Parallel Databases

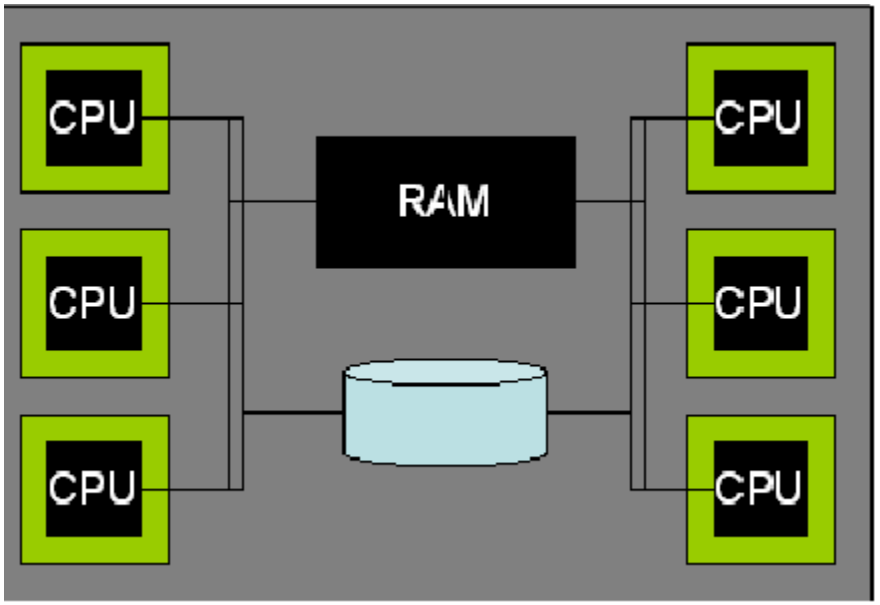
Introduction

- Databases are growing increasingly large
  - “Big Data” is a huge industry buzzword
  - Data considered a competitive advantage
    - Collect as much as possible
    - Use data to make decisions
    - Machine learning models train better with more data
- Single server with single processor not viable any more
  - Need multiple processors
  - Need large amounts of memory
  - Need more disks
- Want to benefit from these additional resources through parallelizing work across them

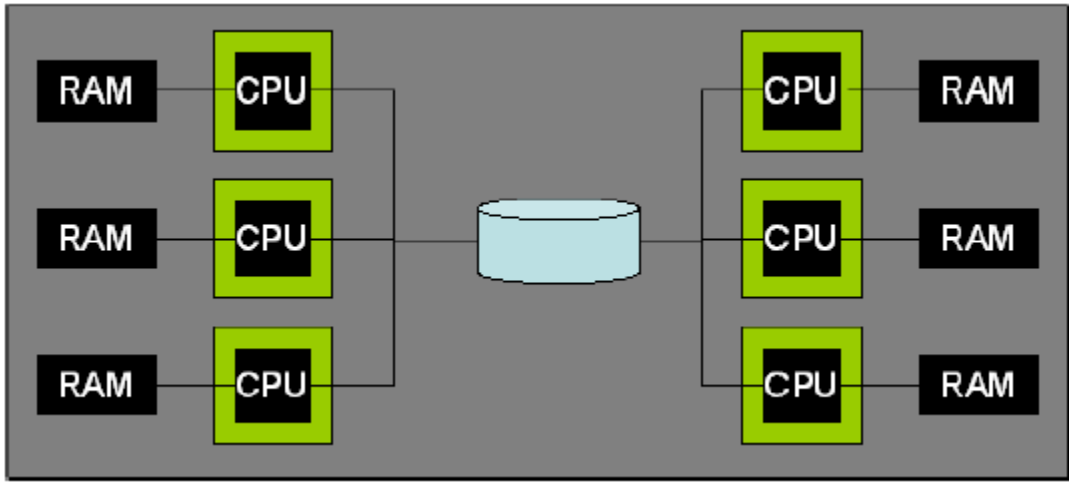
Parallelism in Databases

- For a single query / transaction (xact):
  - CPU work can be partitioned across multiple processors
  - Data can be partitioned across multiple disks for parallel I/O
  - Both of the above can make the query go faster
- For multiple queries / xacts
  - Different processors can work on different queries / xacts
    - Improves system throughput (and also latency because less waiting)
- User does not need to be aware of parallel execution of queries in the system
    - Same interface (SQL)
    - Same final result
    - But better performance

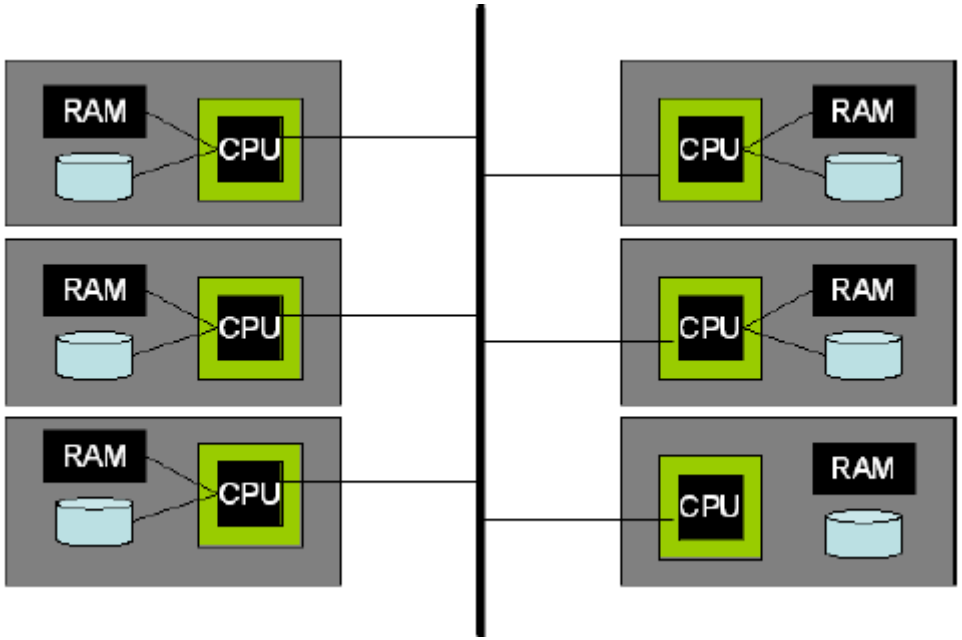
Shared-Memory



Shared-Disk



Shared-Nothing



I/O Parallelism

- Divide tables across multiple disks
  - Seek / rotate disks in parallel to multiply data read per unit time
- Several ways to partition tables
  - Horizontally (different rows on different disks)
    - Most common
  - Vertically (different columns on different disks)
    - Covered in CMSC 624 / 828N
- Partitioning techniques (number of disks = n):
  - **Round robin**
    - Send the *I* th tuple inserted in the relation to disk *i* mod *n*.
  - **Hash partitioning**
    - Choose one (or more) attribute as the partitioning attribute, p.
    - Choose hash function *h* with range 0…*n* - 1
    - For each tuple, t, apply *h* to t.p. Send t to corresponding disk.

I/O Parallelism (Cont.)

- **Range partitioning:**
  - Choose an attribute as the partitioning attribute, p.
  - Divide p into n disjoint ranges that cover the entire domain of p
  - Ideally, choose ranges such that equal % of p in each range
  - Allocate one range per disk
  - For each tuple t, place t on the disk which owns the range in which t.p is located.
  - For example, 26 disks, partitioning attribute is person.name
    - One disk per letter of the alphabet that name starts with
      - But some disks (e.g. the one for ‘X’) would be nearly empty
      - So likely better off using 26 ranges based on the first 2-3 letters instead of just the first

Parallelizing Where Clause Selections

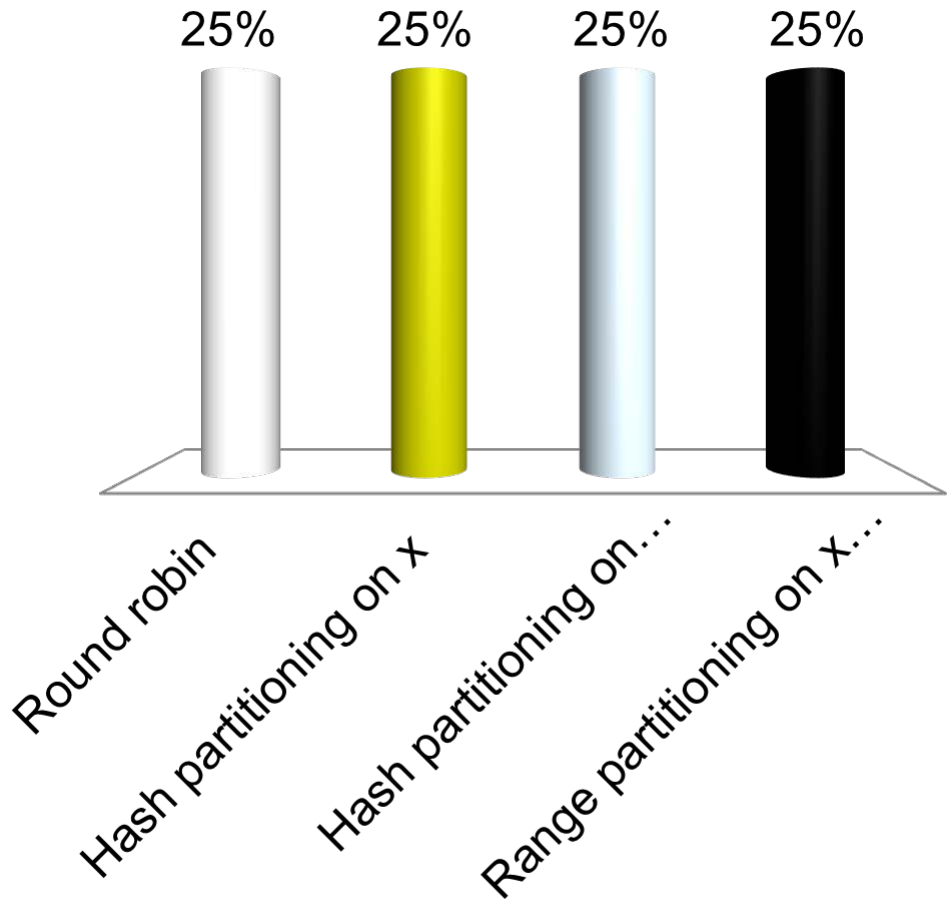
SELECT * FROM table WHERE x = 10

SELECT * FROM table WHERE x > 20

- Assume each disk has own processor (E.g. a shared-nothing architecture)
  - If data partitioned evenly across disks:
    - Each processor (at the same time / in parallel) reads the partition on its disk, searches for x = 10 or x > 20
      - If there are n disks, then each disk has 1/nth of the data
        - So search happens n times faster than if only one disk
    - Results found by each processor assembled at end of the query
      - This cost assumed to be small relative to search cost

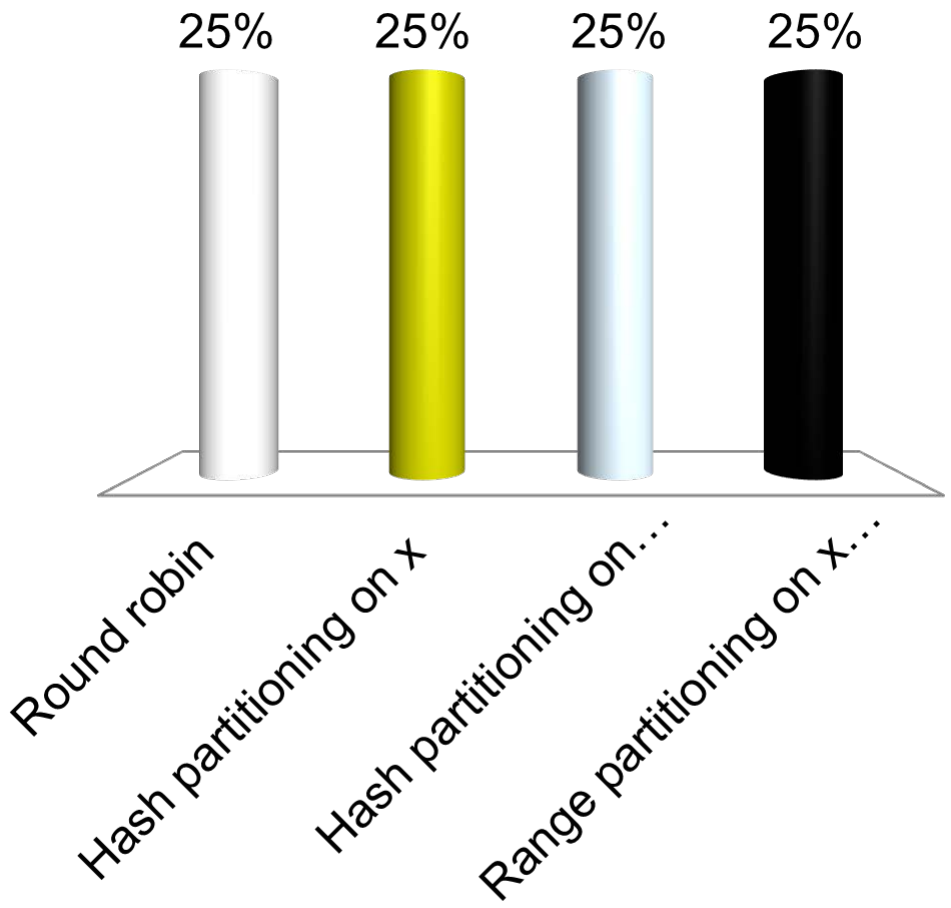
What partitioning scheme is most likely to fall short of being n times faster for SELECT * FROM table WHERE x = 10

A. Round robin
B. Hash partitioning on x
C. Hash partitioning on some other attribute
D. Range partitioning on x or some other attribute



What partitioning scheme is least likely to fall short of being n times faster for SELECT * FROM table WHERE x = 10

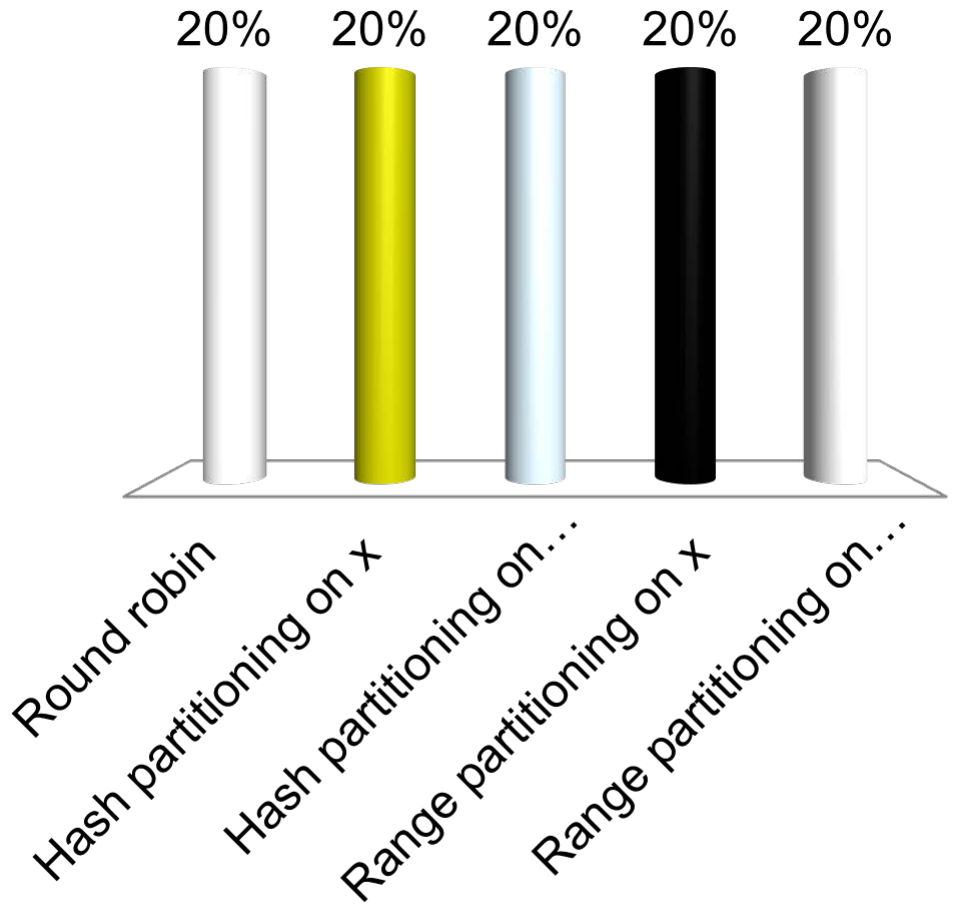
A. Round robin
B. Hash partitioning on x
C. Hash partitioning on some other attribute
D. Range partitioning on x or some other attribute



Note: we are for now ignoring start-up cost and machine skew which are discussed on later slides. If we considered those, choice B is also a viable answer (and maybe even a better one)

What partitioning scheme should be used if every query looks like: SELECT * FROM table WHERE x = <const>

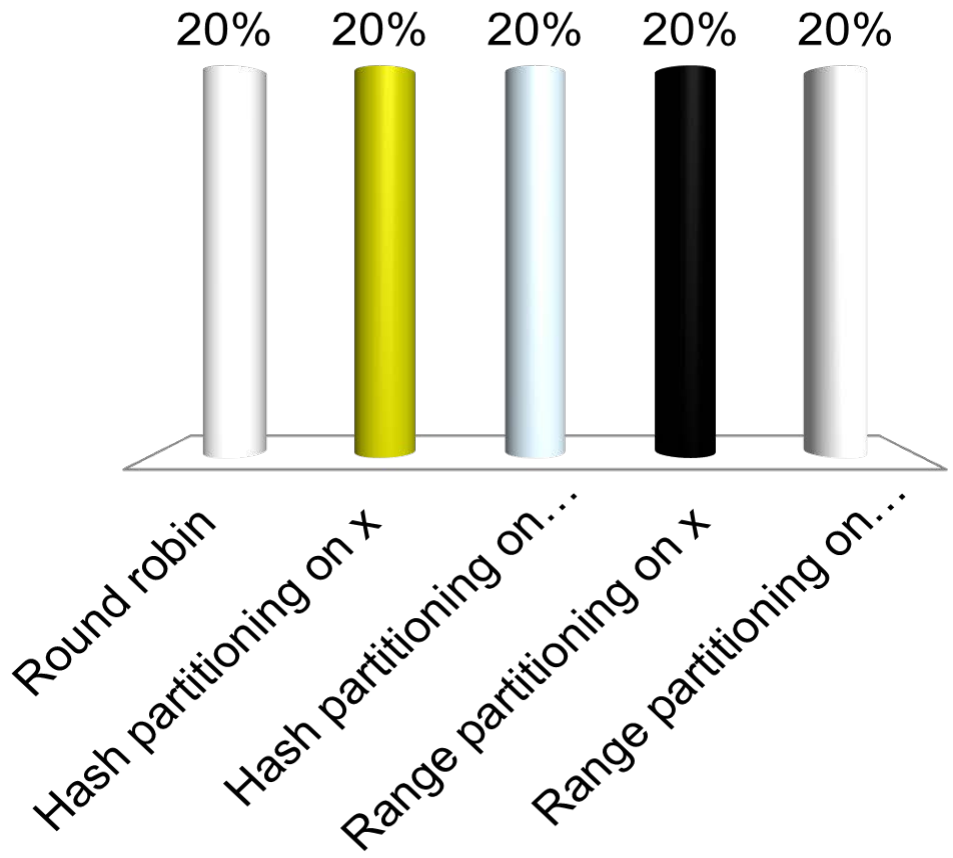
A. Round robin
B. Hash partitioning on x
C. Hash partitioning on some other attribute
D. Range partitioning on x
E. Range partitioning on some other attribute



Choice B allows most of cluster to remain idle, so they can work on other queries.

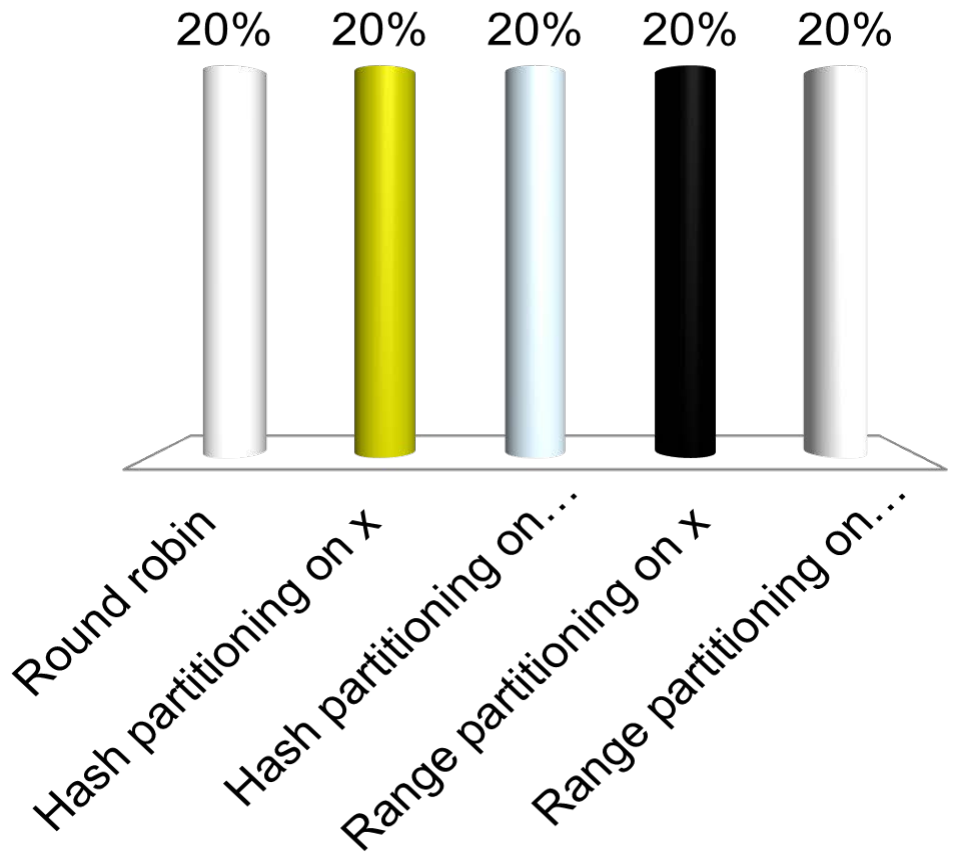
What partitioning to use. Every query looks like: SELECT * FROM table WHERE x = <const> OR SELECT * FROM table WHERE x > <const>

A. Round robin
B. Hash partitioning on x
C. Hash partitioning on some other attribute
D. Range partitioning on x
E. Range partitioning on some other attribute



What partitioning to use. Every query looks like: SELECT x FROM table; OR SELECT agg_func(x) FROM table;

A. Round robin
B. Hash partitioning on x
C. Hash partitioning on some other attribute
D. Range partitioning on x
E. Range partitioning on some other attribute



Comparison of Partitioning Techniques (Cont.)

Round robin:

- Advantages
  - Best suited for sequential scan of entire relation on each query.
  - All disks have almost an equal number of tuples; retrieval work is thus well balanced between disks.
- Disadvantages
  - All disks have to get involved for point (=x) and range queries

Comparison of Partitioning Techniques (Cont.)

Hash partitioning:

- Usually good for sequential access
  - Assuming hash function is good, and partitioning attribute(s) is a key, tuples will be equally distributed between disks
    - But value skew of partitioning attribute can cause problems
  - Retrieval work is then well balanced between disks.
- Good for point queries on partitioning attribute
  - Only one disk gets involved, leaving others available for answering other queries.
- BUT: all disks have to get involved for range queries

Comparison of Partitioning Techniques (Cont.)

Range partitioning:

- OK for sequential access
  - But skew more likely to be issue
- Good for point queries on partitioning attribute: only one disk needs to be accessed.
- For range queries on partitioning attribute, one to a few disks may need to be accessed
  - Remaining disks are available for other queries.

Partitioning a Relation across Disks

- If a relation contains only a few tuples which will fit into a single disk block, then assign the relation to a single disk.
- Large relations are preferably partitioned across all the available disks.
- If a relation consists of *m* disk blocks and there are *n* disks available in the system, then the relation should be allocated **min**(*m,n*) disks.

Handling of Skew

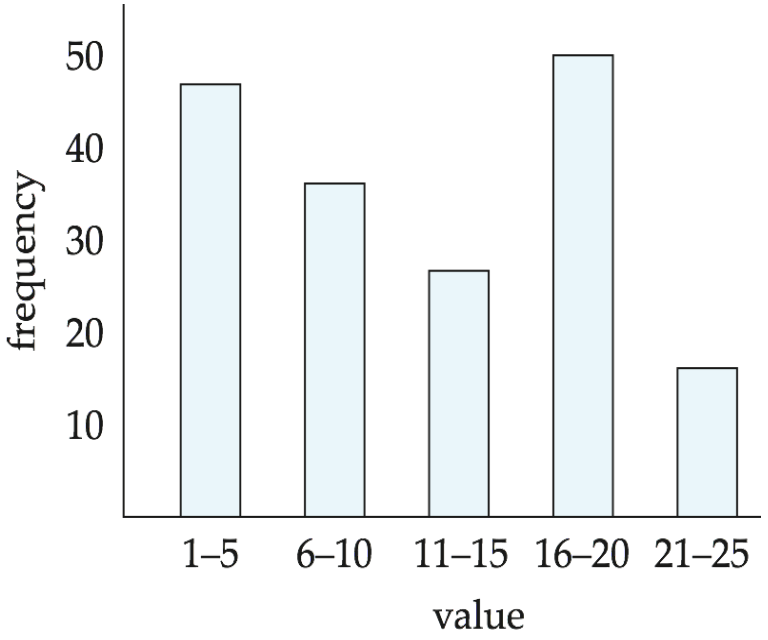
- The distribution of tuples to disks may be **skewed** — that is, some disks have many tuples, while others may have fewer tuples.
- **Types of skew:**
  - **Attribute-value skew**.
    - Some values for partitioning attribute(s) appear frequently; all the tuples with that same value end up in the same partition.
    - Can occur with range-partitioning and hash-partitioning.
  - **Partition skew**.
    - Arises from bad partitioning functions
      - Most likely happens for range partitioning

Handling Skew in Range-Partitioning

- To create a **balanced range partitioning function:**
  - Sort the relation on the partitioning attribute.
  - Construct the partition vector by scanning the relation in sorted order as follows.
    - After every $1/n^{th}$ of the relation has been read, the next new value of the partitioning attribute is the beginning of the next range
  - *n* denotes the number of partitions to be constructed.
  - Duplicate entries or imbalances can result if duplicates are present in partitioning attributes.
- Alternative technique based on **histograms** is faster, but less precise (see next slide)

Handling Skew using Histograms

- Create logical buckets corresponding to m ranges that span domain of p (m >> n)
- Allocate one counter per bucket, initialize all counters to 0
- Sample or scan range partitioning attribute p
  - For every tuple t from sample/scan, add one to the bucket counter in which t.p is located
  - At the end, you have a histogram
- Assume uniform distribution within each range of the histogram
  - Use histogram to create n ranges of approximately same size



Handling Skew Using Virtual Processor Partitioning

- Third option: **virtual processor partitioning**:
  - Create a large number of partitions (say 10 to 20 times the number of processors)
  - Assign virtual processors to partitions either in round-robin fashion or based on hash function or mapping of virtual partition ids
- Basic idea:
  - If any normal partition would have been skewed, it is very likely the skew is spread over a number of virtual partitions
  - Skewed virtual partitions get spread across a number of processors, so work gets distributed evenly!
    - If not, can move virtual partitions around
      - Requires changing hash function or mapping

Intraquery Parallelism

- Execution of a single query in parallel on multiple processors/disks; important for speeding up long-running queries.
- Two complementary forms of intraquery parallelism:
  - **Intraoperation Parallelism** – parallelize the execution of each individual operation in the query.
  - **Interoperation Parallelism** – execute the different operations in a query expression in parallel.
- Intraoperation scales better
  - Number of tuples processed by each operation is typically more than number of operations in a query.

Parallel Processing of Relational Operations

- Assume *n* processors, P_0, ..., P_{n-1}, and *n* disks D_0, ..., D_{n-1}, where disk D_i is associated with processor P_i.
  - Assumption works best in shared-nothing architecture, but:
    - If a processor has multiple disks they can simply simulate a single disk D_i.
    - If m processors per disk, can logically divide disk into D_1, ..., D_m
  - Thus, can work on shared memory and shared disk as well

Parallel Sort

Range-Partitioning Sort

- Choose processors P_0, ..., P_m, where $m \leq n$ -1 to do sorting.
- Create range-partition vector with m entries, on the sorting attributes
- Redistribute the relation using range partitioning
  - all tuples that lie in the i^{th} range are sent to processor P_i
  - P_i stores the tuples it received temporarily on disk D_i.
  - This step requires I/O and communication overhead.
- Each processor P_i sorts its partition of the relation locally.
- Each processors executes same operation (sort) in parallel with other processors, without any interaction with the others (**data parallelism**).
- Final merge operation is trivial: range-partitioning ensures that, for 1 j m, the key values in processor P^i are all less than the key values in P_j.

Parallel Sort (Cont.)

Parallel External Sort-Merge

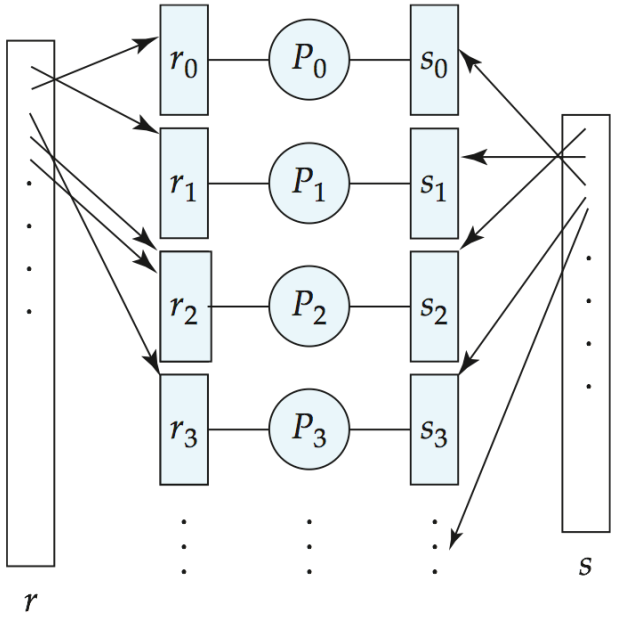
- Assume the relation has already been partitioned among disks D_0, ..., D_{n-1} (in whatever manner).
- Each processor P_i locally sorts the data on disk D_i.
- The sorted runs on each processor are then merged to get the final sorted output.
- Parallelize the merging of sorted runs as follows:
  - The sorted partitions at each processor P_i are range-partitioned across the processors P_0, ..., P_{m-1}.
  - Each processor P_i performs a merge on the streams as they are received, to get a single sorted run.
  - The sorted runs on processors P_0,..., P_{m-1} are concatenated to get the final result.

Parallel Join

- The join operation requires pairs of tuples to be tested to see if they satisfy the join condition, and if they do, the pair is added to the join output.
- Parallel join algorithms attempt to split the pairs to be tested over several processors. Each processor then computes part of the join locally.
- In a final step, the results from each processor can be collected together to produce the final result.

Partitioned Join

- For equi-joins and natural joins only
- Let *r* and *s* be the input relations, and we want to compute $r \bowtie s$.
- *Partition* r and s across the processors; compute join locally at each processor.
  - *r* and *s* each are partitioned into *n* partitions, denoted $r_0, r_1, ..., r_{n-1}$ and $s_0, s_1, ..., s_{n-1}$.
- Can use either *range partitioning* or *hash partitioning* **on the join attributes**
  - Must use same function for both r and s
- Partitions r_i and s_i are sent to processor P_i,
- Each processor P_i locally computes $r_i \bowtie s_i$.
  - Any join method can be used.

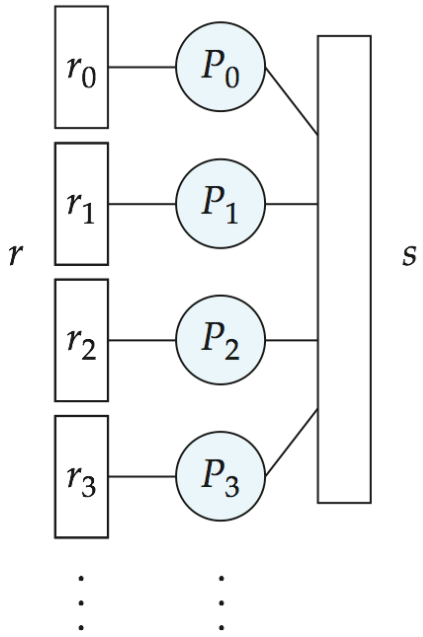


Partitioned Parallel Hash-Join

- Special case of Partitioned join where
  - Partitions are created via a hash function, h_1
  - Join is done via a local hash join (use different hash function, h_2)

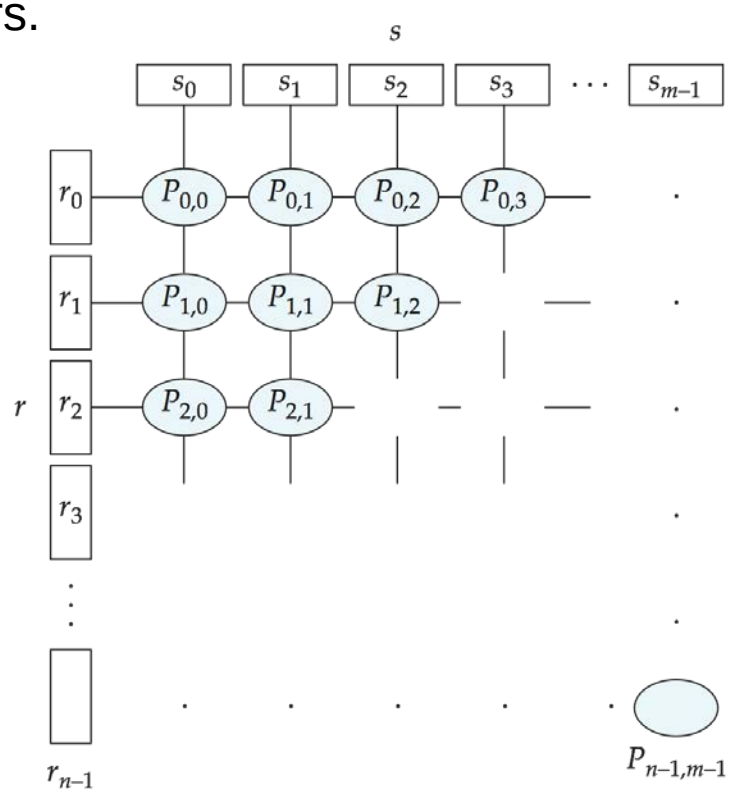
Fragment-and-Replicate Join

- Partitioning not possible for some join conditions
  - E.g., non-equijoin conditions, such as Query 10 from Project 1 (r.A > s.A)
- Need to compare every tuple from r with every tuple from s
  - But want to parallelize by dividing this comparison work across processors
- Choice 1: Broadcast join (book calls it **asymmetric fragment-and-replicate**)
  - One of the relations, say *r*, is partitioned; any partitioning technique can be used.
    - Usually already partitioned prior to storage; if so, skip this step
  - The other relation, *s*, is **replicated** across all the processors.
  - Processor P_i locally computes the join of r_i with all of s using any join technique.
  - Very popular when $b_r >> b_s$



Fragment-and-Replicate Join (Cont.)

- Choice 2 (General case):
  - *r* is partitioned into n partitions,r_0, r_1, ..., r_{n-1};
  - s is partitioned into *m* partitions, s_0, s_1, ..., s_{m-1}.
  - Any partitioning technique may be used.
  - There must be at least m * n processors.
    - $P_{0,0}$, $P_{0,1}$, ..., $P_{0,m-1}$, $P_{1,0}$, ..., P_{n-1m-1}.
    - r_i is replicated to $P_{i,0}$, $P_{i,1}$, ..., $P_{i,m-1}$,
    - s_i is replicated to $P_{0,i}$, $P_{1,i}$, ..., $P_{n-1,i}$
  - $P_{i,j}$ computes the join of r_i with s_j.
    - Any join method can be used



Which Parallel Join to use?

- If non-equi join, partitioned join not an option
- If equi-join
  - If data is not already partitioned by join attribute
    - Partitioned join repartitions both tables
      - (Almost) entirety of both tables must be sent over network
    - Fragment-and-replicate join usually only repartitions one table
      - But other one must be repartitioned **and** replicated n times
    - Partition/replication cost often performance bottleneck
      - So calculating data transfer size for each technique is important

Parallel (Index) Nested-Loop Join

- Special case of broadcast join (asymmetric fragment-and-replicate)
- Assume that
  - relation *s* is much smaller than relation *r*
  - *r* is already partitioned.
- S gets replicated to every partition of r
- As each block of S arrives, keep it in memory
  - For each tuple of that block, scan the local partition of r
    - So partition of r is the inner table, s is the outer table
  - If we have an index on local partition of r, use that instead of scan
  - If local partition of r is larger than memory, and we don't have an index on r, use local block nested-loops join instead

Other Relational Operations

Selection $\sigma_{\theta}(r)$

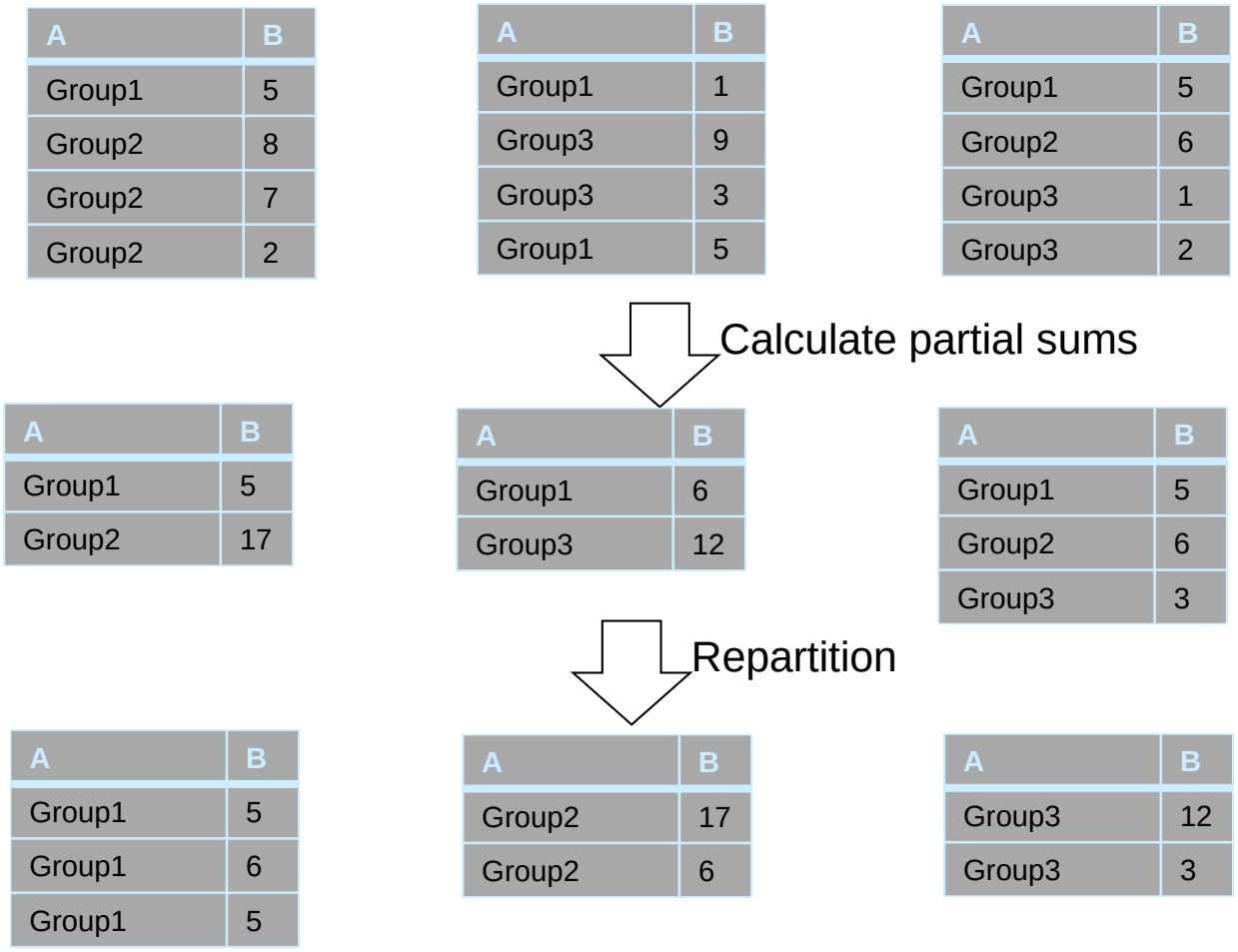
- If θ is of the form $a_i = v$, where a_i is an attribute and v a value.
  - If r is partitioned on a_i the selection is performed at a single processor.
- If θ is of the form $l <= a_i <= u$ (i.e., θ is a range selection) and the relation has been range-partitioned on a_i
  - Selection is performed at each processor whose partition overlaps with the specified range of values.
- In all other cases: the selection is performed in parallel at all the processors.

Other Relational Operations (Cont.)

- Duplicate elimination
  - Perform by using either of the parallel sort techniques
    - eliminate duplicates as soon as they are found during sorting.
  - Can also partition the tuples (using either range- or hash-partitioning) and perform duplicate elimination locally at each processor.
- Projection
  - Projection without duplicate elimination can be performed as tuples are read in from disk in parallel.
  - If duplicate elimination is required, any of the above duplicate elimination techniques can be used.

Grouping/Aggregation

- Partition the relation on the grouping attributes and then compute the aggregate values locally at each processor.
- Optimization: computer partial aggregate values before partitioning.
  - Fewer tuples need to be sent to other processors during partitioning.



Cost of Parallel Evaluation of Operations

- Best case scenario for parallel operators is 1/n speed-up over non-parallel version, but usually fall slightly short:
  - Start-up costs to start to operator on multiple processors
  - A little skew is common
    - Data skew we discussed already
    - Processing skew also issue --- some machines unexpectedly slow
  - Network can become bottleneck
- Cost of parallel operation can be estimated as

$$T_{start} + T_{part} + T_{asm} + \max(T_0, T_1, \ldots, T_{n-1})$$

  - T_{start} is the time for partitioning the relations – small, often ignored
  - T_{part} is the time for (re)partitioning the relations
  - T_{asm} is the time for assembling the results – usually small, often ignored
  - T_i is the time taken for the operation at processor P_i

Interoperator Parallelism

- **Pipelined parallelism**
  - Consider a join of four relations
    - $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$
  - Set up a pipeline that computes the three joins in parallel
    - Let P1 be assigned the computation of
      temp1 = $r_1 \bowtie r_2$
    - And P2 be assigned the computation of temp2 = temp1 $\bowtie$ r_3
    - And P3 be assigned the computation of temp2 $\bowtie$ r_4
  - Each of these operations can execute in parallel, sending result tuples it computes to the next operation even as it is computing further results
    - Provided a pipelineable join evaluation algorithm (e.g., indexed nested loops join) is used

Factors Limiting Utility of Pipeline Parallelism

- Pipeline parallelism is useful since it avoids writing intermediate results to disk
- Useful with small number of processors, but does not scale up well with more processors. One reason is that pipeline chains do not attain sufficient length.
- Cannot pipeline operators which do not produce output until all inputs have been accessed (e.g., aggregate and sort)
- Little speedup is obtained for the frequent cases of skew in which one operator's execution cost is much higher than the others.

Independent Parallelism

- **Independent parallelism**
  - Consider a join of four relations

    $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$
    - Let P_1 be assigned the computation of temp1 = $r_1 \bowtie r_2$
    - And P_2 be assigned the computation of temp2 = $r_3 \bowtie r_4$
    - And P_3 be assigned the computation of temp1 $\bowtie$ $temp_2$
    - P_1 and P_2 can work **independently in parallel**
    - P_3 has to wait for input from P_1 and P_2
      - Can pipeline output of P_1 and P_2 to P_3, combining independent parallelism and pipelined parallelism
  - Does not provide a high degree of parallelism
    - useful with a lower degree of parallelism.
    - less useful in a highly parallel system.

MapReduce: Simplified Data Processing on Large Clusters

Motivation

- Large-Scale Data Processing
  - Want to use 1000s of machines
  - Read-oriented workload
  - Paper we read was published by Google
    - But open source version (Hadoop) not built by Google, very popular

- Scalable/parallel SQL queries one (of many) applications for MapReduce / Hadoop
  - Can convert SQL queries into native MapReduce jobs
    - Can be done by hand
    - There exist ~8-10 SQL parsers that do this automatically
      - HadoopDB, Hadapt, Hive, Impala (sort of), Spark SQL (sort of), etc.
  - Automatic parallelization & distribution across machines in cluster

Map/Reduce

- Map/Reduce
  - Programming model from Lisp
    - (and other functional languages)
- Many problems can be phrased this way
- Easy to distribute across nodes
- Nice retry/failure semantics

Map/Reduce ala Google

- map(key, val) is run on each item in set
  - emits new-key / new-val pairs

- reduce(key, vals) is run for each unique key emitted by map()
  - emits final output

count words in docs

- Input consists of (url, contents) pairs
- map(key=url, val=contents):
  - For each word *w* in contents, emit (w, “1”)
- reduce(key=word, values=uniq_counts):
  - Sum all “1”s in values list
  - Emit result “(word, sum)”

Count, Illustrated

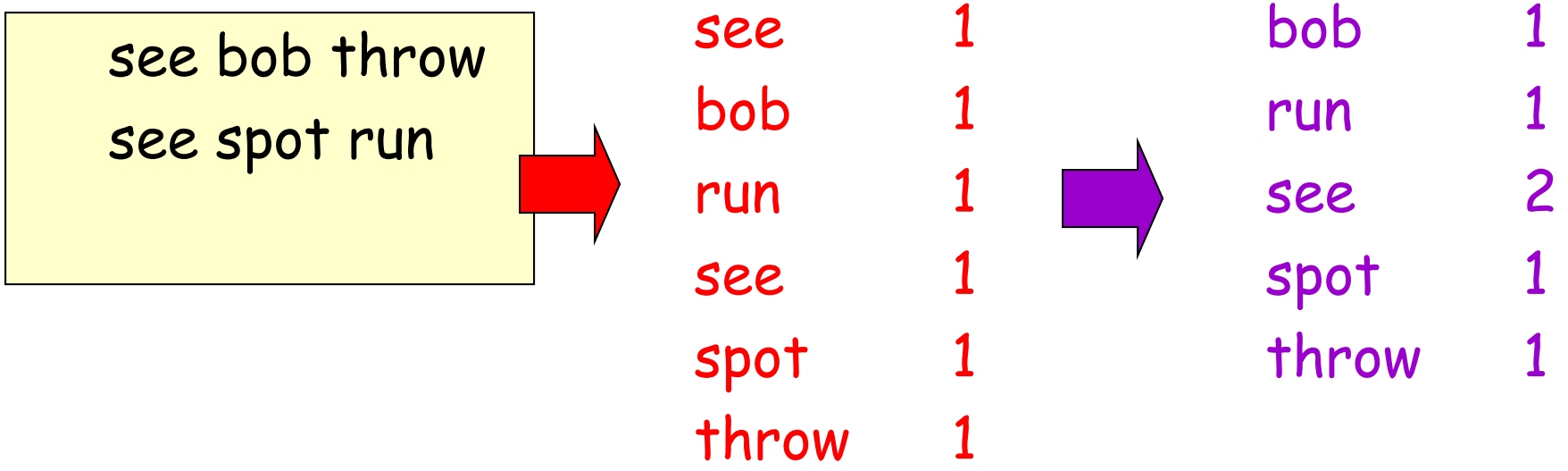
map(key=url, val=contents):

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

reduce(key=word, values=uniq_counts):

Sum all “1”s in values list

Emit result “(word, sum)”



Grep

- Input consists of (url+offset, single line)
- map(key=url+offset, val=line):
  - If contents matches regexp, emit (line, “1”)

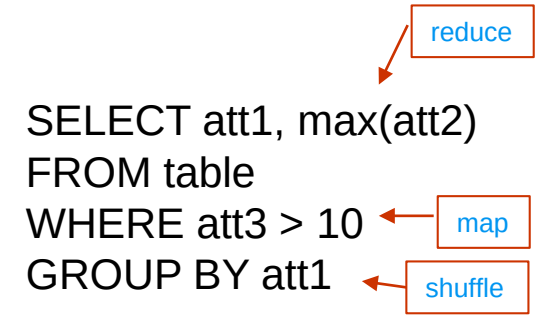
- reduce(key=line, values=uniq_counts):
  - Don’t do anything; just emit line

Execution

- How is this distributed?
  1. Partition input key/value pairs into chunks, run map() tasks in parallel
  2. After all map()s are complete, consolidate all emitted values for each unique emitted key
  3. Now partition space of output map keys (this is called a “shuffle”), and run reduce() in parallel

SQL operators in MapReduce

- Each “processor” from Chapter 18 of textbook corresponds to a partition of a “Map task” or “Reduce task”.
  - These partitions run in parallel across machines in a cluster
- Many of the parallel version of operators we discussed required a repartition
  - E.g. both sort algorithms, partitioned join, parallel hash join, fragment-and-replicate join, grouping/aggregation, etc.
  - Map defines how to do repartition, shuffle repartitions, and the operator done in reduce
    - In some cases, data was already partitioned correctly and this can be skipped
- Operators that do not require a repartition typically done during Map
  - E.g. Operators done during a scan (e.g. selects and projects without deduplication)

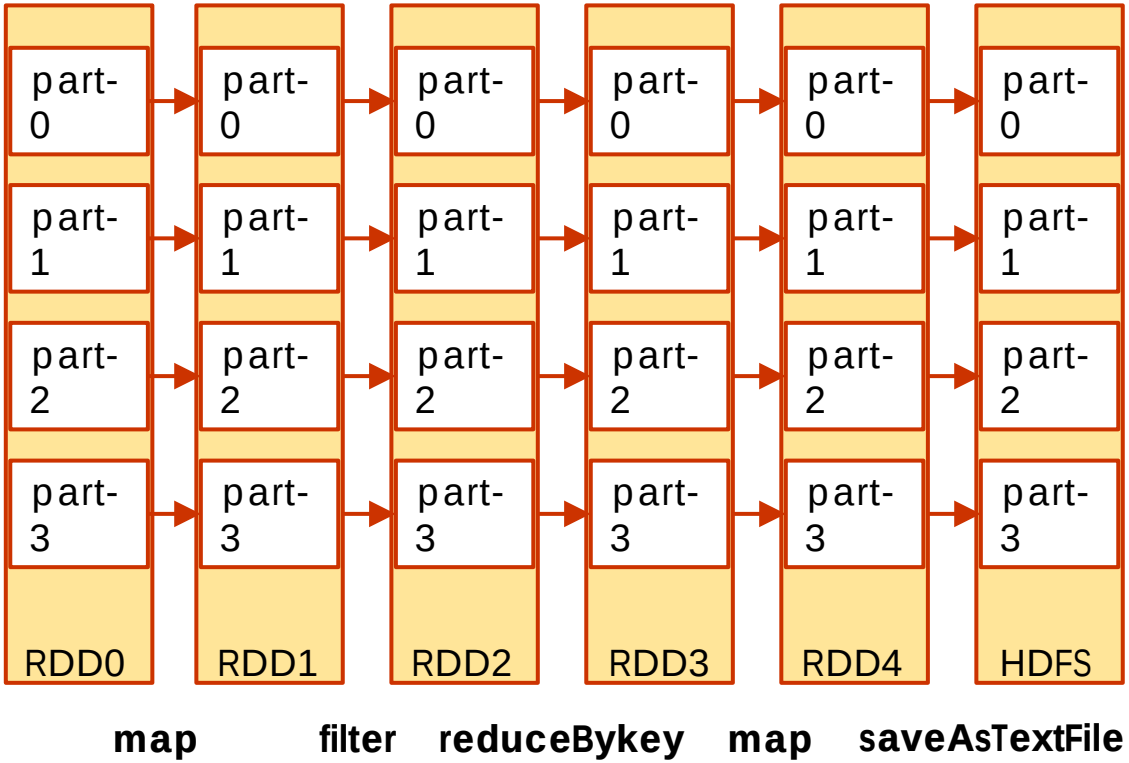


Map pseducode

```
for every tuple
  if att3 > 10
    output (att1, [att1, att2])
  endif
endfor
```

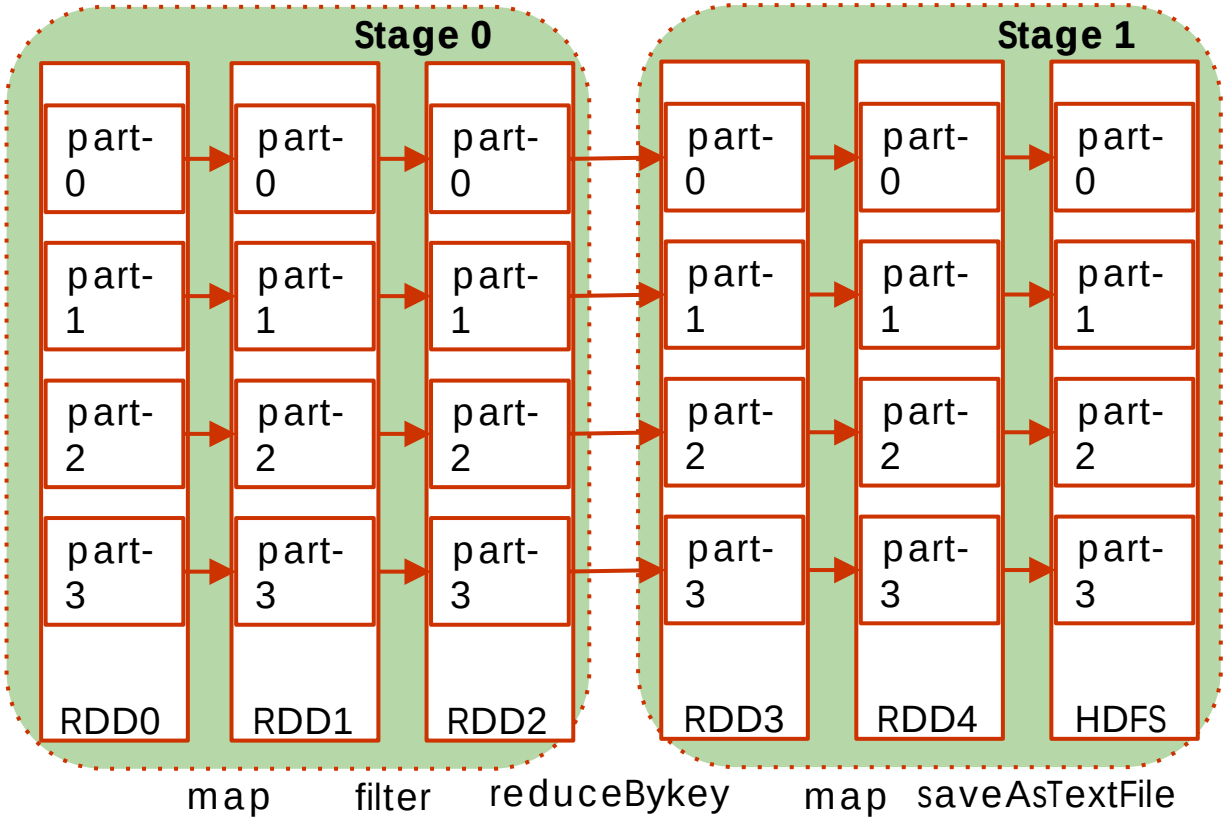
Spark generalizes MapReduce model

```
sc.textFile(hdfsPath)
    .map(parseInput)
    .filter(subThreshold)
    .reduceByKey(tallyCount)
    .map(formatOutput)
    .saveAsTextFile(outPath)
```



Spark generalizes MapReduce model

```
sc.textFile(hdfsPath)
    .map(parseInput)
    .filter(subThreshold)
    .reduceByKey(tallyCount)
    .map(formatOutput)
    .saveAsTextFile(outPath)
```



Spark generalizes MapReduce model

```
sc.textFile(hdfsPath)
    .map(parseInput)
    .filter(subThreshold)
    .reduceByKey(tallyCount)
    .map(formatOutput)
    .saveAsTextFile(outPath)
```