GFS & MapReduce

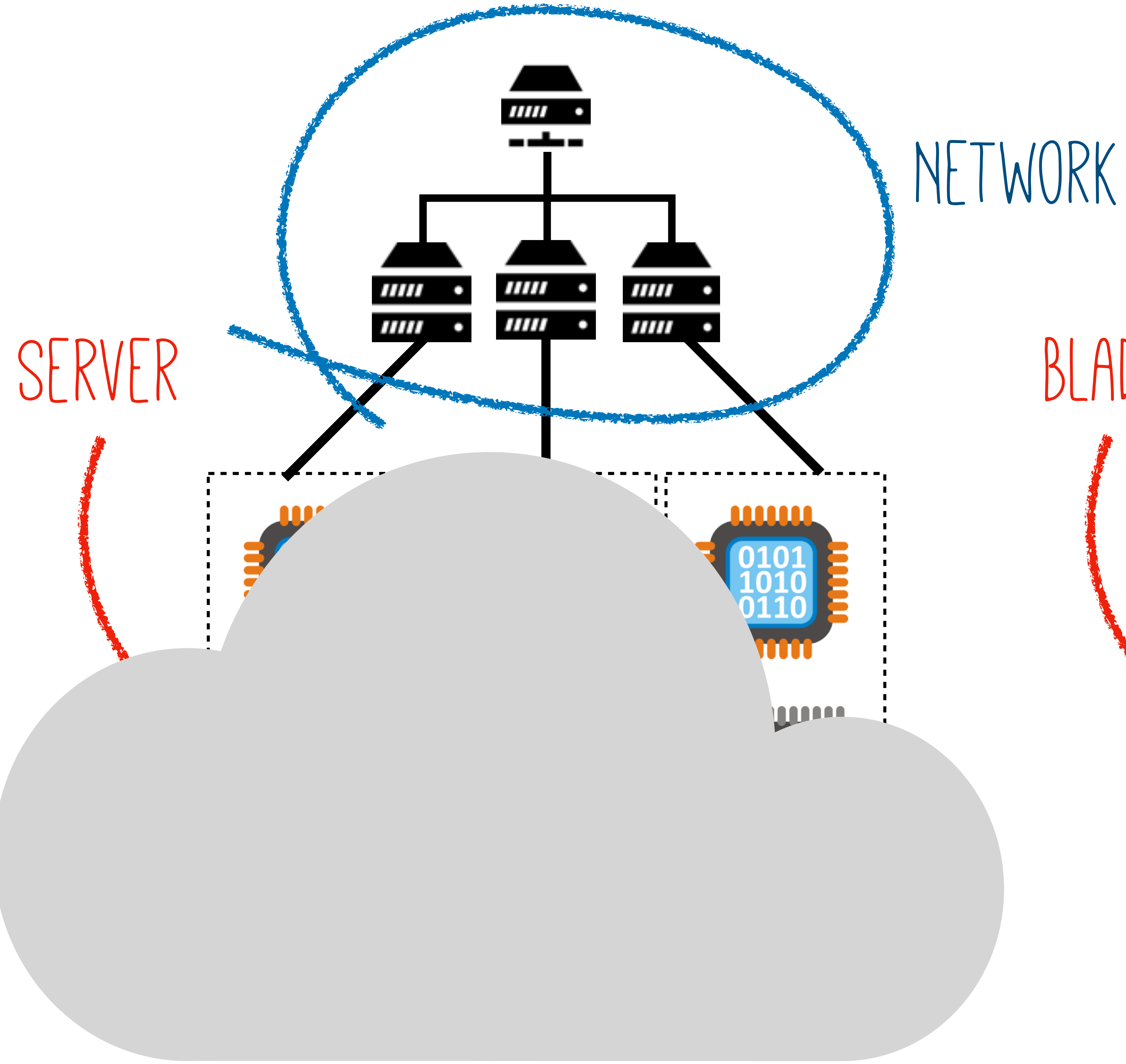
CPSC 438/538: Big Data Systems

Anurag Khandelwal

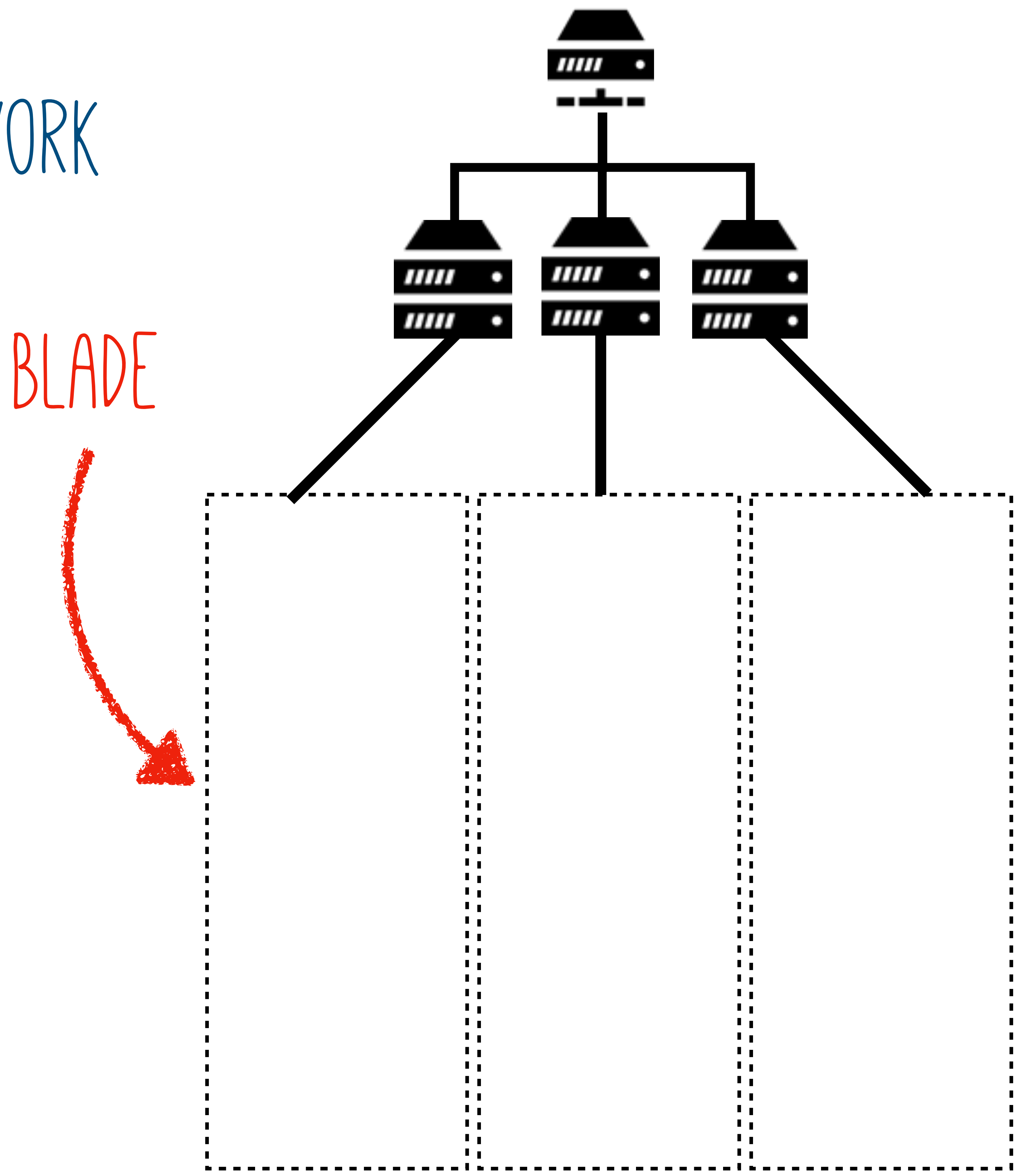
Yale

Possible Research Projects

Serverless and Disaggregated Architectures



Cloud Architectures Today



Disaggregated/Serverless Architectures

How do we design OS for such architectures?

How can traditional applications run on such architectures?

Possible Projects

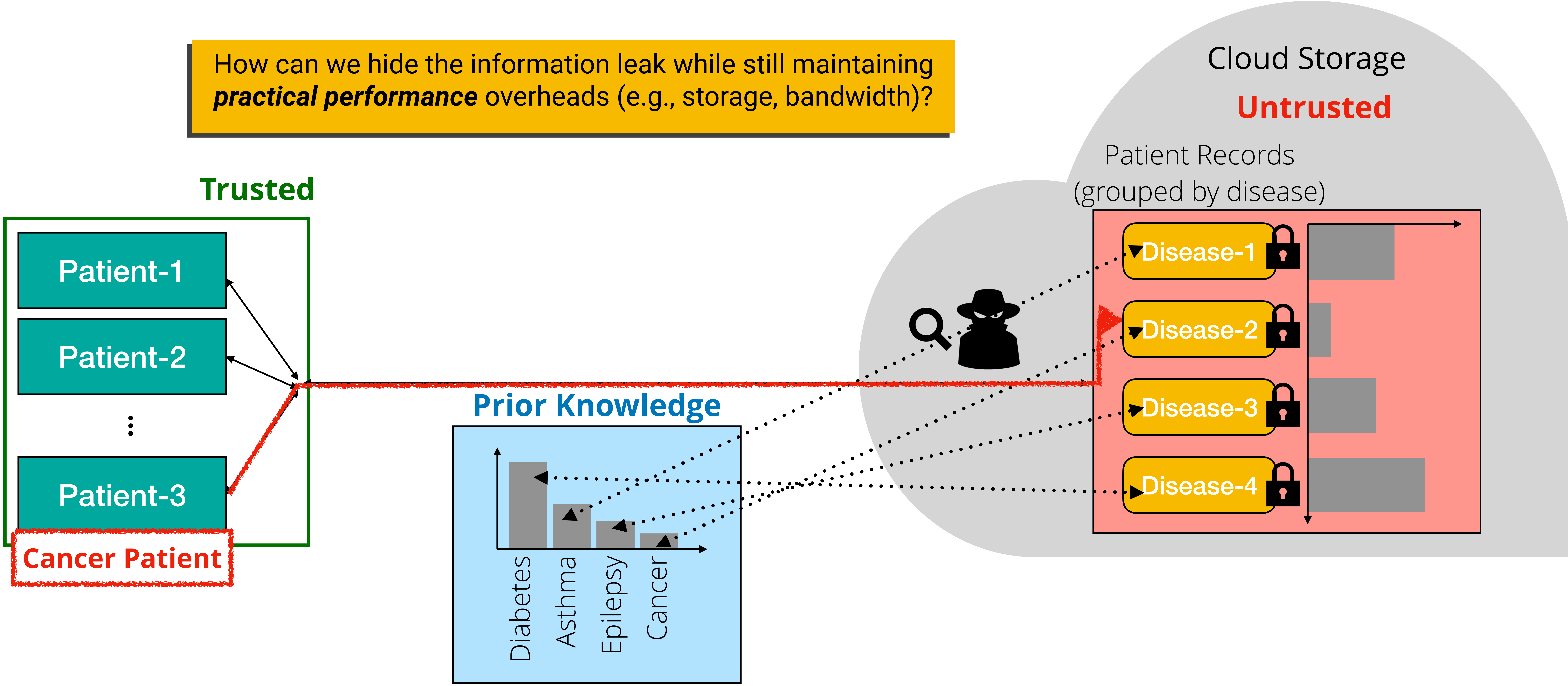
- **OS-level design [see MIND, SOSP'21]**
  - Realizing coherence protocols on the Disaggregated Computer
  - Designing a memory allocator for disaggregated memory
  - Designing a threading library (with IPCs, synchronization, signals, etc.)
  - Designing a file-system and file buffer cache for Disaggregated OS
  - Designing a network abstraction for the disaggregated computer
  - Designing a virtualization layer (e.g., VMs, containers) on Disaggregated OS
  - Designing shared-memory and threading for accelerators (e.g., GPUs)

Possible Projects

- **Application-level design [see MIND, SOSP'21]**
  - Library to accelerate pointer-chasing over disaggregated memory
  - Integrate language support into disaggregated OS (e.g., garbage collection, rust-style isolation)
  - Optimize application for disaggregated architecture: 5G processing
  - Provide support for disaggregated memory for Map-Reduce applications

Secure Data Stores

How can we hide the information leak while still maintaining ***practical performance*** overheads (e.g., storage, bandwidth)?



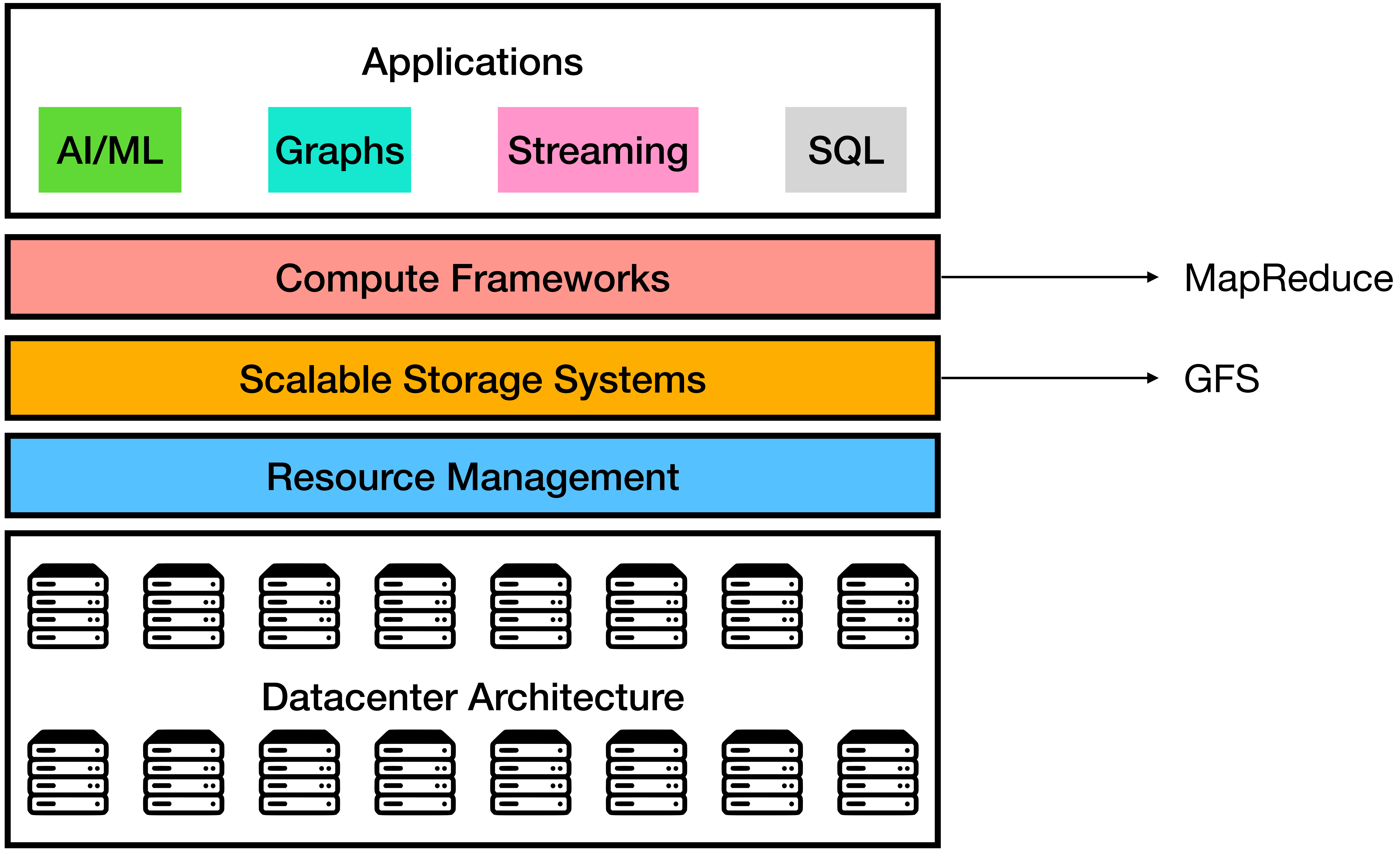
Possible Projects

- **Secure Storage Design [See Pancake, USENIX Security'20]**
    - How do we hide accesses for data structures (e.g., trees, graphs, etc.)?
    - How do we hide network communications in Map-Reduce frameworks?
    - How do we hide length of data items?
    - How do we hide entropy of compressed data items?
- All of these projects will require reasoning about formal security guarantees…

Administrivia

- **By end of this week (Friday, Sep 10th)…**
  - Form groups
  - Fill out presentation preferences
- **Project Deadlines:**
  - Initial Proposal: **Sep 27th** (*Must confirm project with me* ***before*** *submitting report*)
    - This includes literature survey, so reach out to me sooner rather than later!
  - Mid-term report: **Oct 29th**
  - Final Report: **Dec 10th**
  - Final Presentation (Poster): **Dec 8th**

Today's Agenda



Big Data System Stack

What was the problem being addressed?

GFS

- Store **large volumes of data**
  - **Efficiently**
  - With **fault-tolerance**
  - In a **scalable manner**

MapReduce

- Process **large volumes of data**
  - **Efficiently**
  - With **fault-tolerance**
  - In a **scalable manner**

Common Themes

Failures are a part of Datacenter life

Concurrency/Parallelism is key to scale

Batch processing model: *throughput* over *latency*

Simplicity & **flexibility** over generality

Storage-Compute Co-design

How we store data → how we can operate on them

How we interact with data → how store them

Deciding factor:
Workload

Google File System

Workload GFS was designed for?

- **Modest number of huge files**
  - A few million 100MB or larger files
- **Most writes are appends**
  - Some are never read again (*cold* data)
- **Most reads are sequential**
- **High sustained bandwidth (throughput) is more important than latency**
  - *Not* interactive/user-facing

GFS Design Decisions

- **Files = list of chunks**
  - Fixed sized (64MB)
- **Reliability through replication**
  - Chunks replicated across 3+ chunk servers
- **Single master to coordinate access and keep metadata**
  - Simple centralized management
- **No data caching**
  - Little benefits due to large datasets and streaming reads
- **Familiar interface, but customized API**
  - Snapshot & record append

Architecture

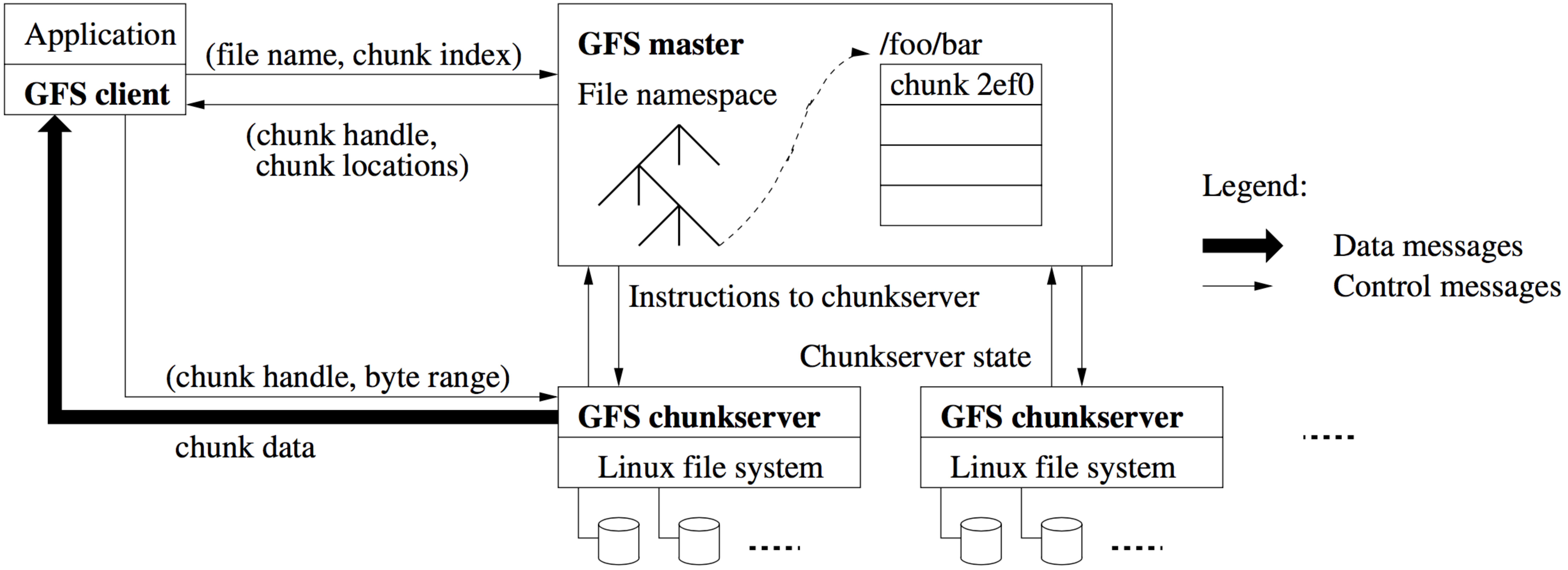


Figure 1: GFS Architecture

Single Master holds Metadata

- **Problems?**
  - Single point of failure
  - Scalability bottleneck
- **GFS Solutions?**
  - Shadow master
  - Minimize master involvement
    - Never move data through master, only used for metadata
    - Large chunk to decrease metadata
    - Master delegates authority to primary replicas in data mutations (chunk leases)

Chunkservers hold Actual Data

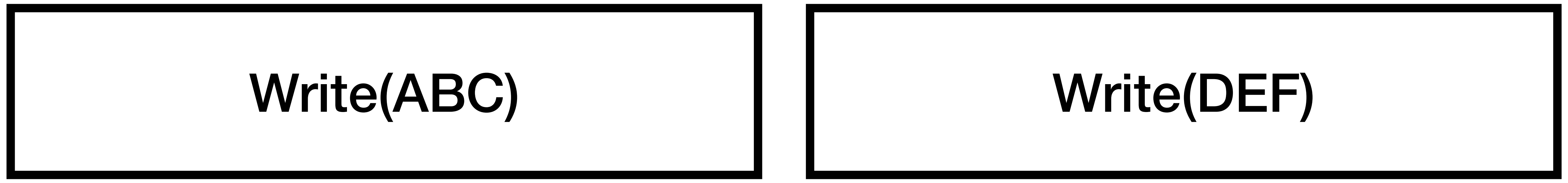
- **Many chunkservers under one master**
  - Free to join and leave
- **Stores actual data**
- **Report chunk locations to master**
  - Refresh master on join
- **Checksums for data integrity**

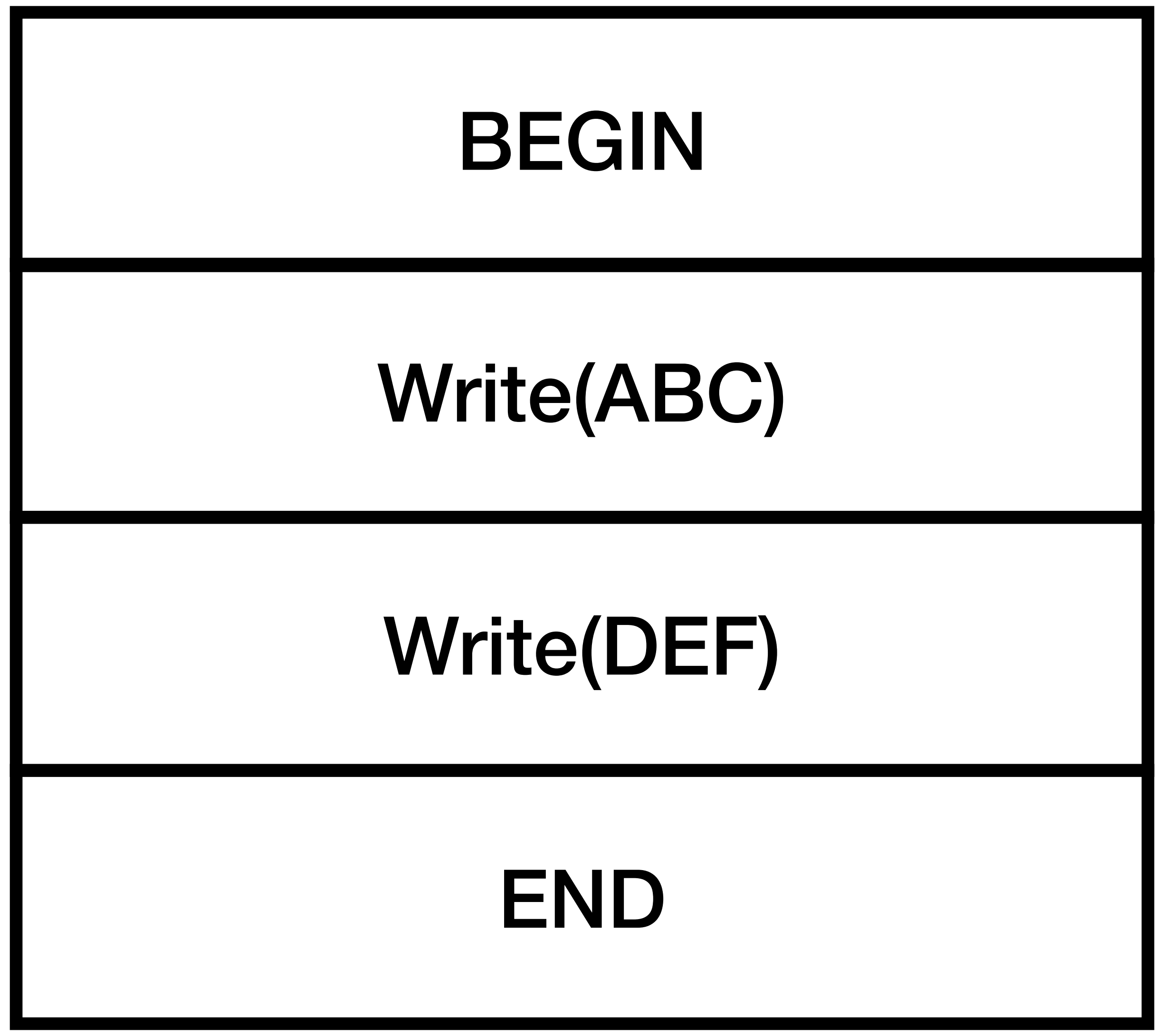
Metadata

- **Metadata is stored on the master**
  - File and chunk namespaces
  - Mapping from files to chunks
  - Locations of each chunk’s replicas
- **All in-memory (64B per chunk)**
  - Fast
  - Easily accessible

Metadata

- **Master has an operation log for persistent logging of critical metadata updates**
  - Persistent on local disk
  - Replicated to the shadow master(s)
  - Checkpoints for faster recovery





Mutations

- **Mutation = write or append record**
  - Must be done for all replicas
- **Goal: minimize master involvement**
- **Lease mechanism**
  - Master picks one replica as primary and gives a “lease” for mutations
  - Primary defines a serial order of mutations
  - All replicas follow this order
- **Data flow decoupled from control flow**

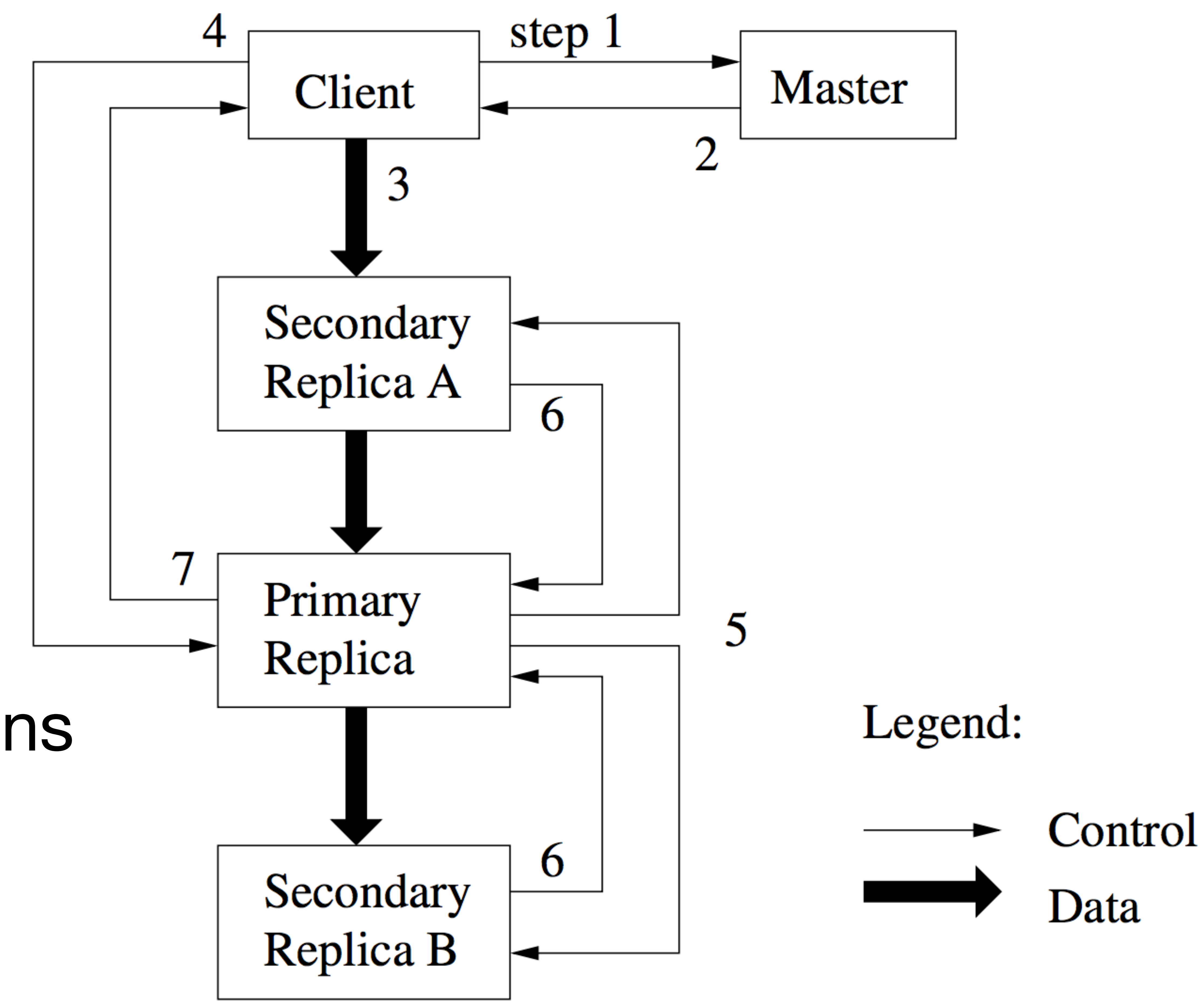


Figure 2: Write Control and Data Flow

Atomic Record Append

- **GFS appends it to the file atomically *at least once***
  - Primary picks the record offset
  - Works for concurrent writers
- **Used heavily by Google applications**
  - For files that serve as multiple-producer/single consumer queues
  - Merge results from multiple writers to one file

Fault-tolerance

- **High availability**
  - Fast recovery
    - Master and chunks server can restart in a few seconds
  - Chunk replication
    - Default is three replicas
  - Shadow masters
- **Data integrity**
  - Checksum every 64kB block in each chunk

Consistency Model

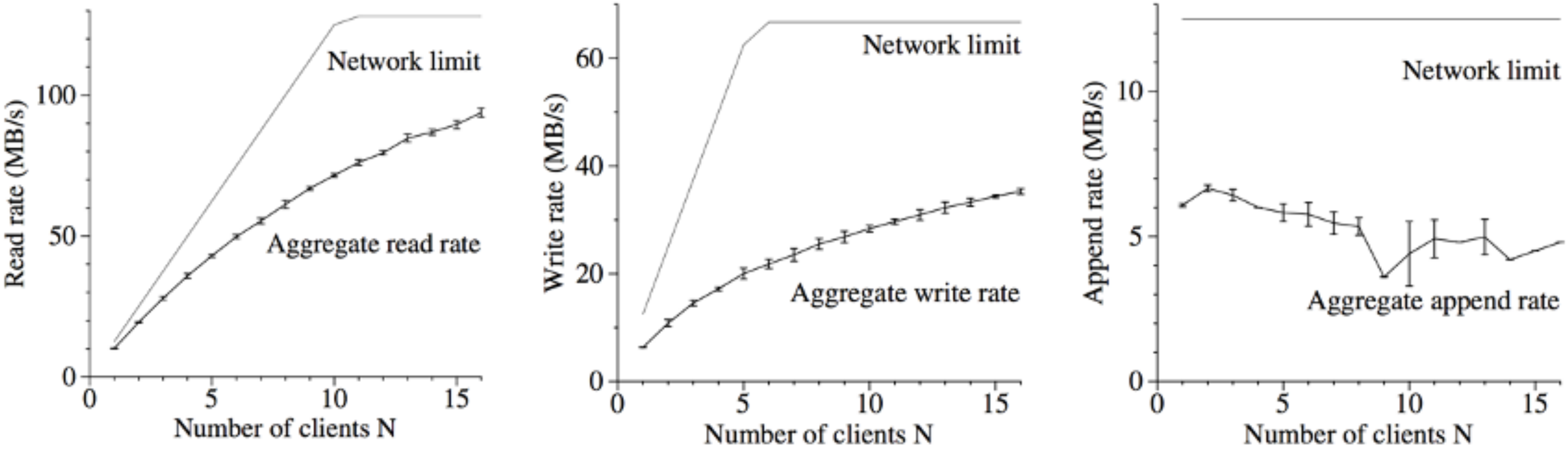
- **What is consistency?**
- ***For now:*** when there are multiple copies of data, do different clients see the copies as same or different? Do they reflect the latest updates or not?
  - Always same view across copies with latest updates: ***strongly consistent***
- We will discuss a more precise characterization later in the course when we cover CAP Theorem…

Consistency Model

<table>
<tr><th></th><th>Write</th><th>Record Append</th></tr>
<tr><td>Serial success</td><td>defined</td><td rowspan="2">defined interspersed with inconsistent</td></tr>
<tr><td>Concurrent successes</td><td>consistent but undefined</td></tr>
<tr><td>Failure</td><td colspan="2">inconsistent</td></tr>
</table>

- **Consistency level**
  - Defined (everyone sees the same, up-to-date data)
  - Consistent (everyone sees the same data, but may not be up-to-date)
  - Inconsistent (not everyone sees the same data)
- **Implications for applications**
  - Rely on appends rather than overwrites
  - Checkpoint

Evaluation: Microbenchmarks



Evaluation: Real-world clusters

- **Cluster A**
  - Research & Development
  - A few MBs to a few TBs of data
  - Tasks run up to hours
- **Cluster B**
  - Production use
  - Continuously generate and process multi-TB data
  - Long running tasks

Storage & Metadata

Cluster	A	B
Chunkservers	342	227
Available disk space	72 TB	180 TB
Used disk space	55 TB	155 TB
Number of Files	735 k	737 k
Number of Dead files	22 k	232 k
Number of Chunks	992 k	1550 k
Metadata at chunkservers	13 GB	21 GB
Metadata at master	48 MB	60 MB

Read/Write Rate

Cluster	A	B
Read rate (last minute)	583 MB/s	380 MB/s
Read rate (last hour)	562 MB/s	384 MB/s
Read rate (since restart)	589 MB/s	49 MB/s
Write rate (last minute)	1 MB/s	101 MB/s
Write rate (last hour)	2 MB/s	117 MB/s
Write rate (since restart)	25 MB/s	13 MB/s
Master ops (last minute)	325 Ops/s	533 Ops/s
Master ops (last hour)	381 Ops/s	518 Ops/s
Master ops (since restart)	202 Ops/s	347 Ops/s

Recovery Time

- **Kill one chunkserver**
  - 15000 chunks containing 600GB data
  - All chunks restored in 23.2 minutes
- **Kill two chunkservers**
  - Each with 16000 chunks and 660GB data
  - Results in 266 single replicas
  - Single replicas restored to at least 2x within 2 mins

Discussion

- **Strengths, Weaknesses?**
- **What issues are you likely to encounter if you increase the scale from O(100s of TB) to O(100s of PB)?**

MapReduce

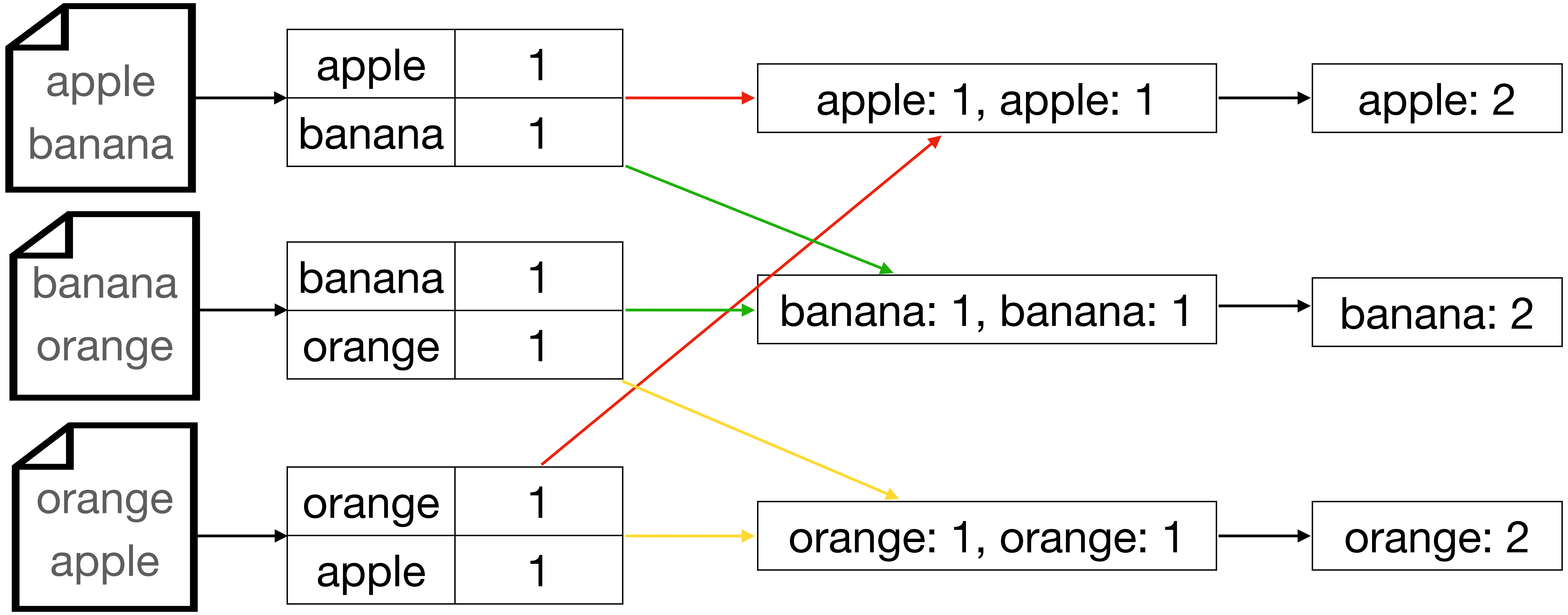
Design Decisions

- **Simple interface & programming model is often sufficient**
  - Map & Reduce
- **Fault-tolerance & scalability should come without user effort**
- **Deterministic Work**
  - Rerunning will result in same output
- **Level of parallelism dictated primarily by the underlying filesystem**
  - Each map task works on one chunk of data on GFS
  - A bit more control over reduce tasks

Programming model: Word Count

```
Map(String key, String value):
  // key: document name
  // value: document contents
  For each word w in value:
    EmitIntermediate(w, “1”)
```

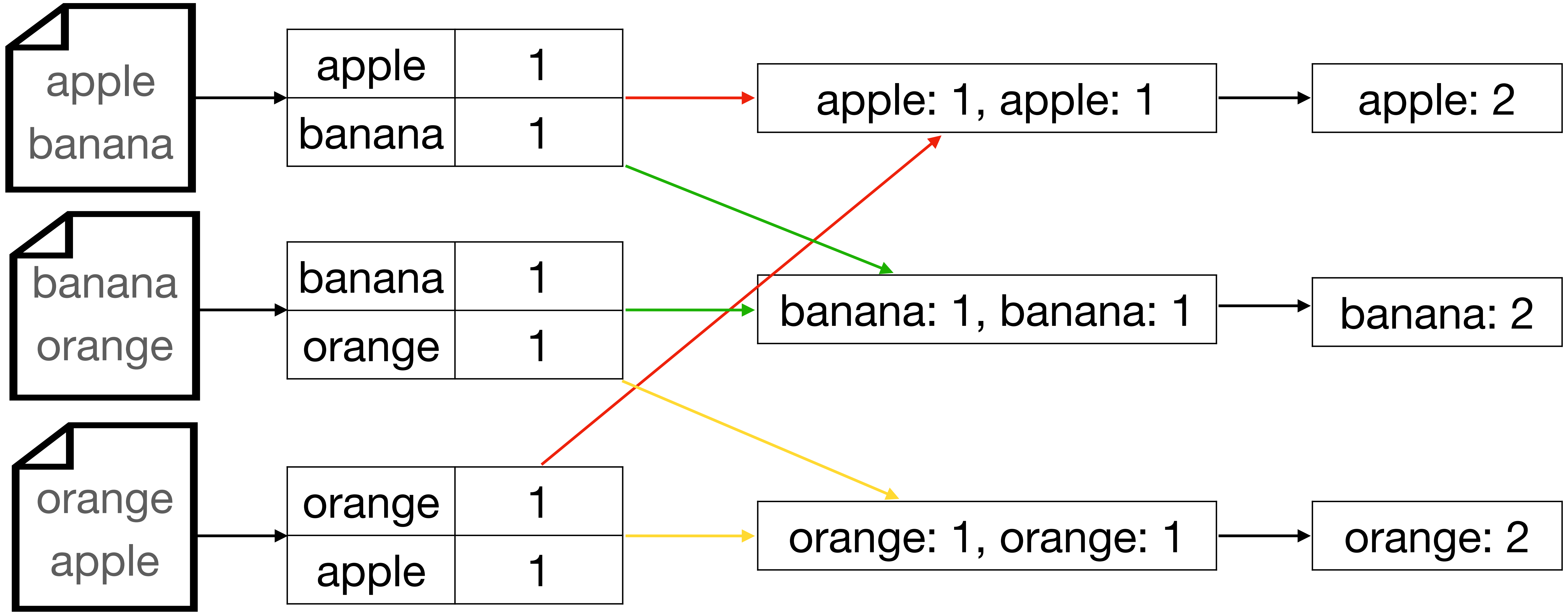
```
Reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0
  For each v in values:
    Result += ParseInt(v);
  Emit(AsString(result));
```



Programming model: Generalization

```
Map(String key, String value):
  // key: document name
  // value: document contents
  For each word w in value:
    EmitIntermediate(w, "1")
```

```
Reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0
  For each v in values:
    Result += ParseInt(v);
  Emit(AsString(result));
```

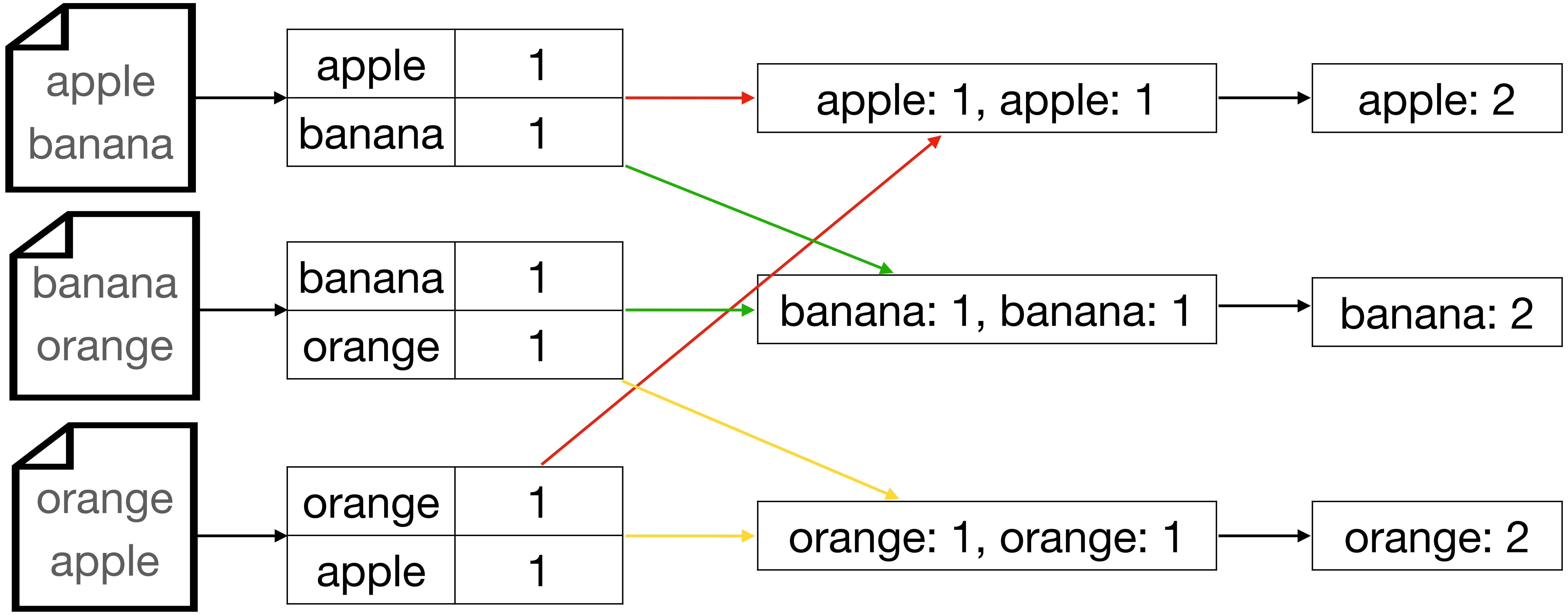


- **Partition -> Shuffle -> Collect**
  - **Remapping:** (docID, content) -> (word, frequency)
  - **Reduction:** Many (word, frequency) -> small number of (word, frequency); Duplicate keys -> Distinct keys

Things to worry about in a real system

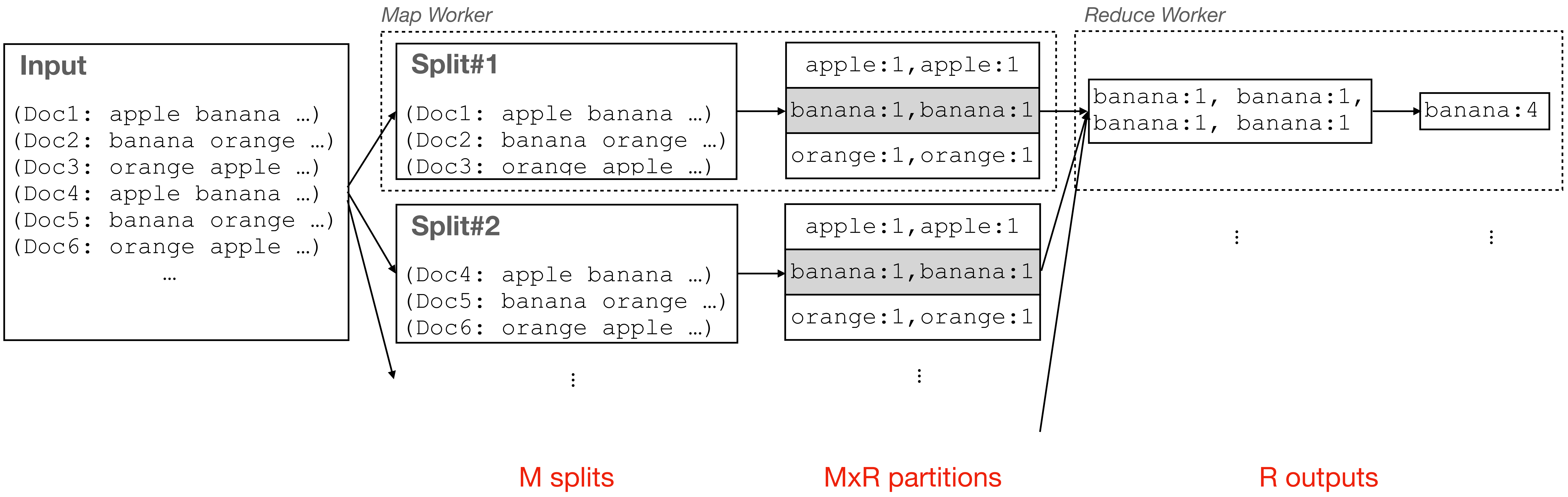
```
Map(String key, String value):
  // key: document name
  // value: document contents
  For each word w in value:
    EmitIntermediate(w, “1”)
```

```
Reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0
  For each v in values:
    Result += ParseInt(v);
  Emit(AsString(result));
```

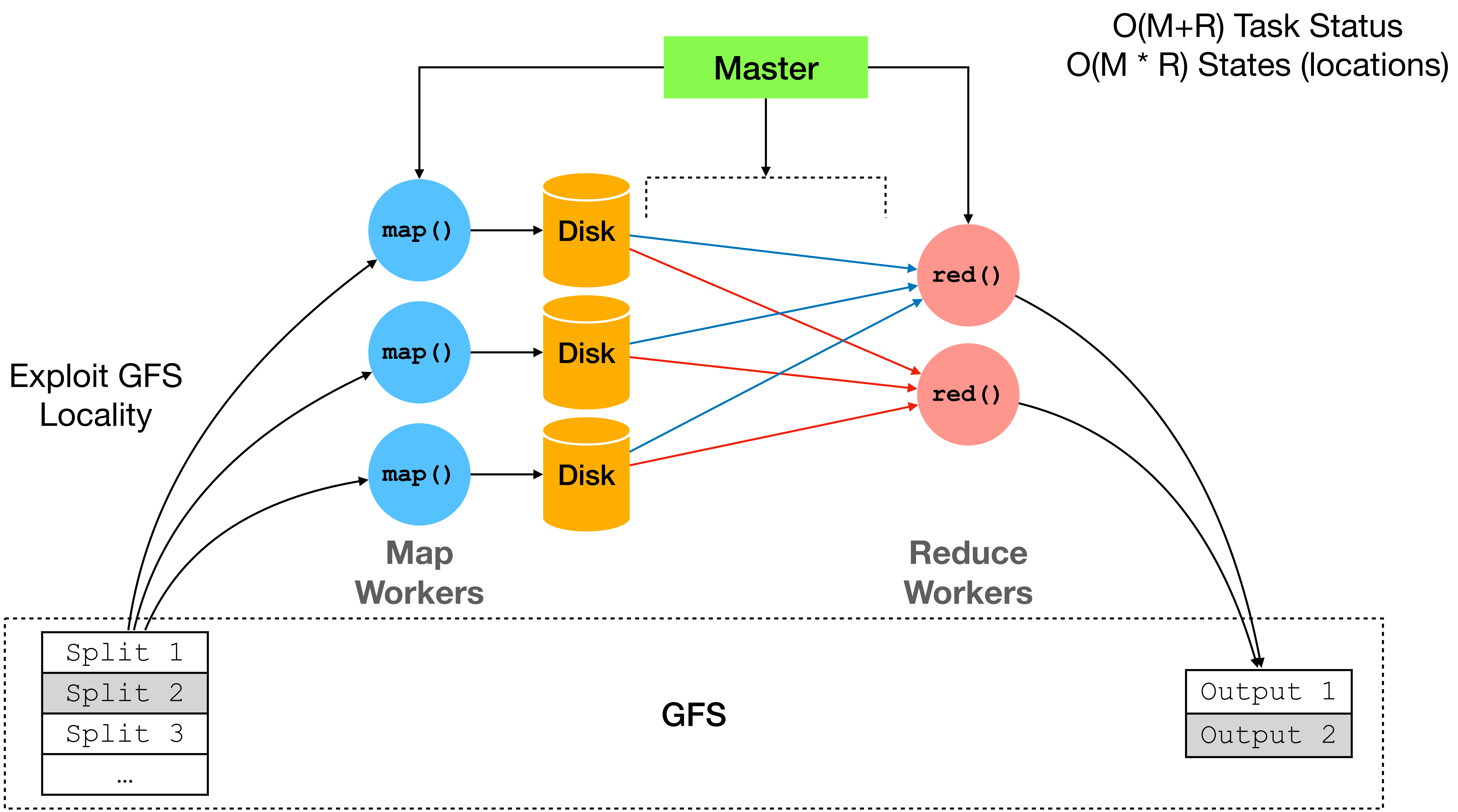


- **Granularity:** How to size inputs for map and reduce tasks? How many map/reduce tasks?
- **Scheduling:** How to (i) assign workers, (ii) coordinate between map/reduce workers?
- **Fault-tolerance:** What if something fails?

“Real” MapReduce with WordCount



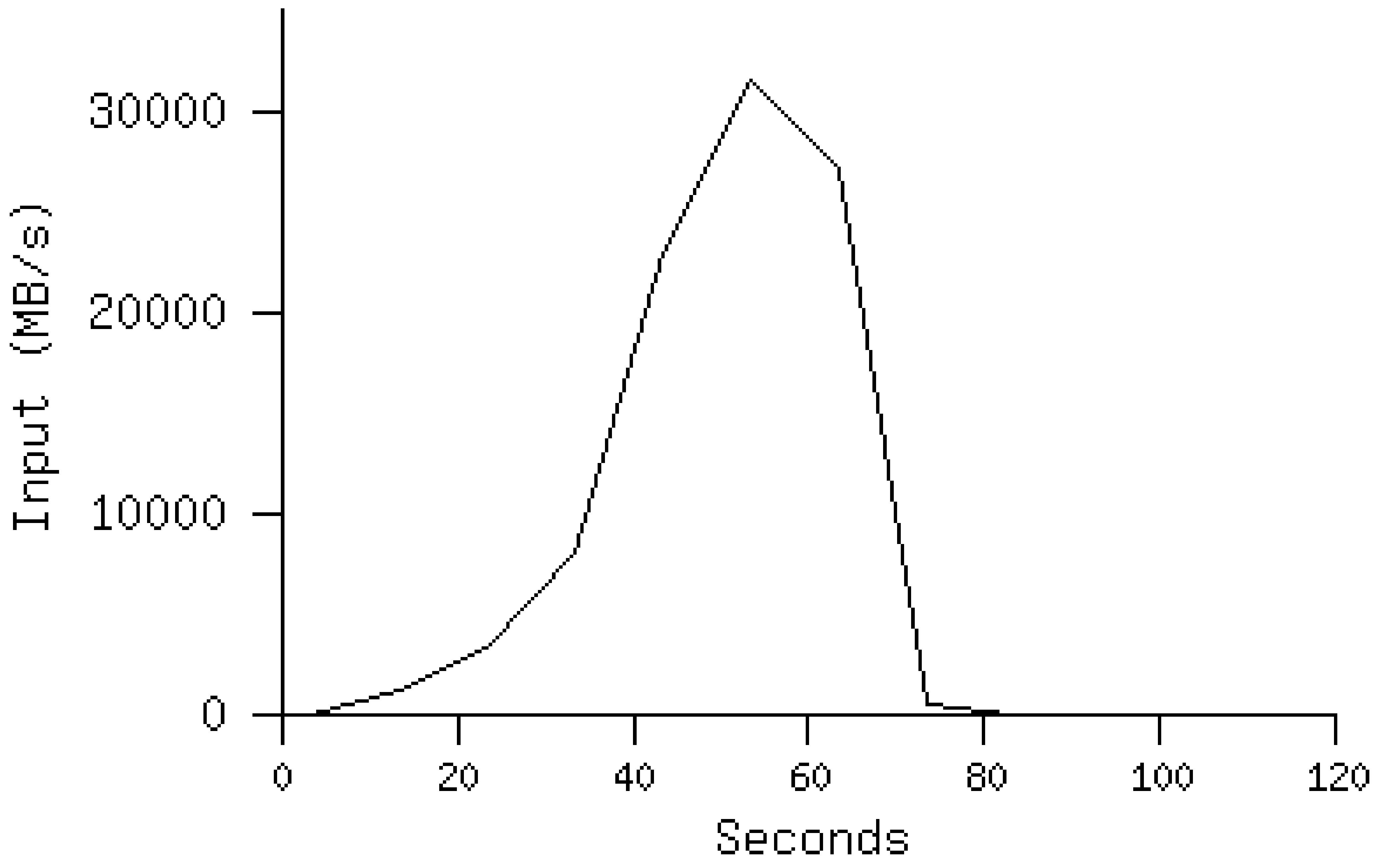
System Overview



Refinements

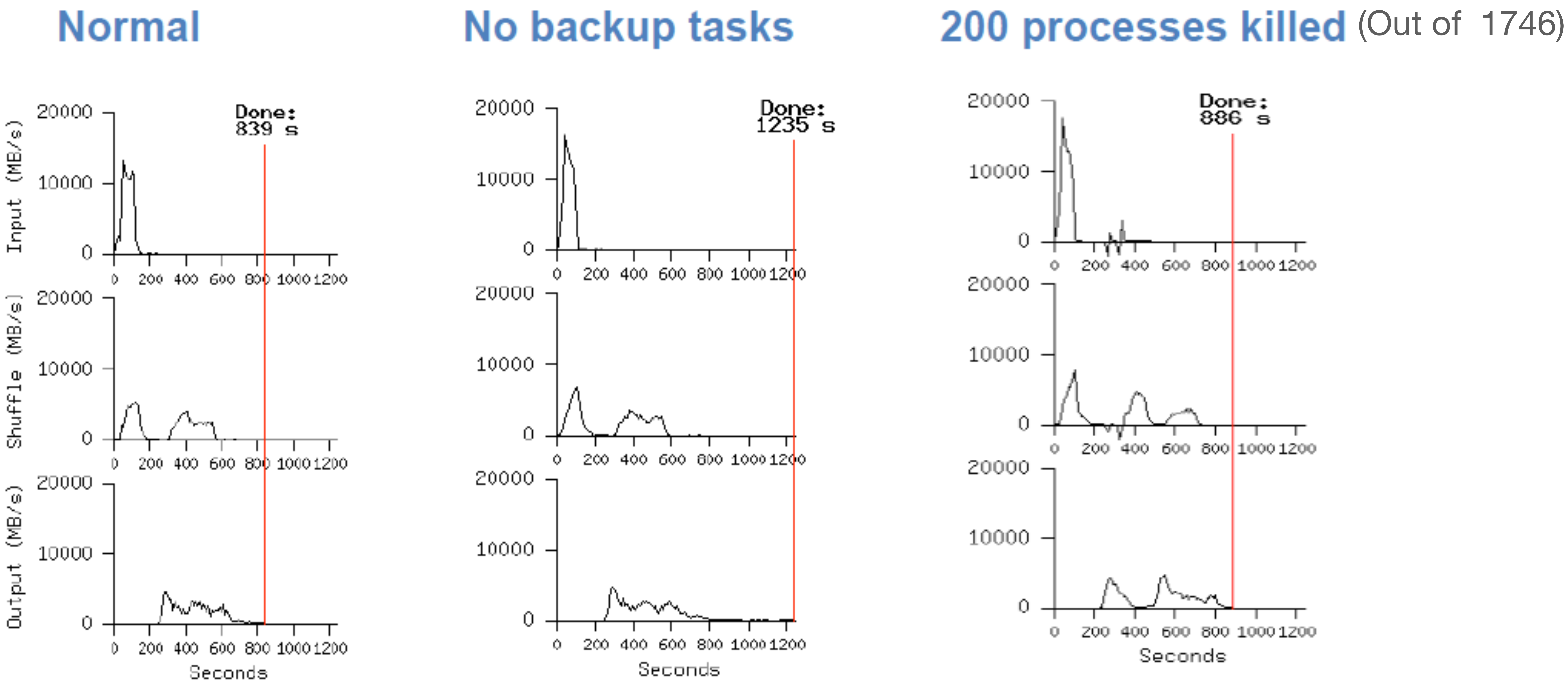
- **Performance**
  - Stragglers: backup tasks
  - Skipping bad records: catch > 1 failures on a specific record
- **Utilities**
  - Sort inside partitions — faster lookup for reduce workers/output consumers
  - Combiner function — “partial reduce”
  - Health, status info
  - Counters
  - Debugging info.

Performance: MR_Grep



- **Locality optimization helps**
  - 180 machines read 1 TB of data at peak ~31 GB/s
  - Without this, rack switch would limit to 10 GB/s
- **Startup overhead is significant for short jobs**

Performance: MR_Sort



M=15000, R=4000

Techniques Used

Technique	Used In
Replication	GFS
Erasure Coding	GFS
Sharding/partitioning	MR tasks, GFS splits
Load balancing	Automatic due to partitioning
Health Checks	MR, GFS
Integrity Checks	MR, GFS
Compression	GFS
Eventual Consistency	GFS Master
Centralized controller	MR, GFS Masters
Redundant Execution	MR

Discussion

- **Strengths, Weaknesses?**
- **Consider:** Indexing pipeline where you start with HTML documents. You want to index the documents after removing the most commonly occurring words.
  - 1. Compute most common words.
  - 2. Remove them and build the index.
  - What are the main shortcomings of using MapReduce?

Summary

- **Exploit domain/workload knowledge** in system design
- Strive for **simplicity**
- **Fault-tolerance** & **scalability** are not optional