SQL on Hadoop
Technology, Architecture & Innovations

“Remember, the other team is counting on Big Data insights based on previous games. So, kick the ball with your other foot.”
Introduction

Sumit Pal
Independent Consultant Big Data Architecture & Solutions
(Spark, Pig, Hive, Impala, Tableau, Scala, Java, Python, R)
palsumitpal@gmail.com
https://sumitpal.wordpress.com/

Work History
• Microsoft SQLServer Replication Development Team – 1996-97
  • Oracle OLAP Server Development Team – 1997-04
    • LeapFrogRX – Principal Engineer – 2006-13
  • Verizon – Associate Dir – Big Data Analytics – 2013-14
Disclaimer

Call out if you are in doubt / I am wrong – I would like to learn too 😊

Plethora of Technologies out there – I am sure I have missed a few

Not used each and every tool in this presentation 😞 (wish I could)

Not going to show any performance benchmarks across the products
Topics

**Why** - SQL on Hadoop

**Limitations** - SQL on Hadoop, Challenges & Solution

**Architecture** – Batch, Interactive, Stream, Operational

**Innovations** – OLAP, BlinkDB, MapD, SQStream
Why SQL on Hadoop

More data on Hadoop

Popular data querying language

Bridge between BA and Big Data

Integration with BI Tools

Scale that traditional DBMS cannot offer
SQL on Hadoop Goals

• Distributed, Scale Out Architecture

• Avoid Expensive Analytic DBs/Appliances

• Avoid Data Movement: HDFS → Analytic DBs

• High Concurrency

• Low Latency
Why SQL on Hadoop

Analytics: ETL Predictive Reporting BI

SQL

Appliances
Why SQL on Hadoop

Analytics: ETL Predictive Reporting BI

SQL

Appliances

ORACLE EXADATA IBM Netezza TERADATA EMC HP Vertica Greenplum

Hadoop

Cloudera Hortonworks Amazon Web Services MA PR
Why SQL on Hadoop

Analytics: ETL  Predictive  Reporting  BI

- 10x-100x Data
- 1/10 HW $cost
- Open Platform

Appliances

COSTLY SCALE UP $$$$ VS. COST-EFFECTIVE SCALE OUT $
SQL in Hadoop Landscape
Problems of Initial SQL on Hadoop

Map Reduce & HDFS – meant to solve Batch Oriented Data

Map Reduce is high-latency

Map Reduce Not designed for long Data Pipelines
(Complex SQL is inefficiently expressed as many MR stages)

Disk IO between Map & Reduce do lot of Shuffling & Sort

HDFS is WORM – How to Support ACID – Changed in HDFS 2.0
Approaches to solve the challenges
Approaches to solve the challenges
Approaches to solve the challenges
How to Build SQL Solutions on Hadoop
Solve the Latency Challenges

Storage layer optimizations
Data retrieval - Ensure Data locality, Optimal Storage Layout / Formats

Indexing (JethroData)

File Formats – Avro, ORC, Parquet, Sequence Files
Choosing the optimal file format in Hadoop is one of the most essential drivers of functionality and query performance

Data Compression (Reduce IO) – GZIP, BZIP2, LZO, Snappy, LZ4
Workloads are IO Bound – Reduce IO – Compression Algorithms

Compression has a gotcha – Must be Splittable for Hadoop
Tradeoff – Storage & Network Bandwidth / CPU
Analytic Types

- Complex Event Processing
- In-Memory DB
- OLTP Reporting

Factors:
- Real-Time
- Operational Data Volumes
- Lag Time
- Big Data Volumes
Batch SQL on Hadoop

• Uses **Map Reduce** in the background

• Primarily for **large ETL jobs** for batch workloads

• **Extensibility** (with UDF/UDAF/UDTF)

• **Loose-Coupling** with its InputFormats and **Ser/De**

• **Limited** support for support of **Unstructured** Data – JSON

• **Not** designed for **OLTP / Real-time**
Batch SQL on Hadoop
SQL to Map-Reduce

```sql
select f1, f2 from relation where f1 > 500
```

```java
map (k, rec) {
    if (rec.f1 > 500) {
        rec1 = <rec.f1, rec.f2>
        collect (k, rec1)
    }
}
```

```sql
select f3, sum(f1), avg(f2) from relation where f1 > 500 groupby f3
```

```java
map (k, rec) {
    if (rec.f1 > 500) {
        rec1 = <rec.f1, rec.f2, rec.f3>
        collect (rec.f3, rec1)
    }
}
reduce(v, list<rec1>) {
    sum := 0
    avg := 0
    for each rec1 in list {
        sum += rec1.f1
        avg += rec1.f2
    }
    avg := avg / size(list)
    rec2 = <rec1.f3, sum, avg>
    store (v, rec2)
}
Interactive SQL
Optimizations – Hive ~ Interactive SQL

**Partitioning & Bucketing** - Bucketing takes care of Data Skew

**Vectorization**

Reduces the CPU usage, for query scans, filters, aggregates, and joins.

Processing a block of 1024 rows at a time

Each column is stored as a Vector

Uses few instructions and fewer clock cycles - processor pipeline and caching

Pipelines data - Execution Stages instead of temporary intermediate files

Reduce Startup, Scheduling and other overheads of MapReduce
Optimizations – Hive ~ Interactive SQL

Map – Reduce
Intermediate results in HDFS

Tez
Optimized Pipeline

Tez with LLAP
Resident process on Nodes

Hortonworks
Optimizations – Hive ~ Interactive SQL
LLAP (Live Long and Process) : Daemon process

Caching & Data Reuse Queries - **Compressed Columnar In Memory** (off-heap)

High Throughput IO - **Asynchronous Elevator Algorithm**

**Reduce Startup** - JIT Compiler to have more time to optimize
Cost based query optimizer (HIVE-5775)

Existing optimizations in Hive are about **Minimizing Shuffling**

Previous Versions of Hive - Onus is on user to Submit Query to Hive - right join order

**Calcite/Optiq**

Open Source CBO and Query Execution Framework.

More than 50 **Query Optimization Rules** - Rewrite Query Tree

**Efficient Plan Pruner** - Select Cheapest Query Plan
**Architecture:** MPP / Full-Scan
(All SQL-on-Hadoop solutions)

**Query:** list books by author “Stephen King”

**Process:** each librarian is assigned a rack, they then view each book, check if author is “Stephen King”, if so, get book title

**Result:** too slow, costly, unscalable
Client: SELECT day, sum(sales) FROM t1 WHERE prod='abc' GROUP BY day

Performance and resources based on the size of the dataset
Impala Architecture

**MPP** (Massively Parallel Processing) execution engine

**LLVM** (Low Level VM) Compile at Runtime - Low Latency

3 Daemons:

- **impalad** Handles client requests & internal requests
- **statetored** Name service & metadata
- **catalogd** Relays metadata changes to all impalad
Impala Architecture

Fast and Efficient IO manager - handle large data spread across array of hard drives (rotational, or SSD)

Designed to run on modern architecture, recommended chipsets (i.e. Sandy Bridge, Bulldozer), as the LLVM-IR compiler will use newer hardware instructions to help maximize IO throughput

Impala’s execution engine is decoupled from the storage engine, allowing it to plug other storage engines underneath

Impala - streams the results in between the nodes - two types of join algorithms: partitioned and broadcast
Impala Architecture – LLVM & Others

Runtime code generation - to improve execution times

Perform just in-time (JIT) compilation to generate machine code

Produce query specific versions of functions critical to performance

Virtual function calls incur a large performance penalty.
If object type is known, use code generation to replace the virtual function call with inline

HDFS feature short-circuit local reads to bypass the Data Node protocol when reading from local disk
Impala Architecture

Common Hive SQL and interface

- SQL application (Beeswax)
- ODBC

Unified metadata store

- Hive metastore
- HDFS Namenode
- Impala statestore

Client connects to impalad and sends SQL request via ODBC or Beeswax Thrift API

Query planner turns request into collections of plan fragments

impalad
- Query planner
- Query coordinator
- Query exec engine

HDFS DN  |  HBase RS

impalad
- Query planner
- Query coordinator
- Query exec engine

HDFS DN  |  HBase RS

impalad
- Query planner
- Query coordinator
- Query exec engine

HDFS DN  |  HBase RS

Cloudera
Impala Architecture
Impala Architecture

Intermediate results are streamed between impalad's, and query results are streamed back to the client.
**Impala Architecture**

**Advantage**  
No Data Movement out of the clusters, No SPOF

**DisAdvantage**  
Daemon on the each Data Node  
**Cannot recover from mid-query failures** -- Restart again

**Working set** of a query to **fit in the physical memory** of the cluster

**Reasons for High Performance**

- C++ Instead of Java  
- **Runtime Code Generation**  
- **New Execution Engine** (Not Map Reduce)  
- Optimized Data/File Format – **Parquet (Columnar Compressed)**
Apache Drill - Architecture

Interactive Analysis: HDFS/Cassandra/Mongo

File Formats - XML, JSON, Avro, Protocol Buffers

Drillbits - DataNodes to provide Data Locality

Query Optimization Can Plug Custom optimizers

Correlated Sub Query, Analytics Function

Dynamic Data Schema Discovery
Apache Drill

Support for user-defined functions (UDF)

**Nested data** as a first-class citizen

Drill — **Schema Discovery on the Fly**

- Relational Engines — Schema on Write
- Hive, Impala — Schema on Read
- Apache Drill — **Schema on the Fly** (evolving schema or schema-less)
Apache Drill - Architecture

SQL-on-Everything with Apache Drill

CLI
Tableau, Excel, Qlik, ...
Web/Custom Applications

REST

Apache Drill (1-1000 nodes)

NoSQL
HBase
MongoDB
Kudu

Search
Elasticsearch

Files
NAS (NetApp, etc.)
HDFS

IaaS/PaaS
Amazon S3

Relational
Oracle
MySQL
SQL Server
Apache Drill - Architecture

1. Query comes to Drillbit (JDBC, ODBC, CLI, REST)
2. Drillbit generates execution plan - query optimization & locality
3. Fragments are farmed to individual nodes
4. Result is returned to driving node
Apache Drill – Architecture - Drillbit
Presto

High Performance, **Interactive** Queries

**Distributed SQL** Query Engine Optimized for **Adhoc Analysis**

ANSI SQL, Aggregations, Joins, and Window function
Presto - Optimizations

Java (Careful use of memory & data structures) - **Uses Direct Memory management** - Avoiding Java object Allocations, Heap Memory and GC

Execution Model - Different from MR

**Operators are Vectorized** - CPU efficiency & Instruction Cache Locality

**Dynamically Compiles Query** Operators to Bytecode (JVM optimize, Generate Native Code)
Presto - Architecture

Client (SQL) → Presto coordinator
Coordinator → Parses, Analyzes, Plans query execution
Scheduler → Wires execution pipeline, Assigns work to nodes closest to the data, and monitors progress
Client pulls data from output stage, which pulls data from underlying stages
Presto - Architecture

In Memory

Pipelined Execution model runs multiple stages at once, and streams data from one stage to the next as it becomes available.

Limitations
UDFs are more involved to develop, build, and deploy as compared to Hive and Pig

Size limitation on the join tables and cardinality of unique groups

Lacks the ability to write output back to tables
(Currently query results are streamed to client)
Spark SQL
Spark SQL

Data sources - Text files, JSON, Hive, HDFS, Parquet, RDBMS

Static & Dynamic Schema

Integration with Spark Streaming – Dstreams transformed to structured format and SQL can be executed

Higher level of programming abstraction - DataFrames
Spark SQL
# MPP Vs Batch

<table>
<thead>
<tr>
<th>Design</th>
<th>MPP</th>
<th>Batch</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Executor</td>
<td>Tasks</td>
</tr>
<tr>
<td></td>
<td>• CPU</td>
<td>Number of tasks has completely no relation to the number of executors.</td>
</tr>
<tr>
<td></td>
<td>• Memory</td>
<td>#Tasks = #InputSplits = #HDFS blocks</td>
</tr>
<tr>
<td></td>
<td>• Disk</td>
<td>Tasks assigned to executors in arbitrary order based on availability</td>
</tr>
<tr>
<td></td>
<td>Processing task is bounded to specific executor holding required data shard</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem</th>
<th>MPP</th>
<th>Batch</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Stragglers - MPP would always have a node with a degraded disk array, which would lead to degraded performance for this node</td>
<td>With MPP, you don’t need to put intermediate data on the Disk.</td>
</tr>
<tr>
<td></td>
<td>Speculative execution helps with the degraded nodes - because of shared storage, which is impossible in MPP</td>
<td>Executor processes a task and STREAMS result to the next task</td>
</tr>
<tr>
<td></td>
<td>Solved in Batch</td>
<td>Have no option but to store the intermediate results on the local drives results in High Latency</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Concurrency</th>
<th>MPP</th>
<th>Batch</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Low</td>
<td>High</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Latency</th>
<th>MPP</th>
<th>Batch</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Low</td>
<td>High</td>
</tr>
</tbody>
</table>
JethroData – Indexes in Hadoop

**Architecture:** Index Access
(Only JethroData’s SQL-on-Hadoop)

**Query:** list books by author “Stephen King”

**Process:** go-to Author index, entry of “Stephen King”, get list of books, fetch only these books

**Result:** Fast, minimal resources, scalable
Client: SELECT day, sum(sales) FROM t1 WHERE prod='abc' GROUP BY day

1. Index Access
2. Read data only for require rows

Performance and resources based on the size of the result-set
1. Selective extraction
- Only relevant data for BI
- Incremental updates

2. Load into JethroData
- Create indexes and col files
- **Incremental** loads: up-to-the-5min
- Stored in Hadoop, S3 or other file systems

3. Jethro Query engine
- Runs as an edge-node and communicates via HDFS client
- No MapReduce / Tez / Spark are used

- 1B rows per hour
- 1TB of raw-data takes ~300GB in Jethro
- Stateless, multiple servers can be added on the fly to increase concurrency capacity
- **Fast to write**
  - Indexes are created as data is loaded - no delay before querying can start
  - Incremental loads – index files are appended, not updated. Duplicate allowed
  - No locks, no random read/write

- **Fast to read**
  - Simple: Inverted-list indexes map each column value to a list of rows
  - Fast: **Direct Access** to a value entry

---

**Appendable Index Structure for Fast Incremental Loads**

1. **Index of a Column**
   - Each column value has a list of row IDs stored as a bitmap

2. **New Data Loaded**
   - Instead of in-place updates, index is being appended, allowing repeated values

3. **Background Maintenance**
   - Eventually, index files gets merged in the background, removing repeated values

<table>
<thead>
<tr>
<th>Country</th>
<th>Row IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA</td>
<td>3</td>
</tr>
<tr>
<td>FR</td>
<td>2,9,10</td>
</tr>
<tr>
<td>JP</td>
<td>6,7</td>
</tr>
<tr>
<td>US</td>
<td>1,4,5,8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Country</th>
<th>Row IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA</td>
<td>3</td>
</tr>
<tr>
<td>FR</td>
<td>2,9,10,13,17</td>
</tr>
<tr>
<td>IL</td>
<td>12,15,20</td>
</tr>
<tr>
<td>JP</td>
<td>6,7</td>
</tr>
<tr>
<td>US</td>
<td>1,4,5,8,11,14,16,18,19</td>
</tr>
</tbody>
</table>
**Apache Drill** - Semi-Structured Data JSON/HBase/Files/Directories using SQL **Without** Up Front Schema Definition

**Schema is discovered on the fly** based on the query

Hive Ser-De - create SQL like interface to files of any structure

Hive works with JSON files if - pre-processed to remove carriage returns and records are flattened
SQL on UnStructured Data

```json
{
  "name":"Yin",
  "age":null,
  "address":
  {
    "city":"Columbus",
    "state":"Ohio"
  }
}
```

**Goal**

```sql
SELECT
  name, age,
  address.city, address.state
FROM jsonTable
```

**With UDFs**

```sql
SELECT
  v1.name, v1.age, v2.city, v2.state
FROM jsonTable jt
  LATERAL VIEW json_tuple(
    jt.json, 'name', 'age', 'address')
  v1 as name, age, address
  LATERAL VIEW json_tuple(
    v1.address, 'city', 'state')
  v2 as city, state;
```
Spark-Mongodb is a library that allows the user to read/write data with Spark SQL from/into MongoDB collections.

(Developed by Stratio and distributed under the Apache License)

MongoDB SparkSQL data source implements the Spark Dataframe API
val path = "some.json";
val dataFrame = sqlCtx.jsonFile(path)

Register the DataFrame as a Temporary Table
dataFrame.registerTempTable("json");

JSON organized into the table and ready for SQL
Datatypes are inferred from the data in JSON

sqlCtx.sql("select * from json").collect().foreach(print)
sqlCtx.sql("select Name, Number from json").collect().foreach(println)

val allRec = sqlCtx.sql("select * from json").agg(Map("number" -> "sum"))
allRec.collect.foreach ( println )
IOT - Streaming has become very important

**Store and Process-second DOES NOT WORK** Real-Time

Hadoop unable to offer Latency & Throughput for Real-Time (Telecoms, IOT and Cybersecurity)

Streams - **Infinite Tables** sorted by Time & Processed as Window of Data

**Standing query** that **executes Continuously** over data

**InMemory** Processing / **Lock-free Data Structures**

Stream analytics - Statistical Models, Algorithms on data that arrives continuously, often as an unbounded sequence of instances.
Spark Streaming With SQL - Architecture

Flume Source Consuming Data from Weblogs and further delivering to Memory Channel.

Read and Apply Schema (Dynamic) and finally create Data Frames

Json RDD received from Streams

Data Source

Appserver-1 (/logs/debug.logs)

Appserver-2 (/logs/debug.logs)

Source (Exec) → Memory Channel → Spark Sink

Agent-1

Source (Exec) → Memory Channel → Spark Sink

Agent-2

Spark Sink which Consumes the Messages from Memory Channel

SPARK

SPARK SQL

SQL Context

Data Source

Execute SQL

Data Frame

Register as Temporary Table with User-Defined Name

Data Frame (SQL Execution Results)

Console
Streaming SQL – PipelineDB

Continuous Processing & Data Distillation - Platform for analyzing fluid, growing, changing datasets in Real Time

Runs predefined SQL queries **Continuously** on streams **without** storing raw data

Fundamental abstraction is what is called a **Continuous View**

Continuous views only store output in the database

Output is continuously updated incrementally as new data flows

```sql
CREATE CONTINUOUS VIEW X AS SELECT COUNT(*) FROM Stream
```
Streaming SQL – Architecture - PipelineDB

NOT an ad-hoc Data Warehouse
NOT as a distributed database for storing granular data

Datapoints are discarded after they’ve been read

Probabilistic Data Structures & Algorithms SQL queries that reduce the cardinality of streaming dataset

Ships with operations aren’t commonly exposed to users by most DBs, but are extremely useful for continuous data processing.

Bloom Filter
Count-Min Sketch
Filtered-Space Saving Top-K
HyperLogLog
T-Digest
Streaming SQL – Other Products

- SQLStream
- Parstream
- Druid
- VoltDB
Transactional/Operational SQL on Hadoop

**Workloads** - Operational, Interactive, Non-Interactive, Batch

Vary - **Response Time** & **Volume of Data Processed**

“Operational” is an emerging Hadoop market - least mature

- ACID transactions Support in Hive (HIVE-5317)
- Trafodian
- Phoenix
- Splice Machine

[http://tephra.io/](http://tephra.io/)
(globally-consistent transactions on top of [Hbase](http://tephra.io/) – Using MVCC)
Trafodian Architecture

HP Labs Enterprise class **SQL-on-Hadoop DBMS engine** - big data

**Transactional / Operational Workloads**

**Extends HBase** - adds support for ACID for low-latency transactions

**Distributed Transaction Management** for distributed transaction across multiple HBase regions

**ANSI SQL** implementation accessible ODBC/JDBC connection

**Relational Schema abstraction** which makes **feel like relational**

Parallel optimizations for both Transactional & Operational
Trafodion innovation

Stores **all columns in a 1 CF** to improve efficiency/speed

**Column Name Encoding to save disk** & reduce messaging

Columns are assigned **data types** when inserting / updating

Extends ACID to transactions can span multiple tables & rows

**Supports Primary Key & Composite Key**

Supports **Secondary Indexes**
Trafodion Architecture
Phoenix - Architecture

Relational layer on top of HBase

Low Latency Query Model & SQL support over HBase API

SQL query compiled it into a series of HBase scans

Orchestrates running of scans to produce regular JDBC resultsets

Metadata is stored in an HBase table and versioned

Pushes Computation to the HBase Region Servers

Coprocessors (Server-Side) Minimize Data Transfer & Prune Data close to source

Uses Native HBase APIs rather than going through the MR framework of HBase

RO to existing HBase tables, RW to NEW HBase Tables created through Phoenix
Phoenix - Architecture

JDBC driver

Dynamic Columns extend schema at runtime

Schema is versioned for free by HBase allowing flashback queries using prior versions of metadata

Query optimizations
• Secondary indexes
• Statistics
• Reverse scans
• Small scans
• Skip scans
Phoenix - Architecture

- Phoenix
  - Query execution engine
- Pig
  - Data Manipulation
- Hive
  - Structured Query
- GraphX
  - Graph analysis framework
- MLLib
  - Data mining
- Sqoop
  - JDBC client
  - RDB Data Collector
- Flume
  - Log Data Collector
  - Distributed Database
- HBase
  - Distributed Database
- YARN (MRv2)
  - Cluster Resource Manager / MapReduce
- Spark
  - Iterative In-Memory Computation
- Zookeeper
  - Coordination
- HDFS 2.0
  - Hadoop Distributed File System
- The Java Virtual Machine
- Hadoop Common JNI
ACID transactions (HIVE-5317)

INSERT INTO tbl SELECT
INSERT INTO tbl VALUES
UPDATE tbl SET ... WHERE ...
DELETE FROM tbl WHERE ...

How is it being done (look at the paper in reference)

Client side merge of the HDFS files and directories
Apache Kylin
Open Source Distributed Analytics Engine, contributed by eBay Inc., provides SQL interface and multi-dimensional analysis (OLAP) on Hadoop supporting extremely large datasets

Atscale.com
Commercial OLAP On Hadoop – with Machine Learning based Cube Aggregate generation techniques
• Extremely Fast OLAP Engine at Scale
• ANSI SQL Interface on Hadoop
• Interactive Query Capability
• MOLAP Cube
• Seamless Integration with BI Tools:

Other Highlights:
- Job Management and Monitoring
- Compression and Encoding Support
- Incremental Refresh of Cubes
- Leverage HBase Coprocessor for query latency
- Approximate Query Capability for distinct Count (HyperLogLog)
- Easy Web interface to manage, build, monitor and query cubes
OLAP on Hadoop - Apache Kylin

- Online Analysis Data Flow
- Offline Data Flow
- Only SQL for End User
- OLAP Cube is transparent to users
Probabilistic SQL on Large Datasets

- Massively Parallel, Approximate Query Engine
- Meaningful approximate results (with error thresholds)
- Creates Offline Samples based on Error Margins & History
- Runs queries on these samples
- Samples are placed as stripes across multiple machines – Disk / Memory

**Time Bound & Error Bound Queries**

*Select avg(sessiontime) from clickstream_table within 1 seconds*

*Select avg(sessiontime) from clickstream_table error .05 and confidence = .95*
SQL Query Engine – MapD

GPU - DB & Visual Analytics Platform
  - MapD Analytical Database
  - MapD Immerse Visualization

Queries Executed in Parallel - 40,000 cores / server (Single Node)

Written in C++ & hooks into CUDA or OpenCL

Caches hot data in GPU

Compiles queries on the fly using LLVM / Vectorizes execution

Highly-optimized kernels for database operations

Executes queries on both CPU & GPU, if working dataset cannot fit in GPU memory
New TPC Benchmarks TPC-DS 2.0

Performance of SQL engines running on big data platforms

**GOALS**

Vendors “cherry picking” parts of the old TPC-DS to give a skewed picture of their SQL engine’s performance

Cater to products that are SQL-based but that have a hard time running stringent rules of TPC-DS

DOES NOT require Compliance to with ACID properties

Elimination of the need to run under enforced constraints (usually against foreign keys and primary keys)
Why SQL on Hadoop is a Bad Idea

Stefan Groschupf - CEO and Chairman of Datameer

References

JethroData – http://www.jethrodata.com/
Impala in Action – Manning – I was one of the reviewers 😊
Hive on Spark - https://issues.apache.org/jira/browse/HIVE-7292
Hive Speed - http://www.slideshare.net/hortonworks/hive-on-spark-is-blazing-fast-or-is-it-final
BlinkDB - http://blinkdb.org/
BlinkDB - http://www.slideshare.net/Hadoop_Summit/t-1205p212agarwalv2
Hive Vectorization - https://issues.apache.org/jira/browse/HIVE-4160
Hive LLAP - http://www.slideshare.net/Hadoop_Summit/llap-longlived-execution-in-hive
( Q & A )
Select Questions from Audience

I will try to optimize the Queries 😊