**AIM**

Recognize use cases for Pig, which include:

• ETL data pipelines

• Researching raw data

• Iterative data processing

**OVERVIEW**

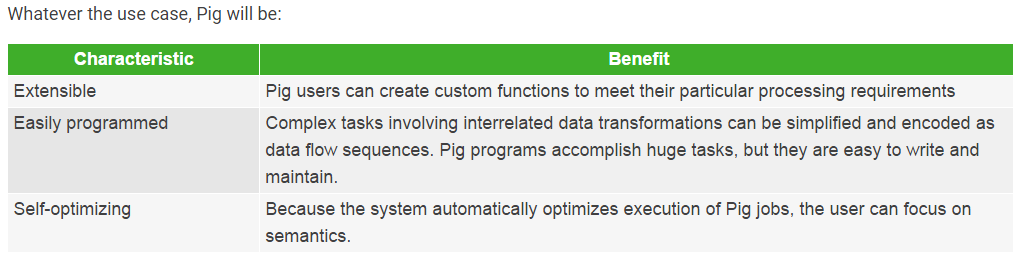
A scripting platform for processing and analyzing large data sets

With YARN as the architectural center of ApacheTM Hadoop, multiple data access engines such as Apache Pig interact with data stored in the cluster. Apache Pig allows Apache Hadoop users to write complex MapReduce transformations using a simple scripting language called Pig Latin. Pig translates the Pig Latin script into MapReduce so that it can be executed within YARN for access to a single dataset stored in the Hadoop Distributed File System (HDFS).

**WHAT PIG DOES**

Pig was designed for performing a long series of data operations, making it ideal for three categories of Big Data jobs:

* **Extract-transform-load (ETL)** data pipelines,
* **Research** on raw data, and
* **Iterative data processing**



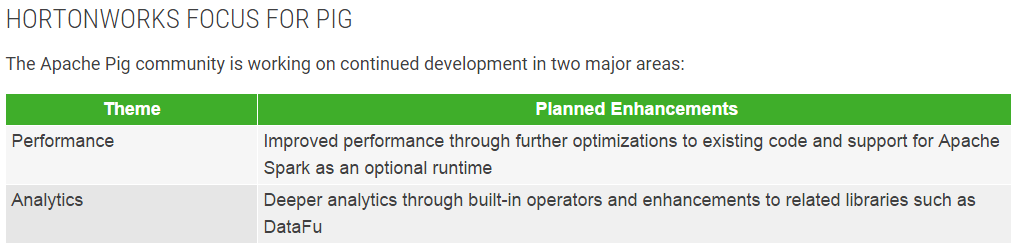
**HOW PIG WORKS**

Pig runs on Apache Hadoop YARN and makes use of MapReduce and the Hadoop Distributed File System (HDFS). The language for the platform is called Pig Latin, which abstracts from the Java MapReduce idiom into a form similar to SQL. While SQL is designed to query the data, Pig Latin allows you to write a data flow that describes how your data will be transformed (such as aggregate, join and sort).

Since Pig Latin scripts can be graphs (instead of requiring a single output) it is possible to build complex data flows involving multiple inputs, transforms, and outputs. Users can extend Pig Latin by writing their own functions, using Java, Python, Ruby, or other scripting languages. Pig Latin is sometimes extended using UDFs (User Defined Functions), which the user can write in any of those languages and then call directly from the Pig Latin.

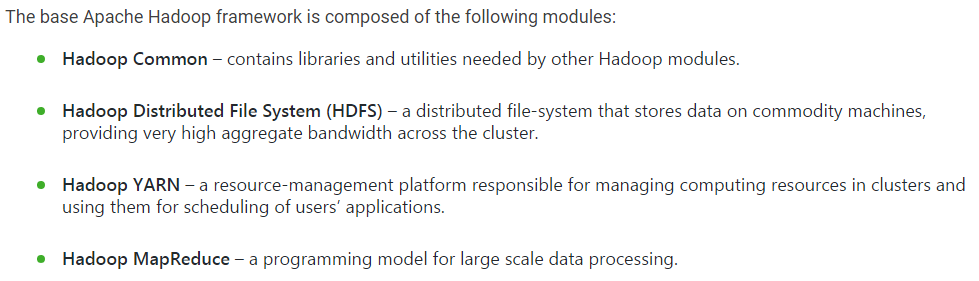
The user can run Pig in two modes, using either the “pig” command or the “java” command:

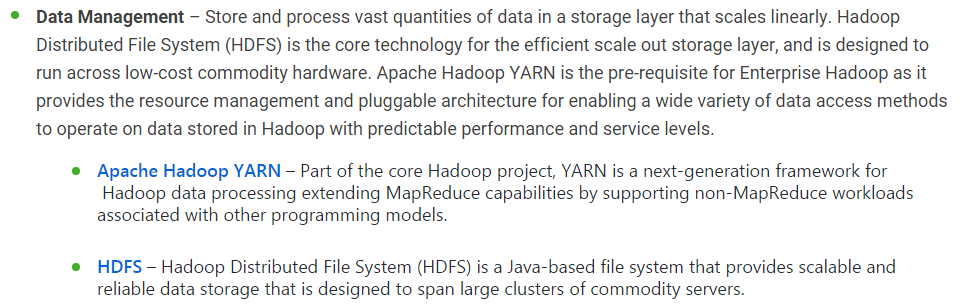
* **MapReduce Mode.** This is the default mode, which requires access to a Hadoop cluster.
* **Local Mode.** With access to a single machine, all files are installed and run using a local host and file system.



**HDP 2.4 – Pig 0.15.0**

* Pig on Tez stablization
* Improved Tez auto-parallelism
* Ability to invoke Hive UDFs from Pig





* [**Apache Hive**](https://hortonworks.com/hadoop/hive) – Built on the MapReduce framework, Hive is a data warehouse that enables easy data summarization and ad-hoc queries via an SQL-like interface for large datasets stored in HDFS.
* [**Apache Pig**](https://hortonworks.com/hadoop/pig) – A platform for processing and analyzing large data sets. Pig consists of a high-level language (Pig Latin) for expressing data analysis programs paired with the MapReduce framework for processing these programs.
* [**MapReduce**](https://hortonworks.com/hadoop/mapreduce/) – MapReduce is a framework for writing applications that process large amounts of structured and unstructured data in parallel across a cluster of thousands of machines, in a reliable and fault-tolerant manner.
* [**Apache Spark**](https://hortonworks.com/hadoop/spark) – Spark is ideal for in-memory data processing. It allows data scientists to implement fast, iterative algorithms for advanced analytics such as clustering and classification of datasets.
* [**Apache Storm**](https://hortonworks.com/hadoop/storm) – Storm is a distributed real-time computation system for processing fast, large streams of data adding reliable real-time data processing capabilities to Apache Hadoop 2.x
* [**Apache HBase**](https://hortonworks.com/hadoop/hbase) – A column-oriented NoSQL data storage system that provides random real-time read/write access to big data for user applications.
* [**Apache Tez**](https://hortonworks.com/hadoop/tez) – Tez generalizes the MapReduce paradigm to a more powerful framework for executing a complex DAG (directed acyclic graph) of tasks for near real-time big data processing.
* [**Apache Kafka**](https://hortonworks.com/hadoop/kafka) – Kafka is a fast and scalable publish-subscribe messaging system that is often used in place of traditional message brokers because of its higher throughput, replication, and fault tolerance.
* [**Apache HCatalog**](https://hortonworks.com/hadoop/hcatalog) – A table and metadata management service that provides a centralized way for data processing systems to understand the structure and location of the data stored within Apache Hadoop.
* [**Apache Slider**](https://hortonworks.com/hadoop/slider) – A framework for deployment of long-running data access applications in Hadoop. Slider leverages YARN’s resource management capabilities to deploy those applications, to manage their lifecycles and scale them up or down.
* [**Apache Solr**](https://hortonworks.com/hadoop/solr) – Solr is the open source platform for searches of data stored in Hadoop. Solr enables powerful full-text search and near real-time indexing on many of the world’s largest Internet sites.
* [**Apache Mahout**](https://hortonworks.com/hadoop/mahout) – Mahout provides scalable machine learning algorithms for Hadoop which aids with data science for clustering, classification and batch based collaborative filtering.
* [**Apache Accumulo**](https://hortonworks.com/hadoop/accumulo) – Accumulo is a high performance data storage and retrieval system with cell-level access control. It is a scalable implementation of Google’s Big Table design that works on top of Apache Hadoop and Apache ZooKeeper.

**Operations** – Provision, manage, monitor and operate Hadoop clusters at scale.

* [**Apache Ambari**](https://hortonworks.com/hadoop/ambari) – An open source installation lifecycle management, administration and monitoring system for Apache Hadoop clusters.
* [**Apache Oozie**](https://hortonworks.com/hadoop/oozie) – Oozie Java Web application used to schedule Apache Hadoop jobs. Oozie combines multiple jobs sequentially into one logical unit of work.
* [**Apache ZooKeeper**](https://hortonworks.com/hadoop/zookeeper) – A highly available system for coordinating distributed processes. Distributed applications use ZooKeeper to store and mediate updates to important configuration information.

**Security** – Address requirements of Authentication, Authorization, Accounting and Data Protection. Security is provided at every layer of the Hadoop stack from HDFS and YARN to Hive and the other Data Access components on up through the entire perimeter of the cluster via Apache Knox.

* [**Apache Knox**](https://hortonworks.com/hadoop/knox) – The Knox Gateway (“Knox”) provides a single point of authentication and access for Apache Hadoop services in a cluster. The goal of the project is to simplify Hadoop security for users who access the cluster data and execute jobs, and for operators who control access to the cluster.
* [**Apache Ranger**](https://hortonworks.com/hadoop/ranger) – Apache Ranger delivers a comprehensive approach to security for a Hadoop cluster. It provides central security policy administration across the core enterprise security requirements of authorization, accounting and data protection.

An HDFS cluster is comprised of a **NameNode**, which manages the cluster metadata, and DataNodes that store the data. Files and directories are represented on the NameNode by inodes. Inodes record attributes like permissions, modification and access times, or namespace and disk space quotas.

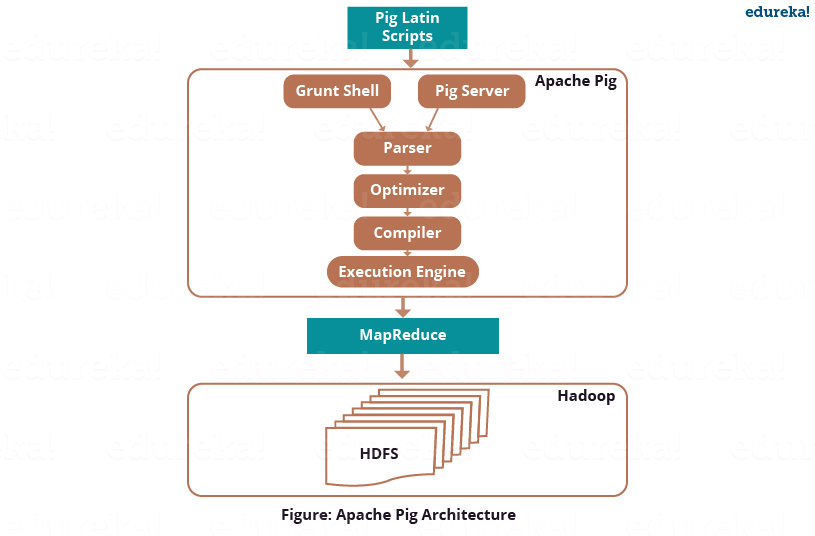
The file content is split into large blocks (typically 128 megabytes), and each block of the file is independently replicated at multiple DataNodes. The blocks are stored on the local file system on the DataNodes.

The Namenode actively monitors the number of replicas of a block. When a replica of a block is lost due to a DataNode failure or disk failure, the NameNode creates another replica of the block. The NameNode maintains the namespace tree and the mapping of blocks to DataNodes, holding the entire namespace image in RAM.

The NameNode does not directly send requests to DataNodes. It sends instructions to the DataNodes by replying to heartbeats sent by those DataNodes. The instructions include commands to:

* replicate blocks to other nodes,
* remove local block replicas,
* re-register and send an immediate block report, or
* shut down the node.

***PIG :***



There are three ways to execute the Pig script:

* ***Grunt Shell*:**This is Pig’s interactive shell provided to execute all Pig Scripts.
* ***Script File*:**Write all the Pig commands in a script file and execute the Pig script file. This is executed by the Pig Server.
* ***Embedded Script*:**If some functions are unavailable in built-in operators, we can programmatically create User Defined Functions to bring that functionalities using other languages like Java, Python, Ruby, etc. and embed it in Pig Latin Script file**.** Then, execute that script file.

**Parser**

The Parser does type checking and checks the syntax of the script. The parser outputs a DAG (directed acyclic graph). DAG represents the Pig Latin statements and logical operators. The logical operators are represented as the nodes and the data flows are represented as edges

**Optimizer**

Then the DAG is submitted to the optimizer. The Optimizer performs the optimization activities like split, merge, transform, and reorder operators  etc. This optimizer provides the automatic optimization feature to Apache Pig. The optimizer basically aims to reduce the amount of data in the pipeline at any instant of time while processing the extracted data,  and for that it performs functions like:

* *PushUpFilter*: If there are multiple conditions in the filter and the filter can be split, Pig splits the conditions and pushes up each condition separately. Selecting these conditions earlier, helps in reducing the number of records remaining in the pipeline.
* *PushDownForEachFlatten*: Applying flatten, which produces a cross product between a complex type such as a tuple or a bag and the other fields in the record, as late as possible in the plan. This keeps the number of records low in the pipeline.
* *ColumnPruner*: Omitting columns that are never used or no longer needed, reducing the size of the record. This can be applied after each operator, so that fields can be pruned as aggressively as possible.
* *MapKeyPruner*: Omitting map keys that are never used, reducing the size of the record.
* *LimitOptimizer*: If the limit operator is immediately applied after a load or sort operator, Pig converts the load or sort operator into a limit-sensitive implementation, which does not require processing the whole data set. Applying the limit earlier, reduces the number of records.

This is just a flavor of the optimization process. Over that it also performs **Join**, **Order By** and **Group By**functions.

To shutdown, automatic optimization, you can execute this command:

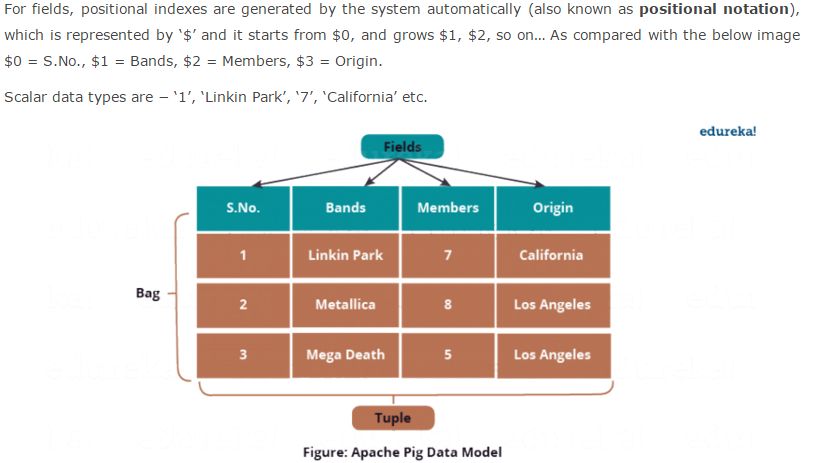
pig -optimizer\_off [opt\_rule | all ]

**Compiler**

After the optimization process, the compiler compiles the optimized code into a series of MapReduce jobs. The compiler is the one who is responsible for converting Pig jobs automatically into MapReduce jobs.

**Execution engine**

Finally, as shown in the figure: Apache Pig Architecture, these MapReduce jobs are submitted for execution to the execution engine. Then the MapReduce jobs are executed  and gives the required result. The results can be displayed on the screen using “**DUMP**” statement and can be stored in the HDFS using “**STORE**” statement.



**Bag**

A bag is a collection of a set of tuples and these tuples are subset of rows or entire rows of a table. A bag can contain duplicate tuples and it is not mandatory that they need to be unique.

The bag has a flexible schema i.e. tuples within the bag can have different number of fields. A bag can also have tuples with different data types.

A bag is represented by ‘{}’ symbol.

Example of a bag − **{(Linkin Park, 7, California), (Metallica, 8), (Mega Death, Los Angeles)}**

Outer bag or relation is nothing but a bag of tuples. Here relations are similar as relations in relational databases. To understand it better let us take an example:

**{(Linkin Park, California), (Metallica, Los Angeles), (Mega Death, Los Angeles)}**

This above bag explains the relation between the *Band* and their place of *Origin*.

On the other hand, an inner bag contains a bag inside a tuple. For Example, if we sort *Band* tuples based on*Band’s Origin*, we will get:

(Los Angeles, **{(Metallica, Los Angeles), (Mega Death, Los Angeles)}**)

(California,**{(Linkin Park, California)}**)

**Maps** are represented by ‘[]’ symbol and key-value are separated by ‘#’ symbol, as you can see in the above image.

Example of maps− [band#Linkin Park, members#7 ], [band#Metallica, members#8 ]

**Schema** is optional in Pig Latin but Pig encourage you to use them whenever possible, as the error checking becomes efficient while parsing the script which results in efficient execution of program.

Few Points on Schema in Pig:

* If the schema only includes the field name, the data type of field is considered as byte array.
* If you assign a name to the field you can access the field by both, the field name and the positional notation, whereas if field name is missing we can only access it by the positional notation i.e. $ followed by the index number.
* If you perform any operation which is a combination of relations (like JOIN, COGROUP, etc.) and if any of the relation is missing schema, the resulting relation will have null schema.
* If the schema is null, Pig will consider it as byte array and the real data type of field will be determined dynamically.

**Relation**

A relation is a bag of tuples.

**Atom**

Any single value in Pig Latin, irrespective of their data, type is known as an **Atom**.

A piece of data or a simple atomic value is known as a **field**.