Application Architecture

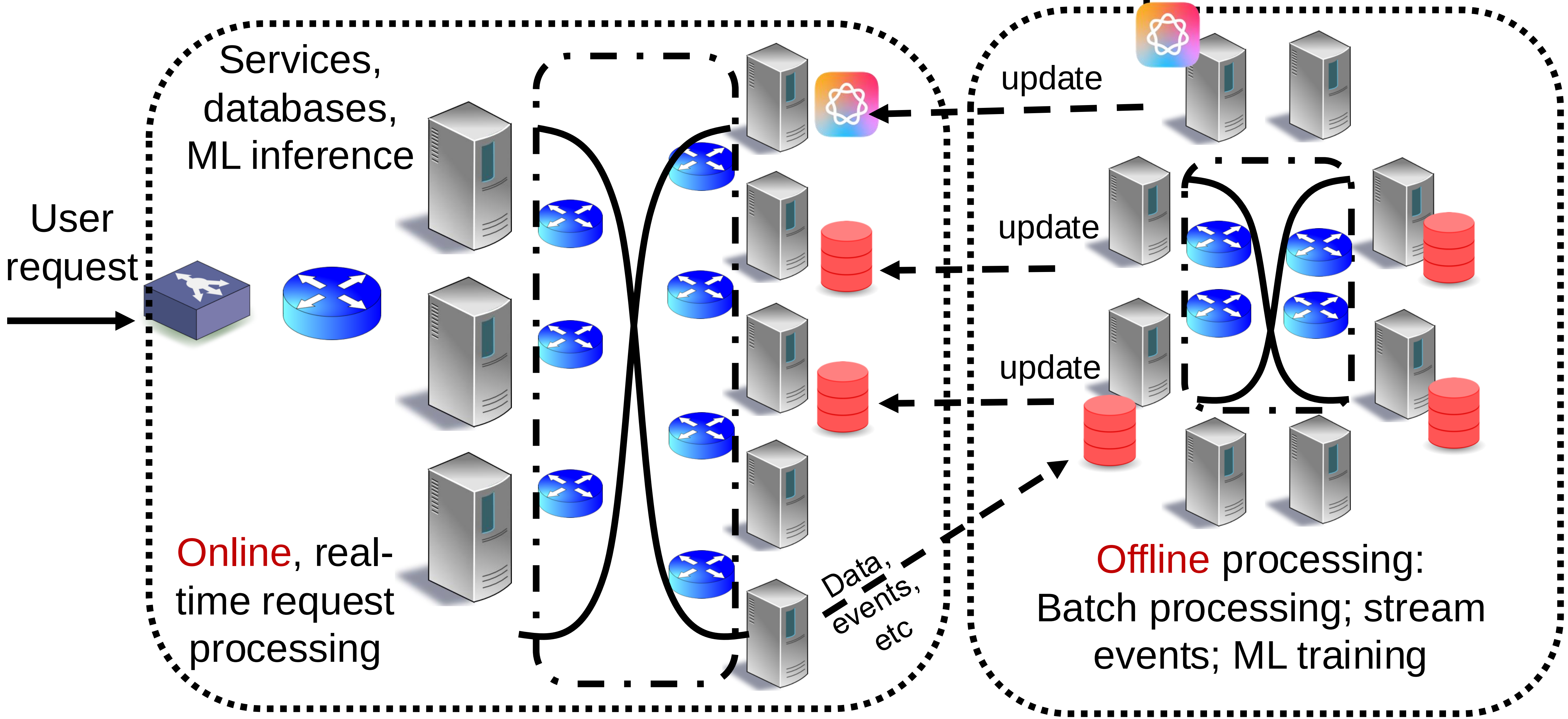
Lecture 6

Srinivas Narayana

http://www.cs.rutgers.edu/~sn624/553-S25

RUTGERS
UNIVERSITY | NEW BRUNSWICK

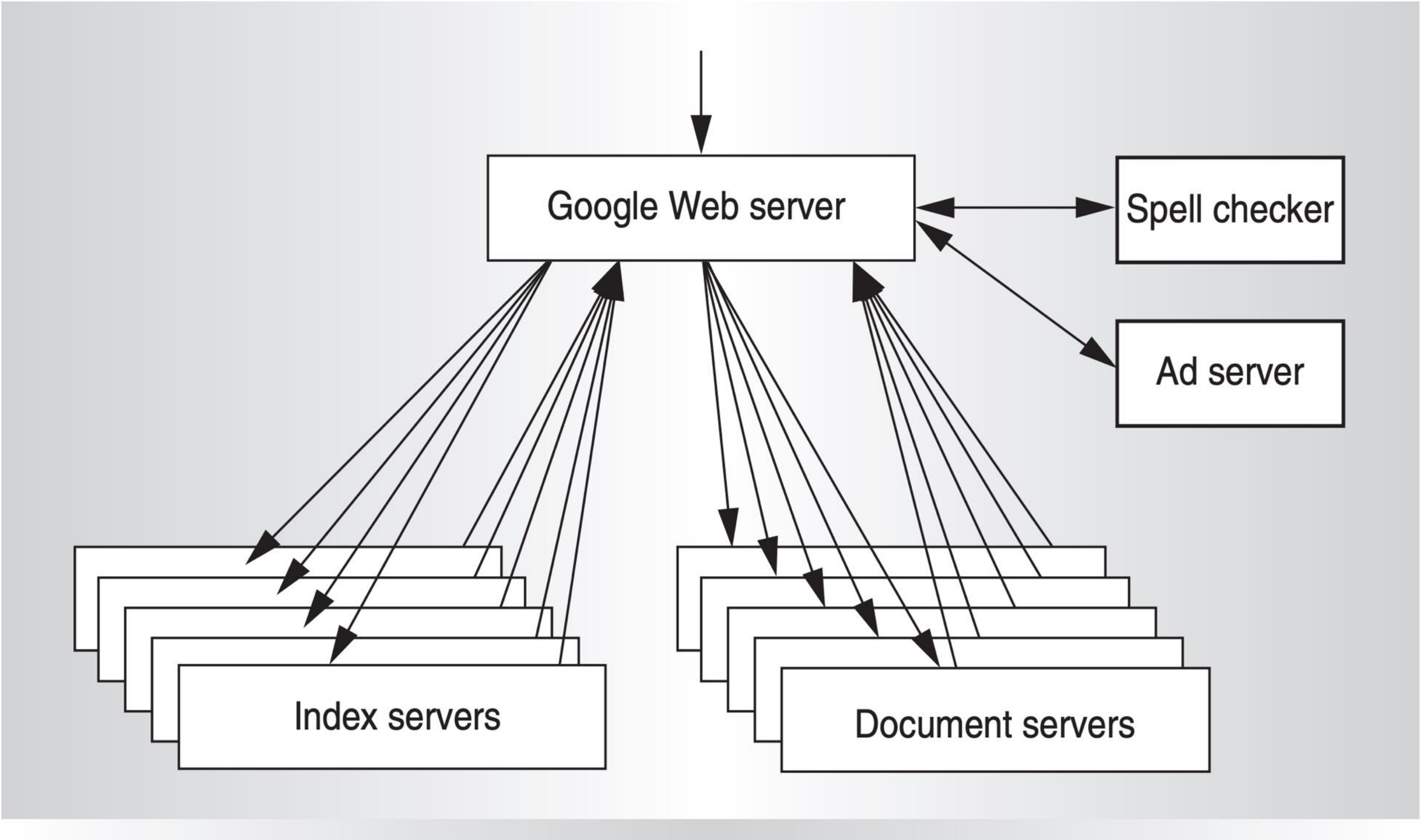
Review: Offline and Online components



Partition-Aggregate

Processing interactive search queries

Review: Google search architecture



Web search for a planet, MICRO'03

Review of the Web Search workload

- Depending on the user's query, decompress a part of an index, then search for document IDs there
- Depending on the user's query, collect snippets from within Web documents
- Data-dependent accesses
- High branch misprediction
- Blocks randomly accessed (OK within block)
- Fewer opportunities for instruction-level parallelism; faster/better servers not better

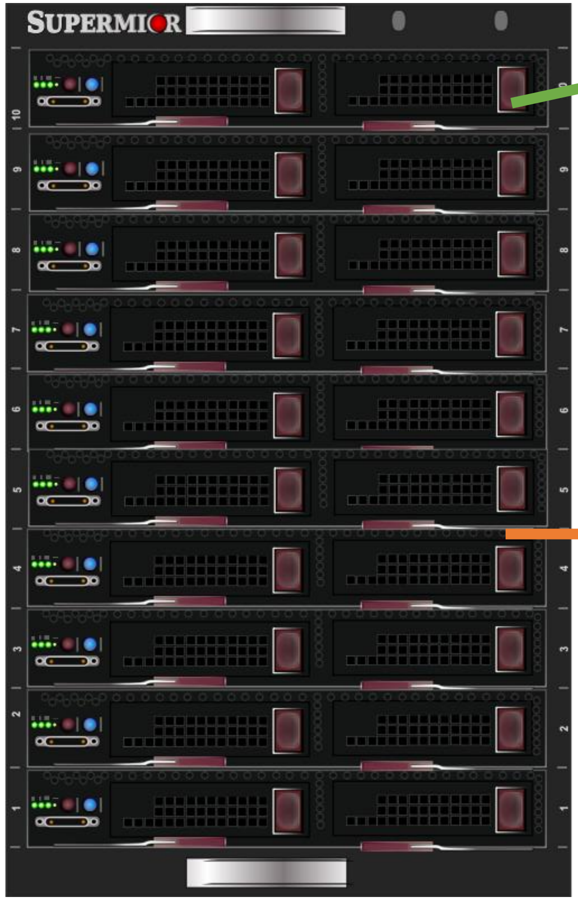
Characteristic	Value
Cycles per instruction	1.1
Ratios (percentage)	
Branch mispredict	5.0
Level 1 instruction miss*	0.4
Level 1 data miss*	0.7
Level 2 miss*	0.3
Instruction TLB miss*	0.04
Data TLB miss*	0.7

* Cache and TLB ratios are per instructions retired.

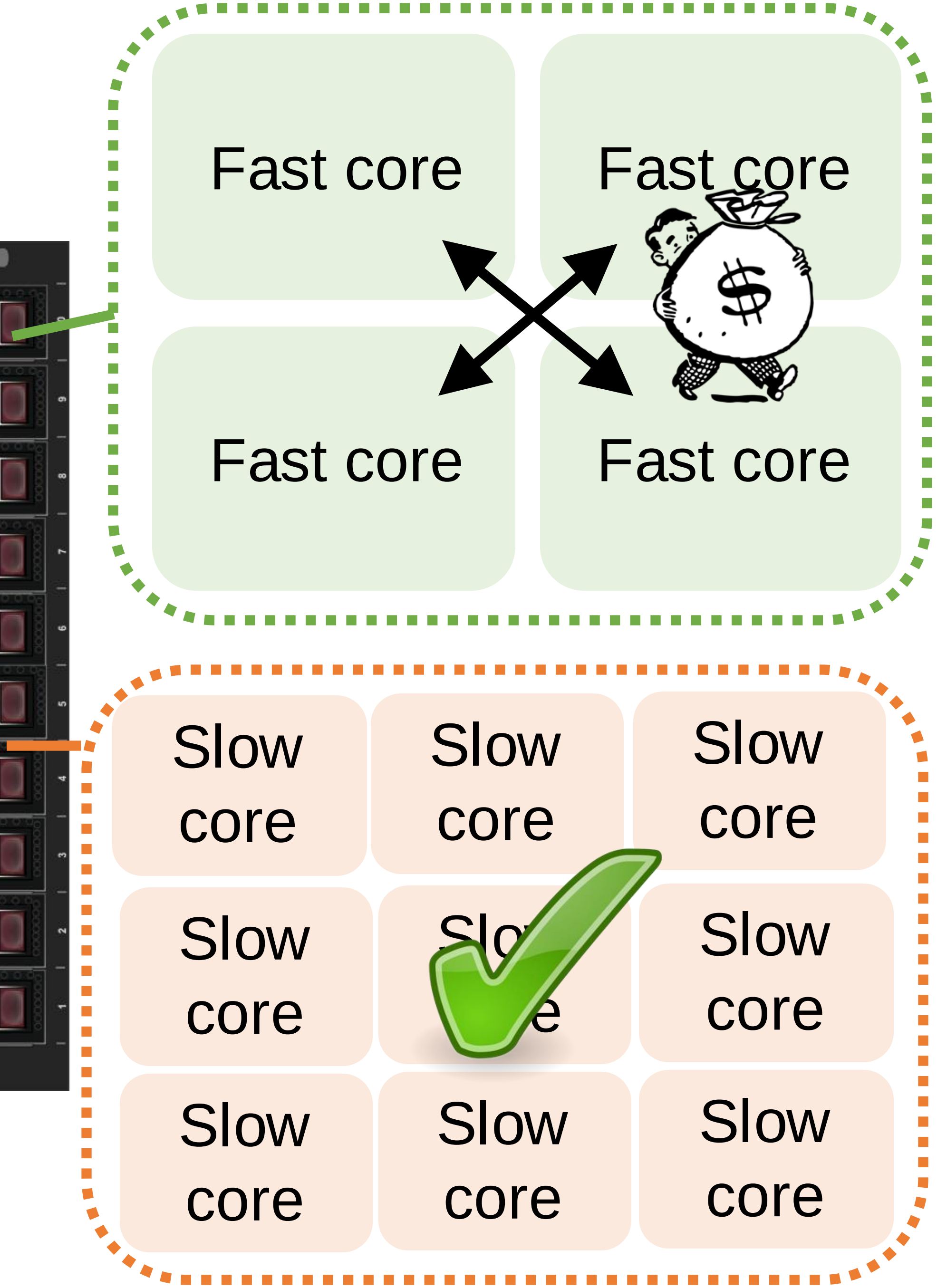
Assume: one thread == one core

How to use parallelism?

- Few fast cores with high-speed interconnect, or more slow cores?
- Cost per query processed is dominated by capital server costs
- Power draw and cooling: faster == denser



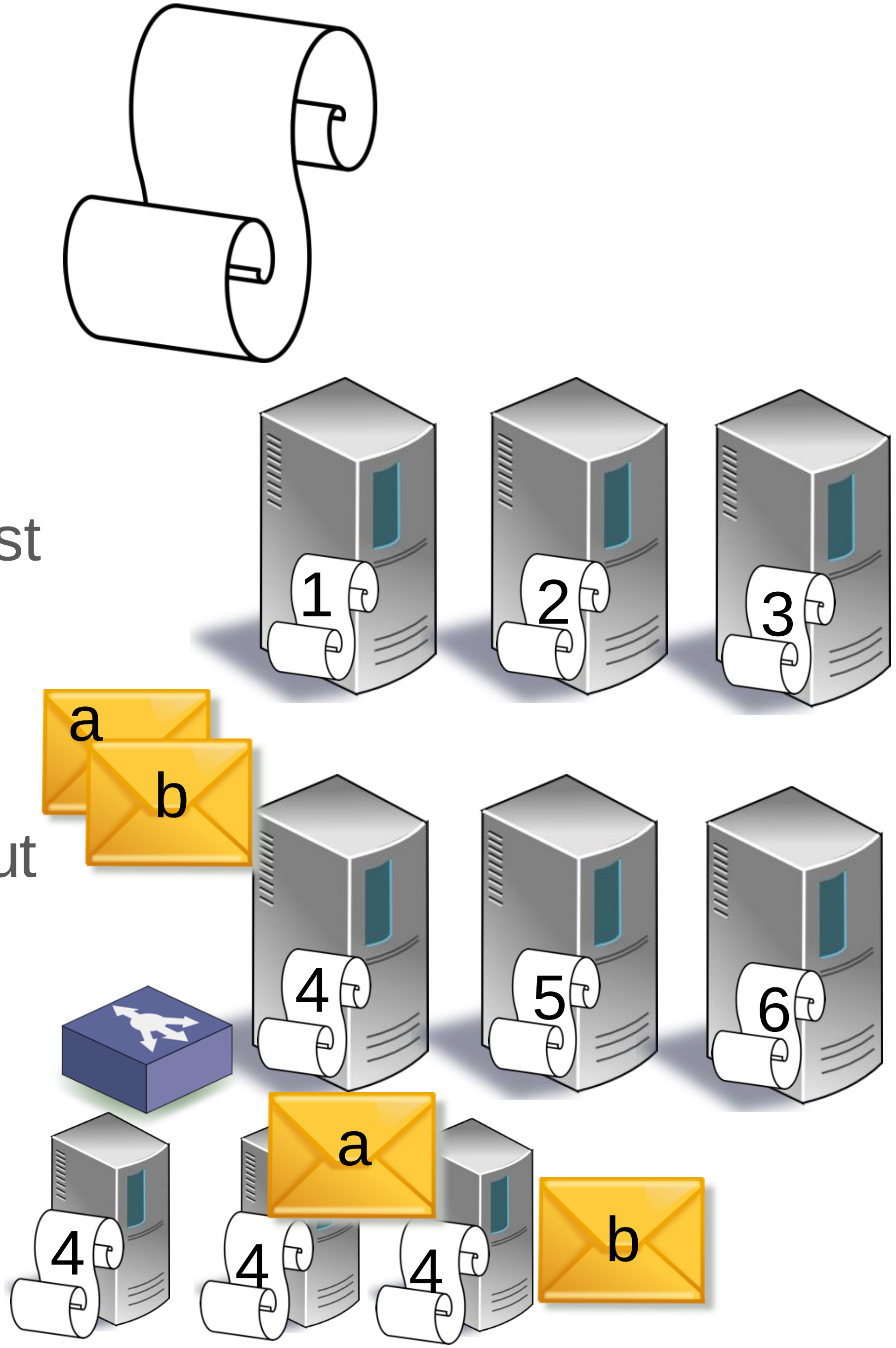
Server rack



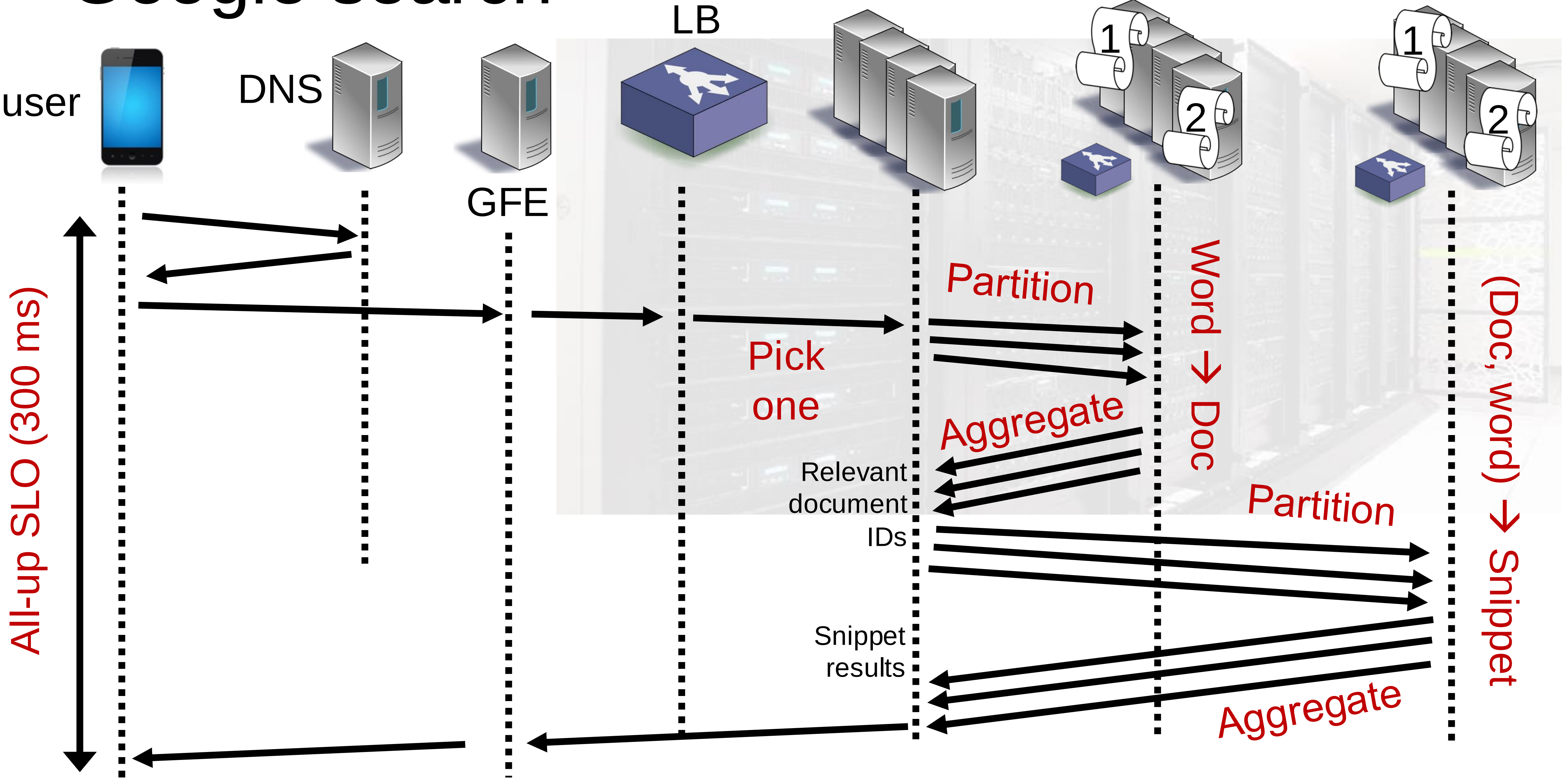
Discussion applies to both hyperthreaded or multicore

Two kinds of parallelism

- Data parallelism: independent compute over shards of data
  - Fast interconnects not as critical
  - Stateless: little coordination within a request
- Request parallelism: independent compute across requests
  - More machines for more requests
  - Shard itself can be replicated for throughput
- Need lower latency?
- Compensate slow cores with smaller shard (add more shards)
  - Each shard becomes more available
- Turn throughput into latency advantage



Google search

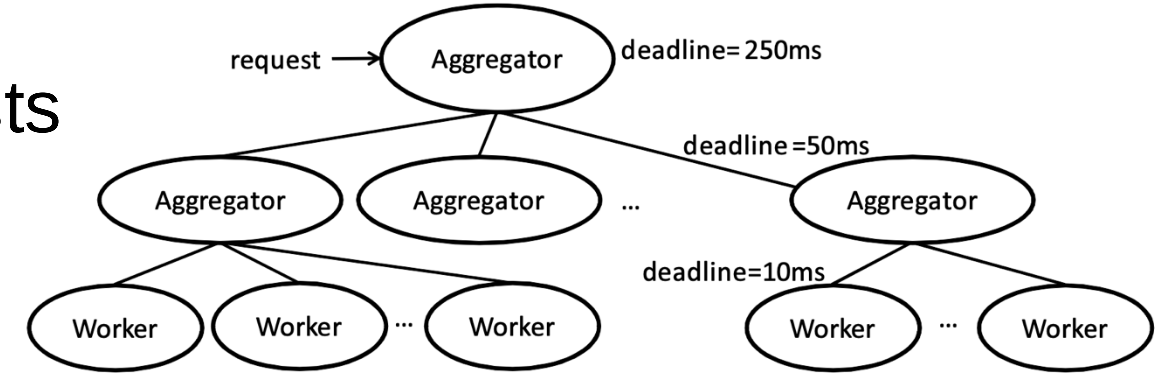


Many apps can use partition-aggregate

- Need low latency, but single-threaded low latency is hard
- Data parallelism
  - Little coordination across shards
  - Inexpensive merges across partial results from shards
- Query parallelism
  - More replicas/machines for more requests
- Use commodity (not fancy) hardware
- Turn high throughput into a latency advantage
- Focus on price per unit performance
- Significant problems: cooling for many compute servers

Tail performance becomes important

- With partition-aggregate, each machine may serve many requests within a single user-level request
- A single user sends multiple requests over a session



- Many shards are queried for a single user-level query
- Delays in any one of them can delay the entire response
  - Leaving the shard out degrades the result
- Example: 1000 shards* 10 user requests per session
  - 1 delay in 10,000 machine-level responses can be visible to a user
  - 99.99^{th} percentile delay matters
- Lots of delay on cutting the tail: hedging, duplication, …

Map Reduce

Batch processing with simple abstractions

Example: Batch data processing

- Server access log: want to get top-5 URLs visited

```
192.0.2.1 - - [07/Dec/2021:11:45:26 -0700] "GET
/index.html HTTP/1.1" 200 4310
```

- Analytics (not real time user query). How would you go about it?
- One way: shell script

```
cat /var/log/nginx/access.log
| awk '{print $7}'
| sort
| uniq -c
| sort -r -n
| head -n 5
```

Example: Batch data processing

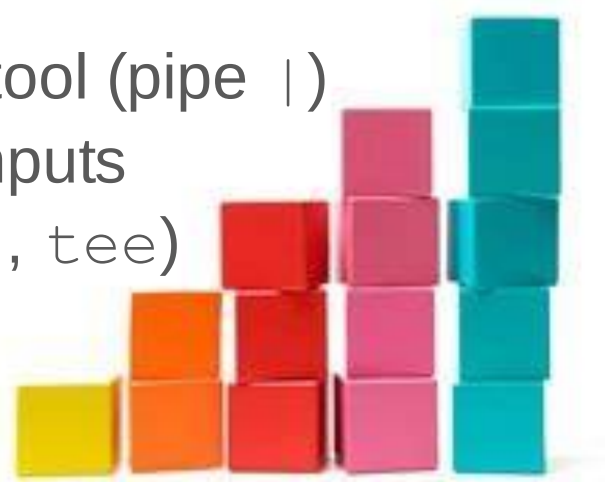
Another way: Python script

```
counts = {}
for line in open(“/var/log/access.log”):
    url = line.split()[6]
    counts[url] += 1
sorted_counts = counts.items().sort()[::-1]
print (sorted_counts[0:5])
```

Which method would you use, and why?

What do we want from implementation?

- Process large log files, even when doesn't fit into memory
- Ability to experiment with different processing steps
  - Without corrupting the original data
- Unix principles help!
- Programs/tools that do one thing well (e.g., sort)
- Separate logic from wiring
  - Any tool can produce for, or consume from, any other tool (pipe `|`)
  - Inputs come from standard input or a file. Immutable inputs
  - A choice to inspect data or write to disk anywhere (e.g., `tee`)
- Inspect output at any point (e.g., less)



Map-Reduce

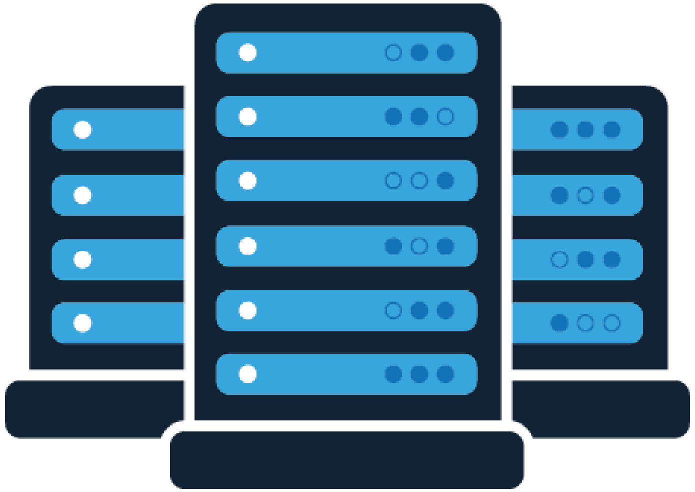
- One way to think about it: a distributed implementation of Unix processing pipelines for large batch processing
  - Large data sets: data comes from a distributed filesystem (GFS, HDFS)
  - Large computations: want to use multiple servers since data-intensive
- Examples:
  - Distributed grep, term frequencies, distributed sort
- Output?
  - A data structure, e.g., a search index
  - A set of pre-computed values for faster reads, e.g., key-value cache
  - Input to load into a traditional relational database (SQL) or view

Distributed system considerations

- Data resides on multiple machines
  - How to bring data together? How to compute with parallel machines?
  - Network bandwidth between servers is a significant consideration

MapReduce

- How to handle failures?
  - Machine failures?
  - What happens to partial computations?
    - Should we replicate compute?
  - What happens to intermediate results?
    - Should you persist it? Replicate it?

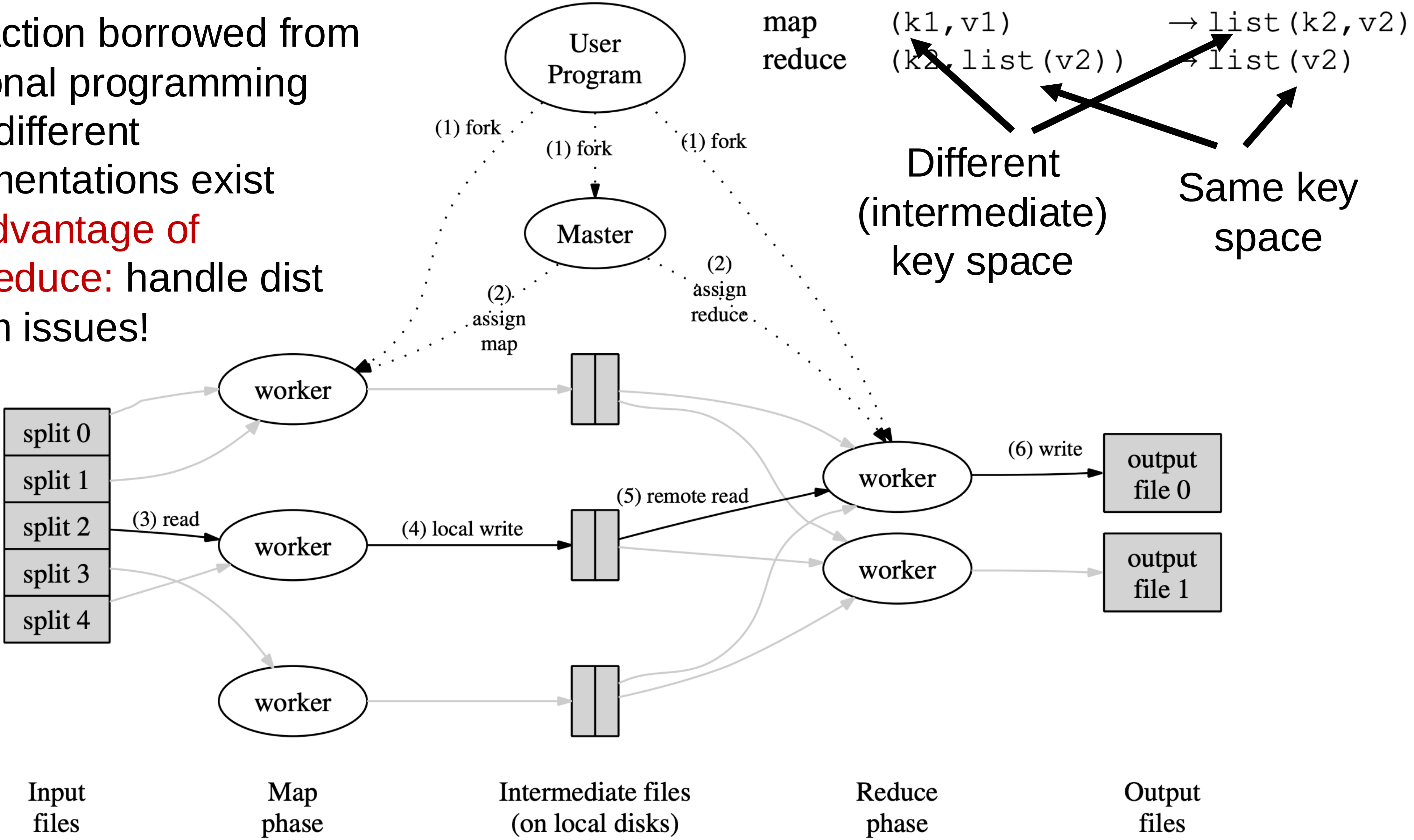


Algorithm developers == Distributed system experts?

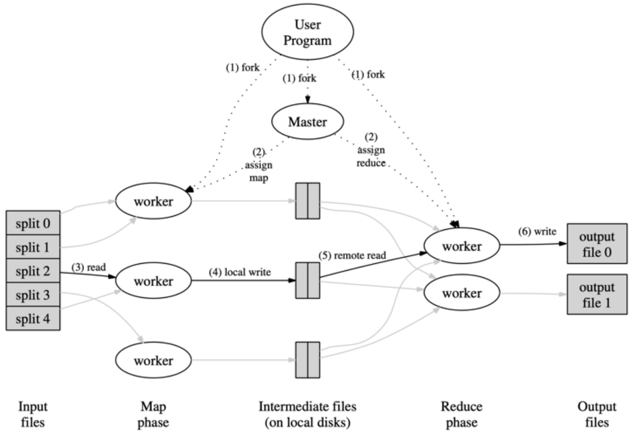
Abstraction borrowed from functional programming

Many different implementations exist

Key advantage of MapReduce: handle dist system issues!

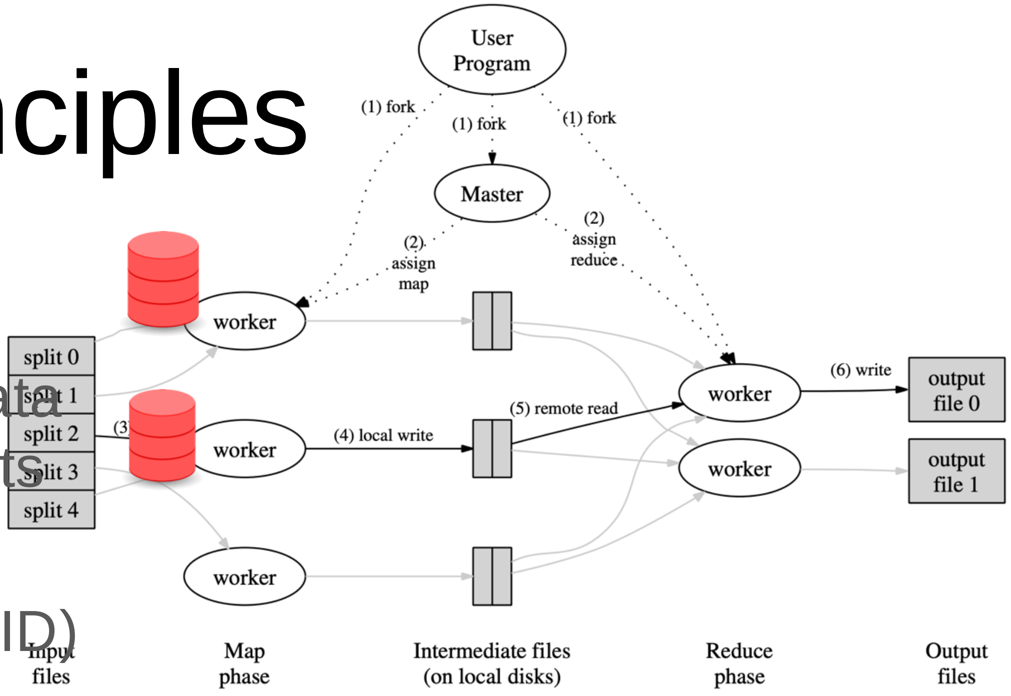


Processing steps in MapReduce



- Input data consumed from a distributed filesystem
- Master ships code to the worker node closest to data, if possible (CPU, memory constraints permitting)
- Each mapper partitions its input data by the reducer key
  - Typically through a hash function, e.g., hash (key) mod R == r
- Sort output data (per partition) by the key; run map function
- Reducers are informed of partial result at each mapper
- Reducer pulls files from mappers through RPC
- Output persisted to distributed filesystem (typically involves replication)
- Result: R output files in the DFS (one per reducer partition)

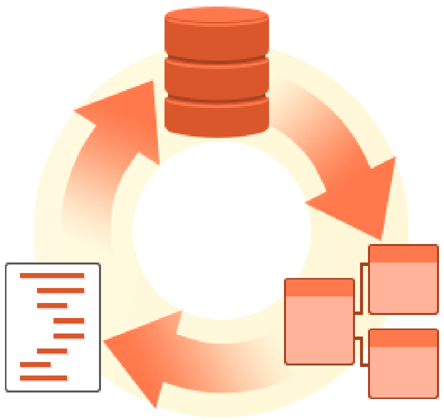
Implementation Key Principles



- Data locality
  - Reduce network bandwidth: ship code to data
  - Locally persist (not DFS) intermediate results
- Handle failures by re-doing compute
  - No fancy hardware fault tolerance (e.g., RAID)
  - Mapper failure: restart map job
  - Assume deterministic operations
  - Reducer failure (after completion): no problem (DFS)
- Identify and skip shards with deterministic faults
- Mitigate stragglers through eager replication of compute close to job completion
- Combiners at mapper: preliminary reduce for associative and commutative functions

More examples of using map-reduce

- Database Joins
  - Example: user activity (e.g. URLs) with user information (e.g. age)
- Grouping (GROUPBY) aggregations:
  - Count, sum, etc
  - Creating the sequence of events in a user session, determining whether e.g. a new version of a web page resulted in better sales
- Large distributed sorting
- Output sorting after mapper: important!



Building on Map-Reduce: (1) Workflows

- One Map-Reduce job isn’t usually enough
- Google web search index: pipeline of 10 jobs; recommendation systems: 50—100
- Workflows: Chains of map-reduce jobs
  - E.g., one MR for counting requests by URL; another to sort count
- Explicit output files from each?
  - Like writing to file at the end of each tool in Unix pipeline
  - Materialization of the intermediate results needed?
- Stragglers make workflows slower
- Separate systems needed just to orchestrate the workflows correctly

Building on Map-Reduce: (2) Dataflow

- Dataflow engines: handle the entire workflow
  - “Operators”: chain map-reduce functions
  - Only persist intermediate outputs to DFS when necessary
  - Chain reducers (no explicit mappers) when the key is the same
  - Don’t wait for stragglers of the previous job

- Stream Processing
  - Incremental execution of batch jobs when new data arrives

- Selectively materialize or recompute intermediate results
  - Lineages (RDD/Spark) or checkpoint