ST446 Distributed Computing for Big Data

Lecture 4

Structured queries over large datasets

Milan Vojnovic

https://github.com/lse-st446/lectures2021

Goals of this lecture

- Learn about main principles of systems for structured querying of large datasets
- Learn about main principles of system architecture
- Learn about application programming interfaces and query languages

Topics of this lecture

- Hive: introduction, system architecture
- Hive: tables
- Hive: queries
- Spark SQL

Hive: introduction, system architecture

Hive

- Hive: an open source data warehousing solution built on top of Apache Hadoop
  - Brings familiar concepts of tables, columns, partitions and declarative programming (SQL like) to the "unstructured world" of Hadoop
  - Still maintaining the extensibility and flexibility of Hadoop
- HiveQL: a SQL-like declarative language
  - Support for tables with data types including primitive types, and collections like arrays and maps, and nested compositions of those
  - Enables users to plug in custom MapReduce scripts into queries
  - Queries compiled into MapReduce jobs executed using Hadoop

Historical developments

- First design in 2007 (Facebook), open source in 2008
- Replaced a commercial RDMBS at Facebook (to deal with data scale)
- Data scales: 15TB (2007), 700TB (2010), 300PB (2014)
- Design published in an ICDE 2010 paper
- Hadoop considered "unfriendly" for end users, especially for users unfamiliar with MapReduce: lacking the expressiveness of SQL-like query languages

Source: for scale numbers Hive paper, FB engineering post

Hive query language

- HiveQL: a subset of SQL plus extensions
- Anyone familiar with SQL can easily query a dataset using a Hive client
- A mixture of SQL-92, MySQL and Oracle's SQL dialect
- Uses traditional SQL constructs like from clause subqueries, joins (inner, left outer, right outer and outer), cartesian products (a.k.a. cross-joins), group by and aggregations, create table
- Hive's non-standard extensions to SQL clauses inspired by MapReduce:
  - Multitable inserts
  - TRANSFORM
  - MAP
  - REDUCE
- Original Hive design did not support inserting into an existing table
  - All inserts overwriting existing data
  - More recent versions provide updates

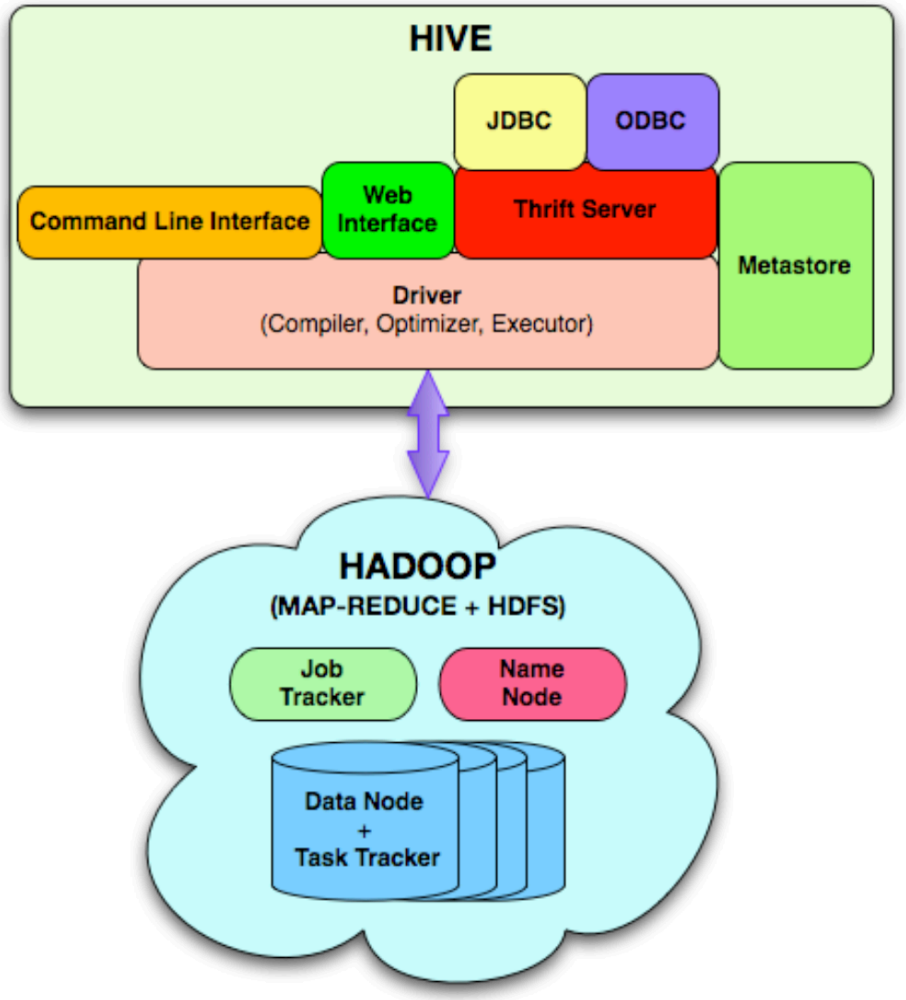
Data model

- Data stored in tables, consisting of rows and columns
- Each column has an associated data type: either primitive or complex
- Primitive data types:
  - Integer: `bigint` (8B), `int` (4B), `smallint` (2B), `tinyint` (1B) (all signed)
  - Float points: `float` (single precision), `double` (double precision)
  - String
- Complex data types:
  - Associative arrays: `map<key-type, value-type>`
  - Lists: `list<element-type>`
  - Structures: `struct<field-name: field-type, ...>`
  - Nested complex types: complex data types can be nested arbitrarily to construct more complex types: ex. `list<map<string, struct<p1:int, p2:int>>>`

Hive vs Python data types

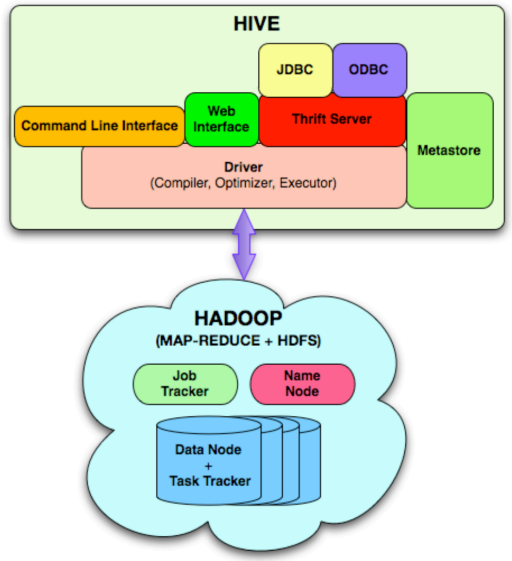
Hive type	Python type
TINYINT	int/long (in range of -128 to 127)
SMALLINT	int/long (in range of -32768 to 32767)
INT	int or long
BIGINT	long
FLOAT	float
DOUBLE	float
DECIMAL	decimal.Decimal
STRING	string
BINARY	bytearray
BOOLEAN	bool
TIMESTAMP	datatime.datetime
ARRAY	list, tuple, or array
MAP	dict
STRUCT	Row

Hive system architecture

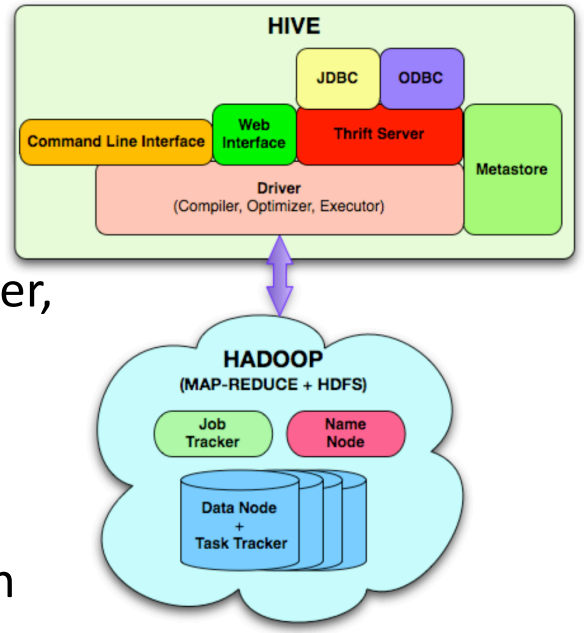


Main system components

- Metastore: stores system catalog and metadata about tables, columns, and partitions
  - Default: Derby - an Apache open source relational database
  - Other, ex. Mysql - another open source relational database
- Driver: manages each HiveQL statement within Hive, session handle and any session statistics
- Query compiler: compiles HiveQL into a directed acyclic graph (DAG) of map/reduce tasks



Main system components (cont'd)



- Execution engine: executes tasks produced by the compiler in a dependency order, interacts with the underlying Hadoop instance
  - MR (deprecated)
  - TEZ: built on Hadoop Yarn for execution of DAGS of data processing tasks
  - Spark
- HiveServer: provides a thrift interface, a JDBC/ODBC server, integrating Hive with applications
- Client: command line interface, web UI, and JDBC/ODBC driver
- Extensibility interfaces: including `SerDe` and `ObjectInspector` interfaces for serialization and deserialization and user defined functions

- Apache Thrift interface (https://thrift.apache.org): an interface definition language and binary communication protocol used for defining and creating services for different programming languages
- JDBC driver: software component allowing a Java application to interact with a database
- ODBC (Open Database Connectivity): an interface that allows applications to access data in a database using SQL

Query execution steps

- client -> driver: a HiveQL query statement submitted to driver via a client interface
  - Client interface is either:
    - Command line interface (CLI)
    - Web UI
    - An external client using a thrift or a JDBC/ODBC interface

- driver -> compiler: driver passes the query to compiler
  - Parse, type check, and semantic analysis phases using metadata in metastore
  - Compiler generates a logical plan using a simple rule-based optimizer, and an optimized plan in the form of a DAG of map-reduce and HDFS tasks

- Execution engine: executes tasks defined by the compiler in the order of their dependencies using Hadoop

HiveServer2 clients

- Hive CLI
- Beeline

- Basic commands:

```
hive> show tables;

hive> describe tables;
```

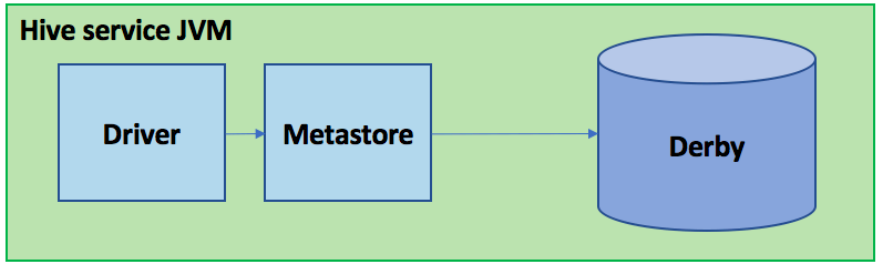
- To probe further: Hive language manual on CLI

Metastore

- Metastore: service and backup store for data

- Three different configurations of a metastore:
  - Embedded
  - Local
  - Remote

- Embedded metastore (default): metastore service runs in the same JVM as the Hive service that contains an embedded Derby database
  - Supports only one Hive session open at any time

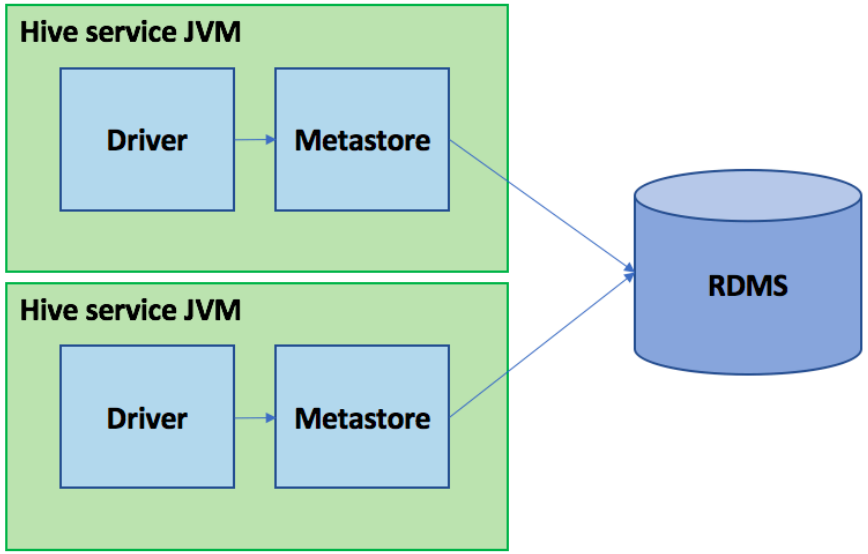


Embedded metastore

JVM: Java virtual machine

Metastore: local

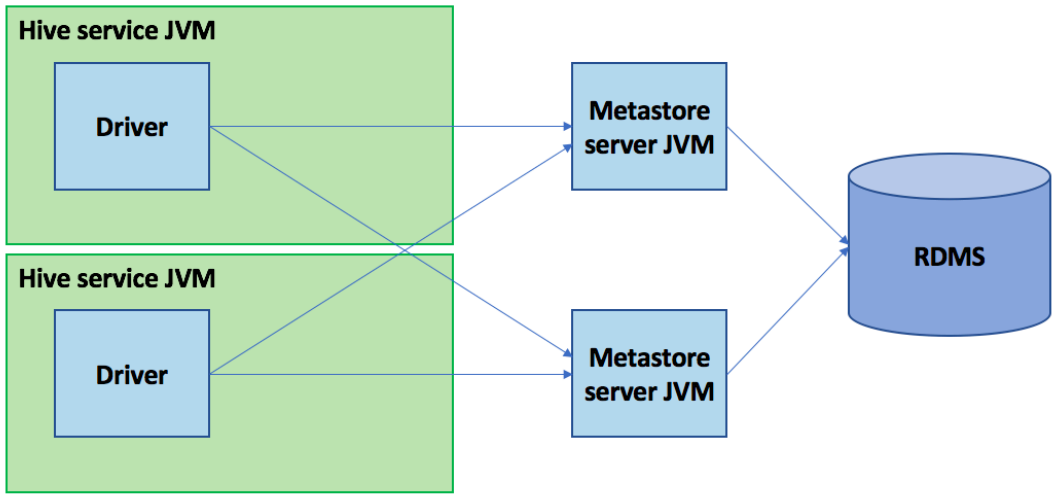
- Local metastore: metastore service using a standalone database
  - Supports multiple concurrent Hive sessions
  - Metastore service runs in the same process as the Hive service, connects to a database running in a separate process
- Any JDBC-compliant database may be used by setting the `java.jdo.option.*` configuration properties
  - Mysql is a popular choice



local metastore

Metastore: remote

- Remote metastore: one or more metastore servers run in separate processes than Hive service processes
- Supports multiple concurrent Hive sessions
- Better manageability and security because the database tier can be completely firewalled and the clients no longer need database credentials

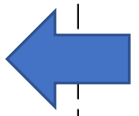


Remote metastore

A look into a Hive metastore

```
mysql> USE metastore;

mysql> show tables;
+---------------------------+
| Tables_in_metastore       |
+---------------------------+
| AUX_TABLE                 |
| BUCKETING_COLS            |
| CDS                       |
| COLUMNS_V2                |
| COMPACTION_QUEUE          |
| COMPLETED_COMPACTIONS     |
| COMPLETED_TXN_COMPONENTS  |
| DATABASE_PARAMS           |
| DBS                       |
| DB_PRIVS                  |
| DELEGATION_TOKENS         |
| DELETEME1514588342142     |
| DELETEME1514590723462     |
| DELETEME1514591104097     |
| DELETEME1514591138450     |
| FUNCS                     |
| FUNC_RU                   |
| GLOBAL_PRIVS              |
| HIVE_LOCKS                |
| IDXS                      |
| INDEX_PARAMS              |
| KEY_CONSTRAINTS           |
| MASTER_KEYS               |
| NEXT_COMPACTION_QUEUE_ID  |
| NEXT_LOCK_ID              |
| NEXT_TXN_ID               |
| NOTIFICATION_LOG          |
| NOTIFICATION_SEQUENCE     |
| NUCLEUS_TABLES            |
| PARTITIONS                |
| PARTITION_EVENTS          |
| PARTITION_KEYS            |
| PARTITION_KEY_VALS        |
| PARTITION_PARAMS          |
| PART_COL_PRIVS            |
| PART_COL_STATS            |
| PART_PRIVS                |
| ROLES                     |
| ROLE_MAP                  |
| SDS                       |
| SD_PARAMS                 |
| SEQUENCE_TABLE            |
| SERDES                    |
| SERDE_PARAMS              |
| SKEWED_COL_NAMES          |
| SKEWED_COL_VALUE_LOC_MAP  |
| SKEWED_STRING_LIST        |
| SKEWED_STRING_LIST_VALUES |
| SKEWED_VALUES             |
| SORT_COLS                 |
| TABLE_PARAMS              |
| TAB_COL_STATS             |
| TBLS                      |
| TBL_COL_PRIVS             |
| TBL_PRIVS                 |
| TXNS                      |
| TXN_COMPONENTS            |
| TYPES                     |
| TYPE_FIELDS               |
| VERSION                   |
| WRITE_SET                 |
+---------------------------+
61 rows in set (0.00 sec)
```



ex. tables

A look into a Hive tables metadata

```
mysql> select * from TBLS;
+--------+-------------+-------+------------------+---------+-----------+-------+------------+---------------+--------------------+--------------------+--------------------+
| TBL_ID | CREATE_TIME | DB_ID | LAST_ACCESS_TIME | OWNER   | RETENTION | SD_ID | TBL_NAME   | TBL_TYPE      | VIEW_EXPANDED_TEXT | VIEW_ORIGINAL_TEXT | IS_REWRITE_ENABLED |
+--------+-------------+-------+------------------+---------+-----------+-------+------------+---------------+--------------------+--------------------+--------------------+
|      1 |  1514460416 |     1 |                0 | vojnovic |         0 |     1 | pokes      | MANAGED_TABLE | NULL               | NULL               |                    |
|      6 |  1514479955 |     1 |                0 | vojnovic |         0 |     6 | invites    | MANAGED_TABLE | NULL               | NULL               |                    |
|     16 |  1514483163 |     1 |                0 | vojnovic |         0 |    16 | u_data     | MANAGED_TABLE | NULL               | NULL               |                    |
|     26 |  1514483567 |     1 |                0 | vojnovic |         0 |    26 | u_data_new | MANAGED_TABLE | NULL               | NULL               |                    |
|     71 |  1514499430 |     1 |                0 | vojnovic |         0 |    71 | word_counts | MANAGED_TABLE | NULL               | NULL               |                    |
|    181 |  1514561105 |     1 |                0 | vojnovic |         0 |   181 | raw_lines  | EXTERNAL_TABLE | NULL               | NULL               |                    |
|    182 |  1514561105 |     1 |                0 | vojnovic |         0 |   182 | word_count | EXTERNAL_TABLE | NULL               | NULL               |                    |
+--------+-------------+-------+------------------+---------+-----------+-------+------------+---------------+--------------------+--------------------+--------------------+
7 rows in set (0.00 sec)
```

Hive: tables

Hive tables

- Hive tables can be created by using `CREATE TABLE clause`
  - Ex:

```
CREATE TABLE pokes (foo INT, bar STRING);
```

- Default input format is a delimiter-separated text

- Tables created in this way are serialized and deserialized using default serializers and deserializes available in Hive

- Flexibility to load data into a table without having to transform the data
  - Accommodates instances where data is prepared by some other programs
  - Can be achieved by providing a jar implementing serialization and deserialization interfaces to Hive

Browsing through tables

- List all tables:

```
SHOW TABLES;
```

- List all tables whose name ends with 's' (patter matching follows Java regular expressions):

```
SHOW TABLES `.*s`
```

- Show the schema of a table:

```
DESCRIBE invites;
```

- Show the schema and detailed information of a table:

```
DESCRIBE EXTENDED invites;
```

Browsing through tables: example

```
hive> show tables;
OK
Time taken: 4.169 seconds

hive> CREATE TABLE pokes (foo INT, bar STRING);
OK
Time taken: 0.739 seconds

hive> describe pokes;
OK
foo                     int
bar                     string
Time taken: 0.115 seconds, Fetched: 2 row(s)
```

Table row format

```
rowFormat
  : ROW FORMAT
    (DELIMITED [FIELDS TERMINATED BY char]
               [COLLECTION ITEMS TERMINATED BY char]
               [MAP KEYS TERMINATED BY char]
               [ESCAPED BY char]
               [LINES SEPARATED BY char]
     |
     SERDE serde_name [WITH SERDEPROPERTIES
                           property_name=property_value,
                           property_name=property_value, ...])
outRowFormat : rowFormat
inRowFormat : rowFormat
outRecordReader : RECORDREADER className
```

SERDE = serialization and deserialization

Hive create table

```
CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name
  [(col_name data_type [COMMENT col_comment], ... [constraint_specification])]
  [COMMENT table_comment]
  [PARTITIONED BY (col_name data_type [COMMENT col_comment], ...)]
  [CLUSTERED BY (col_name, col_name, ...) [SORTED BY (col_name [ASC|DESC], ...)] INTO
num_buckets BUCKETS]
  [SKEWED BY (col_name, col_name, ...)
     ON ((col_value, col_value, ...), (col_value, col_value, ...), ...)
     [STORED AS DIRECTORIES]
  [
   [ROW FORMAT row_format]
   [STORED AS file_format]
     | STORED BY 'storage.handler.class.name' [WITH SERDEPROPERTIES (...)]
  ]
  [LOCATION hdfs_path]
  [TBLPROPERTIES (property_name=property_value, ...)]
  [AS select_statement];
```

To probe further: [Hive Language Manual Create Table]

Managed vs external tables

- Managed table: table data is managed by Hive
  - Hive copies the data into its warehouse directory
  - Created without the EXTERNAL clause
- External table: data resides at a location outside of the warehouse directory

Table partitions

- Hive organizes tables in partitions
- Table divided into coarse-grained parts based on values in a partition columns
- Using partitions makes query evaluation more efficient
- Ex. a log file where each record has a timestamp
  - Queries restricted to a date or set of dates can run much more efficiently because they only need to scan the files in the partitions that the query pertains to

Table partitions (cont'd)

- Partitions are defined at a table creation time using `PARTITION BY` clause

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

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

```
LOAD DATA LOCAL INPATH `<local_file>`
INTO TABLE invites
PARTITION (dt='2017-01-12', country='GB');
```

- File system level: partitions are nested subdirectories of the table directory; the directory structure may look like:

```
/user/hive/warehouse/invites/dt=2017-01-01/country=GB/file1
/user/hive/warehouse/invites/dt=2017-01-01/country=GB/file2
/user/hive/warehouse/invites/dt=2017-01-01/country=FR/file3
/user/hive/warehouse/invites/dt=2017-01-02/country=GB/file4
```

Note: Column definitions in the PARTITION BY clause are full-fledged table columns, called partition columns, but the data files do not contain values for these columns (they are derived from the directory names)

Table buckets

- Buckets: tables or partitions subdivided into buckets
  - May make more efficient query evaluation
    - Ex. sampling of data
    - Ex. join of two tables that are bucketed on the same columns which include the join columns (using map-join - see later)

- Table is bucketed by using `CLUSTERED BY` clause to specify the columns to bucket on and the number of buckets

- Example:

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

  User id is used to determine the bucket which is computed by hashing the value and reducing modulo the number of buckets, so any bucket is effectively assigned a random set of users to it (hash partition)

Table buckets (cont'd)

- Data within a bucket may additionally be sorted by one or more columns
  - Useful for some query evaluations

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

- File system level: each bucket is a file in the table or partition directory
  - n-th bucket is the n-th file arranged in lexicographic order

Altering and dropping tables

```
ALTER TABLE events RENAME TO 3koobecaf;

ALTER TABLE pokes ADD COLUMNS (new_col INT);

ALTER TABLE invites ADD COLUMNS (new_col2 INT COMMENT 'a comment’);

ALTER TABLE invites REPLACE COLUMNS (foo INT, bar STRING, baz INT COMMENT 'baz
replaces new_col2’);
```

`REPLACE COLUMNS` replaces all existing columns and only changes the table's schema, not the data; it can also be used to drop columns from the table's schema:

```
ALTER TABLE invites REPLACE COLUMNS (foo INT COMMENT 'only keep this column’);

DROP TABLE pokes;
```

Loading data into Hive

From a file in the local file system:

```
LOAD DATA LOCAL INPATH './examples/files/kv1.txt' OVERWRITE INTO TABLE pokes;

LOAD DATA LOCAL INPATH './examples/files/kv2.txt' OVERWRITE INTO TABLE invites PARTITION (ds='2008-08-15');
LOAD DATA LOCAL INPATH './examples/files/kv3.txt' OVERWRITE INTO TABLE invites PARTITION (ds='2008-08-08');
```

From a file in HDFS:

```
LOAD DATA INPATH '/examples/files/kv2.txt' OVERWRITE INTO TABLE invites PARTITION (ds='2008-08-15');
```

Generic syntax:

```
LOAD DATA [LOCAL] INPATH 'filepath' [OVERWRITE] INTO TABLE tabname [PARTITION (pcol1=v1, pcol2=v2 ...)]
```

- Schema on read: no verification of data against schema by the load command
  - Unlike to schema on write

Loading data into Hive (cont'd)

From a query (standard syntax):

```
INSERT [OVERWRITE] TABLE tabname1 [PARTITION (pcol1=v1, pcol2=v2 ...) [IF NOT EXISTS]] select stat1 FROM fromst
at;
```

From a query (Hive extension; dynamic partition inserts):

```
INSERT [OVERWRITE] TABLE tabname PARTITION (pcol1[=v1], pcol2[=v2] ...) select stat FROM fromstat;
```

- To probe further: Hive docs

Multitable insert

- Multitable insert: a query with multiple INSERT clauses
- Example: single source table (records), three tables that hold the results from three different queries over the source

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

- More efficient than multiple INSERT statements
  - Source table scanned only once to produce the multiple disjoint outputs

Hive: queries

SELECT

```
[WITH CommonTableExpression (, CommonTableExpression)*]
SELECT [ALL | DISTINCT] select_expr, select_expr, ...
  FROM table_reference
  [WHERE where_condition]
  [GROUP BY col_list]
  [ORDER BY col_list]
  [CLUSTER BY col_list
    | [DISTRIBUTE BY col_list] [SORT BY col_list]
  ]
 [LIMIT [offset,] rows]
```

- `ORDER BY`: result passed on to a single reducer
- `CLUSTER BY`: an alternative to DISTRIBUTE BY and SORT BY
- `DISTRIBUTE BY`: range partition, each partition goes to a different reducer

- Common Table Expression: a temporary result set derived from a simple query specified in a WITH clause, which immediately precedes SELECT or INSERT

SELECT: examples

Output displayed in the console:

```
SELECT a.foo FROM invites a WHERE a.ds='2008-08-15';
```

Output in a HDFS file (number of files depends on the number of mappers used):

```
INSERT OVERWRITE DIRECTORY '/tmp/hdfs_out' SELECT a.* FROM invites a WHERE a.ds='2008-08-15';
```

Example:

```
LSE021353:libexec vojnovic$ hadoop fs -ls /tmp/hdfs_out
Found 1 items
-rwxrwxr-x   3 vojnovic supergroup      11291 2017-12-28 18:20 /tmp/hdfs_out/000000_0
```

Output in a local file:

```
INSERT OVERWRITE LOCAL DIRECTORY '/tmp/local_out' SELECT a.* FROM pokes a;
```

Sampling from a table: TABLESAMPLE

- Sampling ¼ fraction of data:

```
SELECT * FROM bucketed_users
TABLESAMPLE(BUCKET 1 OUT OF 4 ON id);
```

- Sample size does not need to be a multiple of bucket size:

```
SELECT * FROM bucketed_users
TABLESAMPLE(BUCKET 1 OUT OF 8 ON id);
```

- Using buckets more efficient than performing a hash partitioning on entire input dataset:

```
SELECT * FROM bucketed_users
TABLESAMPLE(BUCKET 1 OUT OF 4 ON rand());
```

Joins

- Hive makes performing commonly used operations simple

- Several types of joins

  - Inner join
  - Outer join (including left and right outer joins)
  - Semi joins
  - Map joins

Inner joins

- Inner join: exact match in input tables results in an output row

```
hive> SELECT * FROM sales;
Joe   2
Hank  4
Ali   0
Eve   3
Hank  2

hive> SELECT * FROM products;
2 Tie
4 Coat
3 Hat
1 Scarf

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

Inner joins in MapReduce

- Reduce-side join: identical keys sent to the same reducer
  - Output key of a mapper is the join key
  - Mapper tags each data record with an identifier to differentiate them in the reducer
  - Secondary sorting on data identifiers ensures the order of values sent to the reducer, which generates matched pairs for each join key
  - Reduce-side join belongs to sort-merge join family, scales well for large datasets
  - Less efficient for skewed data: one dataset significantly smaller than other

- Map-side join: used when one dataset is small enough to fit into memory
  - Mappers load the smaller dataset and builds a hash table
  - Process the rows of the larger dataset one-by-one in the map function

To probe further: sort-merge join

Outer joins

- Outer joins allow to find nonmatches in the tables being joined
  - Left outer join: query returns a row for every row in the left table (even if no matching row in the right table exists)
  - Right outer join: same as left outer join but with left and right tables interchanged
  - Full outer join: output has a row for each row from both tables to join

- Left outer join example:

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

Outer joins (cont'd)

- Right outer join example:

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

- Full outer join example:

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

Semi joins

- A query like this

```
SELECT * FROM products WHERE products.id IN (SELECT id FROM sales)
```

can be expressed by using `LEFT SEMI JOIN`

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

ORDER BY

- Similar syntax to Order By in SQL

```
colOrder: ( ASC | DESC )

colNullOrder: (NULLS FIRST | NULLS LAST)

orderBy: ORDER BY colName colOrder? colNullOrder? (',' colName colOrder? colNullOrder?)*

query: SELECT expression (',' expression)* FROM src orderBy
```

SORT BY

- `SORT BY`: sorts the data per reducer
  - Columns in SORT BY used to sort the rows before feeding them to reducers
  - The sort order is dependent on the column types
    - If the column is of numeric type, then the sort order is also in numeric order
    - If the column is of string type, then the sort order will be lexicographical order

- `ORDER BY` vs `SORT BY`:
  - ORDER BY guarantees a total order in the output
  - SORT BY only guarantees ordering of the rows within a reducer
  - If there is more than one reducer, SORT BY may give a partially ordered result

clusterBy and distributeBy

- `clusterBy` and `distributeBy` mainly used by Transform/Map-Reduce Scripts

- Sometimes useful in `SELECT` statements if there is a need to partition and sort the output of a query for subsequent queries

- `clusterBy`: a short-cut for `distributeBy` and `SORT BY`

- Hive uses the columns in `distributeBy` to distribute rows across reducers
  - All rows with the same `distributeBy` column values will go to the same reducer
  - `distributeBy` does not guarantee clustering or sorting properties on the distributed keys

Hive specific queries: TRANSFORM

- General syntax:

```
FROM (
    FROM src
    SELECT TRANSFORM '(' expression (',' expression)* ')'
    (inRowFormat)?
    USING 'my_map_script'
    ( AS colName (',' colName)* )?
    (outRowFormat)? (outRecordReader)?
    ( clusterBy? | distributeBy? sortBy? ) src_alias
  )
  SELECT TRANSFORM '(' expression (',' expression)* ')'
    (inRowFormat)?
    USING 'my_reduce_script'
    ( AS colName (',' colName)* )?
    (outRowFormat)? (outRecordReader)?
```

Example 1

```
FROM (
  FROM pv_users
  SELECT TRANSFORM(pv_users.userid, pv_users.date)
  USING 'map_script'
  AS dt, uid
  CLUSTER BY dt) map_output
INSERT OVERWRITE TABLE pv_users_reduced
  SELECT TRANSFORM(map_output.dt, map_output.uid)
  USING 'reduce_script'
  AS date, count;
```

Example 2

```
FROM (
  FROM src
  SELECT TRANSFORM(src.key, src.value) ROW FORMAT SERDE 'org.apache.hadoop.hive.contrib.serde2.TypedBytesSerDe'
  USING '/bin/cat'
  AS (tkey, tvalue) ROW FORMAT SERDE 'org.apache.hadoop.hive.contrib.serde2.TypedBytesSerDe'
  RECORDREADER 'org.apache.hadoop.hive.contrib.util.typedbytes.TypedBytesRecordReader'
) tmap
INSERT OVERWRITE TABLE dest1 SELECT tkey, tvalue
```

Hive specific queries: MAP and REDUCE

- General syntax:

```
FROM (
  FROM src
  MAP expression (',' expression)*
  (inRowFormat)?
  USING 'my_map_script'
  ( AS colName (',' colName)* )?
  (outRowFormat)? (outRecordReader)?
  ( clusterBy? | distributeBy? sortBy? ) src_alias
)
REDUCE expression (',' expression)*
  (inRowFormat)?
  USING 'my_reduce_script'
  ( AS colName (',' colName)* )?
  (outRowFormat)? (outRecordReader)?
```

Example 1

```
FROM (
  FROM pv_users
  MAP pv_users.userid, pv_users.date
  USING 'map_script'
  AS dt, uid
  CLUSTER BY dt) map_output
INSERT OVERWRITE TABLE pv_users_reduced
  REDUCE map_output.dt, map_output.uid
  USING 'reduce_script'
  AS date, count;
```

Example 2

```
FROM (
  FROM pv_users
  MAP pv_users.userid, pv_users.date
  USING 'map_script'
  CLUSTER BY key) map_output
INSERT OVERWRITE TABLE pv_users_reduced
  REDUCE map_output.key, map_output.value
  USING 'reduce_script'
  AS date, count;
```

- Schema-less mapreduce query: if there is no `AS` clause after `USING 'map_script'`, Hive assumes that the output of the script contains two parts:
  - Key which is before the first tab
  - Value which is the rest after the first tab

  This is different from specifying AS key, value because in that case, value will only contain the portion between the first tab and the second tab if there are multiple tabs
- We can use `CLUSTER BY` key without specifying the output schema of the scripts
- More on transform / mapreduce queries here

Word count example

```
FROM (
        FROM raw_lines
        MAP raw_lines.line
        USING 'word_count_mapper.py'
        AS word, count
        CLUSTER BY word) map_output
INSERT OVERWRITE TABLE word_count
        REDUCE map_output.word, map_output.count
        USING 'word_count_reducer.py'
        AS word, count;
```

- `MAP` clause indicates how the input columns are transformed using a user program into output columns
- `CLUSTER BY` clause in the sub-query specifies the output columns that are hashed to distribute the data to the reducers
- `REDUCE` clause specifies the user program to invoke on the output columns of the sub-query

Spark SQL

Spark SQL

- Spark SQL: a Spark module for structured query data processing
  - Provides an interface for working with structured and semi-structured data

- Two ways to interface with Spark SQL:
  - SQL
  - DataFrame API

- Spark SQL can be used to execute SQL queries
- Spark SQL can be used to interact with a Hive installation
- Spark SQL run within a programming language returns results in a DataFrame

DataFrames

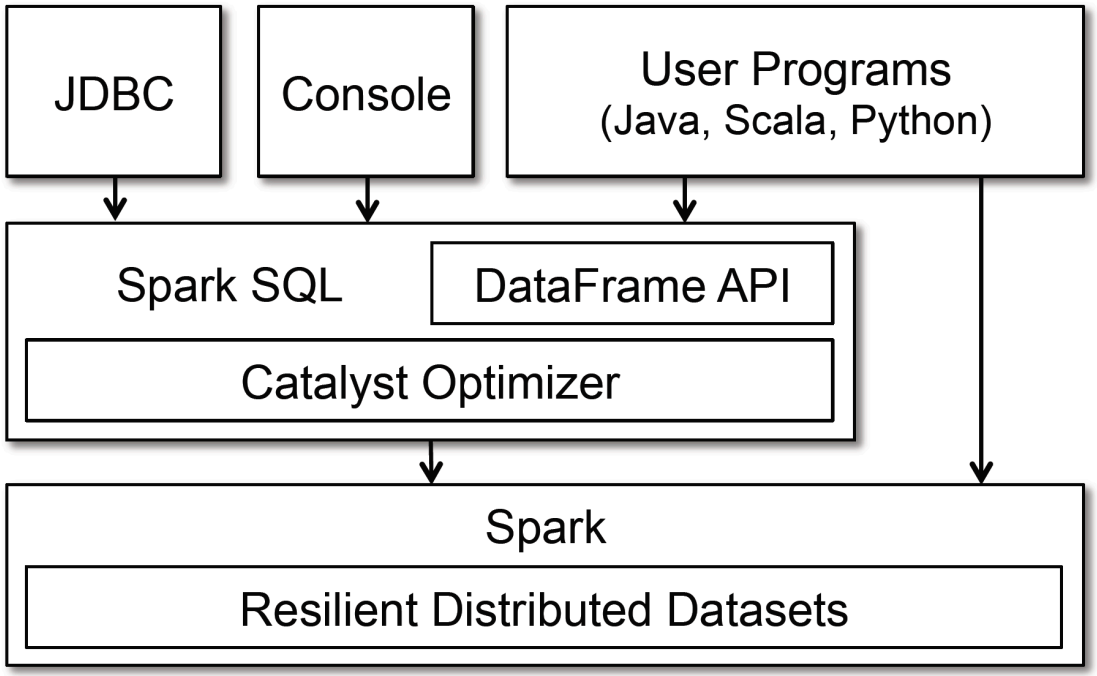
- Spark DataFrame: a dataset organized into named columns
  - DataFrames are Datasets of Rows in Scala and Java API
  - Like a relational database table or a Pandas dataframe in Python
  - But with richer optimizations under the hood

- DataFrame API is available in Scala, Java, Python and R

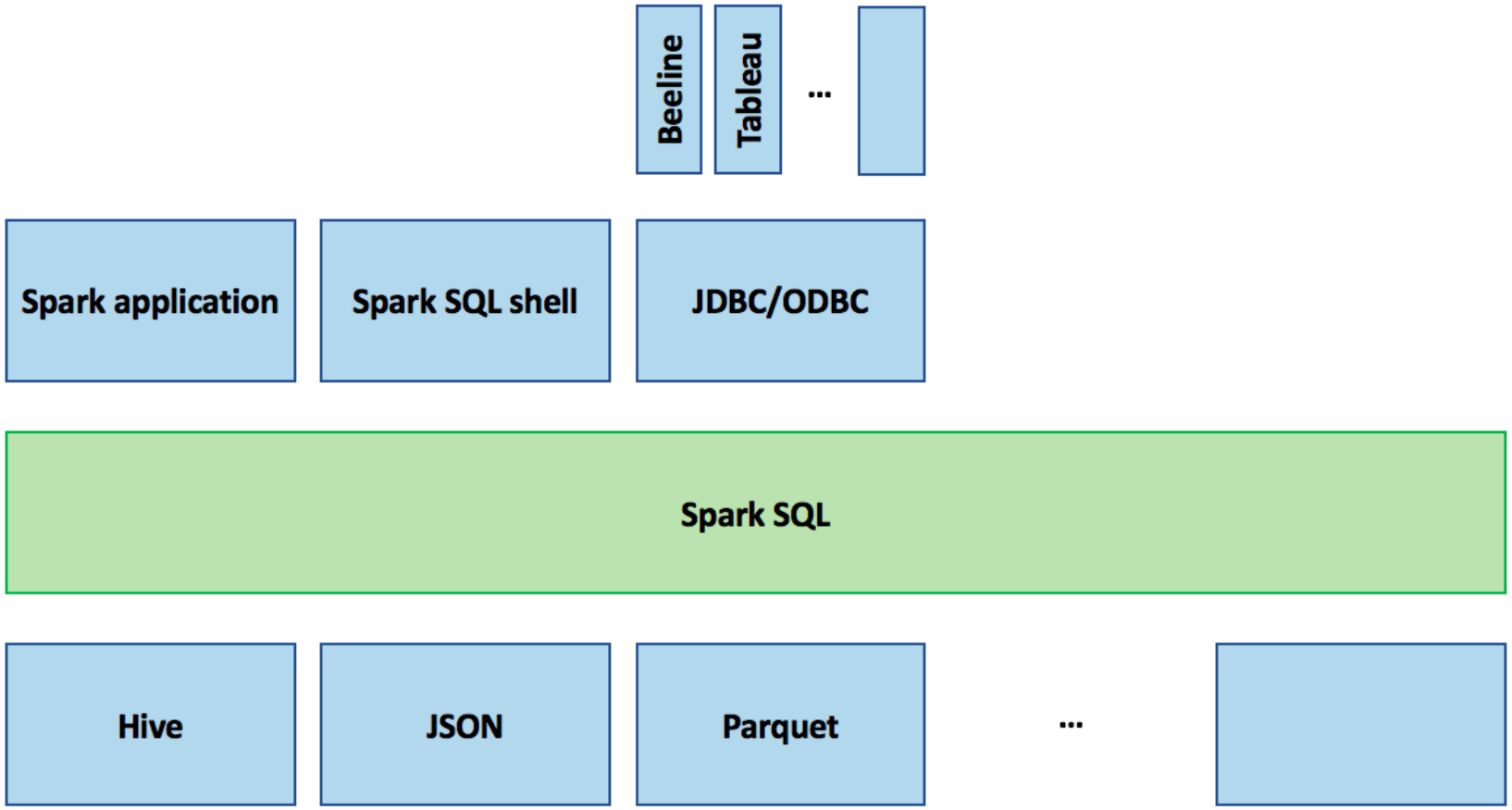
- DataFrame stores data in a more efficient manner than using a native RDD
  - By taking advantage of schema

- Lazy evaluation: a DataFrame object represents a logical plan how to compute a dataset, but no execution occurs until user calls a special output operation such as save
  - Like evaluation of RDD operations

Spark SQL system architecture



Spark SQL inputs and outputs



SparkSession

- `SparkSession`: the entry point to Spark SQL functionalities
- Provides built-in support for Hive features
  - Ex. ability to write queries using HiveQL, access to Hive user-defined functions (UDFs), and ability to read data from Hive tables
- To use these features, it is not required to have an existing Hive installation

- Example:

```
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark SQL basic example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

Creating a DataFrame

- Applications can create a DataFrame within a `SparkSession`
  - From an existing RDD
  - From a Hive table
  - From a Spark data source

- Example: JSON file -> DataFrame

```
df = spark.read.json("examples/src/main/resources/people.json")
```

DataFrame API

- DataFrame API: a domain-specific language for structured data manipulation in Scala, Java, Python and R

Function	Purpose	Example
`show`	Show the content of the DataFrame	`df.show()`
`select`	Select the specified fields/functions	`df.select("name", df("age")+1)`
`filter`	Select only the rows meeting the criteria	`df.filter(df("age") > 19)`
`groupBy`	Group together on a column, needs to be followed by an aggregation function	`df.groupBy(df("name")).min()`

- Examples of aggregation functions: count, min, max, sum, agg

SQL API

- `sql` function of a `SparkSession` object allows allows applications to run SQL queries programmatically and return the result as a DataFrame

- Example:

```
df_input.createOrReplaceTempView("people")

df_output = spark.sql("SELECT * FROM people")
```

- Temporary view in Spark SQL is session scoped
  - Will disappear if the session that created it terminates

Global temporary view

- Global temporary view: a temporary view can be shared across different sessions and keep alive until the Spark application terminates

```
# Global temporary view is tied to a system preserved database `global_temp`
df_input.createGlobalTempView("people")

spark.sql("SELECT * FROM global_temp.people").show()
# +----+-------+
# | age|   name|
# +----+-------+
# |null|Michael|
# |  30|   Andy|
# |  19| Justin|
# +----+-------+

# Global temporary view is cross-session
spark.newSession().sql("SELECT * FROM global_temp.people").show()
# +----+-------+
# | age|   name|
# +----+-------+
# |null|Michael|
# |  30|   Andy|
# |  19| Justin|
# +----+-------+
```

Interoperation of DataFrames and RDDs

- There are two ways of converting an existing RDD into a DataFrame
  - Reflection: schema inferred from an RDD that contains specific object types
  - Programmatic interface: user specifies the schema and applies it to an RDD

Reflection

- Spark SQL can convert an RDD of Row objects to a DataFrame by inferring the data types
- Rows are constructed by passing a list of key-value pairs to the Row class
  - The keys define the column names of the table
- Data types are inferred by sampling the dataset

Reflection (cont'd)

Example: input file $SPARK_HOME/examples/src/main/resources/people.txt

```
Michael, 29
Andy, 30
Justin, 19
```

PySpark code:

```
from pyspark.sql import Row

sc = spark.sparkContext

# Load a text file and convert each line to a Row.
lines = sc.textFile("examples/src/main/resources/people.txt")
parts = lines.map(lambda l: l.split(","))
people = parts.map(lambda p: Row(name=p[0], age=int(p[1])))
```

Reflection (cont'd)

```
# Infer the schema, and register the DataFrame as a table
schemaPeople = spark.createDataFrame(people)
schemaPeople.createOrReplaceTempView("people")

# SQL can be run over DataFrames that have been registered as a table
teenagers = spark.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")

# The results of SQL queries are Dataframe objects
# rdd returns the content as an :class:`pyspark.RDD` of :class:`Row`
teenNames = teenagers.rdd.map(lambda p: "Name: " + p.name).collect()
for name in teenNames:
    print(name)
# Name: Justin
```

Programmatic interface

- Three steps:
  - Create an RDD of tuples or lists from the original RDD
  - Create the schema represented by a `StructType` matching the structure of tuples or lists in the RDD created in step 1
  - Apply the schema to the RDD via `createDataFrame` method provided by `SparkSession`

```
# Import data types
from pyspark.sql.types import *

sc = spark.sparkContext

# Load a text file and convert each line to a Row.
lines = sc.textFile("examples/src/main/resources/people.txt")
parts = lines.map(lambda l: l.split(","))
# Each line is converted to a tuple.
people = parts.map(lambda p: (p[0], p[1].strip()))
```

Programmatic interface (cont'd)

```
# The schema is encoded in a string.
schemaString = "name age"

fields = [StructField(field_name, StringType(), True) for field_name in schemaString.sp
lit()]
schema = StructType(fields)

# Apply the schema to the RDD
schemaPeople = spark.createDataFrame(people, schema)

# Creates a temporary view using the DataFrame
schemaPeople.createOrReplaceTempView("people")

# SQL can be run over DataFrames that have been registered as a table
results = spark.sql("SELECT name FROM people")

results.show()
# +-------+
# |   name|
# +-------+
# |Michael|
# |   Andy|
# | Justin|
# +-------+
```

Data sources

- Spark SQL supports operating on a variety of data sources through the DataFrame interface and SQL

- Here are some examples:

  - Parquet files
    (default, unless otherwise specified by `spark.sql.sources.default`)
  - JSON
  - Hive tables
  - JDBC

Load and save functions

Example:

```
df = spark.read.load("examples/src/main/resources/users.parquet")

df.select("name", "favorite_color").write.save("namesAndFavColors.parquet")
```

Another example:

```
df = spark.read.load("examples/src/main/resources/people.json", format="json")

df.select("name", "age").write.save("namesAndAges.parquet", format="parquet")
```

- SQL queries can be run on files directly:

Example:

```
df=spark.sql("SELECT * FROM parquet.`examples/src/main/resources/users.parquet`")
```

Saving to persistent tables

- `DataFrame` can be saved as a persistent table in a Hive metastore using `saveAsTable` command
- In absence of a Hive metastore, Spark will create a default local Hive metastore (Derby)
- `saveAsTable` materializes the content of the DataFrame and creates a pointer to the data in the Hive metastore
- Persistent tables remain to exist even after a Spark application has restarted, if connection to the same metastore is used
- A DataFrame for a persistent table can be created by calling the table method on a `SparkSession` with the name of the table

Saving to persistent tables (cont'd)

```
df = spark.read.json("examples/src/main/resources/people.json")
df.write.saveAsTable("people")

spark.sql("show tables").show()
# +--------+-----------+-----------+
# |database|  tableName|isTemporary|
# +--------+-----------+-----------+
# | default|    invites|      false|
# | default|     people|      false|
# | default|      pokes|      false|
# | default|  raw_lines|      false|
# | default|     u_data|      false|
# | default| u_data_new|      false|
# | default| word_count|      false|
# | default|word_counts|      false|
# +--------+-----------+-----------+

spark.sql("describe people").show()
# +--------+---------+-------+
# |col_name|data_type|comment|
# +--------+---------+-------+
# |     age|   bigint|   null|
# |    name|   string|   null|
# +--------+---------+-------+
```

Partitions and buckets

- Spark SQL partitions and buckets correspond to Hive partitions and buckets
- For a file source, it is possible to partition and bucket (and sort) the output
- Partition can be used in general:

```
df.write.partitionBy("favorite_color").format("parquet").save("namesPartByColor.parquet")
```

- Bucketing and sorting are applicable only to persistent tables:

```
df.write.bucketBy(42, "name").sortBy("age").saveAsTable("people_bucketed")
```

References

Hive

- Apache Hive Language Manual
- Apache Hive Tutorial
- Apache Hive installation examples: `$HIVE_HOME/examples/queries`
- Google Cloud docs gcloud dataproc jobs submit hive: see here
- Thusoo, A. et al, Hive-A Petabyte Scale Data Warhouse Using Hadoop, ICDE 2010
- Apache Hive Language Manual: Configuration Properties
- Apache foundation Tez, Hive on Tez
- Apache Hive HiveServer2 clients
- Hive JSON SerDe
- Hue: A Hive Web Interface
- He, Y. et al, RCFile: A Fast and Space-efficient Data Placement Structure in MapReduce-based Warehouse Systems, ICDE 2011

References (cont'd)

- Spark SQL programming guide: Spark SQL, DataFrames and Datasets Guide, Apache Spark 2.2.0, 2017
- Armbrust, M., et al, Spark SQL: Relational Data Processing in Spark, ACM SIGMOD 2015
- Armbrust, M., Dataframes: Simple and Fast Analysis of Structured Data, Webinar, 2017
- Spark SQL catalyst optimizer

References (cont'd)

- Presto
  - Presto: Distributed SQL Query Engine for Big Data prestodb.io
  - Sethi et al, Presto: SQL on Everything, ICDE 2019
- BigQuery / Dremel
  - Big Query SQL Reference: Standard and Legacy
  - Dremel made simple with Parquet: an introduction to Parquet format
  - Google developers docs: Protocol Buffer basics: Python
  - Melnik, S. et al, Dremel: Interactive Analysis of Web-Scale Datasets, VLDB 2010
- Other
  - Oracle docs, Java regular expression syntax
  - Chaiken, R. et al, SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets, VLDB 2008
  - Karau, H., Improving PySpark performance: Spark performance beyond the JVM
  - Slee, M., Agarwal, A., and Kwiatkowski, M., Thrift: Scalable Cross-Language Services Implementation, White Paper, 2007

References: books

Hive

- White, T., Hadoop: The Definitive Guide, O'Reilly, 4th Edition, 2015
  - Ch 17: Hive, Ch 12: Avro, Ch 13: Parquet
- Rutherglen, J., Wampler, D., Capriolo, E., Programming Hive, 2nd Edition, O'Reilly, 2017
- Prokopp, C., The Free Hive Book

Spark SQL

- Karau, H., Konwinski, A., Wendell, P. and Zaharia, M., Learning Spark: Lightning-fast Data Analysis, Chapter 9 Spark SQL, O'Reilly, 2015
  - Ch 9: Spark SQL
- Karau, H. and Warren, R., High Performance Spark: Best Practices for Scaling & Optimizing Apache Spark, O'Reilly, 2017
  - Ch 3: DataFrames, Datasets, and Spark SQL, Ch 4: Joins (SQL and Core)
- Drabas, T. and Lee D., Learning PySpark, Packt, 2016
  - Ch 3: DataFrames

Seminar class 4: Hive and Spark SQL

- Hive
  - Get started with using Hive on GCP
  - Run Hive queries
- Spark SQL
  - Query data using Spark DataFrame API and Spark SQL
- Using Spark SQL to query a Hive database

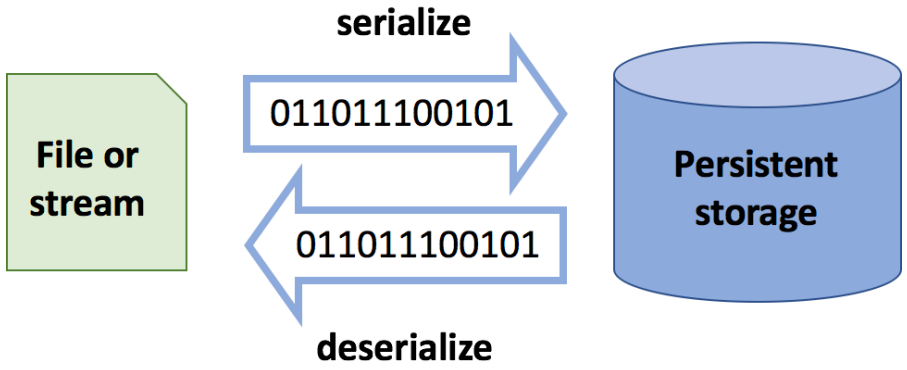
https://github.com/lse-st446/lectures2021/blob/master/Week04/class/README.md

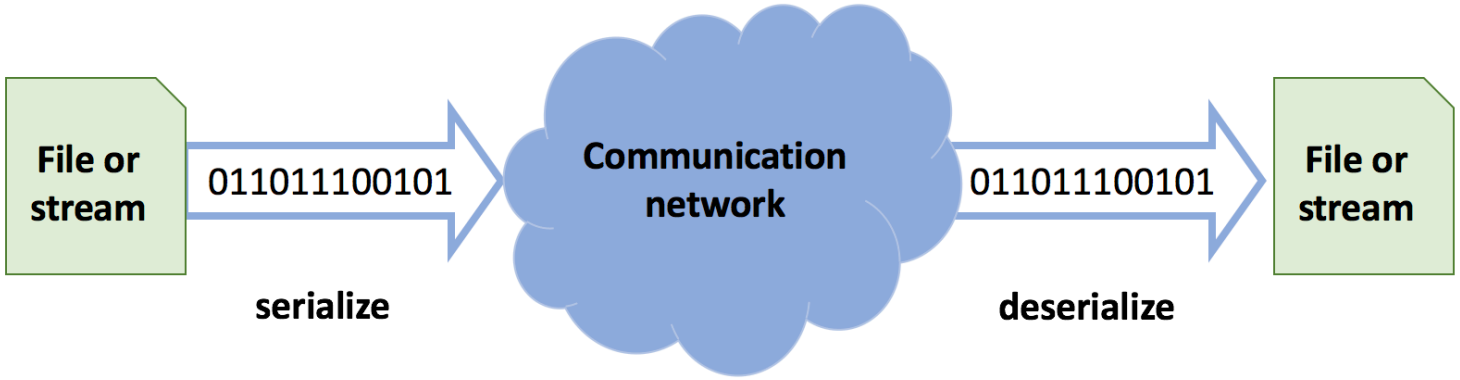
Extras

Data storage formats

- Storage of a table in Hive is determined by the row format and the file format
- Row format: dictates how rows and the fields in a row are stored
  - Defined by a serializer-deserializer (SerDe)
- File format: dictates the container format for fields in a row
- Example file formats:
  - TextFile
  - Avro
  - Parquet
  - SequenceFile
  - RCFile
  - ORCFile
- All above formats are binary except for TextFile which is textual
- File formats can be row-oriented (ex TextFile and Avro) or column-oriented (ex Parquet, RCFile and ORCFile)

Serialization and deserialization





Parquet files

- Spark SQL supports reading and writing Parquet files that automatically preserves the schema of the original data

Example:

```
peopleDF = spark.read.json("examples/src/main/resources/people.json")

# DataFrames can be saved as Parquet files, maintaining the schema information
peopleDF.write.parquet("people.parquet")

# Read in the Parquet file created above (parquet files are self-describing so the schema is preserved)
parquetFile = spark.read.parquet("people.parquet")

# Parquet files can also be used to create a temporary view and then used in SQL statements
parquetFile.createOrReplaceTempView("parquetFile")
teenagers = spark.sql("SELECT name FROM parquetFile WHERE age >= 13 AND age <= 19")
teenagers.show()
# +------+
# |  name|
# +------+
# |Justin|
# +------+
```

- For a partitioned table, data is stored in different directories, with partitioning column values encoded in the path of each partition directory (like Hive)

JSON datasets

- Spark SQL automatically infers the scheme of a JSON dataset and load it as a DataFrame (can be done using `SparkSession.read.json` on a JSON file)

```
# spark is from the previous example
sc = spark.sparkContext

# A JSON dataset is pointed to by path (can be either a single text file or a directory storing text files)
path = "examples/src/main/resources/people.json"
peopleDF = spark.read.json(path)

# The inferred schema can be visualized using the printSchema() method
peopleDF.printSchema()
# root
#  |-- age: long (nullable = true)
#  |-- name: string (nullable = true)

# Creates a temporary view using the DataFrame
peopleDF.createOrReplaceTempView("people")

# SQL statements can be run by using the sql methods provided by spark
teenagerNamesDF = spark.sql("SELECT name FROM people WHERE age BETWEEN 13 AND 19")
teenagerNamesDF.show()
# +------+
# |  name|
# +------+
# |Justin|
# +------+
```

Hive tables

- Spark SQL supports reading and writing data stored in Apache Hive

- When working with Hive, one must instantiate `SparkSession` with Hive support, including connectivity to a persistent Hive metastore, support for Hive serdes, and Hive user-defined functions

- When not configured by `hive-site.xml`, the context automatically creates `metastore_db` in the current directory and creates a directory configured by `spark.sql.warehouse.dir` (default spark-warehouse) in the current directory that the Spark application is started

Using a JDBC/ODBC server

- Spark SQL provides JDBC connectivity, which is useful for connecting other tools to a Spark cluster and for sharing a cluster across multiple clients
  - Any client can cache tables in memory, query them, and so on, and the cluster resources and cached data is shared among all of them
- Spark SQL JDBC server corresponds to the HiveServer2 in Hive
  - It is known as the Thrift server since it uses the Thrift communication protocol

Server: can be launched by running the following in the Spark directory:

```
./sbin/start-thriftserver.shi --master sparkMaster
```

Client (e.g. Beeline: simple SQL shell that let us run commands on the server) can be launched as follows:

```
./bin/beeline -u jdbc:hive2://localhost:10000
```

- Beeline: see Hive docs

Apache Thrift

- Thrift: a reliable, performant communication and data serialization that works across different programming languages
  - Originally developed by Facebook, open sourced April 2007, Apache Incubator May 2008, Apache TLP October 2010

- Some features:
  - Simplicity: Thrift code is simple and approachable, free of unnecessary dependencies
  - Transparency: Thrift conforms to the most common idioms in all languages
  - Consistency: Niche, language-specific features belong in extensions, not the core library
  - Performance: Strive for performance first, elegance second

- To probe further: Apache Thrift

Apache Thrift: Python example

```
def main():
    # Make socket
    transport = TSocket.TSocket('localhost', 9090)
    # Buffering is critical. Raw sockets are very slow
    transport = TTransport.TBufferedTransport(transport)
    # Wrap in a protocol
    protocol = TBinaryProtocol.TBinaryProtocol(transport)
    # Create a client to use the protocol encoder
    client = Calculator.Client(protocol)
    # Connect!
    transport.open()
    client.ping()
    print('ping()')
    sum_ = client.add(1, 1)
```

- Source: here