Pig After MapReduce: Simplifying Big Data Processing

Bridging the gap between complex programming and accessible big data analysis

Recap: What is MapReduce?

A programming model for processing large data sets with a distributed algorithm on a cluster:

- Two main phases: **Map** (filtering & sorting) and **Reduce** (aggregation & summarization)
- Enables parallel processing across thousands of machines
- Powerful but requires **writing complex Java code and managing detailed job configurations**
- Handles fault tolerance and data distribution automatically

Challenges with MapReduce

Steep Learning Curve

Requires deep Java knowledge and understanding of distributed computing concepts

Even simple tasks require substantial code and configuration

Development Complexity

Difficult to write, debug, and maintain large MapReduce jobs

Testing requires cluster setup or simulation environments

Slow Iteration

Long development cycles slow down data analysis and prototyping

Each change requires recompilation and full job execution

Accessibility Barriers

Excludes analysts and data scientists without programming expertise

Creates bottlenecks when engineering resources are limited

Apache Pig: The Easy MapReduce

- Developed by Yahoo! and a top level Apache project
- Immediately makes data on a cluster available to non-Java programmers **via Pig Latin – a dataflow language**
- Interprets **Pig Latin** and generates **MapReduce** jobs that run on the cluster
- Enables easy data summarization, ad-hoc reporting and querying, and analysis of large volumes of data
- **Pig interpreter runs on a client machine** – no administrative overhead required

Apache Pig: The Easy MapReduce

- **Pig is a client application**
  - No cluster software is required
- **Interprets Pig Latin scripts to MapReduce jobs**
  - Parses Pig Latin scripts
  - Performs optimization
  - Creates execution plan
- **Submits MapReduce jobs to the cluster**

Apache Pig: The Easy MapReduce

High-level Platform

Pig Latin language designed specifically for data flow programming

Abstraction Layer

Hides MapReduce complexity, automatically compiles into optimized jobs

Developer Friendly

Enables concise, readable transformations with less code

Rich Data Types

Supports tuples, bags, maps for flexible data modeling

Advantages of Using Pig Over Raw MapReduce

10x

Less Code

Typical Pig scripts require 10x fewer lines of code than equivalent Java MapReduce

5x

Faster Development

Development cycles are typically 5x faster, allowing rapid prototyping and iteration

3x

Fewer Bugs

Less code means fewer bugs and maintenance issues in production pipelines

Additional Benefits

- Built-in optimizations reduce runtime
- Multi-query execution combines operations into fewer jobs
- Schema-on-read flexibility for semi-structured data

Extensibility

- User Defined Functions (UDFs) in Java, Python, JavaScript
- Custom load/store functions for any data format
- Integration with existing MapReduce code when needed

How Pig Connects to MapReduce

Script Creation

Developer writes concise Pig Latin script focusing on the data transformation logic

Parsing & Optimization

Pig engine parses script into a Directed Acyclic Graph (DAG) and applies optimizations

MapReduce Compilation

Optimized logical plan is compiled into one or more efficient MapReduce jobs

Execution Management

Pig handles job orchestration, data movement, and parallel execution automatically

Real-World Use Cases of Pig

Web Log Analysis

Parsing, filtering, and aggregating user clicks and session data

Example: Yahoo! processes 500+ TB of data daily with Pig

ETL Pipelines

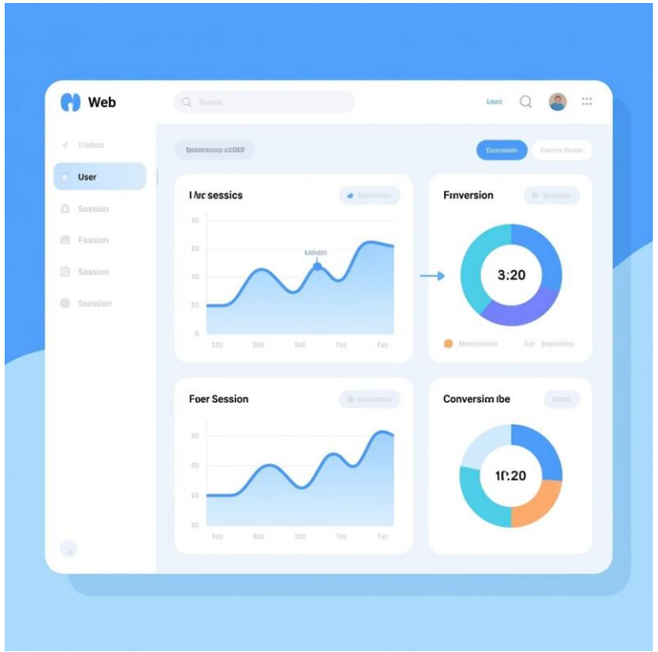
Transforming semi-structured data into structured formats

Example: LinkedIn uses Pig for data cleansing and standardization

Data Preparation

Cleaning and filtering before feeding into Hive or ML workflows

Example: Twitter preprocesses analytics data with Pig



Companies leverage Pig to process petabytes of data daily, making it accessible to more

Visualizing Pig's Execution on Hadoop Cluster

Script Submission

Pig script is submitted to the cluster's resource manager

Job Scheduling

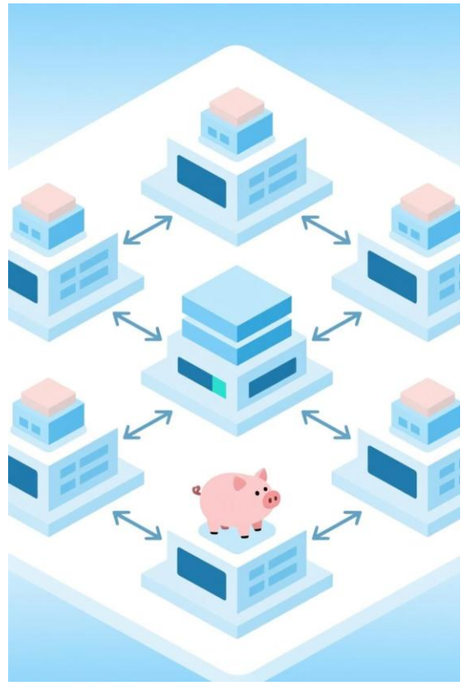
JobTracker/YARN schedules MapReduce tasks on appropriate nodes

Data Locality

Computation moves to data nodes to minimize network traffic

Fault Tolerance

Automatic retries and data replication ensure robustness



Pig Basic Terms

- **All data in Pig one of four types:**
  - An **Atom** is a simple data value - stored as a string but can be used as either a string or a number
  - A **Tuple i**s a data record consisting of a sequence of "fields"
    - Each field is a piece of data of any type (atom, tuple or bag)
  - A **Bag** is a set of tuples (also referred to as a ‘Relation’)
    - The concept of a “kind of a” table
  - A **Map** is a map from keys that are string literals to values that can be any data type
    - The concept of a hash map

Pig Capabilities

- **Support for**
  - Grouping
  - Joins
  - Filtering
  - Aggregation
- **Extensibility**
  - Support for User Defined Functions (UDF’s)
- **Leverages the same massive parallelism as native MapReduce**

Execution Modes

- **Pig has two execution modes**
  - Local Mode - all files are installed and run using your local host and file system
  - MapReduce Mode - all files are installed and run on a Hadoop cluster and HDFS installation
- **Interactive**
  - By using the Grunt shell by invoking Pig on the command line

    $ pig

    grunt>
- **Batch**
  - Run Pig in batch mode using Pig Scripts and the "pig" command

    $ pig –f id.pig –p <param>=<value> ...

Pig Latin

- **Pig Latin scripts are generally organized as follows**
  - A LOAD statement reads data
  - A series of “transformation” statements process the data
  - A STORE statement writes the output to the filesystem
    - A DUMP statement displays output on the screen
- **Logical vs. physical plans:**
  - All statements are stored and validated as a logical plan
  - Once a STORE or DUMP statement is found the logical plan is executed

Example Pig Script

```
-- Load the content of a file into a pig bag named ‘input_lines’
input_lines = LOAD 'CHANGES.txt' AS (line:chararray);

-- Extract words from each line and put them into a pig bag named ‘words’
words = FOREACH input_lines GENERATE FLATTEN(TOKENIZE(line)) AS word;

-- filter out any words that are just white spaces
filtered_words = FILTER words BY word MATCHES '\\w+';

-- create a group for each word
word_groups = GROUP filtered_words BY word;

-- count the entries in each group
word_count = FOREACH word_groups GENERATE COUNT(filtered_words) AS count, group AS word;

-- order the records by count
ordered_word_count = ORDER word_count BY count DESC;

-- Store the results ( executes the pig script )
STORE ordered_word_count INTO 'output’;
```

Basic “grunt” Shell Commands

- **Help is available**

```
$ pig -h
```

- **Pig supports HDFS commands**

```
grunt> pwd
```

  - put, get, cp, ls, mkdir, rm, mv, etc.

About Pig Scripts

- Pig Latin statements grouped together in a file
- Can be run from the command line or the shell
- Support parameter passing
- Comments are supported
  - Inline comments '--'
  - Block comments /* */

Simple Data Types

Type	Description
int	4-byte integer
long	8-byte integer
float	4-byte (single precision) floating point
double	8-byte (double precision) floating point
bytearray	Array of bytes; blob
chararray	String (“hello world”)
boolean	True/False (case insensitive)
datetime	A date and time
biginteger	Java BigInteger
bigdecimal	Java BigDecimal

Complex Data Types

Type	Description
Tuple	Ordered set of fields (a “row / record”)
Bag	Collection of tuples (a “resultset / table”)
Map	A set of key-value pairs Keys must be of type chararray

Pig Data Formats

- **BinStorage**
  - Loads and stores data in machine-readable (binary) format
- **PigStorage**
  - Loads and stores data as structured, field delimited text files
- **TextLoader**
  - Loads unstructured data in UTF-8 format
- **PigDump**
  - Stores data in UTF-8 format
- **YourOwnFormat!**
  - via UDFs

Loading Data Into Pig

- **Loads data from an HDFS file**

```
var = LOAD 'employees.txt';
var = LOAD 'employees.txt' AS (id, name, salary);
var = LOAD 'employees.txt' using PigStorage()
           AS (id, name, salary);
```

- **Each LOAD statement defines a new bag**
  - Each bag can have multiple elements (atoms)
  - Each element can be referenced by name or position ($*n*)
- **A bag is immutable**
- **A bag can be aliased and referenced later**

Input And Output

- **STORE**
  - Writes output to an HDFS file in a specified directory

    grunt> STORE processed INTO 'processed_txt';
    - Fails if directory exists
    - Writes output files, part-[m|r]-xxxxx, to the directory
  - PigStorage can be used to specify a field delimiter
- **DUMP**
  - Write output to screen

    grunt> DUMP processed;

Relational Operators

- **FOREACH**
  - Applies expressions to every record in a bag
- **FILTER**
  - Filters by expression
- **GROUP**
  - Collect records with the same key
- **ORDER BY**
  - Sorting
- **DISTINCT**
  - Removes duplicates

FOREACH . . .GENERATE

- **Use the FOREACH …GENERATE operator to work with rows of data, call functions, etc.**
- **Basic syntax:**

  alias2 = FOREACH alias1 GENERATE expression;
- **Example:**

  DUMP alias1;

  (1,2,3) (4,2,1) (8,3,4) (4,3,3) (7,2,5) (8,4,3)

  alias2 = FOREACH alias1 GENERATE col1, col2;

  DUMP alias2;

  (1,2) (4,2) (8,3) (4,3) (7,2) (8,4)

FILTER. . .BY

- **Use the FILTER operator to restrict tuples or rows of data**
- **Basic syntax:**

```
alias2 = FILTER alias1 BY expression;
```

- **Example:**

```
DUMP alias1;
(1,2,3) (4,2,1) (8,3,4) (4,3,3) (7,2,5) (8,4,3)
alias2 = FILTER alias1 BY (col1 == 8) OR (NOT (col2+col3 >
   col1));
DUMP alias2;
(4,2,1) (8,3,4) (8,4,3)
```

GROUP. . .ALL

- **Use the GROUP…ALL operator to group data**
  - Use GROUP when only one relation is involved
  - Use COGROUP with multiple relations are involved
- **Basic syntax:**

```
alias2 = GROUP alias1 ALL;
```

- **Example:**

```
DUMP alias1;
(John,18,4.0F) (Mary,19,3.8F) (Bill,20,3.9F) (Joe,18,3.8F)
alias2 = GROUP alias1 BY col2;
DUMP alias2;
(18,{(John,18,4.0F),(Joe,18,3.8F)})
(19,{(Mary,19,3.8F)})
(20,{(Bill,20,3.9F)})
```

ORDER. . .BY

- **Use the ORDER…BY operator to sort a relation based on one or more fields**
- **Basic syntax:**

```
alias = ORDER alias BY field_alias [ASC|DESC];
```

- **Example:**

```
DUMP alias1;
(1,2,3) (4,2,1) (8,3,4) (4,3,3) (7,2,5) (8,4,3)
alias2 = ORDER alias1 BY col3 DESC;
DUMP alias2;
(7,2,5) (8,3,4) (1,2,3) (4,3,3) (8,4,3) (4,2,1)
```

DISTINCT. . .

- **Use the DISTINCT operator to remove duplicate tuples in a relation.**
- **Basic syntax:**

```
alias2 = DISTINCT alias1;
```

- **Example:**

```
DUMP alias1;
(8,3,4) (1,2,3) (4,3,3) (4,3,3) (1,2,3)
alias2= DISTINCT alias1;
DUMP alias2;
(8,3,4) (1,2,3) (4,3,3)
```

Relational Operators

- **INNER JOIN**
  - Used to perform an inner join of two or more relations based on common field values
- **OUTER JOIN**
  - Used to perform left, right or full outer joins
- **SPLIT**
  - Used to partition the contents of a relation into two or more relations
- **SAMPLE**
  - Used to select a random data sample with the stated sample size
- **FLATTEN**
  - Used to un-nest tuples as well as bags

INNER JOIN. . .

- **Use the JOIN operator to perform an inner, equi-join join of two or more relations based on common field values**
- **The JOIN operator always performs an inner join**
- **Inner joins ignore null keys**
  - Filter null keys before the join
- **JOIN and COGROUP operators perform similar functions**
  - JOIN creates a flat set of output records
  - COGROUP creates a nested set of output records

INNER JOIN Example

```
DUMP Alias1;
    (1,2,3)
    (4,2,1)
    (8,3,4)
    (4,3,3)
    (7,2,5)
    (8,4,3)
DUMP Alias2;
    (2,4)
    (8,9)
    (1,3)
    (2,7)
    (2,9)
    (4,6)
    (4,9)
```

```
Join Alias1 by Col1 to Alias2 by Col1
    Alias3 = JOIN Alias1 BY Col1,
    Alias2 BY Col1;


Dump Alias3;
    (1,2,3,1,3)
    (4,2,1,4,6)
    (4,3,3,4,6)
    (4,2,1,4,9)
    (4,3,3,4,9)
    (8,3,4,8,9)
    (8,4,3,8,9)
```

OUTER JOIN. . .

- **Use the OUTER JOIN operator to perform left, right, or full outer joins**
  - Pig Latin syntax closely adheres to the SQL standard
- **The keyword OUTER is optional**
  - keywords LEFT, RIGHT and FULL will imply left outer, right outer and full outer joins respectively
- **Outer joins will only work provided the relations which need to produce nulls (in the case of non-matching keys) have schemas**
- **Outer joins will only work for two-way joins**
  - To perform a multi-way outer join perform multiple two-way outer join statements

User-Defined Functions

- **Natively written in Java, packaged as a jar file**
  - Other languages include Jython, JavaScript, Ruby, Groovy, and Python
- **Register the jar with the REGISTER statement**
- **Optionally, alias it with the DEFINE statement**

```
REGISTER /src/myfunc.jar;
A = LOAD 'students';
B = FOREACH A GENERATE myfunc.MyEvalFunc($0);
```

- http://pig.apache.org

Connecting Apache Hive with Pig

A comprehensive guide to integrating Hadoop's powerful data processing tools

What is Apache Hive?

Apache Hive serves as a powerful **data warehousing infrastructure** built on top of the Hadoop ecosystem. It enables:

- SQL-like querying (HiveQL) over massive datasets stored in HDFS
- Schema-on-read capabilities for flexible data processing
- Batch processing and large-scale analytics workloads

Hive's Architecture

User Interface

Command Line Interface (CLI), Web UI (HWI), JDBC/ODBC drivers for application connectivity

Metastore

Central repository that stores metadata about Hive tables, partitions, and schemas

Query Processor

Parses, plans, optimizes, and executes HiveQL queries by converting them to MapReduce/Tez/Spark jobs

Execution Engine

MapReduce, Tez, or Spark processes that execute the compiled HiveQL statements

Why Connect Hive with Pig?

Complementary Strengths

Pig excels at ETL operations and data transformations, while Hive provides SQL-like analytics and warehousing capabilities

Data Processing Pipeline

Use Pig for complex data preparation, then leverage Hive for structured querying and reporting

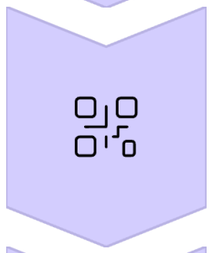
Different Programming Models

Pig's procedural approach suits developers comfortable with data pipelines, while Hive's declarative SQL appeals to analysts

Connecting Hive with Pig: Methods

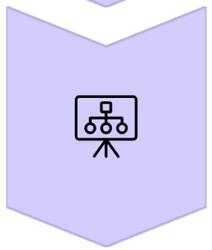
Shared Storage

Both access the same HDFS storage. Pig processes raw data and outputs to a location that Hive tables can reference



HCatalog Integration

Use HCatalog to share schema information between Pig and Hive, allowing Pig to read/write Hive tables directly



Workflow Orchestration

Use Apache Oozie or Airflow to coordinate Pig and Hive jobs in an integrated data pipeline

Key Features of Apache Hive

HiveQL

SQL-like query language that supports DDL, DML operations, and complex analytical queries

Schema Management

Centralized metastore maintains table definitions, partitioning schemes, and storage formats

Flexible Storage

Support for various file formats (ORC, Parquet, Avro) and compression algorithms

Optimizations

Cost-based optimizer, vectorized query execution, and predicate pushdown for performance

Hive Data Types and Table Types

Data Types

- **Primitive:** INT, BIGINT, STRING, FLOAT, DOUBLE, BOOLEAN, DATE
- **Complex:** ARRAY, MAP, STRUCT, UNION

Table Types

- **Managed Tables:** Hive controls data lifecycle
- **External Tables:** Data exists independently

Storage Organization

- **Partitioning:** Divides tables by column values for query efficiency
- **Bucketing:** Hash-based data distribution for join optimizations
- **Storage Formats:** TextFile, SequenceFile, ORC, Parquet, Avro

Hive Query Processing

1. **Parse Query**

   HiveQL statement is parsed into an Abstract Syntax Tree (AST)

2. **Semantic Analysis**

   Type checking and schema resolution against the metastore

3. **Logical Plan Generation**

   Create operators that represent data transformations

4. **Optimization**

   Apply rule-based and cost-based optimizations to improve execution plan

5. **Physical Plan Generation**

   Convert logical operators to physical execution tasks (MapReduce/Tez/Spark)

6. **Execution**

   Submit tasks to execution engine and monitor progress

Best Practices for Hive-Pig Integration

Use appropriate tool for the job

Pig for complex ETL, Hive for SQL analytics and reporting

Leverage HCatalog

Share metadata and schema information between systems

Optimize file formats

Use columnar formats like ORC or Parquet for better performance

Partition wisely

Design shared data partitioning that benefits both systems

Pig	Hive
Pig operates on the client side of a cluster.	Hive operates on the server side of a cluster.
Pig uses pig-latin language.	Hive uses HiveQL language.
Pig is a Procedural Data Flow Language.	Hive is a Declarative SQLish Language.
It was developed by Yahoo.	It was developed by Facebook.
It is used by Researchers and Programmers.	It is mainly used by Data Analysts.
It is used to handle structured and semi-structured data.	It is mainly used to handle structured data.

Pig	Hive
It is used for programming.	It is used for creating reports.
Pig scripts end with .pig extension.	In HIve, all extensions are supported.
It does not support partitioning.	It supports partitioning.
It loads data quickly.	It loads data slowly.
It does not support JDBC.	It supports JDBC.
It does not support ODBC.	It supports ODBC.
Pig does not have a dedicated metadata database.	Hive makes use of the exact variation of dedicated SQL-DDL language by defining tables beforehand.

Pig	Hive
It supports Avro file format.	It does not support Avro file format.
Pig is suitable for complex and nested data structures.	Hive is suitable for batch-processing OLAP systems.
Pig does not support schema to store data.	Hive supports schema for data insertion in tables.
It is very easy to write UDFs to calculate matrices.	It does support UDFs but is much hard to debug.