Big Data technologies and distributed data processing with SQL

Inverted CERN School of Computing 2020

Emil Kleszcz (CERN)

16.03.2020
Table of contents

1. Brief introduction to Big Data and Hadoop ecosystem.
2. Distributed Data processing on Hadoop:
   a. MapReduce
   b. Spark SQL
   c. Presto
3. Comparison of the processing frameworks.
4. An example: Atlas EventIndex project.
Introduction to Big Data

Huge dataset

Strategy to retrieve & store data
What is Big Data?

**Big data (3V)**

- **Volume**
  - Scale of data
  - Large volume: TB, PB, etc.
  - Size, records, transactions, tables, etc.

- **Variety**
  - Different forms of data
  - Multiple data sources
  - Type of data: structured, unstructured, etc.

- **Velocity**
  - Frequency of updates:
    - Batch processing
    - Stream processing
    - Real-time processing
Big Data history & facts

- 2004 - **MapReduce**: Simplified Data Processing on Large Clusters by Google.
- 2005 - **Hadoop** created by Yahoo & built on top of Google’s MapReduce.
- 2008 - Google processes 20PB of data in one day.

- 90% of data created in last 2 years.
- 4.4ZB in 2013, now ~15ZB yearly, expected.
- 44ZB in 2020 (1ZB = 10^21B).
- The whole universe can contain ~10^124 objects (entropy of black holes).
Architecture overview

- Top level abstractions
- Distributed Processing Frameworks
- Cluster Resource Manager
- Distributed File System (DFS)
- Data stores
  - SQL syntax, etc.
  - MapReduce, etc.
  - Resource orchestration

Cluster Node

Cluster Node

Cluster Node

Cluster Node

Cluster Node

Cluster Node
Data models: CAP theorem

- **Availability**
  - Each client can always read and write.
  - The system continues to operate even in the presence of a node failure.

- **Consistency**
  - All clients have always the same view of the data.
  - Atomic commits like across the entire system.

- **Partition Tolerance**
  - The system continues to operate despite the physical network partition failures.

### Models
- **CA (Consistency, Availability)**
  - SQLite
  - SQL Server
  - Oracle
  - MySQL

- **AP (Availability, Partition Tolerance)**
  - CouchDB
  - Cassandra

- **CP (Consistency, Partition Tolerance)**
  - MongoDB
  - Redis
  - Hadoop
  - Apache HBASE

(document-oriented) (column-oriented) (key-value) (all big table-like systems)
Big Data ecosystem

- **HDFS**: Hadoop Distributed File System
- **Presto**: Low latency SQL
- **Spark**: Large scale data processing
- **Sqoop**: Data exchange with RDBMS
- **Pig**: Scripting
- **Hive**: SQL
- **YARN**: Cluster resource manager

**Tools**
- **Kafka**: Data streaming
- **Flume**: Data collector
- **Zookeeper**: Coordination of distributed systems
- **Apache HBase**: NoSQL columnar store
Hadoop ecosystem

- Started at Yahoo in 2006 based on Google File System and MapReduce from 2003-2004
- A framework for large scale data processing
  - Open source
  - Written in Java
  - To be run on a commodity hardware
- 3Vs of Big Data:
  - Data Volume (Terabytes, ... , Zettabytes)
  - Data Variety (Structured, Unstructured)
  - Data Velocity (Batch processing)
Distributed system for data processing

- Split and distribute data across many machines (sharding)
- Storage with multiple data processing interfaces
- Operates at scale by design (shared nothing - scales out)
- Typically on clusters of commodity-type servers/cloud
- Well established in the industry (open source)
- **Distributed data processing**
  - Fast parallel data scanning
  - Profit from **data locality** - high throughput between storage, CPU & Memory
Hadoop Distributed File System (HDFS)

- **HDFS characteristics**
  - **Fault-tolerant**: multiple copies of data, or Erasure Coding (RAID 5/6, XOR-like)
  - **Scalable** - design to deliver high throughputs, sacrificing access latency
  - Files cannot be modified in place (**Write once - Read Many**)  
  - **Permissions** on files and folders like in **POSIX**, also additional ACLs can be set
  - **Minimal data motion** and rebalance

- **HDFS architecture**:
  - Cluster with **master-slave architecture**
    - **Name Node**(s) (1 or more per cluster) - maintains & manages file system metadata (in RAM)
    - **Data Nodes** (many per cluster) - store & manipulate the data (blocks)

- **Ways of accessing and processing data**
  - Can be mounted with Fuse (with fstab entry)
  - Programming bindings: Java, Scala, Python, C++
  - HDFS has web UI where its status can be tracked
    - http://namenode:50070

```
hdfs dfs -ls
hdfs dfs -ls /user
hdfs dfs -du -h /user
hdfs dfs -mkdir newdir
hdfs dfs -put myfile.csv .
```

```hdfs dfs -get myfile.csv .```
#listing home dir
#listing user dir...
#space used
#creating dir
#storing a file on HDFS
#getting a file from HDFS

HDFS architecture

HDFS Client -> Name Node

Name Node -> Secondary Name Node

fsImage

namespace backup

Replication, balancing, Heartbeats etc.

Data Node

local disks

local disks

local disks

local disks

local disks
How HDFS stores the data

1. File to be stored on HDFS of size 1126MB (split into 256MB blocks)

2. Ask Name Node where to put the blocks

3. Blocks with their replicas (by default 3) are distributed across Data Nodes
What to use Hadoop for?

• **Big Data storage** with HDFS and **big data volumes** with MapReduce
• Strong for **batch processing at scale**
  • Data exploration (ad-hoc), reporting, statistics, aggregations, correlation, ML, BI
• **Hadoop is On-Line Analytical Processing** (OLAP)
  • no real-time data but historical or old data moved in batches
• **Write once - read many** (no data modifications allowed only appends)
• **Typical use cases:**
  • Storing and analysing systems’ logs, time series data at big scale
  • Building data warehouses/lakes for structured data
  • Data preparation for Machine Learning (ML)

… and not use Hadoop for:

• **Weak for Online Transaction Processing** system (OLTP)
  • No multi-record transactions
  • No data updates (only appends and overwrites)
  • Typically response time in minutes rather milliseconds
• **Not optimal for systems with complex relational data**
Typical system based on Hadoop ecosystem

1. Data Ingestion
   - 1a. Reprocess the data

2. Analytic processing
   - 2a. Visualise
   - 2b. Low latency store

3. Publish

DATA SOURCE

NameNode
DataNode
DataNode

Shell/Notebook
Graphical UI
Table of contents

1. Brief introduction to Big Data and Hadoop ecosystem.
2. Distributed Data processing on Hadoop:
   a. MapReduce
   b. Spark SQL
   c. Presto
3. Comparison of the processing frameworks.
4. An example: Atlas EventIndex project.
Big Data ecosystem

Kafka
Data streaming

Flume
Data collector

Zookeeper
Coordination of distributed systems

Presto
Low latency SQL

Spark
Large scale data processing

Hive
SQL

Sqoop
Data exchange with RDBMS

HBase
NoSQL columnar store

MapReduce

YARN
Cluster resource manager

HDFS
Hadoop Distributed File System

Emil Kleszcz | Big Data technologies and SQL-like distributed data processing
Hadoop MapReduce

- The first data processing framework for Hadoop
- Programming model for parallel processing of distributed data
  - Executes in parallel user’s Java code
- Optimized on local data access (leverages data-locality)
- Suitable for huge datasets (PBs of data), and batch/offline data processing
- Low level interface

![Diagram of Hadoop MapReduce]

1. Extraction
2. Filtering
3. Transformation
4. Grouping
5. Sorting
6. Aggregating

Data shuffling

Result
“Word Count” example aka. “Hello World”

The overall MapReduce word count process

Input

Splitting

K1, K2, ...

Mapping

List (K, V)

Shuffling

K, List (V)

Reducing

K, sum(List (V))

Final result

List (K,V)

Deer Bear River

Car Car River

Deer Car Bear

Deer, 1
Bear, 1
River, 1

Car, 1
Bear, 1
River, 1

Deer, 1
Car, 1
Bear, 1

Bear, 1
Bear, 1

Car, 1
Car, 1
Car, 1

Deer, 1
Deer, 1

River, 1
River, 1

Bear, 2
Car, 3
Deer, 2
River, 2

List (K,V)
Hadoop MapReduce - weather data forecast

• The problem
  • Question: What happens after two rainy days in the Geneva region?
  • Answer: Monday :-)  
• The goal: Prove if the theory is true or false with MapReduce  
• Solution: Build a histogram of weekdays preceded by 2 or more bad weather days based on meteo data for Geneva.

• The data source (http://rp5.co.uk)  
  • Source:  
    • Last 5 years of weather data taken at GVA airport  
    • CSV format

"Local time in Geneva(airport)";"T";"Po";"P";"Pa";"U";"DD";"El";"ff10";"ff3";"N";"WW";"W1";"W2";"Tn";"Tx";"Cl";"Nh";"H";"Cm";"Ch";"VV";"Td";"RRR";"tR";"E";"Tg";"E";"sss"  
"07.06.2015 05:00"; <other columns>; "State of sky on the whole unchanged. "; <other columns>  
"07.06.2015 04:00" <other columns> ;"",",",",",",",",",",",",",","16.2","",","",":"  
"07.06.2015 02:00"; <other columns> ;"Rain shower(s), slight. "; <other columns>  
"06.06.2015 23:00"; <other columns> ;"Thunderstorm, slight or moderate, without hail, but with rain and/or snow at time of observation. "; <other columns>

• How do we define the bad weather day?  
  • Weather anomalies (col. num. 11) filtered between 8am and 9pm (excl. night time)
Hadoop MapReduce - weather data forecast

Input Data:
Record: Weather report every hour

Reduced data:
Record: Date of good weather preceded by days of bad weather

Reduced data:
Record: Day of a week with counter of occurrences
Weather forecast - 2nd MapReduce

public static class `ByDayMapper` extends `Mapper<LongWritable, Text, IntWritable, IntWritable>` {
    private `IntWritable` rKey = new `IntWritable`();
    private `IntWritable` rValue = new `IntWritable`();
    private `Calendar` c = `Calendar`.getInstance();
    private `SimpleDateFormat` dt = new `SimpleDateFormat`("yyyy.MM.dd");

    @Override
    protected void map(`LongWritable` key, `Text` value, `Context` context)
        throws `Exception` {
        // Splitting the line into columns by tab
        `String[]` split = value.toString().split("\t");
        try {
            // Only 2 columns expected
            if (split.length==2) {
                // Get a day of the week (num.) out of date (1st column)
                c.setTime(dt.parse(split[0]));
                rKey.set(c.get(`Calendar.DAY_OF_WEEK`));
                // Value is optional for our case
                rValue.set(1);

                // Emit kv for good weather day if preceded by 2>= bad days
                if (Integer.parseInt(split[1])>=2) {
                    context.write(rKey, rValue);
                }
            }
        } catch (`Exception` e) { // ...}
    }
}

public static class `ByDayReducer<KEY>` extends `Reducer<KEY, IntWritable, KEY, LongWritable>` {
    private `LongWritable` result = new `LongWritable`();
    public void reduce(`KEY` key, `Iterable<IntWritable>` values,
            `Context` context)
        throws `Exception` {
        // Counting all mapped pairs for given days of a week
        `long` sum = 0;
        for (`IntWritable` val : values) {
                ++sum; // or += val.get(); always 1
        }
        result.set(sum);
        // Emit the result
        context.write(key, result);
    }
}

public `int` run(`String[]` args)
        throws `Exception` {
    // Init the job
    `Job` job = `Job`.getInstance(getConf());
    job.setJarByClass(getClass());
    job.setJobName("Aggregating by week days");
    // Setting input/output paths
    `FileInputFormat`.addInputPath(job, new `Path`(args[0]));
    `FileOutputFormat`.setOutputPath(job, new `Path`(args[1]));
    // Setting mapper and reducer class
    job.setMapperClass(`ByDayMapper`.class);
    job.setReducerClass(`ByDayReducer`.class);
    // Setting output types/classes
    job.setOutputKeyClass(`IntWritable`.class);
    job.setOutputValueClass(`IntWritable`.class);
    return job.waitForCompletion(true) ? 0 : 1;
}
Limitations of MapReduce

- **Not interactive**
  - Process of scheduling job takes significant amount of time
    - Negotiation with YARN, sending client code, application master has to setup (start JVM, etc.)
  - Typically separate executor (data processor) for each data unit (e.g. HDFS block)
    - A lot of executors have to be started (JVM & local environment have to be setup), short life-time

- **Complex processing** requires to launch *multiple MR jobs*
  - Only 2 stages per job
  - Intermediate results have to be dumped to HDFS and it takes time

- **Each data processing task has to be implemented by a user**
  - Time consuming process especially for data exploration cases

- **What other approaches exist that are more user friendly?**
Big Data ecosystem

- Kafka: Data streaming
- Flume: Data collector
- Zookeeper: Coordination of distributed systems
- Presto: Low latency SQL
- Spark: Large scale data processing
- YARN: Cluster resource manager
- HDFS: Hadoop Distributed File System
- Sqoop: Data exchange with RDBMS
- Pig: Scripting
- Hive: SQL
- HBase: NoSQL columnar store
- Presto
- MapReduce
- Spark
Spark as the next generation MapReduce

• A framework for performing distributed computations
• Scalable - applicable for processing TBs of data
• User-friendly API
• Supports Java, Scala, Python, R and SQL

• Optimized for complex processing
  • Not using MapReduce
  • Allows complex Directed-Acyclic-Graph (DAG) of stages
  • Staged data kept in memory
  • Long living executors - processing multiple stages and jobs

• Varied APIs: DataFrames, SQL, MLib, Streaming
• Multiple computing resource schedulers supported (YARN, Mesos, Kubernetes)
• Multiple deployment modes on Hadoop – local, and cluster on YARN
• Multiple data sources: HDFS, HBase, S3, JDBC...
• Many integrations available such as notebooks
import scala.math.random

val slices = 3  // num of parallel executors
val n = 100000 * slices
val rdd = sc.parallelize(1 to n, slices)
val sample = rdd.map { i =>
  val x = random
  val y = random
  // Check if inside the circle
  if (x*x + y*y < 1) 1 else 0
}
val count = sample.reduce(_ + _)
// Geometric probability of a point inside the square to lie inside the circle
println("Pi is roughly " + 4.0 * count / n)
SQL for the Big Data processing

- SQL is a well-defined language standard that exists since 1970s
  - Everyone is familiar with
  - Minimizes the learning curve of using different data processing tools
- It's a syntax that is converted to the natively optimised code
  - It's just a way of expressing what you want to get and not how you want to get it
- Reduces the amount of code users need to write
- Allows performance optimizations transparent to the users
  - SQL planner and query optimizer
- Opens the door for leveraging & integrating lots of existing tooling
- Structured data are easy to understand and maintain

```sql
UPDATE country
SET population = population + 1
WHERE name = 'USA';
```

```sql
select count(*) from phoenix_hadoop3.aei.sevents;
select * from AEI.EVENTS limit 10;
select * from AEI.EVENTS where EVENTNUMBER=852298541;
```
SQL on HDFS needs Metastore

- Problem: **SQL needs tables but on HDFS we have only directories & files**
- Hive Metastore is a relational database repository containing metadata about objects created
- Contains:
  - **Table definitions** (column names, data types, comments)
  - **Data locations** (partitions)
- Acts as a central schema repository
- Can be used by other access tools such as Spark, Presto, MapReduce etc.
- Default DB configuration is for **PostgreSQL**
- Supports multiple file formats:
  - ORC, Parquet, Text file, ...
- **Tables can be partitioned**
  - each partition is a single HDFS directory

In practice - 3 steps:
- Create your own **Hive Metastore - database as a container for tables**
- Define a table on top of your HDFS data
- Run queries on tables with Spark, etc.
Spark SQL module

- Module for structured data processing
- There are two ways to run Spark SQL:
  - Spark SQL CLI (./bin/spark-sql) (easy to use SQL)
  - or DataFrame API with JDBC/Thrift Server (requires
- Spark SQL CLI
  - Convenient tool to run the Hive Metastore service in local mode and
    execute queries input from the command line :-)
  - cannot talk to the Thrift JDBC server :-(
- Limitation: Natively the data can only be read from Hive Metastore
  (using SparkSession)
  - For other databases one need to use JDBC protocol and the Thrift server

Mixing SQL queries with Spark programs

# Apply functions to results of SQL queries
results = spark.sql("SELECT * FROM my_table")
names = results.map(lambda p: p.column_name)

Uniform data access: querying and joining different data sources

# Defining dataframe with schema from parquet files stored on hdfs
val df = spark.read.parquet("/user/ekleszcz/datasets/")

# Counting the number of pre-filtered rows with DF API
df.filter("l1trigchainstap".contains("L1_TAU4")).count

# Counting the number of pre-filtered rows with SQL
df.registerTempTable("my_table")
spark.sql("SELECT count(*) FROM my_table where l1trigchainstap like '%L1_TAU40%'").show
Spark SQL - weather example

Read weather data from csv

\[
\text{val data} = \text{spark.read.format("csv")}.
\text{option("sep", ";");}
\text{option("inferSchema", "true");}
\text{option("header", "true");}
\text{load("data/*")}
\]

Create a temporary table

\[
data.registerTempTable("weatherTable")
\]

Query to compute sunny days after two rainy days

\[
\text{sql("}
\text{with source as (select [...] as time, ww as weather from weatherTable),}
\text{weather as (select time,[...] then 0 else 1 end bad_wather from source where hour(time) between 8 and 20),}
\text{bad_days as (select [...] as time, sum(bad_wather) bad from weather [...]},
\text{checked as (select time, bad, lag(bad,1) over (order by time) bad1, [...] bad2 from bad_days)
select [...] as day_of_a_week, count(*) from checked where bad=0 and bad1>0 and bad2>0 [...]}
"
\text{)}
\]

Mon | Tue | Wed | Thu | Fri | Sat | Sun
--- | --- | --- | --- | --- | --- | ---
Days count

?
Running Spark in Jupyter Notebook

- Service for Web based ANalysis (SWAN) platform for interactive data analysis in the cloud developed @ CERN
- SWAN Platform: [https://swan.web.cern.ch/](https://swan.web.cern.ch/)
- Exercise to run on the workshop, Jupyter Notebook: [http://cern.ch/go/X6Kj](http://cern.ch/go/X6Kj)
Big Data ecosystem

HDFS
Hadoop Distributed File System

Spark
Large scale data processing

Presto
Low latency SQL

YARN
Cluster resource manager

Kafka
Data streaming

Flume
Data collector

Zookeeper
Coordination of distributed systems

Pig
Scripting

Hive
SQL

Apache HBase
NoSQL columnar store

MapReduce

Sqoop
Data exchange with RDBMS
Presto - Massively Parallel Processing (MPP)

- MPP SQL (on-anything) query engine for multiple datastores/databases initiated by Facebook
- Characteristics:
  - Low latency SQL queries (query start up time <100ms)
  - Typically much faster than Spark and MapReduce
  - Executing daemons/workers are up all the time
  - Platform agnostic, can run anywhere
  - doesn’t use Yarn
  - Typically run on top of the Hadoop cluster
- Main benefits:
  - Offers easy-to-use SQL (no other integration/code required),
  - Multiple connectors to data storages with one endpoint
  - Connectors are pluggable (ad-hoc adding)
  - Low latency thanks to:
    - Cost-Based Query Optimizer
    - Leveraging data locality in Hadoop

Similar frameworks:
- Apache Impala
- Apache Drill
- Hive LLAP
Presto Architecture

1. Application, Presto CLI, Notebooks

2. Receives a query from the client, analyzes, parses, plans, and schedules to the workers

Coordinator

Metadata API

Data Location API

Parser / Analyzer → Planner → Scheduler

Data Stream API

Worker

3. Executes schedules tasks, sends the final result to the client

4. Data source plugins
Presto for Hadoop in practice

- **Dedicated connector for HDFS**
  - **Only** the data mapped via Hive Metastore tables can be accessed from HDFS
  - Existing HDFS folders can be easily mapped to Hive tables (if schema is coherent)

- **Each connector can have multiple instances (called catalogs)**
  - Multiple hives (Hadoop clusters) can be accessed simultaneously
  - `select * from hive_hadalytic.my_schema.my_table`

- **Interfaces:**
  - Presto shell (CLI)
  - JDBC/ODBC for binding with applications
Big Data scale-out database example with Presto

- **Ingest**
  - **Real time ingest**
  - **Batch ingest (lower latency)**

- **Access**
  - **OLTP**
  - **Indexed data**
  - **Fast data extraction**
  - **Average analytics performance**

- **OLAP**
  - **Columnar**
  - **Fast analytics**
  - **Average extraction time**

**Big Data Client**

- JDBC Client

**Ingest**
- **High velocity data bus**
Presto SQL - weather forecast example

Actual query to compute sunny days after two rainy days in Geneva

```sql
weather as (select time, case when weather in (',', '') then 0 else 1 end bad_weather
from interesting_data where extract (hour from time) between 8 and 20),
bad_days as(select date_trunc('day',time) as time, sum(bad_weather) bad from weather [...] ),
checked as (select time,bad,lag(bad,1) over (order by time) bad1, [...] bad2 from bad_days),
select date_format(time,'%W') as day_name, count(*) from checked
where bad=0 and bad1>0 and bad2>0 group by [...] ;
```

Actual query to compute sunny days after two rainy days in Geneva.
Table of contents

1. Brief introduction to Big Data and Hadoop ecosystem.
2. Distributed Data processing on Hadoop:
   a. MapReduce
   b. Spark SQL
   c. Presto
3. Comparison of the processing frameworks.
4. An example: Atlas EventIndex project.
Evaluation: Presto vs Spark SQL

- **Spark** designed for *batch or long running ETL jobs*
- **Presto** targets on *interactive queries with low-latency startup*
  - Cluster starts on-demand
  - Declared resources that are available all the time

- **3rd party plugins (connectors) in Spark SQL**
  - Available as jars in Maven Repository
  - May not be stable in future releases of Spark
  - Risky to run in production

- **Summarizing:**
  - The purpose of both projects is slightly different
  - Both are important and can be complementary
Comparison of the 3 frameworks

- **MapReduce**
  - Requires complex coding of jobs - *time consuming*,
  - Intended mainly for **batch processing**

- **Spark SQL**
  - Covers most of the use cases (ETL, advanced batch processing) except low latency queries
  - Only one native connector to the Hive Metastore
  - The data from other data sources cannot be queried without writing some spark code

- **Presto**
  - For interactive data access (low latency queries)
  - Used for:
    - Generation of reports from big datasets
    - Complex analytics with multiple data sources
    - Querying: OLAP (HDFS/Parquet) and OLTP (HBase+Phoenix) systems
Table of contents

1. Brief introduction to Big Data and Hadoop ecosystem.
2. Distributed Data processing on Hadoop:
   a. MapReduce
   b. Spark SQL
   c. Presto
3. Comparison of the processing frameworks.
4. An example: Atlas EventIndex project.
The Atlas EventIndex

- Catalogue of all collisions in the ATLAS detector
  - Over 185 billion of records, 200TBs of data
  - Current ingestion rates: 5kHz, 60TB/year
  - One record has size of ~1.5kB
  - Each indexed event is stored in a MapFile

**Main use-cases**
- Event picking
- Count or select events based on trigger decisions
- Production completeness and consistency checks (corrupted, missing or duplicated events validation)
- Dataset browsing: finding dataset, generating reports

**EventIndex information**
- Event identifiers:
  - Run and event number
  - Trigger Stream
  - Luminosity block
  - Bunch Crossing ID

**Data Production**
- Collisions data file
- MetaFile

**Data Collection**
- Events metadata extraction
- Grid job
- Object Store

**Data Storage and Query Interface**
- Hadoop
- Mapfiles + HBase
- Analytics Web UI
- Table
- RDBMS
- WLCG
- CERN

Data Production

Collisions data file → MetaFile → Events metadata extraction → Grid job → WLCG → CERN → Data enrichment → Analytics Web UI → Event extraction Web UI → Table → RDBMS

Catalogue of all collisions in the ATLAS detector

- Over 185 billion of records, 200TBs of data
- Current ingestion rates: 5kHz, 60TB/year
- One record has size of ~1.5kB
- Each indexed event is stored in a MapFile

Event identifiers:
- Run and event number
- Trigger Stream
- Luminosity block
- Bunch Crossing ID

Main use-cases:
- Event picking
- Count or select events based on trigger decisions
- Production completeness and consistency checks (corrupted, missing or duplicated events validation)
- Dataset browsing: finding dataset, generating reports
The Atlas EventIndex - new architecture proposal

• Proposed changes:
  • Replacing RDBMS with HBase/Phoenix and **Presto layer for SQL queries**
  • Replacing MapFiles with HBase data storing
  • In the future could be also Object Store replacement with Apache Kafka cluster
References

- [phoenix.apache.org](phoenix.apache.org)
- [https://prestodb.io/blog/2019/08/05/presto-unlimited-mpp-database-at-scale](https://prestodb.io/blog/2019/08/05/presto-unlimited-mpp-database-at-scale)
- A prototype for the evolution of ATLAS EventIndex based on Apache Kudu storage, ref. [https://www.epj-conferences.org/articles/epjconf/pdf/2019/19/epjconf_chep2018_04057.pdf](https://www.epj-conferences.org/articles/epjconf/pdf/2019/19/epjconf_chep2018_04057.pdf)
Thank you for your attention!
The Atlas EventIndex - performance comparison

- **Data ingestion speed** improved by rate of 2-10x.
- **Storage efficiency** improved by factor of 10
  - by using HBase + snappy compression on the data
- **Random data access** using HBase
  - typical random data lookup speed is below 500 ms
  - for the MapFile-based solution was around 4s
- **Data analytics** - fast and scalable with rate of 300k records per CPU core (300kHz)
- **Updates are possible** and not only appends
- Combining with Phoenix/Presto allows querying data from multiple data sources with SQL
- **Random lookup test is suboptimal for HBase** as a significant amount of time is spent to set up a query before it really gets executed ~200ms
- **Salting improves parallelism** by distributing data (regions) between different servers (regionservers)
The Atlas EventIndex - some queries and data structure

> show tables from phoenix_hadoop3.aei;

datasets
events
sdatasets
sevents

> use phoenix_hadoop3.aei;

> describe sdatasets;

# Typical AEI queries to find GUID of a file in Castor (with the event information)
> select * from datasets where runnumber=280753;

# Find dspid for the run # dspid = < project, runnumber, streamname, processingStep, version >
# Example: dspid = <data15_13TeV, 00281385, physics_Main, deriv, r9264_p3083_p3213>
> select * from events where dspid in (283,170) and eventnumber=4317812;

# To find the reference to the file and more metadata
# The worst scenario (scanning the whole dataset)
> select count(*) from events;