700 *Updatable* Queries Per Second: Spark as a Real-Time Web Service

Evan Chan June 2016

Who Am I?

- User and contributor to Spark since 0.9, Cassandra since 0.6
  - Datastax Cassandra MVP
- Created Spark Job Server and FiloDB
- Talks at Spark Summit, Cassandra Summit, Strata, Scala Days, etc.

Apache Spark

Usually used for rich analytics, not time-critical.

- Machine learning: generating models, predictions, etc.
- SQL Queries seconds to minutes, low concurrency
- Stream processing

What about for low-latency, highly concurrent queries? Dashboards?

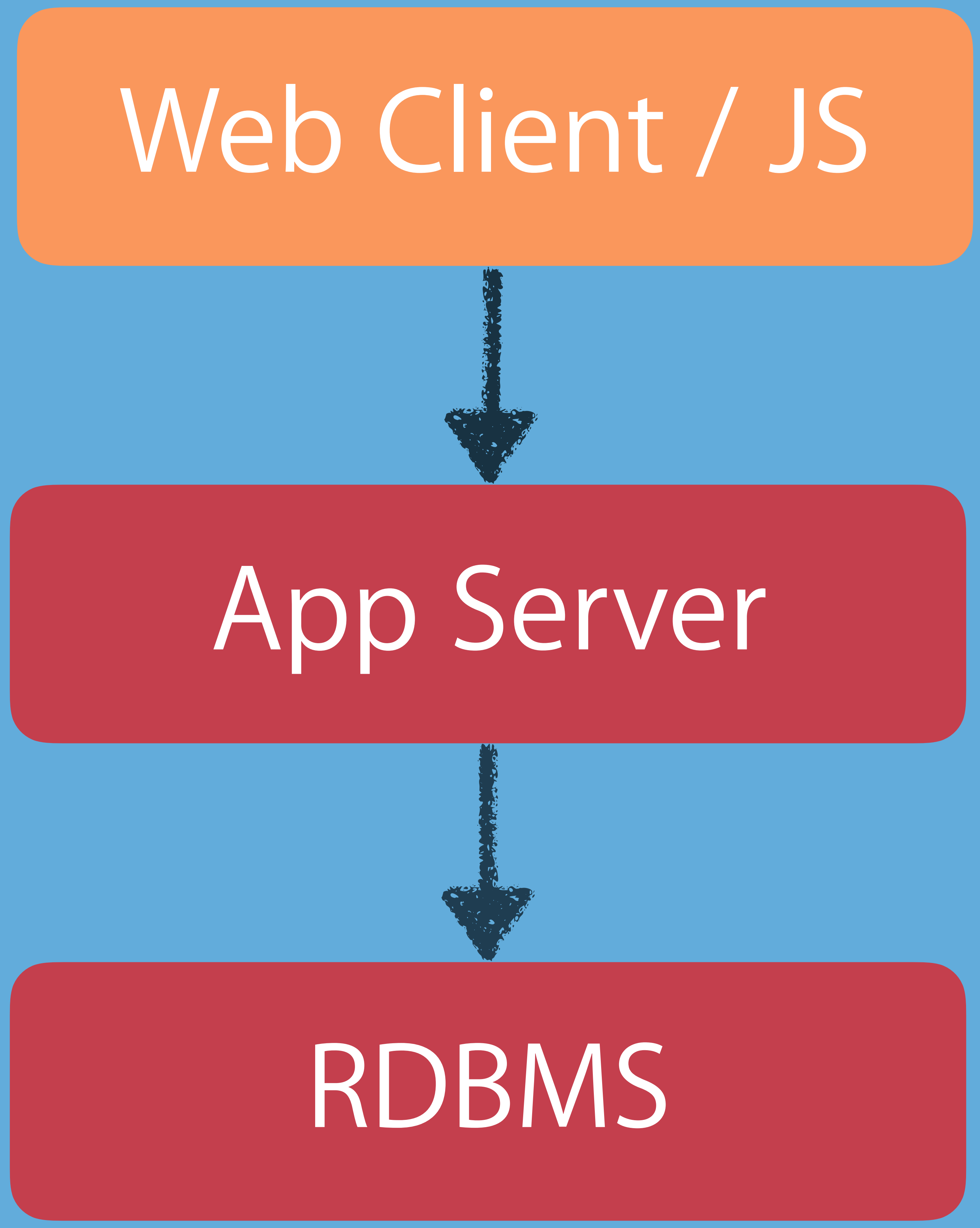
Low-Latency Web Queries

Why is it important?

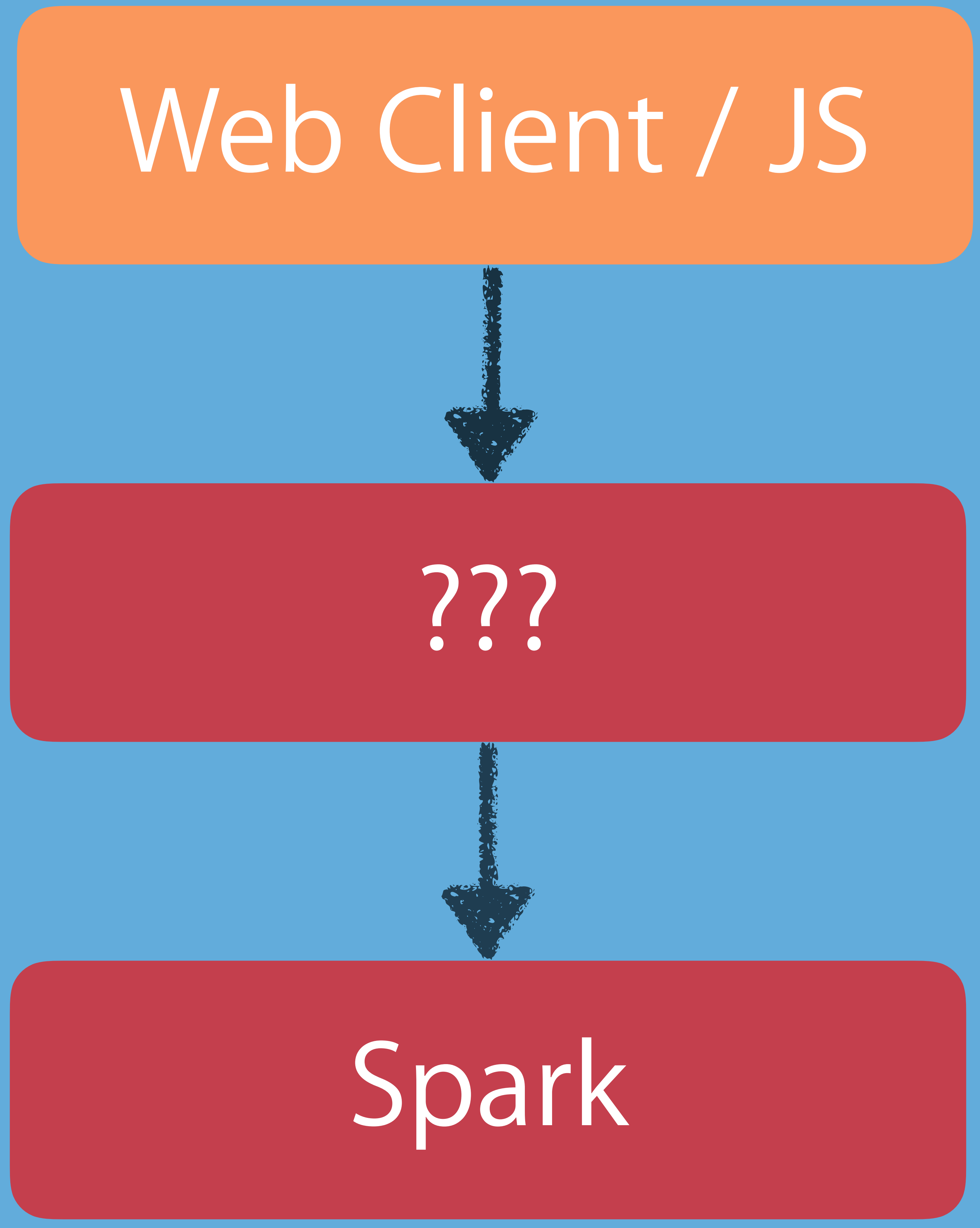
- Dashboards
- Interactive analytics
- Real-time data processing

Why not use the Spark stack for this?

Web Query Stack



Spark-based Low-Latency Stack



Creating a new SparkContext is S-L-O-W

- Start up HTTP/BitTorrent File Server
- Start up UI
- Start up executor processes and wait for confirmation

The bigger the cluster, the slower!

Using a Persistent Context for Low Latency

- Avoid high overhead of Spark application launch
- Standard pattern:
  - Spark Job Server
  - Hive Thrift Server
- Accept queries and run them in context
- Usually means fixed resources - great for SLA predictability

FAIR Scheduling

- FIFO vs FAIR Scheduling
  - FAIR scheduler can co-schedule concurrent Spark jobs even if they take up lots of resources
  - Scheduler pools with individual policies
- Higher concurrency
- FIFO allows concurrency if tasks do not use up all threads
- In Mesos, use coarse-grained mode to avoid launching executors on every Spark task

Low-Latency Game Plan

- Start a persistent Spark Context (ex. the Hive ThriftServer - we'll get to that below)
- Run it in FAIR scheduler mode
- Use fast in-memory storage
- Maximize concurrency by using as few partitions/ threads as possible
- Host the data and run it on a single node - avoid expensive network shuffles

In-Memory Storage

- Is it really faster than on disk files? With OS Caching
  - It's about *consistency* of performance - not just hot data in the page cache, but ALL data.
  - Fast random access
- Making different tradeoffs as new memory technologies emerge (NVRAM etc.)
  - Higher IO -> less need for compression
  - Apache Arrow

So, let's talk about Spark storage in detail…

HDFS? Parquet Files?

- Column pruning speeds up I/O significantly
- Still have to scan lots of files
- File organization not the easiest for filtering
- For low-latency, need much more fine-grained indexing

Cached RDDs

Let's say you have an RDD[T], where each item is of type T.

- Bytes are saved on JVM heap, or optionally heap + disk
- Spark optionally serializes it, using by default Java serialization, so it (hopefully) takes up less space
- Pros: easy (`myRdd.cache()`)
- Cons: have to iterate over every item, no column pruning, slow if need to deserialize, memory hungry, cannot update

Cached DataFrames

Works on a DataFrame (`RDD[Row]` with a schema)

```
sqlContext.cacheTable(tableA)
```

- Uses columnar storage for very efficient storage
- Columnar pruning for faster querying
- Pros: easy, efficient memory footprint, fast!
- Cons: no filtering, cannot update

Why are Updates Important?

- Appends
  - Streaming workloads. Add new data continuously.
  - Real data is *always* changing.   Queries on live real-time data has business benefits.
- Updates
  - Idempotency = really simple ingestion pipelines
  - Simpler streaming later
  - update late events (See Spark 2.0 Structured Streaming)

Advantages of Filtering

- Two methods to lower query latency:
  - Scan data faster (in-memory)
  - Scan less data (filtering)
- RDDs and cached DFs - prune by partition
- Dynamo/BigTable - 2D Filtering
  - Filter by partition
  - Filter within partitions

Workarounds - Updating RDDs

- Union(oldRDD, newRDD)
  - Creates a tree of RDDs - slows down queries significantly
- IndexedRDD

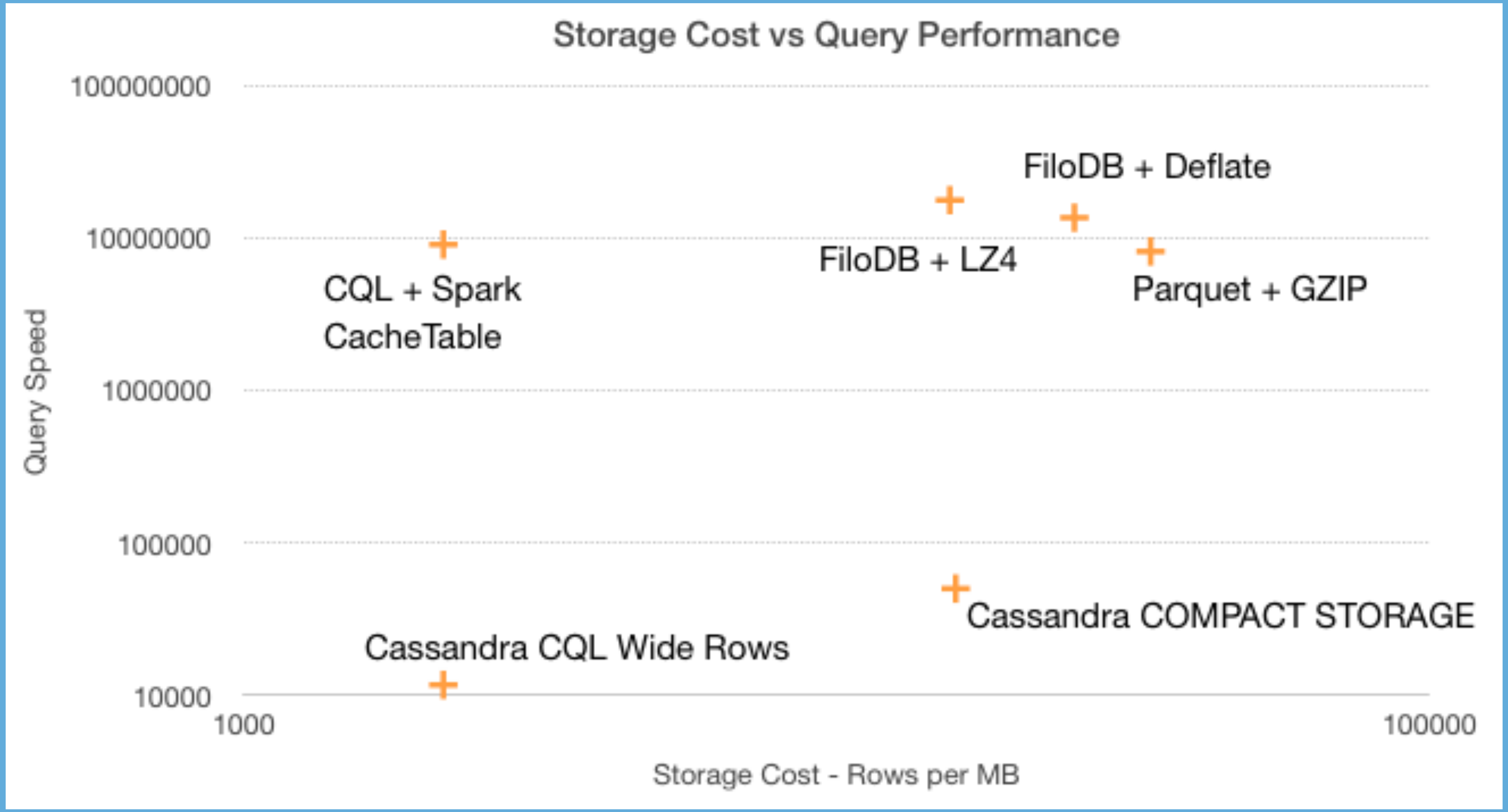
Introducing FiloDB.

A distributed, versioned, columnar analytics database.
Built for streaming.

Fast Analytics Storage

- Scan speeds competitive with Apache Parquet
  - In-memory version significantly faster
- Flexible filtering along two dimensions
  - Much more efficient and flexible partition key filtering
- Efficient columnar storage using dictionary encoding and other techniques
- Updatable

Comparing Storage Costs and Query Speeds



https://www.oreilly.com/ideas/apache-cassandra-for-analytics-a-performance-and-storage-analysis

Robust Distributed Storage

In-memory storage engine, or
Apache Cassandra as the rock-solid storage engine.

Cassandra-like Data Model

	Column A		Column B	
Partition Key 1	Segment 1	Segment 2	Segment 1	Segment 2
Partition Key 2	Segment 1	Segment 2	Segment 1	Segment 2

- **partition keys** - distributes data around a cluster, and allows for fine grained and flexible filtering
- **segment keys** - do range scans within a partition, e.g. by time slice
- primary key based ingestion and updates

Very Flexible Filtering

Unlike Cassandra, FiloDB offers very flexible and efficient filtering on partition keys. Partial key matches, fast IN queries on any part of the partition key.

No need to write multiple tables to work around answering different queries.

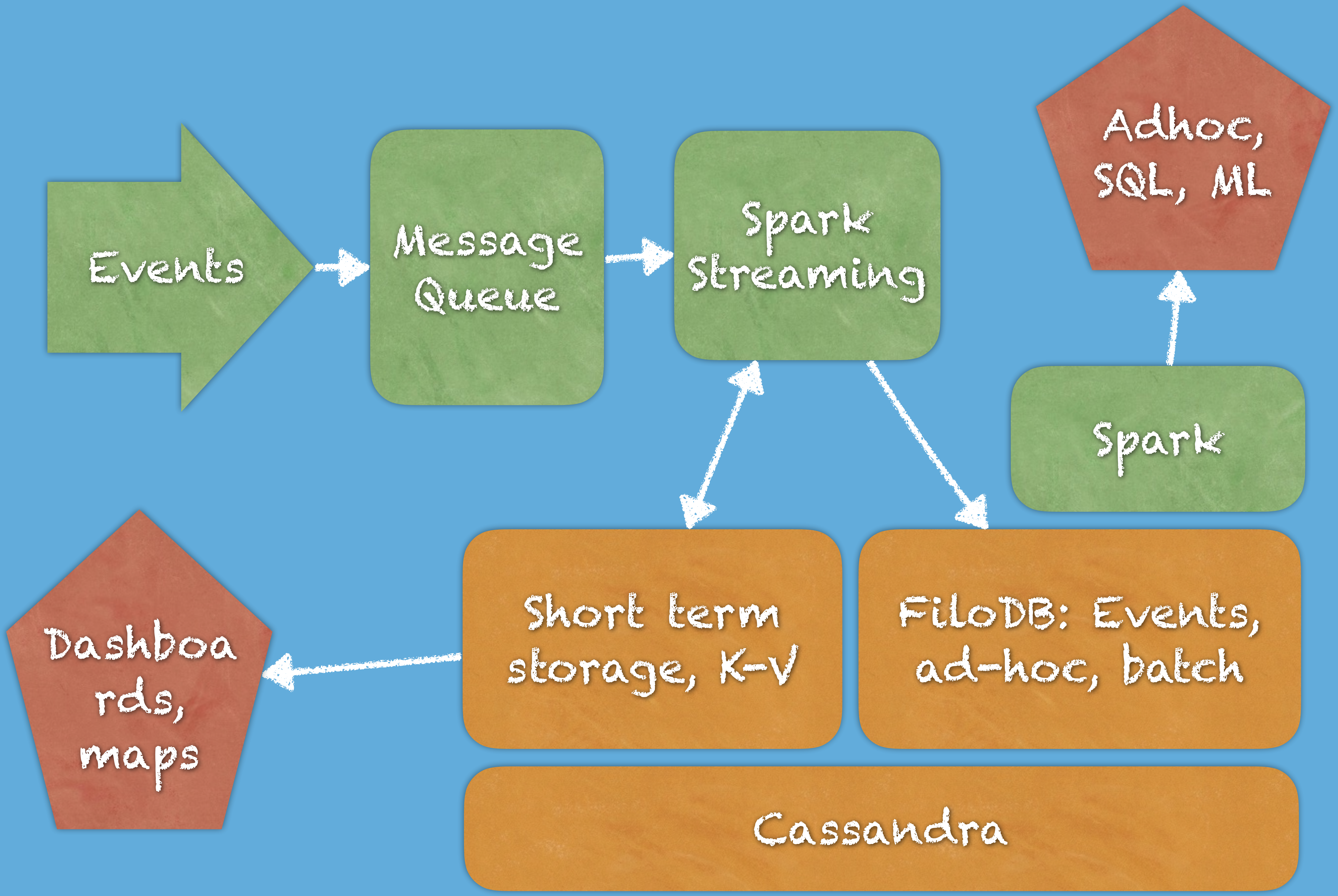
Spark SQL Queries!

```
CREATE TABLE gdelt USING filodb.spark OPTIONS (dataset "gdelt");

SELECT Actor1Name, Actor2Name, AvgTone FROM gdelt ORDER BY AvgTone
DESC LIMIT 15;

INSERT INTO gdelt SELECT * FROM NewMonthData;
```

- Read to and write from Spark Dataframes
- Append/merge to FiloDB table from Spark Streaming
- Use Tableau or any other JDBC tool

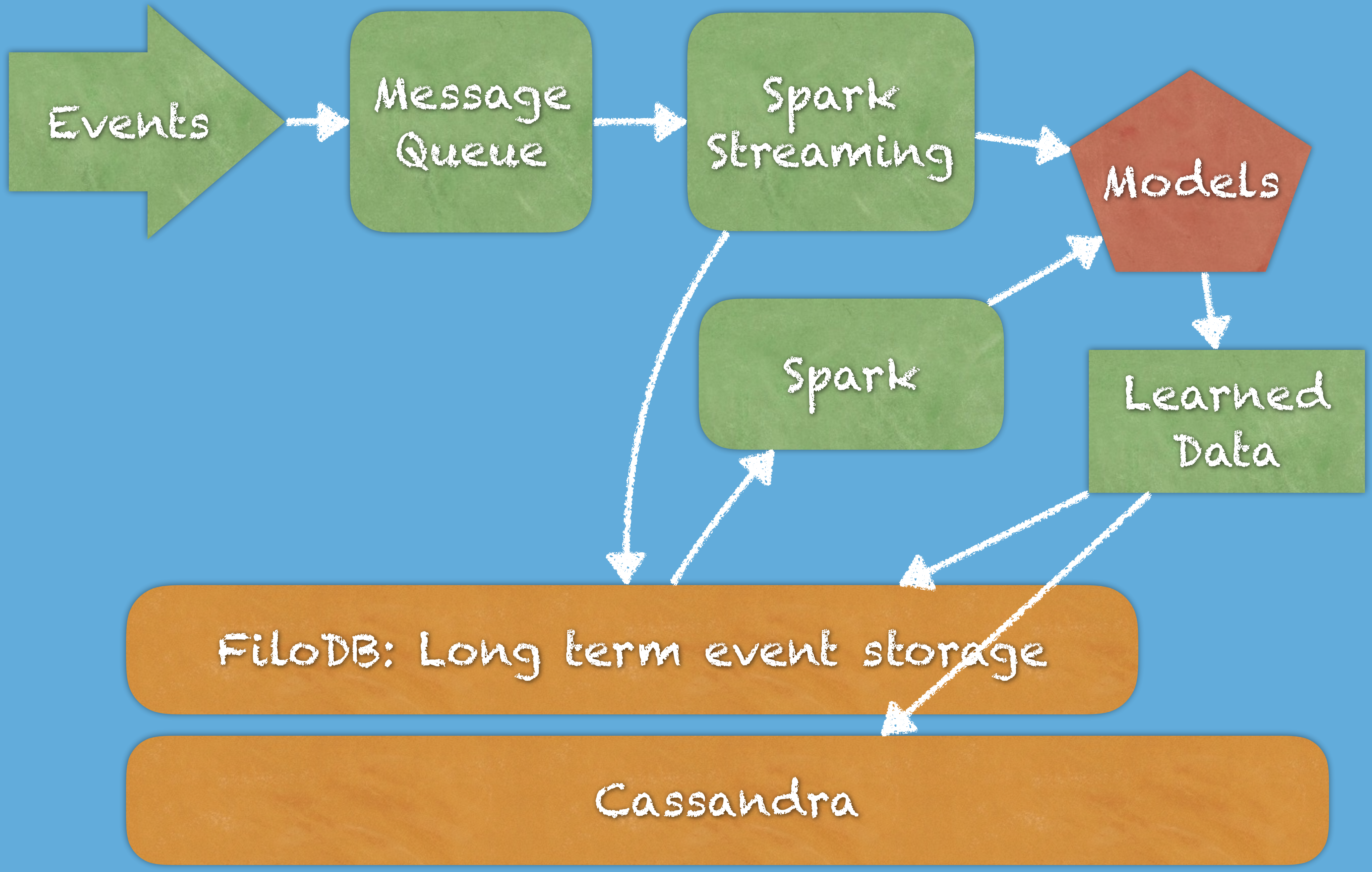
What's in the Name?

Rich, sweet layers of distributed, versioned database goodness


Events
Message Queue
Spark Streaming
Adhoc, SQL, ML
Spark
Dashboards, maps
Short term storage, K-V
FiloDB: Events, ad-hoc, batch
Cassandra

SMACK stack for all your analytics

- Regular Cassandra tables for highly concurrent, aggregate / key-value lookups (dashboards)
- FiloDB + C* + Spark for efficient long term event storage
  - Ad hoc / SQL / BI
  - Data source for MLLib / building models
  - Data storage for classified / predicted / scored data


Events
Message Queue
Spark Streaming
Models
Spark
Learned Data
FiloDB: Long term event storage
Cassandra

Fast SQL Server in Spark

Data: The New York City Taxi Dataset

The public NYC Taxi Dataset contains telemetry (pickup, dropoff locations, times) info on millions of taxi rides in NYC.

Medallion Prefix	1/1 - 1/6	1/7 - 1/12
AA	records	records
AB	records	records

- Partition key - `:stringPrefix medallion 2` - hash multiple drivers trips into ~300 partitions
- Segment key - `:timeslice pickup_datetime 6d`
- Row key - hack_license, pickup_datetime

Allows for easy filtering by individual drivers, and slicing by time.

collectAsync

To support running concurrent queries better, we rely on a relatively unknown feature of Spark's RDD API, `collectAync`:

```
sqlContext.sql(queryString).rdd.collectAsync
```

This returns a Scala Future, which can easily be composed using `Future.sequence` to launch a whole series of asynchronous RDD operations. They will be executed with the help of a separate ForkJoin thread pool.

Initial Results

- Run lots of queries concurrently using collectAsync
- Spark local[*] mode
- SQL queries on first million rows of NYC Taxi dataset
- 50 Queries per Second
- Most of time not running queries but parsing SQL !

Some Observations

1. Starting up a Spark task is actually pretty low latency - milliseconds
2. One huge benefit to filtering is reduced thread/CPU usage. Most of the queries ended up being single partition / single thread.

Lessons

1. Cache the SQL to DataFrame/LogicalPlan parsing. This saves ~20ms per parse, which is not insignificant for low-latency apps
2. Distribute the SQL parsing away from the main thread so it's not gated by one thread

SQL Plan Caching

```
val cachedDF = new collection.mutable.HashMap[String, DataFrame]

def getCachedDF(query: String): DataFrame =
  cachedDF.getOrElseUpdate(query, sql.sql(query))
```

Cache the `DataFrame` containing the logical plan translated from parsing SQL.

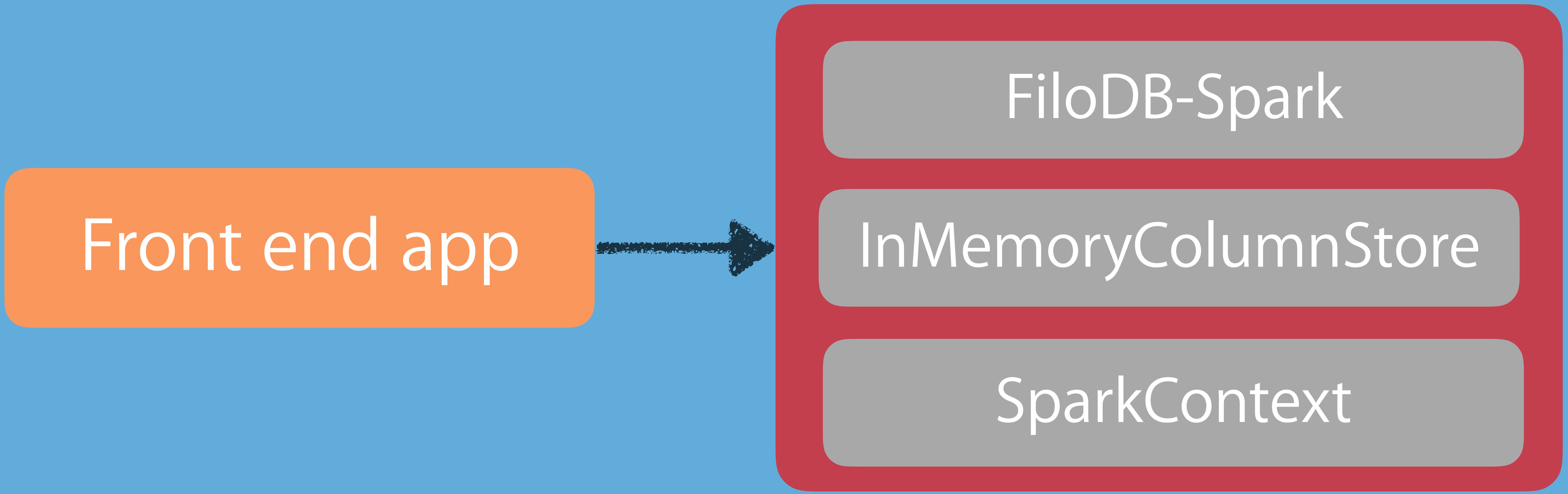
Now - **700 QPS**!!

Scaling with More Data

15 million rows of NYC Taxi data - **still 700 QPS**!

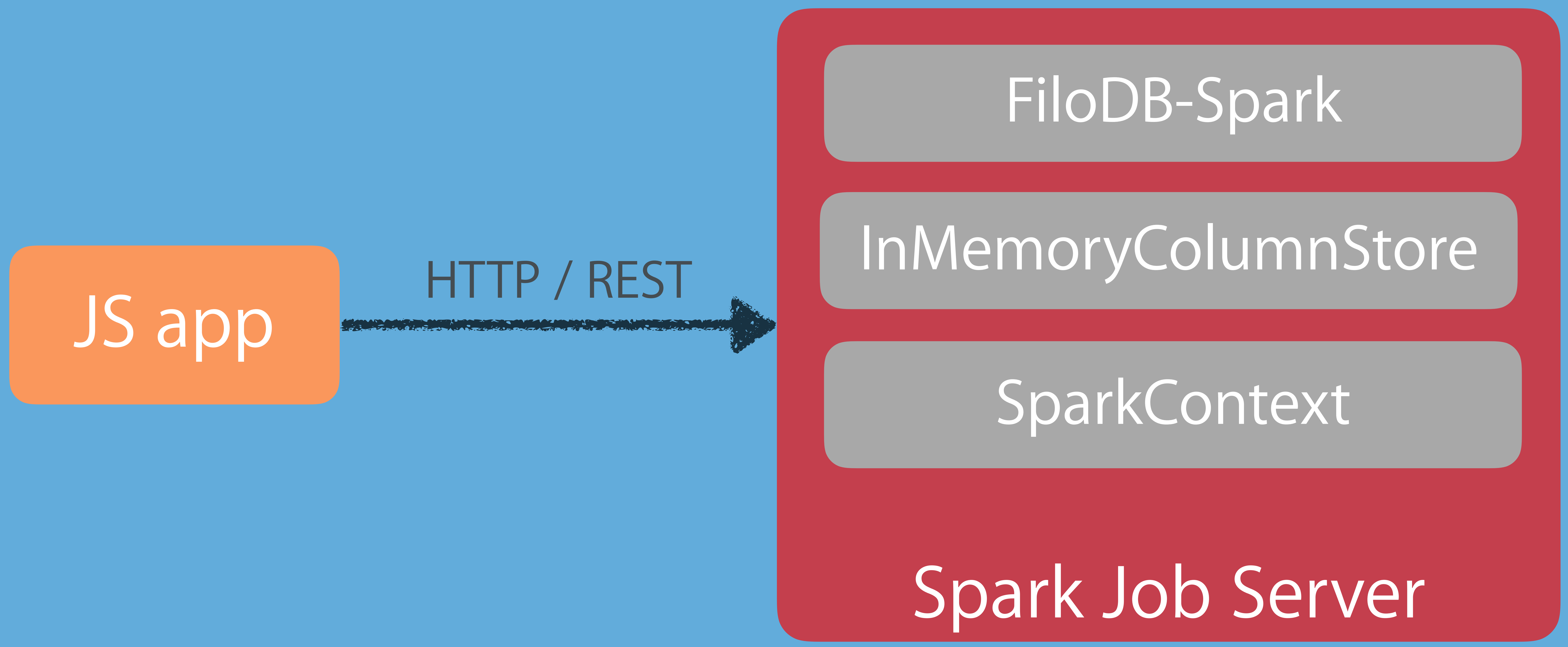
This makes sense due to the efficiency of querying.

Fast Spark Query Stack



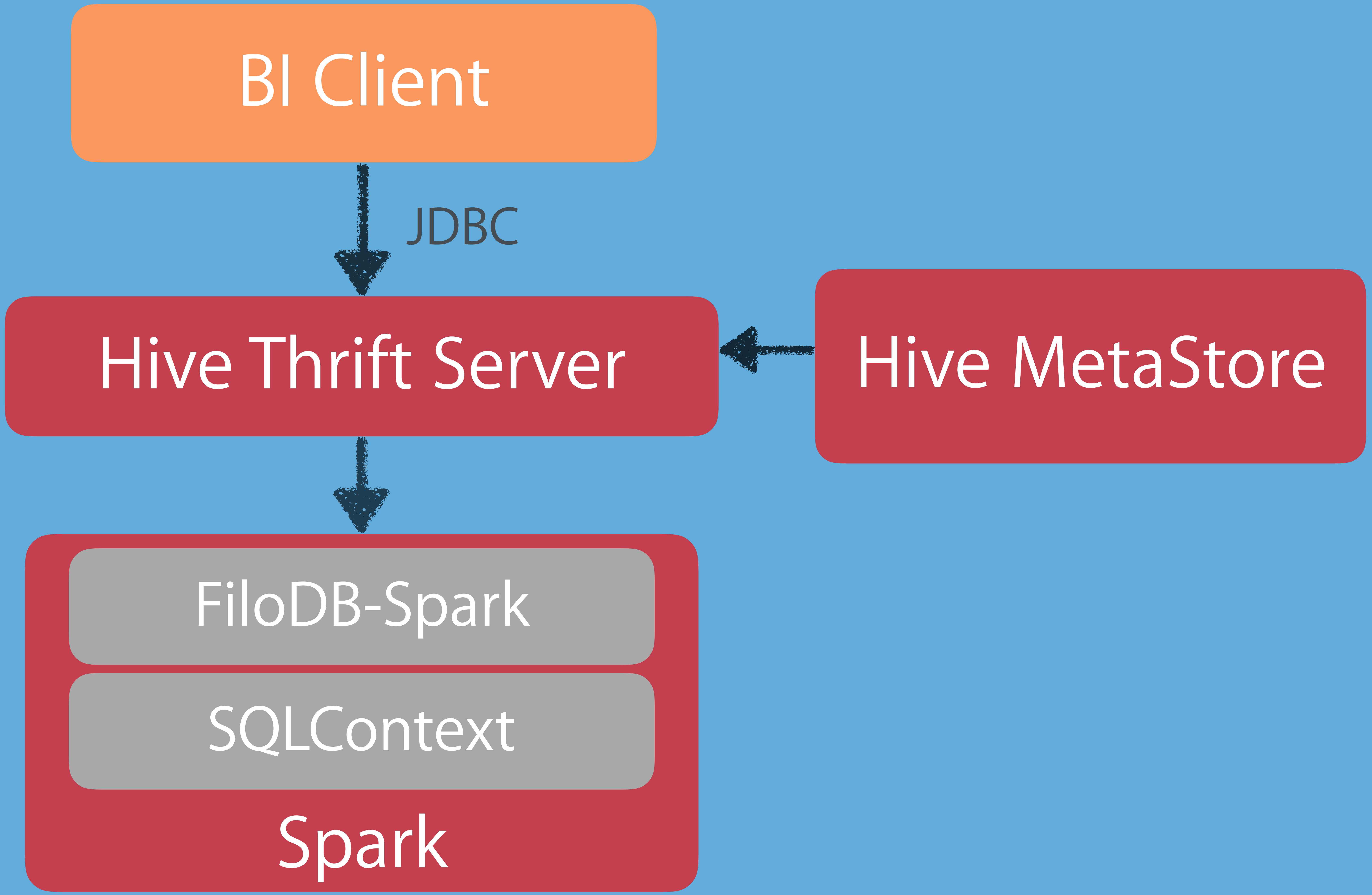
- Run Spark context on heap with `local[*]`
- Load FiloDB-Spark connector, load data in memory
- Very fast queries all in process

Fast Spark Query Stack II



- HTTP/REST using Spark Job Server

Slower: Hive Thrift Server Stack



Your Contributions Welcome!

http://github.com/tuplejump/FiloDB

This is not a contribution