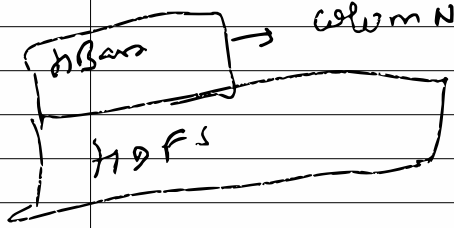
Lecture 9 → HBase Slides

HBase → distributed column oriented data store built on top of HDFS

↳ Apache open source project

↳ storage for hadoop distributed computing

Logically organized into tables → rows and columns



MySQL or SQL → row based DB

HBase	Hadoop
Distributed systems → scale to 100 or 1000s of nodes	distributed systems → scale to 100/1000 of nodes
↳ Fast record lookup	→ good for batch processing
record level insertion	No good → record lookup
updates	updates
↳ by creating new version of values.	incremental addition of small pieces

No random reads / writes → stick to HDFS

HBase run modes
↳ standalone
Distributed

Out of box → HBase runs in standalone mode.
→ configure hbase → edit files
↳ HBase conf directory

edit conf / hbase-env.sh → tell hbase which java to use.
↓
configure
heapsize
options for JVM
location for log files
JAVA_HOME → points to root of java install.

1) standalone Mode

↳ default mode

Hbase does not use HDFS → uses local files

runs all Hbase daemons and a local zookeeper all in the same JVM.

Zookeeper binds to a well known port so clients may talk to Hbase.

Local filesystem & Durability

↳ using Hbase on local filesystem does not currently guarantee durability.

↳ local FS will lose edits if files are not properly closed → very likely to happen when experimenting with a new download.

Need → run Hbase on HDFS to ensure all writes are preserved.

Running on local FS → gets off ground easily and makes familiar with system.

standalone → runs all daemons and zookeeper on same JVM

Forming Hbase →

- Hbase shell ↵

⇒ create table ~~new~~ named ftn2 with a single column family cf.

⇒ create 'ftn2', 'cf' ;

view table

⇒ list 'ftn2' ↵

create → create 'table-name', 'column-family'
view → list

insert → put 'table-name', 'row1', 'cf:a', 'value1'

put 'table-name', 'row2', 'cf:b', 'Value 2'

verify the data by running scan

scan 'table-name' ;

for single row

scan 'table-name', 'row1' ;

delete table

↳ first disable

disable 'table-name'

then drop

drop 'table-name'

stop hbase

⇒ ./bin/stop-hbase.sh

---

2) distributed mode

can be divided in

↳ pseudo-distributed → all daemons run on single node

↳ fully-distributed → daemons are spread across all the nodes in cluster

Pseudo-distributed mode can run against local file system or against an instance of HDFS

Fully-distributed mode can run only on HDFS

Pseudo-distributed

↳ simply a fully distributed mode run on a single host

Use this configuration testing and prototyping on HBase

Don't use this mode for production nor for evaluating Hbase performance.

If want to run on HDFS rather than local FS, setup your HDFS

Ensure you have a working HDFS before proceeding.

Configuration for Pseudo-distributed mode

1) Edit conf/hbase-site.xml

2) Tell hbase to run in (pseudo)-distributed mode rather than in default standalone mode.

↳ set → hbase.cluster.distributed → true

default false

↳ with this Hbase start up
Hbase & master process
Zookeeper server
Region server

temp files will be saved in → hbase-username.

Using HDFS

Setting up Hbase to write in local file system's temporary directory is ephemeral step.

↳ it will lose all the data once the system is shut down.

For permanent setup we make use of an instance of HDFS; Hbase data will be written to the hadoop distributed file system rather than local file system's temp directory

↳ change → hbase-site.xml

```
<name> hbase.rootdir </name>
<value> hdfs://localhost:9000/hbase </value>
```

Starting initial hbase cluster

start hbase → /start-hbase.sh

start back up master → bin/local-master-backup.sh start 1.

we can start upto 10 backups.

start region servers → bin/local-regionservers.sh start 1

we can start total of 100 region servers.

Fully Distributed mode

For running hbase on fully distributed mode set hbase.cluster.distributed to true → TRUE

point hbase.rootdir to your namenode ✋.

If namenode.example is running on port 8020

```
<name> hdfs.rootdir </name>
<value> hdfs://namenode.example.org:8020/hbase
                                        </value>
```

HDFS client configuration

add pointer to hadoop-conf_dir to the hbase_classpath in hbase-env.sh

example of such hdfs client configuration :
dfs.replication.

If we want to run with a replication factor of 5, hbase will create files with default of 3 unless you do the above to make the configuration available to Hbase.

Master host → 16010
Region server → 16020

Hbase Data Model

is based on Google's Bigtable model

↳ key - value pairs.

Row key : column family

timestamp + values.

is for updation of value

Hbase logical view

Row key (implicit primary key in rdbms terms)	Data → is all byte[] in hbase

Diff type of rows have diff set of columns

table is sparse

Different cells may have different values at different timestamp.

Each row has a key ✓

Each record is divided into column families ↳ Each column family consists of one or more columns

Key
↳ - byte array
- serves as the primary key for the table
- indexed for fast lookup

Column family
- Has a name : string
- contains one or more related columns

column
- belong to one column family
- included inside the row
↳ familyName : columnName.

Version Number
- Unique within each key
- By default → system's timestamp
- Datatype is Long.

Value (cell)
- Byte array.

Notes:-
Hbase schema consists of several tables
Each table consists of a set of column families
↳ columns are not part of schema.

Hbase has dynamic columns
↳ because column names are encoded inside the cells
↳ different cells have different columns.

The version number is user supplied

↳ does not have to be inserted in increasing order
version number are unique within each key.

Tables can be very sparse
many cells are empty

keys are indexed as primary key.

Hbase physical model

Each column family is stored in a separate file
↳ HTables.

keys and version number is replicated with each column family

Empty cell are not stored.

Hbase maintains multilevel indexes on values
< key, column family, column name, timestamp >

Example

Row key	Data
cutting	info: {'height': '9', 'state': 'ca'}
	roles: {'ASF': 'Eng'}

info CF

Key	column key	timestamp	cell value
01fs	info:height	1234567...	1

roles CF

Key	column key	timestamp	cell value
01fs	roles: ASF	1234567...	29 a.

Column families

↳ different set of columns may have different properties and access patterns

Configurable by column family
- Compression (none, gzip, LZO)
- version retention policies
- cache priority

CFs stored separately on disk: access one minimizes reading 10 on the other.

Hbase Regions

Each HTable (column family) is partitioned horizontally into regions
↳ comparable to HDFS blocks

HDFS: blocks ⇒ Hbase: HTable → regions

Hbase Architecture:

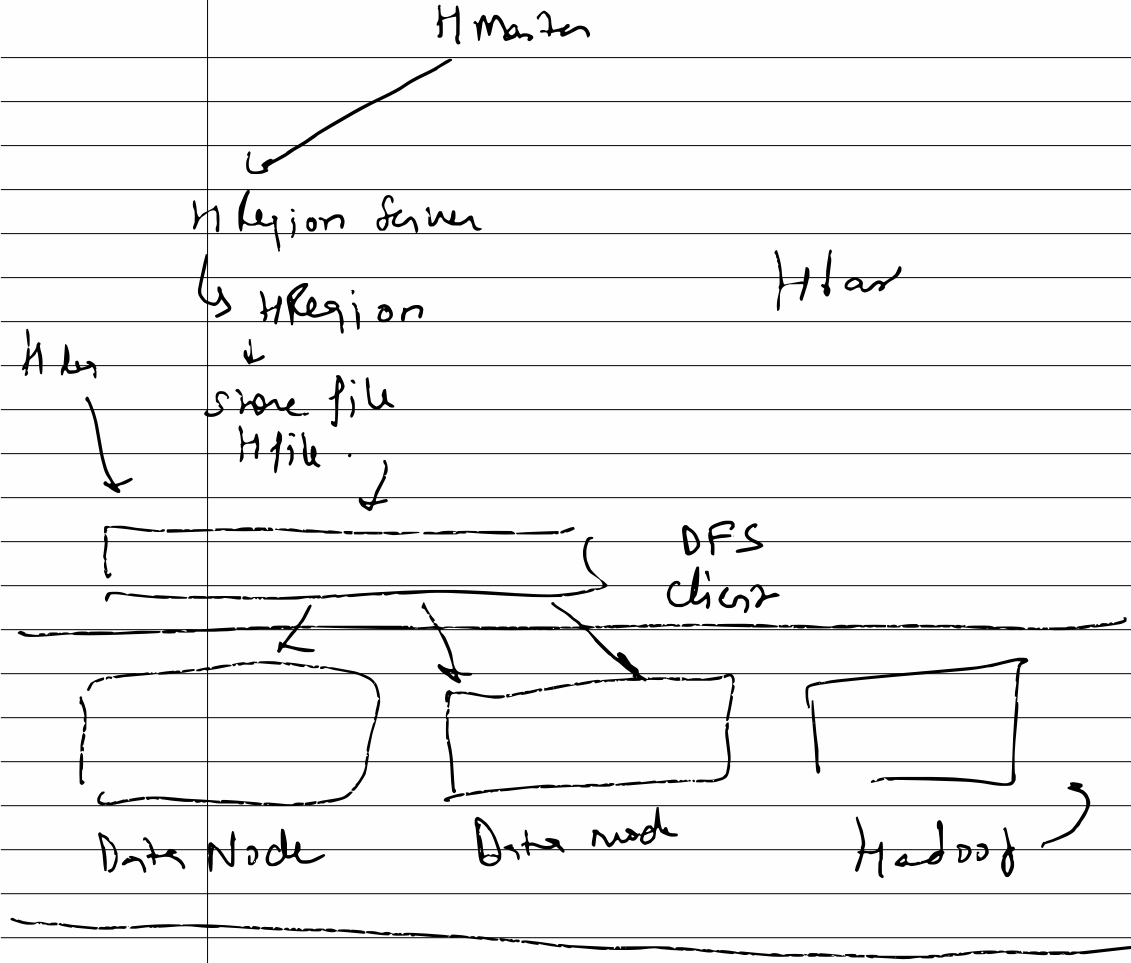
3 major components
- The Hbase master
  - One master
- The HRegion Server
  - Many Region Servers
- The Hbase client

Hbase components

Region
- A subset of table rows
  - like horizontal range partitioning
- automatically done

Region Server (many slaves)
- manages data regions
- serves data for reads and writes (busy - log)

Master
- Responsible for coordinating the slaves
- Assign regions, detects failures
- Admin functions.



Zookeeper

↳ Hbase depends on Zookeeper
→ By default hbase manages the zookeeper instance → starts and stops

→ HMaster and HRegion Server register themselves with Zookeeper.

Creating a Table

```
HBaseAdmin admin = new HBaseAdmin(config);

HColumnDescriptor [] column;

column = new HColumnDescriptor [2];

column[0] = new HColumnDescriptor ("columnFamily1:");
column[1] = new HColumnDescriptor ("columnFamily2:");

HTableDescriptor desc;
  desc = new HTableDescriptor (Bytes.toBytes ("MyTable"));

desc.addFamily (column[0]);
desc.addFamily (column[1]);

admin.createTable (desc);
```

Steps:

1) Create admin
2) create column families
3) create table and add column family to it
4) admin add the table into it.

Operations on Regions :- Get()

→ Given a key → return corresponding record

↓

For each value return the highest version.

⇒ To get current version

```
Get get = new Get(Bytes.toBytes("row1"));
Result r = htable.get(get)
byte[] b = r.getValue(Bytes.toBytes("cf"),
                      Bytes.toBytes("attr"));
```

⇒ We can control number of versions we want
↳ Following code will return last 3 versions of the row.

```
Get get = new Get(Bytes.toBytes("row1"));
get.setMaxVersions(3);
Result r = htable.get(get)
```

Current version as above

All versions by →

```
List<KeyValue> kv = r.getColumn(Bytes.toBytes("cf"),
                                Bytes.toBytes("attr"));
```

↙ all versions

Operations On Regions: Scan ()

↳ It allows iteration over multiple rows for specified attributes.

---

get ()

↳ select value from table where key = "com.apache.www" and label = "anchor:apache.com"

Scan ()

Scan select value from table where anchor = "cnn.com"

Put()

↳ insert a new record with new key.

↳ insert a record for an existing key.

Delete()

Marking table cells as deleted

Multiple levels

↳ can mark an entire column family as deleted

↳ can mark all column family of a given row as deleted.

- All operations are logged by the RegionServers
- The log is flushed periodically.

#Base joins

↳ does not support joins

Can be done in application layer

↳ using scan() and get() operation.

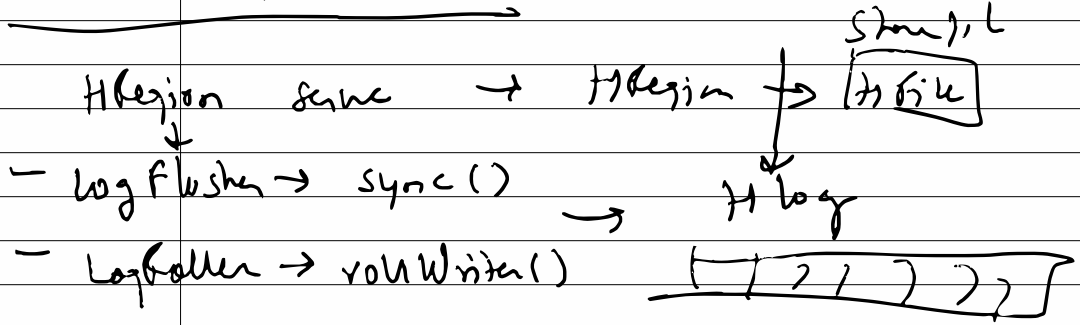
Altering a Table

↳ Disable the table before changing the schema

↳ adding new columnfamily

↳ modifying existing columnfamily.

Logging operation



Hbase Deployment

Master Node → NameNode
Secondary Namenode
HMaster
Job Tracker
Zookeeper

Slave Nodes → RegionServer
Data Nodes
Task Tracker
→ 5 + slaves with HBase, HDFS, MR slave processes.

HBase Vs HDFS

	Plain HDFS/MR	HBase
Write Pattern	Append only	Random write, bulk incremental
Read Pattern	Full table scan, partition table scan	Random read, small range scan or table scan
Hive key →	Very good	4-5x slower.
Structured storage	Do-it-yourself TSV / Seq File / Avro	sparse column family data model.
Max data size →	30 + PB	~ 1 PB.

HBase Vs RDBMS

	RDBMS	HBase
Data layout	Row-oriented	Column family oriented
Transactions →	Multi-row ACID	single row only
Query lan	SQL	get/put/scan/delete etc.
Security	Authentication/Authorization	Work in progress
Indexes	On arbitrary columns	Row-key only
Max data size	TBs	~ 1PB
Read/write throughput limits	1000s queries/second	Millions of queries/second.

When to use HBase

Use if:—

- Need random read, random write, or both (but not neither)
- Need to do many thousands of operations per second on many TB of data
- Your access patterns are well-known and simple.

Don't use HBase if:—

- You only append to your dataset and tend to read the whole thing
- Primarily do ad-hoc analytics (ill-defined access patterns)
- Your data easily fits on one beefy node.

Case Study :- Internet Memory

Google lab published → BigTable

↙

Index designed to manage very large data sets ("petabytes of data") in a cluster of data servers

BigTable supports key search, range search and high throughput file scans, and also provides a flexible storage for structured data.

HBase is open source clone of BigTable.

↳ Assimilated to a large, distributed relational data base.

↳ Presents many aspects common to NOSQL systems :- distribution, fault tolerance, flexible modeling, absence of some features deemed essential in centralized DBMS (e.g. concurrency), etc.

Key Value based storage → like map

↳ No table → No schema

can store value one another map

multi map structure

object remains unaffected by change.

An Hbase table is a multimap structure

↳ Instead of keeping one value for each property of an object, HBases allows the storage of several versions

↳ Each version is identified by a timestamp.

↳ Hbase simply replaces atomic values by a map where the key is the timestamp.

It helps to figure out the power and flexibility of the data representations

Now the document is built from two nested maps,
1) column
2) Timestamp.

The document is globally viewed as a column map.

key | map

column key $\xrightarrow[\text{map}]{}$ as many keys as there are timestamps.

The timestamp maps are completely independent from each others.

We can add as many timestamp keys as we want.

same for column → can add as many columns as we need.

if a table has 2 families meta/data then they are fixed

↳ Unlike column and timestamp maps, the keys in a family map is fixed.

→ We can not add new families to a table once it is created.

→ The family level therefore the equivalent of a relational schema,

↳ Although content of a family value may be a quite complex structure

Row → table

Row has value → key is chosen by use

Operation → get (key, value), put (key, value)

Table map is a sorted map.

↳ rows are grouped in key values.

2) Apache Phoenix

Q) What is phoenix?

Phoenix
↳ open source SQL skin for HBase
↳ Use standard JDBC APIs instead of the regular HBase client APIs to create tables, insert data, query your HBase data

Q)

3) Apache Phoenix

A relational data base layer for Apache HBase

- Query Engine
  - ↳ Transforms SQL queries into native HBase API calls
  - ↳ pushes as much work as possible onto the cluster for parallel execution
- Metadata Repository
  - Typed access to data stored in HBase tables
- A JDBC driver.

Apache HBase - A high performance horizontally scalable data store engine for big data, suitable as the store of record for mission critical data.

Not like a spreadsheet → a sparse, consistent, distributed, multi-dimensional, sorted map.

Scalability by dividing in regions.

↓

Region Servers.

Phoenix puts the SQL back in NoSQL

↳ A complete relational system

↳ reintroduces the familiar declarative SQL interface to data (DDL, DML, etc)

- read only views on existing HBase data
- Dynamic columns extend schema at runtime
- schema is versioned - for free by HBase
  - allowing flashback queries using prior versions of meta data.

→ Reintroduces typed data and query optimizations possible with it

→ secondary indexes, query optimizations, statistics ....

→ integrates the scalable HBase data storage platform as just another JDBC data source.

Accessing HBase data with Phoenix can be substantially faster than direct HBase API use

Parallelizes query
supports secondary indexing
uses every trick based on - Hbase version, meta data, query, reverse scan, small scan, skip scan etc.

Phoenix Jar is installed on the HBase region server classpath.

The Phoenix JDBC driver is installed on the client.

The application speaks SQL to HBase.

Hbase Data model to the relational world.

Phoenix supports read only access to existing HBase tables.

Empty cell will be inserted in each row in the native table to enforce primary key constraints

Note :- serialized bytes in the table must match the expected phoenix type serialization.

6) Mahout is

open source machine learning library from Apache

→ Algorithms fall under broad umbrella of machine learning

At the moment → recommendation engine, clustering and classification

↓

scalable machine learning tool when the data is huge

→ written in Java libraries

No UI, prepackaged server, or an installer

Framework of tools ~~out~~ to be adopted by developers.

Apache Lucene ⟶ Mahout

Much work is to convert mahout algorithms to work at the scale on top of Hadoop.

Many in experimental stage

Three core themes are evident

1) Recommender engines
2) clustering
3) classification

Mr currently has

↳ collaborative filtering
User & Item based recommenders
K-means, Fuzzy K-means clustering
Mean shift clustering
Dirichlet process clustering
Random forest decision tree based classifier
High performance java collections

All techniques work best when provided with large amount of good input data.

Must not only work on large amounts of data but also must produce results very quickly

↳ scalability → Mahout key reason for being.

Hadoop for large data management

↳ Mahout → solving problems suite of Hadoop.

Requires Java 6-
build around maven. → Higher level tool
← title dep → lower level tool.

Add mahout dependency in pom.xml.

Collaborative filtering

↳ recommendation based on user's relationship to item

↓

No prior knowledge of properties of item needed.

Attribute → content based filtering.

content base is good for ~~not~~ very specific

like a content based recommendation system for books will not work for flight recommendation.

Example → User based Recommendation

Input :- large data → in form of preference

↓

Strength of user's ← given a number preference to item

User id, item id, preference

- ids are always a number in mahout.
- preference can be anything as long as large value means strong positive preference.

Data Model → stores and provides access to all the preferences, user and item data needed in the computation.

User Similarity → provides notion of how similar two users are.

User Neighbourhood → group of users and one

Recommender → pull all above together to recommend items to users.

User Test Test data → some training data set aside for testing.

difference → estimate & real value

Root mean square of differences is used

↓

square root of the average of the squares of the difference between actual and estimated values.

↓

heavily penalises diff that are way off.

Precision → portion of top recom from good recom

Recall → portion of good recom that are in top recom.

Spark and Shark

Apache spark is a general purpose cluster in-memory computing system.

Provides high-level APIs in Java, Scala and Python, and an optimized engine that supports general execution graphs.

Provides various high level tools like Spark SQL for structured data preprocessing, MLib for machine learning and more.

Spark framework can be deployed through

- Apache hadoop via YARN
- Apache Mesos
- Spark Cluster Manager.

Spark framework is polyglot

↓

Can be programmed in many languages

- Java
- Python
- Scala

Shark

↳ A fully apache hive compatible data warehousing system that can run 100x faster than Hive.

Spark → 100x faster than Hadoop for certain MapReduce application.

Spark :—

Not a modified version of Hadoop

Separate, fast MR like engine
↳ in memory data storage for very fast iterative queries.
↳ General execution graphs and powerful optimizations
↳ 40x faster than Hadoop.

Compatible with → HDFS, HBase, SequenceFiles etc.

Shark → port of apache hive to run on Spark.

Compatible with existing hive data, metastores, and queries (HiveQL, UDFs etc).

Similar speedup of upto 30x.

MapReduce Successful

↓ User wants

more complex multi stage applications
More interactive ad hoc queries

↓

Need faster data sharing across parallel jobs.

Why spark ?

→ Provides powerful caching and disk persistence capabilities
→ Interactive data analysis
→ faster batch
→ Iterative algorithms
→ Real time stream processing
→ faster decision making.

Data sharing in MapReduce :-

↳ slow due to replication, serialization and disk IO

Data sharing in spark

↳ 10 - 100x faster than network and disk.

Spark Programming Model

Key idea → RDD
↳ Resilient Distributed Datasets

↳ Distributed collections of objects that can be cached in memory across cluster nodes
→ Manipulated through various parallel operators
→ Automatically rebuilt on failure

Interface
↳ Clean language integrated API in Scala
↳ Can be used interactively from Scala console

Example: log mining

scaled to 1 TB in 5-7 sec.

Fault tolerance
↳ RDDs track the series of transformation used to build them to recompute lost data

```
messages = textFile(...).filter(_.contains("error"))
                        .map(_.split('\t')(2))
```

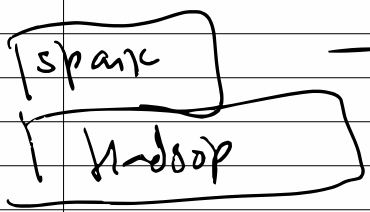
Supported operators
↳ map, reduce, filter, count, groupBy, union, join, sort.

Other engine features

↳ general graphs of operators
↳ hash-based reducer
(faster than Hadoop's sort)
↳ controlled data partitioning to lower communication

Application
↳ In memory analytics & anomaly detection
↳ interactive queries on data streams
↳ exploratory log analysis
↳ traffic estimation w/ GPS data
↳ Twitter spam classification.

Conviva → 40x gain over hive
↓
avoiding
↳ repeated reading, deserialization and filtering.



→ Hive on Spark.

caching hive records as java objects is inefficient due to high performance overhead

Instead, Shark employs column-oriented

storage using arrays of primitive types.

Benefit
↳ similarly compact size to serialized data → 5x faster access

Spark → interactive → multi stage jobs

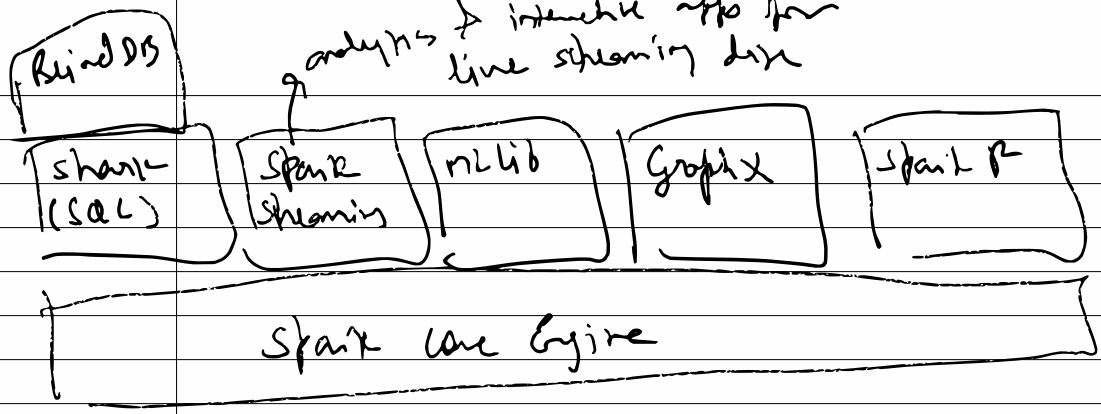
another → stream processing
↳ track and update state in memory as events arrive
↳ large scale reporting, click analysis, spam filtering etc.

· flatmap() · sum() · reduce by window

- Extend spark to perform streaming computation
- Series of small (~1 s) batch jobs, keeping state in memory as fault tolerant RDD.
  ↳ can process 4 GB/s second on 100 nodes at sub second latency.

Spark & Shark → speed up interactive and complex analytics on hadoop data

↳ easy to run locally, on EC2, on Mesos, YARN.


BlinkDB
analytics & interactive apps for live streaming data
Shark (SQL)
Spark Streaming
MLlib
GraphX
Spark R
Spark Core Engine

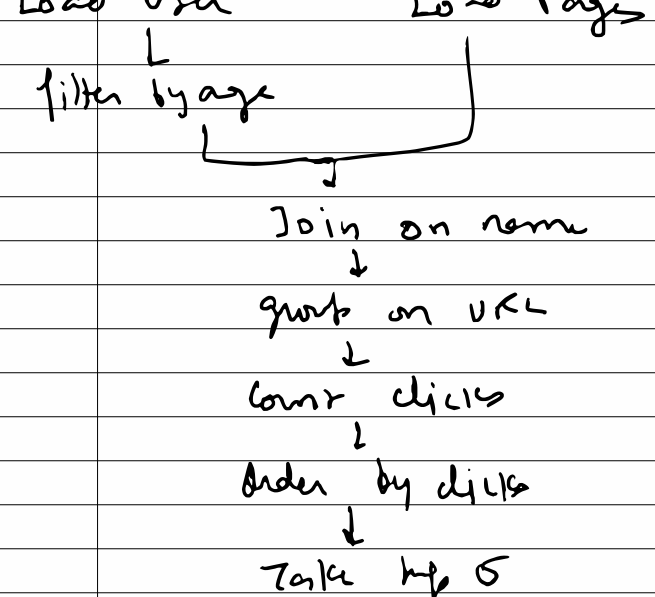
PIG

why

High level language
Small learning curve
Increases productivity

Insulates from complexity of Map Reduce
↳ Job configuration tuning
Mapper Reducer Optimization
Data type use
Job chains

Map Reduce Example
↳ Top 5 most visited by users 18 - 25



Load and filter users by age

```
Users = LOAD 'users' AS (name, age)
users = FILTER users by age >= 18 and age <= 26

load pages

pages = LOAD 'pages' as (users, url)

joined = JOIN users by name, pages by users

grouped = group JOINED By url

summed = FOREACH grouped GENERATE
          group, COUNT (joined) As clicks

sorted = Order summed by clicks desc

top5 = limit sorted 5

Store top5 into '/data/top5sites' ;
```

Hadoop vs Pig

↳ fewer lines of code
↳ less development time
↳ reasonably close to optimal performance

Under the Hood

Pig LATIN programs

↳ Query Parser

↓

Semantic checking

↓

Logical Optimizer

↓

Logical to physical Translator

↓

Physical to M/R translator

↓

Map Reduce Launcher

↓

create a job jar to be submitted to hadoop cluster.

How Pig differs from Map Reduce

Pig provides → join, filter, group by, order by, union etc.

MapReduce → Group by operation directly (shuffle + reduce phase)

Order by → indirectly by the way it implements grouping.

Filter and projection can be implemented trivially in the map phase

other operations (like join) are not provided and must be written by user.

Pig provides some complex, non trivial implementations of these standard data operations

Eg → Records per key is not evenly distributed
↳ some reducers get 10 times more data than other

Pig has join and order by operators that will handle this case and in some cases will rebalance the reducers.

Writing this in map reduce will be time consuming.

In MapR, the data processing inside the map and reduce phase is opaque to the system.

↳ MapR has no time to optimize or check user's code.

Pig can analyze pig latin script written by user and understand the dataflow the user is describing.

↳ Error checking and optimizing

Map Reduce does not have type system

↳ flexibility to user

↳ downside → no code check and error can occur during runtime.

Pig Latin Scripts

```
users = LOAD '/hdfs/user' USING PigStorage('\t')
        as (name, age);

skinnyUsers = Filter users by Age > 150;

skinnyusrnm = Foreach skinnyUsr Generate name, age;

xName = Foreach Users Generate name;
distinctName = Distinct names;
```

Group then Join in Pig Latin

txn = load 'transactions' as (customer, purchase)

grouped = group txns by customer

total = foreach grouped generate group, sum(txn.purchase);

load customer profile

profile = load 'customer_profile' as
(customer, zipcode)

answer = Join total by group, profile by customer

dump answer ;

Using User defined functions

Write code in Java and compile in jar

Pig Latin script
Register myudf.jar

student = load '/student' using PigStorage('\t') as
(name) ;

upper_names = foreach student generate myudf.UPPER(name)

dump upper_names;

Running Pig

↳ Interactive mode

↳ pig -x local → local mode

pig -x mapreduce / map reduce mode on hadoop
or
pig

it will start command line like

> grunt >>

2) Batch mode

write script of pig in pig script and save as id.pig (example)
Running :-
local mode

pig -x local id.pig

MapReduce Mode

pig id.pig
or
pig -x MapReduce id.pig.

multiple line comments

/* .... */

single line comment

--

Pig Latin statements

All statements like input and produce output except LOAD and STORE which read data from and write data to the file system.

Must end with semicolon (;)

- A LOAD statement to read data from filesystem
- A series of transformation statements to process the data
- A dump statement to view results or a Store statements to save the results.

→ DUMP and STORE is required to generate output

→ Pig validates and execute LOAD, FOREACH only while running DUMP/STORE.

Loading Data

Load function

Filter → work with tuples, row of data
For each → columns of data

Group data in single relation

Co group, inner join, outer join → join with two or more relations

Union → merge
Split → partition the content.

Storing intermediate results in hadoop → /tmp
It must be present.

Storage → store final results to the filesystem

Debugging

DUMP	/d	display results
Describe	/de	describe schema
Explain	/e	
Illustrate	/i	step by step execution

/q → quit the grunt shell.

slide 2

Apache Pig

simple to understand data flow language used in the analysis of large data sets.

Pig scripts are automatically converted to MapReduce jobs by pig interpreter.

Simple to understand data flow language for analysis familiar with scripting language

↳ fast iterative language with strong MapReduce compilation engine.

↳ Rich, multivalued, nested operations performed on large data sets.

Resides on user machine

↳ job submitted to cluster & executed on cluster

↳ No need to install anything extra on cluster.

Data type → int, long, double, chararray, byte array

Complex → Bag, tuple, Map.

Pig Data loader

↓

Pig storage
bin storage
binary storage
Text loader
PigDump.

la = load → loader

qp = group by user → group fn → counts

foreach qp generate group, count(la)

Iterating store → storing.

Each statement defines a new dataset, possibly in terms of existing datasets.

Each dataset is immutable.
Datasets can be given aliases to use later.
Use dump

Multiple tuples → returned as bags.

foreach ... generate → another bag

Grouping

grouping → (key, bag)

group key

tuple for every record for that key.

group by makes an output bag containing tuples, containing more bags!

grpd = GROUP log BY user;

in: Bag Of (user, query, time)

out: Bag Of (group, Bag Of (user, query, time), named tuple)

The grouping item is always named 'group'

Thinking like Pig

↳ Data flow language →

HIVE → like SQL → not data flow language

declarative language

Pig eats anything

↓

any data → related, nested, semistructured, unstructured.

↓

support complex data types like bag, tuples that can be nested to form fairly sophisticated data structure.

Hive

A data warehouse infrastructure built on the top of Hadoop for providing data summarization, query and analysis

- HQL
- structure
- Access to different storage

key building principles :-

SQL is a familiar language
Extensibility → Types, functions, formats, scripts
Performance.

intuition

- Make the unstructured data look like tables regardless how it really lay out.
- SQL based query can be directed against these tables
- Generic specify execution plan for this query.

Hive

↳ A data warehousing system to store structured data on Hadoop file system

Provide an easy query these data by execution Hadoop map Reduce plans.

Application

↳ log processing
Text mining
Document indexing
Customer facing BI
Predictive modeling, hypothesis testing.

Hive built on top of HDFS

↓

stores data in HDFS

↓

Compile SQL queries as MapReduce jobs and run the job in the cluster.

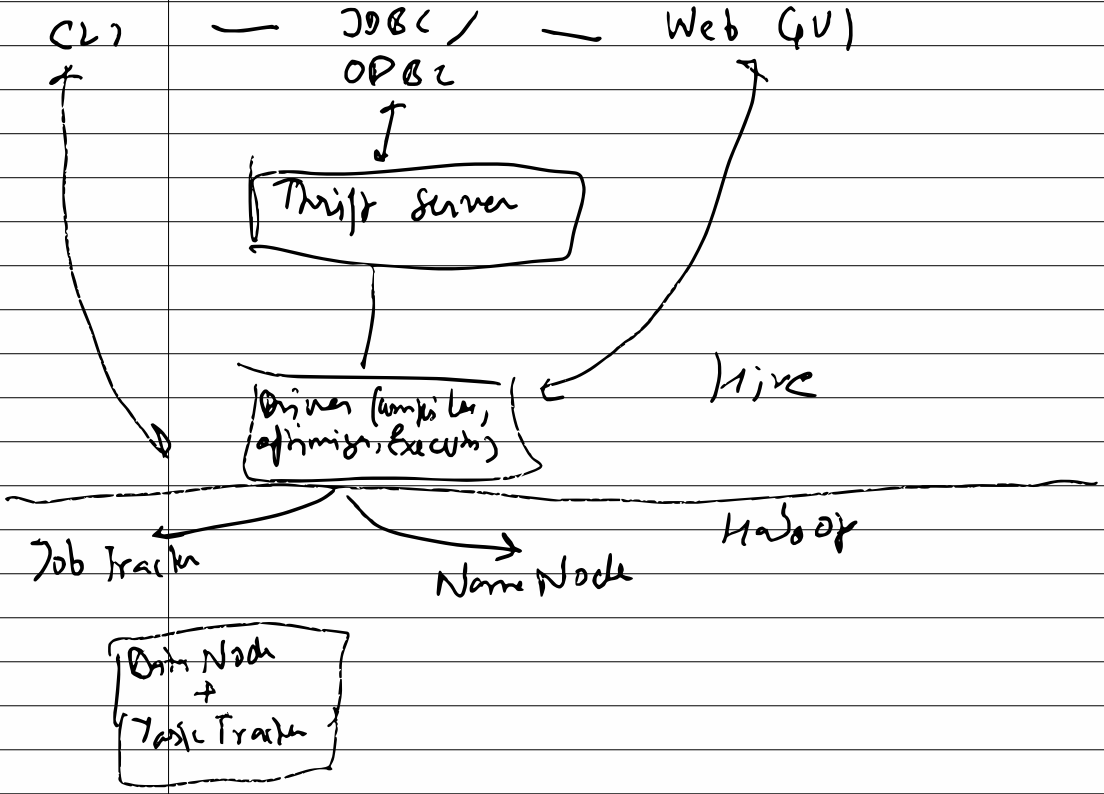
Data Model

↳ Hive structures data into databases
→ tables, columns, rows

Tables
↳ Basic type columns (int, float, boolean)
↳ complex type : List/ map

Partitions
Buckets

Architecture
↓



Architecture

1) External interfaces
   ↳ provides UI & CLI and API like JDBC & ODBC

2) Thrift server
   ↳ Exposes a very simple client API to execute Hive QL statements.

3) Metastore
   ↳ is the system catalog
   All other components of Hive interact with the metastore

4) Driver
   manages the life cycle of a Hive QL statement during compilation, optimization and execution.

5) Compiler → translates into plan.
   ↳ transforms statements into a plan which consists of a DAG of map-reduce jobs.

The driver submits the individual map-reduce jobs from the DAG to the Execution Engine in a topological order.

1) Metastore

↳ system catalog that has metadata about the tables stored in Hive

Database

↳ is the name space for tables

Table

↳ Metadata for table contains list of columns and their types, owner, storage and SerDe information

Partition

↳ Each partition can have its own columns and SerDe and storage information.

---

2) Query compiler

Parser

↳ transforms a query string to a parse tree representation

Semantic Analyzer

↳ transforms the parse tree to a block-based internal query representation

Logical Plan Generator

↳ converts the internal query representation to a logical plan, which consists of a tree of logical operators.

optimizer

↳ performs multiple passes over the logical plan and rewrites it in several ways.

Physical plan generation

↳ Converts logical plan into physical plan consisting of DAG of map-reduce jobs.

Creating table

1) Create table pokes (foo int, bar string) ;

2) Create table pokes (foo int, bar string) partitioned by (ds string)

↙

partition column called ds is virtual column.

Default → tables → text input format
delimiter → ^A (ctrl+A)

browsing through tables

show tables;
↳ list all tables

show tables '.*s';
↳ show tables ending with s.

describe table_name;
↳ shows the list of columns.

Alter table commands

Alter table events rename to 3200 bclay;
Alter table pokes add columns (new_col int)

Replace columns replaces all the existing columns and only changes the table's schema not the data.

Replace columns can also be used to drop columns from table's schema

Alter table invites replace columns (foo int);
it drops the second column

Drop table table_name; → drops the table.

DML operations

Loading data from flat files into Hive:

```
LOAD DATA LOCAL INPATH '/files/k1.txt'
    OVERWRITES INTO TABLE pokes.
```

↓

loads a file that contains two columns separated by ctrl-a into pokes table.

'LOCAL' signifies that the input file is on the local file system.

If LOCAL is omitted then it looks into HDFS.

```
LOAD DATA LOCAL INPATH '/file1.txt'
```

↓ If local is omitted it looks from hadoop

```
OVERWRITE INTO TABLE
pokes partition
(ds = '2008-08-15');
```

If the file is in HDFS it is moved into hive controlled name space.

The operation requires moving so, it is instantaneous.

SQL operations

Select

↓

select a.foo from invites a where

a.ds = '2008-08-15';

It shows result in console.

INSERT command

to save in directory

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

writes in HDFS

OVERWRITE LOCAL DIRECTORY

↓

It will write in local file system.

Performance

1) Group By Operation

Efficient execution plan based on:

Data skew:
- how evenly distributed data across number of physical nodes
- bottleneck vs load balance

Partial Aggregation:
- group the data with the same group by value as soon as possible.
- In memory hash table for mapper.
- Earlier than combiner.

2) Join operation

Traditional map reduce join
Early map side join

↳ very efficient for joining small table with large table.
↳ keep smaller table in memory first
↳ join with a chunk of larger table data each time
↳ space complexity for time complexity.

3) SerDe

↳ How to load data from file to make it look like a table

4) Lazy load

↳ Create the field object when necessary
↳ Reduce the overhead to create unnecessary objects in Hive.
↳ Java is expensive to create objects
↳ increase performance.

Pros

↳ Easy way to process large scale data
↳ Support SQL - based queries
↳ Provide more user defined interfaces to extend
↳ Programmability
↳ efficient execution plans for performance
↳ interoperability with other database tools.

Cons :-

No easy way to append data
files in HDFS are immutable

Hive components

Shell interface : like the MySQL shell

Driver
↳ session handles, fetch, execution

Compiler
↳ Parse, plan, optimize.

Execution Engine
↳ DAG stage
↳ Run map or reduce.

Data Units
↳ Databases
↳ tables
↳ partitions
↳ buckets.

Type System

Primitive Types
- Integers
- Boolean
- Floating point numbers
- String.

Complex types
- structs
- Maps
- Array.

User defined functions

↳ Java code

Registering the class

↳ create function my_lower as 'com.example.hive.udf.Lower' using jar 'hdfs:///path/to/jar';

Using the function

→ select my_lower(title), sum(freq) from titles group by my_lower(title);

Oozie

Workflow scheduler for Hadoop

↓

Designed at yahoo! for their complex search engine workflows.

↓

Now an open source Apache incubator project.

Allows user to create DAG for workflow

↓

Can run parallely / sequentially in hadoop.

→ Can run plain java classes, Pig workflows and interact with HDFS

↳ Nice if you want to delete or move files before a job runs.

→ Major flexibility
↳ start, stop, suspend and re-run jobs.

→ Restart from failure.

Features

- Java client API / command line Interface
- Web service API
- Run periodic jobs.
  - hourly, daily, weekly etc.
- Receive email when job is complete.

How to make workflow

- make job
- make jar
- make workflow.xml with job configurations in it
  - input files
  - output files
  - input readers & writers
  - mappers and reducers
  - job specific arguments

Also need job.properties file. This file defines namenode, job tracker etc.

It also gives the location of the shared jars and other files.

Once ready to run from command line.

Start Node → tells oozie where to start

<start to = "[Name-Node]" />

End Node → signals end of oozie job

<end name = "[Name-Node]" />

Error Node - signals that an error occurred and a message describing the error should be printed out.

<error name = "[Node-name]" />

<message> "[A custom message"]

</message>

</error>

Oozie fork join

Fork → Starts the parallel jobs

join → parallel jobs rejoin at this node. All forked jobs must be completed to continue the workflow.

Can also define decision nodes like switch statements.

for parameters in ${NAMENODE}

1

skip nodes

form failed nodes.

can do either but not both.

Sentiment Analysis