Table of contents

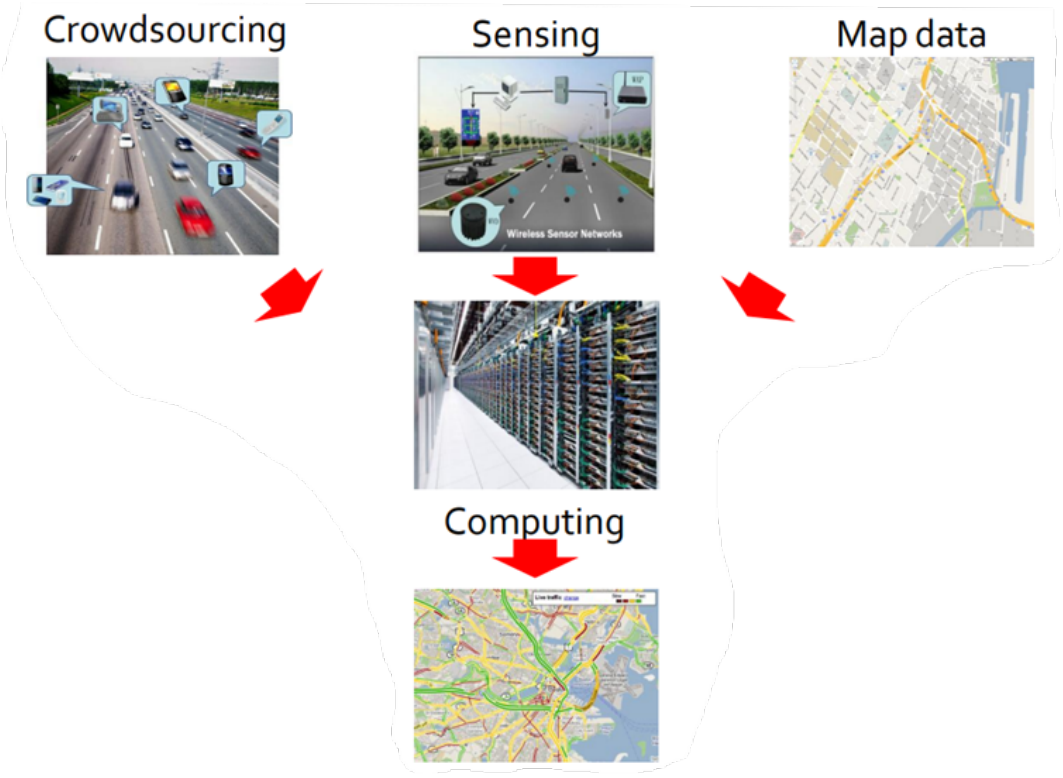
1 Index

2 Introduction to Big data

- User generated content: social networks (web and mobile)
- Health and scientific computing
- Log files: web server log files, machine system log files
- Internet of Things (IoT): sensor networks, RFIDs, smart meters

2.1 Example of Big Data at work

Figure 2.1: Bigdata example



3 The five Vs of Big Data

- Volume: scale of data
- Variety: different forms of data
- Velocity: analysis of streaming data
- Veracity: uncertainty of data
- Value: exploit information provided by data

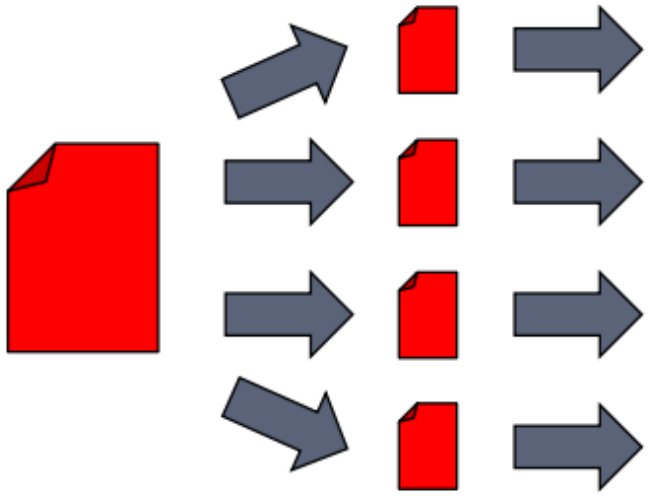
4 The bottleneck and the solution

4.1 Bottleneck

- Processors process data
- Hard drives store data
- We need to transfer data from the disk to the processor

4.2 Solution

- Transfer the processing power to the data
- Multiple distributed disks: each one holding a portion of a large dataset



- Process in parallel different file portions from different disks

5 Big data architectures

From Data Architecture Guide in Microsoft Learn

A big data architecture is designed to handle the ingestion, processing, and analysis of data that is too large or complex for traditional database systems…
Big data solutions typically involve one or more of the following types of workload:

- Batch processing of big data sources at rest
- Real-time processing of big data in motion
- Interactive exploration of big data
- Predictive analytics and machine learning

Consider big data architectures when you need to:

- Store and process data in volumes too large for a traditional database
- Transform unstructured data for analysis and reporting
- Capture, process, and analyze unbounded streams of data in real time, or with low latency

6 Lambda architecture

The most frequently used big data architecture is the Lambda Architecture. The lambda architecture was proposed by Nathan Marz in 2011.

6.1 Definitions

From Nathan Marz

The past decade has seen a huge amount of innovation in scalable data systems. These include large-scale computation systems like Hadoop and databases such as Cassandra and Riak. These systems can handle very large amounts of data, but with serious trade-offs.Hadoop, for example, can parallelize large-scale batch computations on very large amounts of data, but the computations have high latency. You don't use Hadoop for anything where you need low-latency results.
NoSQL databases like Cassandra achieve their scalability by offering you a much more limited data model than you're used to with something like SQL. Squeezing your application into these limited data models can be very complex. And because the databases are mutable, they're not human-fault tolerant.
These tools on their own are not a panacea. But when intelligently used in conjunction with one another, you can produce scalable systems for arbitrary data problems with human-fault tolerance and a minimum of complexity. This is the Lambda Architecture you'll learn throughout the book.

From What is Lambda Architecture? article in Databricks website

Lambda architecture is a way of processing massive quantities of data (i.e. "Big Data") that provides access to batch-processing and stream-processing methods with a hybrid approach.
Lambda architecture is used to solve the problem of computing arbitrary functions.

From Lambda architecture article in Wikipedia

Lambda architecture is a data-processing architecture designed to handle massive quantities of data by taking advantage of both batch and stream-processing methods.
This approach to architecture attempts to balance latency, throughput, and fault tolerance by using batch processing to provide comprehensive and accurate views of batch data, while simultaneously using real-time stream processing to provide views of online data. The two view outputs may be joined before presentation.
Lambda architecture depends on a data model with an append-only, immutable data source that serves as a system of record. It is intended for ingesting and processing timestamped events that are appended to existing events rather than overwriting them. State is determined from the natural

time-based ordering of the data.

6.2 Requirements

Fault-tolerant against both hardware failures and human errors Support variety of use cases that include low latency querying as well as updates Linear scale-out capabilities Extensible, so that the system is manageable and can accommodate newer features easily

6.3 Queries

$$\textbf{query} = \textbf{function}(\textbf{all data})$$

Some query properties

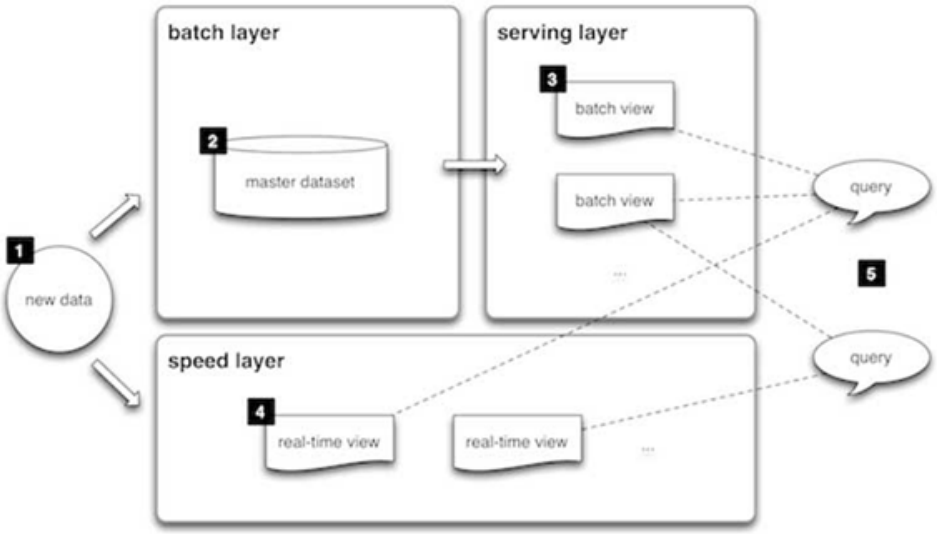
- Latency: the time it takes to run a query
- Timeliness: how up to date the query results are (freshness and consistency)
- Accuracy: tradeoff between performance and scalability (approximations)

It is based on two data paths:

- Cold path (batch layer)
  - It stores all of the incoming data in its raw form and performs batch processing on the data
  - The result of this processing is stored as batch views
- Hot path (speed layer)
  - It analyzes data in real time
  - This path is designed for low latency, at the expense of accuracy

6.4 Basic structure

Figure 6.1: General Lambda architecture



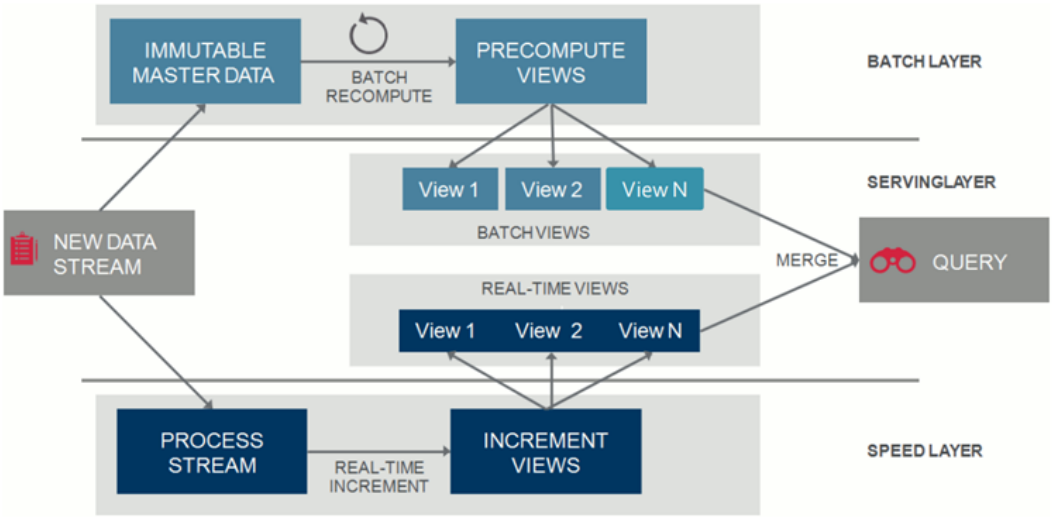
1. All data entering the system is dispatched to both the batch layer and the speed layer for processing
2. The batch layer has two functions:

   i) managing the master dataset(an immutable, append-only set of raw data), and
   ii) to pre-compute the batch views

3. The serving layer indexes the batch views so that they can be queried in low-latency, ad-hoc way
4. The speed layer compensates for the high latency of updates to the serving layer and deals with recent data only
5. Any incoming query can be answered by merging results from batch views and real-time views (e.g., the query looks at the serving layer for days until today, and looks at the speed layer for today's data).

6.5 Detailed view

Figure 6.2: More detailed of Lambda architecture

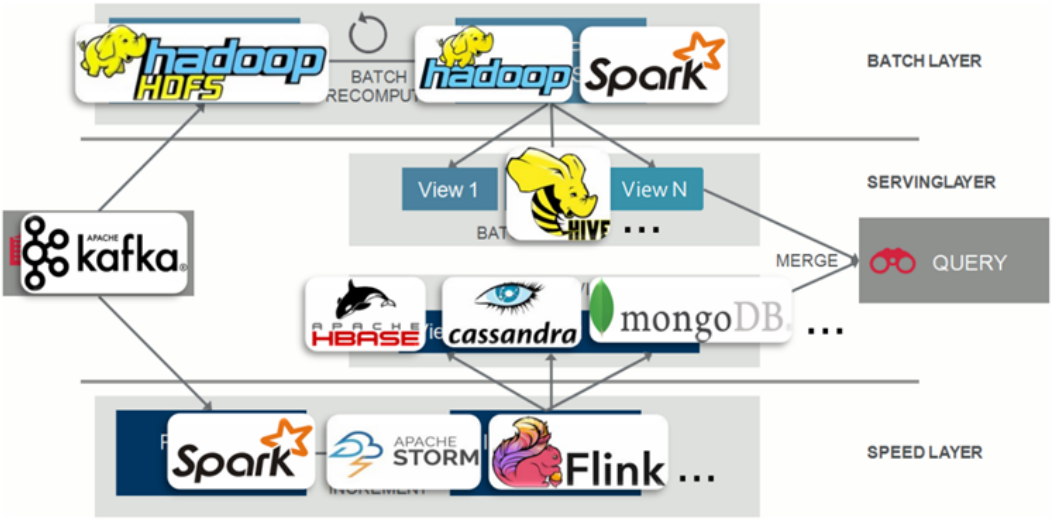


Structure similar to the one described before

0. Data stream
1. Batch layer
    - immutable data
    - precompute views
2. Real-time layer
    - process stream
    - increment views
3. Serving layer

Possible instances

Figure 6.3: More detailed of Lambda architecture



In general, the technologies used are

0. Data stream: Kafka
1. Batch layer:
   - immutable data: Hadoop HDFS
   - precompute views: Hadoop, Spark
   - views: Hive (it is a distributed relational database; SQL-like query language can be used)
2. Real-time layer:
   - process stream and increment views:
     - Spark (it has a module available for managing stream data)
     - Apache Storm (pros: true real-time; cons: sometimes it approximates)
     - Flink (used stream data analysis)
   - views: HBase, Cassandra, MongoDB
3. Serving layer

In general: choose the most suitable technology, but also be able to adapt on what's available.

7 HDFS and Hadoop: command line commands

The content of a HDFS file can be accessed by means of

- Command line commands
- A basic web interface provided by Apache Hadoop. The HDFS content can only be browsed and its files downloaded from HDFS to the local file system, while uploading functionalities are not available.
- Vendor-specific web interfaces providing a full set of functionalities (upload, download, rename, delete, …) (e.g., the HUE web application of Cloudera).

7.1 User folder

Each user of the Hadoop cluster has a personal folder in the HDFS file system. The default folder of a user is `/user/username`

7.2 Command line

The hdfs command can be executed in a Linux shell to read/write/modify/delete the content of the distributed file system. The parameters/arguments of hdfs command are used to specify the operation to execute.

Content of a folder

To list the content of a folder of the HDFS file system, use `hdfs dfs -ls folder`

> **Example**
>
> The command `hdfs dfs -ls /user/garza` shows the content (list of files and folders) of the `/user/garza` folder.
> The command `hdfs dfs -ls .` shows the content of the current folder (i.e., the content of `/user/current_username`).
> Notice that the mapping between the local linux user and the user of the cluster is based on
>
> - A Kerberos ticket if Kerberos is active
> - Otherwise the local linux user is considered

Content of a file

To show the content of a file in the HDFS file system, use `hdfs dfs -cat file_name`

> **i** Example
>
> The command `hdfs dfs -cat /user/garza/document.txt` shows the content of the `/user/garza/document.txt` file stored in HDFS.

Copy a file from local to HDFS

To copy a file from the local file system to the HDFS file system, use `hdfs dfs -put local_file HDFS_path`

> **i** Example
>
> The command `hdfs dfs -put /data/document.txt /user/garza/` copies the local file `/data/document.txt` in the folder `/user/garza` in HDFS.

Copy a file from HDFS to local

To copy a file from the HDFS file system to the local file system, use `hdfs dfs -get HDFS_path local_file`

> **i** Example
>
> The command `hdfs dfs -get /user/garza/document.txt /data/` copies the HDFS file `/user/garza/document.txt` in the local file system folder `/data/`.

Delete a file

To delete a file from the HDFS file system, use `hdfs dfs -rm HDFS_path`

> **i** Example
>
> The command `hdfs dfs -rm /user/garza/document.txt` deletes from HDFS the file `/user/garza/document.txt`

Other commands

There are many other linux-like commands, for example

- `rmdir`
- `du`

- `tail`

See the HDFS commands guide for a complete list.

8 Hadoop

The Hadoop programs are executed (submitted to the cluster) by using the hadoop command. It is a command line program, characterized by a set of parameters, such as

- the name of the jar file containing all the classes of the MapReduce application we want to execute
- the name of the Driver class
- the parameters/arguments of the MapReduce application

8.1 Example

The following command executes/submits a MapReduce application

```
hadoop jar MyApplication.jar \
it.polito.bigdata.hadoop.DriverMyApplication \
1 inputdatafolder/ outputdatafolder/
```

- It executes/submits the application contained in `MyApplication.jar`
- The Driver Class is `it.polito.bigdata.hadoop.DriverMyApplication`
- The application has three arguments
  - Number of reducers (`1`)
  - Input data folder (`inputdatafolder/`)
  - Output data folder (`outputdatafolder/`)