Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

Distributed Data Management
Batch Processing

Thorsten Papenbrock
F-2.04, Campus II
Hasso Plattner Institut

Distributed Data Management

What we learned so far!

Foundations

- What do Moore’s and Amdahl’s Laws tell us about distributed systems?

Encoding and Communication

- How do we (de)serialize and send messages in distributed systems?

Data Models and Query Languages

- How to conceptually design and practically query data in distributed systems?

Storage and Retrieval

- How to physically structure data in distributed systems?

Replication and Partitioning

- How to introduce redundancy and grouping in distributed systems?

Distributed Systems

- How to deal with network and time unreliability in distributed systems?

Consistency and Consensus

- How to enforce ACID and CAP guarantees in distributed systems?

What we learned so far!

Akka

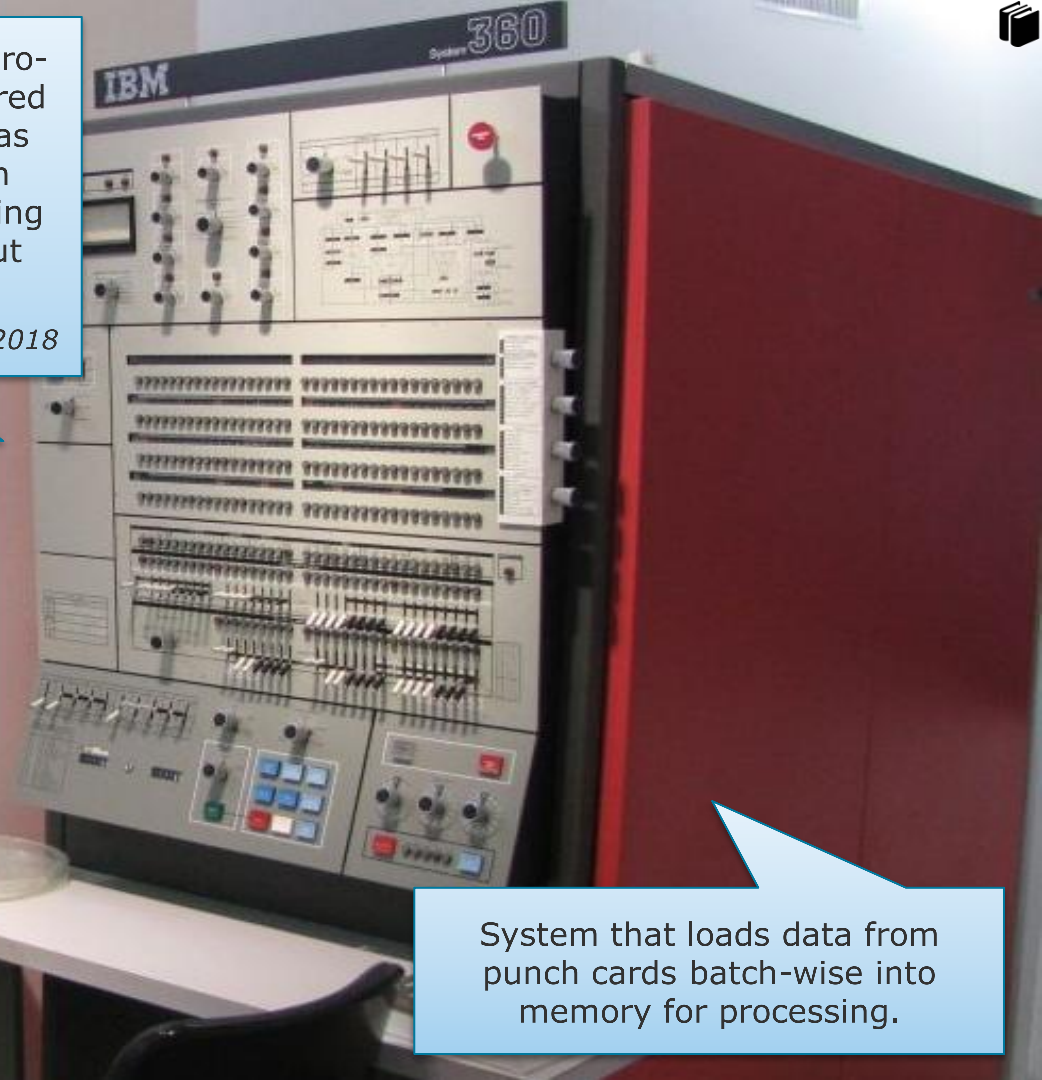
- A toolkit for building distributed systems
- Implements several tools that support the build process
  - Failure detectors, persistence functionality, reliability features, API connectors, …
- Allows us to implement any protocol, application or system that we can imagine.
- Does very little to protect us against programming errors, unstable systems, non-scalable algorithms, inperformant protocols, data loss, …
  - ➢ **Akka is a powerful toolkit but no programming framework!**

Now: Frameworks for building distributed systems

- Observation:
  - Certain data- and task-parallel tasks repeat frequently in distributed systems.
- Idea:
  - Build a framework that solves these tasks efficiently and effectively.

"Computerized batch processing, since the 1964 introduction of the IBM System/360, has primarily referred to the scripted running of one or more programs, as directed by Job Control Language, with no human interaction other than, if JCL-requested, the mounting of one or more pre-determined input and/or output computer tapes."

Wikipedia, 2018



System that loads data from punch cards batch-wise into memory for processing.

Batch Processing

HPI Hasso Plattner Institut

Properties:

- Large (data-intensive) jobs are (automatically) broken down into sequences of smaller tasks (e.g. reading, writing, filtering, sorting, mapping, joining data).
- Tasks are well-understood, recurring, (side-effect free), standard activities.
- No (or only minimal) human interaction is involved.
- Processing is compute intensive (and offline) in nature.

Batch Processing

Why batch processing (frameworks) in distributed computing?

(or why database reads, service calls, and message sends are insufficient)

- Components that we keep re-implementing
  - Mechanisms for reliable data transfer
  - Channels for large data transmission
  - Techniques to ensure fault tolerance
  - Functions for data extraction, transformation, and loading
- Most data-intensive jobs follow the same pattern
  1. Extract data from some (distributed) source.
  2. Transform data in a certain way (utilizing all available resources).
  3. Load result into some (distributed) sink.
  - ➢ All three steps build upon recurring standard steps (i.e., functions/programs).
  - ➢ Much resemblance to what batch processes do.

Types of Systems

HPI Hasso Plattner Institut

Services (online systems)

- Accept requests and send responses
- Performance measure: response time and availability
- Expected runtime: milliseconds to seconds

OLTP

Batch processing systems (offline systems)

- Take (large amounts of) data; run (complex) jobs; produce some output
- Performance measure: throughput (i.e., data per time)
- Expected runtime: minutes to days

now

Stream processing systems (near-real-time systems)

- Consume volatile inputs; operate stream jobs; produce some output
- Performance measure: throughput and precision
- Expected runtime: near-real-time (i.e., as data arrives)

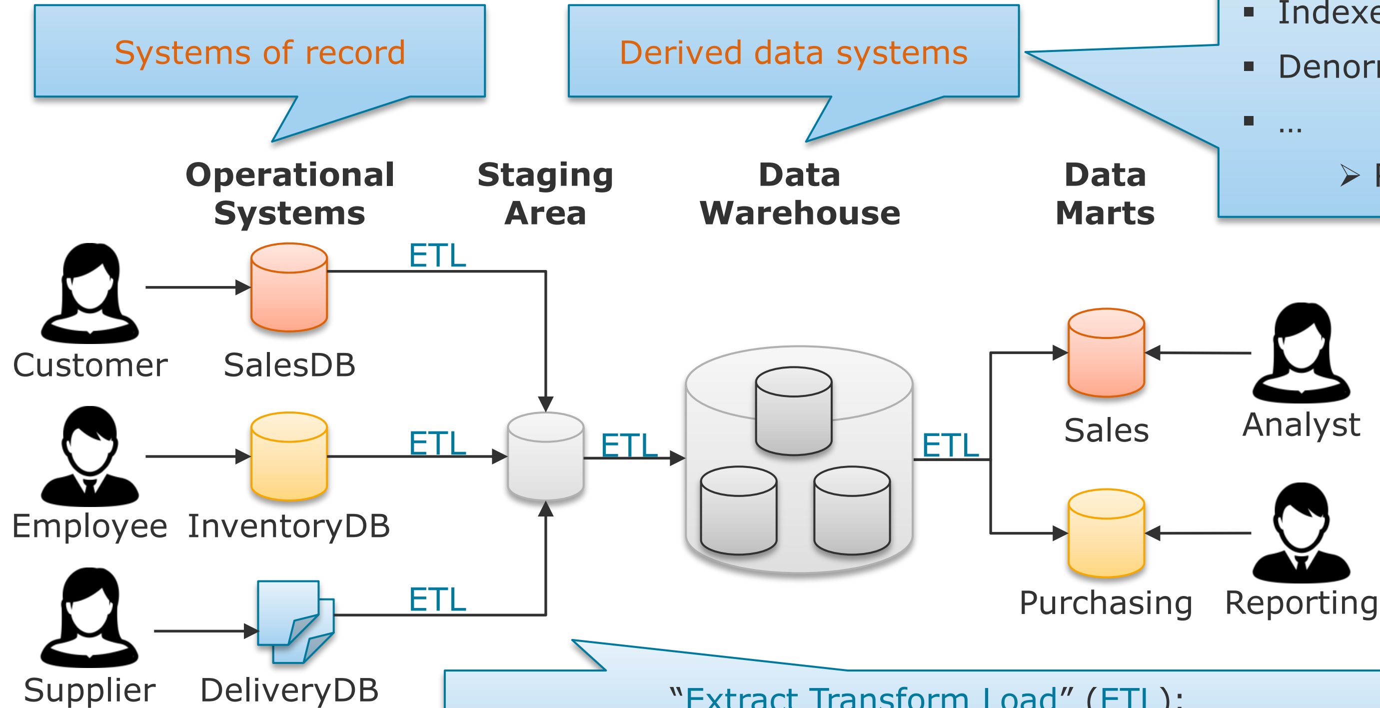
OLAP

ETL Systems

Not only data warehouses:

- Caches
- Materialized views
- Indexes
- Denormalized values
- …

➢ Redundant data!



"Extract Transform Load" (ETL):
Periodic data transformation processes (= batch jobs)
in the context of data warehousing

Use Cases

Transformation Algorithms
(parsing, filtering, translation, …)

Statistical and Numerical Algorithms
(aggregation, counting, summarization, …)

Approximating Algorithms
(similarity search, fuzzy matching, …)

Machine Learning Algorithms
(classification, recommendation, prediction, …)

Spatial Algorithms
(neighbor search, clustering, …)

The Sushi Principle

Sushi Principle:

- Also known as "raw data is better"
  Bobby Johnson and Joseph Adler at Strata+Hadoop World, 2015
- Data are sequences of bytes with arbitrary data model and encoding:
  - ➢ text, sensor readings, matrices, feature vectors, genome sequences, …
- Simply dump all data to disk and figure out how to process it later.
  - ➢ "data lakes" or "enterprise data hubs"
- A strategy for data warehousing:
  - Bringing data together is important (for across-dataset joins/unions).
  - Careful schema design is expensive and often lossy.
  - ➢ Data parsing and interpretation at query time, i.e., in batch processes

Distributed Data Management

Indicators

… **for** (distributed) batch processing:

- Job can be broken down into a sequence of standard tasks.
- Job has data parallel nature.
- Job targets data analytics or data transformation.
- Job uses very large datasets where data management is a central issue for computing the solution (distributed data sources, large intermediate state, …).
- Job is a complex query beyond simple item selections.

➢ Data analytics and data transformation jobs

Distributed Data Management

Indicators

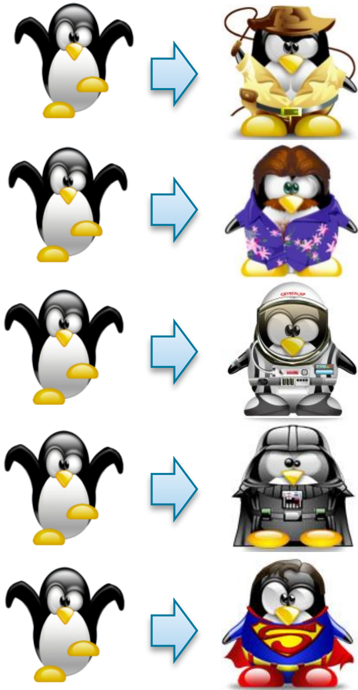
… **against** (distributed) batch processing:

- Job cannot be broken down into standard tasks.
- Job has task parallel nature.
- Job has many side effects.
- Job involves user interaction and other dynamic behavior.
- Job consists of complex branching logic.
- Job processes heterogeneous workloads
(many small, different tasks vs. one, big task).
- Job requires connections to many external systems.

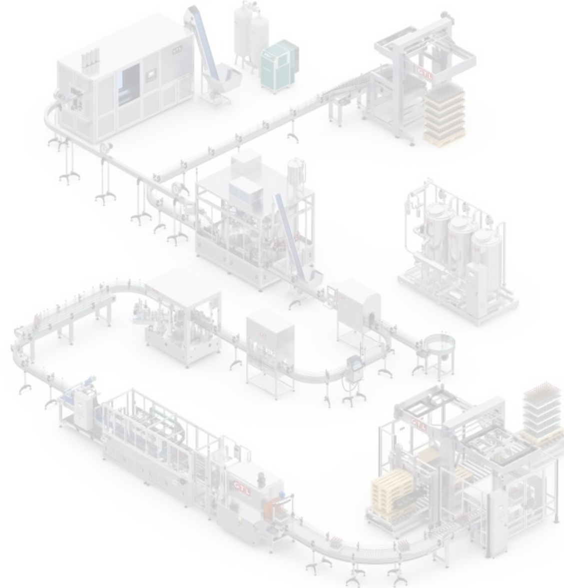
➢ Data intensive, scalable systems and applications

Overview
Batch Processing

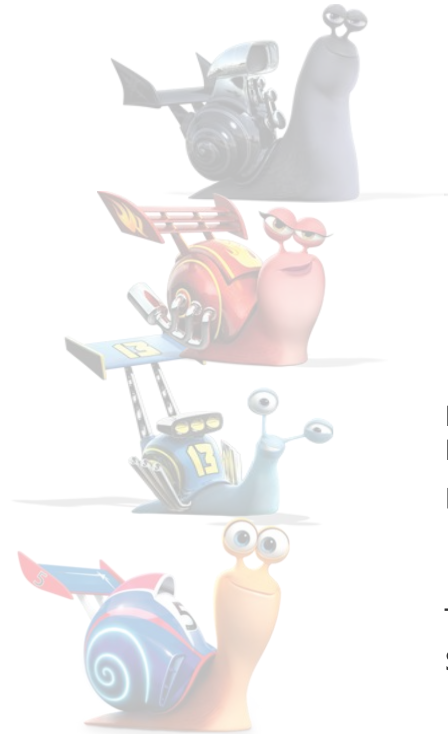
Batch Processing with Unix Tools



Distributed Filesystems and MapReduce



Beyond MapReduce



Analyzing a Log File

Log file analysis example:

- A log file generated by an nginx web server:

The "batch" of data

```
66.249.65.159 - - [06/Feb/2017:19:10:38 +0600] "GET /css/typography.css HTTP/1.1" 404 177 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/40.0.2214.15 Safari/537.36"

66.249.65.3 - - [06/Feb/2017:19:11:24 +0600] "GET /logo.ico HTTP/1.1" 200 4223 "-" "Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.google.com/bot.html)"

66.249.65.62 - - [06/Feb/2017:19:12:14 +0600] "GET /2013/05/24/batch-processing.html HTTP/1.1" 200 4356 "-" "Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.google.com/bot.html)"
```

- Log file format:

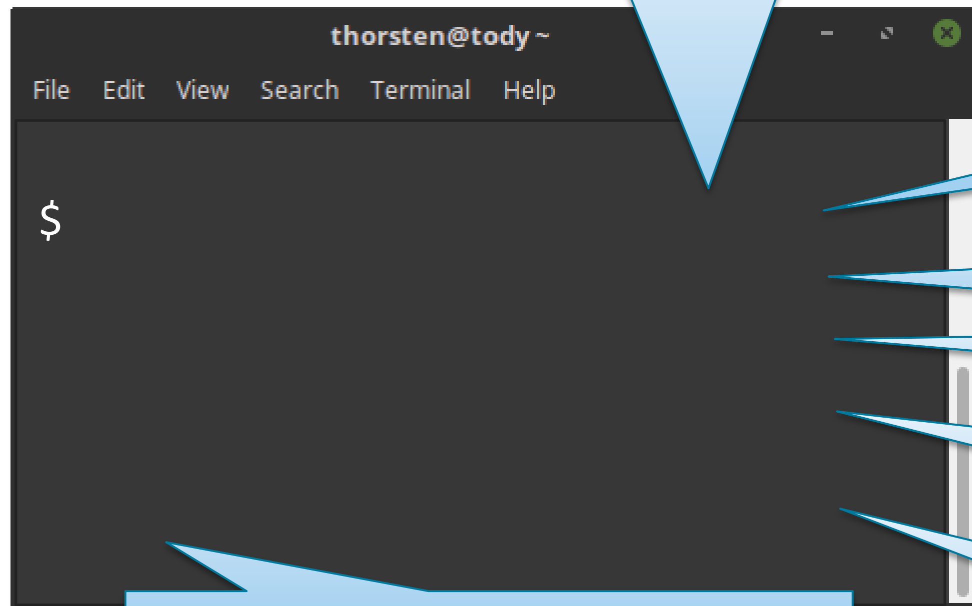
```
$remote_addr - $remote_user [$time_local] "$request" $status
$body_bytes_sent "$http_referer" "$http_user_agent"
```

- Task:
  - "Find the 5 most requested resources in the log"

Analyzing a Log File: Unix

Log file analysis example:

- Unix command chaining:



Unix pipes "|" connect the output of a command to the input of another.

Read the log file.

Split the lines on whitespaces and report line 7 (=request URL).

Sort the list of URLs alphabetically.

Filter out repeated lines and add a count to each distinct value ("-c").

Sort the count-URL-pairs numerically ("n") in reverse order ("r").

Output only the first n=5 count-URL-pairs

Analyzing a Log File: Unix

Log file analysis example:

- Unix command chaining:

```
66.249.65.159 - - [06/Feb/2017:19:10:38 +0600] "GET /css/typography.css HTTP/1.1" 404 177 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/40.0.2214.15 Safari/537.36"
66.249.65.3 - - [06/Feb/2017:19:11:24 +0600] "GET /logo.ico HTTP/1.1" 200 4223 "-" "Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.google.com/bot.html)"
66.249.65.62 - - [06/Feb/2017:19:12:14 +0600] "GET /2013/05/24/batch-processing.html HTTP/1.1" 200 4356 "-" "Mozilla/5.0 (compatible; Googlebot/2.1; +http://www.google.com/bot.html)"
```

thorsten@tody ~

File Edit View Search Terminal Help

```
$ cat /var/log/nginx/access.log  |
    awk '{print $7}'             |
    sort                         |
    uniq -c
    sort -r -n -h
    head -n 5
```

```
4189 /logo.ico
3631 /2013/05/24/batch-processing.html
2124 /2012/01/12/storage-and-retrieval.html
1369 /
 915 /css/typography.css
```

UNIX Command Concept

Philosophy

- Let each program do **one** thing well.
- Expect the **output** of every program to become the **input** of another.
- Build software to be **tried early.**
- Use **tools** in preference of unskilled help.

Larger programs are build by composing smaller programs.

Useful standard input stream commands (programs)

- **`grep`**: Searches for lines of text that match a regular expression.
- **`sed`**: Parses and transforms text in streams (usually for substitution or filtering).
- **`awk`**: Executes user defined AWK programming code on some input.
- **`xargs`**: Passes arguments to UNIX commands that disregard the standard input stream.
- **`sort`**: Sorts a batch of text lines by some criterion.
- **`uniq`**: Collapses adjacent identical text lines.

Commands vs. Custom Program

Hasso Plattner Institut

Commands

```
$ cat /var/log/nginx/access.log   |
    awk '{print $7}'              |
    sort                          |
    uniq -c                       |
    sort -r -n                    |
    head -n 5
```

UNIX

Custom Program

```
counts = Hash.new(0)

File.open('/var/log/nginx/access.log') do |file|
  file.each do |line|
    url = line.split[6]
    counts[url] += 1
  end
end

top5 = counts.map{|url, count| [count,url]}.sort.reverse[0...5]
top5.each{|count, url| puts "#{count} #{url}"}
```

Ruby

The command concept is …

- usually shorter (as operators hide complexity).
- better to maintain and optimize (as operators group functionality).
- **easier to distribute** (due to operator independence).

ThorstenPapenbrock

Although that's exactly what UNIX does **not**!

UNIX Command Concept

User defined functions (UDFs), though, may produce side effects!

What UNIX does right:

- Immutable inputs: Repetitive executions produce same results.
- No side-effects: Inputs are turned into outputs, nothing (bad/unforeseen/…) happens.
- Arbitrary stops: Pipelines can be ended at any point (e.g. via outputting with `less`).
- Persistence: Outputs can be written to files and used as inputs by other commands; allows to (re-)start later stages with rerunning the entire pipeline.

What UNIX does not:

- Distribution: All commands are executed locally on one machine.

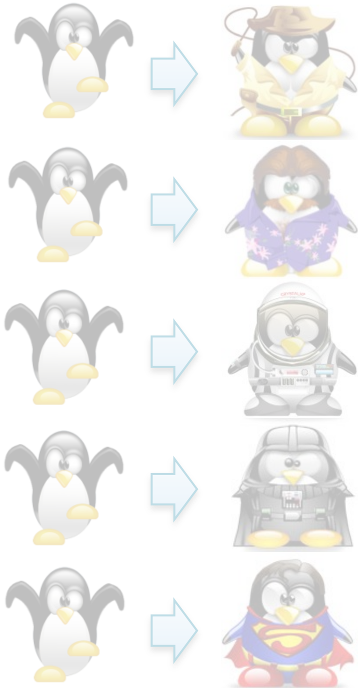
Keep the nice parts but go for **distribution**!

Cluster computing frameworks

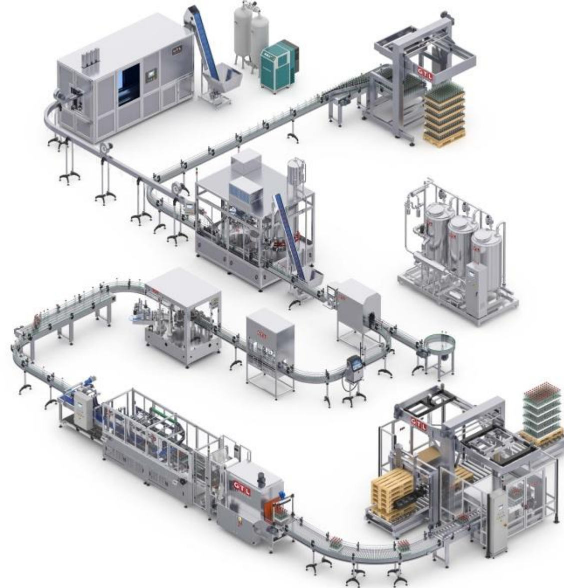
HPI Hasso Plattner Institut

Batch Processing

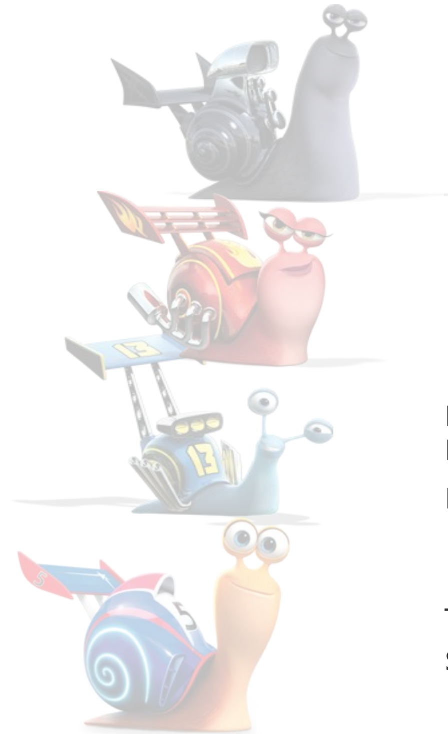
Batch Processing with Unix Tools



Distributed Filesystems and MapReduce



Beyond MapReduce



Distributed File System

Distributed File System

- Stores data (of any format) distributed on multiple shared-nothing machines
- Provides traditional file system interfaces, i.e., appear like local file systems
- Usually implemented as leader-based storage systems:
  - ➢ Availability and fault-tolerance (partitioning and replication)
- Examples:
  - Hadoop Distributed FS (HDFS), GlusterFS, Quantcast FS (QFS), Google FS (GFS) …
- **Not to be confused with …**
  - Network Attached Storage (NAS)
  - Storage Area Network (SAN)
    - ➢ RAID systems

Shared disc architectures with special hardware (head devices; fibre channels)

= Redundancy over several disks

Similar techniques for error detection and recovery, but locally in contrast to distributed FSs

Distributed File System

HPI Hasso Plattner Institut

Distributed File Systems for Distributed Batch Processing

- Serve as data source/sink for inputs/outputs of cluster computing frameworks
  - Similar to `stdin` and `stdout` in UNIX command pipelines
- Advantages for analytical cluster computing:

  - Light-weight: store and retrieve blocks of data (no fancy querying, i.e., joins, aggregations, …)

  - Resource localization: provide means to find the host of a block (to schedule client processes reading that block on the same node)

  - Resource allocation: provide means to explicitly assign blocks to nodes (for load-based resource re-allocation)

Distributed File System

Hadoop Distributed File System (HDFS)

- Arguably the most popular distributed file system implementation
- Subproject of Apache Hadoop; implemented in Java
- Optimized for …
  - Fault-tolerance (node crashes are seen as the norm, not an exception)
  - High throughput (not low latency)
  - Large files (gigabytes to petabytes; broken into 128 MB blocks)
  - Analytical workloads (write once, read many times)
  - Streaming data access (full file scans)
- Philosophy:
  - "Moving computation is cheaper than moving data"
    - Provides interfaces for applications to move themselves closer to where the data is located.

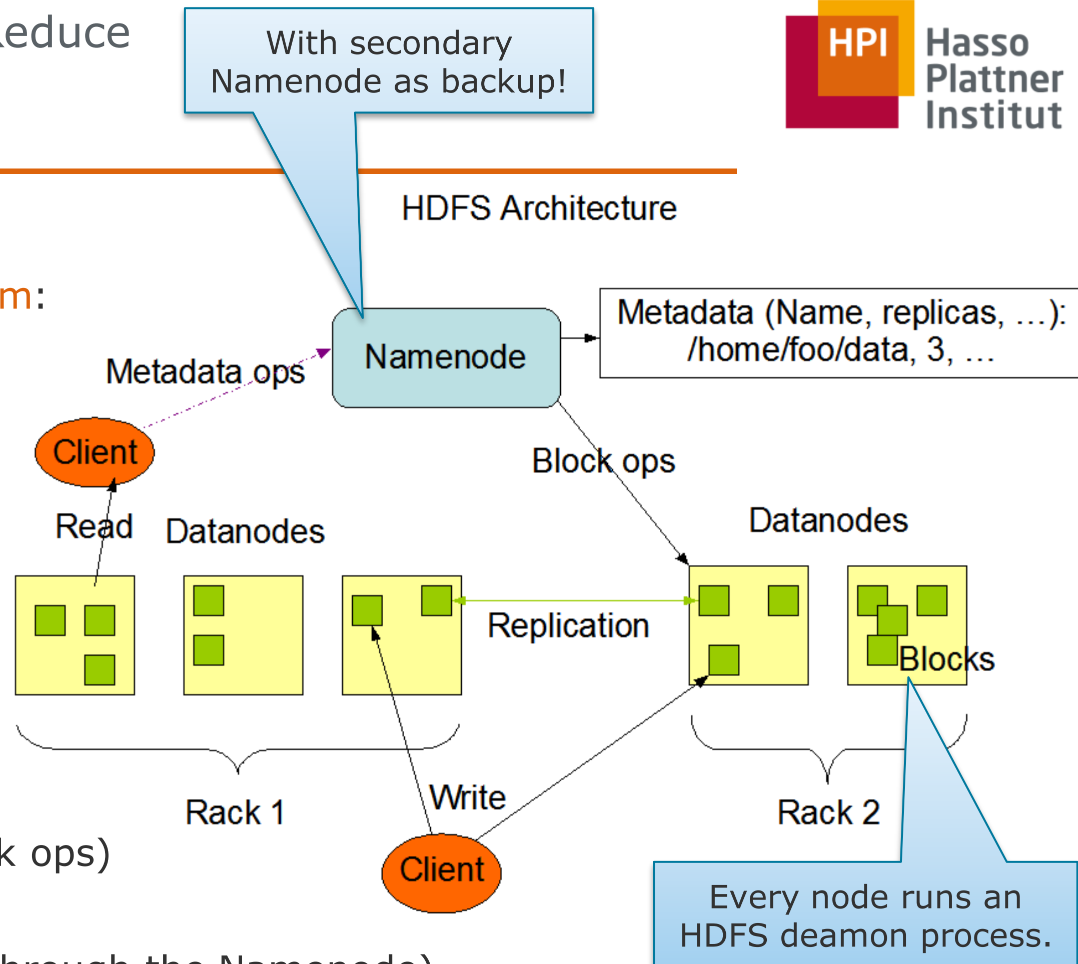
HDFS clusters often contain thousands of nodes!

Full CRUD support, but Reads dominate and Updates mean file overwrite

Distributed File System

Hadoop Distributed File System (HDFS)

- Classical leader-based storage system:
  - Namenode = Leader
  - Replication for fault-tolerance
  - Partitioning for parallel reads
- Namenode:
  - Serves all data management and localization requests
- Datanodes:
  - Handle the Namenode's data management instructions (Block ops)
  - Handle read and write requests from clients (data never flows through the Namenode)



Distributed File System

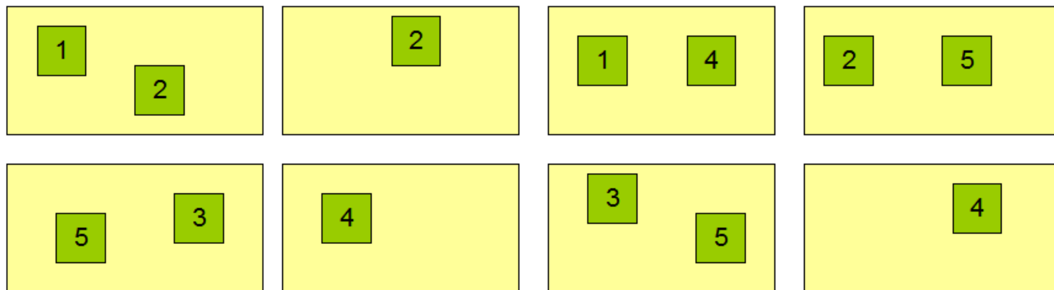
Hadoop Distributed File System (HDFS)

- Namespaces:
  - Traditional hierarchical file names and paths
  - Managed by Namenode
- Replication:
  - Number of replicas can be specified per file (default 3)
  - Replica placement policy can be configured
- Racks:
  - HDFS is aware of racks: nodes within one rack are more likely to fail jointly

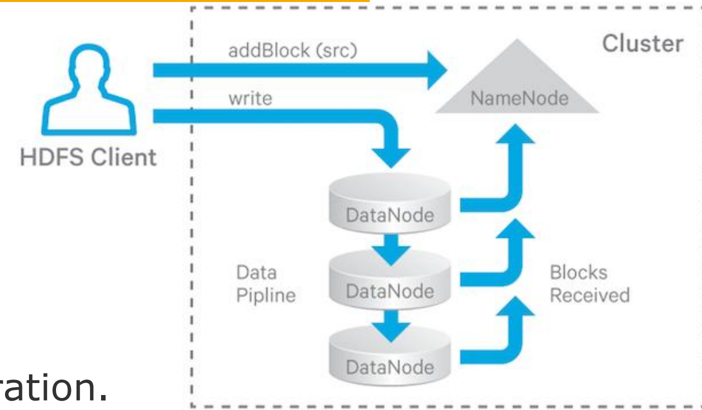
Block Replication

Namenode (Filename, numReplicas, block-ids, …)
/users/sameerp/data/part-0, r:2, {1,3}, …
/users/sameerp/data/part-1, r:3, {2,4,5}, …

Datanodes



Distributed File System



Hadoop Distributed File System (HDFS)

- Writes and Replication:
  1. Client node queries the namenode for a write operation.
  2. Namenode provides the address of the datanode with the target block.
  3. Client writes the data directly to the target datanode (not to any replica).
  4. Datanode starts to replicate the written block with all replicas of that block.
  5. Datanode acknowledges the write to the client.
  - Datanode crash: Namenode sends the address of an active datanode hosting the target block to proceed the write operation.
  - Client crash: *Datanodes may revert the block from a replica or finish the write.* *

    *in theory; might in practice corrupt the file

- Conflict management:
  - File access uses read/write locks, i.e., leases on blocks to avoid inconsistencies.

Distributed File System

Hadoop Distributed File System (HDFS)

- An example HDFS session:

thorsten@tody ~

File Edit View Search Terminal Help

```
$ start-dfs.sh
$ bin/hadoop fs -mkdir /user/input
$ bin/hadoop fs -put /home/file.txt /user/input
$ bin/hadoop fs -ls /user/input
$ bin/hadoop fs -cat /user/output/outfile.txt
$ bin/hadoop fs -get /user/output/ /home/hadoop_tp/
$ stop-dfs.sh
```

Start the HDFS

Create a folder

Copy a local file to the HDFS

List all current files in a folder

Print the content of a file

Copy a file from HDFS to local

Stop the HDFS (files persist)

Distributed File System

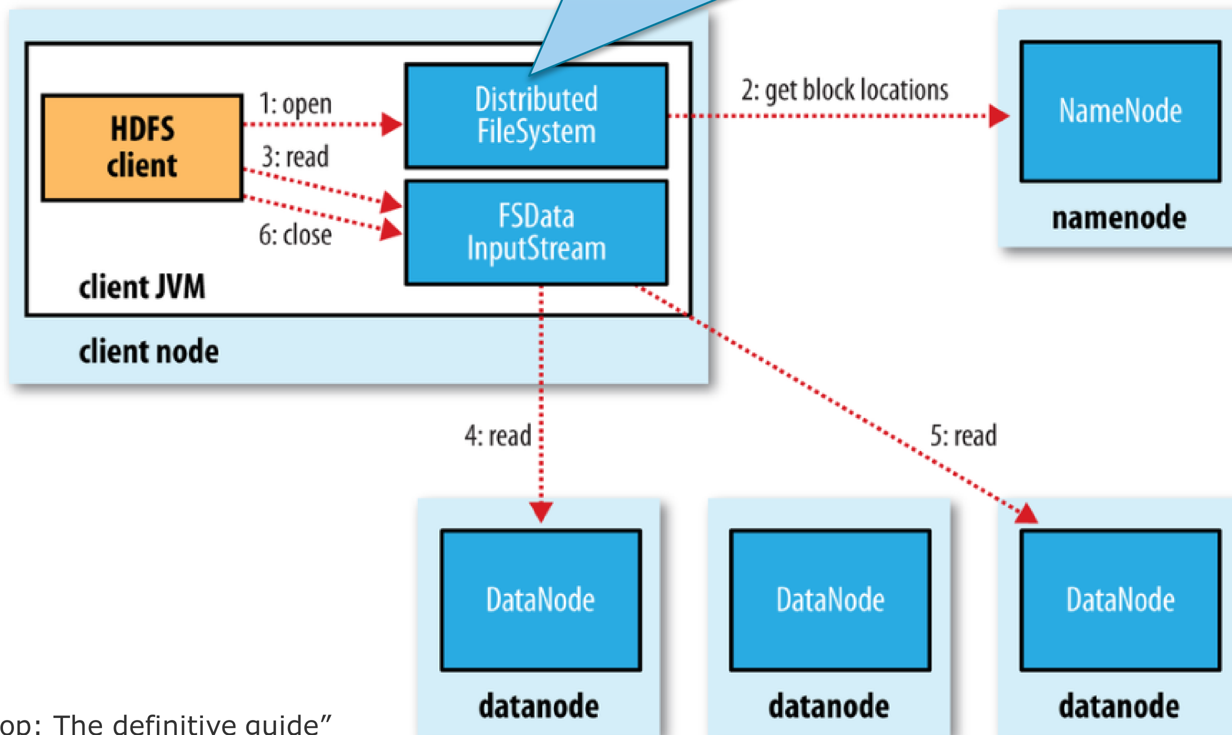
Hadoop Distributed File System (HDFS)

- File system commands:
  - `ls <path>` Lists the contents of the directory (names, owner, size, …).
  - `du <path>` Shows disk usage in bytes.
  - `cp <src> <dest>` Copies the file or directory from src to dest within HDFS.
  - `rm <path>` Removes a file or empty directory.
  - `put <src> <dest>` Copies a local file or directory to DFS.
  - `get <src> <dest>` Copies a file or directory from HDFS to local file system.
  - `cat <file-name>` Displays the contents of file on stdout.
  - `mkdir <path>` Creates a directory in HDFS.
  - `setrep <rep> <path>` Sets the targets replication factor to rep.
  - `chown <path>` Sets the owning user and/or group.
  - `help <cmd-name>` Returns usage information for a command.
  - …

Distributed File System

Hadoop's Java API

Classes that abstract from the distribution,
but what if you need to exploit the distribution?



See book: "Hadoop: The definitive guide"

Distributed File System

Hadoop's Java API: Pushing computation to the relevant blocks

See MapReduce, Spark, Flink, …

1. Programmatically query the block locations.
2. Run the code for these blocks on the same machines.

```
<dependency>
   <groupId>org.apache.hadoop</groupId>
   <artifactId>hadoop-hdfs</artifactId>
   <version>3.1.1</version>
</dependency>
```

```
<dependency>
   <groupId>org.apache.hadoop</groupId>
   <artifactId>hadoop-common</artifactId>
   <version>3.1.1</version>
</dependency>
```

```
FileSystem fs = FileSystem.get(conf);
Path dataset = new Path(fs.getHomeDirectory(), "/path/to/file.csv");
FileStatus status = fs.getFileStatus(dataset);
BlockLocation[] locations = fs.getFileBlockLocations(status, 0, status.getLen());
for (BlockLocation loc : locations) {
   System.out.println("Length: " + loc.getLength());
   for (String host : loc.getHosts()) {
      System.out.println("Host: " + host);
   }
}
```

Multiple hosts due to replication

Distributed Batch Processing

Functional Programming

- A programming paradigm where code is based on the evaluation functions.
- Function:
  - Takes argument(s) and produces output(s)
  - Avoid side-effects: no changing state; no mutable objects
    - ➢ Calls with same arguments produce same outputs.
    - ➢ Very nice property for parallelization/distribution!
  - See UNIX commands
- Functional languages:
  - Fully functional: Lisp, Clojure, Erlang, Haskell, …
  - Functional nature: JavaScript, Scala, …
  - Functional support: C++, C#, Java, PHP, Perl, …
  - **Functional distribution: MapReduce, Spark, Flink, Storm, …**

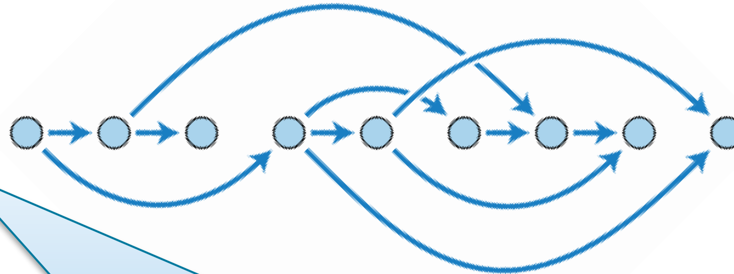
Main difference to procedures!

Allow side-effects and procedural/imperative code

Distributed Batch Processing

Data Transformation Pipeline

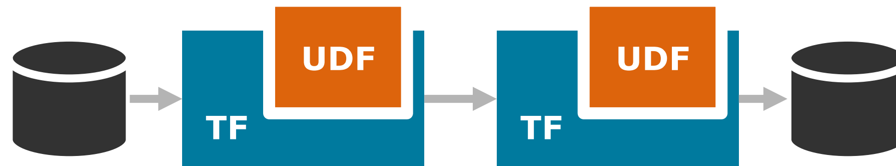
- Chaining of functions:
  - Function outputs are forwarded to inputs of other functions.
  - One output can become the input of more than one other function.
- Directed (acyclic) graphs (DAG):
  - functions = nodes
  - dataflow = edges
  - acyclic: no directed loops
  - ➢ Topological ordering possible



Some distribution frameworks support loops, which are needed for iterative algorithms (machine learning, clustering, data cleaning, …)

Distributed Batch Processing

Data Transformation Pipeline



- Transformation functions (TF):
  - Fixed signature and implementation
  - Used to construct the pipeline topology
  - Have no side-effects
  - Operate on the same data formats
  - May take and apply user defined callback functions
- User defined callback functions (UDF):
  - Implement a certain, transformation-function-specific interface
  - Define the application specific tasks within a transformation function

Distributed Batch Processing

Data Transformation Pipeline

- Example: Function pipelining in Java 1.8+

```
String[] records = {"lisa,17", "mike,32", "carl,12", "nina,24"};
Stream.of(records)
        .distinct()
        .map(s -> s.split(","))
        .filter(t -> Integer.valueOf(t[1]).intValue() >= 18)
        .map(t -> t[0])
        .sorted()
        .forEach(System.out::println);
```

TF
UDF
UDF
UDF
UDF

- Characteristics:
  - Immutable inputs and outputs: Pipelines do no harm; can re-execute and output intermediate results (e.g. for debugging).
  - Separation of logic and wiring: Outputs match inputs; transformation functions can be optimized independently from their UDFs.

Distributed Batch Processing

Data Transformation Pipeline

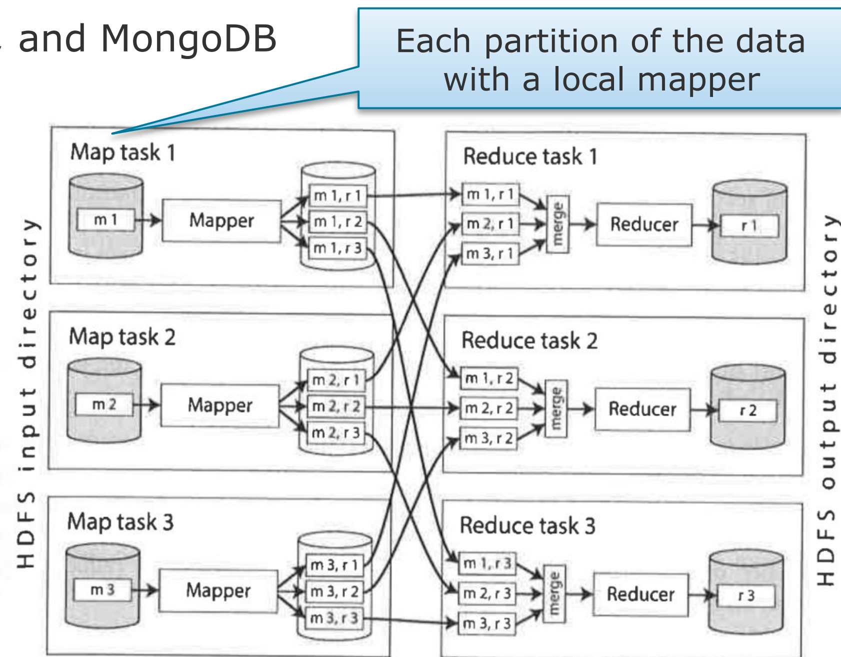
- Data parallel:
  - The same function can be applied to different data in parallel
    (functions can always operate on different data due to stateless nature).
- Task parallel:
  - Consecutive functions can be executed in parallel
    (start working on output before previous function has finished).

MapReduce

MapReduce

- A low-level, functional programming model for analytical cluster computing
- Implemented in, for instance, Hadoop, CouchDB, and MongoDB
- Defines two transformation functions:
  - **map()** `Map(k1,v1)` → `list(k2,v2)`
    Takes one key-value pair (e.g. filename and records); outputs key-value pair(s).
  - **reduce()** `Reduce(k2, list (v2))` → `list(v3)`
    Takes a collection of key-value pairs with same key; outputs value(s).
- Both functions apply a user defined callback function on their inputs.

Each partition of the data with a local mapper



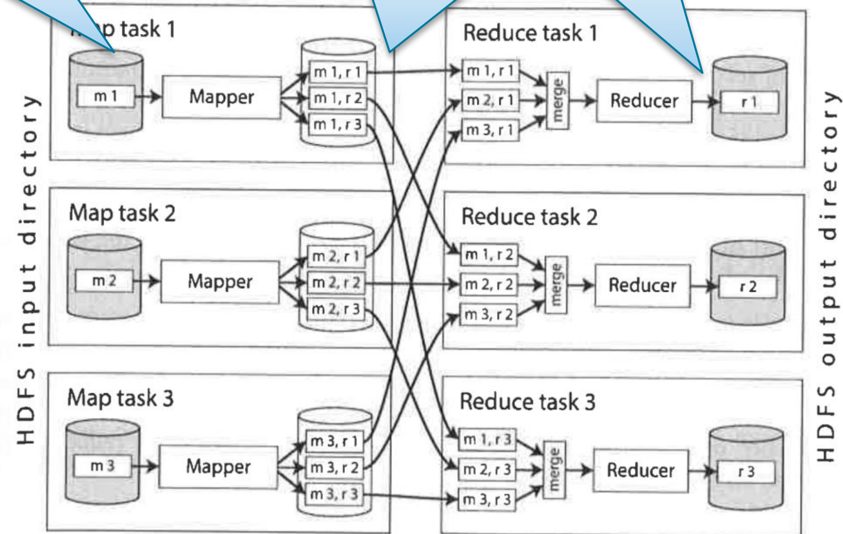
MapReduce

It's rather Map-Sort-Group-Reduce ;-)

MapReduce

- A low-level, functional programming model for analytical cluster computing
- Implemented in, for instance, Hadoop, CouchDB, and MongoDB
- The framework's tasks:
  1. Read the input data from HDFS, split continuous data into records, assign values to nearby mappers.
  2. Write/read key-value pairs to/from HDFS, group pairs with same key (via sorting), assign keys to reducers.
  3. Write reducer results to HDFS.
- Reducer functions get an iterator for their inputs (they can start while pairs are still being added).

1. Input reader
2. Partition & compare
3. Output writer

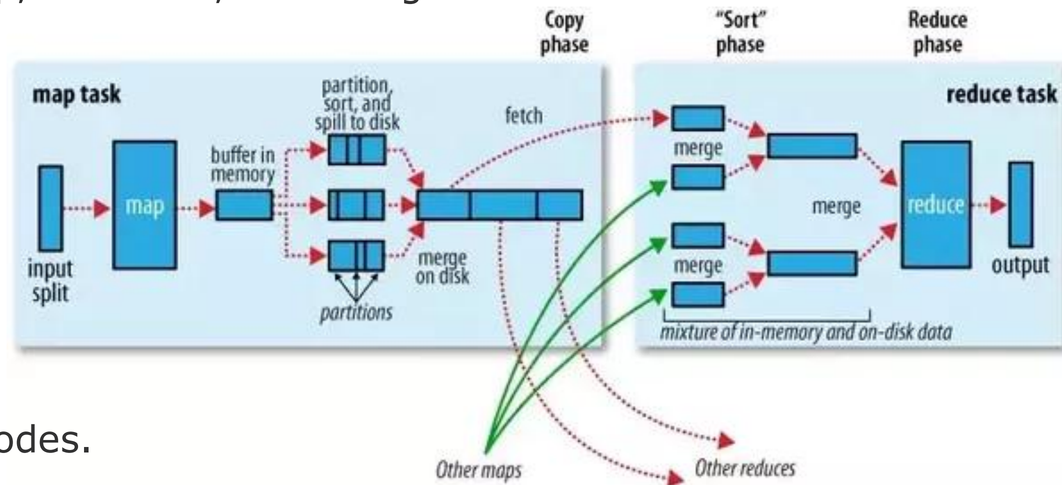


MapReduce

MapReduce

- A low-level, functional programming model for analytical cluster computing
- Implemented in, for instance, Hadoop, CouchDB, and MongoDB

- The MapReduce sort:
  1. Pre-sort the output of all local mappers (Quicksort).
  2. Persist the output on disk.
     - ➢ Block until pre-sort done!
  3. Range partition the key space and assign ranges to reducer nodes.
  4. Sort the pre-sorted partitions on the reducer nodes (Mergesort).



➢ Super optimized and (due to persistence) reliable operation!

MapReduce

MapReduce

- Sandwich

Map

Reduce

"Every task is map-reduce!"

© https://clojurebridgelondon.github.io/workshop/functions/map-reduce-sandwich.html

MapReduce

MapReduce

- Word Count

```
function map(String name, String document):
  // name: document name
  // document: document contents
  for each word w in document:
    emit (w, 1)

function reduce(String word, Iterator partialCounts):
  // word: a word
  // partialCounts: a list of aggregated partial counts
  sum = 0
  for each pc in partialCounts:
    sum += pc
  emit (word, sum)
```

A mapper could also pre-aggregate words in its document.

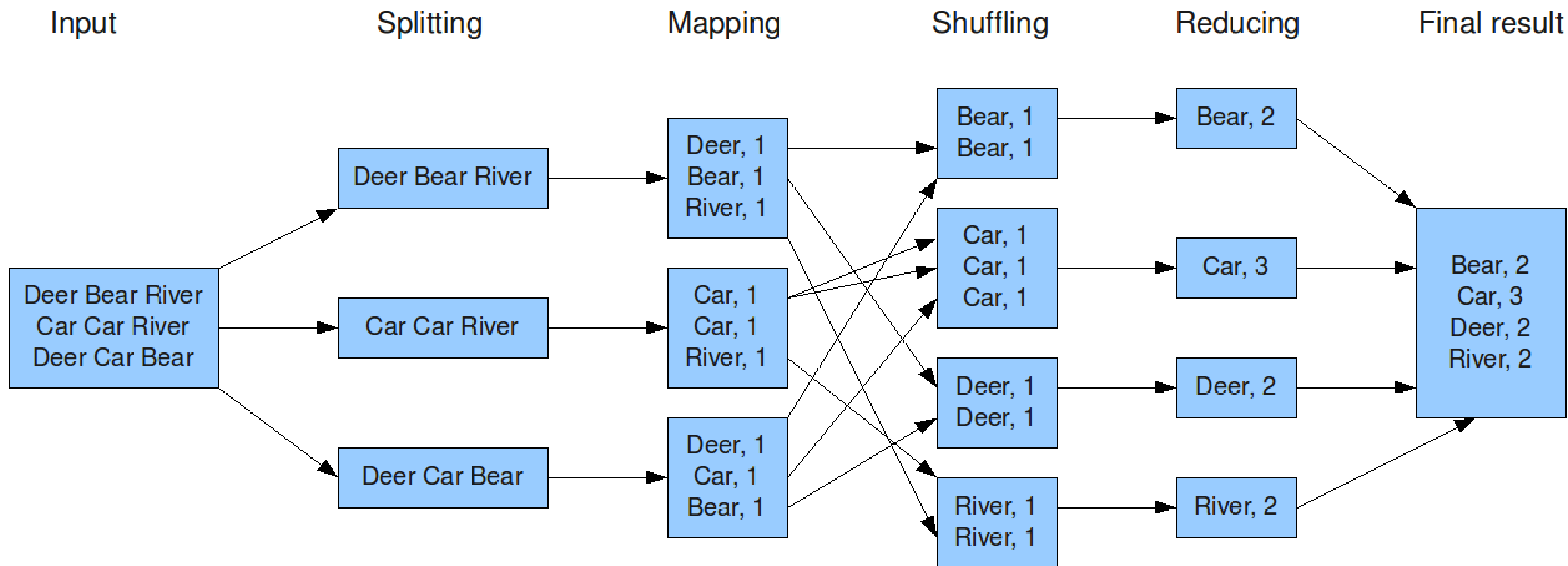
```
public static class TokenizerMapper
     extends Mapper<Object, Text, Text, IntWritable>{

  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(Object key, Text value, Context context
                  ) throws IOException, InterruptedException {
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
      word.set(itr.nextToken());
      context.write(word, one);
    }
  }
}

public static class IntSumReducer
     extends Reducer<Text,IntWritable,Text,IntWritable> {
  private IntWritable result = new IntWritable();

  public void reduce(Text key, Iterable<IntWritable> values,
                     Context context
                     ) throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
      sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
  }
}
```

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

MapReduce

HPI Hasso Plattner Institut

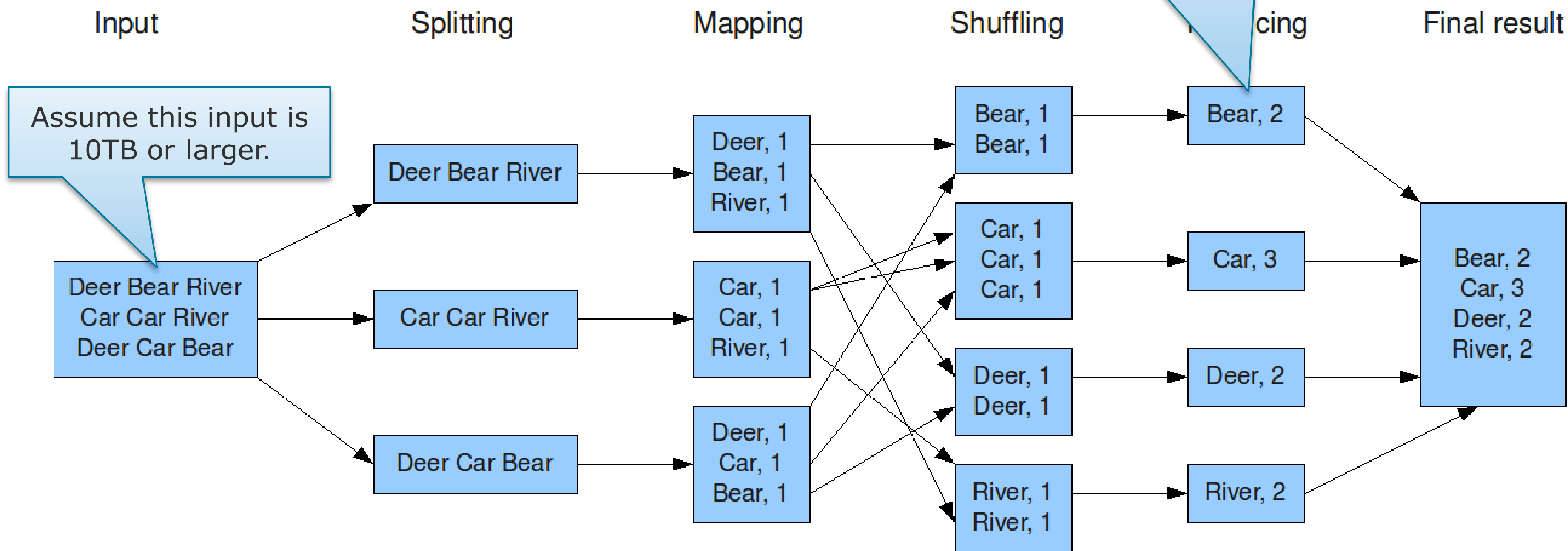
Example: Word Count



MapReduce

Example: Word Count

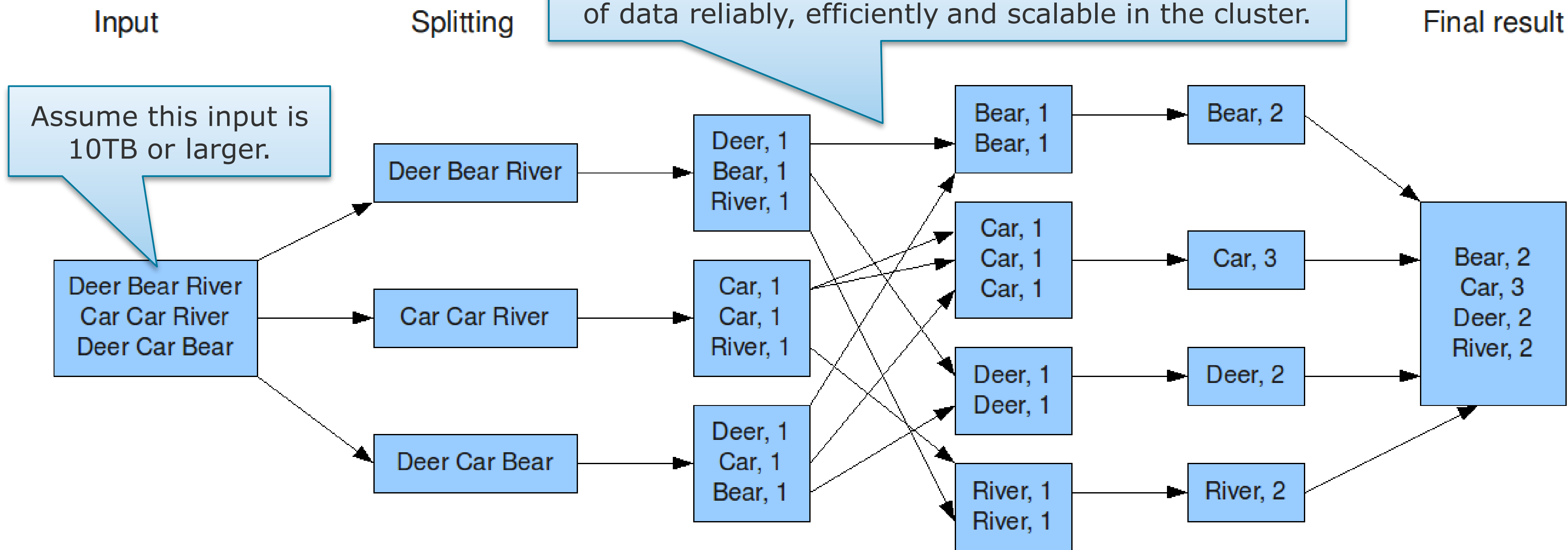
- Why is this cool?



MapReduce

Example: Word Count

- Why is this cool?



MapReduce

MapReduce Scheduler

- Pipeline startup:
  - Assigns transformation functions (and their processes) to nodes.
  - Tries to execute functions near the data that they are started with.
  - Considers that nodes have enough spare RAM and CPU for their jobs.
- Shuffling:
  - Group key-value pairs by key:
    - Pre-sort key-value pairs on each node.
    - Calculate global sortation by merging pre-sorted lists.
    - ➢ Idea similar as in "SSTables and LSMTrees"
  - Distribute load (= key-value pair groups) evenly across reducer nodes.
    - ➢ See "partitioning by hash of key"

MapReduce

Reduce-side, sort-merge Joins

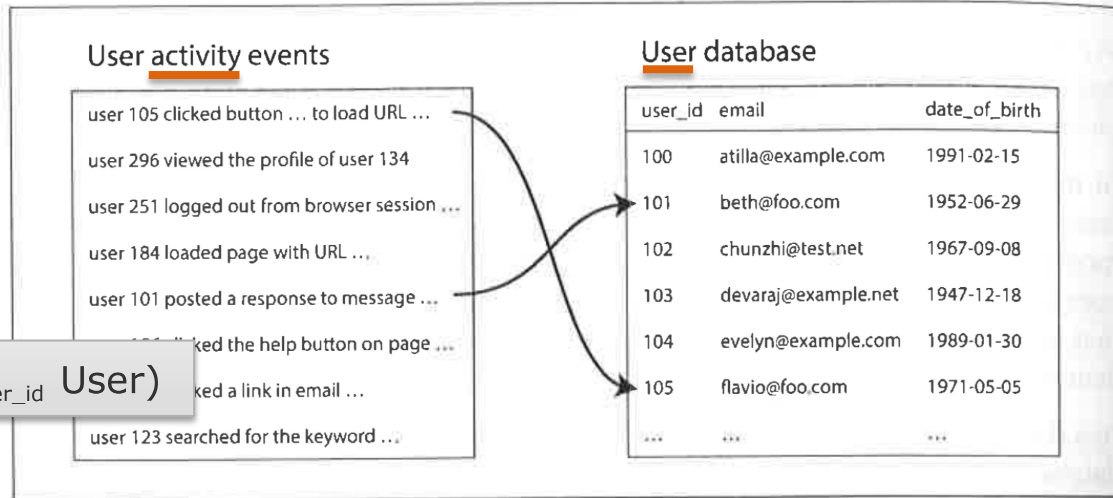
- Example:

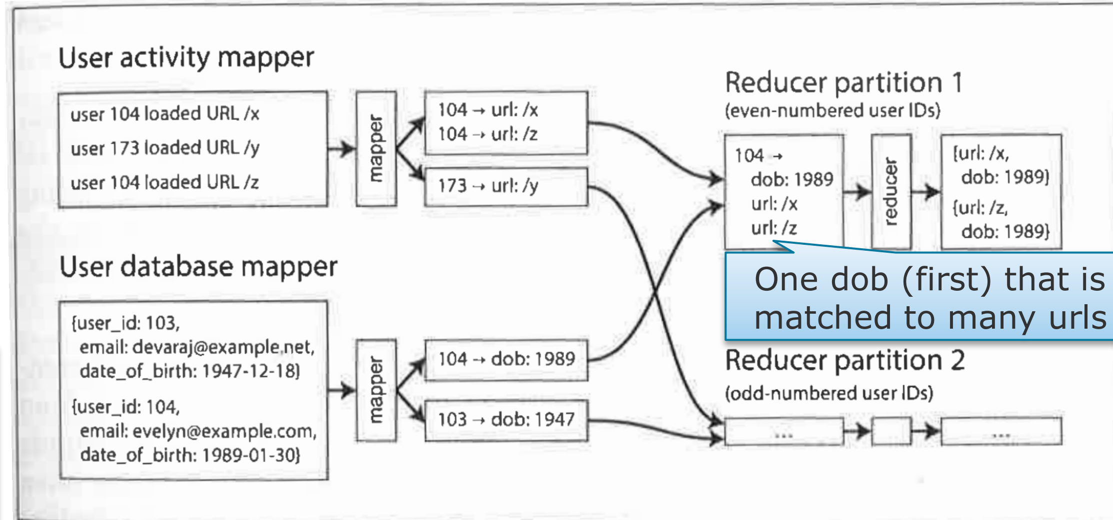
$$\Pi_{Activities.url,User.dob}(Activities \bowtie_{Activities.user=User.user_id} User)$$

- Use two mapper:
  - one for each side of the join
  - mapping key = join key
- Primary sort:
  - by key (user/user_id)
- Secondary sort:
  - by value (url > dob)

In this way, the records from the two relations arrive grouped in each reducer.

➢ Faster pairing algorithm





One dob (first) that is matched to many urls

MapReduce

Join relations in the mapping phase without a reduce phase!

Map-side Joins

- Broadcast hash join:
  - Criterion: One side of the join is small and fits into main memory.
  - Algorithm:
    1. Broadcast the small relation to all mappers which store it in memory.
    2. Mappers read the large partition and hash-join records to in-memory records.
- Partitioned hash join:
  - Algorithm:
    1. Partition both sides of the join by their join attributes.
    2. Mappers read the same partition from both sides and hash-join their records.
- Merge join:
  - Algorithm:
    1. Partition and sort both sides by their join attributes.
    2. Mappers read the same partition from both sides and merge-join their records.

"replicated join" (Pig)
"MapJoin" (Hive)

"bucketed map join" (Hive)

Apache Hadoop

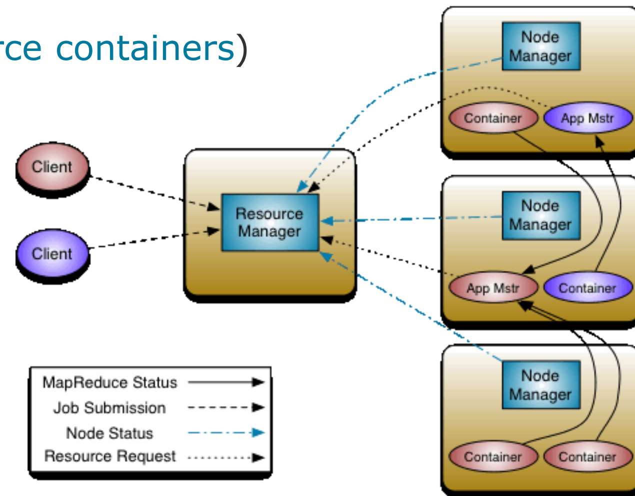
Apache Hadoop

- A framework implementing MapReduce on HDFS (http://hadoop.apache.org/)
- Published by Google and Yahoo in 2004
- Contains the following modules:
  - Hadoop Common: General libraries and utilities for Hadoop modules
  - Hadoop Distributed File System (HDFS): A distributed file system implementation that stores data on commodity machines
  - Hadoop YARN: A platform responsible for managing computing resources in clusters and using them for scheduling applications
  - Hadoop MapReduce: A MapReduce implementation
- Did the Terraby Sort in 2008 on 910 nodes in 3.48 minutes (http://sortbenchmark.org/YahooHadoop.pdf).

Apache Hadoop

Apache Hadoop YARN

- "Yet Another Resource Negotiator" (YARN)
- A cluster manager for physical resources of an HDFS
- Resources: CPU, memory, disk, network, … (grouped in resource containers)
- YARN consists of …
  a. ResourceManager:
     - Global resource management and allocation
  b. ApplicationMaster:
     - Configuration, coordination, and distribution of services (ZooKeeper)
     - Negotiates resources with the ResourceManager
     - Framework-specific (YARN also works with e.g. Spark)
- Takes compiled binaries and distributes their execution in the cluster



https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

Apache Hadoop

Limitations

- Disk-based messaging:
  - Intermediate results always move via disk from mapper to reducer.
- Static job structure:
  - MapReduce jobs always consist of (one) map and (zero or one) reduce.
  - No arbitrary chaining of these two functions.
  - Larger workflows need to run multiple jobs one after the other.
  - ➢ Need for workflow scheduler that run and manage workflows (e.g. Oozie, Azkaban, Luigi, Airflow, and Pinball).
- Only two functions:
  - Most queries can be expressed as sequence of MapReduce jobs, but additional, dedicated functions can improve the performance (e.g. sorting, deduplication, union, joining, …).

Apache Hadoop

Higher-level tools on top of Apache Hadoop

By Yahoo

- Apache Pig:
  - A platform that defines the higher-level query language Pig Latin.
  - Compiles Pig Latin queries to Hadoop MapReduce, Tez, or Spark jobs.
  - Pig Latin is similar to SQL with procedural elements.
  - Example:

```
A = LOAD 'products' USING PigStorage() AS (name:chararry, weight:int, price:int);
B = GROUP A BY price;
C = FOREACH B GENERATE price, COUNT($0);
DUMP C;
```

```
function map(String name, String file):
    for each product p in file
        emit (p.rice, p.name)
function reduce(Int price, Iterator names):
    count = 0
    for each name in names
        count++
    emit (price,count)
```

```
(0.10,14)
(0.25,23)
(0.50,16)
(1.99,82)
(2.50,17)
```

http://pig.apache.org/docs/r0.17.0/basic.html

Apache Hadoop

Hasso Plattner Institut

Higher-level tools on top of Apache Hadoop

- Apache Hive:
  - A tool that defines the higher-level query language HiveQL.
  - Compiles HiveQL queries to Hadoop MapReduce, Tez, or Spark jobs.
  - HiveQL is based on SQL and provides some extensions.
  - Example:

By Facebook

```
INSERT OVERWRITE TABLE pv_users
SELECT pv.*, u.gender, u.age
FROM user u JOIN page_view pv ON (pv.userid = u.id)
WHERE pv.date = '2017-03-03';
```

Results are always written to a table

https://cwiki.apache.org/confluence/display/Hive/Tutorial

Apache Hadoop

Higher-level tools on top of Apache Hadoop

- Hive vs. Pig: Word count

```
1 input_lines = LOAD '/tmp/word.txt' AS (line:chararray);
2 words = FOREACH input_lines GENERATE FLATTEN(TOKENIZE(line)) AS word;
3 filtered_words = FILTER words BY word MATCHES '\\w+';
4 word_groups = GROUP filtered_words BY word;
5 word_count = FOREACH word_groups GENERATE COUNT(filtered_words) AS count, group AS word;
6 ordered_word_count = ORDER word_count BY count DESC;
7 STORE ordered_word_count INTO '/tmp/results.txt';
```

```
1 DROP TABLE IF EXISTS docs;
2 CREATE TABLE docs (line STRING);
3 LOAD DATA INPATH 'input_file' OVERWRITE INTO TABLE docs;
4 CREATE TABLE word_counts AS
5 SELECT word, count(1) AS count FROM
6  (SELECT explode(split(line, '\s')) AS word FROM docs) temp
7 GROUP BY word
8 ORDER BY word;
```

Apache Hadoop

HPI Hasso Plattner Institut

Higher-level tools on top of Apache Hadoop

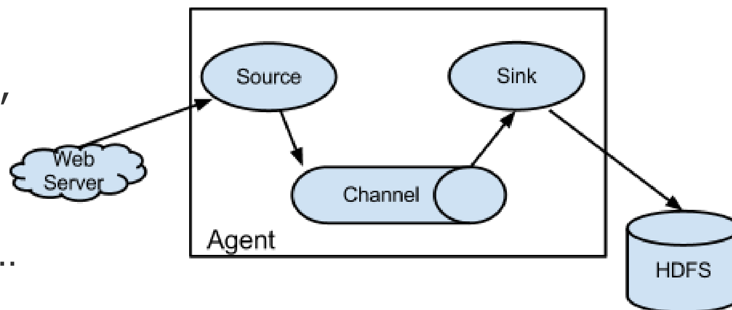
- Hive vs. Pig:

Hive	Pig
Declarative SQLish Language	Procedural Data Flow Language
Designed for creating reports	**Designed for programming**
Mainly used by data analysts	Mainly used by researchers/programmers
No dedicated metadata database	Uses dedicated tables for results
Supports the Avro file format	Does not support it

HPI Hasso Plattner Institut

Higher-level tools on top of Apache Hadoop

- Apache Flume:
  - A distributed service for collecting, aggregating, and moving large amounts of streaming data into HDFS
  - Reliable: Uses transactions to guarantee message delivery.
  - Available: Buffers data in case of load spikes (if sink cannot keep up).
  - Sources:
    - Supported:
      - ElasticSearch, Hbase, Hive, Kafka, …
    - In Development:
      - Storm, Solr, Spark, …



Apache Hadoop

Apache Hadoop Ecosystem

Ambari
Provisioning, Managing and Monitoring Hadoop Clusters

Sqoop
Data Exchange

Flume
Log Collector

Zookeeper
Coordination

Oozie
Workflow

Pig
Scripting

Mahout
Machine Learning

R Connectors
Statistics

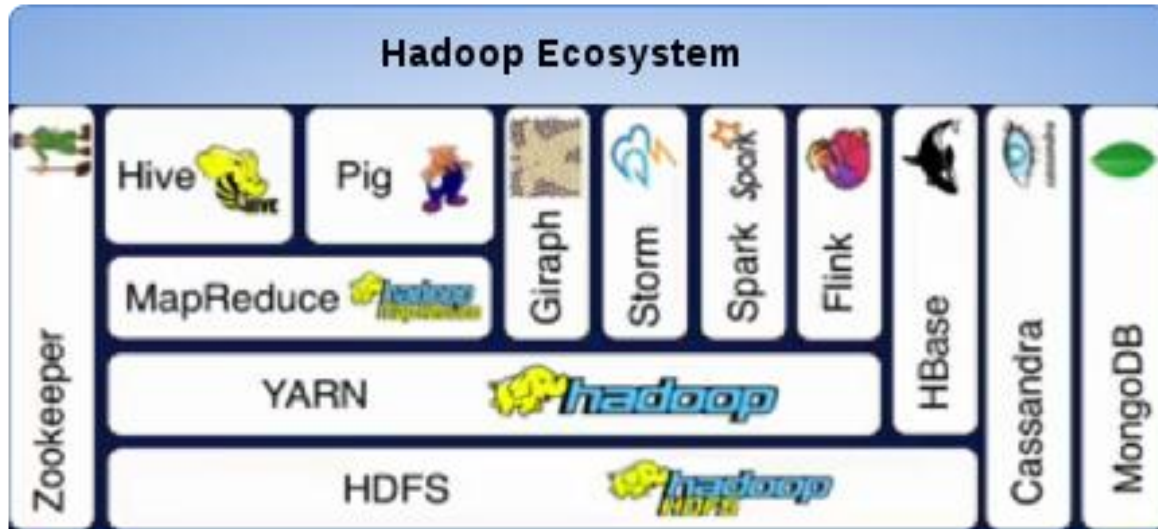
Hive
SQL Query

Hbase
Columnar Store

YARN Map Reduce v2
Distributed Processing Framework

HDFS
Hadoop Distributed File System

Apache Hadoop



Apache Hadoop

A cluster resource manager for arbitrary (also non-Hadoop) technologies

Hadoop vs. MPP Databases

Massively Parallel Processing (MPP) Databases

- Classical databases with distributed storage and parallel query processing
- Use the same concepts as we discussed earlier:
  - Partitioning, replication, distributed indexes, functional parallelism, …

How Hadoop (and other MapReduce variations) differ

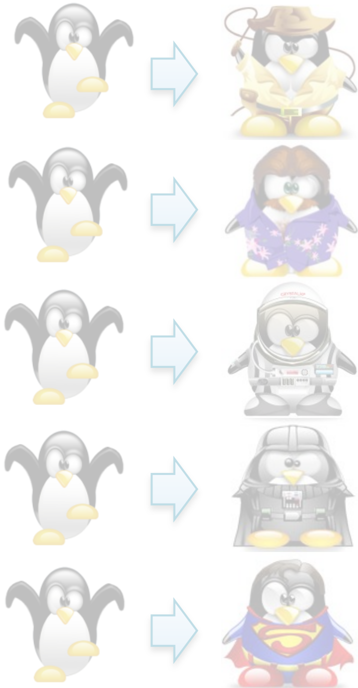
- Diversity of storage
  - Hadoop is data model independent; MPP databases enforce one
- Diversity of processing models
  - Hadoop executes arbitrary UDFs; MPP database only SQL.
- Design for frequent faults
  - Hadoop tries to recover from faults; MPP databases abort failed queries.
- Reading workloads
  - Hadoop answers queries; MPP databases support full data manipulation.

It is difficult to do machine learning, image processing, full-text search, … in SQL!

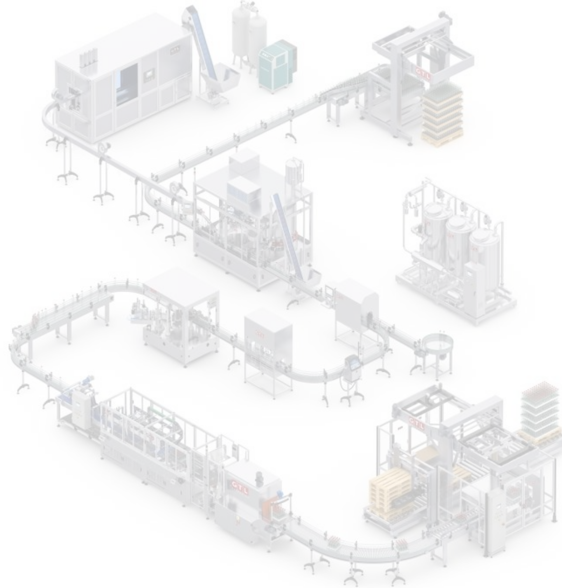
Long running OLAP queries!

Batch Processing

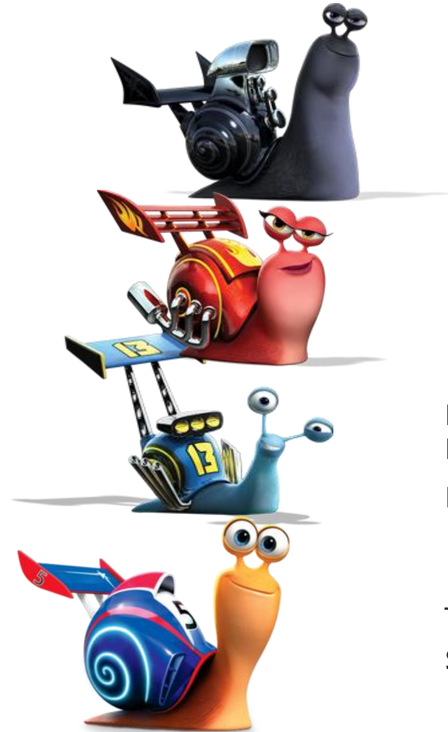
Batch Processing with Unix Tools



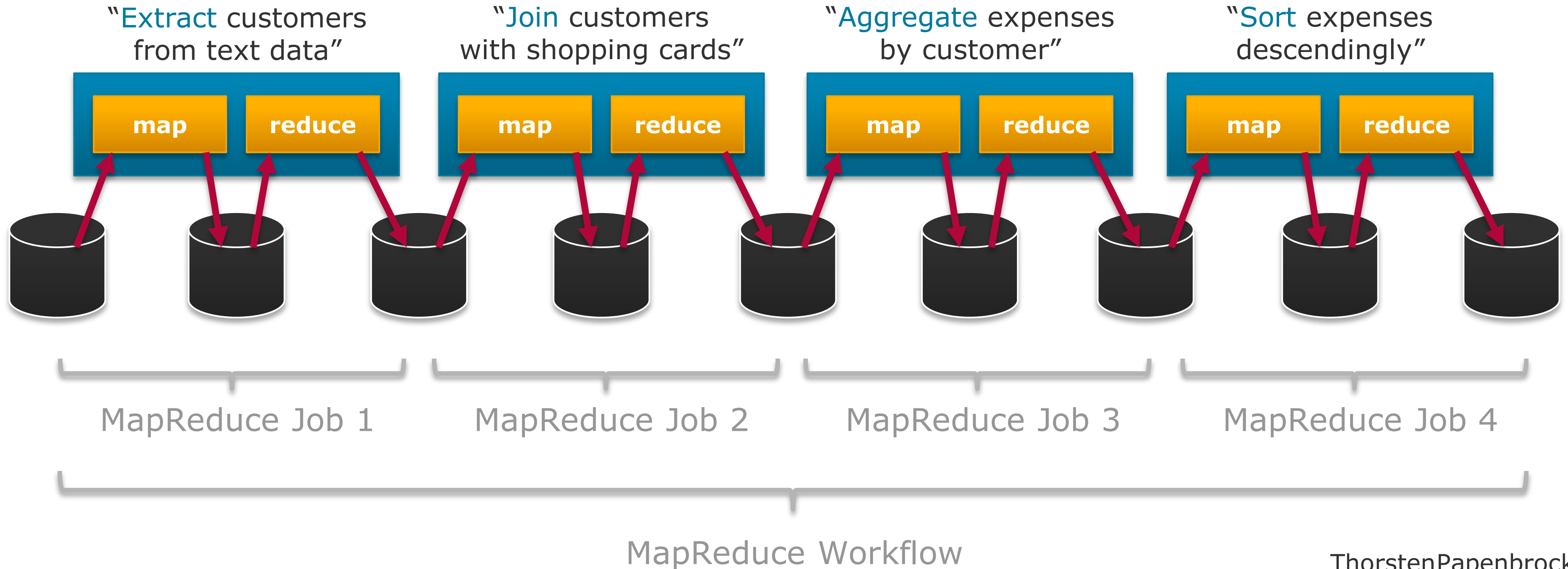
Distributed Filesystems and MapReduce



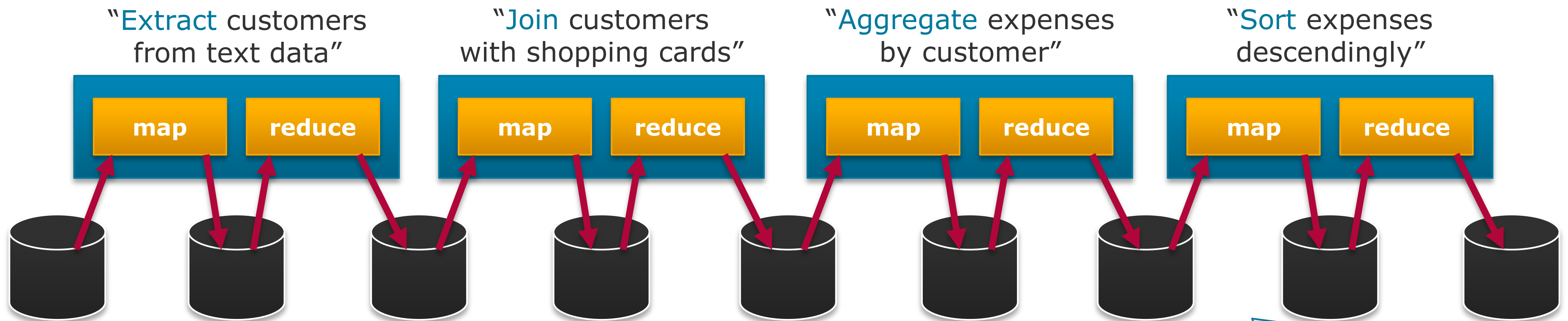
Beyond MapReduce



MapReduce's Workflow Design



MapReduce's Workflow Design



Redundant mapper work:
Mappers often only read back what reducers wrote.

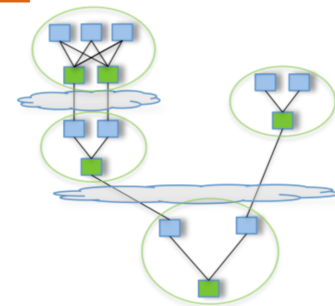
- Data is always written to disk.
  - ➢ Good for reliability
  - ➢ Bad for performance: I/O overhead and no "real" pipelining (i.e., later jobs wait until earlier jobs have completely finished)
- Workflow construction is complicated (without Hive, Pig, Cascading, etc.).

Data Flow Engines

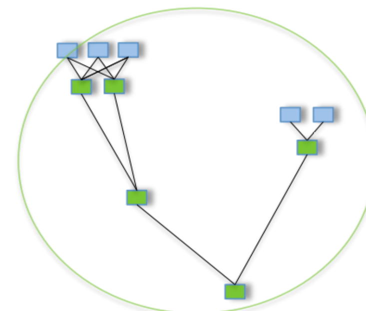
Data Flow Engine

- MapReduce-like distributed batch processing frameworks that handle an entire workflow as one job.
- Directed Acyclic Graph (DAG) processing:
  - Transformation functions (operators) can form arbitrary DAGs.
    - ➢ No strict alternating map-reduce
  - Outputs are directly connected to inputs of other operators.
    - ➢ No mandatory disk usage
  - Sorting is done explicitly.
    - ➢ Not in every shuffle step
  - Scheduler can place subsequent tasks that consume the same data on the same machine, because it knows the entire pipeline.
    - ➢ Less unnecessary data shuffling

"locality optimizations"



Pig/Hive - MR



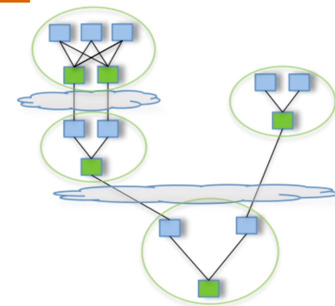
Pig/Hive - Tez

Data Flow Engines

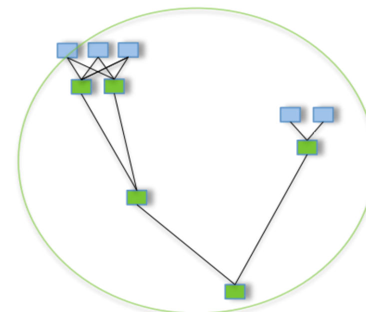
Hasso Plattner Institut

Data Flow Engine

- MapReduce-like distributed batch processing frameworks that handle an entire workflow as one job.
- Directed Acyclic Graph (DAG) processing:
  - Data can (often) be exchanged via local resources of one machine (shared memory buffer or local disk).
    - ➢ Less data exchange via network/HDFS
  - Operators start processing as soon as their input is ready.
    - ➢ No waiting for entire processing stages to finish
  - Existing JVM processes are reused to run further operators.
    - ➢ No process launches for every task



Pig/Hive - MR



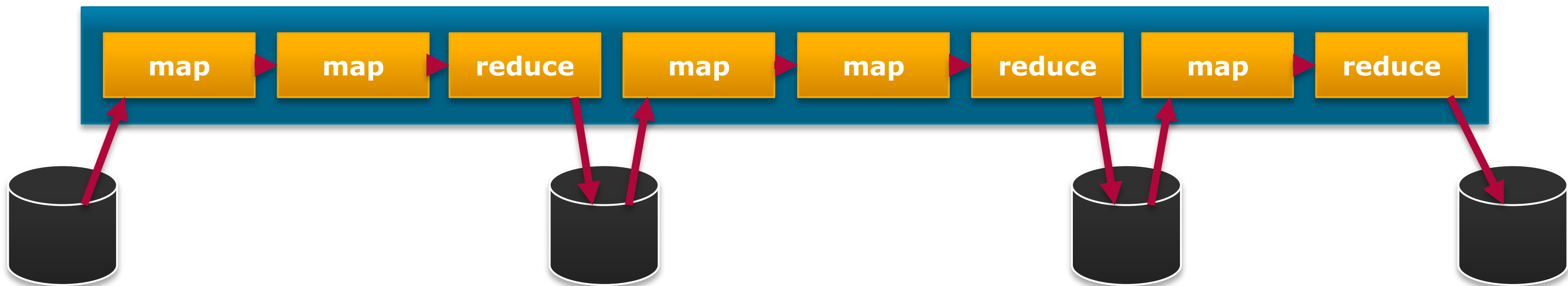
Pig/Hive - Tez

Dryad and Nephele (Stratosphere) were the first research systems that inspired most modern data flow engines

Data Flow Engines

Fault tolerance

- Checkpointing:
  - Some intermediate results are written to disk.
  - If operators crash, pipeline re-computes affected partitions from nearest checkpoint.



- Re-computation:
  - Requires that computations are deterministic.
  - Using randomness, current time, external resources, set orders, … breaks determinism.

Data Flow Engines

Hasso Plattner Institut

Joins (and Groupings)

Most data flow engines use a cost-based query optimizer to automatically choose a join-strategy and join-order.

- Supported options for connecting outputs to inputs:
  a) Broadcast:
     - Send the records of one relation to all partitions.
     - ➢ broadcast hash join

  b) Repartition without sorting:
     - Partition the records of two relations by key.
     - Skip the sorting (saving its overhead).
     - ➢ hash join

  c) Repartition with sorting:
     - Partitions the records of two relations by key and sorts each partition (like in MapReduce's shuffle phase).
     - ➢ sort-merge join

Very important for non-blocking, streaming pipelines

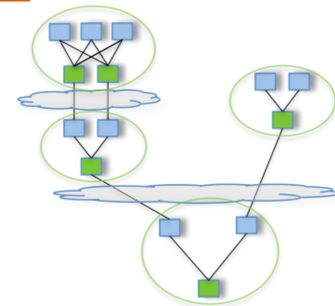
Data Flow Engines

Hasso Plattner Institut

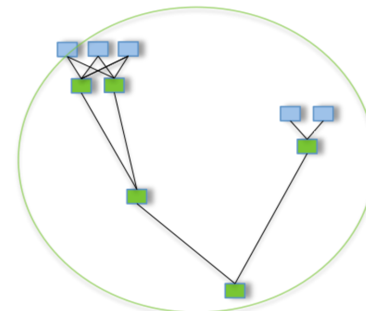
Data Flow Engine

- Most popular implementations:
  - Tez: https://tez.apache.org/
    Bikas Saha, Hitesh Shah, Siddharth Seth, et al.: "**Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications**," at *ACM International Conference on Management of Data (SIGMOD)*, June 2015.
  - Spark: https://spark.apache.org/
    Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, et al.: "**Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing**," at *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2012.
  - Flink: https://flink.apache.org/
    Alexander Alexandrov, Rico Bergmann, Stephan Ewen, et al.: "**The Stratosphere Platform for Big Data Analytics**," *The VLDB Journal*, volume 23, number 6, pages 939–964, May 2014.

Most query engines (e.g. Hive and Pig) can use these instead of MapReduce.



Pig/Hive - MR



Pig/Hive - Tez

Apache Tez

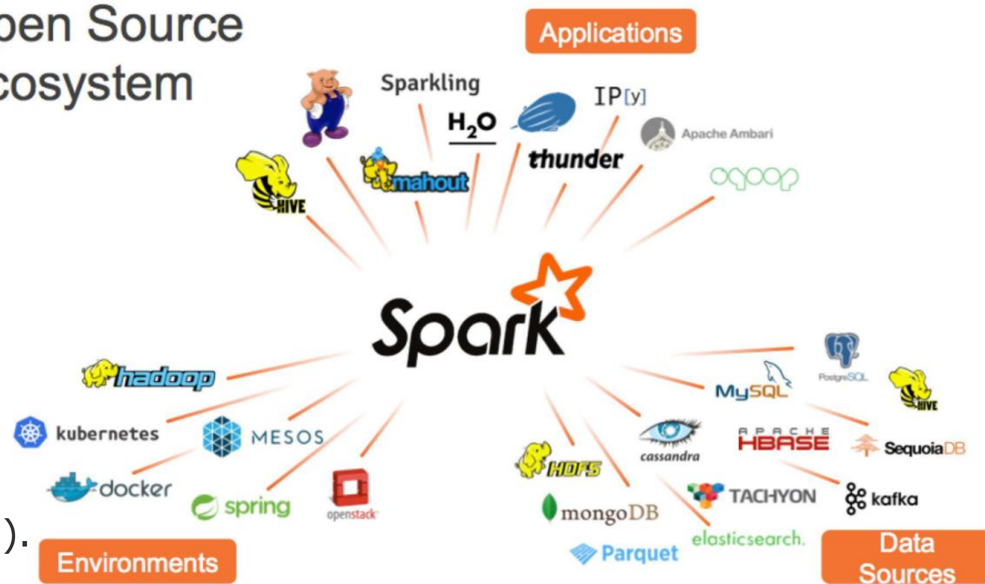
Tez

- A comparatively thin library tightly coupled to YARN
- Designed for …
  - high performance batch applications and
  - interactive data processing applications.
- Clearly improves upon MapReduce's efficiency
- Less mature than Spark and Flink:
  - Less concise and more difficult API
  - No streaming or machine learning support
  - Slower

Apache Spark

Spark

- Large framework with own network communication layer, scheduler, and user-facing APIs
  - Can also use YARN & Mesos
- Achievements:
  - Has won the 100 Terabyte sort competition with 206 nodes in 23 min (MapReduce: 2100 nodes in 72 min).
  - Has performed petabyte sort, for which no competition exists.



Spark – The Word Count Example

```
// Spark wordcount
object WordCount {
  def main(args: Array[String]) {
    val env = new SparkContext("local","wordCount")
    val data = List("hi","how are you","hi")
    val dataSet = env.parallelize(data)
    val words = dataSet.flatMap(value => value.split("\\s+"))
    val mappedWords = words.map(value => (value,1))
    val sum = mappedWords.reduceByKey(_+_)
    println(sum.collect())
  }
}
```

Connect to the local Spark cluster with a new job called wordCount

Pipeline construction

Single point of execution!
(workflows are lazily executed)

Apache Spark

Spark

Spark – The Word Count Example

```
// Spark wordcount
object WordCount {
 def main(args: Array[String]) {
  val env = new SparkContext("local","wordCount")
  val data = List("hi","how are you","hi")
  val dataSet = env.parallelize(data)
  val words = dataSet.flatMap(value => value.split("\\s+"))
  val mappedWords = words.map(value => (value,1))
  val sum = mappedWords.reduceByKey(_+_)
  println(sum.collect())
 }
}
```

RDDs store the ancestry of data:
input partition and operator chain

Resilient Distributed Datasets (RDDs)

- An immutable distributed collection of objects (with arbitrary types: native and user-defined)
- Fundamental data structure of Spark
- Fault-tolerant:
  - Can be persisted on disk
  - Can be reconstructed by re-executing parts of the pipeline
- By default: hold in memory
- Write to disk …
  - if memory is exhausted
  - for checkpointing
  - before network shuffle

Spark

Spark – The Word Count Example

```
// Spark wordcount
object WordCount {
  def main(args: Array[String]) {
    val env = new SparkContext("local","wordCount")
    val data = List("hi","how are you","hi")
    val sum = env.parallelize(data)
                 .flatMap(value => value.split("\\s+"))
                 .map(value => (value,1))
                 .reduceByKey(_+_)
    println(sum.collect())
  }
}
```

Method chaining

Spark

HPI Hasso Plattner Institut

Spark – The Word Count Example

```
// Spark wordcount
object WordCount {
  def main(args: Array[String]) {
    val env = new SparkContext("local","wordCount")
    val data = List("hi","how are you","hi")
    val sum = env.parallelize(data)
                 .flatMap(value => value.split("\\s+"))
                 .map(value => (value,1))
                 .reduceByKey(_+_)
    println(sum.collect())
  }
}
```

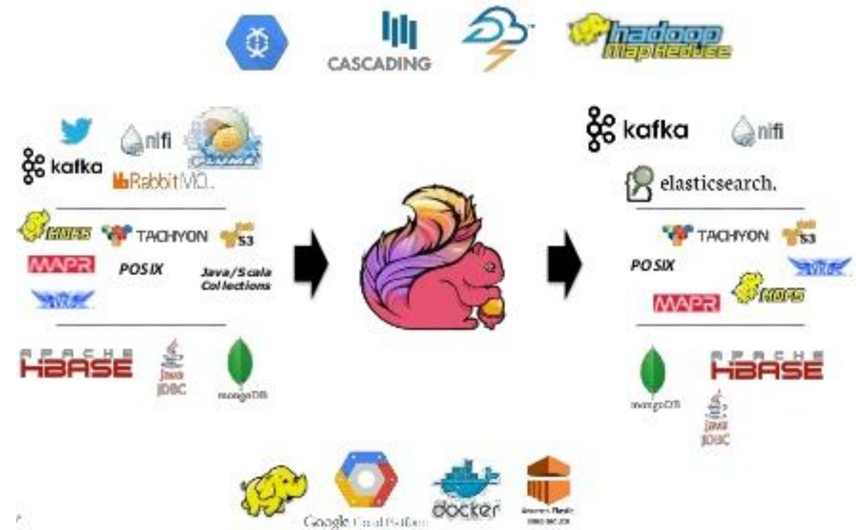
An RDD can be created by …

- parallelizing a local collection or
- referencing a dataset in external storage (HDFS, HBase, Cassandra, MongoDB …)

Apache Flink

Flink

- Large framework with own network communication layer, scheduler, and user-facing APIs
- Designed for stream processing but with batch processing capabilities
- Originated from the Stratosphere project (DFG project: TU Berlin, HU Berlin, and HPI)
- Developed by Ververica (former Data Artisans, which was acquired by Alibaba for €90 million)
- Official publication with 445 citations (as of 2019):
  A. Alexandrov, R. Bergmann, S. Ewen, J. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A Rheinländer, M. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke, **"The Stratosphere Platform for Big Data Analytics"**, *VLDBJ, 2014.*



Flink

HPI Hasso Plattner Institut

Flink – The Word Count Example

```
// Flink wordcount
object WordCount {
 def main(args: Array[String]) {
  val env = ExecutionEnvironment.getExecutionEnvironment
  val data = List("hi","how are you","hi")
  val dataSet = env.fromCollection(data)
  val words = dataSet.flatMap(value => value.split("\\s+"))
  val mappedWords = words.map(value => (value,1))
  val grouped = mappedWords.groupBy(0)
  val sum = grouped.sum(1)
  println(sum.collect())
 }
}
```

Connect to the local Flink cluster via ExecutionEnvironment

Pipeline construction

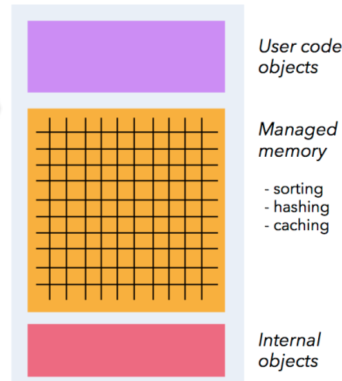
Single point of execution!
(workflows are lazily executed)

Apache Spark vs. Apache Flink

Large frameworks with own network communication layer, scheduler, and user-facing APIs

Both implement custom memory management:

- Own memory management inside the JVM
- Automatic disk spilling if jobs exceed RAM capacity
- Reduced garbage collection overhead



Both support batch and stream processing

Both offer interactive, graph, SQL and machine learning libraries

Spark vs. Flink


```
// Spark wordcount
object WordCount {
  def main(args: Array[String]) {
    val env = new SparkContext("local","wordCount")
    val data = List("hi","how are you","hi")
    val dataSet = env.parallelize(data)
    val words = dataSet.flatMap(value => value.split("\\s+"))
    val mappedWords = words.map(value => (value,1))
    val sum = mappedWords.reduceByKey(_+_)

    println(sum.collect())
  }
}
```

```
// Flink wordcount
object WordCount {
  def main(args: Array[String]) {
    val env = ExecutionEnvironment.getExecutionEnvironment
    val data = List("hi","how are you","hi")
    val dataSet = env.fromCollection(data)
    val words = dataSet.flatMap(value => value.split("\\s+"))
    val mappedWords = words.map(value => (value,1))
    val grouped = mappedWords.groupBy(0)
    val sum = grouped.sum(1)
    println(sum.collect())
  }
}
```

Both Spark and Flink mimic **Scala/Java collection API's**

Apache Spark vs. Apache Flink

Large frameworks with own network communication layer, scheduler, and user-facing APIs

A batch processing framework that emulates stream processing

stream processing = faster batch processing

A stream processing framework that emulates batch processing

batch processing = bounded stream processing

Apache Spark vs. Apache Flink

Hasso Plattner Institut

Pull shuffling model:
Operators pull their input from their predecessors (copy via network) and prepare their output to be pulled when it is ready.

Push shuffling model:
Operators push their outputs to subsequent operators (copy via network) and act on inputs that they receive.

True streaming:
Data elements are immediately "pipelined" through the DAG.

Spark Streaming

- Mini-batch Model:
  Cut event stream into small batches; process each batch in sequence.
- Latency: seconds to minutes

Flink Streaming

- Streaming Model:
  Handle event stream directly; process each event immediately when it arrives.
- Latency: milliseconds to seconds

https://www.linkedin.com/pulse/spark-streaming-vs-flink-bo-yang

Apache Spark

Hasso Plattner Institut

But: native streaming support with DataFrames in development

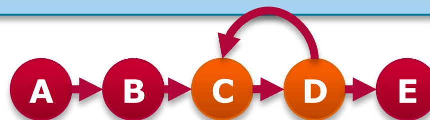
RDDs and DStreams:
RDDs are Java objects;
DStreams are RDDs under the hood;
mixing batches & streams in one job is easy.

Dataset and DataStream:
Datasets are logical plans (optimizable);
Datasets and DataStreams are different;
mixing batches & streams is less clear.

Spark vs Flink

Unfolded iterations :
Spark programs implement iterations by repeatedly appending the iterative part to the DAG, i.e., the loop is fixed into the pipeline.

Controlled iterations:
Flink offers controlled cyclic dependency graphs, i.e., real cycles in their pipeline graphs.



http://blog.madhukaraphatak.com/introduction-to-flink-for-spark-developers-flink-vs-spark/

Apache Spark vs. Apache Flink

No upgrade/resume support:
Spark's checkpoints are tied to specific DAG; changing or resuming a DAG requires a re-start of the entire pipeline.

Upgrade/resume support:
Flink’s checkpoints are not tied to a DAG; when changing or resuming a DAG, Flink can probably use previous checkpoints.

Spark vs Flink

Spark state storage

- Saves state in RDDs, which are periodically checkpointed to disk
- Users cannot choose another checkpointing strategy
(but they can **rdd.persist()** manually)

Flink state storage

- Offers three pluggable state storage options:
  - MemoryStateBackend
  - FsStateBackend
  - RocksDBStateBackend

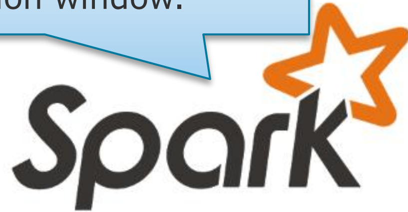
https://www.linkedin.com/pulse/spark-streaming-vs-flink-bo-yang

Apache Spark vs. Apache Flink

HPI Hasso Plattner Institut

Basic window support:
e.g. supports sliding window,
but not session window.

Advanced windowing support:
sliding windows, session windows, custom
event eviction …



Flink

More mature:

- large community
- widely used and tested
- proved to work well in terms of scalability and fault tolerance
- superior SQL and data source support

Less mature:

- smaller community
- younger than Spark
- may still need some work regarding fault tolerance and edge cases
- basic SQL and data source support

https://www.linkedin.com/pulse/spark-streaming-vs-flink-bo-yang

Apache Spark vs. Apache Flink

HPI Hasso Plattner Institut

Choose…

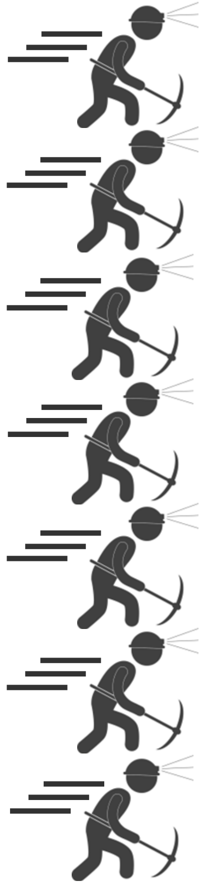
- **for batch processing**
- **if reliability matters most**

Choose…

- **for stream processing**
- **if real-time latencies matter most**

https://www.linkedin.com/pulse/spark-streaming-vs-flink-bo-yang

Graph Processing as Batch Job



Graph Processing as Batch Job

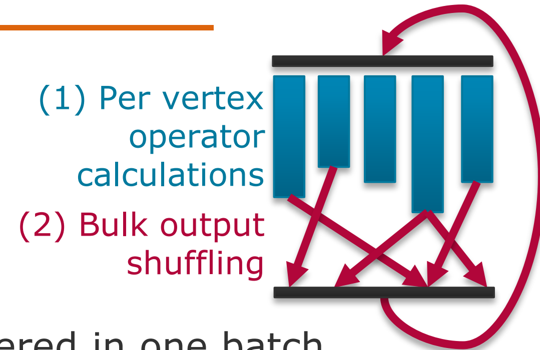
OLAP: point vs. batch queries

- Point query: Perform some action or extraction on a small sub-graph.
- Batch query: Perform some processing or analysis on the entire graph.

Examples

- Cliques search (all subgraphs, in which all vertices are adjacent to each other)
- Minimum spanning tree search (a minimal set of nodes that connects to all other nodes)
- Closure calculation (for each path of length two, the transitive edge)
- Page rank calculation (for each node, the popularity of this node)
- All pairs shortest path search (for each node, the shortest path to all other nodes)
  - ➢ Usually iterative algorithms

Graph Processing as Batch Job



Intuition

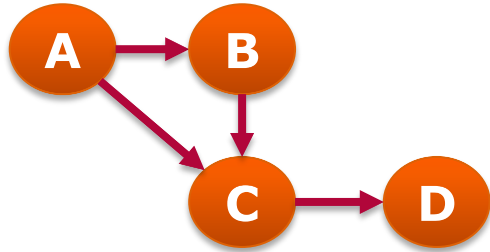
- Think of each graph node as an actor with state and mailbox.
- Think in iterations:
  1. Nodes send sets of messages to other nodes.
  2. All messages are collected, grouped by receiver, and delivered in one batch.
     - 1./2. are separated by the grouping operator (**synchronization barrier**)
     - 1./2. repeat until batch job is done

➢ Bulk Synchronous Parallel (BSP) model:

- Formalizes this intuition of Graph Batch Jobs
- Difference to a simple, iterating MapReduce job:
  - Nodes hold state that carries over from one iteration to the next.
- Implementations:
  - **Pregel**, Apache **Giraph**, Spark's **GraphX** API, Flink's **Gelly** API, …

All Pairs Shortest Path

Example Graph



Optimization?

End iteration, because nothing changed

A→0 | B→0 | C→0 | D→0

B→1 | C→1 | D→1

A→0 B→1 C→1 | B→0 C→1 | C→0 D→1 | D→0

B→1 C→2 | C→1 D→2 | D→1

A→0 B→1 C→1 D→2 | B→0 C→1 D→2 | C→0 D→1 | D→0

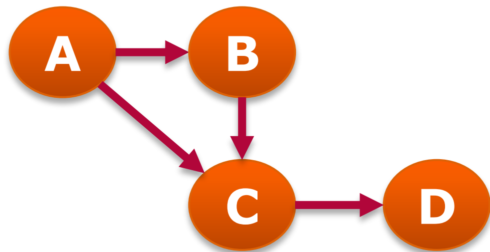
B→1 C→2 D→3 | C→1 D→2 | D→1

A→0 B→1 C→1 D→2 | B→0 C→1 D→2 | C→0 D→1 | D→0

Send local paths list +1 to every incoming node

All Pairs Shortest Path

Example Graph



A→0	B→0	C→0	D→0

B→1 C→1 D→1

A→0 B→1 C→1	B→0 C→1	C→0 D→1	D→0

C→2 D→2

A→0 B→1 C→1 D→2	B→0 C→1 D→2	C→0 D→1	D→0

D→3

A→0 B→1 C→1 D→2	B→0 C→1 D→2	C→0 D→1	D→0

Send changes to local paths +1 to every incoming node

End iteration, because nothing was send

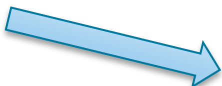
Awesome - Repository of curated References

Awesome Awesome

A curated list of awesome curated lists of many topics.

https://github.com/emijrp/awesome-awesome

- Awesome Awesome
  - Computer management
  - Data processing
  - Editors
  - Front-end development
  - Programming languages
  - Sciences
  - Web browsers
  - Websites
  - Web platforms
  - Other



Data processing

- awesome-bigdata - Big data frameworks, resources and other awesomeness.
- awesome-hadoop - Hadoop ecosystem resources.

Awesome - Repository of curated References

Hasso Plattner Institut

Awesome Big Data

https://github.com/onurakpolat/awesome-bigdata

A curated list of awesome big data frameworks, resources and other awesomeness. Inspired by awesome-php, awesome-python, awesome-ruby, hadoopecosystemtable & big-data.

- Awesome Big Data
  - RDBMS
  - Frameworks
  - Distributed Programming
  - Distributed Filesystem
  - Key-Map Data Model
  - Document Data Model
  - Key-value Data Model
  - Graph Data Model
  - NewSQL Databases
  - Columnar Databases
  - Time-Series Databases
  - SQL-like processing
  - Data Ingestion
  - Service Programming
  - Scheduling
  - Machine Learning
  - Benchmarking
  - Security
  - System Deployment
  - Applications
  - Search engine and framework
  - MySQL forks and evolutions
  - PostgreSQL forks and evolutions
  - Memcached forks and evolutions
  - Embedded Databases
  - Business Intelligence
  - Data Visualization
  - Internet of things and sensor data
  - Interesting Readings
  - Interesting Papers

Awesome - Repository of curated References

Distributed Programming

- AddThis Hydra - distributed data processing and storage system originally developed at AddThis.
- AMPLab SIMR - run **Spark** on **Hadoop MapReduce** v1.
- Apache APEX - a unified, enterprise platform for big data stream and batch processing.
- Apache Beam - an unified model and set of language-specific SDKs for defining and executing data processing workflows.
- Apache Crunch - a simple Java API for tasks like joining and data aggregation that are tedious to implement on plain **MapReduce**.
- Apache DataFu - collection of user-defined functions for **Hadoop** and **Pig** developed by LinkedIn.
- **Apache Flink** - high-performance runtime, and automatic program optimization.
- Apache Gearpump - real-time big data streaming engine based on **Akka**.
- Apache Gora - framework for in-memory data model and persistence.
- Apache Hama - BSP (Bulk Synchronous Parallel) computing framework.
- **Apache MapReduce** - programming model for processing large data sets with a parallel, distributed algorithm on a cluster.
- **Apache Pig** - high level language to express data analysis programs for **Hadoop**.
- Apache REEF - retainable evaluator execution framework to simplify and unify the lower layers of big data systems.
- Apache S4 - framework for stream processing, implementation of S4.
- **Apache Spark** - framework for in-memory cluster computing.
- **Apache Spark Streaming** - framework for stream processing, part of **Spark**.

Awesome - Repository of curated References

Distributed Programming (cont.)

- Apache Storm - framework for stream processing by Twitter also on **YARN**.
- Apache Samza - stream processing framework, based on Kafka and **YARN**.
- **Apache Tez** - application framework for executing a complex DAG (directed acyclic graph) of tasks, built on **YARN**.
- Apache Twill - abstraction over **YARN** that reduces the complexity of developing distributed applications.
- Cascalog - data processing and querying library.
- Cheetah - High Performance, Custom Data Warehouse on Top of **MapReduce**.
- Concurrent Cascading - framework for data management/analytics on **Hadoop**.
- Damballa Parkour - **MapReduce** library for Clojure.
- Datasalt Pangool - alternative MapReduce paradigm.
- DataTorrent StrAM - real-time engine to enable distributed, asynchronous, real time in-memory big-data computations in as unblocked a way as possible, with minimal overhead and impact on performance.
- Facebook Corona - **Hadoop** enhancement which removes single point of failure.
- Facebook Peregrine - **MapReduce** framework.
- Facebook Scuba - distributed in-memory datastore.
- Google Dataflow - create data pipelines to help themæingest, transform and analyze data.
- Google MapReduce - **MapReduce** framework.

Beyond MapReduce

Awesome - Repository of curated References

Hasso Plattner Institut

Distributed Programming (cont.)

- Google MillWheel - fault tolerant stream processing framework.
- IBM Streams - platform for distributed processing and real-time analytics. Provides toolkits for advanced analytics like geospatial, time series, etc. out of the box.
- JAQL - declarative programming language for working with structured, semi-structured and unstructured data.
- Kite - a set of libraries, tools, examples, and documentation focused on making it easier to build systems in the **Hadoop** ecosystem.
- Metamarkets Druid - framework for real-time analysis of large datasets.
- Netflix PigPen - **MapReduce** for Clojure which compiles to **Apache Pig**.
- Nokia Disco - **MapReduce** framework developed by Nokia.
- Onyx - distributed computation for the cloud.
- Pinterest Pinlater - asynchronous job execution system.
- Pydoop - Python **MapReduce** and **HDFS** API for **Hadoop**.
- Rackerlabs Blueflood - multi-tenant distributed metric processing system
- Skale - high performance distributed data processing in NodeJS.
- Stratosphere - general purpose cluster computing framework.
- Streamdrill - useful for counting activities of event streams over different time windows and finding the most active one.
- streamsx.topology - Libraries to enable building IBM Streams application in Java, Python or Scala.

Awesome - Repository of curated References

Hasso Plattner Institut

Distributed Programming (cont.)

- Tuktu - Easy-to-use platform for batch and streaming computation, built using Scala, **Akka** and Play!
- Twitter Heron - Heron is a realtime, distributed, fault-tolerant stream processing engine from Twitter replacing Storm.
- Twitter Scalding - Scala library for **MapReduce** jobs, built on Cascading.
- Twitter Summingbird - Streaming **MapReduce** with Scalding and Storm, by Twitter.
- Twitter TSAR - TimeSeries AggregatoR by Twitter.
- Wallaroo - ultrafast and elastic data processing engine. Big or fast data - no fuss, no Java needed.

Frameworks

- IBM Streams - platform for distributed processing and real-time analytics. Integrates with many of the popular technologies in the Big Data ecosystem (Kafka, **HDFS**, **Spark**, etc.)
- **Apache Hadoop** - framework for distributed processing including **MapReduce** (parallel processing), **YARN** (job scheduling) and **HDFS** (distributed file system).
- Tigon - high Throughput Real-time Stream Processing Framework.
- Pachyderm - data storage platform built on Docker and Kubernetes for reproducible data processing and analysis.

Mahout
DELTA
MACHINE
OF
LEARNING
To Stream Processing
(Chapter 11)
Terrapin
Voldemort
RECOMMENDATION
ENGINES
Flink
To Kingdom of Analytics
(Chapter 3) and
Data Integration
(Chapter 12)
Spark
SYSTEMS
DATAFLOW
RIVER OF
BATCH OUTPUTS
PROCESSING
Hive
MPP
Tez
DISTRIBUTED FILESYSTEMS
Impala
DATABASES
SEARCH INDEXES
GraphX
ITERATIVE
Giraph
Solr
MapReduce
workflows
ETL
Harbor
SINGLE-NODE
PROCESSING
GraphChi
Unix tools
SEA
Chapter 10. Batch Processing
Chapter 12. The Futu
Spreadsheets
Reactiv
prograr