Parallel Data Processing†

Introduction to Databases

CompSci 316 Fall 2019

DUKE
COMPUTER SCIENCE

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

Announcements (Wed., Nov. 20)

- Homework 4 due Mon. after Thanksgiving break
- Piazza project weekly progress update due today

Announcements (Mon., Nov. 25)

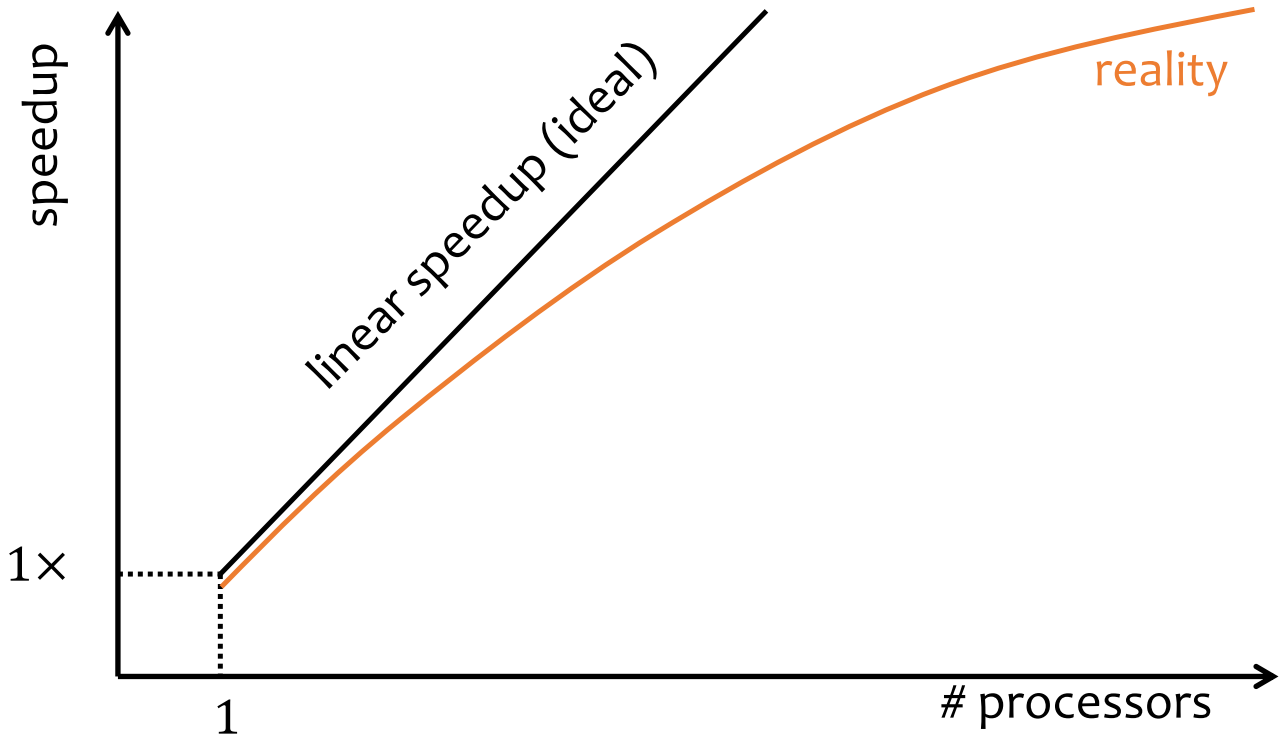
- Homework 4 due in a week
- No Piazza project weekly update due this week

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)

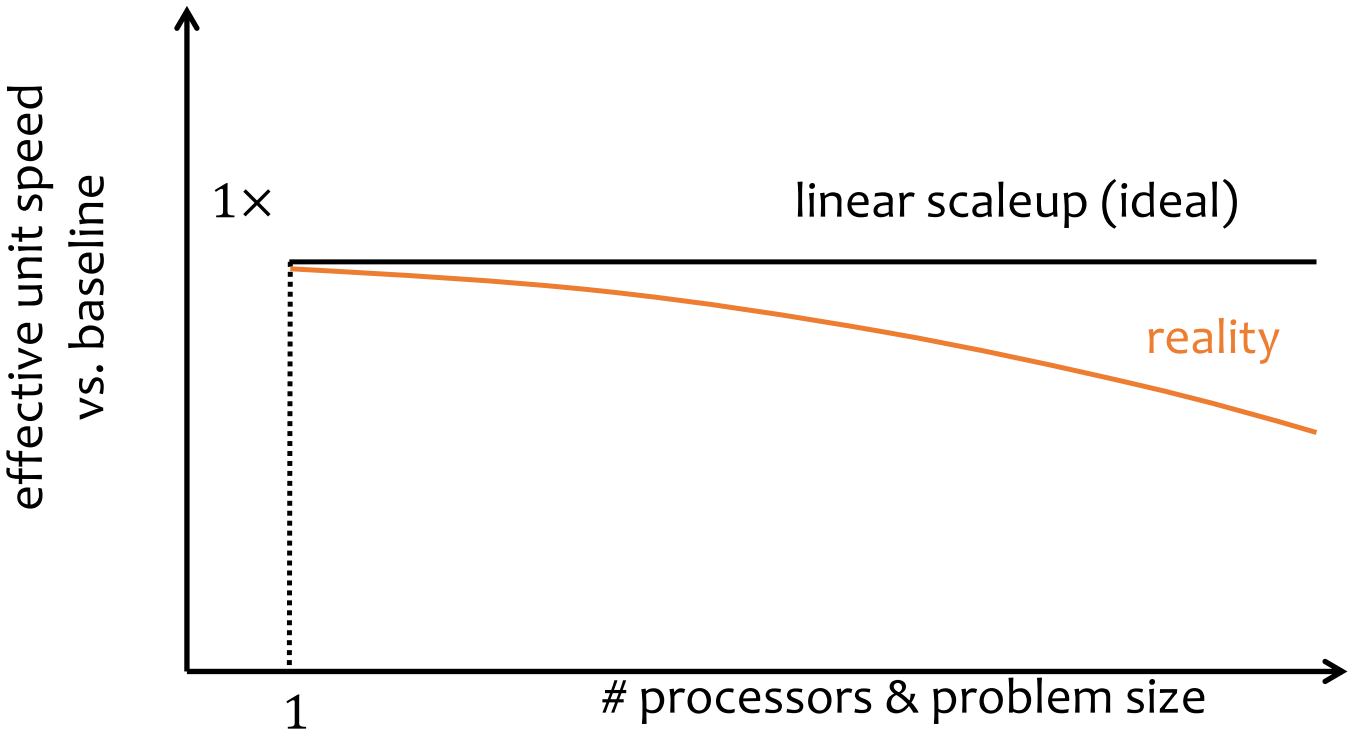
Speedup

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



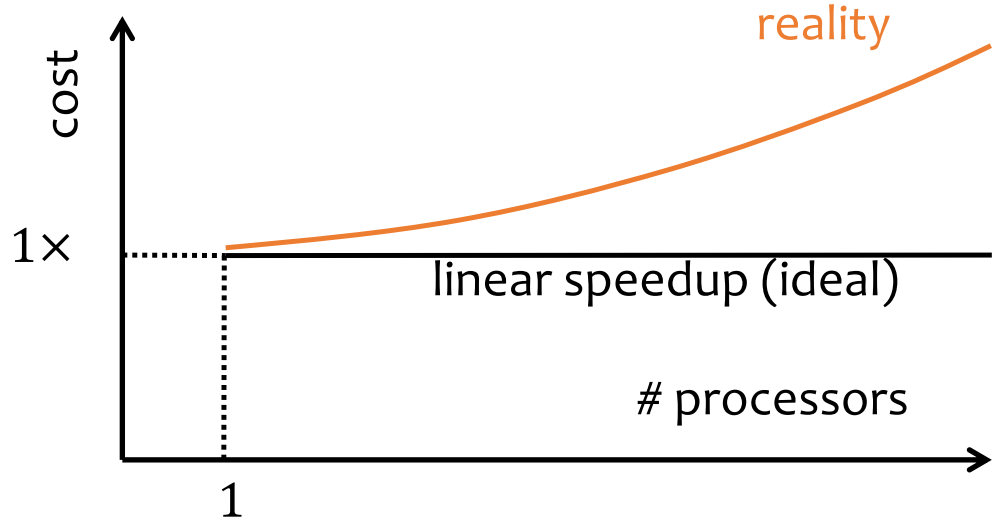
Scaleup

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

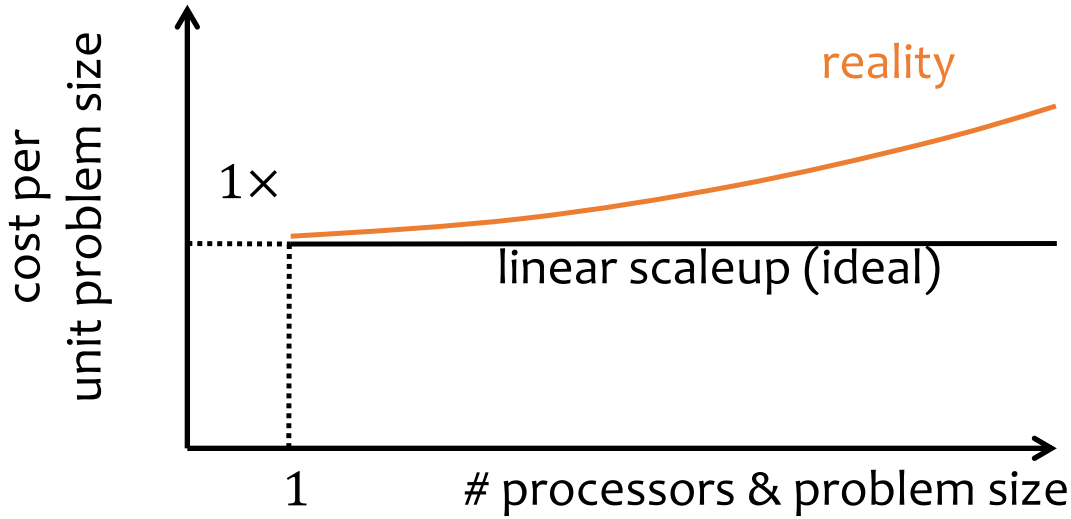


Cost

- Fix problem size



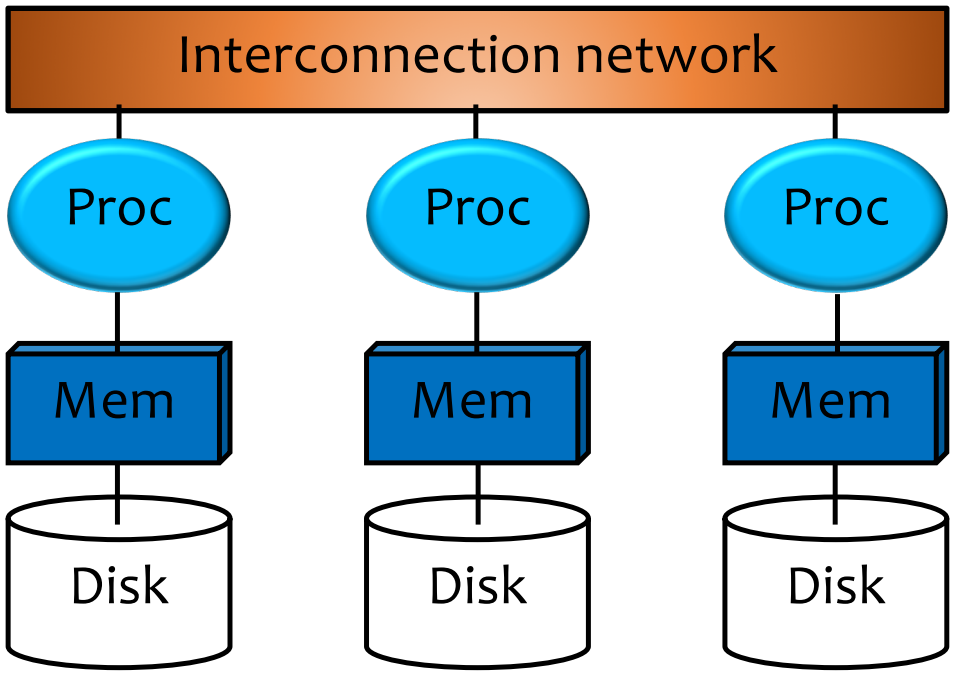
- Increase problem size proportionally with # processors



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

  ☞Focus of this lecture

A brief tour of three approaches

- "DB": parallel DBMS, e.g., Teradata
  - Same abstractions (relational data model, SQL, transactions) as a regular DBMS
  - Parallelization handled behind the scene
- "BD (Big Data)" 15 years go: MapReduce, e.g., Hadoop
  - Easy scaling out (e.g., adding lots of commodity servers) and failure handling
  - Input/output in files, not tables
  - Parallelism exposed to programmers
- "BD" today: Spark
  - Compared to MapReduce: smarter memory usage, recovery, and optimization
  - Higher-level DB-like abstractions (but still no updates)

Parallel DBMS

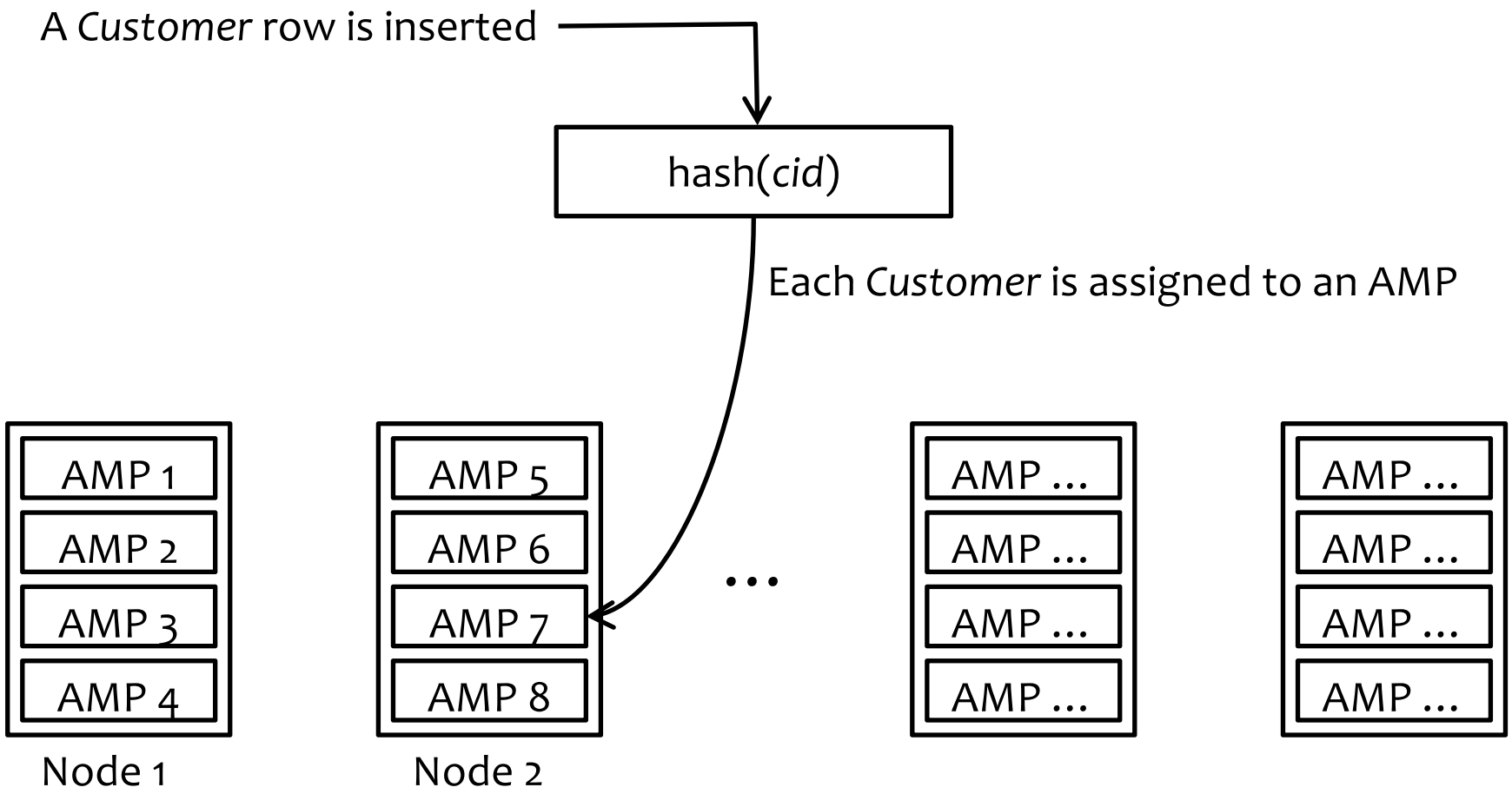
E.g.: TERADATA

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

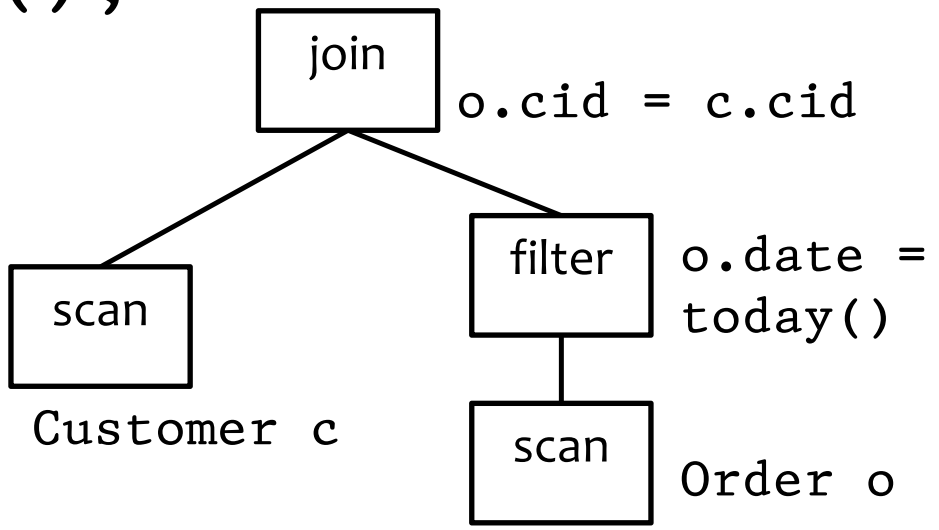
- Hash-based partitioning of *Customer* on *cid*



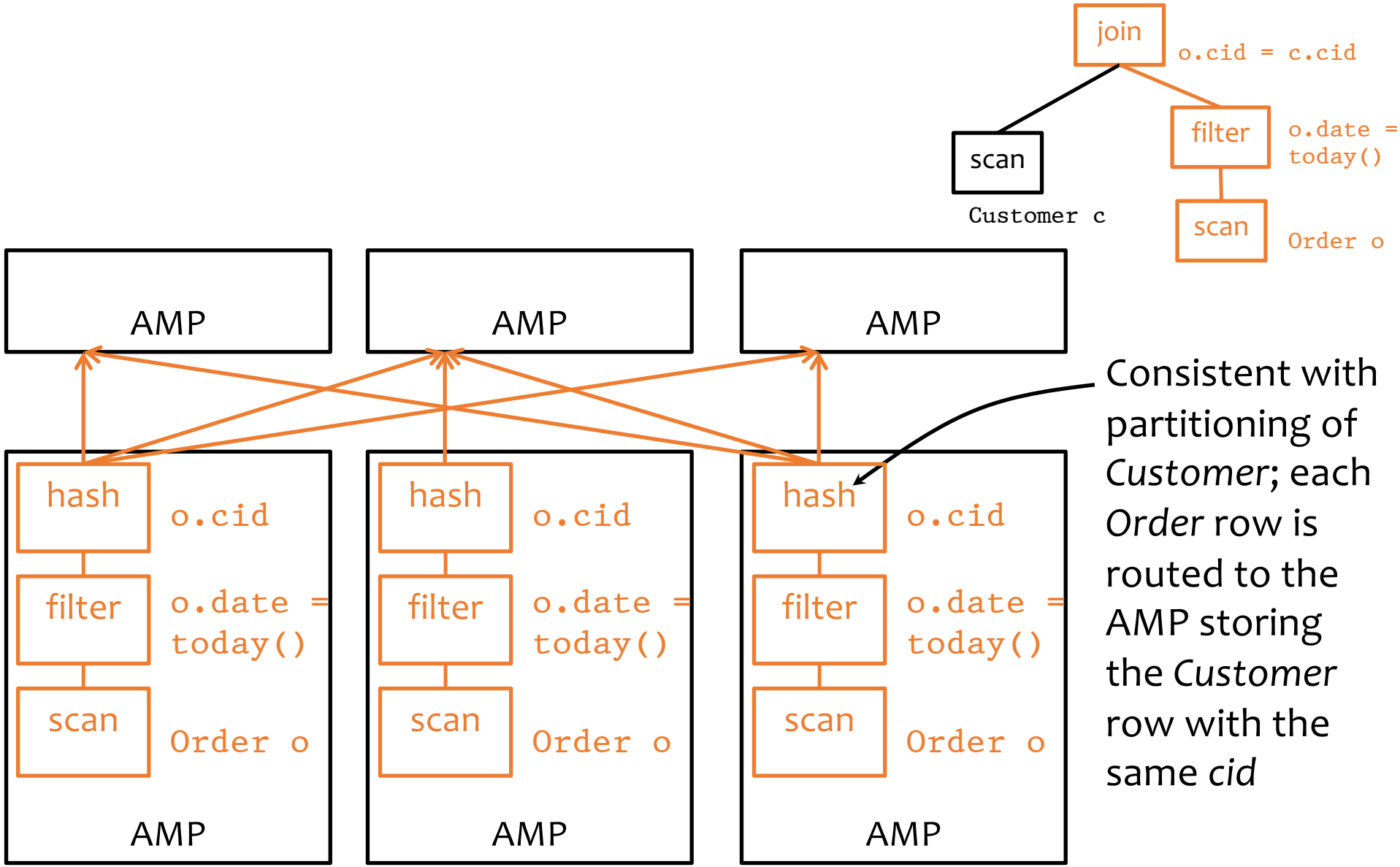
Example query in Teradata

- Find all orders today, along with the customer info

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

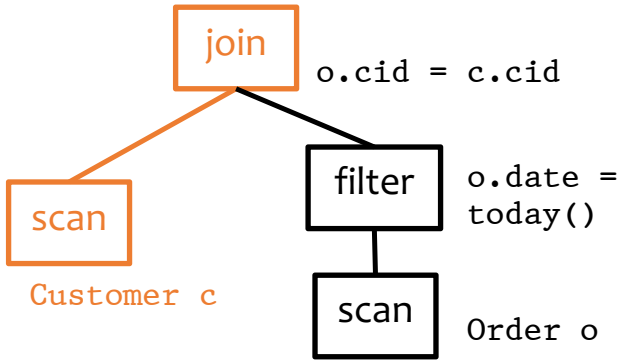


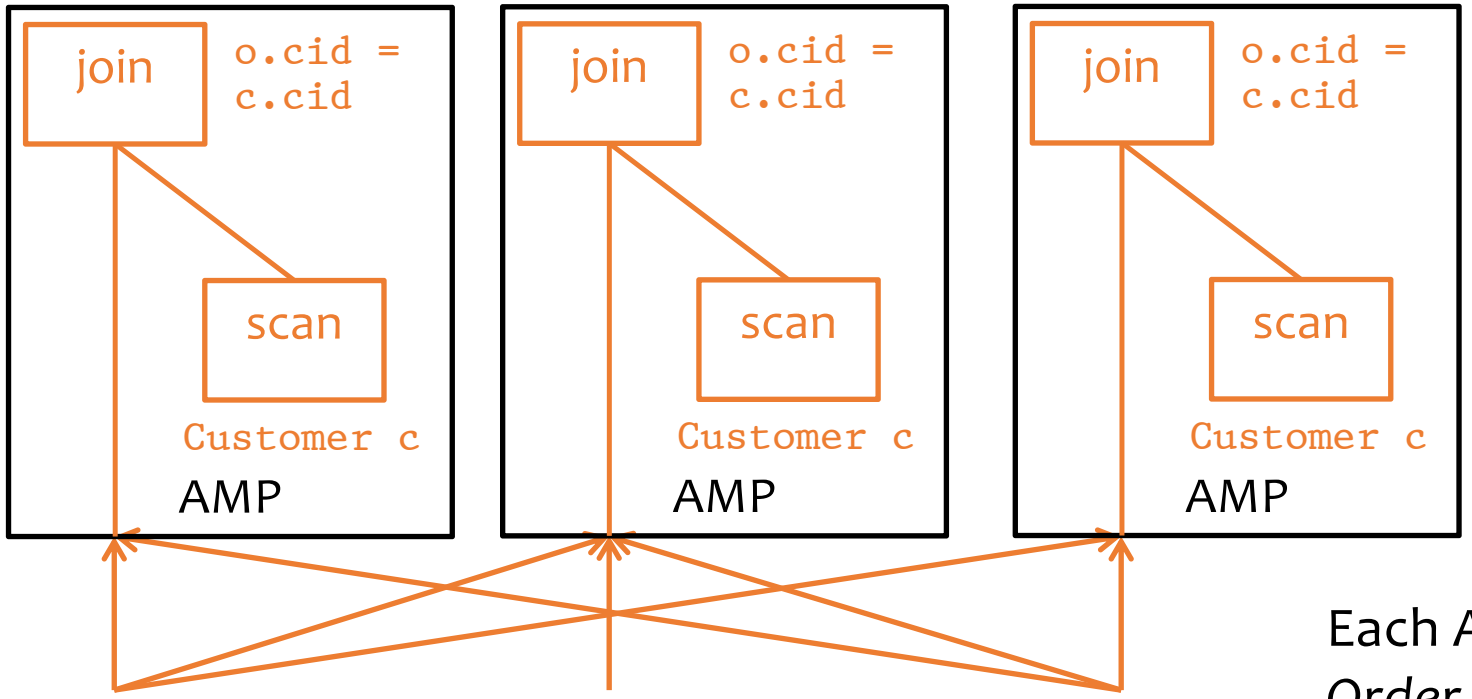
Teradata example: scan-filter-hash



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


hadoop
Map Reduce

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)

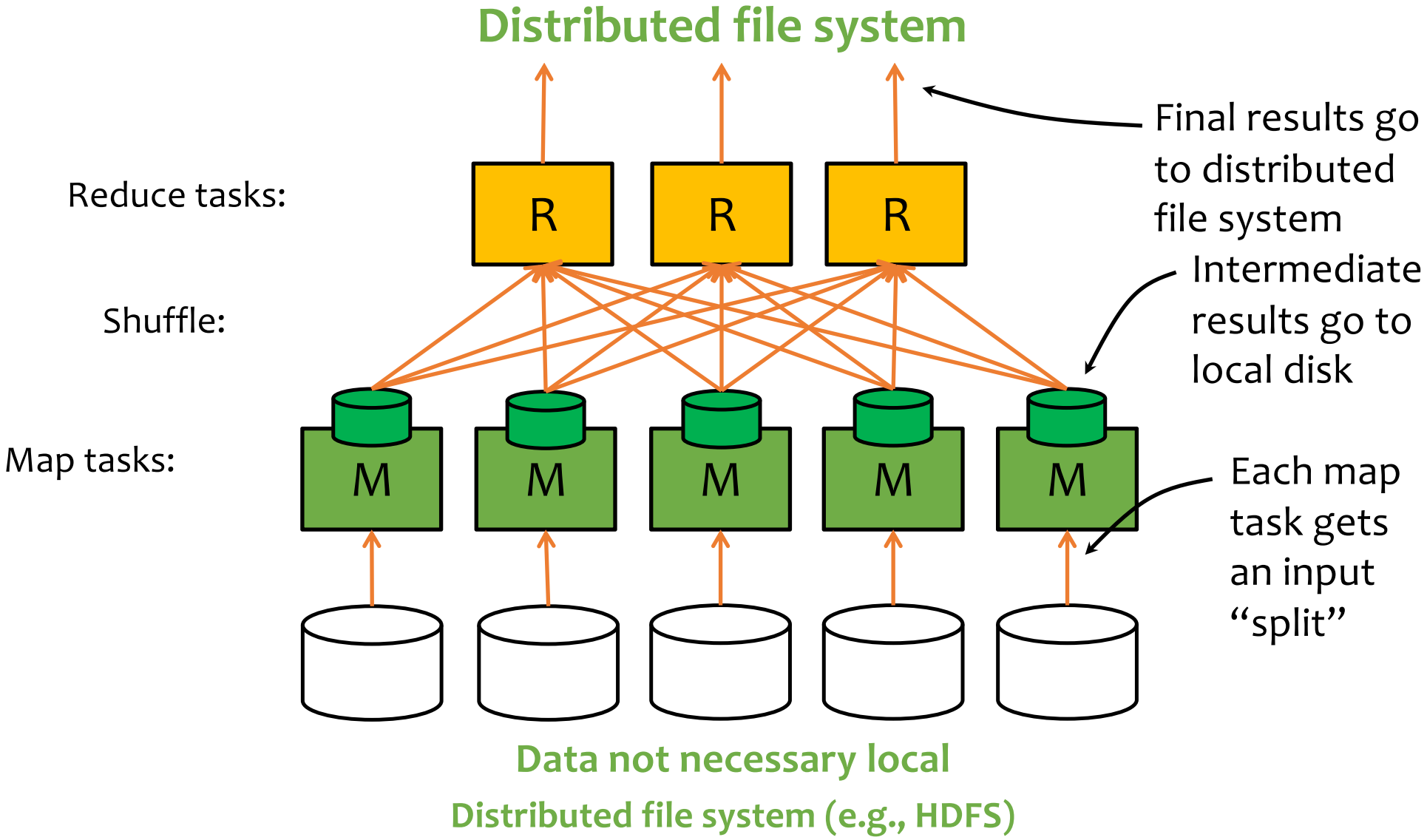
☞ Make this pattern easy to program and efficient to run

  - Original Google paper in *OSDI* 2004
  - Hadoop has been the most popular open-source implementation
  - Spark still supports it

M/R programming model

- Input/output: each a collection of key/value pairs
- Programmer specifies two functions
  - map: $(k_1, v_1) \rightarrow \text{list}(k_2, v_2)$
    - Processes each input key/value pair, and produces a list of intermediate key/value pairs
  - reduce: $\left(k_2, \text{list}(v_2)\right) \rightarrow \text{list}(v_3)$
    - 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



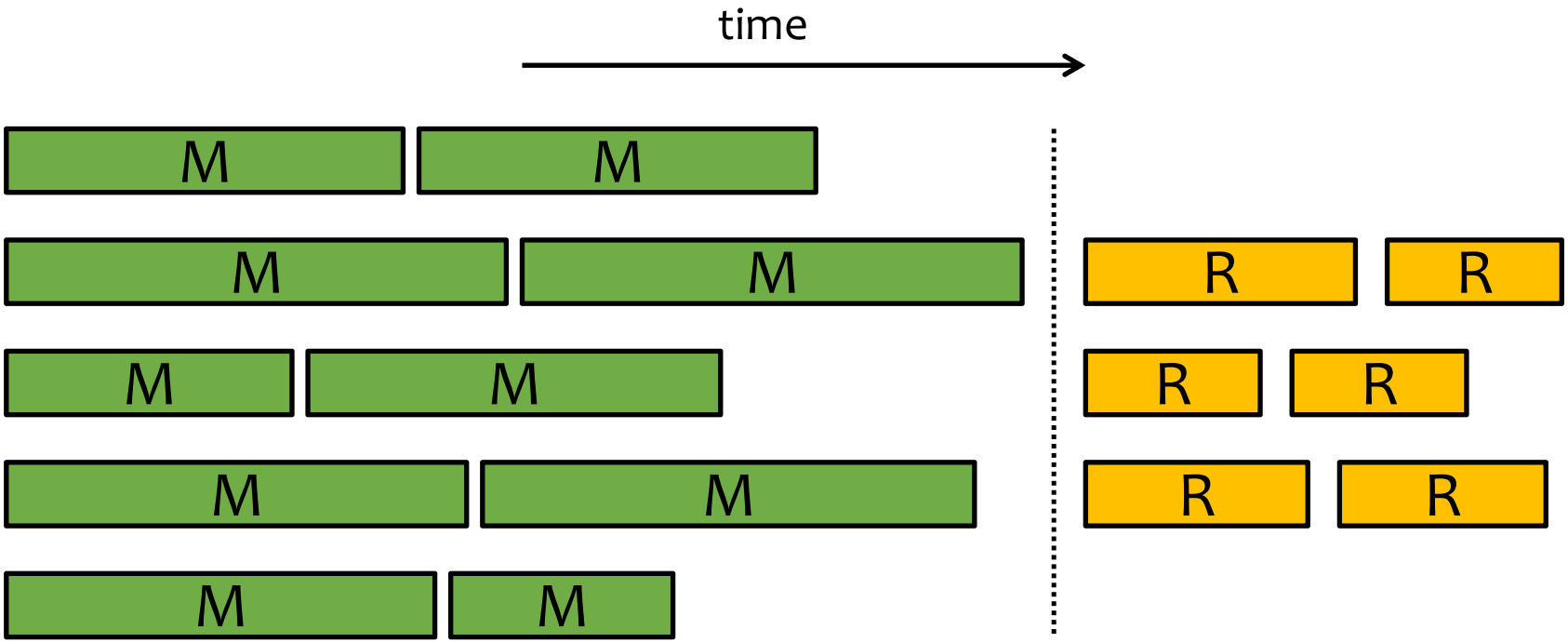
M/R example: word count

- 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
  - $\text{map}(k, \text{payload}) \rightarrow (j, \langle k, \text{payload} \rangle)$
    - If k falls within the j-th subrange
  - $\text{reduce}\big(j, \text{list}(\langle k, \text{payload} \rangle)\big) \rightarrow \text{list}(k, \text{payload})$
    - 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

We will focus on the Python dialect,
although Spark supports multiple languages

Addressing inefficiencies in Hadoop

- Hadoop: no automatic optimization

☞Spark

  - Allow program to be a DAG of DB-like operators, with less reliance on black-box code
  - Delay evaluation as much as possible
  - Fuse operators into stages and compile each stage

- Hadoop: too many I/Os
  - E.g., output of each M/R job is always written to disk
  - But such checkpointing simplifies failure recovery

☞Spark

  - Keep intermediate results in memory
  - Instead of checkpointing, use "lineage" for recovery

RDDs

- Spark stores all intermediate results as Resilient Distributed Datasets (RDDs)
  - Immutable, memory-resident, and distributed across multiple nodes
- Spark also tracks the "lineage" of RDDs, i.e., what expressions computed them
  - Can be done at the partition level

What happens to a RDD if a node crashes?

- The partition of this RDD on this node will be lost
- But with lineage, the master simply recomputes the a lost partition when needed
  - Requires recursive recomputation if input RDD partitions are also missing

Example: votes & explanations

- Raw data reside in lots of JSON files obtained from ProPublica API
- Each vote: URI (id), question, description, date, time, result
- Each explanation: member id, name, state, party, vote URI, date, text, category
  - E.g., “P000523”, “David E. Price”, “NC”, “D”, “https://api.propublica.org/congress/v1/115/house/sessions/2/votes/269.json”, “2018-06-20”, “Mr. Speaker, due to adverse weather and numerous flight delays and cancellations in North Carolina, I was unable to vote yesterday during Roll Call 269, the motion…”, “Travel difficulties”

Basic M/R with Spark RDD

```
explain_fields = ('member_id', 'name', 'state', 'party', 'vote_api_uri',
                  'date', 'text', 'category')

# Map function: map(k_1, v_1) → list(k_2, v_2)
def map(record):
    if len(record) == len(explain_fields):
        return [(record[explain_fields.index('category')], 1)]
    else:
        return []

# Reduce function: reduce(k_2, list(v_2)) → list(v_3)
def reduce(record):
    key, vals = record
    return [(key, len(vals))]
```

Basic M/R with Spark RDD

```
# setting up one RDD that contains all the input:
rdd = sc. ...
# count number of explanations by category; order by
# number (descending) and then category (ascending):
result = rdd\
        .flatMap(map)\
        .groupByKey()\
        .flatMap(reduce)\
        .sortBy(lambda x: (-x[1], x[0]))
for row in result.collect():
    print('|'.join(str(field) for field in row))
```

Be lazy: build up a DAG of "transformations," but no evaluation yet!

Optimize & evaluate the whole DAG only when needed, e.g., triggered by "actions" like collect()

Be careful: Spark RDDs support map() and reduce() too, but they are not the same as those in MapReduce

Moving "BD" to "DB"

Each element in a RDD is an opaque object—hard to program

- Why don't we make each element a "row" with named columns—easier to refer to in processing
  - RDD becomes a *DataFrame* (name from the *R* language)
  - Still immutable, memory-resident, and distributed
- Then why don't we have database-like operators instead of just MapReduce?
  - Knowing their semantics allows more optimization
- Spark in fact pushed the idea further
  - Spark *Dataset* = DataFrame with type-checking
  - And just run SQL over Datasets using *SparkSQL*!

Spark DataFrame

```
from pyspark.sql import functions as F
explain_fields = ('member_id', 'name', 'state', 'party', 'vote_api_uri',
                  'date', 'text', 'category')

# setting up a DataFrame of explanations:
df_explain = sc. ...

# count number of explanations by category; order by
# number (descending) and then category (ascending):
df_explain.groupBy('category')\
          .agg(F.count('name'))\
          .withColumnRenamed('count(name)', 'count')\
          .sort(['count', 'category'], ascending=[0, 1])\
          .show(20, truncate=False)
```

Another Spark DataFrame Example

```
from pyspark.sql import functions as F
vote_fields = ('vote_uri','question','description','date','time','result')
explain_fields = ('member_id', 'name', 'state', 'party', 'vote_api_uri',
                  'date','text', 'category')
# setting up DataFrames for each type of data:
df_votes = sc. ...
df_explain = sc. ...
# what does the following do?
df_votes.join(df_explain.select('vote_api_uri', 'name'),
              df_votes.vote_uri == df_explain.vote_api_uri, 'left_outer')\
        .groupBy('vote_uri', 'date', 'time', 'question', 'description', 'result')\
        .agg(F.count('name'), F.collect_list('name'))\
        .withColumnRenamed('count(name)', 'count')\
        .withColumnRenamed('collect_list(name)', 'names')\
        .sort(['count', 'date', 'time'], ascending=[0, 0, 0])\
        .select('vote_uri', 'date', 'time', 'question', 'description', 'result',
                'count', 'names')\
        .show(20, truncate=False)
```

For each vote, find out which legislators provided explanations; order by the number of such legislators (descending), then date and time (descending)

Summary

- "DB": parallel DBMS
  - Standard relational operators
  - Automatic optimization
  - Transactions
- "BD" 10 years go: MapReduce
  - User-defined map and reduce functions
  - Mostly manual optimization
  - No updates/transactions
- "BD" today: Spark
  - Still supporting user-defined functions, but more standard relational operators than older "BD" systems
  - More automatic optimization than older "BD" systems
  - No updates/transactions