NoSQL

Do you need a relational SQL DB?

1. difficult for large scale data ("planet-wide")
2. fast when de-normalized
3. transactions

4. export from hadoop to mysql?

5. Example Infrastructure
6. Data Source -> Spark Streaming -> MongoDB -> Web FrontEnd
7. "CAP Theorum"
8. Consistency
    1. diff. profile thumbs per-request
9. Availability

10. Partition Tolerance <- non-negotiable for big data

HBase

1. NoSQL database on top of HDFS
2. Based on BigTable (Open Source Version)
3. cf. Google's published papers
4. CRUD operations
5. Region Servers
6. (aprox. Shards = partitions of distributed data)
7. On top of HDFS (itself distributed)
8. HMaster is master over all reigions
    1. Zookeeper Orchestrates
9. Data Model
10. Rows referenced by unique KEY
11. Rows have "COLUMN FAMILIES"
    1. with arbitrary number of columns
    2. helpful for sparse data
12. Rows have "CELL"s -- row/col intersection
    1. versioned by timestamp
13. Example: Google's Problem
    1. com.cnn.www <- lexographic key choice min read
    2. "Contents" column family with single column
        1. so that content cells are versioned
    3. "Anchor" column family
        1. Contains may "Anchor" columns -- eg., millions
        2. One per url linking
14. HBase APIs
    1. REST
        1. Open Port on VM
        2. "starbase" python client
    2. Spark, Hive, Pig, ...
15. Example: User Ratings
    1. RowID = User ID
    2. Column Family = {Rating:, Rating:, Rating:,...}
16. starbase with python
    1. Limited: python is in-memory, not bigdata
    2. .create() creates column family
    3. .drop(), .close(), ...

        4. .batch() (~connection)
            1. .update() (~insert)
17. Interactive shell
    1. create 'users'
    2. list
    3. scan 'users'
    4. disable 'users'; drop 'users'
18. Pig
    1. Big Data
    2. create table upfront, unique keys, etc.
    3. hbase://
    4. USING ...HBaseStorage

Cassandra

1. Like HBase but no master node
2. HBase has HMaster, Zookeeper
3. CQL -- Cassandra Query Language
4. Non-relational
5. No Joins
6. All queries on primary key (or secondary)
7. Shell (CQLSH)
8. Eventually Consistent
9. CAP -- compromise on C
10. Fast Access to Rows
11. Two Systems (OLTPish and OLAPish)
12. Ring Architecture
13. (Region ID ranges in Ring)
14. Gossip Protocol (negotiation between nodes)
15. Nodes share data, gossip to find out which has it
16. DataStax = Spark + Cassandra
17. Cassandra appears as a DataFrame

Hadoop

Background

1. Grouplens.org 100,000 Movie Data Set (5MB)
2. load into HDFS file system
3. u.data, u.item, u.user

Purpose & Ecosystem

1. Ambari -- Admin Management & Installation
2. HDFS -- File System
3. Pig --

MapReduce

1. Example
2. IMDB Movie Data Set
3. Python Libraries
    1. mrjob
    2. simulated or on hadoop
    3. mapper method; reducer method

4. Movies -> Counts
5. Oldest & Hardest to Use
6. Long dev. cycle

Pig

1. PigLatin Scripts
2. PigView in admin console
3. Grunt prompt
4. Data Analysis without writing Mappers & Reducers
5. Can be faster than MapReduce with TEX
6. Spark is preffered, Pig not due to performance
7. Historical: Fixed with TEX
8. On top of MapReduce & TEZ
9. TEZ = Directed Acyclical Graph for Optimizing jobs
10. Pig can go via TEZ or MR
11. Example:
12. "relation" = variable = data set
13. "as" provides schema
    1. schemaless default expectation
14. expects tab delimitated by default
15. FOREACH relation GENERATE new-schema
16. transformation
17. GROUP relation BY field
18. "bags" data
19. DESCRIBE
20. dumps relation structure
21. FILTER relation BY test
22. JOIN relation BY fields BY relation
23. joins relations
24. renames fields, including full path from original
25. ORDER BY
26. IMPORT, DEFINE, REGISTER
    1. interfacing with user-defined functions (JVM)

Spark

1. In-fashion
2. "a fast and general engine for data processing"
3. vs. Pig
4. Rich Ecosystem (eg., Machine Learning)
5. Same Pattern as Hadoop
6. Driver PRogram -> Manager -> Cache
7. Can use hadoop, doesn't need to
8. In-Memory Processing System
9. vs. MapReduce -- File System
10. ~(10 to 100)x F.Sys
11. Built in TEZ-like optimization system
12. One Concept: "Resilient Distributed Dataset (RDD)"
13. "Data set" add in later version, more SQL-like
14. Libraries Included
15. Spark Streaming
    1. Realtime Analysis (vs., Batch)
16. Spark SQL
    1. SQL interface to Spark
    2. Heavy optimization work (>2.0)
17. MLLib

    1. Machine Learning Lib
    2. vs., eg., with MapReduce (hard to do ML)
18. GraphX
    1. (Social) graph analytics
19. Written in Scala
20. Python Libs available
21. Compiles to bytecode -- always faster than python
22. Programming model fits spark more naturally
    1. eg., data transformation via ann. fns.
23. RDDs
    1. SparkContext creates RDDs
    2. Creating
    3. .parrellize([data])
    4. .textFile
        1. hdfs://
    5. HiveContext, Cassandra, ElasticSearch, ..
    6. Transforming
    7. FilterMonadic: .map, .flatMap, .filter
    8. take, top, reduce, count, etc.
    9. Lazy Evaluation
    10. graph of dependent actions built up
    11. nothing happens until "action" is called
    12. Example: Find Lowest Rating
    13. Spark 1 -- RDD Interface
        1. MapReduce-like
        2. .map, .reduceByKey, .mapValues, .sortBy, .take
    14. Ambari / Manager
        1. Spark Log Level (INFO -> ERROR)
    15. Spark SQL
    16. Extends RDD to DataFrame
        1. cf. R, Pandas, ...
    17. Data Frames
        1. from Spark 2.0, lingua franca across libs
        2. DataSet of Row Objects
        3. SQL Queries
        4. Schema
        5. Read/Write to Json, etc.
    18. Create DataFrame
        1. from json
        2. from HiveContext
    19. DataFrame
        1. .sql
        2. .select, .filter, .mean, .groupBy, ...
        3. .rdd() extracts to RDD level
        4. user-defined functions
    20. Example
        1. SparkSession
            1. provides SparkContext
            2. .getOrCreate() recovery
    21. MLLib with Spark
    22. Recommendation Example
        1. ALS alg. -- netflix-prize recommendation alg.
            1. Predict rating for given user with ratings history
        2. from pyspark.ml.recommedation import ALS
        3. read from hdfs://
            1. .cache() in memory
        4. plug in data into model.trasform & predict

Hive

1. SQL on top of MapReduce & TEZ
2. builds on existing SQL knowledge
3. interactive prompt
4. Easy OLAP
5. ie., long-time processing queries
6. Not good for real-time analytics (ie., OLTP)
7. "few minutes" but massive data sets
8. Pretend relational, HDFS is schemaless
9. No inserts/deletes, etc.
10. Pig/Spark more powerful
11. MySQL-like
12. VIEWs
13. Example
    1. HiveView
    2. DROP TABLE
    3. Upload Table, Tab file + Pipe File
    4. CREATE VIEW / SELECT / GROUP BY / JOIN, etc
14. "Schema on Read"
    1. unstructured data -> structured as read
    2. metastore holds schema
    3. CREATE TABLE
    4. ROW FORMAT
    5. FIELDS TERMINATED BY
    6. STORED AS
    7. OVERWRITE INTO TABLE
    8. LOAD DATA, LOAD DATA LOCAL (copy)
    9. No relational DB, just parsing structure
    10. Managed Tables -- Hive Owned, vs., External
    11. DROP'able, etc.
    12. PARTITITONS
    13. sub dirs
    14. significant optimization when relevant