

# Data/Hive/Hive 2.0 Tutorial Concepts

[\[edit\]](#)

## What is Hive 2.0

[\[edit\]](#)

Hive 2.0 is the next generation infrastructure made with the goal of providing tools to enable easy data summarization, adhoc querying and analysis of detail data. Just like Hive 1.0, Hive 2.0 provides a mechanism to put structure on the data. In addition it also provides a simple query language called QL which is based on SQL and which enables users familiar with SQL to do adhoc querying, summarization and data analysis. At the same time, this language also allows traditional map/reduce programmers to be able to plug in their custom mappers and reducers to do more sophisticated analysis which may not be supported by the built in capabilities of the language.

## What is NOT Hive 2.0

[\[edit\]](#)

Hive is based on hadoop which is a batch processing system. Accordingly, this system does not and cannot promise low latencies on queries. The paradigm here is strictly of submitting jobs and being notified when the jobs are completed as opposed to real time queries. As a result it should not be compared with systems like Oracle where analysis is done on a significantly smaller amount of data but the analysis proceeds much more iteratively with the response times between iterations being less than a few minutes. For Hive queries response times for even the smallest jobs can be of the order of 5-10 minutes and for larger jobs this may even run into hours.

In the following sections we provide a tutorial on the capabilities of the system. We start by describing the concepts of data types, tables and partitions (which are very similar to what you would find in a traditional relational database) and then illustrate the capabilities of the language with the help of some examples:

## Data Units

[\[edit\]](#)

In order of granularity - Hive data is organized into:

- **Tables:** Homogenous units of data which have the same schema. An example of a table could be *page\_views* table. Each row of this table could comprise of the following columns, which define the schema of the table:
  - *timestamp* - which corresponds to a datetime type that captures when the page was viewed.
  - *userid* - which is a long integer identifying the user who viewed the page.
  - *page\_url* - which is a string that captures the location of the page.
  - *referer\_url* - which is a string that captures the location of the page from where we arrived at the current page.
  - *IP* - which is a string that captures the IP address from where the page request was made.
- **Partitions:** Each Table can have one or more partition Keys which determines how the data is stored. Partitions - apart from being storage units - also allow the user to efficiently identify the rows that satisfy a certain criteria. e.g a *date\_partition* of type Datestamp and *country\_partition* of type String. Each unique specification of the partition keys defines a partition of the Table e.g. all "US" data from "2008-02-02" is a partition of the *page\_views* table. Therefore, if you have to run analysis on only the "US" data for 2008-02-02, you can run that analysis only on the relevant

- Concepts
  - What is Hive 2.0
  - What is NOT Hive 2.0
  - Data Units
  - Type System
  - Language capabilities
  - Usage and Examples
    - Creating Tables
    - Describing and Showing Tables
    - Loading Data
    - Simple Query
    - Partition Based Query
    - Joins
    - Aggregations
    - Multi Table/File Inserts
    - Inserting into local files
    - Sampling
    - Union all
    - Array Operations
    - Map Operations
    - Custom map/reduce scripts
    - Co groups
    - Useful built in Functions

partition of the table thereby speeding up the analysis significantly.

- Buckets: Data in each partition may in term be divided into Buckets based on hash of some column of the Table. For example the *page\_views* table may be bucketed by *userid* (which is one of the columns of the *page\_view* table. These can be used to efficiently sample the data.

Note that it is not necessary for tables to be partitioned or bucketed. But these abstractions allow us significant optimizations in query processing.

## Type System

[\[edit\]](#)

Types are associated with the columns in the tables. The following *Primitive* types are supported:

- Characters
- Integers (small is 2 bytes , medium is 4 bytes and big is 8 bytes)
- Floating point numbers (Single and Double precision)
- Strings
- Datetime
- Boolean

*Composite* Types can be built up from primitive types and other composite types using:

- Composition (structs/records)
- Maps (key-value tuples)
- Arrays

e.g. a type User may comprise of the following fields:

- id - which is a 4 byte integer.
- name - which is a string.
- age - which is an integer.
- weight - which is a floating point number.
- friends - which is a array of ids(integers).
- gender - which is an integer.
- active - which is a boolean.

## Language capabilities

[\[edit\]](#)

Hive 2.0's query language provides the basic SQL like operations. These operations work on tables or partitions. These operations are:

- Ability to filter rows from a table using a where clause.
- Ability to select certain columns from the table using a select clause.
- Ability to do equi-joins between two tables.
- Ability to evaluate aggregations on multiple group bys columns for the data stored in a table.
- Ability to store the results of a query into another table.
- Ability to download the contents of a table to a local file.
- Ability to store the results of a query in a hadoop dfs file.
- Ability to manage types (create, drop and alter).
- Ability to manage tables and partitions (create, drop and alter).
- Ability to plug in custom scripts in the language of choice for custom map/reduce jobs.

## Usage and Examples

[\[edit\]](#)

The following examples highlight some salient features of the system.

### Creating Tables

[\[edit\]](#)

An example statement that would create the *page\_view* table mentioned above would be like:

```
CREATE TABLE page_view(viewTime DATETIME, userid MEDIUMINT,
```

```

        page_url STRING, referrer_url STRING,
        friends ARRAY<BIGINT>, properties MAP<STRING, STRING>
        ip STRING COMMENT 'IP Address of the User')
COMMENT 'This is the page view table'
PARTITIONED BY(date DATETIME, country STRING)
BUCKETED ON (userid) INTO 32 BUCKETS
ROW FORMAT DELIMITED
        FIELDS TERMINATED BY \001
        COLLECTION ITEMS TERMINATED BY \002
        MAP KEYS TERMINATED BY \003
        LINES TERMINATED BY \012
STORED AS COMPRESSED

```

In this example the columns that comprise of the table row are specified in a similar way as the definition of types. Comments can be attached both at the column level as well as at the table level. Additionally the partitioned by clause defines the partitioning columns which are different from the data columns and are actually not stored in the data. The bucketed on clause specifies which column to use for bucketing as well as how many buckets to create. The delimited row format specifies how the rows are stored in the hive table. In the case of the delimited format, this specifies how the fields are terminated, how the items within collections (arrays or maps) are terminated and how the map keys are terminated. STORED AS compressed indicates that this data is compressed and stored in a binary format (using hadoop SequenceFiles) on hdfs. Other than COMPRESSED, the data may also be stored as TEXT. The values shown for the ROW FORMAT and STORED AS clauses in the above example represent the system defaults.

## Describing and Showing Tables

[\[edit\]](#)

Describing and Showing of tables uses similar syntax as describing and showing of types. Accordingly all of the following statements return table names which match the specified criteria.

```

SHOW TABLES;
SHOW TABLES WHERE name = 'page_view';
SHOW TABLE WHERE COMMENT like '%user table%';

```

In order to look at the entire information of a table one could use a normal describe statement like:

```

DESCRIBE TABLE page_views;

```

## Loading Data

[\[edit\]](#)

There are multiple mechanisms of loading data into Hive tables. The user can create an external table that points to a specified location within hdfs. In this particular usage, the user can copy a file into the specified location using the hdfs put or copy commands and create a table pointing to this location with all the relevant row format information. Once this is done, the user can transform this data and insert into any other Hive tables. e.g. if the file /tmp/pv\_2008-06-08.txt contains comma separated page views served on 2008-06-08, and this needs to be loaded into the page\_view table in the appropriate partition, the following sequence of commands can achieve this:

```

CREATE EXTERNAL TABLE page_view_stg(viewTime DATETIME, userid MEDIUMINT,
        page_url STRING, referrer_url STRING,
        ip STRING COMMENT 'IP Address of the User',
        country STRING COMMENT 'country of origination')
COMMENT 'This is the staging page view table'
ROW FORMAT DELIMITED FIELDS TERMINATED BY \054 LINES TERMINATED BY \012
LOCATION '/user/facebook/staging/page_view';

```

```

hadoop dfs -put /tmp/pv_2008-06-08.txt /user/facebook/staging/page_view

```

and finally

```

FROM page_view_stg pvs
INSERT OVERWRITE TABLE page_view PARTITION(date=2008-06-08, country='US')

```

```
SELECT pvs.viewTime, pvs.userid, pvs.page_url, pvs.referrer_url, null, null, pvs.ip
WHERE pvs.country = 'US';
```

In the example above nulls are inserted for the array and map types in the destination tables but potentially these can also come from the external table if the proper row formats are specified.

This method is useful if there is already legacy data in hdfs on which the user wants to put some metadata so that that the data can be queried and manipulated using hive.

Additionally, the system also supports syntax that can load the data from a file in the local files system directly into a hive table for the case when no transformations are required. If /tmp/pv\_2008-06-08\_us.txt already contains the data for US, then we do not need any additional filtering as shown in the previous example. The load in this case can be done using the following syntax:

```
LOAD DATA LOCAL INFILE `/tmp/pv_2008-06-08_us.txt` INTO TABLE page_view
PARTITION(date=2008-06-08, country='US')
FIELDS TERMINATED BY \054 LINES TERMINATED BY \012;
```

In the case that the input file /tmp/pv\_2008-06-08\_us.txt is very large, the user may decide to do a parallel load of the data. For parallel load to work, the file location should be accessible through an nfs mount point to all the machines in the cluster. The following syntax can achieve the parallel load:

```
LOAD DATA INFILE '/mnt/data/pv_2008-06-08_us.txt' INTO TABLE page_view
PARTITION(date=2008-06-08, country='US')
FIELDS TERMINATED BY \054 LINES TERMINATED BY \012;
```

It is assumed that the array and map fields in the input .txt files are null fields for these examples.

In either cases, the load command figures out if any transformations have to be done in order to change from the input format to the table format in hive. If no transformations are needed then the load command implicitly moves the hdfs files into the hive name space. If the hive table is bucketed then the load command does in fact hash on the bucket column to send rows from the input files to proper table buckets.

## Simple Query

[\[edit\]](#)

For all the active users, one can use the query of the following form:

```
FROM user
INSERT OVERWRITE TABLE user_active
SELECT user.*
WHERE user.active = true;
```

Note that unlike SQL, we always insert the results into a table. We will illustrate later how the user can inspect these results and even dump them to a local file.

## Partition Based Query

[\[edit\]](#)

What partitions to use in a query is determined automatically by the system on the basis of where clause conditions on partition columns. e.g. in order to get all the page\_views in the month of 03/2008 referred from domain xyz.com, one could write the following query:

```
FROM page_views
INSERT OVERWRITE TABLE xyz_com_page_views
SELECT page_views.*
WHERE page_views.date >= 2008-03-01 AND page_views.date <= 2008-03-31 AND
page_views.referrer_url like '%xyz.com';
```

## Joins

[\[edit\]](#)

In order to get a demographic breakdown(by gender) of page\_view of 2008-03-03 one would need to join the page\_view table and the user table on the userid column. This can be accomplished with a

join as shown in the following query:

```
FROM page_view pv JOIN user u ON (pv.userid = u.id)
INSERT OVERWRITE TABLE pv_users
SELECT pv.*, u.gender, u.age
WHERE pv.date = 2008-03-03;
```

Note that in Hive 2.0 we only support equi-joins. In order to do outer joins the user can qualify the join with LEFT OUTER, RIGHT OUTER or FULL OUTER keywords in order to indicate the kind of outer join (left preserved, right preserved or both sides preserved). e.g. in order to do a full outer join in the query above, the corresponding syntax would look like the following query:

```
FROM page_view pv FULL OUTER JOIN user u ON (pv.userid = u.id)
INSERT OVERWRITE TABLE pv_users
SELECT pv.*, u.gender, u.age
WHERE pv.date = 2008-03-03;
```

In order to join more than one tables, the user can use the following syntax:

```
FROM page_view pv JOIN user u ON (pv.userid = u.id) JOIN friend_list f ON (u.id = f.uid)
INSERT OVERWRITE TABLE pv_friends
SELECT pv.*, u.gender, u.age, f.friends
WHERE pv.date = 2008-03-03;
```

## Aggregations

[\[edit\]](#)

In order to count the number of distinct users by gender one could write the following query:

```
FROM pv_users
INSERT OVERWRITE TABLE pv_gender_sum
SELECT pv_users.gender, count (DISTINCT pv_users.userid)
GROUP BY pv_users.gender;
```

Multiple aggregations can be done at the same time, however, no two aggregations can have different DISTINCT columns .e.g while the following is possible

```
FROM pv_users
INSERT OVERWRITE TABLE pv_gender_agg
SELECT pv_users.gender, count(DISTINCT pv_users.userid), count(), sum(DISTINCT
pv_users.userid)
GROUP BY pv_users.gender;
```

however, the following query is not allowed

```
FROM pv_users
INSERT OVERWRITE TABLE pv_gender_agg
SELECT pv_users.gender, count(DISTINCT pv_users.userid), count(DISTINCT pv_users.ip)
GROUP BY pv_users.gender;
```

In hive we also support the ability to compute multiple group bys on the same data stream, with each group by populating a different table. e.g. if suppose we want to aggregate by gender and by ip and get the counts of unique users for each, the following query syntax can be used:

```
FROM pv_users
INSERT OVERWRITE TABLE pv_gender_uu
    SELECT pv_users.gender, count(DISTINCT pv_users.userid)
    GROUP BY pv_users.gender
INSERT OVERWRITE TABLE pv_ip_uu
    SELECT pv_users.ip, count(DISTINCT pv_users.id)
    GROUP BY pv_users.ip;
```

## Multi Table/File Inserts

[\[edit\]](#)

The output of the aggregations or simple selects can be further sent into multiple tables or even to hadoop dfs files (which can then be manipulated using hdfs utilities). e.g. if along with the gender breakdown, one needed to find the breakdown of unique page views by age, one could accomplish that with the following query:

```
FROM pv_users
INSERT OVERWRITE TABLE pv_gender_sum
  SELECT pv_users.gender, count_distinct(pv_users.userid)
  GROUP BY pv_users.gender
INSERT OVERWRITE DIRECTORY '/user/facebook/tmp/pv_age_sum.txt'
  SELECT pv_users.age, count_distinct(pv_users.userid)
  GROUP BY pv_users.age;
```

The first insert clause sends the results of the first group by to a Hive table while the second one sends the results to a hadoop dfs files.

## Inserting into local files

[\[edit\]](#)

In certain situations you would want to write the output into a local file so that you could load it into an excel spreadsheet. This can be accomplished with the following command:

```
FROM pv_gender_sum
INSERT OVERWRITE LOCAL DIRECTORY '/tmp/pv_gender_sum.csv'
  FIELD DELIMITER ',' ROW DELIMITER \013
  SELECT pv_gender_sum.*
```

## Sampling

[\[edit\]](#)

The sampling clause allows the users to right queries on samples of the data instead of the whole data. Currently the sampling is done on the columns that are specified in the BUCKETED ON clause of the CREATE TABLE statement. In the following example we choose 1 out of the 64 buckets of the pv\_gender\_sum table. Suppose the bucketing is done in order to create 32 buckets, the following example would get every other row from one of the randomly selected buckets of the table:

```
FROM pv_gender_sum TABLESAMPLE(BUCKET 1 OUT OF 64)
INSERT OVERWRITE TABLE pv_gender_sum_sample
  SELECT pv_gender_sum.*;
```

## Union all

[\[edit\]](#)

The language also supports union all .e.g. if suppose there are two different tables that track which user has published a video and which user has published a comment, the following query joins the results of a union all with the user table to create a single annotated stream for all the video publishing and comment publishing events:

```
FROM (
  FROM action_video av
  SELECT av.uid AS uid
  WHERE av.date = 2008-06-03
  UNION ALL
  FROM action_comment ac
  SELECT ac.uid AS uid
  WHERE ac.date = 2008-06-03
) actions JOIN users u ON(u.id = actions.uid)
INSERT OVERWRITE TABLE actions_users
  SELECT u.id, actions.date;
```

## Array Operations

[\[edit\]](#)

The user can get a specific element in the array by its index or a subarray of the array by using the following command:

```
FROM page_views pv
```

```
INSERT OVERWRITE page_view_array
  SELECT pv.friends[2], pv.friends[1:10], pv.friends[3:];
```

The select expressions get the second friend, the subarray of the friends array containing elements from 1 to 10 (or less number of elements if there are less than 10 elements in the subarray) and a subarray from the 3rd element to the last element respectively.

Additionally, Hive also supports COLLECT and EXPLODE operators to create create arrays and to convert arrayss into multiple rows .e.g in order to create an array of all page urls that a user has visited, the following command can be used:

```
FROM page_view pv
INSERT OVERWRITE page_view_array
  SELECT pv.userid, COLLECT(pv.page_url)
  GROUP BY pv.userid;
```

Similarly in order to explode the created array, the following command can be used:

```
FROM page_view_array pva
INSERT OVERWRITE page_view_expl
  SELECT pva.userid, EXPLODE(pva.user_array);
```

Additionally, the user can also get the length of the array using the # operator as shown below:

```
FROM page_view pv
INSERT OVERWRITE page_view_len
  SELECT pv.userid, #pv.friends;
```

## Map Operations [\[edit\]](#)

Maps provide collections similar to associative arrays. Accordingly, in the following command:

```
FROM page_views pv
INSERT OVERWRITE page_views_map
  SELECT pv.userid, pv.properties['page type'];
```

allows one to select the 'page\_type' property from the page\_views table.

## Custom map/reduce scripts [\[edit\]](#)

Users can also plug in their own custom mappers and reducers in the data stream by using features natively supported in the Hive 2.0 language. e.g. in order to run a custom mapper script - map\_script - and a custom reducer script - reduce\_script - the user can issue the following command which used the TRANSFORM clause to embed the mapper and the reducer scripts:

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

## Co groups [\[edit\]](#)

Amongst the user community using map/reduce, cogroup is a fairly common operation wherein the data from multiple tables are sent to a custom reducer such that the rows are grouped by the values of certain columns on the tables. With the UNION ALL operator and the CLUSTER BY specification, this can be achieved in the Hive query language in the following way. Suppose we wanted to cogroup the rows from the actions\_video and action\_comments table on the uid column and send them to the 'reduce\_script' custom reducer, the following syntax can be used by the user:

```
FROM (
```

```

FROM (
    FROM action_video av
    SELECT av.uid AS uid, av.id AS id, av.date AS date
    UNION ALL
    FROM action_comment ac
    SELECT ac.uid AS uid, ac.id AS id, ac.date AS date
) union_actions
SELECT union_actions.uid, union_actions.id, union_actions.date
CLUSTER BY union_actions.uid) map
INSERT OVERWRITE TABLE actions_reduced
SELECT TRANSFORM(map.uid, map.id, map.date) USING 'reduce_script' AS (uid, id,
reduced_val);

```

## Useful built in Functions

[\[edit\]](#)

Amongst some useful built in functions, Hive supports case statement, e.g. if the user wants to change the gender column from integer to a string representation (1 for 'Male', 2 for 'Female' and any other value for 'Unknown'), the following statement could be used:

```

FROM pv_gender_sum pvgs
INSERT INTO pv_gender_sum_annotated
SELECT CASE WHEN pvgs.gender = 1 THEN 'Male'
           WHEN pvgs.gender = 2 THEN 'Female'
           ELSE 'Unknown'
END CASE, pvgs.count;

```

## Altering Tables

[\[edit\]](#)

In the initial version of hive the user can only add new columns to the table and rename the table. In order to rename the table the following syntax can be used:

```

RENAME TABLE old_table_name TO new_table_name;

```

In order to add columns c1 and c2 of types to the table tab1 the following syntax can be used:

```

ALTER TABLE tab1 ADD COLUMN (c1 MEDIUMINT COMMENT 'a new medium int column', c2
STRING DEFAULT 'def val');

```

Note that a change in the schema (such as the adding of the columns), preserves the schema for the old partitions of the table in case it is a partitioned table. All the queries that access these columns and run over the old partitions implicitly return a null value or the specified default values for these columns. Accordingly, once table tab1 has been altered, any queries referring c2 on the old partitions of tab1 will return the value 'def val' while anything referring to c1 will return a null value.

In the later versions we can make the behavior of assuming certain values as opposed to throwing an error in case the column is not found in a particular partition configurable.

## Dropping Tables

[\[edit\]](#)

Dropping tables is fairly trivial. A drop on the table would implicitly drop any indexes (this is a future feature) that would have been built on the table. The associated command is

```

DROP TABLE pv_users;

```

## Future: Interfaces to User Defined Types

[\[edit\]](#)

In the first release the system will not support explicit type creation by the user. However, in the future releases we will support a much more rich interface to support manipulation of types. Some illustrative examples of what can be done with types in future versions of hive are described in the following sections.

## Creating Types

[\[edit\]](#)

The User type mentioned in the previous sections can be created using the following statement:

```
CREATE TYPE User AS OBJECT(id MEDIUMINT COMMENT 'Id of the user',
                           name STRING,
                           age SMALLINT,
                           weight FLOAT,
                           friends ARRAY<MEDIUMINT>,
                           gender TINYINT,
                           active BOOLEAN DEFAULT true)
COMMENT 'This is a user type';
```

Creating a type involves specifying the name of the type, the name of the fields and the types of each of the fields. Additionally default values can be specified as shown in the case of the "active" field in the user type created above. Comments can be associated with the fields as shown in the case of the "id" field. Comments can also be associated with the type as shown above.

In case the table has to be created on a predefined type, the following statement can be used

```
CREATE TABLE user USING TYPE User
COMMENT 'The user table';
```

There are certain default values used for ROW\_FORMAT, STORED AS and LOCATION in case these are not specified.

## Describing and Showing Types

[\[edit\]](#)

In order to list all the types as well as the types that satisfy certain restrictions you can run commands like:

```
SHOW TYPES;
SHOW TYPES WHERE name = 'User';
SHOW TYPES WHERE comment like '%user type%';
```

These would list the names of all the types, all the information corresponding to the User type and the name of the types whose comment field has a 'user type' string embedded in it. In order to describe a specific type one could use the following statement as well:

```
DESCRIBE TYPE User;
```

## Altering Types

[\[edit\]](#)

Type can only be altered in case it is not being used by any table or any other type. New attributes can be added to a type as illustrated in the following example where in the user is adding a new attribute new\_attr to a pre-existing type typ1:

```
ALTER TYPE typ1 ADD ATTRIBUTE(new_attr STRING);
```

Similar to renaming tables, the types can also be renamed e.g. the following statement would rename the type old\_type to new\_type:

```
RENAME TYPE old_type TO new_type;
```

## Dropping Types

[\[edit\]](#)

In order to drop a type the user has to ensure that the type is not being used by any other type or a table. In case such a dependency is discovered, the system would throw an error indicating the anomaly. The associated command is

```
DROP TYPE User;
```

## Other Future Features

[\[edit\]](#)

### Useful Operators

[\[edit\]](#)

We would be adding some other useful operators that we do not have in the first release. Some useful

ones that are missing include:

- HAVING clause
- ORDER BY or TOP N
- More functions on maps and lists

## Unloading data from Hive to an RDBMS using through JDBC

[\[edit\]](#)

In the future we would provide mechanisms to unload the data from hive tables into an RDBMS through JDBC. For the first release the best way to do this is to dump out the data in the local directory and then use RDBMS utilities like sqldr for Oracle to upload data to Oracle. We will package this capability to make it seem less to achieve this through the command line interface.

---

[Permalink](#)  
[Help](#)

[What Links Here](#)

[Recent Changes](#)

[Main Page](#)

[Recent changes](#)

[Random page](#)

Maintained by [Marcel Laverdet](#)