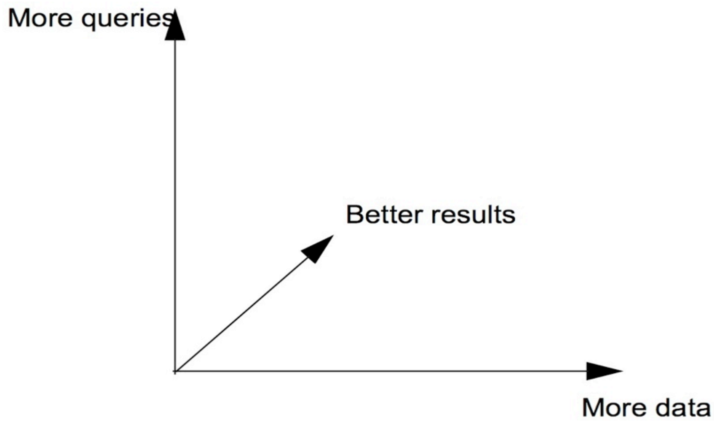
Platforms as a Service

Cloud Services

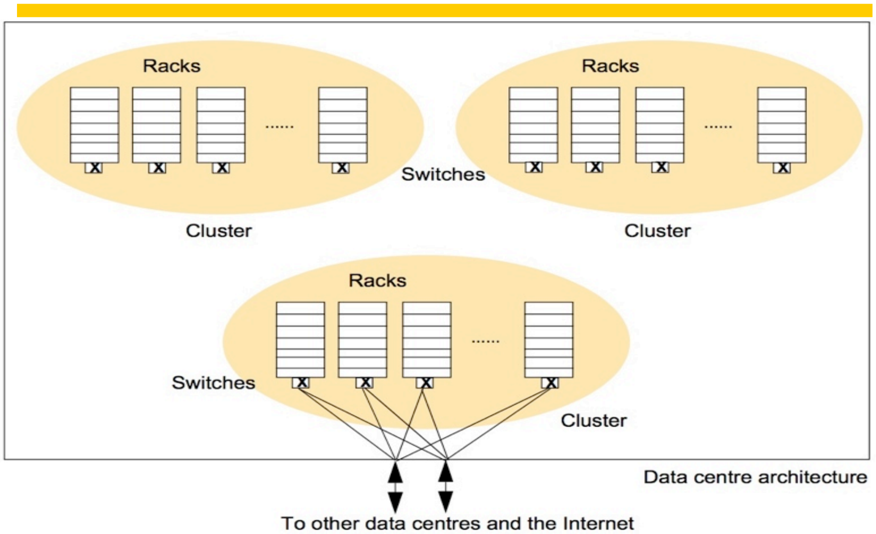
- Infrastructure as a Service (IaaS)
  - Provide machines in form of virtual machines
  - Guarantees in terms of main memory, CPU…
- Platform as a Service (Paas)
  - Provide abstractions of
    - General purpose DBS / key/value-store to store data
      - (user generates the database and writes application programs to access data)
    - Application Server
      - (user writes application programs and runs them in application server environment)
    - …
  - The Hadoop Stack is a Paas
- Software as a Service (SaaS)
  - Provide the application (Google Docs)
  - (often includes the storage)

What does it mean to scale

- SLA:
  - response time not above certain limit

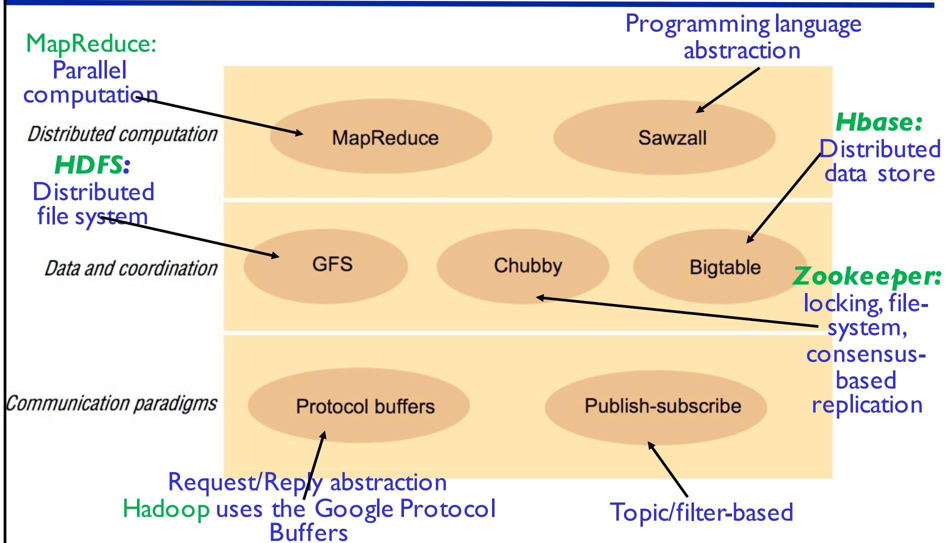


Physical Infrastructure



(To avoid clutter the Ethernet connections are shown from only one of the clusters to the external links)

Google (Yahoo) Middleware used by all services



- Logging and monitoring EVERYWHERE

Publish-Subscribe

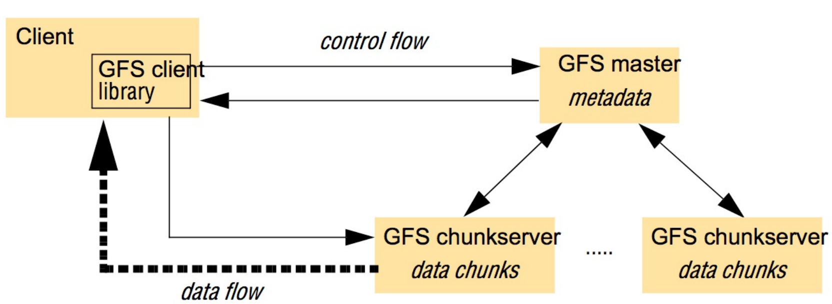
- Topic based with filters
  - Subscription to a topic
  - Filters allow to receive only a subset of notifications published on a topic
- Scalable high-throughput topics
  - Multicast tree of brokers per topic
    - Root it the publisher, leaves are the subscribers
    - Early filtering
- Reliability
  - Replication: two independent trees per topic
- Timely delivery
  - Control flow (rate scheme, overload protection…)

GFS/HDFS

- Support of application
  - Super-large files (Gigabytes, Terabytes)
  - Tens of millions of files on hundreds of nodes
  - Mostly read-only (scan/streaming) and append
  - Throughput over latency
- Interface
  - Create / delete / open / close / read / write / append
- File split into GFS Chunks / HDFS Blocks
  - default: 64 MB

GFS/HDFS coordination

- File split into GFS Chunks / HDFS Blocks
- One GFS master / HDFS NameNode
  - Maintains meta-information: location of chunks/blocks
  - No data-flow through master / NameNode
  - Clients cache meta-information
- Many GFS chunkservers / HDFS DataNodes
  - Keep replicas of the chunks/blocks

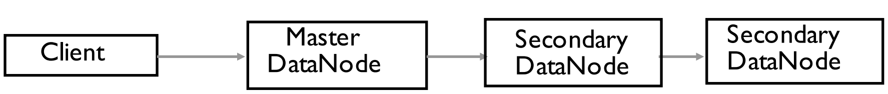


GFS Master/NameNode concept

- Master / NameNode concept works because:
  - Only control flow / no data flow
    - Data flow only between clients and chunk servers
  - Large chunk size → little meta data
  - Caching of meta-information at clients
- NameNode persists everything!
  - For availability:
    - Meta-information files can be replicated
    - (BookKeeper; but not yet integrated)
- NameNode remains single point of failure
- Other tasks
  - Re-replication in case chunk replica fails
  - Load-balancing

HDFS Replication

- Blocks are write-once / immutable
- Typically each block is replicated three times
  - Two times in the same rack
  - One in another rack
- Each block has a master DataNode and two secondary DataNodes
- Write: Chain/Pipeline replication
  - From client to master DataNode to secondaries
  - In a stream (parallel writing)
  - Ack by last secondary
- Read: from closest replica



Hbase (BigTable)

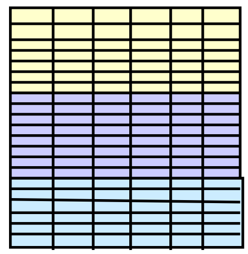
- Data Model
  - Hybrid between table (relation), key/value, and semi-structured
  - A table can have many columns (attributes)
  - Columns are bundled into *column families*
  - One mandatory column is the key
  - New columns can be added dynamically
  - Each row must have a unique value in the key column
  - Each row can have values in the other columns
  - Hbase maintains versions on a column basis
    - A row can have more than one value in a given column
    - Each value is considered a version (timestamped)

Hbase Query Language

- Get one row, or columns from one row
  - Indicate row-key, columns, timestamp
- Scan
  - Read all data of the table
- Scan with filter
  - Put a condition
    - On primary key (e.g., a range)
    - On other attributes (salary > 5000)
- No joins, ops on more than one table, aggregation….

Hbase Horizontal Data Partitioning

- **Horizontal** Partitioning
  - A table can be partitioned into many *regions*
  - Regions are maintained by region servers
  - Each region has a set of rows
  - **Range Partitioning** by primary key attribute
    - Each partition holds rows with primary keys within a range
    - E.g., partition with rows with rowid [1;10000], [10001,20000], …
  - Complete and disjoint
    - Each tuple is in exactly one partition



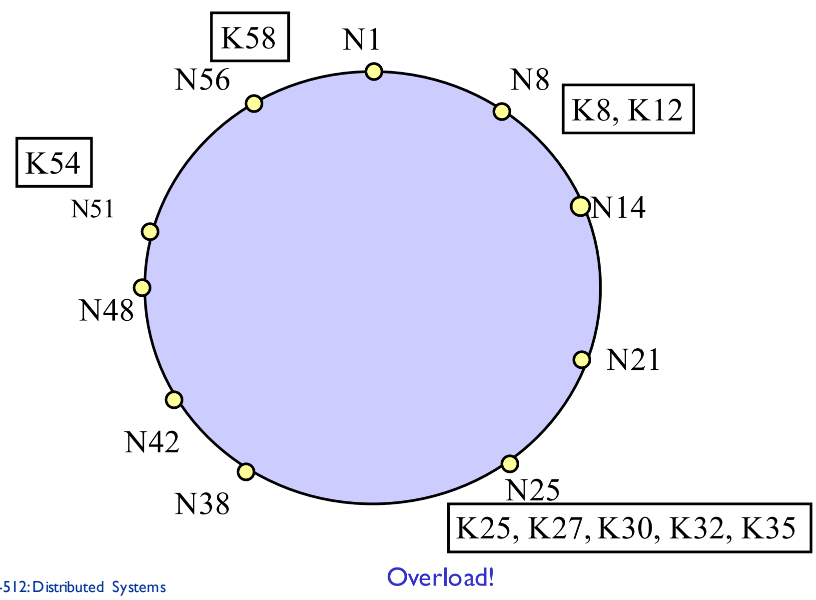
Querying partitioned data

- Queries:
  - Get one tuple (point query):
    - access only region that has that key
    - Localized access
  - Scan (range query):
    - scan all relevant regions
    - Possibly in *parallel*
    - Intra-query parallelism

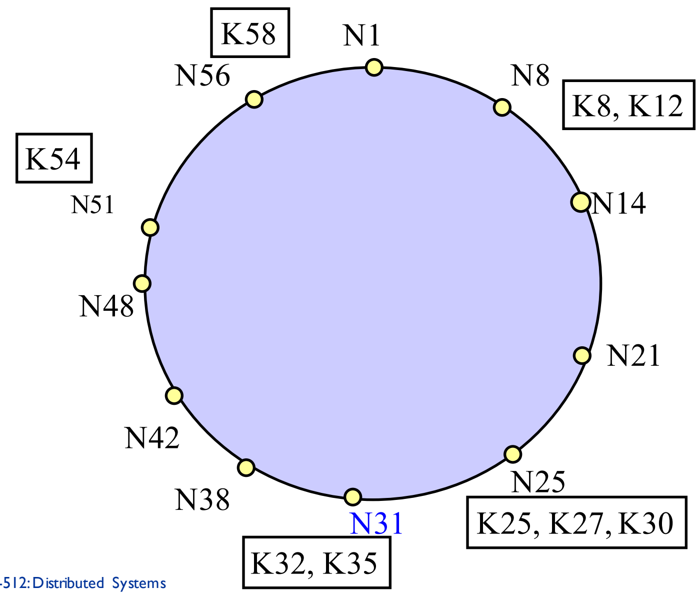
Non Hbase excursion: Hash Partitioning

- By Mod operator (bad idea)
  - Nodes have identifiers 0, …. N-1
  - Let key of a row have value K, then row is assigned to node
    - K mod N
  - Problem: If a new node is added, all keys need to be reshuffled
- Consistent Hashing: Cassandra, Riak, ….
  - Keys are hashed using a hash function that maps to a space between 0 and x (x being very very large)
  - Node identifiers are hashed to the same space (i.e., each node gets a random identifier between 0 and x)
  - Let n1 and n2 be two nodes with hashed id(n1) < id(n2) and there is no node n3 such that id(n1) < id(n3) < id(n2); i.e., n1 and n2 are "neighbors"
  - Let r be a row with hashed key id(r) such that id(n1) <= id(r) < id(n2): then r is stored on n1

Consistent Hashing



Consistent Hashing

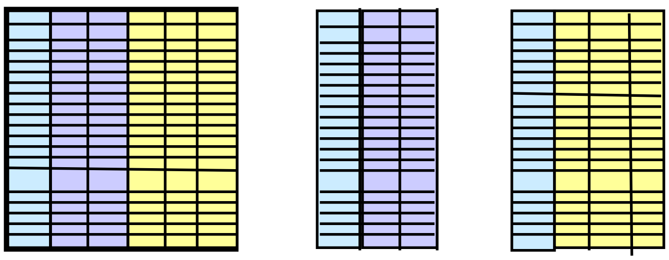


Virtual Node Consistent Hashing

- Standard Consistent Hashing
  - One node can get overloaded if range is a hot spot
  - Adding nodes:
    - Load of one overloaded node is distributed across two nodes
- Fine-grained load balancing
  - Every node has many virtual nodes, each with its own identifier
  - Virtual nodes serve much smaller range; hopefully sum of many virtual nodes does not represent a hot spot
  - Adding a new node:
    - Get a few virtual nodes from each existing node

Hbase Vertical Partitioning

- Different column families can be stored in different files
- In order to identify records, primary key might be replicated in all files
- Query: retrieve only columns of one family: only relevant files are accessed
  - SELECT column1, column2 from table where primary key = x



Hbase Architecture

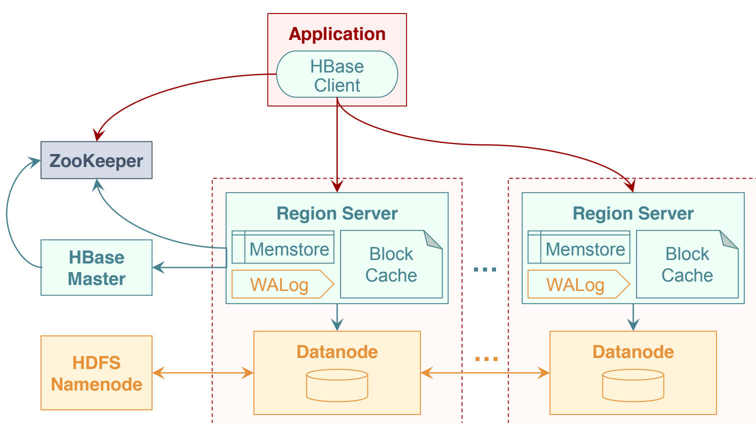


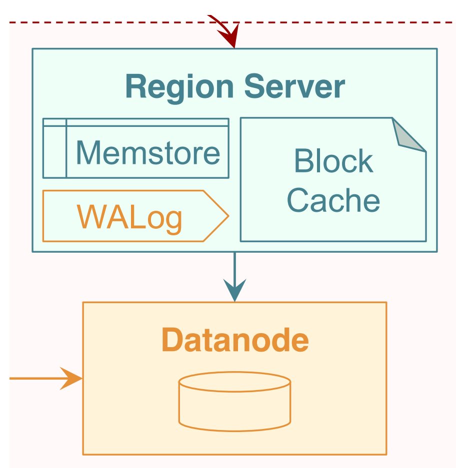
Figure 3.1: HBase Architecture

Hbase Architecture

- Regions are stored on Region Servers
  - Data Distribution
  - No Data Replication at Hbase layer
- Hbase Master
  - Meta-data:
    - location of each region
    - Reliably stored in Zookeeper (fault-tolerant!)
  - Load-balancing: initiates move of regions
  - Failure Handling:
    - detects failed region servers
    - Restarts regions on other region servers
- Hbase Client
  - Retrieves loation information from Zookeeper
  - Interacts with all relevant region servers

Region Server

- Updates
  - Each update creates a new, timestamped version
  - New versions appended to *Memstore*
  - After threshold of new versions, a new file is written to HDFS
    - (file replicated at HDFS for fault-tolerance);
    - one copy at local data node for fast access
- Block Cache
  - Holds most recent file blocks
- WALog: fast logging of ops



Finding and managing records

- Index for primary key
- Index in each HDFS block
- Version management:
  - Compaction: take several HDFS files for same region and merge
    - Collocate versions of same record
    - Delete old versions

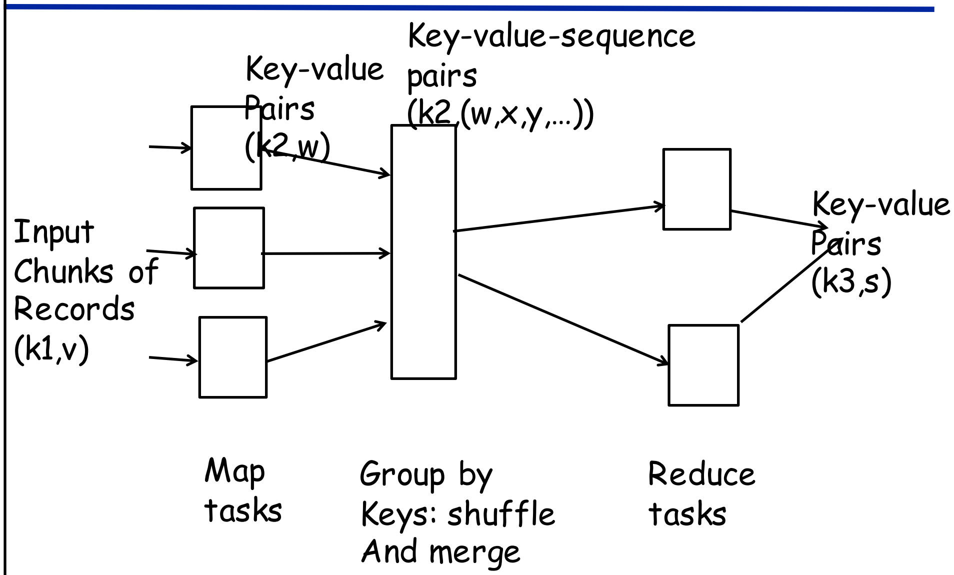
Map Reduce: Data Processing at Massive Scale

- Massive Scale
  - Petabytes of data
  - 100s, 1000s, 10000s of servers
  - Many hours
- Failure becomes an issue
  - If medium-time-between failure is 1 year
  - Then 10000 servers have one failure / hour
  - Query execution must succeed even if individual nodes fail

The basics

- High-level programming model AND implementation for large-scale parallel data processing
- Programming model
  - Read a lot of data records (key-value pairs)
  - ***Map tasks:*** extract something interesting from records and output a new set of data records (key-value pairs)
  - Shuffle and sort (same key to same reduce task)
  - ***Reduce tasks:*** aggregate, summarize, filter
  - Write the results
- Closed model
  - Input and output of map-reduce are key/value pairs

From left to right



Overview

- Input and output considered key/value pairs in order to be able to compose several map/reduce instances
- Map and Reduce functions are written by programmer
- Number of map tasks and reduce tasks given at start of program
- The rest done automatically (at least conceptually)

Example: Word Count

- Given: Document Set DS(K, documenttext)
- Output: For each word *w* occurring at least in one document of DS: indicate the number of occurrences of *w* in DS

Map Step

- Input Parameters from User
  - Number m of map tasks
  - Number r of reduce tasks
  - Data set = document set DS
- Map function written by User

  WordCountMap:

  For each input key/value pair (dkey, dtext)

  For each word w of dtext

  Output key-value pair (w, 1)
- System splits input set into m partitions
- System creates m map tasks gives each one partition
- Each map task executes map function on its partition
- Map step only completes once all map tasks are done

Shuffle and Reduce Step

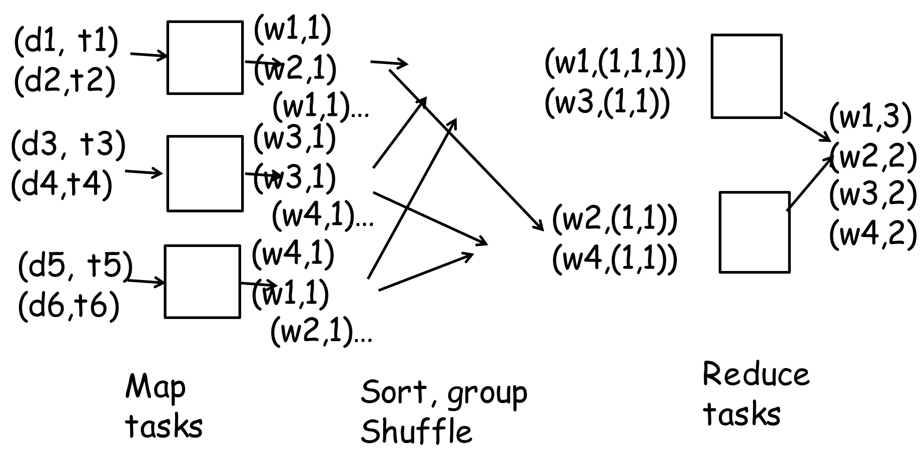
- System sorts map outputs by key and transforms all key/value pairs $(k, v_1), (k, v_2), \ldots (k, v_n)$ with same key k to one key/value-list pair $(k, (v_1, v_2, \ldots v_n))$
  - For Word count: all ('and', 1), ('and', 1), ('and', 1) … are transformed into one ('and', (1,1,1,….))
- System partitions output by key into *r* partitions
- Systems creates *r* reduce tasks and assignes each one partition
- Each reduce task executes user written reduce function

  WordCountReduce:

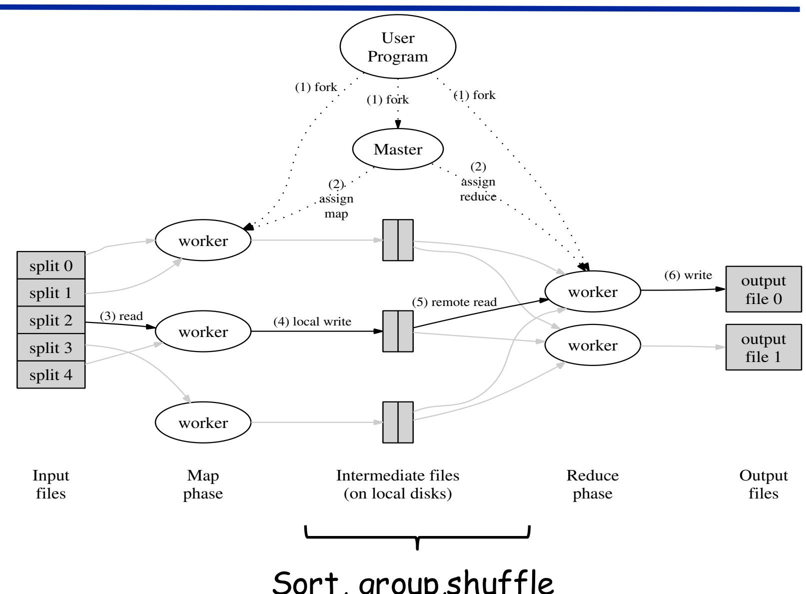
  For each input key/value-list pair $(k, (v_1, v_2, \ldots v_n)$

  Output (k, n)

Example Execution



Once more



Phase Summary

- Split input into partitions
- At map tasks
  - Record reader
  - Map function
  - Write to local file
- Group and shuffle
  - Group keys and aggregate value-lists
  - Copy from map location to reduce location
  - group keys and aggregate value-lists
- At reduce tasks
  - Reduce function and write to file system
- Several "waves" possible

Friends Calculation?

Red: key
Green: value

Friends information

A -> B C B -> A C D

C -> A B D D -> B C

```
Map(person, (list of friends))
  for each friend in list of friends
     if "person" alphabetically before "friend"
        output ( (person, friend) , (list of friends))
     else
        output ( (friend, person) , (list of friends))
Reduce( (p1, p2) , ((list of friends 1), (list of friends 2)) )
     output ((p1, p2) , (intersection of two friend lists) )
```

Implementation

- There is one master node controlling execution
- Master partitions file into m partitions
- Master assigns workers (server processes) to m map tasks
- Workers executing map tasks write to local disk
- Master assigns workers to r reduce tasks
- Reduce workers implement group and shuffle (read from map disks) and execute reduce tasks
  - Pull approach

Failures

- Failures are detected by master
  - Failure of map task during map phase
    - master assigns new worker to map task
  - Failure of map task during reduce phase
    - Master assigns new worker to map task to redo (as data stored locally)
  - Failure of reduce task during reduce phase
    - Master assigns new worker to reduce task
- Straggler
  - A machine that takes unusually long to complete one of its last tasks
    - Maybe some I/O problem, too many other tasks…
  - Solution: back execution of last few remaining in-progress tasks

Discussion

- Simple programming model
  - Easy to understand
  - Sometimes complicated to implement a task
  - More and more support for data-declarative languages
    - E.g., SQL
- Built-in fault-tolerance
- But Lot of persistence
  - After map and after reduce
  - Time consuming
- Rigid
  - Everything must be map/shuffle/reduce
  - Simple things often require map/reduce workflows

Relational Operators

- R(a, b, c) (a primary key)

SELECT a, b

FROM R

WHERE c < 50

Map (a, (b,c))

output (a, b) if c < 50

Reduce?

SELECT b, average(c)

FROM R

GROUP BY b

Relational Languages on top of Map/Recue

- Hive, Pig Latin

Complex Queries

- Require a sequence of concatenated map/reduce jobs
- Fault-tolerance achieved by persisting each output of each map and each reduce task
- Expensive

Spark

- Resilient Distributed Datasets (RDD)
  - In main-memory
  - Distributed among machines
- Many transformations/actions defined on RDD
  - Map, reduce, group, join, union, ….
- Reslience:
  - Log *lineage* (sequence of transformations/actions) performed on a partition
  - Easy reconstruct in case of failure.
- Originally a functional programming API
- Now all kinds of things (incuding SQL, machine learning)