**UNIT V HADOOP RELATED TOOLS**

**Hbase – data model and implementations – Hbase clients – Hbase examples – praxis. Pig –Grunt – pig data model – Pig Latin – developing and testing Pig Latin scripts.Hive – data types and file formats – HiveQL data definition – HiveQL data manipulation – HiveQL queries.**

**5.1 Hbase**

HBase is an open-source, distributed, non-relational, and scalable NoSQL database system built on top of Apache Hadoop. It provides real-time read and write access to large datasets, making it suitable for handling massive amounts of structured or semi-structured data. HBase is modeled after Google's Bigtable and is often used for applications that require low-latency access to vast amounts of data.

**5.1.1Key features of HBase include:**

Column-Family Based Storage: HBase organizes data into tables, which consist of rows and column families. Column families can have multiple columns, and each column can have multiple versions, which allows efficient storage and retrieval of sparse data.

* **Linear Scalability:** HBase is designed to scale horizontally across multiple nodes, making it suitable for big data scenarios. As data grows, you can add more nodes to the HBase cluster to handle the increased workload.
* **High Availability**: HBase ensures high availability by replicating data across multiple nodes. In the event of node failure, data can be retrieved from replicas, maintaining data integrity and availability.
* **Consistency:** HBase provides eventual consistency, where all read and write operations are guaranteed to eventually return the most recent data. However, it may not be immediately consistent across all nodes.
* **Fault Tolerance:** HBase handles node failures by replicating data and redistributing regions across the cluster. This fault-tolerance mechanism ensures data durability.
* **Data Model:** HBase is a column-oriented database, where each row key is associated with multiple column families, and each column family can contain multiple columns. Data in HBase is stored in a sorted order based on the row keys, allowing efficient range scans.
* **Integration with Hadoop Ecosystem:** HBase is part of the Apache Hadoop ecosystem and can work seamlessly with other components like HDFS (Hadoop Distributed File System), Hive, MapReduce, and Apache Spark.

Typical use cases for HBase include time-series data storage, sensor data storage, Internet of Things (IoT) applications, real-time analytics, and other scenarios where low-latency access to large-scale data is crucial.

HBase provides a Java API for data manipulation and can also be accessed using HBase Shell or other client libraries. The query language used in HBase is not SQL-based like traditional relational databases, but it offers filtering and scanning capabilities to retrieve data based on row keys and column values.

**5.2 HBASE DATA MODEL:**

The data model of HBase is different from traditional relational databases and is based on the principles of a column-family-based storage system. HBase organizes data into tables, which consist of rows and column families. Understanding the HBase data model is crucial for efficiently storing, accessing, and querying data. Here are the key components of the HBase data model.

**5.2.1 Table:**

An HBase database consists of one or more tables. Each table is identified by a unique name and contains rows of data. Tables in HBase are sparse, meaning they don't require a fixed schema. Different rows can have different columns, and you can add columns on the fly without affecting other rows.

* **Row Key:**

Each row in an HBase table is uniquely identified by a row key. Row keys are used to store and retrieve data and are generally sorted in lexicographic order. Efficient row key design is crucial for optimal data retrieval and performance. Row keys are typically strings or binary data.

* **Column Families:**

HBase stores data in column families, which are groups of related columns. Each table can have one or more column families. Column families must be defined when creating a table, and once defined, the number of column families cannot be changed. All rows in an HBase table share the same set of column families, though not necessarily the same columns.

* **Columns:**

Columns within a column family are identified by unique names. Unlike column families, columns can be added or removed dynamically for each row without affecting other rows. Columns are addressed using their column family and column qualifier (name).

* **Versions:**

HBase allows the storage of multiple versions of a cell (value) for a given row, column family, and column qualifier. Each version of a cell is timestamped, allowing data to be versioned and historically tracked. By default, HBase retains only the most recent version, but you can configure the number of versions to keep.

* **Cells:**

Cells are the basic unit of data storage in HBase. A cell consists of a combination of row key, column family, column qualifier, timestamp, and value. The row key, column family, and column qualifier together are called the "cell address" or "cell key."

* **Regions:**

To enable scalability and distribution, HBase divides a table into regions. Each region is a subset of the table's data, and each region is stored on a separate region server. As data grows, HBase dynamically splits regions to distribute the data evenly across the cluster.

The HBase data model, with its column-family-based design and distributed architecture, allows for scalable and efficient storage and retrieval of vast amounts of data. When designing an HBase data model, careful consideration of row key design, column family layout, and access patterns is essential to achieve optimal performance and scalability for specific use cases.

**5.3 Hbase implementation:**

Implementing HBase involves setting up a distributed HBase cluster, designing the data model, and interacting with the database using appropriate APIs or client libraries. Below are the general steps to implement HBase:

* **Set Up a Hadoop Cluster:**

HBase is built on top of Apache Hadoop, so you need to have a working Hadoop cluster before setting up HBase. Install Hadoop on each node of the cluster and ensure that the HDFS (Hadoop Distributed File System) is properly configured and running.

* **Install HBase:**

Download the latest version of HBase from the Apache HBase website. Extract the HBase package on each node of the Hadoop cluster.

* **Configure HBase:**

HBase comes with various configuration files, such as hbase-site.xml, hbase-env.sh, and hbase-default.xml. Customize these files based on your cluster requirements, such as specifying the ZooKeeper quorum, HDFS data directory, and other HBase settings.

* **Start HBase Services:**

Start the HBase services on each node of the cluster. HBase has several daemons, including the HMaster, RegionServers, and ZooKeeper, which work together to manage the data storage and distribution.

* **Design the Data Model:**

Design the HBase data model based on the requirements of your application. Determine the tables, row keys, column families, and columns that will be used to store the data. Careful consideration of data access patterns and performance requirements is crucial in this step.

* **Create HBase Tables:**

Using the HBase shell or HBase APIs, create the tables with the defined data model. Specify the column families and other table properties during table creation.

* **Interact with HBase:**

To interact with HBase, you can use the HBase shell for simple operations or use programming languages like Java, Python, or other supported languages to connect to HBase using the appropriate client libraries (e.g., HBase Java API). Through the client libraries, you can perform CRUD (Create, Read, Update, Delete) operations, scan data, and interact with HBase tables programmatically.

* **Monitor and Maintain the Cluster:**

Regularly monitor the health and performance of the HBase cluster using various monitoring tools provided with HBase. Keep an eye on cluster metrics, node status, and data distribution to ensure smooth operation. Regularly maintain the cluster by performing tasks like region splitting and compacting to optimize data storage.

* **Backup and Disaster Recovery:**

Implement a backup and disaster recovery strategy to ensure data safety in case of node failures or other critical issues. Consider using Hadoop's HDFS snapshot feature or external backup solutions for HBase data.

It's important to note that implementing HBase can be complex, especially in large-scale production environments. It's advisable to refer to the official Apache HBase documentation and seek expert guidance when deploying HBase in a production environment.

**5.3 Hbase clients**

HBase provides several client libraries and interfaces that allow applications to interact with the HBase database. These clients enable developers to perform CRUD (Create, Read, Update, Delete) operations, scanning, and other data manipulation tasks. Here are some of the common HBase clients:

* **HBase Java API:**

The HBase Java API is one of the primary and most commonly used client libraries for HBase. It provides a comprehensive set of classes and methods to interact with HBase programmatically using the Java programming language. The Java API offers features like table creation, data insertion, data retrieval, filtering, and administrative operations.

* **HBase Shell:**

HBase Shell is a command-line interface that comes bundled with HBase. It allows users to interact with HBase using simple commands. The shell provides basic CRUD operations, scanning, and table administration commands. It's useful for quick testing and prototyping.

* **HBase REST API:**

HBase also provides a RESTful web service interface, known as the HBase REST API. This allows applications to interact with HBase using HTTP methods (GET, PUT, POST, DELETE) and JSON or XML payloads. The REST API is suitable for web and mobile applications that need to access HBase data over the web.

* **HBase Thrift API:**

The HBase Thrift API is a cross-language interface that enables applications to access HBase using Thrift, which is a software framework for scalable cross-language services development. Thrift allows clients in different programming languages (e.g., Java, Python, Ruby, C++, etc.) to communicate with HBase using a common interface.

* **HBase Async API:**

The HBase Async API is an asynchronous Java client library that provides non-blocking access to HBase. It allows developers to perform operations concurrently, which can be beneficial for applications that require high-performance, asynchronous data access.

* **HBase MapReduce Integration:**

HBase integrates with Apache Hadoop's MapReduce framework, allowing MapReduce jobs to read data from HBase tables and write results back to HBase. This integration is particularly useful for large-scale data processing tasks that require data residing in HBase.

* **HBase Spark Integration:**

Similar to HBase's integration with MapReduce, HBase can also be integrated with Apache Spark. This allows Spark applications to read and write data from HBase directly, facilitating real-time data processing and analytics.

When selecting an HBase client, consider the programming language and the specific requirements of your application. For Java-based applications, the HBase Java API is the most popular choice. For web applications, the HBase REST API might be more suitable. Thrift API and other language-specific clients are helpful when working with languages other than Java.

**5.4HBASE EXAMPLES:**

Here are some examples of how to use HBase with the HBase Java API:

**5.4.1 Initializing HBase Configuration:**

Before using the HBase Java API, you need to initialize the HBase configuration and create an HBase connection.

import org.apache.hadoop.hbase.HBaseConfiguration;

import org.apache.hadoop.hbase.client.Connection;

import org.apache.hadoop.hbase.client.ConnectionFactory;

public class HBaseExample {

public static void main(String[] args) {

try {

// Initialize HBase configuration

org.apache.hadoop.conf.Configuration config = HBaseConfiguration.create();

// Create HBase connection

Connection connection = ConnectionFactory.createConnection(config);

// Use the connection for HBase operations

// Don't forget to close the connection when done

connection.close();

} catch (Exception e) {

e.printStackTrace();

}

}

}

**5.4.2 CREATING A TABLE AND ADDING DATA:**

Here's an example of how to create an HBase table, add data to it, and retrieve data from the table.

import org.apache.hadoop.hbase.TableName;

import org.apache.hadoop.hbase.client.Admin;

import org.apache.hadoop.hbase.client.Put;

import org.apache.hadoop.hbase.client.Table;

public class HBaseExample {

public static void main(String[] args) {

try {

// Initialize HBase configuration and create a connection (as shown in the previous example)

// Create an HBase table

TableName tableName = TableName.valueOf("my\_table");

Table table = connection.getTable(tableName);

// Add data to the table

Put put1 = new Put("row1".getBytes());

put1.addColumn("cf1".getBytes(), "col1".getBytes(), "value1".getBytes());

table.put(put1);

Put put2 = new Put("row2".getBytes());

put2.addColumn("cf1".getBytes(), "col1".getBytes(), "value2".getBytes());

table.put(put2);

// Retrieve data from the table

Get get = new Get("row1".getBytes());

Result result = table.get(get);

byte[] value = result.getValue("cf1".getBytes(), "col1".getBytes());

System.out.println("Value for row1: " + new String(value));

// Don't forget to close the table when done

table.close();

connection.close();

} catch (Exception e) {

e.printStackTrace();

}

}

}

**5.4.3 Scanning Data:**

You can use the HBase Scan class to perform a range scan on the table.

import org.apache.hadoop.hbase.client.ResultScanner;

import org.apache.hadoop.hbase.client.Scan;

import org.apache.hadoop.hbase.util.Bytes;

public class HBaseExample {

public static void main(String[] args) {

try {

// Initialize HBase configuration and create a connection (as shown in the first example)

// Create an HBase table

TableName tableName = TableName.valueOf("my\_table");

Table table = connection.getTable(tableName);

// Define the scan range

Scan scan = new Scan();

scan.withStartRow(Bytes.toBytes("row1"));

scan.withStopRow(Bytes.toBytes("row3"));

// Retrieve data using the scan

ResultScanner scanner = table.getScanner(scan);

for (Result result : scanner) {

byte[] value = result.getValue(Bytes.toBytes("cf1"), Bytes.toBytes("col1"));

System.out.println("Value: " + new String(value));

}

// Don't forget to close the scanner and table when done

scanner.close();

table.close();

connection.close();

} catch (Exception e) {

e.printStackTrace();

}

}

}

These are some basic examples of how to interact with HBase using the HBase Java API.

**5.5 PRAXIS:**

"praxis" refers to applying the theoretical understanding of HBase's data model, architecture, and features to real-world scenarios and use cases. It involves practical implementation and utilization of HBase in various applications, enabling developers and data engineers to leverage its capabilities effectively.

Here are some examples of praxis in HBase:

* **Data Modeling:** Designing the HBase data model based on the specific requirements of the application is a crucial aspect of praxis. This involves determining the row key design, column families, and columns based on the access patterns and query requirements. Praxis in data modeling ensures efficient data storage and retrieval.
* **Table Creation and Management:** Practicing the creation and management of HBase tables involves defining schema, column families, and other table properties using the HBase Java API or HBase shell. This praxis ensures that tables are created optimally to suit the application's needs.
* **Data Ingestion:** Implementing praxis in HBase data ingestion involves loading data from various sources into HBase tables. It may include batch data loading using tools like Apache HBase Bulk Load or real-time data ingestion using frameworks like Apache Kafka and Apache HBase Kafka Connector.
* **Data Retrieval:** Utilizing HBase APIs to perform CRUD operations and retrieve data based on row keys, column families, and column qualifiers is a practical application of praxis. This ensures that data is retrieved efficiently for specific application use cases.
* **Secondary Indexing:** Praxis in secondary indexing involves setting up secondary indexes on HBase tables to facilitate efficient querying and searching based on non-row-key attributes. This can be accomplished using techniques like HBase Coprocessors or integrating with external indexing systems.
* **Data Versioning:** Understanding and implementing data versioning in HBase is a praxis that enables applications to maintain historical data and track changes over time. It involves using timestamps for cells and efficiently managing data versions.
* **Bulk and Incremental Processing:** Leveraging HBase's integration with Apache Hadoop and Apache Spark for bulk and incremental data processing is a praxis to achieve efficient analytics and data transformations.
* **Fault Tolerance and Replication:** Implementing praxis in HBase fault tolerance involves setting up data replication across HBase regions and nodes, ensuring data availability and durability in case of node failures.

Overall, praxis in HBase involves hands-on experience in designing data models, creating tables, loading data, querying, and understanding the performance implications of various HBase operations. It enables practitioners to effectively use HBase in real-world applications and leverage its strengths in managing large-scale, distributed data

**5.6 PIG**

Pig provides an engine for executing data flows in parallel on Hadoop. It includes a language, Pig Latin, for expressing these data flows. Pig Latin includes operators for many of the traditional data operations (join, sort, filter, etc.), as well as the ability for users to develop their own functions for reading, processing, and writing data.

Pig is an Apache open source project. This means users are free to download it as source or binary, use it for themselves, contribute to it, and—under the terms of the ApacheLicense—use it in their products and change it as they see fit.

**5.6.1Pig on Hadoop**

Pig runs on Hadoop. It makes use of both the Hadoop Distributed File System, HDFS, and Hadoop’s processing system, MapReduce. HDFS is a distributed filesystem that stores files across all of the nodes in a Hadoop cluster. It handles breaking the files into large blocks and distributing them across different machines, including making multiple copies of each block so that if any one machine fails no data is lost. By default, Pig reads input files from HDFS, uses HDFS to store intermediate data between MapReduce jobs, and writes its output to HDFS.

MapReduce is a simple but powerful parallel data-processing paradigm. Every job in

MapReduce consists of three main phases: map, shuffle, and reduce. In the map phase, the application has the opportunity to operate on each record in the input separately. In the shuffle phase, which happens after the map phase, data is collected together by the key the user has chosen and distributed to different machines for the reduce phase. Every record for a given key will go to the same reducer. In the reduce phase, the application is presented each key, together with all of the records containing that key. Again this is done in parallel on many machines. After processing each group, the reducer can write its output.

**5.6.2 MapReduce’s hello world**

Consider a simple MapReduce application that counts the number of times each word

appears in a given text. This is the “hello world” program of MapReduce. In this example the map phase will read each line in the text, one at a time. It will then split out each word into a separate string, and, for each word, it will output the word and a 1 to indicate it has seen the word one time. The shuffle phase will use the word as the key, hashing the records to reducers. The reduce phase will then sum up the number of times each word was seen and write that together with the word as output. Let’s consider the case of the nursery rhyme “Mary Had a Little Lamb.” Our input will be:

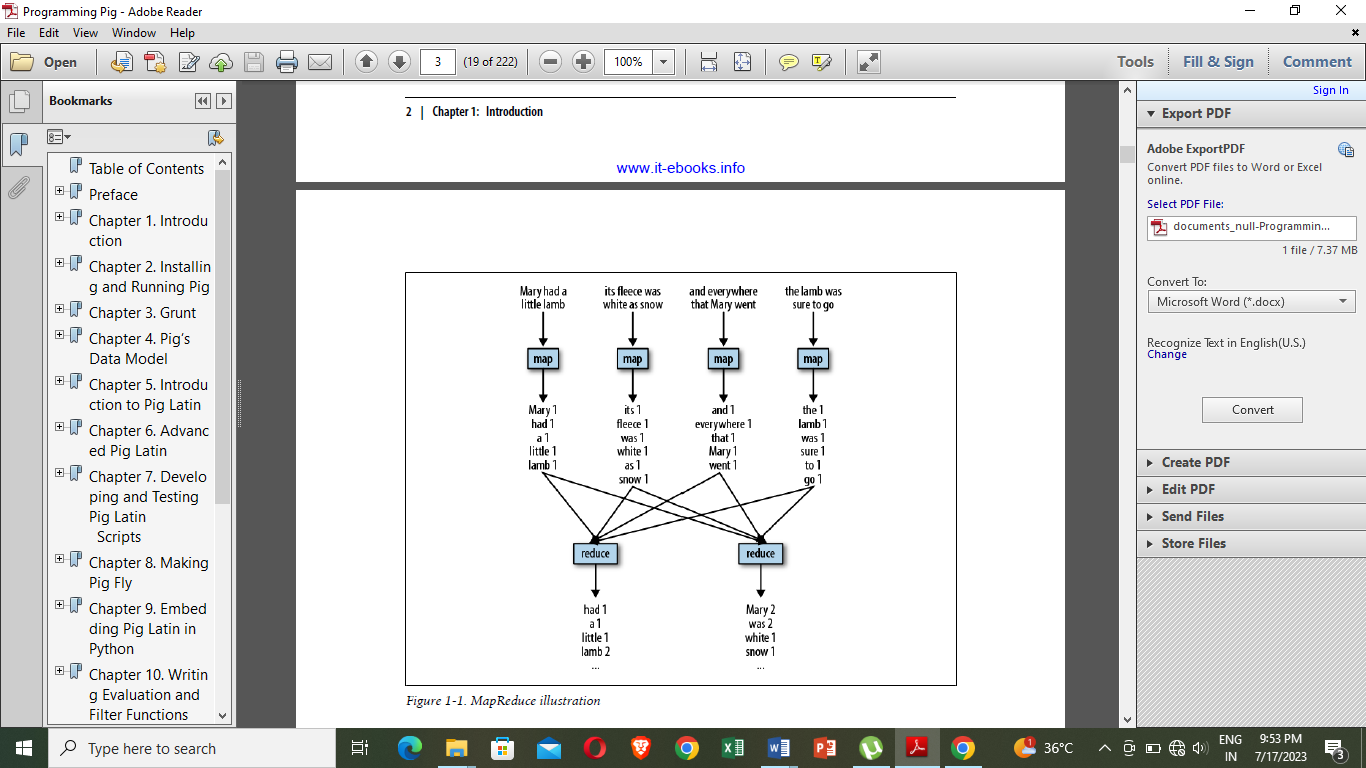
**Mary had a little lamb**

**its fleece was white as snow**

**and everywhere that Mary went**

**the lamb was sure to go.**

Pig uses MapReduce to execute all of its data processing. It compiles the Pig Latin scripts that users write into a series of one or more MapReduce jobs that it then executes. Pig Latin script that will do a word count of “Mary Had a Little Lamb.”



**Fig5.6.1 reduce map**

**Example 5.6-1. Pig counts Mary and her lamb**

-- Load input from the file named Mary, and call the single

-- field in the record 'line'.

input = load 'mary' as (line);

-- TOKENIZE splits the line into a field for each word.

-- flatten will take the collection of records returned by

-- TOKENIZE and produce a separate record for each one, calling the single

-- field in the record word.

words = foreach input generate flatten(TOKENIZE(line)) as word;

-- Now group them together by each word.

grpd = group words by word;

-- Count them.

cntd = foreach grpd generate group, COUNT(words);

-- Print out the results.

dump cntd;

**5.6.3 Pig Latin, a Parallel Dataflow Language**

Pig Latin is a dataflow language. This means it allows users to describe how data from

one or more inputs should be read, processed, and then stored to one or more outputs in parallel. These data flows can be simple linear flows like the word count example given previously. They can also be complex workflows that include points where multiple inputs are joined, and where data is split into multiple streams to be processed by different operators. To be mathematically precise, a Pig Latin script describes a directed acyclic graph (DAG), where the edges are data flows and the nodes are operators that process the data. Pig Latin looks different from many of the programming languages. Thereare no if statements or for loops in Pig Latin. This is because traditional procedural and object-oriented programming languages describe control flow, and data flow is a side effect of the program. Pig Latin instead focuses on data flow. For information on how to integrate the data flow described by a Pig Latin script with control flow.

**5.6.4 Comparing query and dataflow languages**

Pig Latin is a procedural version of SQL. SQL is a query language. Its focus is to allow users to form queries. It allows users to describe what question they want answered, but not how they want it answered. In Pig Latin, on the other hand, the user describes exactly how to process the input data.

Another major difference is that SQL is oriented around answering one question. When users want to do several data operations together, they must either write separate queries, storing the intermediate data into temporary tables, or write it in one query using subqueries inside that query to do the earlier steps of the processing.

Pig, however, is designed with a long series of data operations in mind, so there is no

need to write the data pipeline in an inverted set of subqueries or to worry about storing data in temporary tables.

Consider a case where a user wants to group one table on a key and then join it with a second table. Because joins happen before grouping in a SQL query, this must be expressed

either as a subquery or as two queries with the results stored in a temporary table.

**Example 5.6-2. Group then join in SQL**

CREATE TEMP TABLE t1 AS

SELECT customer, sum(purchase) AS total\_purchases

FROM transactions

GROUP BY customer;

SELECT customer, total\_purchases, zipcode

FROM t1, customer\_profile

WHERE t1.customer = customer\_profile.customer;

In Pig Latin, on the other hand, this looks like

**Example 5.6-3. Group then join in Pig Latin**

-- Load the transactions file, group it by customer, and sum their total purchases

txns = load 'transactions' as (customer, purchase);

grouped = group txns by customer;

total = foreach grouped generate group, SUM(txns.purchase) as tp;

-- Load the customer\_profile file

profile = load 'customer\_profile' as (customer, zipcode);

-- join the grouped and summed transactions and customer\_profile data

answer = join total by group, profile by customer;

-- Write the results to the screen dump answer;

**5.6.5 How Pig differs from MapReduce**

Pig provides users with several advantages over using MapReduce directly. Pig Latin

provides all of the standard data-processing operations, such as join, filter, group by, order by, union, etc. MapReduce provides the group by operation directly (that is what the shuffle plus reduce phases are), and it provides the order by operation indirectly through the way it implements the grouping. Filter and projection can be implemented trivially in the map phase. But other operators, particularly join, are not provided and must instead be written by the user.

Pig provides some complex, nontrivial implementations of these standard data operations.For example, because the number of records per key in a dataset is rarely evenly

distributed, the data sent to the reducers is often skewed. Pig has join and order by operators that will handle this case and (in some cases) rebalance the reducers.

In MapReduce, the data processing inside the map and reduce phases is opaque to the

system. This means that MapReduce has no opportunity to optimize or check the user’s code. Pig, on the other hand, can analyze a Pig Latin script and understand the data flow that the user is describing.

MapReduce does not have a type system. This is intentional, and it gives users the flexibility to use their own data types and serialization frameworks. But the downside is that this further limits the system’s ability to check users’ code for errors both before and during runtime.

All of these points mean that Pig Latin is much lower cost to write and maintain than

Java code for MapReduce. In one very unscientific experiment, I wrote the same operation in Pig Latin and MapReduce. Given one file with user data and one with click data for a website, the Pig Latin script in example will find the five pages most visited by users between the ages of 18 and 25.

**Example 5.6.4. Finding the top five URLs**

Users = load 'users' as (name, age);

Fltrd = filter Users by age >= 18 and age <= 25;

Pages = load 'pages' as (user, url);

Jnd = join Fltrd by name, Pages by user;

Grpd = group Jnd by url;

Smmd = foreach Grpd generate group, COUNT(Jnd) as clicks;

Srtd = order Smmd by clicks desc;

Top5 = limit Srtd 5;

store Top5 into 'top5sites';

The first line of this program loads the file users and declares that this data has two fields: name and age. It assigns the name of Users to the input. The second line applies a filter to Users that passes through records with an age between 18 and 25, inclusive. All other records are discarded. Now the data has only records of users in the age range `we are interested in. The results of this filter are named Fltrd.

The second load statement loads pages and names it Pages. It declares its schema to

have two fields, user and url. The line Jnd = join joins together Fltrd and Pages using Fltrd.name and Pages.user as the key. After this join we have found all the URLs each user has visited. The line Grpd = group collects records together by URL. So for each value of url, such as pignews.com/frontpage, there will be one record with a collection of all records that

have that value in the url field. The next line then counts how many records are collected together for each URL. So after this line we now know, for each URL, how many times it was visited by users aged 18–25.

The next thing to do is to sort this from most visits to least. The line Srtd = order sorts on the count value from the previous line and places it in desc (descending) order. Thus the largest value will be first. Finally, we need only the top five pages, so the last line limits the sorted results to only five records. The results of this are then stored back to HDFS in the file top5sites.

**5.7 GRUNT**

Grunt is Pig’s interactive shell. It enables users to enter Pig Latin interactively and provides a shell for users to interact with HDFS.

To enter Grunt, invoke Pig with no script or command to run. Typing:

**pig -x local**

will result in the prompt:

**grunt>**

If you omit the **-x**

local and have a cluster configuration set in PIG\_CLASSPATH, this will put you in a Grunt shell that will interact with HDFS on your cluster. Grunt provides command-line history and editing, as well as Tab completion. It does not provide filename completion via the Tab key.

That is, if you type kil and then press the Tab key, it will complete the command as kill. But if you have a file foo in your local directory and type ls fo, and then hit Tab, it will not complete it as **ls foo**.

To exit Grunt you can type quit or enter **Ctrl-D**.

**5.7.1 Entering Pig Latin Scripts in Grunt**

One of the main uses of Grunt is to enter Pig Latin in an interactive session. You can enter Pig Latin directly into Grunt. Pig will not start executing the Pig Latin you enter until it sees either a store or dump. However, it will do basic syntax and semantic checking to help you catch errors quickly. If you do make a mistake while entering a line of Pig Latin in Grunt, you can reenter the line using the same alias, and Pig will take the last instance of the line you enter. For example:

**pig -x local**

**grunt> dividends = load 'NYSE\_dividends' as (exchange, symbol, date, dividend);**

**grunt> symbols = foreach dividends generate symbl;**

**...Error during parsing. Invalid alias: symbl ...**

**grunt> symbols = foreach A generate symbol;**

**...**

**5.7.2 HDFS Commands in Grunt**

Grunt’s other major use is to act as a shell for HDFS. In versions 0.5 and later of Pig, all hadoop fs shell commands are available. They are accessed using the keyword fs. The dash (-) used in the hadoop fs is also required:

**grunt>fs –ls**

A number of the commands come directly from Unix shells and will operate in ways that are familiar: chgrp, chmod, chown, cp, du, ls, mkdir, mv, rm, and stat. A few of them either look like Unix commands you are used to but behave slightly differently or are unfamiliar, including:

**cat filename**

Print the contents of a file to stdout. You can apply this command to a directory and it will apply itself in turn to each file in the directory.

**copyFromLocal localfile hdfsfile**

Copy a file from your local disk to HDFS. This is done serially, not in parallel.

**copyToLocal hdfsfile localfile**

Copy a file from HDFS to your local disk. This is done serially, not in parallel.

**rmr filename**

Remove files recursively. This is equivalent to rm -r in Unix. Use this with caution.In versions of Pig before 0.5, hadoop fs commands were not available. Instead, Grunt had its own implementation of some of these commands: cat, cd, copyFromLocal, copy ToLocal, cp, ls, mkdir, mv, pwd, rm (which acted like Hadoop’s rmr, not Hadoop’s rm), and rmf. As of Pig 0.8, all of these commands are still available. However, with the exception of cd and pwd, these commands are deprecated in favor of using hadoop fs, and they might be removed at some point in the future. In version 0.8, a new command was added to Grunt: sh. This command gives you access to the local shell, just as fs gives you access to HDFS.

**5.7.3 Controlling Pig from Grunt**

Grunt also provides commands for controlling Pig and MapReduce:

1. kill jobid
2. exec
3. run

**1. kill jobid:**

Kill the MapReduce job associated with jobid. The output of the pig command that spawned the job will list the ID of each job it spawns. You can also find the job’s ID by looking at Hadoop’s JobTracker GUI, which lists all jobs currently running on the cluster. If your Pig job contains other MapReduce jobs that do not depend on the killed MapReduce job, these jobs will still continue. If you want to kill all of the Map-Reduce jobs associated with a particular Pig job, it is best to terminate the process running Pig, and then use this command to kill any MapReduce jobs that are still running. Make sure to terminate the Pig process with a Ctrl-C or a Unix kill, not a Unix kill -9.

**2. exec [[-param param\_name = param\_value]] [[-param\_file filename]] script**

Execute the Pig Latin script script. Aliases defined in script are not imported into Grunt. This command is useful for testing your Pig Latin scripts while inside a Grunt session.

**3. run [[-param param\_name = param\_value]] [[-param\_file filename]] script**

Execute the Pig Latin script script in the current Grunt shell. Thus all aliases referenced in script are available to Grunt, and the commands in script are accessible via the shell history.

**5.8. Pig’s Data Model**

**5.8.1Pig data types**

Pig’s data types can be divided into two categories: scalar types and complex types.

**5.8.1.1.Scalar Types**

Pig’s scalar types are simple types that appear in most programming languages. With

the exception of bytearray, they are all represented in Pig interfaces by java.lang classes, making them easy to work with in UDFs:

1.int

2.long

3.float

4.double

5.chararray

6.bytearray

**1.int:**

An integer. Ints are represented in interfaces by java.lang.Integer. They store a four byte signed integer. Constant integers are expressed as integer numbers, for example,42.

**2. long**

A long integer. Longs are represented in interfaces by java.lang.Long. They store an eight-byte signed integer. Constant longs are expressed as integer numbers with an L appended, for example, 5000000000L.

**3.float**

A floating-point number. Floats are represented in interfaces by java.lang.Float and use four bytes to store their value. Constant floats are expressed as a floating-point number with an f appended. Floating-point numbers can be expressed in simple format, 3.14f, or in exponent format, 6.022e23f.

**4. double**

A double-precision floating-point number. Doubles are represented in interfaces by java.lang.Double and use eight bytes to store their value. Constant doubles are expressed as a floating-point number in either simple format, 2.71828, or in exponent format, 6.626e-34.

**5. chararray**

A string or character array. Chararrays are represented in interfaces by java.lang.String. Constant chararrays are expressed as string literals with single quotes, for example, 'fred'. In addition to standard alphanumeric and symbolic characters,we can express certain characters in chararrays by using backslash codes, such as \t for Tab and \n for Return. Unicode characters can be expressed as \u followed by their four-digit hexadecimal Unicode value. For example, the value for Ctrl-A is expressed as \u0001.

**6. bytearray**

A blob or array of bytes. Bytearrays are represented in interfaces by a Java class DataByteArray that wraps a Java byte[]. There is no way to specify a constant bytearray.

**5.8.1.2Complex Types**

Pig has three complex data types: maps, tuples, and bags

**1. Maps**

A map in Pig is a chararray to data element mapping, where that element can be any Pig type, including a complex type. The chararray is called a key and is used as an index to find the element, referred to as the value. Because Pig does not know the type of the value, it will assume it is a bytearray. If the value is of a type other than bytearray, Pig will figure that out at runtime and handle it. Map constants are formed using brackets to delimit the map, a hash between keys and values, and a comma between key-value pairs. For example, ['name'#'bob','age'#55] will create a map with two keys, “name” and “age”. The first value is a chararray, and the second is an integer.

**2.Tuple:**

A tuple is a fixed-length, ordered collection of Pig data elements. Tuples are divided into fields, with each field containing one data element. These elements can be of any type—they do not all need to be the same type. A tuple is analogous to a row in SQL, with the fields being SQL columns. Because tuples are ordered, it is possible to refer to the fields by position; Tuple constants use parentheses to indicate the tuple and commas to delimit fields in the tuple. For example, ('bob', 55) describes a tuple constant with two fields.

**3. Bag:**

A bag is an unordered collection of tuples. Because it has no order, it is not possible to reference tuples in a bag by position. Like tuples, a bag can, but is not required to, have a schema associated with it. In the case of a bag, the schema describes all tuples within the bag.

Bag constants are constructed using braces, with tuples in the bag separated by commas. For example, **{('bob', 55), ('sally', 52), ('john', 25)}** constructs a bag with three tuples, each with two fields. It is possible to mimic a set type using the bag, by wrapping the desired type

in a tuple of one field. bags are used to store collections when grouping, bags can become quite large. Pig has the ability to spill bags to disk when necessary, keeping only partial sections of the bag in memory. The size of the bag is limited to the amount of local disk available for spilling the bag.

**5.8.2 Nulls**

Pig includes the concept of a data element being null. Data of any type can be null. In Pig a null data element means the value is unknown. This might be because the data is missing, an error occurred in processing it, etc. In most procedural languages, a data value is said to be null when it is unset or does not point to a valid address or object. This difference in the concept of null is important and affects the way Pig treats null data, especially when operating on it.

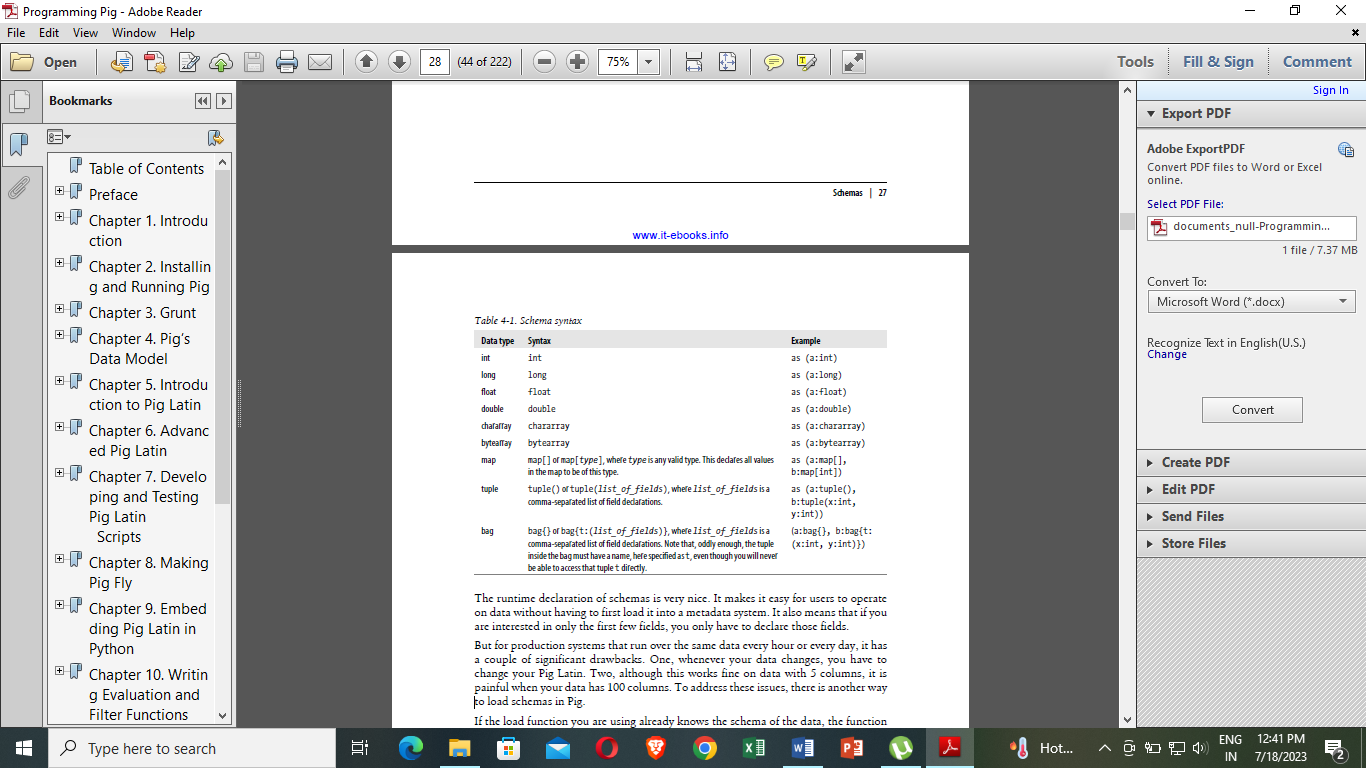
**5.8.3 schema**

Pig has a very lax attitude when it comes to schemas. If a schema for the data is available, Pig will make use of it, both for up-front error checking and for optimization. But if no schema is available, Pig will still process the data, making the best guesses it can based on how the script treats the data. The easiest way to communicate the schema of your data to Pig is to explicitly tell Pig what it is when you load the data:

**dividends = load 'NYSE\_dividends' as**

**(exchange:chararray, symbol:chararray, date:chararray, dividend:float);**

Pig now expects your data to have four fields. If it has more, it will truncate the extra ones. If it has less, it will pad the end of the record with nulls. It is also possible to specify the schema without giving explicit data types. In this case, the data type is assumed to be bytearray: dividends = load 'NYSE\_dividends' as (exchange, symbol, date, dividend);



**5.8.1 Schema syntax**

when you declare a schema, you do not have to declare the schema of complex types, but you can if you want to. For example, if your data has a tuple in it, you can declare that field to be a tuple without specifying the fields it contains. You can also declare that field to be a tuple that has three columns, all of which are integers. The runtime declaration of schemas is very nice. It makes it easy for users to operate on data without having to first load it into a metadata system. But for production systems that run over the same data every hour or every day, it has a couple of significant drawbacks. One, whenever your data changes, you have to change your Pig Latin. Two, although this works fine on data with 5 columns, it is painful when your data has 100 columns. To address these issues, there is another way to load schemas in Pig. If the load function you are using already knows the schema of the data, the function can communicate that to Pig. Load functions might already know the schema because it is stored in a metadata repository such as HCatalog, or it might be stored in the data itself. you can still refer to fields by name because Pig will fetch the schema from the load function before doing error checking on your script:

**mdata = load 'mydata' using HCatLoader();**

**cleansed = filter mdata by name is not null;**

**...**

Pig will determine whether it can adapt the one returned by the loader to match the one you gave. For example, if you specified a field as a long and the loader said it was an int, Pig can and will do that cast. However, if it cannot determine a way to make the loader’s schema fit the one you gave, it will give an error.

--no\_schema.pig

daily = load 'NYSE\_daily';

calcs = foreach daily generate $7 / 1000, $3 \* 100.0, SUBSTRING($0, 0, 1), $6 - $3;

In the expression $7 / 1000, 1000 is an integer, so it is a safe guess that the eighth field of NYSE\_daily is an integer or something that can be cast to an integer. In the same way, $3 \* 100.0 indicates $3 is a double, and the use of $0 in a function that takes a chararray as an argument indicates the type of $0. But what about the last expression, $6 - $3? The - operator is used only with numeric types in Pig, so Pig can safely guess that $3 and $6 are numeric. But should it treat them as integers or floating-point numbers? Here Pig plays it safe and guesses that they are floating points, casting them to doubles. This is the safer bet because if they actually are integers, those can be represented as floating-point numbers, but the reverse is not true. However, because floating-point arithmetic is much slower and subject to loss of precision, if these values really are integers, you should cast them so that Pig uses integer types in this case. There are also cases where Pig cannot make any intelligent guess:

--no\_schema\_filter

daily = load 'NYSE\_daily';

fltrd = filter daily by $6 > $3;

It is a valid operator on numeric, chararray, and bytearray types in Pig Latin. So, Pig has no way to make a guess. In this case, it treats these fields as if they were bytearrays, which means it will do a byte-to-byte comparison of the data in these fields. Pig also has to handle the case where it guesses wrong and must adapt on the fly. Consider the following:

--unintended\_walks.pig

player = load 'baseball' as (name:chararray, team:chararray,

pos:bag{t:(p:chararray)}, bat:map[]);

unintended = foreach player generate bat#'base\_on\_balls' - bat#'ibbs';

Because the values in maps can be of any type, Pig has no idea what type bat#'base\_on\_balls' and bat#'ibbs' are. By the rules laid out previously, Pig will assume they are doubles. But let’s say they actually turn out to be represented internally as integers. Pig will need to adapt at runtime and convert what it thought was a cast from bytearray to double into a cast from int to double. Note that it will still produce a double output and not an int output. This might seem nonintuitive; Finally, Pig’s knowledge of the schema can change at different points in the Pig Latin script. In all of the previous examples where we loaded data without a schema and then passed it to a foreach statement, the data started out without a schema. But after the foreach, the schema is known. Similarly, Pig can start out knowing the schema, but if the data is mingled with other data without a schema, the schema can be lost. That is, lack of schema is contagious:

**--no\_schema\_join.pig**

**divs = load 'NYSE\_dividends' as (exchange, stock\_symbol, date, dividends);**

**daily = load 'NYSE\_daily';**

**jnd = join divs by stock\_symbol, daily by $1;**

In this example, because Pig does not know the schema of daily, it cannot know the schema of the join of divs and daily.

**5.8.4 Casts**

The previous sections have referenced casts in Pig without bothering to define how casts work. The syntax for casts in Pig is the same as in Java—the type name in parentheses before the value:

**--unintended\_walks\_cast.pig**

**player = load 'baseball' as (name:chararray, team:chararray,**

**pos:bag{t:(p:chararray)}, bat:map[]);**

**unintended = foreach player generate (int)bat#'base\_on\_balls' - (int)bat#'ibbs';**

The syntax for specifying types in casts is exactly the same as specifying them in schemas.Not all conceivable casts are allowed.The following table describes which casts are allowed between scalar types. Casts to bytearrays are never allowed because Pig does not know how

to represent the various data types in binary format. Casts from bytearrays to any type are

allowed. Casts to and from complex types currently are not allowed, except from bytearray.

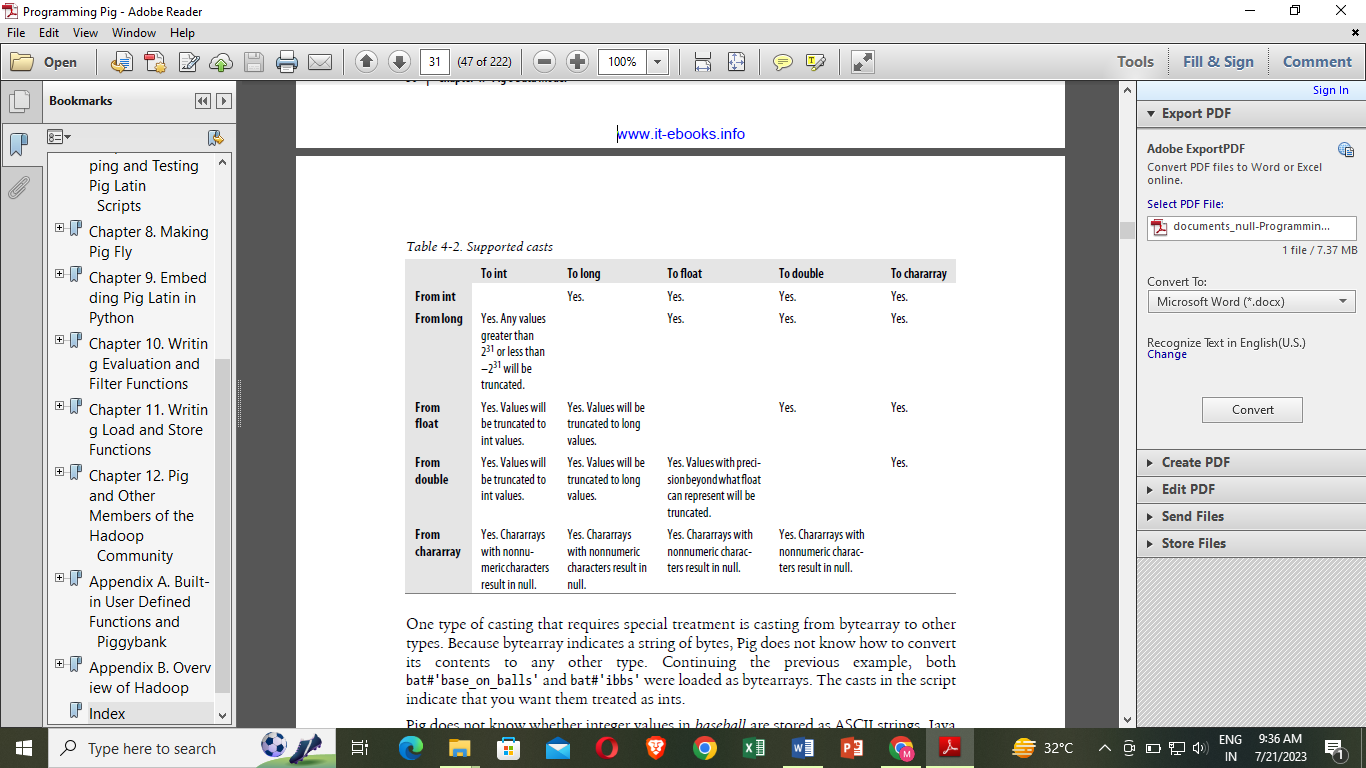


Fig :5.8.2 Supported casts

One type of casting that requires special treatment is casting from bytearray to other types. Because bytearray indicates a string of bytes, Pig does not know how to convert its contents to any other type. Continuing the previous example, both bat#'base\_on\_balls' and bat#'ibbs' were loaded as bytearrays. The casts in the script indicate that you want them treated as ints.

Pig does not know whether integer values in baseball are stored as ASCII strings, Java serialized values, binary-coded decimal, or some other format. So it asks the load function, because it is that function’s responsibility to cast bytearrays to other types. In general this works nicely, but it does lead to a few corner cases where Pig does not know how to cast a bytearray. In particular, if a UDF returns a bytearray, Pig will not know how to perform casts on it because that bytearray is not generated by a load function.

Before leaving the topic of casts, we need to consider cases where Pig inserts casts for the user. These casts are implicit, compared to explicit casts where the user indicates the cast. Consider the following:

--total\_trade\_estimate.pig

daily = load 'NYSE\_daily' as (exchange:chararray, symbol:chararray,

date:chararray, open:float, high:float, low:float, close:float,

volume:int, adj\_close:float);

rough = foreach daily generate volume \* close;

In this case, Pig will change the second line to (float)volume \* close to do the operation without losing precision. In general, Pig will always widen types to fit when it needs to insert these implicit casts. So, int and long together will result in a long; int or long and float will result in a float; and int, long, or float and double will result in a double. There are no implicit casts between numeric types and chararrays or other types.

**5.9 Pig Latin**

**5.9.1 Preliminary Matters**

Pig Latin is a dataflow language. Each processing step results in a new data set, or relation. In input = load 'data', input is the name of the relation that results from loading the data set data. A relation name is referred to as an alias. Relation names look like variables, but they are not. Once made, an assignment is permanent. It is possible to reuse relation names; for example, this is legitimate:

**A = load 'NYSE\_dividends' (exchange, symbol, date, dividends);**

**A = filter A by dividends > 0;**

**A = foreach A generate UPPER(symbol);**

However, it is not recommended. It looks here as if you are reassigning A, but really you are creating new relations called A, losing track of the old relations called A. It leads to confusion when trying to read your programs and when reading error messages.

Both relation and field names must start with an alphabetic character, and then they can have zero or more alphabetic, numeric, or \_ (underscore) characters. All characters in the name must be ASCII.

**5.9.2 Case Sensitivity**

Pig Latin cannot decide whether it is case-sensitive. Keywords in Pig Latin are not case-sensitive; for example, LOAD is equivalent to load. But relation and field names are. So A = load 'foo'; is not equivalent to a = load 'foo';. UDF names are also case-sensitive, thus COUNT is not the same UDF as count.

**5.9.3 Comments**

Pig Latin has two types of comment operators: SQL-style single-line comments (--) and Java-style multiline comments (/\* \*/). For example:

**A = load 'foo'; --this is a single-line comment**

**/\***

**\* This is a multiline comment.**

**\*/**

**B = load /\* a comment in the middle \*/'bar';**

# **5.9.4 Input and Output**

# we need to be able to add inputs and outputs to your data flows.

## 5.9.4.1Load

The first step to any data flow is to specify your input. In Pig Latin this is done with the load statement. By default, load looks for your data on HDFS in a tab-delimited file using the default load function PigStorage. divs = load '/data/examples/NYSE\_dividends'; will look for a file called NYSE\_dividends in the directory /data/examples. You can also specify relative path names. By default, your Pig jobs will run in your home directory on HDFS, /users/yourlogin. Unless you change directories, all relative paths will be evaluated from there. You can also specify a full URL for the path, for example, hdfs://nn.acme.com/data/examples/NYSE\_dividends to read the file from the HDFS instance that has nn.acme.com as a NameNode.

For example, if you wanted to load your data from HBase, you would use the loader for HBase:

**divs = load 'NYSE\_dividends' using HBaseStorage();**

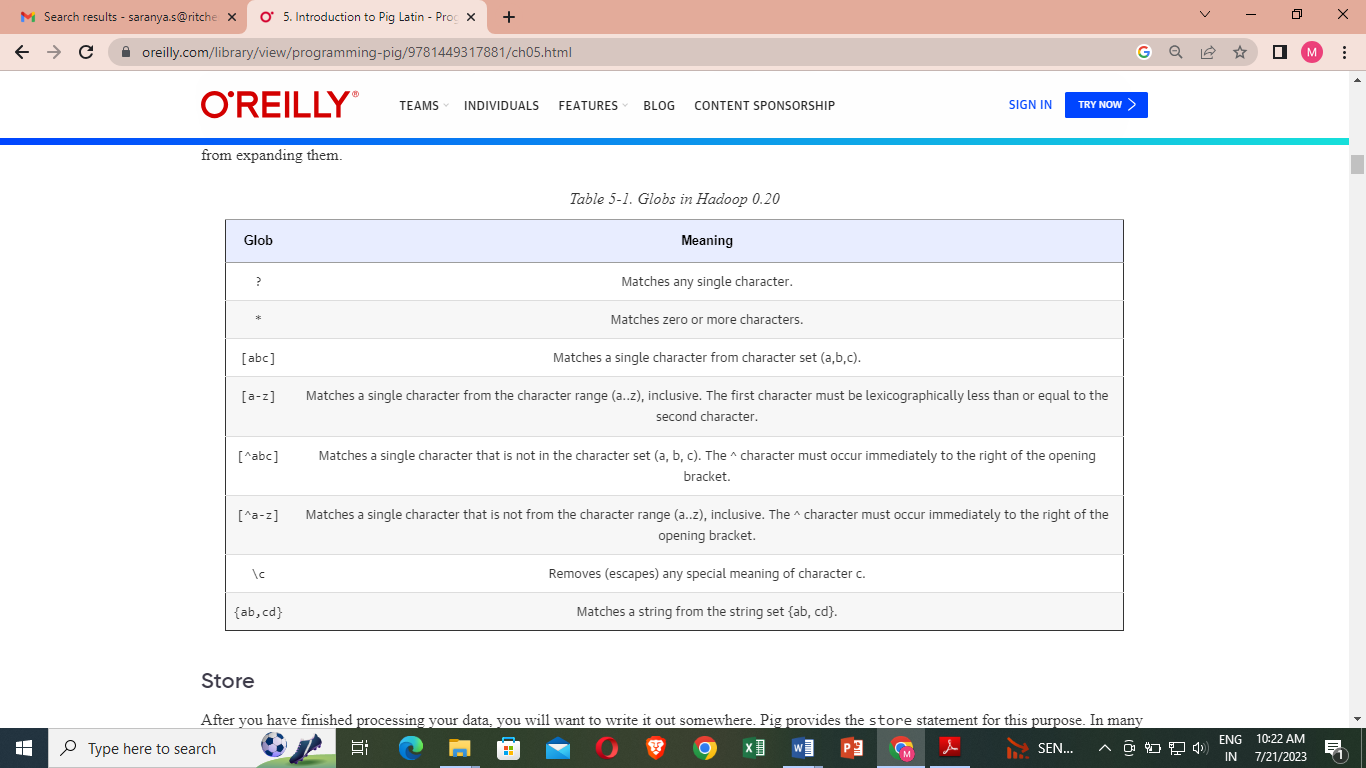
For example, if you are reading comma-separated text data, PigStorage takes an argument to indicate which character to use as a separator:

**divs = load 'NYSE\_dividends' using PigStorage(',');**

The load statement also can have an as clause, which allows you to specify the schema of the data you are loading.

**divs = load 'NYSE\_dividends' as (exchange, symbol, date, dividends);**

PigStorage and TextLoader, the two built-in Pig load functions that operate on HDFS files, support globs



**Fig 5.9.1 support globs**

**5.9.4.2 Store**

After you have finished processing your data, you will want to write it out somewhere. Pig provides the store statement for this purpose. In many ways it is the mirror image of the load statement. By default, Pig stores your data on HDFS in a tab-delimited file using PigStorage

**store processed into '/data/examples/processed';**

If you do not specify a store function, PigStorage will be used. You can specify a different store function with a using clause:

**store processed into 'processed' using**

**HBaseStorage();**

PigStorage takes an argument to indicate which character to use as a separator:

**store processed into 'processed' using PigStorage(',');**

**5.9.4.3 Dump**

In most cases you will want to store your data somewhere when you are done processing it. But occasionally you will want to see it on the screen. This is particularly useful during debugging and prototyping sessions. It can also be useful for quick ad hoc jobs. dump directs the output of your script to your screen:

dump processed;

Up through version 0.7, the output of dump matches the format of constants in Pig Latin. So, longs are followed by an L, and floats by an F, and maps are surrounded by [] (brackets), tuples by () (parentheses), and bags by {} (braces).

**5.9.4.4Relational Operations**

Relational operators are the main tools Pig Latin provides to operate on your data. They allow you to transform it by sorting, grouping, joining, projecting, and filtering. This section covers the basic relational operators

**1.foreach**  
foreach takes a set of expressions and applies them to every record in the data pipeline, hence the name foreach. for example, the following code loads an entire record, but then removes all but the user and id fields from each record:

**A = load 'input' as (user:chararray, id:long, address:chararray, phone:chararray,**

**preferences:map[]);**

**B = foreach A generate user, id;**

### **2.Expressions in foreach**

### foreach supports an array of expressions. The simplest are constants and field references.

prices = load 'NYSE\_daily' as (exchange, symbol, date, open, high, low, close,

volume, adj\_close);

gain = foreach prices generate close - open;

gain2 = foreach prices generate $6 - $3;

Null values are viral for all arithmetic operators. That is, x + null = null for all values of x.

Pig also provides a binary condition operator, often referred to as bincond. It begins with a Boolean test, followed by a ?, then the value to return if the test is true, then a :, and finally the value to return if the test is false.

**2 == 2 ? 1 : 4 --returns 1**

**2 == 3 ? 1 : 4 --returns 4**

**null == 2 ? 1 : 4 -- returns null**

**2 == 2 ? 1 : 'fred' -- type error; both values must be of the same type**

To extract data from complex types, use the projection operators. For maps this is # (the pound or hash), followed by the name of the key as a string.

**bball = load 'baseball' as (name:chararray, team:chararray,**

**position:bag{t:(p:chararray)}, bat:map[]);**

**avg = foreach bball generate bat#'batting\_average';**

Tuple projection is done with ., the dot operator.

**A = load 'input' as (t:tuple(x:int, y:int));**

**B = foreach A generate t.x, t.$1;**

### **3.UDFs in foreach**

User Defined Functions (UDFs) can be invoked in foreach. These are called evaluation functions, or eval funcs.

**-- udf\_in\_foreach.pig**

**divs = load 'NYSE\_dividends' as (exchange, symbol, date, dividends);**

**--make sure all strings are uppercase**

**upped = foreach divs generate UPPER(symbol) as symbol, dividends;**

**grpd = group upped by symbol; --output a bag upped for each value of symbol**

**--take a bag of integers, produce one result for each group**

**sums = foreach grpd generate group, SUM(upped.dividends);**

### **4. Naming fields in foreach**

The result of each foreach statement is a new tuple, usually with a different schema than the tuple that was an input to foreach

**divs = load 'NYSE\_dividends' as (exchange:chararray, symbol:chararray,**

**date:chararray, dividends:float);**

**sym = foreach divs generate symbol;**

**describe sym;**

**sym: {symbol: chararray}**

## 5.Filter

The filter statement allows you to select which records will be retained in your data pipeline. A filter contains a predicate. If that predicate evaluates to true for a given record, that record will be passed down the pipeline. Otherwise, it will not.

Predicates can contain the equality operators you expect, including == to test equality, and !=, >, >=, <, and <=. These comparators can be used on any scalar data type. == and != can be applied to maps and tuples.

Pig Latin follows the operator precedence that is standard in most programming languages, where arithmetic operators have precedence over equality operators. So, x + y == a + b is equivalent to (x + y) == (a + b).

For chararrays, users can test to see whether the chararray matches a regular expression:

**-- filter\_matches.pig**

**divs = load 'NYSE\_dividends' as (exchange:chararray, symbol:chararray,**

**date:chararray, dividends:float);**

**startswithcm = filter divs by symbol matches 'CM.\*';**

## 6.Group

The group statement collects together records with the same key. It is the first operator we have looked at that shares its syntax with SQL, but it is important to understand that the grouping operator in Pig Latin is fundamentally different than the one in SQL.

**-- count.pig**

**daily = load 'NYSE\_daily' as (exchange, stock);**

**grpd = group daily by stock;**

**cnt = foreach grpd generate group, COUNT(daily);**

That example groups records by the key stock and then counts them. It is just as legitimate to group them and then store them for processing at a later time:

**-- group.pig**

**daily = load 'NYSE\_daily' as (exchange, stock);**

**grpd = group daily by stock;**

**store grpd into 'by\_group';**

**You can also group on multiple keys, but the keys must be surrounded by parentheses.**

**--twokey.pig**

**daily = load 'NYSE\_daily' as (exchange, stock, date, dividends);**

**grpd = group daily by (exchange, stock);**

**avg = foreach grpd generate group, AVG(daily.dividends);**

**describe grpd;**

**grpd: {group: (exchange: bytearray,stock: bytearray),daily: {exchange: bytearray,**

**stock: bytearray,date: bytearray,dividends: bytearray}}**

You can also use all to group together all of the records in your pipeline:

**--countall.pig**

**daily = load 'NYSE\_daily' as (exchange, stock);**

**grpd = group daily all;**

**cnt = foreach grpd generate COUNT(daily);**

The record coming out of group all has the chararray literal all as a key.

**7. Order by**

The order statement sorts your data for you, producing a total order of your output data. Total order means that not only is the data sorted in each partition of your data, it is also guaranteed that all records in partition n are less than all records in partition n - 1 for all n.

**--order.pig**

**daily = load 'NYSE\_daily' as (exchange:chararray, symbol:chararray,**

**date:chararray, open:float, high:float, low:float, close:float,**

**volume:int, adj\_close:float);**

**bydate = order daily by date;**

**--order2key.pig**

**daily = load 'NYSE\_daily' as (exchange:chararray, symbol:chararray,**

**date:chararray, open:float, high:float, low:float,**

**close:float, volume:int, adj\_close:float);**

**bydatensymbol = order daily by date, symbol;**

**8.Distinct**

The distinct statement is very simple. It removes duplicate records. It works only on entire records, not on individual fields:

--distinct.pig

-- find a distinct list of ticker symbols for each exchange

-- This load will truncate the records, picking up just the first two fields.

daily = load 'NYSE\_daily' as (exchange:chararray, symbol:chararray);

uniq = distinct daily;

**9.Join**

join is one of the workhorses of data processing, and it is likely to be in many of your Pig Latin scripts. join selects records from one input to put together with records from another input. This is done by indicating keys for each input.

**--join.pig**

**daily = load 'NYSE\_daily' as (exchange, symbol, date, open, high, low, close,**

**volume, adj\_close);**

**divs = load 'NYSE\_dividends' as (exchange, symbol, date, dividends);**

**jnd = join daily by symbol, divs by symbol;**

Like foreach, join preserves the names of the fields of the inputs passed to it. It also prepends the name of the relation the field came from, followed by a ::. Adding describe jnd; to the end of the previous example produces:

**jnd: {daily::exchange: bytearray,daily::symbol: bytearray,daily::date: bytearray,**

**daily::open: bytearray,daily::high: bytearray,daily::low: bytearray,**

**daily::close: bytearray,daily::volume: bytearray,daily::adj\_close: bytearray,**

**divs::exchange: bytearray,divs::symbol: bytearray,divs::date: bytearray,**

**divs::dividends: bytearray}**

Pig also supports outer joins. A full outer join means records from both sides are taken even when they do not have matches:

--leftjoin.pig

daily = load 'NYSE\_daily' as (exchange, symbol, date, open, high, low, close,

volume, adj\_close);

divs = load 'NYSE\_dividends' as (exchange, symbol, date, dividends);

jnd = join daily by (symbol, date) left outer, divs by (symbol, date);

Pig can also do multiple joins in a single operation, as long as they are all being joined on the same key(s). This can be done only for inner joins:

A = load 'input1' as (x, y);

B = load 'input2' as (u, v);

C = load 'input3' as (e, f);

alpha = join A by x, B by u, C by e;

Self joins are supported, though the data must be loaded twice:

--selfjoin.pig

-- For each stock, find all dividends that increased between two dates

divs1 = load 'NYSE\_dividends' as (exchange:chararray, symbol:chararray,

date:chararray, dividends);

divs2 = load 'NYSE\_dividends' as (exchange:chararray, symbol:chararray,

date:chararray, dividends);

jnd = join divs1 by symbol, divs2 by symbol;

increased = filter jnd by divs1::date < divs2::date and

divs1::dividends < divs2::dividends;

**10.Limit**

Sometimes you want to see only a limited number of results. limit allows you do this:

**--limit.pig**

**divs = load 'NYSE\_dividends';**

**first10 = limit divs 10;**

The example here will return at most 10 lines (if your input has less than 10 lines total, it will return them all).

**11.Sample**

Sample offers a simple way to get a sample of your data. It reads through all of your data but returns only a percentage of rows. What percentage it returns is expressed as a double value, between 0 and 1. So, in the following example, 0.1 indicates 10%:

**--sample.pig**

**divs = load 'NYSE\_dividends';**

**some = sample divs 0.1;**

**12.Parallel**

One of Pig’s core claims is that it provides a language for parallel data processing.

The parallel clause can be attached to any relational operator in Pig Latin. However, it controls only reduce-side parallelism, so it makes sense only for operators that force a reduce phase. These are: group\*, order, distinct, join\*, limit, cogroup\*, and cross

**--parallel.pig**

**daily = load 'NYSE\_daily' as (exchange, symbol, date, open, high, low, close,**

**volume, adj\_close);**

**bysymbl = group daily by symbol parallel 10;**

**13.User Defined Functions**

Much of the power of Pig lies in its ability to let users combine irs operators with their own or others’ code via UDFs. Up through version 0.7, all UDFs must be written in Java and are implemented as Java classes. Pig itself comes packaged with some UDFs.

Piggybank is a collection of user-contributed UDFs that is packaged and released along with Pig. Piggybank UDFs are not included in the Pig JAR, and thus you have to register them manually in your script.

**14.Registering UDFs**

When you use a UDF that is not already built into Pig, you have to tell Pig where to look for that UDF. This is done via the register command.

**--register.pig**

**register 'your\_path\_to\_piggybank/piggybank.jar';**

**divs = load 'NYSE\_dividends' as (exchange:chararray, symbol:chararray,**

**date:chararray, dividends:float);**

**backwards = foreach divs generate**

**org.apache.pig.piggybank.evaluation.string.Reverse(symbol);**

This example tells Pig that it needs to include code from your\_path\_to\_piggybank/piggybank.jar when it produces a JAR to send to Hadoop.

**15.Registering Python UDFs**

Register is also used to locate resources for Python UDFs that you use in your Pig Latin scripts. In this case you do not register a JAR, but rather a Python script that contains your UDF. The Python script must be in your current directory. Using the examples provided in the example code, copying udfs/python/production.py to the data directory looks like this:

--batting\_production.pig

register 'production.py' using jython as bballudfs;

players = load 'baseball' as (name:chararray, team:chararray,

pos:bag{t:(p:chararray)}, bat:map[]);

nonnull = filter players by bat#'slugging\_percentage' is not null and

bat#'on\_base\_percentage' is not null;

calcprod = foreach nonnull generate name, bballudfs.production(

(float)bat#'slugging\_percentage',

(float)bat#'on\_base\_percentage');

**16.Define and UDF**

Define can be used to provide an alias so that you do not have to use full package names for your Java UDFs. It can also be used to provide constructor arguments to your UDFs. define also is used in defining streaming commands, but this section covers only its UDF-related features.

**--define.pig**

**register 'your\_path\_to\_piggybank/piggybank.jar';**

**define reverse org.apache.pig.piggybank.evaluation.string.Reverse();**

**divs = load 'NYSE\_dividends' as (exchange:chararray, symbol:chararray,**

**date:chararray, dividends:float);**

**backwards = foreach divs generate reverse(symbol);**

**17.Calling Static Java Function**  
 Java has a rich collection of utilities and libraries. Because Pig is implemented in Java, some of these functions can be exposed to Pig users. Any public static Java function that takes no arguments or some combination of int, long, float, double, String, or arrays thereof and returns int, long, float, double, or String can be invoked in this way. Because Pig Latin does not support overloading on return types, there is an invoker for each return type: InvokeForInt, InvokeForLong, InvokeForFloat, InvokeForDouble, and InvokeForString. You must pick the appropriate invoker for the type you wish to return. For example, if you wanted to use Java’s Integer class to translate decimal values to hexadecimal values, you could do:

--invoker.pig

define hex InvokeForString('java.lang.Integer.toHexString', 'int');

divs = load 'NYSE\_daily' as (exchange, symbol, date, open, high, low,

close, volume, adj\_close);

nonnull = filter divs by volume is not null;

inhex = foreach nonnull generate symbol, hex((int)volume);

**5.10 Developing and Testing pig Latin Script**

**5.10.1 Development Tools**

Pig provides several tools and diagnostic operators to help you develop your applications.**s**ome tools others have written to make it easier to develop Pig with standard editors and integrated development environments (IDEs).

**5.10.1.1 Syntax Highlighting and Checking**

Syntax highlighting often helps users write code correctly, at least syntactically, the first time around. Syntax highlighting packages exist for several popular editors.

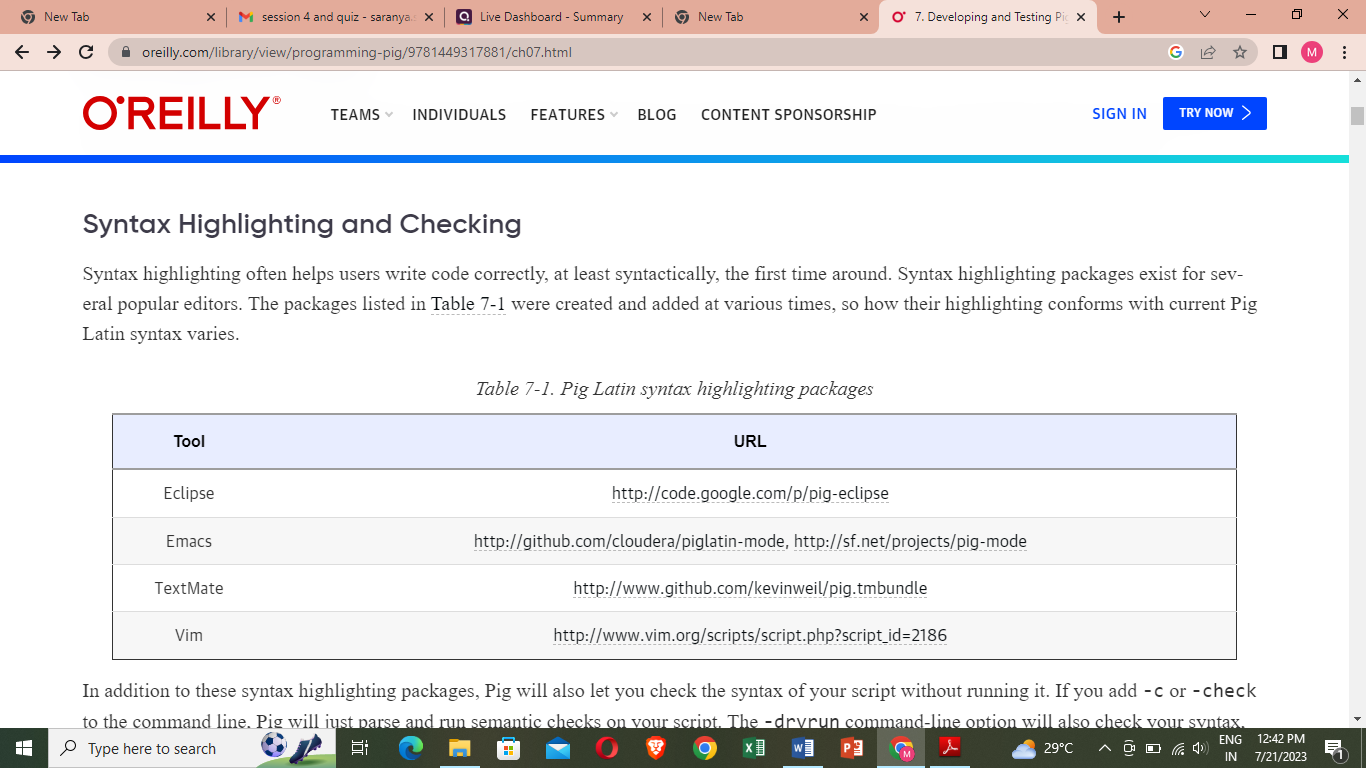


Fig 5.10.1

If you add -c or -check to the command line, Pig will just parse and run semantic checks on your script. The -dryrun command-line option will also check your syntax, expand any macros and imports, and perform parameter substitution.

**5.10.1.2 Describe**

describe shows you the schema of a relation in your script. This can be very helpful as you are developing your scripts. It is especially useful as you are learning Pig Latin and understanding how various operators change the data. describe can be applied to any relation in your script, and you can have multiple describes in a script:

**--describe.pig**

**divs = load 'NYSE\_dividends' as (exchange:chararray, symbol:chararray,**

**date:chararray, dividends:float);**

**trimmed = foreach divs generate symbol, dividends;**

**grpd = group trimmed by symbol;**

**avgdiv = foreach grpd generate group, AVG(trimmed.dividends);**

**describe trimmed;**

**describe grpd;**

**describe avgdiv;**

**trimmed: {symbol: chararray,dividends: float}**

**grpd: {group: chararray,trimmed: {(symbol: chararray,dividends: float)}}**

**avgdiv: {group: chararray,double}**

**5.10.1.3 Explain**

Explain is particularly helpful when you are trying to optimize your scripts or debug errors. There are two ways to use explain. You can explain any alias in your Pig Latin script, which will show the execution plan Pig would use if you stored that relation. You can also take an existing Pig Latin script and apply explain to the whole script in Grunt. This has a couple of advantages.

**--explain.pig**

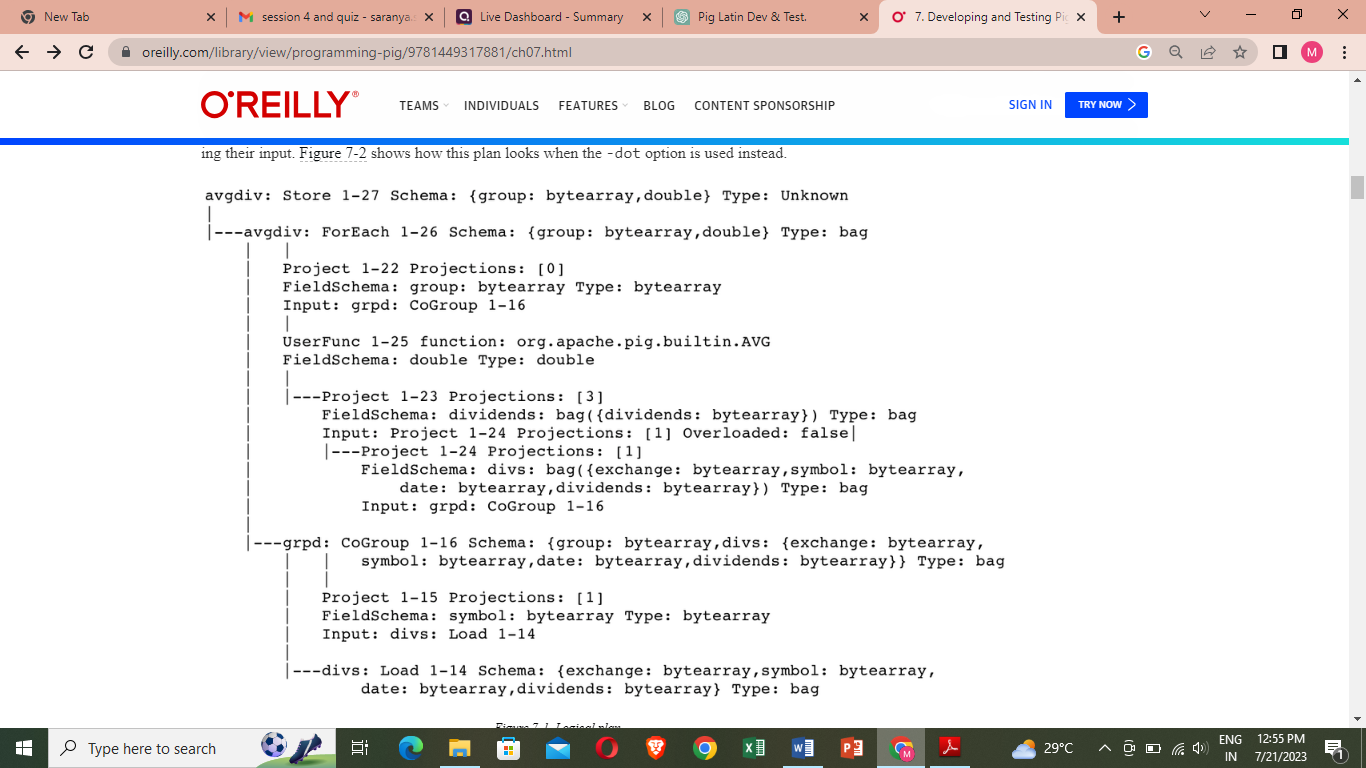
**divs = load 'NYSE\_dividends' as (exchange, symbol, date, dividends);**

**grpd = group divs by symbol;**

**avgdiv = foreach grpd generate group, AVG(divs.dividends);**

**store avgdiv into 'average\_dividend';**

**bin/pig -x local -e 'explain -script explain.pig'**

Fig 5.10.2 Logical plan

## 

Fig 5.10.3 Logical plan diagram

Pig goes through several steps to transform a Pig Latin script to a set of MapReduce jobs. After doing basic parsing and semantic checking, it produces a logical plan. This plan describes the logical operators that Pig will use to execute the script. Some optimizations are done on this plan.

The flow of this chart is bottom to top so that the Load operator is at the very bottom. The lines between operators show the flow. Each of the four operators created by the script (Load, CoGroup, ForEach, and Store) can be seen. Each of these operators also has a schema, described in standard schema syntax. The CoGroup and ForEach operators also have expressions attached to them

The ForEach operator has a projection expression that projects field 0 (the group field) and a UDF expression, which indicates that the UDF being used is org.apache.pig.builtin.AVG.

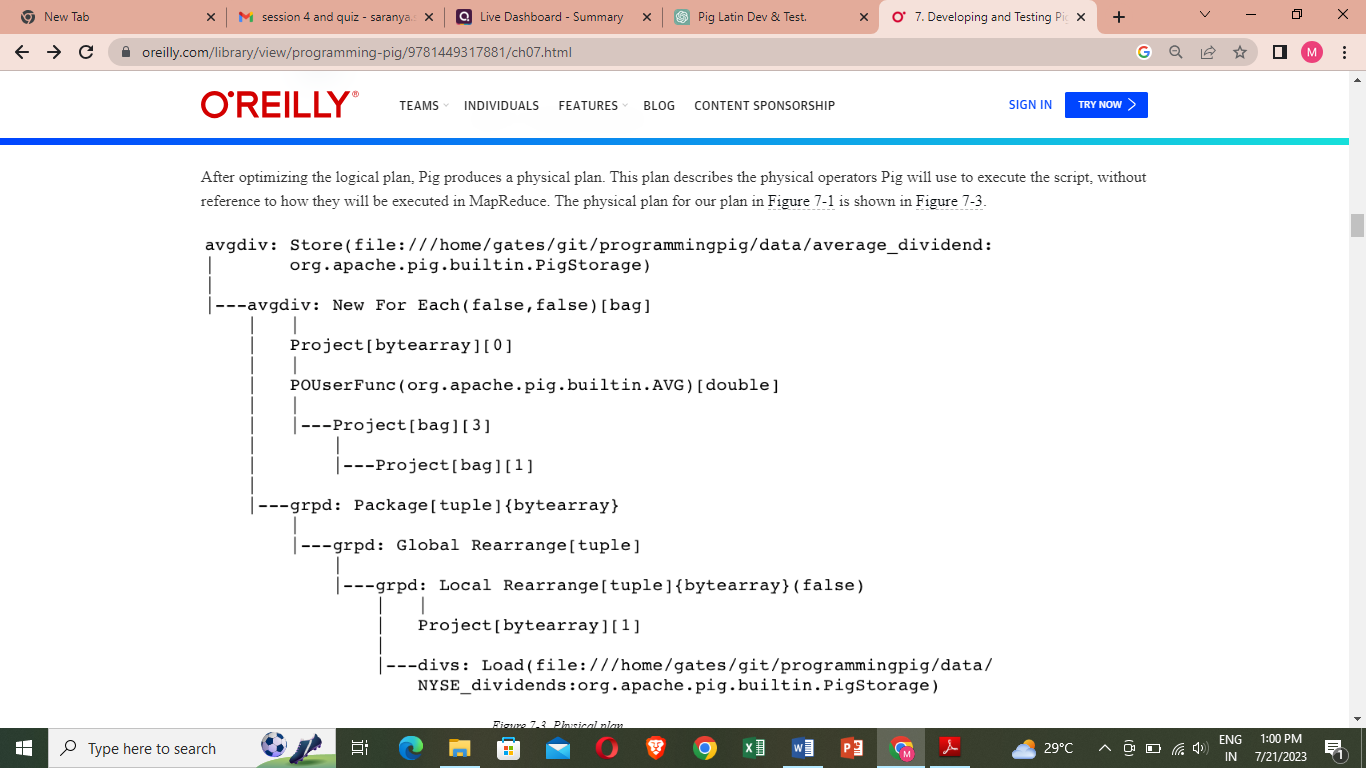


Fig 5.10.4 Physical plan

After optimizing the logical plan, Pig produces a physical plan. This plan describes the physical operators Pig will use to execute the script, without reference to how they will be executed in MapReduce.

This looks like the logical plan, but with a few differences. The load and store functions that will be used have been resolved. The other noticeable difference is that the CoGroup operator was replaced by three operators, Local Rearrange, Global Rearrange, and Package. Local Rearrange is the operator Pig uses to prepare data for the shuffle by setting up the key. Global Rearrange is a stand-in for the shuffle. Package sits in the reduce phase and directs records to the proper bag.

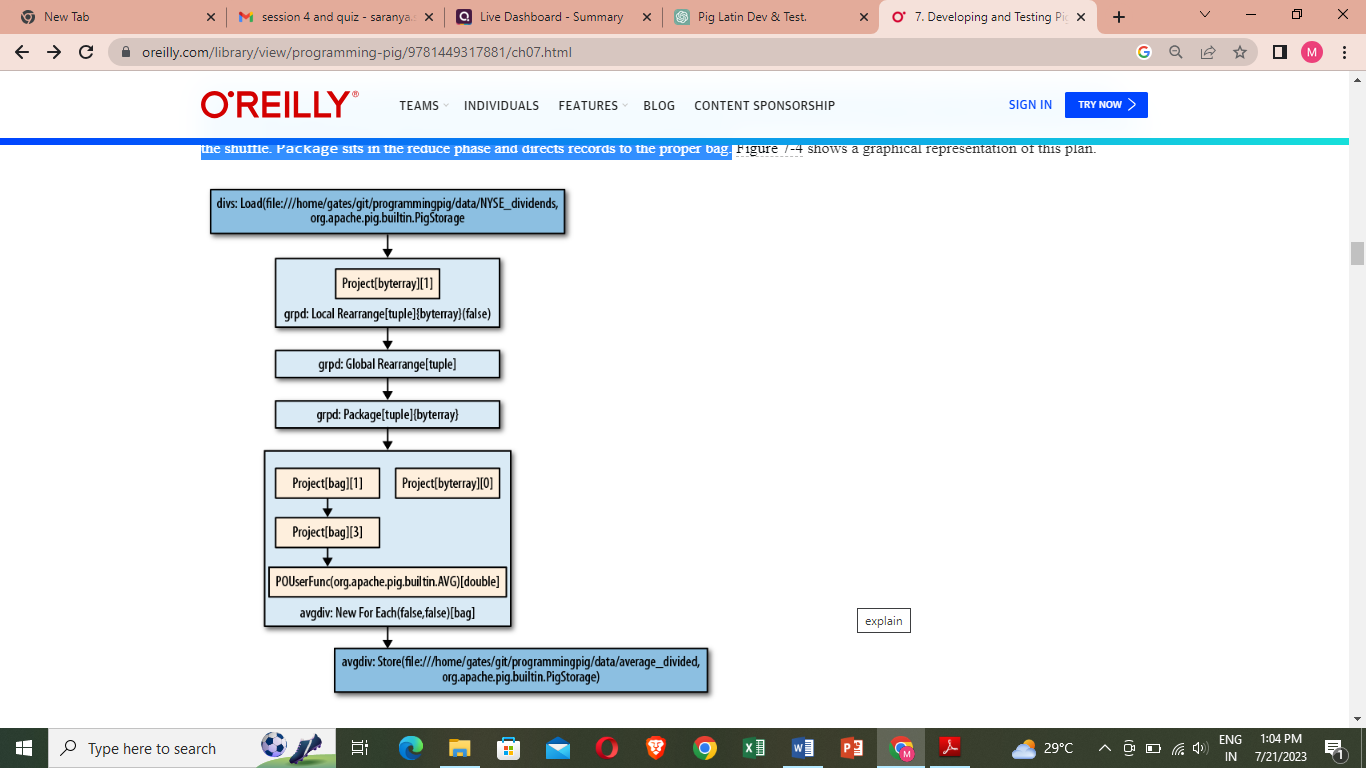


Fig 5.10.5Physical plan diagram

Finally, Pig takes the physical plan and decides how it will place its operators into one or more MapReduce jobs. First, it walks the physical plan looking for all operators that require a new reduce. This occurs anywhere there is a Local Rearrange, Global Rearrange, and Package. After it has done this, it sees whether there are places that it can do physical optimizations. The pipeline is now broken into three stages: map, combine, and reduce. The Global Rearrange operator is gone because it was a stand-in for the shuffle.The AVG UDF has been broken up into three stages: Initial in the map, Intermediate in the combiner, and Final in the reduce. If there were multiple MapReduce jobs in this example, they would all be shown in this output.



Fig 5.10.6 Map reduce plan

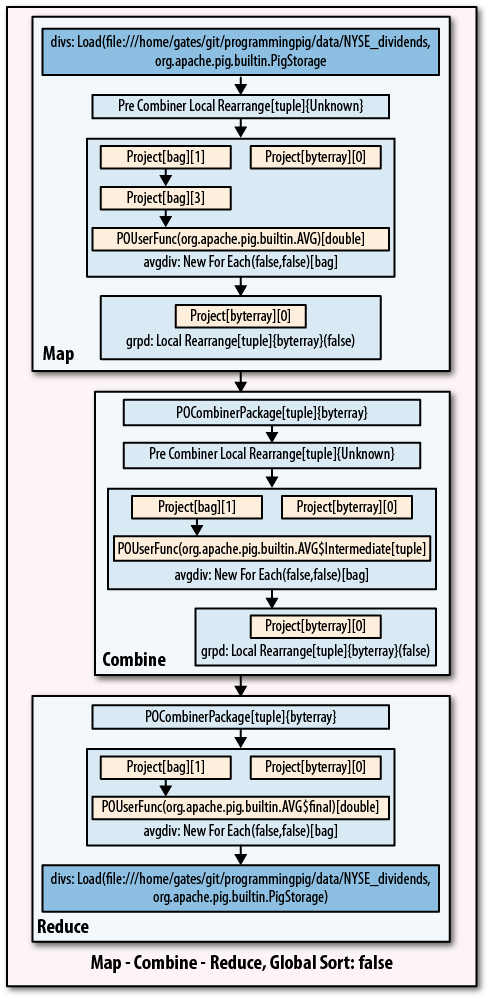


Fig 5.10.7 Map reduce plan diagram

**5.10.1.4 illustrate**

one of the best ways to debug your Pig Latin script is to run your data through it. But if you are using Pig, the odds are that you have a large data set. If it takes several hours to process your data, this makes for a very long debugging cycle. To use illustrate, apply it to an alias in your script, just as you would describe.

**--illustrate.pig**

**divs = load 'NYSE\_dividends' as (e:chararray, s:chararray, d:chararray, div:float);**

**recent = filter divs by d > '2009-01-01';**

**trimmd = foreach recent generate s, div;**

**grpd = group trimmd by s;**

**avgdiv = foreach grpd generate group, AVG(trimmd.div);**

**illustrate avgdiv;**

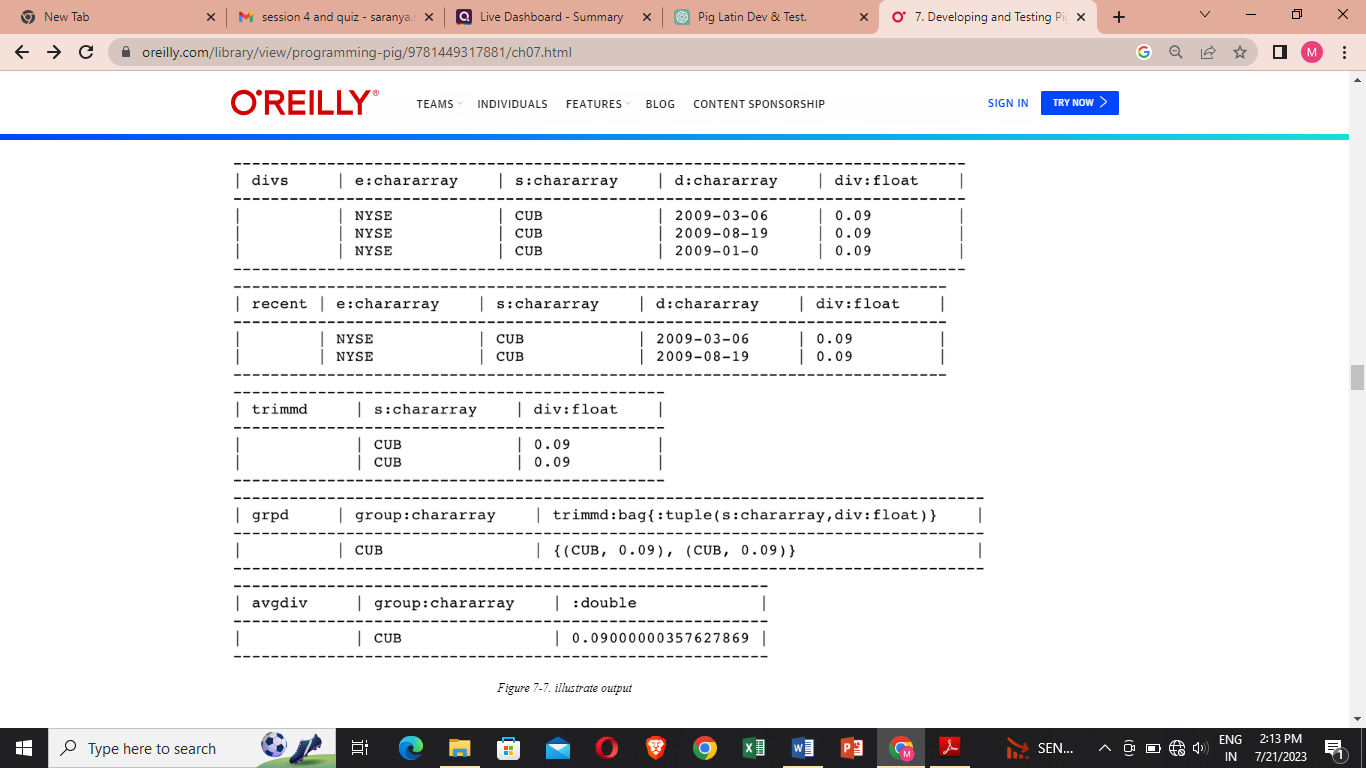


Fig 5.10.8 Illustrate output

**5.10.1.5 Pig Statistics**

Pig produces a summary set of statistics at the end of every run:

**--stats.pig**

**a = load '/user/pig/tests/data/singlefile/studenttab20m' as (name, age, gpa);**

**b = load '/user/pig/tests/data/singlefile/votertab10k'**

**as (name, age, registration, contributions);**

**c = filter a by age < '50';**

**d = filter b by age < '50';**

**e = cogroup c by (name, age), d by (name, age) parallel 20;**

**f = foreach e generate flatten(c), flatten(d);**

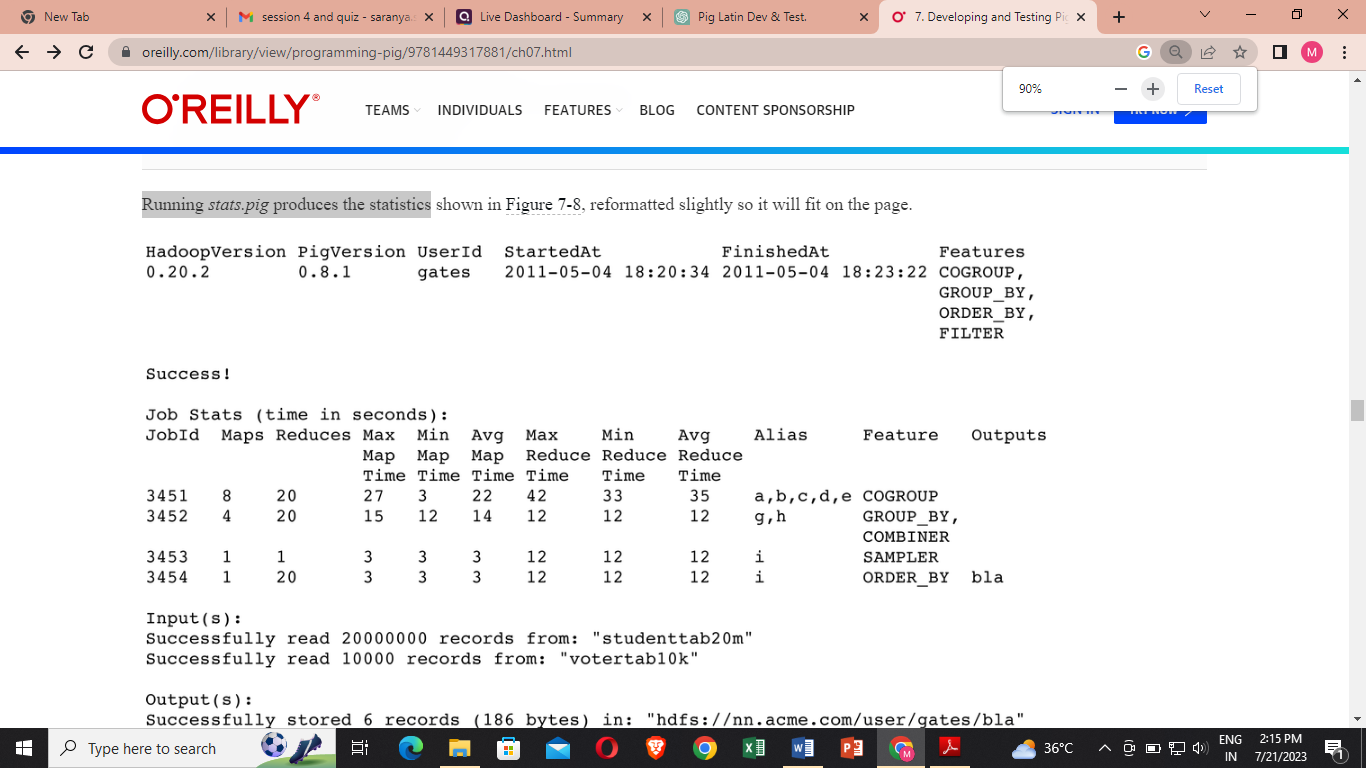
**g = group f by registration parallel 20;**

**h = foreach g generate group, SUM(f.d::contributions);**

**i = order h by $1, $0 parallel 20;**

**store i into 'student\_voter\_info';**

Running stats.pig produces the statistics



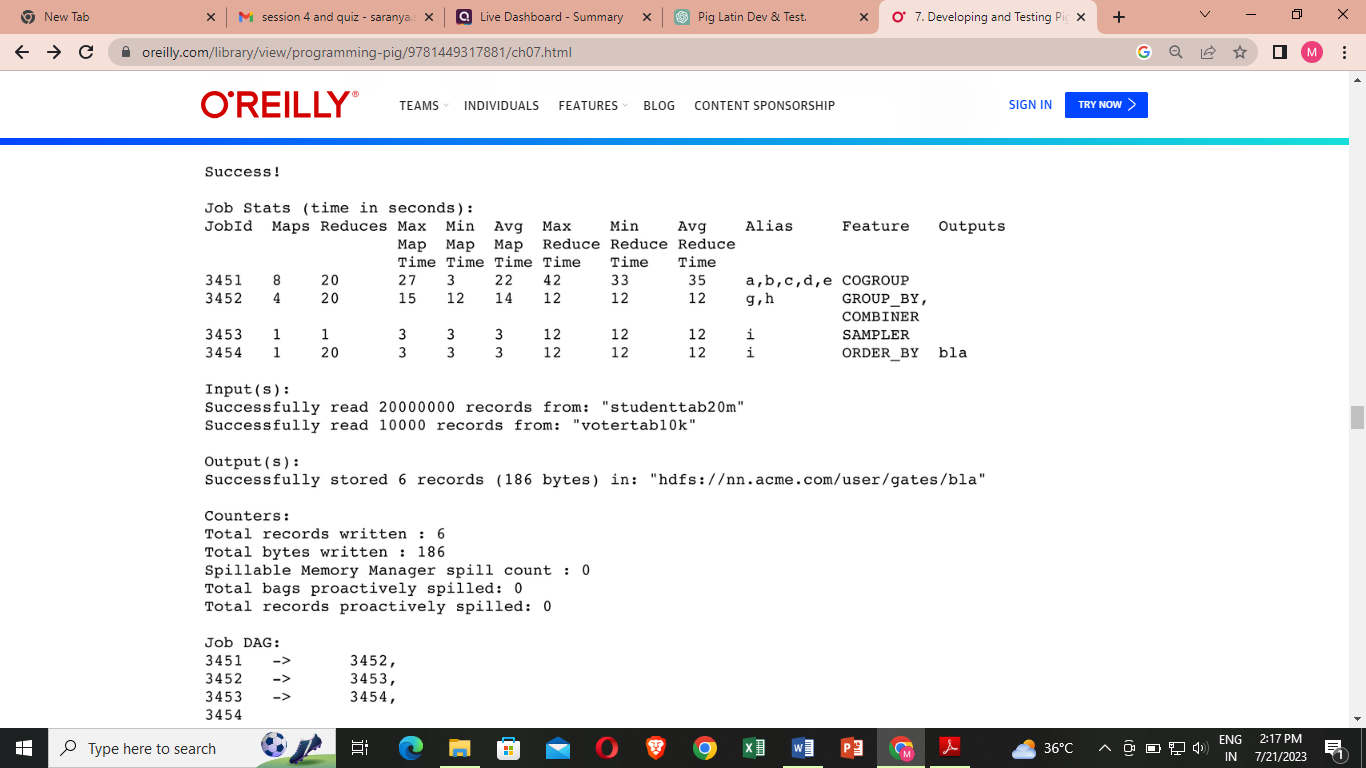


Fig 5.10.9 Statistics output of stats.pig

The first couple of lines give a brief summary of the job. StartedAt is the time Pig submits the job, not the time the first job starts running the Hadoop cluster.  FinishedAt is the time Pig finishes processing the job, which will be slightly after the time the last MapReduce job finishes. The section labeled Job Stats gives a breakdown of each MapReduce job that was run. The Input, Output, and Counters sections are self-explanatory. The statistics on spills record how many times Pig spilled records to local disk to avoid running out of memory. The Job DAG section at the end describes how data flowed between MapReduce jobs.

**5.10.1.6MapReduce Job Status**

When you are running your Pig Latin scripts on your Hadoop cluster, finding the status and logs of your job can be challenging. Logs generated by Pig while it plans and manages your query are stored in the current working directory. You can select a different directory by passing -l logdir on the command line. All data written to stdout and stderr by map and reduce tasks is also kept in the logs on the task nodes. The first step to locating your logs is to connect to the JobTracker’s web page. Generally, it is located at http://jt.acme.com:50030/jobtracker.jsp, where jt.acme.com is the address of your JobTracker.

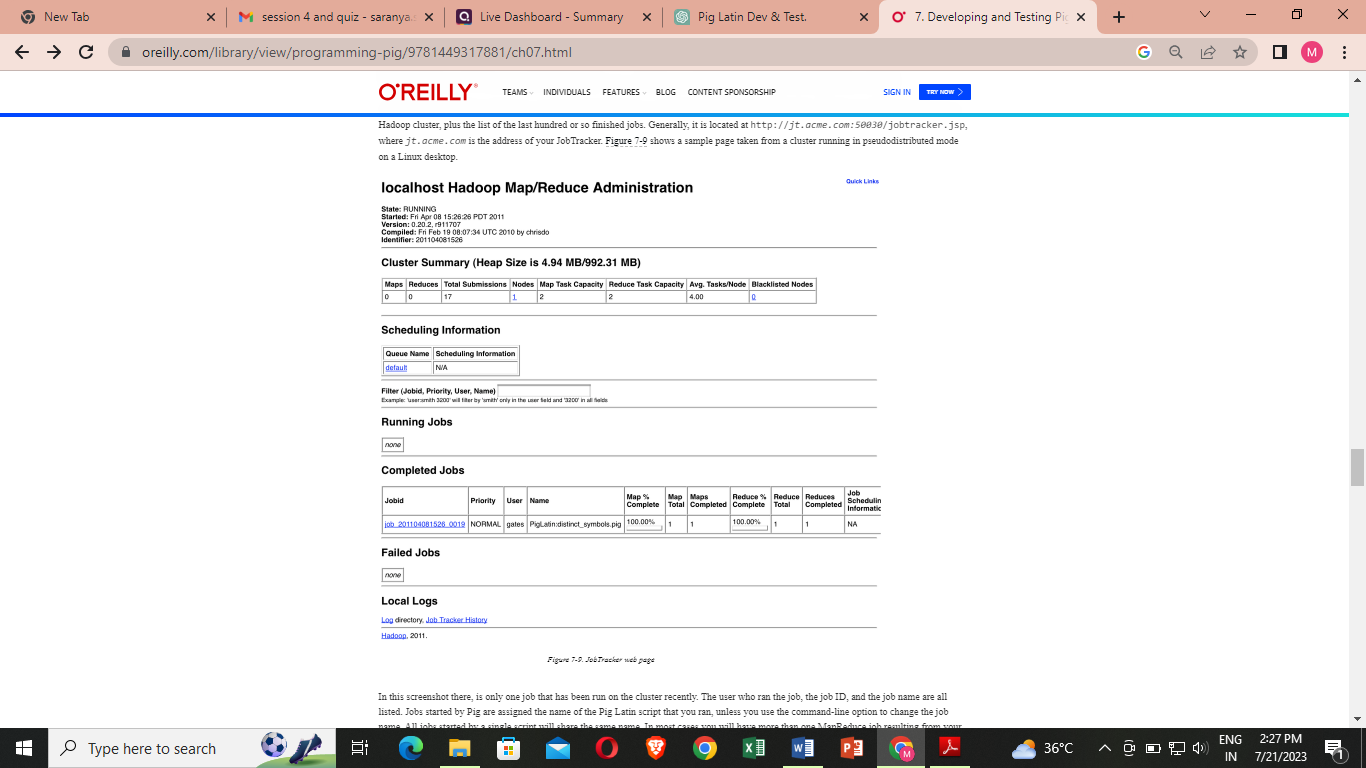


Fig 5.10.10 JobTracker web page

In this screenshot there, is only one job that has been run on the cluster recently. The user who ran the job, the job ID, and the job name are all listed. Jobs started by Pig are assigned the name of the Pig Latin script that you ran, unless you use the command-line option to change the job name. All jobs started by a single script will share the same name.

**Job Stats (time in seconds):**

**JobId ... Alias Feature**

**job\_201104081526\_0019 daily,grpd,uniqcnt GROUP\_BY,COMBINER**

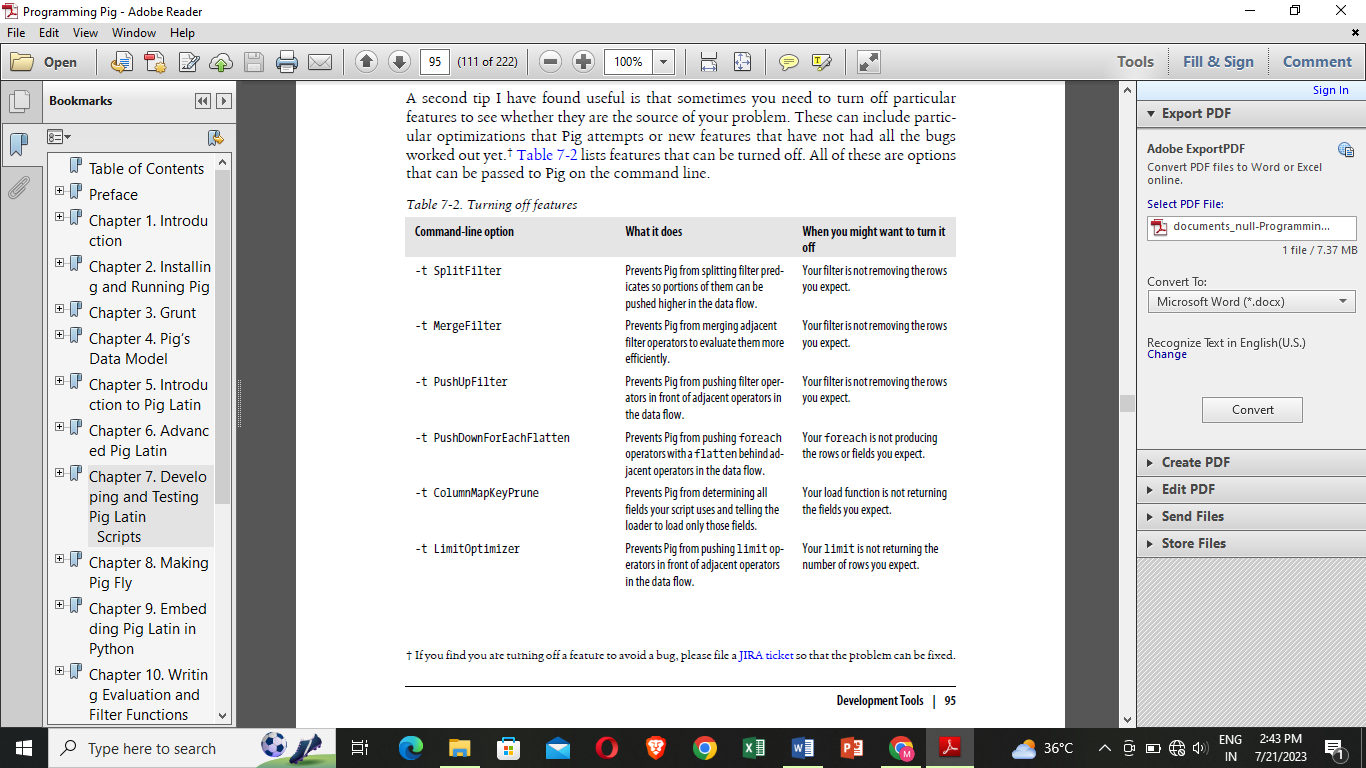
Clicking on the job ID will take you to a screen that summarizes the execution of the job, including when the job started and stopped, how many maps and reduces it ran, and the results of all of the counters.



Fig 5.10.11 Job web page

**5.10.1.7Debugging Tips**

There are a few things I have found useful in debugging Pig Latin scripts. First, if illustrate does not do what you need, use local mode to test your script before running it on your Hadoop cluster. Two, the logs for your operations appear on your screen, instead of being left on a task node somewhere. Three, local mode runs all in your local process. This means that you can attach a debugger to the process. This is particularly useful when you need to debug your UDFs. A second tip I have found useful is that sometimes you need to turn off particular features to see whether they are the source of your problem.



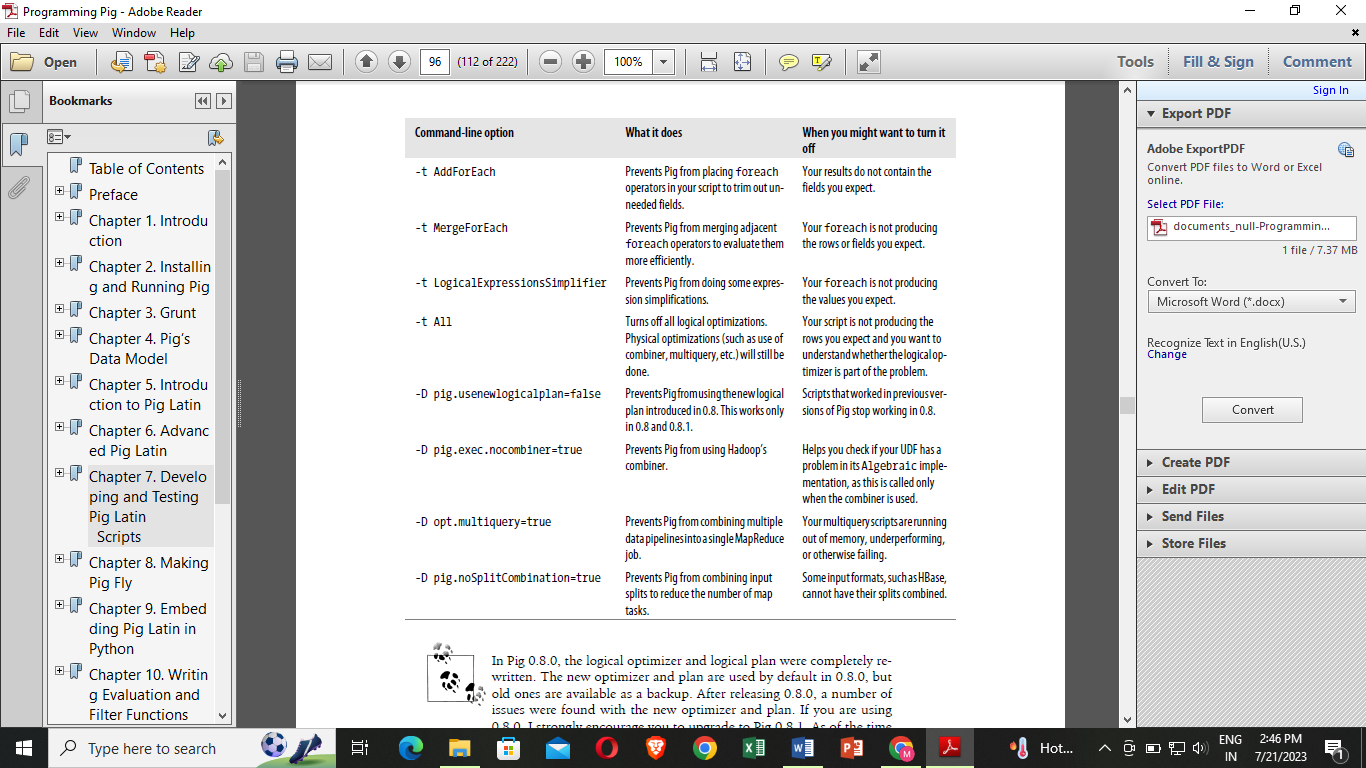


Fig 5.10.12 Turning off features

**5.10.2Testing Your Scripts with PigUnit**

PigUnit provides a unit-testing framework that plugs into JUnit to help you write unit tests that can be run on a regular basis.

**--pigunit.pig**

**divs = load 'NYSE\_dividends' as (exchange, symbol, date, dividends);**

**grpd = group divs all;**

**avgdiv = foreach grpd generate AVG(divs.dividends);**

**store avgdiv into 'average\_dividend';**

Second, you will need the pigunit.jar JAR file. This is not distributed as part of the standard Pig distribution, but you can build it from the source code included in your distribution. To do this, go to the directory your distribution is in and type ant jar pigunit-jar. Once this is finished, there should be two files in the directory: pig.jar and pigunit.jar. You will need to place these in your classpath when running PigUnit tests. Third, you need data to run through your script. You can use an existing input file, or you can manufacture some input in your test and run that through your script.

// java/example/PigUnitExample.java

public class PigUnitExample {

private PigTest test;

private static Cluster cluster;

@Test

public void testDataInFile() throws ParseException, IOException {

// Construct an instance of PigTest that will use the script

// pigunit.pig.

test = new PigTest("../pigunit.pig");

// Specify our expected output. The format is a string for each line.

// In this particular case we expect only one line of output.

String[] output = { "(0.27305267014925455)" };

// Run the test and check that the output matches our expectation.

// The "avgdiv" tells PigUnit what alias to check the output value

// against. It inserts a store for that alias and then checks the

// contents of the stored file against output.

test.assertOutput("avgdiv", output);

}

}

**5.11 HIVE**

Hive is a data warehousing and SQL-like data processing tool built on top of Apache Hadoop. It was developed by Facebook to simplify querying and analyzing large-scale datasets stored in Hadoop Distributed File System (HDFS) or other compatible storage systems.

**Key features of Hive include:**

* **HiveQL:** Hive Query Language (HiveQL) is a SQL-like language used to write queries for data processing. It allows users to express complex data transformations and analytics tasks in a familiar SQL syntax.
* **Schema on Read:** Hive provides a schema-on-read approach, which means the schema is applied when data is read, rather than when it is ingested. This flexibility allows Hive to handle semi-structured and unstructured data efficiently.
* **Metastore:** Hive maintains a metastore, typically backed by a relational database, to store metadata about the tables, columns, partitions, and other relevant information. This enables Hive to understand the structure of the data and optimize query execution.
* **Data Partitioning and Buckets:** Hive supports data partitioning and bucketing, which improves query performance by organizing data into smaller, manageable parts.
* **Integration with Hadoop Ecosystem:** Hive seamlessly integrates with other components of the Hadoop ecosystem, such as Hadoop Distributed File System (HDFS), Apache HBase, and Apache Spark.
* **Extensibility**: Hive is extensible, allowing users to add custom user-defined functions (UDFs) and user-defined aggregates (UDAs) to perform specialized operations on data.
* **Optimization:** Hive optimizes queries by using techniques like query optimization, predicate pushdown, and join optimization.
* Hive is particularly useful for analysts and data engineers who are familiar with SQL and want to leverage their SQL skills to work with big data. It abstracts the complexities of the underlying distributed computing infrastructure and allows users to focus on data analysis.
* To use Hive, you typically interact with it using its command-line interface (CLI) or through various data processing tools that support Hive connectivity. Hive queries are translated into MapReduce jobs (or other processing engines like Apache Tez or Apache Spark) for execution on the Hadoop cluster.

Keep in mind that Hive might not be the best choice for real-time data processing due to its batch-oriented nature. For real-time or interactive analytics, other technologies like Apache Spark with SparkSQL or Apache Impala might be more suitable.

**5.12 HIVE DATA TYPES AND FILE FORMATS:**

In Hive, data types define the type of data that can be stored in a column, and file formats determine how data is stored physically on disk. Hive supports various data types and file formats to accommodate different use cases and optimize data storage and processing. Below are some commonly used data types and file formats in Hive:

**5.12.1Hive Data Types:**

**1.Primitive Data Types:**

* **TINYINT**: 1-byte signed integer (-128 to 127)
* **SMALLINT**: 2-byte signed integer (-32,768 to 32,767)
* **INT or INTEGER**: 4-byte signed integer (-2,147,483,648 to 2,147,483,647)
* **BIGINT:** 8-byte signed integer (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)
* **FLOAT:** 4-byte single-precision floating-point number
* **DOUBLE**: 8-byte double-precision floating-point number
* **BOOLEAN:** Boolean (true or false)
* **STRING:** Variable-length character string
* **CHAR:** Fixed-length character string
* **VARCHAR:** Variable-length character string with a specified maximum length
* **DATE:** Date value in the format 'YYYY-MM-DD'
* **TIMESTAMP**: Timestamp value in the format 'YYYY-MM-DD HH:MM:SS.sss'

**2.Complex Data Types:**

* **ARRAY:** Ordered collection of elements of the same data type
* **MAP:** Collection of key-value pairs, where keys and values can have different data types
* **STRUCT:** Similar to a struct or record in programming, can have multiple named fields with different data types
* **UNIONTYPE:** A union of multiple data types

**5.12.2 Hive File Formats:**

* **TextFile:** Default file format in Hive, which stores data in plain text format. It is human-readable but not the most space-efficient format for large datasets.
* **SequenceFile:** A binary file format optimized for large datasets, offering better compression and efficient serialization/deserialization. It is widely used in the Hadoop ecosystem.
* **ORC (Optimized Row Columnar): ORC** is a columnar storage format that provides better compression and improved query performance. It organizes data into columns, enabling efficient data retrieval for specific columns during query execution.
* **Parquet:** Parquet is another columnar storage format that offers efficient compression and encoding techniques. It is commonly used in conjunction with Apache Spark and other big data processing frameworks.
* **Avro:** Avro is a data serialization system that allows schema evolution. It is a binary format with a JSON-like schema definition, making it compact and versatile.
* **RCFile (Record Columnar File):** RCFile is a columnar storage format that splits data into row groups, reducing the overhead of reading unnecessary columns during query execution.

Choosing the appropriate data type and file format depends on your data characteristics, query patterns, and storage and performance requirements. For example, for analytical workloads with large datasets, ORC or Parquet are often preferred due to their superior compression and columnar storage optimizations. On the other hand, for smaller datasets or when human readability is a priority, TextFile might be suitable.

**5.13 HIVEQL DATA DEFINITION:**

HiveQL is the Hive query language. Hive offers no support for rowlevel inserts, updates, and deletes. Hive doesn’t support transactions. which are used for creating, altering, and dropping databases, tables, views, functions, and indexes.

**5.13.1 Databases in Hive**

The Hive concept of a database is essentially just a catalog or namespace of tables. If you don’t specify a database, the default database is used. The simplest syntax for creating a database is shown in the following example:

**5.13.1.1 CREATE DATABASE**

hive> **CREATE DATABASE** financials;

hive> **CREATE DATABASE** IF **NOT EXISTS** financials;

You can also use the keyword SCHEMA instead of DATABASE in all the database-related

commands.

hive> **CREATE DATABASE** human\_resources;

hive> **SHOW** DATABASES;

**default**

financials

human\_resources

You can override this default location for the new directory as shown in this example:

hive> **CREATE DATABASE** financials

> **LOCATION** '/my/preferred/directory';

You can add a descriptive comment to the database. DESCRIBE DATABASE <database> command.

hive> **CREATE DATABASE** financials

> **COMMENT** 'Holds all financial tables';

hive> **DESCRIBE DATABASE** financials;

financials Holds **all** financial tables

hdfs://master-server/**user**/hive/warehouse/financials.db

Note that DESCRIBE DATABASE also shows the directory location for the database. If you are running in pseudo-distributed mode, then the master server will be localhost. For local mode, the path will be a local path, file:///user/hive/warehouse/financials.db.

The USE command sets a database as your working database, analogous to changing working directories in a filesystem:

hive> USE financials;

Now, commands such as SHOW TABLES; will list the tables in this database. Finally, you can drop a database:

hive> **DROP DATABASE** IF **EXISTS** financials;

hive> **DROP DATABASE** IF **EXISTS** financials **CASCADE**;

Using the RESTRICT keyword instead of CASCADE is equivalent to the default behavior,

where existing tables must be dropped before dropping the database.

**5.13.1.2 Alter Database**

We can set key-value pairs in the DBPROPERTIES associated with a database using the ALTER DATABASE command. No other metadata about the database can be changed,including its name and directory location:

hive> **ALTER DATABASE** financials **SET** DBPROPERTIES ('edited-by' = 'Joe Dba');

There is no way to delete or “unset” a DBPROPERTY.

**5.13.1.3 Creating Tables**

The CREATE TABLE statement follows SQL conventions, but Hive’s version offers significant extensions to support a wide range of flexibility where the data files for tables are stored, the formats used, etc.

**CREATE TABLE** IF **NOT EXISTS** mydb.employees (

name STRING **COMMENT** 'Employee name',

salary FLOAT **COMMENT** 'Employee salary',

subordinates ARRAY<STRING> **COMMENT** 'Names of subordinates',

deductions **MAP**<STRING, FLOAT>

**COMMENT** 'Keys are deductions names, values are percentages',

address STRUCT<street:STRING, city:STRING, **state**:STRING, zip:INT>

**COMMENT** 'Home address')

**COMMENT** 'Description of the table'

TBLPROPERTIES ('creator'='me', 'created\_at'='2012-01-02 10:00:00', ...)

**LOCATION** '/user/hive/warehouse/mydb.db/employees';

Hive automatically adds two table properties: last\_modified\_by holds the username of the last user to modify the table, and last\_modified\_time holds the epoch time in seconds of that modification.

The SHOW TABLES command lists the tables. With no additional arguments, it shows the tables in the current working database.

hive> USE mydb;

hive> **SHOW** TABLES;

employees

table1

table2

If we aren’t in the same database, we can still list the tables in that database:

hive> USE **default**;

hive> **SHOW** TABLES **IN** mydb;

employees

We can also use the DESCRIBE EXTENDED mydb.employees command to show details about

the table.

hive> **DESCRIBE** EXTENDED mydb.employees;

name string Employee name

salary float Employee salary

subordinates array<string> **Names of** subordinates

deductions **map**<string,float> Keys **are** deductions **names**, **values are** percentages

address struct<street:string,city:string,**state**:string,zip:int> Home address

Detailed **Table** Information **Table**(tableName:employees, dbName:mydb, **owner**:me,

...

**location**:hdfs://master-server/**user**/hive/warehouse/mydb.db/employees,

**parameters**:{creator=me, created\_at='2012-01-02 10:00:00',

last\_modified\_user=me, last\_modified\_time=1337544510,

**comment**:Description **of** the **table**, ...}, ...)

If you only want to see the schema for a particular column, append the column to the

table name. Here, EXTENDED adds no additional output:

hive> **DESCRIBE** mydb.employees.salary;

salary float Employee salary

**5.13.1.4 Managed Tables**

The tables we have created so far are called managed tables or sometimes called internal tables, because Hive controls the lifecycle of their data. When we drop a managed table Hive deletes the data in the table.

**5.13.1.5 External Tables**

The following table declaration creates an external table that can read all the data filesfor this comma-delimited data in /data/stocks:

**CREATE EXTERNAL TABLE** IF **NOT EXISTS** stocks (

exchange STRING,

symbol STRING,

ymd STRING,

price\_open FLOAT,

price\_high FLOAT,

price\_low FLOAT,

price\_close FLOAT,

volume INT,

price\_adj\_close FLOAT)

**ROW** FORMAT DELIMITED FIELDS TERMINATED **BY** ','

**LOCATION** '/data/stocks';

The EXTERNAL keyword tells Hive this table is external and the LOCATION … clause is required to tell Hive where it’s located. Because it’s external, Hive does not assume it owns the data. Therefore, dropping thetable does not delete the data, although the metadata for the table will be deleted.

You can tell whether or not a table is managed or external using the output of DESCRIBE EXTENDED tablename. Near the end of the Detailed Table Information output, you will see the following for managed tables:

... tableType:MANAGED\_TABLE)

For external tables, you will see the following:

... tableType:EXTERNAL\_TABLE)

As for managed tables, you can also copy the schema (but not the data) of an existing table:

**CREATE EXTERNAL TABLE** IF **NOT EXISTS** mydb.employees3

**LIKE** mydb.employees

**LOCATION** '/path/to/data';

**5.10.1.6 Partitioned, Managed Tables**

Hive has the notion of partitioned tables. partition the data first by country and then by state:

**CREATE TABLE** employees (

name STRING,

salary FLOAT,

subordinates ARRAY<STRING>,

deductions **MAP**<STRING, FLOAT>,

address STRUCT<street:STRING, city:STRING, **state**:STRING, zip:INT>

)

PARTITIONED **BY** (country STRING, **state** STRING);

Partitioning tables changes how Hive structures the data storage. If we create this table in the mydb database, there will still be an employees directory for the table: hdfs://master\_server/user/hive/warehouse/mydb.db/employees. However, Hive will now create subdirectories reflecting the partitioning structure. For example:

...

.../employees/country=CA/state=AB

.../employees/country=CA/state=BC

...

.../employees/country=US/state=AL

.../employees/country=US/state=AK

...

Once created, the partition keys, When we add predicates to WHERE clauses that filter on partition values, these predicates are called partition filters.

You can see the partitions that exist with the SHOW PARTITIONS command:

hive> **SHOW** PARTITIONS employees;

...

Country=CA/**state**=AB

country=CA/**state**=BC

...

country=US/**state**=AL

country=US/**state**=AK

...

The DESCRIBE EXTENDED employees command shows the partition keys:

hive> **DESCRIBE** EXTENDED employees;

name string,

salary float,

...

address struct<...>,

country string,

**state** string

Detailed **Table** Information...

partitionKeys:[FieldSchema(name:country, **type**:string, **comment**:**null**),

FieldSchema(name:**state**, **type**:string, **comment**:**null**)],

...

We create partitions in managed tables by loading data into them.

**LOAD DATA LOCAL** INPATH '${env:HOME}/california-employees'

**INTO TABLE** employees

PARTITION (country = 'US', **state** = 'CA');

**5.13.1.7 External Partitioned Tables**

You can use partitioning with external tables.

**CREATE EXTERNAL TABLE** IF **NOT EXISTS** log\_messages (

hms INT,

severity STRING,

server STRING,

process\_id INT,

message STRING)

PARTITIONED **BY** (**year** INT, **month** INT, **day** INT)

**ROW** FORMAT DELIMITED FIELDS TERMINATED **BY** '\t';

An interesting benefit of this flexibility is that we can archive old data on inexpensive storage, like Amazon’s S3, while keeping newer, more “interesting” data in HDFS. For example, each day we might use the following procedure to move data older than a month to S3:

• Copy the data for the partition being moved to S3. For example, you can use the hadoop distcp command:

hadoop distcp /data/log\_messages/2011/12/02 s3n://ourbucket/logs/2011/12/02

• Alter the table to point the partition to the S3 location:

**ALTER TABLE** log\_messages PARTITION(**year** = 2011, **month** = 12, **day** = 2)

**SET LOCATION** 's3n://ourbucket/logs/2011/01/02';

• Remove the HDFS copy of the partition using the hadoop fs -rmr command:

hadoop fs -rmr /data/log\_messages/2011/01/02

As for managed partitioned tables, you can see an external table’s partitions with SHOW PARTITIONS:

hive> **SHOW** PARTITIONS log\_messages;

...

**year**=2011/**month**=12/**day**=31

**year**=2012/**month**=1/**day**=1

**year**=2012/**month**=1/**day**=2

...

**5.12.1.8 Customizing Table Storage Formats**

Hive defaults to a text file format, which is indicated by the optional clause STORED AS TEXTFILE, and you can overload the default values for the various delimiters when creating the table.

**CREATE TABLE** employees (

name STRING,

salary FLOAT,

subordinates ARRAY<STRING>,

deductions **MAP**<STRING, FLOAT>,

address STRUCT<street:STRING, city:STRING, **state**:STRING, zip:INT>

)

**ROW** FORMAT DELIMITED

FIELDS TERMINATED **BY** '\001'

COLLECTION ITEMS TERMINATED **BY** '\002'

**MAP** KEYS TERMINATED **BY** '\003'

LINES TERMINATED **BY** '\n'

STORED **AS** TEXTFILE;

TEXTFILE implies that all fields are encoded using alphanumeric characters, includingthose from international character sets, although we observed that Hive uses nonprinting characters as “terminators” (delimiters), by default. When TEXTFILE is used, each line is considered a separate record. You can replace TEXTFILE with one of the other built-in file formats supported by Hive,including SEQUENCEFILE and RCFILE, both of which optimize disk space usage and I/O bandwidth performance using binary encoding and optional compression. The record encoding is handled by an input format object (e.g., the Java code behind TEXTFILE.) Hive uses a Java class (compiled module) named org.apache .hadoop.mapred.TextInputFormat.

The record parsing is handled by a serializer/deserializer or SerDe for short. For completeness, there is also an output format that Hive uses for writing the output of queries to files and to the console. The ROW FORMAT SERDE … specifies the SerDe to use. Hive provides the WITH SERDEPRO PERTIES feature that allows users to pass configuration information to the SerDe. Finally, the STORED AS INPUTFORMAT … OUTPUTFORMAT … clause specifies the Java classes to use for the input and output formats, respectively. If you specify one of these formats, you are required to specify both of them.

**5.13.1.9 Dropping Tables**

The familiar DROP TABLE command from SQL is supported:

**DROP TABLE** IF **EXISTS** employees;

The IF EXISTS keywords are optional. If not used and the table doesn’t exist, Hive returns an error. For managed tables, the table metadata and data are deleted.

**5.13.1.10 Alter Table**

Most table properties can be altered with ALTER TABLE statements, which change

metadata about the table but not the data itself.

**5.13.1.11 Renaming a Table**

Use this statement to rename the table log\_messages to logmsgs:

**ALTER TABLE** log\_messages **RENAME TO** logmsgs;

**5.13.1.12 Adding, Modifying, and Dropping a Table Partition**

As we saw previously, ALTER TABLE table ADD PARTITION … is used to add a new partition to a table.

**ALTER TABLE** log\_messages **ADD** IF **NOT EXISTS**

PARTITION (**year** = 2011, **month** = 1, **day** = 1) **LOCATION** '/logs/2011/01/01'

PARTITION (**year** = 2011, **month** = 1, **day** = 2) **LOCATION** '/logs/2011/01/02'

PARTITION (**year** = 2011, **month** = 1, **day** = 3) **LOCATION** '/logs/2011/01/03'

...;

We can change a partition location, effectively moving it:

**ALTER TABLE** log\_messages PARTITION(**year** = 2011, **month** = 12, **day** = 2)

**SET LOCATION** 's3n://ourbucket/logs/2011/01/02';

This command does not move the data from the old location, nor does it delete the old data.

Finally, you can drop a partition:

**ALTER TABLE** log\_messages **DROP** IF **EXISTS** PARTITION(**year** = 2011, **month** = 12, **day** = 2);

**5.13.1.13 Changing Columns**

You can rename a column, change its position, type, or comment:

**ALTER TABLE** log\_messages

CHANGE **COLUMN** hms hours\_minutes\_seconds INT

**COMMENT** 'The hours, minutes, and seconds part of the timestamp'

**AFTER** severity;

**5.13.1.14 Adding Columns**

You can add new columns to the end of the existing columns, before any partition columns.

**ALTER TABLE** log\_messages **ADD** COLUMNS (

app\_name STRING **COMMENT** 'Application name',

session\_id LONG **COMMENT** 'The current session id');

**5.13.1.15 Deleting or Replacing Columns**

The following example removes all the existing columns and replaces them with the new columns specified:

**ALTER TABLE** log\_messages **REPLACE** COLUMNS (

hours\_mins\_secs INT **COMMENT** 'hour, minute, seconds from timestamp',

severity STRING **COMMENT** 'The message severity'

message STRING **COMMENT** 'The rest of the message');

This statement effectively renames the original hms column and removes the server and process\_id columns from the original schema definition. As for all ALTER statements, only the table metadata is changed.

**5.13.1.16 Alter Table Properties**

You can add additional table properties or modify existing properties, but not remove them:

**ALTER TABLE** log\_messages **SET** TBLPROPERTIES (

'notes' = 'The process id is no longer captured; this column is always NULL');

**5.13.1.17 Alter Storage Properties**

There are several ALTER TABLE statements for modifying format and SerDe properties.

**ALTER TABLE** log\_messages

PARTITION(**year** = 2012, **month** = 1, **day** = 1)

**SET** FILEFORMAT SEQUENCEFILE;

The following example demonstrates how to add new SERDEPROPERTIES for the current

SerDe:

**ALTER TABLE** table\_using\_JSON\_storage

**SET** SERDEPROPERTIES (

'prop3' = 'value3',

'prop4' = 'value4');

**5.14 HiveQL: Data Manipulation**

The Hive query language, focusing on the data manipulation language parts that are used to put data into tables and to extract data from tables to the filesystem.

**5.14.1 Loading Data into Managed Tables**

Hive has no row-level insert, update, and delete operations, the only way to put data into an table is to use one of the “bulk” load operations. Or you can just write files in the correct directories by other means.

**LOAD DATA LOCAL** INPATH '${env:HOME}/california-employees'

OVERWRITE **INTO TABLE** employees

PARTITION (country = 'US', **state** = 'CA');

This command will first create the directory for the partition, if it doesn’t already exist, then copy the data to it. If the target table is not partitioned, you omit the PARTITION clause.

**5.14.2 Inserting Data into Tables from Queries**

The INSERT statement lets you load data into a table from a query.

**INSERT** OVERWRITE **TABLE** employees

PARTITION (country = 'US', **state** = 'OR')

**SELECT** \* **FROM** staged\_employees se

**WHERE** se.cnty = 'US' **AND** se.st = 'OR';

With OVERWRITE, any previous contents of the partition (or whole table if not partitioned)are replaced. If you drop the keyword OVERWRITE or replace it with INTO, Hive appends the data rather than replaces it.

**5.14.3 Dynamic Partition Inserts**

Hive also supports a dynamic partition feature, where it can infer the partitions to create based on query parameters. By comparison, up until now we have considered only static partitions.

**INSERT** OVERWRITE **TABLE** employees

PARTITION (country, **state**)

**SELECT** ..., se.cnty, se.st

**FROM** staged\_employees se;

Hive determines the values of the partition keys, country and state, from the last two columns in the SELECT clause.

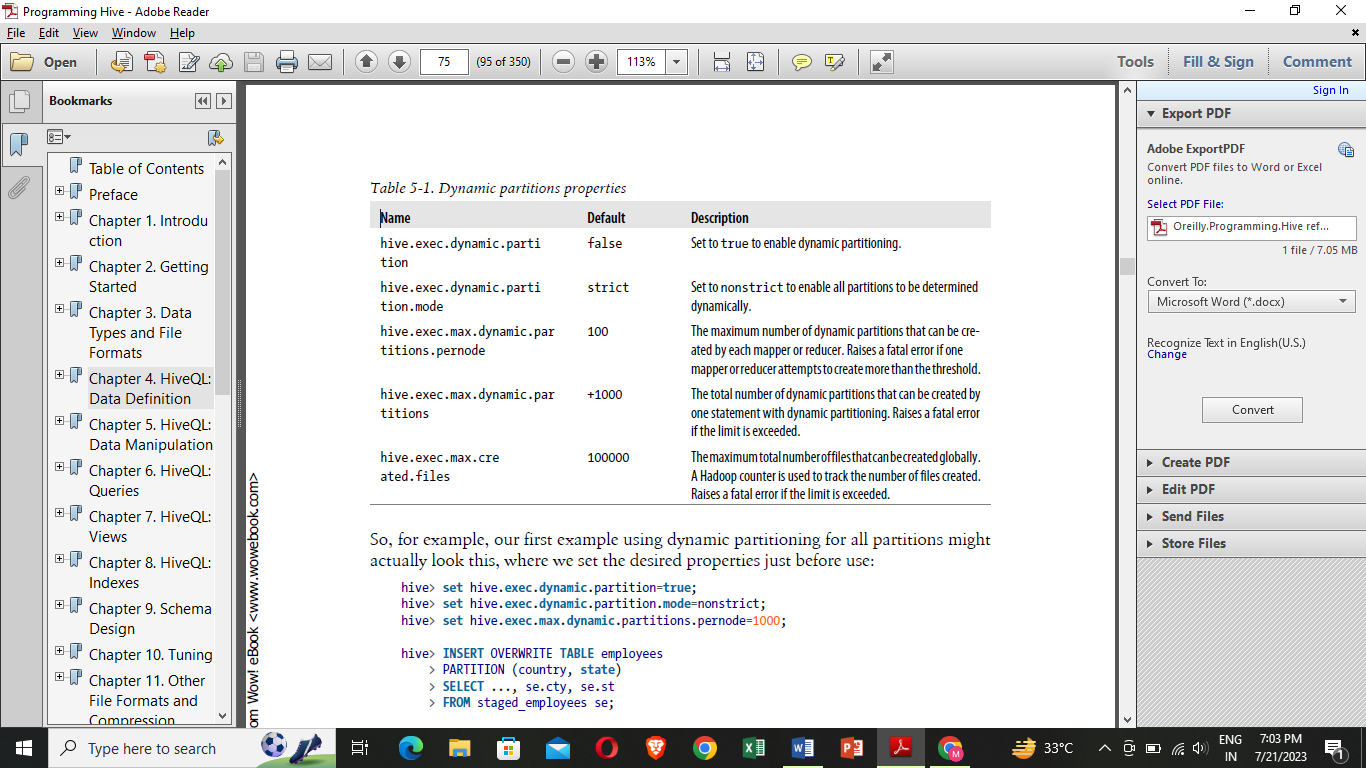


Fig 5.14.1 Dynamic partitions properties

**5.14.4 Creating Tables and Loading Them in One Query**

You can also create a table and insert query results into it in one statement:

**CREATE TABLE** ca\_employees

**AS SELECT** name, salary, address

**FROM** employees

**WHERE** se.**state** = 'CA';

This table contains just the name, salary, and address columns from the employee table records for employees in California. The schema for the new table is taken from the SELECT clause.

**5.14.5 Exporting Data**

If the data files are already formatted the way youwant, then it’s simple enough to copy the directories or files: hadoop fs -cp source\_path target\_path Otherwise, you can use INSERT … DIRECTORY …, as in this example:

**INSERT** OVERWRITE **LOCAL** DIRECTORY '/tmp/ca\_employees'

**SELECT** name, salary, address

**FROM** employees

**WHERE** se.**state** = 'CA';

OVERWRITE and LOCAL have the same interpretations as before and paths are interpreted following the usual rules. One or more files will be written to /tmp/ca\_employees, depending on the number of reducers invoked.

The specified path can also be a full URI (e.g., hdfs://master server/tmp/ca\_employees). Independent of how the data is actually stored in the source table, it is written to files with all fields serialized as strings. Hive uses the same encoding in the generated output files as it uses for the tables internal storage. Just like inserting data to tables, you can specify multiple inserts to directories:

**FROM** staged\_employees se

**INSERT** OVERWRITE DIRECTORY '/tmp/or\_employees'

**SELECT** \* **WHERE** se.cty = 'US' **and** se.st = 'OR'

**INSERT** OVERWRITE DIRECTORY '/tmp/ca\_employees'

**SELECT** \* **WHERE** se.cty = 'US' **and** se.st = 'CA'

**INSERT** OVERWRITE DIRECTORY '/tmp/il\_employees'

**SELECT** \* **WHERE** se.cty = 'US' **and** se.st = 'IL';

There are some limited options for customizing the output of the data (other than writing a custom OUTPUTFORMAT.

**5.15 HiveQL: Queries**

**5.15.1 SELECT … FROM Clauses**

SELECT is the projection operator in SQL. The FROM clause identifies from which table,view, or nested query we select records. For a given record, SELECT specifies the columns to keep, as well as the outputs of function calls on one or more columns (e.g., the aggregation functions like count(\*)).

hive> **SELECT** name, salary **FROM** employees;

John Doe 100000.0

Mary Smith 80000.0

hive> **SELECT** name, salary **FROM** employees;

hive> **SELECT** e.name, e.salary **FROM** employees e;

First, let’s select the subordinates, an

ARRAY, where a comma-separated list surrounded with […] is used.

hive> **SELECT** name, subordinates **FROM** employees;

John Doe ["Mary Smith","Todd Jones"]

Mary Smith ["Bill King"]

Todd Jones []

Bill King []

The deductions is a MAP, where the JSON representation for maps is used, namely a comma-separated list of key:value pairs, surrounded with {...}:

hive> **SELECT** name, deductions **FROM** employees;

John Doe {"Federal Taxes":0.2,"State Taxes":0.05,"Insurance":0.1}

Mary Smith {"Federal Taxes":0.2,"State Taxes":0.05,"Insurance":0.1}

Todd Jones {"Federal Taxes":0.15,"State Taxes":0.03,"Insurance":0.1}

Bill King {"Federal Taxes":0.15,"State Taxes":0.03,"Insurance":0.1}

First, ARRAY indexing is 0-based, as in Java. Here is a query that selects the first element of the subordinates array:

hive> **SELECT** name, subordinates[0] **FROM** employees;

John Doe Mary Smith

Mary Smith Bill King

Todd Jones **NULL**

Bill King **NULL**

Note that referencing a nonexistent element returns NULL. To reference a MAP element, you also use ARRAY[...] syntax, but with key values instead of integer indices:

hive> **SELECT** name, deductions["State Taxes"] **FROM** employees;

John Doe 0.05

Finally, to reference an element in a STRUCT, you use “dot” notation, similar to the table\_alias.column mentioned above:

hive> **SELECT** name, address.city **FROM** employees;

John Doe Chicago

Mary Smith Chicago

Todd Jones Oak Park

Bill King Obscuria

**5.15.2 Specify Columns with Regular Expressions**

We can even use regular expressions to select the columns we want. The following query selects the symbol column and all columns from stocks whose names start with the

prefix price:1

hive> **SELECT** symbol, `price.\*` **FROM** stocks;

AAPL 195.69 197.88 194.0 194.12 194.12

AAPL 192.63 196.0 190.85 195.46 195.46

AAPL 196.73 198.37 191.57 192.05 192.05

AAPL 195.17 200.2 194.42 199.23 199.23

AAPL 195.91 196.32 193.38 195.86 195.86

...

**5.15.3 Computing with Column Values**

you can manipulate column values using function calls and arithmetic expressions. We could call a built-in function map\_values to extract all the values from the deductions map and then add them up with the built-in sum function. The following query is long enough that we’ll split it over two lines. Note the secondary prompt that Hive uses, an indented greater-than sign (>):

hive> **SELECT upper**(name), salary, deductions["Federal Taxes"],

> round(salary \* (1 - deductions["Federal Taxes"])) **FROM** employees;

**5.15.4 Arithmetic Operators**

All the typical arithmetic operators are supported. Arithmetic operators take any numeric type. No type coercion is performed if the two operands are of the same numeric type. Otherwise, if the types differ, then the value of the smaller of the two types is promoted to wider type of the other value.

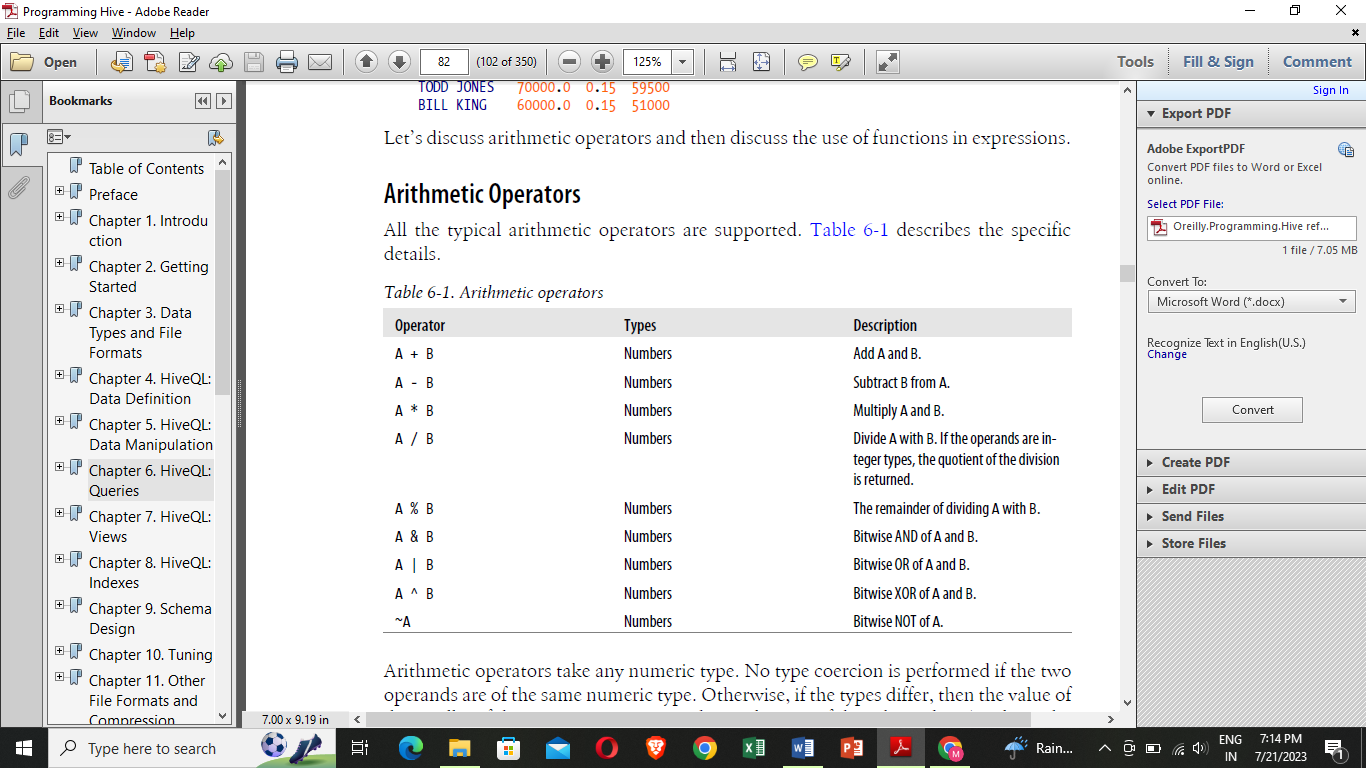
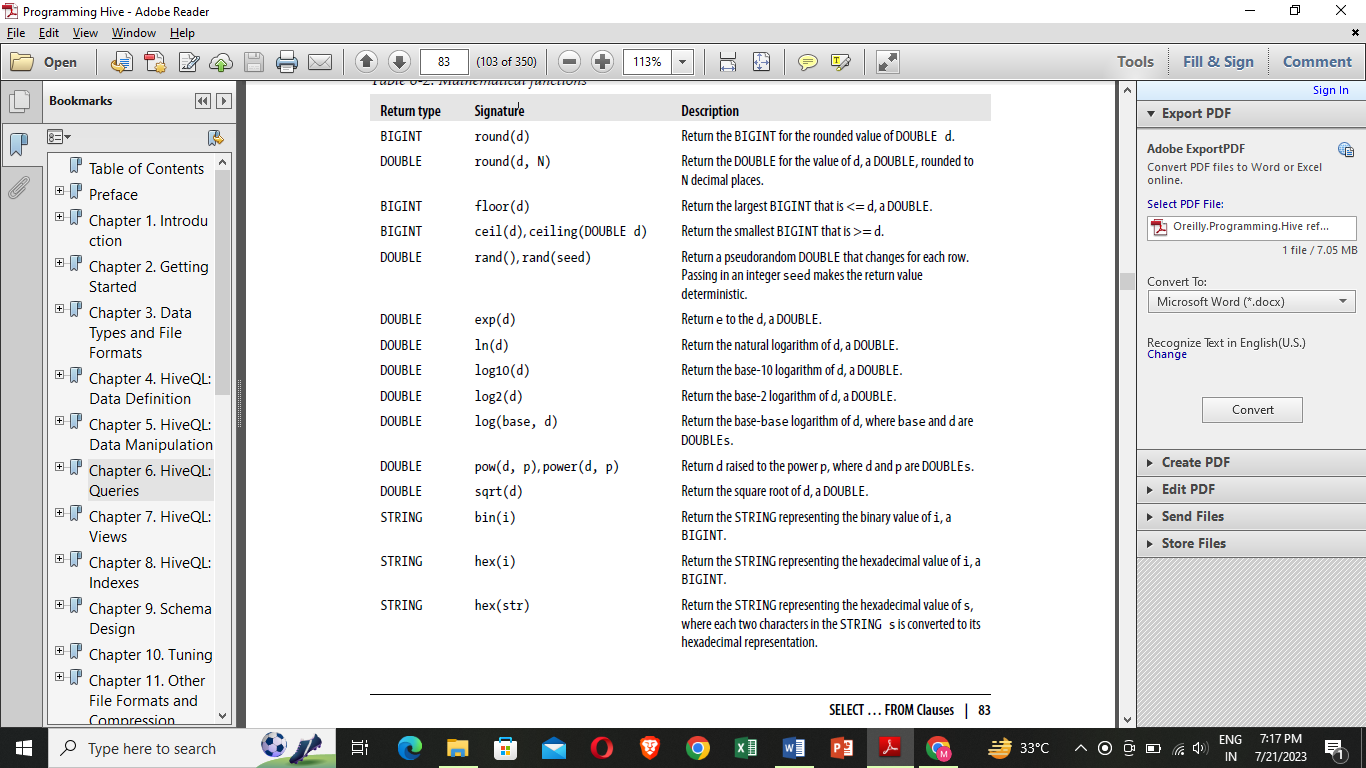


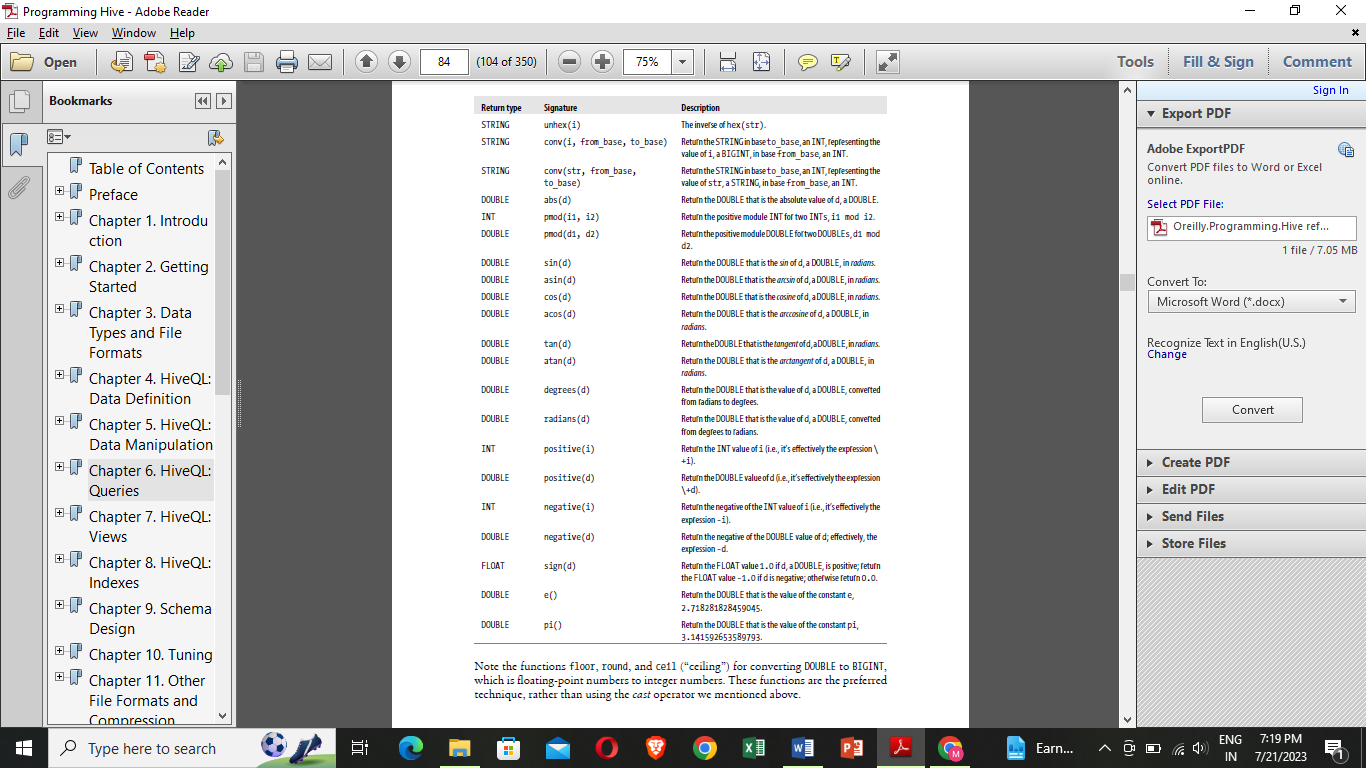
Fig 5.15.1Arithmetic operators

**5.15.5 Using Functions**

Our tax-deduction example also uses a built-in mathematical function, round(), for finding the nearest integer for a DOUBLE value.

**5.15.6 Mathematical functions**



Fig 5.15.2 Mathematical functions

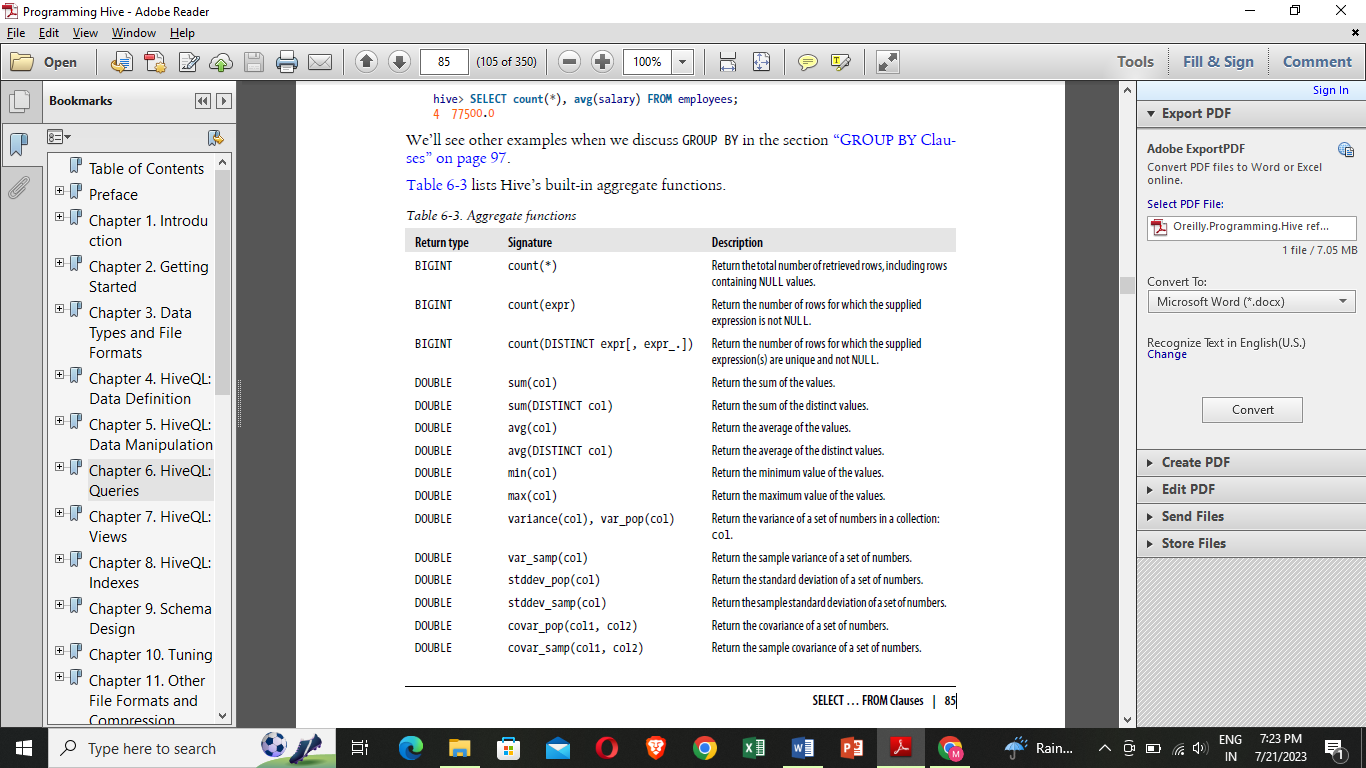
Note the functions floor, round, and ceil (“ceiling”) for converting DOUBLE to BIGINT, which is floating-point numbers to integer numbers. These functions are the preferred technique, rather than using the cast operator we mentioned above.

**5.15.7 Aggregate functions**

A special kind of function is the aggregate function that returns a single value resulting from some computation over many rows. Perhaps the two best known examples are count, which counts the number of rows (or values for a specific column), and avg, which returns the average value of the specified column values. Here is a query that counts the number of our example employees and averages their salaries:

hive> **SELECT count**(\*), **avg**(salary) **FROM** employees;

4 77500.0



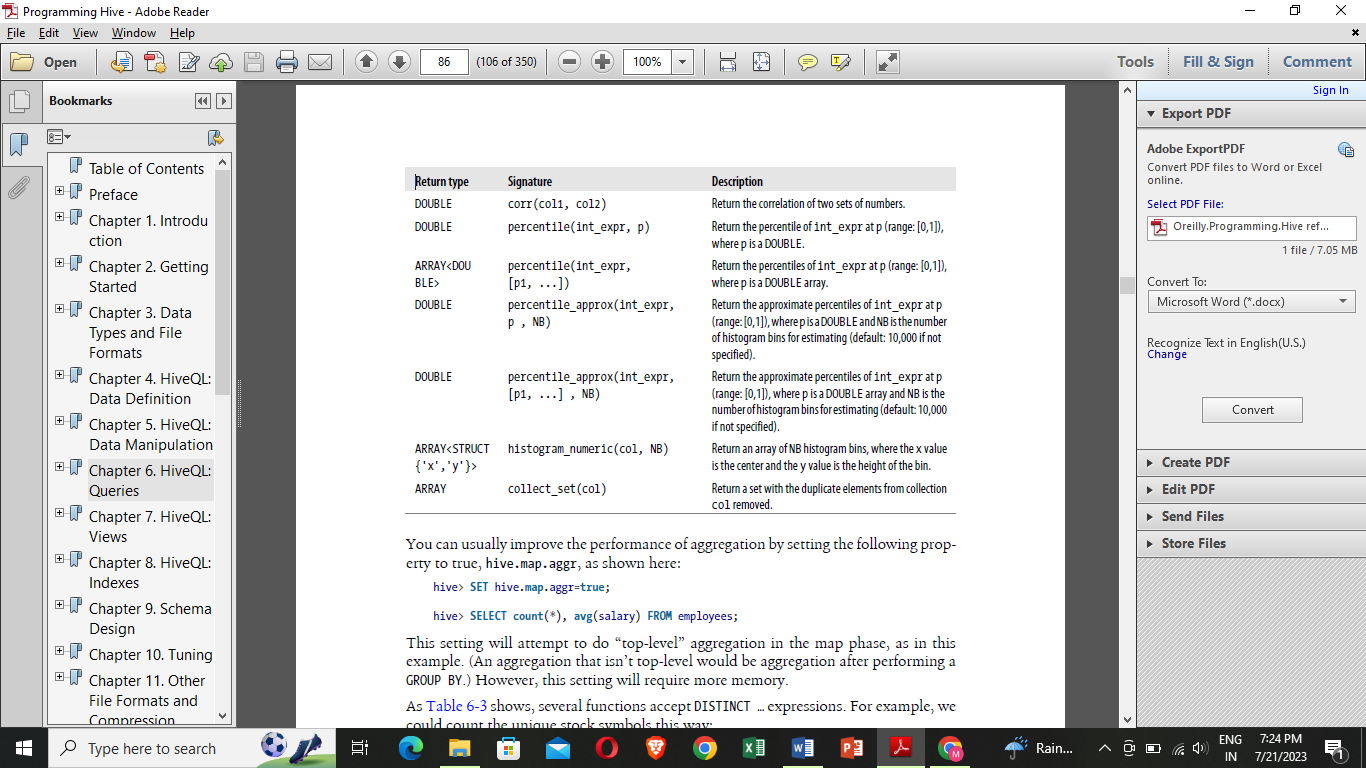


Fig 5.15.3 Aggregate functions

usually improve the performance of aggregation by setting the following property to true, hive.map.aggr, as shown here:

hive> **SET** hive.**map**.aggr=**true**;

hive> **SELECT count**(\*), **avg**(salary) **FROM** employees;

**5.15.8 Table generating functions**

The “inverse” of aggregate functions are so-called table generating functions, which takesingle columns and expand them to multiple columns or rows.To explain by way of an example, the following query converts the subordinate arrayin each employees record into zero or more new records. If an employee record has an empty subordinates array, then no new records are generated. Otherwise, one new

record per subordinate is generated:

hive> **SELECT** explode(subordinates) **AS** sub **FROM** employees;

Mary Smith

Todd Jones

Bill King

We used a column alias, sub, defined using the AS sub clause. When using table generating functions, column aliases are required by Hive.

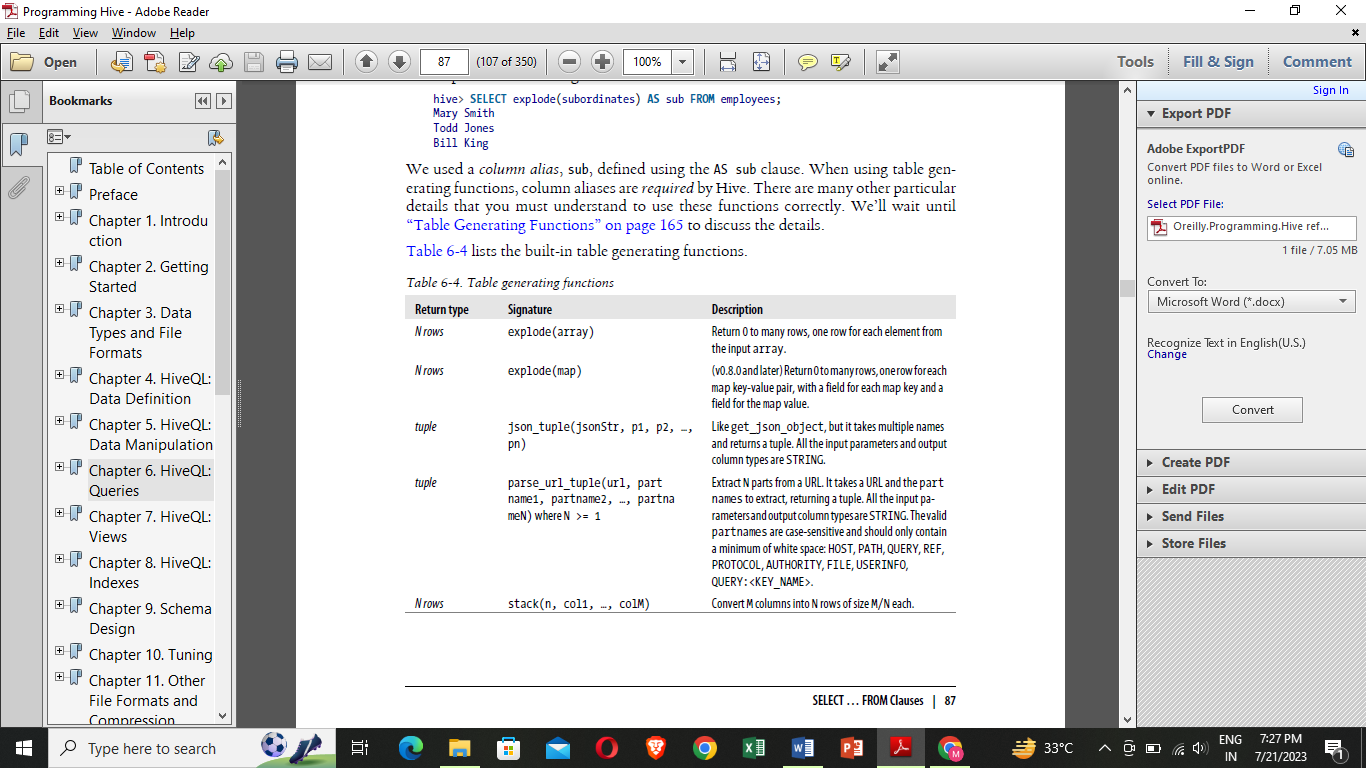


Fig 5.15.4 Table generated functions

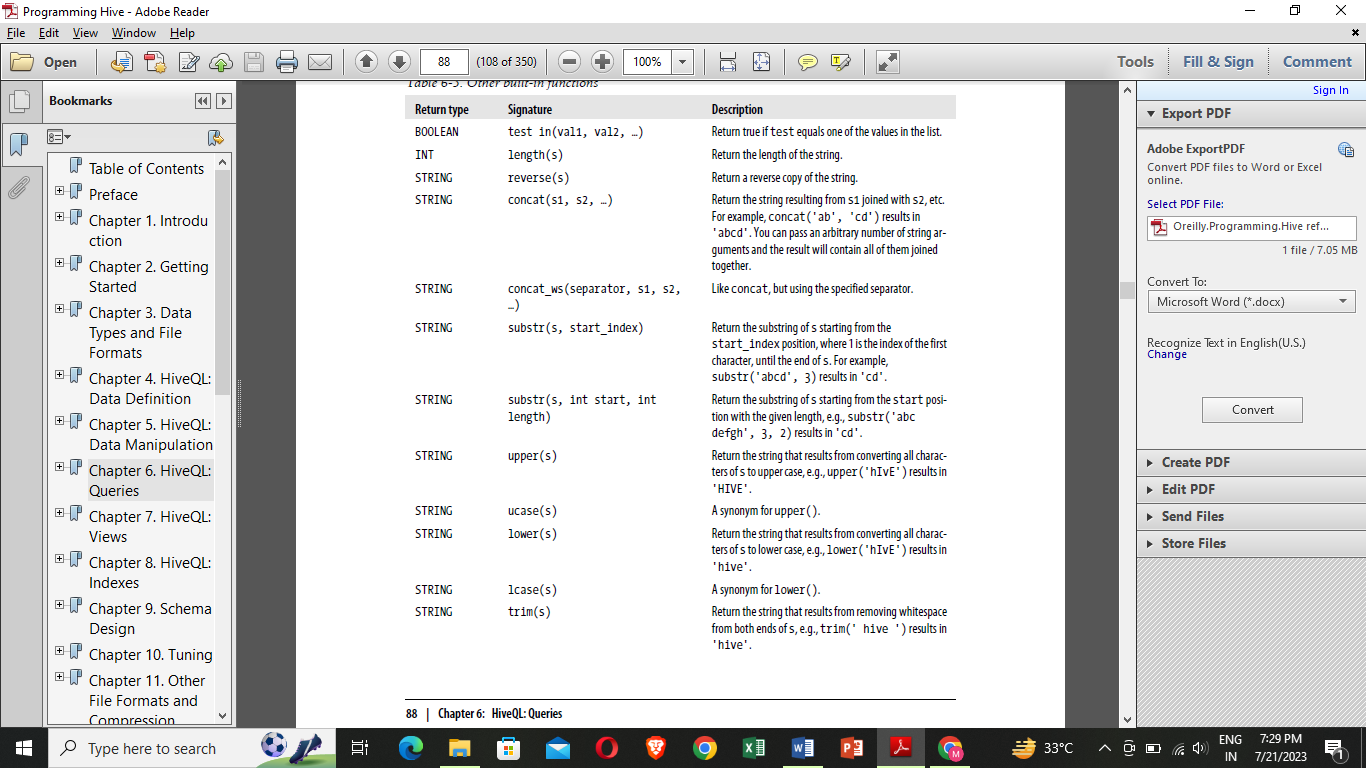
Here is an example that uses parse\_url\_tuple where we assume a url\_table exists that contains a column of URLs called url:

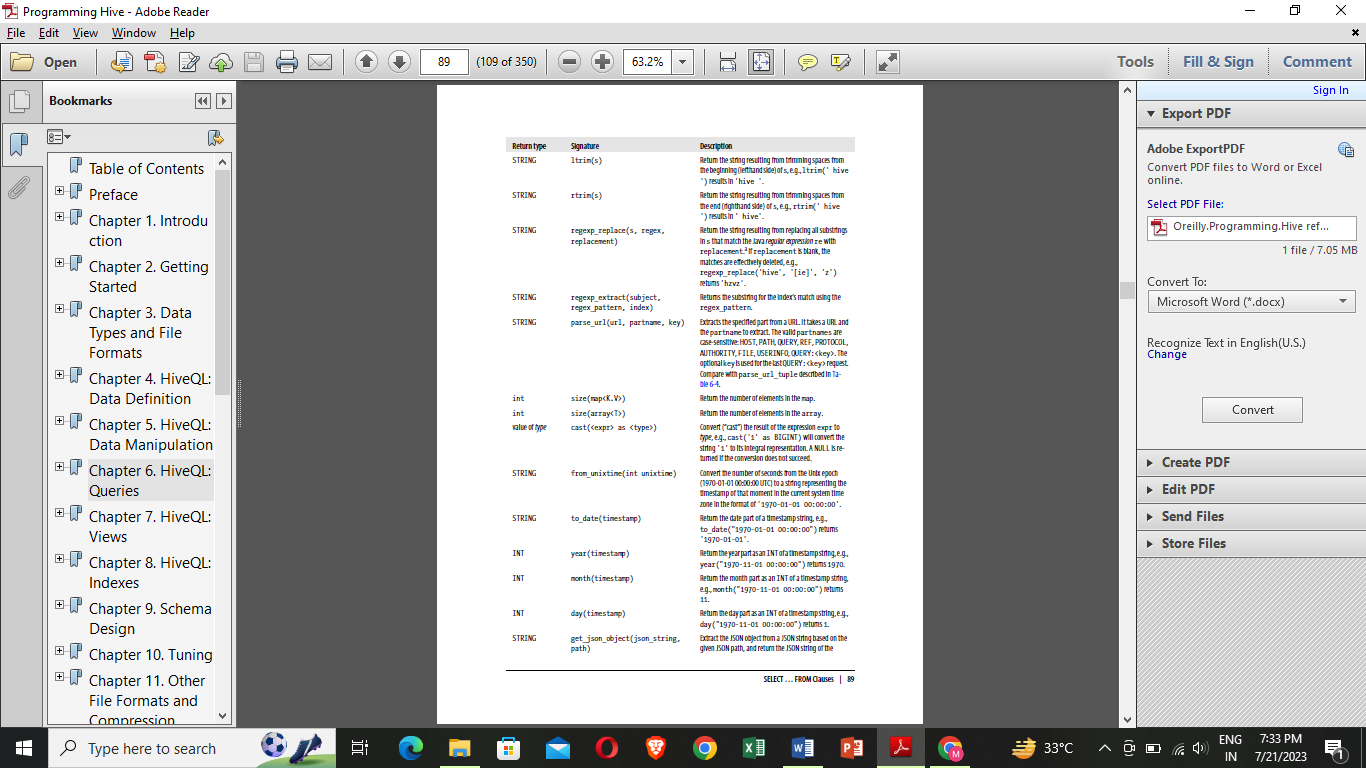
**SELECT** parse\_url\_tuple(url, 'HOST', 'PATH', 'QUERY') **as** (**host**, path, query)

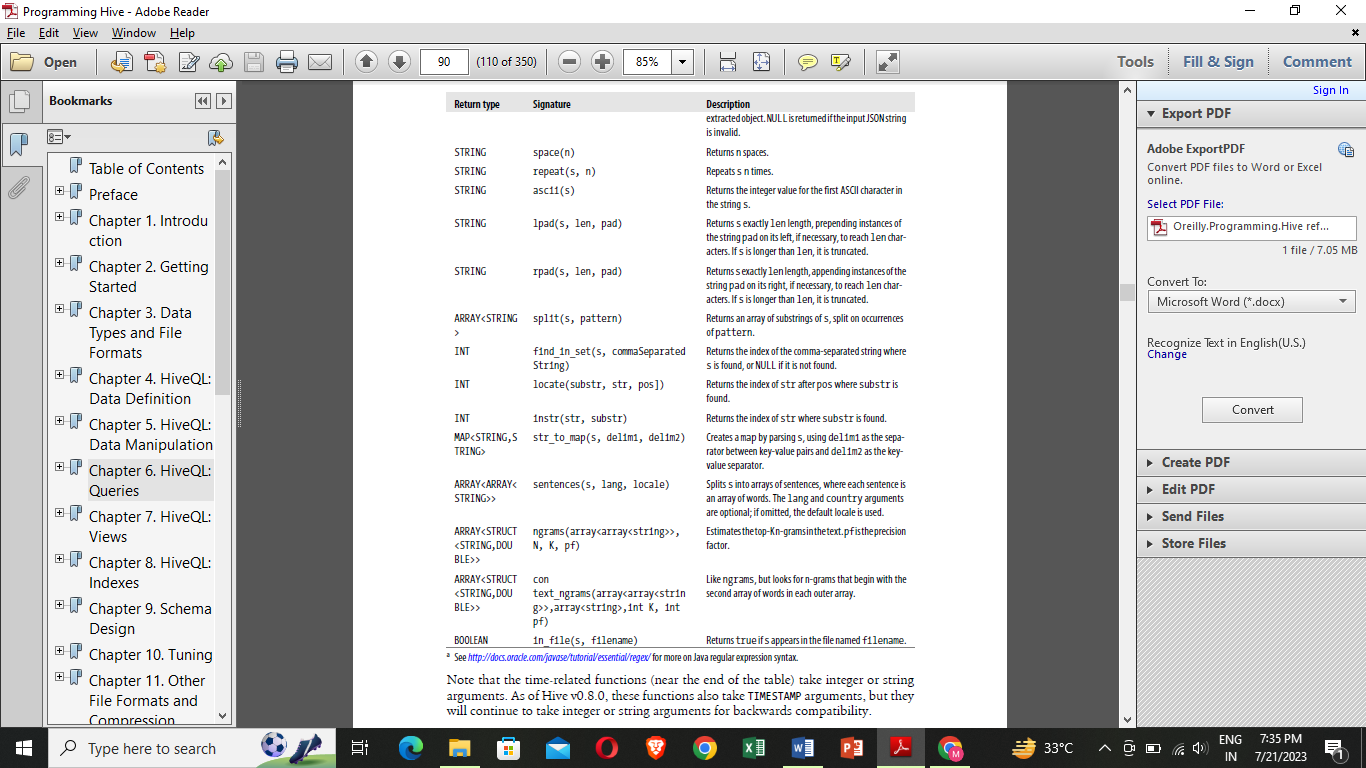
**FROM** url\_table;

**5.15.9 Other built-in functions**

The rest of the built-in functions for working with strings, maps, arrays, JSON, and timestamps, with or without the recently introduced TIMESTAMP type





Fig 5.15.5 Other Built-in function

**5.15.10 LIMIT Clause**

The results of a typical query can return a large number of rows. The LIMIT clause putsan upper limit on the number of rows returned:

hive> **SELECT upper**(name), salary, deductions["Federal Taxes"],

> round(salary \* (1 - deductions["Federal Taxes"])) **FROM** employees

> **LIMIT** 2;

JOHN DOE 100000.0 0.2 80000

MARY SMITH 80000.0 0.2 64000

**5.15.11 Column Aliases**

Query as returning a new relation with new columns, some of which are anonymous results of manipulating columns in employees. It’s sometimes useful to give those anonymous columns a name, called a column alias.

Hive> **SELECT upper**(name), salary, deductions["Federal Taxes"] **as** fed\_taxes,

> round(salary \* (1 - deductions["Federal Taxes"])) **as** salary\_minus\_fed\_taxes

> **FROM** employees **LIMIT** 2;

JOHN DOE 100000.0 0.2 80000

MARY SMITH 80000.0 0.2 64000

**5.15.12 Nested SELECT Statements**

The column alias feature is especially useful in nested select statements. Let’s use the previous example as a nested query:

hive> **FROM** (

> **SELECT upper**(name), salary, deductions["Federal Taxes"] **as** fed\_taxes,

> round(salary \* (1 - deductions["Federal Taxes"])) **as** salary\_minus\_fed\_taxes

> **FROM** employees

> ) e

> **SELECT** e.name, e.salary\_minus\_fed\_taxes

> **WHERE** e.salary\_minus\_fed\_taxes > 70000;

JOHN DOE 100000.0 0.2 80000

**5.15.13 CASE … WHEN … THEN Statements**

The CASE … WHEN … THEN clauses are like if statements for individual columns in query results. For example:

hive> **SELECT** name, salary,

> **CASE**

> **WHEN** salary < 50000.0 **THEN** 'low'

> **WHEN** salary >= 50000.0 **AND** salary < 70000.0 **THEN** 'middle'

> **WHEN** salary >= 70000.0 **AND** salary < 100000.0 **THEN** 'high'

> **ELSE** 'very high'

> **END AS** bracket **FROM** employees;

John Doe 100000.0 very high

Mary Smith 80000.0 high

Todd Jones 70000.0 high

Bill King 60000.0 middle

Boss Man 200000.0 very high

Fred Finance 150000.0 very high

Stacy Accountant 60000.0 middle

...

**5.15.14 When Hive Can Avoid MapReduce**

Hive implements some kinds of queries without using MapReduce, in so-called local mode, for example:

**SELECT** \* **FROM** employees;

In this case, Hive can simply read the records from employees and dump the formatted output to the console. This even works for WHERE clauses that only filter on partition keys, with or without LIMIT clauses:

**SELECT** \* **FROM** employees

**WHERE** country = 'US' **AND state** = 'CA'

**LIMIT** 100;

Furthermore, Hive will attempt to run other operations in local mode if the hive.exec.mode.local.auto property is set to true:

**set** hive.**exec**.**mode**.**local**.auto=**true**;

Otherwise, Hive uses MapReduce to run all other queries.

**5.15.15 WHERE Clauses**

While SELECT clauses select columns, WHERE clauses are filters; they select which records to return.

**SELECT** \* **FROM** employees

**WHERE** country = 'US' **AND state** = 'CA';

The following variation eliminates the duplication, using a column alias, but unfortunately it’s not valid:

hive> **SELECT** name, salary, deductions["Federal Taxes"],

> salary \* (1 - deductions["Federal Taxes"]) **as** salary\_minus\_fed\_taxes

> **FROM** employees

> **WHERE** round(salary\_minus\_fed\_taxes) > 70000;

FAILED: Error **in** semantic analysis: Line 4:13 Invalid **table alias or column** reference 'salary\_minus\_fed\_taxes': (possible **column names are**:name, salary, subordinates, deductions, address)

**5.15.16 Predicate Operators**

The predicate operators, which are also used in JOIN … ON and HAVING clauses.

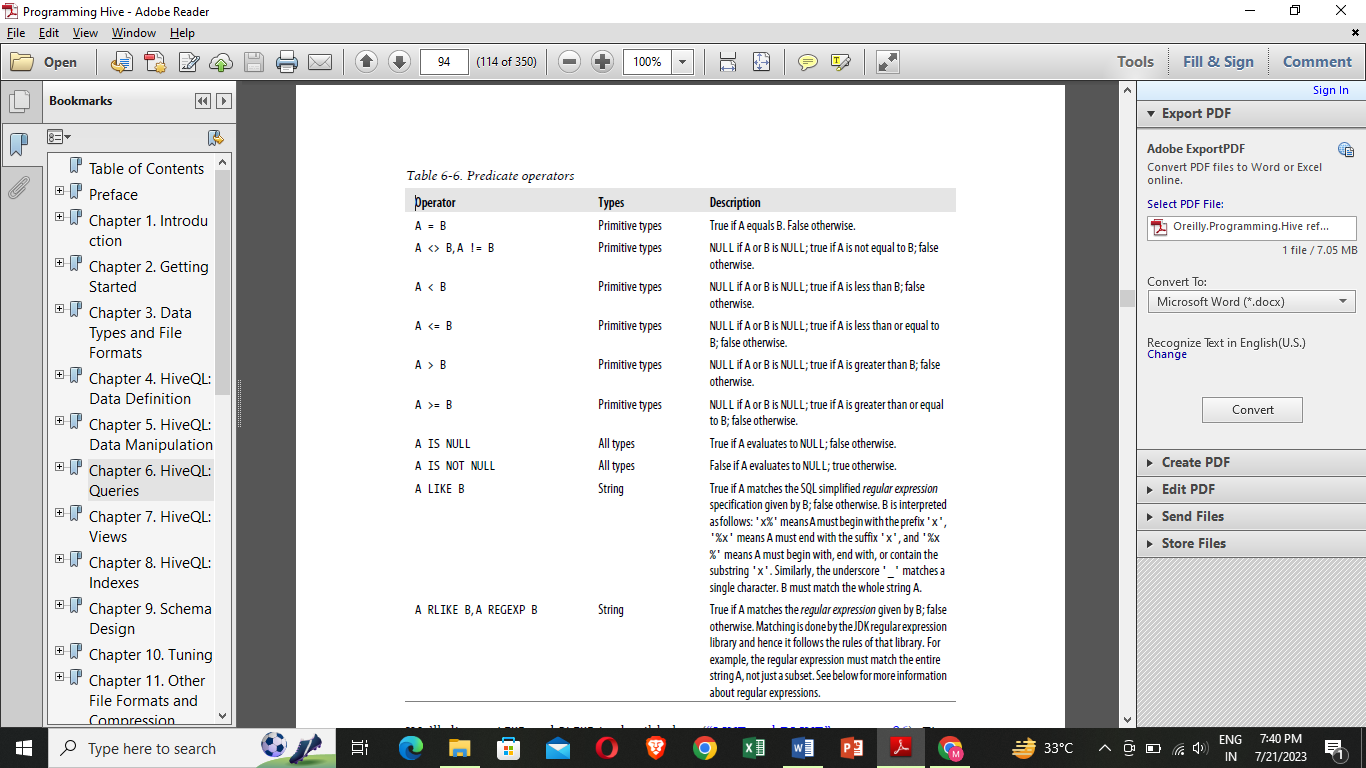


Fig 5.15.6 Predicate operators

**5.15.17 Gotchas with Floating-Point Comparisons**

A common gotcha arises when you compare floating-point numbers of different types.

hive> **SELECT** name, salary, deductions['Federal Taxes']

> **FROM** employees **WHERE** deductions['Federal Taxes'] > 0.2;

John Doe 100000.0 0.2

Mary Smith 80000.0 0.2

**5.15.18 LIKE and RLIKE**

It lets us match on strings that begin with or endwith a particular substring, or when the substring appears anywhere within the string.

hive> **SELECT** name, address.street **FROM** employees **WHERE** address.street **LIKE** '%Ave.';

John Doe 1 Michigan Ave.

Todd Jones 200 Chicago Ave.

hive> **SELECT** name, address.city **FROM** employees **WHERE** address.city **LIKE** 'O%';

Todd Jones Oak Park

Bill King Obscuria

hive> **SELECT** name, address.street **FROM** employees **WHERE** address.street **LIKE** '%Chi%';

Todd Jones 200 Chicago Ave.

A Hive extension is the RLIKE clause, which lets us use Java regular expressions, a more powerful minilanguage for specifying matches.

hive> **SELECT** name, address.street

> **FROM** employees **WHERE** address.street RLIKE '.\*(Chicago|Ontario).\*';

Mary Smith 100 Ontario St.

Todd Jones 200 Chicago Ave.

**5.15.19 GROUP BY Clauses**

The GROUP BY statement is often used in conjunction with aggregate functions to group the result set by one or more columns and then perform an aggregation over each group.

hive> **SELECT year**(ymd), **avg**(price\_close) **FROM** stocks

> **WHERE** exchange = 'NASDAQ' **AND** symbol = 'AAPL'

> **GROUP BY year**(ymd);

1984 25.578625440597534

1985 20.193676221040867

1986 32.46102808021274

1987 53.88968399108163

**5.15.20 HAVING Clauses**

The HAVING clause lets you constrain the groups produced by GROUP BY in a way that could be expressed with a subquery, using a syntax that’s easier to express.

hive> **SELECT year**(ymd), **avg**(price\_close) **FROM** stocks

> **WHERE** exchange = 'NASDAQ' **AND** symbol = 'AAPL'

> **GROUP BY year**(ymd)

> **HAVING avg**(price\_close) > 50.0;

1987 53.88968399108163

1991 52.49553383386182

**5.15.21 JOIN Statements**

Hive supports the classic SQL JOIN statement, but only equi-joins are supported.

**5.15.21.1Inner JOIN**

In an inner JOIN, records are discarded unless join criteria finds matching records in every table being joined

hive> **SELECT** a.ymd, a.price\_close, b.price\_close

> **FROM** stocks a **JOIN** stocks b **ON** a.ymd = b.ymd

> **WHERE** a.symbol = 'AAPL' **AND** b.symbol = 'IBM';

2010-01-04 214.01 132.45

2010-01-05 214.38 130.85

Here is an inner JOIN between stocks and dividends for Apple, where we use the ymd and symbol columns as join keys:

hive> **SELECT** s.ymd, s.symbol, s.price\_close, d.dividend

> **FROM** stocks s **JOIN** dividends d **ON** s.ymd = d.ymd **AND** s.symbol = d.symbol

> **WHERE** s.symbol = 'AAPL';

1987-05-11 AAPL 77.0 0.015

1987-08-10 AAPL 48.25 0.015

**5.15.21.2 Join Optimizations**

In the previous example, every ON clause uses a.ymd as one of the join keys. In this case,Hive can apply an optimization where it joins all three tables in a single MapReduce job. Hive also assumes that the last table in the query is the largest. It attempts to buffer theother tables and then stream the last table through, while performing joins on individual records.

**5.15.21.3 LEFT OUTER JOIN**

The left-outer join is indicated by adding the LEFT OUTER keywords:

hive> **SELECT** s.ymd, s.symbol, s.price\_close, d.dividend

> **FROM** stocks s **LEFT OUTER JOIN** dividends d **ON** s.ymd = d.ymd **AND** s.symbol = d.symbol

> **WHERE** s.symbol = 'AAPL';

...

1987-05-01 AAPL 80.0 **NULL**

1987-05-04 AAPL 79.75 **NULL**

1987-05-05 AAPL 80.25 **NULL**

1987-05-06 AAPL 80.0 **NULL**

**5.15.21.4 OUTER JOIN Gotcha**

hive> **SELECT** s.ymd, s.symbol, s.price\_close, d.dividend

> **FROM** stocks s **LEFT OUTER JOIN** dividends d **ON** s.ymd = d.ymd **AND** s.symbol = d.symbol

> **WHERE** s.symbol = 'AAPL'

> **AND** s.exchange = 'NASDAQ' **AND** d.exchange = 'NASDAQ';

1987-05-11 AAPL 77.0 0.015

1987-08-10 AAPL 48.25 0.015

**5.15.21.5 RIGHT OUTER JOIN**

Right-outer joins return all records in the righthand table that match the WHERE clause. NULL is used for fields of missing records in the lefthand table. Here we switch the places of stocks and dividends and perform a righthand join, but leave the SELECT statement unchanged:

hive> **SELECT** s.ymd, s.symbol, s.price\_close, d.dividend

> **FROM** dividends d **RIGHT OUTER JOIN** stocks s **ON** d.ymd = s.ymd **AND** d.symbol = s.symbol

> **WHERE** s.symbol = 'AAPL';

...

1987-05-07 AAPL 80.25 **NULL**

1987-05-08 AAPL 79.0 **NULL**

1987-05-11 AAPL 77.0 0.015

**5.15.21.6 FULL OUTER JOIN**

Finally, a full-outer join returns all records from all tables that match the WHERE clause. NULL is used for fields in missing records in either table.

hive> **SELECT** s.ymd, s.symbol, s.price\_close, d.dividend

> **FROM** dividends d **FULL OUTER JOIN** stocks s **ON** d.ymd = s.ymd **AND** d.symbol = s.symbol

> **WHERE** s.symbol = 'AAPL';

...

1987-05-07 AAPL 80.25 **NULL**

1987-05-08 AAPL 79.0 **NULL**

**5.15.21.7 LEFT SEMI-JOIN**

A left semi-join returns records from the lefthand table if records are found in the right hand table that satisfy the ON predicates.

hive> **SELECT** s.ymd, s.symbol, s.price\_close

> **FROM** stocks s **LEFT** SEMI **JOIN** dividends d **ON** s.ymd = d.ymd **AND** s.symbol = d.symbol;

...

1962-11-05 IBM 361.5

1962-08-07 IBM 373.25

**5.15.21.8 Cartesian Product JOINs**

A Cartesian product is a join where all the tuples in the left side of the join are paired with all the tuples of the right table. If the left table has 5 rows and the right table has 6 rows, 30 rows of output will be produced:

SELECTS \* **FROM** stocks **JOIN** dividends;

**5.15.21.9 Map-side Joins**

If all but one table is small, the largest table can be streamed through the mappers while the small tables are cached in memory. Hive can do all the joining map-side, since it can look up every possible match against the small tables in memory, thereby eliminating the reduce step required in the more common join scenarios

**SELECT** /\*+ MAPJOIN(d) \*/ s.ymd, s.symbol, s.price\_close, d.dividend

**FROM** stocks s **JOIN** dividends d **ON** s.ymd = d.ymd **AND** s.symbol = d.symbol

**WHERE** s.symbol = 'AAPL';

**5.15.22 ORDER BY and SORT BY**

The ORDER BY clause is familiar from other SQL dialects. It performs a total ordering of the query result set. This means that all the data is passed through a single reducer, which may take an unacceptably long time to execute for larger data sets. Hive adds an alternative, SORT BY, that orders the data only within each reducer, thereby performing a local ordering, where each reducer’s output will be sorted.

Here is an example using ORDER BY:

**SELECT** s.ymd, s.symbol, s.price\_close

**FROM** stocks s

**ORDER BY** s.ymd **ASC**, s.symbol **DESC**;

Here is the same example using SORT BY instead:

**SELECT** s.ymd, s.symbol, s.price\_close

**FROM** stocks s

SORT **BY** s.ymd **ASC**, s.symbol **DESC**;

**5.15.22 DISTRIBUTE BY with SORT BY**

DISTRIBUTE BY controls how map output is divided among reducers. All data that flows through a MapReduce job is organized into key-value pairs.

hive> **SELECT** s.ymd, s.symbol, s.price\_close

> **FROM** stocks s

> DISTRIBUTE **BY** s.symbol

> SORT **BY** s.symbol **ASC**, s.ymd **ASC**;

1984-09-07 AAPL 26.5

1984-09-10 AAPL 26.37

**5.15.23 CLUSTER BY**

The same columns are used in both clauses and all columns are sorted by ascending order (the default). In this case, the CLUSTER BY clause is a shor-hand way of expressing the same query.

hive> **SELECT** s.ymd, s.symbol, s.price\_close

> **FROM** stocks s

> **CLUSTER BY** s.symbol;

2010-02-08 AAPL 194.12

2010-02-05 AAPL 195.46

hive> **SELECT** s.ymd, s.symbol, s.price\_close

> **FROM** stocks s

> **CLUSTER BY** s.symbol;

2010-02-08 AAPL 194.12

2010-02-05 AAPL 195.46

**5.15.24Casting**

Hive will perform some implicit conversions, called casts, of numeric data types, as needed. For example, when doing comparisons between two numbers of different types. cast() function that allows you to explicitly convert a value of one type to another. The following example casts the values to FLOAT before performing a comparison:

**SELECT** name, salary **FROM** employees

**WHERE cast**(salary **AS** FLOAT) < 100000.0;

**5.15.24.1 Casting BINARY Values**

The new BINARY type introduced in Hive v0.8.0 only supports casting BINARY to STRING.

**SELECT** (2.0\***cast**(**cast**(b **as** string) **as** double)) **from** src;

**5.15.25 Queries that Sample Data**

For very large data sets, sometimes you want to work with a representative sample of a query result, not the whole thing. Hive supports this goal with queries that sample tables organized into buckets.In the following example, assume the numbers table has one number column with values 1−10.We can sample using the rand() function, which returns a random number. In the first two queries, two distinct numbers are returned for each query. In the third query, no results are returned:

hive> **SELECT** \* **from** numbers TABLESAMPLE(BUCKET 3 **OUT OF** 10 **ON** rand()) s;

2

4

**5.15.26 Block Sampling**

Hive offers another syntax for sampling a percentage of blocks of an input path as an

alternative to sampling based on rows:

hive> **SELECT** \* **FROM** numbersflat TABLESAMPLE(0.1 PERCENT) s;

**5.15.27 Input Pruning for Bucket Tables**

From a first look at the TABLESAMPLE syntax, an astute user might come to the conclusion that the following query would be equivalent to the TABLESAMPLE operation:

hive> **CREATE TABLE** numbers\_bucketed (number int) CLUSTERED **BY** (number) **INTO** 2 BUCKETS;

hive> **SET** hive.enforce.bucketing=**true**;

hive> **INSERT** OVERWRITE **TABLE** numbers\_bucketed **SELECT** number **FROM** numbers;

hive> dfs -ls /**user**/hive/warehouse/mydb.db/numbers\_bucketed;

/**user**/hive/warehouse/mydb.db/numbers\_bucketed/000000\_0

/**user**/hive/warehouse/mydb.db/numbers\_bucketed/000001\_0

**5.15.28 UNION ALL**

UNION ALL combines two or more tables. Each subquery of the union query must produce the same number of columns, and for each column, its type must match all the column types in the same position. For example, if the second column is a FLOAT, then the second column of all the other query results must be a FLOAT. Here is an example the merges log data:

**SELECT** log.ymd, log.**level**, log.message

**FROM** (**SELECT** l1.ymd, l1.**level**,

l1.message, 'Log1' **AS source**

**FROM** log1 l1

**UNION ALL**

**SELECT** l2.ymd, l2.**level**,

l2.message, 'Log2' **AS source**

**FROM** log1 l2

) log

SORT **BY** log.ymd **ASC**;