Data Engineering

LTAT.02.007

Ass Prof. Riccardo Tommasini

Assistants: Fabiano Spiga, Mohamed Ragab, Hassan Eldeeb

https://courses.cs.ut.ee/2020/dataeng

Forum

Moodle

Talk Announcement

Thursday 19th Lecture Time!

Herminio works as a consultant in IOVIO a Consulting Firm with specialization in Data Projects. In his career Herminio has already worked in more than 50 countries, in all continents, and lived in 13. His career was shaped in the lines of General Electric Nuclear Energy, with a devotion to innovation and quality for mission critical applications. In his professional path he has run through the implementation of data quality solutions around the globe for a wide range of industries including Financial Institutions, Automobile, Insurance, Farma, Mining, Health and Bioengineering.

His enthusiasm to close the gap between academia and industry, has given him the opportunity to share his experience in seminars in collaboration with the University of Valencia, the Politecnido di Milano and the Polytechnic of Zurich.

Today, established in Zurich, Switzerland; he splits his time between his family, friends and nature.

video teaser

Data Velocity[19]

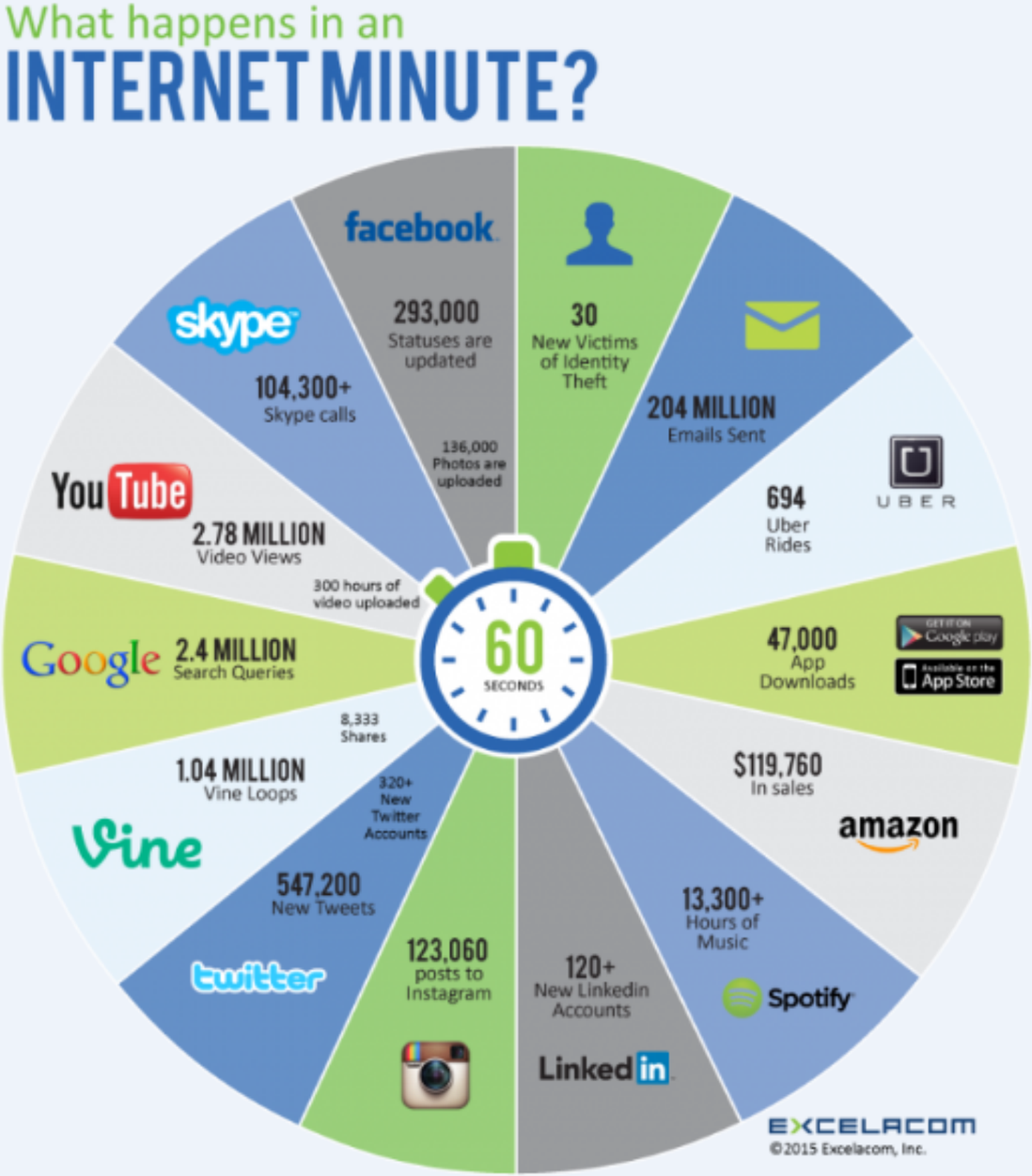
[19] Data velocity represents the speed at which data are consumed and insights are produced.

Data Volume & Velocity

Nowadays, we produce massive amount of data.

And we produce them fast...

...the picture on the right shows what was happening in a single minute of internet traffic in 2015



Moreover, in 2015...

IBM reported that T-Mobile processed more than 17 billion events, including text messages, phone calls, and internet traffic.

According to Hitachi, the amount of data generated by a Level 2[18] connected vehicle exceeded 25GB per hour.

Bombardier showcased its C Series jetliner which is fitted with 5,000 sensors that generate up to 10 GB of data per second.

---

[18] only a limited set of automated functions

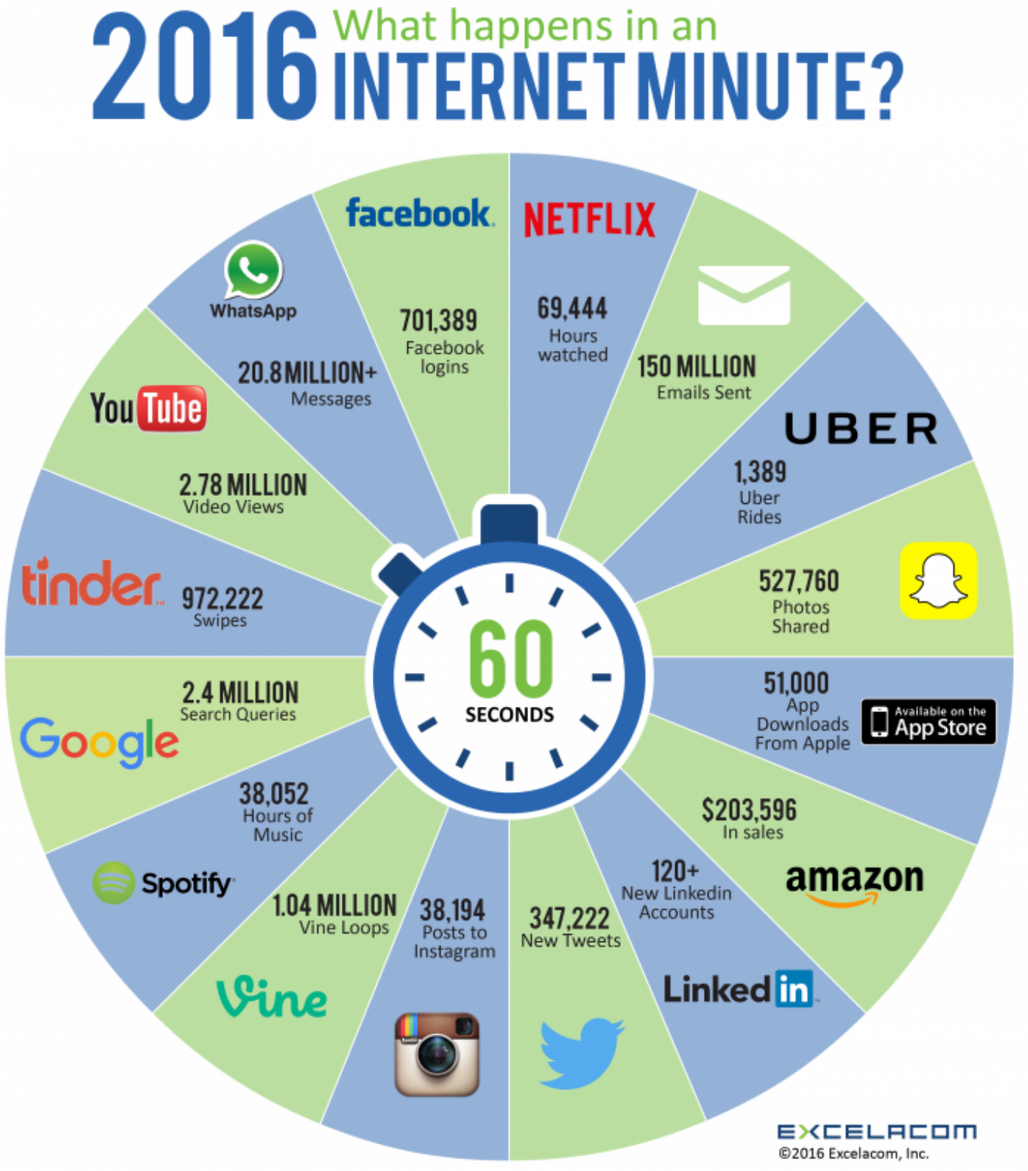
Today?

According to Statista[17] 75.44 billion of IoT devices will be installed worldwide by 2025.

The avionics of a modern commercial jets outputs 10 terabytes (TB) of data, per engine, during every half hour of operation.

and the Web...

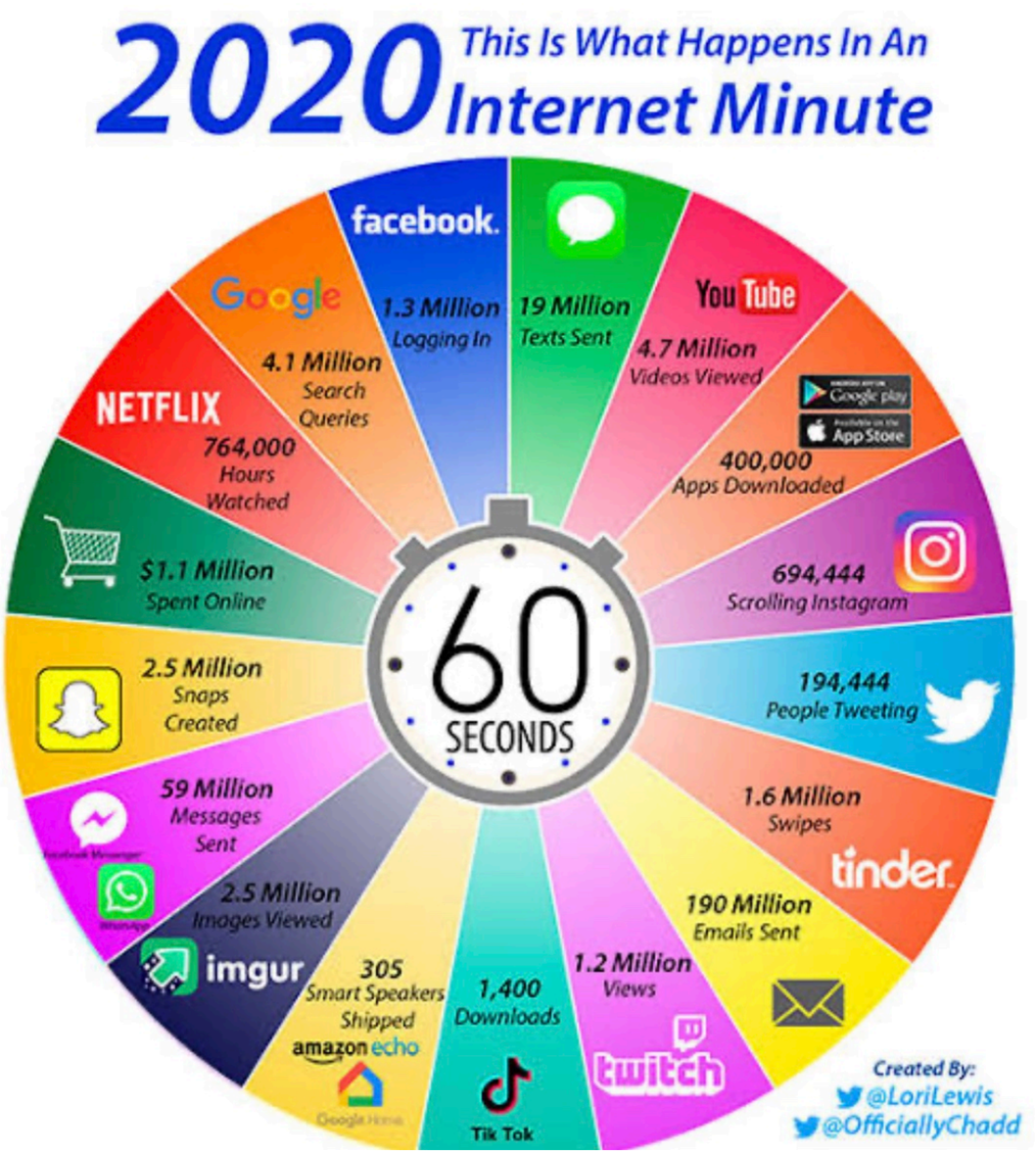
[17] Courtesy of Emanuele Della Valle/Marco Balduini


2016 What happens in an INTERNET MINUTE?
facebook
701,389 Facebook logins
NETFLIX
69,444 Hours watched
150 MILLION Emails Sent
UBER
1,389 Uber Rides
527,760 Photos Shared
51,000 App Downloads From Apple
Available on the App Store
$203,596 In sales
amazon
120+ New Linkedin Accounts
LinkedIn
347,222 New Tweets
38,194 Posts to Instagram
1.04 MILLION Vine Loops
Vine
38,052 Hours of Music
Spotify
2.4 MILLION Search Queries
Google
972,222 Swipes
tinder
2.78 MILLION Video Views
YouTube
20.8 MILLION+ Messages
WhatsApp
60 SECONDS
EXCELACOM
©2016 Excelacom, Inc.

2017 This Is What Happens In An Internet Minute

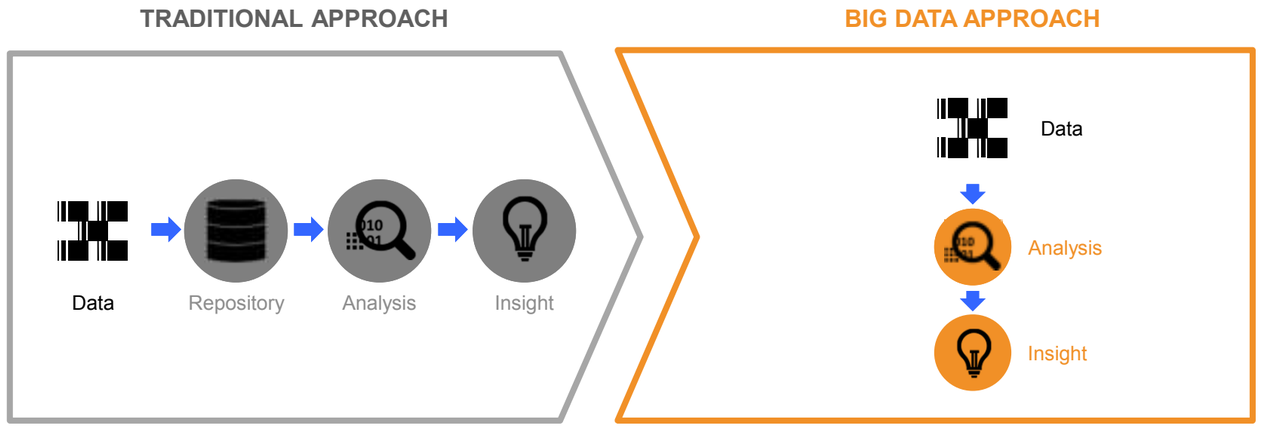

2018 This Is What Happens In An Internet Minute
facebook
973,000 Logins
18 Million Text Messages
YouTube
4.3 Million Videos Viewed
375,000 Apps Downloaded
174,000 Scrolling Instagram
481,000 Tweets Sent
1.1 Million Swipes
187 Million Emails Sent
936,073 Views
67 Voice-First Devices Shipped
38 Million Messages
25,000 GIFs Sent via Messenger
2.4 Million Snaps Created
$862,823 Spent Online
266,000 Hours Watched
NETFLIX
3.7 Million Search Queries
Google
60 SECONDS
Created By:
@LoriLewis
@OfficiallyChadd


2019 This Is What Happens In An Internet Minute
facebook
1 Million
Logging In
18.1 Million
Texts Sent
YouTube
4.5 Million
Videos Viewed
Google
3.8 Million
Search
Queries
NETFLIX
694,444
Hours
Watched
390,030
Apps Downloaded
$996,956
Spent Online
347,222
Scrolling Instagram
60
SECONDS
2.1 Million
Snaps
Created
87,500
People Tweeting
41.6 Million
Messages
Sent
Facebook Messenger
WhatsApp
1.4 Million
Swipes
tinder
4.8 Million
Gifs Served
GIPHY
188 Million
Emails Sent
180
Smart Speakers
Shipped
amazon echo
Google Home
41
Music
Streaming
Subscriptions
1 Million
Views
twitch
Created By:
@LoriLewis
@OfficiallyChadd


2020 This Is What Happens In An Internet Minute
facebook.
1.3 Million
Logging In
19 Million
Texts Sent
YouTube
4.7 Million
Videos Viewed
400,000
Apps Downloaded
694,444
Scrolling Instagram
194,444
People Tweeting
1.6 Million
Swipes
tinder
190 Million
Emails Sent
1.2 Million
Views
twitch
1,400
Downloads
Tik Tok
305
Smart Speakers
Shipped
amazon echo
2.5 Million
Images Viewed
imgur
59 Million
Messages
Sent
2.5 Million
Snaps
Created
$1.1 Million
Spent Online
NETFLIX
764,000
Hours
Watched
Google
4.1 Million
Search
Queries
60
SECONDS
Created By:
@LoriLewis
@OfficiallyChadd

Paradigm shifts enabled by big data

Leverage data as it is captured



Analyze data ***after*** it's been processed and landed in a warehouse or mart

Analyze data ***in motion*** as it's generated, in real-time

Traditional Approach

We have addressed the Volume aspect of big Data

Traditional Approach

We have addressed the Volume aspect of big Data

- Static large data sets

Traditional Approach

We have addressed the Volume aspect of big Data

- Static large data sets
- Partitioned across different nodes

Traditional Approach

We have addressed the Volume aspect of big Data

- Static large data sets
- Partitioned across different nodes
- Processing jobs eventually terminate

Big Data Technologies (1st Gen)

Ingestion (Storage)

Processing (Batch)

Big Data Technologies (1st Gen)

Ingestion (Storage)

- HDFS

Processing (Batch)

Big Data Technologies (1st Gen)

Ingestion (Storage)

- HDFS
- S3

Processing (Batch)

Big Data Technologies (1st Gen)

Ingestion (Storage)

- HDFS
- S3
- Flume

Processing (Batch)

Big Data Technologies (1st Gen)

Ingestion (Storage)

- HDFS
- S3
- Flume

Processing (Batch)

Big Data Technologies (1st Gen)

Ingestion (Storage)

- HDFS
- S3
- Flume

Processing (Batch)

- MapReduce (Hadoop)

Big Data Technologies (1st Gen)

Ingestion (Storage)

- HDFS
- S3
- Flume

Processing (Batch)

- MapReduce (Hadoop)
- Hive

Big Data Technologies (1st Gen)

Ingestion (Storage)

- HDFS
- S3
- Flume

Processing (Batch)

- MapReduce (Hadoop)
- Hive
- Spark

So What?

In many applications domains, the traditional data processing infrastructures are challenged:

So What?

In many applications domains, the traditional data processing infrastructures are challenged:

- Electronic trading

So What?

In many applications domains, the traditional data processing infrastructures are challenged:

- Electronic trading
- Network monitoring

So What?

In many applications domains, the traditional data processing infrastructures are challenged:

- Electronic trading
- Network monitoring
- Fraud detection

So What?

In many applications domains, the traditional data processing infrastructures are challenged:

- Electronic trading
- Network monitoring
- Fraud detection

So What?

In many applications domains, the traditional data processing infrastructures are challenged:

- Electronic trading
- Network monitoring
- Fraud detection
- Social network analysis

So What?

In many applications domains, the traditional data processing infrastructures are challenged:

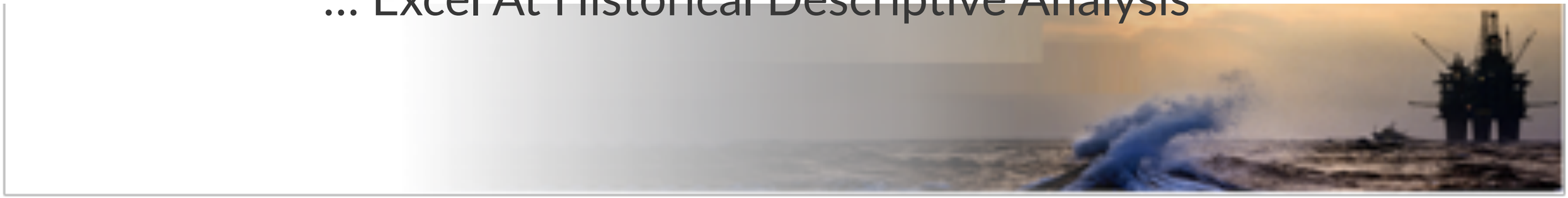
- Electronic trading
- Network monitoring
- Fraud detection
- Social network analysis
- IoT Applications

So What?

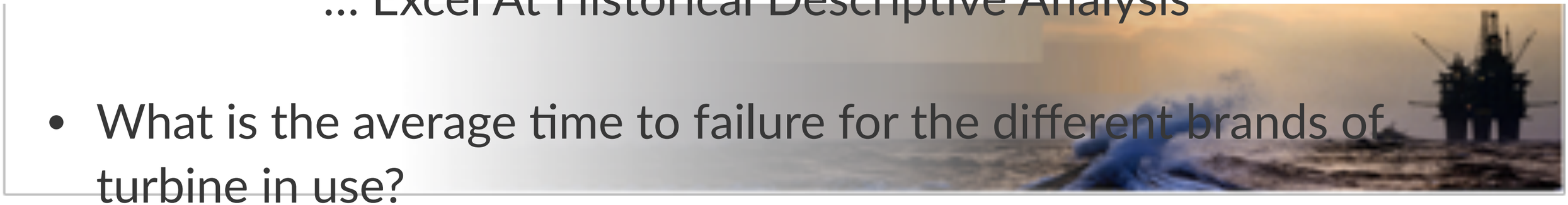
In many applications domains, the traditional data processing infrastructures are challenged:

- Electronic trading
- Network monitoring
- Fraud detection
- Social network analysis
- IoT Applications
- Smart cities

… Excel At Historical Descriptive Analysis

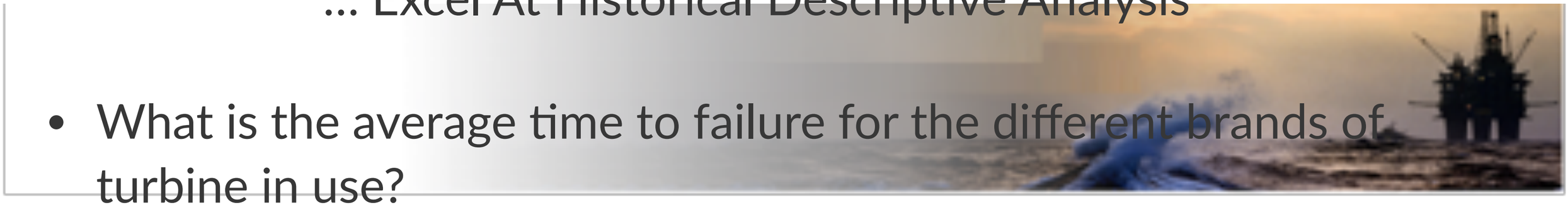


… Excel At Historical Descriptive Analysis



- What is the average time to failure for the different brands of turbine in use?

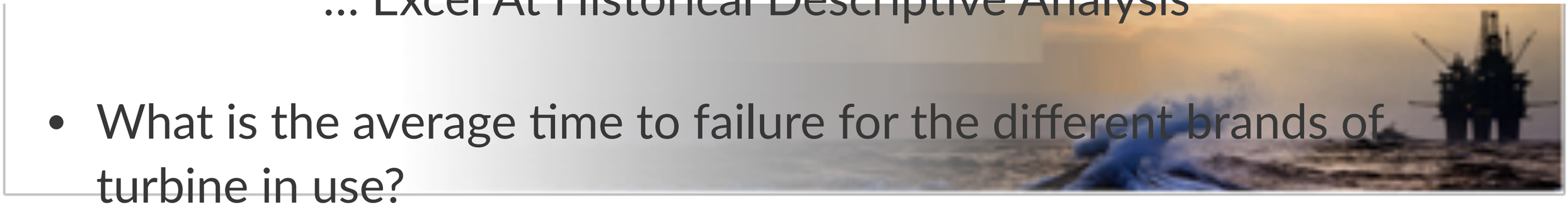
… Excel At Historical Descriptive Analysis



- What is the average time to failure for the different brands of turbine in use?

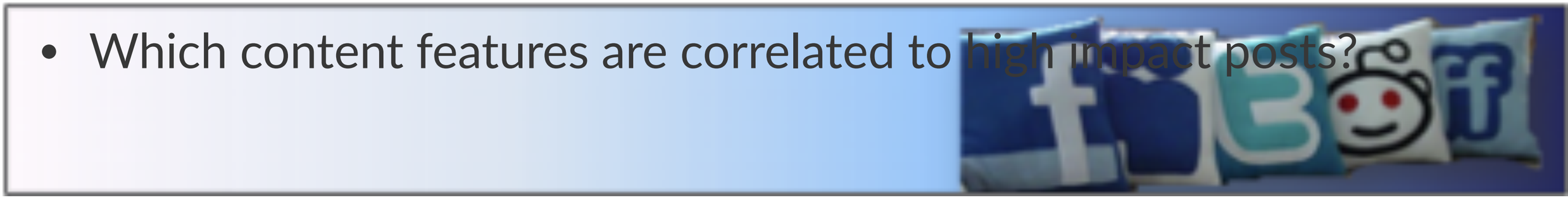
- What is the maximum delay of the public transport per city district?

… Excel At Historical Descriptive Analysis

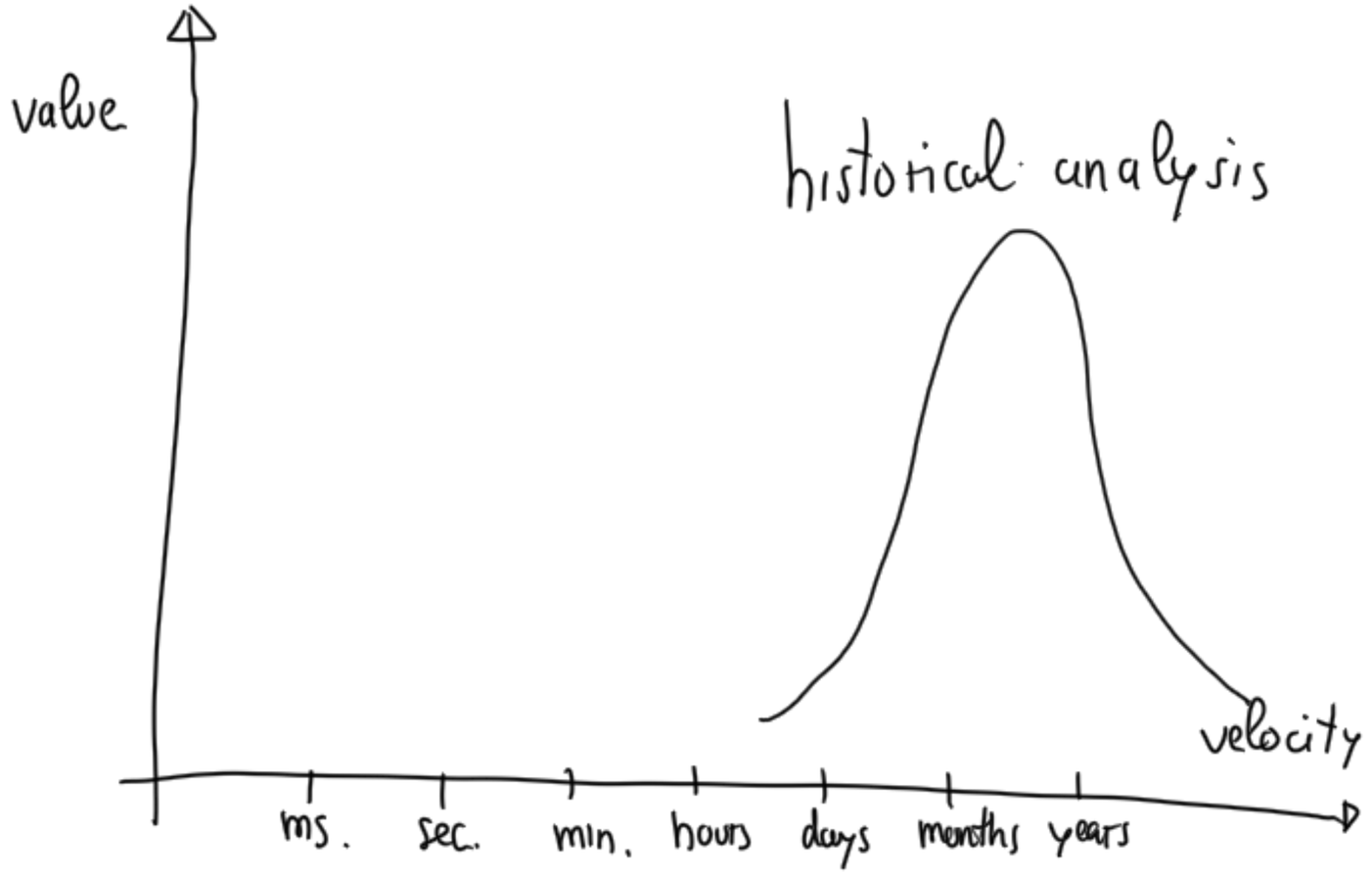


- What is the average time to failure for the different brands of turbine in use?

- What is the maximum delay of the public transport per city district?

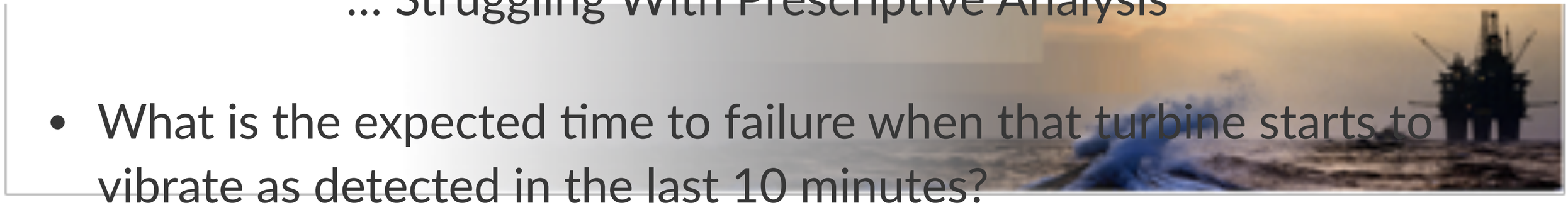


- Which content features are correlated to high impact posts?


value
historical analysis
velocity
ms.
sec.
min.
hours
days
months
years

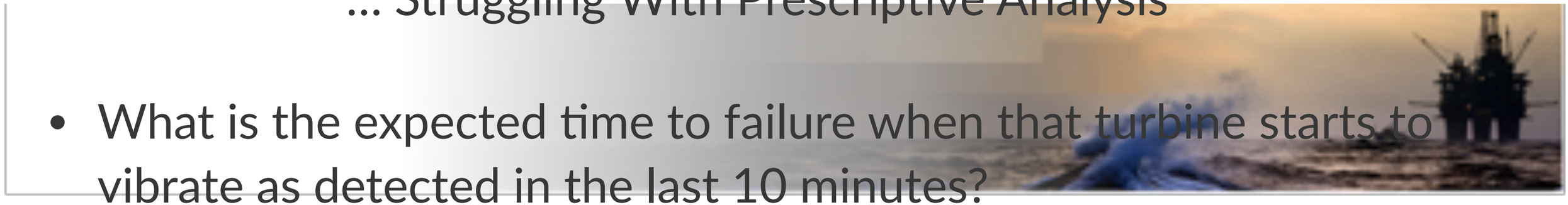
… Struggling With Prescriptive Analysis

… Struggling With Prescriptive Analysis



- What is the expected time to failure when that turbine starts to vibrate as detected in the last 10 minutes?

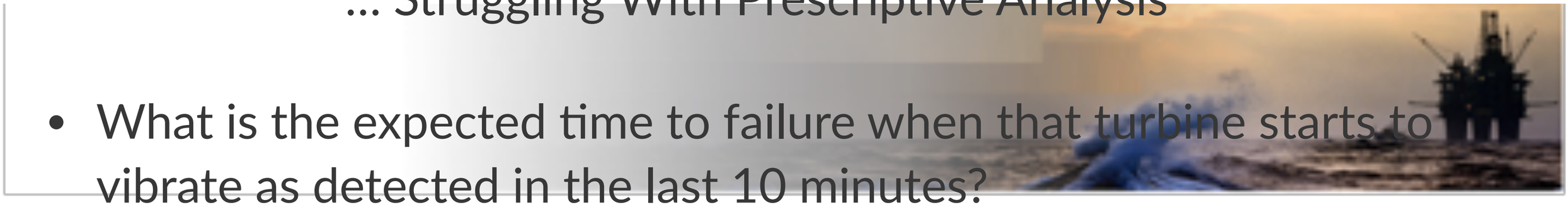
… Struggling With Prescriptive Analysis



- What is the expected time to failure when that turbine starts to vibrate as detected in the last 10 minutes?

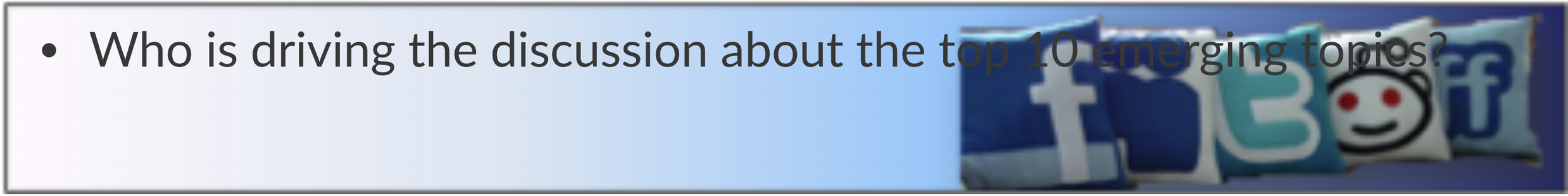
- Can I get to that meeting in the next 15 min using public transport?

… Struggling With Prescriptive Analysis

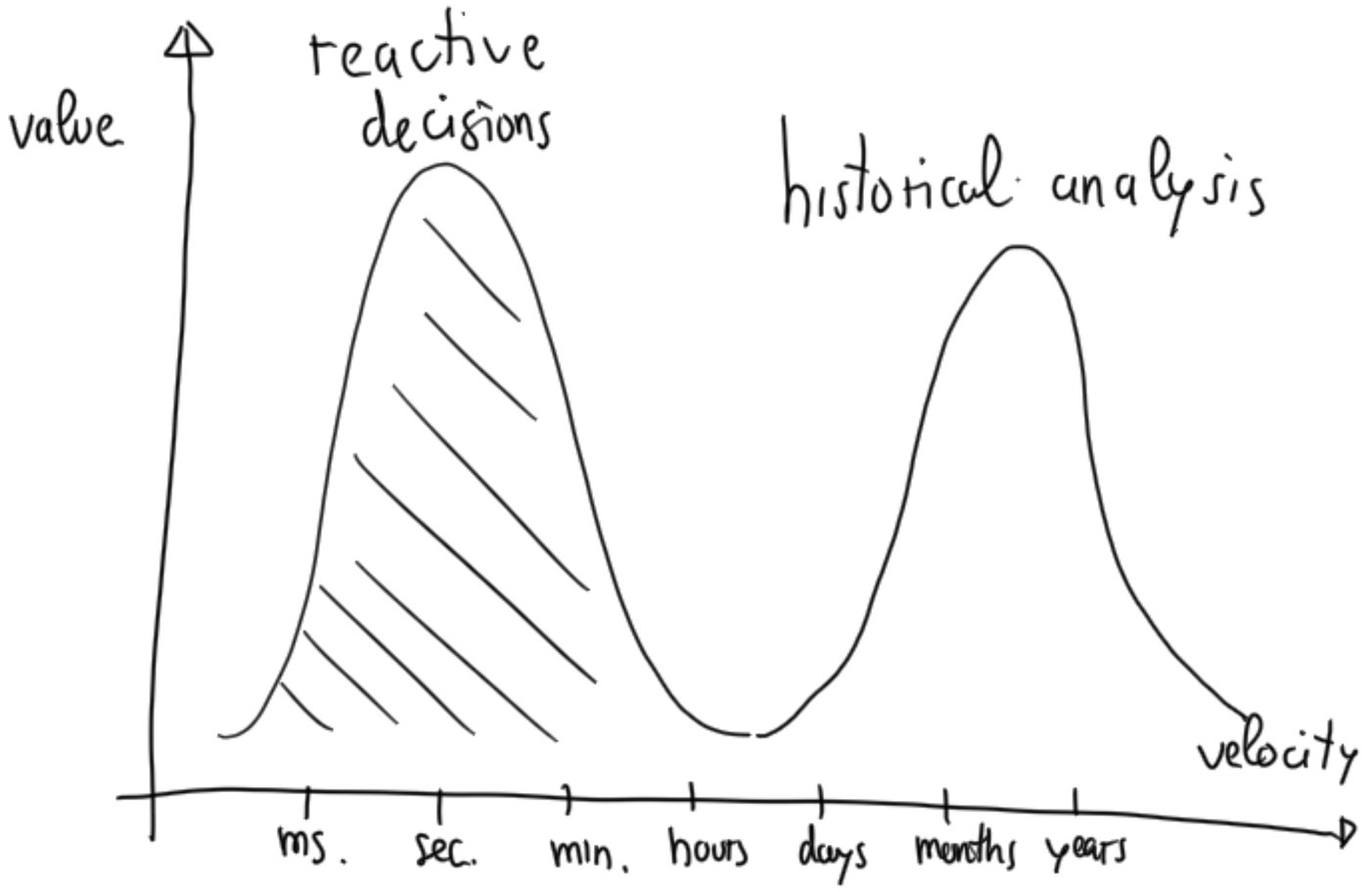


- What is the expected time to failure when that turbine starts to vibrate as detected in the last 10 minutes?

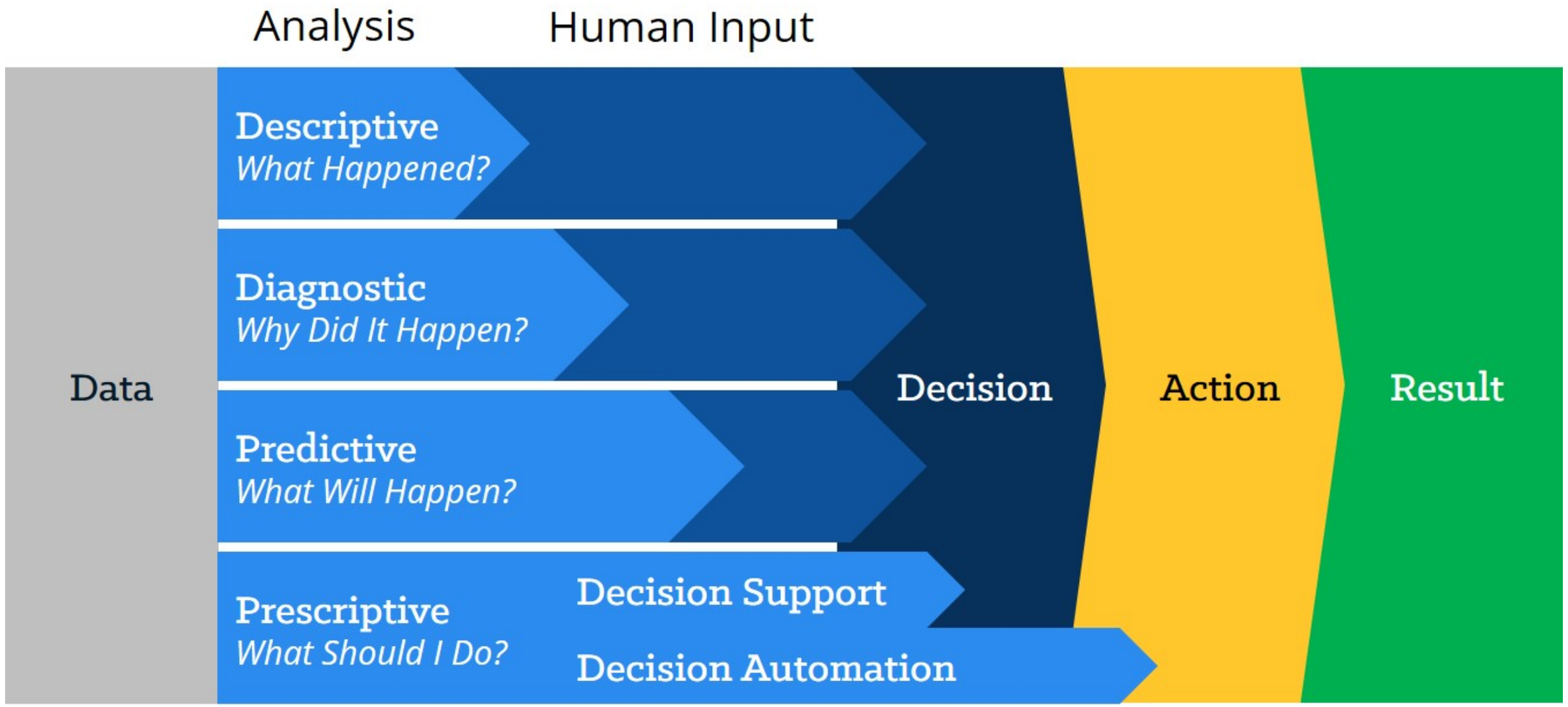
- Can I get to that meeting in the next 15 min using public transport?



- Who is driving the discussion about the top 10 emerging topics?


value
reactive
decisions
historical analysis
velocity
ms.
sec.
min.
hours
days
months
years

Decision Framework



Adapted from Gartner Research, 2015

Stream Processing

What for?

Three interesting characteristics distinguish streams from the data:

What for?

Three interesting characteristics distinguish streams from the data:

- Data Streams are unbounded or infinite

What for?

Three interesting characteristics distinguish streams from the data:

- Data Streams are unbounded or infinite
- Data Streams are ordered, typically time-wise.

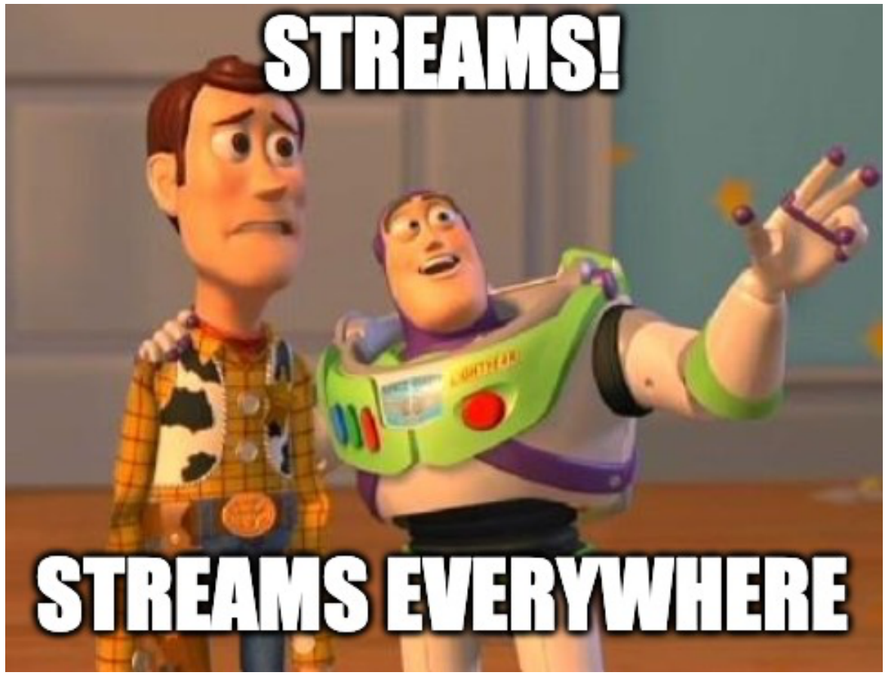
What for?

Three interesting characteristics distinguish streams from the data:

- Data Streams are unbounded or infinite
- Data Streams are ordered, typically time-wise.
- Data are shared in active way, i.e., push model vs pull. is controlled by the source

Where are the streams?

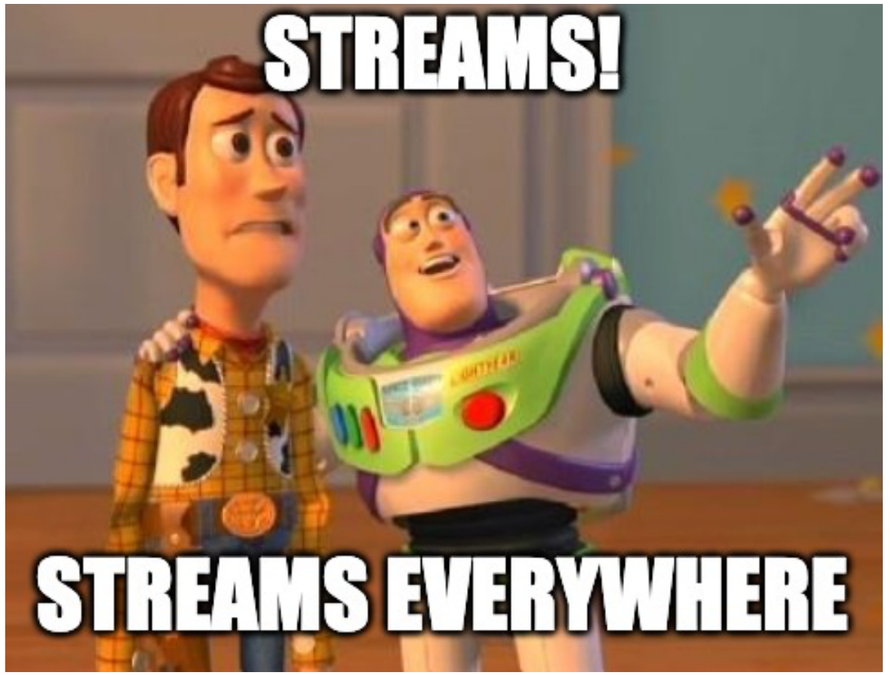
Several sources share data in a streaming form.



Where are the streams?

Several sources share data in a streaming form.

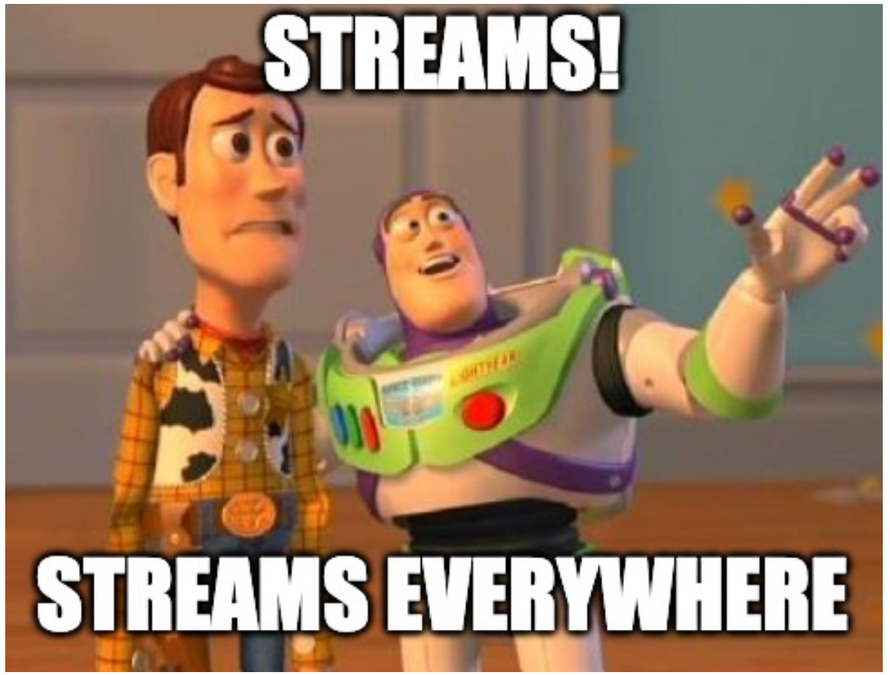
- Stream from clusters, e.g., traces, metrics, and logs.



Where are the streams?

Several sources share data in a streaming form.

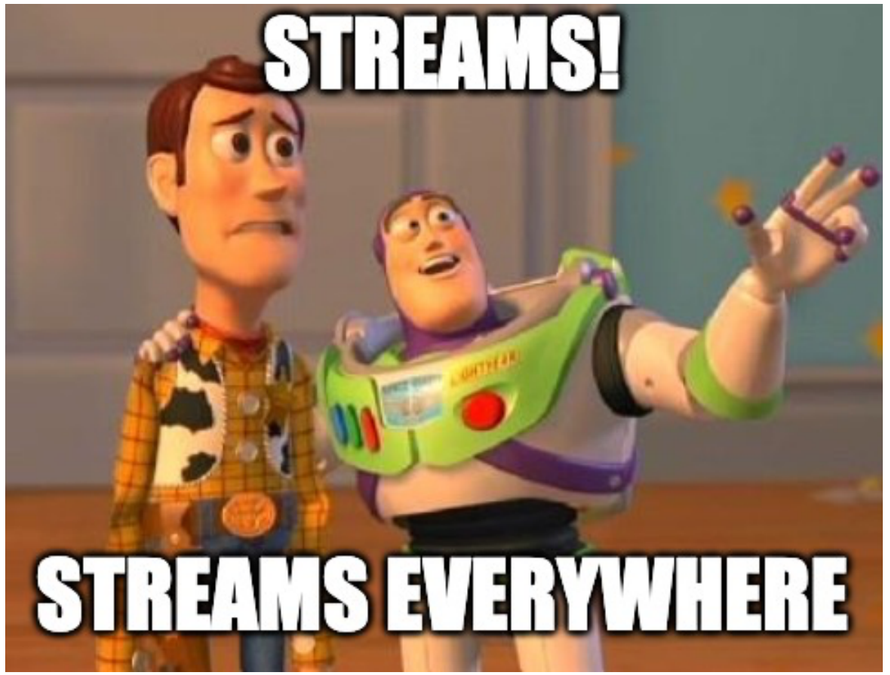
- Stream from clusters, e.g., traces, metrics, and logs.
- Stream from social media, e.g., Twitter feeds.



Where are the streams?

Several sources share data in a streaming form.

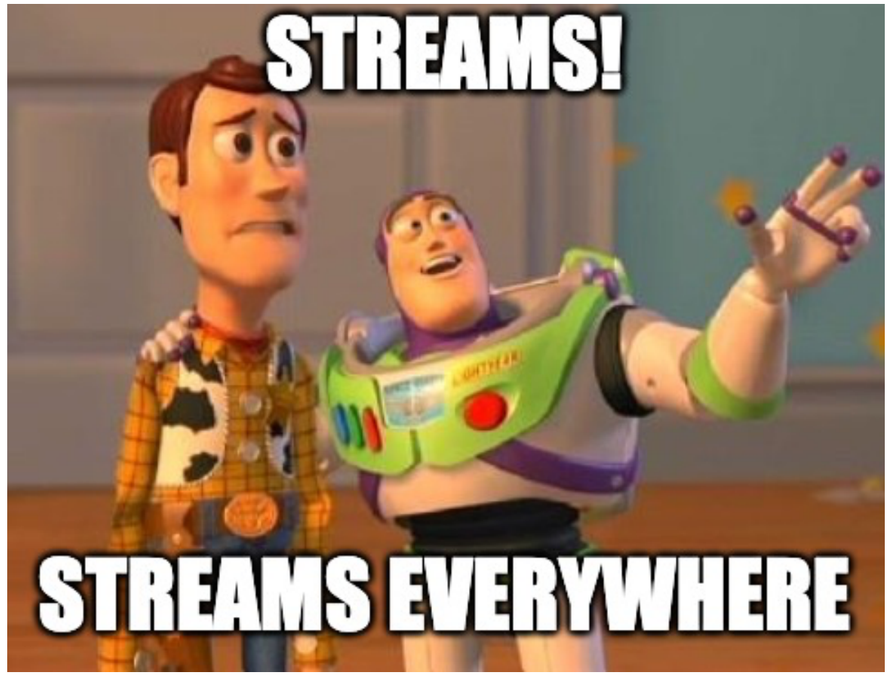
- Stream from clusters, e.g., traces, metrics, and logs.
- Stream from social media, e.g., Twitter feeds.
- Stream from news, e.g., GDELT and Google news.



Where are the streams?

Several sources share data in a streaming form.

- Stream from clusters, e.g., traces, metrics, and logs.
- Stream from social media, e.g., Twitter feeds.
- Stream from news, e.g., GDELT and Google news.
- Stream from sensor networks, e.g., from smart cities.



Data Streams

A stream is an unbounded sequence of data. Typically it is modeled as a sequence of pairs (o, i), where o is an document/tuple/graph and i is a timestamp.

$$(o1, 1), (o2, 2), (o3, 3), (o4, 4), (o5, 5), (oi, i) \ldots\ldots$$

How to Process Streams?

Due to the *unbounded* nature of stream, traditional processing techniques are not adequate.
Moreover, stream analysis typically has strict time constrains.

Thus, **Stream processing** requires a paradigm shift, i.e., from data at rest and post-hoc analyses, to process data **in-motion** and in-motion insights.

8 Requirements for Stream Processing[1]

[1] Stonebraker, Michael, Ugur Cetintemel, and Stan Zdonik. "*The 8 requirements of real-time stream processing*." ACM Sigmod Record 34.4 (2005): 42-47

8 Requirements for Stream Processing[1]

- Keep the data moving

[1] Stonebraker, Michael, Ugur Cetintemel, and Stan Zdonik. "*The 8 requirements of real-time stream processing*." ACM Sigmod Record 34.4 (2005): 42-47

8 Requirements for Stream Processing[1]

- Keep the data moving
  - Prefer active (push) data sharing

[1] Stonebraker, Michael, Ugur Cetintemel, and Stan Zdonik. "*The 8 requirements of real-time stream processing*." ACM Sigmod Record 34.4 (2005): 42-47

8 Requirements for Stream Processing[1]

- Keep the data moving
  - Prefer active (push) data sharing
- Declarative Access

[1] Stonebraker, Michael, Ugur Cetintemel, and Stan Zdonik. "*The 8 requirements of real-time stream processing*." ACM Sigmod Record 34.4 (2005): 42-47

8 Requirements for Stream Processing[1]

- Keep the data moving
  - Prefer active (push) data sharing
- Declarative Access
  - E.g., streamSQL, CQL

[1] Stonebraker, Michael, Ugur Cetintemel, and Stan Zdonik. "*The 8 requirements of real-time stream processing*." ACM Sigmod Record 34.4 (2005): 42-47

8 Requirements for Stream Processing[1]

- Keep the data moving
  - Prefer active (push) data sharing
- Declarative Access
  - E.g., streamSQL, CQL
- Handle imperfections

[1] Stonebraker, Michael, Ugur Cetintemel, and Stan Zdonik. "*The 8 requirements of real-time stream processing*." ACM Sigmod Record 34.4 (2005): 42-47

8 Requirements for Stream Processing[1]

- Keep the data moving
  - Prefer active (push) data sharing
- Declarative Access
  - E.g., streamSQL, CQL
- Handle imperfections
  - Late, missing, unordered items

---

[1] Stonebraker, Michael, Ugur Cetintemel, and Stan Zdonik. "*The 8 requirements of real-time stream processing*." ACM Sigmod Record 34.4 (2005): 42-47

8 Requirements for Stream Processing[1]

- Keep the data moving
  - Prefer active (push) data sharing
- Declarative Access
  - E.g., streamSQL, CQL
- Handle imperfections
  - Late, missing, unordered items
- Predictable outcomes

---

[1] Stonebraker, Michael, Ugur Cetintemel, and Stan Zdonik. "*The 8 requirements of real-time stream processing*." ACM Sigmod Record 34.4 (2005): 42-47

8 Requirements for Stream Processing[1]

- Keep the data moving
  - Prefer active (push) data sharing
- Declarative Access
  - E.g., streamSQL, CQL
- Handle imperfections
  - Late, missing, unordered items
- Predictable outcomes
  - Consistency, event-time processing

---

[1] Stonebraker, Michael, Ugur Cetintemel, and Stan Zdonik. "*The 8 requirements of real-time stream processing*." ACM Sigmod Record 34.4 (2005): 42-47

8 Requirements for Stream Processing[1]

- Keep the data moving
  - Prefer active (push) data sharing
- Declarative Access
  - E.g., streamSQL, CQL
- Handle imperfections
  - Late, missing, unordered items
- Predictable outcomes
  - Consistency, event-time processing

---

[1] Stonebraker, Michael, Ugur Cetintemel, and Stan Zdonik. "*The 8 requirements of real-time stream processing*." ACM Sigmod Record 34.4 (2005): 42-47

8 Requirements for Stream Processing[1]

- Keep the data moving
  - Prefer active (push) data sharing
- Declarative Access
  - E.g., streamSQL, CQL
- Handle imperfections
  - Late, missing, unordered items
- Predictable outcomes
  - Consistency, event-time processing
- Integrate stored and streaming data

---

[1] Stonebraker, Michael, Ugur Cetintemel, and Stan Zdonik. "*The 8 requirements of real-time stream processing*." ACM Sigmod Record 34.4 (2005): 42-47

8 Requirements for Stream Processing[1]

- Keep the data moving
  - Prefer active (push) data sharing
- Declarative Access
  - E.g., streamSQL, CQL
- Handle imperfections
  - Late, missing, unordered items
- Predictable outcomes
  - Consistency, event-time processing
- Integrate stored and streaming data
  - Hybrid stream and batch

[1] Stonebraker, Michael, Ugur Cetintemel, and Stan Zdonik. "*The 8 requirements of real-time stream processing*." ACM Sigmod Record 34.4 (2005): 42-47

8 Requirements for Stream Processing[1]

- Keep the data moving
  - Prefer active (push) data sharing
- Declarative Access
  - E.g., streamSQL, CQL
- Handle imperfections
  - Late, missing, unordered items
- Predictable outcomes
  - Consistency, event-time processing
- Integrate stored and streaming data
  - Hybrid stream and batch
- Data safety and availability

---

[1] Stonebraker, Michael, Ugur Cetintemel, and Stan Zdonik. "*The 8 requirements of real-time stream processing*." ACM Sigmod Record 34.4 (2005): 42-47

8 Requirements for Stream Processing[1]

- Keep the data moving
  - Prefer active (push) data sharing
- Declarative Access
  - E.g., streamSQL, CQL
- Handle imperfections
  - Late, missing, unordered items
- Predictable outcomes
  - Consistency, event-time processing
- Integrate stored and streaming data
  - Hybrid stream and batch
- Data safety and availability
  - Fault tolerance

---

[1] Stonebraker, Michael, Ugur Cetintemel, and Stan Zdonik. "*The 8 requirements of real-time stream processing*." ACM Sigmod Record 34.4 (2005): 42-47

8 Requirements for Stream Processing[1]

- Keep the data moving
  - Prefer active (push) data sharing
- Declarative Access
  - E.g., streamSQL, CQL
- Handle imperfections
  - Late, missing, unordered items
- Predictable outcomes
  - Consistency, event-time processing
- Integrate stored and streaming data
  - Hybrid stream and batch
- Data safety and availability
  - Fault tolerance
- Partitioning and Scaling

---

[1] Stonebraker, Michael, Ugur Cetintemel, and Stan Zdonik. "*The 8 requirements of real-time stream processing*." ACM Sigmod Record 34.4 (2005): 42-47

8 Requirements for Stream Processing[1]

- Keep the data moving
  - Prefer active (push) data sharing
- Declarative Access
  - E.g., streamSQL, CQL
- Handle imperfections
  - Late, missing, unordered items
- Predictable outcomes
  - Consistency, event-time processing
- Integrate stored and streaming data
  - Hybrid stream and batch
- Data safety and availability
  - Fault tolerance
- Partitioning and Scaling
  - Distributed processing

---

[1] Stonebraker, Michael, Ugur Cetintemel, and Stan Zdonik. "*The 8 requirements of real-time stream processing*." ACM Sigmod Record 34.4 (2005): 42-47

8 Requirements for Stream Processing[1]

- Keep the data moving
  - Prefer active (push) data sharing
- Declarative Access
  - E.g., streamSQL, CQL
- Handle imperfections
  - Late, missing, unordered items
- Predictable outcomes
  - Consistency, event-time processing
- Integrate stored and streaming data
  - Hybrid stream and batch
- Data safety and availability
  - Fault tolerance
- Partitioning and Scaling
  - Distributed processing
- Instantaneous response

---

[1] Stonebraker, Michael, Ugur Cetintemel, and Stan Zdonik. "*The 8 requirements of real-time stream processing*." ACM Sigmod Record 34.4 (2005): 42-47

8 Requirements for Stream Processing[1]

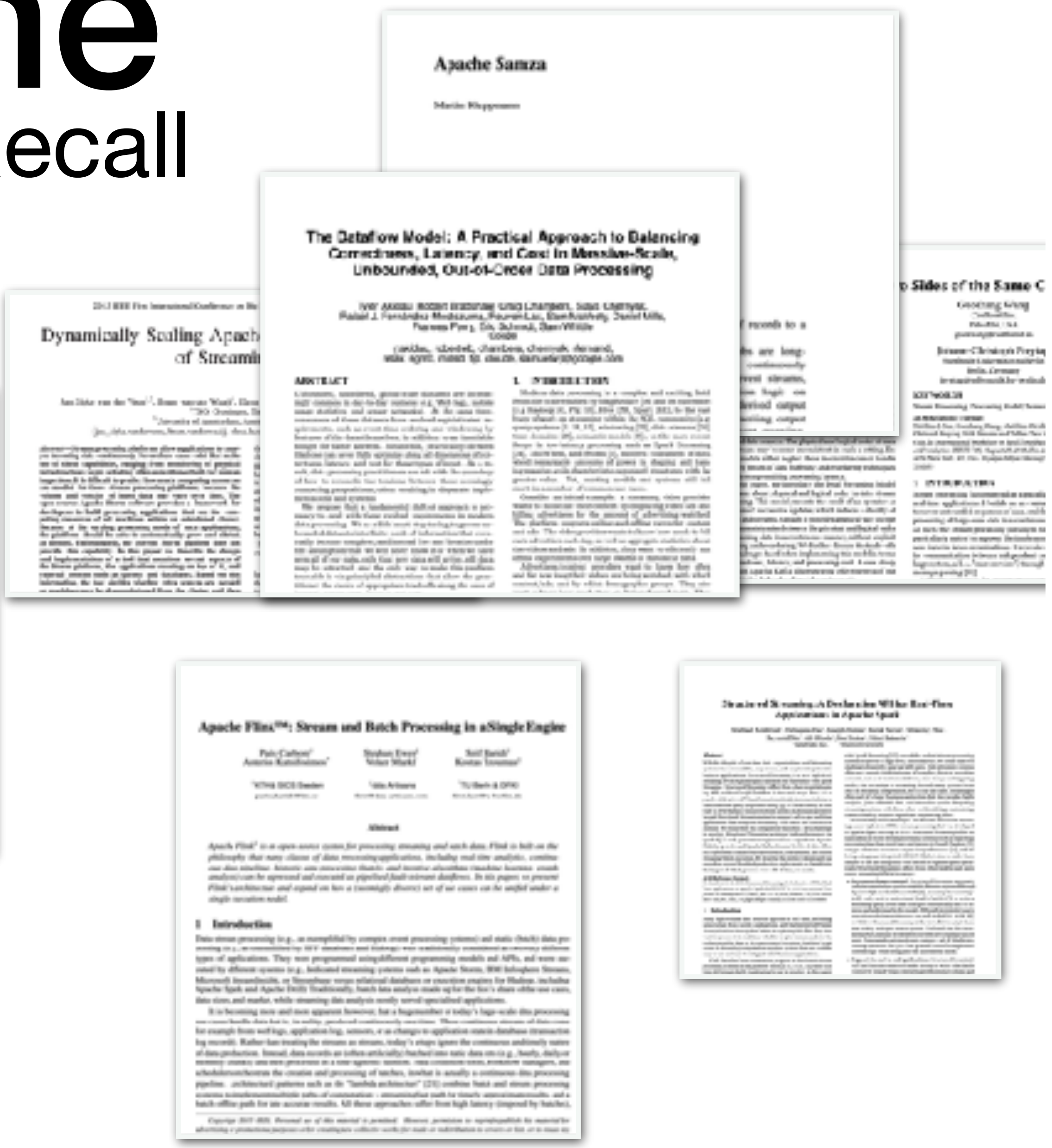
- Keep the data moving
  - Prefer active (push) data sharing
- Declarative Access
  - E.g., streamSQL, CQL
- Handle imperfections
  - Late, missing, unordered items
- Predictable outcomes
  - Consistency, event-time processing
- Integrate stored and streaming data
  - Hybrid stream and batch
- Data safety and availability
  - Fault tolerance
- Partitioning and Scaling
  - Distributed processing
- Instantaneous response
  - Low latency

---

[1] Stonebraker, Michael, Ugur Cetintemel, and Stan Zdonik. "*The 8 requirements of real-time stream processing*." ACM Sigmod Record 34.4 (2005): 42-47

Timeline

Precision not Recall



2002 2005 2006 2007 2008 2010 2014 2015 2018

Stream Processing State of the Art

Stream Analytics (SA) (Our focus)
- Obtain aggregates over streaming data within time boundaries

Event Processing (EP)

```
SELECT AVG(value) as avg
FROM TemperatureStream WINDOW 5 Min
GROUP BY room
HAVING avg > 40
```

```
Fire AS MATCH
HighTemperature SEQ SmokeDetection
WITHIN (5 Minutes)
```

Stream Processing State of the Art

Stream Analytics (SA) (Our focus)
- Obtain aggregates over streaming data within time boundaries

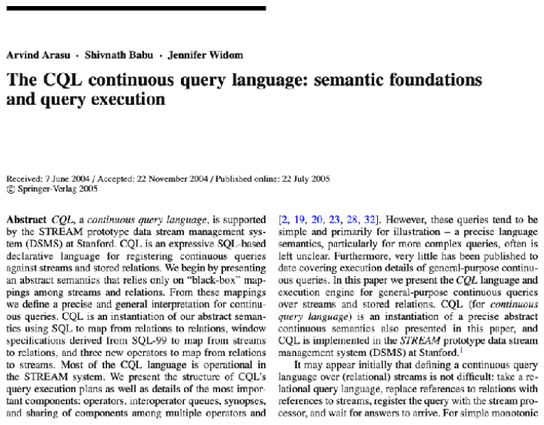
```
SELECT AVG(value) as avg
FROM TemperatureStream WINDOW 5 Min
GROUP BY room
HAVING avg > 40
```

Event Processing (EP)

- Interested in sequences of event, called composite events, defined using regular expressions

```
Fire AS MATCH
HighTemperature SEQ SmokeDetection
WITHIN (5 Minutes)
```

A (Query) Language Perspective



Arvind Arasu · Shivnath Babu · Jennifer Widom

The CQL continuous query language: semantic foundations and query execution

Received: 7 June 2004 / Accepted: 22 November 2004 / Published online: 22 July 2005

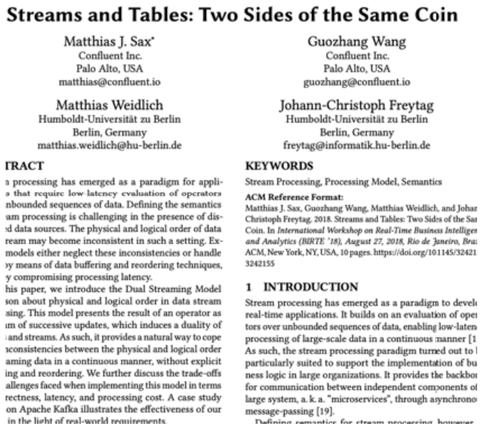
Abstract *CQL*, a *continuous query language*, is supported by the STREAM prototype data stream management system (DSMS) at Stanford. CQL is an expressive SQL-based declarative language for registering continuous queries against streams and stored relations. We begin by presenting an abstract semantics that relies only on "black-box" mappings among streams and relations. From these mappings we define a precise and general interpretation for continuous queries. CQL is an instantiation of our abstract semantics using SQL to map from relations to relations, window specifications derived from SQL-99 to map from streams to relations, and three new operators to map from relations to streams. Most of the CQL language is operational in the STREAM system. We present the structure of CQL's query execution plans as well as details of the most important components: operators, interoperator queues, synopses, and sharing of components among multiple operators and [2, 19, 20, 23, 28, 32]. However, these queries tend to be simple and primarily for illustration – a precise language semantics, particularly for more complex queries, often is left unclear. Furthermore, very little has been published to date covering execution details of general-purpose continuous queries. In this paper we present the *CQL* language and execution engine for general-purpose continuous queries over streams and stored relations. CQL (for *continuous query language*) is an instantiation of a precise abstract continuous semantics also presented in this paper, and CQL is implemented in the *STREAM* prototype data stream management system (DSMS) at Stanford.

It may appear initially that defining a continuous query language over (relational) streams is not difficult: take a relational query language, replace references to relations with references to streams, register the query with the stream processor, and wait for answers to arrive. For simple monotonic

It addresses the problem of manipulating and managing data-streams**.** It stresses on the **operations** that are **necessary** to build **applications**. Some **Assumptions** are made on the underlying **system**(s)**.**

A ~~new~~ hybrid ~~hope~~ perspective

It addresses the problems in between. How do we map the abstraction of an high-level language to the underlying system?



Streams and Tables: Two Sides of the Same Coin

Matthias J. Sax*
Confluent Inc.
Palo Alto, USA
matthias@confluent.io

Guozhang Wang
Confluent Inc.
Palo Alto, USA
guozhang@confluent.io

Matthias Weidlich
Humboldt-Universität zu Berlin
Berlin, Germany
matthias.weidlich@hu-berlin.de

Johann-Christoph Freytag
Humboldt-Universität zu Berlin
Berlin, Germany
freytag@informatik.hu-berlin.de

TRACT

s processing has emerged as a paradigm for appli- s that require low latency evaluation of operators nbounded sequences of data. Defining the semantics am processing is challenging in the presence of dis- d data sources. The physical and logical order of data ream may become inconsistent in such a setting. Ex- models either neglect these inconsistencies or handle y means of data buffering and reordering techniques, y compromising processing latency.

his paper, we introduce the Dual Streaming Model son about physical and logical order in data stream sing. This model presents the result of an operator as m of successive updates, which induces a duality of and streams. As such, it provides a natural way to cope nconsistencies between the physical and logical order aming data in a continuous manner, without explicit ing and reordering. We further discuss the trade-offs allenges faced when implementing this model in terms rectness, latency, and processing cost. A case study on Apache Kafka illustrates the effectiveness of our in the light of real-world requirements.

KEYWORDS

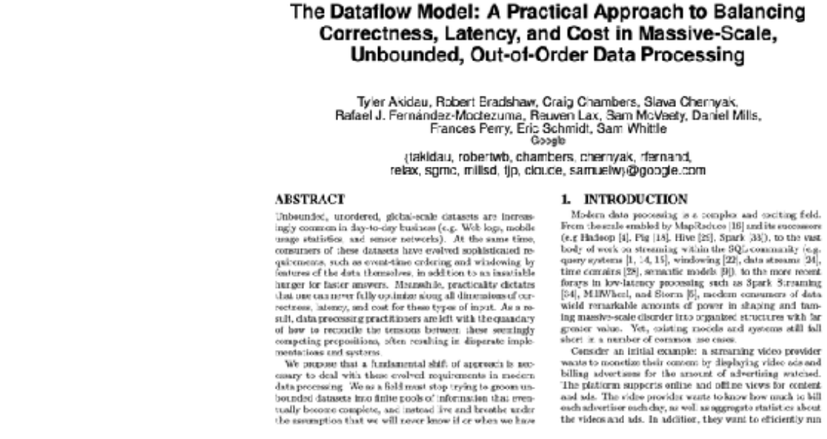
Stream Processing, Processing Model, Semantics

ACM Reference Format:
Matthias J. Sax, Guozhang Wang, Matthias Weidlich, and Johan Christoph Freytag. 2018. Streams and Tables: Two Sides of the Sa Coin. In *International Workshop on Real-Time Business Intelliger and Analytics (BIRTE '18), August 27, 2018, Rio de Janeiro, Bra* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/32421 3242155

1 INTRODUCTION

Stream processing has emerged as a paradigm to develo real-time applications. It builds on an evaluation of oper tors over unbounded sequences of data, enabling low-laten processing of large-scale data in a continuous manner [1 As such, the stream processing paradigm turned out to particularly suited to support the implementation of bu ness logic in large organizations. It provides the backbo for communication between independent components o large system, a. k. a. "microservices", through asynchrono message-passing [19].

A System Perspective



The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing

Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, Sam Whittle
Google
{takidau, robertwb, chambers, chernyak, rfernand, relax, sgmc, millsd, fjp, clo cle, samuelw}@google.com

ABSTRACT

Unbounded, unordered, global-scale datasets are increasingly common in day-to-day business (e.g. Web logs, mobile usage statistics, and sensor networks). At the same time, consumers of these datasets have evolved sophisticated requirements, such as event-time ordering and windowing by features of the data themselves, in addition to an insatiable hunger for faster answers. Meanwhile, practicality dictates that one can never fully optimize along all dimensions of correctness, latency, and cost for these types of input. As a result, data processing practitioners are left with the quandary of how to reconcile the tensions between these seemingly competing propositions, often resulting in disparate implementations and systems.

We propose that a fundamental shift of approach is necessary to deal with these evolved requirements in modern data processing. We as a field must stop trying to groom unbounded datasets into finite pools of information that eventually become complete, and instead live and breathe under the assumption that we will never know if or when we have

1. INTRODUCTION

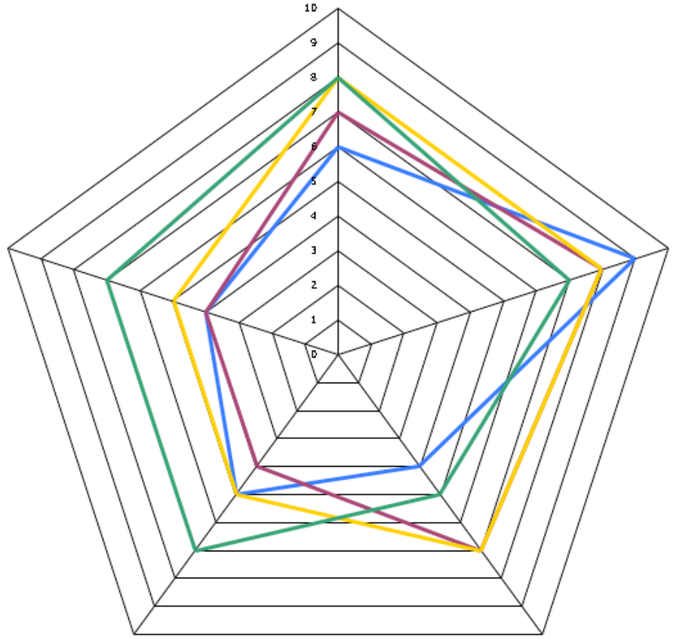
Modern data processing is a complex and exciting field. From the roots enabled by MapReduce [16] and its successors (e.g. Hadoop [4], Pig [18], Hive [25], Spark [33]), to the vast body of work on streaming within the DBMS community (e.g. query systems [1, 7, 15], windowing [22], data streams [24], time semantics [28], semantic models [9]), to the more recent forays in low-latency processing such as Spark Streaming [34], MillWheel, and Storm [6], modern consumers of data yield remarkable amounts of power in shaping and taming massive-scale disorder into organized structures with far greater value. Yet, existing models and systems still fall short in a number of common use cases.

Consider an initial example: a streaming video provider wants to monetize their content by displaying video ads and billing advertisers for the amount of advertising watched. The platform supports online and offline views for content and ads. The video provider wants to know how much to bill each advertiser each day, as well as aggregate statistics about the videos and ads. In addition, they want to efficiently run

It address the problem of dealing with **unbounded data**. Discuss the systems' **primitives** that are necessary to guarantee **low-latency** and **fault-tolerance in** presence of *uncertainty*, e.g., late arrivals.

Dimensions to Consider

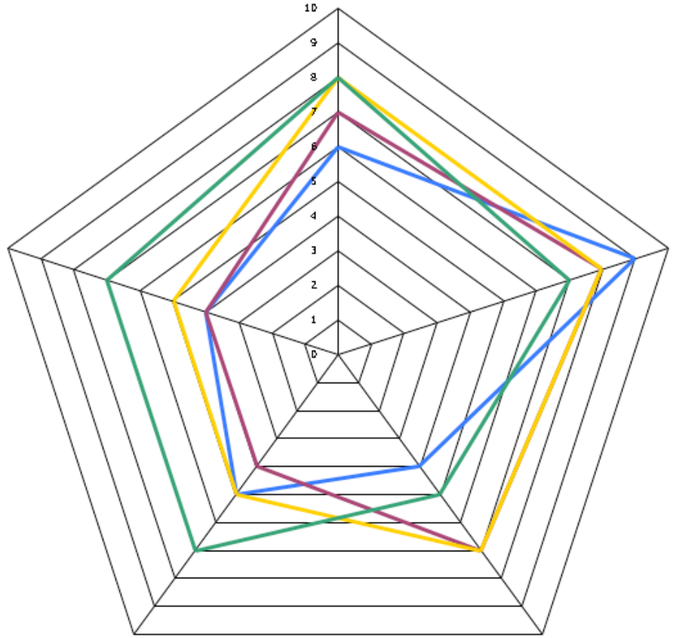
Five dimensions are important when considering stream processing engines for big data.



Dimensions to Consider

Five dimensions are important when considering stream processing engines for big data.

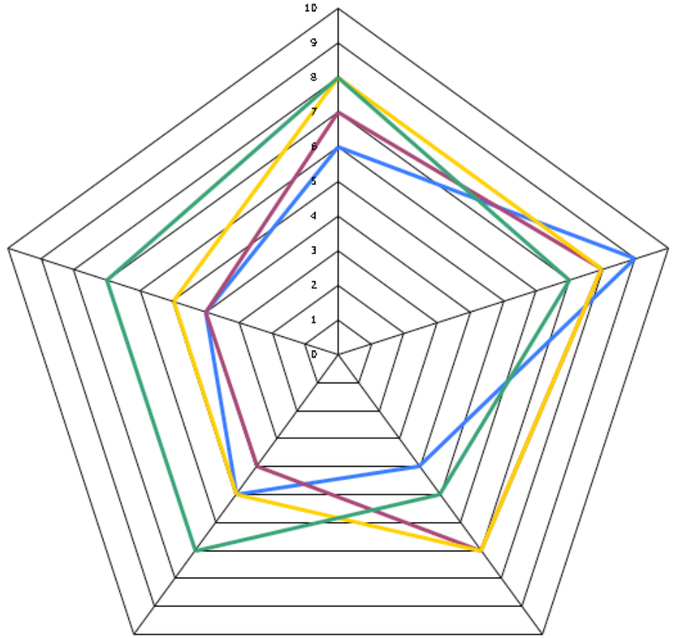
- Notions of Time



Dimensions to Consider

Five dimensions are important when considering stream processing engines for big data.

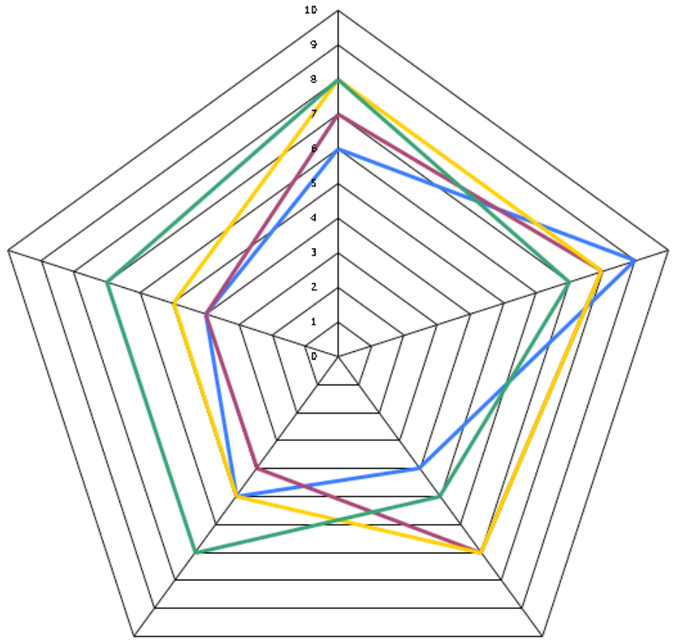
- Notions of Time
- Continuous Processing



Dimensions to Consider

Five dimensions are important when considering stream processing engines for big data.

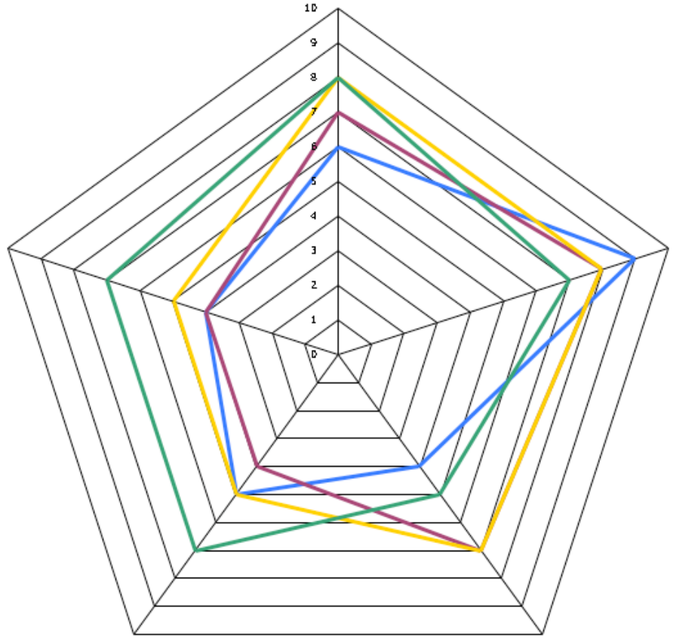
- Notions of Time
- Continuous Processing
- Architectural View



Dimensions to Consider

Five dimensions are important when considering stream processing engines for big data.

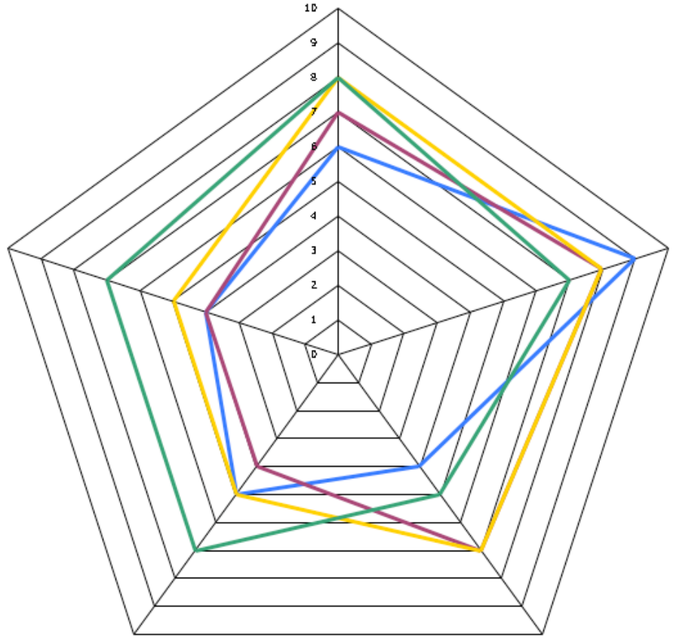
- Notions of Time
- Continuous Processing
- Architectural View
- Fault Tolerance



Dimensions to Consider

Five dimensions are important when considering stream processing engines for big data.

- Notions of Time
- Continuous Processing
- Architectural View
- Fault Tolerance
- Programming Model



Different Notions of Time (1/2)

In the SP literature, many notion of time have been proposed. For sake of simplicity, we use the nomenclature suggested by Tyler Akidau [2].

- **Ingestion time**: the time at which a stream element arrives in the system
- **Processing time**: the time at which an operator in the processing pipeline reads the stream element
- **Event time**: The original time at which the data was generated

NB: in the following we ignore ingestion time without loss of generality.

---

[2] Akidau, Tyler, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing.(2015).

Different Notions of Time (2/2)

For both ingestion and processing time, the progress of time is controlled by stream processing engine and the data ordering is strictly monotonic.

On the other hand, event time is controlled by the source. Thus, the data ordering is only monotonic. Indeed, in event-time, it is possible to have late arriving elements and element occurring simultaneously.

Depending on the type of processing one needs to do, late arrivals may be taken into account.

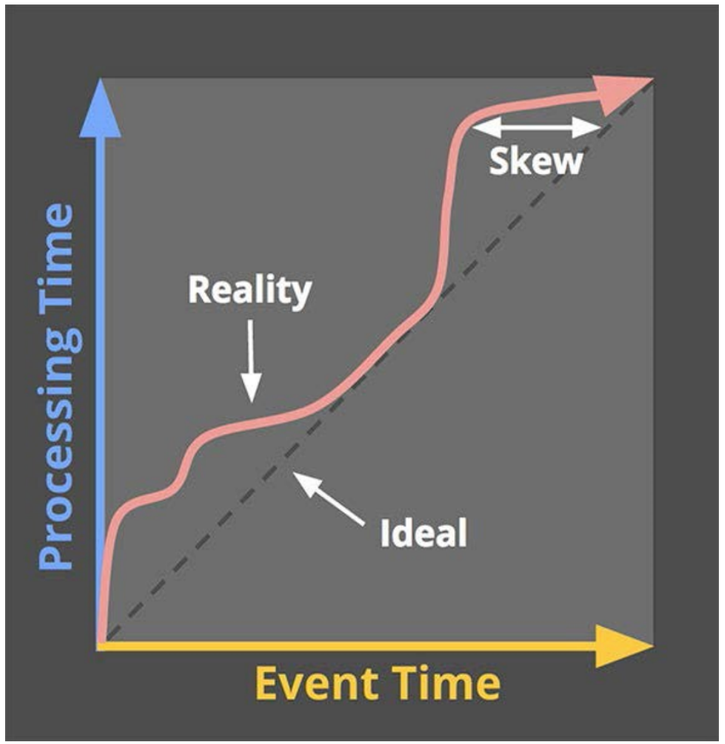
Event Time vs Processing Time (1/2)

Ideally, one would like the data to reach the system when they are generated in the real world.

However, this is *physically* not possible, due to the network delay.

Moreover, in a distributed system, where the components are connected to a *non-reliable* network, events can arrive late, i.e., out of order.

Event Time vs Processing Time (2/2)[2]



[2] Akidau, Tyler, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing.(2015).

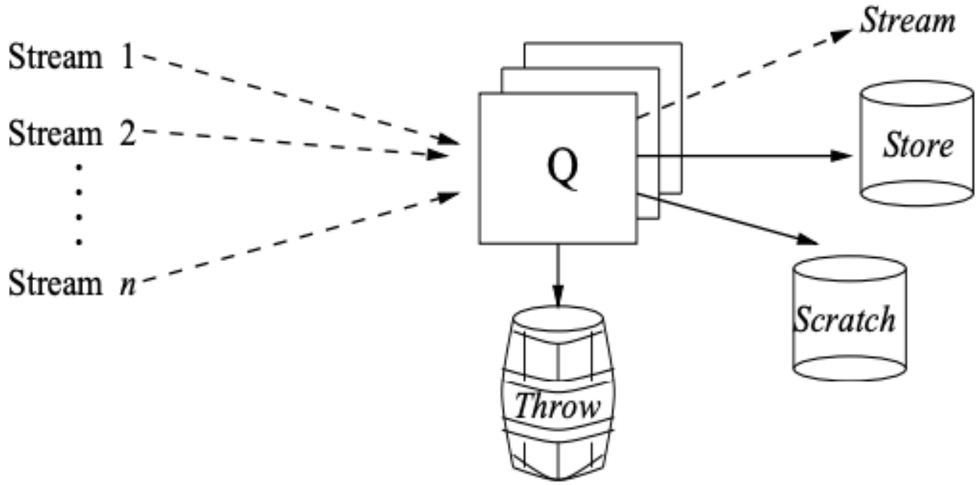
Continuous Processing (1/2)

The infinite nature of streams requires a change of paradigm in the way we process data.

Continuous semantics: the results of a continuous query is the set of data that would be returned if the query were executed at every instant in time[3].

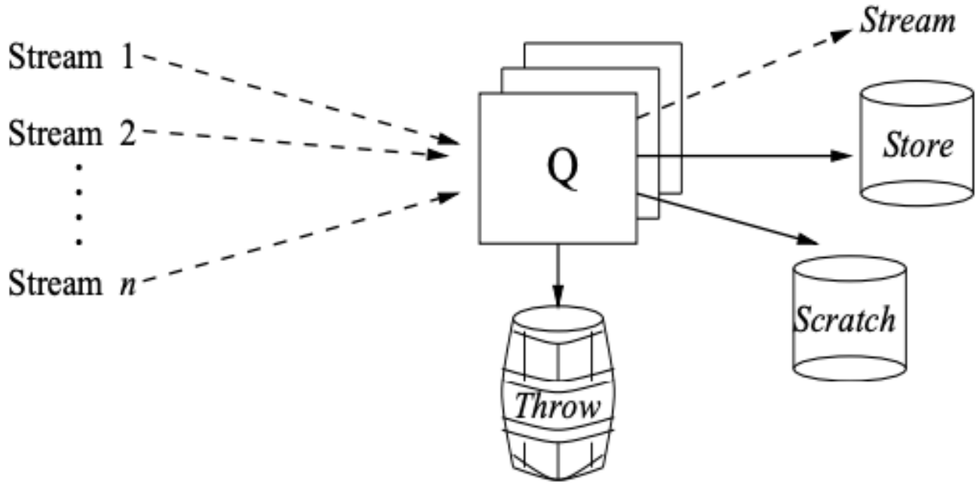
[3] Terry, Douglas, et al. "Continuous queries over append-only databases." Acm Sigmod Record 21.2 (1992): 321-330.

Continuous Processing (2/2)



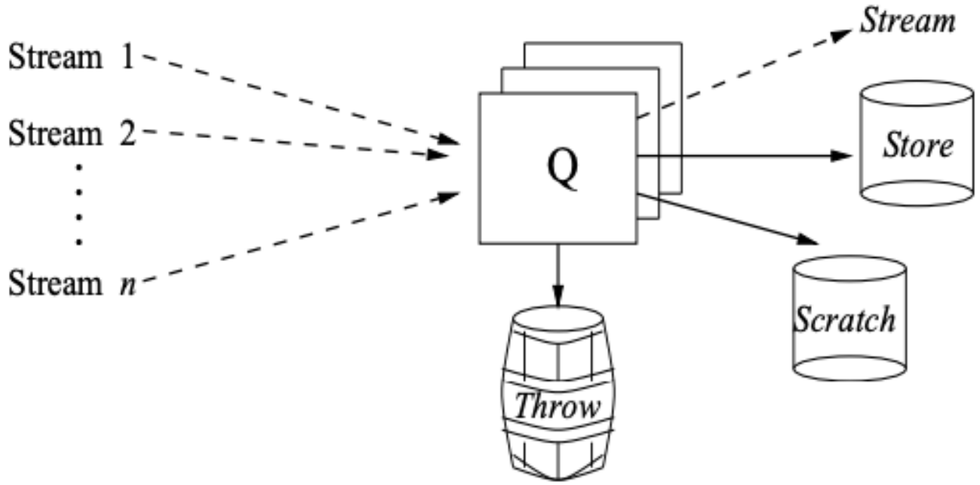
Continuous Processing (2/2)

- The Stream is filled with the elements of the answer that are produced and never changed;



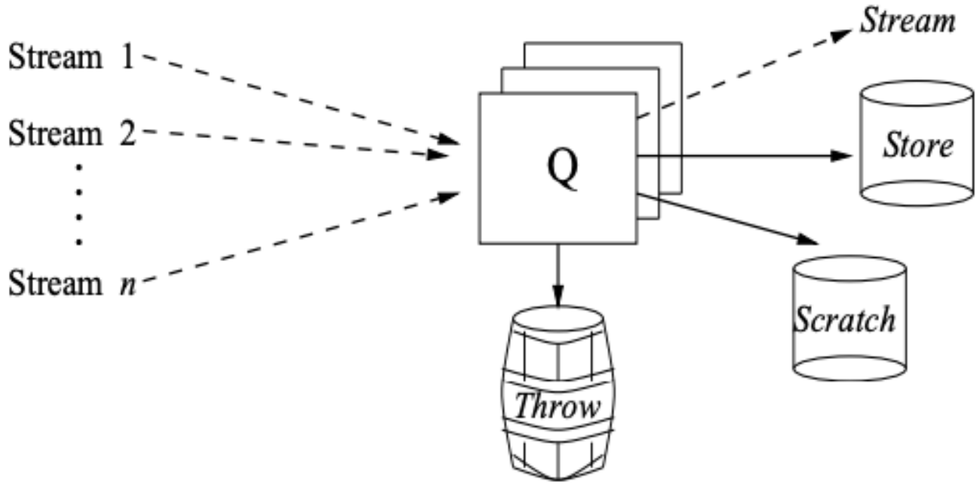
Continuous Processing (2/2)

- The Stream is filled with the elements of the answer that are produced and never changed;
- The Store is filled with parts of the answer that may change in the future;



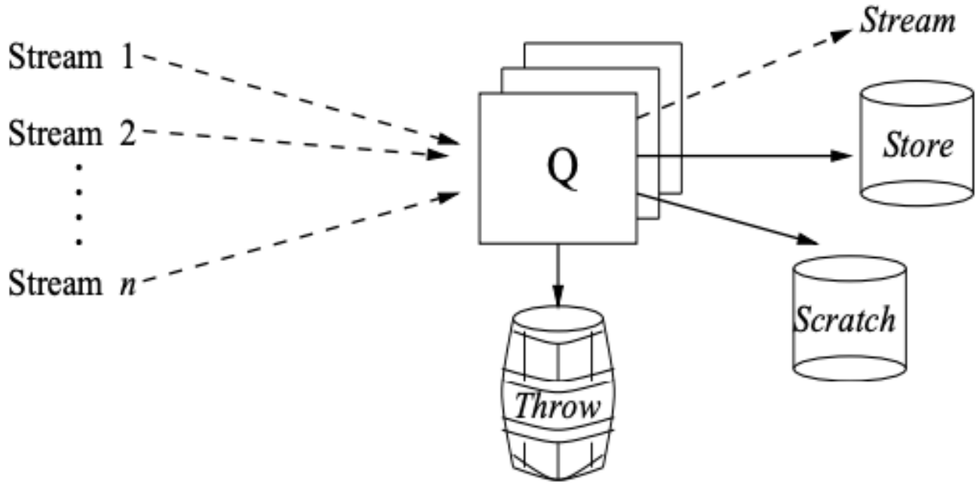
Continuous Processing (2/2)

- The Stream is filled with the elements of the answer that are produced and never changed;
- The Store is filled with parts of the answer that may change in the future;
- The Scratch is used to store data that are not part of the answer but are used to compute it;

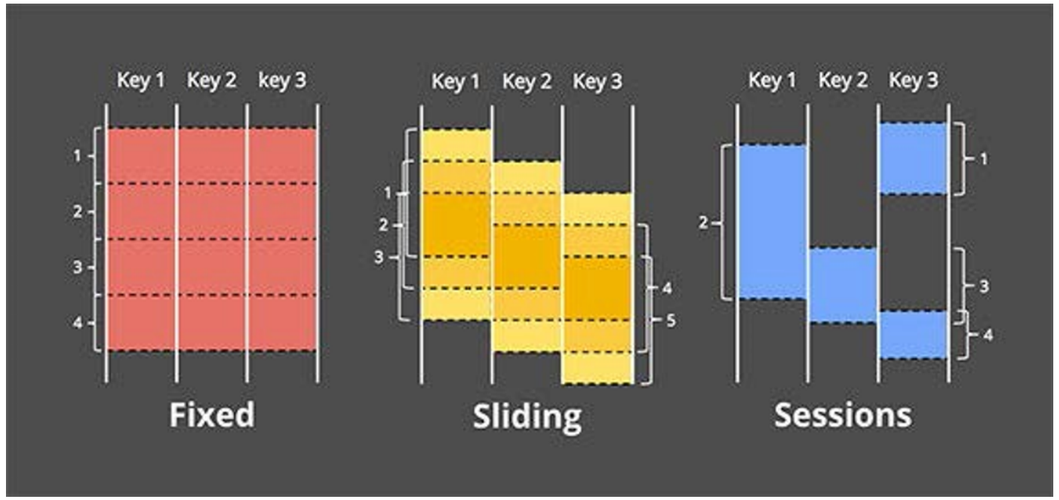


Continuous Processing (2/2)

- The Stream is filled with the elements of the answer that are produced and never changed;
- The Store is filled with parts of the answer that may change in the future;
- The Scratch is used to store data that are not part of the answer but are used to compute it;
- The Throw is used to store unneeded tuples.



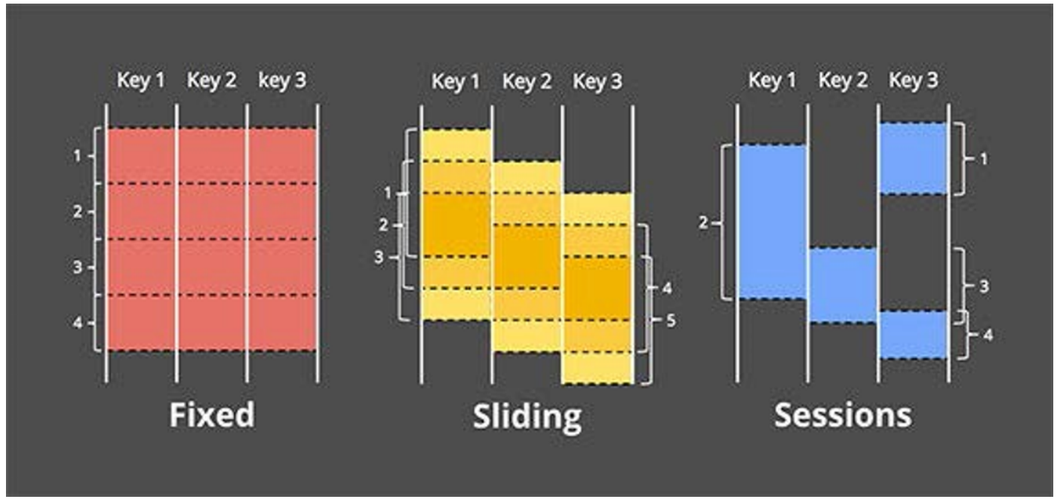
Dealing With Unboundedness: Window Operators[2]



[2] Akidau, Tyler, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing.(2015).

Dealing With Unboundedness: Window Operators[2]

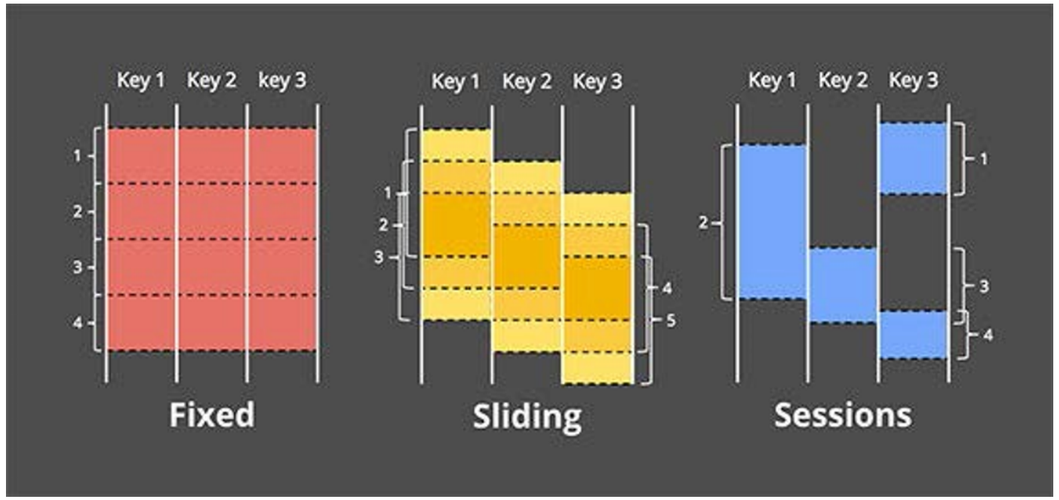
- Time windows



[2] Akidau, Tyler, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing.(2015).

Dealing With Unboundedness: Window Operators[2]

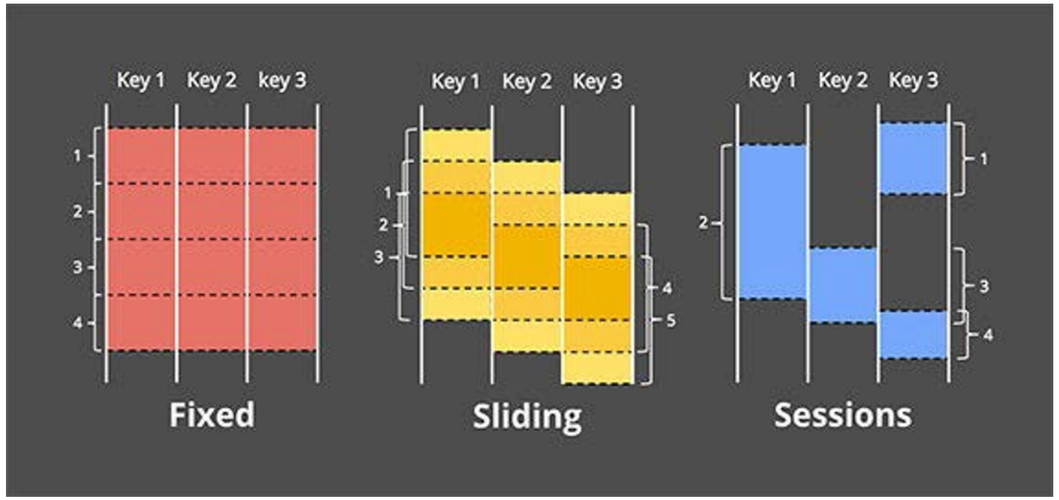
- Time windows
  - Sliding



[2] Akidau, Tyler, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing.(2015).

Dealing With Unboundedness: Window Operators[2]

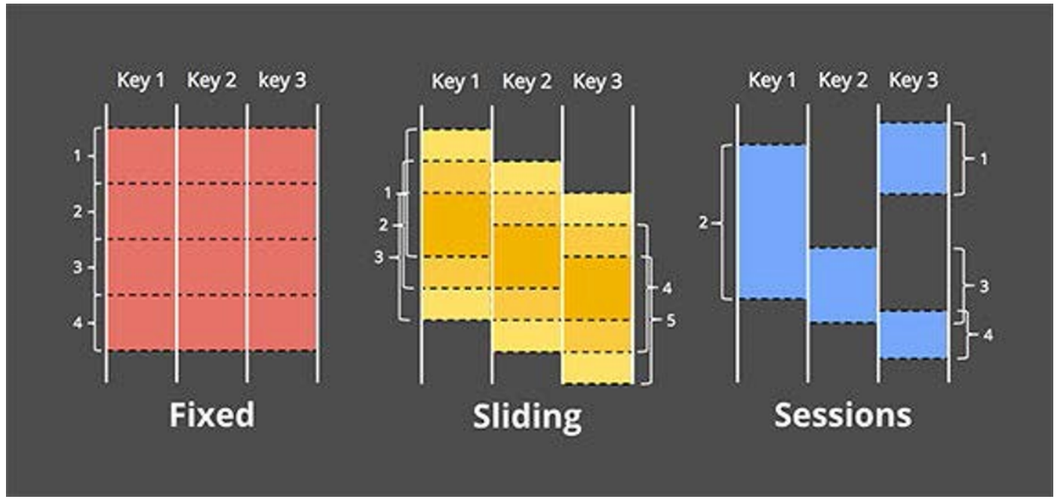
- Time windows
  - Sliding
  - Tumbling



[2] Akidau, Tyler, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing.(2015).

Dealing With Unboundedness: Window Operators[2]

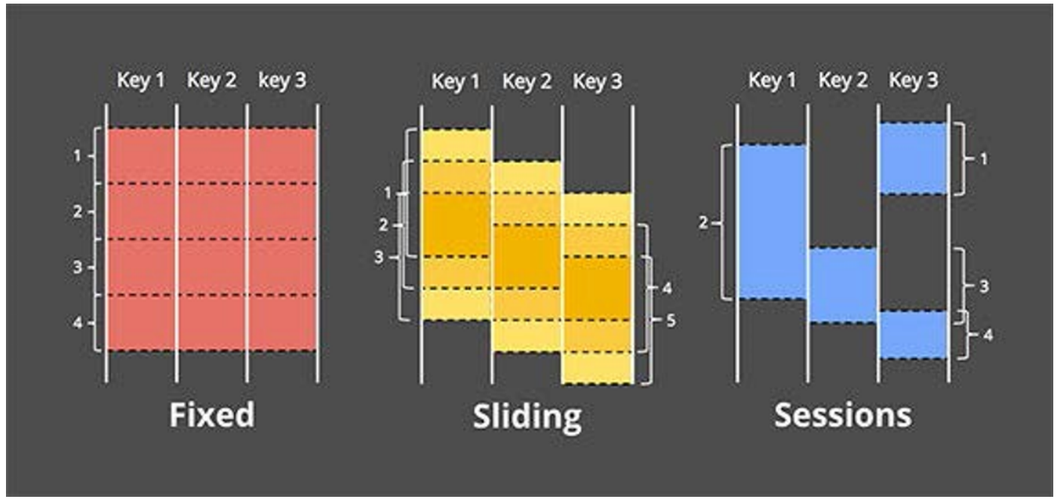
- Time windows
  - Sliding
  - Tumbling
- Tuple windows



[2] Akidau, Tyler, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing.(2015).

Dealing With Unboundedness: Window Operators[2]

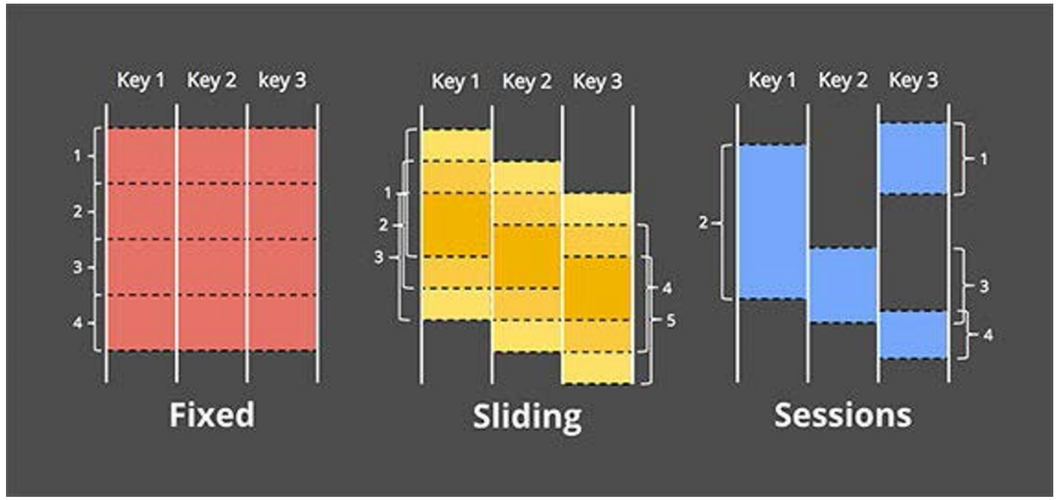
- Time windows
  - Sliding
  - Tumbling
- Tuple windows
  - Also called physical windows



[2] Akidau, Tyler, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing.(2015).

Dealing With Unboundedness: Window Operators[2]

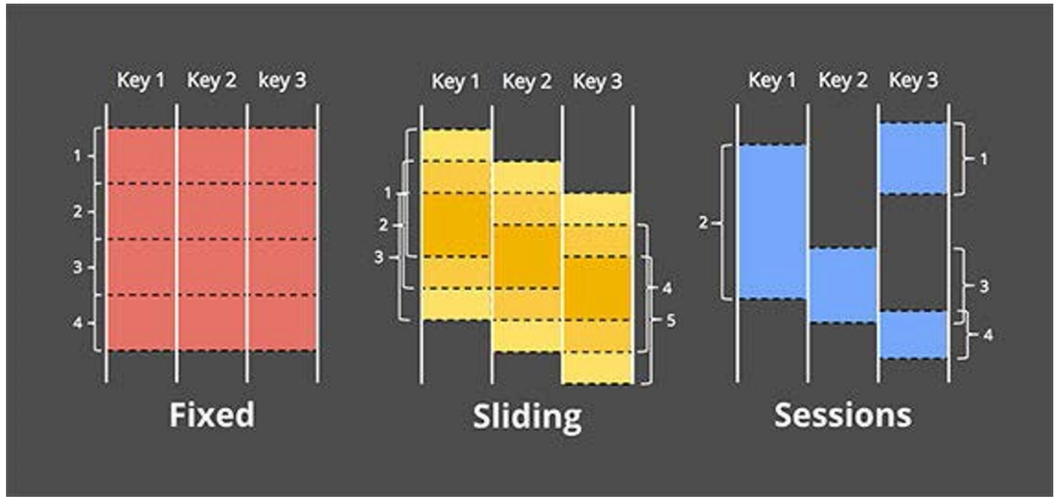
- Time windows
  - Sliding
  - Tumbling
- Tuple windows
  - Also called physical windows
- Data-driven windows



[2] Akidau, Tyler, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing.(2015).

Dealing With Unboundedness: Window Operators[2]

- Time windows
  - Sliding
  - Tumbling
- Tuple windows
  - Also called physical windows
- Data-driven windows
  - Session windows



[2] Akidau, Tyler, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing.(2015).

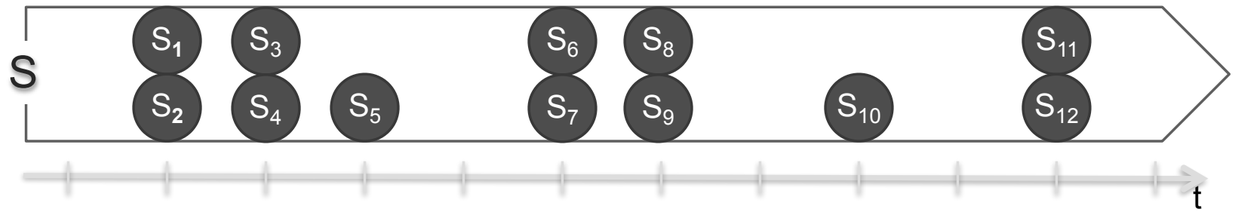
Time-Based Window Operator (1/2)

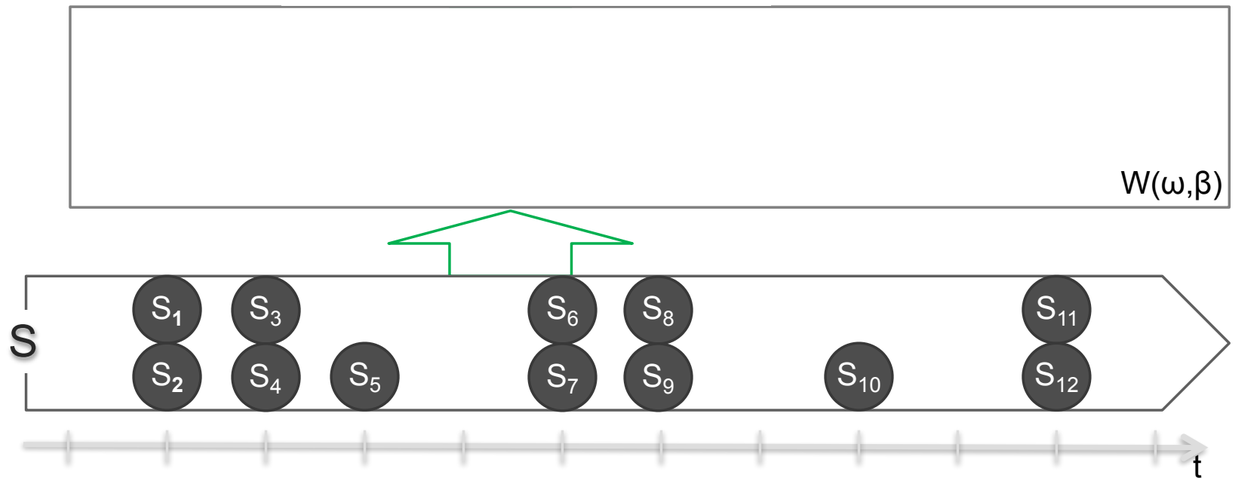
A time-based sliding window operator consists of two parameter $W = P(\omega, \beta)$ where:
- ω represents the window width;
- β is called sliding parameter.

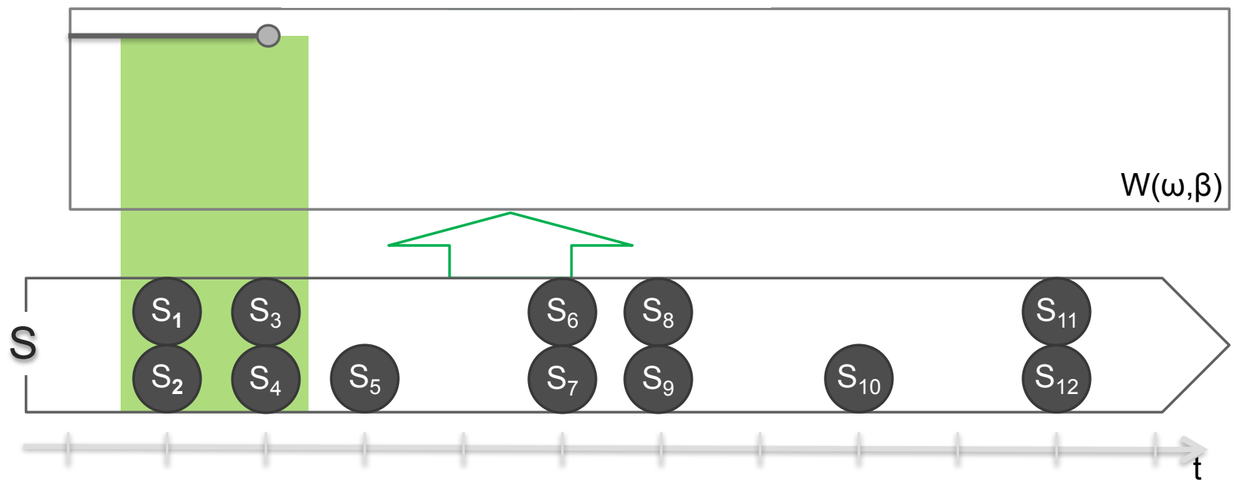
For each point in time, the time-based sliding window operator defines a set of windows.

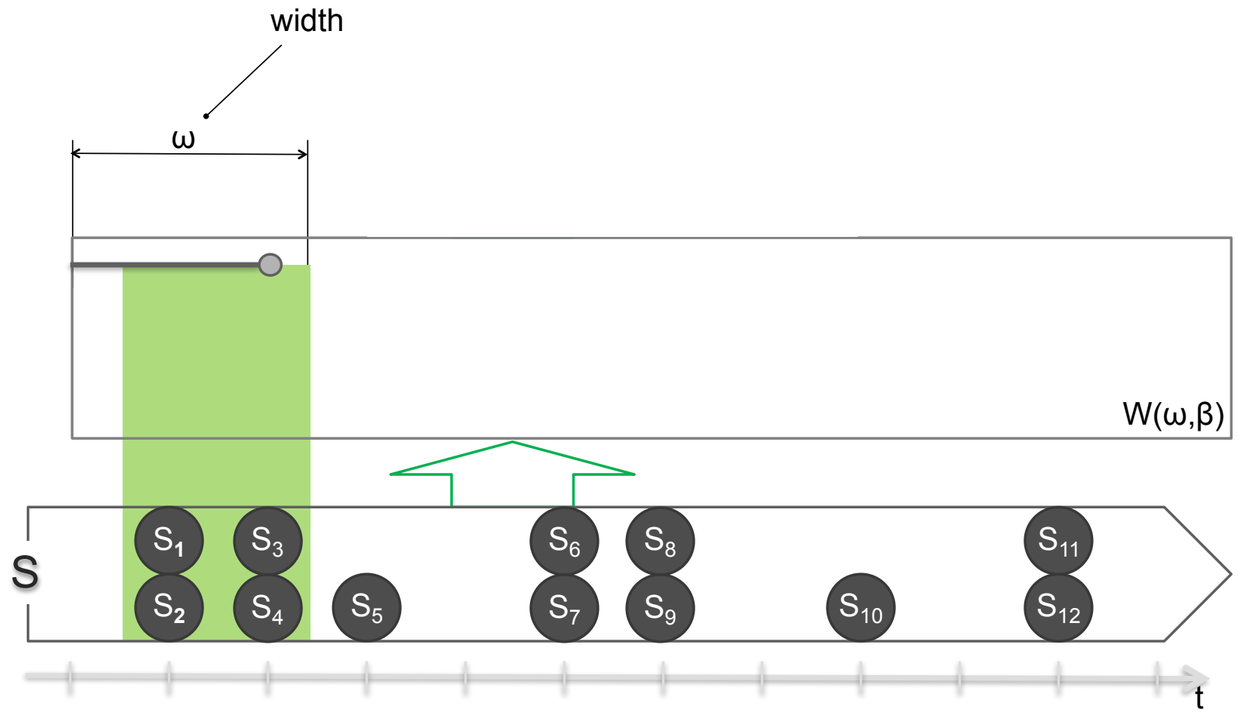
$$\mathcal{W} = (< o, c > | c > o, |c - o| = \omega, |o - o'| = |c - c'| = beta)$$

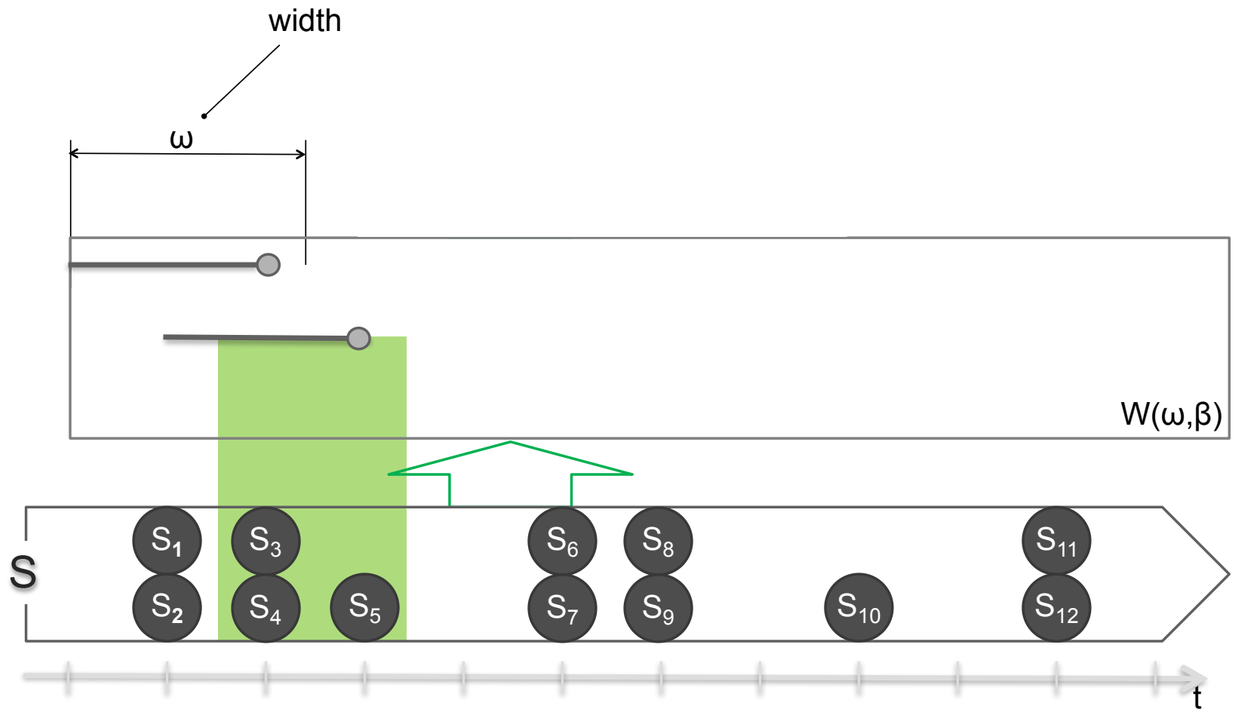
Time-Based HOPPING Window Operator

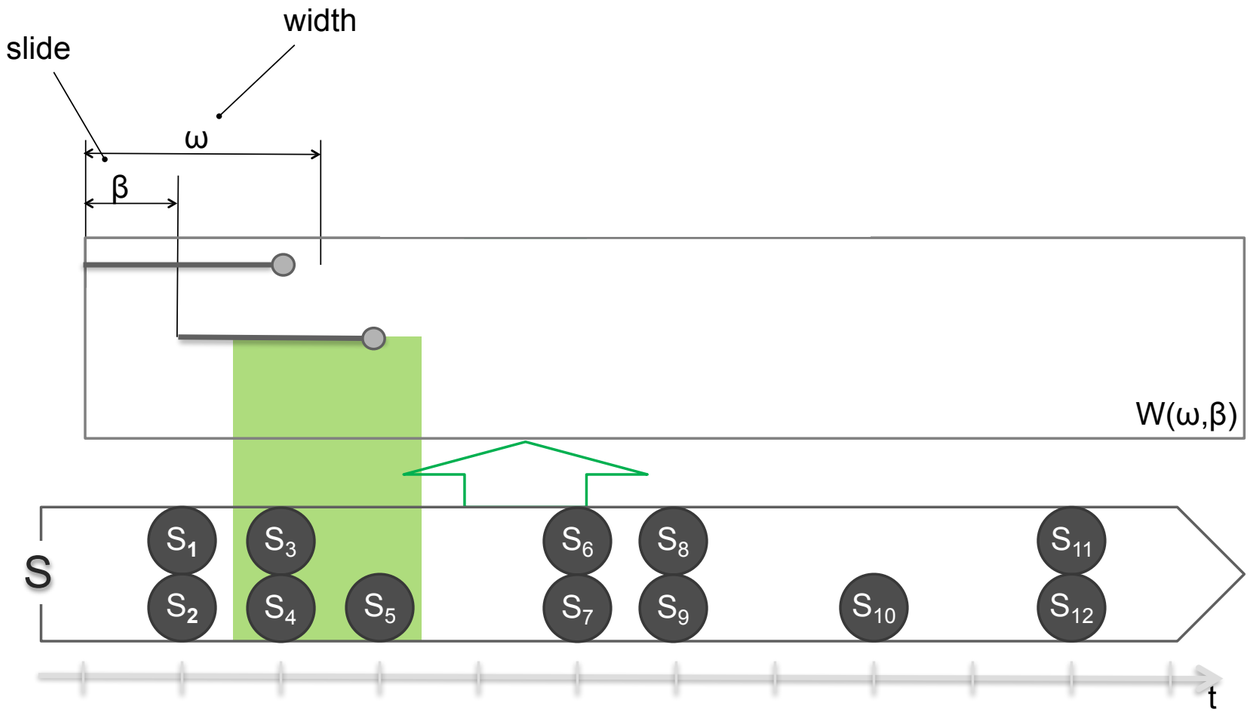

S
S1
S2
S3
S4
S5
S6
S7
S8
S9
S10
S11
S12
t

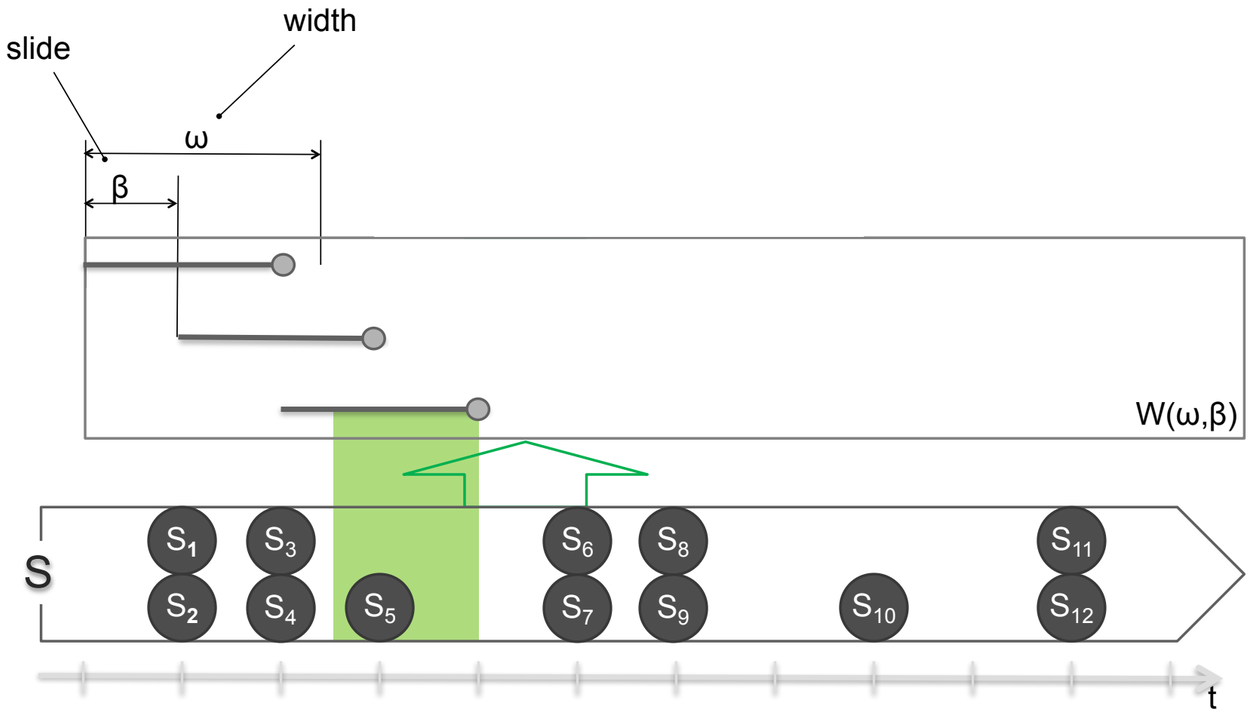

W(ω,β)
S
S1
S2
S3
S4
S5
S6
S7
S8
S9
S10
S11
S12
t


W(ω,β)
S
S1
S2
S3
S4
S5
S6
S7
S8
S9
S10
S11
S12
t


width
ω
W(ω,β)
S
S1
S3
S6
S8
S11
S2
S4
S5
S7
S9
S10
S12
t


width
ω
W(ω,β)
S
S1
S2
S3
S4
S5
S6
S7
S8
S9
S10
S11
S12
t


slide
width
ω
β
W(ω,β)
S
S1
S2
S3
S4
S5
S6
S7
S8
S9
S10
S11
S12
t


slide
width
ω
β
W(ω,β)
S
S1
S3
S6
S8
S11
S2
S4
S5
S7
S9
S10
S12
t

Windowing in Processing Time

Windowing in Processing Time

- elements' timestamps are controlled by the system

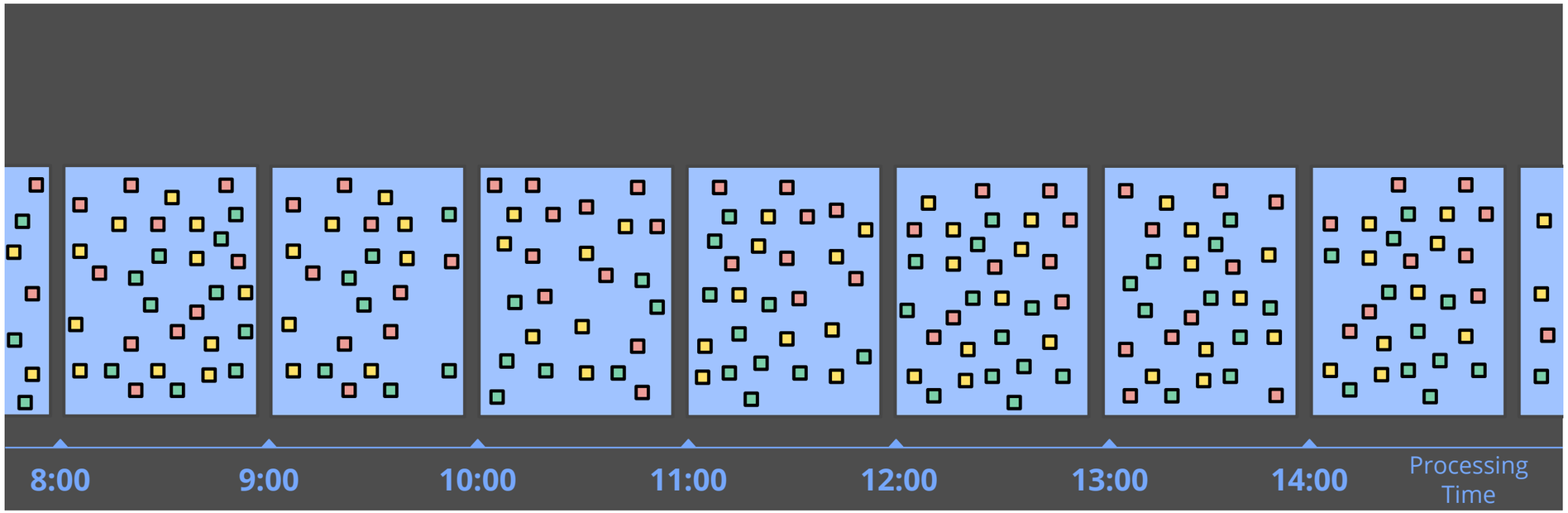
Windowing in Processing Time

- elements' timestamps are controlled by the system
- time progresses according to system's internal clock

Windowing in Processing Time

- elements' timestamps are controlled by the system
- time progresses according to system's internal clock
- no chance for out of order or late arrival

Windowing in Processing Time[2]



source

[2] Akidau, Tyler, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing.(2015).

Windowing in Event Time

Windowing in Event Time

- Windows are based on timestamp info in the stream

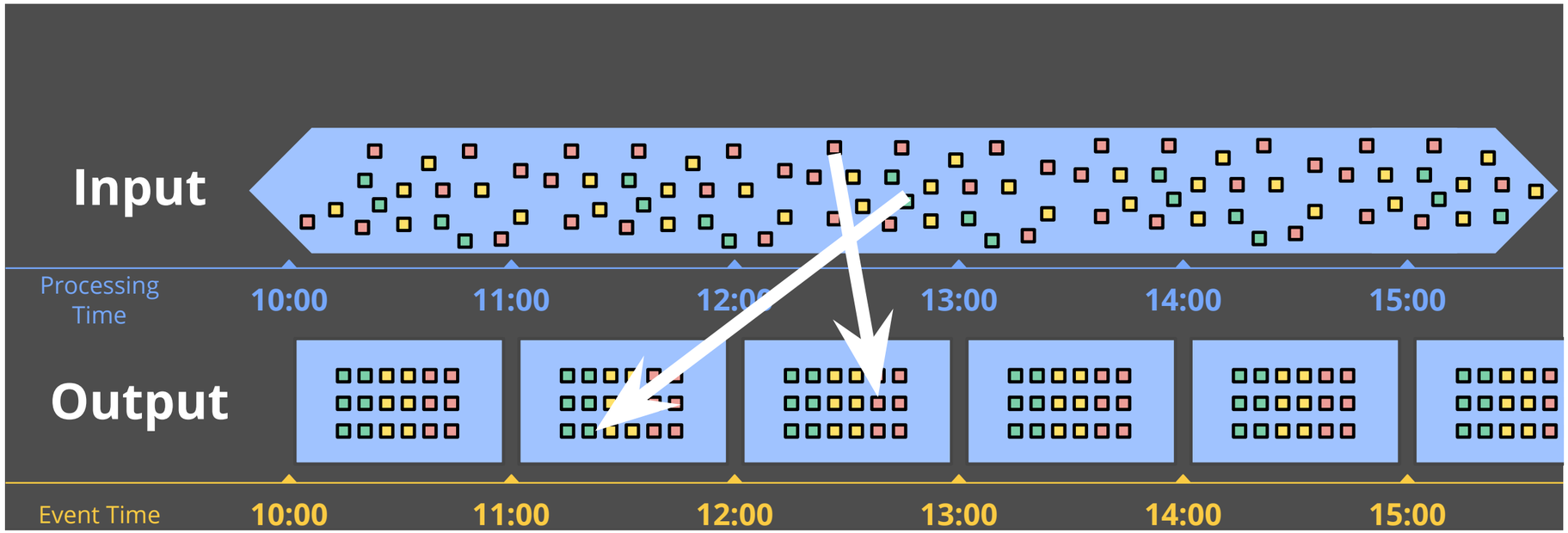
Windowing in Event Time

- Windows are based on timestamp info in the stream
- Buffering is needed to deal with late arrivals

Windowing in Event Time

- Windows are based on timestamp info in the stream
- Buffering is needed to deal with late arrivals
- Needs an external time progress indicator

Windowing in Event Time[2]



source

[2] Akidau, Tyler, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing.(2015).

Data Driven Windowing[2]

[2] Akidau, Tyler, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing.(2015).

Data Driven Windowing[2]

- Windows are based on data semantics

[2] Akidau, Tyler, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing.(2015).

Data Driven Windowing[2]

- Windows are based on data semantics
- Buffering is needed to deal with late arrivals

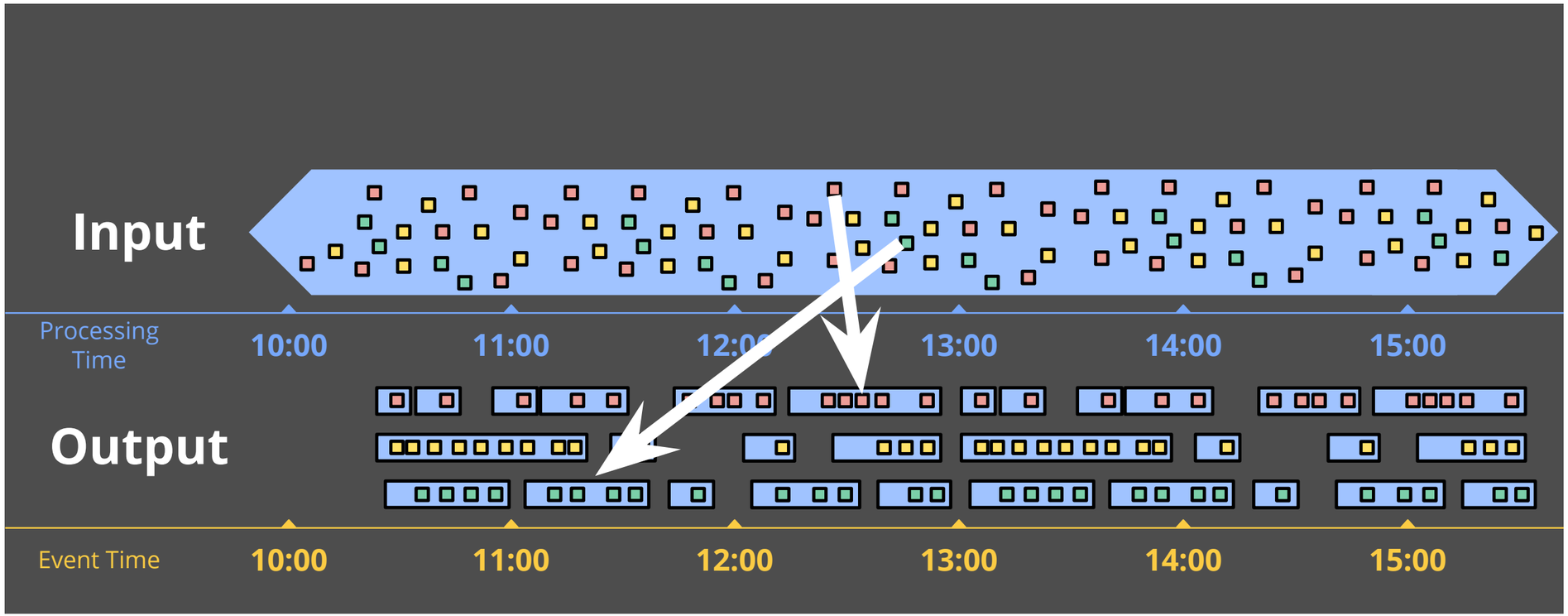
[2] Akidau, Tyler, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing.(2015).

Data Driven Windowing[2]

- Windows are based on data semantics
- Buffering is needed to deal with late arrivals
- Needs an external time progress indicator

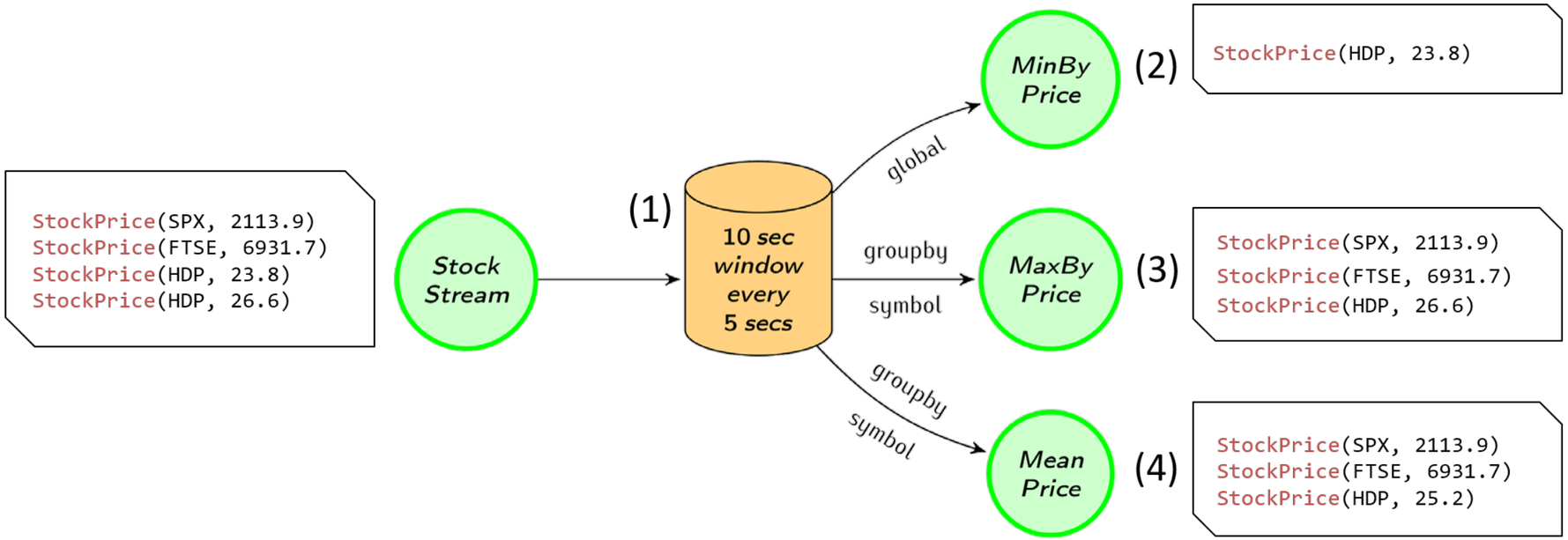
[2] Akidau, Tyler, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing.(2015).

Data Driven Windowing[2]



[2] Akidau, Tyler, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing.(2015).

Example Windowed Aggregation

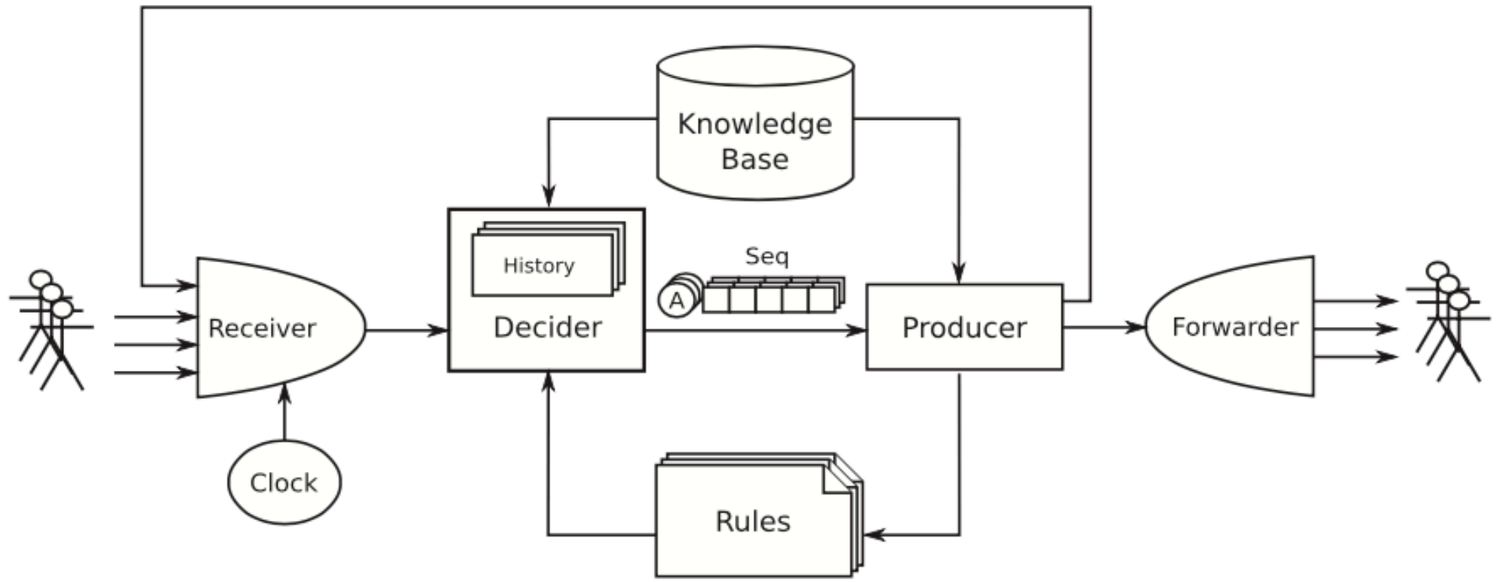



```
(1)val windowedStream = stockStream.window(Time.of(10, SECONDS)).every(Time.of(5, SECONDS))
(2)val lowest = windowedStream.minBy("price")
(3)val maxByStock = windowedStream.groupBy("symbol").maxBy("price")
(4)val rollingMean = windowedStream.groupBy("symbol").mapWindow(mean _)
```

source

Architectural Approaches to Stream Processing (1/2)

The architecture of Information Flow Processing (IFP) system defined by Cugola and Margara in[4].



[4] Cugola, Gianpaolo, and Alessandro Margara. "Processing flows of information: From data stream to complex event processing." ACM Computing Surveys (2012).

Architectural Approaches to Stream Processing (2/2)

Two SP architectures had big industrial tractions:

Architectural Approaches to Stream Processing (2/2)

Two SP architectures had big industrial tractions:

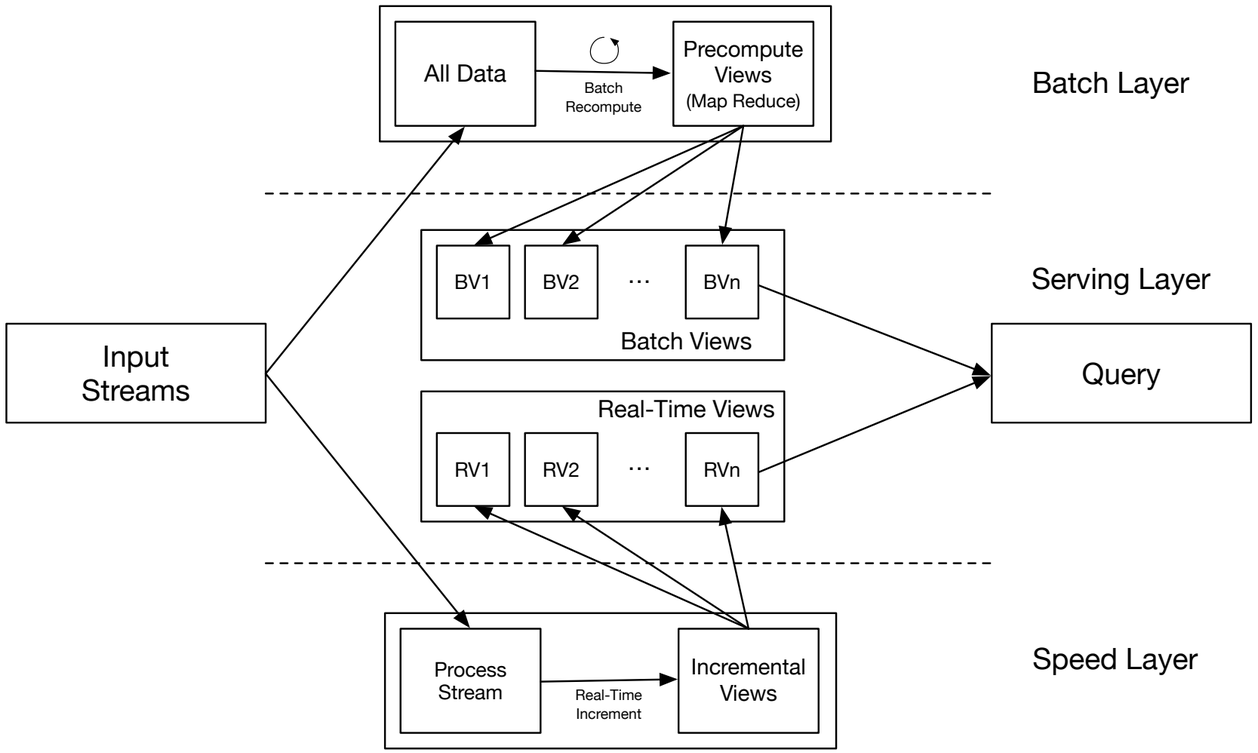
- The Lambda Architecture, which combines streaming and batch processing.

Architectural Approaches to Stream Processing (2/2)

Two SP architectures had big industrial tractions:

- The Lambda Architecture, which combines streaming and batch processing.
- The Kappa Architecture, which relies on fault-tolerant stream log.

Lambda Architecture[17]



The lambda architecture is used when approximate results are needed quickly and more accurate results can come later.

Moreover, it is suitable for cases where pure stream processing is not fault tolerant and more accurate results require to wait for late arrivals.

[17] Courtesy of Emanuele Della Valle/Marco Balduini

Kappa Architecture[17]

The Kappa architecture was designed to address the limitation of the lambda architecture.

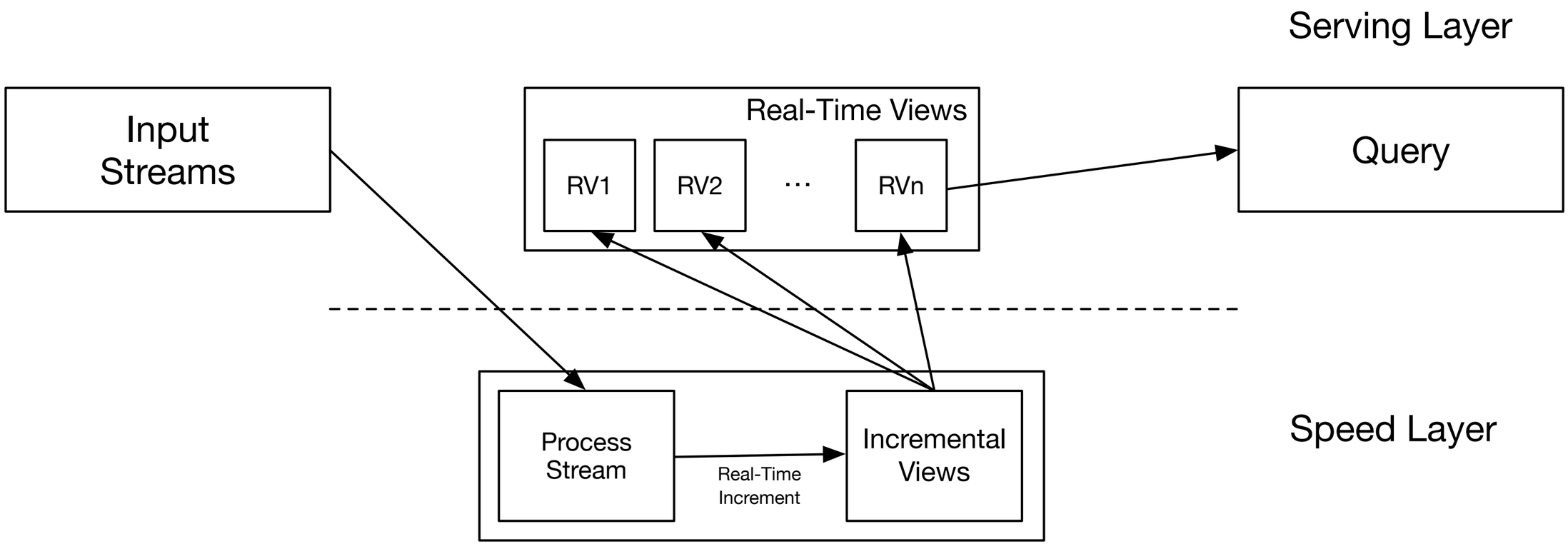
It leverages only a speed layer but it relies on a fault-tolerant stream storage, e.g., a distributed log.

The Kappa architecture is simpler to maintain and less costly to operate than the lambda architecture.

---

[17] Courtesy of Emanuele Della Valle/Marco Balduini

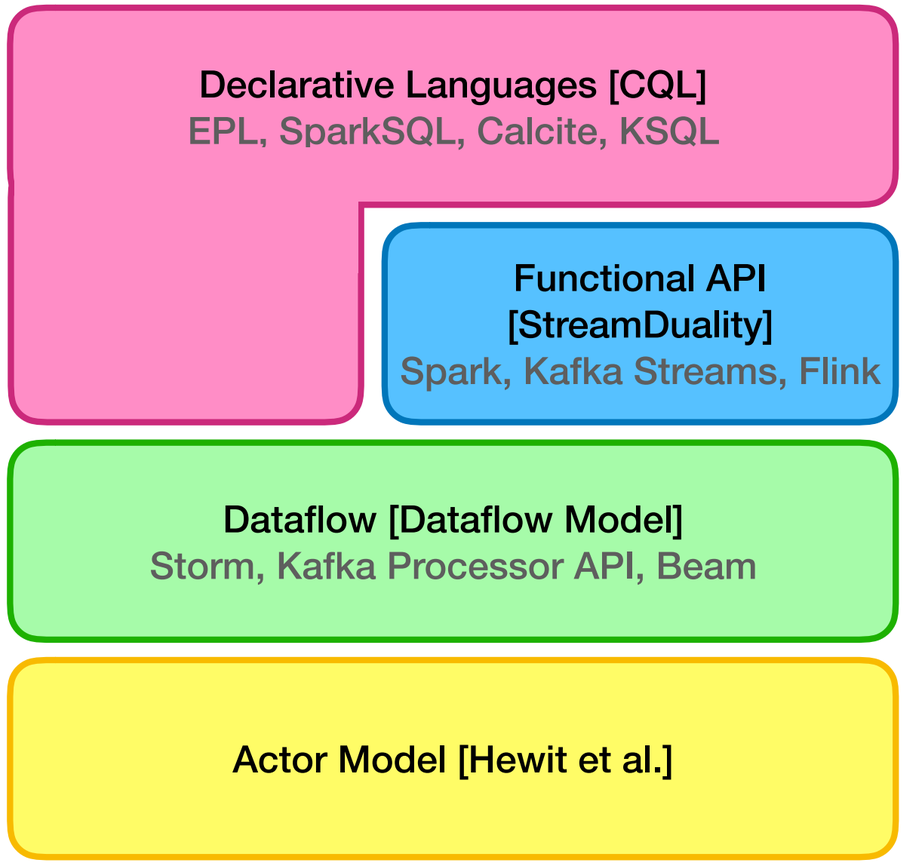
Kappa Architecture



Programming with Streams

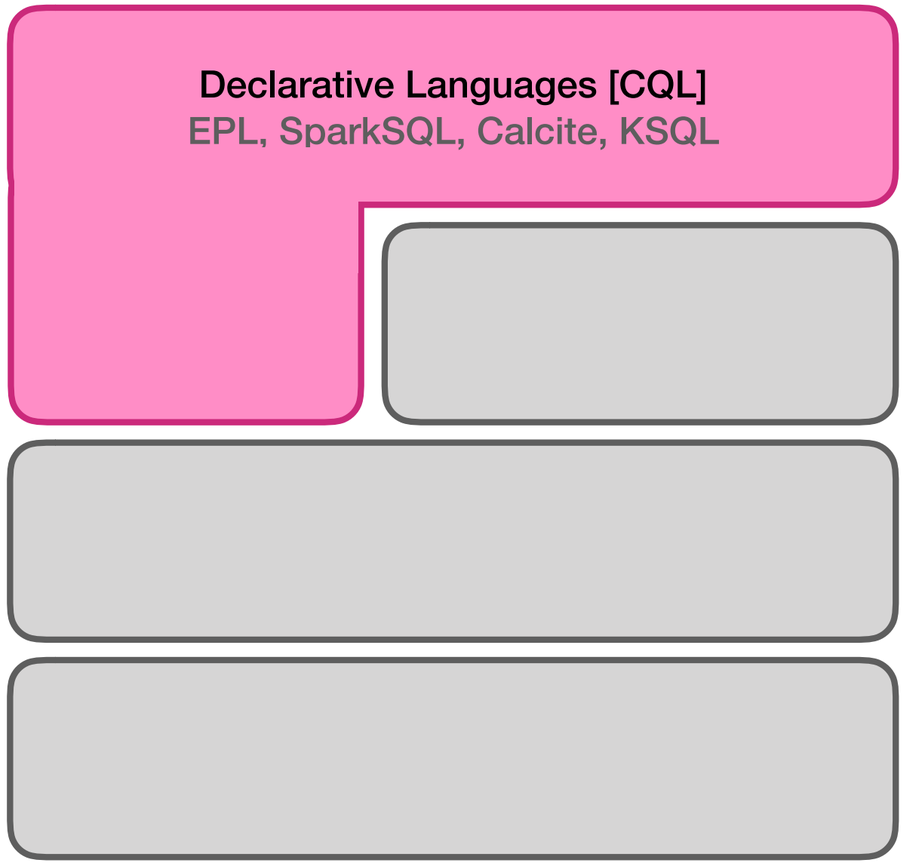
Stream processing frameworks hide execution details from the programmers, and manage them in the background.

There are different abstraction levels that a programmer can use to express streaming computations.



SQL-like Languages

Prominent batch-processing solutions provide SQL interfaces e.g., Hive, PIG, SparkSQL. The reasons include the access to a wider audience and all the benefits of declarative languages. Similarly, Stream Processing systems are migrating towards SQL-like languages. Can you guess what kind of extensions they have? Exactly! Window Operators!



Continuous Query Language

The infinite nature of streams requires a change of paradigm in the way we process data. A first approach to this is given by the Continuous
Query Language (CQL)[5]

[5] Arasu, A., Babu, S., & Widom, J. (2006). The CQL continuous query language: semantic foundations and query execution. The VLDB Journal, 15(2), 121-142.

Core Abstractions

A **Stream** S is a possibly infinite multi-set of elements <s,t> where s is a tuple belonging to the schema of S and t is a timestamp.

Relation R is a mapping from each time instant in T to a finite but unbounded bag of tuples belonging to the schema of R.

Operator Families

CQL includes three families of operators that reconcile stream processing with traditional relational algebra

Operator Families

CQL includes three families of operators that reconcile stream processing with traditional relational algebra

- Stream-to-Relation operators

Operator Families

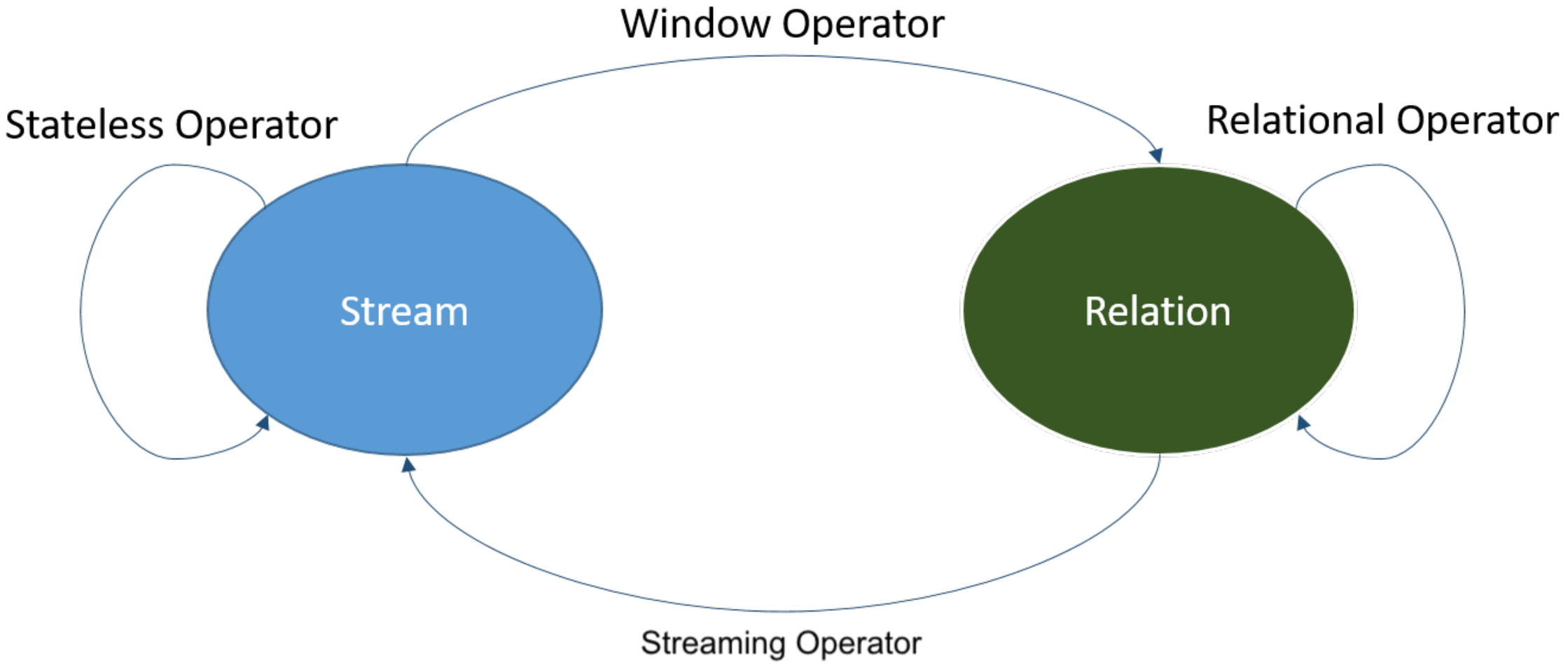
CQL includes three families of operators that reconcile stream processing with traditional relational algebra

- Stream-to-Relation operators
- Relation-to-Relation operators

Operator Families

CQL includes three families of operators that reconcile stream processing with traditional relational algebra

- Stream-to-Relation operators
- Relation-to-Relation operators
- Relation-to-Stream operators


Window Operator
Stateless Operator
Relational Operator
Stream
Relation
Streaming Operator

Stream-to-Relation Operators

Stream-to-Relation Operators

- Time-based Window Operators

Stream-to-Relation Operators

- Time-based Window Operators
- Count based Window operators

Stream-to-Relation Operators

- Time-based Window Operators
- Count based Window operators
- Data Driven window operators

Relation-to-Relation Operators

Relation-to-Relation Operators

- Extension of relational algebra

Relation-to-Stream operators

Relation-to-Stream operators

- Rstream: streams out all data in the last step

Relation-to-Stream operators

- Rstream: streams out all data in the last step
- Istream: streams out data in the last step that wasn’t on the previous step, i.e. streams out what is new

Relation-to-Stream operators

- Rstream: streams out all data in the last step
- Istream: streams out data in the last step that wasn't on the previous step, i.e. streams out what is new
- Dstream: streams out data in the previous step that isn't in the last step, i.e. streams out what is old

KSQL-DB

Core Abstractions

Core Operations

DDL(Data Definition Language) Statements

```
CREATE STREAM ratings (
    rating_idlong,
    user_idint,
    stars int,
    route_idint,
    rating_timelong,
    channel varchar,
    message varchar)
WITH(
    value_format='JSON',
    kafka_topic='ratings');
```

```
CREATE TABLE users (
    uid int,
    name varchar,
    elite varchar
     )
WITH(
    Key= 'uid',
    value_format='JSON',
    kafka_topic='mysql-users');
```

DML(Data Manipulation Language) Aggregation & Windows

TUMBLING

```
SELECT uid, name, count(*) as rating_count
FROM vip_poor_ratings
WINDOW TUMBLING(size 2 minutes)
GROUPBY uid, name;
```

HOPPING

```
SELECT uid, name, count(*) as rating_count
FROM vip_poor_ratings
WINDOW HOPPING(size 1 minutes) ADVANCED BY 30 SECONDS
GROUPBY uid, name;
```

DML(Data Manipulation Language) Aggregation & Windows

- Tumbling

TUMBLING

```
SELECT uid, name, count(*) as rating_count
FROM vip_poor_ratings
WINDOW TUMBLING(size 2 minutes)
GROUPBY uid, name;
```

HOPPING

```
SELECT uid, name, count(*) as rating_count
FROM vip_poor_ratings
WINDOW HOPPING(size 1 minutes) ADVANCED BY 30 SECONDS
GROUPBY uid, name;
```

DML(Data Manipulation Language) Aggregation & Windows

- Tumbling
- Hopping

TUMBLING

```
SELECT uid, name, count(*) as rating_count
FROM vip_poor_ratings
WINDOW TUMBLING(size 2 minutes)
GROUPBY uid, name;
```

HOPPING

```
SELECT uid, name, count(*) as rating_count
FROM vip_poor_ratings
WINDOW HOPPING(size 1 minutes) ADVANCED BY 30 SECONDS
GROUPBY uid, name;
```

DML(Data Manipulation Language) Aggregation & Windows

- Tumbling
- Hopping
- Session

TUMBLING

```
SELECT uid, name, count(*) as rating_count
FROM vip_poor_ratings
WINDOW TUMBLING(size 2 minutes)
GROUPBY uid, name;
```

HOPPING

```
SELECT uid, name, count(*) as rating_count
FROM vip_poor_ratings
WINDOW HOPPING(size 1 minutes) ADVANCED BY 30 SECONDS
GROUPBY uid, name;
```

DML(Data Manipulation Language) Aggregation & Windows

- Tumbling
- Hopping
- Session
- COUNT

TUMBLING

```
SELECT uid, name, count(*) as rating_count
FROM vip_poor_ratings
WINDOW TUMBLING(size 2 minutes)
GROUPBY uid, name;
```

HOPPING

```
SELECT uid, name, count(*) as rating_count
FROM vip_poor_ratings
WINDOW HOPPING(size 1 minutes) ADVANCED BY 30 SECONDS
GROUPBY uid, name;
```

DML(Data Manipulation Language) Aggregation & Windows

- Tumbling
- Hopping
- Session
- COUNT
- SUM

TUMBLING

```
SELECT uid, name, count(*) as rating_count
FROM vip_poor_ratings
WINDOW TUMBLING(size 2 minutes)
GROUPBY uid, name;
```

HOPPING

```
SELECT uid, name, count(*) as rating_count
FROM vip_poor_ratings
WINDOW HOPPING(size 1 minutes) ADVANCED BY 30 SECONDS
GROUPBY uid, name;
```

DML(Data Manipulation Language) Aggregation & Windows

- Tumbling
- Hopping
- Session
- COUNT
- SUM
- MIN

TUMBLING

```
SELECT uid, name, count(*) as rating_count
FROM vip_poor_ratings
WINDOW TUMBLING(size 2 minutes)
GROUPBY uid, name;
```

HOPPING

```
SELECT uid, name, count(*) as rating_count
FROM vip_poor_ratings
WINDOW HOPPING(size 1 minutes) ADVANCED BY 30 SECONDS
GROUPBY uid, name;
```

DML(Data Manipulation Language) Aggregation & Windows

- Tumbling
- Hopping
- Session
- COUNT
- SUM
- MIN
- MAX

TUMBLING

```
SELECT uid, name, count(*) as rating_count
FROM vip_poor_ratings
WINDOW TUMBLING(size 2 minutes)
GROUPBY uid, name;
```

HOPPING

```
SELECT uid, name, count(*) as rating_count
FROM vip_poor_ratings
WINDOW HOPPING(size 1 minutes) ADVANCED BY 30 SECONDS
GROUPBY uid, name;
```

DML Statements: JOINs

```
CREATE STREAM vip_poor_ratings AS
SELECT uid, name, elite,
stars, route_id, rating_time, message
FROM poor_ratingsr LEFT JOIN users u ON r.user_id= u.uid
WHERE u.elite= 'P';
```

DML Statements: JOINs

- Join

```
CREATE STREAM vip_poor_ratings AS
SELECT uid, name, elite,
stars, route_id, rating_time, message
FROM poor_ratingsr LEFT JOIN users u ON r.user_id= u.uid
WHERE u.elite= 'P';
```

DML Statements: JOINs

- Join
  - Stream-Stream

```
CREATE STREAM vip_poor_ratings AS
SELECT uid, name, elite,
stars, route_id, rating_time, message
FROM poor_ratingsr LEFT JOIN users u ON r.user_id= u.uid
WHERE u.elite= 'P';
```

DML Statements: JOINs

- Join
  - Stream-Stream
  - **Stream-Table**

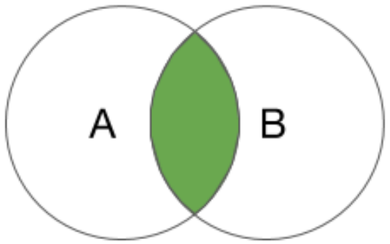
```
CREATE STREAM vip_poor_ratings AS
SELECT uid, name, elite,
stars, route_id, rating_time, message
FROM poor_ratingsr LEFT JOIN users u ON r.user_id= u.uid
WHERE u.elite= 'P';
```

DML Statements: JOINs

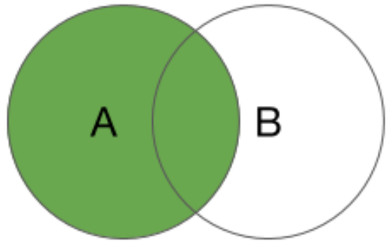
- Join
  - Stream-Stream
  - **Stream-Table**
  - Table-Table

```
CREATE STREAM vip_poor_ratings AS
SELECT uid, name, elite,
stars, route_id, rating_time, message
FROM poor_ratingsr LEFT JOIN users u ON r.user_id= u.uid
WHERE u.elite= 'P';
```

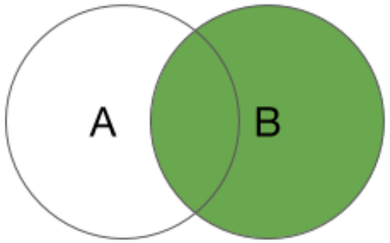
Traditional Table-Table Joins



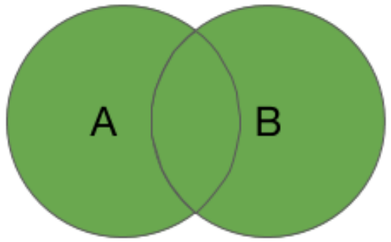
INNER JOIN



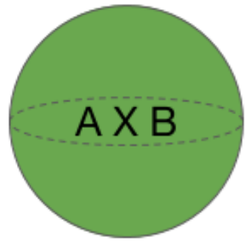
LEFT OUTER JOIN



RIGHT OUTER JOIN

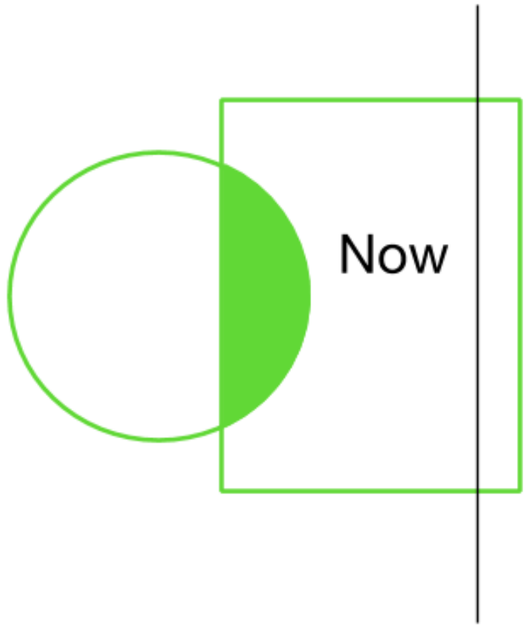


FULL OUTER JOIN

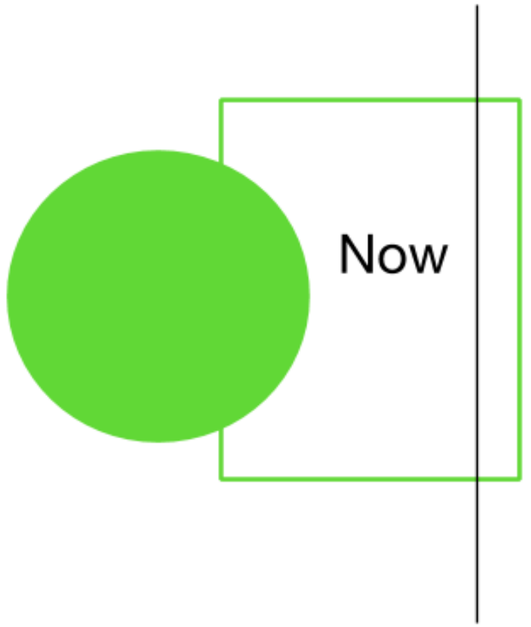


CARTESIAN (CROSS) JOIN

Stream-Table Joins

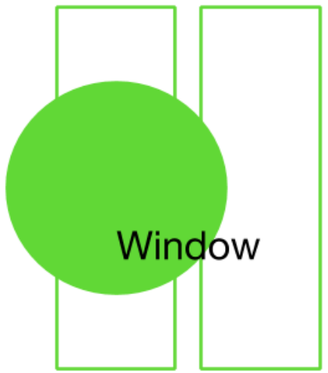


Inner

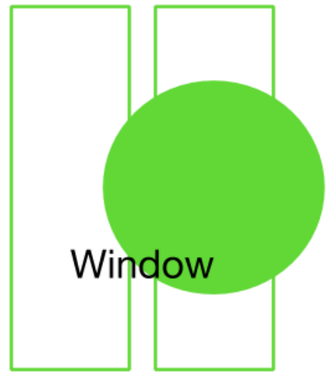


Left Outer

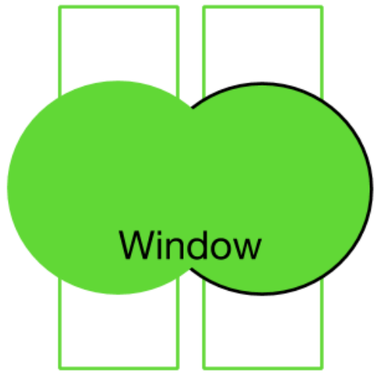
Stream-Stream Joins



Left Outer



Right Outer



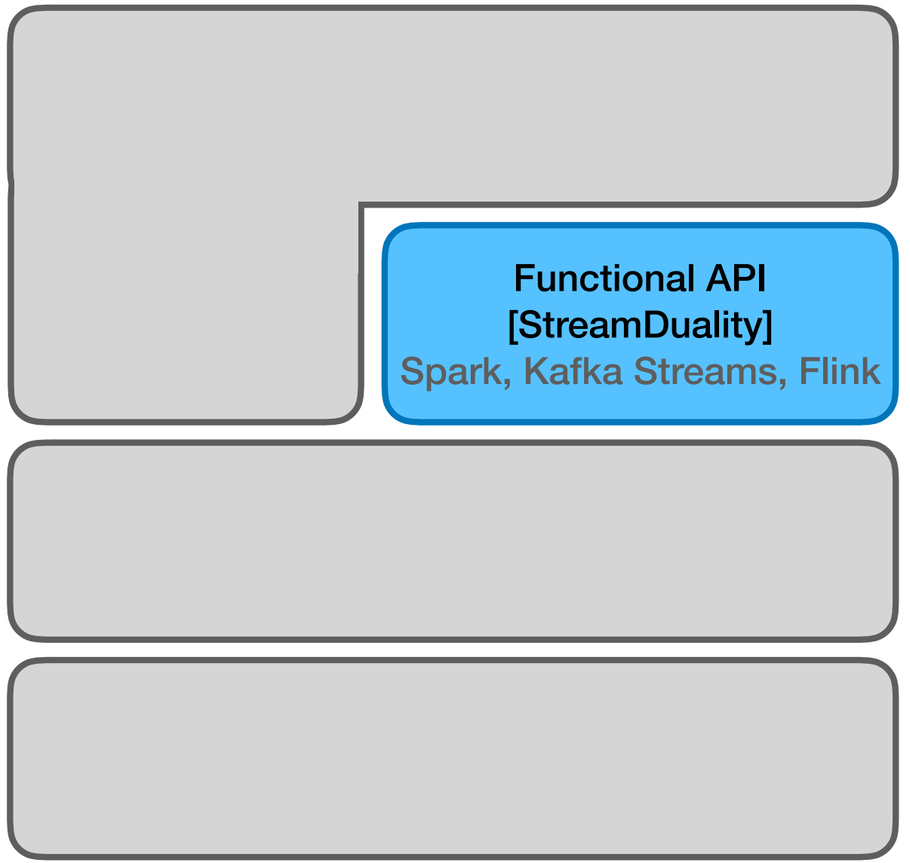
Full Outer

Functional Languages

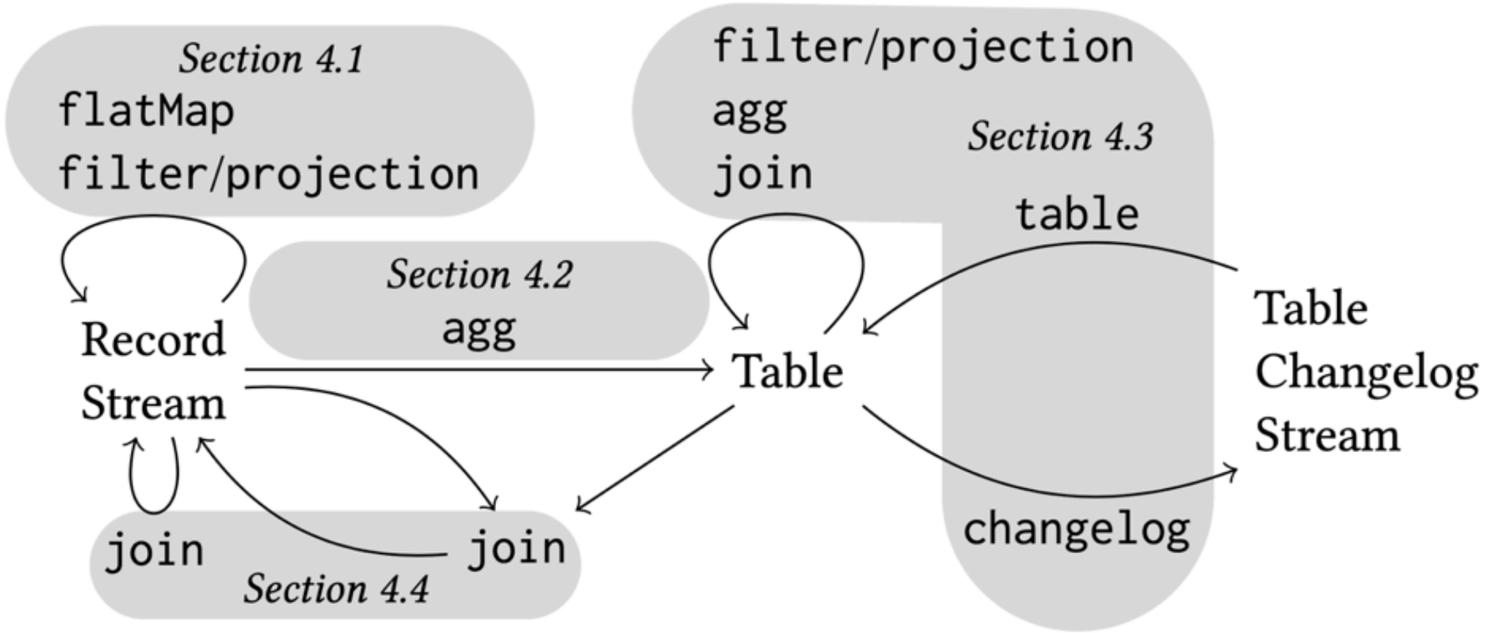
Stream Processing frameworks offer functional APIs to directly write streaming programs.

Function programming provides a mechanism for the representation of Streaming Transformation using higher-order function such as filter, maps, and flatmaps.

Functional APIs are still declarative. However, they give more freedom to the developer who needs to design specialized operations, e.g., aggregations.

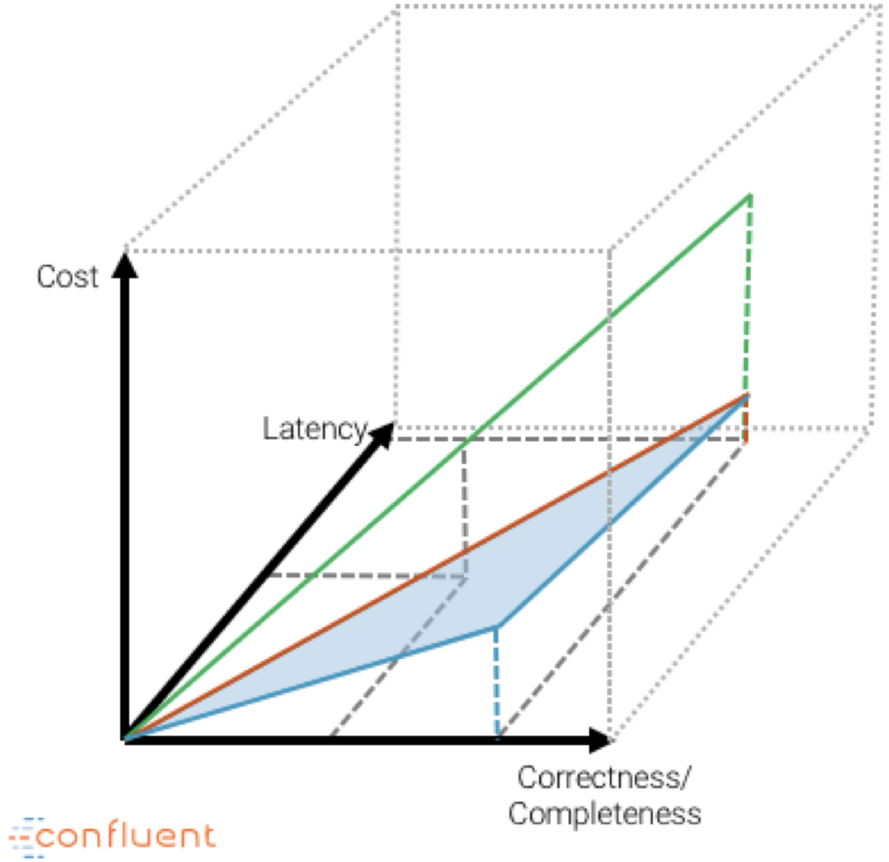


Reference Model[6]



[6] Sax, Matthias J., et al. "Streams and tables: Two sides of the same coin." Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics. 2018.

Design Space

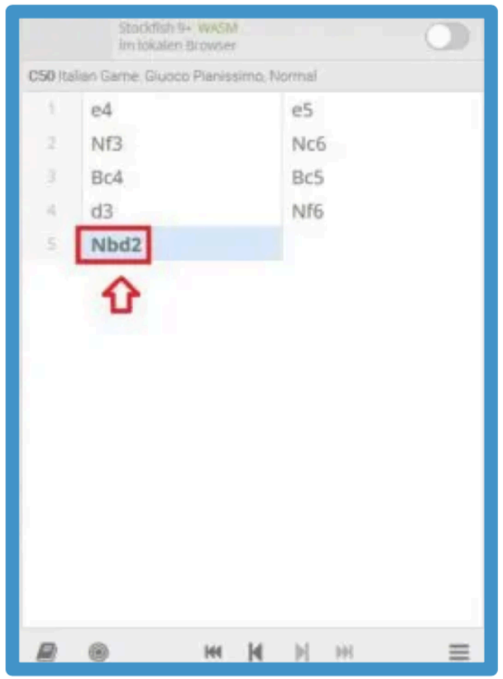


Buffering and Reordering
- Ref: CQL[1], Trill[2]

Punctuations/Watermarks
- Ref: Li et al.[3], Krishnamurthy et al.[4]

Dual Streaming Model
- Continuous updates/changelogs
- Decouple latency from correctness
- Trade-off latency and cost
- Trade-off cost and completeness (retention time)

Intuition



Streams record History
"The sequence of moves."

Tables represent State
"The state of the board at last move."

Trasformations

Stateless

Stateful

Trasformations

Stateless **Stateful**

- map

Trasformations

Stateless

- map
- flatMap

Stateful

Trasformations

Stateless

- map
- flatMap
- filter

Stateful

Trasformations

Stateless

- map
- flatMap
- filter

Stateful

Trasformations

Stateless

- map
- flatMap
- filter

Stateful

- join

Trasformations

Stateless

- map
- flatMap
- filter

Stateful

- join
  - based on the partitioning key, not relational equijoin

Trasformations

Stateless

- map
- flatMap
- filter

Stateful

- join
  - based on the partitioning key, not relational equijoin
- aggregation,

Trasformations

Stateless

- map
- flatMap
- filter

Stateful

- join
  - based on the partitioning key, not relational equijoin
- aggregation,
  - requires key selection

Operator	Input 1	Input 2	Output
filter/mapValue	KStream		KStream
filter/mapValue	KTable		KTable
map, flatMap	KStream		KStream
groupBy -> agg	KStream		KTable
groupBy -> agg	KTable		KTable
groupBy + windowBy -> agg	KStream		KTable
inner-/left/outer-join	KStream	KStream	KStream
inner-/left/outer-join	KTable	KTable	KTable
inner-/left-join	KStream	KTable	KStream

Kafka Stream

```
KStream<String, String> textLines =
    builder.stream("TextLinesTopic");

KTable<String, Long> wordCounts = textLines
.flatMapValues(textLine ->
    Arrays.asList(
        textLine
        .toLowerCase()
        .split("\\W+")))
.groupBy((key, word) -> word)
.count("Counts");

wordCounts.to(
    Serdes.String(),
    Serdes.Long(),
    "WordsWithCountsTopic");
```

Core Abstractions

Core Abstractions

- KStream is an abstraction of a record stream (of key-value pairs) INSERT ONLY

Core Abstractions

- KStream is an abstraction of a record stream (of key-value pairs) INSERT ONLY
- KTable is an abstraction of a changelog stream. Data records are interpreted as

Core Abstractions

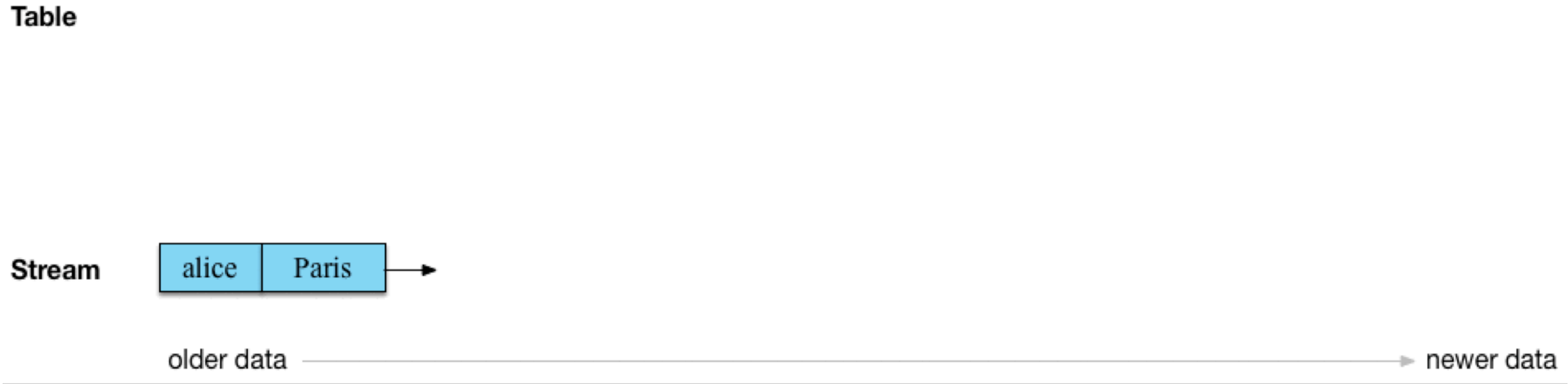
- KStream is an abstraction of a record stream (of key-value pairs) INSERT ONLY
- KTable is an abstraction of a changelog stream. Data records are interpreted as
  - "UPDATE" of the last value for the same record key

Core Abstractions

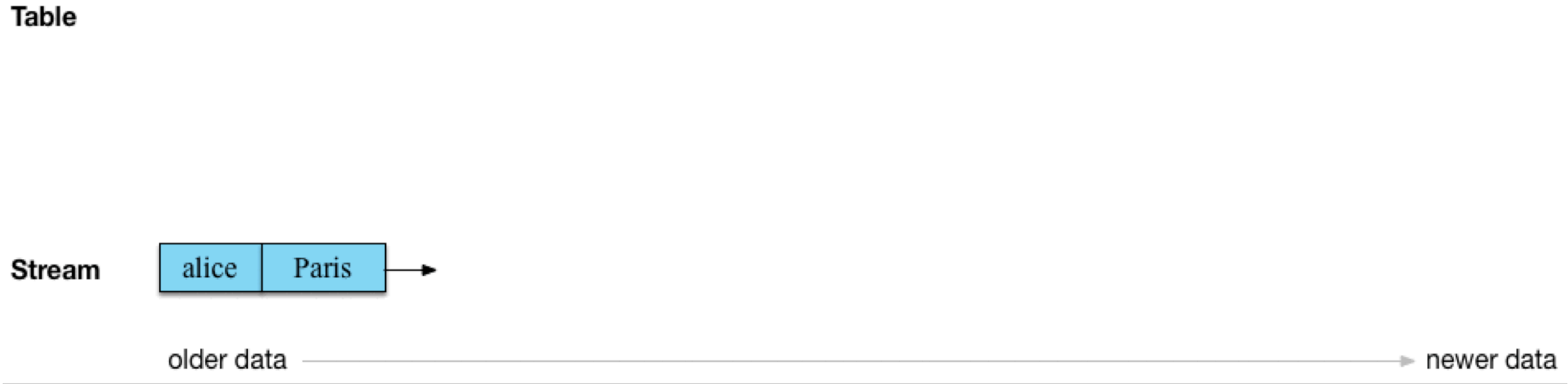
- KStream is an abstraction of a record stream (of key-value pairs) INSERT ONLY
- KTable is an abstraction of a changelog stream. Data records are interpreted as
  - "UPDATE" of the last value for the same record key
  - "INSERT" if a given key doesn't exist yet "DELETE" or tombstone if the record value is null

Core Abstractions

- KStream is an abstraction of a record stream (of key-value pairs) INSERT ONLY
- KTable is an abstraction of a changelog stream. Data records are interpreted as
  - "UPDATE" of the last value for the same record key
  - "INSERT" if a given key doesn't exist yet "DELETE" or tombstone if the record value is null
- GlobalKTable is similar to a KTable but is fully replicated per KafkaStreams instance



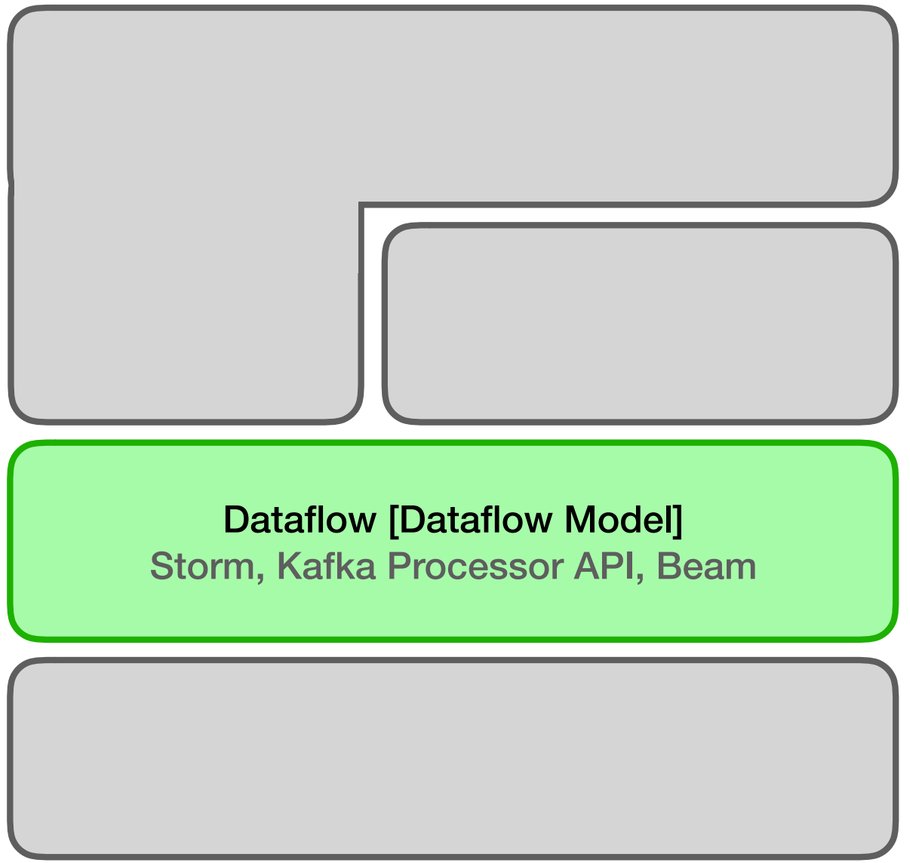

```
KStream<String, String> textLines =
    builder.stream("people-places");

KTable<String, String> wherenow =
    .groupByKey()
    .reduce((oldPlace, newPlace) -> newPlace);
```




```
KStream<String, String> textLines =
    builder.stream("people-places");

KTable<String, Long> wherenow =
    .groupByKey()
    .count()
```

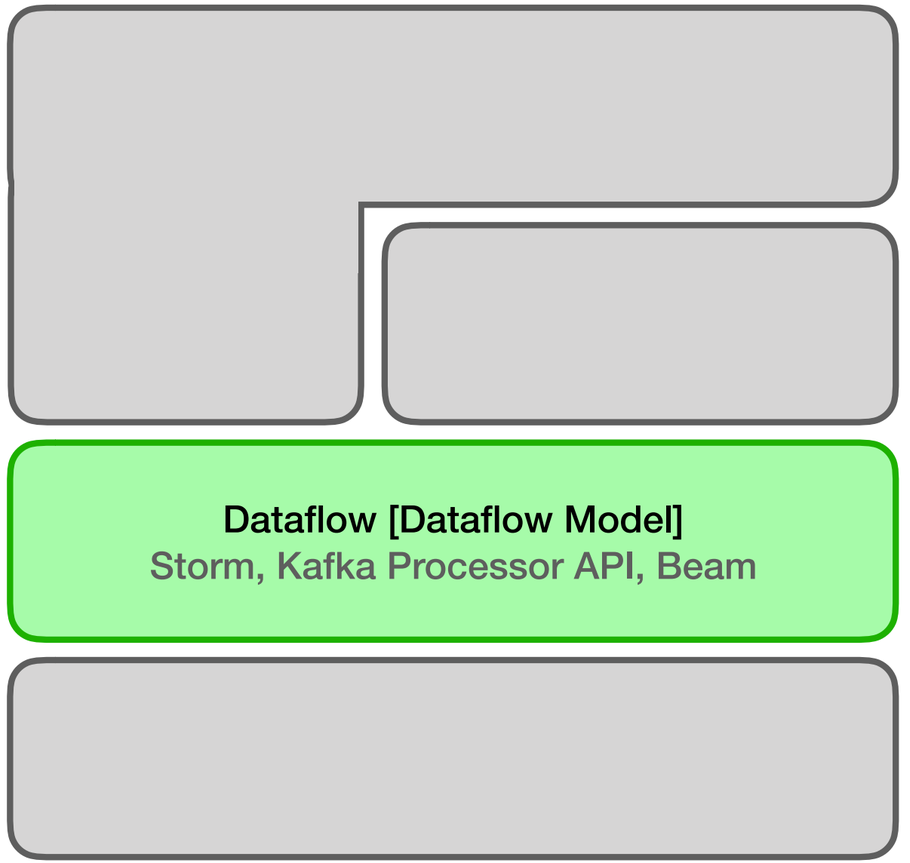
Core Operations

Programming with Streams: Dataflow



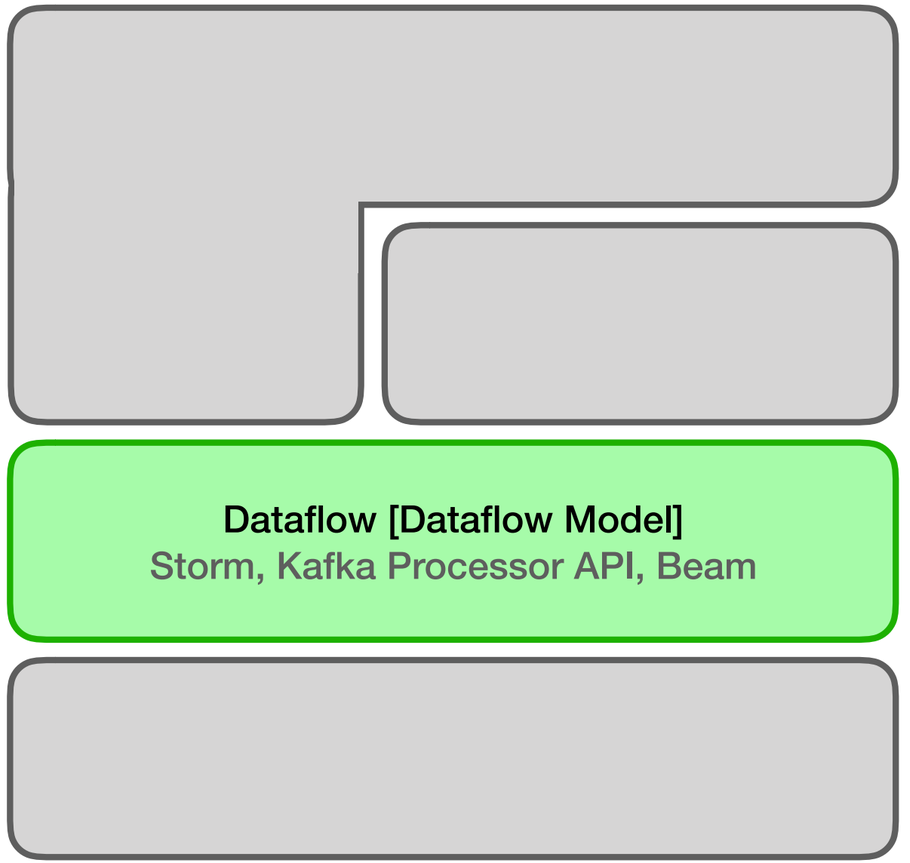
Programming with Streams: Dataflow

- As dataflow networks were the first type of streaming programs to appear in the literature.



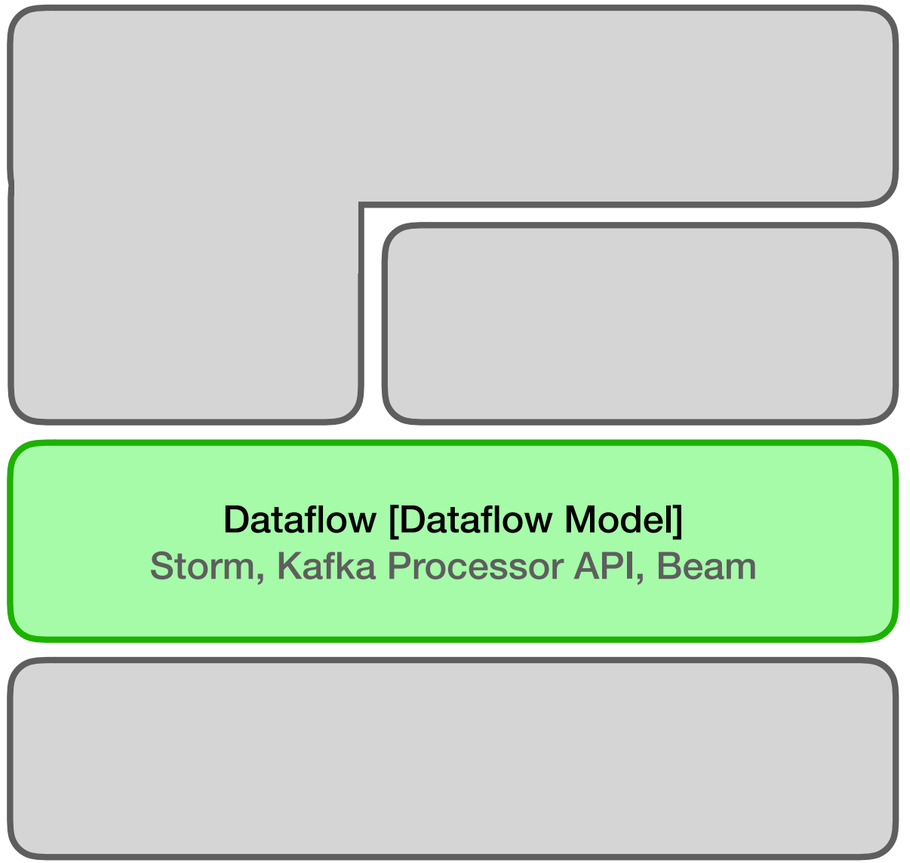
Programming with Streams: Dataflow

- As dataflow networks were the first type of streaming programs to appear in the literature.
- A dataflow network represents a program as nodes, i.e., operators, and edges, i.e., data streams.



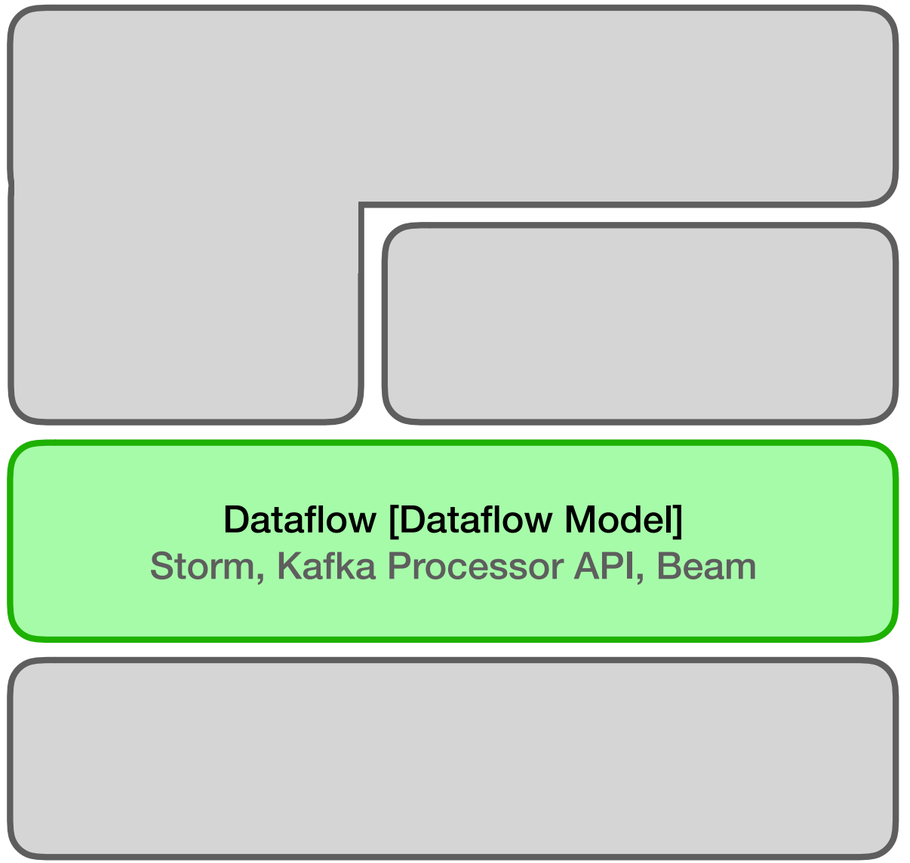
Programming with Streams: Dataflow

- As dataflow networks were the first type of streaming programs to appear in the literature.
- A dataflow network represents a program as nodes, i.e., operators, and edges, i.e., data streams.
- Nodes can only communicate with each other by their input and output connections.



Programming with Streams: Dataflow

- As dataflow networks were the first type of streaming programs to appear in the literature.
- A dataflow network represents a program as nodes, i.e., operators, and edges, i.e., data streams.
- Nodes can only communicate with each other by their input and output connections.
- Languages for dataflow programming offer programmers the primitives to implement custom business logic as topologies of nodes.



Logical Dataflow

Logical Dataflow

- What programmer design is actually the logical dataflow plan.

Logical Dataflow

- What programmer design is actually the logical dataflow plan.
- A stream processing system distributes a dataflow graph across multiple machines.

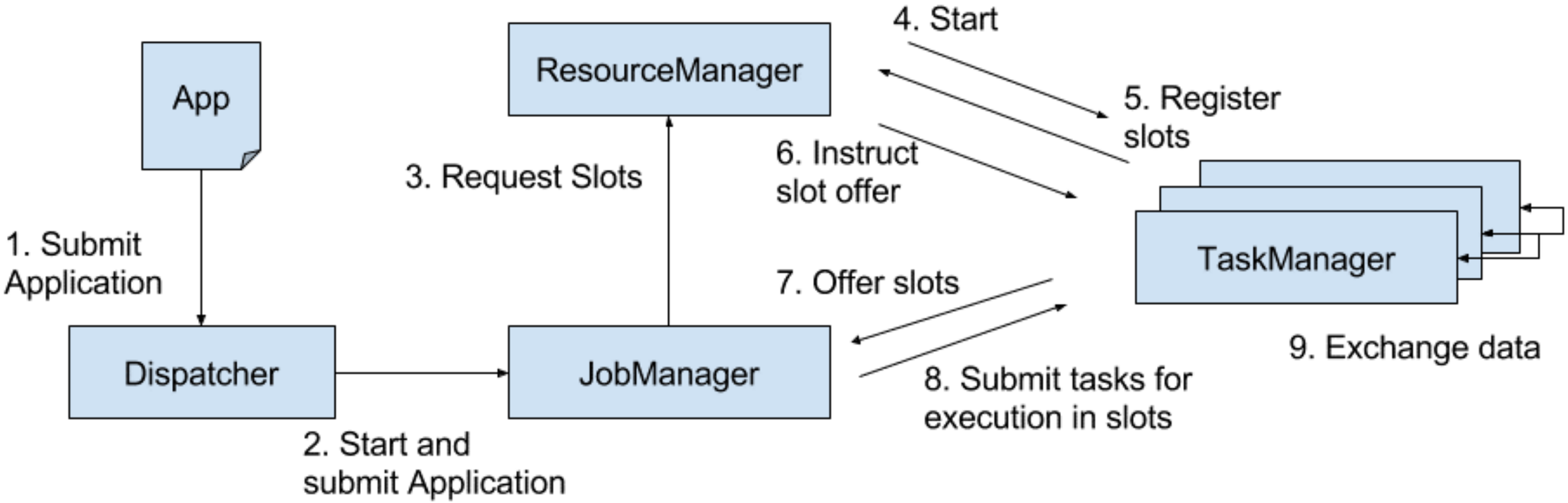
Logical Dataflow

- What programmer design is actually the logical dataflow plan.
- A stream processing system distributes a dataflow graph across multiple machines.
- The system is also responsible for managing the partitioning of data, the network communication, as well as program recovery in case of machine failure.

Logical Dataflow

- What programmer design is actually the logical dataflow plan.
- A stream processing system distributes a dataflow graph across multiple machines.
- The system is also responsible for managing the partitioning of data, the network communication, as well as program recovery in case of machine failure.
- In dataflow programming, the programmers can control the degree of parallelism and, thus, part of the physical execution.

Physical Dataflow Network



Dataflow Programming: Stateful Operators

Dataflow Programming: Stateful Operators

- Unlike a simple operator such as `filter`, certain operators need to keep mutable state.

Dataflow Programming: Stateful Operators

- Unlike a simple operator such as `filter`, certain operators need to keep mutable state.
- For instance, an operator that `Count` all the occurrence of a certain event must keep a state of the current counts.

Dataflow Programming: Stateful Operators

- Unlike a simple operator such as `filter`, certain operators need to keep mutable state.
- For instance, an operator that `Count` all the occurrence of a certain event must keep a state of the current counts.
- In the the word counting example, counting the word occurrences received by an operator, requires storing the words received thus far along with their respective counts.

The Dataflow Model[2]

The model provides a framework to answer four questions:

[2] Akidau, Tyler, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing.(2015).

The Dataflow Model[2]

The model provides a framework to answer four questions:

- **What** results are we computing?

[2] Akidau, Tyler, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing.(2015).

The Dataflow Model[2]

The model provides a framework to answer four questions:

- **What** results are we computing?
- **Where** in event-time are they computed?

[2] Akidau, Tyler, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing.(2015).

The Dataflow Model[2]

The model provides a framework to answer four questions:

- **What** results are we computing?
- **Where** in event-time are they computed?
- **When** in processing time are they materialized?

[2] Akidau, Tyler, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing.(2015).

The Dataflow Model[2]

The model provides a framework to answer four questions:

- **What** results are we computing?
- **Where** in event-time are they computed?
- **When** in processing time are they materialized?
- **How** do results are reported ?

---

[2] Akidau, Tyler, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing.(2015).

PCollection[2]

```
PCollection<String> raw = IO.read(...);
PCollection<KV<String, Integer>> input = raw.apply(ParDo.of(new ParseFn());
PCollection<KV<String, Integer>> scores = input.apply(Sum.integersPerKey());
```

[2] Akidau, Tyler, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing.(2015).

PCollection[2]

- A `PCollection <T>` is potentially **unbounded** collection of data Type `T` , initially created from backing data stores like a file or a websocket.

```
PCollection<String> raw = IO.read(...);
PCollection<KV<String, Integer>> input = raw.apply(ParDo.of(new ParseFn());
PCollection<KV<String, Integer>> scores = input.apply(Sum.integersPerKey());
```

[2] Akidau, Tyler, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing.(2015).

PCollection[2]

- A `PCollection <T>` is potentially **unbounded** collection of data Type `T` , initially created from backing data stores like a file or a websocket.
- Each element has an Event time timestamp and a processing time timestamp.

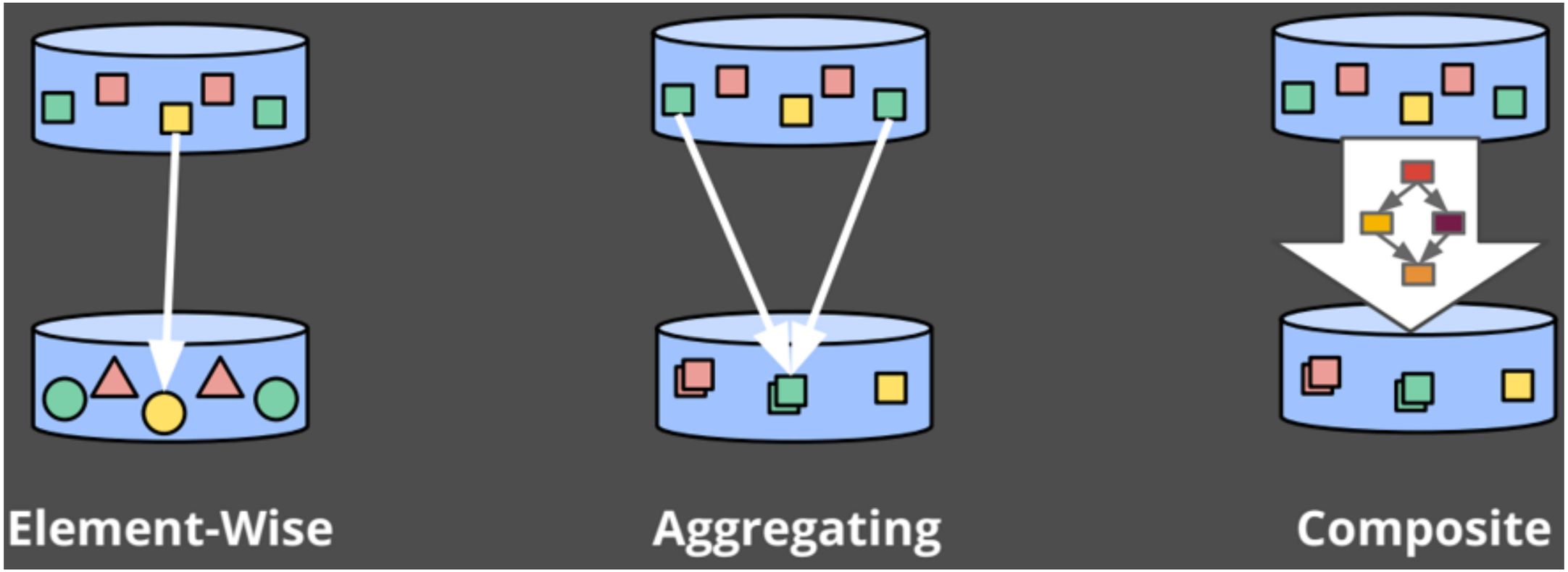
```
PCollection<String> raw = IO.read(...);
PCollection<KV<String, Integer>> input = raw.apply(ParDo.of(new ParseFn());
PCollection<KV<String, Integer>> scores = input.apply(Sum.integersPerKey());
```

---

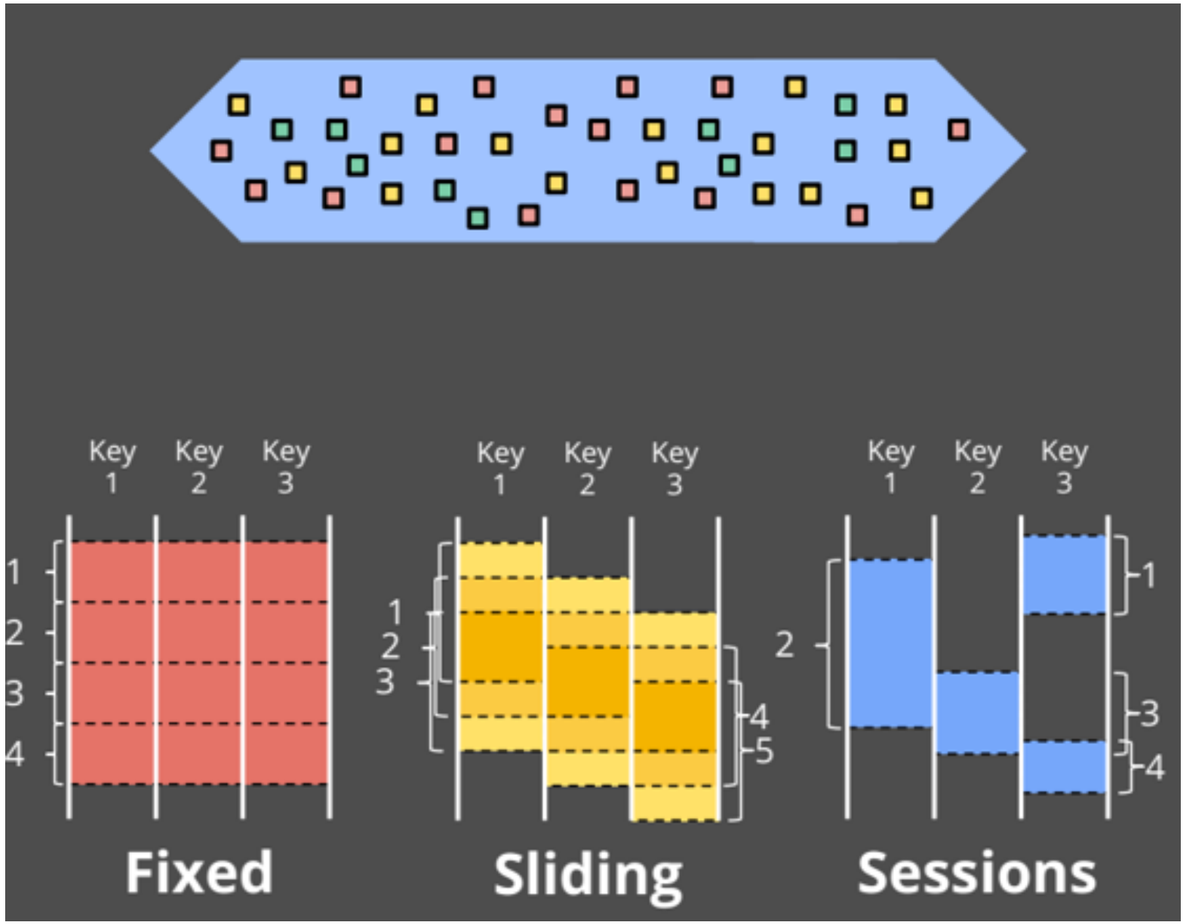
[2] Akidau, Tyler, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing.(2015).

PTransformations

`PTransforms` transform `PCollections` into other `PCollections`



Where: Windowing Over Event-Time



When in Processing Time?

When in Processing Time?

- Triggers control when results are materialized

When in Processing Time?

- Triggers control when results are materialized
  - Watermark trigger

When in Processing Time?

- Triggers control when results are materialized
  - Watermark trigger
  - Processing time trigger

When in Processing Time?

- Triggers control when results are materialized
  - Watermark trigger
  - Processing time trigger
  - Count trigger

When in Processing Time?

- Triggers control when results are materialized
  - Watermark trigger
  - Processing time trigger
  - Count trigger
  - Delta trigger (not supported by Dataflow)

When in Processing Time?

- Triggers control when results are materialized
  - Watermark trigger
  - Processing time trigger
  - Count trigger
  - Delta trigger (not supported by Dataflow)
- Multiple triggers

When in Processing Time?

- Triggers control when results are materialized
  - Watermark trigger
  - Processing time trigger
  - Count trigger
  - Delta trigger (not supported by Dataflow)
- Multiple triggers
  - **Early**: useful to get early results, e.g., a 24-hours window

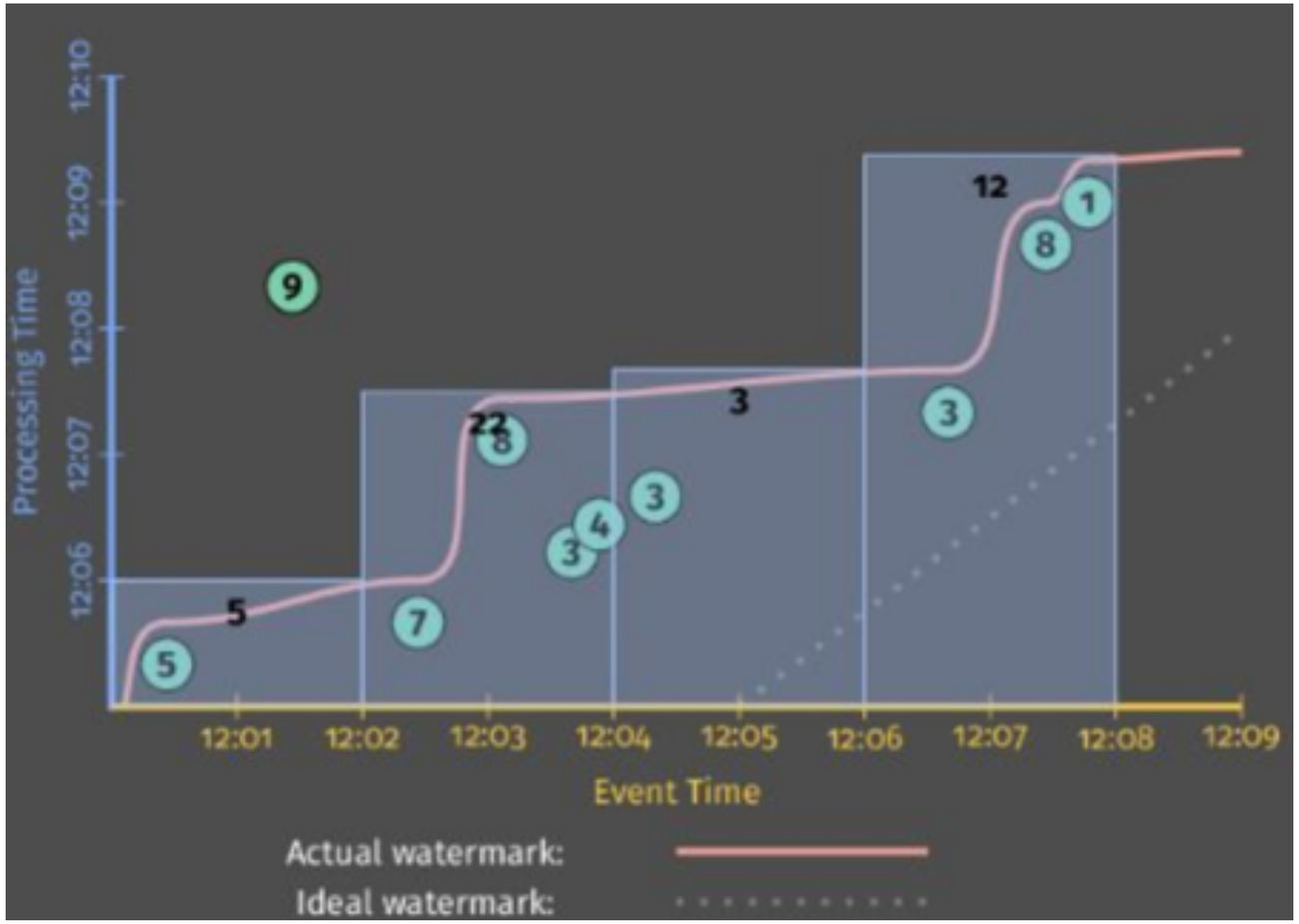
When in Processing Time?

- Triggers control when results are materialized
  - Watermark trigger
  - Processing time trigger
  - Count trigger
  - Delta trigger (not supported by Dataflow)
- Multiple triggers
  - **Early**: useful to get early results, e.g., a 24-hours window
  - **On**: At the window closure time

When in Processing Time?

- Triggers control when results are materialized
  - Watermark trigger
  - Processing time trigger
  - Count trigger
  - Delta trigger (not supported by Dataflow)
- Multiple triggers
  - **Early**: useful to get early results, e.g., a 24-hours window
  - **On**: At the window closure time
  - **Late**: Responding to late arrivals

Trigger at Watermark



How to Refine Results?

How to Refine Results?

- **Discarding**: report the results for the latest firing only

How to Refine Results?

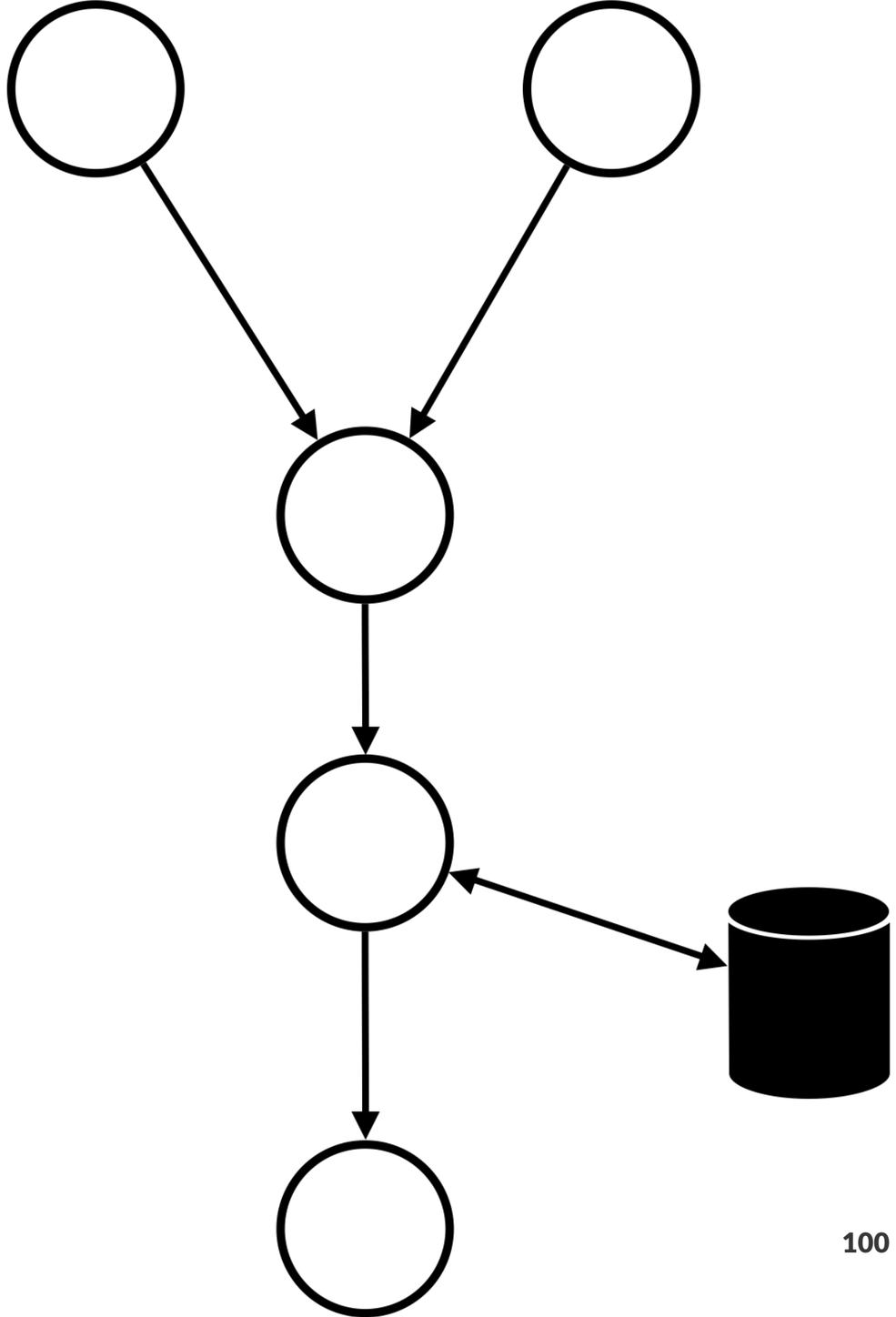
- **Discarding**: report the results for the latest firing only
- **Accumulating**: add to the previous result (requires state)

How to Refine Results?

- **Discarding**: report the results for the latest firing only
- **Accumulating**: add to the previous result (requires state)
- **Accumulating** and Retracting: remove the last update, put the new value

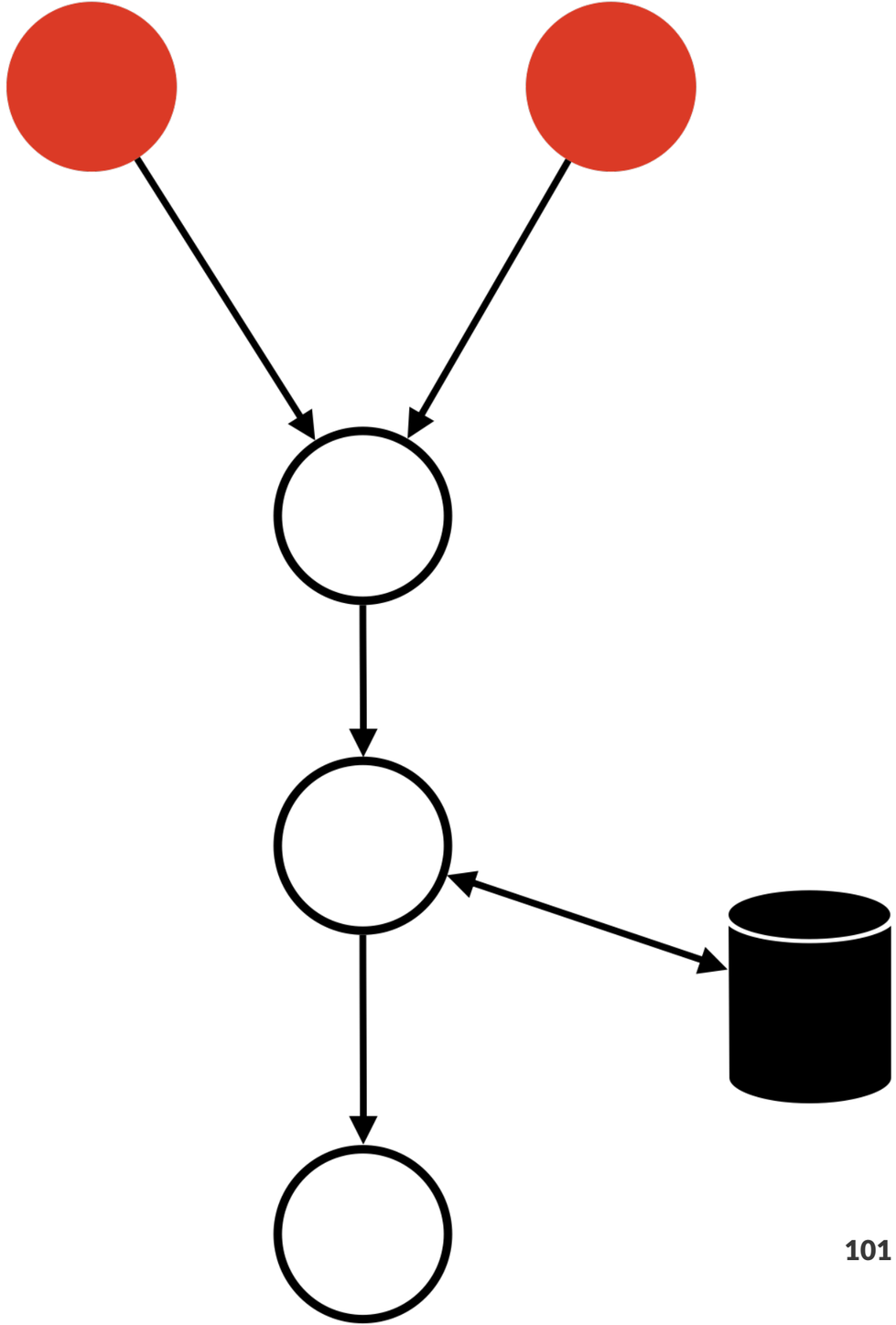
Kafka Streams: Processor API

```
KStream<..> stream1 = builder.stream("topic1");
KStream<..> stream2 = builder.stream("topic2");

KStream<..> joined = stream1.leftJoin(stream2, ...);

KTable<..> aggregated = joined.aggregateByKey(...);
aggregated.to(”topic3”);

builder
    .addSource("Source1", "topic1")
    .addSource("Source2", "topic2")

    .addProcessor("Join", MyJoin:new, "Source1", "Source2")
    .addProcessor("Aggregate", MyAggregate:new, "Join")
    .addStateStore(Stores.persistent().build(), "Aggregate")

    .addSink("Sink", "topic3", "Aggregate")
```



Sources

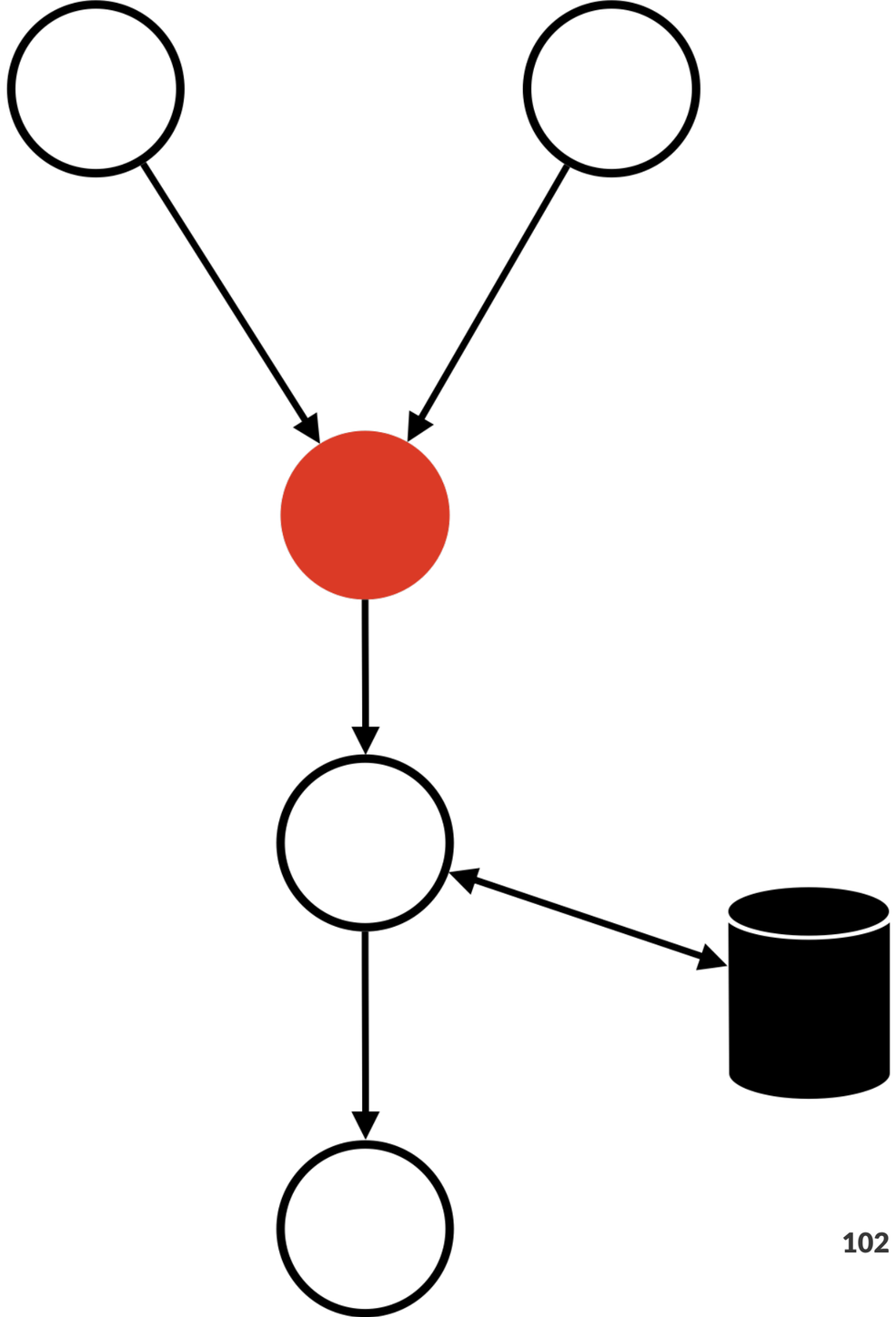
```
KStream<..> stream1 = builder.stream("topic1");
KStream<..> stream2 = builder.stream("topic2");

builder
    .addSource(
        "Source1",
        "topic1")
    .addSource(
        "Source2",
        "topic2")
```



Join

```
KStream<..> joined =
    stream1.leftJoin(stream2, ...Fn);
```

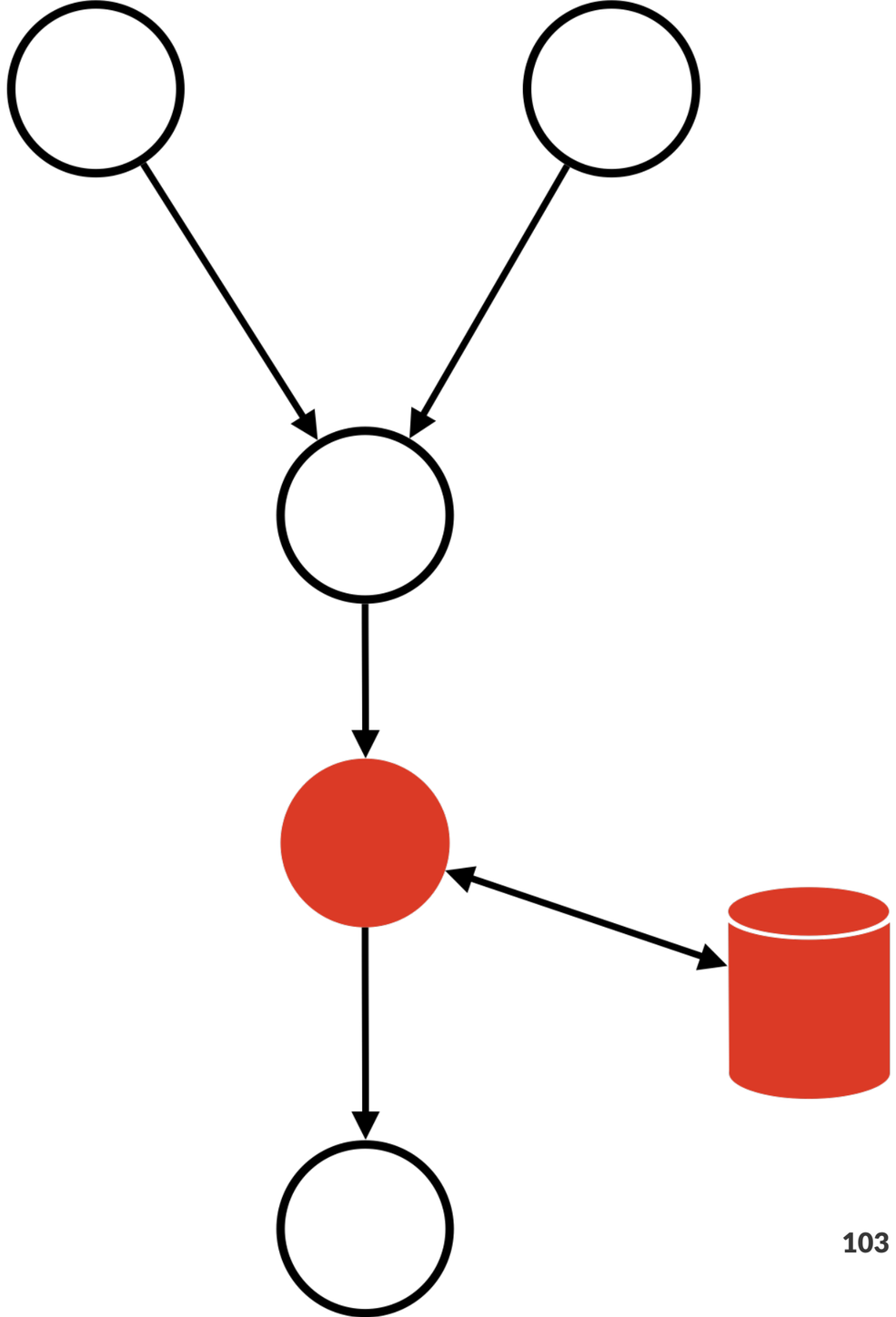
```
builder.addProcessor(
    "Join",
    MyJoin:new,
    "Source1",
    "Source2")
```



Stateful Aggregation

```
KTable<..> aggregated =
    joined.aggregateByKey(...);
```

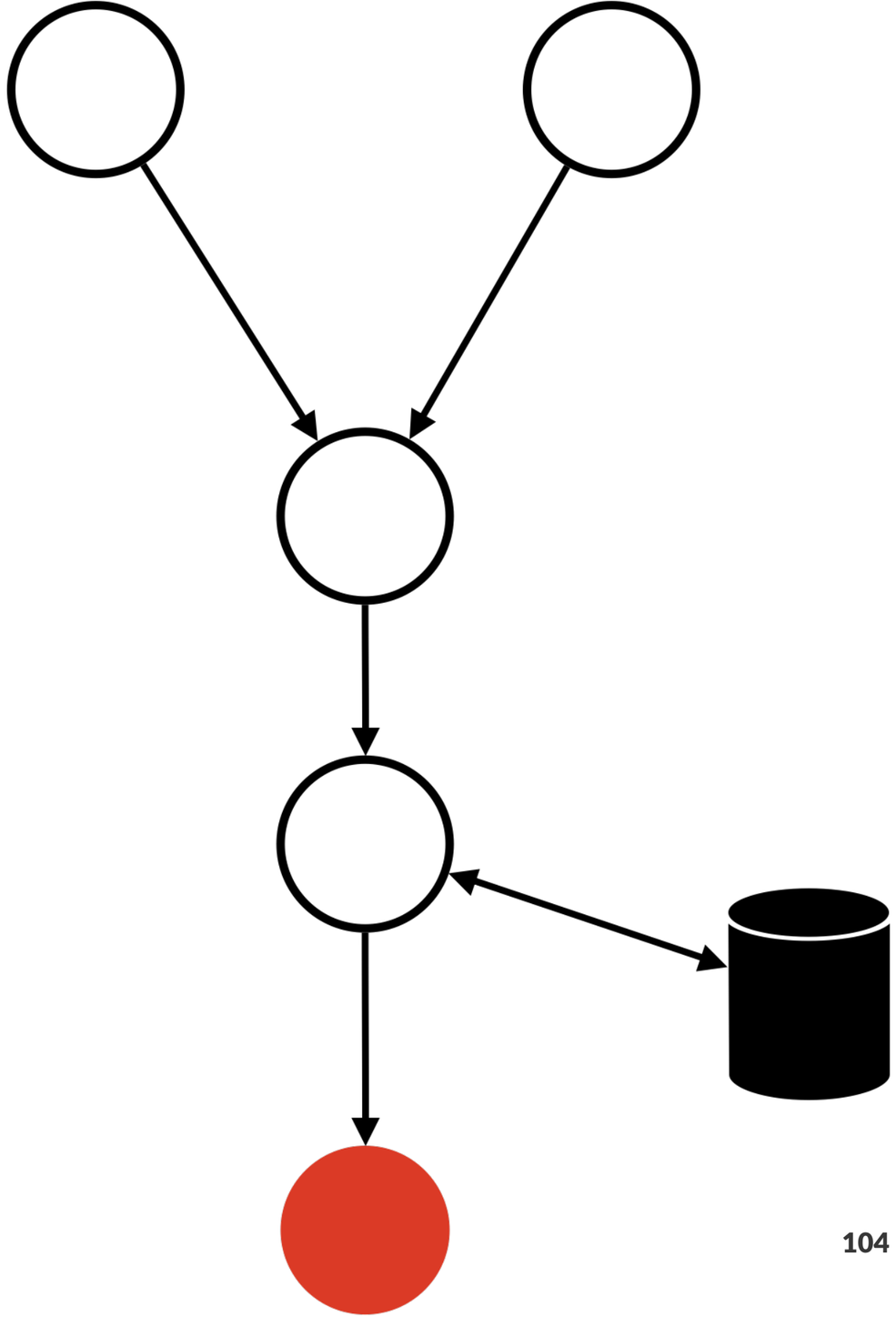
```
builder
    .addProcessor(
        "Aggregate",
        MyAggregate:new, "
        Join")
    .addStateStore(
        Stores.persistent().build(),
        "Aggregate")
```



Sink

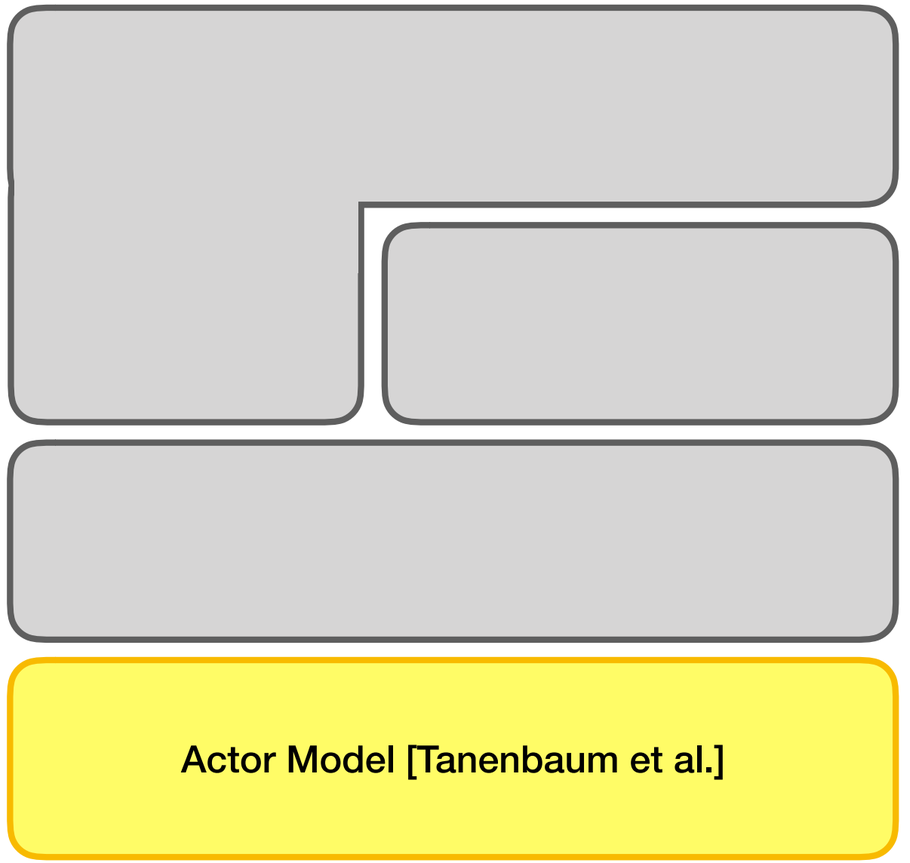
```
aggregated.to(”topic3”);
```

```
builder.addSink(”Sink”, ”topic3”, ”Aggregate”)
```

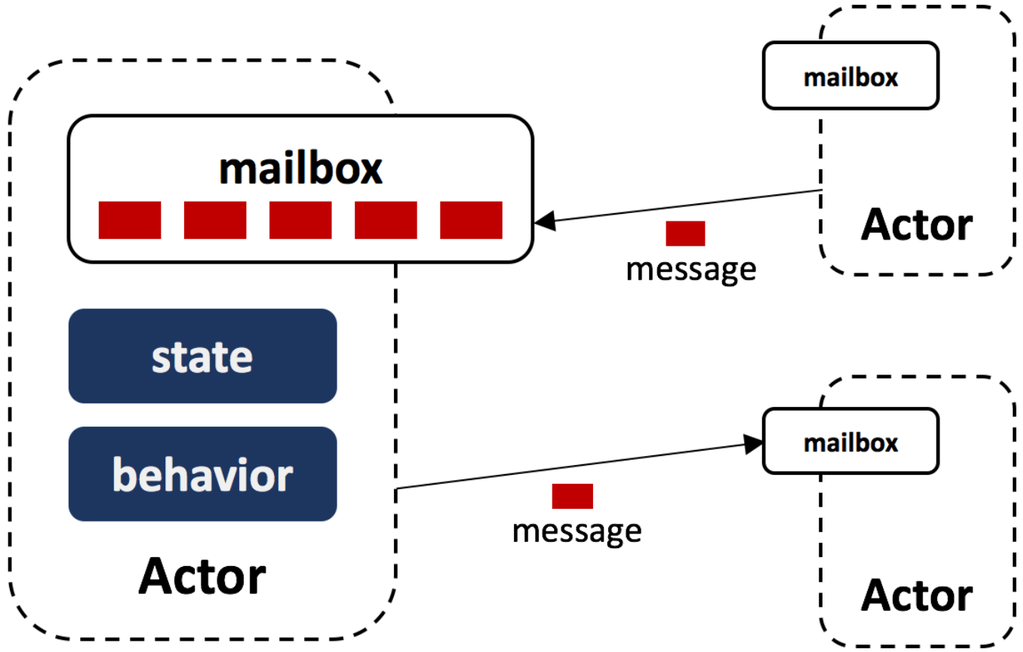


Actor Model

Starting from the seminal work of Hewitt et al., actors were thought as a model for concurrency computing. This theory became the foundation of several programming languages.

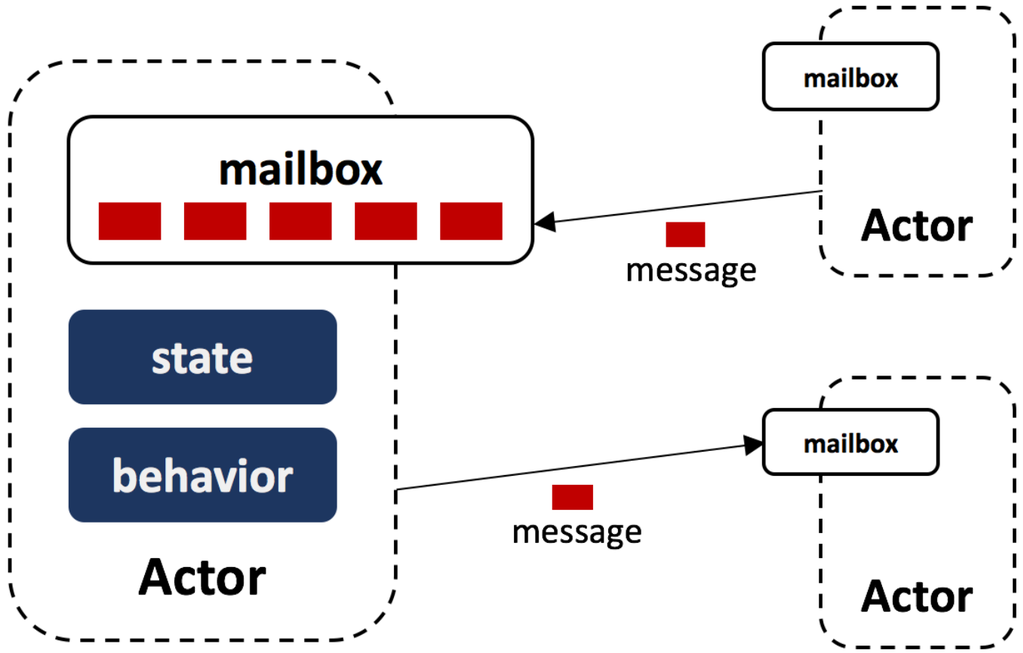


Actors


mailbox
mailbox
Actor
message
state
behavior
Actor
mailbox
message
Actor

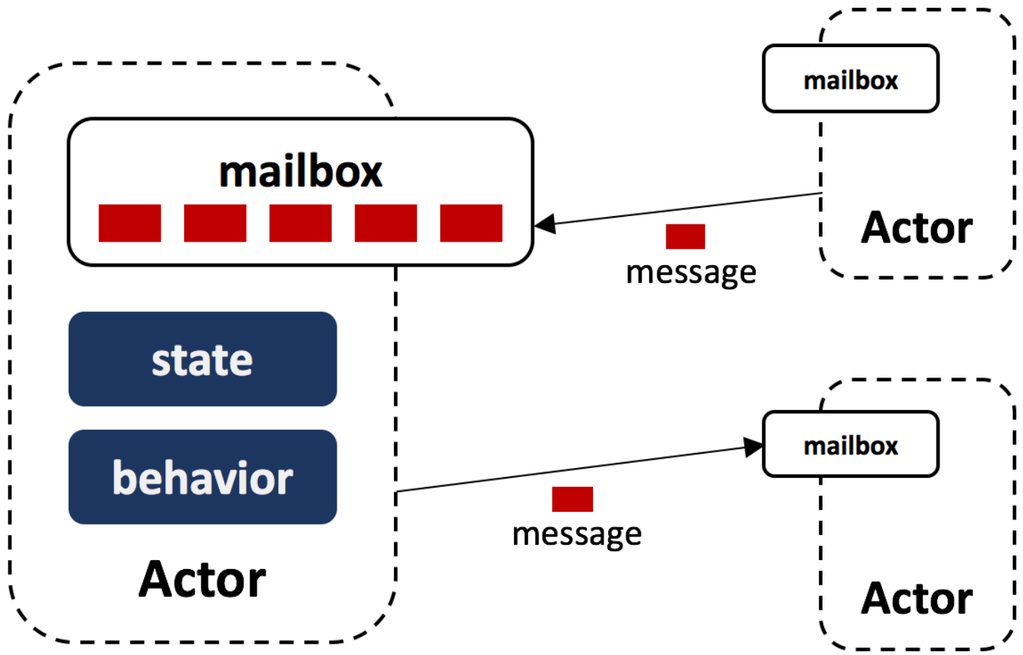
Actors

- Actors are lightweight objects that encapsulate a *state* and a *behavior*.



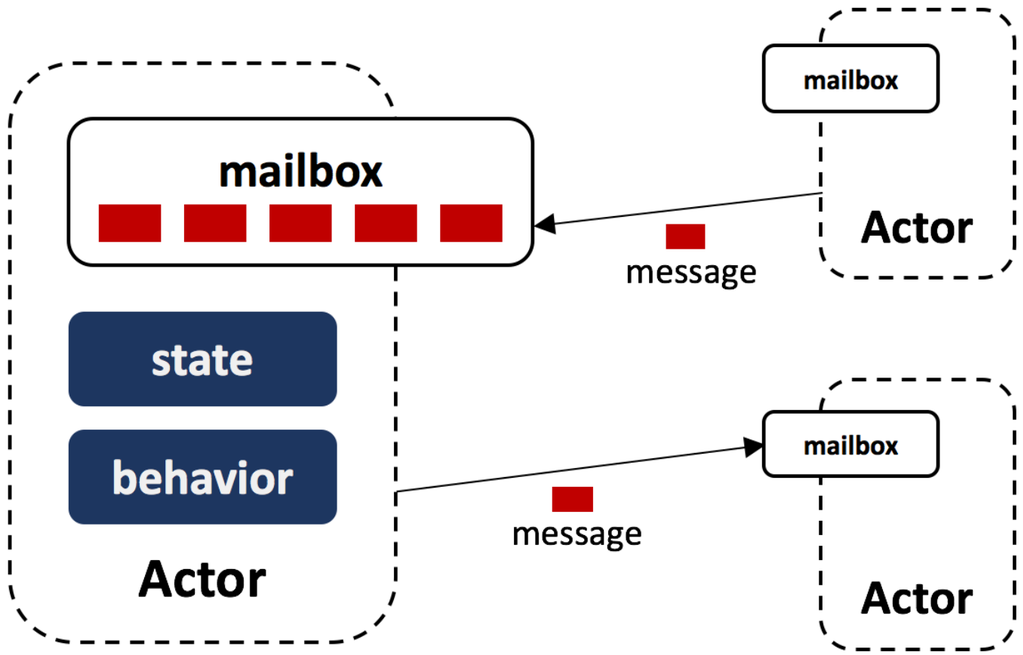
Actors

- Actors are lightweight objects that encapsulate a *state* and a *behavior*.
- They share no mutable state among them, and in fact the only way to communicate is through asynchronous message passing.



Actors

- Actors are lightweight objects that encapsulate a *state* and a *behavior*.
- They share no mutable state among them, and in fact the only way to communicate is through asynchronous message passing.
- To manage the incoming messages, each actor has a mailbox.



Actor Model And Stream Processing Execution

Immutable state, no-sharing and asynchronous processing are common requirements for this Stream Processing systems, e.g., Flink or Storm.

The asynchronous message-passing communication that governs actor interactions is a key feature that allows providing a loose-coupled architecture where blocking operators are avoided.

Indeed, these characteristics are particularly interesting for stream processing systems, especially for those where high scalability and parallel processing of streams are needed.

Actor Model: Partitioning

Partitioning strategies determine the allocation of records between the parallel tasks of two connected logical operators.

Actor Model: Partitioning

Partitioning strategies determine the allocation of records between the parallel tasks of two connected logical operators.

- **Random partitioning**: each output record of a task is shipped to a uniformly random assigned task of a receiving operator.

Actor Model: Partitioning

Partitioning strategies determine the allocation of records between the parallel tasks of two connected logical operators.

- **Random partitioning**: each output record of a task is shipped to a uniformly random assigned task of a receiving operator.
- **Broadcast partitioning**: send records to every parallel task of the next operator.

Actor Model: Partitioning

Partitioning strategies determine the allocation of records between the parallel tasks of two connected logical operators.

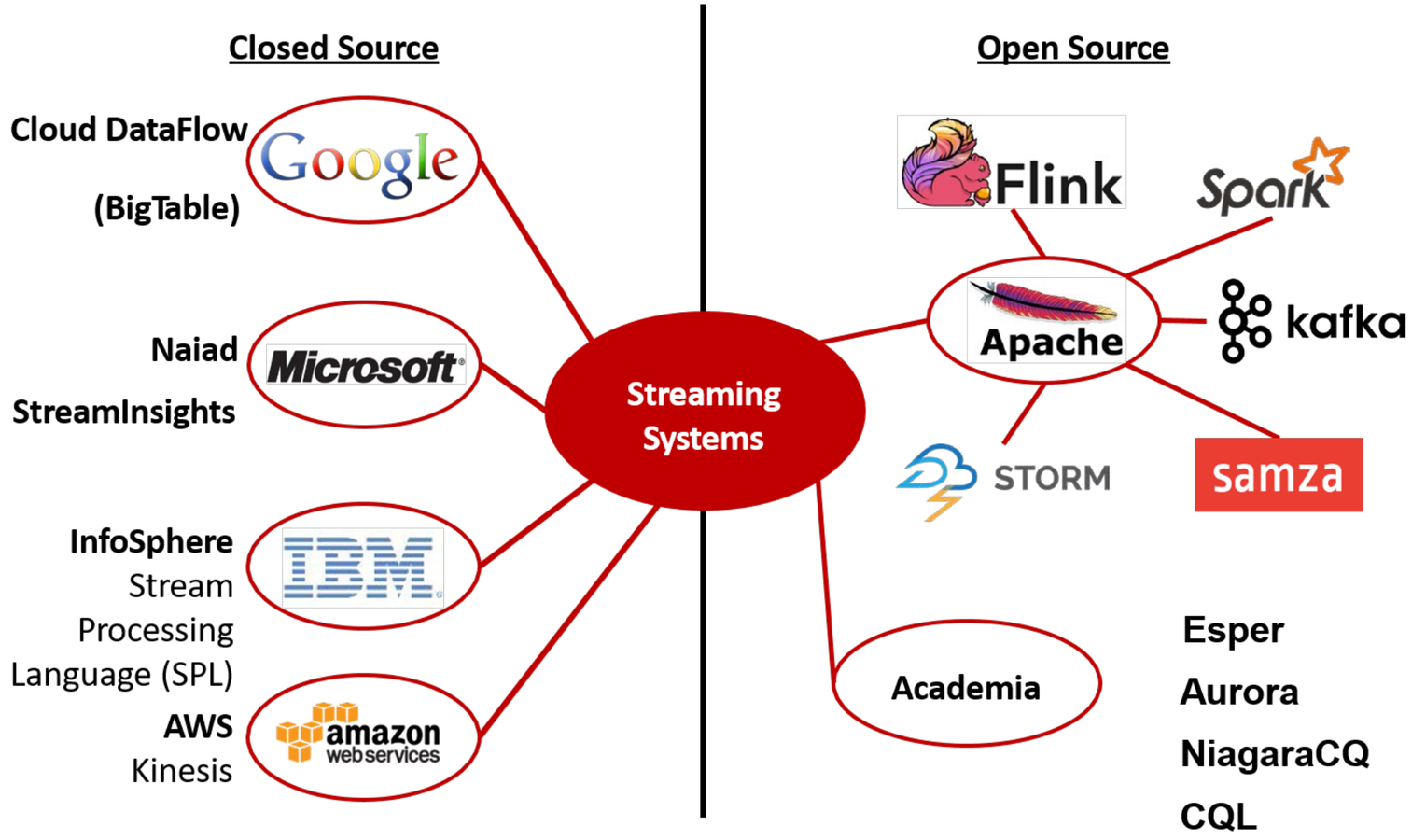
- **Random partitioning**: each output record of a task is shipped to a uniformly random assigned task of a receiving operator.
- **Broadcast partitioning**: send records to every parallel task of the next operator.
- **Partitioning by key**: guarantees that records with the same key (e.g., declared by the user) are sent to the same parallel task of consuming operators

Actor Model: Partitioning

Partitioning strategies determine the allocation of records between the parallel tasks of two connected logical operators.

- **Random partitioning**: each output record of a task is shipped to a uniformly random assigned task of a receiving operator.
- **Broadcast partitioning**: send records to every parallel task of the next operator.
- **Partitioning by key**: guarantees that records with the same key (e.g., declared by the user) are sent to the same parallel task of consuming operators
- **User defined partitioning functions**:(e.g., geo-partitioning or machine learning model selection).

Open-Source Systems Overview



Large-Scale Data Stream Processing on Commodity Clusters

Large-Scale Data Stream Processing on Commodity Clusters

- Some of the first open source SPs for commodity clusters were `Yahoo! S4`[7] and `Twitter Storm`[8].

---

[7] S4

[8] Storm

Large-Scale Data Stream Processing on Commodity Clusters

- Some of the first open source SPs for commodity clusters were `Yahoo! S4`[7] and `Twitter Storm`[8].
- The more systems provide richer semantics and higher-level programming abstractions for data streams in order to simplify the writing of data stream analysis applications. Examples of such systems are `Apache Flink`[9], `Beam`[10] , `Samza`[11], `Spark Streaming`[12], `APEX`[13], and `Kafka Streams`[14].

---

[7] S4

[8] Storm

[9] Flink

[10] Beam

[11] Samza

[12] SparkStreaming

[13] Apex

[14] Kafka Streams


Thank You
Mahalo
Kiitos
Tack
Toda
Grazie
Thanks
Obrigado
Takk
Merci
Gracias

The End