# Processing of massive data sets II

Krzysztof Dembczyński

Intelligent Decision Support Systems Laboratory (IDSS)
Poznań University of Technology, Poland

Bachelor studies, seventh semester
Academic year 2018/19 (winter semester)

# Review of previous lectures

- Mining of massive datasets.
- Evolution of database systems.
- Dimensional modeling.
- Processing of massive data sets I:
  - Physical storage and data access
  - Materialization, denormalization and summarization

'

# Outline

# Motivation

- Computational burden $\rightarrow$ divide and conquer

# Motivation

- Computational burden $\rightarrow$ divide and conquer
  - Data partitioning

# Motivation

- Computational burden $\rightarrow$ divide and conquer
  - Data partitioning
  - Distributed systems

# Outline

## Data partitioning

- In general, partitioning divides data (e.g., tables and indexes) into smaller pieces, enabling these pieces to be managed and accessed at a finer level of granularity.

# Data partitioning

- In general, partitioning divides data (e.g., tables and indexes) into smaller pieces, enabling these pieces to be managed and accessed at a finer level of granularity.
- Partitioning concerns tables in distributed systems like MapReduce (sometimes referred to as sharding), distributed and parallel databases, but also conventional tables and datasets.

# Data partitioning

- In general, partitioning divides data (e.g., tables and indexes) into smaller pieces, enabling these pieces to be managed and accessed at a finer level of granularity.
- Partitioning concerns tables in distributed systems like MapReduce (sometimes referred to as sharding), distributed and parallel databases, but also conventional tables and datasets.
- Partitioning can provide benefits by improving manageability, performance, and availability.

# Data partitioning

- In general, partitioning divides data (e.g., tables and indexes) into smaller pieces, enabling these pieces to be managed and accessed at a finer level of granularity.
- Partitioning concerns tables in distributed systems like MapReduce (sometimes referred to as sharding), distributed and parallel databases, but also conventional tables and datasets.
- Partitioning can provide benefits by improving manageability, performance, and availability.
- Partitioning is transparent for database queries.

# Data partitioning

- In general, partitioning divides data (e.g., tables and indexes) into smaller pieces, enabling these pieces to be managed and accessed at a finer level of granularity.
- Partitioning concerns tables in distributed systems like MapReduce (sometimes referred to as sharding), distributed and parallel databases, but also conventional tables and datasets.
- Partitioning can provide benefits by improving manageability, performance, and availability.
- Partitioning is transparent for database queries.
- Horizontal vs. vertical vs. chunk partitioning.

# Data partitioning

- Table or index is subdivided into smaller pieces.

# Data partitioning

- Table or index is subdivided into smaller pieces.
- Each piece of database object is called a partition.

## Data partitioning

- Table or index is subdivided into smaller pieces.
- Each piece of database object is called a partition.
- Each partition has its own name, and may have its own storage characteristics (e.g. table compression).

# Data partitioning

- Table or index is subdivided into smaller pieces.
- Each piece of database object is called a partition.
- Each partition has its own name, and may have its own storage characteristics (e.g. table compression).
- From the perspective of a database administrator, a partitioned object has multiple pieces which can be managed either collectively or individually.

# Data partitioning

- Table or index is subdivided into smaller pieces.
- Each piece of database object is called a partition.
- Each partition has its own name, and may have its own storage characteristics (e.g. table compression).
- From the perspective of a database administrator, a partitioned object has multiple pieces which can be managed either collectively or individually.
- From the perspective of the application, however, a partitioned table is identical to a non-partitioned table.

# Data partitioning

- Tables are partitioned using a 'partitioning key', a set of columns which determines in which partition a given row will reside.

# Data partitioning

- Tables are partitioned using a 'partitioning key', a set of columns which determines in which partition a given row will reside.
- Different techniques for partitioning tables:

# Data partitioning

- Tables are partitioned using a 'partitioning key', a set of columns which determines in which partition a given row will reside.
- Different techniques for partitioning tables:
  - Hash partitioning: Rows divided into partitions using a hash function

# Data partitioning

- Tables are partitioned using a 'partitioning key', a set of columns which determines in which partition a given row will reside.
- Different techniques for partitioning tables:
  - Hash partitioning: Rows divided into partitions using a hash function
  - Range partitioning: Each partition holds a range of attribute values

# Data partitioning

- Tables are partitioned using a 'partitioning key', a set of columns which determines in which partition a given row will reside.
- Different techniques for partitioning tables:
  - Hash partitioning: Rows divided into partitions using a hash function
  - Range partitioning: Each partition holds a range of attribute values
  - List partitioning: Rows divided according to lists of values that describe the partition

# Data partitioning

- Tables are partitioned using a 'partitioning key', a set of columns which determines in which partition a given row will reside.
- Different techniques for partitioning tables:
  - Hash partitioning: Rows divided into partitions using a hash function
  - Range partitioning: Each partition holds a range of attribute values
  - List partitioning: Rows divided according to lists of values that describe the partition
  - Composite Partitioning: partitions data using the range method, and within each partition, subpartitions it using the hash or list method.

# Data partitioning

- **Example**:

```
CREATE TABLE sales_list (
  salesman_id NUMBER(5),
  salesman_name VARCHAR2(30),
  sales_state VARCHAR2(20),
  sales_amount NUMBER(10),
  sales_date DATE)
  PARTITION BY LIST(sales_state)
  (
    PARTITION sales_west VALUES('California', 'Hawaii'),
    PARTITION sales_east VALUES ('New York', 'Virginia'),
    PARTITION sales_central VALUES('Texas', 'Illinois')
    PARTITION sales_other VALUES(DEFAULT)
  )
);
```

# Partitioning and manageability

- Maintenance operations can be focused on particular portions of tables,

# Partitioning and manageability

- Maintenance operations can be focused on particular portions of tables,
- Partial compression,

# Partitioning and manageability

- Maintenance operations can be focused on particular portions of tables,
- Partial compression,
- Partial backups,

# Partitioning and manageability

- Maintenance operations can be focused on particular portions of tables,
- Partial compression,
- Partial backups,
- Data recovery can concern partitions,

# Partitioning and manageability

- Maintenance operations can be focused on particular portions of tables,
- Partial compression,
- Partial backups,
- Data recovery can concern partitions,
- "Divide and conquer" approach to data management.

# Data partitioning and star schema

- Partition fact table:

# Data partitioning and star schema

- Partition fact table:
  - Fact tables are big,

## Data partitioning and star schema

- Partition fact table:
  - Fact tables are big,
  - Process queries in parallel for each partition,

## Data partitioning and star schema

- Partition fact table:
    - Fact tables are big,
    - Process queries in parallel for each partition,
    - Divide the work among the nodes in the cluster,

# Data partitioning and star schema

- Partition fact table:
  - Fact tables are big,
  - Process queries in parallel for each partition,
  - Divide the work among the nodes in the cluster,
  - Specific queries would access only few partitions.

## Data partitioning and star schema

- Partition fact table:
  - Fact tables are big,
  - Process queries in parallel for each partition,
  - Divide the work among the nodes in the cluster,
  - Specific queries would access only few partitions.
- Replicate dimension tables across cluster nodes:

# Data partitioning and star schema

- Partition fact table:
  - Fact tables are big,
  - Process queries in parallel for each partition,
  - Divide the work among the nodes in the cluster,
  - Specific queries would access only few partitions.
- Replicate dimension tables across cluster nodes:
  - Dimension tables are small,

# Data partitioning and star schema

- Partition fact table:
  - Fact tables are big,
  - Process queries in parallel for each partition,
  - Divide the work among the nodes in the cluster,
  - Specific queries would access only few partitions.
- Replicate dimension tables across cluster nodes:
  - Dimension tables are small,
  - Storing multiple copies of them is cheap,

# Data partitioning and star schema

- Partition fact table:
    - Fact tables are big,
    - Process queries in parallel for each partition,
    - Divide the work among the nodes in the cluster,
    - Specific queries would access only few partitions.
- Replicate dimension tables across cluster nodes:
    - Dimension tables are small,
    - Storing multiple copies of them is cheap,
    - No communication needed for parallel joins.

# Data partitioning and star schema

- Partition fact table:
  - Fact tables are big,
  - Process queries in parallel for each partition,
  - Divide the work among the nodes in the cluster,
  - Specific queries would access only few partitions.
- Replicate dimension tables across cluster nodes:
  - Dimension tables are small,
  - Storing multiple copies of them is cheap,
  - No communication needed for parallel joins.
- One big dimension:

# Data partitioning and star schema

- Partition fact table:
  - Fact tables are big,
  - Process queries in parallel for each partition,
  - Divide the work among the nodes in the cluster,
  - Specific queries would access only few partitions.
- Replicate dimension tables across cluster nodes:
  - Dimension tables are small,
  - Storing multiple copies of them is cheap,
  - No communication needed for parallel joins.
- One big dimension:
  - Sometimes one dimension table is quite big (e.g. customer),

## Data partitioning and star schema

- Partition fact table:
  - Fact tables are big,
  - Process queries in parallel for each partition,
  - Divide the work among the nodes in the cluster,
  - Specific queries would access only few partitions.
- Replicate dimension tables across cluster nodes:
  - Dimension tables are small,
  - Storing multiple copies of them is cheap,
  - No communication needed for parallel joins.
- One big dimension:
  - Sometimes one dimension table is quite big (e.g. customer),
  - Partition the big dimension table,

# Data partitioning and star schema

- Partition fact table:
  - Fact tables are big,
  - Process queries in parallel for each partition,
  - Divide the work among the nodes in the cluster,
  - Specific queries would access only few partitions.
- Replicate dimension tables across cluster nodes:
  - Dimension tables are small,
  - Storing multiple copies of them is cheap,
  - No communication needed for parallel joins.
- One big dimension:
  - Sometimes one dimension table is quite big (e.g. customer),
  - Partition the big dimension table,
  - Partition fact table on key of big dimension,

# Data partitioning and star schema

- Partition fact table:
  - Fact tables are big,
  - Process queries in parallel for each partition,
  - Divide the work among the nodes in the cluster,
  - Specific queries would access only few partitions.
- Replicate dimension tables across cluster nodes:
  - Dimension tables are small,
  - Storing multiple copies of them is cheap,
  - No communication needed for parallel joins.
- One big dimension:
  - Sometimes one dimension table is quite big (e.g. customer),
  - Partition the big dimension table,
  - Partition fact table on key of big dimension,
  - The join operation can be performed on smaller tables.

# Data partitioning and star schema

- Reducing load time via partitioning:

# Data partitioning and star schema

- Reducing load time via partitioning:
  - Often fact tables are partitioned on Date,

# Data partitioning and star schema

- Reducing load time via partitioning:
  - Often fact tables are partitioned on Date,
  - Newly loaded records go into the last partition,

# Data partitioning and star schema

- Reducing load time via partitioning:
  - Often fact tables are partitioned on Date,
  - Newly loaded records go into the last partition,
  - Only indexes and aggregates for that partition need to be updated,

# Data partitioning and star schema

- Reducing load time via partitioning:
  - Often fact tables are partitioned on Date,
  - Newly loaded records go into the last partition,
  - Only indexes and aggregates for that partition need to be updated,
  - All other partitions remain unchanged.

## Data partitioning and star schema

- Reducing load time via partitioning:
  - Often fact tables are partitioned on Date,
  - Newly loaded records go into the last partition,
  - Only indexes and aggregates for that partition need to be updated,
  - All other partitions remain unchanged.
- Expiring old data:

# Data partitioning and star schema

- Reducing load time via partitioning:
  - Often fact tables are partitioned on Date,
  - Newly loaded records go into the last partition,
  - Only indexes and aggregates for that partition need to be updated,
  - All other partitions remain unchanged.
- Expiring old data:
  - Often older data is less useful / relevant for data analysts,

# Data partitioning and star schema

- Reducing load time via partitioning:
  - Often fact tables are partitioned on Date,
  - Newly loaded records go into the last partition,
  - Only indexes and aggregates for that partition need to be updated,
  - All other partitions remain unchanged.
- Expiring old data:
  - Often older data is less useful / relevant for data analysts,
  - To reduce database size, old data is often deleted,

# Data partitioning and star schema

- Reducing load time via partitioning:
  - Often fact tables are partitioned on Date,
  - Newly loaded records go into the last partition,
  - Only indexes and aggregates for that partition need to be updated,
  - All other partitions remain unchanged.
- Expiring old data:
  - Often older data is less useful / relevant for data analysts,
  - To reduce database size, old data is often deleted,
  - If data is partitioned on date, simply delete or compress the oldest partitions.

# Partitioning and sorting

- External-memory sorting:

## Partitioning and sorting

- External-memory sorting:
  - Let data be of size $n$ and main memory be of size $k+1$ units ($k$ input and one output buffer).

# Partitioning and sorting

- External-memory sorting:
  - Let data be of size $n$ and main memory be of size $k + 1$ units ($k$ input and one output buffer).
  - Partition data into $n/k$ parts (does not have to be made explicitly).

# Partitioning and sorting

- External-memory sorting:
  - Let data be of size $n$ and main memory be of size $k + 1$ units ($k$ input and one output buffer).
  - Partition data into $n/k$ parts (does not have to be made explicitly).
  - For each partition (each uses $k$ memory units):

# Partitioning and sorting

- External-memory sorting:
  - Let data be of size $n$ and main memory be of size $k+1$ units ($k$ input and one output buffer).
  - Partition data into $n/k$ parts (does not have to be made explicitly).
  - For each partition (each uses $k$ memory units):
    - Read to main memory

# Partitioning and sorting

- External-memory sorting:
  - Let data be of size $n$ and main memory be of size $k + 1$ units ($k$ input and one output buffer).
  - Partition data into $n/k$ parts (does not have to be made explicitly).
  - For each partition (each uses $k$ memory units):
    - Read to main memory
    - Sort partition

# Partitioning and sorting

- External-memory sorting:
  - Let data be of size $n$ and main memory be of size $k+1$ units ($k$ input and one output buffer).
  - Partition data into $n/k$ parts (does not have to be made explicitly).
  - For each partition (each uses $k$ memory units):
    - Read to main memory
    - Sort partition
    - Write sorted partition to disk

# Partitioning and sorting

- External-memory sorting:
  - Let data be of size $n$ and main memory be of size $k+1$ units ($k$ input and one output buffer).
  - Partition data into $n/k$ parts (does not have to be made explicitly).
  - For each partition (each uses $k$ memory units):
    - Read to main memory
    - Sort partition
    - Write sorted partition to disk
  - Read the first $k/n$ of data from each sorted partition to main memory (use all $k$ input buffers).

## Partitioning and sorting

- External-memory sorting:
  - Let data be of size $n$ and main memory be of size $k+1$ units ($k$ input and one output buffer).
  - Partition data into $n/k$ parts (does not have to be made explicitly).
  - For each partition (each uses $k$ memory units):
    - Read to main memory
    - Sort partition
    - Write sorted partition to disk
  - Read the first $k/n$ of data from each sorted partition to main memory (use all $k$ input buffers).
  - Do

# Partitioning and sorting

- External-memory sorting:
  - Let data be of size $n$ and main memory be of size $k+1$ units ($k$ input and one output buffer).
  - Partition data into $n/k$ parts (does not have to be made explicitly).
  - For each partition (each uses $k$ memory units):
    - Read to main memory
    - Sort partition
    - Write sorted partition to disk
  - Read the first $k/n$ of data from each sorted partition to main memory (use all $k$ input buffers).
  - Do
    - Perform $k$-way merge sort using the output buffer to store globally sorted data.

# Partitioning and sorting

- External-memory sorting:
  - Let data be of size $n$ and main memory be of size $k+1$ units ($k$ input and one output buffer).
  - Partition data into $n/k$ parts (does not have to be made explicitly).
  - For each partition (each uses $k$ memory units):
    - Read to main memory
    - Sort partition
    - Write sorted partition to disk
  - Read the first $k/n$ of data from each sorted partition to main memory (use all $k$ input buffers).
  - Do
    - Perform $k$-way merge sort using the output buffer to store globally sorted data.
    - Write output buffer to disk if it is filled.

# Partitioning and sorting

- External-memory sorting:
  - Let data be of size $n$ and main memory be of size $k+1$ units ($k$ input and one output buffer).
  - Partition data into $n/k$ parts (does not have to be made explicitly).
  - For each partition (each uses $k$ memory units):
    - Read to main memory
    - Sort partition
    - Write sorted partition to disk
  - Read the first $k/n$ of data from each sorted partition to main memory (use all $k$ input buffers).
  - Do
    - Perform $k$-way merge sort using the output buffer to store globally sorted data.
    - Write output buffer to disk if it is filled.
    - If the $i$th input buffer is exhausted, read next portion from $i$th partition.

## Partitioning and sorting

- External-memory sorting:
  - Let data be of size $n$ and main memory be of size $k+1$ units ($k$ input and one output buffer).
  - Partition data into $n/k$ parts (does not have to be made explicitly).
  - For each partition (each uses $k$ memory units):
    - Read to main memory
    - Sort partition
    - Write sorted partition to disk
  - Read the first $k/n$ of data from each sorted partition to main memory (use all $k$ input buffers).
  - Do
    - Perform $k$-way merge sort using the output buffer to store globally sorted data.
    - Write output buffer to disk if it is filled.
    - If the $i$th input buffer is exhausted, read next portion from $i$th partition.
- Remark that $k \geq \sqrt{n}$ (otherwise we need additional merge passes).

# Partitioning and sorting

- External-memory sorting:
  - Let data be of size $n$ and main memory be of size $k+1$ units ($k$ input and one output buffer).
  - Partition data into $n/k$ parts (does not have to be made explicitly).
  - For each partition (each uses $k$ memory units):
    - Read to main memory
    - Sort partition
    - Write sorted partition to disk
  - Read the first $k/n$ of data from each sorted partition to main memory (use all $k$ input buffers).
  - Do
    - Perform $k$-way merge sort using the output buffer to store globally sorted data.
    - Write output buffer to disk if it is filled.
    - If the $i$th input buffer is exhausted, read next portion from $i$th partition.
- Remark that $k \geq \sqrt{n}$ (otherwise we need additional merge passes).
- External-memory sorting is used in merge-join of large data sets.

## Partitioning and sorting

- External-memory sorting:
  - Let data be of size $n$ and main memory be of size $k+1$ units ($k$ input and one output buffer).
  - Partition data into $n/k$ parts (does not have to be made explicitly).
  - For each partition (each uses $k$ memory units):
    - Read to main memory
    - Sort partition
    - Write sorted partition to disk
  - Read the first $k/n$ of data from each sorted partition to main memory (use all $k$ input buffers).
  - Do
    - Perform $k$-way merge sort using the output buffer to store globally sorted data.
    - Write output buffer to disk if it is filled.
    - If the $i$th input buffer is exhausted, read next portion from $i$th partition.
- Remark that $k \geq \sqrt{n}$ (otherwise we need additional merge passes).
- External-memory sorting is used in merge-join of large data sets.
- Similarly one can generalize hash-join to the so-called partitioned hash-join.

# Outline

# MapReduce-based systems

- Traditional DBMS vs. NoSQL

# MapReduce-based systems

- Traditional DBMS vs. NoSQL
- New emerging applications: search engines, social networks, online shopping, online advertising, recommender systems, etc.

# MapReduce-based systems

- Traditional DBMS vs. NoSQL
- New emerging applications: search engines, social networks, online shopping, online advertising, recommender systems, etc.
- New computational challenges: WordCount, PageRank, etc.

# MapReduce-based systems

- Traditional DBMS vs. NoSQL
- New emerging applications: search engines, social networks, online shopping, online advertising, recommender systems, etc.
- New computational challenges: WordCount, PageRank, etc.
- Computational burden $\rightarrow$ distributed systems

# MapReduce-based systems

- Traditional DBMS vs. NoSQL
- New emerging applications: search engines, social networks, online shopping, online advertising, recommender systems, etc.
- New computational challenges: WordCount, PageRank, etc.
- Computational burden $\rightarrow$ distributed systems
  - Scaling-out instead of scaling-up

## MapReduce-based systems

- Traditional DBMS vs. NoSQL
- New emerging applications: search engines, social networks, online shopping, online advertising, recommender systems, etc.
- New computational challenges: WordCount, PageRank, etc.
- Computational burden $\rightarrow$ distributed systems
  - Scaling-out instead of scaling-up
  - Move-code-to-data

# MapReduce-based systems

- Accessible – run on large clusters of commodity machines or on cloud computing services such as AWS (Amazon Web Services).

# MapReduce-based systems

- Accessible – run on large clusters of commodity machines or on cloud computing services such as AWS (Amazon Web Services).
- Robust – are intended to run on commodity hardware; designed with the assumption of frequent hardware malfunctions; they can gracefully handle most such failures.

# MapReduce-based systems

- Accessible – run on large clusters of commodity machines or on cloud computing services such as AWS (Amazon Web Services).
- Robust – are intended to run on commodity hardware; designed with the assumption of frequent hardware malfunctions; they can gracefully handle most such failures.
- Scalable – scales linearly to handle larger data by adding more nodes to the cluster.

# MapReduce-based systems

- Accessible – run on large clusters of commodity machines or on cloud computing services such as AWS (Amazon Web Services).
- Robust – are intended to run on commodity hardware; designed with the assumption of frequent hardware malfunctions; they can gracefully handle most such failures.
- Scalable – scales linearly to handle larger data by adding more nodes to the cluster.
- Simple – allow users to quickly write efficient parallel code.

# MapReduce: Two simple procedures

- Word count: A basic operation for every search engine.
- Matrix-vector multiplication: A fundamental step in many algorithms, for example, in PageRank.

## MapReduce: Two simple procedures

- Word count: A basic operation for every search engine.
- Matrix-vector multiplication: A fundamental step in many algorithms, for example, in PageRank.

- How to implement these procedures for efficient execution in a distributed system?

# MapReduce: Two simple procedures

- Word count: A basic operation for every search engine.
- Matrix-vector multiplication: A fundamental step in many algorithms, for example, in PageRank.

- How to implement these procedures for efficient execution in a distributed system?
- How much can we gain by such implementation?

## MapReduce: Two simple procedures

- Word count: A basic operation for every search engine.
- Matrix-vector multiplication: A fundamental step in many algorithms, for example, in PageRank.

- How to implement these procedures for efficient execution in a distributed system?
- How much can we gain by such implementation?
- Let us focus on the word count problem ...

# Word count

- Count the number of times each word occurs in a set of documents:

*Do as I say, not as I do.*

| Word | Count |
|---|---|
| as | 2 |
| do | 2 |
| i | 2 |
| not | 1 |
| say | 1 |

## Word count

- Let us write the procedure in pseudo-code for a single machine:

## Word count

- Let us write the procedure in pseudo-code for a single machine:

```
define wordCount as Multiset;

for each document in documentSet {
    T = tokenize(document);

    for each token in T {
        wordCount[token]++;
    }

}

display(wordCount);
```

# Word count

- Let us write the procedure in pseudo-code for many machines:

# Word count

- Let us write the procedure in pseudo-code for many machines:
  - First step:

```
define wordCount as Multiset;

for each document in documentSubset {
    T = tokenize(document);
    for each token in T {
        wordCount[token]++;
    }
}

sendToSecondPhase(wordCount);
```

## Word count

- Let us write the procedure in pseudo-code for many machines:
  - First step:

```
define wordCount as Multiset;

for each document in documentSubset {
    T = tokenize(document);
    for each token in T {
        wordCount[token]++;
    }
}

sendToSecondPhase(wordCount);
```

  - Second step:

```
define totalWordCount as Multiset;

for each wordCount received from firstPhase {
   multisetAdd (totalWordCount, wordCount);
}
```

## Word count

- To make the procedure work properly across a cluster of distributed machines, we need to add a number of functionalities:

# Word count

- To make the procedure work properly across a cluster of distributed machines, we need to add a number of functionalities:
  - Store files over many processing machines (of phase one).

# Word count

- To make the procedure work properly across a cluster of distributed machines, we need to add a number of functionalities:
  - Store files over many processing machines (of phase one).
  - Write a disk-based hash table permitting processing without being limited by RAM capacity.

# Word count

- To make the procedure work properly across a cluster of distributed machines, we need to add a number of functionalities:
  - Store files over many processing machines (of phase one).
  - Write a disk-based hash table permitting processing without being limited by RAM capacity.
  - Partition the intermediate data (that is, wordCount) from phase one.

# Word count

- To make the procedure work properly across a cluster of distributed machines, we need to add a number of functionalities:
  - Store files over many processing machines (of phase one).
  - Write a disk-based hash table permitting processing without being limited by RAM capacity.
  - Partition the intermediate data (that is, wordCount) from phase one.
  - Shuffle the partitions to the appropriate machines in phase two.

# Word count

- To make the procedure work properly across a cluster of distributed machines, we need to add a number of functionalities:
  - Store files over many processing machines (of phase one).
  - Write a disk-based hash table permitting processing without being limited by RAM capacity.
  - Partition the intermediate data (that is, wordCount) from phase one.
  - Shuffle the partitions to the appropriate machines in phase two.
  - Ensure fault tolerance.

# MapReduce

- MapReduce programs are executed in two main phases, called mapping and reducing:

# MapReduce

- MapReduce programs are executed in two main phases, called mapping and reducing:
  - Map: the map function is written to convert input elements to key-value pairs.

# MapReduce

- MapReduce programs are executed in two main phases, called mapping and reducing:
  - Map: the map function is written to convert input elements to key-value pairs.
  - Reduce: the reduce function is written to take pairs consisting of a key and its list of associated values and combine those values in some way.

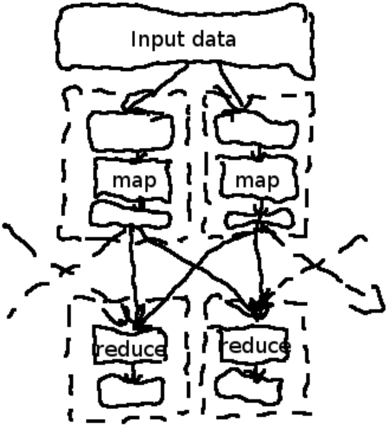
# MapReduce

- The complete data flow:

| | Input | Output |
|---|---|---|
| map | `(<k1, v1>)` | `list(<k2, v2>)` |
| reduce | `(<k2, list(<v2>)` | `list(<k3, v3>)` |

# MapReduce

Figure: The complete data flow



# MapReduce

- The complete data flow:

# MapReduce

- The complete data flow:
  - The input is structured as a list of key-value pairs: `list(<k1,v1>)`.

# MapReduce

- The complete data flow:
  - The input is structured as a list of key-value pairs: `list(<k1,v1>)`.
  - The list of key-value pairs is broken up and each individual key-value pair, `<k1,v1>`, is processed by calling the map function of the mapper (the key `k1` is often ignored by the mapper).

# MapReduce

- The complete data flow:
  - The input is structured as a list of key-value pairs: `list(<k1,v1>)`.
  - The list of key-value pairs is broken up and each individual key-value pair, `<k1,v1>`, is processed by calling the map function of the mapper (the key `k1` is often ignored by the mapper).
  - The mapper transforms each `<k1,v1>` pair into a list of `<k2,v2>` pairs.

# MapReduce

- The complete data flow:
  - The input is structured as a list of key-value pairs: `list(<k1,v1>)`.
  - The list of key-value pairs is broken up and each individual key-value pair, `<k1,v1>`, is processed by calling the map function of the mapper (the key `k1` is often ignored by the mapper).
  - The mapper transforms each `<k1,v1>` pair into a list of `<k2,v2>` pairs.
  - The key-value pairs are processed in arbitrary order.

# MapReduce

- The complete data flow:
  - The input is structured as a list of key-value pairs: `list(<k1,v1>)`.
  - The list of key-value pairs is broken up and each individual key-value pair, `<k1,v1>`, is processed by calling the map function of the mapper (the key `k1` is often ignored by the mapper).
  - The mapper transforms each `<k1,v1>` pair into a list of `<k2,v2>` pairs.
  - The key-value pairs are processed in arbitrary order.
  - The output of all the mappers are (conceptually) aggregated into one giant list of `<k2,v2>` pairs. All pairs sharing the same `k2` are grouped together into a new aggregated key-value pair: `<k2,list(v2)>`.

# MapReduce

- The complete data flow:
  - The input is structured as a list of key-value pairs: `list(<k1,v1>)`.
  - The list of key-value pairs is broken up and each individual key-value pair, `<k1,v1>`, is processed by calling the map function of the mapper (the key `k1` is often ignored by the mapper).
  - The mapper transforms each `<k1,v1>` pair into a list of `<k2,v2>` pairs.
  - The key-value pairs are processed in arbitrary order.
  - The output of all the mappers are (conceptually) aggregated into one giant list of `<k2,v2>` pairs. All pairs sharing the same `k2` are grouped together into a new aggregated key-value pair: `<k2,list(v2)>`.
  - The framework asks the reducer to process each one of these aggregated key-value pairs individually.

# Combiner and partitioner

- Beside map and reduce there are two other important elements that can be implemented within the MapReduce framework to control the data flow.

## Combiner and partitioner

- Beside map and reduce there are two other important elements that can be implemented within the MapReduce framework to control the data flow.
- **Combiner** – perform local aggregation (the reduce step) on the map node.

# Combiner and partitioner

- Beside map and reduce there are two other important elements that can be implemented within the MapReduce framework to control the data flow.
- **Combiner** – perform local aggregation (the reduce step) on the map node.
- **Partitioner** – divide the key space of the map output and assign the key-value pairs to reducers.

## WordCount in MapReduce

- Map:
  - For a pair `<k1,document>` produce a sequence of pairs `<token,1>`, where `token` is a token/word found in the document.

```
map(String filename, String document) {
    List<String> T = tokenize(document);

    for each token in T {
      emit ((String)token, (Integer) 1);
    }

}
```

# WordCount in MapReduce

- Reduce
  - For a pair `<word, list(1, 1, ..., 1)>` sum up all ones appearing in the list and return `<word, sum>`, where `sum` is the sum of ones.

```
reduce(String token, List<Integer> values) {
    Integer sum = 0;

    for each value in values {
        sum = sum + value;
    }

    emit ((String)token, (Integer) sum);
}
```

## Matrix-vector Multiplication

- Let $\boldsymbol{A}$ to be large $n \times m$ matrix, and $\boldsymbol{x}$ a long vector of size $m$.
- The matrix-vector multiplication is defined as:

## Matrix-vector Multiplication

- Let $\boldsymbol{A}$ to be large $n \times m$ matrix, and $\boldsymbol{x}$ a long vector of size $m$.
- The matrix-vector multiplication is defined as:

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{v},$$

where $\boldsymbol{v} = (v_1, \ldots, v_n)$ and

$$v_i = \sum_{j=1}^{m} a_{ij} x_j.$$

## Matrix-vector multiplication

- Let us first assume that $m$ is large, but not so large that vector $\boldsymbol{x}$ cannot fit in main memory, and be part of the input to every Map task.
- The matrix $\boldsymbol{A}$ is stored with explicit coordinates, as a triple $(i, j, a_{ij})$.
- We also assume the position of element $x_j$ in the vector $\boldsymbol{x}$ will be stored in the analogous way.

# Matrix-vector multiplication

- **Map**:

# Matrix-vector multiplication

- **Map**: each map task will take the entire vector $\boldsymbol{x}$ and a chunk of the matrix $\boldsymbol{A}$. From each matrix element $a_{ij}$ it produces the key-value pair $(i, a_{ij}x_j)$. Thus, all terms of the sum that make up the component $v_i$ of the matrix-vector product will get the same key.

# Matrix-vector multiplication

- **Map**: each map task will take the entire vector $\boldsymbol{x}$ and a chunk of the matrix $\boldsymbol{A}$. From each matrix element $a_{ij}$ it produces the key-value pair $(i, a_{ij}x_j)$. Thus, all terms of the sum that make up the component $v_i$ of the matrix-vector product will get the same key.
- **Reduce**:

# Matrix-vector multiplication

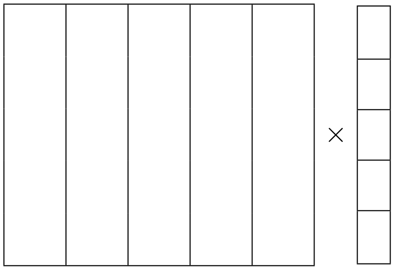
- **Map**: each map task will take the entire vector $\boldsymbol{x}$ and a chunk of the matrix $\boldsymbol{A}$. From each matrix element $a_{ij}$ it produces the key-value pair $(i, a_{ij}x_j)$. Thus, all terms of the sum that make up the component $v_i$ of the matrix-vector product will get the same key.
- **Reduce**: a reduce task has simply to sum all the values associated with a given key $i$. The result will be a pair $(i, v_i)$ where:

$$v_i = \sum_{j=1}^{m} a_{ij}x_j.$$

# Matrix-Vector Multiplication with Large Vector $v$

## Matrix-Vector Multiplication with Large Vector $v$

- Divide the matrix into vertical stripes of equal width and divide the vector into an equal number of horizontal stripes, of the same height.



- The $i$th stripe of the matrix multiplies only components from the $i$th stripe of the vector.
- Thus, we can divide the matrix into one file for each stripe, and do the same for the vector.

## Matrix-Vector Multiplication with Large Vector $v$

- Each Map task is assigned a chunk from one the stripes of the matrix and gets the entire corresponding stripe of the vector.
- The Map and Reduce tasks can then act exactly as in the case where Map tasks get the entire vector.

# Outline

# Spark

- Spark is a fast and general-purpose cluster computing system.

## Spark

- Spark is a fast and general-purpose cluster computing system.
- It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.

# Spark

- Spark is a fast and general-purpose cluster computing system.
- It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.
- It also supports a rich set of higher-level tools including:

# Spark

- Spark is a fast and general-purpose cluster computing system.
- It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.
- It also supports a rich set of higher-level tools including:
    - Spark SQL for SQL and structured data processing,

## Spark

- Spark is a fast and general-purpose cluster computing system.
- It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.
- It also supports a rich set of higher-level tools including:
  - Spark SQL for SQL and structured data processing,
  - MLlib for machine learning,

## Spark

- Spark is a fast and general-purpose cluster computing system.
- It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.
- It also supports a rich set of higher-level tools including:
  - Spark SQL for SQL and structured data processing,
  - MLlib for machine learning,
  - GraphX for graph processing,

# Spark

- Spark is a fast and general-purpose cluster computing system.
- It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.
- It also supports a rich set of higher-level tools including:
    - Spark SQL for SQL and structured data processing,
    - MLlib for machine learning,
    - GraphX for graph processing,
    - and Spark Streaming.

## Spark

- Spark is a fast and general-purpose cluster computing system.
- It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs.
- It also supports a rich set of higher-level tools including:
  - Spark SQL for SQL and structured data processing,
  - MLlib for machine learning,
  - GraphX for graph processing,
  - and Spark Streaming.
- For more check `https://spark.apache.org/`

# Spark

- Spark collaborates with Hadoop which is a popular open-source implementation of MapReduce.

# Spark

- Spark collaborates with Hadoop which is a popular open-source implementation of MapReduce.
- Hadoop works in a master/slave architecture for both distributed storage and distributed computation.

# Spark

- Spark collaborates with Hadoop which is a popular open-source implementation of MapReduce.
- Hadoop works in a master/slave architecture for both distributed storage and distributed computation.
- Hadoop Distributed File System (HDFS) is responsible for distributed storage.

## Installation of Spark

- Download Spark from

# Installation of Spark

- Download Spark from

  `http://spark.apache.org/downloads.html`

# Installation of Spark

- Download Spark from

  `http://spark.apache.org/downloads.html`

- Untar the spark archive:

# Installation of Spark

- Download Spark from

  `http://spark.apache.org/downloads.html`

- Untar the spark archive:

  ```
  tar xvfz spark-2.2.0-bin-hadoop2.7.tar
  ```

# Installation of Spark

- Download Spark from

  `http://spark.apache.org/downloads.html`
- Untar the spark archive:

  ```
  tar xvfz spark-2.2.0-bin-hadoop2.7.tar
  ```
- To play with Spark there is no need to install HDFS ...

# Installation of Spark

- Download Spark from

  `http://spark.apache.org/downloads.html`
- Untar the spark archive:

  ```
  tar xvfz spark-2.2.0-bin-hadoop2.7.tar
  ```
- To play with Spark there is no need to install HDFS ...
- But, you can try to play around with HDFS.

# HDFS

- Create new directories:

# HDFS

- Create new directories:

```
hdfs dfs -mkdir /user
```

# HDFS

- Create new directories:

```
hdfs dfs -mkdir /user
hdfs dfs -mkdir /user/myname
```

# HDFS

- Create new directories:

```
hdfs dfs -mkdir /user
hdfs dfs -mkdir /user/myname
```

- Copy the input files into the distributed filesystem:

# HDFS

- Create new directories:

```
hdfs dfs -mkdir /user
hdfs dfs -mkdir /user/myname
```

- Copy the input files into the distributed filesystem:

```
hdfs dfs -put data.txt /user/myname/data.txt
```

# HDFS

- Create new directories:

```
hdfs dfs -mkdir /user
hdfs dfs -mkdir /user/myname
```

- Copy the input files into the distributed filesystem:

```
hdfs dfs -put data.txt /user/myname/data.txt
```

- View the files in the distributed filesystem:

# HDFS

- Create new directories:

```
hdfs dfs -mkdir /user
hdfs dfs -mkdir /user/myname
```

- Copy the input files into the distributed filesystem:

```
hdfs dfs -put data.txt /user/myname/data.txt
```

- View the files in the distributed filesystem:

```
hdfs dfs -ls /user/myname/
```

# HDFS

- Create new directories:

```
hdfs dfs -mkdir /user
hdfs dfs -mkdir /user/myname
```

- Copy the input files into the distributed filesystem:

```
hdfs dfs -put data.txt /user/myname/data.txt
```

- View the files in the distributed filesystem:

```
hdfs dfs -ls /user/myname/
hdfs dfs -cat /user/myname/data.txt
```

# WordCount in Hadoop

```java
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 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);
      }
    }
  }
  (...)
```

# WordCount in Hadoop

```java
(...)
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);
   }
 }

 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);
 }
}
```

# WordCount in Spark

- The same code is much simpler in Spark
- To run the Spark shell type: `./bin/spark-shell`
- The code

```
val textFile = sc.textFile("~/data/all-bible.txt")
val counts = (textFile.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _))
counts.saveAsTextFile("~/data/all-bible-counts.txt")
```

Alternatively:

```
val textFile = spark.read.textFile("~/data/all-bible.txt")
val wordCounts = textFile.flatMap(line => line.split(" ")).groupByKey(identity).
     count()
```

# Matrix-vector multiplication in Spark

- The Spark code is quite simple:

```
val x = sc.textFile("~/data/x.txt").map(line => {val t = line.split(","); (t(0).
    trim.toInt, t(1).trim.toDouble)})
val vectorX = x.map{case (i,v) => v}.collect
val broadcastedX = sc.broadcast(vectorX)
val matrix = sc.textFile("~/data/M.txt").map(line => {val t = line.split(","); (t
    (0).trim.toInt, t(1).trim.toInt, t(2).trim.toDouble)})
val v = matrix.map { case (i,j,a) => (i, a * broadcastedX.value(j-1))}.reduceByKey(
    _ + _)
v.toDF.orderBy("_1").show
```

# Outline

# Summary

- Computational burden $\rightarrow$ data partitioning, distributed systems.
- Data partitioning
- New data-intensive challenges like search engines.
- MapReduce: The overall idea and simple algorithms.
- Spark: MapReduce in practice.

# Bibliography


- J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2014
  `http://infolab.stanford.edu/~ullman/mmds.html`
- J.Lin and Ch. Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan and Claypool Publishers, 2010
  `http://lintool.github.com/MapReduceAlgorithms/`
- Ch. Lam. *Hadoop in Action*. Manning Publications Co., 2011