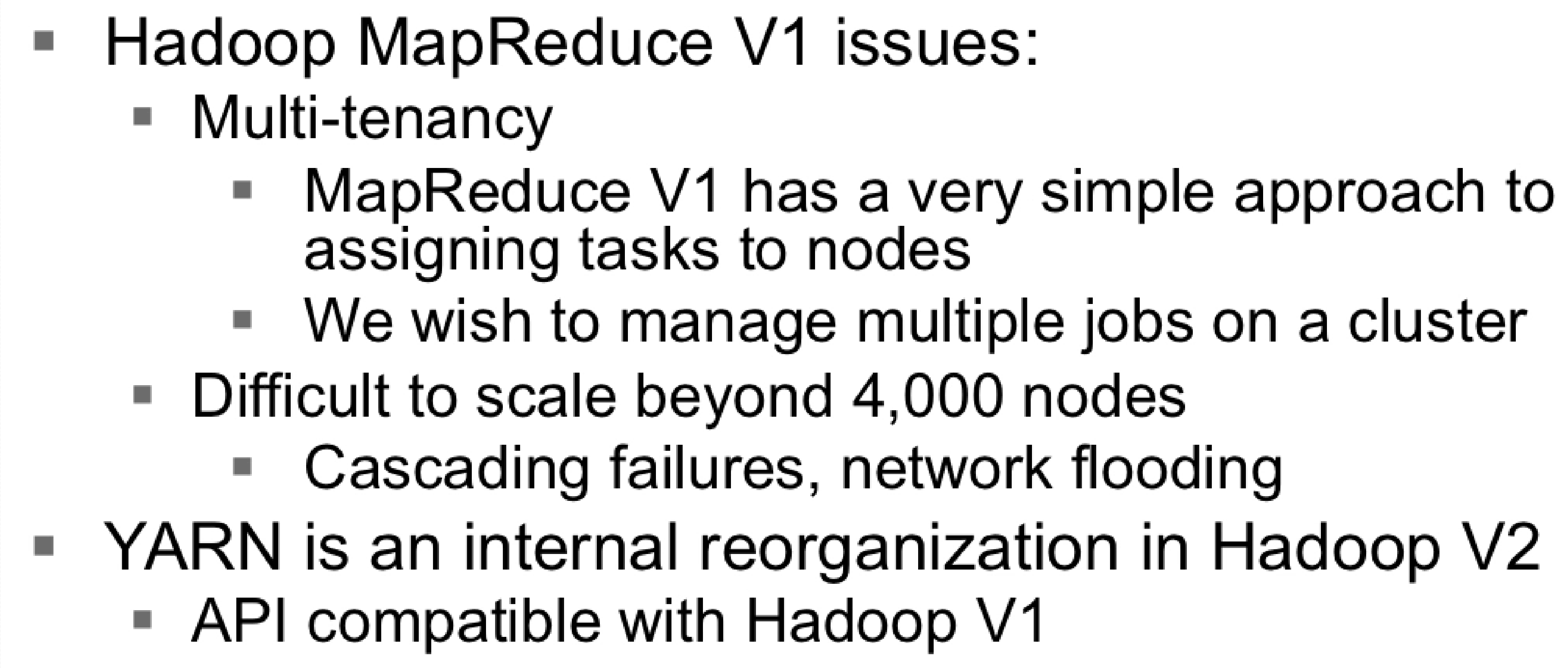
CLASS 2.
Hadoop Ecosystem. **Hive**.

CSCI 6830

BIG DATA ANALYTICS WITH HADOOP AND R

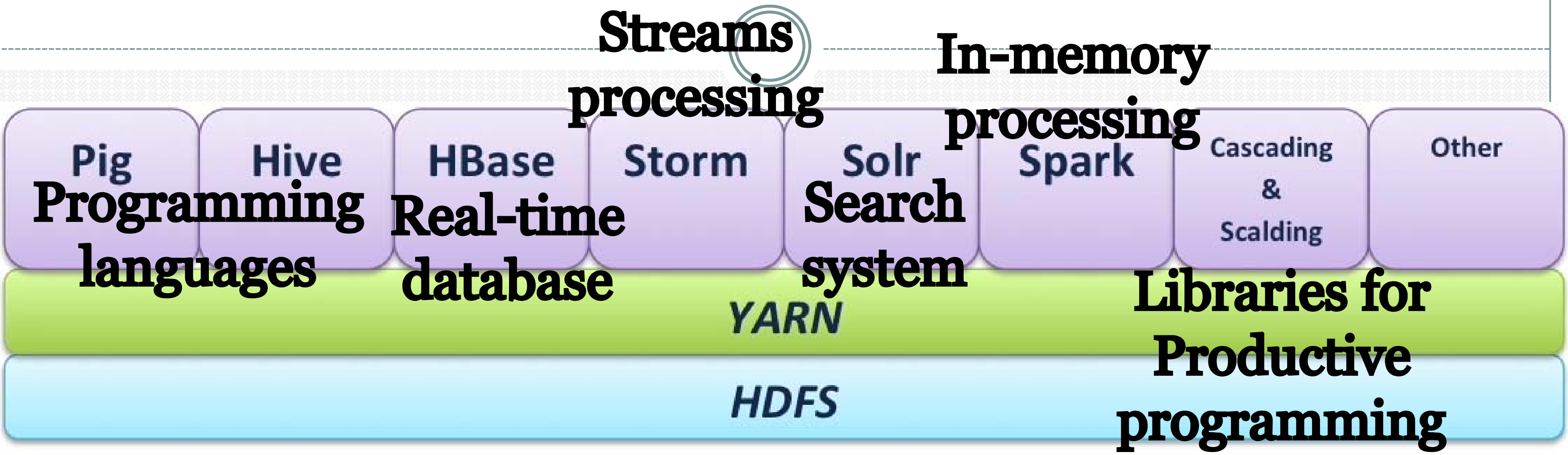
YARN

- Yet Another Resource Negotiator



- Hadoop MapReduce V1 issues:
  - Multi-tenancy
    - MapReduce V1 has a very simple approach to assigning tasks to nodes
    - We wish to manage multiple jobs on a cluster
  - Difficult to scale beyond 4,000 nodes
    - Cascading failures, network flooding
- YARN is an internal reorganization in Hadoop V2
  - API compatible with Hadoop V1

Resource Management with YARN



Streams processing

In-memory processing

Programming languages

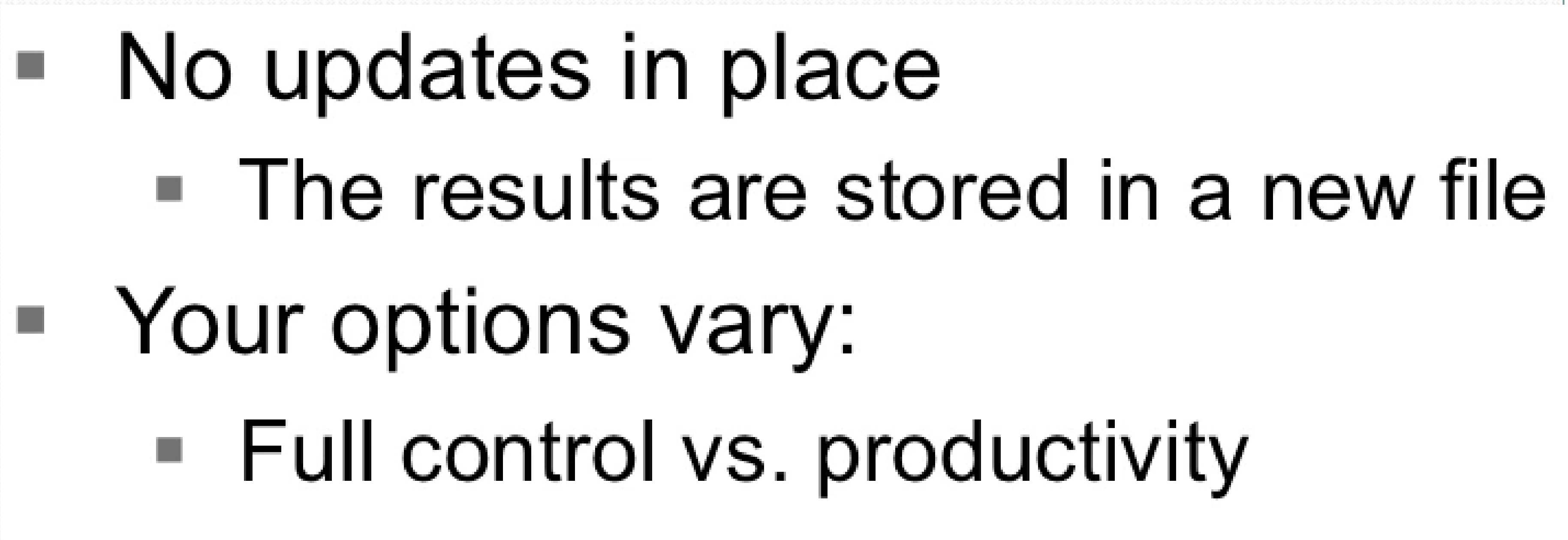
Real-time database

Search system

Libraries for Productive programming

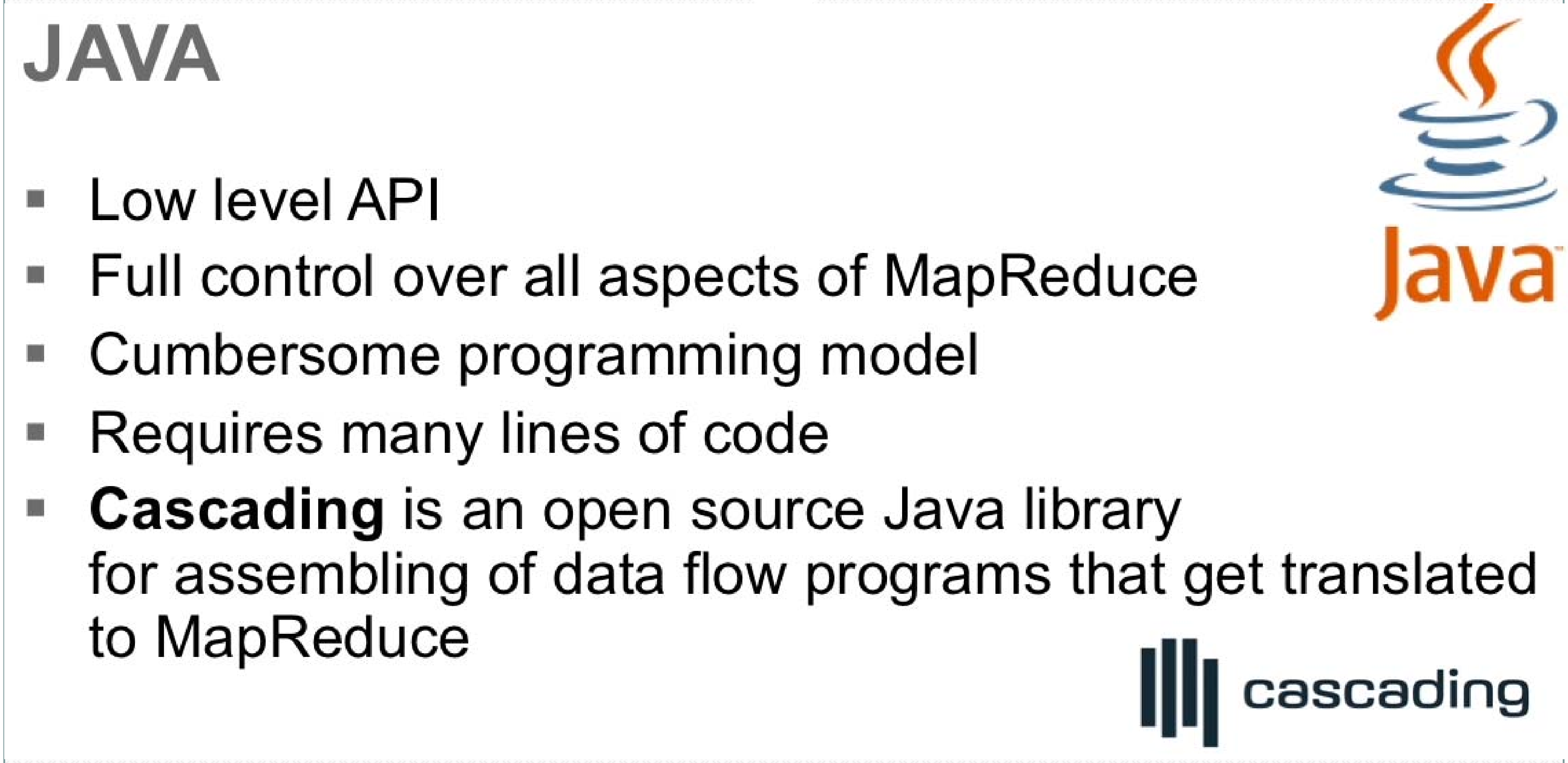
- Yarn: A common resource management for applications
  - Scalability
  - Improved cluster utilization
  - Workloads other than MapReduce

Programming Hadoop



- No updates in place
  - The results are stored in a new file
- Your options vary:
  - Full control vs. productivity

Programming Hadoop with **Java**



JAVA

- Low level API
- Full control over all aspects of MapReduce
- Cumbersome programming model
- Requires many lines of code
- **Cascading** is an open source Java library for assembling of data flow programs that get translated to MapReduce

Java
cascading

Programming with Hadoop Streaming

- Typically used from languages like Python or Ruby
- Simpler model than using Java API
  - Less efficient
- One writes map and reduce functions
  - The system feeds data as input stream, writes as output stream

Programming Hive

- A SQL-like language
- Great starting point for data professionals
- Significantly more productive than Java
- Not as rich as modern SQL
- It can be extended through user defined functions

Programming Pig

- A data flow language
- You can build your programs out of small steps
- Significantly more productive than Java API
- Designed to handle simple flows
  - No control structures or iterations
- It can be extended through user defined functions

Programming Scalding

- A library built on top of Scala
- Elegant model: programs look like manipulating in-memory data structures, get translated into MapReduce
- Very short programs
- Full programming environment
  - Full development ecosystem

Hive

- Data warehousing infrastructure based on the Hadoop
- Massive scale out and fault tolerance
- Originated at Facebook in 2007, now Apache project
- HiveQL: Query language based on SQL
  - Tables are defined on top of HDFS files
  - Queries transformed into MapReduce tasks
- Good candidate when transitioning from Rdb to Hadoop
  - HiveQL will feel familiar

Typical Hive Applications

- ***Store data coming in various formats in HDFS, then apply tools and processes similar to SQL***

- Log processing
- Text processing
- Indexing
- ETL
- Business analytics

Hive—It’s All Files

- Hive table are defined directly over the existing data
- Hive Data Definition is just a mapping
- Hive operates directly over the HDFS files
- Tables can be partitioned into several files for scalability

Hive Workflow

- Typical workflow:
  1. Application dumps data into HDFS
  2. Map data with Hive table definitions
  3. Run queries
  4. Store results into HDFS
  5. Copy/move data from HDFS to regular file system

- Internally, Hive creates the needed MapReduce jobs
- Users see SQL

Hive Example: Table Mapping

```
CREATE TABLE apachelog (
  host STRING,
  identity STRING,
  user STRING,
  time STRING,
  request STRING,
  status STRING,
  size STRING,
  referer STRING,
  agent STRING)
ROW FORMAT SERDE 'org.apache.hadoop.hive.contrib.serde2.RegexSerDe'
WITH SERDEPROPERTIES (
  "input.regex" = "([^]*) ([^]*) ([^]*) (-|\\[^\\]*\\]) ([^ \"]*|\"[^\"]*\") (-|[0-9]*) (-|
[0-9]*)(?: ([^ \"]*|\".*\") ([^ \"]*|\".*\"))?",
  "output.format.string" = "%1$s %2$s %3$s %4$s %5$s %6$s %7$s %8$s %9$s"
)
STORED AS TEXTFILE;
```

Serialization and deserialization properties

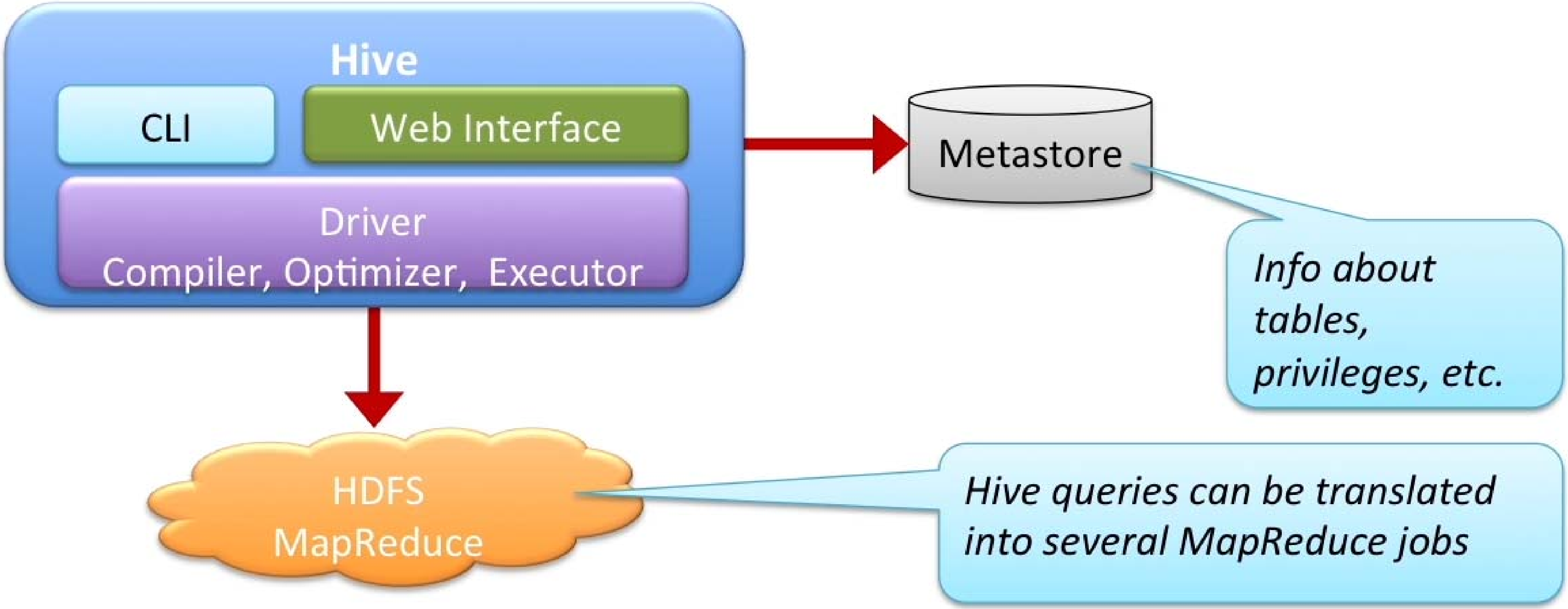
Row format described as Regular Expression

Hive Query

```
SELECT * from apachelog  WHERE host= ’123.456.123.456';
```

- Query language is a subset of SQL
- Hive engine breaks down the query into multiple MapReduce flows, if needed
- Easy to learn and use – much more productive than Java API
- Ideal choice for first touch with Hadoop
- Beware: this is a batch execution – it may take some time to get the results

Hive Architecture



Hive

CLI

Web Interface

Driver

Compiler, Optimizer, Executor

Metastore

HDFS

MapReduce

Info about tables, privileges, etc.

Hive queries can be translated into several MapReduce jobs

Hive vs. Relational Database

Hive	Relational Database
SQL	SQL
Analytics	OLTP or analytics
Batch only	Real-time or batch
No transactions	Transactions
No INSERT or UPDATE Adding through partitions	Random INSERT or UPDATE
Distributed processing - 100s of nodes	Depends on the system - If available, < 100
Achieve high performance on commodity hardware	Achieve high performance on proprietary hardware
Low cost for huge amounts of storage	Expensive, limited compared to Hadoop based solutions

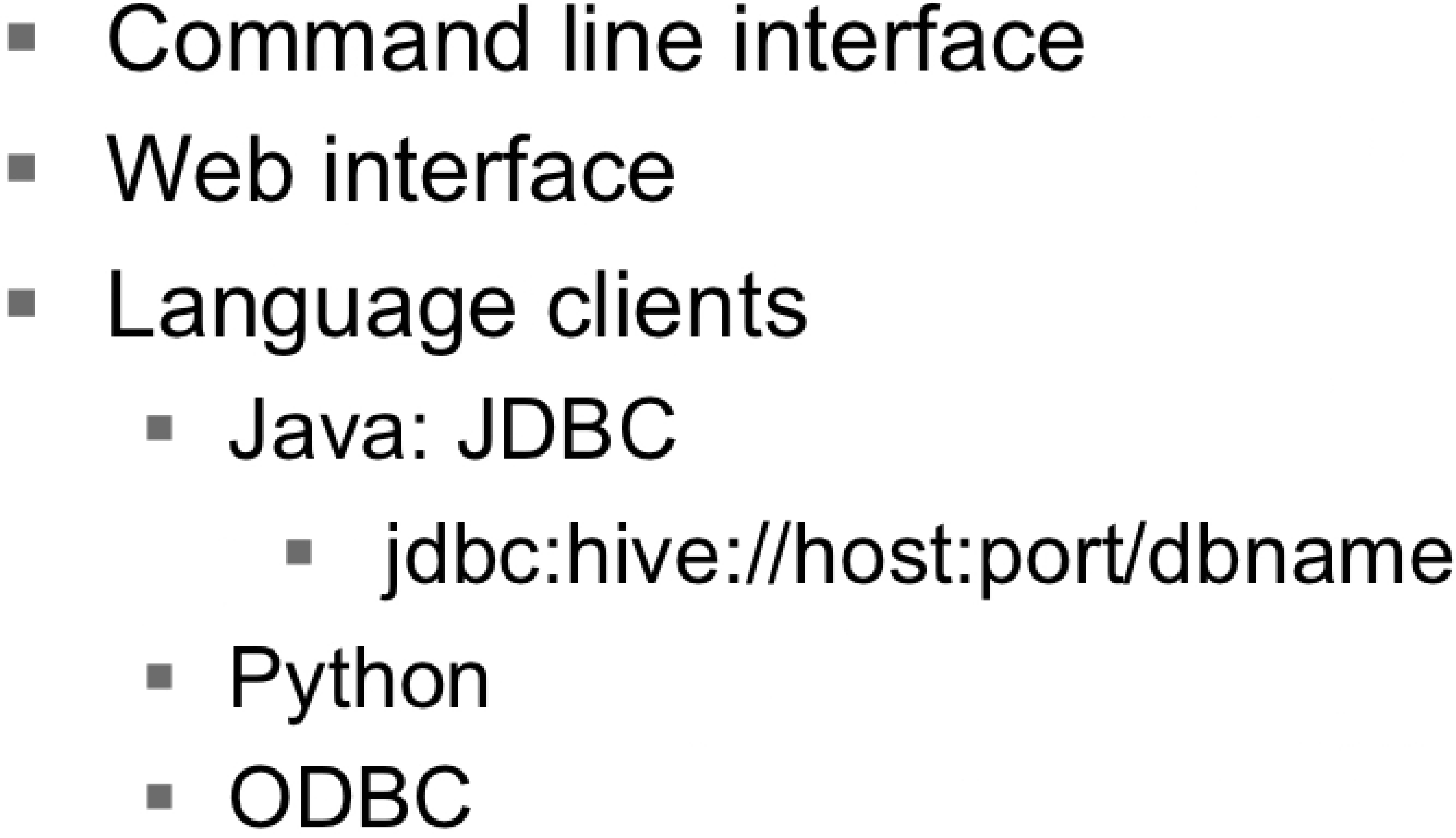
Hive and MapReduce

- A Hive query gets translated into one or more MapReduce jobs
  - The jobs then run on the Hadoop cluster
- Much easier to write HiveQL than Java
- Dramatically increased productivity

Metadata and Metastore

- Metadata internally stored in a relational database
- Internally implemented with DataNucleus ORM
  - Enables easy migration to different RDBs
- It is only metadata, so the store is not large
- The default database is Apache Derby
  - Pure Java, lightweight RDB
  - Good for development
- Production database: typically MySQL

Interacting with Hive



- Command line interface
- Web interface
- Language clients
  - Java: JDBC
    - jdbc:hive://host:port/dbname
  - Python
  - ODBC

Hive Data Model

- Data model follows the familiar relational database layout
- There is no such thing as a Hive format!
  - Hive data model is just a mapping to files
- **Database**
  - Namespace for separation of tables
- **Table**
  - Unit of data with the same schema
  - Tables have columns
  - Columns are mapped to files in HDFS

Hive Table Partitions

- Partition
  - Optional, but useful storage unit for tables
- A table can have one or more partition keys
  - Date
  - Country
- Each value of the partition key defines a partition of the table
  - We can run analytics limited to files US, 2015-01-01
- Partition columns are virtual columns

Hive Buckets (Clusters)

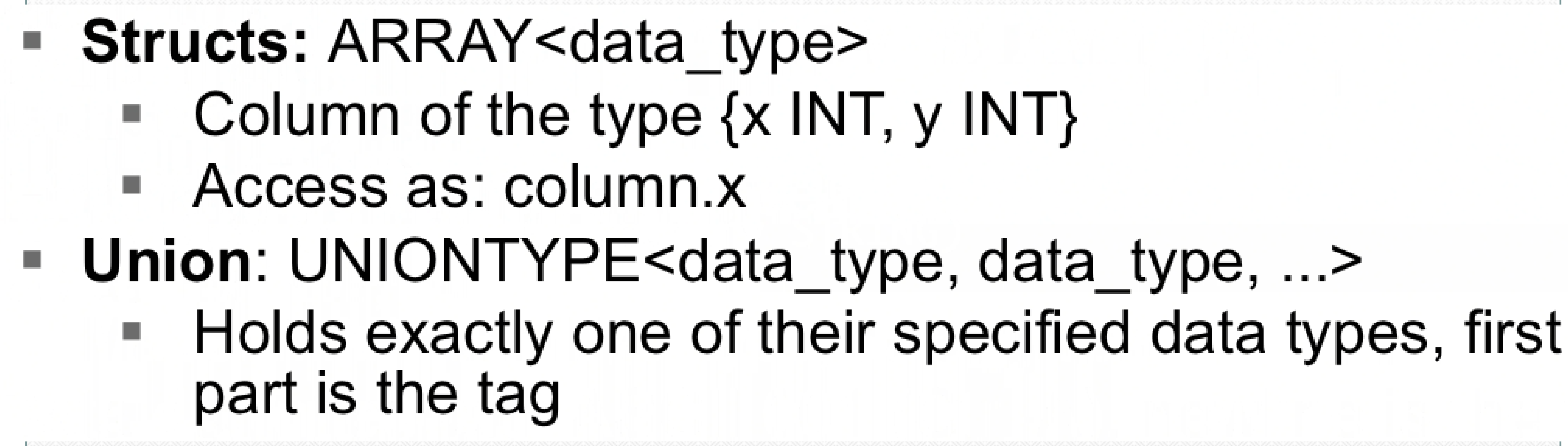
- Data in a partition may be bucketed based on some hash function taking input from a column in a table
- Example:
  - Bucket based on the userid

Partitions and buckets are a convenient way to significantly speed up execution by pruning large quantities of data

Hive Data Types

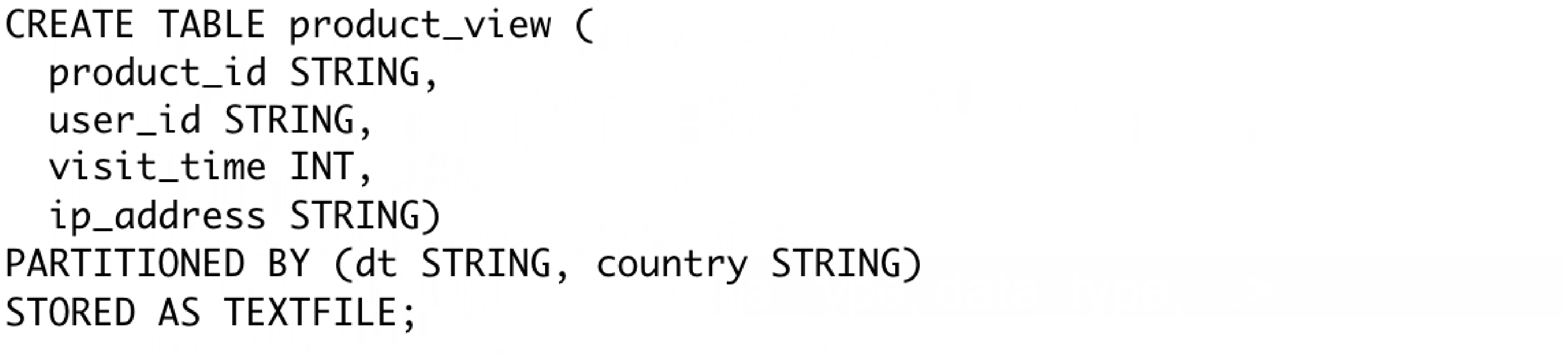
- **Primitive types** -- similar to SQL
  - STRING, INT, BOOLEAN, TIMESTAMP, DATE,…
- **Complex types**
  - **Arrays:** ARRAY<data_type>
    - Column[0]
  - **Maps:** MAP<primitive_type, data_type>
    - column[key]

Hive Data Types (continued)



- **Structs:** ARRAY<data_type>
  - Column of the type {x INT, y INT}
  - Access as: column.x
- **Union**: UNIONTYPE<data_type, data_type, ...>
  - Holds exactly one of their specified data types, first part is the tag

Creating Tables

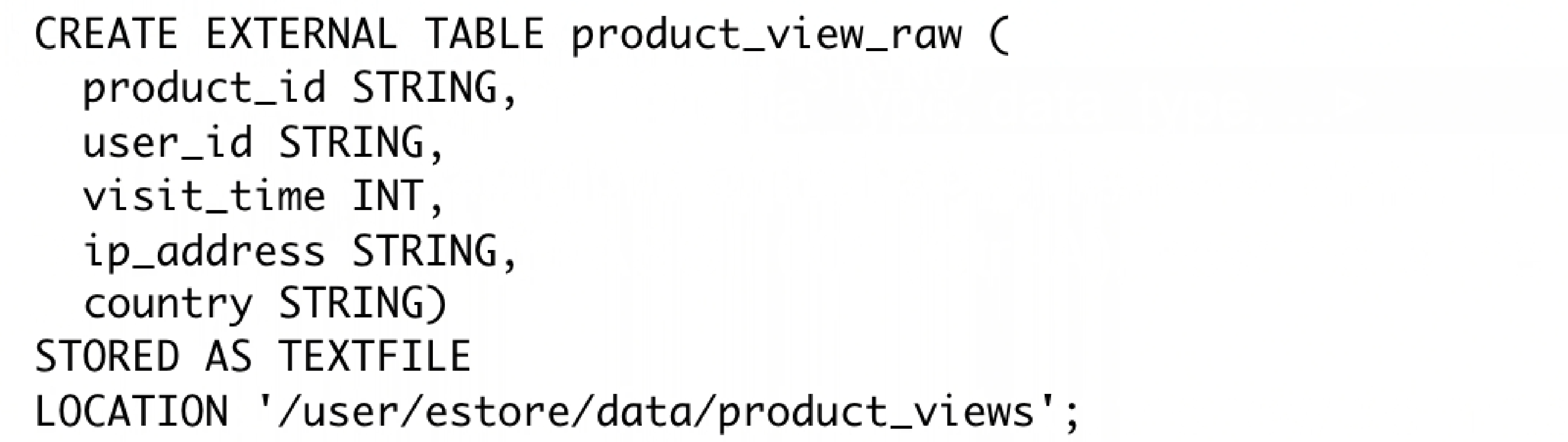


```
CREATE TABLE product_view (
  product_id STRING,
  user_id STRING,
  visit_time INT,
  ip_address STRING)
PARTITIONED BY (dt STRING, country STRING)
STORED AS TEXTFILE;
```

- Data delimited with ASCII 001 (Ctrl-A), newline is the row delimiter
- Data delimiter is configurable, but not the row delimiter

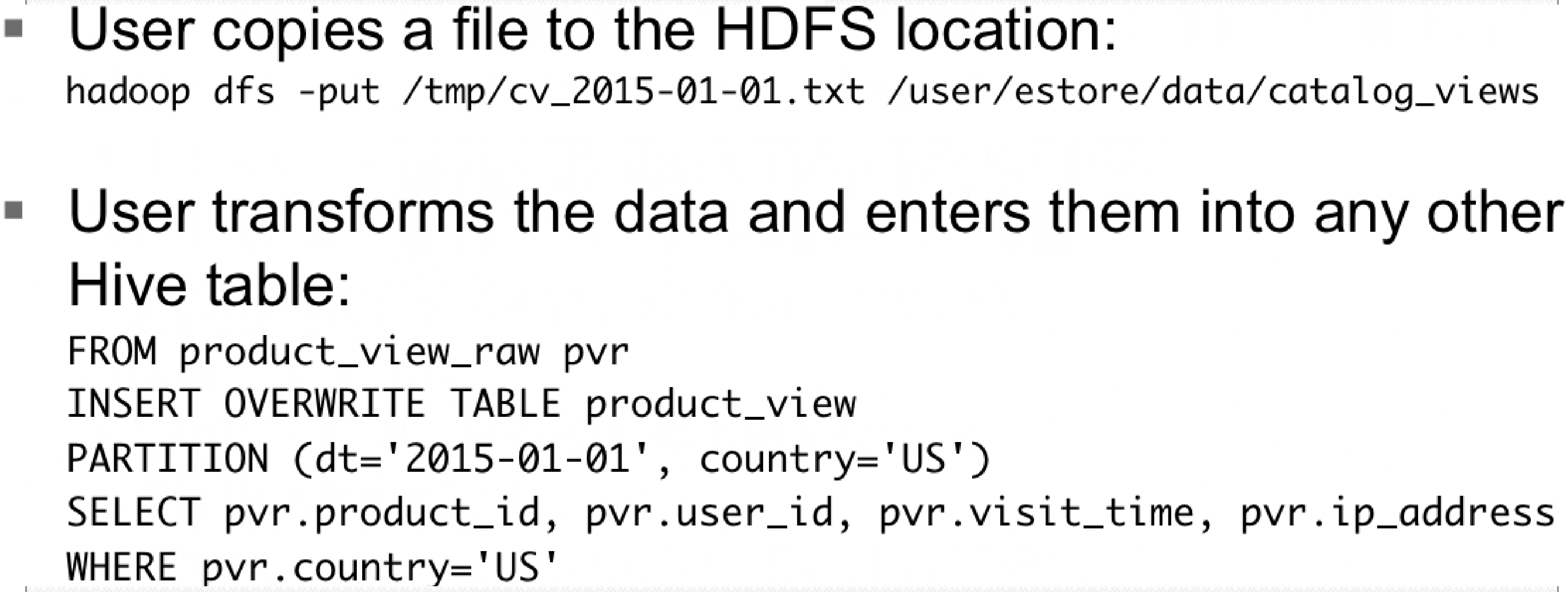
Loading Data

- User creates an external table that points to an HDFS location:



```
CREATE EXTERNAL TABLE product_view_raw (
  product_id STRING,
  user_id STRING,
  visit_time INT,
  ip_address STRING,
  country STRING)
STORED AS TEXTFILE
LOCATION '/user/estore/data/product_views';
```

Loading Data (continued)



- User copies a file to the HDFS location:

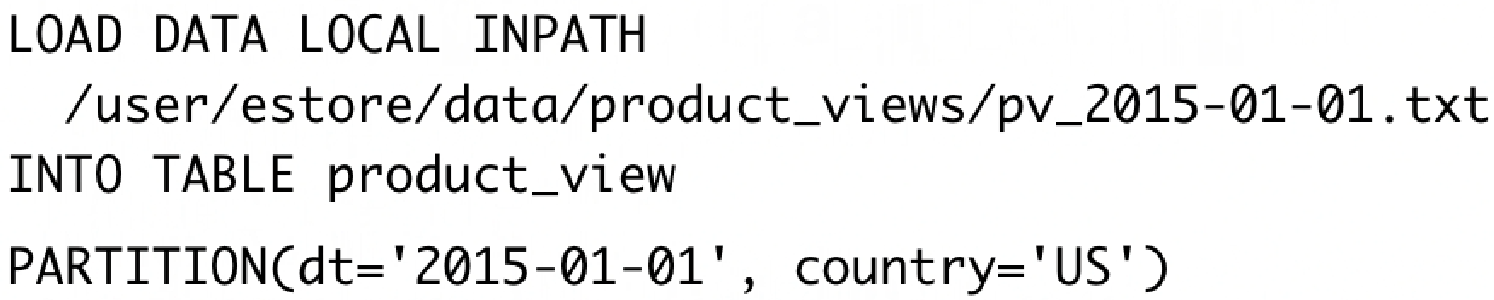
```
hadoop dfs -put /tmp/cv_2015-01-01.txt /user/estore/data/catalog_views
```

- User transforms the data and enters them into any other Hive table:

```
FROM product_view_raw pvr
INSERT OVERWRITE TABLE product_view
PARTITION (dt='2015-01-01', country='US')
SELECT pvr.product_id, pvr.user_id, pvr.visit_time, pvr.ip_address
WHERE pvr.country='US'
```

Loading when format is the same

- When the input file format is the same as the Hive table format, we can use the LOAD statement:



```
LOAD DATA LOCAL INPATH
  /user/estore/data/product_views/pv_2015-01-01.txt
INTO TABLE product_view
PARTITION(dt='2015-01-01', country='US')
```

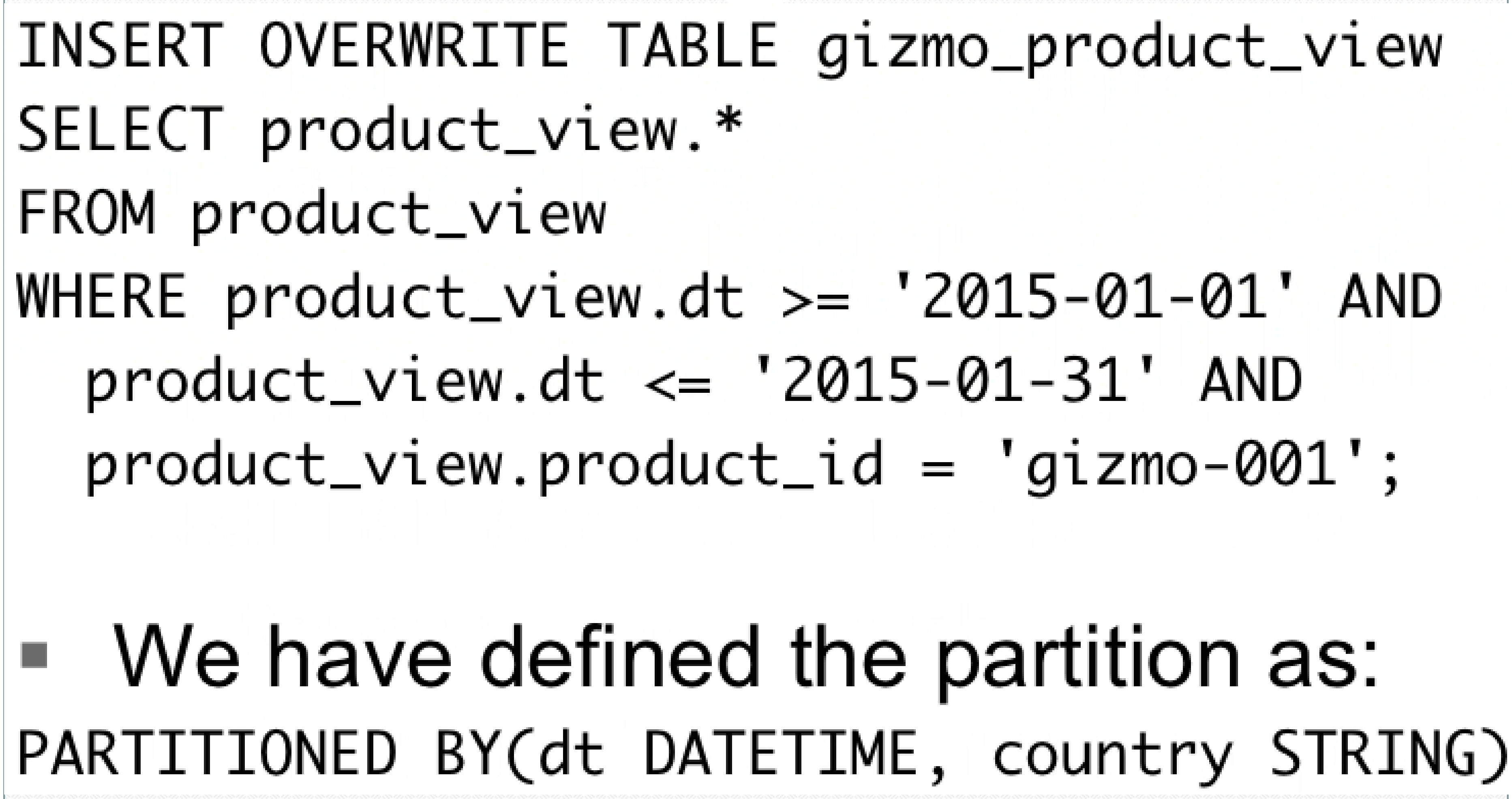
Hive Queries

- All query results are always inserted into a table
  - User can inspect it later, or store them into a local file

```
INSERT OVERWRITE TABLE customers
INSERT OVERWRITE DIRECTORY '/user/data/tmp/’
```

- User can also run queries in Hive CLI
  - Beware: Hive is batch
  - CLI: result is stored into a temporary file, and then presented

Hive Queries and Partitions

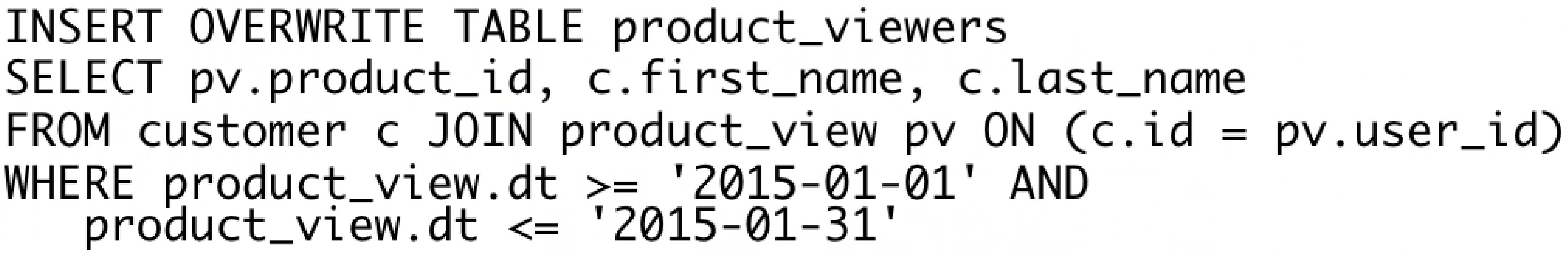


```
INSERT OVERWRITE TABLE gizmo_product_view
SELECT product_view.*
FROM product_view
WHERE product_view.dt >= '2015-01-01' AND
  product_view.dt <= '2015-01-31' AND
  product_view.product_id = 'gizmo-001';
```

- We have defined the partition as:

```
PARTITIONED BY(dt DATETIME, country STRING)
```

Joins in Hive



```
INSERT OVERWRITE TABLE product_viewers
SELECT pv.product_id, c.first_name, c.last_name
FROM customer c JOIN product_view pv ON (c.id = pv.user_id)
WHERE product_view.dt >= '2015-01-01' AND
   product_view.dt <= '2015-01-31'
```

- Join control:
  - LEFT OUTER, RIGHT OUTER or FULL OUTER

- For performance, it is best to put the largest table to the right most position in join

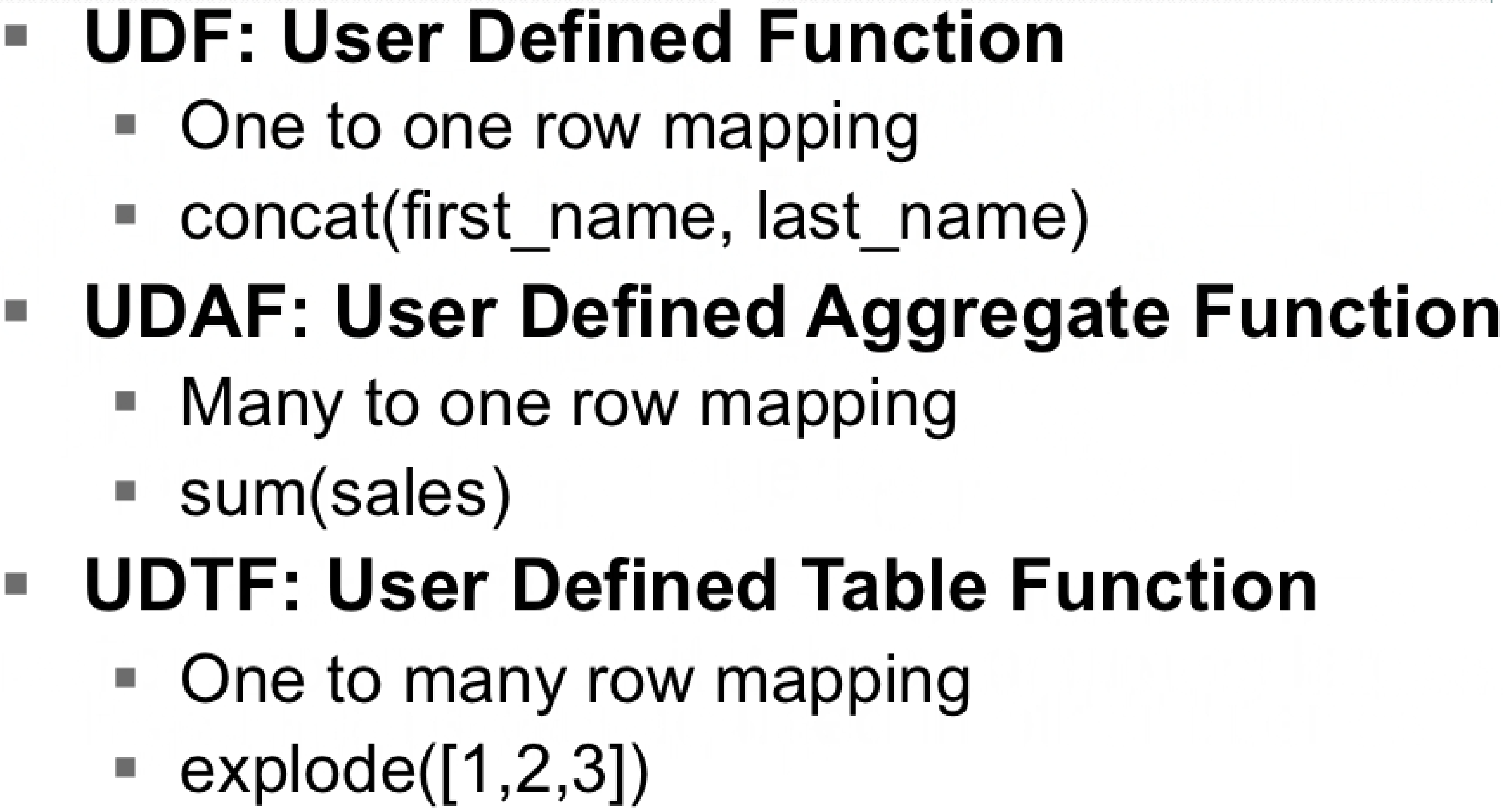
Hive. Storing Results

- Plain SELECT statements give results to the console
- To store result in HDFS use:

```
INSERT OVERWRITE TABLE results
  SELECT  …
```

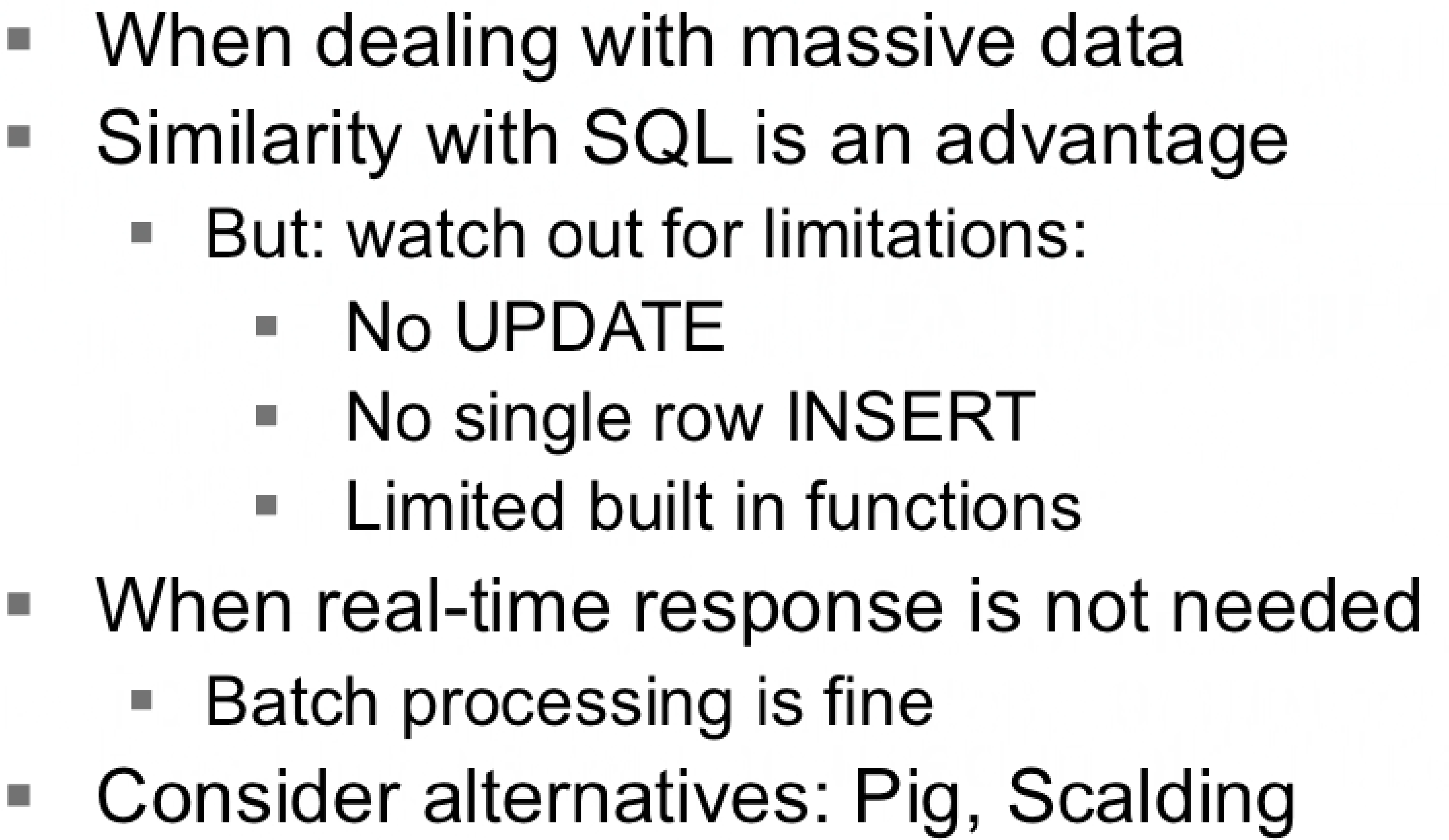
- The results table is a set of files in HDFS
- Result table can be used in other queries

User-Defined Functions



- **UDF: User Defined Function**
  - One to one row mapping
  - concat(first_name, last_name)
- **UDAF: User Defined Aggregate Function**
  - Many to one row mapping
  - sum(sales)
- **UDTF: User Defined Table Function**
  - One to many row mapping
  - explode([1,2,3])

When to use Hive?



- When dealing with massive data
- Similarity with SQL is an advantage
  - But: watch out for limitations:
    - No UPDATE
    - No single row INSERT
    - Limited built in functions
- When real-time response is not needed
  - Batch processing is fine
- Consider alternatives: Pig, Scalding

When **NOT** to use Hive

- Data is in the GB range
  - Consider alternative RDB/NoSQL store
  - Exception: if the conventional solution is to expensive
- Data has no structure (schema)
- You need real-time response

More on Hive

- Why do Pig and Hive exist ?
  - they seem to do much of the same thing.
- Hive pros:
  - its HQL = SQL like query language
  - Often used as the interface to an Apache Hadoop based data warehouse.
  - Hive is friendlier and more familiar than Pig for SQL users
- Pig pros:
  - data flow strengths for bringing data into Apache Hadoop
  - Easy data querying.

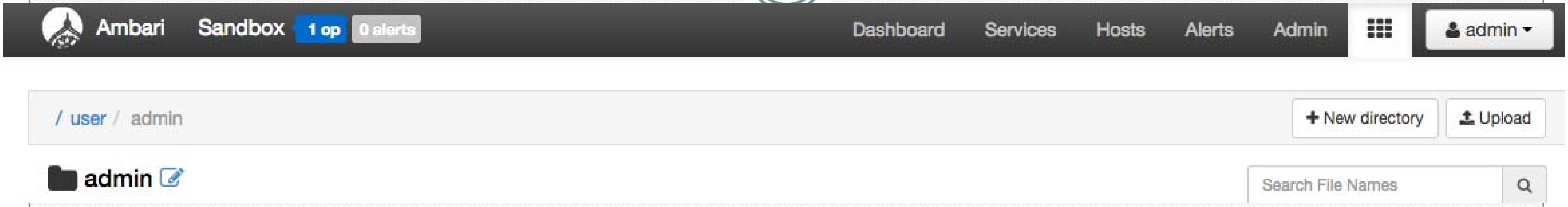
Use Case: Baseball Statistics

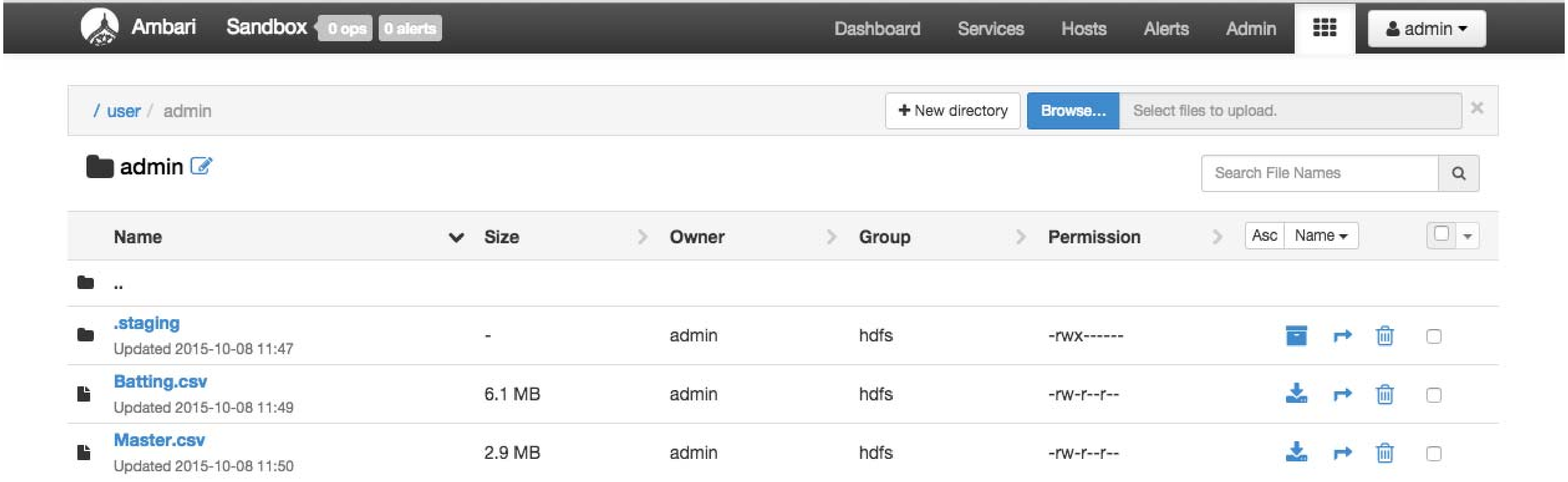
- The data files we are using comes from the site www.seanlahman.com. You can download the data file from:

- http://seanlahman.com/files/database/lahman591-csv.zip

- Once you have the file you will need to unzip it into a directory. We will be uploading just the Master.csv and Batting.csv files from the dataset.

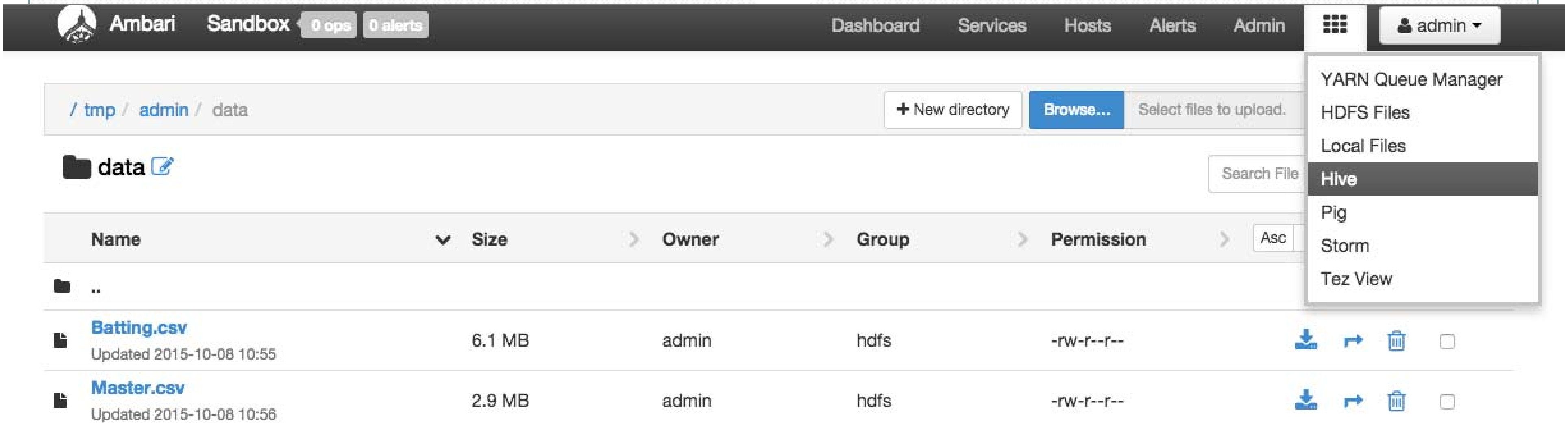
Uploading the data files





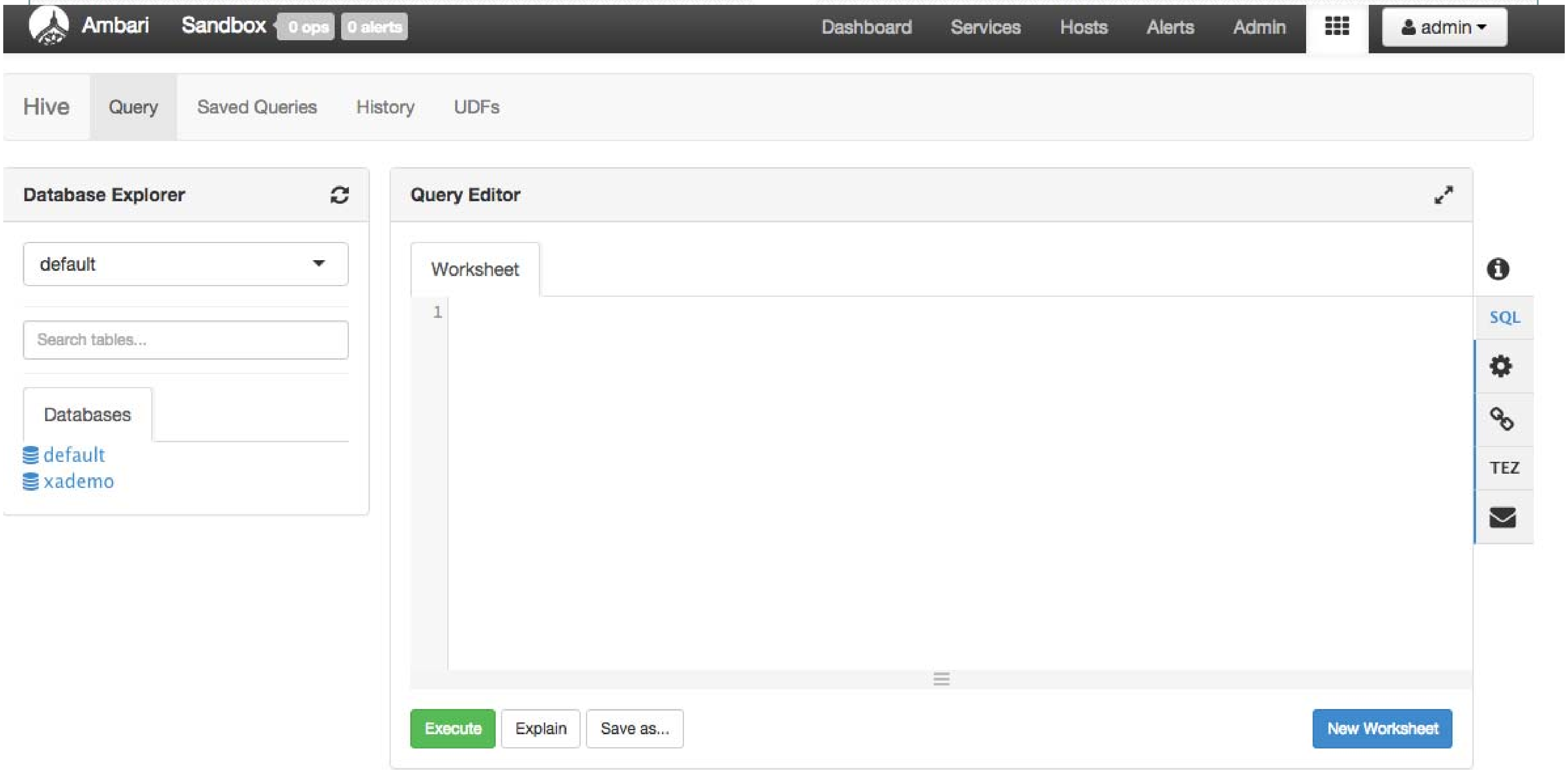
Name	Size	Owner	Group	Permission
..				
.staging Updated 2015-10-08 11:47	-	admin	hdfs	-rwx------
Batting.csv Updated 2015-10-08 11:49	6.1 MB	admin	hdfs	-rw-r--r--
Master.csv Updated 2015-10-08 11:50	2.9 MB	admin	hdfs	-rw-r--r--

Starting the Hive View



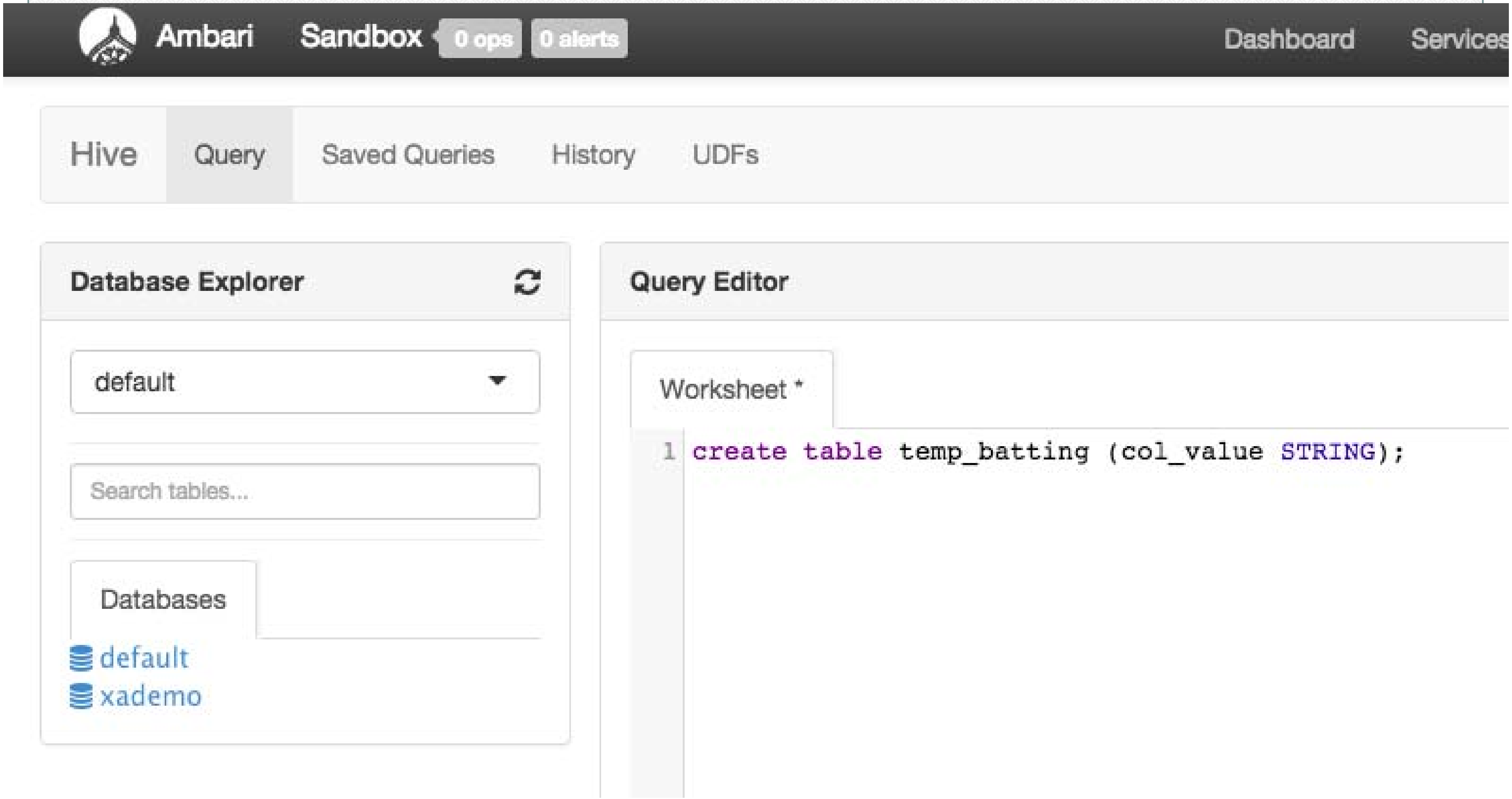
Name	Size	Owner	Group	Permission
..				
Batting.csv Updated 2015-10-08 10:55	6.1 MB	admin	hdfs	-rw-r--r--
Master.csv Updated 2015-10-08 10:56	2.9 MB	admin	hdfs	-rw-r--r--

Hive Query Editor



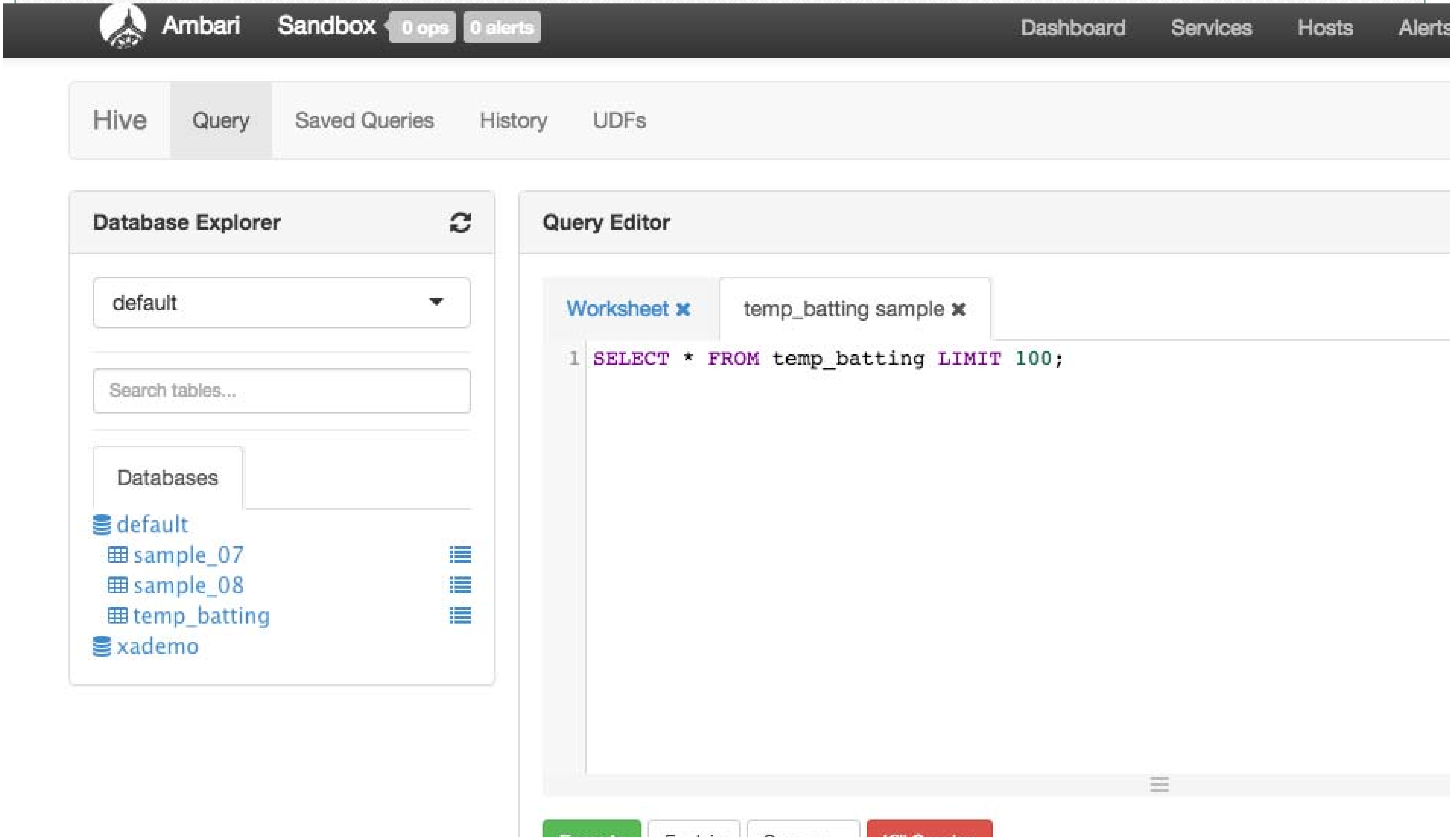
Create a Table

- create table temp_batting (col_value STRING);



```
create table temp_batting (col_value STRING);
```

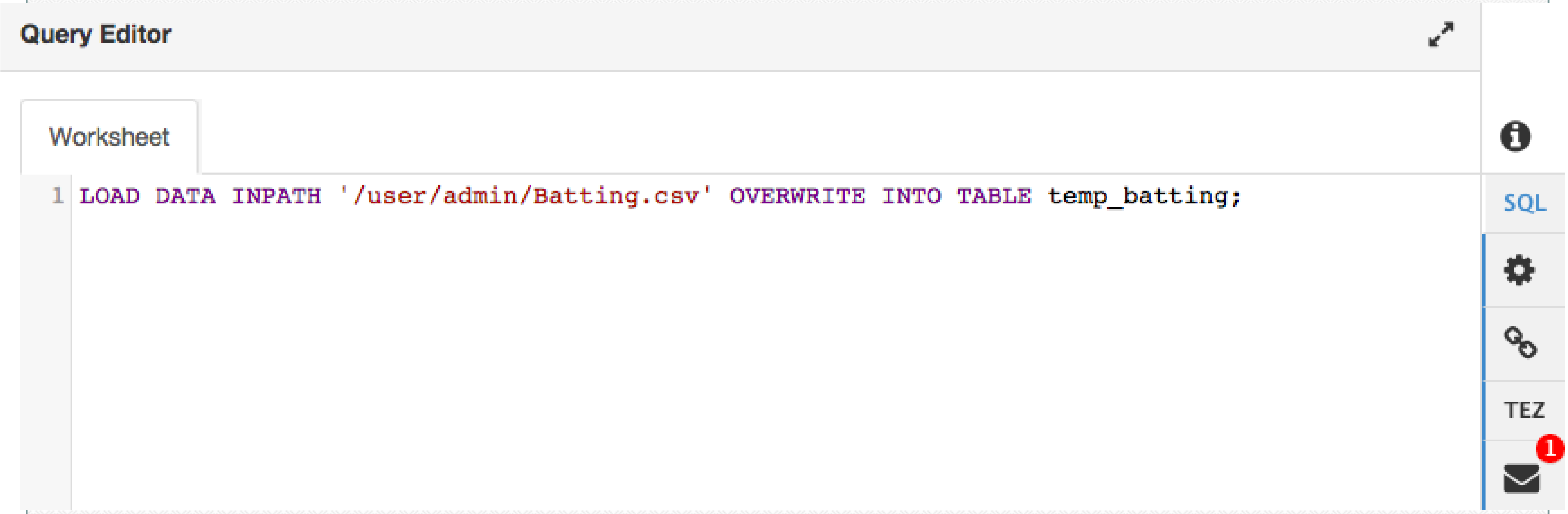
Browse New Data



```
SELECT * FROM temp_batting LIMIT 100;
```

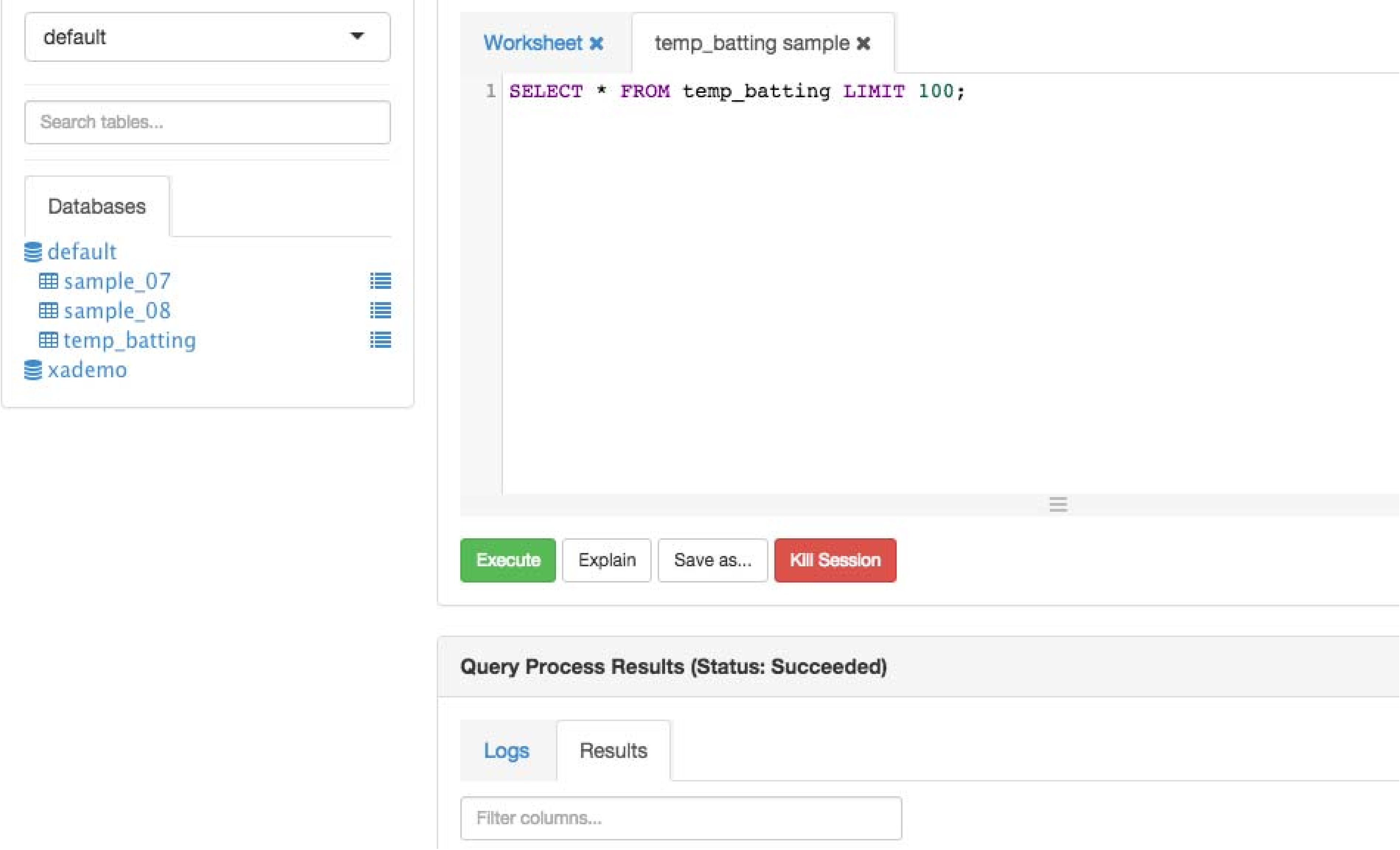
Load More Data into a Temporary Table

- Load the data file Batting.csv into the table temp_batting



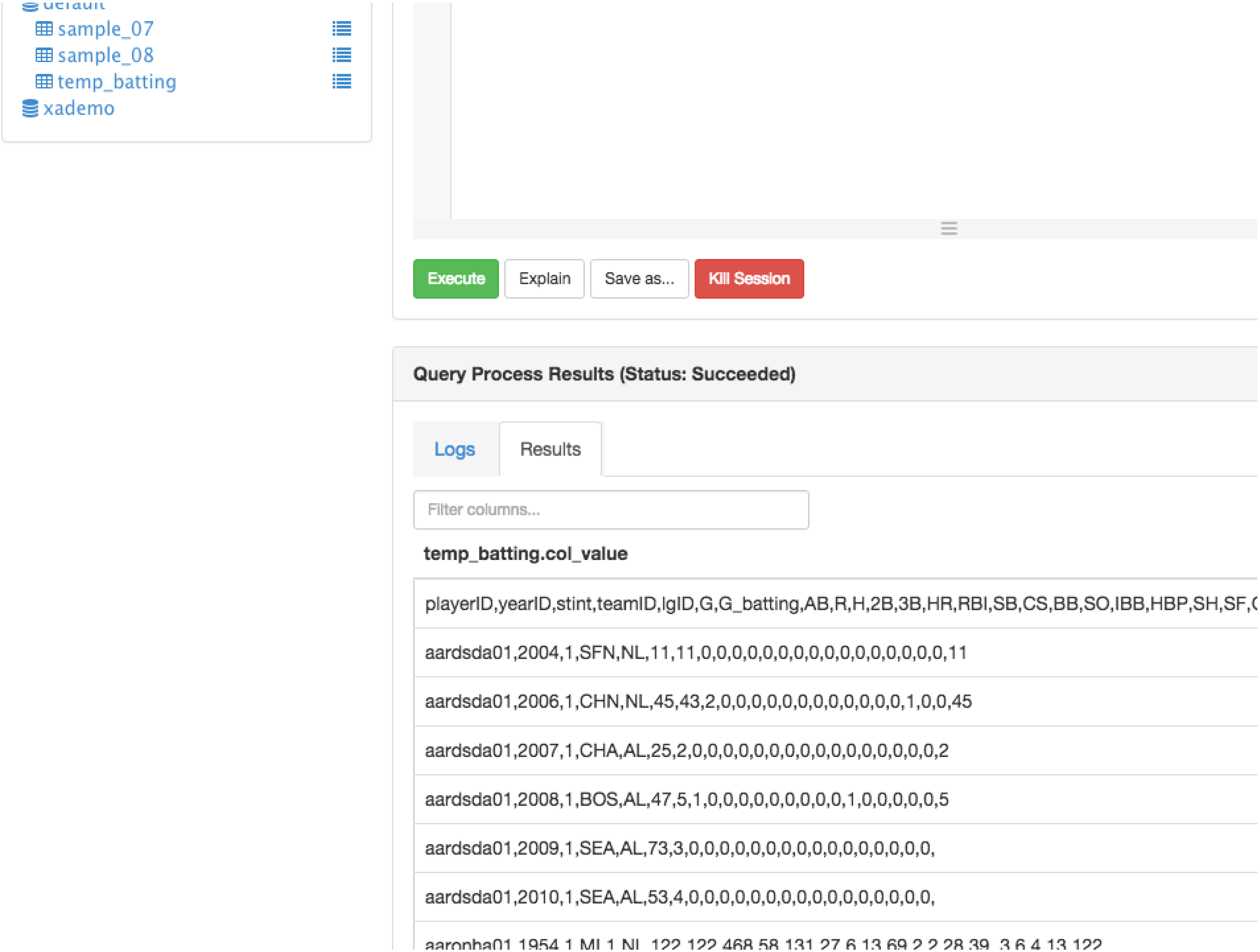
```
LOAD DATA INPATH '/user/admin/Batting.csv' OVERWRITE INTO TABLE temp_batting;
```

Check the Data



```
SELECT * FROM temp_batting LIMIT 100;
```

Query Process Results (Status: Succeeded)

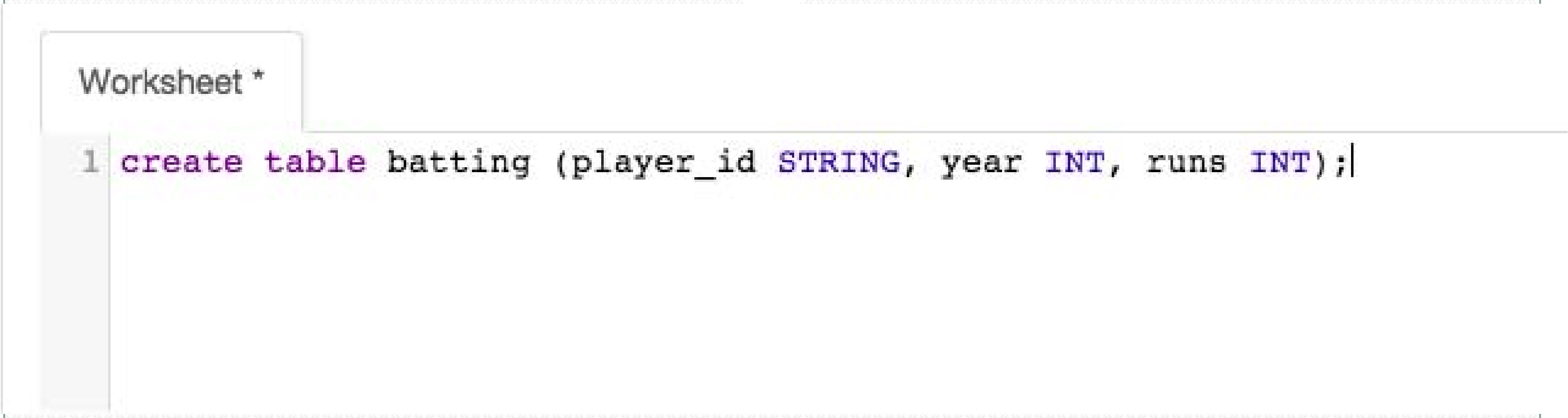


Query Process Results (Status: Succeeded)

temp_batting.col_value

```
playerID,yearID,stint,teamID,lgID,G,G_batting,AB,R,H,2B,3B,HR,RBI,SB,CS,BB,SO,IBB,HBP,SH,SF,
aardsda01,2004,1,SFN,NL,11,11,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,11
aardsda01,2006,1,CHN,NL,45,43,2,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,45
aardsda01,2007,1,CHA,AL,25,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2
aardsda01,2008,1,BOS,AL,47,5,1,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,5
aardsda01,2009,1,SEA,AL,73,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
aardsda01,2010,1,SEA,AL,53,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
aaronha01,1954,1,ML1,NL,122,122,468,58,131,27,6,13,69,2,2,28,39,,3,6,4,13,122
```

Create a Permanent Table

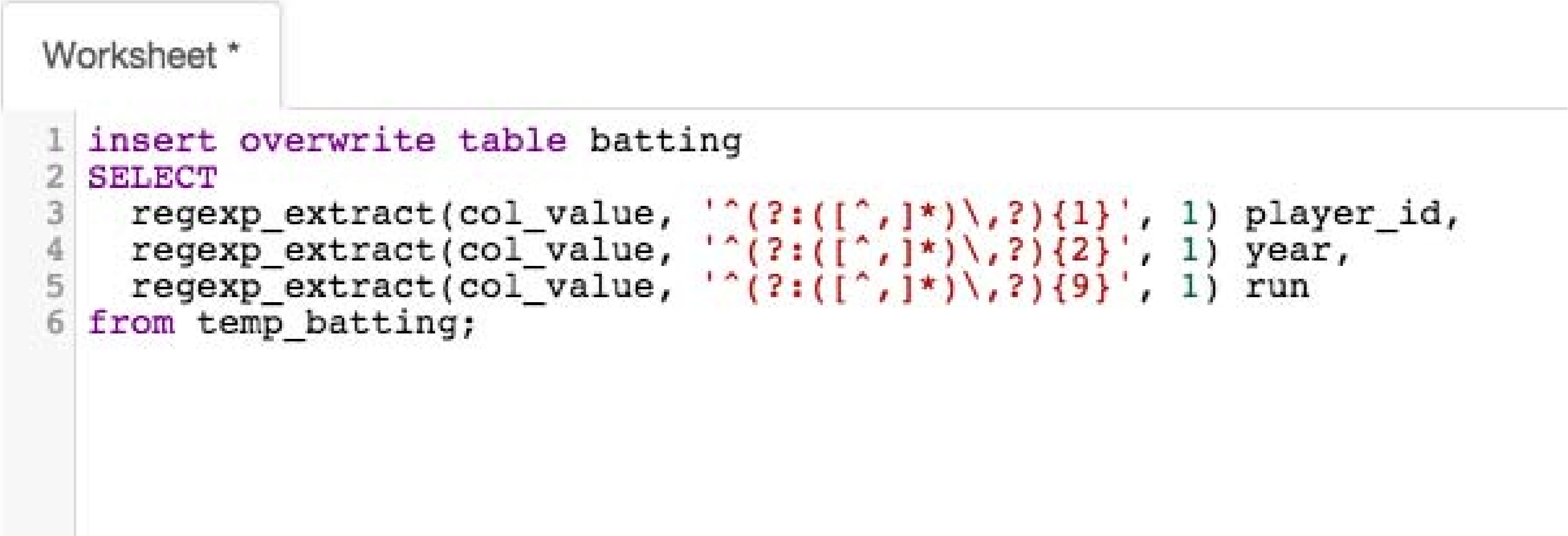


```
create table batting (player_id STRING, year INT, runs INT);
```

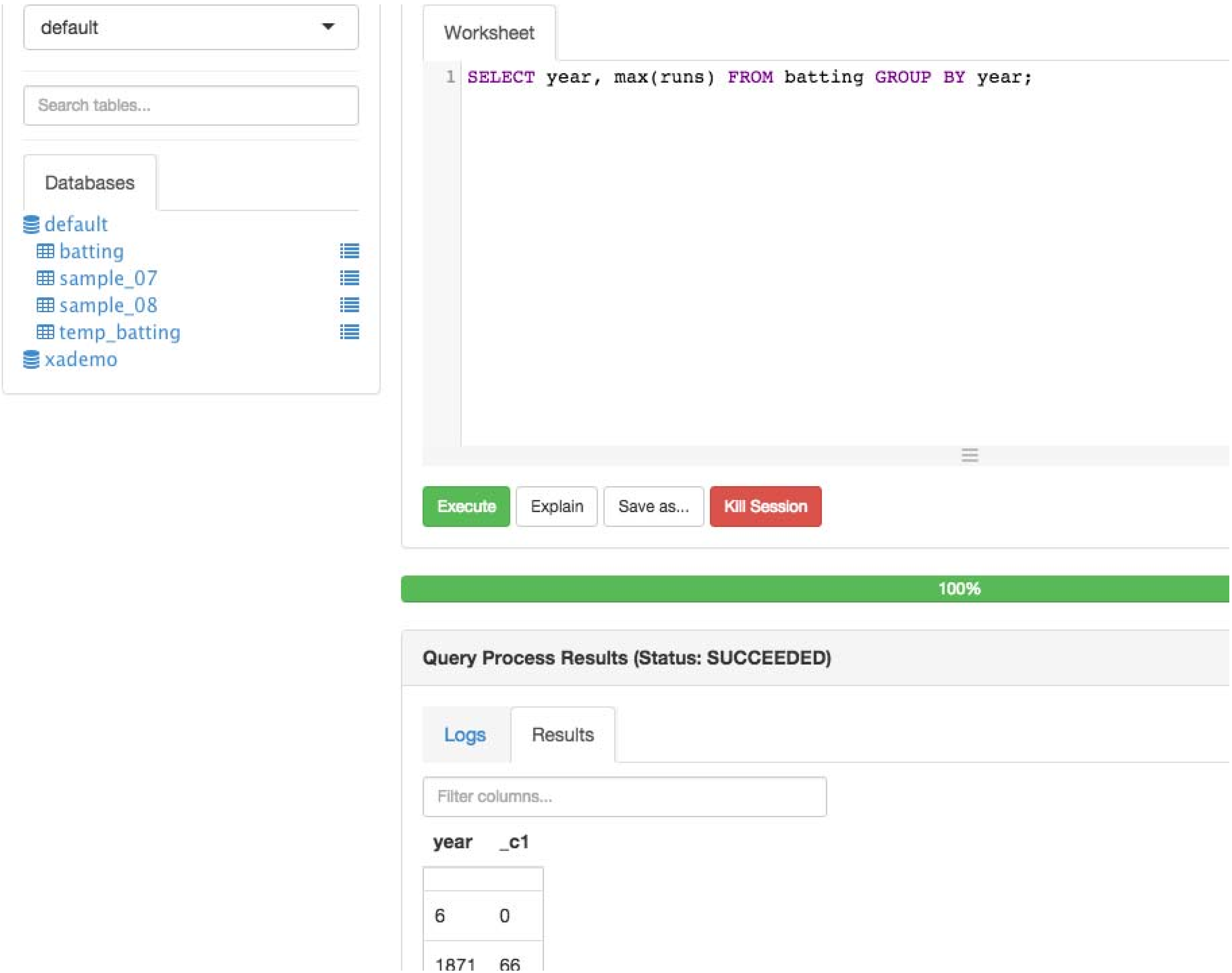
Use RegEx

- Then we extract the data we want from temp_batting and copy it into batting.
- Use a regexp pattern.
- Build up a multi-line query.
- The first line of the query createa the table batting.
- The three regexp_extract calls are going to extract the **player_id**, **year** and **run** fields from the table temp_batting.

Query Using RegEx



```
insert overwrite table batting
SELECT
  regexp_extract(col_value, '^(?:([^,]*)\,?){1}', 1) player_id,
  regexp_extract(col_value, '^(?:([^,]*)\,?){2}', 1) year,
  regexp_extract(col_value, '^(?:([^,]*)\,?){9}', 1) run
from temp_batting;
```



```
SELECT year, max(runs) FROM batting GROUP BY year;
```

Query Process Results (Status: SUCCEEDED)

year	_c1
6	0
1871	66