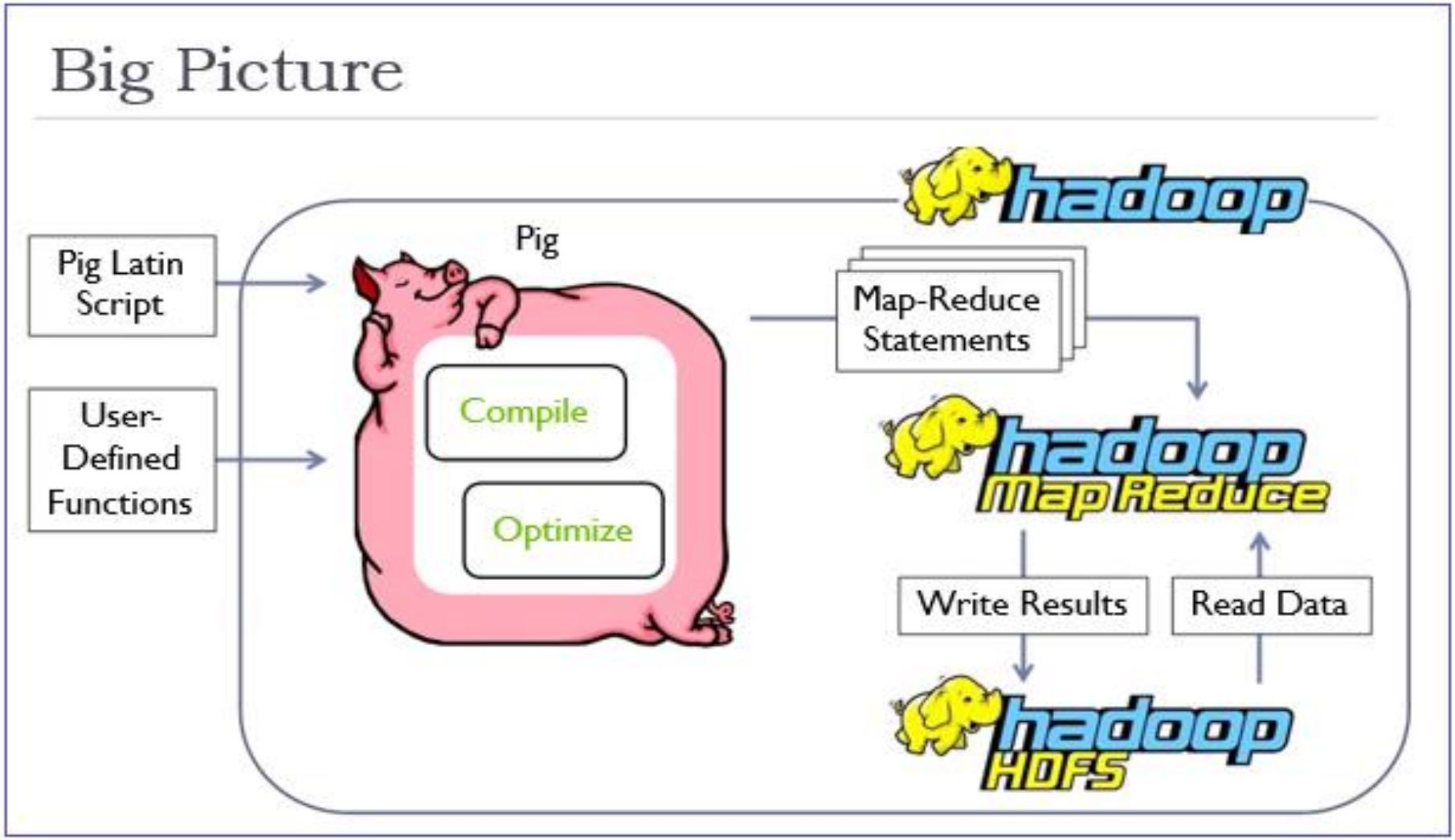
Unit-V
Analytics Framework

Applications on Big Data Using Pig

- Apache Pig is an **abstraction over MapReduce**.
- It is a tool/platform which is used to analyze larger sets of data representing them as data flows.
- Pig is generally used with **Hadoop**; we can perform all the data manipulation operations in Hadoop using Apache Pig.
- To write data analysis programs, Pig provides a high-level language known as **Pig Latin**.
- This language provides various operators using which programmers can develop their own functions for reading, writing, and processing data.
- To analyze data using **Apache Pig**, programmers need to write scripts using Pig Latin language.
- All these scripts are internally converted to Map and Reduce tasks.
- Apache Pig has a component known as **Pig Engine** that accepts the Pig Latin scripts as input and converts those scripts into MapReduce jobs.

Pig


Big Picture
Pig Latin Script
User-Defined Functions
Pig
Compile
Optimize
hadoop
Map-Reduce Statements
hadoop MapReduce
Write Results
Read Data
hadoop HDFS

Apache Pig – History

- In **2006**, Apache Pig was developed as a research project at Yahoo, especially to create and execute MapReduce jobs on every dataset.
- In **2007**, Apache Pig was open sourced via Apache incubator.
- In **2008**, the first release of Apache Pig came out.
- In **2010**, Apache Pig graduated as an Apache top-level project.

Why Do We Need Apache Pig?

- Programmers who are not so good at Java normally used to struggle working with Hadoop, especially while performing any MapReduce tasks.
- Apache Pig is a boon for all such programmers.
  - Using **Pig Latin**, programmers can perform MapReduce tasks easily without having to type complex codes in Java.
  - Apache Pig uses **multi-query approach**, thereby reducing the length of codes.
  - Pig Latin is **SQL-like language** and it is easy to learn Apache Pig when you are familiar with SQL.
  - Apache Pig provides many built-in operators to support data operations like joins, filters, ordering, etc.
  - It also provides nested data types like tuples, bags, and maps that are missing from MapReduce.

Features of Pig

Apache Pig comes with the following features −

- **Rich set of operators** − It provides many operators to perform operations like join, sort, filer, etc.
- **Ease of programming** − Pig Latin is similar to SQL and it is easy to write a Pig script if you are good at SQL.
- **Optimization opportunities** − The tasks in Apache Pig optimize their execution automatically, so the programmers need to focus only on semantics of the language.
- **Extensibility** − Using the existing operators, users can develop their own functions to read, process, and write data.
- **UDF's** − Pig provides the facility to create **User-defined Functions** in other programming languages such as Java and invoke or embed them in Pig Scripts.
- **Handles all kinds of data** − Apache Pig analyzes all kinds of data, both structured as well as unstructured. It stores the results in HDFS.

Pig – Sample Code

- Let's look at a simple example by writing the program to calculate the maximum recorded temperature by year for weather dataset in Pig Latin.
- The complete program is only a few lines long:

```
-- max_temp.pig: Finds the maximum temperature by year
records = LOAD 'input/ncdc/micro-tab/sample.txt'
AS (year:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999 AND
quality IN (0, 1, 4, 5, 9);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
MAX(filtered_records.temperature);
DUMP max_temp;
```

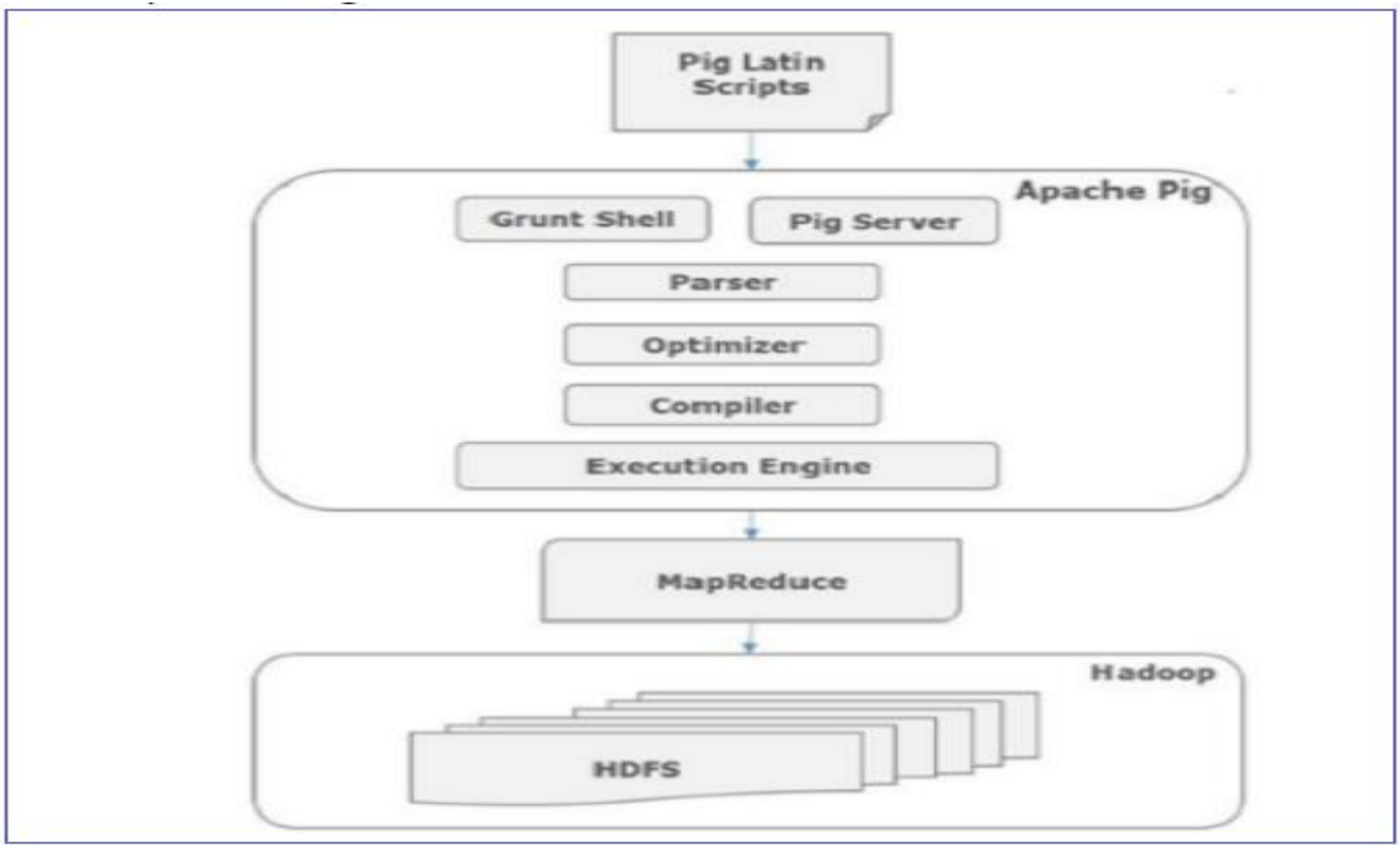
Apache Pig Vs MapReduce

Apache Pig	MapReduce
Apache Pig is a data flow language.	MapReduce is a data processing paradigm.
It is a high level language.	MapReduce is low level and rigid.
Performing a Join operation in Apache Pig is pretty simple.	It is quite difficult in MapReduce to perform a Join operation between datasets.
Any novice programmer with a basic knowledge of SQL can work conveniently with Apache Pig.	Exposure to Java is must to work with MapReduce.
Apache Pig uses multi-query approach, thereby reducing the length of the codes to a great extent.	MapReduce will require almost 20 times more the number of lines to perform the same task.
There is no need for compilation. On execution, every Apache Pig operator is converted internally into a MapReduce job.	MapReduce jobs have a long compilation process.

Apache Pig Vs SQL

Pig	SQL
Pig Latin is a **procedural** language.	SQL is a **declarative** language.
In Apache Pig, **schema** is optional. We can store data without designing a schema (values are stored as $01, $02 etc.)	Schema is mandatory in SQL.
The data model in Apache Pig is **nested relational**.	The data model used in SQL **is flat relational**.
Apache Pig provides limited opportunity for **Query optimization**.	There is more opportunity for query optimization in SQL.

Apache Pig - Architecture

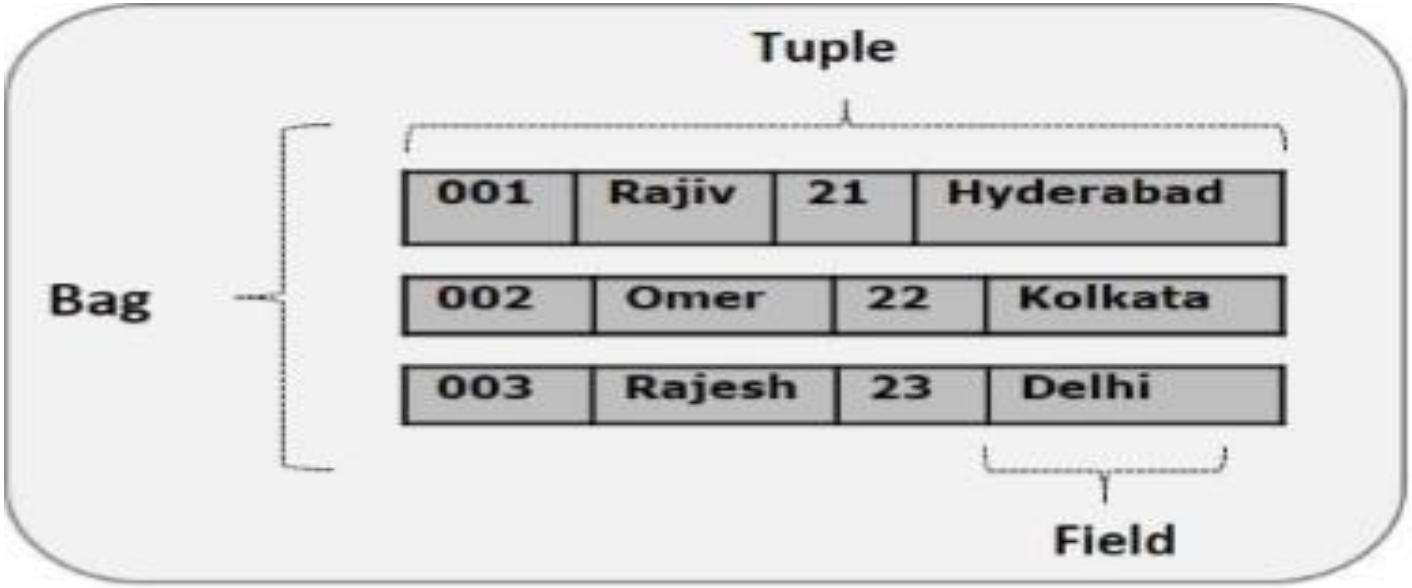


Apache Pig Components

- Parser
- Optimizer
- Compiler
- Execution engine

Pig Latin Data Model

- The data model of Pig Latin is fully nested and it allows complex non-atomic datatypes such as **map** and **tuple**.



- In its data model, Pig Latin has those four types
  1. **Atom**
  2. **Tuple**
  3. **Bag**
  4. **Map**

Pig Latin Data Model

Atom

- Any single value in Pig Latin, irrespective of their data, type is known as an **Atom**.
- It is stored as string and can be used as string and number.
- int, long, float, double, chararray, and bytearray are the atomic values of Pig.
- A piece of data or a simple atomic value is known as a **field**.
- **Example** − ‘raja’ or ‘30’

Pig Latin Data Model

Tuple

- A record that is formed by an ordered set of fields is known as a tuple, the fields can be of any type.
- A tuple is similar to a row in a table of RDBMS.
- Example − (Raja, 30)

Pig Latin Data Model

Bag

- A bag is an unordered set of tuples.
- In other words, a collection of tuples (non-unique) is known as a bag.
- Each tuple can have any number of fields (flexible schema).
- A bag is represented by '{}'. It is similar to a table in RDBMS
- Unlike a table in RDBMS,
  - it is not necessary that every tuple contain the same number of fields or
  - that the fields in the same position (column) have the same type.
- Example − {(Raja, 30), (Mohammad, 45)}
- A bag can be a field in a relation; in that context, it is known as inner bag.
- Example − {Raja, 30, {9848022338, raja@gmail.com,}}

Pig Latin Data Model

Map

- A map (or data map) is a set of key-value pairs.
- The key needs to be of type chararray and should be unique.
- The value might be of any type.
- It is represented by '[]'
- Example – [name#Raja, age#30]

Pig Latin Data Model

Relation

- A relation is a bag of tuples.
- The relations in Pig Latin are unordered (there is no guarantee that tuples are processed in any particular order).

Apache Pig Execution Modes

- You can run Apache Pig in two modes, namely,
  1. Local Mode and
  2. HDFS mode.
- **Local Mode**
  - In this mode, all the files are installed and run from your local host and local file system.
  - There is no need of Hadoop or HDFS.
  - This mode is generally used for testing purpose.
- **MapReduce Mode**
  - MapReduce mode is where we load or process the data that exists in the Hadoop File System (HDFS) using Apache Pig.
  - When Pig Latin statements are executed to process the data, a MapReduce job is invoked in the back-end

ApachePig Execution Mechanisms

Apache Pig scripts can be executed in three ways

- **Interactive Mode (Grunt shell) –**
  - ✓ You can run Apache Pig in interactive mode using the Grunt shell.
  - ✓ You can enter the Pig Latin statements and get output (using Dump operator).
- **Batch Mode (Script) –**
  - ➢ You can run Apache Pig in Batch mode by writing the Pig Latin script in a single file with **.pig** extension.
- **Embedded Mode (UDF) –**
  - ➢ Apache Pig provides the provision of defining our own functions (**U**ser **D**efined **F**unctions) in programming languages such as Java, and using them in our script.

Invoking the Grunt Shell

You can invoke the Grunt shell in a desired mode (local/MapReduce) using the **–x** option as shown below.

Local mode	MapReduce mode
Command – $ ./pig –x local	**Command –** $ ./pig -x mapreduce
Output – 15/09/28 10:13:03 INFO pig.Main: Logging error messages to: /home/Hadoop/pig_1443415383991.log 2015-09-28 10:13:04,838 [main] INFO org.apache.pig.backend.hadoop.execution engine.HExecutionEngine - Connecting to hadoop file system at: file:/// grunt>	**Output –** 15/09/28 10:28:46 INFO pig.Main: Logging error messages to: /home/Hadoop/pig_1443416326123.log 2015-09-28 10:28:46,427 [main] INFO org.apache.pig.backend.hadoop.execution engine.HExecutionEngine - Connecting to hadoop file system at: file:/// grunt>

Executing Apache Pig in Batch Mode

- You can write an entire Pig Latin script in a file and execute it using **–x command**.
- Let us suppose we have a Pig script in a file named **sample_script.pig**

Sample_script.pig

```
student = LOAD 'hdfs://localhost:9000/pig_data/student.txt' USING
   PigStorage(',') as (id:int,name:chararray,city:chararray);

Dump student;
```

Now, you can execute the script in the above file as shown below.

Local mode	MapReduce mode
$ pig -x local **Sample_script.pig**	$ pig -x mapreduce **Sample_script.pig**

Pig – An Example

- Let's look at a simple example by writing the program to calculate the maximum recorded temperature by year for weather dataset in Pig Latin.
- The complete program is only a few lines long:

```
-- max_temp.pig: Finds the maximum temperature by year
records = LOAD 'input/ncdc/micro-tab/sample.txt'
AS (year:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999 AND
quality IN (0, 1, 4, 5, 9);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
MAX(filtered_records.temperature);
DUMP max_temp;
```

Generating Examples

- Above used a small sample dataset with just a handful of rows to make it easier to follow the data flow and aid debugging.
- Creating a cut-down dataset is an art, as ideally it should be rich enough to cover all the cases to exercise your queries
- Using a random sample doesn't work well in general because join and filter operations tend to remove all random data, leaving an empty result, which is not illustrative of the general data flow.
- With the ILLUSTRATE operator, Pig provides a tool for generating a reasonably complete and concise sample dataset.

Generating Examples

Here is the output from running ILLUSTRATE on our dataset

```
grunt> ILLUSTRATE max_temp;
-------------------------------------------------------------------------------
| records     | year:chararray      | temperature:int      | quality:int      |
-------------------------------------------------------------------------------
|             | 1949                | 78                   | 1                |
|             | 1949                | 111                  | 1                |
|             | 1949                | 9999                 | 1                |
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
| filtered_records   | year:chararray    | temperature:int    | quality:int    |
-------------------------------------------------------------------------------
|                    | 1949              | 78                 | 1              |
|                    | 1949              | 111                | 1              |
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
| grouped_records  | group:chararray   | filtered_records:bag{:tuple(           |
                                         year:chararray,temperature:int,      |
                                         quality:int)}                        |
-------------------------------------------------------------------------------
|                  | 1949              | {(1949, 78, 1), (1949, 111, 1)}      |
-------------------------------------------------------------------------------
-----------------------------------------------
| max_temp     | group:chararray      | :int     |
-----------------------------------------------
|              | 1949                 | 111      |
-----------------------------------------------
```

Pig Latin

- The Pig Latin is a data flow language used by Apache Pig to analyze the data in Hadoop.
- It is a textual language that abstracts the programming from the Java MapReduce idiom into a notation.
- The Pig Latin statements are used to process the data.
- It is an operator that accepts a relation as an input and generates another relation as an output.

Pig Latin

- A Pig Latin program consists of a collection of statements.
- A statement can be thought of as an operation or a command.[5]

```
grouped_records = GROUP records BY year;
```

- Statements are usually terminated with a semicolon
- Statements can be split across multiple

```
records = LOAD 'input/ncdc/micro-tab/sample.txt'
AS (year:chararray, temperature:int, quality:int);
```

- Pig Latin has two forms of comments - **Double hyphens**

```
-- My program
DUMP A; -- What's in A?
```

C-style comments

```
/*
* 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

- As a Pig Latin program is executed, each statement is parsed in turn.
- If there are syntax errors or other (semantic) problems interpreter will halt.
- Interpreter builds a *logical plan* for every relational operation- core of a Pig Latin program.
- *Important Note* : no data processing takes place while the logical plan of the program is being constructed.
- 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*.
- You can see the logical and physical plans created by Pig using the EXPLAIN command on a relation (EXPLAIN max_temp;, for example).

Pig Latin relational operators

The relational operators that can be a part of a logical plan

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

Pig Latin relational operators

The relational operators that can be a part of a logical plan

Category	Operator	Description
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

Pig Latin

- Statements that are not added to the logical plan.
  - Diagnostic operators—DESCRIBE, EXPLAIN, and ILLUSTRATE—are provided to allow the user to interact with the logical plan for debugging purposes
  - DUMP is a sort of diagnostic operator, too, since it is used only to allow interactive debugging of small result sets or in combination with LIMIT to retrieve a few rows from a larger relation.
- The STORE statement should be used when the size of the output is more than a few lines, as it writes to a file rather than to the console.

Pig Latin

Pig Latin diagnostic operators

Operator (Shortcut)	Description
DESCRIBE (\de)	Prints a relation's schema
EXPLAIN (\e)	Prints the logical and physical plans
ILLUSTRATE (\i)	Shows a sample execution of the logical plan, using a generated subset of the input

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

Pig Latin

Pig provides commands to interact with Hadoop file systems and MapReduce as well as a few utility 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

Pig Latin

Pig provides commands to interact with Hadoop file systems and MapReduce as well as a few utility commands

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

Pig Latin

- Two commands 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.
- 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.
- Multiquery execution, where Pig executes a batch of statements in one go is used only by exec, not run.

Pig Latin - 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.

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`

Pig Latin - Expressions

Category	Expressions	Description	Examples
Comparison	`x == y, x != y`	Equals, does not equal	`quality == 0, tempera ture != 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 matches y`	Pattern matching with regular expression	`quality matches '[01459]'`
	`x is null`	Is null	`temperature is null`
	`x is not null`	Is not null	`temperature is not null`
Boolean	`x OR y`	Logical OR	`q == 0 OR q == 1`
	`x AND y`	Logical AND	`q == 0 AND r == 0`
	`NOT x`	Logical negation	`NOT q matches '[01459]'`
	`IN x`	Set membership	`q IN (0, 1, 4, 5, 9)`
Functional	*`fn(f1,f2,...)`*	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)`

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	'0.11000100000000000000001'
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']

Pig Latin - Schemas

- A relation in Pig may have an associated schema, which gives the fields in the relation names and types.
- AS clause in a LOAD statement is used to attach a schema to a relation:

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt'
>> AS (year:int, temperature:int, quality:int);
grunt> DESCRIBE records;
records: {year: int,temperature: int,quality: int}
```

- It's possible to omit type declarations completely, too:

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt'
>> AS (year, temperature, quality);
grunt> DESCRIBE records;
records: {year: bytearray,temperature: bytearray,quality: bytearray}
```

- Don't need to specify types for every field

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt'
>> AS (year, temperature:int, quality:int);
grunt> DESCRIBE records;
records: {year: bytearray,temperature: int,quality: int}
```

Pig Latin - Schemas

- Schema is optional and can be omitted by not specifying an AS clause

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt';
grunt> DESCRIBE records;
Schema for records unknown.
```

- Fields in a relation with no schema can be referenced using only positional notation: $0 refers to the first field in a relation

```
grunt> projected_records = FOREACH records GENERATE $0, $1, $2;
grunt> DUMP projected_records;
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
(1949,78,1)
grunt> DESCRIBE projected_records;
projected_records: {bytearray,bytearray,bytearray}
```

Pig Latin - Validation and nulls

- In SQL database, trying to load a string into a column that is declared to be a numeric type will fail.
- In Pig, if the value cannot be cast to the type declared in the schema, it will substitute a null value.

```
1950 0 1
1950 22 1
1950 e 1
1949 111 1
1949 78 1
```

- Pig handles the corrupt line by producing a null for the offending value

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample_corrupt.txt'
>> AS (year:chararray, temperature:int, quality:int);
grunt> DUMP records;
(1950,0,1)
(1950,22,1)
(1950,,1)
(1949,111,1)
(1949,78,1)
```

Pig Latin - Validation and nulls

- We can pull out all of the invalid records in one go

```
grunt> corrupt_records = FILTER records BY temperature is null;
grunt> DUMP corrupt_records;
(1950,,1)
```

- We can find the number of corrupt records using the following idiom

```
grunt> grouped = GROUP corrupt_records ALL;
grunt> all_grouped = FOREACH grouped GENERATE group, COUNT(corrupt_records);
grunt> DUMP all_grouped;
(all,1)
```

- Another useful technique is to use the SPLIT operator to partition the data into "good" and "bad" relations

```
grunt> SPLIT records INTO good_records IF temperature is not null,
        >> bad_records OTHERWISE;
grunt> DUMP good_records;
(1950,0,1)
(1950,22,1)
(1949,111,1)
(1949,78,1)
grunt> DUMP bad_records;
(1950,,1)
```

Pig Latin - Functions

Functions in Pig come in four types:

- *Eval function*
- *Filter function*
- *Load function*
- *Store function*

Pig Latin - Functions

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 `()`.
	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 `[]`.
	TOP	Calculates the top *n* tuples in a bag.
	TOTUPLE	Converts one or more expressions to a tuple. A synonym for `{}`.
Filter	IsEmpty	Tests whether a bag or map is empty.

Pig Latin - Functions

Category	Function	Description
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.[a]
	TextLoader	Loads relations from a plain-text format. Each line corresponds to a tuple whose single field is the line of text.
	JsonLoader, JsonStorage	Loads or stores relations from or to a (Pig-defined) JSON format. Each tuple is stored on one line.
	AvroStorage	Loads or stores relations from or to Avro datafiles.
	ParquetLoader, ParquetStorer	Loads or stores relations from or to Parquet files.
	OrcStorage	Loads or stores relations from or to Hive ORCFiles.
	HBaseStorage	Loads or stores relations from or to HBase tables.

Pig Latin - Macros

- Macros provide a way to package reusable pieces of Pig Latin code from within Pig Latin itself.
- For example, we can extract the part of our Pig Latin program that performs grouping on a relation and then finds the maximum value in each group by defining a macro as follows:

```
DEFINE max_by_group(X, group_key, max_field) RETURNS Y {
A = GROUP $X by $group_key;
$Y = FOREACH A GENERATE group, MAX($X.$max_field);
};
```

- The macro is used as follows:

```
records = LOAD 'input/ncdc/micro-tab/sample.txt'
AS (year:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999 AND
      quality IN (0, 1, 4, 5, 9);
max_temp = max_by_group(filtered_records, year, temperature);
DUMP max_temp
```

- At runtime, Pig will expand the macro using the macro definition

Pig Latin - User-Defined Functions

- Pig's designers realized that the ability to plug in custom code is crucial for all but the most trivial data processing jobs.
- For this reason, they made it easy to define and use user-defined functions.
- Can use Java UDFs, but be aware that you can also write UDFs in Python, JavaScript, Ruby, or Groovy

Pig Latin - User-Defined Functions

A Filter UDF

- Let's demonstrate by writing a filter function for filtering out weather records that do not have a temperature quality reading of satisfactory
- The idea is to change this line:

```
filtered_records = FILTER records BY temperature != 9999 AND
quality IN (0, 1, 4, 5, 9);
```

to:

```
filtered_records = FILTER records BY temperature != 9999 AND
isGood(quality);
```

- This achieves two things:
  1. it makes the Pig script a little more concise, and
  2. it encapsulates the logic in one place so that it can be easily reused in other scripts.

Pig Latin - User-Defined Functions

```
public class IsGoodQuality extends FilterFunc
{
        @Override
        public Boolean exec(Tuple tuple) throws IOException
        {
                if (tuple == null || tuple.size() == 0)
                {
                        return false;
                }
                try
                 {
                        Object object = tuple.get(0);
                        if (object == null)
                        {
                                return false;
                            }
                        int i = (Integer) object;
                        return i == 0 || i == 1 || i == 4 || i == 5 || i == 9;
                }
                catch (ExecException e) { hrow new IOException(e); }
        }
}
```

Pig Latin - User-Defined Functions

- To use the new function, compile it and package it in a JAR file
- Then we tell Pig about the JAR file with the REGISTER operator, which is given the local path to the filename (and is *not* enclosed in quotes):

```
grunt> REGISTER pig-examples.jar;
```

- Finally, we can invoke the function:

```
grunt> filtered_records = FILTER records BY temperature !=
9999 AND com.hadoopbook.pig.IsGoodQuality(quality);
```

Pig Latin - Data Processing Operators

- Loading and Storing Data
- Filtering Data
- Grouping and Joining Data
- Sorting Data
- Combining and Splitting Data

Pig Latin - Data Processing Operators

Loading and Storing Data

- Seen how to load data from external storage for processing in Pig.
- Here's an example of using PigStorage to store tuples as plain-text values separated by a colon character:

```
grunt> STORE A INTO 'out' USING PigStorage(':');
grunt> cat out
Joe:cherry:2
Ali:apple:3
Joe:banana:2
Eve:apple:7
```

- Other built-in storage functions were
  - TextLoader
  - JsonLoader, JsonStorage
  - AvroStorage
  - ParquetLoader, ParquetStorer
  - OrcStorage,
  - HBaseStorage

Pig Latin - Data Processing Operators

Filtering Data

- Once you have some data loaded into a relation, often the next step is to filter it to remove the data that you are not interested in.
- By filtering early in the processing pipeline, you minimize the amount of data flowing through the system, which can improve efficiency.

```
records = LOAD 'input/ncdc/micro-tab/sample.txt'
AS (year:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999 AND
                   quality IN (0, 1, 4, 5, 9);
macro_max_by_group_A_0 = GROUP filtered_records by (year);
max_temp = FOREACH macro_max_by_group_A_0 GENERATE group,
MAX(filtered_records.(temperature));
DUMP max_temp
```

Pig Latin - Data Processing Operators

FOREACH...GENERATE

- Seen how to remove rows from a relation using the FILTER operator
- The FOREACH...GENERATE operator is used to act on every row in a relation.
- It can be used to remove fields or to generate new ones.

```
grunt> DUMP A;
(Joe,cherry,2)
(Ali,apple,3)
(Joe,banana,2)
(Eve,apple,7)
grunt> B = FOREACH A GENERATE $0, $2+1, 'Constant';
grunt> DUMP B;
(Joe,3,Constant)
(Ali,4,Constant)
(Joe,3,Constant)
(Eve,8,Constant)
```

- Created a new relation, B, with three fields (third field of A ($2) with 1 added to it)

Pig Latin - Data Processing Operators

The FOREACH...GENERATE operator has a nested form to support more complex processing.

```
-- year_stats.pig
REGISTER pig-examples.jar;
DEFINE isGood com.hadoopbook.pig.IsGoodQuality();
records = LOAD 'input/ncdc/all/19{1,2,3,4,5}0*'
USING com.hadoopbook.pig.CutLoadFunc('5-10,11-15,16-19,88-92,93-93')
AS (usaf:chararray, wban:chararray, year:int, temperature:int, quality:int);
grouped_records = GROUP records BY year PARALLEL 30;
year_stats = FOREACH grouped_records {
    uniq_stations = DISTINCT records.usaf;
    good_records = FILTER records BY isGood(quality);
    GENERATE FLATTEN(group), COUNT(uniq_stations) AS station_count,
    COUNT(good_records) AS good_record_count, COUNT(records) AS record_count;
}
DUMP year_stats;
```

Running it on a few years' worth of data, we get the following:

```
(1920,8L,8595L,8595L)
(1950,1988L,8635452L,8641353L)
(1930,121L,89245L,89262L)
(1910,7L,7650L,7650L)
(1940,732L,1052333L,1052976L)
```

Pig Latin - Data Processing Operators

STREAM

- The STREAM operator allows you to transform data in a relation using an external program or script.
- It is named by analogy with Hadoop Streaming
- STREAM can use built-in commands with arguments.
- Here is an example that uses the Unix cut command to extract the second field of each tuple in A.

```
grunt> C = STREAM A THROUGH `cut -f 2`;
grunt> DUMP C;
(cherry)
(apple)
(banana)
(apple)
```

- STREAM operator uses PigStorage to serialize and deserialize relations to and from the program's standard input and output streams.
- Pig streaming is most powerful when you write custom processing scripts.

Pig Latin - Data Processing Operators

Grouping and Joining Data

- Joining datasets in MapReduce takes some work on the part of the programmer
- Pig has very good built-in support for join operations, making it much more approachable.
- Since the large datasets that are suitable for analysis by Pig (and MapReduce in general) are not normalized, however, joins are used more infrequently in Pig than they are in SQL.

Pig Latin - Data Processing Operators

JOIN

- Let's look at an example of an inner join. Consider the relations A and B:

```
grunt> DUMP A;
(2,Tie)
(4,Coat)
(3,Hat)
(1,Scarf)

grunt> DUMP B;
(Joe,2)
(Hank,4)
(Ali,0)
(Eve,3)
(Hank,2)
```

- We can join the two relations on the numerical (identity) field in each:

```
grunt> C = JOIN A BY $0, B BY $1;
grunt> DUMP C;
(2,Tie,Hank,2)
(2,Tie,Joe,2)
(3,Hat,Eve,3)
(4,Coat,Hank,4)
```

Pig Latin - Data Processing Operators

JOIN

- Pig also supports outer joins using a syntax that is similar to SQL's
- For example:

```
grunt> C = JOIN A BY $0 LEFT OUTER, B BY $1;
grunt> DUMP C;
(1,Scarf,,)
(2,Tie,Hank,2)
(2,Tie,Joe,2)
(3,Hat,Eve,3)
(4,Coat,Hank,4)
```

```
grunt> DUMP A;
(2,Tie)
(4,Coat)
(3,Hat)
(1,Scarf)

grunt> DUMP B;
(Joe,2)
(Hank,4)
(Ali,0)
(Eve,3)
(Hank,2)
```

Pig Latin - Data Processing Operators

COGROUP

- JOIN always gives a flat structure: a set of tuples.
- The COGROUP statement is similar to JOIN, but instead creates a nested set of output tuples.
- This can be useful if you want to exploit the structure in subsequent statements:

```
grunt> D = COGROUP A BY $0, B BY $1;
grunt> DUMP D;
(0,{},{(Ali,0)})
(1,{(1,Scarf)},{})
(2,{(2,Tie)},{(Hank,2),(Joe,2)})
(3,{(3,Hat)},{(Eve,3)})
(4,{(4,Coat)},{(Hank,4)})
```

```
grunt> DUMP A;
(2,Tie)
(4,Coat)
(3,Hat)
(1,Scarf)

grunt> DUMP B;
(Joe,2)
(Hank,4)
(Ali,0)
(Eve,3)
(Hank,2)
```

- COGROUP generates a tuple for each unique grouping key.

Pig Latin - Data Processing Operators

- You can suppress rows with empty bags by using the INNER keyword, which gives COGROUP inner join semantics.
- The INNER keyword is applied per relation, so the following suppresses rows only when relation A has no match

```
grunt> E = COGROUP A BY $0 INNER, B BY $1;
grunt> DUMP E;
(1,{(1,Scarf)},{})
(2,{(2,Tie)},{(Hank,2),(Joe,2)})
(3,{(3,Hat)},{(Eve,3)})
(4,{(4,Coat)},{(Hank,4)})
```

- We can flatten this structure to discover who bought each of the items in relation A:

```
grunt> F = FOREACH E GENERATE FLATTEN(A), B.$0;
grunt> DUMP F;
(1,Scarf,{})
(2,Tie,{(Hank),(Joe)})
(3,Hat,{(Eve)})
(4,Coat,{(Hank)})
```

```
grunt> DUMP A;
(2,Tie)
(4,Coat)
(3,Hat)
(1,Scarf)

grunt> DUMP B;
(Joe,2)
(Hank,4)
(Ali,0)
(Eve,3)
(Hank,2)
```

Pig Latin - Data Processing Operators

- Using a combination of COGROUP, INNER, and FLATTEN (which removes nesting) it's possible to simulate an (inner) JOIN:

```
grunt> G = COGROUP A BY $0 INNER, B BY $1 INNER;
grunt> H = FOREACH G GENERATE FLATTEN($1), FLATTEN($2);
grunt> DUMP H;
(2,Tie,Hank,2)
(2,Tie,Joe,2)
(3,Hat,Eve,3)
(4,Coat,Hank,4)
```

- This gives the same result as JOIN A BY $0, B BY $1.

Pig Latin - Data Processing Operators

- Example of a join in Pig for calculating the maximum temperature for every station over a time period controlled by the input:

```
-- max_temp_station_name.pig
REGISTER pig-examples.jar;
DEFINE isGood com.hadoopbook.pig.IsGoodQuality();
stations = LOAD 'input/ncdc/metadata/stations-fixed-width.txt'
USING com.hadoopbook.pig.CutLoadFunc('1-6,8-12,14-42')
AS (usaf:chararray, wban:chararray, name:chararray);
trimmed_stations = FOREACH stations GENERATE usaf, wban, TRIM(name);
records = LOAD 'input/ncdc/all/191*'
USING com.hadoopbook.pig.CutLoadFunc('5-10,11-15,88-92,93-93')
AS (usaf:chararray, wban:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999 AND isGood(quality);
grouped_records = GROUP filtered_records BY (usaf, wban) PARALLEL 30;
max_temp = FOREACH grouped_records GENERATE FLATTEN(group),
MAX(filtered_records.temperature);
max_temp_named = JOIN max_temp BY (usaf, wban), trimmed_stations BY (usaf, wban)
PARALLEL 30;
max_temp_result = FOREACH max_temp_named GENERATE $0, $1, $5, $2;
STORE max_temp_result INTO 'max_temp_by_station';
```

Pig Latin - Data Processing Operators

CROSS

- Pig Latin includes the cross-product operator (also known as the Cartesian product)
- CROSS joins every tuple in a relation with every tuple in a second relation
- The size of the output is the product of the size of the inputs, potentially making the output very large:

grunt> **I = CROSS A, B;**

grunt> **DUMP I;**

(2,Tie,Joe,2)	(2,Tie,Joe,2)
(2,Tie,Hank,4)	(2,Tie,Hank,4)
(2,Tie,Ali,0)	(2,Tie,Ali,0)
(2,Tie,Eve,3)	(2,Tie,Eve,3)
(2,Tie,Hank,2)	(2,Tie,Hank,2)
(4,Coat,Joe,2)	(4,Coat,Joe,2)
(4,Coat,Hank,4)	(4,Coat,Hank,4)
(4,Coat,Ali,0)	(4,Coat,Ali,0)
(4,Coat,Eve,3)	(4,Coat,Eve,3)
(4,Coat,Hank,2)	(4,Coat,Hank,2)
(3,Hat,Joe,2)	(3,Hat,Joe,2)

Pig Latin - Data Processing Operators

GROUP

- Where COGROUP groups the data in two or more relations, the GROUP statement groups the data in a **single relation**.
- GROUP supports grouping by more than equality of keys: you can use an expression or user-defined function as the group key.
- For example, consider the following relation A:

```
grunt> DUMP A;
(Joe,cherry)
(Ali,apple)
(Joe,banana)
(Eve,apple)
```

- Let's group by the number of characters in the second field:

```
grunt> B = GROUP A BY SIZE($1);
grunt> DUMP B;
(5,{(Eve,apple),(Ali,apple)})
(6,{(Joe,banana),(Joe,cherry)})
```

Pig Latin - Data Processing Operators

GROUP

- There are two special grouping operations: ALL and ANY.
- ALL groups all the tuples in a relation in a single group, as if the GROUP function were a constant:

```
grunt> C = GROUP A ALL;
grunt> DUMP C;
(all,{(Eve,apple),(Joe,banana),(Ali,apple),(Joe,cherry)})
```

- Note that there is no BY in this form of the GROUP statement.
- The ALL grouping is commonly used to count the number of tuples in a relation
- The ANY keyword is used to group the tuples in a relation randomly, which can be useful for sampling.

Pig Latin - Data Processing Operators

Sorting Data

- Relations are unordered in Pig. Consider a relation A:

```
grunt> DUMP A;
(2,3)
(1,2)
(2,4)
```

- No guarantee which order the rows will be processed in.
- When retrieving the contents of A using DUMP or STORE, the rows may be written in any order.
- If you want to impose an order on the output using ORDER operator to sort a relation by one or more fields.
- The following example sorts A by the first field in ascending order and by the second field in descending order:

```
grunt> B = ORDER A BY $0, $1 DESC;
grunt> DUMP B;
(1,2)
(2,4)
(2,3)
```

Pig Latin - Data Processing Operators

- Any further processing on a sorted relation is not guaranteed to retain its order.
- For example:

```
grunt> C = FOREACH B GENERATE *;
```

- Even though relation C has the same contents as relation B, its tuples may be emitted in any order by a DUMP or a STORE.
- It is for this reason that it is usual to perform the ORDER operation just before retrieving the output.

Pig Latin - Data Processing Operators

- LIMIT statement is useful for limiting the number of results as a quick-and-dirty way to get a sample of a relation.
- It can be used immediately after the ORDER statement to retrieve the first *n* tuples.
- Usually, LIMIT will select any *n* tuples from a relation, but when used immediately after an ORDER statement, the order is retained (in an exception to the rule that processing a relation does not retain its order):

```
grunt> D = LIMIT B 2;
grunt> DUMP D;
(1,2)
(2,4)
```

- If limit is greater than number of tuples in relation, all tuples are returned.
- Should always use LIMIT if you are not interested in the entire output.

Pig Latin - Data Processing Operators

Combining and Splitting Data

- Several relations that need combine into one.
- For this, the UNION statement is used. For example:

```
grunt> DUMP A;
(2,3)
(1,2)
(2,4)
grunt> DUMP B;
(z,x,8)
(w,y,1)
grunt> C = UNION A, B;
grunt> DUMP C;
(2,3)
(z,x,8)
(1,2)
(w,y,1)
(2,4)
```

Pig Latin - Data Processing Operators

Combining and Splitting Data

- Pig merges schemas from the relations that UNION is operating on.
- In this case, they are incompatible, so C has no schema:

```
grunt> DESCRIBE A;
A: {f0: int,f1: int}
grunt> DESCRIBE B;
B: {f0: chararray,f1: chararray,f2: int}
grunt> DESCRIBE C;
```

- Schema for C unknown.
- If the output relation has no schema, your script needs to be able to handle tuples that vary in the number of fields and/or types.
- The SPLIT operator is the opposite of UNION: it partitions a relation into two or more relations.

Applications on Big Data Using Hive

- Hive is a data warehouse infrastructure tool to process structured data in Hadoop.
- Resides on top of Hadoop to summarize Big Data, and makes querying and analyzing easy.
- Initially Hive was developed by Facebook, later Apache Software Foundation took it up and developed it further as an open source under the name Apache Hive.
- Used by Amazon for its Amazon Elastic MapReduce.
- Hive is not
  - A relational database
  - A design for Online Transaction Processing (OLTP)
  - A language for real-time queries and row-level updates
- Using Hive, we **can skip requirement** of traditional approach of writing complex MapReduce programs

Hive

- Apache Hive is a data warehouse software project built on top of Apache Hadoop for providing data query and analysis.
- Hive gives an SQL-like interface to query data stored in various databases and file systems that integrate with Hadoop.
- Traditional SQL queries must be implemented in the MapReduce Java API to execute SQL applications and queries over distributed data.
- Hive provides the necessary SQL abstraction to integrate SQL-like queries (HiveQL) into underlying Java without need to implement queries in the low level Java API.

--- Wikipedia

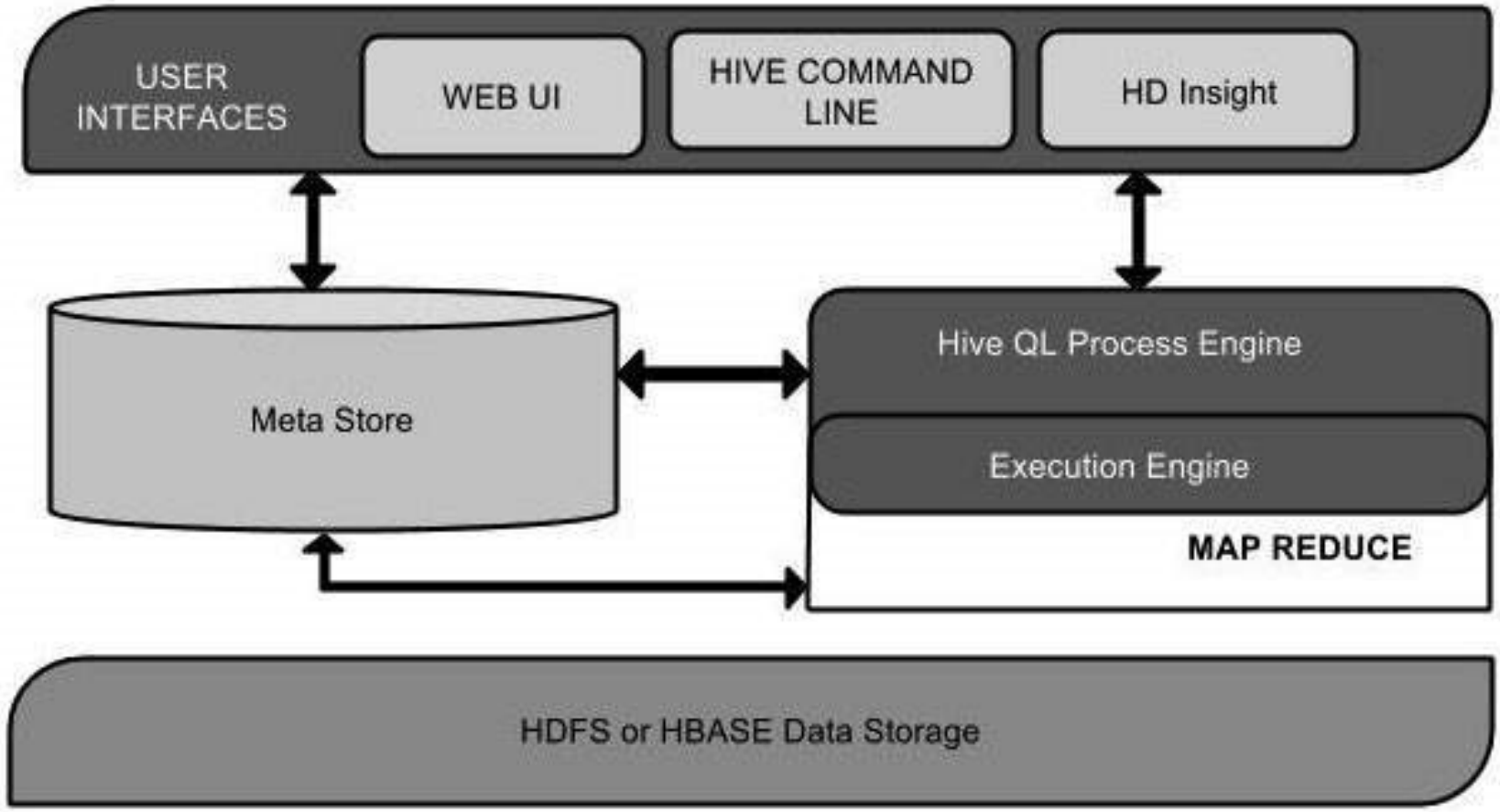
Features of Hive

- It stores schema in a database and processed data into HDFS.
- It is designed for OLAP (Online Analytical Processing)
- Hive is fast and scalable.
- It provides SQL-like queries (i.e., HQL) that are implicitly transformed to MapReduce or Spark jobs.
- It is capable of analyzing large datasets stored in HDFS.
- It allows different storage types such as plain text, RCFile, and HBase.
- It uses indexing to accelerate queries.
- It can operate on compressed data stored in the Hadoop ecosystem.
- It supports user-defined functions (UDFs) where user can provide its functionality.

Limitations of Hive

- Hive is not capable of handling real-time data.
- It is not designed for online transaction processing (OLTP).
- Hive queries contain high latency.

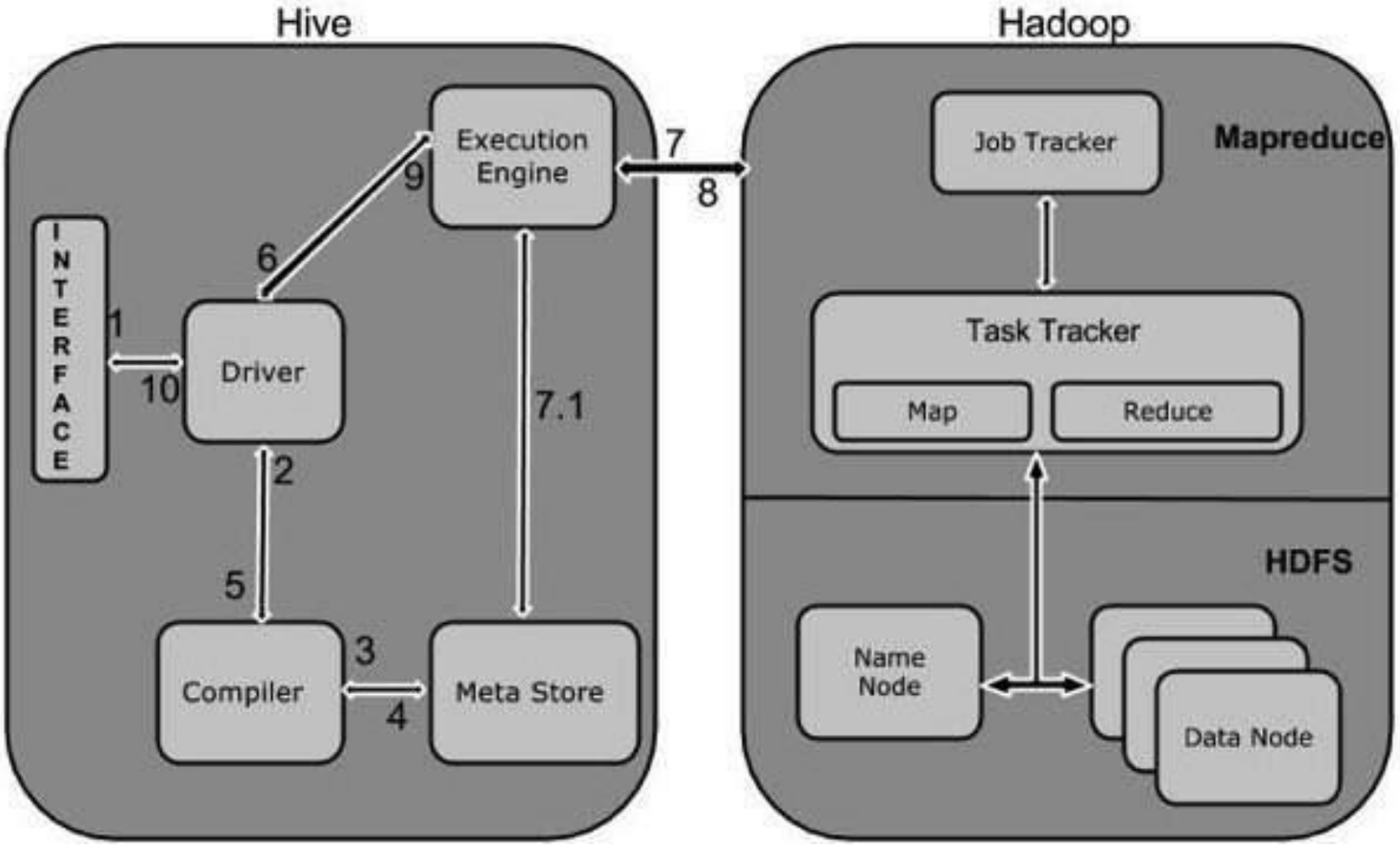
Architecture of Hive



Architecture of Hive

Unit Name	Operation
User Interface	Hive is a data warehouse infrastructure software that can create interaction between user and HDFS. The user interfaces that Hive supports are Hive Web UI, Hive command line, and Hive HD Insight (In Windows server).
Meta Store	Hive chooses respective database servers to store the schema or Metadata of tables, databases, columns in a table, their data types, and HDFS mapping.
HiveQL Process Engine	HiveQL is similar to SQL for querying on schema info on the Metastore. It is one of the replacements of traditional approach for MapReduce program. Instead of writing MapReduce program in Java, we can write a query for MapReduce job and process it.
Execution Engine	The conjunction part of HiveQL process Engine and MapReduce is Hive Execution Engine. Execution engine processes the query and generates results as same as MapReduce results.
HDFS or HBASE	Hadoop distributed file system or HBASE are the data storage techniques to store data into file system.

Working of Hive



Hive Data Hierarchy

- Hive is organised hierarchically into:
  - Databases: namespaces that separate tables and other objects
  - Tables: homogeneous units of data with the same schema
    - Analogous to tables in an RDBMS
  - Partitions: determine how the data is stored
    - Allow efficient access to subsets of the data
  - Buckets/clusters
    - For sub-sampling within a partition
    - Join optimization

Hive Vs Map Reduce

Feature	Hive	Map Reduce
Language	It Supports SQL like query language for interaction and for Data modeling	•It compiles language with two main tasks present in it. One is map task, and another one is a reducer. •We can define these task using Java or Python
Level of abstraction	Higher level of Abstraction on top of HDFS	Lower level of abstraction
Efficiency in Code	Comparatively lesser than Map reduce	Provides High efficiency
Extent of code	Less number of lines code required for execution	More number of lines of codes to be defined
Type of Development work required	Less Development work required	More development

Apache Pig Vs Hive

- Both Apache Pig and Hive are used to create MapReduce jobs.
- In some cases, Hive operates on HDFS in a similar way Apache Pig does.

Apache Pig	Hive
Apache Pig uses a language called **Pig Latin**. It was originally created at **Yahoo**.	Hive uses a language called **HiveQL**. It was originally created at **Facebook**.
Pig Latin is a data flow language.	HiveQL is a query processing language.
Pig Latin is a procedural language and it fits in pipeline paradigm.	HiveQL is a declarative language.
Apache Pig can handle structured, unstructured, and semi-structured data.	Hive is mostly for structured data.

Hive vs Pig vs SQL

CRITERIA	HIVE	PIG	SQL
Languages used	Uses HiveQL, a declarative language	Uses Pig latin, a procedural data flow languages	SQL itself is a declarative language
Definition	An open source built with an analytical focus used for Analytical queries	An open source and high-level data flow language with a Multi-query approach	General purpose database language for analytical and transactional queries
Developed by	Facebook	Yahoo	Oracle
Suitable for	Batch processing OLAP (Online Analytical Processing)	Complex & nested data structure	Business demands for fast data analysis
Operational for	Structured data	Structured and semi-structured data	Relational database management
Compatibility with MapReduce	Yes	Yes	Yes
Schema Support	Support schema for data insertion	Doesn't support schema	Strictly support schema for data storage
Mainly Used by	Data Analysts	Researchers and Programmers	Data Analysts, Data Scientists, and Programmers

Hive

- Hive runs on your workstation and converts your SQL query into a series of jobs for execution on a Hadoop cluster.
- Hive organizes data into tables, which provide a means for attaching structure to data stored in HDFS.
- Metadata—such as table schemas— is stored in a database called the metastore.
- When starting out with Hive, it is convenient to run the metastore on your local machine.
- In this configuration, which is the default, the Hive table definitions that you create will be local to your machine, so you can't share them with other users.

Hive - An Example

- Let's see how to use Hive to run a query on the weather dataset
- The first step is to load the data into Hive's managed storage.
- Here we'll have Hive use the local filesystem for storage
- Just like an RDBMS, Hive organizes its data into tables.
- We create a table to hold the weather data using the CREATE TABLE statement:

```
CREATE TABLE records (year STRING, temperature INT, quality INT)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t';
```

- The ROW FORMAT clause, however, is particular to HiveQL.
- Declaration is saying that each row in data file is tab-delimited text.
- Hive expects there to be three fields in each row, corresponding to table columns, with fields separated by tabs and rows by newlines.

Hive - An Example

- Next, we can populate Hive with the data.

  LOAD DATA LOCAL INPATH 'input/ncdc/micro-tab/sample.txt'
  OVERWRITE INTO TABLE records;

- Running this command tells Hive to put the specified local file in its warehouse directory.
- There is no attempt, for example, to parse the file and store it in an internal database format, because Hive does not mandate any particular file format.
- Files are stored verbatim; they are not modified by Hive.
- Files for records table are found in /user/hive/warehouse/records
- OVERWRITE keyword in the LOAD DATA statement tells Hive to delete any existing files in the directory for the table.

Hive - An Example

- Now that the data is in Hive, we can run a query against it:

```
hive> SELECT year, MAX(temperature)
> FROM records
> WHERE temperature != 9999 AND quality IN (0, 1, 4, 5, 9)
> GROUP BY year;
1949 111
1950 22
```

- This SQL query is unremarkable.
- Hive transforms this query into a job, which it executes on our behalf
- Hive's ability to execute SQL queries against our raw data gives Hive its power

Hive - Execution engines

- Hive was originally written to use MapReduce as its execution engine
- Possible to run Hive using Apache Tez as its execution engine, and Apache 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 (at the request of the Hive planner).
- The execution engine is controlled by the hive.execution.engine property, which defaults to mr (for MapReduce).
- It's easy to switch the execution engine on a per-query basis, so you can see the effect of a different engine on a particular query.
- Set Hive to use Tez as follows:

```
hive> SET hive.execution.engine=tez;
```

Hive Services

- The Hive shell is only one of several services that you can run using the hive command.
- You can specify the service to run using the --service option.
- Type hive –service help to get a list of available service names; some of the most useful ones

Hive Services

Cli :

- The command-line interface to Hive (the shell). This is the default service.

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).

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.

Hive Services

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.
- Can use Hue for a more fully featured Hadoop web interface that includes applications for running Hive queries and browsing the Hive metastore.

jar

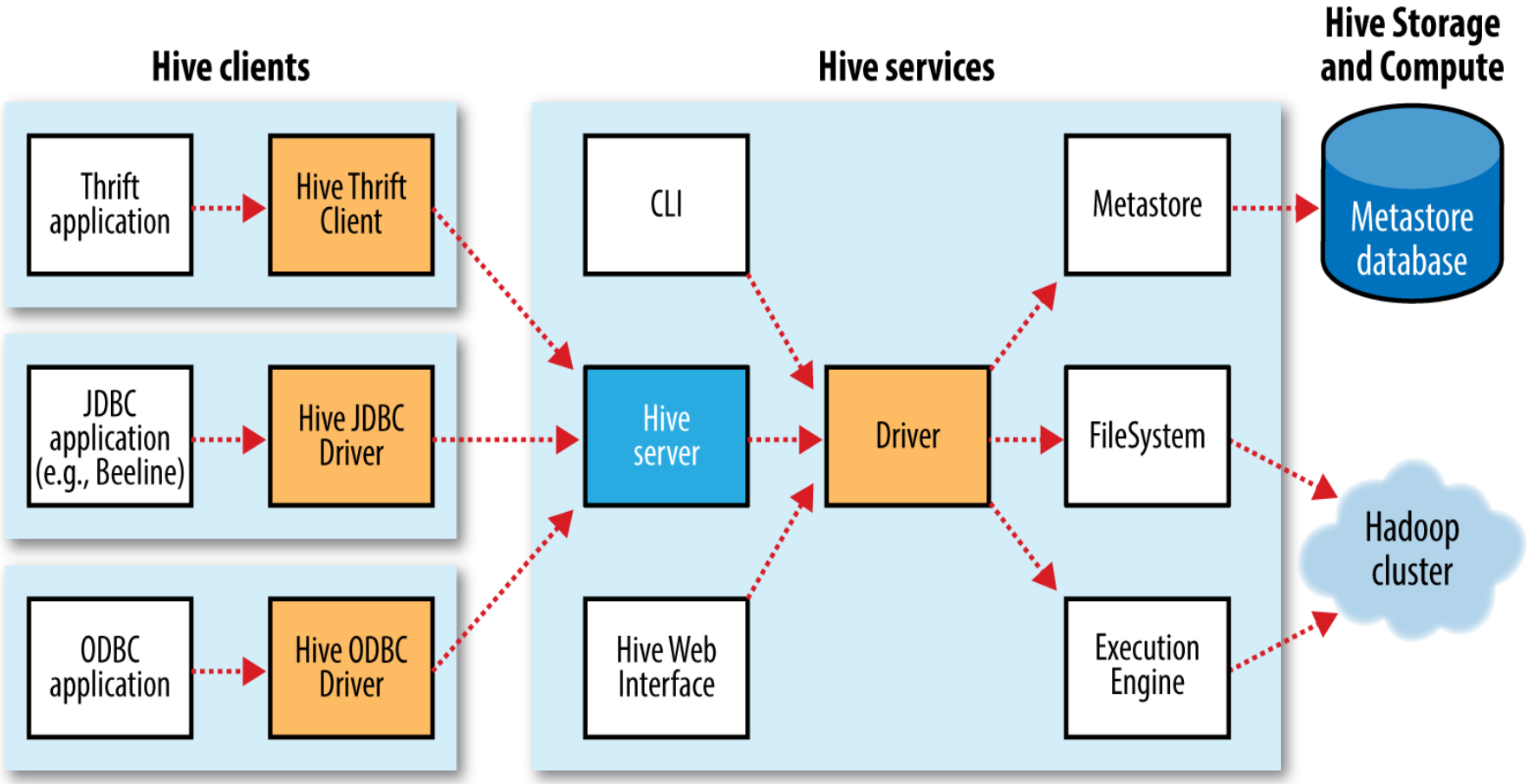
- The Hive equivalent of hadoop jar, a convenient way to run Java applications that includes both Hadoop and Hive classes on the classpath.

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.
- Set the METASTORE_PORT environment variable (or use the -p command-line option) to specify the port the server will listen on (defaults to 9083).

Hive Clients

- If you run Hive as a server (hive --service hiveserver2), there are a number of different mechanisms for connecting to it from applications



Hive architecture

Hive Clients

Thrift Client

- The Hive server is exposed as a Thrift service, so it's possible to interact with it using any programming language that supports Thrift.
- There are third-party projects providing clients for Python and Ruby

JDBC driver

- Hive provides a Type 4 (pure Java) JDBC driver, defined in the class org.apache.hadoop.hive.jdbc.HiveDriver.
- When configured with a JDBC URI of the form jdbc:hive2://*host*:*port*/*dbname*, a Java application will connect to a Hive server running in a separate process at the given host and port.
- You may alternatively choose to connect to Hive via JDBC in embedded mode using the URI jdbc:hive2://.
- In this mode, Hive runs in the same JVM as the application invoking it; there is no need to launch it as a standalone server
- The Beeline CLI uses the JDBC driver to communicate with Hive.

Hive Clients

ODBC driver

- An ODBC driver allows applications that support the ODBC protocol (such as business intelligence software) to connect to Hive.
- The Apache Hive distribution does not ship with an ODBC driver, but several vendors make one freely available.
- Like the JDBC driver, ODBC drivers use Thrift to communicate with the Hive server.

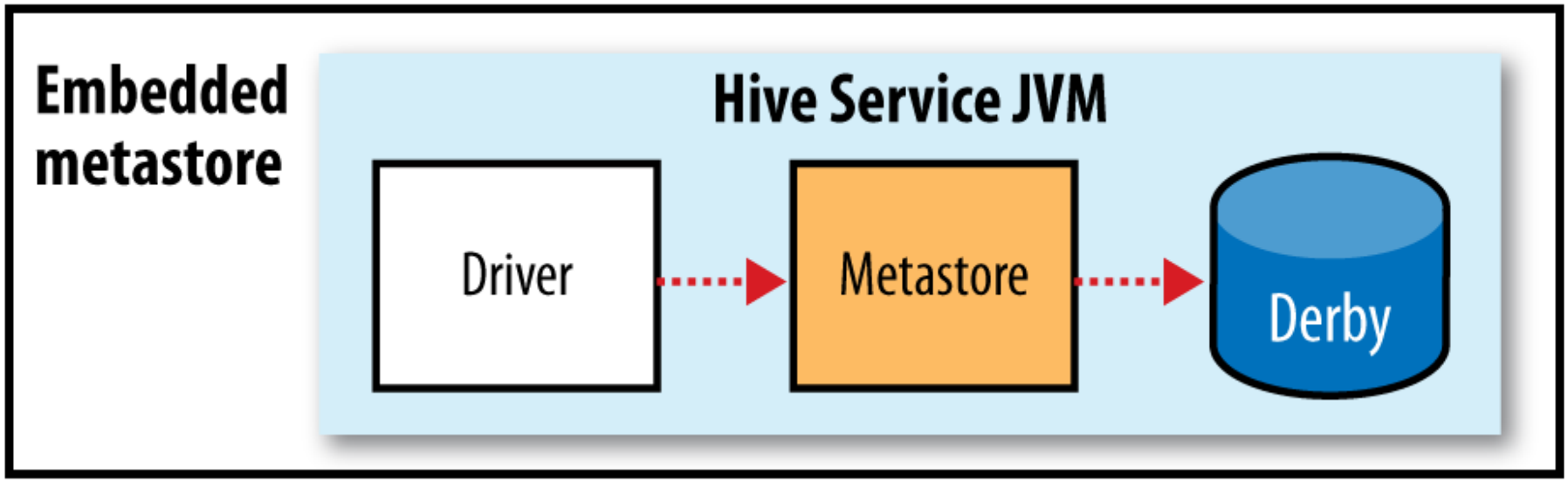
Metastore

- The *metastore* is the central repository of Hive metadata. The metastore is divided into two pieces:
  1. a service and
  2. the backing store for the data.
- Metastore configurations
  - embedded metastore
  - Local metastore
  - Remote metastore

Metastore

Embedded Metastore

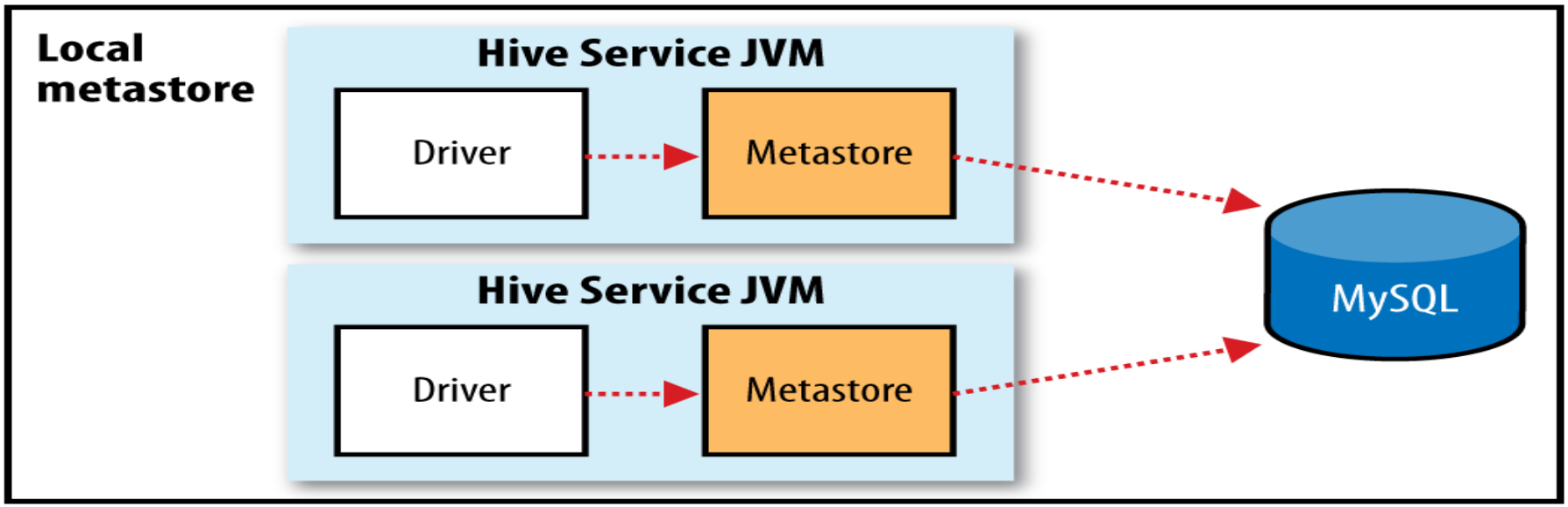
- By default, the metastore service runs in the same JVM as the Hive service and contains an embedded Derby database instance backed by the local disk.
- This is called the *embedded metastore* configuration.
- Drawback: have only one Hive session open at a time that accesses the same metastore.



Metastore

Local Metastore

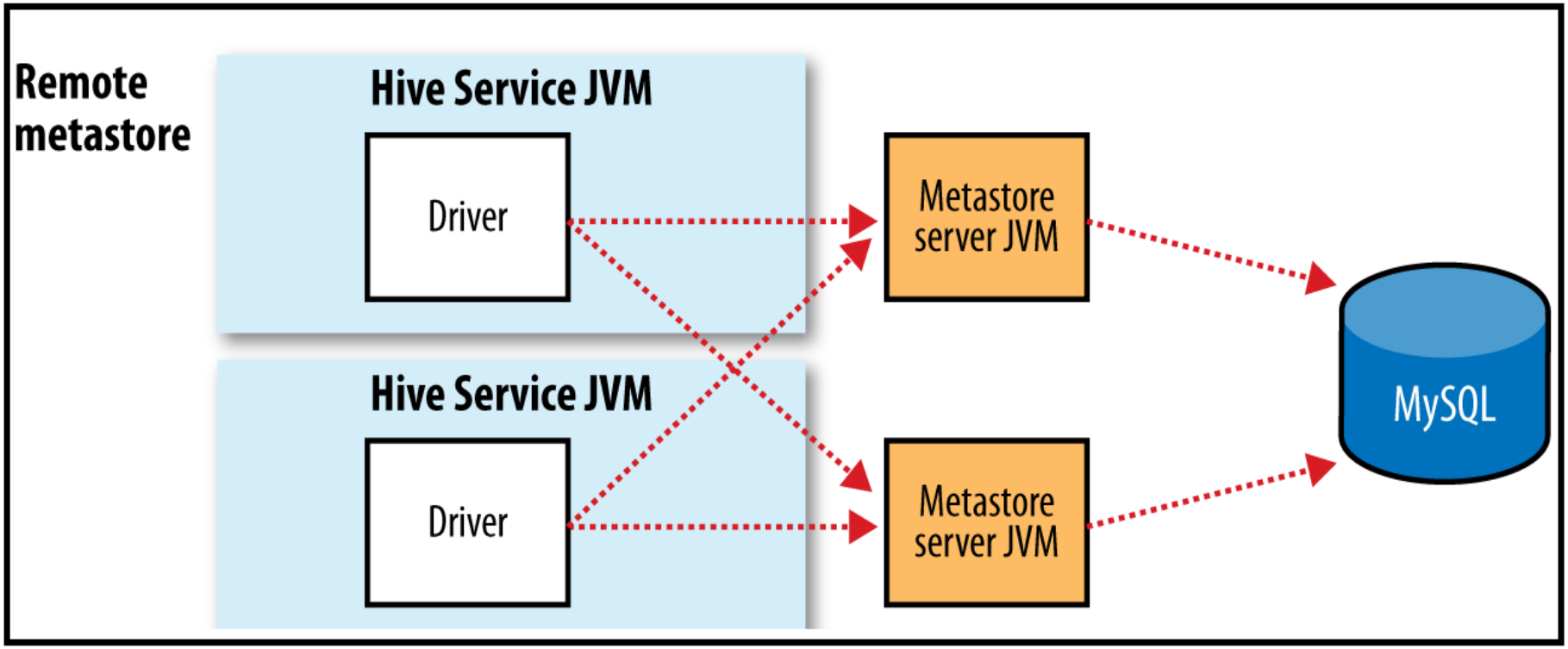
- The solution to supporting multiple sessions (and therefore multiple users) is to use a standalone database.
- This configuration is referred to as a *local metastore*, since the metastore service still runs in the same process as the Hive service but connects to a database running in a separate process
- Any JDBC-compliant database may be used by setting the javax.jdo.option.* configuration properties



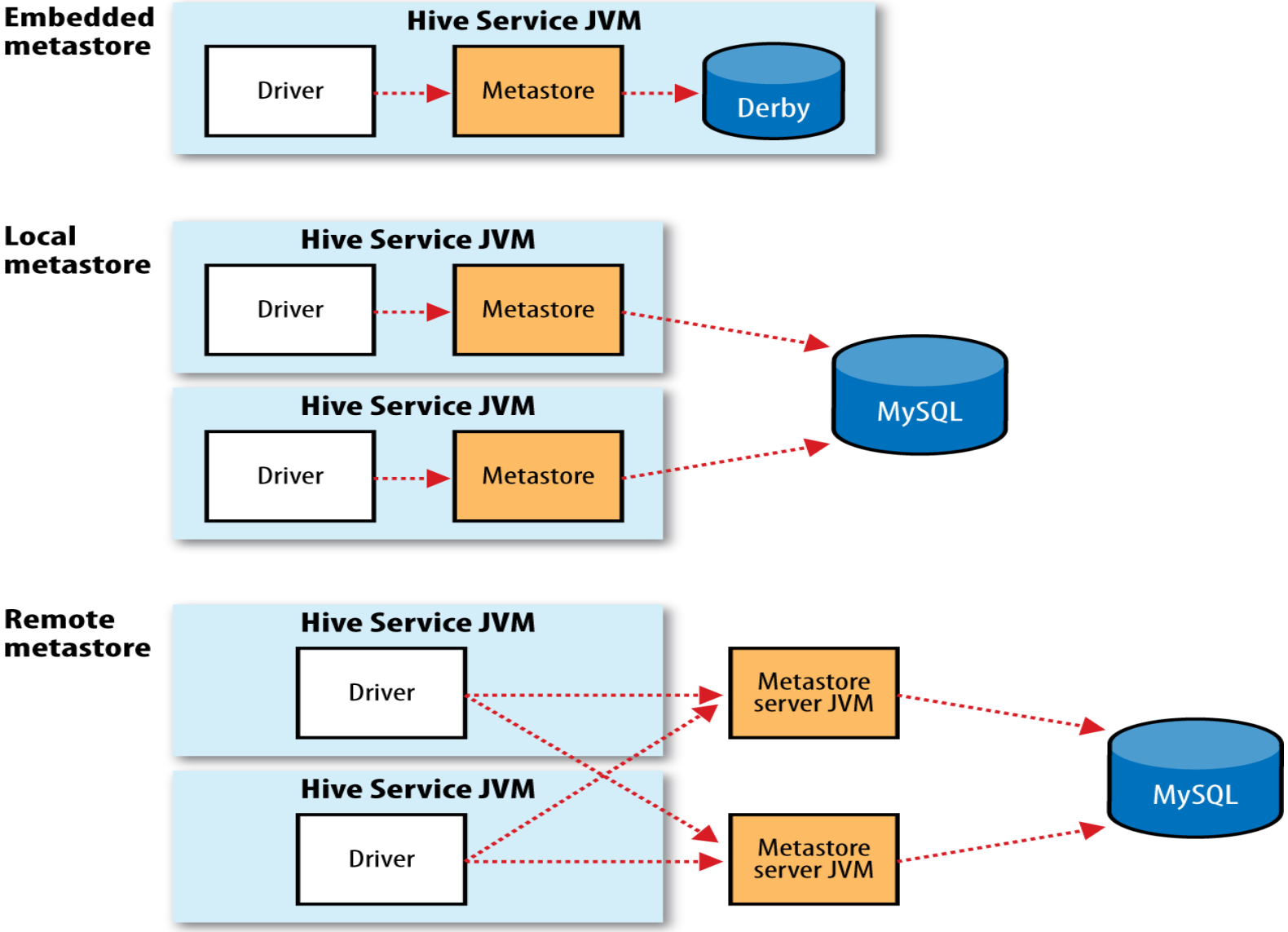
Metastore

Remote Metastore

- 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 database tier can be completely firewalled off, and clients don't need database credentials.



Metastore Configurations



Important metastore configuration properties

Property name	Type	Default value	Description
hive.metastore.warehouse.dir	URI	/user/hive/warehouse	The directory relative to fs.defaultFS where managed tables are stored.
hive.metastore.uris	Comma-separated URIs	Not set	If not set (the default), use an in-process metastore; otherwise, connect to one or more remote metastores, specified by a list of URIs. Clients connect in a round-robin fashion when there are multiple remote servers.
javax.jdo.option.ConnectionURL	URI	jdbc:derby:;databaseName=metastore_db;create=true	The JDBC URL of the metastore database.
javax.jdo.option.ConnectionDriverName	String	org.apache.derby.jdbc.EmbeddedDriver	The JDBC driver classname.
javax.jdo.option.ConnectionUserName	String	APP	The JDBC username.
javax.jdo.option.ConnectionPassword	String	mine	The JDBC password.

Comparison with Traditional Databases

Hive	RDBMS
It is used to maintain data warehouse.	It is used to maintain database.
It uses HQL (Hive Query Language).	It uses SQL (Structured Query Language).
Schema varies in it.	Schema is fixed in RDBMS.
Normalized and de-normalized both type of data is stored.	Normalized data is stored.
Table in hive are dense.	Tables in rdms are sparse.
It supports automation partition.	It doesn't support partitioning.
Sharding method is used for partition.	No partition method is used.

Comparison with Traditional Databases

Hive	RDBMS
Schema on READ – it's does not verify the schema while it's loaded the data	Schema on WRITE – table schema is enforced at data load time i.e if the data being loaded does't conformed on schema in that case it will rejected
It's very easily scalable at low cost	Not much Scalable, costly scale up.
It's based on hadoop notation that is Write once and read many times	In traditional database we can read and write many time
Record level updates is not possible in Hive	Record level updates, insertions and deletes, transactions and indexes are possible
OLTP (On-line Transaction Processing) is not yet supported in Hive but it's supported OLAP (On-line Analytical Processing)	Both OLTP (On-line Transaction Processing) and OLAP (On-line Analytical Processing) are supported in RDBMS

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 (also known as analytic functions) from SQL:2003.
- Some of Hive's nonstandard extensions to SQL were inspired by MapReduce, such as multitable inserts and the TRANSFORM, MAP, and REDUCE clauses.

HiveQL

Feature	SQL	HiveQL
Updates	UPDATE, INSERT, DELETE	UPDATE, INSERT, DELETE
Transactions	Supported	Limited support
Indexes	Supported	Supported
Data types	Integral, floating-point, fixed-point, text and binary strings, temporal	Boolean, integral, floating-point, fixed-point, text and binary strings, temporal, array, map, struct
Functions	Hundreds of built-in functions	Hundreds of built-in functions
Multitable inserts	Not supported	Supported
CREATE TABLE...AS SELECT	Not valid SQL-92, but found in some databases	Supported

HiveQL

Feature	SQL	HiveQL
SELECT	SQL-92	SQL-92. SORT BY for partial ordering, LIMIT to limit number of rows returned
Joins	SQL-92, or variants (join tables in the FROM clause, join condition in the WHERE clause)	Inner joins, outer joins, semi joins, map joins, cross joins
Subqueries	In any clause (correlated or noncorrelated)	In the FROM, WHERE, or HAVING clauses (uncorrelated subqueries not supported)
Views	Updatable (materialized or nonmaterialized)	Read-only (materialized views not supported)
Extension points	User-defined functions, stored procedures	User-defined functions, MapReduce scripts

Hive – 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 primitive types correspond roughly to Java's, although some names are influenced by MySQL's type names.
- 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.

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'

Hive – Data Types

Category	Type	Description	Literal examples
Complex	ARRAY	An ordered collection of fields. The fields must all be of the same type.	`array(1, 2)`[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.	`map('a', 1, 'b', 2)`
	STRUCT	A collection of named fields. The fields may be of different types.	`struct('a', 1, 1.0)`,[b] `named_struct('col1', 'a', 'col2', 1, 'col3', 1.0)`
	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.	`create_union(1, 'a', 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 both MySQL and Hive.
- Hive comes with a large number of built-in functions.
- Get 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
```

- No no built-in function, you can write your own User-Defined Functions

Hive 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.
- Hive stores the metadata in a relational 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.
- When you load data into a managed table, it is moved into Hive's warehouse directory. For example, this:

```
CREATE TABLE managed_table (dummy STRING);
LOAD DATA INPATH '/user/tom/data.txt' INTO table managed_table;
```

  will *move* the file *hdfs://user/tom/data.txt* into Hive's warehouse directory for the managed_table table, which is *hdfs://user/hive/warehouse/managed_table*

- If the table is later dropped, using:

```
DROP TABLE managed_table;
```

the table, including its metadata *and its data*, is deleted.

Managed Tables and External Tables

- 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:

```
CREATE EXTERNAL TABLE external_table (dummy STRING)
LOCATION '/user/tom/external_table';
LOAD DATA INPATH '/user/tom/data.txt' INTO TABLE external_table;
```

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

Partitions and Buckets

- Hive organizes tables into *partitions*—a way of dividing a table into coarse-grained parts based on the value of a *partition column*, such as a date.
- Using partitions can make it faster to do queries on slices of the data.
- Tables or partitions may be subdivided further into *buckets* to give extra structure to the data that may be used for more efficient queries.
- For example, bucketing by user ID means we can quickly evaluate a user-based query by running it on a randomized sample of the total set of users.

Partitions

- 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
- Notice that partitioning doesn't preclude more wide-ranging queries: it is still feasible to query the entire dataset across many partitions.
- A table may be partitioned in multiple dimensions.
- For example, in addition to partitioning logs by date, we might also *subpartition* each date partition by country to permit efficient queries by location.

Partitions

- 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:

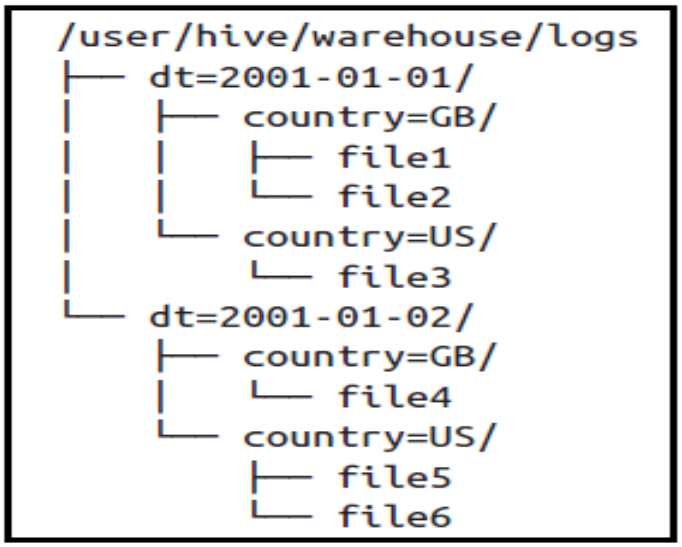
```
CREATE TABLE logs (ts BIGINT, line STRING)
PARTITIONED BY (dt STRING, country STRING);
```

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

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

Partitions

- 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:



- We can ask Hive for the partitions in a table using SHOW PARTITIONS:

```
hive> SHOW PARTITIONS logs;
dt=2001-01-01/country=GB
dt=2001-01-01/country=US
dt=2001-01-02/country=GB
dt=2001-01-02/country=US
```

Partitions

- One thing to bear in mind is that the column definitions in the PARTITIONED BY clause are full-fledged table columns, called *partition columns*; however, the datafiles do not contain values for these columns, since they are derived from the directory names.
- 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*.
- Note: query returns the values of the dt partition column, which Hive reads from the directory names since they are not in the datafiles.

Buckets

- Two reasons why you might want to organize your tables (or partitions) into buckets.
  1. The first is to enable more efficient queries.
  2. 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.
- To tell Hive that a table should be bucketed 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;
```

- Data within a bucket may additionally be sorted by one or more columns
- This allows even more efficient map-side joins.

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

Buckets

- Take an unbucketed users table:

```
hive> SELECT * FROM users;
0 Nat
2 Joe
3 Kay
4 Ann
```

- To populate bucketed table, we need to set hive.enforce.bucketing property to true.
- 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
- Running this command:

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

shows that four files were created

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

Buckets

- First bucket contains users with IDs 0 and 4

```
hive> dfs -cat /user/hive/warehouse/bucketed_users/000000_0;
0Nat
4Ann
```

- 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
```

- It's possible to sample a number of buckets by specifying a different proportion

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

Importing Data

- We've already seen how to use the LOAD DATA operation to import data into a Hive table (or partition) by copying or moving files to the table's directory.

```
CREATE TABLE managed_table (dummy STRING);
LOAD DATA INPATH '/user/tom/data.txt' INTO table managed_table;
```

- You can also populate a table with data from another Hive table using an INSERT statement, or at creation time using the *CTAS* construct (abbreviation used to refer to CREATE TABLE...AS SELECT).
- If you want to import data from a relational database directly into Hive use Sqoop
- Inserts - Here's an example of an INSERT statement:

```
INSERT OVERWRITE TABLE target
SELECT col1, col2
FROM source;
```

- For partitioned tables, you can specify the partition to insert into by supplying a PARTITION clause:

```
INSERT OVERWRITE TABLE target
PARTITION (dt='2001-01-01')
SELECT col1, col2
FROM source;
```

Importing Data

- You can specify the partition dynamically by determining the partition value from the SELECT statement:

```
INSERT OVERWRITE TABLE target
PARTITION (dt)
SELECT col1, col2, dt
FROM source;
```

- This is known as a *dynamic partition insert*.

Multitable insert

- In HiveQL, you can turn INSERT statement around and start with the FROM clause

```
FROM source
INSERT OVERWRITE TABLE target
SELECT col1, col2;
```

- Possible to have multiple INSERT clauses in the same query.
- This so-called *multitable insert* is more efficient than multiple INSERT statements because source table needs to be scanned only once to produce multiple disjoint outputs.
- Here's an example that computes various statistics over weather dataset:

```
FROM records2
INSERT OVERWRITE TABLE stations_by_year
SELECT year, COUNT(DISTINCT station)
GROUP BY year
INSERT OVERWRITE TABLE records_by_year
SELECT year, COUNT(1)
GROUP BY year
INSERT OVERWRITE TABLE good_records_by_year
SELECT year, COUNT(1)
WHERE temperature != 9999 AND quality IN (0, 1, 4, 5, 9)
GROUP BY year;
```

- There is a single source table (records2), but three tables to hold the

CREATE TABLE...AS SELECT

- Convenient to store the output of a Hive query in a new table, perhaps because it is too large to be dumped to the console or because there are further processing steps to carry out on the result.
- The new table's column definitions are derived from the columns retrieved by the SELECT clause.
- In the following query, the target table has two columns named col1 and col2 whose types are the same as the ones in the source table:

```
CREATE TABLE target
AS
SELECT col1, col2
FROM source;
```

- A CTAS operation is atomic, so if the SELECT query fails for some reason, the table is not created.

Altering Tables

- Because Hive uses the schema-on-read approach, table's definition can be to changed after the table has been created.
- It is up to develelopers to ensure that the data is changed to reflect the new structure.
- You can rename a table using the ALTER TABLE statement:

  ALTER TABLE source RENAME TO target;
- In addition to updating the table metadata, ALTER TABLE moves the underlying table directory so that it reflects the new name.
- In the current example, */user/hive/warehouse/ source* is renamed to */user/hive/warehouse/target*.
- Hive allows you to change the definition for columns, add new columns, or even replace all existing columns in a table with a new set.

  ALTER TABLE target ADD COLUMNS (col3 STRING);
- Because Hive does not permit updating existing records, you will need to arrange for the underlying files to be updated by another mechanism.
- For this reason, it is more common to create a new table that defines new columns and populates them using a SELECT statement.

Dropping Tables

- The DROP TABLE statement deletes the data and metadata for a table.
- In the case of external tables, only the metadata is deleted; the data is left untouched.
- If you want to delete all the data in a table but keep the table definition, use TRUNCATE TABLE.
- For example:

```
TRUNCATE TABLE my_table;
```

- This doesn't work for external tables; instead, use dfs -rmr (from the Hive shell) to remove the external table directory directly.
- In a similar vein, if you want to create a new, empty table with the same schema as another table, then use the LIKE keyword:

```
CREATE TABLE new_table LIKE existing_table;
```

Querying Data

- Will discusses how to use various forms of SELECT statement to retrieve data from Hive.
- Sorting data in Hive can be achieved by using a standard ORDER BY clause.
- ORDER BY performs a parallel total sort of the input
- When a globally sorted result is not required, you can use Hive's nonstandard extension, SORT BY, instead.
- SORT BY produces a sorted file per reducer.
- In some cases, you want to control which reducer a particular row goes to—typically so you can perform some subsequent aggregation.
- This is what Hive's DISTRIBUTE BY clause does.
- Here's an example to sort the weather dataset by year and temperature, in such a way as to ensure that all the rows for a given year end up in the same reducer partition:

```
hive> FROM records2
> SELECT year, temperature
> DISTRIBUTE BY year
> SORT BY year ASC, temperature DESC;
1949 111
1949 78
1950 22
1950 0
1950 -11
```

MapReduce Scripts

- Using an approach like Hadoop Streaming, the TRANSFORM, MAP, and REDUCE clauses make it possible to invoke an external script or program from Hive.
- Suppose we want to use a script to filter out rows that don't meet some condition

```
import re
import sys
for line in sys.stdin:
(year, temp, q) = line.strip().split()
if (temp != "9999" and re.match("[01459]", q)):
print "%s\t%s" % (year, temp)
```

- We can use the script as follows:

```
hive> ADD FILE /Users/tom/book-workspace/hadoop-book/ch17-hive/
src/main/python/is_good_quality.py;
hive> FROM records2
> SELECT TRANSFORM(year, temperature, quality)
> USING 'is_good_quality.py'
> AS year, temperature;
1950 0
1950 22
1950 -11
1949 111
1949 78
```

MapReduce Scripts

- Previous example has no reducers.
- If we use a nested form for the query, we can specify a map and a reduce function.
- This time we use the MAP and REDUCE keywords, but SELECT TRANSFORM in both cases would have the same result.

```
FROM (
FROM records2
MAP year, temperature, quality
USING 'is_good_quality.py'
AS year, temperature) map_output
REDUCE year, temperature
USING 'max_temperature_reduce.py'
AS year, temperature;
```

MapReduce Scripts

Reduce function for maximum temperature in Python

```
#!/usr/bin/env python
import sys
(last_key, max_val) = (None, -sys.maxint)
for line in sys.stdin:
        (key, val) = line.strip().split("\t")
        if last_key and last_key != key:
                print "%s\t%s" % (last_key, max_val)
                (last_key, max_val) = (key, int(val))
        else:
                (last_key, max_val) = (key, max(max_val, int(val)))
        if last_key:
                print "%s\t%s" % (last_key, max_val)
```

Joins

- Join operations are a case in point, given how involved they are to implement in MapReduce

Inner joins

- Simplest kind of join is inner join - each match in input tables results in a row in output.

```
hive> SELECT * FROM sales;
Joe 2
Hank 4
Ali 0
Eve 3
Hank 2
hive> SELECT * FROM things;
2 Tie
4 Coat
3 Hat
1 Scarf
```

- We can perform an inner join on the two tables as follows:

```
hive> SELECT sales.*, things.*
> FROM sales JOIN things ON (sales.id = things.id);
Joe 2 2 Tie
Hank 4 4 Coat
Eve 3 3 Hat
Hank 2 2 Tie
```

Joins

- Hive only supports equijoins, which means that only equality can be used in the join predicate, which here matches on the id column in both tables.
- In Hive, you can join on multiple columns in the join predicate by specifying a series of expressions, separated by AND keywords.
- You can also join more than two tables by supplying additional JOIN...ON... clauses in the query.
- Hive is intelligent about trying to minimize the number of MapReduce jobs to perform the joins.

Joins

- A single join is implemented as a single MapReduce job, but multiple joins can be performed in less than one MapReduce job per join if the same column is used in the join condition
- You can see how many MapReduce jobs Hive will use for any particular query by prefixing it with the EXPLAIN keyword:

```
EXPLAIN
SELECT sales.*, things.*
FROM sales JOIN things ON (sales.id = things.id);
```

- The EXPLAIN output includes many details about the execution plan for the query,
  - including the abstract syntax tree,
  - the dependency graph for the stages that Hive will execute, and
  - information about each stage.
- Stages may be MapReduce jobs or operations such as file moves.
- For even more detail, prefix the query with EXPLAIN EXTENDED.
- Hive currently uses a rule-based query optimizer for determining how to execute a query, but a cost-based optimizer is available from **Hive 0.14.0**.

Outer joins

- Outer joins allow you to find nonmatches in the tables being joined.
- If we change the join type to LEFT OUTER JOIN, the query will return a row for every row in the left table (sales), even if there is no corresponding row in the table it is being joined to (things):

```
hive> SELECT sales.*, things.*
> FROM sales LEFT OUTER JOIN things ON (sales.id = things.id);
Joe 2 2 Tie
Hank 4 4 Coat
Ali 0 NULL NULL
Eve 3 3 Hat
Hank 2 2 Tie
```

Outer joins

- Hive also supports right outer joins
- In this case, all items from the things table are included, even those that weren't purchased by anyone (a scarf):

```
hive> SELECT sales.*, things.*
> FROM sales RIGHT OUTER JOIN things ON (sales.id = things.id);
Joe 2 2 Tie
Hank 2 2 Tie
Hank 4 4 Coat
Eve 3 3 Hat
NULL NULL 1 Scarf
```

- Finally, there is a full outer join, where the output has a row for each row from both tables in the join:

```
hive> SELECT sales.*, things.*
> FROM sales FULL OUTER JOIN things ON (sales.id = things.id);
Ali 0 NULL NULL
NULL NULL 1 Scarf
Hank 2 2 Tie
Joe 2 2 Tie
Eve 3 3 Hat
Hank 4 4 Coat
```

Semi Joins

- Consider this IN subquery, which finds all the items in the things table that are in the sales table:

```
SELECT *
FROM things
WHERE things.id IN (SELECT id from sales);
```

- We can also express it as follows:

```
hive> SELECT *
> FROM things LEFT SEMI JOIN sales ON (sales.id = things.id);
2 Tie
4 Coat
3 Hat
```

- There is a restriction that we must observe for LEFT SEMI JOIN queries: the right table (sales) may appear only in the ON clause. It cannot be referenced in a SELECT expression.

Map joins

- Consider the original inner join again:

```
SELECT sales.*, things.*
FROM sales JOIN things ON (sales.id = things.id);
```

- If one table is small enough to fit in memory, as things is here, Hive can load it into memory to perform the join in each of the mappers. This is called a **map join**.
- The job to execute this query has no reducers, so this query would not work for a RIGHT or FULL OUTER JOIN, since absence of matching can be detected only in an aggregating (reduce) step across all the inputs.
- Map joins can take advantage of bucketed tables

Subqueries

- A subquery is a SELECT statement that is embedded in another SQL statement.
- Hive has limited support for subqueries, permitting a subquery in the FROM clause of a SELECT statement, or in the WHERE clause in certain cases.
- The following query finds the mean maximum temperature for every year and weather station:

```
SELECT station, year, AVG(max_temperature)
FROM (
SELECT station, year, MAX(temperature) AS max_temperature
FROM records2
WHERE temperature != 9999 AND quality IN (0, 1, 4, 5, 9)
GROUP BY station, year
) mt
GROUP BY station, year;
```

Subqueries

- **Problem :** The subquery produces a single (station, year, max_temperature) record for each (station, year) grouping ... so the outer select computes the "average" of a single temperature.

```
SELECT station, year, AVG(max_temperature)
FROM (
      SELECT station, year, MAX(temperature) AS max_temperature
      FROM records2
      WHERE temperature != 9999 AND quality IN (0, 1, 4, 5, 9)
      GROUP BY station, year  date
) mt
GROUP BY station, year;
```

Views

- A view is a sort of "virtual table" that is defined by a SELECT statement.
- Views can be used to present data to users in a way that differs from the way it is actually stored on disk.
- Often, the data from existing tables is simplified or aggregated in a particular way that makes it convenient for further processing.
- Views may also be used to restrict users' access to particular subsets of tables that they are authorized to see.
- In Hive, a view is not materialized to disk when it is created; rather, the view's SELECT statement is executed when the statement that refers to the view is run.
- If a view performs extensive transformations on the base tables or is used frequently, you may choose to manually materialize it by creating a new table that stores the contents of the view

Views

- When we create a view, query is not run; it is simply stored in metastore.
- Views are included in the output of the SHOW TABLES command
- You can see more details about a particular view, including the query used to define it, by issuing the DESCRIBE EXTENDED *view_name* command.
- Next, let's create a second view of maximum temperatures for each station and year. It is based on the valid_records view:

```
CREATE VIEW max_temperatures (station, year, max_temperature)
AS
SELECT station, year, MAX(temperature)
FROM valid_records
GROUP BY station, year;
```

- With the views in place, we can now use them by running a query:

```
SELECT station, year, AVG(max_temperature)
FROM max_temperatures
GROUP BY station, year;
```

Fundamentals of HBase

- HBase is a distributed column-oriented database built on top of HDFS.
- HBase is the Hadoop application to use when you require real-time read/write random access to very large datasets.
- Many vendors offer replication and partitioning solutions to grow the database beyond the confines of a single node, but these add-ons are generally an afterthought and are complicated to install and maintain.
- HBase approaches the scaling problem from the ground up to scale linearly just by adding nodes.
- HBase is not relational and does not support SQL, but given the proper problem space, it is able to do what an RDBMS cannot: host very large, sparsely populated tables on clusters made from commodity hardware.

HBase

- The canonical HBase use case is the *webtable*, a table of crawled web pages and their attributes (such as language and MIME type) keyed by the web page URL.
- The *webtable* is large, with row counts that run into the billions.
- Batch analytic and parsing MapReduce jobs are continuously run against the *webtable*, deriving statistics and adding new columns of verified MIME-type and parsed-text content for later indexing by a search engine.
- Concurrently, the table is randomly accessed by crawlers running at various rates and updating random rows while random web pages are served in real time as users click on a website's cached-page feature.

Hbase - Backdrop

- The HBase project was started toward the end of 2006 by Chad Walters and Jim Kellerman at Powerset.
- It was modeled after Google's Bigtable, which had just been published.
- In February 2007, Mike Cafarella made a code drop of a mostly working system that Jim Kellerman then carried forward.
- The first HBase release was bundled as part of Hadoop 0.15.0 in October 2007.
- In May 2010, HBase graduated from a Hadoop subproject to become an Apache Top Level Project.
- Today, HBase is a mature technology used in production across a wide range of industries.

When to use HBase

- You need random write, random read, or both (*but not neither*)
- You need to do many thousands of operations per second on multiple TB of data
- Your access patterns are well-known and simple

HBase vs. HDFS

- Both are distributed systems that scale to hundreds or thousands of nodes

- ***HDFS*** is good for batch processing (scans over big files)
  - Not good for record lookup
  - Not good for incremental addition of small batches
  - Not good for updates

HBase vs. HDFS

- ***HBase*** is designed to efficiently address the above points
  - Fast record lookup
  - Support for record-level insertion
  - Support for updates (not in place)

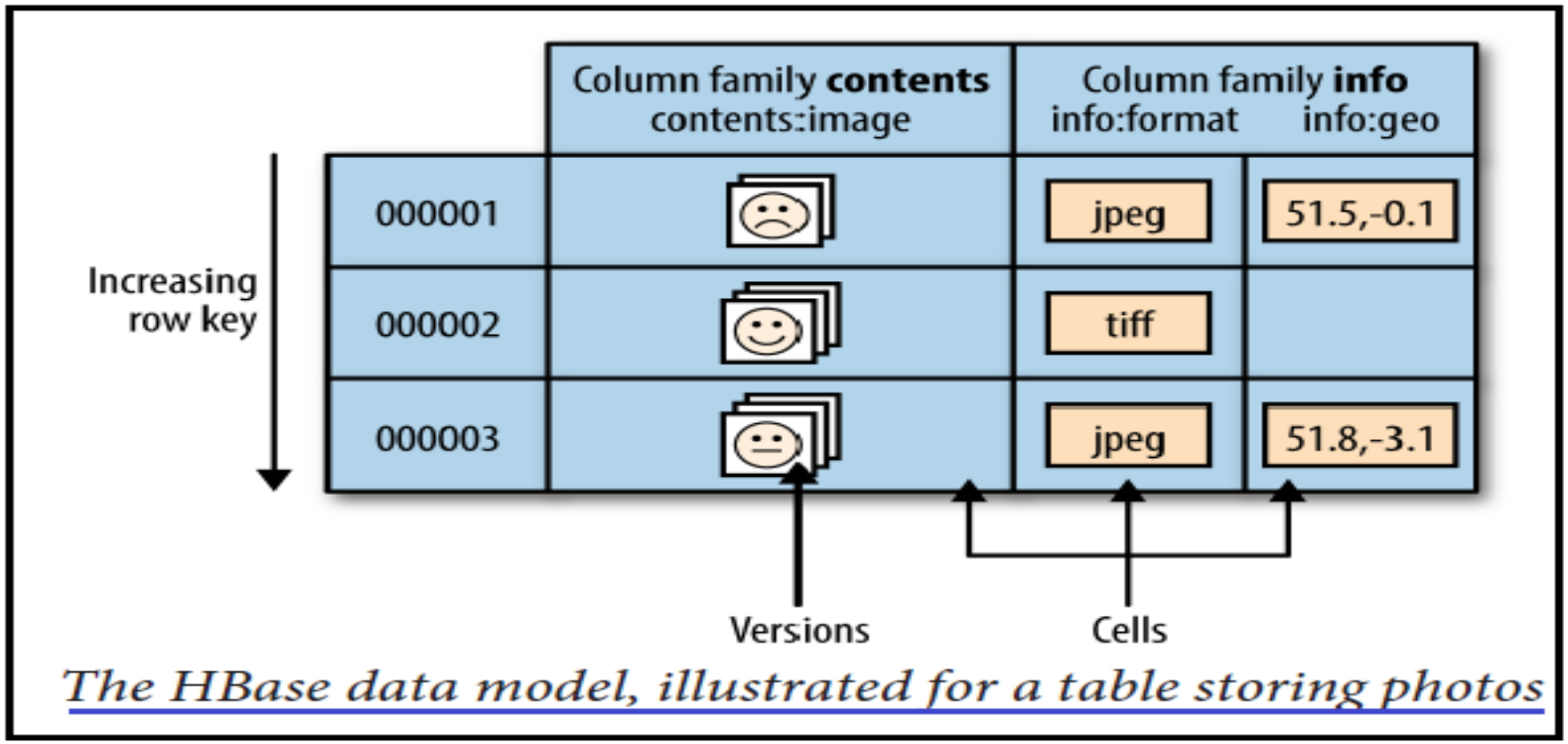
- HBase updates are done by creating new versions of values

HBase vs. HDFS

	Plain HDFS/MR	HBase
Write pattern	Append-only	Random write, bulk incremental
Read pattern	Full table scan, partition table scan	Random read, small range scan, or table scan
Hive (SQL) performance	Very good	4-5x slower
Structured storage	Do-it-yourself / TSV / SequenceFile / Avro / ?	Sparse column-family data model
Max data size	30+ PB	~1PB

Whirlwind Tour of the Data Model

- Applications store data in labeled tables.
- Tables are made of rows and columns.
- Table cells—the intersection of row and column coordinates—are versioned.
- By default, their version is a timestamp auto-assigned by HBase at the time of cell insertion.
- A cell's content is an uninterpreted array of bytes.



The HBase data model, illustrated for a table storing photos

Whirlwind Tour of the Data Model

- Table row keys are also byte arrays
- Table rows are sorted by row key, aka the table's primary key.
- The sort is byte-ordered.
- All table accesses are via the primary key.
- Row columns are grouped into *column families*.
- All column family members have a common prefix, for example, the columns info:format and info:geo are both members of the info column family, whereas contents:image belongs to the contents family.
- The column family prefix must be composed of *printable* characters.
- The qualifying tail, the column family *qualifier*, can be made of any arbitrary bytes.
- The column family and the qualifier are always separated by a colon character (:).

Whirlwind Tour of the Data Model

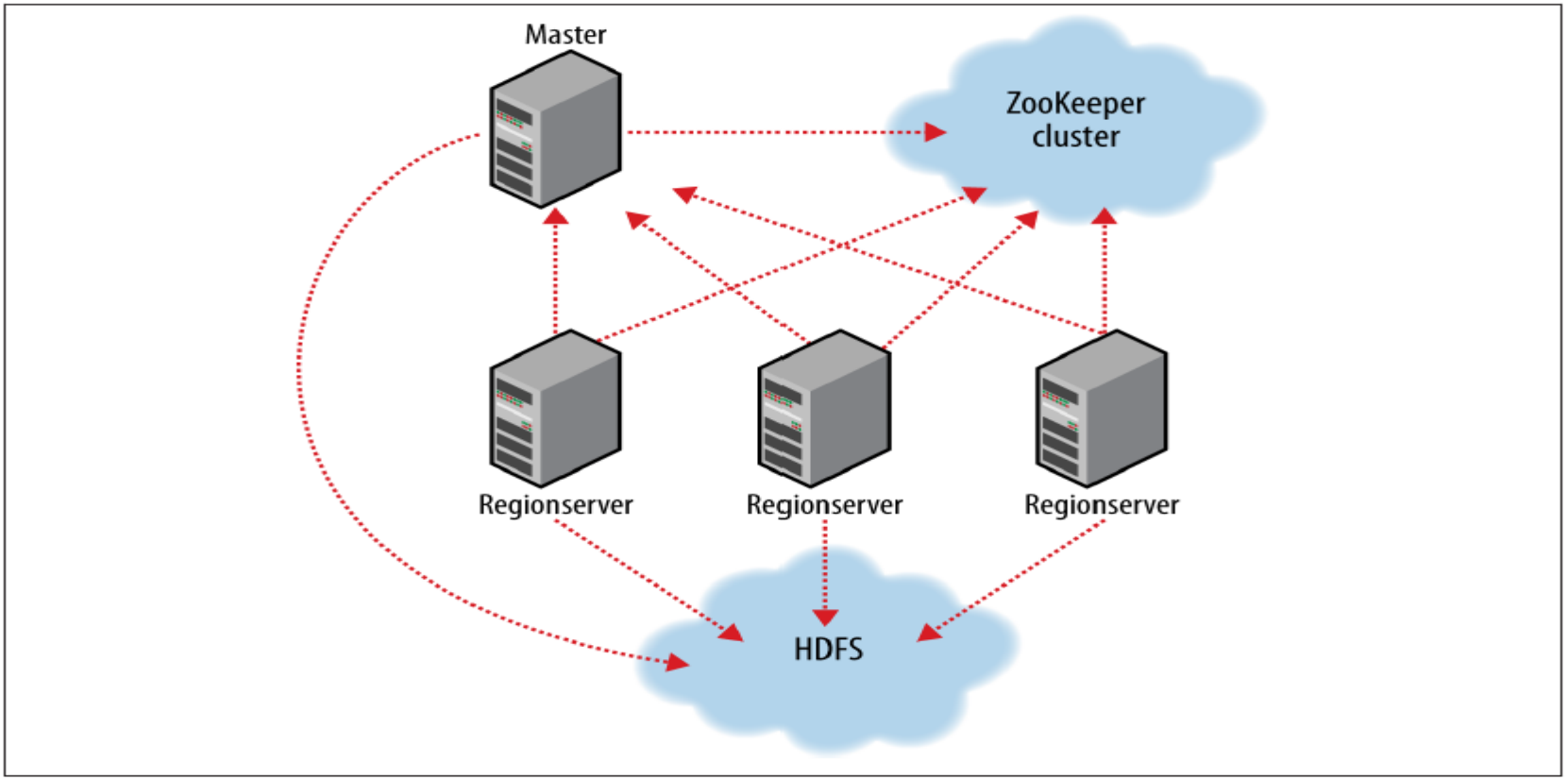
- Physically, all column family members are stored together on the filesystem.
- Described HBase as a column-oriented store, it would be more accurate if it were described as a column-*family*-oriented store.
- Because tuning and storage specifications are done at the column family level, it is advised that all column family members have the same general access pattern and size characteristics.
- For the photos table, the image data, which is large (megabytes), is stored in a separate column family from the metadata, which is much smaller in size (kilobytes).
- HBase tables are like those in an RDBMS, only cells are versioned, rows are sorted, and columns can be added on the fly.

Regions

- Tables are automatically partitioned horizontally by HBase into *regions*.
- Each region comprises a subset of a table's rows.
- A region is denoted by the table it belongs to, its first row (inclusive), and its last row (exclusive).
- Initially, a table comprises a single region, but as the region grows it eventually crosses a configurable size threshold, at which point it splits at a row boundary into two new regions of approximately equal size.
- Until this first split happens, all loading will be against the single server hosting the original region.
- As the table grows, the number of its regions grows.
- Regions are the units that get distributed over an HBase cluster.
- In this way, a table that is too big for any one server can be carried by a cluster of servers, with each node hosting a subset of the table's total regions.

Hbase Implementation

HBase made up of an HBase *master* node directing a cluster of one or more *regionserver* workers.



HBase cluster members

Hbase Implementation

- The HBase master is responsible for
  - assigning regions to registered regionservers, and
  - recovering regionserver failures.
- The master node is lightly loaded.
- The regionservers carry zero or more regions and field client read/write requests.
- They also manage region splits, informing the HBase master about the new daughter regions.
- HBase depends on ZooKeeper. The ZooKeeper ensemble hosts vitals such as the location of the hbase:meta catalog table and the address of the current cluster master.
- Regionserver worker nodes are listed in the HBase *conf/regionservers* file.
- A cluster's site-specific configuration is done in the HBase *conf/hbase-site.xml and conf/hbase-env.sh*
- HBase persists data via the Hadoop filesystem API.

HBase vs. RDBMS

	RDBMS	HBase
Data layout	Row-oriented	Column-family-oriented
Transactions	Multi-row ACID	Single row only
Query language	SQL	get/put/scan/etc *
Security	Authentication/Authorization	Work in progress
Indexes	On arbitrary columns	Row-key only
Max data size	TBs	~1PB
Read/write throughput limits	1000s queries/second	Millions of queries/second

ZooKeeper

- Apache ZooKeeper is an open-source server for highly reliable distributed coordination of cloud applications.
- It is a project of the Apache Software Foundation.
- ZooKeeper is essentially a service for distributed systems offering a hierarchical key-value store, which is used
  - to provide a distributed configuration service,
  - synchronization service, and
  - naming registry for large distributed systems
- ZooKeeper was a sub-project of Hadoop but is now a top-level Apache project in its own right.
- ZooKeeper's architecture supports high availability through redundant services.
- The clients can thus ask another ZooKeeper leader if the first fails to answer.

---- Wikipedia

ZooKeeper

> ZooKeeper allows distributed processes to coordinate with each other through a shared hierarchical name space of data registers.
>
> *- ZooKeeper Wiki*

ZooKeeper is much more than a distributed lock server!

What is ZooKeeper

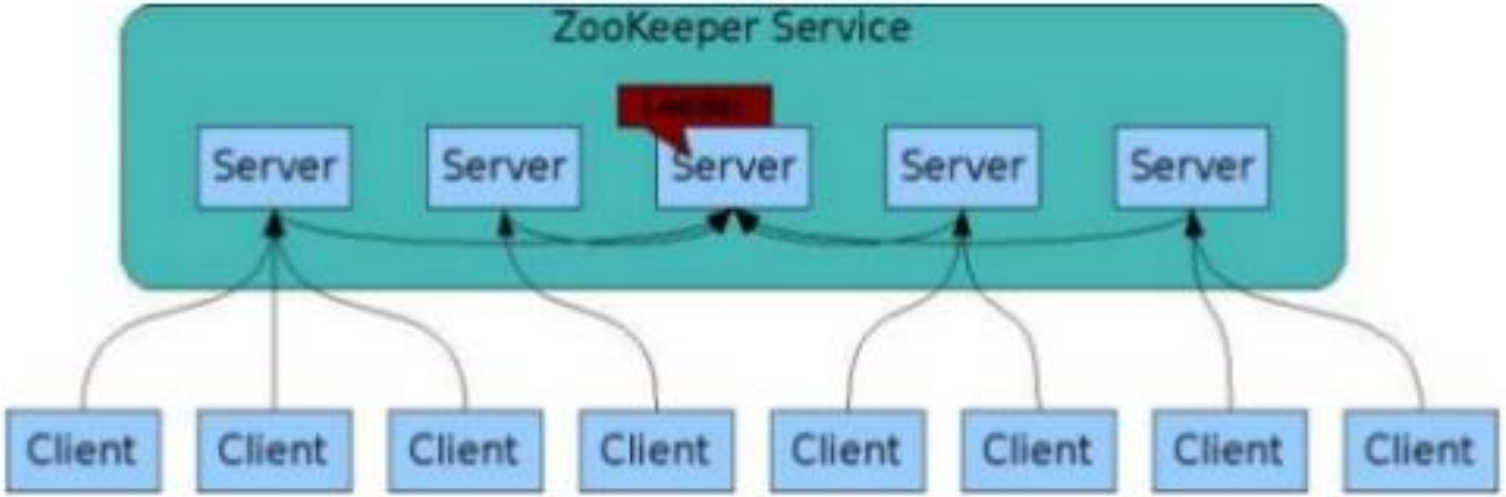
- An open source, high-performance coordination service for distributed applications.
- Exposes common services in simple interface:
  - naming
  - configuration management
  - locks & synchronization
  - group services

    ... developers don't have to write them from scratch
- Build your own on it for specific needs.

ZooKeeper Use Cases

- Configuration Management
  - Cluster member nodes bootstrapping configuration from a centralized source in unattended way
  - Easier, simpler deployment/provisioning
- Distributed Cluster Management
  - Node join / leave
  - Node statuses in real time
- Naming service – e.g. DNS
- Distributed synchronization - locks, barriers, queues
- Leader election in a distributed system.
- Centralized and highly reliable (simple) data registry

ZooKeeper Service



- ZooKeeper Service is replicated over a set of machines
- All machines store a copy of the data (in memory)
- A leader is elected on service startup
- Clients only connect to a single ZooKeeper server & maintains a TCP connection.
- Client can read from any Zookeeper server, writes go through the leader & needs majority consensus.

Image: https://cwiki.apache.org/confluence/display/ZOOKEEPER/ProjectDescription

ZooKeeper

- Writing distributed applications is hard. It's hard primarily because of partial failure.
- ZooKeeper can't make partial failures go away, since they are intrinsic to distributed systems.
- ZooKeeper also has the following characteristics:
  - *ZooKeeper is simple*
  - *ZooKeeper is expressive*
  - *ZooKeeper is highly available*
  - *ZooKeeper facilitates loosely coupled interactions*
  - *ZooKeeper is a library*
- ZooKeeper is highly performant.
- At Yahoo!, where it was created, the throughput for a ZooKeeper cluster has been benchmarked at over 10,000 operations per second for write-dominant workloads generated by hundreds of clients.
- For workloads where reads dominate, which is the norm, the throughput is several times higher

ZooKeeper commands: the four-letter words

Category	Command	Description
Server status	`ruok`	Prints `imok` if the server is running and not in an error state.
	`conf`	Prints the server configuration (from *zoo.cfg*).
	`envi`	Prints the server environment, including ZooKeeper version, Java version, and other system properties.
	`srvr`	Prints server statistics, including latency statistics, the number of znodes, and the server mode (standalone, leader, or follower).
	`stat`	Prints server statistics and connected clients.
	`srst`	Resets server statistics.
	`isro`	Shows whether the server is in read-only (`ro`) mode (due to a network partition) or read/write mode (`rw`).
Client connections	`dump`	Lists all the sessions and ephemeral znodes for the ensemble. You must connect to the leader (see `srvr`) for this command.
	`cons`	Lists connection statistics for all the server's clients.
	`crst`	Resets connection statistics.
Watches	`wchs`	Lists summary information for the server's watches.
	`wchc`	Lists all the server's watches by connection. Caution: may impact server performance for a large number of watches.
	`wchp`	Lists all the server's watches by znode path. Caution: may impact server performance for a large number of watches.
Monitoring	`mntr`	Lists server statistics in Java properties format, suitable as a source for monitoring systems such as Ganglia and Nagios.

Thank You !!!