

Introduction to Pig and Pig Latin

What is Pig and Pig Latin?

Pig is an open-source high level data flow system. It provides a simple language called Pig Latin, for queries and data manipulation. Pig Latin high level operations are compiled in to MapReduce jobs that run on Hadoop.

Pig was developed by Yahoo! and later contributed to the Apache Software Foundation, where it became an Apache top-level project.

The primary goal of Apache Pig is to provide a simple and flexible platform for processing and analysing large datasets in Hadoop. Pig allows users to express complex data transformations using a scripting language called Pig Latin, which is designed to be easy to understand and use. Pig Latin scripts are then executed on a Hadoop cluster, taking advantage of its distributed processing capabilities.

Pig is made up of two pieces:

- The language used to express data flows, called Pig Latin.
- The execution environment to run Pig Latin programs. There are currently two environments: local execution in a single JVM and distributed (MapReduce) execution on a Hadoop cluster.

Here are some key components and concepts of Apache Pig:

Pig Latin:

Pig Latin is a high-level scripting language used for expressing data transformations. It abstracts the complexity of writing MapReduce programs, making it more accessible to users who may not have a deep understanding of distributed computing concepts.

Data Flow Language: Pig Latin scripts describe a series of data transformations, forming a directed acyclic graph (DAG) of operations. This data flow language makes it easy to express complex data processing tasks.

Installing and Running Pig

Pig runs as a client-side application. Even if you want to run Pig on a Hadoop cluster, there is nothing extra to install on the cluster: Pig launches jobs and interacts with HDFS (or other Hadoop filesystems) from your workstation.

Download a stable release from <http://pig.apache.org/releases.html>, and unpack the tarball in a suitable place on your workstation:

```
% tar xzf pig-x.y.z.tar.gz
```

It's convenient to add Pig's binary directory to your command-line path. For example:

```
% export PIG_HOME=~/.sw/pig-x.y.z
% export PATH=$PATH:$PIG_HOME/bin
```

You also need to set the `JAVA_HOME` environment variable to point to a suitable Java installation.

Execution Types:

In Apache Pig, there are two primary execution types:


- Local mode
- MapReduce mode

Local Mode:

Description: In local mode, Pig runs on a single machine using a local file system instead of the Hadoop Distributed File System (HDFS).

Use Case: Local mode is suitable for small-scale data processing tasks during development, testing, or debugging. It allows users to quickly iterate and experiment with Pig scripts on a small dataset without the need for a Hadoop cluster.

Execution Command:



```
bash
pig -x local script.pig
```

MapReduce Mode:

Description: In MapReduce mode, Pig takes advantage of the Hadoop MapReduce framework to process data in a distributed and parallelized manner across a Hadoop cluster.

Use Case: MapReduce mode is designed for large-scale data processing tasks where data is distributed across multiple nodes in a Hadoop cluster. It provides scalability and fault tolerance, making it suitable for handling massive datasets.

Execution Command:

Copy code

```
pig script.pig
```

Running Pig Programs

There are three ways of executing Pig programs, all of which work in both local and MapReduce mode.

Script

Create a Pig Latin script using a text editor. Save it with a ".pig" extension. The script will contain a series of Pig Latin statements that define the data loading, transformation, and storage operations.

Example (sample.pig):

pig

Copy code

```
-- Sample Pig Latin script
data = LOAD 'input.txt' USING PigStorage(',') AS (id:int, name:chararray, age:int);
filtered_data = FILTER data BY age > 25;
grouped_data = GROUP filtered_data BY name;
result = FOREACH grouped_data GENERATE group, AVG(filtered_data.age) AS avg_age;
STORE result INTO 'output';
```

Execute Pig Script:

Local Mode:

For testing and development on a single machine.

```
bash
pig -x local sample.pig
```

MapReduce Mode:

For large-scale processing on a Hadoop cluster.

```
bash
pig sample.pig
```

Grunt:

Pig provides an interactive shell called Grunt, which allows users to interactively develop and test Pig Latin scripts before executing them on a Hadoop cluster.

Execute Pig in local mode:

Local Mode: Run the following command to start the Grunt shell in local mode:

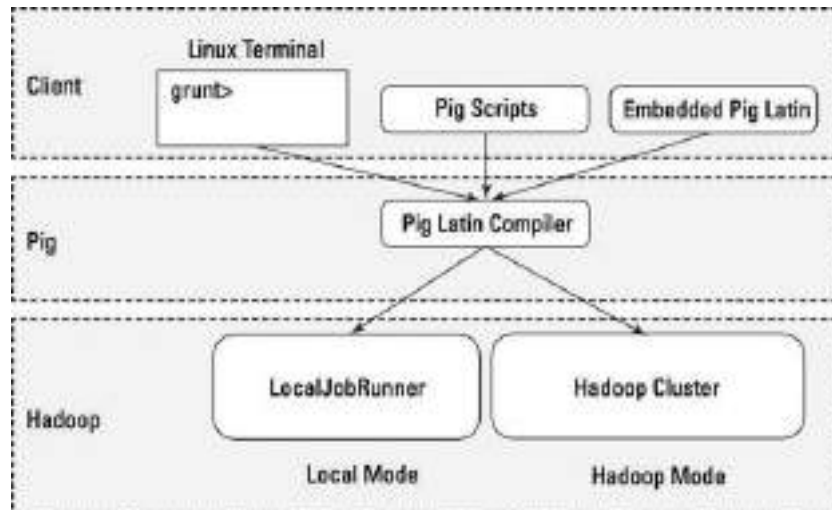
```
bash
pig -x local
```

MapReduce Mode: Run the following command to start the Grunt shell in MapReduce mode:

```
☐ Copy code
pig
```

Embedded Mode

You can run Pig programs from Java using the PigServer class, much like you can use JDBC to run SQL programs from Java. For programmatic access to Grunt, use PigRunner.



Grunt:

Grunt is the interactive shell and command-line interface provided by Apache Pig for developing, testing, and executing Pig Latin scripts. It allows users to interactively work with Pig without the need to write a full script file. The Grunt shell provides a convenient environment for exploring data, testing Pig Latin commands, and incrementally developing Pig scripts. Here are the key features and usage details of the Grunt shell:

- Grunt has line-editing facilities like those found in GNU Readline.
- For instance, the Ctrl-E key combination will move the cursor to the end of the line.
- Grunt remembers command history, too, and you can recall lines in the history buffer using Ctrl-P or Ctrl-N (for previous and next), or equivalently, the up or down cursor keys.
- Another handy feature is Grunt's completion mechanism, which will try to complete Pig Latin keywords and functions when you press the Tab key. For example, consider the following incomplete line:
`grunt> a = foreach b ge`
- If you press the Tab key at this point, `ge` will expand to `generate`, a Pig Latin keyword:
`grunt> a = foreach b generate`

- You can get a list of commands using the help command.
- When you've finished your Grunt session, you can exit with the quit command.

Pig Latin

This section gives an informal description of the syntax and semantics of the Pig Latin programming language.

Structure of Pig Latin:

- A Pig Latin program consists of a collection of statements.
- A statement can be thought of as an operation or a command.
- For example, a GROUP operation is a type of statement:
grouped_records = GROUP records BY year;
- For example, The command to list the files in a Hadoop filesystem is another example of a statement:
ls /
- Statements are usually terminated with a semicolon.
- Operations needs to be terminated by semicolon whereas Hadoop commands, as well as the diagnostic operators in pig latin need not to be ended with a semicolon.
- Pig Latin has two forms of comments.
 - Double hyphens are used for single-line comments.
-- My program
DUMP A; -- What's in A?
 - C-style comments are more flexible since they delimit the beginning and end of the comment block with /* and */ markers. They can span lines or be embedded in a single line:
/*
* Description of my program spanning
* multiple lines.
*/
A = LOAD 'input/pig/join/A';
B = LOAD 'input/pig/join/B';
C = JOIN A BY \$0, /* ignored */ B BY \$1;
DUMP C;
- Pig Latin has a list of keywords that have a special meaning in the language and cannot be used as identifiers.

- These include the operators (LOAD, ILLUSTRATE), commands (cat, ls), expressions (matches, FLATTEN), and functions (DIFF, MAX).
- Pig Latin has mixed rules on case sensitivity.
 - Operators and commands are not case sensitive.
 - Aliases and function names are case sensitive.

Statements:

- As a Pig Latin program is executed, each statement is parsed in turn.
- If there are syntax errors or other (semantic) problems, such as undefined aliases, the interpreter will halt and display an error message.
- The interpreter builds a logical plan for every relational operation, which forms the core of a Pig Latin program.
- If no errors, the statement is added to the logical plan for the program so far, and then the interpreter moves on to the next statement.
- It's important to note that no data processing takes place while the logical plan of the program is being constructed.
- For Example:


```
data = LOAD 'input.txt' USING PigStorage(',') AS (col1:int, col2:int, col3:double);
```
- When the Pig Latin interpreter sees the first line containing the LOAD statement, it confirms that it is syntactically and semantically correct and adds it to the logical plan, but it does not load the data from the file.
- The trigger for Pig to start execution is the DUMP statement.
- At that point, the logical plan is compiled into a physical plan and executed.
- The physical plan that Pig prepares is a series of MapReduce jobs, which in local mode Pig runs in the local JVM and in MapReduce mode Pig runs on a Hadoop cluster.
- The relational operators that can be a part of a logical plan in Pig are summarized in below table.

Table. Pig Latin relational operators

Category	Operator	Description
Loading and storing	LOAD	Loads data from the filesystem or other storage into a relation
	STORE	Saves a relation to the filesystem or other storage
	DUMP (\d)	Prints a relation to the console
Filtering	FILTER	Removes unwanted rows from a relation
	DISTINCT	Removes duplicate rows from a relation
	FOREACH...GENERATE	Adds or removes fields to or from a relation
	MAPREDUCE	Runs a MapReduce job using a relation as input
	STREAM	Transforms a relation using an external program
	SAMPLE	Selects a random sample of a relation
	ASSERT	Ensures a condition is true for all rows in a relation; otherwise, fails
Grouping and joining	JOIN	Joins two or more relations
	COGROUP	Groups the data in two or more relations
	GROUP	Groups the data in a single relation
	CROSS	Creates the cross product of two or more relations
	CUBE	Creates aggregations for all combinations of specified columns in a relation
Sorting	ORDER	Sorts a relation by one or more fields
	RANK	Assign a rank to each tuple in a relation, optionally sorting by fields first
	LIMIT	Limits the size of a relation to a maximum number of tuples
Combining and splitting	UNION	Combines two or more relations into one
	SPLIT	Splits a relation into two or more relations

- There are other types of statements that are not added to the logical plan. For example, the diagnostic operators—DESCRIBE, EXPLAIN, and ILLUSTRATE—are provided to allow the user to interact with the logical plan for debugging purposes

Table. Pig Latin diagnostic operators

Operator (Shortcut)	Description
DESCRIBE (\de)	Prints a relation's schema
EXPLAIN (\e)	Prints the logical and physical plans
ILLUSTRATE (\t)	Shows a sample execution of the logical plan, using a generated subset of the input

- Pig Latin also provides three statements—REGISTER, DEFINE, and IMPORT—that make it possible to incorporate macros and user-defined functions into Pig scripts.

Table. Pig Latin macro and UDF statements

Statement	Description
REGISTER	Registers a JAR file with the Pig runtime
DEFINE	Creates an alias for a macro, UDF, streaming script, or command specification
IMPORT	Imports macros defined in a separate file into a script

- Because they do not process relations, commands are not added to the logical plan; instead, they are executed immediately.
- Pig provides commands to interact with Hadoop filesystems and MapReduce, as well as a few utility commands.

Table. Pig Latin commands

Category	Command	Description
Hadoop filesystem	cat	Prints the contents of one or more files
	cd	Changes the current directory
	copyFromLocal	Copies a local file or directory to a Hadoop filesystem
	copyToLocal	Copies a file or directory on a Hadoop filesystem to the local filesystem
	cp	Copies a file or directory to another directory
	fs	Accesses Hadoop's filesystem shell
	ls	Lists files
	mkdir	Creates a new directory
	mv	Moves a file or directory to another directory
	pwd	Prints the path of the current working directory
	rm	Deletes a file or directory
	rmf	Forcibly deletes a file or directory (does not fail if the file or directory does not exist)
Hadoop MapReduce	kill	Kills a MapReduce job

Category	Command	Description
Utility	clear	Clears the screen in Grunt
	exec	Runs a script in a new Grunt shell in batch mode
	help	Shows the available commands and options
	history	Prints the query statements run in the current Grunt session
	quit (\q)	Exits the interpreter
	run	Runs a script within the existing Grunt shell
	set	Sets Pig options and MapReduce job properties
	sh	Runs a shell command from within Grunt

- There are two commands in above Table for running a Pig script, exec and run.
- The difference is that exec runs the script in batch mode in a new Grunt shell, so any aliases defined in the script are not accessible to the shell after the script has completed.
- On the other hand, when running a script with run, it is as if the contents of the script had been entered manually, so the command history of the invoking shell contains all the statements from the script.

Expressions:

An expression is something that is evaluated to yield a value. Expressions can be used in Pig as a part of a statement containing a relational operator. Pig has a rich variety of expressions, many of which will be familiar from other programming languages. They are listed in Table, with brief descriptions and examples.

Table. Pig Latin expressions

Category	Expressions	Description	Examples
Constant	Literal	Constant value (see also the "Literal example" column in Table 16-6)	1.0, 'a'
Field (by position)	$\$n$	Field in position n (zero-based)	$\$0$
Field (by name)	f	Field named f	year
Field (disambiguate)	$r::f$	Field named f from relation r after grouping or joining	A::year
Projection	$c.\$n, c.f$	Field in container c (relation, bag, or tuple) by position, by name	records. $\$0$, records.year
Map lookup	$m\#k$	Value associated with key k in map m	items#'Coat'
Cast	$(t) f$	Cast of field f to type t	(int) year
Arithmetic	$x + y, x - y$	Addition, subtraction	$\$1 + \$2, \$1 - \2
	$x * y, x / y$	Multiplication, division	$\$1 * \$2, \$1 / \2
	$x \% y$	Modulo, the remainder of x divided by y	$\$1 \% \2
	$+x, -x$	Unary positive, negation	+1, -1
Conditional	$x ? y : z$	Bincond/ternary; y if x evaluates to true, z otherwise	quality == 0 ? 0 : 1
	CASE	Multi-case conditional	CASE q WHEN 0 THEN 'good' ELSE 'bad' END

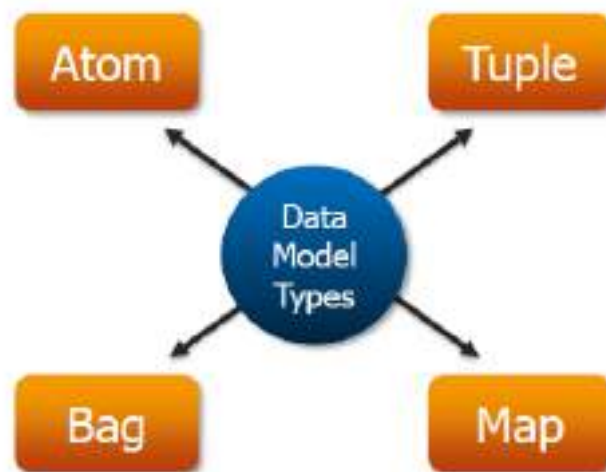
Category	Expressions	Description	Examples
Comparison	$x == y, x != y$	Equals, does not equal	quality == 0, temperature != 9999
	$x > y, x < y$	Greater than, less than	quality > 0, quality < 10
	$x >= y, x <= y$	Greater than or equal to, less than or equal to	quality >= 1, quality <= 9
	$x \text{ matches } y$	Pattern matching with regular expression	quality matches '[01459]'
	$x \text{ is null}$	Is null	temperature is null
	$x \text{ is not null}$	Is not null	temperature is not null
Boolean	$x \text{ OR } y$	Logical OR	$q == 0 \text{ OR } q == 1$
	$x \text{ AND } y$	Logical AND	$q == 0 \text{ AND } r == 0$
	NOT x	Logical negation	NOT $q \text{ matches '[01459]'}$
	IN x	Set membership	$q \text{ IN } (0, 1, 4, 5, 9)$
Functional	$fn(f1, f2, \dots)$	Invocation of function fn on fields $f1, f2$, etc.	isGood(quality)
Flatten	FLATTEN(f)	Removal of a level of nesting from bags and tuples	FLATTEN(group)

Types:

- Pig has a boolean type and six numeric types: int, long, float, double, bigint, and bigdecimal, which are identical to their Java counterparts.
- There is also a bytearray type, like Java's byte array type for representing a blob of binary data, and chararray, which, like java.lang.String, represents textual data in UTF-16 format.
- The datetime type is for storing a date and time with millisecond precision and including a time zone.
- Pig does not have types corresponding to Java's byte, short, or char primitive types.
- These are all easily represented using Pig's int type, or chararray for char.
- The Boolean, numeric, textual, binary, and temporal types are simple atomic types.
- Pig Latin also has three complex types for representing nested structures: tuple, bag, and map.

Table 16-6. Pig Latin types

Category	Type	Description	Literal example
Boolean	boolean	True/false value	true
Numeric	int	32-bit signed integer	1
	long	64-bit signed integer	1L
	float	32-bit floating-point number	1.0F
	double	64-bit floating-point number	1.0
	biginteger	Arbitrary-precision integer	'10000000000'
	bigdecimal	Arbitrary-precision signed decimal number	'0.11000100000000000000000001'
Text	chararray	Character array in UTF-16 format	'a'
Binary	bytearray	Byte array	Not supported
Temporal	datetime	Date and time with time zone	Not supported, use ToDate built-in function
Complex	tuple	Sequence of fields of any type	(1, 'pomegranate')
	bag	Unordered collection of tuples, possibly with duplicates	{(1, 'pomegranate'), (2)}
	map	Set of key-value pairs; keys must be character arrays, but values may be any type	['a' #'pomegranate']



Schemas:

In Apache Pig, a schema defines the structure of the data being processed. It specifies the field names and their corresponding data types in a relation. Schemas are essential for Pig to understand how to interpret and process the data. When you load data into Pig or define relations, you often provide a schema to describe the structure of the data.

Here are some key concepts related to schemas in Pig:

1. Defining a Schema:

- When loading data into Pig, you can define a schema using the AS clause.
- The syntax is (field_name:datatype, field_name:datatype, ...).

```
data = LOAD 'input.txt' USING PigStorage(',') AS (name:chararray, age:int, salary:double);
```

This example specifies a schema with three fields: name of type chararray, age of type int, and salary of type double.

2. Using Schema in FOREACH:

When processing data in a FOREACH statement, you can reference the fields using the schema.

```
transformed_data = FOREACH data GENERATE name, age * 2 AS doubled_age;
```

- Fields are assigned data types with the help of schemas.
- However it is not mandatory to assign schema always since If we don't assign data types, default type *bytearray* is assigned to fields and implicit conversions are applied to the fields depending on the context in which the field is being used.

```
data = LOAD 'input.txt' USING PigStorage(',') AS (name, age, salary);
```

- If a schema is defined as part of a load statement, the load function tries to assign the given schema. However, If the data does not conform to given schema, pig will generate a null value or an error.
- Example: Input data
4, BDHS

```
A = LOAD 'input' AS (sn:int, sub: int);
```

```
Dump A;
```

```
O/P : (4,)
```

In the above example the input data is 4,BDHS which is (int, chararray) but the schema of relation “A” is (int, int) so for the sub field its null value.

- In case explicit cast is not supported, pig will report an error
Example: You cannot cast a *chararray* to *int* in pig

```
A = LOAD 'xyz.txt' AS (name: chararray, age:int);
```

```
B = FOREACH A GENERATE (int)name;
```

error.....

- If Pig cannot resolve incompatible types through implicit casts, pig will report an error

Example: You cannot add *chararray* and *float*

```
A = LOAD 'edureka' AS (name: chararray, age: float);
```

```
B = FOREACH A GENERATE name + age;
```

error.....

Functions

Functions in Pig come in four types:

Eval function

- A function that takes one or more expressions and returns another expression.
- An example of a built-in eval function is MAX, which returns the maximum value of the entries in a bag.
- Some eval functions are aggregate functions, which means they operate on a bag of data to produce a scalar value; MAX is an example of an aggregate function.
- Furthermore, many aggregate functions are algebraic, which means that the result of the function may be calculated incrementally.

Filter function

- A special type of eval function that returns a logical Boolean result.
- As the name suggests, filter functions are used in the FILTER operator to remove unwanted rows.
- They can also be used in other relational operators that take Boolean conditions, and in general, in expressions using Boolean or conditional expressions.
- An example of a built-in filter function is IsEmpty, which tests whether a bag or a map contains any items.

Load function

- A function that specifies how to load data into a relation from external storage.

Store function

- A function that specifies how to save the contents of a relation to external storage.

Often, load and store functions are implemented by the same type. For example, PigStorage, which loads data from delimited text files, can store data in the same format.

Pig comes with a collection of built-in functions, a selection of which are listed in below Table.

Category	Function	Description
Eval	AVG	Calculates the average (mean) value of entries in a bag.
	CONCAT	Concatenates byte arrays or character arrays together.
	COUNT	Calculates the number of non-null entries in a bag.
	COUNT_STAR	Calculates the number of entries in a bag, including those that are null.
	DIFF	Calculates the set difference of two bags. If the two arguments are not bags, returns a bag containing both if they are equal; otherwise, returns an empty bag.
	MAX	Calculates the maximum value of entries in a bag.
	MIN	Calculates the minimum value of entries in a bag.
	SIZE	Calculates the size of a type. The size of numeric types is always 1; for character arrays, it is the number of characters; for byte arrays, the number of bytes; and for containers (tuple, bag, map), it is the number of entries.
	SUM	Calculates the sum of the values of entries in a bag.
	TOBAG	Converts one or more expressions to individual tuples, which are then put in a bag. A synonym for <code>()</code> .
	TOKENIZE	Tokenizes a character array into a bag of its constituent words.
	TOMAP	Converts an even number of expressions to a map of key-value pairs. A synonym for <code>[]</code> .
	TOP	Calculates the top <i>n</i> tuples in a bag.
	TOTUPLE	Converts one or more expressions to a tuple. A synonym for <code>{}</code> .
Filter	IsEmpty	Tests whether a bag or map is empty.
Load/Store	PigStorage	Loads or stores relations using a field-delimited text format. Each line is broken into fields using a configurable field delimiter (defaults to a tab character) to be stored in the tuple's fields. It is the default storage when none is specified. ⁴
	TextLoader	Loads relations from a plain-text format. Each line corresponds to a tuple whose single field is the line of text.

Macros:

In Apache Pig, macros are a way to define reusable pieces of Pig Latin code that can be invoked within Pig scripts. Macros help in modularizing Pig scripts, promoting code reuse, and improving script readability. Here's an overview of how macros work in Pig Latin:

Defining a Macro:

To define a macro, you use the `DEFINE` statement followed by the macro name and the Pig Latin code associated with the macro. The syntax is as follows:

```
pig Copy code  
  
DEFINE macro_name(arguments) RETURNS return_type {  
    -- Pig Latin code  
}
```

Here, `macro_name` is the name you give to your macro, `arguments` are the parameters the macro can take, and `return_type` is the type of data the macro returns.

Example: Simple Macro

Let's create a simple macro that takes two parameters and calculates their sum:

```
pig Copy code  
  
DEFINE add_numbers(a, b) RETURNS result {  
    $result = a + b;  
}
```

Using a Macro:

You can use a macro in your Pig script by calling it with the `INVOKE` statement. The syntax is:

```
pig Copy code  
  
result_relation = INVOKE macro_name(input_parameters);
```


Here's an example of using the `add_numbers` macro:

```
pig Copy code  
  
-- Using the add_numbers macro  
data = LOAD 'input.txt' AS (a:int, b:int);  
sum_result = INVOKE add_numbers(data.a, data.b);
```

Parameterized Macros:

Macros can take parameters, making them more flexible and reusable. You can specify parameters when defining the macro and pass values when invoking it.

```
pig Copy code  
  
DEFINE multiply_by_factor(input, factor) RETURNS result {  
    $result = FOREACH $input GENERATE $0 * $factor AS multiplied_value;  
}
```

Using Parameterized Macros:

```
pig Copy code  
  
-- Using the multiply_by_factor macro  
data = LOAD 'input.txt' AS (value:int);  
result = INVOKE multiply_by_factor(data, 2);
```

Benefits of Macros:

- **Code Reusability:** Macros allow you to define a piece of logic once and reuse it in multiple parts of your script.
- **Modularity:** By breaking down your script into smaller, modular pieces, it becomes easier to understand and maintain.
- **Parameterization:** Macros can accept parameters, enabling you to create more flexible and customizable code.
- **Abstraction:** Macros provide a level of abstraction, hiding the details of the implementation and focusing on the higher-level functionality.

Limitations:

- **No Return Statements:** Pig macros don't have explicit return statements. Instead, they generate code in the global namespace, and variables defined in a macro are accessible globally.
- **No Scope Isolation:** Macros share the same namespace, and variables defined in one macro can be accessed by other macros or the main script.
- **Code Expansion:** Macros are expanded at runtime, and their code is inserted into the script. This may lead to larger scripts if macros are heavily used.

Pig Latin Data Processing Operations

We will use these two Files for executing Pig Latin Operations

- 1) Create a document named pigdata using command ==> vim pigdata

1,2,3
4,2,1
8,3,4
4,3,3

- 2) Create a document named pigdata1 using command ==> vim pigdata1

1,2,3
4,2,1
4,3,4
4,3,4

- 3) Launch the grunt shell in the local mode using the command ==> pig -x local

Here are some common operations in Pig Latin:

Category Loading and Storing Data:

LOAD:

Syntax: LOAD 'path' [USING function] [AS schema];

Description: Loads data from the specified path into a relation. The optional USING clause can specify a loader function, and the optional AS clause can define the schema.

Example:

Load

```
grunt> a = load '/home/cloudera/data' using PigStorage(',') as(a1:int,a2:int,a3:int);  
grunt> b = load '/home/cloudera/data1' using PigStorage(',') as (a1:int,a2:int,a3:int);
```

STORE:

Syntax: STORE relation INTO 'output_path' [USING function];

Description: Stores the contents of a relation into the specified output path. The optional USING clause can specify a storage function.

Example:

Store

```
grunt> store a into '/home/cloudera/dataout' using PigStorage('.');  
grunt> cd /home/cloudera/dataout  
grunt> cat part-m-00000  
1.2.3  
4.2.1  
8.3.4  
4.3.3
```

DUMP:

Syntax: DUMP relation;

Description: Displays the contents of a relation on the console.

Example:

```
grunt> dump a;  
(1,2,3)  
(4,2,1)  
(8,3,4)  
(4,3,3)
```

```
grunt> dump b;  
(1,2,3)  
(4,2,1)  
(4,3,4)  
(3,2,1)
```

Category Combining and Splitting Data:**UNION:**

Syntax: UNION relation [, relation ...];

Description: Combines the records of multiple relations into a single relation.

Example:

```
Union  
grunt> c = union a,b;  
grunt> dump c;  
(1,2,3)  
(4,2,1)  
(4,3,4)  
(3,2,1)  
(1,2,3)  
(4,2,1)  
(8,3,4)  
(4,3,3)
```

SPLIT:

Syntax: SPLIT relation INTO destination1 IF condition1, destination2 IF condition2, ...;

Description: Splits a relation into multiple output relations based on specified conditions.

Example:

```
Split  
grunt> describe c;  
c: {a1: int,a2: int,a3: int}
```

```
grunt> split c into d if(a1>3),e if(a1>=1 and a1<=3);
grunt> dump d;
(4,2,1)
(8,3,4)
(4,3,3)
(4,2,1)
(4,3,4)
grunt> dump e;
(1,2,3)
(1,2,3)
(3,2,1)
```

Category Filtering Data :

FILTER:

Syntax: FILTER relation BY condition;

Description: Filters records in a relation based on the specified condition.

Example:

```
Filter
grunt> dump c;
(1,2,3)
(4,2,1)
(4,3,4)
(3,2,1)
(1,2,3)
(4,2,1)
(8,3,4)
(4,3,3)
grunt> i = filter c by $0 == 1 and $1 == 2;
grunt> dump i;
(1,2,3)
(1,2,3)
```

FOREACH:

Syntax: FOREACH relation GENERATE expression [AS alias, ...];

Description: Applies a transformation to each record in a relation and generates a new set of fields.

Example:

```
Foreach .... Generate
grunt> k = foreach j generate $0,$1,$2-1;
grunt> dump k;
(1,2,2)
```

(3,2,0)
(4,2,0)
(4,3,2)
(4,3,3)
(8,3,3)

DISTINCT:

Syntax: DISTINCT relation;

Description: Removes duplicate rows from a relation

Example:

Distinct
grunt> j = distinct c;
grunt> dump j;
(1,2,3)
(3,2,1)
(4,2,1)
(4,3,3)
(4,3,4)
(8,3,4)

Category Grouping and Joining Data :

GROUP:

Syntax: GROUP relation BY {field | expression} [PARALLEL n];

Description: Groups records in a relation based on the specified field or expression.

Example:

Group
grunt> dump c;
(1,2,3)
(4,2,1)
(4,3,4)
(3,2,1)
(1,2,3)
(4,2,1)
(8,3,4)
(4,3,3)
grunt> p = group c by \$0;
grunt> dump p;
(1,{{(1,2,3),(1,2,3)}})
(3,{{(3,2,1)}})
(4,{{(4,3,4),(4,2,1),(4,3,3),(4,2,1)}})
(8,{{(8,3,4)}})

Cogroup:

Syntax: COGROUP relation BY {fields} [, relation BY {fields} [, ...] INTO alias [, alias ...];

Description: Groups multiple relations by common fields into one or more aliases.

Example:

```
Co-Group
grunt> q = cogroup a by $0, b by $0;
grunt> dump q;
(1,{{(1,2,3)},{(1,2,3)}})
(3,{{(3,2,1)}})
(4,{{(4,3,3),(4,2,1)},{(4,3,4),(4,2,1)}})
(8,{{(8,3,4)},{}})
```

JOIN:

Syntax: JOIN relation BY (fields) [, relation BY (fields) [, ...]] [USING 'join_type'];

Description: Joins two or more relations based on common fields.

Example:

```
grunt> dump a;
(1,2,3)
(4,2,1)
(8,3,4)
(4,3,3)
grunt> dump b;
(1,2,3)
(4,2,1)
(4,3,4)
(3,2,1)
Join
grunt> l = join a by $0, b by $2;
grunt> dump l;
(1,2,3,3,2,1)
(1,2,3,4,2,1)
(4,3,3,4,3,4)
(4,2,1,4,3,4)
```

Left Outer Join

```
grunt> m = join a by $0 left Outer, b by $2;
grunt> dump m;
(1,2,3,3,2,1)
(1,2,3,4,2,1)
(4,3,3,4,3,4)
(4,2,1,4,3,4)
(8,3,4,,)
```

Right Outer Join

grunt> n = join a by \$0 right, b by \$2;

grunt> dump n;

(1,2,3,3,2,1)

(1,2,3,4,2,1)

(,,1,2,3)

(4,3,3,4,3,4)

(4,2,1,4,3,4)

CROSS:

Syntax: CROSS relation1, relation2 [, relation3, ...];

Description: Computes the cross product of two or more relations.

Example:

Cross

grunt> r = cross a,b;

grunt> dump r;

(4,3,3,3,2,1)

(4,3,3,4,3,4)

(4,3,3,4,2,1)

(4,3,3,1,2,3)

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

(8,3,4,4,3,4)

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

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

(4,2,1,3,2,1)

(4,2,1,4,3,4)

(4,2,1,4,2,1)

(4,2,1,1,2,3)

(1,2,3,3,2,1)

(1,2,3,4,3,4)

(1,2,3,4,2,1)

(1,2,3,1,2,3)

SAMPLE:

Syntax: SAMPLE relation n;

Description: Randomly samples n records from a relation.

CUBE:

Syntax: CUBE relation BY {field | expression} [, {field | expression} [, ...]];

Description: Generates all possible combinations of grouping values in a relation.

Category Sorting Data:

ORDER:

Syntax: ORDER relation BY {fields} [ASC | DESC] [PARALLEL n];

Description: Orders the records in a relation based on the specified fields.

Example:

Order

```
grunt> dump b;
```

```
(1,2,3)
```

```
(4,2,1)
```

```
(4,3,4)
```

```
(3,2,1)
```

```
grunt> f = order b by $0 DESC;
```

```
grunt> dump f;
```

```
(4,3,4)
```

```
(4,2,1)
```

```
(3,2,1)
```

```
(1,2,3)
```

```
grunt> g = order b by $0 ASC;
```

```
grunt> dump g;
```

```
(1,2,3)
```

```
(3,2,1)
```

```
(4,3,4)
```

```
(4,2,1)
```

LIMIT:

Syntax: LIMIT relation n;

Description: Limits the number of records in a relation to the specified value n.

Example:

Limit

```
grunt> h = limit g 2;
```

```
grunt> dump h;
```

```
(1,2,3)
```

```
(3,2,1)
```

RANK:

Syntax: ranked_data = RANK relation BY {field | expression} [DENSE] [ASC | DESC];

Description: Assigns a rank to each record based on the specified field or expression.

Category Diagnostic Operators on Data:

DESCRIBE:

Syntax: DESCRIBE relation;

Description: Provides the schema information of a relation.

Example:

```
Describe
grunt> describe k;
k: {a1: int,a2: int,int}
```

EXPLAIN:

Syntax: EXPLAIN relation;

Description: Displays the execution plan for a Pig Latin script.

Example:

```
Dump
grunt> dump k;
(1,2,2)
(3,2,0)
(4,2,0)
(4,3,2)
(4,3,3)
(8,3,3)
Explain
grunt> explain k;
#-----
# New Logical Plan:
#-----
k: (Name: LOStore Schema: a1#1470:int,a2#1471:int,#1474:int)
|
|--k: (Name: LOForEach Schema: a1#1470:int,a2#1471:int,#1474:int)
#-----
# Physical Plan:
#-----
k: Store(fakefile:org.apache.pig.builtin.PigStorage) - scope-814
|
|--k: New For Each(false,false,false)[bag] - scope-813
#-----
# Map Reduce Plan
#-----
MapReduce node scope-817
Map Plan
Local Rearrange[tuple]{tuple}(true) - scope-819
```

ILLUSTRATE:

Syntax: ILLUSTRATE relation;

Description: Shows a sample execution of the logical plan, using a generated subset of the input.

Example:

Illustrate

grunt> illustrate k;

| a | a1:int | a2:int | a3:int |

| | 1 | 2 | 3 |

| b | a1:int | a2:int | a3:int |

| | 1 | 2 | 3 |

| c | a1:int | a2:int | a3:int |

| | 1 | 2 | 3 |

| | 1 | 2 | 3 |

| j | a1:int | a2:int | a3:int |

| | 1 | 2 | 3 |

| k | a1:int | a2:int | :int |

| | 1 | 2 | 2 |

Category Macro and UDF statements:

DEFINE:

Example:

```
grunt> DEFINE myfilter(relvar,colvar) returns x{ $x = filter $relvar by $colvar<=2; };
```

```
grunt> dump c;
```

```
(1,2,3)
```

```
(4,2,1)
```

```
(8,3,4)
```

```
(4,3,3)
```

```
(1,2,3)
```

```
(4,2,1)
```

```
(4,3,4)
```

```
(3,2,1)
```

```
grunt> z = myfilter(c,$1);
```

```
grunt> dump z;
```

```
(1,2,3)
```

```
(4,2,1)
```

```
(1,2,3)
```

```
(4,2,1)
```

```
(3,2,1)
```

IMPORT:

Example:

```
[cloudera@quickstart ~]$ hdfs dfs -ls /
```

```
drwxr-xr-x - cloudera supergroup 0 2023-04-05 09:34 /demo
```

```
[cloudera@quickstart ~]$ hdfs dfs -put data /demo
```

```
[cloudera@quickstart ~]$ hdfs dfs -put data1 /demo
```

```
[cloudera@quickstart ~]$ hdfs dfs -ls /demo
```

```
-rw-r--r-- 1 cloudera supergroup 24 2023-04-09 06:42 /demo/data
```

```
-rw-r--r-- 1 cloudera supergroup 24 2023-04-09 06:42 /demo/data1
```

Create a new file with mymacro.macro with following content

```
=====
DEFINE myfilter(relvar,colvar) returns x{ $x = filter $relvar by $colvar<=2; };
```

Create a file my.pig with following content

```
=====
```

```
IMPORT '/home/cloudera/mymacro.macro';
```

```
a = load '/demo/data' using PigStorage(',') as (a1:int,a2:int,a3:int);
```

```
b = load '/demo/data1' using PigStorage(',') as (a1:int,a2:int,a3:int);
```

```
c = union a,b;  
data1 = myfilter(c,a1);  
dump data1;
```

REGISTER:

Syntax: REGISTER 'path/to/jar'; DEFINE function_name function_class;

Description: Registers and defines a custom loader or function.

Example:

```
REGISTER  
grunt> REGISTER /home/cloudera/workspace/pigudf.jar;  
grunt> DEFINE myfun TestUpper();  
grunt> a = load '/home/cloudera/demo' using PigStorage(',') as  
(id:int,name:chararray,age:int,city:chararray,sal:int);  
grunt> emp_Upper = foreach a generate myfun(name);  
grunt> dump emp_Upper;
```

Introduction to Hive and HiveQL

Introduction to Hive:

Apache Hive is an open-source data warehousing and SQL-like query language (HiveQL) tool built on top of the Hadoop Distributed File System (HDFS). It was developed by the Facebook and contributed to Apache Software Foundation. Hive provides a high-level interface for managing and querying large datasets stored in Hadoop clusters. The primary goal of Hive is to make it easier for users with SQL skills to process and analyze big data stored in Hadoop without the need to write complex MapReduce programs.

Use Cases of Apache Hive:

- **Data Warehousing:** Storing and querying large volumes of structured and semi-structured data in a data warehousing environment.
- **Data Analysis:** Performing data analysis and business intelligence tasks on big data sets using SQL-like queries.
- **ETL (Extract, Transform, Load):** Transforming and loading data from various sources into Hadoop for further analysis.
- **Log Processing:** Analyzing log files generated by applications or web servers stored in Hadoop.
- **Ad Hoc Querying:** Allowing users to run ad hoc queries on large datasets without the need for extensive programming.

Installing Hive:

- 1) Download a release, and unpack the tarball in a suitable place on your workstation:

```
% tar xzf apache-hive-x.y.z-bin.tar.gz
```

- 2) It's handy to put Hive on your path to make it easy to launch:

```
% export HIVE_HOME=~/.sw/apache-hive-x.y.z-bin
```

```
% export PATH=$PATH:$HIVE_HOME/bin
```

- 3) Now type hive to launch the Hive shell:

```
% hive
```

```
hive>
```

In Apache Hive, there are two modes for executing Hive Query Language (HiveQL) queries:

1) Interactive Mode

2) Batch Mode

Interactive Mode:

Interactive mode is also known as the Hive shell or CLI (Command Line Interface).

The Hive Shell:

- The shell is the primary way that we will interact with Hive, by issuing commands in HiveQL.
- HiveQL is Hive's query language, a dialect of SQL.
- In this mode, users can interactively type HiveQL queries and get immediate results.
- The Hive shell provides a command-line interface for users to submit queries and commands directly to Hive.
- Like SQL, HiveQL is generally case insensitive (except for string comparisons), so show tables; works equally well here.
- The Tab key will autocomplete Hive keywords and functions.

To start the Hive shell, you typically use the following command in the terminal:

\$ hive

Once in the Hive shell, you can enter HiveQL queries and receive results interactively. This mode is suitable for exploring data, running ad-hoc queries, and performing interactive data analysis.

Example:

-- Interactive HiveQL query

hive> SHOW DATABASES;

OK

Time taken: 0.473 seconds

hive> SHOW TABLES;

OK

Time taken: 0.473 seconds

hive> SELECT * FROM my_table LIMIT 10;

Batch Mode:

- Batch mode, also known as script mode or non-interactive mode, involves executing HiveQL queries from a script file.
- Users can write a series of HiveQL queries in a script file and then submit the entire script to Hive for execution.
- The -f option runs the commands in the specified file, which is script.q in this example:
- To run Hive in batch mode, you use the following command in the terminal:

```
% hive -f script.q
```

- The script file (script.q) contains one or more HiveQL queries, and the queries are executed sequentially.

Example script file (example_script.q):

```
-- Batch mode HiveQL queries  
CREATE TABLE IF NOT EXISTS my_output_table AS  
SELECT column1, column2  
FROM my_input_table  
WHERE column3 > 100;  
ANALYZE TABLE my_output_table COMPUTE STATISTICS;
```

Running the script:

```
hive -f example_script.q
```

These two modes cater to different use cases. Interactive mode is suitable for exploratory data analysis and quick queries, while batch mode is more appropriate for automated processes, scheduled jobs, and scenarios where multiple queries need to be executed in sequence.

Configuring Hive:

- Hive is configured using an XML configuration file called hive-site.xml and is located in Hive's conf directory.
- This file is where you can set properties that you want to set every time you run Hive.
- The same directory contains hive-default.xml, which documents the properties that Hive exposes and their default values.

- You can override the configuration directory that Hive looks for in hive-site.xml by passing the --config option to the hive command:

% hive --config /Users/tom/dev/hive-conf

Edit hive-site.xml:

```
<!-- hive-site.xml -->
<configuration>
  <!-- Hive Execution Engine -->
  <property>
    <name>hive.execution.engine</name>
    <value>tez</value>
    <description>Execution engine: mr, tez</description>
  </property>
  <!-- Hive Warehouse Directory -->
  <property>
    <name>hive.metastore.warehouse.dir</name>
    <value>/user/hive/warehouse</value>
    <description>Location of Hive warehouse directory</description>
  </property>
  <!-- Hive Metastore URI -->
  <property>
    <name>javax.jdo.option.ConnectionURL</name>
    <value>jdbc:derby::;databaseName=metastore_db;create=true</value>
    <description>JDBC connect string for a JDBC metastore</description>
  </property>
</configuration>
```

Common Configuration Properties:

- **Hive Warehouse Directory (hive.metastore.warehouse.dir):** Defines the default location where Hive stores table data. Make sure the specified directory exists and is writable by the Hive user.

- **Hive Metastore URI (javax.jdo.option.ConnectionURL):** Specifies the URI for the Hive Metastore database. The default is often set to Derby, but for production, consider using a more robust database like MySQL or PostgreSQL.

Hive Execution Engine (hive.execution.engine):

- Specifies the execution engine for Hive queries.
- Hive was originally written to use MapReduce as its execution engine, and that is still the default.
- It is now also possible to run Hive using Apache Tez as its execution engine, and work is underway to support Spark.
- Both Tez and Spark are general directed acyclic graph (DAG) engines that offer more flexibility and higher performance than MapReduce.
- For example, unlike MapReduce, where intermediate job output is materialized to HDFS, Tez and Spark can avoid replication overhead by writing the intermediate output to local disk, or even store it in memory.
- The execution engine is controlled by the `hive.execution.engine` property, which defaults to `mr` (for MapReduce).

Hive Services / Hive Architecture:

Hive CLI (Command Line Interface):

- The Hive CLI is an interactive command-line interface that allows users to interactively submit HiveQL queries.
- It provides a simple way for users to explore and analyze data using SQL-like queries directly from the command line.

hiveserver2:

- Runs Hive as a server exposing a Thrift service, enabling access from a range of clients written in different languages.
- HiveServer 2 improves on the original Hive-Server by supporting authentication and multiuser concurrency.
- Applications using the Thrift, JDBC, and ODBC connectors need to run a Hive server to communicate with Hive.

- Set the `hive.server2.thrift.port` configuration property to specify the port the server will listen on (defaults to 10000).

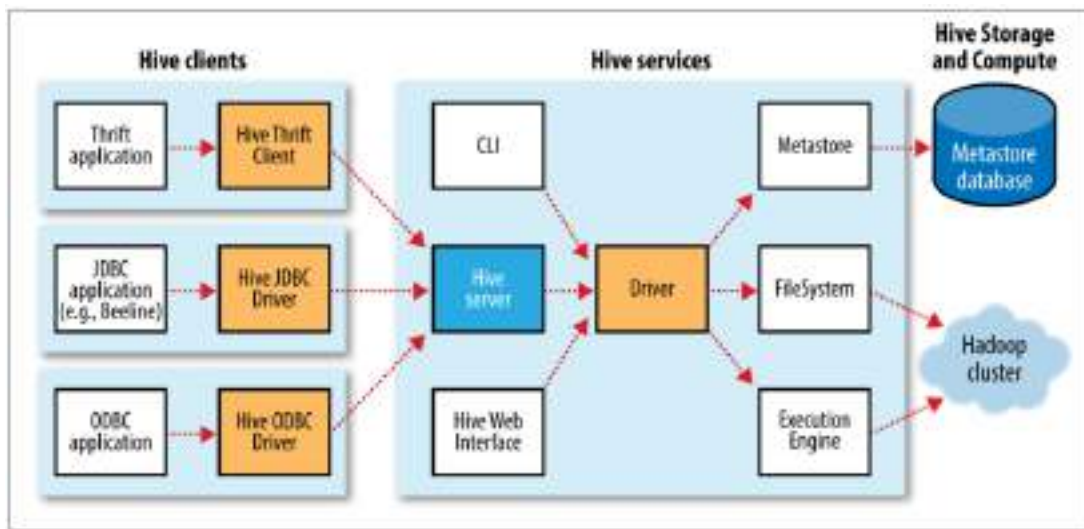


Figure 17-1. Hive architecture

Beeline:

A command-line interface to Hive that works in embedded mode (like the regular CLI), or by connecting to a HiveServer 2 process using JDBC.

Hwi:

The Hive Web Interface. A simple web interface that can be used as an alternative to the CLI without having to install any client software.

Jar:

The Hive equivalent of `hadoop jar`, a convenient way to run Java applications that includes both Hadoop and Hive classes on the classpath.

Hive Execution Engine:

Hive supports multiple execution engines for processing queries. The two main execution engines are:

- **MapReduce:** The traditional execution engine used in the earlier versions of Hive.
- **Apache Tez:** A more modern and efficient execution engine designed for faster query processing. Tez allows for better performance and optimization of query plans.

Metastore:

By default, the metastore is run in the same process as the Hive service. Using this service, it is possible to run the metastore as a standalone (remote) process.

The Meta Store:

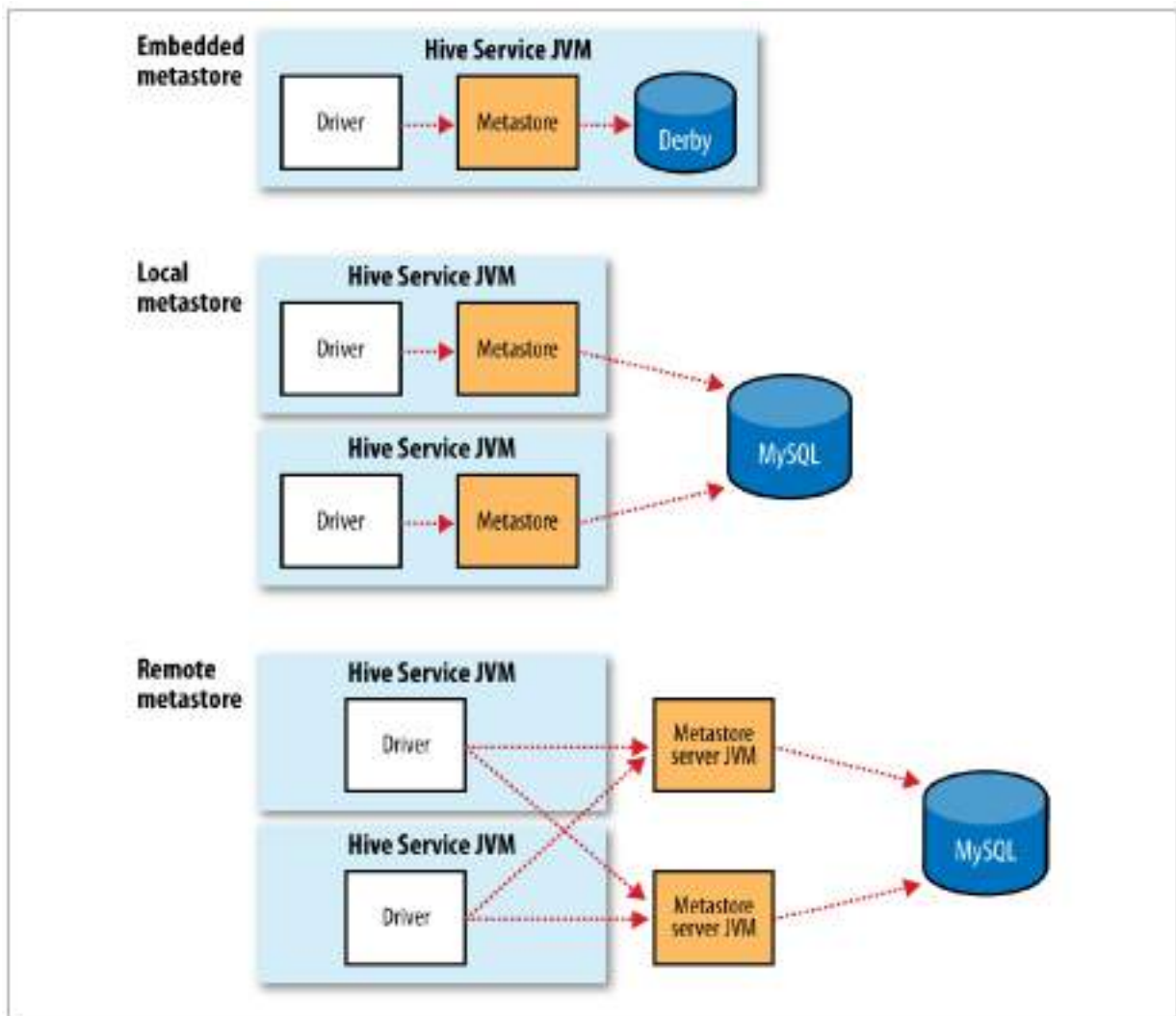


Figure 17-2. Metastore configurations

- The metastore is the central repository of Hive metadata.
- The metastore is divided into two pieces:
 - A Hive services
 - The backing store for the data.

Embedded Metastore:

- By default, the metastore service runs in the same JVM as the Hive service and contains an embedded Derby database instance backed by the local disk.
- This is called the embedded metastore configuration (see Figure 17-2).
- Using an embedded metastore is a simple way to get started with Hive.
- However, only one embedded Derby database can access the database files on disk at any one time, which means you can have only one Hive session open at a time that accesses the same metastore.
- Trying to start a second session produces an error when it attempts to open a connection to the metastore.

Local Metastore:

- The solution to supporting multiple sessions (and therefore multiple users) is to use a standalone database.
- This configuration is referred to as a local metastore, since the metastore service still runs in the same process as the Hive service but connects to a database running in a separate process, either on the same machine or on a remote machine.
- Any JDBC-compliant database may be used by setting the `javax.jdo.option.*` configuration properties.
- MySQL is a popular choice for the standalone metastore.
- In this case, two properties need to be set

```
<property> <!--Property 1>
    <name>javax.jdo.option.ConnectionURL</name>
    <value>jdbc:mysql://host/dbname?createDatabaseIfNotExist=true</value>
</property>
```

```
<property> <!--Property 2>
    <name> javax.jdo.option.ConnectionDriverName </name>
    <value> com.mysql.jdbc.Driver </value>
</property>
```

The JDBC driver JAR file for MySQL (Connector/J) must be on Hive's classpath, which is simply achieved by placing it in Hive's lib directory.

Remote Metastore:

Going a step further, there's another metastore configuration called a remote metastore, where one or more metastore servers run in separate processes to the Hive service. This brings better manageability and security because the database tier can be completely firewalled off, and the clients no longer need the database credentials.

A Hive service is configured to use a remote metastore by setting `hive.metastore.uris` to the metastore server URI(s), separated by commas if there is more than one. Metastore server URIs are of the form `thrift://host:port`, where the port corresponds to the one set by `METASTORE_PORT` when starting the metastore server.

Comparison with Traditional Databases:

1. Data Model:

Hive (Hadoop Ecosystem):

- Hive follows a schema-on-read approach. Data is stored in Hadoop Distributed File System (HDFS), and the schema is applied at the time of reading the data.
- Supports structured, semi-structured, and unstructured data.

Traditional Databases:

- Follow a schema-on-write approach. Data must conform to a predefined schema before being inserted into the database.
- Primarily designed for structured data with well-defined schemas.

2. Query Language:

Hive:

- Uses Hive Query Language (HiveQL), which is SQL-like but adapted for big data processing.
- Optimized for batch processing and large-scale data analysis.

Traditional Databases:

- Use standard SQL for querying and managing data.
- Optimized for transactional processing and real-time data access.

3. Processing Paradigm:

Hive:

- Leverages MapReduce, Apache Tez, or other execution engines for distributed data processing.
- Well-suited for batch processing and long-running analytical queries.

Traditional Databases:

- Typically use a row-oriented storage and processing model.
- Designed for transactional processing with support for ACID properties.

4. Performance:**Hive:**

- Designed for scalability and parallel processing of large datasets.
- Performance may be slower than traditional databases for OLAP queries.

Traditional Databases:

- Optimized for low-latency transactional processing and OLAP queries on smaller datasets.
- May struggle with the scale of big data analytics.

5. Storage:**Hive:**

- Stores data in distributed file systems like HDFS.
- Supports various file formats, including ORC, Parquet, and Avro.

Traditional Databases:

- Data is stored in relational database management systems (RDBMS) like MySQL, PostgreSQL, or Oracle.
- Typically uses row-based storage.

6. Use Cases:**Hive:**

- Well-suited for analytical processing, data warehousing, and batch-oriented ETL (Extract, Transform, Load) jobs.
- Ideal for processing large volumes of data where low-latency is not a primary concern.

Traditional Databases:

- Suitable for transactional processing, real-time data access, and OLAP queries on smaller datasets.

- Commonly used in online transaction processing (OLTP) systems.

7. Scale:

Hive:

- Scales horizontally by adding more nodes to the Hadoop cluster.
- Well-suited for handling petabytes of data.

Traditional Databases:

- Scales vertically by adding more resources to a single server.
- May have limitations in handling extremely large datasets.

Schema on Read Versus Schema on Write

In a traditional database, a table's schema is enforced at data load time. If the data being loaded doesn't conform to the schema, then it is rejected. This design is sometimes called schema on write because the data is checked against the schema when it is written into the database.

Hive, on the other hand, doesn't verify the data when it is loaded, but rather when a query is issued. This is called schema on read.

There are trade-offs between the two approaches.

- **Schema on read makes** for a very fast initial load, since the data does not have to be read, parsed, and serialized to disk in the database's internal format.
- The load operation is just a file copy or move.
- It is more flexible, too: consider having two schemas for the same underlying data, depending on the analysis being performed.
- **Schema on write makes** query time performance faster because the database can index columns and perform compression on the data.
- The trade-off, however, is that it takes longer to load data into the database.

Updates, Transactions, and Indexes

- Updates, transactions, and indexes are mainstays of traditional databases.
- Yet, until recently, these features have not been considered a part of Hive's feature set.

- This is because Hive was built to operate over HDFS data using MapReduce, where full-table scans are the norm and a table update is achieved by transforming the data into a new table.
- For a data warehousing application that runs over large portions of the dataset, this works well.
- Hive has long supported adding new rows in bulk to an existing table by using INSERT INTO to add new data files to a table.
- In addition, it is possible to UPDATE and DELETE rows in a table.
- HDFS does not provide in-place file updates, so changes resulting from inserts, updates, and deletes are stored in small delta files.
- Delta files are periodically merged into the base table files by MapReduce jobs that are run in the background by the metastore.
- These features only work in the context of transactions (introduced in Hive 0.13.0), so the table they are being used on needs to have transactions enabled on it. Queries reading the table are guaranteed to see a consistent snapshot of the table.
- Hive also has support for table- and partition-level locking.
- Locks prevent, for example, one process from dropping a table while another is reading from it.
- Locks are managed transparently using ZooKeeper, so the user doesn't have to acquire or release them, although it is possible to get information about which locks are being held via the SHOW LOCKS statement.
- By default, locks are not enabled.
- Hive indexes can speed up queries in certain cases.
- A query such as SELECT * from t WHERE x = a, for example, can take advantage of an index on column x, since only a small portion of the table's files need to be scanned.
- There are currently two index types:
 - compact and
 - bitmap
- Compact indexes store the HDFS block numbers of each value, rather than each file offset, so they don't take up much disk space but are still effective for the case where values are clustered together in nearby rows.

- Bitmap indexes use compressed bitsets to efficiently store the rows that a particular value appears in, and they are usually appropriate for low-cardinality columns (such as gender or country).

SQL-on-Hadoop Alternatives

In the years since Hive was created, many other SQL-on-Hadoop engines have emerged to address some of Hive's limitations these are.

- Cloudera Impala
- Stinger
- Presto
- Apache Drill
- Spark SQL
- Apache Phoenix

Cloudera Impala:

- Cloudera Impala, an open-source interactive SQL engine, was one of the first, giving an order of magnitude performance boost compared to Hive running on MapReduce.
- Impala uses a dedicated daemon that runs on each datanode in the cluster.
- When a client runs a query, it contacts an arbitrary node running an Impala daemon, which acts as a coordinator node for the query.
- The coordinator sends work to other Impala daemons in the cluster and combines their results into the full result set for the query.
- Impala uses the Hive metastore and supports Hive formats and most HiveQL constructs (plus SQL-92), so in practice it is straightforward to migrate between the two systems, or to run both on the same cluster.
- Hive has not stood still, though, and since Impala was launched.

Stinger:

- The “Stinger” initiative by Hortonworks has improved the performance of Hive through support for Tez as an execution engine, and the addition of a vectorized query engine among other improvements.

Other prominent open-source Hive alternatives include Presto from Facebook, Apache Drill, and Spark SQL.

- Presto and Drill have similar architectures to Impala, although Drill targets SQL:2011 rather than HiveQL.
- Spark SQL uses Spark as its underlying engine, and lets you embed SQL queries in Spark programs.
- Apache Phoenix takes a different approach entirely: it provides SQL on HBase.
- SQL access is through a JDBC driver that turns queries into HBase scans and takes advantage of HBase coprocessors to perform server-side aggregation.
- Metadata is stored in HBase, too.

HiveQL

- Hive's SQL dialect, called HiveQL, is a mixture of SQL-92, MySQL, and Oracle's SQL dialect.
- The level of SQL-92 support has improved over time, and will likely continue to get better.
- HiveQL also provides features from later SQL standards, such as window functions from SQL:2003.
- Some of Hive's non-standard extensions to SQL were inspired by MapReduce, such as multi-table inserts and the TRANSFORM, MAP, and REDUCE clauses.

Data Types:

Hive supports both primitive and complex data types. Primitives include numeric, Boolean, string, and timestamp types. The complex data types include arrays, maps, and structs. Hive's data types are listed in Table 17-3.

Primitive types

- Hive's primitive types correspond roughly to Java's data types, although some names are influenced by MySQL's type names.
- There is a BOOLEAN type for storing true and false values.

- There are four signed integral types: TINYINT, SMALLINT, INT, and BIGINT, which are equivalent to Java's byte, short, int, and long primitive types, respectively (they are 1-byte, 2-byte, 4-byte, and 8-byte signed integers).
- Hive's floating-point types, FLOAT and DOUBLE, correspond to Java's float and double, which are 32-bit and 64-bit floating-point numbers.
- The DECIMAL data type is used to represent arbitrary-precision decimals, like Java's BigDecimal, and are commonly used for representing currency values. DECIMAL values are stored as unscaled integers. The precision is the number of digits in the unscaled value, and the scale is the number of digits to the right of the decimal point. So, for example, DECIMAL(5,2) stores numbers between -999.99 and 999.99. If the scale is omitted then it defaults to 0, so DECIMAL(5) stores numbers in the range -99,999 to 99,999 (i.e., integers). If the precision is omitted then it defaults to 10, so DECIMAL is equivalent to DECIMAL(10,0). The maximum allowed precision is 38, and the scale must be no larger than the precision.

Table 17-3. Hive data types

Category	Type	Description	Literal examples
Primitive	BOOLEAN	true/false value.	TRUE
	TINYINT	1-byte (8-bit) signed integer, from -128 to 127.	1Y
	SMALLINT	2-byte (16-bit) signed integer, from -32,768 to 32,767.	1S
	INT	4-byte (32-bit) signed integer, from -2,147,483,648 to 2,147,483,647.	1
	BIGINT	8-byte (64-bit) signed integer, from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.	1L
	FLOAT	4-byte (32-bit) single-precision floating-point number.	1.0
	DOUBLE	8-byte (64-bit) double-precision floating-point number.	1.0
	DECIMAL	Arbitrary-precision signed decimal number.	1.0
	STRING	Unbounded variable-length character string.	'a', "a"
	VARCHAR	Variable-length character string.	'a', "a"
	CHAR	Fixed-length character string.	'a', "a"
	BINARY	Byte array.	Not supported
	TIMESTAMP	Timestamp with nanosecond precision.	1325502245000, '2012-01-02 03:04:05.123456789'
	DATE	Date.	'2012-01-02'

Category	Type	Description	Literal examples
Complex	ARRAY	An ordered collection of fields. The fields must all be of the same type.	<code>array(1, 2)</code> ^a
	MAP	An unordered collection of key-value pairs. Keys must be primitives; values may be any type. For a particular map, the keys must be the same type, and the values must be the same type.	<code>map('a', 1, 'b', 2)</code>
	STRUCT	A collection of named fields. The fields may be of different types.	<code>struct('a', 1, 1.0)</code> ^b <code>named_struct('col1', 'a', 'col2', 1, 'col3', 1.0)</code>
	UNION	A value that may be one of a number of defined data types. The value is tagged with an integer (zero-indexed) representing its data type in the union.	<code>create_union(1, 'a', 63)</code>

- There are three Hive data types for storing text.
- STRING is a variable-length character string with no declared maximum length.
- VARCHAR types are similar except they are declared with a maximum length between 1 and 65535; for example, VARCHAR(100).
- CHAR types are fixed-length strings that are padded with trailing spaces if necessary; for example, CHAR(100). Trailing spaces are ignored for the purposes of string comparison of CHAR values.
- The BINARY data type is for storing variable-length binary data.
- The TIMESTAMP data type stores timestamps with nanosecond precision.
- The DATE data type stores a date with year, month, and day components.

Complex types

- Hive has four complex types: ARRAY, MAP, STRUCT, and UNION.
- ARRAY and MAP are like their namesakes in Java, whereas a STRUCT is a record type that encapsulates a set of named fields. A UNION specifies a choice of data types; values must match exactly one of these types.
- Complex types permit an arbitrary level of nesting. Complex type declarations must specify the type of the fields in the collection, using an angled bracket notation, as illustrated in this table definition with three columns (one for each complex type):

```
CREATE TABLE complex (
  c1 ARRAY<INT>,
  c2 MAP<STRING, INT>,
  c3 STRUCT<a:STRING, b:INT, c:DOUBLE>,
  c4 UNIONTYPE<STRING, INT>
```

);

- If we load the table with one row of data for ARRAY, MAP, STRUCT, and UNION, as shown in the “Literal examples” column in Table 17-3. The following query demonstrates the field accessor operators for each type:

```
hive> SELECT c1[0], c2['b'], c3.c, c4 FROM complex;
```

```
1      2      1.0 {1:63}
```

Operators and Functions

The usual set of SQL operators is provided by Hive:

- **relational operators** (such as `x = 'a'` for testing equality, `x IS NULL` for testing nullity, and `x LIKE 'a%'` for pattern matching),
- **arithmetic operators** (such as `x + 1` for addition), and
- **logical operators** (such as `x OR y` for logical OR).
- The operators match those in MySQL, which deviates from SQL-92 because `||` is logical OR, not string concatenation.
- Use the `concat` function for the latter in both MySQL and Hive.

Hive comes with a large number of built-in functions - divided into categories that include:

- Mathematical and statistical functions
- String functions
- Date functions
- Conditional functions
- Aggregate functions and
- Functions for working with XML (using the `xpath` function) and JSON.

You can retrieve a list of functions from the Hive shell by typing **SHOW FUNCTIONS**.

To get brief usage instructions for a particular function, use the `DESCRIBE` command:

```
hive> DESCRIBE FUNCTION length;
```

`length(str | binary)` - Returns the length of `str` or number of bytes in binary data.

Tables in HiveQL With DML and DDL Operators

Tables

A Hive table is logically made up of the data being stored and the associated metadata describing the layout of the data in the table. The data typically resides in HDFS, although it may reside in any Hadoop filesystem, including the local filesystem or S3.

Multiple Database/Schema Support

Many relational databases have a facility for multiple namespaces, which allows users and applications to be segregated into different databases or schemas. Hive supports the same facility and provides commands such as **CREATE DATABASE dbname**, **USE dbname**, and **DROP DATABASE dbname**. You can fully qualify a table by writing **dbname.table_name**. If no database is specified, tables belong to the default database.

Managed Tables and External Tables:

When you create a table in Hive, by default Hive will manage the data, which means that Hive moves the data into its warehouse directory. Alternatively, you may create an external table, which tells Hive to refer to the data that is at an existing location outside the warehouse directory.

The difference between the two table types is seen in the LOAD and DROP semantics.

Managed Tables:

1) Create a Managed Table:

```
CREATE TABLE managed_table ( column1 INT, column2 STRING, ... );
```

2) Load Data into the Managed Table:

```
LOAD DATA INPATH '/demo/data.txt' INTO TABLE managed_table;
```

When you load data into a managed table, the file is moved into Hive's warehouse directory. For example, load operation will move the file `hdfs://demo/data.txt` into Hive's warehouse directory for the `managed_table` table, which is `hdfs://user/hive/warehouse/managed_table`.

3) If the table is later dropped, using:

```
DROP TABLE managed_table;
```

The table, including its metadata and its data, is deleted. It bears repeating that since the initial LOAD performed a move operation, and the DROP performed a delete operation, the data no longer exists anywhere. This is what it means for Hive to manage the data.

External Table:

An external table behaves differently. You control the creation and deletion of the data. The location of the external data is specified at table creation time.

External tables point to data stored outside the Hive warehouse, and Hive does not manage the lifecycle of the data. External tables provide a way to access data without moving or copying it into the Hive warehouse.

```
CREATE EXTERNAL TABLE my_external_table( column1 INT, column2 STRING, ...)  
LOCATION '/path/to/external/data';
```

With the EXTERNAL keyword, Hive knows that it is not managing the data, so it doesn't move it to its warehouse directory. Indeed, it doesn't even check whether the external location exists at the time it is defined. This is a useful feature because it means you can create the data lazily after creating the table.

When you drop an external table, Hive will leave the data untouched and only delete the metadata.

Partitions and Buckets

Partitioning involves dividing a table into smaller, more manageable sub-tables based on one or more columns. Each partition contains rows that share the same values in the partitioning columns. Hive organizes the data physically on disk by partition, making it easier to prune unnecessary data when querying.

How to Create a Partitioned Table:

```
CREATE TABLE partitioned_table ( column1 INT, column2 STRING, ...)  
PARTITIONED BY (partition_column STRING);
```


Bucketing, also known as clustering, involves dividing data into a fixed number of buckets based on a hash function applied to one or more columns. This helps in achieving more even distribution of data, which can improve query performance by reducing the amount of data that needs to be processed during a query.

How to Create a Bucketed Table:

```
CREATE TABLE bucketed_table ( column1 INT, column2 STRING, ... )  
CLUSTERED BY (column1) INTO num_buckets BUCKETS;
```

Partitioned Table:

To take an example where partitions are commonly used, **imagine logfiles** where each record includes a **timestamp**. If we partition by date, then records for the same date will be stored in the same partition.

The advantage to this scheme is that queries that are restricted to a particular date or set of dates can run much more efficiently, because they only need to scan the files in the partitions that the query pertains to.

A table may be partitioned in multiple dimensions. For example, in addition to partitioning logs by date, we might also sub-partition each date partition by country to permit efficient queries by location.

Partitions are defined at table creation time using the **PARTITIONED BY** clause, which takes a list of column definitions. For the hypothetical logfiles example, we might define a table with records comprising a timestamp and the log line itself:

```
CREATE TABLE logs (ts BIGINT, line STRING)  
PARTITIONED BY (dt STRING, country STRING);
```

When we load data into a partitioned table, the partition values are specified explicitly:

```
LOAD DATA INPATH '/input/file1'  
INTO TABLE logs  
PARTITION (dt='2001-01-01', country='GB');
```

At the filesystem level, partitions are simply nested subdirectories of the table directory. After loading a few more files into the logs table, the directory structure might look like this:

```

/user/hive/warehouse/logs
├── dt=2001-01-01/
│   ├── country=GB/
│   │   ├── file1
│   │   └── file2
│   └── country=US/
│       └── file3
└── dt=2001-01-02/
    ├── country=GB/
    │   └── file4
    └── country=US/
        ├── file5
        └── file6

```

The logs table has two date partitions (2001-01-01 and 2001-01-02, corresponding to subdirectories called dt=2001-01-01 and dt=2001-01-02); and two country subpartitions (GB and US, corresponding to nested subdirectories called country=GB and country=US). The datafiles reside in the leaf directories.

You can use partition columns in SELECT statements in the usual way. Hive performs input pruning to scan only the relevant partitions. For example:

```

SELECT ts, dt, line

FROM logs

WHERE country='GB';

```

will only scan file1, file2, and file4. Notice, too, that the query returns the values of the dt partition column, which Hive reads from the directory names since they are not in the datafiles.

Bucketing Table:

There are two reasons, to organize tables (or partitions) into buckets.

- The first is to enable more efficient queries. Bucketing imposes extra structure on the table, which Hive can take advantage of when performing certain queries. In particular, a join of two tables that are bucketed on the same columns—which include the join columns—can be efficiently implemented as a map-side join.
- The second reason to bucket a table is to make sampling more efficient. When working with large datasets, it is very convenient to try out queries on a fraction of your dataset while you are in the process of developing or refining them.

First, for a table to be bucketed, We use the CLUSTERED BY clause to specify the columns to bucket on and the number of buckets:

```
CREATE TABLE bucketed_users (id INT, name STRING)
CLUSTERED BY (id) INTO 4 BUCKETS;
```

Here we are using the user ID to determine the bucket, so any particular bucket will effectively have a random set of users in it.

Take an unbucketed users table:

```
hive> SELECT * FROM users;

0 Nat
2 Joe
3 Kay
4 Ann
```

To populate the bucketed table, we need to set the **hive.enforce.bucketing** property to **true** so that Hive knows to create the number of buckets declared in the table definition.

Then it is just a matter of using the INSERT command:

```
INSERT OVERWRITE TABLE bucketed_users
SELECT * FROM users;
```

Physically, each bucket is just a file in the table (or partition) directory. The filename is not important, but bucket n is the nth file when arranged in lexicographic order. In fact, buckets correspond to MapReduce output file partitions: a job will produce as many buckets (output files) as reduce tasks. Running this command:

```
hive> dfs -ls /user/hive/warehouse/bucketed_users;
```

shows that four files were created, with the following names (the names are generated by Hive):

```
000000_0
000001_0
000002_0
000003_0
```

The first bucket contains the users with IDs 0 and 4, since for an INT the hash is the integer itself, and the value is reduced modulo the number of buckets—four, in this case:

```
hive> dfs -cat /user/hive/warehouse/bucketed_users/000000_0;
```

```
0      Nat
4      Ann
```

We can see the same thing by sampling the table using the TABLESAMPLE clause, which restricts the query to a fraction of the buckets in the table rather than the whole table:

```
hive> SELECT * FROM bucketed_users
      > TABLESAMPLE(BUCKET 1 OUT OF 4 ON id);
      4 Ann
      0 Nat
```

Bucket numbering is 1-based, so this query retrieves all the users from the first of four buckets. For a large, evenly distributed dataset, approximately one-quarter of the table's rows would be returned. It's possible to sample a number of buckets by specifying a different proportion (which need not be an exact multiple of the number of buckets, as sampling is not intended to be a precise operation). For example, this query returns half of the buckets:

```
hive> SELECT * FROM bucketed_users
      > TABLESAMPLE(BUCKET 1 OUT OF 2 ON id);
      4 Ann
      0 Nat
```

Storage Formats:

Importing Data - Altering Tables, Dropping Tables, Querying Data, Sorting and Aggregating, Map Reduce Scripts, Joins, Sub queries, Views

User-Defined Functions

Sometimes the query you want to write can't be expressed easily (or at all) using the built-in functions that Hive provides. By allowing you to write a user-defined function (UDF), Hive makes it easy to plug in your own processing code and invoke it from a Hive query.

UDFs have to be written in Java, the language that Hive itself is written in. For other languages, consider using a `SELECT TRANSFORM` query, which allows you to stream data through a user-defined script.

There are three types of UDF in Hive:

- (regular) UDFs
- User-defined aggregate functions (UDAFs), and
- User-defined table-generating functions (UDTFs).

They differ in the number of rows that they accept as input and produce as output:

- A UDF operates on a single row and produces a single row as its output. Most functions, such as mathematical functions and string functions, are of this type.
- A UDAF works on multiple input rows and creates a single output row. Aggregate functions include such functions as `COUNT` and `MAX`.
- A UDTF operates on a single row and produces multiple rows—a table—as output.

Writing a UDF

To illustrate the process of writing and using a UDF, we'll write a simple UDF to square the numbers.

```
import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.IntWritable;
public class Square extends UDF {
    public IntWritable evaluate(IntWritable input) {
        if (input == null)
            return null;
        return new IntWritable(input.get() * input.get());
    }
}
```

A UDF must satisfy the following two properties:

- A UDF must be a subclass of `org.apache.hadoop.hive.ql.exec.UDF`.
- A UDF must implement at least one `evaluate()` method.

The `evaluate()` method is not defined by an interface, since it may take an arbitrary number of arguments, of arbitrary types, and it may return a value of arbitrary type.

Hive introspects the UDF to find the `evaluate()` method that matches the Hive function that was invoked.

The `Square` class has one `evaluate()` method. `IntWritable evaluate(IntWritable input)`.

To use the UDF in Hive, we first need to package the compiled Java class in a JAR file.

Next, we register the function in the metastore and give it a name using the `CREATE FUNCTION` statement:

```
hive> ADD JAR /home/cloudera/workspace/SquareUDF.jar;
```

```
hive> CREATE TEMPORARY FUNCTION squ AS 'Square';
```

Writing a UDAF

An aggregate function is more difficult to write than a regular UDF. Values are aggregated in chunks (potentially across many tasks), so the implementation has to be capable of combining partial aggregations into a final result. The code to achieve this is best explained by example, so let's look at the implementation of a simple UDAF for calculating the maximum of a collection of integers.

A UDAF for calculating the maximum of a collection of integers

```
import org.apache.hadoop.hive.ql.exec.UDAF;
import org.apache.hadoop.hive.ql.exec.UDAFEvaluator;
import org.apache.hadoop.io.IntWritable;
public class Maximum extends UDAF {
    public static class MaximumIntUDAFEvaluator implements UDAFEvaluator {
        private IntWritable result;

        public void init() {
            result = null;
        }
    }
}
```

```

    public boolean iterate(IntWritable value) {
        if (value == null) {
            return true;
        }
        if (result == null) {
            result = new IntWritable(value.get());
        } else {
            result.set(Math.max(result.get(), value.get()));
        }
        return true;
    }

    public IntWritable terminatePartial() {
        return result;
    }

    public boolean merge(IntWritable other) {
        return iterate(other);
    }

    public IntWritable terminate() {
        return result;
    }
}

```

The class structure is slightly different from the one for UDFs.

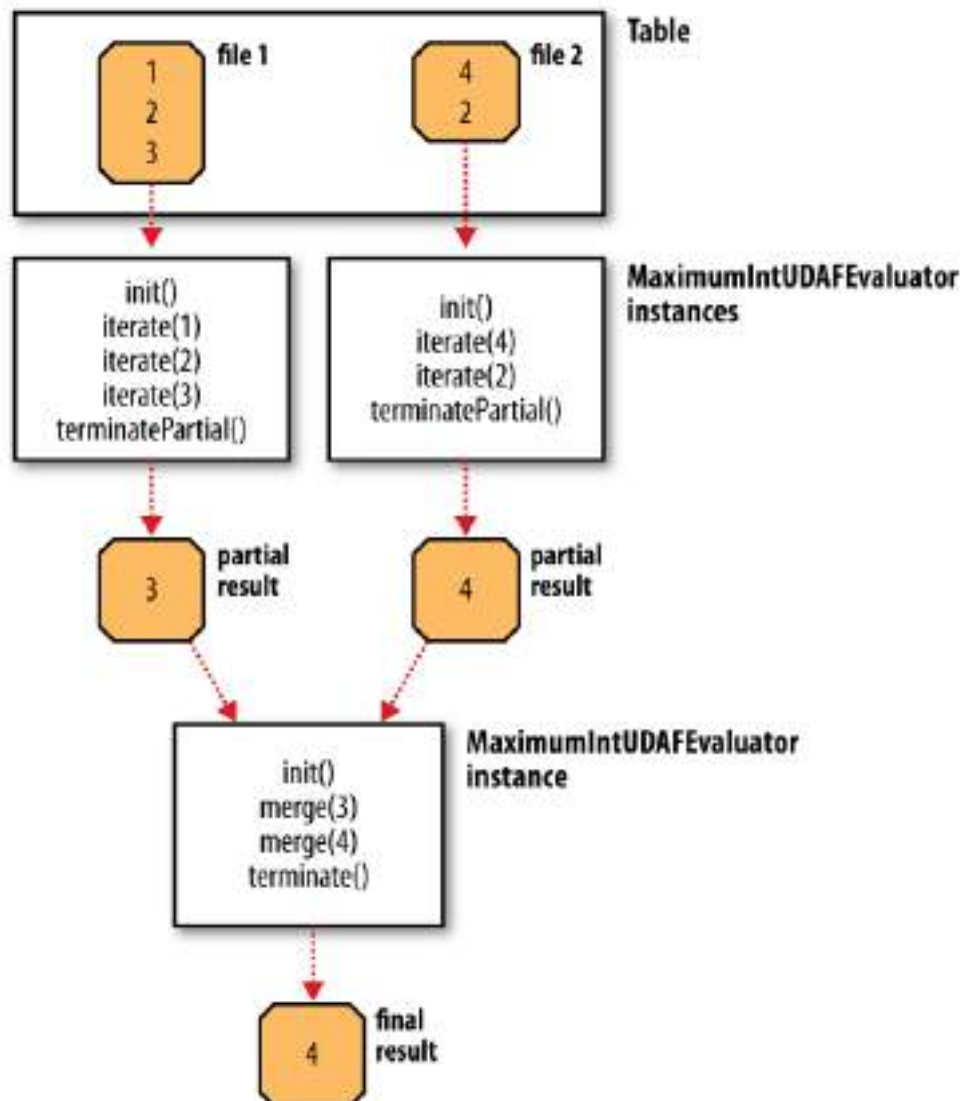
A UDAF must be a subclass of `org.apache.hadoop.hive.ql.exec.UDAF` and contain one or more nested static classes implementing `org.apache.hadoop.hive.ql.exec.UDAFEvaluator`. In this example, there is a single nested class, `MaximumIntUDAFEvaluator`, but we could add more evaluators, such as `MaximumLongUDAFEvaluator`, `MaximumFloatUDAFEvaluator`, and so on, to provide overloaded forms of the UDAF for finding the maximum of a collection of longs, floats, and so on.

An evaluator must implement five methods, described in turn here (the flow is illustrated):

init()

- The `init()` method initializes the evaluator and resets its internal state.
- In `MaximumIntUDAFEvaluator`, we set the `IntWritable` object holding the final result to null.
- We use null to indicate that no values have been aggregated yet, which has the desirable effect of making the maximum value of an empty set NULL.

Figure. Data flow with partial results for a UDAF



iterate()

- The `iterate()` method is invoked every time there is a new value to be aggregated.
- The evaluator should update its internal state with the result of performing the aggregation.
- The arguments that `iterate()` takes correspond to those in the Hive function from which it was invoked.
- In this example, there is only one argument.
- The value is first checked to see whether it is null, and if it is, it is ignored.

- Otherwise, the result instance variable is set either to value's integer value or to the larger of the current result and value.
- We return true to indicate that the input value was valid.

terminatePartial()

- The terminatePartial() method is called when Hive wants a result for the partial aggregation.
- The method must return an object that encapsulates the state of the aggregation.
- In this case, an IntWritable suffices because it encapsulates either the maximum value seen or null if no values have been processed.

merge()

- The merge() method is called when Hive decides to combine one partial aggregation with another.
- The method takes a single object, whose type must correspond to the return type of the terminatePartial() method.
- In this example, the merge() method can simply delegate to the iterate() method because the partial aggregation is represented in the same way as a value being aggregated.
- This is not generally the case, and the method should implement the logic to combine the evaluator's state with the state of the partial aggregation.

terminate()

- The terminate() method is called when the final result of the aggregation is needed.
- The evaluator should return its state as a value. In this case, we return the result instance variable.

Let's exercise our new function:

```
hive> CREATE TEMPORARY FUNCTION maximum AS 'hive.Maximum';
```

```
hive> SELECT maximum(temperature) FROM records;
```