Using Kafka and Kudu for fast, low-latency SQL analytics on streaming data

Mike Percy, Software Engineer, Cloudera
Ashish Singh, Software Engineer, Cloudera

The problem

- Building a system that supports low-latency streaming and batch workloads simultaneously is hard
- Current solutions to this are complicated and error-prone (e.g. lambda arch)
- Too much expertise is currently required to make it work

Building a near-real time data architecture

How can we...

1. Enable data to flow into our query system quickly, reliably, and efficiently
2. Allow for stream-processing of this data if desired
3. Enable batch processing to access up-to-the-second information

...while keeping complexity at a minimum?

Problem domains that require stream + batch

Credit Card & Monetary Transactions

Identify fraudulent transactions as soon as they occur.

Healthcare

Continuously monitor patient vital stats and proactively identify at-risk patients.
Report on this data.

Retail

- Real-time in-store Offers and Recommendations.
- Email and marketing campaigns based on real-time social trends

Digital Advertising & Marketing

Optimize and personalize digital ads based on real-time information.

Consumer Internet, Mobile & E-Commerce

Optimize user engagement based on user's current behavior. Deliver recommendations relevant "in the moment"

Manufacturing

- Identify equipment failures and react instantly
- Perform proactive maintenance.
- Identify product quality defects immediately to prevent resource wastage.

Security & Surveillance

Identify threats and intrusions, both digital and physical, in real-time.

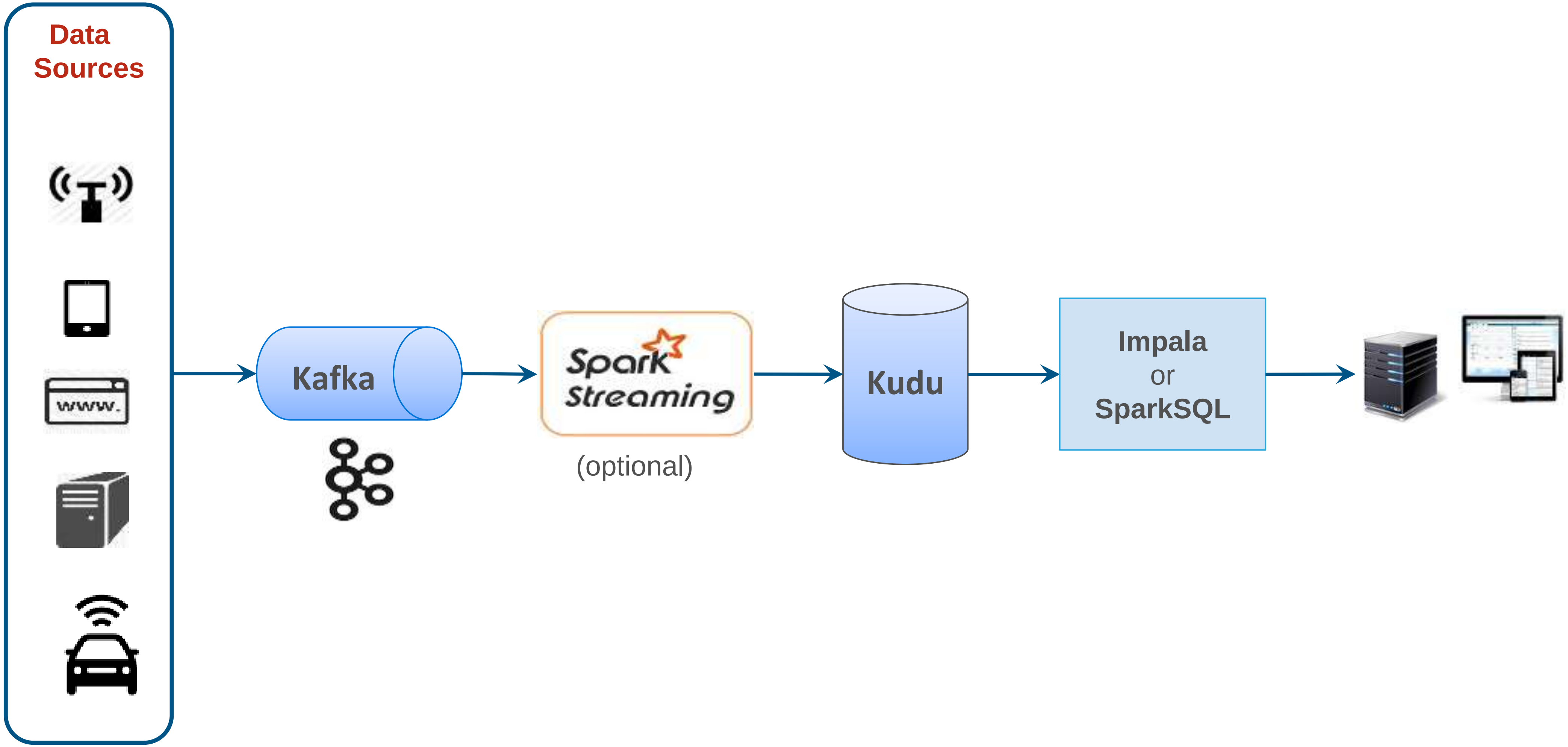
Transportation & Logistics

- Real-time traffic conditions
- Tracking fleet and cargo locations and dynamic re-routing to meet SLAs

Agenda

- A new low-latency, high throughput architecture for analytics
- Building an ingest pipeline
- Storing and querying structured data
- Design tradeoffs
- Demo
- Q&A

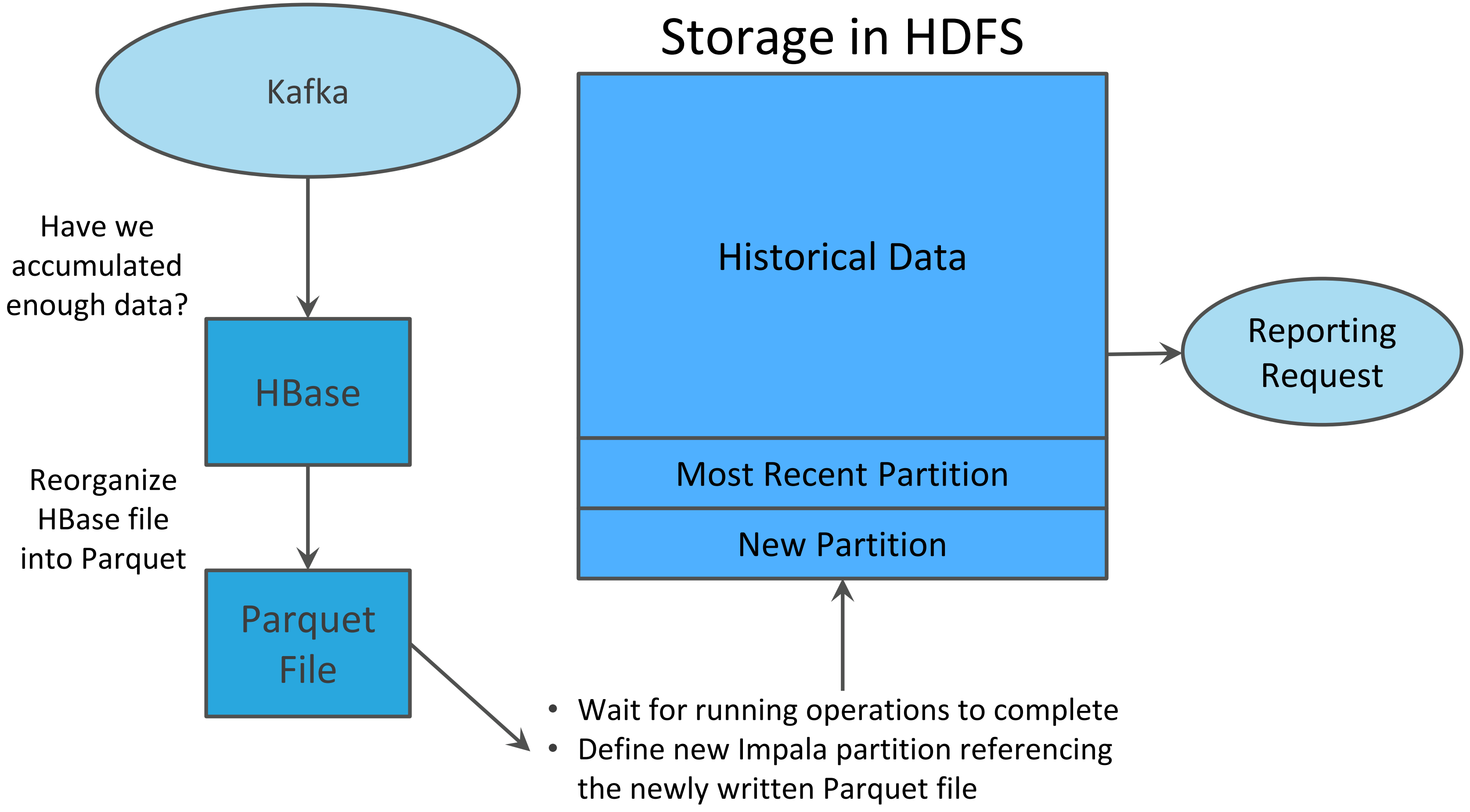
A modern, low-latency analytics architecture



cloudera

“Traditional” real-time analytics in Hadoop

Fraud detection in the real world means storage complexity

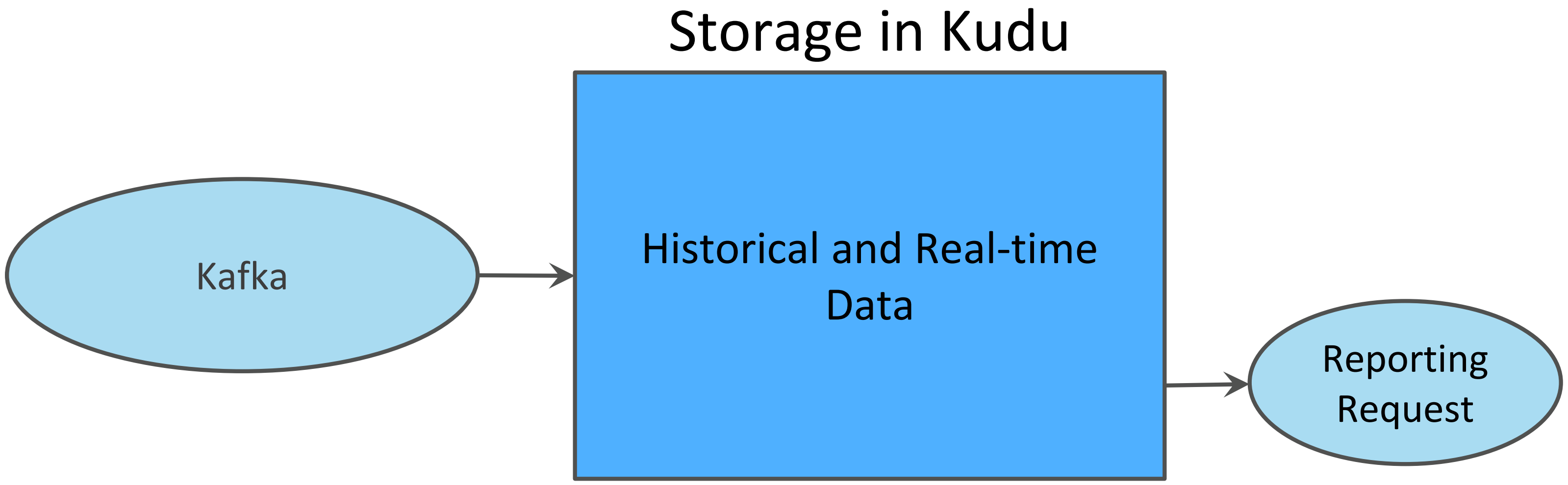


- Wait for running operations to complete
- Define new Impala partition referencing the newly written Parquet file

Considerations:

- How do I handle failure during this process?
- How often do I reorganize data streaming in into a format appropriate for reporting?
- When reporting, how do I see data that has not yet been reorganized?
- How do I ensure that important jobs aren’t interrupted by maintenance?

Real-time analytics in Hadoop with Kudu



Improvements:

- Fewer systems to operate
- No cron jobs or background processes
- Handle late arrivals or data corrections with ease
- New data available immediately for analytics or operations

Xiaomi use case

- World’s 4th largest smart-phone maker (most popular in China)
- Gather important RPC tracing events from mobile app and backend service.
- Service monitoring & troubleshooting tool.

High write throughput

- >5 Billion records/day and growing

Query latest data and quick response

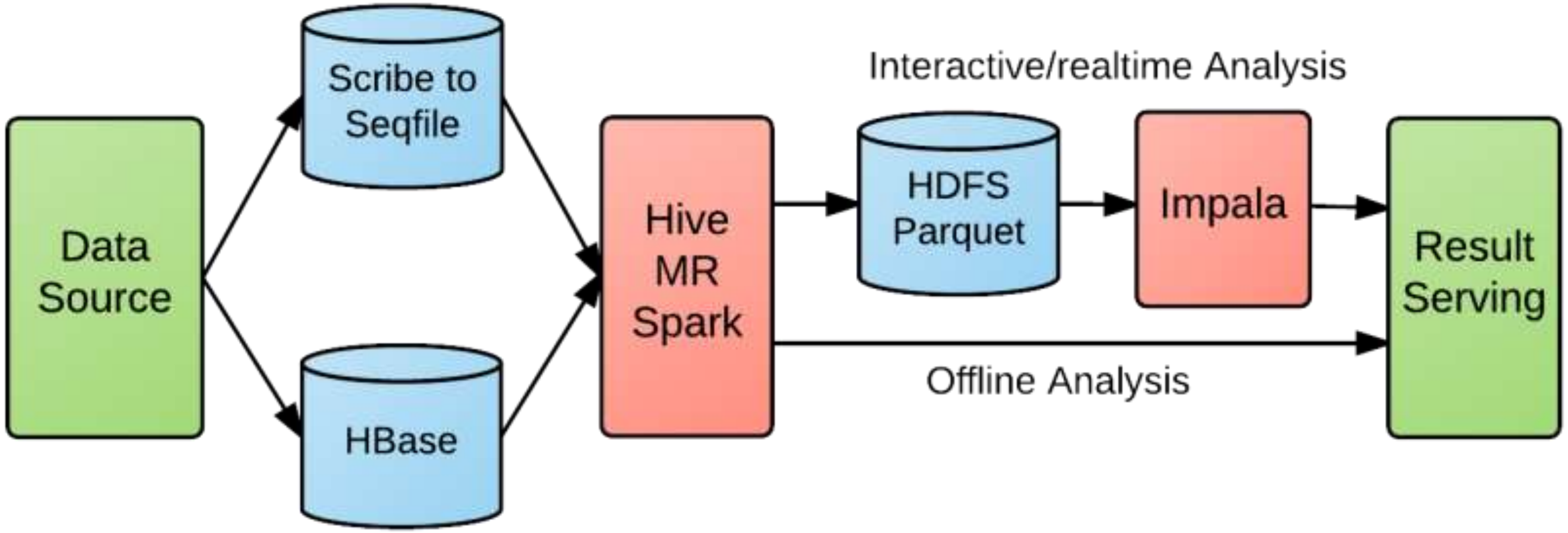
- Identify and resolve issues quickly

Can search for individual records

- Easy for troubleshooting

Xiaomi big data analytics pipeline

Before Kudu



Large ETL pipeline delays

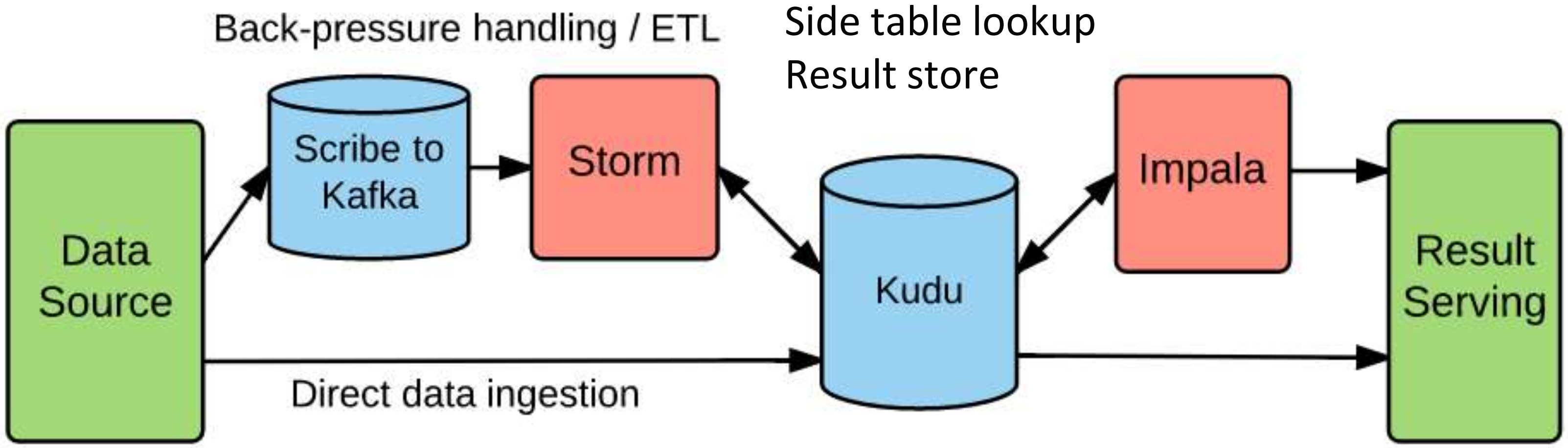
- High data visibility latency (from 1 hour up to 1 day)
- Data format conversion woes

Ordering issues

- Log arrival (storage) not exactly in correct order
- Must read 2 – 3 days of data to get all of the data points for a single day

Xiaomi big data analytics pipeline

Simplified with Kafka and Kudu



Back-pressure handling / ETL

Data Source

Scribe to Kafka

Storm

OLAP scan
Side table lookup
Result store

Kudu

Impala

Result Serving

Direct data ingestion

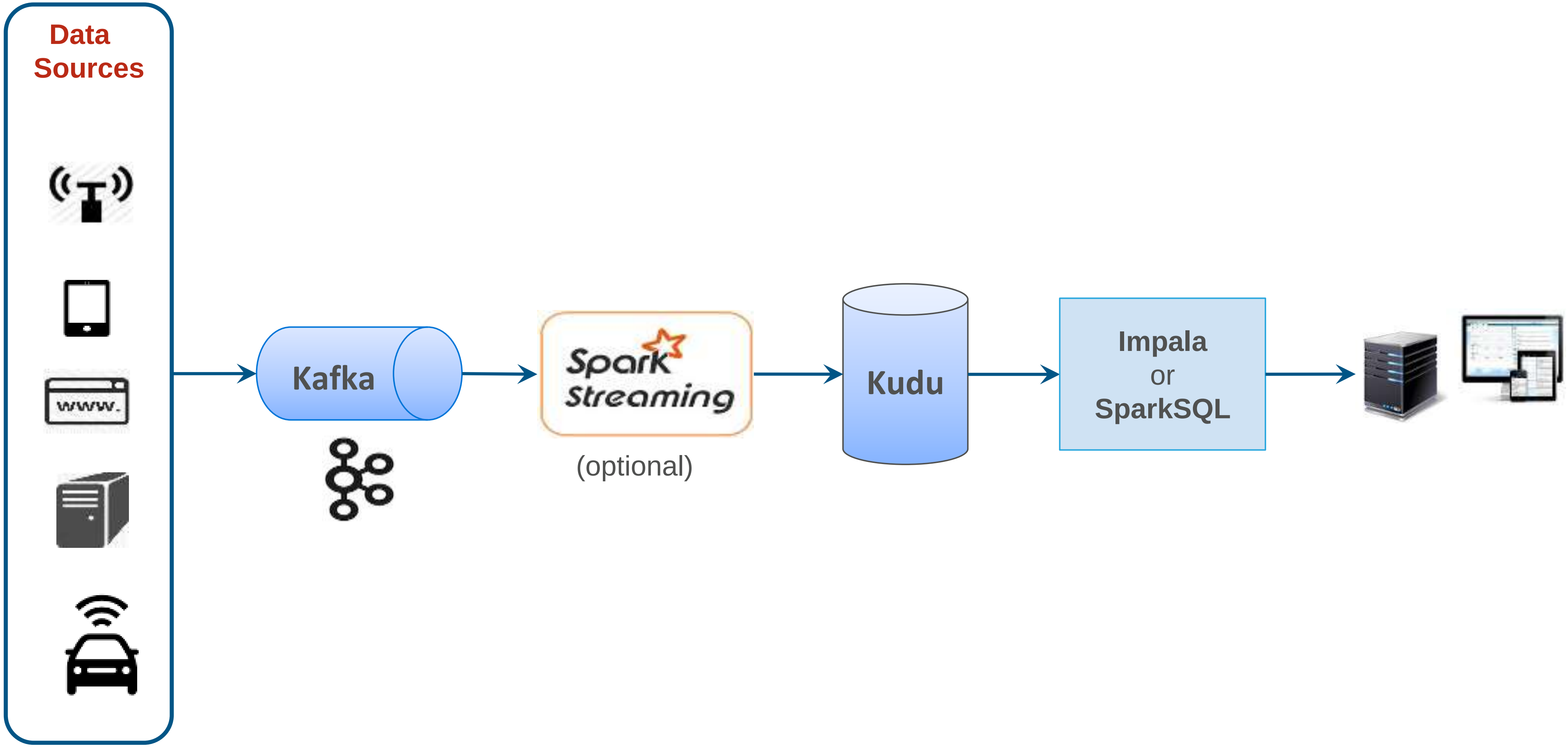
Low latency ETL pipeline

- ~10s data latency
- For apps that need to avoid direct backpressure or need ETL for record enrichment

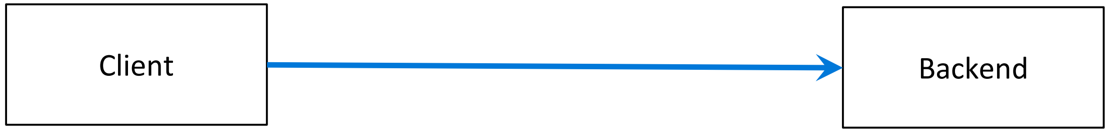
Direct zero-latency path

- For apps that can tolerate backpressure and can use the NoSQL APIs
- Apps that don’t need ETL enrichment for storage / retrieval

A modern, low-latency analytics architecture

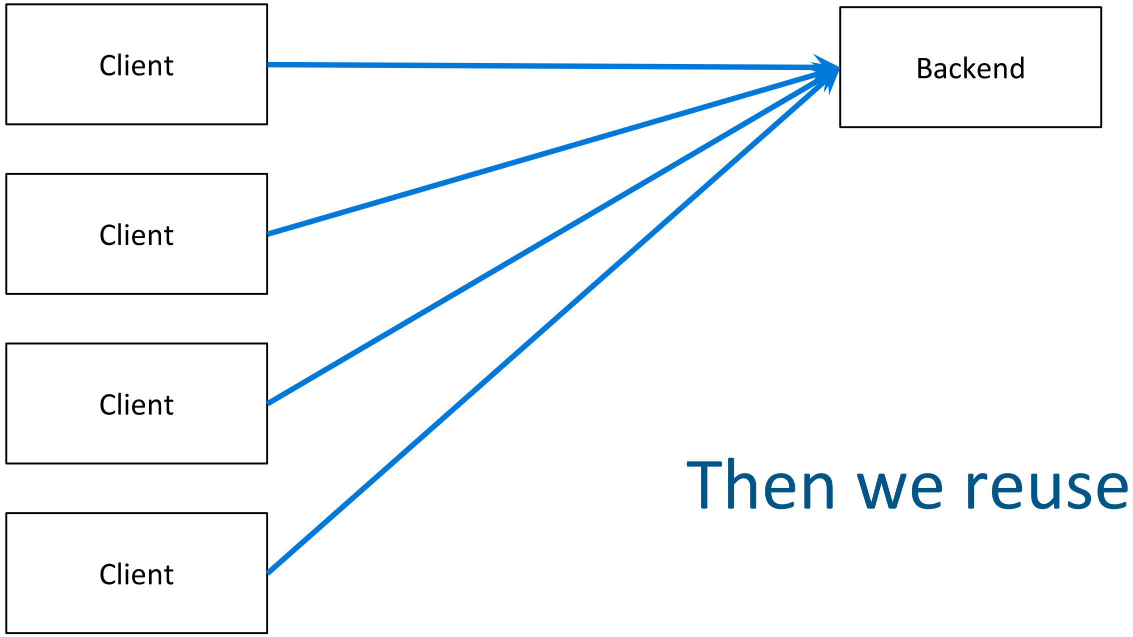


kafka

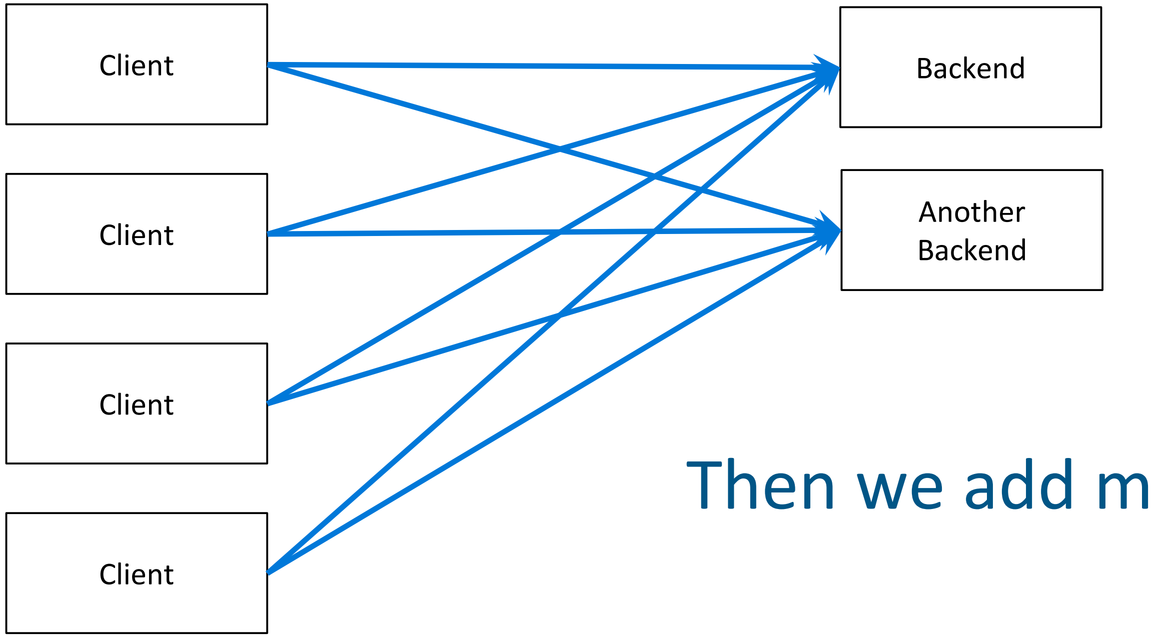


Data Pipelines Start like this.

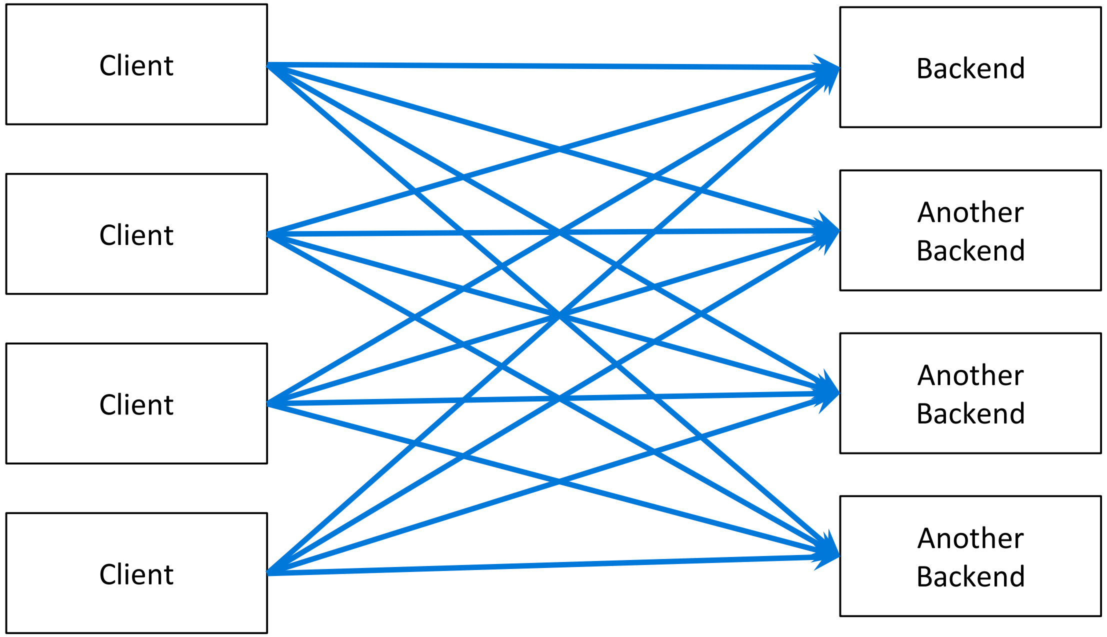
cloudera



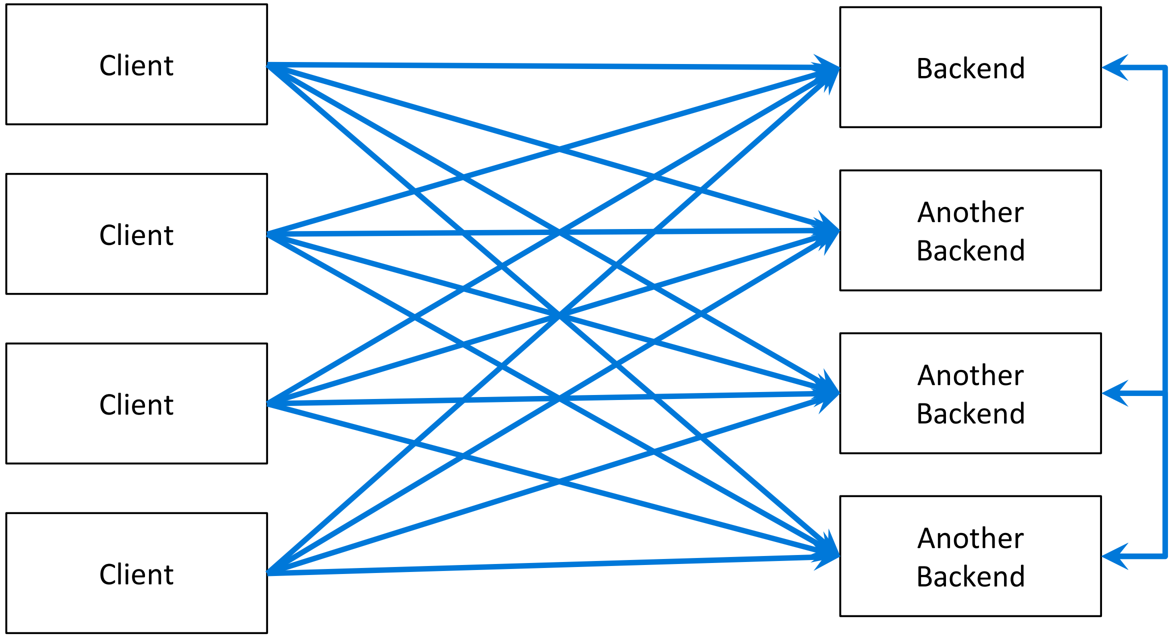
Then we reuse them



Then we add multiple backends



Then it starts to look like this



With maybe some of this

Adding applications should be easier

We need:

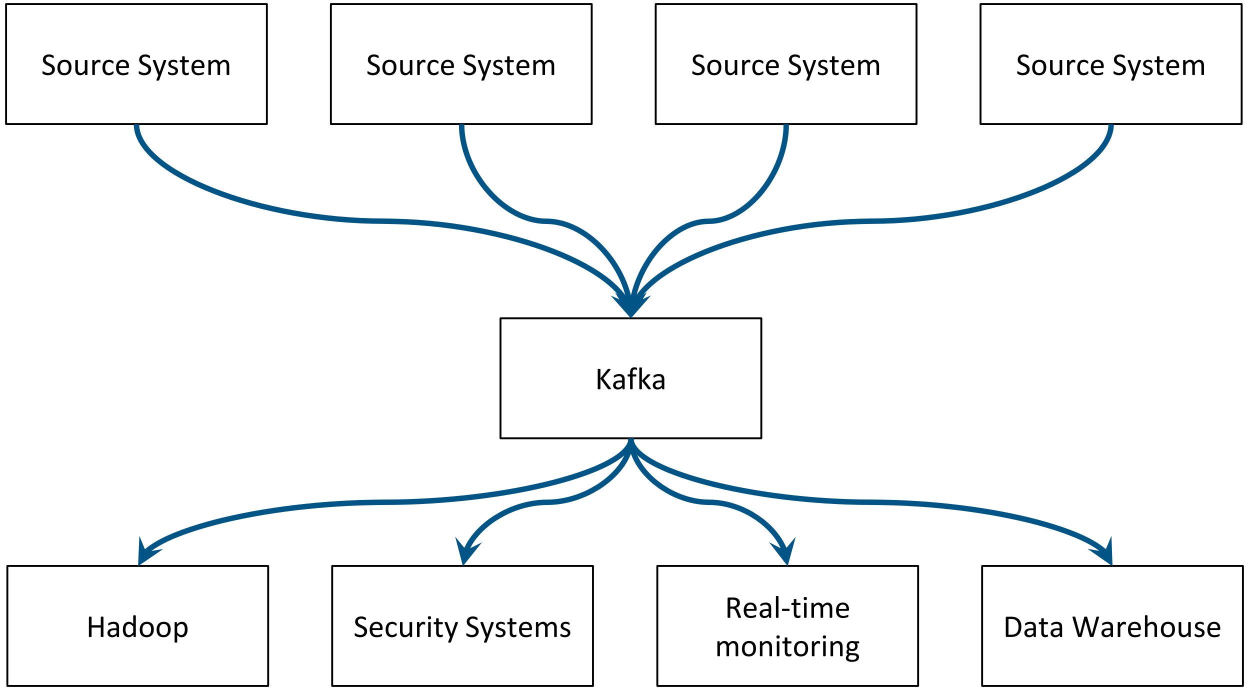
- Shared infrastructure for sending records
- Infrastructure must scale
- Set of agreed-upon record schemas

Why Kafka

Producers

Broker

Consumers



Kafka decouples data pipelines

About Kafka

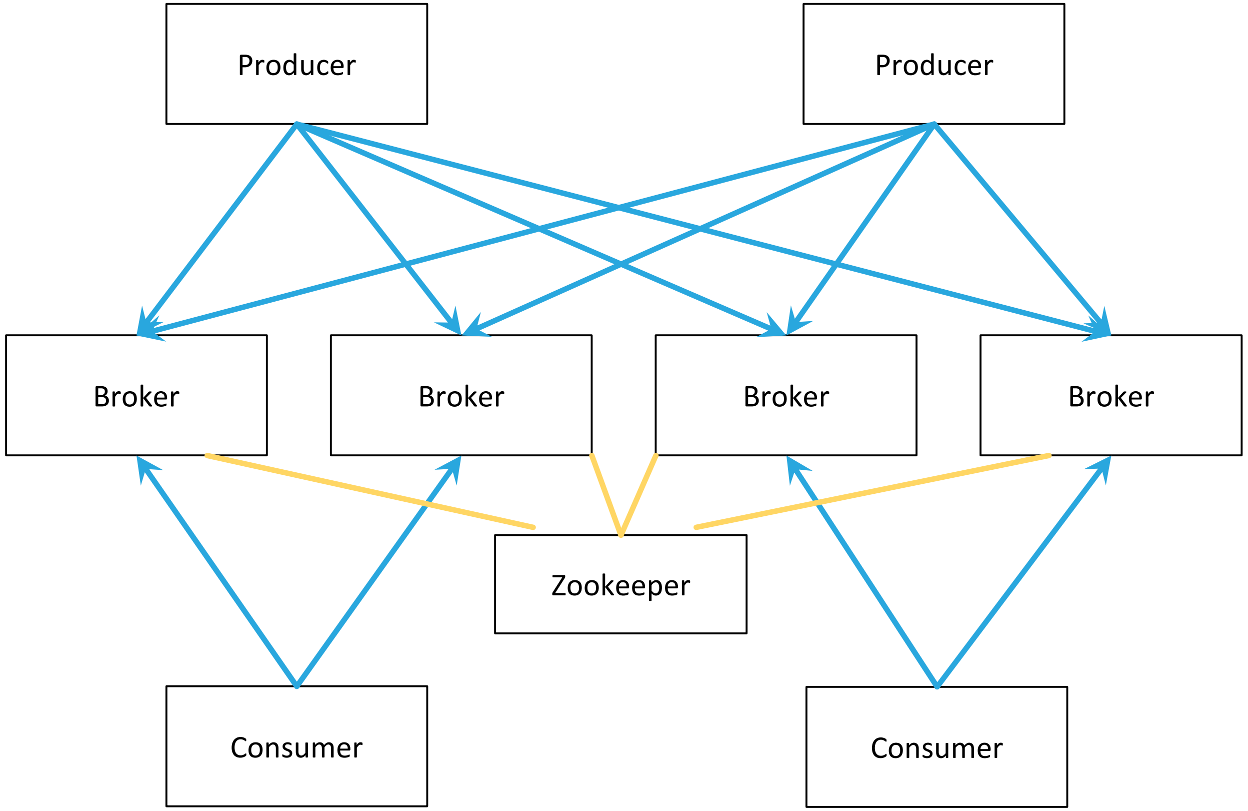
- Publish/Subscribe Messaging System From LinkedIn
- High throughput (100’s of k messages/sec)
- Low latency (sub-second to low seconds)
- Fault-tolerant (Replicated and Distributed)
- Supports Agnostic Messaging
- Standardizes format and delivery
- Huge community

Architecture

Producers

Kafka Cluster

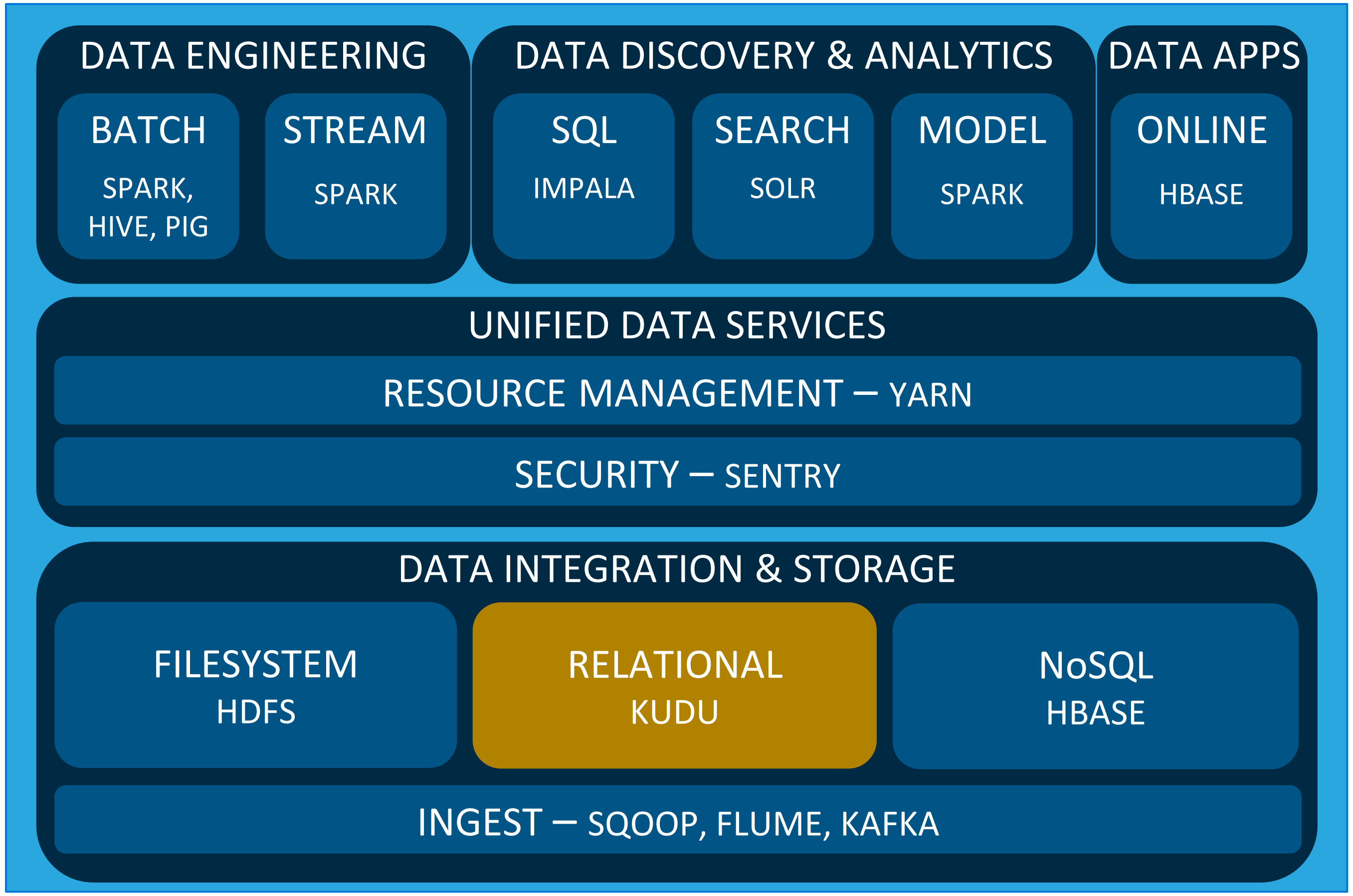
Consumers



KUDU

Kudu is a high-performance distributed storage engine

Storage for fast (low latency) analytics on fast (high throughput) data



- Simplifies the architecture for building analytic applications on changing data
- Optimized for fast analytic performance
- Natively integrated with the Hadoop ecosystem of components

Kudu: Scalable and fast tabular storage

Scalable

- Tested up to 275 nodes (~3PB cluster)
- Designed to scale to 1000s of nodes and tens of PBs

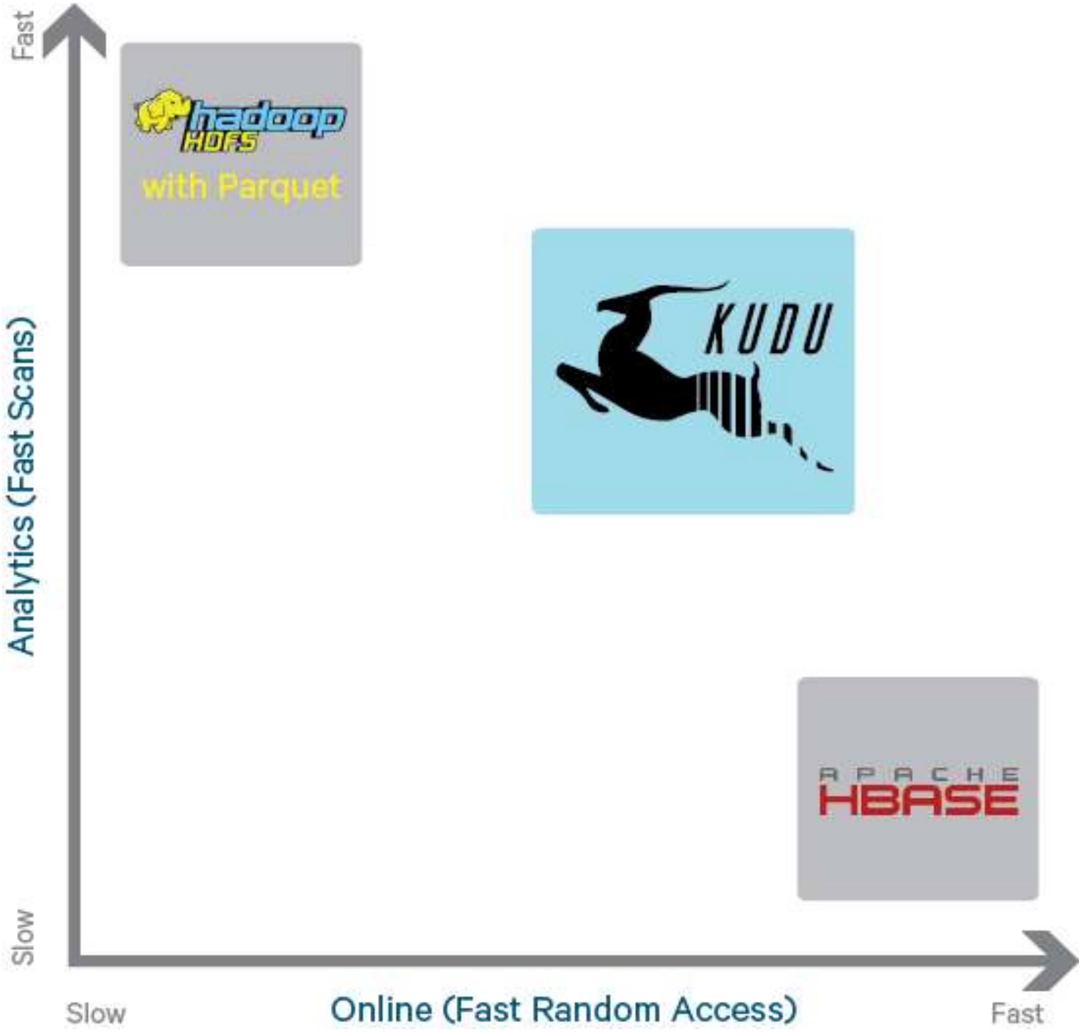
Fast

- Millions of read/write operations per second across cluster
- Multiple GB/second read throughput per node

Tabular

- Store tables like a normal database
- Individual record-level access to 100+ billion row tables

Kudu design goals



- High throughput for big scans
  Goal: Within 2x of Parquet

- Low-latency for short accesses
  Goal: 1ms read/write on SSD

- Database-like semantics
  Initially, single-row atomicity

- Relational data model
  - SQL queries should be natural and easy
  - Include NoSQL-style scan, insert, and update APIs

Kudu storage system interfaces

- A Kudu table has a SQL-like schema
  - And a finite number of columns (unlike HBase/Cassandra)
  - Types: BOOL, INT8, INT16, INT32, INT64, FLOAT, DOUBLE, STRING, BINARY, TIMESTAMP
  - Some subset of columns makes up a possibly-composite primary key
  - Fast ALTER TABLE
- Java, C++, and Python NoSQL-style APIs
  - Insert(), Update(), Delete(), Scan()
- Integrations with Kafka, MapReduce, Spark, Flume, and Impala
  - Apache Drill work-in-progress

Kudu use cases

Kudu is best for use cases requiring:

- Simultaneous combination of sequential and random reads and writes
- Minimal to zero data latencies

Time series

- Examples: Streaming market data, fraud detection / prevention, risk monitoring
- Workload: Insert, updates, scans, lookups

Machine data analytics

- Example: Network threat detection
- Workload: Inserts, scans, lookups

Online reporting / data warehousing

- Example: Operational data store (ODS)
- Workload: Inserts, updates, scans, lookups

Tables and tablets

- Each table is horizontally partitioned into tablets
  - *Range* or *hash* partitioning
    - `PRIMARY KEY (host, metric, timestamp) DISTRIBUTE BY HASH(timestamp) INTO 100 BUCKETS`
    - Translation: `bucketNumber = hashCode(row[‘timestamp’]) % 100`
- Each tablet has N replicas (3 or 5), kept consistent with Raft consensus
- Tablet servers host tablets on local disk drives

Metadata

- Replicated master
  - Acts as a tablet directory
  - Acts as a catalog (which tables exist, etc)
  - Acts as a load balancer (tracks TS liveness, re-replicates under-replicated tablets)
- Caches all *metadata* in RAM for high performance
- Client configured with master addresses
  - Asks master for tablet locations as needed and caches them

Impala integration

- `CREATE TABLE … DISTRIBUTE BY HASH(col1) INTO 16 BUCKETS AS SELECT … FROM …`
- `INSERT/UPDATE/DELETE`

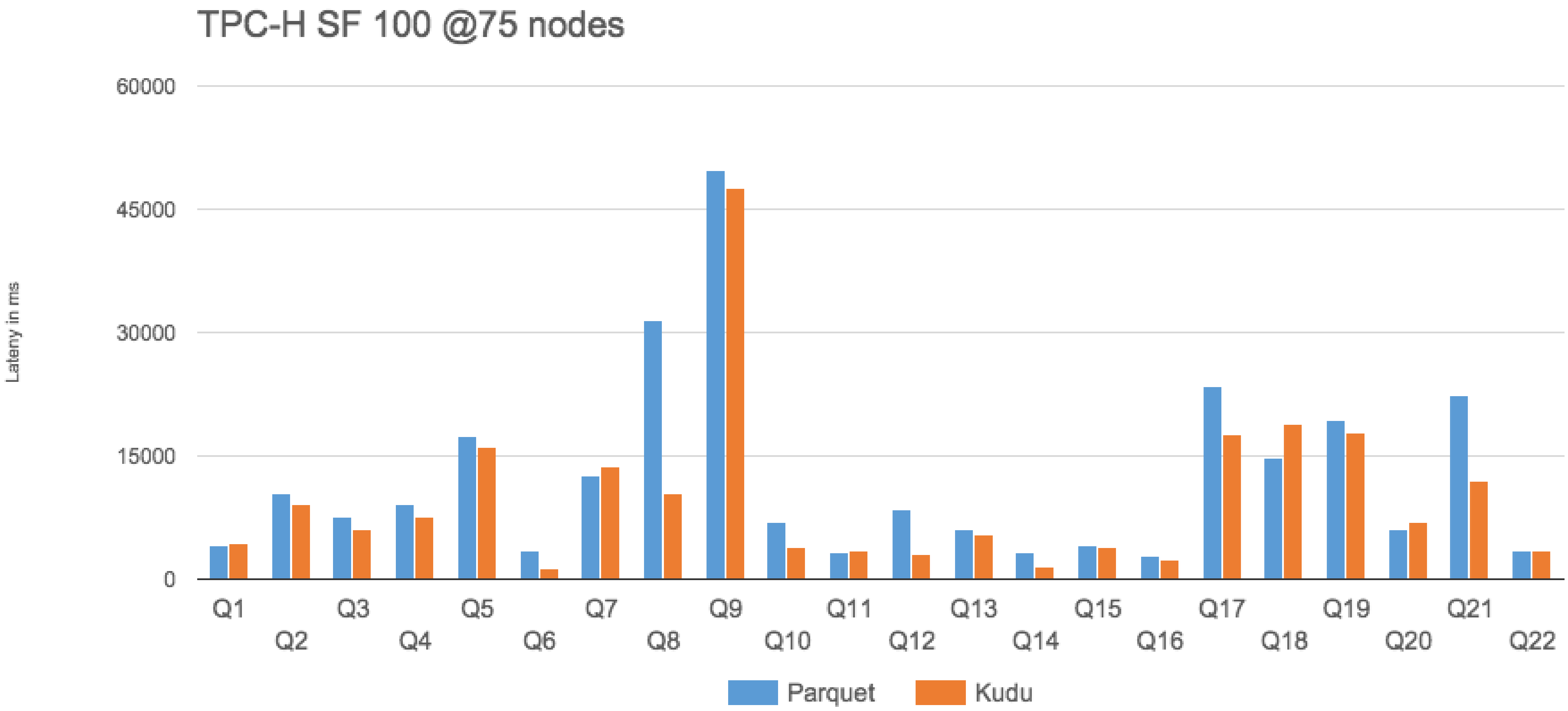
- Optimizations like predicate pushdown, scan parallelism, plans for more on the way

Spark DataSource integration

```
sqlContext.load("org.kududb.spark",
     Map("kudu.table" -> “foo”,
         "kudu.master" -> “master.example.com”))
   .registerTempTable(“mytable”)
df = sqlContext.sql(
   “select col_a, col_b from mytable “ +
   “where col_c = 123”)
```

TPC-H (analytics benchmark)

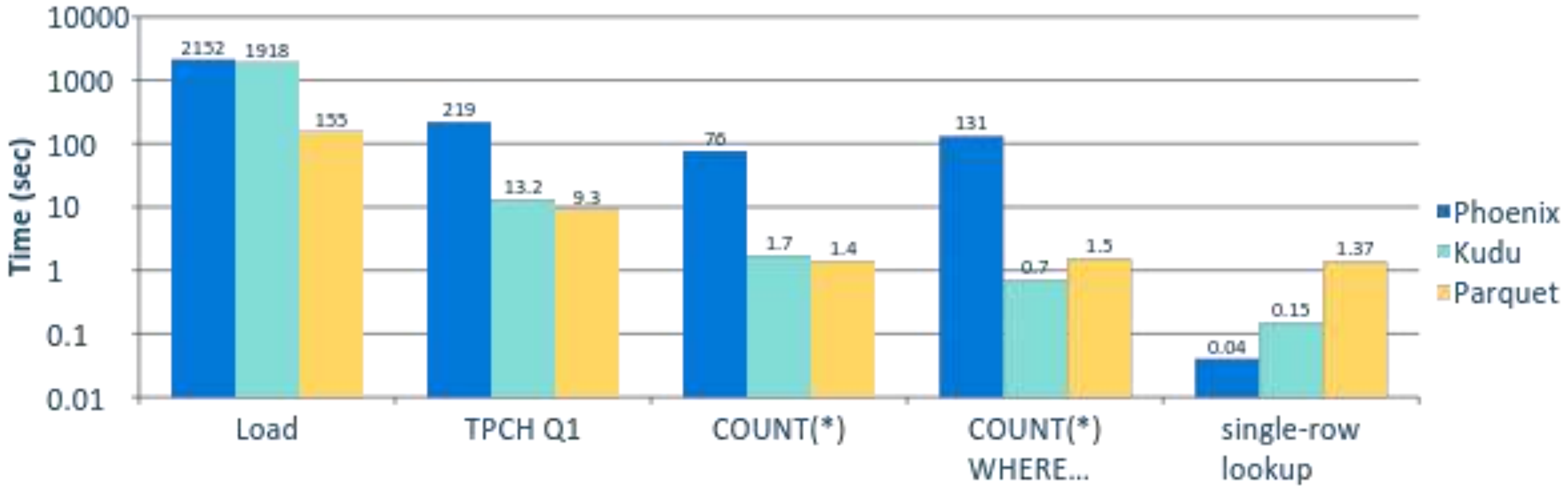
- 75 server cluster
  - 12 (spinning) disks each, enough RAM to fit dataset
  - TPC-H Scale Factor 100 (100GB)
- Example query:
  - `SELECT n_name, sum(l_extendedprice * (1 - l_discount)) as revenue FROM customer, orders, lineitem, supplier, nation, region WHERE c_custkey = o_custkey AND l_orderkey = o_orderkey AND l_suppkey = s_suppkey AND c_nationkey = s_nationkey AND s_nationkey = n_nationkey AND n_regionkey = r_regionkey AND r_name = 'ASIA' AND o_orderdate >= date '1994-01-01' AND o_orderdate < '1995-01-01’ GROUP BY n_name ORDER BY revenue desc;`



- Kudu outperforms Parquet by 31% (geometric mean) for RAM-resident data

Versus other NoSQL storage

- **Apache Phoenix: OLTP SQL engine built on HBase**
- 10 node cluster (9 worker, 1 master)
- TPC-H LINEITEM table only (6B rows)



cloudera

Getting Data from Kafka into Kudu

- Custom client, i.e., Kafka consumer that writes to Kudu

Getting Data from Kafka into Kudu

- Custom client, i.e., Kafka consumer that writes to Kudu
- Kafka-Flume source/channel + Kudu-Flume sink

Getting Data from Kafka into Kudu

- Custom client, i.e., Kafka consumer that writes to Kudu
- Kafka-Flume source/channel + Kudu-Flume sink
- Kafka connect

Kafka + Kudu: A low latency data visibility path

- Upstream application pushes data to Kafka
- Kafka then acts as a buffer in order to handle backpressure from Kudu
- The Kafka Connect plugin pushes data to Kudu as it becomes available
- As soon as the data is ingested into Kudu, it becomes available

Tradeoffs

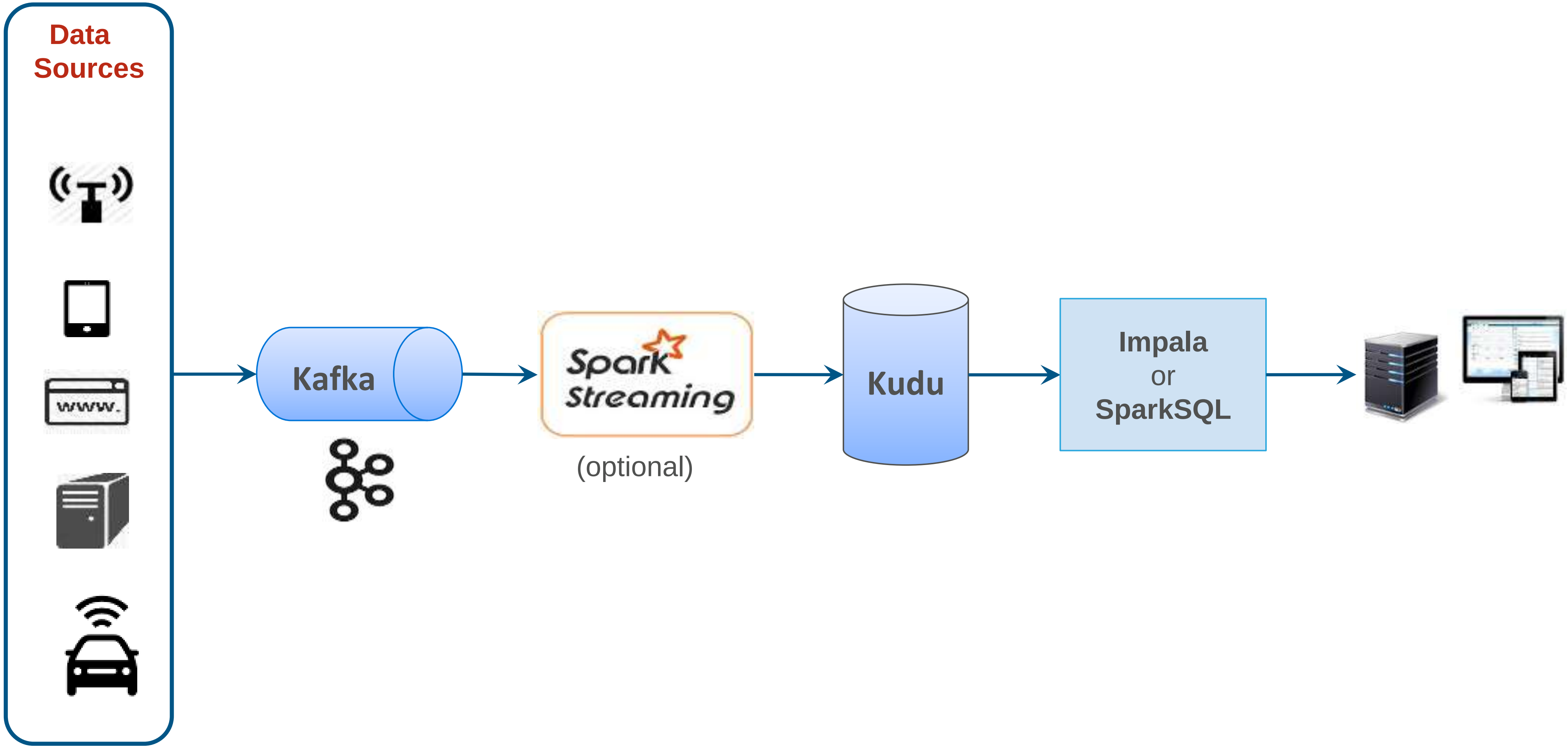
What if I need a zero-latency path?

- Possible to write to Kudu directly using the NoSQL API
- However, the app will need to tolerate queueing and backpressure itself

What if I want to store unstructured data or large binary blobs?

- Consider using Kafka + HBase instead of Kudu
    - But you won’t get the same SQL query performance

Demo



cloudera

About the Kudu project

- Apache Software Foundation incubating project
- Latest version 0.8.0 (beta) released in April
- Plans are for a 1.0 version to be released in August
- Web site: getkudu.io (also kudu.incubator.apache.org soon)
- Slack chat room for devs and users (auto-invite): getkudu-slack.herokuapp.com
- Twitter handle: @ApacheKudu
- Code: github.com/apache/incubator-kudu

Want to hear more about Kudu and Spark?

- Come to the Vancouver Spark meetup tonight here at the Hyatt at 6pm
- More info: www.meetup.com/Vancouver-Spark/

cloudera

Thank you

cloudera

Questions?

Mike Percy | @mike_percy
mpercy@cloudera.com

Ashish Singh | @singhasdev
asingh@cloudera.com