Parallel Data Processing†

Introduction to Databases

CompSci 316 Fall 2017

DUKE
COMPUTER SCIENCE

†*Most contents are drawn and adapted from slides by Madga Balazinska at U. Washington*

Announcements (Thu., Dec. 5)

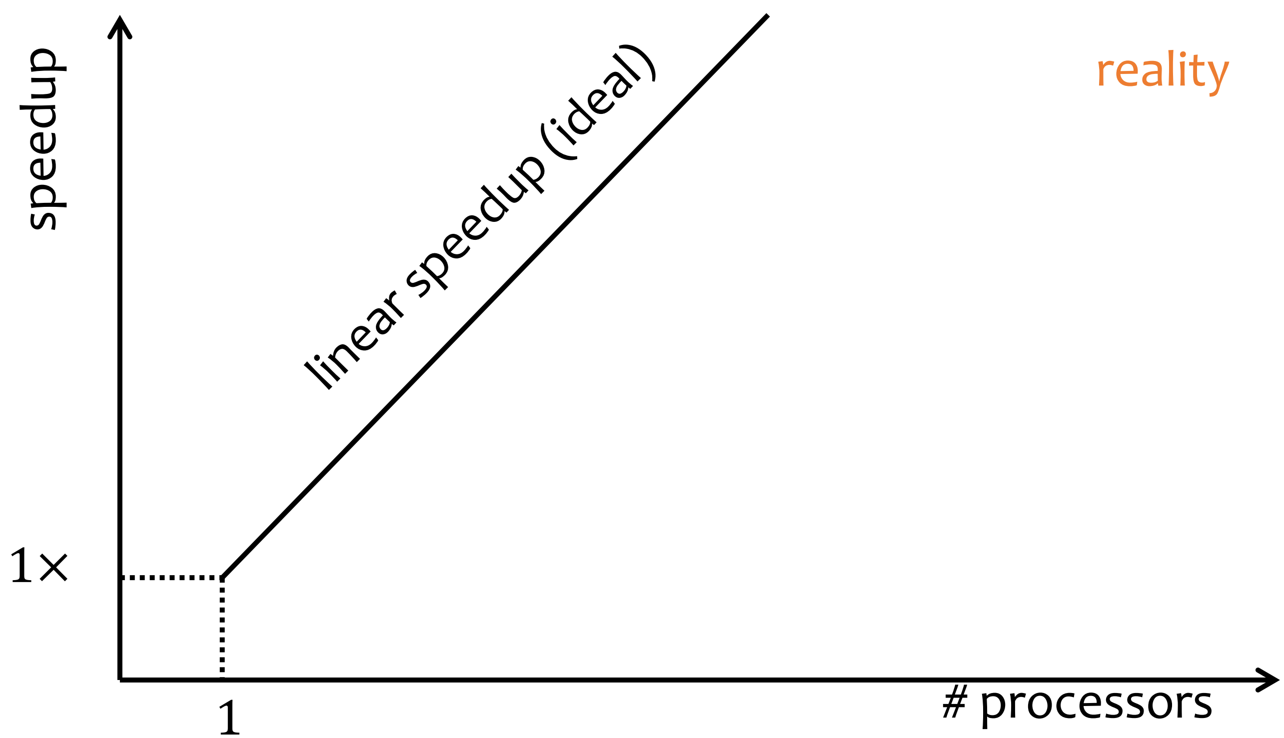
- Homework #4 sample solution to be posted
- Final exam Sat. Dec. 16 2-5pm
  - **This room**
  - Open-book, open-notes
  - Comprehensive, but with strong emphasis on the second half of the course
  - Sample final + solution posted on Sakai
- Project demos starting
  - Check your email for schedule
  - Submit report/code before demo
  - Today: ***LegiToken***, by Josh, Alex, Austin, Oscar, Stuart, and Trenton

Parallel processing

- Improve performance by executing multiple operations in parallel
- Cheaper to scale than relying on a single increasingly more powerful processor
- Performance metrics
  - Speedup, in terms of completion time
  - Scaleup, in terms of time per unit problem size
  - Cost: completion time × # processors × (cost per processor per unit time)

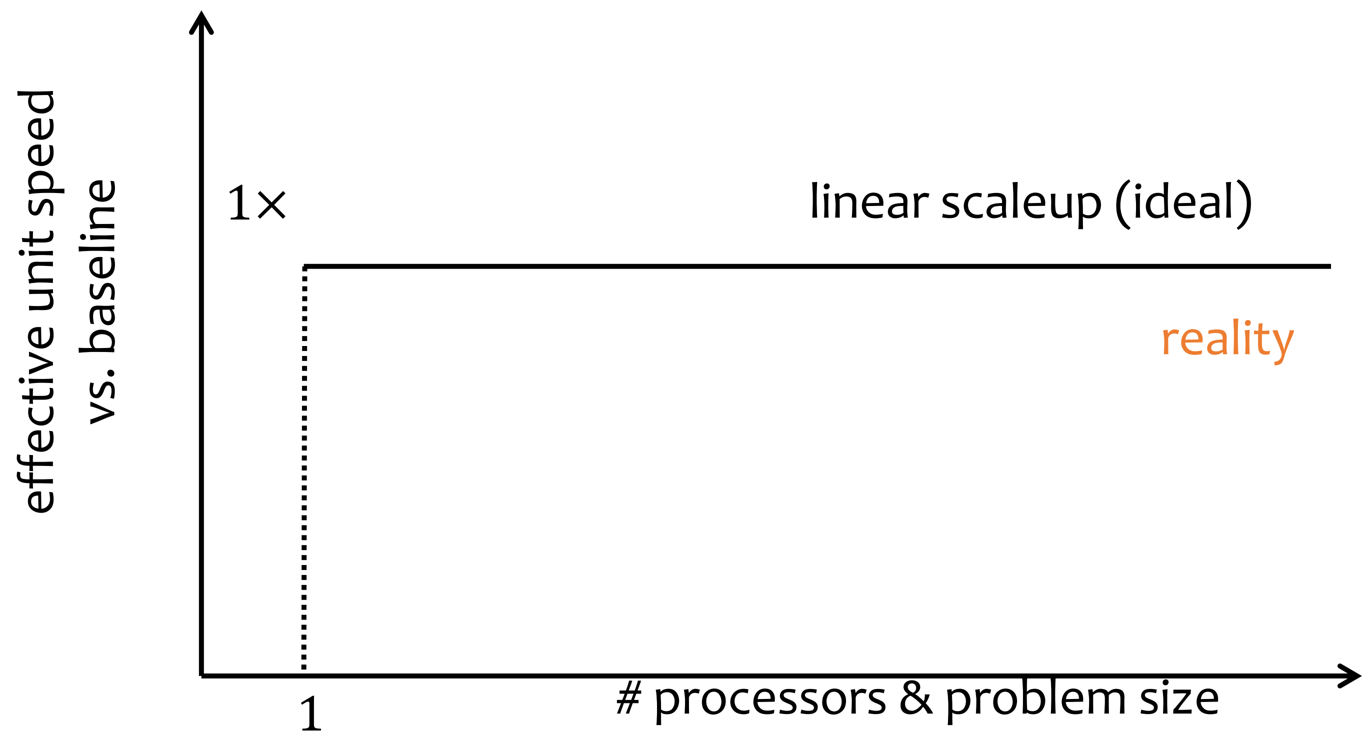
Speedup

- Increase # processors → how much faster can we solve the same problem?
  - Overall problem size is fixed



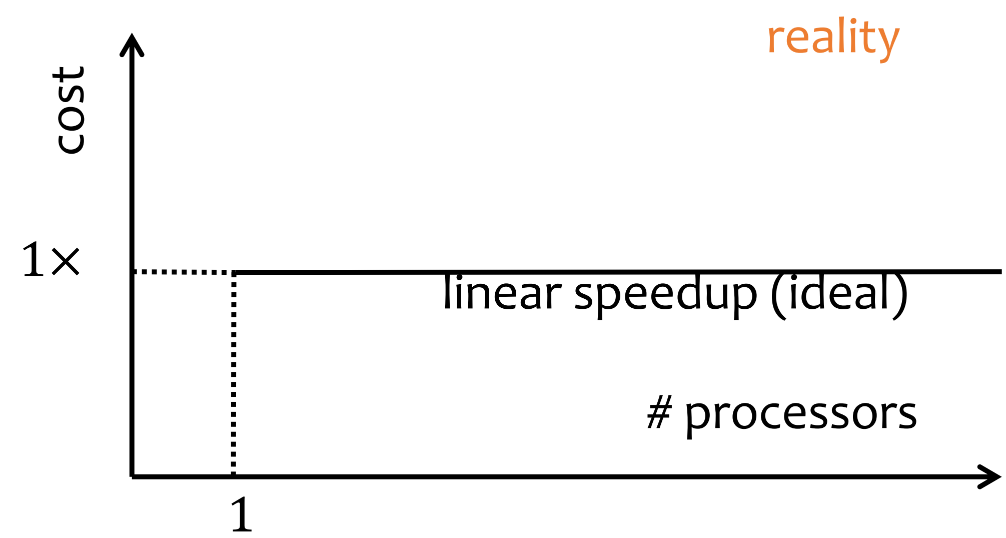
Scaleup

- Increase # processors and problem size proportionally → can we solve bigger problems in the same time?
  - Per-processor problem size is fixed

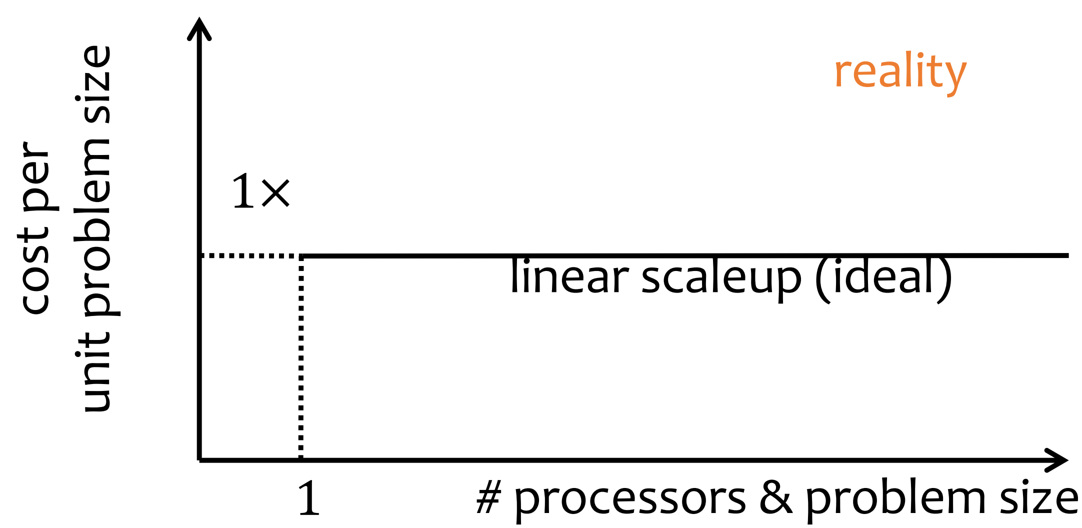


Cost

- Fix problem size



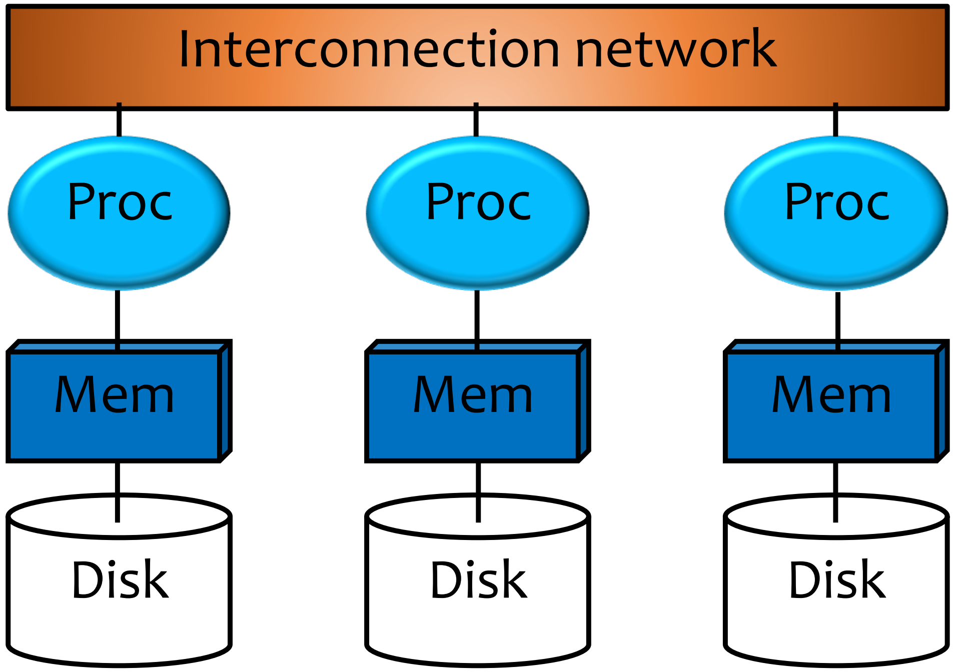
- Increase problem size proportionally with # processors



Why linear speedup/scaleup is hard

- Startup
  - Overhead of starting useful work on many processors
- Communication
  - Cost of exchanging data/information among processors
- Interference
  - Contention for resources among processors
- Skew
  - Slowest processor becomes the bottleneck

Shared-nothing architecture



- Most scalable (vs. shared-memory and shared-disk)
  - Minimizes interference by minimizing resource sharing
  - Can use commodity hardware
- Also most difficult to program

Parallel query evaluation opportunities

- Inter-query parallelism
  - Each query can run on a different processor
- Inter-operator parallelism
  - A query runs on multiple processors
  - Each operator can run on a different processor
- Intra-operator parallelism
  - An operator can run on multiple processors, each working on a different "split" of data/operation

A brief tour of two systems

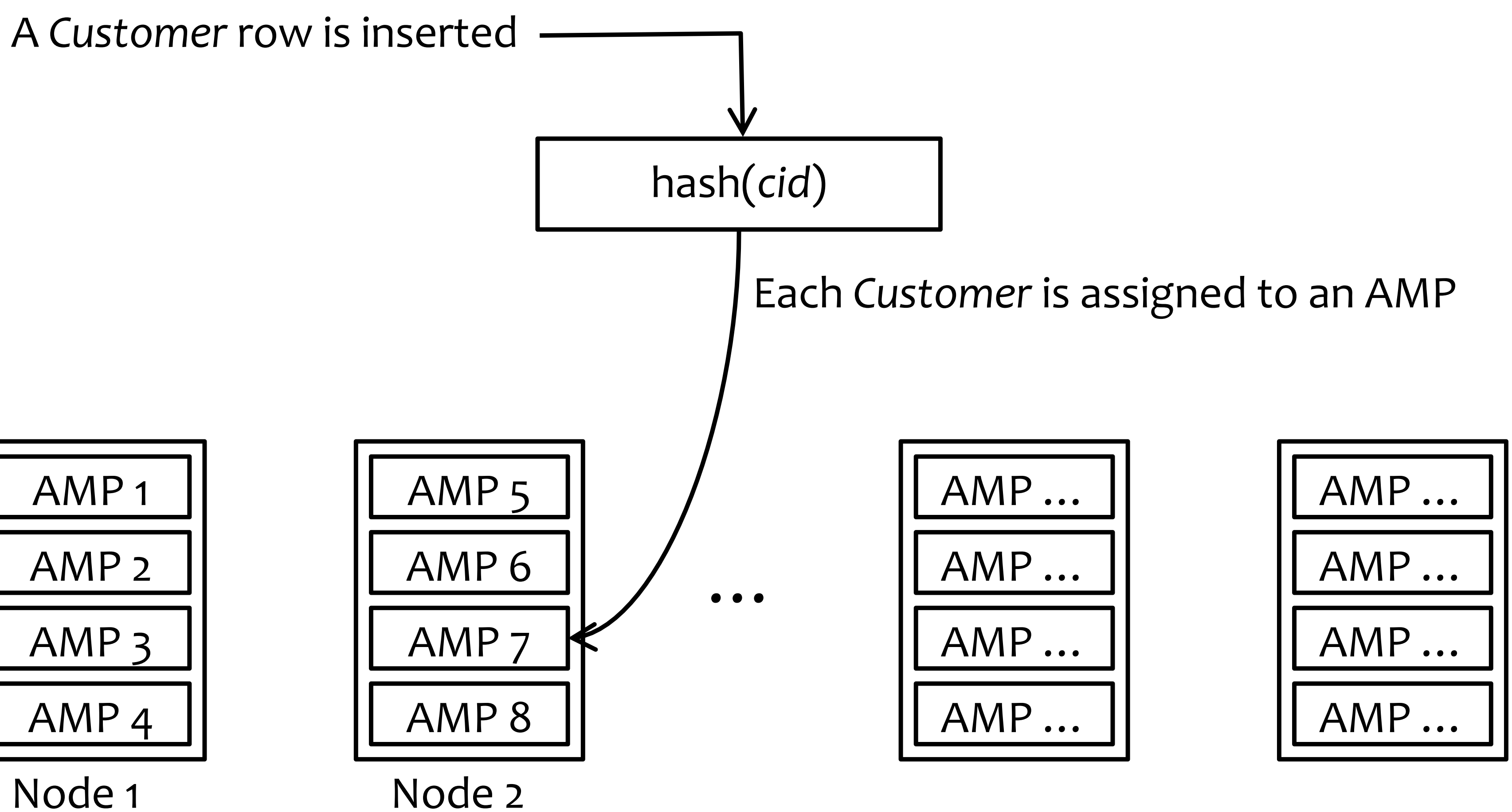
- Parallel DBMS (e.g.,. Teradata)
  - Provides the same abstractions (e.g., relational data model, SQL, transactions) as a regular DBMS
  - Parallelization handled behind the scene
- MapReduce (e.g., Hadoop)
  - Supports easy scaling out (e.g., adding lots of commodity servers) and failure handling
  - Does not require loading data into tables
  - Exposes parallelism to programmers
    - Other tools built on top of MapReduce can provide higher-level abstractions

Horizontal data partitioning

- Split a table R into p chunks, each stored at one of the p processors
- Splitting strategies:
  - Round robin assigns the i-th row assigned to chunk $(i \bmod p)$
  - Hash-based partitioning on attribute A assigns row r to chunk $(h(r.A) \bmod p)$
  - Range-based partitioning on attribute A partitioning the range of $R.A$ values into p ranges, and assigns row r to the chunk whose corresponding range contains $r.A$

Teradata: an example parallel DBMS

- Hash-based partitioning of *Customer* on *cid*

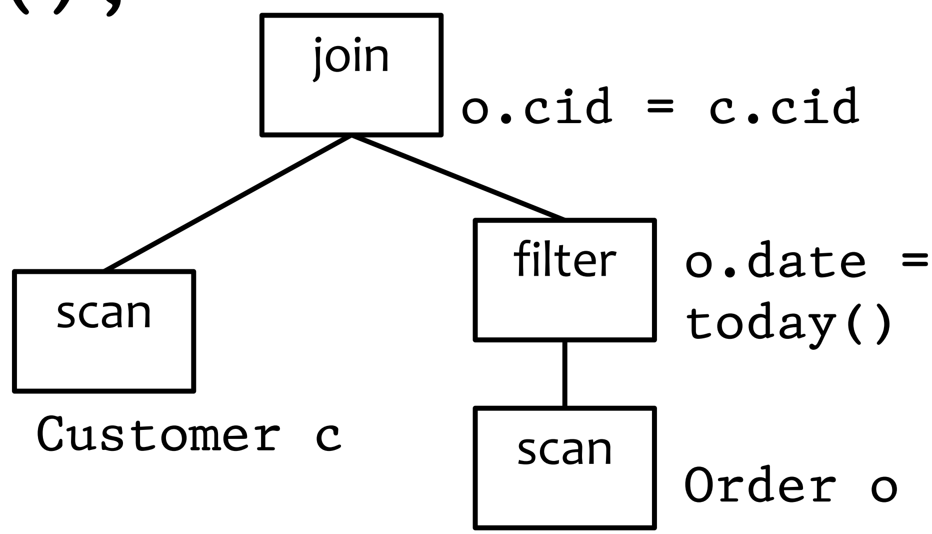


AMP = unit of parallelism in Teradata

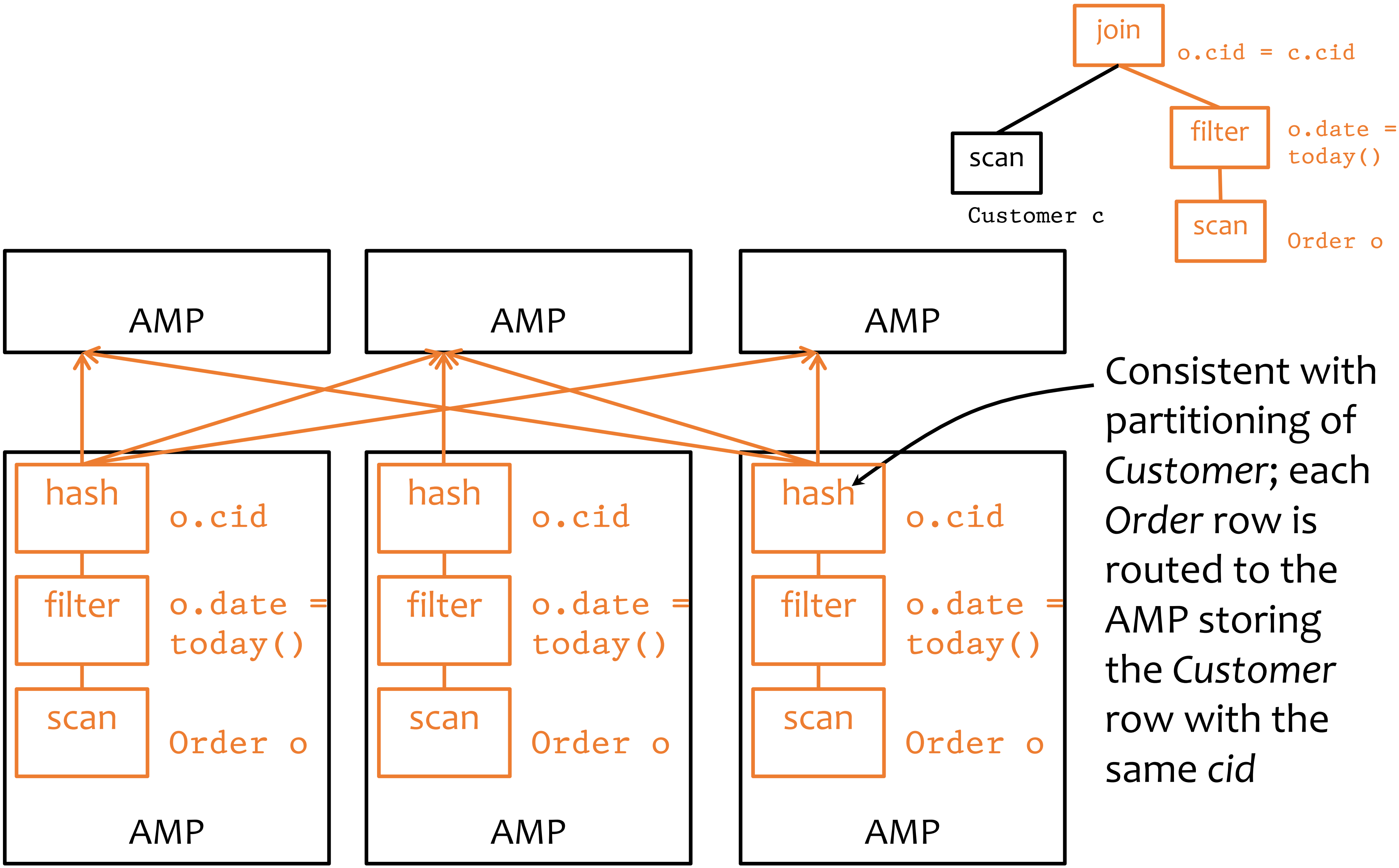
Example query in Teradata

- Find all orders today, along with the customer info

```
SELECT *
FROM Order o, Customer c
WHERE o.cid = c.cid
AND o.date = today();
```

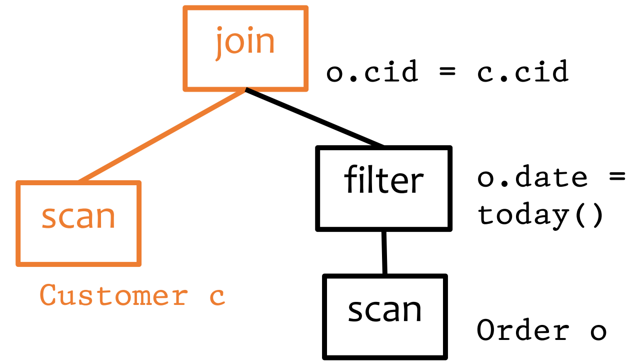


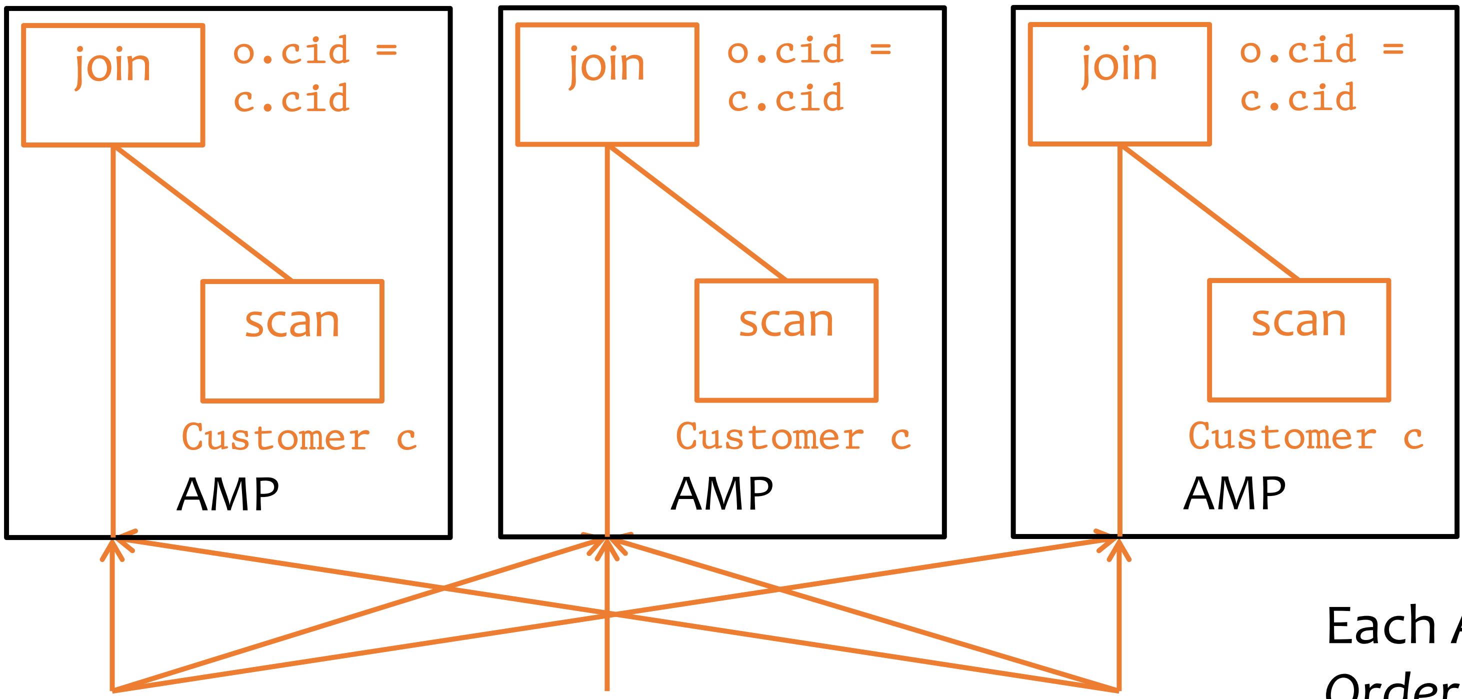
Teradata example: scan-filter-hash



Consistent with partitioning of *Customer*; each *Order* row is routed to the AMP storing the *Customer* row with the same *cid*

Teradata example: hash join





Each AMP processes *Order* and *Customer* rows with the same *cid* hash

MapReduce: motivation

- Many problems can be processed in this pattern:
  - Given a lot of unsorted data
  - Map: extract something of interest from each record
  - Shuffle: group the intermediate results in some way
  - Reduce: further process (e.g., aggregate, summarize, analyze, transform) each group and write final results

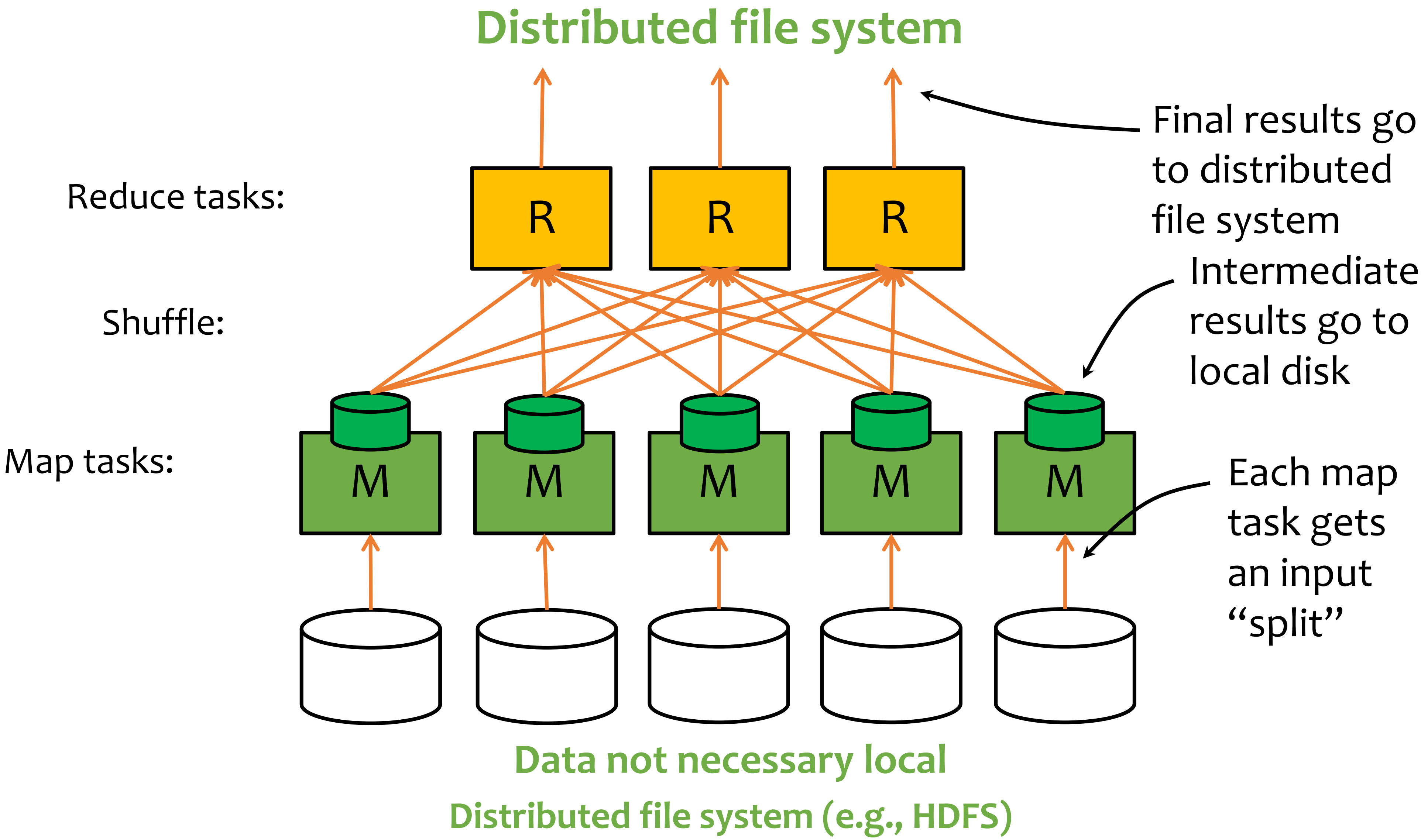
  (Customize map and reduce for problem at hand)

☞ Make this pattern easy to program and efficient to run
  - Original Google paper in *OSDI* 2004
  - Hadoop has been the most popular open-source implementation

M/R programming model

- Input/output: each a collection of key/value pairs
- Programmer specifies two functions
  - $\mathrm{map}(k_1, v_1) \rightarrow \mathrm{list}(k_2, v_2)$
    - Processes each input key/value pair, and produces a list of intermediate key/value pairs
  - $\mathrm{reduce}\big(k_2, \mathrm{list}(v_2)\big) \rightarrow \mathrm{list}(v_3)$
    - Processes all intermediate values associated with the same key, and produces a list of result values (usually just one for the key)

M/R execution



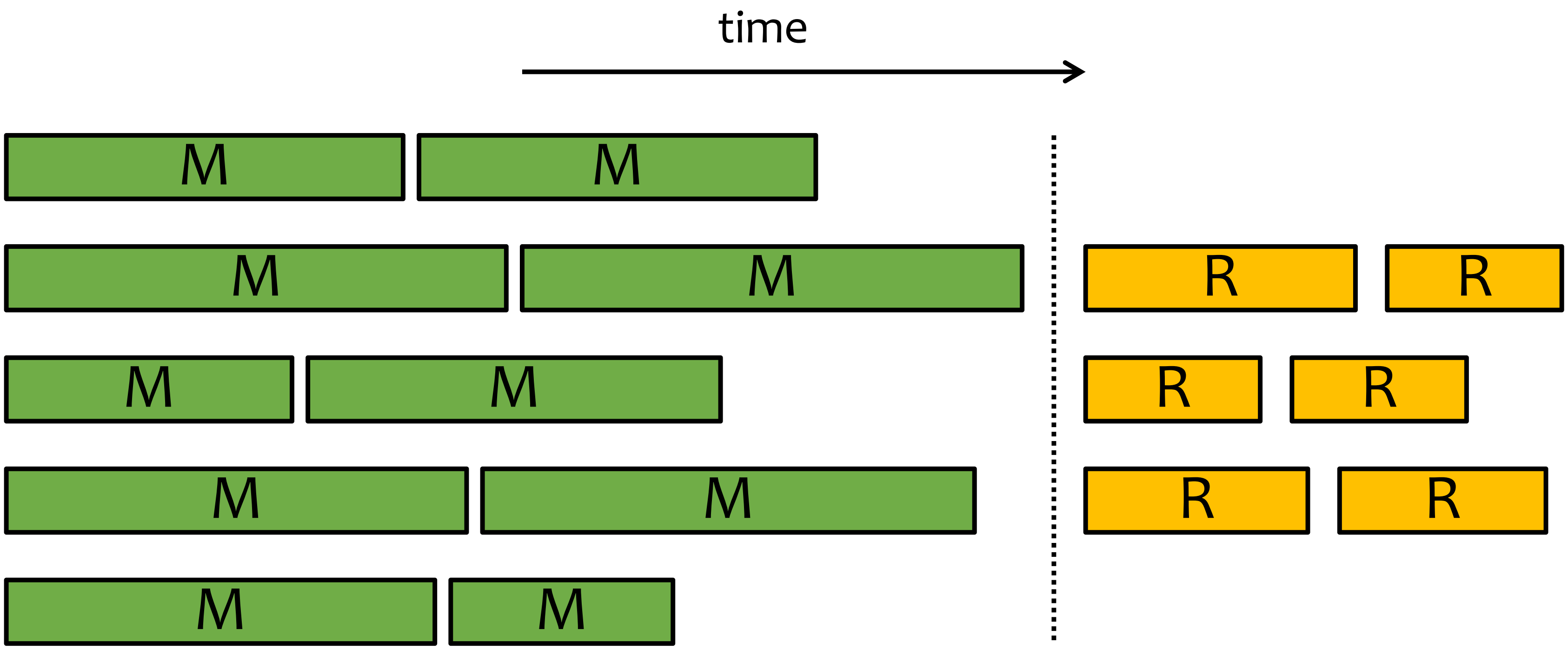
M/R example: word count

- Expected input: a huge file (or collection of many files) with millions of lines of English text
- Expected output: list of (word, count) pairs
- Implementation
  - $\text{map}(\_, \text{line}) \rightarrow \text{list}(\text{word}, \text{count})$
    - Given a line, split it into words, and output $(w, 1)$ for each word w in the line
  - $\text{reduce}\big(\text{word}, \text{list}(\text{count})\big) \rightarrow (\text{word}, \text{count})$
    - Given a word w and list L of counts associated with it, compute $s = \sum_{\text{count} \in L} \text{count}$ and output (w, s)
  - Optimization: before shuffling, map can pre-aggregate word counts locally so there is less data to be shuffled
    - This optimization can be implemented in Hadoop as a "combiner"

Some implementation details

- There is one “master” node
- Input file gets divided into m “splits,” each a contiguous piece of the file
- Master assigns m map tasks (one per split) to “workers” and tracks their progress
- Map output is partitioned into r “regions”
- Master assigns r reduce tasks (one per region) to workers and tracks their progress
- Reduce workers read regions from the map workers’ local disks

M/R execution timeline



- When there are more tasks than workers, tasks execute in "waves"
  - Boundaries between waves are usually blurred
- Reduce tasks can't start until all map tasks are done

More implementation details

- Numbers of map and reduce tasks
  - Larger is better for load balancing
  - But more tasks add overhead and communication
- Worker failure
  - Master pings workers periodically
  - If one is down, reassign its split/region to another worker
- “Straggler”: a machine that is exceptionally slow
  - Pre-emptively run the last few remaining tasks redundantly as backup

M/R example: Hadoop TeraSort

- Expected input: a collection of (key, payload) pairs
- Expected output: sorted (key, payload) pairs
- Implementation
  - Using a small sample of input, find $r - 1$ key values that divides the key range into r subranges where # pairs is roughly equal across them
  - $\text{map}(k, \text{payload}) \to (j, \langle k, \text{payload} \rangle)$
    - If k falls within the j-th subrange
  - $\text{reduce}\big(j, \text{list}(\langle k, \text{payload} \rangle)\big) \to \text{list}(k, \text{payload})$
    - Sort the list of $(k, \text{payload})$ pairs by k and output

Parallel DBMS vs. MapReduce

- Parallel DBMS
  - Schema + intelligent indexing/partitioning
  - Can stream data from one operator to the next
  - SQL + automatic optimization
- MapReduce
  - No schema, no indexing
  - Higher scalability and elasticity
    - Just throw new machines in!
  - Better handling of failures and stragglers
  - Black-box map/reduce functions → hand optimization
  - ☞But newer systems (e.g., Hive, Spark SQL) have added schema, declarative languages, indexing, and automatic optimization