Processing of massive data sets II

Krzysztof Dembczyński

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

Bachelor studies, eighth semester
Academic year 2018/19 (summer semester)

Review of previous lectures

- Processing of massive datasets
- Evolution of database systems
- OLTP and OLAP systems
- ETL
- Dimensional modeling
- Data processing
  - Physical storage and data access
  - Materialization

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.

- **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)
  )
);
```

Data partitioning

- **Example**:

```
peopleDF
  .write
  .partitionBy("favorite_color")
  .bucketBy(42, "name")
  .saveAsTable("people-partitioned-bucketed")
```

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 ith input buffer is exhausted, read next portion from ith 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 ith input buffer is exhausted, read next portion from ith 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 ith input buffer is exhausted, read next portion from ith 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 ith input buffer is exhausted, read next portion from ith 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);
}
```

- Should be there one or many workers running the `totalWordCount` procedure?

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

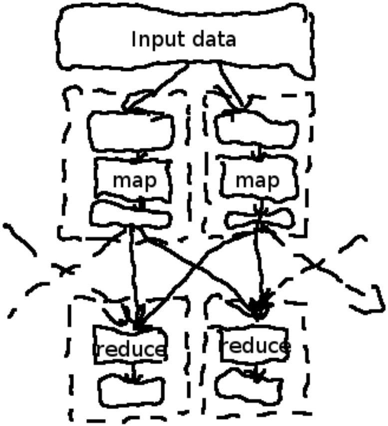
- MapReduce programs are executed in two main phases, called mapping and reducing:
  - Map: the map function is written to convert input elements to key-value pairs.
  - Reduce: the reduce function is written to take pairs consisting of a key and its list of associated values and combine those values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

MapReduce

- The complete data flow:

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

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);
  }

}
```

- 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:

- 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

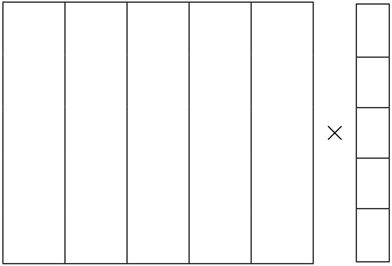
- **Map**: Each map task will take the entire vector $\boldsymbol{x}$ and a chunk of the matrix $\boldsymbol{A}$. From each matrix element a_{ij} it produces the key-value pair $(i, a_{ij}x_j)$. Thus, all terms of the sum that make up the component v_i of the matrix-vector product will get the same key.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

Matrix-Vector multiplication with large vector x

- 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 ith stripe of the matrix multiplies only components from the ith 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 x

- 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

Algorithms in MapReduce

- How to implement fundamental algorithms in MapReduce?
  - Relational-Algebra Operations.
  - Matrix multiplication.

Relational-algebra operations

Example (Relation **Links**)

From	To
`url1`	`url2`
`url1`	`url3`
`url2`	`url3`
`url2`	`url4`
...	...

Relational-algebra operations

- We assume that input and output are *real* relations (no duplicated rows)

Relational-algebra operations

- We assume that input and output are *real* relations (no duplicated rows)
- Operations:
  - Selection
  - Projection
  - Union, intersection, and difference
  - Natural join
  - Grouping and aggregation

Relational-algebra operations

- We assume that input and output are *real* relations (no duplicated rows)
- Operations:
  - Selection
  - Projection
  - Union, intersection, and difference
  - Natural join
  - Grouping and aggregation
- Notation:
  - R, S - relation
  - t, t' - a tuple
  - $\mathcal{C}$ - a condition of selection
  - A, B, C - subset of attributes
  - a, b, c - attribute values for a given subset of attributes

Selection

- **Operation**: $\mathrm{Select}_{\mathcal{C}}(R)$

Selection

- **Operation**: $\mathrm{Select}_{\mathcal{C}}(R)$
- **Map**:

Selection

- **Operation**: $\text{Select}_{\mathcal{C}}(R)$
- **Map**: For each tuple t in R, test if it satisfies $\mathcal{C}$. If so, produce the key-value pair (t, t). That is, both the key and value are t.
- **Reduce**:

Selection

- **Operation**: $\text{Select}_{\mathcal{C}}(R)$
- **Map**: For each tuple t in R, test if it satisfies $\mathcal{C}$. If so, produce the key-value pair (t, t). That is, both the key and value are t.
- **Reduce**: The Reduce function is the identity. It simply passes each key-value pair to the output.

Selection

- **Operation**: $\mathrm{Select}_{\mathcal{C}}(R)$
- **Map**: For each tuple t in R, test if it satisfies $\mathcal{C}$. If so, produce the key-value pair (t, t). That is, both the key and value are t.
- **Reduce**: The Reduce function is the identity. It simply passes each key-value pair to the output.

	Input	Output
map	<k1,t>	list(<t,t>)
reduce	(<t,list(t)>)	list(<t,t>)

Projection

- **Operation**: $\mathrm{Project}_A(R)$

Projection

- **Operation**: $\text{Project}_A(R)$
- **Map**:

Projection

- **Operation**: $\mathrm{Project}_A(R)$
- **Map**: For each tuple t in R, construct a tuple t' by eliminating from t those components whose attributes are not in A. Output the key-value pair (t', t').
- **Reduce**:

Projection

- **Operation**: $\text{Project}_A(R)$
- **Map**: For each tuple t in R, construct a tuple t' by eliminating from t those components whose attributes are not in A. Output the key-value pair (t', t').
- **Reduce**: For each key t' produced by any of the Map tasks, there will be one or more key-value pairs (t', t'). The Reduce function turns $(t', [t', t', \ldots, t'])$ into (t', t'), so it produces exactly one pair (t', t') for this key t'.

Projection

- **Operation**: $\text{Project}_A(R)$
- **Map**: For each tuple t in R, construct a tuple t' by eliminating from t those components whose attributes are not in A. Output the key-value pair (t', t').
- **Reduce**: For each key t' produced by any of the Map tasks, there will be one or more key-value pairs (t', t'). The Reduce function turns $(t', [t', t', \ldots, t'])$ into (t', t'), so it produces exactly one pair (t', t') for this key t'.

	Input	Output
map	`<k1,t>`	`list(<t’,t’>)`
reduce	`(<t’,list(t’,...,t’)>)`	`list(<t’,t’>)`

Union

- **Operation**: $\text{Union}(R, S)$

Union

- **Operation**: $\mathrm{Union}(R, S)$
- **Map**:

Union

- **Operation**: $\text{Union}(R, S)$
- **Map**: Turn each input tuple t either from relation R or S into a key-value pair (t, t).
- **Reduce**:

Union

- **Operation**: $\text{Union}(R, S)$
- **Map**: Turn each input tuple t either from relation R or S into a key-value pair (t, t).
- **Reduce**: Associated with each key t there will be either one or two values. Produce output (t, t) in either case.

Union

- **Operation**: $\text{Union}(R, S)$
- **Map**: Turn each input tuple t either from relation R or S into a key-value pair (t, t).
- **Reduce**: Associated with each key t there will be either one or two values. Produce output (t, t) in either case.

	Input	Output
map	`<k1,t)>`	`list(<t,t>)`
reduce	`(<t,list(t)>)` or `(<t,list(t,t)>)`	`list(<t,t>)`

Intersection

- **Operation**: $\mathrm{Intersection}(R, S)$

Intersection

- **Operation**: $\text{Intersection}(R, S)$
- **Map**:

Intersection

- **Operation**: $\text{Intersection}(R, S)$
- **Map**: Turn each input tuple t either from relation R or S into a key-value pair (t, t).
- **Reduce**:

Intersection

- **Operation**: $\text{Intersection}(R, S)$
- **Map**: Turn each input tuple t either from relation R or S into a key-value pair (t, t).
- **Reduce**: If key t has value list $[t, t]$, then produce (t, t). Otherwise, produce nothing.

Intersection

- **Operation**: $\text{Intersection}(R, S)$
- **Map**: Turn each input tuple t either from relation R or S into a key-value pair (t, t).
- **Reduce**: If key t has value list $[t, t]$, then produce (t, t). Otherwise, produce nothing.

	Input	Output
map	`<k1,t)>`	`list(<t,t>)`
reduce	`(<t,list(t)>)` or	`list(<t,t>)` if
	`(<t,list(t,t)>)`	`(<t,list(t,t)>)`

Minus

- **Operation**: $\text{Minus}(R, S)$

Minus

- **Operation**: $\text{Minus}(R, S)$
- **Map**:

Minus

- **Operation**: $\text{Minus}(R, S)$
- **Map**: For a tuple t in R, produce key-value pair $(t, \texttt{name}(R))$, and for a tuple t in S, produce key-value pair $(t, \texttt{name}(S))$.
- **Reduce**:

Minus

- **Operation**: $\text{Minus}(R, S)$
- **Map**: For a tuple t in R, produce key-value pair $(t, \texttt{name}(R))$, and for a tuple t in S, produce key-value pair $(t, \texttt{name}(S))$.
- **Reduce**: For each key t, do the following.
  1. If the associated value list is $[\texttt{name}(R)]$, then produce (t, t).
  2. If the associated value list is anything else, which could only be $[\texttt{name}(R), \texttt{name}(S)]$, $[\texttt{name}(S), \texttt{name}(R)]$, or $[\texttt{name}(S)]$, produce nothing.

Minus

- **Operation**: $\text{Minus}(R, S)$
- **Map**: For a tuple t in R, produce key-value pair $(t, \texttt{name}(R))$, and for a tuple t in S, produce key-value pair $(t, \texttt{name}(S))$.
- **Reduce**: For each key t, do the following.
  1. If the associated value list is $[\texttt{name}(R)]$, then produce (t, t).
  2. If the associated value list is anything else, which could only be $[\texttt{name}(R), \texttt{name}(S)]$, $[\texttt{name}(S), \texttt{name}(R)]$, or $[\texttt{name}(S)]$, produce nothing.

	Input	Output
map	`<k1,(t,R)>` or	`list(<t,R>)` or
	`<k1,(t,S)>` or	`list(<t,S>)`
reduce	`(<t,list(R)>)` or	`list(<t,t>)` if
	`(<t,list(S)>)` or	`(<t,list(R)>)`
	`(<t,list(R,S)>)` or	
	`(<t,list(S,R)>)`	

Natural Join

- **Operation**: $\text{Join}_B(R, S)$

Natural Join

- **Operation**: $\mathrm{Join}_B(R, S)$
- Assume that we join relation $R(A, B)$ with relation $S(B, C)$ that share the same attribute B.
- **Map**:

Natural Join

- **Operation**: $\text{Join}_B(R, S)$
- Assume that we join relation $R(A, B)$ with relation $S(B, C)$ that share the same attribute B.
- **Map**: For each tuple (a, b) of R, produce the key-value pair $(b, (\texttt{name}(R), a))$. For each tuple (b, c) of S, produce the key-value pair $(b, (\texttt{name}(S), c))$.
- **Reduce**:

Natural Join

- **Operation**: $\mathrm{Join}_B(R, S)$
- Assume that we join relation $R(A, B)$ with relation $S(B, C)$ that share the same attribute B.
- **Map**: For each tuple (a, b) of R, produce the key-value pair $(b, (\texttt{name}(R), a))$. For each tuple (b, c) of S, produce the key-value pair $(b, (\texttt{name}(S), c))$.
- **Reduce**: Each key value b will be associated with a list of pairs that are either of the form $(\texttt{name}(R), a)$ or $(\texttt{name}(S), c)$. Construct all pairs consisting of one with first component $\texttt{name}(R)$ and the other with first component S, say $(\texttt{name}(R), a)$ and $(\texttt{name}(S), c)$. The output for key b is a list $(b, (a1, b, c1))$, $(b, (a2, b, c2))$, $\ldots$.

Natural Join

- **Operation**: $\text{Join}_B(R, S)$
- Assume that we join relation $R(A, B)$ with relation $S(B, C)$ that share the same attribute B.
- **Map**: For each tuple (a, b) of R, produce the key-value pair $(b, (\texttt{name}(R), a))$. For each tuple (b, c) of S, produce the key-value pair $(b, (\texttt{name}(S), c))$.
- **Reduce**: Each key value b will be associated with a list of pairs that are either of the form $(\texttt{name}(R), a)$ or $(\texttt{name}(S), c)$. Construct all pairs consisting of one with first component $\texttt{name}(R)$ and the other with first component S, say $(\texttt{name}(R), a)$ and $(\texttt{name}(S), c)$. The output for key b is a list $(b, (a1, b, c1))$, $(b, (a2, b, c2))$, ...

	Input	Output
map	<k1,(t,R)> or	list(<b,(a,R)>) or
	<k1,(t,S)> or	list(<b,(c,S)>)
reduce	<b,list((a1,R), ...,	list(<b,(a1,b,c1)>,...)
	(c1,S),...)>	

Grouping and Aggregation

- **Operation**: $\text{Aggregate}_{(\theta,A,B)}(R)$

Grouping and Aggregation

- **Operation**: $\mathrm{Aggregate}_{(\theta, A, B)}(R)$
- Assume that we group a relation $R(A, B, C)$ by attributes A and aggregate values of B by using function θ.
- **Map**:

Grouping and Aggregation

- **Operation**: $\text{Aggregate}_{(\theta, A, B)}(R)$
- Assume that we group a relation $R(A, B, C)$ by attributes A and aggregate values of B by using function θ.
- **Map**: For each tuple (a, b, c) produce the key-value pair (a, b).
- **Reduce**:

Grouping and Aggregation

- **Operation**: $\text{Aggregate}_{(\theta,A,B)}(R)$
- Assume that we group a relation $R(A, B, C)$ by attributes A and aggregate values of B by using function θ.
- **Map**: For each tuple (a, b, c) produce the key-value pair (a, b).
- **Reduce**: Each key a represents a group. Apply the aggregation operator θ to the list $[b_1, b_2, \ldots, b_n]$ of B-values associated with key a. The output is the pair (a, x), where x is the result of applying θ to the list. For example, if θ is `SUM`, then $x = b_1 + b_2 + \ldots + b_n$, and if θ is `MAX`, then x is the largest of $b_1, b_2, \ldots, b_n$.

Grouping and Aggregation

- **Operation**: $\text{Aggregate}_{(\theta,A,B)}(R)$
- Assume that we group a relation $R(A,B,C)$ by attributes A and aggregate values of B by using function θ.
- **Map**: For each tuple (a,b,c) produce the key-value pair (a,b).
- **Reduce**: Each key a represents a group. Apply the aggregation operator θ to the list $[b_1, b_2, \ldots, b_n]$ of B-values associated with key a. The output is the pair (a,x), where x is the result of applying θ to the list. For example, if θ is `SUM`, then $x = b_1 + b_2 + \ldots + b_n$, and if θ is `MAX`, then x is the largest of $b_1, b_2, \ldots, b_n$.

	Input	Output
map	`<k1,t>`	`list(<a,b>)`
reduce	`<a,list((b1,b2,...)>`	`list(<a,f(b1,b2,...)>)`

Matrix Multiplication

- If M is a matrix with element m_{ij} in row i and column j, and N is a matrix with element n_{jk} in row j and column k, then the product:

$$P = MN$$

is the matrix P with element p_{ik} in row i and column k, where:

$$p_{ik} =$$

Matrix Multiplication

- If M is a matrix with element m_{ij} in row i and column j, and N is a matrix with element n_{jk} in row j and column k, then the product:

  $$P = MN$$

  is the matrix P with element p_{ik} in row i and column k, where:

  $$p_{ik} = \sum_j m_{ij} n_{jk}$$

Matrix Multiplication

- We can think of a matrix M and N as a relation with three attributes: the row number, the column number, and the value in that row and column, i.e.,:

$$M(I, J, V) \quad \text{and} \quad N(J, K, W)$$

  with the following tuples, respectively:

$$(i, j, m_{ij}) \quad \text{and} \quad (j, k, n_{jk}).$$

- In case of sparsity of M and N, this relational representation is very efficient in terms of space.
- The product MN is almost a natural join followed by grouping and aggregation.

Matrix Multiplication

Matrix Multiplication

- **Map**:

Matrix Multiplication

- **Map**: Send each matrix element m_{ij} to the key value pair:

  $$(j, (M, i, m_{ij}))\,.$$

  Analogously, send each matrix element n_{jk} to the key value pair:

  $$(j, (N, k, n_{jk}))\,.$$

- **Reduce**:

Matrix Multiplication

- **Map**: Send each matrix element m_{ij} to the key value pair:

$$(j, (M, i, m_{ij}))\,.$$

Analogously, send each matrix element n_{jk} to the key value pair:

$$(j, (N, k, n_{jk}))\,.$$

- **Reduce**: For each key j, examine its list of associated values. For each value that comes from M, say (M, i, m_{ij}), and each value that comes from N, say (N, k, n_{jk}), produce the tuple

$$(i, k, v = m_{ij} n_{jk}),$$

The output of the Reduce function is a key j paired with the list of all the tuples of this form that we get from j:

$$(j, [(i_1, k_1, v_1), (i_2, k_2, v_2), \ldots, (i_p, k_p, v_p)])\,.$$

Matrix Multiplication

Matrix Multiplication

- **Map**:

Matrix Multiplication

- **Map**: From the pairs that are output from the previous Reduce function produce p key-value pairs:

$$((i_1, k_1), v_1)\,, ((i_2, k_2), v_2)\,, \ldots, ((i_p, k_p), v_p)\,.$$

- **Reduce**:

Matrix Multiplication

- **Map**: From the pairs that are output from the previous Reduce function produce p key-value pairs:

$$((i_1, k_1), v_1), ((i_2, k_2), v_2), \ldots, ((i_p, k_p), v_p).$$

- **Reduce**: For each key (i, k), produce the sum of the list of values associated with this key. The result is a pair

$$((i, k), v),$$

where v is the value of the element in row i and column k of the matrix

$$P = MN.$$

Matrix Multiplication with One Map-Reduce Step

- **Map**:

Matrix Multiplication with One Map-Reduce Step

- **Map**: For each element m_{ij} of M, produce a key-value pair

$$((i,k),(M,j,m_{ij}))\,,$$

  for $k = 1, 2, \ldots$, up to the number of columns of N.
  Also, for each element n_{jk} of N, produce a key-value pair

$$((i,k),(N,j,n_{jk}))\,,$$

  for $i = 1, 2, \ldots$, up to the number of rows of M.

Matrix Multiplication with One Map-Reduce Step

- **Reduce**:

Matrix Multiplication with One Map-Reduce Step

- **Reduce**: Each key (i, k) will have an associated list with all the values

$$(M, j, m_{ij}) \quad \text{and} \quad (N, j, n_{jk}),$$

  for all possible values of j. We connect the two values on the list that have the same value of j, for each j:
  - We sort by j the values that begin with M and sort by j the values that begin with N, in separate lists,
  - The jth values on each list must have their third components, m_{ij} and n_{jk} extracted and multiplied,
  - Then, these products are summed and the result is paired with (i, k) in the output of the Reduce function.

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.
- Algorithms Using Map-Reduce

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