
ONEWAY
SOLUTION

One Way Solution

The Processors

Data Engineering – [Day 3]

LUAN MORENO

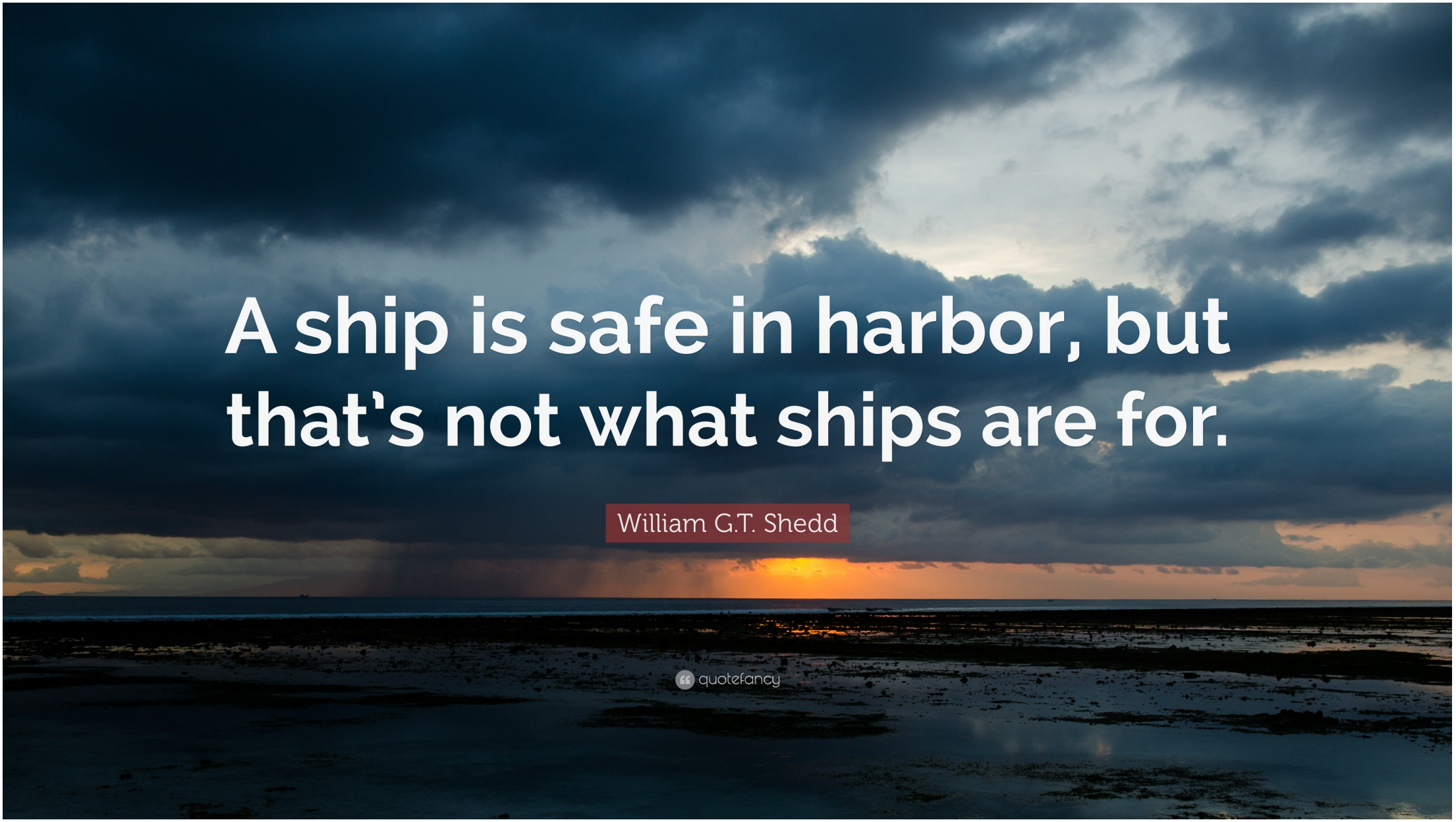
CEO & Data Architect

Data Engineer & MVP

MATEUS OLIVEIRA

Big Data Architect

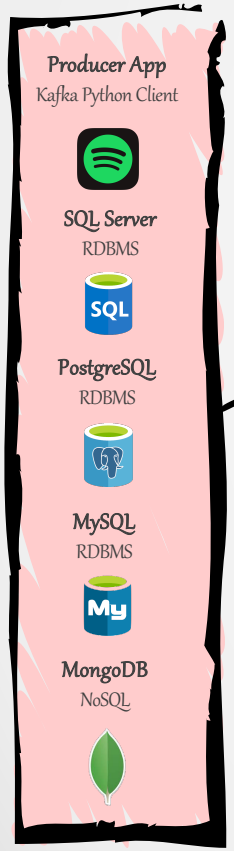
Data In-Motion Specialist


A ship is safe in harbor, but that's not what ships are for.
William G.T. Shedd
quotefancy

Use Case ~ [Data Processing]

Producers

sending data in

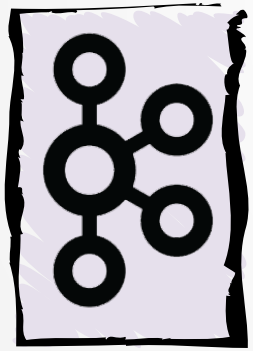


Streams

process data inside of

EDH

enterprise data hub



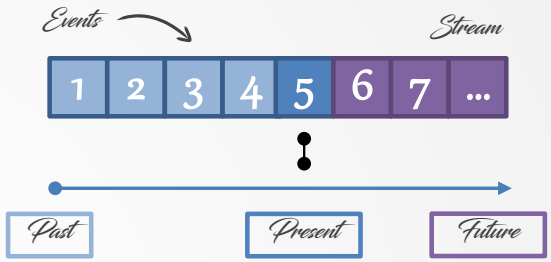

With enough courage, you can do without a reputation.
Margaret Mitchell
quotefancy

Event Stream [ES]

Event Stream [ES] = Representation of an Unbounded DataSet
Unbounded = Infinite & Ever Growing

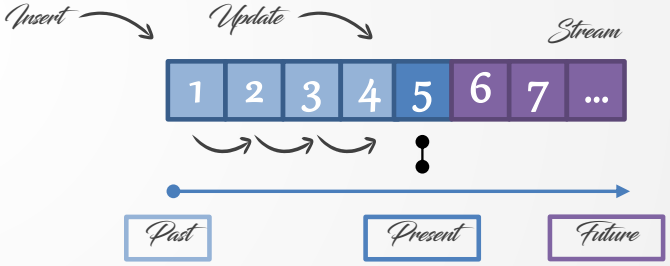
Event Streams [Ordered]

There is an inherent notion of which events occur before or after other events. This is clearest when looking at financial events. A sequence in which I first put money in my account and later spend the money is very different from a sequence at which I first spend the money and later cover my debt by depositing money back. The latter will incur overdraft charges while the former will not. Note that this is one of the differences between an event stream and a database table records in a table are always considered unordered and the order by clause of SQL is not part of the relational model.



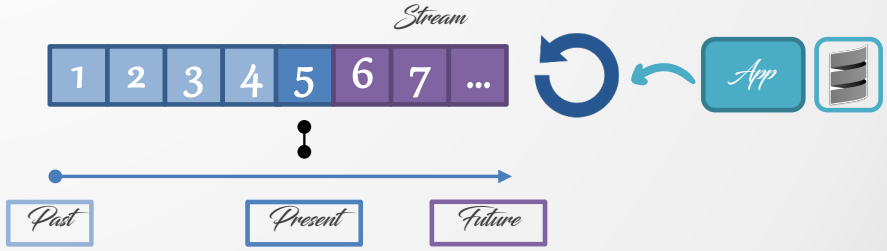
Immutable Data Records

Events, once occurred, can never be modified. A financial transaction that is cancelled does not disappear. Instead, an additional event is written to the stream, recording a cancellation of previous transaction. When a customer returns merchandise to a shop, we don't delete the fact that the merchandise was sold to him earlier, rather we record the return as an additional event. This is another difference between a data stream and a database table. We can delete or update records in a table, but those are all additional transactions that occur in the database, and as such can be recorded in a stream of events that records all transactions. If you are familiar with binlogs, WALs, or redo logs in databases you can see that if we insert a record into a table and later delete it, the table will no longer contain the record, but the redo log will contain two transactions - the insert and the delete.

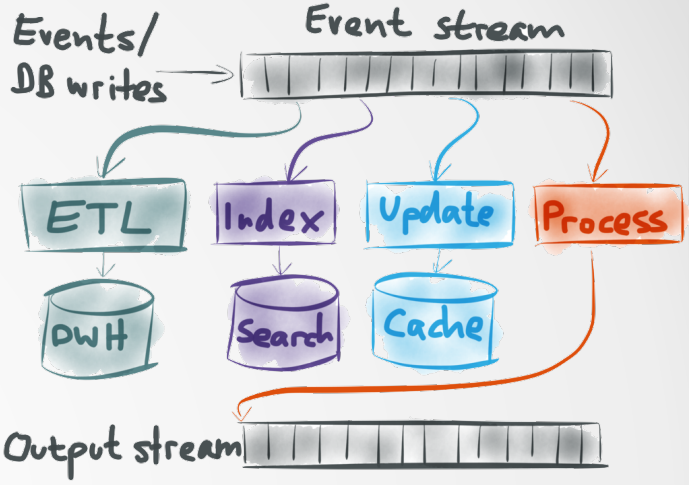


Event Streams [Replayable]

This is a desirable property. While it is easy to imagine nonreplayable streams (TCP packets streaming through a socket are generally nonreplayable), for most business applications, it is critical to be able to replay a raw stream of events that occurred months (and sometimes years) earlier. This is required in order to correct errors, try new methods of analysis, or perform audits. This is the reason we believe Kafka made stream processing so successful in modern businesses—it allows capturing and replaying a stream of events. Without this capability, stream processing would not be more than a lab toy for data scientists.



Event Stream Processing [ESP]

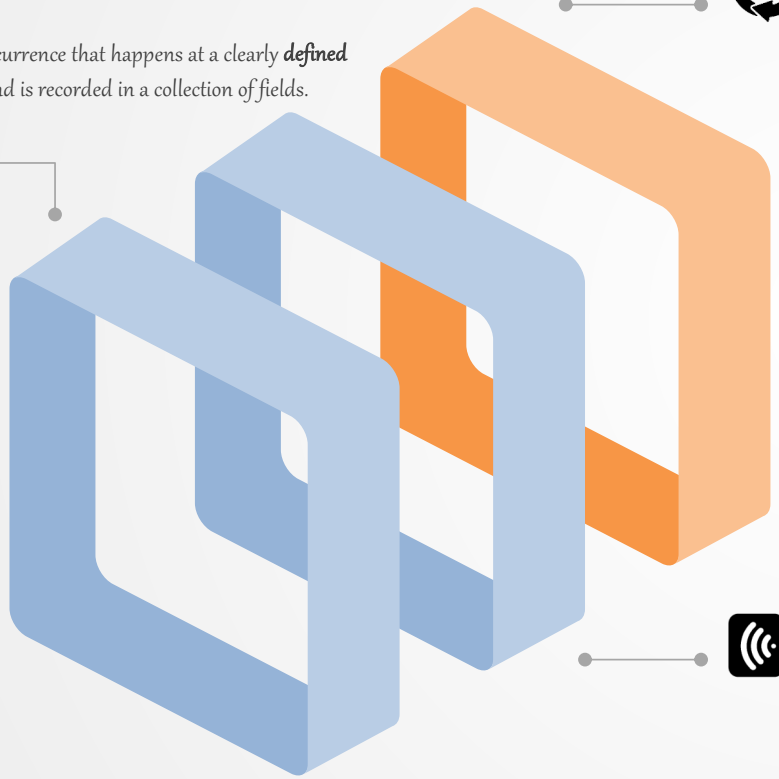


Event

any occurrence that happens at a clearly **defined time** and is recorded in a collection of fields.

Processing

the act of **analyzing data** and perform data analytics on top of that.



Stream

constant **flow of data** events, or a steady rush of data that flows into and around your business.

ESP for Processing Changes

- Store Data Reliability
- Process Incoming Data
- Perform Queries & Analytics
- Push Results Immediately to Subscribers

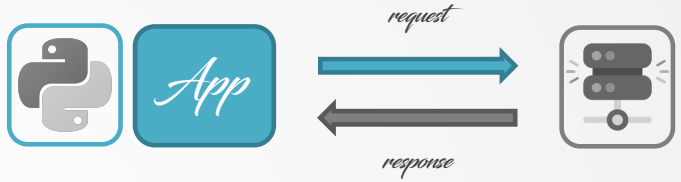
Stream Processing [Understanding]

Event Stream [ES] = Representation of an Unbounded DataSet
Unbounded = Infinite & Ever Growing
Event Stream Processing = Process Unbounded Datasets

Request-Response

this is **the lowest latency** paradigm, with response times ranging from submilliseconds to few milliseconds, usually with the expectation that response times will be highly consistent. for database is know as online transaction processing (OLTP).

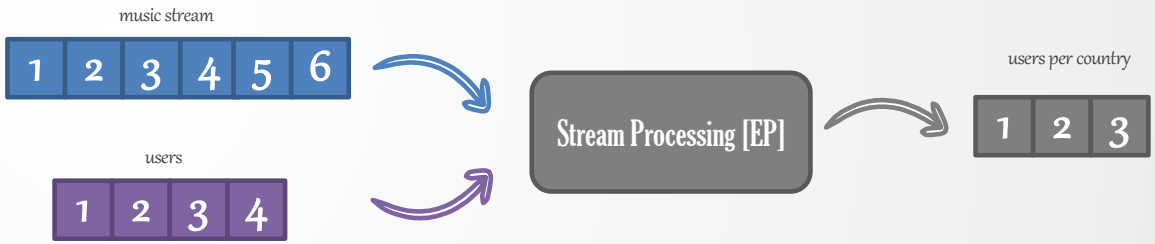
Used for = Point-of-Sales, Credit Card Processing, and Time Tracking



Stream-Processing

this is the option that **fills the gap between** the request-response and the batch-processing. Most of business do not require an immediate response within milliseconds but can't wait 24 hours either. actual business, process data continuously.

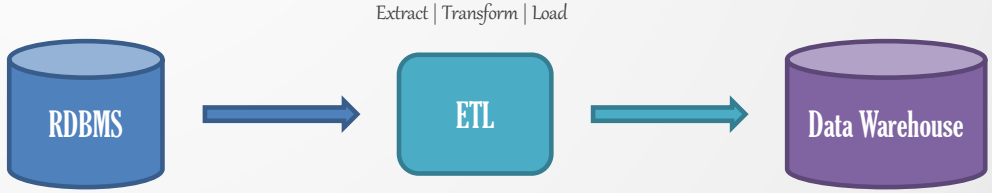
Used for = Credit Card Transactions, Supply & Demand, Real-Time ETL



Batch-Processing

This is the **high-latency & high-throughput** option. reads large datasets, apply business logics and writes into a destination. it runs in a schedule basis.

Used for = Data Warehouse and Business Intelligence Systems



Stream Processing [Concepts]

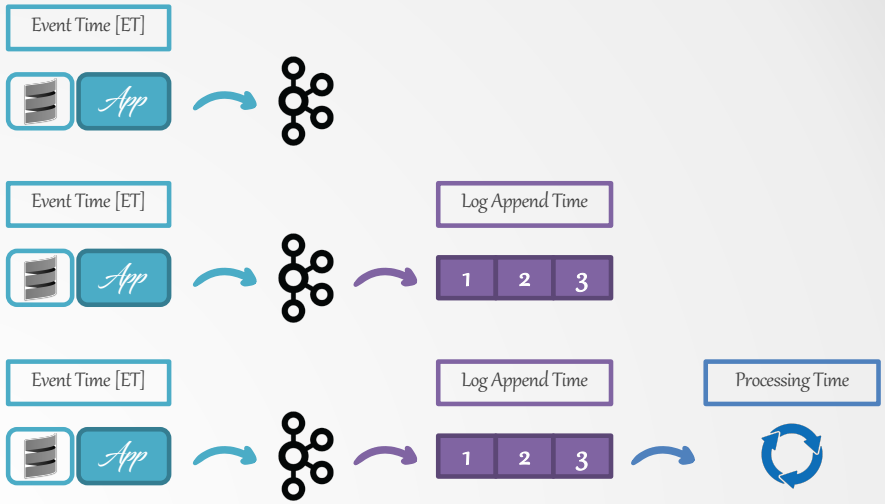
Time

operations occurs in a time window

Event-Time = time that the event we are tracking occurred and the record was created. Kafka automatically adds the current time to producer records at the time they are created

Log Append Time = time that the event arrived to the kafka broker and was stored there. Kafka brokers will automatically add this time to records they receive if kafka is configured

Processing Time = time at which stream-processing application received the event in order to perform some calculation, this can be in milliseconds, hours, days after the event occurred

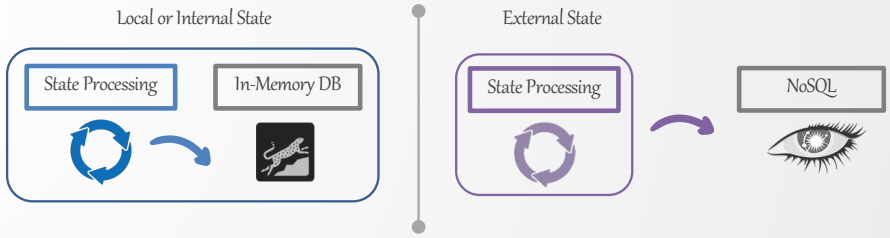


State

process each event individually, stream processing is a very simple activity

Local or Internal State = accessible only by a specific instance of the stream processing application. usually maintained and managed with an embedded, in-memory database running within the application

External State = maintained by an external datastore, often a NoSQL system. ban be queried from different applications at the same time. increases latency and complexity (often avoid this approach)

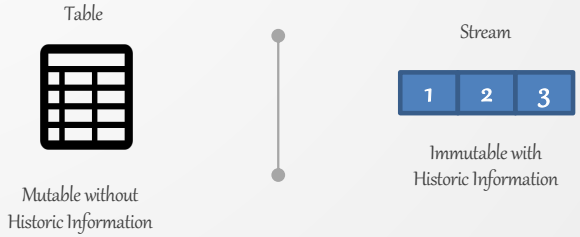


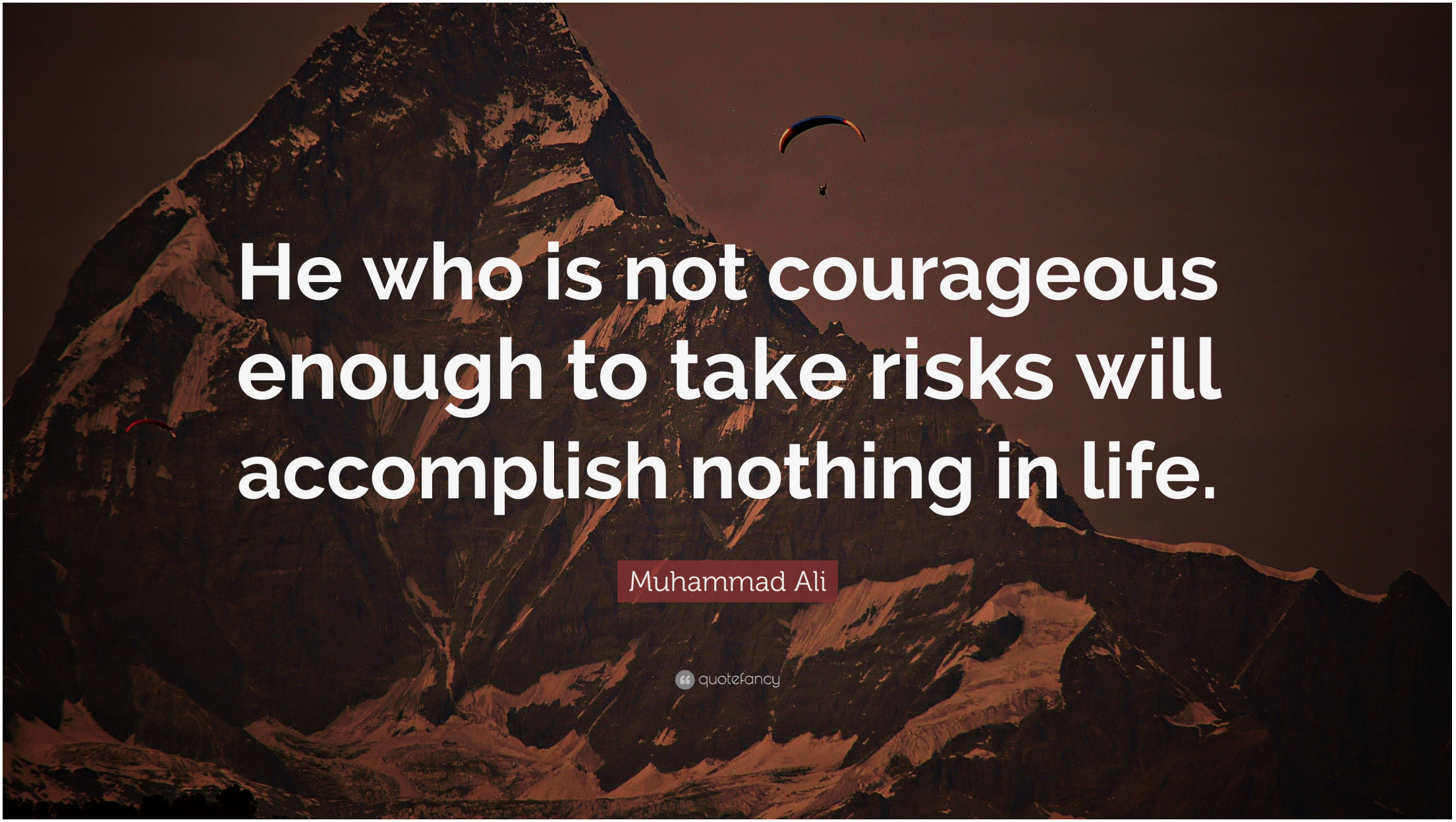
Streams-Table Duality

stream processing engines needs to provide streams and tables options

Tables = records are mutable, often does not offer historic information and stores the current state (snapshot)

Streams = unbounded dataset, immutable by nature and does provide historical information with a timestamp attached on it




He who is not courageous enough to take risks will accomplish nothing in life.
Muhammad Ali
quotefancy

Real-Time Stream Processing [Engines]

Open-Source Platform [OSS]

- Apache Kafka
- Apache Spark
- Apache Apex
- Apache Flink
- Apache Storm
- Apache Beam

Microsoft Azure

- HDInsight
- Azure Synapse Analytics
- Azure Stream Analytics
- Azure Functions

Google Cloud Platform [GCP]

- Cloud DataProc
- Cloud DataFlow
- Cloud Functions

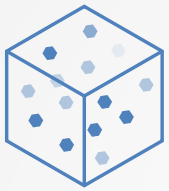
Amazon Web Services [AWS]

- Amazon EMR
- Amazon Kinesis Data Streams
- Amazon Kinesis Data Analytics
- AWS Lambda

StreamingSQL [Engines]

Apache Flink

Flink SQL
2016

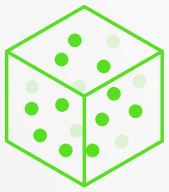


Apache Spark

Structured Streaming
2016

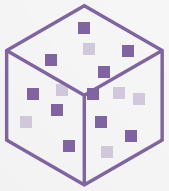
Apache Kafka

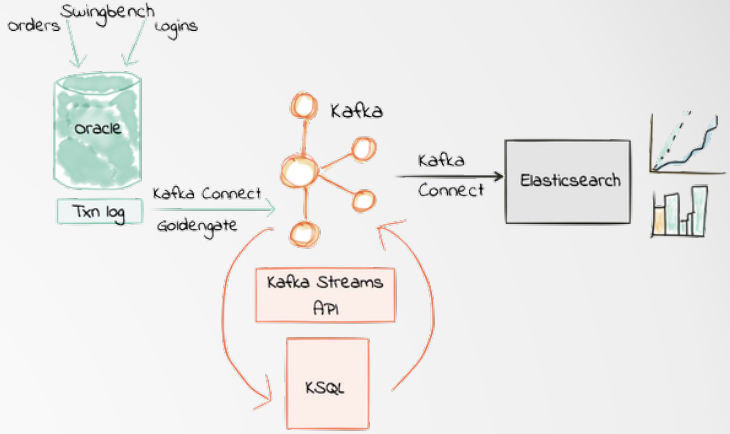
KSQL
2017



Apache Beam

Beam SQL
2017

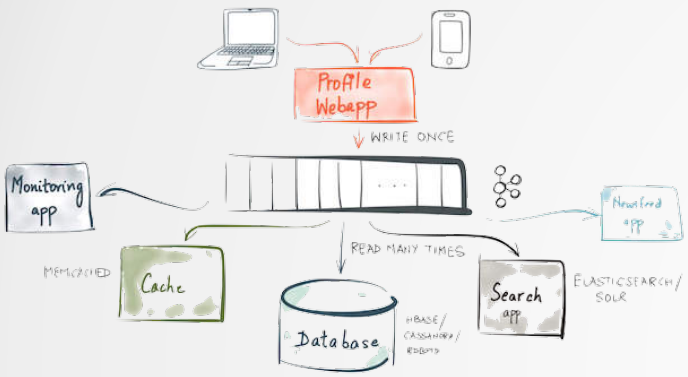




Data Processing

- Manipulate Streams
- Query over Continuous Flow of Data
- Select | Join | Union & Merge | Window & Aggregation
- Real-Time Analytics
- Predictions & ML

Python & Streaming Engines



Apache Flink

Table API for a Unified Stream & Batch Processing Experience

Table API

- Language-Integrated Query API for Scala & Java
- Batch & Streaming Input without Code Changes
- Fully Integrated with DataStream and DataSet APIs
- Integrated with Complex Event Processing API

Faust

Python Stream Processing Library

Faust API

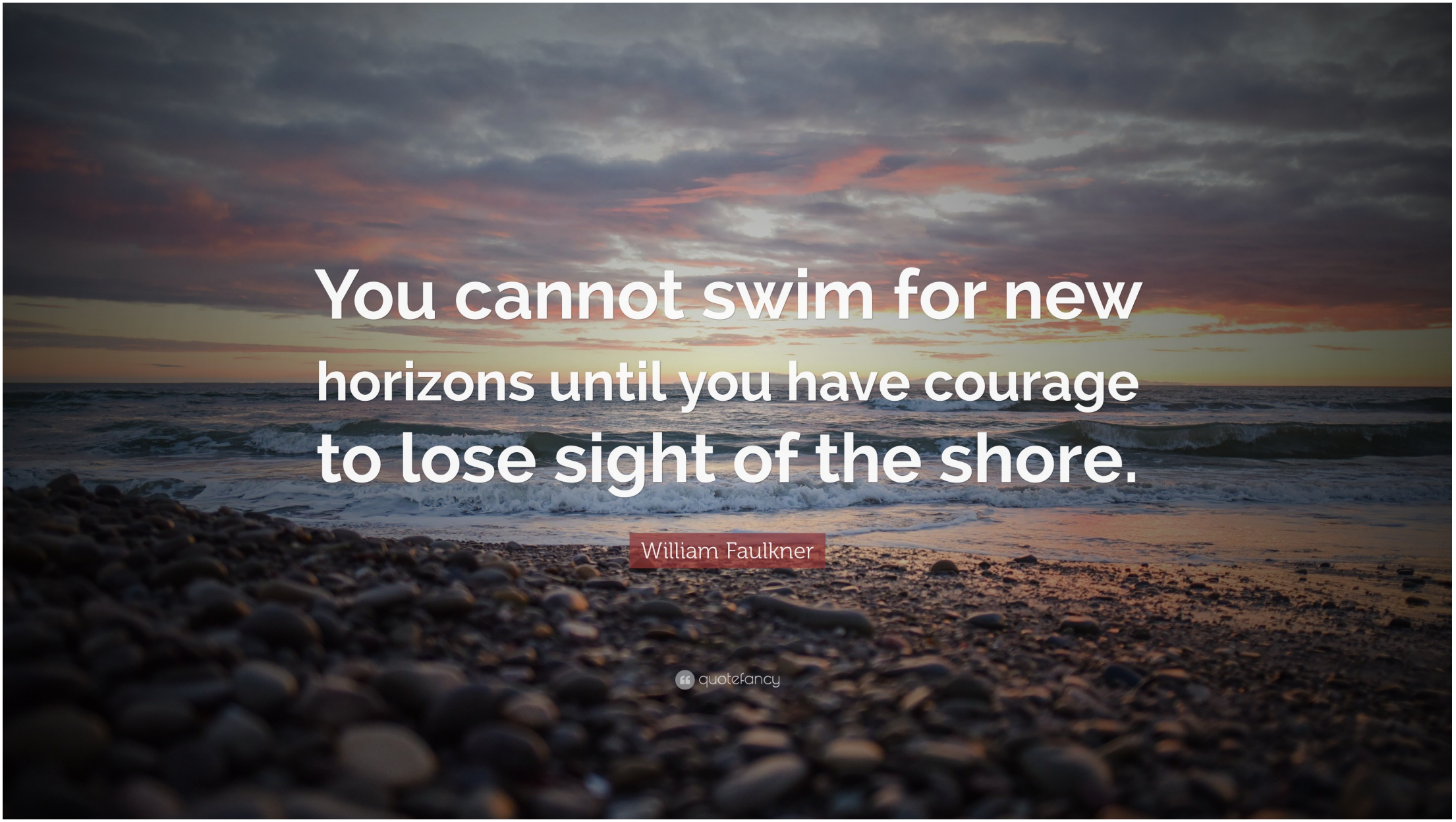
- Stream Processing ~ Kafka Streams, Spark, Storm, Samza, Flink
- Numpy, PyTorch, Pandas, NLTK, Django
- Tables ~ Distributed Key & Value
- Use Regular Python Dictionaries

Apache Spark

PySpark ~ Python API for Apache Spark Engine

PySpark API

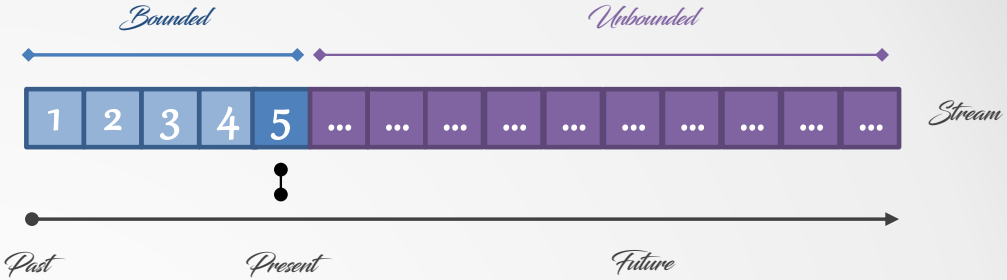
- PySpark ~ Spark DataFrame
- Distributed Table ~ Apache Spark Cluster
- SQL, Streaming, ML ~ Packages
- RDD, DStream & DataFrame


You cannot swim for new horizons until you have courage to lose sight of the shore.
William Faulkner
quotefancy

Apache Flink [Basics]

Apache Flink

- Stateful Computations ~ Bounded & Unbounded Data Streams
- Unified Stream & Batch Data Processing Capabilities
- Streaming Applications At Scale
- Guarantee Exactly-Once State Consistency

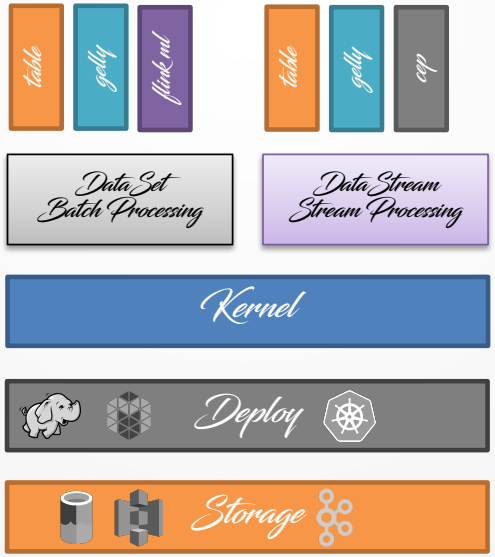


Ecosystem

founded by Data Artisans company and donated to Apache Software Foundation.

- DataSet API = Batch Operations
- DataStreams API = Stream Operations

- Table API = Relational using SQL Expressions
- CEP API = Detect Event Patterns
- Gelly API = Large Scale Graph Analysis
- ML API = Machine Learning Pipelines



APIs & Libraries ~ [Java, Scala, Python & SQL]

Distributed Streaming DataFlow

Standalone, Apache Yarn, Apache Mesos, Kubernetes., Google GCE, Amazon EC2, Azure VM

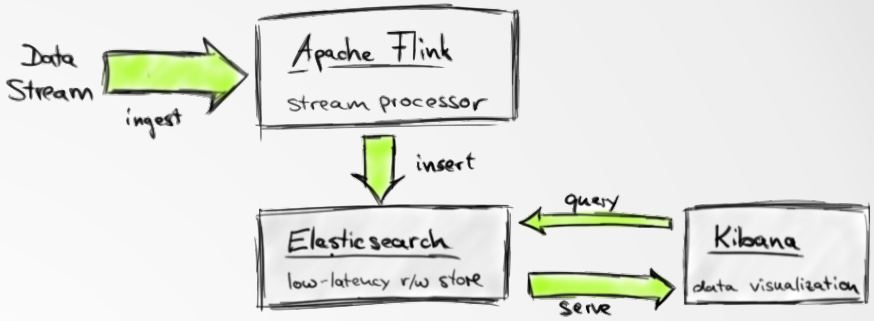
Local Filesystem, Apache HDFS, S3, MongoDB, HBase, RabbitMQ, Apache Kafka

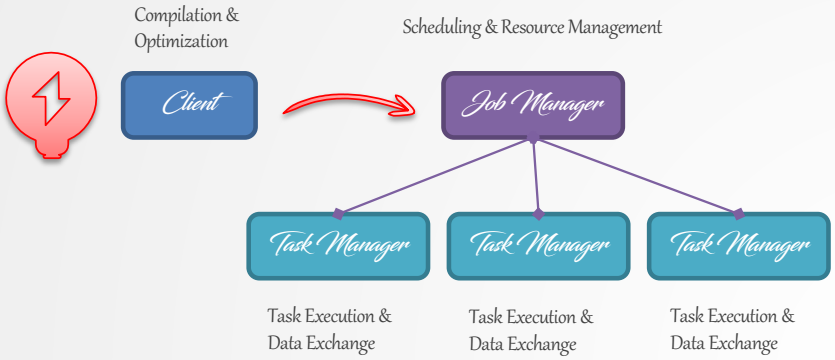
Trillions of Events per Day

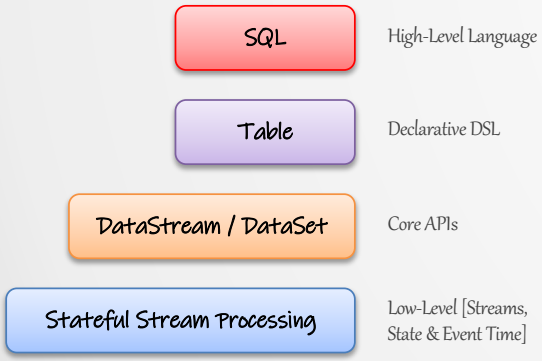
Multiple Terabytes of State

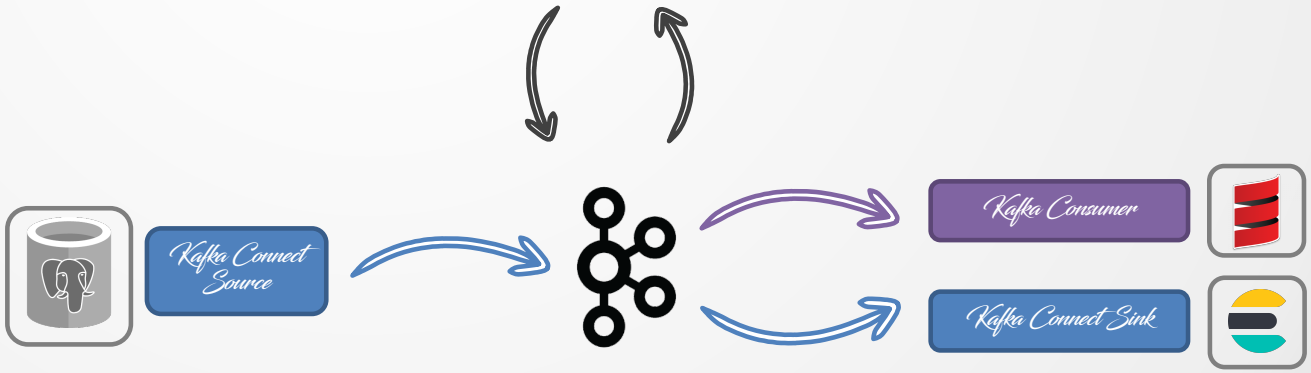
Running on Thousands of Cores

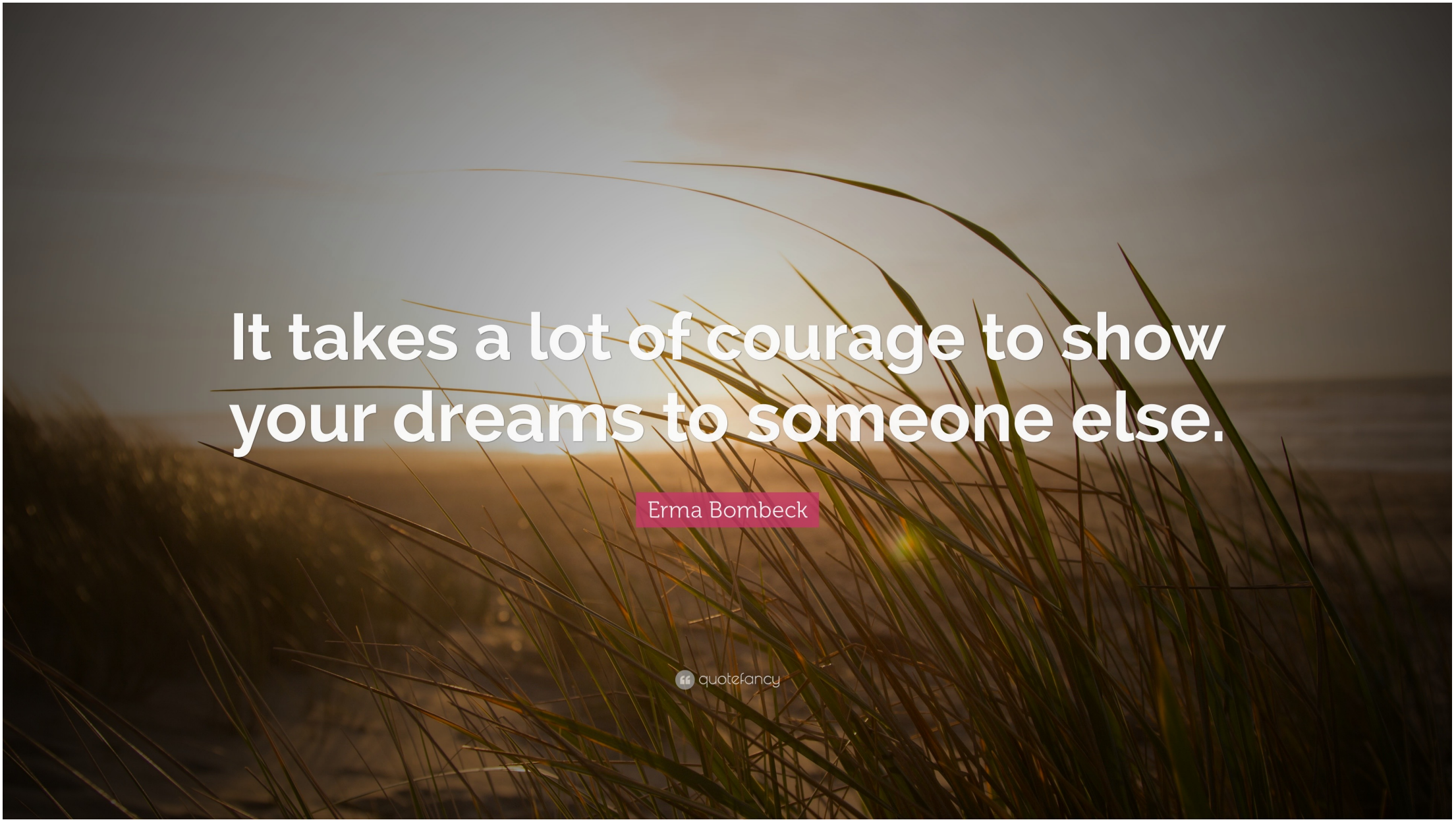
Apache Flink [Concepts]



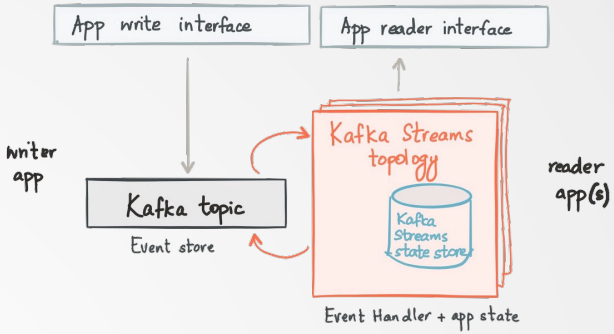







It takes a lot of courage to show your dreams to someone else.
Erma Bombeck
quotefancy

Kafka Streams [Basics]



Intro

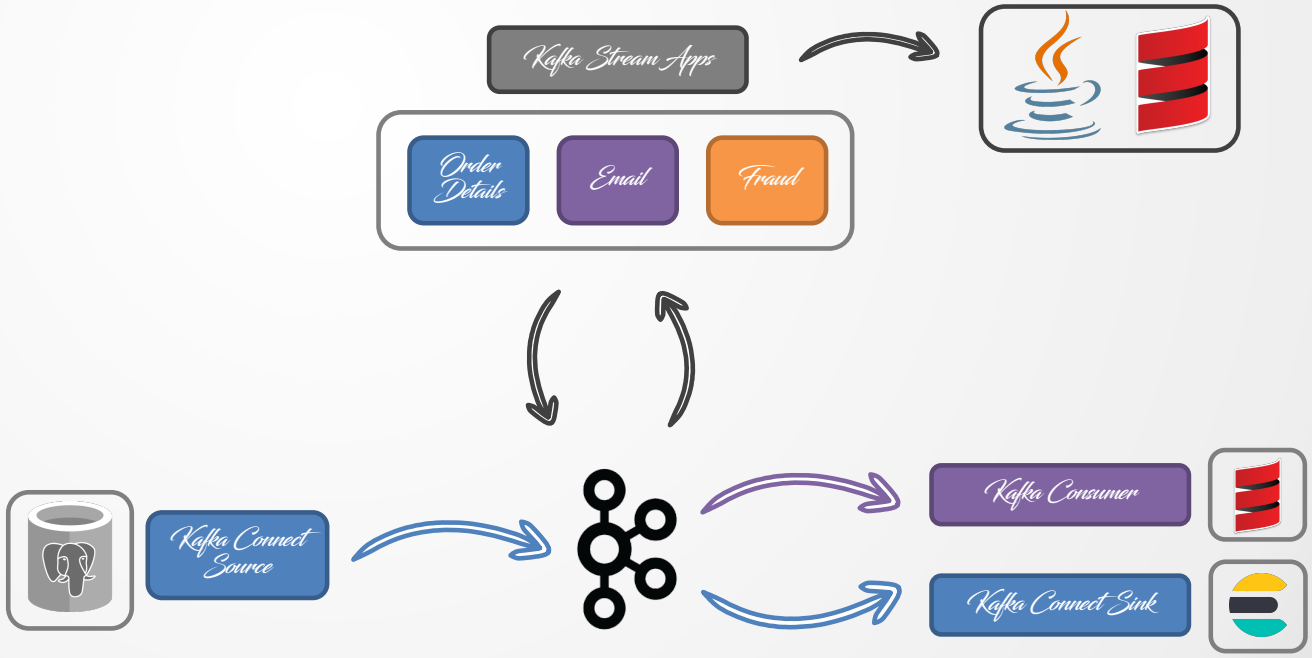
Mission-Critical Real-Time Applications and Microservices

is a client library for building applications and microservices, where the input and output data are stored in kafka clusters. it combines the simplicity of writing and deploying standard java and scala applications on the client side with the benefits of kafka's server-side cluster technology.

Characteristics

Kafka Streams Applications [App]

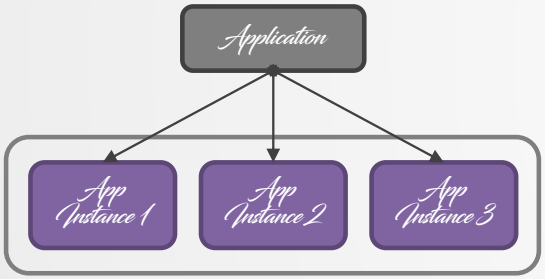
- Elastic, High Scalable & Fault-Tolerant
- Deploy ~ Containers & VMs
- Small, Medium & Large Use-Cases
- Fully Integrated with Kafka Security
- Exactly-Once Processing Semantics
- One-Record-At-a-Time Processing
- High-Level Streams DSL
- Low-Level Processor API
- Decoupled Processing Engine [JVM]
- Develop ~ Mac, Linux & Windows



Kafka Streams [Concepts]

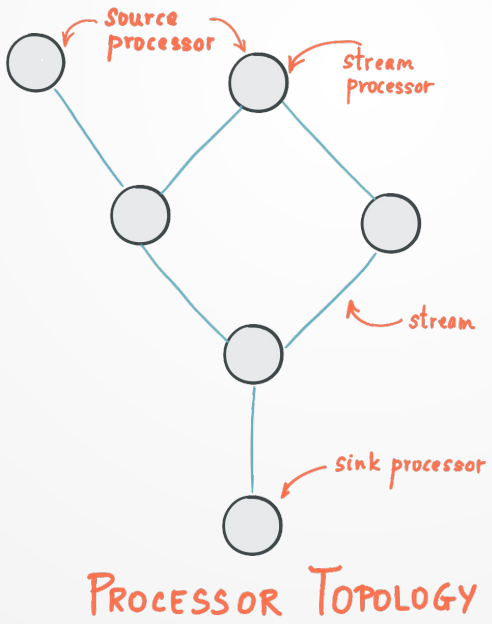
Application Instances

is any running instance "copy" of your application.
elastically scale & parallelize for fault-tolerant purpose.



Processor Topology

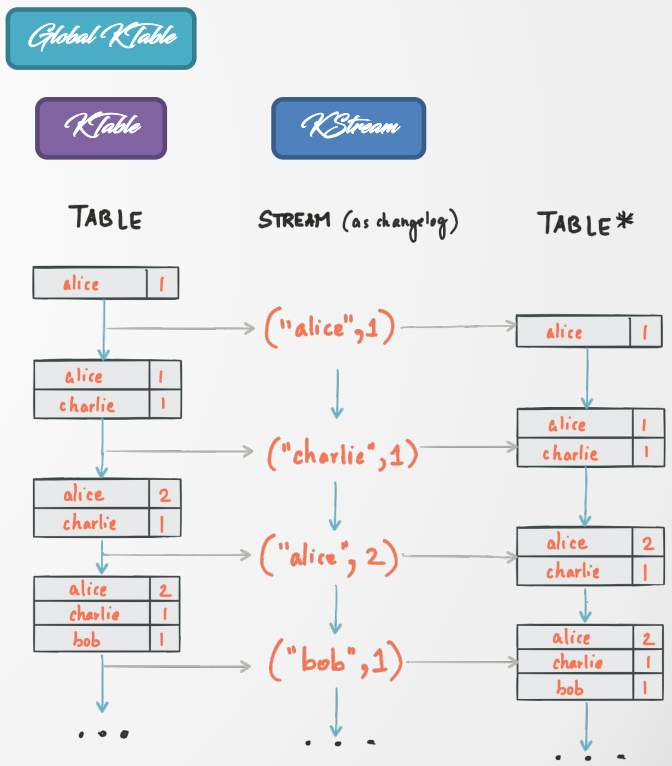
defines computational logic of the data processing. graph of stream processors (nodes) that are connected by streams (edge)
~ DAGs – use low level processor api or kafka streams dsl.



Stream Processing

Stateless = processing of messages is independent from the processing of other messages. transform one message at a time, filter out messages bases on some condition

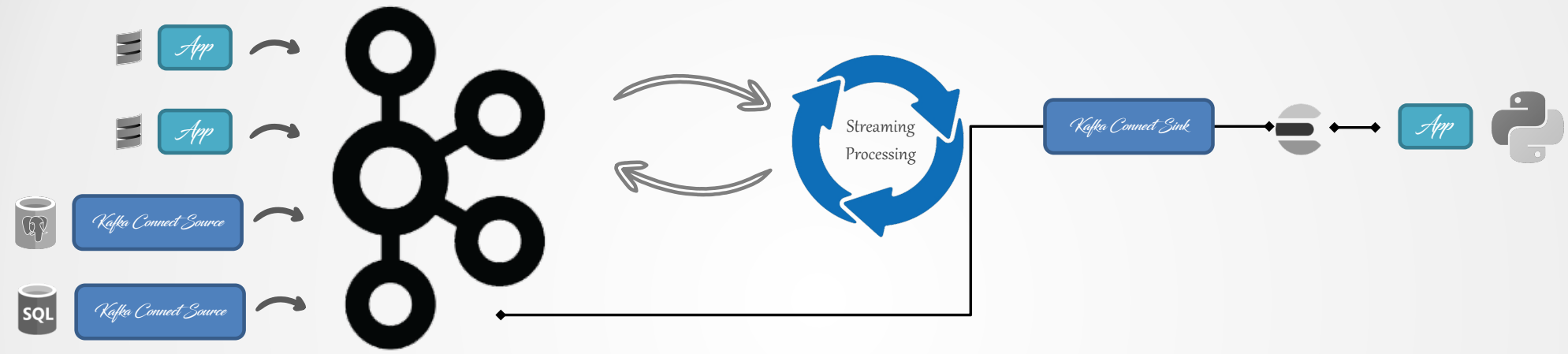
Stateful = needs to join, aggregate, windows (save state). provides a powerful, elastic, highly-scalable, and fault-tolerant stateful processing capability



Shallow men believe in luck or in circumstance. Strong men believe in cause and effect.

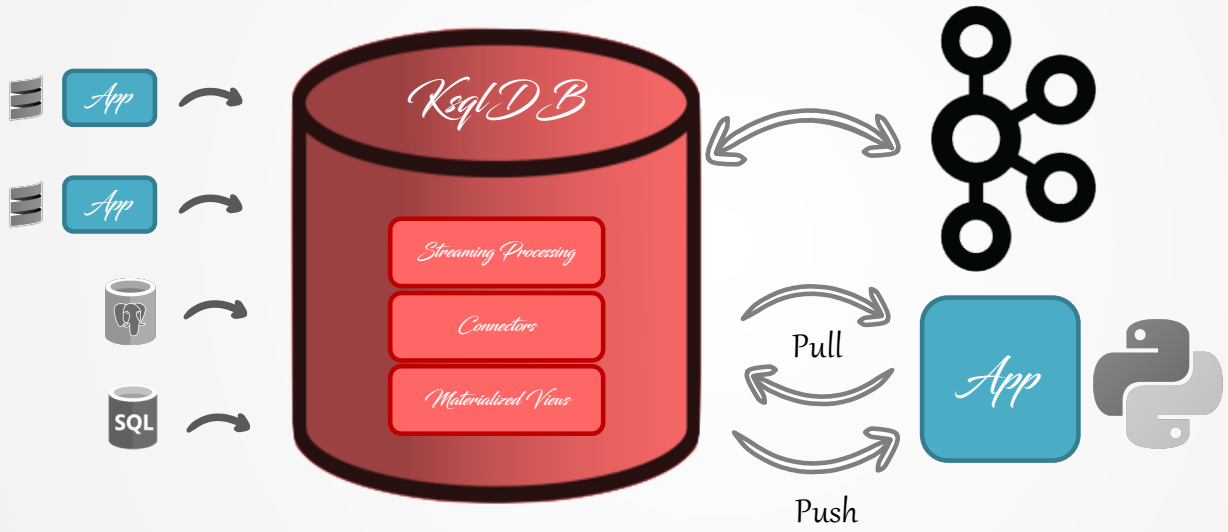
Ralph Waldo Emerson

Apache Kafka ~ Storage and Processing



KSQLDB [Single Mental Model]

With KSQLDB We Can Simplify Data Pipeline using as a Database On
Top of Apache Kafka as Middleware of Applications and Datastores



With a Lightweight, Familiar SQL Syntax, KSQLDB Presents a **Single Mental Model** for Working with Event Streams Across Your Entire Stack: Event Capture, Continuous Event Transformations, Aggregations, and Serving Materialized Views

KSQLDB [Basics]

Intro

Streaming SQL Engine for Apache Kafka

it provides an easy-to-use yet powerful interactive SQL interface for stream processing on Kafka, without the need to write code in a programming language such as Java or Python. KSQL is scalable, elastic, fault-tolerant, and real-time. it supports a wide range of streaming operations, including data filtering, transformations, aggregations, joins, windowing, and sessionization.

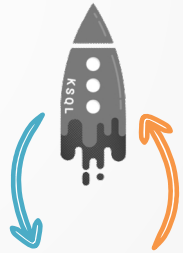
Use-Cases

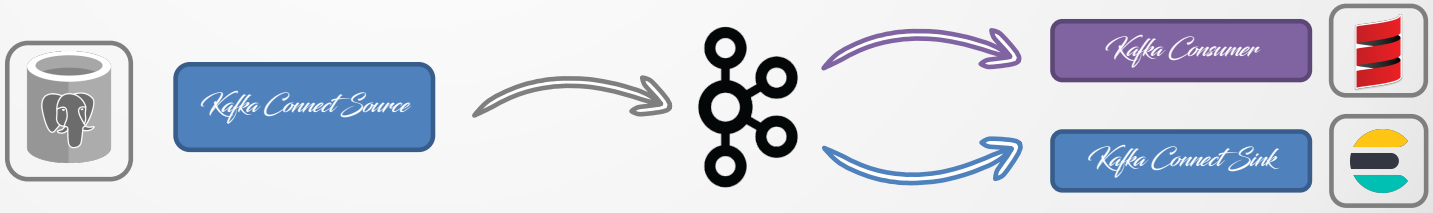
Use-Cases for KSQL

- Streaming ETL
- Real-Time Monitoring & Analytics
- Data Exploration & Discovery
- Anomaly Detection
- Personalization
- Sensor Data & IoT
- Customer 360 View

ETL in Real-Time

1) Ingest Data from PostgreSQL using Kafka Connect Source API
2) Apply Transformations using KSQL [Kafka Streams API]
3) Store Transformed Data into Kafka Topic
4) Send Data to Destinations using Kafka Connect Sink API





KSQLDB [Architecture & Guideline]

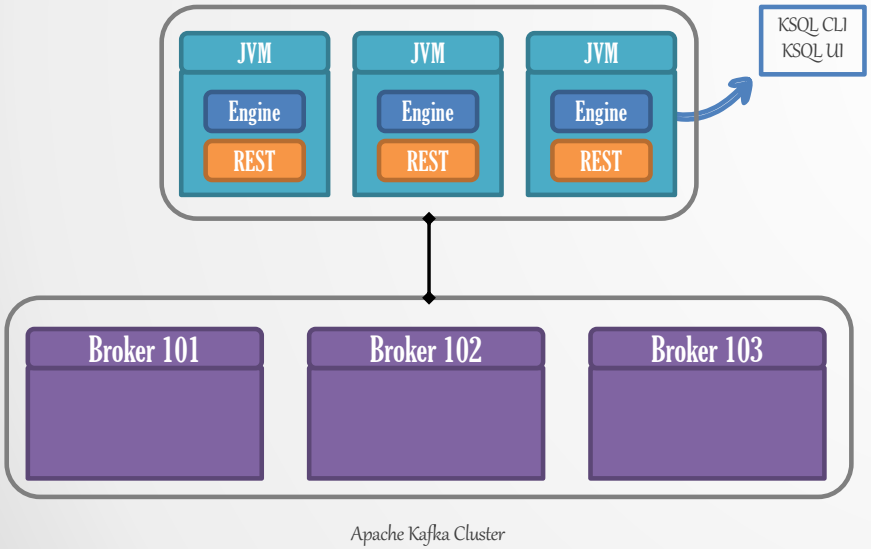
Components

KSQL Engine = Executes KSQL Statements & Queries [Kafka Streams API]
KSQL REST Interface = Server Interface for Communication to KSQL Engine

Deployment Modes

Interactive = Data Exploration & Pipeline Development
Headless = Long-Running Production Environments

Interactive KSQL Deployment Mode



Apache Kafka Cluster

General [Production-Size]

CPU = Consume to Serialize & Deserialize Streams and Tables ~ [4 Cores]
Disk = Persist Temporary State for Agg & Joins ~ [SSD of 100 GB]
Memory = States and Join Operations ~ [32 GB]
Network = Heavily in a Fast & Optimal Throughput ~ [1 Gbit NIC]

Language Elements

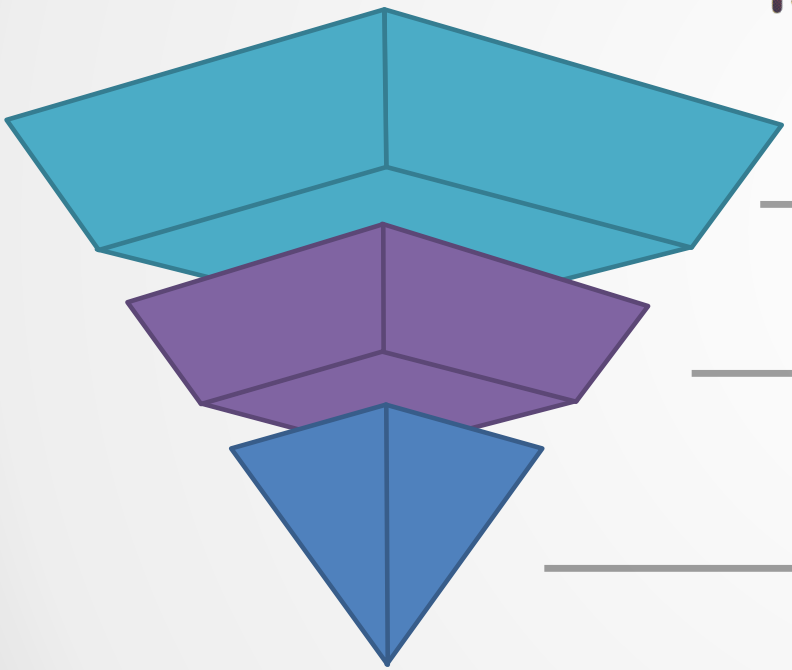
Data Definition Language [DDL]

CREATE STREAM
CREATE TABLE
DROP STREAM
DROP TABLE
CREATE STREAM AS SELECT (CSAS)
CREATE TABLE AS SELECT (CTAS)

Data Manipulation Language [DML]

SELECT
INSERT INTO
CREATE STREAM AS SELECT (CSAS)
CREATE TABLE AS SELECT (CTAS)

KSQLDB & Kafka Streams



Use KSQL for

1) New to Streaming & Kafka
2) Preferences to Write SQL [Java or Scala]
3) Enriching Data, Join Sources, Filtering, Transforming
4) Option to Add User Defined Functions [UDF]
5) KSQL Rest API for Python, GO, C#, JavaScript

Differences	KSQL	Kafka Streams
Write	KSQL Statements	JVM Applications
Data Formats	Avro, Json, CSV	Any Data Format
REST API	Yes	No
Queryable State	No	Yes

KSQL

Highest Level of Abstraction for Implementing Real-Time Streaming Business Logic

Kafka Streams

Programming API Interface for Java & Scala Applications [JVM]

Kafka Producer & Consumer

First APIs Released for Apache Kafka, Producer Writes Data into Kafka and Consumer Reads Data from Kafka

Easy

CREATE STREAM, CREATE TABLE, SELECT, JOIN, GROUP BY, SUM

KStream, KTable, Filter(), Map(), FlatMap(), Join(), Aggregate()

subscribe(), poll(), send(), flush(), beginTransaction()

Flexibility

Use-Case = Near Real-Time ETL using Kafka, KsqlDB in Spotify

Source

Music data and other datasets

Ingestion

Apache Kafka

Enricher

KsqlDB

Joined dataset

Serving

Apache kafka

KSQLDB – Data Processing [Streams & Tables]

1. Exploring KSQLDB Capabilities
2. Processing Events using SQL
3. Applying Data Enrichment [ETL]
4. Store Results into Output Topics


Freedom lies in being bold.
Robert Frost
quotefancy

Apache Spark [Fundamentals]

Apache Spark

- Open-Source Distributed Cluster-Computing Framework
- **Implicit Data Parallelism** & Fault Tolerance
- Optimized for Memory Computation
- Written In - Scala
- 100x ~ MapReduce Jobs & 10x – Disk-Based Operations

History

- University of California, Berkeley's AMPLab
- Open-Sourced in 2014 – Top-Level Apache Project
- Databricks – New World Record in Large Scale Sorting [**2014**]
- Ali Ghodsi | Reynold Xin | Matei Zaharia ~ **Databricks**
- 1,000 Contributors in 2015

Key Capabilities

- Unified Stack for Interactive, Streaming & Predictive Analysis
- Batch & Streaming in an Unified Platform
- Designed for Large-Scale Data Processing

Performance

Daytona Gray

- 100 TB in **23 Minutes** with 206 EC2 VMs
- 100 TB in 72 Minutes with 2,100 Nodes

Core APIs

- SQL
- Java
- Scala
- Python
- R

Processing Structures

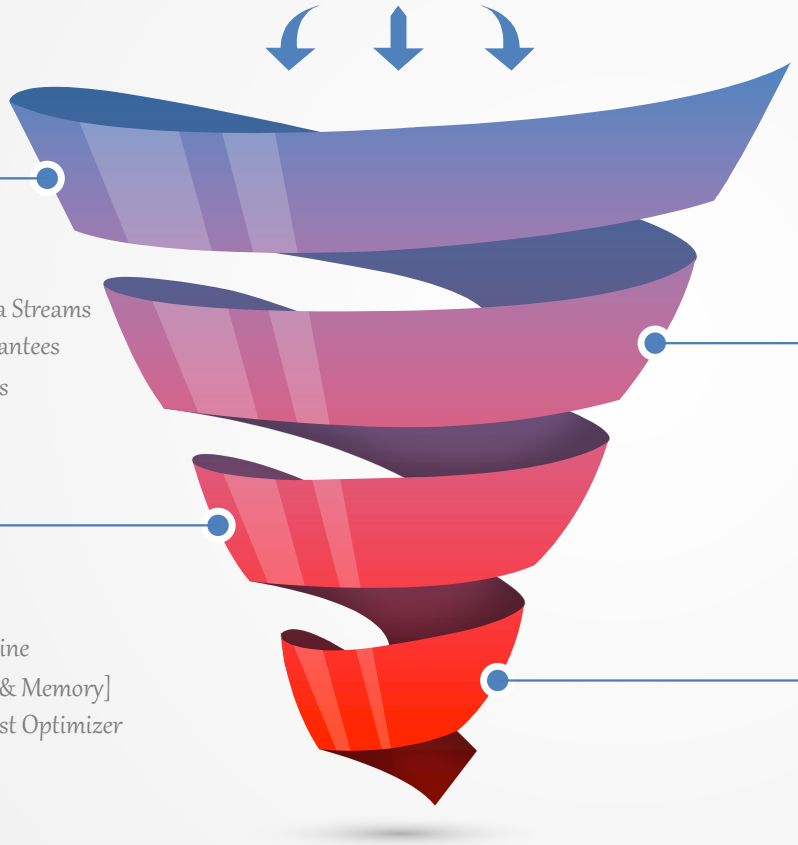
- RDD – Resilient Distributed DataSet
- Spark Streaming – Processing Data Streams using DStreams

- Spark-SQL, DataSets & DataFrames – Processing Structured Data
- Structured Streaming – Processing Structured Data Streams

Apache Spark [Structured Streaming]

Philosophy [Moto]

Treat Data Streams = **Unbounded Tables**
Incremental Query over Streams



Guarantees

Scalable & Fault-Tolerant
Incremental & Continuous Data Streams
End-to-End Exactly-Once Guarantees
Checkpoint & Write Ahead Logs

Seamless

Express Streaming Computation = Batch Computation

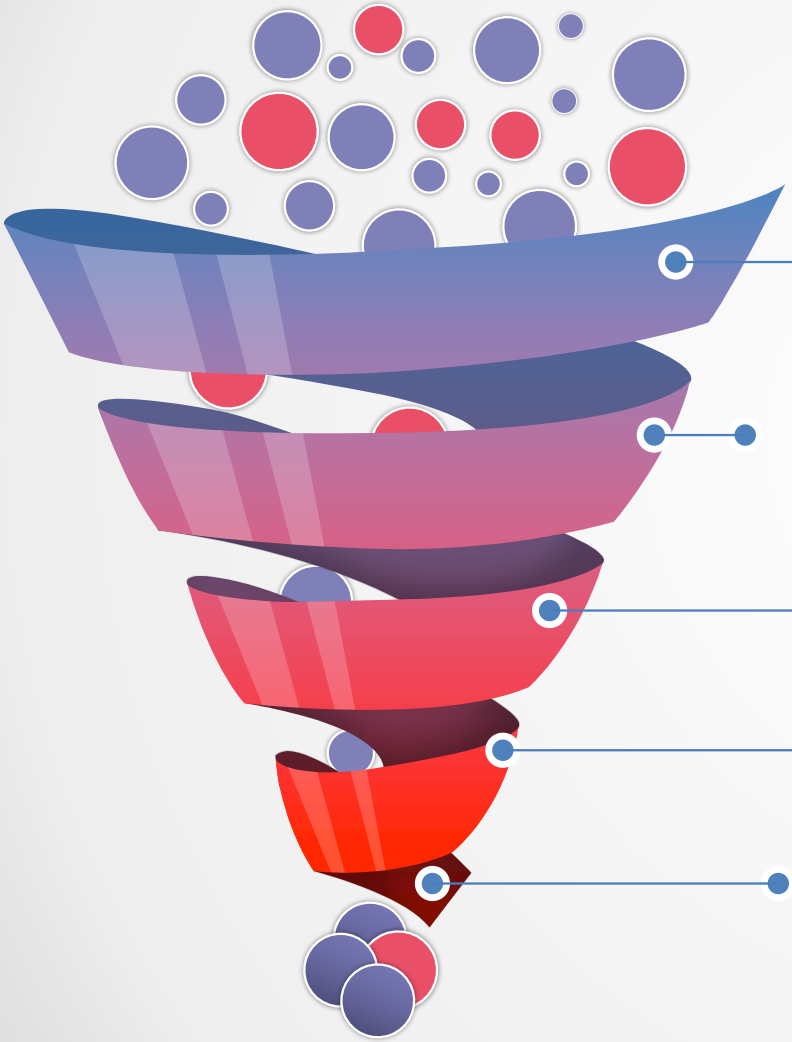
Engine

Built on Spark SQL Engine
Project Tungsten [CPU & Memory]
Optimized using Catalyst Optimizer

Processing

Uses Micro-Batch Processing Engine
100 milliseconds with Exactly-Once Latency

Apache Spark [Anatomy of a Streaming Query]



Source

```
spark.readStream
.format("kafka")
.option("subscribe", "input")
.load()
```

Transformation

```
.groupBy('value.cast("string") as 'key)
.agg(count("*") as 'value)
```

Sink

```
.writeStream
.format("kafka")
.option("topic", "output")
```

Trigger

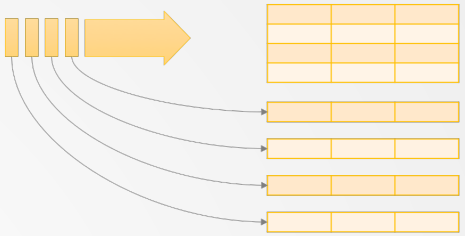
```
.trigger("1 minute")
.outputMode("update")
```

Checkpoint

```
.option("checkpointLocation", "…")
.start()
```

Data Streams

Unbounded Table



data stream as an unbounded table

new data in the data stream = new rows appended to an unbounded table

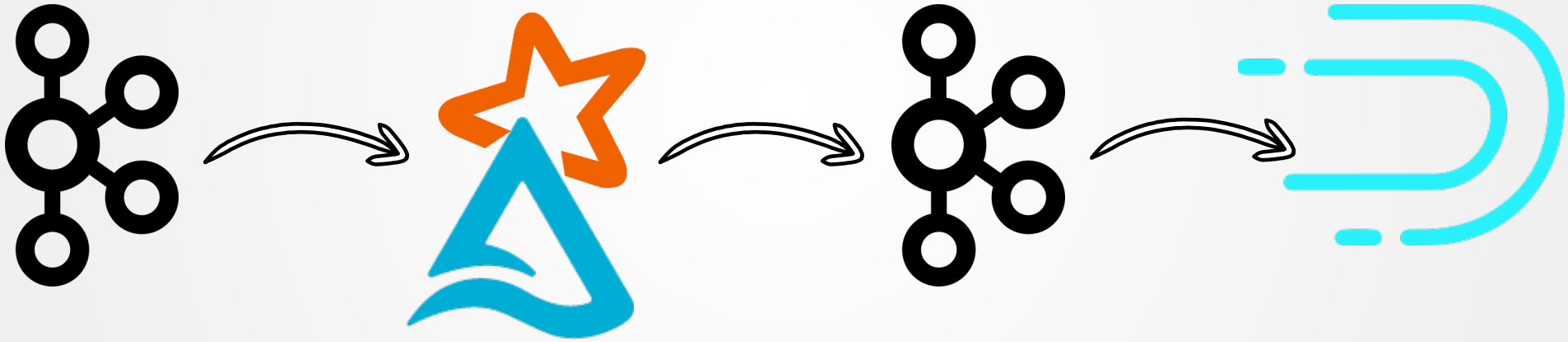
Use-Case = Near Real-Time ETL using Kafka, Spark and Delta Lake

Source
Apache Kafka

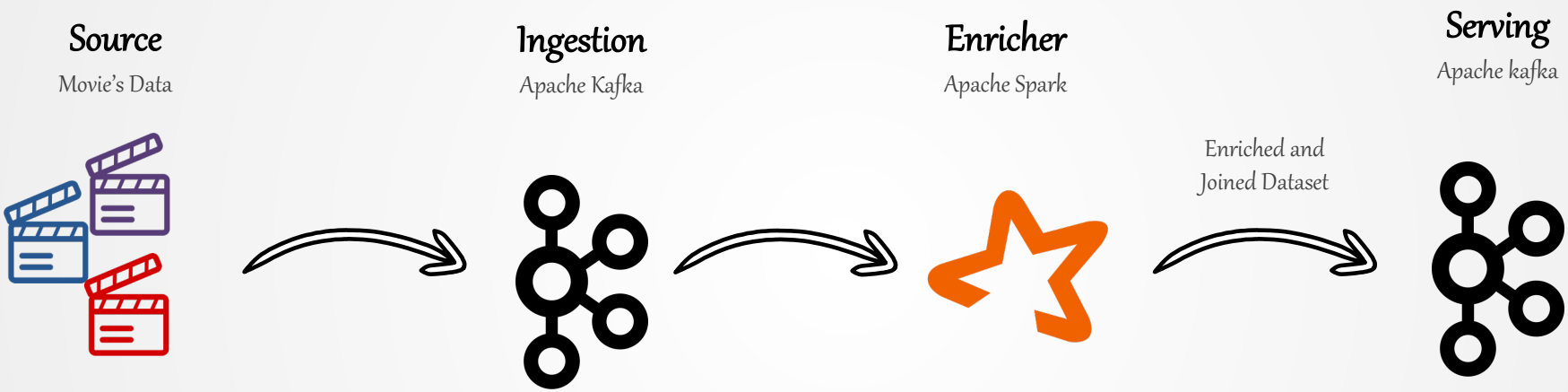
Transformations
Apache Spark [Structured Streaming]

Sink
Apache Kafka

Serving
Apache Druid



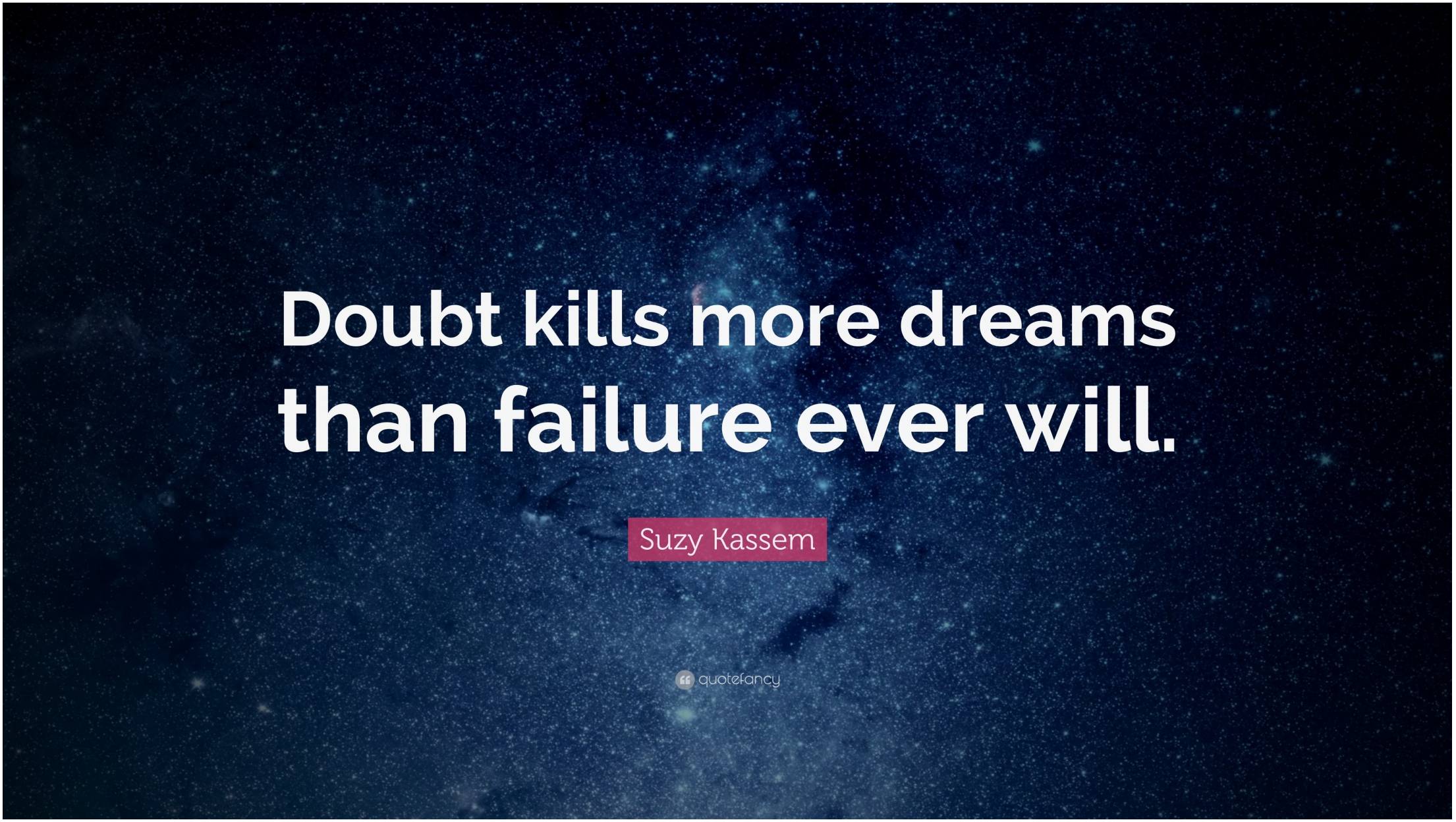
Use-Case = Near Real-Time ETL using Kafka, Spark in Netflix



Apache Kafka & Apache Spark [Structured Streaming]

1. Structured Streaming ~ Batch & Stream
2. Read from Topic [Apache Kafka]
3. Perform Computation [T] - Transformation
4. Write into Topic [Apache Kafka]

http://bit.ly/owshq-spark


Doubt kills more dreams
than failure ever will.
Suzy Kassem
quotefancy

Faust [Basics]

Python Streaming Processing

Faust Library

- Stream Processing Library ~ [Kafka Streams]
- High-Performance Distributed Systems in Real-Time
- Data Pipelines Processing Billions of Events per Day

Stream Processing & Event Processing [Similarity]

- Kafka Streams
- Apache Spark
- Apache Storm
- Apache Samza
- Apache Flink

Use-Cases

Use-Cases for Faust

- Event Processing [EP]
- Distributed Joins & Aggregations
- Machine Learning [ML]
- Asynchronous Tasks
- Distributed Computing
- Data Denormalization
- Intrusion Detection

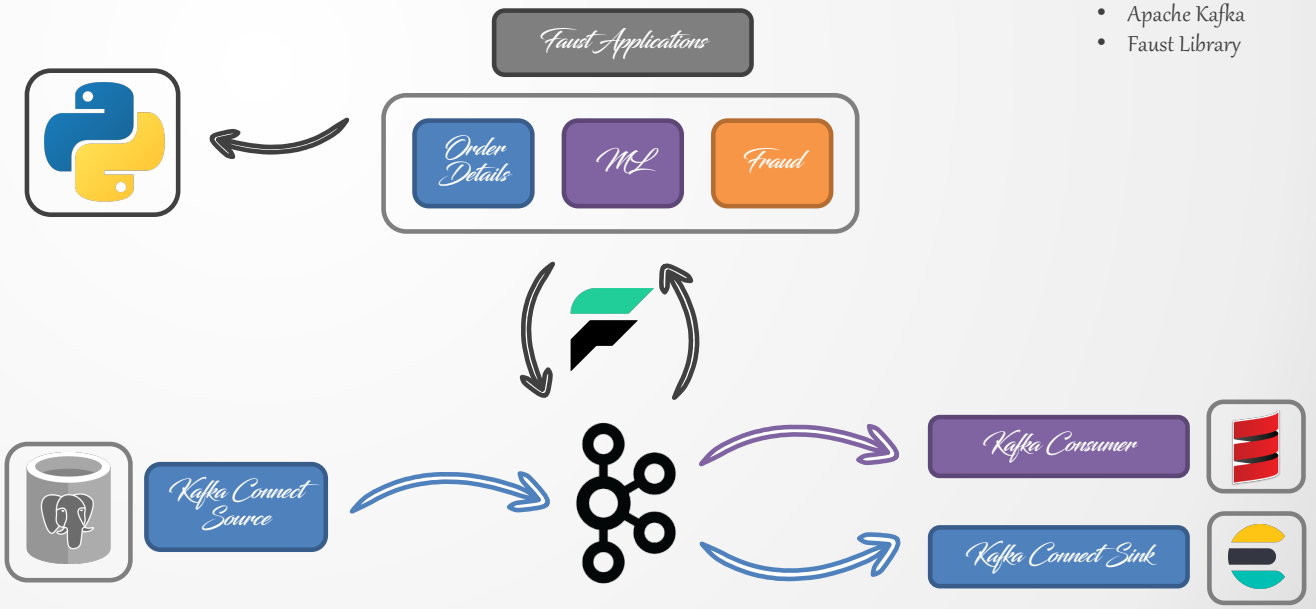
Requisites

Use Faust

- Python 3.6
- Apache Kafka
- Faust Library

Robinhood

1) High Performance Distributed & Real-Time Data Pipelines
2) Billions of Events Every Day
3) GitHub ~ 4,088 [Commits] & 72 [Contributors]



Faust [Concepts]

Agents

process infinite streams in a straightforward manner using asynchronous generators. the concept of "agents" comes from the actor model, and means the stream processor can execute concurrently on many CPU cores, and on hundreds of machines at the same time.

Tables

tables are sharded dictionaries that enable stream processors to be stateful with persistent and durable data. streams are partitioned to keep relevant data close, and can be easily repartitioned to achieve the topology you need.

Table

Mutable without
Historic Information

Fast

A single-core Faust worker instance can already process tens of thousands of events every second, and we are reasonably confident that throughput will increase once we can support a more optimized kafka client.

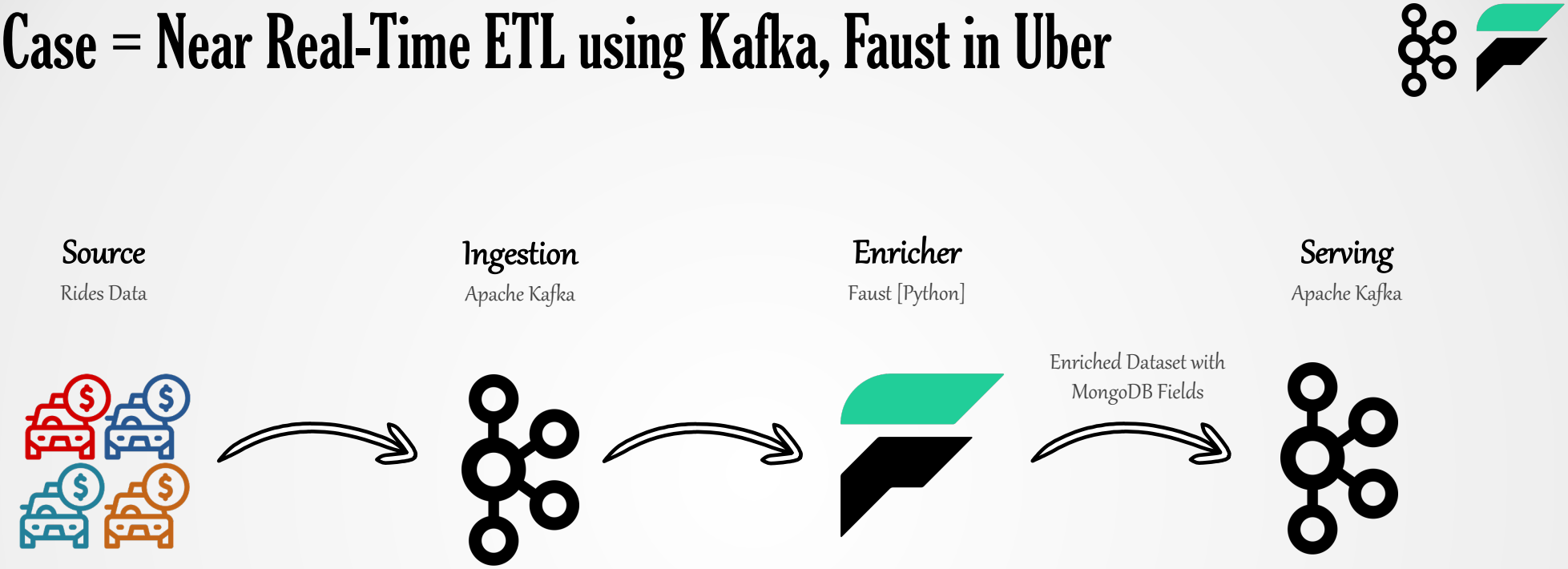
Highly-Available

Faust is highly available and can survive network problems and server crashes. in the case of node failure, it can automatically recover, and tables have standby nodes that will take over.

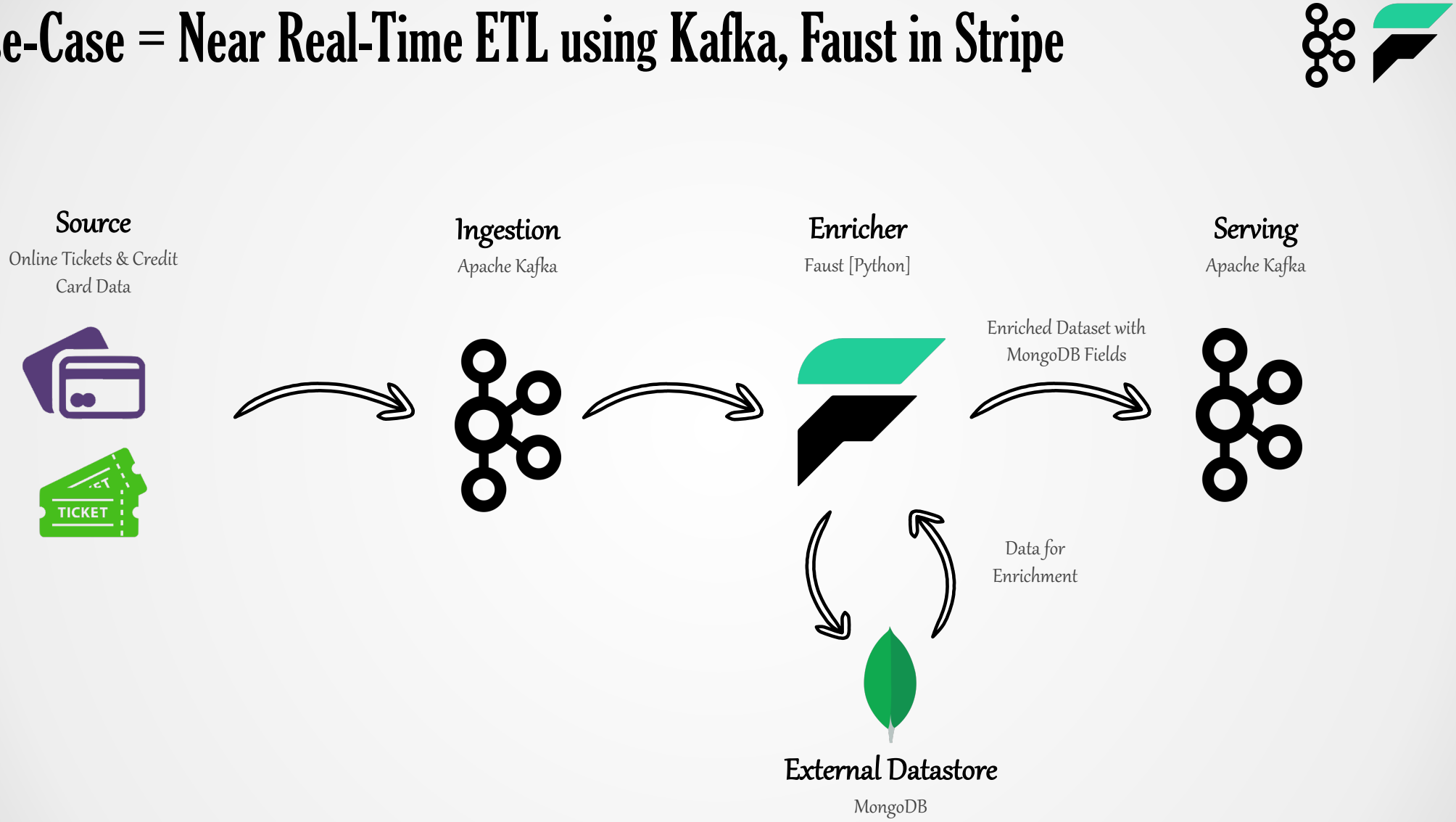
Flexible

Faust is just Python, and a stream is an infinite asynchronous iterator. if you know how to use Python, you already know how to use Faust, and it works with your favorite Python libraries like Django, Flask, SQLAlchemy, NTLK, NumPy, SciPy, TensorFlow, etc.

Use-Case = Near Real-Time ETL using Kafka, Faust in Uber



Use-Case = Near Real-Time ETL using Kafka, Faust in Stripe



Apache Kafka & Faust [Python]

1. Configure Faust App
2. Read from Topic [Apache Kafka]
3. Perform Computation [T] - Transformation
4. Write into Topic [Apache Kafka]


Those who dare to fail
miserably can achieve greatly.
John F. Kennedy
quotefancy

Stream Processing Engines [Comparison]

DataStream API

- Robust Stateful Streaming API
- Single Runtime for Batch & Streaming]
- High Throughput
- Automatic Memory Management <> [GC]
- Snapshots ~ Chandy-Lamport Algorithm
- Exactly-Once Semantics Guarantee

Use-Case

Structured Streaming

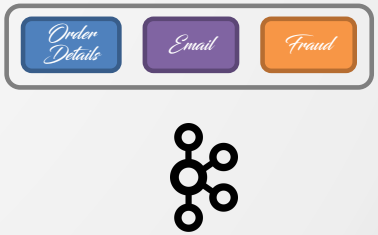
- 2nd Generation
- Micro-Batching Processing ~ 100 ms
- Structured API ~ [DataFrames]
- Support for Java, Scala, Python, R & SQL
- Exactly-Once Semantics Guarantee

Use-Case

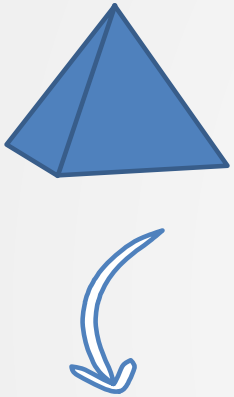
Kafka Streams & KSQL

- Simple & Lightweight Library [Java & Scala]
- Event-at-a-Time Processing ~ 10 ms
- Without External Dependencies
- KSQL – Kafka SQL
- Exactly-Once Semantics Guarantee

Use-Case



Stream Processing [Design Pattern] = Single Event Processing

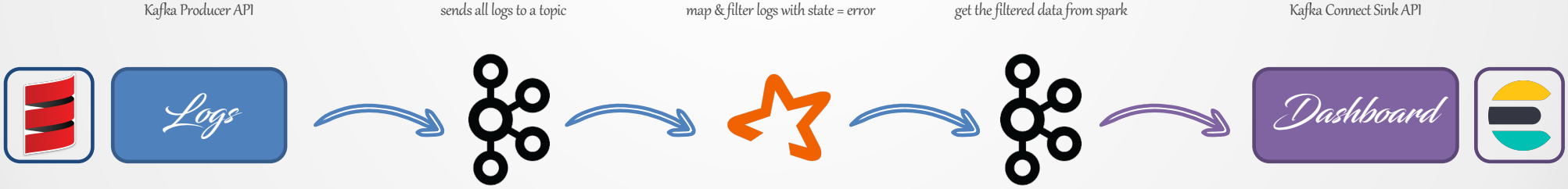


Single Event Processing

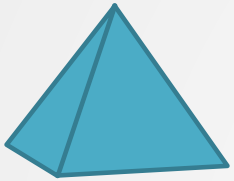
the most basic pattern of stream processing is the processing of each event in isolation. this is also known as a map & filter pattern because it is commonly used to filter unnecessary events from the stream or transform each event. [the term "map" is based on the map & reduce pattern in which the map stage transforms events and the reduce stage aggregates them.]

Info

Reads Log Messages
Writes into Apache Kafka
Process using Apache Spark – Structured Streaming



Stream Processing [Design Pattern] = Processing with Local State

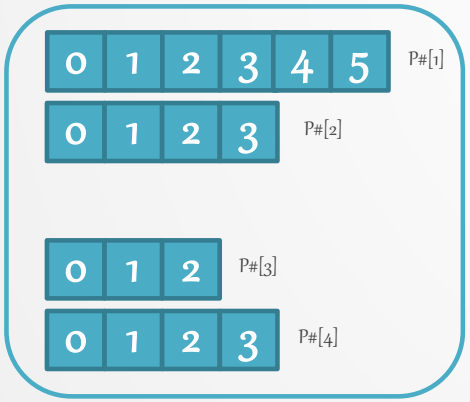


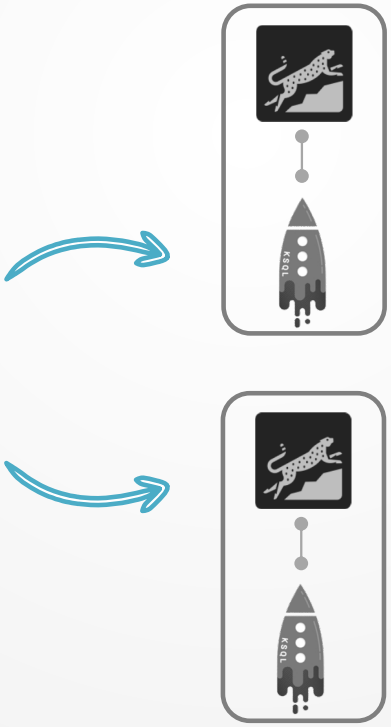
Processing with Local State

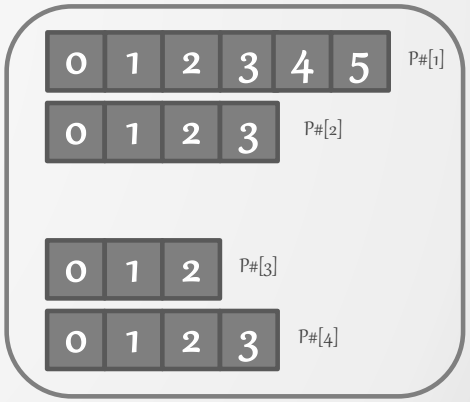
most stream-processing applications are concerned with aggregating information, especially time-window aggregation. an example of this is finding the minimum and maximum stock prices for each day of trading and calculating a moving average.

Info

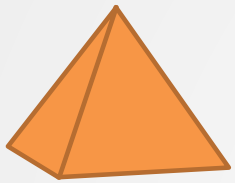
Memory Usage = Local State must Fit into Memory
Persistence = Local Last State
Rebalancing = Partitions get Reassigned to a Different Consumer







Stream Processing [Design Pattern] = Multiphase Processing

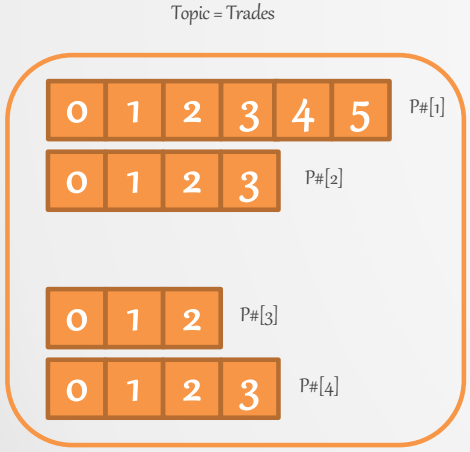


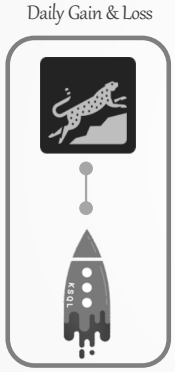
Multiphase Processing or Repartitioning

type of multiphase processing is very familiar to those who write map-reduce code, where you often have to resort to multiple reduce phases. if you've ever written map-reduce code, you'll remember that you needed a separate app for each reduce step. unlike mapreduce, most stream-processing frameworks allow including all steps in a single app, with the framework handling the details of which application instance (or worker) will run reach step.

Info

Two-Phase Approach
[1] = Compute Gain & Loss for Each Stock Symbol
[2] = Find Top 10 Stocks of Day [Summary]

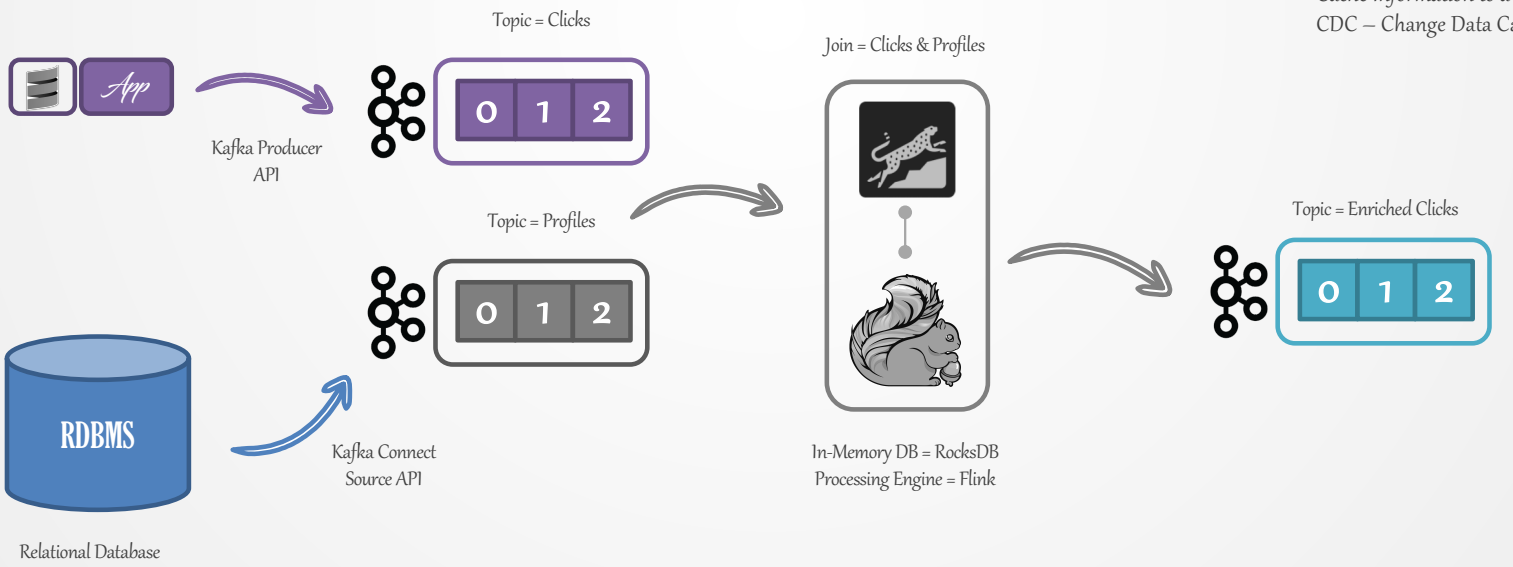




Stream Processing [Design Pattern] = Stream-Table Join

Processing with External Lookup ~ Stream-Table Join

stream processing requires integration with data external to the stream - validating transactions against a set of rules stored in a database, or enriching click- stream information with data about the users who clicked. In order to get good performance and scale, we need to cache the information from the database in our stream-processing application. Managing this cache can be challenging though—how do we prevent the information in the cache from getting stale? If we refresh events too often, we are still hammering the database and the cache isn't helping much. If we wait too long to get new events, we are doing stream processing with stale information.



Info

Data Enrichment using Lookup Strategy [High-Latency]

External Data Stores = 5-15 milliseconds per Record [10K]
Stream Processing Systems = 100K ~ 500K per Second

Cache Information to a Stream-Processing Application
CDC – Change Data Capture Strategy

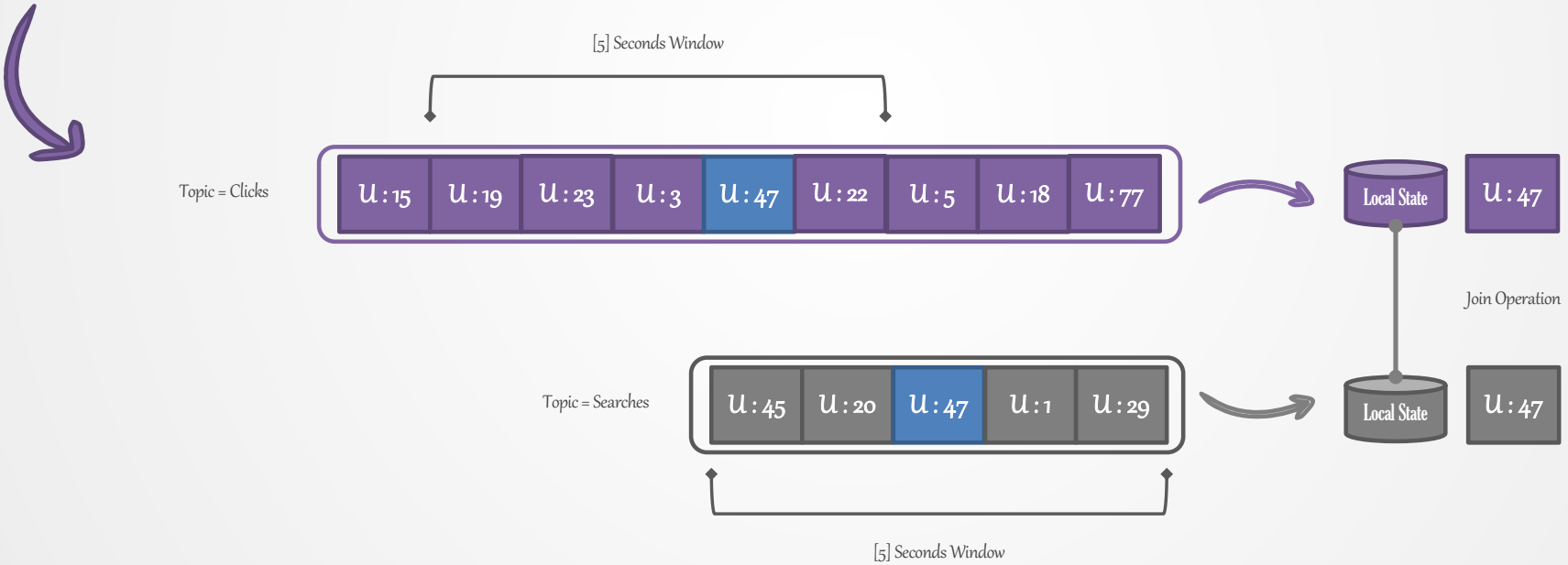
Stream Processing [Design Pattern] = Streaming Join

Streaming Join

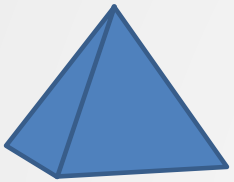
join two real event streams rather than a stream with a table. remember that streams are unbounded [infinite and ever growing]. When you perform streams-table join you ignore most of the history in the stream because you only care about the current state in the table. but joining two-streams you're joining the entire history, trying to match events in one stream with events based in a key, this is called = windowed-join.

Info

Stream-to-Stream Join
Perform Joins using In-Memory DB [RocksDB]
Only Time-Window Joins



Stream Processing [Design Pattern] = Out-of-Sequence Events



Out-of-Sequence Events

handling events that arrive at the stream at the wrong time is a challenge not just in stream processing but also in traditional ETL systems.

recognize that event is out of sequence, processor needs to examine the event times

define a time period during which it will attempt to reconcile out-of-sequence events. e.g. [3 hours reconcile and 3 weeks discard]

have capacity to reconcile event. process engine must handle both old and new events at any give moment

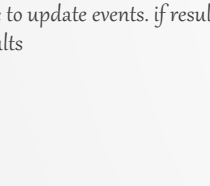
able to update events. if results written into a database, a put or update is enough to update results

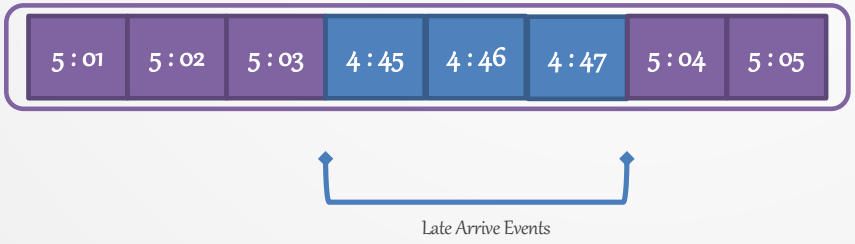
Info

Feature = Watermarking

- Google Cloud DataFlow [Apache Beam]
- Kafka Streams & KSQL
- Apache Spark [Structured Streaming]
- Apache Flink

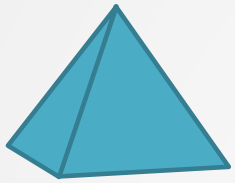
typically done by maintaining multiple aggregation windows available for update in the local state and giving ability to configure this windows aggregations.





find the out of sequence events by adding values into a compacted topic, which will have the latest value for each key

Stream Processing [Design Pattern] = Reprocessing



Reprocessing Data

pattern to process events.

1 – improved or new version of an app want to run the newest code on the same stream as the old, produce a new stream of results that does not replace the first version, compare results and move clients to the new app

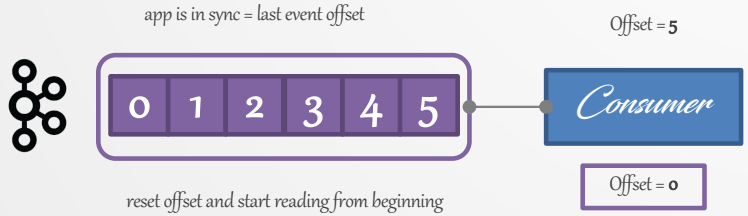
2 – existing stream-processing app is buggy. we fix the bug and we want to reprocess the event streams and recalculate our results

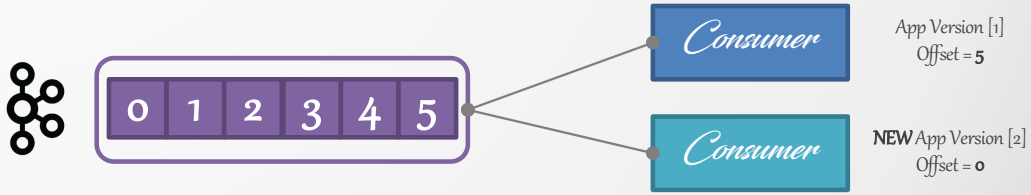
Pattern = Reprocess Event Stream

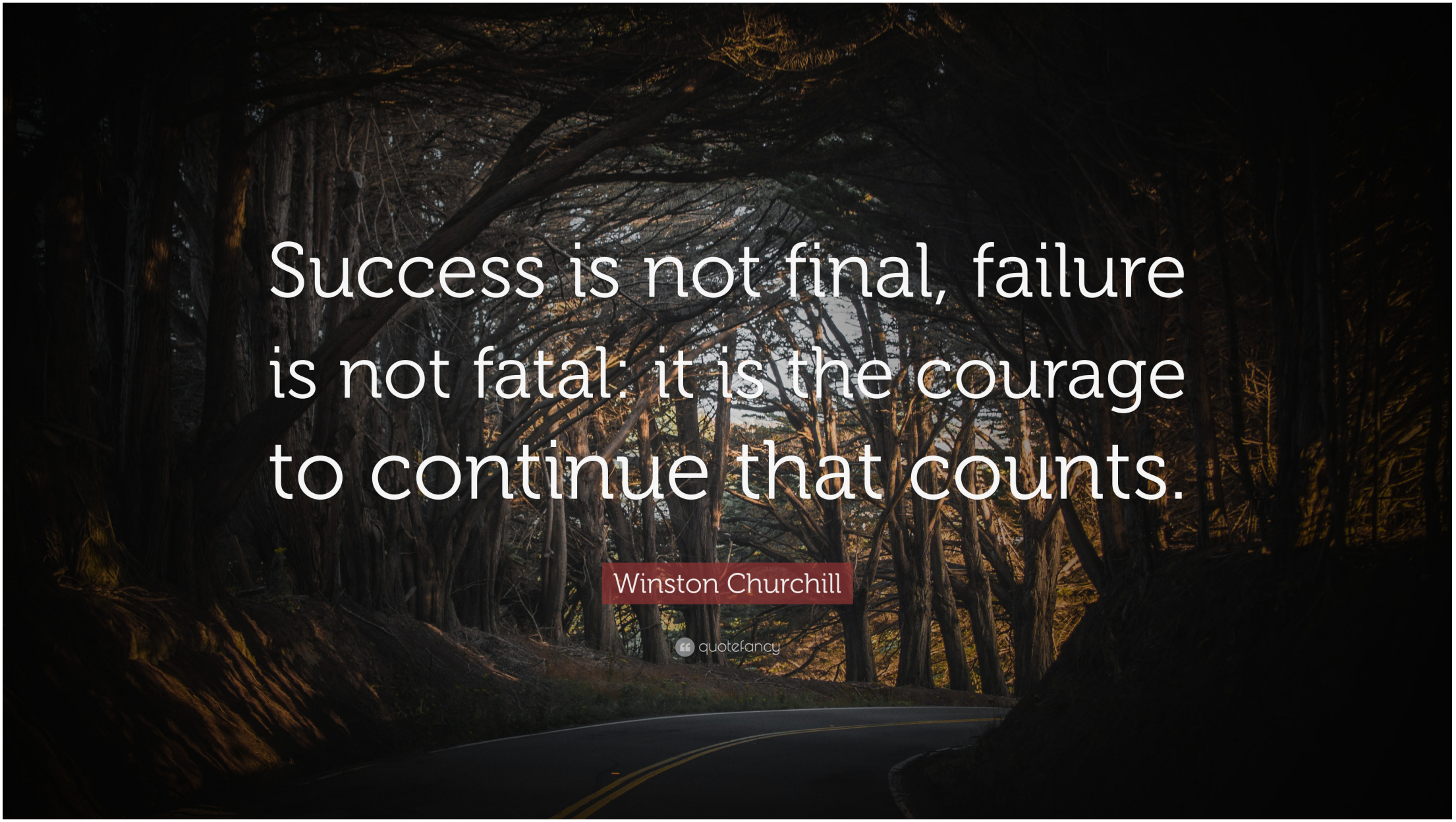
- reset offset of an existing app to start processing back at the beginning of the input streams
- first method is much safer, it allows switching back and forth between multiple version

Pattern = New App Version

- spinning up the new version of the app as a new consumer group
- configuring the new version to start processing from the first offset of the input topic
- letting the new app continue processing and switching the client app to the new result stream






Success is not final, failure is not fatal: it is the courage to continue that counts.
Winston Churchill
quotefancy


ONEWAY
SOLUTION