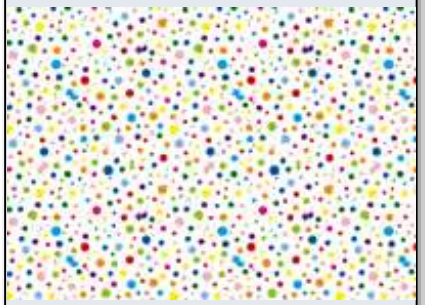
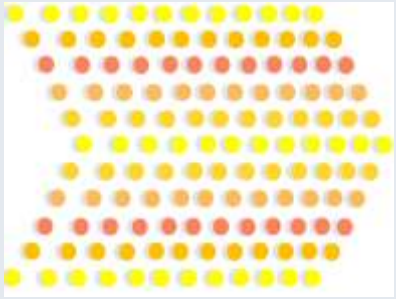
Einführung
Big Data Analytics

Prof. Dr. Stephan Trahasch
Hochschule Offenburg

Charakterisierung von Big Data durch drei V's:

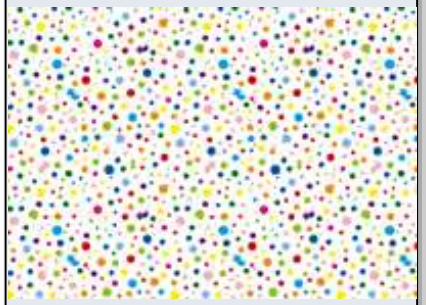
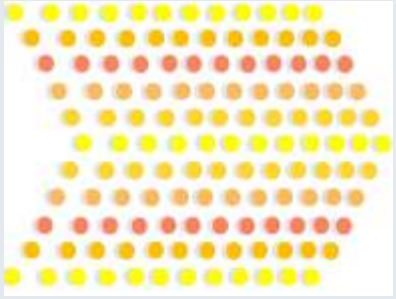
Volume	Velocity	Variety
Data at scale Terabyte to Petabytes	Data in motion Streaming Data	Data in many forms Structured, semi-structured, unstructured



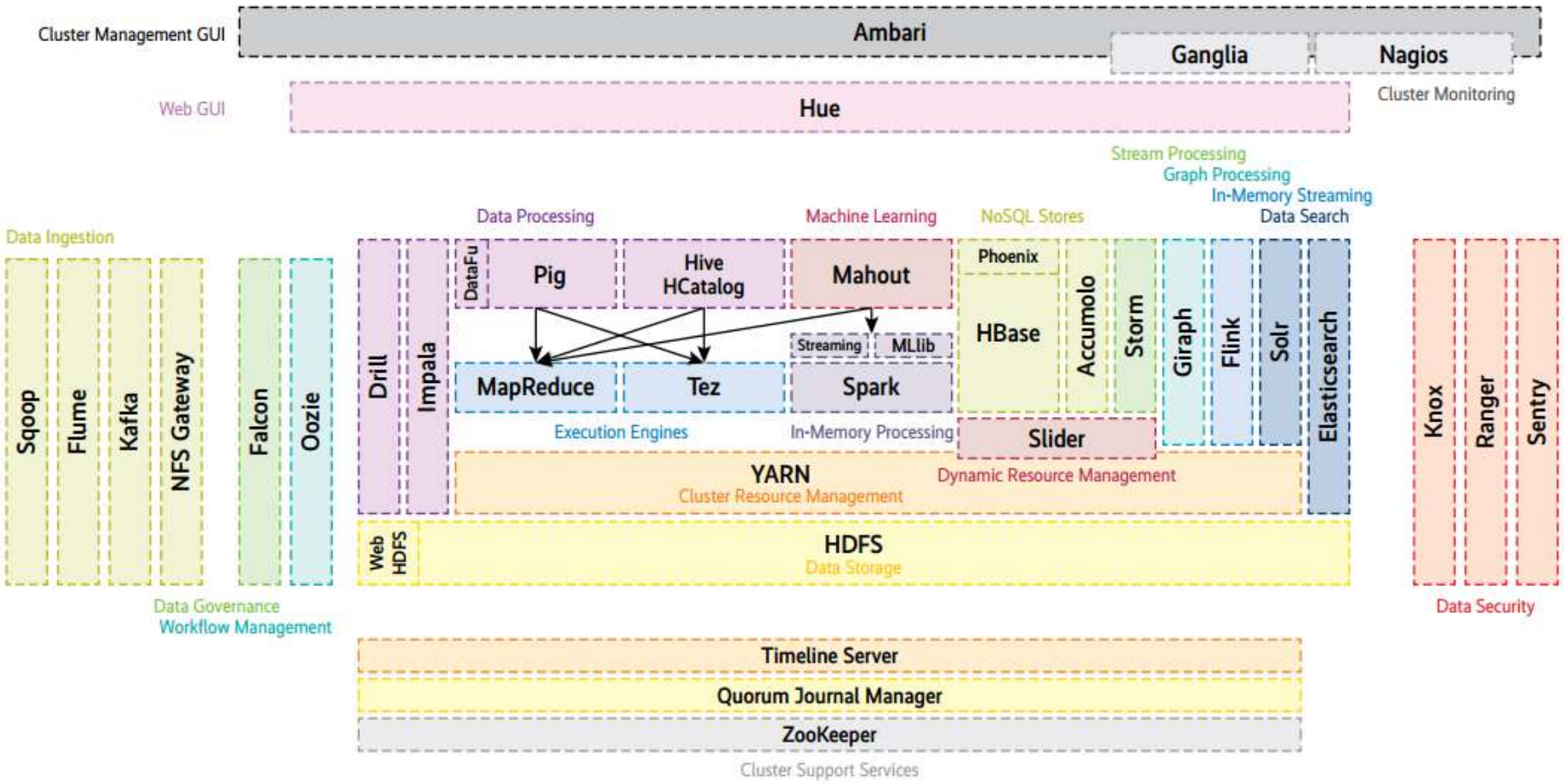


Diese Definition der Eigenschaften von Big Data erfolgt durch Gartner 2011 [1]. Das darin verwendete 3-V-Modell geht auf einen Forschungsbericht das Analysten Doug Laney zurück, der die Herausforderungen des Datenwachstums als dreidimensional bezeichnet hat [2].

Manchmal auch 4 Vs oder auch mehr Vs…

Volume	Velocity	Variety	Veracity
Data at scale Terabyte to Petabytes	Data in motion Streaming Data	Data in many forms Structured, semi-structured, unstructured	Data uncertainity Unzuverlässigkeit und Unschärfe
			

Viele Tools. Welche (Versionen) passen zusammen?



Hadoop Distributed File System

Prof. Dr. Stephan Trahasch
Hochschule Offenburg

Google File System

Reasons for developing your own file system at Google

- Very large number of read operations
- Reliability
- High througput: 40 GB/s read/write throughput
- Data is deleted, overwritten, compressed in extremely rare cases
- Data are usually appended or read out
- Petabyte of data

More than 200 clusters, each cluster with more than 5,000 servers (estimated in 2008)

Design Features

- Memory and processing scale to petabyte size.
- Storage and processing of very large files can be several hundred gigabytes in size.
- Efficient data processing as batch processing once, read-many-times
- Import of "arbitrary" data without a schema definition (schema on read vs. schema on write).
- Optimized for streaming reads of files Optional access complex.
- commodity hardware
- HDFS continues to work even during a node failure without noticeable interruption

Hadoop is less suitable

- for many small, structured data sets
- Latency times in the millisecond range
- when transactions are required
- Data must be changed frequently

Default Replication Policy

Goal of first replica: speed

- Goes to data node closest to the client

Goal of second replica: availability

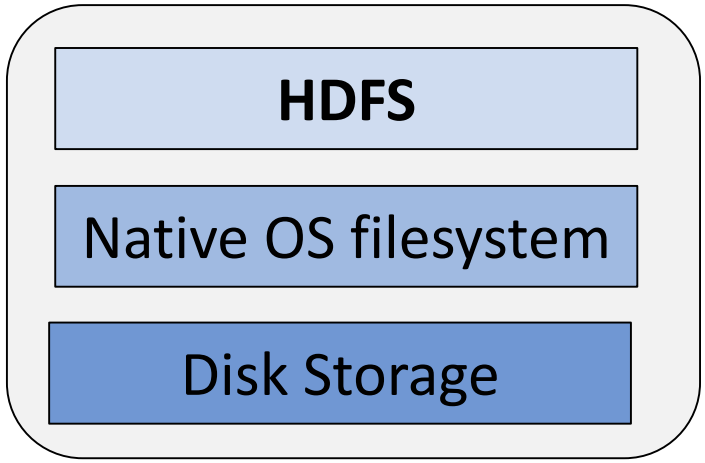
- If rack awareness if configured, each DataNode will have a rack id associated with it
- Second replica goes to a different rack in the cluster

Goal of third replica: minimizing inter-rack network traffic

- Goes to a different node on the same rack as second replica

Hadoop File System (HDFS)

- Basiert auf GFS
- HDFS ist ein Filesystem, programmiert in Java
- Setzt auf einem vorhanden Filesystem auf
- Block Size 128MB
- Blocks werden 3x über DataNodes repliziert
- MasterNode (master) und DataNode (worker)



Why HDFS Blocks are Large in Size?

HDFS blocks are large in size → reduce cost of seek time.
Seek time is 10ms and disk transfer rate is 100MB/s. To make the seek time 1% of the disk transfer rate, block size should be 100MB. Default size HDFS block is 128MB.

Advantages of HDFS Block

- fixed size, it is very easy to calculate the number of blocks that can be stored on a disk.
- If size of a file is less than HDFS block size, then the file does not occupy the complete block storage.
- It is easy to store a file that is larger than disk size. Data blocks are distributed and stored on multiple nodes in a cluster.
- Blocks are easy to replicate between the datanodes and thus provide fault tolerance and high availability.

Hadoop setzt sich aus zwei unterschiedlichen Nodes

1. **NameNode**
   Verwaltet die Metadaten für HDFS.
   Optional: Secondary oder Standby NameNode
   - Secondary NameNode übernimmt Hintergrundarbeiten für NameNode, ist aber kein Backup oder Hot-Standby
   - Standby NameNode für HA Modus
     Im Hochverfügbarkeitsmodus ist dieser NameNode gleichzeitig Hot-Standby als auch Hintergrundarbeiter.

2. **DataNode**
   Speichert die eigentlichen HDFS-Datenblöcke.

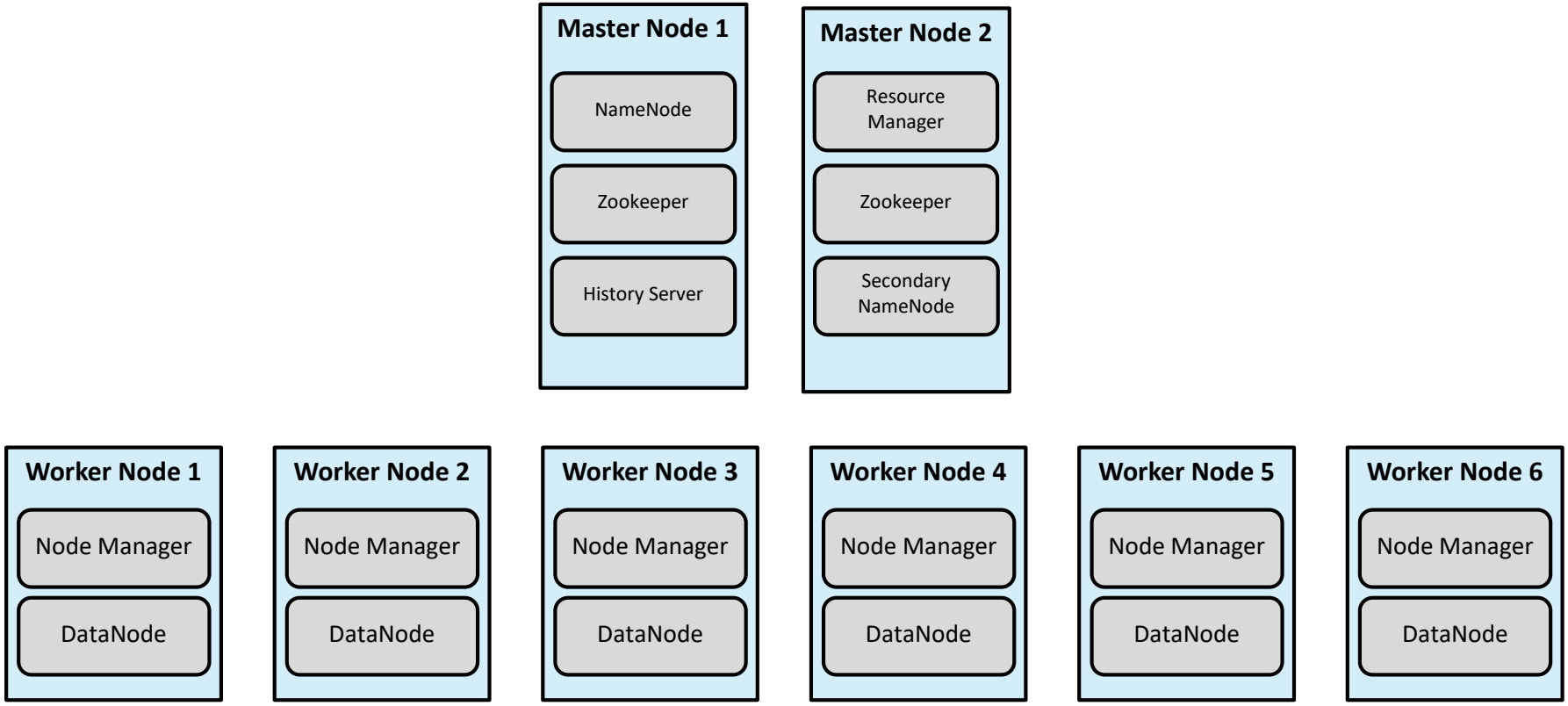
Auf den Nodes laufen mehrere Prozesse.

Hadoop cluster is made up of master and worker (data) nodes

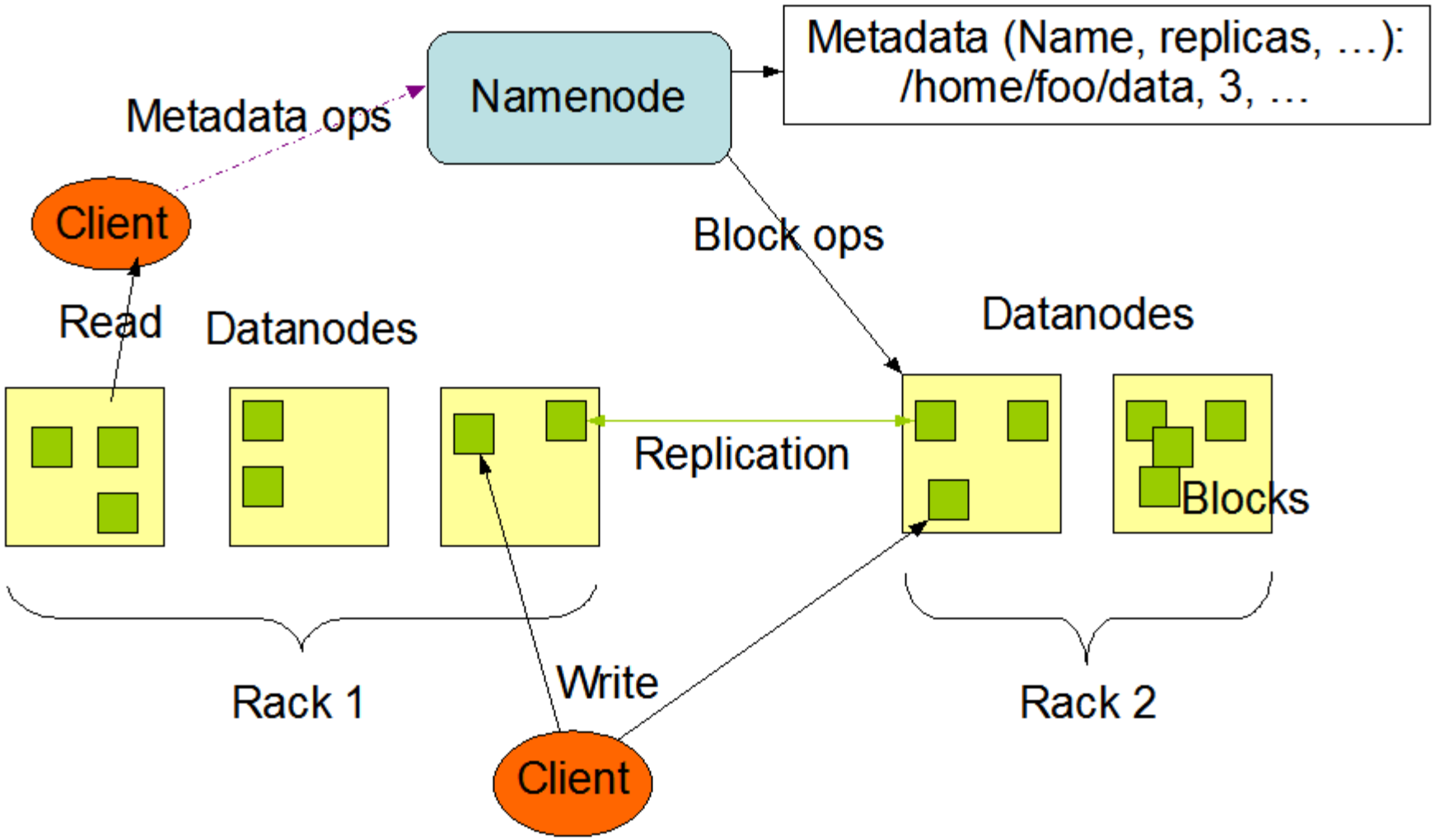
Master nodes manage (NameNode) the infrastructure

Without metadata on NameNode, there is no way to access files

Worker nodes contain the distributed data and perform processing

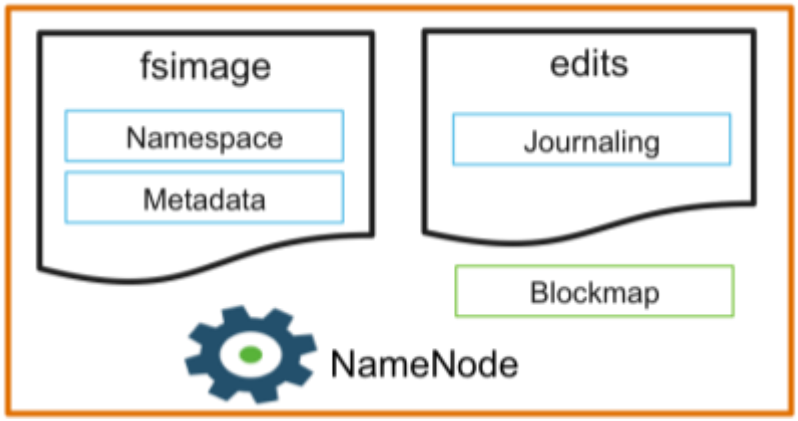


HDFS-Architektur



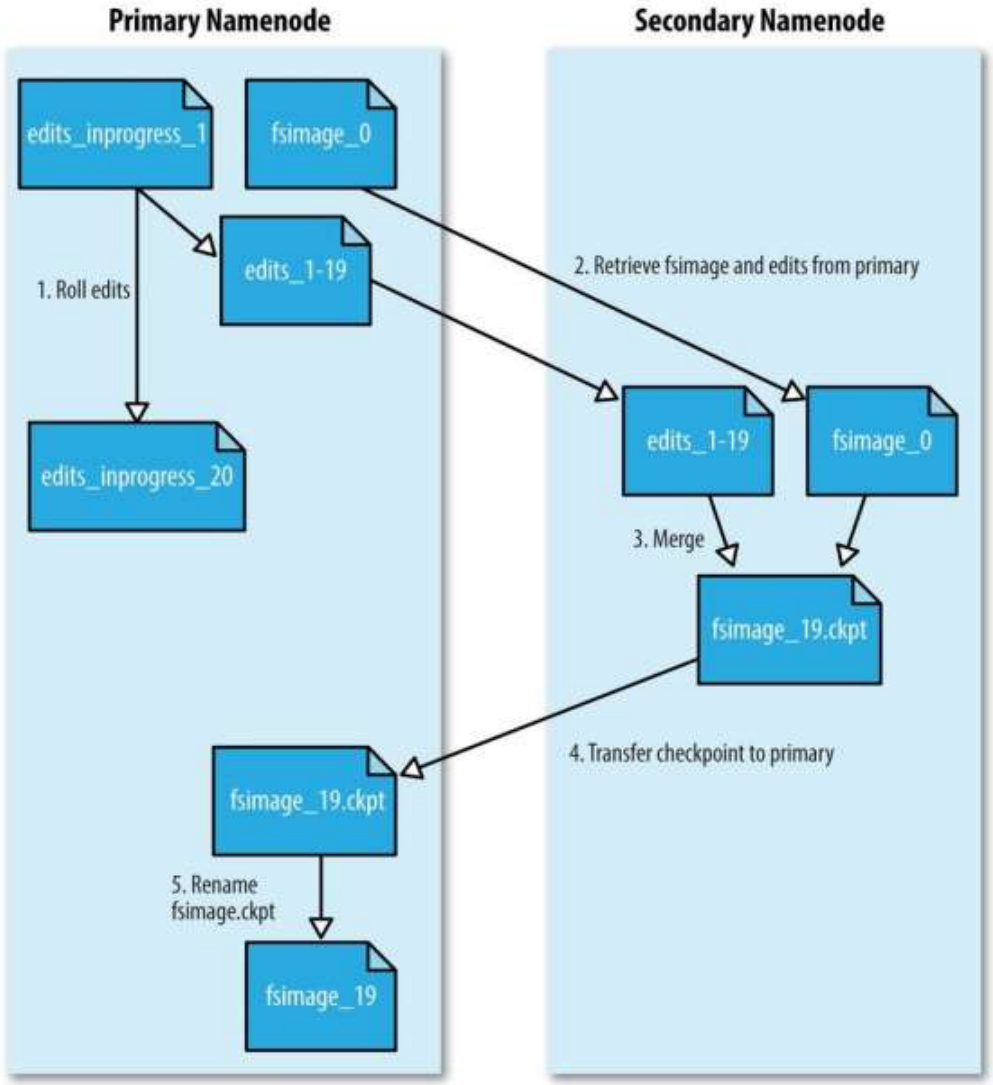
NameNode tracks changes with edit-log and fsimage

```
${dfs.namenode.name.dir}/
├──current
│   ├── VERSION
│   ├── edits_0000000001-0000000019
│   ├── edits_inprogress_0000000000020
│   ├── fsimage_0000000000000000
│   ├── fsimage_0000000000000000.md5
│   ├── fsimage_0000000000000019
│   ├── fsimage_0000000000000019.md5
│   └── seen_txid
└── in_use.lock
```



1. NameNode starts: reads **fsimage_N** and **edits_N**
2. Transactions in **edits_N** merged with **fsimage_N**
3. **fsimage_N+1** is written, **edits_N+1** is created

Checkpoint Process with Secondary NameNode

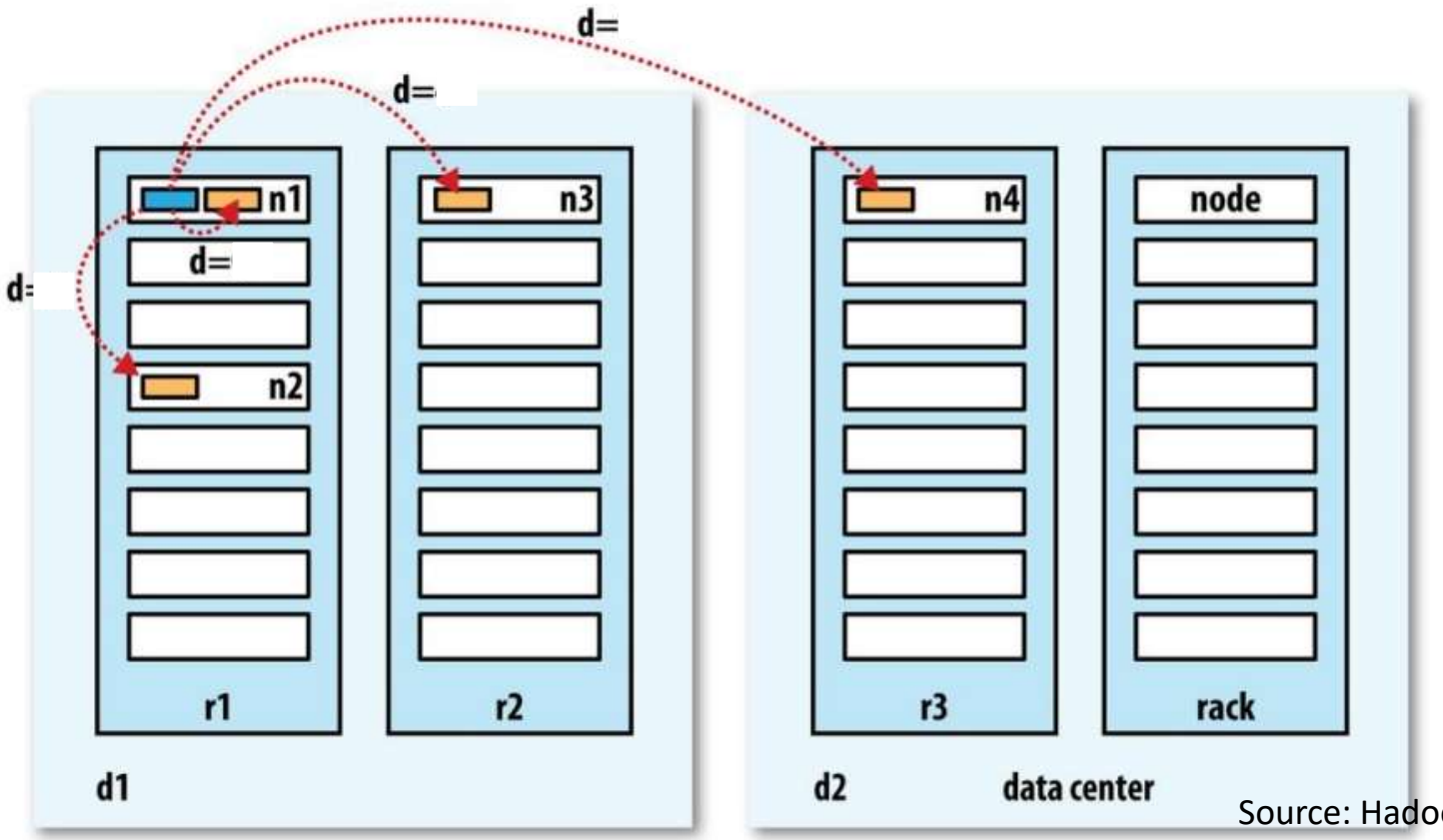


Source: Hadoop – The Definitive Guide

Network Distance

Network wird als Baum repräsentiert.

Distanz zwischen Knoten ist Summe der Distanz zum ersten gemeinsamen Elternknoten.



Source: Hadoop – The Definitive Guide

MapReduce Algorithms

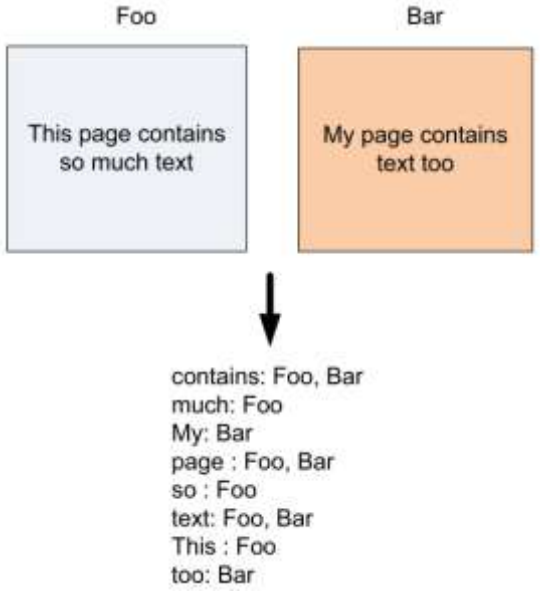
Prof. Dr. Stephan Trahasch
Hochschule Offenburg

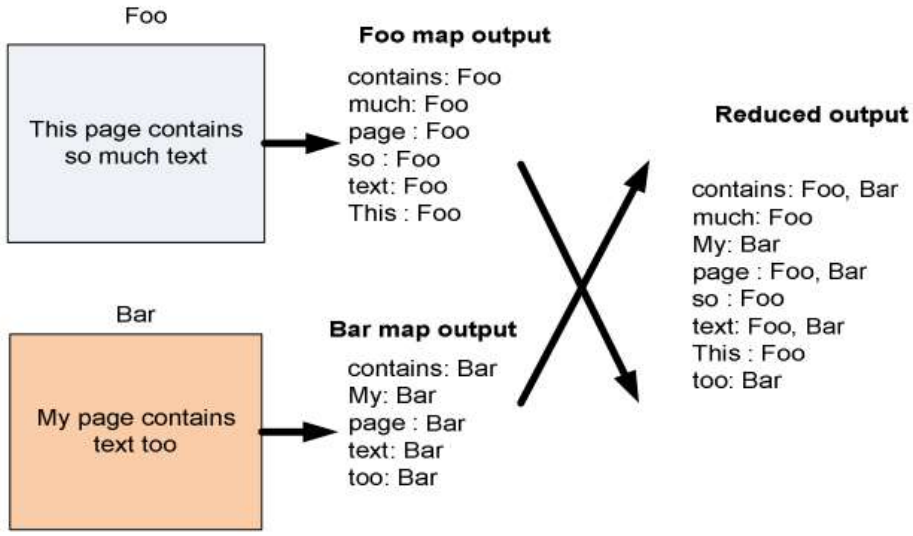
Source: Parsian, Mahmoud (2015): Data algorithms. Recipes for scaling up with Hadoop and Spark. O'Reilly Media.

Indexing

Inverted Index for a Search Engine

- Assume the input is a set of files containing lines of text
- Key is the byte offset of the line, value is the line itself
- We can retrieve the name of the file using the Context object





Inverted Index: MapReduce

Mapper:

```
For each word in the line, emit (word, filename)
```

Reducer: Identity function

Collect together all values for a given key (i.e., all filenames for a particular word)

```
Emit (word, filename_list)
```

Secondary Sort

A secondary sort problem relates to sorting values associated with a key in the reduce phase.

MapReduce framework automatically sorts the keys generated by mappers.

But values passed to each reducer are not sorted at all; they can be in any order.

$\texttt{map}(\texttt{key}_1, \texttt{value}_1) \rightarrow \texttt{list}(\texttt{key}_2, \texttt{value}_2)$

$\texttt{reduce}(\texttt{key}_2, \texttt{list}(\texttt{value}_2)) \rightarrow \texttt{list}(\texttt{key}_3, \texttt{value}_3)$

Consider key-value pair ($\texttt{key}_2$, $\texttt{list}(\texttt{value}_2)$) as an input for a reducer

$\texttt{list}(\texttt{value}_2) = (V_1, V_2, \ldots, V_n)$

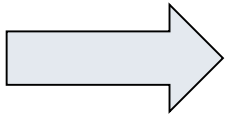
Secondary Sort sorts the values received by a reducer in some order.

$SORT(V_1, V_2, ..., V_n) = (S_1, S_2, ..., S_n)$

$list(value_2) = (S_1, S_2, ..., S_n)$

where: $S_1 < S_2 < ... < S_n$, or $S_1 > S_2 > ... > S_n$

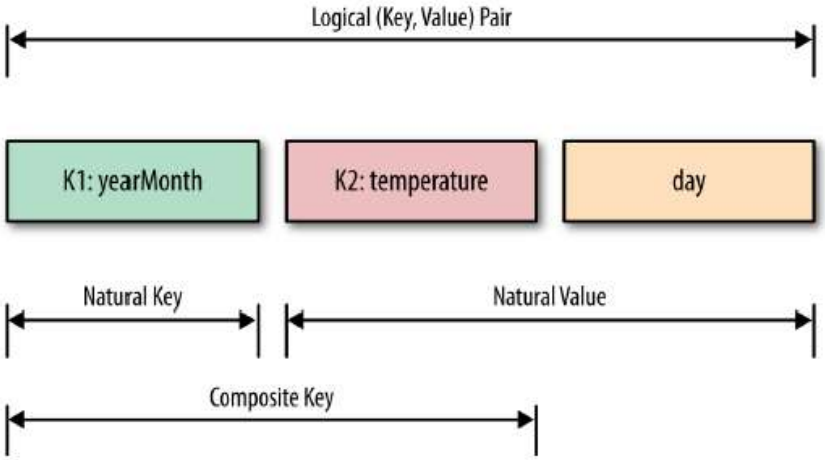
```
2012, 01, 01, 5
2012, 01, 02, 45
2012, 01, 03, 35
2012, 01, 04, 10
...
2001, 11, 01, 46
2001, 11, 02, 47
2001, 11, 03, 48
2001, 11, 04, 40
...
2005, 08, 20, 50
2005, 08, 21, 52
2005, 08, 22, 38
2005, 08, 23, 70
```



```
2012-01: 5, 10, 35, 45, ...
2001-11: 40, 46, 47, 48, ...
2005-08: 38, 50, 52, 70, ...
```

Use the *Value-to-Key Conversion* design pattern

1. Form a composite intermediate key, (K, V_1), where V_1 is the secondary key. Here, K is called a *natural key*. To inject a value (i.e., V_1) into a reducer key, simply create a composite key.
2. Let the MapReduce execution framework do the sorting
3. Preserve state across multiple key-value pairs to handle processing; you can achieve this by having proper mapper output partitioners



Source: Data Algorithms

Top N List

Problem:

Given a set of (key-as-string, value-as-integer) pairs, we want to create a top N (where N > 0) list.

Examples for this type of problem:

TopN formalized

Let L be a List<Tuple2<T,Integer>>, where T can be any type (such as a string or URL); L.size() = S and S > N with N > 0.

Elements of L: $\{(K_i, V_i), 1 \leq i \leq S\}$

- where K_i has a type of T and
- V_i is an Integer type (this is the frequency of K_i).

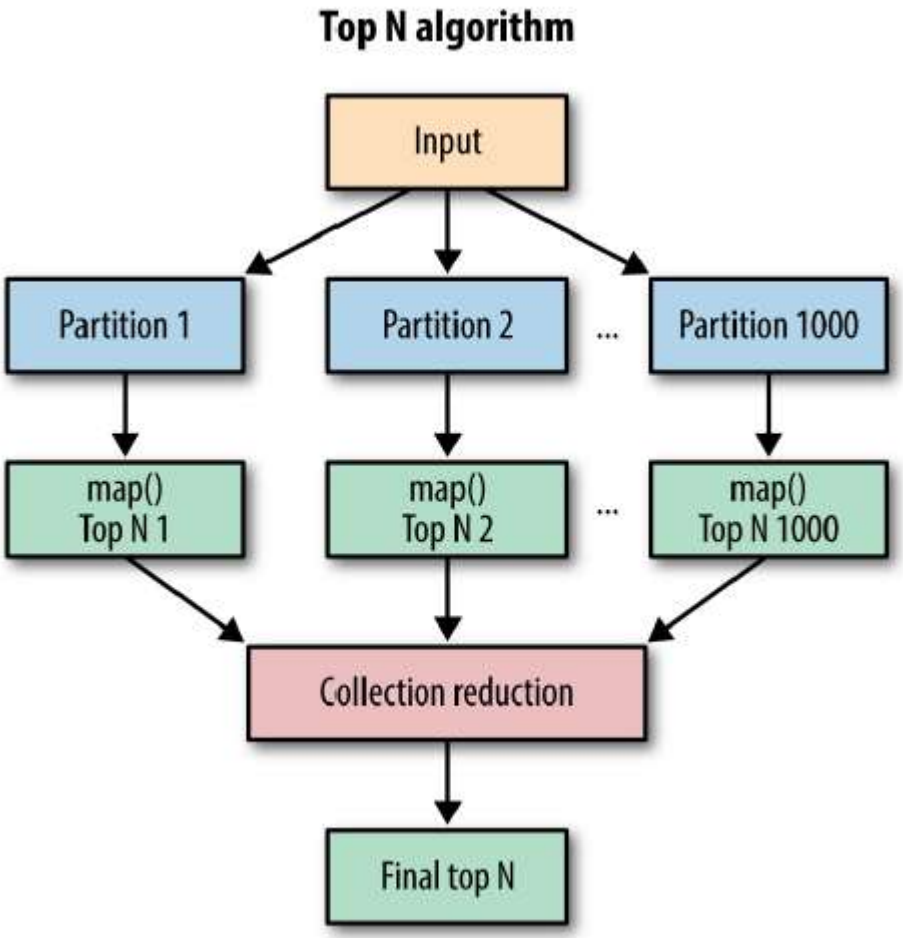
sort(L) returns sorted values of L by using the frequency as a key:

$\{(A_j, B_j), 1 \leq j \leq S; B_1 \geq B_2 \geq \ldots \geq B_S\}$ where $(A_j, B_j) \in L$.

The top N of L is defined as:

$\text{topN}(L) = \{(A_j, B_j), 1 \leq j \leq N, B_1 \geq B_2 \geq \ldots \geq B_N \geq B_{N+1} \geq \ldots \geq B_S\}$

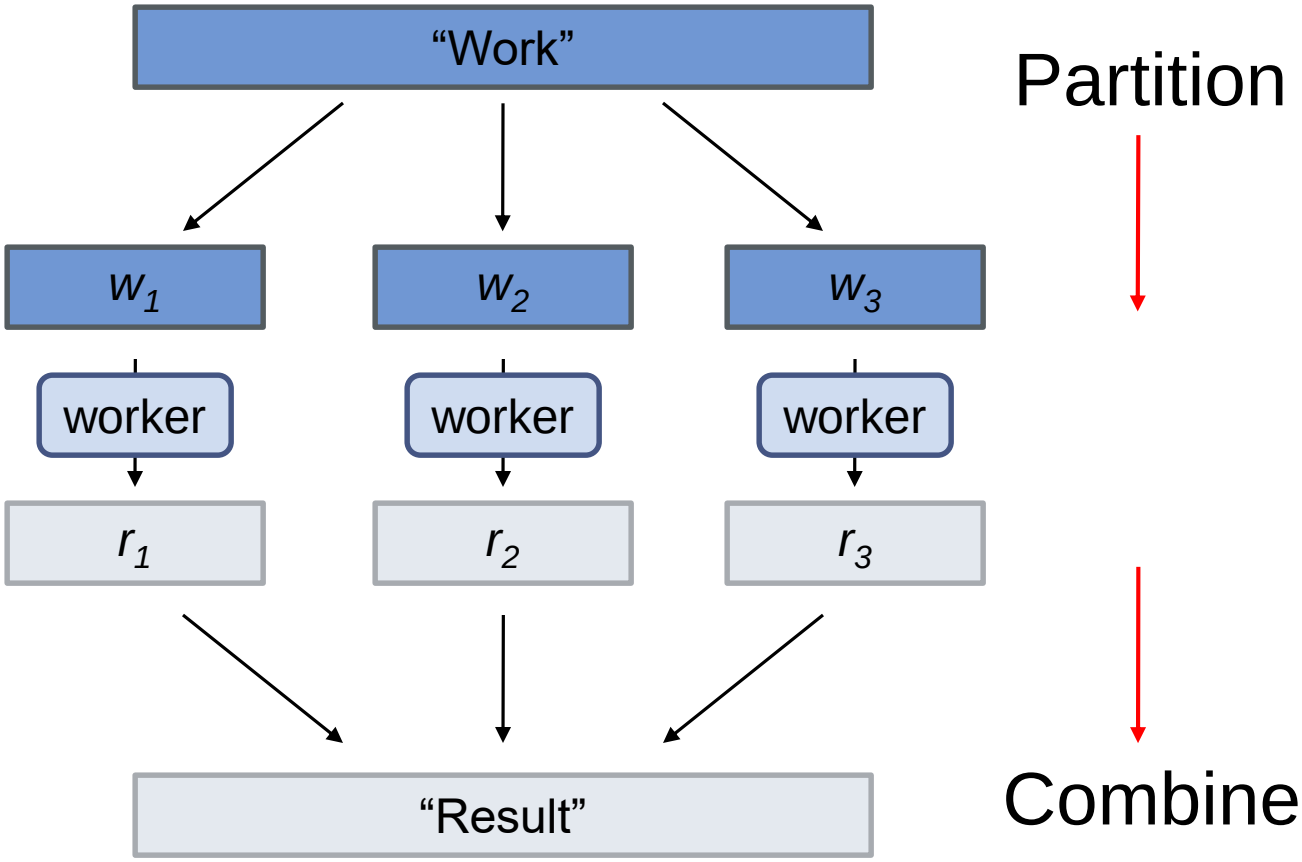
TopN: Overview MapReduce



Introduction to MapReduce

Prof. Dr. Stephan Trahasch
Hochschule Offenburg

Divide and Conquer



MapReduce

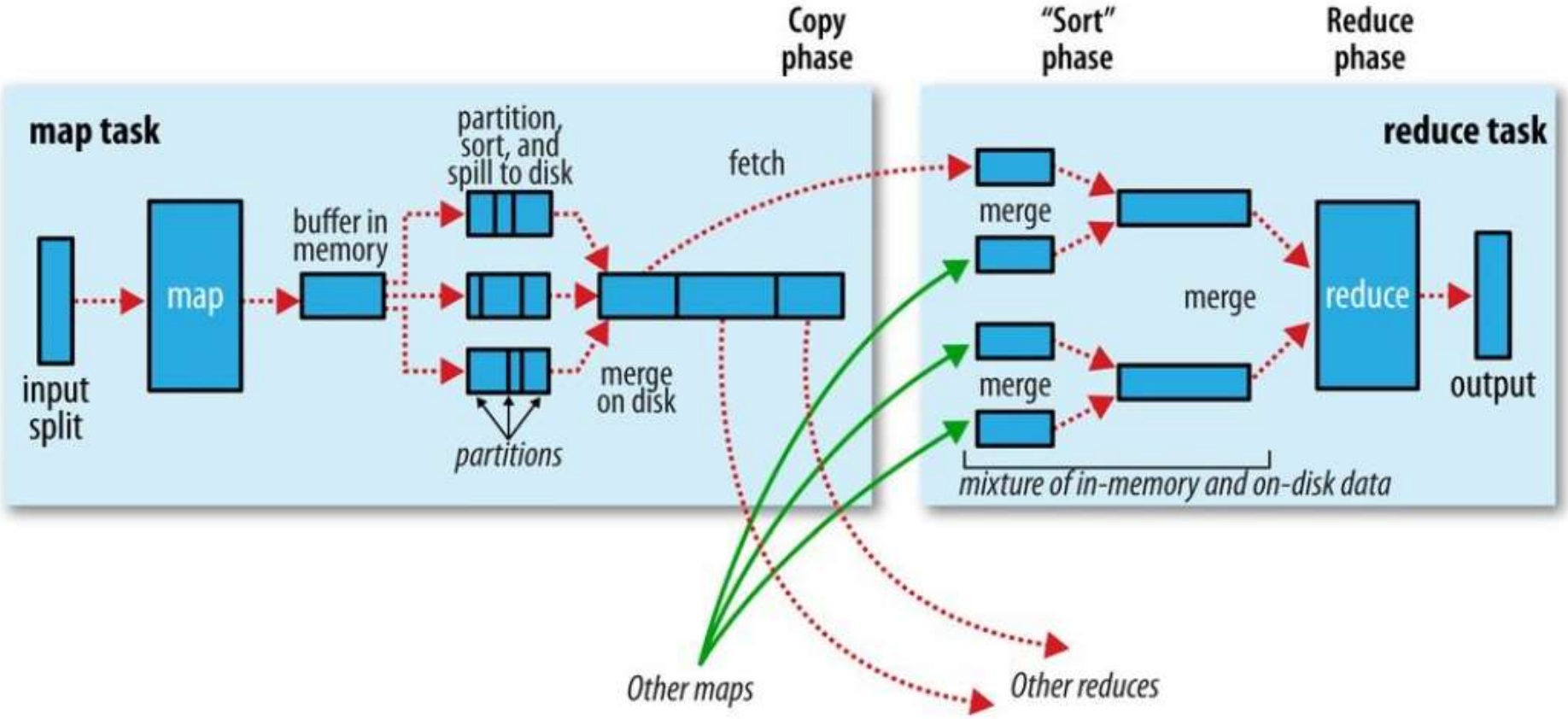
1. **Map**
   - Mappers map input data to intermediate key/value pairs
     parse, filter, or transform the data
   - Each Map task operates on a single HDFS block
   - run on the node where the block is stored

2. **Shuffle and Sort**
   - Sorts and consolidates intermediate data from all mappers
   - Happens after all Map tasks are complete and before Reduce tasks start → place for optimization

3. **Reduce**
   - Reducers process Mapper output into final key/value pairs
     aggregate data using statistical functions
   - Operates on Map task output (shuffled/sorted intermediate data)

The execution framework handles everything else…

Details zu 2. Shuffle and Sort



Quelle: Hadoop – The Definitive Guide

Map: Buffer and Combiner

- Each map task has circular memory buffer that it writes output to. Default buffer size is 100 MB.
- If buffer size reaches a threshold size (default 80%)
  → background thread starts to spill the contents to disk
- Spills are written in round-robin fashion to directories in a job-specific subdirectory
- Before it writes to disk, thread divides the data into partitions corresponding to the reducers that they will be sent to. Within each partition, the thread performs an in-memory sort by key.
- Combiner Function
  If there is a combiner, it is run on output of sort. Running the combiner function makes for a more compact map output. There is less data to write to local disk and to transfer to the reducer.

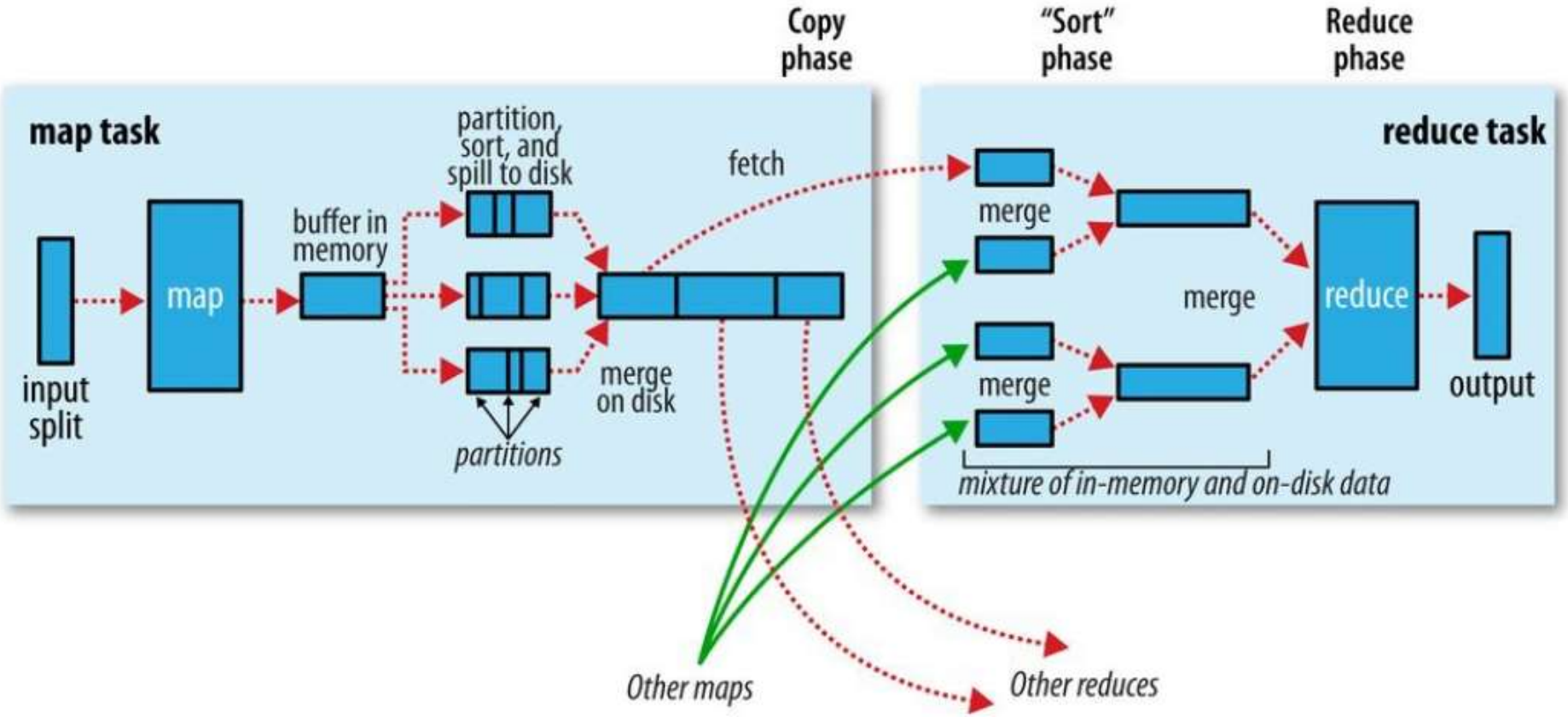
Map: Spill

- Each time the memory buffer reaches the spill threshold, a new spill file is created.
- After the map task has written its last output record, there could be several spill files.
- Before the task is finished, the spill files are merged into a single partitioned and sorted output file.
- Per default 10 files are merged at once.
- If there are at least three spill files the combiner is run again before the output file is written.
- Combiners may be run repeatedly over the input without affecting the final result.

Map: Compression

Compressing the map output as it is written to disk

→ writing is faster, saves disk space, reduces amount of data to transfer to the reducer.



Reduce Part of Shuffle & Sort

- Map output file is sitting on the local disk of the “Map Machine”
  - Map outputs always written to local disk,
  - Reduce outputs may not be.
- Data of partition is needed by the “Reduce Machine”
- Moreover, reduce task needs the map output from several map tasks across the cluster!
- Map tasks may finish at different times.
- Reduce task starts copying the outputs of Map tasks as soon as each completes: “copy phase of the reduce task”.
- Reduce task has a small number of copier threads so that it can fetch map outputs in parallel. Default 5 threads.

How do reducers know which machines to fetch map output from?

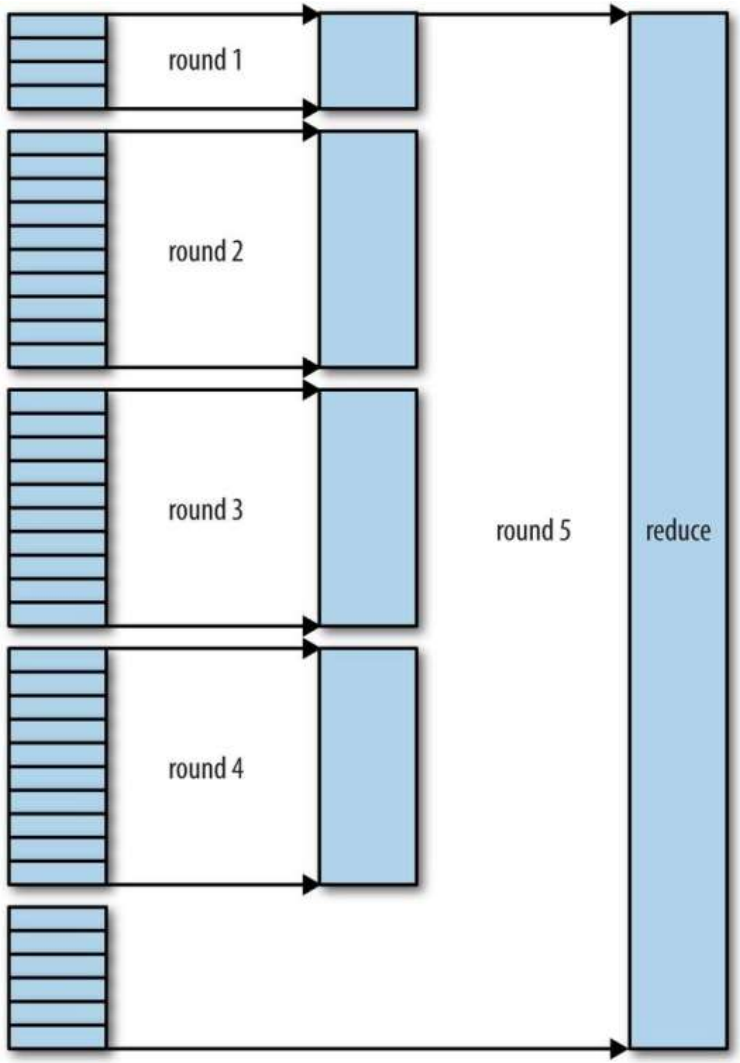
- As map tasks complete successfully, they notify their application master using the heartbeat mechanism.
- For a given job, the application master knows the mapping between map outputs and hosts.
- A thread in the reducer periodically asks master for map output hosts until it has retrieved them all.

- Hosts do not delete map outputs from disk as soon as the first reducer has retrieved them.
- Reducer may subsequently fail!
- Instead, they wait until they are told to delete them by the application master, which is after the job has completed.

Copy Phase of the Reduce Task

- Map outputs are copied to reduce task JVM's memory if they are small enough; otherwise, they are copied to disk.
- When the in-memory buffer reaches a threshold size or reaches a threshold number of map outputs
  → then it is merged and spilled to disk.
- If a combiner is specified, it will be run during the merge to reduce the amount of data written to disk.
- As the copies accumulate on disk, a background thread merges them into larger, sorted files.
- This saves some time merging later on.

Sort Phase



When all the map outputs have been copied, the reduce task starts the sort phase.

Merges map outputs, maintaining their sort ordering.

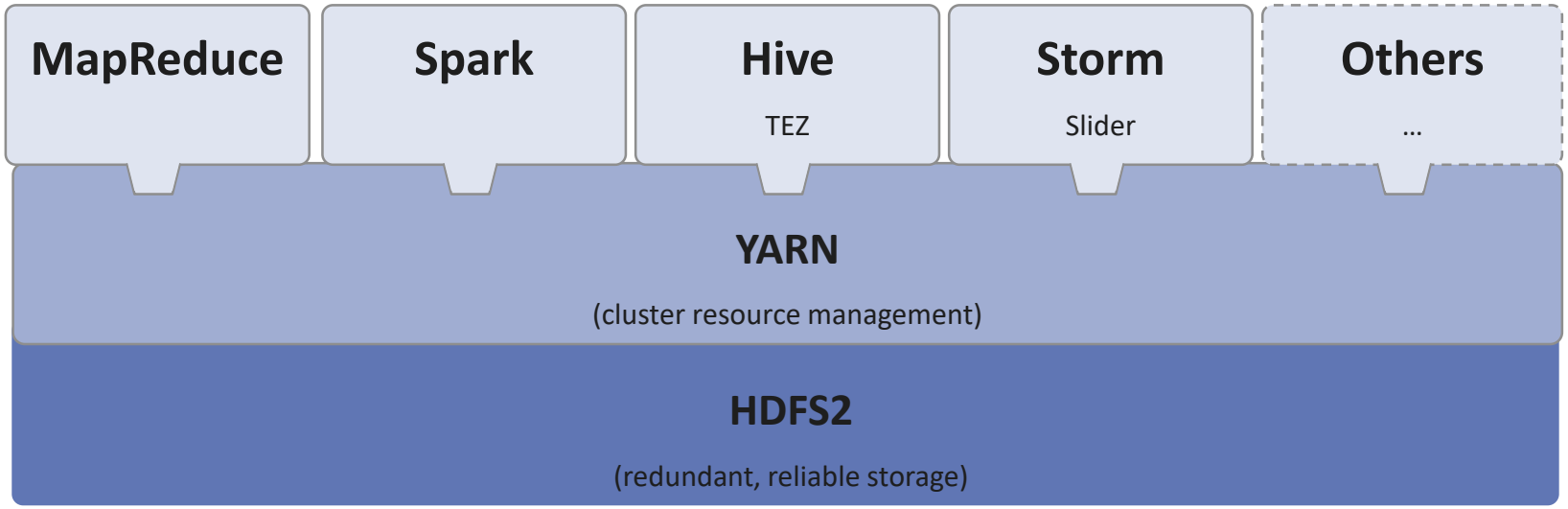
This is done in rounds.

Example, if there were 50 map outputs and the merge factor was 10, there would be five rounds. Each round would merge 10 files into 1, so at the end there would be 5 intermediate files.

YARN
Yet Another Resource Negotiator

Prof. Dr. Stephan Trahasch
Hochschule Offenburg

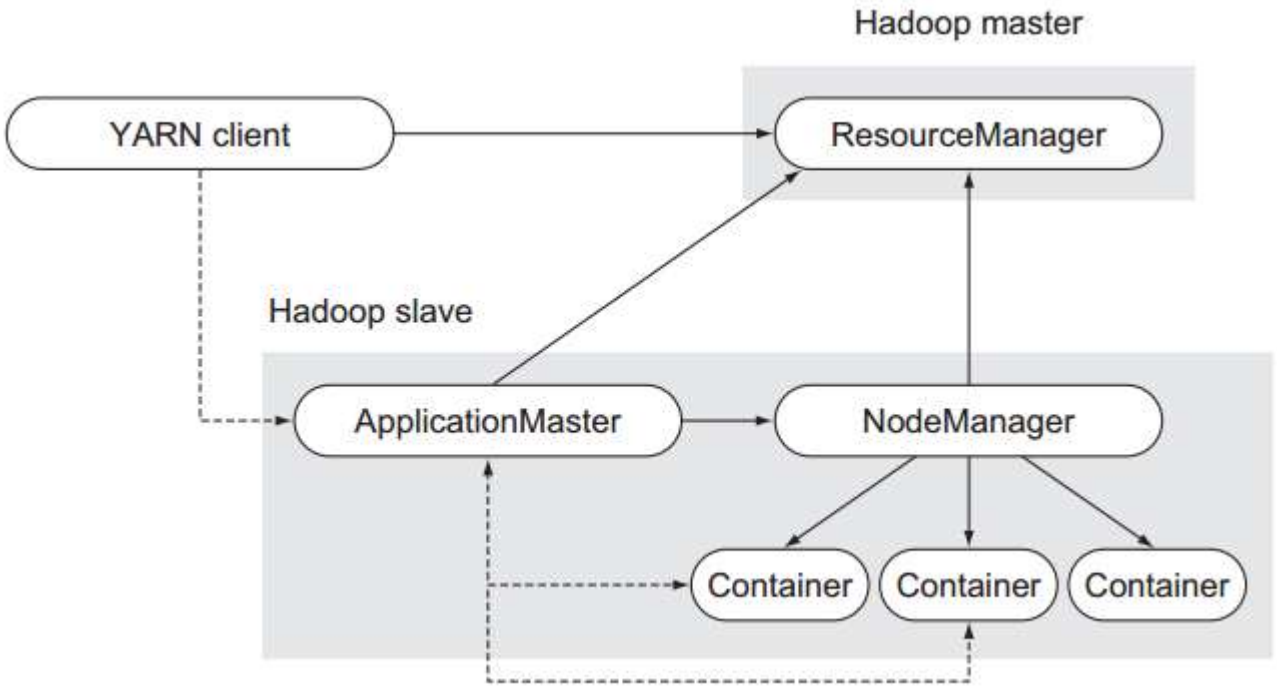
YARN - Yet Another Resource Negotiator



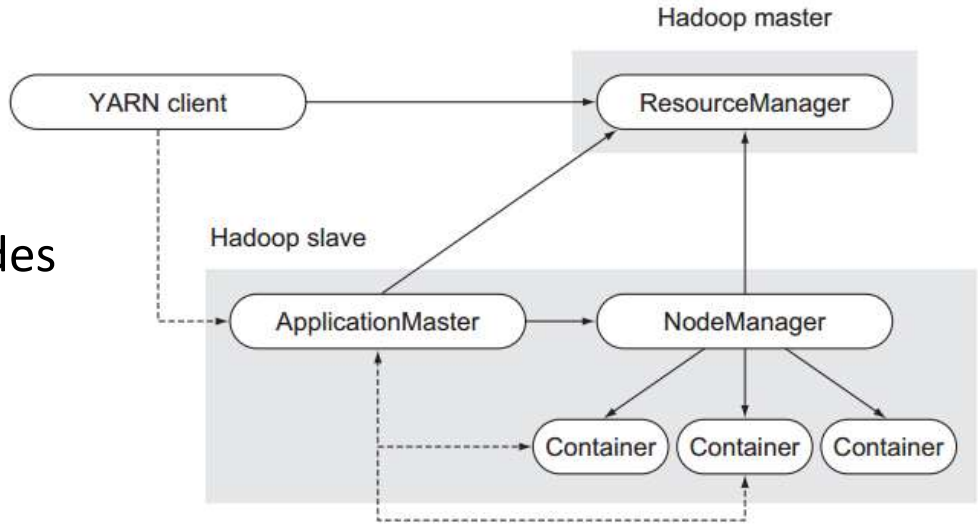
- Cluster Resource Management for distributed data processing
- Dynamic Allocation of resources for tasks.
- allows multiple data processing engines to run on a single Hadoop cluster: MapReduce, Spark, Tez, Storm, Giraph …

YARN - Splits up the two major responsibilities into separate entities

- a global ResourceManager (RM)
- a per-application ApplicationMaster (AM)
- a per-node slave NodeManager (NM)
- a per-application Container running on a NodeManager



Responsibility



1. **Resource Manager**
   - One per Cluster
   - Starts Application Masters
   - Allocates Resources on Slave Nodes
2. **Application Master**
   - One per Job (Application)
   - Framework/application specific
   - Runs in a container
   - Requests more containers to run application tasks
3. **Node Manager**
   - One per Slave
   - Managed Resources on a Slave Node
4. **Container**
   - Created by the RM upon request
   - Allocate a certain amount of resources (memory, CPU) on slave node
   - Applications run in one or more containers

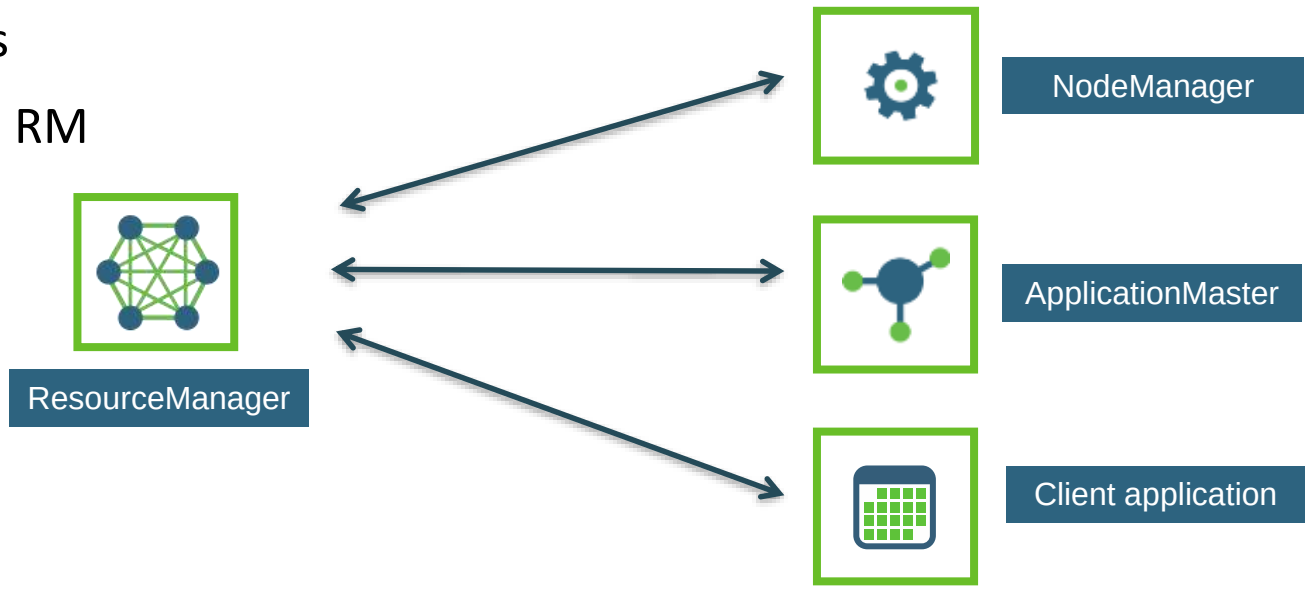
Resource Manager and Node Manager

Resource Manager (RM)

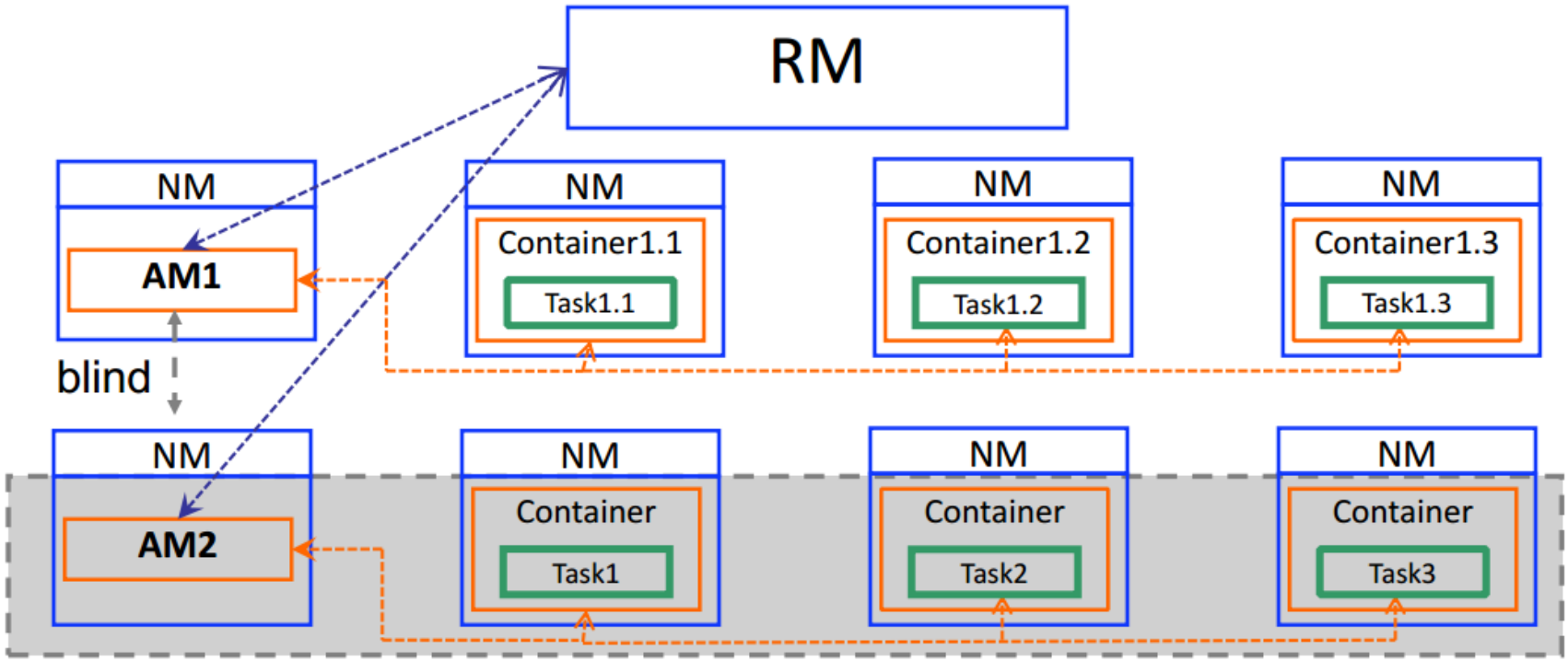
- Runs on master node and is the global resource scheduler
- scheduler supports different algorithms (capacity, fair scheduler, etc.)
- communicates with the NodeManagers, ApplicationMasters, and Client applications

Node Manager (NM)

- Runs on slave nodes
- Communicates with RM



Visibility of Application Masters (AM)

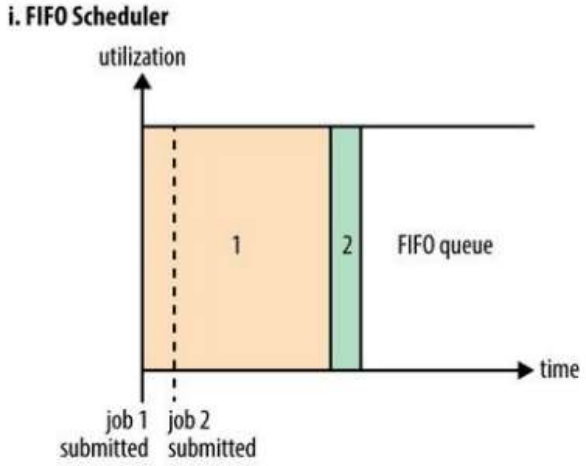


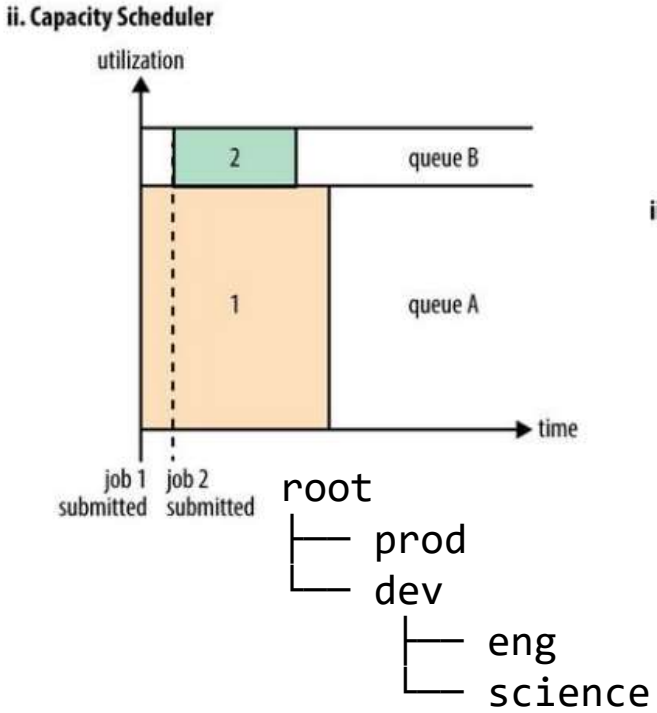
- AM is like the job tracker in Hadoop 1.0
  - Creates and manages task lifecycle
  - Monitors task status
- AM has no view of other running applications.

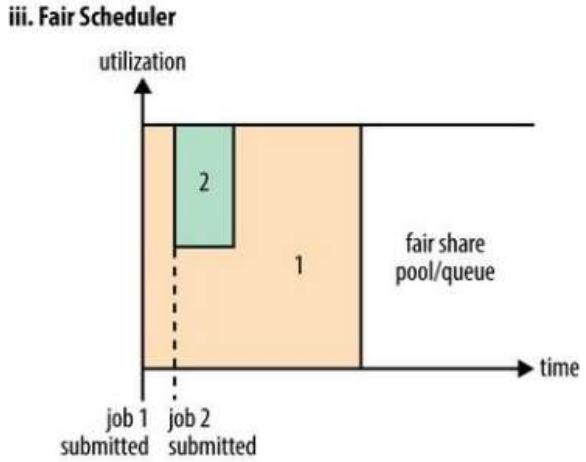
Scheduling in YARN, part of Resource Manager

Scheduler is responsible for allocating resources to the various running applications.

FIFO, Capacity, Fair Schedulers:







Source: Hadoop – The Definitive Guide

Apache Hive

Prof. Dr. Stephan Trahasch
Hochschule Offenburg

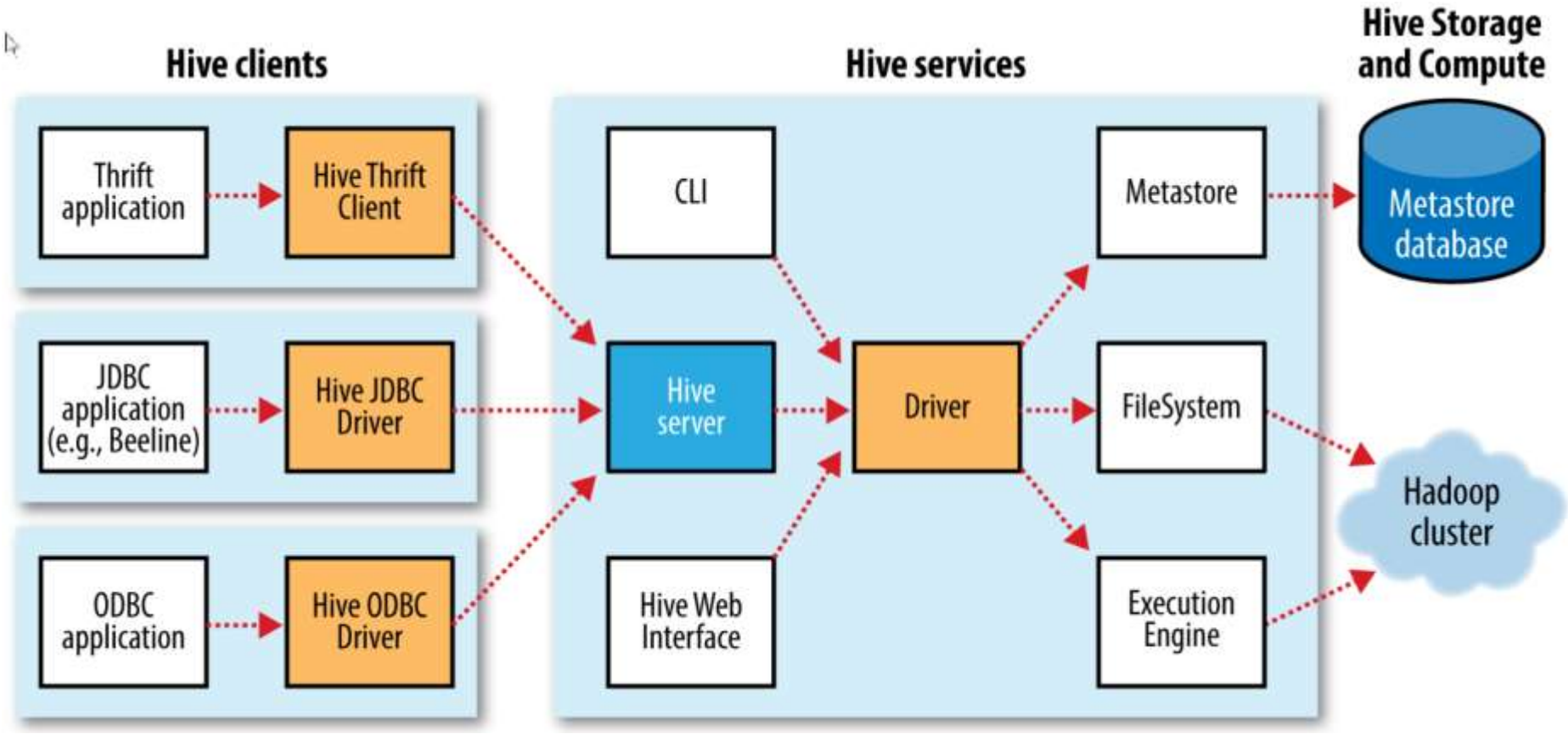
What Is Hive?

- A data warehousing infrastructure based on Hadoop
- Hive is designed and great for batch processing
- Immediately makes data on a cluster available to non-Java programmers via SQL like queries
- Built on HiveQL (HQL), a SQL-like query language
- Interprets HiveQL and generates MapReduce jobs that run on the cluster
- Enables easy data summarization, ad-hoc reporting and querying, and analysis of large volumes of data
- Developed by Facebook and a top-level Apache project

What Hive Is Not

- Hive, like Hadoop, is designed for batch processing of large datasets
- Not an OLTP or real-time system
- Latency and throughput are both high compared to a traditional RDBMS!
- Even when dealing with relatively small data (<100 MB)

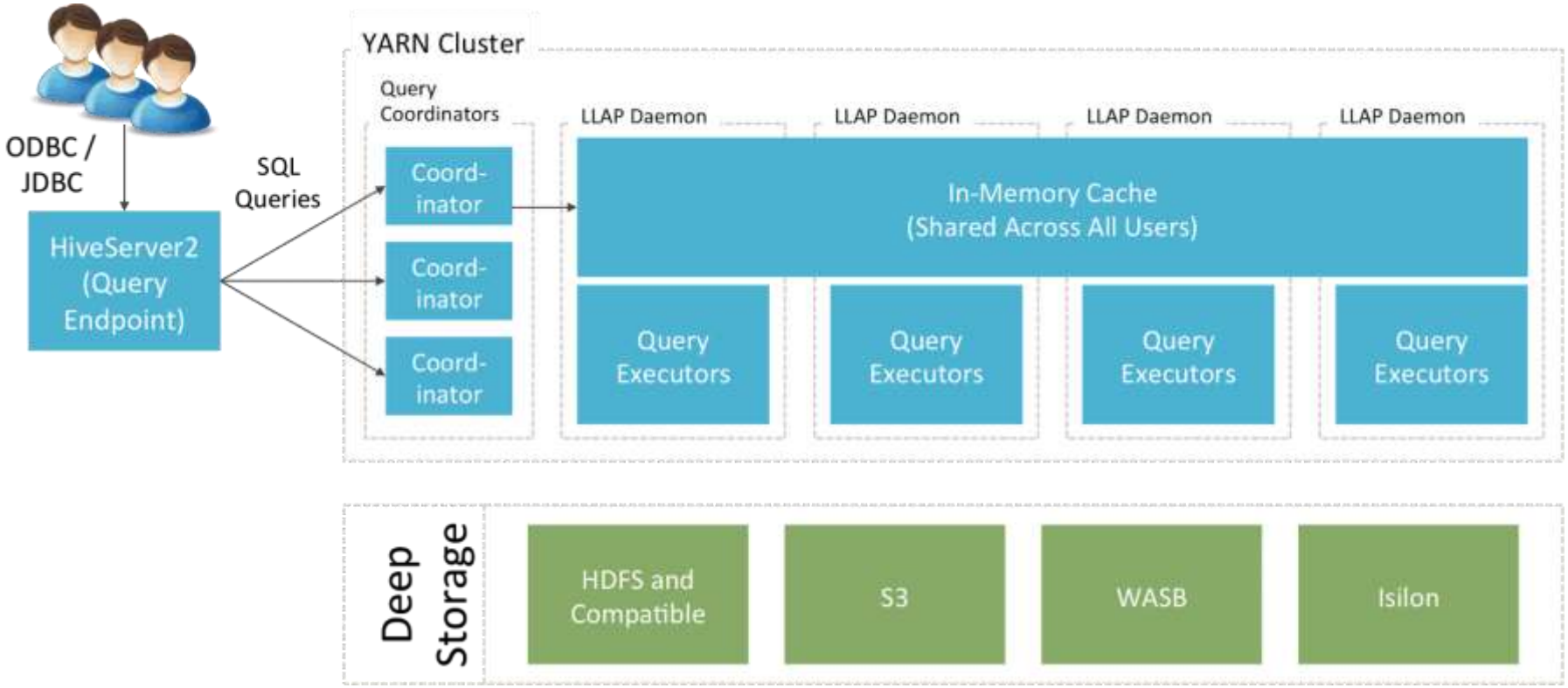
Hive Architecture



Source: Hadoop – The definitive Guide

Hive 2

Hive 2 shifts from a disk-centric architecture to a memory-centric architecture through Hive LLAP (Live Long and Process).



https://de.hortonworks.com/blog/apache-hive-going-memory-computing/

Hive LLAP - Live Long and Process

- LLAP uses persistent query servers to avoid long startup times and deliver fast SQL.
- LLAP shares its in-memory cache among all SQL users, maximizing the use of this scarce resource.
- Pre-fetching and caching of column chunks
- LLAP has fine-grained resource management and preemption, making it great for highly concurrent access across many users.
- Multi-threaded JIT-friendly operator pipelines

LLAP provides a hybrid execution model.

- LLAP consists of a long-lived daemon which replaces direct interactions with the HDFS DataNode, and a tightly integrated DAG-based framework.
- Functionality such as caching, pre-fetching, some query processing and access control are moved into the daemon.
- Small/short queries are largely processed by this daemon directly, while any heavy lifting will be performed in standard YARN containers.
- Similar to the DataNode, LLAP daemons can be used by other applications as well.

Persistent Daemon runs on the worker nodes

- To facilitate caching and JIT optimization, and to eliminate most of the startup costs. It handles I/O, caching, and query fragment execution.
- These nodes are stateless. Any request to an LLAP node contains the data location and metadata. It processes local and remote locations; locality is the caller's responsibility (YARN).
- Recovery/resiliency
  Failure and recovery is simplified because any data node can still be used to process any fragment of the input data. The Tez AM can thus simply rerun failed fragments on the cluster.
- Communication between nodes
  LLAP nodes are able to share data (e.g., fetching partitions, broadcasting fragments). This is realized with the same mechanisms used in Tez.

Execution Engine does not replace the existing execution model but rather enhances it.

The daemons are optional. Hive can work without them and also is able to bypass them even if they are deployed and operational. Feature parity with regard to language features is maintained.

- External orchestration and execution engines.
  LLAP is not an execution engine like MapReduce or Tez. Execution is scheduled and monitored by an existing Hive execution engine (such as Tez) transparently over both LLAP nodes, as well as regular containers.
- Partial execution.
  The result of the work performed by an LLAP daemon can either form part of the result of a Hive query, or be passed on to external Hive tasks, depending on the query.
- Resource Management.
  YARN remains responsible for the management and allocation of resources. The YARN container delegation model is used to allow the transfer of allocated resources to LLAP.

Controlling Table Data Location

- By default, table data is stored in the warehouse directory
- This is not always ideal
  Data might be shared by several users
- Use LOCATION to specify the directory where table data resides

```
CREATE TABLE jobs (
    id INT, title STRING, salary INT, posted TIMESTAMP
  )
  ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
  LOCATION '/loudacre/jobs';
```

Source: Cloudera

Externally Managed Tables

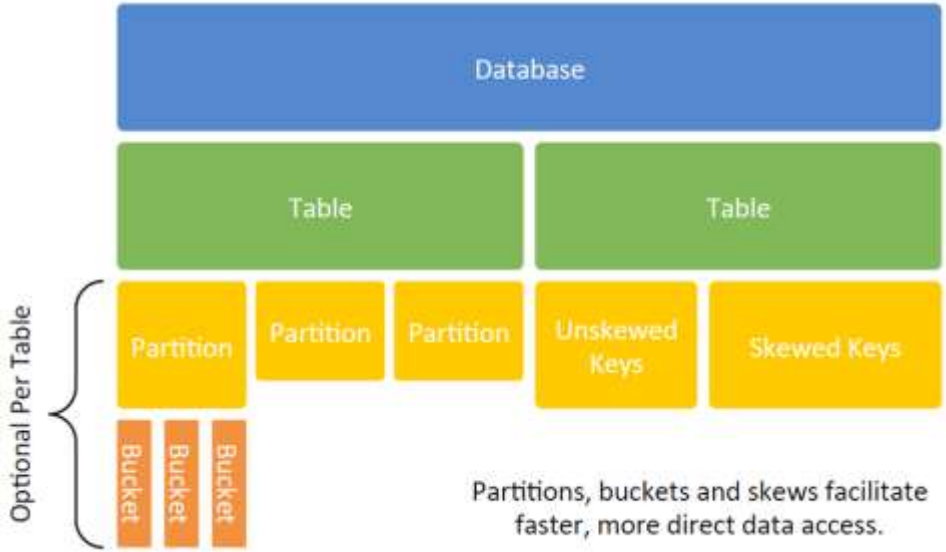
- CAUTION: Dropping a table removes its data in HDFS
- Tables are “managed” or “internal” by default
- Using EXTERNAL when creating the table avoids this behavior
- Dropping an external table removes only its metadata

```
CREATE EXTERNAL TABLE adclicks
  ( campaign_id STRING,
    click_time TIMESTAMP,
    keyword STRING,
    site STRING,
    placement STRING,
    was_clicked BOOLEAN,
    cost SMALLINT)
  LOCATION '/loudacre/ad_data';
```

Source: Cloudera

Hive is organised hierarchically into

- Databases
  namespaces that separate tables and other objects
- Tables
  homogeneous units of data with the same schema
- Partitions
  determine how the data is stored
  - Allow efficient access to subsets of the data
- Buckets/clusters
  - For subsampling within a partition
  - Join optimization



Hive Partitions

Use **partitioned by** clause to define a partition when creating table:

```
create table employees (id int, name string, salary double)
partitioned by (dept string);
```

Subfolders are created based on the partition values:

```
/apps/hive/warehouse/employees
        /dept=hr/
        /dept=support/
        /dept=engineering/
        /dept=training/
```

Can make some queries faster

Divide data based on partition column

Use PARTITION BY clause when creating table

Use PARTITION clause when loading data

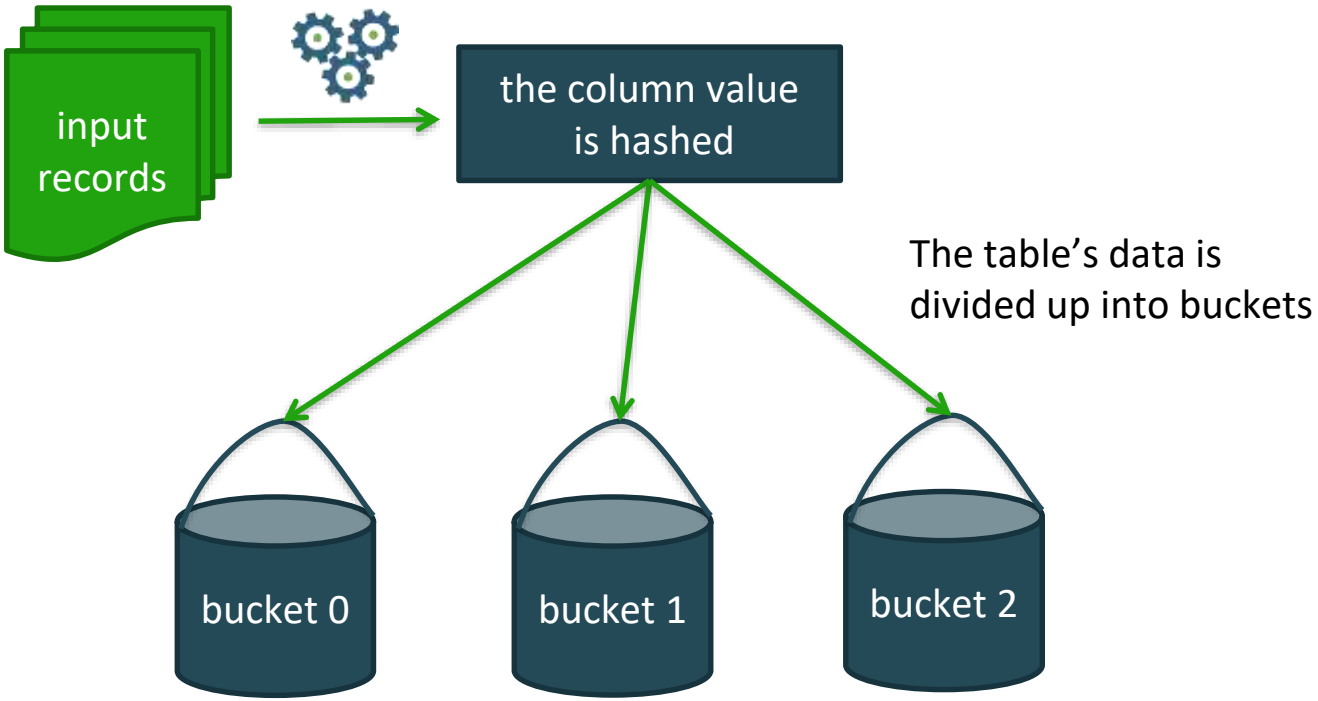
SHOW PARTITIONS will show a table's partitions

Reasons for organizing your tables (or partitions) into buckets.

1. More efficient queries.
   Bucketing imposes extra structure on the table, which Hive can take advantage of when performing certain queries.
   In particular, a join of two tables that are bucketed on the same columns - which include the join columns - can be efficiently implemented as a map-side join.

2. Make sampling more efficient.
   When working with large datasets, it is very convenient to try out queries on a fraction of your dataset while you are in the process of developing or refining them.

Hive Buckets



- Can speed up queries that involve sampling the data
  Sampling works without bucketing, but Hive has to scan the entire dataset
- Use CLUSTERED BY when creating table
  For sorted buckets, add SORTED BY
- To query a sample of your data, use TABLESAMPLE

Metastore is the central repository of metadata

- The metastore is the central repository of Hive metadata.
- The metastore is divided into two pieces:
  a service and the backing store for the data.
- By default, the metastore service runs in the same JVM as the Hive service and contains an embedded Derby database instance → embedded metastore configuration
  - one embedded Derby database can access the database files on disk at any one time
  - You can have only one Hive session open at a time that accesses the same metastore!
- The solution to supporting multiple sessions and users is to use a standalone database → local metastore
- Metastore service still runs in the same process as the Hive service but connects to a database running in a separate process

Hadoop Data Formats

Prof. Dr. Stephan Trahasch
Hochschule Offenburg

Avro

- Doug Cutting created Avro, a data serialization and RPC library, to help improve data interchange, interoperability, and versioning in Hadoop Eco System.
- Serialization & RPC Library and also storage format.

What led to a new serialization mechanism?

- Thrift is not splittable and codegen required.
- Dynamic reading is not possible.
- Sequence files does not have schema evolution.
- Evolved as in-house serialization and RPC library for Hadoop. Good overall performance.

Some Features of Avro

- Dynamic Access – No need of Code generation for accessing the data.
- Untagged Data – Which allows better compression
- Platform in-dependent – Has libraries in Java, Scala, Python, Ruby, C and C#. Compressible and splittable – Complements the parallel processing systems such as MR and Spark.
- Schema Evolution:
  "Data models evolve over time", and it's important that your data formats support your need to modify your data models. Schema evolution allows you to add, modify, and in some cases delete attributes, while at the same time providing backward and forward compatibility for readers and writers

Some Features of Avro

- Row Based
- Direct mapping from/to JSON
- Best compatibility for evolving data schemas
- Provides rich data structures
- Map keys can only be strings (could be seen as a limitation)
- Compact binary form
- Extensible schema language
- Untagged data
- Bindings for a wide variety of programming languages
- Dynamic typing
- Provides a remote procedure call
- Supports block compression
- Avro files are splittable

PARQUET - Introduction

- Columnar storage format that come out of a collaboration between Twitter and Cloudera based on Dremel
- To have a state of the art columnar storage available across the Hadoop platform
  - Hadoop is very reliable for big long running queries but also IO heavy.
  - Incrementally take advantage of column based storage in existing framework.
  - Not tied to any framework in particular
- Columnar Storage
  - Limits IO to data actually needed:
    loads only the columns that need to be accessed.
  - Saves space:
    Columnar layout compresses better
  - Type specific encodings.

A	B	C
A1	B1	C1
A2	B2	C2
A3	B3	C3

A1	A2	A3	B1	B2	B3	C1	C2	C3

PARQUET - Introduction

- Allows compression. Currently supports Snappy and Gzip.
- Well supported over Hadoop eco system.
- Very well integrated with Spark SQL and DataFrames.
- Predicate pushdown:
  Projection and predicate pushdowns involve an execution engine pushing the projection and predicates down to the storage format to optimize the operations at the lowest level possible.
- I/O to a minimum by reading from a disk only the data required for the query.
- Language independent. Supports Scala, Java, C++, Python.

Apache ORC – Optimized Row-Columnar File

- Columnar format
- Enables user to read & decompress just the bytes they need
- Fast
- Indexed
- Self-describing
  - Includes all of the information about types and encoding
- Rich type system
  - All of Hive's types including timestamp, struct, map, list, and union

File Structure

- File contains a list of stripes, which are sets of rows
  - Default size is 256MB
  - Large stripe size enables efficient reads
- Footer
  - Contains the list of stripe locations
  - Type description
  - File and stripe statistics
- Postscript
  - Compression parameters
  - File format version

Predicate Push Down

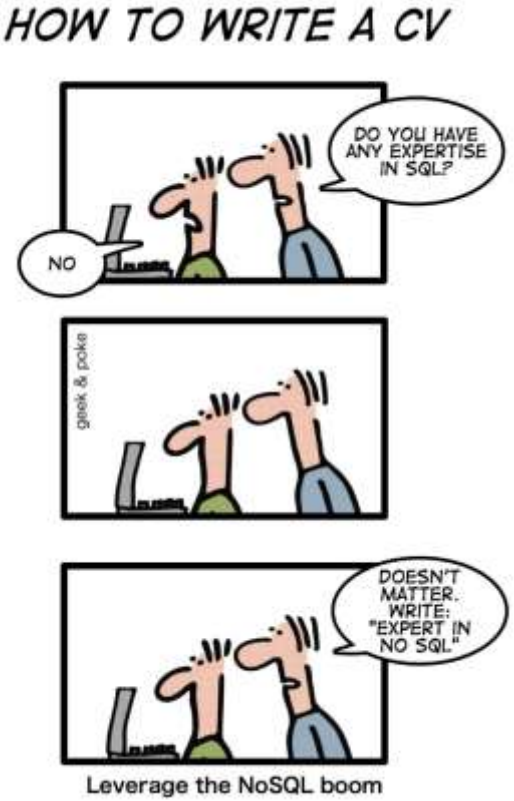
- Reader is given a SearchArgument
  - Limited set predicates over column and literal value
  - Reader will skip over any parts of file that can’t contain valid rows

- ORC indexes at three levels:
  - File
  - Stripe
  - Row Group (10k rows)

- Reader still needs to apply predicate to filter out single rows

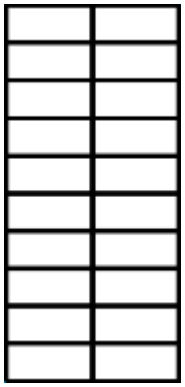
NoSQL
Apache HBase

Prof. Dr. Stephan Trahasch
Hochschule Offenburg

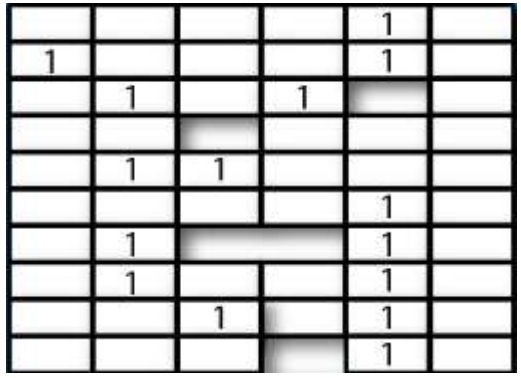


(Major) Types of NoSQL databases

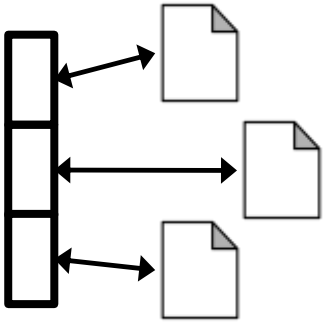
Key Value



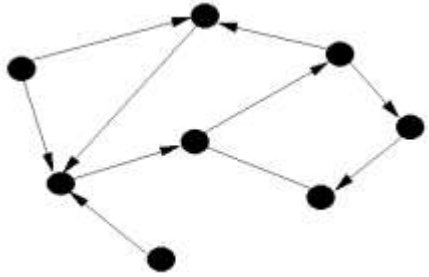
Column Family



Document



Graph



CAP Theorem (also known as Brewer's theorem)

CAP theorem states that in a distributed database you can only have two of the following properties:

1. **Consistency** equivalent to having a single up-to-date copy of the data
   All requests at the same time retrieve the same value.
2. High **Availability** of that data
   The retrieval of data is always possible as long as at least one server is running.
3. Tolerance to Network **Partitions**
   The system will function even if the communication is broken.

CAP theorem stated at the Symposium on Principles of Distributed Computing (PODC) by Eric Brewer in 2000

Formal proof by Seth Gilbert, Nancy Lynch in 2002

Apache HBase

- Non-relational, distributed, column-based database
- Table consists of rows, column families and columns
- Stored as key value pairs
  - Key, CF, CQ and value
- Distribution to so-called regions
  - Partitions to which data is assigned
- Storage on distributed file system HDFS or other fs

Row Key | Column Family

Row Key	BaseData		TestData			
ProcessID	Machine	Product	Temperature	Voltage	Accuracy	Size
1	M1	Thermometer	23.1	3.3	0.1	
2	M1	LightSensor		5.1		0.4
3	M2	Microchip		1.7		0.05
4	M3	Service				

Column Qualifiers

Cell

Data Model

- A table in HBase is a sparse, distributed, persistent multidimensional **sorted map**
- Map indexed by a row key, column key, and a timestamp (Table, RowKey, Family, Column, Timestamp) → Value
- Supports lookups, inserts, deletes
  - Single row transactions only

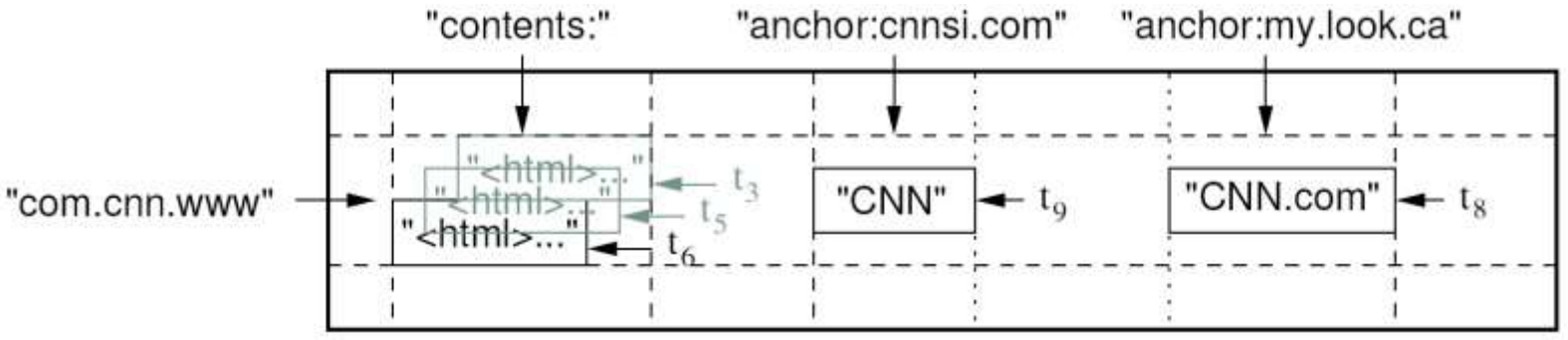


Image Source: Chang et al., OSDI 2006

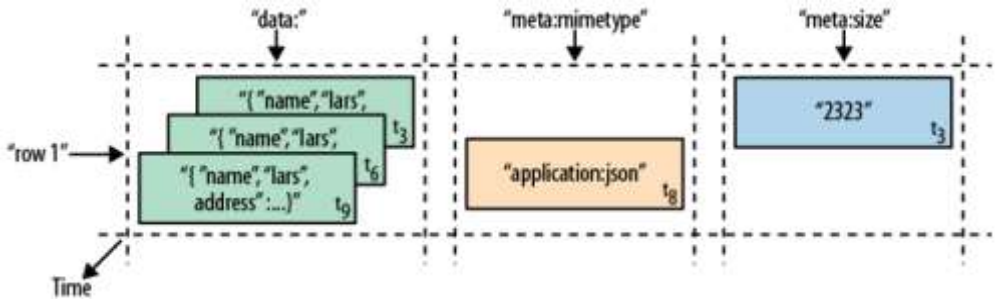
Rows and Columns

- Rows are composed of columns, which are grouped into column families
- Column families group semantically related values
  - Column key = `family:qualifier`
  - Each column family is stored as one file (HFile) in HDFS
  - One column family can have millions of columns
  - Hint: not more than 3 families
- Rows maintained in sorted lexicographic order
  - Applications can exploit this property for efficient row scans
  - Row ranges dynamically partitioned into tablets

At the end of the day, it's all key-value pairs!

Timestamp

- Versioning: used to store different versions of data in a cell
  - 64-bits integer (UNIX timestamp)
  - New writes default to current time, but timestamps for writes can also be set explicitly by clients
- Lookup options
  - “Return most recent K values”
  - “Return all values in timestamp range (or all values)”
- Garbage collection:
  - “Only retain most recent K values in a cell”
  - “Keep values until they are older than K seconds”



Column Families

Columns are managed in isolation. Different applications access only a subset of the columns.

Advantage of vertical partitioning: unbound attributes of an entity do not need to be explicitly saved

Konzeptionell

Row Key	Time Stamp	Column "contents:"	Column "anchor:"		Column "mime:"
"com.cnn.www"	t9	"<html>abc..."	"anchor: cnnsi.com"	"CNN"	
	t8	"<html>def..."	"anchor: my.look.ca"	"CNN.com"	
	t6	"<html>ghi..."			"text/html"

Intern

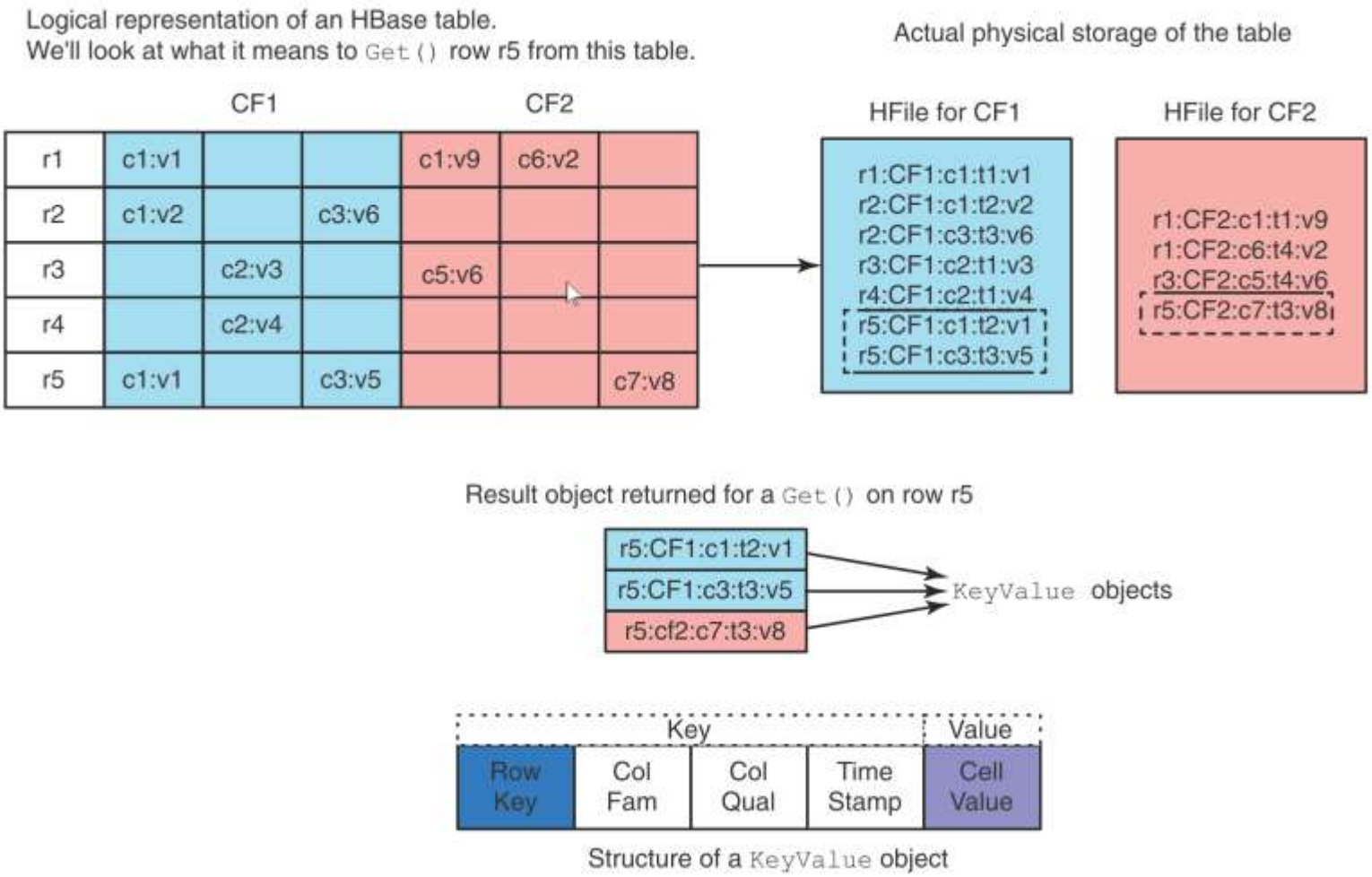
Row Key	Time Stamp	Column "contents:"
"com.cnn.www"	t9	"<html>abc..."
	t8	"<html>def..."
	t6	"<html>ghi..."

Row Key	Time Stamp	Column "anchor:"
"com.cnn.www"	t9	"anchor: cnnsi.com"
	t8	"anchor: my.look.ca"

...

Partitions are then saved individually, because multiple Column Families are not accessed at the same time (definition!) in one request.

Logical to physical translation of an HBase table

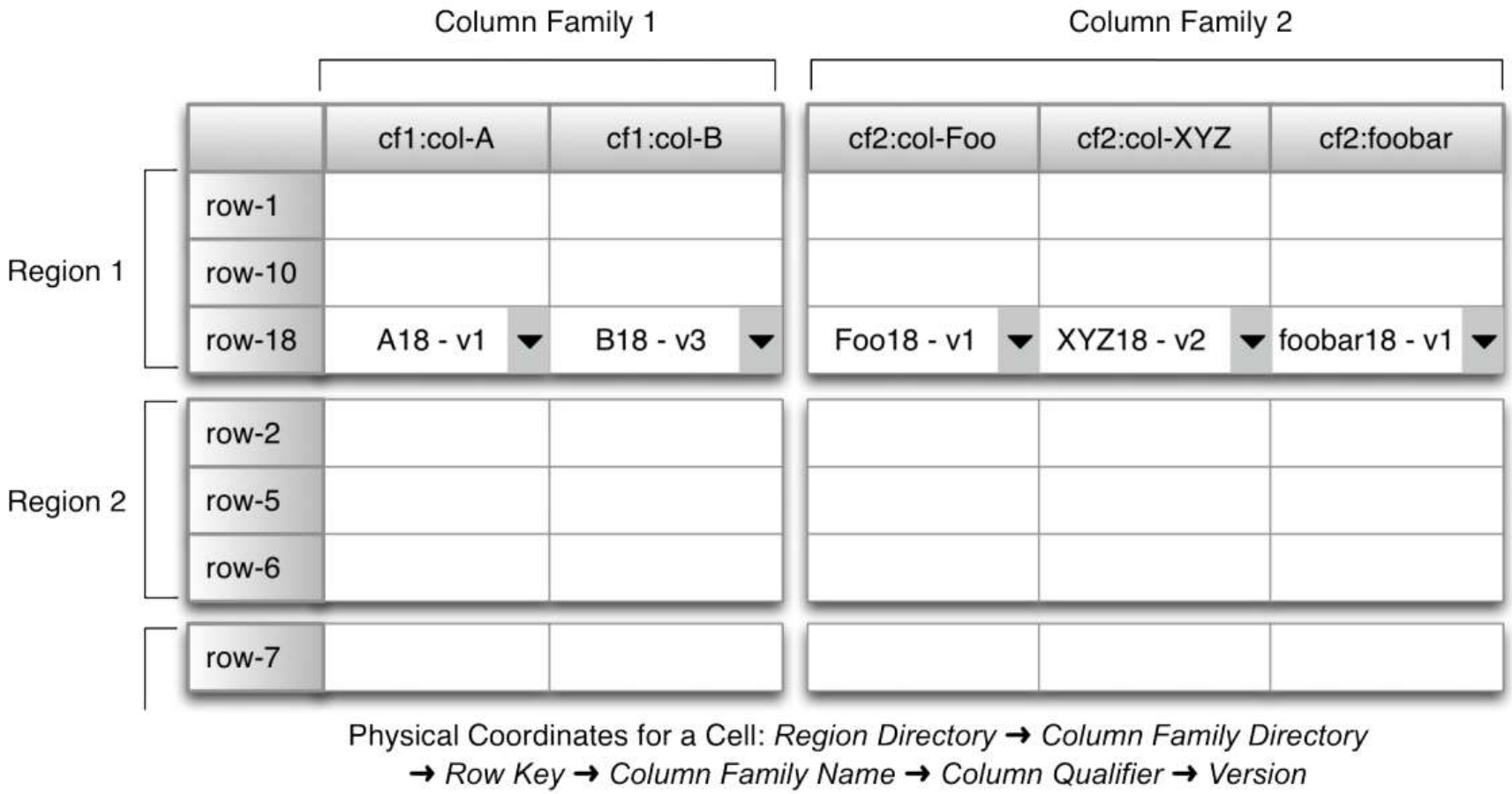


Dimiduk, N., & Khurana, A. (2013). HBase in action. Shelter Island, NY: Manning.

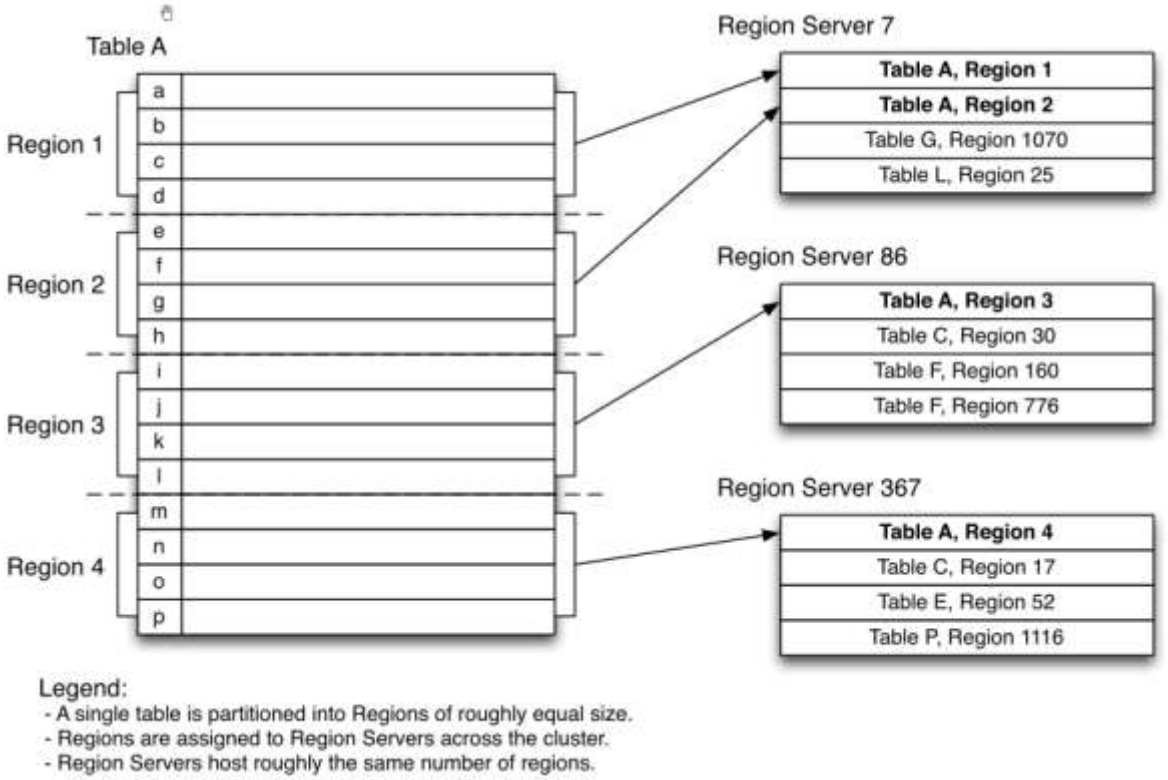
HBase Regions

For scalability, tables are split into regions by key.

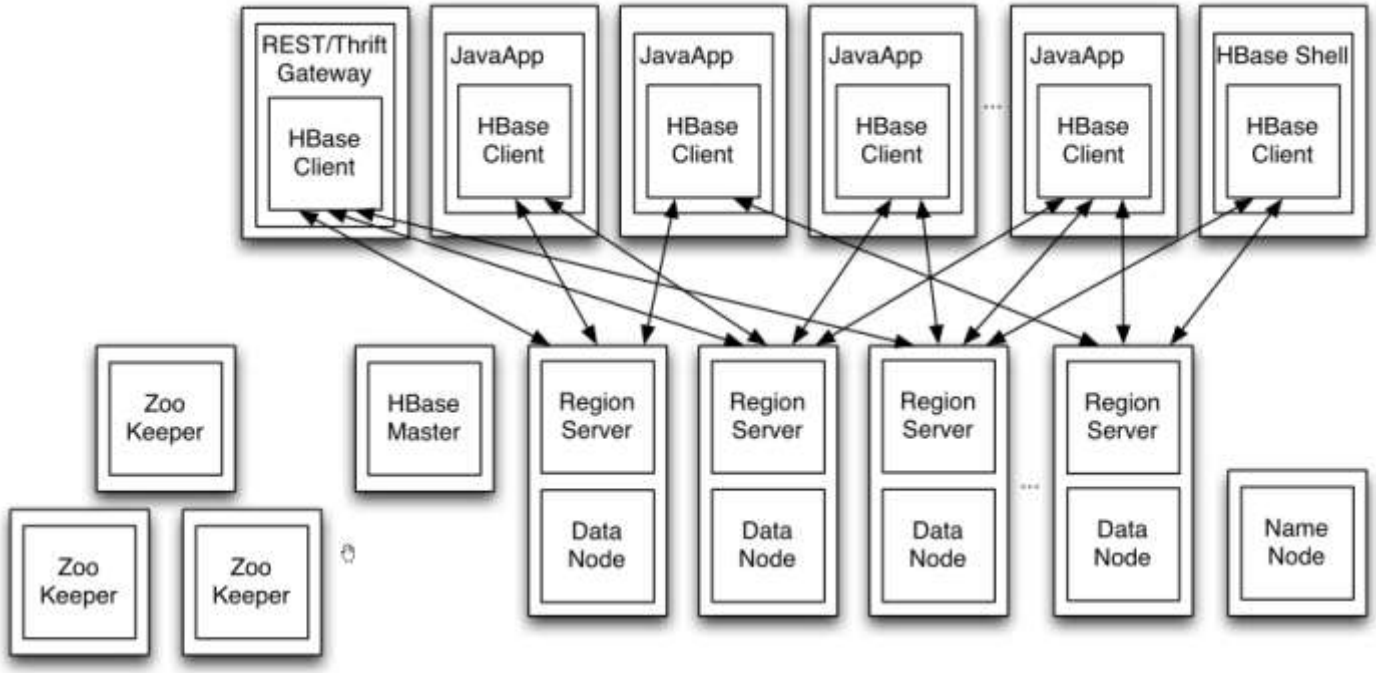
Each region is assigned to a region server.



Logical Architecture



Physical Architecture



Legend:
- An HBase RegionServer is collocated with an HDFS DataNode.
- HBase clients communicate directly with Region Servers for sending and receiving data.
- HMaster manages Region assignment and handles DDL operations.
- Online configuration state is maintained in ZooKeeper.
- HMaster and ZooKeeper are NOT involved in data path.

Architecture Components and Responsibilities

- **Master**
  - Monitor RegionServer
  - Runs LoadBalancer to transfer regions between servers
  - Check and clean the meta table
  - Typically runs on HDFS NameNode
- **RegionServer**
  - Hosts a subsequent span of keys (Region) for tables
  - Executes Client Request Filters
  - Runs periodic compaction
  - Typically runs on HDFS DataNode
  - Memstore: Accumulates all writes
  - If filled, data is flushed to new store files
  - Multiple smaller files can be compacted into fewer
  - After flushes/compaction the region may be split
- **Client**
  - Identify location of HBase:meta from ZooKeeper
  - Query HBase:meta for identifying the RegionServers
  - May use Client Request Filters

How many Column Families?

- Rule of thumb: not more than 3
- All data for a given column family goes into a single store on HDFS. A store may consist of multiple HFiles, but ideally, on compaction, you achieve a single HFile.
- Columns in a column family are all stored together on disk, and that property can be used to isolate columns with different access patterns by putting them in different column families. This is also why HBase is called a column-family-oriented store.

Outline

- Introduction
- HBase
- Table Design
- Bloom Filter
- Use Case

Approximate set membership problem

Trade-off between the space and the false positive probability.

Generalize the hashing idea.

Suppose we have a set $S = \{s_1, s_2, \ldots, s_m\} \subseteq$ universe U

Represent S in such a way we can quickly answer

"Is x an element of S ?"

To take as little space as possible,
we allow false positive (i.e. $x \notin S$, but we answer yes)

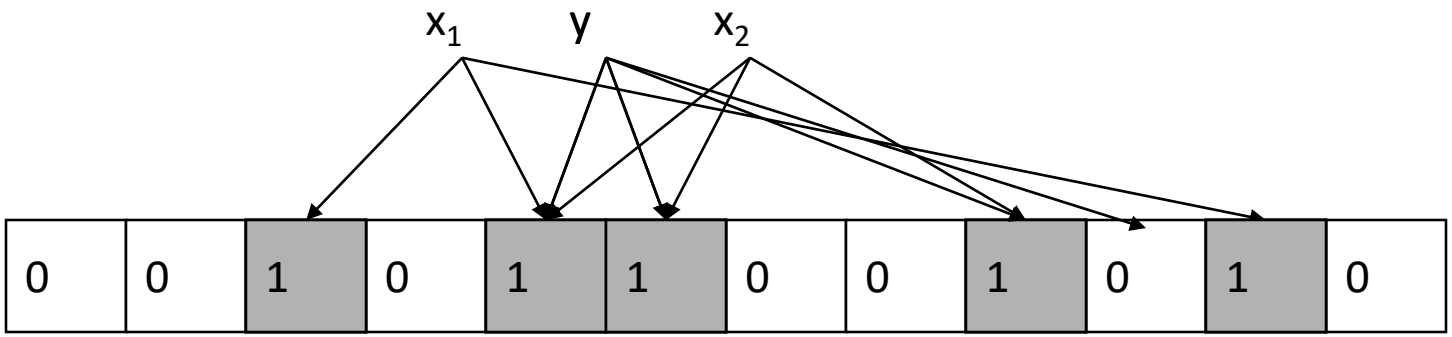
If $x \in S$, we must answer yes .

Bloom filter

Consist of an arrays A[n] of n bits (space), and k independent random hash functions

$h_1,\ldots,h_k : U \dashrightarrow \{0, 1, \ldots, n-1\}$

1. Initially set the array to 0
2. $\forall s \in S$, $A[h_i(s)] = 1$ for $1 \leq i \leq k$
   (an entry can be set to 1 multiple times, only the first times has an effect)
3. To check if $x \in S$, we check whether all location $A[h_i(x)]$ for $1 \leq i \leq k$ are set to 1
   a. If not, clearly $x \notin S$.
   b. If all $A[h_i(x)]$ are set to 1 ,we assume $x \in S$



If only 1s appear, conclude that y is in S
This may yield false positive

Bloom filter

- A Bloom filter is like a hash table, and simply uses one bit to keep track whether an item hashed to the location.
- If k=1 , it's equivalent to a hashing based fingerprint system.
- If n=cm for small constant c, such as c=8 , then k=5 or 6, the false positive probability is just over 2% .
- It's interesting that when k is optimal

  k=ln(2)*(n/m) , then p= 1/2.

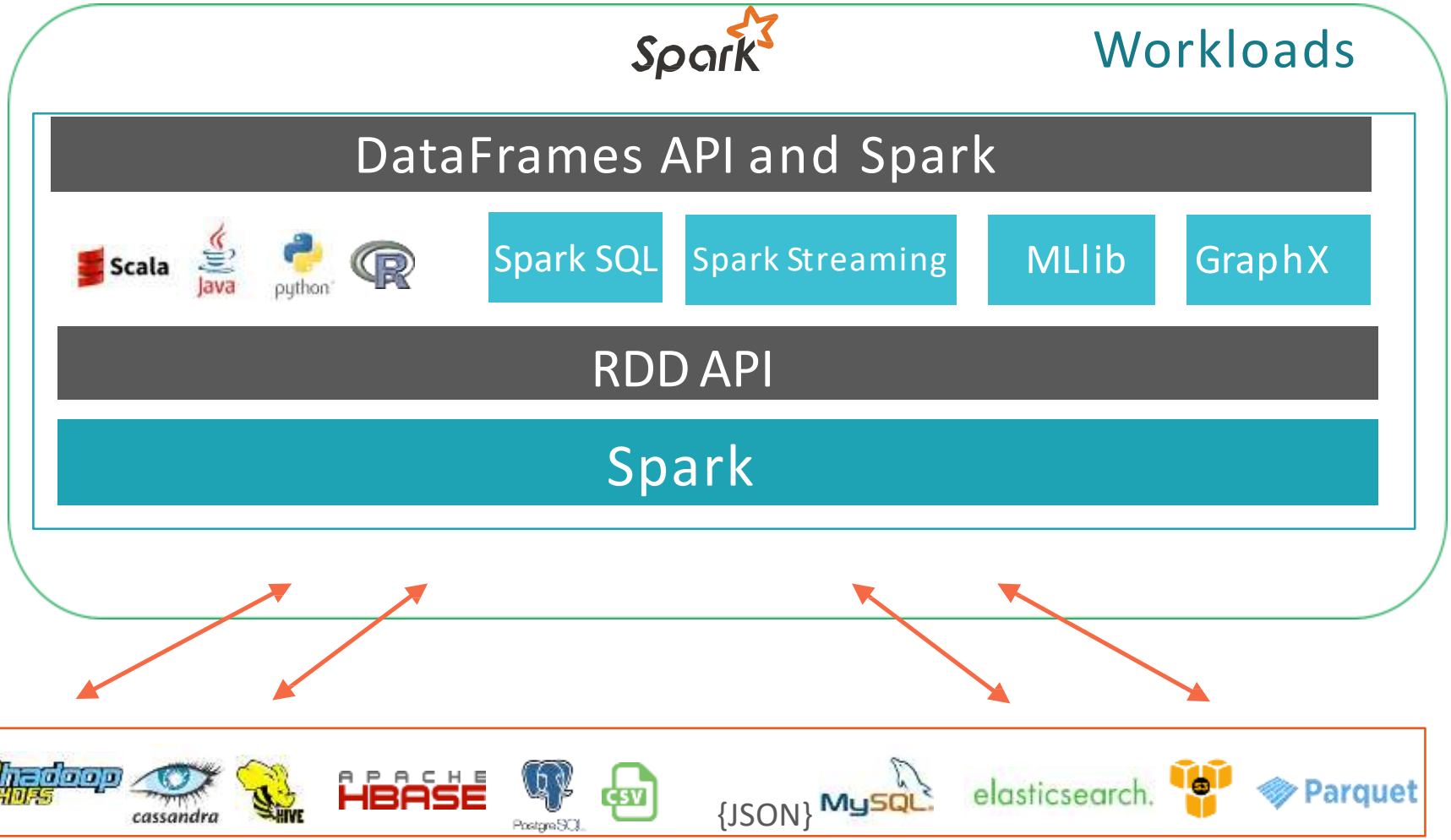
  An optimized Bloom filter looks like a random bit-string

Apache Spark

http://spark.apache.org/

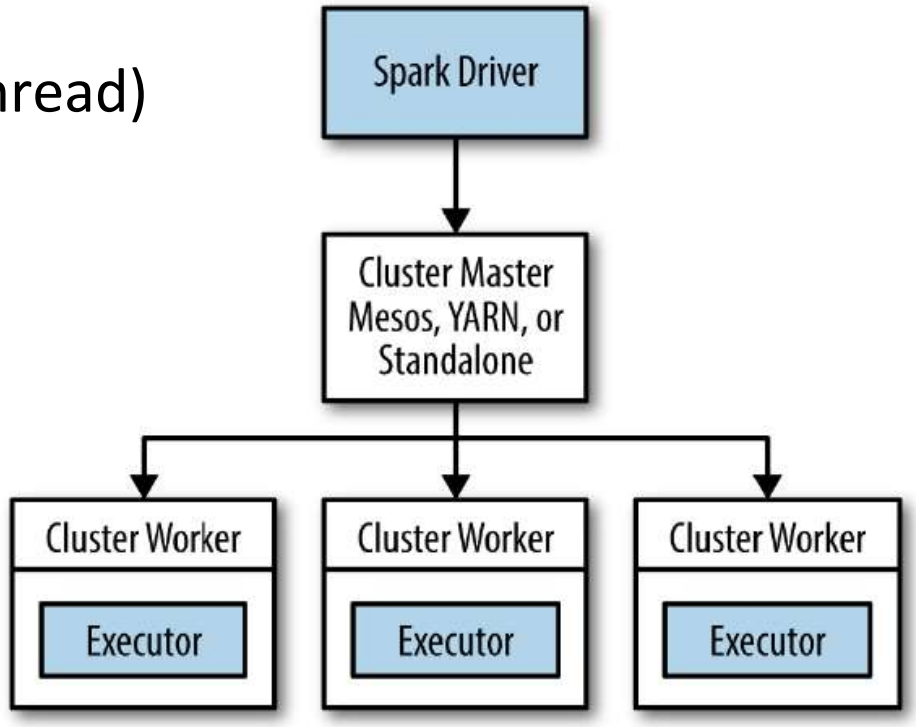
Prof. Dr. Stephan Trahasch

Apache Spark Engine

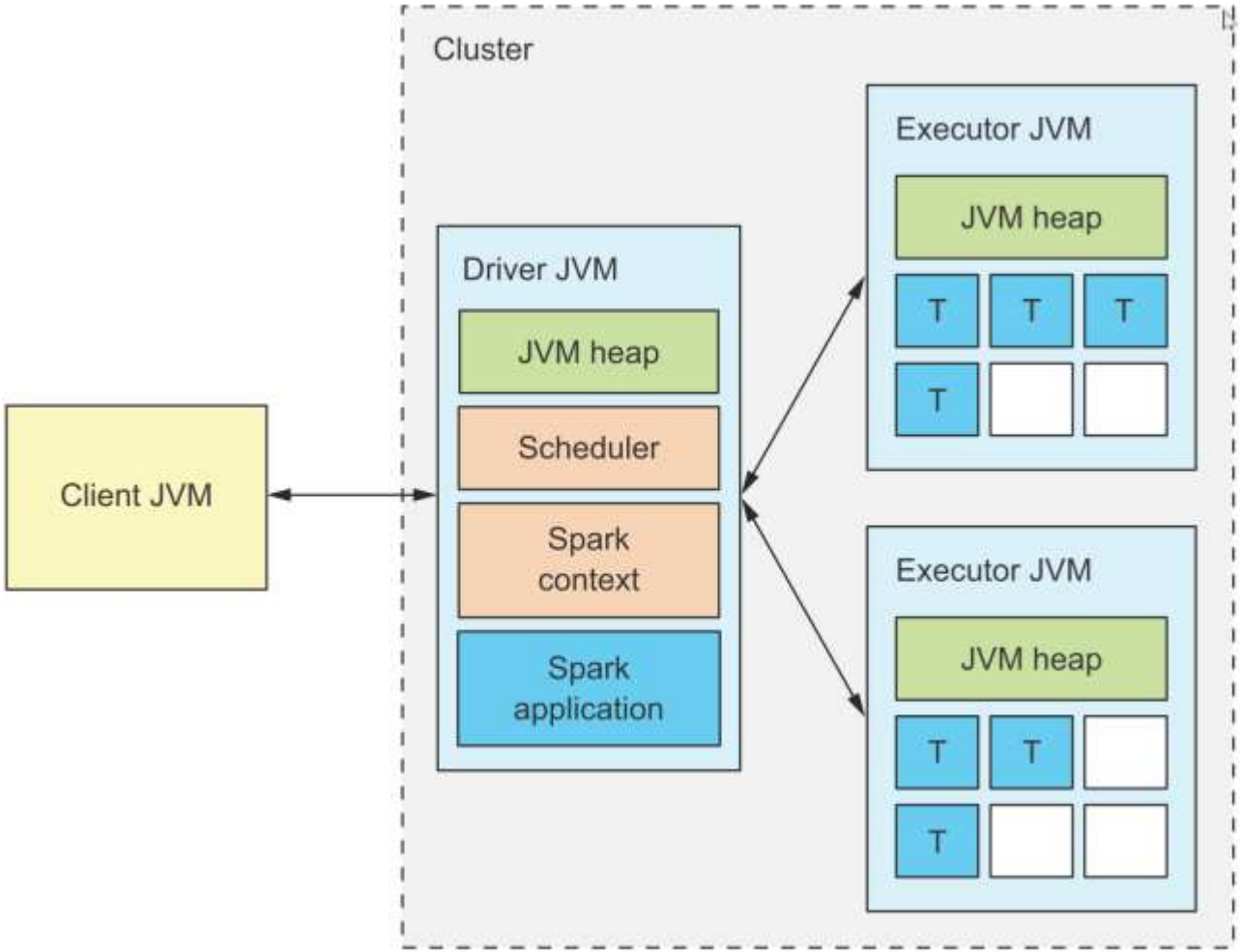


Components of a Spark cluster

- Master / Slave architecture
- Master: coordination (Driver Thread)
  - Executes programs
  - *SparkSession / SparkContext* as entrypoint
  - Definition of RDDs and actions
- Executor
- Ressource coordination with cluster manager



Components running in a cluster: client, driver, executors



Source: Spark in Action

Spark Cluster Types

- Standalone Cluster
  - Provides faster job startup than those jobs running on YARN.
- YARN Cluster
  - YARN lets you run different types of Java applications, so you can mix legacy Hadoop and Spark applications.
  - YARN provides methods for isolating and prioritizing applications among users and organizations.
  - It's the only cluster type that supports Kerberos-secured HDFS.
- MESOS Cluster
  - Mesos is a scalable and fault-tolerant distributed systems kernel
  - Mesos is a "scheduler of scheduler frameworks"
  - provides scheduling of other types of resources (CPU, disk space ..)
- Spark Local Mode
  - Special case of a Spark standalone cluster running on a single machine

Core Concept: Resilient Distributed Dataset – RDD

Resilient
if data in memory is lost, it can be recreated

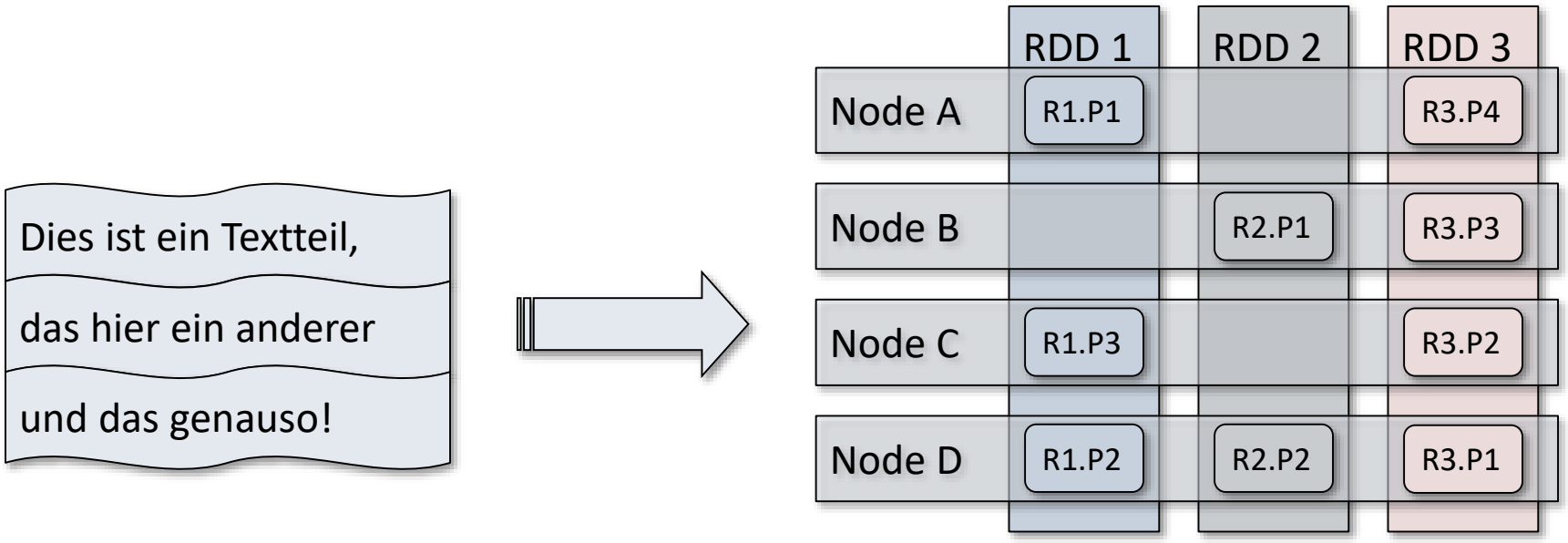
Distributed
processed across the cluster

Dataset
initial data can come from a Data Source or be created programmatically

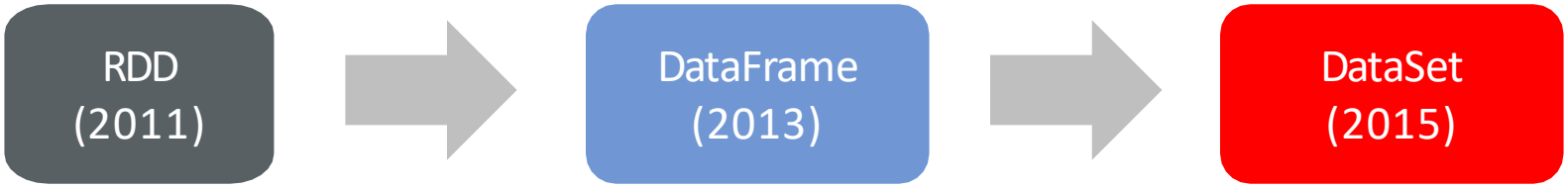
- RDDs are the fundamental unit of data in Spark
- RDDs are immutable
  - Data in an RDD is never changed
  - Transform in sequence to modify the data as needed
- Spark programming consists of performing operations on RDDs

Resilient Distributed Datasets

- Fault-tolerant collection of elements partitioned across the nodes of the cluster that can be operated on in parallel.
- Immutable



History of Spark APIs

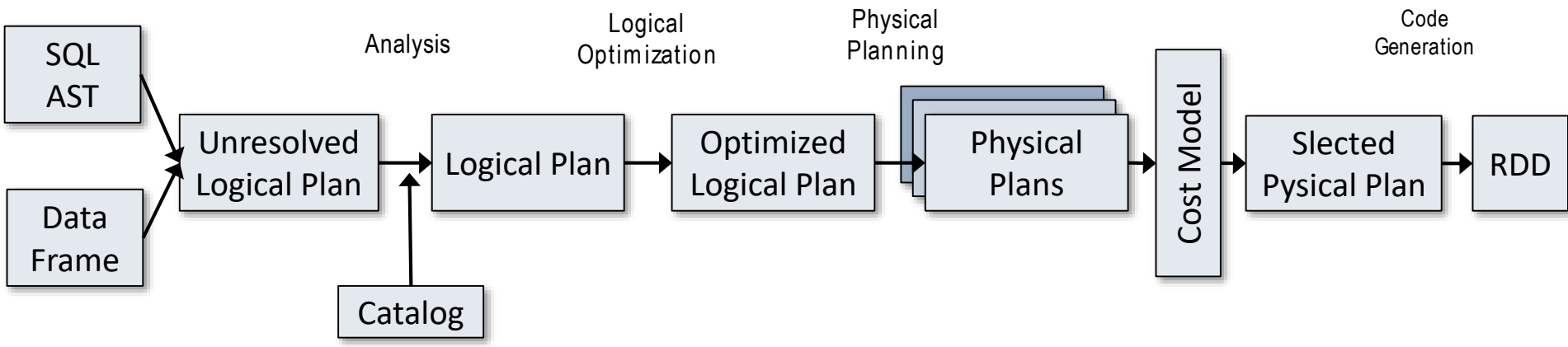


- Distribute collection of JVM objects
- Functional Operators (map, filter, etc.)

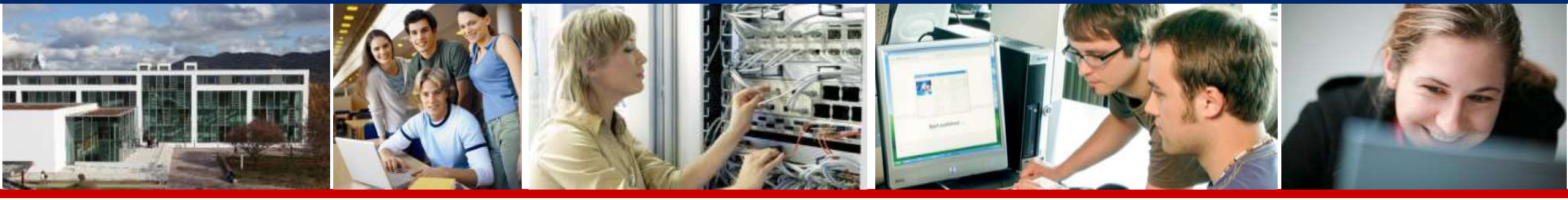
- Distribute collection of Row objects
- Expression-based operations and UDFs
- Logical plans and optimizer
- Fast/efficient internal representations

- Internally rows, externally JVM objects
- "Best of both worlds" type safe + fast

Plan Optimization & Execution



DataFrames and SQL share the same optimization/execution pipeline.



Parallel Processing in Spark

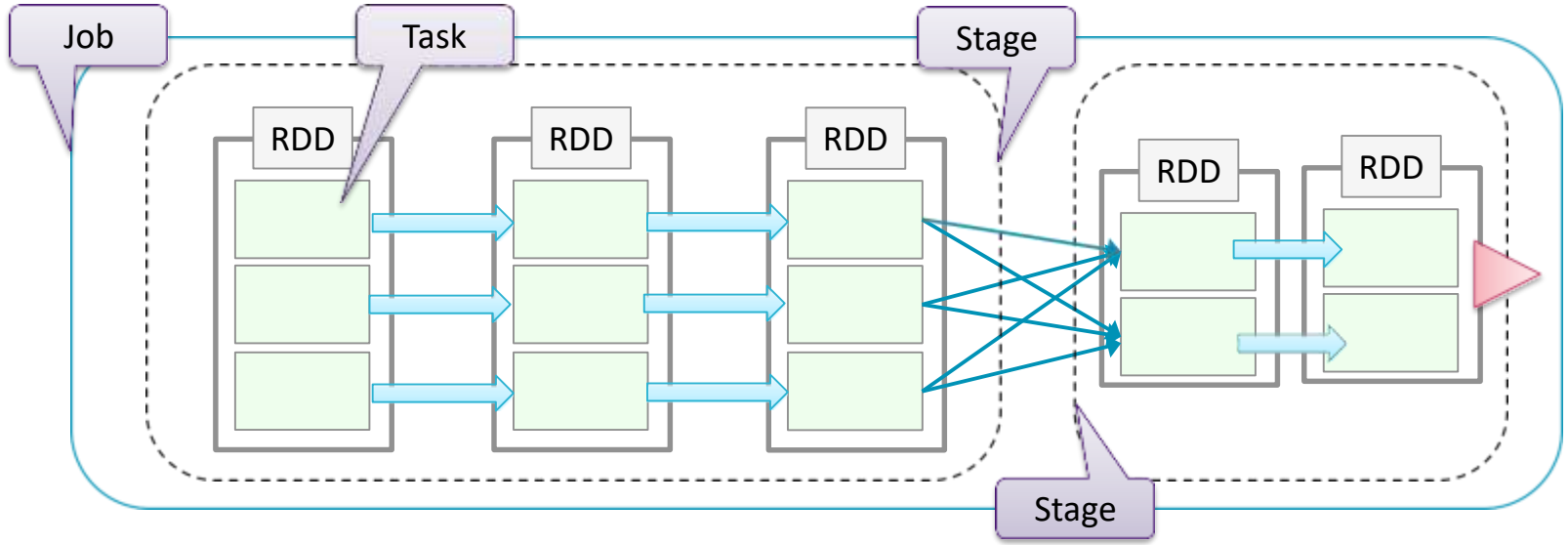
Based on Slides of Cloudera. Copyrigth Cloudera.

Stages

- Operations that can run on the same partion are executed in stages
- Tasks within a stage are pipelined together
- Developers should be aware of stages to improve performance

Summary of Spark Terminology

- Job – a set of tasks executed as a result of an action
- Stage – a set of tasks in a job that can be executed in parallel
- Task – an individual unit of work sent to one executor
- Application – can contain any number of jobs managed by a single driver



How Spark Calculates Stages

Spark constructs a DAG (Directed Acyclic Graph) of RDD dependencies

Narrow dependencies

- Only one child depends on the RDD
- No shuffe required between nodes
- Can be collapsed into a single stage
  e.g., map, filter, union

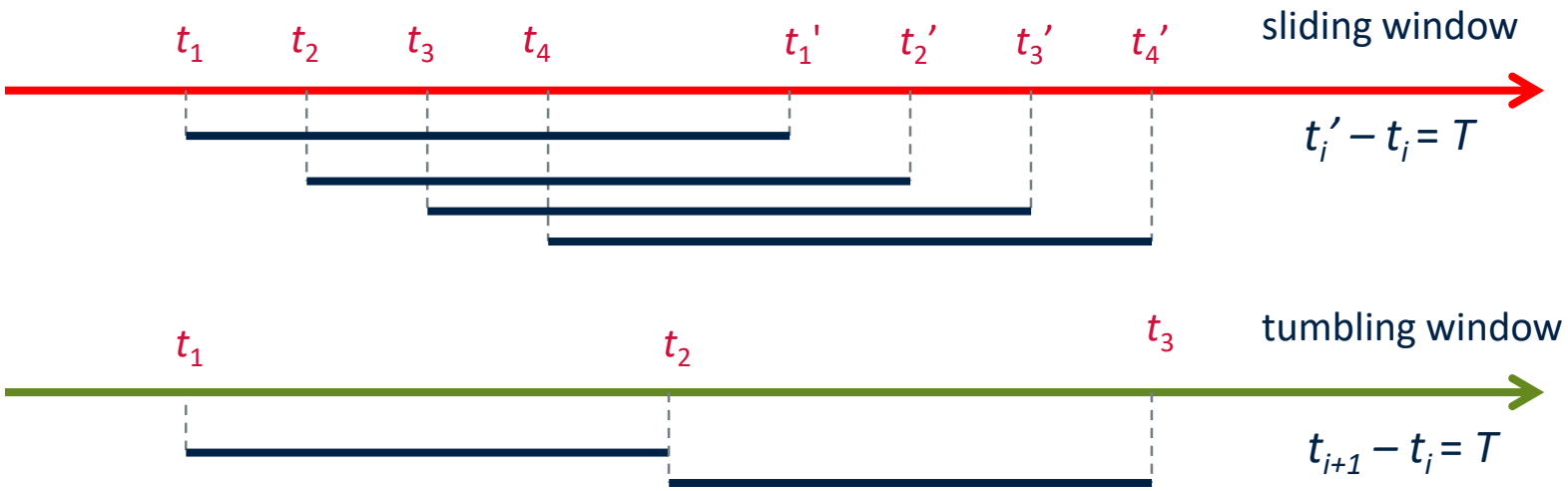
- **Wide (or shuffle) dependencies**
- Multiple children depend on the RDD
- Defines a new stage
  e.g., reduceByKey, join, groupByKey

Spark Streaming

Prof. Dr. Stephan Trahasch
Hochschule Offenburg

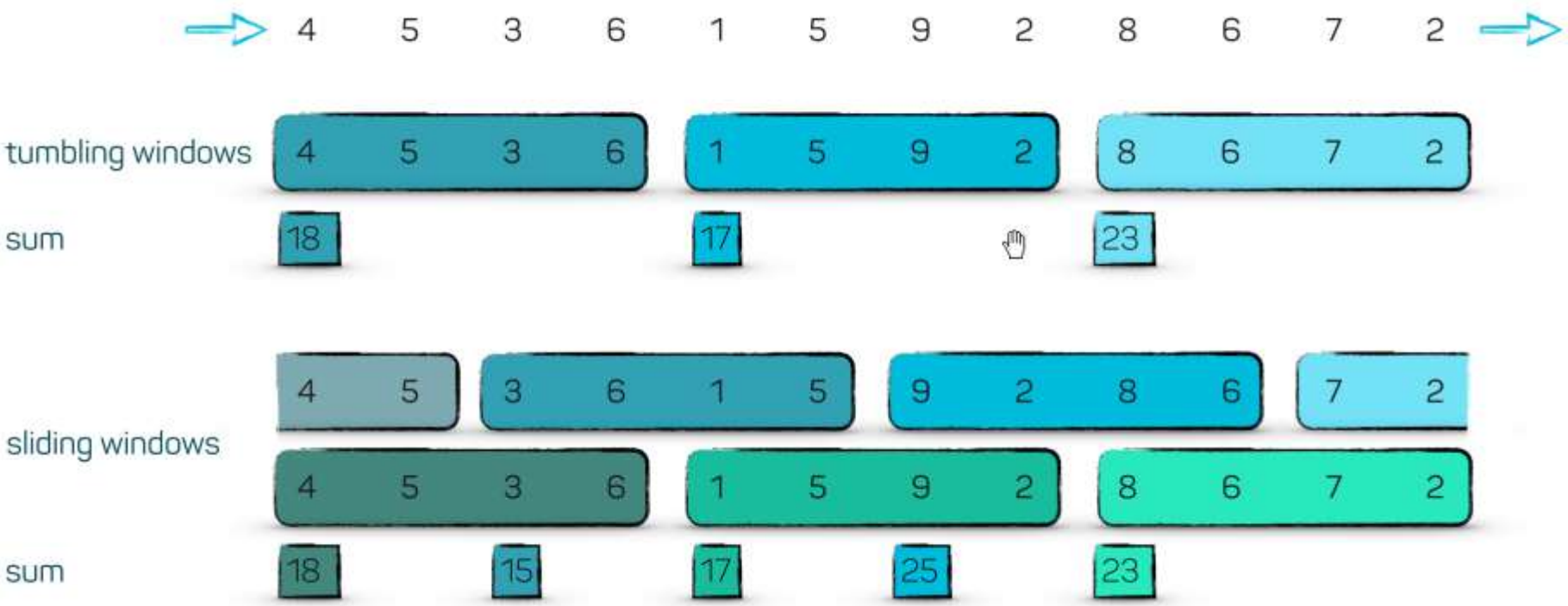
Windows on Ordering Attributes

- Assumes the existence of an attribute that defines the order of stream elements (e.g., time)
- Let *T* be the window size in units of the ordering attribute



Source: Peter Bonz

Tumbling & Sliding Windows



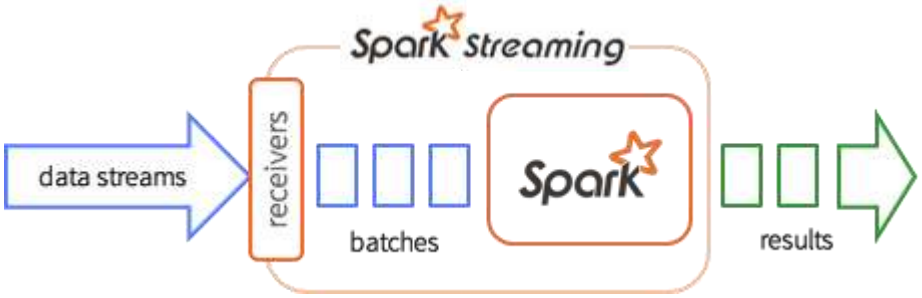
Windows on Counts

- Window of size *N* elements (sliding, tumbling) over the stream
- Challenges:
  - Problematic with non-unique timestamps: non-deterministic output
  - Unpredictable window size (and storage requirements)

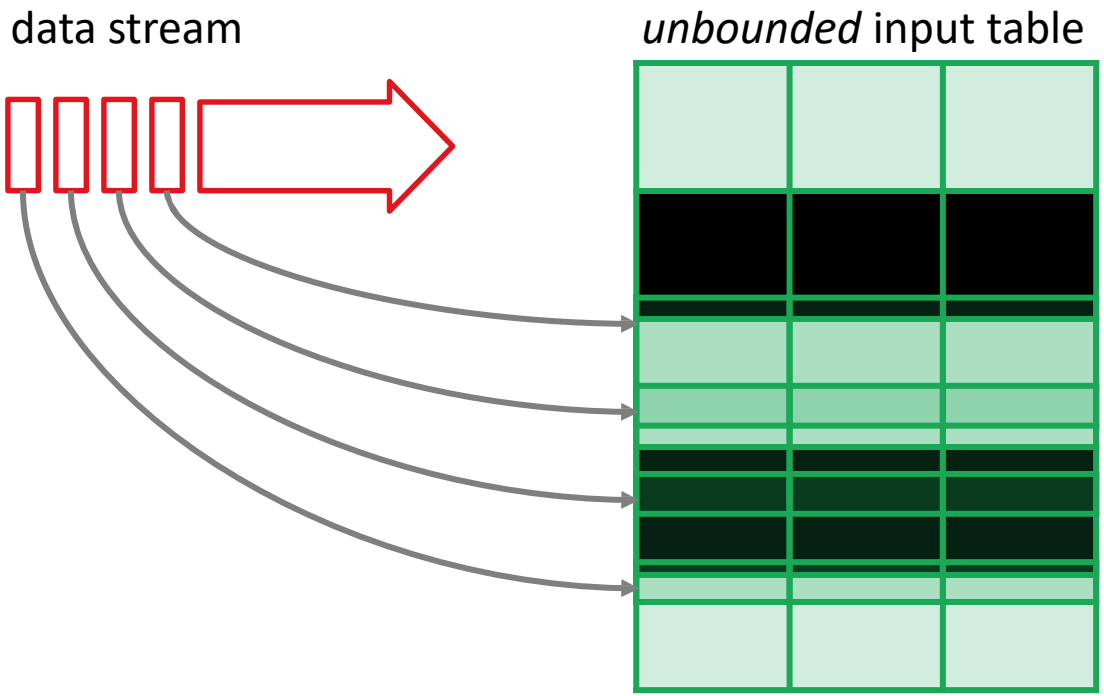
Source: Peter Bonz

Apache Spark Streaming

- Microbatching
- Similiar, partly unified, programming model as with batch processing
- State and window operations
- Missing support for event time
- DStream and Structured Streaming



Treat Streams as Unbounded Tables



new data in the
data stream

=

new rows appended
to a unbounded table