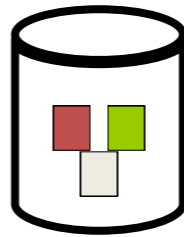


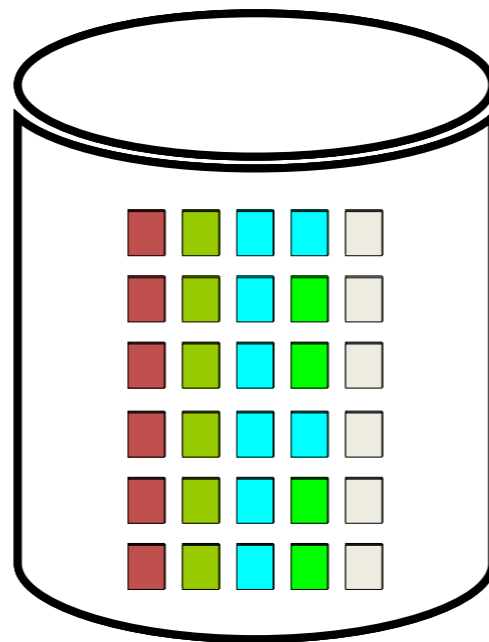
A Brief History of Stream Processing



Data

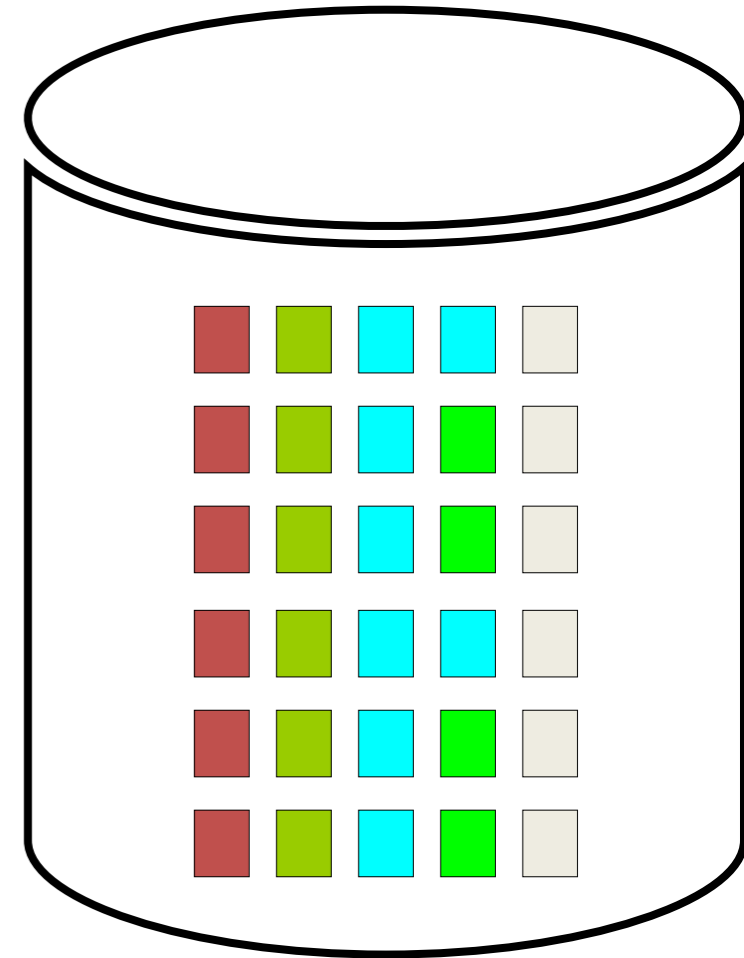
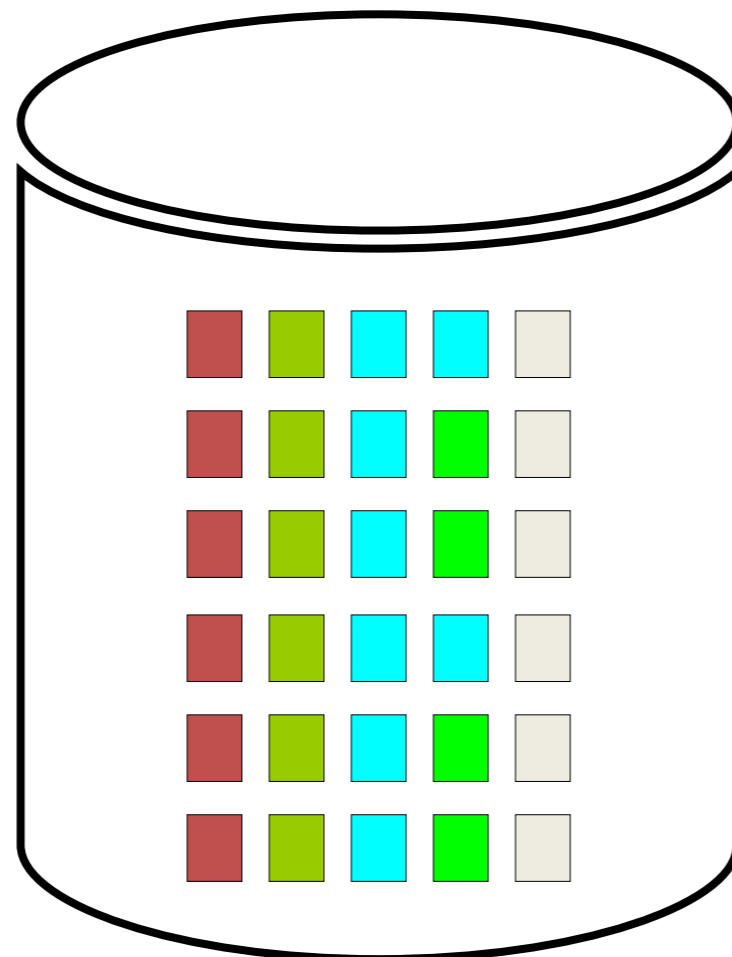
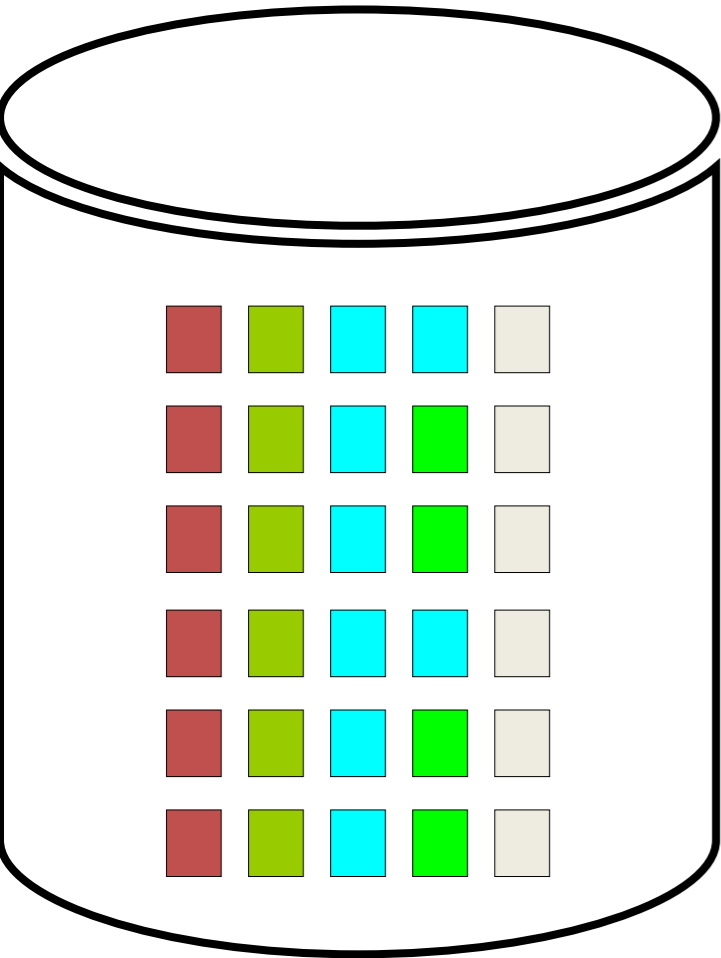


Can Be Big

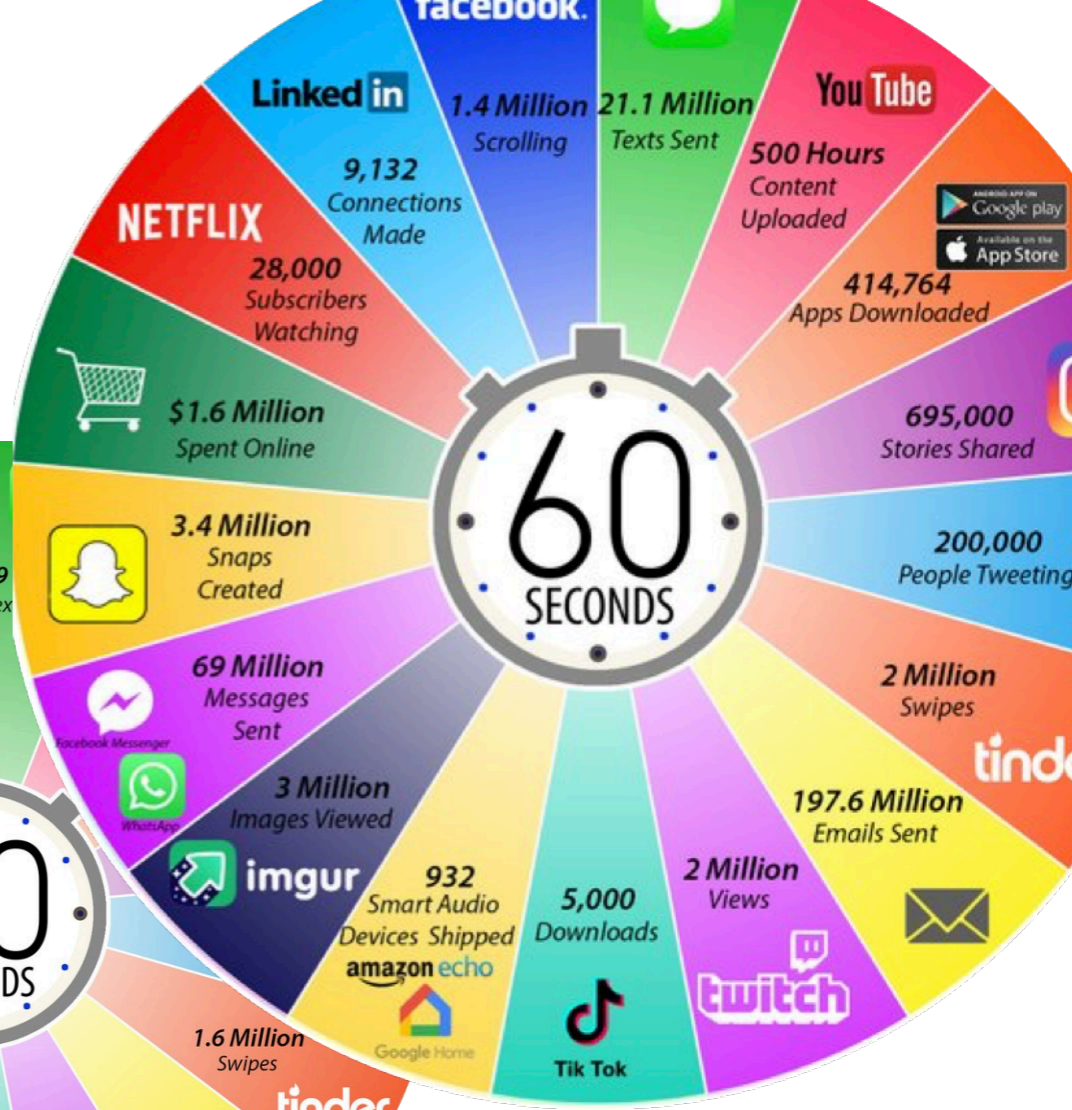
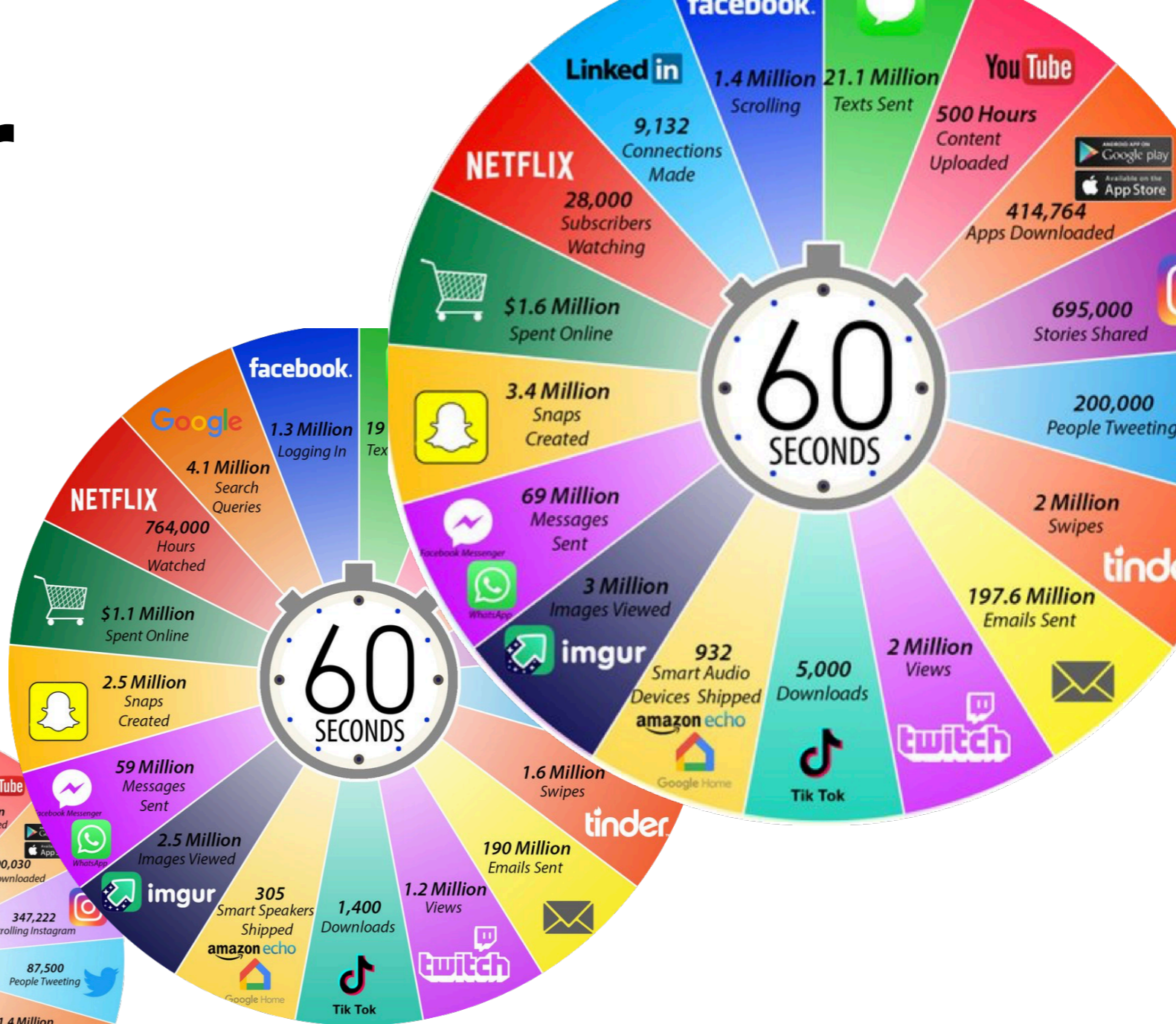




Very Big



Data Never Sleep!



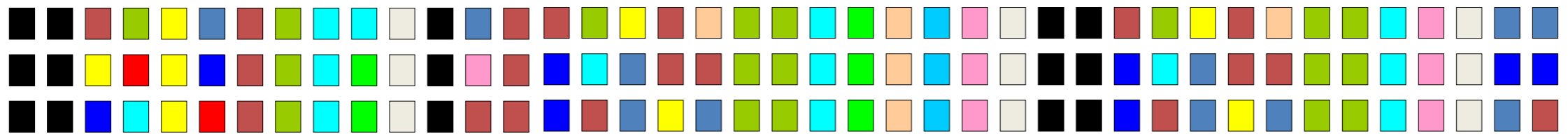
2018

2019

2020

2021

So big they never end



time



So What?

The traditional data processing infrastructures are challenged:

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

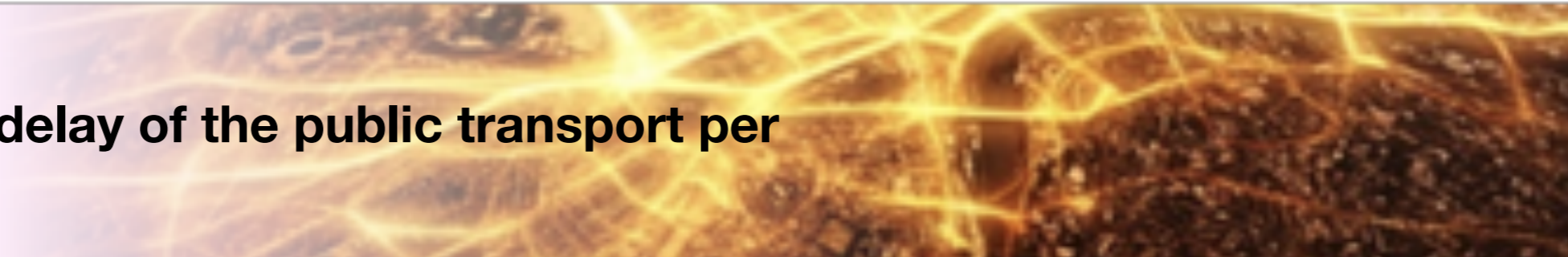


... Excel At Historical Descriptive Analysis

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

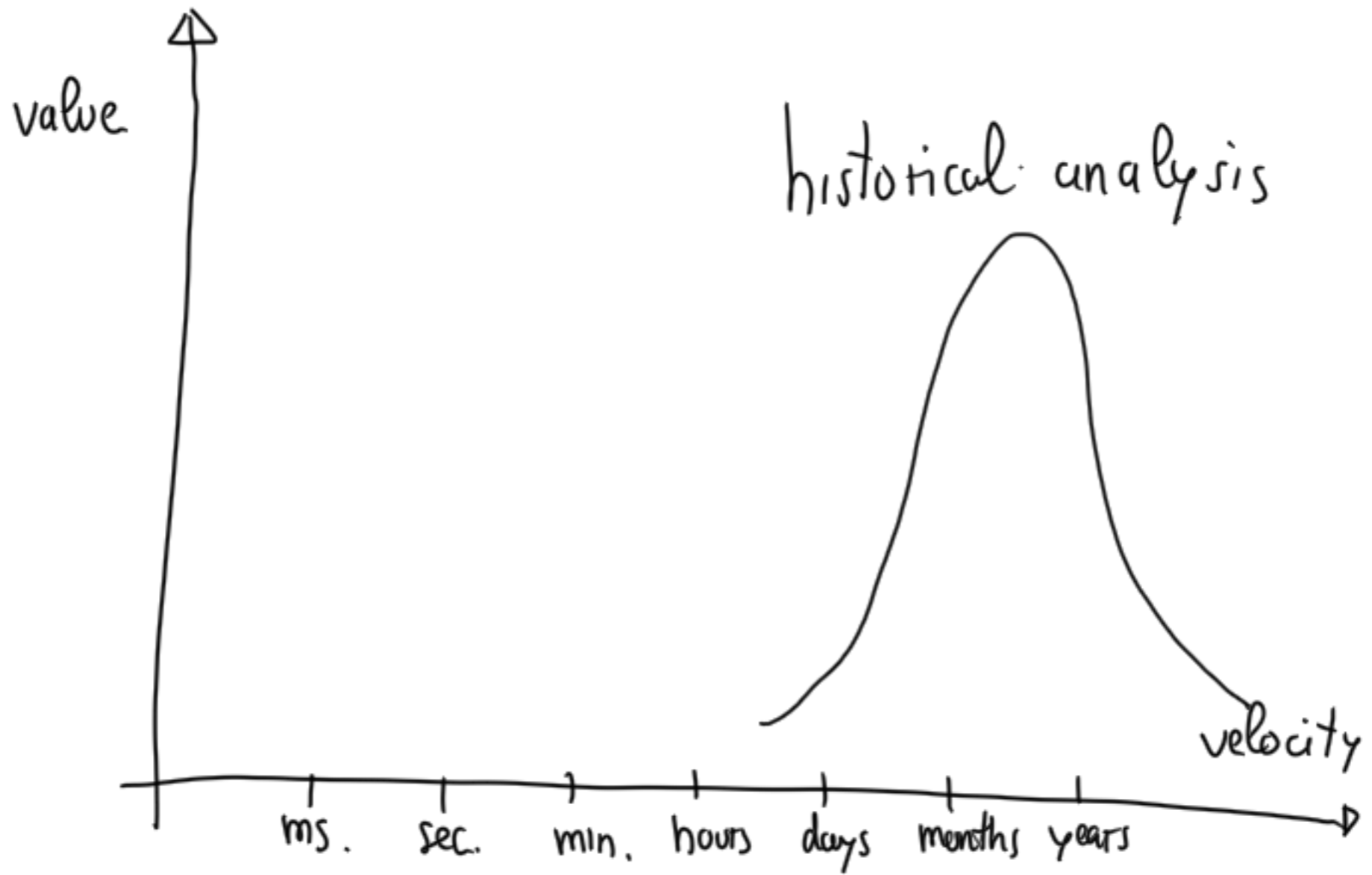


What is the maximum delay of the public transport per city district?



Which content features are correlated to high impact posts?





... Struggling With Prescriptive Analysis

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

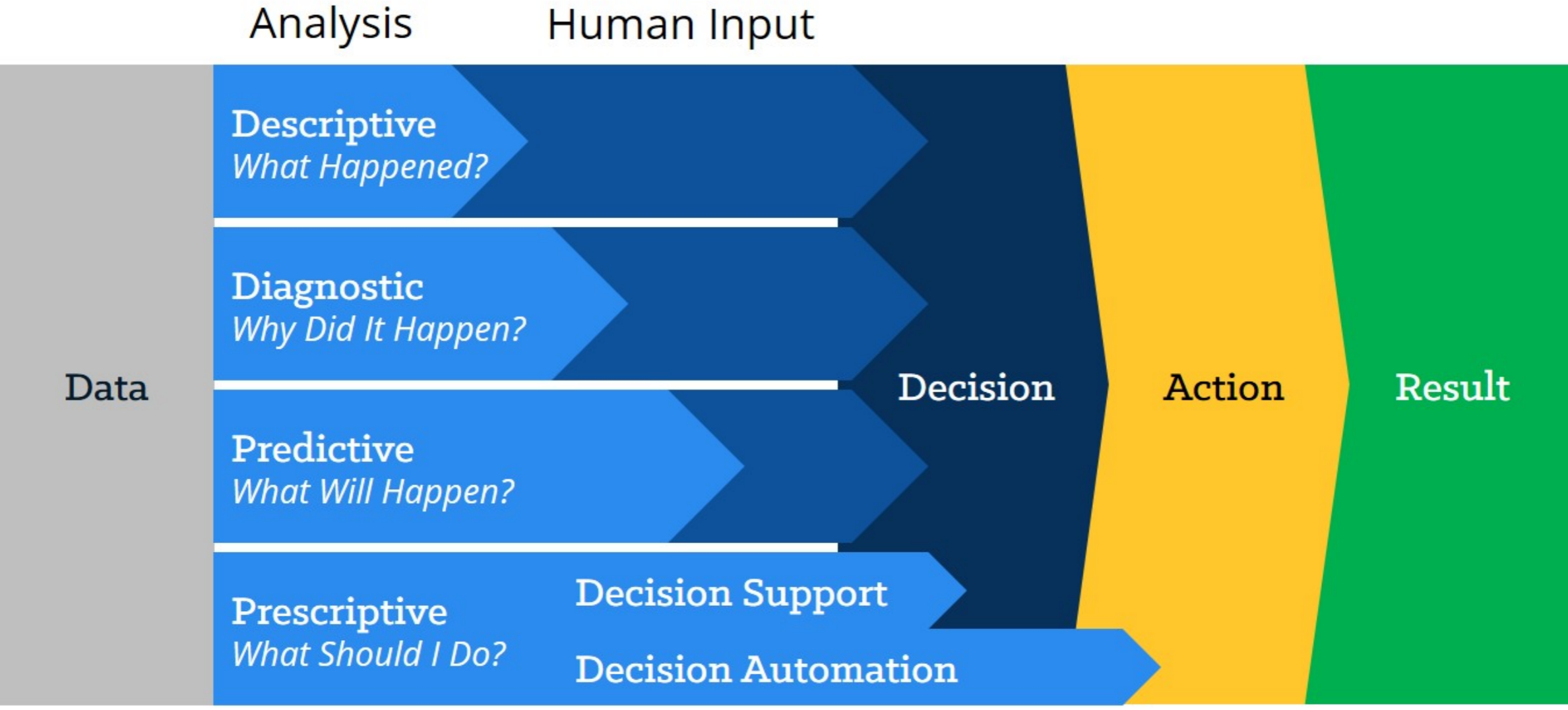


Can I get to that meeting in the next 15 min using public transport?



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

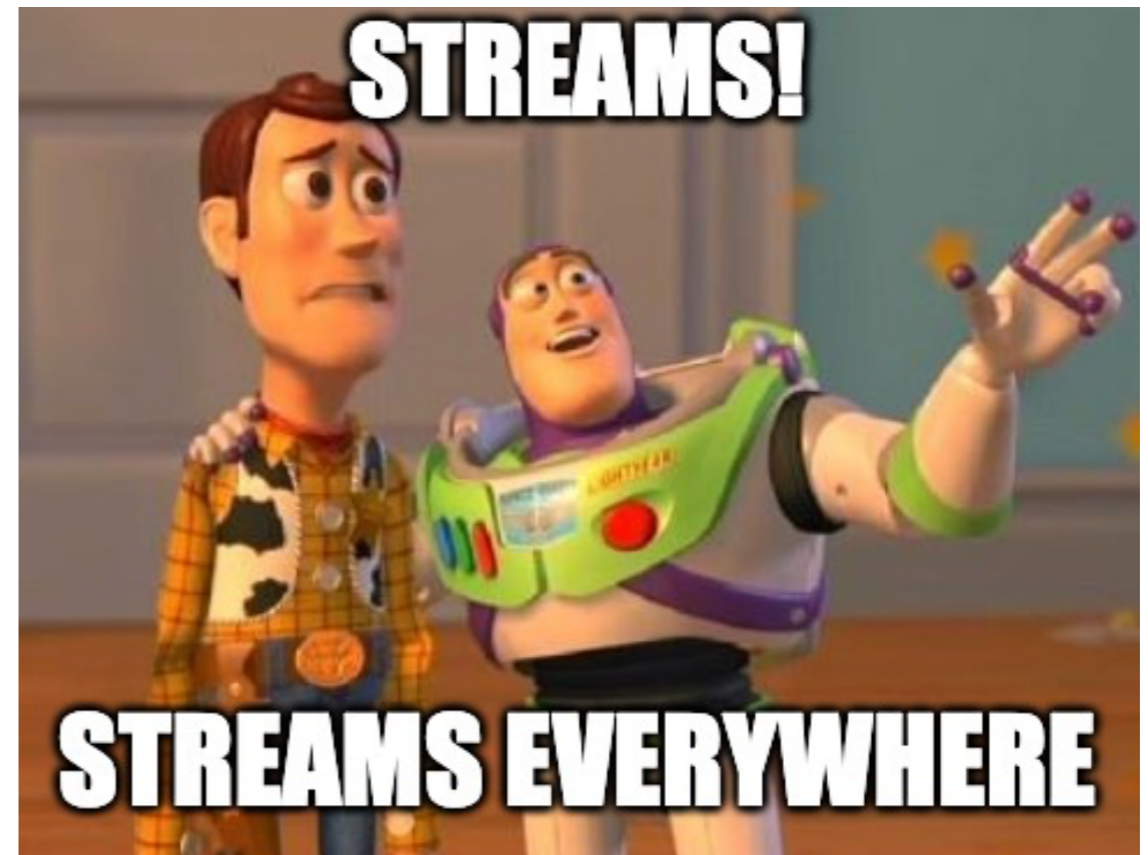




So What?

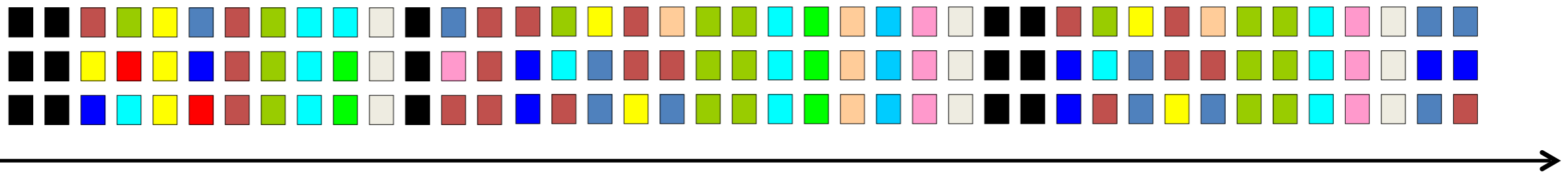
The traditional data processing infrastructures are challenged:

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



What is a Stream?

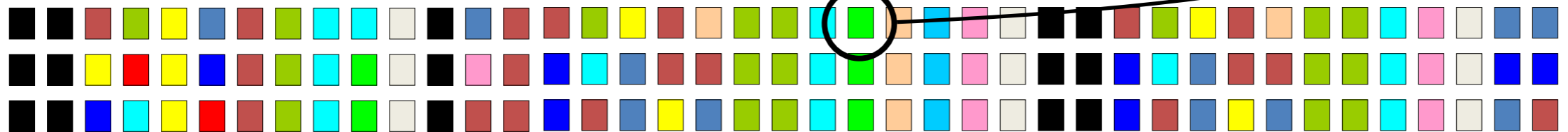
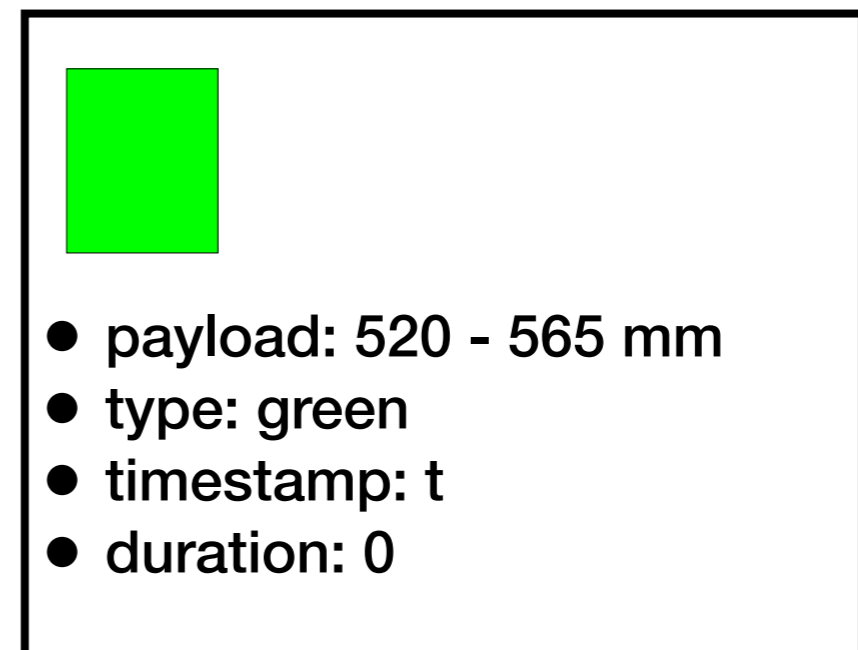
- **Streams:** unbounded partially ordered sequence of data in form of object-timestamp pairs $\langle o, t \rangle$, e.g.,
 - o is a data item
 - t is a natural number



time

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



time

**How to process a
stream?**

Stream Computing

Continuous Algorithms

Sort out all the colours in the streams



order

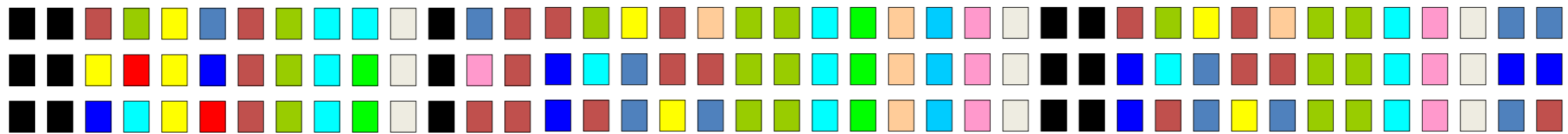
Querying Streams

Data Streams Management Systems

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

(, 15)

1 minute wide window



time

Timeline

Precision not Recall

Models and Issues in Data Stream Systems *

Brian Babcock Shivnath Babu Mayur Datar Rajeev Motwani Jennifer Widom

Department of Computer Science
Stanford University
Stanford, CA 94305
{babcock,shivnath,datar,rajeev,widom}@cs.stanford.edu

Abstract

In this overview paper we motivate the need for and research issues arising from a new model of data processing. In this model, data does not take the form of persistent relations, but rather arrives in multiple, continuous, rapid, time-varying data streams. In addition to reviewing past work relevant to data stream systems and current projects in the area, the paper explores topics in stream query languages, new requirements and challenges in query processing, and algorithmic issues.

1 Introduction

Recently a new class of data-intensive applications has become widely recognized: applications in which the data is modeled best not as persistent relations but rather as transient *data streams*. Examples of such applications include financial applications, network monitoring, security, telecommunications data management, web applications, manufacturing, sensor networks, and others. In the data stream model, individual data items may be relational tuples, e.g., network measurements, call records, web page visits, sensor readings, and so on. However, their continuous arrival in multiple, rapid, time-varying, possibly unpredictable and unbounded streams appears to yield some fundamentally new research problems.

In all of the applications cited above, it is not feasible to simply load the arriving data into a traditional database management system (DBMS) and operate on it there. Traditional DBMS's are not designed for rapid and continuous loading of individual data items, and they do not directly support the *continuous queries* [84] that are typical of data stream applications. Furthermore, it is recognized that both *approximation* [13] and *adaptivity* [8] are key ingredients in executing queries and performing other processing (e.g., data analysis and mining) over rapid data streams, while traditional DBMS's focus largely on the opposite goal of precise answers computed by stable query plans.

In this paper we consider fundamental models and issues in developing a general-purpose *Data Stream Management System* (DSMS). We are developing such a system at Stanford [82], and we will touch on some of our own work in this paper. However, we also attempt to provide a general overview of the area, along with its related and current work. (Any glaring omissions are, naturally, our own fault.) We begin in Section 2 by considering the data stream model and queries over streams. In this section we

Apache Samza

Martin Kleppmann

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-Medvedev, Reuven Lax, Sam McClellan, Daniel Mills, Frances Perry, Eric Schmidt, Sam Whittam

Google

{akidau,robertw,chambers,chernyak,fernan,relax,sgmc,milstd,tip,clouds,samuelw}@google.com

Dynamically Scaling of S

2015 IEEE First International C

Jan Sipke van der Veen¹, Bram van d

¹TN
University of
{jan_sipke,vanderveen, bram.van

ABSTRACT

Unbounded, unordered, global-scale datasets are increasingly common in data-intensive businesses (e.g. Web logs, mobile user 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 insurmountable 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 those seemingly competing propositions, often resulting in disparate implementations and systems.

We propose that a fundamental shift of perspective is necessary to deal with those evolved requirements in modern data processing. We propose a field-level step-trying to green up bounded datasets into finite pools of information that eventually become complete, and instead live and breathe under the assumption that we will never know if when we have seen all of our data, only that new data will arrive, old data may be retraced, 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

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 [38]), 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 [26], semantic models [9]), to the more recent forays in low-latency processing such as Spark Streaming [34], MihVioel, and Storm [5], modern consumers of data would appreciate amounts of power in shaping and fast-igniting 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 operations 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

the writes of records to a

processes jobs are long-
er application logic on
producing derived output
potentially writing output

instream data sources. The physical and logical order of data
in a stream may become investment in such a setting. Exis-

ing models either neglect these dependencies or handle them by means of data buffering and recordering techniques thereby compromising processing latency.

In this paper, we introduce the Dataflow Model as a stream of data, differentiating the Data Streaming Model from other bounded physical and logical order data streams. This model presents the result of an operator as a stream of ordered updates, which induces a stream of results and streams. As such, it provides natural way to cope with inconsistencies between the physical and logical order of streaming data in a continuous manner, without explicit buffering and recordering. We further discuss the trade-offs and challenges faced when implementing this model in terms of correctness, latency, and processing cost. A case study based on Apache Kudu illustrates the effectiveness of our

o Sides of the Same C

Guozhang Wang
Cosilium Inc.
Palo Alto, USA
gzwang@cosilium.io

KEYWORDS

Stream Processing, Processing Model, Semantic Model, ACM Reference Format

Martianus J. van Gemund, Wang, Matthias Weinert, Christoph Freytag, 2015, Streams and Tables: Two Sides of the Same Coin. In International Workshop on Real-Time Data and Analytics (StreamDTA '15), August 27, 2015, Rio de Janeiro, Brazil.

1 INTRODUCTION

Stream processing has emerged as a parallel real-time over synchronized requests of data, enabling processing of large-scale data in a continuous manner. As such, the stream processing paradigm has particularly suited to support the requirements for new logic in large organizations. It provides the flexibility for communication between independent data streams, large systems, a.k.a. "microservices", through message-passing [19].

The VLDB Journal (2009.12).12:1-147
DOI: 10.1007/s00375-009-0242-7

REGULAR PAPER

Arnold Aron Shivnath Babu Jennifer Widom

The CQL continuous query language: semantic foundations and query execution

Received: 1 June 2009 / Accepted: 22 November 2009 / Published online: 22 July 2009

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 stream and related 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 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, interpreter queries, synopsis, and sharing of components among multiple operators and queries. Examples throughout the paper are drawn from the Linear Algebra benchmark recently proposed for DSMS. We also contain a public repository of data stream applications that includes a wide variety of queries expressed in CQL.

Keywords continuous streams · continuous queries · declarative query language · relational database management systems (DBMS) at Stanford

Journal of Machine Learning Research 11 (2010) 1401–1494

Submitted 11/09, Published 4/10

MOA: Massive Online Analysis

Albert Bifet
Geoff Holmes

Richard Kirshy
Bernhard Pfahringer

Department of Computer Science
University of Waikato
Hamilton, New Zealand

Editor: Mihko Trawn

Abstract

Massive Online Analysis (MOA) is a software environment for implementing algorithms and running experiments for online learning from evolving data streams. MOA includes a collection of offline and online methods as well as tools for evaluation. In particular, it implements boosting, bagging, and Hoeffding Trees, all with and without Naive Bayes classifiers at the leaves. MOA supports bidirectional interaction with WEKA, the Waikato Environment for Knowledge Analysis, and is released under the GNU GPL license.

Keywords: data streams, classification, ensemble methods, java, machine learning software

1. Introduction

Green computing is the study and practice of using computing resources efficiently. A main approach to green computing is based on algorithmic efficiency. In the data stream model, data arrive at high speed, and an algorithm must process them under very strict constraints of space and time. MOA is an open-source framework for dealing with massive evolving data streams. MOA is related to WEKA, the Waikato Environment for Knowledge Analysis, which is an award-winning open-source workbench containing implementations of a wide range of batch machine learning methods.

A data stream environment has different requirements from the traditional batch learning setting. The most significant are the following:

- Requirement 1 Process an example at a time, and inspect it only once (at most)
- Requirement 2 Use a limited amount of memory

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

Paris Carbone¹
Asterics Kassandrafmios²
paris.carbone@kth.se

Stephan Ewen³
Völkner Mark³
'data Artisans
first.data-artisans.com

Seif Haridi¹
Kostas Tzoumas¹
'TU Berlin & OFKI
first.haridi@tu-berlin.de

Abstract

Apache Flink™ is an open-source system for processing streaming and batch data. Flink is built on the philosophy that many classes of data processing applications, including real-time analytics, continuous data pipelines, historic data processing (batch), and iterative algorithms (machine learning, graph analysis) can be expressed and executed as pipeline-like-tolerant workflows. In this paper, we present Flink's architecture and expand on how a (seemingly diverse) set of use cases can be unified under a single execution model.

1 Introduction

Data-stream processing (e.g., as exemplified by complex event processing systems) and static (batch) data processing (e.g., as exemplified by MPP databases and Hadoop) were traditionally considered as two very different types of applications. They were programmed using different programming models and APIs, and were executed by different systems (e.g., dedicated streaming systems such as Apache Storm, IBM InfoSphere Streams, Microsoft StreamInsight, or Streambase versus relational databases or execution engines for Hadoop, including Apache Spark and Apache Drill). Traditionally, batch data analysis made up for the lion's share of the use cases, data sizes, and market, while streaming data analysis mostly served specialized applications.

It is becoming more and more apparent, however, that a huge number of today's large-scale data processing use cases handle data that is, in reality, produced continuously over time. These continuous streams of data come for example from web logs, application logs, sensors, or as changes to application state in databases (transaction log records). Rather than treating the streams as streams, today's setups ignore the continuous and timely nature of data production. Instead, data records are (often artificially) batched into static data sets (e.g., hourly, daily, or monthly chunks) and then processed in a time-agnostic fashion. Data collection tools, workflow managers, and schedulers orchestrate the creation and processing of batches, in what is actually a continuous data processing pipeline. Architectural patterns such as the “lambda architecture” [21] combine batch and stream processing systems to implement multiple paths of computation: a streaming fast path for timely approximate results, and a batch offline path for late accurate results. All of these approaches suffer from high latency (imposed by batches),

Copyright 2015 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyright notice is hereby acknowledged.

The 8 Requirements of Real-Time Stream Processing

Michael Stonebraker
Department of Computer Science
Intellectual Laboratory, MIT, and
StreamBase Systems, Inc.
stonebraker@csail.mit.edu

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

Stan Zdonik

The VLDB Journal (2009.12).12:1-142
DOI: 10.1007/s00375-009-0242-7

REGULAR PAPER

Arnold Aron Shivnath Babu Jennifer Widom

The CQL continuous query language: semantic foundations and query execution

Received: 1 June 2009 / Accepted: 22 November 2009 / Published online: 22 July 2009

ABSTRACT Applications that require real-time processing of high-volume data streams 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 “run change” caused by these stream-based technology takes hold, we expect to see everything of material significance on the planet get “sensor tagged” and report its state on location in real time. This stream of the real world will lead to a “green field” of novel monitoring and control applications with high-volume and low-latency processing requirements.

Similar requirements networks for details. Real-time feed data streams 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 “run change” caused by these stream-based technology takes hold, we expect to see everything of material significance on the planet get “sensor tagged” and report its state on location in real time. This stream 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, multi-line data without network, technology, and system overhead. At the same time, some existing technologies, such as main memory DBMSs and relational engines, are also being “reimagined” by marketing departments to address these applications.

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

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 stream and related 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 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, interpreter queries, synopsis, and sharing of components among multiple operators and queries. Examples throughout the paper are drawn from the Linear Algebra benchmark recently proposed for DSMS. We also contain a public repository of data stream applications that includes a wide variety of queries expressed in CQL.

Keywords continuous streams · continuous queries · declarative query language · relational database management systems (DBMS) at Stanford

THE SEMANTIC WEB

Editor: Dieter Fensel, University of Bremen, Germany

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

Domenec Dells Valls and Stefano Ceri, Professors of Informatics
Frank van Harmelen, Vrije Universiteit Amsterdam
Dieter Fensel, University of Bremen

W

hether he is traffic jam on the highway? Can we reason over the data on the Internet? Or can we discover the changes in the news Web portal in extracting the most current? Which navigation patterns would lead users to other news related to that content? Do trends in medical records indicate any new disease spread-

The state of the art in reasoning upon changing worlds is based on temporal logic and belief revision, these are heavyweight logics, besides the data that changes is low volume as low frequency. Similarly, the problem of changing environments and evolving ontologies has undergone thorough investigation, but even the standard practices within configuration management techniques taken from software engineering, such as scalability and ontology engineering. There are outliers for ontologies that are more or less, network-based, but none have been



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 streams 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 evaluating 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

The 8 Requirements of Real-Time Stream Processing

Rule 1: Keep the Data Moving

Michael J. Computer Science
Intelligence Lab

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

Uğur Ç. Department of Computer Science
Brown University

Rule 6: Guarantee Data Safety and Availability

Stan Department of Computer Science
Brown University

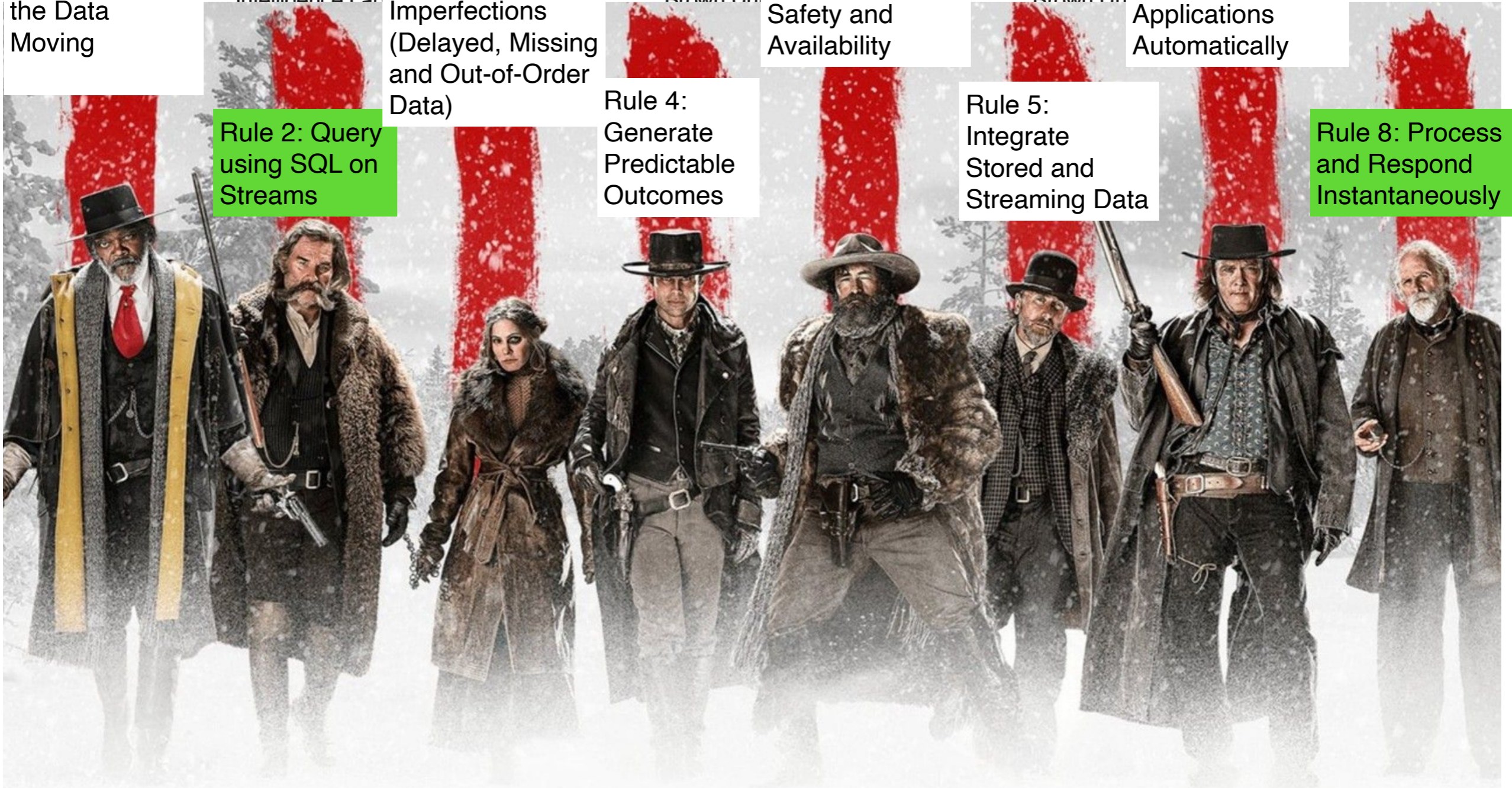
Rule 7: Partition and Scale Applications Automatically

Rule 2: Query using SQL on Streams

Rule 4: Generate Predictable Outcomes

Rule 5: Integrate Stored and Streaming Data

Rule 8: Process and Respond Instantaneously



Timeline

Precision not Recall

Models and Issues in Data Stream Systems*

Brian Babcock Shivnath Babu Mayur Datar Rajeev Motwani Jennifer Widom
Department of Computer Science
Stanford University
Stanford, CA 94305
{babcock,shivnath,datar,rajeev,widom}@cs.stanford.edu

Abstract

In this overview paper we motivate the need for and research issues arising from a new model of data processing. In this model, data does not take the form of persistent relations, but rather arrives in multiple, continuous, rapid, time-varying data streams. In addition to reviewing past work relevant to data stream systems and current projects in the area, the paper explores trends in stream query languages, new requirements and challenges in query processing, and algorithmic issues.

1 Introduction

Recently a new class of data-intensive applications has become widely recognized: applications in which the data is modeled best not as persistent relations but rather as transient *data streams*. Examples of such applications include financial applications, network monitoring, security, telecommunications data management, web applications, manufacturing, sensor networks, and others. In the data stream model, individual data items may be relational tuples, e.g., network measurements, call records, web page visits, sensor readings, and so on. However, their continuous arrival in multiple, rapid, time-varying, possibly unpredictable and unbounded streams appears to yield some fundamentally new research problems. In all of the applications cited above, it is not feasible to simply load the arriving data into a traditional database management system (DBMS) and operate on it there. Traditional DBMS's are not designed for rapid and continuous loading of individual data items, and they do not directly support the *continuous queries* [84] that are typical of data stream applications. Furthermore, it is recognized that both *approximation* [13] and *adaptivity* [8] are key ingredients in executing queries and performing other processing (e.g., data analysis and mining) over rapid data streams, while traditional DBMS's focus largely on the opposite goal of precise answers computed by stable query plans. In this paper we consider fundamental models and issues in developing a general-purpose *Data Stream Management System* (DSMS). We are developing such a system at Stanford [82], and we will touch on some of our own work in this paper. However, we also attempt to provide a general overview of the area, along with its related and current work. (Any glaring omissions are, naturally, our own fault.) We begin in Section 2 by considering the data stream model and queries over streams. In this section we

The VLDB Journal (2006) 13(2): 131-142
DOI 10.1007/s00375-006-0127-0

REGULAR PAPER

Artur Avraam · Shivnath Babu · Jennifer Widom

The CQL continuous query language: semantic foundations and query execution

Received: 1 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 stream and related 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, interpreter queries, synopsis, and sharing of components among multiple operators and queries. Examples throughout the paper are drawn from the *Linear Algebra* benchmark recently proposed for DSMS. We also contain a public repository of data stream applications that includes a wide variety of queries expressed in CQL.

12, 19, 20, 21, 28, 73. 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 related relations. CQL (for continuous query language) 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, interpreter queries, synopsis, and sharing of components among multiple operators and queries. Examples throughout the paper are drawn from the *Linear Algebra* benchmark recently proposed for DSMS. We also contain a public repository of data stream applications that includes a wide variety of queries expressed in CQL.

The 8 Requirements of Real-Time Stream Processing

Michael Stonebraker Department of Computer Science, MIT, and StreamBase Systems, Inc. stonebraker@csail.mit.edu

Ugor Cacirotel Department of Computer Science, Brown University, and StreamBase Systems, Inc. ucaci@cs.brown.edu

ABSTRACT Applications that require real-time processing of high-volume data streams 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 “run change” caused by these stream-based technology takes hold, we expect to see everything of material significance on the planet get “sense tagged” and report its state on location in real time. This revolution of the real world will lead to a “green field” of novel monitoring and control applications with high-volume and low-latency processing requirements.

Similar requirements networks for details. Real-time feed data streams 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 “run change” caused by these stream-based technology takes hold, we expect to see everything of material significance on the planet get “sense tagged” and report its state on location in real time. This revolution of the real world will lead to a “green field” of novel monitoring and control applications with high-volume and low-latency processing requirements.

The VLDB Journal (2006) 13(2): 121-142
DOI 10.1007/s00375-006-0127-0

REGULAR PAPER

Artur Avraam · Shivnath Babu · Jennifer Widom

The CQL continuous query language: semantic foundations and query execution

Received: 1 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 stream and related 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, interpreter queries, synopsis, and sharing of components among multiple operators and queries. Examples throughout the paper are drawn from the *Linear Algebra* benchmark recently proposed for DSMS. We also contain a public repository of data stream applications that includes a wide variety of queries expressed in CQL.

12, 19, 20, 21, 28, 73. 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 related relations. CQL (for continuous query language) 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, interpreter queries, synopsis, and sharing of components among multiple operators and queries. Examples throughout the paper are drawn from the *Linear Algebra* benchmark recently proposed for DSMS. We also contain a public repository of data stream applications that includes a wide variety of queries expressed in CQL.

THE SEMANTIC WEB
Frank van Harmelen, Vitor Santosuosso, and Dieter Fensel, University of Innsbruck

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

Received: 1 June 2004 / Accepted: 22 November 2004 / Published online: 22 July 2005
© Springer Verlag 2005

Abstract In this paper we explore the challenges of reasoning upon rapidly changing information. 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, interpreter queries, synopsis, and sharing of components among multiple operators and queries. Examples throughout the paper are drawn from the *Linear Algebra* benchmark recently proposed for DSMS. We also contain a public repository of data stream applications that includes a wide variety of queries expressed in CQL.

The use of the term in reasoning upon changing worlds is based on temporal logic and belief revision theory, which are key ingredients in reasoning about data that changes in low volume or low frequency. Similarly, the problem of changing requirements and evolving capabilities has undergone thorough investigation, but even the standard practice relies on configuration management techniques taken from software engineering, such as modularity and ontology engineering. There are outside for ontologies that describe our worlds, or methods based, but not

Apache Samza

Martin Kleppmann

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-Medina, Reuven Lax, Sam McVoey, Daniel Mills, Frances Perry, Eric Schmidt, Sam Whittle, Google
{akidau,robertw,chambers,chernyak,fernan,relax,sgmc,milstd,tip,clouds,samuelw}@google.com

ABSTRACT Unbounded, unordered, global-scale datasets are increasingly common in data-intensive businesses (e.g. Web logs, mobile user 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 insurmountable hunger for faster answers, MapReduce, practically dictates that one can never fully optimize along all dimensions of correctness, latency, and cost for these types of inputs. As a result, data processing practitioners are left with the quandary of how to reconcile the tensions between those seemingly competing propositions, often resulting in disparate implementations and systems. We propose that a fundamental shift of approach is necessary to deal with those evolved requirements in modern data processing. We take a first-step attempt to address 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 retraced, 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

Two Sides of the Same Coin

Guozhang Wang, Corflint Inc. Palo Alto, USA guozhang@corflint.io

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

the writes of records to a processing jobs are long-goeses that continuously the or more event streams, me application logic on producing derived output potentially writing output

ABSTRACT In a stream may become investment in such a setting. Existing models either neglect these considerations or handle them by means of data buffering and recoding techniques. In this paper, we introduce the Dual Streaming Model, a stream of data influencing the Dual Streaming Model. This model presents the result of an operator as a stream of ordered updates, which induces a stream of results and streams. As such, it provides a natural way to cope with inconsistencies between the physical and logical order of streaming data in a continuous manner, without explicit buffering and recoding. We further discuss the trade-offs and challenges faced when implementing this model in terms of correctness, latency, and processing cost. A case study based on Apache Kudu illustrates the effectiveness of our

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

Paris Carbonne, Asterics Kasifodinos, Stephan Ewen, Volker Markl, Seif Haridi, Kostas Tzoumas
{KTK & SICS Sweden paris.carbonne@kth.se first-data-artisans.com first.seif@berlin.de

Abstract Apache Flink™ is an open-source system for processing streaming and batch data. Flink is built on the philosophy that many classes of data processing applications, including real-time analytics, continuous data pipelines, historic data processing (batch), and iterative algorithms (machine learning, graph analysis) can be expressed and executed as pipeline fault-tolerant workflows. In this paper, we present Flink’s architecture and expand on how a (seemingly diverse) set of use cases can be unified under a single execution model.

1 Introduction

Data-stream processing (e.g., as exemplified by complex event processing systems) and static (batch) data processing (e.g., as exemplified by MPP databases and Hadoop) were traditionally considered as two very different types of applications. They were programmed using different programming models and APIs, and were executed by different systems (e.g., dedicated streaming systems such as Apache Storm, IBM InfoSphere Streams, Microsoft StreamInsight, or Streambase versus relational databases or execution engines for Hadoop, including Apache Spark and Apache Drill). Traditionally, batch data analysis made up for the lion’s share of the use cases, data sizes, and market, while streaming data analysis mostly served specialized applications. It is becoming more and more apparent, however, that a huge number of today’s large-scale data processing use cases handle data that is, in reality, produced continuously over time. These continuous streams of data come for example from web logs, application logs, sensors, or as changes to application state in databases (transaction log records). Rather than treating the streams as streams, today’s setups ignore the continuous and timely nature of data production. Instead, data records are (often artificially) batched into static data sets (e.g., hourly, daily, or monthly chunks) and then processed in a time-agnostic fashion. Data collection tools, workflow managers, and schedulers orchestrate the creation and processing of batches, in what is actually a continuous data processing pipeline. Architectural patterns such as the “lambda architecture” [21] combine batch and stream processing systems to implement multiple paths of computation: a streaming fast path for timely approximate results, and a batch offline path for late accurate results. All of these approaches suffer from high latency (imposed by batches),

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

Michael Armbrust, Tathagata Das, Joseph Torres, Brank Vucelja, Shiqiang Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, Matei Zaharia
Databricks Inc. Stanford University

Abstract While the advent of real-time data processing has opened up new opportunities for data-driven applications, the challenges of ingesting, processing, and analyzing data in real time are still a topic of ongoing research. In this paper, we present Structured Streaming, a new high-level API for batch and stream processing in Apache Spark. Structured Streaming is built on top of the existing Structured Query Language (SQL) engine in Apache Spark. It allows users to express streaming applications as declarative SQL queries, which are then executed by the Structured Streaming engine. Structured Streaming is designed to be fault-tolerant, scalable, and easy to use. It provides a simple and intuitive API for writing streaming applications, and it is built on top of the existing Structured Query Language (SQL) engine in Apache Spark. It allows users to express streaming applications as declarative SQL queries, which are then executed by the Structured Streaming engine. Structured Streaming is designed to be fault-tolerant, scalable, and easy to use. It provides a simple and intuitive API for writing streaming applications, and it is built on top of the existing Structured Query Language (SQL) engine in Apache Spark.

1 Introduction

High-volume data streams are a reality in our world, including sensor networks, Web-scale applications, and the Internet of Things. The challenges of ingesting, processing, and analyzing data in real time are still a topic of ongoing research. In this paper, we present Structured Streaming, a new high-level API for batch and stream processing in Apache Spark. Structured Streaming is built on top of the existing Structured Query Language (SQL) engine in Apache Spark. It allows users to express streaming applications as declarative SQL queries, which are then executed by the Structured Streaming engine. Structured Streaming is designed to be fault-tolerant, scalable, and easy to use. It provides a simple and intuitive API for writing streaming applications, and it is built on top of the existing Structured Query Language (SQL) engine in Apache Spark.

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

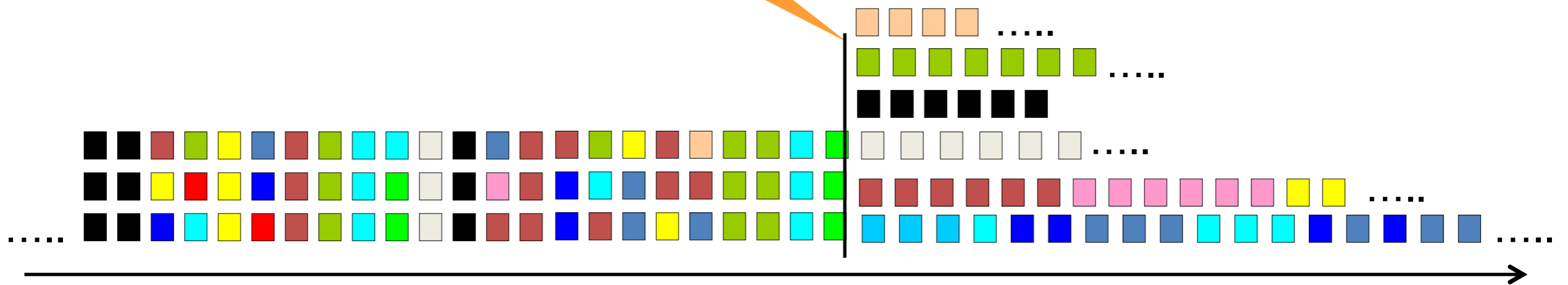
Abstract *CQL*, a *continuous query language*, is supported by the *STREAM* prototype data stream management system (DSMS) at Stanford. CQL is an expressive SQL-based declarative language for registering continuous queries against streams and stored relations. We begin by presenting an abstract semantics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

Sort out all the colours in the streams



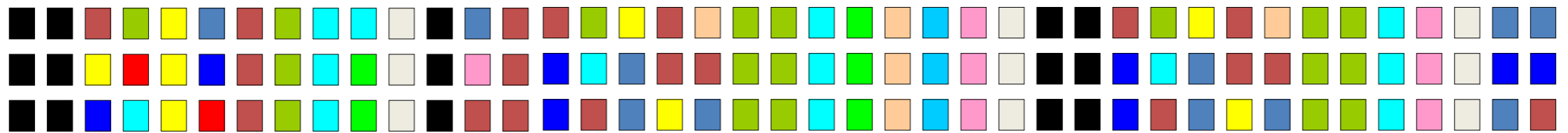
order

Continuous Querying

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

(, 15)

1 minute wide window



time

CQL in 3 Slides

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.

1. INTRODUCTION
Modern data processing is a complex and exciting field. From the early work by Magidhoo [26] and his 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, 35], windowing [22], data streams [25], time domains [28], semantic models [9]), to the more recent focus on low-latency processing such as Speed Streaming [34], MillWheel, and Storm [3], modern consumers of data wield considerable amounts of power in shaping and tuning 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 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 $\langle s, t \rangle$ 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, \dots, d_n) , where each element d_j is a member of D_j , a data domain¹.

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

CQL in ~~8~~ 4 Slides

A Stream **S** is a possibly infinite **multi-set** of elements $\langle s, t \rangle$ 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, \dots, d_n) , where each element d_j is a member of D_j , a data domain¹.~~

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

Ok, CQL in ~~4~~ 5 Slides

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

Received 12/15/2004 / Accepted 22 November 2004 / Published online: 22 July 2005
© Springer 2005

Abstract. CQL, a continuous query language, is a declarative language for specifying continuous queries over streams and stored relations. We begin by presenting the abstract semantics of CQL, which is based on the notion of a stream. We then present the 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

A Stream **S** is a possibly infinite **multi-set** of elements $\langle s, t \rangle$ 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 query execution plans as well as details of the most important components: operators, interpreter queries, synopsis, and sharing of components among multiple operators and

[2, 19, 20, 23, 28, 32]. However, these queries tend to be simple and primarily for illustrative – 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 $\langle s, t \rangle$ 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**.

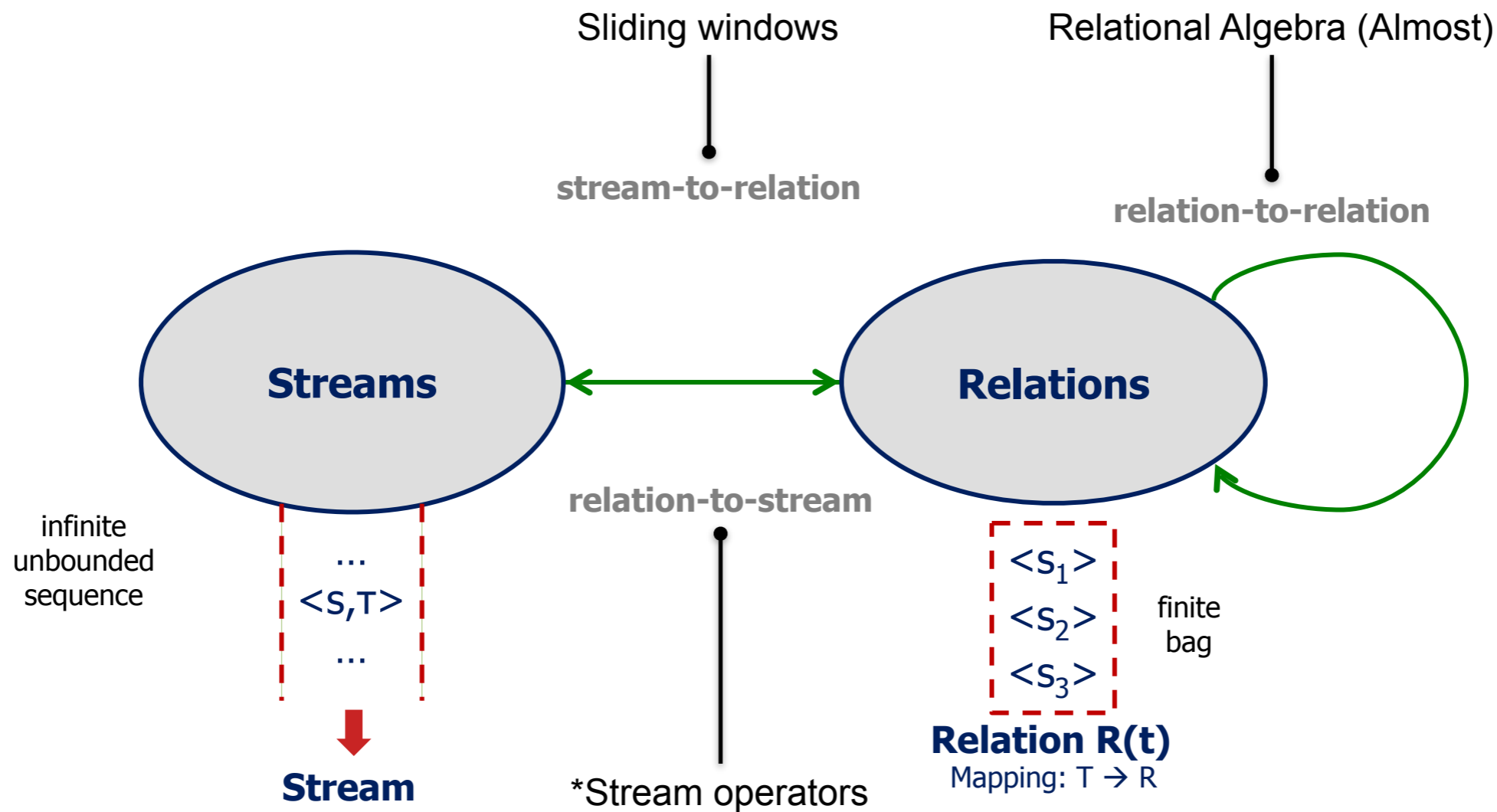
¹ a Data Domain refers to all the values which a data element may contain.

CQL in 5 Slides

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, query execution plans as well as details of the most important components: operators, interpreter queues, synopsis, and sharing of components among multiple operators and

[2, 19, 20, 23, 28, 32]. However, these queries tend to be simple and primarily for illustrative – 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



CQL in ~~5~~ 6 Slides

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]

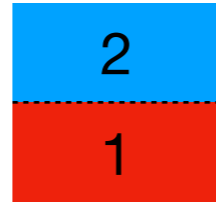
¹ a Data Domain refers to all the values which a data element may contain.

CQL in 6 Slides

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

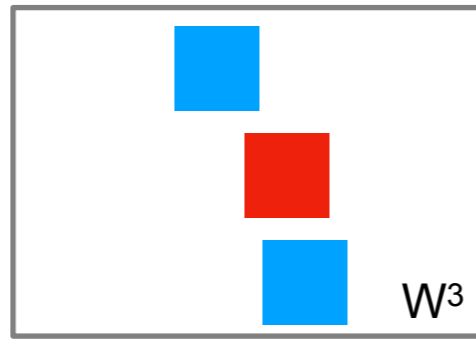
Abstract CQL, a continuous query language, is supported by the STREAM prototype data stream management system (DSMS) at Stanford. CQL is an expressive SQL-based declarative language for registering continuous queries against streams and stored relations. We begin by presenting an abstract semantics that 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 relations to streams, and three new operators to map from relations 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 query execution plans as well as details of the most important components: operators, interoperator queues, synopsis, and sharing of components among multiple operators and

[2, 19, 20, 23, 28, 32]. However, these queries tend to be simple and primarily for illustrative – a precise language 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

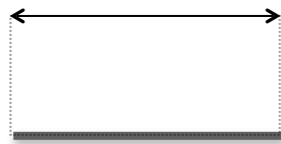


R2R operator

COUNT

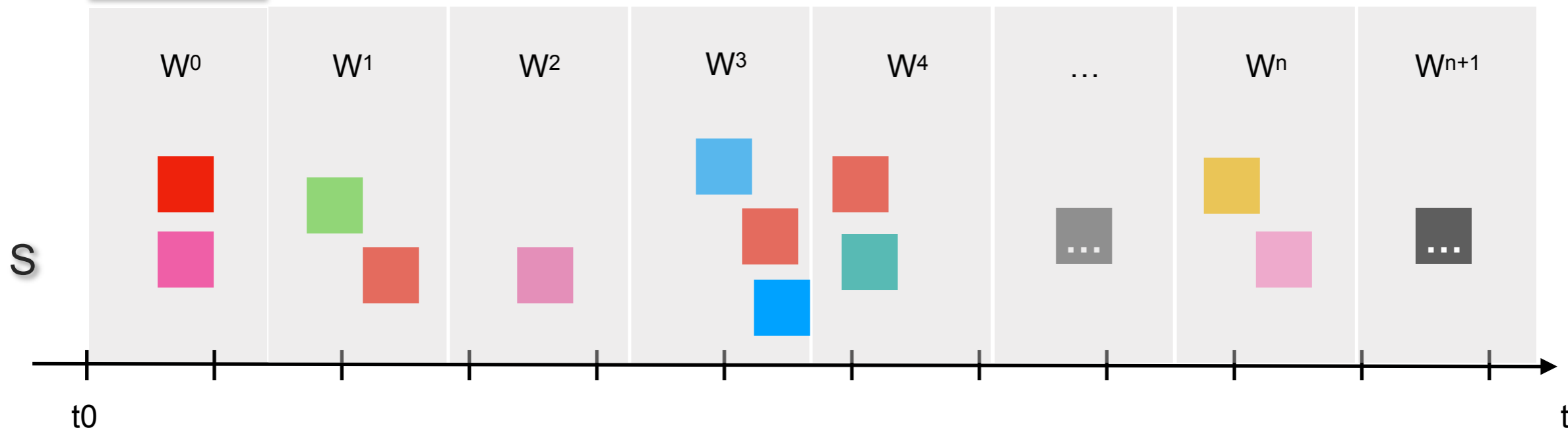
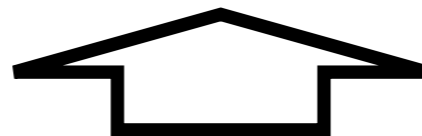


width
 ω



S2R operator

$W(\omega)$



CQL in 6 Slides

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

Models and Issues in Data Stream Systems*

Brian Babcock Shivnath Babu Mayur Datar Rajeev Motwani Jennifer Widom
 Department of Computer Science
 Stanford University
 Stanford, CA 94305
 {babcock,shivnath,datar,rajeev,widom}@cs.stanford.edu

Abstract

In this overview paper we motivate the need for and research issues arising from a new model of data processing. In this model, data does not take the form of persistent relations, but rather arrives in multiple, continuous, rapid, time-varying data streams. In addition to reviewing past work relevant to data stream systems and current projects in the area, the paper explores topics in stream query languages, new requirements and challenges in query processing, and algorithmic issues.

1 Introduction

Recently a new class of data-intensive applications has become widely recognized: applications in which the data is modeled best not as persistent relations but rather as transient *data streams*. Examples of such applications include financial applications, network monitoring, security, telecommunications data management, web applications, manufacturing, sensor networks, and others. In the data stream model, individual data items may be relational tuples, e.g., network measurements, call records, web page visits, sensor readings, and so on. However, their continuous arrival in multiple, rapid, time-varying, possibly unpredictable and unbounded streams appears to yield some fundamentally new research problems. In all of the applications cited above, it is not feasible to simply load the arriving data into a traditional database management system (DBMS) and operate on it there. Traditional DBMS's are not designed for rapid and continuous loading of individual data items, and they do not directly support the *continuous queries* [84] that are typical of data stream applications. Furthermore, it is recognized that both *approximation* [13] and *adaptivity* [8] are key ingredients in executing queries and performing other processing (e.g., data analysis and mining) over rapid data streams, while traditional DBMS's focus largely on the opposite goal of precise answers computed by stable query plans. In this paper we consider fundamental models and issues in developing a general-purpose *Data Stream Management System* (DSMS). We are developing such a system at Stanford [82], and we will touch on some of our own work in this paper. However, we also attempt to provide a general overview of the area, along with its related and current work. (Any glaring omissions are, naturally, our own fault.) We begin in Section 2 by considering the data stream model and queries over streams. In this section we

Apache Samza

Martin Kleppmann

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-Métezuma, Reuven Livni, Sam McVoey, Daniel Mills, Frances Perry, Eric Schmidt, Sam Whittle

{akidau,robertw,chambers,chernyak,fernan,relax,sgmc,milstd,tip,clouds,sarnuelw}@google.com

Dynamically Scaling of S

2015 IEEE First International C

Submitted 11/09, Published 4/10

Jan Sipke van der Veen^{1,2}, Bram van de

University of

{jan_sipke,vanderveen,bram.van

Abstract—Stream processing platform allow applica

ye incoming data continuously. Several use cases ex

one of these capabilities, ranging from monitoring

infrastructure to pre-selecting video surveillance foot

inspection. It is difficult to predict how much compari

are needed for these stream processing platform, m

volume and velocity of input data may vary we

open source Apache Storm software provides a fra

developers to build processing applications that i

ping resources of all machines within an estate

because of the varying processing needs of such

the platform should be able to automatically grow

as needed. Unfortunately, the current Storm platfo

provide this capability. In this paper we describ

and implementation of a tool that monitors stre

Storm platform, the applications running on it

external systems such as queues and databases. B

information, the tool decides whether extra server

or shutdown must be decommissioned from the st

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 immediate 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 those seemingly competing propositions, often resulting in disparate implementations and systems.

We propose that a fundamental shift of perspective is necessary to deal with those evolved requirements in modern data processing. We do a field-study step trying to grok 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 retraced, 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

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 [26], 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 would remarkable amounts of power in shaping and tailoring 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 operations 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

the writes of records to a

processing jobs are long-

me application logic on

producing derived output

potentially writing output

in a stream may become investment in such a setting. Ex-

isting models either neglect these considerations or handle

them by means of data buffering and recirculating techniques

thereby compromising processing latency.

In this paper, we introduce the Dataflow Model, a stream of data referring to the Dual Streaming Model

a reason of business updates, which induces a steady flow of results and streams. As such, it provides a natural way to cope

with inconsistencies between the physical and logical order of streaming data in a continuous manner, without explicit buffering and reordering. We further discuss the trade-offs and challenges faced when implementing this model in terms

of correctness, latency, and processing cost. A case study based on Apache Kudu illustrates the effectiveness of our

o Sides of the Same C

Guozhang Wang
 Corflint Inc.
 Palo Alto, USA
 guozhang@corflint.io

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

KEYWORDS

Stream Processing, Processing Model, Semantic Model

ACM Reference Format

Matthew J. Sax, Guozhang Wang, Matthias Weidner, Christoph Freytag, 2015, Streams and Tables: Two Sides of the Same Coin. In International Workshop on Real Time Data Analytics (STREAM '15), August 27, 2015, Rio de Janeiro, Brazil. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/2742111>.

1 INTRODUCTION

Stream processing has emerged as a parallel real-time over distributed systems of data, enabling processing of large-scale data in a continuous manner. As such, the stream processing paradigm has particularly suited to support the requirements of new logic in large organizations. It provides a framework for communication between independent of large systems, a.k.a. "microservices", through message-passing [19].

REGULAR PAPER

Arvid Avraam · 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 stream and related 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, interpreter queues, synopsis, and sharing of components among multiple operators and queries. Examples throughout the paper are drawn from the Linear Algebra benchmark recently proposed for DSMS. We also contain a public repository of data stream applications that include a wide variety of queries expressed in CQL.

Keywords data streams, classification, ensemble methods, java, machine learning software

MOA: Massive Online Analysis

Albert Bifet
 Geoff Holmes
 Richard Kirkby
 Bernhard Pfahringer
 Department of Computer Science
 University of Waikato
 Hamilton, New Zealand

Editor: Mikko Brun

Submitted 11/09, Published 4/10

Journal of Machine Learning Research 11 (2010) 1401–1404

Abstract

Massive Online Analysis (MOA) is a software environment for implementing algorithms and running experiments for online learning from evolving data streams. MOA includes a collection of offline and online methods as well as tools for evaluation. In particular, it implements boosting, bagging, and Hoefding Trees, all with and without Naïve Bayes classifiers at the leaves. MOA supports bidirectional interaction with WEKA, the Waikato Environment for Knowledge Analysis, and is released under the GNU GPL license.

Keywords: data streams, classification, ensemble methods, java, machine learning software

1. Introduction

Green computing is the study and practice of using computing resources efficiently. A main approach to green computing is based on algorithmic efficiency. In the data stream model, data arrive at high speed, and an algorithm must process them under very strict constraints of space and time.

MOA is an open-source framework for dealing with massive evolving data streams. MOA is related to WEKA, the Waikato Environment for Knowledge Analysis, which is an award-winning open-source webtool containing implementations of a wide range of batch machine learning methods.

A data stream environment has different requirements from the traditional batch learning setting. The most significant are the following:

Requirement 1 Process an example at a time, and inspect it only once (at most)

Requirement 2 Use a limited amount of memory

Requirement 3 Work in a limited amount of time

THE 8 REQUIREMENTS OF REAL-TIME STREAM PROCESSING

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

Ugor Centimel
 Department of Computer Science, Brown University, and StreamBase Systems, Inc.
 uc@cs.csbrown.edu

Stan Zdonik
 The VLDB Journal (2006) 15(2): 1–142
 DOI 10.1007/s00375-006-0125-2

REGULAR PAPER

Arvid Avraam · 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 stream and related 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, interpreter queues, synopsis, and sharing of components among multiple operators and queries. Examples throughout the paper are drawn from the Linear Algebra benchmark recently proposed for DSMS. We also contain a public repository of data stream applications that include a wide variety of queries expressed in CQL.

Keywords data streams, classification, ensemble methods, java, machine learning software

THE SEMANTIC WEB

Enrico Della Valle and Stefano Ceri, *Professione di Milano*
 Frank van Harmelen, *Vrije Universiteit Amsterdam*
 Dieter Fensel, *University of Innsbruck*

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

Received 2005-05-01, Accepted 2005-05-01, Published 2005-05-01

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 stream and related 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, interpreter queues, synopsis, and sharing of components among multiple operators and queries. Examples throughout the paper are drawn from the Linear Algebra benchmark recently proposed for DSMS. We also contain a public repository of data stream applications that include a wide variety of queries expressed in CQL.

Keywords data streams, classification, ensemble methods, java, machine learning software

Timeline

Precision not Recall

Models and Issues in Data Stream Systems*

Brian Babcock Shivnath Babu Mayur Datar Rajeev Motwani Jennifer Widom
 Department of Computer Science
 Stanford University
 Stanford, CA 94305
 {babcock,shivnath,datar,rajeev,widom}@cs.stanford.edu

Abstract

In this overview paper we motivate the need for and research issues arising from a new model of data processing. In this model, data does not take the form of persistent relations, but rather arrives in multiple, continuous, rapid, time-varying data streams. In addition to reviewing past work relevant to data stream systems and current projects in the area, the paper explores topics in stream query languages, new requirements and challenges in query processing, and algorithmic issues.

1 Introduction

Recently a new class of data-intensive applications has become widely recognized: applications in which the data is modeled best not as persistent relations but rather as transient *data streams*. Examples of such applications include financial applications, network monitoring, security, telecommunications data management, web applications, manufacturing, sensor networks, and others. In the data stream model, individual data items may be relational tuples, e.g., network measurements, call records, web page visits, sensor readings, and so on. However, their continuous arrival in multiple, rapid, time-varying, possibly unpredictable and unbounded streams appears to yield some fundamentally new research problems. In all of the applications cited above, it is not feasible to simply load the arriving data into a traditional database management system (DBMS) and operate on it there. Traditional DBMS's are not designed for rapid and continuous loading of individual data items, and they do not directly support the *continuous queries* [84] that are typical of data stream applications. Furthermore, it is recognized that both *approximation* [13] and *adaptivity* [8] are key ingredients in executing queries and performing other processing (e.g., data analysis and mining) over rapid data streams, while traditional DBMS's focus largely on the opposite goal of precise answers computed by stable query plans.

In this paper we consider fundamental models and issues in developing a general-purpose *Data Stream Management System (DSMS)*. We are developing such a system at Stanford [82], and we will touch on some of our own work in this paper. However, we also attempt to provide a general overview of the area, along with its related and current work. (Any glaring omissions are, naturally, our own fault.) We begin in Section 2 by considering the data stream model and queries over streams. In this section we

REGULAR PAPER

The VLDB Journal (2006) 13(2): 173-187
 DOI 10.1007/s00778-006-0127-2

Arvid Arasu Shivnath Babu Jennifer Widom

The CQL continuous query language: semantic foundations and query execution

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

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

Big Stream Processing Starts



The 8 Requirements of Real-time Analytics

Michael Stonebraker
 Department of Computer Science
 Massachusetts Institute of Technology
 Cambridge, MA 02139
 stonebraker@csail.mit.edu

Abstract Applications that require real-time processing of high-volume data streams 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 "run-shops" caused by these stream-based technology take hold, we expect to see everything of material significance on the planet get "sense-tagged" and report its state on location in real time. This revolution 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 real-time systems, are also being "re-purposed" by marketing departments to address these applications. In this paper, we outline eight requirements that a system software should meet to meet a variety of real-time stream processing applications. Our goal is to provide high-level guidance to infrastructure technologists so that they will know what to look for when evaluating alternative stream processing solutions. As such,

Journal of Machine Learning Research 11 (2010) 1401-1404 Submitted 1/10, Published 4/10

MOA: Massive Online Analysis

Albert Bifet Geoff Holmes Bernhard Pfahringer
 Department of Computer Science
 University of Waikato
 Hamilton, New Zealand

Editor: Mikko Bleum

Abstract Massive Online Analysis (MOA) is a software framework for conducting experiments for online learning from streaming data. It includes a set of offline and online methods as well as tools for visualization, debugging, and Hoefding Trees, all of which are implemented in a user-friendly environment for Knowledge Analysis. MOA is released under the Creative Commons Attribution-NonCommercial license.

Keywords: data streams, classification, regression, machine learning software

Abstract In this paper we describe a new practice of using computing resources efficiently. A main aspect of algorithmic efficiency in the data stream model, data arrive must process them under very strict constraints of space and time. A network for dealing with massive evolving data streams. MOA is Environment for Knowledge Analysis, which is an award-winning implementation of a wide range of batch machine learning algorithms. MOA is designed to be used as a different requirements from the traditional batch learning setting. MOA is designed to be used as a different requirements from the traditional batch learning setting. MOA is designed to be used as a different requirements from the traditional batch learning setting.

Abstract In this paper we describe a new practice of using computing resources efficiently. A main aspect of algorithmic efficiency in the data stream model, data arrive must process them under very strict constraints of space and time. A network for dealing with massive evolving data streams. MOA is Environment for Knowledge Analysis, which is an award-winning implementation of a wide range of batch machine learning algorithms. MOA is released under the Creative Commons Attribution-NonCommercial license.

Apache Samza

Martin Kleppmann

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

Tyler Akkida, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Medezama, Reuben Lax, Sam McVoey, Daniel Mills, Frances Perry, Eric Schmidt, Sam Whitte

Abstract 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 [26], 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 would remarkable amounts of power in shaping and tuning 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 wealth 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

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

Paris Carbonne*, Stephan Ewen†, Seif Haridi†, Asterios Katsifodimos†, Volker Markl†, Kostas Tzoumas†

Abstract Apache Flink™ is an open-source system for processing streaming and batch data. Flink is built on the philosophy that many classes of data processing applications, including real-time analytics, continuous data pipelines, historic data processing (batch), and iterative algorithms (machine learning, graph analysis) can be expressed and executed as pipelined fault-tolerant dataflows. In this paper, we present Flink's architecture and expand on how a (seemingly diverse) set of use cases can be unified under a single execution model.

1 Introduction Data-stream processing (e.g., as exemplified by complex event processing systems) and static (batch) data processing (e.g., as exemplified by MPP databases and Hadoop) were traditionally considered as two very different types of applications. They were programmed using different programming models and APIs, and were executed by different systems (e.g., dedicated streaming systems such as Apache Storm, IBM InfoSphere Streams, Microsoft StreamInsight, or Streambase versus relational databases or execution engines for Hadoop, including Apache Spark and Apache Drill). Traditionally, batch data analysis made up for the lion's share of the use cases, data sizes, and markets, while streaming data analysis mostly served specialized applications. It is becoming more and more apparent, however, that a huge number of today's large-scale data processing use cases handle data that is, in reality, produced continuously over time. These continuous streams of data come for example from web logs, application logs, sensors, or as changes to application state in databases (transaction log records). Rather than treating the streams as streams, today's setups ignore the continuous and timely nature of data production. Instead, data records are (often artificially) batched into static data sets (e.g., hourly, daily, or monthly chunks) and then processed in a time-agnostic fashion. Data collection tools, workflow managers, and schedulers orchestrate the creation and processing of batches, in what is actually a continuous data processing pipeline. Architectural patterns such as the "lambda architecture" [21] combine batch and stream processing systems to implement multiple paths of computation: a streaming fast path for timely approximate results, and a batch offline path for late accurate results. All these approaches suffer from high latency (imposed by batches),

Two Sides of the Same Coin

Guozhang Wang
 Guozhang Wang
 Palo Alto, USA
 guozhang@cs.stanford.edu

Abstract In this paper we describe a new practice of using computing resources efficiently. A main aspect of algorithmic efficiency in the data stream model, data arrive must process them under very strict constraints of space and time. A network for dealing with massive evolving data streams. MOA is Environment for Knowledge Analysis, which is an award-winning implementation of a wide range of batch machine learning algorithms. MOA is released under the Creative Commons Attribution-NonCommercial license.

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

Michael Armbrust*, Tathagata Das†, Joseph Torres†, Burkhard Teyer†, Shiqiang Zhu†, Reynold Xin, Ali Ghodsi†, Ion Stoica*, Matei Zaharia*

Abstract With the advent of real-time data processing, the need for a single engine that can handle both batch and streaming data has become increasingly clear. Apache Spark is a distributed system that can handle both batch and streaming data. In this paper, we present Structured Streaming, a declarative API for real-time applications in Apache Spark. Structured Streaming is built on top of Apache Spark and provides a simple and intuitive way to write real-time applications. It is designed to be used as a different requirements from the traditional batch learning setting. MOA is designed to be used as a different requirements from the traditional batch learning setting. MOA is designed to be used as a different requirements from the traditional batch learning setting.

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

Matthias J. Sax*

Arvind Arasu · Shivnath B

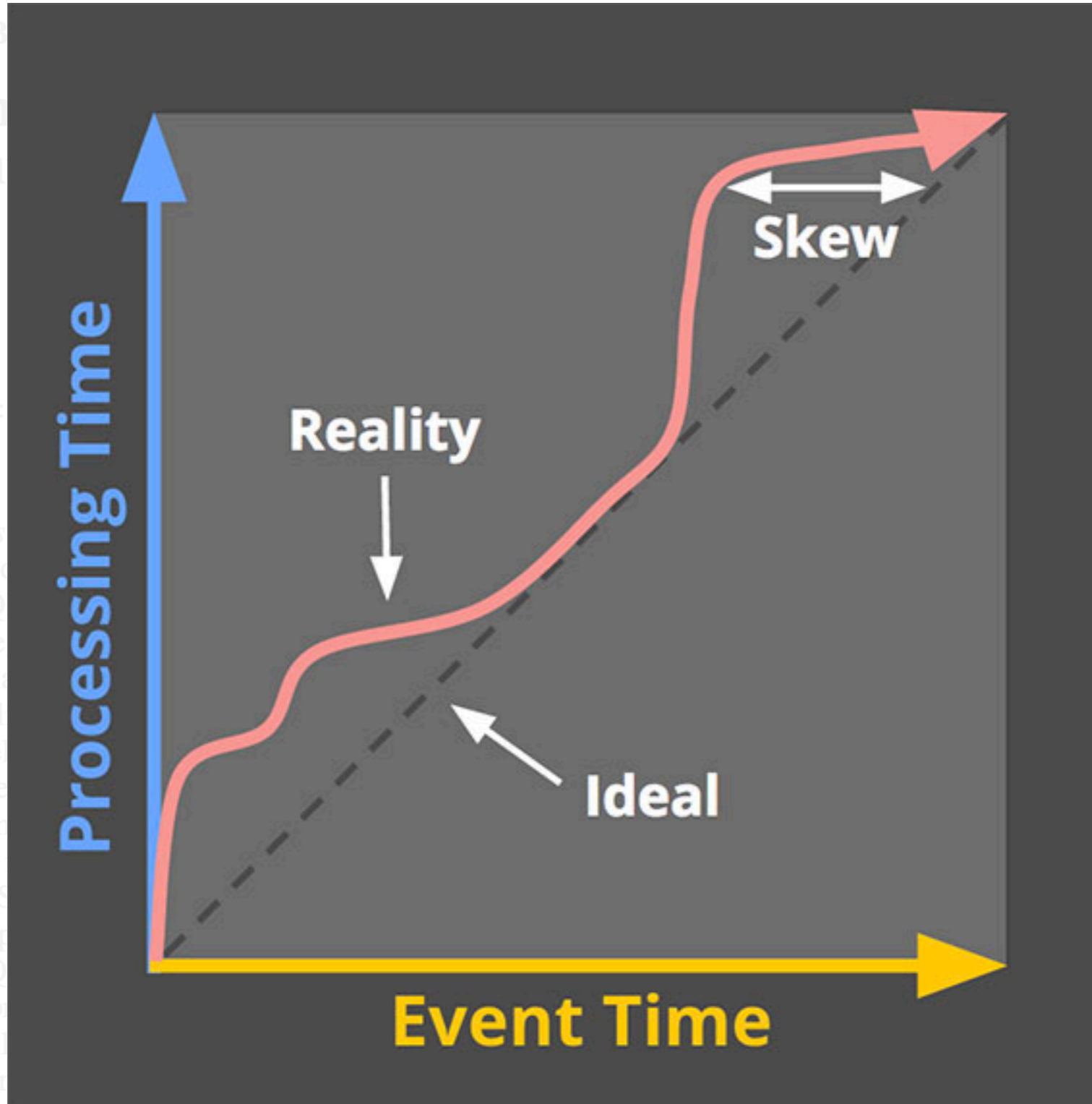
Tyler Akidau, Robert Bradsh
el J. Fernández-Moctezuma
Frances Perry, I

The CQL contin
and query execu

Received: 7 June 2004 / Accepted:
© Springer-Verlag 2005

Abstract *CQL*, a continuous
by the STREAM prototype
tem (DSMS) at Stanford. CQ
declarative language for re
against streams and stored rel
an abstract semantics that rel
plings among streams and rel
we define a precise and gene
ous queries. CQL is an instan
tics using SQL to map from
specifications derived from S
to relations, and three new op
to streams. Most of the CQ
the STREAM system. We pr
query execution plans as well
tant components: operators, in
and sharing of components among
queries. Examples throughout the paper

Linear Read benchmark recently propo
model in the light of real-world requirements



based on Apache Kafka illustrates the effectiveness o
model in the light of real-world requirements

{takidau, robertwb, c
relax, sgmc, millsd, fjp

red, global-scale datasets are incre
-to-day business (e.g. Web logs, mo
sensor networks). At the same ti
datasets have evolved sophisticated
event-time ordering and windowing
hemselves, in addition to an insati
wers. Meanwhile, practicality dict
lly optimize along all dimensions of
l cost for these types of input. As a
practitioners are left with the quanc
the tensions between these seemin
ons, often resulting in disparate in
ems.

A fundamental shift of approach is
these evolved requirements in mod
as a field must stop trying to groom
o finite pools of information that e
ete, and instead live and breathe un
we will never know if or when we h
only that new data will arrive, old c
nd the only way to make this prob
ripled abstractions that allow the p
appropriate tradeoffs along the axe

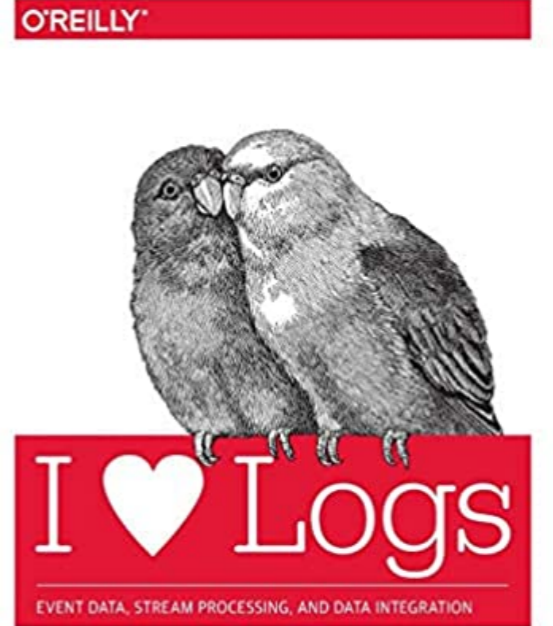
interest: correctness, latency, and cost

Timeline

Precision not Recall

Apache Samza

Martin Kleppmann



ables: Two Sides of the
Guozhi
Cun
Palo
guozhang
Johann-Ch
Humboldt
Berlin
freytag@info
KEYWORDS
Stream Processing, Prec
of operations
ling the semantics
the presence of the
Con. In International
and Analytic (BI) '08,
ACM New York, USA, 1
2412153

words to a
are long-
continuously
streams,
logic on
ed output

In this paper, we introduce the Dual Streaming Model to reason about physical and logical order in data stream processing. This model presents the result of an operator as a stream of successive updates, which induces a duality of results and streams. As such, it provides a natural way to cope with inconsistencies between the physical and logical order of streaming data in a continuous manner, without explicit buffering and reordering. We further discuss the trade-offs and challenges faced when implementing this model in terms of correctness, latency, and processing cost. A case study based on Apache Kafka illustrates the effectiveness of our

Models and Issues in Data Stream Systems*

Brian Babcock Shivnath Babu Mayur Datar Rajeev Motwani Jennifer Widom
Department of Computer Science
Stanford University
Stanford, CA 94305
{babcock,shivnath,datar,rajeev,widom}@cs.stanford.edu

Abstract
In this overview paper we motivate the need for and research issues arising from a new model of data processing. In this model, data does not take the form of persistent relations, but rather arrives in multiple, continuous, rapid, time-varying data streams. In addition to reviewing past work relevant to data stream systems and current projects in the area, the paper explores topics in stream query languages, new requirements and challenges in query processing, and algorithmic issues.

1 Introduction

Recently a new class of data-intensive applications has become widely recognized: applications in which the data is modeled best not as persistent relations but rather as transient *data streams*. Examples of such applications include financial applications, network monitoring, security, telecommunications data management, web applications, manufacturing, sensor networks, and others. In the data stream model, individual data items may be relational tuples, e.g., network measurements, call records, web page visits, sensor readings, and so on. However, their continuous arrival in multiple, rapid, time-varying, possibly unpredictable and unbounded streams appears to yield some fundamentally new research problems. In all of the applications cited above, it is not feasible to simply load the arriving data into a traditional database management system (DBMS) and operate on it there. Traditional DBMS's are not designed for rapid and continuous loading of individual data items, and they do not directly support the *continuous queries* [84] that are typical of data stream applications. Furthermore, it is recognized that both *approximation* [13] and *adaptivity* [8] are key ingredients in executing queries and performing other processing (e.g., data analysis and mining) over rapid data streams, while traditional DBMS's focus largely on the opposite goal of precise answers computed by stable query plans. In this paper we consider fundamental models and issues in developing a general-purpose *Data Stream Management System (DSMS)*. We are developing such a system at Stanford [82], and we will touch on some of our own work in this paper. However, we also attempt to provide a general overview of the area, along with its related and current work. (Any glaring omissions are, naturally, our own fault.) We begin in Section 2 by considering the data stream model and queries over streams. In this section we

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 as abstract set algebra expressions. We define a primitive query language for continuous queries, and the STREAM query execution component and describing continuous queries. Stream Linear Road's data center #1 that includes 1. This release is

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
Ugur Cetintemel Department of Computer Science, Brown University, and StreamBase Systems, Inc. ucg@cs.brown.edu
Stan Zdonik Department of Computer Science, Brown University, and StreamBase Systems, Inc. szdz@cs.brown.edu

ABSTRACT 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, mining from oil refineries to core fabric factories, will give rise to such "hotspot" data volumes and sub-second latency requirements. There is a "sea change" arising from the advances in micro-sensor technology. Although RFID has gotten the most press recently, there are a variety of other technologies with various price points, capabilities, and footprints (e.g., new II and LoRa). On a real-time, this sea change will cause everything of material significance to be sensorized to report in 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 one is out of 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 recreation of lost



Big Stream Processing Starts

MOA: Massive Online Analysis
Albert Bifet Geoff Holmes Richard Kirkby Bernhard Pfahringer Department of Computer Science University of Waikato Hamilton, New Zealand
Editor: Mikko Braum

Abstract Massive Online Analysis (MOA) is a software platform for conducting experiments for online learning from streaming data. It provides a rich set of offline and online methods as well as tool support for visualization, debugging, and Hoefding Trees, all of which are implemented in the MOA framework. MOA supports bi-directional interaction between the user and the system. It is released under the GPL. **Keywords:** data streams, classification, machine learning software

1 Introduction Data-stream processing (e.g., as exemplified by MPP databases and Hadoop) were traditionally considered as two very different types of applications. They were programmed using different programming models and APIs, and were executed by different systems (e.g., dedicated streaming systems such as Apache Storm, IBM InfoSphere Streams, Microsoft StreamInsight, or StreamBase Apache Spark and Apache Drill). Traditionally, batch data analysis made up for the lion's share of the use cases, data sizes, and market, while streaming data analysis mostly served specialized applications. It is becoming more and more apparent, however, that a huge number of today's large-scale data processing use cases handle data that is, in reality, produced continuously over time. These continuous streams of data come for example from web logs, applications, sensors, or as changes to application state in databases (transaction logs records). Rather than treating these streams as streams, today's setups ignore the continuous and timely nature of data production. Instead, data records are (often artificially) batched into static data sets (e.g., hourly, daily, or monthly chunks) and then processed in a time-agnostic fashion. Data collection tools, workflow managers, and schedulers orchestrate the creation and processing of batches, in what is actually a continuous data processing pipeline. Architectural patterns and systems to implement multiple paths of computation: a streaming fast path for timely approximate results, and a batch offline path for late accurate results. All these approaches suffer from high latency (imposed by batches),

2015 IEEE First International Conference on Data Stream Management
Dynamically Scaling of S
Jan Sipke van der Veen^{1,2}, Bram van de
1University of
(jan_sipke.vanderveen, bram.vand

Abstract—Stream processing platforms allow application developers to build processing applications that integrate the processing of all machines within an established. Because of the varying processing needs of such the platform should be able to automatically grow as needed. Unfortunately, the current Stream platforms provide little capability. In this paper we describe and implementation of a tool that monitors servers the Stream platform, the applications running on it, external systems such as queues and databases. Based on this information, the tool decides whether extra servers are needed and automatically provisions them from the cloud.

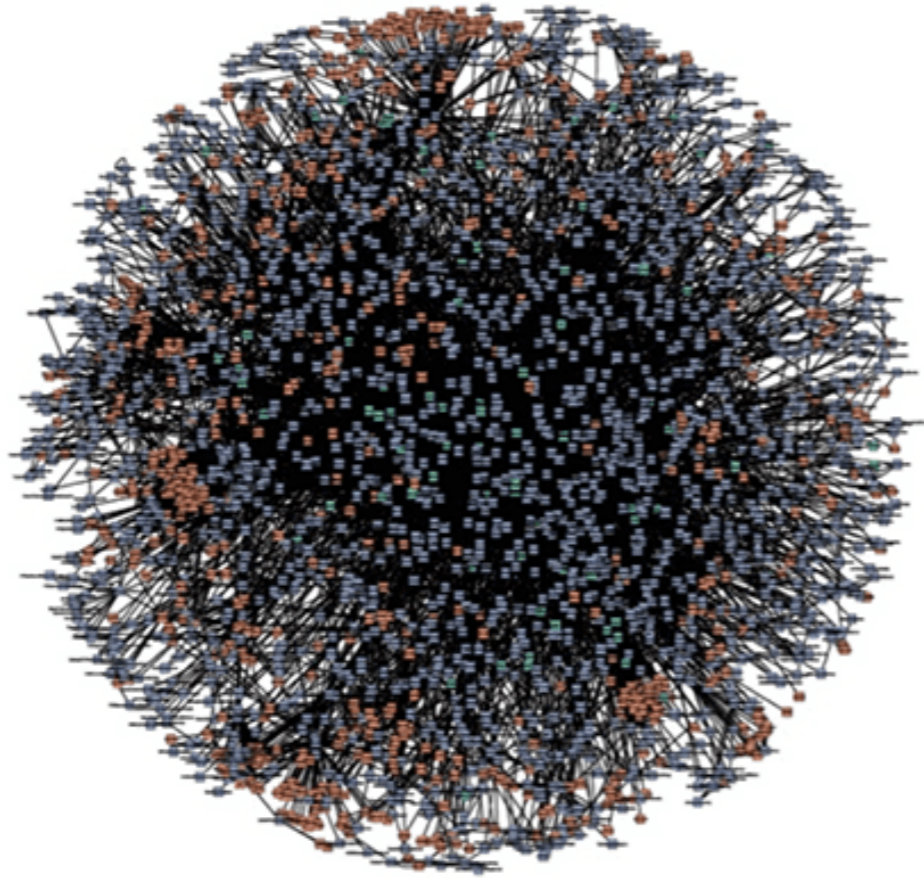
1 Introduction Data-stream processing (e.g., as exemplified by MPP databases and Hadoop) were traditionally considered as two very different types of applications. They were programmed using different programming models and APIs, and were executed by different systems (e.g., dedicated streaming systems such as Apache Storm, IBM InfoSphere Streams, Microsoft StreamInsight, or StreamBase Apache Spark and Apache Drill). Traditionally, batch data analysis made up for the lion's share of the use cases, data sizes, and market, while streaming data analysis mostly served specialized applications. It is becoming more and more apparent, however, that a huge number of today's large-scale data processing use cases handle data that is, in reality, produced continuously over time. These continuous streams of data come for example from web logs, applications, sensors, or as changes to application state in databases (transaction logs records). Rather than treating these streams as streams, today's setups ignore the continuous and timely nature of data production. Instead, data records are (often artificially) batched into static data sets (e.g., hourly, daily, or monthly chunks) and then processed in a time-agnostic fashion. Data collection tools, workflow managers, and schedulers orchestrate the creation and processing of batches, in what is actually a continuous data processing pipeline. Architectural patterns and systems to implement multiple paths of computation: a streaming fast path for timely approximate results, and a batch offline path for late accurate results. All these approaches suffer from high latency (imposed by batches),

Jay Kreps

Streaming: A Declarative API for Real-Time Applications in Apache Spark
Rustam, Tarhagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, Matej Zaharia
Stanford University

Abstract Streaming is a new high-level API for stream processing that we developed in Apache Spark starting in 2016. Stream processing builds on more ideas in recent stream processing systems, such as relational processing time from event time and triggers in Google Dataflow [2], using a relational execution engine for performance [12], and declarative language integrated API [11, 17]. But aims to make them simpler to use and integrated with the rest of Apache Spark. Specifically, Stream processing differs from other widely used open source streaming APIs in two ways. **Incremental query models:** Stream processing incrementally recomputes the results of a query as new data is processed through Spark's SQL and Dataframe APIs [16], meaning that users typically only need to understand Spark's batch APIs to write a streaming query. Query time concepts are especially easy to express and understand in this model. Although incremental query execution and view maintenance are well studied [1, 14, 28, 36], we believe Stream processing is the first effort to make them in a widely used open source system. We found that this incremental API naturally worked well for both batch and streaming users. For example, advanced users can use a set of powerful processing operators (e.g., stateful operators, windowing, sliding window, custom logic while fitting into the incremental model). **Support for end-to-end applications:** Stream processing's API and batch integration make it easy to write code that is "born by default" when interacting with external systems and

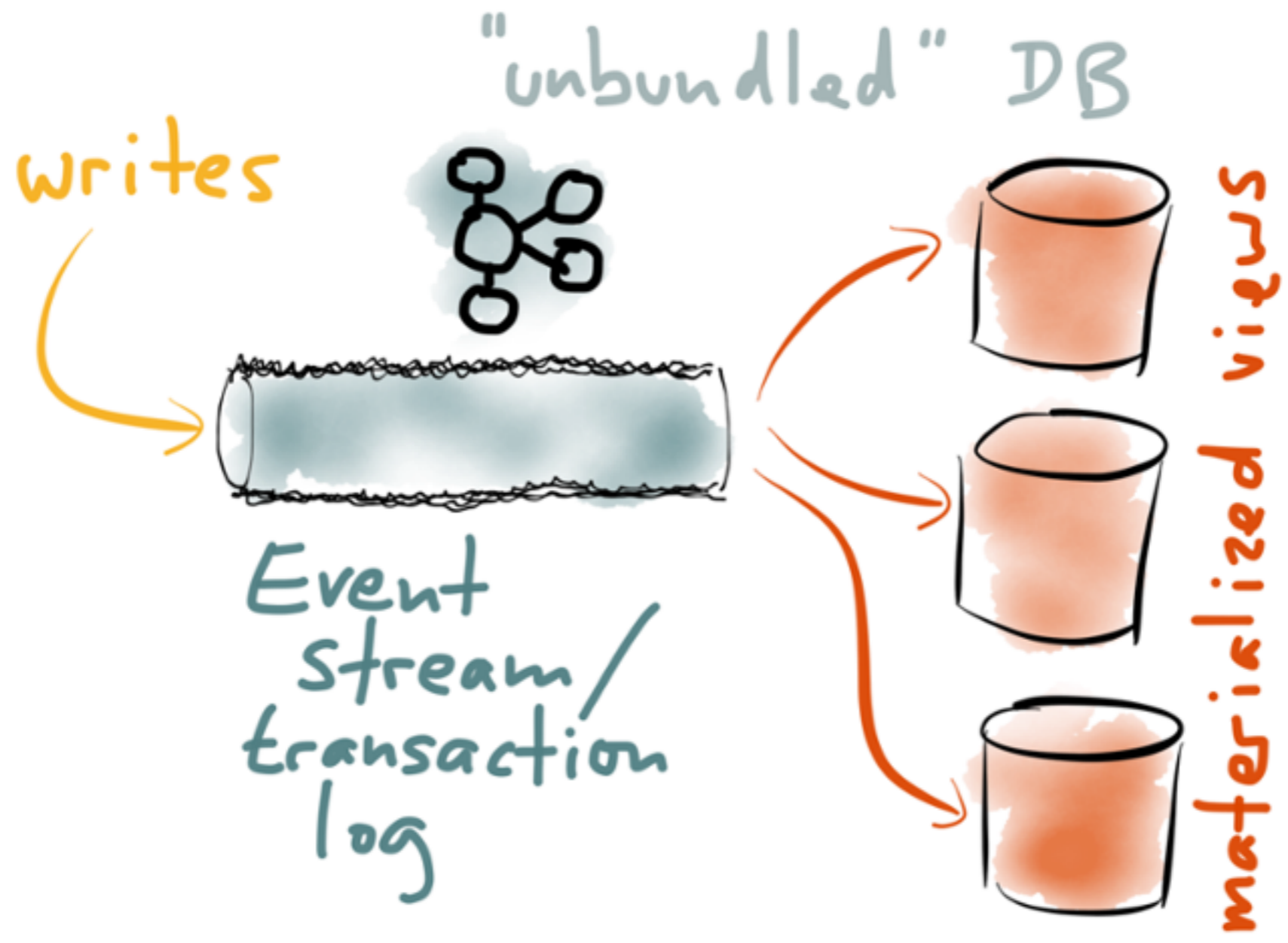




amazon.com®

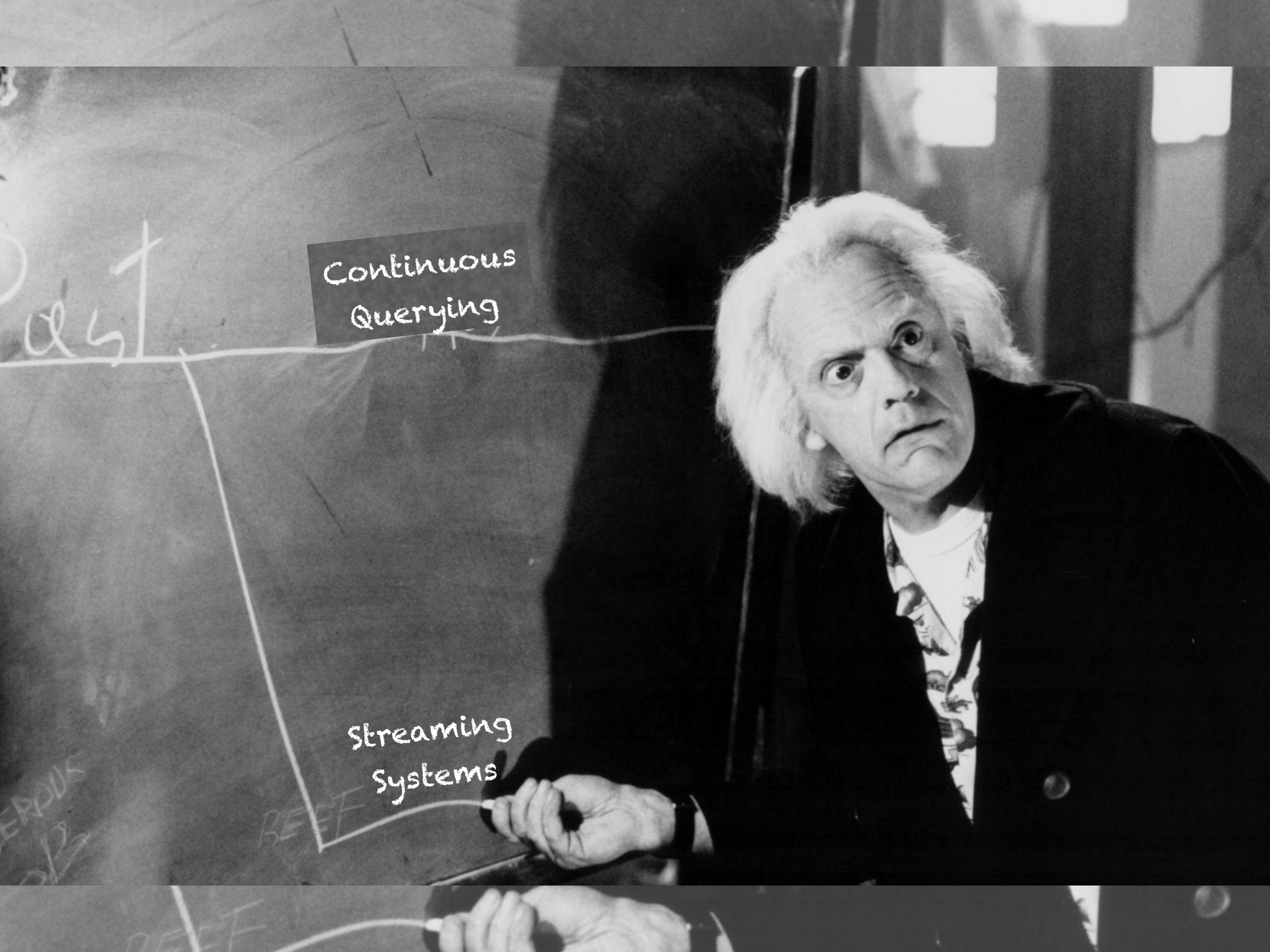


NETFLIX



Continuous
Querying

Streaming
Systems



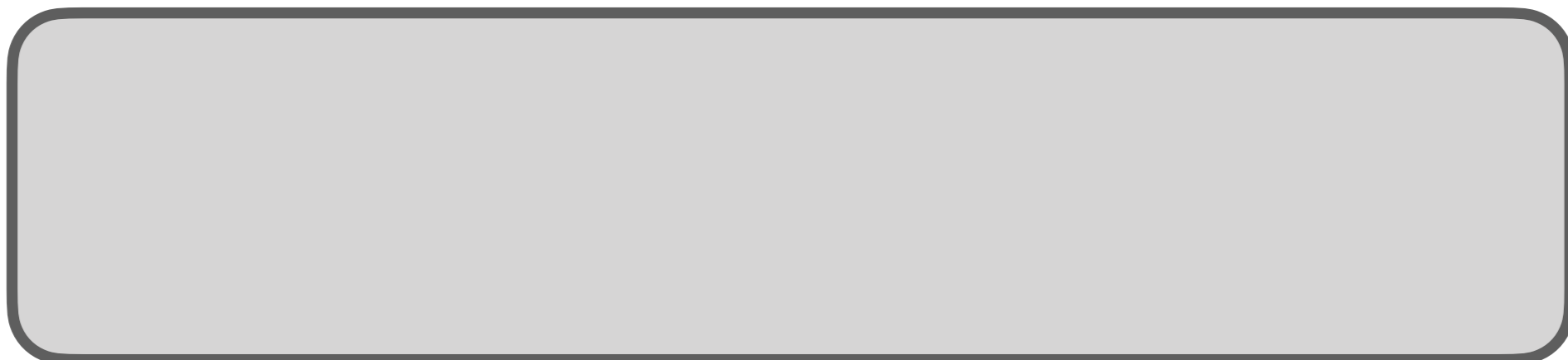
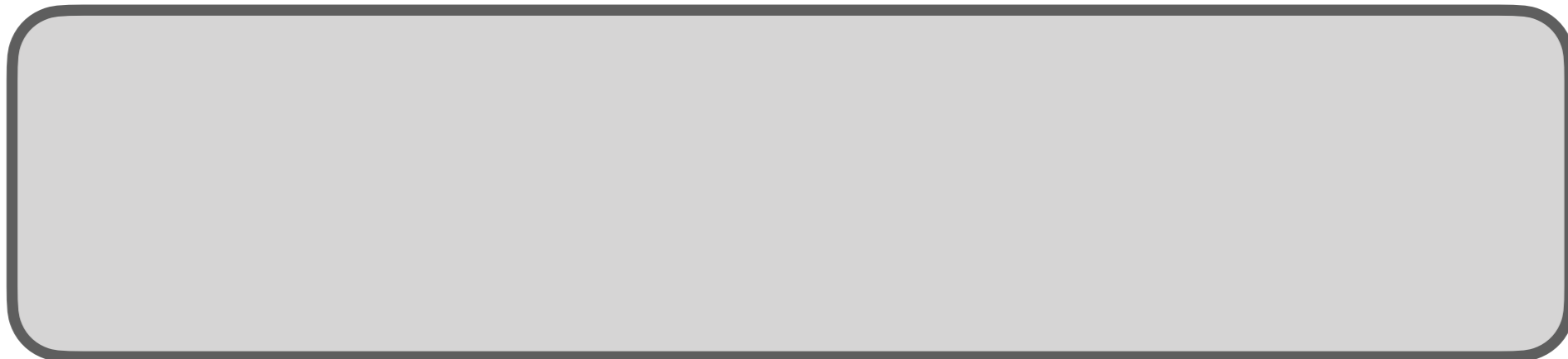
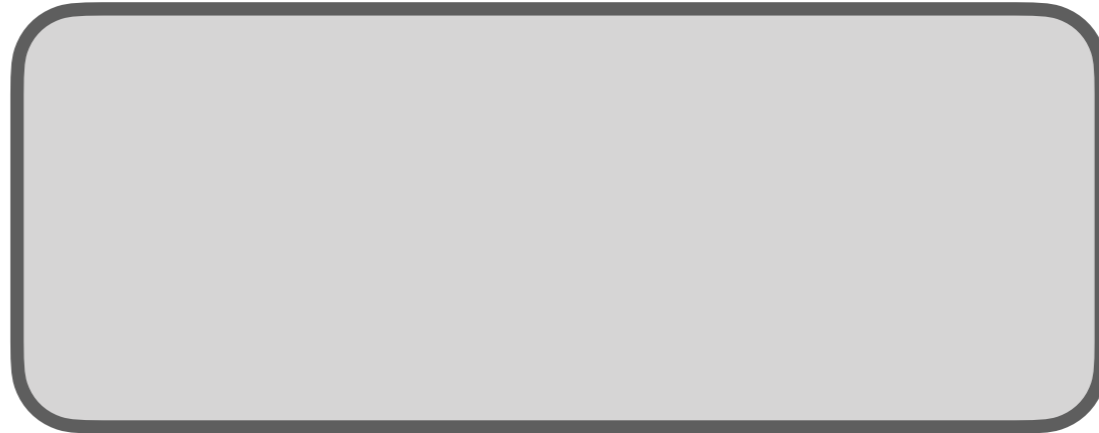
Declarative Languages [CQL]
KSQL, FlinkSQL, SparkSQL

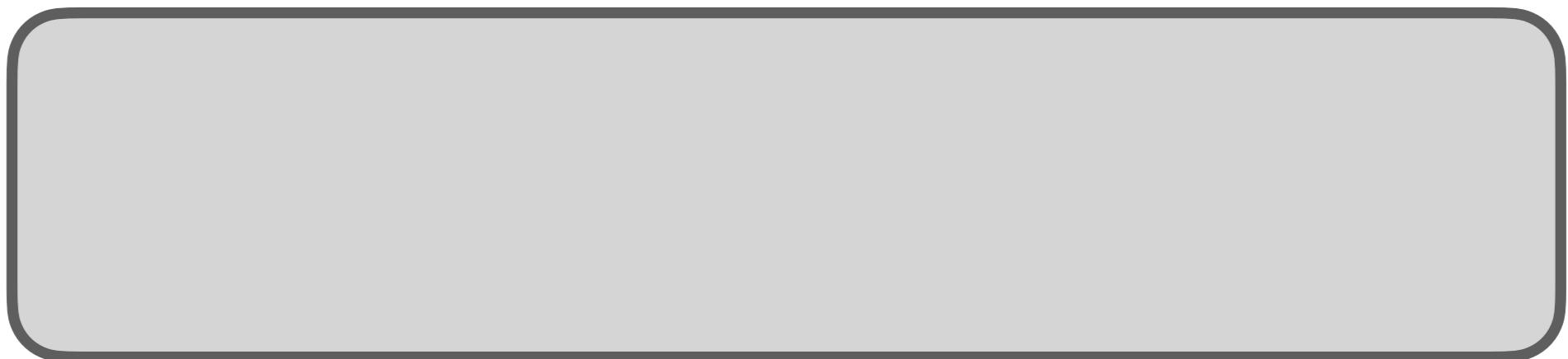
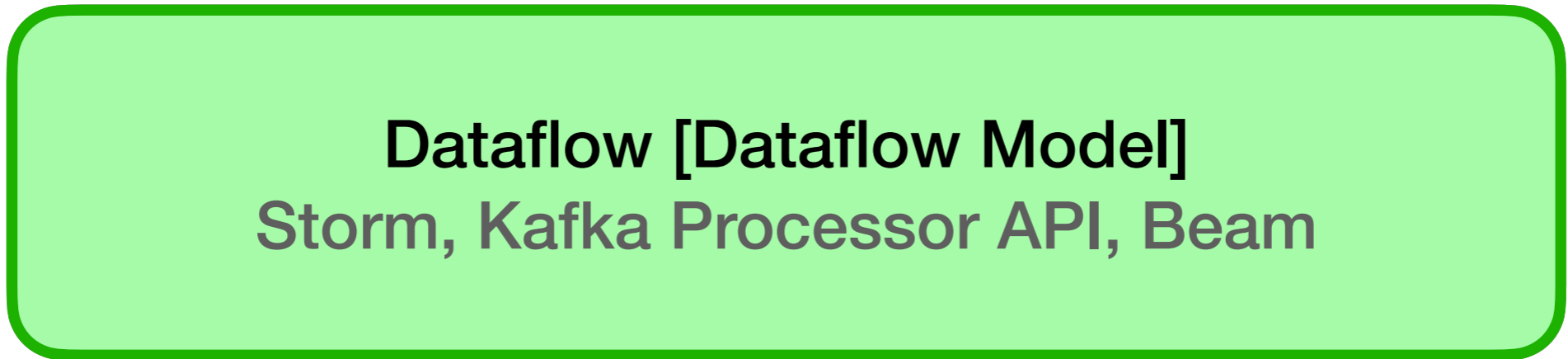
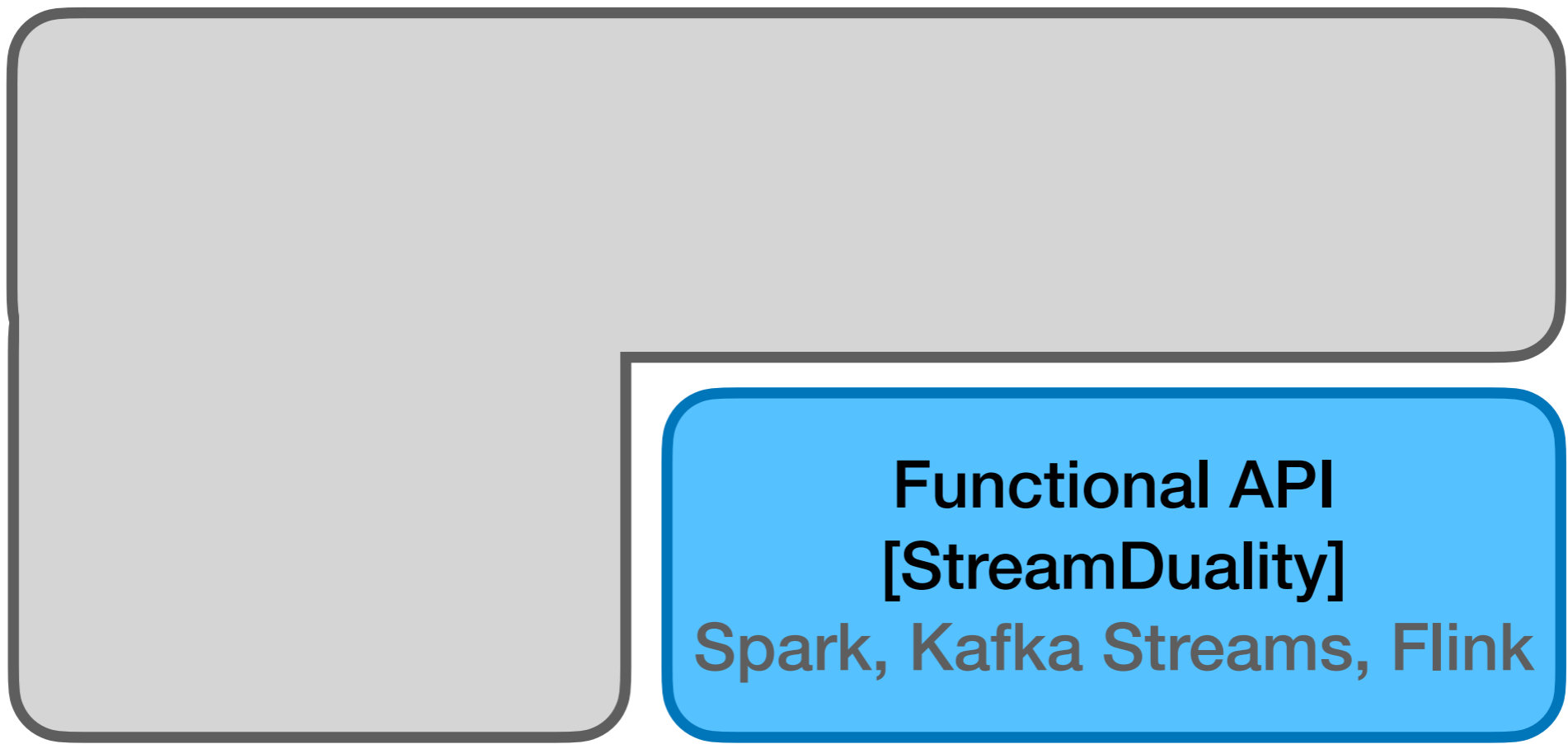
Functional API
DataFrames, KafkaStreams,
Flink Stream API

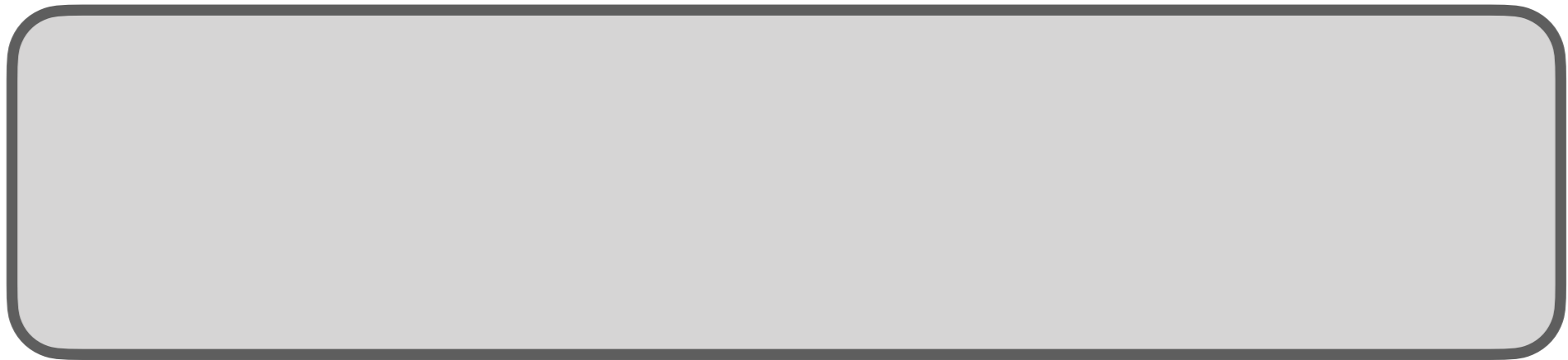
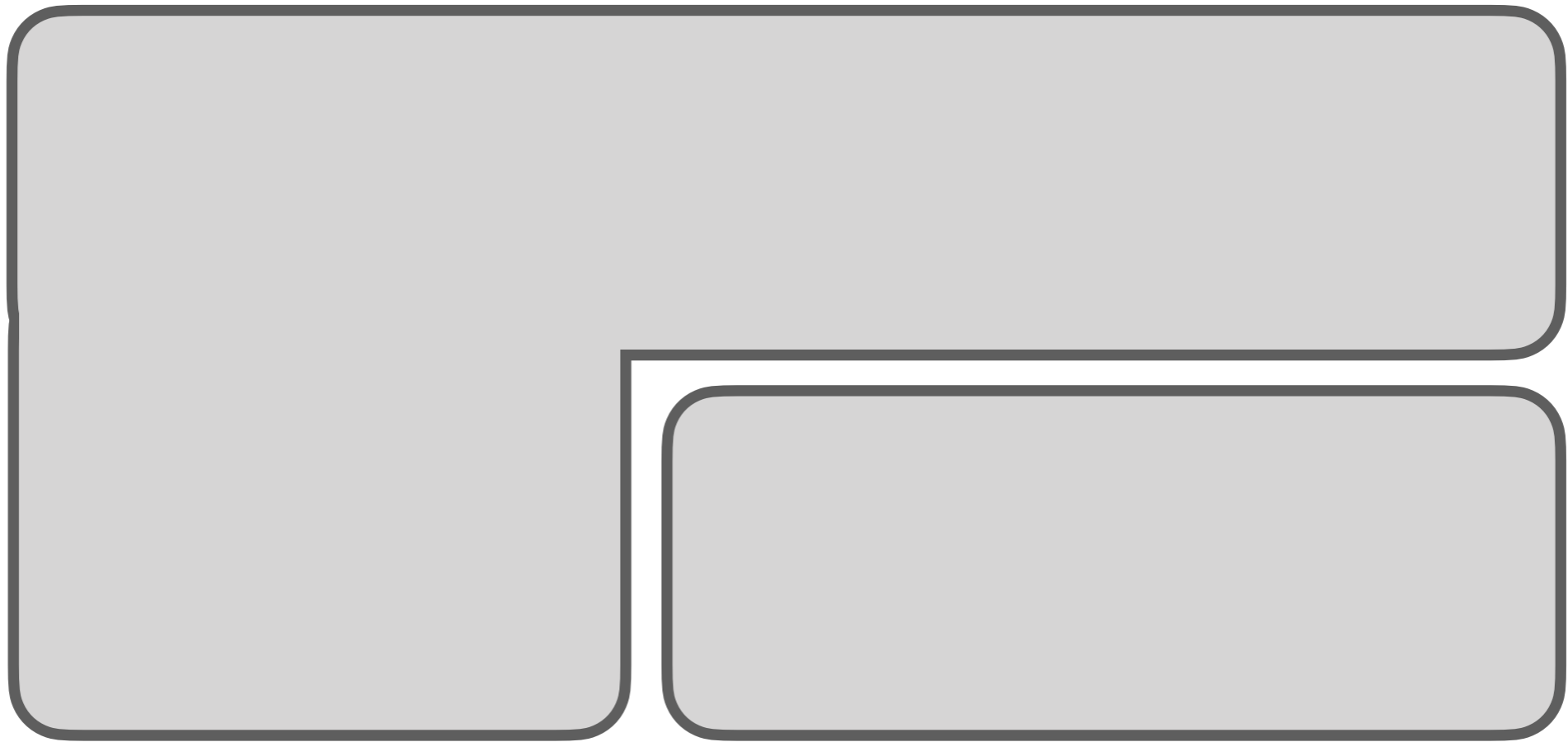
Dataflow Model
Kafka Processor API, Flink Process Function

Actor Model [Hewit et al.]

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



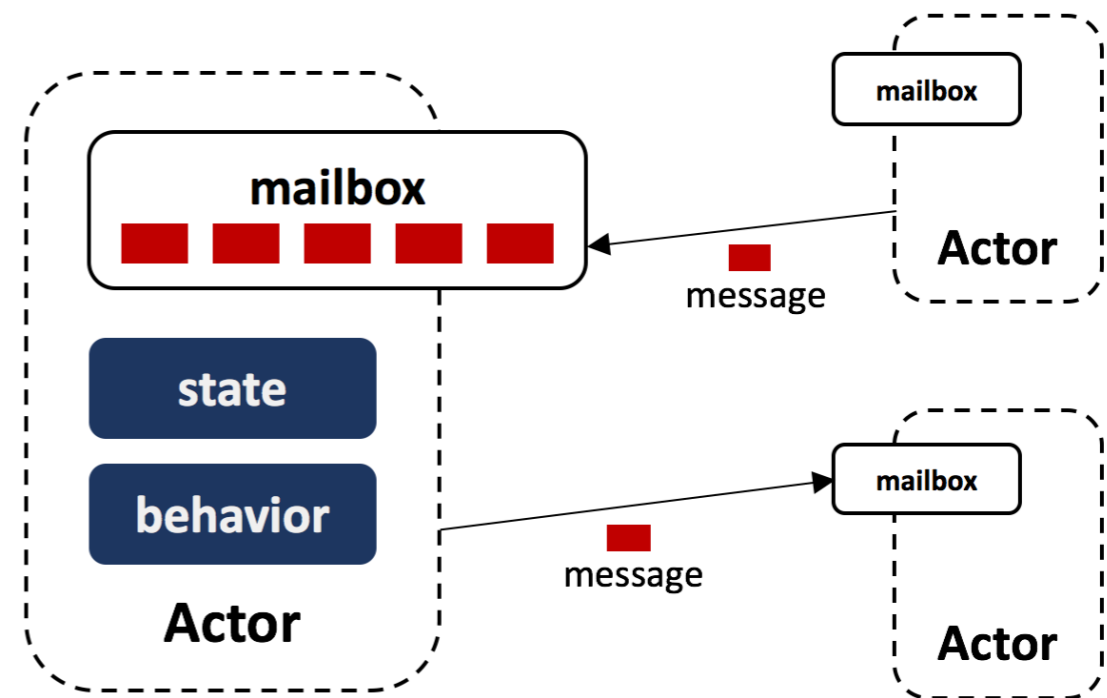




Actor Model [Hewit et al.]

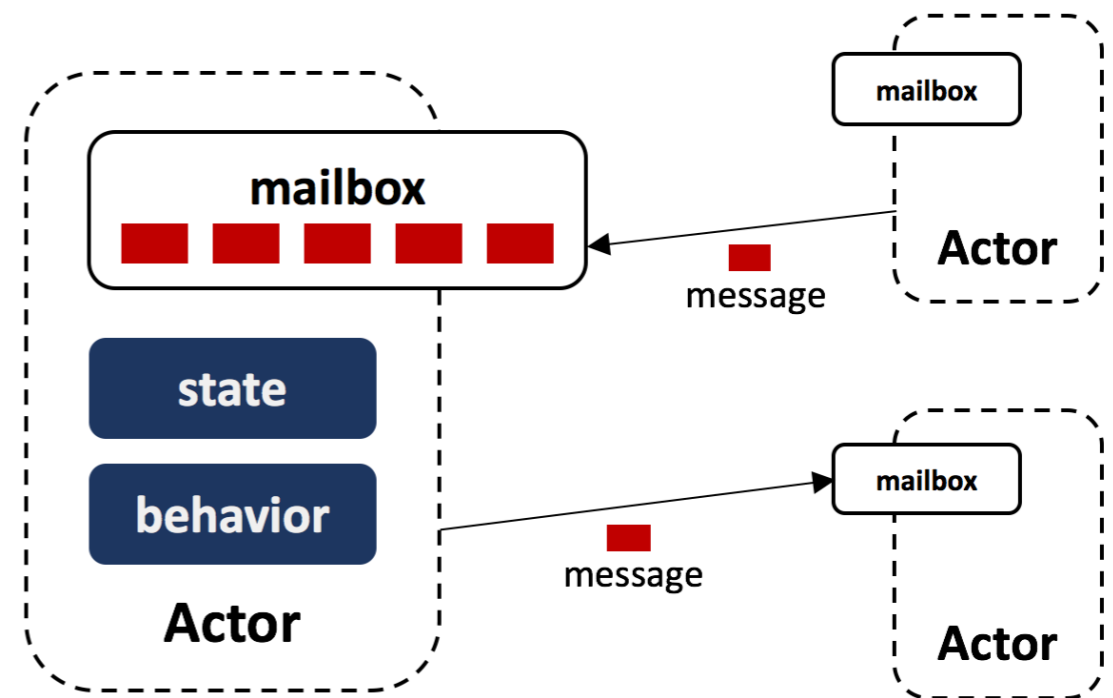
Actors

- 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



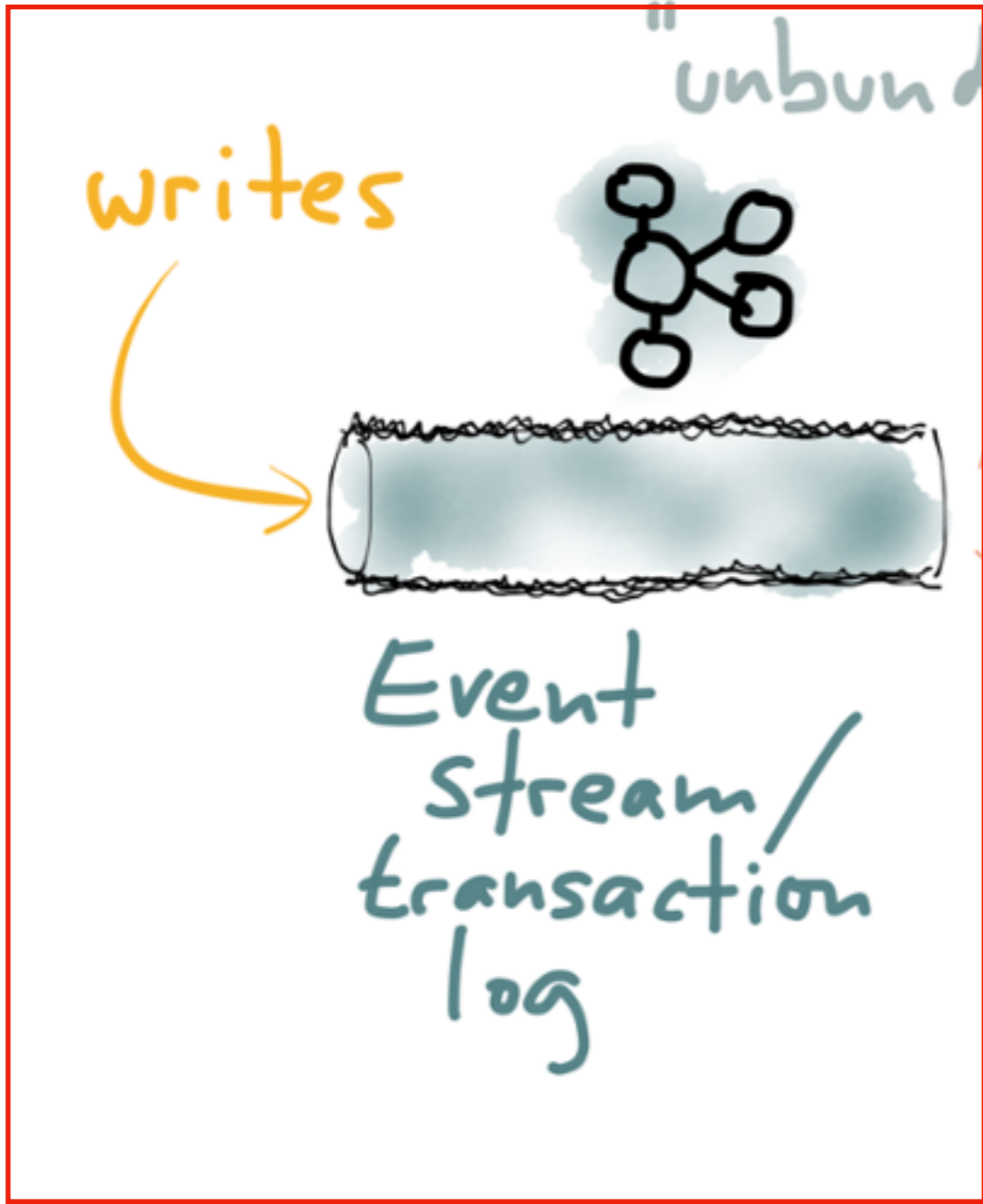
microservice 2



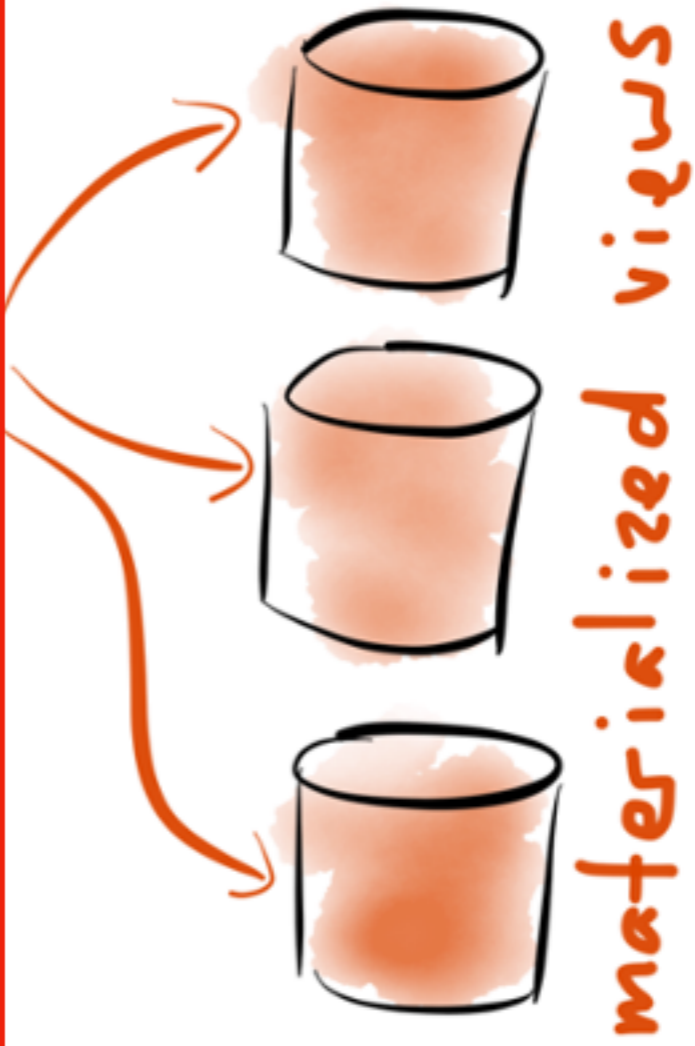
“
**L'ETAT
C'EST
MOI**
“

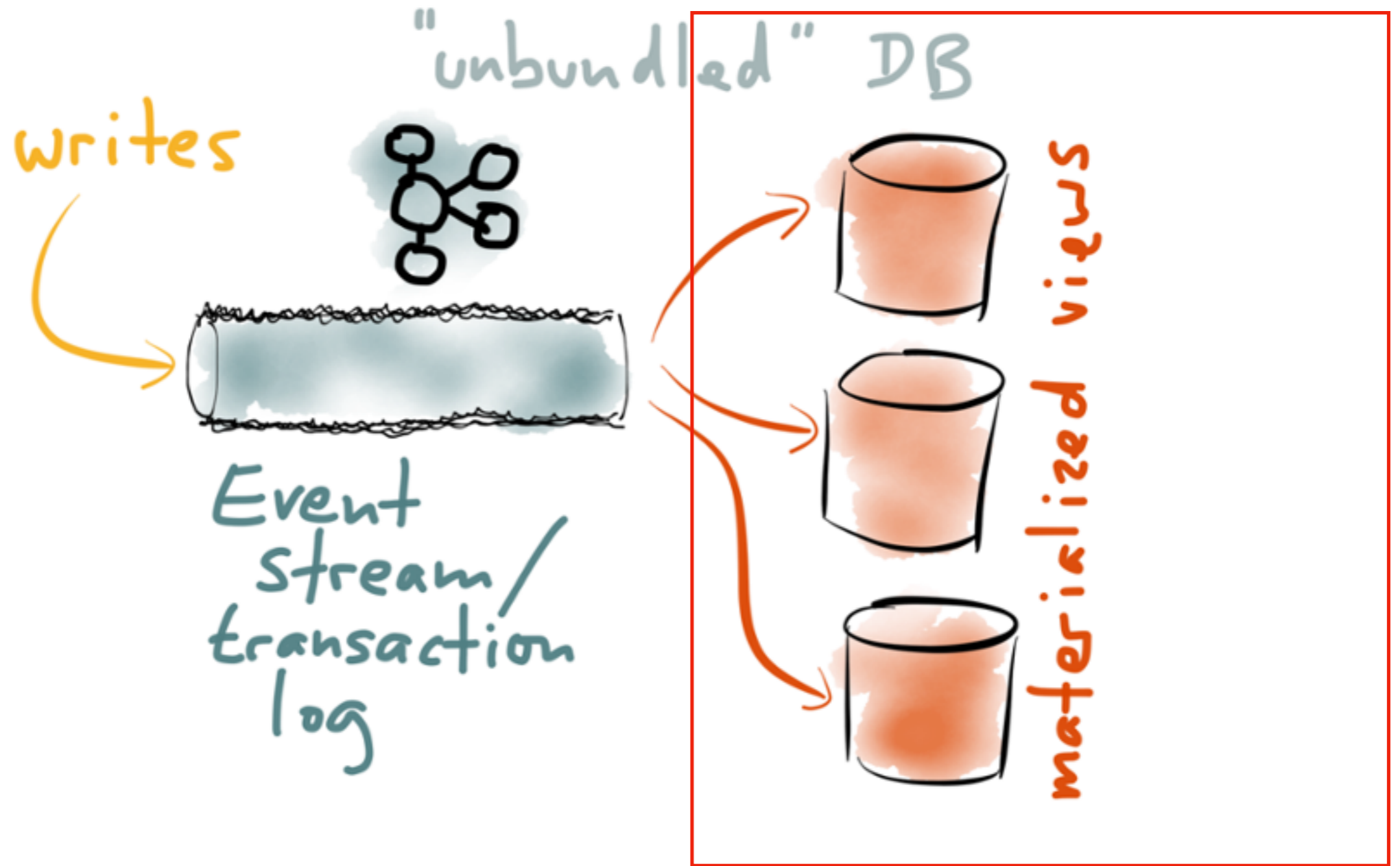
—
—
**Louis XIV
ApacheKafka**

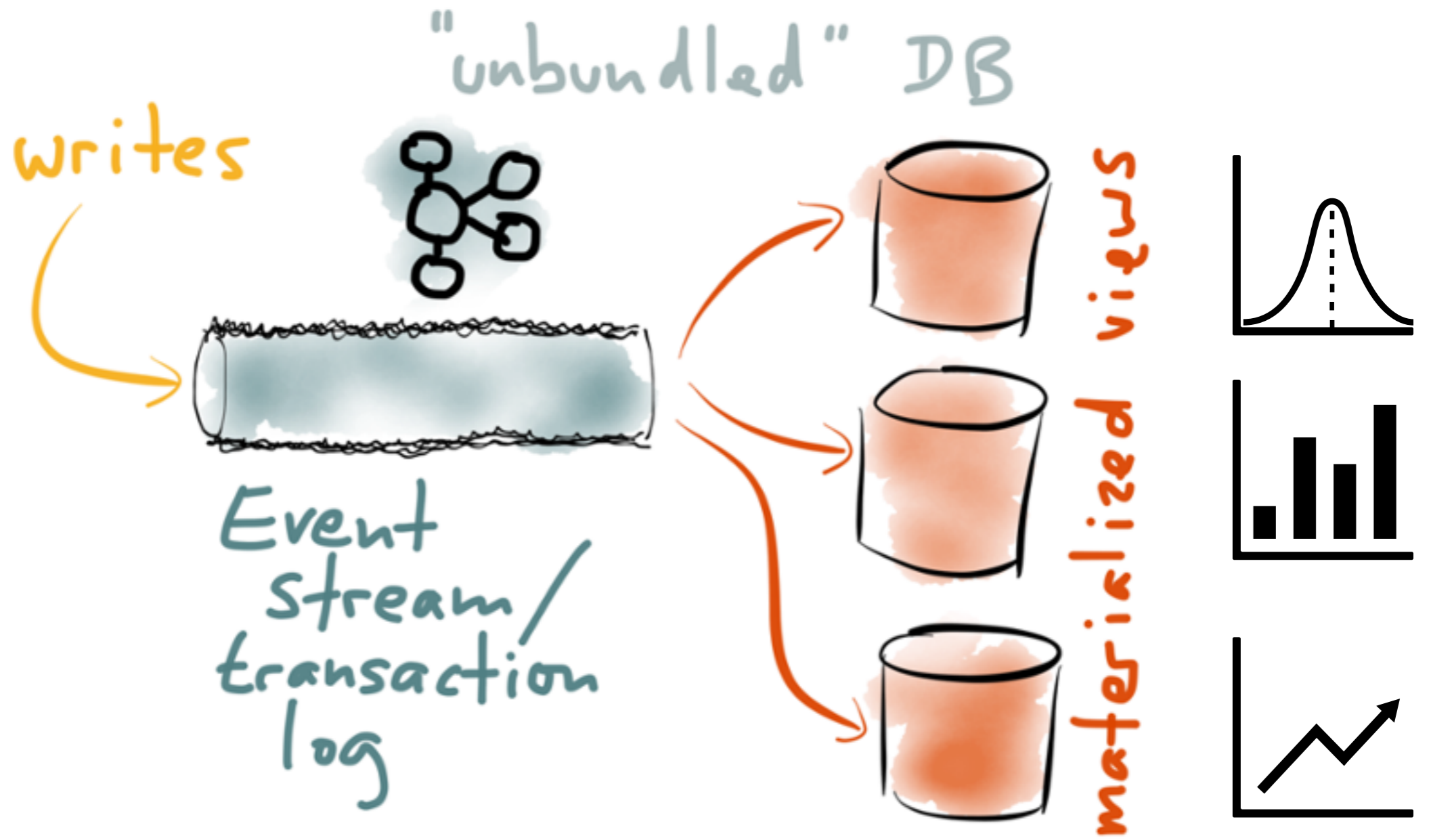




"unbundled" DB





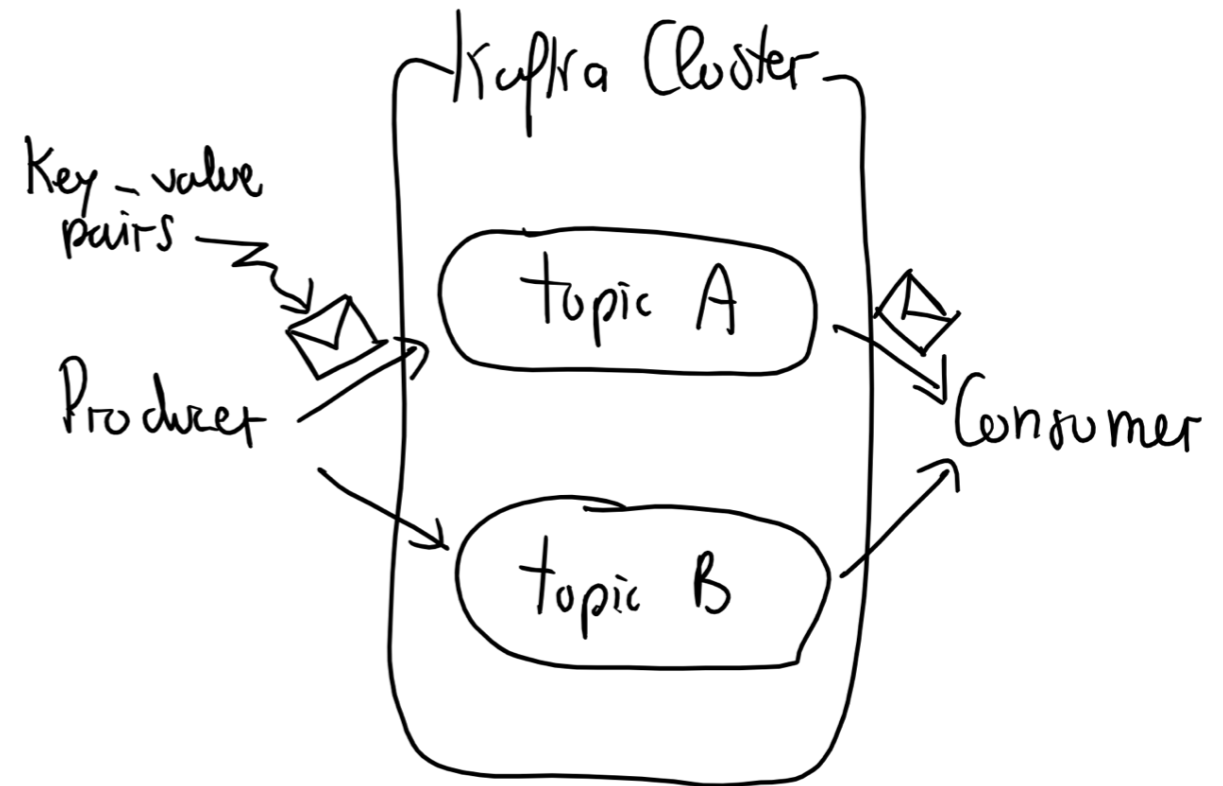


Should I Take a Step back?



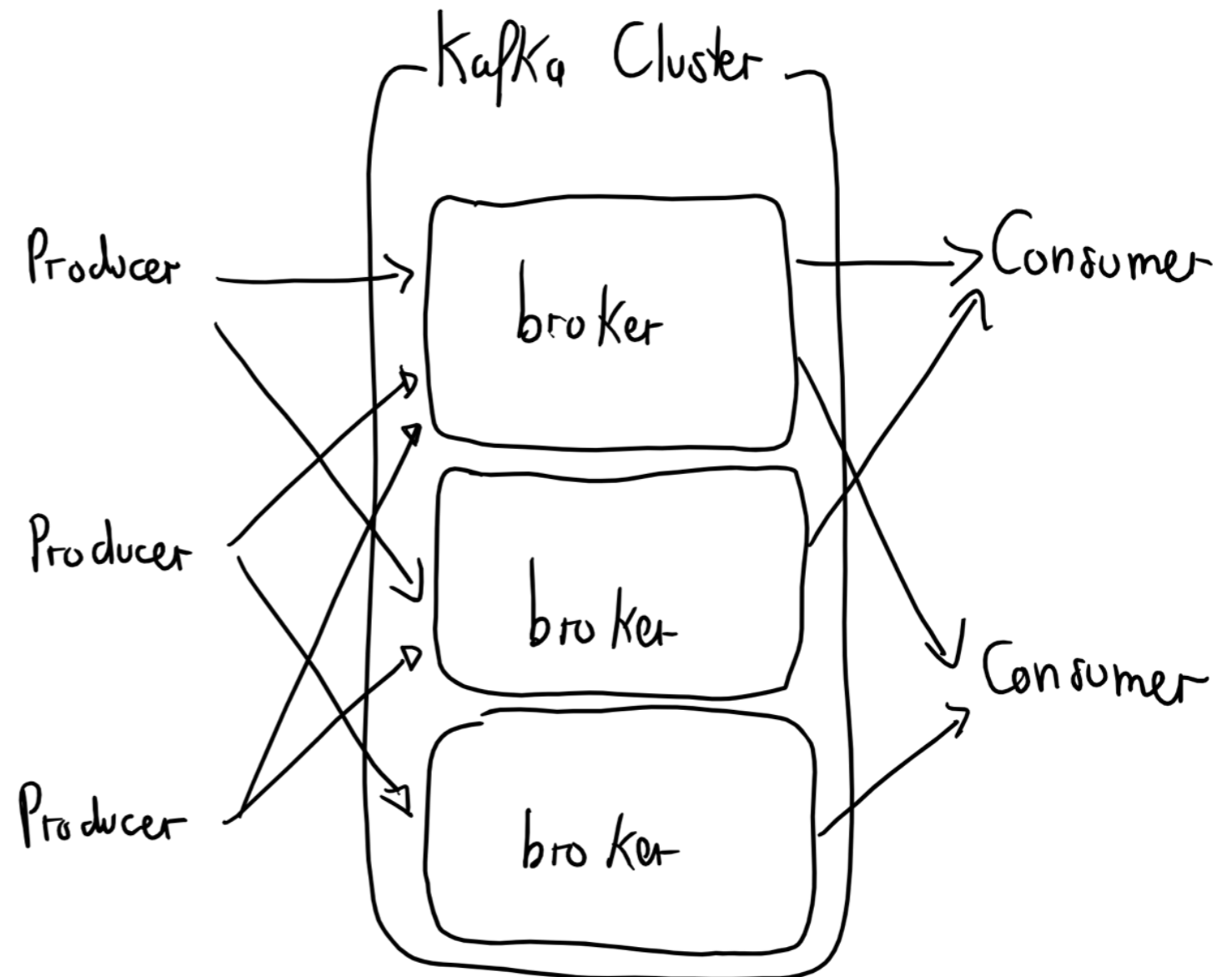
A Conceptual View of Kafka

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



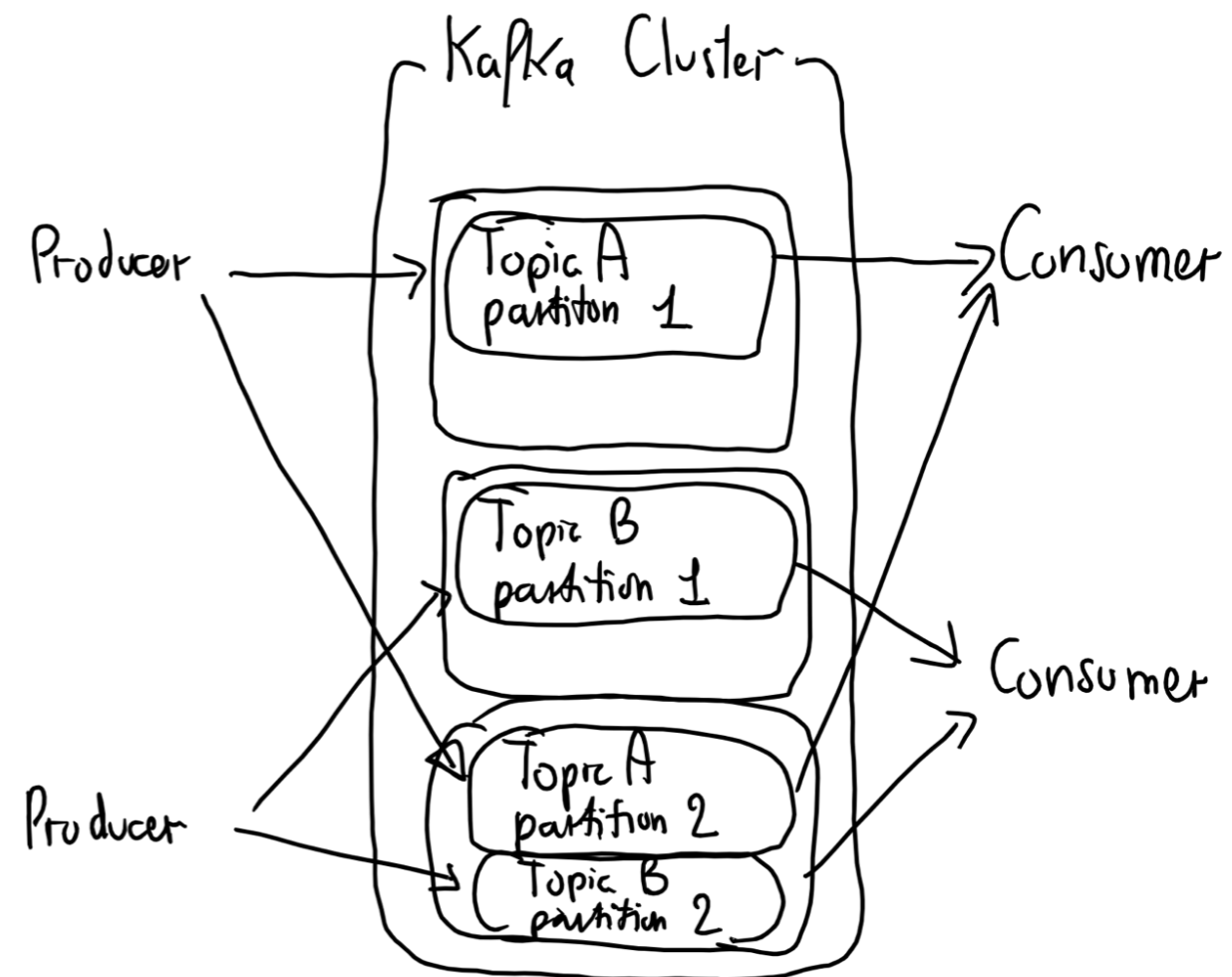
A Logical View of Kafka

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



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



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

