



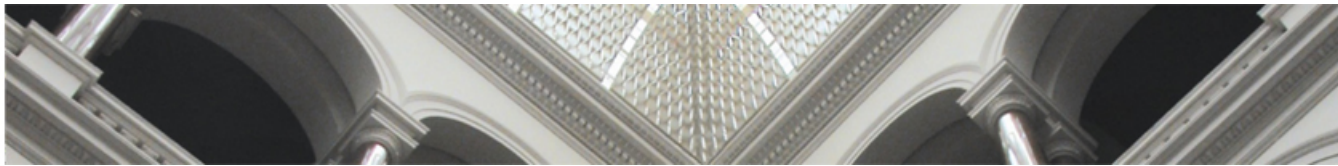
Abstractions for Massively Parallel Dataflow Processing

Sebastian Schelter | Database Group, TU Berlin | Scalable Data Science: Systems and Methods



Overview

- A little bit of history: From Relational Databases to Massively Parallel Dataflow Processing
- Distributed Shared-Nothing Filesystems
- Abstractions for Massively Parallel Dataflow Processing
 - MapReduce
 - Parallelization Contracts & Iterative Dataflows
 - Resilient Distributed Datasets
- Summary



Overview

- **A little bit of history: From Relational Databases to Massively Parallel Dataflow Processing**
- Distributed Shared-Nothing Filesystems
- Abstractions for Massively Parallel Dataflow Processing
 - MapReduce
 - Parallelization Contracts & Iterative Dataflows
 - Resilient Distributed Datasets
- Summary



From Relational Databases to Massively Parallel Dataflow Systems

- **relational model** (Codd 1970) gives rise to relational database management systems
- large data processing challenges in enterprise data management
 - enterprises collect historical business data in **data warehouses**
 - **reporting and business analytics** on this data
- scalability issues lead to development of **parallel database systems** in the mid 1980s
 - **shared-nothing architecture**: autonomous machines that only communicate over the network via message passing
 - introduction of '**divide-and-conquer**' **parallelism** based on hash-partitioning the data for storage and relational query processing
 - **commercial adoption** of these systems in the mid 1990s
 - database community considered **parallel query processing solved**



From Relational Databases to Massively Parallel Dataflow Systems

- **rise of the world wide web** in the 1990s produces growing need to query and index the data available online
- search engine companies found **database technology neither well suited nor cost-effective**
 - ACID paradigm of relational data processing **mismatch for web search**:
 - mostly **read-only queries**
 - **high availability** much more important than consistency
 - **dirty, semi-structured web data** hard to fit into clearly defined relational schema
 - **new types of 'queries'** very different from traditional SQL-based data analysis, e.g.,
 - ranking of search results based on link structure of the web (**graph processing**)
 - personalized advertising (**machine learning**)



From Relational Databases to Massively Parallel Dataflow Systems

- **Google developed a new breed of storage and data processing systems**
 - aimed at cost-effective shared-nothing clusters built from commodity hardware
 - **Google File System (GFS)**: distributed, web-scale storage system
 - **MapReduce**: simple programming model and execution paradigm for parallel data processing
- Google's publications gave birth to **Apache Hadoop**, an open-source variant of these systems
- **huge ecosystem** evolved (Pig, Hive, Mahout, Jaql, Zookeeper, Hbase, ...)
- currently, **second generation of distributed data processing engines** coming up (Apache Spark, Apache Flink)



Overview

- A little bit of history: From Relational Databases to Massively Parallel Dataflow Processing
- **Distributed Shared-Nothing Filesystems**
- Abstractions for Massively Parallel Dataflow Processing
 - MapReduce
 - Parallelization Contracts & Iterative Dataflows
 - Resilient Distributed Datasets
- Discussion: Large-Scale Machine Learning on Dataflow Systems



Distributed Shared-Nothing Filesystems

- most modern massively parallel data processing engines work on large datasets stored in distributed filesystems modeled after **Google's File System (GFS)**
- GFS is a **scalable, shared-nothing filesystem** for distributed data-intensive applications
- design decisions and goals:
 - **high fault tolerance** in clusters of **hundreds and thousands of machines**, high failure rates
 - storage of **large, multi-gigabyte files**
 - write workload: **large sequential writes**, no random access
 - read workload: **large streaming reads**, few small random reads
 - favoring high bandwidth for bulk reads over low latency access to individual files

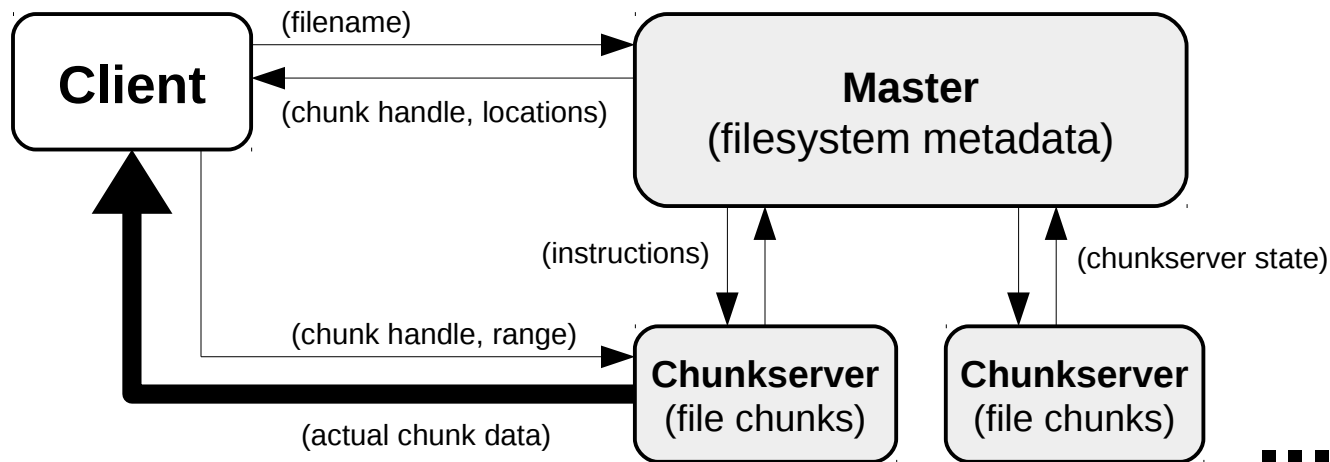


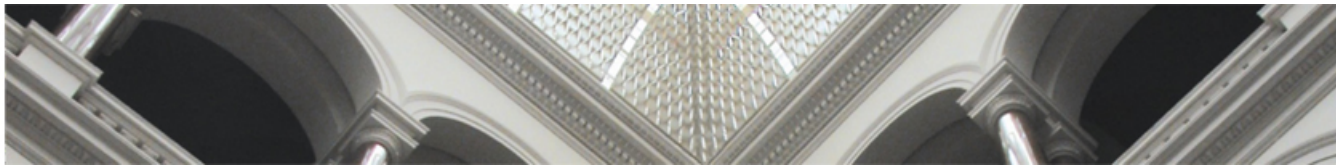
Distributed Shared-Nothing Filesystems

- GFS uses a **master-slave architecture**
- files are divided into **chunks** with a typical size of several dozen megabytes
- **master server**
 - orchestrates file operations
 - store the metadata of the filesystem
- **slaves (chunk servers)**
 - store replicas of the chunks on their local disks
- **fault tolerance**
 - master maintains replicated write-ahead log of critical metadata changes
 - master regularly sends heartbeat messages to chunk servers
 - master initiates re-replication of chunks in case of machine failures or data corruption

Distributed Shared-Nothing Filesystems

- clients communicate with master for metadata only (e.g., location of chunks)
- master redirects client to chunk servers
- client directly conducts read and write operations on chunk servers





Overview

- A little bit of history: From Relational Databases to Massively Parallel Dataflow Processing
- Distributed Shared-Nothing Filesystems
- **Abstractions for Massively Parallel Dataflow Processing**
 - **MapReduce**
 - Parallelization Contracts & Iterative Dataflows
 - Resilient Distributed Datasets
- Summary



MapReduce

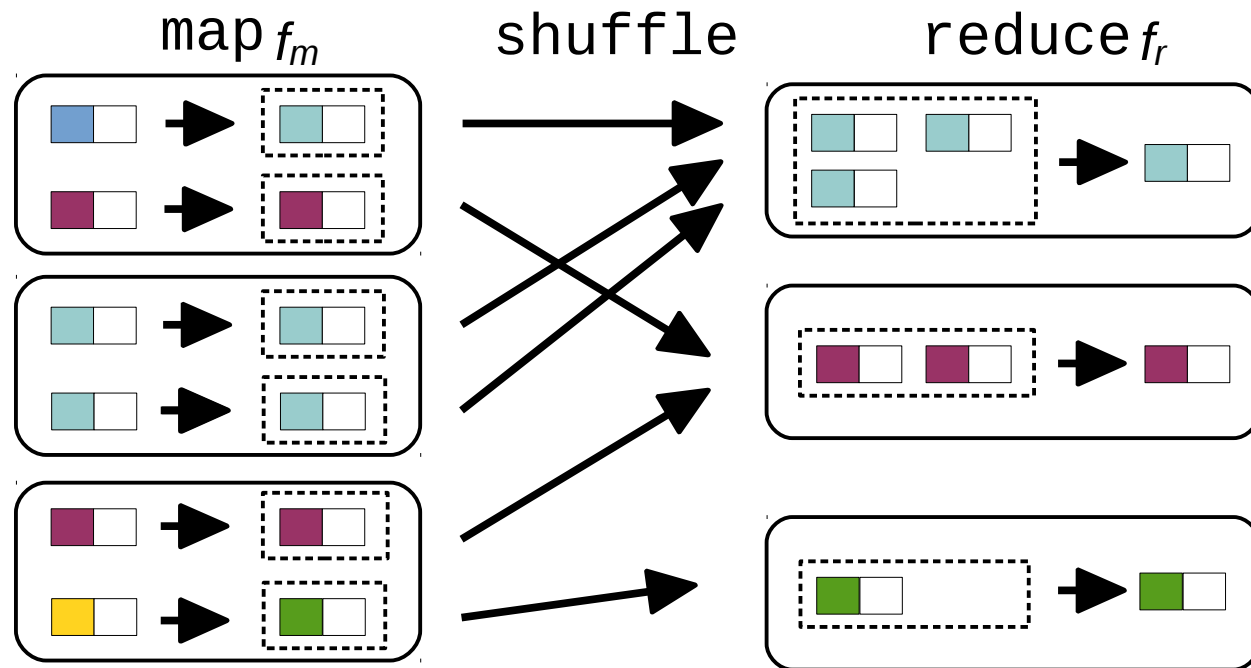
- programming model and paradigm for distributed data processing based on **two second-order functions map and reduce**
- program expressed by **two first order functions f_m and f_r** which operate on key-value pairs

$$f_m: (k_1, v_1) \rightarrow list(k_2, v_2)$$

$$f_r: (k_2, list(v_2)) \rightarrow list(k_2, v_2)$$

- execution in three phases
 - **map-phase:** system individually applies f_m to all input key-value pairs in parallel
 - **shuffle phase:** systems groups all key-value pairs emitted in the map-phase by key k_2
 - **reduce-phase:** system applies f_r to all groups in parallel

MapReduce





Wordcount Example

- **Task:** count the number of occurrences of every word in a large set of documents with MapReduce
- proxy for many workloads related to large search engines
(e.g. inverted index generation, calculation of tf-idf score)

```
function f_m (document):
```

```
    words = tokenize(document)
```

```
    for (word in words):
```

```
        emit(word, 1)
```

```
function f_r(word, counts):
```

```
    num_occurrences = sum(counts)
```

```
    emit(word, num_occurrences)
```




WordCount

"Hello World"

"Hello Galaxy"

"Hello Moon"

"Hello World"

WordCount: Map-Phase

$\text{map } f_m$

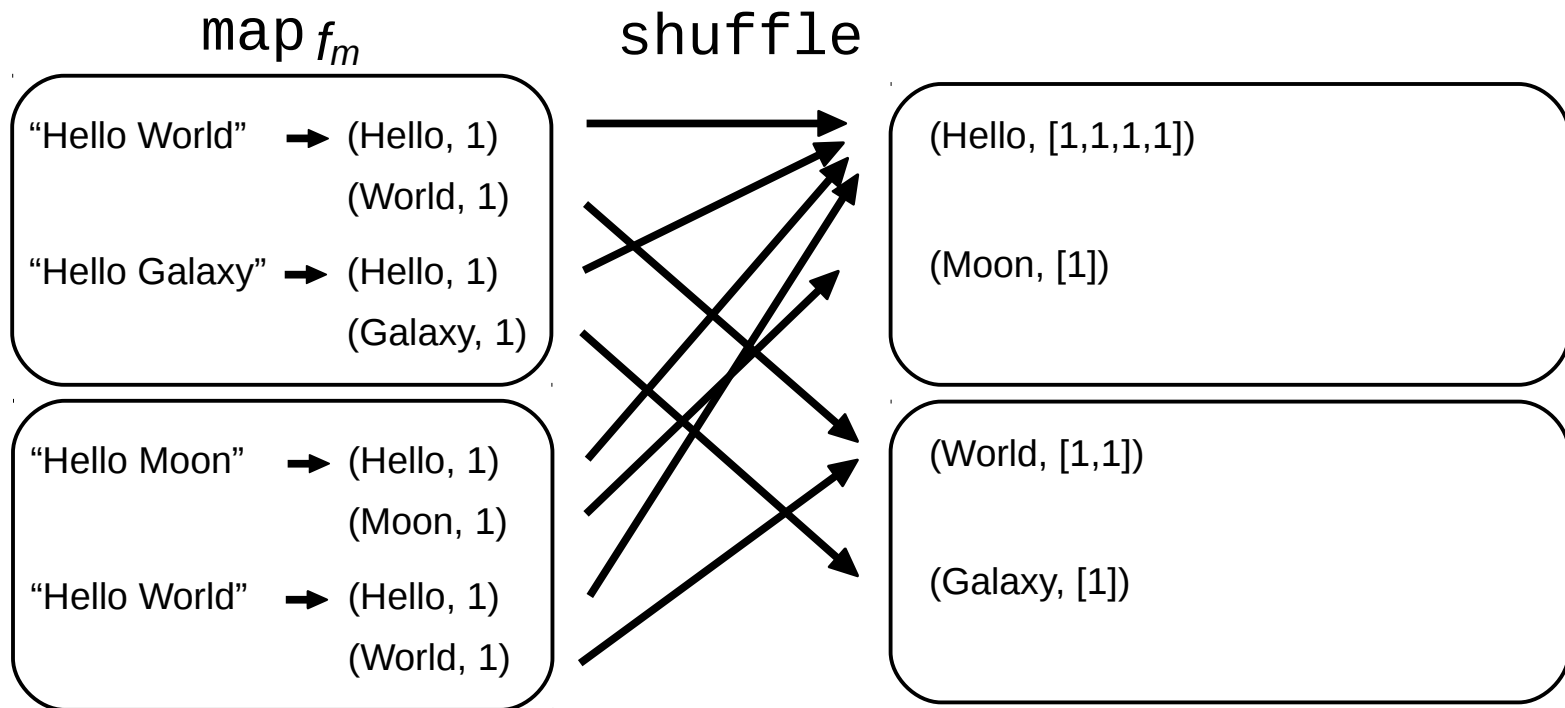
“Hello World” \rightarrow (Hello, 1)
(World, 1)

“Hello Galaxy” \rightarrow (Hello, 1)
(Galaxy, 1)

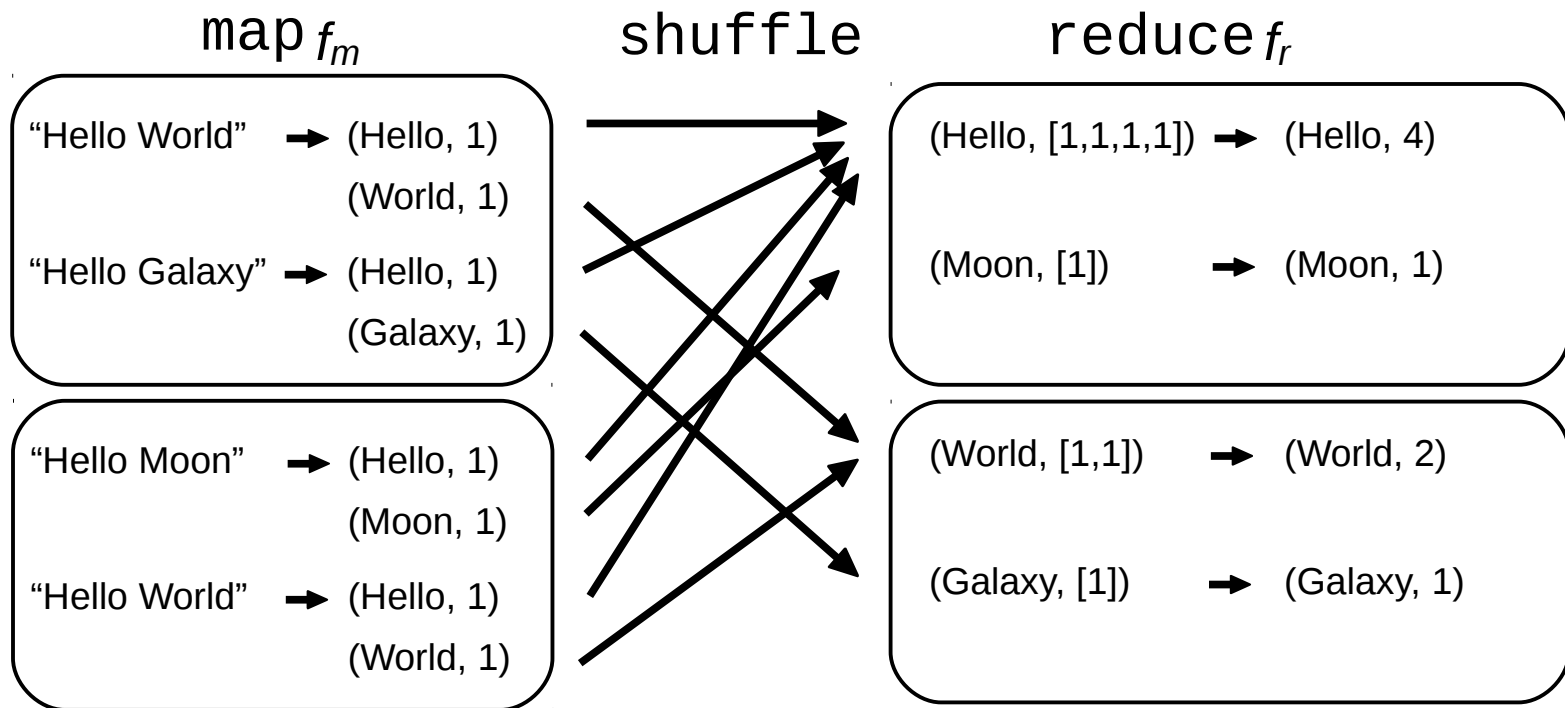
“Hello Moon” \rightarrow (Hello, 1)
(Moon, 1)

“Hello World” \rightarrow (Hello, 1)
(World, 1)

WordCount: Shuffle-Phase



WordCount: Reduce-Phase

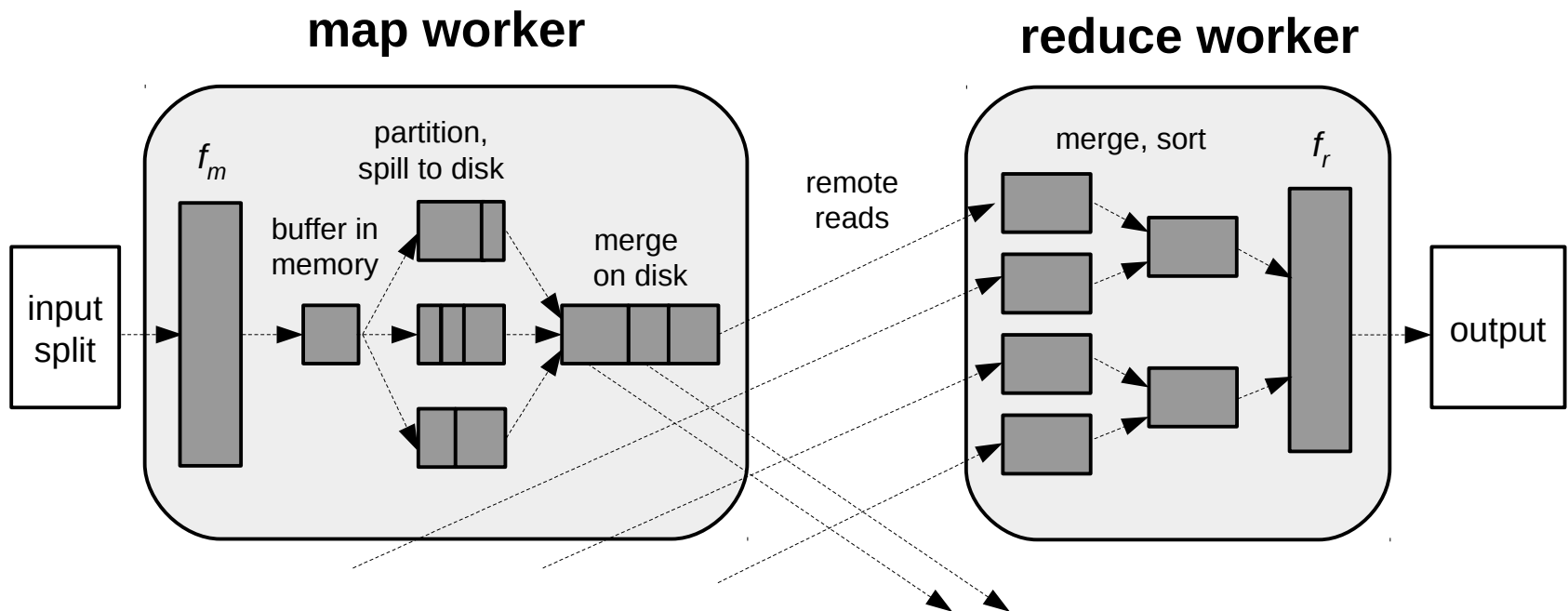




MapReduce

- system implemented with a **master-slave architecture** complementary to the architecture of the underlying distributed filesystem
- **master** orchestrates system
 - **schedules individual map and reduce tasks** on slaves
 - monitors progress and reacts to failures
 - tries to exploit **data locality**
- **slaves execute map and reduce tasks**
- extremely simple way to achieve **fault tolerance**
 - regular heartbeat to check slaves presence, **rescheduling of failed map and reduce tasks**
 - atomic commits for intermediate map and reduce outputs to avoid corruption

MapReduce





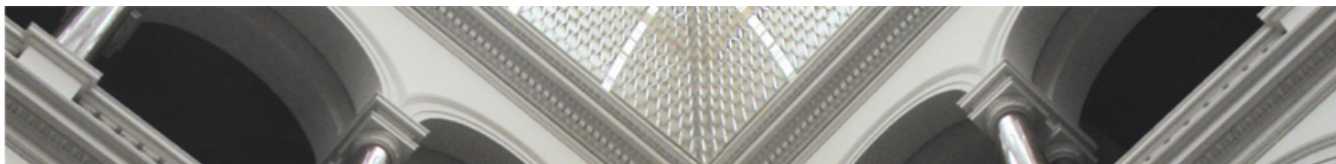
MapReduce

- drawbacks of MapReduce
 - **performance problems**
 - always uses **disk-backed execution**
 - **no global view of programs** consisting of many MR jobs (missing optimization potential)
 - **low performance for iterative computations** (e.g., no caching of loop-invariant data)
 - difficult to program
 - **no operators for combining multiple datasets** (e.g., joins)
 - very **low level interface**



Overview

- A little bit of history: From Relational Databases to Massively Parallel Dataflow Processing
- Distributed Shared-Nothing Filesystems
- Abstractions for Massively Parallel Dataflow Processing
 - MapReduce
 - **Parallelization Contracts & Iterative Dataflows**
 - Resilient Distributed Datasets
- Summary

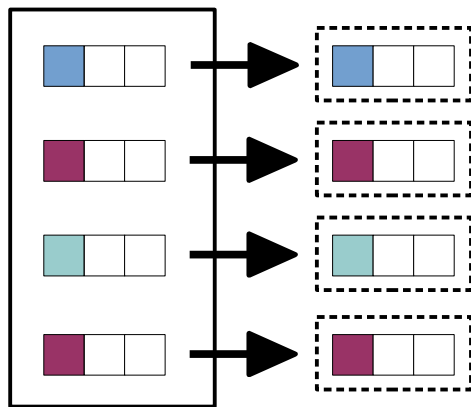


Parallelization Contracts (PACTs) and Iterative Dataflows

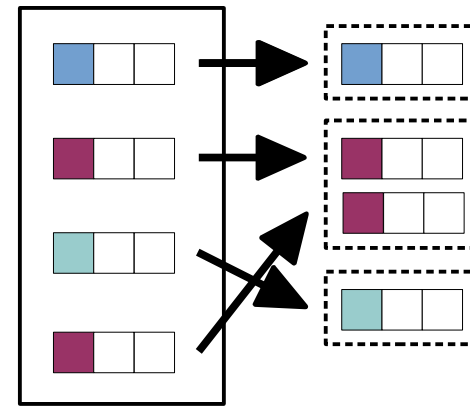
- core abstraction of the **distributed data processing system Apache Flink** (formerly Stratosphere)
- main differences to MapReduce
 - **wide variety of operators** with support for combining multiple datasets
 - **automatic optimization** of programs
 - dedicated **support for iterative computations**
- core operators (second-order functions, generalization of MapReduce)
 - **Map**
 - **Reduce**
 - **Cross** (cartesian product + UDF)
 - **Join** (equi join + UDF)
 - **CoGroup**



Parallelization Contracts (PACTs) and Iterative Dataflows

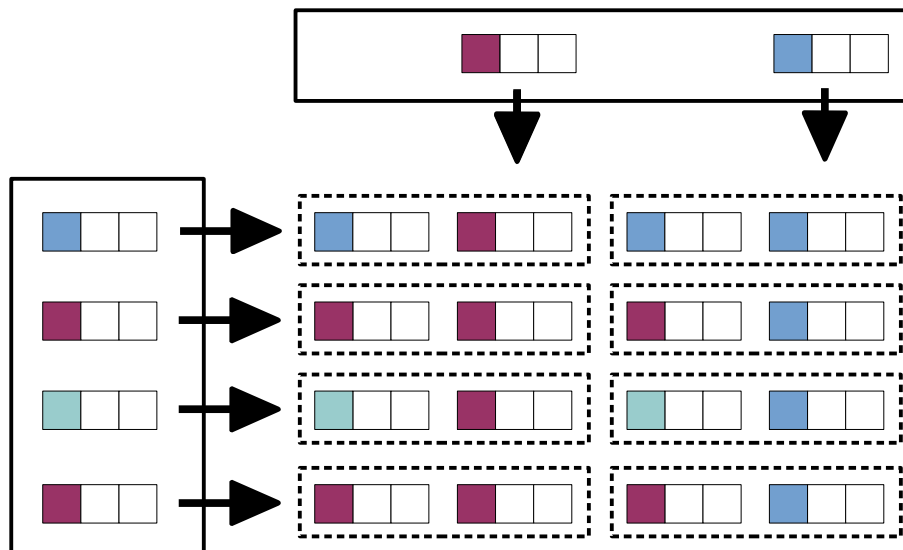


map



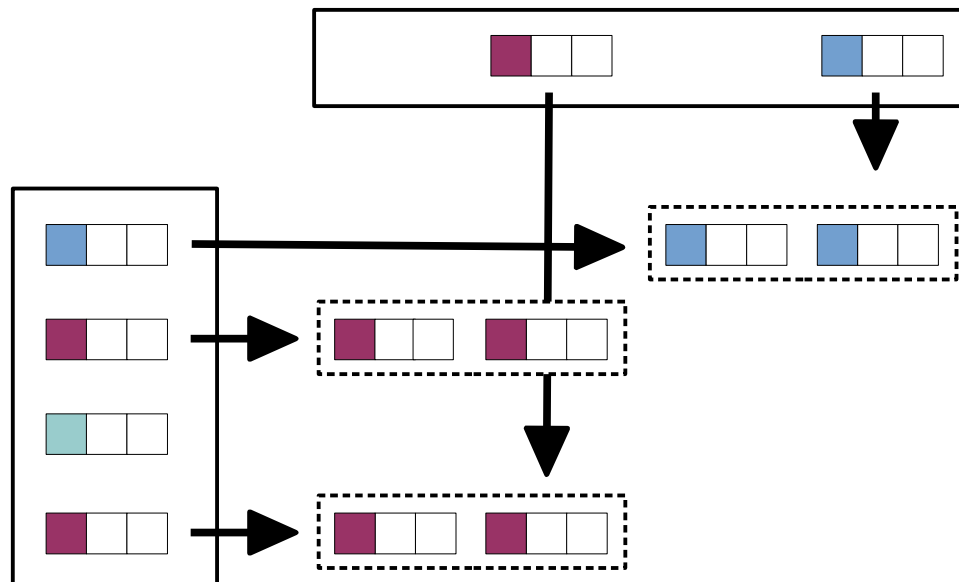
reduce

Parallelization Contracts (PACTs) and Iterative Dataflows

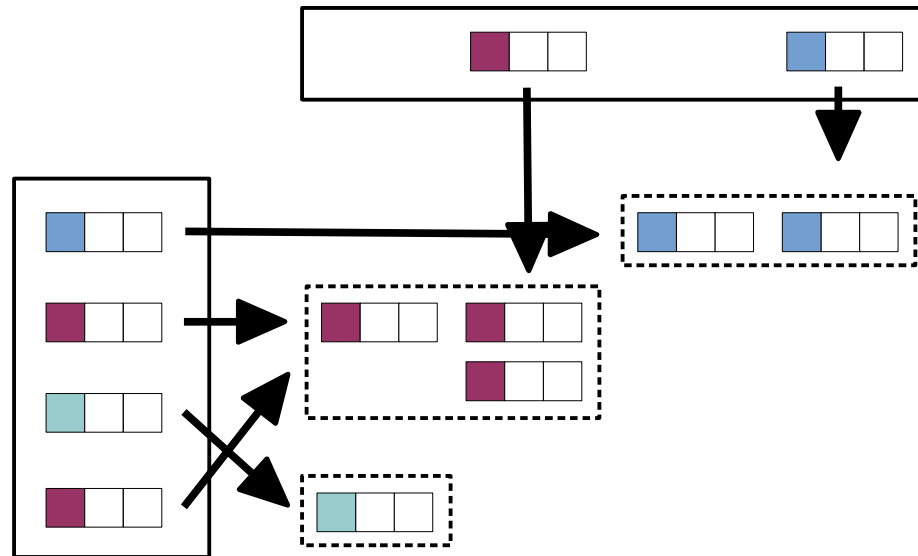


cross

Parallelization Contracts (PACTs) & Iterative Dataflows



Parallelization Contracts (PACTs) & Iterative Dataflows

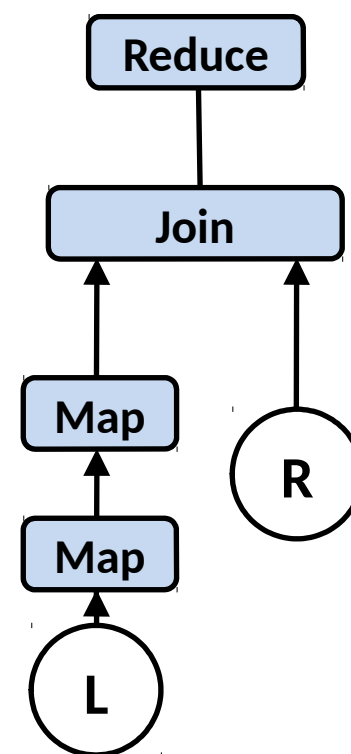


cogroup



Parallelization Contracts (PACTs) & Iterative Dataflows

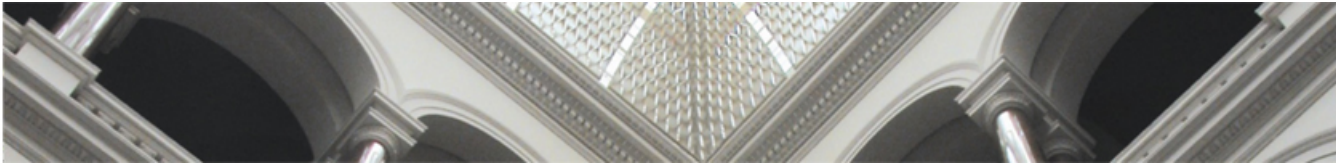
- a **PACT program consists of a directed acyclic graph (DAG)**
 - vertices: PACT operators and their corresponding UDFs
 - edges: represent exchange of data between operators
- **transformation of PACT program into a low level job graph** of the distributed processing engine Nephele
- **optimization** inspired by optimizers of parallel database systems
 - logical plan equivalences, cost models, interesting properties
- optimization much **more difficult than in relational setting**
 - non-fully specified semantics due to UDFs, hard to derive estimates for intermediate result sizes
 - no predefined schema present



Parallelization Contracts (PACTs) & Iterative Dataflows

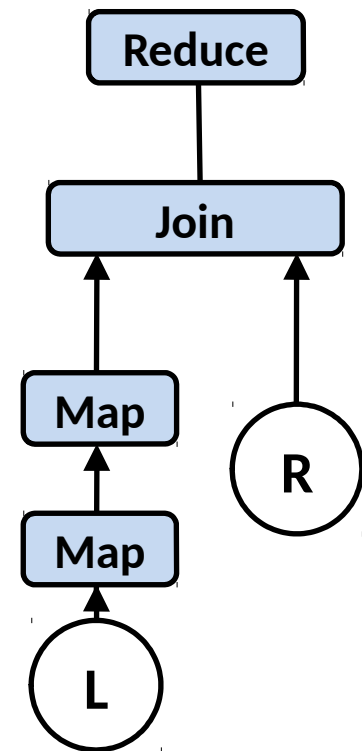
- WordCount in Flink's Scala API

```
val counts = corpus
  .flatMap { document => document.split("\\W+") }
  .map { term => (term, 1) }
  .groupByKey()
  .sum()
```



Parallelization Contracts (PACTs) & Iterative Dataflows

- optimization conducted in two phases
 - **logical plan rewriting**
 - generation of **equivalent plans by reordering operators**
 - conditioned on conflicting value accesses and preservation of group cardinalities
 - **physical optimization**
 - **cost-based approach** to pick strategies for **data shipping** and **local operator execution** (e.g., broadcast- or repartition-based data shipping, sort- or hash-based join execution)
 - optimizer keeps track of **interesting properties** such as sorting, grouping and partitioning



Parallelization Contracts (PACTs) & Iterative Dataflows

```
val orders = DataSource(...)
val items = DataSource(...)

val filtered = orders filter { ... }

val priced = filtered join items where { _.id } isEqualTo { _.id }
    map { (o, i) => PricedOrder(o.id, o.priority, i.price) }

val sales = priced groupBy { p => (p.id, p.priority) } aggregate ({ _.price}, SUM)
```

```
case class Order(id: Int, priority: Int, ...)
case class Item(id: Int, price: Double, ...)
case class PricedOrder(...)
```

Parallelization Contracts (PACTs) & Iterative Dataflows

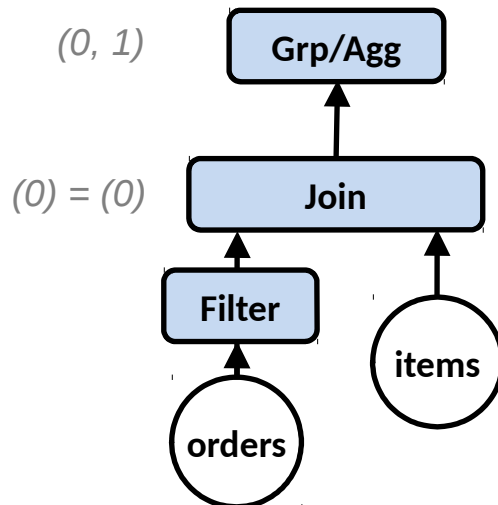
```
val orders = DataSource(...)
val items = DataSource(...)

val filtered = orders filter { ... }

val priced = filtered join items where { _.id } isEqualTo { _.id }
    map { (o, i) => PricedOrder(o.id, o.priority, i.price) }

val sales = priced groupBy { p => (p.id, p.priority) } aggregate ({ _.price}, SUM)
```

```
case class Order(id: Int, priority: Int, ...)
case class Item(id: Int, price: Double, ...)
case class PricedOrder(...)
```



Parallelization Contracts (PACTs) & Iterative Dataflows

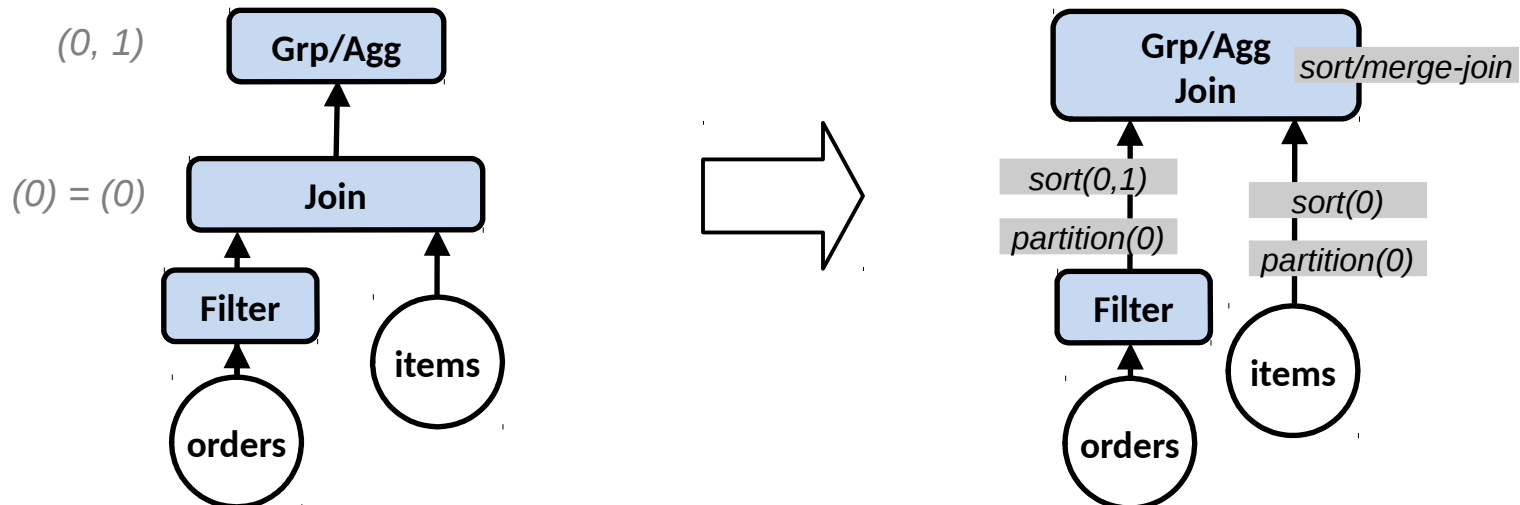
```
val orders = DataSource(...)
val items = DataSource(...)

val filtered = orders filter { ... }

val priced = filtered join items where { _.id } isEqualTo { _.id }
    map { (o, i) => PricedOrder(o.id, o.priority, i.price) }

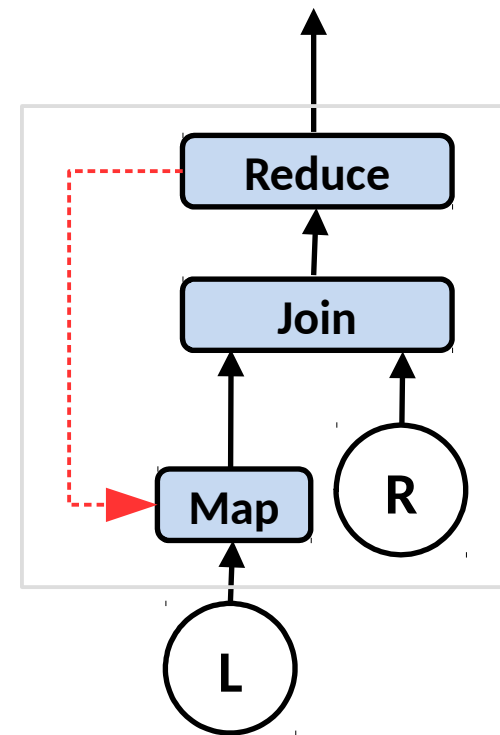
val sales = priced groupBy { p => (p.id, p.priority) } aggregate ({ _.price}, SUM)
```

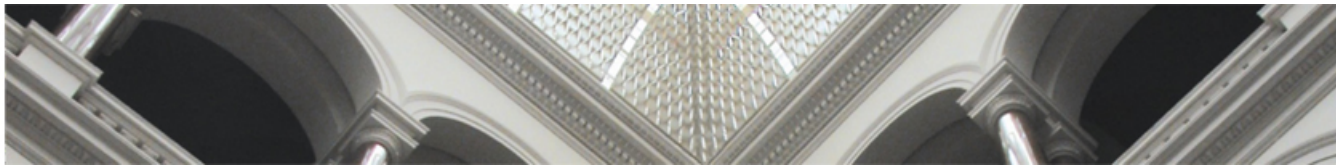
```
case class Order(id: Int, priority: Int, ...)
case class Item(id: Int, price: Double, ...)
case class PricedOrder(...)
```



Parallelization Contracts (PACTs) & Iterative Dataflows

- **efficient execution of iterative computations is crucial** for graph processing and machine learning workloads
- PACTs offer special support for embedding iterative computations into a DAG
- abstraction: iteration starts with initial state $s^{(0)}$, step function f is repeatedly applied until **fixpoint** $s^* = f(s^*)$ is reached
- **Iterative Dataflows**
 - user marks part of the DAG is iterative
 - **system repeatedly executes this part of the DAG** by feeding back the output of its last operator to its first operator until a convergence criterion is met (*bulk iterations*)
 - special mode for iterative computations that only recompute parts of the state in each iteration (*delta iterations*)





Overview

- A little bit of history: From Relational Databases to Massively Parallel Dataflow Processing
- Distributed Shared-Nothing Filesystems
- Abstractions for Massively Parallel Dataflow Processing
 - MapReduce
 - Parallelization Contracts & Iterative Dataflows
 - **Resilient Distributed Datasets**
- Summary



Resilient Distributed Datasets (RDDs)



- core abstraction of the distributed **data processing engine Apache Spark**
- motivated by growing need to **efficiently execute applications that re-use intermediate results** multiple across multiple operations (e.g., machine learning, graph processing, ad-hoc data analysis)
- **distributed, shared-memory abstraction for MapReduce-like computations**
- **coarse-grained transformations** rather than fine-grained updates for simple fault tolerance
- parallel computations on RDDs using a set of **high level operators** and UDFs (analogous to PACTs and MapReduce)
- system automatically handles parallelization, work distribution and fault tolerance



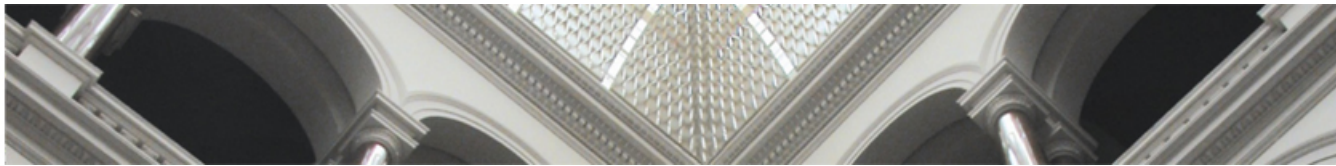
Resilient Distributed Datasets (RDDs)

- **read-only, fault-tolerant, partitioned, parallel data structures**
- allow users to
 - **explicitly persist intermediate results** in memory
 - **control the partitioning** of the data for placement optimization
 - create new RDDs from stable storage or by transforming existing RDDs using a **rich set of operators**
- **immutability and bulk operations** enable
 - straggler mitigation through speculative execution
 - graceful out-of-core execution for bulk reads under memory pressure
 - scheduling based on data locality
 - **lineage-based recovery** for fault tolerance

Resilient Distributed Datasets (RDDs)

- WordCount in Spark's Scala API

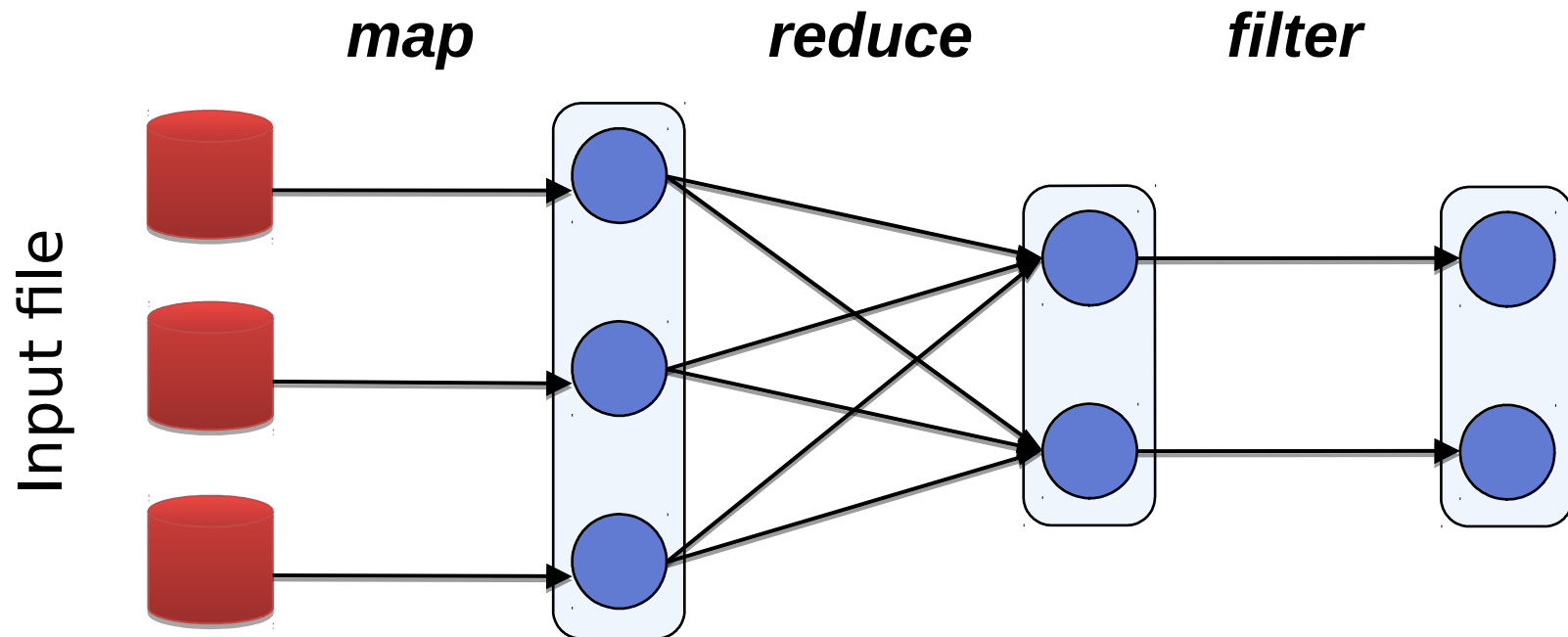
```
val counts = documents
  .flatMap(document => document.split("\\W+"))
  .map(term => (term, 1))
  .reduceByKey(_ + _)
```



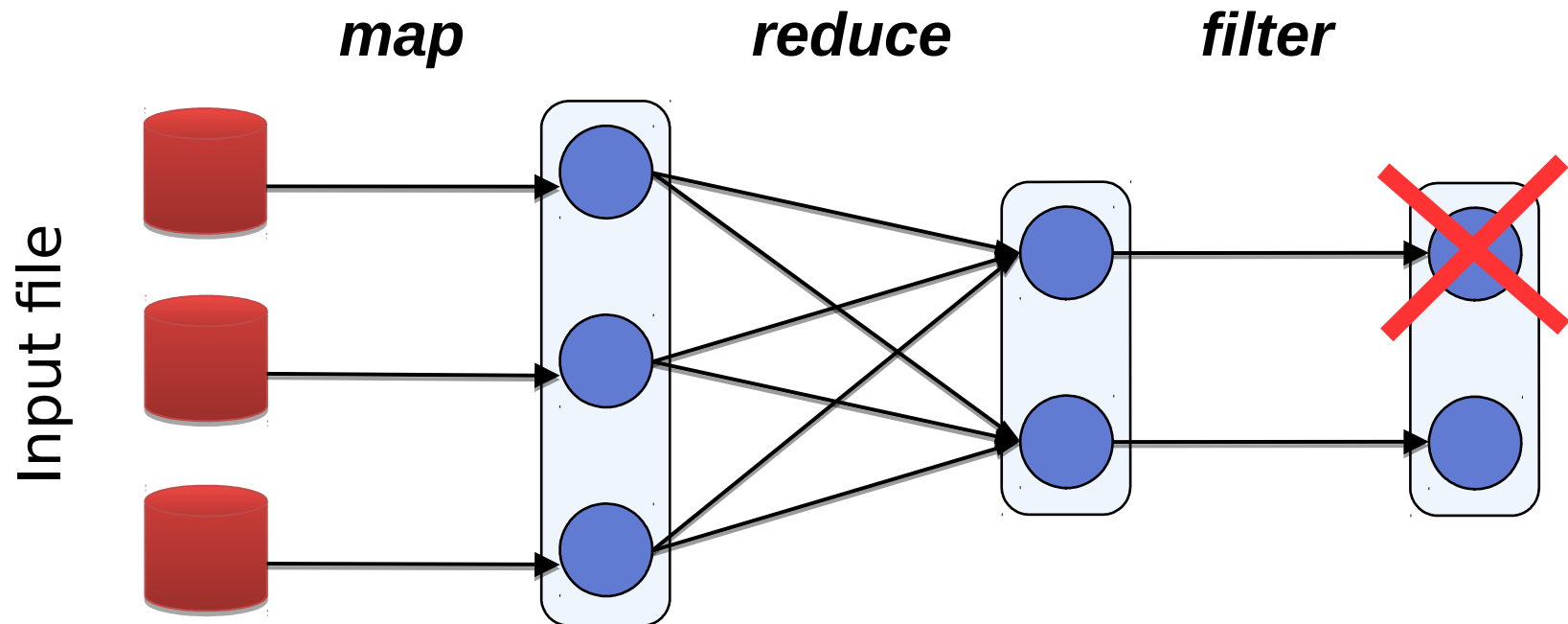
Resilient Distributed Datasets (RDDs)

- main idea of **lineage-based recovery**: log transformations applied to data instead of the resulting data itself,
- **system represents lineage as a graph**
 - vertices: individual partitions of RDDs
 - edges: data dependencies of transformations between RDDs
- two kinds of dependencies
 - *narrow dependencies*: **one-to-one relation between partitions of parent and child RDD**, (map transformations or joins on co-partitioned data)
 - simply re-compute lost partitions in case of failures
 - *wide dependencies*: **all-to-all relation between partitions of parent and child RDD**, (transformations that require to re-partition the data)
 - require full re-computation of partitions of parent RDD in case of failures

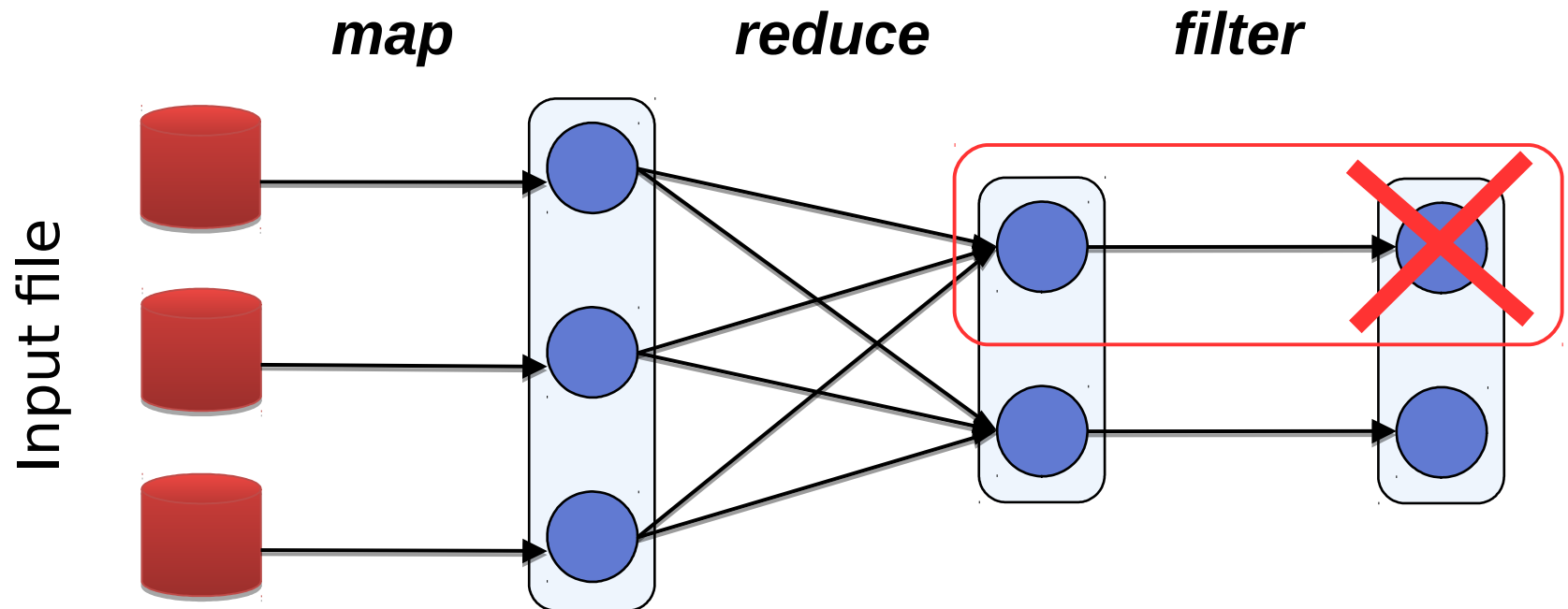
Lineage-based Recovery



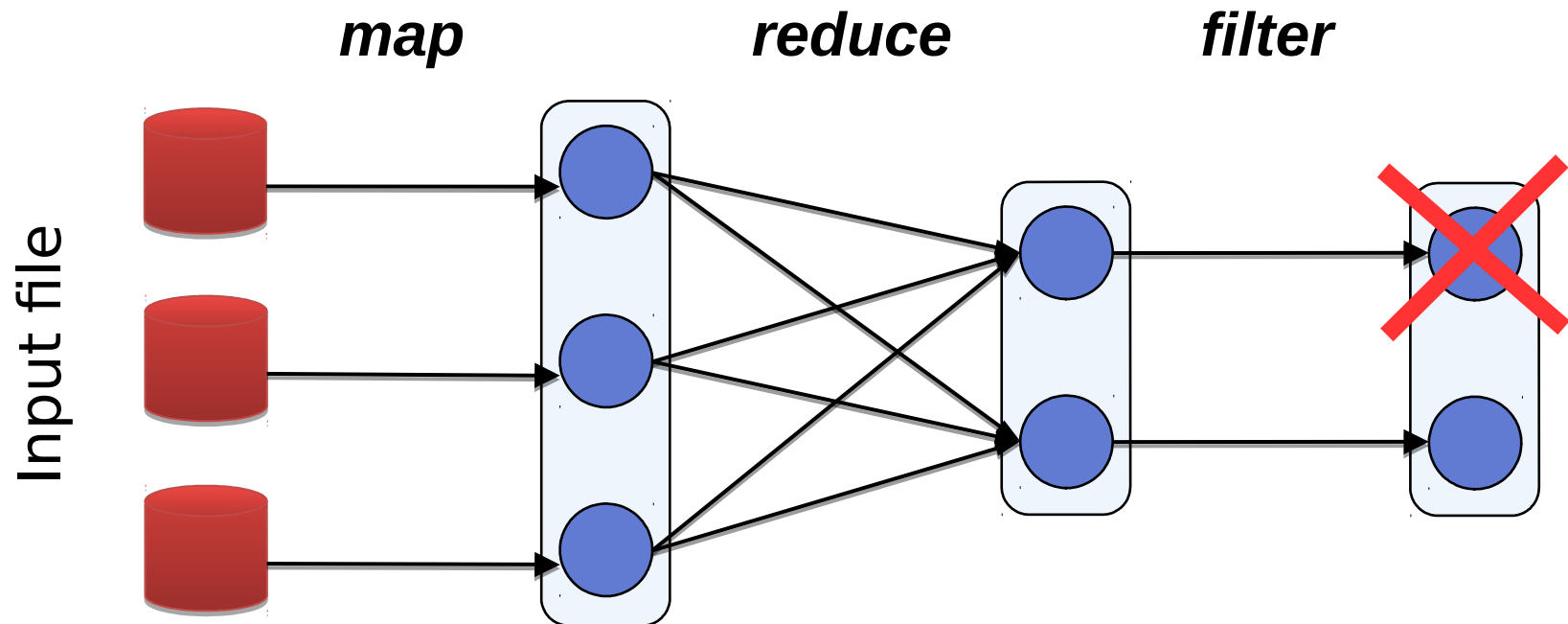
Lineage-based Recovery



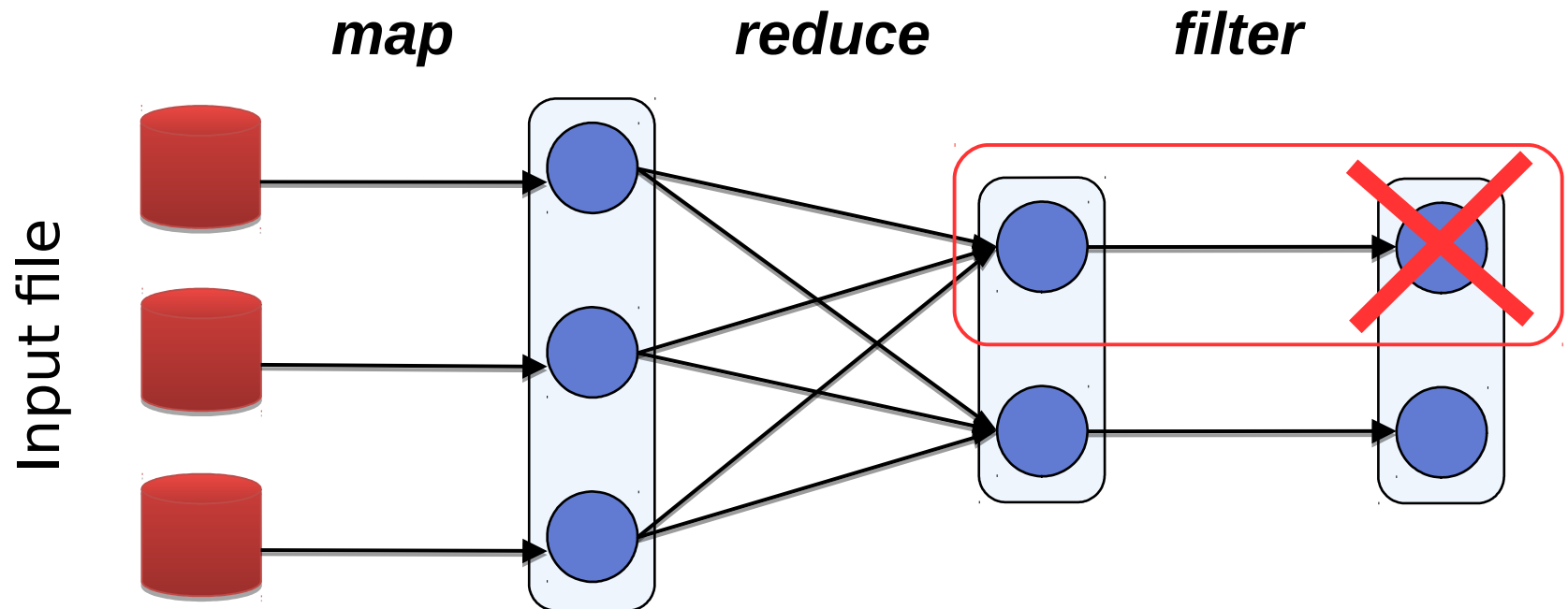
Lineage-based Recovery



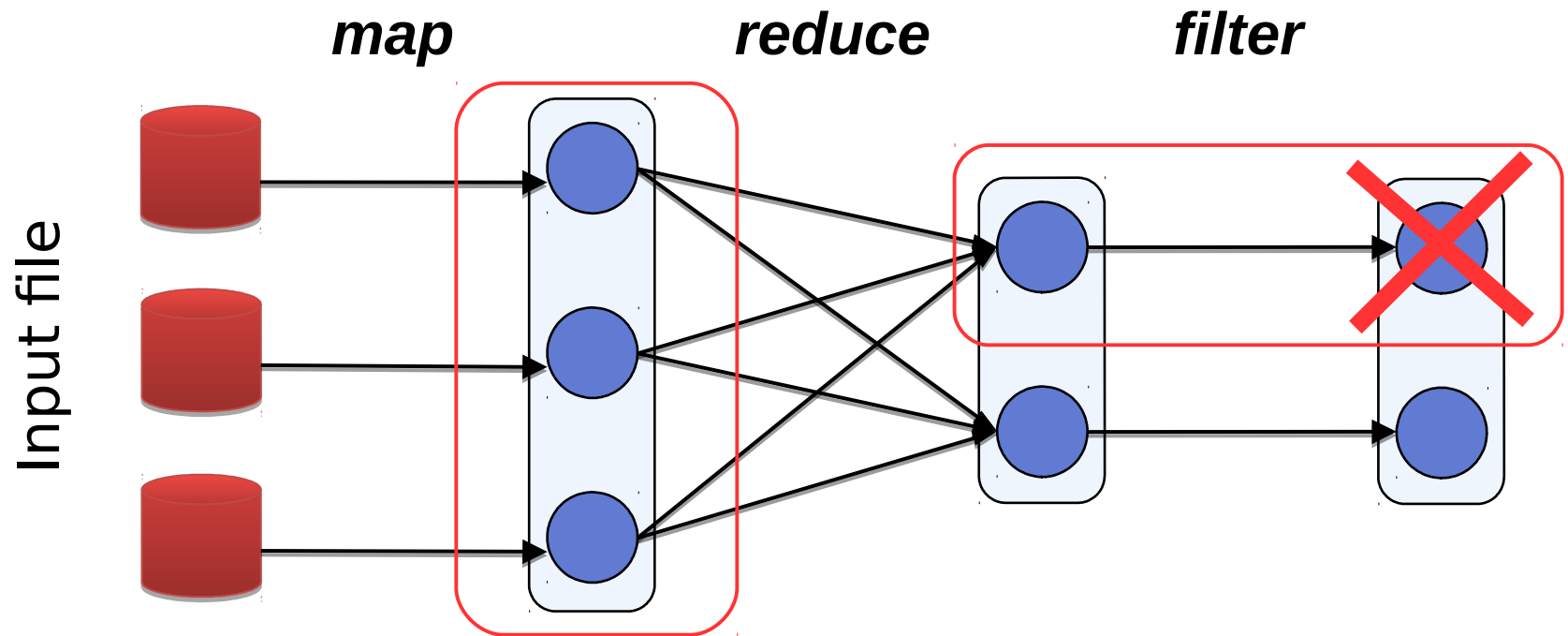
Lineage-based Recovery



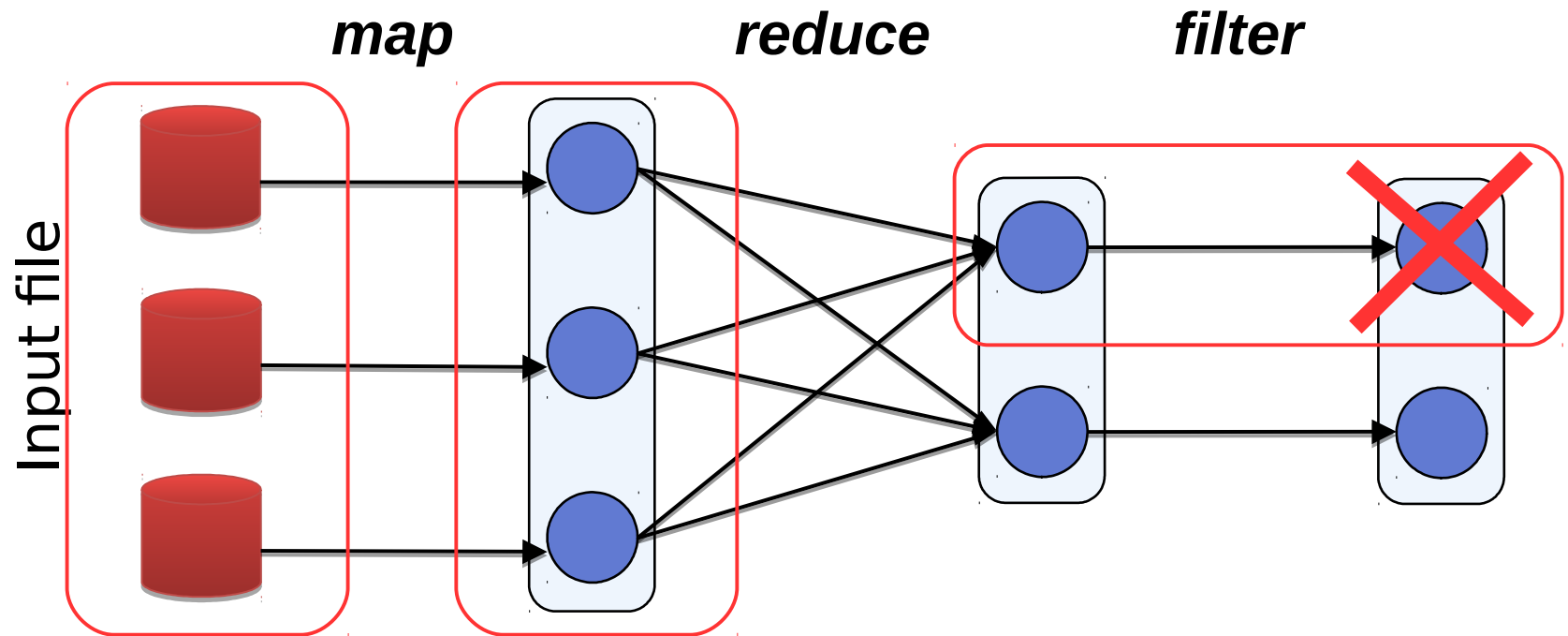
Lineage-based Recovery



Lineage-based Recovery



Lineage-based Recovery





Resilient Distributed Datasets (RDDs)

- **master-slave architecture** used for execution
- **driver program** connects to master and a cluster of workers, issues instructions to create and transform RDDs
- **execution is deferred** until an *action* is requested (an operation that either sends data to the driver program or requires to materialize an RDD)
- **driver tracks lineage**, workers store and process partitioned RDDs
- **execution divided** into *stages*: all pipelineable transformations with narrow dependencies until a transformation with wide dependencies is encountered



Overview

- A little bit of history: From Relational Databases to Massively Parallel Dataflow Processing
- Distributed Shared-Nothing Filesystems
- Abstractions for Massively Parallel Dataflow Processing
 - MapReduce
 - Parallelization Contracts & Iterative Dataflows
 - Resilient Distributed Datasets
- **Summary**



Summary

- **parallel query processing solved** in the 1990s for relation data processing
- **rise of the internet** produces new data processing challenges:
 - large clusters built from commodity hardware for **scalability**
 - dirty, **semi-structured data** from the web
 - **new workloads** (search engines, machine learning, graph processing)
- distributed shared-nothing filesystems
 - designed to store extremely large datasets with **high fault tolerance guarantees**
 - exploitation of high sequential bandwidth of spinning disks
- MapReduce
 - **simple programming model and execution paradigm for distributed data processing**
 - based on **second-order functions** *map* and *reduce*
 - automatically handles parallelization, scalability, failures and concurrency



Summary

- Parallelization Contracts & Iterative Dataflows
 - **generalization of the second-order function paradigm** of MapReduce
 - programs consist of **large dataflow graphs**
 - **automatic optimization**
 - dedicated abstraction for iterative computations
- Resilient Distributed Datasets
 - **distributed, shared-memory abstraction** for MapReduce-like computations
 - **coarse-grained transformations**
 - simple fault tolerance via **lineage-based recovery**



Further Reading

- Vinayak Borkar, Michael J Carey, and Chen Li. *Inside big data management: ogres, onions, or parfaits?* ACM International Conference on Extending Database Technology, pp. 3–14, 2012.
- Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. *The Google File System*. ACM SIGOPS Operating Systems Review, 37, pp. 29–43, 2003.
- Jeffrey Dean and Sanjay Ghemawat. *MapReduce: simplified data processing on large clusters*. Communication of the ACM, 51, pp. 107–113, 2008.
- Alexander Alexandrov, Rico Bergmann, Stephan Ewen, et al. *The stratosphere platform for big data analytics*. Journal on Very Large Data Bases, 23(6), pp. 939–964, 2014.
- Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, et al. *Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing*. USENIX Conference on Networked Systems Design and Implementation, pp. 2–2., 2012.