

ISIT312 Big Data Management

Cluster Computing

Dr Guoxin Su and Dr Janusz R. Getta

School of Computing and Information Technology -
University of Wollongong

Cluster Computing

Outline

Big Data

Traditional Data Architectures

Meet Hadoop !

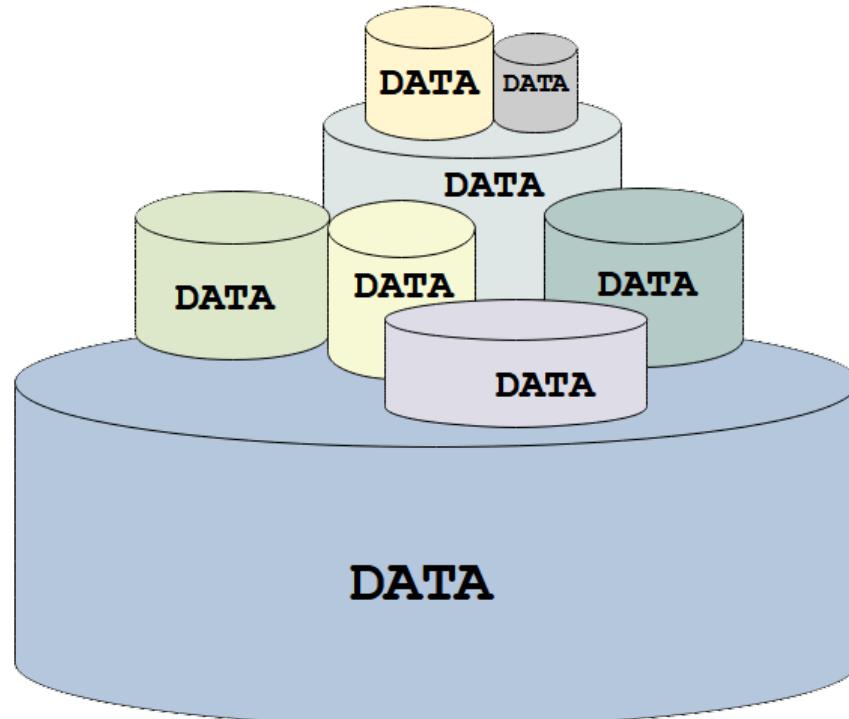
[TOP](#)

ISIT312 Big Data Management, SIM, Session 2, 2021

2/20

Big Data

What does **Big Data** mean and how big is **Big Data** ?



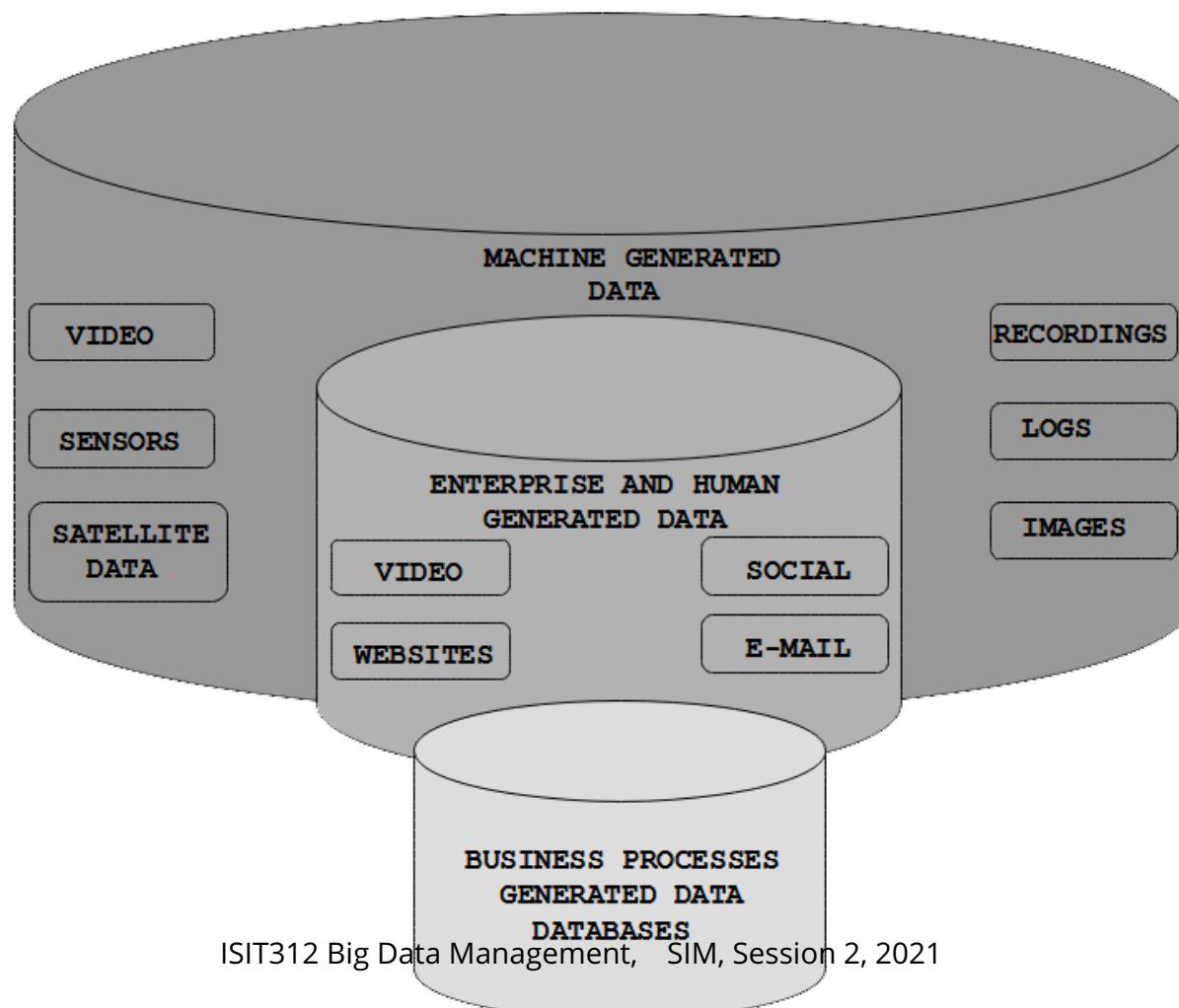
Big Data is so big that it cannot be stored on the persistent storage devices attached to a single computer system

[TOP](#) **Big Data** may also mean **an infinite amount of data**

3/20

Big Data

What are the source of **Big Data** ?



[TOP](#)

Big Data

Big Data is characterized by so called 3V features:

- **Volume**: e.g., billions of rows ? millions of columns
- **Variety**: Complexity of data types and structures
- **Velocity**: Speed of new data creation and growth

Additional Vs:

- **Veracity**: Ability to represent and process uncertain and imprecise data
- **Value**: Data is the driving force of the next-generate business
- **Viability**: Benefits we can potentially have from data analysis

There are many, many other Vs, the largest number of Vs I found on Web was 42 !

- **Vagueness**: The meaning of found data is often very unclear, regardless of how much data is available
- **Validity**: Rigor in analysis is essential for valid predictions where data is the driving force of the next-generate business
- **Vane**: Data science can aid decision making by pointing in the correct direction
- and many, many others

Big Data

Examples of Big Data:

- Clickstream data
- Call centre data
- E-mail and instant-messaging
- Sensor data
- Unstructured data
- Geographic data
- Satellite data
- Image data
- Temporal data
- and more ...

Cluster Computing

Outline

Big Data

Traditional Data Architectures

Meet Hadoop !

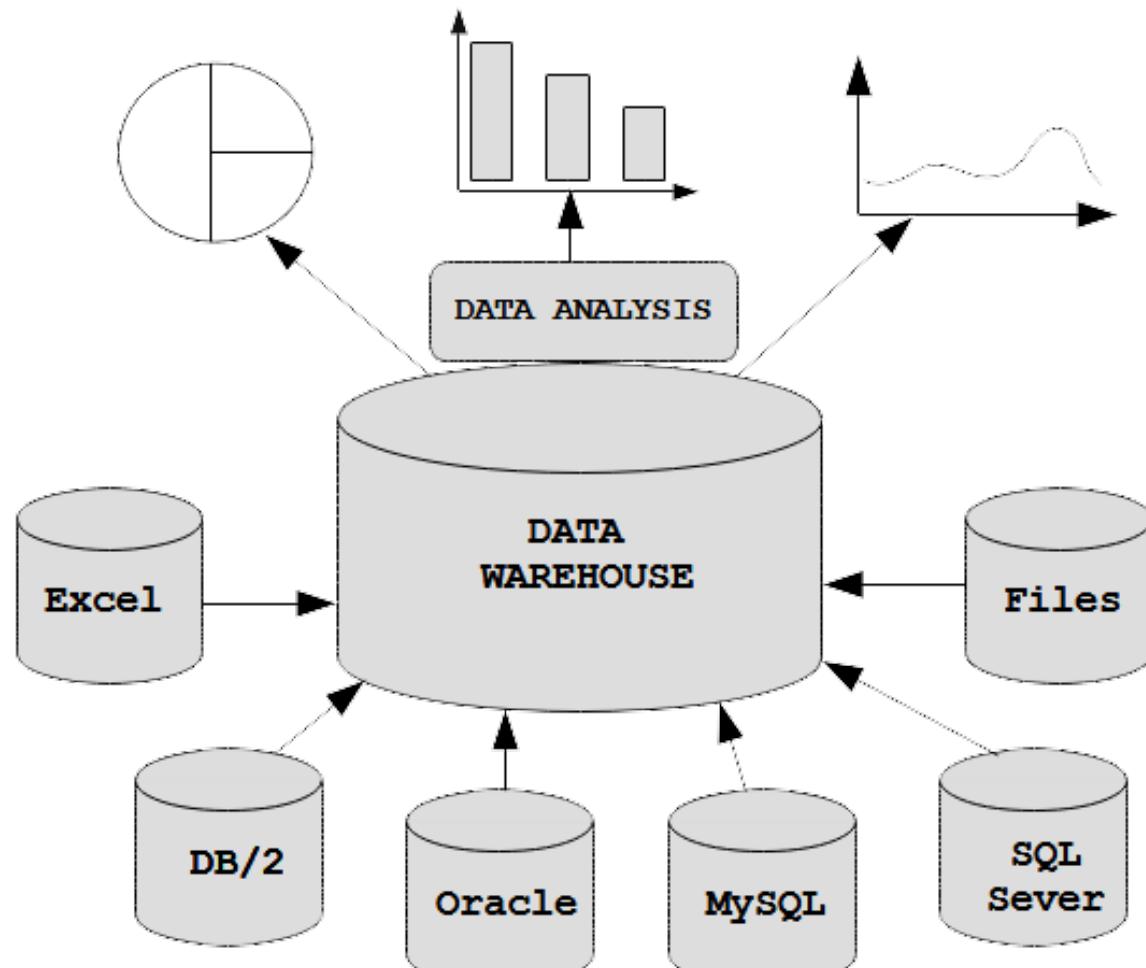
[TOP](#)

ISIT312 Big Data Management, SIM, Session 2, 2021

7/20

Traditional Data Architectures

Data warehousing technologies



Traditional Data Architectures

The strength of **traditional data architectures**:

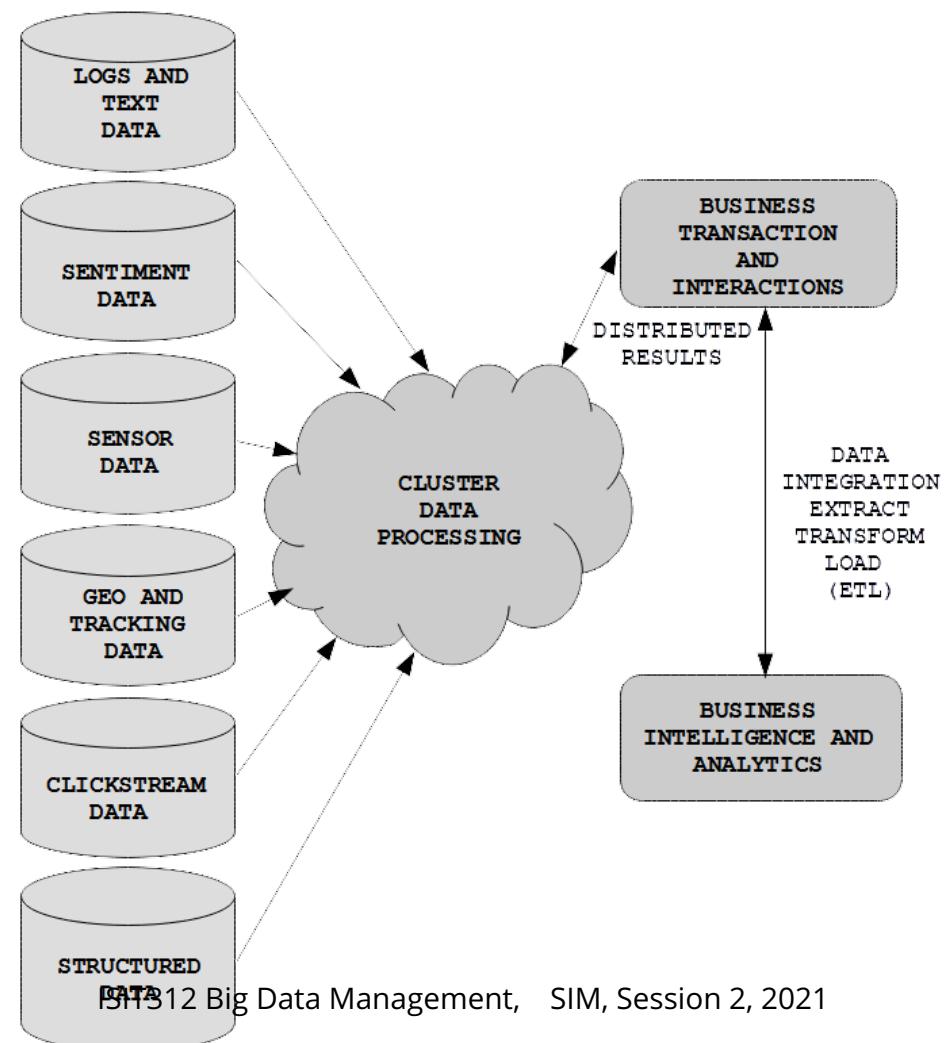
- Centralised governance of data repositories
- Light-fast inquiries performed regularly in daily business
- Optimisation for OLTP and OLAP
- Security and access control
- Fault-Tolerance and backup

The challenges for **traditional data architectures**:

- New types of data such as unstructured data and semi-structured data
- Increasingly large amounts of data flowing into organisations
- New computational paradigms use non-traditional NoSQL databases to rapidly mine and analyse very large data sets
- Increasing cost of storing and analysing the large amounts of data
- Increasing use of data analytics, which requires significant storage and processing capabilities

Traditional Data Architectures

A sample **Data Lake** architecture



Traditional Data Architectures

Hardware for **Big Data** has two scalability dimensions



Cluster Computing

Outline

Big Data

Traditional Data Architectures

Meet Hadoop !

Meet Hadoop !

Hadoop, in terms of its developers, is a project that develops open-source software for reliable, scalable, distributed computing

Features of Hadoop

- Capability to handle large data sets, e.g. simple scalability and coordination
- File size range from gigabytes to terabytes
- Can store millions of those files
- High fault tolerance
- Supports data replication
- Supports streaming access to data
- Supports batch processing
- Support interactive, iterative and stream processing
- Implements a data consistency model of [write-once-read-many](#) access model
- Run on commodity hardware, not high-performance computers
- Inexpensive
- It can be deployed on [premises or in the cloud](#)

[TOP](#)

ISIT212 Big Data Management | SIM Session 2, 2021

13/20

Meet Hadoop !

Core components of Hadoop

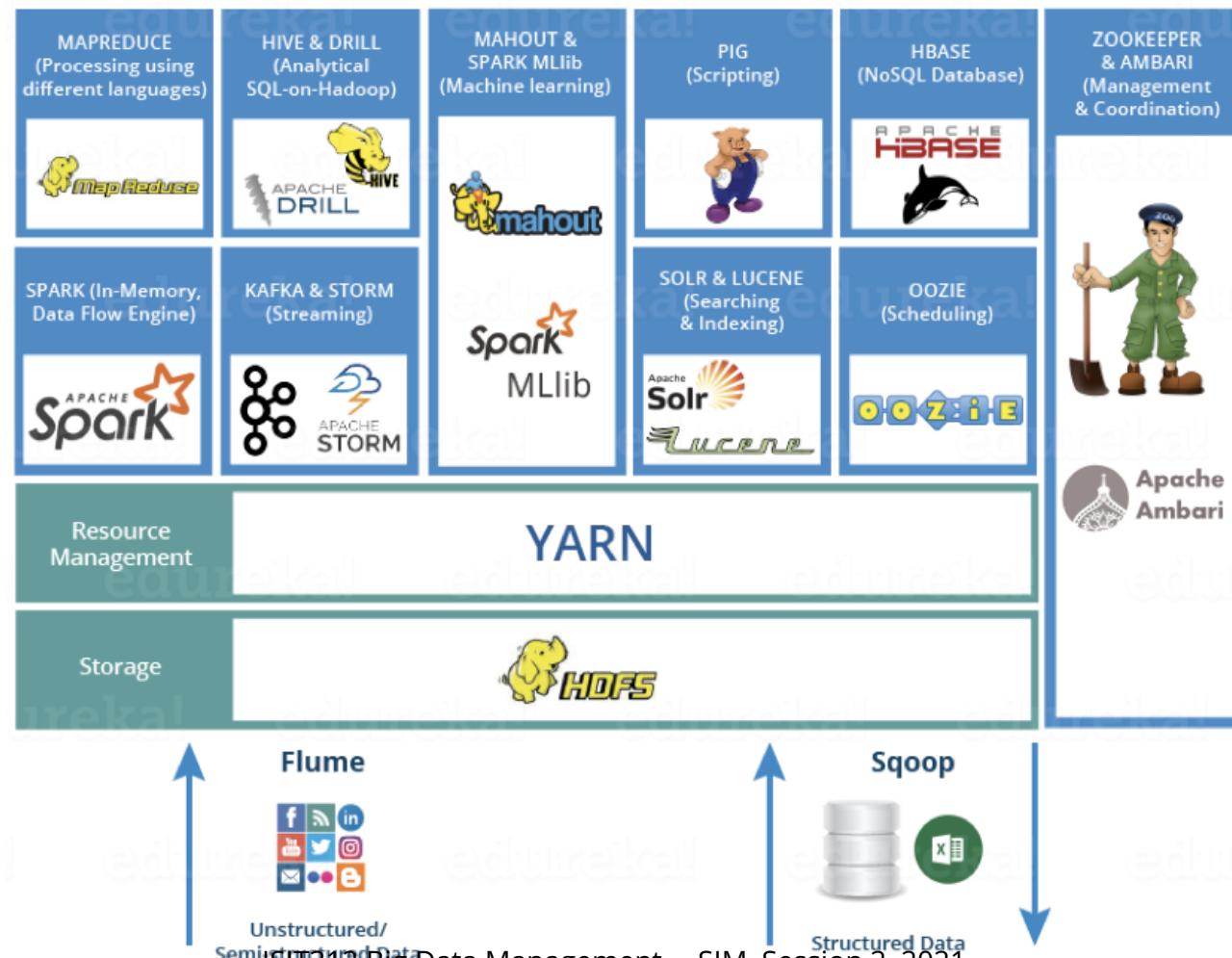
Different data-processing frameworks
(e.g., MapReduce)

YARN: An Operating System for Hadoop
(Hadoop Cluster Resource Management)

HDFS
(Hadoop Distributed File System)

Hadoop Ecosystem

Hadoop ecosystem



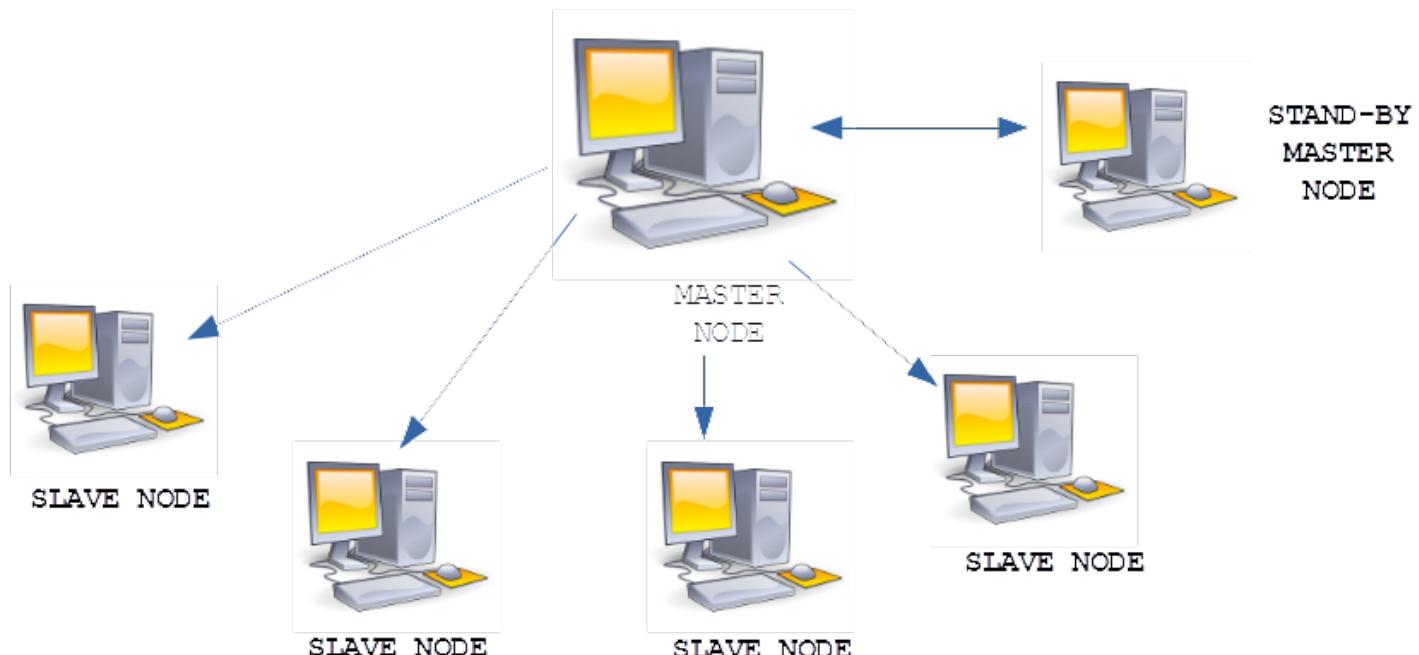
Commercial Hadoop Landscape

Commercial Hadoop landscape



Meet Hadoop !

Master-slave architecture of Hadoop clusters



Meet Hadoop !

Hadoop clusters can support up to 10,000 server and receives near-to-linear scalability in computing power

A typical Hadoop cluster consists of:

- A set of **master nodes** (servers) where the daemons supporting key Hadoop frame-works run
- A set of **worker nodes** that host the storage (HDFS) and computing (YARN) work
- One or more **edge servers**, which are used for accessing the Hadoop cluster to launch applications
- One or more **relational databases** such as MySQL for storing the metadata repositories
- **Dedicated servers** for special frameworks such as Kafka

Meet Hadoop !

Hadoop also support the pseudo-distributed mode

- All HDFS and YARN daemons running on a single node.
- Highly simulate the full cluster
- Easy for beginner's practice
- Easy for testing and debug

Our lab setting is the pseudo-distributed mode

- The single node is a Ubuntu 14.04 Virtual Machine (VM)

References

White T., Hadoop The Definitive Guide: Storage and analysis at Internet scale, O'Reilly, 2015 (Available through UOW library)

Vohra D., Practical Hadoop ecosystem: a definitive guide to Hadoop-related frameworks and tools, Apress, 2016 (Available through UOW library)

Aven J., Hadoop in 24 Hours, SAMS Teach Yourself, SAMS 2017

Alapati S. R., Expert Hadoop Administration: Managing, tuning, and securing Spark, YARN and HDFS, Addison-Wesley 2017

ISIT312 Big Data Management

MapReduce Framework

Dr Guoxin Su and Dr Janusz R. Getta

School of Computing and Information Technology -
University of Wollongong

MapReduce Framework

Outline

MapReduce

Real world scenario: log data analysis

MapReduce implementation in Hadoop

MapReduce

MapReduce is the most important processing framework in Hadoop

Many high-level data processing languages are abstractions of MapReduce, e.g. Pig and Hive or are heavily influenced by MapReduce concepts e.g. Spark

Historically, Hadoop version 1 supported MapReduce only

MapReduce is also a platform and language-independent programming model at the heart of most big data and NoSQL platforms

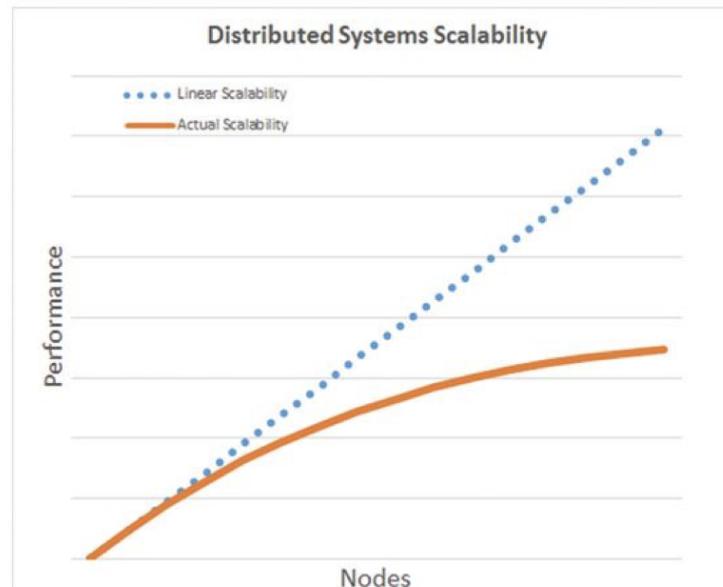
A programming model means a pattern/format in accordance to which we write our programs

The logic of a MapReduce application consists of a Map phase and a Reduce phase

MapReduce

Limitations of early distributed computing and grid computing frameworks:

- Complexity in parallel programming
- Hardware failures
- Bottlenecks in data exchange
- Scalability problem



MapReduce

The 2004 Google MapReduce white papers determined the following design goals of MapReduce

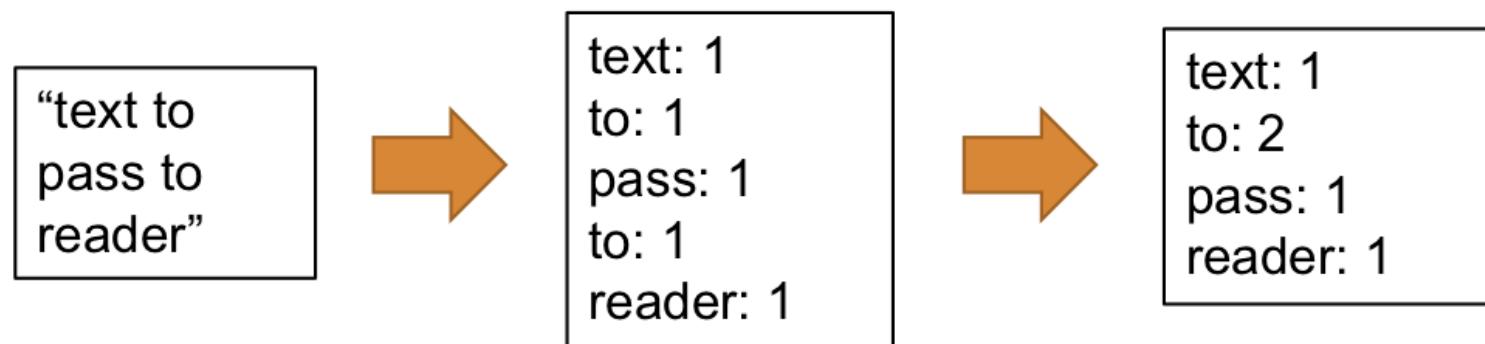
- Automatic parallelization and distribution
- Fault tolerance
- Input/output (I/O) scheduling
- Status and monitoring

MapReduce

MapReduce model uses **key-value** pairs for processing data

Key	Value
City	Sydney
Date	[02-03-2017, 02-04-2016]

WordCount: MapReduce Hello World example



MapReduce Framework

Outline

MapReduce

Real world scenario: log data analysis

MapReduce implementation in Hadoop

A Real-World scenario: log data analysis

In online purchasing, users sometimes abandon their shopping carts before completing the purchase

In order to improve their business, companies are usually interested to find out more about the nature of these abandoned purchases

A MapReduce job for this analysis

1. The final pages visited by the users
2. The contents of the abandoned shopping carts
3. The user session's transaction state

Map phase

1. aggregated data for the total number of abandoned carts
2. the most common final page visited by the users when they ended their website visit, abandoning their shopping carts.

Reduce phase

MapReduce Framework

Outline

MapReduce

Real world scenario: log data analysis

MapReduce implementation in Hadoop

MapReduce implementation in Hadoop

Hadoop MapReduce frees the users from the low-level communication and coordination of nodes and processes

Let programmers focus on the MapReduce implementation and a few configuration parameters

As the data file is usually too large to be stored in a single disk (of the commodity hardware), Hadoop handles the shipment of code to data fragments (aka, data locality)

This can dramatically reduces the overhead of network transmits

MapReduce implementation in Hadoop

Why Hadoop is useful to Big Data ?

- Cost-effective fault-tolerant storage (HDFS)
- Scalability
- Data that is ingested may be interpreted at runtime
- Low cost in storing unstructured and semi-structured data
- Fast transfer of data into storage
- Separation of programming logic and scheduling/management
- Multiple levels of distributed system abstractions: Hive, Pig, Spark
- Multi-language tooling: Java: MapReduce; SQL: Hive; data-flow: Pig; Scala, Python: Spark;

ISIT312 Big Data Management

Hadoop Architecture

Dr Guoxin Su and Dr Janusz R. Getta

School of Computing and Information Technology -
University of Wollongong

Hadoop Architecture

Outline

Hadoop Distributed File System (HDFS)

NameNode metadata

DataNode and secondary node

Yet Another Resource Negotiator (YARN)

ResourceManger

NodeManager

ApplicationMaster

HDFS: Hadoop Distributed File System

HDFS is designed for:

- Very large files
- Stream data access
- Commodity hardware

But not for:

- Low-latency data access
- Lots of small files
- Multiple writers, arbitrary file modifications

HDFS: Hadoop Distributed File System

HDFS contains the following key components:

NameNode:

- HDFS master node process
- manages the filesystem metadata
- does not store a file itself

SecondaryNameNode and Standby NameNode

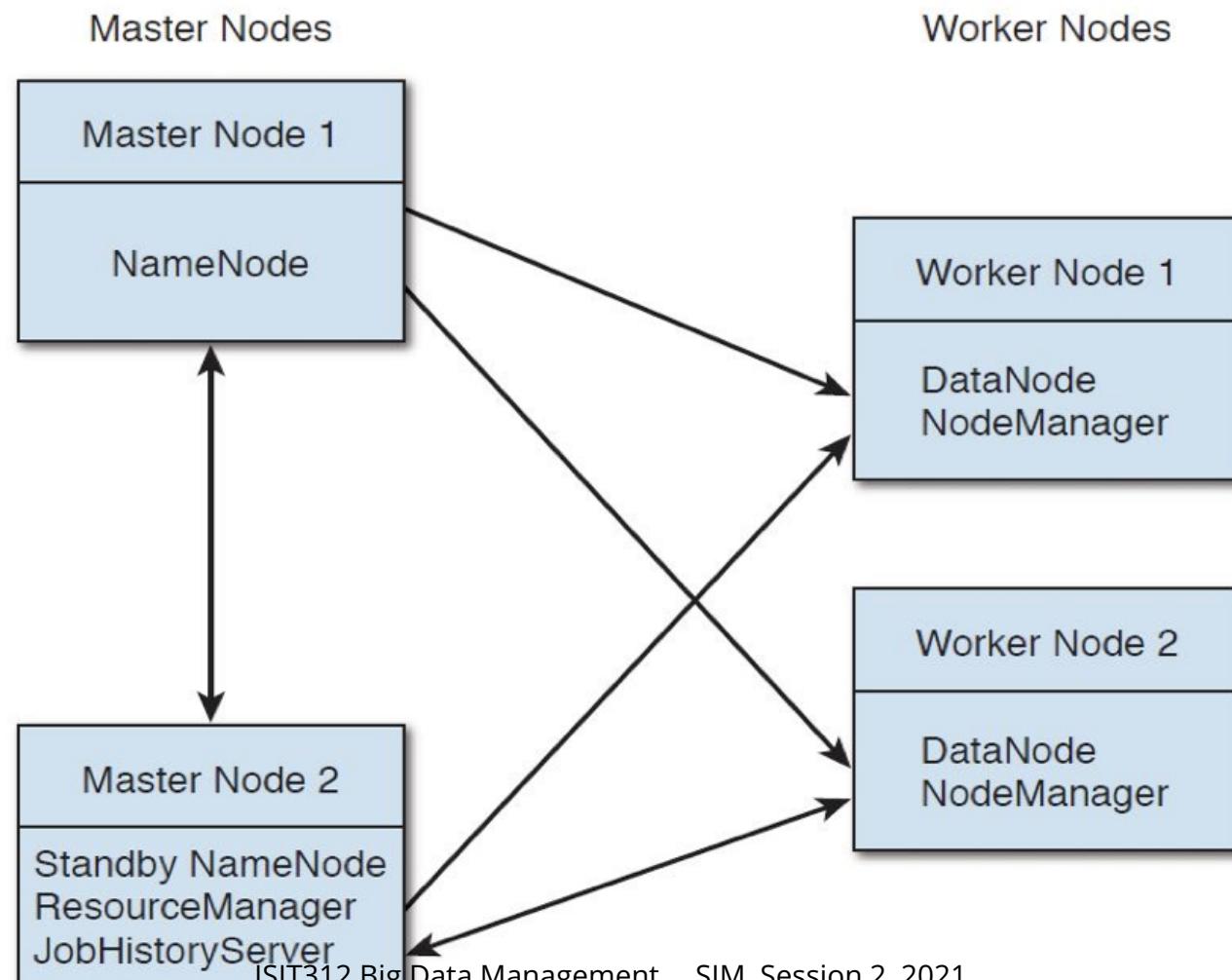
- SecondaryNameNode expedites the filesystem metadata recovery
- Standby NameNode (optional) provides high availability

DataNode

- runs HDFS slave node process
- manages block storage and access for reading or writing of data, block replication

HDFS: Hadoop Distributed File System

Architecture of HDFS



HDFS: Hadoop Distributed File System

HDFS is a virtual filesystem

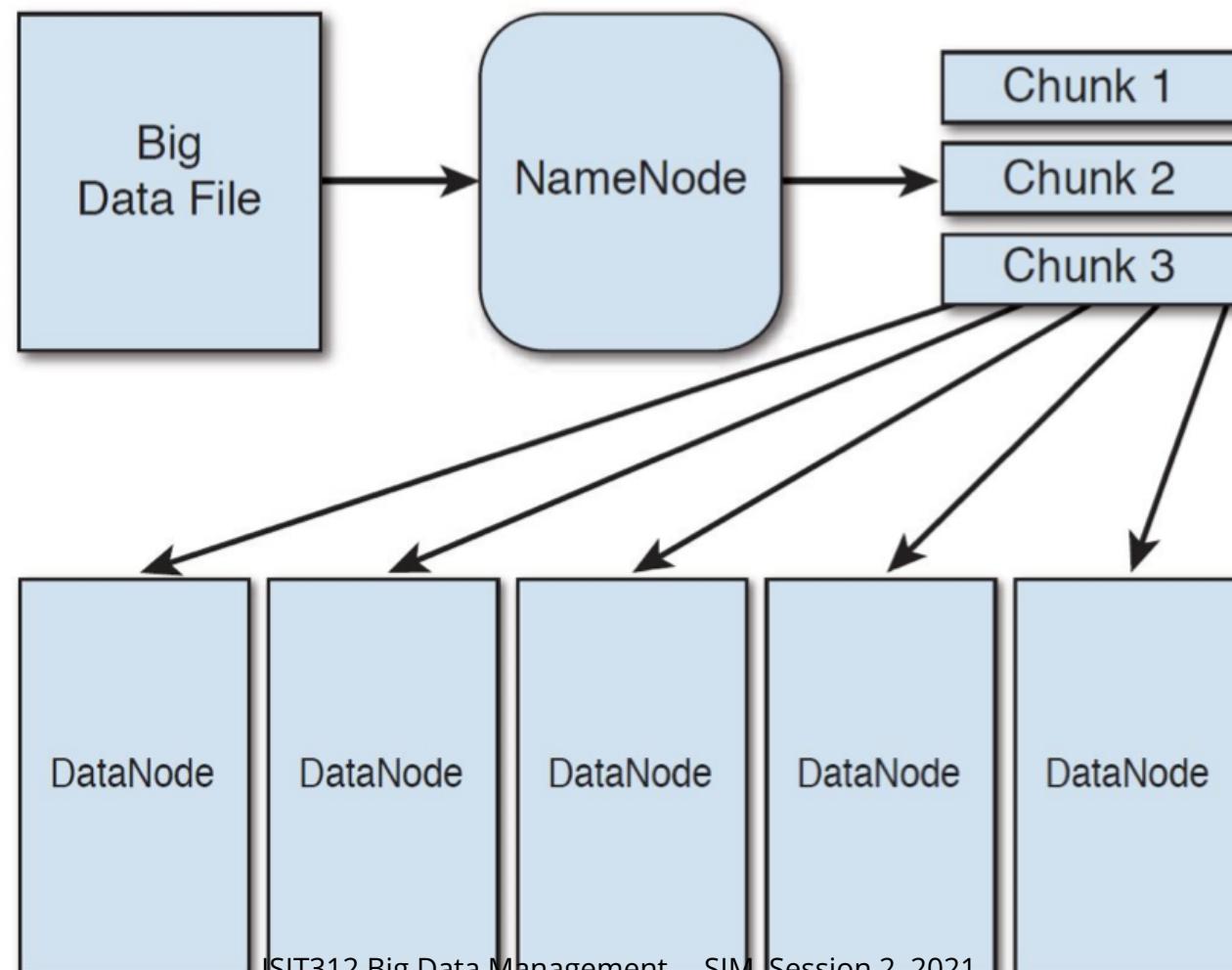
- appears to a client as if one system, but the data are stored in multiple different locations
- deployed on top of the native filesystems (such as ext3, ext4 and xfs in Linux)

Each file in **HDFS** consists of blocks

- The size of each block defaults to 128MB but is configurable
- The default number of replicates for blocks is 3, but also configurable

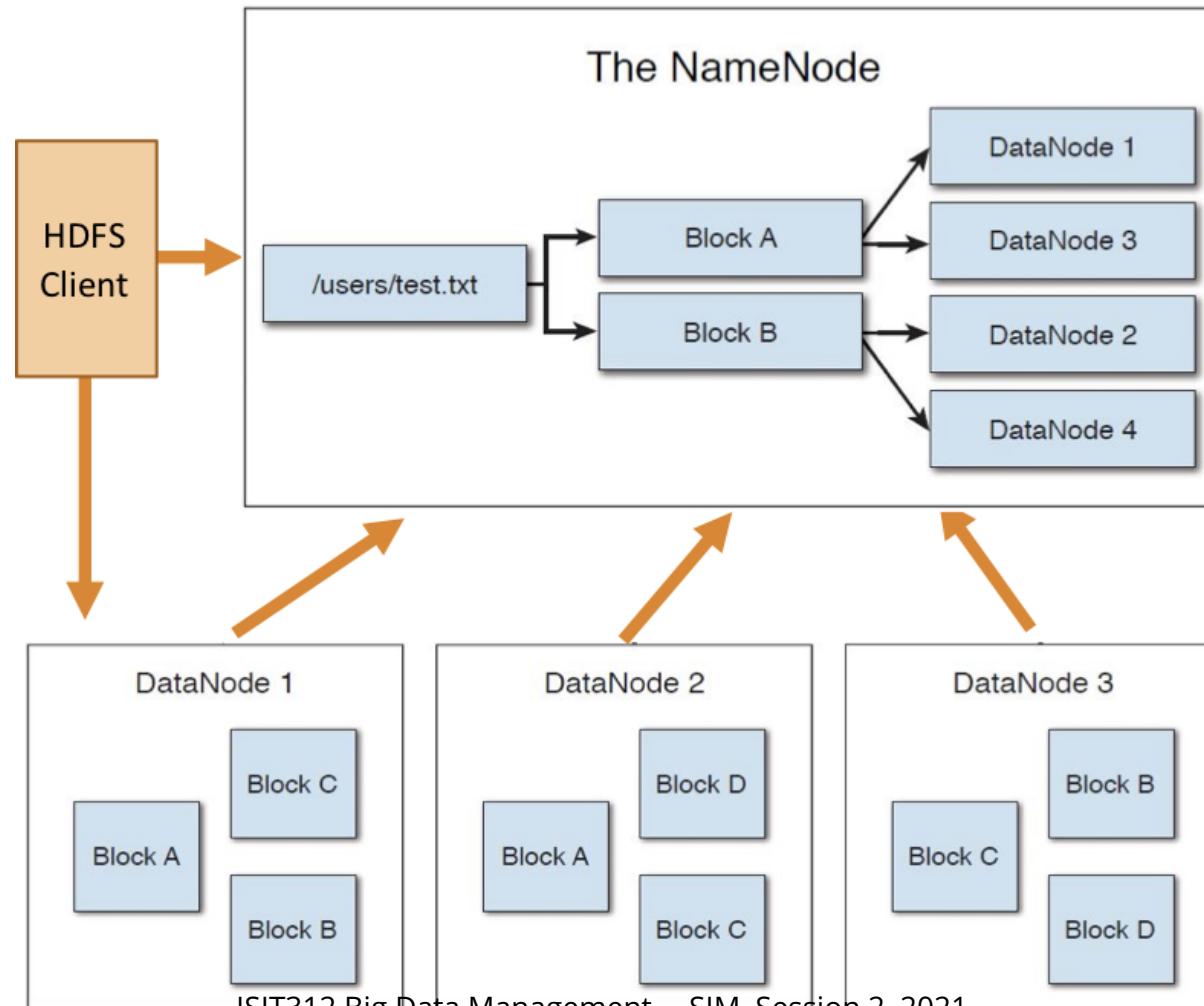
HDFS: Hadoop Distributed File System

Logical view of data storage



HDFS: Hadoop Distributed File System

Physical implementation of data file storage



Hadoop Architecture

Outline

Hadoop Distributed File System (HDFS)

NameNode metadata

DataNode and secondary node

Yet Another Resource Negotiator (YARN)

ResourceManger

NodeManager

ApplicationMaster

NameNode Metadata

NameNode stores the metadata of the files in **HDFS**

object	block_id	seq	locations	ACL	Checksum
/data/file.txt	blk_00123	1	[node1,node2,node3]	-rwxrwxrwx	8743b52063..
/data/file.txt	blk_00124	2	[node2,node3,node4]	-rwxrwxrwx	cd84097a65..
/data/file.txt	blk_00125	3	[node2,node4,node5]	-rwxrwxrwx	d1633f5c74..

NameNode functions:

- Maintain the metadata pertaining to the file system (e.g., the file hierarchy and the block locations for each file)
- Manage user access to the data files
- Map the data blocks to the **DataNodes** in the cluster
- Perform file system operations (e.g., opening and closing the files and directories)
- Provide registration services and periodic heartbeats for **DataNodes**

Hadoop Architecture

Outline

Hadoop Distributed File System (HDFS)

NameNode metadata

DataNode and Secondary node

Yet Another Resource Negotiator (YARN)

ResourceManger

NodeManager

ApplicationMaster

DataNode and Secondary node

DataNode functions:

- Provide the block storage by storing blocks on the local file system
- Fulfil the read/write requests
- Replicating data across the cluster
- Keeping in touch with the NameNode by sending periodic block reports and heartbeats
- A heartbeat confirms the DataNode is alive and healthy, and a block report shows the blocks being managed by the DataNode

Secondary NameNode and Standby NameNode functions:

- Without a **NameNode**, there is no way to know to which files the blocks stored on the **DataNodes** correspond to
- In essence, all files in **HDFS** are lost
- **Secondary NameNode** periodically backups the metadata in the (primary) **NameNode**, which is usually for recovery
- **Standby NameNode** is a hot node that runs together with the (primary) **NameNode** in the cluster, facilitating high availability

Hadoop Architecture

Outline

Hadoop Distributed File System (HDFS)

NameNode metadata

DataNode and secondary node

Yet Another Resource Negotiator (YARN)

ResourceManger

NodeManager

ApplicationMaster

Yet Another Resource Negotiator (YARN)

YARN: the core subsystem in Hadoop responsible for governing, allocating, and managing the finite distributed processing resources available on a Hadoop cluster

- introduced in Hadoop 2 to improve the MapReduce implementation, but general enough to support other distributed computing paradigms

YARN provides its core services via two types of long-running daemons:

- A **ResourceManager** (one per cluster) to manage the use of resources across the cluster, and
- **NodeManagers** running on all the nodes in the cluster to launch and monitor containers

Yet Another Resource Negotiator (YARN)

Architecture of [YARN](#)

A client is the program that submits jobs to the cluster

- May also be the gateway machine that the client program runs on

A job, also called an application, contains one or more tasks

- A task in a MapReduce job can be either a mapper and a reducer task

Each mapper and reducer takes runs within a container

- Containers are logical constructs that represent a specific amount of memory and other resources, such as the processing cores (CPU)
- For example, a container can represent 2GB memory and 2 processing cores
- Containers may also refer to the running environment of an application

Yet Another Resource Negotiator (YARN)

Architecture of YARN

ResourceManager: YARN's daemon running in a master node

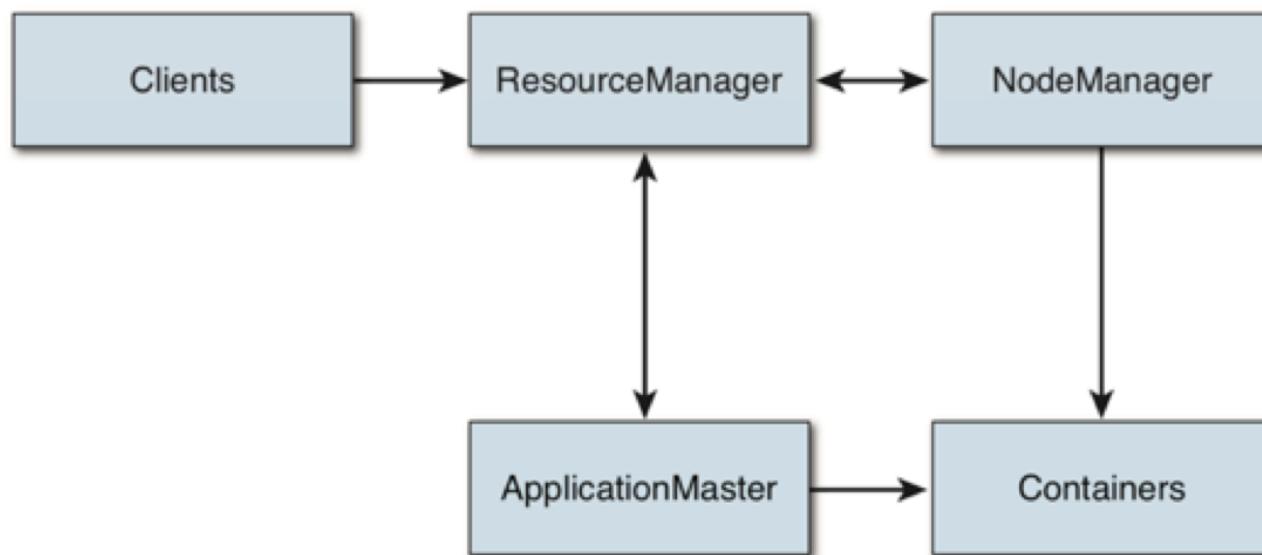
- **ResourceManager** is responsible for granting cluster computing resources to applications running on the cluster
- Resources are granted items of containers

NodeManager: YARN's daemon running in a slave node.

- **NodeManager** manages containers on a slave node
- **ApplicationMaster:** the first container allocated by the **ResourceManager** to run on a **NodeManager** for each application

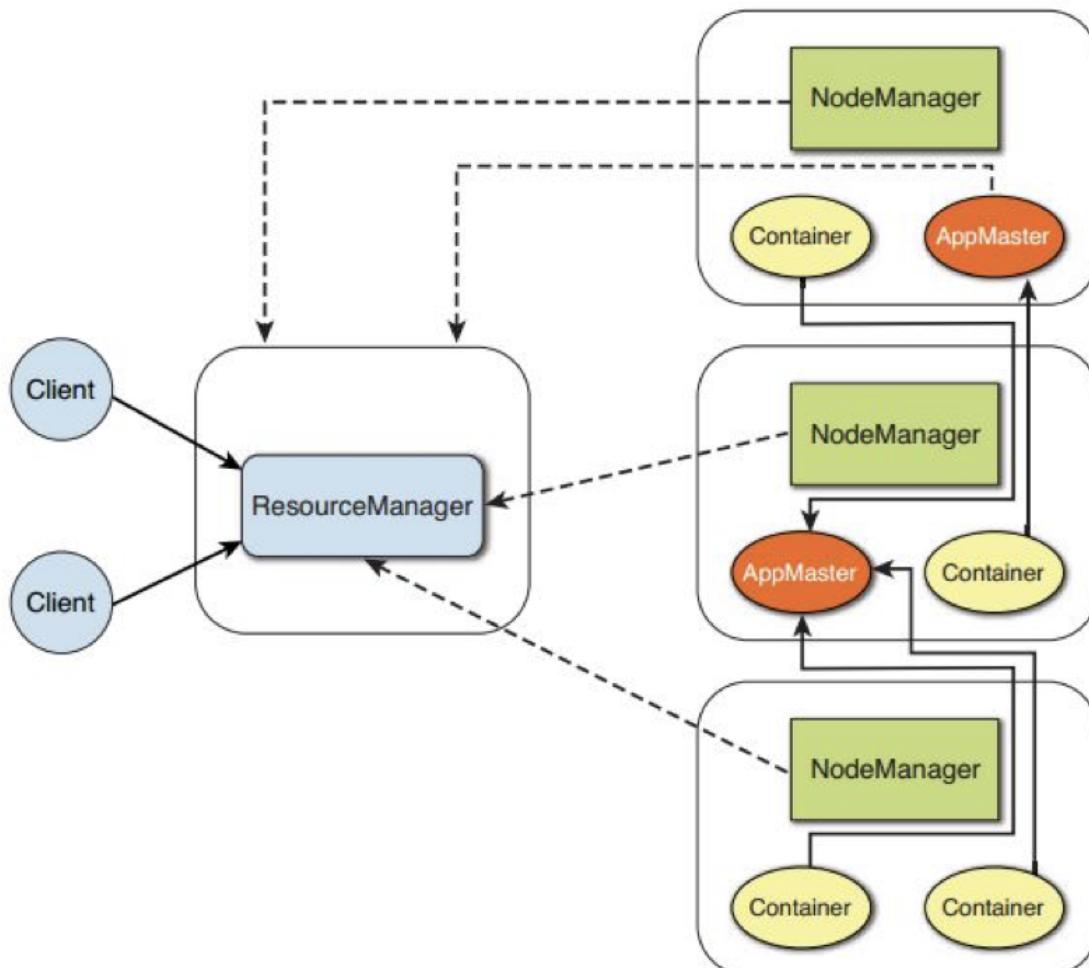
Yet Another Resource Negotiator (YARN)

Architecture of YARN



Yet Another Resource Negotiator (YARN)

Architecture of YARN



[TOP](#)

ISIT312 Big Data Management, SIM, Session 2, 2021

18/25

Hadoop Architecture

Outline

Hadoop Distributed File System (HDFS)

NameNode metadata

DataNode and secondary node

Yet Another Resource Negotiator (YARN)

ResourceManger

NodeManager

ApplicationMaster

ResourceManager

There is one **ResourceManager** per cluster, which consists of two key components: **Scheduler** and **ApplicationManager**

Key functions of **ResourceManager**:

- Creates the first container for an application to run **ApplicationMaster** for that application
- Tracks the heartbeats from **NodeManagers** to manage **DataNodes**
- Runs **Scheduler** to determine resource allocation among the clusters
- Manages cluster level security
- Manages the resource requests from **ApplicationMasters**
- Monitors the status of **ApplicationMasters** and restarts that container upon its failure
- Deallocates the containers when the application completes or after they expire

The role of **ResourceManager** is pure management and scheduler

It does not perform any actual data processing, e.g., the Map and Reduce functions in a **MapReduce** application

[TOP](#)

ISIT312 Big Data Management, SIM, Session 2, 2021

20/25

Hadoop Architecture

Outline

Hadoop Distributed File System (HDFS)

NameNode metadata

DataNode and secondary node

Yet Another Resource Negotiator (YARN)

ResourceManger

NodeManager

ApplicationMaster

NodeManager

Each **DataNode** runs a **NodeManager** daemon for performing **YARN** functions

Main functions of a **NodeManager** daemon:

- Communicates with **ResourceManager** through health heartbeats and container status notifications.
- Registers and starts the application processes
- Launches both **ApplicationMaster** and the rest of an application's resource containers (that is, the map and reduce tasks that run in the containers) on request from **ApplicationMaster**
- Oversees the lifecycle of the application containers
- Monitors, manages and provides information regarding the resource consumption (CPU/memory) by the containers
- Tracks the health of **DataNode**
- Provides auxiliary services to **YARN** applications, such as services used by the MapReduce framework for its shuffle and sort operations

Hadoop Architecture

Outline

Hadoop Distributed File System (HDFS)

NameNode metadata

DataNode and secondary node

Yet Another Resource Negotiator (YARN)

ResourceManger

NodeManager

ApplicationMaster

ApplicationMaster

For each **YARN** application, there is a dedicated **ApplicationMaster**

Functions of **ApplicationMaster**:

- Managing task scheduling and execution

- Allocating resources locally for the application's tasks

ApplicationMaster is running within a container

ApplicationMaster's existence is associated with the running application

When an application is completed, its **ApplicationMaster** no longer exists

Once created, **ApplicationMaster** is in charge of requesting resources with **ResourceManager** to run the application

The resource request are very specific, for example:

- the file blocks needed to process the job,
- the amount of the resource, in terms of the number of containers to create for the application,
- the size of the containers, etc.

Summary

Terminologies

*For convenience, we use the **names of HDFS and YARN processes** to refer to both the **hosts** and the **daemons** running on the corresponding hosts*

*For example, **RecourseManager** refers to both a **master node** and the **RecourseManager daemon** on that master node; **DataNode** refers to both a **slave node** and the **DataNode daemon** on that slave node.*

Hadoop is a leading platform for big data

Hadoop consists of a storage layer (**HDFS**), a coordination and management layer (**YARN**) and a processing layer (e.g., **MapReduce**)

HDFS and **YARN** have key services (daemons)

MapReduce is a fundamental computing model (i.e., batch processing) for big data

Next: Interaction with **Hadoop**; dive into the **MapReduce** framework

[TOP](#)

ISIT312 Big Data Management, SIM, Session 2, 2021

25/25

ISIT312 Big Data Management

HDFS Interfaces

Dr Guoxin Su and Dr Janusz R. Getta

School of Computing and Information Technology -
University of Wollongong

HDFS Interfaces

Outline

[Hadoop Cluster vs. Pseudo-Distributed Hadoop](#)

[Shell Interface to HDFS](#)

[Web Interface to HDFS](#)

[Java Interface to HDFS](#)

[Internals of HDFS](#)

Hadoop Cluster vs. Pseudo-Distributed Hadoop

A **Hadoop** cluster is deployed in a cluster of computer nodes

- As **Hadoop** is developed in Java, all Hadoop services sit on Java Virtual Machines running on the cluster nodes

Hadoop provides a pseudo-distributed mode on a single machine

- All Java Virtual Machines for necessary **Hadoop** services are running on a single machine
- In our case this machine is a Virtual Machine running under Ubuntu 14.04

HDFS provides the following interfaces to read, write, interrogate, and manage the filesystem

- The file system shell (Command-Line Interface): **hadoop fs** or **hdfs dfs**
- **Hadoop** Filesystem Java API
- **Hadoop** simple Web User Interface
- Other interfaces, such as RESTful proxy interfaces (e.g.,HttpFS)

HDFS Interfaces

Outline

[Hadoop Cluster vs. Pseudo-Distributed Hadoop](#)

[Shell Interface to HDFS](#)

[Web Interface to HDFS](#)

[Java Interface to HDFS](#)

[Internals of HDFS](#)

Shell Interface to HDFS

Commands are provided in the shell Bash

```
$ which bash  
/bin/bash
```

Bash shell

Hadoop's home directory

```
$ cd $HADOOP_HOME  
$ ls  
bin include libexec logs README.txt share  
etc lib LICENSE.txt NOTICE.txt sbin
```

Home of Hadoop

You will mostly use scripts in the **bin** and **sbin** folders, and use jar files in the **share** folder

Hadoop Daemons

```
$ jps  
28530 SecondaryNameNode  
11188 NodeManager  
28133 NameNode  
28311 DataNode  
10845 ResourceManager  
3542 Jps
```

Hadoop daemons

[TOP](#)

ISIT312 Big Data Management, SIM, Session 2, 2021

5/27

Hadoop is running properly only if the above services are running

Shell Interface to HDFS

Create a [HDFS](#) user account (already created in a virtual machine used by us)

```
$ bin/hadoop fs -mkdir -p /user/bigdata
```

Creating home of user account

Create an folder [input](#)

```
$ bin/hadoop fs -mkdir input
```

Creating a folder

View the folders in Hadoop home

```
$ bin/hadoop fs -ls  
Found 1 item  
drwxr-xr-x - bigdata supergroup 0 2017-07-17 16:33 input
```

Listing home of user account

Upload a file to [HDFS](#)

```
$ bin/hadoop fs -put README.txt input  
$ bin/hadoop fs -ls input  
-rw-r--r-- 1 bigdata supergroup 1494 2017-07-12 17:53 input/README.txt
```

Uploading a file

Read a file in [HDFS](#)

```
$ bin/hadoop fs -cat input/README.txt  
<contents of README.txt goes here>
```

Listing a file

[TOP](#)

ISYE312 Big Data Management, SIM, Session 2, 2021

6/27

Shell Interface to HDFS

The path in **HDFS** is represented as a URI with the prefix **hdfs://**

For example

- **hdfs://<hostname>:<port>/user/bigdata/input** refers to the **input** directory in **HDFS** under the user of **bigdata**
- **hdfs ://<hostname>:<port>/user/bigdata/input /README.txt** refers to the file **README.txt** in the above **input** directory in **HDFS**

When interacting with **HDFS** interface in the default setting, one can omitt IP, port, and user, and simply mention the directory or file

Thus, the full-spelling of **hadoop fs -ls input** is

```
hadoop fs -ls hdfs://<hostname>:<port>/user/bigdata  
/input
```

Shell Interface to HDFS

Some of frequently used commands

Commands of Hadoop shell interface

Command	Description
-put	Upload a file (or files) from the local filesystem to HDFS
-mkdir	Create a directory in HDFS
-ls	List the files in a directory in HDFS
-cat	Read the content of a file (or files) in HDFS
-copyFromLocal	Copy a file from the local filesystem to HDFS (similar to put)
-copyToLocal	Copy a file (or files) from HDFS to the local filesystem
-rm	Delete a file (or files) in HDFS
-rm -r	Delete a directory in HDFS

HDFS Interfaces

Outline

[Hadoop Cluster vs. Pseudo-Distributed Hadoop](#)

[Shell Interface to HDFS](#)

[Web Interface to HDFS](#)

[Java Interface to HDFS](#)

[Internals of HDFS](#)

Web Interface of HDFS

Overview 'localhost:8020' (active)

Started:	Thu Jul 27 10:53:21 AEST 2017
Version:	2.7.3, rbaa91f7c6bc9cb92be5982de4719c1c8af91ccff
Compiled:	2016-08-18T01:41Z by root from branch-2.7.3
Cluster ID:	CID-50dd23ff-1ec0-4860-b1fd-f8b0067f5b57
Block Pool ID:	BP-680435313-10.0.2.15-1499005796923

Summary

Security is off.

Safemode is off.

126 files and directories, 62 blocks = 188 total filesystem object(s).

Heap Memory used 39.84 MB of 263 MB Heap Memory. Max Heap Memory is 889 MB.

Non Heap Memory used 46.08 MB of 46.84 MB Committed Non Heap Memory. Max Non Heap Memory

Web Interface of HDFS

Browse Directory

/								Go!
Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name	
drwxr-xr-x	bigdata	supergroup	0 B	12/07/2017, 1:42:53 pm	0	0 B	hbase_data	
drwxrwx---	bigdata	supergroup	0 B	10/07/2017, 2:23:00 pm	0	0 B	tmp	
drwxr-xr-x	bigdata	supergroup	0 B	03/07/2017, 1:29:24 am	0	0 B	user	

Hadoop, 2016.

HDFS Interfaces

Outline

[Hadoop Cluster vs. Pseudo-Distributed Hadoop](#)

[Shell Interface to HDFS](#)

[Web Interface to HDFS](#)

[Java Interface to HDFS](#)

[Internals of HDFS](#)

Java Interface to HDFS

A file in a **Hadoop** filesystem is represented by a **Hadoop Path** object

- Its syntax is URI
- For example,

`hdfs://localhost:8020/user/bigdata/input/README.txt`

To get an instance of **FileSystem**, use the following factory methods

Factory methods

```
public static FileSystem get(Configuration conf) throws IOException  
public static FileSystem get(URI uri, Configuration conf) throws IOException  
public static FileSystem get(URI uri, Configuration conf, String user)  
                           throws IOException
```

The following method gets a local filesystem instance

Get local file system method

```
public static FileSystem getLocal(Configuration conf) throws IOException
```

Java interface to HDFS

A **Configuration** object is determined by the Hadoop configuration files or user-provided parameters

Using the default configuration, one can simply set

```
Configuration conf = new Configuration()
```

Configuration object

With a **FileSystem** instance in hand, we invoke an **open()** method to get the input stream for a file

```
public FSDataInputStream open(Path f) throws IOException  
public abstract FSDataInputStream open(Path f, int bufferSize) throws IOException
```

Open method

A **Path** object can be created by using a designated URI

```
Path f = new Path(uri)
```

Path object

Java interface to HDFS

Putting together, we can create the following file reading application

```
public class FileSystemCat {  
    public static void main(String[] args) throws Exception {  
        String uri = args[0];  
        Configuration conf = new Configuration();  
        FileSystem fs = FileSystem.get(URI.create(uri), conf);  
        FSDataInputStream in = null;  
        Path path = new Path(uri);  
        in = fs.open(path);  
        IOUtils.copyBytes(in, System.out, 4096, true);  
    }  
}
```

Class FileSystemCat

Java interface to HDFS

The compilation simply uses the `javac` command, but it needs to point the dependencies in the class path.

Compilation

```
export HADOOP_CLASSPATH=$( $HADOOP_HOME/bin/hadoop classpath)
javac -cp $HADOOP_CLASSPATH FileSystemCat.java
```

Then, a `jar` file is created and run as follows

jar file and processing

```
jar cvf FileSystemCat.jar FileSystemCat*.class
hadoop jar FileSystemCat.jar FileSystemcat input/README.txt
```

The output is the same as processing a command `hadoop fs -cat`

Java interface to HDFS

Suppose an input stream is created to read a local file

To write a file on **HDFS**, the simplest way is to take a **Path** object for the file to be created and return an output stream to write to

```
public FSDataOutputStream create(Path f) throws IOException
```

Path object

And then just copy the input stream to the output stream

Another, more flexible, way is to read the input stream into a buffer and then write to the output stream

Java interface to HDFS

A file writing application

File writing

```
public class FileSystemPut {  
    public static void main(String[] args) throws Exception {  
        String localStr = args[0];  
        String hdfsStr = args[1];  
        Configuration conf = new Configuration();  
        FileSystem local = FileSystem.getLocal(conf);  
        FileSystem hdfs = FileSystem.get(URI.create(hdfsStr), conf);  
        Path localFile = new Path(localStr);  
        Path hdfsFile = new Path(hdfsStr);  
        FSDataInputStream in = local.open(localFile);  
        FSDataOutputStream out = hdfs.create(hdfsFile);  
        IOUtils.copyBytes(in, out, 4096, true);  
    }  
}
```

Java interface to HDFS

Another file writing application

```
public class FileSystemPutAlt {  
    public static void main(String[] args) throws Exception {  
        String localStr = args[0];  
        String hdfsStr = args[1];  
        Configuration conf = new Configuration();  
        FileSystem local = FileSystem.getLocal(conf);  
        FileSystem hdfs = FileSystem.get(URI.create(hdfsStr), conf);  
        Path localFile = new Path(localStr);  
        Path hdfsFile = new Path(hdfsStr);  
        FSDataInputStream in = local.open(localFile);  
        FSDataOutputStream out = hdfs.create(hdfsFile);  
        byte[] buffer = new byte[256];  
        int bytesRead = 0;  
        while( (bytesRead = in.read(buffer)) > 0) {  
            out.write(buffer, 0, bytesRead);  
        }  
        in.close();  
        out.close();  
    }  
}
```

File writing

[TOP](#)

Java interface to HDFS

Other file system API methods

The method `mkdirs()` creates a directory

The method `getFileStatus()` gets the meta information for a single file or directory

The method `listStatus()` lists contents of files in a directory

The method `exists()` checks whether a file exists

The method `delete()` removes a file

The Java API enables the implementation of customised applications to interact with [HDFS](#)

HDFS Interfaces

Outline

[Hadoop Cluster vs. Pseudo-Distributed Hadoop](#)

[Shell Interface to HDFS](#)

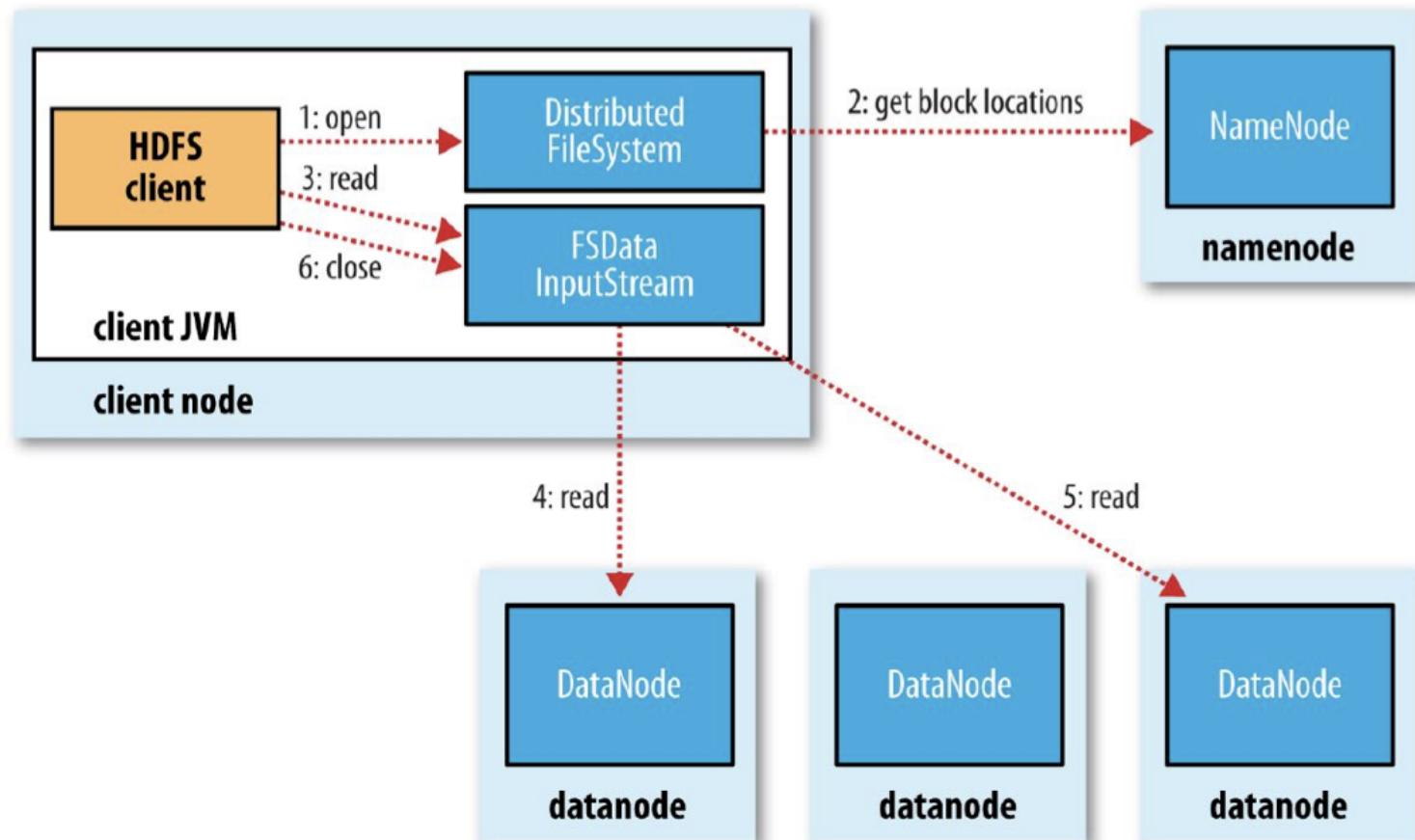
[Web Interface to HDFS](#)

[Java Interface to HDFS](#)

[Internals of HDFS](#)

Internals of HDFS

What happens "inside" when we read data into **HDFS** ?



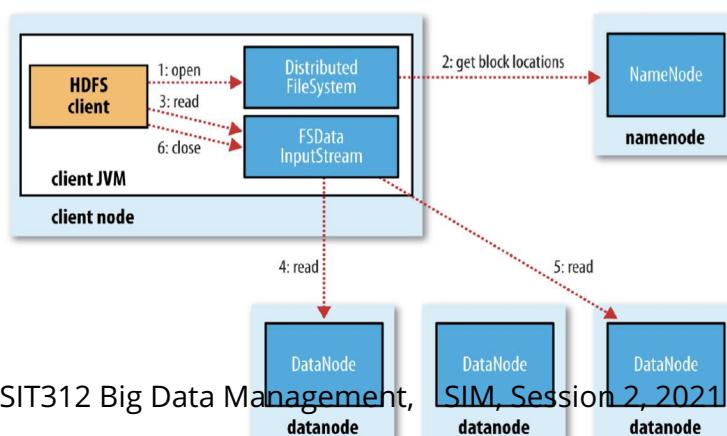
Internals of HDFS

Read data from [HDFS](#)

Step 1: The client opens the file it wishes to read by calling `open()` on the `FileSystem` object, which for `HDFS` is an instance of `DistributedFileSystem`

Step 2: `DistributedFileSystem` calls the `namenode`, using remote procedure calls (RPCs), to determine the locations of the first few blocks in the file

Step 3: The `DistributedFileSystem` returns an `FSDataInputStream` to the client and the client calls `read()` on the stream

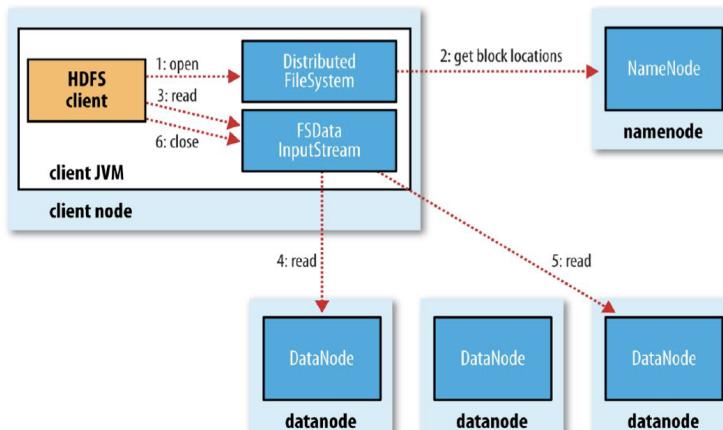


Internals of HDFS

Step 4: `FSDataInputStream` connects to the first datanode for the first block in the file, and then data is streamed from the datanode back to the client, by calling `read()` repeatedly on the stream

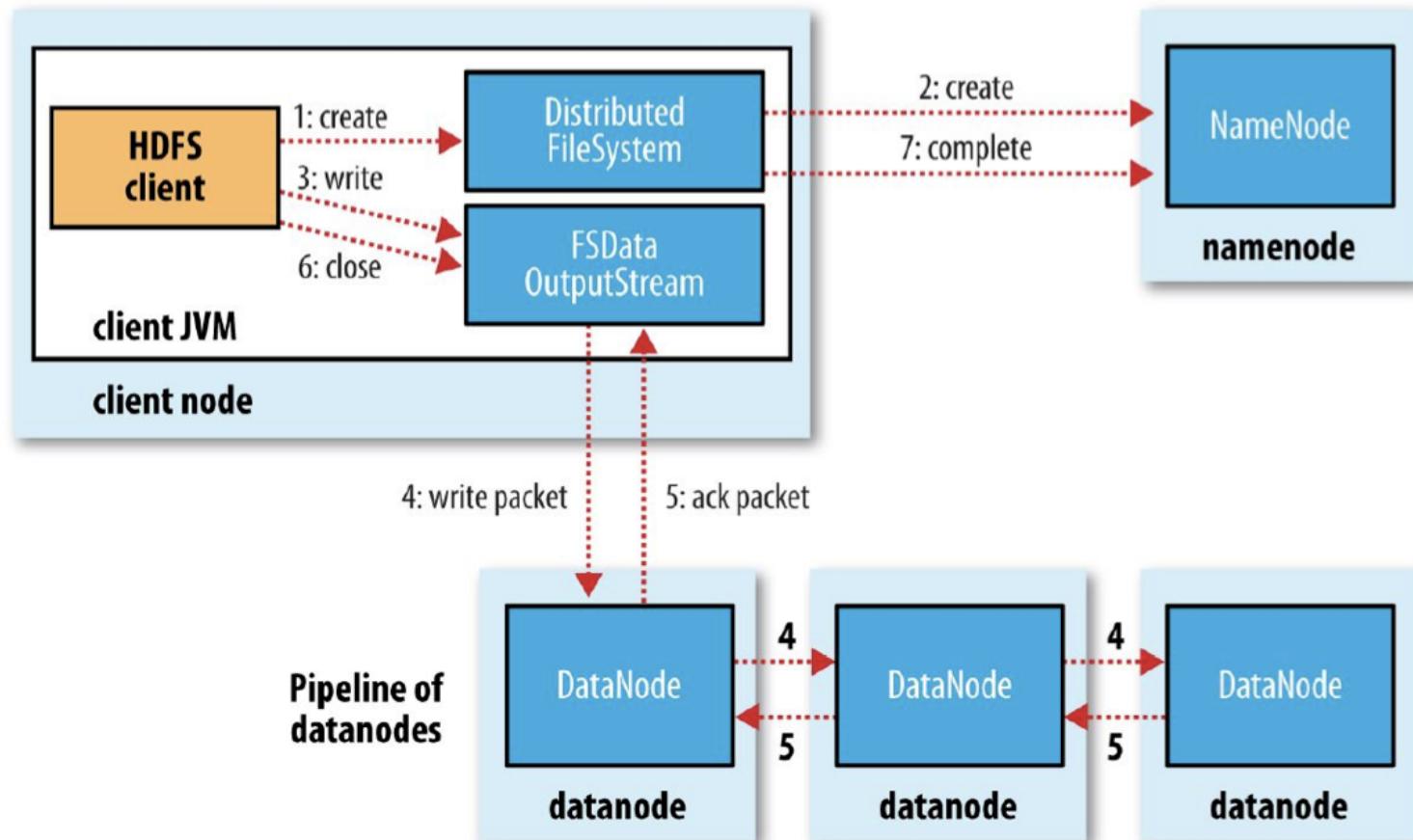
Step 5: When the end of the block is reached, `FSDataInputStream` will close the connection to the datanode, then find the best (possibly the same) datanode for the next block

Step 6: When the client has finished reading, it calls `close()` on the `FSDataInputStream`



Internals of HDFS

Write data into HDFS



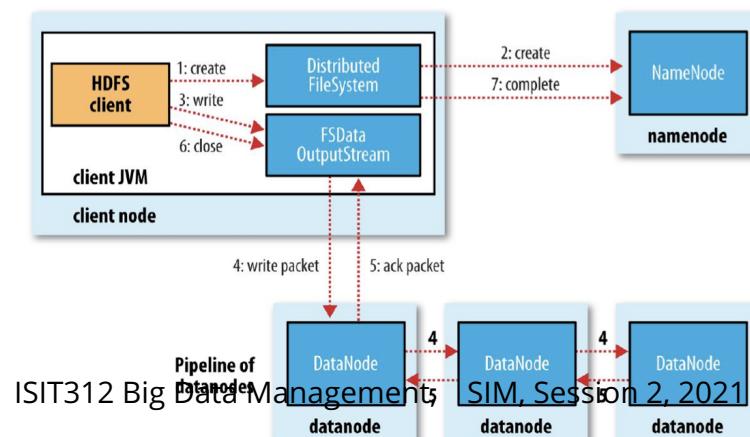
Internals of HDFS

Step 1: The client creates the file by calling `create()` on `DistributedFileSystem`

Step 2: `DistributedFileSystem` makes an RPC call to the `namenode` to create a new file in the file system namespace and returns an `FSDataOutputStream` for the client to start writing data to

Step 3: The client writes data into the `FSDataOutputStream`

Step 4: Data wrapped by the `FSDataOutputStream` is split into packages, which are flushed into a queue; data packages are sent to the blocks in a datanode and forwarded to other (usually two) datanodes

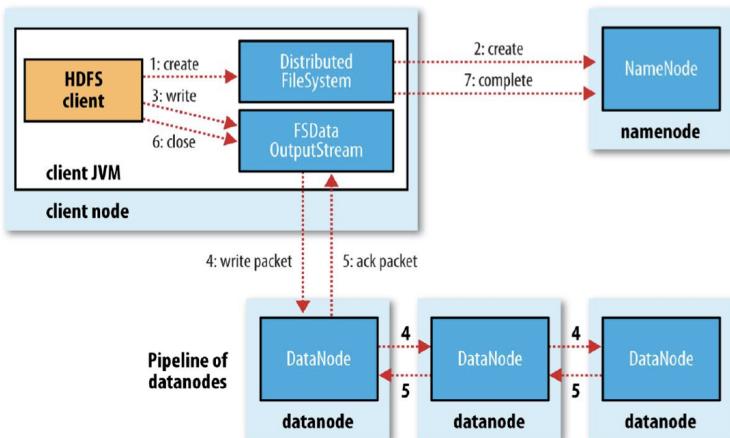


Internals of HDFS

Step 5: If `FSDataOutputStream` receives an ack signal from the datanode the data packages are removed from the queue

Step 6: When the client has finished writing data, it calls `close()` on the stream

Step 7: The client signals the namenode that the writing is completed



ISIT312 Big Data Management MapReduce Data Processing Model

Dr Guoxin Su and Dr Janusz R. Getta

School of Computing and Information Technology -
University of Wollongong

MapReduce Data Processing Model

Outline

Key-value pairs

MapReduce model

Map phase

Reduce phase

Shuffle and sort

Combine phase

Example

Key-value pairs

Key-Value pairs: MapReduce basic data model

Key	Value	sql
City	Sydney	
Employer	Cloudera/pre>	

Input, output, and intermediate records in MapReduce are represented as **key-value pairs** (aka **name-value/attribute-value pairs**)

A **key** is an identifier, for example, a name of attribute

- In MapReduce, a **key** is not required to be unique.

A **value** is a data associated with a **key**

- It may be **simple value** or a **complex object**

MapReduce Data Processing Model

Outline

[Key-value pairs](#)

[MapReduce model](#)

[Map phase](#)

[Reduce phase](#)

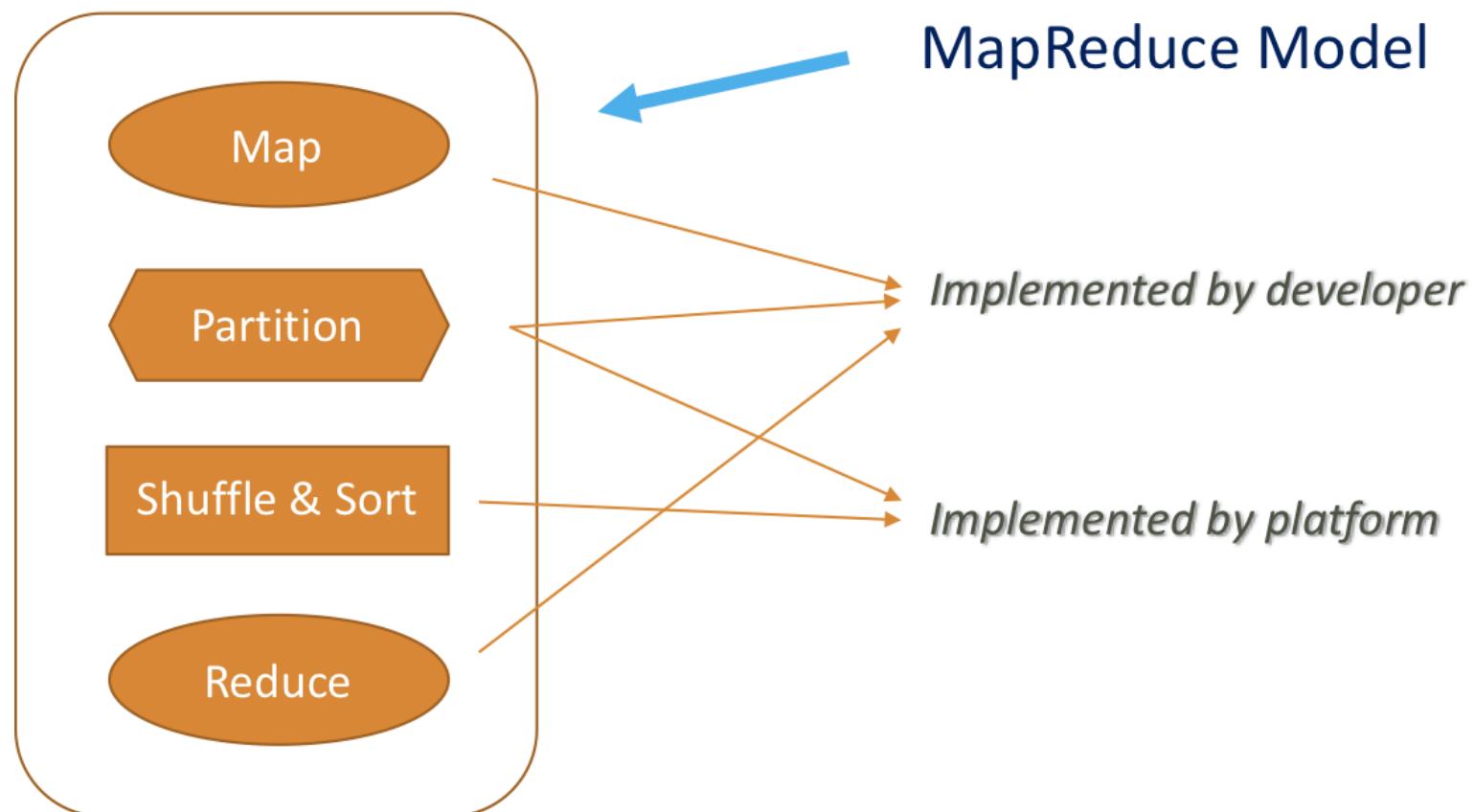
[Shuffle and sort](#)

[Combine phase](#)

[Example](#)

MapReduce Model

MapReduce data processing model is a sequence of Map, Partition, Shuffle and Sort, and Reduce stages



MapReduce Model

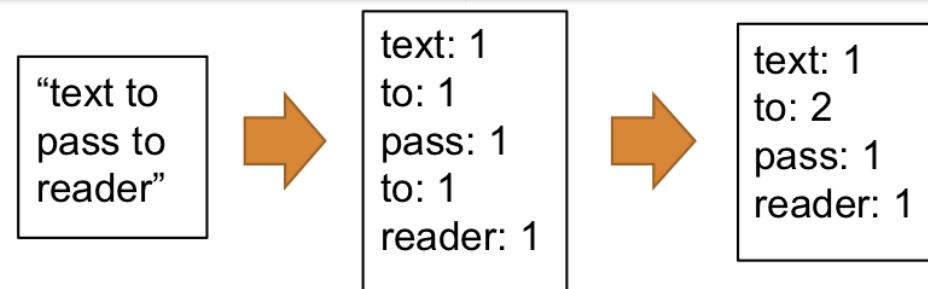
An abstract MapReduce program: WordCount

```
function Map(Long lineNumber, String line):  
    lineNumber: the position no. of a line in the text  
    line: a line of text  
        for each word w in line:  
            emit (w, 1)
```

Function Map

```
function Reduce(String w, List loc):  
    w: a word  
    loc: a list of counts outputted from map instances  
        sum = 0  
        for each c in loc:  
            sum += c  
        emit (word, sum)
```

Function Reduce



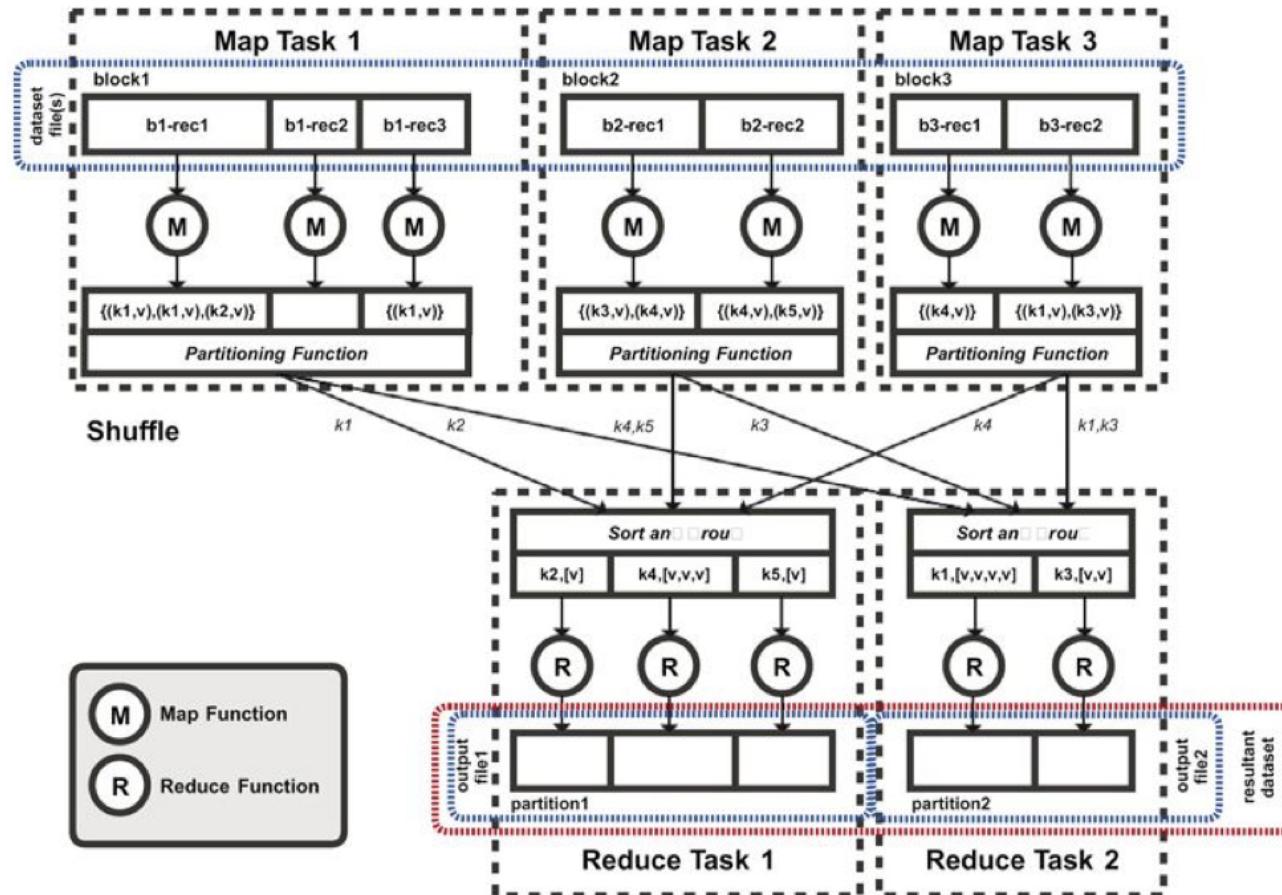
[TOP](#)

ISIT312 Big Data Management, SIM, Session 2, 2021

6/27

MapReduce Model

A diagram of data processing in MapReduce model



MapReduce Data Processing Model

Outline

[Key-value pairs](#)

[MapReduce model](#)

[Map phase](#)

[Reduce phase](#)

[Shuffle and sort](#)

[Combine phase](#)

[Example](#)

Map phase

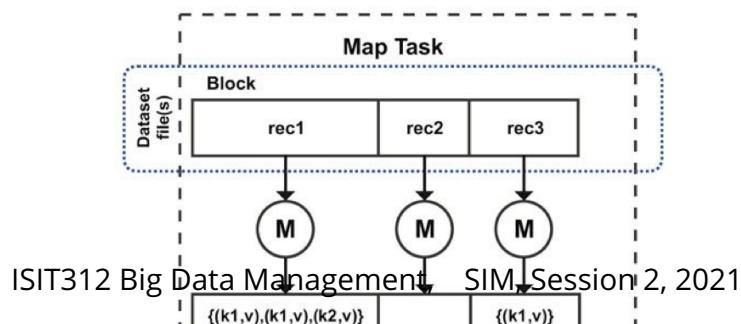
Map phase uses input format and record reader functions to derive records in the form of **key-value pairs** for the input data

Map phase applies a function or functions to each **key-value** pair over a portion of the dataset

- In the case of a dataset hosted in **HDFS**, this portion is usually called as a block
- If there are n blocks of data in the input dataset, there will be at least n **Map** tasks (also referred to as **Mappers**)

Each Map task operates against one filesystem (**HDFS**) block

In the diagram fragment, a **Map** task will call its **map()** function, represented by M in the diagram, once for each record, or **key-value pair**; for example, rec1, rec2, and so on.



Map phase

Each call of the `map()` function accepts one **key-value pair** and emits zero or **more key-value pairs**

A call of `Map()` function

```
map (in_key, in_value) -> list (intermediate_key, intermediate_value)
```

The emitted data from **Mapper**, also in the form of lists of **key-value pairs**, will be subsequently processed in the **Reduce phase**

Different **Mappers** do not communicate or share data with each other

Common `Map()` functions include filtering of specific keys, such as filtering log messages if you only wanted to count or analyse ERROR log messages

Sample `Map()` function

```
Map (k, v) = if (ERROR in v) then emit (k, v)
```

Another example of `Map()` function would be to manipulate values, such as a function that converts a text value to lowercase

Sample `Map()` function

[TOP](#)

```
Map (k, v) = emit (k, v.toLowerCase())
```

ST312

Big Data Management, SIM, Session 2, 2021

10/27

Map phase

Partition function, or Partitioner, ensures each key and its list of values is passed to one and only one Reduce task or Reducer

The number of partitions is determined by the (default or user-defined) number of Reducers

Custom Partitioners are developed for various practical purposes

MapReduce Data Processing Model

Outline

[Key-value pairs](#)

[MapReduce model](#)

[Map phase](#)

[Reduce phase](#)

[Shuffle and sort](#)

[Combine phase](#)

[Example](#)

Reduce Phase

Input of the **Reduce phase** is output of the **Map phase** (via shuffle-and sort)

Each **Reduce task** (or **Reducer**) executes a `reduce()` function for each intermediate key and its list of associated intermediate values

The output from each `reduce()` function is zero or more key-values

A call of `Reduce()` function

```
reduce (intermediate_key, list (intermediate_value)) -> (out_key, out_value)
```

Note that, in the reality, an output from **Reducer** may be an input to another **Map phase** in a complex multistage computational workflow

Example of Reduce Functions

The simplest and most common `reduce()` function is the **Sum Reducer**, which simply sums a list of values for each key

```
reduce (k, list ) =  
{  
    sum = 0  
    for int i in list :  
        sum += i  
    emit (k, sum)  
}
```

Sum reducer

A count operation is as simple as summing a set of numbers representing instances of the values you wish to count

Other examples of `reduce()` function are `max()` and `average()`

MapReduce Data Processing Model

Outline

[Key-value pairs](#)

[MapReduce model](#)

[Map phase](#)

[Reduce phase](#)

[Shuffle and sort](#)

[Combine phase](#)

[Example](#)

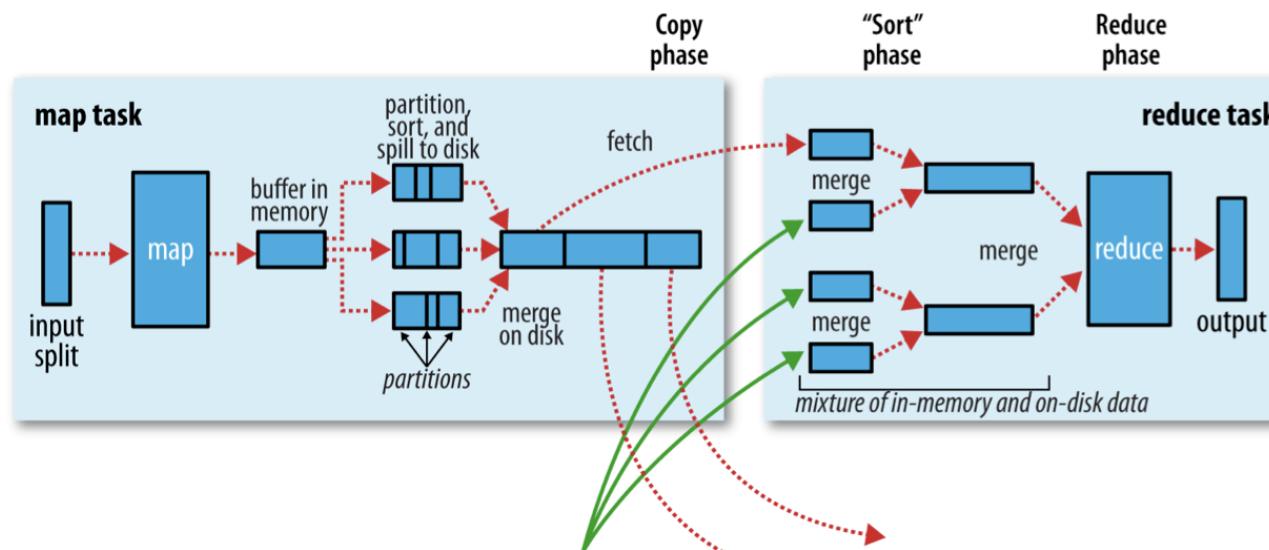
Shuffle and Sort

Shuffle-and-sort is the process where data are transferred from **Mapper** to **Reducer**

- It is the heart of **MapReduce** where the "magic" happens

The most important purpose of **Shuffle-and-sort** is to minimise data transmission through a network

In general, in **Shuffle-and-Sort**, the **Mapper** output is sent to the target **Reduce task** according to the partitioning function



MapReduce Data Processing Model

Outline

[Key-value pairs](#)

[MapReduce model](#)

[Map phase](#)

[Reduce phase](#)

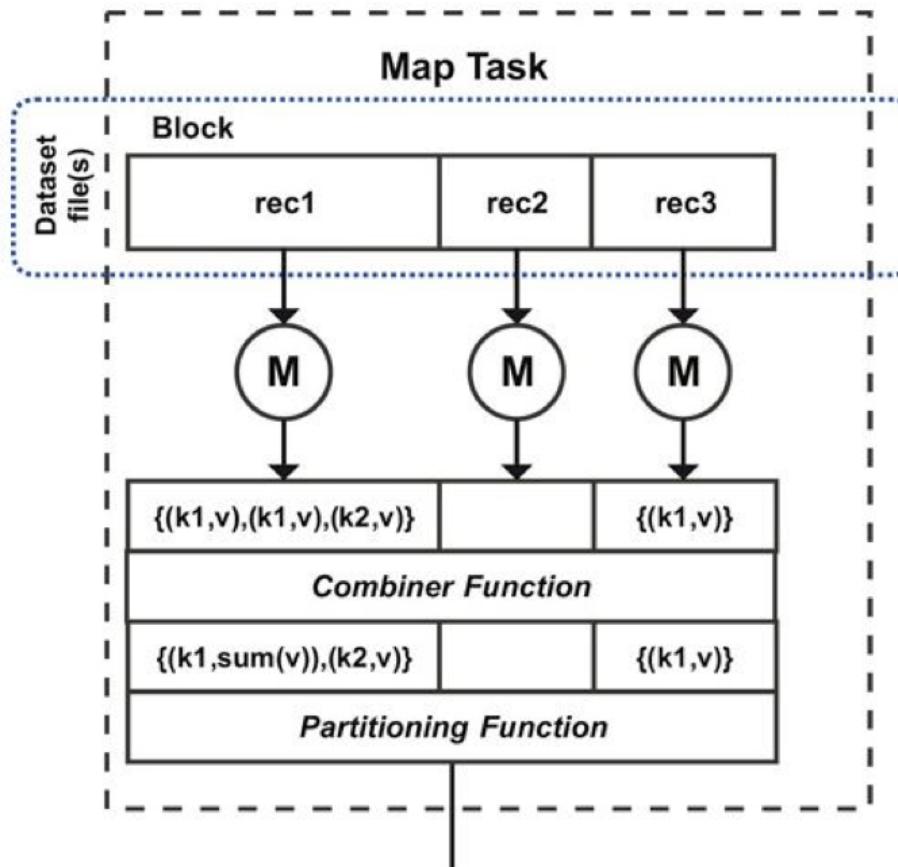
[Shuffle and sort](#)

[Combine phase](#)

[Example](#)

Combine phase

A structure of **Combine phase**



Combine phase

If the **Reduce function** is **commutative** and **associative** then it can be performed before the **Shuffle-and-Sort phase**

In this case, the **Reduce function** is called a **Combiner function**

For example, **sum** and **count** is **commutative and associative**, but **average** is not

The use of a **Combiner** can minimise the amount of data transferred to **Reduce phase** and in such a way reduce the network transmit overhead

A **MapReduce** application may contain zero **Reduce tasks**

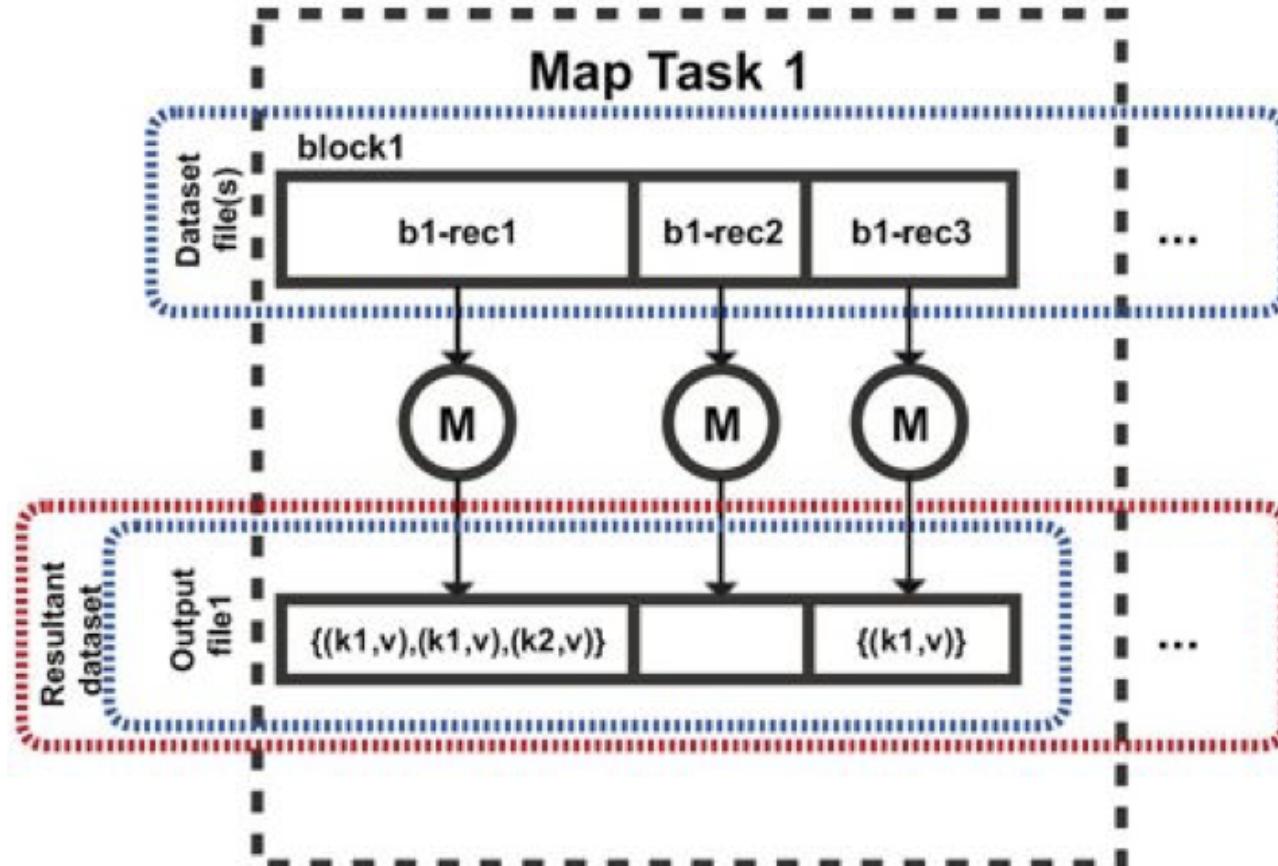
In this case, it is a **Map-Only** application

Examples of **Map-only MapReduce** jobs

- ETL routines without data summarization, aggregation and reduction
- File format conversion jobs
- Image processing jobs

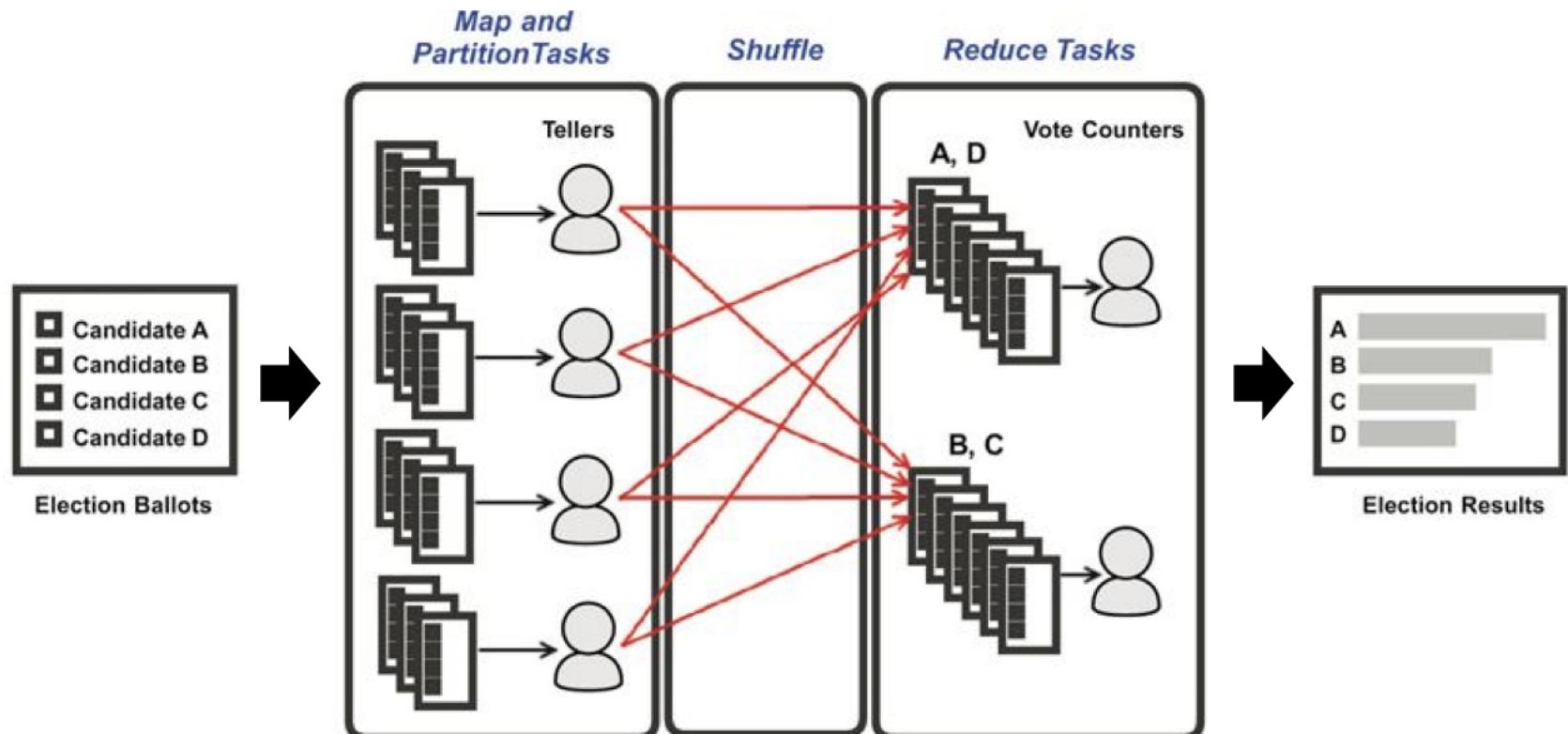
Combine phase

Map-Only MapReduce



Combine phase

An election Analogy for MapReduce



MapReduce Data Processing Model

Outline

[Key-value pairs](#)

[MapReduce model](#)

[Map phase](#)

[Reduce phase](#)

[Shuffle and sort](#)

[Combine phase](#)

[Example](#)

Example

For a database of 1 billion people, compute the average number of social contacts a person has according to age

In SQL like language

```
SELECT age, AVG(contacts)
FROM social.person
GROUP BY age
```

SELECT statement

If the records are stored in different datanodes then in **Map** function is the following

```
function Map is
    input: integer K between 1 and 1000, representing a batch of 1
    million social.person records
    for each social.person record in the K-th batch do
        let Y be the person age
        let N be the number of contacts the person has
        produce one output record (Y, (N,1))
    repeat
end function
```

Map function

[TOP](#)

Example

Then **Reduce** function is the following

```
function Reduce is
    input: age (in years) Y
    for each input record (Y,(N,C)) do
        Accumulate in S the sum of N*C
        Accumulate in C_new the sum of C
    repeat
        let A be S/C_new
        produce one output record (Y,(A,C_new ))
    end function
```

Reduce function

MapReduce sends the codes to the location of each data batch (not the other way around)

Question: the output from **Map** is multiple copies of $(Y, (N, 1))$, but the input to **Reduce** is $(Y, (N, C))$, so what fills the gap?

Example

A **MapReduce** application in **Hadoop** is a **Java** implementation of the **MapReduce model** for a specific problem, for example, word count



Example

Sample processing on a screen

```
[root@hadoop2 sbin]# hadoop jar /opt/yarn/hadoop-2.6.0/share/hadoop/mapreduce/
  hadoop-mapreduce-examples-2.6.0.jar wordcount /shakes shake_output
16/05/22 10:44:00 INFO input.FileInputFormat: Total input paths to process : 1
16/05/22 10:44:01 INFO mapreduce.JobSubmitter: number of splits:1
16/05/22 10:44:01 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1440603749764_0001
16/05/22 10:44:03 INFO impl.YarnClientImpl: Submitted application application_1440603749764_0001
16/05/22 10:44:03 INFO mapreduce.Job: The url to track the job: http://hadoop1.localdomain:8081/
  proxy/application_1440603749764_0001/
16/05/22 10:44:44 INFO mapreduce.Job: Job job_1440603749764_0001 running in uber mode : false
16/05/22 10:44:44 INFO mapreduce.Job: map 0% reduce 0%
16/05/22 10:45:07 INFO mapreduce.Job: map 67% reduce 0%
16/05/22 10:45:17 INFO mapreduce.Job: map 100% reduce 0%
16/05/22 10:45:33 INFO mapreduce.Job: map 100% reduce 100%
16/05/22 10:45:34 INFO mapreduce.Job: Job job_1440603749764_0001 completed successfully
16/05/22 10:45:36 INFO mapreduce.Job: Counters: 49
  File System Counters
    FILE: Number of bytes read=983187
    FILE: Number of bytes written=2178871
    HDFS: Number of bytes read=5590008
    HDFS: Number of bytes written=720972
    HDFS: Number of read operations=6

    HDFS: Number of write operations=2
```

The application

Example

Sample processing on a screen

```
Job Counters
    Launched map tasks=1
    Launched reduce tasks=1
    Data-local map tasks=1
    Total time spent by all maps in occupied slots (ms)=30479
    Total time spent by all reduces in occupied slots (ms)=13064
    Total time spent by all map tasks (ms)=30479
    Total time spent by all reduce tasks (ms)=13064
    Total vcore-seconds taken by all map tasks=30479
    Total vcore-seconds taken by all reduce tasks=13064
    Total megabyte-seconds taken by all map tasks=31210496
    Total megabyte-seconds taken by all reduce tasks=13377536

Map-Reduce Framework
    Map input records=124787
    Map output records=904061
    Map output bytes=8574733
    Map output materialized bytes=983187
    Input split bytes=119
    Combine input records=904061
    Combine output records=67779
    Reduce input groups=67779
    Reduce shuffle bytes=983187
    Reduce input records=67779
    Reduce output records=67779
    Spilled Records=135558
    Shuffled Maps =1
        Merged Map outputs=1
    GC time elapsed (ms)=454
    CPU time spent (ms)=10520
    Physical memory (bytes) snapshot=302411776
    Virtual memory (bytes) snapshot=1870229504
    Total committed heap usage (bytes)=168497152

File Input Format Counters
    Bytes Read=5589889
File Output Format Counters
    Bytes Written=710972
```

ISIT312 Big Data Management

Java MapReduce Application

Dr Guoxin Su and Dr Janusz R. Getta

School of Computing and Information Technology -
University of Wollongong

Common building blocks of MapReduce program

Mapper, Reducer, Combiner, and Partitioner classes correspond to their counterparts in the MapReduce model

- These classes implement the MapReduce logic

The Driver or ToolRunner in a MapReduce program represents the client program

- The main method of a MapReduce program is in the Driver or ToolRunner
- The code of the two is very standard

An elementary MapReduce program consists of a Mapper class, a Reducer class and a Driver

As the main method is contained in the Driver, sometimes (but not always) it is convenient to make Mapper and Reducer as inner classes in Driver, which contains routine codes

Driver

Driver is the program which sets up and starts a **MapReduce** application

Driver code is executed on the client; this code submits the application to the **ResourceManager** along with the application's configuration

Driver can submit the job asynchronously (in a non-blocking fashion) or synchronously (waiting for the application to complete before performing another action)

Driver can also configure and submit more than one application; for instance, running a workflow consisting of multiple **MapReduce** applications

Mapper

Mapper Java class contains a `map()` method

Its object instance iterates through the input split to execute a `map()` method, using the **InputFormat** and its associated **RecordReader**

The number of **HDFS** blocks for the file determines the number of input splits, which, in turn, determines the number of **Mapper** objects (or **Map** tasks) in a **MapReduce** application

Mappers can also include setup and cleanup code to run in any given object lifespan

Mappers do most of the heavy lifting in data processing in **MapReduce**, as they read the entire input file for the application

Reducer

Reducer runs against a partition and each key and its associated values are passed to a `reduce()` method inside **Reducer class**

Reduce's InputFormat matches **Mapper's OutputFormat**

While **Mapper** usually do the data preparation, for example, filtering and extracting), **Reducer** usually contains the main application logic

- For example, summation, counting, and averaging operations are implemented in **Reducers**

The runtime of **Reducer** instances is usually faster (and much faster in some cases) than the runtime of **Mapper** instances

Word Count: The "Hello, World" of MapReduce

WordCount: Read a text file and count occurrences of each word

Consider a text document containing a fragment of the works of Shakespeare

Romeo and Juliet

```
O Romeo, Romeo! wherefore art thou Romeo?  
Deny thy father, and refuse thy name
```

The input format is **TextInputFormat**

After the text is read, the input to **Map** task, i.e. the process running **Mapper** is the following

Romeo and Juliet

```
(0, 'O Romeo , Romeo ! wherefore art thou Romeo ?')  
(45, 'Deny thy father, and refuse thy name')
```

Word Count: The "Hello, World" of MapReduce

The output of the Map task is the following

- It is possible to filter out in **Mapper** some trivial words such as "a" and "and"

```
('O', 1)
('Romeo', 1)
('Romeo', 1)
('wherefore', 1)
('art', 1)
('thou', 1)
('Romeo', 1)
('Deny', 1)
('thy', 1)
('father', 1)
('and', 1)
('refuse', 1)
('thy', 1)
('name', 1)
```

Key-Value pairs

Word Count: The "Hello, World" of MapReduce

Before sending data to **Reduce** task, there is a **shuffle-and-sort stage**

Shuffle-and-sort is usually hidden from a programmer

The following is the input to **Reduce** task

Key-value pairs after shuffle-and sort

```
('and', [1])
('art', [1])
('Deny', [1])
('father', [1])
('name', [1])
('O', [1])
('refuse', [1])
('Romeo', [1,1,1])
('thou', [1])
('thy', [1,1])
('wherefore', [1])
```

Word Count: The "Hello, World" of MapReduce

The following is the final output from **Reduce** task:

- Note that we use plain texts to illustrate the data passing through the **MapReduce stages**, but in the Java implementation, all texts are wrapped in some object that implements the **Writable interface**

Final output from Reduce

```
('and', 1)
('art', 1)
('Deny', 1)
('father', 1)
('name', 1)
('O', 1)
('refuse', 1)
('Romeo', 3)
('thou', 1)
('thy', 2)
('wherefore', 1)
```

Hadoop data type objects

In most programming languages, when defining most data elements, we usually use simple, or primitive, datatypes such as `int`, `long`, or `char`

However, in Hadoop a key or a value is an object that is an instantiation of a class, with attributes and defined methods

A key or a value contains (or encapsulates) the data with methods defined for reading and writing data from and to the object

Writable interface

Hadoop serialisation format is **Writable** interface

For example, a class that implements **Writable** is **IntWritable**, which a wrapper for a Java **int**

One can create such a class and set its value in the following way

Creating IntWritable object

```
IntWritable writable = new IntWritable();
writable.set(163);
```

Alternatively, we can write

Creating IntWritable object

```
IntWritable writable = new IntWritable(163);
```

WritableComparable interface

IntWritable implements WritableComparable interface

It is interface of Writable and java.lang.Comparable interfaces

Creating WritableComparable interface

```
package org.apache.hadoop.io;
public interface WritableComparable extends Writable, Comparable { ... }
```

Comparison is crucial for MapReduce, because MapReduce contains a sorting phase during which keys are compared with one another

WritableComparable permits to compare records read from a stream without deserialising them into objects, thereby avoiding any overhead of object creation

Hadoop primitive Writable wrappers

Writable Wrapper	Java Primitive	Writable Wrappers
BooleanWritable	boolean	
ByteWritable	byte	
IntWritable	int	
FloatWritable	float	
LongWritable	long	
DoubleWritable	double	
NullWritable	null	
Text	String	

Input and output formats

FileInputFormat (the base class of **InputFormat**) reads data (keys and values) from a given path, using the default or user-defined format

- The default input format is **LongWritable** for the keys and **Text** for the values

FileOutputFormat (the base class of **OutputFormat**) writes data into a file in a given path

- The output format is usually defined by a programmer
- For example, the output format is **Text** for the keys and **IntWritable** for the values

Some imported package members

Imported package members

```
import java.io.IOException;
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;
```

Java code of Driver

Java code of Driver

```
public class WordCount {  
    public static void main(String[] args) throws Exception {  
        Configuration conf = new Configuration();  
        Job job = Job.getInstance(conf, "word count");  
        job.setJarByClass(WordCount.class);  
        job.setMapperClass(MyMapper.class); //the Mapper class  
        job.setReducerClass(MyReducer.class); //the Reducer 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);  
    }  
}
```

Job object and configuration

Driver class instantiates a **Job** object

A **Job** object creates and stores the configuration options for a **Job**, including the classes to be used as **Mapper** and **Reducer**, input and output directories, etc

The configuration options are specified in (one or more of) the following places

- Hadoop defaults (`*-default.xml`, e.g., `core-default.xml`)
- A default configuration is documented in the Apache Hadoop documentation
- The `*-site.xml` files on the client node where **Driver** code is processed
- The `*-site.xml` files on the slave nodes where **Mapper** runs on
- **Configuration** properties set at the command line as arguments to a **MapReduce** application (in a **ToolRunner** object)
- **Configuration** properties set explicitly in code and compiled through a **Job** object

Driver routines

Parses the command line for positional arguments - `input` file(s)/directory and `output` directory

Creates a new `Job` object instance, using `getConf()` method to obtain configuration from the various sources (`*-default.xml` and `*-site.xml`)

Gives a `Job` a friendly name (the name you will see in the ResourceManager UI)

Sets the `InputFormat` and `OutputFormat` for a `Job` and determines the input splits for a `Job`

Defines `Mapper` and `Reducer` classes to be used for a Job (They must be available in the Java classpath where `Driver` is run - typically these classes are packaged alongside the Driver)

Sets the final output key and value classes, which will be written out files in the output directory

[TOP](#) Submits a `Job` object through `job.waitForCompletion(true)`

Java code of Mapper

Java code of Mapper

```
public static class MyMapper  
extends Mapper{  
    private final static IntWritable one = new IntWritable(1);  
    private Text wordObject = new Text();  
    public void map(Object key, Text value, Context context  
        ) throws IOException, InterruptedException {  
        String line = value.toString();  
        for (String word : line.split("\\w+")) {  
            if (word.length() > 0) {  
                wordObject.set(word);  
                context.write(wordObject, one);  
            }  
        }  
    }  
}
```

Mapper class

A class **MyMapper** extends a base **Mapper** class included within the [Hadoop libraries](#)

In the example, the four generics in
`Mapper<Object, Text, Text, IntWritable>` represent
`<map_input_key, map_input_value, map_output_key,
map_output_value>`

These generics must correspond to

- Key-value pair types as defined by **InputFormat** in **Driver** (may be the default one)
- `job.setMapOutputKeyClass` and `job.setMapOutputValueClass` defined in **Driver**
- Input and output to the **map()** method

Mapper class

In `map()` method, before performing any functions against a key or a value (such as `split()`), we need to get the value contained in the serialised `Writable` or `WritableComparable` object, by using the `value.toString()` method

After performing operations against the input data (**key-value pairs**), the output data (intermediate data, also **key-value pairs**) are `WritableComparable` and `Writable` objects, both of which are emitted using a `Context` object

In the case of a **Map-only** job, the output from **Map phase**, namely the set of **key-value pairs** emitted from all `map()` methods in all map tasks, is the final output, without intermediate data or **Shuffle-and-Sort phase**

Context object

A **Context** object is used to pass information between processes in Hadoop

We mostly invoke its `write()` method to write the output data from **Mapper** and **Reducer**

Other functions of **Context** object are the following

- It contains configuration and state needed for processes within the **MapReduce** application, including enabling parameters to be passed to distributed processes
- It is used in the optional `setup()` and `cleanup()` methods within a **Mapper** or **Reducer**

Java code of Reducer

```
public static class MyReducer
    extends Reducer {
    private IntWritable result = new IntWritable();
    public void reduce(Text key, Iterable values,
        Context context)
        throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
    }
}
```

sql

Reducer class

A class **MyReducer** extends the based **Reducer** class included with the **Hadoop** libraries.

The four generics in

Reducer<Text, IntWritable, Text, IntWritable< represents
<reduce_input_key, reduce_input_value, reduce_output_key,
reduce_output_value>

A **reduce()** method accepts a key and an **Iterable** list of values as input, denoted by the angle brackets **<>**, for example

Iterable<IntWritable>

As in **Mapper**, to operate or perform Java string or numeric operations against keys or values from the input list of values, we first extract a value included in **Hadoop** object

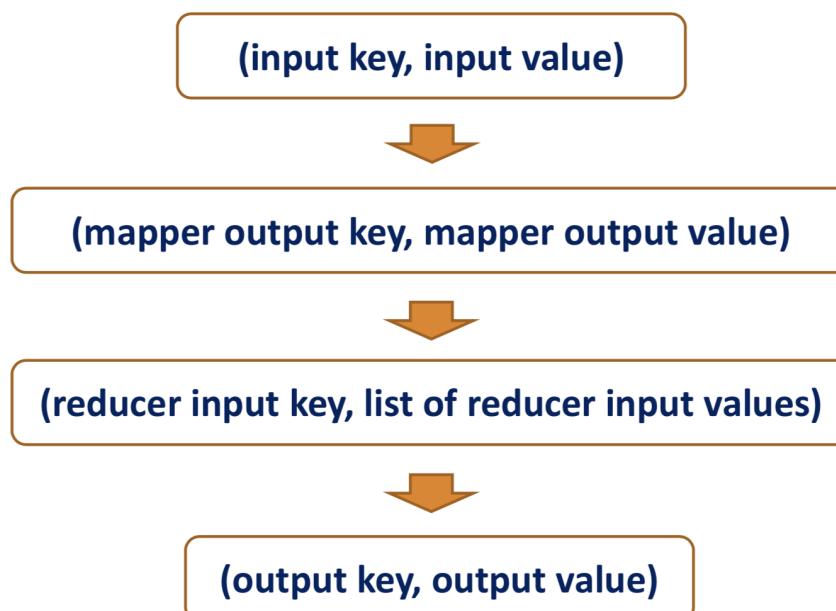
Also, the emit of **key-value pairs** in the form of **WritableComparable** objects for keys and values uses a **Context** object

Reducer class

Reducer class implements the main application logic

For example, the actual counting of WordCount is implemented in Reducer

Data flow of keys and values



ToolRunner

Despite optional, **Driver** can leverage a class called **ToolRunner**, which is used to parse command-line options

A class ToolRunner

```
// ... Originally imported package members of WordCount
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;
import org.apache.hadoop.conf.Configured;
public class WordCountTR extends Configured implements Tool {
    public static void main(String[] args)
        throws Exception {
        int res = ToolRunner.run(new Configuration(), new
            WordCount(), args);
        System.exit(res);
    }
// "run" method of ToolRunner (in the next slide)
}
```

ToolRunner

run method of ToolRunner

```
run method of ToolRunner  
@Override  
public int run(String[] args) throws Exception {  
    Configuration conf = this.getConf();  
    Job job = Job.getInstance(conf, "word count with ToolRunner");  
    job.setJarByClass(WordCountTR.class);  
    job.setMapperClass(MyMapper.class); //the Mapper class  
    job.setReducerClass(MyReducer.class); //the Reducer class  
    job.setOutputKeyClass(Text.class);  
    job.setOutputValueClass(IntWritable.class);  
    FileInputFormat.addInputPath(job, new Path(args[0]));  
    FileOutputFormat.setOutputPath(job, new Path(args[1]));  
    return job.waitForCompletion(true) ? 0 : 1;  
}
```

ToolRunner in the command line

ToolRunner enables flexibility in supplying configuration parameters at the command line when submitting a **MapReduce** job

The following submission has a command to specify the number of Reduce tasks from the command line

ToolRunner in a command line

```
hadoop jar mr.jar MyDriver -D mapreduce.job.reduces=10 myinputdir myoutputdir
```

The following submission specifies the location of **HDFS**

ToolRunner in a command line

```
hadoop jar mr.jar MyDriver  
-D fs.defaultFS=hdfs://localhost:9000 myinputdir myoutputdir
```

Setting up a local running environment

A local running environment is often convenient in MapReduce application development

- Before sending it out to the whole cluster.
- Used for debugging and testing

Create a configuration file, say, `hadoop-local.xml`

```
<property>
  <name>fs.defaultFS</name>
  <value>file:///</value>
</property>
<property>
  <name>mapreduce.framework.name</name>
  <value>local</value>
</property>
```

`hadoop-local.xml`

Setting local environment

```
hadoop jar mr.jar MyDriver -conf hadoop-local.xml myinputdir myoutputdir
```

Combiner API

Combiner functions can decrease the amount of intermediate data sent between **Mappers** and **Reducers** as part of a **Shuffle-and-Sort process**

- In a sense, **Combiner** is the "map-side reducers"

One can reuse **Reducer** code to implement **Combiner** if

- A **combiner()** function is identical to a **reduce()** function defined in your **Reducer** class
- The output key and value object types from a **map()** function implemented in **Mapper** match the input to the function used in **Combiner**
- The output key and value object types from the function used in **Combiner** match the input key and value object types used in **Reducer's reduce()** method
- The operation to be performed is commutative and associative.

Partitioner API

Partitioner divides the output keyspace for a **MapReduce** application, controlling which **Reducers** get which intermediate data

- It is useful in process distribution or load balancing, for example, getting more **Reduce** tasks running in parallel
- It can be used to segregate the outputs, for example, creating a file for monthly data in a year

A default **Partitioner** is a **HashPartitioner**, which arbitrarily hashes the key space such that

- the same keys go to the same **Reducers** and
- the keyspace is distributed roughly equally among the number of **Reducers** when determined by a programmer

In case of one **Reduce** task (default in pseudo-distributed mode), the **Partitioner** is "academic" because all intermediate data goes to the same Reducer

Example: LetterPartitioner

LetterPartitioner

```
// ... other imported package members
import org.apache.hadoop.mapreduce.Partitioner;
public static class LetterPartitioner
    extends Partitioner {
    @Override
    public int getPartition(Text key, IntWritable value, int
                           numReduceTasks) {
        String word = key.toString();
        if (word.toLowerCase().matches("^[a-m].*\$")) {
            // if word starts with a to m, go to the first Reducer or partition
            return 0;
        } else {
            // else go to the second Reducer or partition
            return 1;
        }
    }
}
```

Declare Combiner and Partitioner in a Driver

Declaring Combiner and Partitioner in a Driver

```
public class WordCountWithLetPar {  
    public static void main(String[] args) throws Exception {  
        ...  
        job.setMapperClass(MyMapper.class);      // Mapper class  
        job.setCombinerClass(MyReducer.class); // Combiner class, which is  
                                              same as Reducer class in this program  
        job.setPartitionerClass(MyPartitioner.class); // Partitioner class  
        job.setReducerClass(MyReducer.class); // Reducer class  
        ...  
    }  
}
```

ToolRunner options

ToolRunner options	
Option	Description
<code>-D property=value</code>	Sets the given Hadoop configuration property to the given value. Overrides any default or site properties in the configuration and any properties set via the <code>-conf</code> option.
<code>-conf filename ...</code>	Adds the given files to the list of resources in the configuration. This is a convenient way to set site properties or to set a number of properties at once.
<code>-fs uri</code>	Sets the default filesystem to the given URI. Shortcut for <code>-D fs.defaultFS=uri</code> .
<code>-jt host:port</code>	Sets the YARN resource manager to the given host and port. (In Hadoop 1, it sets the jobtracker address, hence the option name.) Shortcut for <code>-D yarn.resourcemanager.address=host:port</code> .

ToolRunner options

Option	Description
<code>-files file1, file2, ...</code>	Copies the specified files from the local filesystem (or any filesystem if a scheme is specified) to the shared filesystem used by MapReduce (e.g. HDFS) and makes them available to MapReduce programs in the working directory of task.
<code>-archives archive1, archive2, ...</code>	Copies the specified archives from the local filesystem (or any filesystem if a scheme is specified) to the shared filesystem used by MapReduce (usually HDFS), unarchives them, and makes them available to MapReduce programs in the working directory of task.
<code>-libjars jar1, jar2, ...</code>	Copies the specified JAR files from the local filesystem (or any filesystem if a scheme is specified) to the shared filesystem used by MapReduce (usually HDFS) and adds them to the MapReduce classpath of task. This option is a useful way of shipping JAR files that a job is dependent on.

ISIT312 Big Data Management

Hive

Dr Janusz R. Getta

School of Computing and Information Technology -
University of Wollongong

Hive

Outline

Hive ? What is it ?

Deployment and configuration

Metastore

Interfaces

HQL

Hive versus relational DBMSs

Hive ? What is it ?

Hive is a software system that provides tabular view of data stored in HDFS and SQL-like methods for manipulating data in HDFS

Apache Hive project started at Facebook in 2010 to provide a high-level interface to HDFS

Contrary to Pig, Hive provides SQL-like abstractions on top of MapReduce

A language called HQL (Hive Query Language) implements SQL-92 standard (almost)

HQL provides a tabular view of data and it can be used to access data located in HDFS

Hive frees data analysts from Java MapReduce programming skills (not completely)

HQL statements are parsed by the Hive client and translated into a sequence of Java MapReduce operations, which are later on processed by Hadoop

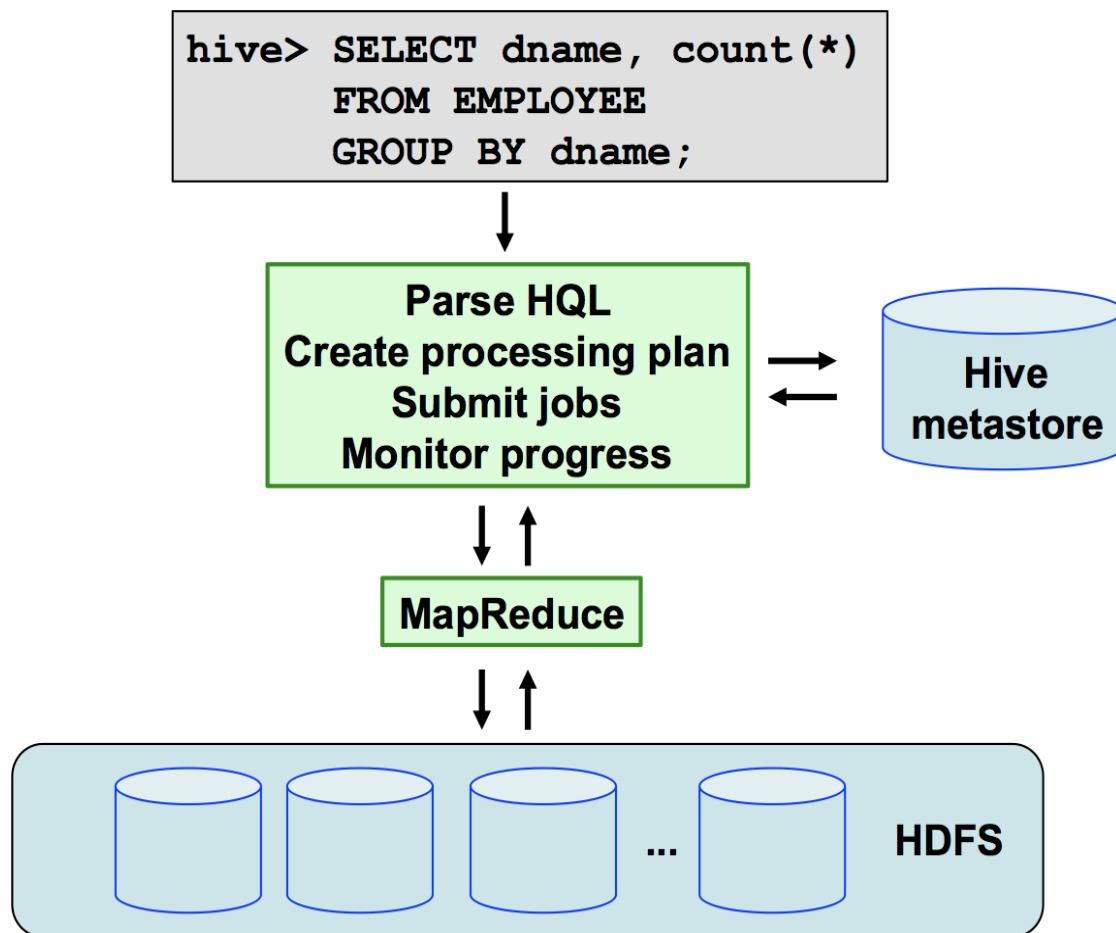
[TOP](#)

Created by Janusz R. Getta, ISIT312 Big Data Management, SIM, Session 2, 2021

3/16

Hive ? What is it ?

The results of processing by **Hadoop** are returned to the client or saved in **HDFS**



Hive

Outline

[Hive ? What is it ?](#)

[Deployment and configuration](#)

[Metastore](#)

[Interfaces](#)

[HQL](#)

[Hive versus relational DBMSs](#)

[TOP](#)

Created by Janusz R. Getta, ISIT312 Big Data Management, SIM, Session 2, 2021

5/16

Deployment and Configuration

Hive is available on all of commercial distributions of Hadoop and on Hadoop installation on our virtual machine

A relational embedded database system Derby is used for implementation of metastore

It is possible to use other relational database systems for implementation of metastore like for example MySQL

To use Hive Hadoop and HDFS must be "up and running"

A top level view of data provided by Hive consists of databases and tables

Hive

Outline

[Hive ? What is it ?](#)

[Deployment and configuration](#)

[Metastore](#)

[Interfaces](#)

[HQL](#)

[Hive versus relational DBMSs](#)

[TOP](#)

Created by Janusz R. Getta, ISIT312 Big Data Management, SIM, Session 2, 2021

7/16

Metastore

Metastore contains the mappings of tables to the directory locations in HDFS

Metastore is a relational database read and written by Hive client

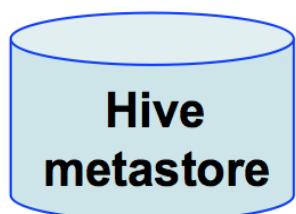
Metastore also includes the input and output formats for the files represented by the table objects, e.g. CSV InputFormat, etc, and SerDes (Serialization/ Deserialization) functions

Input and output formats for the files and functions are used by Hive to extract records and fields from the files

Metastore

```
hive> CREATE TABLE DEPARTMENT  
      ( dname string,  
        budget bigint,  
        cdate date );
```

↓ Saved in



↓ Retrieved from

```
hive> SELECT dname  
      FROM DEPARTMENT  
     WHERE budget > 100000;
```

Hive

Outline

[Hive ? What is it ?](#)

[Deployment and configuration](#)

[Metastore](#)

[Interfaces](#)

[HQL](#)

[Hive versus relational DBMSs](#)

[TOP](#)

Created by Janusz R. Getta, ISIT312 Big Data Management, SIM, Session 2, 2021

10/16

Interfaces

Hive provides Command Line Interface (CLI) that accepts and parses HQL commands

Hive provides JDBC/ODBC connector (drivers) to work with other tools such as:

- [beeline](#) (CLI),
- [Oracle SQL Developer](#) (GUI),
- [Talend Open Studio](#) (data extraction, transformation, loading, and integration tools),
- [Jasper reports](#), [QlikView](#) (business intelligence reporting tools),
- [Microsoft Excel 2013](#) (data analysis tools), and [Tableau](#) (data visualization tools)

Hive provides a storage handler mechanism to integrate with [HBase](#)

[HUE](#) (Hadoop User Experience) provides a unified web interface to [HDFS](#) and [Hive](#) in an interactive environment

[HCatalog](#) provides metadata management system for [Hadoop](#), [Pig](#), [Hive](#), and [MapReduce](#)

[TOP](#)

Created by Janusz R. Getta, ISIT312 Big Data Management, SIM, Session 2, 2021

11/16

Hive

Outline

[Hive ? What is it ?](#)

[Deployment and configuration](#)

[Metastore](#)

[Interfaces](#)

[HQL](#)

[Hive versus relational DBMSs](#)

[TOP](#)

Created by Janusz R. Getta, ISIT312 Big Data Management, SIM, Session 2, 2021

12/16

HQL

HQL consists of Data Definition Language, Data Selection and Scope Language, Data Manipulation Language, and Data Aggregation and Sampling Language

Data Definition Language is used for creating, deleting, and altering schema objects like database tables, views, partitions, and buckets

Data Selection and Scope Language is used for querying data, linking data, and limiting the data ranges or scopes

Data Manipulation Language is used for exchanging, moving, sorting, and transforming data

Data Aggregation and Sampling Language is used for exchanging, moving, sorting, and transforming data

Hive

Outline

[Hive ? What is it ?](#)

[Deployment and configuration](#)

[Metastore](#)

[Interfaces](#)

[HQL](#)

[Hive versus relational DBMSs](#)

[TOP](#)

Created by Janusz R. Getta, ISIT312 Big Data Management, SIM, Session 2, 2021

14/16

Hive versus relational DBMS

Similarities

- Tabular view of data objects in [HDFS](#)
- Directories and files viewed as tables
- Types of columns in tables
- Access to tables through [HQL](#) very similar to [SQL](#)
- API interface the same as [JDBC](#) programming interface

Differences

- Load and read-only data management system based on implementation of [HDFS](#)
- It is still possible to access data visible in tabular format in Hive directly through [HDFS](#)
- [UPDATE](#) supported as coarse-grained transformation instead of [fine-grained](#) transformation in relational DBMSs
- No transaction processing system
- No verification of consistency constraints, e.g. primary keys, foreign keys, domains constraints, etc

References

Gross C., GuptaA., Shaw S., Vermeulen A. F., Kjerrumgaard D., Practical Hive: A guide to Hadoop's Data Warehouse System, Apress 2016, Chapter 4 (Available through UOW library)

Lee D., Instant Apache Hive essentials how-to: leverage your knowledge of SQL to easily write distributed data processing applications on Hadoop using Apache Hive, Packt Publishing Ltd. 2013 (Available through UOW library)

Apache Hive TM, <https://hive.apache.org/>

ISIT312 Big Data Management

Hive Data Structures

Dr Janusz R. Getta

School of Computing and Information Technology -
University of Wollongong

Hive Data Structures

Outline

[Primitive Data Types](#)

[Complex Data Types](#)

[Databases](#)

[Tables](#)

[Partitions](#)

[Buckets](#)

[Views](#)

Primitive Data Types

TINYINT, 1 byte, example: 10Y

SMALLINT, 2 bytes, example: 10S

INT, 4 bytes, example: 10

BIGINT, 8 bytes, example: 10L

FLOAT, 4 bytes, example: 0.1234567

DOUBLE, 8 bytes, example: 0.1234567891234

DECIMAL, (m,n), example: 3.14

BINARY, n bytes, example: 1011001

BOOLEAN, 1 byte example: TRUE

STRING, 2G bytes, example: 'Abcdef'

CHAR, 255 bytes, example: 'Hello'

Primitive Data Types

VARCHAR, 1 byte, example: '**Hive**'

DATE, YYYY-MM-DD, example: '**2017-05-03**'

TIMESTAMP, YYY-MM-DD HH:MM:SS[.fff...] example: '**2017-05-03
15:10:00.345**'

Hive Data Structures

Outline

[Primitive Data Types](#)

[Complex Data Types](#)

[Databases](#)

[Tables](#)

[Partitions](#)

[Buckets](#)

[Views](#)

Complex Data Types

ARRAY: list of values of the same types,

example: ['Hadoop', 'Pig', 'Hive']
access: bigdata[1]

Array type

MAP: a set of key-value pairs,

example: {'k1': 'Hadoop', 'k2': 'Pig'}
access: bigdata['k2']

Map type

STRUCT: user defined structure of any type of fields,

example: {name: 'Hadoop', age: 24, salary: 50000.06}
access: bigdata.name

Struct type

Complex Data Types

The following **CREATE TABLE** command creates a table **types** with complex data types columns

```
CREATE TABLE types(
    array_col array<string>,
    map_col map<int,string>,
    struct_col struct<a:string, b:int, c:double> );
```

A table with columns of types

SELECT statement can be used to load data into a table with complex data types columns

```
INSERT INTO types
    SELECT array('bolt', 'nut', 'screw'),
           map(1,'bolt', 2,'nut', 3,'screw'),
           named_struct('a','bolt', 'b',5, 'c',0.5)
    FROM DUAL;
```

Inserting values into a table

Hive Data Structures

Outline

[Primitive Data Types](#)

[Complex Data Types](#)

[Databases](#)

[Tables](#)

[Partitions](#)

[Buckets](#)

[Views](#)

Databases

Database is a collection of conceptually related tables, i.e. tables that implement a conceptual schema

Database is implemented as a folder/directory in **HDFS**

A **default database** is located at `/user/hive/warehouse`

A new database is created in a folder `/user/hive/warehouse`

For example, a database `tpchr` is located at `/user/hive/warehouse/tpchr.db`

Databases

The following **CREATE DATABASE** command creates a database **tpchr**

Creating a database

```
CREATE DATABASE tpchr;
```

To find more information about a database we can use **DESCRIBE DATABASE** command

Listing a database

```
DESCRIBE DATABASE tpchr;
```

A command **USE** makes a database "current" (there is no need to prefix a table name with a database name)

Making a database current

```
USE tpchr;
```

To delete a database we can use **DROP DATABASE** command

Dropping a database

```
DROP DATABASE tpchr;
```

Hive Data Structures

Outline

[Primitive Data Types](#)

[Complex Data Types](#)

[Databases](#)

[Tables](#)

[Partitions](#)

[Buckets](#)

[Views](#)

Tables

An **internal table** (or **managed table**) is a table created by **Hive** in **HDFS**

If data is already stored in **HDFS** then an **external Hive table** can be created to provide a tabular view of the data

Location in **HDFS** of data stored in an external table is specified in the **LOCATION** properties instead of the default warehouse directory

Hive fully manages the life cycle (add/delete data, create/drop table) of **internal tables** and data in the **internal tables**

When an **external table** is deleted its metadata information is deleted from a **metastore** and the data is kept in **HDFS**

Tables

CREATE TABLE statement creates an internal table

Creating an internal table

```
CREATE TABLE IF NOT EXISTS intregion(  
    R_REGIONKEY DECIMAL(12),  
    R_NAME VARCHAR(25),  
    R_COMMENT VARCHAR(152) )  
    ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'  
    STORED AS TEXTFILE;
```

LOAD DATA statement loads data into an internal table

Loading data into an internal table

```
LOAD DATA LOCAL INPATH 'region.tbl' INTO TABLE intregion;
```

Tables

CREATE EXTERNAL TABLE statement creates an external table

Creating an external table

```
CREATE EXTERNAL TABLE IF NOT EXISTS extregion(  
    R_REGIONKEY DECIMAL(12),  
    R_NAME VARCHAR(25),  
    R_COMMENT VARCHAR(152) )  
    ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'  
    STORED AS TEXTFILE LOCATION '/user/tpchr/region';
```

LOAD DATA statement loads data into an external table

Loading data into an external table

```
LOAD DATA LOCAL INPATH 'region.tbl' INTO TABLE extregion;
```

Tables

An **external table** can be created "over" an already existing file in **HDFS**

Loading a file to HDFS

```
hadoop fs -mkdir /user/tpchr/nation
hadoop fs -put nation.tbl /user/tpchr/nation
hadoop fs -ls /user/tpchr/nation
-rw-r--r-- 3 janusz supergroup 401 2017-07-02 10:24 /user/tpchr/nation/nation.tbl
```

Creating an external table over a file in HDFS

```
CREATE EXTERNAL TABLE IF NOT EXISTS extnation(
    N_NATIONKEY DECIMAL(12),
    N_NAME      VARCHAR(25),
    N_COMMENT   VARCHAR(152) )
    ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'
    STORED AS TEXTFILE LOCATION '/user/tpchr/nation';
```

Hive Data Structures

Outline

[Primitive Data Types](#)

[Complex Data Types](#)

[Databases](#)

[Tables](#)

[Partitions](#)

[Buckets](#)

[Views](#)

Partitions

To eliminate unnecessary scans of entire table when only a fragment is needed a table can be divided into **partitions**

A **partition** corresponds to predefined columns and it is stored as subfolder in **HDFS**

When a table is searched only required **partitions** are accessed

Creating a partitioned table

```
CREATE TABLE IF NOT EXISTS part(
    P_PARTKEY DECIMAL(12),
    P_NAME VARCHAR(55),
    P_TYPE VARCHAR(25),
    P_SIZE DECIMAL(12),
    P_COMMENT VARCHAR(23) )
PARTITIONED BY (P_BRAND VARCHAR(20))
ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'
STORED AS TEXTFILE;
```

Partitions

A partition must be added before data is loaded

Adding a partition

```
ALTER TABLE part ADD PARTITION (P_BRAND='GoldenBolts');
```

Listing partitions

```
show partitions part;  
OK  
p_brand=GoldenBolts  
Time taken: 0.072 seconds, Fetched: 1 row(s)
```

A command that loads a file into a table can be used to load a partition

Loading data into a partition

```
LOAD DATA LOCAL INPATH '/local/home/janusz/HIVE-EXAMPLES/TPCHR/part.txt'  
OVERWRITE INTO TABLE part PARTITION (P_BRAND='GoldenBolts');
```

A partition is stored in HDFS as a subfolder

Finding a partition in HDFS

```
hadoop fs -ls /user/hive/warehouse/part
```

Found 1 items

```
drwxrwxr-x - janusz supergroup 0 2017-07-01
```

19:00 /user/hive/warehouse/part/p_brand=GoldenBolts

TOP

Created by Janusz R. Getta, ISIT312 Big Data Management, SIM, Session 2, 2021

18/24

Hive Data Structures

Outline

[Primitive Data Types](#)

[Complex Data Types](#)

[Databases](#)

[Tables](#)

[Partitions](#)

[Buckets](#)

[Views](#)

Buckets

Another way to speed up processing of a table is to divide it into **buckets**

A **bucket** corresponds to segment of file in **HDFS**

The values in a **bucket column** will be hashed by a user defined number into buckets.

Creating a table with buckets

```
CREATE TABLE customer(
    C_CUSTKEY DECIMAL(12),
    C_NAME VARCHAR(25),
    C_PHONE CHAR(15),
    C_ACCTBAL DECIMAL(12,2) )
CLUSTERED BY (C_CUSTKEY) INTO 2 BUCKETS
ROW FORMAT DELIMITED FIELDS TERMINATED BY '|';
```

Setting MapReduce and Hive parameters

```
set map.reduce.tasks = 2;
set hive.enforce.bucketing = true;
```

Buckets

INSERT should be used to populate a bucket table

Inserting into a table with buckets

```
INSERT INTO customer  
values(1, 'Customer#00000001', '25-989-741-2988', 711.56);  
INSERT INTO customer  
values(2, 'Customer#00000002', '23-768-687-3665', 121.65);  
INSERT INTO customer  
values(3, 'Customer#00000003', '11-719-748-3364', 7498.12);  
INSERT INTO customer  
values(4, 'Customer#00000004', '14-128-190-5944', 2866.83);  
INSERT INTO customer  
values(5, 'Customer#00000005', '13-750-942-6364', 794.47)
```

Hive Data Structures

Outline

[Primitive Data Types](#)

[Complex Data Types](#)

[Databases](#)

[Tables](#)

[Partitions](#)

[Buckets](#)

[Views](#)

Views

Views are logical data structures that simplify queries

Views do not store data or get materialized

Once a views is created its definition is frozen and changes in the tables used in the view definition are not reflected in the view schema

Creating a view

```
CREATE VIEW vcustomer AS
    SELECT C_CUSTKEY, C_NAME, C_PHONE
    FROM CUSTOMER
    WHERE C_CUSTKEY < 5;
```

References

Gross C., GuptaA., Shaw S., Vermeulen A. F., Kjerrumgaard D., Practical Hive: A guide to Hadoop's Data Warehouse System, Apress 2016, Chapter 4 (Available through UOW library)

Lee D., Instant Apache Hive essentials how-to: leverage your knowledge of SQL to easily write distributed data processing applications on Hadoop using Apache Hive, Packt Publishing Ltd. 2013 (Available through UOW library)

Apache Hive TM, <https://hive.apache.org/>

ISIT312 Big Data Management

Data Warehouse Concepts

Dr Janusz R. Getta

School of Computing and Information Technology -
University of Wollongong

Data Warehouse Concepts

Outline

OLAP versus OLTP

The Multidimensional Model

OLAP Operations

Data Warehouse Architecture

OLAP versus OLTP

Traditional database systems designed and tuned to support the day-to-day operation:

- Ensure fast, concurrent access to data
- Transaction processing and concurrency control
- Focus on online update data consistency
- Known as **operational databases** or **online transaction processing (OLTP)**

OLTP database characteristics:

- Detailed data
- Do not include historical data
- Highly normalized
- Poor performance on complex queries including joins and aggregation

Data analysis requires a new paradigm: **online analytical processing (OLAP)**

- Typical **OLTP** query: pending orders for a customer
- Typical **OLAP** query: total sales amount by a product and by a customer

OLAP versus OLTP

OLAP characteristics

- OLTP paradigm focused on transactions, OLAP focused on analytical queries
- Normalization not good for analytical queries, reconstructing data requires a high number of joins
- OLAP databases support a heavy query load
- OLTP indexing techniques not efficient in OLAP: oriented to access few records; OLAP queries typically include aggregation

The need for a different database model to support OLAP was clear: led to **data warehouses**

Data warehouse: (usually) large repositories that consolidate data from different sources (internal and external to the organization), are updated offline, follow the **multidimensional data model**, designed and optimized to efficiently support **OLAP** queries

Data Warehouse Concepts

Outline

[OLAP versus OLTP](#)

[The Multidimensional Model](#)

[OLAP Operations](#)

[Data Warehouse Architecture](#)

The Multidimensional Model

A view of data in n-dimensional space: a **data cube**

A **data cube** is composed of **dimensions** and **facts**

Dimensions: Perspectives used to analyze the data

- Example: A three-dimensional cube for sales data with dimensions **Product**, **Time**, and **Customer**, and a measure **Quantity**

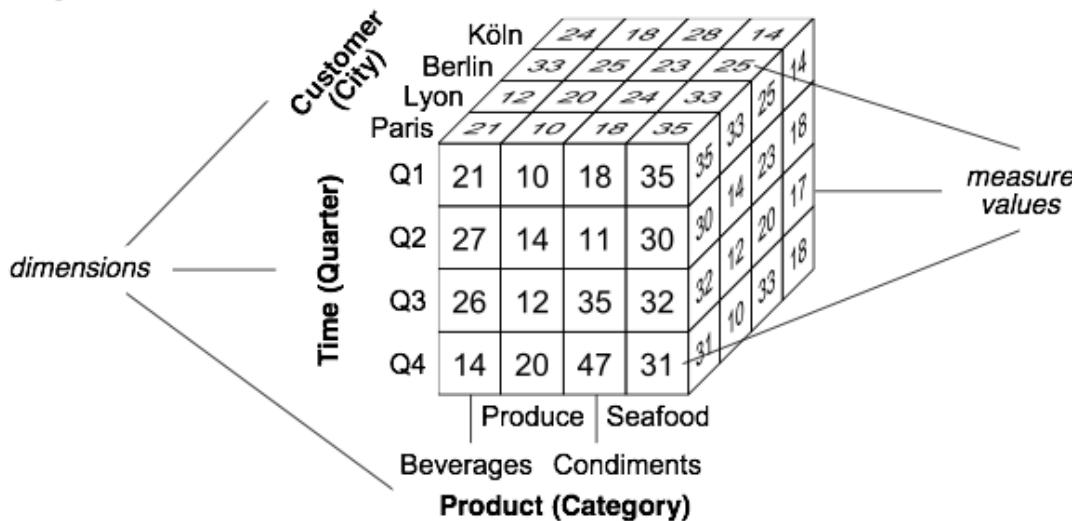
The diagram illustrates a 3D data cube with dimensions and measures. The vertical axis is labeled 'Time (Quarter)' with categories Q1, Q2, Q3, and Q4. The horizontal axis is labeled 'Customer (City)' with categories Köln, Berlin, Lyon, and Paris. The depth axis is labeled 'Product (Category)' with categories Produce, Beverages, Condiments, and Seafood. The cube contains numerical values representing 'measure values' at each intersection of the dimensions. Lines from the text 'dimensions' and 'measure values' point to their respective axes and labels.

Köln	24	18	28	14
Berlin	33	25	23	25
Lyon	12	20	24	33
Paris	21	10	18	35
Q1	21	10	18	35
Q2	27	14	11	30
Q3	26	12	35	32
Q4	14	20	47	31
	Produce	Seafood		
	Beverages	Condiments		
	Product (Category)			
				measure values
				dimensions

Attributes describe dimensions

- [TOP](#) - Product dimension may have attributes **ProductNumber** and **UnitPrice** (not shown in the figure)

The Multidimensional Model



The **cells** or **facts** of a data cube have associated numeric values called **measures**

Each **cell** of the **data cube** represents **Quantity** of units sold by **category**, **quarter**, and **customer's city**

Data granularity: level of detail at which measures are represented for each dimension of the cube

- TOP - Example: sales figures aggregated to granularities **Category**, **Quarter**, and **City**

The Multidimensional Model

Instances of a dimension are called **members**

- Example: **Seafood** and **Beverages** are **members** of the **Product** at the granularity **Category**

A **data cube** contains several measures, e.g. **Amount**, indicating the total sales amount (not shown)

A **data cube** may be **sparse** (typical case) or **dense**

- Example: not all customers may have ordered products of all categories during all quarters

Hierarchies: allow viewing data at several granularities

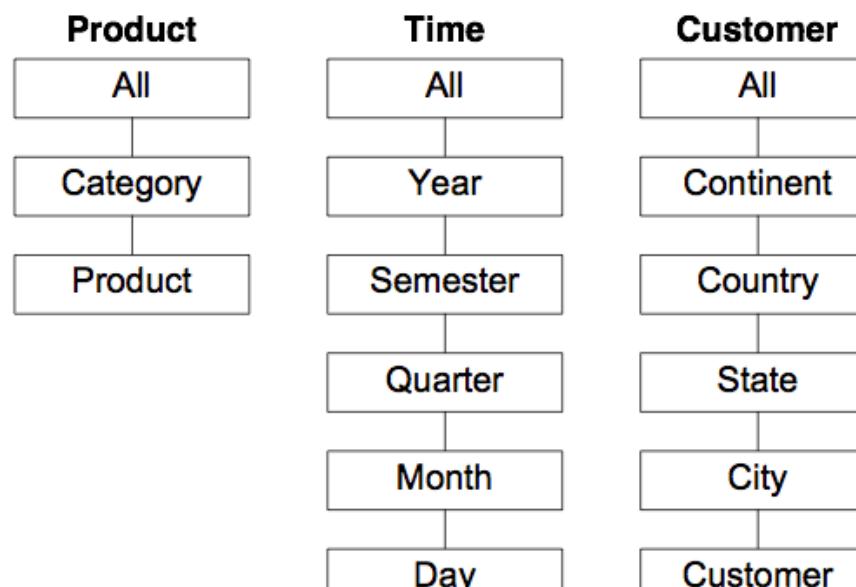
- Define a sequence of mappings relating lower-level, detailed concepts to higher-level ones
- The lower level is called the **child** and the higher level is called the **parent**
- The hierarchical structure of a dimension is called the dimension **schema**
- A dimension **instance** comprises all members at all levels in a dimension

The Multidimensional Model

In the previous figure, granularity of each dimension indicated between parentheses: Category for the **Product** dimension, Quarter for **Time**, and City for **Customer**

We may want sales figures at a finer granularity (**Month**), or at a coarser granularity (**Country**)

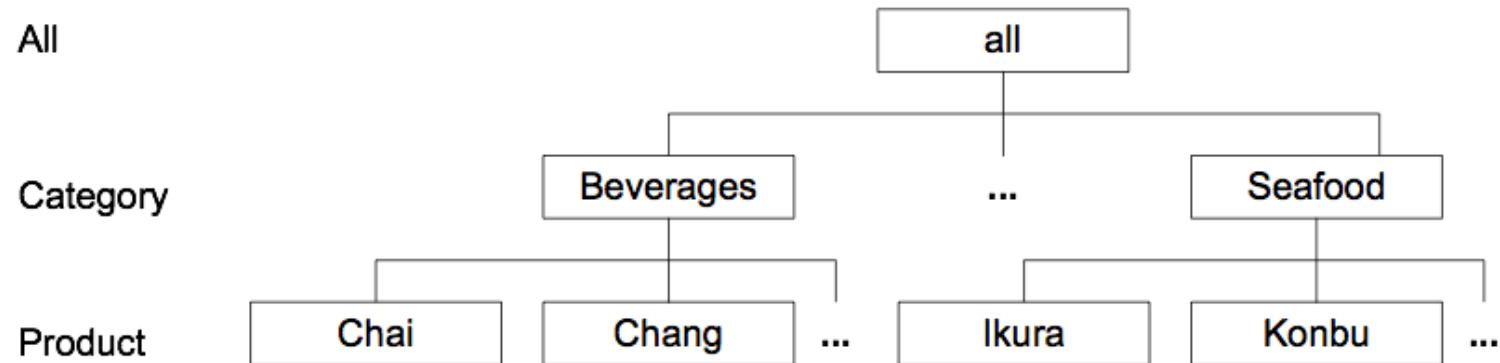
Hierarchies of the **Product**, **Time**, and **Customer** dimensions



[TOP](#)

The Multidimensional Model

Members of a hierarchy **Product - Category**



The Multidimensional Model: Measures

Aggregation of measures changes the abstraction level at which data in a cube are visualized

Measures can be:

- **Additive**: can be meaningfully summarized along all the dimensions, using addition; The most common type of measures
- **Semiadditive**: can be meaningfully summarized using addition along some dimensions; Example: inventory quantities, which cannot be added along the Time dimension
- **Nonadditive measures** cannot be meaningfully summarized using addition across any dimension; Example: item price, cost per unit, and exchange rate

The Multidimensional Model: Measures

Another classification of measures:

- **Distributive**: defined by an aggregation function that can be computed in a distributed way; Functions `count`, `sum`, `minimum`, and `maximum` are distributive, `distinct count` is not; Example: $S = \{3, 3, 4, 5, 8, 4, 7, 3, 8\}$ partitioned in subsets $\{3, 3, 4\}$, $\{5, 8, 4\}$, $\{7, 3, 8\}$ gives a result of 8, while the answer over the original set is 5
- **Algebraic measures** are defined by an aggregation function that can be expressed as a scalar function of distributive ones; example: `average`, computed by dividing the sum by the count

Data Warehouse Concepts

Outline

[OLAP versus OLTP](#)

[The Multidimensional Model](#)

[OLAP Operations](#)

[Data Warehouse Architecture](#)

OLAP Operations

		Customer (City)			
		Köln	18	28	14
Time (Quarter)	Berlin	33	25	23	25
	Lyon	12	20	24	33
	Paris	21	10	18	35
	Q1	21	10	18	35
	Q2	27	14	11	30
		35	33	33	18
		35	14	23	17
		30	12	20	18
		32	10	33	18
		31	11	33	18
		Produce	Seafood		
		Beverages	Condiments		
Product (Category)					

Original cube

		Customer (City)			
		Köln	6	9	5
Time (Quarter)	Berlin	10	8	11	8
	Lyon	4	7	8	14
	Paris	7	2	6	20
	Jan	7	2	6	20
	Feb	8	4	8	8
	Mar	6	4	4	7

	Dec	4	4	16	7
		7	7	7	7
		7	10	10	3
		Produce	Seafood		
		Beverages	Condiments		
Product (Category)					

Drill-down to the Month level

		Customer (Country)			
		Germany	57	43	51
Time (Quarter)	France	33	30	42	68
	Q1	33	30	42	68
	Q2	39	26	41	44
	Q3	30	22	46	44
	Q4	25	29	49	41
		Produce	Seafood		
		Beverages	Condiments		
Product (Category)					

Roll-up to the Country level

		Customer (City)			
		Köln	24	28	18
Time (Quarter)	Berlin	33	23	25	25
	Lyon	12	24	20	33
	Paris	21	18	10	35
	Q1	21	18	10	35
	Q2	27	11	14	30
	Q3	26	35	12	32
	Q4	14	47	20	31
		35	14	23	17
		30	12	20	18
		32	10	33	18
		Condiments	Seafood		
		Beverages	Produce		
Product (Category)					

Sort product by name

OLAP Operations

Starting cube: quarterly sales (in thousands) by product category and customer cities for 2012

We first compute the sales quantities by country: a **roll-up** operation to the **Country** level along the **Customer** dimension

Sales of category Seafood in France significantly higher in the first quarter

- To find out if this occurred during a particular month, we take cube back to **City** aggregation level, and **drill-down** along **Time** to the **Month** level

To explore alternative visualizations, we **sort** products by name

To see the cube with the **Time** dimension on the x axis, we rotate the axes of the original cube, without changing granularities → **pivoting** (see next 2 slides)

OLAP Operations

To visualize the data only for Paris → **slice** operation, results in a 2-dimensional sub-cube, basically a collection of time series (see next slide)

To obtain a 3-dimensional sub-cube containing only sales for the first two quarters and for the cities Lyon and Paris, we go back to the original cube and apply a **dice** operation

OLAP Operations

Pivot

		Time (Quarter)				Customer (City)	Product (Category)			
		Q1	Q2	Q3	Q4		Seafood	Condiments	Produce	Beverages
Customer (City)	Paris	21	27	26	14	35	30	32	31	
	Lyon	12	14	11	13	18	17	21	33	
	Berlin	33	28	35	32	32	28	20	18	
	Köln	24	23	25	18	18	19	47	31	
		Q1	Q2	Q3	Q4	Seafood	Condiments	Produce	Beverages	

Slice on City='Paris'

	Time (Quarter)				Customer (City)	Product (Category)			
	Q1	Q2	Q3	Q4		Seafood	Condiments	Produce	Beverages
Q1	21	10	18	35	Paris	35	30	31	33
Q2	27	14	11	30		32	28	20	18
Q3	26	12	35	32		33	21	17	14
Q4	14	20	47	31		31	35	27	23

		Time (Quarter)				Customer (City)	Product (Category)			
		Q1	Q2	Q3	Q4		Seafood	Condiments	Produce	Beverages
Customer (City)	Paris	21	10	18	35	35	30	31	33	
	Lyon	12	20	24	33	32	28	20	18	
		21	10	18	35	35	30	31	33	

Dice on City='Paris' or 'Lyon' and Quarter='Q1' or 'Q2'

[TOP](#)

Created by Janusz R. Getta, ISIT312 Big Data Management, SIM, Session 2, 2021

17/26

OLAP Operations

The operations in the previous slides can be defined using the following algebraic operators.

Roll-up: aggregates measures along a dimension hierarchy (using an aggregate function) to obtain measures at a coarser granularity

```
ROLLUP(CubeName, (Dimension → Level)*, AggFunction(Measure)*)
ROLLUP(Sales, Customer → Country, SUM(Quantity))
```

OLAP

Extended roll-up: similar to rollup, but drops all dimensions not involved in the operation

```
ROLLUP*(CubeName, [(Dimension → Level)*], AggFunction(Measure)*)
ROLLUP*(Sales, Time → Quarter, SUM(Quantity))
ROLLUP*(Sales, Time → Quarter, COUNT(Product) AS ProdCount)
```

OLAP

Recursive roll-up: aggregates over a recursive hierarchy (a level rolls-up to itself)

```
RECROLLUP(CubeName, Dimension → Level, AggFunction(Measure)*)
```

OLAP

OLAP Operations

Drill-down moves from a more general level to a more detailed level in a hierarchy

```
DRILLDOWN(CubeName, (Dimension → Level)*)  
DRILLDOWN(Sales, Time → Month)
```

OLAP

Sort returns a cube where the members of a dimension have been sorted according to the value of Expression

```
SORT(CubeName, Dimension, Expression [ASC | DESC])  
SORT(Sales, Product, NAME)
```

OLAP

- **NAME** is a predefined keyword in the algebra representing the name of a member

OLAP Operations

Pivot

`PIVOT(CubeName, (Dimension → Axis)*)`

OLAP

- where the axes are specified as {X, Y, Z, X₁, Y₁, Z₁, ... }.

`PIVOT(Sales, Time → X, Customer → Y, Product → Z)`

OLAP

Slice:

`SLICE(CubeName, Dimension, Level = Value)`

OLAP

- Dimension will be dropped by fixing a single Value in the Level, other dimensions unchanged

`SLICE(Sales, Customer, City = 'Paris')`

OLAP

- Slice supposes that the granularity of the cube is at the specified level of the dimension

OLAP Operations

Dice:

DICE(CubeName, ?)

OLAP

- where ? is a Boolean condition over dimension levels, attributes, and measures.

DICE(Sales, (Customer.City = 'Paris' OR Customer.City = 'Lyon') AND
(Time.Quarter = 'Q1' OR Time.Quarter = 'Q2'))

OLAP

Data Warehouse Concepts

Outline

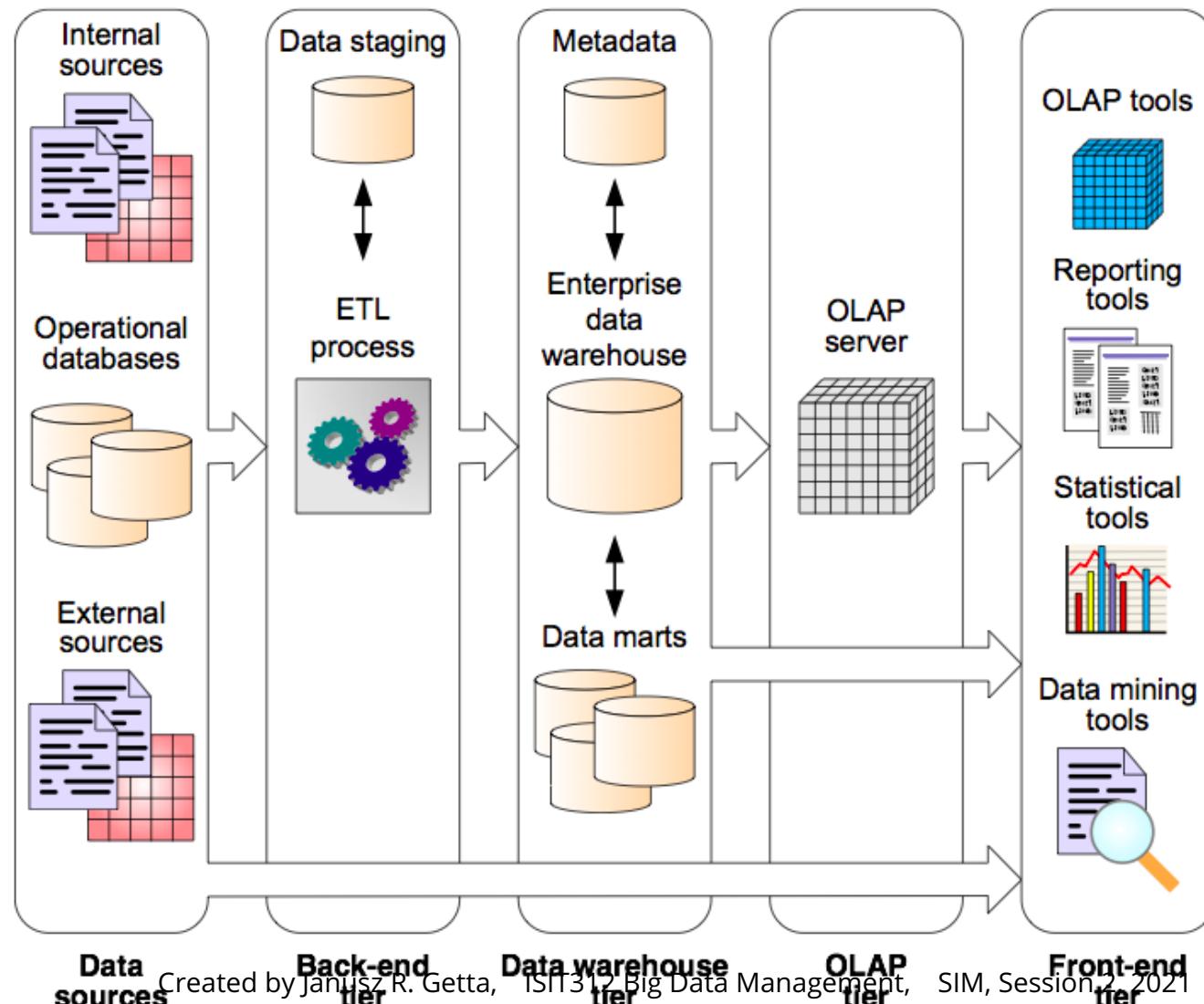
[OLAP versus OLTP](#)

[The Multidimensional Model](#)

[OLAP Operations](#)

[Data Warehouse Architecture](#)

Typical Data Warehouse Architecture



Data Warehouse Architecture

General data warehouse architecture: **several tiers**

Back-end tier composed of:

- The **extraction, transformation, and loading (ETL)** tools: Feed data into the data warehouse from operational databases and internal and external data sources
- The **data staging area**: An intermediate database where all the data integration and transformation processes are run prior to the loading of the data into the data warehouse

Data warehouse tier composed of:

- An **enterprise data warehouse** and/or **several data marts**
- A **metadata repository** storing information about the data warehouse and its contents

OLAP tier composed of:

- An **OLAP server** which provides a multidimensional view of the data, regardless the actual way in which data are stored

Data Warehouse Architecture

Front-end tier is used for data analysis and visualization

- Contains client tools such as OLAP tools, reporting tools, statistical tools, and data-mining tools

References

A. VAISMAN, E. ZIMANYI, Data Warehouse Systems: Design and Implementation, Chapter 3 Data Warehouse Concepts, Springer Verlag, 2014

ISIT312 Big Data Management Conceptual Data Warehouse Design

Dr Janusz R. Getta

School of Computing and Information Technology -
University of Wollongong

Conceptual Data Warehouse Design

Outline

MultiDim: A Conceptual Model for Data Warehouses

MultiDim Model: Notation

Dimension Hierarchies

MultiDim: A Conceptual Multidimensional Model

Conceptual data models

- Allow better communication between designers and users to understand application requirements
- More stable than implementation-oriented (logical) schema, which changes with the platform
- Provide better support for visual user interfaces

No well-established conceptual model for multidimensional data

Several proposals based on UML, on the ER model, or using specific notations

Problems:

- Cannot express complex kinds of hierarchies
- Lack of a mapping to the implementation platform

MultiDim: A Conceptual Multidimensional Model

Currently, data warehouses are designed using mostly logical models (star and snowflake schemas)

- Difficult to express requirements (technical knowledge required)
- Limit users to defining only elements that the underlying implementation systems can manage

MultiDim data model is based on the entity-relationship model

Includes concepts like:

- dimensions
- hierarchies
- facts
- measures

Supports various kinds of hierarchies existing in real-world applications

Can be mapped to star or snowflake relational structures

Conceptual Datawarehouse Design

Outline

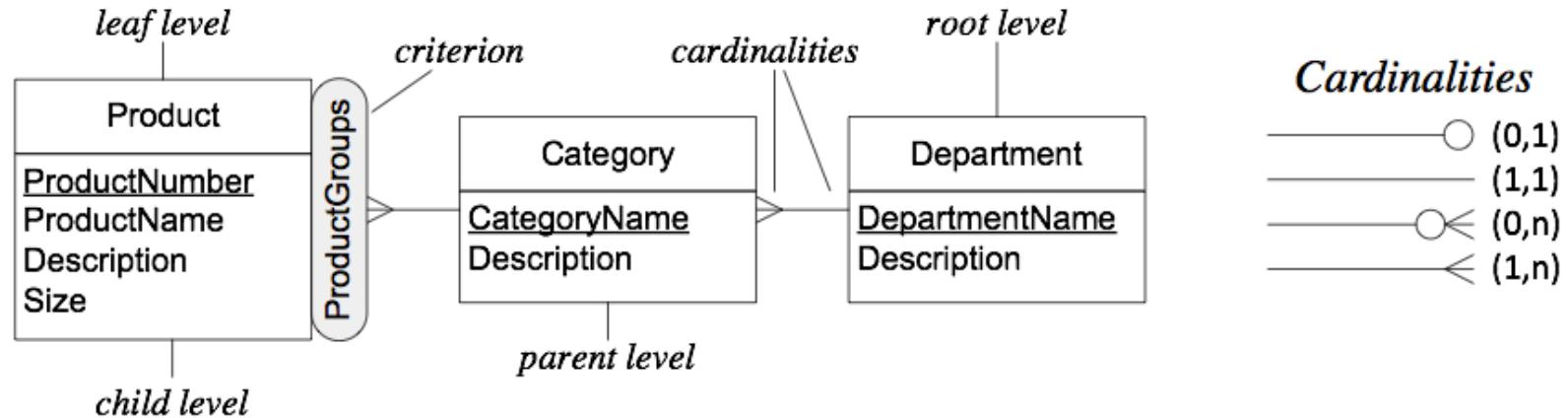
[MultiDim: A Conceptual Model for Data Warehouses](#)

[MultiDim Model: Notation](#)

[Dimension Hierarchies](#)

MultiDim Model: Notation

A graphical notation used for a sample **hierarchy**



Dimension: level or one or more **hierarchies**

Hierarchy: several related levels

Level: entity type

Member: every instance of a level

Child and parent levels: the lower and higher levels

Leaf and root levels: first and last levels in a hierarchy

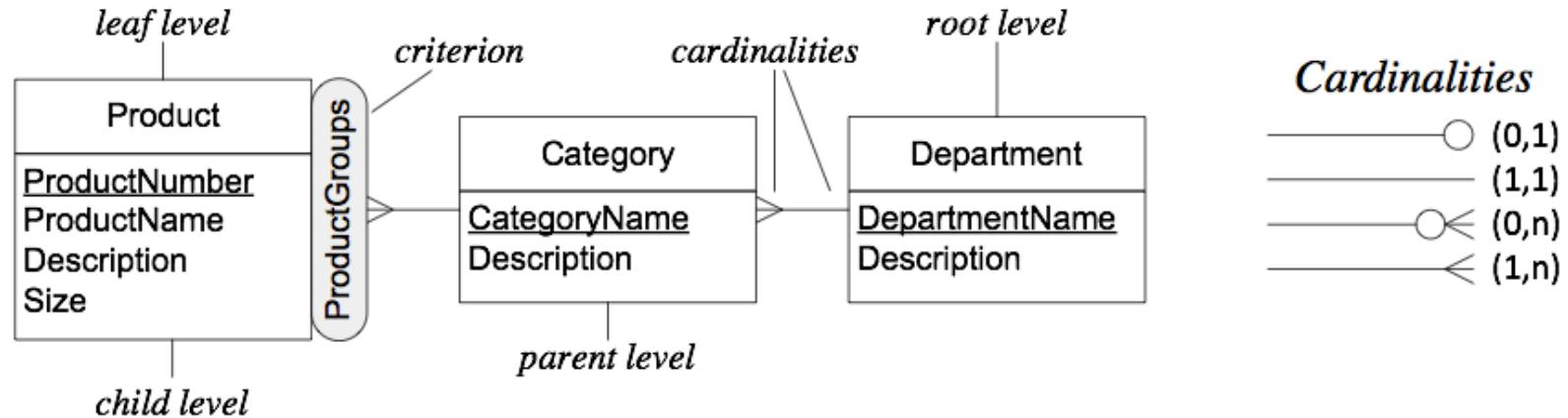
[TOP](#)

Created by Janusz R. Getta, ISIT312 Big Data Management, SIM, Session 2, 2021

6/18

MultiDim Model: Notation

A graphical notation used for a sample **hierarchy**



Cardinality: minimum/maximum numbers of members in a level related to members in another level

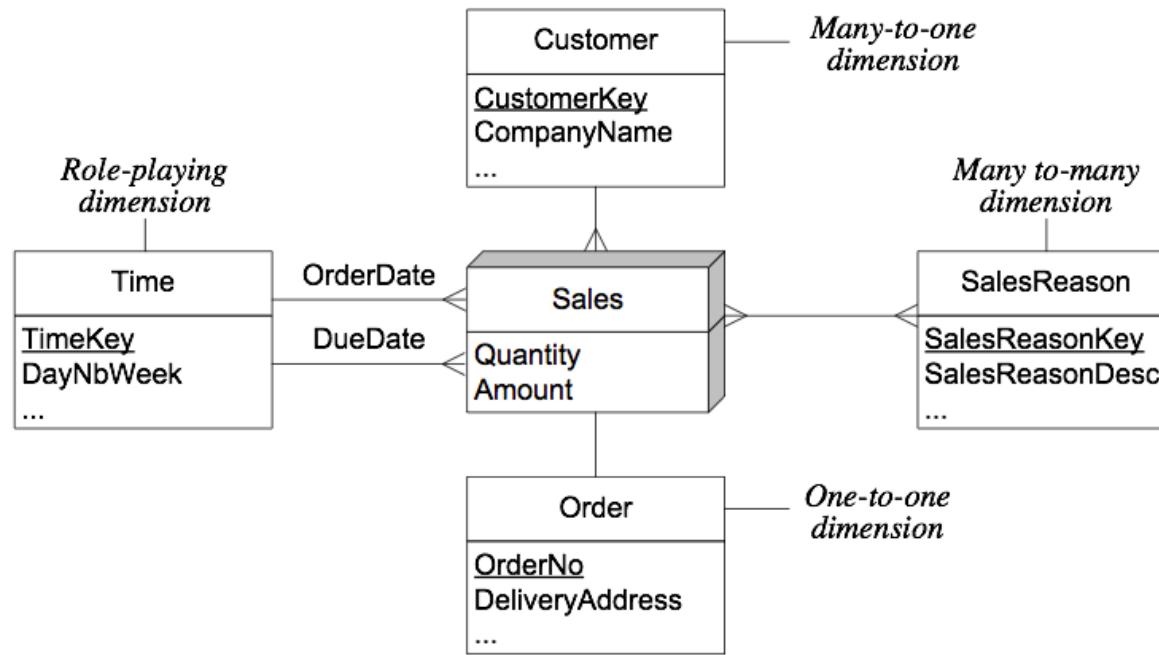
Criterion: expresses different hierarchical structures used for analysis

Key attribute: indicates how child members are grouped

Descriptive attributes: describe characteristics of members

MultiDim Model: Notation

A sample **fact** with 5 dimensions



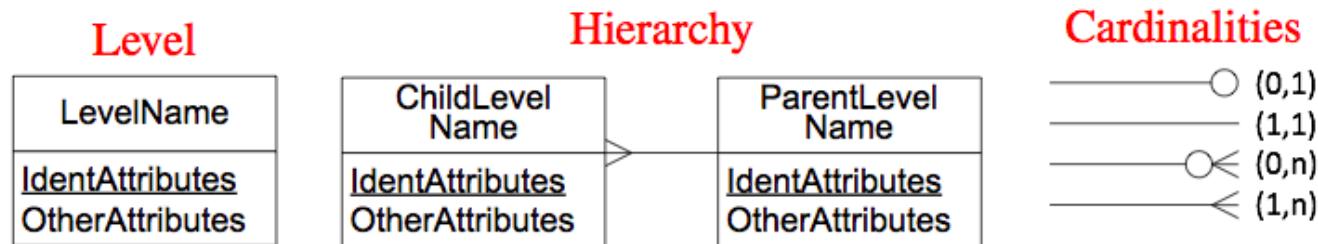
Fact: relates measures to leaf levels in dimensions

Dimensions can be related to fact with **one-to-one**, **one-to-many**, or **many-to-many**

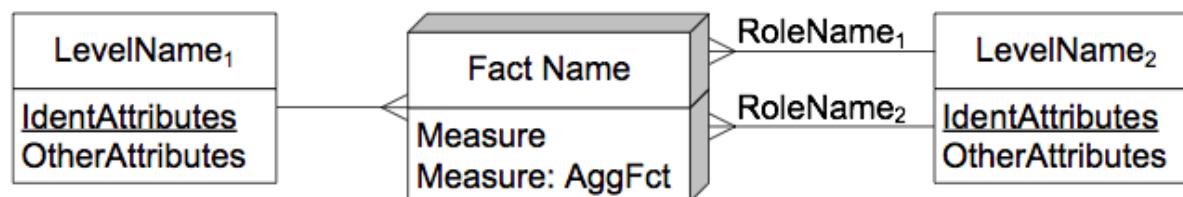
[TOP](#) Dimension can be related several times to a fact with **different roles**

MultiDim Model: Notation

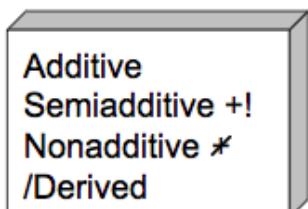
Summary



Fact with measures and associated levels



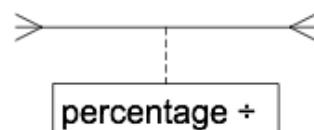
Types of measures



Analysis criterion



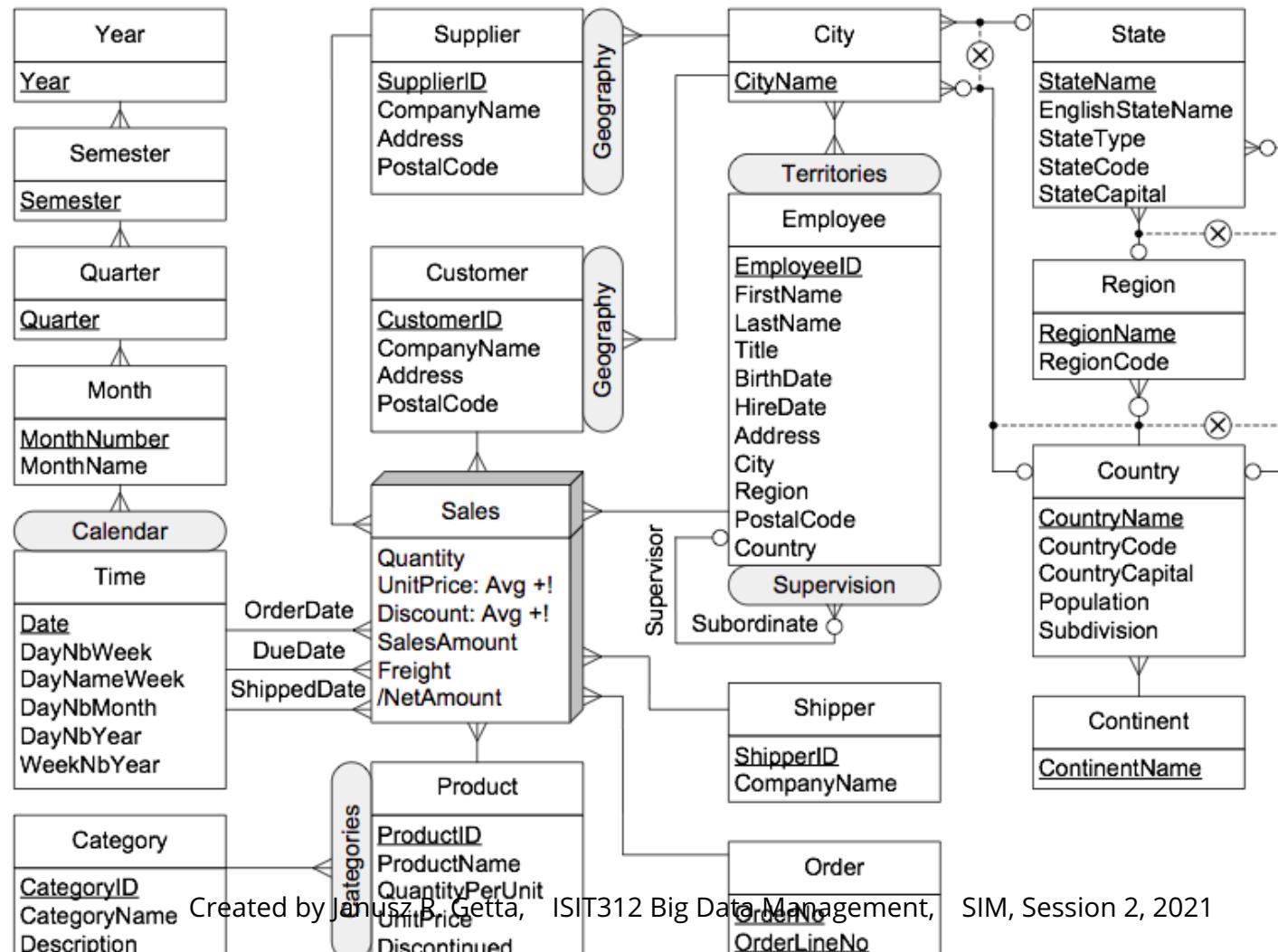
Distributing factor



Exclusive relationships



MultiDim Conceptual Schema of the Northwind Data Warehouse



Conceptual Data Warehouse Design

Outline

[MultiDim: A Conceptual Model for Data Warehouses](#)

[MultiDim Model: Notation](#)

[Dimension Hierarchies](#)

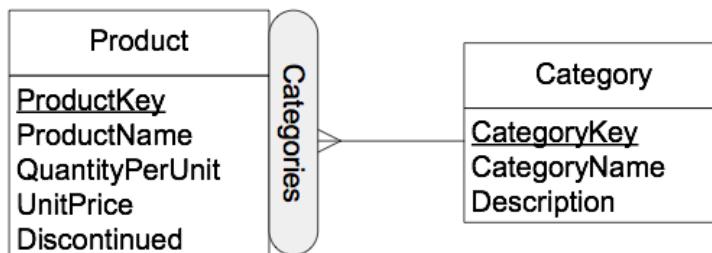
[TOP](#)

Created by Janusz R. Getta, ISIT312 Big Data Management, SIM, Session 2, 2021

11/18

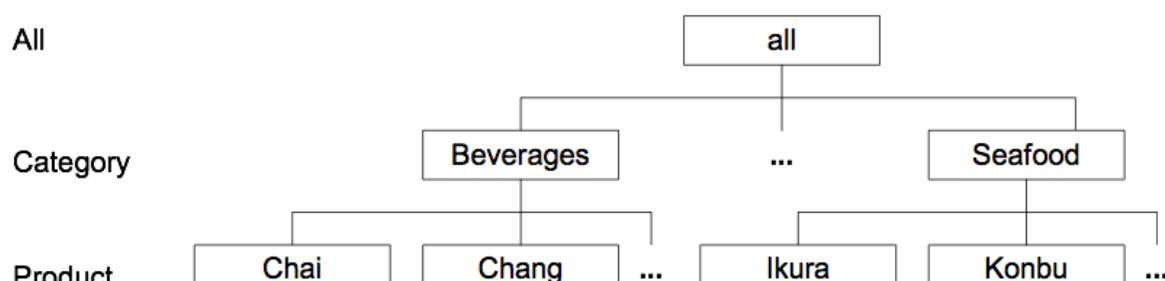
Balanced Hierarchies

At **schema level**: only one path where all parent-child relationships are many-to-one and mandatory



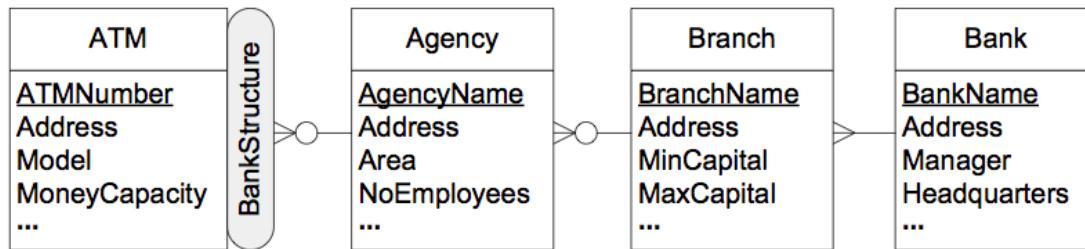
At **instance level**: members form a balanced tree (all the branches have the same length)

All parent members have at least one child member, and a child belongs exactly to one parent

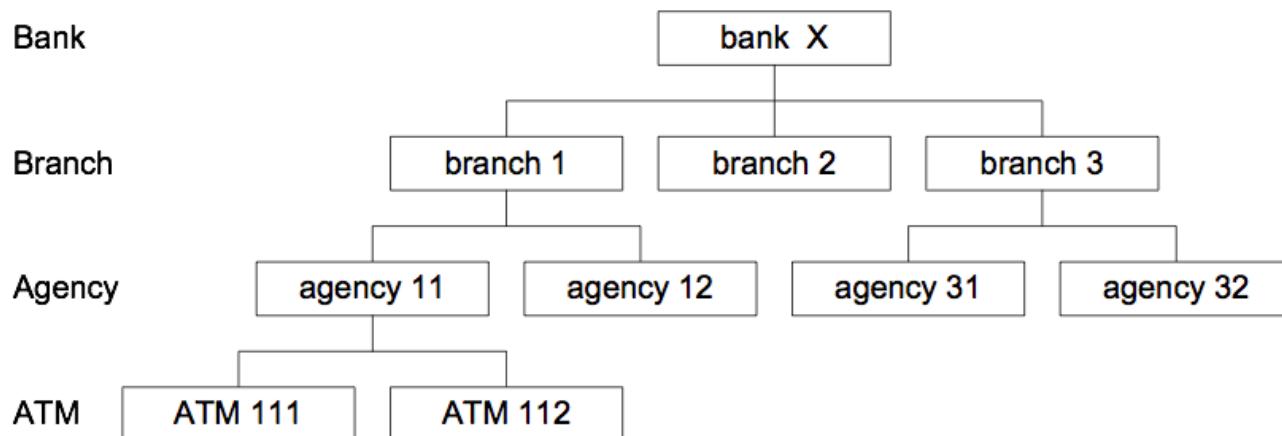


Unbalanced Hierarchies

At **schema level**: one path where all parent-child relationships are many-to-one, but some are optional



At **instance level**: members form a unbalanced tree



Recursive Hierarchies

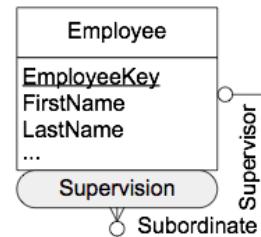
A special case of unbalanced hierarchies

The **same level** is linked by the two roles of a parent-child relationship

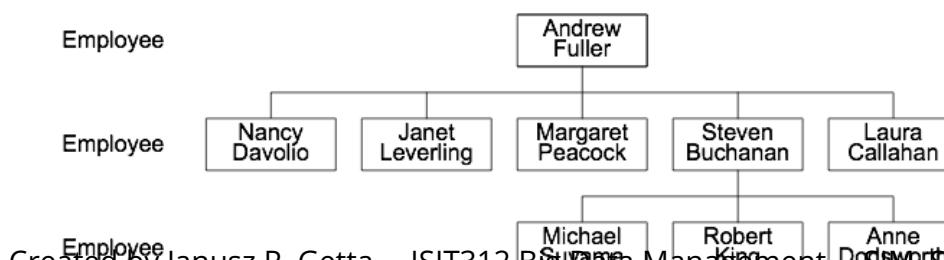
Used when all hierarchy levels express the same semantics

The characteristics of the parent and child are similar (or the same)

Schema level



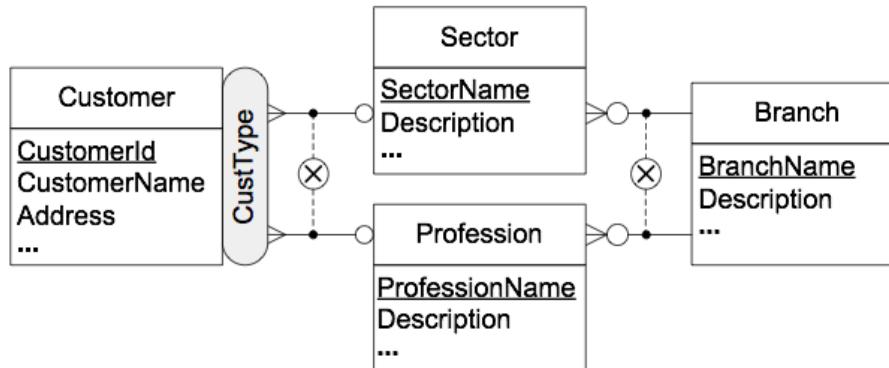
Instance level



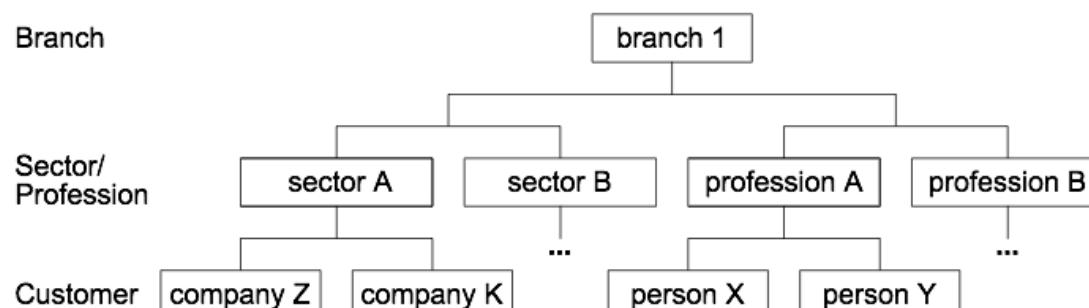
Generalized Hierarchies

At **schema level**: multiple exclusive paths sharing at least the leaf level; may also share other levels

Two aggregation paths, one for each type of customer



At **instance level**: each member belongs to only one path

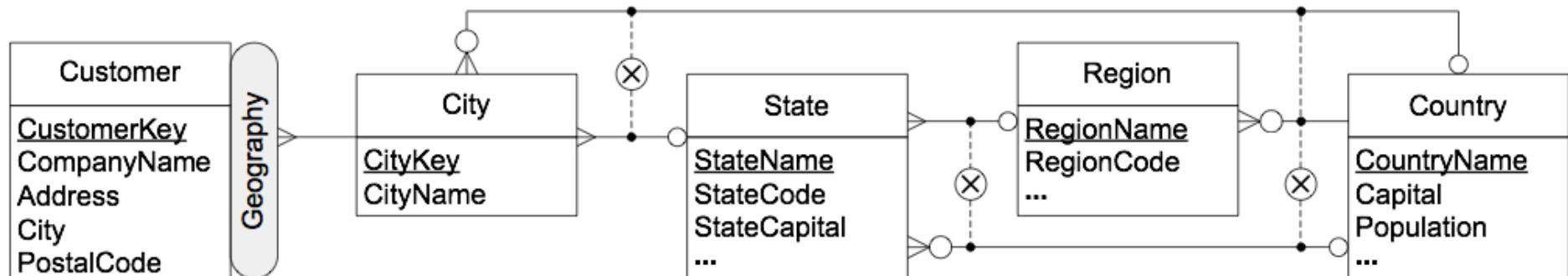


Noncovering Hierarchies

Also known as **ragged** or **level-skipping** hierarchies

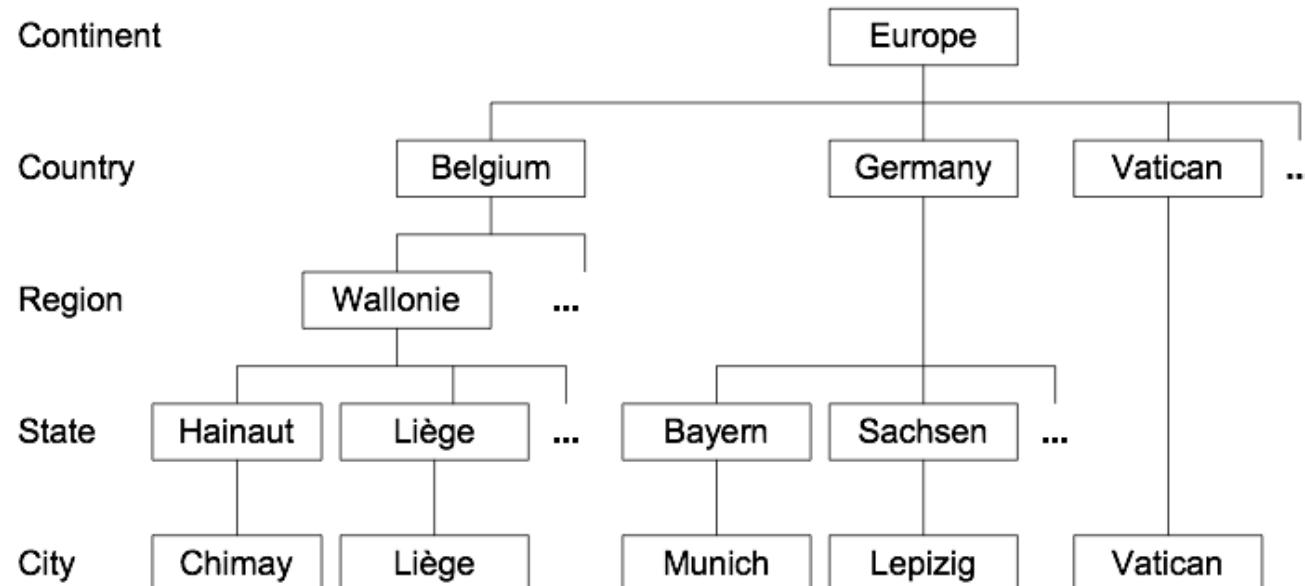
A **special case of generalized hierarchies**

At the **schema level**: Alternative paths are obtained by skipping one or several intermediate levels



Noncovering Hierarchies

At **instance level**: Path length from the leaves to the same parent can be different for different members



References

A. VAISMAN, E. ZIMANYI, Data Warehouse Systems: Design and Implementation, Chapter 4 Conceptual Data Warehouse Design, Springer Verlag, 2014

ISIT312 Big Data Management

Logical Data Warehouse Design

Dr Janusz R. Getta

School of Computing and Information Technology -
University of Wollongong

Logical Data Warehouse Design

Outline

OLAP Technologies

Relational Data Warehouse Design

Relational Implementation of Conceptual Model

The Time Dimension

OLAP Technologies

Relational OLAP (ROLAP): Stores data in relational databases, supports extensions to SQL and special access methods to efficiently implement the model and its operations

Multidimensional OLAP (MOLAP): Stores data in special data structures (e.g., arrays) and implement OLAP operations in these structures

- Better performance than ROLAP for query and aggregation, less storage capacity than ROLAP

Hybrid OLAP (HOLAP): Combines both technologies

- For example, detailed data stored in relational databases, aggregations kept in a separate **MOLAP** store, etc

Logical Data Warehouse Design

Outline

[OLAP Technologies](#)

[Relational Data Warehouse Design](#)

[Relational Implementation of Conceptual Mode](#)

[The Time Dimension](#)

Relational Data Warehouse Design

In **ROLAP** systems, tables organized in specialized structures

Star schema: One **fact table** and a set of **dimension tables**

- Referential integrity constraints between fact table and dimension tables
- Dimension tables may contain redundancy in the presence of hierarchies
- Dimension tables denormalized, fact tables normalized

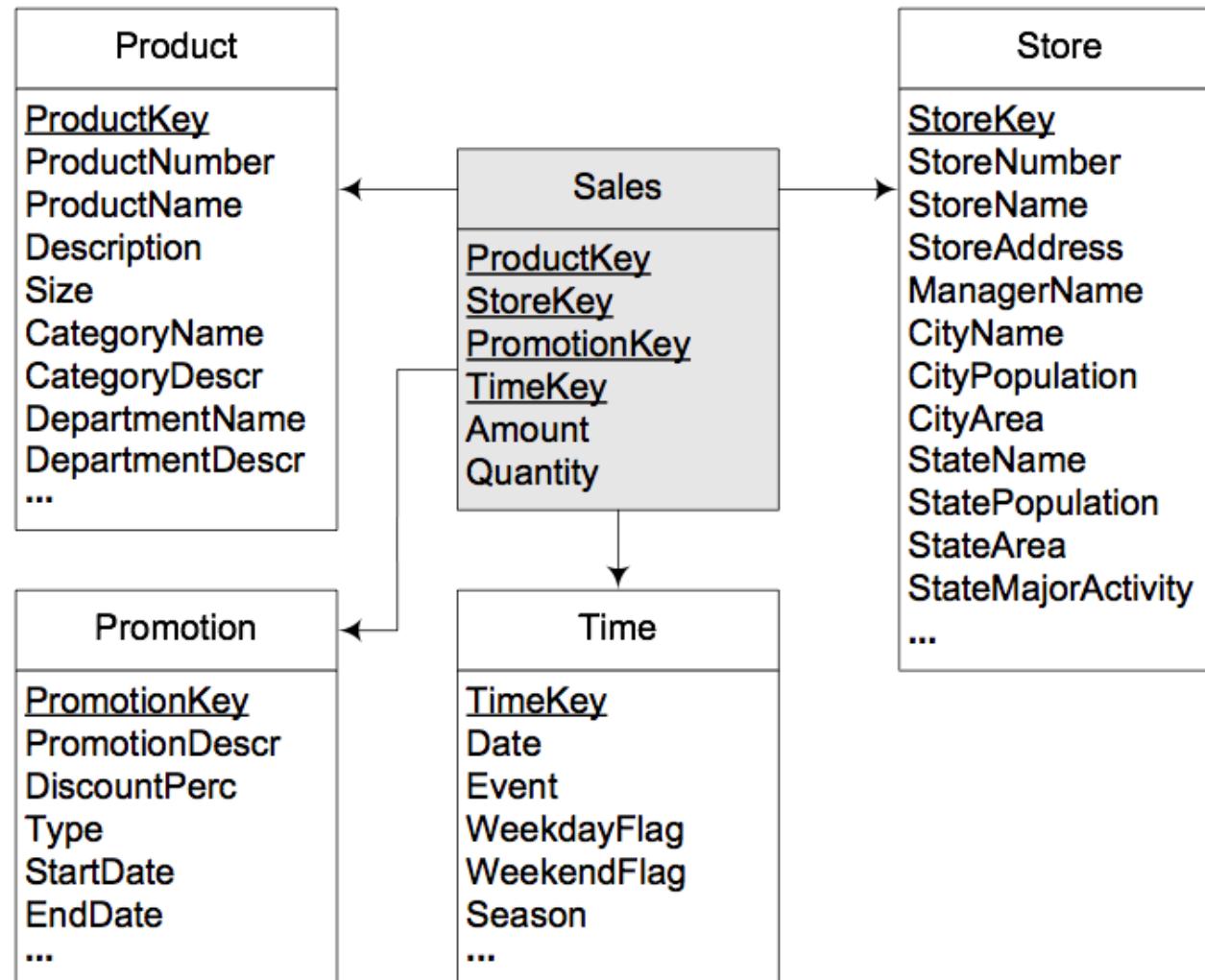
Snowflake schema: Avoids redundancy of star schemas by normalizing dimension tables

- Normalized tables optimize storage space, but decrease performance

Starflake schema: Combination of the star and snowflake schemas, some dimensions normalized, other not

Constellation schema: Multiple fact tables that share dimension tables

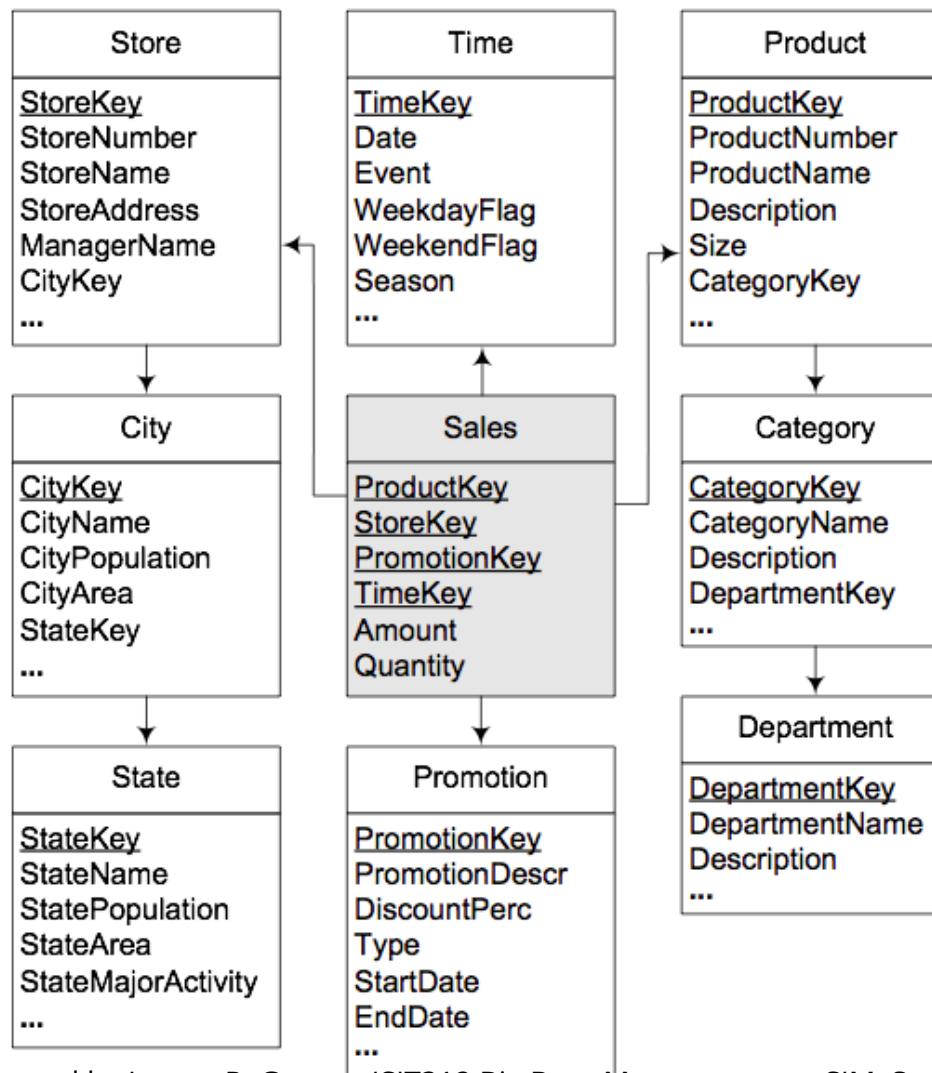
Example of a Star Schema

[TOP](#)

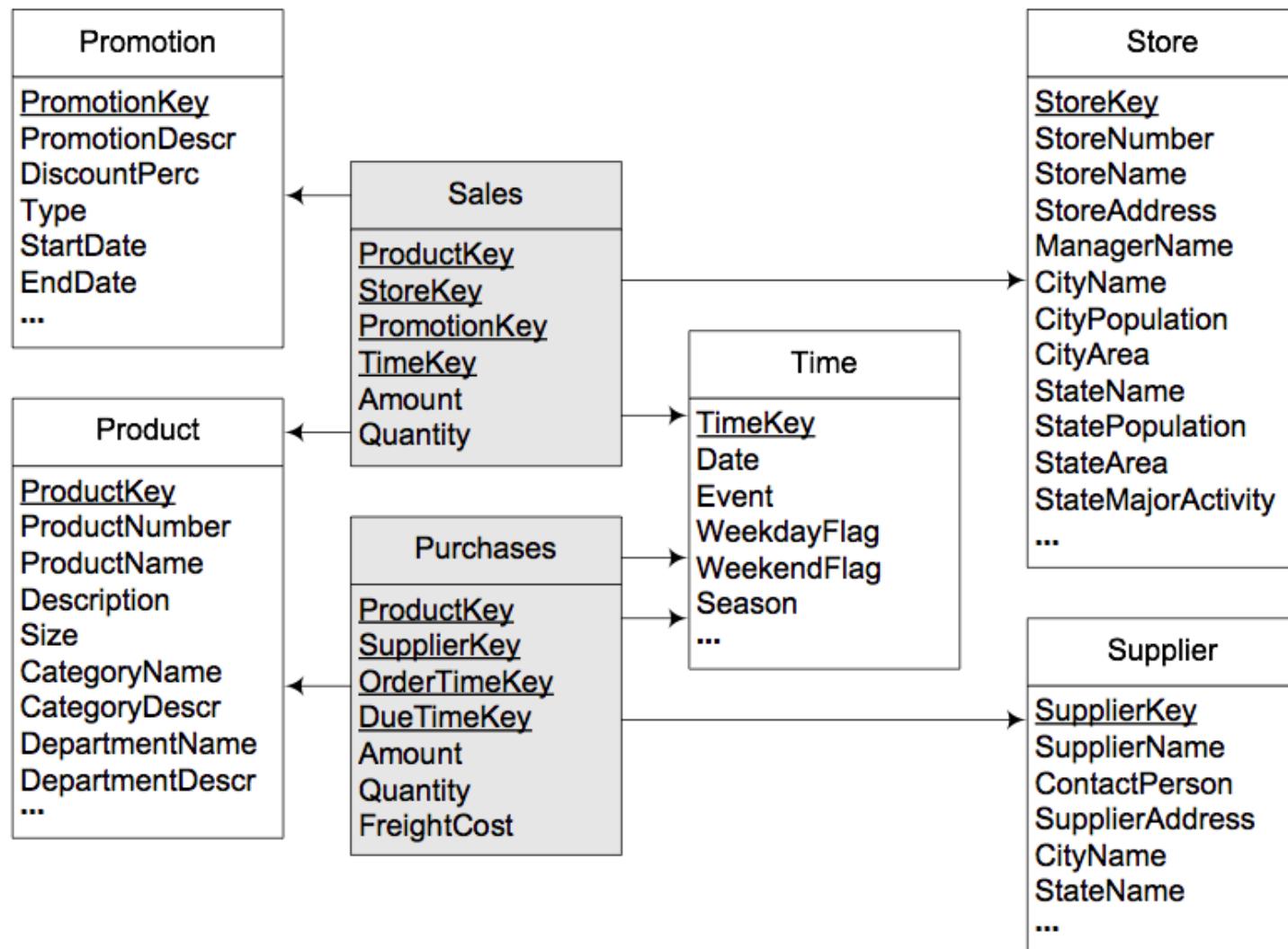
Created by Janusz R. Getta, ISIT312 Big Data Management, SIM, Session 2, 2021

6/18

Example of a Snowflake Schema



Example of a Constellation Schema



Logical Data Warehouse Design

Outline

[OLAP Technologies](#)

[Relational Data Warehouse Design](#)

[Relational Implementation of Conceptual Model](#)

[The Time Dimension](#)

Relational Implementation of the Conceptual Model

A set of rules to translate the conceptual model (the **MultiDim model**) into the relational mode

Rule 1: A level L, provided it is not related to a fact with a one-to-one relationship, is mapped to a table TL that contains all attributes of the level

- A surrogate key may be added to the table, otherwise the identifier of the level will be the key of the table
- Additional attributes will be added to this table when mapping relationships using Rule 3 below

Rule 2: A fact F is mapped to a table TF that includes as attributes all measures of the fact

- A surrogate key may be added to the table
- Additional attributes will be added to this table when mapping relationships using Rule 3 below

Relational Implementation of the Conceptual Model

Rule 3: A relationship between either a fact F and a dimension level L, or between dimension levels LP and LC (standing for the parent and child levels, respectively), can be mapped in three different ways, depending on its cardinalities:

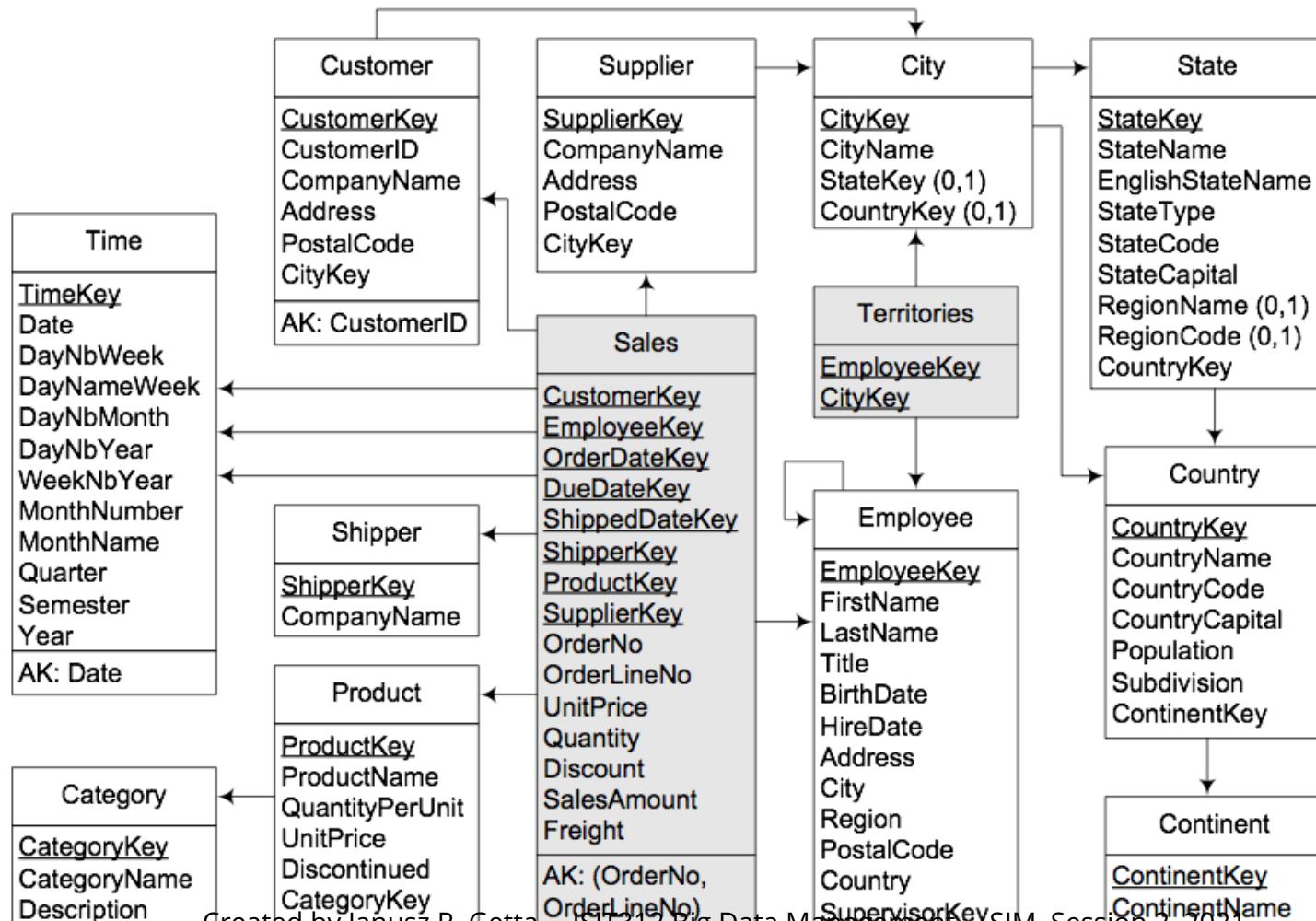
Rule 3a: If the relationship is one-to-one, the table corresponding to the fact (TF) or to the child level (TC) is extended with all the attributes of the dimension level or the parent level, respectively

Rule 3b: If the relationship is one-to-many, the table corresponding to the fact (TF) or to the child level (TC) is extended with the surrogate key of the table corresponding to the dimension level (TL) or the parent level (TP), respectively, that is, there is a foreign key in the fact or child table pointing to the other table

Relational Implementation of the Conceptual Model

Rule 3c: If the relationship is many-to-many, a new table TB (standing for bridge table) is created that contains as attributes the surrogate keys of the tables corresponding to the fact (TF) and the dimension level (TL), or the parent (TP) and child levels (TC), respectively. If the relationship has a distributing attribute, an additional attribute is added to the table to store this information

Relational Representation of the Northwind Data Warehouse

[TOP](#)

13/18

Relational Representation of the Northwind Data Warehouse

The **Sales** table includes one FK for each level related to the fact with a one-to-many relationship

For **Time**, several roles: **OrderDate**, **DueDate**, and **ShippedDate**

Order: related to the fact with a one-to-one relationship, called a **degenerate**, or a **fact dimension**

Fact table contains five attributes representing the measures:

- **UnitPrice**, **Quantity**, **Discount**, **SalesAmount**, and **Freight**.

The many-to-many parent-child relationship between **Employee** and **Territory** is mapped to the table **Territories**, containing two foreign keys

Customer has a surrogate key **CustomerKey** and a database key **CustomerAltKey**

[TOP](#) **SupplierKey** in **Suppliers** is a database key

Created by Janusz R. Gostylewski, Database Management, SIM, Session 2, 2021

14/18

Logical Data Warehouse Design

Outline

OLAP Technologies

Relational Data Warehouse Design

Relational Implementation of Conceptual Mode

The Time Dimension

The Time Dimension

Data warehouse: a historical database

Time dimension present in almost all data warehouses.

In a star or snowflake schema, time is included both as foreign key(s) in a fact table and as a time dimension containing the aggregation levels

OLTP databases: temporal information is usually derived from attributes of a **DATE** data type

- Example: A weekend is computed on-the-fly using appropriate functions

In a data warehouse time information is stored as explicit attributes in the time dimension

- Easy to compute: Total sales during weekends

```
SELECT SUM(SalesAmount)
FROM Time T, Sales S
WHERE T.TimeKey = S.TimeKey AND T.WeekendFlag
```

SELECT statement filtering time dimension

The Time Dimension

The granularity of the time dimension varies depending on their use

Time dimension with a granularity month spanning 5 years will have $5 * 12 = 60$ tuples

References

A. VAISMAN, E. ZIMANYI, Data Warehouse Systems: Design and Implementation, Chapter 5 Logical Data Warehouse Design, Springer Verlag, 2014

ISIT312 Big Data Management

SQL for Data Warehousing

Dr Janusz R. Getta

School of Computing and Information Technology -
University of Wollongong

SQL for Data Warehousing

Outline

SQL/OLAP Operations

Window partitioning

Window ordering

Window framing

SQL/OLAP Operations

Consider the **SALES** fact table

To compute all possible aggregations along the dimensions **Product** and **Customer** we must scan the whole relational table **SALES** several times

It can be implemented in SQL using **NULL** and **UNION** in the following way:

```
SELECT ProductKey, CustomerKey, SalesAmount  
FROM Sales  
    UNION  
SELECT ProductKey, NULL, SUM(SalesAmount)  
FROM Sales  
GROUP BY ProductKey  
    UNION  
SELECT NULL, CustomerKey, SUM(SalesAmount)  
FROM Sales  
GROUP BY CustomerKey  
    UNION  
SELECT NULL, NULL, SUM(SalesAmount)  
FROM Sales;
```

Finding aggregations along many dimensions

SQL/OLAP Operations

A data cube created through **UNION** of individual **SELECT** statements each one creating one combination of dimensions looks in the following way

Data cube

ProductKey	CustomerKey	SalesAmount
p1	c1	100
p2	c1	70
p3	c1	30
NULL	c1	200
p1	c2	105
p2	c2	60
p3	c2	40
NULL	c2	205
p1	c3	100
p2	c3	40
p3	c3	50
NULL	c3	190
p1	NULL	305
p2	NULL	170
p3	NULL	120
NULL	NULL	595

[TOP](#)

Created by Janusz R. Getta, ISIT312 Big Data Management, SIM, Session 2, 2021

4/18

SQL/OLAP Operations

Computing a cube with n dimensions requires $(2*2*2*... *2)(n \text{ times})$

SELECT statements with **GROUP BY** clause

SQL/OLAP extends the **GROUP BY** clause with the **ROLLUP** and **CUBE** operators

ROLLUP computes group subtotals in the order given by a list of attributes

CUBE computes all totals of such a list

Shorthands for a more powerful operator, **GROUPING SETS**

Equivalent queries

```
SELECT ProductKey, CustomerKey, SUM(SalesAmount)
FROM Sales
GROUP BY ROLLUP(ProductKey, CustomerKey);
```

Sample application of ROLLUP operation

```
SELECT ProductKey, CustomerKey, SUM(SalesAmount)
FROM Sales
GROUP BY GROUPING SETS((ProductKey,CustomerKey),(ProductKey),());
```

Sample application of GROUPING SET operation

[TOP](#)

Created by Janusz R. Getta, ISIT312 Big Data Management, SIM, Session 2, 2021

5/18

SQL/OLAP Operations

Equivalent queries

Sample application of CUBE operation

```
SELECT ProductKey, CustomerKey, SUM(SalesAmount)
FROM Sales
GROUP BY CUBE(ProductKey, CustomerKey);
```

Sample application of GROUPING SETS operation

```
SELECT ProductKey, CustomerKey, SUM(SalesAmount)
FROM Sales
GROUP BY GROUPING SETS((ProductKey, CustomerKey), (ProductKey), (CustomerKey), ());
```

SQL/OLAP Operations

GROUP BY ROLLUP

ProductKey	CustomerKey	SalesAmount
p1	c1	100
p1	c2	105
p1	c3	100
p1	NULL	305
p2	c1	70
p2	c2	60
p2	c3	40
p2	NULL	170
p3	c1	30
p3	c2	40
p3	c3	50
p3	NULL	120
NULL	NULL	595

GROUP BY CUBE

ProductKey	CustomerKey	SalesAmount
p1	c1	100
p2	c1	70
p3	c1	30
NULL	c1	200
p1	c2	105
p2	c2	60
p3	c2	40
NULL	c2	205
p1	c3	100
p2	c3	40
p3	c3	50
NULL	c3	190
NULL	NULL	595
p1	NULL	305
p2	NULL	170
p3	NULL	120

SQL for Data Warehousing

Outline

SQL/OLAP Operations

Window partitioning

Window ordering

Window framing

Window partitioning

Allows to compare detailed data with aggregate values

For example, find a relevance of each customer with respect to the sales of the product

Sample window partitioning

```
SELECT ProductKey, CustomerKey, SalesAmount,  
       MAX(SalesAmount) OVER (PARTITION BY ProductKey) AS MaxAmount  
FROM SALES;
```

First three columns are obtained from the **Sales** table

The fourth column is created in the following way

- Create a window called **partition** that contains all tuples of the same product
- **SalesAmount** is aggregated over this window using **MAX** function

Window partitioning

Sample window partitioning

```
SELECT ProductKey, CustomerKey, SalesAmount,  
       MAX(SalesAmount) OVER (PARTITION BY ProductKey) AS MaxAmount  
FROM SALES;
```

ProductKey	CustomerKey	SalesAmount	MaxAmount
p1	c1	100	105
p1	c2	105	105
p1	c3	100	105
p2	c1	70	70
p2	c2	60	70
p2	c3	40	70
p3	c1	30	50
p3	c2	40	50
p3	c3	50	50

SQL for Data Warehousing

Outline

SQL/OLAP Operations

Window partitioning

Window ordering

Window framing

Window ordering

ORDER BY clause allows the rows within a partition to be ordered

It is useful to compute rankings, with a function **RANK()**

For example, how does each product rank in the sales of each customer

Sample window ordering

```
SELECT ProductKey, CustomerKey, SalesAmount,  
       RANK() OVER (PARTITION BY CustomerKey ORDER BY SalesAmount DESC) AS RowNo  
FROM Sales;
```

Product Key	Customer Key	Sales Amount	RowNo
p1	c1	100	1
p2	c1	70	2
p3	c1	30	3
p1	c2	105	1
p2	c2	60	2
p3	c2	40	3
p1	c3	100	1
p3	c3	50	2
p2	c3	40	3

[TOP](#)

Created by Janusz R. Getta, ST312 Big Data Management, SIM, Session 2, 2021

12/18

SQL for Data Warehousing

Outline

SQL/OLAP Operations

Window partitioning

Window ordering

Window framing

Window framing

It is possible to define a size of a partition

It can be used to compute statistical functions over time series, like moving average

For example, three-month moving average of sales by product

Sample window framing

```
SELECT ProductKey, Year, Month, SalesAmount,  
       AVG(SalesAmount) OVER (PARTITION BY ProductKey  
                               ORDER BY Year, Month  
                               ROWS 2 PRECEDING) AS MovAvg  
FROM SALES;
```

Processing of a query opens a window with the rows pertaining to the current product

Then, it orders the window by year and month and computes the average over the current row and the previous two ones if they exist

Window framing

Sample window framing

```
SELECT ProductKey, Year, Month, SalesAmount,  
       AVG(SalesAmount) OVER (PARTITION BY ProductKey  
                               ORDER BY Year, Month  
                               ROWS 2 PRECEDING) AS MovAvg  
  FROM SALES;
```

Product Key	Year	Month	Sales Amount	MovAvg
p1	2011	10	100	100
p1	2011	11	105	102.5
p1	2011	12	100	101.67
p2	2011	12	60	60
p2	2012	1	40	50
p2	2012	2	70	56.67
p3	2012	1	30	30
p3	2012	2	50	40
p3	2012	3	40	40

Window framing

Another example, a year-to-date sum of sales by product

Sample window framing

```
SELECT ProductKey, Year, Month, SalesAmount,  
       SUM(SalesAmount) OVER (PARTITION BY ProductKey, Year  
                               ORDER BY Month  
                               ROWS UNBOUNDED PRECEDING) AS YTD  
  FROM SALES;
```

Processing of a query, opens a window with the tuples of the current product and year ordered by month

AVG() is applied to all rows before the current row (**ROWS UNBOUNDED PRECEDING**)

Window framing

Sample window framing

```
SELECT ProductKey, Year, Month, SalesAmount,  
       SUM(SalesAmount) OVER (PARTITION BY ProductKey, Year  
                               ORDER BY Month  
                               ROWS UNBOUNDED PRECEDING) AS YTD  
  FROM SALES;
```

Product Key	Year	Month	Sales Amount	YTD
p1	2011	10	100	100
p1	2011	11	105	205
p1	2011	12	100	305
p2	2011	12	60	60
p2	2012	1	40	40
p2	2012	2	70	110
p3	2012	1	30	30
p3	2012	2	50	80
p3	2012	3	40	120

References

A. VAISMAN, E. ZIMANYI, Data Warehouse Systems: Design and Implementation, Chapter 5 Logical Data Warehouse Design, Springer Verlag, 2014

ISIT312 Big Data Management

Hive Programming

Dr Janusz R. Getta

School of Computing and Information Technology -
University of Wollongong

Hive Programming

Outline

Data Selection and Scope

Data Manipulation

Data Aggregation and Sampling

Data Selection and Scope

To query data **Hive** provides **SELECT** statement

Typically **SELECT** statement projects the rows satisfying the query conditions specified in the **WHERE** clause and returns the result set

SELECT statement is usually used with **FROM**, **DISTINCT**, **WHERE**, and **LIMIT** keywords

```
SELECT C_NAME, C_PHONE  
FROM customer  
WHERE C_ACCTBAL > 0  
LIMIT 2;
```

SELECT statement with LIMIT clause

Data Selection and Scope

Multiple **SELECT** statements can be combined into complex queries using **nested queries** or **subqueries**

Subqueries can use **Common Table Expressions (CTE)** in the format of **WITH clause**

When using **subqueries**, an **alias** should be given for the subquery

```
WITH clause  
WITH cord AS ( SELECT *  
    FROM customer JOIN orders  
        ON c_custkey = o_custkey)  
SELECT c_name, c_phone, o_orderkey, o_orderstatus  
FROM cord;
```

Data Selection and Scope

Multiple **SELECT** statements can be combined into complex queries using **nested queries** or **subqueries**

Nested queries can use **SELECT** statement wherever a table is expected or a scalar value is expected

Nested query

```
SELECT c_name, c_phone, o_orderkey, o_orderstatus
FROM ( SELECT *
        FROM customer JOIN orders
          ON c_custkey = o_custkey) cord
```

Data Selection and Scope

When **inner join** is performed between multiple tables the **MapReduce** jobs are created to process data in **HDFS**

It is recommended to put the big table right at the end for better because the last table in the sequence is streamed through the reducers where the others are buffered in the reducer by default

```
SELECT /*+ STREAMTABLE(lineitem) */ c_name, o_orderkey, l_linenumber
FROM customer JOIN orders
    ON c_custkey = o_custkey
    JOIN lineitem
    ON l_orderkey = o_orderkey;
```

Inner join

Data Selection and Scope

Outer join (left, right, and full) and cross join preserve their HQL semantics

Map join means that join is computed only by map job without reduce job

In map join all data are read from a small table to memory and broadcasted to all maps

During map phase each row in from a big table is compared with the rows in small tables against join conditions

Join performance is improved because there is no reduce phase

Data Selection and Scope

Map join

```
SELECT /*+ MAPJOIN(orders) */ c_name, c_phone, o_orderkey, o_orderstatus  
FROM customer JOIN orders  
    ON c_custkey = o_custkey;
```

Hive automatically converts the JOIN to MAPJOIN at runtime when
`hive.auto.convert.join` setting is set to true

Bucket map join is a special type of MAPJOIN that uses bucket columns
in join condition.

Then instead of fetching the whole table bucket map join only fetches
the required bucket data.

A variable `hive.optimize.bucketmapjoin` must be set to true to
enable bucket map join

Data Selection and Scope

Hive supports **LEFT SEMI JOIN**

```
SELECT c_name, c_phone  
FROM customer LEFT SEMI JOIN orders  
    ON c_custkey = o_custkey;
```

Left semi join

In **LEFT SEMI JOIN** the right-hand side table should only be referenced in the join condition and not in **WHERE** or **SELECT** clauses

Data Selection and Scope

Hive supports **UNION ALL** it does not support **INTERSECT** and **MINUS** operations

```
SELECT p_name  
FROM PART  
UNION ALL  
SELECT c_name  
FROM CUSTOMER;
```

UNION ALL operation

INTERSECT operation can be implemented as **JOIN** operation

MINUS operation can be implemented as **LEFT OUTER JOIN** operation with **IS NULL** condition in **WHERE** clause

Hive Programming

Outline

Data Selection and Scope

Data Manipulation

Data Aggregation and Sampling

Data Manipulation

LOAD statement can be used to load data to **Hive** tables from local file system or from **HDFS**

Load data to **Hive** table from a local file

Loading data from a local file

```
LOAD DATA LOCAL INPATH '/local/home/janusz/HIVE-EXAMPLES/TPCHR/part.txt'  
OVERWRITE INTO TABLE part;
```

Load data to **Hive** partitioned table from a local file

Loading data into partitioned table from a local file

```
LOAD DATA LOCAL INPATH '/local/home/janusz/HIVE-EXAMPLES/TPCHR/part.txt'  
OVERWRITE INTO TABLE part PARTITION  
(P_BRAND='GoldenBolts');
```

LOCAL keyword determines a location of the input files

Data Manipulation

Load **HDFS** data to the **Hive** table using the default system path

```
LOAD DATA INPATH '/user/janusz/part.txt'  
OVERWRITE INTO TABLE part;
```

Loading data from HDFS

Load **HDFS** data to the **Hive** table using using full **URI**

```
LOAD DATA INPATH 'hdfs://10.9.28.14:8020/user/janusz/part.txt'  
OVERWRITE INTO TABLE part;
```

Loading data from HDFS

If **LOCAL** keyword is not specified, the files are loaded from the full **URI** specified after **INPATH** or the value from the **fs.default**

OVERWRITE keyword decides whether to append or replace the existing data in the target table/partition

Data Manipulation

EXPORT and **IMPORT** statements are available to support the import and export of data in HDFS for data migration or backup/restore purposes

EXPORT statement exports both data and metadata from a table or partition

Exporting table to HDFS

```
EXPORT TABLE part TO '/user/tpchr/part'
```

Metadata is exported to a file called **_metadata**

Contents of HDFS

```
-rwxr-xr-x 3 janusz supergroup 2739 2017-07-09  
14:37 /user/tpchr/part/_metadata  
drwxr-xr-x - janusz supergroup 0 2017-07-09  
14:37 /user/tpchr/part/p_brand=GoldenBolts
```

After **EXPORT** the exported files can be copied to other **Hive** instances or to other **HDFS** clusters

Data Manipulation

IMPORT statement imports files exported from other **HIVE** instances into an internal table

```
IMPORT table new_part FROM '/user/tpchr/part';
```

HQL

An imported table is located in a default **HIVE** location in **HDFS**

```
drwxrwxr-x - janusz supergroup 0 2017-07-09  
14:56 /user/hive/warehouse/new_part
```

Importing data from HDFS

IMPORT EXTERNAL statement imports a file exported from other **HIVE** instances into an external table

```
IMPORT EXTERNAL table new_extpart FROM '/user/tpchr/  
part';
```

Importing external table from HDFS

An imported table is located in a default **HIVE** location in **HDFS**

```
drwxrwxr-x - janusz supergroup 0 2017-07-09  
15:04 /user/hive/warehouse/new_extpart
```

Contents of HDFS

Data Manipulation

ORDER BY sorts the results of **SELECT** statement

An order is maintained across all of the output from every reducer and global sort is performed using only one reducer

```
SELECT p_partkey, p_name  
FROM part  
ORDER BY p_name ASC;
```

ORDER BY clause

SORT BY does the same job **ORDER BY** and indicates which columns to sort when ordering the reducer input records

SORT BY completes sorting before sending data to the reducer

SORT BY statement does not perform a global sort and only makes sure data is locally sorted in each reducer

```
SET mapred.reduce.tasks = 2;  
SELECT p_partkey, p_name  
FROM part  
SORT BY p_name ASC;
```

SORT BY clause

Data Manipulation

When **DISTRIBUTE BY** clause is applied rows with matching column values are partitioned by the same reducer

DISTRIBUTE BY clause

```
SELECT p_partkey, p_name FROM part
DISTRIBUTE BY p_partkey
SORT BY p_name;
```

DISTRIBUTE BY clause is similar to **GROUP BY** in relational systems in terms of deciding which reducer is used to distribute the mapper

When using with **SORT BY**, **DISTRIBUTE BY** must be specified before the **SORT BY** statement

Data Manipulation

CLUSTER BY clause is a shorthand operator to perform **DISTRIBUTE BY** and **SORT BY** operations on the same group of columns.

CLUSTER BY clause

```
SELECT p_partkey, p_name  
FROM part  
CLUSTER BY p_name;
```

ORDER BY performs a global sort, while **CLUSTER BY** sorts in each distributed group

To fully utilize all the available reducers we can do **CLUSTER BY** first and then **ORDER BY**

CLUSTER BY clause

```
SELECT p_partkey, p_name  
FROM part  
ORDER BY p_name  
CLUSTER BY p_name;
```

[TOP](#)

Created by Janusz R. Getta, ISIT312 Big Data Management, SIM, Session 2, 2021

18/28

Hive Programming

Outline

Data Selection and Scope

Data Manipulation

Data Aggregation and Sampling

Data Aggregation and Sampling

Hive supports several aggregation functions, analytic functions working with **GROUP BY** and **PARTITION BY**, and windowing clauses

Hive supports advanced aggregation by using **GROUPING SETS**, **ROLLUP**, **CUBE**, analytic functions, and windowing

Basic aggregation uses **GROUP BY** clause and aggregation functions

```
SELECT p_type, count(*)  
FROM part  
GROUP BY p_type;
```

GROUP BY clause

To aggregate into sets a function **collect_set** can be used

```
SELECT p_type, collect_set(p_name), count(*)  
FROM part  
GROUP BY p_type;
```

GROUP BY clause with collect_set function

Data Aggregation and Sampling

GROUPING SETS clause implements advanced multiple **GROUP BY** operations against the same set of data

GROUPING SETS clause

```
SELECT p_type, p_name, count(*)
FROM part
GROUP BY p_type, p_name
GROUPING SETS ( (p_type), (p_name) );
```

ROLLUP clause allows to calculate multiple levels of aggregations across a specified group of dimensions

ROLLUP clause

```
SELECT p_type, p_name, count(*)
FROM part
GROUP BY p_type, p_name WITH ROLLUP;
```

CUBE clause allows to create aggregations over all possible subsets of attributes in a given set

CUBE clause

```
SELECT p_type, p_name, count(*)
FROM part
GROUP BY p_type, p_name WITH CUBE;
```

TOP

Created by Janusz R. Getta, ISIT312 Big Data Management, SIM, Session 2, 2021

21/28

Data Aggregation and Sampling

GROUPING_ID function works as an extension to distinguish entire rows from each other

GROUPING_ID function

```
SELECT GROUPING_ID, p_type, p_name, count(*)  
FROM part  
GROUP BY p_type, p_name WITH CUBE  
ORDER BY grouping_id;
```

HAVING can be used for the conditional filtering of GROUP BY results

GROUPING_ID function

```
SELECT GROUPING_ID, p_type, p_name, count(*)  
FROM part  
GROUP BY p_type, p_name WITH CUBE  
HAVING count(*) > 1  
ORDER BY grouping_id;
```

Data Aggregation and Sampling

Analytic functions scan multiple input rows to compute each output value

Analytic functions are usually used with `OVER`, `PARTITION BY`, `ORDER BY`, and the windowing specification

Analytic functions operate on windows where the input rows are ordered and grouped using flexible conditions expressed through an `OVER PARTITION` clause

Syntax is the following

Syntax of analytic functions

```
function (arg1,..., argn)
OVER ([PARTITION BY <...>]
[ORDER BY <....>] [])
```

For standard aggregation function `(arg1,..., argn)` can be either `COUNT()`, `SUM()`, `MIN()`, `MAX()`, or `AVG()`

Data Aggregation and Sampling

Typical aggregations implemented as analytic functions in the following way

```
SELECT p_name,  
       COUNT(*) OVER (PARTITION BY p_name)  
  FROM PART;
```

PARTITION BY clause

Other analytic functions are used as follows

```
SELECT l_orderkey, l_partkey, l_quantity,  
       RANK() OVER (ORDER BY l_quantity),  
       DENSE_RANK() OVER (ORDER BY l_quantity)  
  FROM lineitem;
```

ORDER BY clause

```
SELECT l_orderkey, l_partkey, l_quantity,  
       RANK() OVER (PARTITION BY l_orderkey ORDER BY l_quantity),  
       DENSE_RANK() OVER (PARTITION BY l_orderkey ORDER BY l_quantity)  
  FROM lineitem;
```

PARTITION BY clause

Data Aggregation and Sampling

More analytic functions ...

```
SELECT l_orderkey, l_partkey, l_quantity,  
       FIRST_VALUE(l_quantity) OVER (PARTITION BY l_orderkey ORDER BY l_quantity),  
       LAST_VALUE(l_quantity) OVER (PARTITION BY l_orderkey ORDER BY l_quantity)  
FROM lineitem;;
```

PARTITION BY clause

```
SELECT l_orderkey, l_partkey, l_quantity,  
       MAX(l_quantity) OVER (PARTITION BY l_orderkey ORDER BY l_partkey  
                             ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)  
FROM lineitem;
```

PARTITION BY clause

```
SELECT l_orderkey, l_partkey, l_quantity,  
       MAX(l_quantity) OVER (PARTITION BY l_orderkey ORDER BY l_partkey  
                             ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)  
FROM lineitem;
```

PARTITION BY clause

Data Aggregation and Sampling

When data volume is extra large we can use a subset of data to speed up data analysis.

Random sampling uses the `RAND()` function and `LIMIT` clause to get the samples of data

```
SELECT *  
FROM lineitem DISTRIBUTE BY RAND() SORT BY RAND() LIMIT 5;
```

DISTRIBUTE BY clause

`DISTRIBUTE` and `SORT` clauses are used here to make sure the data is also randomly and efficiently distributed among mappers and reducers

Bucket table sampling is a special sampling optimized for bucket tables

```
SELECT *  
FROM customer TABLESAMPLE(BUCKET 3 OUT OF 8 ON rand());
```

Bucket sampling

Data Aggregation and Sampling

Block sampling allows to randomly pick up **N** rows of data, percentage (**n** percentage) of data size, or **N** byte size of data

```
SELECT *  
FROM lineitem TABLESAMPLE(4 ROWS);
```

Block sampling

```
SELECT *  
FROM lineitem TABLESAMPLE(50 PERCENT);
```

Block sampling

```
SELECT *  
FROM lineitem TABLESAMPLE(20B);
```

Block sampling

References

Gross C., GuptaA., Shaw S., Vermeulen A. F., Kjerrumgaard D., Practical Hive: A guide to Hadoop's Data Warehouse System, Apress 2016, Chapter 4 (Available through UOW library)

Lee D., Instant Apache Hive essentials how-to: leverage your knowledge of SQL to easily write distributed data processing applications on Hadoop using Apache Hive, Packt Publishing Ltd. 2013 (Available through UOW library)

Apache Hive TM, <https://hive.apache.org/>

ISIT312 Big Data Management

Physical Data Warehouse Design

Dr Janusz R. Getta

School of Computing and Information Technology -
University of Wollongong

Physical Data Warehouse Design

Outline

Techniques for Physical Data Warehouse Design

Materialized View

Indexes for Data Warehouses

Evaluation of Star Queries

Data Warehouse Partitioning

Techniques for Physical Data Warehouse Design

Materialized Views

- A view physically stored in the DB
- Typical problems: view update, view selection

Indexing

- Used in Data Warehouse together with materialized views
- Specific for Data Warehouse: bitmap and join indexes

Partitioning

- Divides the contents of a relation in several files
- Horizontal and vertical partitioning

Physical Data Warehouse Design

Outline

Techniques for Physical Data Warehouse Design

Materialized View

Indexes for Data Warehouses

Evaluation of Star Queries

Data Warehouse Partitioning

Materialized Views

Materialized view is a relational table that contains the rows that would be returned by the view definition - usually **SELECT** statement of SQL

If we consider relational views as **stored queries** then **materialized views** can be considered as **stored results**

Materialized views are created and used to reduce an amount of time needed to compute **SELECT** statements, for example join materialized views eliminate the needs to join the relational table

There are two ways how materialized view can be used:

- brute force method
- transparent query rewrite

In **brute force method** SQL is written to explicitly access the view

Transparent query rewrite method is applied when a query optimizer detects that a query can be computed against a materialized view instead of the source relational tables

Materialized Views

View maintenance means that when base relational tables are updated, view must be updated too

Incremental view maintenance means that updated view is computed from the individual modifications to the relational tables and not from the entire relational tables

Creating **materialized view**

```
CREATE MATERIALIZED VIEW MV_ORDERS
REFRESH ON COMMIT
ENABLE QUERY REWRITE
AS( SELECT O_ORDERKEY, O_CUSTKEY, O_TOTALPRICE, O_ORDERDATE
    FROM ORDERS
    WHERE O_ORDERDATE > TO_DATE('31-DEC-1986', 'DD-MON-YYYY') );
```

Creating materialized view

Direct access to **materialized view**

```
SELECT *
FROM MV_ORDERS
WHERE O_ORDERDATE = TO_DATE('01-JAN-1992', 'DD-MON-YYYY')
```

Direct access to materialized view

Materialized Views

Access to materialized view through query rewriting

Indirect access to materialized view through query rewriting

```
SELECT O_ORDERKEY, O_CUSTKEY, O_TOTALPRICE, O_ORDERDATE  
FROM ORDERS  
WHERE O_ORDERDATE > TO_DATE('31-DEC-1986', 'DD-MON-YYYY');
```

- The results from EXPLAIN PLAN statement

Query processing plan

PLAN_TABLE_OUTPUT

0 SELECT STATEMENT	108K 2539K 507 (1) 00:00:01
* 1 MAT_VIEW REWRITE ACCESS FULL MV_ORDERS 108K 2539K 507 (1) 00:00:01	

Predicate Information (identified by operation id):

```
1 - filter("MV_ORDERS"."O_ORDERDATE">>TO_DATE(' 1986-12-31 00:00:00',  
'syyyy-mm-dd hh24:mi:ss'))
```

Physical Data Warehouse Design

Outline

Techniques for Physical Data Warehouse Design

Materialized View

Indexes for Data Warehouses

Evaluation of Star Queries

Data Warehouse Partitioning

Indexes for Data Warehouses

An index provides a quick way to locate data of interest

Sample query

```
SELECT *  
FROM EMPLOYEE  
WHERE EmployeeKey = 007;
```

SELECT statement with equality condition in WHERE clause

With the help of an index over a column **EmployeeKey** (primary key in **EMPLOYEE** table), a single disk block access will suffice to answer the query

Without this index, we should perform a complete scan of table **EMPLOYEE**

Drawback: Almost every update on an indexed attribute also requires an index update

Too many indexes may degrade performance

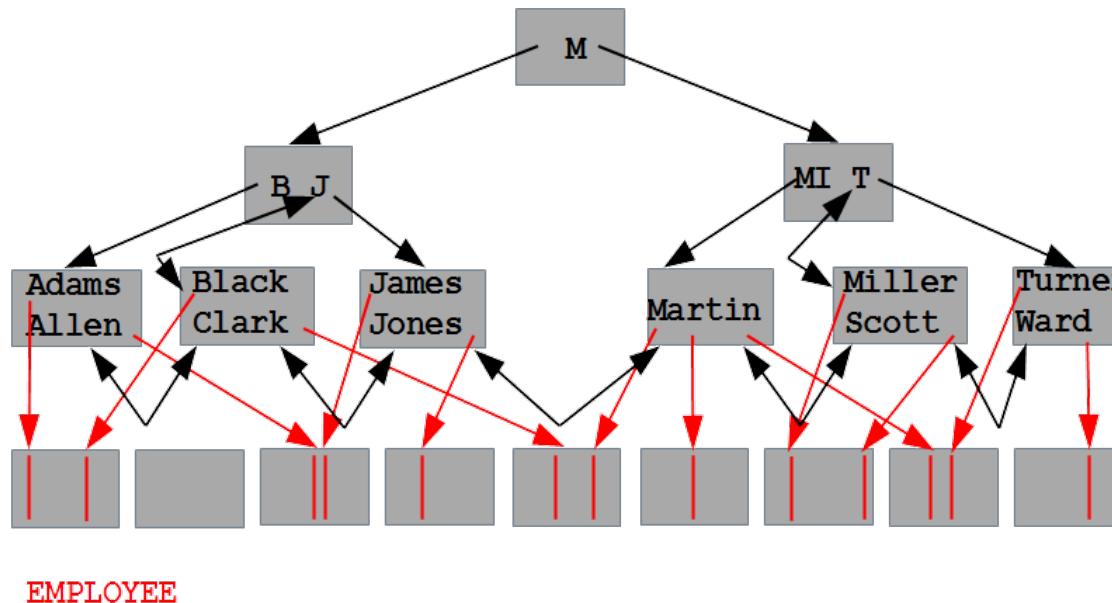
Most popular indexing techniques in relational databases include **B*-trees** and **bitmap indexes**

Created by Janusz R. Getta, ISIT312 Big Data Management, SIM, Session 2, 2021

TOP

9/23

B*-tree index implementation



B*-tree can be traversed either:

- vertically from **root** to **leaf level** of a tree
- horizontally either from **left corner** of **leaf level** to **right corner** of **leaf level** or the opposite
- vertically and later on horizontally either towards **left lower corner** or **right lower corner** of **leaf level**

[TOP](#)

Created by Janusz R. Getta, ISIT312 Big Data Management, SIM, Session 2, 2021

10/23

Bitmap Indexes

ProductKey	ProductName	QuantityPerUnit	UnitPrice	Discontinued	CategoryKey
p1	prod1	25	60	No	c1
p2	prod2	45	60	Yes	c1
p3	prod3	50	75	No	c2
p4	prod4	50	100	Yes	c2
p5	prod5	50	120	No	c3
p6	prod6	70	110	Yes	c4

Product dimension table

	25	45	50	70
p1	1	0	0	0
p2	0	1	0	0
p3	0	0	1	0
p4	0	0	1	0
p5	0	0	1	0
p6	0	0	0	1

Bitmap index for attribute QuantityPerUnit

	60	75	100	110	120
p1	1	0	0	0	0
p2	1	0	0	0	0
p3	0	1	0	0	0
p4	0	0	1	0	0
p5	0	0	0	0	1
p6	0	0	0	1	0

Bitmap index for attribute UnitPrice

Bitmap Indexes: Example

Products having between 45 and 55 pieces per unit, **and** with a unit price between 100 and 200

	45	50	OR1
p1	0	0	0
p2	1	0	1
p3	0	1	1
p4	0	1	1
p5	0	1	1
p6	0	0	0

OR for QuantityPerUnit

	100	110	120	OR2
p1	0	0	0	0
p2	0	0	0	0
p3	0	0	0	0
p4	1	0	0	1
p5	0	0	1	1
p6	0	1	0	1

OR for UnitPrice

	OR1	OR2	AND
p1	0	0	0
p2	1	0	0
p3	1	0	0
p4	1	1	1
p5	1	1	1
p6	0	1	0

AND operation

Indexes for Data Warehouses: Requirements

Symmetric partial match queries

- All dimensions of the cube should be symmetrically indexed, to be searched simultaneously

Indexing at multiple levels of aggregation

- Summary tables must be indexed in the same way as base nonaggregated tables

Efficient batch update

- The refreshing time of a data warehouse must be considered when designing the indexing schema

Sparse data

- Typically, only 20% of the cells in a data cube are nonempty
- The indexing schema must deal efficiently with sparse and nonsparse data

Physical Data Warehouse Design

Outline

Techniques for Physical Data Warehouse Design

Materialized View

Indexes for Data Warehouses

Evaluation of Star Queries

Data Warehouse Partitioning

Star Queries

Queries over star schemas are called **star queries**

Join the fact table with the dimension tables

A typical star query: total sales of discontinued products, by customer name and product name

```
SELECT ProductName, CustomerName, SUM(SalesAmount)
FROM Sales S, Customer C, Product P
WHERE S.CustomerKey = C.CustomerKey AND S.ProductKey = P.ProductKey AND
      P.Discontinued = 'Yes'
GROUP BY C.CustomerName, P.ProductName;
```

Star query

Three basic steps to evaluate the query:

- (1) Evaluation of the join conditions
- (2) Evaluation of the selection conditions over the dimensions
- (3) Aggregation of the tuples that passed the filter

Evaluation of Star Queries with Bitmap Indexes: Example

Product Key	Product Name	...	Discontinued	...
p1	prod1	...	No	...
p2	prod2	...	Yes	...
p3	prod3	...	No	...
p4	prod4	...	Yes	...
p5	prod5	...	No	...
p6	prod6	...	Yes	...

Product table

Yes	No
0	1
1	0
0	1
1	0
0	1
1	0

Bitmap for Discontinued

Customer Key	Customer Name	Address	Postal Code	...
c1	cust1	35 Main St.	7373	...
c2	cust2	Av. Roosevelt 50	1050	...
c3	cust3	Av. Louise 233	1080	...
c4	cust4	Rue Gabrielle	1180	...

Customer table

Product Key	Customer Key	Time Key	Sales Amount
p1	c1	t1	100
p1	c2	t1	100
p2	c2	t2	100
p2	c2	t3	100
p3	c3	t3	100
p4	c3	t4	100
p5	c4	t5	100

Sales fact table

c1	c2	c3	c4
1	0	0	0
0	1	0	0
0	1	0	0
0	1	0	0
0	0	1	0
0	0	1	0
0	0	0	1

Bitmap for Customer

p1	p2	p3	p4	p5	p6
1	0	0	0	0	0
1	0	0	0	0	0
0	1	0	0	0	0
0	1	0	0	0	0
0	0	1	0	0	0
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0

Bitmap for ProductKey

Yes	No
0	1
0	1
1	0
1	0
0	1
1	0
0	1

Bitmap join index for Discontinued

[TOP](#)

Created by Janusz R. Getta, ISYTB12 Big Data Management, SIM, Session 2, 2021

16/23

Evaluation of Star Queries using Bitmap Indexes

Evaluation of star query requires

- a B+ tree over **CustomerKey** and **ProductKey**
- Bitmap indexes on the foreign key columns in **Sales** and on **Discontinued** in **Product**

Example of query evaluation

- (1) Obtain the record numbers of the records that satisfy the condition **Discontinued = 'Yes'**
- Answer: Records with **ProductKey** values **p2**, **p4**, and **p6**
- (2) To access the bitmap vectors in **Sales** with these labels perform a join between **Product** and **Sales**
- (3) Vectors labeled **p2** and **p4** match, no fact record for **p6**
- (4) Obtain the values for the **CustomerKey** in these records (**c2** and **c3**)
- (5) Use B+-tree index on **ProductKey** and **CustomerKey** to find the names of products and customers
- (6) Answer: (**cust2, prod2, 200**) and (**cust3, prod4, 100**)

[TOP](#)

Created by Janusz R. Getta, ISIT312 Big Data Management, SIM, Session 2, 2021

17/23

Physical Data Warehouse Design

Outline

Techniques for Physical Data Warehouse Design

Materialized View

Indexes for Data Warehouses

Evaluation of Star Queries

Data Warehouse Partitioning

Data Warehouse Partitioning

Partitioning (or fragmentation) divides a table into smaller data sets (each one called a partition)

Applied to tables and indexes

Vendors provide several different partitioning methods

Vertical partitioning splits the attributes of a table into groups that can be independently stored

- E.g., most often used attributes are stored in one partition, less often used attributes in another one
- More records fit into main memory, reducing their processing time

Horizontal partitioning divides a table into smaller tables with same structure than the full table

- For example, if some queries require the most recent data, partition horizontally according to time

Queries over Partitioned Databases

Partition pruning is the typical way of improving query performance using partitioning

Example: A **Sales** fact table in a warehouse can be partitioned by month

A query requesting orders for a single month only needs to access the partition of such a month

Joins also enhanced by using partitioning:

- When the two tables are partitioned on the join attributes
- When the reference table is partitioned on its primary key
- Large join is broken down into smaller joins

Partitioning Strategies

Three partitioning strategies: Range partitioning, hash partitioning, and list partitioning

Range partitioning maps records to partitions based on ranges of values of the partitioning key

Time dimension is a natural candidate for range partitioning

Example: A table with a date column defined as the partitioning key

- January-2012 partition will contain rows with key values from January 1 to January 31, 2012

Hash partitioning uses a hashing algorithm over the partitioning key to map records to partitions

- Hashing algorithm distributes rows among partitions in a uniform fashion, yielding, ideally, partitions of the same size
- Typically used when partitions are distributed in several devices, and when data are not partitioned based on time

Partitioning Strategies

List partitioning specifies a list of values for the partitioning key

Some vendors (e.g. Oracle) support the notion of composite partitioning, combining the basic data distribution methods

Thus, a table can be range partitioned, and each partition can be subdivided using hash partitioning

References

A. VAISMAN, E. ZIMANYI, Data Warehouse Systems: Design and Implementation, Chapter 7 Physical Data Warehouse Design, Springer Verlag, 2014

ISIT312 Big Data Management Extraction, Transformation, and Loading

Dr Janusz R. Getta

School of Computing and Information Technology -
University of Wollongong

Extraction, Transformation, and Loading

Outline

Extraction, Transformation, and Loading

Conceptual ETL Design using BPMN

Conceptual Design of the Northwind ETL

Extraction, Transformation, and Loading (ETL)

Extract data from internal and external sources, transform data, and load data into a data warehouse (ETL)

No agreed way to specify ETL at a conceptual level

We study conceptual ETL design

Conceptual model based on the Business Process Modeling Notation (BPMN)

- Users already familiar with BPMN do not need to learn another language to design ETL
- BPMN provides a conceptual and implementation-independent specification of processes
- Processes expressed in BPMN can be translated into executable specifications(e.g., Microsoft's Integration Services)

Extraction, Transformation, and Loading

Outline

[Extraction, Transformation, and Loading](#)

[Conceptual ETL Design using BPMN](#)

[Conceptual Design of the Northwind ETL](#)

Conceptual ETL Design using BPMN

Basic assumption for using **BPMN** as conceptual model: **ETL** process is a type of business process

There is no standard model for defining **ETL** processes

Each tool provides its own model, too detailed to be conceptual

Using **BPMN** constructs we define the most common **ETL** tasks and define a **BPMN** notation for **ETL**

ETL process: A combination of **control** and **data processes**

- Control processes manage the coarse-grained groups of tasks
- Data processes detail how input data are transformed and output data are produced

Two kinds of tasks in **ETL** conceptual modeling

- **Control tasks** highlight the control procedures provided by **BPMN**. Represent a **workflow** (arrows represent the precedence between activities)
- **Data tasks** refer to the tasks that directly manipulate data during an ETL process. Represent a **data flow** (arrows represent data ‘flowing’ along them)

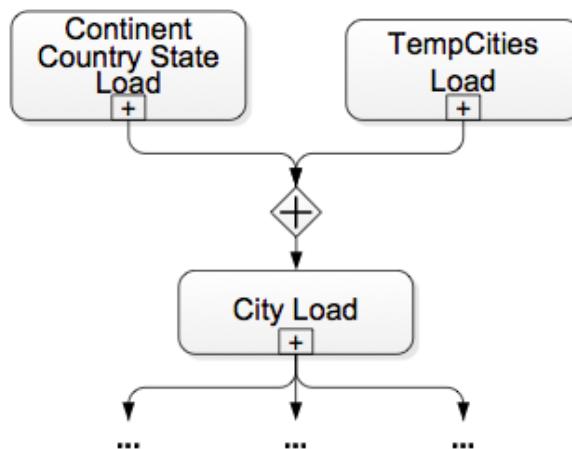
Control Tasks

Represent the workflow sequence or **orchestration** of the **ETL** process independently of the data flow

Control tasks are represented by means of **BPMN** constructs described

For example, gateways are used to control the sequence of activities in an **ETL** process

The most used types of gateways in an **ETL** context are exclusive and parallel



Data Tasks

Show how data are manipulated **within** an activity

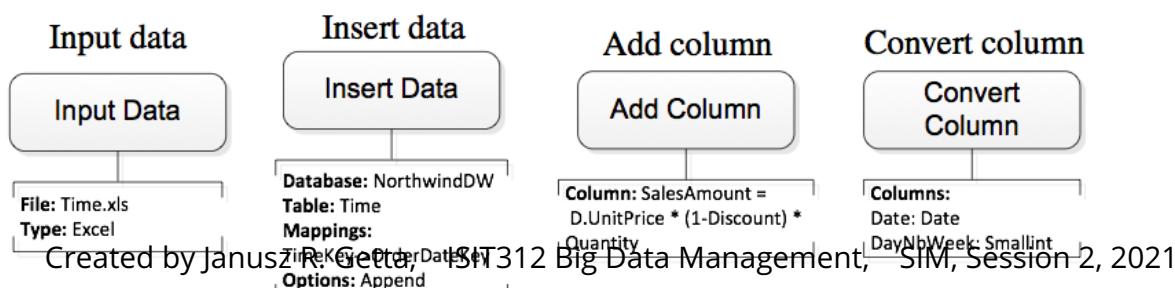
At lower abstraction level than control tasks

Represent activities typically carried out to manipulate data: input and output data, data conversion and transformation(for instance, change the data type of an attribute, add a column, remove duplicates, and so on)

We denote these tasks **unary data tasks** since they receive one input flow

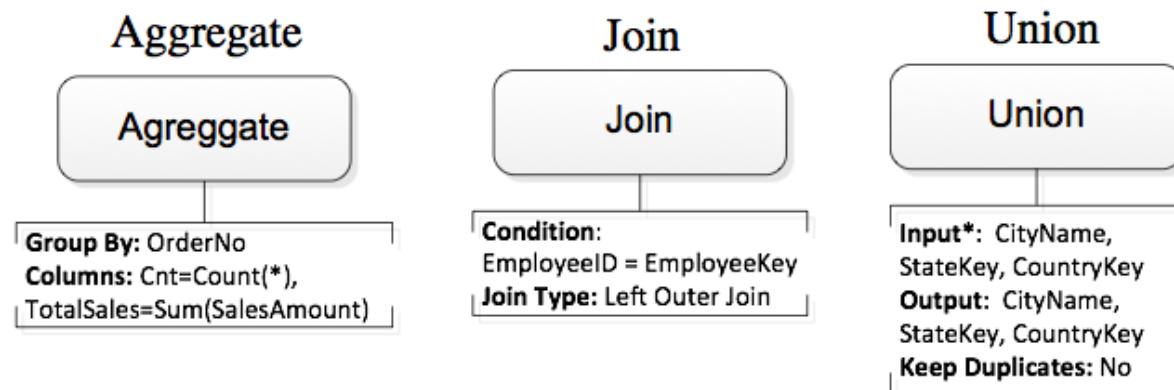
n-ary data tasks receive as input more than one flow (e.g., this is the case of union, join, difference,...)

Row operations are the transformations applied to the source or target data on a row-by-row basis, e.g., updating the value of a column



Rowset Data Tasks

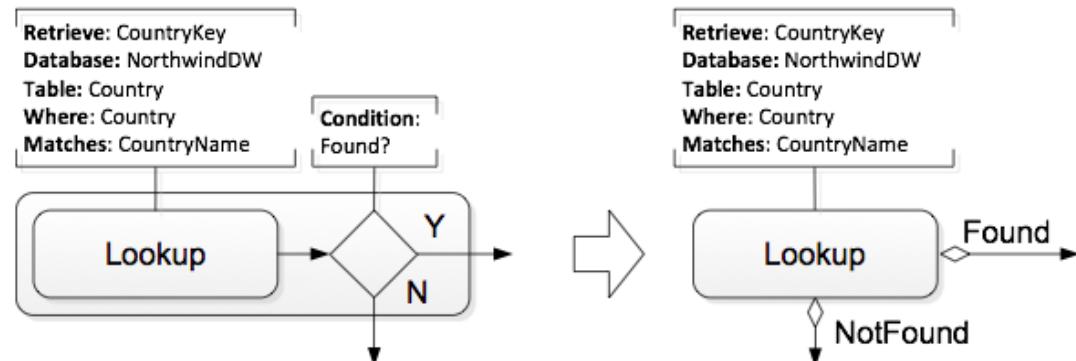
Rowset operations deal with a set of rows, e.g., aggregation is a rowset operation



Lookup Data Tasks

Lookup Data Tasks check if some value is present in a file. Immediately followed by an exclusive gateway with a branching condition. We use a shorthand replacing these two tasks by 2 conditional flows.

Shorthand notation for the lookup task



Extraction, Transformation, and Loading

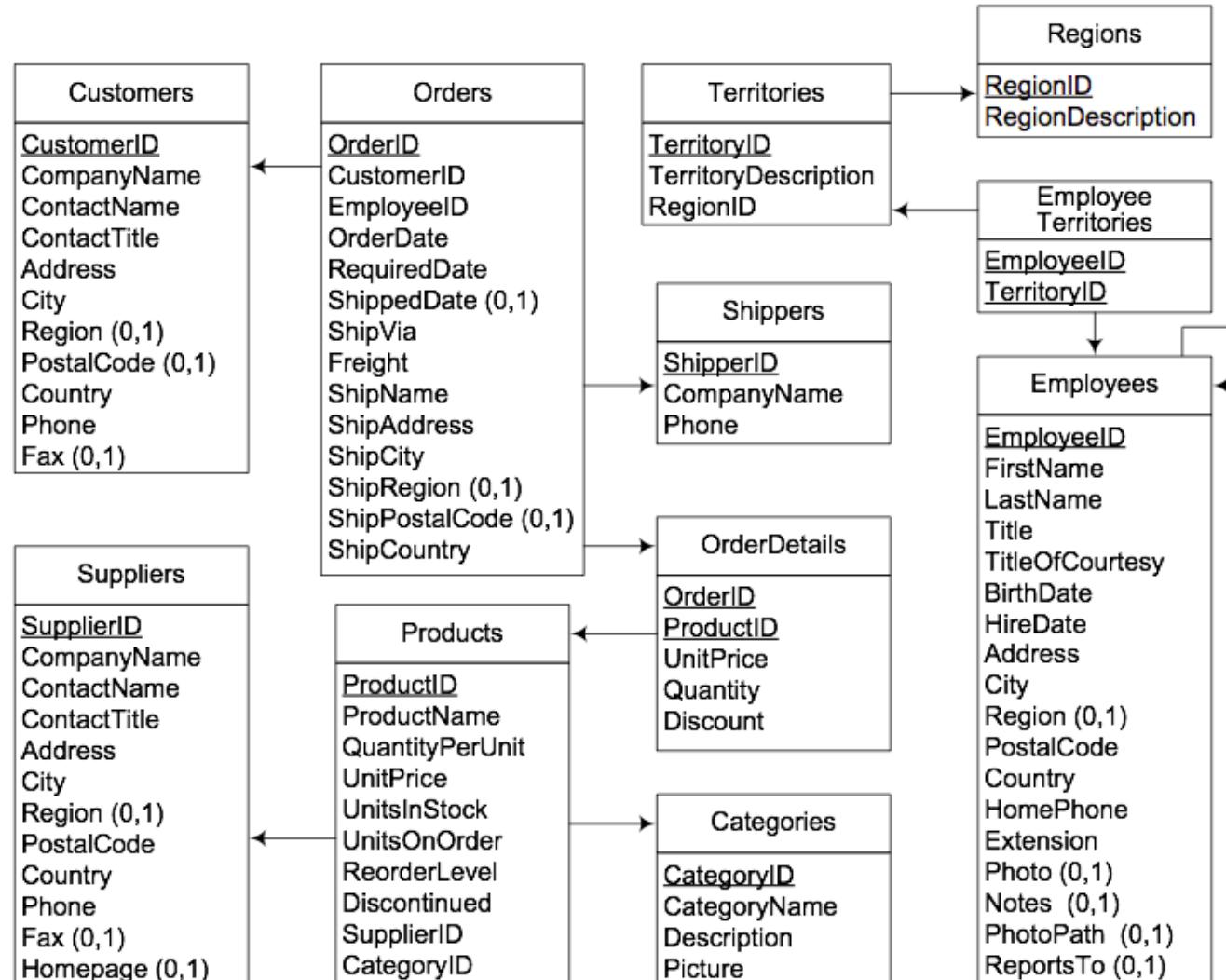
Outline

[Extraction, Transformation, and Loading](#)

[Conceptual ETL Design using BPMN](#)

[Conceptual Design of the Northwind ETL](#)

Schema of the Northwind Operational Database

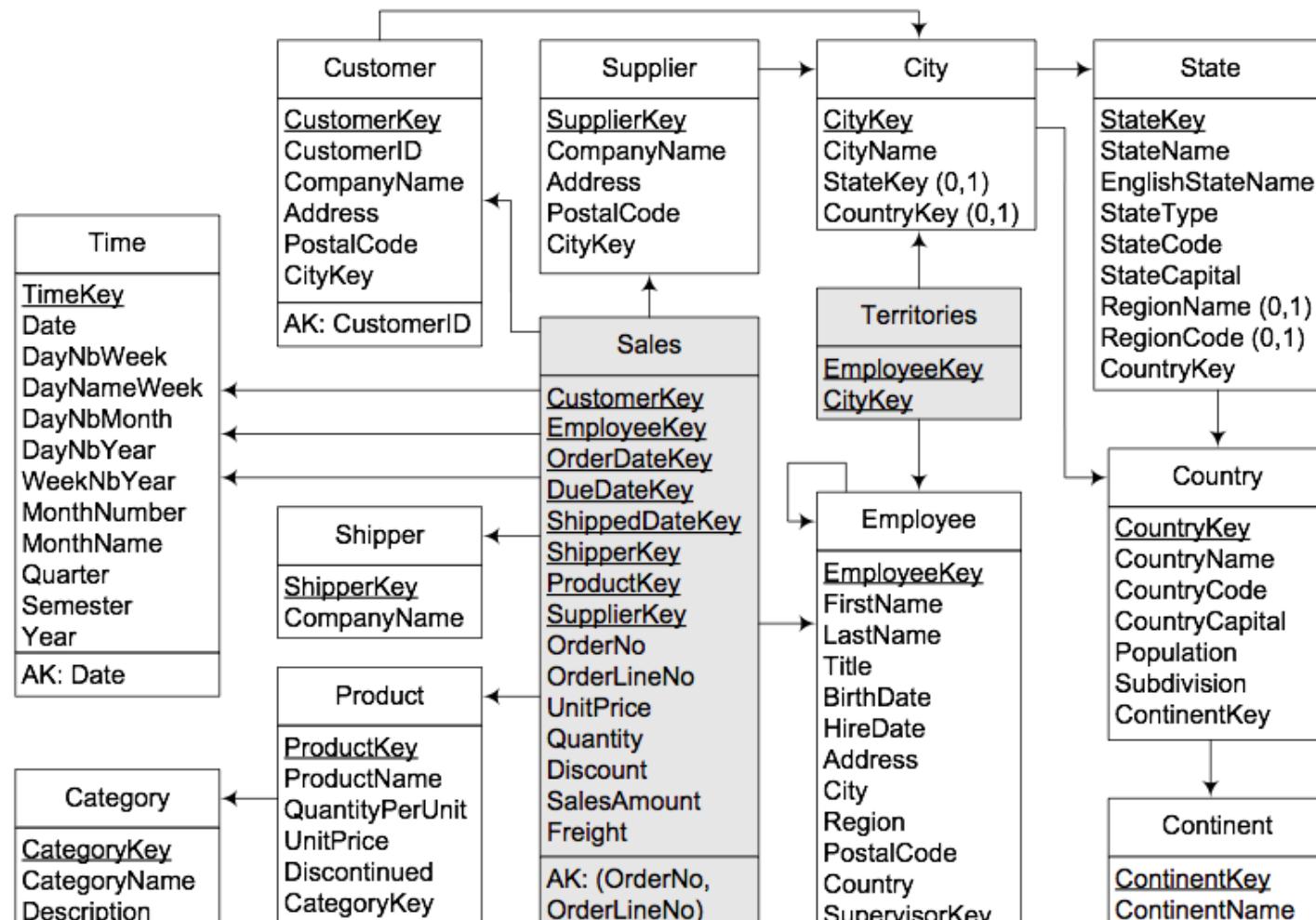
[TOP](#)

Created by Janusz R. Getta, ISIT312 Big Data Management,

SIM, Session 2, 2021

11/18

Schema of the Northwind Data Warehouse



Conceptual Design of the Northwind ETL: Data Sources

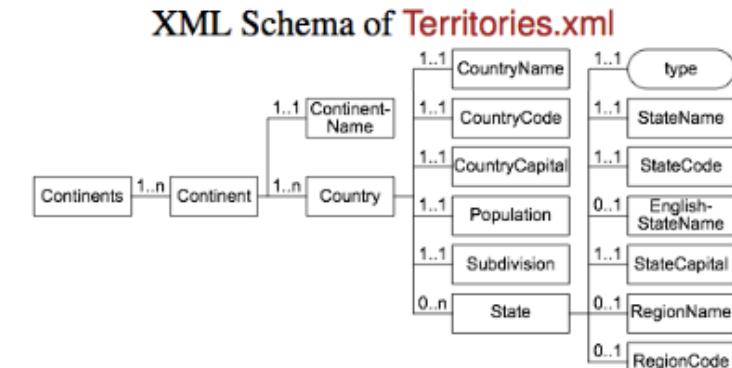
File **Time.xls** contains data for loading the **Time** dimension, spanning the dates in table **Orders** of the operational database

Dimensions **Customer** and **Supplier** share the geographic hierarchy starting at the **City** level

Data for the hierarchy **State** → **Country** → **Continent** loaded from **Territories.xml**

Start of the file Territories.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Continents>
  <Continent>
    <ContinentName>Europe</ContinentName>
    <Country>
      <CountryName>Austria</CountryName>
      <CountryCode>AT</CountryCode>
      <CountryCapital>Vienna</CountryCapital>
      <Population>8316487</Population>
      <Subdivision>Austria is divided into nine Bundeslnder,  
or simply Lnder (states; sing. Land).</Subdivision>
      <State type="state">
        <StateName>Burgenland</StateName>
        <StateCode>BU</StateCode>
        <StateCapital>Eisenstadt</StateCapital>
      </State>
      <State type="state">
        <StateName>Krnten</StateName>
        <StateCode>KA</StateCode>
        <EnglishStateName>Carinthia</EnglishStateName>
        <StateCapital>Klagenfurt</StateCapital>
      </State>
    ...>
```



Conceptual Design of the Northwind ETL: Data Sources

File called **Cities.txt** identifies to which state or province a city belongs

Contains three fields separated by tabs and begins as shown below

For cities located in countries that do not have states (e.g. Singapore), second field is set to null

The file is also used to identify to which state corresponds the city in the attribute TerritoryDescription of table **Territories**

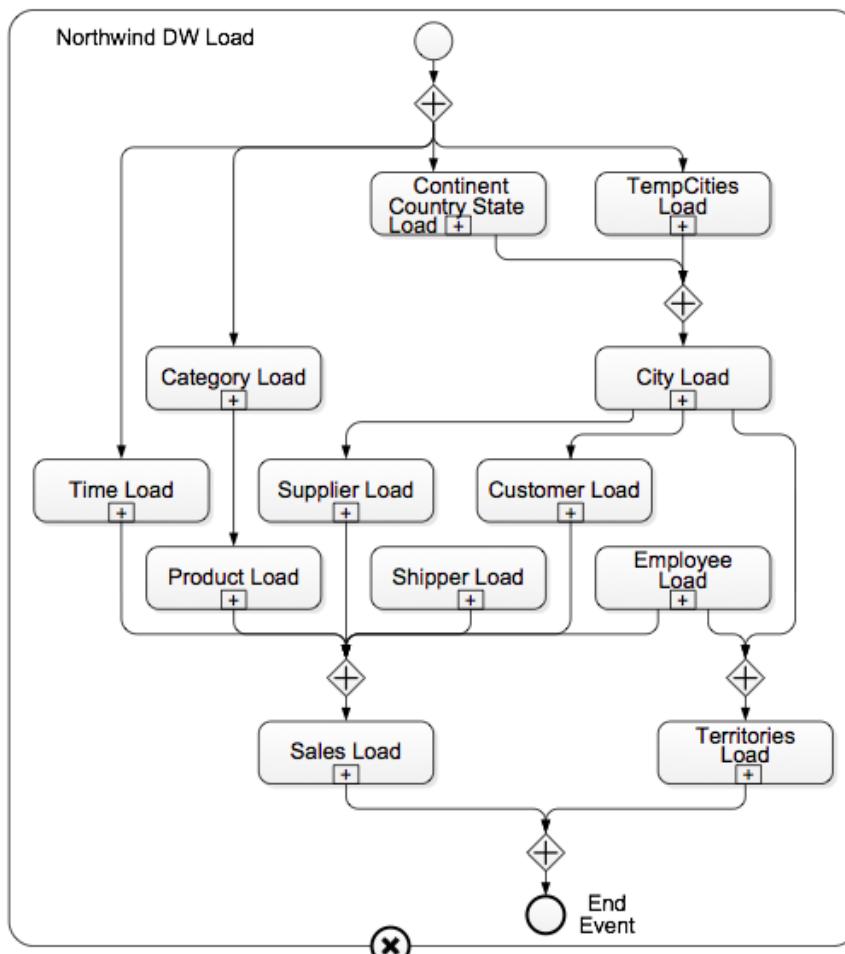
City → State → Country
Aachen → North Rhine-Westphalia → Germany
Albuquerque → New Mexico → USA
Anchorage → Alaska → USA
Ann Arbor → Michigan → USA
Annecy → Haute-Savoie → France
...

Beginning of the file **Cities.txt**

TempCities
City
State
Country

Associated table **TempCities**

Conceptual Design of the Northwind ETL: Overall View

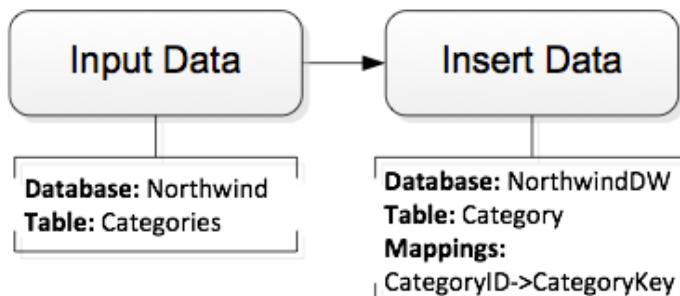
[TOP](#)

Created by Janusz R. Getta, ISIT312 Big Data Management, SIM, Session 2, 2021

15/18

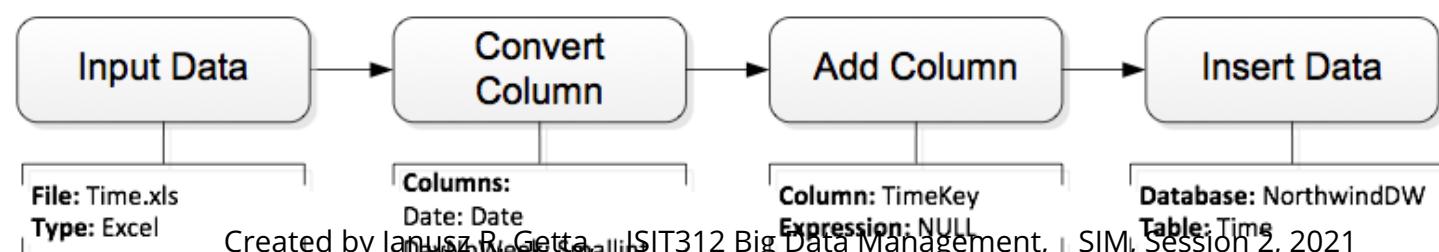
Conceptual Design of the Northwind ETL

Load of the **Category** dimension table



- Input task loads table **Categories** from the operational database
- Insert task loads the table **Category** in the data warehouse, mapping **CategoryID** to **CategoryKey** attribute in the **Category** table

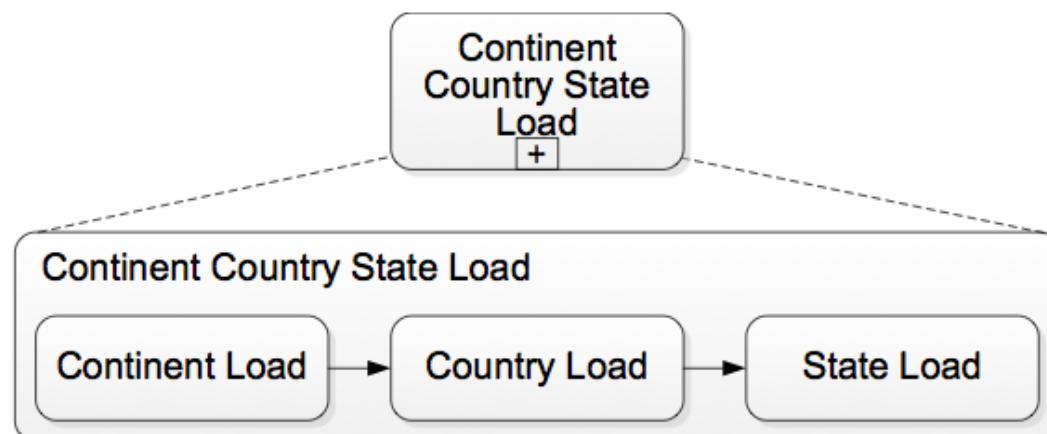
Loading the **Time** dimension table from an Excel file is similar, but includes a data type conversion, and an addition of the column **TimeKey**



Conceptual Design of the Northwind ETL

Loading the **City** level first requires loading the **Geography** hierarchy
State → Country → Continent

Associated control task



Load of the **Continent** table



References

A. VAISMAN, E. ZIMANYI, Data Warehouse Systems: Design and Implementation, Chapter 8 Extraction, Transformation, and Loading, Springer Verlag, 2014

ISIT312 Big Data Management

HBase Data Model

Dr Janusz R. Getta

School of Computing and Information Technology -
University of Wollongong

[TOP](#)

Created by Janusz R. Getta, ISIT312 Big Data Management, SIM, Session 4, 2020

HBase Data Model

Outline

Background

Logical view of data

Design fundamentals

Physical implementation

Performance

Background

Hbase is open source distributed database based on a data model of Google's BigTable

HBase provides a BigTable view of data stored in HDFS

HBase is also called as Hadoop DataBase

HBase still provides a tabular view of data however it is also very different from the traditional relational data model

HBase data model is a sparse, distributed, persistent multidimensional sorted map

It is indexed by a row key, column key, and timestamp

HBase Data Model

Outline

Background

Logical view of data

Design fundamentals

Physical implementation

Performance

Logical view of data

HBase organizes data into **tables**

HBase table consists of **rows**

Each **row** is uniquely identified by a **row key**

Data within a **row** is grouped by a **column family**

Column families have an important impact on the **physical implementation** of HBase table

Every **row** has the same **column families** although some of them can be **empty**

Data within a **column family** is addressed via its **column qualifier**, or simply, **column name**

Hence, a combination of **row key**, **column family**, and **column qualifier** uniquely identifies a **cell**

Values in cells do not have a data type and are always treated as **sequences of bytes**

Logical view of data

The table is lexicographically sorted on the row keys

Each cell has multiple versions, typically represented by the timestamp of when they were inserted into the table

Timestamp1 Timestamp2

Row Key	Column Family - Personal		Column Family - Office	
	Name	Residence phone	Phone	Address
00001	John	415-111-1234	415-212-5544	1021 Market St
00002	Paul	408-432-9922	415-212-5544	1021 Market St
00003	Ron	415-993-2124	415-212-5544	1021 Market St
00004	Rob	818-243-9988	408-998-4322	4455 Bird Ave
00005	Carly	206-221-9123	408-998-4325	4455 Bird Ave
00006	Scott	818-231-2566	650-443-2211	543 Dale Ave

Cells

Values within a cell have multiple versions

Versions are identified by their version number, which by default is a timestamp when the cell was written

[TOP](#)

Created by Janusz R. Getta, ISYS12 Big Data Management, SIM, Session 2, 2021

6/35

Logical view of data

If a **timestamp** is not determined at write time, then the **current timestamp** is used

If a **timestamp** is not determined during a read, the **latest one** is returned

The **maximum allowed number of cell value versions** is determined for each **column family**

The **default number of cell versions** is three

Logical view of data

A view of HBase table as a nested structure

```
{"Row-0001":  
    {"Home":  
        {"Name":  
            {"timestamp-1": "James"}  
        "Phones":  
            {"timestamp-1": "2 42 214339"  
             "timestamp-2": "2 42 213456"  
             "timestamp-3": "+61 2 4567890"}  
        }  
    "Office":  
        {"Phone":  
            {"timestamp-4": "+64 345678"}  
        "Address":  
            {"timestamp-5": "10 Ellenborough Pl"}  
        }  
    }  
}
```

HBase Table

Logical view of data

A view of HBase table as a nested structure



Logical view of data

A view HBase table as a key->value store

```
"Row-0001"-->{"Home": {"Name": {"timestamp-1": "James"},  
                         "Phones": {"timestamp-2": "2 42 214339",  
                                    "timestamp-3": "2 42 213456",  
                                    "timestamp-4": "+61 2 4567890"},  
                         },  
                 "Office": {"Phone": {"timestamp-5": "+64 345678"},  
                            "Address": {"timestamp-6": "10 Ellenborough Pl"},  
                            },  
                 }  
  
"Row-0001" "Home"--> {"Name": {"timestamp-1": "James"},  
                         "Phones": {"timestamp-2": "2 42 214339",  
                                    "timestamp-3": "2 42 213456",  
                                    "timestamp-4": "+61 2 4567890"},  
                         }  
  
"Row-0001" "Home" "Phones"--> {"timestamp-2": "2 42 214339",  
                                 "timestamp-3": "2 42 213456",  
                                 "timestamp-4": "+61 2 4567890"}  
Key->value  
Key->value  
Key->value
```

Logical view of data

A **key** can be **row key** or a combination of a **row key**, **column family**, **qualifier**, and **timestamp** depending on what supposed to be retrieved

If all the **cells** in a row are of interest then a **key** is a **row key**

If only specific **cells** are of interest, the appropriate **column families** and **qualifiers** are a part of a **key**

HBase Data Model

Outline

[Background](#)

[Logical view of data](#)

[Design fundamentals](#)

[Physical implementation](#)

[Performance](#)

Design Fundamentals

When designing **Hbase table** we have to consider the following questions:

- What should be a **row key** and what should it contain?
- How many **column families** should a **table** have?
- What **columns (qualifiers)** should be included in each **column family**?
- What information should go into the **cells**?
- How many **versions** should be stored for each **cell**?

In fact **HBase table** is a four level **hierarchical structure** where a **table** consists of **rows**, **rows** consists of **column families**, **column families** consist of **columns** and **columns** consists of **versions**

If cells contain the keys then **HBase table** becomes a **network/graph structure**

Design Fundamentals

Important facts to remember:

- Indexing is performed only for a **row key**
- **Hbase tables** are stored sorted based on a **row key**
- Everything in **Hbase tables** is stored as **untyped sequence of bytes**
- **Atomicity** is guaranteed only at a row level and there are no multi-row transactions
- **Column families** must be defined at **Hbase table** creation time
- **Column qualifiers** are dynamic and can be defined at write time
- **Column qualifiers** are stored as sequences of bytes such that they can represent data

Design Fundamentals

Implementation of Entity type

CUSTOMER	
cnumber	ID
first-name	
last-name	
phone	
email	

```
{"007":  
  {"CUSTOMER":  
    {"first-name": {"timestamp-1":"James"},  
     "last-name": {"timestamp-2":"Bond"},  
     "phone": {"timestamp-1":"007"}  
   }  
}
```

HBase Table

Design Fundamentals

Implementation of one-to-one relationship

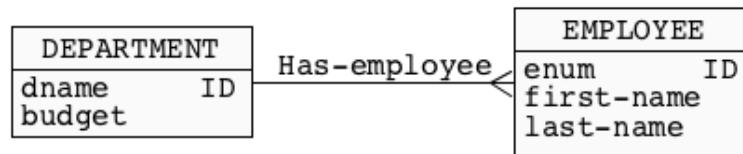


```
{"Sales":  
  {"DEPARTMENT":  
    {"budget": {"timestamp-1": "1000"}  
    }  
  }  
  {"MANAGER":  
    {"enumber": {"timestamp-2": "007"},  
     "first-name": {"timestamp-3": "James"},  
     "last-name": {"timestamp-4": "Bond"}  
    }  
  }  
}
```

HBase Table

Design Fundamentals

Implementation of one-to-many relationship

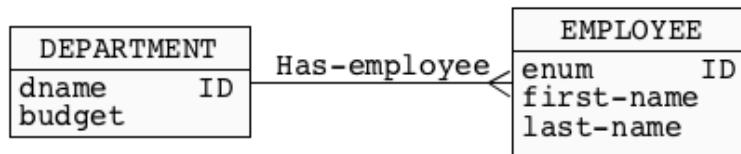


```
{"007":  
  {"EMPLOYEE":  
    {"enum": {"timestamp-1": "007"},  
     "first-name": {"timestamp-2": "James"},  
     "last-name": {"timestamp-3": "Bond"},  
     "department": {"timestamp-4": "Sales"}  
    }  
  }  
}
```

HBase Table

Design Fundamentals

Another implementation of one-to-many relationship

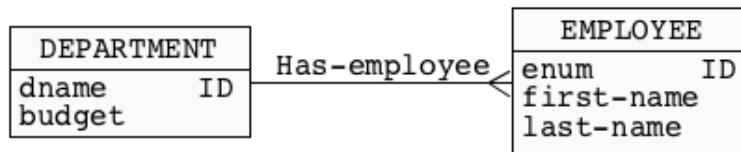


```
{"Sales":  
  {"DEPARTMENT":  
    {"budget": {"timestamp-1": "1234567"}  
  },  
  {"EMPLOYEE":  
    {"007": {"timestamp-2": "James Bond"},  
     "008": {"timestamp-3": "Harry Potter"},  
     "009": {"timestamp-4": "Robin Hood"}  
    ...  
  }  
}
```

HBase Table

Design Fundamentals

Yet another implementation of one-to-many relationship

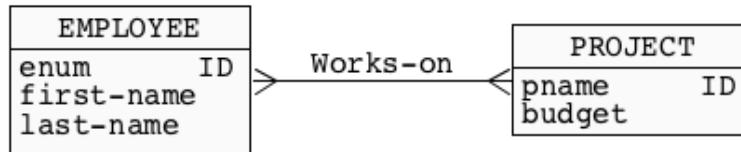


```
{"Sales":  
  {"DEPARTMENT":  
    {"budget": {"timestamp-1": "1000"}  
  }  
  {"HAS-EMPLOYEES":  
    {"employees": {"timestamp-2": "007 James Bond",  
                 {"timestamp-3": "008 Harry Potter"},  
                 {"timestamp-3": "009 Robin Hood"},  
                 ...  
    }  
  }  
}
```

HBase Table

Design Fundamentals

Implementation of many-to-many relationship

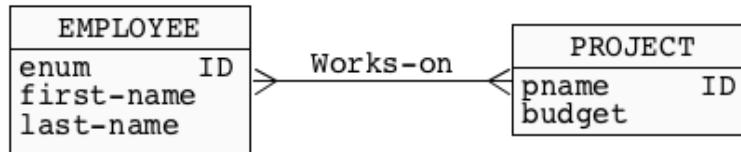


```
{"participation-1":  
    {"EMPLOYEE":  
        {"enum": {"timestamp-1": "007"},  
         "first-name": {"timestamp-2": "James"},  
         "last-name": {"timestamp-3": "Bond"},  
         "pnumber": {"timestamp-4": "project-1"},  
         {"timestamp-5": "project-2"},  
         "..."  
        }  
    }  
}
```

HBase Table

Design Fundamentals

Another implementation of many-to-many relationship

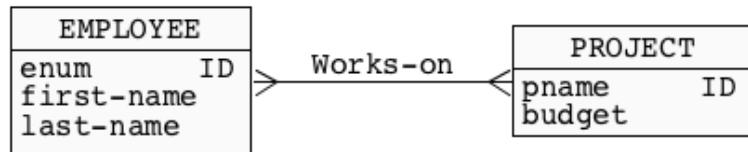


```
{"participation-1":  
    {"PROJECT":  
        {"pnumber": {"timestamp-1": "project-1"},  
         "budget": {"timestamp-2": "12345.25"},  
         "employee": {"timestamp-3": "007"},  
                  {"timestamp-4": "008"},  
                  {"timestamp-5": "009"},  
         ...  
     }  
}
```

HBase Table

Design Fundamentals

Another implementation of many-to-many relationship



```
{"participation-1":  
    {"PARTICIPATION":  
        {"pnumber": {"timestamp-1": "project-1"},  
         "employee": {"timestamp-2": "employee-007"}  
     }  
}  
  
{"participation-2":  
    {"PARTICIPATION":  
        {"pnumber": {"timestamp-1": "project-1"},  
         "employee": {"timestamp-2": "employee-008"}  
     }  
}
```

HBase Table

HBase Table

Design Fundamentals

Note, that it is possible to group in one **Hbase table** rows of different types

```
{"employee-007":  
    {"EMPLOYEE":  
        {"enumber": {"timestamp-1": "007"},  
         "first-name": {"timestamp-2": "James"},  
         "last-name": {"timestamp-3": "Bond"}  
    },  
    {"PROJECT":  
        {"pnumber": {"timestamp-4": "1"},  
         "budget": {"timestamp-5": "12345.25"}  
    },  
    {"PARTICIPATION":  
        {"pnumber": {"timestamp-1": "project-1"},  
         "employee": {"timestamp-2": "employee-007"}  
    }  
},  
"project-1":  
    {"PROJECT":  
        {"pnumber": {"timestamp-4": "1"},  
         "budget": {"timestamp-5": "12345.25"}  
    },  
    {"PARTICIPATION":  
        {"pnumber": {"timestamp-1": "project-1"},  
         "employee": {"timestamp-2": "employee-007"}  
    }  
},  
"participation-2":  
    {"PARTICIPATION":  
        {"pnumber": {"timestamp-1": "project-1"},  
         "employee": {"timestamp-2": "employee-007"}  
    }  
},  
"TOP":  
    {}}
```

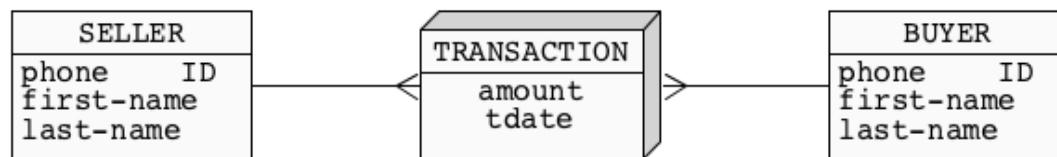
HBase Table

Created by Janusz R. Getta, ISIT312 Big Data Management, SIM, Session 2, 2021

23/35

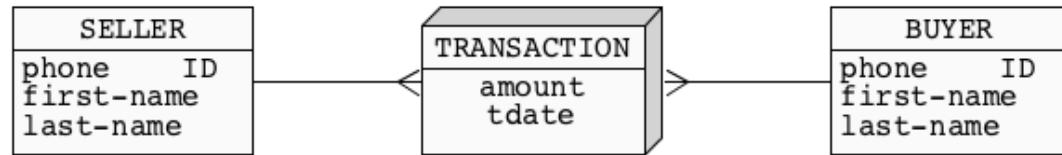
Design Fundamentals

Implementation of fact with dimensions



Design Fundamentals

Implementation of fact with dimensions



```
{"1234567":  
    {"MEASURE":  
        {"amount": {"timestamp-1": "1000000"}  
     }  
    "BUYER":  
        {"phone": {"timestamp-1": "242214339"},  
         "first-name": {"timestamp-1": "James"},  
         "last-name": {"timestamp-1": "Bond"}  
     }  
    "SELLER":  
        {"phone": {"timestamp-1": "242215612"},  
         "first-name": {"timestamp-1": "Harry"},  
         "last-name": {"timestamp-1": "potter"}  
     }  
}
```

HBase Table

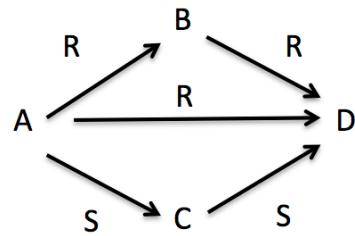
[TOP](#)

Created by Janusz R. Getta, ISIT312 Big Data Management, SIM, Session 2, 2021

25/35

Design Fundamentals

Implementation of graph structure



```
{"A":  
  {"R":  
    {"1":{"timestamp-1":"B"},  
     "2":{"timestamp-1":"D"}  
    }  
  "S":  
    {"1":{"timestamp-1":"C"}  
  }  
}
```

HBase Table

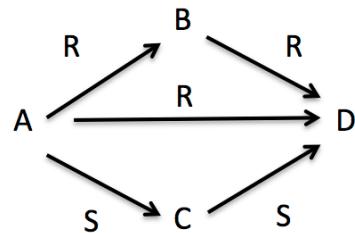
```
{"B":  
  {"R":  
    {"1":{"timestamp-1":"D"}  
  }  
}
```

HBase Table

[TOP](#)

Design Fundamentals

Implementation of graph structure



```
{"C":  
  {"S":  
    {"1":{"timestamp-1":"D"}  
  }  
}  
}  
  
{"D":  
  {}  
}
```

HBase Table

HBase Table

HBase Data Model

Outline

Background

Logical view of data

Design fundamentals

Physical implementation

Performance

Physical implementation

HBase is a database built on top of HDFS

HBase tables can scale up to billions of rows and millions of columns

Because Hbase tables can grow up to terabytes or even petabytes, Hbase tables are split into smaller chunks of data that are distributed across multiple servers

Chunks of data are called as regions and servers that host regions are called as region servers

Region servers are usually collocated with data nodes of HDFS

The splits of Hbase tables are usually horizontal, however, it is also possible to benefit from vertical splits separating column families

Region assignments happen when Hbase table grows in size or when a region server is malfunctioning or when a new region server is added

Physical implementation

Two special HBase tables **-ROOT-** and **.META.** keep information where the **regions** for the tables are hosted

A table **-ROOT-** never splits into more than one region, while **.META.** can be distributed over many **regions**

When a client application wants to access a particular row **-ROOT-** points it to the region of the **.META.** table that contains information about a row location

The entry point for an **HBase** system is provided by another system called **ZooKeeper**

ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services

HBase Data Model

Outline

Background

Logical view of data

Design fundamentals

Physical implementation

Performance

Performance

Too many **regions** affect performance

More regions means smaller memory flushes to persistent storage and smaller **HFiles** stored in **HDFS**

It requires **HBase** to process many compaction to keep the number of **HFiles** low

HBase can handle regions from 10 to 40 Gb

Too many regions may occur due to

- Over-splitting with **HBase's split feature**
- Improper presplitting with **HBase presplit feature**

Offline **merging regions** can be used to reduce number of regions

Performance

Too many **column families** affect performance

A magic total number of **column families** number is 3

Column families are built to group data with similar format or similar access pattern

Consequences of too many **column families**

- **Memory:** HBase shares write cache among all column families of the same regions, more column families means less transient memory for each one of them
- **Compactions:** the total number of **column families** affects the total number of store files created during flushes and subsequently number of compactions
- **Split:** HBase stores **column families** into separate files and directories, when a directory becomes too large it is **split**; **split** affects all **column families**, if some of them are small only few cells are stored in a directory

A solution is a correct schema design followed by **delete column family**, **merge column family**, or **separate column family** in a new **HBase table**

Performance

All writes and reads should be uniformly distributed across the regions

Hotspotting occurs when a given region serviced by a single Region Server receives most or all of the read or write requests

Causes of **hotspotting**:

- **Monotonically incrementing keys**: only last bits or bytes are slowly changing, e.g. timestamp used as a key, all write to a consecutive area of keys go to the same region
- **Poorly distributed keys**: e.g. keys are always appended with the same prefix going to the same region
- **Small reference (lookup) tables**: All reference to lookup tables must be handled by the same region server
- **Application issues**: e.g. writes always performed on the same region
- **Meta region hotspotting**: e.g. creating too many connections

References

Kerzner M., Maniyam S., HBase Design Patterns, Packt Publishing 2014
(Available from UoW Library)

Jiang, Y. HBase Administration Cookbook, Pack Publishing, 2012
(Available from UoW Library)

Dimiduk N., Khurana A., HBase in Action, Manning Publishers, 2013

Spaggiari J-M., O'Dell K., Architecting HBase Applications, O'Reilly, 2016

ISIT312 Big Data Management

HBase Operations

Dr Janusz R. Getta

School of Computing and Information Technology -
University of Wollongong

HBase Operations

Outline

HBase shell

Data definition commands

Data manipulation commands

HBase Java AP

HBase shell

HBase provides extensible JRuby-based ([Java Interactive Ruby - JIRB](#)) shell as a feature to execute some commands

The shell is a typical [Read–Eval–Print–Loop \(REPL\)](#) shell also known as a [language shell](#)

It is a simple, interactive computer programming environment that takes single user inputs evaluates them, and returns the result to the user; a program written in a [REPL](#) environment is executed piecewise

It means that [HBase shell](#) allows for computation of Ruby scripts and brings all features enabled in [JIRB shell](#)

It allows to process a script of [HBase](#) commands saved in a file [file-name.hb](#) in the following way:

```
source 'file-name.hb'
```

HBase shell

HBase Operations

Outline

[HBase shell](#)

[Data definition commands](#)

[Data manipulation commands](#)

[HBase Java AP](#)

Data definition commands

In Hbase, a set of **data definition** commands includes: `create`, `list`, `describe`, `disable`, `disable_all`, `enable`, `enable_all`, `drop`, `drop_all`, `show_filters`, `alter`, `alter_status`

Create a table '`student`' with a column family '`personal`'

```
create 'student', 'personal'
```

HBase shell

Show a structure of a table '`student`'

```
describe 'student'
```

HBase shell

Implement a column family '`personal`' in transient memory

```
alter 'student', {NAME=>'personal', IN_MEMORY=>true}
```

HBase shell

Add a column family '`uni`' to a table '`student`'

```
alter 'student', {NAME=>'uni', VERSIONS=>'4'}
```

HBase shell

Data definition commands

Delete a column family '**uni**' from a table '**student**'

```
alter 'student','delete'=>'uni'
```

HBase shell

Add a column family '**university**' to a table **student** and allow for 5 versions in each cell in the column family

```
alter 'student',{NAME=>'university',VERSIONS=>5}
```

HBase shell

Increase a number of allowed versions in a column family '**personal**' to 3

```
alter 'student',{NAME=>'personal',VERSIONS=>3}
```

HBase shell

HBase Operations

Outline

[HBase shell](#)

[Data definition commands](#)

[Data manipulation commands](#)

[HBase Java AP](#)

Data Manipulation commands

In Hbase, a set of data manipulation commands includes: **count**, **put**, **get**, **delete**, **delete_all**, **truncate**, **scan**

Put a value 'James' into a cell in a column family '**personal**', qualification '**first-name**', row key '**007**',

```
put 'student','007','personal:first-name','James'
```

HBase shell

Put a value 'Bond' into a cell in a column family '**personal**', qualification '**last-name**', row key '**007**'

```
put 'student','007','personal:last-name','Bond'
```

HBase shell

Put a value '**01-OCT-1960**' into a cell in a column family '**personal**', qualification **dob**', row key '**007**',

```
put 'student','007','personal:dob','01-OCT-1960'
```

HBase shell

Data Manipulation commands

List the contents of a table '**student**'

```
scan 'student'
```

HBase shell

Put a value '**02-OCT-1960**' as the second version into a cell in a column family '**personal**', qualification **dob**', row key '**007**',

```
put 'student','007','personal:dob','02-OCT-1960'
```

HBase shell

Get no more than **5** versions of a cell '**dob**' in a column family '**personal**' from a row '**007**' in a table '**student**'

```
get 'student','007',{COLUMN=>'personal:dob', VERSIONS=>5}
```

HBase shell

Get no more than **5** versions of a cell '**dob**' in a column family '**personal**', from a table '**student**'

```
scan 'student',{COLUMN=>'personal:dob', VERSIONS=>5}
```

HBase shell

Data Manipulation commands

Get all column families in a row '**666**' in a table '**student**'

```
get 'student','666'
```

HBase shell

Get no more than 5 versions of values from all cells in a column family '**grade**' in a row '**666**' in a table '**student**'

```
get 'student','666',{COLUMN=>'grade',VERSIONS=>5}
```

HBase shell

Get no more than 5 versions of values from a cell '**CSCI235**' in a column family '**grade**' in a row '**666**' in a table '**student**'

```
get 'student','666',{COLUMN=>'grade:CSCI235',VERSIONS=>5}
```

HBase shell

Get no more than 5 versions of values from a cell '**dob**' in a column family '**grade**' in a row '**666**' in a table '**student**'

```
get 'student','007',{COLUMN=>'personal:dob',VERSIONS=>5}
```

HBase shell

Data Manipulation commands

Count total number of rows in a table '**student**'

```
count 'student'
```

HBase shell

Get entire table '**student**', one version per cell

```
scan 'student'
```

HBase shell

Get entire table '**student**', at most 5 versions per cell

```
scan 'student',{VERSIONS=>5}
```

HBase shell

Get all cells '**dob**' from in a column family '**personal**' from entire table '**student**', at most 5 versions per cell

```
scan 'student',{COLUMNS=>'personal:dob',VERSIONS=>5}
```

HBase shell

Data Manipulation commands

Get all cells from the column families '**personal**' and '**university**' from entire table '**student**'

```
scan 'student',{COLUMNS=>['personal','university']}
```

HBase shell

Get at most 5 versions of cells 'dob' with timestamps in a range [1,1502609828830], from a column family '**personal**' from entire table '**student**'

```
scan 'student',{COLUMNS=>'personal:dob',TIMERANGE=>[1,1502609828830],  
VERSIONS=>5}
```

HBase shell

Get at most 5 versions of cells 'dob' with timestamps in a range [1,1502609828830], from a column family '**personal**' from entire table '**student**'

```
scan 'student',{COLUMNS=>'personal:dob',  
FILTER=>"TimestampsFilter(1,1502609828830)",VERSIONS=>5}
```

HBase shell

Data Manipulation commands

Get all cells whose name is \geq than 'f' in a table 'student'

```
scan 'student',{FILTER=>"QualifierFilter(>=,'binary:f'))"}
```

HBase shell

Get all rows from a table 'student' that have value of a cell \geq than 'J'

```
scan 'student',{FILTER=>"ValueFilter(>=,'binary:J'))"}
```

HBase shell

Get all rows from a table 'student' that have value of a cell in a range
['J' , 'K']

```
scan 'student',{FILTER=>"ValueFilter(>=,'binary:J') AND  
ValueFilter(<=,'binary:K'))"}
```

HBase shell

Get all values of cells 'dob' in a column family 'personal' from rows in a table 'student' where a cell 'dob' has a value '02-OCT-1960'

```
scan 'student',{COLUMNS=>'personal:dob',FILTER=>  
"QualifierFilter(=,'binary:dob') AND  
ValueFilter(=,'binary:02-OCT-1960'))"}
```

HBase shell

Data Manipulation commands

Delete a cell 'CSCI235' from a column family 'student' in a row '666' in a table 'student'

```
delete 'student', '666', 'grade:CSCI235'
```

HBase shell

Delete entire row '007' from a table 'student'

```
deleteall 'student', '007'
```

HBase shell

HBase Operations

Outline

[HBase shell](#)

[Data definition commands](#)

[Data manipulation commands](#)

[HBase Java AP](#)

HBase Java API

HBase Java Application Program Interface allows to access HBase tables from programs written in Java

The client APIs provide both data definition and data manipulation features

Creating a table 'my-table' and column families 'Address' and 'Name'

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.HColumnDescriptor;
import org.apache.hadoop.hbase.HTableDescriptor;
import org.apache.hadoop.hbase.TableName;
import org.apache.hadoop.hbase.client.HBaseAdmin;

public class CreateTable {
    public static void main(String[] args) throws Exception {
        Configuration conf = HBaseConfiguration.create();
        HBaseAdmin admin = new HBaseAdmin(conf);
        HTableDescriptor tableDescriptor = new HTableDescriptor(TableName.valueOf("my-table"));
        tableDescriptor.addFamily(new HColumnDescriptor("Address"));
        tableDescriptor.addFamily(new HColumnDescriptor("Name"));
        admin.createTable(tableDescriptor);
        boolean tableAvailable = admin.isTableAvailable("my-table");
        System.out.println("tableAvailable = " + tableAvailable); } }
```

Java

Java

HBase Java API

Inserting data into HBase table 'my-table'

```
public class PutRow {  
    public static void main(String[] args) throws Exception {  
        Configuration conf = HBaseConfiguration.create();  
        HTable table = new HTable(conf, "my-table");  
        Put put = new Put(Bytes.toBytes("007"));  
        put.add(Bytes.toBytes("Address"), Bytes.toBytes("City"), Bytes.toBytes("Dapto"));  
        put.add(Bytes.toBytes("Address"), Bytes.toBytes("Street"), Bytes.toBytes("Ellenborough"));  
        put.add(Bytes.toBytes("Name"), Bytes.toBytes("First"), Bytes.toBytes("James"));  
        put.add(Bytes.toBytes("Name"), Bytes.toBytes("Last"), Bytes.toBytes("Bond"));  
        table.put(put);  
        table.flushCommits();  
        table.close();  
    }  
}
```

Java

HBase Java API

Getting data from HBase table 'my-table'

```
import java.util.Map;                                     Java
import java.util.NavigableMap

public class GetRow {
    public static void main(String[] args) throws Exception {
        Configuration conf = HBaseConfiguration.create();
        HTable table = new HTable(conf, "my-table");
        Get get = new Get(Bytes.toBytes("007"));
        get.setMaxVersions(3);
        get.addFamily(Bytes.toBytes("Address"));
        get.addColumn(Bytes.toBytes("Name"), Bytes.toBytes("First"));
        get.addColumn(Bytes.toBytes("Name"), Bytes.toBytes("Last"));

        // Get a specific value
        Result result = table.get(get);
        String row = Bytes.toString(result.getRow());
```

HBase Java API

Getting data from HBase table 'my-table'

Java

```
String specificValue = Bytes.toString(result.getValue(Bytes.toBytes("Address"),
                                                       Bytes.toBytes("City")));
System.out.println("Latest Address:City is: " + specificValue);
specificValue = Bytes.toString(result.getValue(Bytes.toBytes("Address"),
                                               Bytes.toBytes("Street")));
System.out.println("Latest Address:Street is: " + specificValue);
specificValue = Bytes.toString(result.getValue(Bytes.toBytes("Name"),
                                               Bytes.toBytes("First")));
System.out.println("Latest Name:First is: " + specificValue);
specificValue = Bytes.toString(result.getValue(Bytes.toBytes("Name"),
                                               Bytes.toBytes("Last")));
System.out.println("Latest Name>Last is: " + specificValue)
```

HBase Java API

```
// Traverse entire returned row
    System.out.println("Row key: " + row);
    NavigableMap<> map = result.getMap();
    for (Map.Entry<> navigableMapEntry : map.entrySet()) {
        String family = Bytes.toString(navigableMapEntry.getKey());
        System.out.println("\t" + family);
        NavigableMap<> familyContents = navigableMapEntry.getValue();
        for (Map.Entry<> mapEntry : familyContents.entrySet()) {
            String qualifier = Bytes.toString(mapEntry.getKey());
            System.out.println("\t\t" + qualifier);
            NavigableMap<> qualifierContents = mapEntry.getValue();
            for (Map.Entry<> entry : qualifierContents.entrySet()) {
                Long timestamp = entry.getKey();
                String value = Bytes.toString(entry.getValue());
                System.out.printf("\t\t\t%s, %d\n", value, timestamp);
            }
        }
    }
    table.close();
}
```

Java

References

HBase shell commands, <https://learnhbase.wordpress.com/2013/03/02/hbase-shell-commands/>

HBase shell and General commands, <https://www.guru99.com/hbase-shell-general-commands.html#4>

HBase Java API, <https://dzone.com/articles/handling-big-data-hbase-part-4>

Kerzner M., Maniyam S., HBase Design Patterns, Packt Publishing 2014
(Available from UoW Library)

Jiang, Y. HBase Administration Cookbook, Pack Publishing, 2012
(Available from UoW Library)

Dimiduk N., Khurana A., HBase in Action, Manning Publishers, 2013

Spaggiari J-M., O'Dell K., Architecting HBase Applications, O'Reilly, 2016

ISIT312 Big Data Management

Apache Pig

Dr Janusz R. Getta

School of Computing and Information Technology -
University of Wollongong

Apache Pig

Outline

What is Pig ?

Grunt: Interactive shell

Data model

What is Pig ?

Pig provides an engine for processing data flows in parallel on Hadoop

It includes a language Pig Latin, for representation of the data flows

Pig Latin includes operators for many of the traditional data operations ([join](#), [sort](#), [filter](#), etc.), as well as the ability for users to develop their own functions for reading, processing, and writing data

Pig runs on Hadoop, in the sense that it makes use of both HDFS, and is built on top of Hadoop processing framework MapReduce

Data flow language

The language of **Pig**, called **Pig Latin** is a **dataflow language**

It allows the users to describe how data from one or more inputs should be read, processed, and then stored to one or more outputs in parallel

The data flow in **Pig** is represented by a **Direct Acyclic Graph (DAG)**

There is no if statement or for loops in **the Pig Latin**

Comparing query and data flow languages:

- **SQL** is oriented around answering one question, while **Pig** specifies how data are manipulated
- **SQL** is oriented around answering one query, while **Pig** is designed for sequences of operations

Pig Latin

WordCount example in Pig Latin

```
input = load 'file' as (line);
-- Load input from the file named Mary, and call the single field in the record
  'line'

words = foreach input generate flatten(TOKENIZE(line)) as word;
-- TOKENIZE splits the line into a field for each word
-- flatten takes the collection of records returned by TOKENIZE and produce
  a separate record for each one, calling the single field in the record word

grpd = group words by word;
-- Now group them together by each word

cntd = foreach grpd generate group, COUNT(words);
-- Count them

dump cntd;
-- Print out the results
```

WordCount example

Differences between Pig and MapReduce

MapReduce programming model is expressive but relatively low-level, while Pig can be seen as a linguistic abstraction on MapReduce

- For example, the implementation of standard data-processing operations (such as `join`, `filter`, `group by`, etc.) is non-trivial in MapReduce, but these operations are primitive in Pig Latin

Pig Latin reduces the development time of MapReduce programming and is much lower cost to write and maintain than Java code for MapReduce

On the other hand, Pig does not support the object-oriented development

Also, the execution of Java programs for MapReduce is usually more efficient than an equivalent Pig script

A short history of Pig

Pig started out as a research project in Yahoo! Research, where Yahoo! scientists designed it and produced an initial implementation

As explained in a paper presented at SIGMOD in 2008, the researchers felt that the MapReduce paradigm presented by Hadoop is too low-level and rigid, and leads to a great deal of custom user code that is hard to maintain and reuse

At the same time they observed that many MapReduce users were not comfortable with declarative languages such as SQL

Thus they set out to produce a new language called Pig Latin that we have designed to fit in a sweet spot between the declarative style of SQL, and the low-level, procedural style of MapReduce

Yahoo! Hadoop users started to adopt Pig and a team of development engineers was assembled to take the research prototype and build it into a production-quality product

About this same time, in fall 2007, the first Pig release came a year later in September 2008. Later that same year, Pig graduated from the

7/22

Incorporated and became a sub-project of Apache Hadoop

Apache Pig

Outline

[What is Pig ?](#)

[Grunt: Interactive shell](#)

[Data model](#)

Grunt: Interactive shell

Grunt enables users to enter Pig Latin interactively and provides a shell for users to interact with HDFS

To enter Grunt, type

```
|- pig
```

Starting Grunt

will result in the prompt

```
|- grunt >
```

Grunt prompt

This will interact with a cluster configuration set in the classpath of Pig; the following command will give a shell ability to interact with a local file system

```
|- pig -x local
```

Starting Grunt

Entering Pig Latin scripts in Grunt

Pig Latin commands can be entered directly into Grunt. For example:

Pig Lagtin commands

```
pig
grunt> dividends = load "NYSE_dividends" as (exchange, symbol, data,dividend)
grunt> symbols = foreach dividends generate symbl;
... Error during parsing. Invalid alias: symbl ...
grunt> symbols = foreach dividends generate symbol;
grunt> dump symbols;
```

Pig will not start executing the Pig Latin you enter until it sees either a **store** or **dump**

However, it will do basic syntax and semantic checking to help you catch errors quickly

To quit **Grunt**, type

Quiting Grunt

```
grunt> quit
```

HDFS commands in Grunt

Besides entering **Pig Latin** interactively, the other major use of **Grunt** is to act as a shell for **HDFS**

In versions 0.5 and later of **Pig**, all hadoop **fs** shell commands are available

For example

Hadoop commands in grunt

```
grunt> fs -ls
```

lists all the files and folders at the current **HDFS** user home folder

Other commands include **cat**, **copyFromLocal**, **rm**, ...

Apache Pig

Outline

What is Pig ?

Grunt: Interactive shell

Data model

Types

Data types in **Pig** can be divided into two categories: **scalar types**, which contain a single value, and **complex types**, which contain other types

Scalar Types

`int`

- An integer; integers are represented in interfaces by `Java.lang.Integer` and stored a four-byte signed integer numbers

`long`

- A long integer; long integers are represented in interfaces by `java.lang.Long` and store an eight-byte signed integer numbers

`float`

- A floating-point number; floating point numbers are represented in interfaces by `java.lang.Float` and use four bytes to store their values

Scalar Types

double

- A double-precision floating-point number; double precision floating point numbers are represented in interfaces by `java.lang.Double` and use eight bytes to store their values

chararray

- A string or character array; string or character arrays are represented in interfaces by `java.lang.String`
- Constant chararrays are expressed as string literals with single quotes, for example, '`fred`'

bytearray

- A blob or array of bytes; blobs or arrays of bytes are represented in interfaces by a Java class `DataByteArray` that wraps a Java `byte[]`

Complex Types

map

A map in **Pig** is a **chararray** to data element mapping, where that element can be any Pig type, including a complex type.

The **chararray** is called a key and is used as an index to find the element, referred to as the value

By default there is no requirement that all values in a **map** must be of the same type

map constants are formed using brackets to delimit the map, a hash between keys and values, and a comma between key-value pairs

For example, ['name' #'bob' , 'age' #55] will create a map with two keys, "name" and "age"

Note that the first value is a **chararray**, and the second is an **int**

Complex Types

tuple

A **tuple** is a fixed-length, ordered collection of **Pig** data elements

tuples are divided into fields, with each field containing one data element

These elements can be of any type and they do not all need to be the same type

A **tuple** is analogous to a row in **SQL**, with the fields being **SQL** columns

Tuple constants use parentheses to indicate the tuple and commas to delimit fields in the tuple

For example, `('bob', 55)` describes a tuple constant with two fields

Complex Types

bag

A **bag** is an unordered collection of tuples

Because it has no order, it is not possible to reference tuples in a bag by position

Like tuples, a **bag** can, but is not required to, have a schema associated with it

In the case of a **bag**, the schema describes all tuples within the **bag**

bag constants are constructed using braces, with tuples in the **bag** separated by commas

For example, `{('bob', 55), ('sally', 52), ('john', 25)}` constructs a **bag** with three tuples, each with two fields

Null Data

Pig includes the concept of a data element being null

Data of any type can be null

In Pig, the concept of null is the same as in SQL, which is completely different from the concept of null in C, Java, Python, etc

In Pig a null data element means the value is unknown

Schemas

The easiest way to communicate the schema of your data to **Pig** is to explicitly tell **Pig** what it is when you load the data

Loading with a schema

```
dividends = load 'NYSE_dividends' as  
    (exchange:chararray, symbol:chararray, date:chararray, dividend:float);
```

Pig now expects your data to have four fields and iff it has more, it will truncate the extra ones

If it has less, it will pad the end of the record with **nulls**

It is also possible to specify the schema without giving explicit data types. In this case, the data type is assumed to be **bytearray**

Schema with no data types

```
dividends = load 'NYSE_dividends' as (exchange, symbol, date, dividend);
```

Schemas

You can declare data as complex data types in a schema:

Declaring complex data types

Data type Syntax

map `map[] or map[type]`, where type is any valid type

This declares all values in the map to be of this type

Example

as (a:map[], b:map[int])

tuple

`tuple() or tuple(list_of_fields)`, where list_of_fields as (a:tuple(), b:tuple(x:int, y:int))

is a comma-separated list of field declarations

bag

`tuple() or tuple(list_of_fields)`, where list_of_fields as (a:bag{}, b:bag{t:(x:int, y:int)})

is a comma-separated list of field declarations

Schema

We can get the schema of a relation by using the **describe** operation

Describing a schema

```
dividends = load 'NYSE_dividends' as  
    (exchange:chararray, symbol:chararray, date:chararray, dividend:float);  
described dividends;  
grpd: {exchange: chararray, stock: chararray, date: chararray, dividends: float}
```

Question: what is the output of the following

Describing a schema

```
dividends = load 'NYSE_dividends' as (exchange, symbol, date, dividend);  
described dividends;
```

References

Gates A., Programming Pig, O'Reilly Media, Inc., 2011, (Available in **READINGS** folder)

Vaddeman B., Beginning Apache Pig: Big Data Processing Made Easy, Apress 2016 (Available in **READINGS** folder)

ISIT312 Big Data Management

Introduction to Spark

Dr Guoxin Su and Dr Janusz R. Getta

School of Computing and Information Technology -
University of Wollongong

Introduction to Spark

Outline

MapReduce Challenges

Meet Spark !

Features of Spark

Spark Architecture

Overview of Spark Components

Spark Glossary

MapReduce Challenges

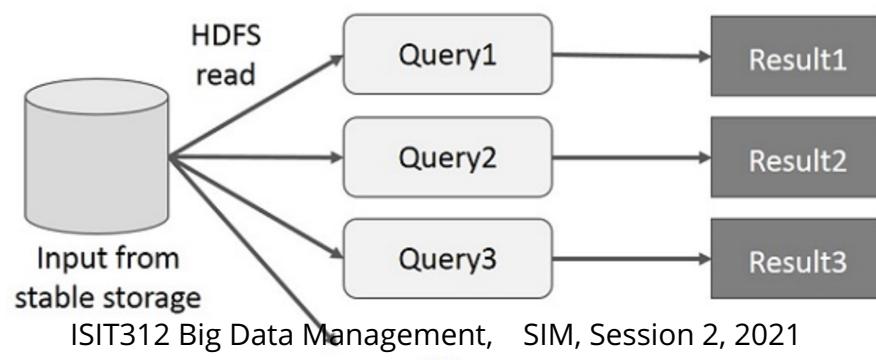
Google introduced **MapReduce**, which was a breakthrough in the history of big data technologies

- It makes the processing of big data feasible and practical

However, **Hadoop MapReduce framework** releases the developer from the distributing computing trickiness, its **Java API** is still too low-level

- Think how would you implement an inner join in **MapReduce** ?
- **Hive**, **Pig** and other frameworks based on **MapReduce** can help

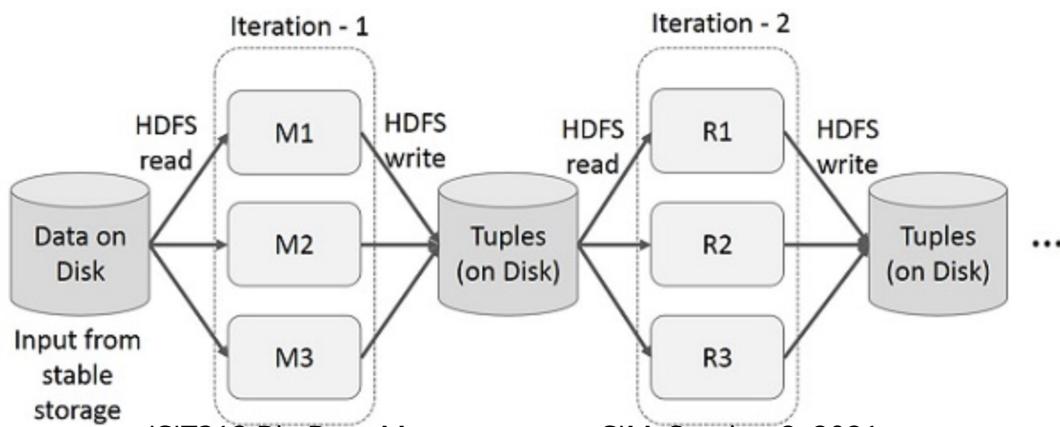
More importantly, the persistent storage I/O makes the interactive and iterative computation (two important forms of data processing) very inefficient and leads to potential I/O latency



MapReduce Challenges

Data Sharing in Hadoop

- In the [Hadoop MapReduce](#) framework, the reuse data between computations (e.g., between two [MapReduce](#) jobs) is to write it to an external stable storage system, for example [HDFS](#))
- Although [Hadoop](#) provides numerous abstractions for data access, data sharing is slow due to replication, serialisation and persistent storage IO
- **More than 90% of the time for running a MapReduce job is doing HDFS read-write operations**
- Substantial overhead is caused by data replication, persistent storage I/O, and serialization



Introduction to Spark

Outline

[MapReduce Challenges](#)

[Meet Spark !](#)

[Features of Spark](#)

[Spark Architecture](#)

[Overview of Spark Components](#)

[Spark Glossary](#)

Meet Spark !

Apache Spark, as a cluster computing platform, is designed to be **fast** and **general-purpose**

On the speed, **Spark** extends the **MapReduce** model to efficient support more types of computations, for example, interactive and iterative computations

- This is mainly due to **Spark's in-memory computing**, although **Spark** is also more efficient than **MapReduce** for persistent storage complex applications

On the generality, **Spark** covers a wide ranges of workloads, including batch applications, iterative queries, streaming and advanced analytics

- It provides a **unified environment** makes the combination of those different engines easy and inexpensive
- It provides **multiple language APIs**, including **Scala**, **Python**, **Java**, **SQL**, and **R**, as well as a very rich (yet fast-growing) built-in libraries

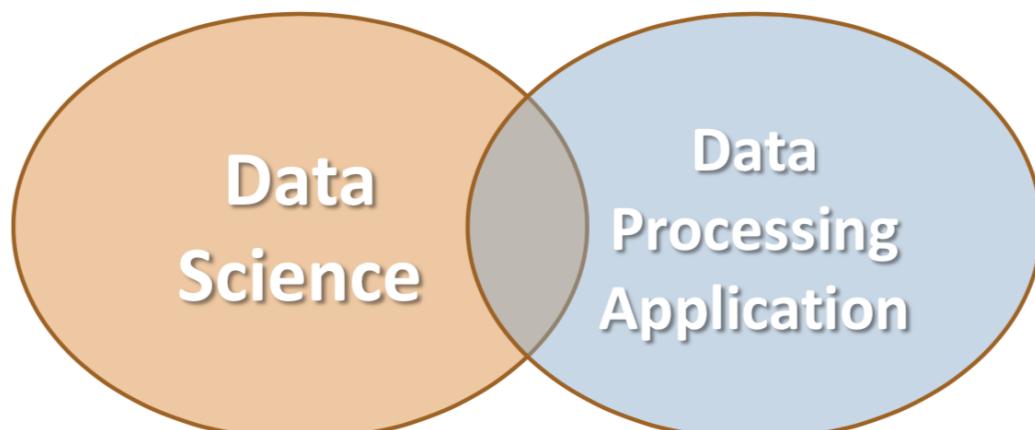
Meet Spark !

Data Scientists use Spark to:

- Analyse and model data
- Make prediction based on data
- Build data pipelines to fulfil certain tasks

Data Engineers use Spark to:

- Develop data processing applications
- Deploy the output of data scientists in production



Introduction to Spark

Outline

[MapReduce Challenges](#)

[Meet Spark !](#)

[Features of Spark](#)

[Spark Architecture](#)

[Overview of Spark Components](#)

[Spark Glossary](#)

Features of Spark

As mentioned, **Spark** supports not only batch computing (as **MapReduce** does) but also **interactive**, **iterative**, and **real-time** computing

The main feature of **Spark** is its in-memory cluster computing that increases the processing speed of an application

The programming model of **Spark** is based on **Directed Acyclic Graphs (DAG)** and it is more flexible than the **MapReduce** model

Unlike **MapReduce**, **Spark** has built-in high-level APIs to process structured data, e.g., tables in **Hive**

Spark reduces the management burden of maintaining separate tools

Features of Spark

Speed

Spark helps to run an application in Hadoop cluster, up to 100 times faster in memory, and 10 times faster when running on disk

- This is achieved by reducing number of read/write operations to persistent storage
- It stores the intermediate processing data in memory

Support for multiple languages

Spark provides built-in APIs in Java, Scala, or Python

Advanced Analytics

Spark not only supports MapReduce. It also supports SQL queries, Streaming data, Machine Learning (ML), and Graph algorithms

Introduction to Spark

Outline

[MapReduce Challenges](#)

[Meet Spark !](#)

[Features of Spark](#)

[Spark Architecture](#)

[Overview of Spark Components](#)

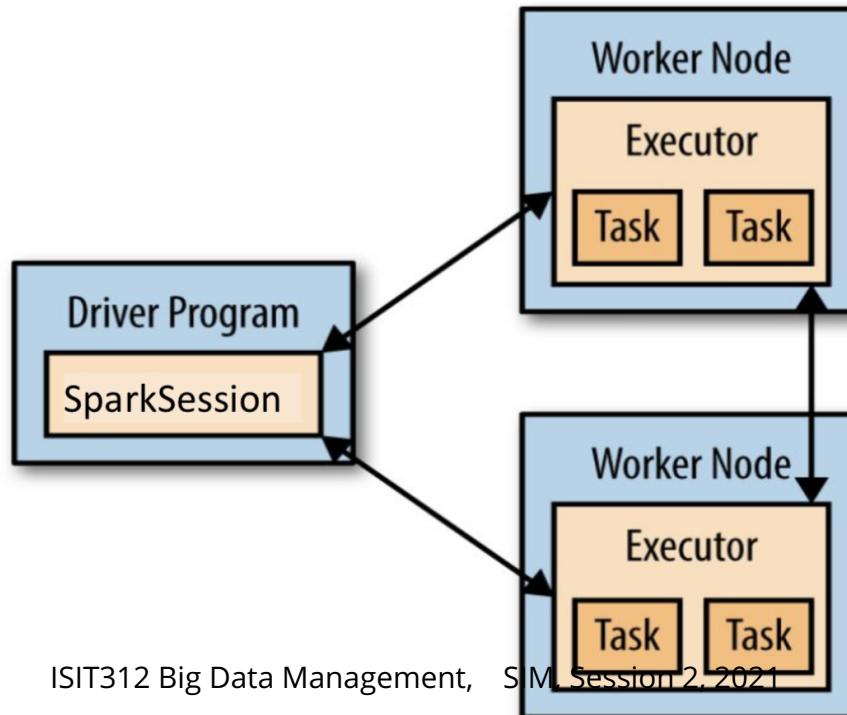
[Spark Glossary](#)

Spark Architecture

Driver program is the main **Spark** application that consists of the data processing logic

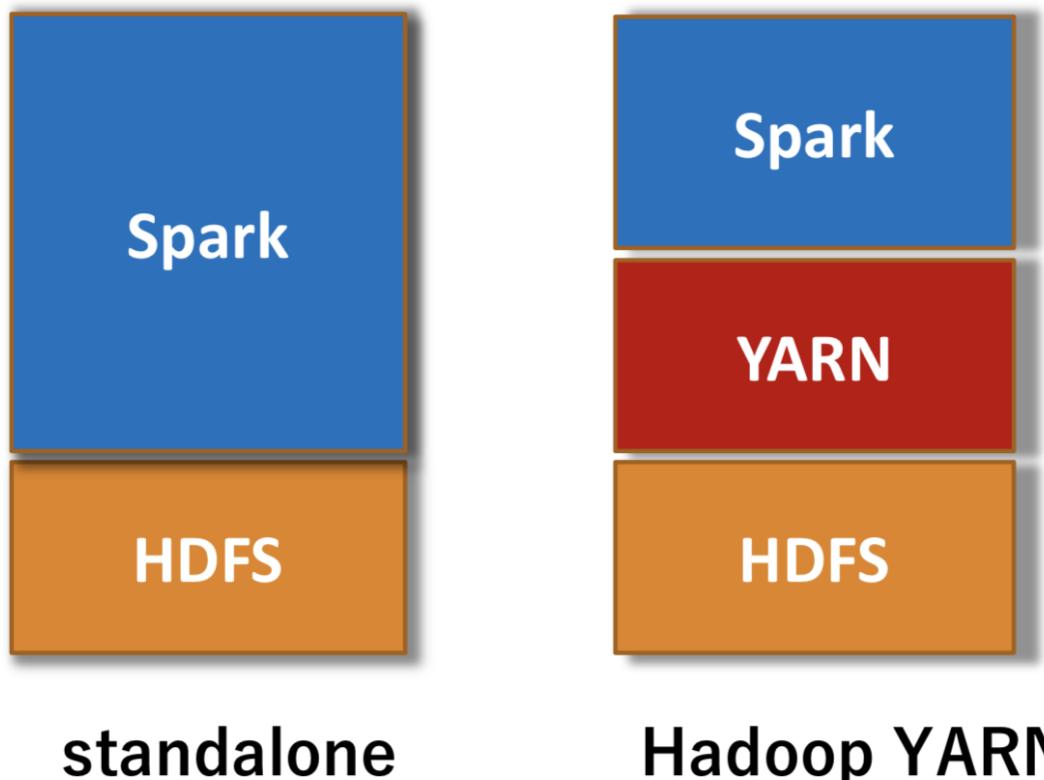
Executor is a **JVM** process that runs on each worker node and processes the jobs that the driver program submits

Task is a subcomponent of a data processing job



Spark Architecture

Two modes of Building Spark on Hadoop

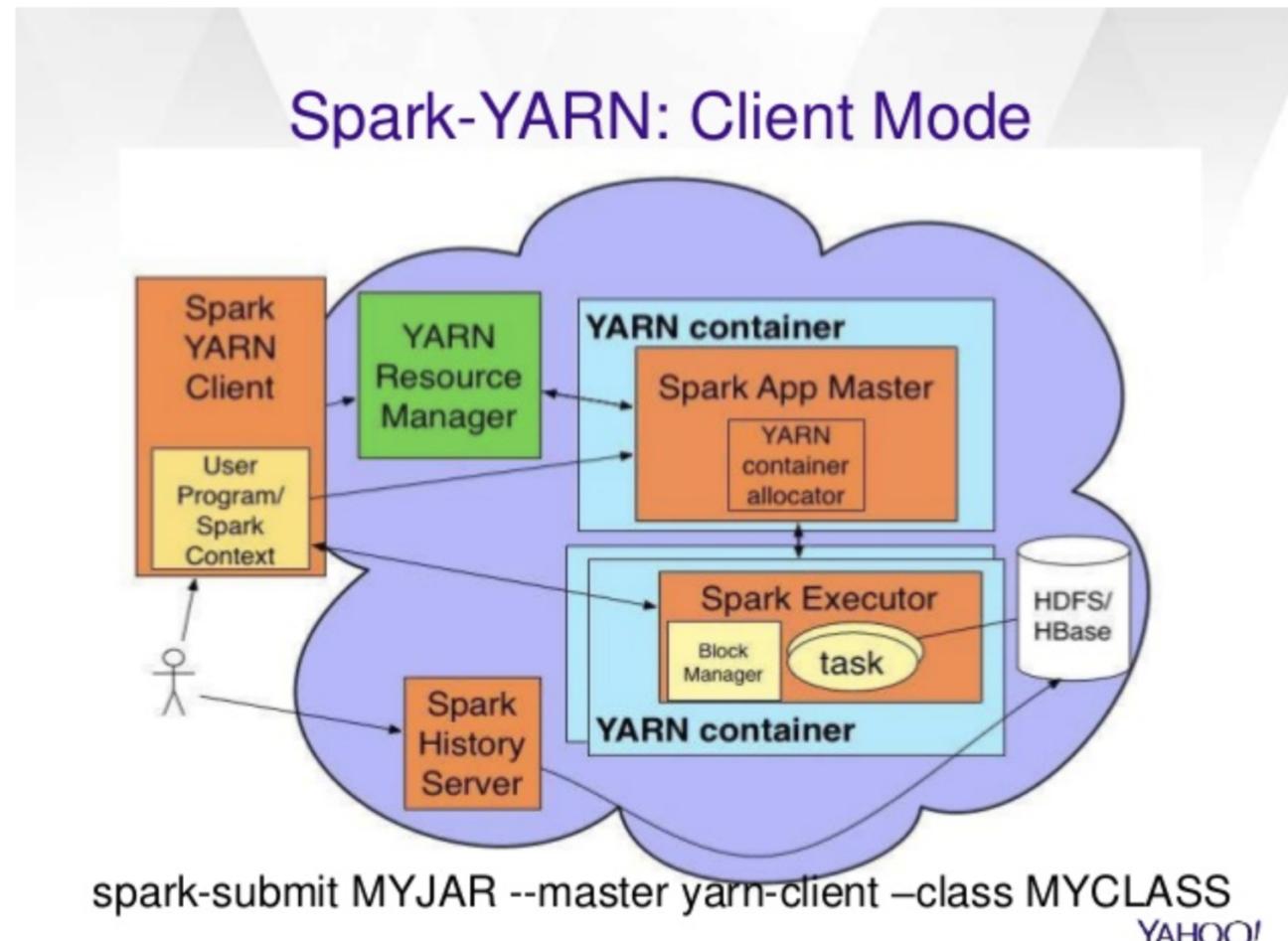


These modes are different from full-distributed mode and pseudo-distributed mode

[TOP](#)

Spark Architecture

Spark on YARN: Client-Mode



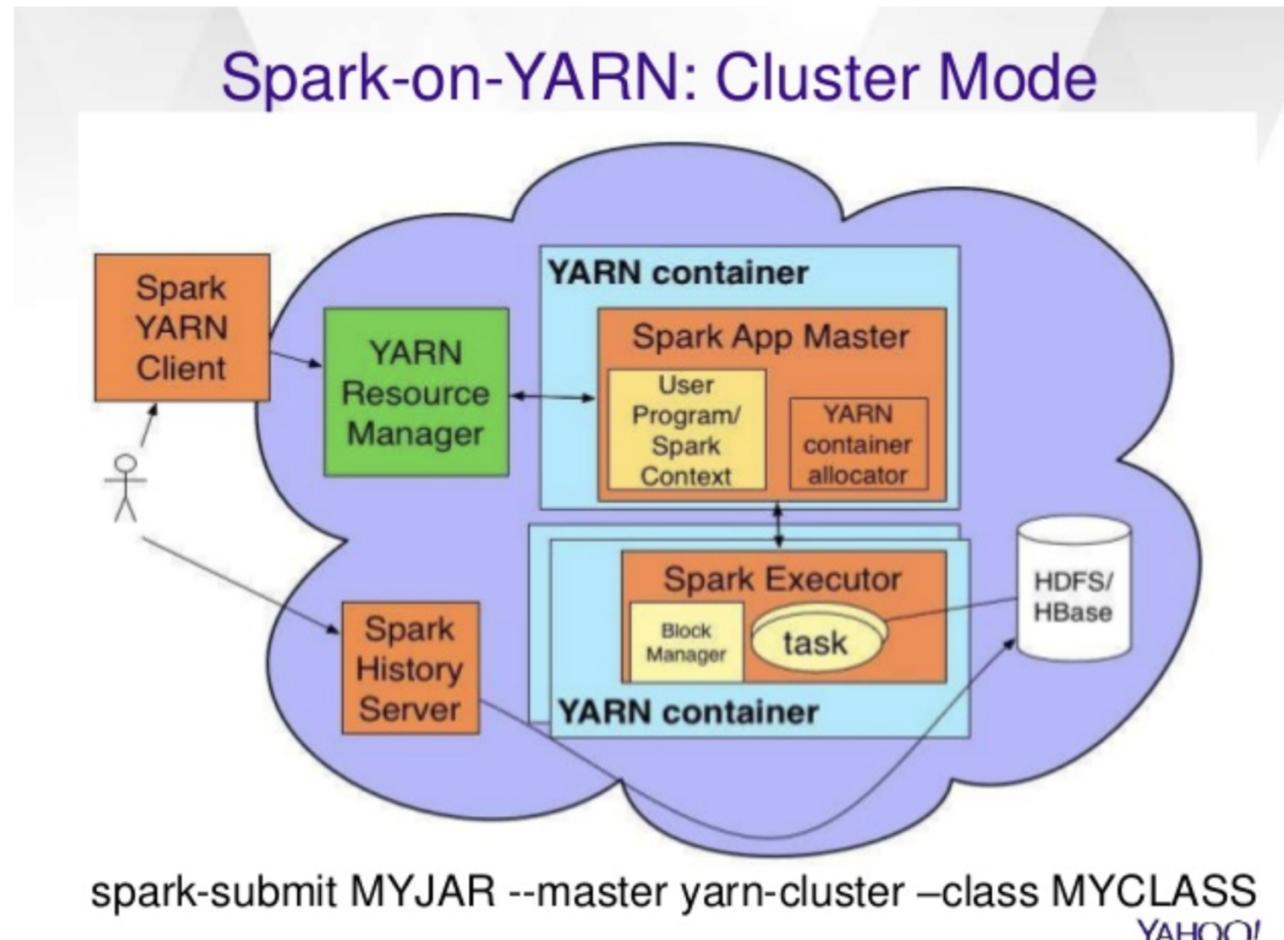
[TOP](#)

ISIT312 Big Data Management, SIM, Session 2, 2021

14/25

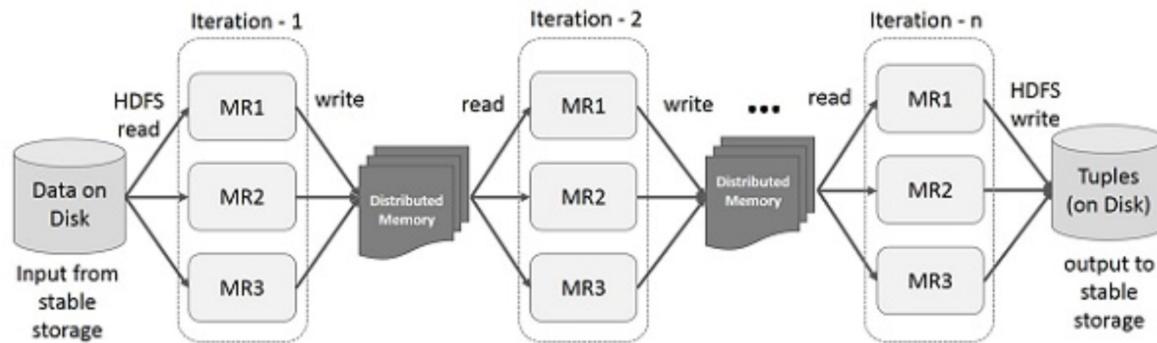
Spark Architecture

Spark on YARN: Cluster-Mode

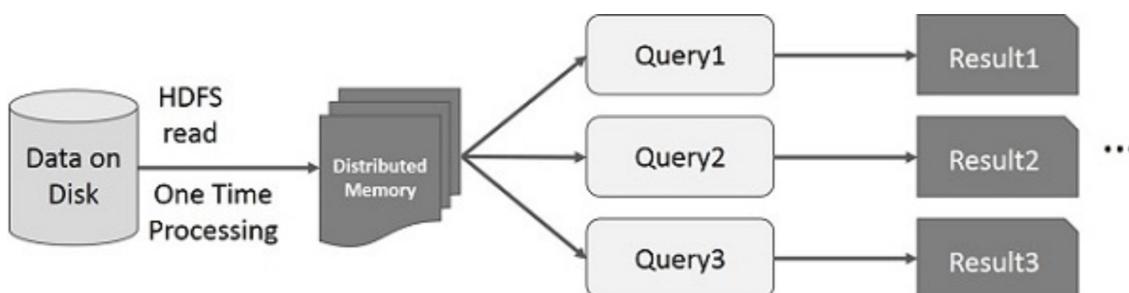


Spark Architecture

Writing to and reading from memory in Spark reduces I/O overhead



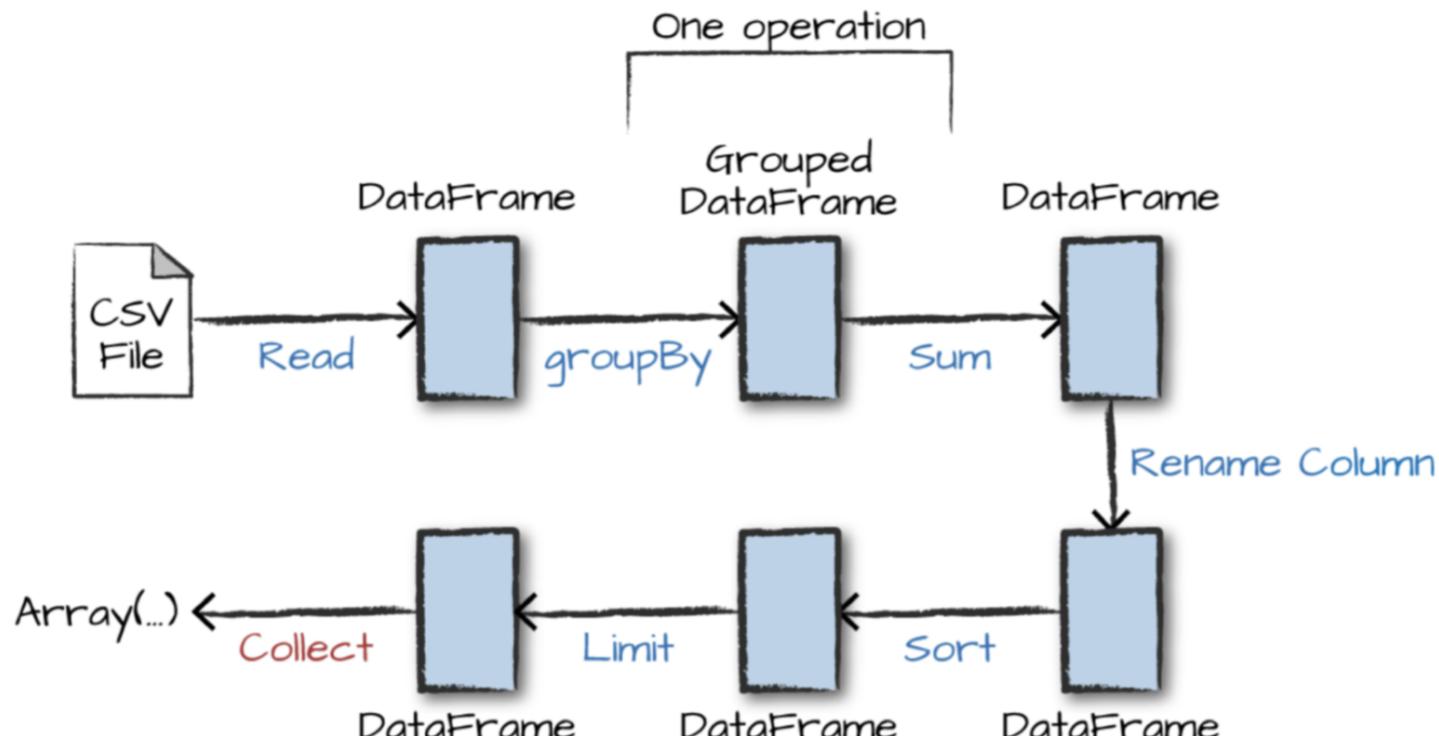
Data persists in memory and enables fast access



Spark Architecture

High-level API of **Spark** eases the development of a data processing pipeline

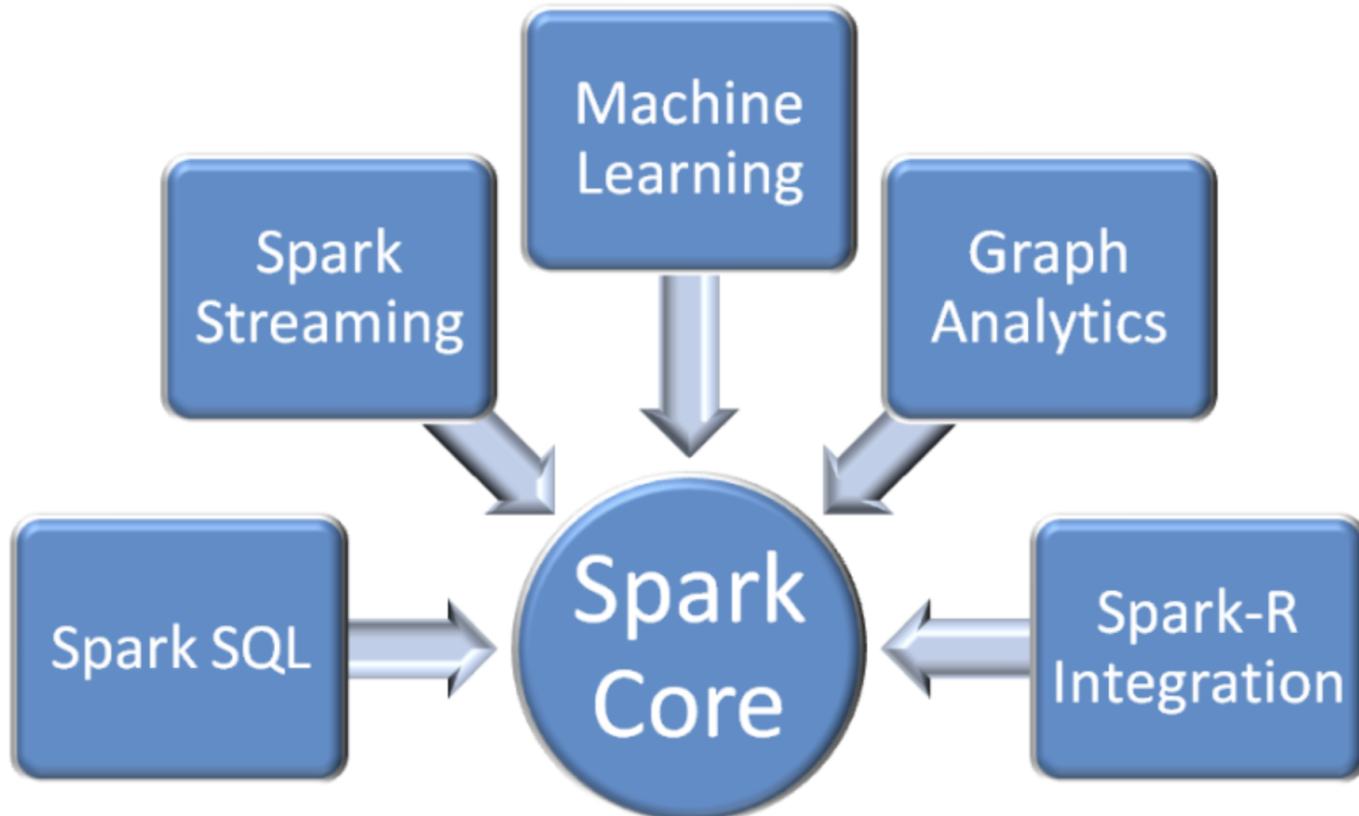
- No need to conform to a specific pattern, for example MapReduce

[TOP](#)

- Under the hood, **Spark** determines the best logical plan and translates it to a physical plan

Spark Architecture

Components of Spark



Introduction to Spark

Outline

[MapReduce Challenges](#)

[Meet Spark !](#)

[Features of Spark](#)

[Spark Architecture](#)

[Overview of Spark Components](#)

[Spark Glossary](#)

Overview of Spark Components

Spark Core

- **Spark Core** is the underlying general execution engine for **Spark** platform that all other functionality is built upon
- It provides **In-Memory** computing and referencing datasets in external storage systems

Spark SQL

- **Spark SQL** is a component on top of **Spark Core** that introduces a new data abstraction called **SchemaRDD**, which provides support for structured and semi-structured data

Spark Streaming

- **Spark Streaming** leverages **Spark Core** fast scheduling capability to perform streaming analytics

Overview of Spark Components

SparkML

- **SparkML** is a distributed machine learning framework above **Spark** because of the distributed memory-based **Spark** architecture

GraphX

- **GraphX** is a distributed graph-processing framework on top of Spark
- It provides an API for expressing graph computation and provides an optimized runtime for this abstraction

SparkR

- **SparkR** is an **R** package that provides a light-weight frontend to use **Spark** from **R**

Introduction to Spark

Outline

[MapReduce Challenges](#)

[Meet Spark !](#)

[Features of Spark](#)

[Spark Architecture](#)

[Overview of Spark Components](#)

[Spark Glossary](#)

Spark Glossary

Term	Meaning
Application	User program built on Spark. Consists of a <i>driver program</i> and <i>executors</i> on the cluster.
Application jar	A jar containing the user's Spark application. Usually jar not include Hadoop/Spark libraries, which are added at runtime.
Driver program	The process running the main() function of the application and creating the SparkContext
Cluster manager	An external service for acquiring resources on the cluster (e.g. standalone manager, YARN, Mesos)
Worker node	Any node that can run application code in the cluster

[TOP](#)

ISIT312 Big Data Management, SIM, Session 2, 2021

23/25

Spark Glossary

Term	Meaning
Deploy mode	Distinguishes where the driver process runs. In "cluster" mode, the framework launches the driver inside of the cluster. In "client" mode, the submitter launches the driver outside of the cluster.
Executor	A process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage across them. Each application has its own executors.
Task	A unit of work that will be sent to one executor
Job	A parallel computation consisting of multiple tasks that gets spawned in response to a Spark <i>action</i> (e.g. save, collect)
Stage	A subset of tasks in a job that depend on each other (similar to the map and reduce stages in MapReduce)

References

A Gentle Introduction to Spark, Databricks, (Available in **READINGS** folder)

Spark Overview

Karau H., Fast data processing with Spark Packt Publishing, 2013
(Available from UOW Library)

Srinivasa, K.G., Guide to High Performance Distributed Computing: Case Studies with Hadoop, Scalding and Spark Springer, 2015 (Available from UOW Library)

Chambers B., Zaharia M., Spark: The Definitive Guide, O'Reilly 2017

ISIT312 Big Data Management

Spark Data Model

Dr Guoxin Su and Dr Janusz R. Getta

School of Computing and Information Technology -
University of Wollongong

Introduction to Spark

Outline

Resilient Distributed Data Sets (RDDs)

DataFrames

SQL Tables/View

Datasets

Resilient Distributed Data Sets (RDDs)

Resilient Distributed Datasets (RDDs) is the lowest level (and the oldest) data abstraction available to the users

An RDD represents an immutable, partitioned collection of elements that can be operated on in parallel

Every row in an RDD is a Java object

RDD does not need to have any schema defined in advance

It makes RDD very flexible for various applications but in the same moment makes the manipulations on data more complicated

Users must implement their own functions to perform simple tasks like for example aggregation functions: average, count, maximum, etc

RDDs provide more control on how data is distributed over a cluster and how it is operated

Resilient Distributed Data Sets (RDDs)

RDD is characterized by the following properties

- A list of partitions
- A function for computing each split
- A list of dependencies on other RDDs
- Optionally, a Partitioner for **key-value RDDs**
- Optionally, a list of preferred locations to compute each split

Specification of custom partitions may provide significant performance improvements when using **key-value RDDs**

The properties of **RDDs** determine how Spark schedules and processes the applications

Different **RDDs** may use different properties depending on the required outcomes

Resilient Distributed Data Sets (RDDs)

RDD can be created from a list

```
val hello = "Hello World !".split(" ")
```

Creating RDD

It is possible to determine the total number of partitions to distribute an array

```
val words = spark.sparkContext.parallelize(hello, 2)
```

Distributing over partitions

Listing array of strings

```
words.collect()
```

Listing RDD

Creating RDD from a file by reading line-by-line

```
val lines = spark.sparkContext.textFile("/bigdata/path/lines.txt")
```

Creating RDD

Introduction to Spark

Outline

[Resilient Distributed Data Sets \(RDDs\)](#)

[DataFrames](#)

[SQL Tables/Views](#)

[Datasets](#)

DataFrames

A **DataFrame** is a table of data with rows and columns

A list of columns in **DataFrame** together with the types of columns is called as a schema

A **DataFrame** can span over many systems in a cluster

The concepts similar to **DataFrame** has been used in **Python** and **R**

A **DataFrame** is the simplest data model available in Spark

A **DataFrame** consist of zero or more **partitions**

- To parallelize data processing all data are divided into chunks called as **partitions**
- A **partition** is a collection of rows located on a single system in a cluster
- When operating **DataFrame** Spark simultaneously processes all relevant **partitions**
- **Shuffle operation** is performed to share data among the systems in a cluster

DataFrames

A **DataFrame** can be created in the following way

Creating a DataFrame

```
val dataFrame = spark.read.json("/bigdata/people.json")
```

The contents of a DataFrame can be listed in the following way

Listing results

```
dataFrame.show()
```

age	name
null	Michael
30	Andy
19	Justin

DataFrames

A schema of a **DataFrame** can be listed in the following way

Listing a schema

```
dataFrame.printSchema()  
  
root  
|--- age: long (nullable = true)  
|--- name: string (nullable = true)
```

Introduction to Spark

Outline

[Resilient Distributed Data Sets \(RDDs\)](#)

[DataFrames](#)

[SQL Tables/Views](#)

[Datasets](#)

SQL Tables/Views

SQL can be used to operate on DataFrames

It is possible to register any DataFrame as a table or view (a temporary table) and apply SQL to it

There is no performance overhead when registering a DataFrame as **SQL Table** and when processing it

A DataFrame `dataFrame` can be registered as **SQL temporary view people** in the following way

```
dataFrame.createOrReplaceTempView("people")
val sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
```

DataFrame as SQL Table

age	name
null	Michael
30	Andy
19	Justin

Results

TOP

SQL Tables/Views

SQL Tables are logically equivalent to **DataFrames**

A difference between **SQL Tables** and **DataFrames** is such that **DataFrames** are defined within the scope of a programming language while **SQL Tables** are defined within a database

When created, **SQL table** belongs to a **default database**

SQL table can be created in the following way

```
CREATE TABLE flights(  
    DEST_COUNTRY_NAME STRING,  
    ORIGIN_COUNTRY_NAME STRING, count LONG)  
    USING JSON OPTIONS ( path '/mnt/defg/chapter-1-data/json/2015-summary.json' )
```

CREATE TABLE

SQL view can be created in the following way

```
CREATE VIEW just_usa_view AS  
    SELECT *  
    FROM flights  
    WHERE dest_country_name = 'United States'
```

CREATE VIEW

SQL Tables/Views

Global SQL view can be created in the following way

CREATE VIEW

```
CREATE GLOBAL VIEW just_usa_global AS  
SELECT *  
FROM flights  
WHERE dest_country_name = 'United States'
```

Temporary SQL view can be created in the following way

CREATE VIEW

```
CREATE TEMP VIEW just_usa_global AS  
SELECT *  
FROM flights  
WHERE dest_country_name = 'United States'
```

Global and Temporary SQL view can be created in the following way

CREATE VIEW

```
CREATE GLOBAL TEMP VIEW just_usa_global AS  
SELECT *  
FROM flights  
WHERE dest_country_name = 'United States'
```

TOP

SQL Tables/Views

Spark SQL view includes three core complex types: **sets**, **lists**, and **structs**

Structs allow to create nested data

CREATE VIEW

```
CREATE VIEW nested_data AS
  SELECT (DEST_COUNTRY_NAME, ORIGIN_COUNTRY_NAME) as country, count
  FROM flights
```

The functions **collect_set** and **collect_list** functions create **sets** and **lists**

CREATE VIEW

```
SELECT DEST_COUNTRY_NAME as new_name,
       collect_list(count) as flight_counts,
       collect_set(ORIGIN_COUNTRY_NAME) as origin_set
  FROM flights
 GROUP BY DEST_COUNTRY_NAME
```

SQL Tables/Views

Managed versus unmanaged tables

- Managed table is a table that stored data and metadata, it is equivalent to an internal table in Hive
- Unmanaged table is a table that stores only data, it is equivalent to an external table in Hive

Creating unmanaged (external) table in Spark

```
CREATE EXTERNAL TABLE hive_flights(  
  DEST_COUNTRY_NAME STRING,  
  ORIGIN_COUNTRY_NAME STRING,  
  count LONG)  
  ROW FORMAT DELIMITED FIELDS TERMINATED BY ','  
  LOCATION '/mnt/defg/flight-data-hive/'
```

CREATE EXTERNAL TABLE

Introduction to Spark

Outline

[Resilient Distributed Data Sets \(RDDs\)](#)

[DataFrames](#)

[SQL Tables/Views](#)

[Datasets](#)

Datasets

A **Dataset** is a distributed collection of data

A **DataFrame**, is a **Dataset** of type **Row**

Datasets consists of the objects included in the rows; one object per row

Datasets are a strictly JVM language feature that only work with **Scala** and **Java**

A **Dataset** can be constructed from JVM and then manipulated using functional transformations

In **Scala** a **case class** object defines a schema of an object

Datasets use a specialized **Encoder** to serialize the objects for processing or transmitting over the network

Dataset supports all operations of **DataFrame**

Datasets

A **Dataset** can be defined, created, and used in the following way

```
case class Person(name: String, age: Long)
```

Creating Dataset Schema

```
val caseClassDS = Seq(Person("Andy", 32)).toDS()
```

Creating Dataset

```
caseClassDS.show()
```

Listing Dataset

```
+----+----+
|name|age|
+----+----+
|Andy| 32|
+----+----+
```

Results

```
caseClassDS.select($"name").show()
```

Using a Dataset

```
+----+
| name|
+----+
|James|
+----+
```

Results

[TOP](#)

References

A Gentle Introduction to Spark, Databricks, (Available in **READINGS** folder)

[RDD Programming Guide](#)

[Spark SQL, DataFrames and Datasets Guide](#)

Karau H., Fast data processing with Spark Packt Publishing, 2013
(Available from UOW Library)

Srinivasa, K.G., Guide to High Performance Distributed Computing: Case Studies with Hadoop, Scalding and Spark Springer, 2015 (Available from UOW Library)

Chambers B., Zaharia M., Spark: The Definitive Guide, O'Reilly 2017

ISIT312 Big Data Management

Spark Operations

Dr Guoxin Su and Dr Janusz R. Getta

School of Computing and Information Technology -
University of Wollongong

Spark Operations

Outline

The Programming Language Scala

Quick Start

Self Contained Application

Web User Interface

Operations on Resilient Distributed Datasets (RDDs)

Operations on Datasets

Operations on DataFrames

SQL Module

The Programming Language Scala

Spark has built-in APIs for Java, Scala, and Python, and is also integrated with R

Among all languages, Scala is the most supported language

Also, Spark project is implemented using Scala

Therefore, we choose Scala as our working language in Spark

Scala is a Java-like programming language which unifies object-oriented and functional programming

Scala is a pure object-oriented language in the sense that every value is an object

Types and behaviour of objects are described by classes

Scala is a functional programming language in the sense that every function is a value

Nesting of function definitions and higher-order functions are naturally supported

[TOP](#)

ISIT312 Big Data Management, SIM, Session 2, 2021

3/42

The Programming Language Scala

Hello World ! in Scala

```
object Hello {  
    def main(args: Array[String]) = {  
        println("Hello, world")  
    }  
}
```

Hello World ! in Scala

Instead of including `main` method, it can be extended with `App` trait

```
object Hello2 extends App {  
    println("Hello, world")  
}
```

Extending App trait

Using command line arguments

```
object HelloYou extends App {  
    if (args.size == 0)  
        println("Hello, you")  
    else  
        println("Hello, " + args(0))  
}
```

Command line arguments

[TOP](#)

The Programming Language Scala

Difference between **var**, **val**, and **def**

```
var x = 7  
x = x * 2
```

Variable

```
val x = 7  
x = x * 2  
'error: reassignment to val'
```

Value

```
def hello(name: String) = "Hello : " + name  
hello("James") // "Hello : James"  
hello("") // "Hello : "
```

Function declaration

When **lazy** keyword is used then a value is only computed when it is needed

```
lazy val x = {  
    println("calculating value of x")  
    13 }  
val y = {  
    println("calculating value of y")  
    20 }
```

Lazy evaluation

[TOP](#)

The Programming Language Scala

Defining a class

```
class Point(var x: Int, var y: Int) {  
  
    def move(dx: Int, dy: Int): Unit = {  
        x = x + dx  
        y = y + dy  
    }  
  
    override def toString: String =  
        s"($x, $y)"  
}  
  
val point1 = new Point(2, 3)  
println(point1.x)          // 2  
println(point1)            // prints (2, 3)
```

Class Point

Method move

Method toString

Applications

Spark Operations

Outline

[The Programming Language Scala](#)

[Quick Start](#)

[Self Contained Application](#)

[Web User Interface](#)

[Operations on Resilient Distributed Datasets \(RDDs\)](#)

[Operations on Datasets](#)

[Operations on DataFrames](#)

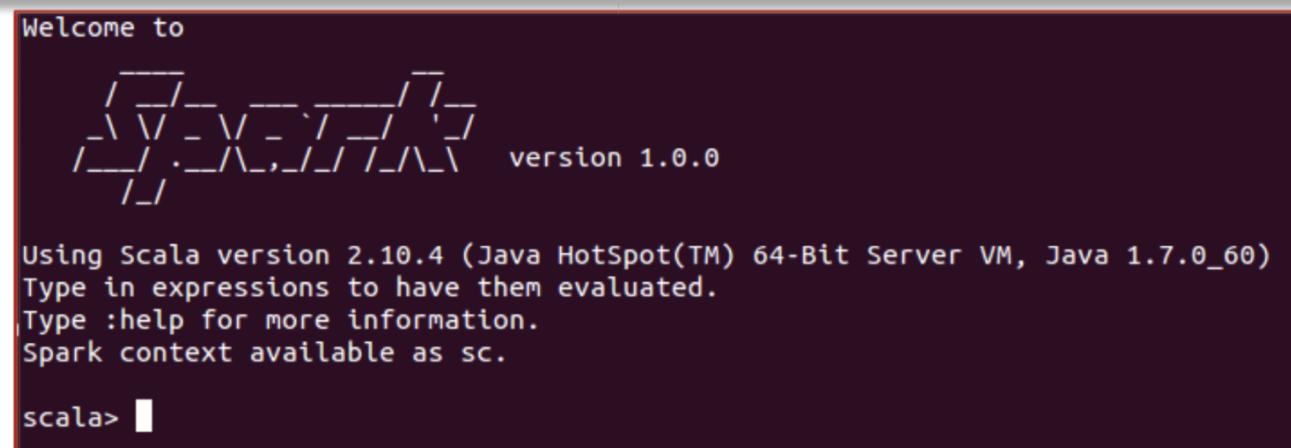
[SQL Module](#)

Quick Start

To open **Scala** version of **Spark shell** in standalone mode process the following command

```
bin/spark-shell --master local[*]
```

Starting Spark shell in standalone mode



```
Welcome to
    _/ \_ _\ \ / \ _/ \
   / \ / . \ \ / \ / \ / \ \ / \
  / \ / \ / \ / \ / \ / \ / \ / \
  / \ / \ / \ / \ / \ / \ / \ / \
version 1.0.0

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_60)
Type in expressions to have them evaluated.
Type :help for more information.
Spark context available as sc.

scala>
```

To open **Spark shell** shell with **YARN**, process the following command

```
bin/spark-shell --master yarn
```

Starting Spark shell with Yarn

Quick Start

A **SparkSession** instance is an entry to a **Spark** application

- If you type **spark** in the spark-shell interface then you get the following messages

Spark messages

```
res0: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@...
```

You can use **SparkSession** instance **spark** to interact with **Spark** and to develop your data processing pipeline

For example,

Interacting with Spark

```
// in Scala
val myRange = spark.range(1000).toDF("number")
```

Quick Start

Sample processing of a file **README.md**

```
val YOUR_SPARK_HOME ="path-to-your-Spark-home"                                Setting Spark Home folder

val textFile = spark.read.textFile("$YOUR_SPARK_HOME/README.md")                  Reading a text file

textFile.count()                                                                Counting rows
// res0: Long = 32

textFile.first()                                                               Reading the first row
// res1: String = # Apache Spark

textFile.filter(line => line.contains("Spark")).count()                         Filtering and counting rows
// How many lines contain "Spark"?
// res2: Long = 11
```

Quick Start

More operations on a file

Import Spark modules

```
import spark.implicits._
```

Counting number of words in the longest line

```
textFile.map(line => line.split(" ")).size().reduce((a, b) => if (a > b) a else b)  
// res3: Int = 66
```

Filtering and counting rows

```
val wordCounts = textFile.flatMap(line => line.split(" ")).  
groupByKey(identity).count()  
// wordCounts: org.apache.spark.sql.Dataset[(String, Long)]  
// = [value: string, count(1): bigint]
```

Quick Start

```
wordCounts.show(3)
+-----+-----+
|    value|count(1)|
+-----+-----+
|    some|      2|
|  graphs|      1|
| Building|      1|
+-----+
only showing top 3 rows
```

Filtering and counting rows

```
wordCounts.collect()
// res6: Array[(String, Long)] = Array((some,2),
// (graphs,1), ("Building",1), ...)
```

Filtering and counting rows

Spark Operations

Outline

The Programming Language Scala

Quick Start

Self Contained Application

Web User Interface

Operations on Resilient Distributed Datasets (RDDs)

Operations on Datasets

Operations on DataFrames

Spark SQL Module

Self-Contained Application

A sample self-contained application

SimpleApp.scala

```
import org.apache.spark.sql.SparkSession
object SimpleApp {
    def main(args: Array[String]) {
        val logFile = "YOUR_SPARK_HOME/README.md"
        // Should be some file on your system
        val spark = SparkSession.builder
            .appName("Simple Application")
            .config("spark.master", "local[*]")
            .getOrCreate()
        val logData = spark.read.textFile(logFile).cache()
        val numAs = logData.filter(line => line.contains("a")).count()
        val numBs = logData.filter(line => line.contains("b")).count()
        println(s"Lines with a: $numAs, Lines with b: $numBs")
        spark.stop()
    }
}
```

Self-Contained Application

Compiling **Scala** source code using **scalac**

Compiling Scala source code

```
scalac -classpath "$SPARK_HOME/jars/*" SimpleApp.scala
```

Creating a jar file in the following way

Creating jar

```
jar cvf app.jar SimpleApp*.class
```

Process it with Spark-shell in the following way

Processing

```
$SPARK_HOME/bin/spark-submit --master local[*] --class SimpleApp app.jar
```

Spark Operations

Outline

The Programming Language Scala

Quick Start

Self Contained Application

Web User Interface

Operations on Resilient Distributed Datasets (RDDs)

Operations on Datasets

Operations on DataFrames

Spark SQL Module

Web UI

Each driver program has a Web UI, typically on port **4040**

Spark Web UI displays information about running tasks, executors, and storage usage.

The screenshot shows the Apache Spark 2.1.0-SNAPSHOT Web UI interface. At the top, there is a navigation bar with tabs: Jobs (selected), Stages, Storage, Environment, Executors, SQL, and a link to the Spark shell application UI. Below the navigation bar, the main content area is titled "Spark Jobs (?)". It displays the following statistics:

- User: jacek
- Total Uptime: 35 s
- Scheduling Mode: FIFO
- Active Jobs: 1
- Completed Jobs: 1
- Failed Jobs: 1

Below these statistics is a link to "Event Timeline". The "Active Jobs (1)" section contains a table with one row, showing a job with Job ID 2, Description "show at <console>:24", Submitted on 2016/09/29 14:01:20, Duration 5 s, Stages Succeeded/Total 0/1, and Tasks (for all stages) Succeeded/Total 0/1. The "Completed Jobs (1)" section contains a table with one row, showing a job with Job ID 0, Description "show at <console>:24", Submitted on 2016/09/29 14:01:07, Duration 0.3 s, Stages Succeeded/Total 1/1, and Tasks (for all stages) Succeeded/Total 1/1. The "Failed Jobs (1)" section contains a table with one row, showing a job with Job ID 1, Description "show at <console>:24", Submitted on 2016/09/29 14:01:14, Duration 87 ms, Stages Succeeded/Total 0/1 (1 failed), and Tasks (for all stages) Succeeded/Total 0/1 (1 failed).

Spark Operations

Outline

The Programming Language Scala

Quick Start

Self Contained Application

Web User Interface

Operations on Resilient Distributed Datasets (RDDs)

Operations on Datasets

Operations on DataFrames

Spark SQL Module

Operations on Resilient Distributed Datasets (RDDs)

Operations on **RDDs** are performed on raw **Java** or **Scala** objects

Creating a simple **RDD** with words and distributing over 2 partitions

Creating RDD

```
val myCollection = "Spark The Definitive Guide : Big Data Processing Made Simple".split(" ")
val words = spark.sparkContext.parallelize(myCollection, 2)
```

Eliminating duplicates and counting words

Distinct and counting

```
words.distinct().count()
```

Filtering

Filtering function

```
def startsWithS(individual:String) = { individual.startsWith("S") }
```

Filtering

```
val onlyS = words.filter(word => startsWithS(word))
```

Results of filtering

```
onlyS.collect()
```

Operations on Resilient Distributed Datasets (RDDs)

Sorting of **RDD** uses **sortBy** method and a function that extracts a value from the objects

Sorting

```
words.sortBy(word => word.length() * -1).take(2)
```

Random split into Array

Split into Array

```
val fiftyFiftySplit = words.randomSplit(Array[Double](0.5, 0.5))
```

Reduce **RDD** to one value

Reducing

```
def wordLengthReducer(leftWord:String, rightWord:String): String = {
  if (leftWord.length >= rightWord.length)
    return leftWord
  else
    return rightWord }
```

Reducing

```
words.reduce(wordLengthReducer)
```

Operations on Resilient Distributed Datasets (RDDs)

Some operations on **RDDs** are available on key-value pairs

The most common ones are distributed "shuffle" operations, such as grouping or aggregating the elements by a key

For example, **reduceByKey** operation on key-value pairs can be used to count how many times each line of text occurs in a file

```
val lines = sc.textFile("data.txt")
val pairs = lines.map(s => (s, 1))
val counts = pairs.reduceByKey((a, b) => a + b)
```

Sorting

Some of the transformations of **RDDs**

map(func):	passes each element of RDD through a function	Transformations
filter(func):	selects all elements for which a function returns true	
sample(withReplacement, fraction, seed):	extracts sample from RDD	
union(otherDataset):	unions two RDDs	
intersection(otherDataset):	finds intersection of two RDDs	
distinct([numPartitions]):	eliminates duplicates	
groupByKey([numPartitions]):	when called on RDD with (K, V) pairs, returns RDD with (K, Iterable) pairs	
sortByKey([ascending], [numPartitions]):	when called on (K, V) pairs where K implements Ordered, returns (K, V) pairs sorted by keys	ISIT312 Big Data Management, SIM, Session 2, 2021 21/42

[TOP](#)

Spark Operations

Outline

The Programming Language Scala

Quick Start

Self Contained Application

Web User Interface

Operations on Resilient Distributed Datasets (RDDs)

Operations on Datasets

Operations on DataFrames

Spark SQL Module

Operations on Datasets

Operations on a **Dataset** start from creation of **case class**

```
case class Person(name: String, age: Long)
```

Creating case class

```
val caseClassDS = Seq(Person("Andy", 32)).toDS()
```

Creating Dataset

```
caseClassDS.show()
```

Listing Dataset

```
+---+---+
| name | age |
+---+---+
| Andy | 32 |
+---+---+
```

Results

Dataset supports all operations of **DataFrame**

```
caseClassDS.select($"name").show()
```

Using a Dataset

Operations on Datasets

Operations on **Datasets** start from creation of **case class**

Creating case class

```
case class Flight(DEST_COUNTRY_NAME: String, ORIGIN_COUNTRY_NAME: String, count: BigInt)
```

Next we create a **DataFrame**

Creating DataFrame

```
val flightsDF = spark.read.parquet("/mnt/defg/chapter-1-data/parquet/2010-summary.parquet/")
```

Finally, **DataFrame** is casted to **Dataset**

Creating Dataset

```
val flights = flightsDF.as[Flight]
```

Filtering a **Dataset**

Defining a function

```
def originIsDestination(flight_row: Flight): Boolean = {
  return flight_row.ORIGIN_COUNTRY_NAME == flight_row.DEST_COUNTRY_NAME}
```

```
flights.filter(flight_row => originIsDestination(flight_row)).first()
```

Filtering

Mapping a **Dataset**

Mapping

```
val destinations = flights.map(f => f.DEST_COUNTRY_NAME)
```

Spark Operations

Outline

The Programming Language Scala

Quick Start

Self Contained Application

Web User Interface

Operations on Resilient Distributed Datasets (RDDs)

Operations on Datasets

Operations on DataFrames

Spark SQL Module

Operations on DataFrames

Dataset and DataFrame are the data abstractions for Spark SQL

Dataset is a distributed collection of data

- It supports the use of self-defined functions to process data
- For example, map and reduce functions in the previous slides
- Dataset is typed; typing is checked at compiling time

DataFrame is a Dataset organized into named columns.

- It is conceptually equivalent to a table in a relational database or a data frame in R/Python
- To use self-defined functions, you need to register them with Spark
- DataFrame is untyped, i.e., typing is checked at runtime
- DataFrame is more performance-optimal than Dataset

Operations on DataFrames

DataFrame can be created in the following way

```
val df = spark.read.json("people.json")
df.show()
```

Creating a DataFrame

age	name
null	Michael
30	Andy
19	Justin

Results

```
df.printSchema()
```

Results

```
root
|-- age: long (nullable = true)
|-- name: string (nullable = true)
```

Results

[TOP](#)

Operations on DataFrames

Select on a DataFrame

```
df.select($"name", $"age" + 1).show()
```

Selecting from a DataFrame

```
+-----+  
| name |(age + 1)|  
+-----+  
| Michael|      null|  
| Andy|      31|  
| Justin|      20|  
+-----+
```

Results

```
df.filter($"age" > 21).show()
```

Filtering a DataFrame

```
+---+---+  
| age|name|  
+---+---+  
| 30|Andy|  
+---+---+
```

Results

[TOP](#)

Operations on DataFrames

Count people by age

```
df.groupBy("age").count().show()
```

Counting in a DataFrame

Results

age	count
19	1
null	1
30	1

Operations on DataFrames

Register a **DataFrame** as **SQL** temporary view

Registering and selecting from DataFrame

```
df.createOrReplaceTempView("people")
val sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
```

Results

age	name
null	Michael
30	Andy
19	Justin

Operations on DataFrames

When to use **DataFrames** ?

Except for the following few cases, you can use them interchangeable (if performance is not a concern). You also can convert one to the other easily.

- In the Bigdata pipeline, you read an unstructured data source, for example, a text file as a **Dataset** and continue processing the data
- You can directly read an structured source like Hive table, JSON document as a **DataFrame**
- If you expect to use self-defined function easily, especially in the data cleaning or preprocessing stage of the pipeline, you should use a Dataset

Operations on DataFrames

Create a **Dataset** of **Person** objects from a text file and convert it to a **DataFrame**

Converting a Dataset to DataFrame

```
val peopleDF = spark.sparkContext  
  .textFile("examples/src/main/resources/people.txt")  
  .map(_.split(","))  
  .map(attributes => Person(attributes(0), attributes(1).trim.toInt))  
  .toDF()
```

Results

```
peopleDF: org.apache.spark.sql.DataFrame = [name: string, age: bigint]
```

Operations on DataFrames

Convert DataFrame to Dataset

```
case class Employee(name: String, salary: Long)
val ds =
    spark.read.json(".../examples/src/main/resources/employees.json").as[Employee]
```

Converting a Dataset to DataFrame

```
ds: org.apache.spark.sql.Dataset[Employee] = [name: string, salary: bigint]
```

Results

Operations on DataFrames

Spark DataFrame/Dataset support two types of operations:
transformations and actions

Transformations are operations on **DataFrames/Datasets** that return a new **DataFrame/Dataset**

- For example `select()`, `groupBy()`, `map()`, and `filter()`

Actions are operations that return a result to the driver program or write it to storage, and kick off a computation

- For example `show()`, `count()`, and `first()`

Return type difference: **transformations** return **DataFrames/Datasets**, whereas **actions** return some other data type

Spark treats the two operations very differently

Operations on DataFrames

Transformations are **lazily evaluated**, meaning that **Spark** will not begin to execute until it sees an action

Instead, **Spark** internally records metadata to indicate that some transformation operation has been requested

For example **transformation** creates another **DataFrame**

```
val sqlDF = spark.sql("SELECT * FROM people")
```

Creating a DataFrame

Action triggers the computation

```
sqlDF.show()
```

Action on a DataFrame

Operations on DataFrames

The **lazy evaluation** to reduce the number of passes it has to take over the dataset

In **Hadoop MapReduce**, developers often have to consider how to group together operations to minimize the number of MapReduce passes

In Spark, there is no substantial benefit to writing a single complex map instead of chaining together many simple operations

Thus, users are free to organize their program into smaller, more manageable operations

Spark Operations

Outline

The Programming Language Scala

Quick Start

Self Contained Application

Web User Interface

Operations on Resilient Distributed Datasets (RDDs)

Operations on Datasets

Operations on DataFrames

Spark SQL

Spark SQL

Spark SQL is a Spark module for general data processing and analytics

It can be used for all sorts of data, from unstructured log files to semi-structured CSV files and highly structured Parquet files

To interact with Spark SQL, you can either use SQL or Spark Structured API, or both

The same execution engine is used, independent of which API/language you use to express the computation

The APIs of Spark SQL provide a rich set of pre-built, high-level operations for accomplishing sophisticated data processing and ETL jobs, and mechanism to implement your own operations, for example self-defined functions and aggregations

Spark SQL

Spark SQL has two data abstractions

- DataFrame
- Dataset (available in Scala/Java APIs, but not Python/R APIs)
- DataFrame can be represented as SQL tables and views

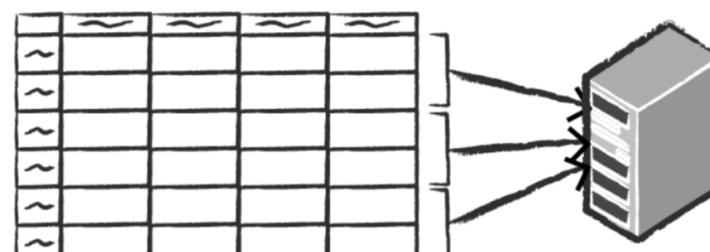
Both are distributed table-like collections with well-defined rows and columns.

- DataFrame vs. spreadsheet

Spreadsheet on
a single machine



Table or Data Frame
partitioned across servers
in a data center



Spark SQL

Spark SQL allows to code **SQL** statements in **Scala**, **Java** and **Python** language APIs.

To use **SQL** to manipulate a **DataFrame**, we first need to create a temporal view for it

```
df.createOrReplaceTempView("dfTable")
```

Creating a temporal view

All standard **SQL** statements + functions are applicable in **Spark SQL**

Spark implements a subset of [ANSI SQL:2003](#)

Spark SQL

Using SQL

```
spark.sql(  
    "SELECT DEST_COUNTRY_NAME, sum(count)  
     FROM dfTable  
    GROUP BY DEST_COUNTRY_NAME"  
)  
.where("DEST_COUNTRY_NAME like 'S%'''")  
.where("sum(count) > 10")  
.show(2)
```

Applying sql method

DEST_COUNTRY_NAME	sum(count)
Senegal	40
Sweden	118

Results

References

The Scala Programming Language

A Gentle Introduction to Spark, Databricks, (Available in **READINGS** folder)

RDD Programming Guide

Spark SQL, DataFrames and Datasets Guide

Karau H., Fast data processing with Spark Packt Publishing, 2013
(Available from UOW Library)

Srinivasa, K.G., Guide to High Performance Distributed Computing: Case Studies with Hadoop, Scalding and Spark Springer, 2015 (Available from UOW Library)

Chambers B., Zaharia M., Spark: The Definitive Guide, O'Reilly 2017

ISIT312 Big Data Management

Spark Stream Processing

Dr Guoxin Su and Dr Janusz R. Getta

School of Computing and Information Technology -
University of Wollongong

Spark Structured Data and Stream Processing

Outline

Spark Stream Processing Modules

Stream Processing

Structured Stream Processing

Programming Model

Spark Stream Processing Modules

Stream processing is a key requirement in many big data applications

As soon as an application computes something of value, for example, a report or a machine learning model, an organization may want to compute this result continuously in a production environment

This capability is lacked in **Hadoop MapReduce** framework due to slowness of hard-disk IO

In-memory computation implemented in **Spark** make stream processing possible

Spark Streaming based on its low-level API **Resilient Distributed Dataset** is available since Spark 1.2

Spark Structured Streaming based on the **Spark SQL** engine is available since Spark 2.1

Luckily, our VM has installation of Spark 2.1.1

Spark Structured Data and Stream Processing

Outline

Spark Stream Processing Modules

Stream Processing

Structured Stream Processing

Programming Model

Stream Processing

Stream processing is the act of continuously incorporating the new data in the stream to compute a result

Sample sources of streams

- Bank transactions
- Clicks on a website
- Sensor readings from IoT devices
- Scientific observations and experiments
- Manufacturing processes, and the others

Stream processing vs. batch processing

- Batch processing runs to a fixed set of data, but stream processing handles an unbounded set of data
- Batch processing has low timeliness requirement, but stream processing requires to work at near realtime

Stream Processing

Use cases of stream processing

- Notifications and alerting
- Real-time reporting
- Incremental ETL
- Update data to serve in real time
- Real-time decision making
- Online machine learning

Stream Processing

To see the challenges of stream processing, we consider the following example

Suppose we received the following data from a sensor

Sample data

```
{value: 1, time: "2017-04-07T00:00:00"}  
{value: 2, time: "2017-04-07T01:00:00"}  
{value: 5, time: "2017-04-07T02:00:00"}  
{value: 10, time: "2017-04-07T01:30:00"}  
{value: 7, time: "2017-04-07T03:00:00"}
```

What actions should be performed when receiving single values, say, 5 ?

How to react to a pattern, say, 2 -> 10 -> 5

What if data arrives out-of-order, for example, 10 before 5

Other issues: What if a machine in the system fails, losing some state?

What if the load is imbalanced? How can an application signal downstream consumers when analysis for some event is done, and so

Stream Processing

Main challenges of stream processing are the following

- Processing out-of-order data based on application timestamps (also called event time)
- Maintaining large amounts of states
- Supporting high-data throughput
- Processing each event exactly once despite machine failures
- Handling load imbalance and stragglers
- Responding to events at low latency
- Joining with external data in other storage systems
- Determining how to update output sinks as new events arrive
- Writing data transactionally to output systems
- Updating application business logic at runtime

Spark Structured Data and Stream Processing

Outline

Spark Stream Processing Modules

Stream Processing

Structured Stream Processing

Programming Model

Structured Processing

Structured Streaming Processing suppose to provide fast, scalable, fault-tolerant, end-to-end exactly-once stream processing without the user having to reason about streaming

A streaming version of the word-count example

```
Reading a stream
val lines = spark.readStream
    .format("socket")          // socket source
    .option("host", "localhost") // listen to the localhost
    .option("port", 9999)        // and port 9999
    .load()

Importing methods
import spark.implicits._

sql
val words = lines.as[String].flatMap(_.split(" "))

Grouping
val wordCounts = words.groupBy("value").count()

Writing stream
val query = wordCounts.writeStream
    .outputMode("complete") // accumulate the counting result
    .format("console")      // use the console as the sink
```

Structured Processing

The input is simulated by Netcat (a small utility found in most Unix-like systems) as a data server

Starting Netcat

```
nc -lk 9999
```

In a different Terminal, we start Spark-shell and input the Scala code from the previous slides

If we input in the first Terminal session

Starting Netcat

```
nc -lk 9999
apache spark
apache hadoop
...
```

Structured Processing

Then we should see the right hand-side output in Spark-shell

Batch: 0	
value	count
apache	1
spark	1

Batch: 1	
value	count
apache	2
spark	1
hadoop	1

...

Spark Structured Data and Stream Processing

Outline

Spark Stream Processing Modules

Stream Processing

Structured Stream Processing

Programming Model

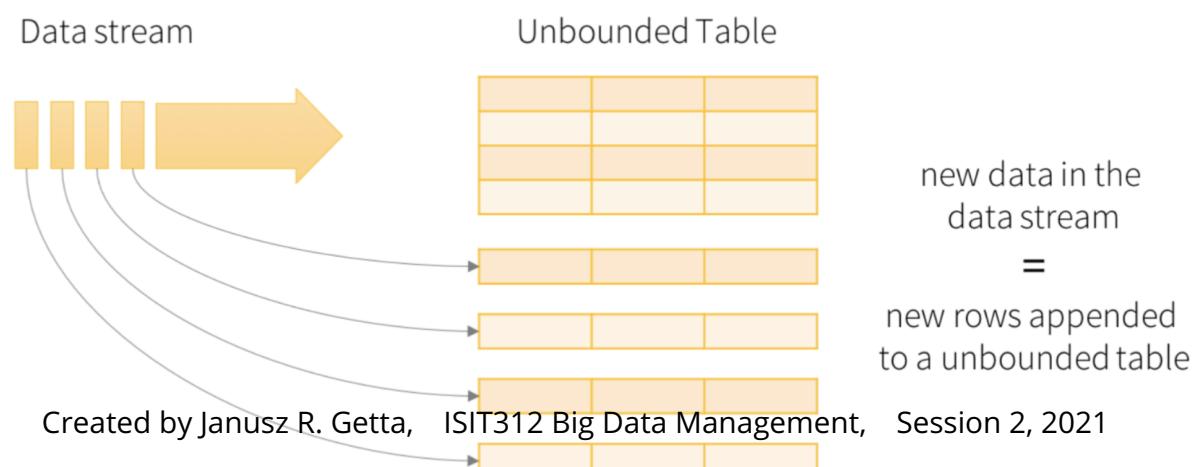
Programming Model

The key idea in **Structured Streaming** is to treat a live data stream as a table that is being continuously appended

This leads to a new stream processing model that is very similar to a batch processing model

Users can express the streaming computation as standard batch-like query as on a static table, and **Spark** runs it as an incremental query on the unbounded **Input Table**

A new data item arriving on the stream is like a new row being appended to **Input Table**



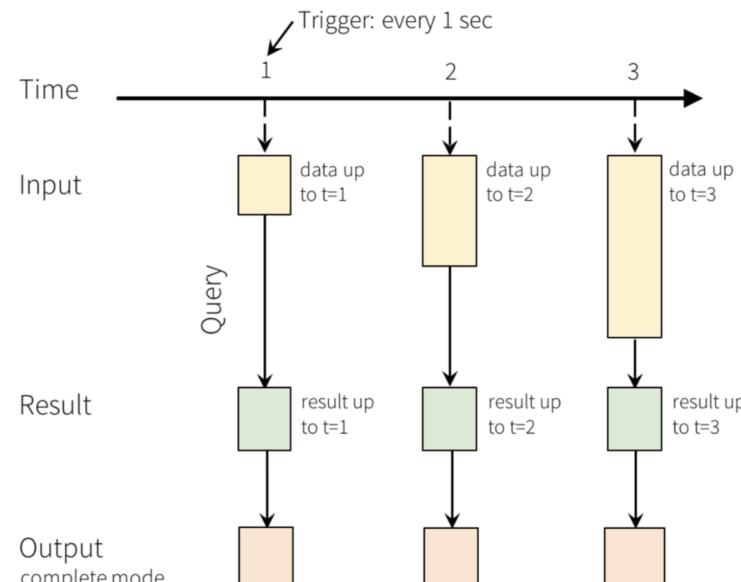
Programming Model

A query on the input will generate **Result Table**

Every trigger interval, let us say, every X seconds, the new rows get appended to **Input Table**

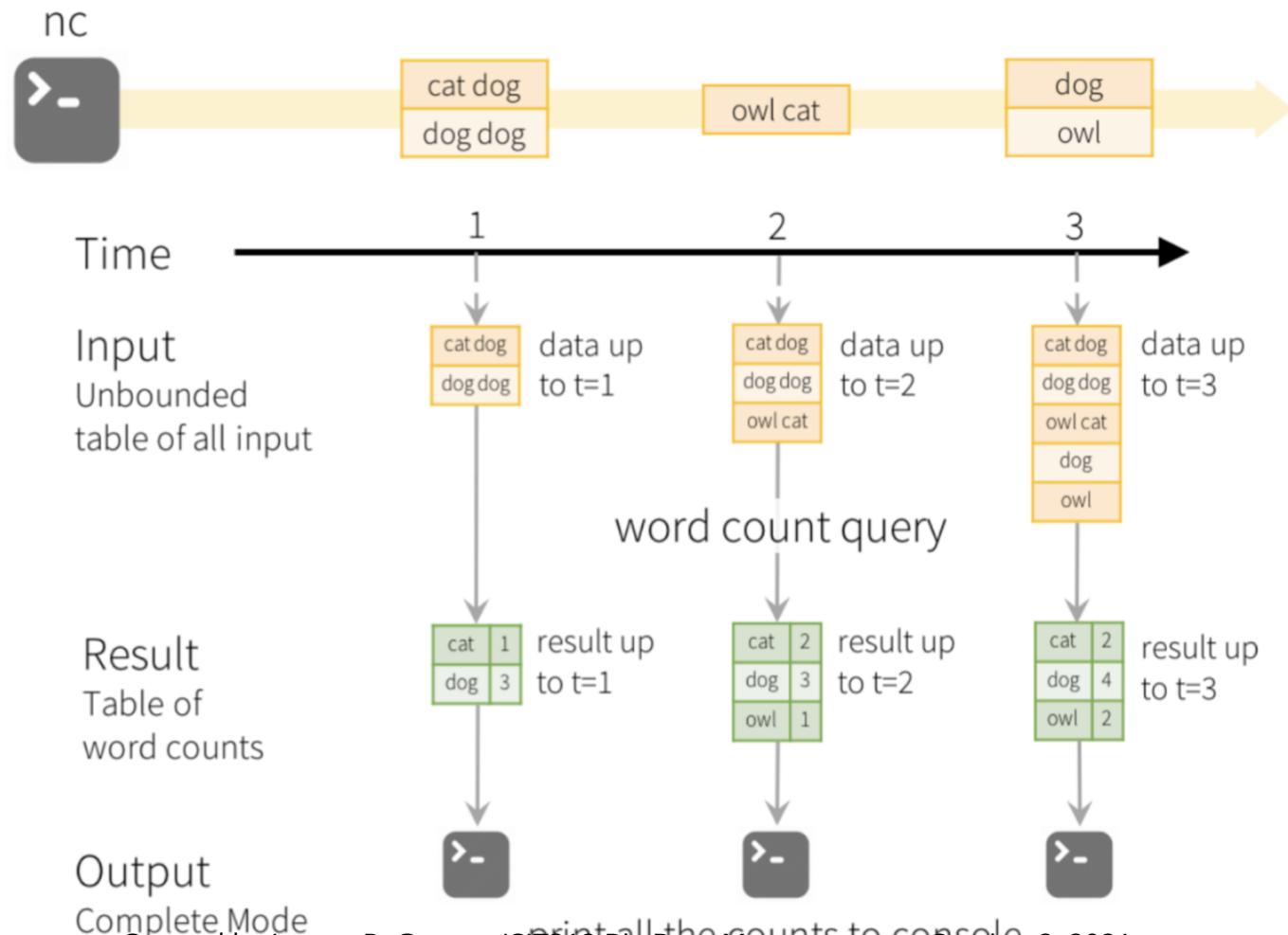
It will eventually updates **Result Table**

Whenever **Result Table** gets updated, we would want to write the changed result rows to an external sink



Programming Model

A complete process



References

A Gentle Introduction to Spark, Databricks, (Available in **READINGS** folder)

[RDD Programming Guide](#)

[Spark SQL, DataFrames and Datasets Guide](#)

Karau H., Fast data processing with Spark Packt Publishing, 2013
(Available from UOW Library)

Srinivasa, K.G., Guide to High Performance Distributed Computing: Case Studies with Hadoop, Scalding and Spark Springer, 2015 (Available from UOW Library)

Chambers B., Zaharia M., Spark: The Definitive Guide, O'Reilly 2017