

Summary literature DSS final

1.	(Lazer et al., 2014) The Parable of Google Flu: Traps in Big Data Analysis	2
	Big Data Hubris.....	2
	Algorithm Dynamics.....	2
	Transparency, Granularity, and All-Data	3
2.	(Broniatowski et al., 2014) Twitter: Big data opportunities	3
	Response.....	3
3.	(Chambers et al., 2018) Apache Spark. The Definitive Guide. Big Data Processing Made Simple	4
	Chapter 1. What Is Apache Spark?	4
	Chapter 2. A Gentle Introduction to Spark	5
	Chapter 3. A Tour of Spark's Toolset.....	8
	Chapter 10. Spark SQL.....	9
4.	(Chapman et al., 2000) CRISP-DM 1.0 Step-by-step Data Mining Guide	14
	Data mining problem types	16
5.	(Pritzker et al., 2015) NIST Big Data Interoperability Framework	16
	Big Data Definitions	16
	Data Science Definitions	17
	Big Data Features.....	18

1. (Lazer et al., 2014) The Parable of Google Flu: Traps in Big Data Analysis

Google Flu Trends (GFT) made the headlines because it was predicting more than double the proportion of doctor visits than the Center for Disease Control and Prevention (CDC). This paper identifies problems, not limited to GFT but to all big data applications. The data used in these applications definitely hold value, but the methods in big data applications do not yet yield more benefit than traditional methods or theories. The two issues explored in this paper are: big data hubris and algorithm dynamics.

Big Data Hubris

Big Data Hubris is the assumption that big data can replace traditional methods or theories in stead of supplementing it. The quantity of data does not mean you can ignore foundational issues or measurement and construct validity. The popular attention big data received, does not follow from instruments which produce valid and reliable data.

In the initial version of GFT they tried to find matches among 50 million search terms to fit 1152 data points. This led to the overfitting the big data to the small number of cases, the developers weeded out seasonal search terms, making them miss a nonseasonal flu pandemic. In short, it was more a winter detector than a flu detector.

After an update, the new GFT overestimates flu prevalence. The errors from the GFT are not randomly distributed, and the direction and magnitude of the error varies with the time of the year (seasonality). This means that GFT overlooks information that could be extracted with traditional statistical methods.

A study showed that the original CDC data (former method for predicting flu) does the same job as GFT, sometimes even better.

Is GFT useless? No, greater value can be obtained by combining GFT with other near-real-time health data. For example, by combining GFT with lagged CDC data and dynamically recalibrating GFT.

Algorithm Dynamics

All empirical research stands on a foundation of measurement. GFT was unstable since algorithm dynamics affected the search algorithm. Algorithm dynamics are the changes made by engineers to improve the commercial service and by consumers in using that service. Changes in the algorithm and in user behaviour affected GFT. Like a media-stoked panic. A more likely culprit is changes made by Google's search algorithm itself. Search patterns are the result of thousands of decisions made by the programmers and by millions of consumers worldwide.

There are challenges in replicating GFT's original algorithm:

- No documentation of the original 45 search terms used.
- Google Correlate does allow to find correlations with a given time series. However, that's on a national level, whereas GFT was used at regional levels.
- Google Correlate also fails to return sample search terms report in GFT publications.

Google Correlate did reveal interesting differences. Like searching for treatments and for whether it is the flu or just a cold. These differences are almost the same as the errors of GFT. The introduction of suggested additional search increased this difference.

The introduction of recommended searches based on what others have searched also increases the magnitude of certain searches.

These explanations for changes in search behaviour are ‘blue team’ dynamics. Here the algorithm producing the data is modified by the service provider in accordance with their business model. This not only counts for Google, but also for Twitter and Facebook. Since they are constantly being reengineered, the replication of studies is highly questioned.

‘Red team’ dynamics is when research subjects try to manipulate the data generating process. For example, making sure your topic is trending on Twitter or Facebook (like the Russians did with Trump?). This is not a problem for GFT though.

Transparency, Granularity, and All-Data

Critical lessons learned from GFT for big data analysis:

- Transparency and Replicability
 - Supporting materials for GFT were not enough to replicate the research.
 - What is at stake is twofold:
 - Scientists need to be able to assess work on which they are building.
 - The growth of knowledge needs the data (which is now withhold).
- Use Big Data to Understand the Unknown
 - The CDC data did a fine job, at 90%, why waste time to gain the last 10%? It is more valuable to find other things, like on another granularity level.
- Study the Algorithm
 - Because of the constantly change in Google and others, scientists need a better understanding of how these changes occur over time using data sources across time and other data sources to ensure robustness.
- It’s Not Just About Size of the Data
 - Big data research and traditional statistic don’t trust each other. Big data offers huge possibilities. ‘Small data’ often offer information that is not contained in big data. We should therefore combine the two in an ‘all data revolution’.

2. (Broniatowski et al., 2014) Twitter: Big data opportunities

The letter supposes that the limitations from the previous paper are overcome. Analyses that use Twitter account for the concerns of: 1) replicability, 2) overfitting, 3) construct validity, 4) granularity, and 5) temporal confounds.

GFT cannot be replicated because it is based on proprietary data, Twitter is open. Changes to the social media platform need not impact replicability if the data is open. They agree that GFT overestimated influenza by media attention etc with signals of actual infection. They claim that these signals are separable on Twitter, and that their Twitter evaluations do not suffer from peak overestimation. Finally, their analysis controls for seasonality and temporal autocorrelation. A blind evaluation using municipal data confirm the strong relation, successfully demonstrating the ability to understand the prevalence of flu at local levels.

They finish with saying that concerns to GFT should not be generalised to other big data analyses.

Response

They start with saying that indeed the problems of GFT are not present in all big data projects. They meant to provide constructive critique by highlighting possible pitfalls for big data analyses. They agree Twitter has potential. It provides an excellent representation of those who choose to express an opinion publicly. They emphasize the possible ‘red team’ dynamics (armies of bots producing content) and state the open question whether the data can be used to represent (not everyone is on Twitter) the entire US.

They finish with claiming that social media perhaps one day can predict flu prevalence. But this would require a careful evaluation and recalibration of methodologies, replication, and error processes. By building strong collaborations and adhering to rigorous standards, we should be able to extract considerably more information from these highly informative new data sources.

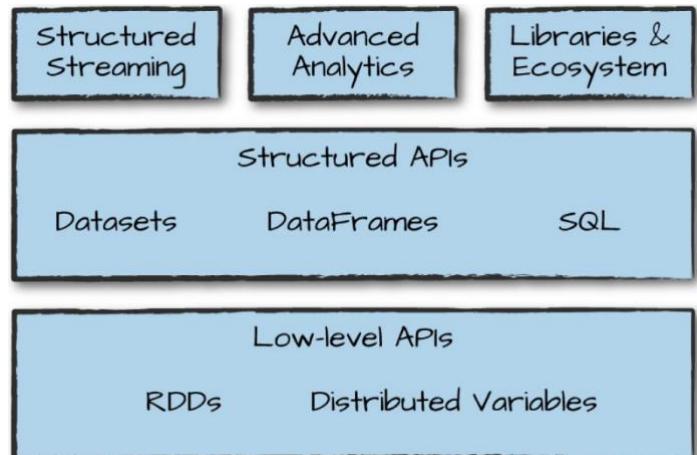
3. (Chambers et al., 2018) Apache Spark. The Definitive Guide. Big Data Processing Made Simple

Chapter 1. What Is Apache Spark?

Spark is a unified computing engine and a set of libraries for parallel data processing on computer clusters. It became a standard tool for developers and data scientists and supports Python, Java, Scala, and R. It is an easy system to start with a scale-up. The figure illustrates the components and libraries that Spark offers.

Spark's Philosophy

A unified computing engine and set of libraries for big data.



- Unified
 - Support a wide range of data analytics tasks
 - The unified nature makes tasks easier and more efficient to write:
 - Provides consistent, composable APIs to build your own user program.
 - APIs designed to enable high performance by optimising across the different libraries and functions.
 - A unified platform for data scientists with a unified set of libraries (e.g. Python or R), and web developers (e.g. Node.js or Django) with a unified set of frameworks.
- Computing engine
 - The scope is to handle loading data and performing computation on it, not the storage itself. It can be used with a wide variety of persistent storage systems.
 - Motivation: the data usually already is somewhere, transportation is expensive, so we move the computation to the data.
 - Different from Hadoop and MapReduce since these two have to be used together, and Spark can be used more broadly.
- Libraries
 - The libraries build on the design as a unified engine providing a unified API for common data analysis tasks. Spark supports a wide array of external libraries. The Spark core has changed little since it was first released, but the libraries have grown to provide more functionalities. Some are:
 - SQL and structured data (Spark SQL)
 - Machine learning (MLlib)
 - Stream processing (Spark Streaming and Structured Streaming)
 - Graph analytics (GraphX)

Context: The Big Data Problem

Why do we need a new engine and programming model for data analytics?

Processor speed increase came to a stop in 2005, due to hard limits in heat dissipation. Therefore, a switch to parallel CPU cores. This meant applications needed to be modified, paving the road for models such as Apache Spark.

The technologies for storing (costs per TB) and collecting (sensors) data did not slow down in 2005. This results in an inexpensive data collection, but an expensive data processing. The processing requires large, parallel computations, often on clusters of machines: Spark.

History of Spark

Spark began as a research project when Hadoop MapReduce was dominant in parallel programming. By working with MapReduce, the makers of Spark found a list of problems and use cases and start designing from there. Two things became clear:

1. MapReduce made it challenging and inefficient to build large applications.
2. Cluster computing held tremendous potential.

The first version of Spark could succinctly express multistep applications in an API over a new engine that could perform in-memory data sharing. Only able to support batch applications. Soon it became clear that interactive data science and ad hoc queries were necessary. This was solved by plugging the Scala interpreter into Spark. This idea was the basis for Shark, an engine that could run SQL queries over Spark.

It quickly became clear that the power lies in the new libraries. So, they made sure that these APIs would be highly interoperable. In 2013 the project became widespread, where in 2014 Spark 1.0 was released, and Spark 2.0 in 2016.

Finally, the core idea of composable APIs has been refined. Over time, the project added a plethora of new APIs that build on this more powerful structured foundation.

The Present and Future of Spark

Many new projects push the boundaries of what's possible with the system, like Structured Streaming. The technology is a huge part of companies solving massive-scale data challenges, from technology companies and institutions to scientific data analysis.

Spark will be a cornerstone of companies doing big data analysis.

Chapter 2. A Gentle Introduction to Spark

Spark's Basic Architecture

Spark manages and coordinates the execution of tasks on data across a cluster of computers. The cluster is managed by Spark's standalone cluster manager, YARN, or Mesos. Next, Spark Applications are submitted to these cluster managers, which will grant resources to the application so that it can complete the work.

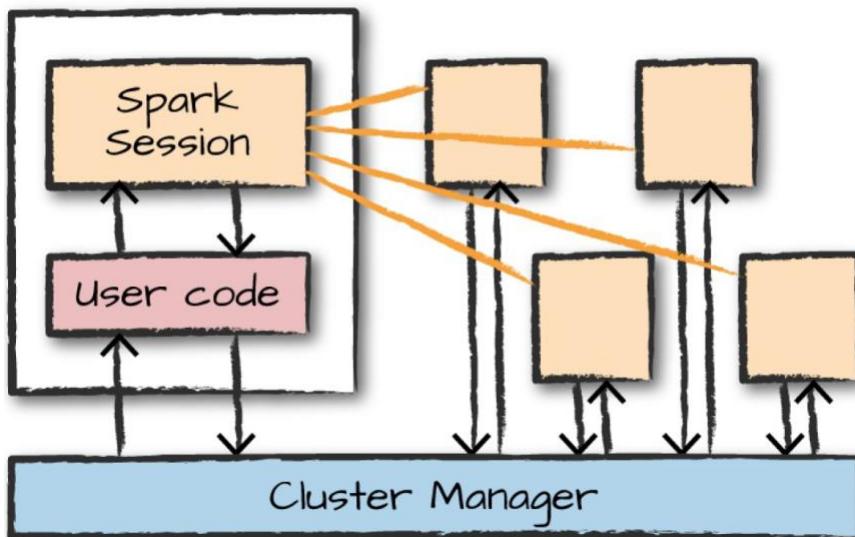
Spark Applications consist of a *driver* process and a set of *executor* processes. The *driver* runs our main() function, sits on a node in the cluster, is absolutely essential, and is the heart of a Spark Application and is responsible for:

- 1) Maintaining information about the Spark Application
- 2) Responding to a user's program or input
- 3) Analysing, distributing, and scheduling work across the executors.

The *executors* are responsible for carrying out the work. Responsible for:

- 1) Executing code assigned to it
- 2) Report the state of the computation back to the *driver*

Driver Process



Executors

Important so far:

- Spark employs a cluster manager that keeps track of the resources available
- The driver process is responsible for executing the driver program's commands across the executors to complete a given task.

Mostly, the executors run on Spark code, but the driver can be 'driven' from a number of different languages, with the help of:

Spark's Language APIs

Spark presents some core concepts in every language, the concepts are then translated into Spark code. If you use just the Structured APIs, you can expect all languages to have similar performance.

- Scala
 - Spark is primarily written in Scala, making it Spark's default language.
- Java
 - Spark's authors have been careful to ensure that you can write Spark code in Java.
- Python
 - Python supports nearly all constructs that Scala supports.
- SQL
 - Spark supports a subset of the ANSI SQL 2003 standard, making it easy for analysts and non-programmers to take advantage of the big data powers of Spark.
- R
 - Two commonly used R libraries: one as a part of Spark core (SparkR) and another as an R-community driven package (sparklyr).

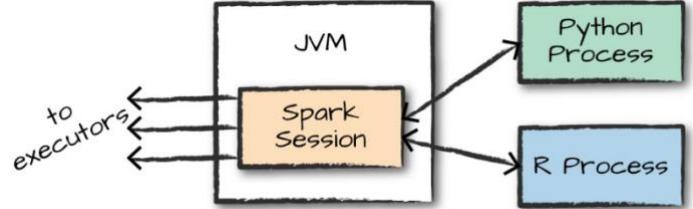


Figure 2-2. The relationship between the `SparkSession` and Spark's Language API

Each language API maintains the same core concepts as described above. The `SparkSession` object is available to the user. This `SparkSession` is the driver process, from which the user controls the Spark Application. When using R or Python, you don't write explicit JVM instructions, but you write R and Python code that Spark translates into code that it can run on the executor JVM.

Spark has two fundamental sets of APIs: the low-level 'unstructured' APIs, and the higher-level structured APIs.

DataFrames

A DataFrame is the most common structured API, simply represents a table of data with rows and columns. The list that defines these rows and columns and the types within those columns is called the *schema*. The difference between a DataFrame in Spark and in R and Python is that in R and Python they exist on one machine rather than multiple machines. However, because the language interfaces, it's quite easy to convert Python and R DataFrames to Spark DataFrames.

Partitions

To allow every executor to perform work in parallel, Spark breaks up the data into *partitions*. A partition is a collection of rows that sit on one physical machine in your cluster. You still have a parallelism of one when: there is one partition, or one executor.

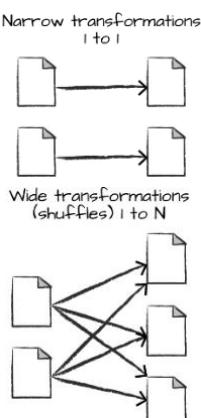
With DataFrame, you don't manipulate partitions manually or individually. You simply specify high-level transformations of data, and Spark determines how this work will execute on the cluster.

Transformations

The core data structure (RDDs) are *immutable*, meaning they cannot be changed. Through *transformations* you instruct Spark on what you like to modify. Transformations are the core of how you express your business logic using Spark. Two types:

- Narrow transformations
 - Where each input partition will contribute to only one output partition.
 - Wide transformations
 - Where input partitions contributing to many output partitions, referred to as a *shuffle* whereby Spark exchanges partitions across the cluster.

With narrow transformations, Spark will perform *pipelining*, meaning that if we specify multiple filters on DataFrame, they'll all be performed in-memory. When performing a shuffle, Spark writes the results to disks.



Lazy evaluation means that Spark will wait until the very last moment to execute the graph of computation instructions. This provides benefits because Spark can optimize the entire data flow from end to end. Spark makes a *directed acyclic graph* (DAG), which is an execution plan of transformations.

Actions

Transformations allow us to build up our logical transformation plan. To trigger the computation, run an *action*. There are three kinds of actions:

- 1) Actions to view data in the console
 - 2) Actions to collect data to native objects in the respective language
 - 3) Actions to write to output data sources

To read data from csv, you can use *schema inference*, which means we want Spark to take a best guess at what the schema of our DataFrame should be. And specify that the first row is the header in the file. Code:

```
# in Python
flightData2015 = spark\
    .read\
    .option("inferSchema", "true")\
    .option("header", "true")\
    .csv("/data/flight-data/csv/2015-summary.csv")
```

The logical plan of transformation that you can build up for a DataFrame sits at the heart of Spark's programming model. Since Spark then know how to recompute any partition by performing all of the operations it had before on the same input data.

DataFrames and SQL

With Spark SQL you can register any DataFrame as a table and query it using pure SQL. There is no performance difference between writing SQL queries or writing DataFrame code, since they both compile to the same underlying plan that we specify in DataFrame code.

Chapter 3. A Tour of Spark's Toolset

This chapter presents some APIs and a few of the main libraries.

Running production applications

Spark makes it easy to develop and create big data programs. You can easily turn them into production with *spark-submit*, a built-in command-line tool. It lets you send your application code to a cluster and launch it to execute there. The application will run until it completes the task or encounters an error, this with the help of Spark's support cluster managers (Standalone, Mesos, and YARN).

Spark-submit offers several controls with which you can specify the resources your application needs and how it should be run.

The applications can be written in any language Spark supports.

Datasets: type-safe structured APIs

Datasets is a type-safe API for writing statically typed code in Java and Scala, Python and R are not available (those languages are dynamic). The Dataset API gives users the ability to assign a Java/Scala class to records within a DataFrame. The APIs available on Datasets are type-safe, meaning that you cannot accidentally view the objects in a Dataset as begin of another class than the class you put in initially. Making it attractive for large applications, which multiple software engineers.

The Dataset class is parameterised with the type of object contained inside. Dataset[Person] will be guaranteed to contain objects of class Person.

Datasets can be used only when you need or want to. For example, in a lower level in the DataFrame, where on a higher level more rapid analysis can be performed.

Another advantage: when you call *collect* or *take* on a Dataset, it will collect objects of the proper type, not DataFrame rows.

Structured streaming

You can take the same operations that you perform in batch mode and run them in a streaming fashion with Structured Streaming API. Reducing latency and allow for incremental processing. Allows you to rapidly and quickly extract value out of streaming systems with no code changes. Easy to conceptualise because you can write a batch job as a prototype and if it works simply convert it to a streaming job.

Machine learning and advanced analytics

Mlib allows for pre-processing, munging, training of models, and making prediction at scale on data. You can even use models trained in Mlib to make predictions in Structured Streaming. The sophisticated machine learning API provides a variety of machine learning tasks, from classification to regression, and clustering to deep learning.

Lower-level APIs

Spark includes a few lower-level primitives to allow for arbitrary Java and Python object manipulation via Resilient Distributed Datasets (RDDs). For the most part, stick to Structured APIs, RDDs are only used instead of the structured APIs for manipulating some very raw unprocessed and unstructured data.

SparkR

It's a tool for running R on Spark. Follows the same principles as all of Spark's other language bindings. To use it, simply import it into your environment and run your code.

Conclusion

One of the best parts about Spark is the ecosystem of packages and tools that the community has created. Spark's simple, robust programming model makes it easy to apply to a large number of problems, and the vast array of packages that have crept up around it, created by hundreds of different people, are a true testament to Spark's ability to robustly tackle a number of business problems and challenges.

Chapter 10. Spark SQL

With Spark SQL you can run SQL queries against views or tables organised in databases, arguably one of the most important and powerful features in Spark.

SQL or Structured Query Language is domain-specific language for expressing relational operations over data. It is used in all relational databases and in many 'NoSQL' databases through a dialect. Spark implements a subset of ANSI SQL:2003.

Before Spark's rise, *Apache Hive* was the de facto big data SQL access layer. With the release of Spark 2.0, its authors created a superset of Hive's support, writing a native SQL parser that supports both ANSI-SQL as well as HiveQL queries. This, along with its interoperability with DataFrames, makes it a powerful tool for all sorts of companies.

The power of Spark SQL derives from several key facts:

- 1) SQL analysts can now take advantage of Spark's computation abilities.
- 2) Data engineers and scientists can use Spark SQL where appropriate in any data flow.

Spark SQL is intended to operate as an online analytic processing (OLAP) database, not an online transaction processing (OLTP) database.

Spark's Relationship to Hive

Spark SQL can connect to Hive metastores. Metastores is the way in which Hive maintains table information for use across sessions. With Spark SQL you can connect to your Hive metastore, making migrating from a legacy Hadoop environment popular.

The Spark SQL CLI is a convenient tool with which you can make basic Spark SQL queries in local mode from the command line.

Spark provides a Java Database Connectivity (JDBC) interface by which either you or a remote program connects to the Spark driver in order to execute Spark SQL queries.

Catalog

The highest level of abstraction in Spark SQL is the Catalog. It's the storage of metadata about the data stored in your table as well as other helpful things like databases, tables, functions, and views.

Tables

To do anything useful with Spark SQL, you first need to define tables. Tables are equivalent to a DataFrame, they are a structure of data against which you can run commands. We

can join tables, filter them, aggregate them, and perform different manipulations. The core difference between DataFrames and tables is that you define DataFrames in the scope of a programming language, whereas you define tables within a database.

Important is that in Spark, tables always contain data. This is important because if you go to drop a table, you can risk losing the data when doing so.

One important note is the concept of *managed* (having the data yourself) versus *unmanaged* (only having the metadata) tables. There is data in the tables, and data about the tables (metadata). If you define a table from files on disk, only having the metadata, it is unmanaged. When you use `saveAsTable` on a DataFrame, you are creating a managed table for which Spark will track of all the relevant information.

Tables

- Creating tables
 - You can create tables from a variety of sources. Unique is the capability of reusing the entire Data Source API within SQL.

```
CREATE TABLE flights (
    DEST_COUNTRY_NAME STRING,
    ORIGIN_COUNTRY_NAME STRING, count LONG)
USING JSON OPTIONS (path '/data/flight-data/json/2015-summary.json')
```

 - When you do not use the `USING` statement, Spark will default to a Hive SerDe configuration. Which is slower than Spark's native.
 - Add comments to help other developers

```
CREATE TABLE flights_csv (
    DEST_COUNTRY_NAME STRING,
    ORIGIN_COUNTRY_NAME STRING COMMENT "remember, the US will be most prevalent",
    count LONG)
USING csv OPTIONS (header true, path '/data/flight-data/csv/2015-summary.csv')
```
 - Create table from query

```
CREATE TABLE flights_from_select USING parquet AS SELECT * FROM flights
```
 - Control the layout of table by writing a portioned dataset

```
CREATE TABLE partitioned_flights USING parquet PARTITIONED BY (DEST_COUNTRY_NAME)
AS SELECT DEST_COUNTRY_NAME, ORIGIN_COUNTRY_NAME, count FROM flights LIMIT 5
```
 - The tables will be available in Spark even through sessions, temporary tables do not exist. You must create a temporary view.
- Creating external tables
 - If you want to port your legacy Hive statements to Spark SQL you can use this. You create an *unmanaged* table.

```
CREATE EXTERNAL TABLE hive_flights (
    DEST_COUNTRY_NAME STRING, ORIGIN_COUNTRY_NAME STRING, count LONG)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' LOCATION '/data/flight-data-hive/'
```
- Inserting into Tables
 - ```
INSERT INTO flights_from_select
SELECT DEST_COUNTRY_NAME, ORIGIN_COUNTRY_NAME, count FROM flights LIMIT 20
```
- Describing table metadata
  - To view the comments, and other metadata

```
DESCRIBE TABLE flights_csv
```
  - See the partitioning scheme

```
SHOW PARTITIONS partitioned_flights
```
- Refreshing table metadata
  - Important to ensure that you're reading from the most recent set of data.

```
REFRESH table partitioned_flights
```
- Dropping tables
  - You cannot delete tables, you can only drop them. If you drop a managed table, both the data and the table definition will be removed. If you drop an

unmanaged table, no data will be removed but you will no longer be able to refer to this data by the table name.

```
DROP TABLE flights_csv;
```

- Caching tables

```
CACHE TABLE flights
```

## Views

- A view specifies a set of transformations on top of an existing table.
- Creating views
  - To an end user, views are displayed as tables, except than rewriting all of the data to a new location, they simply perform a transformation.

```
CREATE VIEW just_usa_view AS
 SELECT * FROM flights WHERE dest_country_name = 'United States'
```
  - Like tables, you can create temporary views that are available only during the current session, using 'CREATE TEMP VIEW'.
  - Or global temp view, are viewable across the entire Spark application, but removed at the end of the session, using 'CREATE GLOBAL TEMP VIEW'
  - Or specify that you would like to overwrite a view. You can overwrite temp views and regular views, using: 'CREATE OR REPLACE (TEMP) VIEW'
  - You can query views just as if it is a table.

```
SELECT * FROM just_usa_view_temp
```
- Dropping views, no data is removed, just the view  

```
DROP VIEW IF EXISTS just_usa_view;
```

## Databases

- Databases are a tool for organising tables. There is a default database, which Spark will use if none is defined. SQL statements from Spark are executed within the context of a database, when you change the database, any user-defined tables remain in the previous database.
  - See all databases  

```
SHOW DATABASES
```
  - Create databases  

```
CREATE DATABASE some_db
```
  - You might want to use a database to perform a certain query. After you set this database, some queries might not work anymore.  

```
USE some_db
SHOW tables
```

```
SELECT * FROM flights -- fails with table/view not found
```
  - You can fix this by using correct prefixes  

```
SELECT * FROM default.flights
```
  - You can see what database you're currently using  

```
SELECT current_database()
```
  - Dropping databases  

```
DROP DATABASE IF EXISTS some_db;
```

## Select statements

Queries in Spark support the following ANSI SQL requirements:

```
SELECT [ALL|DISTINCT] named_expression[, named_expression, ...]
 FROM relation[, relation, ...]
 [lateral_view[, lateral_view, ...]]
 [WHERE boolean_expression]
 [aggregation [HAVING boolean_expression]]
 [ORDER BY sort_expressions]
 [CLUSTER BY expressions]
 [DISTRIBUTE BY expressions]
 [SORT BY sort_expressions]
 [WINDOW named_window[, WINDOW named_window, ...]]
 [LIMIT num_rows]

named_expression:
 : expression [AS alias]

relation:
 | join_relation
 | (table_name|query|relation) [sample] [AS alias]
 : VALUES (expressions)[, (expressions), ...]
 [AS (column_name[, column_name, ...])]

expressions:
 : expression[, expression, ...]

sort_expressions:
 : expression [ASC|DESC][, expression [ASC|DESC], ...]
```

- Often you need conditionally replacement value in your query. You can do this by using a case...when...then...end style statement.

```
SELECT
 CASE WHEN DEST_COUNTRY_NAME = 'UNITED STATES' THEN 1
 WHEN DEST_COUNTRY_NAME = 'Egypt' THEN 0
 ELSE -1 END
 FROM partitioned_flights
```

- Complex types are a departure from standard SQL, they are a powerful feature and there are three core complex types in Spark SQL: structs, lists, and maps.

### Structs

- Structs are like maps. They provide a way of creating nested data. To create one:

```
CREATE VIEW IF NOT EXISTS nested_data AS
 SELECT (DEST_COUNTRY_NAME, ORIGIN_COUNTRY_NAME) as country, count FROM flights
```

- To use it, you can now call the nested\_data

```
SELECT * FROM nested_data
```

- You can query individual columns and subvalues from a struct.

```
SELECT country.DEST_COUNTRY_NAME, count FROM nested_data
 SELECT country.* , count FROM nested_data
```

### Lists

- Use collect\_list to create a list of values, or use collect\_set to create an array without duplicate values. Both are aggregation functions and can therefore only be used in aggregations.

- ```

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

```
- Create an array manually within a column:
`SELECT DEST_COUNTRY_NAME, ARRAY(1, 2, 3) FROM flights`
 - Query lists by position using a Python-like array query syntax:
`SELECT DEST_COUNTRY_NAME as new_name, collect_list(count)[0]
 FROM flights GROUP BY DEST_COUNTRY_NAME`
 - You can also convert an array back into rows. You can do this with the explode function, it does the opposite of 'collect' and returns the original DataFrame. First create an array:
`CREATE OR REPLACE TEMP VIEW flights_agg AS
 SELECT DEST_COUNTRY_NAME, collect_list(count) as collected_counts
 FROM flights GROUP BY DEST_COUNTRY_NAME`
- Explode the array:
- ```
SELECT explode(collected_counts), DEST_COUNTRY_NAME FROM flights_agg
```

## Functions

- In addition to complex types, Spark SQL provides functions.
    - To see a list of functions in Spark:  
`SHOW FUNCTIONS`
    - There is a difference between system- and user functions. System functions are incorporated in Spark, user functions are defined by users  
`SHOW SYSTEM FUNCTIONS`  
`SHOW USER FUNCTIONS`
    - Filter the SHOW commands, the LIKE is optional, the \* is a wildcard for all characters.  
`SHOW FUNCTIONS LIKE "collect*";`
    - If you want full documentation about the specific function, use DESCRIBE
  - User-defined functions
    - You can define functions, writing the functions in the language of your choice and then registering it appropriately:  
`def power3(number:Double):Double = number * number * number`  
`spark.udf.register("power3", power3(_:Double):Double)`
- ```
SELECT count, power3(count) FROM flights
```

Subqueries

With subqueries, you can specify queries within other queries, making it possible to specify some logic within your SQL. In Spark, there are two fundamental subqueries. Correlated subqueries and uncorrelated subqueries. Where correlated subqueries use information from outer scope. Spark also supports predicate subqueries, allowing filtering based on values

- Uncorrelated predicate subqueries

```
SELECT * FROM flights
WHERE origin_country_name IN (SELECT dest_country_name FROM flights
                                GROUP BY dest_country_name ORDER BY sum(count) DESC LIMIT 5)
```

- Correlated predicate subqueries

```
SELECT * FROM flights f1
WHERE EXISTS (SELECT 1 FROM flights f2
               WHERE f1.dest_country_name = f2.origin_country_name)
AND EXISTS (SELECT 1 FROM flights f2
               WHERE f2.dest_country_name = f1.origin_country_name)
```

4. (Chapman et al., 2000) CRISP-DM 1.0 Step-by-step Data Mining Guide

CRoss-Industry Standard Process for Data Mining

CRIPS-DM is a hierarchical process model. Top level, the data mining process is organized into a number of phases. The second level are generic tasks which belong to phases. Third level, the specialized task level, is the place where is described how to execute generic tasks in specific situations. The last and fourth level hold process instances, a record of the actions, decisions, and results of an actual data mining engagement.

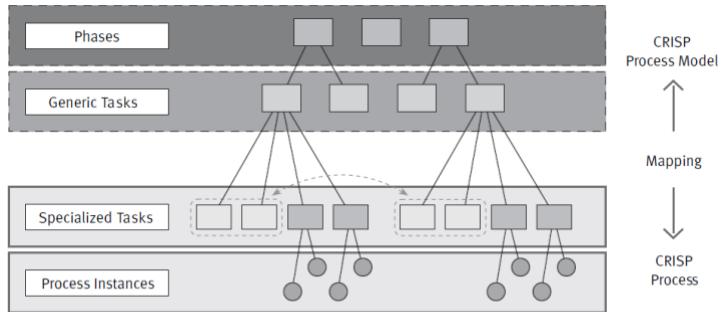


Figure 1: Four level breakdown of the CRISP-DM methodology

It is important to know the data mining context. In CRIPS-DM, they distinguish four different dimensions:

1. Application domain
2. Data mining problem type
3. Technical aspect

4. Tool and technique

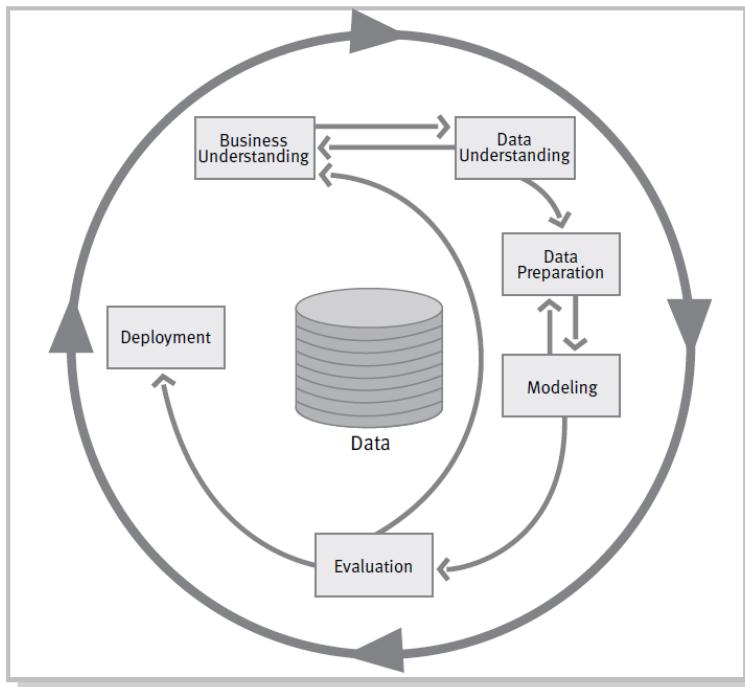


Figure 2: Phases of the CRISP-DM reference model

Business Understanding	Data Understanding	Data Preparation	Modeling	Evaluation	Deployment
Determine Business Objectives <i>Background Business Objectives Business Success Criteria</i> Assess Situation <i>Inventory of Resources Requirements, Assumptions, and Constraints Risks and Contingencies Terminology Costs and Benefits</i> Determine Data Mining Goals <i>Data Mining Goals Data Mining Success Criteria</i> Produce Project Plan <i>Project Plan Initial Assessment of Tools and Techniques</i>	Collect Initial Data <i>Initial Data Collection Report</i> Describe Data <i>Data Description Report</i> Explore Data <i>Data Exploration Report</i> Verify Data Quality <i>Data Quality Report</i>	Select Data <i>Rationale for Inclusion/Exclusion</i> Clean Data <i>Data Cleaning Report</i> Construct Data <i>Derived Attributes Generated Records</i> Integrate Data <i>Merged Data</i> Format Data <i>Reformatted Data</i> <i>Dataset Dataset Description</i>	Select Modeling Techniques <i>Modeling Technique Modeling Assumptions</i> Generate Test Design <i>Test Design</i> Build Model <i>Parameter Settings Models Model Descriptions</i> Assess Model <i>Model Assessment Revised Parameter Settings</i>	Evaluate Results <i>Assessment of Data Mining Results w.r.t. Business Success Criteria Approved Models</i> Review Process <i>Review of Process</i> Determine Next Steps <i>List of Possible Actions Decision</i>	Plan Deployment <i>Deployment Plan</i> Plan Monitoring and Maintenance <i>Monitoring and Maintenance Plan</i> Produce Final Report <i>Final Report Final Presentation</i> Review Project <i>Experience Documentation</i>

Figure 3: Generic tasks (bold) and outputs (italic) of the CRISP-DM reference model

Business understanding:

Understand the project objectives and requirements, convert into a data mining problem and a preliminary plan.

Data understanding: Initial data collection, become familiar with the data: data quality, first insights, detect subsets.

Data preparation: Construct the final data set.

Modelling: Select modelling techniques, typically several.

Evaluation: Evaluate models and review steps, make sure business objective is satisfied.

Deployment: Organize and present the knowledge gained, depending on the requirements.

Data mining problem types

- Data description and summarization
 - o Give a concise description of characteristics of the data, typically in elementary and aggregated form
- Segmentation
 - o The separation of the data into interesting and meaningful subgroups or classes. Techniques:
 - Clustering, Neural networks, Visualization
- Concept descriptions
 - o An understandable description of concepts or classes. Techniques:
 - Rule induction methods, Conceptual clustering
- Classification
 - o Making sets of objects, characterized by attributes or features that belong to different classes. Techniques:
 - Discriminant analysis, Rule induction methods, Decision tree learning, Neural networks, K nearest neighbour, Case-based reasoning, Genetic algorithms
- Prediction
 - o Similar to classification, but the target attribute is not discrete, but continuous. Also: regression. Techniques:
 - Regression analysis, Regression trees, Neural networks, K nearest neighbour, Box-Jenkins methods, Genetic algorithms.
- Dependency analysis
 - o Describe significant dependencies, or associations, between data items or events. Techniques:
 - Correlation analysis, Regression analysis, Association rules, Bayesian networks, Inductive logic programming, Visualization techniques.

5. (Pritzker et al., 2015) NIST Big Data Interoperability Framework

Big data is not only more volume, but there is also a big increase in the quantity of unstructured data. The shift to relation data modelling was an evolution, Big Data revolution is another shift in architecture. To understand this revolution, four aspects must be considered:

- 1) The characteristics of the datasets
- 2) The analysis of the datasets
- 3) The performance of the systems that handle the data
- 4) The business considerations of cost-effectiveness

Big Data Definitions

Big Data refers to the inability of traditional data architectures to efficiently handle the new datasets. Characteristics are:

- Volume
- Variety
- Velocity
- Variability

Big Data consists of extensive datasets, primarily in the characteristics of volume, variety, and/or variability, that requires a scalable architecture for efficient storage, manipulation, and analysis.

Scaling can be done vertically or horizontally. Vertical is increasing systems parameters, this is limited by physical capabilities. Horizontal is making use of distributed individual resources integrated to act as a single system, which is at the heart of Big Data.

The **Big Data paradigm** consists of the distribution of data systems across horizontally coupled, independent resources to achieve the scalability needed for the efficient processing of extensive datasets.

Massively parallel processing refers to a multitude of individual processors working in parallel to execute a particular program.

This however led to a couple of complications: 1) message parsing, 2) data movement, 3) latency in the consistency across resources, 4) load balancing, and 5) system inefficiencies.

Big Data engineering includes advanced techniques that harness independent resources for building scalable data systems when the characteristics of the datasets require new architectures for efficient storage, manipulation, and analysis.

Non-relational models, frequently referred to as **NoSQL**, refer to logical data models that do not follow relational algebra for the storage and manipulation of data.

Schema-on-read is the application of a data schema through preparation steps such as transformations, cleansing, and integration at the time the data is read from the database.

Computational portability is the movement of the computation to the location of the data.

Critical aspects in parallel data architectures: 1) data locality, 2) interoperability (tools working together), 3) reusability, and 4) extendibility.

More characteristics: 1) Veracity (accuracy of the data), 2) value, 3) volatility (does the data change?), and 4) validity (appropriateness of the data for its intended use).

Data Science Definitions

Data science is the fourth paradigm of science, following experiment, theory, and computational science. Two forms: 1) data exploration, no hypothesis in front, and 2) an empirical method with a hypothesis, a collection of data to address the hypothesis. Many times, there is combination of the two.

Data science is the extraction of actionable knowledge directly from data through a process of discovery, or hypothesis formulation and hypothesis testing.

The **data life cycle** is the set of processes in an application that transform raw data into actionable knowledge, which includes data collection, preparation, analytics, visualization, and access.

The **analytics** process is the synthesis of knowledge from information.

Data science applications implement data transformation processes from the data life cycle in the context of Big Data Engineering

A **data scientist** is a practitioner who has sufficient knowledge in the overlapping regimes of business needs, domain knowledge, analytical skills, and software and systems engineering to manage the end-to-end data processes in the data life cycle.

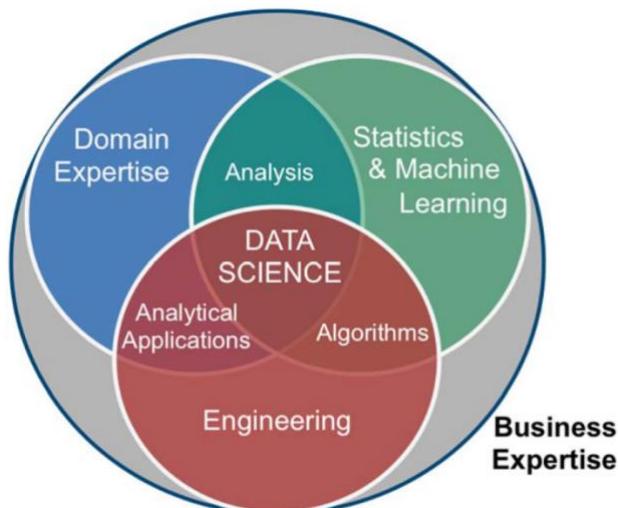


Figure 1: Skills Needed in Data Science

Big Data Features

Important feature in Big Data is metadata, data about data.

Data can be at rest, or in motion. Volume and variety are typical characteristics important for data at rest. New types of non-relational storage for data records are:

- Shared-disk File Systems → suffered from data locking and a performance bottleneck which were overcome by:
- Distributed File Systems
- Distributed Computing
- Resource Negotiation

Velocity and variability are typical characteristics important for data in motion. High velocity is referred to as streaming data.

Data Science Life cycle Model for Big Data

1. Collection
2. Preparation
3. Analysis
4. Action

Data governance refers to administering, or formalizing, discipline (e.g. behavior patterns) around the management of data.

Two issues are debated in the data science community. One is data sampling, and the other the idea that more data is superior to better algorithms.

Data sampling because nowadays we're able to analyse the whole dataset, instead of a sample of it. But have to keep in mind that it is still a sample of a population.