IBM

Smarter education for smarter planet

IBM Innovation Centre for Education

Welcome to:

# Unit 4 – Hadoop Reporting and Analysis

# Unit objectives



After completing this unit, you should be able to:

- Understand the Approaches to Big Data reporting and analysis
- Do Business Intelligence, reporting  using Hadoop Architecture
- Do Direct Batch Reporting on Hadoop
- Explore  Big Data 'R'
- Do Indirect Batch Analysis on Hadoop

# Overview



- Pig is a high-level language
  - Designed for ease of programming
- The code is compiled into a sequence of Map / Reduce programs
  - Tasks are encoded in such a way that permits the system to optimize their execution automatically
- Is extensible
  - Users can create their own functions

# Executing Pig



...pig/bin> **pig -x local**
grunt>

Execute Pig in Interactive local mode

- Runs on a single machine
- All files are in the local file system

...pig/bin> **pig -x**
grunt>

Execute Pig in Interactive MapReduce mode

- Runs in a Hadoop cluster
- Is the default mode

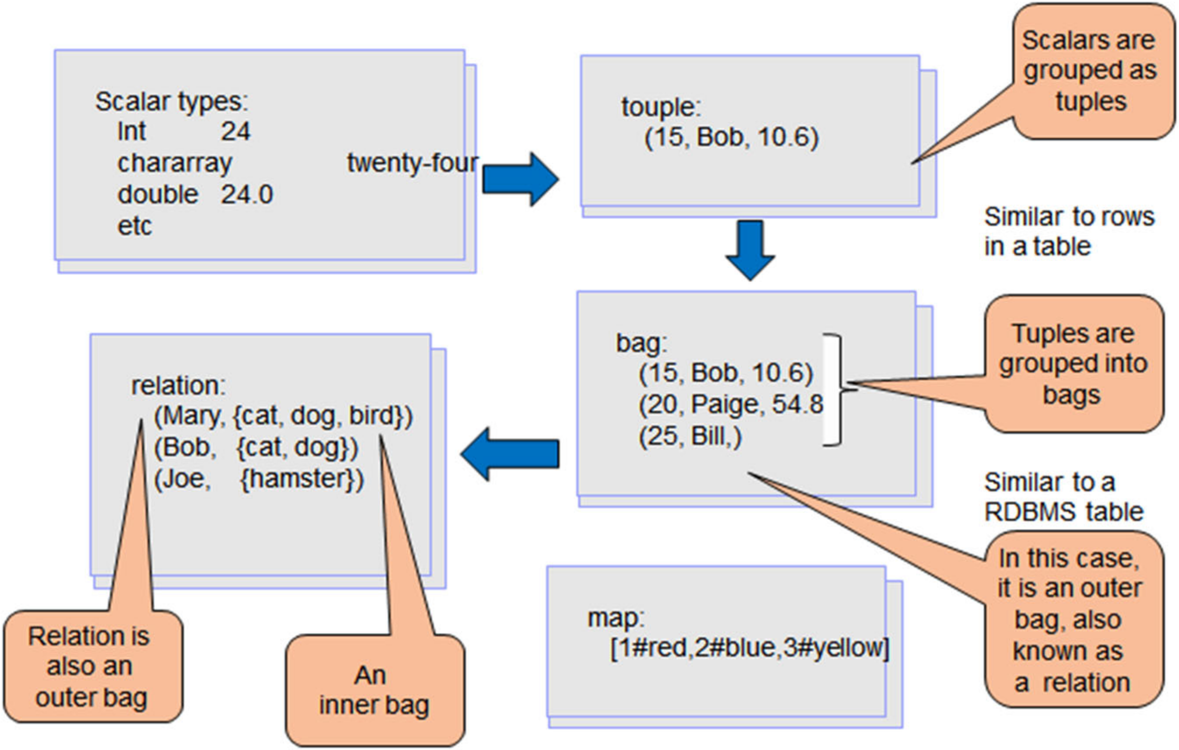
...pig/bin> **pig -x local myscript.pig**
...pig/bin> **pig myscript.pig**

Execute a Pig script

- Script is a text file with Pig commands
- Scripts can be run in local or MapReduce mode

# First look at Pig data





# Pig Latin statement basics



- Most operators take a relation as input and emits a relation
  - The LOAD operator emits a relation
  - The DUMP and STORE operators only take a relation as input
    - Used to generate output

- Statements are terminated with a semicolon

- Comments
  - /* … */ for a block of comments
  - Double dashes (--) indicate that the rest of the line is commented

# Input



- LOAD operator is used to read data
  - Several built-in load functions provide support for different data types
    - BinStorage() – data in a machine readable format
      - Used internally by Pig to store temporary data
    - PigStorage('field_delimiter')
      - Default delimiter is the tab
      - Specified delimiter is enclosed in single quotes
      - It is the default load function
    - TextLoader()
      - Loads unstructured text
      - Resulting tuple contains a single field with one line of input text
    - JsonLoader()
      - Loads data in a standard JSON format

```
A = load '/datadir/datafile' using PigStorage('\t');
A = load '/datadir/datafile';
```

These are equivalent

# LOAD operator continued

- Several built-in functions provide support for different data types
  - You can define your own load function as well

- Format

  data = LOAD '<data>' [using <function()>] [as (<schema>)];

```
A = load '/datadir/datafile' using PigStorage(',') as (f1:int, f2:chararray, f3:float);
```

Without a schema, fields are not named and all fields default to the bytearray data type

# Accessing data

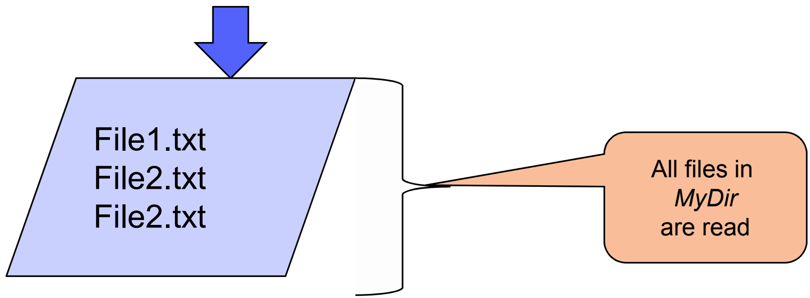


- The LOAD statement can read data from a specific file

```
A = load '/MyDir/File1.txt' using PigStorage(',') as (f1:int, f2:chararray, f3:float);
```

The LOAD statement can read all files is a specific directory

```
A = load '/MyDir' using PigStorage(',') as (f1:int, f2:chararray, f3:float);
```



# Case sensitivity

- The names of relations and fields are case sensitive

```
Data = load 'MyFile.txt' using PigStorage(',') as (f1:int, f2:chararray);
data = load 'MyFile.txt' using PigStorage(',') as (f1:int, F1:chararray);
```

Data and data are two different relations

F1 and f1 are different fields

Function names are case sensitive

```
Data = load 'MyFile.txt' using PigStorage(',') as (f1:int, f2:chararray);   OK
Data= load 'MyFile.txt' using pigstorage(',') as (f1:int, F1:chararray);   Error
```

Keywords are not case sensitive

LOAD, USING, AS, GROUP, BY…

```
Data = load 'MyFile.txt' using PigStorage(',') as (f1:int, f2:chararray);
Data= LOAD 'MyFile.txt' using Pigstorage(',') as (f1:int, F1:chararray);
```

Both OK

# Field reference



- Fields may be referenced by position
  - Assume a schema of
    (name: chararray, age: int, salary: float)
  - The field *age* may be referenced by position *$1*

```
a = load 'books.csv' using PigStorage(',') as (bknum:int, title:chararray);
b = load 'reviews.csv' using PigStorage(',') as (bknum:int, reviewer:chararray);

c = join a on bknum, b on bknum;
d = join a on $0, b on $0;
```

Both joins do the same thing

If a schema is not specify, then fields can only be referenced by position

# Pig data types



| Data type | Description | Example |
| --- | --- | --- |
| int | signed 32-bit integer | 100 |
| long | signed 64-bit integer | 100L or 100l |
| float | 32-bit floating point | 1.5F, 1.5f, 1.5e2f, 1.5E2F |
| double | 64-bit floating point | 1.5, 1.5e2, 1.5E2 |
| chararray | a string | abcde |
| bytearray | a blob | |
| tuple | ordered set of fields | |
| bag | collection of tuples | |
| map | set of key value pairs | |

# Operators



Basic operators

**Arithmetic**

```
+ : a + b, "a" + "b"
- : a - b, -a
/ : a / b
* : a * b
% - modulo
? - binary condition
```

**Boolean**

```
and    : a and b
or     : a or b
not    : a and not b,
         not (a and b)
```

**Comparison**

```
== : a == b
!= : a != b
<  : a < b    <=
: a <= b
>  : a > b    >=
: a >= b
is null
is not null
```

# Parameter substitution



- Parameters can be passed into a Pig script
  - This allows for coding flexibility
    - For example, the same script can be used to process multiple filesNo change to the script is required
- Parameters are referenced in the form of $<identifier>

Assume this statement in
a pig script called *myscript.pig*

```
a = load '$dir/myfile.txt' as PigStorage('\t');
```

Assume this is entered on the
command line

```
$ > ./pig -param dir='/labfiles' myscript.pig
```

What gets executed

```
a = load '/labfiles/myfile.txt' as PigStorage('\t');
```

Assume this statement in
a pig script called *myscript.pig*

```
a = load '$dir/myfile.txt' as PigStorage('\t');
```

Assume this is in the file *myparams.txt*

```
dir = '/labfiles'
```

Assume this is entered on the
command line

```
$ > ./pig -param_file myparams.txt myscript.pig
```

What gets executed

```
a = load '/labfiles/myfile.txt' as PigStorage('\t');
```

# Output



- The *STORE* operator saves results to the file system
  - This operator also uses built-in functions, similar to the *load* operator to handle the type of data
    - PigStorage()
      - The default function
    - BinStorage()
    - PigDump()
    - JsonStorage()
  - Format

    STORE alias into '<directory>' [using function()];

- The *DUMP* operator write the results to the console

  DUMP alias;

# Hive Data Units



- Organization of Hive data (order of granularity)

  Database
  - ->Table
  - ->Partition
  - ->Buckets

  – **Databases**: Namespaces that separate tables and other data units from naming confliction.

  – **Tables**: Homogeneous units of data which have the same schema.

  – **Partitions**: A virtual column which defines how data is stored on the file system based on its values. Each table can have one or more partitions (and one or more levels of partition).

  – **Buckets** (or **Clusters**): In each partition, data can be divided into buckets based on the hash value of a column in the table (useful for sampling, join optimization).

  – Note that it is not necessary for tables to be partitioned or bucketed, but these abstractions allow the system to prune large quantities of data during query processing, resulting in faster query execution.

# Physical Layout – Data in Hive



Data files are just regular HDFS files

- Variety of storage and record formats can be used
- Internal format can vary table-to-table (delimited, sequence, etc.)

- Warehouse directory in HDFS
  - Specified by "*hive.metastore.warehouse.dir*" in hive-site.xml (can be overridden)
  - E.g. /user/hive/warehouse
- One can think tables, partitions and buckets as directories, subdirectories and files respectively

| Hive Entity | Sample | Sample location<br>*In this example $WH is a variable that holds warehouse path* |
|---|---|---|
| database | testdb | $WH/testdb.db |
| table | T | $WH/testdb.db/T |
| partition | date='01012013' | $WH/testdb.db/T/date=01012013 |

We could also Bucket…

| | | |
|---|---|---|
| Bucketing column | userid | $WH/testdb.db/T/date=01012013/000000_0<br>… …<br>$WH/testdb.db/T/date=01012013/000032_0 |

# Creating Databases



- Create a database named “mydatabase”

```
hive> CREATE DATABASE mydatabase;
```

- Create a database named “mydatabase” and override the Hive warehouse configured location

```
hive> CREATE DATABASE mydatabase
    > LOCATION '/myfavorite/folder/;
```

- Create a database and add a descriptive comment

```
hive> CREATE DATABASE mydatabase
    > COMMENT 'This is my database';
```

- Create a database and some descriptive properties

```
hive> CREATE DATABASE mydatabase
    > WITH DBPROPERTIES ('createdby' = 'bigdatauser', 'date' =
'2014-01-01');
```

# Database Show/Describe



- List the Database(s) in the Hive system

```
hive> SHOW DATABASES;
```

- Show some basic information about a Database

```
hive> DESCRIBE DATABASE mydatabase;
```

- Show more detailed information about a Database

```
hive> DESCRIBE DATABASE EXTENDED mydatabase;
```

# Database Use/Drop/Alter



- Hive "default" database is used if database is not specified

- Tell Hive that your following statements will use a specific database

```
hive> USE mydatabase;
```

- Delete a database

```
hive> DROP DATABASE IF EXISTS mydatabase;
```

  – Note: Database directory is also deleted

- Alter a database

```
hive> ALTER DATABASE mydatabase SET DBPROPERTIES ('createdby' =
'bigdatauser2');
```

  – Note: Only possible to update the DBPROPRTIES metadata

# Primitive Data Types

- Types are associated with the columns in the tables.
- Primitive types in Hive (subset of typical RDBMS types):
  - **Integers**
    - TINYINT - 1 byte integer
    - SMALLINT - 2 byte integer
    - INT - 4 byte integer
    - BIGINT - 8 byte integer
  - **Boolean type**
    - BOOLEAN - TRUE/FALSE
  - **Floating point numbers**
    - FLOAT - single precision
    - DOUBLE - Double precision
    - DECIMAL - provide precise values and greater range than Double
  - **String types**
    - STRING - sequence of characters in a specified character set
    - VARCHAR – specify length of character string (between 1 and 65,355)
  - **Date/Time types**
    - TIMESTAMP - YYYY-MM-DD HH:MM:SS.fffffffff
    - DATE - YYYYMMDD
  - **Binary type**
    - Binary - array of bytes (similar to VARBINARY in RDBMS)

  - On comparison, Hive does some implicit casting
    - Any integer to larger of two integer types

# Complex Data Types



- Complex Types can be built up from primitive types and other composite types.
- **Arrays** - Indexable lists containing elements of the same type.
  - Format: ARRAY<data_type>
  - Literal syntax example: array('user1', 'user2')
  - Accessing Elements: [n] notation where n is an index into the array. E.g. *arrayname[0]*
- **Structs** - Collection of elements of different types.
  - Format: STRUCT<col_name : data_type, ...>
  - Literal syntax example: struct('Jake', '213')
  - Accessing Elements: DOT (.) notation. E.g. *structname.firstname*
- **Maps** – Collection of key-value tuples.
  - Format: MAP<primitive_type, data_type>
  - Literal syntax example: map('business_title', 'CEO', 'rank', '1')
  - Accessing Elements: ['element name'] notation. E.g. *mapname['business_title']*
- **Union** – at any one point can hold exactly one of their specified data types.
  - Format: UNIONTYPE<data_type, data_type, ...>
  - Accessing Elements: Use the create_union UDF (see Hive docs for more info)

# Creating a Table

IBM ICE (Innovation Centre for Education)

- Creating a delimited table

```
hive> CREATE TABLE users
(
  id        INT,
  office_id INT,
  name      STRING,
  children  ARRAY<STRING>
)
ROW FORMAT DELIMITED
   FIELDS TERMINATED BY '|'
   COLLECTION ITEMS TERMINATED BY ':'
STORED AS TEXTFILE;
```

**file: users.dat**

```
1|1|Bob Smith|Mary
2|1|Frank Barney|James:Liz:Karen
3|2|Ellen Lacy|Randy:Martin
4|3|Jake Gray|
5|4|Sally Fields|John:Fred:Sue:Hank:Robert
```

- Inspecting tables:

```
hive> SHOW TABLES;
OK
users
Time taken: 2.542 seconds
hive> DESCRIBE users;
OK
id          int
office_id   int
name        string
children    array<string>
Time taken: 0.129 seconds
```

Note: Can also use "DESCRIBE EXTENDED *tablename*" for more metadata

# Table partitioning



- Creating a partitioned table:

```
hive> CREATE TABLE users
(
  id         INT,
  office_id INT,
  name       STRING,
  children   ARRAY<STRING>
)
PARTITIONED BY (division STRING)
ROW FORMAT DELIMITED
   FIELDS TERMINATED BY '|'
   COLLECTION ITEMS TERMINATED BY ':'
STORED AS TEXTIFLE;
```

- Table partitioned on “Division”.
- PARTITION BY clause defines the virtual columns which are different from the data columns and are actually not stored with the data
- Our Hive queries can now take advantage of the partitioned data (each partition is a separate directory that stores the data for that partition)
  - Better performance for certain queries (WHERE clauses)

# Managed Vs External Tables



## Managed Tables

- By default Hive tables are “Managed”.
  - Hive controls the metadata AND the lifecycle of the data
  - Data stored in subdirectories within the hive.metastore.warehouse.dir location
  - Dropping a managed table deletes the data in the table

## External Tables

- Store table in a directory outside of Hive
- Useful if sharing your data with other tools
- Hive does not assume it owns the data in the table
  - Dropping table deletes the tables metadata - NOT the actual data
- Must add the EXTERNAL and LOCATION keywords to CREATE statement

```
CREATE EXTERNAL TABLE users
            …
            …
LOCATION '/path/to/your/data';
```

# Drop/Alter Table



- Delete a table

```
hive> DROP TABLE IF EXISTS users;
```

- Change name of table

```
hive> ALTER TABLE users RENAME TO employees;
```

- Add two columns to end of table

```
hive> ALTER TABLE users ADD COLUMNS (
      location  STRING,
      age        INT);
```

- Variety of other Alter statements
  - Delete columns
  - Alter table properties
  - Alter storage properties
  - Alter partitions

# Indexes



- Goal of Hive indexing:  improve speed of query lookup on certain columns of a table.

- Speed improved at cost of processing and disk space (index data stored in another table)

- **Create an Index**

```
hive> CREATE INDEX table01_index ON TABLE table01 (column2) AS
   'COMPACT';
```

- **Show index**

```
hive> SHOW INDEX ON table01;
```

- **Delete an index**

```
hive> DROP INDEX table01_index ON table01;
```

- Variety of other Indexing topics including Bitmap indexes

# Loading Data into Hive – From a File



- **Loading data from input file (Schema on Read)**

```
hive> LOAD DATA LOCAL INPATH '/tmp/data/users.dat'
         OVERWRITE INTO TABLE users;
```

  - The "LOCAL" indicates the source data is on the local filesystem
  - Local data is **copied** to final location
  - Otherwise file is assumed to be on HDFS and is **moved** to final location
  - Hive does not do any transformation while loading data into tables

- **Loading data into a partition requires PARTITION clause**

```
hive> LOAD DATA LOCAL INPATH '/tmp/data/usersny.dat'
         OVERWRITE INTO TABLE users;
         PARTITION (country = 'US', state = 'NY')
```

  - HDFS directory is created: /user/hive/warehouse/mydb.db/users/country=US/state=NY
    - usersny.dat file is copied to this HDFS directory

# Loading Data from a Directory



- **Load data from an HDFS directory instead of single file**

```
hive> LOAD DATA INPATH '/user/biadmin/userdatafiles'
         OVERWRITE INTO TABLE users;
```

  – Lack of "LOCAL" keyword means source data is on the HDFS file system
  – Data is **moved** to final location
  – All of the files in the /user/biadmin/userdatafiles directory are copied over into Hive
  – OVERWRITE keyword causes contents of target table to be deleted and replaced.
    - Leaving out the OVERWRITE means files will be added to the table
    - If target has file name collision then new file will overwrite existing Hive file

# Loading Data into Hive – From a Query



- Tables may be created using queries on other tables

```
CREATE TABLE emps_by_state
AS
SELECT o.state AS state, count(*) AS employees
  FROM office o LEFT OUTER JOIN users  u
     ON u.office_id = o.office_id
GROUP BY o.state;
```

Or using INSERT OVERWRITE …

```
CREATE table emps_by_state ... STORED AS textfile;

INSERT OVERWRITE TABLE emps_by_state
SELECT o.state AS state, count(*) AS employees
  FROM office o LEFT OUTER JOIN users u
     ON u.office_id = o.office_id
GROUP BY o.state;
```

# Exporting Data out of Hive



- If data files are already format you like, can just copy them out of HDFS
- Query results can be inserted into file system directories (local or HDFS)
- if LOCAL keyword is used, Hive will write data to the directory on the local file system.

```
INSERT OVERWRITE LOCAL DIRECTORY '/mydirectory/dataexports'
SELECT sale_id, product, date
FROM sales
WHERE date='2014-01-01';
```

- Data written to the file system is by default serialized as text with columns separated by ^A and rows separated by newlines.
  – Non-primitive columns serialized to JSON format.
  – Delimiters and file format may be specified.

- INSERT OVERWRITE statements to HDFS is good way to extract large amounts of data from Hive. Hive can write to HDFS directories in parallel from within a MapReduce job.

- **Warning: The directory is OVERWRITTEN!**
  – if the specified path exists, it is clobbered and replaced with output.

# SELECT FROM

- If you know SQL, then HiveQL's DML has few surprises

| users |
| --- |
| id<br>office_id<br>name<br>children |

- Simple SELECT all query:

```
hive> SELECT * FROM users LIMIT 3;
1       1       Bob Smith       ["Mary"]
2       1       Frank Barney    ["James","Liz","Karen"]
3       2       Ellen Lacy      ["Randy","Martin"]
```

  – Note the last column is ARRAY data type. Hive outputs the array elements in brackets.
  – LIMIT puts an upper limit on number rows returned

- SELECT query (ARRAY indexing):

```
hive> SELECT name, children[0] FROM users;
Bob Smith       Mary
Frank Barney    James
Ellen Lacy      Randy
```

  – First element from children ARRAY is selected.

- SELECT query with STRUCT column (address):

| Customer |
| --- |
| id<br>cust_name<br>phone<br>Address<br>ytd_sales |

```
hive> SELECT cust_name, address FROM customer;
Bruce Smith     {“city”:”Burlington”,”zipcode”:05401}
Chuck Barney    {“city”:”Jericho”,”zipcode”:05465}
```

  – The address column (type STRUCT) prints out in JSON format

# WHERE

- Predicate Operators
  =, <>, !=, <, <=, >, >=, IS NULL, IS NOT NULL, LIKE, RLIKE

- SELECT WHERE query with STRUCT column (address):

```
hive> SELECT cust_name, address.city FROM customer
        WHERE address.city = 'Burlington';
        Bruce Smith     Burlington
```

  – Dot notation used to access struct data

- SELECT WHERE query with GROUP BY clause:

```
hive> SELECT address.city, sum(ytd_sales) FROM customer
        WHERE address.city = 'Burlington'
        GROUP BY address.city;
```

  – GROUP BY usually used with aggregate functions
  – Can also use the HAVING clause with GROUP BY

# Column Alias and Nested Select



- SELECT query with column alias

```
hive> SELECT cust_name, address.city as city
        FROM customer;
```

  - Address.city is given the column alias “city”

- Column alias comes in handy when doing nested SELECT…

```
hive> FROM(
            SELECT cust_name, round(ytd_sales * .01) as rewards
            FROM customer;
       ) c
      SELECT c.name, c.rewards
      WHERE c.rewards > 25;
```

  - The first query is aliased as “c”. We then select from the results of that query.

# Selecting from Partitions

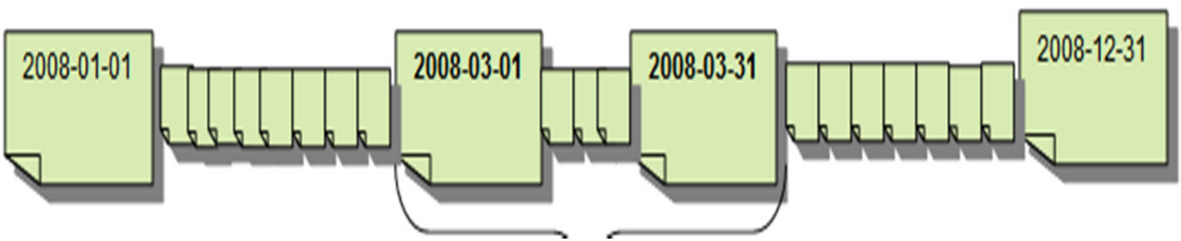


**Taking advantage of partitions**

- Partition pruning – Hive scans only a fraction of the table relevant to the partitions specified by the query.
- Hive does partition pruning if the partition predicates are specified in the WHERE clause or the ON clause in a JOIN.
- Example: If table `page_views` is partitioned on `log_date`, the following query retrieves rows for just days between `2008-03-01` and `2008-03-31`.

```
hive> SELECT * FROM page_views
        WHERE log_date >= '2008-03-01' AND log_date <= '2008-03-31';
```

  – Imagine we had a year of page_view data partitioned into 365 partition files. Hive only needs to open and read the 31 partitioned data files in the above example.



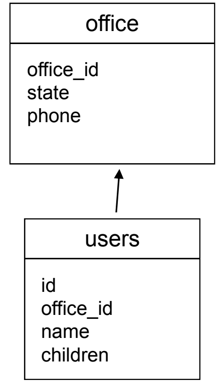
# Joins



- Hive supports joins via ANSI join syntax only

```
select ...
  from T1, T2
 where T1.a = T2.b
```

```
select ...
  from T1 JOIN T2
        on T1.a = T2.b
```




- Example join

```
hive> SELECT o.state as state, count(*) as employees
  FROM office o LEFT OUTER JOIN users  u
     on u.office_id = o.office_id
GROUP BY o.state
ORDER BY state;
```

- Note Hive only supports equi-joins
- Inner Join, Left Outer, Right Outer, Full Outer, Left Semi-Join are supported
- Largest table on right for performance
- Limited support for subqueries, only permitted in the FROM clause of a SELECT statement.
  - Sometimes can be rewritten using Joins

# Order BY / SORT BY

- ORDER BY clause is similar to other versions of SQL
  - Performs total ordering of result set - through single reducer (Hadoop)
  - Large data sets = high latency
- Hive's SORT BY clause
  - Data is ordered in each reducer - each reducer's output is sorted
  - If multiple reducers - final results may not be sorted as desired

# Casting



- Implicit conversions when compare one type to another
- Use the cast() function for explicitly conversions
- If salary is a STRING data type, we can explicitly cast it to FLOAT as seen below

```
hive> SELECT first_name, team
        FROM players
        WHERE cast(salary AS FLOAT) > 100.0;
```

If “salary” not a string that could be converted to a floating point number, then Null

# Views



- Allow query to be saved and treated like table
- Logical – doesn't store data
- Hides complexity of long query
- Earlier example of a "complex" query

```
hive> FROM(
              SELECT cust_name, round(ytd_sales * .01) as rewards FROM customer;
            ) c
      SELECT c.name, c.rewards
      WHERE c.rewards > 25;
```

Can be simplified using a View….

```
hive> CREATE VIEW rewards_view AS
          SELECT cust_name, round(ytd_sales * .01) as rewards FROM customer;

hive> SELECT name, rewards FROM rewards_view
      WHERE rewards > 25;
```

- Drop a view

```
hive> DROP VIEW IF EXISTS rewards_view
```

# Explain



- The explain keyword generates a Hive explain plan for the query

```
hive> EXPLAIN SELECT o.state as state, count(*) as employees
  FROM office o LEFT OUTER JOIN users  u
     on u.office_id = o.office_id
GROUP BY o.state
ORDER BY state;

STAGE DEPENDENCIES:
  Stage-1 is a root stage
  Stage-2 depends on stages: Stage-1
  Stage-3 depends on stages: Stage-2
  Stage-0 is a root stage
...
```

- Query requires three MapReduce jobs to implement
  - One to join the two inputs
  - One to perform the GROUP BY
  - One to perform the final sort

- FYI – there also is the EXPLAIN EXTENDED keyword which gives even more detail

- Details are beyond the scope of this presentation

# History of Hive

- Initially developed at Facebook in 2007 to handle massive amounts of growth.
  - Dataset growth from 15TB to 700TB over a few years
  - RDBMS Data Warehouse was taking too long to process daily jobs
  - Moved their data into scalable open-source Hadoop environment
  - Using Hadoop / creating MapReduce programs was not easy for many users
  - Vision: Bring familiar database concepts to the unstructured world of Hadoop, while still maintaining Hadoop's extensibility and flexibility
  - Hive was open sourced in August 2008
  - Currently used at Facebook for reporting dashboards and ad-hoc analysis

# What is Hive?

- Data Warehouse system built on top of Hadoop
  – Takes advantage of Hadoop distributed processing power
- Facilitates easy data summarization, ad-hoc queries, analysis of large datasets stored in Hadoop
- Hive provides a SQL interface (HQL) for data stored in Hadoop
  – Familiar, Widely known syntax
  – Data Definition Language and Data Manipulation Language
- HQL queries implicitly translated to one or more Hadoop MapReduce job(s) for execution
  – Saves you from having to write the MapReduce programs!
  – Clear separation of defining the *what* (you want) vs. the *how* (to get it)
- Hive provides mechanism to project structure onto Hadoop datasets
  – Catalog ("metastore") maps file structure to a tabular form

# What Hive is not…

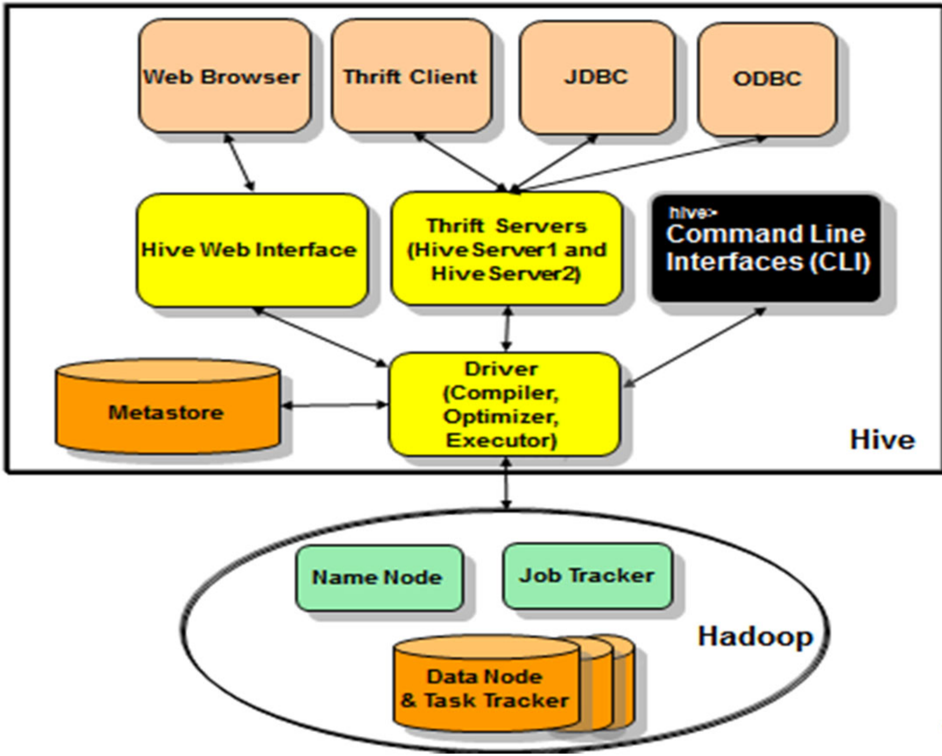


- **Hive is not a full database - but it fits alongside your RDBMS.**
- Is not a real-time processing system
  - Best for heavy analytics and large aggregations – Think Data Warehousing.
  - Latencies are often much higher than RDBMS
  - Schema on Read
    - Fast loads and flexibility – at the cost of query time
    - Use RDBMS for fast run queries.

  Not SQL-92 compliant
  - Does not provide row level inserts, updates or deletes
  - Doesn't support transactions
  - Limited subquery support
- Query optimization still a work in progress
- See HBase for rapid queries and row-level updates and transactions

# Hive Components





# Hive Directory Structure

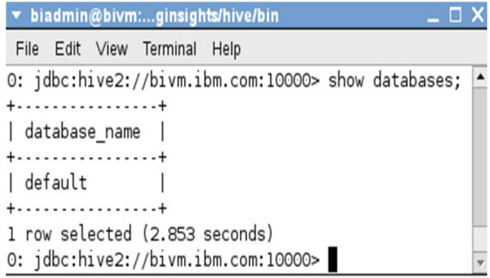


- Lib directory
  - $HIVE_HOME/lib
  - Location of Hive JAR files
  - Contain the actual Java code that implement the Hive functionality
- Bin directory
  - $HIVE_HOME/bin
  - Location of Hive Scripts/Services
- Conf directory
  - HIVE_HOME/conf
  - Location of configuration files

# CLI (Command Line Interface)



- Most common way to interact with Hive
- From the shell you can
  - Perform queries, DML, and DDL
  - View and manipulate table metadata
  - Retrieve query explain plans (execution strategy)



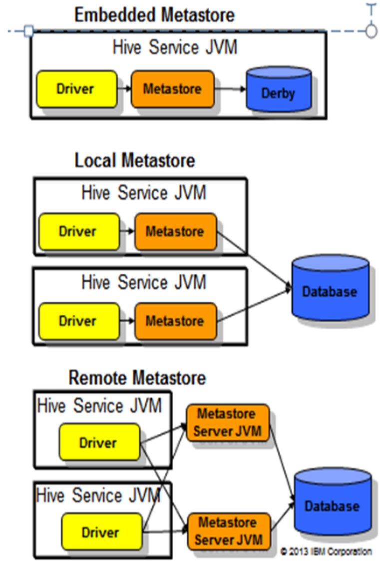

- The Hive Beeline shell and original CLI are located in **$HIVE_HOME**/bin/hive

```
$ $HIVE_HOME/bin/hive
2013-01-14 23:36:52.153 GMT : Connection obtained for host: master-
Logging initialized using configuration in file:/opt/ibm/biginsight
Hive history file=/var/ibm/biginsights/hive/query/biadmin/hive_job

hive> show tables;
mytab1
mytab2
mytab3
OK
Time taken: 2.987 seconds
hive> quit;
```

# Metastore



- 2 pieces – Service & Datastore
- Stores Hive Metadata in 1 of 3 configs:
  - **Embedded:** in-process metastore, in-process database
  - **Local:** in-process metastore, out-of-process database
  - **Remote:** out-of-process metastore, out-of-process database
  - If metastore not configured - Derby database is used
    - Derby metastore allows only one user at a time
  - Can be configured to use a wide variety of storage options (DB2, MySQL, Oracle, XML files, etc.) for more robust metastore



# Real world use cases



- CNET: “We use Hive for data mining, internal log analysis and ad hoc queries.”
- Digg: “We use Hive for data mining, internal log analysis, R&D, and reporting/analytics.”
- Grooveshark: “We use Hive for user analytics, dataset cleaning, and machine learning R&D.”
- Papertrail: “We use Hive as a customer-facing analysis destination for our hosted syslog and app log management service.”
- Scribd: “We use hive for machine learning, data mining, ad-hoc querying, and both internal and user-facing analytics.”
- VideoEgg: “We use Hive as the core database for our data warehouse where we track and analyze all the usage data of the ads across our network.”

# Overview of Sqoop

- Transfers data between Hadoop and relational databases
  - Uses JDBC
  - Must copy the JDBC driver JAR files for any relational databases to $SQOOP_HOME/lib
- Uses the database to describe the schema of the data
- Uses MapReduce to import and export the data
  - Import process creates a Java class
    - Can encapsulate one row of the imported table
  - The source code of the class is provided to you
    - Can help you to quickly develop MapReduce applications that use HDFS-stored records

# Sqoop connection



- Database connection requirements are the same for
  - Import
  - Export
- Specify
  - JDBC connection string
  - Username
  - Password

# Sqoop import

- Imports data from relational tables into HDFS
  - Each row in the table becomes a separate record in HDFS
- Data can be stored
  - Text files
  - Binary files
  - Into HBase
  - Into Hive
- Imported data
  - Can be all rows of a table
  - Can limit the rows and columns
  - Can specify your own query to access relational data
- --target-dir
  - Specifies the directory in HDFS in which to place the data
    - If omitted, the name of the directory is the name of the table

# Sqoop import examples



- Import all rows from a table

```
sqoop import --connect jdbc:db2://your.db2.com:50000/yourDB \ --
 username db2user --password db2password --table db2table \ --
 target-dir sqoopdata
```

- Addition parameters

--split-by tbl_primarykey

--columns "empno,empname,salary"

--where "salary > 40000"

--query 'SELECT e.empno, e.empname, d.deptname FROM employee e JOIN department d on (e.deptnum = d.deptnum)'

--as-textfile

--as-avrodatafile

--as-sequencefile

# Sqoop exports

- Exports a set of files from HDFS to a relation database system
  - Table must already exist
  - Records are parsed based upon user's specifications
- Default mode is *insert*
  - Inserts rows into the table
- *Update* mode
  - Generates update statements
  - Replaces existing rows in the table
  - Does not generate an *upsert*
    - Missing rows are not inserted
    - Not detected as an error
- *Call* mode
  - Makes a stored procedure call for each record
- --export-dir
  - Specifies the directory in HDFS from which to read the data

# Sqoop exports



- Basic export from files in a directory to a table

```
sqoop export --connect jdbc:db2://your.db2.com:50000/yourDB \       --
 username db2user --password db2password --table employee \         --
 export-dir /employeedata/processed
```

- Example calling a stored procedure

```
sqoop export --connect jdbc:db2://your.db2.com:50000/yourDB \       --
 username db2user --password db2password --call empproc \       --
 export-dir /employeedata/processed
```

- Example updating a table

```
sqoop export --connect jdbc:db2://your.db2.com:50000/yourDB \       --
 username db2user --password db2password --table employee \         --
 update_key empno --export-dir /employeedata/processed
```

# Additional export information



- Parsing data
  - Default is comma separated fields with newline separated records
  - Can provide input arguments that override the default
    - If the records to be exports loaded into HDFS using the import command
      - The original generated Java class can be used to read the data

- Transactions
  - Sqoop uses multi-row insert syntax
  - Inserts up to 100 rows per statement
  - Commits work every 100 inserts
    - Commit every 10,000 rows
  - Each export map task operates as a separate transaction

# Distributed Systems



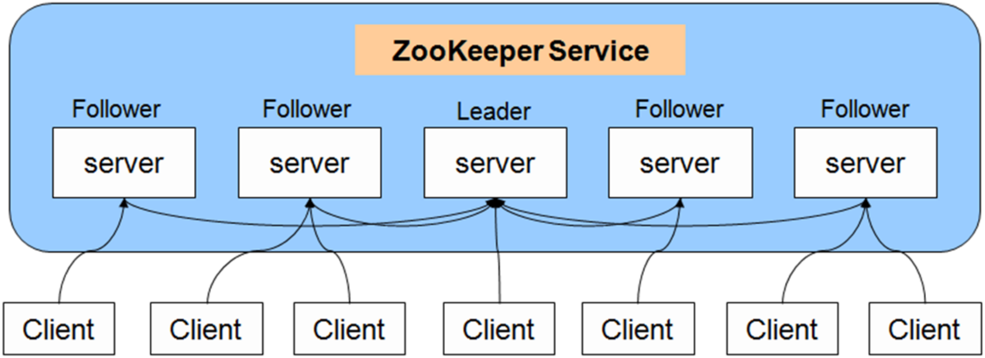
- Multiple software components on multiple computers, but run as a single system
- Computers can be physically close (local network), or geographically distant (WAN)
- The goal of distributed computing is to make such a network work as a single computer
- Distributed systems offer many benefits over centralized systems
  - **Scalability**
    - System can easily be expanded by adding more machines as needed
  - **Redundancy**
    - Several machines can provide the same services, so if one is unavailable, work does not stop
    - Smaller machines can be used, redundancy not prohibitively expensive

# What is ZooKeeper?

- Distributed applications require coordination
  - Develop your own service (lot's of work!) or use robust pre-existing (ZooKeeper)
- Distributed open-source centralized coordination service for:
  - Maintaining configuration information
    - E.g. - Sharing config info across all "nodes"
  - Naming
    - E.g. - Find a machine in a cluster of 1,000s of servers – Naming Service
  - Providing distributed synchronization
    - E.g. - Locks, Barriers, Queues…
  - Providing group services
    - E.g. - Leader election and more
- The ZooKeeper service is Distributed, Reliable, Fast….and Simple!

# ZooKeeper Service – Replicated Mode

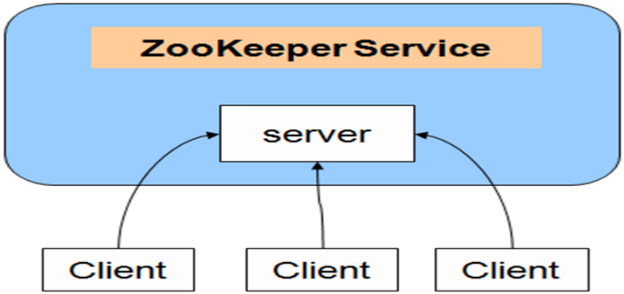





- ZooKeeper runs on cluster of machines – "Ensemble"
- High availability and Consistency
  - Requires majority of servers – 7 node ensemble can lose 3 nodes
- Servers that make up the ZooKeeper service know about each other
  - Maintain an in-memory image of state
- Clients connect to only one server
  - If they loose the connection, can connect to another server automatically
- Writes go through leader – requires majority consensus

# ZooKeeper Service – Standalone Mode






- Single ZooKeeper server
- Good for testing/learning
- Lacks benefits of Replicated mode
  - No guarantee of high-availability or resilience

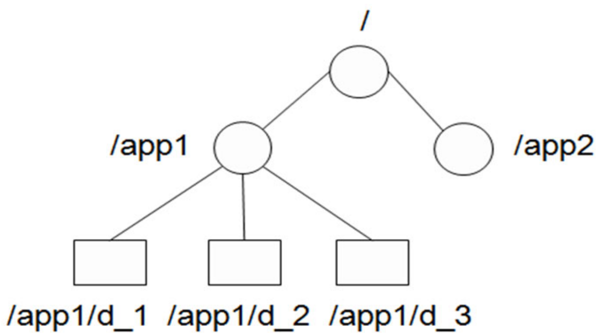
# Consistency Guarantees

- Sequential Consistency
  - Client updates are applied in order
- Atomicity
  - Updates succeed OR fail
- Single system image
  - Client sees same view of ZooKeeper service regardless of server
- Reliability
  - If update succeeds then it persists
- Timeliness
  - Client view of system is guaranteed up-to-date within a time bound
    - Generally within tens of seconds
    - If client does not see system changes within time bound, then service-outage

# ZooKeeper Structure – Data Model



- Distributed processes coordinate through shared hierarchal namespaces
  - These are organized very similarly to standard UNIX and Linux file systems
- A namespace consists of data registers
  - Called znodes
  - Similar to files and directories
  - node holds data, children, or both



# Role in Hadoop Infrastructure



- HBase
  - Use ZooKeeper for master election, server lease management, bootstrapping, and coordination between servers.
- Hadoop and MapReduce
  - Using ZooKeeper to aid in high availability of ResourceManager
  - Adaptive MapReduce (IBM Big Insights)
- Flume
  - Using ZooKeeper for configuration purposes in recent releases

# Real world use cases



- **Rackspace** – **"***The Email & Apps team uses ZooKeeper to coordinate sharding and responsibility changes in a distributed e-mail client that pulls and indexes data for search. ZooKeeper also provides distributed locking for connections to prevent a cluster from overwhelming servers."*

- **Twitter –** Service Discovery

- **Vast.com** – *"Used internally as a part of sharding services, distributed synchronization of data/index updates, configuration management and failover support"*

- **Yahoo! –** *"ZooKeeper is used for a myriad of services inside Yahoo! for doing leader election, configuration management, sharding, locking, group membership etc."*

- ***Zynga*** **–** *"…used for a variety of services including configuration management, leader election, sharding and more..."*

*(sources: Apache ZooKeeper wiki and blog.twitter.com)*

# Problem with unstructured data

- Structured data has:
  - Known attribute types
    - Integer
    - Character
    - Decimal
  - Known usage
    - Represents salary versus zip code
- Unstructured data has:
  - No known attribute types nor usage
- Usage is based upon context
  - Tom Brown has brown eyes
- A computer program has to be able view a word in context to know its meaning

# Need to harvest unstructured data

- Most data is unstructured
- Most data used for personal communication is unstructured
  - email
  - instant messages
  - tweets
  - blogs
  - forums
- Opinions are expressed when people communicate
  - beneficial for marketing
  - give insight of customer sentiment of you, as well as your competitors

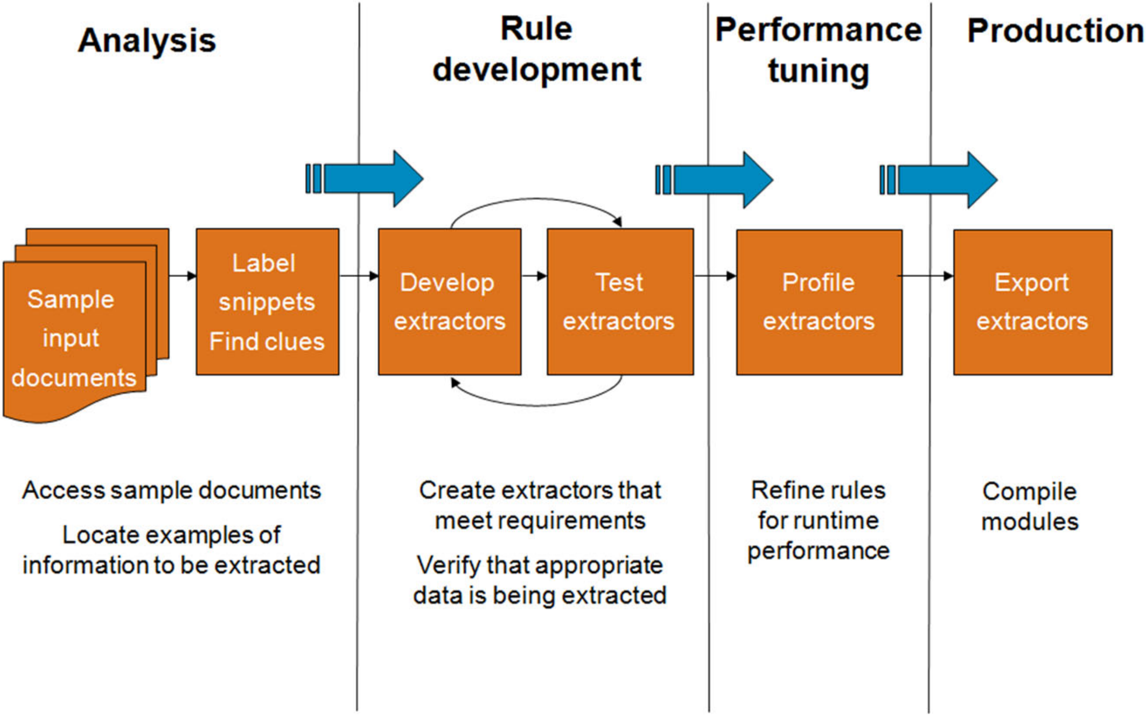
# Need for structured data



- Business intelligence tools work with structured data
  - OLAP
  - data mining
- To use unstructured data with business intelligence tools requires that structured data to be extracted from unstructured and semi-structured data
- IBM BigInsights provides a language, Annotation Query Language (AQL)
  - syntax is similar to that of Structured Query Language (SQL)
  - builds extractors to extract structured data from
    - unstructured data
    - semi-structured data

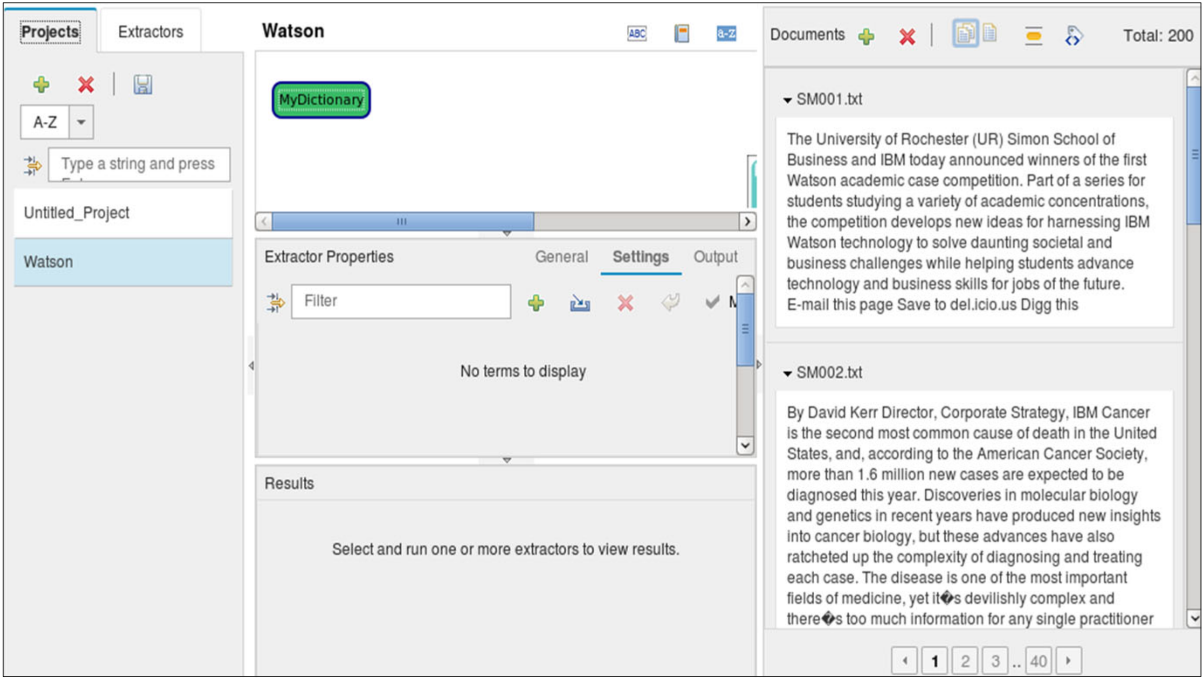
# Approach for text analytics





# Web Tooling overview





# Basic components of an extractor



- **Literal**
  - Match a single term
- **Dictionary**
  - Match from a list of terms
  - Case sensitive or insensitive
  - Can be imported from a text file
  - You can match multiple terms to the same term with a **mapping table**
    - Example: match personal names to common nicknames
- **Regular Expression**
- **Proximity Rule**
  - Extract spans that occur within specified distance of each other
  - Distance measured in characters or tokens (words)

# What is open source R?



- R is a powerful programming language and environment for statistical computing and graphics.
- R offers a rich analytics ecosystem:
  - Full analytics life-cycle
    - Data exploration
    - Statistical analysis
    - Modeling, machine learning, simulations
    - Visualization
  - Highly extensible via user-submitted packages
    - Tap into innovation pipeline contributed to by highly-regarded statisticians
    - Currently 4700+ statistical packages in the repository
    - Easily accessible via CRAN, the Comprehensive R Archive Network
  - R is the fastest growing data analysis software
    - Deeply knowledgeable and supportive analytics community
    - The most popular software used in data analysis competitions
    - Gaining speed in corporate, government, and academic settings

# The R appeal: what attracts users?



- R is an integrated suite of software facilities
  - Simple and effective programming language
  - Variety of open source GUIs for increased productivity
  - Publication-quality graphics capabilities
- Cutting edge algorithms
  - Statisticians usually first contribute their algorithms to R
  - Contributors are often working on today's most challenging data analysis
  - Algorithms developed for life sciences, finance, marketing, etc.
- Accessibility and education
  - Open source with many free online educational tools to help you learn R
  - Universities often teach data science skills with R
  - The network effect of R and its highly extensible packages system

# Companies currently using R

AMERICAN EXPRESS

JOHN DEERE

NOVARTIS

facebook

LLOYD'S

ORBITZ

McGraw Hill Education

P&G

FDA

mozilla Firefox

TEXAS INSTRUMENTS

Ford

The New York Times

Thomas Cook

Google

NORDSTROM

# What is the R programming language?



- Multi-paradigm programming language
  - Designed from the ground-up for statistical computation and graphics
  - http://cran.r-project.org/manuals.html
- Interactive, functional programming semantics
  - Allows "computing on the language"
  - Systematize repetitive work with functions, packages, scripts
- Simple expressions, with strong support for object orientation
  - Incremental and interactive addition of user-defined object orientation
  - More support for OOP than other statistics languages

# Limitations of open source R

- R was originally created as a single user tool
  - Not naturally designed for parallelism
  - Can not easily leverage modern multi-core CPUs
- Big data > RAM
  - R is designed to run in-memory on a shared memory machine
  - Constrains the size of the data that you can reasonably work with
- Memory capacity limitation
  - Forces R users to use smaller datasets
  - Sampling can lead to inaccurate or sub-optimal analysis

**Key Take-Away**

Open Source R is a powerful tool, however, it has limited functionality in terms of **parallelism and memory**, thereby bounding the ability to analyze big data.

# Open source R packages to boost performance



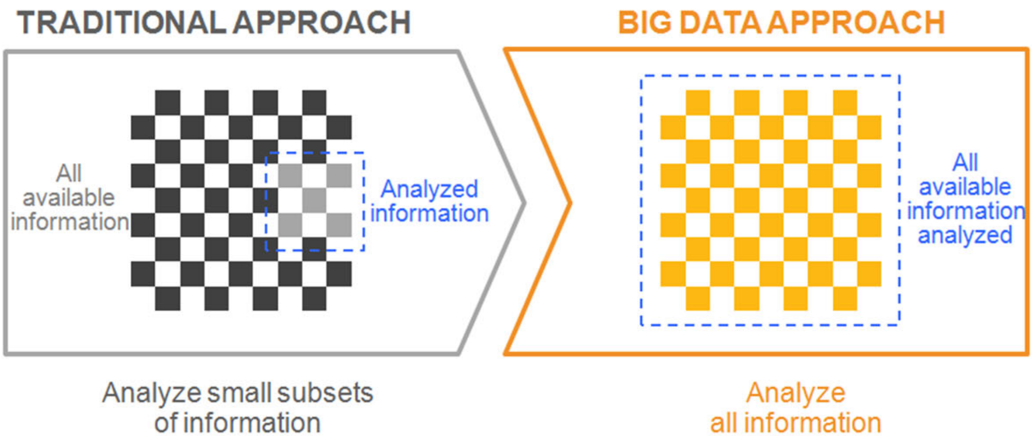
- Packages that mitigate R's parallelism and memory capacity problems
  - Rhadoop
  - RHIPE
  - Hadoop Streaming
  - Parallel
  - Snow
  - Multicore
  - BigMemory

**Key Take-Away**

Open Source R has packages for helping to deal with parallelism and memory constraints, however, these packages **assume advanced parallel programming skills** and **require significant time-to-value.**

# Challenges with running large-scale analytics





# 3 key capabilities in Big R



- End-to-end integration of R into BigInsights Hadoop

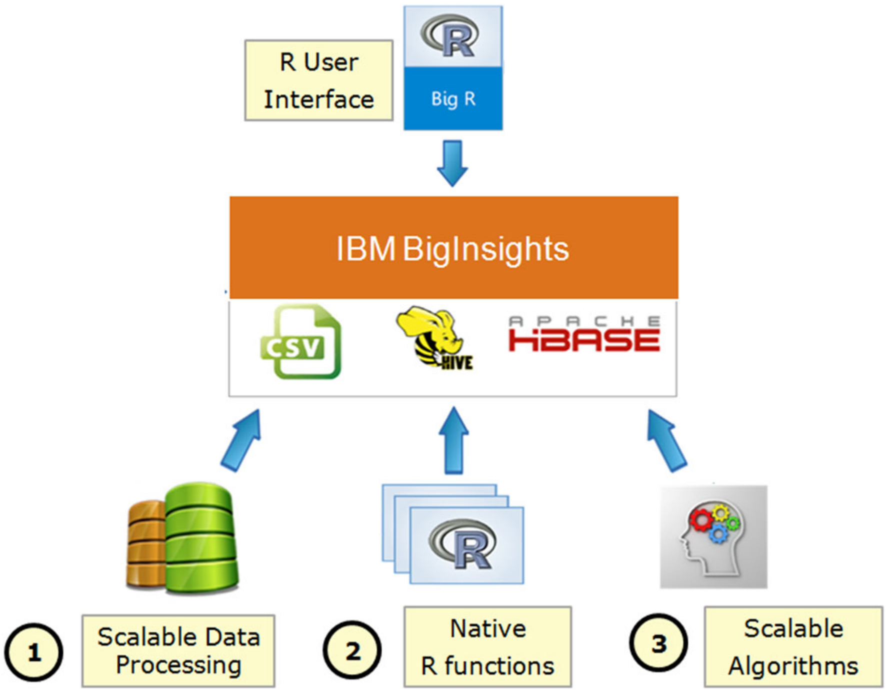
1. Use of R as a language on Big Data
- Scalable data processing

2. Running native R functions in Hadoop
- Can leverage existing R assets

3. Running scalable algorithms beyond R in Hadoop
- wide class of algorithms and growing
- R-like syntax to develop new algorithms and customize existing algorithms

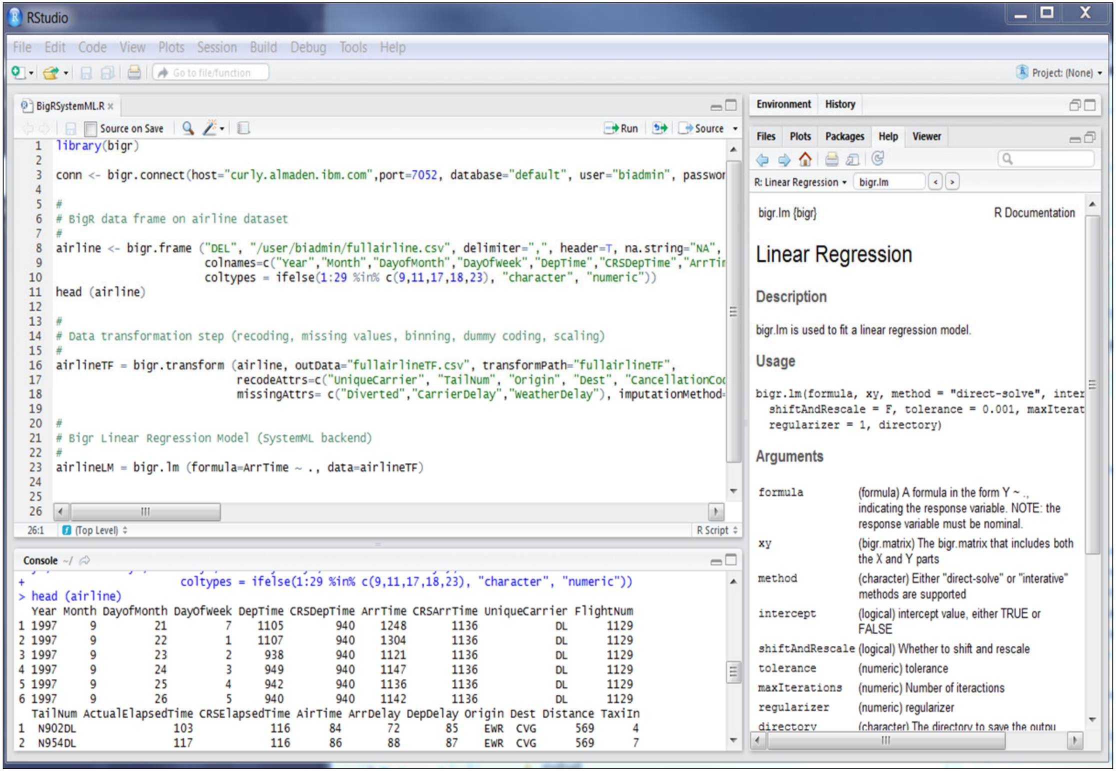
# Big R architecture





# User experience for Big R





# What's behind running Big R's scalable algorithms?

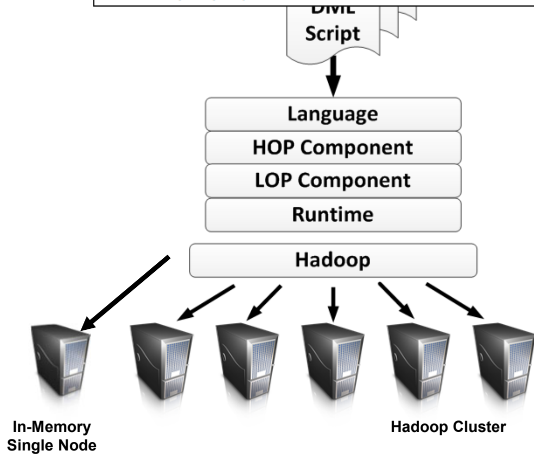
IBM ICE (Innovation Centre for Education)

- Declarative analytics:
  1) Future-proof algorithm investment
  2) Automatic performance tuning

- High-level declarative language with R-like syntax shields your algorithm investment from platform progression
- Cost-based compilation of algorithms to generate execution plans
  – Compilation and parallelization
    - Based on data characteristics
    - Based on cluster and machine characteristics
  – In-Memory single node and MR execution

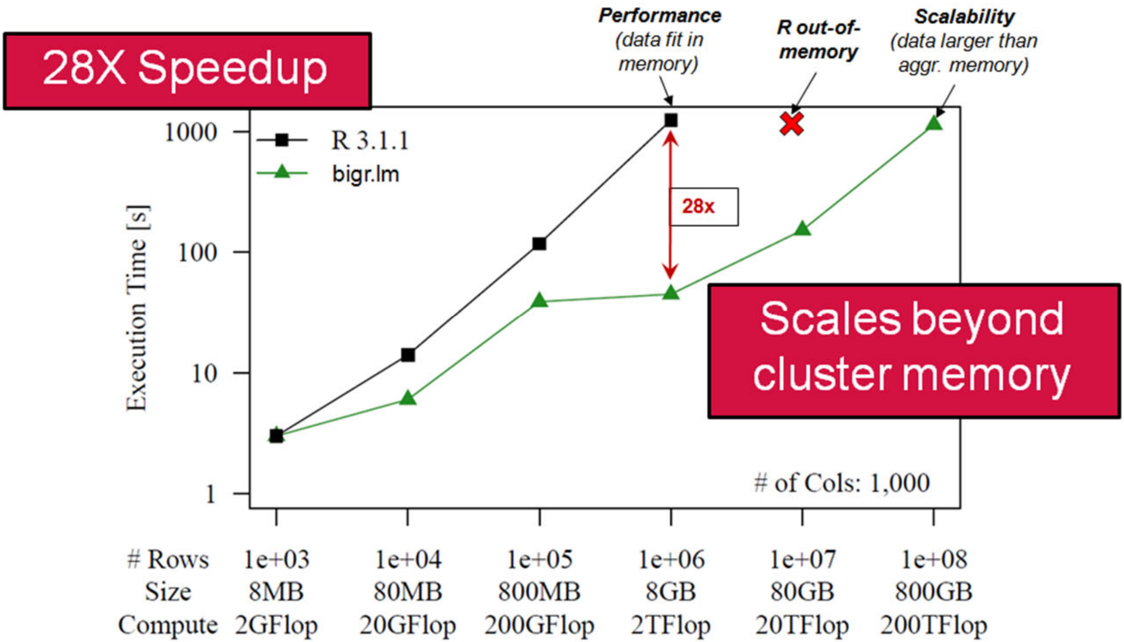
- Enable algorithm developer productivity to build additional algorithms (scalability, numeric stability and optimizations)

```
# THIS SCRIPT SOLVES LINEAR REGRESSION USING A
#   DIRECT SOLVER FOR (X^T X + lambda) and beta = X^T y

X = read ($X);    # explanatory variables
y = read ($Y);    # predicted variables

n = nrow (X);
m = ncol (X);

# Rescale the columns of X if needed
scale_lambda = matrix (1, rows = 1, cols = m);
lambda = t(scale_lambda) * $reg;

# Construct and solve system of equations
A = t(X) %*% X + diag (lambda);
b = t(X) %*% y;

beta = solve (A, b);
...
write (beta, $B);
```



# Big R machine learning: scalability and performance





# Simple Big R example



```
# Connect to BigInsights
> bigr.connect(host="192.168.153.219", user="bigr", password="bigr")

# Construct a bigr.frame to access large data set
> air <- bigr.frame(dataSource="DEL", dataPath="airline_demo.csv", …)

# Filter flights delayed by 15+ mins at departure or arrival
> airSubset <- air[air$Cancelled == 0
       & (air$DepDelay >= 15 | air$ArrDelay >= 15),
      c("UniqueCarrier", "Origin", "Dest",
       "DepDelay", "ArrDelay", "CRSElapsedTime")]
# What percentage of flights were delayed overall?
> nrow(airSubset) / nrow(air)
```

- [1] 0.2269586
- # What are the longest flights?
- > bf <- sort(air, by = air$Distance, decreasing = T)
- > bf <- bf[,c("Origin", "Dest", "Distance")]
- > head(bf, 3)
- Origin Dest Distance
- 1  HNL JFK   4983
- 2  EWR HNL   4962
- 3  HNL EWR   4962

# Checkpoint (1 of 4)



1. Expand HDFS
   a. Hadoop Distributed File System
   b. Hadoop Data File System
   c. Hadoop De-centralized File System
   d. Hadoop Distributed Fact System

2. HDFS stores data across multiple
   a. Volumes
   b. Databases
   c. Nodes
   d. Disks

# Checkpoint solution (1 of 4)



1. Expand HDFS
   a. Hadoop Distributed File System
   b. Hadoop Data File System
   c. Hadoop De-centralized File System
   d. Hadoop Distributed Fact System

2. HDFS stores data across multiple
   a. Volumes
   b. Databases
   c. Nodes
   d. Disks

# Checkpoint (2 of 4)



3. ________ Contains common utilities and libraries that support the other Hadoop sub projects
   a. Hadoop Common
   b. Hadoop Specific
   c. HDFS
   d. HIVE

4. Which is not an accelerator in Hadoop
   a. MDA
   b. SDA
   c. TEDA
   d. PIG

# Checkpoint solutions (2 of 4)



3. _________ Contains common utilities and libraries that support the other Hadoop sub projects
   a. Hadoop Common
   b. Hadoop Specific
   c. HDFS
   d. HIVE

4. Which is not an accelerator in Hadoop
   a. MDA
   b. SDA
   c. TEDA
   d. PIG

# Checkpoint (3 of 4)



5. Identify the structured database among the following

a. HIVE
b. PIG
c. JAQL
d. Oracle

6. Which component of IBM Infosphere is used for Reporting?

a. JAQL
b. PIG
c. Bigsheets
d. Sqoop

# Checkpoint solution (3 of 4)



5. Identify the structured database among the following

a. HIVE
b. PIG
c. JAQL
d. Oracle

6. Which component of IBM Infosphere is used for Reporting?

a. JAQL
b. PIG
c. Bigsheets
d. Sqoop

# Checkpoint (4 of 4)



7. What is the name of interface which is used to perform analytics activities under big data technology?

a. Big R
b. SPSS
c. Cognos
d. HIVE

8. Which type of data is needed to perform reporting activities using Business intelligence technique?

a. Structured
b. Semi Structured
c. Unstructured
d. Live

# Checkpoint solution (4 of 4)



7. What is the name of interface which is used to perform analytics activities under big data technology?

a. Big R
b. SPSS
c. Cognos
d. HIVE

8. Which type of data is needed to perform reporting activities using Business intelligence technique?

a. Structured
b. Semi Structured
c. Unstructured
d. Live

# Unit summary



Having completed this unit, you should be able to:

- Understand the Approaches to Big Data reporting and analysis
- Do Business Intelligence, reporting using Hadoop Architecture
- Do Direct Batch Reporting on Hadoop
- Explore Big Data ‘R’
- Do Indirect Batch Analysis on Hadoop