Database Management Systems - I, CS 157A

Advanced Topics of Interest: “MapReduce and SQL”

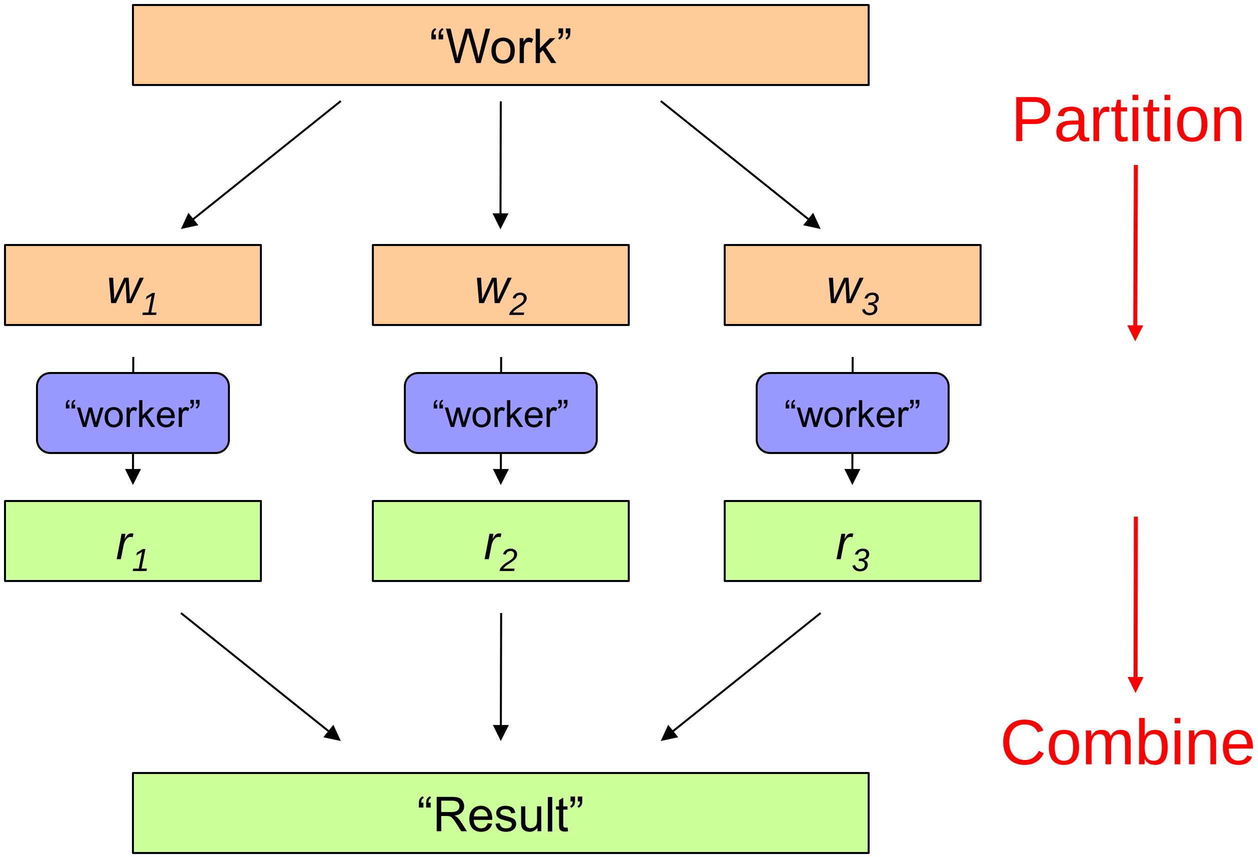
Outline

- MapReduce and SQL

MapReduce and SQL

Introduction

- It is all about divide and conquer



Introduction

- **Different workers:**
  - Different threads in the same core
  - Different cores in the same CPU
  - Different CPUs in a multi-processor system
  - Different machines in a distributed system

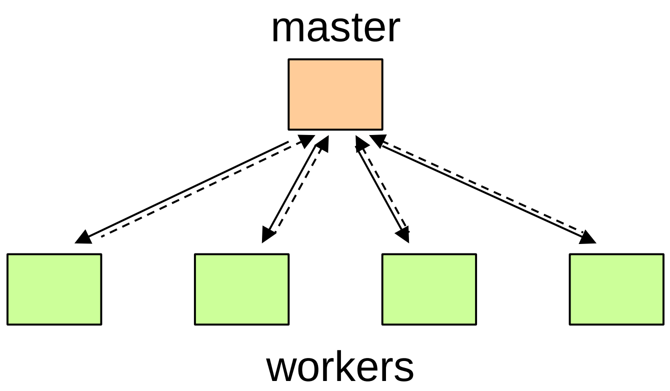
- **Parallelization Problems:**
  - How do we assign work units to workers?
  - What if we have more work units than workers?
  - What if workers need to share partial results?
  - How do we aggregate partial results?
  - How do we know all the workers have finished?
  - What if some workers die?

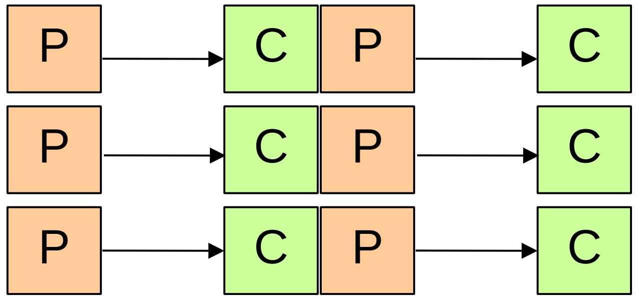
Introduction

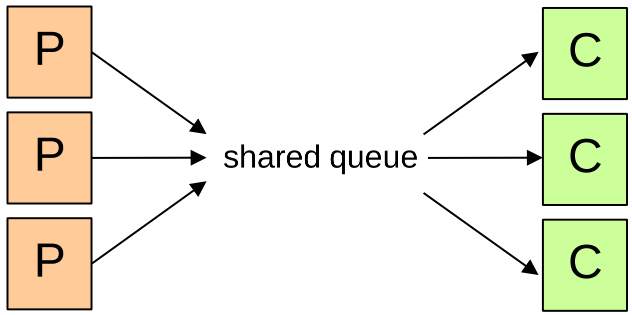
- **General Themes:**
  - Parallelization problems arise from:
    - Communication between workers
    - Access to shared resources (e.g., data)
  - Thus, we need a synchronization support!
  - This is tricky:
    - Finding bugs is hard
    - Solving bugs is even harder

Introduction

- **Patterns for Parallelism:**
  - Master/Workers
  - Producer/Consumer Flow (pipeline)
  - Work Queues







Introduction: Evolution

- Functional Programming
- MapReduce
- Google File System (GFS)

Introduction

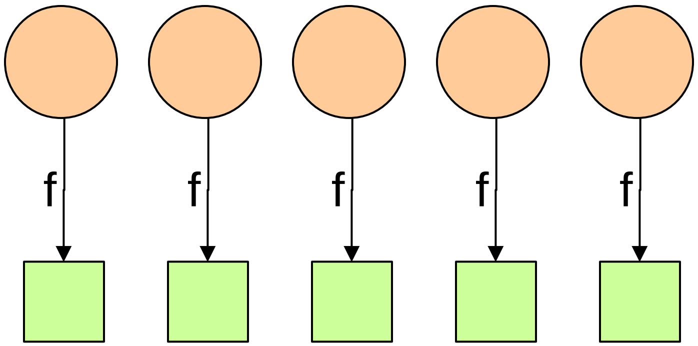
- **Functional Programming:**
  - MapReduce = functional programming meets distributed processing on steroids
    - Not a new idea… dates back to the 50's (or even 30's)
  - What is functional programming?
    - Computation as application of functions
    - Theoretical foundation provided by lambda calculus
  - How is it different?
    - Traditional notions of "data" and "instructions" – are not applicable
    - Data flows are implicit in program
    - Different orders of execution are possible
  - Exemplified by LISP and ML (MetaLanguage – general purpose Functional Programming Language)

Introduction: Lisp → MapReduce?

- What does this have to do with MapReduce?
- After all, Lisp is about processing *lists*
- **Two important concepts in functional programming:**
  - Map: do something to everything in a list
  - Fold: combine results of a list in some way

Introduction: Map

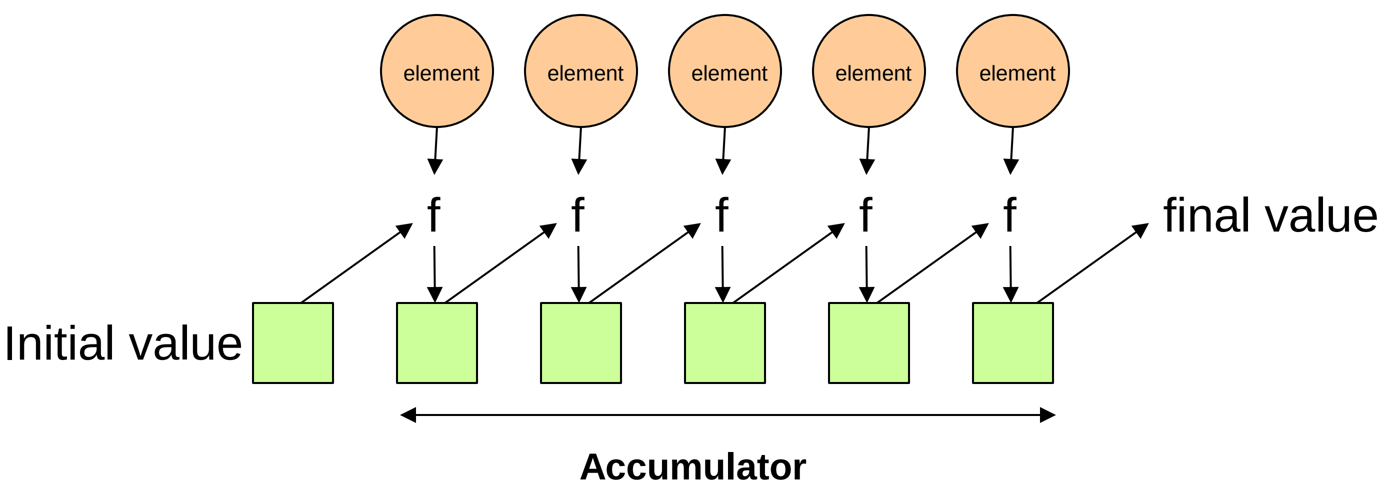
- Map is a higher-order function* (function that takes one or more functions as arguments)
- **How map works:**
  - Function is applied to every element in a list
  - Result is a new list



* https://en.wikipedia.org/wiki/Higher-order_function

Introduction: Fold

- Fold is also a higher-order function
- **How fold works:**
  - Accumulator set to initial value
  - Function applied to list element and the accumulator
  - Result stored in the accumulator
  - Repeated for every item in the list
  - Result is the final value in the accumulator



Lisp → MapReduce

- **Let's assume a long list of records:** imagine if...
  - We can distribute the execution of map operations to multiple nodes
  - We have a mechanism for bringing map results back together in the fold operation
- That's MapReduce! (and Hadoop)
- **Implicit parallelism:**
  - We can parallelize execution of map operations since they are isolated
  - We can reorder folding if the fold function is commutative (a+b = b+a) and associative ((2*3) *4) = (2*(3*4))

Typical Problem

- Iterate over a large number of records
- **Map:** extract something of interest from each
- Shuffle and sort intermediate results
- **Reduce:** aggregate intermediate results
- Generate final output

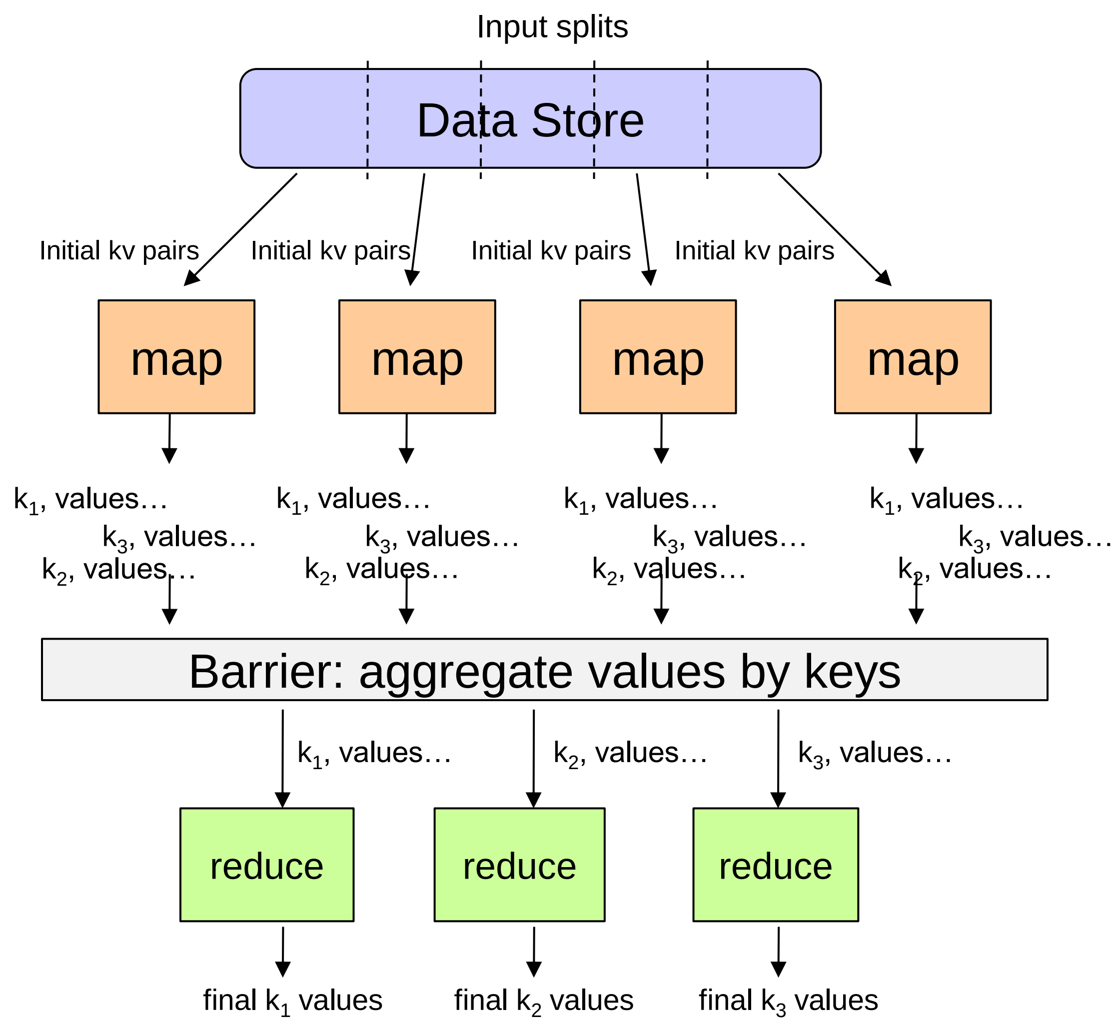
Key idea: provide an abstraction at the point of these two operations

MapReduce

- **Programmers specify two functions:**
  map (k, v) → <k’, v’>*
  reduce (k’, v’) → <k’, v’>*
  - All v’ with the same k’ are reduced together

- **Usually, programmers also specify:**
  partition (k’, number of partitions) → partition for k’
  - Often a simple hash of the key, e.g. hash(k’) mod N
  - Allows reduce operations for different keys in parallel

It's just divide and conquer!



Recall these problems?

- How do we assign work units to workers?
- What if we have more work units than workers?
- What if workers need to share partial results?
- How do we aggregate partial results?
- How do we know all the workers have finished?
- What if some workers die?

MapReduce Runtime

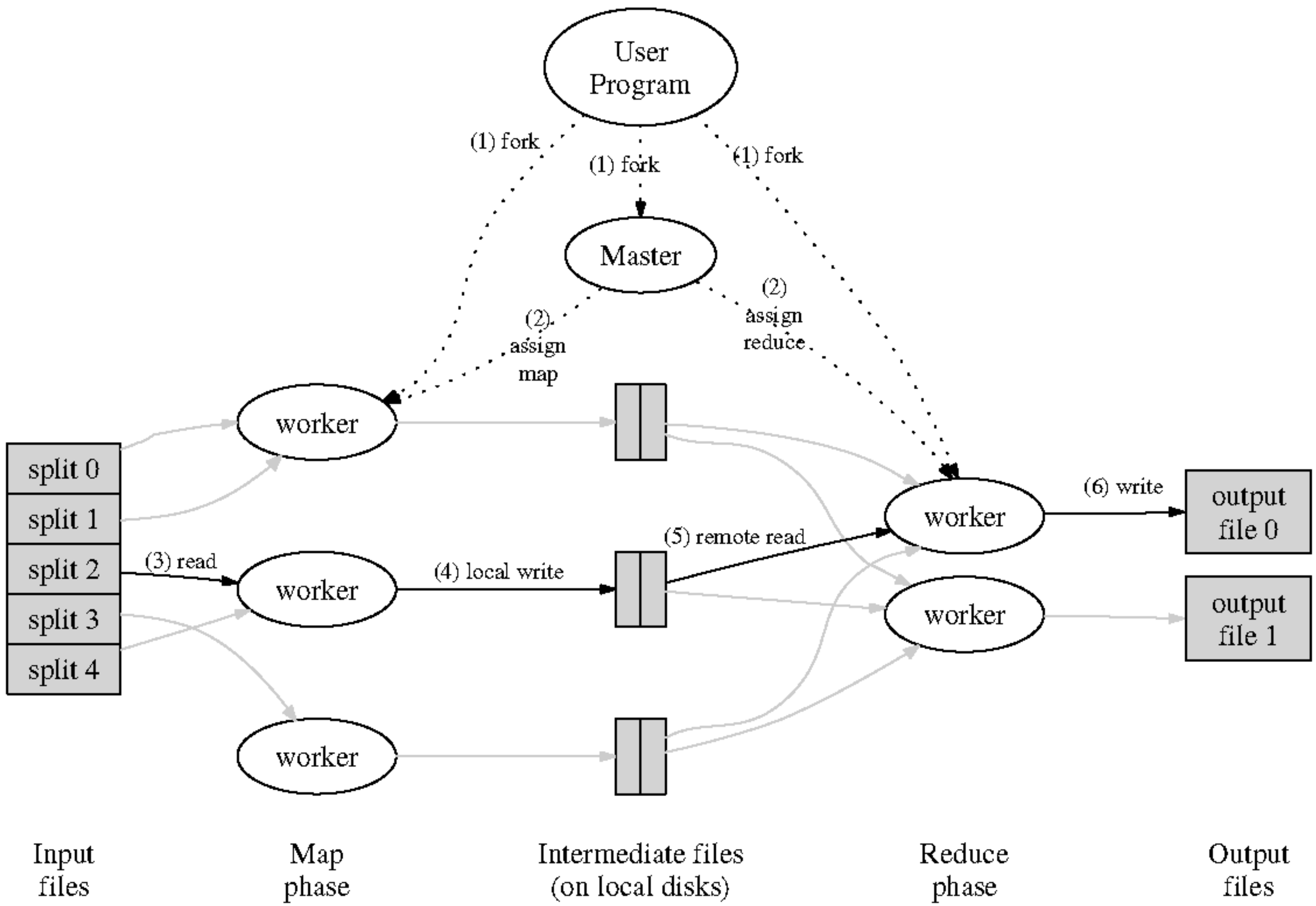
- **Handles data distribution:**
  - Gets initial data to map workers
  - Shuffles intermediate key-value pairs to reduce workers
  - Optimizes for locality whenever possible
- **Handles scheduling:**
  - Assigns workers to map and reduce tasks
- **Handles faults:**
  - Detects worker failures and restarts
- Everything happens on top of GFS (later)

“Hello World”: Word Count

```
Map(String  input_key, String  input_value):
    // input_key: document name
    // input_value: document contents
    for each word w in input_values:
        EmitIntermediate(w, "1");

Reduce(String key, Iterator  intermediate_values):
    // key: a word, same for input and output
    // intermediate_values: a list of counts
    int result = 0;
    for each v in intermediate_values:
        result += ParseInt(v);
        Emit(AsString(result));
```

Behind the scenes…



Bandwidth Optimizations

- **Take advantage of locality**
  - Move the process to where the data is!
- **Use "Combiner" functions – output of the mapper**
  - Executed on same machine as mapper
  - Results in a "mini-reduce" right after the map phase
  - Reduces key-value pairs to save bandwidth

When can you use combiners?

* http://www.tutorialspoint.com/map_reduce/map_reduce_combiners.htm

Skew Problem

- **Issue:** reduce is only as fast as the slowest map
- **Solution:** redundantly execute map operations, use results of first to finish
  - Addresses hardware problems...
  - But not issues related to inherent distribution of data

- **Data, Data, More Data:**
  - All of this depends on a storage system for managing all the data…
  - That’s where GFS (Google File System), and by extension HDFS in Hadoop

Assumptions

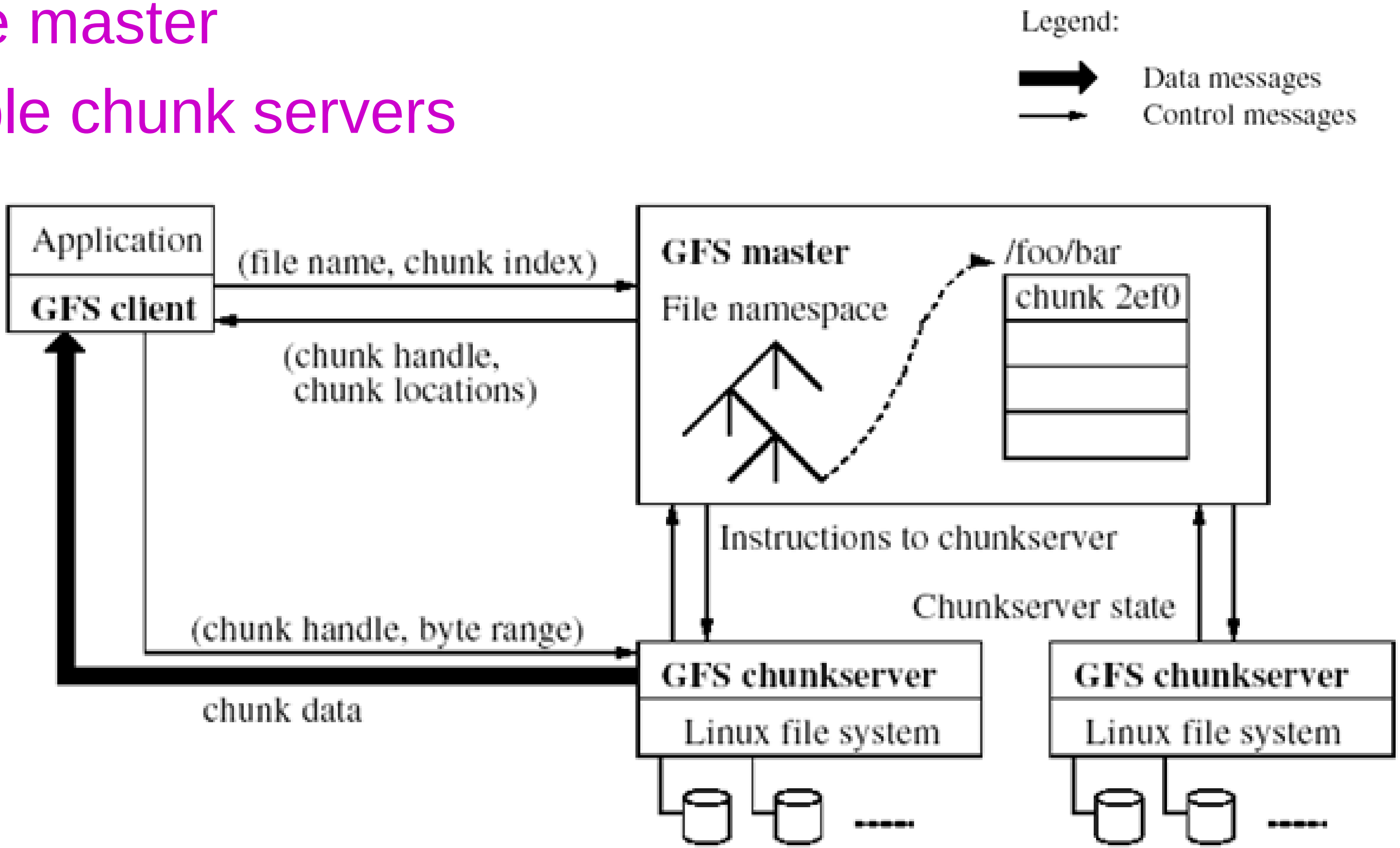
- **High component failure rates**
  - Inexpensive commodity components fail all the time
- **“Modest” number of HUGE files**
  - Just a few millions (!!!)
  - Each is 100MB or larger; multi-GB files is typical
- **Files are write-once, mostly appended to**
  - Perhaps concurrently
- Large streaming reads
- High sustained throughput favoured over low latency

GFS Design Decisions

- **Files stored as chunks**
  - Fixed size (64MB)
- **Reliability through replication**
  - Each chunk replicated across 3+ chunk servers
- **Single master to coordinate access, keep metadata**
  - Simple centralized management
- **No data caching**
  - Little benefit due to large data sets, streaming reads
- **Familiar interface, but customize the API**
  - Simplify the problem; focus on Google apps
  - Add snapshot and record append operations

GFS Architecture

- Single master
- Multiple chunk servers



Can anyone see a potential weakness in this design?

Single master

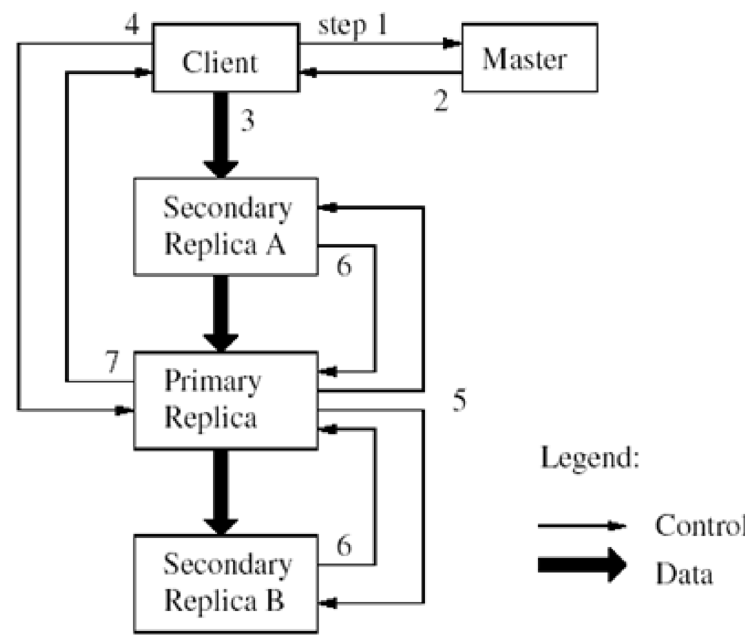
- **From distributed systems we know this is a**
  - Single point of failure
  - Scalability bottleneck
- **GFS solutions:**
  - Shadow masters
  - Minimize master involvement
    - Never move data through it, use only for metadata (and cache metadata at clients)
    - Large chunk size (minimize seeks)
    - Master delegates authority to primary replicas in data mutations (chunk leases)
- Simple, and good enough!

Metadata

- **Global metadata is stored on the master**
  - File and chunk namespaces
  - Mapping from files to chunks
  - Locations of each chunk’s replicas
- **All in memory (64 bytes for Metadata / chunk)**
  - Fast
  - Easily accessible
- **Master has an operation (Edit) log for persistent logging of critical metadata updates**
  - Persistent on local disk
  - Replicated
  - Checkpoints for faster recovery

Mutations

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



Relaxed Consistency Model

- "**Consistent**" = all replicas have the same value
- "**Defined**" = replica reflects the mutation, consistent
- **Some properties:**
  - Concurrent writes leave region consistent, but possibly undefined
  - Failed writes leave the region inconsistent
- **Some work has moved into the applications:**
  - E.g., self-validating, self-identifying records
  - Google apps can live with it
  - What about other apps?

Master’s Responsibilities (1/2)

- Metadata storage
- Namespace management/locking
- **Periodic communication with chunk servers**
  - Give instructions, collect state, track cluster health
- **Chunk creation, re-replication, rebalancing**
  - Balance space utilization and access speed
  - Spread replicas across racks to reduce correlated failures
  - Re-replicate data if redundancy falls below threshold
  - Rebalance data to smooth out storage and request load

Master's Responsibilities (2/2)

- **Garbage Collection**
  - Simpler, more reliable than traditional file delete
  - Master logs the deletion, renames the file to a hidden name
  - Lazily garbage collects hidden files
- **Stale replica deletion**
  - Detect "stale" replicas using chunk version numbers

Fault Tolerance

- **High availability**
  - **Fast recovery:** master and chunk servers re-startable in a few seconds
  - **Chunk replication:** default 3 replicas
  - **Shadow masters**

- **Data integrity**
  - Checksum every 64KB block in each chunk

Parallelization Problems

- How do we assign work units to workers?
- What if we have more work units than workers?
- What if workers need to share partial results?
- How do we aggregate partial results?
- How do we know all the workers have finished?
- What if some workers die?

Managing Dependencies

- **Remember:** Mappers run in isolation
  - You have no idea in what order the mappers run
  - You have no idea on what node the mappers run
  - You have no idea when each mapper finishes
- **Question:** what if your computation is a non-commutative operation on mapper results?
- **Answer:** Cleverly “hide” dependencies in the reduce stage
  - The reducer can hold state across multiple map operations
  - Careful choice of partition function
  - Careful choice of sorting function
- **Example:** computing conditional probabilities

Other things to beware of…

- Object creation overhead
- Reading in external resources is tricky
  - Possibility of creating hotspots in underlying file system

M/R Application:
Cost Measures for Algorithms

1. ***Communication cost*** = total I/O of all processes.
2. ***Elapsed communication cost*** = max # of I/O along any path.
3. (***Elapsed***) ***computation costs*** analogous, but count only running time of processes.

M/R Application: Example: Cost Measures

- **For a map-reduce algorithm:**
  - **Communication cost** = input file size + 2 $\times$ (sum of the sizes of all files passed from Map processes to Reduce processes) + the sum of the output sizes of the Reduce processes
  - **Elapsed communication cost** is the sum of the largest input + output for any map process, plus the same for any reduce process

M/R Application: What Cost Measures Mean

- Either the I/O (communication) or processing (computation) cost dominates:
  - Ignore one or the other.
- Total costs tell what you pay in rent from your friendly neighborhood cloud.
- Elapsed costs are **wall-clock time** using parallelism

Join By Map-Reduce

- **Our first example** of an algorithm in this framework is a map-reduce example
- Compute the natural join R(A,B) $\bowtie$ S(B,C)
- R and S each are stored in files
- Tuples are pairs (a,b) or (b,c)

- Use a hash function ***h*** on B-values to [1..*k*] *buckets*.
- A Map process turns input tuple R(a,b) into key-value pair (b,(a,R)) and each input tuple S(b,c) into (b,(c,S))

Map-Reduce Join – (2)

- Map processes send each key-value pair with key *b* to Reduce process h(b) = b mod k
  - Hadoop does this automatically; just tell it what *k* is
- Each Reduce process matches all the pairs (b,(a,R)) with all (b,(c,S)) and outputs (a,b,c)

Cost of Map-Reduce Join

- Total communication cost = $O(|R|+|S|+|R \bowtie S|)$
- Elapsed communication cost = O(*s*)
  - We're going to pick *k* and the number of Map processes so I/O limit *s* is respected
- With proper indexes, computation cost is linear in the input + output size
  - So computation costs are like comm costs

END