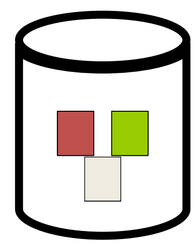
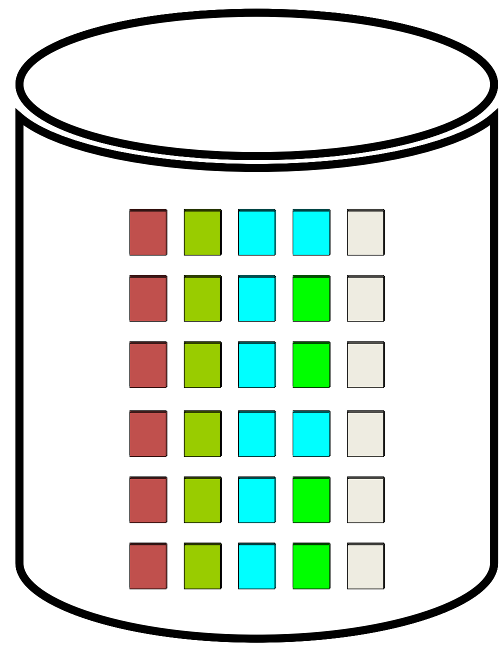
A Brief History of Stream Processing

Data



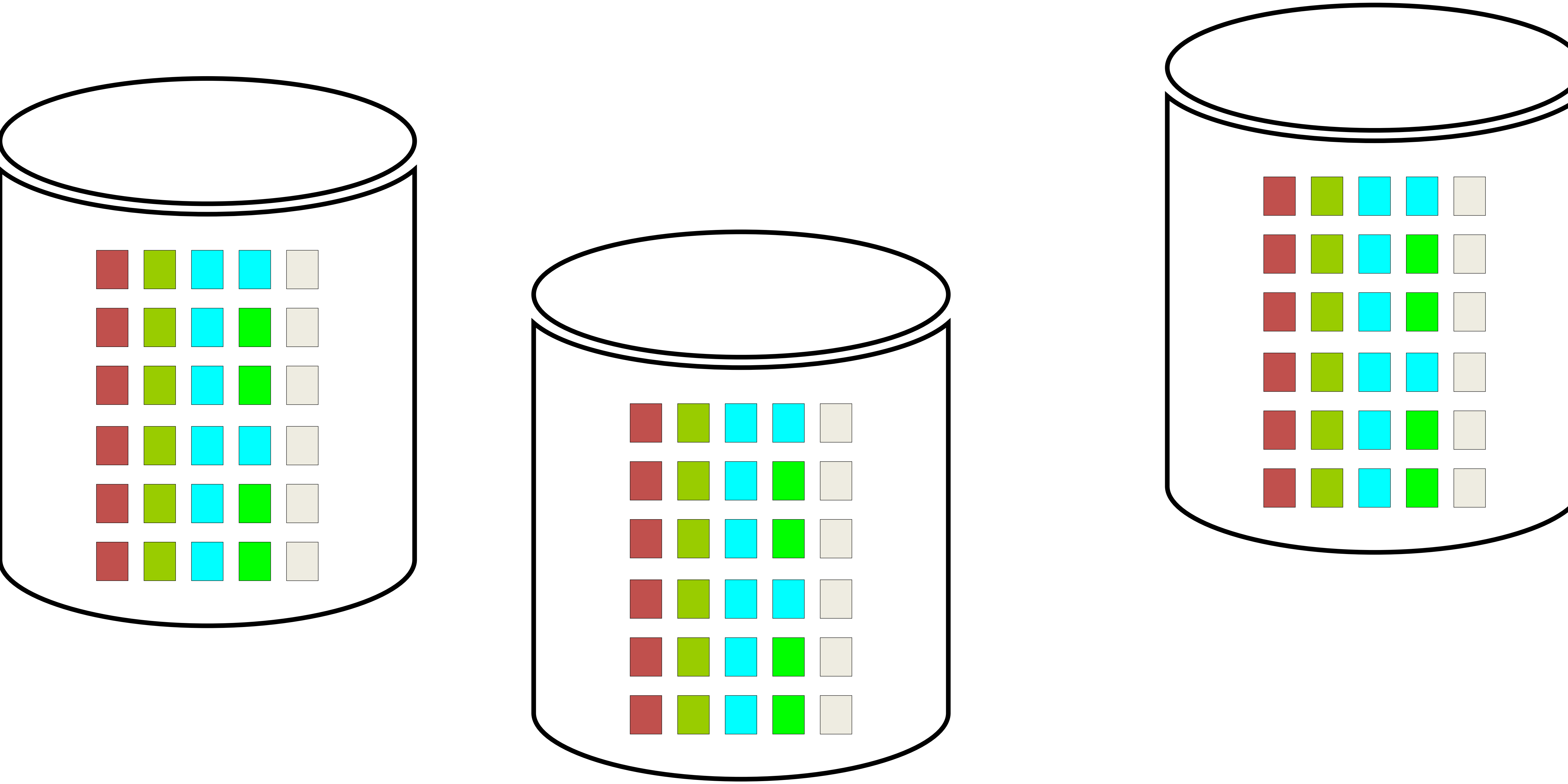
credits to Sara Robinson, Felipe Hoffa

Can Be Big



credits to Sara Robinson, Felipe Hoffa

Very Big

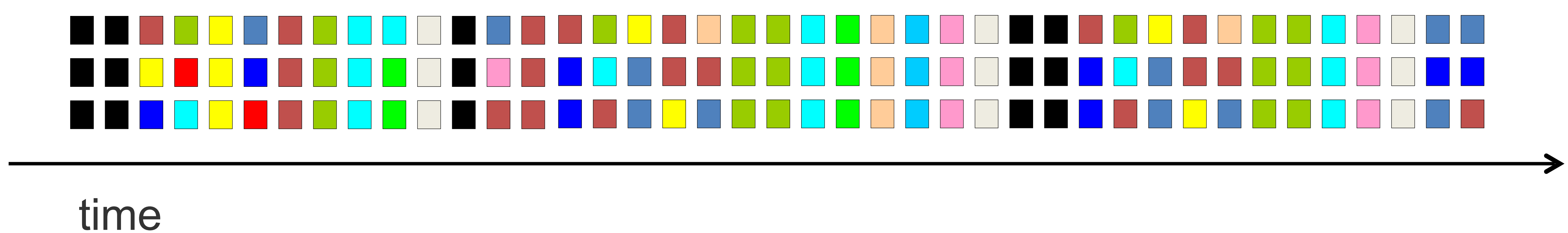


credits to Sara Robinson, Felipe Hoffa

Data Never Sleep!

2018 2019 2020 2021

So big they never end



So What?

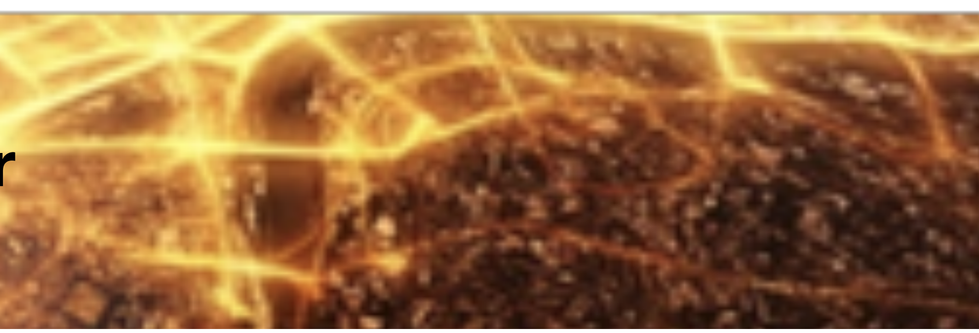
The traditional data processing infrastructures are challenged:

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

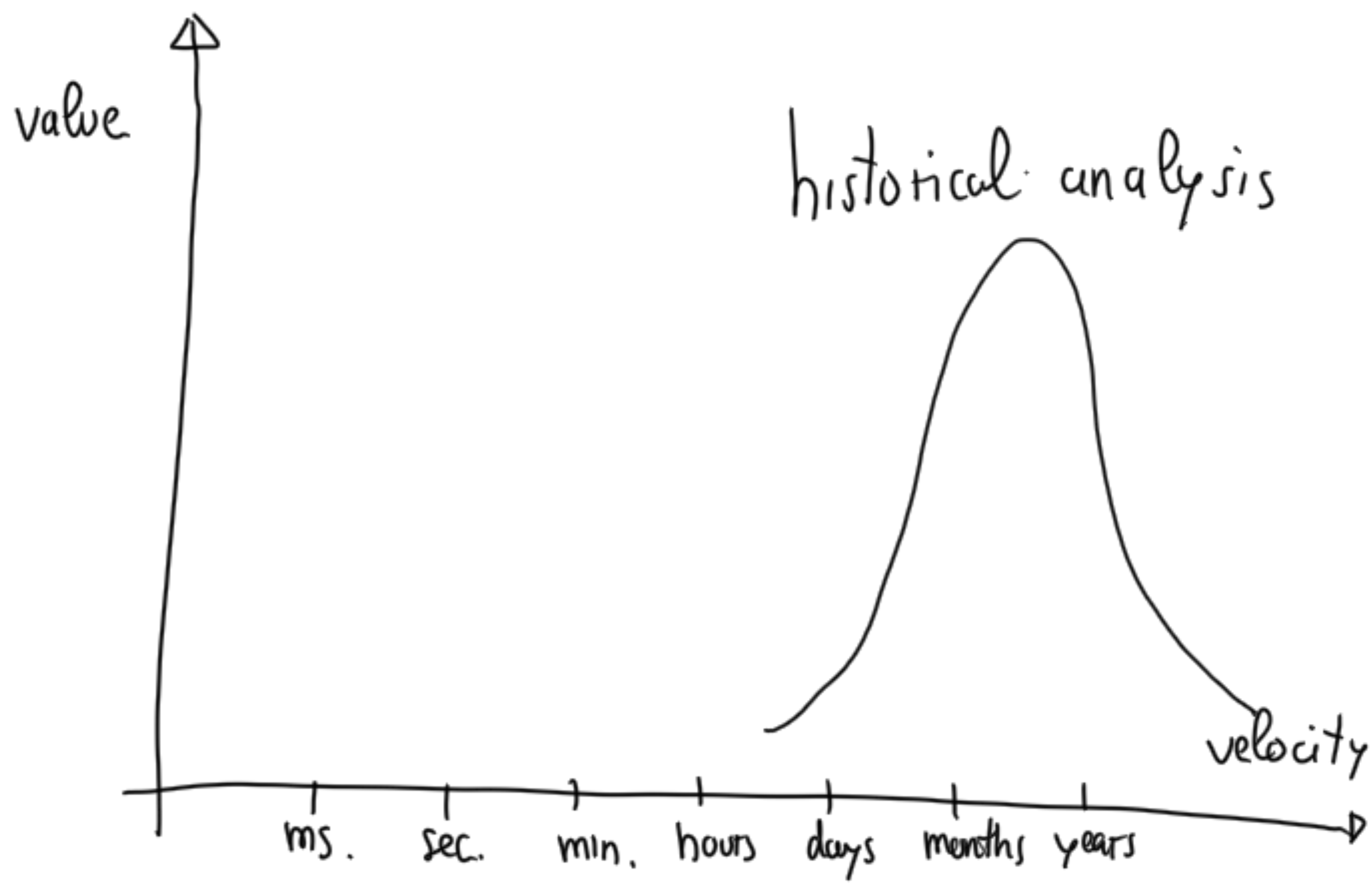
… Excel At Historical Descriptive Analysis

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?



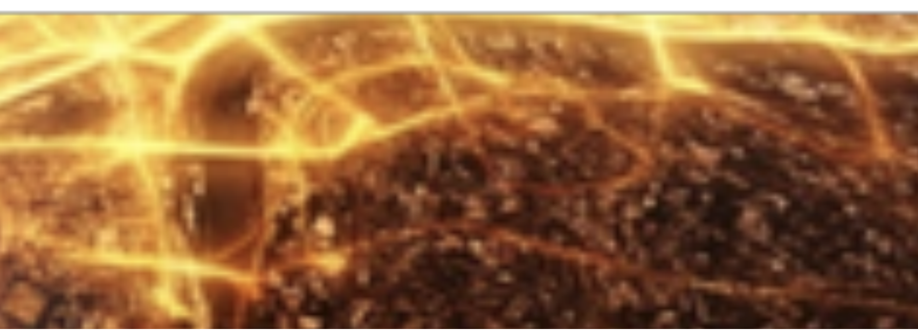
Which content features are correlated to high impact posts?


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

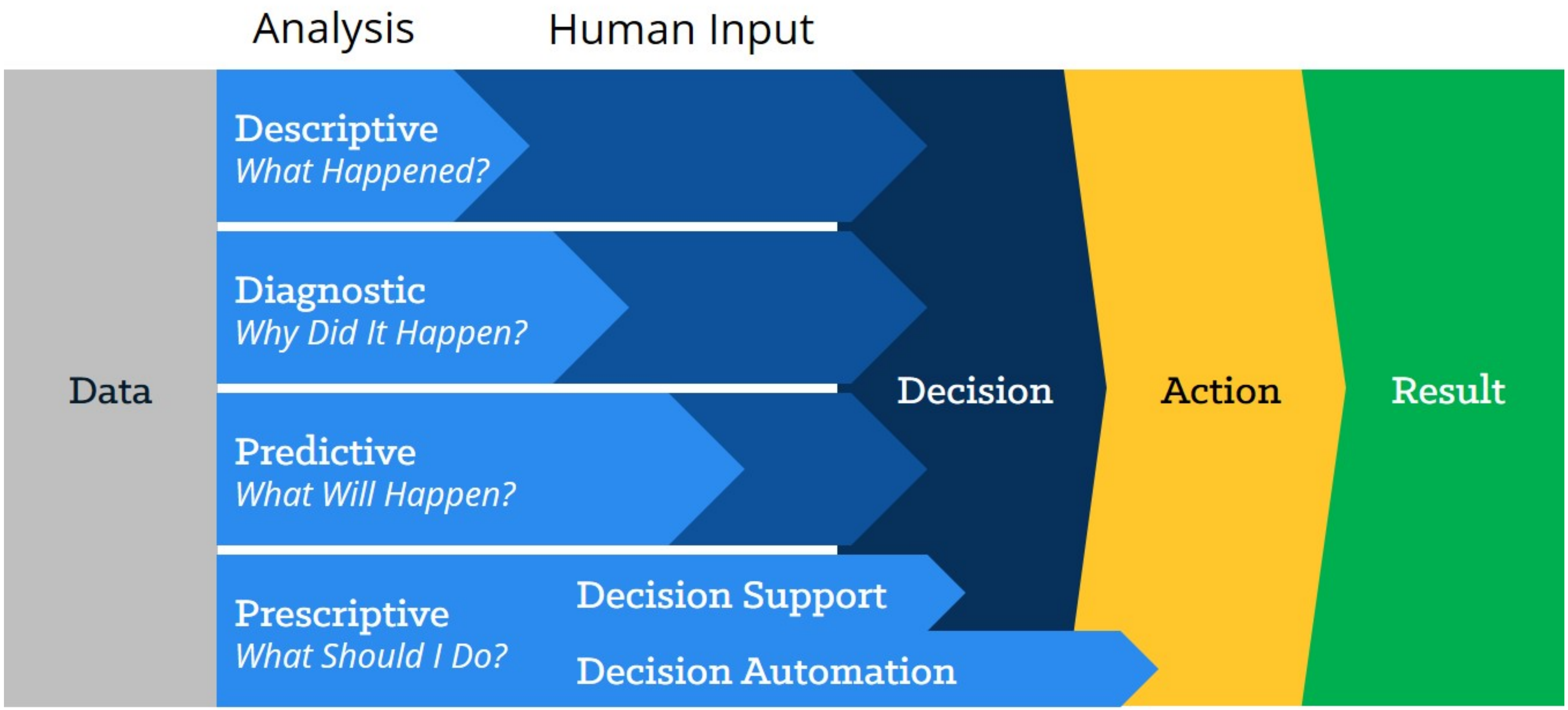
… Struggling With Prescriptive Analysis

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?



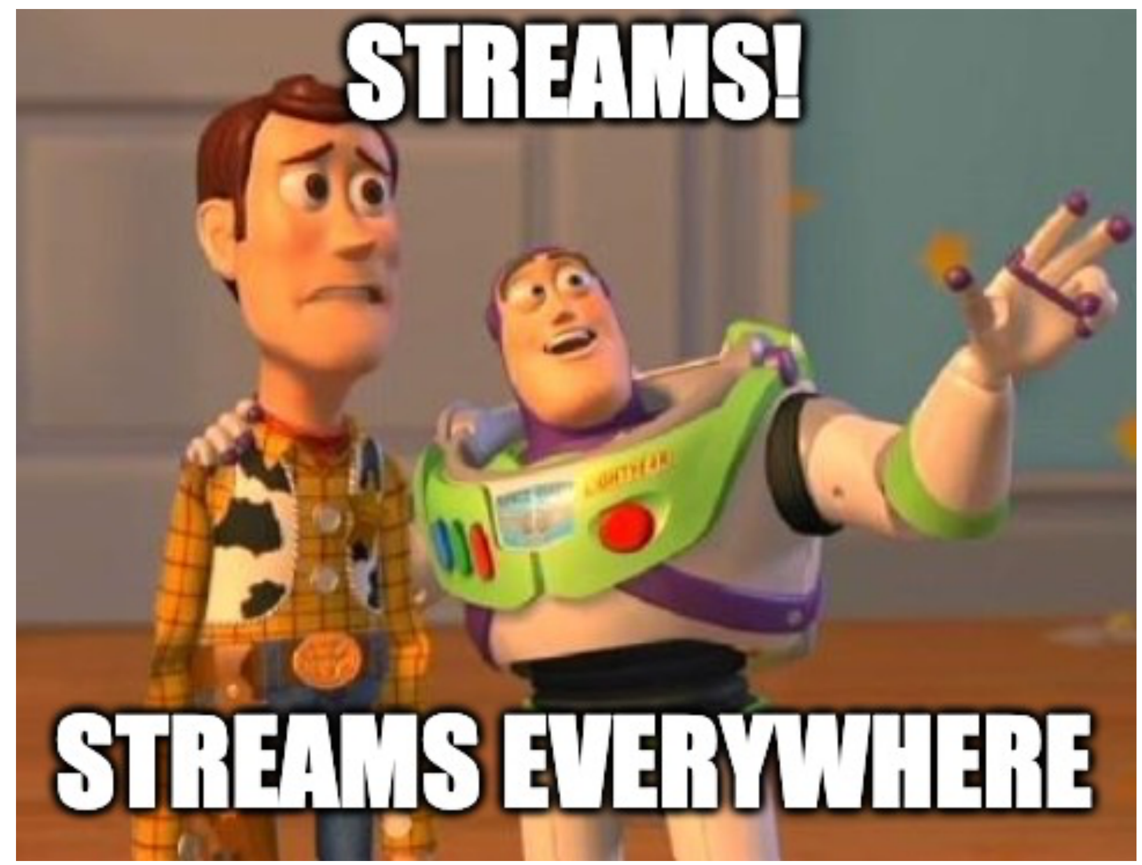
Who is driving the discussion about the top 10 emerging topics?


Analysis
Human Input
Data
Descriptive
What Happened?
Diagnostic
Why Did It Happen?
Predictive
What Will Happen?
Prescriptive
What Should I Do?
Decision Support
Decision Automation
Decision
Action
Result

So What?

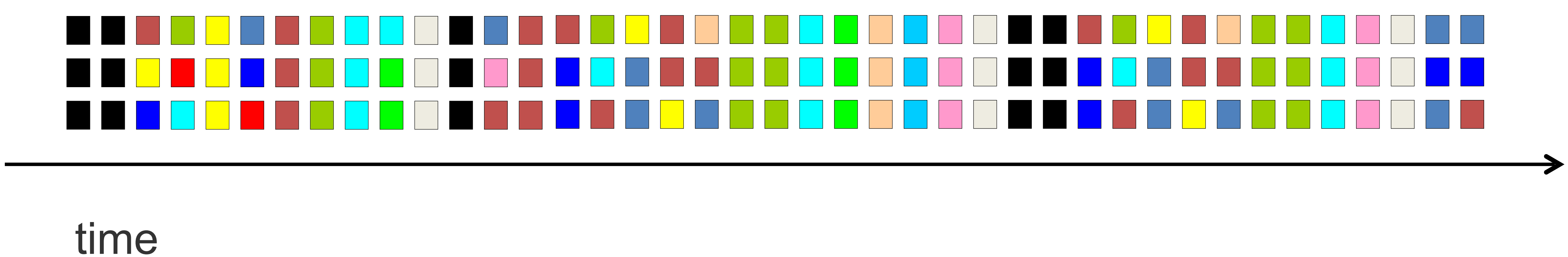
The traditional data processing infrastructures are challenged:

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



What is a Stream?

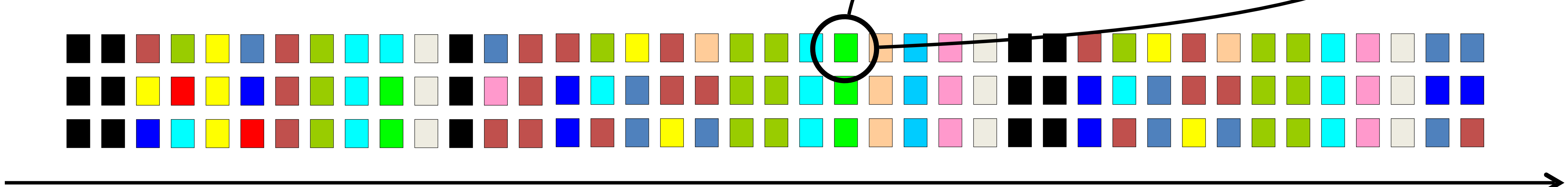
- **Streams**: unbounded partially ordered sequence of data in form of object-timestamp pairs <o,t>, e.g.,
  - o is a data item
  - t is a natural number



What is an Event?

- **Event**: time-based notification of a known fact defined by
  - p a key-value payload
  - τ, a type
  - t, a timestamp
  - d, an optional duration

- payload: 520 - 565 mm
- type: green
- timestamp: t
- duration: 0

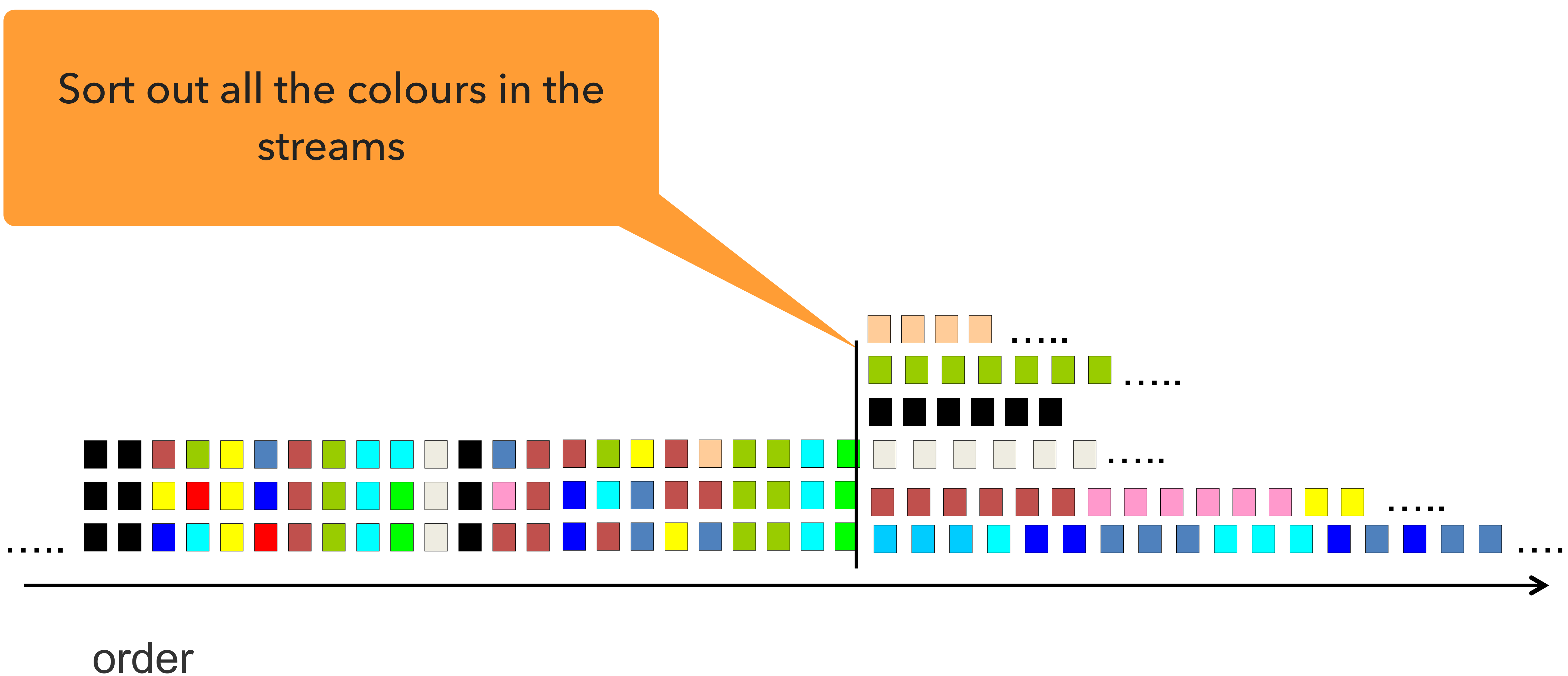


time

How to process a stream?

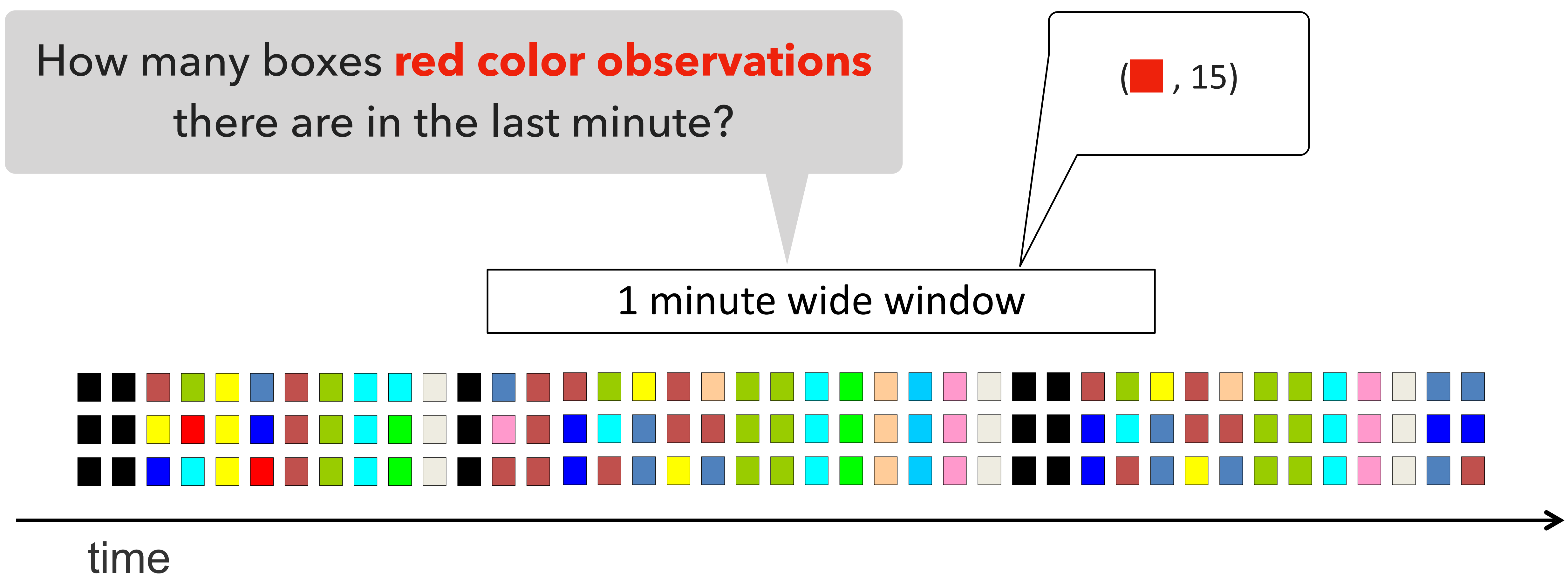
Stream Computing

Continuous Algorithms



Querying Streams

Data Streams Management Systems



Timeline

Precision not Recall

Models and Issues in Data Stream Systems — Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, Jennifer Widom

The 8 Requirements of Real-Time Stream Processing — Michael Stonebraker, Uğur Çetintemel, Stan Zdonik

The CQL continuous query language: semantic foundations and query execution — Arvind Arasu, Shivnath Babu, Jennifer Widom

The CQL continuous query language: semantic foundations and query execution — Arvind Arasu, Shivnath Babu, Jennifer Widom

It's a Streaming World! Reasoning upon Rapidly Changing Information — Emanuele Della Valle, Stefano Ceri, Frank van Harmelen, Dieter Fensel

MOA: Massive Online Analysis — Albert Bifet, Geoff Holmes, Richard Kirkby, Bernhard Pfahringer

Dynamically Scaling ... of ... — Jan Sipke van der Veen, Bram van d...

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

Apache Samza — Martin Kleppmann

Apache Flink™: Stream and Batch Processing in a Single Engine — Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, Kostas Tzoumas

... Sides of the Same C... — Guozhang Wang, Johann-Christoph Freytag

Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark — Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, Matei Zaharia

2002 2005 2006 2007 2008 2010 2014 2015 2018

The Vision

The 8 Requirements of Real-Time Stream Processing

Michael Stonebraker
Computer Science and Artificial Intelligence Laboratory, M.I.T., and StreamBase Systems, Inc.
stonebraker@csail.mit.edu

Uğur Çetintemel
Department of Computer Science, Brown University, and StreamBase Systems, Inc.
ugur@cs.brown.edu

Stan Zdonik
Department of Computer Science, Brown University, and StreamBase Systems, Inc.
sbz@cs.brown.edu

ABSTRACT

Applications that require real-time processing of high-volume data steams are pushing the limits of traditional data processing infrastructures. These stream-based applications include market feed processing and electronic trading on Wall Street, network and infrastructure monitoring, fraud detection, and command and control in military environments. Furthermore, as the "sea change" caused by cheap micro-sensor technology takes hold, we expect to see everything of material significance on the planet get "sensor-tagged" and report its state or location in real time. This sensorization of the real world will lead to a "green field" of novel monitoring and control applications with high-volume and low-latency processing requirements.

Recently, several technologies have emerged—including off-the-shelf stream processing engines—specifically to address the challenges of processing high-volume, real-time data without requiring the use of custom code. At the same time, some existing software technologies, such as main memory DBMSs and rule engines, are also being "repurposed" by marketing departments to address these applications.

In this paper, we outline eight requirements that a system software should meet to excel at a variety of real-time stream processing applications. Our goal is to provide high-level guidance to information technologists so that they will know what to look for when evaluation alternative stream processing solutions. As such,

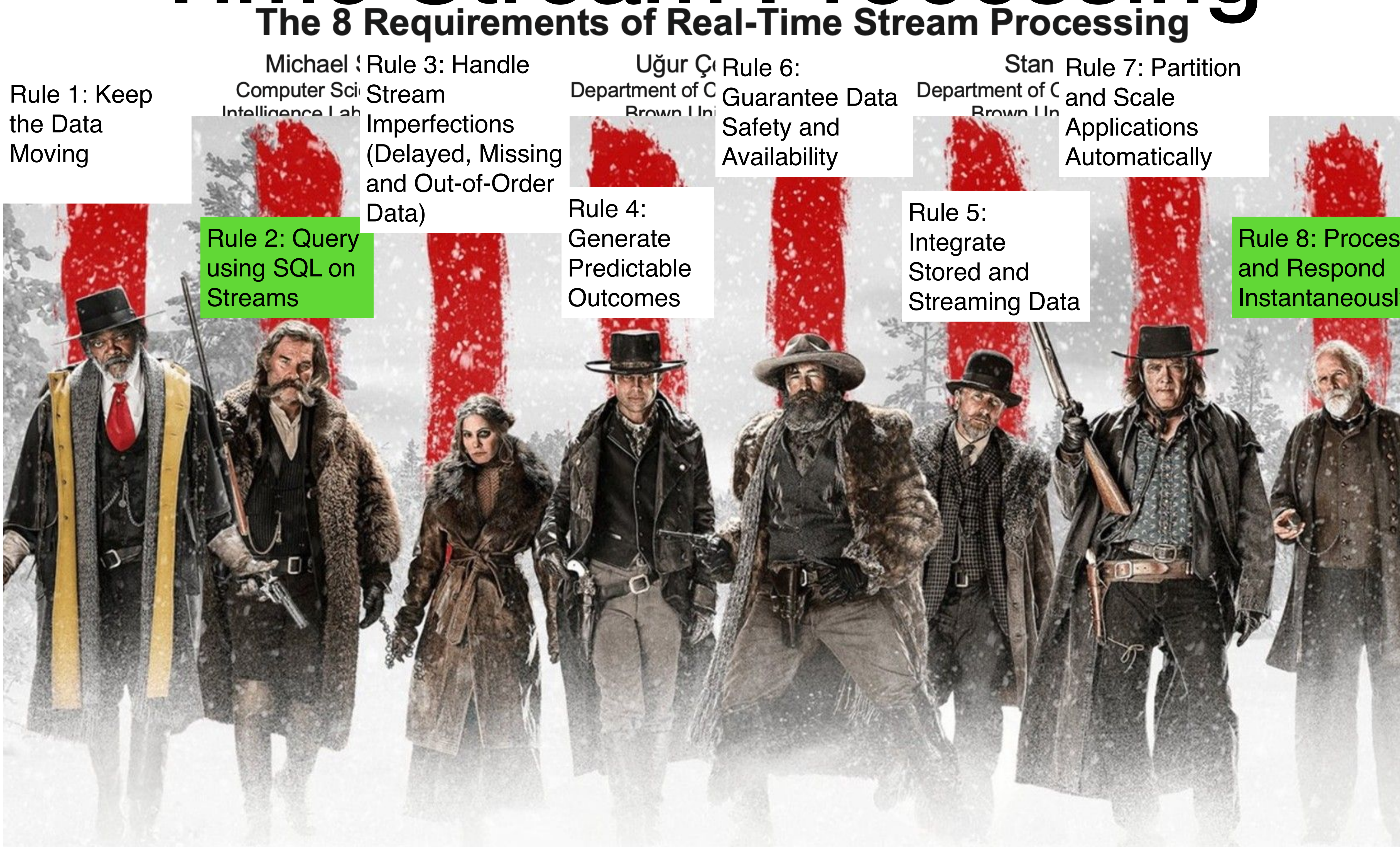
Similar requirements are present in monitoring computer networks for denial of service and other kinds of security attacks. Real-time fraud detection in diverse areas from financial services networks to cell phone networks exhibits similar characteristics. In time, process control and automation of industrial facilities, ranging from oil refineries to corn flakes factories, will also move to such "firehose" data volumes and sub-second latency requirements.

There is a "sea change" arising from the advances in micro-sensor technologies. Although RFID has gotten the most press recently, there are a variety of other technologies with various price points, capabilities, and footprints (e.g., mote [1] and Lojack [2]). Over time, this sea change will cause everything of material significance to be sensor-tagged to report its location and/or state in real time.

Military has been an early driver and adopter of wireless sensor network technologies. For example, the US Army has been investigating putting vital-signs monitors on all soldiers. In addition, there is already a GPS system in many military vehicles, but it is not connected yet into a closed-loop system. Using this technology, the army would like to monitor the position of all vehicles and determine, in real time, if they are off course.

Other sensor-based monitoring applications will come over time in non-military domains. Tagging will be applied to customers at amusement parks for ride management and prevention of lost

8 Requirements of Real-Time Stream Processing



Rule 1: Keep the Data Moving

Rule 2: Query using SQL on Streams

Rule 3: Handle Stream Imperfections (Delayed, Missing and Out-of-Order Data)

Rule 4: Generate Predictable Outcomes

Rule 5: Integrate Stored and Streaming Data

Rule 6: Guarantee Data Safety and Availability

Rule 7: Partition and Scale Applications Automatically

Rule 8: Process and Respond Instantaneously

Timeline

Precision not Recall

Models and Issues in Data Stream Systems

The CQL continuous query language: semantic foundations and query execution

The 8 Requirements of Real-Time Stream Processing

The CQL continuous query language: semantic foundations and query execution

It's a Streaming World! Reasoning upon Rapidly Changing Information

MOA: Massive Online Analysis

Dynamically Scaling

Apache Samza

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

Apache Flink™: Stream and Batch Processing in a Single Engine

Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark

2002 2005 2006 2007 2008 2010 2014 2015 2018

The VLDB Journal (2006) 15(2): 121–142
DOI 10.1007/s00778-004-0147-z

REGULAR PAPER

Arvind Arasu · Shivnath Babu · Jennifer Widom

The CQL continuous query language: semantic foundations and query execution

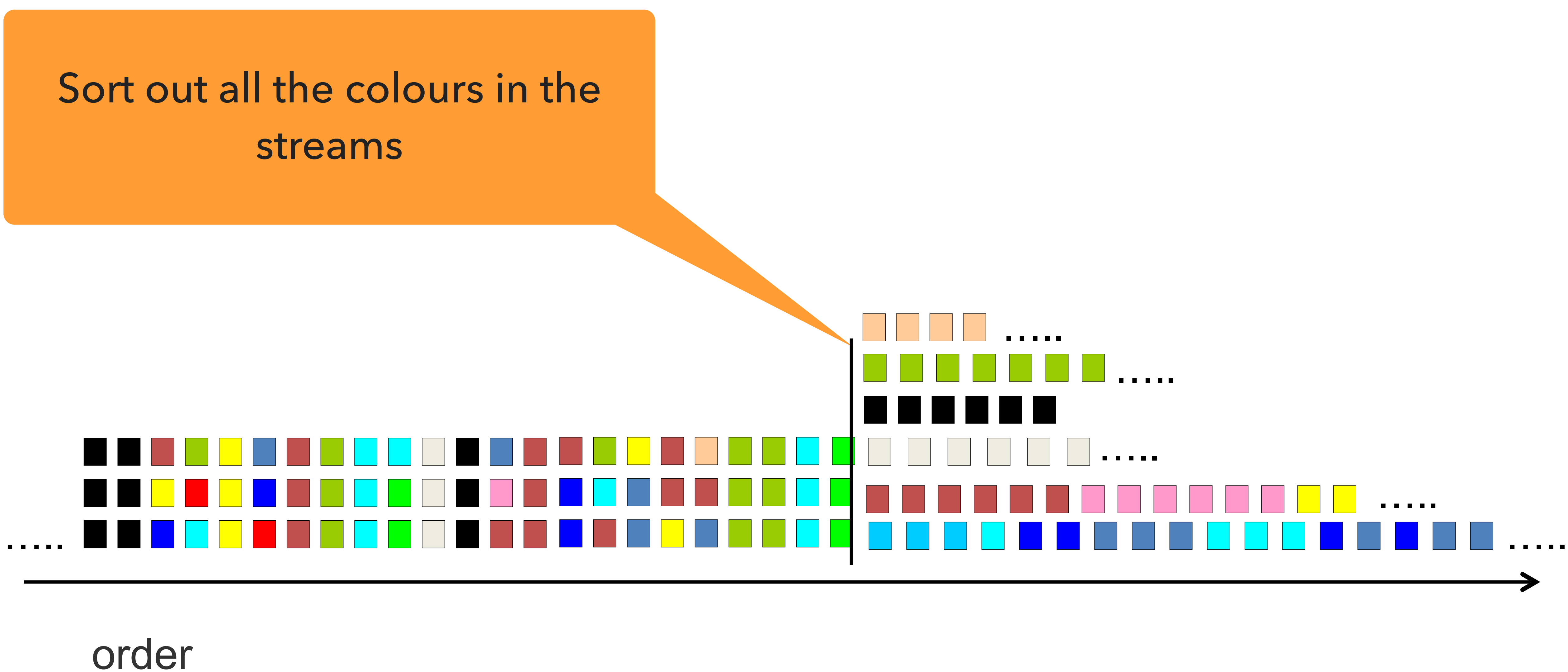
Received: 7 June 2004 / Accepted: 22 November 2004 / Published online: 22 July 2005
© Springer-Verlag 2005

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 queries. Examples throughout the paper are drawn from the *Linear Road* benchmark recently proposed for DSMSs. We also curate a public repository of data stream applications that includes a wide variety of queries expressed in CQL. The relative ease of capturing these applications in CQL is

[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.[1]

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 queries over complete stream histories, indeed this approach is nearly sufficient. However, as queries get more complex – when we add aggregation, subqueries, windowing constructs, relations mixed with streams, etc. – the situation becomes much murkier. Consider the following simple query:

Stream Computing

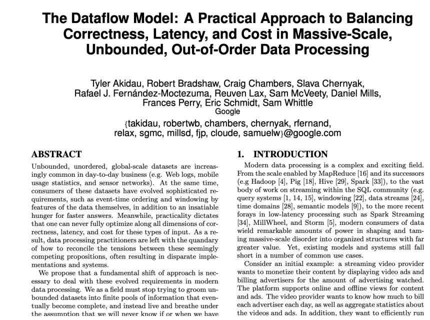


Continuous Querying

How many boxes **red color observations** there are in the last minute?

(■, 15)

CQL in 3 Slides



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, cloude, 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 scale enabled by MapReduce [16] and its successors (e.g Hadoop [4], Pig [18], Hive [29], Spark [33]), to the vast body of work on streaming within the SQL community (e.g. query systems [1, 14, 15], windowing [22], data streams [24], time domains [28], semantic models [9]), to the more recent forays in low-latency processing such as Spark Streaming [34], MillWheel, and Storm [5], modern consumers of data wield 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.

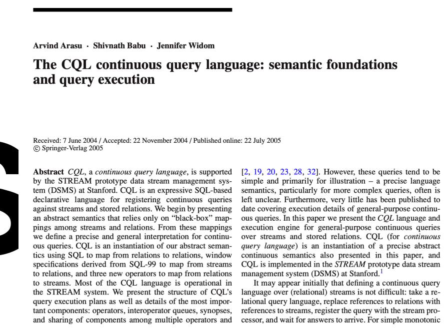
Consider an initial example: a streaming video provider wants to monetize their content by displaying video ads and billing advertisers for the amount of advertising watched. The platform supports online and offline views for content and ads. The video provider wants to know how much to bill each advertiser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

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 set of tuples (d_1, d_2, ..., d_n), where each element d_j is a member of D_j, a data domain[1].

[1] a Data Domain refers to all the values which a data element may contain.

CQL in ~~3~~ 4 Slides



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
© Springer-Verlag 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.[1]

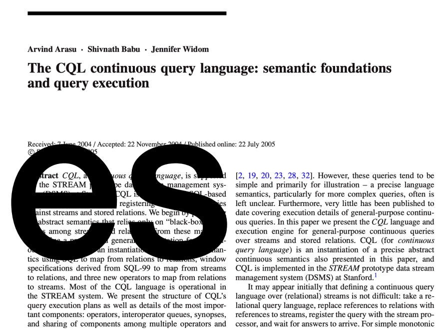
It may appear initially that defining a continuous query language over (relational) streams is not difficult: take a relational query language, replace references to relations with references to streams, register the query with the stream processor, and wait for answers to arrive. For simple monotonic

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 set of tuples (d_1, d_2, ..., d_n), where each element d_j is a member of D_j, a data domain[1].~~

[1] *a Data Domain refers to all the values which a data element may contain.*

Ok, CQL in ~~4~~ 5 Slides


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

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.[1]

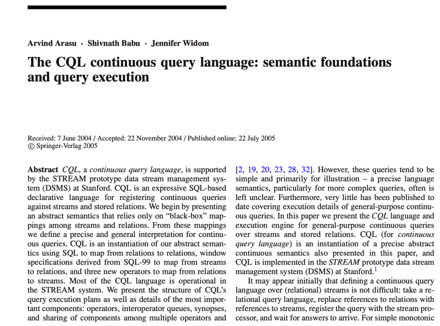
It may appear initially that defining a continuous query language over (relational) streams is not difficult: take a relational query language, replace references to relations with references to streams, register the query with the stream processor, and wait for answers to arrive. For simple monotonic

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**.

[1] *a Data Domain refers to all the values which a data element may contain.*

CQL in 5 Slides



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
© Springer-Verlag 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.[1]

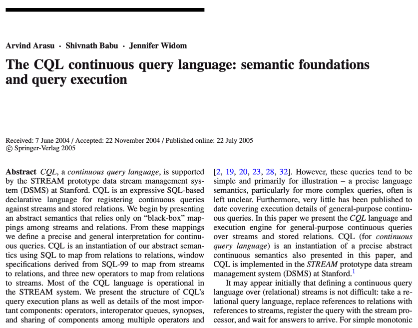
It may appear initially that defining a continuous query language over (relational) streams is not difficult: take a relational query language, replace references to relations with references to streams, register the query with the stream processor, and wait for answers to arrive. For simple monotonic

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**.

[1] *a Data Domain refers to all the values which a data element may contain.*

CQL in 5 Slides



Arvind Arasu · Shivnath Babu · Jennifer Widom

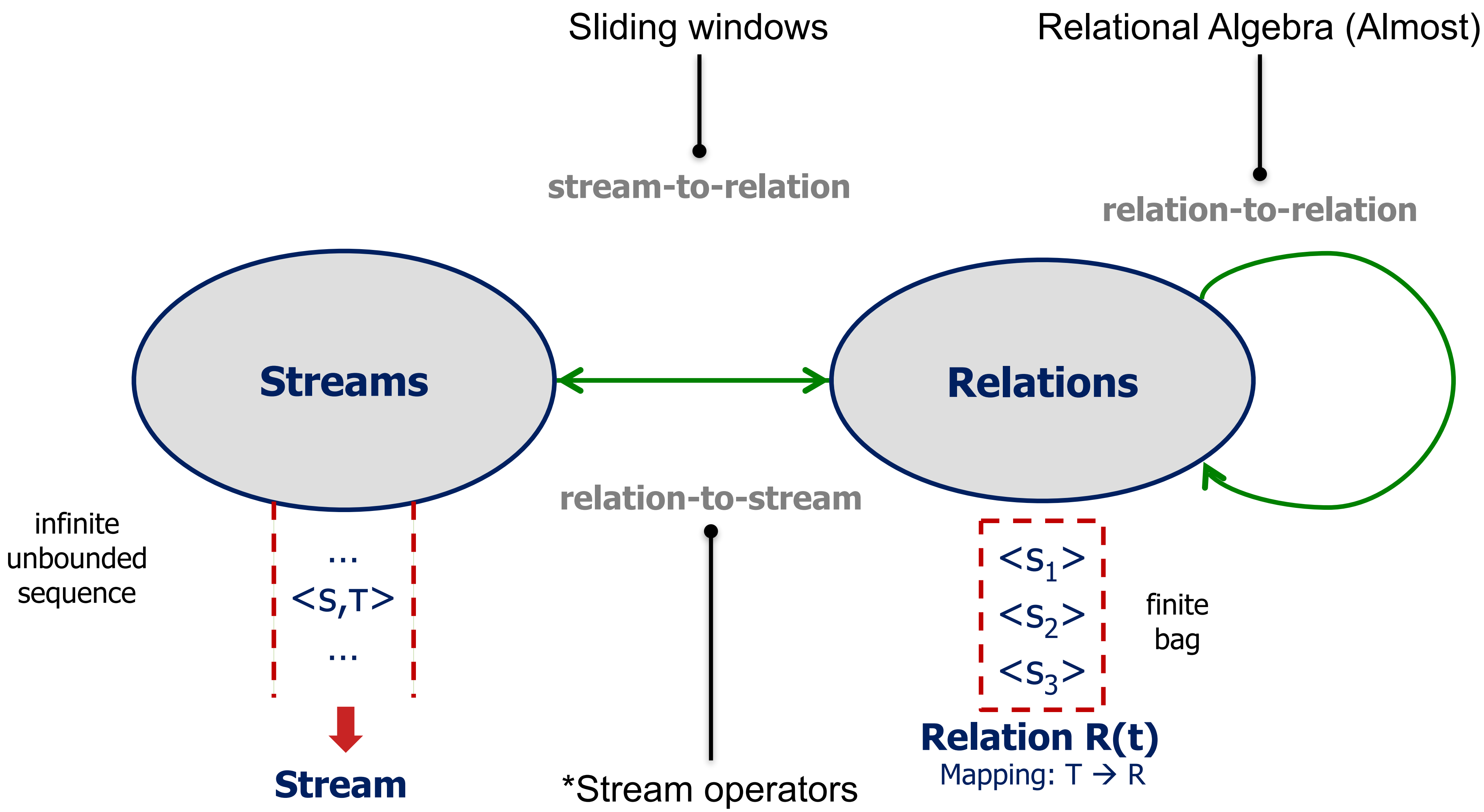
The CQL continuous query language: semantic foundations and query execution

Received: 7 June 2004 / Accepted: 22 November 2004 / Published online: 22 July 2005
© Springer-Verlag 2005

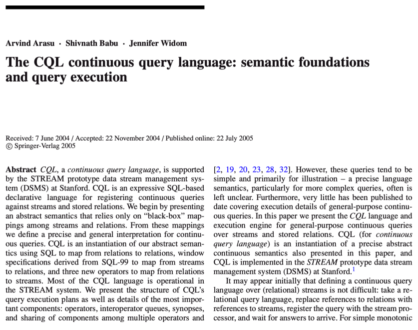
Abstract *CQL*, a *continuous query language*, is supported by the STREAM prototype data stream management system (DSMS) at Stanford. CQL is an expressive SQL-based declarative language for registering continuous queries against streams and stored relations. We begin by presenting an abstract semantics 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.[1]

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



CQL in ~~5~~ 6 Slides



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
© Springer-Verlag 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.[1]

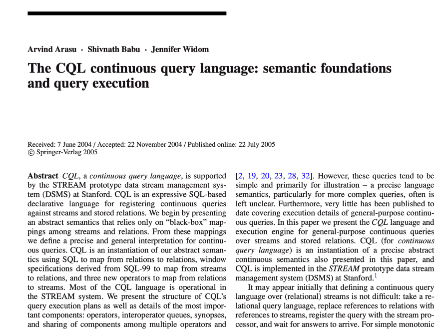
It may appear initially that defining a continuous query language over (relational) streams is not difficult: take a relational query language, replace references to relations with references to streams, register the query with the stream processor, and wait for answers to arrive. For simple monotonic

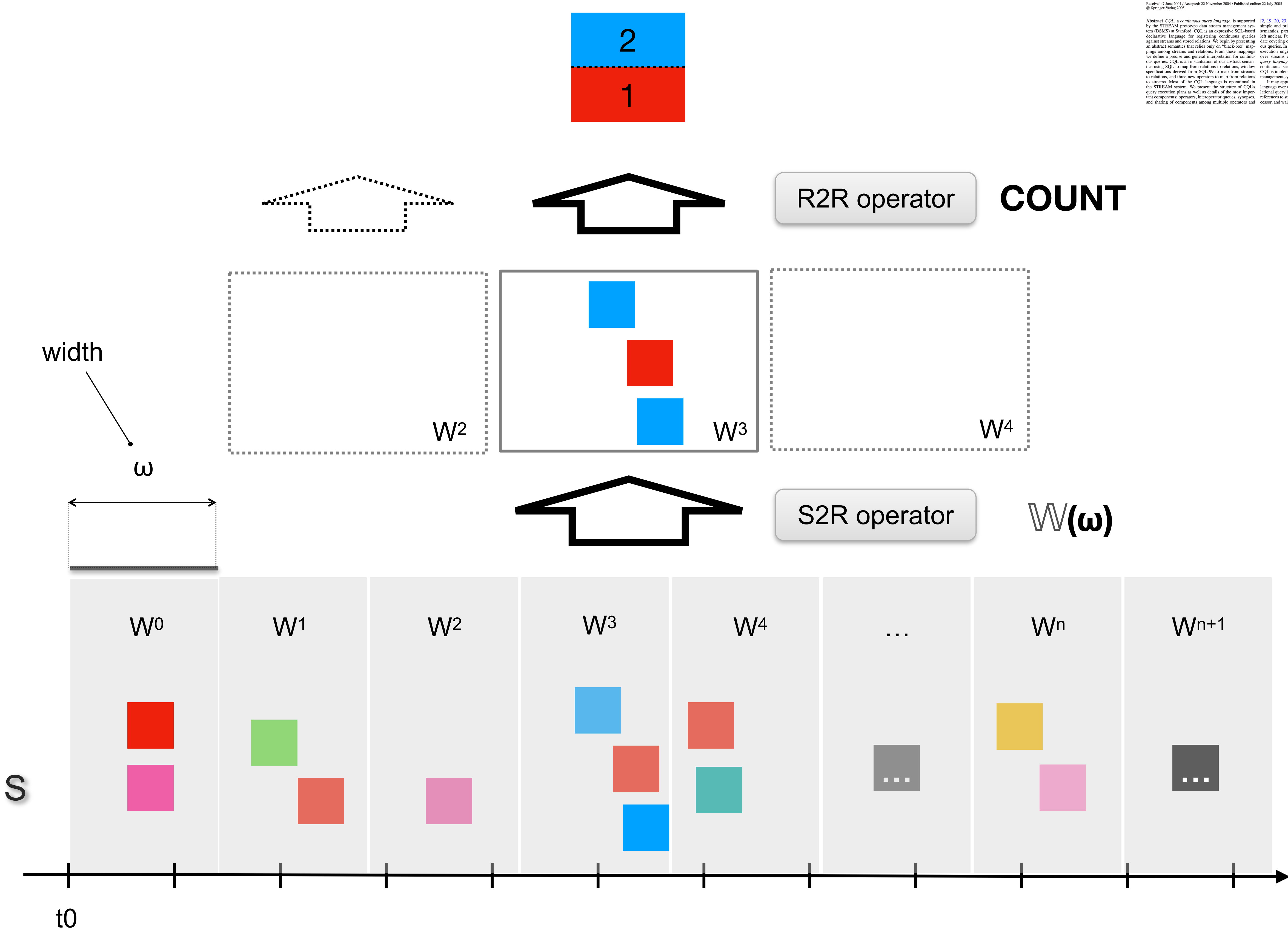
Stream-to-Relation Operators:

- **Sliding Window**:
  FROM S [RANGE 5 Minutes]
- **Parametric Sliding Windows:**
  FROM S [RANGE 5 Minutes Slide 1 Min]
- **Partitioned Windows**:
  FROM S [PARTITIONED BY $A_1..A_n$ ROW m]

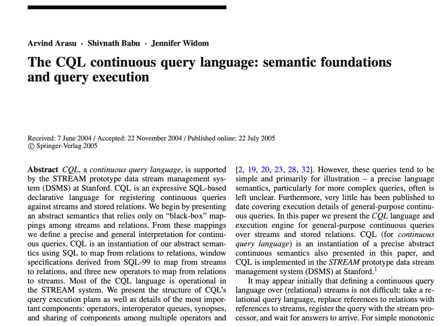
[1] *a Data Domain refers to all the values which a data element may contain.*

CQL in 6 Slides


Arvind Arasu · Shivnath Babu · Jennifer Widom
The CQL continuous query language: semantic foundations and query execution



CQL in 6 Slides



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
© Springer-Verlag 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.[1]

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

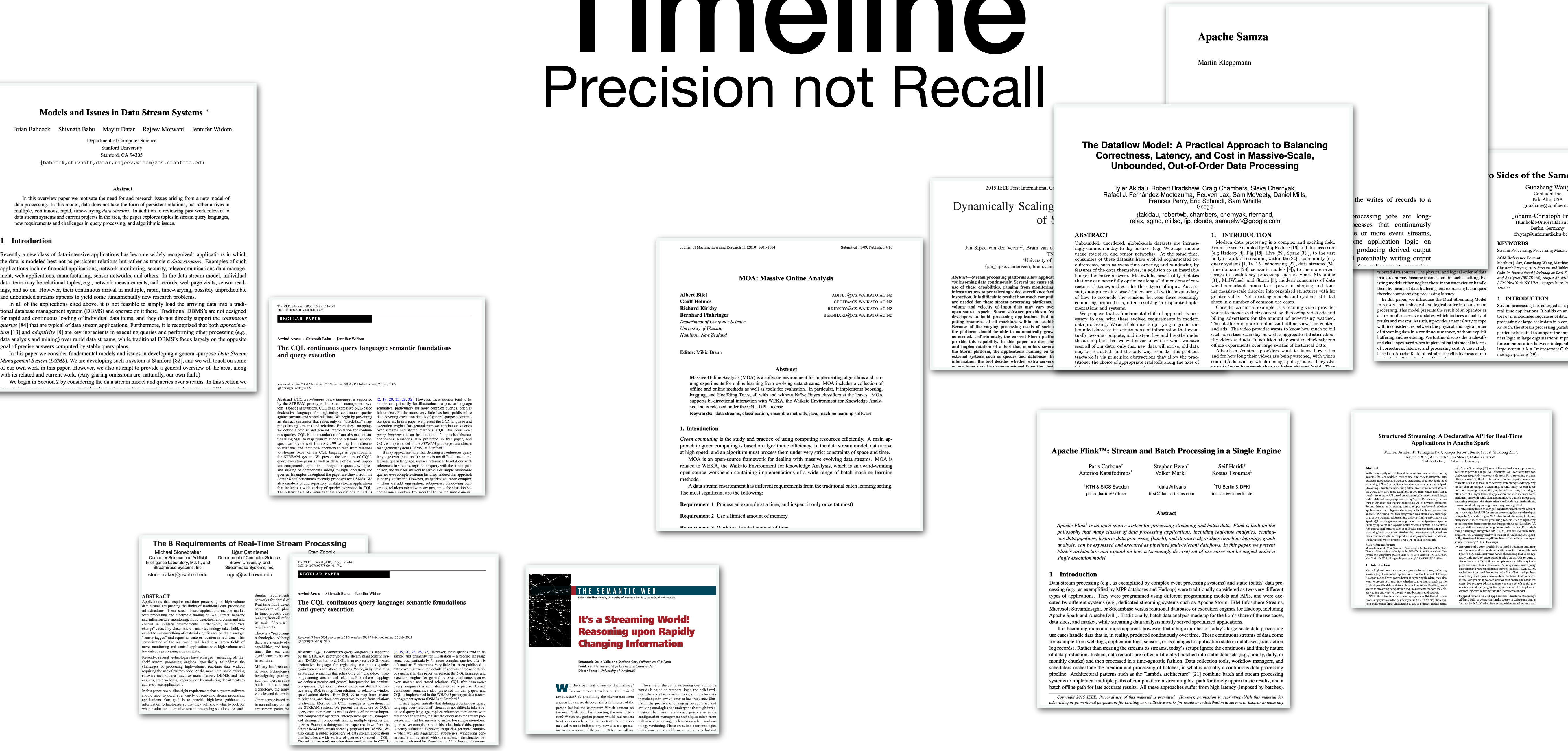
Relation-to-Stream Operators:

- **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

[1] a Data Domain refers to all the values which a data element may contain.

Timeline

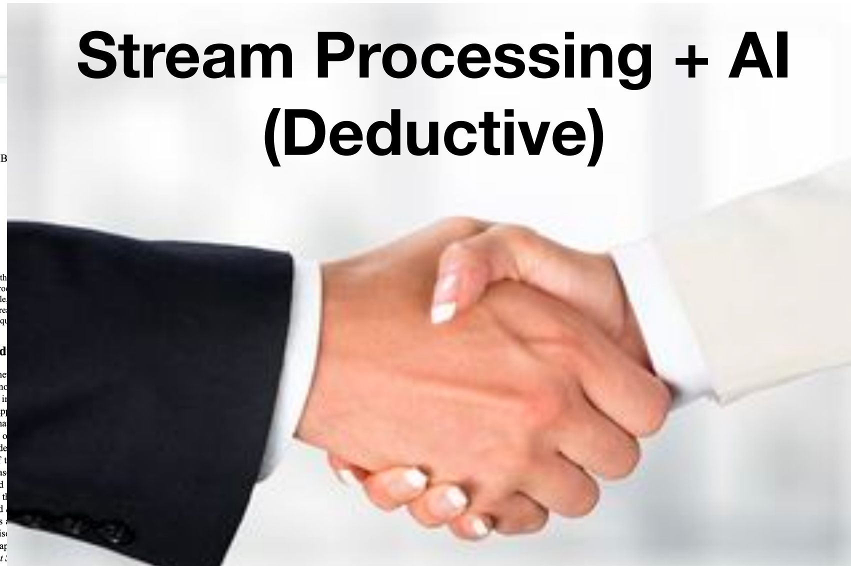
Precision not Recall



2002 2005 2006 2007 2008 2010 2014 2015 2018

Timeline

Precision not Recall



The 8 Requirements of Real-Time Stream Processing

The CQL continuous query language: semantic foundations and query execution

It's a Streaming World! Reasoning upon Rapidly Changing Information

MOA: Massive Online Analysis

Dynamically Scaling

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

Apache Samza

2002 2005 2006 2007 2008 2010 2014 2015 2018

Timeline

Precision not Recall

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, cloude, 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 seen all of our data, only that new data will arrive, old data may be retracted, and the only way to make this problem tractable is via principled abstractions that allow the practitioner the choice of appropriate tradeoffs along the axes of interest: correctness, latency, and cost.

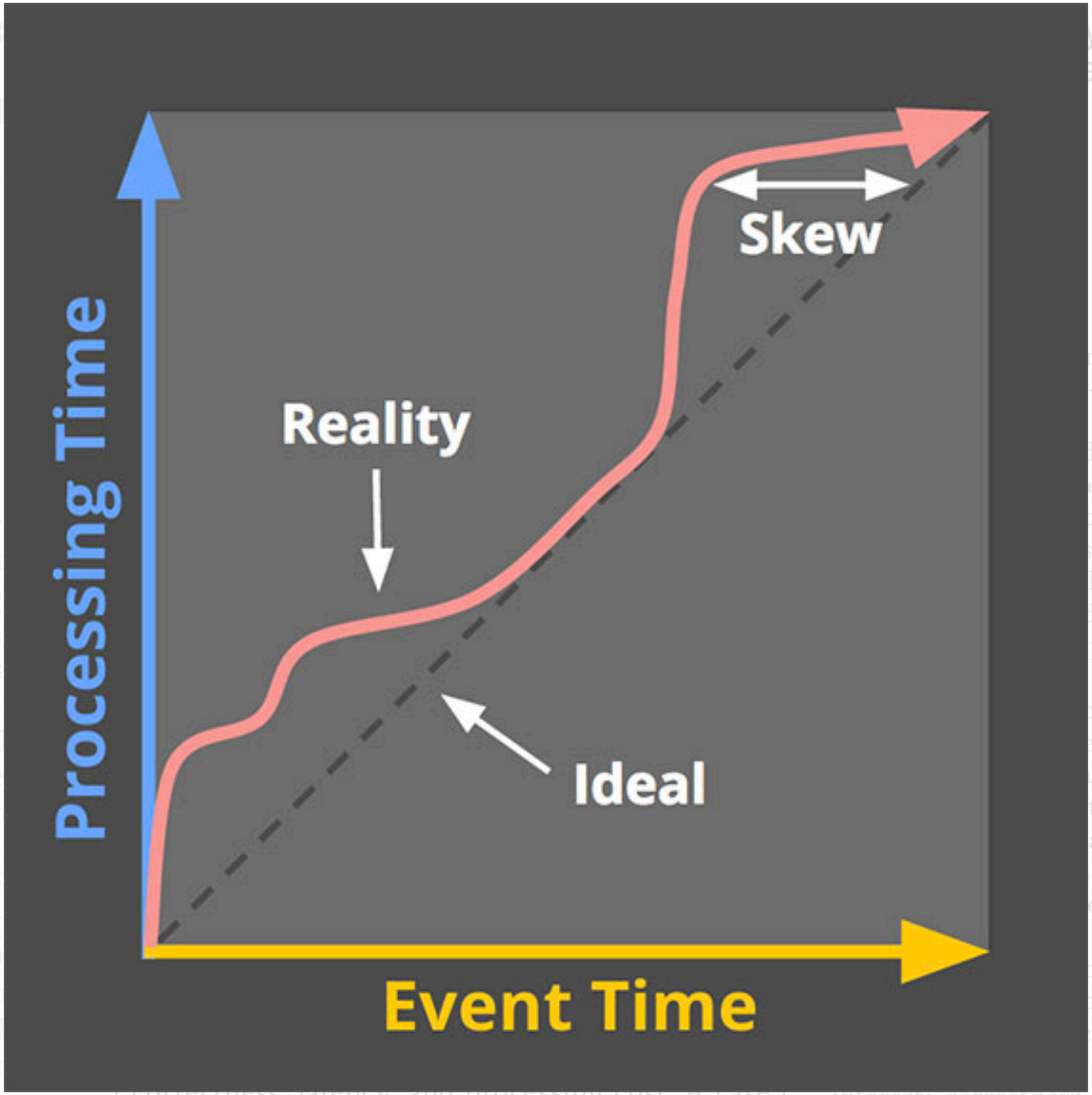
1. INTRODUCTION

Modern data processing is a complex and exciting field. From the scale enabled by MapReduce [16] and its successors (e.g Hadoop [4], Pig [18], Hive [29], Spark [33]), to the vast body of work on streaming within the SQL community (e.g. query systems [1, 14, 15], windowing [22], data streams [24], time domains [28], semantic models [9]), to the more recent forays in low-latency processing such as Spark Streaming [34], MillWheel, and Storm [5], modern consumers of data wield 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 offline experiments over large swaths of historical data.

Advertisers/content providers want to know how often and for how long their videos are being watched, with which content/ads, and by which demographic groups. They also want to know how much they are being charged/paid. They

Watermark



Timeline

Precision not Recall

Models and Issues in Data Stream Systems *

Brian Babcock Shivnath Babu Mayur Datar Rajeev Motwani Jennifer Widom

The 8 Requirements of Real-Time Stream Processing

Michael Stonebraker Uğur Çetintemel Stan Zdonik

MOA: Massive Online Analysis

Albert Bifet
Geoff Holmes
Richard Kirkby
Bernhard Pfahringer

Dynamically Scaling

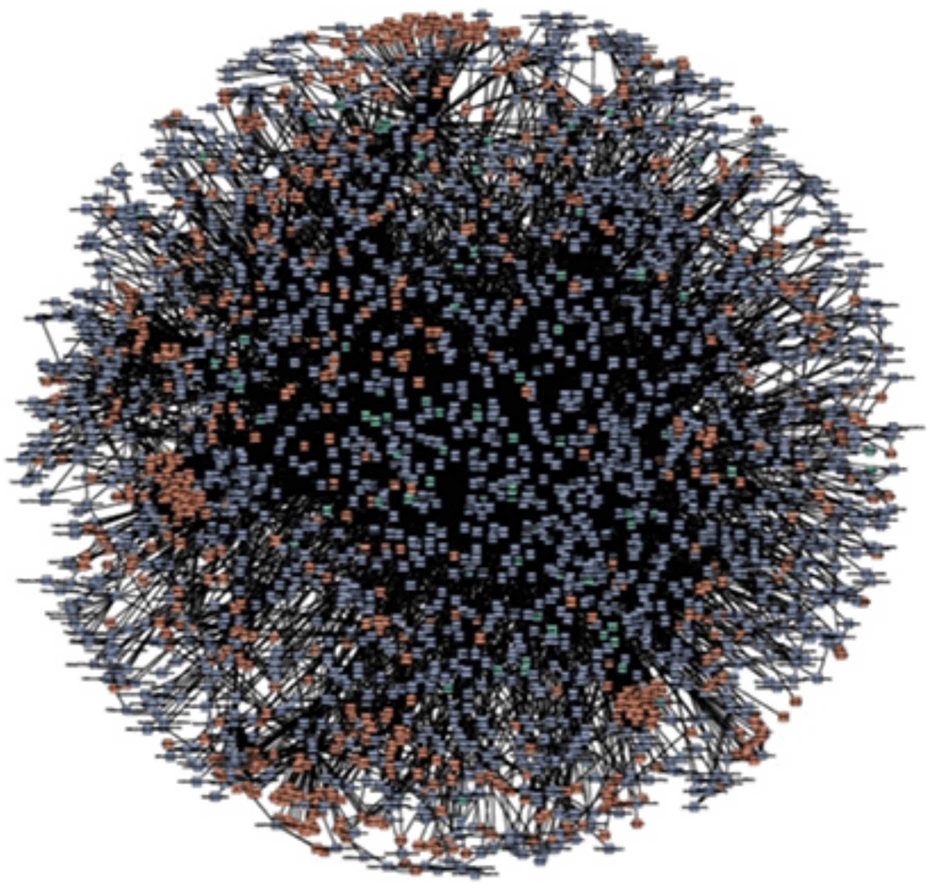
Apache Samza

Martin Kleppmann

Streaming: A Declarative API for Real-Time Applications in Apache Spark

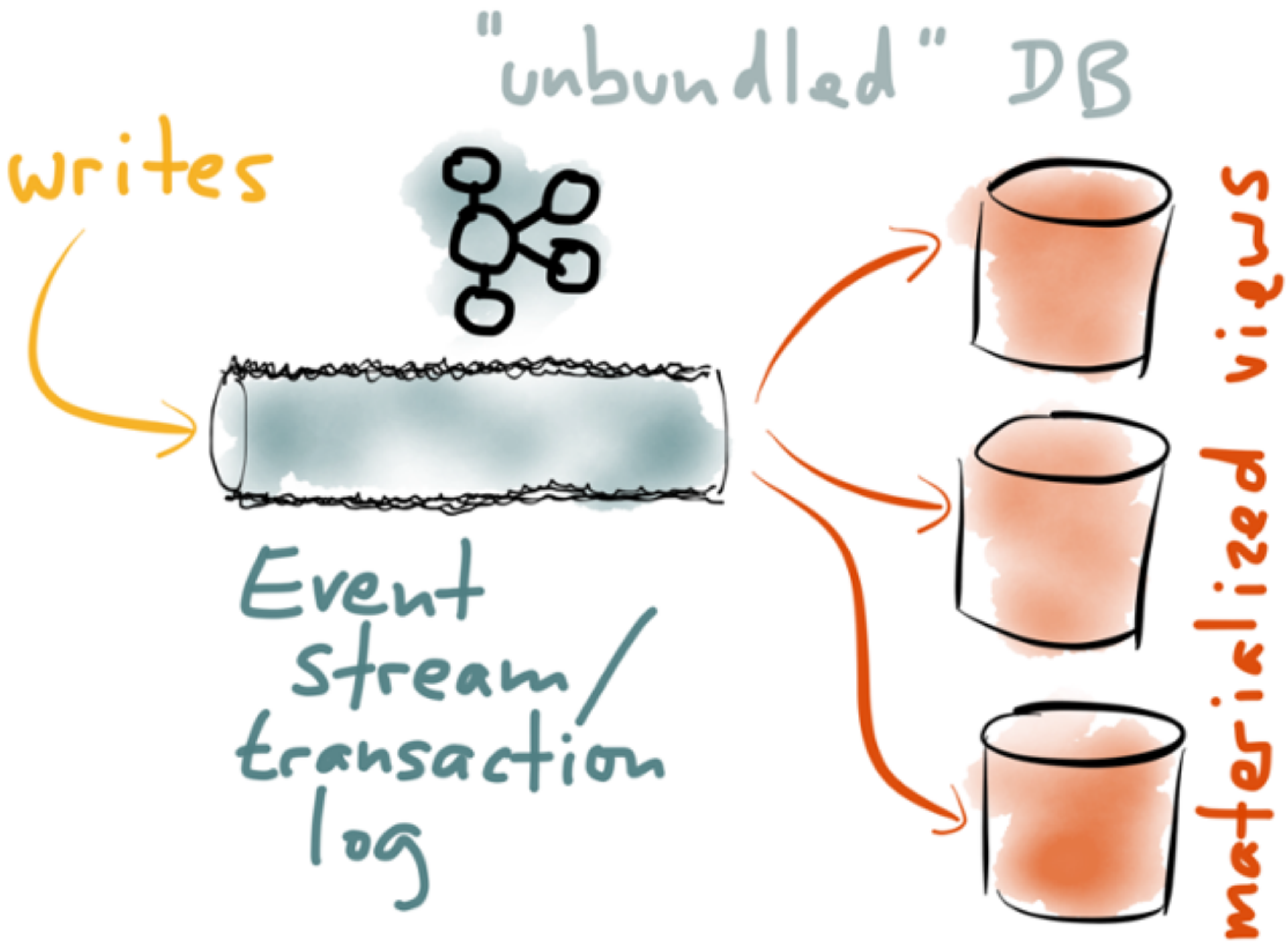
2002 2005 2006 2007 2008 2010 2014 2015 2018

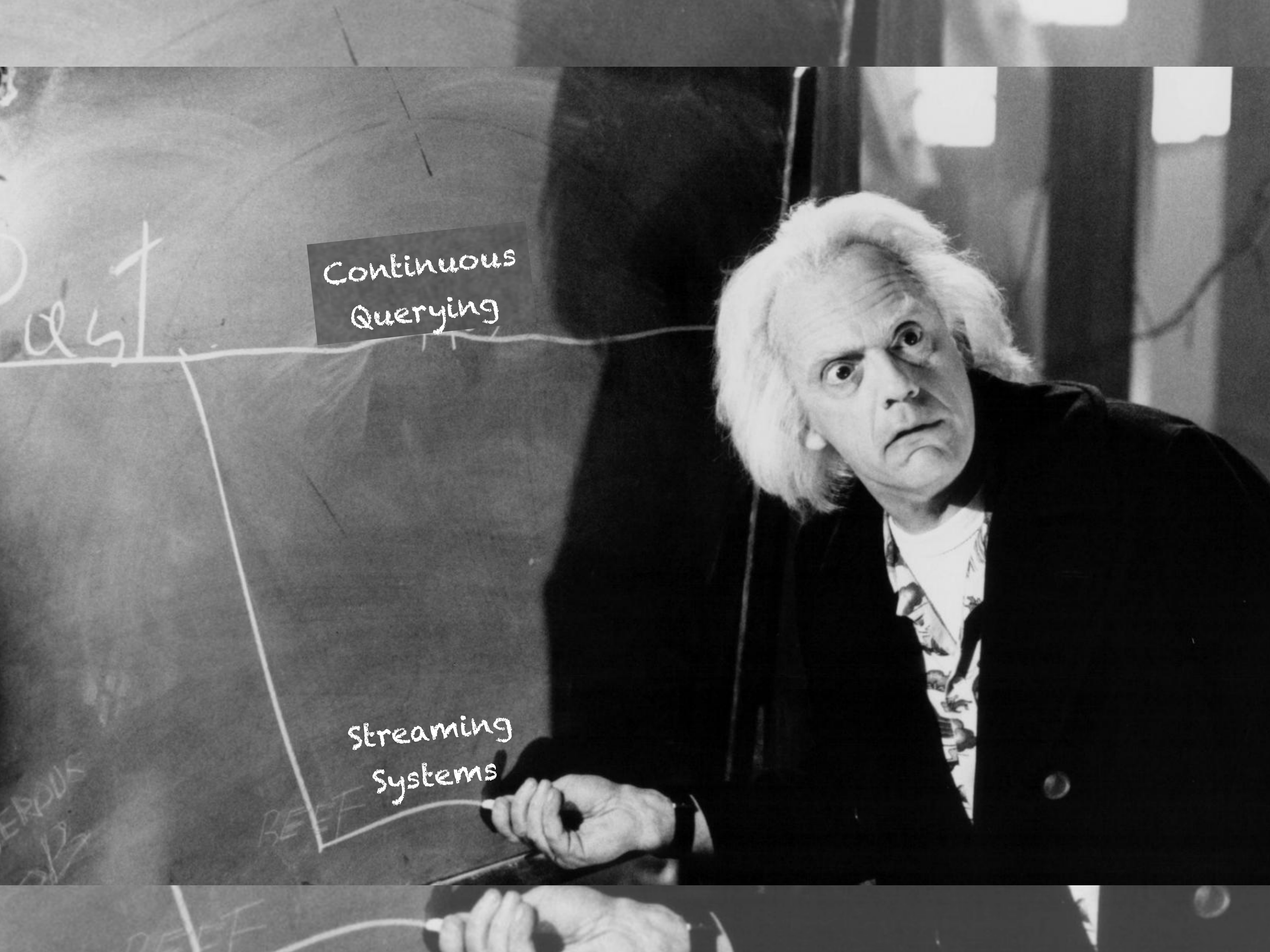

LOGS DON'T LIE
imgflip.com

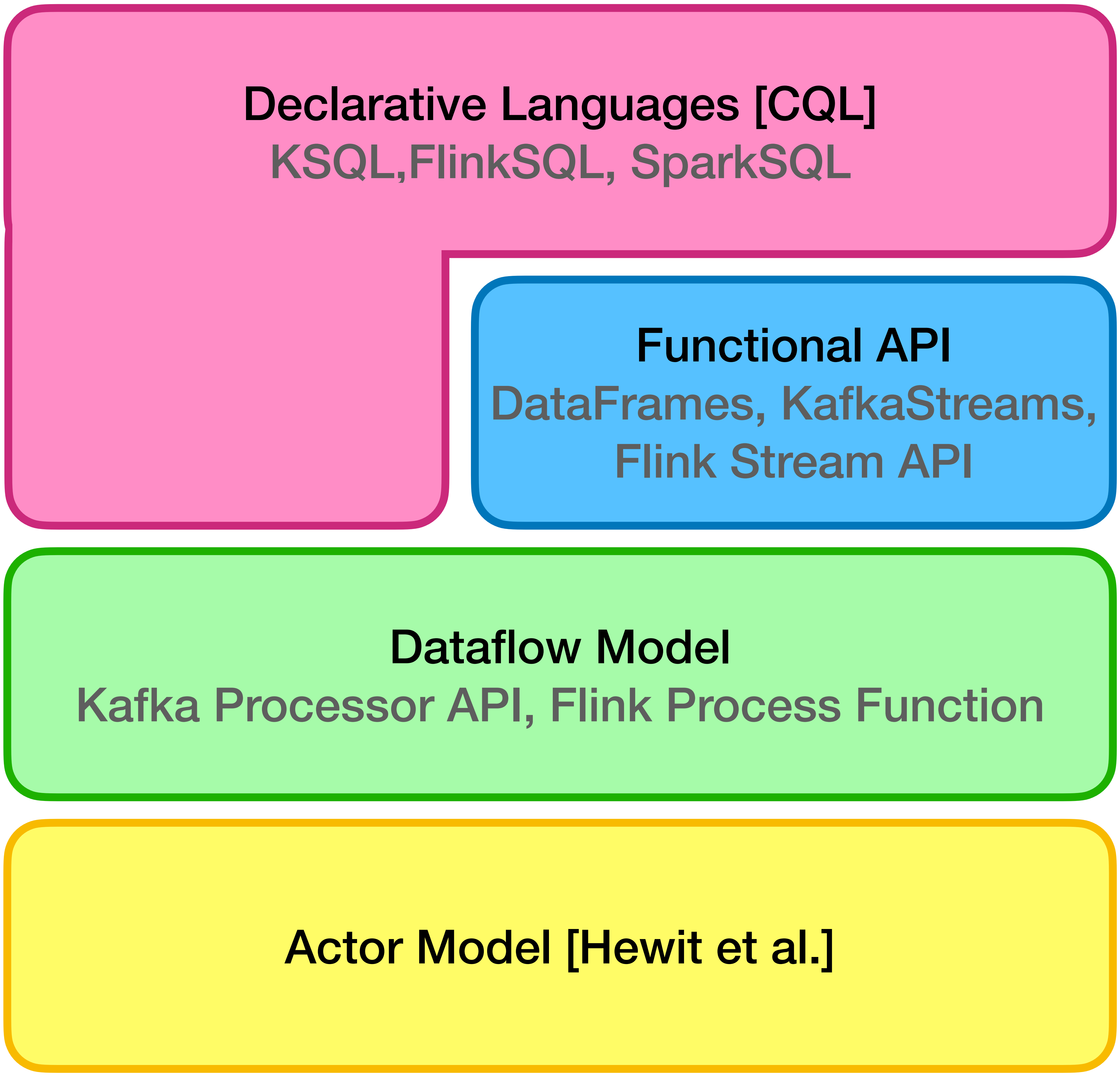


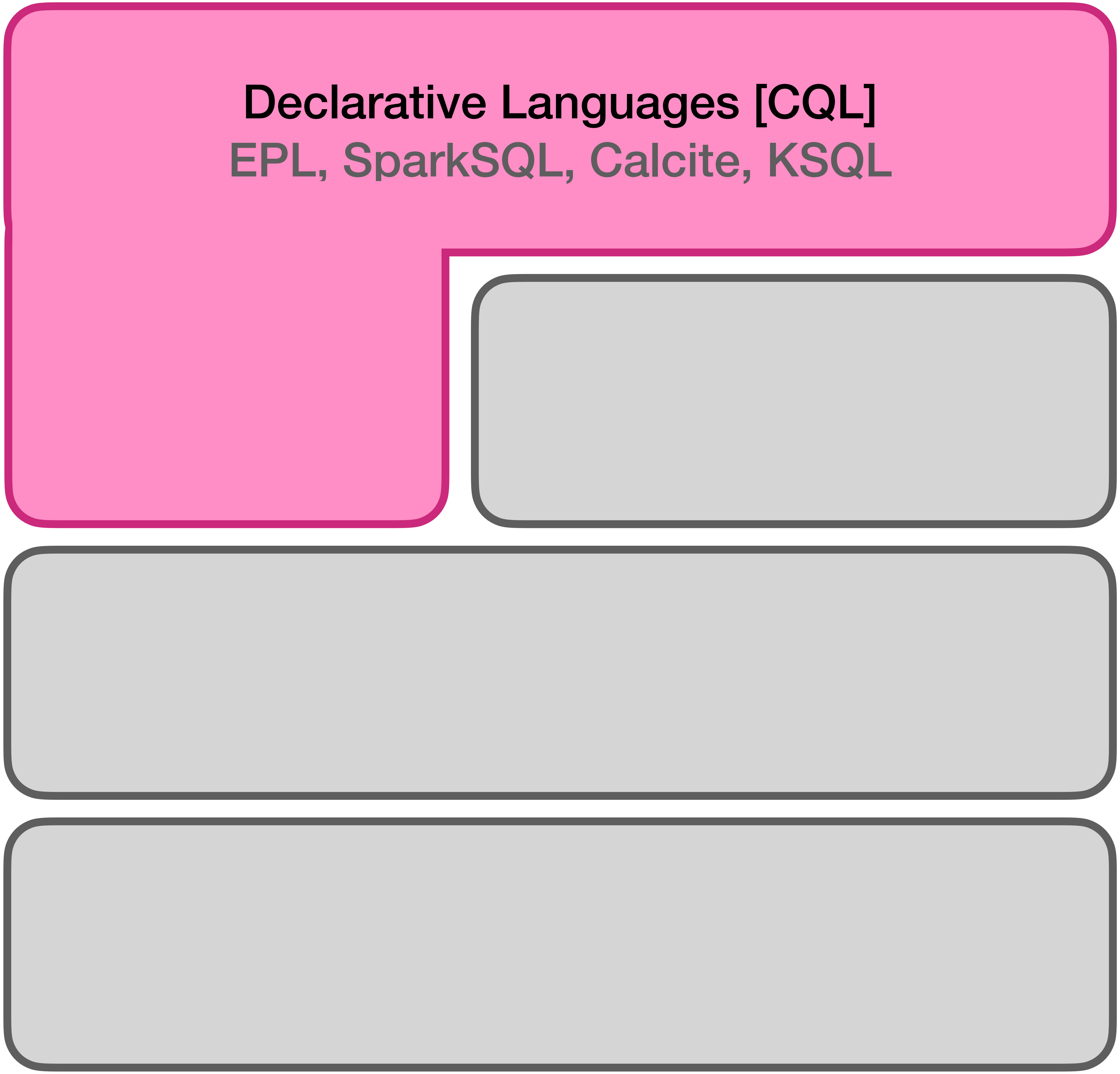
amazon.com

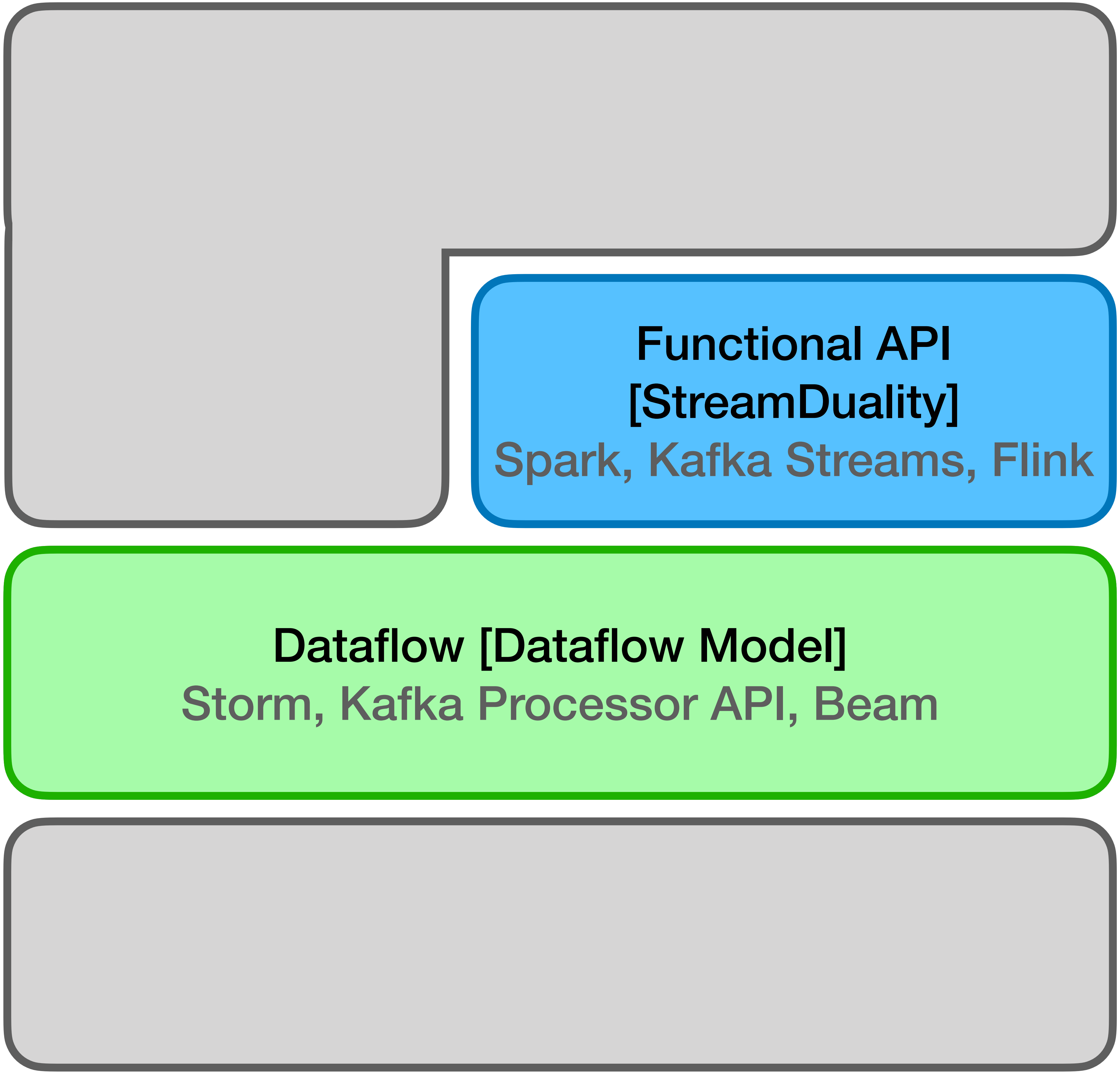
NETFLIX


"unbundled" DB
writes
Event stream/transaction log
materialized views

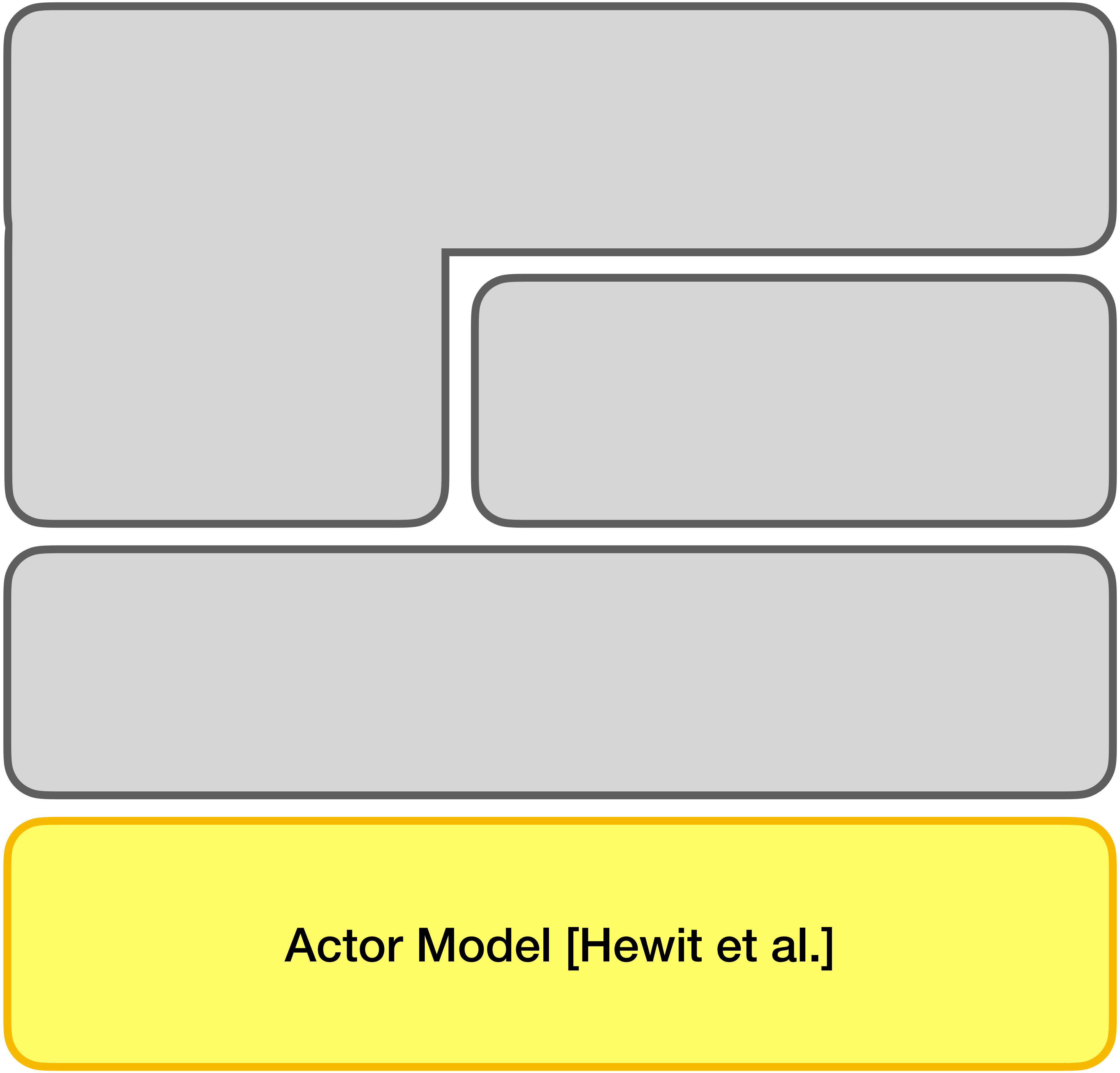

Continuous Querying
Streaming Systems


Declarative Languages [CQL]
KSQL,FlinkSQL, SparkSQL
Functional API
DataFrames, KafkaStreams,
Flink Stream API
Dataflow Model
Kafka Processor API, Flink Process Function
Actor Model [Hewit et al.]


Declarative Languages [CQL]
EPL, SparkSQL, Calcite, KSQL

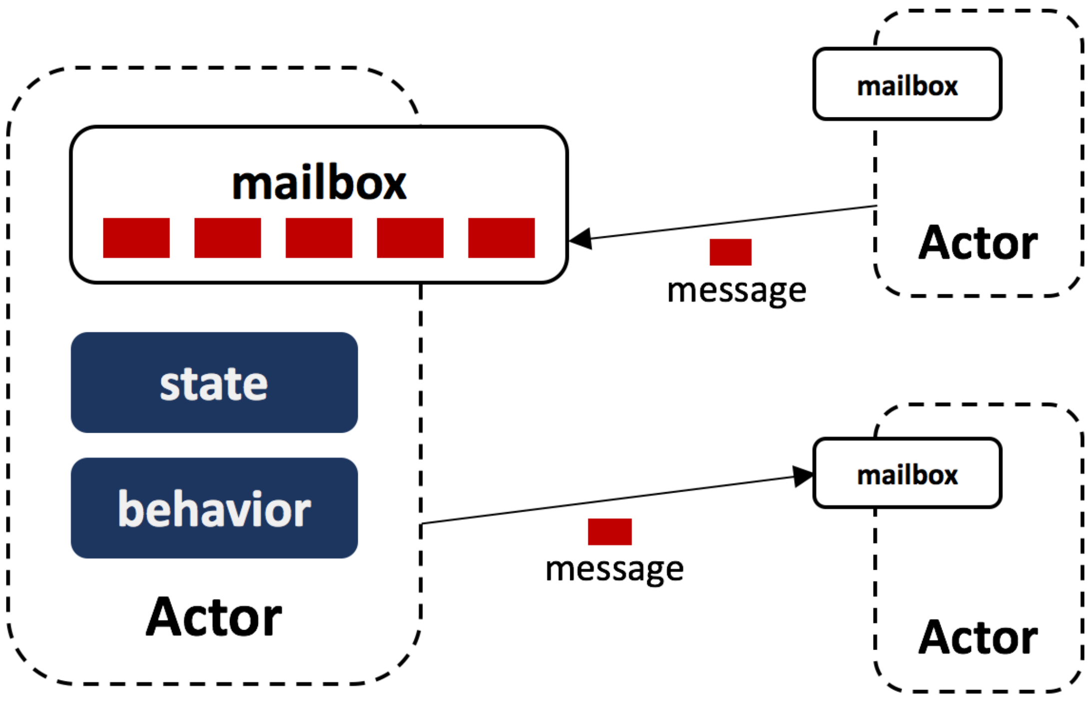


Functional API
[StreamDuality]
Spark, Kafka Streams, Flink
Dataflow [Dataflow Model]
Storm, Kafka Processor API, Beam


Actor Model [Hewit et al.]

Actors

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

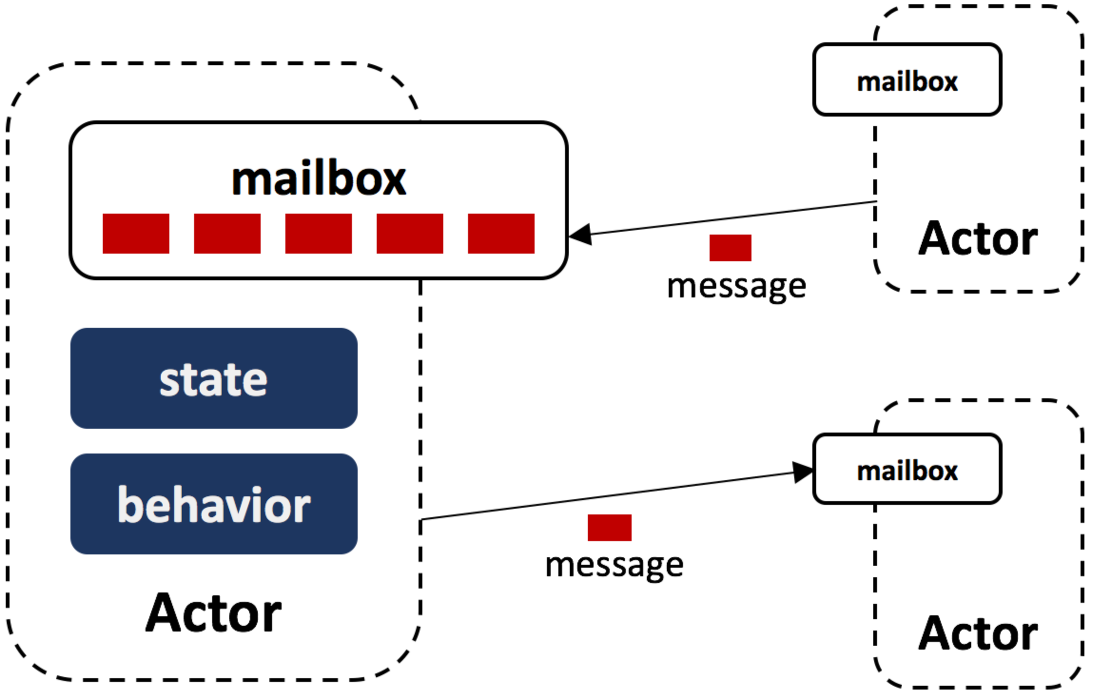


Actor Model & Stream Processing

- 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.



echo “hello” | figlet 2 | cowsay

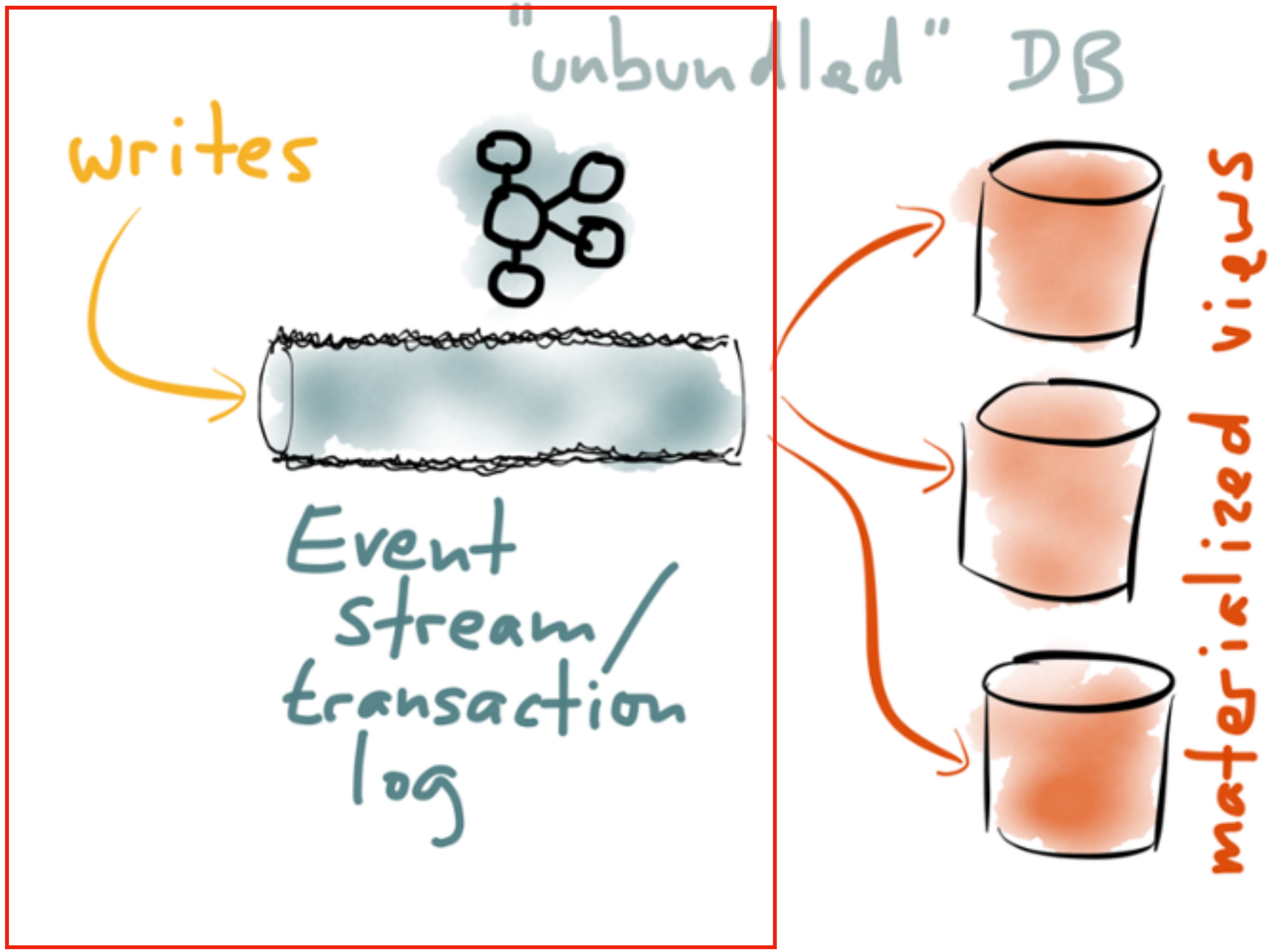
microservice1 | microservice 2 | microservice3

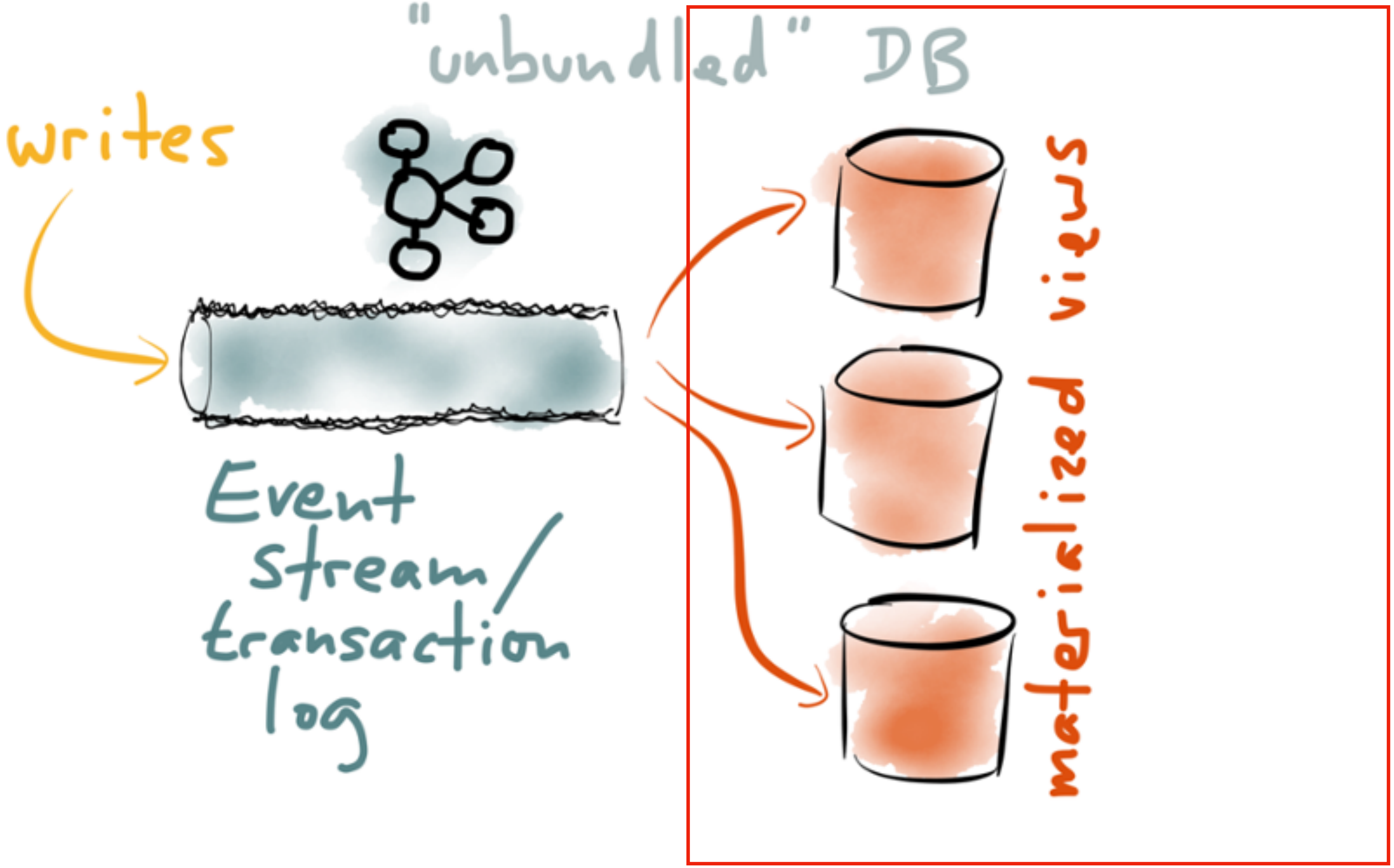
microservice1

microservice 2

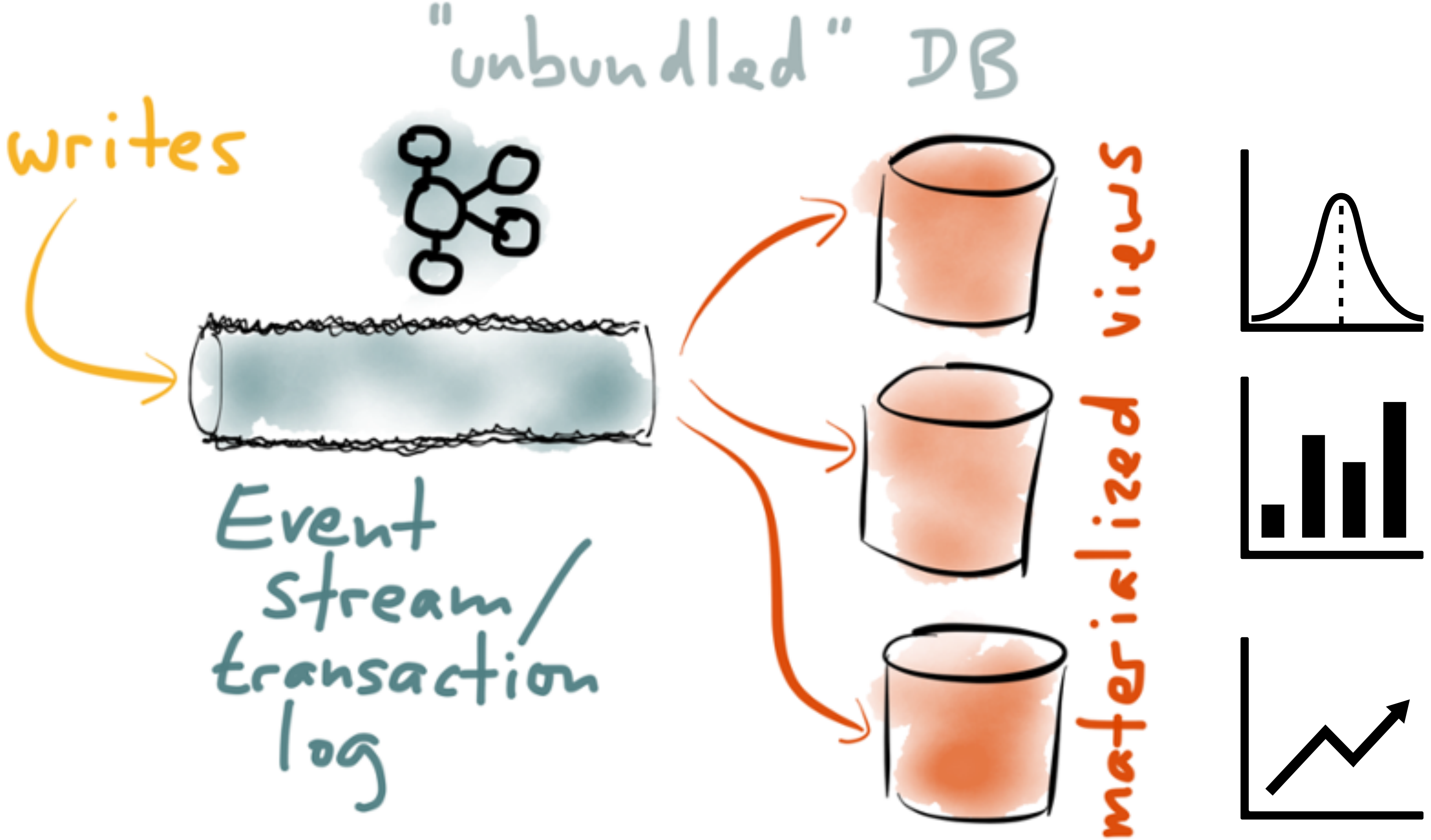
microservice 2


“
L’ETAT
C’EST
MOI
“
- Louis XIV
- ApacheKafka


"unbundled" DB
writes
Event stream/ transaction log
materialized views



"unbundled" DB
writes
Event stream/ transaction log
materialized views

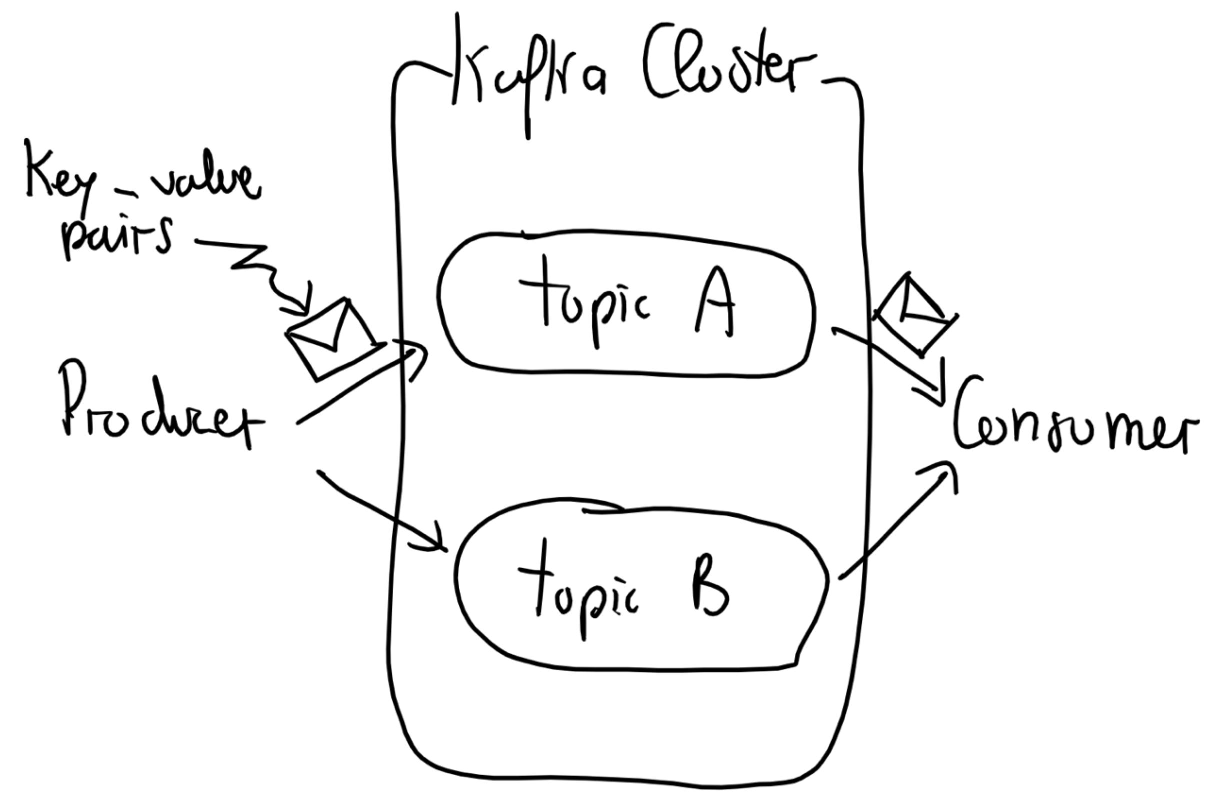


"unbundled" DB
writes
Event stream/ transaction log
materialized views


Should I Take a Step back?

A Conceptual View of Kafka

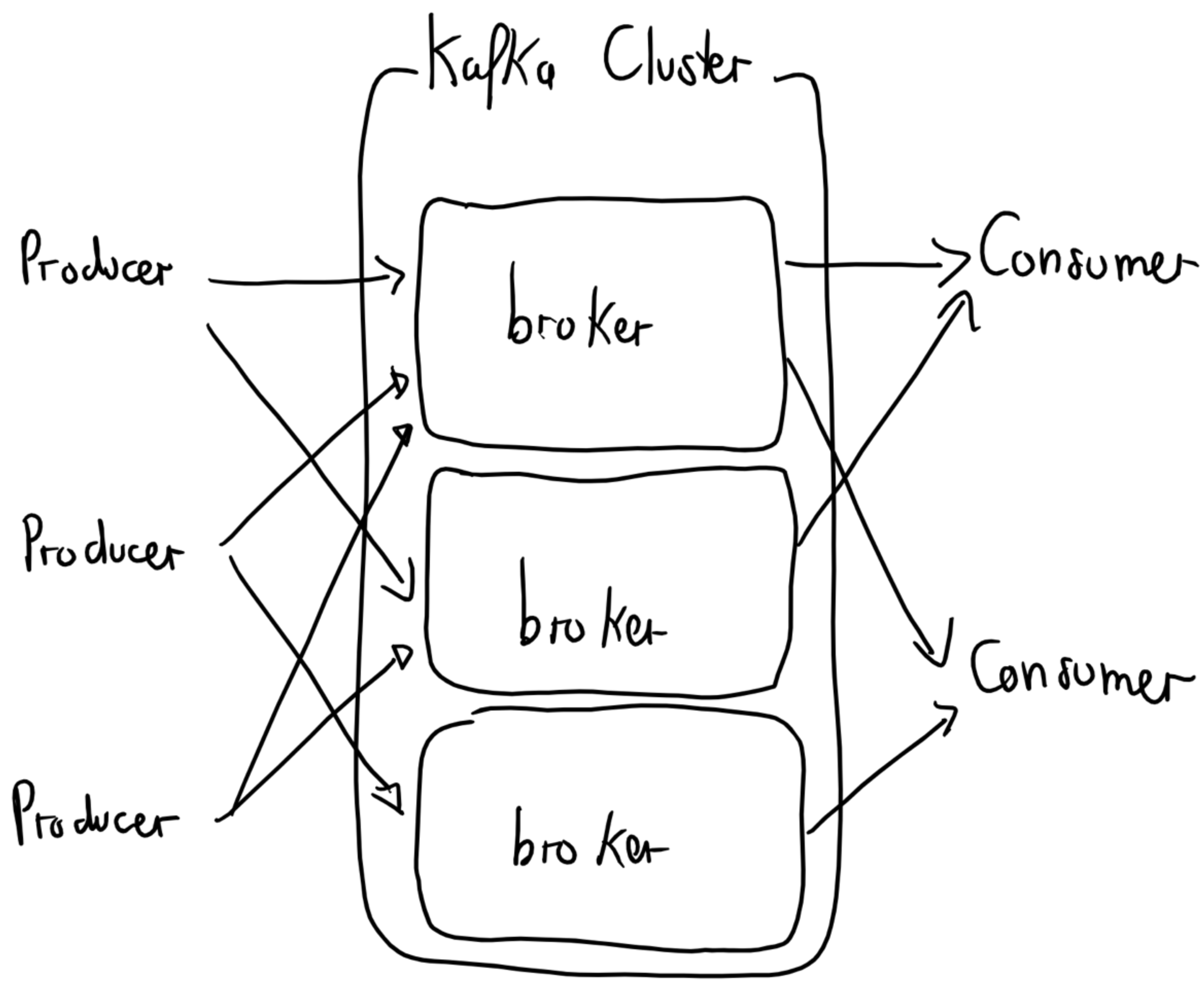
- **Producers** send messages on topics
- **Consumers** read messages from topics
- **Messages** are key-value pairs
- **Topics** are streams of messages
- Kafka cluster manages topics



curtesy of Emanuele Della Valle - http://emanueledellavalle.org

A Logical View of Kafka

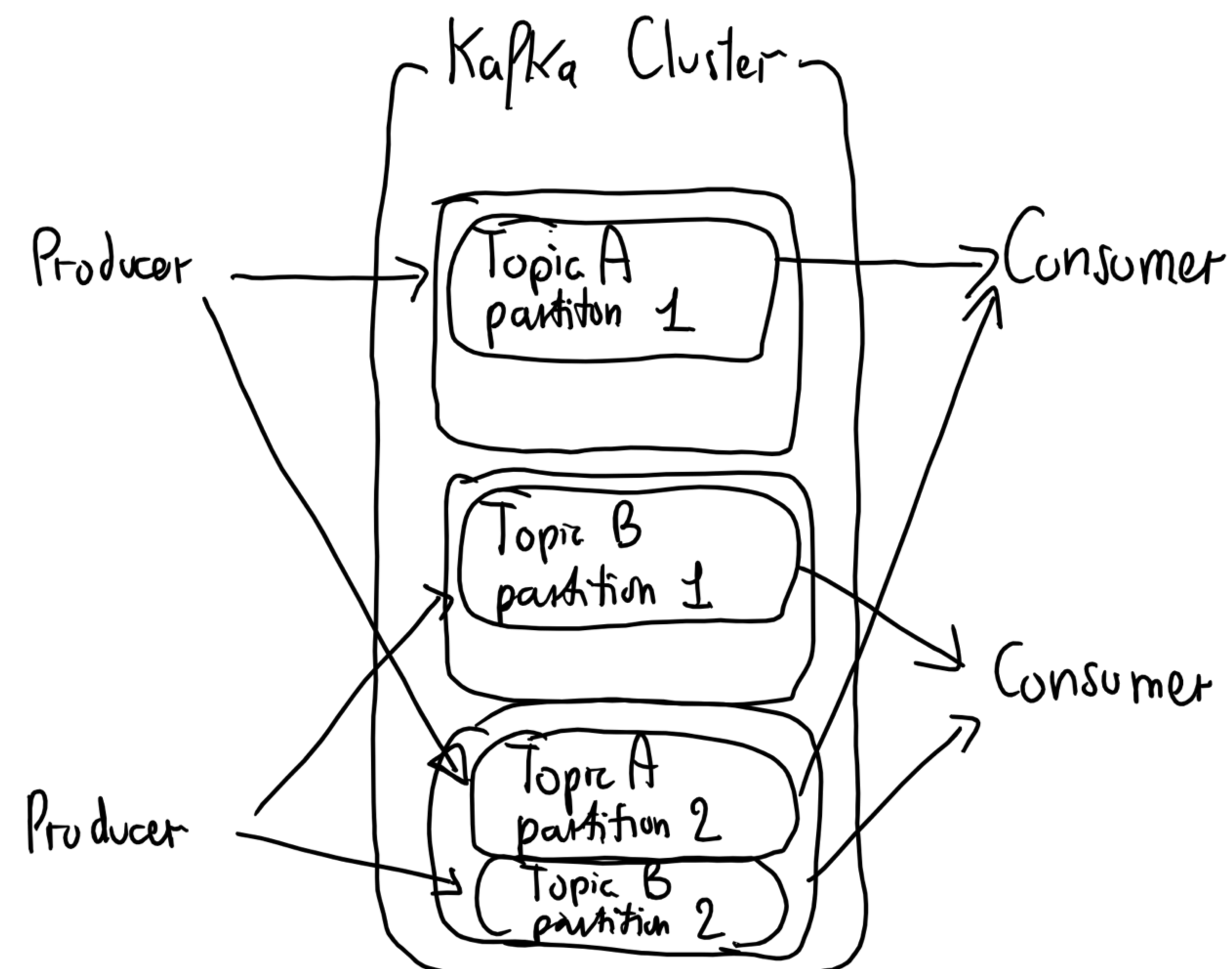
- **Brokers** are the main storage and messaging components of the Kafka cluster



curtesy of Emanuele Della Valle - http://emanueledellavalle.org

Reconciling the two views of Kafka

- Topics are partitioned across brokers
- Producers shard messages over the partitions of a certain topic
- Typically, the message key determines which Partition a message is assigned to



curtesy of Emanuele Della Valle - http://emanueledellavalle.org

Topic partitioning invites distributed consumption

- Different Consumers can read data from the same Topic
  - By default, each Consumer will receive all the messages in the Topic
- Multiple Consumers can be combined into a Consumer Group
  - Consumer Groups provide scaling capabilities
  - Each Consumer is assigned a subset of Partitions for consumption