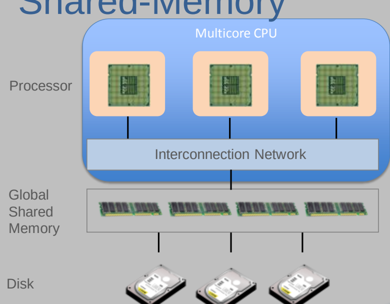
Challenges

For linear speedup and scaleup:

- Startup cost
  Cost to start processes
- Interference
  Contention for resources between processors
- Skew
  Different sized jobs between processors

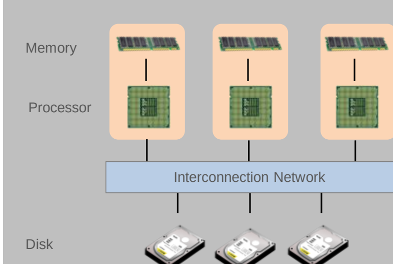
Parallel Architectures

Shared-Memory

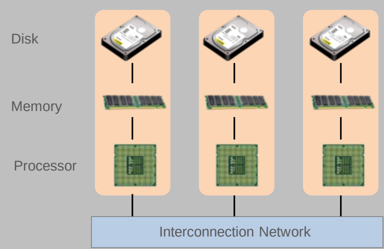


Parallel Architectures

Shared-Disk



Parallel Architectures
Shared-Nothing



Architecture Comparison

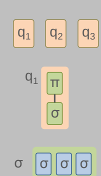
- Shared memory
  - ⊕ Good for load-balancing, easiest to program
  - ⊖ Memory contention: Does not scale
- Shared disk
  - ⊖ Disk contention: Does not scale
- ★ Shared nothing
  - ⊕ Scales linearly, can use commodity H/W
  - ⊖ Hardest to program

GREENPLUM
TERADATA

Types of Parallelism

In query evaluation:

- Inter-query parallelism
  Each query runs on one processor
- Inter-operator parallelism
  A query runs on multiple processors
  Each operator runs on one processor
- ★ Intra-operator parallelism
  An operator runs on multiple processors
  - ⊕ Most scalable



Horizontal Data Partitioning

Divide tuples of a relation among n nodes:

- Round robin
  Send tuple t_i to node [i mod n]
- Hash partitioning on an attribute C
  Send tuple t to node [h(t.C) mod n]
- Range partitioning on an attribute C
  Send tuple t to node i if $v_{i-1} < t.C < v_i$

Which is better? Let's see…

Parallel Operators

- Selection
- Group-By
- Join
- Sort
- Duplicate Elimination
- Projection

Parallel Selection

$\sigma_{A = v}$ or $\sigma_{v1 < A < v2}$

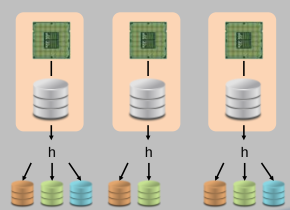
Done in parallel in:

- Round robin
  All nodes
- Hash partitioning
  One node for C = v
  All nodes for v1 < C < v2, all nodes for A = v
- Range partitioning
  All nodes whose range overlaps with the selection

Parallel Group By

$\gamma_{A;\ SUM(B)}$

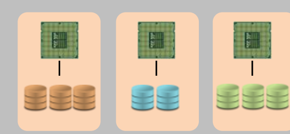
1) Each node partitions its data using hash function h(t.A)
to at most as many partitions as nodes



Parallel Group By

$\gamma_{A;\ SUM(B)}$

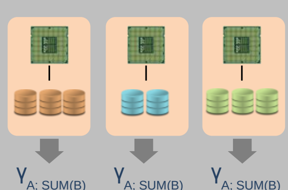
2) Each node sends each partition to one node
All tuples with the same group by value get collected in one node



Parallel Group By

$\gamma_{A;\ SUM(B)}$

3) Each node computes aggregate for its partition

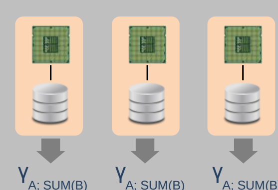


⊖ High communication cost (although << I/O cost)

Parallel Group By Optimization

$\gamma_{A;\ SUM(B)}$

0) Each node does local aggregation first
and then applies previous steps on the aggregation result



Parallel Join (Equi-join)

$R \bowtie_{R.A = S.B} S$

Follow similar logic to Group By:

- Partition data of R and S using hash functions
- Send partitions to corresponding nodes
- Compute join for each partition locally on each node

Parallel Join (General)

$R \bowtie_{R.A < S.B} S$

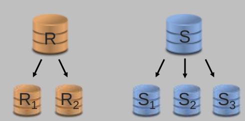
- Does the previous solution work?
- If no, why?

Parallel Join (General)

$R \bowtie_{R.A < S.B} S$

Use “fragment and replicate”:

1) Fragment both relations
   Partition R into m partitions & S into n partitions

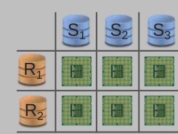


Parallel Join (General)

$R \bowtie_{R.A < S.B} S$

Use fragment and replicate:

2) Assign each possible combination of partitions to one node (requires m x n nodes)
   Each partition is replicated across many nodes



Parallel Sort

Use same partitioning idea as in group by and equi-join:

- Partition data of R and S using range partitioning
- Send partitions to corresponding nodes
- Compute sort for each partition locally on each node

Note: Different partitioning techniques suitable for different operators

Partitioning Revisited

- Round robin
  - ⊕ Good for load-balancing
  - ⊖ Have to access always all the data
- Hash partitioning
  - ⊕ Good for load-balancing
  - ⊖ Works only for equality predicates
- Range partitioning
  - ⊕ Works for range queries
  - ⊖ May suffer from skew

Other Operators

Duplicate Elimination:

- Use sorting or
- Use partitioning (range or hash)

Projection (wo duplicate elimination):

- Independently at each node

Parallel Query Plans

- The same relational operators
- With special split and merge operators
  To handle data routing, buffering and flow control

Cost of Parallel Execution

Ideally:

- Parallel Cost = Sequential Cost / #of nodes

but

- Parallel evaluation introduces overhead
- The partitioning may be skewed

Cost of Parallel Execution

Parallel Cost = $T_{part} + T_{asm} + \max(T_1, \ldots, T_n)$

- T_{part}: Time to partition the data
- T_{asm}: Time to assemble the results
- T_i: Time to execute the operation at node i

Parallel Databases Review

- Parallel Architectures
- Ways to parallelize a database
- Partitioning Schemes
- Algorithms

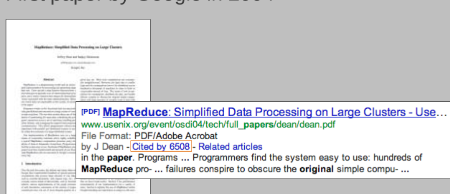
Shared-Nothing Parallelism and MapReduce

Large-scale data processing

- High-level programming model (API) &
- Corresponding implementation
- Don't worry about:
  - Parallelization
  - Data distribution
  - Load balancing
  - Fault tolerance

A little history

First paper by Google in 2004


[PDF] MapReduce: Simplified Data Processing on Large Clusters - Use...
www.usenix.org/event/osdi04/tech/full_papers/dean/dean.pdf
File Format: PDF/Adobe Acrobat
by J Dean - Cited by 6508 - Related articles
in the **paper**. Programs ... Programmers find the system easy to use: hundreds of **MapReduce** pro- ... failures conspire to obscure the **original** simple compu- ...

It was used to regenerate Google's index

A little history

- Apache Hadoop: Popular open-source implementation

hadoop

- Nowadays you can run hadoop without even setting up your own infrastructure

amazon web services Elastic MapReduce

MR Performance

How long does it take to sort 9TB of data on 900 nodes?
What about 20TB on 2000 nodes?

From Hadoop FAQs:

1.3. How well does Hadoop scale?

Hadoop has been demonstrated on clusters of up to 4000 nodes. Sort performance on 900 nodes is good (sorting 9TB of data on 900 nodes takes around 1.8 hours) and improving using these non-default configuration values:

Sort performances on 1400 nodes and 2000 nodes are pretty good too - sorting 14TB of data on a 1400-node cluster takes 2.2 hours; sorting 20TB on a 2000-node cluster takes 2.5 hours. The updates to the above configuration being:

Anatomy of a MR Program

- Input:
  bag of (input key, value) pairs
- Output:
  bag of output values
- Structure:
  Map function
  Reduce function

Map function

- Input:
  (input key, value) pair
- Output:
  (intermediate key, value) pair
- Semantics:
  System applies the function in parallel to all (input key, value) pairs in the input

Reduce function

- Input:
  (intermediate key, bag of values) pair
- Output:
  bag of output values
- Semantics:
  System groups all pairs with the same intermediate key and passes the bag of values to the Reduce function

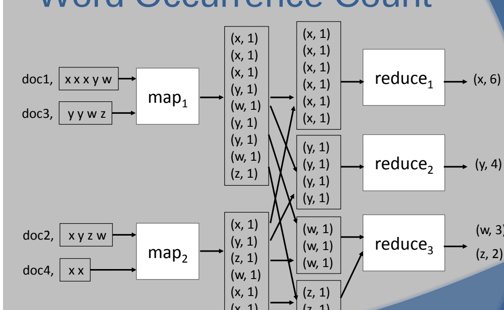
Example 1
Word Occurrence Count

For each word in a set of documents, compute the number of its occurrences in the entire set

```
map (String key, String value) {
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1"); }

reduce (String key, Iterator values) {
    // key: a word
    // value: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    EmitString(key, AsString(result));}
```

Example 1
Word Occurrence Count



Example 1

Word Occurrence Count

- Do you see any way to improve this solution?

Example 1

Word Occurrence Count

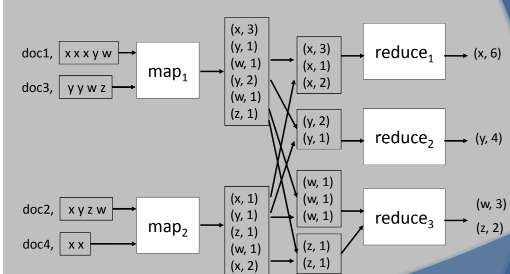
Improve the MR program by reducing the number of intermediate pairs produced by each mapper:

```
map (String key, String value) {
    // key: document name
    // value: document contents
    H = new AssociativeArray;
    for each word w in value:
        H(w) += 1;
    for each word w in H:
        EmitIntermediate(w, H(w)); }
```

Each mapper sums up the counts for each document

Example 1

Word Occurrence Count



Example 1

Word Occurrence Count

- Do you see any other possibility to improve performance?

Introducing the Combiner

- Combiner:
  Combine values output by each Mapper (since they already exist in main memory). Similar to an intermediate reduce for each individual Mapper.
- Input:
  bag of (intermediate key, bag of values) pairs
- Output:
  bag of (intermediate key, bag of values) pairs
- Semantics:
  System groups for each mapper separately "all" pairs with the same intermediate key and passes the bag of values to the Combiner function

Example 1

Word Occurrence Count

Improve the MR program by utilizing a combiner:

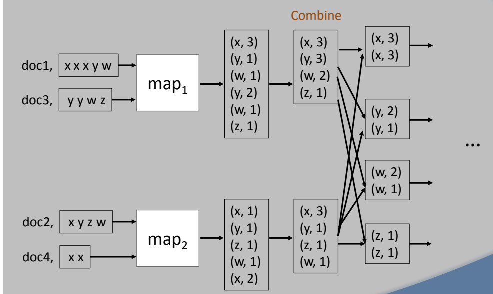
```
combine(String key, Iterator values) {
    // key: word
    // value: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    EmitString(key, AsString(result));}
```

Does it look familiar?

It has the same code as the reduce function!

Example 1

Word Occurrence Count



More examples

- Sorting
- Selection
- Join

More examples

- Sorting
  Leverage the fact that data are sorted before being pushed to the reducers and also reducers themselves are sorted
  Map: (k, v) ➔ (v, _)
  Reduce: (v, _) ➔ v

More examples

- Join
  Hash on join attribute
  Have to encode the relation name as well
  Map: (table, bag of tuples) ➔ (join attribute, (tuple, relName))
  Reduce: (join attribute, (tuple, relName)) ➔ output tuple
- Other possibility?
- Join on mappers
  If one of the relations fits in main memory

Implementation

- One Master node: Scheduler & Coordinator
- Many Workers: Servers for Map/Reduce tasks

Implementation

- Map Phase (need M workers)
  - Master partitions input file into M splits, by key
  - Master assigns workers to the M map tasks
  - Workers execute the map task, write their output to local disk, partitioned into R regions according to some hash function
- Reduce Phase (need R workers)
  - Master assigns workers to the R reduce tasks
  - Each worker reads data from the corresponding mapper's disk, groups it by key and execute the reduce function

Implementation

- Filesystem
  - Input & Output of MapReduce are stored in a distributed file system
  - Each file is partitioned into chunks
  - Each chunk is replicated on multiple machines
  - Implementations:
    GFS (Google File System): Proprietary
    HDFS (Hadoop File System): Open source

Some more details

- Fault Tolerance
  - Master pings each worker periodically
  - If it does not reply, tasks assigned to it are reset to their initial state and rescheduled on other workers

Some more details

- Tuning
  - Developer has to specify M & R:
    M: # of map tasks
    R: # of reduce tasks
  - Larger values: Better load balancing
  - Limit: Master need O(M x R) memory
  - Also specify: 100 other parameters (50 of which affect runtime significantly)
  - Automatic tuning?

MR: the downsides

- Problems
  - Batch oriented: Not suited for real-time processes
  - A phase (e.g., Reduce) has to wait for the previous phase (e.g., Map) to complete
  - Can suffer from stragglers: workers taking a long time to complete
  - Data Model is extremely simple for databases: Not everything is a flat file
  - Tuning is hard