AIM3 – Scalable Data Mining and Data Analysis

02 – Distributed filesystems and MapReduce

Sebastian Schelter, Christoph Boden, Volker Markl

Fachgebiet Datenbanksysteme und Informationsmanagement
Technische Universität Berlin

http://www.dima.tu-berlin.de/

Recap

A new set of tools

- Distributed filesystems
  - store petabytes of data in the cluster
  - transparently handle reads, writes and replication

- Parallel processing platforms
  - offer a parallel programming model to allow developers to write distributed applications
  - move computation to data, not data to computation
  - relieve the developer from handling concurrency, network communication and machine failures

Our algorithms have to change

- Each machine will only see a small portion of the data
  - we cannot use random access anymore, we must always work on partitioned data
  - joining data become very costly as lots of machines will be involved

- Communication via network and disk becomes the bottleneck
  - our algorithms must try to locally aggregate as much as possible
  - minimizing network traffic becomes the key to scaling out algorithms

- Concurrency and recovery must be hidden from the developer
  - algorithms must fit into a simple, parallelizable programming model
  - the system (not the developer) handles concurrency and recovery

Agenda

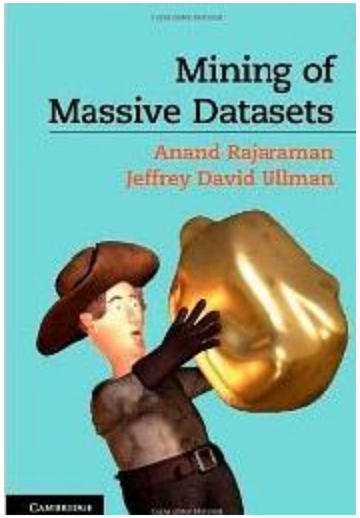
- Topics of the course
  - Motivation, Overview
  - MapReduce & Distributed filesystems
  - MapReduce: Joins, Patterns & Extensions
  - Stratosphere
  - Clustering
  - Dimensionality Reduction
  - Data Stream Mining
  - Graph Processing & Social Network Analysis
  - Graph Processing: Google Pregel
  - Collaborative Filtering: Neighborhood Methods
  - Collaborative Filtering: Latent Factor Models
  - Classification
  - Textmining
  - Specialized Machine Learning approaches

Tasks in the course

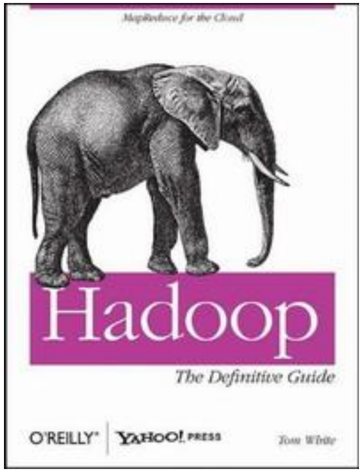
- 3 two week homework assignments
  - available as Java project on github
  - implement your solution and send us a patch
  - present your solution in the course
- six week project (in groups of 2-3 students)
  - implement a data mining algorithm on a parallel processing platform
  - demonstrate your solution on a real world dataset
  - 3 ten minute presentations: problem and planned solution, prototypical implementation, final presentation with results on real world data
- oral exam

- Mining of Massive Datasets (Rajaraman, Ullman)

  free PDF version available at:
  http://infolab.stanford.edu/~ullman/mmds.html



- Hadoop: The definitive guide (White)



Resources

- ISIS course page
  https://www.isis.tu-berlin.de/course/view.php?id=6535

- mailinglist for the lecture
  aim3@dima.tu-berlin.de

- source code for the homework
  https://github.com/dimalabs/scalable-datamining-class

Distributed filesystems

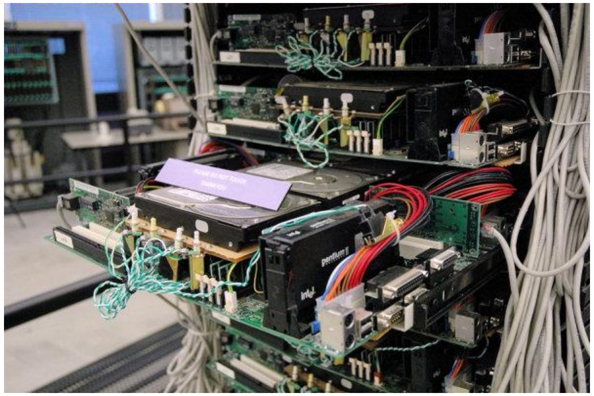
Google 1997: one machine is not enough…

Motivation for having distributed systems

- Economic and technical drivers for distributed systems
  - **Costs**: better price/performance as long as commodity hardware is used for the component computers
  - **Performance**: by using the combined processing and storage capacity of many nodes, performance levels can be reached that are out of the scope of centralized machines
  - **Scalability/Elasticity**: resources such as processing and storage capacity can be increased incrementally
  - **Availability**: by having redundant components, the impact of hardware and software faults on users can be reduced

Google Servers in the early days

- Each server rack holds 40 to 80 commodity-class x86 PC servers with custom Linux
  - each server runs slightly outdated hardware
  - each system has its own 12V battery to counter unstable power supplies
  - no cases used, racks are setup in standard shipping containers and are just wired together



- very unstable, but also very cheap
  → high “bang-for-buck” ratio

Typical first year for a new cluster (consisting of several racks)

- ~**0.5 overheating** (power down most machines in <5 mins, ~1-2 days to recover)
- **~1 PDU (power distribution unit) failure** (~500-1000 machines suddenly disappear, ~6 hours to come back)
- ~**1 rack-move** (plenty of warning, ~500-1000 machines powered down, ~6 hours)
- ~**1 network rewiring** (rolling ~5% of machines down over 2-day span)
- ~**20 rack failures** (40-80 machines instantly disappear, 1-6 hours to get back)
- ~**5 racks go wonky** (40-80 machines see 50% packet loss)
- ~**8 network maintenances** (might cause ~30-minute random connectivity losses)
- ~**12 router reloads** (takes out DNS and external VIPs for a couple minutes)
- ~**3 router failures** (traffic immediately pulled for an hour)
- ~**dozens of minor 30-second DNS blips**
- ~**1000 individual machine failures**
- **thousands of hard drive failures, countless slow disks, bad memory, misconfigured machines, etc**.

Challenges to data center software

- Challenges to the data center software
  - deal with all these hardware failures while avoiding any data loss and ~100% global uptime
  - decrease maintenance costs to minimum
  - allow flexible extension of data centers

- Solution
  - use cloud technologies
  - GFS (Google File System)
  - HDFS (Hadoop Distributed File System), an open source implementation of GFS

Designing a distributed filesystem

- Design constraints and considerations
  - run on potentially unreliable commodity hardware
  - files are large (usually ranging from 100 MB to multiple GBs of size)
  - billions of files need to be stored
  - most write operations are appends
  - random writes or updates are rare
  - most files are write-once, read-many
  - appends are much more resilient than random updates
  - most applications rely on MapReduce which naturally results in file appends

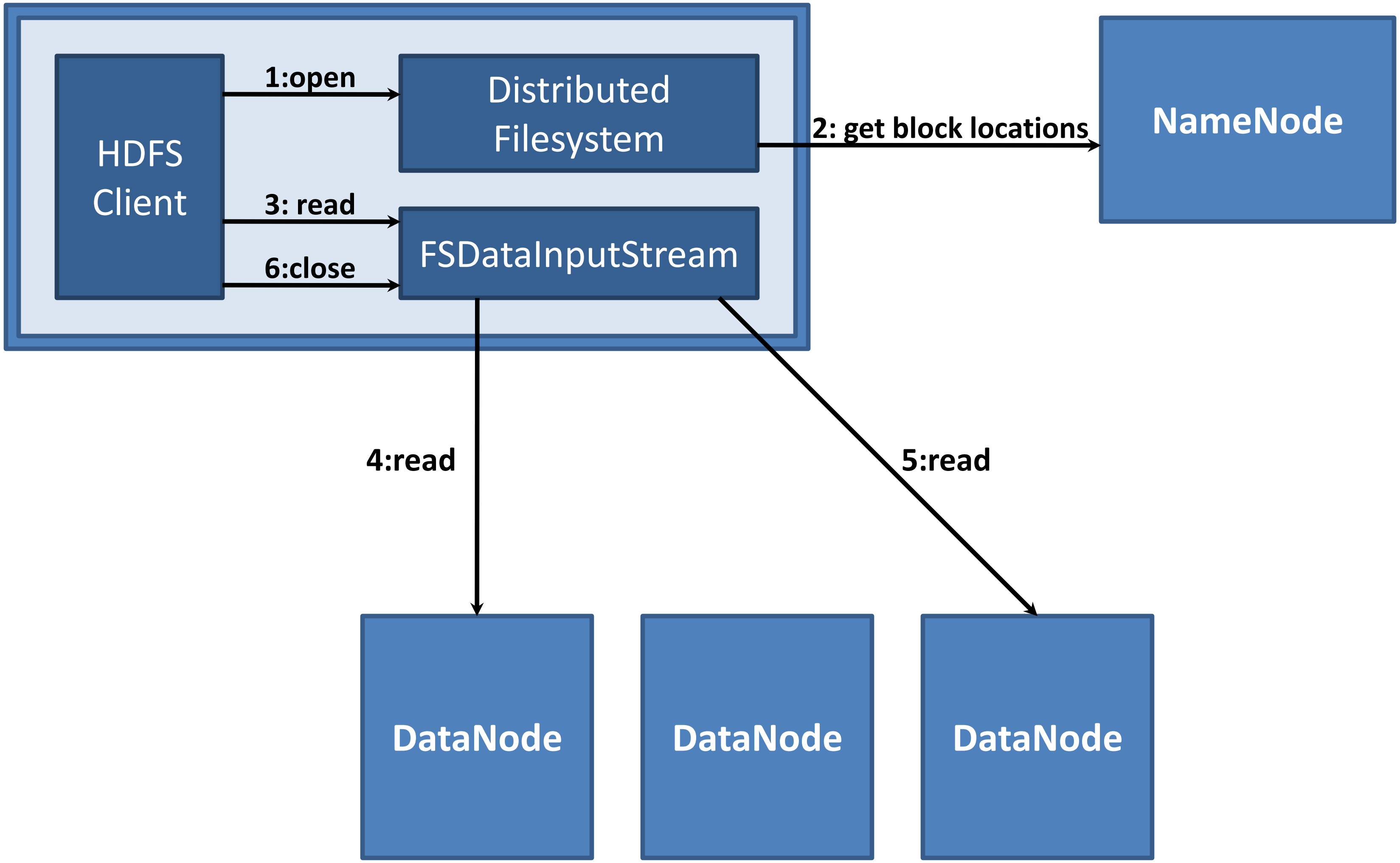
- Most common read operation: sequential streams of large data quantities
  - (e.g. streaming video, transferring a web index chunk, etc)
  - frequent streaming renders caching useless
  - cus of GFS is on high overall bandwidth, not latency

- File system API must be simple and expandable
  - Flat file namespace suffices
  - file path is treated as string (no directory listing possible)
  - qualifying file names consist of namespace and file name
  - no POSIX compatibility needed
  - Additional support for file appends and snapshot operations

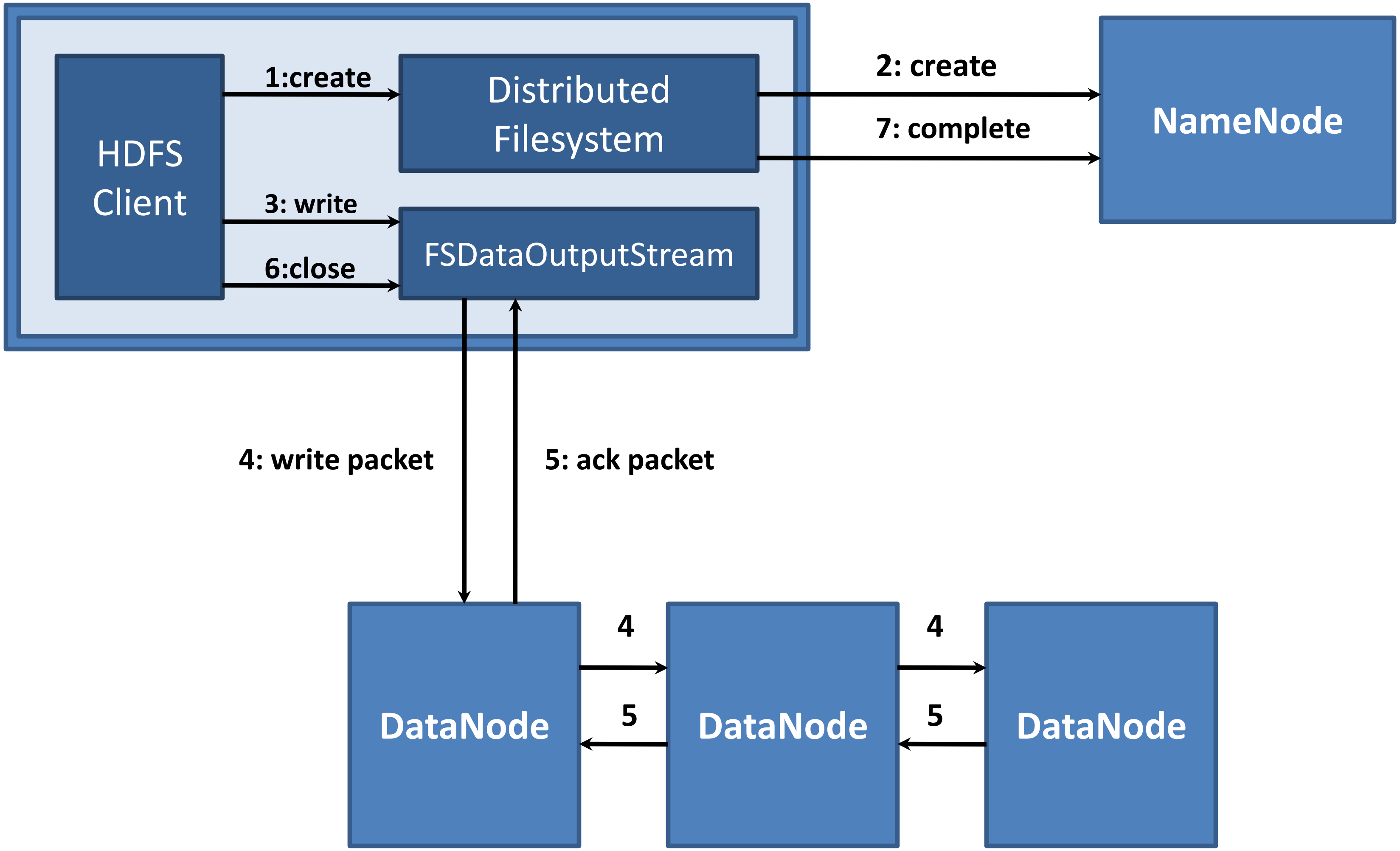
HDFS Architecture

- blocks
  - files are broken into block-sized chunks
  - blocks are stored as independent units
- a single master server (NameNode)
  - manages the filesystem namespace
  - maintains the filesystem tree and metadata for all files
  - knows the data nodes on which all the blocks for a given file are located
- multiple workers (DataNodes)
  - store and retrieve blocks (either initiated by the NameNode or a client)
  - communicate with NameNode about the blocks they store
- replication
  - blocks are redundantly stored on multiple DataNodes

Anatomy of a file read in HDFS



Anatomy of a file write in HDFS



Replica placement & coherency

- default strategy for replica placement
  - 3 copies (replicas) of each block
  - first replica on the local or random node
  - second replica on a node of a different rack (*off-rack*)
  - third replica on a node of the same rack

- Coherency model
  - file is visible after creation
  - no guarantee that written contents are visible
  - data becomes visible once a complete block is written
  - *sync* method (similar to *fsync*) forces all buffers to be flushed to the datanodes, subsequently created readers will see the data

MapReduce

Where traditional Databases are unsuitable

- Analysis over raw (unstructured) data
  - Text processing
  - In general: If relational schema does not suit the problem well
    - XML, RDF
- Where cost-effective scalability is required
  - Use commodity hardware
  - Adaptive cluster size (horizontal scaling)
  - Incrementally growing, add computers without requirement for expensive reorganization that halts the system
- In unreliable infrastructures
  - Must be able to deal with failures – hardware, software, network
    - Failure is expected rather than exceptional
  - Transparent to applications
    - very expensive to build reliability into each application

Example Use Case: Web Index Creation

- A Search Engine scenario:
  - Have crawled the internet and stored the relevant documents
  - Documents contain words (Doc-URL, [list of words])
  - Documents contain links (Doc-URL, [Target-URLs])

- Need to build a search index
  - Invert the files (word, [list of URLs])
  - Compute a ranking (e.g. page rank),
    which requires an inverted graph: (Doc-URL, [URLs-pointing-to-it])

- Obvious reasons against relational databases here
  - Relational schema and algebra do not suit the problem well
  - Importing the documents, converting them to the storage format is expensive

- A mismatch between what Databases were designed for and what is really needed:
  - Databases come originally from transactional processing. They give hard guarantees about absolute consistencies in the case of concurrent updates.
  - Analytics are added on top of that
  - Here: The documents are never updated, they are read only. It is only about analytics here!

A ongoing Re-Design…

- Driven by companies like Google, Facebook, Yahoo

- Use heavily distributed system
  - Google used 450,000 low-cost commodity servers in 2006 in cluster of 1000 – 5000 nodes

- Redesign infrastructure and architectures completely with the key goal to be
  - Highly scalable
  - Tolerant of failures

- Stay generic and schema free in the data model

Retrieving and Analyzing Data

- Data is stored as custom records in files
  - Most generic data model that is possible
- Records are read and written with data model specific (de)serializers
- Analysis or transformation tasks must be written directly as a program
  - Not possible to generate it from a higher level statement
  - Like a query-plan is automatically generated from SQL
- Programs must be parallel, highly scalable, fault tolerant
  - Extremely hard to program
  - Need a programming model and framework that takes care of that
  - The **map/reduce** model has been suggested and successfully adapted on a broad scale

What is Map/Reduce?

- Programming model
  - borrows concepts from functional programming
  - suited for parallel execution – automatic parallelization & distribution of data and computational logic
  - clean abstraction for programmers

- Functional programming influences
  - treats computation as the evaluation of mathematical functions and avoids state and mutable data
  - no changes of states (no side effects)
  - output value of a function depends only on its arguments

- `Map` and `Reduce` are higher-order functions
  - take user-defined functions as argument
  - return a function as result
  - to define a map/reduce job, the user implements the two functions

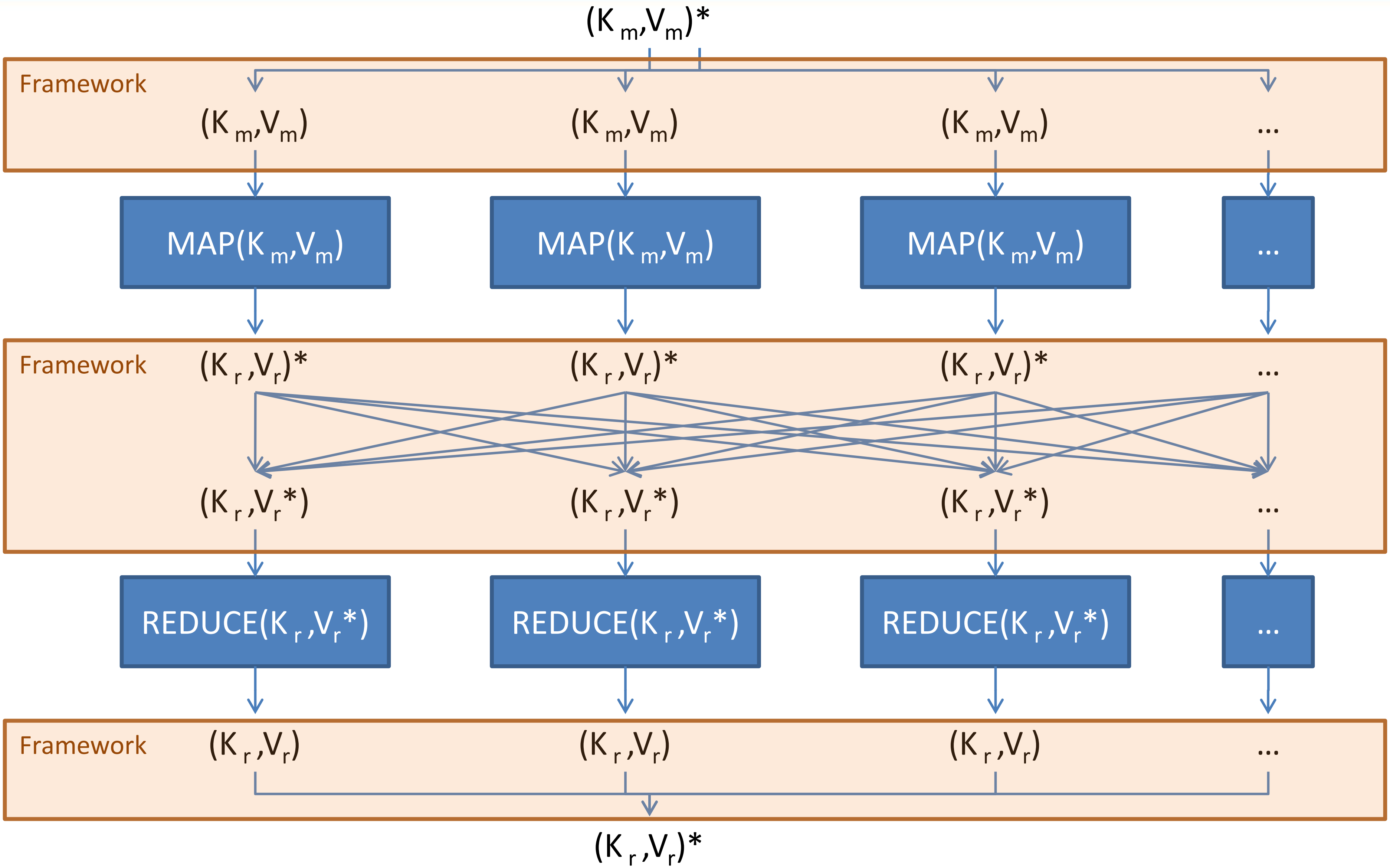
User Defined Functions

- The data model
  - key/value pairs $(K \times V)$
  - e.g. (int, string)

- The user defines two functions
  - map: $\mathcal{M} : (K_m \times V_m) \mapsto (K_r \times V_r)^*$
    - input key-value pairs: $(k, v)\ k \in K_m,\ v \in V_m$
    - output key-value pairs: $(g, w)\ g \in K_r,\ w \in V_r$
  - reduce: $\mathcal{R} : (K_r, V_r^*) \mapsto (K_r, V_r)$
    - input key $\in K_r$ and a list of values $\in V_r^*$
    - output key $\in K_r$ and a single value $\in V_r$

- The framework
  - accepts a list $(K_m \times V_m)^*$
  - outputs result pairs $(K_r, V_r)^*$

Data Flow in Map/Reduce



Map Reduce Illustrated (1)

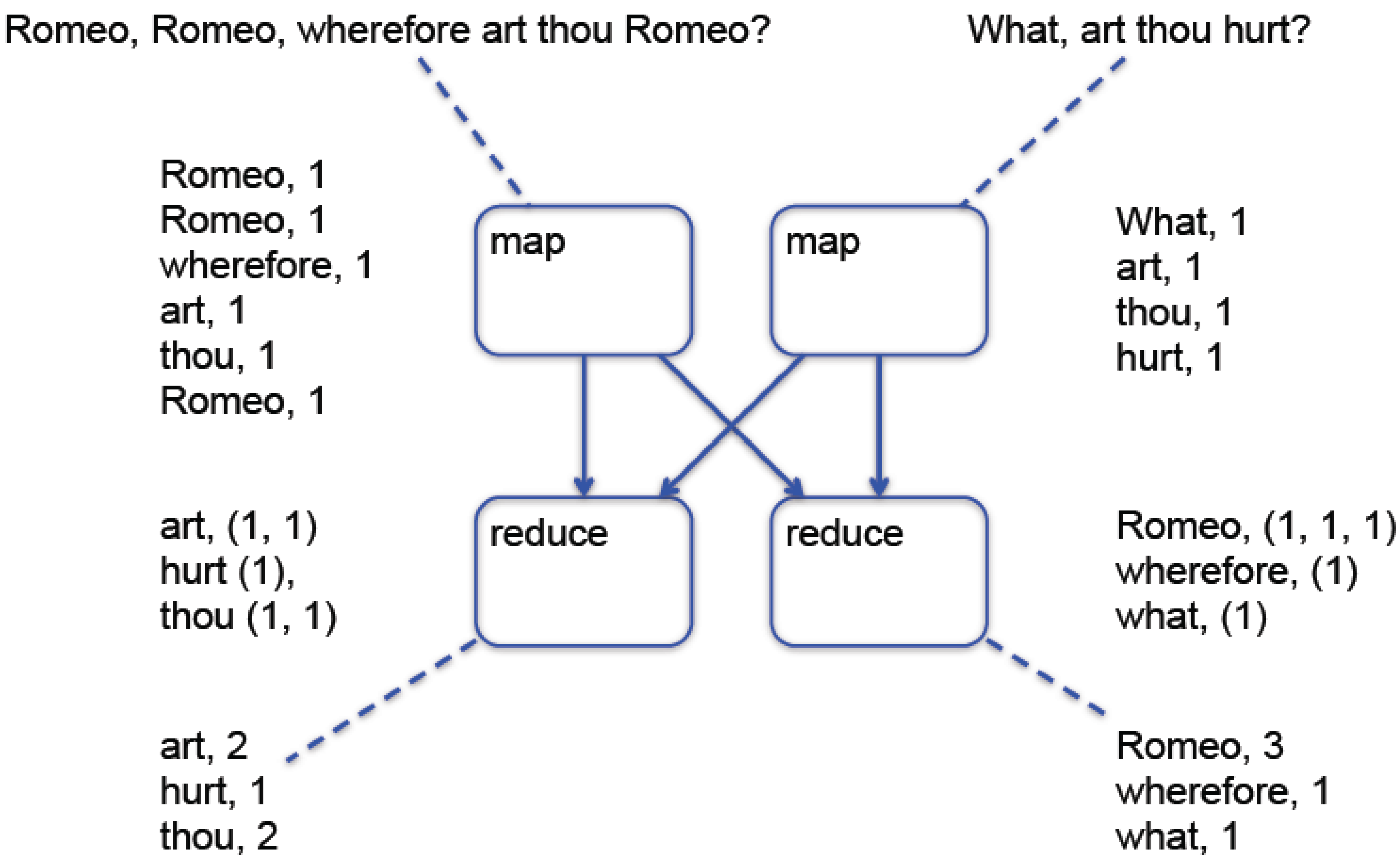
- *Problem*: Counting words in a parallel fashion
  - How many times different words appear in a set of files
  - **juliet.txt**: Romeo, Romeo, wherefore art thou Romeo?
  - **benvolio.txt:** What, art thou hurt?
  - Expected output: Romeo (3), art (2), thou (2), art (2), hurt (1), wherefore (1), what (1)

- *Solution*: Map-Reduce Job

```
map(filename, line) {
  foreach (word in line)
     emit(word, 1);
 }

reduce(word, numbers) {
  int sum = 0;
  foreach (value in numbers) {
    sum += value;
  }
  emit(word, sum);
}
```

Map Reduce Illustrated (2)



Hadoop – A map/reduce Framework

- Hadoop: Apache Top Level Project
  - open Source
  - written in Java

- Hadoop provides a stack of
  - distributed file system (HDFS) – modeled after the Google File System
  - Map/Reduce engine
  - data processing languages (Pig Latin, Hive SQL)

- Runs on
  - Linux, Mac OS/X, Windows, Solaris
  - Commodity hardware

Hadoop Map/Reduce Engine

- Master / Slave architecture

- Map/Reduce Master: JobTracker
  - accepts jobs submitted by clients
  - assigns map and reduce tasks to TaskTrackers
  - monitors execution status, re-executes tasks upon failure

- Map/Reduce Slave: TaskTracker
  - runs map / reduce tasks upon instruction from the task tracker
  - manage storage, sorting and transmission of intermediate output

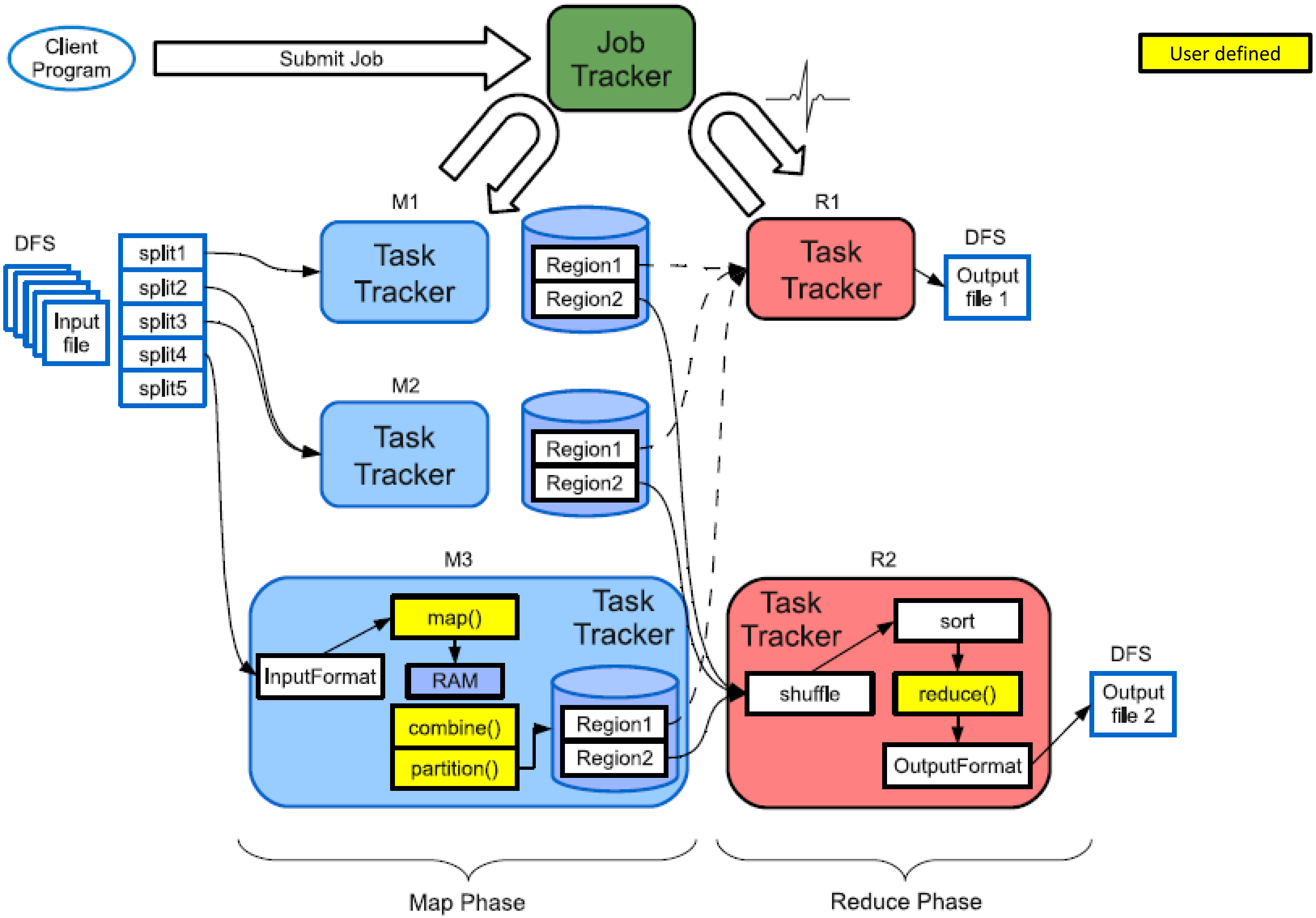
Hadoop Map/Reduce Engine

- Jobs are executed like a Unix pipeline:
  - `cat * | grep | sort | uniq -c | cat     > output`
  - `Input | Map   | Shuffle & Sort | Reduce | Output`

- Workflow
  - *input phase:* generates a number of FileSplits from input files (one per Map task)
  - *map phase:* executes a user function to transform input kv-pairs into a new set of kv-pairs
  - *sort & shuffle:* sort and distribute the kv-pairs to output nodes
  - *reduce phase:* combines all kv-pairs with the same key into new kv-pairs
  - *output phase* writes the resulting pairs to files

- All phases are distributed with many tasks doing the work
  - Framework handles scheduling of tasks on cluster
  - Framework handles recovery when a node fails

Hadoop Map/Reduce Engine



Hadoop Fault Tolerance

- Inputs are stored in a fault tolerant way by the DFS

- Mapper crashed
  - Detected when no report is given for a certain time
  - Restarted at a different node, reads a different copy of the same input split

- Reducer crashed
  - Detected when no report is given for a certain time
  - Restarted at a different node also. Pulls the results for its partition from each Mapper again.

- The key points are:
  - The input is redundantly available
  - Each intermediate result (output of the mapper) is materialized on disk
  - → Very expensive, but makes recovery of lost processes very simple and cheap