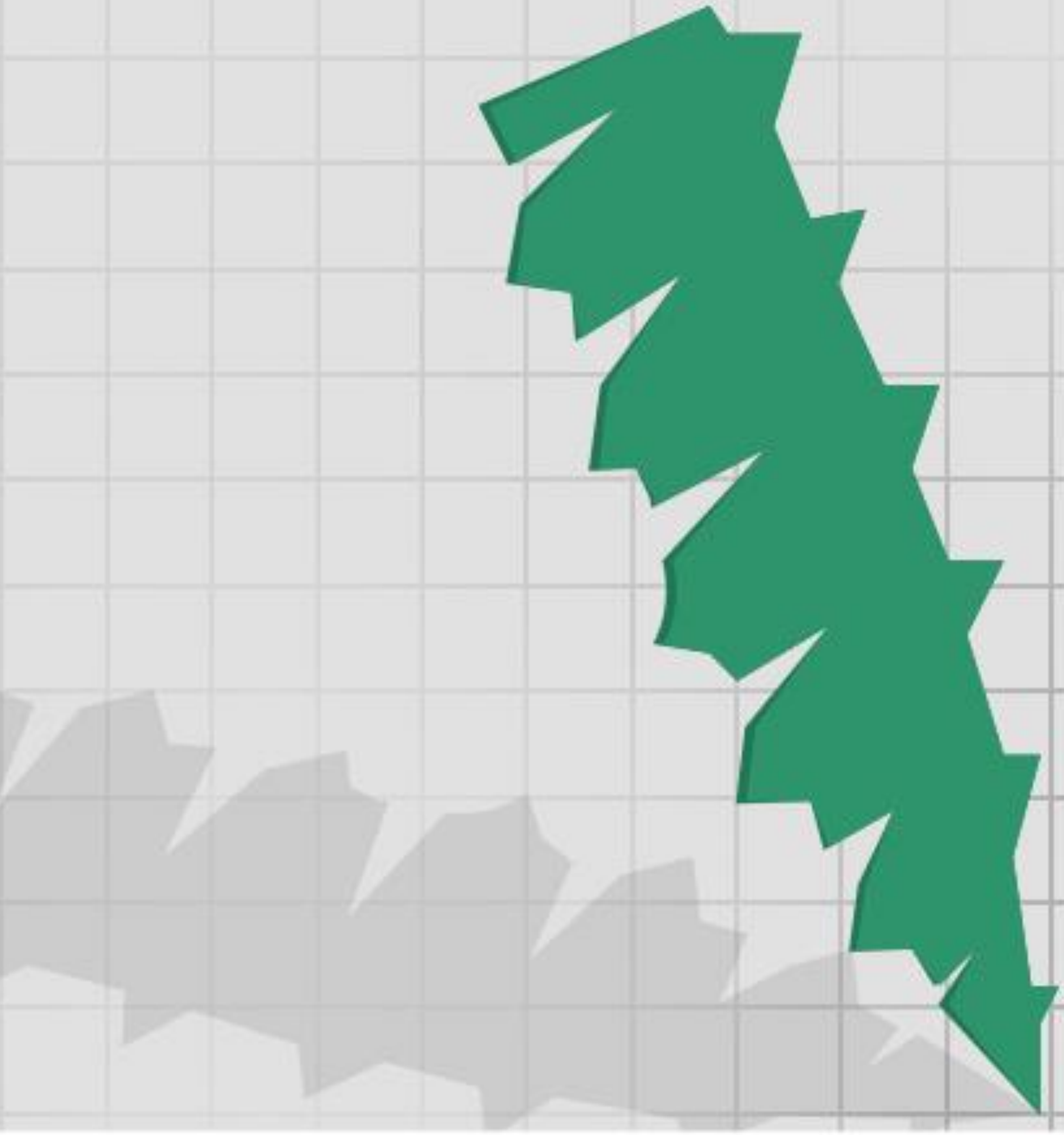
Apache Drill



tutorialspoint

SIMPLY EASY LEARNING

About the Tutorial

Apache Drill is first schema-free SQL engine. Unlike Hive, it does not use MR job internally and compared to most distributed query engines, it does not depend on Hadoop. Apache Drill is observed as the upgraded version of Apache Sqoop. Drill is inspired by Google Dremel concept called BigQuery and later became an open source Apache project.

This tutorial will explore the fundamentals of Drill, setup and then walk through with query operations using JSON, querying data with Big Data technologies and finally conclude with some real-time applications.

Audience

This tutorial has been prepared for professionals aspiring to make a career in Big Data Analytics. This tutorial will give you enough understanding on Drill process, about how to install it on your system and its other operations.

Prerequisites

Before proceeding with this tutorial, you must have a good understanding of Java, JSON and any of the Linux operating system.

Copyright & Disclaimer

Table of Contents

1. Apache Drill – Introduction

In this chapter, we will discuss about the basic overview of Apache Drill, its benefits and key features. Apart from this, we will also get some basic knowledge on Google Dremel.

Overview of Google Dremel/BigQuery

Google manages big data every second of every day to provide services like Search, YouTube, Gmail and Google Docs. Google uses an efficient technology to scan big data at a blazing speed which is called as "Dremel". Well, Dremel is a query service that allows you to run SQL-like queries against very large data sets and return accurate results in seconds.

Dremel can scan 35 billion rows without an index within ten-seconds. Dremel stores data in a columnar storage model, which means that it separates a record into column values and then stores each value on a different storage volume. But at the same time, traditional databases store the whole record on one volume. This columnar approach is the main reason that it makes Dremel drastically fast.

Google has been using Dremel in production since year 2006 and has been continuously evolving it for the applications like Spam analysis, Debugging of map tiles on Google Maps, etc. For this reason, Drill is inspired by Dremel. Recently, Google released BigQuery and it is the public implementation of Dremel that was launched for general businesses or developers to use.

What is Drill?

Apache Drill is a low latency schema-free query engine for big data. Drill uses a JSON document model internally which allows it to query data of any structure. Drill works with a variety of non-relational data stores, including Hadoop, NoSQL databases (MongoDB, HBase) and cloud storage like Amazon S3, Azure Blob Storage, etc. Users can query the data using a standard SQL and BI Tools, which doesn't require to create and manage schemas.

Benefits

Following are some of the most important benefits of Apache Drill:

- Drill can scale data from a single node to thousands of nodes and query petabytes of data within seconds.

- Drill supports user defined functions.

- Drill's symmetrical architecture and simple installation makes it easy to deploy and operate very large clusters.

- Drill has flexible data model and extensible architecture.

- Drill columnar execution model performs SQL processing on complex data without flattening into rows.

- Supports large datasets

Key Features

Following are some of the most significant key features of Apache Drill:

- Drill's pluggable architecture enables connectivity to multiple datastores.
- Drill has a distributed execution engine for processing queries. Users can submit requests to any node in the cluster.
- Drill supports complex/multi-structured data types.
- Drill uses self-describing data where a schema is specified as a part of the data itself, so no need for centralized schema definitions or management.
- Flexible deployment options either local node or cluster.
- Specialized memory management that reduces the amount of main memory that a program uses or references while running and eliminates garbage collections.
- Decentralized data management.

Use Cases

Apache Drill can work along with a few other softwares, some of which are:

- **Cloud JSON and Sensor Analytics:** Drill's columnar approach leverages to access JSON data and expose those data via REST API to apply sensor analytics information.
- **Works well with Hive:** Apache Drill serves as a complement to Hive deployments with low latency queries. Drill's hive metastore integration exposes existing datasets at interactive speeds.
- **SQL for NoSQL:** Drill's ODBC driver and powerful parallelization capabilities provide interactive query capabilities.

Need for Drill

Apache Drill comes with a flexible JSON-like data model to natively query and process complex/multi-structured data. The data does not need to be flattened or transformed either at the design time or runtime, which provides high performance for queries. Drill exposes an easy and high performance **Java API** to build custom functions. Apache Drill is built to scale to big data needs and is not restricted by memory available on the cluster nodes.

Drill Integration

Drill has to integrate with a variety of data stores like relational data stores or non-relational data stores. It has the flexibility to add new data stores.

Integration with File Systems

- **Traditional file system:** Local files and NAS (Network Attached Storage)
- **Hadoop:** HDFS and MAPR-FS (MAPR-File System)
- **Cloud storage:** Amazon S3, Google Cloud Storage, Azure Blob Storage

Integration with NoSQL Databases

- MongoDB
- HBase
- HIVE
- MapR-DB

2. Apache Drill – Fundamentals

In this chapter, we will discuss about the nested data model, JSON, Apache Avro, nested query language along with some other components in detail.

Drill Nested Data Model

Apache Drill supports various data models. The initial goal is to support the column-based format used by Dremel, then it is designed to support schema less models such as JSON, BSON (Binary JSON) and schema based models like Avro and CSV.

JSON

JSON (JavaScript Object Notation) is a lightweight text-based open standard designed for human-readable data interchange. JSON format is used for serializing and transmitting structured data over network connection. It is primarily used to transmit data between a server and web applications. JSON is typically perceived as a format whose main advantage is that it is simple and lean. It can be used without knowing or caring about any underlying schema.

Following is a basic JSON schema, which covers a classical product catalog description –

```
{
   "$schema": "http://json-schema.org/draft-04/schema#",
   "title": "Product",
   "description": "Classical product catalog",
   "type": "object",

   "properties": {

      "id": {
         "description": "The unique identifier for a product",
         "type": "integer"
      },

      "name": {
         "description": "Name of the product",
         "type": "string"
      },
         "price": {
         "type": "number",
         "minimum": 0,
         "exclusiveMinimum": true
      }
   },

   "required": ["id", "name", "price"]
}
```

The JSON Schema has the capability to express basic definitions and constraints for data types contained in objects, and it also supports some more advanced features such as properties typed as other objects, inheritance, and links.

Apache Avro

Avro is an Apache open source project that provides data serialization and data exchange services for Hadoop. These services can be used together or independently. Avro is a schema-based system. A language-independent schema is associated with its read and write operations. Using Avro, big data can be exchanged between programs written in any language. Avro supports a rich set of primitive data types including numeric, binary data and strings, and a number of complex types including arrays, maps, enumerations and records. A key feature of Avro is the robust support for data schemas that change over time.

Simple Avro Schema

Avro schema is created in JavaScript Object Notation (JSON) document format, which is a lightweight text-based data interchange format.

For example:

The given schema defines a (record type) document within "AvroSample" namespace. The name of document is "Employee" which contains two "Fields" → Name and Age.

```
{
 " type " : "record",
 " namespace " : "AvroSample",
 " name " : "Employee",
 " fields " : [
 { "name" : " Name" , "type" : "string" },
 { "name" : "age" , "type" : "int" }
 ]
}
```

The above schema contains four attributes, they have been briefly described here:

- **type –** Describes document type, in this case a "record"
- **namespace –** Describes the name of the namespace in which the object resides
- **name –** Describes the schema name
- **fields –** This is an attribute array which contains the following
- **name –** Describes the name of field
- **type –** Describes data type of field

Nested Query Language

Apache Drill supports various query languages. The initial goal is to support the SQL-like language used by Dremel and Google BigQuery. DrQL and Mongo query languages are an examples of Drill nested query languages.

DrQL

The DrQL (Drill Query Language) is a nested query language. DrQL is SQL like query language for nested data. It is designed to support efficient column-based processing.

Mongo Query Language

The MongoDB is an open-source document database, and leading NoSQL database. MongoDB is written in C++ and it is a cross-platform, document-oriented database that provides, high performance, high availability, and easy scalability. MongoDB works on the concept of collection and documenting.

Wherein, collection is a group of MongoDB documents. It is the equivalent of an RDBMS table. A collection exists within a single database. A document is a set of key-value pairs.

Drill File Format

Drill supports various file formats such as **CSV, TSV, PSV, JSON and Parquet**. Wherein, "Parquet" is the special file format which helps Drill to run faster and its data representation is almost identical to Drill data representation.

Parquet

Parquet is a columnar storage format in the Hadoop ecosystem. Compared to a traditional row-oriented format, it is much more efficient in storage and has better query performance. Parquet stores binary data in a column-oriented way, where the values of each column are organized so that they are all adjacent, enabling better compression.

It has the following important characteristics:

- Self-describing data format
- Columnar format
- Flexible compression options
- Large file size

Flat Files Format

The Apache Drill allows access to structured file types and plain text files (flat files). It consists of the following types –

- CSV files (comma-separated values)
- TSV files (tab-separated values)
- PSV files (pipe-separated values)

CSV file format: A CSV is a comma separated values file, which allows data to be saved in a table structured format. CSV data fields are often separated or delimited by comma (,). The following example refers to a CSV format.

```
firstname, age
Alice,21
Peter,34
```

This CSV format can be defined as follows in a drill configuration.

```
"formats": {
     "csv": {
                    "type": "text",
                    "extensions": [
                          “csv2"
                    ],
                    "delimiter": “,”
     }
}
```

TSV file format: The TSV data fields are often separated or delimited by a tab and saved with an extension of ".tsv" format. The following example refers to a TSV format.

```
firstname   age
Alice       21
Peter       34
```

The TSV format can be defined as follows in a drill configuration.

```
"tsv": {
     "type": "text",
            "extensions": [
                   "tsv"
            ],
            "delimiter": "\t"
},
```

PSV file format: The PSV data fields are separated or delimited by a pipe (|) symbol. The following example refers to a PSV format.

```
firstname|age
Alice|21
Peter|34
```

The PSV format can be defined as follows in a drill configuration.

```
"formats": {
     "psv": {
                   "type": "text",
                   "extensions": [
                          "tbl"
                   ],
            "delimiter": "|"
     }
}
```

These PSV files are saved with an extension of ".tbl" format.

Scalable Data Sources

Managing millions of data from multiple data sources requires a great deal of planning. When creating your data model, you need to consider the key goals such as the impact on speed of processing, how you can optimize memory usage and performance, scalability when handling growing volumes of data and requests.

Apache Drill provides the flexibility to immediately query complex data in native formats, such as schema-less data, nested data, and data with rapidly evolving schemas.

Following are its key benefits:

- High-performance analysis of data in its native format including self-describing data such as Parquet, JSON files and HBase tables.
- Direct querying of data in HBase tables without defining and maintaining a schema in the Hive metastore.
- SQL to query and work with semi-structured/nested data, such as data from NoSQL stores like MongoDB and online REST APIs.

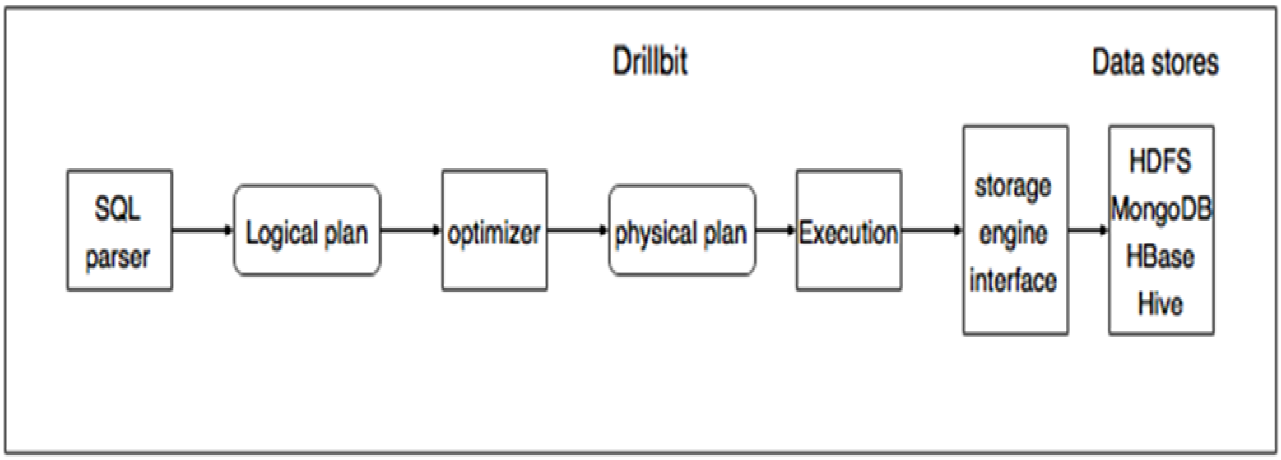
Drill Clients

Apache Drill can connect to the following clients –

- Multiple interfaces such as JDBC, ODBC, C++ API, REST using JSON
- Drill shell
- Drill web console (http://localhost:8047)
- BI tools such as Tableau, MicroStrategy, etc.
- Excel

3. Apache Drill – Architecture

As of now, you are aware of the Apache Drill fundamentals. This chapter will explain about its architecture in detail. Following is a diagram that illustrates the Apache Drill core module.



The above diagram consists of different components. Let’s take a look at each of these components in detail.

- **DrillBit:** Apache Drill consists of a Daemon service called the DrillBit. It is responsible for accepting requests from the client, processing queries, and returning results to the client. There is no master-slave concept in DrillBit.

- **SQL Parser:** The SQL parser parses all the incoming queries based on the open source framework called Calcite.

- **Logical Plan:** A Logical plan describes the abstract data flow of a query. Once a query is parsed into a logical plan, the Drill optimizer determines the most efficient execution plan using a variety of rule-based and cost-based techniques, translating the logical plan into a physical plan.

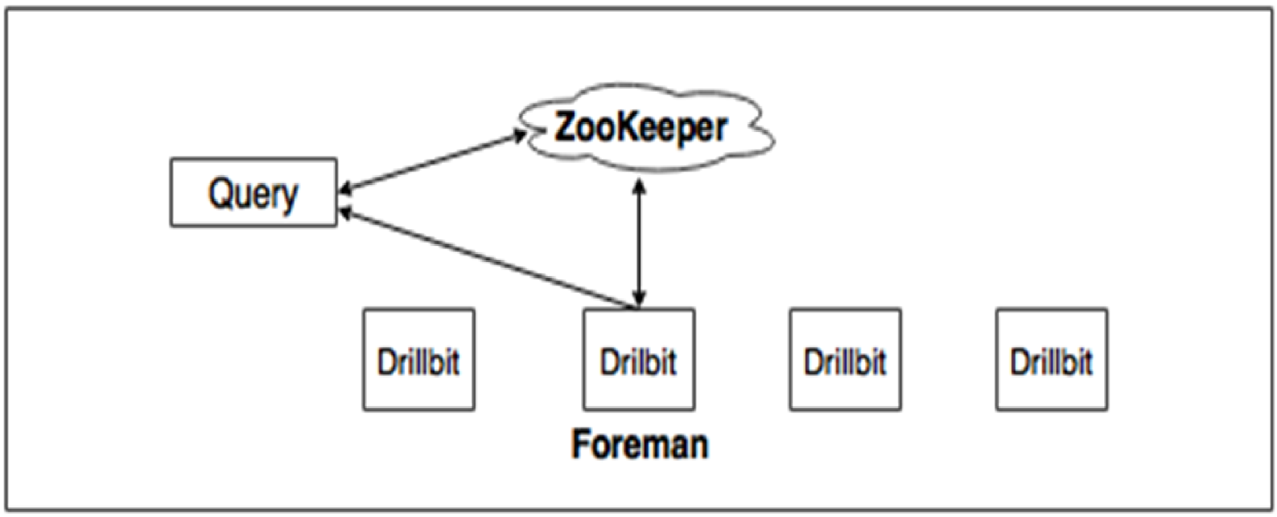
- **Optimizer:** Apache Drill uses various database optimizations such as rule based/cost based, as well as other optimization rules exposed by the storage engine to re-write and split the query. The output of the optimizer is a distributed physical query plan. Optimization in Drill is pluggable so you can provide rules for optimization at various parts of the query execution.

- **Physical Plan:** A Physical plan is also called as the execution plan. It represents the most efficient and fastest way to execute the query across the different nodes in the cluster. The physical plan is a DAG (directed acyclic graph) of physical operators, and each parent-child relationship implies how data flows through the graph.

- **Storage Engine interface:** A Storage plugin interfaces in Drill represent the abstractions that Drill uses to interact with the data sources. The plugins are extensible, allowing you to write new plugins for any additional data sources.

Query Execution Diagram

The following image shows a DrillBit query execution diagram:



The above diagram involves the following steps –

- The Drill client issues a query. Any Drillbit in the cluster can accept queries from clients.
- A Drillbit then parses the query, optimizes it, and generates an optimized distributed query plan for fast and efficient execution.
- The Drillbit that accepts the initial query becomes the Foreman (driving Drillbit) for the request. It gets a list of available Drillbit nodes in the cluster from ZooKeeper.
- The foreman gets a list of available Drillbit nodes in the cluster from ZooKeeper and schedules the execution of query fragments on individual nodes according to the execution plan.
- The individual nodes finish their execution and return data to the foreman.
- The foreman finally returns the results back to the client.

4. Apache Drill – Installation

This chapter will cover how to install Apache Drill on your machine. We have two modes of installation in Drill.

- **Embedded mode:** This mode refers to install Drill on a single node (local) on your machine. It doesn't require ZooKeeper setup.

- **Distributed mode:** Install Apache Drill on a distributed environment. ZooKeeper is mandatory for this mode because it co-ordinates clusters. Once you installed successfully, then you will be able to connect and query Hive, HBase or any other distributed data sources.

Now let's continue with the embedded mode steps for installation.

Embedded Mode Installation

Embedded mode is a quick way to install. You can install Apache Drill in the embedded mode on Linux, Mac OS or Windows Operating System.

Step 1: Verify Java Installation

Hopefully, you have already installed java on your machine, so you just verify it using the following command.

```
$ java -version
```

If Java is successfully installed on your machine, you could see the version of installed Java. Otherwise download the latest version of JDK by visiting the following link –

http://www.oracle.com/technetwork/java/javase/downloads/index.html

After downloading the latest version, extract those files, move to the directory after setting the path and add Java alternatives. Then Java will be installed on your machine.

Step 2: Apache Drill Installation

Download the latest version of Apache Drill by visiting the following link –

http://www.apache.org/dyn/closer.cgi/drill/drill-1.6.0/apache-drill-1.6.0.tar.gz

Now apache-drill-1.6.0.tar.gz will be downloaded on your machine.

You can then extract the tar file using the following program –

```
$ cd opt/

$ tar apache-drill-1.6.0.tar.gz

$ cd apache-drill-1.6.0
```

Step 3: Start Drill

To start the Drill shell in the embedded mode, use the following command. Internally, the command uses a **jdbc connection string** and identifies the local node as the **ZooKeeper node**.

```
$ bin/drill-embedded
```

After the command, you can see the following response:

```
$ 0: jdbc:drill:zk=local>
```

Where,

- **0 -** is the number of connections to Drill, which can be only one in embedded node
- **jdbc -** is the connection type
- **zk=local -** means the local node substitutes for the ZooKeeper node

Once you get this prompt, you will be able to run your queries on Drill.

Step 4: Exit Drill

To exit the Drill shell, issue the following command:

```
$ !quit
```

Distributed Mode Installation

You will have to follow the subsequent steps to ensure a proper Distributed Mode Installation on your system.

Step 1: Verify Java installation

```
$ java -version
```

If java is successfully installed on your machine, you could see the version of installed Java. Otherwise download latest version of JDK by visiting the following link –

http://www.oracle.com/technetwork/java/javase/downloads/index.html

After downloading the latest version, extract those files and move them to the directory after setting the path and adding Java alternatives. Then Java will be installed on your machine.

Step 2: Verify ZooKeeper Installation

Hopefully, you have installed Apache ZooKeeper on your machine. To verify the installation, you can issue the following command –

```
$ bin/zkServer.sh start
```

Then you will get the following program on your screen –

```
$ JMX enabled by default
$ Using config: /Users/../zookeeper-3.4.6/bin/../conf/zoo.cfg
$ Starting zookeeper ... STARTED
```

Step 3: Apache Drill Installation

You can start with downloading the latest version of Apache Drill by visiting the following link –

http://www.apache.org/dyn/closer.cgi/drill/drill-1.6.0/apache-drill-1.6.0.tar.gz

Now apache-drill-1.6.0.tar.gz will be downloaded on your machine.

The next step is to extract the tar file by issuing the following command –

```
$ cd opt/
$ tar apache-drill-1.6.0.tar.gz
$ cd apache-drill-1.6.0
```

Step 4: Configuration Settings

Open the drill-override file by using the following command.

```
$ vi conf/drill-override.conf
```

Then you will see the following response:

```
drill.exec: {
  cluster-id: "drillbits1",
  zk.connect: "localhost:2181"
}
```

Here cluster-id: "drillbits1" indicates we have one instance running. If two or more instances are running, then drillbits also increases.

Step 5: Start Drillbit shell

To start the drillbit shell you can use the following command.

```
$ bin/drillbit.sh start
```

Then you will see the following response:

```
$ 0: jdbc:drill:zk=<zk1host>:
```

Step 6: Exit the Drill Shell

To exit the Drill shell, you can issue the following command:

```
$ 0: jdbc:drill:zk=<zk1host>: !quit
```

Step 7: Stop Drill

Navigate to the Drill installation directory, and issue the following command to stop a Drillbit.

```
$ bin/drillbit.sh stop
```

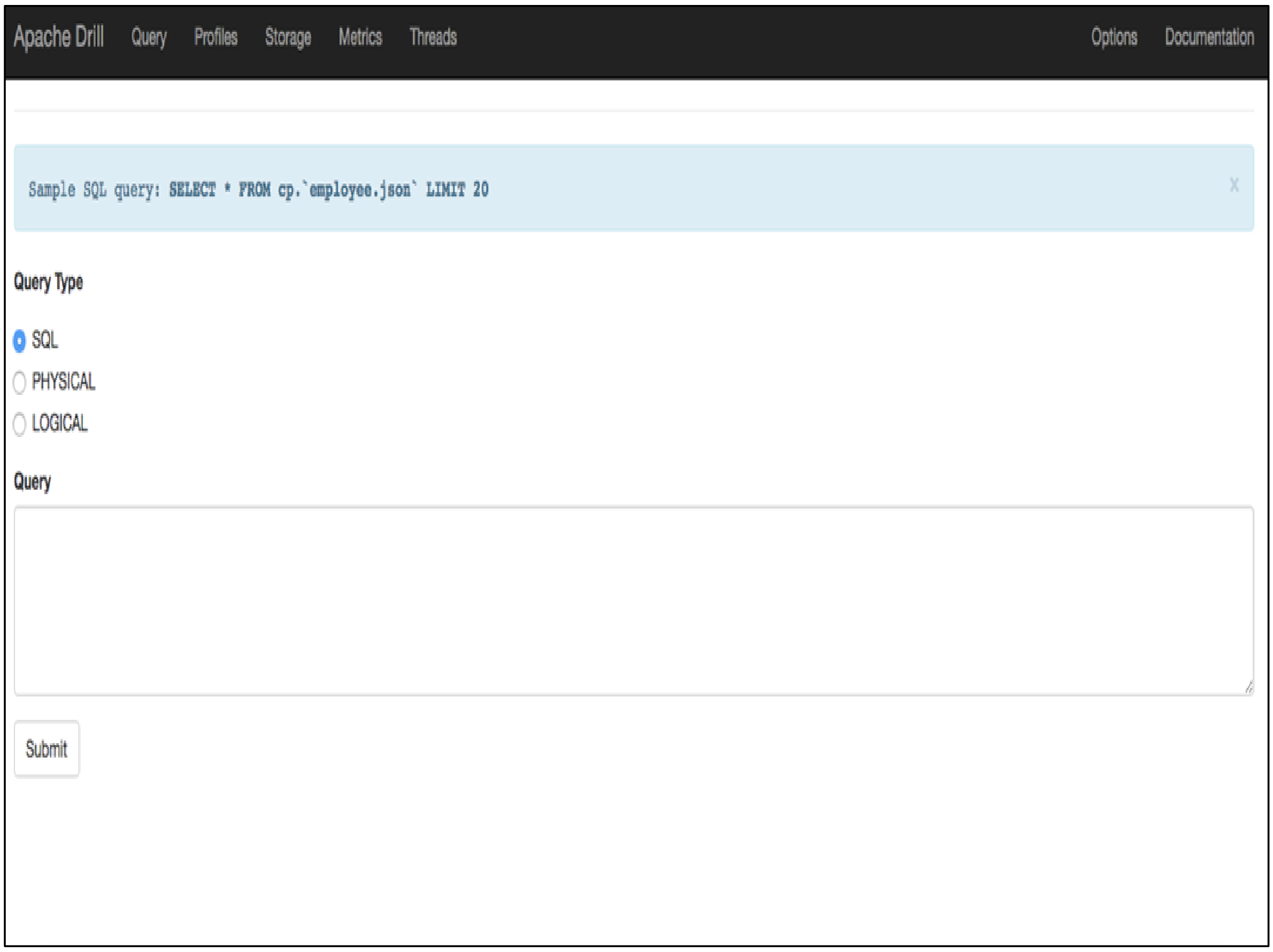
Step 8: Start Drill in Web Console

Apache Drill Web Console is one of the client interfaces to access Drill.

To open this Drill Web Console, launch a web browser, and then type the following URL –

http://localhost:8047

Now you will see the following screen which is similar to the Query option.



Sample SQL query: SELECT * FROM cp.`employee.json` LIMIT 20

Query Type

[✓ SQL]
PHYSICAL
LOGICAL

Query

5. Apache Drill – SQL Operations

Apache Drill is an open-source SQL-On-Everything engine. It allows SQL queries to be executed on any kind of data source, ranging from a simple CSV file to an advanced SQL and NoSQL database servers.

To execute a query in a Drill shell, open your terminal move to the Drill installed directory and then type the following command.

```
$ bin/drill-embedded
```

Then you will see the response as shown in the following program.

```
0: jdbc:drill:zk=local>
```

Now you can execute your queries. Otherwise you can run your queries through web console application to the url of http://localhost:8047. If you need any help info type the following command.

```
$ 0: jdbc:drill:zk=local> !help
```

Primitive Data Types

Apache Drill supports the following list of data types.

Datatype	Description
BIGINT	8-byte signed integer in the range -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
BINARY	Variable-length byte string
BOOLEAN	True or false
DATE	Years, months, and days in YYYY-MM-DD format since 4713 BC
DECIMAL(p,s), DEC(p,s), or NUMERIC(p,s)*	38-digit precision number, precision is p, and scale is s
INTEGER or INT	4-byte signed integer in the range -2,147,483,648 to 2,147,483,647
INTERVAL	A day-time or year-month interval

SMALLINT	2-byte signed integer in the range -32,768 to 32,767
FLOAT	4-byte floating point number
DOUBLE	8-byte floating point number
TIME	24-hour based time in hours, minutes, seconds format: HH:mm:ss
TIMESTAMP	JDBC timestamp in year, month, date hour, minute, second, and optional milliseconds format: yyyy-MM-dd HH:mm:ss.SSS
CHARACTER VARYING, CHAR or VARCHAR	variable-length string. The default limit is 1 character. The maximum character limit is 2,147,483,647.

Let us continue with simple examples on the data types.

Date, Time and Timestamp

Apache Drill supports time functions in the range from 1971 to 2037. The processing logic of data types can be easily tested by "VALUES()" statement. The following query returns date, time and timestamp for the given values.

Query:

```
0: jdbc:drill:zk=local> select DATE '2016-04-07',TIME '12:12:23',TIMESTAMP '2016-04-
07 12:12:23' from (values(1));
```

Result:

```
+-------------+----------+-----------------------+
|   EXPR$0    |  EXPR$1  |        EXPR$2         |
+-------------+----------+-----------------------+
| 2016-04-07  | 12:12:23  | 2016-04-07 12:12:23.0  |
+-------------+----------+-----------------------+
```

Interval

- The INTERVALYEAR and INTERVALDAY internal types represent a period of time.
- The INTERVALYEAR type specifies values from a year to a month.
- The INTERVALDAY type specifies values from a day to seconds.

INTERVALYEAR Query

```
0: jdbc:drill:zk=local> select timestamp '2016-04-07 12:45:50' + interval '10'
year(2) from (values(1));
```

Result:

```
+------------------------+

|         EXPR$0         |

+------------------------+

| 2026-04-07 12:45:50.0  |

+————————————————————————-+

1 row selected (0.251 seconds)
```

In the above query, INTERVAL keyword followed by 10 adds 10 years to the timestamp. The 2 in parentheses in YEAR(2) specifies the precision of the year interval, 2 digits in this case to support the ten interval.

INTERVALDAY Query

```
0: jdbc:drill:zk=local> select timestamp '2016-04-07 12:45:52' + interval '1' day(1)
from (values(1));
```

Result:

```
+------------------------+

|         EXPR$0         |

+------------------------+

| 2016-04-08 12:45:52.0  |

+————————————————————————-+
```

Here INTERVAL '1' indicates that two days will be added from that specified day.

Operators

The following operators are used in Apache Drill to perform the desired operations.

Operators	Description
Logical Operators	AND, BETWEEN, IN , LIKE , NOT , OR
Comparison Operators	<, > , <= , >= , = , <> , IS NULL , IS NOT NULL , IS FALSE , IS NOT FALSE , IS TRUE , IS NOT TRUE, Pattern Matching Operator - LIKE

Math Operators	+,-,*,/
Subquery Operators	EXISTS, IN

Drill Scalar Functions

Apache Drill scalar functions supports Math and Trig functions. Most scalar functions use data types such as INTEGER, BIGINT, FLOAT and DOUBLE.

Math Functions

The following table describes the list of "Math functions" in Apache Drill.

Function	Description
ABS(x)	Returns the absolute value of the input argument x.
CBRT(x)	Returns the cubic root of x.
CEIL(x)	Returns the smallest integer not less than x.
CEILING(x)	Same as CEIL.
DEGREES(x)	Converts x radians to degrees.
E()	Returns 2.718281828459045.
EXP(x)	Returns e (Euler's number) to the power of x.
FLOOR(x)	Returns the largest integer not greater than x.
LOG(x)	Returns the natural log (base e) of x.
LOG(x, y)	Returns log base x to the y power.
LOG10(x)	Returns the common log of x.

LSHIFT(x, y)	Shifts the binary x by y times to the left.
MOD(x, y)	Returns the remainder of x divided by y.
NEGATIVE(x)	Returns x as a negative number.
PI	Returns pi.
POW(x, y)	Returns the value of x to the y power.
RADIANS	Converts x degrees to radians.
RAND	Returns a random number from 0-1.
ROUND(x)	Rounds to the nearest integer.
ROUND(x, y)	Rounds x to y decimal places. Return type is decimal
RSHIFT(x, y)	Shifts the binary x by y times to the right.
SIGN(x)	Returns the sign of x.
SQRT(x)	Returns the square root of x.
TRUNC(x, y)	Truncates x to y decimal places. Specifying y is optional. Default is 1.
TRUNC(x, y)	Truncates x to y decimal places.

Now let’s run queries for the scalar functions. The Drill scalar functions can be easily tested by the **values()** statement, otherwise you can also use the **select** statement.

ABS(x)

The output of this function type is the same as the same input type.

Query:

```
0: jdbc:drill:zk=local> values(ABS(1.899));
```

or

```
0: jdbc:drill:zk=local> select ABS(1.899) from (values(1));
```

The result will be as shown in the following program:

```
+---------+
| EXPR$0  |
+---------+
| 1.899   |
+---------+
```

CBRT(x)

This cubic root returns the output type as a float type.

Query:

```
0: jdbc:drill:zk=local> values(CBRT(125));
```

Result:

```
+---------+
| EXPR$0  |
+---------+
| 5.0     |
+---------+
```

CEIL(x)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> values(ceil(4.6));
```

Result:

```
+---------+
| EXPR$0  |
+---------+
| 5.0     |
+---------+
```

The output is returned as the largest following value.

Degrees(x)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> values(degrees(30));
```

Result:

```
+---------------------+
|       EXPR$0        |
+---------------------+
| 1718.8733853924698  |
+---------------------+
```

The deg(30) value is returned as the output.

Exp(x)

This function returns the floating point value.

Query:

```
0: jdbc:drill:zk=local> values(Exp(3));
```

Result:

```
+---------------------+
|       EXPR$0        |
+---------------------+
| 20.085536923187668  |
+─────────────────────+
```

The output result is the exponential value of 3.

floor(x)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> values(floor(3.9));
```

Result:

```
+---------+
| EXPR$0  |
+---------+
| 3.0     |
+---------+
```

The given input value 3.9 is changed to the floor value which is 3.0.

log(x,y)

This function returns a floating type value.

Query:

```
0: jdbc:drill:zk=local> values(log(2,10));
```

Result:

```
+---------------------+
|       EXPR$0        |
+---------------------+
| 3.3219280948873626  |
+———————————-+
```

The output is a logarithmic value for the given input.

Round(x)

The following program shows the query for this function:

Query:

```
0: jdbc:drill:zk=local> values(round(2.7));
```

Result:

```
+---------+
| EXPR$0  |
+---------+
| 3.0     |
+---------+
```

The output value is rounded to the next integer or a floating point value.

trunc(x)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> values(trunc(4.99));
```

Result:

```
+---------+
| EXPR$0  |
+---------+
| 4.0     |
+—————————-+
```

Here the input 4.99 is truncated to 4.0.

Similarly, you can try to run the other functions from the above given table.

Trig Functions

Apache Drill supports the following trig functions and these functions' return type is a floating point value.

Function	Description
SIN(x)	Sine of angle x in radians
COS(x)	Cosine of angle x in radians

TAN(x)	Tangent of angle x in radians
ASIN(x)	Inverse sine of angle x in radians
ACOS(x)	Inverse cosine of angle x in radians
ATAN(x)	Inverse tangent of angle x in radians
SINH()	Hyperbolic sine of hyperbolic angle x in radians
COSH()	Hyperbolic cosine of hyperbolic angle x in radians
TANH()	Hyperbolic tangent of hyperbolic angle x in radians

Let's go through some simple examples on the above mentioned Trig functions.

sin(x)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> values(sin(45));
```

Result:

```
+----------------------+
|        EXPR$0        |
+----------------------+
| 0.8509035245341184   |
+----------------------+
```

Here the **sin 45 value** is returned as the output.

cosh(x)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> values(cosh(90));
```

Result:

```
+-----------------------+
|        EXPR$0         |
+-----------------------+
| 6.102016471589204E38  |
+-----------------------+
```

The output result is a **hyperbolic cosine value** for the angle 90.

Acos(x)

This acos(x) function returns the Inverse Cosine of the elements of x.

Query:

```
0: jdbc:drill:zk=local> select acos(0.3) as inversecosine from (values(1));
```

Result:

```
+---------------------+
|    inversecosine    |
+---------------------+
| 1.2661036727794992  |
+---------------------+
```

The output is inverse cosine for the given value.

Data Type Conversion

In Apache Drill, you can cast or convert data to the required type for moving data from one data source to another. Drill also supports the following functions for casting and converting data types:

Function	Return type	Description
CAST(x AS y)	Data type of y	Converts the data type of x to y
CONVERT_TO(x,y)	Data type of y	Converts binary data (x) to Drill internal types (y) based on the little or big endian encoding of the data.
CONVERT_FROM(x,y)	Data type of y	Converts binary data (x) from Drill internal types (y) based on the little or big endian encoding of the data.

CAST(x AS y)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select cast('3' as double) from (values(1));
```

Result:

```
+--------+
| EXPR$0 |
+--------+
| 3.0    |
+--------+
```

Here the input value integer 3 is casting as double 3.0.

CONVERT_FROM()

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> Select CONVERT_FROM ('{x:1, y:2}' ,'JSON') AS Convertion from
(values(1));
```

Result:

```
+--------------------+
|     Conversion     |
+--------------------+
| {"x":1,"y":2}      |
+--------------------+
```

The above query converts **varchar data** to JSON format. Similarly, you can use other data types to Drill supported data format. For naming columns, you can use the alias method.

CONVERT_TO()

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select CONVERT_TO(2,'int') as conversion from (values(1));
```

Result:

```
+―――――――――――――+
| conversion  |
+―――――――――――――-+
| 00000040    |
+―――――――――――――-+
```

Here the output is returned as the hexadecimal value for 2.

Date - Time Functions

Apache Drill supports time functions based on the Gregorian calendar and in the range from 1971 to 2037. The following table describes the list of Date/Time functions.

Function	Return Type	Description
AGE(x [, y])	INTERVALDAY or INTERVALYEAR	Returns interval between two timestamps or subtracts a timestamp from midnight of the current date.
CURRENT_DATE	DATE	Returns current date
CURRENT_TIME	TIME	Returns current time
CURRENT_TIMESTAMP	TIMESTAMP	Returns current timestamp

DATE_ADD(x,y)	DATE, TIMESTAMP	Returns the sum of the sum of a date/time and a number of days/hours, or of a date/time and date/time interval. Where, x- date,time or timestamp y - integer or an interval expression.
DATE_SUB(x,y)	DATE, TIMESTAMP	Subtracts an interval (y) from a date or timestamp expression (x).
DATE_PART(x,y)	DOUBLE	Returns a field of a date, time, timestamp, or interval. where, x-year, month, day, hour, minute, or second y-date, time, timestamp, or interval literal
EXTRACT(x FROM y)	DOUBLE	Extracts a time unit from a date or timestamp expression (y). This must be one of the following values: SECOND, MINUTE, HOUR, DAY, MONTH, and YEAR.
LOCALTIME	TIME	Returns the local current time.
LOCALTIMESTAMP	TIMESTAMP	Returns the local current timestamp.
NOW()	TIMESTAMP	Returns current timestamp
TIMEOFDAY()	VARCHAR	Returns current timestamp for UTC time zone.
UNIX_TIMESTAMP ([x])	BIGINT	If x is specified as timestamp then the number of seconds since the UNIX epoch and the timestamp x is returned. If x is not specified then it returns the number of seconds since the UNIX epoch (January 1, 1970 at 00:00:00).

Age(x, [,y]) function

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select age('2000-04-13') from (values(1));
```

Result:

```
+-----------------+
|     EXPR$0      |
+-----------------+
| P195M4DT66600S  |
+—————————————————+
```

The output result is the interval limit from the specified year to midnight of the current day.

CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP Function

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select CURRENT_DATE,CURRENT_TIME,CURRENT_TIMESTAMP from
(values(1));
```

Result:

```
+---------------+---------------+-------------------------+
| current_date  | current_time  |    current_timestamp    |
+---------------+---------------+-------------------------+
| 2016-04-07    | 11:50:34.384  | 2016-04-24 11:50:34.384 |
+————————+————————+—————————————————————————————————————————+
```

The output returns current date, time, and timestamp for the day.

LOCALTIME, LOCALTIMESTAMP Functions

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select LOCALTIME,LOCALTIMESTAMP from (values(1));
```

Result:

```
+---------------+-------------------------+
|   LOCALTIME   |      LOCALTIMESTAMP     |
+---------------+-------------------------+
| 15:17:46.333  | 2016-04-24 15:17:46.333 |
+---------------+-------------------------+
```

NOW(),TIMEOFDAY()

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select NOW(),TIMEOFDAY() from (values(1));
```

Result:

```
+--------------------------+---------------------------------------+
|          EXPR$0          |                EXPR$1                 |
+--------------------------+---------------------------------------+
| 2016-04-24 15:19:23.975  | 2016-04-24 15:19:24.243 Asia/Kolkata  |
+--------------------------+---------------------------------------+
```

Here, TIMEOFDAY() returns the result for UTC time zone.

DATE_ADD(x, integer)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select DATE_ADD('2016-04-07',3) FROM (VALUES(1));
```

Result:

```
+———————-+
|   EXPR$0
+————————+
| 2016-04-10
+————————+
```

From this result, 3 days will be added.

DATE_ADD(x, interval)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select date_add(‘2016-04-07',6-2) from (values(1));
```

Result:

```
+—————————+
|   EXPR$0
+————————-+
| 2016-04-11
+————————-+
```

Here the interval limit 6-2 gives the result as 4, then the result 4 will be added to the given date.

DATE_SUB(x,y)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select date_sub('2016-04-07',2) from (values(1));
```

Result:

```
+-------------+
|   EXPR$0    |
+-------------+
| 2016-04-05  |
+-------------+
```

The output indicates 2 days subtracted from the specified day.

EXTRACT(x from y)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select EXTRACT(SECOND FROM TIME '12:20:40') from(values(1));
```

Result:

```
+---------+
| EXPR$0
+---------+
| 40.0
+---------+
```

The seconds extracted from the given time.

String Manipulation Function

Apache Drill supports the following list of string functions.

Function	Return type	Description
BYTE_SUBSTR(x,y [, z])	BINARY or VARCHAR	Returns in binary format a substring y of the string x.
CHAR_LENGTH(x)	INTEGER	Returns the length of the alphanumeric argument x.
CONCAT(x,y)	VARCHAR	Combines the two alphanumeric values x and y. Has the same effect as the || operator.

INITCAP(x)	VARCHAR	Returns x in which the first character is capitalized.
LENGTH(x)	INTEGER	Returns the length in bytes of the alphanumeric value x.
LOWER(x)	VARCHAR	Converts all upper-case letters of x to lower-case letters.
LPAD(x,y [, z])	VARCHAR	The value of x is filled in the front (the left-hand side) with the value of z until the total length of the value is equal y's length. If no z value then blanks are used to fill the position.
LTRIM(x)	VARCHAR	Removes all blanks that appear at the beginning of x.
POSITION(x IN y)	INTEGER	Returns the start position of the string x in the string y.
REGEXP_REPLACE(x,y,x)	VARCHAR	Substitutes new text for substrings that match Java regular expression patterns. In the string x, y is replaced by z. Y is the regular expression.
RPAD(x,y,z)	VARCHAR	The value of x is filled in the front (the right-hand side) with the value of z just until the total length of the value is equal to that of y.
RTRIM(x)	VARCHAR	Removes all blanks from the end of the value of x.
STRPOS(x,y)	INTEGER	Returns the start position of the string y in the string x.
SUBSTR(x,y,z)	VARCHAR	Extracts characters from position 1 - x of x an optional y times.
TRIM(x)	VARCHAR	Removes all blanks from the start and from the end of x. Blanks in the middle are not removed.
UPPER(x)	VARCHAR	Converts all lower-case letters of x to upper-case letters.

Now let's continue to query on string functions.

BYTE_SUBSTR(x,y [, z])

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select BYTE_SUBSTR('Drill',1,2) from (values(1));
```

Result:

```
+————————-+
| EXPR$0
+————————-+
| 4472
+————————-+
```

The above query returns a binary format of the substring position of the string Drill.

CHAR_LENGTH(x)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select char_length('Drill') from (values(1));
```

Result:

```
+————————+
| EXPR$0
+———————-+
| 5
+———————-+
```

The query returns the output value length as 5 for the string “Drill”.

CONCAT(x,y)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select concat('apache','drill') from (values(1));
```

Result:

```
+——————————————-+
|    EXPR$0
+——————————————-+
| apachedrill
+——————————————-+
```

The above query produces the result of concatenation of two specified strings.

INITCAP(x)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select initcap('drill') from (values(1));
```

Result:

```
+--------+
| EXPR$0
+--------+
| Drill
+--------+
```

The Initcap function returns the result as the **first character** of the string becomes capitalized.

LENGTH(x)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select length('apache drill') from (values(1));
```

Result:

```
+---------+
| EXPR$0
+---------+
| 12
+---------+
```

This query returns the length of the given string.

LOWER(x)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select lower('APACHE DRILL') from (values(1));
```

Result:

```
+-------------+
|    EXPR$0

+-------------+
| apache drill

+-------------+
```

It converts the given string to a lower case format.

UPPER(x)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select upper('apache drill') from (values(1));
```

Result:

```
+---------------+
|    EXPR$0     |
+---------------+
| APACHE DRILL  |
+---------------+
```

It converts the given string to the upper case format.

LPAD(x,y [, z])

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select lpad('drill',2,4) from (values(1));
```

Result:

```
+---------+
| EXPR$0  |
+---------+
| dr      |
+---------+
```

Left pad the value of the given string “drill” from the position of 2 to 4, so the result will be just **dr**.

RPAD(x,y,z)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select rpad('tutorialspoint',2,4) from (values(1));
```

Result:

```
+--------+
| EXPR$0
+--------+
| tu
+--------+
```

Right pad the value of the given string.

RTRIM(x)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select rtrim('tutorialspoint','point') from (values(1));
```

Result:

```
+-----------+
|   EXPR$0
+-----------+
| tutorials
+-----------+
```

Right trimming the characters form the two given strings.

LTRIM(x)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select ltrim('tutorialspoint','point') from (values(1));
```

Result:

```
+───────────────-+
|     EXPR$0
+───────────────-+
| utorialspoint
+────────────────+
```

Left trimming the character from the given string.

REGEXP_REPLACE(x,y,x)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select regexp_replace('new','e','o') from (values(1));
```

Result:

```
+────────-+
| EXPR$0  |
+────────-+
| now     |
+────────-+
```

Here the given string **new** is replaced as **now**.

STRPOS(x,y)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select strpos('tutorialpoint','o') from (values(1));
```

Result:

```
+──────+
| EXPR$0
+──────+
| 4
+──────+
```

The output indicates the position of 'o' occurs first in the given string.

POSITION(x IN y)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select position('o' in 'tutorialspoint') from (values(1));
```

Result:

```
+-------+
| EXPR$0
+--------+
| 4
+--------+
```

SUBSTR(x,y,z)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select substr('tutorialspint',4,7) from (values(1));
```

Result:

```
+----------+
|  EXPR$0  |
+----------+
| orialsp  |
+----------+
```

Null Handling Function

Apache Drill supports the following list of null handling functions.

Function	Return type	Description
COALESCE(x, y [, y]...)	Data type of y	Returns the first non-null argument in the list of y's.
NULLIF(x,y)	Data type of y	Returns the value of the x if x and y are not equal, and returns a null value if x and y are equal.

COALESCE(x, y [, y]...)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select coalesce(3,1,7) from (values(1));
```

Result:

```
+---------+
| EXPR$0  |
+---------+
| 3       |
+————————-+
```

Here first **arg** is non null, so it returns the value 3.

NULLIF(x,y)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select NULLIF(3,3) from (values(1));
```

Result:

```
+---------+
| EXPR$0  |
+---------+
| null    |
+---------+
```

Here both the arguments are same, so the result is NULL.

6. Apache Drill – Query Using JSON

Apache Drill supports JSON format for querying data. Drill treats a JSON object as SQL record. One object equals one row in a Drill table.

Querying JSON File

Let us query the sample file, "employee.json" packaged as part of the drill. This sample file is Foodmart data packaged as JAR in Drill's classpath: ./jars/3rdparty/foodmart-data-json.0.4.jar. The sample file can be accessed using namespace, cp.

Start the Drill shell, and select the first row of data from the "employee.json" file installed.

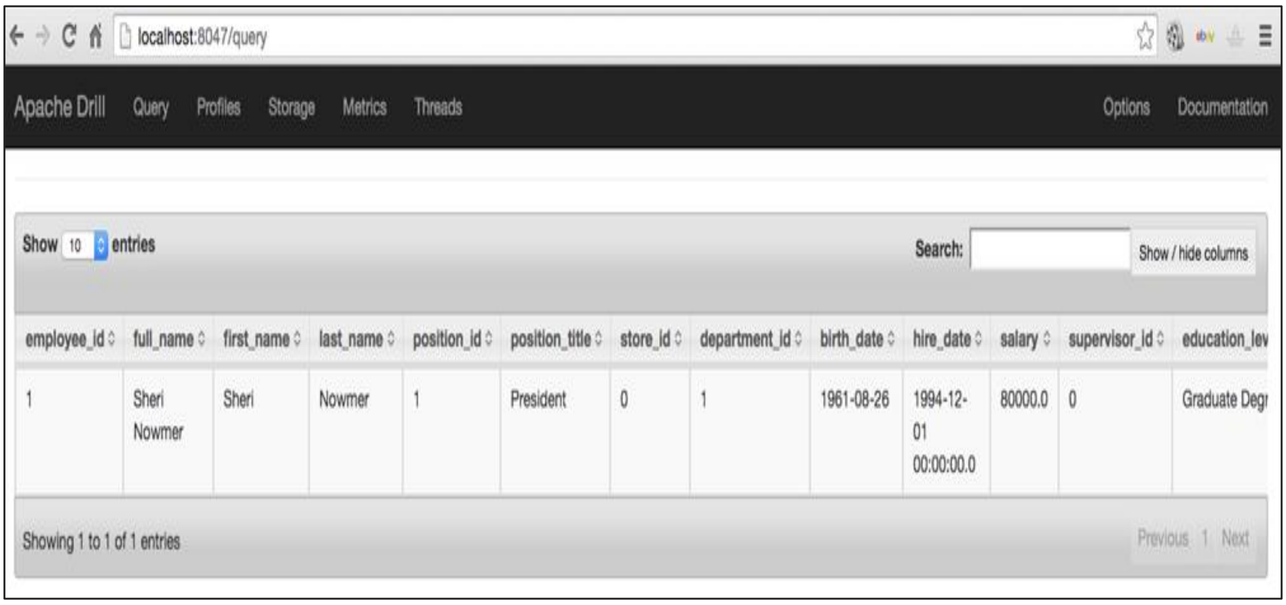
Query:

```
0: jdbc:drill:zk=local> select * from cp.`employee.json` limit 1;
```

Result:

```
+--------------+---------------+-------------+------------+--------------+-----------
------+-----------+----------------+-------------+------------------------+----------
+----------------+------------------+-----------------+---------+-------------------+

| employee_id  |   full_name   | first_name    | last_name   | position_id  |
position_title |   store_id    | department_id | birth_date  | hire_date    |
salary   | supervisor_id  | education_level   | marital_status  | gender       |
management_role    |

+--------------+---------------+-------------+------------+--------------+-----------
------+-----------+----------------+-------------+------------------------+----------
+----------------+------------------+-----------------+---------+--------------------
+

| 1          | Sheri Nowmer   | Sheri        | Nowmer      | 1           | President
| 0          | 1              | 1961-08-26   | 1994-12-01 00:00:00.0  | 80000.0   | 0
| Graduate Degree   | S       | F            | Senior Management |

+--------------+---------------+-------------+------------+--------------+-----------
------+-----------+----------------+-------------+------------------------+----------
+----------------+------------------+-----------------+---------+--------------------
+
```

The same result can be viewed in the web console as –



employee_id	full_name	first_name	last_name	position_id	position_title	store_id	department_id	birth_date	hire_date	salary	supervisor_id	education_lev
1	Sheri Nowmer	Sheri	Nowmer	1	President	0	1	1961-08-26	1994-12-01 00:00:00.0	80000.0	0	Graduate Degr

Showing 1 to 1 of 1 entries

Storage Plugin Configuration

You can connect Drill to a file system through a storage plugin. On the Storage tab of the Drill Web Console (http://localhost:8047), you can view and reconfigure a storage plugin.

The Drill installation contains the following default storage plugin configurations.

- **cp -** Points to the JAR files in the Drill classpath.
- **dfs -** Points to the local file system, but you can configure this storage plugin to point to any distributed file system, such as a Hadoop or S3 file system.
- **hbase -** Provides a connection to the HBase.
- **hive -** Integrates Drill with the Hive metadata abstraction of files, HBase, and libraries to read data and operate on SerDes and UDFs.
- **mongo -** Provides a connection to MongoDB data.

Storage Plugin Configuration Persistence

- **Embedded mode:** Apache Drill saves the storage plugin configurations in a temporary directory. The temporary directory clears when you reboot.
- **Distributed mode:** Drill saves storage plugin configurations in ZooKeeper.

Workspace

The workspace defines the location of files in subdirectories of a local or distributed file system. One or more workspaces can be defined in a plugin.

Create JSON file

As of now we have queried an already packaged "employee.json" file. Let us create a new JSON file named "student_list.json" as shown in the following program.

```
 {
 "ID" : "001",
 "name" : "Adam",
  "age" : 12,
  "gender" : "male",
  "standard" : "six",
  "mark1" : 70,
  "mark2" : 50,
  "mark3" : 60,
  "addr" : "23 new street",
  "pincode" : 111222
}
{
"ID" : "002",
 "name" : "Amit",
  "age" : 12,
  "gender" : "male",
  "standard" : "six",
  "mark1" : 40,
  "mark2" : 50,
  "mark3" : 40,
  "addr" : "12 old street",
  "pincode" : 111222
}
{
 "ID" : "003",
 "name" : "Bob",
  "age" : 12,
  "gender" : "male",
  "standard" : "six",
  "mark1" : 60,
  "mark2" : 80,
  "mark3" : 70,
  "addr" : "10 cross street",
  "pincode" : 111222
}
{
"ID" : "004",
 "name" : "David",
  "age" : 12,
  "gender" : "male",
  "standard" : "six",
  "mark1" : 50,
  "mark2" : 70,
  "mark3" : 70,
  "addr" : "15 express avenue",
  "pincode" : 111222
}
{
"ID" : "005",
 "name" : "Esha",
  "age" : 12,
  "gender" : "female",
  "standard" : "six",
  "mark1" : 70,
  "mark2" : 60,
  "mark3" : 65,
  "addr" : "20 garden street",
  "pincode" : 111222
}
{
"ID" : "006",
 "name" : "Ganga",
  "age" : 12,
  "gender" : "female",
  "standard" : "six",
 "mark1" : 100,
 "mark2" : 95,

  "mark3" : 98,
  "addr" : "25 north street",
  "pincode" : 111222
}
{
"ID" : "007",
 "name" : "Jack",
  "age" : 13,
  "gender" : "male",
  "standard" : "six",
  "mark1" : 55,
  "mark2" : 45,
  "mark3" : 45,
  "addr" : "2 park street",
  "pincode" : 111222
}
{
"ID" : "008",
 "name" : "Leena",
  "age" : 12,
  "gender" : "female",
  "standard" : "six",
  "mark1" : 90,
  "mark2" : 85,
  "mark3" : 95,
  "addr" : "24 south street",
  "pincode" : 111222
}
{
"ID" : "009",
 "name" : "Mary",
  "age" : 13,
  "gender" : "female",
  "standard" : "six",
  "mark1" : 75,
  "mark2" : 85,

  "mark3" : 90,
  "addr" : "5 west street",
  "pincode" : 111222
}
{
"ID" : "010",
 "name" : "Peter",
  "age" : 13,
  "gender" : "female",
  "standard" : "six",
  "mark1" : 80,
  "mark2" : 85,
  "mark3" : 88,
  "addr" : "16 park avenue",
  "pincode" : 111222
}
```

Now, let us query the file to view its full records.

Query:

```
0: jdbc:drill:zk=local> select * from dfs.`/Users/../workspace/Drill-
samples/student_list.json`;
```

Result:

```
ID    name   age  gender  standard mark1  mark2   mark3   addr                pincode
001   Adam   12   male    six       70    50      60      23 new street        111222
002   Amit   12   male    six       40    50      40      12 old street        111222
003   Bob    12   male    six       60    80      70      10 cross street      111222
004   David  12   male    six       50    70      70      15 express avenue    111222
```

```
005   Esha   12   female   six      70    60       65       20 garden street    111222
006   Ganga  12   female   six      100   95       98       25 north street     111222
007   Jack   13   male     six      55    45       45       2 park street       111222
008   Leena  12   female   six      90    85       95       24 south street     111222
009   Mary   13   female   six      75    85       90       5 west street       111222
010   Peter  13   female   six      80    85       88       16 park avenue      111222
```

SQL Operators

This section will cover the operations on SQL operators using JSON.

AND Operator

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select * from dfs.`/Users/../workspace/Drill-
samples/student_list.json` where age = 12 and mark3 = 70;
```

Result:

```
ID    name   age    genderstandard mark1 mark2   mark3    addr                pincode


003   Bob    12     male   six    60     80     70     10 cross street     111222
004   David  12     male   six    50     70     70     15 express avenue   111222
```

Here, the AND operator produces the result when the condition matches to age=12 and mark3=70.

OR Operator

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select * from dfs.`/Users/../workspace/Drill-
samples/student_list.json` where ID = '007' or mark3 = 70;
```

Result:

```
ID    name   age   gender  standard mark1   mark2    mark3    addr              pincode
003   Bob    12     male   six      60      80       70     10 cross street     111222
004   David  12     male   six      50      70       70     15 express avenue    111222
007   Jack   13     male   six      55      45       45     2 park street       111222
```

Here, the OR operator produces the result if anyone condition matches from the above query.

IN Operator

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select name,age,addr from dfs.`/Users/../workspace/Drill-
samples/student_list.json` where ID in (‘001','003');
```

Result:

```
name        age          addr
Adam        12     23 new street
Bob         12     10 cross street
```

The IN operator returns result in the set condition.

Between Operator

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select name,age,addr from dfs.`/Users/../workspace/Drill-
samples/student_list.json` where mark1 between 50 and 70;
```

Result:

```
name        age    addr
Adam        12     23 new street
Bob         12     10 cross street
David       12     15 express avenue
Esha        12     20 garden street
Jack        13     2 park street
```

LIKE Operator

The Like Operator is used for pattern matching.

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select name from dfs.`/Users/../workspace/Drill-
samples/student_list.json` where name like ‘A%';
```

Result:

```
name
Adam
Amit
```

The above query returns this result, when the name first letter is starting with 'A'.

NOT Operator

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select * from dfs.`/Users/../workspace/Drill-
samples/student_list.json` where mark1 not in (80,75,70);
```

Result:

```
ID   name   age  gender  standard mark1 mark2  mark3    addr             pincode
002  Amit   12   male    six      40    50     40     12 old street      111222
003  Bob    12   male    six      60    80     70     10 cross street    111222
004  David  12   male    six      50    70     70     15 express avenue  111222
006  Ganga  12   female  six      100   95     98     25 north street    111222
007  Jack   13   male    six      55    45     45     2 park street      111222
008  Leena  12   female  six      90    85     95     24 south street    111222
```

Aggregate Functions

The aggregate functions produce a single result from a set of input values. The following table lists out the functions in further detail.

Function	Description
AVG(expression)	Averages a column of all records in a data source
COUNT(*)	Returns the number of rows that match the given criteria.
COUNT([DISTINCT] expression)	Returns the number of distinct values in the column.
MAX(expression)	Returns the largest value of the selected column.
MIN(expression)	Returns the smallest value of the selected column.
SUM(expression)	Return the sum of given column.

Avg(exp)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select avg(mark1) from dfs.`/Users/../workspace/Drill-
samples/student_list.json`;
```

Result:

```
EXPR$0

69.0
```

Here, the output is the average result of mark1 column.

COUNT(*)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select count(*) from dfs.`/Users/../workspace/Drill-
samples/student_list.json`;
```

Result:

```
EXPR$0

10
```

This count(*) function returns the total number of records

COUNT(DISTINCT(exp))

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select count(distinct(mark3)) from
dfs.`/Users/../workspace/Drill-samples/student_list.json`;
```

Result:

```
EXPR$0

9

count(distinct(mark3)) returns the no of distinct records for the column mark3.
```

MAX(exp)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select max(mark2) from dfs.`/Users/../workspace/Drill-
samples/student_list.json`;
```

Result:

```
EXPR$0

95

Max of mark2 column is 95
```

MIN(exp)

In this, there is MIN(column) function, which returns the smallest value of the selected column.

Query:

```
0: jdbc:drill:zk=local> select min(mark2) from dfs.`/Users/../workspace/Drill-
samples/student_list.json`;
```

Result:

```
EXPR$0

45
```

SUM(exp)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select sum(mark1) from dfs.`/Users/../workspace/Drill-
samples/student_list.json`;
```

Result:

```
EXPR$0

690
```

Here mark1 column values are summed and then returns the result as 690.

Statistical Function

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select stddev(mark2) from dfs.`/Users/../workspace/Drill-
samples/student_list.json`;
```

Result:

```
EXPR$0

18.020050561034015
```

Query:

```
0: jdbc:drill:zk=local> select variance(mark2) from dfs.`/Users/../workspace/Drill-
samples/student_list.json`;
```

Result:

```
EXPR$0

324.7222222222223
```

Variance of mark2 column result is returned as the output.

7. Apache Drill – Window Functions using JSON

Window functions execute on a set of rows and return a single value for each row from the query. The term window has the meaning of the set of rows for the function.

A Window function in a query, defines the window using the OVER() clause. This OVER() clause has the following capabilities:

- Defines window partitions to form groups of rows. (PARTITION BY clause)
- Orders rows within a partition. (ORDER BY clause)

Aggregate Window Functions

The Aggregate window function can be defined over a partition by and order by clause.

Avg()

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select mark1,gender,avg(mark1)  over (partition by gender )
as avgmark1 from dfs.`/Users/../workspace/Drill-samples/student_list.json`;
```

Result:

```
mark1       gender       avgmark1


70          female       83.0
100         female       83.0
90          female       83.0
75          female       83.0
80          female       83.0
70          male         55.0
40          male         55.0
60          male         55.0
50          male         55.0
55          male         55.0
```

This result shows that partition by clause is used for the gender column. So, it takes the average of mark1 from female gender which is 83.0 and then replaces that value to all the male and female gender. The mark1 avg result is now 55.0 and hence it replaces the same to all genders.

Count(*)

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select name, gender, mark1, age, count(*) over(partition by
age) as cnt from dfs.`/Users/../workspace/Drill-samples/student_list.json`;
```

Result:

```
name         gender        mark1  age    cnt


Adam         male          70     12     7
Amit         male          40     12     7
Bob          male          60     12     7
David        male          50     12     7
Esha         female        70     12     7
Ganga        female        100    12     7
Leena        female        90     12     7
Jack         male          55     13     3
Mary         female        75     13     3
Peter        female        80     13     3
```

Here, there are two age groups 12 and 13. The age count of 12 is for 7 students and 13 age count is for 3 students. Hence count(*) over partition by age replaces 7 for 12 age group and 3 for 13 age group.

MAX()

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select name,age,gender,mark3,max(mark3) over (partition by
gender) as maximum from dfs.`/Users/../workspace/Drill-samples/student_list.json`;
```

Result:

```
name         age    gender     mark3     maximum


Esha         12     female        65     98
Ganga        12     female        98     98
Leena        12     female        95     98
Mary         13     female        90     98
Peter        13     female        88     98
Adam         12     male          60     70
```

```
Amit        12     male         40     70
Bob         12     male         70     70
David       12     male         70     70
Jack        13     male         45     70
```

From the above query, maximum mark3 is partitioned by gender, hence female gender max mark 98 is replaced to all female students and male gender max mark 70 is replaced to all male students.

MIN()

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select mark2,min(mark2) over (partition by age ) as minimum
from dfs.`/Users/../workspace/Drill-samples/student_list.json`;
```

Result:

```
mark2       minimum
50          50
50          50
80          50
70          50
60          50
95          50
85          50
45          45
85          45
85          45
```

SUM()

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select name,age,sum(mark1+mark2) over (order by age ) as
summation from dfs.`/Users/../workspace/Drill-samples/student_list.json`;
```

Result:

```
name  age    summation
Adam  12     970
Amit  12     970
```

```
Bob   12     970
David12     970
Esha  12     970
Ganga12     970
Leena12     970
Jack  13     1395
Mary  13     1395
Peter13     1395
```

Here mark1+mark2 result is replaced separately to each male and female student.

Ranking Window Functions

Following is the table listed out with ranking window functions.

Function	Return Type	Description
CUME_DIST()	DOUBLE	Calculates the relative rank of the current row within a window partition (number of rows preceding or peer with current row) / (total rows in the window partition)
DENSE_RANK()	BIGINT	Rank of a value in a group of values based on the ORDER BY expression and the OVER clause. Each value is ranked within its partition. Rows with equal values receive the same rank. If two or more rows have the same rank then no gaps in the rows.
NTILE()	INTEGER	The NTILE window function divides the rows for each window partition, as equally as possible, into a specified number of ranked groups.
PERCENT_RANK()	DOUBLE	Calculates the percent rank of the current row using the following formula: (x - 1) / (number of rows in window partition - 1) where x is the rank of the current row.
RANK()	BIGINT	The RANK window function determines the rank of a value in a group of values. For example, if two rows are ranked 1, the next rank is 3.
ROW_NUMBER()	BIGINT	Gives unique row numbers for the rows in a group.

CUME_DIST()

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select name,age,gender,cume_dist() over (order by age) as
relative_rank from dfs.`/Users/../workspace/Drill-samples/student_list.json`;
```

Result:

```
name  age    gender relative_rank


Adam  12     male   0.7
Amit  12     male   0.7
Bob   12     male   0.7
David 12     male   0.7
Esha  12     female 0.7
Ganga 12     female 0.7
Leena 12     female 0.7
Jack  13     male   1.0
Mary  13     female 1.0
Peter 13     female 1.0
```

Dense_Rank()

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select mark1,mark2,mark3,dense_rank() over (order by age) as
denserank from dfs.`/Users/../workspace/Drill-samples/student_list.json`;
```

Result:

```
mark1  mark2   mark3  dense_rank


70     50      60     1
40     50      40     1
60     80      70     1
50     70      70     1
70     60      65     1
100    95      98     1
90     85      95     1
55     45      45     2
```

```
75    85     90     2
80    85     88     2
```

NTILE()

The NTILE window function requires the ORDER BY clause in the OVER clause.

Query:

```
0: jdbc:drill:zk=local> select name,gender,ntile(3) over (order by gender) as
row_partition from dfs.`/Users/../workspace/Drill-samples/student_list.json`;
```

Result:

```
name  gender row_partition


Esha  female 1
Ganga female 1
Leena female 1
Mary  female 1
Peter female 2
Adam  male   2
Amit  male   2
Bob   male   3
David male   3
Jack  male   3
```

Percent_rank

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select name,age,percent_rank() over (order by age) as
percentrank from dfs.`/Users/../workspace/Drill-samples/student_list.json`;
```

Result:

```
+--------+------+---------------------+
|  name  | age  |     percentrank     |
+--------+------+---------------------+
| Adam   | 12   | 0.0                 |
| Amit   | 12   | 0.0                 |
| Bob    | 12   | 0.0                 |
| David  | 12   | 0.0                 |
| Esha   | 12   | 0.0                 |
| Ganga  | 12   | 0.0                 |
| Leena  | 12   | 0.0                 |
| Jack   | 13   | 0.7777777777777778  |
| Mary   | 13   | 0.7777777777777778  |
| Peter  | 13   | 0.7777777777777778  |
+--------+------+---------------------+
```

Rank()

The ORDER BY expression in the OVER clause determines the value.

Query:

```
0: jdbc:drill:zk=local> select name,age,rank() over (order by age) as percentrank
from dfs.`/Users/../workspace/Drill-samples/student_list.json`;
```

Result:

```
+--------+------+--------------+
|  name  | age  | percentrank  |
+--------+------+--------------+
| Adam   | 12   | 1            |
| Amit   | 12   | 1            |
| Bob    | 12   | 1            |
| David  | 12   | 1            |
| Esha   | 12   | 1            |
| Ganga  | 12   | 1            |
| Leena  | 12   | 1            |
| Jack   | 13   | 8            |
| Mary   | 13   | 8            |
| Peter  | 13   | 8            |
+--------+------+--------------+
```

Row_number()

The ORDER BY expression in the OVER clause determines the number. Each value is ordered within its partition. Rows with equal values for the ORDER BY expressions receive different row numbers non-deterministically.

Query:

```
select *,row_number() over (order by age) as rownumber from
dfs.`/Users/../workspace/Drill-samples/student_list.json`;
```

Result:

```
+------+-------+-----+---------+----------+--------+--------+--------+------------
--------+----------+------------+

|  ID  |  name  | age  | gender  | standard  | mark1  | mark2  | mark3  |        addr
| pincode  | rownumber  |

+------+-------+-----+---------+----------+--------+--------+--------+------------
--------+----------+------------+

| 001  | Adam    | 12   | male    | six       | 70     | 50     | 60     | 23 new
street       | 111222   | 1           |

| 002  | Amit    | 12   | male    | six       | 40     | 50     | 40     | 12 old
street       | 111222   | 2           |

| 003  | Bob     | 12   | male    | six       | 60     | 80     | 70     | 10 cross
street     | 111222   | 3           |

| 004  | David   | 12   | male    | six       | 50     | 70     | 70     | 15 express
avenue   | 111222   | 4           |

| 005  | Esha    | 12   | female  | six       | 70     | 60     | 65     | 20 garden
street    | 111222   | 5           |

| 006  | Ganga   | 12   | female  | six       | 100    | 95     | 98     | 25 north
street     | 111222   | 6           |

| 008  | Leena   | 12   | female  | six       | 90     | 85     | 95     | 24 south
street     | 111222   | 7           |

| 007  | Jack    | 13   | male    | six       | 55     | 45     | 45     | 2 park
street       | 111222   | 8           |

| 009  | Mary    | 13   | female  | six       | 75     | 85     | 90     | 5 west
street       | 111222   | 9           |

| 010  | Peter   | 13   | female  | six       | 80     | 85     | 88     | 16 park
avenue      | 111222   | 10          |

+------+-------+-----+---------+----------+--------+--------+--------+------------
--------+----------+------------+
```

8. Apache Drill – Querying Complex Data

In this chapter, we will discuss in detail about which all composite data types does Apache Drill supports.

- **Array -** An array is a repeated list of values. A value in an array can be a scalar type, such as string or int, or an array can be a complex type, such as a map or another array.

- **Map -** A map is a set of name/value pairs. A value in a map can be a scalar type, such as string or int, or a complex type, such as an array or another map.

Apache Drill uses map and array data types internally for reading complex and nested data structures from data sources.

FLATTEN

FLATTEN separates the elements in a repeated field into individual records.

Syntax:

```
FLATTEN(x)
```

Where,

x - JSON array.

Create a JSON file named "array.json" as shown in the following program.

```
{
"num1" : 10,
"num2" : [10,20,30],
"num3" : " simple json array",
"num4" : 50.5
}
```

Now we can execute this query in Drill.

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select *,flatten(num2) as flatten from
dfs.`/Users/../workspace/Drill-samples/array.json`;
```

Result:

```
num1        num2              num3                num4         flatten


10          [10,20,30]        simple json array   50.5         10

10          [10,20,30]        simple json array   50.5         20

10          [10,20,30]        simple json array   50.5         30
```

KVGEN

This function returns a list of the keys that exist in the map.

Syntax:

```
KVGEN(column name)
```

Create a simple JSON map file named "student_map" as shown in the following program.

```
{
"student_ID" : {
"001" : "Adam",
"002" : "Amit"
                        }
}
{
"student_ID" : {
"003" : "Bob",
"004" : "David"
                        }
}
```

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select kvgen(student_ID) from dfs.`/Users/../workspace/Drill-
samples/student_map.json`;
```

Result:

```
EXPR$0


[{"key":"001","value":"Adam"},{"key":"002","value":"Amit"}]

[{“key":"003","value":"Bob"},{"key":"004","value":"David"}]
```

REPEATED_COUNT

This function counts the values in an array.

Syntax:

```
REPEATED_COUNT (array)
```

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select REPEATED_COUNT(num2) from
dfs.`/Users/../workspace/Drill-samples/array.json`;
```

Result:

```
EXPR$0

3
```

REPEATED CONTAINS

Searches for a keyword in an array. If the keyword is present in an array, the result will be true otherwise false.

Syntax:

```
REPEATED_CONTAINS(array_name, keyword)
```

The array_name is a simple array. The following program shows the query for this function:

```
0: jdbc:drill:zk=local> select REPEATED_CONTAINS(num2,10) from
dfs.`/Users/../workspace/Drill-samples/array.json`;
```

Result:

```
true
```

Query:

```
0: jdbc:drill:zk=local> select REPEATED_CONTAINS(num2,40) from
dfs.`/Users/deiva/workspace/Drill-samples/array.json`;
```

Result:

```
false
```

9. Apache Drill – Data Definition Statements

This section will cover Data definition statements. Let's go through each of these commands in detail.

Create Statement

You can create tables in Apache Drill by using the following two CTAS commands.

Method 1

Syntax:

```
CREATE TABLE name [ (column list) ] AS query;
```

Where,

Query - select statement.

Method 2

Syntax:

```
CREATE TABLE name [ ( <column list> ) ] [ PARTITION BY ( <column_name> [ , ... ] ) ]
AS <select statement>
```

Where,

- **name** - unique directory name
- **column list** - optional list of column names or aliases in the new table.
- **PARTITION BY** - partitions the data by the first column_name

To create a table, you should adhere to the following steps:

- Set the workspace to a writable workspace.
- You can only create new tables in df.tmp workspace. You cannot create tables using storage plugins, such as Hive and HBase.

For example:

```
"tmp": {
   "location": "/tmp",
  "writable": true,
   }
```

Example Query:

```
0: jdbc:drill:zk=local> use dfs.tmp;
```

Result:

```
+-------+--------------------------------------+
|  ok   |               summary                |
+-------+--------------------------------------+
| true  | Default schema changed to [dfs.tmp]  |
+-------+--------------------------------------+
```

Query:

```
0: jdbc:drill:zk=local> create table students as select * from
dfs.`/Users/../workspace/Drill-samples/student_list.json`;
```

Result:

```
+-----------+----------------------------+
| Fragment  | Number of records written  |
+-----------+----------------------------+
| 0_0       | 10                         |
+-----------+----------------------------+
```

To view records –

```
0: jdbc:drill:zk=local> select * from students;
```

Result:

```
+------+-------+-----+---------+----------+--------+--------+--------+------------
--------+----------+
|  ID  |  name  | age  | gender  | standard  | mark1  | mark2  | mark3  |        addr
| pincode  |
+------+-------+-----+---------+----------+--------+--------+--------+------------
--------+----------+
| 001  | Adam   | 12   | male    | six       | 70     | 50     | 60     | 23 new
street      | 111222   |
| 002  | Amit   | 12   | male    | six       | 40     | 50     | 40     | 12 old
street      | 111222   |
| 003  | Bob    | 12   | male    | six       | 60     | 80     | 70     | 10 cross
street    | 111222   |
| 004  | David  | 12   | male    | six        | 50      | 70      | 70      | 15 express
avenue  | 111222   |
| 005  | Esha   | 12   | female  | six        | 70      | 60      | 65      | 20 garden
street    | 111222   |
| 006  | Ganga  | 12   | female  | six        | 100     | 95      | 98      | 25 north
street     | 111222   |
| 007  | Jack   | 13   | male    | six        | 55      | 45      | 45      | 2 park
street       | 111222   |
| 008  | Leena  | 12   | female  | six        | 90      | 85      | 95      | 24 south
street     | 111222   |
| 009  | Mary   | 13   | female  | six        | 75      | 85      | 90      | 5 west
street       | 111222   |
| 010  | Peter  | 13   | female  | six        | 80      | 85      | 88      | 16 park
avenue      | 111222   |
+------+--------+------+---------+----------+--------+--------+--------+------------
--------+----------+
```

The following program shows the query for this function:

```
0: jdbc:drill:zk=local> create table student_new  partition by (gender) as select *
from dfs.`/Users/../workspace/Drill-samples/student_list.json`;
```

Result:

```
+-----------+---------------------------+
| Fragment  | Number of records written  |
+-----------+---------------------------+
| 0_0       | 10                         |
+-----------+---------------------------+
```

To view the records of the table –

```
0: jdbc:drill:zk=local> select * from student_new;
```

Result:

```
+------+--------+------+---------+----------+--------+--------+--------+------------
--------+----------+
|  ID  |  name  | age  | gender  | standard  | mark1  | mark2  | mark3  |        addr
| pincode  |
+------+--------+------+---------+----------+--------+--------+--------+------------
--------+----------+
| 005  | Esha    | 12   | female  | six        | 70     | 60     | 65     | 20 garden
street    | 111222    |
| 006  | Ganga   | 12   | female  | six        | 100    | 95     | 98     | 25 north
street     | 111222    |
| 008  | Leena   | 12   | female  | six        | 90     | 85     | 95     | 24 south
street     | 111222    |
| 009  | Mary    | 13   | female  | six        | 75     | 85     | 90     | 5 west
street        | 111222    |
| 010  | Peter   | 13   | female  | six        | 80     | 85     | 88     | 16 park
avenue       | 111222    |
| 001  | Adam    | 12   | male    | six        | 70     | 50     | 60     | 23 new
street        | 111222    |
| 002  | Amit    | 12   | male    | six        | 40     | 50     | 40     | 12 old
street        | 111222    |
| 003  | Bob     | 12   | male    | six        | 60     | 80     | 70     | 10 cross
street     | 111222    |
| 004  | David   | 12   | male    | six        | 50     | 70     | 70     | 15 express
avenue   | 111222    |
| 007  | Jack    | 13   | male    | six        | 55     | 45     | 45     | 2 park
street        | 111222    |
+------+--------+------+---------+----------+--------+--------+--------+------------
--------+----------+
```

Here the table records are partitioned by gender.

Alter Statement

The ALTER SYSTEM command permanently changes a system setting.

Syntax:

```
ALTER SYSTEM SET `option_name` = value;
```

To reset the system settings, use the following syntax.

```
ALTER SYSTEM RESET `option_name`;


ALTER SYSTEM RESET ALL;
```

Query:

Here is the sample query that enables the Decimal data type –

```
0: jdbc:drill:zk=local> ALTER SYSTEM SET `planner.enable_decimal_data_type` = true;
```

Result:

```
+-------+------------------------------------------+
|  ok   |                 summary                  |
+-------+------------------------------------------+
| true  | planner.enable_decimal_data_type updated.  |
+-------+------------------------------------------+
```

By default, Apache Drill disables the decimal data type. To reset all the changes, you will need to key-in the following command –

```
0: jdbc:drill:zk=local> ALTER SYSTEM RESET all;
```

Result:

```
+-------+---------------+
|  ok   |    summary    |
+-------+---------------+
| true  | ALL updated.  |
+-------+---------------+
```

Create View Statement

The CREATE VIEW command creates a virtual structure for the result set of a stored query. A view can combine data from multiple underlying data sources and provide the illusion that all of the data is from one source.

Syntax:

```
CREATE [OR REPLACE] VIEW [workspace.]view_name [ (column_name [, ...]) ] AS query;
```

Where,

- **workspace -** The location where you want the view to exist. By default, the view can be created in “dfs.tmp”.

- **view_name -** The name that you give to the view. This view must have a unique name.

Query:

```
0: jdbc:drill:zk=local> create view student_view as select * from
dfs.`/Users/../workspace/Drill-samples/student_list.json`;
```

Result:

```
+-------+-------------------------------------------------------------+
|  ok   |                           summary                           |
+-------+-------------------------------------------------------------+
| true  | View 'student_view' created successfully in 'dfs.tmp' schema  |
+-------+-------------------------------------------------------------+
```

To see the records, you can use the following query.

```
0: jdbc:drill:zk=local> select * from student_view;
```

Result:

```
+------+-------+-----+---------+----------+--------+--------+--------+------------
--------+----------+
|  ID  |  name | age | gender | standard | mark1 | mark2 | mark3 |         addr
| pincode  |
+------+-------+-----+---------+----------+--------+--------+--------+------------
--------+----------+
| 001  | Adam   | 12   | male    | six        | 70     | 50     | 60     | 23 new
street       | 111222   |
| 002  | Amit   | 12   | male    | six        | 40     | 50     | 40     | 12 old
street       | 111222   |
| 003  | Bob    | 12   | male    | six        | 60     | 80     | 70     | 10 cross
street     | 111222   |
| 004  | David  | 12   | male    | six        | 50     | 70     | 70     | 15 express
avenue   | 111222   |
| 005  | Esha   | 12   | female  | six        | 70     | 60     | 65     | 20 garden
street    | 111222   |
| 006  | Ganga  | 12   | female  | six        | 100    | 95     | 98     | 25 north
street     | 111222   |
| 007  | Jack   | 13   | male    | six        | 55     | 45     | 45     | 2 park
street       | 111222   |
| 008  | Leena  | 12   | female  | six        | 90     | 85     | 95     | 24 south
street     | 111222   |
| 009  | Mary   | 13   | female  | six        | 75     | 85     | 90     | 5 west
street       | 111222   |
| 010  | Peter  | 13   | female  | six        | 80     | 85     | 88     | 16 park
avenue      | 111222   |
+------+-------+-----+---------+----------+--------+--------+--------+------------
--------+----------+
```

Drop Table

The drop table statement is used to drop the table from a DFS storage plugin.

Syntax:

```
DROP TABLE [workspace.]name;
```

Query:

```
0: jdbc:drill:zk=local> drop table student_new;
```

Result:

```
+-------+------------------------------+
|  ok   |            summary           |
+-------+------------------------------+
| true  | Table [student_new] dropped  |
+-------+------------------------------+
```

Drop View

Similar to the table, a view can be dropped by using the following command –

```
0: jdbc:drill:zk=local> drop view student_view;
```

Result:

```
+-------+---------------------------------------------------------------------+
|  ok   |                              summary                                |
+-------+---------------------------------------------------------------------+
| true  | View [student_view] deleted successfully from schema [dfs.tmp].    |
+-------+---------------------------------------------------------------------+
```

10. Apache Drill – Querying Data

In this chapter, we will learn about how Apache Drill allows us to query plain text files such as CSV, TSV and PSV.

CSV File

Create a CSV file named "students.csv" as shown in the following program:

```
001,Adam,23 new street

002,Amit,12 old street

003,Bob,10 cross street

004,David,15 express avenue

005,Esha,20 garden street

006,Ganga,25 north street

007,Jack,2 park street

008,Leena,24 south street

009,Mary,5 west street

010,Peter,16 park avenue
```

After saving the file, you can return to the terminal again and type the following query to view that CSV file.

```
0: jdbc:drill:zk=local> select * from dfs.`/Users/../workspace/Drill-
samples/students.csv`;
```

Result:

```
+--------------------------------------+

|                columns               |

+--------------------------------------+

| ["001","Adam","23 new street"]       |

| ["002","Amit","12 old street"]       |

| ["003","Bob","10 cross street"]      |

| ["004","David","15 express avenue"]  |

| ["005","Esha","20 garden street"]    |

| ["006","Ganga","25 north street"]    |

| ["007","Jack","2 park street"]       |

| ["008","Leena","24 south street"]    |
| ["009","Mary","5 west street"]       |
| ["010","Peter","16 park avenue"]     |
+--------------------------------------+
```

From this output we can conclude that, Apache Drill recognizes each row as an array of values and returns one column for each row.

Finding Columns[n]

The **COLUMNS[n] syntax** is used to return CSV rows in a column by the column format, where n starts from 0 to n-1.

Query:

```
0: jdbc:drill:zk=local>select columns[0],columns[1],columns[2] from
dfs.`/Users/../workspace/Drill-samples/students.csv`;
```

Result:

```
+---------+---------+--------------------+
| EXPR$0  | EXPR$1  |       EXPR$2       |
+---------+---------+--------------------+
| 001     | Adam    | 23 new street      |
| 002     | Amit    | 12 old street      |
| 003     | Bob     | 10 cross street    |
| 004     | David   | 15 express avenue  |
| 005     | Esha    | 20 garden street   |
| 006     | Ganga   | 25 north street    |
| 007     | Jack    | 2 park street      |
| 008     | Leena   | 24 south street    |
| 009     | Mary    | 5 west street      |
| 010     | Peter   | 16 park avenue     |
+---------+---------+--------------------+
```

If you want to assign an alias name for columns, use the following query :

```
0: jdbc:drill:zk=local>select columns[0] as ID,columns[1] as Name,columns[2] as
Address from dfs.`/Users/../workspace/Drill-samples/students.csv`;
```

Result:

```
+------+--------+--------------------+
|  ID  |  Name  |      Address       |
+------+--------+--------------------+
| 001  | Adam   | 23 new street      |
| 002  | Amit   | 12 old street      |
| 003  | Bob    | 10 cross street    |
| 004  | David  | 15 express avenue  |
| 005  | Esha   | 20 garden street   |
| 006  | Ganga  | 25 north street    |
| 007  | Jack   | 2 park street      |
| 008  | Leena  | 24 south street    |
| 009  | Mary   | 5 west street      |
| 010  | Peter  | 16 park avenue     |
+------+--------+--------------------+
```

Create Table

Like in JSON, you can create table for plain text files. Following is a sample query:

```
0: jdbc:drill:zk=local> create table CSV as select * from
dfs.`/Users/../workspace/Drill-samples/students.csv`;
```

Result:

```
+-----------+---------------------------+
| Fragment  | Number of records written |
+-----------+---------------------------+
| 0_0       | 10                        |
+-----------+---------------------------+
```

To view the file contents, type the following query:

```
0: jdbc:drill:zk=local> select * from CSV;
```

Result:

```
+--------------------------------------+
|               columns                |
+--------------------------------------+
| ["001","Adam","23 new street"]       |
| ["002","Amit","12 old street"]       |
| ["003","Bob","10 cross street"]      |
| ["004","David","15 express avenue"]  |
| ["005","Esha","20 garden street"]    |
| ["006","Ganga","25 north street"]    |
| ["007","Jack","2 park street"]       |
| ["008","Leena","24 south street"]    |
| ["009","Mary","5 west street"]       |
| ["010","Peter","16 park avenue"]     |
+--------------------------------------+
```

TSV File

Create a TSV file named "students.tsv" as shown in the following program:

```
ID          Name                Age                 Standard            Remark
001 id  "name is Adam"  "for the age of 12"  "studying sixth std" "Having good marks"
```

Now we can execute this TSV file in Apache Drill by using the following query:

```
0: jdbc:drill:zk=local> select * from dfs.`/Users/../workspace/Drill-
samples/student.tsv`;
```

Result:

```
+------------------------------------------------------------------------------------
--------+
|                                          columns
|
+------------------------------------------------------------------------------------
--------+
| ["ID","Name","Age","Standard","Marks","Addr","pincode"]
|
| ["001 id ","name is adam","for the age of 12","studying sxith std\" \"Having good
marks"]  |
+--------------------------------------------------------------------------------------
--------+
```

Create Table

As shown in the CSV file above, you can also create a table for the TSV file.

Query:

```
0: jdbc:drill:zk=local> select * from dfs.`/Users/../workspace/Drill-
samples/student.tsv`;
```

Result:

```
+-----------+---------------------------+
| Fragment  | Number of records written  |
+-----------+---------------------------+
| 0_0       | 2                          |
+-----------+---------------------------+
1 row selected (0.347 seconds)
```

Query:

```
0: jdbc:drill:zk=local> select * from TSV;
```

Result:

```
+--------------------------------------------------------------------------------------
--------+
|                                           columns
|
+--------------------------------------------------------------------------------------
--------+
| ["ID","Name","Age","Standard","Marks","Addr","pincode"]
|
| ["001 id ","name is adam","for the age of 12","studying sxith std\" \"Having good
marks"]  |
+--------------------------------------------------------------------------------------
--------+
```

PSV (Pipe Separated Value) File

Create a psv file named “sample.tbl” as shown in the following program.

```
Tutorialspoint|Apache|Drill|article
```

Now we can execute this PSV file in Drill,

Query:

```
0: jdbc:drill:zk=local> select * from dfs.`/Users/../workspace/Drill-
samples/sample.tbl`;
```

Result:

```
+------------------------------------------------+
|                    columns                     |
+------------------------------------------------+
| ["Tutorialspoint","Apache","Drill","article"]  |
+------------------------------------------------+
```

Now, similar to the CSV and TSV files, try for yourself to create a table for PSV file.

11. Apache Drill – Querying Data Using HBase

HBase is a distributed column-oriented database built on top of the Hadoop file system. It is a part of the Hadoop ecosystem that provides random real-time read/write access to data in the Hadoop File System. One can store the data in HDFS either directly or through HBase. The following steps are used to query HBase data in Apache Drill.

How to Start Hadoop and HBase?

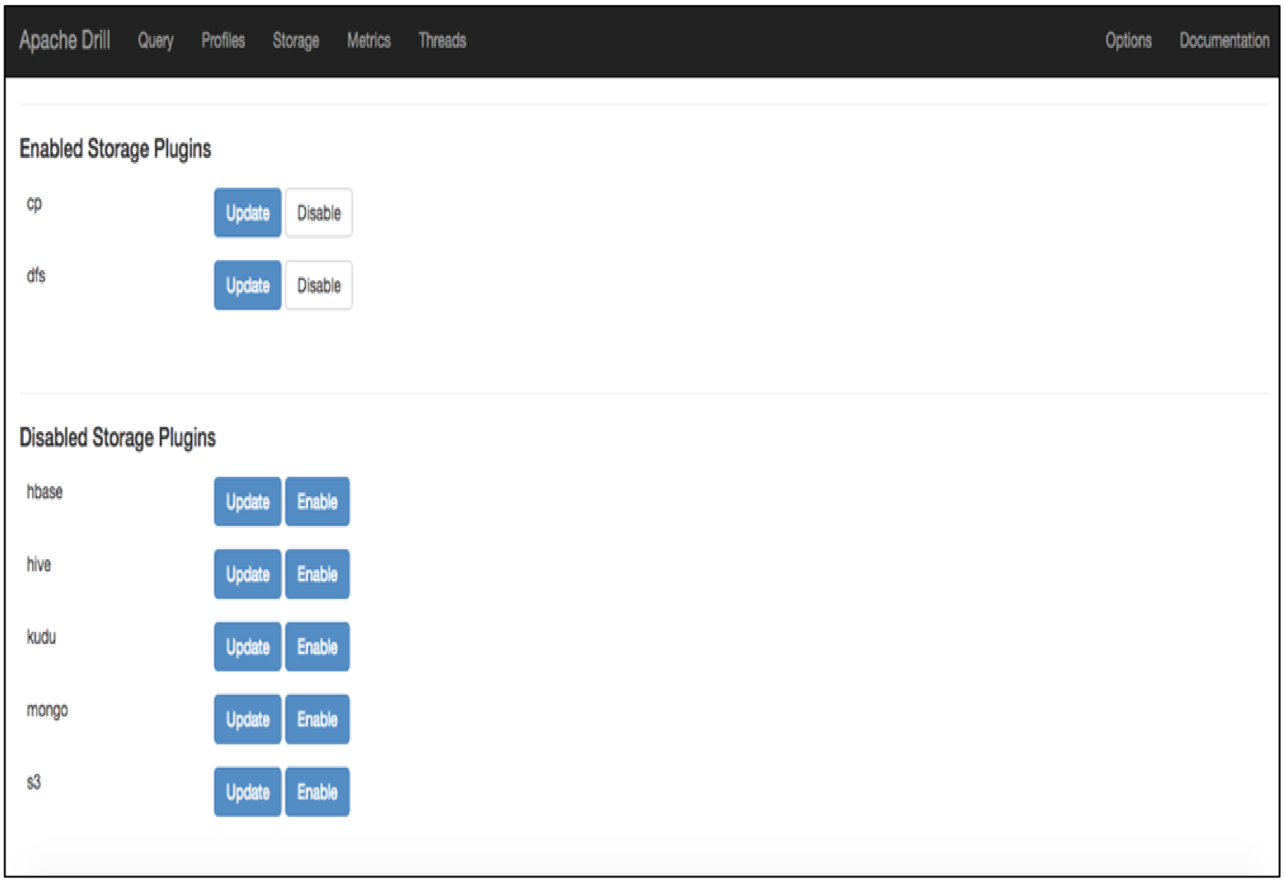
Step 1: Prerequisites

Before moving on to querying HBase data, you must need to install the following:

- Java installed version 1.7 or greater
- Hadoop
- HBase

Step 2: Enable Storage Plugin

After successful installation navigate to Apache Drill web console and select the storage menu option as shown in the following screenshot.



Then choose HBase Enable option, after that go to the update option and now you will see the response as shown in the following program.

```
{
  "type": "hbase",
  "config": {
    "hbase.zookeeper.quorum": "localhost",
    "hbase.zookeeper.property.clientPort": "2181"
  },
  "size.calculator.enabled": false,
  "enabled": true
}
```

Here the config settings "hbase.zookeeper.property.clientPort" : "2181" indicates ZooKeeper port id. In the embedded mode, it will automatically assign it to the ZooKeeper, but in the distributed mode, you must specify the ZooKeeper port id's separately. Now, HBase plugin is enabled in Apache Drill.

Step 3: Start Hadoop and HBase

After enabling the plugin, first start your Hadoop server then start HBase.

Creating a Table Using HBase Shell

After Hadoop and HBase has been started, you can start the HBase interactive shell using "hbase shell" command as shown in the following query.

```
/bin/hbase shell
```

Then you will see the response as shown in the following program.

```
hbase(main):001:0>
```

To query HBase, you should complete the following steps:

Create a Table:

Pipe the following commands to the HBase shell to create a "customer" table.

```
hbase(main):001:0> create 'customers','account','address'
```

Load Data into the Table:

Create a simple text file named "hbase-customers.txt" as shown in the following program.

```
put 'customers','Alice','account:name','Alice'
put 'customers','Alice','address:street','123 Ballmer Av'
put 'customers','Alice','address:zipcode','12345'
put 'customers','Alice','address:state','CA'
put 'customers','Bob','account:name','Bob'
put 'customers','Bob','address:street','1 Infinite Loop'
put 'customers','Bob','address:zipcode','12345'
put 'customers','Bob','address:state','CA'
put 'customers','Frank','account:name','Frank'
put 'customers','Frank','address:street','435 Walker Ct'
put 'customers','Frank','address:zipcode','12345'
put 'customers','Frank','address:state','CA'
put 'customers','Mary','account:name','Mary'
put 'customers','Mary','address:street','56 Southern Pkwy'
put 'customers','Mary','address:zipcode','12345'
put 'customers','Mary','address:state','CA'
```

Now, issue the following command in hbase shell to load the data into a table.

```
hbase(main):001:0> cat ../drill_sample/hbase/hbase-customers.txt | bin/hbase shell
```

Query:

Now switch to Apache Drill shell and issue the following command.

```
0: jdbc:drill:zk=local> select * from hbase.customers;
```

Result:

```
+-------------+---------------------+-----------------------------------------------
-----------------------------+
|   row_key   |       account       |                                  address
|
+-------------+---------------------+-----------------------------------------------
-----------------------------+
| 416C696365  | {"name":"QWxpY2U="}  |
{"state":"Q0E=","street":"MTIzIEJhbGxtZXIgQXY=","zipcode":"MTIzNDU="}       |
| 426F62      | {"name":"Qm9i"}      |
{"state":"Q0E=","street":"MSBJbmZpbml0ZSBMb29w","zipcode":"MTIzNDU="}       |
| 4672616E6B  | {"name":"RnJhbms="}  |
{"state":"Q0E=","street":"NDM1IFdhbGtlciBDdA==","zipcode":"MTIzNDU="}       |
| 4D617279    | {"name":"TWFyeQ=="}  |
{"state":"Q0E=","street":"NTYgU291dGhlcm4gUGt3eQ==","zipcode":"MTIzNDU="}  |

+------------+--------------------+--------------------------------------------
-----------------------------+
```

The output will be 4 rows selected in 1.211 seconds.

Apache Drill fetches the HBase data as a binary format, which we can convert into readable data using **CONVERT_FROM** function available in drill. Check and use the following query to get proper data from drill.

```
0: jdbc:drill:zk=local> SELECT CONVERT_FROM(row_key, 'UTF8') AS customer_id,

. . . . . . . . . . . . >         CONVERT_FROM(customers.account.name, 'UTF8') AS
customers_name,

. . . . . . . . . . . . >         CONVERT_FROM(customers.address.state, 'UTF8') AS
customers_state,

. . . . . . . . . . . . >         CONVERT_FROM(customers.address.street, 'UTF8') AS
customers_street,

. . . . . . . . . . . . >         CONVERT_FROM(customers.address.zipcode, 'UTF8') AS
customers_zipcode

. . . . . . . . . . . . > FROM hbase.customers;
```

Result:

```
+--------------+-----------------+------------------+-------------------+------------
--------+

| customer_id  | customers_name  | customers_state  | customers_street  |
customers_zipcode  |

+--------------+-----------------+------------------+-------------------+------------
--------+

| Alice        | Alice           | CA               | 123 Ballmer Av    | 12345
|

| Bob          | Bob             | CA               | 1 Infinite Loop   | 12345
|

| Frank        | Frank           | CA               | 435 Walker Ct     | 12345
|

| Mary         | Mary            | CA               | 56 Southern Pkwy  | 12345
|

+--------------+-----------------+------------------+-------------------+------------
--------+
```

12. Apache Drill – Querying Data Using Hive

Hive is a data warehouse infrastructure tool to process structured data in Hadoop. It resides on top of Hadoop to summarize Big Data, and makes querying and analyzing easy. Hive stores schema in a database and processed data into HDFS.

How to Query Hive Data in Apache Drill?

Following are the steps that are used to query Hive data in Apache Drill.

Step 1: Prerequisites

You must need to install the following components first –

- Java installed version 1.7 or greater
- Hadoop
- Hive
- ZooKeeper

Step 2: Start Hadoop, ZooKeeper and Hive

After the installation, start all the services (Hadoop, ZooKeeper and Hive) one by one in a new terminal.

Step 3: Start Hive metastore

You can start the Hive metastore using the following command –

```
hive --service metastore
```

Apache Drill uses Hive metastore service to get hive table's details.

Step 4: Start Apache Drill in Distributed Mode

To start Drill shell in a distributed mode, you can issue the following command –

```
bin/drillbit.sh start
```

Step 5: Enable Storage Plugin

Like HBase, open Apache Drill web console and choose Hive storage plugin enable option then add the following changes to hive storage plugin "update" option,

```
{
     "type": "hive",
     "enabled": false,
     "configProps": {
           "hive.metastore.uris": "thrift://localhost:9083",
           "hive.metastore.sasl.enabled": "false",
           "fs.default.name": "hdfs://localhost/"
     }
}
```

Step 6: Create a Table

Create a table in hive shell using the following command.

```
create table customers (Name string, address string) row format delimited fields
terminated by ',' stored as textfile;
```

Step 7: Load Data

Load data in the hive shell using the following command.

```
load data local inpath '/path/to/file/customers.csv' overwrite into table customers;
```

Step 8: Query Data in Drill

You can query data in the hive shell using the following command.

```
select * from hive.`customers`;
```

Result:

```
'Alice','123 Ballmer Av'
'Bob','1 Infinite Loop'
'Frank','435 Walker Ct'
'Mary','56 Southern Pkwy'
```

13. Apache Drill – Querying Parquet Files

Parquet is a columnar storage format. Apache Drill uses Parquet format for easy, fast and efficient access.

Create a Table

Before moving to create a table in parquet, you must change the Drill storage format using the following command.

```
0: jdbc:drill:zk=local> alter session set `store.format`= 'parquet';
```

Result:

```
+-------+------------------------+
|  ok   |         summary        |
+-------+------------------------+
| true  | store.format updated.  |
+——-+------------------------+
```

You can create a table using the following syntax.

```
0: jdbc:drill:zk=local> create table dfs.tmp.`/Users/../workspace` as select * from
dfs.`/Users/../workspace/Drill-samples/student_list.json`;
```

Result:

```
+-----------+---------------------------+
| Fragment  | Number of records written  |
+-----------+---------------------------+
| 0_0       | 10                         |
+-----------+---------------------------+
```

To see the table contents, type-in the following query –

```
0: jdbc:drill:zk=local> select * from dfs.tmp.`/Users/../workspace`;
```

Result:

```
+------+-------+------+---------+----------+--------+--------+--------+------------
--------+----------+
|  ID  |  name  | age  | gender  | standard  | mark1  | mark2  | mark3  |        addr
| pincode  |
+------+-------+------+---------+----------+--------+--------+--------+------------
--------+----------+
| 001  | Adam   | 12   | male    | six       | 70     | 50     | 60     | 23 new
street       | 111222   |
| 002  | Amit   | 12   | male    | six       | 40     | 50     | 40     | 12 old
street       | 111222   |
| 003  | Bob    | 12   | male    | six       | 60     | 80     | 70     | 10 cross
street     | 111222   |
| 004  | David  | 12   | male    | six       | 50     | 70     | 70     | 15 express
avenue   | 111222   |
| 005  | Esha   | 12   | female  | six       | 70     | 60     | 65     | 20 garden
street    | 111222   |
| 006  | Ganga  | 12   | female  | six       | 100    | 95     | 98     | 25 north
street     | 111222   |
| 007  | Jack   | 13   | male    | six       | 55     | 45     | 45     | 2 park
street       | 111222   |
| 008  | Leena  | 12   | female  | six       | 90     | 85     | 95     | 24 south
street     | 111222   |
| 009  | Mary   | 13   | female  | six       | 75     | 85     | 90     | 5 west
street       | 111222   |
| 010  | Peter  | 13   | female  | six       | 80     | 85     | 88     | 16 park
avenue      | 111222   |
+------+-------+------+---------+----------+--------+--------+--------+------------
--------+----------
```

14. Apache Drill – JDBC Interface

Apache Drill provides JDBC interface to connect and execute queries. We can use JDBC interface in JDBC based SQL Client like "SquirreL SQL Client" and work on all the features of drill. We can use the same JDBC interface to connect drill from our Java based application. Let us see how to connect drill and execute commands in our sample Java application using JDBC interface in this section.

Java Application

Apache Drill provides a JDBC driver as a single jar file and it is available @ **/path/to/drill/jars/jdbc-driver/drill-jdbc-all-1.6.0.jar**. The connection string to connect the drill is of the following format –

```
jdbc:drill:zk=<zk_host>:<zk_port>

jdbc:drill:zk=<zk_host>:<zk_port>/<zk_drill_path>/<zk_drillbit_name

jdbc:drill:zk=<zk_host>:<zk_port>/<zk_drill_path>/<zk_drillbit_name;schema=hive
```

Considering ZooKeeper is installed in the local system, the port configured is 2181, the drill path is "drill" and drillbit name is "drillbits1", the connection string may be among the following commands.

```
jdbc:drill:zk=localhost:2181

jdbc:drill:zk=localhost:2181/drill/dillbits1

jdbc:drill:zk=localhost:2181/drill/dillbits1;schema=hive
```

if the drill is installed in a distributed mode, we can replace the "localhost" with the list of drill installed system IP/name as shown in the following command.

```
jdbc:drill:zk=1.2.3.4:2181,5.6.7.8:2181/drill/dillbits1;schema=hive
```

The connection to drill is just like any other JDBC interface. Now, create a new maven project with "com.tutorialspoint.drill.samples" as the package name and "connect-drill" as the application name.

Then, update the following code in "App.java" file. The coding is simple and self-explanatory. The query used in the application is the default JSON file packaged into drill.

Coding:

```
package com.tutorialspoint.drill.samples;


import java.sql.*;

import java.lang.*;
```

```
public class App
{
    public static void main( String[] args ) throws SQLException,
ClassNotFoundException
    {
        // load the JDBC driver
        Class.forName("org.apache.drill.jdbc.Driver");

           // Connect the drill using zookeeper drill path
        Connection connection
=DriverManager.getConnection("jdbc:drill:zk=localhost:2181/drill/drillbits1");

        // Query drill
        Statement st = connection.createStatement();
        ResultSet rs = st.executeQuery("SELECT * from cp.`employee.json` LIMIT 3");

        // Fetch and show the result
        while(rs.next()){
              System.out.println("Name: " + rs.getString(2));
        }
    }
}
```

Now add following drill dependency tag to “pom.xml” file.

```
<dependency>
     <groupId>org.apache.drill.exec</groupId>
     <artifactId>drill-jdbc-all</artifactId>
     <version>1.1.0</version>
</dependency>
```

Now, you can compile the application by using the following command.

```
mvn clean package
```

Once the application is compiled, execute it using the following command.

```
java -cp target/connect-drill-1.0.jar:/path/to/apache-drill-1.6.0/jars/jdbc-
driver/drill-jdbc-all-1.6.0.jar com.tutorialspoint.drill.samples.App
```

The output of this application list is the name of the first three employees available in “employee.json” file and it will show in the console as shown in the following program.

Result:

```
Name: Sheri Nowmer
Name: Derrick Whelply
Name: Michael Spence
```

15. Apache Drill – Custom Function

Apache Drill has an option to create custom functions. These custom functions are reusable SQL functions that you develop in Java to encapsulate the code that processes column values during a query.

Custom functions can perform calculations and transformations that the built-in SQL operators and functions do not provide. Custom functions are called from within a SQL statement, like a regular function, and return a single value. Apache Drill has custom aggregate function as well and it is still evolving. Let us see how to create a simple custom function in this section.

IsPass Custom Function

Apache Drill provides a simple interface, "DrillSimpleFunc", which we have to implement to create a new custom function. The "DrillSimpleFunc" interface has two methods, "setup" and "eval". The "setup" method is to initialize necessary variables. "eval" method is actual method used to incorporate the custom function logic. The "eval" method has certain attributes to set function name, input and output variables.

Apache Drill provide a list of datatype to hold input and output variable like BitHolder, VarCharHolder, BigIntHolder, IntHolder, etc. We can use these datatypes to pass on information between drill and custom function. Now, let us create a new application using Maven with "com.tutorialspoint.drill.function" as the package name and "is-pass" as the library name.

```
mvn archetype:generate -DgroupId=com.tutorialspoint.drill.function -DartifactId=is-
pass -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

Here,

- -**DgroupId** - package name
- -**DartifactId** - argument

Then remove the **App.java** file and create new java file and name it as "IsPassFunc.java". This java file will hold out custom function logic. The custom function logic is to check whether the particular student is secured pass in a particular subject by checking his mark with cutoff mark. The student mark will be first input and it will change according to the record.

The second input is the cutoff mark, which will be a constant and does not change for different records. The custom function will implement "DrillSimpleFunc" interface and just check whether the given input is higher than the cutoff. If the input is higher, then if returns true, otherwise false.

The coding is as follows –

Coding: IsPassFunc.java

```
package com.tutorialspoint.drill.function;


import com.google.common.base.Strings;
import io.netty.buffer.DrillBuf;
import org.apache.drill.exec.expr.DrillSimpleFunc;
import org.apache.drill.exec.expr.annotations.FunctionTemplate;
import org.apache.drill.exec.expr.annotations.Output;
import org.apache.drill.exec.expr.annotations.Param;
import org.apache.drill.exec.expr.holders.BigIntHolder;
import org.apache.drill.exec.expr.holders.BitHolder;
import org.apache.drill.exec.expr.holders.NullableVarCharHolder;
import org.apache.drill.exec.expr.holders.VarCharHolder;


import javax.inject.Inject;


// name of the function to be used in drill
@FunctionTemplate(
        name = “ispass",
        scope = FunctionTemplate.FunctionScope.SIMPLE,
        nulls = FunctionTemplate.NullHandling.NULL_IF_NULL
)
public class IsPassFunc implements DrillSimpleFunc {


    // input - student mark
    @Param
    BigIntHolder input;


    // input - cutoff mark, constant value
    @Param(constant = true)
    BigIntHolder inputCutOff;


    // output - true / false
    @Output
    BitHolder out;

    public void setup() {
    }

    // main logic of the function. checks mark with cutoff and returns true / false.
    public void eval() {
     int mark = (int) input.value;
     int cutOffMark = (int) inputCutOff.value;

     if(mark >= cutOffMark)
            out.value = 1;
     else
            out.value = 0;
    }
}
```

Now, you can create a resource file @ **is-pass/src/main/resources/drill-module.conf** and place the following code into it.

```
drill {
  classpath.scanning {
    base.classes : ${?drill.classpath.scanning.base.classes} [
     com.tutorialspoint.drill.function.IsPassFunc
    ],
    packages : ${?drill.classpath.scanning.packages} [
     com.tutorialspoint.drill.function
    ]
  }
}
```

Apache Drill uses this configuration file to find the custom function class in the jar file. A jar file can have any number of custom function and it should be properly configured here.

Finally, add the following configuration in "pom.xml" to properly compile the custom function in maven.

pom.xml

Change the following settings in “pom.xml” file.

```
<dependencies>
     <dependency>
            <groupId>org.apache.drill.exec</groupId>
            <artifactId>drill-java-exec</artifactId>
            <version>1.1.0</version>
     </dependency>
</dependencies>

<build>
     <plugins>
            <plugin>
                   <groupId>org.apache.maven.plugins</groupId>
                   <artifactId>maven-source-plugin</artifactId>
                   <version>2.4</version>
                   <executions>
                          <execution>
                                 <id>attach-sources</id>
                                 <phase>package</phase>
                                 <goals>
                                        <goal>jar-no-fork</goal>
                                 </goals>
                          </execution>
                   </executions>
            </plugin>
            <plugin>
                   <artifactId>maven-compiler-plugin</artifactId>
                   <version>3.0</version>
                   <configuration>
                          <verbose>true</verbose>
                          <compilerVersion>1.7</compilerVersion>
                          <source>1.7</source>
                          <target>1.7</target>
                   </configuration>
            </plugin>
     </plugins>

</build>
```

After making all the changes, create a package using the following command.

```
mvn clean package
```

Maven will create the necessary jars, **is-pass-1.0.jar & is-pass-1.0-sources.jar** in the "target" folder. Now, copy the jar files and place it @ **/path/to/apache-drill/jars/3rdparty** in all the drill nodes.

After jar files are place properly in all the drillbits, restart all the drillbits, open a new drill shell and then execute the query as shown in the following program.

```
select name, ispass(mark1, 35) as is_pass from
dfs.`/Users/../Workspace/drill_sample/student_list.json` limit 3;
```

Result:

```
name  is_pass


Adam  true

Amit  true

Bob   true
```

Apache Drill custom functions are simple to create and provides great extension capabilities to drill query language.

16. Apache Drill – Contributors

Apache Drill supports many of today's best industrial applications. Some of these contributors are –

- Oracle
- IBM Netezza
- Clustrix
- Pentaho

Conclusion

Apache Drill is a schema free SQL engine and scales up to 10,000 servers or more to process petabytes of data and trillions of records in less than a second. Apache Drill uses pure data flow through the memory and extensible architecture.