Inspire...Educate...Transform.

Inspire…Educate…Transform.

Applying ML to Big Data using Hadoop and Spark Ecosystem

Spark and towards Spark ML

Dr. Prasad M Deshpande

Slides adapted from Dr. Manoj Duse

Agenda

- Spark
- RDDs, Partitions
- Actions, Transformers
- Managing partitions
- Spark ML Overview

Another Parallelization platform: SPARK

- Learnings from Hadoop MR led to Spark
- SPARK on YARN
- SPARK and MR can coexist
- A platform for real-time, batch, ML

What is Spark?

Fast and Expressive Cluster Computing System
Compatible with Apache Hadoop

Up to **10×** faster on disk,
100× in memory

Efficient

- General execution graphs
- In-memory storage

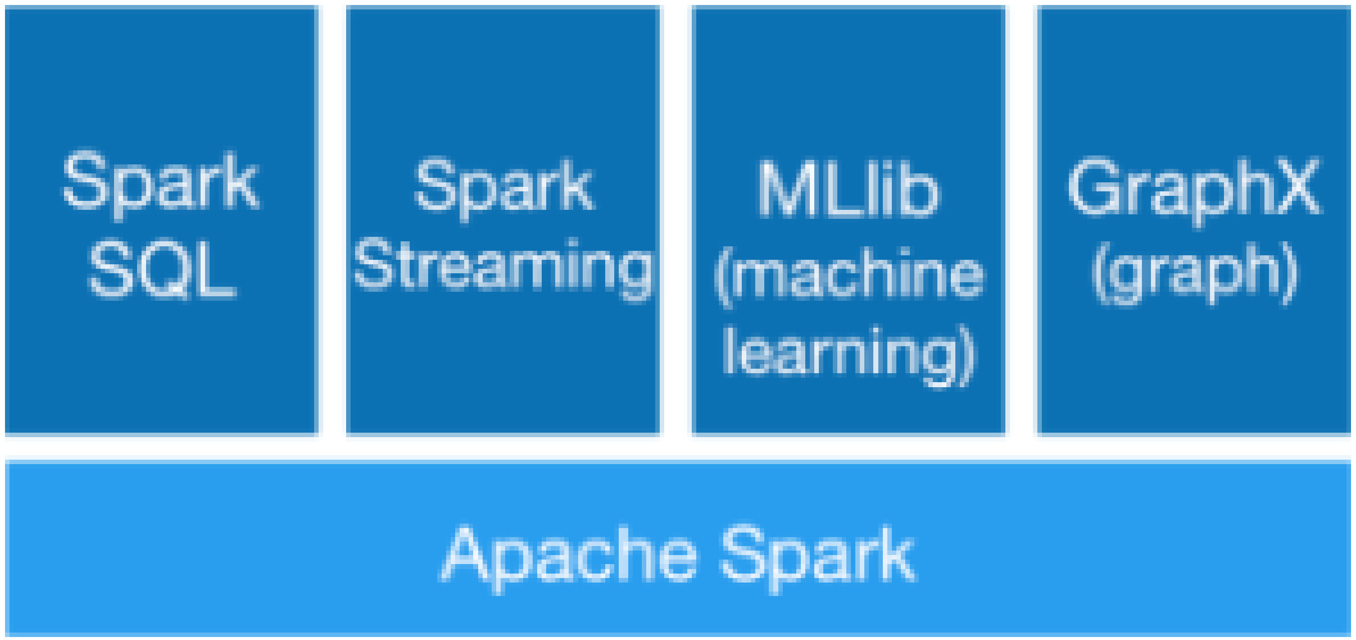
2-5× less code

Usable

- Rich APIs in Java, Scala, Python
- Interactive shell

From McDonough Spark tutorial from Spark Summit 2013

SPARK ecosystem



Spark Core:

Responsible for:

- ✓ Memory Management and fault recovery

- ✓ Supports/implements key concepts of RDDs and Actions

- ✓ Scheduling, Monitoring, Distributing jobs on cluster [via YARN]

Lightning-Fast Cluster Computing

Key Concepts

Write programs in terms of transformations on distributed datasets

Resilient Distributed Datasets

- Collections of objects spread across a cluster, stored in RAM or on Disk
- Built through parallel transformations
- Automatically rebuilt on failure

Operations

- Transformations (e.g. map, filter, groupBy)
- Actions (e.g. count, collect, save)

Spark Terminology

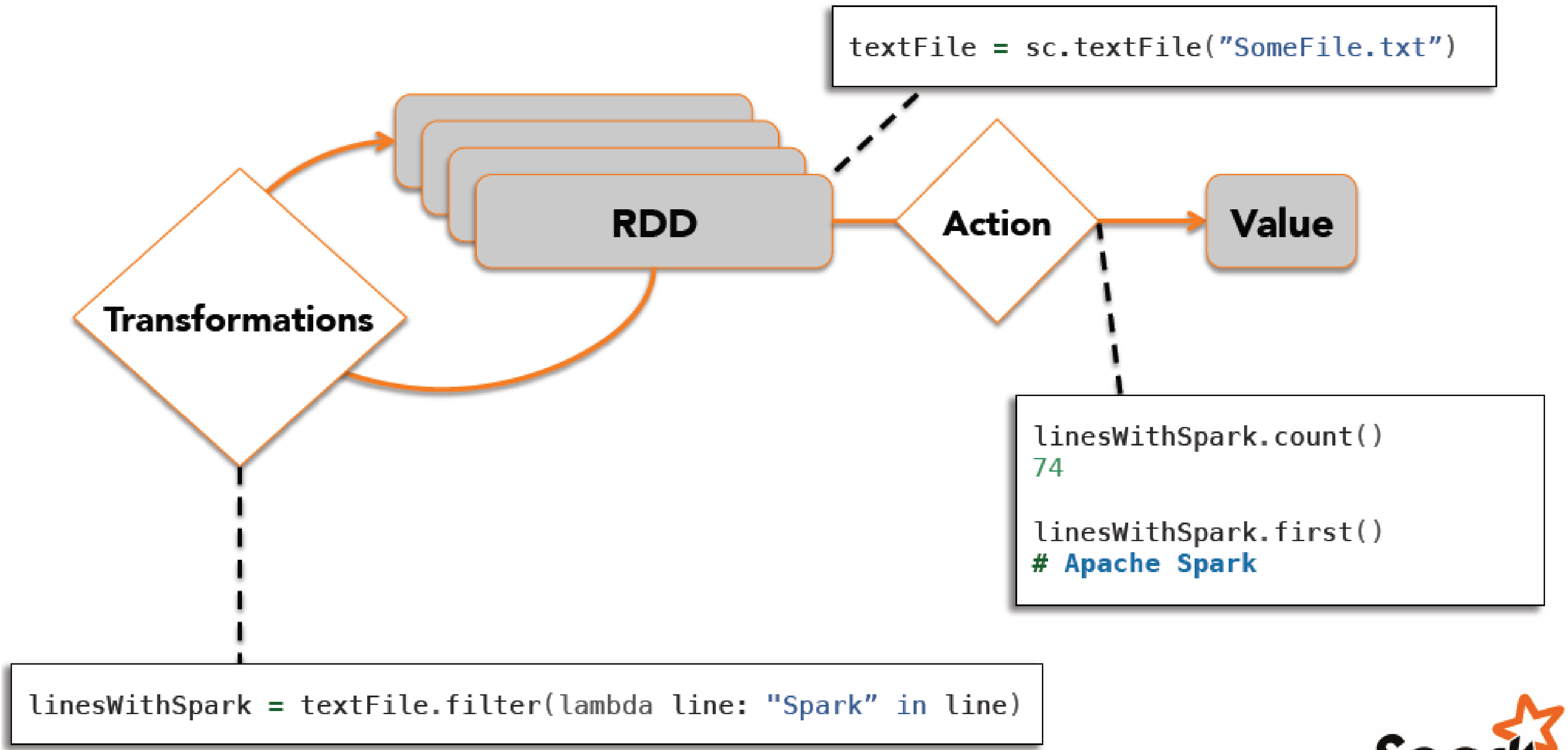
Driver program	The process running the main() function of the application and creating the SparkContext
Executor	A process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage across them. Each application has its own executors.

SparkContext works with RM

Programming languages supported by Spark include:

- Java
- Python [PySpark]
- Scala
- SQL
- R [SparkR]

Working With RDDs



Spark

RDDs are immutable

Transformation on one RDD results into a new RDD

Spark Example: Log Mining

- Load error messages from a log into memory, then interactively search for various patterns

Base RDD

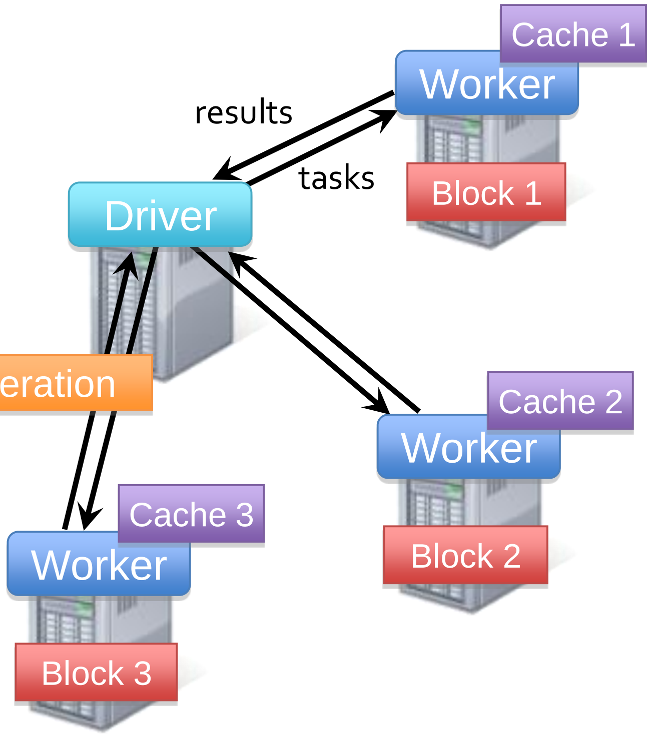
Transformed RDD

Cached RDD

Parallel operation

```
lines = sc.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains("database")).count
cachedMsgs.filter(_.contains("memory")).count
. . .
```



You should cache RDDs you work with when you want to execute two or more actions on it for a better performance

Reference to be added

Lazy Initialization

- Populating of blocks into memory deferred until action is invoked.
- RDD created but with no data

- Simply put, an action triggers actual evaluation of the RDD .
- Only actions can materialize the entire processing pipeline with real data.

Examples of Transformations

- map
- filter
- flatMap
- groupByKey
- sortByKey

Examples of Actions

- Count
- Top(k)

Broadcast variable and Accumulator

- Broadcast variable is a read-only variable
- made available from the driver program that runs the SparkContext object to the nodes that will execute the computation.
- useful in applications that need to make the same [typically reference data] available to the worker nodes in an efficient manner, such as machine learning algorithms.
- one time thing : distributed to the workers only once

- An accumulator is also a variable that is broadcasted to the worker nodes.
- The key difference between a broadcast variable and an accumulator is that while the broadcast variable is read-only, the accumulator can be added to.

Printing contents of a RDD

- myRDD.collect().foreach(println)

- Very useful for debugging

Parallelizing Data

How many partitions my RDD is split into?

myRDD.partitions.size

How to enforce "degree of parallelism"

myRDD = sc.parallelize(1 to 500, 5)

myRDD.partitions.size
res27: Int = 5

repartition vs coalesce

repartition : will shuffle the original partitions and repartition them

coalesce : will just combine original partitions to the new number of partitions.

shuffling could be very costly,
if all you want is to reduce the number of partitions:
use coalesce

RDD ➔ DataFrames➔ DataSets

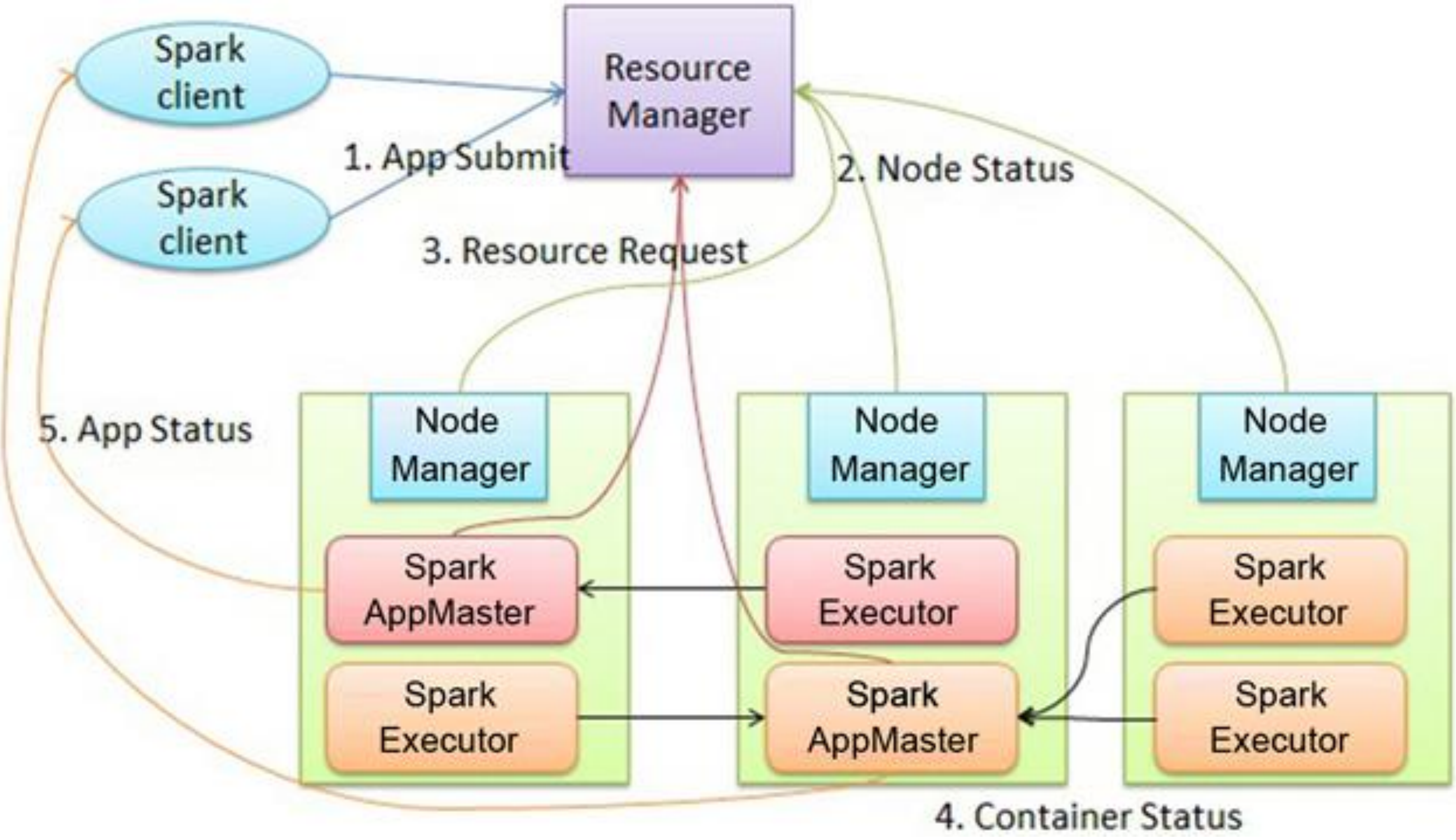
- **Spark Dataframe APIs –**
- Unlike an RDD, data organized into named columns.
- For example a table in a relational database.
- Allows developers to impose a structure onto a RDD

- **Spark Dataset APIs –**
- Datasets in Apache Spark are an extension of DataFrame API which provides type-safe [compile time], object-oriented programming interface.

- One can seamlessly move between DataFrame or Dataset and RDDs by simple API method calls like .rdd or .toDF
- DataFrames and Datasets are built on top of RDDs.

- **RDD –** The RDD APIs have been on Spark since the 1.0 release.
- **DataFrames –** Spark introduced DataFrames in Spark 1.3 release.
- **DataSet –** Spark introduced Dataset in Spark 1.6 release.

Relating back to YARN



SPARK vs MR

- Ease of use
- Developer productivity
- Speed
- Ecosystem: Spark R, Spark MLLib, PySpark, SparkSQL
- Lot of apache projects also moving to support/leverage Spark

Spark ML

At a high level, it provides tools such as:

- ML Algorithms: common learning algorithms such as classification, regression, clustering, and collaborative filtering

- Featurization: feature extraction, transformation, dimensionality reduction, and selection

- Pipelines: tools for constructing, evaluating, and tuning ML Pipelines

- Persistence: saving and load algorithms, models, and Pipelines

- Utilities: linear algebra, statistics, data handling, etc.

ML Lib vs ML

ML spark.ml

- New
- DataFrames
- Pipelines

ML Lib spark.mllib

- Old
- RDDS
- But more features but ML catching up
- In maintenance mode; no new functionality will be added

What is the future direction…

- After reaching feature parity (roughly estimated for Spark 2.3), the RDD-based API will be deprecated.

- The RDD-based API is expected to be removed in Spark 3.0.

- **Transformer**: A Transformer is an algorithm which can transform one DataFrame into another DataFrame.
- E.g., an ML model is a Transformer which transforms a DataFrame with features into a DataFrame with predictions.

- **Estimator**: An Estimator is an algorithm which can be fit on a DataFrame to produce a Transformer.
- E.g., a learning algorithm is an Estimator which trains on a DataFrame and produces a model.

- **Pipeline**: A Pipeline chains multiple Transformers and Estimators together to specify an ML workflow.

Pipeline

A Estimator implements a method fit(), which accepts a DataFrame and produces a Model, which is a Transformer.

For example, a learning algorithm such as LogisticRegression is an Estimator, and calling fit() trains a LogisticRegressionModel, which is a Model and hence a Transformer.

Pipeline, which consists of a sequence of **PipelineStages**
➔ Estimators and Transformers to be run in a specific order

To get a rough idea :

```
val lr = new LogisticRegression() .setMaxIter(10) .setRegParam(0.001)

val pipeline = new Pipeline() .setStages(Array(tokenizer, hashingTF, lr))

// Fit the pipeline to training documents.
val model = pipeline.fit(training)

 // Now we can optionally save the fitted pipeline to disk
model.write.overwrite().save("/tmp/spark-logistic-regression-model")

// Make predictions on test documents.
model.transform(test)
```

So what all we have learnt ?

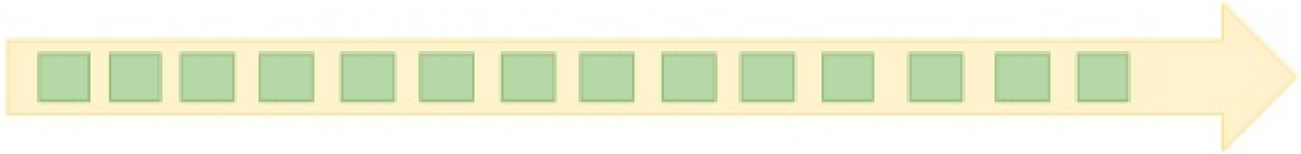
Web:	http://www.insofe.edu.in
Facebook:	https://www.facebook.com/insofe
Twitter:	https://twitter.com/Insofeedu
YouTube:	http://www.youtube.com/InsofeVideos
SlideShare:	http://www.slideshare.net/INSOFE
LinkedIn:	http://www.linkedin.com/company/international-school-of-engineering

Streaming Data Analysis

Data at rest Vs Data in motion

- At rest:
  - Dataset is fixed
  - a.k.a bounded
  - can go back and forth on the data
- In motion:
  - continuously incoming data
  - a.k.a unbounded
  - too large to store and then process
  - need to process in one pass

- Generally Big data has velocity
  - continuous data
- Difference lies in when are you analyzing your data?
  - after the event occurs ⇒ at rest
  - as the event occurs ⇒ in motion



Examples

- Data at rest
  - Finding stats about group in a closed room
  - Analyzing sales data for last month to make strategic decisions
- Data in motion
  - Finding stats about group in a marathon
  - Monitoring the health of a data center

Batch processing

- Problem statement :
    - Process this entire data
    - give answer for X at the end

- Characteristics
    - Access to entire data
    - Split decided at the launch time.
    - Capable of doing complex analysis (e.g. Model training)
    - Optimize for Throughput (data processed per sec)
- Example frameworks : Map Reduce, Apache Spark

Stream processing

- Problem statement :
  - Process incoming stream of data
  - to give answer for X at this moment.
- Characteristics
  - Results for X are based on the current data
  - Computes function on one record or smaller window.
  - Optimizations for latency (avg. time taken for a record)
- Example frameworks: Apache Storm, Apache Flink, Amazon Kinesis

Why Streaming?

Many important applications must process large streams of live data and provide results in near-real-time

- Intrusion detection systems
- Fraud detection
- Location analysis in transportation
- Social network trends, website statistics, etc

Batch vs Streaming

Batch

Streaming

When to use Batch vs Streaming

- Answers for current snapshot, low latency requirements (< 1s) ⇒ Real-time
  - Answers at the end ⇒ Open
- Complex calculations, multiple iterations over entire data ⇒ Batch
  - Simple computations, each record can be processed independently ⇒ Open
- Depends on use-case
  - Some use-cases can be solved by any one
  - Some other might need combination of two.

Can one replace the other?

- Batch processing is designed for ‘data at rest’. ‘data in motion’ becomes stale; if processed in batch mode.
- Real-time processing is designed for ‘data in motion’. But, can be used for ‘data at rest’ as well (in many cases).

Quiz : is this Batch or Real-time?

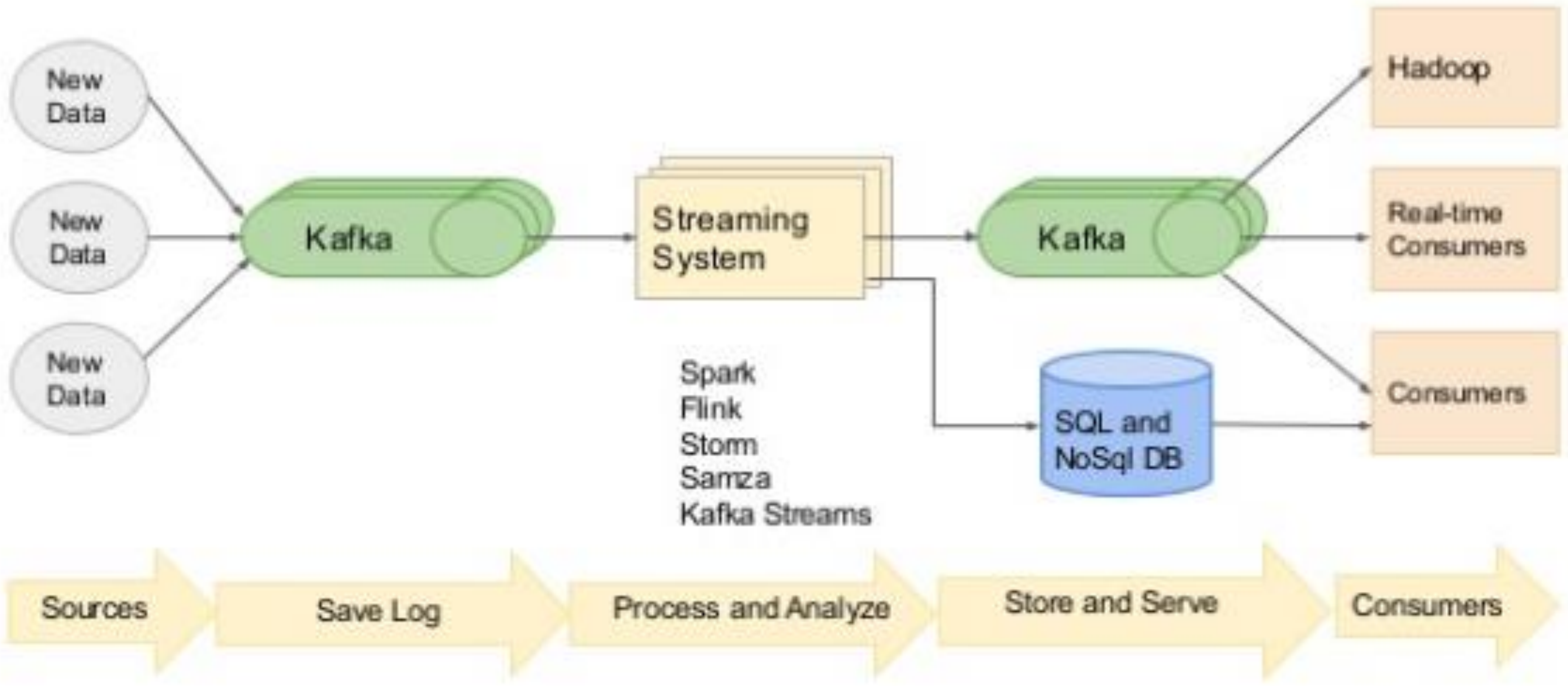
Queue for roller coaster ride

Queue at the petrol pump

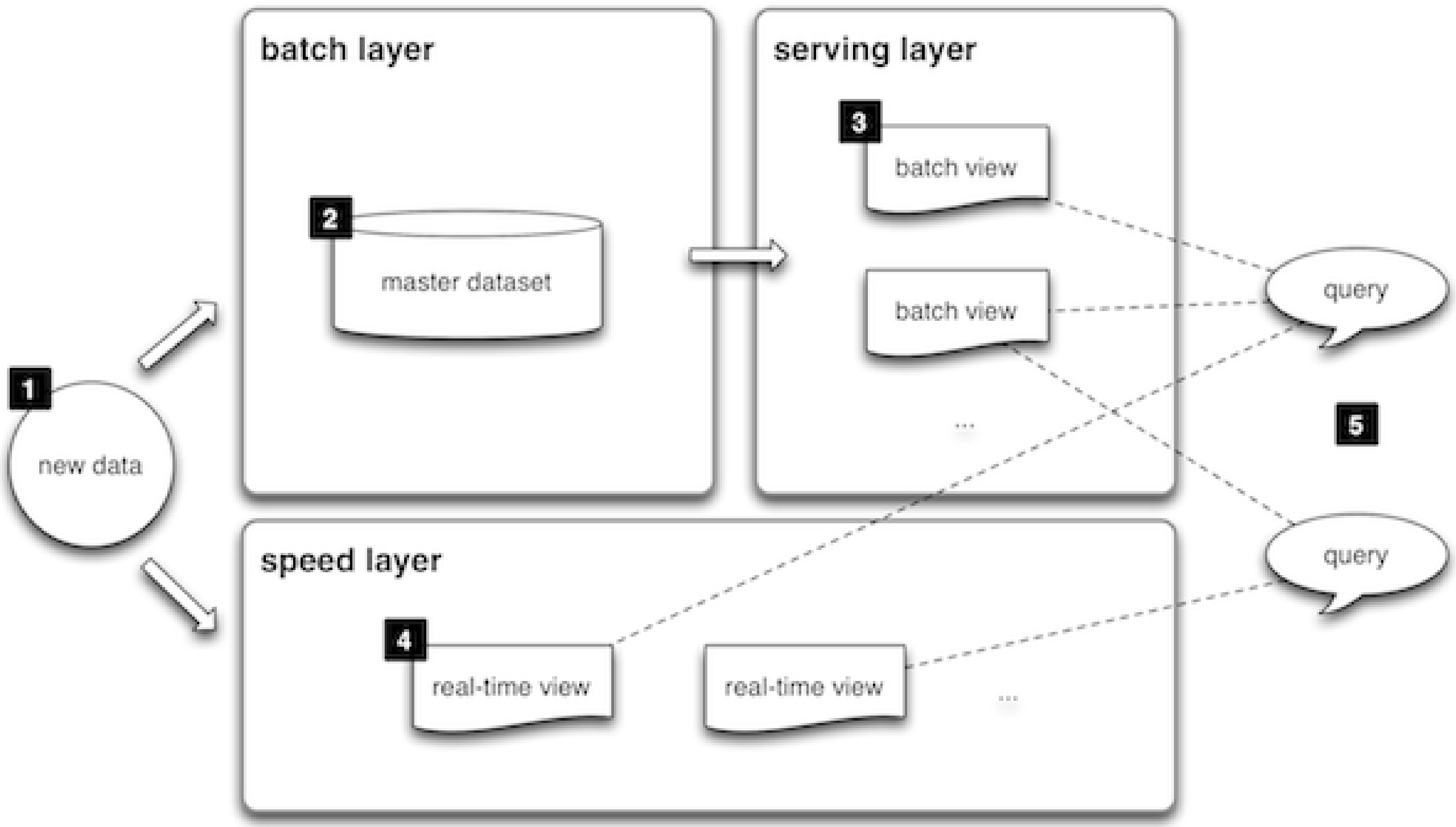
Micro-batching

- A special case of batch processing with very small batch sizes (tiny)
- A nice mix between batching and streaming
- At cost of latency
- Allows Stateful computation, making windowing an easy task
- Example Frameworks: Spark Streaming, Storm Trident

Streaming Architecture

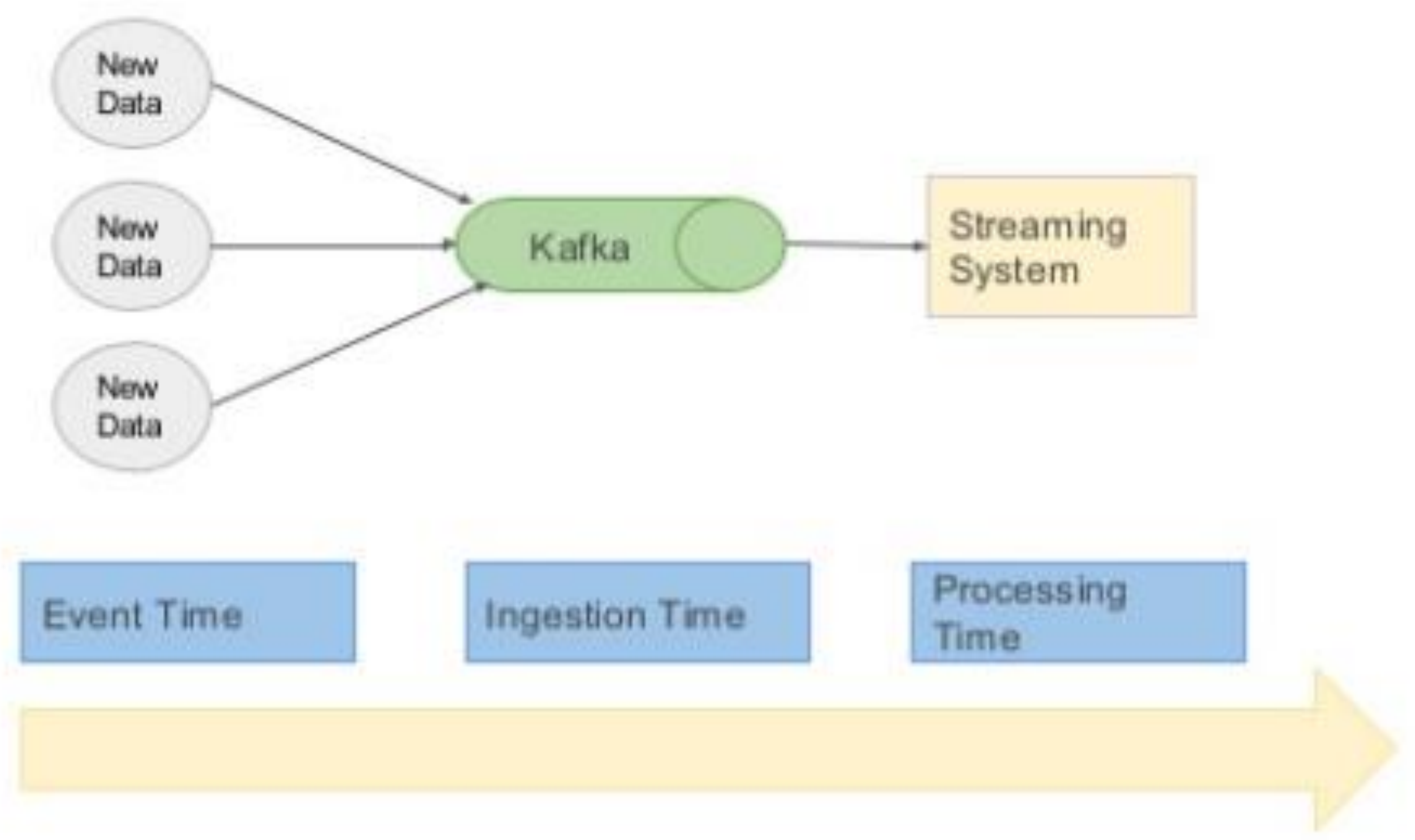


Lambda Architecture



Streaming Concepts

Streaming Concepts - Time



Streaming Concepts - Window

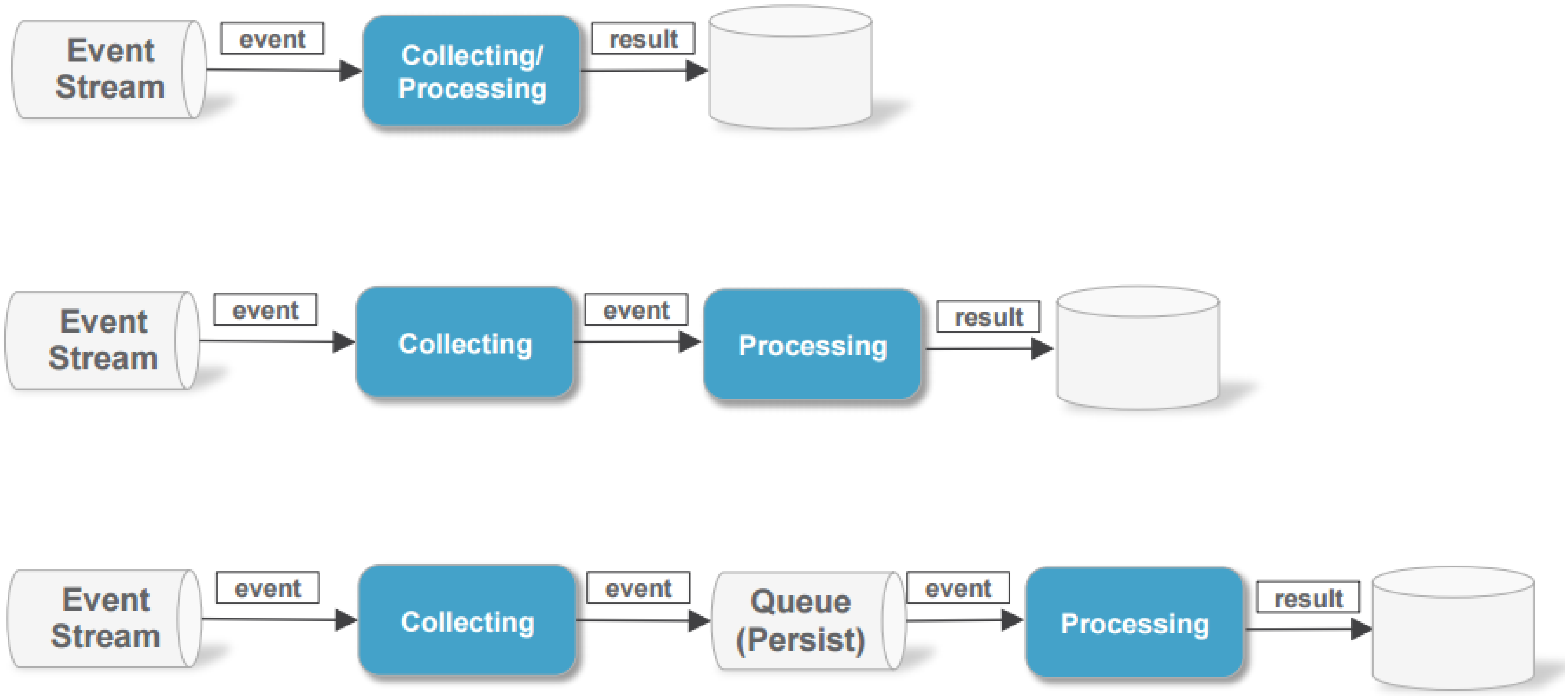
Streaming Concepts - Message Delivery

- At most once [0,1]
  - Messages may be lost
  - Messages never re-delivered
- At least once [1 .. n]
  - Messages will never be lost
  - but messages may be re-delivered (might be ok if consumer can handle it)
- Exactly once [1]
  - Messages are never lost
  - Messages are never re-delivered
  - Perfect message delivery
  - Incurs higher latency for transactional semantics

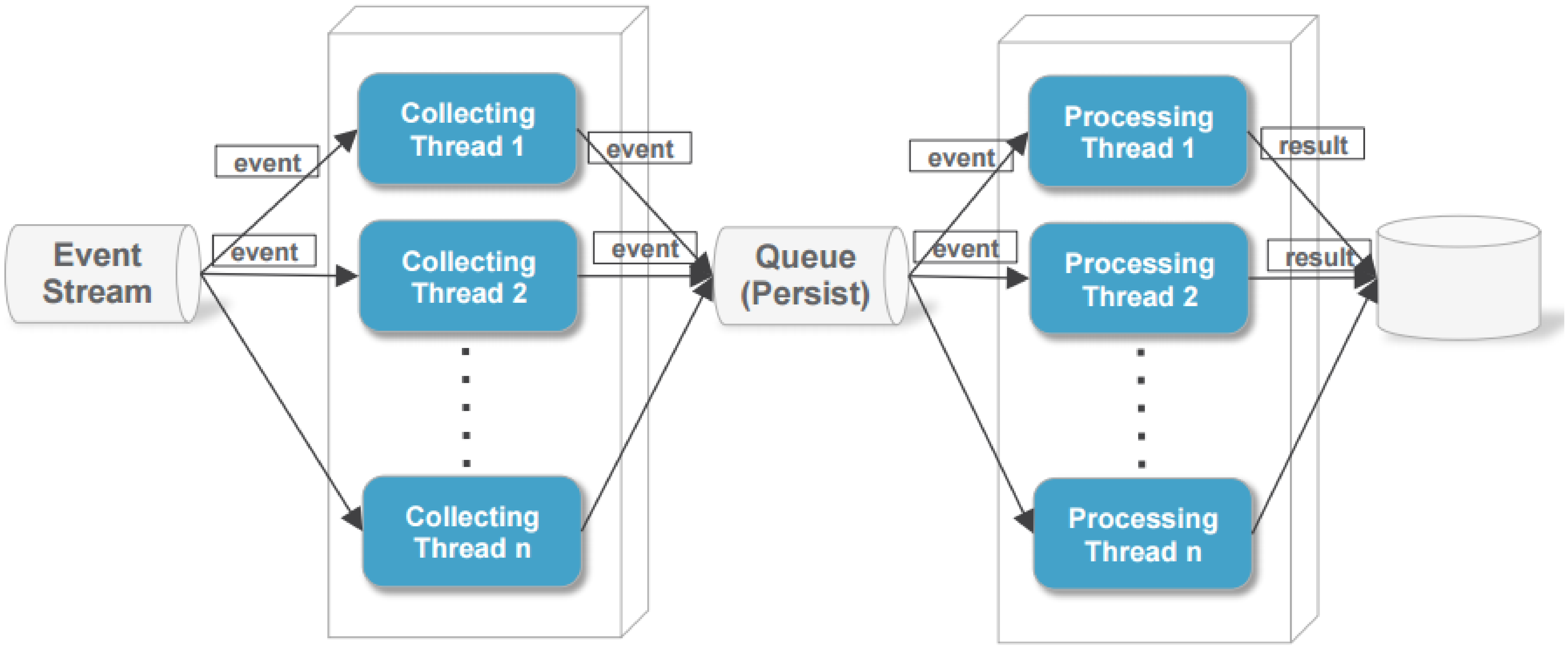
Designing a Streaming System - Requirements

- **Scalable** to large clusters
- **Second-scale** latencies
- **Simple** programming model
- **Integrated** with batch & interactive processing
- **Efficient fault-tolerance** in stateful computations

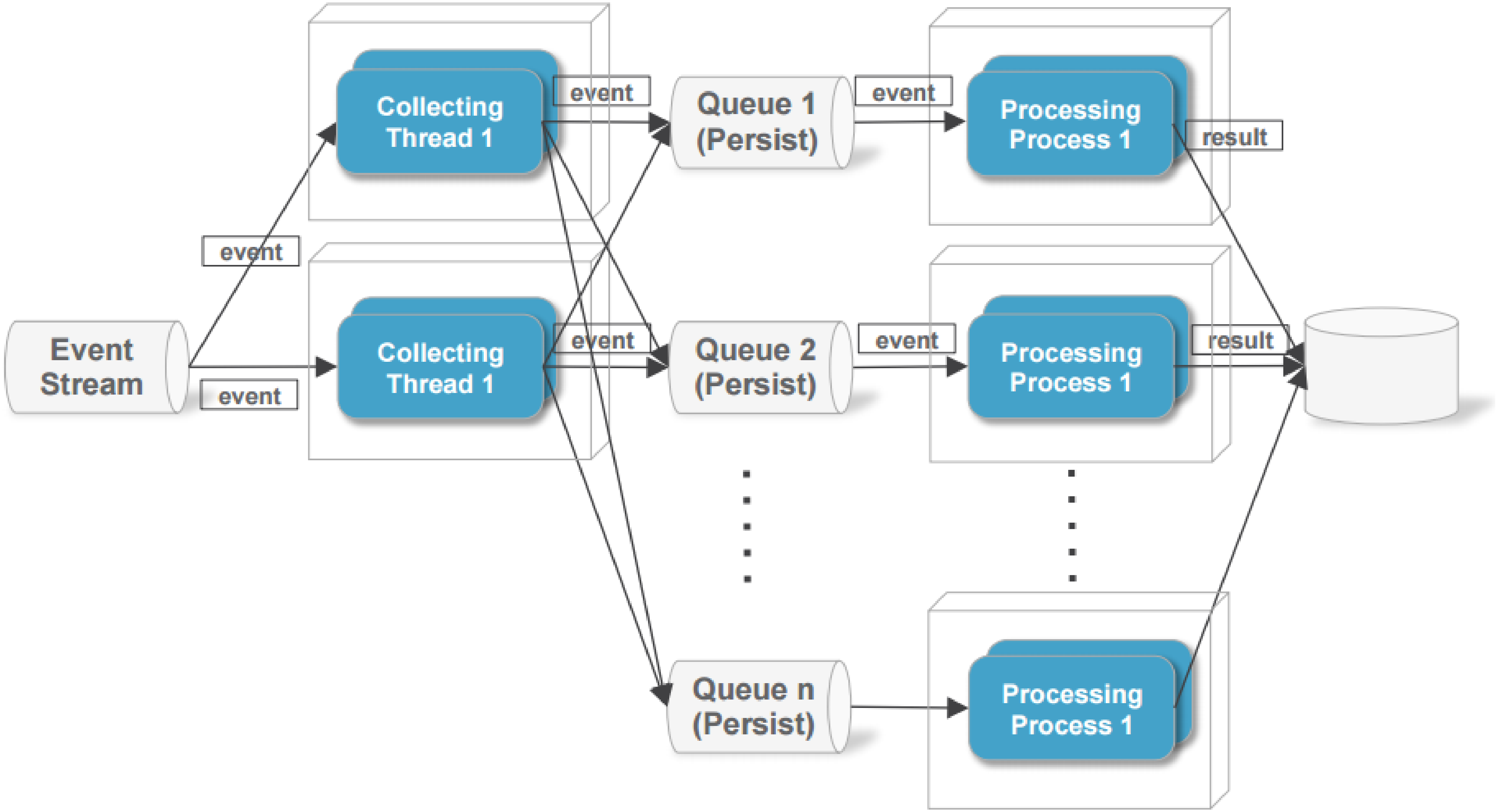
How to design a Stream Processing System?



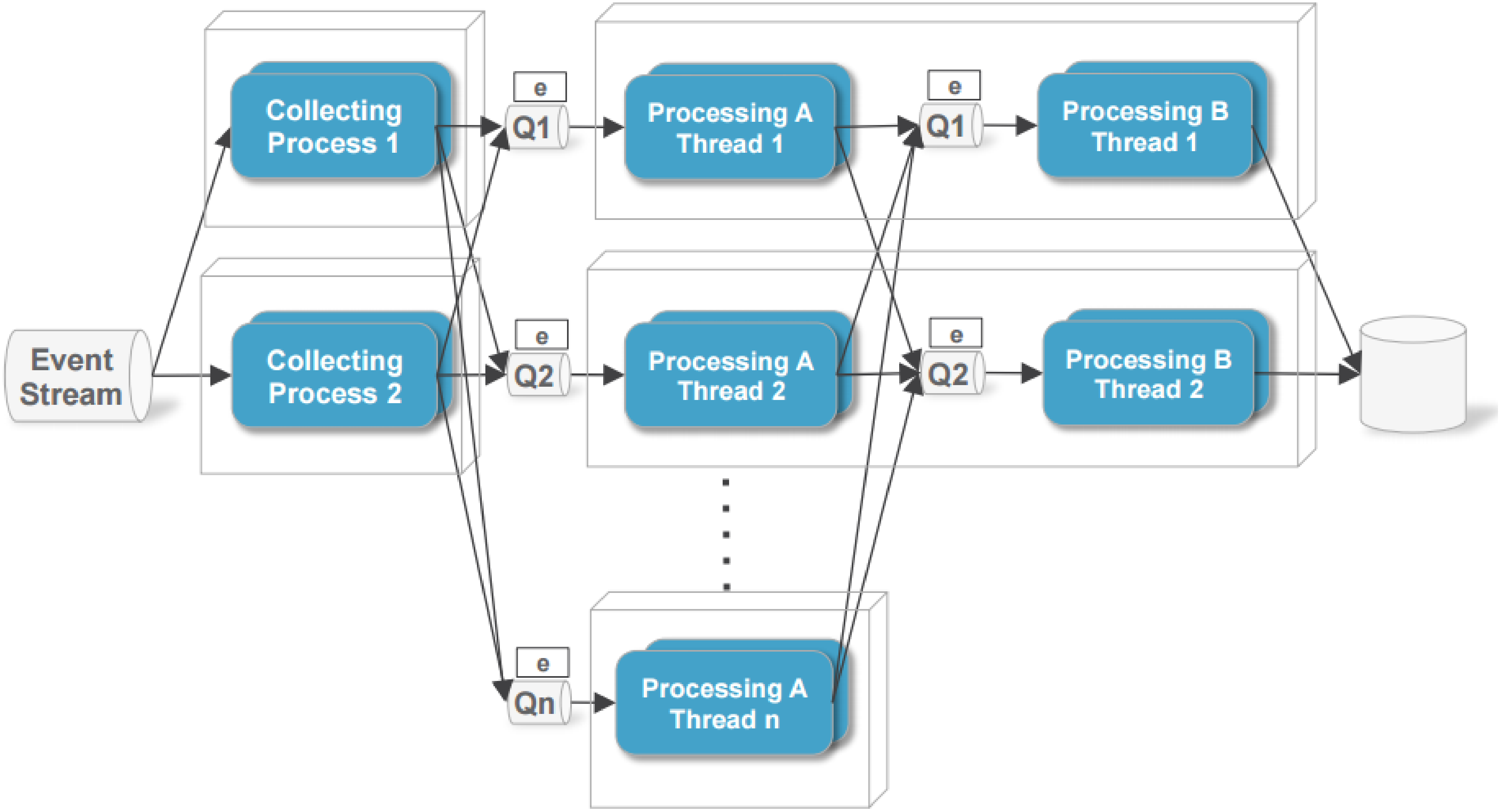
How to scale a Stream Processing System?



How to scale a Stream Processing System?

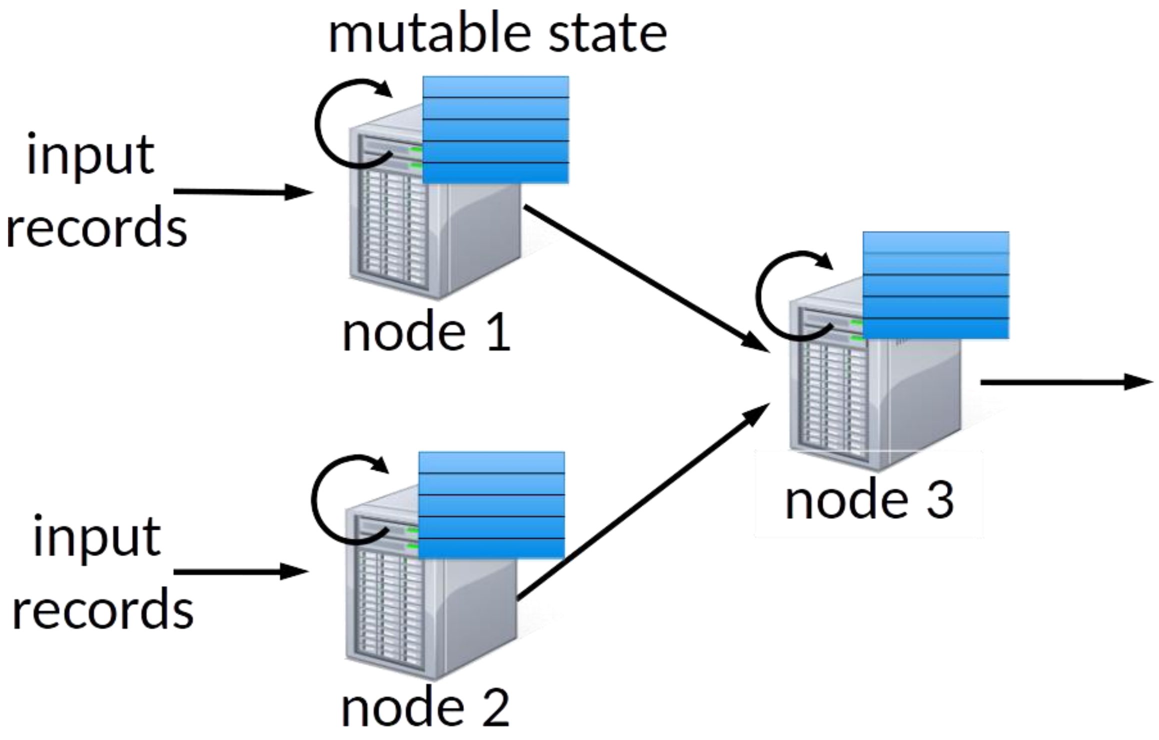


How to scale a Stream Processing System?



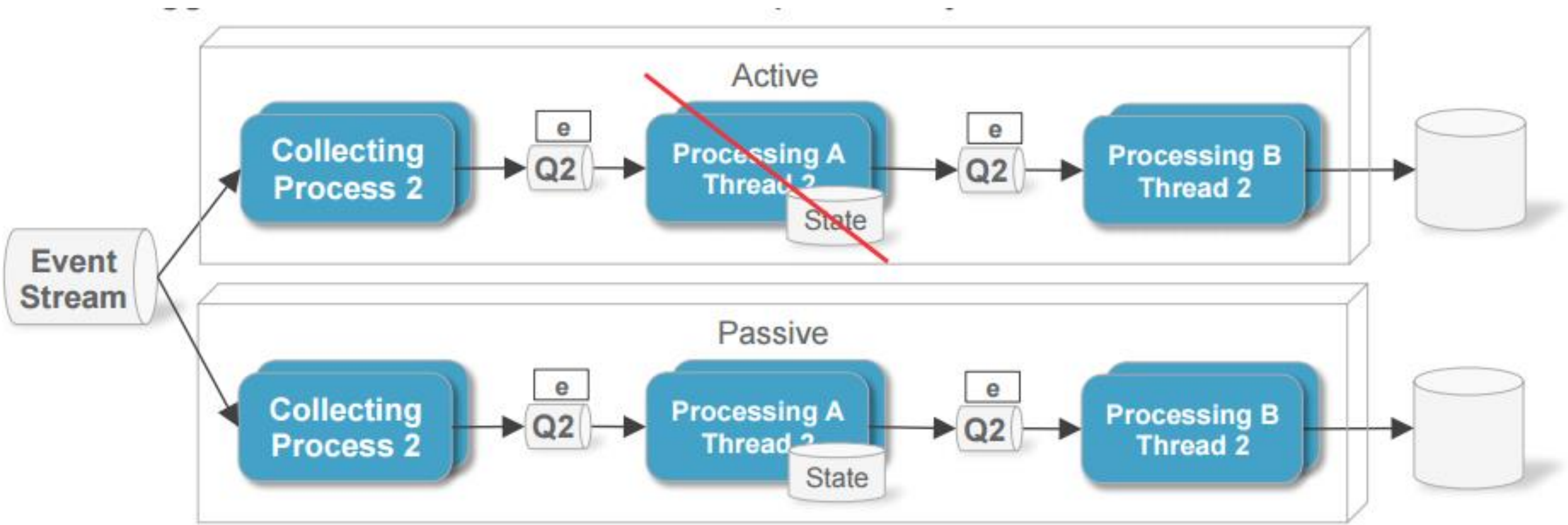
Stateful Stream Processing

- Traditional streaming systems have a event-driven record-at-a-time processing model
  - Each node has mutable state
  - For each record, update state & send new records
- State is lost if node dies!
- Making stateful stream processing be fault-tolerant is challenging



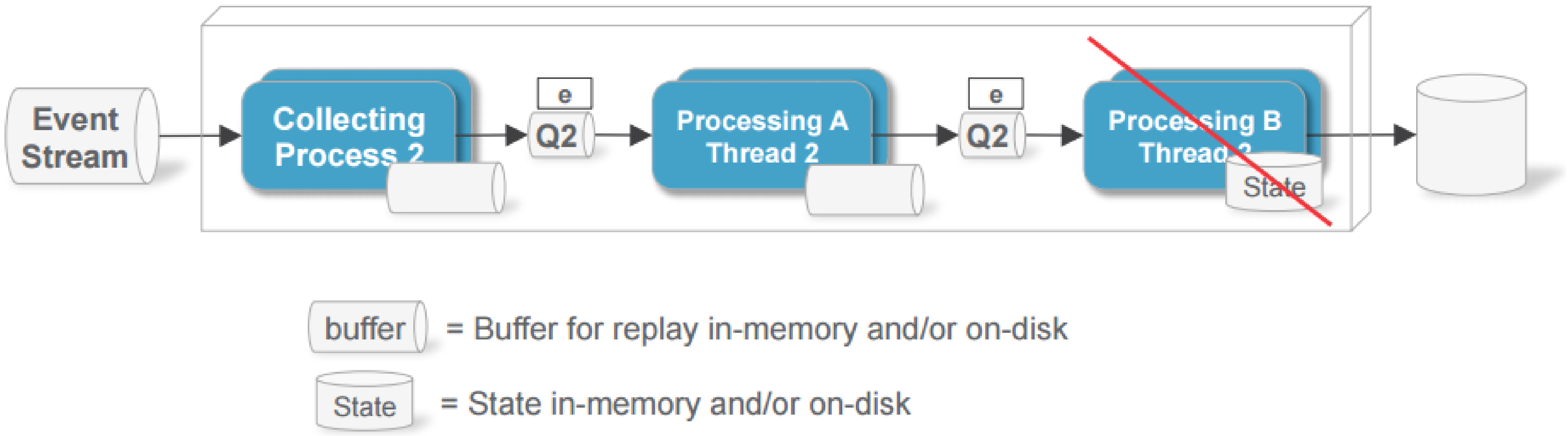
How to make (stateful) Stream Processing System reliable?

- Solution 1: using active/passive system (hot replication)
  - Both systems process the full load
  - In case of a failure, automatically switch and use the “passive” system
  - Stragglers slow down both active and passive system



How to make (stateful) Stream Processing System reliable?

- Solution 2: Upstream backup
  - Nodes buffer messages and replay them to new node in case of failure
  - Stragglers are treated as failures



Levels of abstraction

- Basic
  - Low level apis
  - User controls the topology and distribution
  - More control, but also more burden on the developer
- High
  - Provides abstraction for commonly occurring patterns and operators
  - For example:
    - table and SQL interface over streams by systems such as Spark Streams, Amazon Kinesis
    - pattern matching semantics by systems such as Cayuga from Cornell, WSO2, TIBCO StreamBase
  - System optimizes execution, but lesser control to the developer

Comparison of Streaming Systems

	Flume	NiFi	Gearpump	Apex	Kafka Streams	Spark Streaming	Storm	Storm + Trident	Samza	Flink	Ignite Streaming	Beam (*GC DataFlow)
Current version	1.6.0	0.6.1	incubating	3.3.0	0.9.0.1* (available in 0.10)	1.6.1	1.0.0	1.0.0	0.10.0	1.0.2	1.5.0	incubating
Category	DC/SEP	DC/SEP	SEP	DC/ESP	ESP	ESP	ESP/CEP	ESP/CEP	ESP	ESP/CEP	ESP/CEP	SDK
Event size	single	single	single	single	single	micro-batch	single	mini-batch	single	single	single	single
Available since (incubator since)	June 2012 (June 2011)	July 2015 (Nov 2014)	(Mar 2016)	Apr 2016 (Aug 2015)	Apr 2016 (July 2011)	Feb 2014 (2013)	Sep 2014 (Sep 2013)	Sep 2014 (Sep 2013)	Jan 2014 (July 2013)	Dec 2014 (Mar 2014)	Sep 2015 (Oct 2014)	(Feb 2016)
Contributors	26	67	19	53	160	838	207	207	48	159	56	80
Main backers	Apple Cloudera	Hortonworks	Intel Lightbend	Data Torrent	Confluent	AMPLab Databricks	Backtype Twitter	Backtype Twitter	LinkedIn	dataArtisans	GridGain	Google
Delivery guarantees	at least once	at least once	exactly once at least once (with non-fault-tolerant sources)	exactly once	at least once	exactly once at least once (with non-fault-tolerant sources)	at least once	exactly once	at least once	exactly once	at least once	exactly once*
State management	transactional updates	local and distributed snapshots	checkpoints	checkpoints	local and distributed snapshots	checkpoints	record acknowledgements	record acknowledgements	local snapshots distributed snapshots (fault-tolerant)	distributed snapshots	checkpoints	transactional updates*
Fault tolerance	yes (with file channel only)	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes*
Out-of-order processing	no	no	yes	no	yes	no	yes	yes	yes (but not within a single partition)	yes	yes	yes*
Event prioritization	no	yes	programmable	programmable	programmable	programmable	programmable	programmable	yes	programmable	programmable	programmable
Windowing	no	no	time-based	time-based	time-based	time-based	time-based count-based	time-based count-based	time-based	time-based count-based	time-based count-based	time-based
Back-pressure	no	yes	yes	yes	N/A	yes	yes	yes	yes	yes	yes	yes*
Primary abstraction	Event	FlowFile	Message	Tuple	KafkaStream	DStream	Tuple	TridentTuple	Message	DataStream	IgniteDataStreamer	PCollection
Data flow	agent	flow (process group)	streaming application	streaming application	process topology	application	topology	topology	job	streaming dataflow	job	pipeline
Latency	low	configurable	very low	very low	very low	medium	very low	medium	low	low (configurable)	very low	low*
Resource management	native	native	YARN	YARN	Any process manager (e.g. YARN, Mesos, Chef, Puppet, Salt, Kubernetes, ...)	YARN Mesos	YARN Mesos	YARN Mesos	YARN	YARN	YARN Mesos	integrated*
Auto-scaling	no	no	no	yes	yes	yes	no	no	no	no	no	yes*
In-flight modifications	no	yes	yes	yes	yes	no	yes (for resources)	yes (for resources)	no	no	no	no
API	declarative	compositional	declarative	declarative	declarative	declarative	compositional	compositional	compositional	declarative	declarative	declarative
Primarily written in	Java	Java	Scala	Java	Java	Scala	Clojure	Java	Scala	Java	Java	Java
API languages	text files Java	REST (GUI)	Scala Java	Java	Java	Scala Java Python	Scala Java Clojure Python Ruby	Java Python Scala	Java	Java Scala Python	Java .NET C++	Java*
Notable users	Meebo Sharethrough SimpleGeo	N/A	Intel Levi's Honeywell	Capital One GE Predix PubMatic	N/A	Kelkoo Localytics AsiaInfo Opentable Faimdata Guavus	Yahoo! Spotify Groupon Flipboard The Weather Channel Alibaba Baidu Yelp WebMD	Klout GumGum CrowdFlower	LinkedIn Netflix Intuit Uber	King Otto Group	GridGain	N/A

https://twitter.com/ianhellstrom/status/710917506412716033

Simplified!

Uses Kafka

	Apache Spark	Flink	Apache Storm	samza	Kafka Stream
Processing Model	Mini Batch	Event level	Event level	Event level	Event level
Guarantee	Exactly Once	Exactly Once	At least once	At least once	At least once
State Management	Yes	Yes	No	Yes	Yes
Latency	Medium	Low	Low	Low	Low
Built in primitives	Batch and streaming	Batch and streaming	Low Level API	Low level API	Streaming only
Back Pressure	Yes	Yes	No	via Kafka	via Kafka

Batch first　　Stream first

References & Acknowledgements (for Slides on Streaming!)

- Introduction to Real-time data processing. Yogi Devendra. https://www.slideshare.net/DevendraVyavahare/batch-processing-vs-real-time-data-processing-streaming
- Streaming Analytics. Ashish Gupta, Neera Agarwal https://www.slideshare.net/NeeraAgarwal2/streaming-analytics
- Apache Storm vs. Spark Streaming – Two Stream Processing Platforms compared. Guido Schmutz. https://www.slideshare.net/gschmutz/apache-storm-vs-spark-streaming-two-stream-processing-platforms-compared
- Spark Streaming. Tathagata Das. Strata 2013.
- A Deep Dive into Structured Streaming. Tathagata Das. https://www.slideshare.net/databricks/a-deep-dive-into-structured-streaming

- https://www.linkedin.com/pulse/apache-spark-rdd-vs-dataframe-dataset-chandan-prakash
- Kafka https://kafka.apache.org/
- https://www.domo.com/blog/data-never-sleeps-4-0/
- The world beyond batch: Streaming 101 http://radar.oreilly.com/2015/08/the-world-beyond-batch-streaming-101.html
- Data in motion vs. data at rest | Internap http://www.internap.com/2013/06/20/data-in-motion-vs-data-at-rest/
- How FAST is Credit Card Fraud Detection | FICO http://www.fico.com/en/latest-thinking/infographic/how-fast-is-credit-card-frauddetection
- Lambda Architecture http://lambda-architecture.net/