

Big Data Processing – Streaming Data (Velocity)

JOSEPH BONELLO

JOSEPH.BONELLO@UM.EDU.MT

Agenda

- Big Data - Velocity
- Introduction to Streams
 - Features of a Stream Processing System
 - Features of Data Stream Processing Systems
 - The Stream Model
- Tools in Handling Velocity
 - Storm
 - Spark

Aims

- By the end of this lecture, you should:
 - Understand the Velocity element of Big Data
 - Identify situations where Velocity is present
 - Understand the basics of stream management
 - Understand the complexity of handling stream data
 - Know what a Stream Processing System looks like
 - Appreciate the complex techniques employed in Stream Management Systems

Big Data – The Velocity Aspect

Velocity

The New York Stock Exchange captures

1 TB OF TRADE INFORMATION

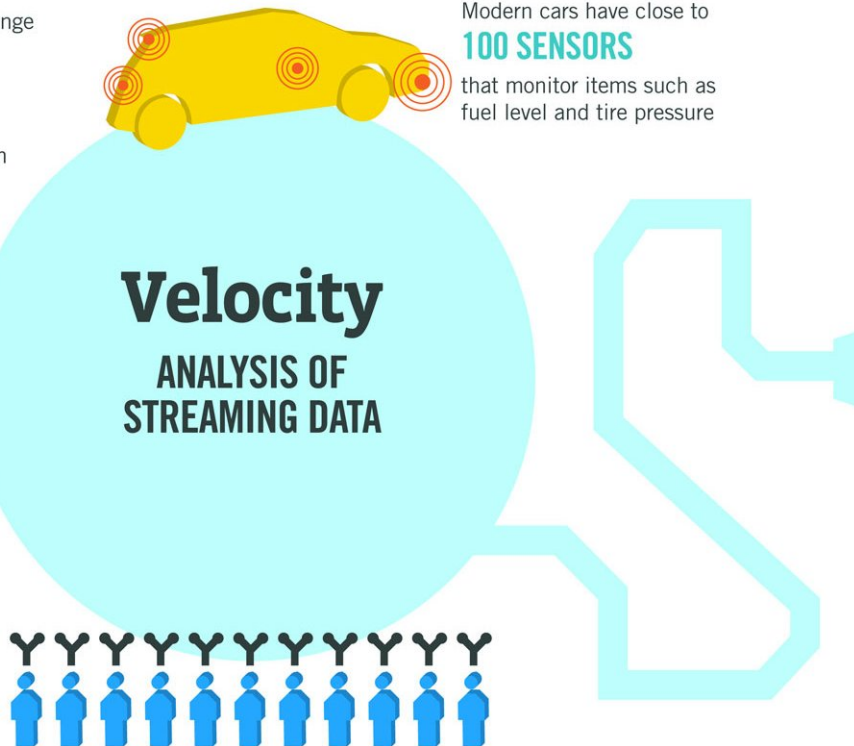
during each trading session



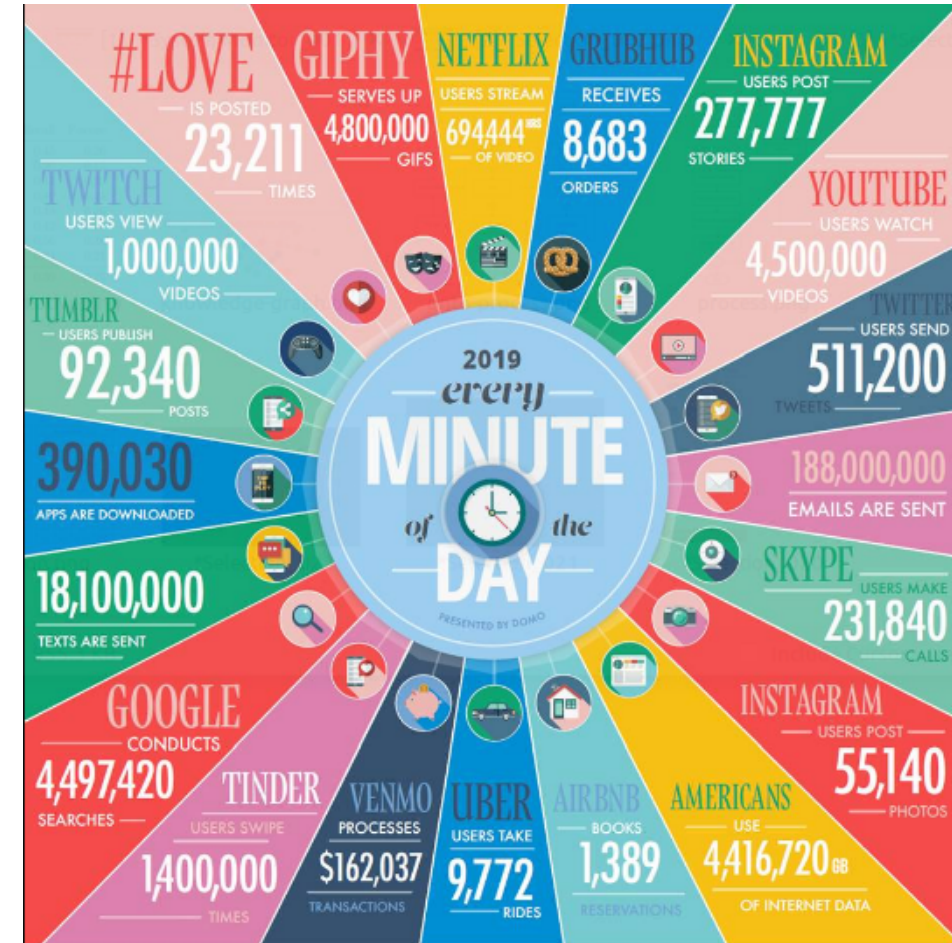
By 2016, it is projected there will be

18.9 BILLION NETWORK CONNECTIONS

— almost 2.5 connections per person on earth



Modern cars have close to **100 SENSORS** that monitor items such as fuel level and tire pressure



Velocity

Data is streaming in at unprecedented speed

Must be dealt with in a **timely** manner

- Ideally in near-real time

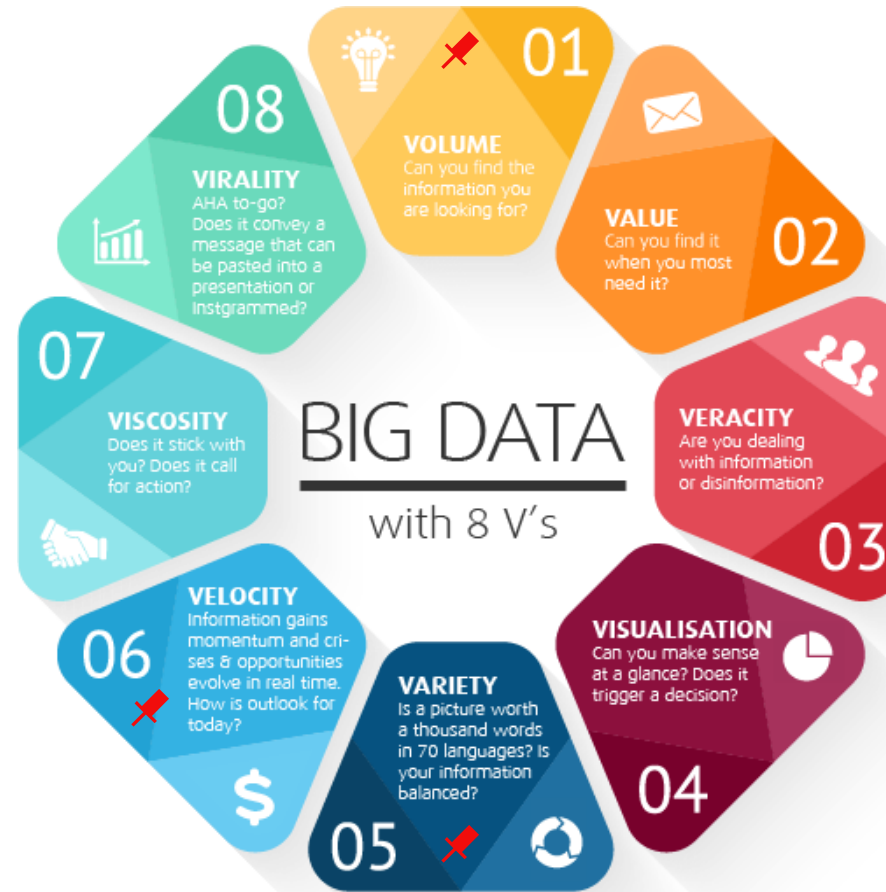
Reacting quickly enough to deal with data velocity is a **challenge** for most organizations

Velocity in a nutshell

Term refers to **how fast** data is being **produced** and how fast the data must be **processed** to meet demand

- How to deal with torrents of data **in near-real time?**

Big Data: The Vs



Where can we find Velocity?

Clickstreams and **ad impressions** capture user behaviour at millions of events per second

High Frequency stock **trading** algorithms reflect market changes within microseconds

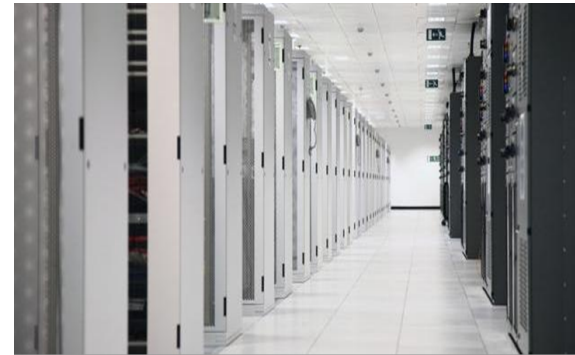
Machine-to-Machine processes exchange data between billions of devices

Infrastructure and **sensors** generate massive log data in realtime

Online gaming systems support millions of concurrent users, each producing multiple inputs per second

Where can we find Velocity?

Smart meter: records consumption of **electric energy** in **intervals** and communicates that information to the utility for **monitoring** and **billing** purposes



Every 15m



Smart Meter Case Study

Ontario's Meter Data Management and Repository (MDM/R):
storing, processing and managing all smart meter data in Ontario,
Canada

Characteristics:

- Provides **hourly billing** quantity and extensive reports
- **4.6 million** smart meters.
 - Storage/Bandwidth: **4.6M meters x 0.5K message** (typical HTTP) = **2.3 GB / round**
- **110 million** meter reads per day
 - on an **annual** basis, exceeds the number of debit card transactions processed in the Canada itself!

Source: Smart Metering Entity: <http://www.smi-ieso.ca/mdmr>

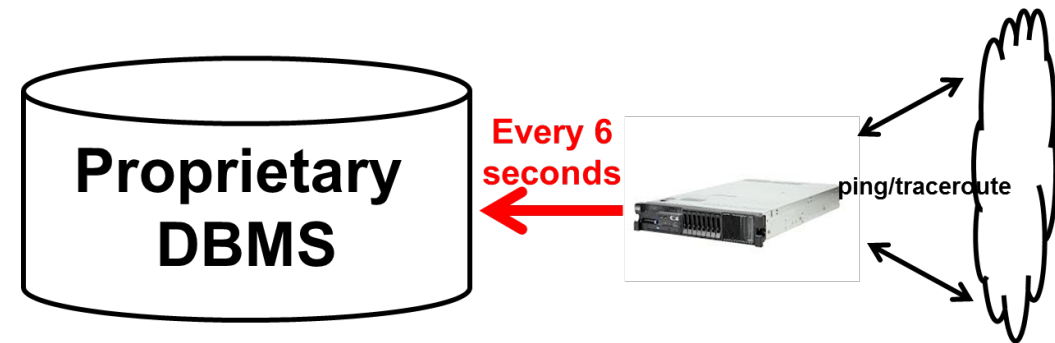
Where can we find Velocity?

Akamai:

- **CDN** serving 15-30% of all Web traffic (10TB/sec)
 - One out of every three Global 500® companies
 - All of the top Internet portals
- Has a picture of the global traffic every **6 seconds**

How?

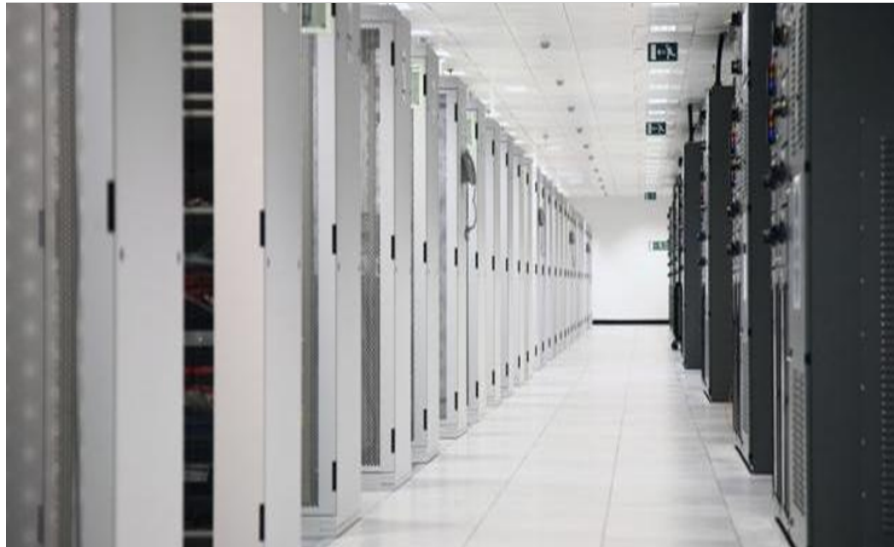
- **119,000** servers in **80** countries within over 1,100 networks.
- Servers report to a **proprietary database** network health information (latency/loss) every **6 seconds**.



Where can we find Velocity?

Analyse **online conversations** in Social Nets.

Accelerated responses to marketplace shifts



Continuously



Over
Web2.0
protocols



Introduction to Data Streams

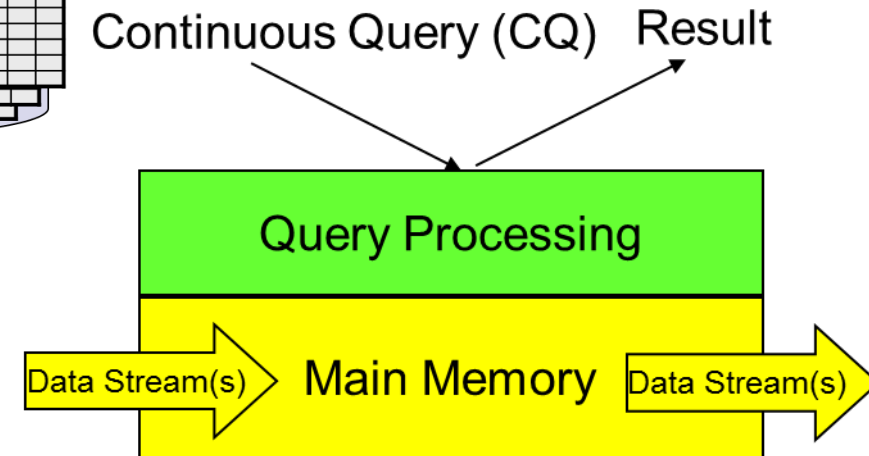
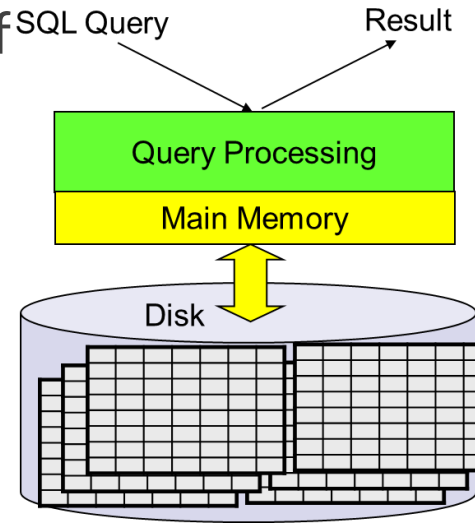
Data Management Vs Stream Management

In a DBMS, **input** is **under the control** of SQL Query
the programming staff

- SQL INSERT commands
- SQL bulk loaders

Stream management is important when
the **input rate** is **controlled externally**

- Example: Search Engine queries



Features of DBMS and DSMS

Traditional DBMS:

- **stored** sets of relatively **static records** with **no pre-defined notion of time**
- good for applications that require **persistent data storage** and **complex querying**

DSMS:

- support **on-line analysis** of **rapidly changing** data streams
- *data stream*: **real-time, continuous, ordered** (implicitly by arrival time or explicitly by timestamp) sequence of items, **too large** to store entirely, not ending
- **continuous queries**

Features of DBMS and DSMS

DBMS

Persistent relations (relatively static, stored)
One-time queries
Random access
“**Unbounded**” disk store
Only **current state** matters
No real-time services
Relatively **low update** rate
Data at **any granularity**
Assume **precise** data
Access plan **determined by query**
processor, physical **DB design**

DSMS

Transient streams (on-line analysis)
Continuous queries (CQs)
Sequential access
Bounded main memory
Historical data is important
Real-time requirements
Possibly **multi-GB arrival** rate
Data at **fine granularity**
Data **stale/imprecise**
Unpredictable/variable data arrival and characteristics

Applications

Mining **query streams**

- Google wants to know **what queries** are **more frequent today** than **yesterday**

Mining **click streams**

- Yahoo! wants to know **which** of its pages are getting an **unusual** number of hits in the **past hour**
 - Often caused by annoyed users clicking on a **broken page**

IP packets can be monitored at a **switch**

- Gather information for **optimal routing**
- **Detect** denial-of-service (**DOS**) attacks

DSMS Applications

Sensor Networks

- E.g. TinyDB

Network Traffic Analysis

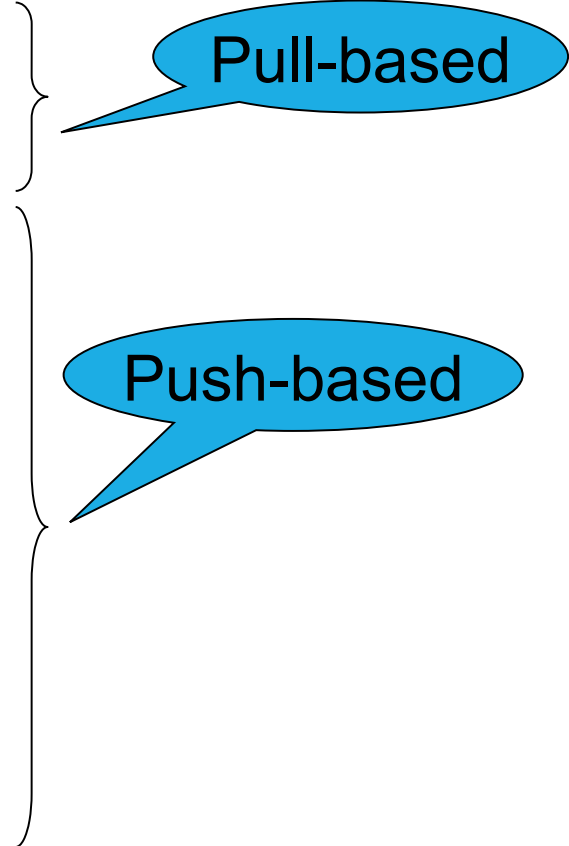
- Real time analysis of Internet traffic. E.g., Traffic statistics and critical condition detection

Financial Tickers

- On-line analysis of stock prices, discover correlations, identify trends

Transaction Log Analysis

- E.g. Web click streams and telephone calls



Pull-based

Push-based

Data Streams - Terms

A **data stream** is a (potentially unbounded) sequence of tuples

- Each tuple consist of a set of attributes, similar to a row in database table

Transactional data streams: log interactions between entities

- Credit card: purchases by consumers from merchants
- Telecommunications: phone calls by callers to dialed parties
- Web: accesses by clients of resources at servers

Measurement data streams: monitor evolution of entity states

- Sensor networks: physical phenomena, road traffic
- IP network: traffic at router interfaces
- Earth climate: temperature, moisture at weather stations

Why do we need Steam Processing

Massive data sets:

- **Huge numbers of users**, e.g. (from 2008):
 - AT&T long-distance: ~ 300M calls/day
 - AT&T IP backbone: ~ 10B IP flows/day
- **Highly detailed measurements**, e.g.,
 - NOAA: satellite-based measurements of earth geodetics (the study of measurement and representation of the earth)
- **Huge number of measurement points**, e.g.,
 - Sensor networks with huge number of sensors

Why do we need Steam Processing

Near real-time analysis

- ISP: controlling service levels
- NOAA: tornado detection using weather radar
- Hospital: Patient monitoring

Traditional data feeds

- **Simple queries** (e.g., value lookup) needed in real-time
- **Complex queries** (e.g., trend analyses) performed off-line

Requirements

Data model and query semantics: order- and time-based operations

- Selection
- Nested aggregation
- Frequent item queries
- Joins
- Windowed queries

Requirements

Query processing:

- Streaming query plans must use **non-blocking** operators
- Only **single-pass algorithms** over data streams

Data reduction: approximate summary structures

- Synopses, digests => no exact answers

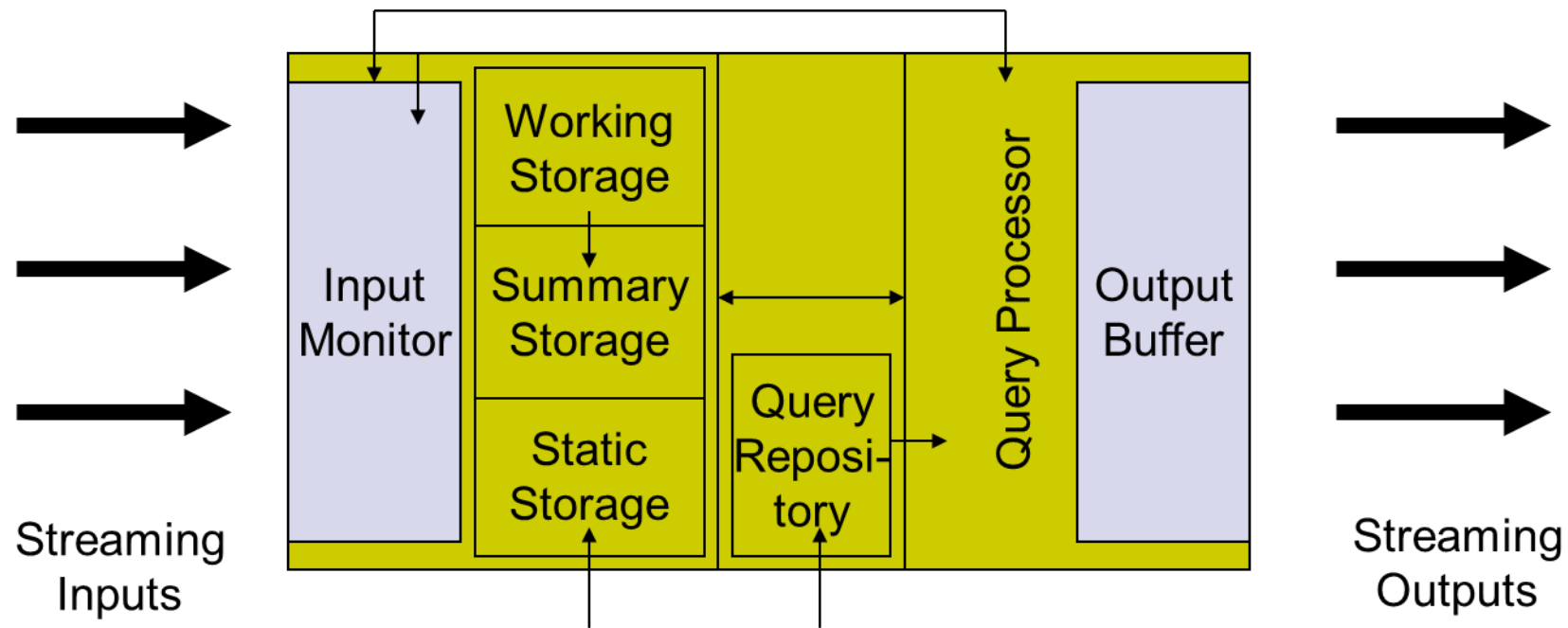
Requirements

Real-time reactions for monitoring applications => active mechanisms

Long-running queries: variable system conditions

Scalability: shared execution of many continuous queries, monitoring multiple streams

Generic Architecture



The Stream Model

Input **tuples** enter at a **rapid rate**, at **one or more** input **ports**

The **system** cannot store the **entire stream** accessibly

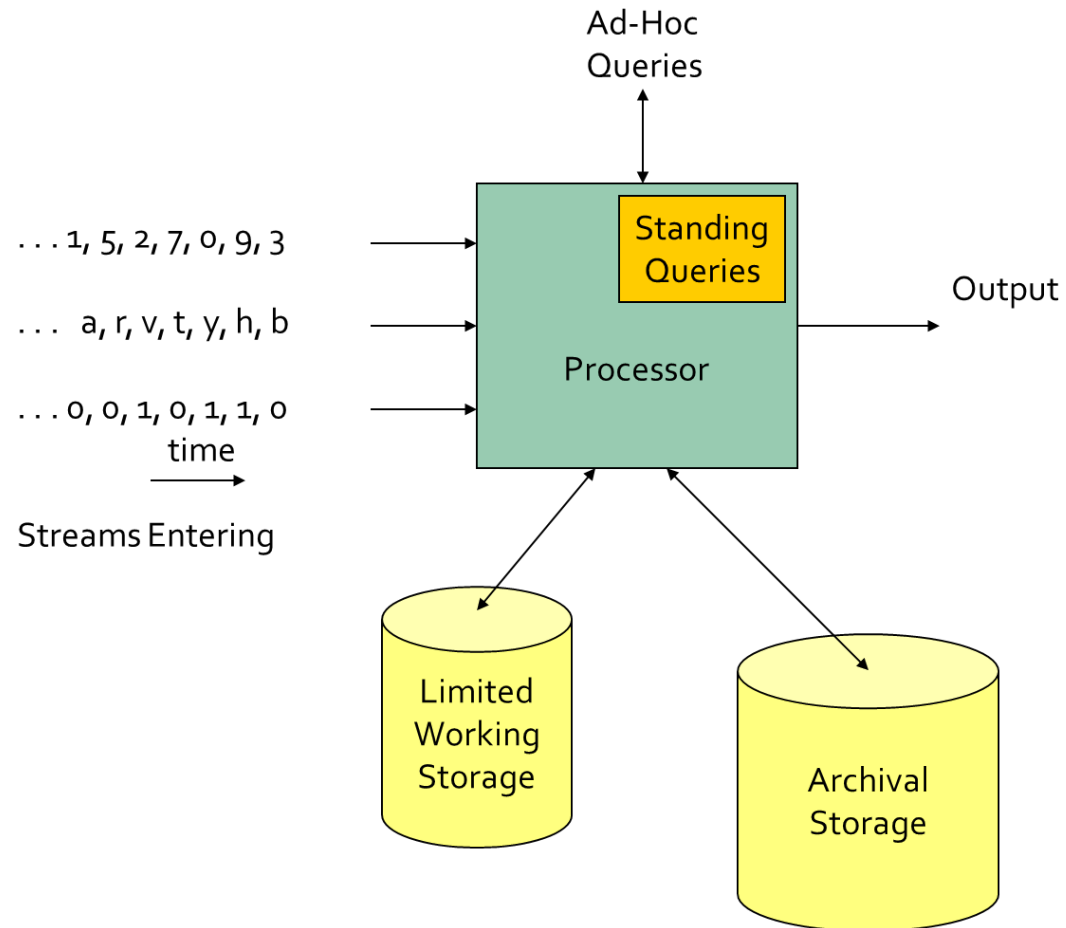
How do you make critical calculations about the stream using a limited amount of (primary or secondary) memory?

The Stream Model

Tuples

- **Finite ordered** list of elements
- An *n-tuple* is a **sequence** of *n* elements, where *n* is a **non-negative** integer ($n \in \mathbb{N}$)
- A **0-tuple** is the **empty sequence**
- Tuples are usually written by **listing the elements within parenthesis**
 - Example: (2,4,6,8,10)
- Unlike a set, tuples can contain **multiple instances** of the **same element**

Stream Management Outline



Sliding Windows

A useful model of stream processing is that **queries are about a window of length N** – the **N most recent** elements received

- **Alternative:** elements received **within a time interval T**

Interesting case: N is so large it cannot be stored in main memory

- **Or,** there are so many streams that windows for all do not fit in main memory

Sliding Windows

q w e r t y u i o p a **s d f g h j** k l z x c v b n m

q w e r t y u i o p a s **d f g h j k** l z x c v b n m

q w e r t y u i o p a s d **f g h j k l** z x c v b n m

q w e r t y u i o p a s d f **g h j k l z** x c v b n m

← Past Future →

Existing Tools

Storm?



“Distributed and fault-tolerant real-time computation”

<http://storm.apache.org/>

Originated at BackType/Twitter, open sourced in late 2011

Implemented in Clojure, some Java

Current version of Storm is 2.3.0, released September 2021

Where has Storm been used?

Twitter: personalization, search, revenue optimization, ...

- 200 nodes, 30 topologies, 50B msg/day, avg latency <50ms, Jun 2013

Yahoo: user events, content feeds, and application logs

- 320 nodes (YARN), 130k msg/s, June 2013

Spotify: recommendation, ads, monitoring, ...

- v0.8.0, 22 nodes, 15+ topologies, 200k msg/s, Mar 2014

Alibaba, Cisco, Flickr, PARC, WeatherChannel, ...

- Netflix is looking at Storm and Samza, too.

Data in Storm

(1.1.1.1, "foo.com")
(2.2.2.2, "bar.net")
(3.3.3.3, "foo.com")
(4.4.4.4, "foo.com")
(5.5.5.5, "bar.net")

DNS queries



(("foo.com", 3)
("bar.net", 2))

Top queried
domains

Functional Programming

((1.1.1.1, "foo.com")
 (2.2.2.2, "bar.net")
 (3.3.3.3, "foo.com")
 (4.4.4.4, "foo.com")
 (5.5.5.5, "bar.net"))

DNS queries



("foo.com", "bar.net", "foo.com",
 "foo.com", "bar.net")

f



{"bar.net" -> 2, "foo.com" -> 3}

g



(("foo.com", 3)
 ("bar.net", 2))

h

Functional Programming

$h(g(f(\text{data})))$

→ λ -calculus

Storm Core Concepts

A First Look

Storm is **distributed** Functional
Programming -like
processing of data streams.

Same idea, many machines.
(but there's more of course)

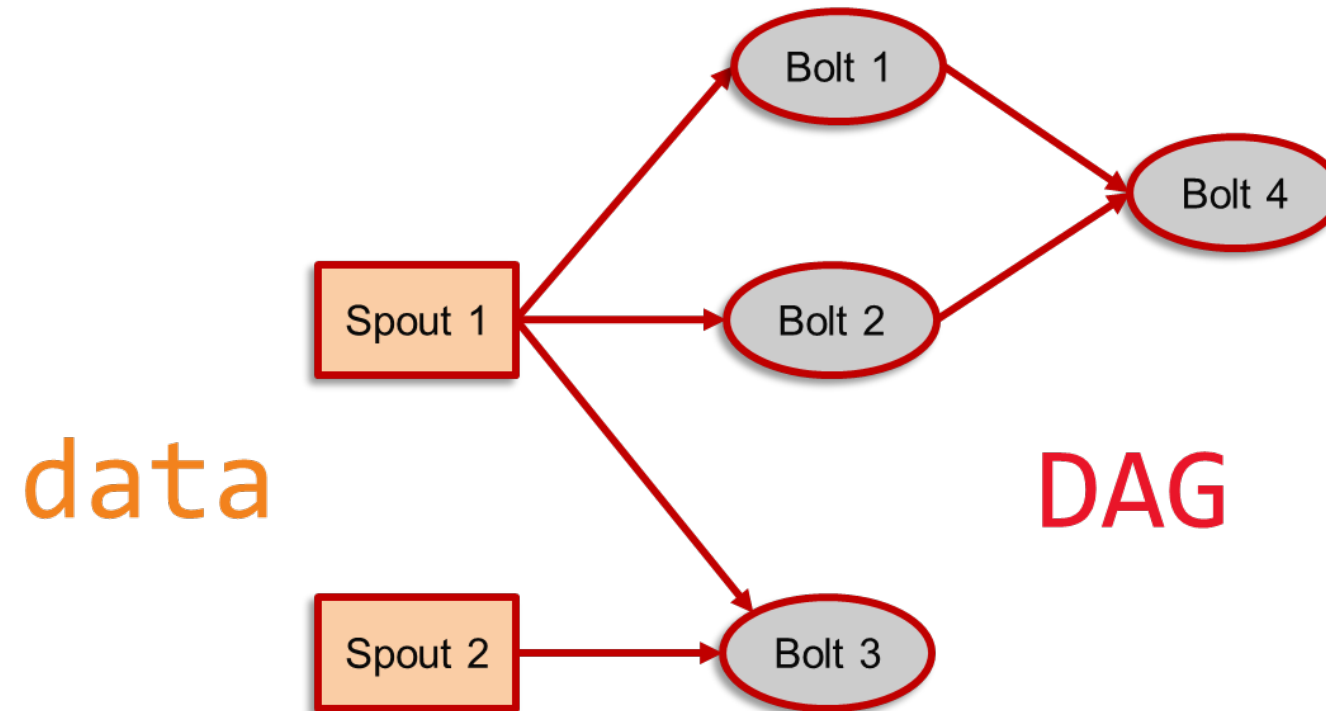
Storm Topology

A topology in Storm wires
data and *functions* via a
Directed Acyclic Graph

Executes on many machines
like a Map/Reduce job in
Hadoop

Storm Topology

functions



Apache Spark

Apache Spark is “a fast and general engine for large-scale data processing”

Available from <http://spark.apache.org/>

Current versions are Spark 2.4.8, released in May 2021, Spark 3.0.3, released Jun, 2021, Spark 3.1.2, released Jun, 2021 and 3.2.0 released Oct, 2021



But what is Spark?

Fast and Expressive **Cluster Computing Engine** Compatible with Apache Hadoop

Efficient

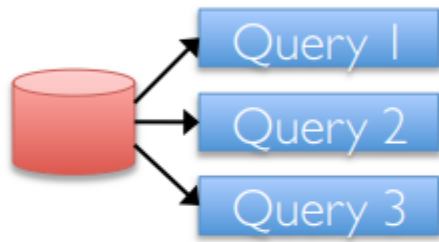
- General execution graphs
- In-memory storage
- Claims to be up to 10 times faster on disk, and up to 100 times faster in memory

Usable

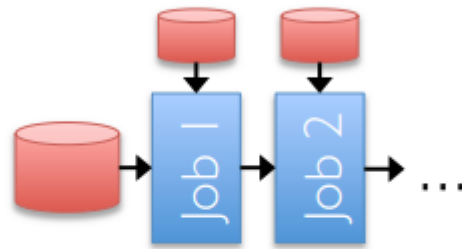
- Rich APIs in Java, Scala, Python, R and SQL
- Interactive shell
- Claims to require 2 to 5 times less code

Motivation for Spark

- Using Map Reduce for complex jobs, interactive queries and online processing involves *lots of disk I/O*



Interactive mining



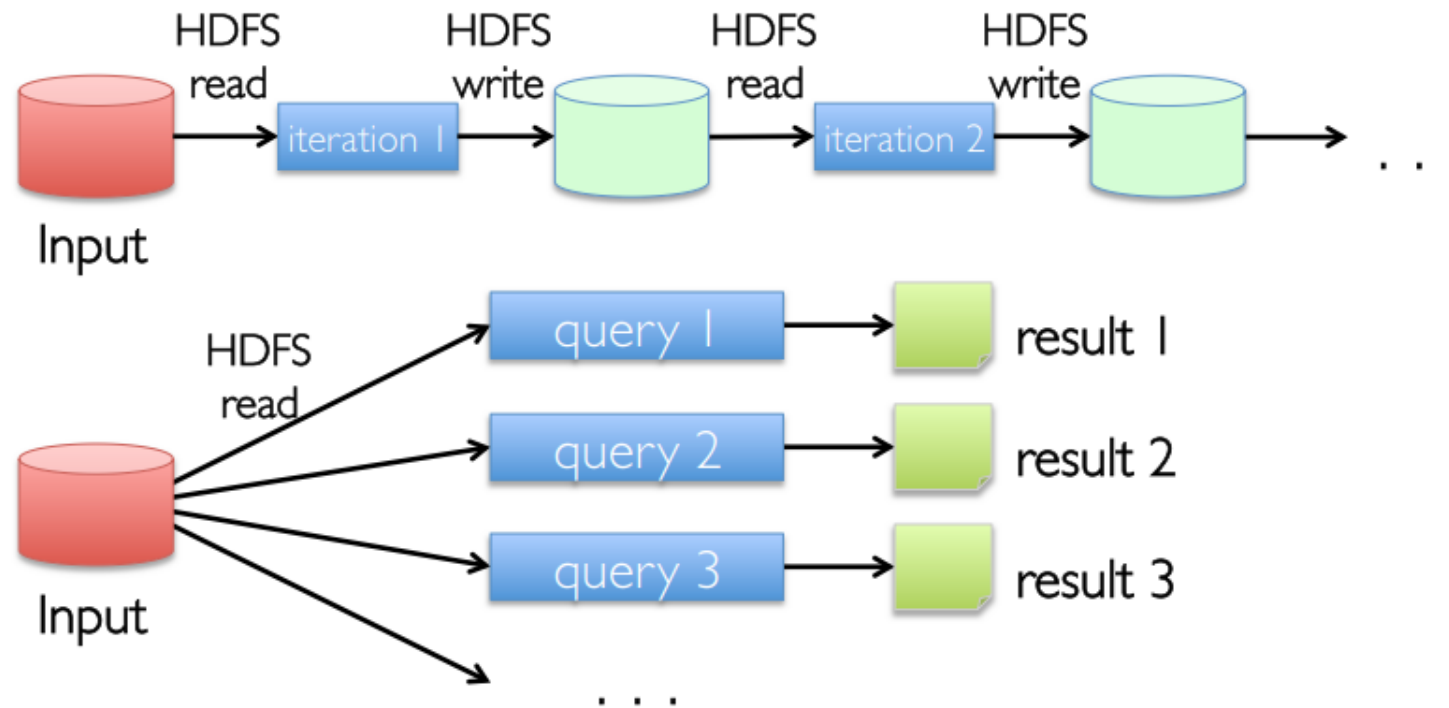
Stream processing

Also, iterative jobs

Disk I/O is very slow

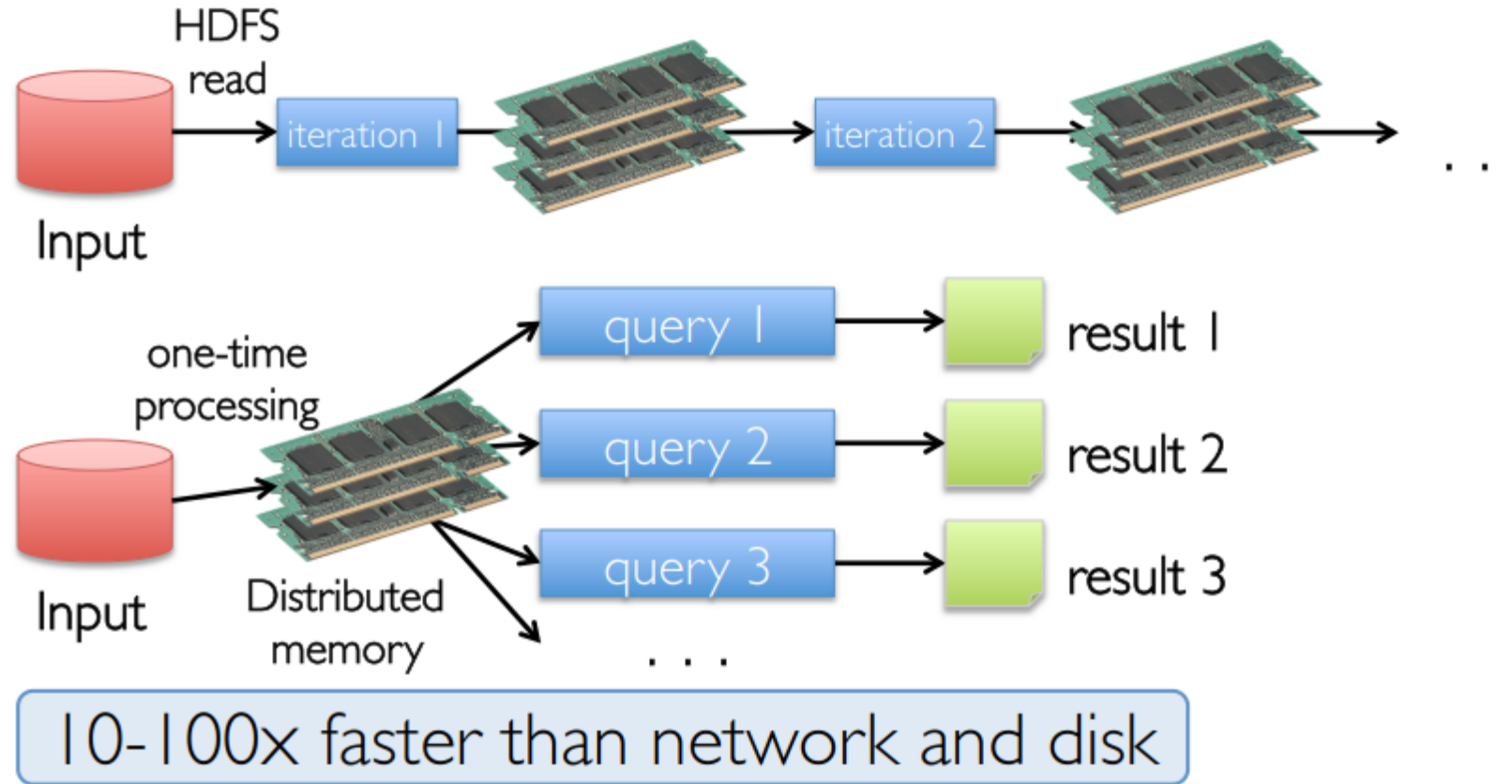
How to solve this problem?

Use Memory Instead of Disk

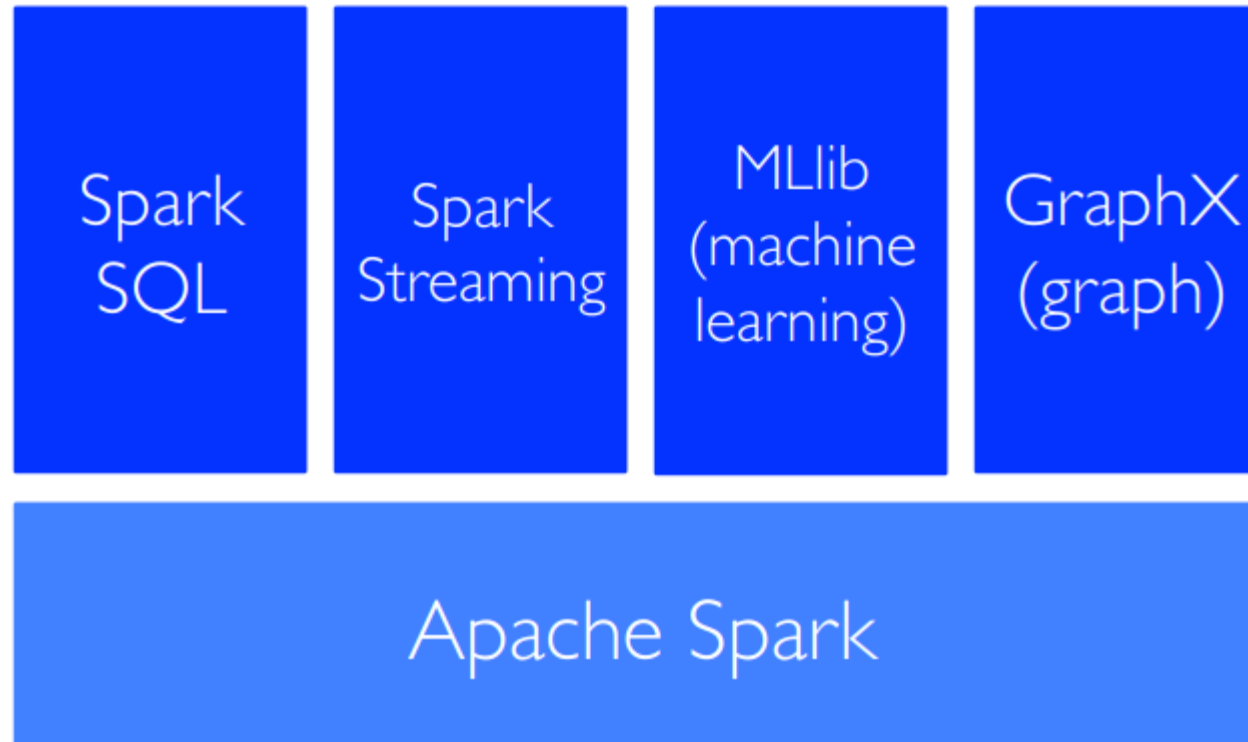


How to solve this problem?

In-Memory Data Sharing



The Spark Stack



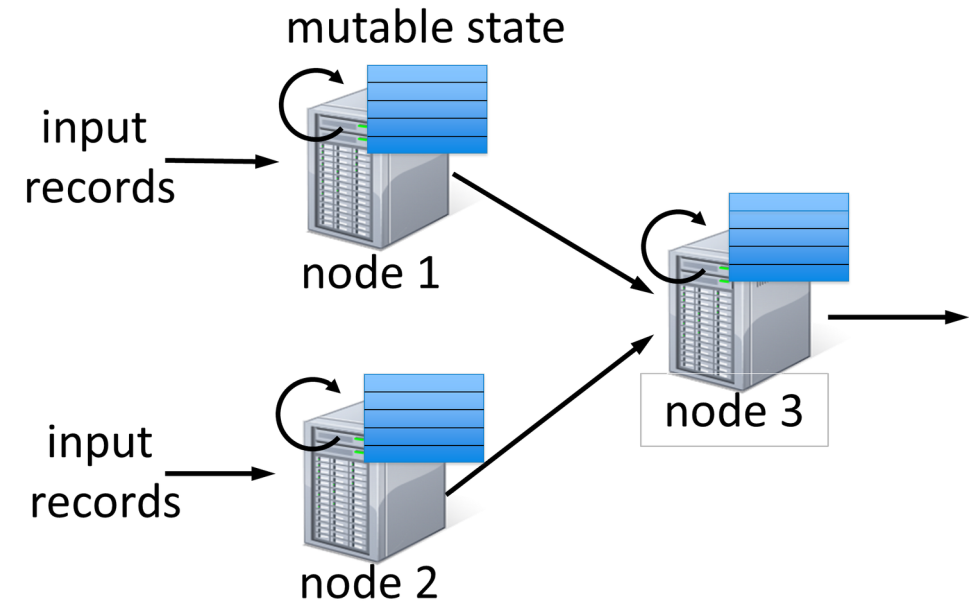
Stateful Stream Processing

Traditional streaming systems have a **event-driven record-at-a-time** processing model

- Each node has **mutable** state
- For each record, update state & send new records

State is **lost** if node **dies**!

Making **stateful** stream processing be fault-tolerant is **challenging**



Spark compared to other Streaming Systems

Storm

- Replays record if not processed by a node
- Processes each **record at least once**
- **May update mutable** state twice!
- Mutable state **can be lost** due to failure!

Trident – Use transactions to update state

- Processes each record **exactly once**
- Per state transaction **updates slow**

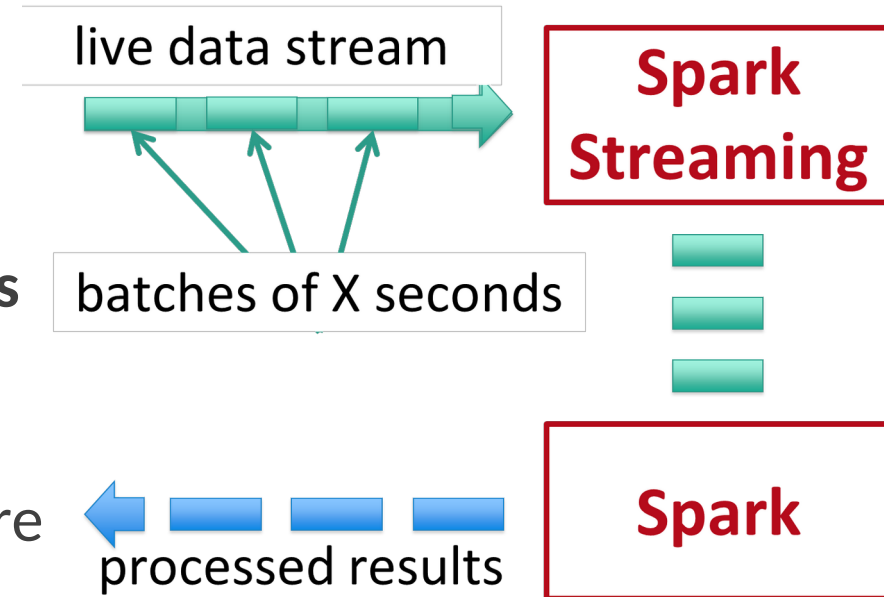
Discretised Stream Processing

Run a streaming computation as a **series of very small, deterministic batch jobs**

Chop up the live stream into **batches** of X seconds

Spark treats each **batch** of data as RDDs and **processes** them using **RDD** operations

Finally, the **processed results** of the RDD operations are **returned in batches**

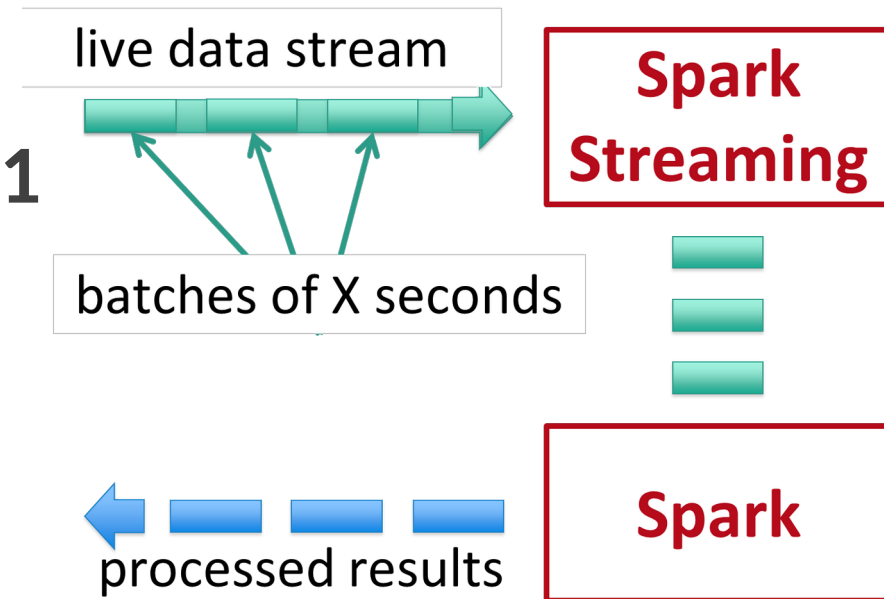


Discretised Stream Processing

Run a streaming computation as a **series of very small, deterministic batch jobs**

Batch sizes as low as $\frac{1}{2}$ second, **latency** ~ 1 second

Potential for **combining batch processing and streaming processing** in the same system



An example: getting hashtags from Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
```

DStream: a sequence of RDD representing a stream of data

Twitter Streaming API

batch @ t

batch @ t+1

batch @ t+2



tweets DStream



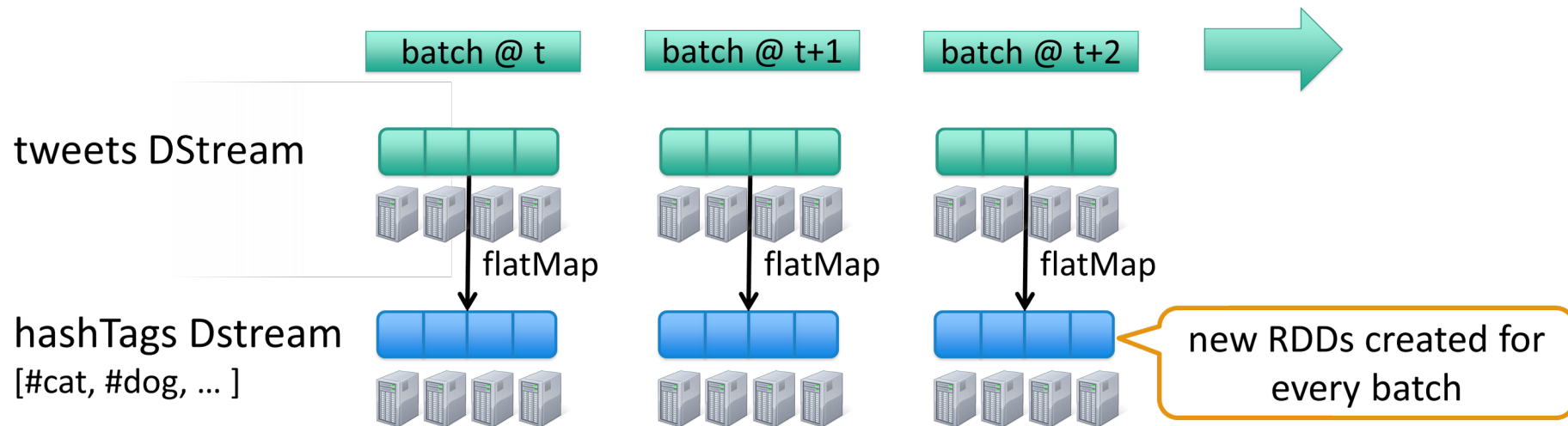
stored in memory as an RDD
(immutable, distributed)

An example: getting hashtags from Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)  
val hashTags = tweets.flatMap(status => getTags(status))
```

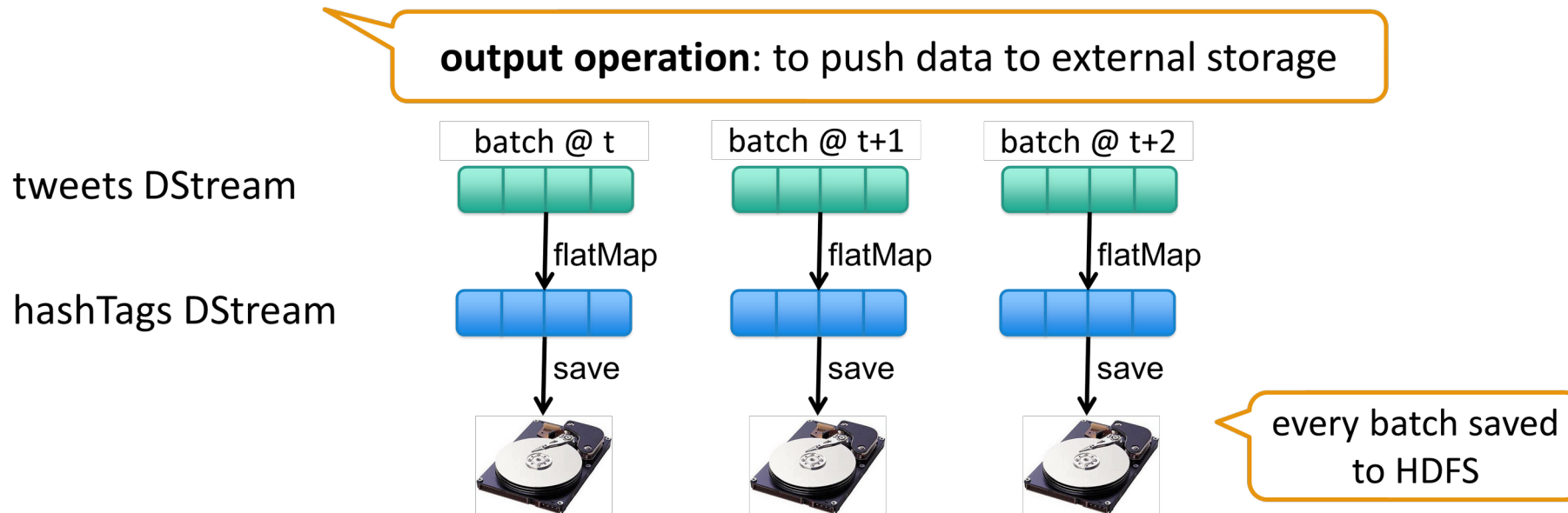
new DStream

transformation: modify data in one Dstream to create another DStream



An example: getting hashtags from Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)  
val hashTags = tweets.flatMap (status => getTags(status))  
hashTags.saveAsHadoopFiles("hdfs://...")
```



Key Concepts

Resilient Distributed Datasets (RDD) in practice:

- Write programs in terms of **operations** on **distributed datasets**
- **Partitioned collections** of objects **spread across a cluster**, stored in memory or on **disk**
- RDDs **built and manipulated** through a **diverse set of parallel transformations** (map, filter, join) and **actions** (count, collect, save)
- RDDs **automatically rebuilt on machine failure**

Questions and Answers
