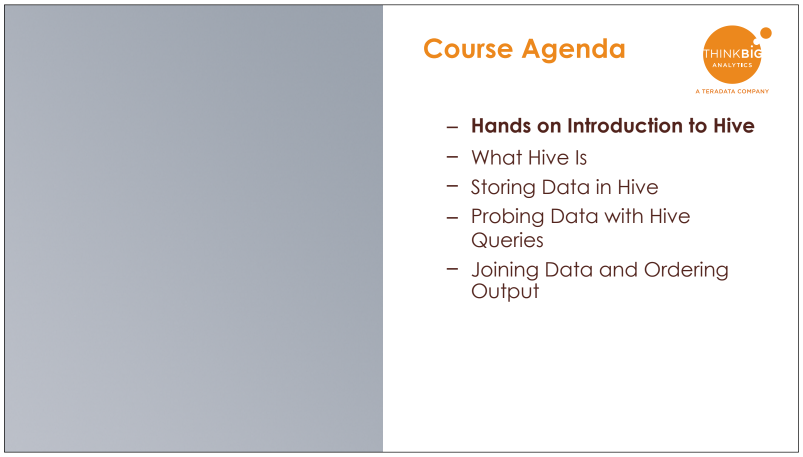
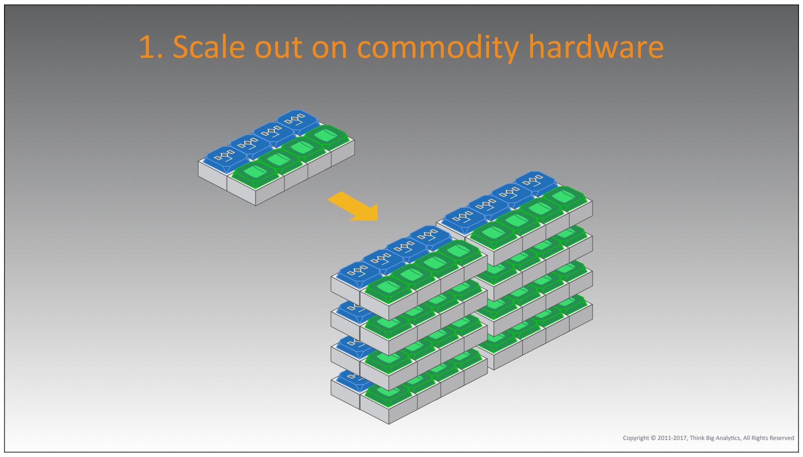
Hive for Spark Developers



Course Agenda

- **Hands on Introduction to Hive**
- What Hive Is
- Storing Data in Hive
- Probing Data with Hive Queries
- Joining Data and Ordering Output

Hive is a killer app, in our opinion, for data warehouse teams migrating to Hadoop, because it gives them a familiar SQL language that hides the complexity of MR programming. We say “data warehousing” because Hive+Hadoop is not a good option for OLTP applications, as we’ll see.



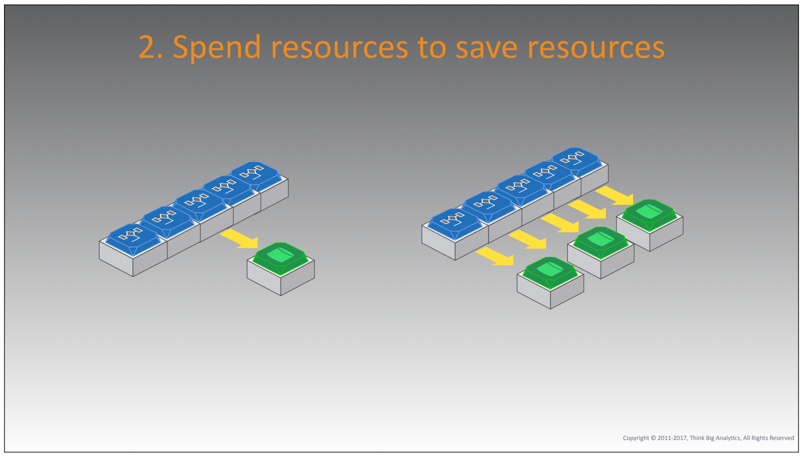
1. Scale out on commodity hardware

Recall our 5 principles of Hadoop clusters. The first is scale out on commodity hardware.

Hadoop doesn't use specialized servers. It uses garden-variety Intel servers you can buy from any of 100 different server vendors.

Further, when you need to upgrade, you don't need forklift upgrades. Instead, Hadoop uses arrays of commodity hardware configured in what's called scale-out. That means if you need to handle 50% more traffic, you don't replace anything or change any software -- you simply buy 50% more servers. <<click for build>>

This is a similar idea to our principle #2.....



2. Spend resources to save resources

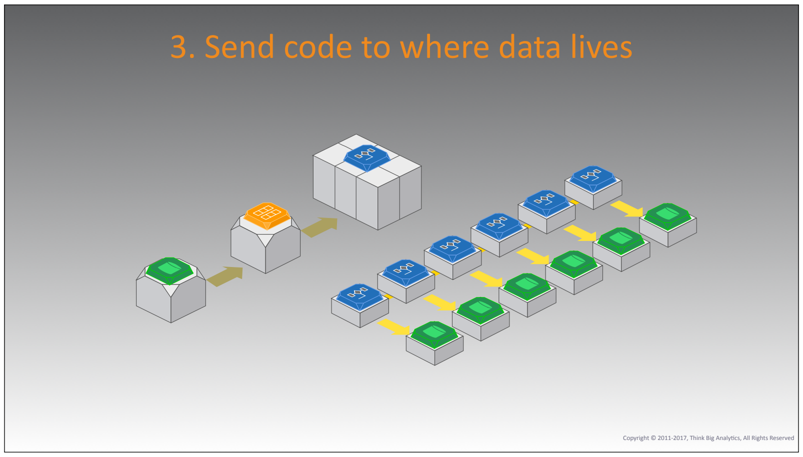
Principle #2 is to spend resources to save resources (usually time)

Typical IT infrastructures have been carefully optimized to make best use of hardware resources. Because Hadoop uses cheap commodity hardware instead of expensive specialized gear, hardware isn't our most precious resource -- it's a commodity! So we are willing to buy extra copies of cheap hardware to save time and to make our computations easier to build. <<click for build>>

An example is thinking about HDFS, the Hadoop Distributed File System. An expensive dedicated storage box would likely use a disk array configured with expensive RAID controllers for reliability. Hadoop takes a different approach, as you'll hear when we talk about HDFS. It simply stores multiple copies of the files on cheap commodity disk drives. If one fails, the software simply routes around it, uses a good copy and creates another copy for further redundancy.

With a 3TB disk costing only about $90, it makes sense to just keep more copies. We're spending resources to save resources.

Our third principle is the first that really comes about because of Big Data....



3. Send code to where data lives

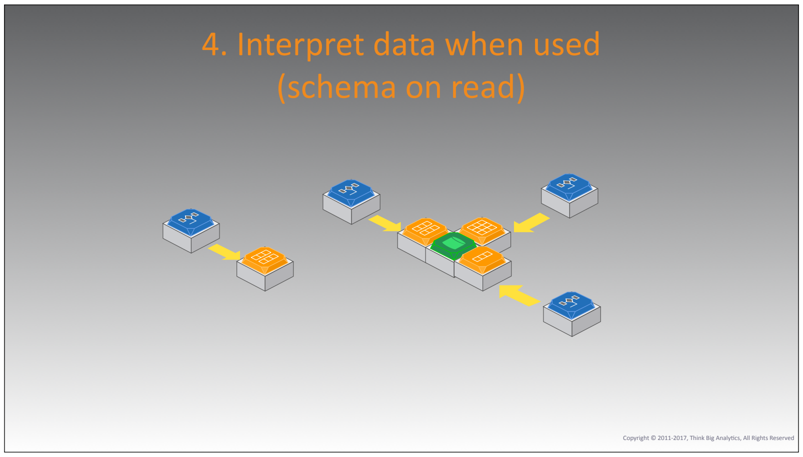
Principle #3: Send code to where the data lives.

In the olden days of mainframe computing, when a company would update its billing records for all its customers, the mainframe would send instructions to the disk controllers to fetch the data. The disk controllers would in turn ask the disks, and all that data would roll into the CPU. The CPU would do its updates, and then send the data back down the channel to the disk controller who would promptly write it to disk.

That worked fine when databases were megabytes or perhaps gigabytes. After all, that wasn't that much bigger than the programs we were running.

But today's clusters deal with Big Data -- data measured in the Terabyte and Petabyte range. That data takes a long time to move, no matter how fast your processors. Meanwhile, our programs haven't really grown that much. So instead of moving the data to the CPU, Hadoop moves the code to where the data lives. <<click for build>>. After all, the code is only megabytes, but the data is terabytes -- moving the cost is thousands of times faster.

Principle #4 is the one that tends to flummox some Database Admins....



4. Interpret data when used (schema on read)

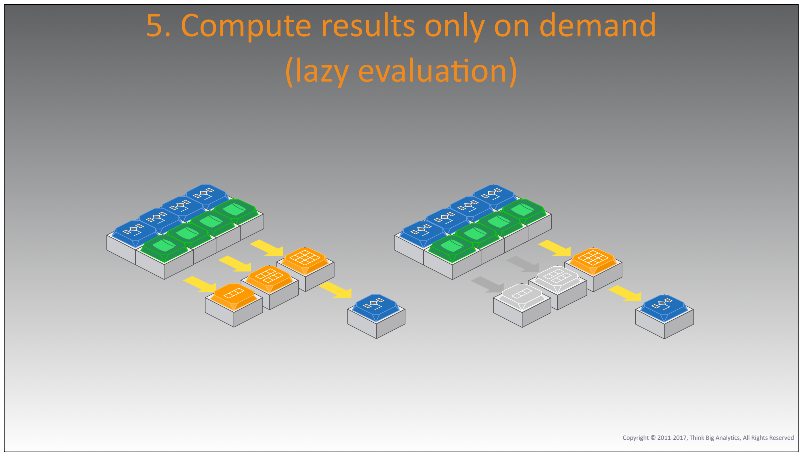
Principal #4 is Interpret Data When Used. This is also called Schema on Read.

Traditional databases relied on a different model: Schema on Write. The programmer would define a schema when she or he created a table, and the database would reserve space on the disk according to that schema to ensure that it would have room for those fields. That's Schema on Write.

With Schema on Read, we don't do anything when a schema is created. Instead, we use schemas to interpret existing files. That means that for a single file, one program could interpret that file as 2 tables <<click for build>>, another program could interpret that same file as only one table <<click for build>, a third program might interpret that as 3 tables. <<click for build>>.

This approach gives us incredible flexibility in how we process Big Data. It also means we can work with datasets even when they have no table schemas at all.

Finally, we have principle #5....



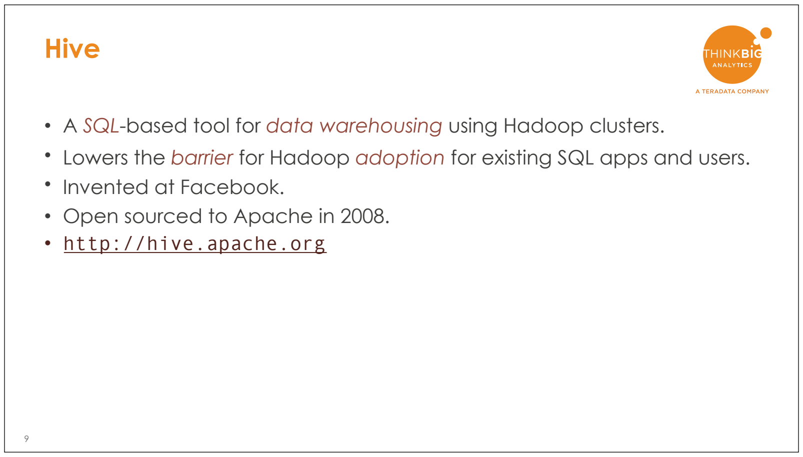
5. Compute results only on demand (lazy evaluation)

Principle #5 is to compute results only on demand (also known as lazy evaluation).

Rather than running every computation on a schedule, whether its results will be used or not, Hadoop clusters only run computations when their results are requested <<click for build>>. It's kind of like running the computation backwards -- we wait for a request and do the bare minimum to satisfy that request.

Think of this approach being like a college student who never attends class and only studies the materials the night before an exam. If the student knows that the professor will only ask about a certain topic, he or she can simply study that topic and leave the rest for another time (or exam). That's lazy evaluation.

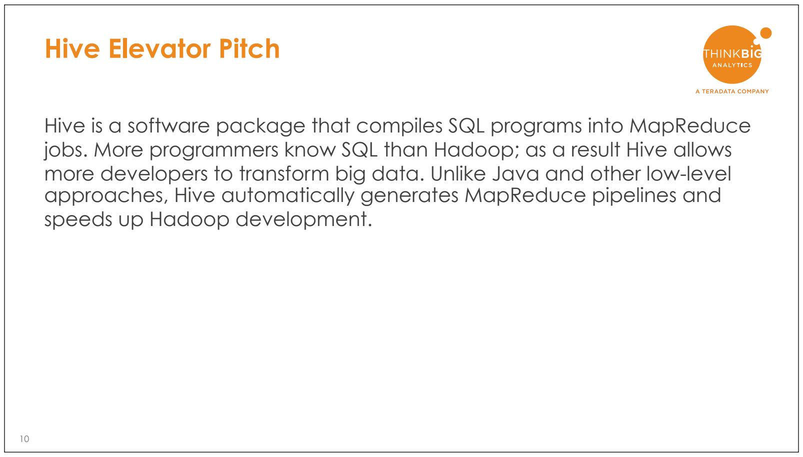
What Is *Hive*?



Hive

- A *SQL*-based tool for *data warehousing* using Hadoop clusters.
- Lowers the *barrier* for Hadoop *adoption* for existing SQL apps and users.
- Invented at Facebook.
- Open sourced to Apache in 2008.
- `http://hive.apache.org`

Hive is a killer app, in our opinion, for data warehouse teams migrating to Hadoop, because it gives them a familiar SQL language that hides the complexity of MR programming. We say “data warehousing” because Hive+Hadoop is not a good option for OLTP applications, as we’ll see.

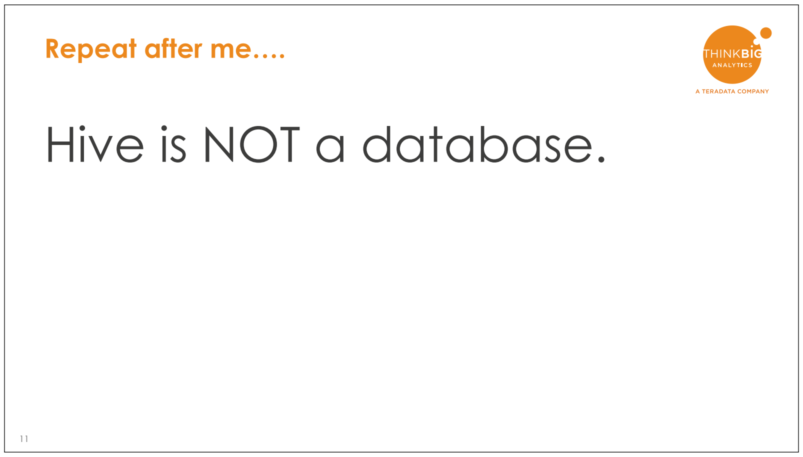


Hive Elevator Pitch

Hive is a software package that compiles SQL programs into MapReduce jobs. More programmers know SQL than Hadoop; as a result Hive allows more developers to transform big data. Unlike Java and other low-level approaches, Hive automatically generates MapReduce pipelines and speeds up Hadoop development.

10

Hive is a killer app, in our opinion, for data warehouse teams migrating to Hadoop, because it gives them a familiar SQL language that hides the complexity of MR programming. We say “data warehousing” because Hive+Hadoop is not a good option for OLTP applications, as we’ll see.

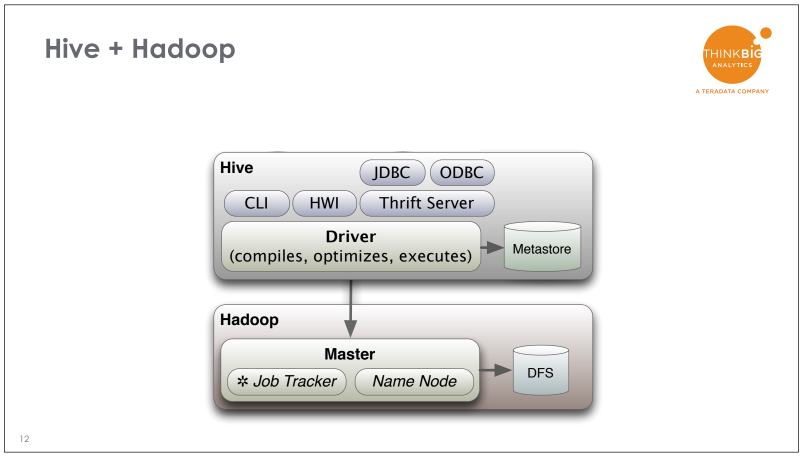


Repeat after me….

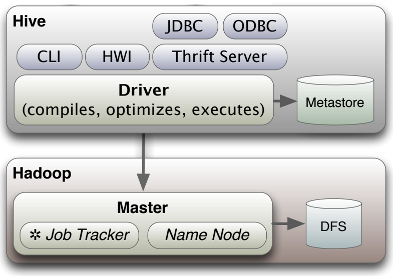
Hive is NOT a database.

11

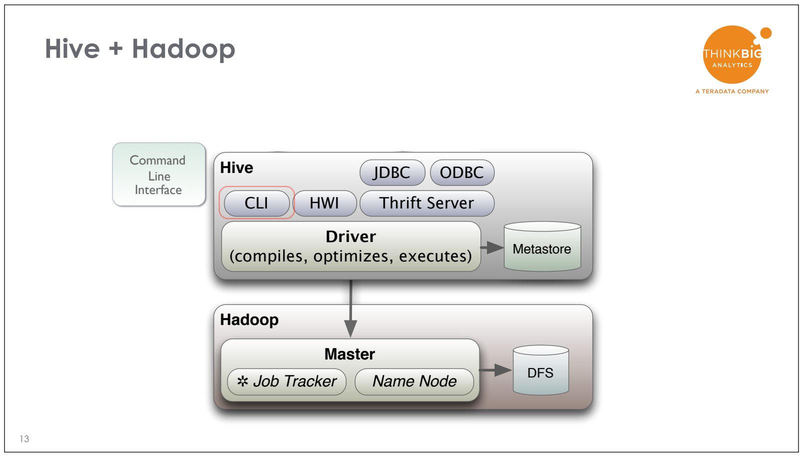
Hive is a killer app, in our opinion, for data warehouse teams migrating to Hadoop, because it gives them a familiar SQL language that hides the complexity of MR programming. We say “data warehousing” because Hive+Hadoop is not a good option for OLTP applications, as we’ll see.



Hive + Hadoop

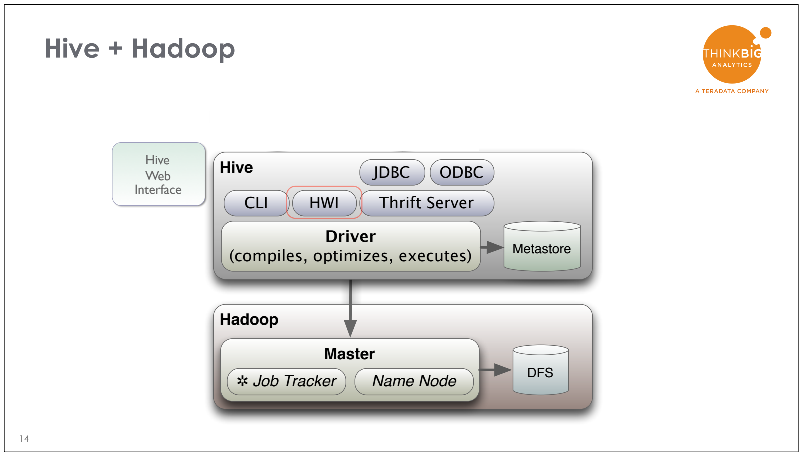


Most Hive queries generate MapReduce jobs. (Some operations don't invoke MapReduce, e.g., those that just write updates to the metastore and "select * from table;" queries.) We've omitted some arrows within the Hive bubble for clarity. They go "down", except for the horizontal connection between the driver and the metastore.



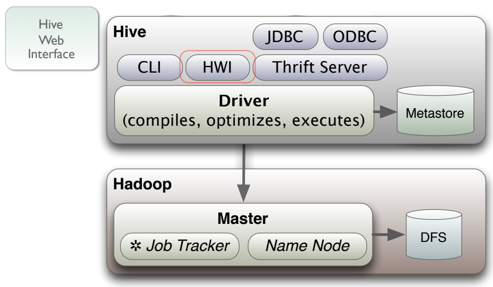
Hive + Hadoop

CLI = Command Line Interface.
HWI = Hive Web Interface.
We'll discuss the operation of the driver shortly.

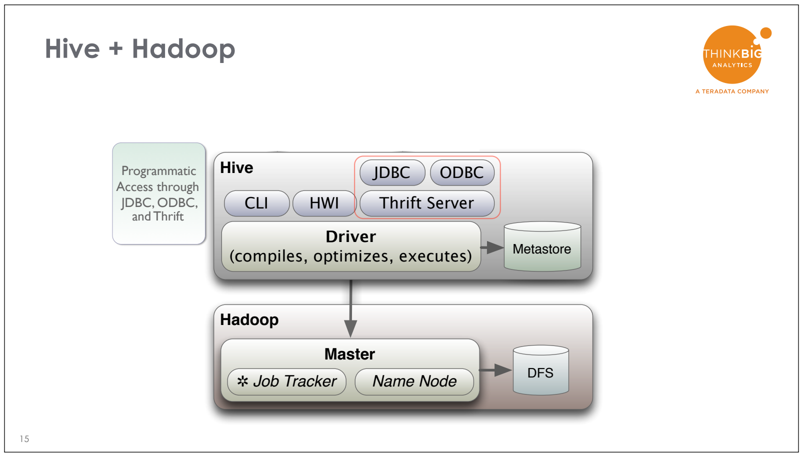


Hive + Hadoop

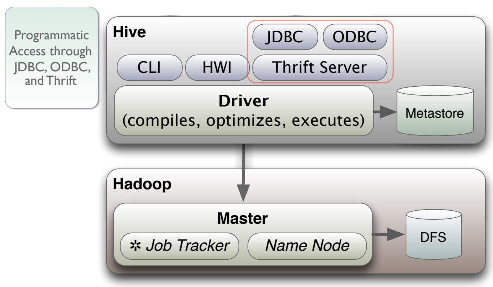
THINKBIG ANALYTICS
A TERADATA COMPANY



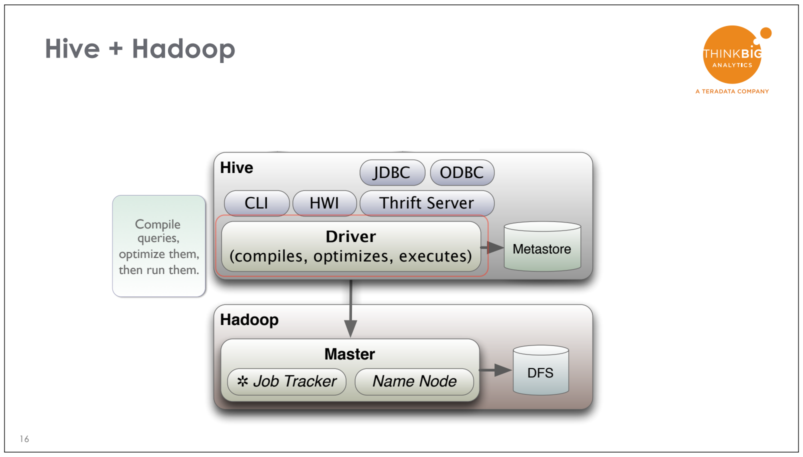
CLI = Command Line Interface.
HWI = Hive Web Interface.
We’ll discuss the operation of the driver shortly.



Hive + Hadoop

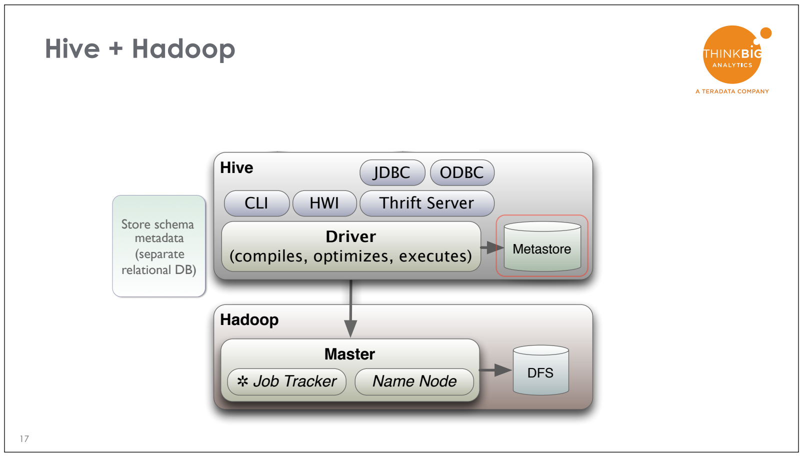


You can also drive Hive from Java programs using JDBC and other languages using ODBC. These interfaces sit on top of a Thrift server, where Thrift is an RPC system invented by Facebook.



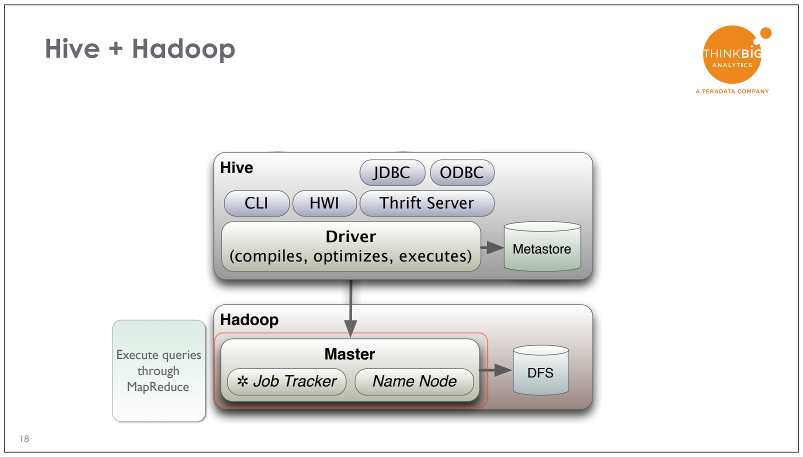
Hive + Hadoop

The Driver compiles the queries, optimizes them, and executes them, by *usually* invoking MapReduce jobs, but not always, as we'll see.



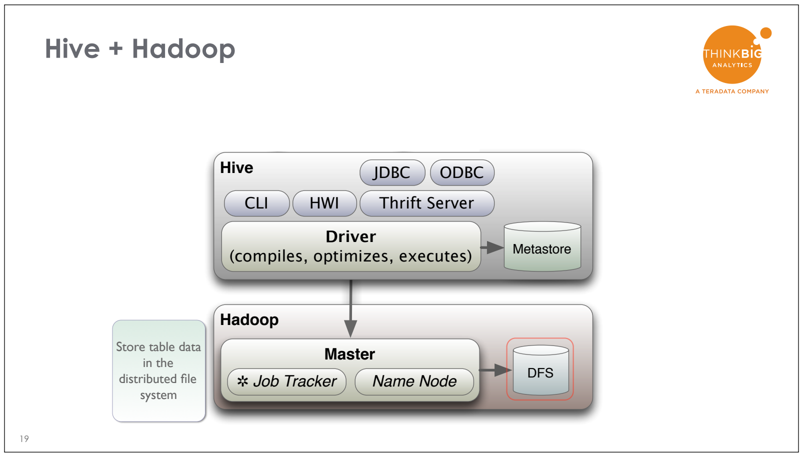
Hive + Hadoop

A separate relational DB is required to store metadata, such as table schema. We'll discuss it more shortly.

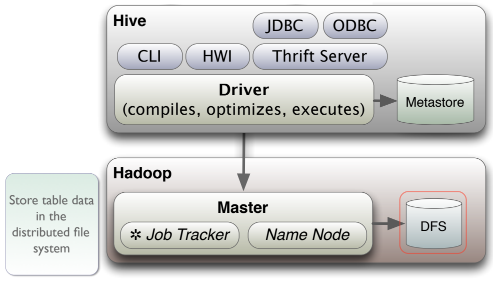


Hive + Hadoop

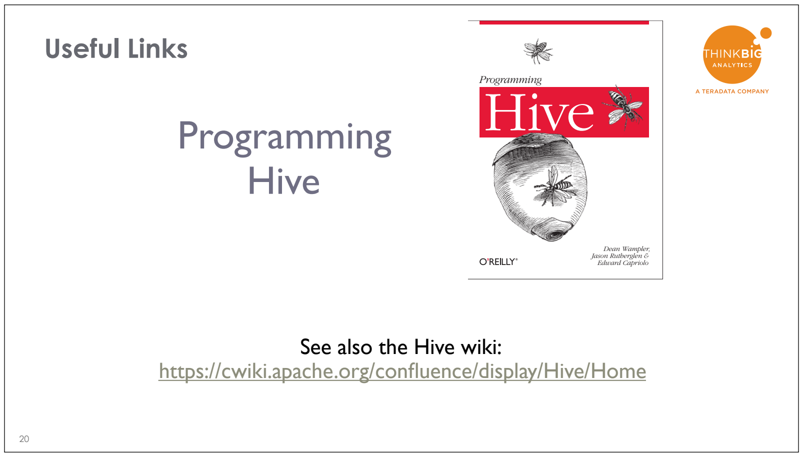
Most Hive queries generate MapReduce jobs. (Some operations don’t invoke MapReduce, e.g., those that just write updates to the metastore and “select * from table;” queries.)



Hive + Hadoop

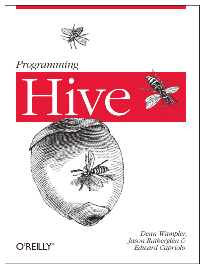


Most Hive queries generate MapReduce jobs. (Some operations don't invoke MapReduce, e.g., those that just write updates to the metastore and "select * from table;" queries.) We've omitted some arrows within the Hive bubble for clarity. They go "down", except for the horizontal connection between the driver and the metastore.



Useful Links

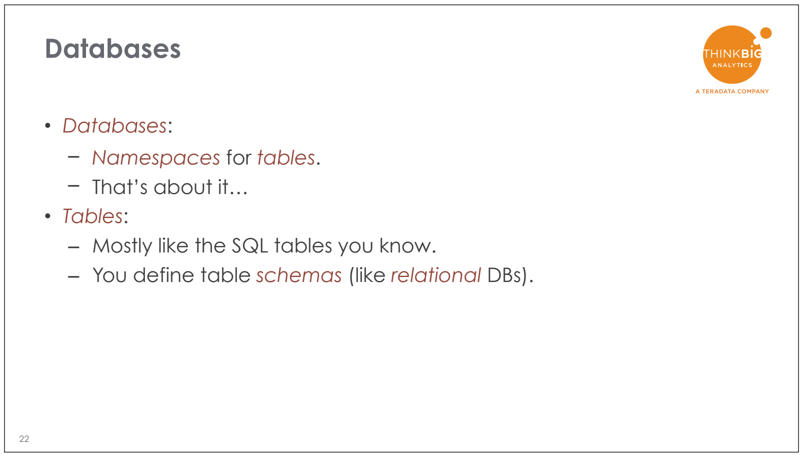
Programming
Hive



See also the Hive wiki:
https://cwiki.apache.org/confluence/display/Hive/Home

This book, written by two former Think Big Analytics consultants, became available in early October 2012. You'll also find lots of documentation the Hive wiki.

Hive Concepts Overview



Databases

- *Databases*:
  - *Namespaces* for *tables*.
  - That's about it…
- *Tables*:
  - Mostly like the SQL tables you know.
  - You define table *schemas* (like *relational* DBs).

22

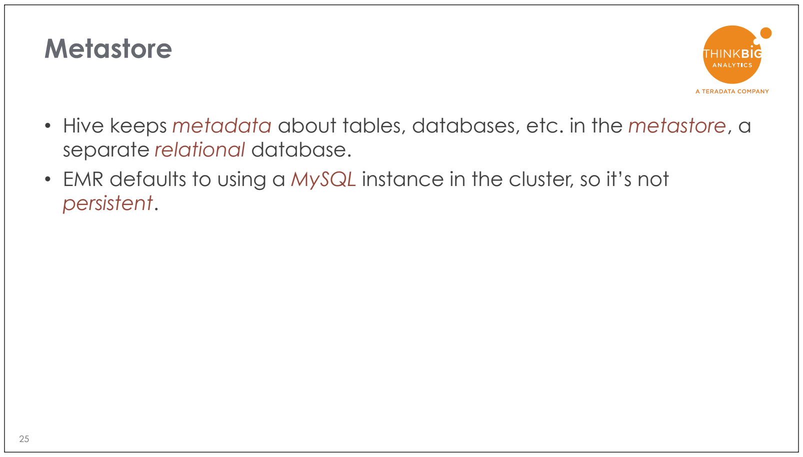
There isn't much to the database concept in Hive. You use them keep you "foo" table separate from other people's "foo" tables. Hive tries to give as much of the familiar SQL ideas as it can, especially for tables. We'll see what's the same and what's different.

Table Storage

- Hive can *own and manage* the storage files...
- Or *you can* own and manage them!
  - They are *external* to Hive.
  - Better for sharing with other tools.
- *File*, *record*, and *field* formats are configurable.

Batch vs. Interactive?

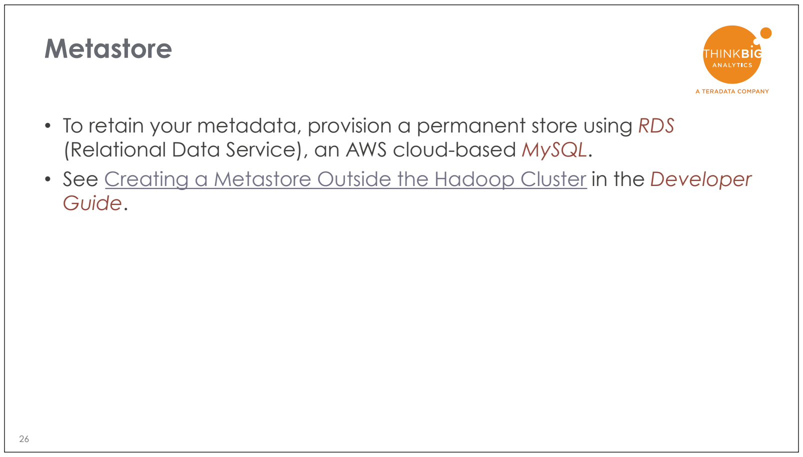
- Traditional databases are good for OLTP (online transaction processing) use.
- Because Hive uses MapReduce, queries have high latency, often minutes, better for:
- Batch mode workflows.
- Data warehouse applications.
- General OLAP (online analytical processing).



Metastore

- Hive keeps *metadata* about tables, databases, etc. in the *metastore*, a separate *relational* database.
- EMR defaults to using a *MySQL* instance in the cluster, so it's not *persistent*.

By default, when you start an cluster for a job flow, a MySQL instance is created, but it gets deleted when the cluster is terminated.

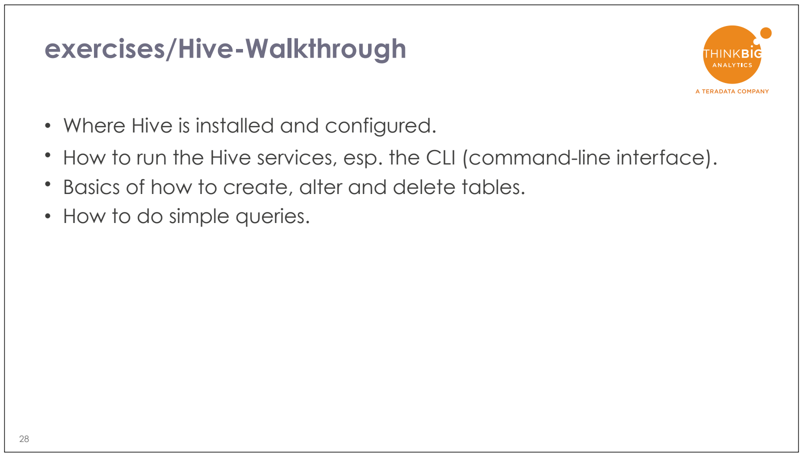


Metastore

- To retain your metadata, provision a permanent store using *RDS* (Relational Data Service), an AWS cloud-based *MySQL*.
- See Creating a Metastore Outside the Hadoop Cluster in the *Developer Guide*.

Using RDS lets you set up a truly persistent store! The “Creating a Metastore Outside the Hadoop Cluster” section (as linked in the API version 2009-11-30 document) in the Developer Guide provides the details for setting this up.

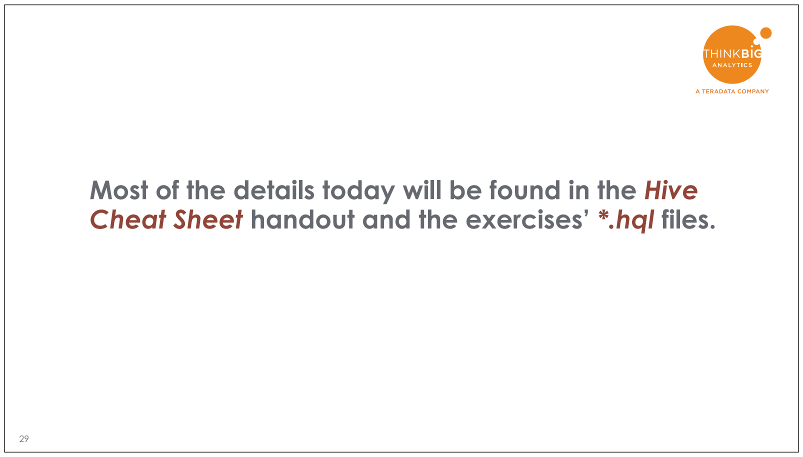
Lab: Hive-Walkthrough



exercises/Hive-Walkthrough

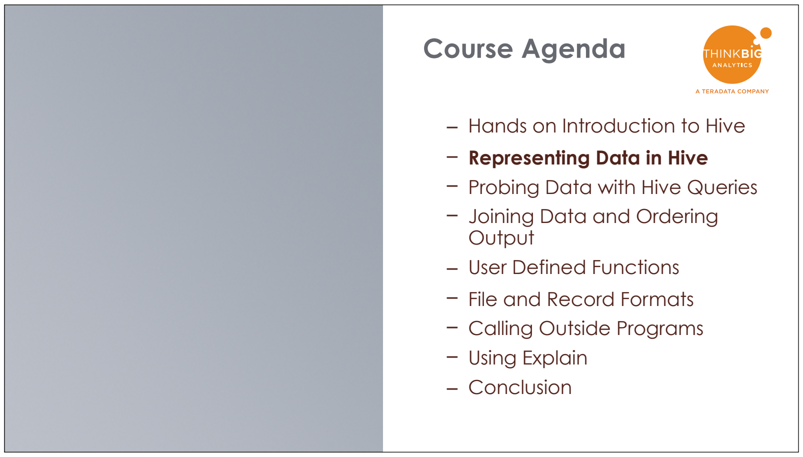
- Where Hive is installed and configured.
- How to run the Hive services, esp. the CLI (command-line interface).
- Basics of how to create, alter and delete tables.
- How to do simple queries.

Walk through the code.



Most of the details today will be found in the ***Hive Cheat Sheet*** handout and the exercises' ***.hql*** files.

These notes will provide high-level summaries, but we'll spend most of our time working out these other documents, which more easily contain complex HiveQL statements and which are easier to copy from for pasting to the hive CLI.

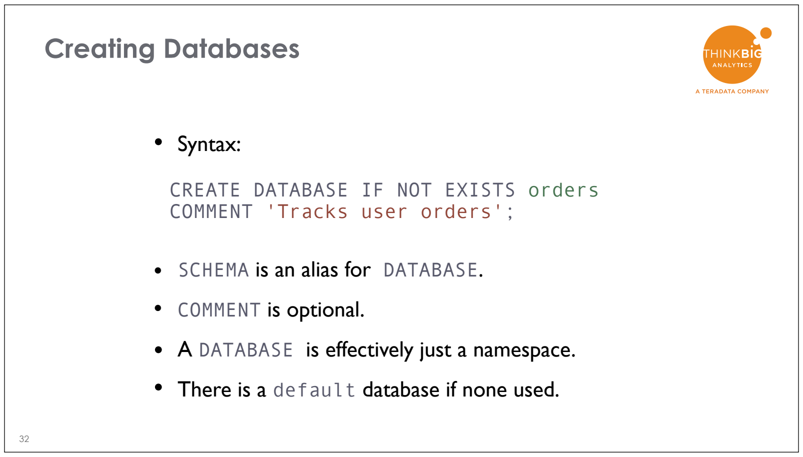


Course Agenda

- Hands on Introduction to Hive
- **Representing Data in Hive**
- Probing Data with Hive Queries
- Joining Data and Ordering Output
- User Defined Functions
- File and Record Formats
- Calling Outside Programs
- Using Explain
- Conclusion

Hive Databases

We’ll start with databases (schemas), then spend most of our time on tables. A later module will briefly discuss indexes and views, two very new features. We’ll cover those later.



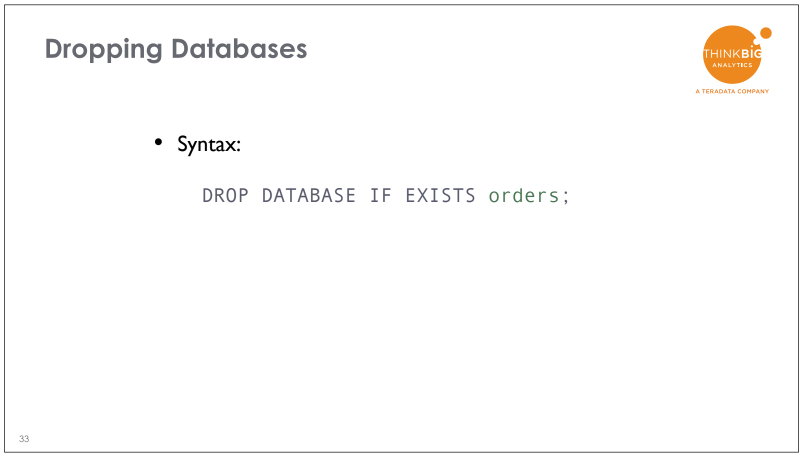
Creating Databases

- Syntax:

```
CREATE DATABASE IF NOT EXISTS orders
COMMENT 'Tracks user orders';
```

- `SCHEMA` is an alias for `DATABASE`.
- `COMMENT` is optional.
- A `DATABASE` is effectively just a namespace.
- There is a `default` database if none used.

I’m omitting some of the more obscure options, including a few that are documented but don’t appear to actually work!
Namespace in the sense that we can create tables like “create table db1.table1 …;”



Dropping Databases

- Syntax:

```
DROP DATABASE IF EXISTS orders;
```

There are RESTRICT | CASCADE options shown in the language manual that cause parse errors!

Creating Hive Tables, Part I

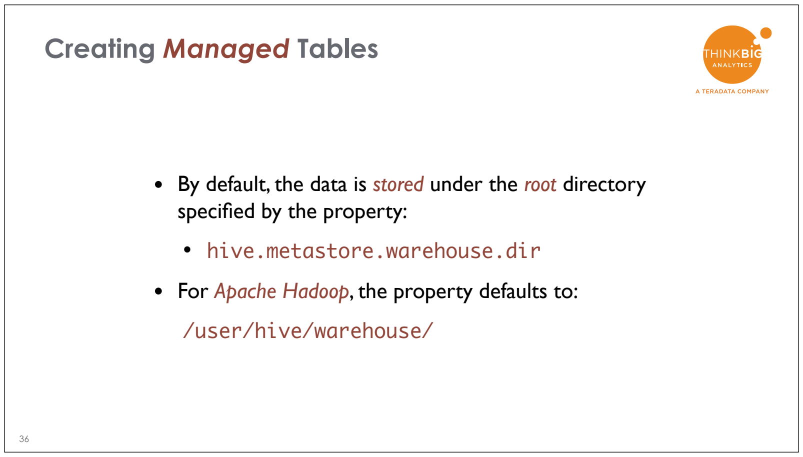
https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL#LanguageManualDDL-CreateTable

Creating *Managed* Tables

- Example:

```
CREATE TABLE IF NOT EXISTS demo1 (
 id INT, name STRING);
```

- `IF NOT EXISTS` optional.
- Suppresses the warning if the table already exists.

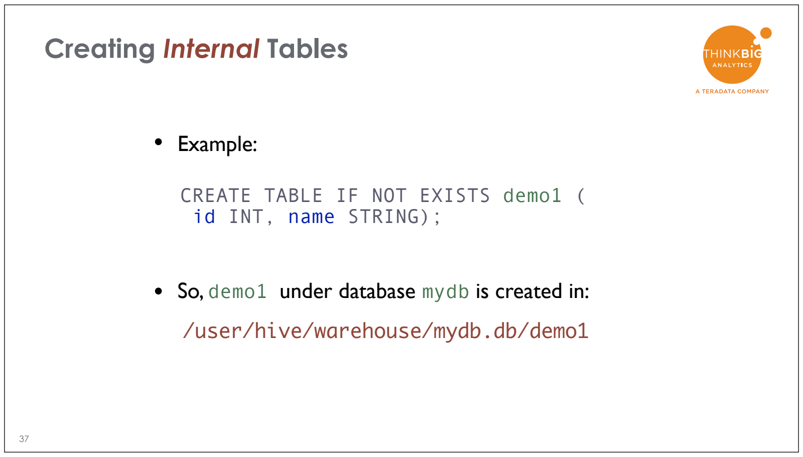


Creating *Managed* Tables

- By default, the data is *stored* under the *root* directory specified by the property:
  - `hive.metastore.warehouse.dir`
- For *Apache Hadoop*, the property defaults to:

  `/user/hive/warehouse/`

“hive.metastore.warehouse.dir” is a property you can set. This directory is in the distributed file system. You can run hadoop fs –ls on this directory.



Creating *Internal* Tables

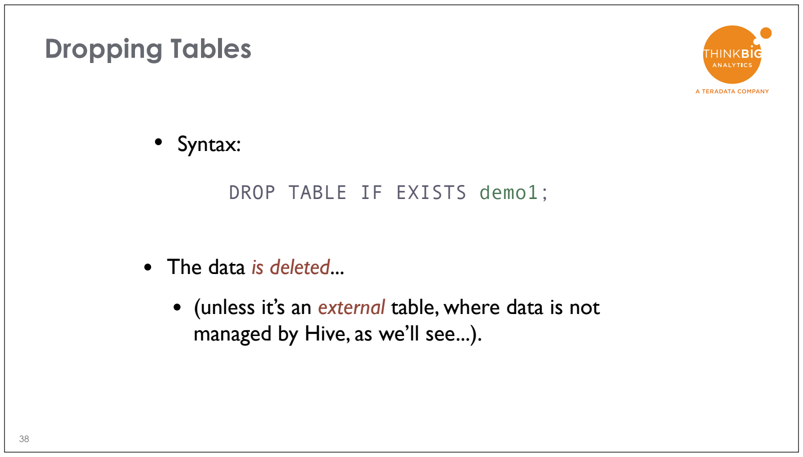
- Example:

```
CREATE TABLE IF NOT EXISTS demo1 (
 id INT, name STRING);
```

- So, `demo1` under database `mydb` is created in:

  `/user/hive/warehouse/mydb.db/demo1`

Note that the directory for the table is under the directory for the database!



Dropping Tables

- Syntax:

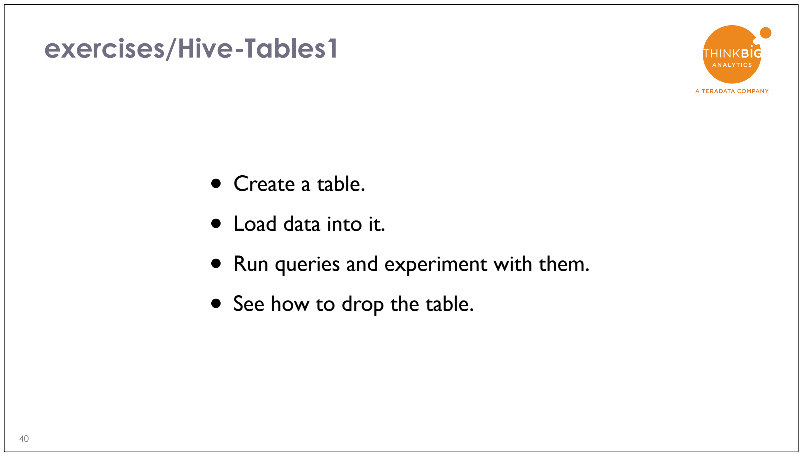
```
DROP TABLE IF EXISTS demo1;
```

- The data *is deleted*...
  - (unless it's an *external* table, where data is not managed by Hive, as we'll see...).

38

As we'll see, external tables are not managed by Hive, but managed by you, "external" to Hive.

Lab: Hive-Tables1



exercises/Hive-Tables1

- Create a table.
- Load data into it.
- Run queries and experiment with them.
- See how to drop the table.

Walk through the code.

Hive Schemas (and File Encodings)

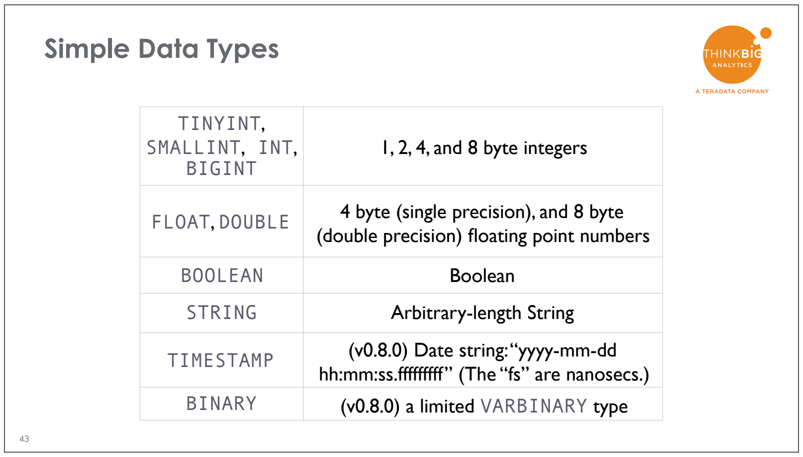
https://cwiki.apache.org/confluence/display/Hive/Tutorial#Tutorial-TypeSystem

Specifying Table Schemas

- Example:

```
CREATE TABLE IF NOT EXISTS demo1
 (id INT, name STRING);
```

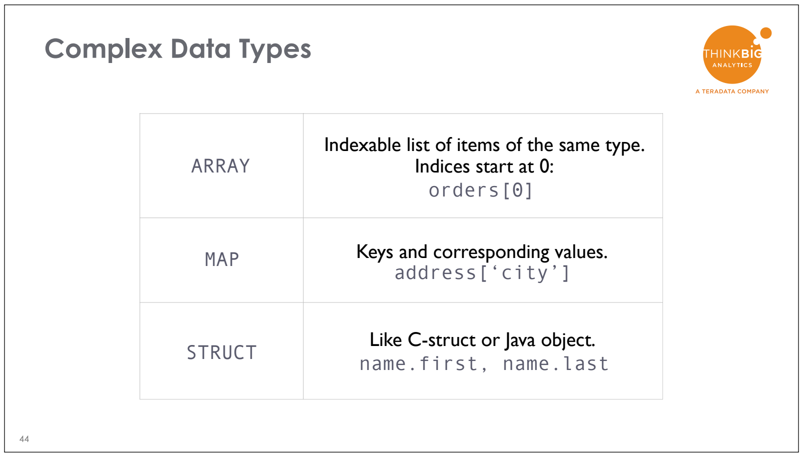
- Two columns:
  - One of type `INT`.
  - One of type `STRING`.



Simple Data Types

TINYINT, SMALLINT, INT, BIGINT	1, 2, 4, and 8 byte integers
FLOAT, DOUBLE	4 byte (single precision), and 8 byte (double precision) floating point numbers
BOOLEAN	Boolean
STRING	Arbitrary-length String
TIMESTAMP	(v0.8.0) Date string: "yyyy-mm-dd hh:mm:ss.fffffffff" (The "fs" are nanosecs.)
BINARY	(v0.8.0) a limited VARBINARY type

All the types reflect underlying Java types. TIMESTAMP and BINARY are new to v0.8.0. Use an a string for pre-0.8.0 timestamps (or BIGINT for Unix epoch seconds, etc.). BINARY has limited support for representing VARBINARY objects. Note that this isn't a BLOB type, because those are stored separately, while BINARY data is stored within the record.

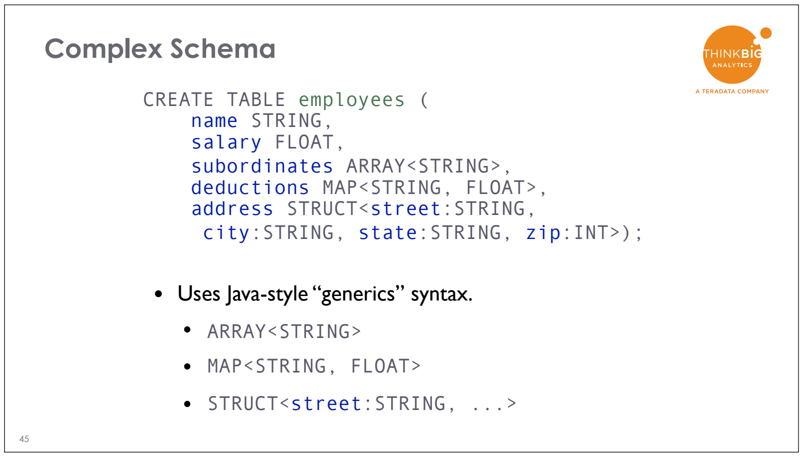


Complex Data Types

`ARRAY`	Indexable list of items of the same type. Indices start at 0: `orders[0]`
`MAP`	Keys and corresponding values. `address['city']`
`STRUCT`	Like C-struct or Java object. `name.first, name.last`

44

All the types reflect underlying Java types. TIMESTAMP and BINARY are new to v0.8.0. Use an integer type or strings for pre-0.8.0. Use BINARY as the last "column" in a schema to as a way of saying "ignore the rest of this record".

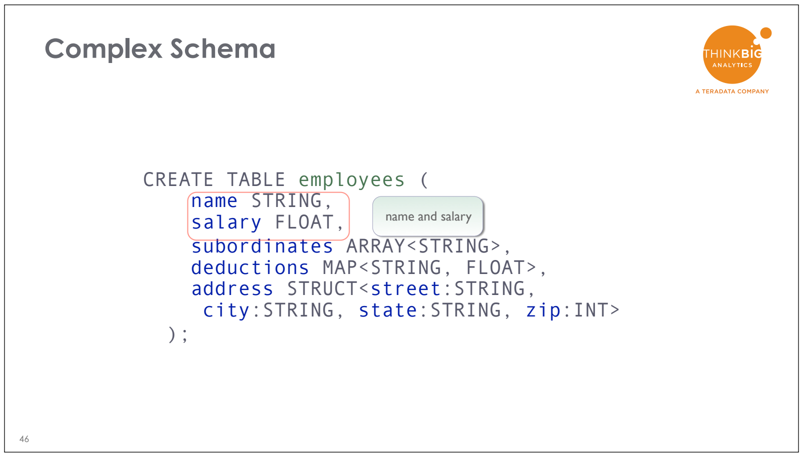


Complex Schema

```
CREATE TABLE employees (
    name STRING,
    salary FLOAT,
    subordinates ARRAY<STRING>,
    deductions MAP<STRING, FLOAT>,
    address STRUCT<street:STRING,
     city:STRING, state:STRING, zip:INT>);
```

- Uses Java-style "generics" syntax.
  - `ARRAY<STRING>`
  - `MAP<STRING, FLOAT>`
  - `STRUCT<street:STRING, ...>`

The <...> is a Java convention. We have to say what type of things the complex values hold. Note that we also name the elements of the STRUCT.

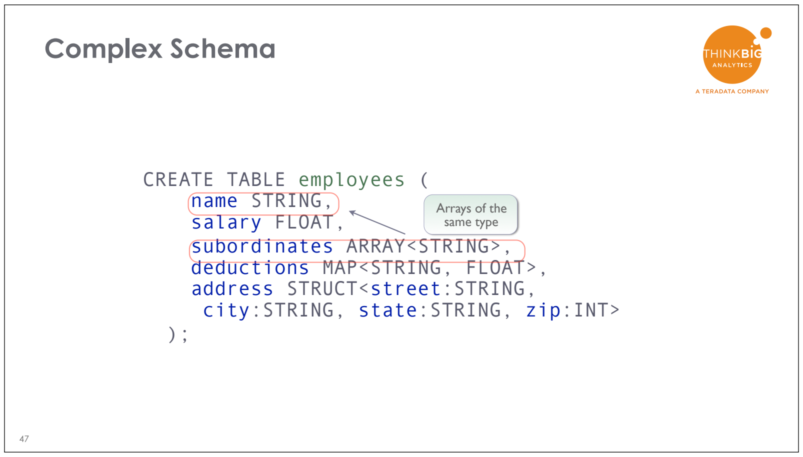


Complex Schema

```
CREATE TABLE employees (
    name STRING,
    salary FLOAT,
    subordinates ARRAY<STRING>,
    deductions MAP<STRING, FLOAT>,
    address STRUCT<street:STRING,
     city:STRING, state:STRING, zip:INT>
  );
```

name and salary

Let’s walk through this...

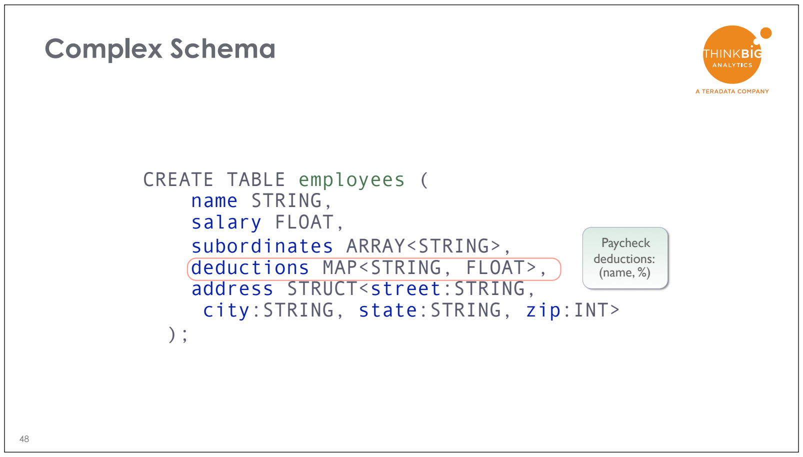


Complex Schema

```
CREATE TABLE employees (
    name STRING,
    salary FLOAT,
    subordinates ARRAY<STRING>,
    deductions MAP<STRING, FLOAT>,
    address STRUCT<street:STRING,
     city:STRING, state:STRING, zip:INT>
  );
```

Arrays of the same type

Let’s walk through this...



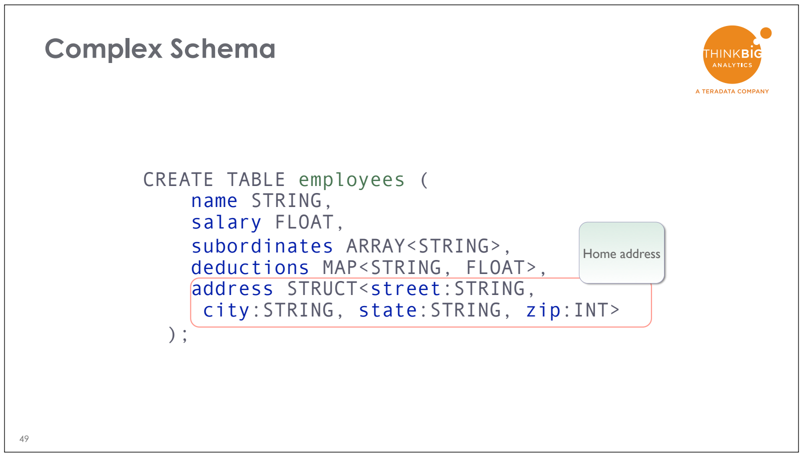
Complex Schema

```
CREATE TABLE employees (
    name STRING,
    salary FLOAT,
    subordinates ARRAY<STRING>,
    deductions MAP<STRING, FLOAT>,
    address STRUCT<street:STRING,
     city:STRING, state:STRING, zip:INT>
  );
```

Paycheck deductions: (name, %)

48

Let’s walk through this...

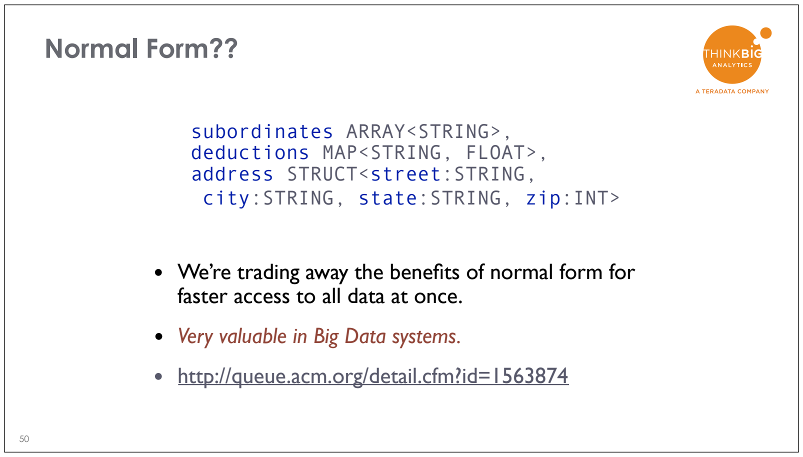


Complex Schema

```
CREATE TABLE employees (
    name STRING,
    salary FLOAT,
    subordinates ARRAY<STRING>,
    deductions MAP<STRING, FLOAT>,
    address STRUCT<street:STRING,
     city:STRING, state:STRING, zip:INT>
  );
```

Home address

Let’s walk through this...



Normal Form??

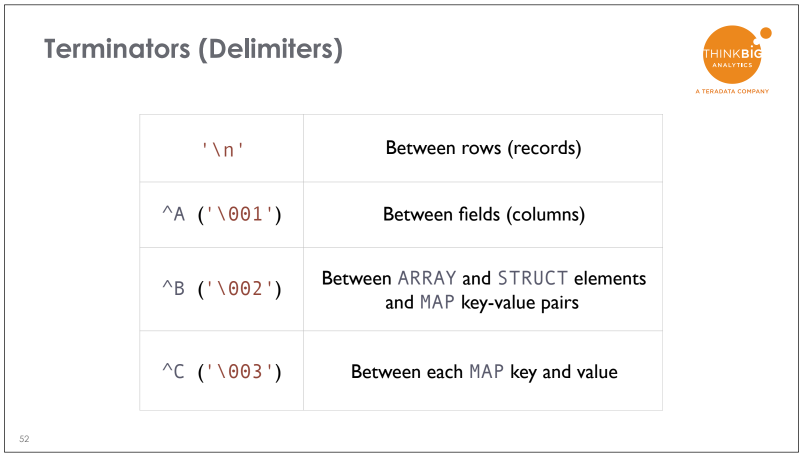
```
subordinates ARRAY<STRING>,
deductions MAP<STRING, FLOAT>,
address STRUCT<street:STRING,
 city:STRING, state:STRING, zip:INT>
```

- We're trading away the benefits of normal form for faster access to all data at once.
- *Very valuable in Big Data systems.*
- http://queue.acm.org/detail.cfm?id=1563874

The link is to an ACM Queue article that explains the performance issues with RDBMSs for big data sets and how techniques like denormalizing data help address the performance issues. RDBMSs don't use complex structures. Instead preferring separate tables and using joins with foreign keys. This is dramatically slower than a straight disk scan, but normal form does optimize space utilization.

Storage Formats

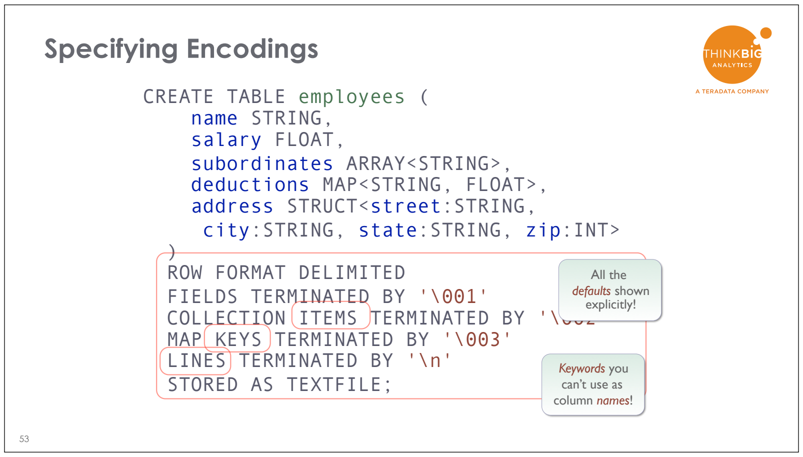
- So far, we've used a *plain-text file format*.
- Let's explore its properties.
- We'll see other formats later.



Terminators (Delimiters)

`'\n'`	Between rows (records)
`^A ('\001')`	Between fields (columns)
`^B ('\002')`	Between `ARRAY` and `STRUCT` elements and `MAP` key-value pairs
`^C ('\003')`	Between each `MAP` key and value

Hive uses the term “terminators” in table definitions, but they are really delimiters or separators between “things”.
“^A” means “control-A”. The corresponding ‘\001 is the “octal code” for how you write the control character in CREATE TABLE statements.



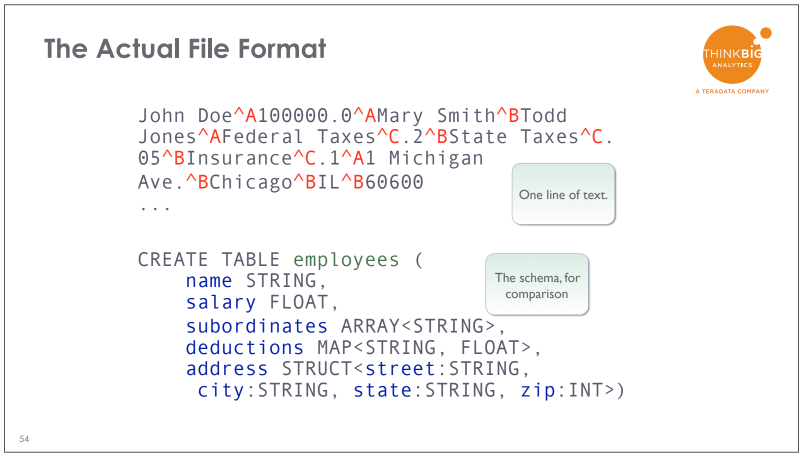
Specifying Encodings

```
CREATE TABLE employees (
    name STRING,
    salary FLOAT,
    subordinates ARRAY<STRING>,
    deductions MAP<STRING, FLOAT>,
    address STRUCT<street:STRING,
     city:STRING, state:STRING, zip:INT>
  )
  ROW FORMAT DELIMITED
  FIELDS TERMINATED BY '\001'
  COLLECTION ITEMS TERMINATED BY '\002'
  MAP KEYS TERMINATED BY '\003'
  LINES TERMINATED BY '\n'
  STORED AS TEXTFILE;
```

All the *defaults* shown explicitly!

Keywords you can't use as column *names*!

Our original employees table, now written to show all the default values used for the terminators and the fact that it's stored as a text file.



The Actual File Format

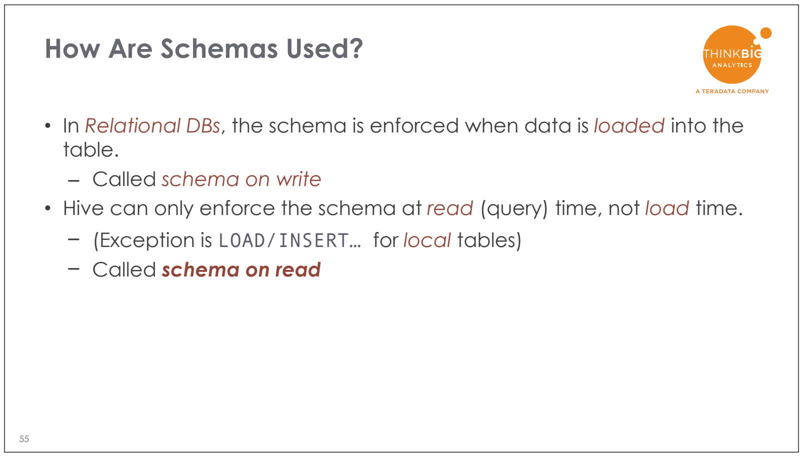
```
John Doe^A100000.0^AMary Smith^BTodd
Jones^AFederal Taxes^C.2^BState Taxes^C.
05^BInsurance^C.1^A1 Michigan
Ave.^BChicago^BIL^B60600
...
```

One line of text.

```
CREATE TABLE employees (
    name STRING,
    salary FLOAT,
    subordinates ARRAY<STRING>,
    deductions MAP<STRING, FLOAT>,
    address STRUCT<street:STRING,
     city:STRING, state:STRING, zip:INT>)
```

The schema, for comparison

This is what’s actually stored in the file. The first line, a single record, of the file we’ll use is shown here, with all the default delimiters.

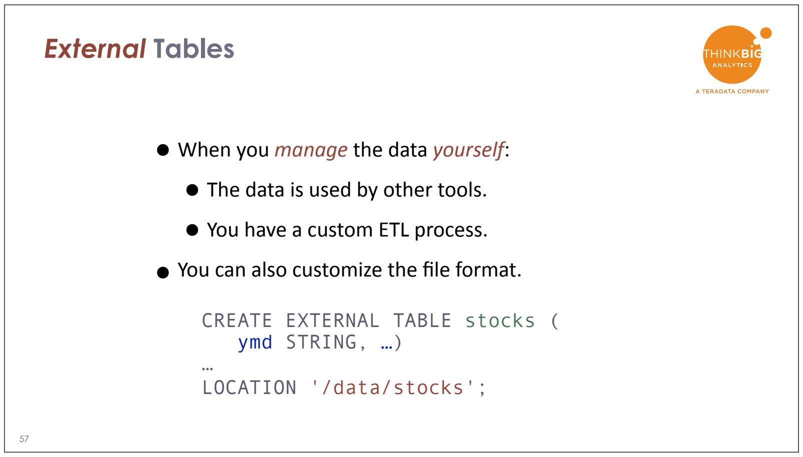


How Are Schemas Used?

- In *Relational DBs*, the schema is enforced when data is *loaded* into the table.
  - Called *schema on write*
- Hive can only enforce the schema at *read* (query) time, not *load* time.
  - (Exception is `LOAD/INSERT`… for *local* tables)
  - Called ***schema on read***

Traditional relational DBs enforce the schema as data is loaded into tables. This makes queries faster and loads slower. Since Hive has less control over the table contents (e.g., “external” tables), it waits to very the schema when you actually query the data. This slows reads a bit, but it also gives you a lot more flexibility to interpret file contents as you see fit, even using different “interpretations” (schema) at different times for the same data.
We’ll explain loading/inserting data into internal tables shortly.

Lab: Hive-Schemas

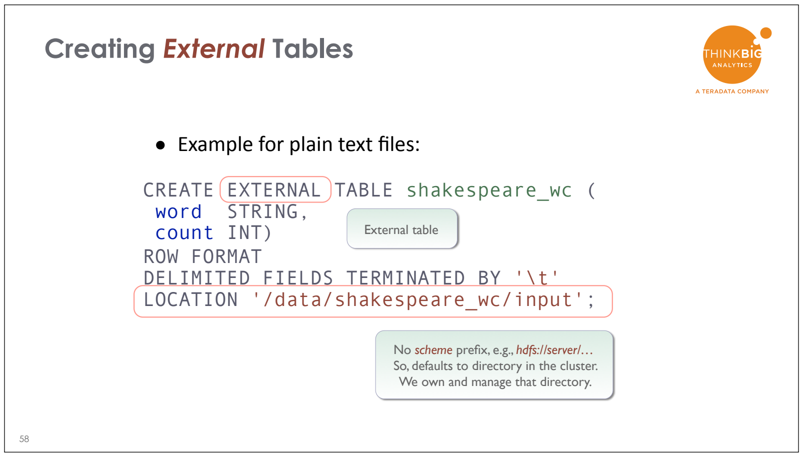


External Tables

- When you *manage* the data *yourself*:
  - The data is used by other tools.
  - You have a custom ETL process.
- You can also customize the file format.

```
CREATE EXTERNAL TABLE stocks (
   ymd STRING, …)
…
LOCATION '/data/stocks';
```

Note: ymd = year-month-day. If you use ‘date’ as the name here, you get a compilation error. However, you can use ‘date’ in other places; see the Hive tutorial on the apache site. The LOCATION will be in the cluster, unless you’re running in Hadoop’s local/ standalone mode.


Creating *External* Tables

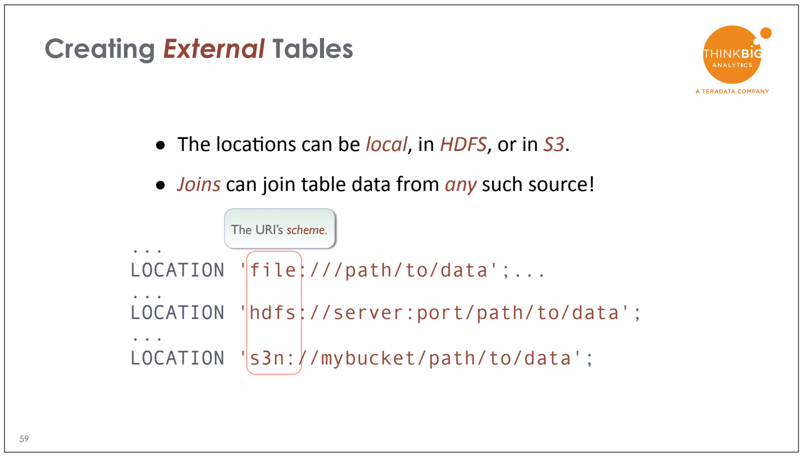
- Example for plain text files:

```
CREATE EXTERNAL TABLE shakespeare_wc (
 word  STRING,
 count INT)
ROW FORMAT
DELIMITED FIELDS TERMINATED BY '\t'
LOCATION '/data/shakespeare_wc/input';
```

External table

No *scheme* prefix, e.g., *hdfs://server/…*
So, defaults to directory in the cluster.
We own and manage that directory.

Recall that previously we defined an identical shakespeare_wc table that was internal and we had to subsequently LOAD the data into the table. If we already have the data in our cluster, why duplicate it?!
Note that LOCATION is a directory. Hive will just read all the files underneath.



Creating *External* Tables

- The locations can be *local*, in *HDFS*, or in *S3*.
- *Joins* can join table data from *any* such source!

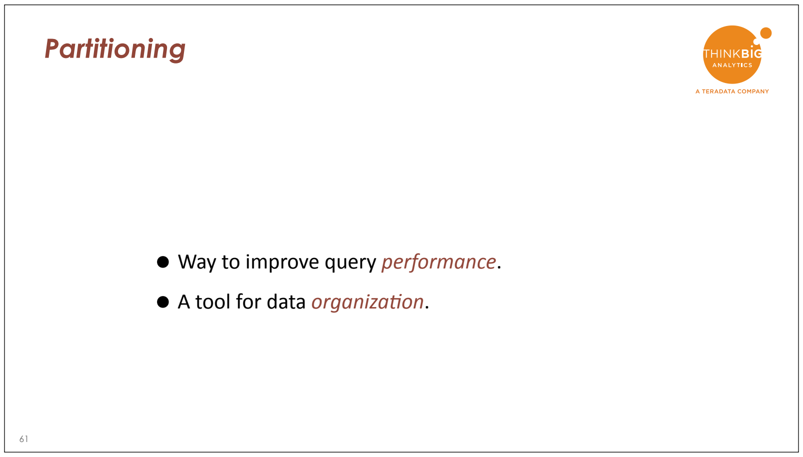
The URI's *scheme*.

```
...
LOCATION 'file:///path/to/data';...
...
LOCATION 'hdfs://server:port/path/to/data';
...
LOCATION 's3n://mybucket/path/to/data';
```

So, you might have a table pointing to “hot” data in HDFS, a table pointing to a local temporary file created by an ETL staging process, and some longer-lived data in S3 and do joins on all of them!

Dropping External Tables

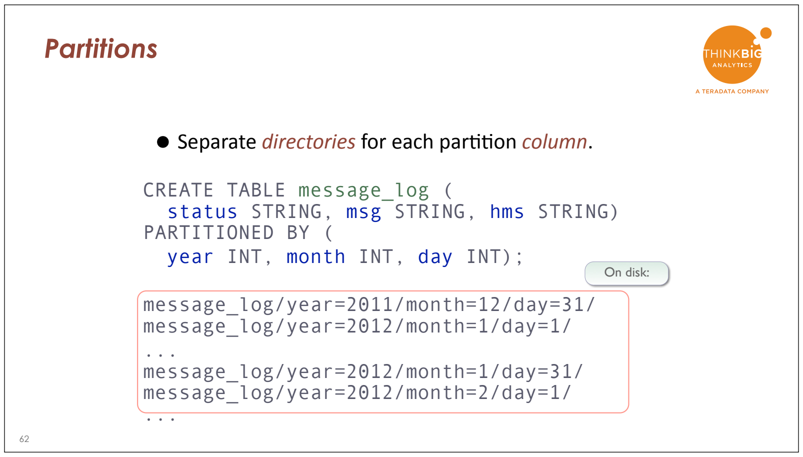
- Because you *manage* the data *yourself*:
  - The table *contents are not deleted* when you drop the table.
  - The table *metadata is deleted* from the *metastore*.



Partitioning

- Way to improve query *performance*.
- A tool for data *organization*.

Partitioning in Hive is similar to partitioning in many DB systems.



Partitions

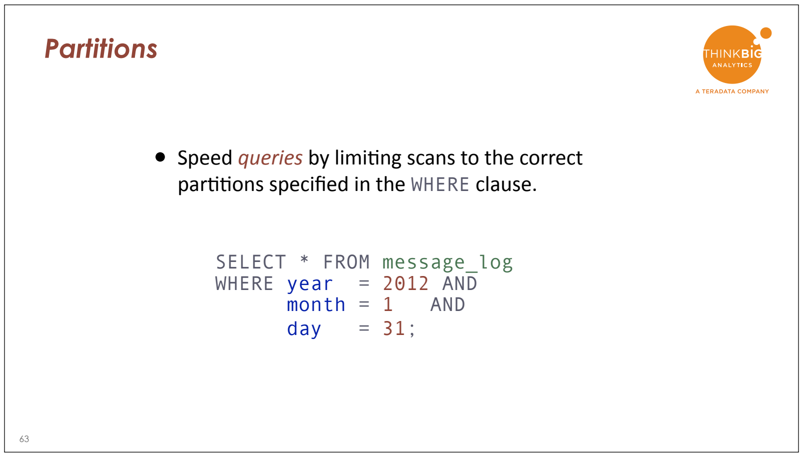
- Separate *directories* for each partition *column*.

```
CREATE TABLE message_log (
  status STRING, msg STRING, hms STRING)
PARTITIONED BY (
  year INT, month INT, day INT);
```

On disk:

```
message_log/year=2011/month=12/day=31/
message_log/year=2012/month=1/day=1/
...
message_log/year=2012/month=1/day=31/
message_log/year=2012/month=2/day=1/
...
```

This is an INTERNAL table and the directory structure shown will be in Hive's warehouse cluster directory. The actual directories, e.g., .../year=2012/month=01/day=01 (yes, that's the naming scheme), will be created when we load the data, discussed in the next module.
(Note that "hms" is the remaining hours-minutes-seconds...)

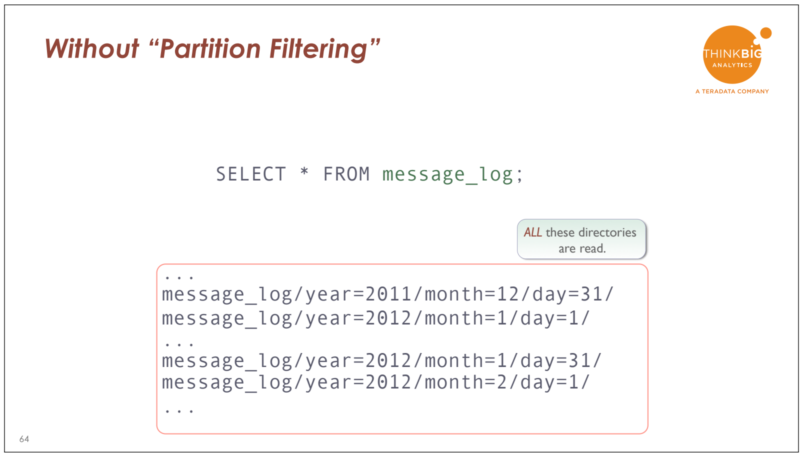


Partitions

- Speed *queries* by limiting scans to the correct partitions specified in the WHERE clause.

```
SELECT * FROM message_log
WHERE year  = 2012 AND
      month = 1    AND
      day   = 31;
```

In SELECT and WHERE clauses, you use the partitions just like ordinary columns, but they significant performance implications.



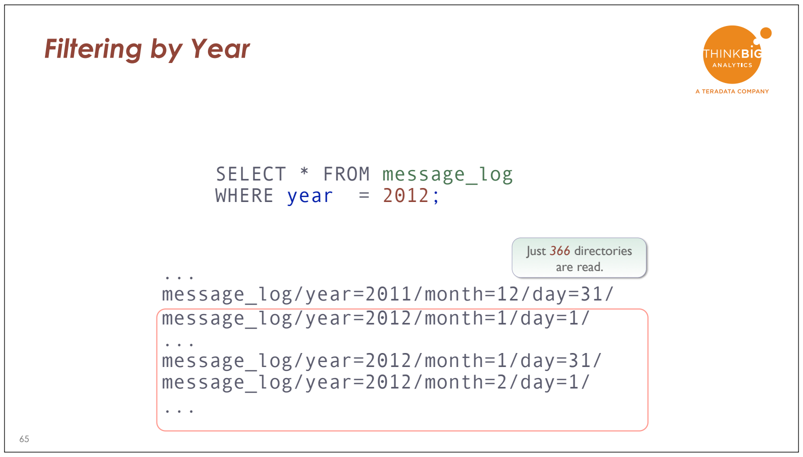
Without “Partition Filtering”

```
SELECT * FROM message_log;
```

ALL these directories are read.

```
...
message_log/year=2011/month=12/day=31/
message_log/year=2012/month=1/day=1/
...
message_log/year=2012/month=1/day=31/
message_log/year=2012/month=2/day=1/
...
```

Without a WHERE clause that limits the result set, Hive has to read the files in EVERY DIRECTORY ever created for “message_log”. Sometimes, that’s what you want, but the point is that often, you’re likely to do queries between time ranges, so scanning all the data is wasteful.



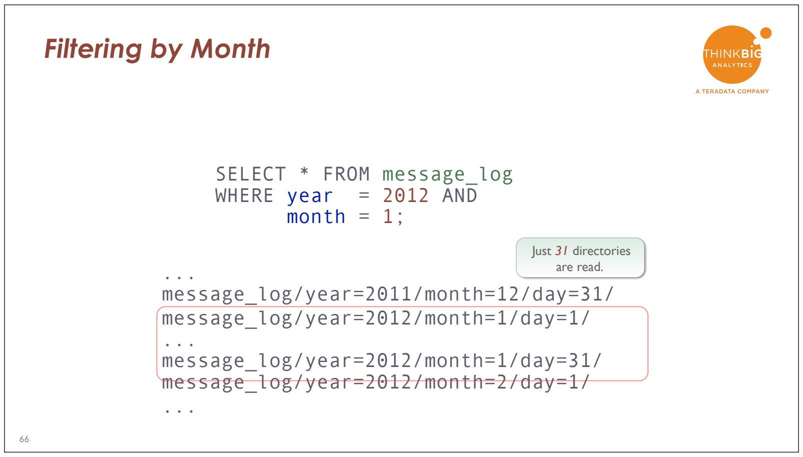
Filtering by Year

```
SELECT * FROM message_log
WHERE year  = 2012;
```

Just *366* directories are read.

```
...
message_log/year=2011/month=12/day=31/
message_log/year=2012/month=1/day=1/
...
message_log/year=2012/month=1/day=31/
message_log/year=2012/month=2/day=1/
...
```

If you filter by year, you have to read only 365 or 366 directories, that is the days under the months which are under the year.



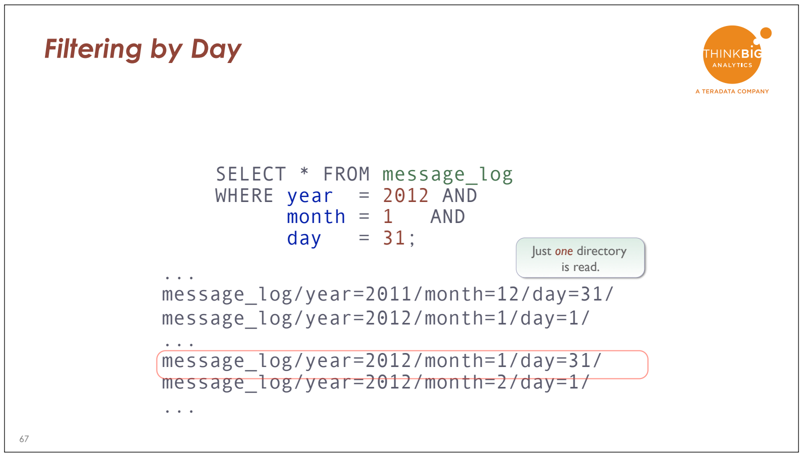
Filtering by Month

```
SELECT * FROM message_log
WHERE year  = 2012 AND
      month = 1;
```

Just *31* directories are read.

```
...
message_log/year=2011/month=12/day=31/
message_log/year=2012/month=1/day=1/
...
message_log/year=2012/month=1/day=31/
message_log/year=2012/month=2/day=1/
...
```

If you filter by month, you need to read only those directories for that month, such as 31 directories for January.



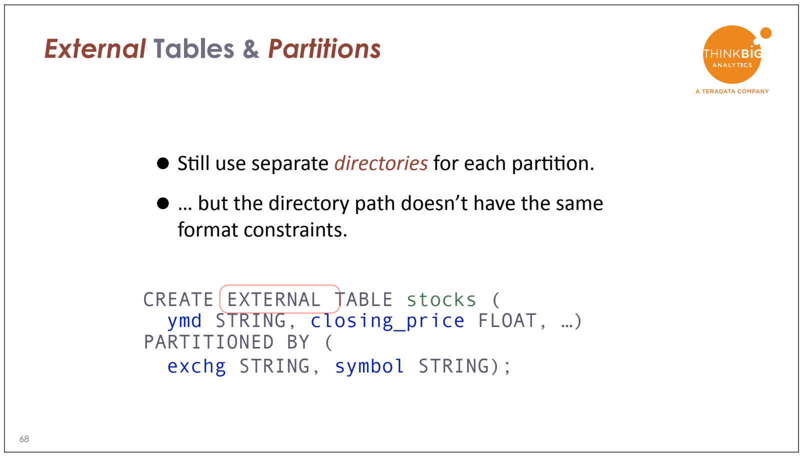
Filtering by Day

```
    SELECT * FROM message_log
    WHERE year  = 2012 AND
          month = 1    AND
          day   = 31;
```

Just *one* directory is read.

```
...
message_log/year=2011/month=12/day=31/
message_log/year=2012/month=1/day=1/
...
message_log/year=2012/month=1/day=31/
message_log/year=2012/month=2/day=1/
...
```

Finally, if you filter by all three, year, month, and day, Hive only has to read one directory!
The point is that partitions drastically reduce the amount of data Hive has to scan through, but it's only useful if you pick a partitioning scheme that represents common WHERE clause filtering, like date ranges in this example.

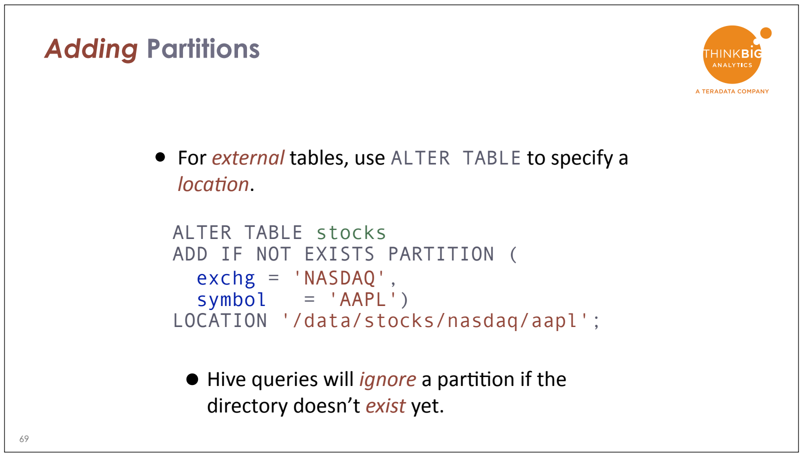


External Tables & *Partitions*

- Still use separate *directories* for each partition.
- … but the directory path doesn't have the same format constraints.

```
CREATE EXTERNAL TABLE stocks (
  ymd STRING, closing_price FLOAT, …)
PARTITIONED BY (
  exchg STRING, symbol STRING);
```

This is an INTERNAL table and the directory structure shown will be in Hive's warehouse cluster directory. The actual directories, e.g., …/year=2012/month=01/day=01, will be created when we load the data, discussed in the next module.



Adding Partitions

- For *external* tables, use `ALTER TABLE` to specify a *location*.

```
ALTER TABLE stocks
ADD IF NOT EXISTS PARTITION (
  exchg = 'NASDAQ',
  symbol   = 'AAPL')
LOCATION '/data/stocks/nasdaq/aapl';
```

  - Hive queries will *ignore* a partition if the directory doesn't *exist* yet.

We'll see more about ALTER TABLE later.
Technically, we don't have to nest "AAPL" under "NASDAQ", but it's generally a good idea to reflect the data hierarchy this way. (It might improve performance for table scans over multiple "symbol" values, including all of them.)
Ignoring missing partitions is convenient. If you have no data, there's no need for empty directories and/or files. Also, if it's easier, you can create a lot of "future" partitions before you actually have data for them (like a month's worth of message_log partitions).

Lab: Hive-Tables2

exercises/Hive-Tables2

- Create an *external* and *partitioned* `stocks` table.
- List the *partitions*.
- We'll populate the data in the next exercise.

Loading Data Into Tables

https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DML

Loading Data

- Use LOAD to load data from a *file* or *directory*:

```
LOAD DATA LOCAL INPATH '/logs-20120131'
OVERWRITE INTO TABLE logs
PARTITION (year=2012, month=1, day=31);
```

- The distributed file system is assumed for the path *unless* `LOCAL` used.
- Data is *appended unless* `OVERWRITE` used.
- `PARTITION` is required if the table uses them.

Loading Data

- Use LOAD to load data from a *file* or *directory*:

```
LOAD DATA LOCAL INPATH '/logs-20120131'
OVERWRITE INTO TABLE logs
PARTITION (year=2012, month=1, day=31);
```

- When you use LOCAL, the data is *copied*.
- When you *don't* use LOCAL, the data is *moved*.

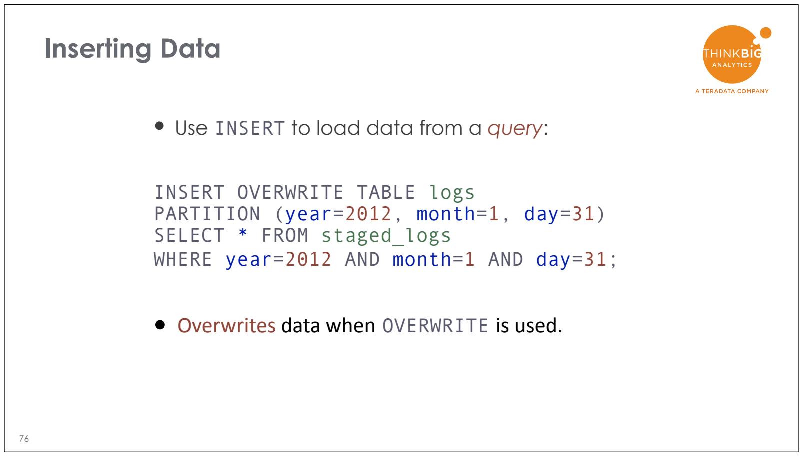
Loading Data

- You can use full URIs:

```
LOAD DATA INPATH
 'hdfs://server/logs-20120131' …;

LOAD DATA LOCAL INPATH
 'file:///logs-20120131' …;
```

- But `s3n://` URIs aren’t supported.


Inserting Data

- Use `INSERT` to load data from a *query*:

```
INSERT OVERWRITE TABLE logs
PARTITION (year=2012, month=1, day=31)
SELECT * FROM staged_logs
WHERE year=2012 AND month=1 AND day=31;
```

- Overwrites data when `OVERWRITE` is used.

76

Imagine we've staged log data into one table and now we're using this scheme to put into the final, partitioned table.

Inserting Data

- Use INSERT to load data from a *query*:

```
INSERT INTO TABLE logs
PARTITION (year=2012, month=1, day=31)
SELECT * FROM staged_logs
WHERE year=2012 AND month=1 AND day=31;
```

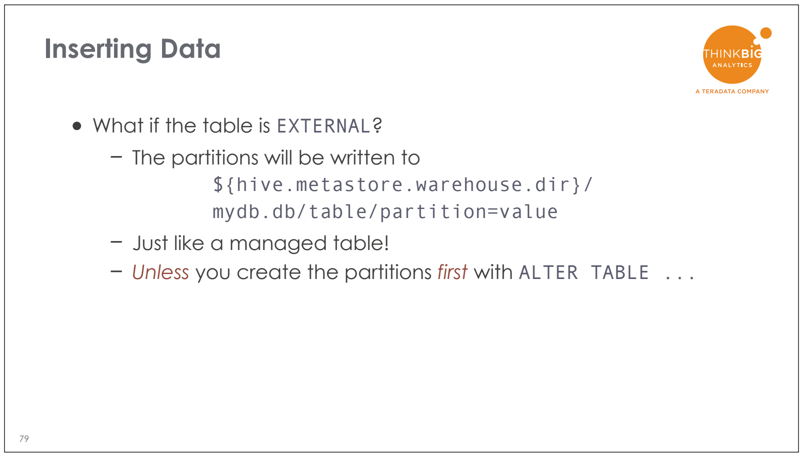
- Appends data if INTO is used (instead of OVERWRITE)

Inserting Data

- Use `INSERT` to load data from a *query*:

```
INSERT INTO TABLE logs
PARTITION (year=2012, month=1, day=31)
SELECT * FROM staged_logs
WHERE year=2012 AND month=1 AND day=31;
```

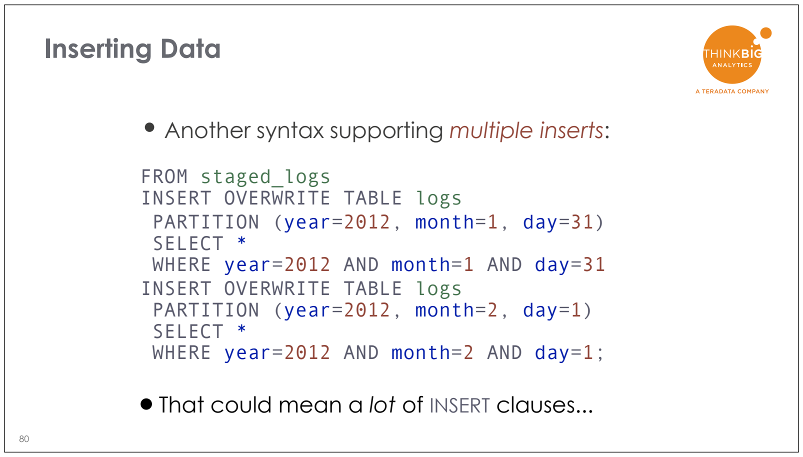
- Requires `PARTITION` if the table is partitioned.



Inserting Data

- What if the table is `EXTERNAL`?
  - The partitions will be written to
    `${hive.metastore.warehouse.dir}/`
    `mydb.db/table/partition=value`
  - Just like a managed table!
  - *Unless* you create the partitions *first* with `ALTER TABLE ...`

So, a partitioning query can be used to populate an external table, but if you don't want the default warehouse location used for the data, you have to create the partitions in advance! We'll come back to this with an example shortly.



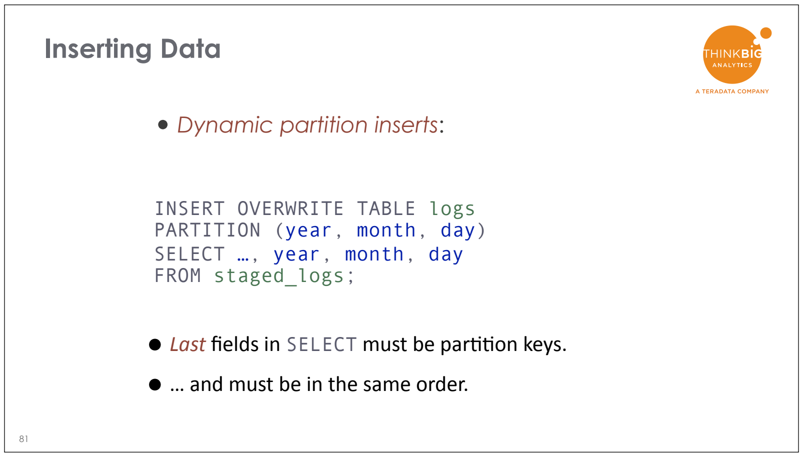
Inserting Data

- Another syntax supporting *multiple inserts*:

```
FROM staged_logs
INSERT OVERWRITE TABLE logs
 PARTITION (year=2012, month=1, day=31)
 SELECT *
 WHERE year=2012 AND month=1 AND day=31
INSERT OVERWRITE TABLE logs
 PARTITION (year=2012, month=2, day=1)
 SELECT *
 WHERE year=2012 AND month=2 AND day=1;
```

- That could mean a *lot* of INSERT clauses...

Very useful technique for starting with one table and extracting data through SELECTS that get written to many tables or partitions. It also scans “staged_orders” ONCE, which is a performance improvement if this table is huge!



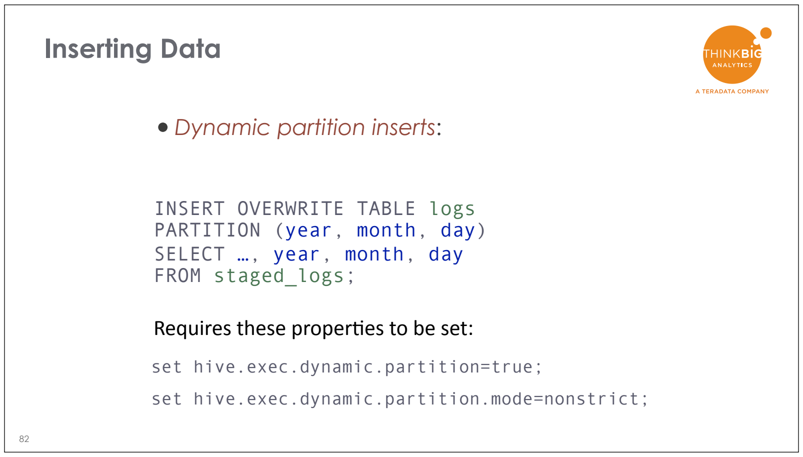
Inserting Data

- *Dynamic partition inserts*:

```
INSERT OVERWRITE TABLE logs
PARTITION (year, month, day)
SELECT …, year, month, day
FROM staged_logs;
```

- *Last* fields in `SELECT` must be partition keys.
- … and must be in the same order.

The data is partitioned “dynamically” based on the values for the select fields. The partitions are created by Hive automatically, based on the partition key values.



Inserting Data

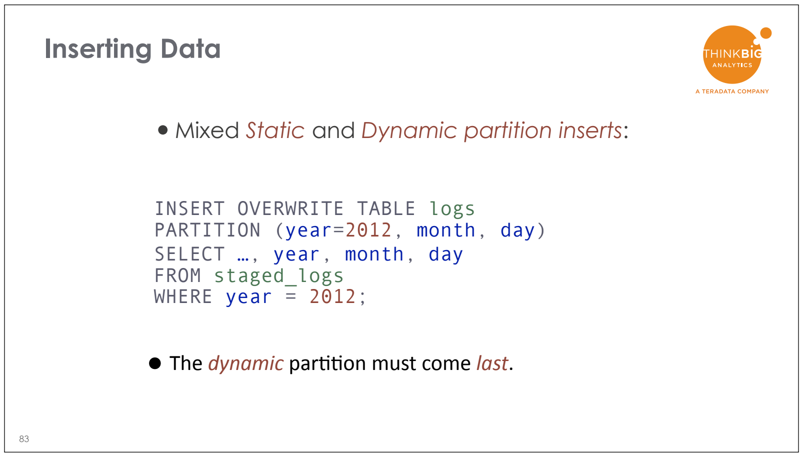
- *Dynamic partition inserts*:

```
INSERT OVERWRITE TABLE logs
PARTITION (year, month, day)
SELECT …, year, month, day
FROM staged_logs;
```

Requires these properties to be set:

```
set hive.exec.dynamic.partition=true;
set hive.exec.dynamic.partition.mode=nonstrict;
```

You have to enable these properties first.
Note: There are additional properties to fine tune the allowed number of partitions, resource utilization in data nodes, etc. See the Tuning chapter of Programming Hive for more details.



Inserting Data

- Mixed *Static* and *Dynamic partition inserts*:

```
INSERT OVERWRITE TABLE logs
PARTITION (year=2012, month, day)
SELECT …, year, month, day
FROM staged_logs
WHERE year = 2012;
```

- The *dynamic* partition must come *last*.

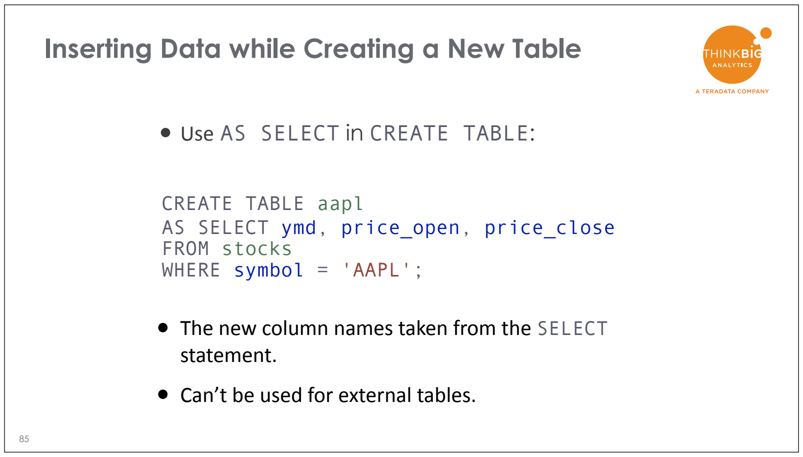
83

You can mix static partitions, where you specify a value, and dynamic. The static ones must come before the dynamic partitions.

Inserting Data

- *Create* the partitions *first* for an *external* table.

```
CREATE EXTERNAL TABLE logs (…)
ALTER TABLE logs
ADD PARTITION (year=2012, month=1, day=31);
...
INSERT INTO TABLE logs
PARTITION (year, month, day)
SELECT …, year, month, day
FROM staged_logs;
```



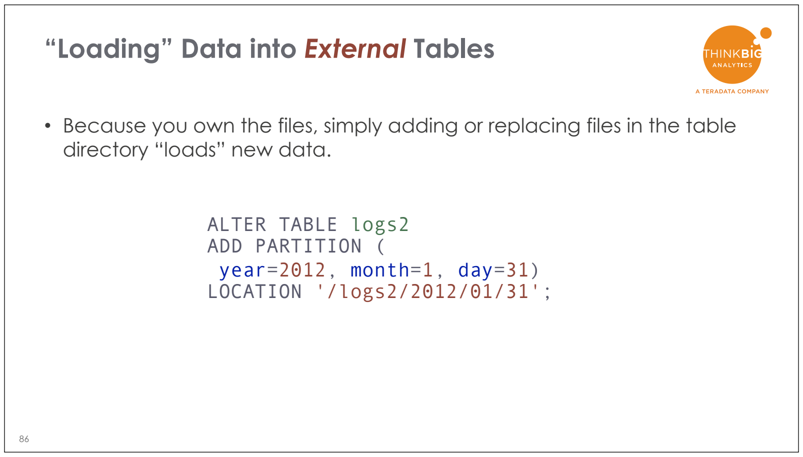
Inserting Data while Creating a New Table

- Use AS SELECT in CREATE TABLE:

```
CREATE TABLE aapl
AS SELECT ymd, price_open, price_close
FROM stocks
WHERE symbol = 'AAPL';
```

- The new column names taken from the SELECT statement.
- Can't be used for external tables.

Even though you can't assign new column names, you have a few options, 1) you can use ALTER TABLE to change the names afterwards, 2) you can give the name "aliases" within the SELECT clause.

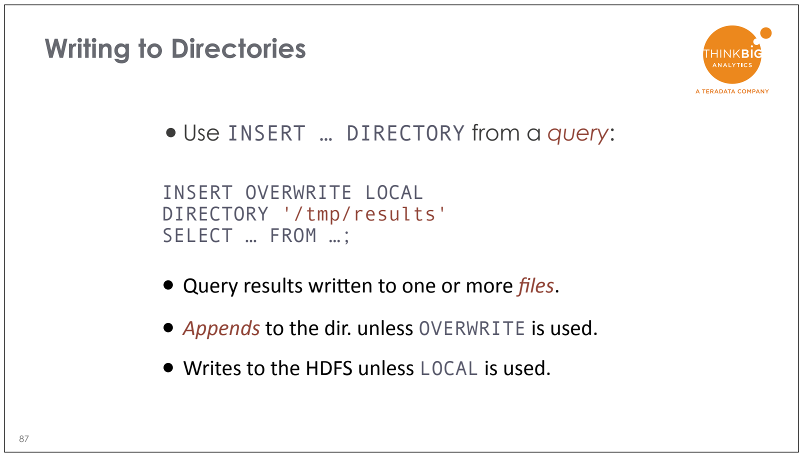


“Loading” Data into *External* Tables

- Because you own the files, simply adding or replacing files in the table directory “loads” new data.

```
ALTER TABLE logs2
ADD PARTITION (
 year=2012, month=1, day=31)
LOCATION '/logs2/2012/01/31';
```

We’ll discuss altering tables in more detail next.
Similarly, if you delete files, the data is deleted.



Writing to Directories

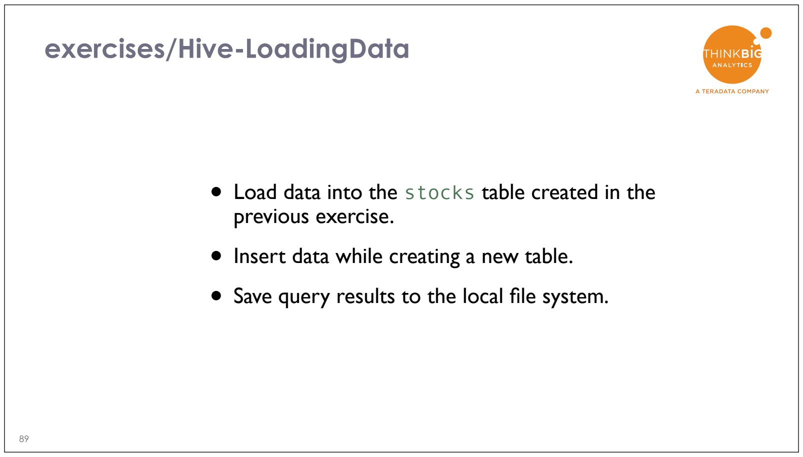
- Use `INSERT … DIRECTORY` from a *query*:

```
INSERT OVERWRITE LOCAL
DIRECTORY '/tmp/results'
SELECT … FROM …;
```

- Query results written to one or more *files*.
- *Appends* to the dir. unless `OVERWRITE` is used.
- Writes to the HDFS unless `LOCAL` is used.

If you want the WHOLE table, you can just copy the file in the cluster, but if you want a query result, this is the technique. It can write to a new cluster directory or to a local filesystem directory.
The Hive documentation for other forms of this statement.

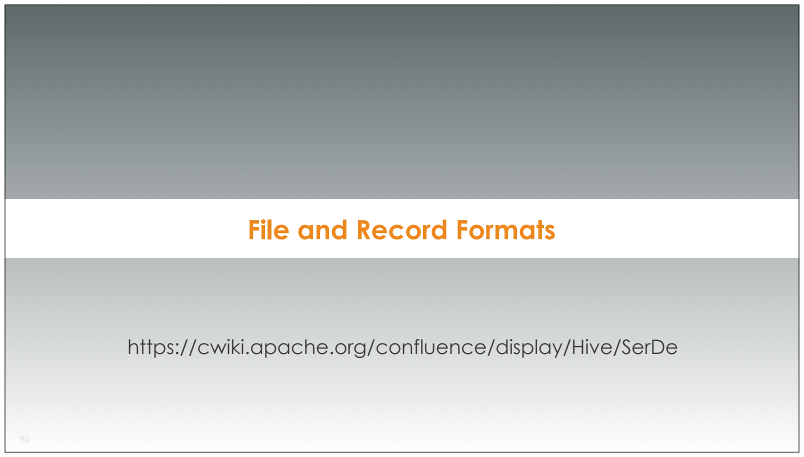
Lab: Hive-LoadingData



exercises/Hive-LoadingData

- Load data into the `stocks` table created in the previous exercise.
- Insert data while creating a new table.
- Save query results to the local file system.

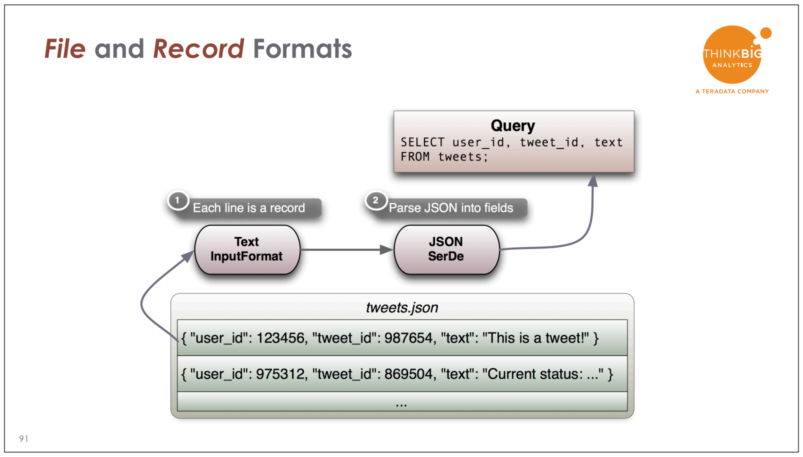
Walk through the code.



File and Record Formats

https://cwiki.apache.org/confluence/display/Hive/SerDe

How to support new file and record formats. We have used the defaults almost exclusively today, but the file and record formats, and whether or not (and how) you compress files have important benefits for disk space and network IO overhead, MR performance, how easy it is to work with files using other tools (either within or outside the Hadoop “ecosystem”), etc. These choices are usually made by the IT team when architecting the whole system and particular data usage scenarios. We discuss these choices in depth in our Developer Course aimed at Java Developers. Also, the Bonus Material section contains an expanded version of this section.

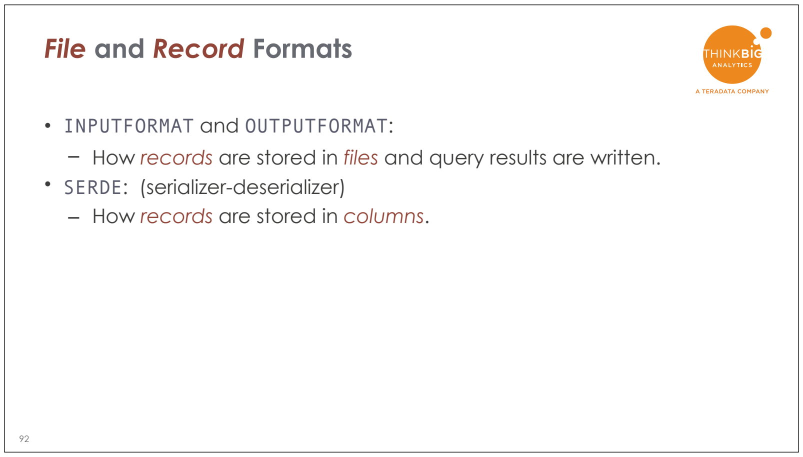


File and *Record* Formats

```
SELECT user_id, tweet_id, text
FROM tweets;
```

```
{ "user_id": 123456, "tweet_id": 987654, "text": "This is a tweet!" }
{ "user_id": 975312, "tweet_id": 869504, "text": "Current status: ..." }
```

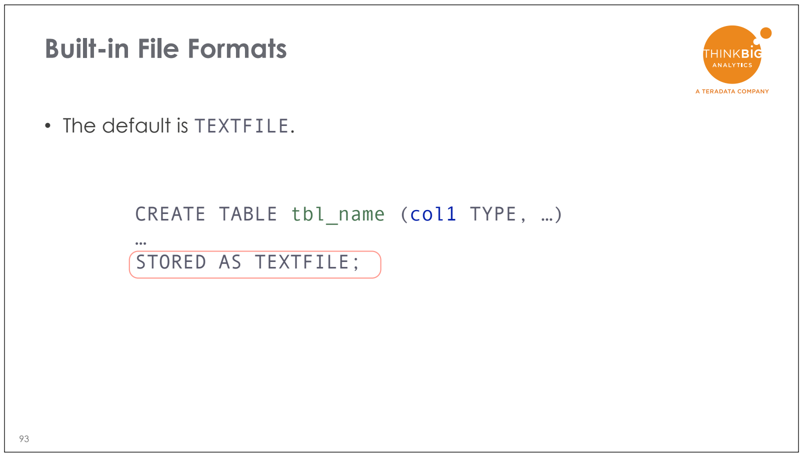
All the InputFormat does is split the file into records. It knows nothing about the format of those records. The SerDe (serializer/deserializer) parses each record into fields/columns.



File and *Record* Formats

- INPUTFORMAT and OUTPUTFORMAT:
  - How *records* are stored in *files* and query results are written.
- SERDE: (serializer-deserializer)
  - How *records* are stored in *columns*.

This is an important distinction; how records are encoded in files and how columns/fields are encoded in records.
INPUTFORMATs are responsible for splitting an input stream into records.
OUTPUTFORMATs are responsible for writing records to an output stream (i.e., query results). Two separate classes are used.
SERDEs are responsible for tokenizing a record into columns/fields and also encoding columns/fields into records. Unlike the *PUTFORMATs, there is one class for both tasks.

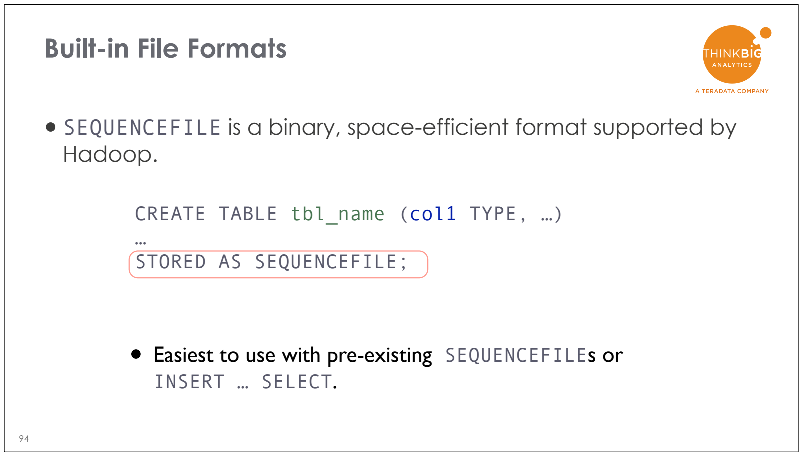


Built-in File Formats

- The default is `TEXTFILE`.

```
CREATE TABLE tbl_name (col1 TYPE, …)
…
STORED AS TEXTFILE;
```

We have been using TEXTFILE, the default, all day.
We'll discuss several of the most common options, but there are many more to choice from. In your projects, the whole development team will want to pick the most appropriate formats that balance the various concerns of disk space and network utilization, sharing with other tools, etc.



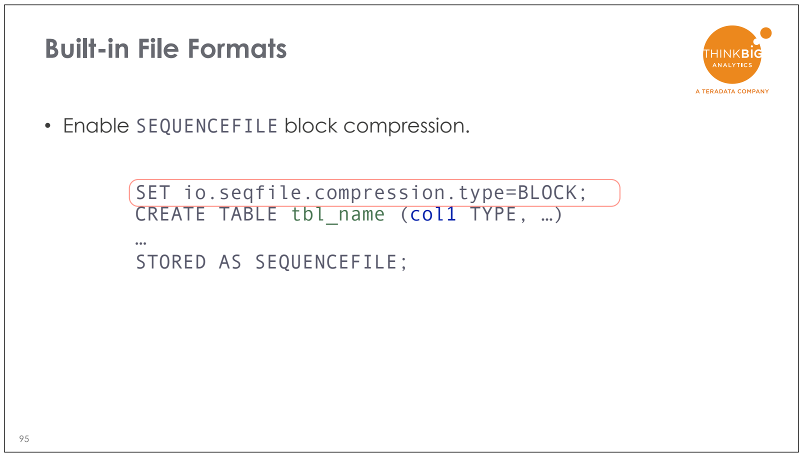
Built-in File Formats

- SEQUENCEFILE is a binary, space-efficient format supported by Hadoop.

```
CREATE TABLE tbl_name (col1 TYPE, …)
…
STORED AS SEQUENCEFILE;
```

- Easiest to use with pre-existing SEQUENCEFILEs or INSERT … SELECT.

SEQUENCEFILE is a Hadoop MapReduce format that uses binary encoding of fields, rather than plain text, so it’s more space efficient, but less convenient for sharing with non-Hadoop tools.

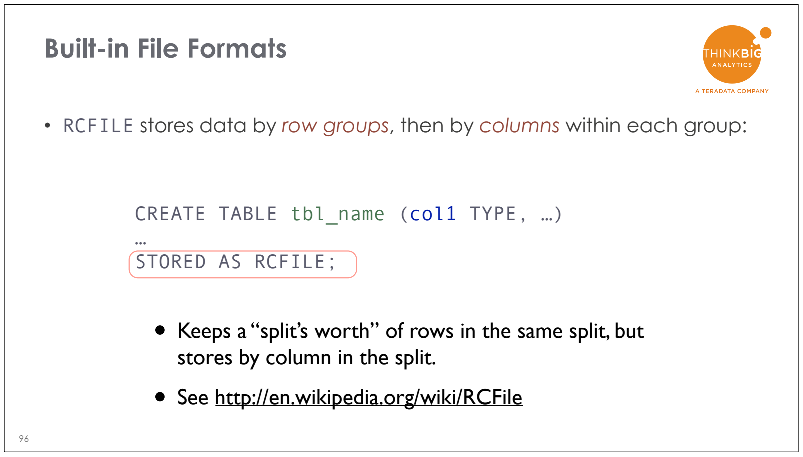


Built-in File Formats

- Enable `SEQUENCEFILE` block compression.

```
SET io.seqfile.compression.type=BLOCK;
CREATE TABLE tbl_name (col1 TYPE, …)
…
STORED AS SEQUENCEFILE;
```

We won't discuss file compression in more detail here. (Our Hadoop training for Java Developers covers this topic in depth.) Compression gives further space and network IO savings. Compressing files by "block" (chunks of rows or bytes), rather than all at once has important practical consequences for MapReduce's ability to split a file into "splits", where each split is sent to a separate Map process. If a file can't be split, then no matter how big it is, it has to be sent to one task, reducing the benefit of a cluster! Block compressed files can be split by MR on block boundaries. Not all compression schemes support block compression. BZip2 does, but GZip does not. SEQUENCEFILEs lend themselves well to block compression.



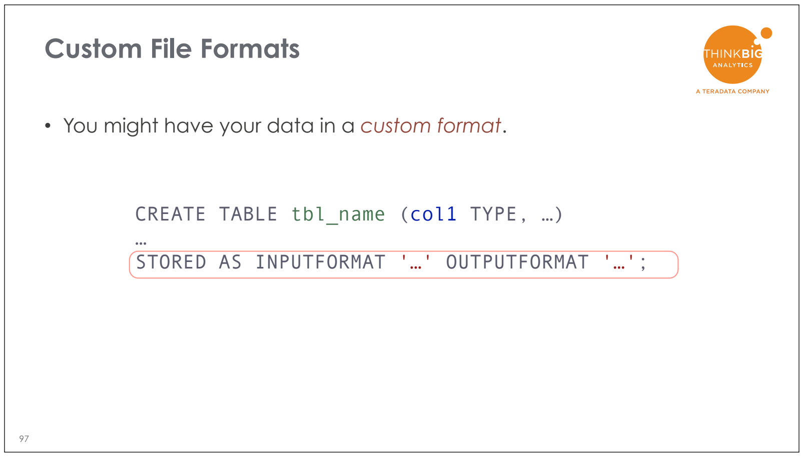
Built-in File Formats

- RCFILE stores data by *row groups*, then by *columns* within each group:

```
CREATE TABLE tbl_name (col1 TYPE, …)
…
STORED AS RCFILE;
```

- Keeps a "split's worth" of rows in the same split, but stores by column in the split.
- See http://en.wikipedia.org/wiki/RCFile

Column-oriented storage is great when you have very long rows and queries typically only need a few columns. Rather than scanning the entire N rows, you just read a smaller amount of data off disk for each column. However, in a distributed system, you might end with rows split across the cluster, so RCFile keeps rows together in split-sized chunks first, but it actually stores the data for those rows in column order, giving you a good compromise of performance tradeoffs.

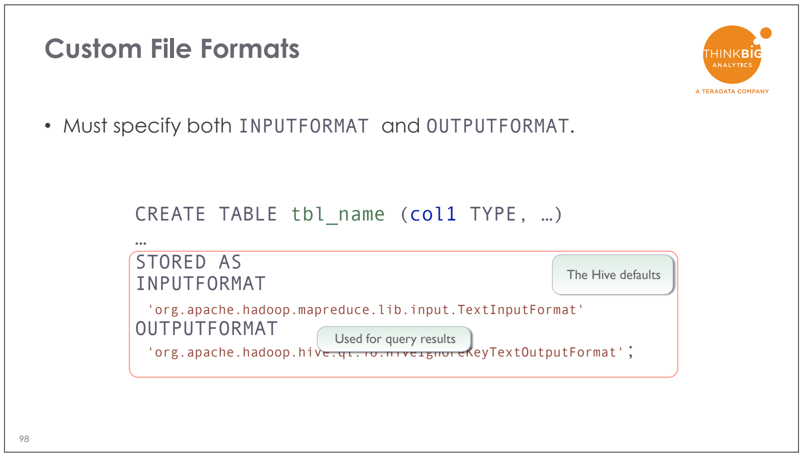


Custom File Formats

- You might have your data in a *custom format*.

```
CREATE TABLE tbl_name (col1 TYPE, …)
…
STORED AS INPUTFORMAT '…' OUTPUTFORMAT '…';
```

You can also use other formats not built into Hive by specifying Java classes that implement them.



Custom File Formats

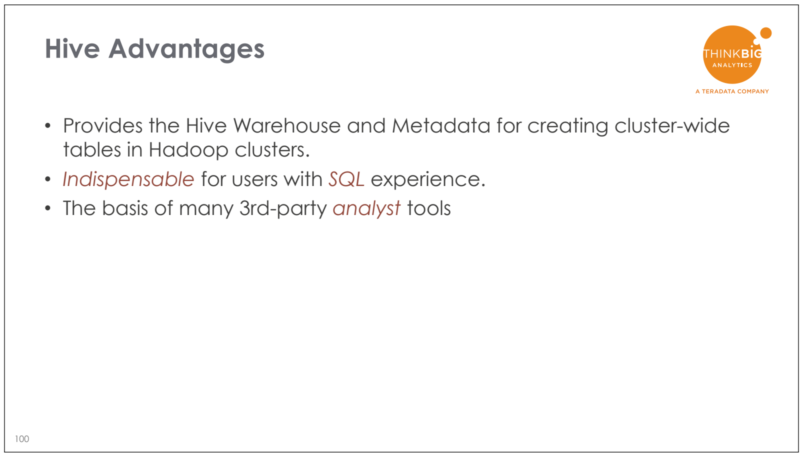
- Must specify both `INPUTFORMAT` and `OUTPUTFORMAT`.

```
CREATE TABLE tbl_name (col1 TYPE, …)
…
STORED AS
INPUTFORMAT
 'org.apache.hadoop.mapreduce.lib.input.TextInputFormat'
OUTPUTFORMAT
 'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat';
```

The Hive defaults
Used for query results

Note that the default INPUTFORMAT is a general Hadoop MapReduce type, while the OUTPUTFORMAT is Hive-specific type.
If you specify INPUTFORMAT, you must also specify OUTPUTFORMAT.

Summary



Hive Advantages

- Provides the Hive Warehouse and Metadata for creating cluster-wide tables in Hadoop clusters.
- *Indispensable* for users with *SQL* experience.
- The basis of many 3rd-party *analyst* tools

I can't emphasize the first point enough. Almost every Hadoop cluster has Hive, even if programmers never use it. Why? Because it's THE repository for traditional SQL tables in a Hadoop cluster. Even if you do all your work in Spark or Pig, you still want Hive for its Warehouse and Metastore.

Hive is truly indispensible for SQL programmers because it provides a familiar interface, and it's the basis of many 3rd party tools.

Hive Advantages

- *Simplifies* many *common* data analysis *tasks*.
  - *Far simpler* than Java MapReduce programming.
  - *Optimizer* for common scenarios.
  - *Extensible* with Java plugins.
- *Integration* with other Hadoop tools.

Hive Disadvantages

- *Not a complete SQL implementation:*
  - Transactions, row updates, etc.
  - … But some will be added over time.
- Hadoop batch mode has *high latency*.
  - But latency amortized over big queries.
- *Documentation* isn't very user friendly

Summary

- If you use Hadoop clusters, you need to know Hive
- Hive allows you to apply familiar SQL tools with raw files of many formats using schema on read
- Hive's ability to create partitions can dramatically speed up queries
- We will frequently use Hive's metadata and metastore from other tools such as Spark