SNAPPY

Efficient state management with Spark 2.0 and scale-out databases

Jags Ramnarayan

Co-founder, CTO, SnappyData

Barzan Mozafari

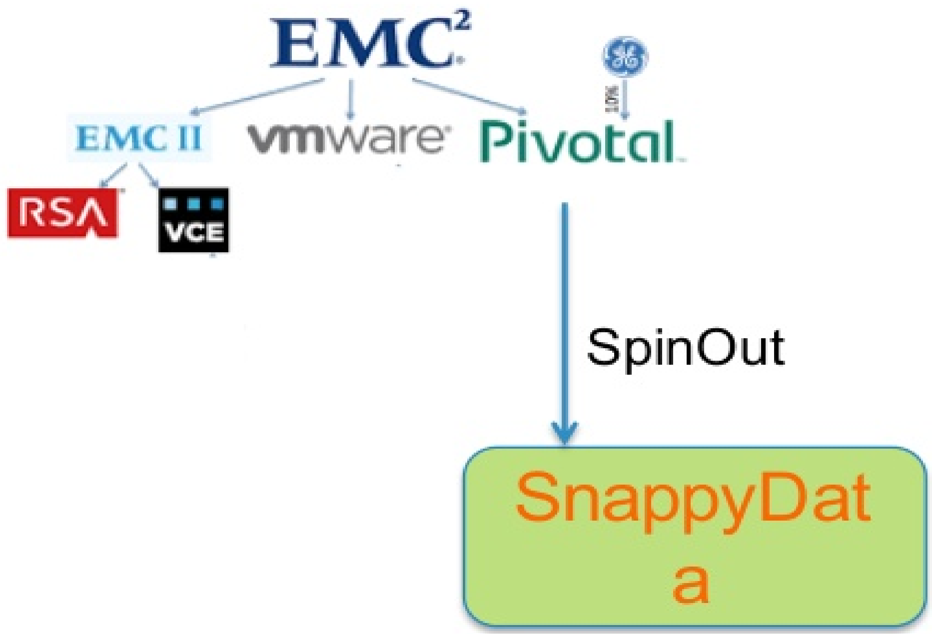
Strategic Advisor @ Snappydata

Assistant Professor,
Univ. of Michigan, Ann Arbor

Spark summit

SPARK SUMMIT 2016
DATA SCIENCE AND ENGINEERING AT SCALE
JUNE 6-8, 2016 SAN FRANCISCO

SnappyData's focus – DB on Spark



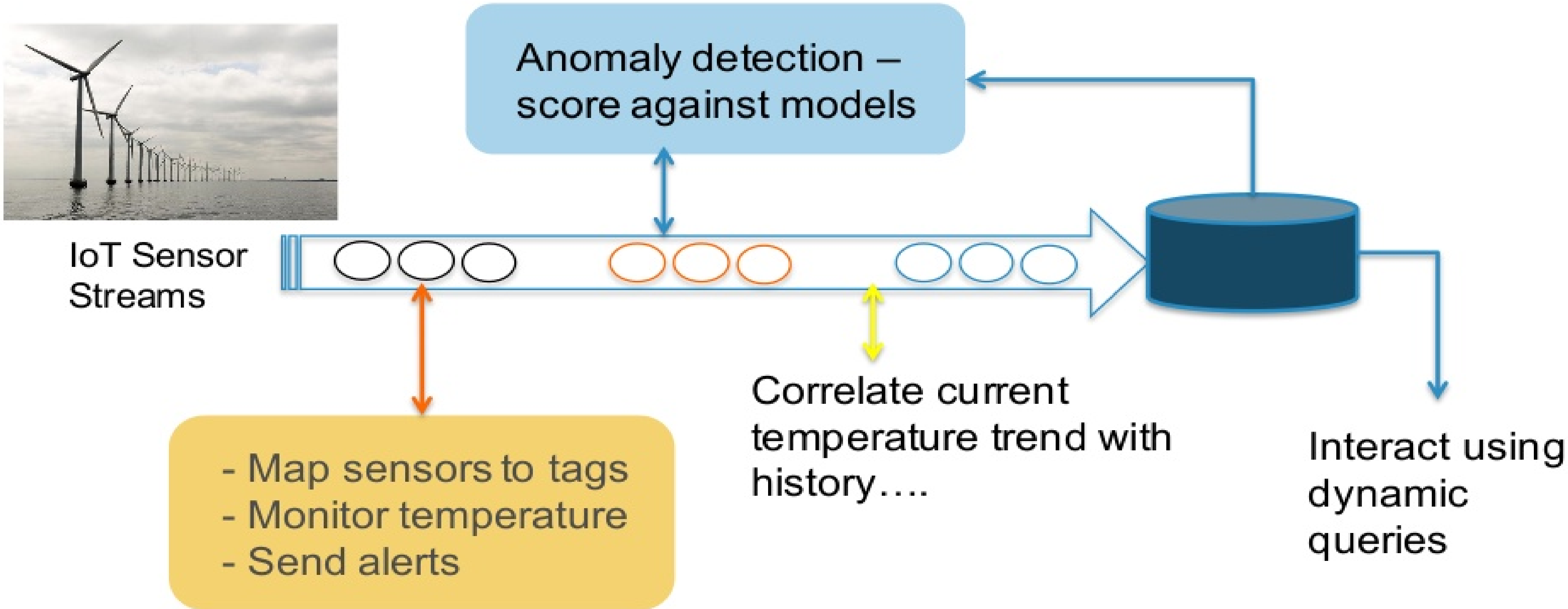
Funded by Pivotal, GE, GTD Capital

- New Spark-based open source project started by Pivotal GemFire founders+engineers
- Decades of in-memory data management experience
- Focus on real-time, operational analytics: Spark inside an OLTP+OLAP database

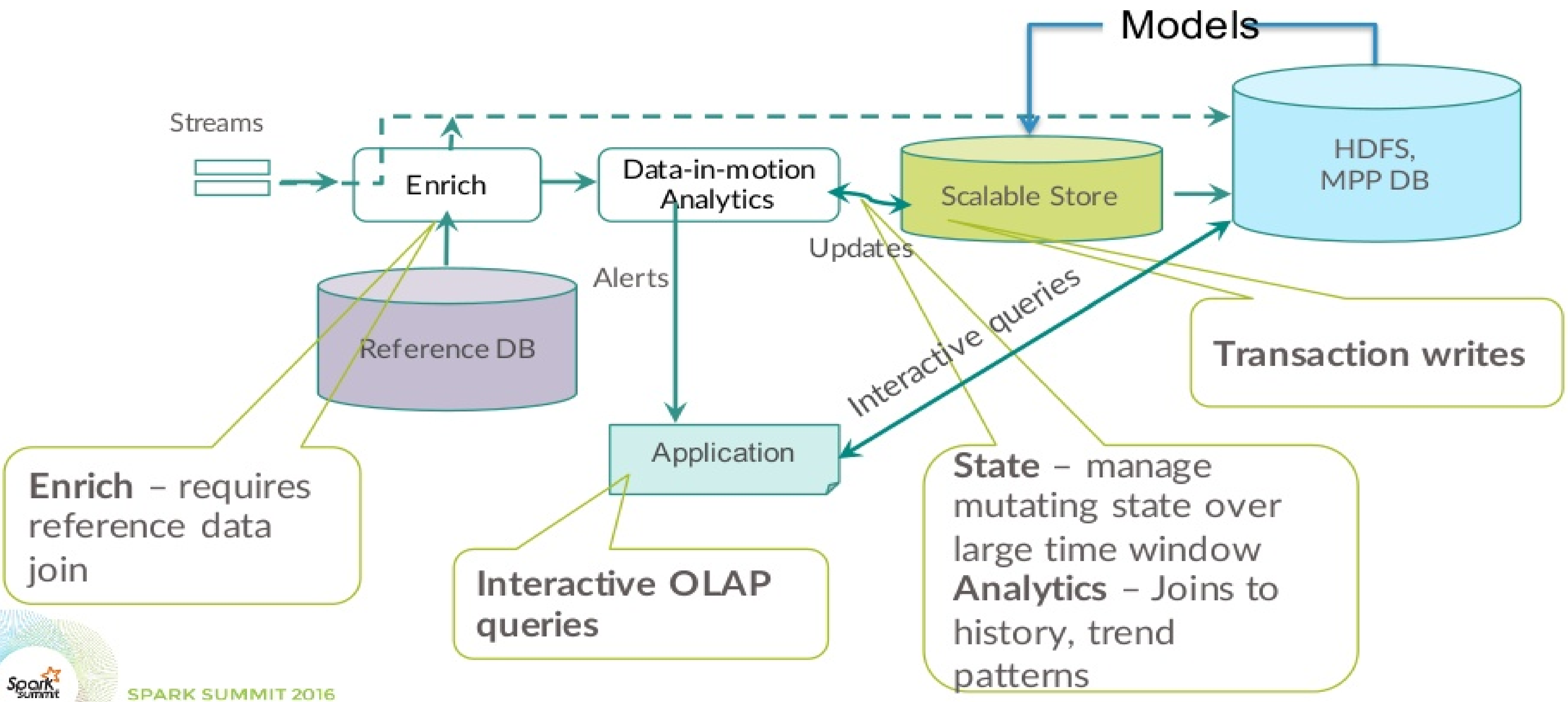
Agenda

- Mixed workloads are important but complex today
- State management challenges with such workloads in spark Apps
- The SnappyData solution
- Approximate query processing
- How do we extend spark for real time, mixed workloads?
- Q&A

Mixed Workloads are increasingly common



Mixed workload Architecture is Complex (Lambda)



Lamba Architecure is Complex

- Complexity: learn and master multiple products → data models, disparate APIs, configs
- Slower
- Wasted resources

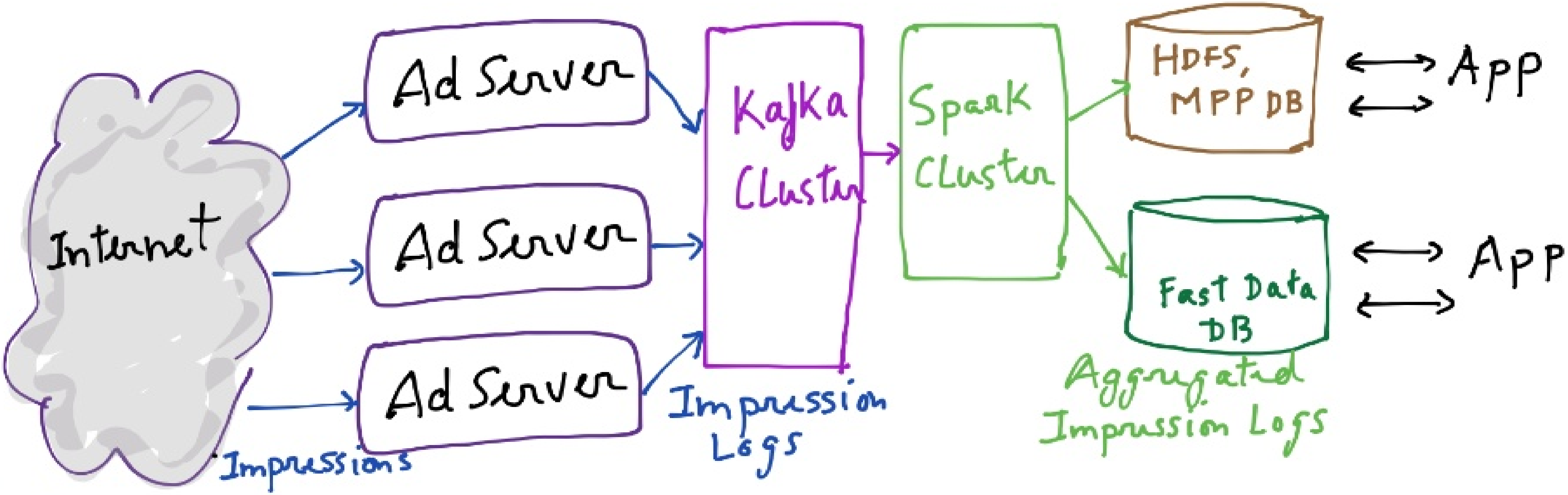
Can we simplify & optimize?

Perhaps a single clustered DB that can manage stream state, transactional data and run OLAP queries?

Deeper Look into Managing State in Spark Applications

Deeper Look – Ad Impression Analytics

Ad Network Architecture – Analyze log impressions in real time



Ad Impression Analytics

To simplify, let's consider that impression logs are in this format:

```
timestamp                 publisher advertiser  website geo bid     cookie
--------------------------------------------------------------------------
2013-01-28 13:21:12      pub1      adv10       abc.com NY  0.0001  1214
2013-01-28 13:21:13      pub1      adv10       abc.com NY  0.0005  1214
2013-01-28 13:21:14      pub2      adv20       xyz.com CA  0.0003  4321
2013-01-28 13:21:15      pub2      adv20       xyz.com CA  0.0001  5675
```

Our goal is to aggregate these logs by publisher and geo, and compute the average bid, the number of impressions and the number of uniques by minute. So the aggregation will look something like:

```
timestamp                 publisher geo  avg_bid imps  uniques
--------------------------------------------------------------------------
2013-01-28 13:21:00      pub1      NY   0.0003  256   104
2013-01-28 13:21:00      pub2      CA   0.0002  121   15
2013-01-28 13:22:00      pub1      NY   0.0001  190   98
2013-01-28 13:22:00      pub2      CA   0.0007  137   19
```


Ref - https://chimpler.wordpress.com/2014/07/01/implementing-a-real-time-data-pipeline-with-spark-streaming/

Bottlenecks in the write path

```
val input :DataFrame= sqlContext.read
    .options(kafkaOptions).format(..)
    .stream("Kafka url")
```

Stream micro batches in parallel from Kafka to each Spark executor

```
val result :DataFrame = input
  .where("geo != 'UnknownGeo'")
  .groupBy(
   window("event-time", "1min"),
   "Publisher", "geo")
  .agg(avg("bid"), ....
```

Filter and collect event for 1 minute. Reduce to 1 event per Publisher, Geo every minute

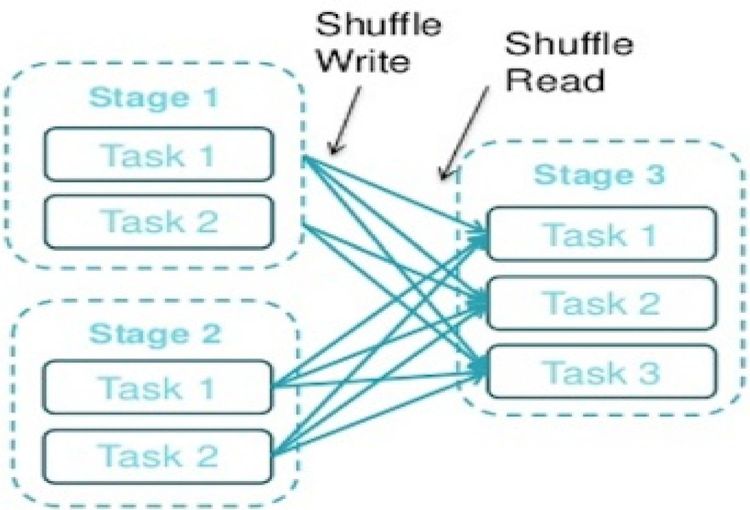
Execute GROUP BY … Expensive Spark Shuffle …

```
val query = result.write.format("My Favorite NoSQLDB")
  .outputMode("append")
  .startStream("dest-path")
```

Shuffle again in DB cluster … data format changes … serialization costs

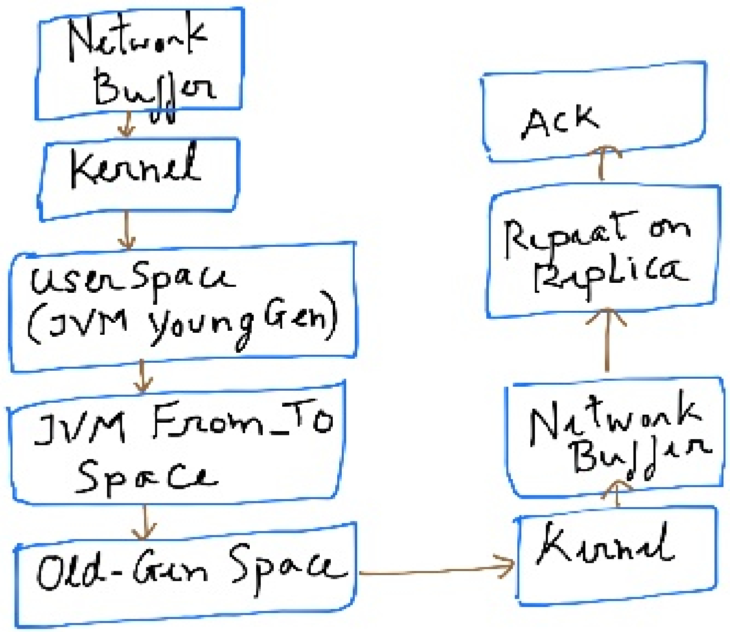
Bottlenecks in the Write Path

Shuffle Costs (Copying, Serialization)

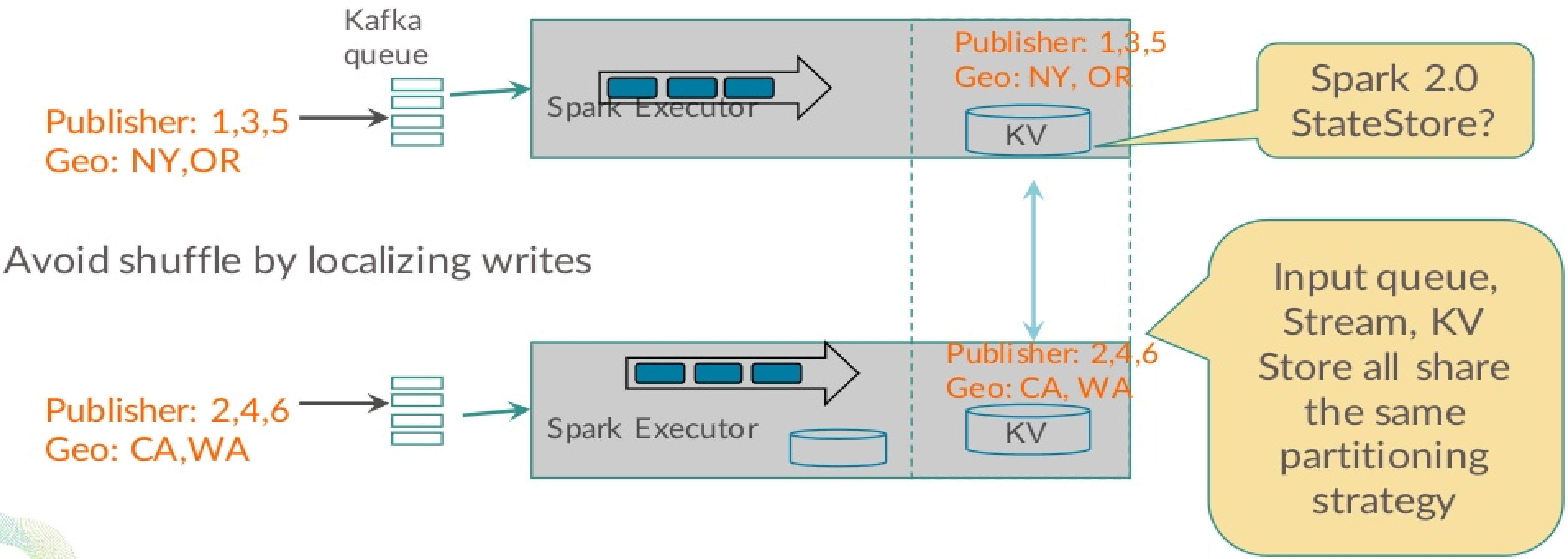


- Aggregations – GroupBy, MapReduce
- Joins with other streams, Reference data

Excessive copying in Java based Scale out stores

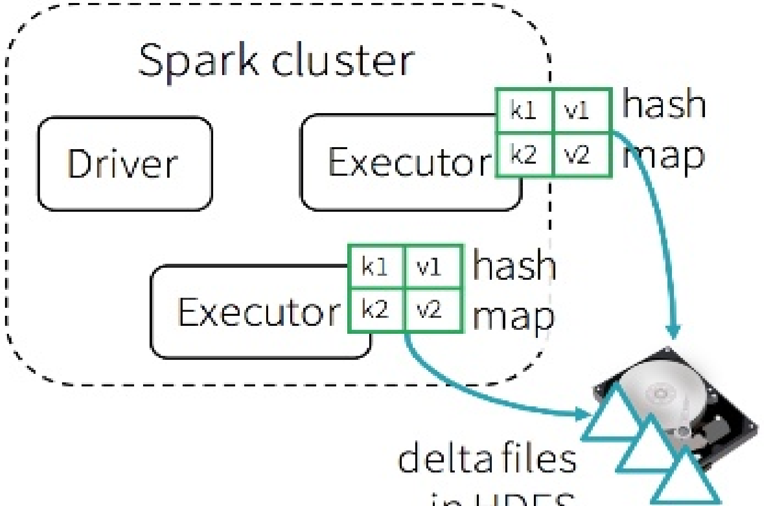


Avoid bottleneck - Localize process with State



New State Management API in Spark 2.0 – KV Store

- Preserve state across streaming aggregations across multiple batches
- Fetch, store KV pairs
- Transactional
- Supports versioning (batch)
- Built in store that persists to HDFS



Source: Databricks presentation

Impedance mismatch with KV stores?

We want to run interactive "scan"-intensive queries:

- `Find total` *`uniques`* `for Ads grouped on geography`
- `Impression trends for Advertisers(group by query)`

Two alternatives: row-stores(all KV Stores) vs. column stores

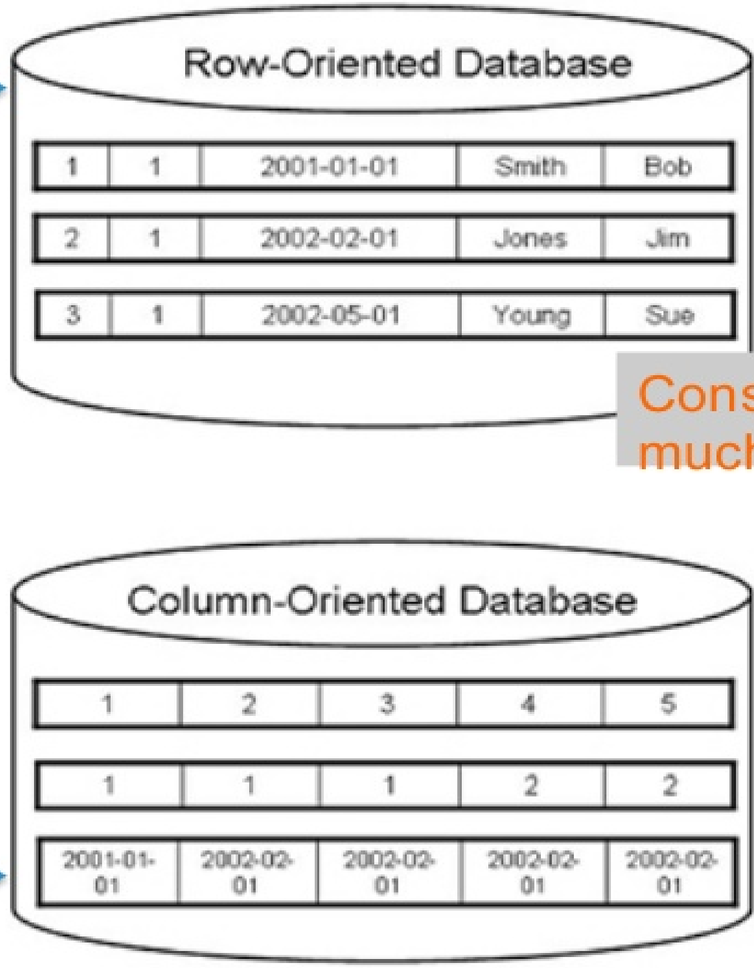
Fast key-based lookup

But, too slow to run aggregations, scan based interactive queries

Emp_no	Dept_id	Hire_date	Emp_ln	Emp_fn
1	1	2001-01-01	Smith	Bob
2	1	2002-02-01	Jones	Jim
3	1	2002-05-01	Young	Sue
4	2	2003-02-01	Stemle	Bill
5	2	1999-06-15	Aurora	Jack
6	3	2000-08-15	Jung	Laura

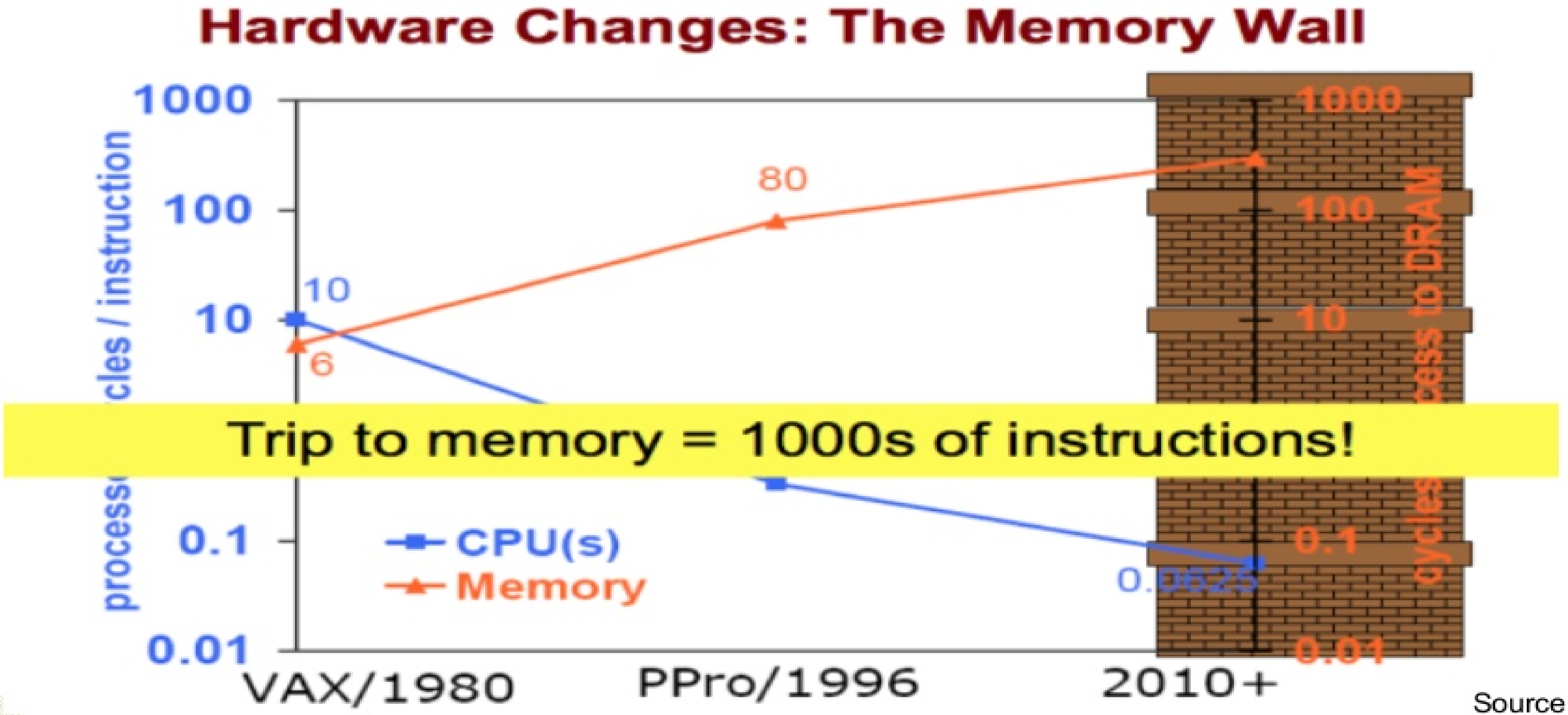
Fast scans, aggregations

Updates and random writes are very difficult



Consume too much memory

Why columnar storage in-memory?

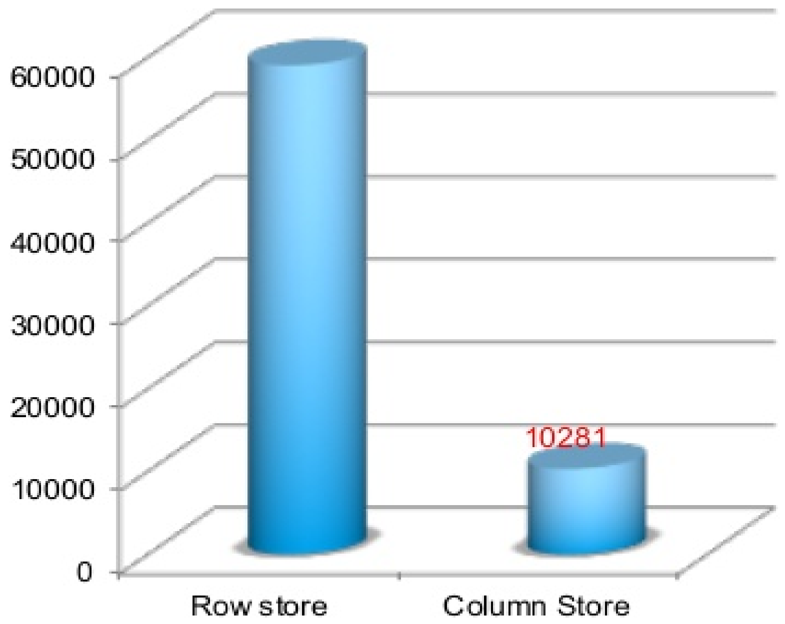


Source: MonetDB

But, are in-memory column-stores fast enough?

AWS c4.2xlarge ; 4 x (8 cores, 15GB)
Column Table: AdImpressions; 450million rows

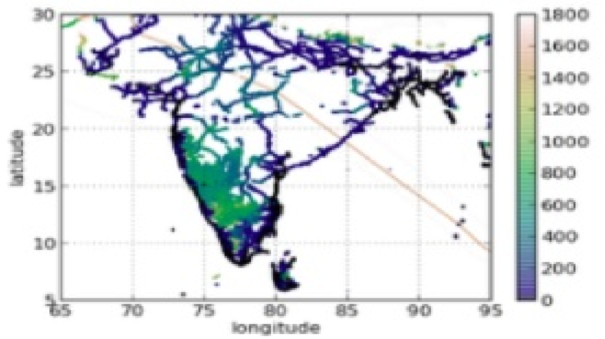
```
select count(*) from adImpressions
    group by geo
    order by count desc
    limit 20;
```



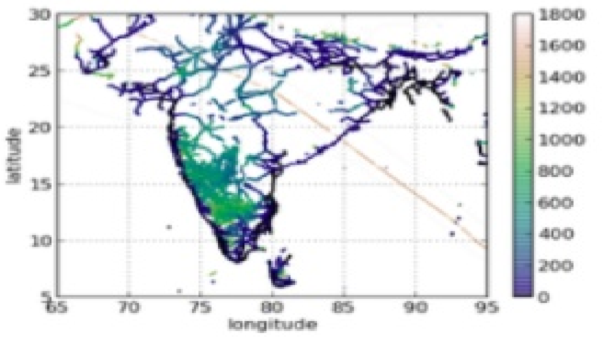
Single User, 10+seconds. Is this Interactive speed?

Use statistical techniques to shrink data!

- Most apps happy to tradeoff 1% accuracy for 200x speedup!
  - Can usually get a 99.9% accurate answer by only looking at a tiny fraction of data!
- Often can make perfectly accurate decisions with imperfect answers!
  - A/B Testing, visualization, ...
- The data itself is usually noisy
  - Processing entire data doesn't necessarily mean exact answers!
- Inference is probabilistic anyway



Original (2 billion points), 71 mins

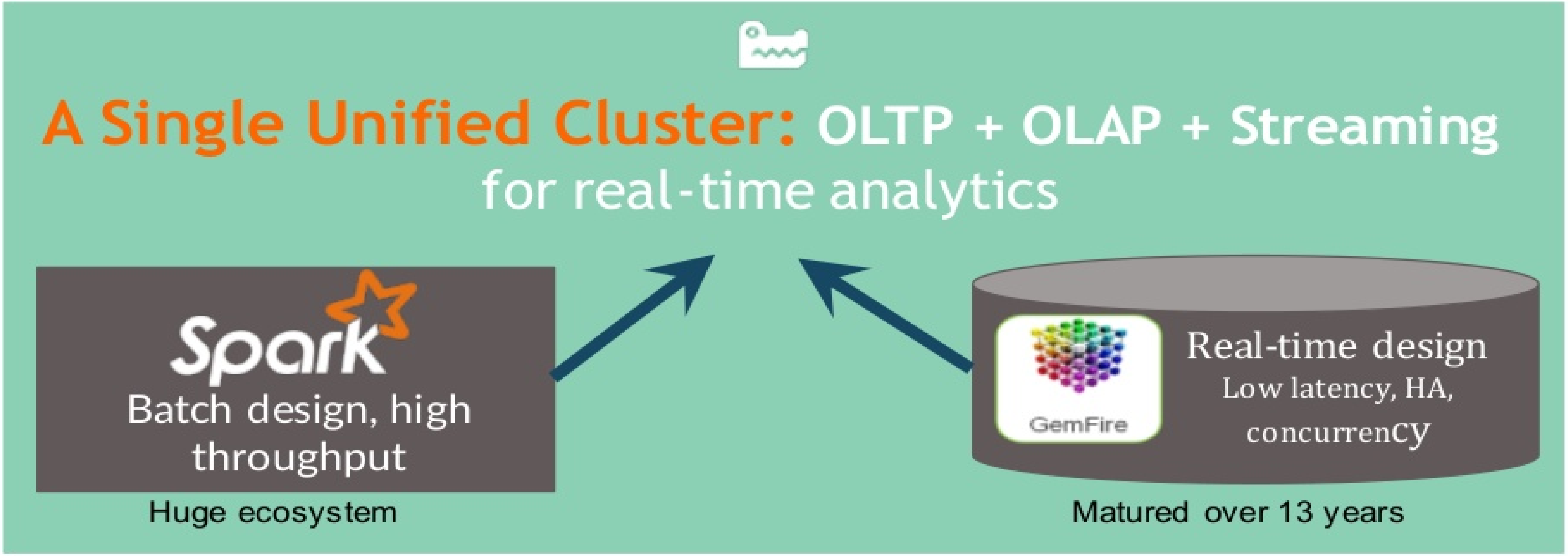


VAS (1M), 3 secs

Our Solution: SnappyData

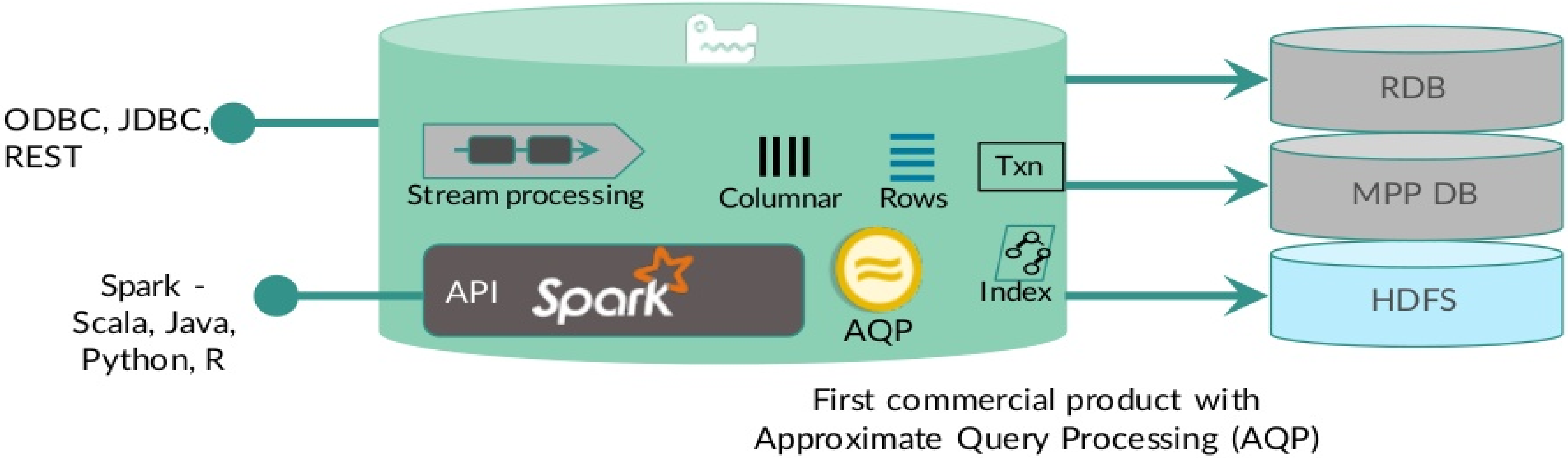
Open Sourced @ https://github.com/SnappyDataInc/snappydata

SnappyData: A New Approach



Vision: Drastically reduce the cost and complexity in modern big data

Unified In-memory DB for Streams, Txn, OLAP queries



Real-time operational Analytics – TBs in memory

Features

- Deeply integrated database for Spark
  - 100% compatible with Spark
  - Extensions for Transactions (updates), SQL stream processing
  - Extensions for High Availability
  - Approximate query processing for interactive OLAP

- OLTP+OLAP Store
  - Replicated and partitioned tables
  - Tables can be Row or Column oriented (in-memory & on-disk)
  - SQL extensions for compatibility with SQL Standard
    - create table, view, indexes, constraints, etc

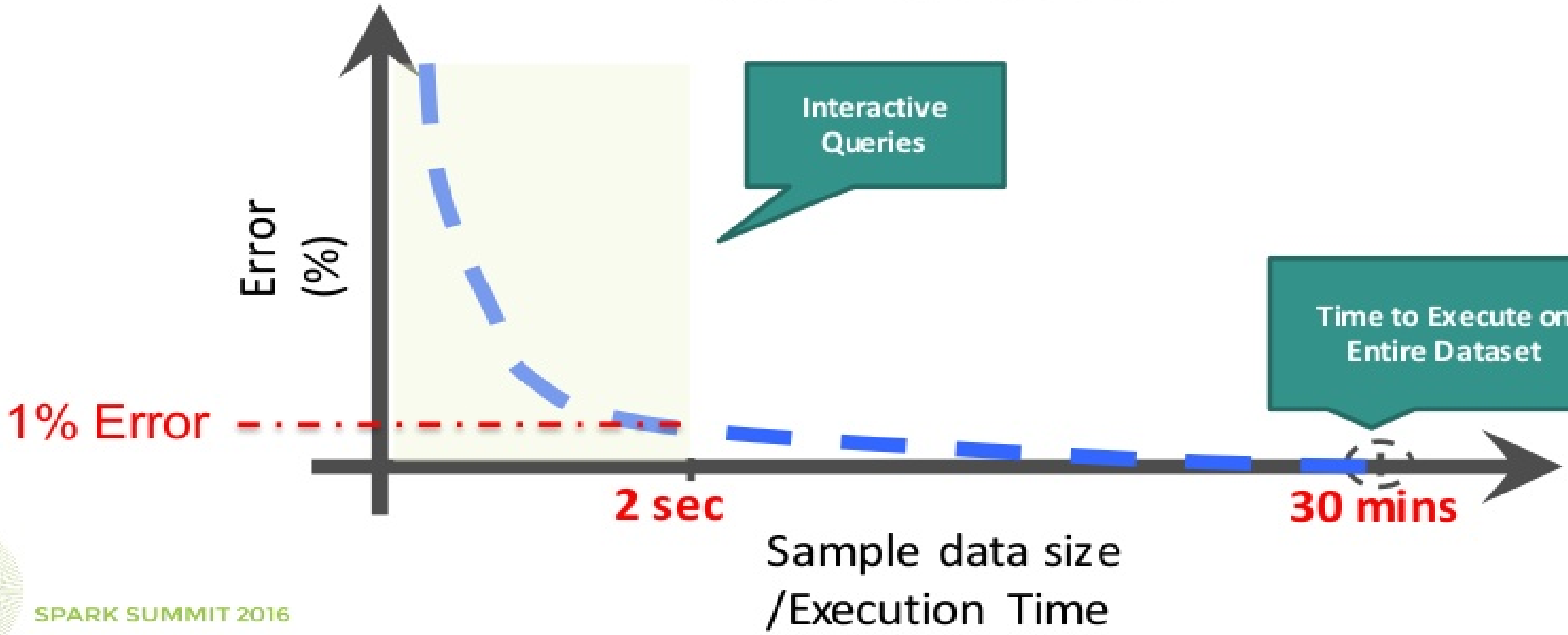
Approximate Query Processing Features

- Uniform random sampling
- Stratified sampling
  - Solutions exist for stored data (BlinkDB)
  - SnappyData works for infinite streams of data too
- Support for synopses
  - Top-K queries, heavy hitters, outliers, ...
- Exponentially decaying windows over time

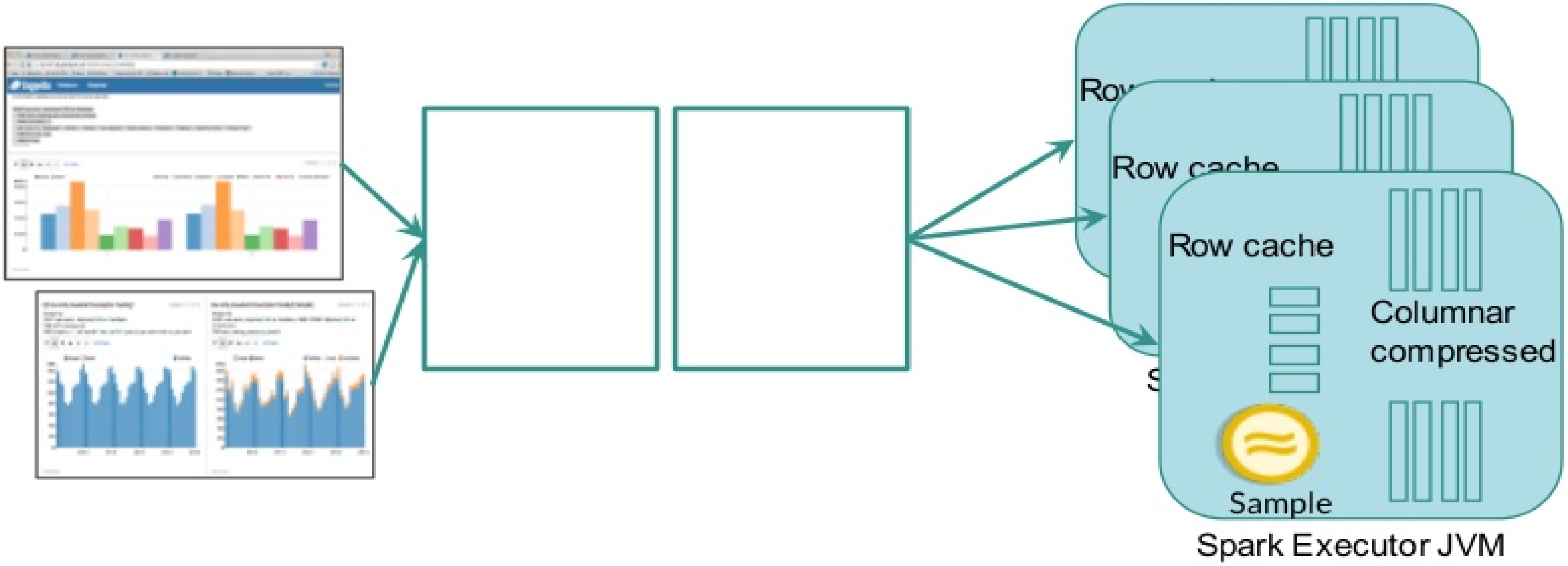
Interactive-Speed Analytic Queries – Exact or Approximate

```
Select avg(Bid), Advertiser from T1 group by Advertiser                → 100 secs
Select avg(Bid), Advertiser from T1 group by Advertiser with error 0.1 → 2 secs
```

Speed/Accuracy tradeoff

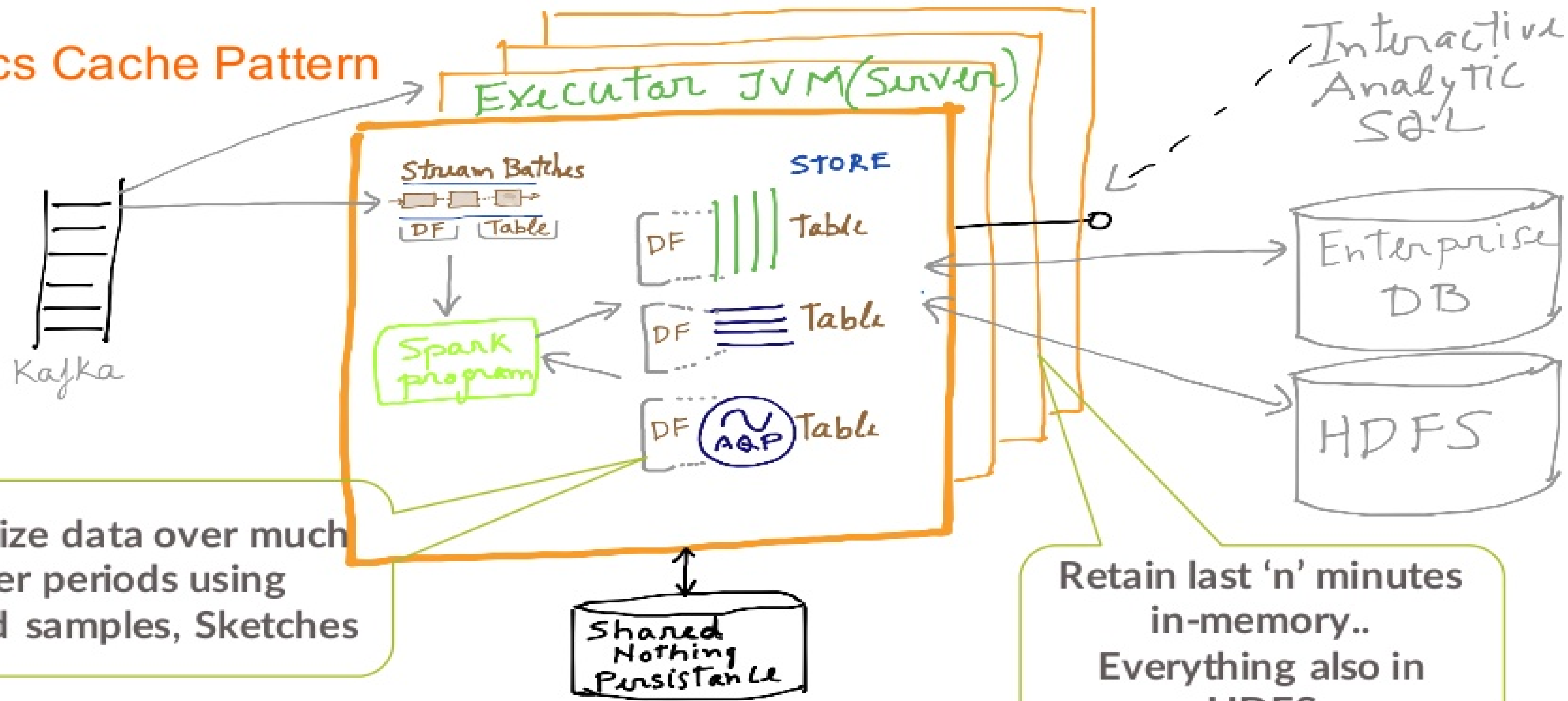


Airline Ontime performance Analytics – Demo



Revisiting AdAnalytics – Spark with colocated store

Analytics Cache Pattern

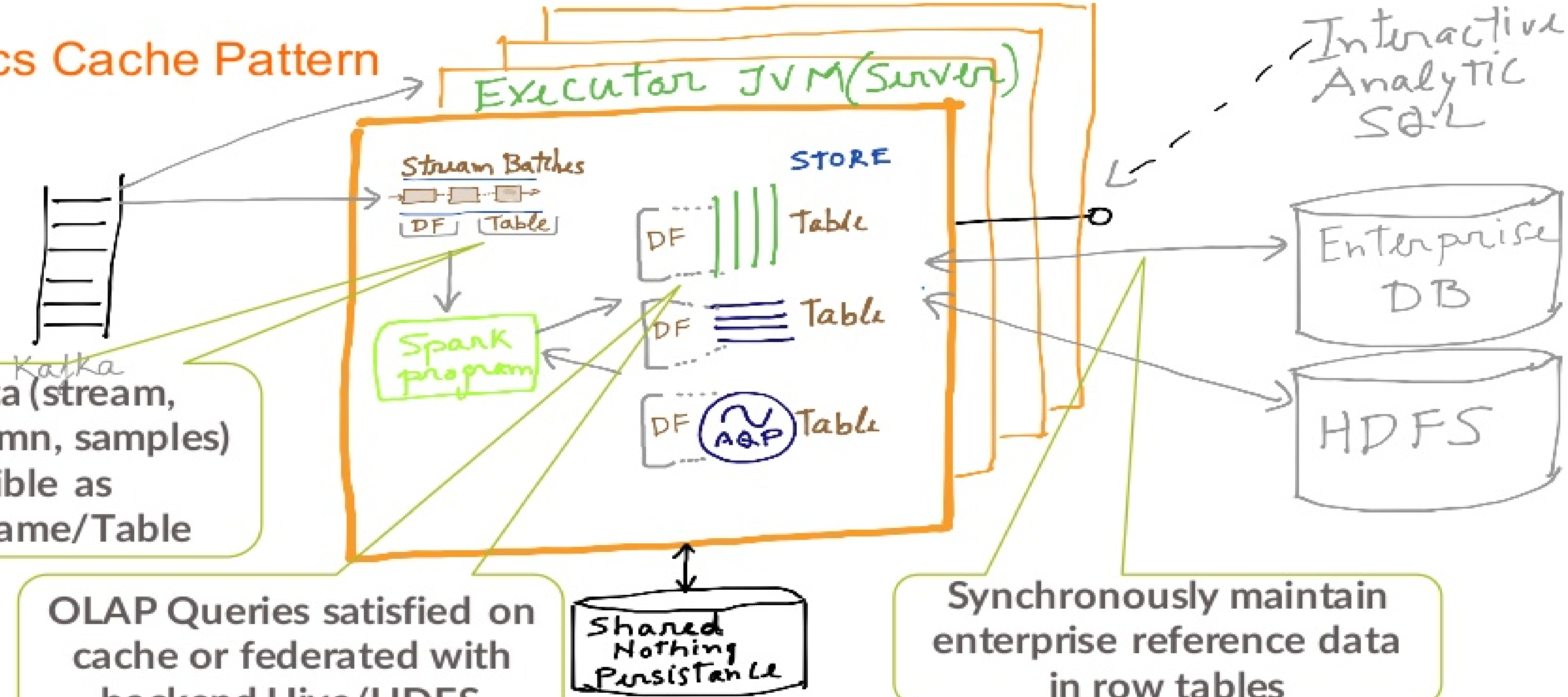


Summarize data over much longer periods using Stratified samples, Sketches

Retain last 'n' minutes in-memory.. Everything also in HDFS

Revisiting AdAnalytics – Spark with colocated store

Analytics Cache Pattern



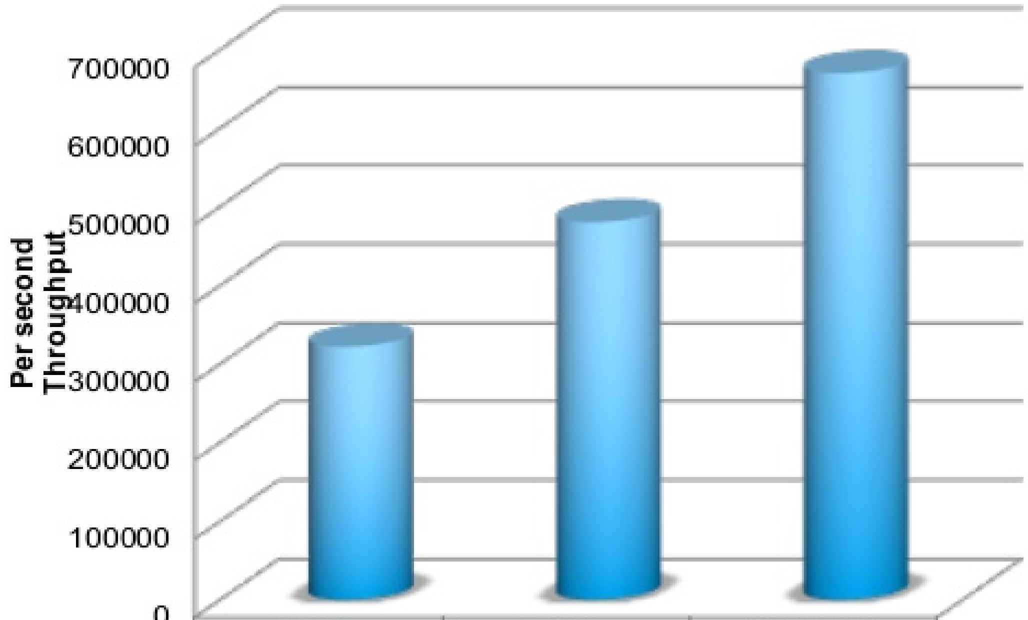
All data (stream, row/column, samples) visible as DataFrame/Table

OLAP Queries satisfied on cache or federated with backend Hive/HDFS

Synchronously maintain enterprise reference data in row tables

Concurrent Ingest + Query Performance

Stream ingestion rate
(On 4 nodes with cap on CPU to allow for queries)

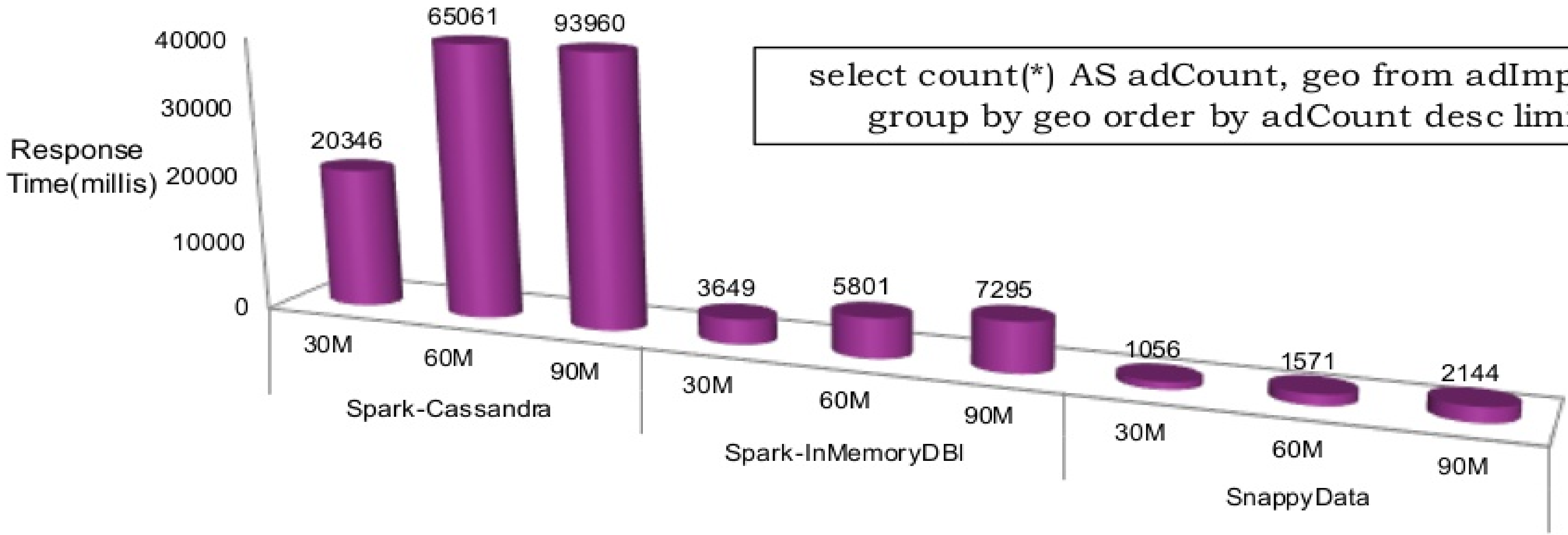


	Spark-Cassandra	Spark-InMemoryDB	SnappyData
Series1	322000	480000	670000

- AWS 4 c4.2xlarge instances
  - 8 cores, 15GB mem
- Each node parallely ingests stream from Kafka
- Parallel batch writes to store (32 partitions)
- Only few cores used for Stream writes as most of CPU reserved for OLAP queries

https://github.com/SnappyDataInc/snappy-poc

Concurrent Ingest + Query Performance



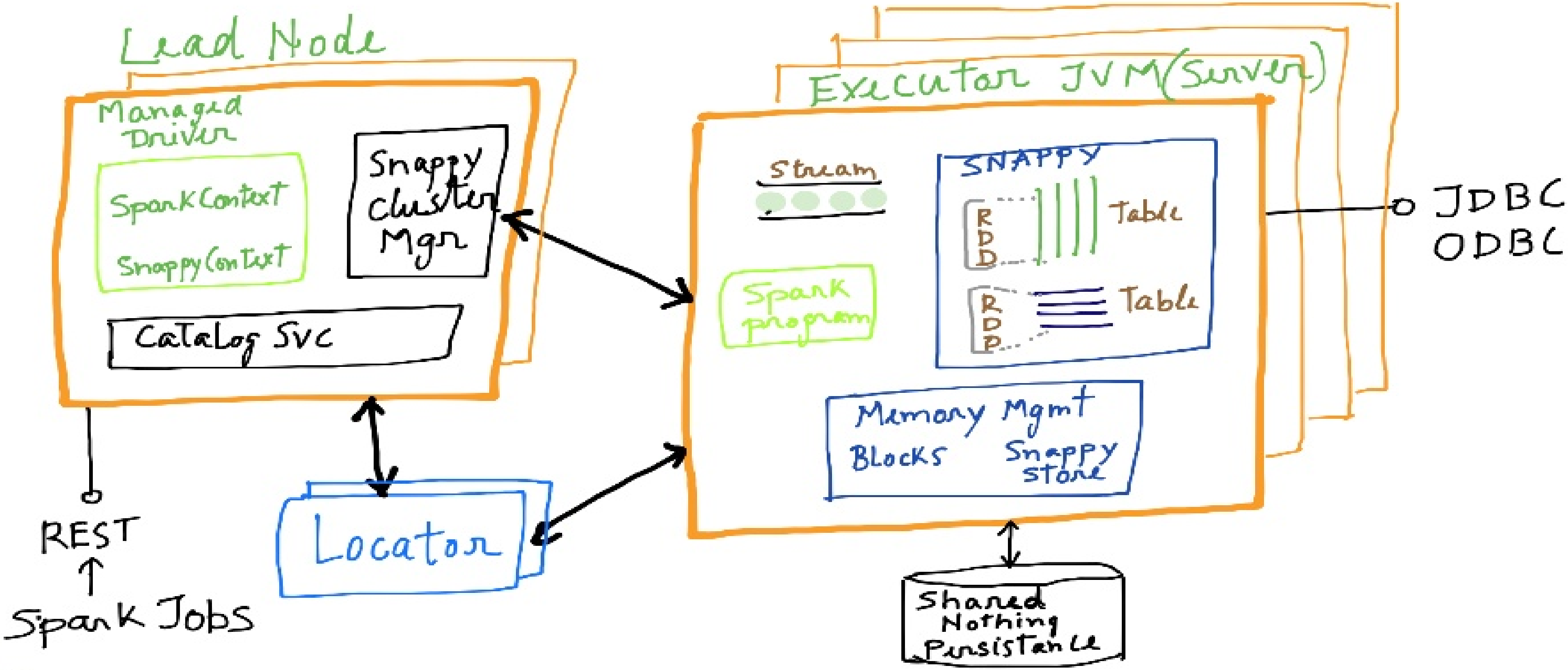
select count(*) AS adCount, geo from adImpressions group by geo order by adCount desc limit 20;

Sample "scan" oriented OLAP query(Spark SQL) performance executed while ingesting data

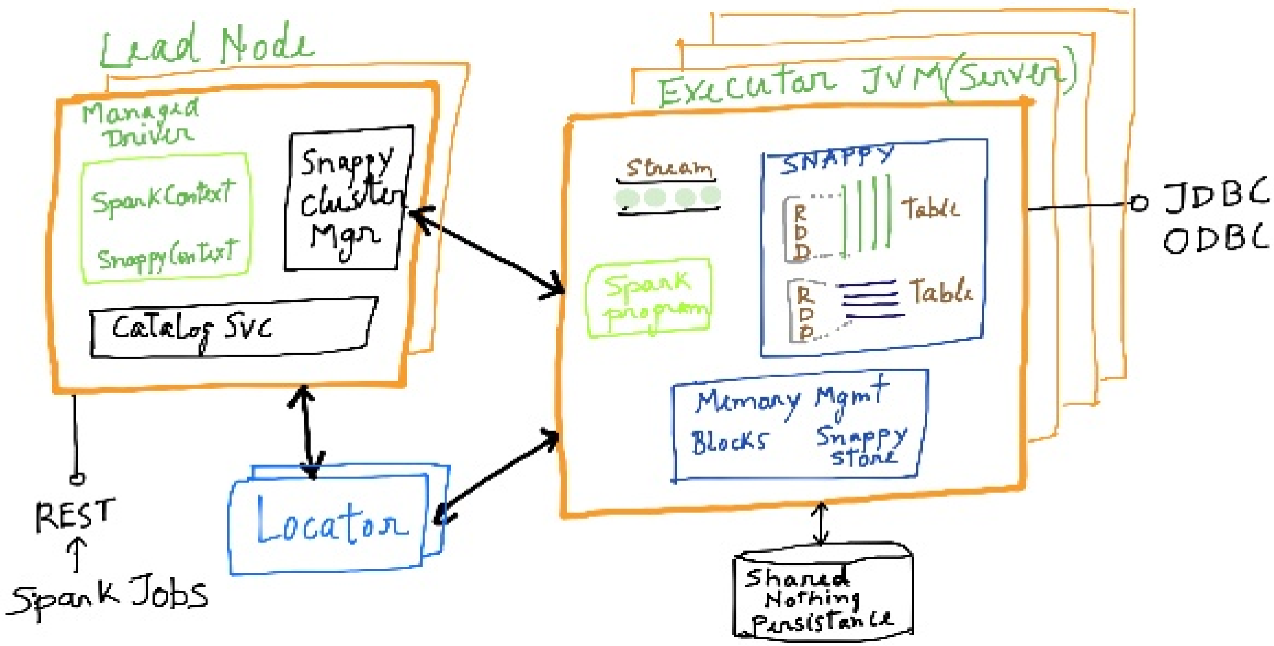
https://github.com/SnappyDataInc/snappy-poc

How SnappyData Extends Spark

Unified Cluster Architecture

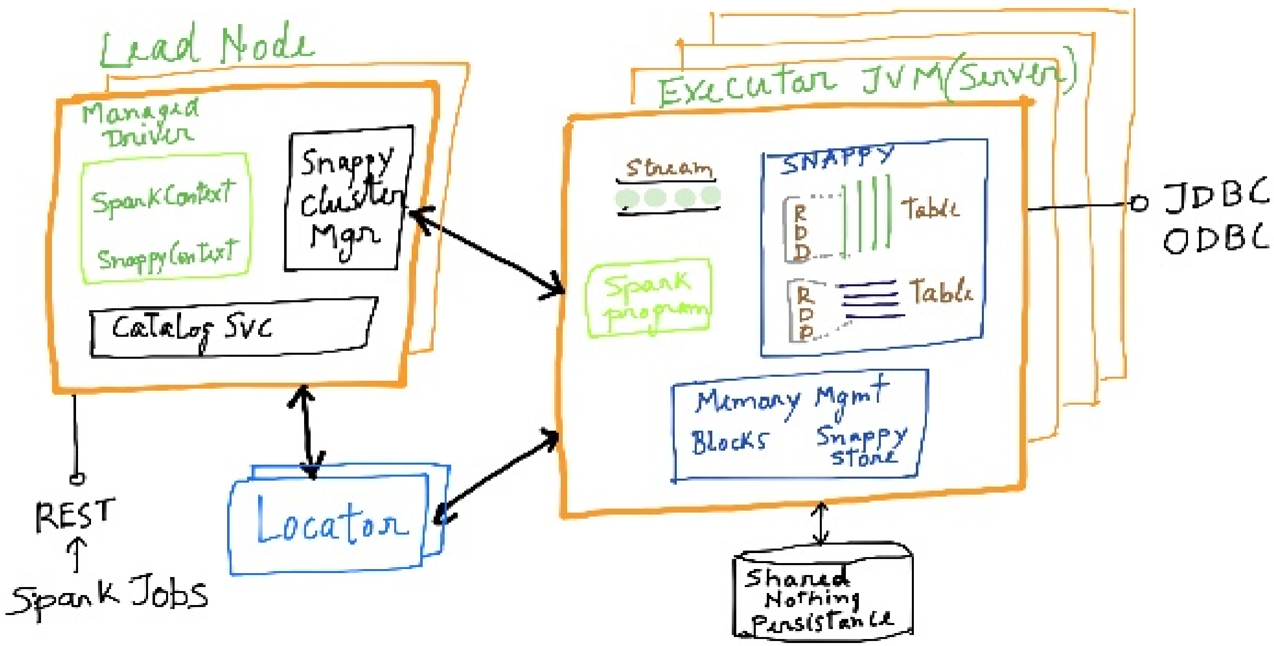


How do we extend Spark for Real Time?



- Spark Executors are long running. Driver failure doesn't shutdown Executors
- Driver HA – Drivers run "Managed" with standby secondary
- Data HA – Consensus based clustering integrated for eager replication

How do we extend Spark for Real Time?



- By pass scheduler for low latency SQL
- Deep integration with Spark Catalyst(SQL) – collocation optimizations, indexing use, etc
- Full SQL support – Persistent Catalog, Transaction, DML

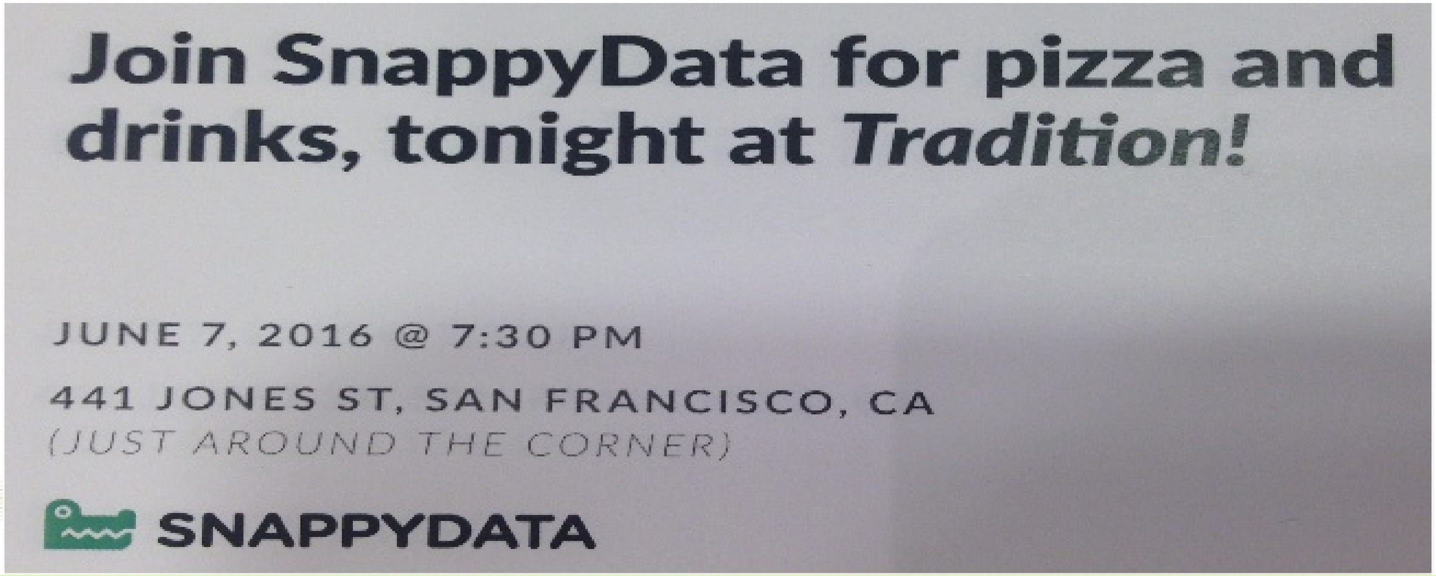
Unified OLAP/OLTP/streaming with Spark

- Far fewer resources: TB problem becomes GB.
  - CPU contention drops
- Far less complex
  - single cluster for stream ingestion, continuous queries, interactive queries and machine learning
- Much faster
  - compressed data managed in distributed memory in columnar form reduces volume and is much more responsive

SnappyData is Open Source

- Ad Analytics example/benchmark - https://github.com/SnappyDataInc/snappy-poc
- https://github.com/SnappyDataInc/snappydata

SNAPPY



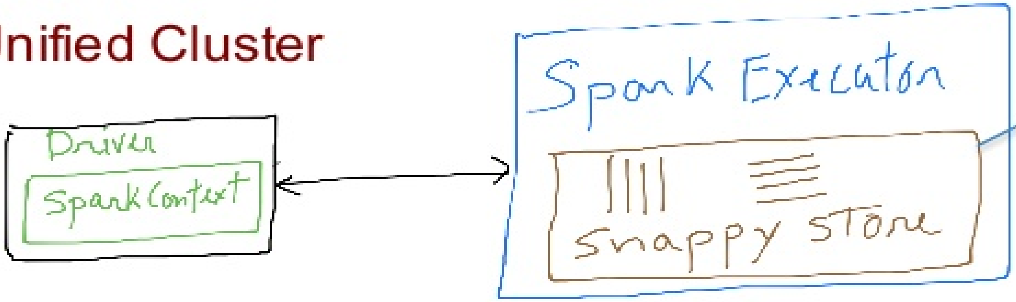
THANK YOU.

Drop by our booth to learn more.

SPARK SUMMIT 2016
DATA SCIENCE AND ENGINEERING AT SCALE
JUNE 6-8, 2016 SAN FRANCISCO

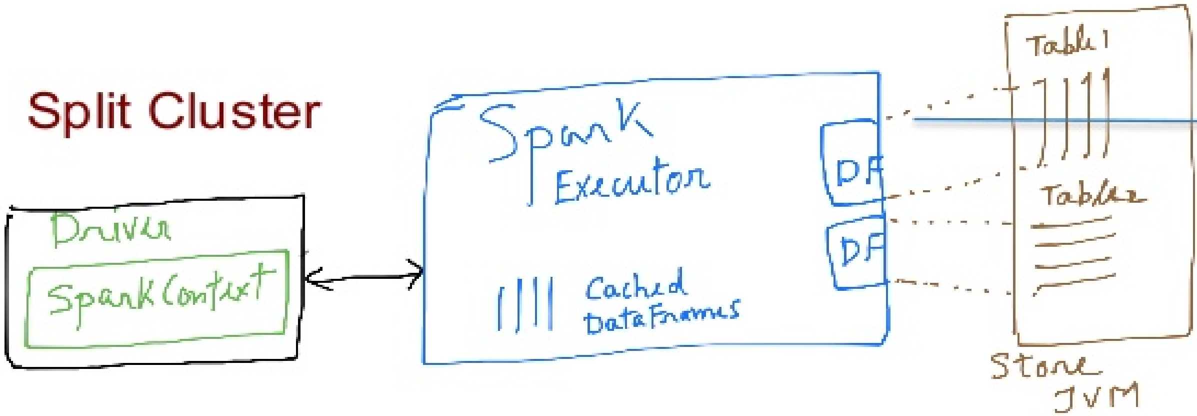
Snappy Spark Cluster Deployment topologies

Unified Cluster



- Snappy store and Spark Executor share the JVM memory
- Reference based access – zero copy

Split Cluster



- SnappyStore is isolated but use the same COLUMN FORMAT AS SPARK for high throughput

Simple API – Spark Compatible

- Access Table as DataFrame
Catalog is automatically recovered

```
val impressionLogs: DataFrame = context.table(colTable)
val campaignRef: DataFrame = context.table(rowTable)

val parquetData: DataFrame = context.table(parquetTable)
<... Now use any of DataFrame APIs ... >
```

- Store RDD[T]/DataFrame can be stored in SnappyData tables
- Access from Remote SQL clients
- Addtional API for updates, inserts, deletes

```
//Save a dataFrame using the Snappy or spark context ...
context.createExternalTable("T1", "ROW", myDataFrame.schema, props );

//save using DataFrame API
dataDF.write.format("ROW").mode(SaveMode.Append).options(props).saveAsTable("T1");
```

Extends Spark

```
CREATE [Temporary] TABLE [IF NOT EXISTS] table_name
  (
    <column definition>
  ) USING 'JDBC | ROW | COLUMN '
OPTIONS (
  COLOCATE_WITH 'table_name',          // Default none
  PARTITION_BY 'PRIMARY KEY | column name', // will be a replicated table, by default
  REDUNDANCY      '1' ,                // Manage HA
 PERSISTENT  "DISKSTORE_NAME ASYNCHRONOUS |  SYNCHRONOUS",
                    // Empty string will map to default disk store.
 OFFHEAP "true | false"
 EVICTION_BY "MEMSIZE 200 | COUNT 200 | HEAPPERCENT",
.....
 [AS select_statement];
```

Simple to Ingest Streams using SQL

- Consume from stream
- Transform raw data
- Continuous Analytics
- Ingest into in-memory Store
- Overflow table to HDFS

```
Create stream table AdImpressionLog
(<Columns>) using directkafka_stream options (
<socket endpoints>
"topics 'adnetwork-topic' ",
"rowConverter ' AdImpressionLogAvroDecoder' )
```

```
// Register CQ
streamingContext.registerCQ(
    "select publisher, geo, avg(bid) as avg_bid, count(*) imps,
    count(distinct(cookie)) uniques from AdImpressionLog
    window (duration '2' seconds, slide '2' seconds)
    where geo != 'unknown' group by publisher, geo")

.foreachDataFrame(df => {
    df.write.format("column").mode(SaveMode.Append)
    .saveAsTable("adImpressions")
```