HBasics

HBase is a distributed column-oriented database built on top of HDFS. HBase is the Hadoop application to use when you require real-time read/write random-access to very large datasets.

. Apache HBase is an open-source, distributed, versioned, non-relational database modeled after Google's Bigtable: A Distributed Storage System for Structured Data.

Features

- Linear and modular scalability.
- Strictly consistent reads and writes.
- Automatic and configurable sharding of tables
- Automatic failover support between RegionServers.
- Convenient base classes for backing Hadoop MapReduce jobs with Apache HBase tables.
- Easy to use Java API for client access.
- Block cache and Bloom Filters for real-time queries.
- Query predicate push down via server side Filters.
- Thrift gateway and a REST-ful Web service that supports XML, Protobuf, and binary data encoding options
- Extensible jruby-based (JIRB) shell.

Concepts

Creating a Table using HBase Shell

The **syntax** to create a table in HBase shell is shown below.

```
create '<table name>','<column family>'
```

```
EX:
create 'CustomerContactInformation',' CustomerName' , ' ContactInfo'
```

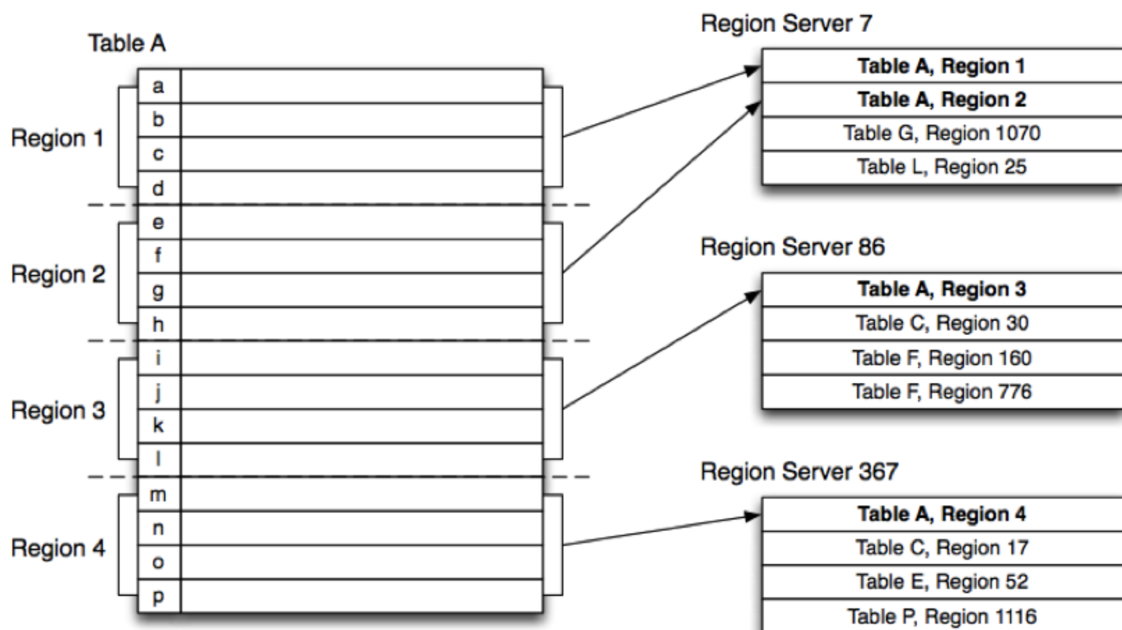
Logical View of **CustomerContactInformation table** in HBase :

Row Key	Column Family: {Column Qualifier:Version:Value}
00001	CustomerName: {‘FN’: 1383859182496:‘John’, ‘LN’: 1383859182858:‘Smith’, ‘MN’: 1383859183001:’Timothy’, ‘MN’: 1383859182915:’T’} ContactInfo: {‘EA’: 1383859183030:‘John.Smith@xyz.com’, ’SA’: 1383859183073:’1 Hadoop Lane, NY 11111’}
00002	CustomerName: {‘FN’: 1383859183103:‘Jane’, ‘LN’: 1383859183163:‘Doe’, ContactInfo: { ’SA’: 1383859185577:’7 HBase Ave, CA 22222’}

In the HBase data model *column qualifiers* are specific names assigned to your data values in order to make sure you’re able to accurately identify them.

Regions

Tables are automatically partitioned horizontally by HBase into regions. Each region comprises a subset of a table’s rows. A region is denoted by the table it belongs to.



Locking

Row updates are atomic, no matter how many row columns constitute the row-level transaction. This keeps the locking model simple.

Implementation

Components of Apache HBase Architecture

HBase architecture has 3 important components- HMaster, Region Server and ZooKeeper.

HMaster

HBase HMaster is a lightweight process that assigns regions to region servers in the Hadoop cluster for load balancing. Responsibilities of HMaster –

- Manages and Monitors the Hadoop Cluster
- Performs Administration (Interface for creating, updating and deleting tables.)
- Controlling the failover
- DDL operations are handled by the HMaster
- Whenever a client wants to change the schema and change any of the metadata operations, HMaster is responsible for all these operations.

Region Server

These are the worker nodes which handle read, write, update, and delete requests from clients. Region Server process, runs on every node in the hadoop cluster. Region Server runs on HDFS DataNode and consists of the following components –

- Block Cache – This is the read cache. Most frequently read data is stored in the read cache and whenever the block cache is full, recently used data is evicted.
- MemStore- This is the write cache and stores new data that is not yet written to the disk. Every column family in a region has a MemStore.
- Write Ahead Log (WAL) is a file that stores new data that is not persisted to permanent storage.
- HFile is the actual storage file that stores the rows as sorted key values on a disk.

Zookeeper

ZooKeeper service keeps track of all the region servers that are there in an HBase cluster- tracking information about how many region servers are there and which region servers are holding which DataNode.

Various services that Zookeeper provides include –

- Establishing client communication with region servers.
- Tracking server failure and network partitions.
- Maintain Configuration Information
- Provides ephemeral nodes, which represent different region servers.

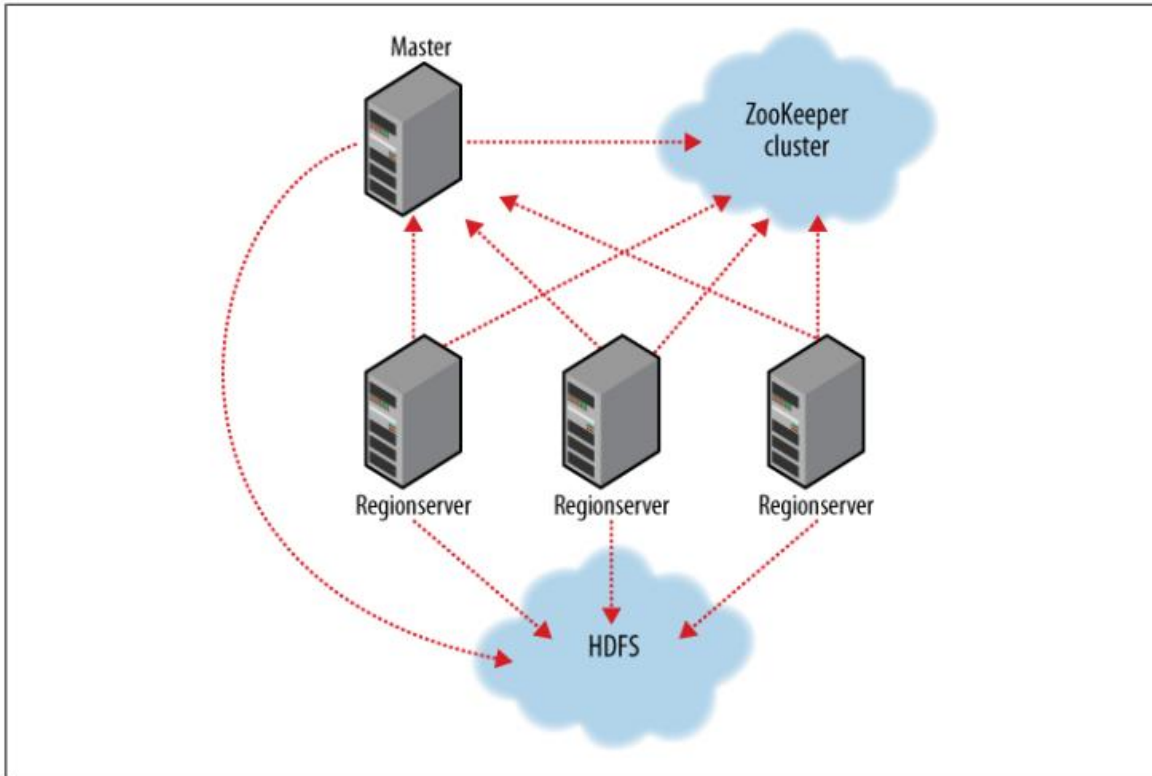


Figure 13-1. HBase cluster members

Clients

There are a number of client options for interacting with an HBase cluster.Creating table and inserting data in HBase table are shown in the following program.

```
public class ExampleClient

{

        public static void main(String[] args) throws IOException

        {

                Configuration config = HBaseConfiguration.create();

                // Create table

                HBaseAdmin admin = new HBaseAdmin(config);

                HTableDescriptor htd = new HTableDescriptor("test");
```

```
            HColumnDescriptor hcd = new HColumnDescriptor("data");
            htd.addFamily(hcd);
            admin.createTable(htd);

            byte [] tablename = htd.getName();

            // Run some operations -- a put
            HTable table = new HTable(config, tablename);
            byte [] row1 = Bytes.toBytes("row1");
            Put p1 = new Put(row1);
            byte [] databytes = Bytes.toBytes("data");
            p1.add(databytes, Bytes.toBytes("FN"), Bytes.toBytes("value1"));
            table.put(p1);
        }
}
```

Map Reduce:

TableMapper:

Hbase TableMapper is an abstract class extending Hadoop Mapper.

The source can be found at :
HBASE_HOME/src/java/org/apache/hadoop/hbase/mapreduce/TableMapper.java

```
package org.apache.hadoop.hbase.mapreduce;

import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.io.ImmutableBytesWritable;
import org.apache.hadoop.mapreduce.Mapper;

public abstract class TableMapper<KEYOUT, VALUEOUT>
extends Mapper<ImmutableBytesWritable, Result, KEYOUT, VALUEOUT> {

}
```

Notice how TableMapper parameterizes Mapper class.

Param	class	comment
KEYIN (k1)	ImmutableBytesWritable	fixed. This is the row_key of the current row being processed
VALUEIN (v1)	Result	fixed. This is the value (result) of the row
KEYOUT (k2)	user specified	customizable
VALUEOUT (v2)	user specified	customizable

TableReducer

src :
HBASE_HOME/src/java/org/apache/hadoop/hbase/mapreduce/TableReducer.java

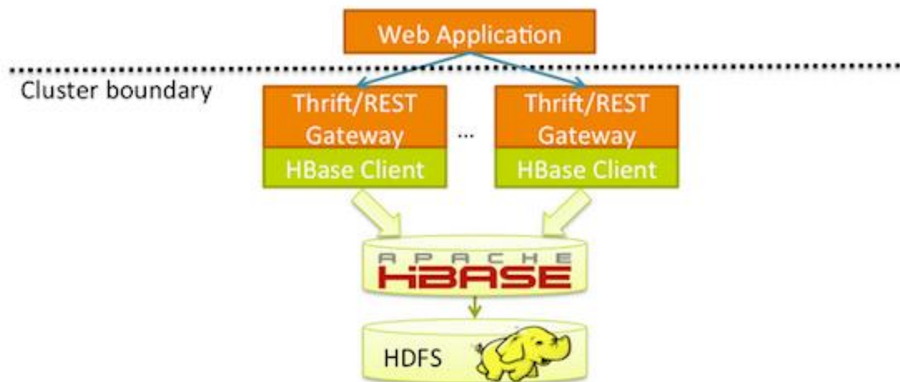
```
package org.apache.hadoop.hbase.mapreduce;


import org.apache.hadoop.io.Writable;
import org.apache.hadoop.mapreduce.Reducer;


public abstract class TableReducer<KEYIN, VALUEIN, KEYOUT>
extends Reducer<KEYIN, VALUEIN, KEYOUT, Writable> {
}
```

Lets look at the parameters:

Param	Class	Comment
KEYIN (k2 – same as mapper keyout)	user-specified (same class as K2 ouput from mapper)	
VALUEIN(v2 – same as mapper valueout)	user-specified (same class as V2 ouput from mapper)	
KEYIN (k3)	user-specified	
VALUEOUT (k4)	must be Writable	

TableReducer can take any KEY2 / VALUE2 class and emit any KEY3 class, and a Writable VALUE4 class.

Interfaces:



Avro, REST, and Thrift HBase ships with Avro, REST, and Thrift interfaces. These are useful when the interacting application is written in a language other than Java. In all cases, a Java server hosts an instance of the HBase client brokering application Avro, REST, and Thrift requests in and out of the HBase cluster.

Loading Data :

let's assume that there are billions of individual observations to be loaded. This kind of import is normally an extremely complex and long-running database operation, but MapReduce and HBase's distribution model allow us to make full use of the cluster. Copy the raw input data onto HDFS and then run a MapReduce job that can read the input and write to HBase.

Web Queries :

To implement the web application, we will use the HBase Java API directly. Here it becomes clear how important your choice of schema and storage format is.

Hive

Hive, a framework for data warehousing on top of Hadoop. Hive was created to make it possible for analysts with strong SQL skills (but meager Java programming skills) to run queries on the huge volumes of data that Facebook stored in HDFS.

Today, Hive is a successful Apache project used by many organizations as a general-purpose, scalable data processing platform.

Of course, SQL isn't ideal for every big data problem—it's not a good fit for building complex machine learning algorithms, for example—but it's great for many analyses, and it has the huge advantage of being very well known in the industry.

Installing Hive

In normal use, Hive runs on your workstation and converts your SQL query into a series of MapReduce jobs for execution on a Hadoop cluster. Hive organizes data into tables, which provide a means for attaching structure to data stored in HDFS. Metadata— such as table schemas—is stored in a database called the metastore.

Installation of Hive is straightforward. Java 6 is a prerequisite; and on Windows, you will need Cygwin, too.

Download a release at **http://hive.apache.org/releases.html**, and unpack the tarball in a suitable place on your workstation:

% tar xzf hive-x.y.z-dev.tar.gz

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

% export HIVE_INSTALL=/home/tom/hive-x.y.z-dev

% export PATH=$PATH:$HIVE_INSTALL/bin

Now type hive to launch the Hive shell:

% hive hive>

The Hive Shell

The shell is the primary way that we will interact with Hive, by issuing commands in HiveQL. HiveQL is Hive's query language, a dialect of SQL. It is heavily influenced by MySQL.

Check that it is working by listing its tables:

hive> SHOW TABLES;
OK
Time taken: 10.425 seconds

Like SQL, HiveQL is generally case insensitive.

The database stores its files in a directory called **metastore_db**, which is relative to where you ran the hive command from.

You can also run the Hive shell in non-interactive mode. The -f option runs the commands in the specified file **script.q** as follows.

In this example: % hive -f script.q

In both interactive and non-interactive mode, Hive will print information to standard error—such as the time taken to run a query—during the course of operation. You can suppress these messages using the -S option at launch time, which has the effect of only showing the output result for queries:

% hive -S 'SELECT * FROM dummy'

An Example

Let's see how to use Hive to run a query on the weather dataset.

We create a table to hold the weather data using the CREATE TABLE statement:

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

Hive expects there to be three fields in each row, corresponding to the table columns, with fields separated by tabs, and rows by newlines.

Next we can populate Hive with the data.

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

Running this command tells Hive to put the specified local file in its warehouse directory.

Thus, the files for the records table are found in the /user/hive/warehouse/records directory on the local filesystem:

% ls /user/hive/warehouse/records/ sample.txt

Now that the data is in Hive, we can run a query against it:
hive> SELECT year, MAX(temperature)
> FROM records
> WHERE temperature != 9999
> AND (quality = 0 OR quality = 1 OR quality = 4 OR quality = 5 OR quality = 9)
> GROUP BY year;
Output:
1949 111
1950 22

Hive transforms this query into a MapReduce job, which it executes on our behalf, then prints the results to the console.

Hive Services

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

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

hiveserver:

Runs Hive as a server exposing a Thrift service, enabling access from a range of clients written in different languages. Applications using the Thrift, JDBC, and ODBC connectors need to run a Hive server to communicate with Hive. Set the HIVE_PORT environment variable to specify the port the server will listen on (defaults to 10,000).

hwi:

The Hive Web Interface. **% hive --service hwi**

jar:

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

metastore:

By default, the metastore is run in the same process as the Hive service. Using this service, it is possible to run the metastore as a standalone (remote) process. Set the METASTORE_PORT environment variable to specify the port the server will listen on.

Hive clients:

If you run Hive as a server (hive --service hiveserver), then there are a number of different mechanisms for connecting to it from applications. The relationship between Hive clients and Hive services is illustrated as follows.

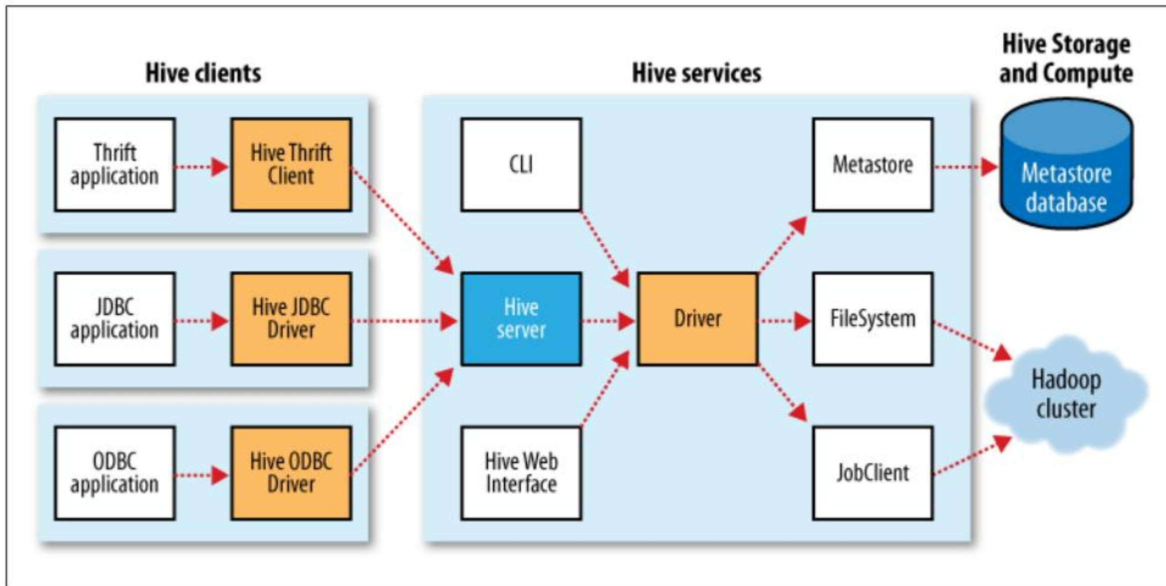


Figure 12-1. Hive architecture

Thrift Client

The Hive Thrift Client makes it easy to run Hive commands from a wide range of programming languages. Thrift bindings for Hive are available for C++, Java, PHP, Python, and Ruby.

JDBC Driver

Hive provides a Type 4

ODBC Driver:

The Hive ODBC Driver allows applications that support the ODBC protocol to connect to Hive.

The Metastore :

The metastore is the central repository of Hive metadata. The metastore is divided into two pieces: a service and the backing store for the data.

Embedded Mode

In this mode, the metastore uses a Derby database, and both the database and the metastore service are embedded in the main HiveServer process. Both are started for you when you start the HiveServer process.

It can support only one active user at a time and is not certified for production use.

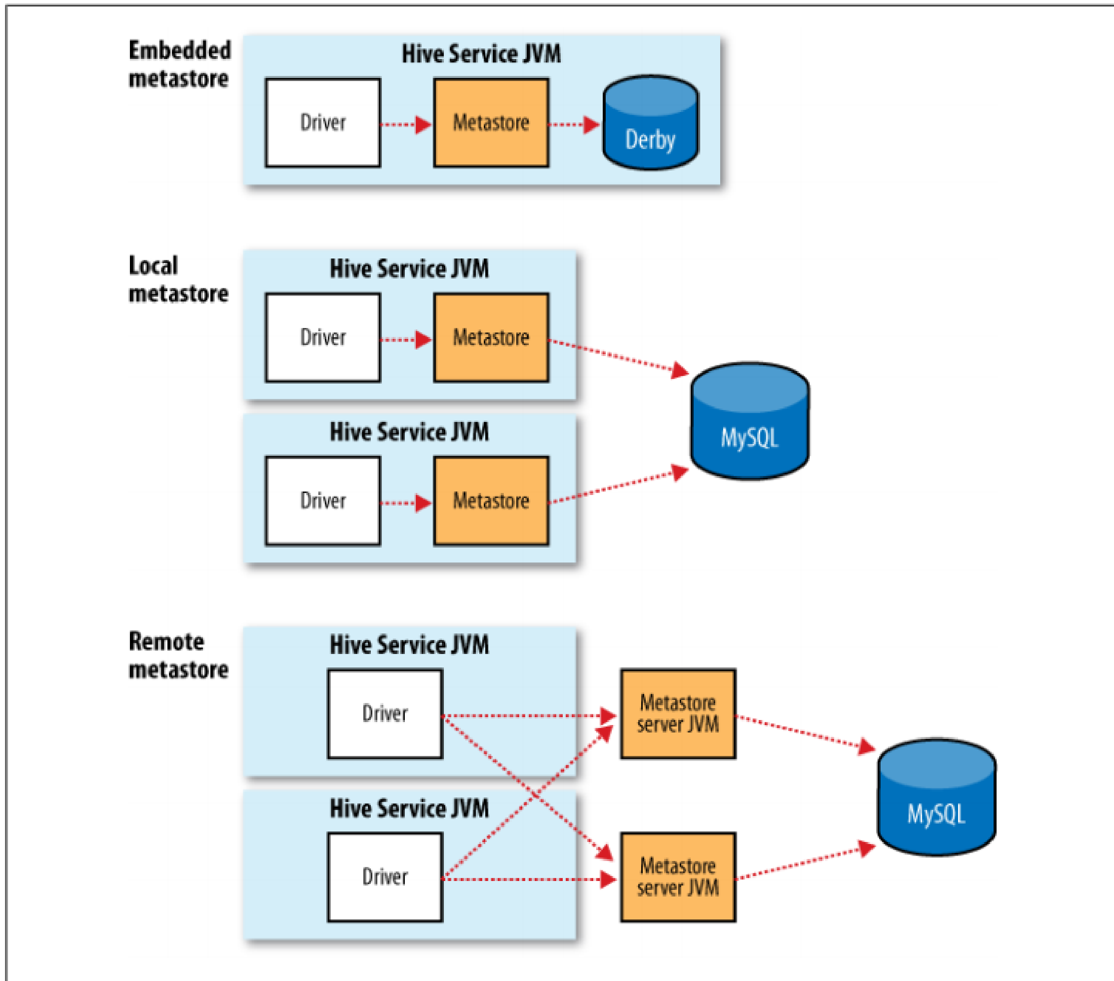


Figure 12-2. Metastore configurations

Local Mode:

In Local mode, the Hive metastore service runs in the same process as the main HiveServer process, but the metastore database runs in a separate process, and can be on a separate host. The embedded metastore service communicates with the metastore database over JDBC.

Remote Metastore

there's another metastore configuration called a remote metastore, where one or more metastore servers run in separate processes to the Hive service. This brings better manageability and security.

Table 12-1. Important metastore configuration properties

Property name	Type	Default value	Description
hive.metastore.warehouse.dir	URI	/user/hive/warehouse	The directory relative to fs.default.name where managed tables are stored.
hive.metastore.local	boolean	true	Whether to use an embedded metastore server (true), or connect to a remote instance (false). If false, then hive.metastore.uris must be set.
hive.metastore.uris	comma-separated URIs	Not set	The URIs specifying the remote metastore servers to connect to. Clients connect in a round-robin fashion if there are multiple remote servers.
javax.jdo.option.ConnectionURL	URI	jdbc:derby:;databaseName=metastore_db;create=true	The JDBC URL of the metastore database.
javax.jdo.option.ConnectionDriverName	String	org.apache.derby.jdbc.EmbeddedDriver	The JDBC driver classname.
javax.jdo.option.ConnectionUserName	String	APP	The JDBC user name.
javax.jdo.option.ConnectionPassword	String	mine	The JDBC password.

MySQL is a popular choice for the standalone metastore. In this case, **javax.jdo.option.ConnectionURL** is set to **jdbc:mysql://host/dbname?createDatabaseIfNotExist=true**, and **javax.jdo.option.ConnectionDriverName** is set to **com.mysql.jdbc.Driver**.

Comparison with Traditional Databases :

Schema on Read Versus Schema on Write

- In a traditional database, a table's schema is enforced at data load time. If the data being loaded doesn't conform to the schema, then it is rejected. This design is sometimes called **schema on write**, since the data is checked against the schema when it is written into the database.
- Hive, on the other hand, doesn't verify the data when it is loaded, but rather when a query is issued. This is called schema on read.
- There are trade-offs between the two approaches. Schema on read makes for a very fast initial load. The load operation is just a file copy or move.
- Schema on write makes query time performance faster, since the database can index columns and perform compression on the data. The trade-off, however, is that it takes longer to load data into the database.

Updates, Transactions, and Indexes

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

Hive Ql

Table 12-2. A high-level comparison of SQL and HiveQL

Feature	SQL	HiveQL	References
Updates	UPDATE, INSERT, DELETE	INSERT OVERWRITE TABLE (populates whole table or partition)	"INSERT OVERWRITE TABLE" on page 438, "Updates, Transactions, and Indexes" on page 422
Transactions	Supported	Not supported	
Indexes	Supported	Not supported	
Latency	Sub-second	Minutes	
Data types	Integral, floating point, fixed point, text and binary strings, temporal	Integral, floating point, boolean, string, array, map, struct	"Data Types" on page 424
Functions	Hundreds of built-in functions	Dozens of built-in functions	"Operators and Functions" on page 426
Multitable inserts	Not supported	Supported	"Multitable insert" on page 439
Create table as select	Not valid SQL-92, but found in some databases	Supported	"CREATE TABLE...AS SELECT" on page 440
Select	SQL-92	Single table or view in the FROM clause. SORT BY for partial ordering. LIMIT to limit number of rows returned.	"Querying Data" on page 441
Joins	SQL-92 or variants (join tables in the FROM clause, join condition in the WHERE clause)	Inner joins, outer joins, semi joins, map joins. SQL-92 syntax, with hinting.	"Joins" on page 443
Subqueries	In any clause. Correlated or noncorrelated.	Only in the FROM clause. Correlated subqueries not supported	"Subqueries" on page 446
Views	Updatable. Materialized or nonmaterialized.	Read-only. Materialized views not supported	"Views" on page 447
Extension points	User-defined functions. Stored procedures.	User-defined functions. MapReduce scripts.	"User-Defined Functions" on page 448, "MapReduce Scripts" on page 442

Data Types :

Table 12-3. Hive data types

Category	Type	Description	Literal examples
Primitive	TINYINT	1-byte (8-bit) signed integer, from -128 to 127	1
	SMALLINT	2-byte (16-bit) signed integer, from -32,768 to 32,767	1
	INT	4-byte (32-bit) signed integer, from -2,147,483,648 to 2,147,483,647	1
	BIGINT	8-byte (64-bit) signed integer, from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	1
	FLOAT	4-byte (32-bit) single-precision floating-point number	1.0
	DOUBLE	8-byte (64-bit) double-precision floating-point number	1.0
	BOOLEAN	true/false value	TRUE
	STRING	Character string	'a', "a"
	BINARY	Byte array	Not supported
	TIMESTAMP	Timestamp with nanosecond precision	1325502245000, '2012-01-02 03:04:05.123456789'
Complex	ARRAY	An ordered collection of fields. The fields must all be of the same type.	array(1, 2) [a]
	MAP	An unordered collection of key-value pairs. Keys must be primitives; values may be any type. For a particular map, the keys must be the same type, and the values must be the same type.	map('a', 1, 'b', 2)
	STRUCT	A collection of named fields. The fields may be of different types.	struct('a', 1, 1.0) [b]

Operators and Functions :

The usual set of SQL operators is provided by Hive: relational operators (such as x = 'a' for testing equality, x IS NULL for testing nullity, x LIKE 'a%' for pattern matching), arithmetic operators (such as x + 1 for addition), and logical operators (such as x OR y for logical OR).

Hive comes with a large number of built-in functions—too many to list here—divided into categories including mathematical and statistical functions, string functions, date functions (for operating on

string representations of dates), conditional functions, ag gregate functions, and functions for working with XML (using the xpath function) and JSON.