

Lecture 16 -- Parallel & Distributed Databases

Parallel/distributed databases: goal provide exactly the same API (SQL) and abstractions (relational tables), but partition data across a bunch of machines -- let us store more data and process it faster.

Parallel refers a single multi-processor machine, or a cluster of machines. Distributed typically refers to multiple machines that can fail independently.

Huge market -- essentially all high performance databases work this way

Some notes:

No "administrative boundaries" in parallel

All sites cooperative in parallel

Parallel machines don't need to be as tolerant to failures as distributed machines

One way to solve this is to build custom-built parallel machines.

Many special purpose parallel architectures have failed. Why?

prohibitively expensive (no economy of scale)

slow to evolve

requires a tool set (i.e., compilers, OS, etc.) -- painful to build from scratch!

Increasingly parallelism is achieved through software running on commodity HW

Performance metrics:

Speed up = $\frac{\text{oldtime}}{\text{newtime}}$ for a given problem

Scale up = $\frac{\text{small system runtime on small problem}}{\text{large system runtime on large problem}}$

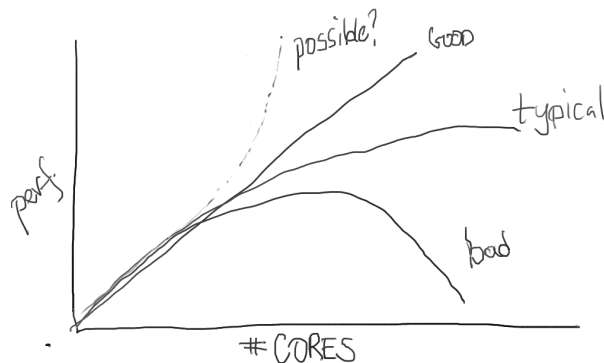
Not necessarily identical -- a small problem may be harder to parallelize.

Transaction scaleup: N times as many TPC-C's for N machines

Batch scaleup : N times as big a query for N machines

What kind of speedups are we looking for?

Linear! (say that scaleup = 1 is "linear")



What are the key properties of a parallelizable workload?

Illustrates linear speedup.

Can be decomposed into small units that can be executed independently
"embarrassingly parallel"

What are the barriers to linear speedup:

Startup times (e.g., process per parallel operation may be a bad idea)

Interference (processors depend on some shared resource)

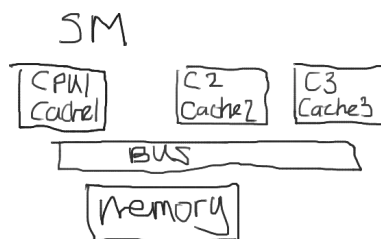
Skew (workload not of equal size on each processor)

Consider barriers in both transactional and analytic workloads

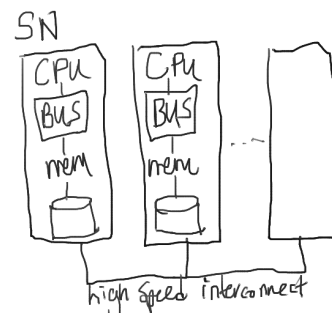
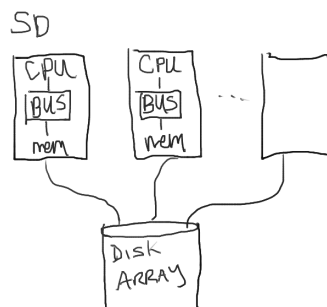
Suppose we want to use multiple machines -- what are possible parallel architectures

Shared memory, shared disk, shared nothing.

Show design.



Cache coherence
protocols arbitrate
concurrent accesses to
main memory



Shared Memory:

- + easy to program
- + no changes to CC+R
- performance/scalability
- fault tolerance

Shared Disk:

- + better scalability
- + better fault tolerance
- complex cache coherency
- not very scalable

Shared Nothing (partitioned data):

- + cost
- + scalability
- + fault tolerance
- new CC+R
- new executor
- maintenance

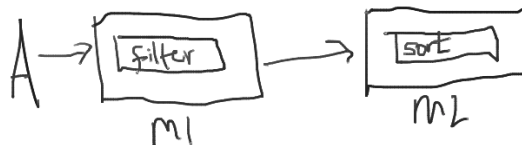
What about a system like HDFS as the storage substrate? Can be used like SN or SD, and inherits properties of either.

How does a II DBMS provide linear scaleup in performance of a single query?

2 Types of parallelism:

Pipelined

Sequence of operators, each running on a different processor
output of nth stage used as input into n+1st stage. Diagram:



Why is pipelined parallelism hard to exploit?

Only works when each pipeline stage is about the same speed

Short pipelines

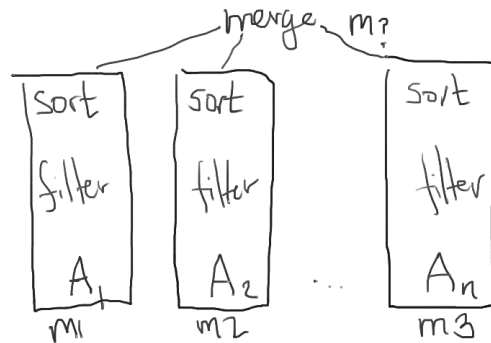
Inputs to stage i+1 depend on stage i

If stage i blocks (i.e., sorts), breaks pipeline

Partitioned

Identical subproblems each running on a subset of the data on a different processor

Diagram:



How should we partition data?

Three types of partitioning

	<u>Pros</u>	<u>Cons</u>
Round-robin	Perfect load balancing	All nodes process associative queries (e.g., <code>ename = 'Smith'</code>)
Hash	Good load balancing Better associativity than RR	Clustering, range queries (e.g., <code>sal > 2,000,000</code>)
Range	Good sequential + associative perf. Good Clustering	Load balancing problems (Hot spots) -- skew

Typically, partitioning is specified by the database administrator apriori (e.g., data is pre-partitioned in some way prior to the query running.)

How to fix range partitioning problems?

Use different size partitions based on their popularity

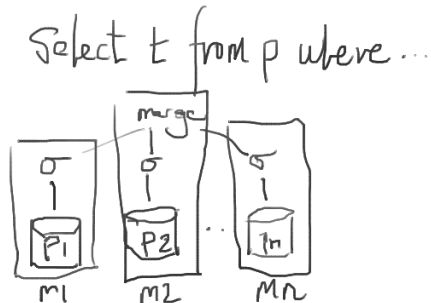
Or create many small partitions (hash buckets) and balance those

Sometimes we can answer a query with data from just 1 site, depending on partitioning used. Often, however, we will have to combine data from many sites. How do we do that?

Which operators can be partitioned?

Scans, selections, projections, aggregates, joins

Need specialized operators — split & merge — to make this work
Selection Example



Joins are a little tricky -- must ensure that all tuples that could join see each other if we want to parallelize.

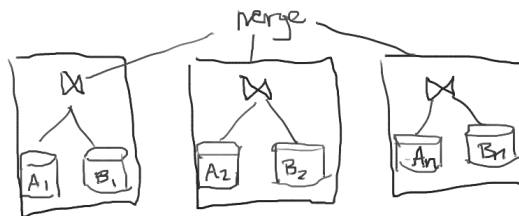
Example: partitioned on join attribute

SELECT * FROM A,B WHERE A.a = B.b

n machines, hash fxn H

Hash A on a, $H(a) \rightarrow 1..n$

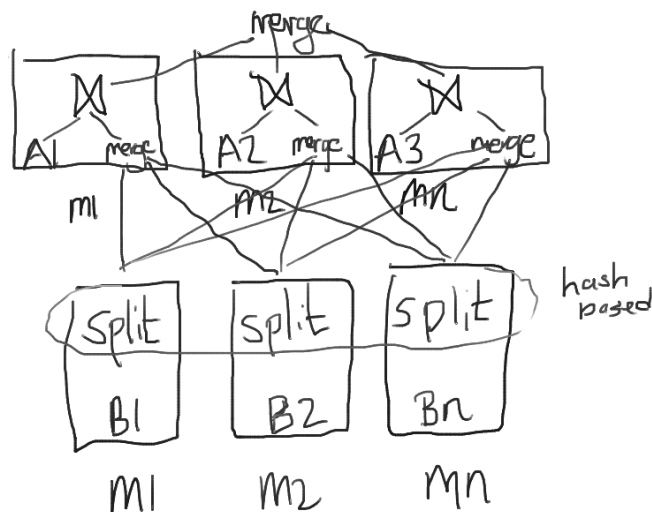
Hash B on b, $B(b) \rightarrow 1..n$



Example: not partitioned on join attribute

Have to repartition one of the tables

Hash B on c, $H(c) \rightarrow 1..n$



Can choose to repart A, repart B, or both.

Best choice depends on sizes of relations.

Other options for joins --

1) replicate small tables on all nodes -- avoid repartitioning altogether, if tables fit

2) "semijoin":

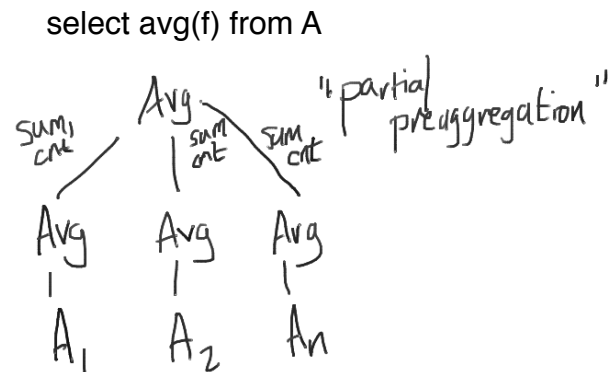
send list of all values in each partition of B to A,
then send list of matching tuples from A to B,
then compute join at B.

Good for selective joins of wide tables.

Aggregation:

Can run aggregates in parallel, then merge groups.

Example:



Standard way of expressing aggregates:

INIT
MERGE
FINAL

Some other issues:

Scheduling -- what if one machine runs way ahead of another

Fault tolerance -- what to do if one machine fails

Transactions

...

Summary:

Databases workloads are "embarrassingly parallelizable" -- one of the great advantages of the relational algebra.

Qs:

Is it always good to parallelize?

(No, not if there is a high startup cost).

Suppose we are running lots of little transactions, each of which does very little work on its own piece of the database? What is optimal partitioning strategy then?

(Partition according to pieces transactions operate on, so each can run in parallel.)

I heard that databases don't scale, is that true?

(No, not really. The workloads parallelize just fine -- extremely well, in fact, in most cases. In the wide area ("internet scale") making transactions work can be tricky, as we'll discuss next time.)

Some Themes in Parallel DBs

(that distinguish them from other parallel programming tasks):

o Hooray for the relational model

- apps don't change when you parallelize system (physical data independence!). can tune, scale system without informing apps too
- ability to partition records arbitrarily, w/o synchronization

o essentially no synchronization except setup & teardown

- no barriers, cache coherence, etc.
- DB transactions work fine in parallel

- + data updated in place, with 2-phase locking transactions
- + replicas managed only at EOT via 2-phase commit (next lecture)
- + coarser grain, higher overhead than cache coherency stuff

o Bandwidth much more important than latency (in analytics at least)

- often pump 1-1/n % of a table through the network
- aggregate net BW should match aggregate disk BW
- bus BW should match about 3x disk BW (NW send, NW receive, disk)
- Latency, schmatency. Insignificance makes a BIG difference in what architectures are needed.