Data Integration and Large Scale Analysis

Slides credit: Matthias Boehm - Shafaq Siddiqi

11- Stream Processing

Lucas Iacono. PhD. - 2024

TU Graz
SCIENCE
PASSION
TECHNOLOGY

ISDS

Part B

Large-Scale Data Management & Analysis

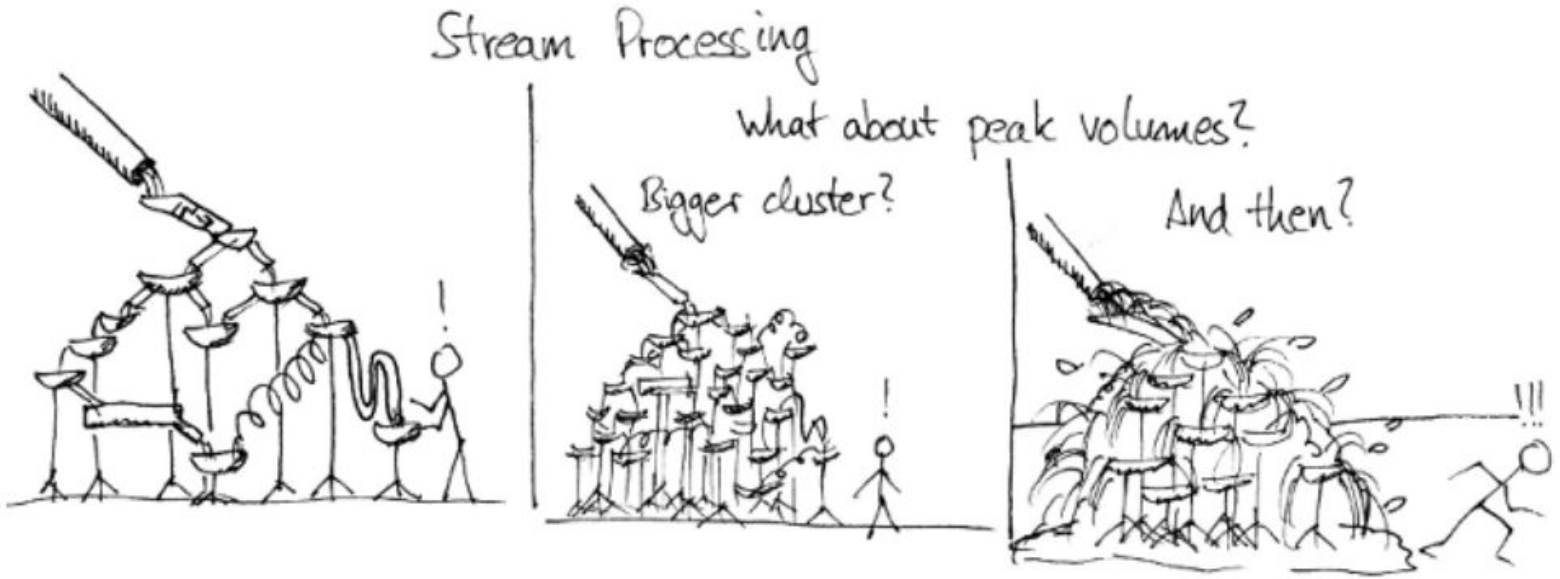
- LU4. Large-Scale Data Analysis
  - Distributed, Data-Parallel Computation [Dec 20]
  - **Distributed Stream Processing [Jan 10]**
  - Distributed Machine Learning Systems **[Jan 17]**

Agenda

- Data Stream Processing
- Distributed Stream Processing
- Data Stream Mining

Distributed Stream Processing

Data Stream Processing



Data Stream Processing: Terminology

Ubiquitous Data Streams

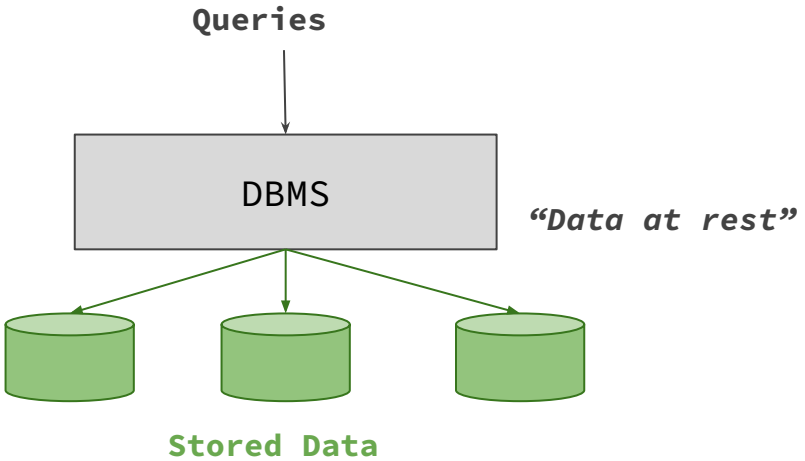
Event and message streams (e.g., clicks, twitter, IoT, Sensor Networks, Machines)

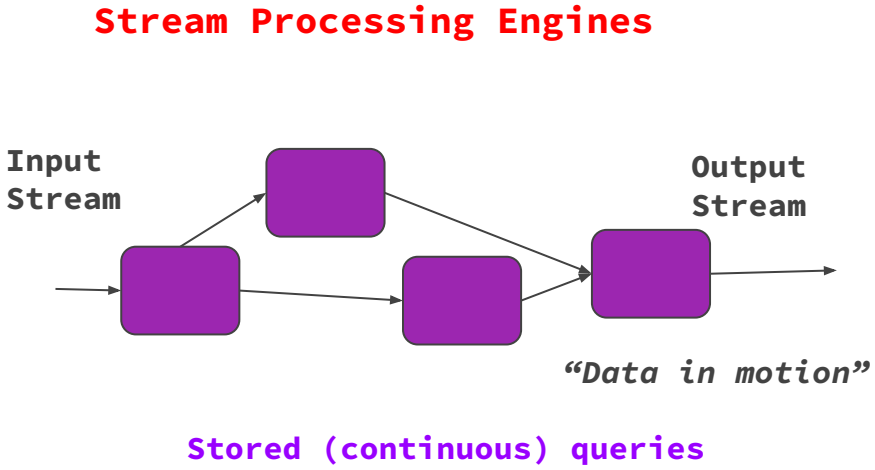
Characteristics	
Event Streams	**Message Streams**
• Time-oriented. • Immutable • Real-time processing	• System communication • Asynchronous • Structured

Data Stream Processing: Terminology

Stream Processing Architecture

- **Infinite input streams**, often with windows semantics
- Continuous queries





Data Stream Processing: Terminology

Use Cases

- **Monitoring and alerting** (notifications on events / patterns)
- **Real-time reporting** (aggregate statistics for dashboards)
- **Real-time ETL** and event-driven data updates
- Real-time decision making (fraud detection)
- Data stream mining (summary statistics w/ limited memory)

Data Stream

- Unbounded stream of data tuples S = (s1,s2, …) with si=(ti,di)
- *S={(t1 ,22.5),(t2 ,22.7),(t3 ,22.8),…}*

Data Stream Processing: Terminology

Real-time Latency Requirements

- **Real-time:** guaranteed task completion by a given deadline (30 fps)
- **Near Real-time:** few milliseconds to seconds
- In practice, used with much weaker meaning

Challenges in Real-Time Systems:

- Resource Constraints
- Latency
- Concurrency

Data Stream Processing: History of Stream Processing Systems

2000s

- **Data stream management systems:** STREAM(Stanford'01), Aurora (Brown/MIT/Brandeis'02), TelegraphCQ (Berkeley'03) but **mostly unsuccessful in industry/practice**
- **Message-oriented middleware** and **Enterprise Application Integration** (EAI): IBM Message Broker, SAP eXchange Infra

Academic DSMS	**MOM (Message-oriented middlewares)**	**EAI (Enterprise Application Integration)**
Real-time data stream processing	Asynchronous message communication between systems	Workflow orchestration for enterprise systems
SQL-like queries on streams	Sending simple or complex messages	Integration of complex applications
Time windows and event-based operations	Event-based message delivery	Event-driven task coordination
Academic prototypes	Industrial implementations (Kafka)	Enterprise suites IBM MQ

Data Stream Processing: History of Stream Processing Systems

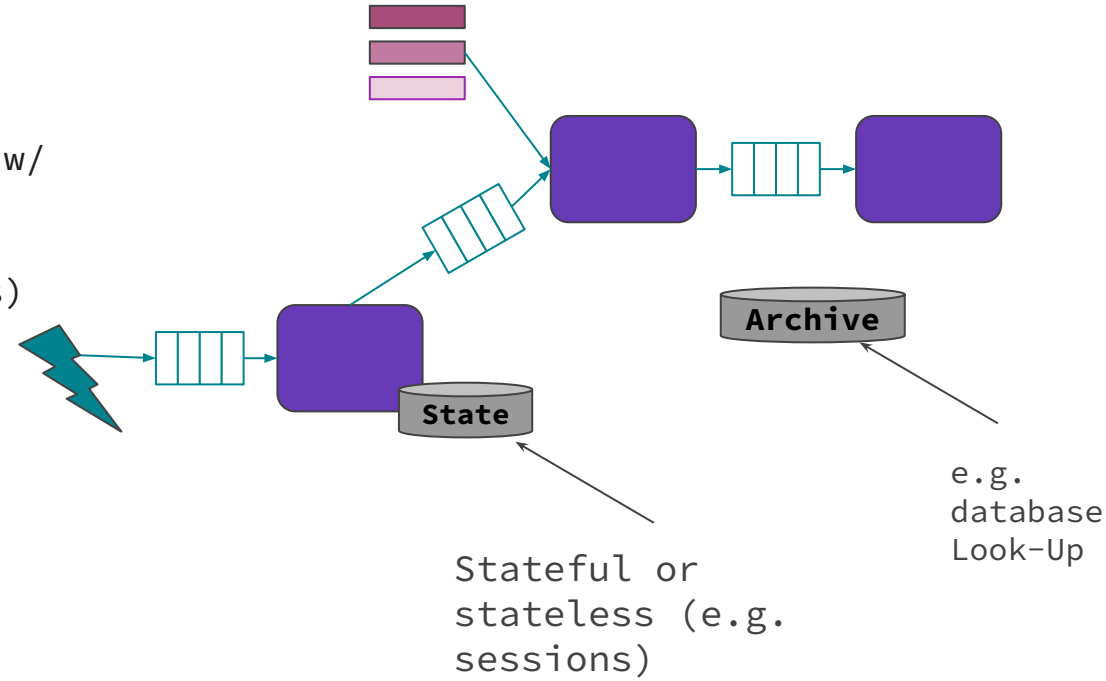
2010s

- **Distributed stream processing engines**, and “unified” batch/stream processing
- **Proprietary systems**: Google Cloud Dataflow, MS StreamInsight / Azure Stream Analytics, IBM InfoSphere Streams / Streaming Analytics, AWS Kinesis
- **Open-source systems:** Apache Spark Streaming (Databricks), **Apache Flink** (Data Artisans), **Apache Kafka** (Confluent), **Apache Storm**

System Architecture - Native Streaming

Basic System Architecture

- Data flow graphs (potentially w/ multiple consumers)
- **Nodes** asynchronous ops (w/ state) (e.g., separate threads)
- **Edges** data dependencies (tuple/message streams)
- **Push model** data production controlled by source



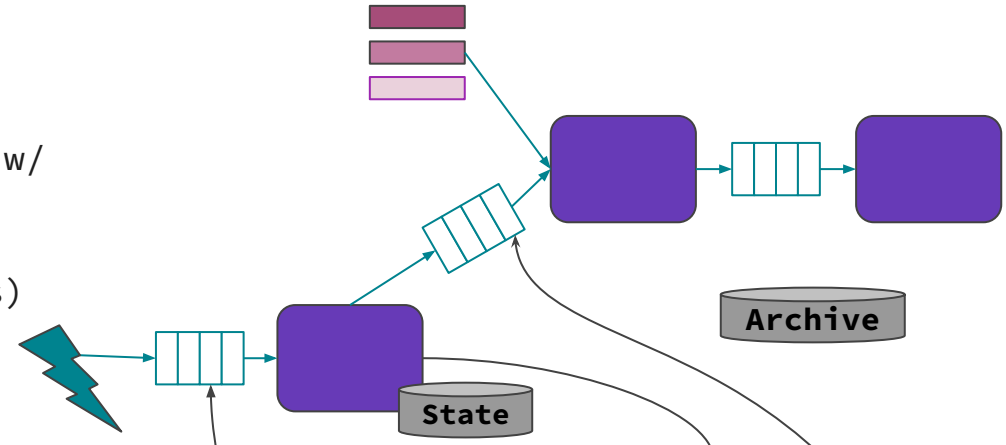
System Architecture - Native Streaming

Basic System Architecture

- Data flow graphs (potentially w/ multiple consumers)
- **Nodes** asynchronous ops (w/ state) (e.g., separate threads)
- **Edges** data dependencies (tuple/message streams)
- **Push model** data production controlled by source

Operator Model

- Read from input queue
- Write to potentially many output queues




```
while( !stopped )
{
r = in.dequeue(); // blocking
if( pred(r.A) ) // A==7
for( Queue o : out )
o.enqueue(r); // blocking
}
```

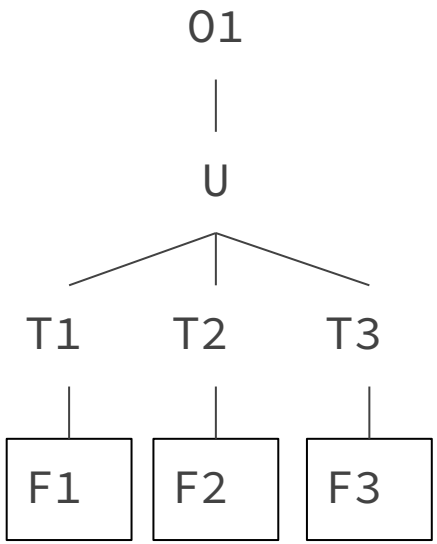
System Architecture - Sharing

Multi-Query Optimization

- Given **set of continuous queries** compile minimal DAG w/o redundancy **-> subexpression elimination -> avoid redundant operations and share intermediate results between queries to improve system efficiency.**
- **Operator** and **Queue** Sharing

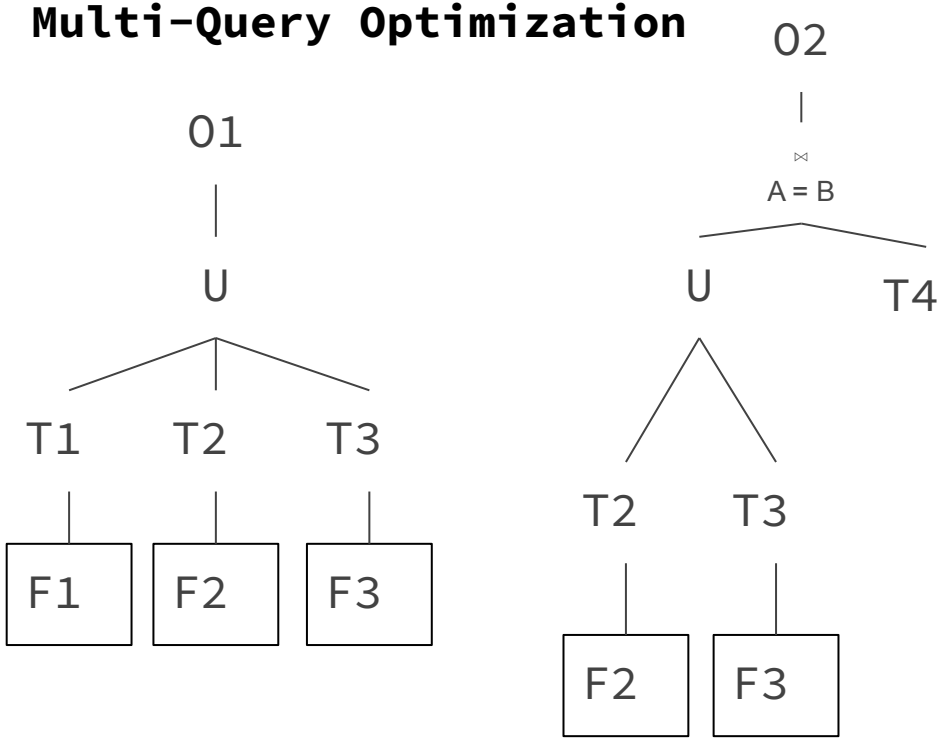
System Architecture - Sharing

Multi-Query Optimization



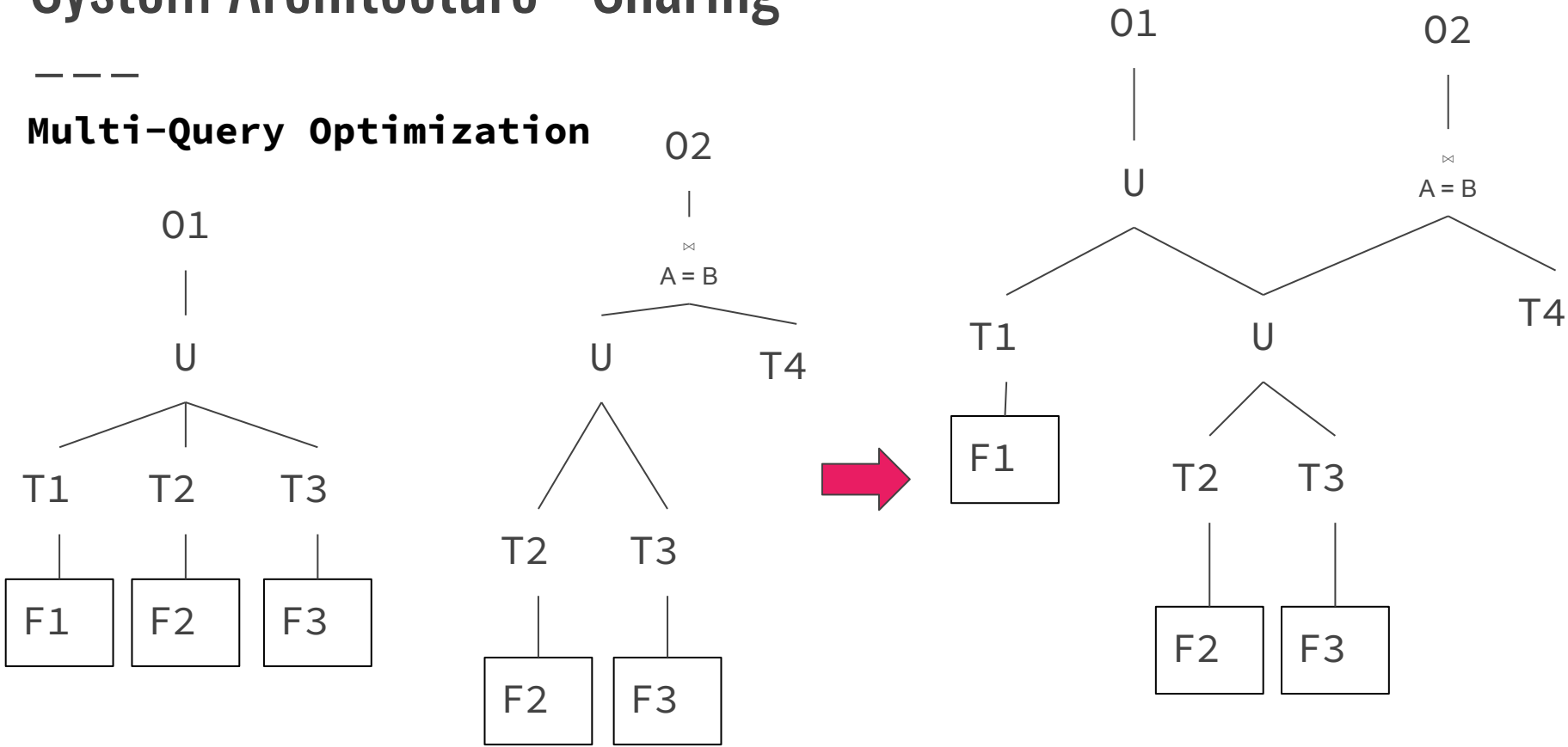
System Architecture - Sharing

Multi-Query Optimization



System Architecture - Sharing

Multi-Query Optimization

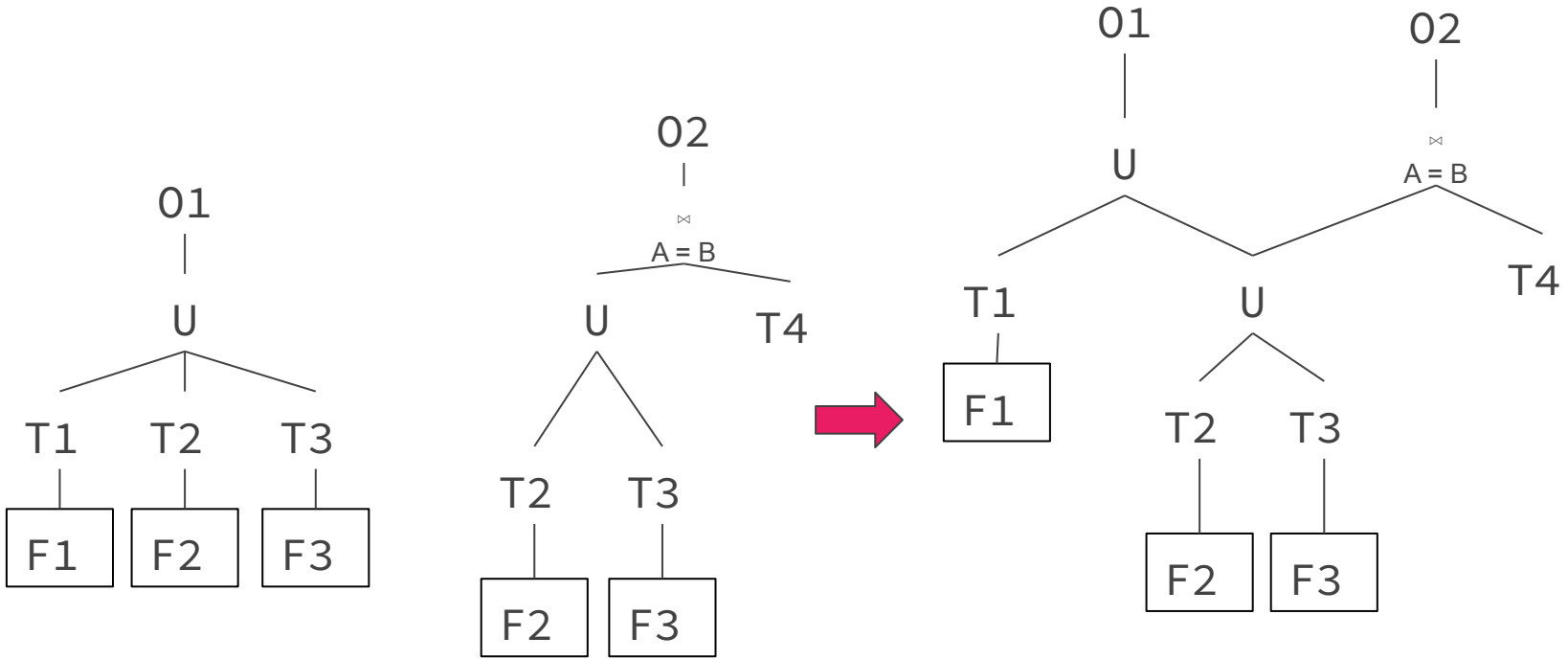


System Architecture - Sharing

Operator Sharing: complex ops w/ multiple predicates for adaptive reordering

Queue Sharing: share results with multiple queries

Multi-Query Optimization



System Architecture - Sharing

Operator sharing: complex ops w/ multiple predicates for adaptive reordering

```
// Query 1: Average of pressures greater than 100
average_pressures = FILTER "pressures" WHERE value > 100
                    GROUP BY 1-minute_interval
                    COMPUTE AVERAGE(value)

// Query 2: Count of pressures greater than 100
count_pressures = FILTER "pressures" WHERE value > 100
                  GROUP BY 1-minute_interval
                  COMPUTE COUNT(value)
// Execute both queries
EXECUTE average_pressures
EXECUTE count_pressures
```

Same filter applied twice in Q1 & Q2 -> duplicated work.

System Architecture - Sharing

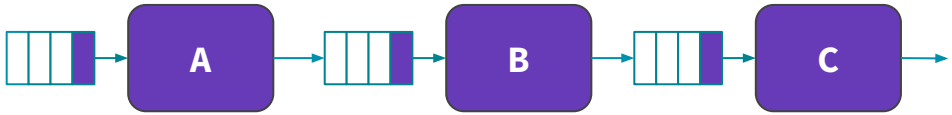
Operator sharing

```
// Query 1: Average of pressures greater than 1
average_pressures = FILTER "pressures" WHERE va
> 100
                    GROUP BY 1-minute_interv
                    COMPUTE AVERAGE(value)

// Query 2: Count of pressures greater than 100
count_pressures = FILTER "pressures" WHERE valu
1000
                  GROUP BY 1-minute_interval
                  COMPUTE COUNT(value)
// Execute both queries
EXECUTE average_pressures
EXECUTE count_pressures
```

```
//Apply the common filter only once
filtered_pressures = FILTER "pressures" WHERE
value > 1000

// Query 1: Average value of the filtered
pressures
average_pressures = GROUP filtered_pressures BY
1-minute_interval
                        COMPUTE AVERAGE(value)

// Query 2: Count of the filtered pressures
count_pressures = GROUP filtered_pressures BY
1-minute_interval
                      COMPUTE COUNT(value)

// Execute both queries in parallel
EXECUTE average_pressures
EXECUTE count_pressures
```

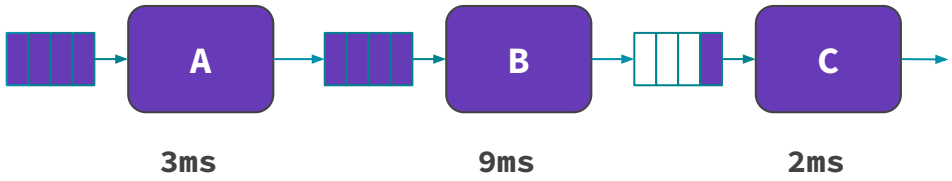
System Architecture - Handling Overload

- **Back Pressure**
  - Graceful handling of overload w/o data loss
  - **Slow down sources**
  - E.g. blocking queues



System Architecture - Handling Overload

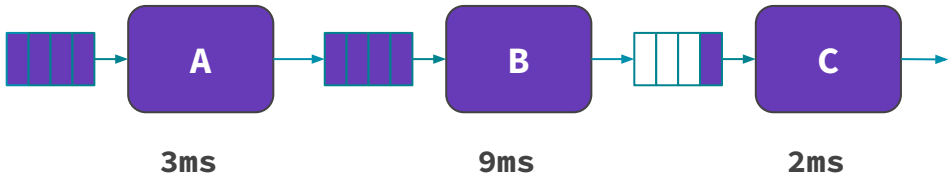
- **Back Pressure**
  - Graceful handling of overload w/o data loss
  - **Slow down sources**
  - E.g. blocking queues



Self-adjusting operator scheduling
Pipeline runs at rate of **slowest op**

System Architecture - Handling Overload

- **Back Pressure**
  - Graceful handling of overload w/o data loss
  - **Slow down sources**
  - E.g. blocking queues



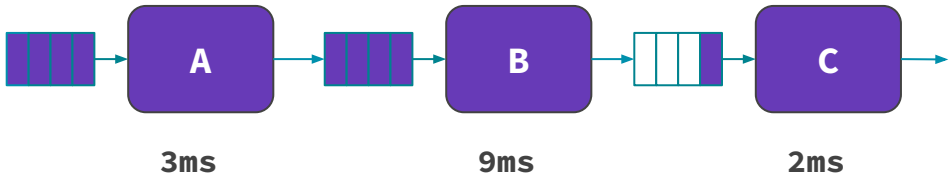
Self-adjusting operator scheduling
Pipeline runs at rate of slowest op

- **Load Shedding**
  - **Random-sampling**-based load shedding
  - **Relevance-based** load shedding
  - **Summary-based** load shedding (synopses)
  - Given SLA, select queries and shedding placement that minimize error and satisfy constraints

[Nesime Tatbul et al: Load Shedding in a Data Stream Manager. VLDB 2003]

System Architecture - Handling Overload

- **Back Pressure**
  - Graceful handling of overload w/o data loss
  - **Slow down sources**
  - E.g. blocking queues



Self-adjusting operator scheduling
Pipeline runs at rate of slowest op

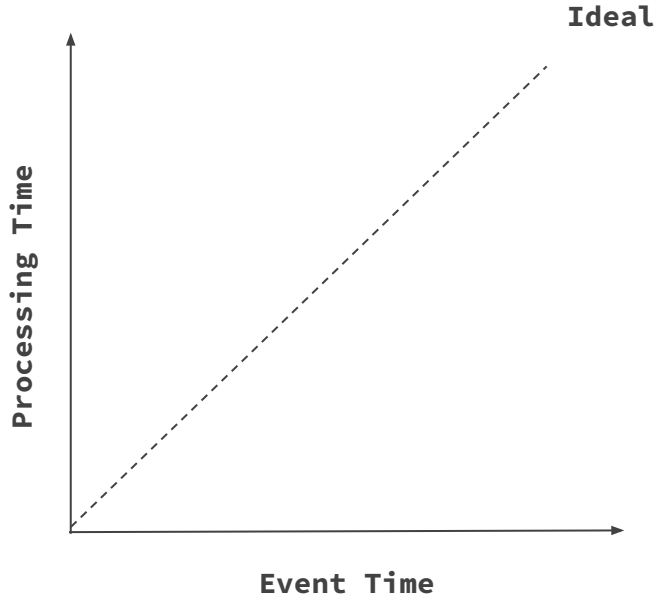
- **Load Shedding**
  - **Random-sampling**-based load shedding
  - **Relevance-based** load shedding
  - **Summary-based** load shedding (synopses)
  - Given SLA, select queries and shedding placement that minimize error and satisfy constraints

[Nesime Tatbul et al: Load Shedding in a Data Stream Manager. VLDB 2003]

- **Distributed Stream Processing**
  - Data flow partitioning (distribute the **query**)
  - Key range partitioning (distribute the **data** stream)

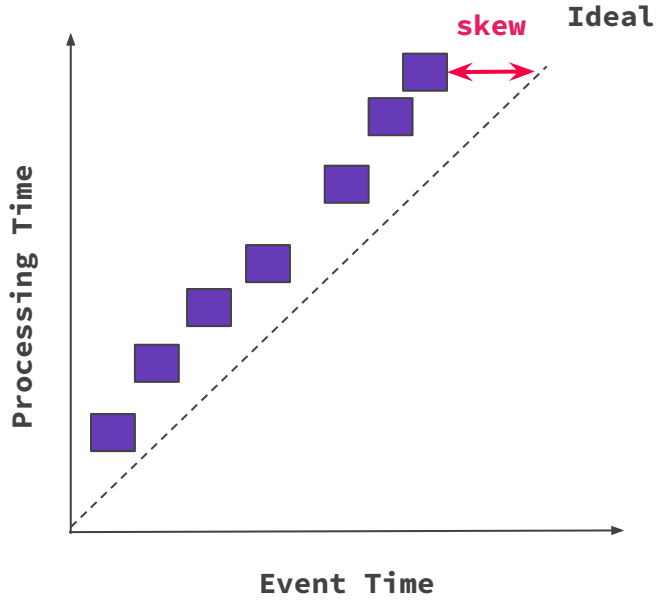
Time (Event, System, Processing)

- **Event Time**
  - Real time when the event/data item was created
- **Ingestion**
  - System time when the data item was received
- **Processing Time**
  - System time when the data item is processed



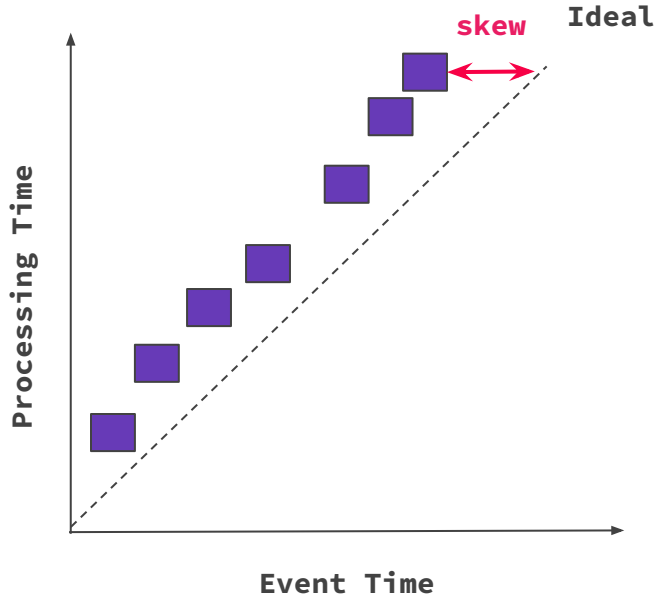
Time (Event, System, Processing)

- **Event Time**
  - Real time when the event/data item was created
- **Ingestion**
  - System time when the data item was received
- **Processing Time**
  - System time when the data item is processed



Time (Event, System, Processing)

- **Event Time**
  - Real time when the event/data item was created
- **Ingestion**
  - System time when the data item was received
- **Processing Time**
  - System time when the data item is processed
- **In practice**
  - **Delayed and unordered** data items
  - Use of heuristics (e.g **watermarks**, **delays**)
  - Use of more complex triggers (speculative and late results)



Durability and Consistency Guarantees

- **At Most Once**
  - **“Send and forget”**, ensure data is never counted twice
  - Might cause data loss on failures

Durability and Consistency Guarantees

- **At Most Once**
  - **“Send and forget”**, ensure data is never counted twice
  - Might cause data loss on failures
- **At Least Once**
  - **“Store and forward”** deliver the message until reception of the acknowledgements from receiver
  - Might create incorrect state (processed multiple times)

Durability and Consistency Guarantees

- **At Most Once**
  - **“Send and forget”**, ensure data is never processed twice
  - Might cause data loss on failures
- **At Least Once**
  - **“Store and forward”** deliver the message until reception of the acknowledgements from receiver
  - Might create incorrect state (processed multiple times)
- **Exactly Once**
  - **“Store and forward” w/ guarantees** regarding state updates and sent msgs
  - Often via dedicated transaction mechanisms (**hand-shaking protocols**)

Window Semantics

- **Windowing Approach**
  - Many operations like joins/aggregation **undefined over unbounded streams**
  - Compute operations over windows of (a) time or (b) elements counts

Window Semantics

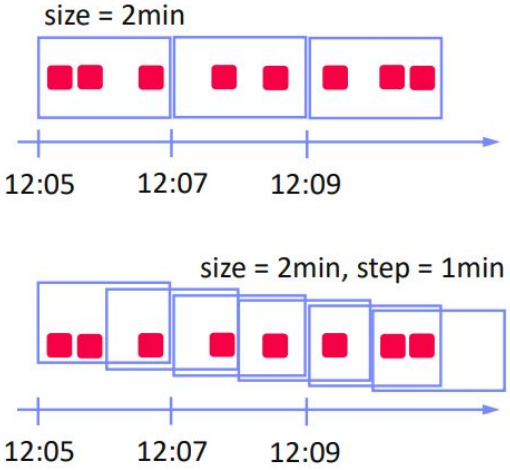
- **Windowing Approach**
  - Many operations like joins/aggregation **undefined over unbounded streams**
  - Compute operations over windows of (a) time or (b) elements counts

- **Tumbling Window**
  - Every data item is only part of a single window
  - Aka Jumping window

Window Semantics

- **Windowing Approach**
  - Many operations like joins/aggregation **undefined over unbounded streams**
  - Compute operations over windows of (a) time or (b) elements counts

- **Tumbling Window**
  - Every data item is only part of a single window
  - Aka Jumping window
- **Sliding Window**
  - Time- or tuple-based sliding windows
  - Insert new and expire old data items



Stream Joins I

Basic Stream Join

- **Tumbling window:** use classic join methods
- **Sliding window** (symmetric for both R and S)
  - Applies to arbitrary join pred

Stream Joins I

Basic Stream Join

- **Tumbling window:** use classic join methods
- **Sliding window** (symmetric for both R and S)
  - Applies to arbitrary join pred

Example: join two streams where an event in stream R (sensor_reading) should be matched with a value in stream S (control command) in a 3-second sliding window.

For each new r in R:

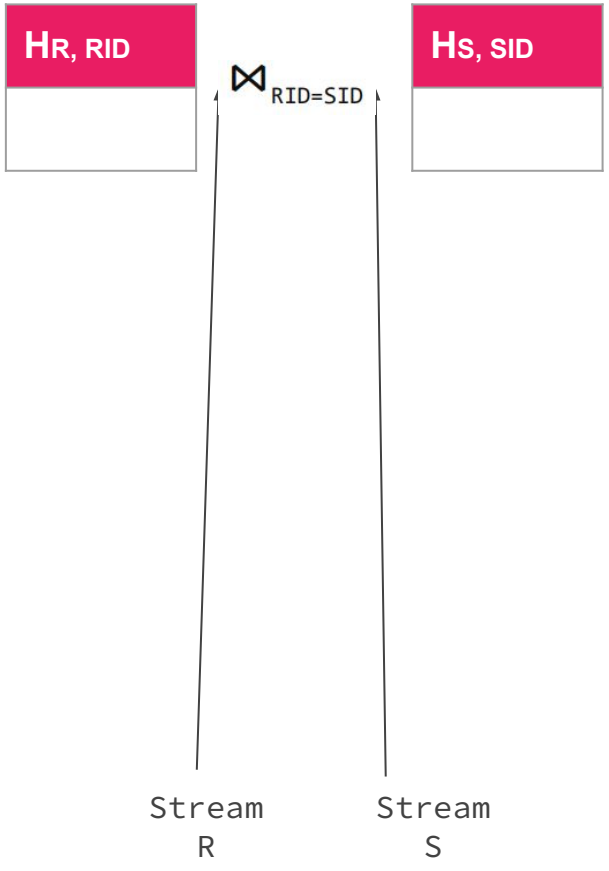
a. **Scan** window of stream S to find match tuples

b. **Insert** new r into window of stream R

c. **Invalidate** expired tuples in window of stream R

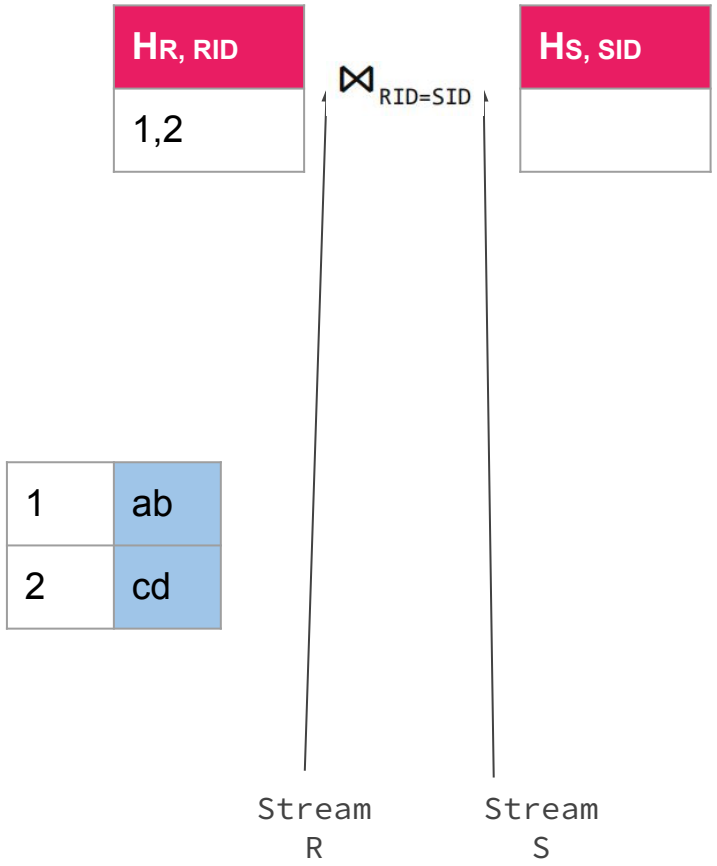
Stream Joins II

- **Double-Pipelined Hash Join**
  - Join of bounded streams (or unbounded w/ invalidation)
  - Equi join predicate, symmetric and non-blocking
  - For every incoming tuple (e.g. left): probe (right)+emit, and build (left)



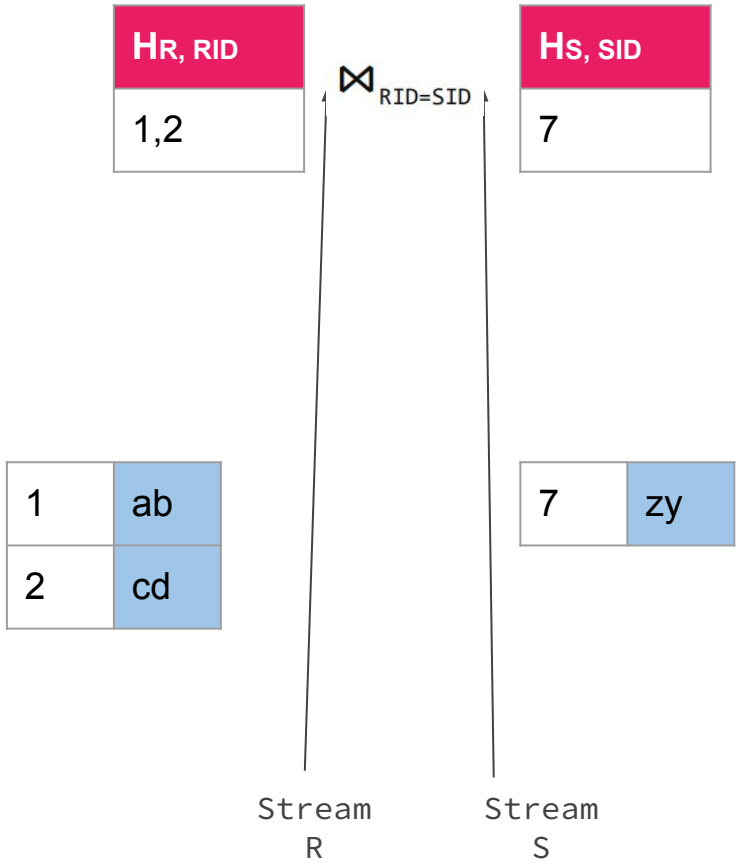
Stream Joins II

- **Double-Pipelined Hash Join**
  - Join of bounded streams (or unbounded w/ invalidation)
  - Equi join predicate, symmetric and non-blocking
  - For every incoming tuple (e.g. left): probe (right)+emit, and build (left)



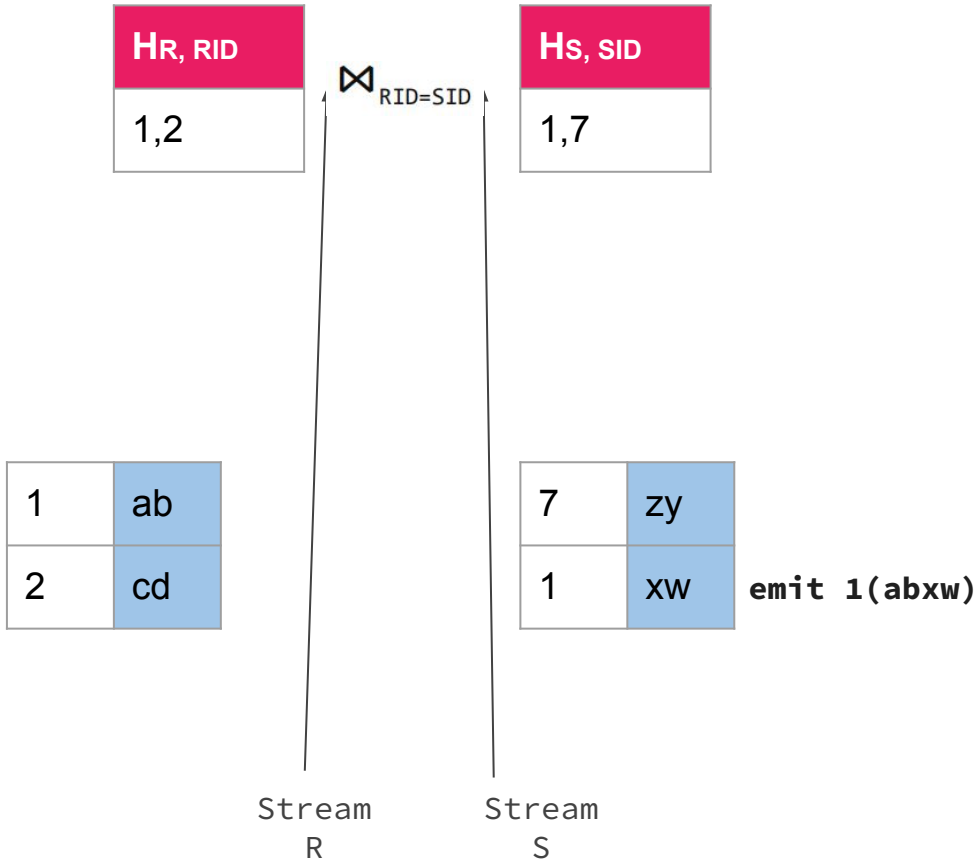
Stream Joins II

- **Double-Pipelined Hash Join**
  - Join of bounded streams (or unbounded w/ invalidation)
  - Equi join predicate, symmetric and non-blocking
  - For every incoming tuple (e.g. left): probe (right)+emit, and build (left)



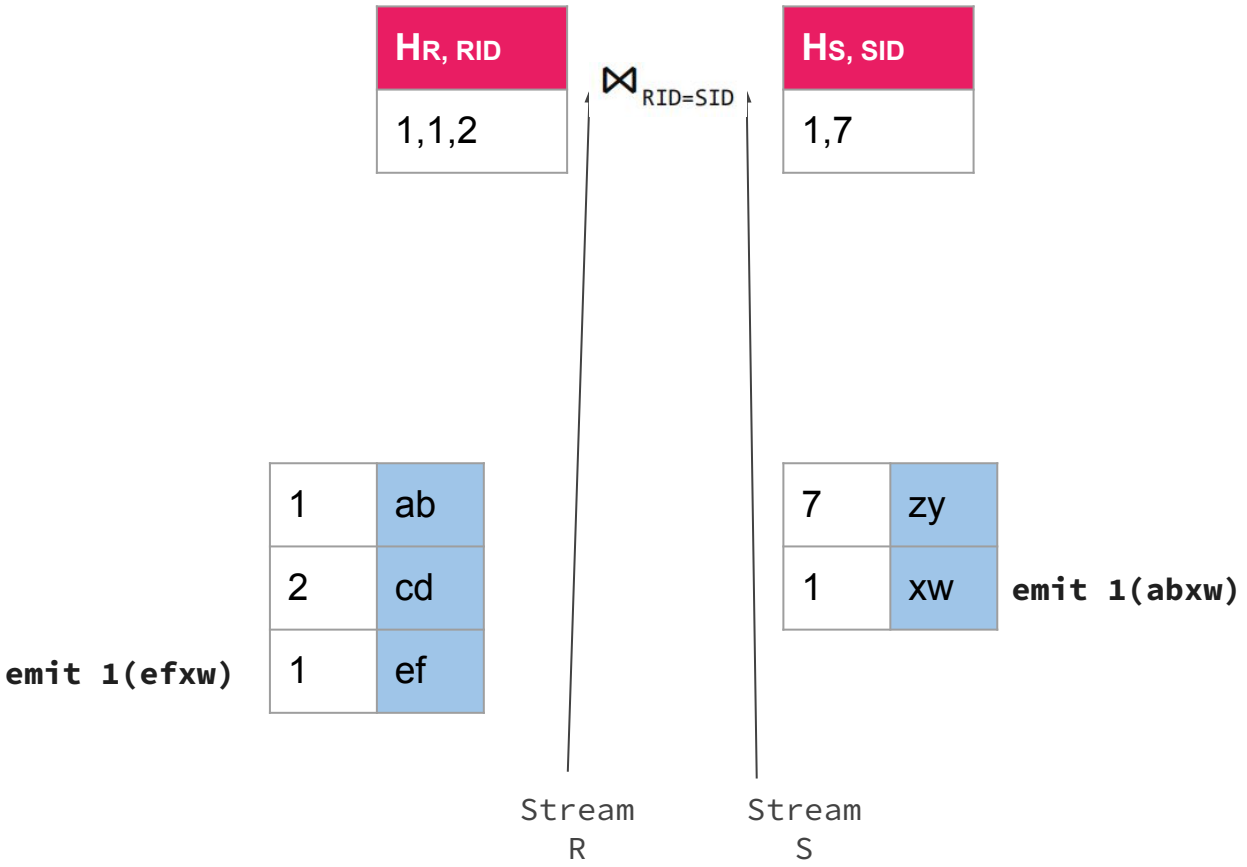
Stream Joins II

- **Double-Pipelined Hash Join**
  - Join of bounded streams (or unbounded w/ invalidation)
  - Equi join predicate, symmetric and non-blocking
  - For every incoming tuple (e.g. left): probe (right)+emit, and build (left)



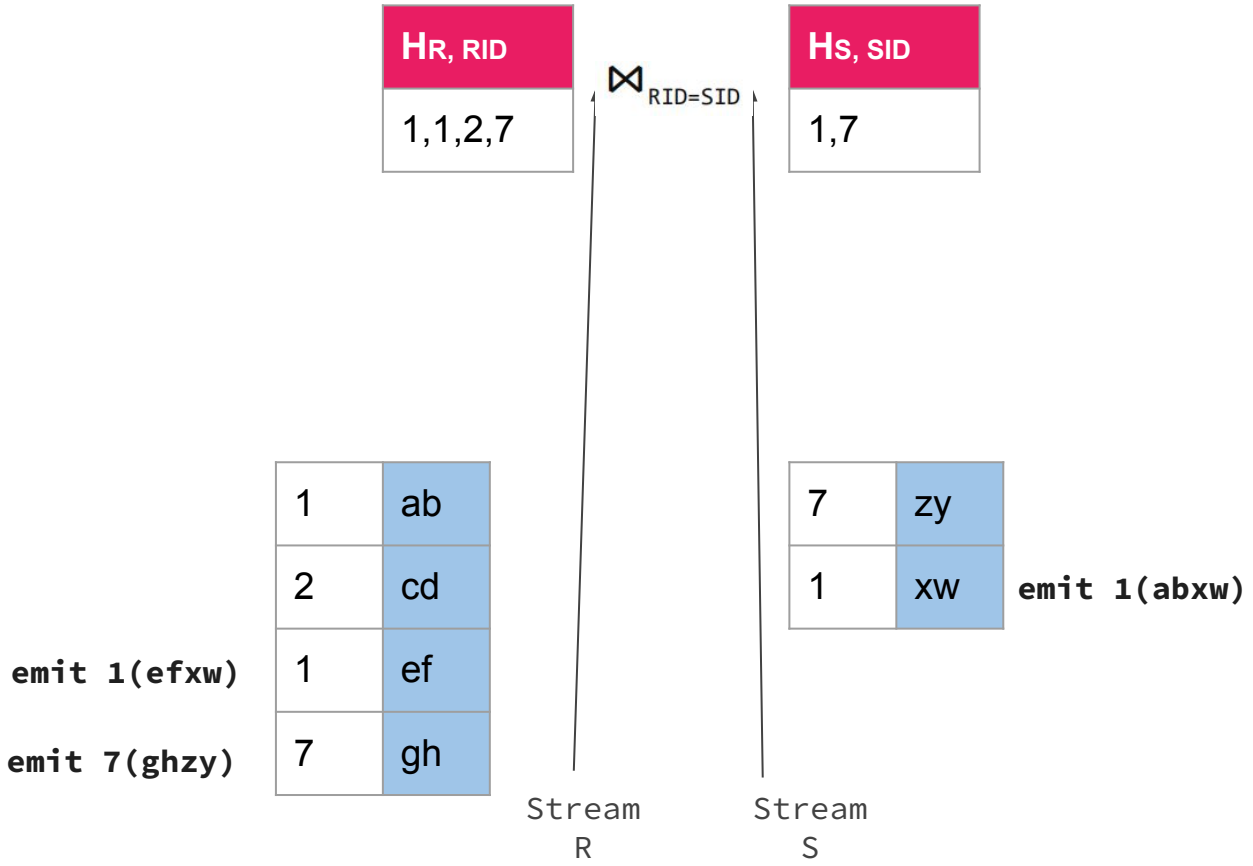
Stream Joins II

- **Double-Pipelined Hash Join**
  - Join of bounded streams (or unbounded w/ invalidation)
  - Equi join predicate, symmetric and non-blocking
  - For every incoming tuple (e.g. left): probe (right)+emit, and build (left)



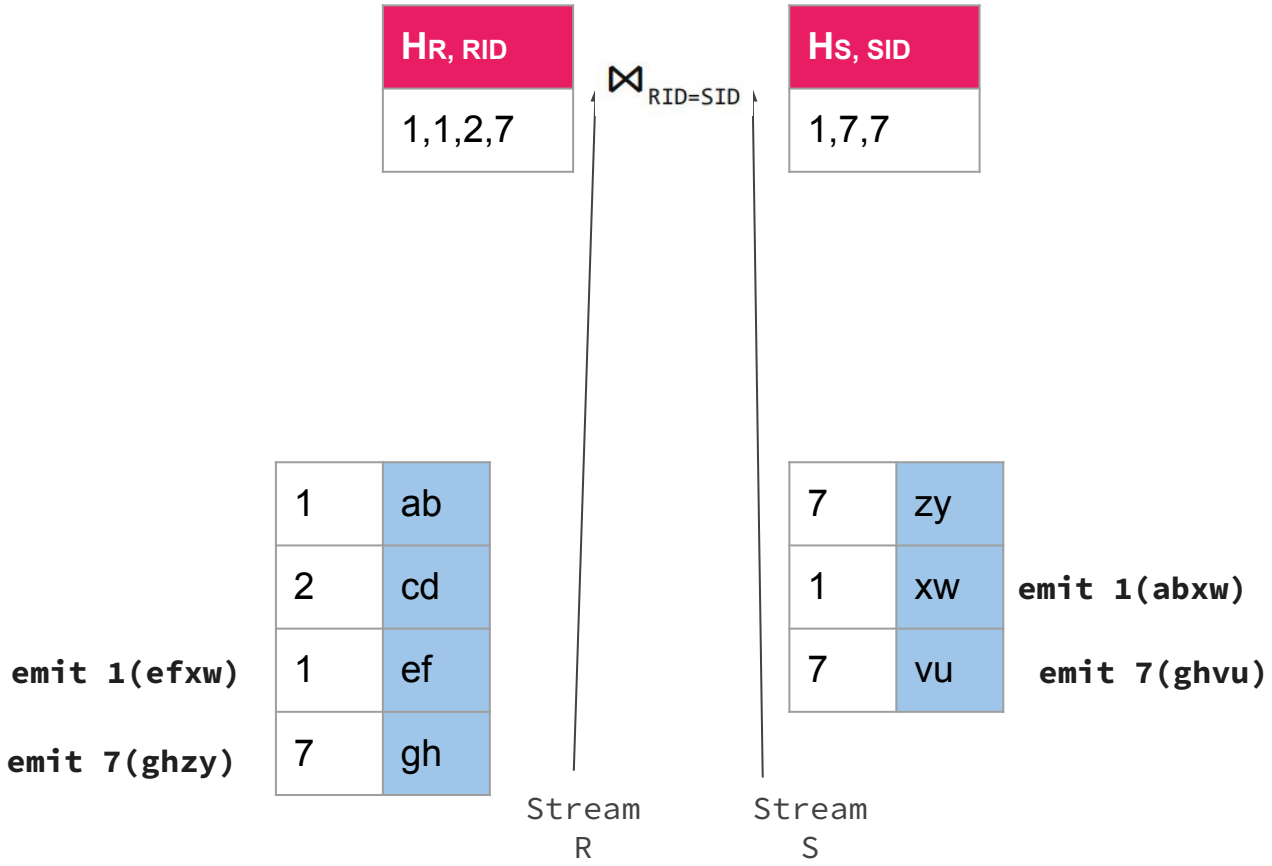
Stream Joins II

- **Double-Pipelined Hash Join**
  - Join of bounded streams (or unbounded w/ invalidation)
  - Equi join predicate, symmetric and non-blocking
  - For every incoming tuple (e.g. left): probe (right)+emit, and build (left)



Stream Joins II

- **Double-Pipelined Hash Join**
  - Join of bounded streams (or unbounded w/ invalidation)
  - Equi join predicate, symmetric and non-blocking
  - For every incoming tuple (e.g. left): probe (right)+emit, and build (left)



Distributed Stream Processing

Query-Aware Stream Partitioning

Example Use Case

- AT&T network monitoring with Gigascope (e.g., OC768 network)
- 2x40 Gbit/s traffic 112M packets/s 26 cycles/tuple on 3Ghz CPU
- Complex query sets (apps w/ ~50 queries) and massive data rates

Query-Aware Partitioning for Monitoring Massive Network Data Streams

Theodore Johnson
AT&T Labs-Research
johnsont@research.att.com

S. Muthukrishnan
Rutgers University
muthu@cs.rutgers.edu

Vladislav Shkapenyuk
AT&T Labs-Research
vlad@research.att.com

Oliver Spatscheck
AT&T Labs-Research
spatsch@research.att.com

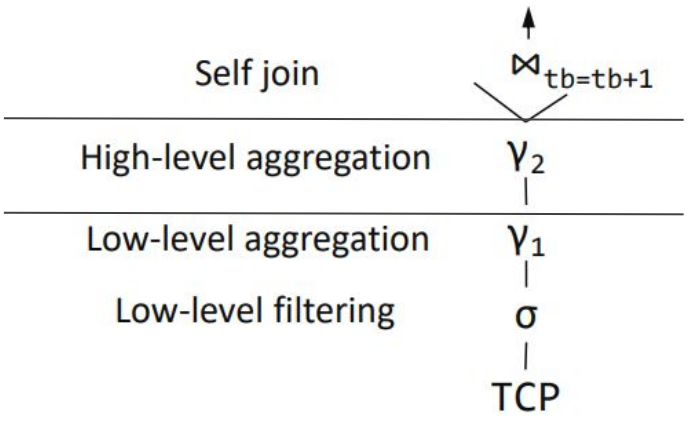
ABSTRACT

T. Johnson et.al, Query-aware partitioning for monitoring massive network data streams. **SIGMOD 2008**

Query-Aware Stream Partitioning

T. Johnson et.al, Query-aware partitioning for monitoring massive network data streams. **SIGMOD 2008**

Baseline Query Execution Plan



Query FLOW PAIRS: users generating heavy-loads during an specific period

Query HEAVY FLOWS: maximum queries per IP

Query FLOWS: composed by 3 sub-queries about how many requests are made by each IP in the network (per minute)

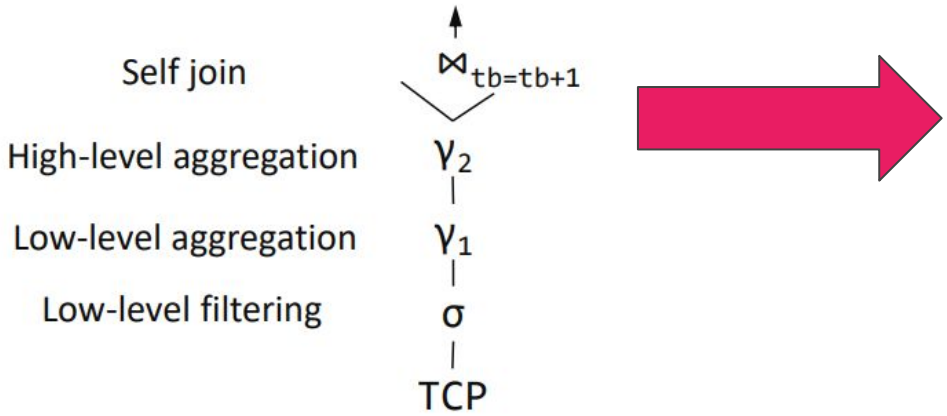
Query-Aware Stream Partitioning

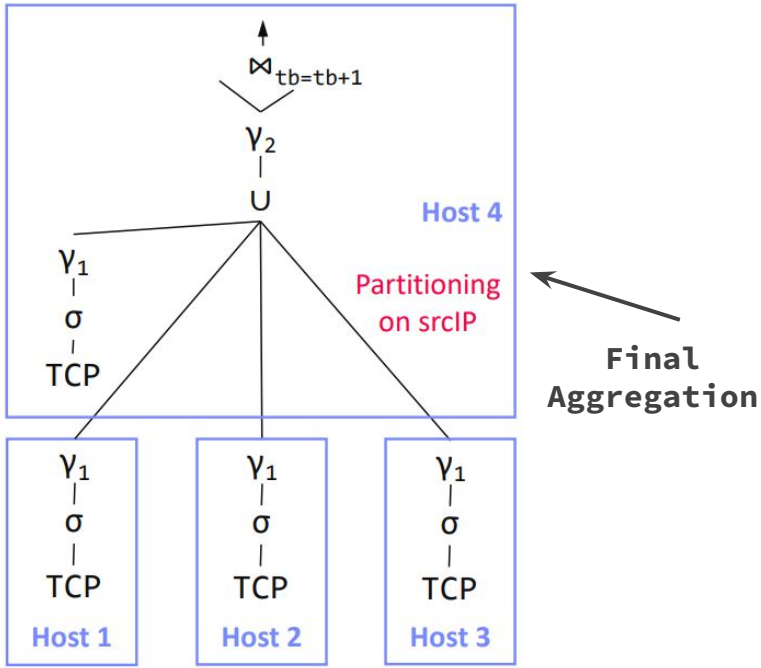
T. Johnson et.al, Query-aware partitioning for monitoring massive network data streams. **SIGMOD 2008**

Solution divide in sub-queries and distribute

Optimized Plan:

- Distributed Plan operators
- Pipeline and task parallelism
- **Not always enough**



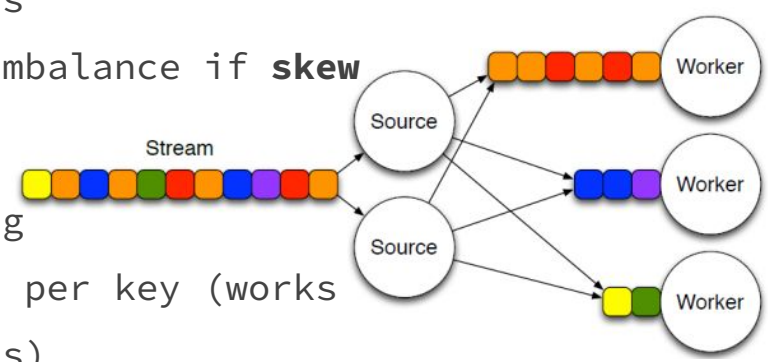


Stream Group Partitioning

- **Large-Scale Stream Processing**
  - Limited pipeline parallelism and task parallelism (independent subqueries)
  - **Combine with data-parallelism over stream groups**
- **Shuffle Grouping**
  - Tuples are randomly distributed across consumer tasks
  - Good load balance

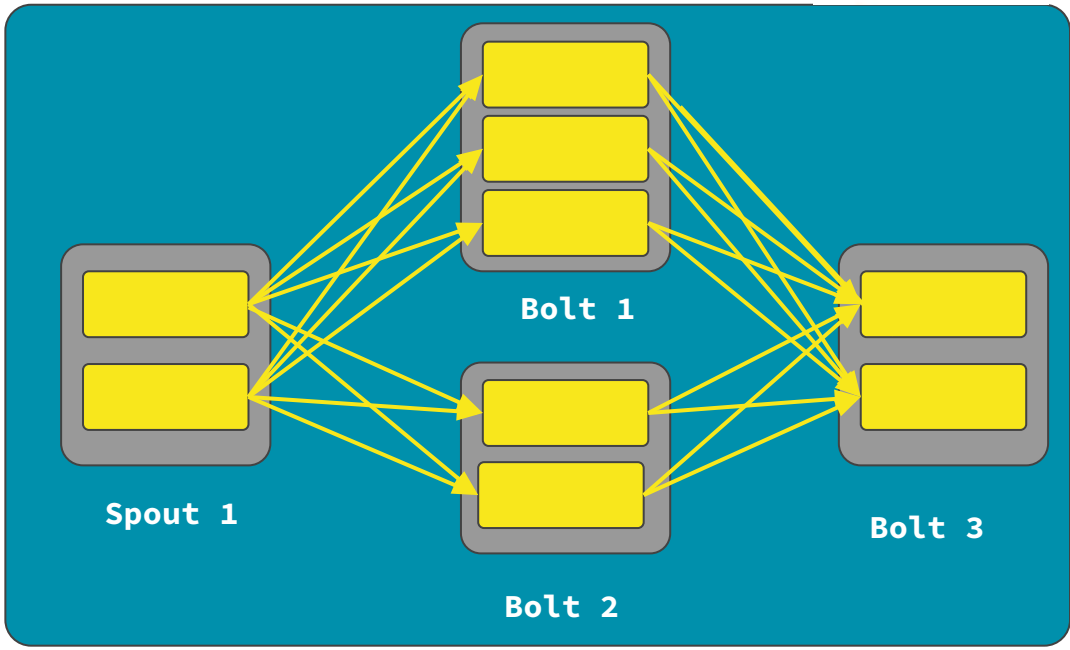
Stream Group Partitioning

- **Fields Grouping**
  - Tuples partitioned by grouping attributes
  - Guarantees order within **keys**, but load imbalance if **skew**
- **Partial Key Grouping**
  - Apply **“power of two choices”** to streaming
  - Key splitting: select among **2 candidates** per key (works for all associative aggregation functions)



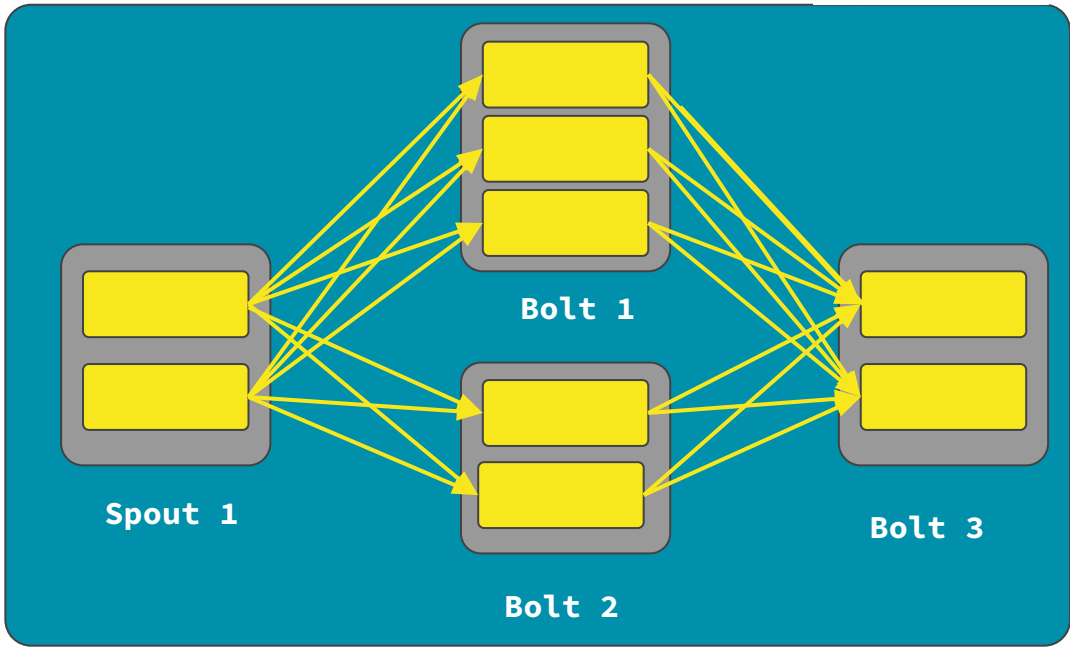
Example Apache Storm

- **Example Topology DAG**
  - **Spouts:** sources of streams
  - **Bolts:** UDF compute ops
  - Tasks mapped to worker processes and executors (threads)



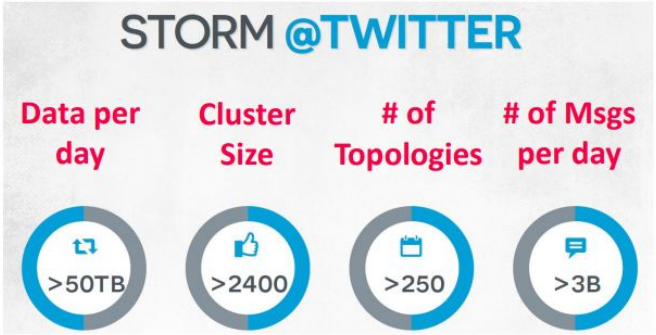
Example Apache Storm

- **Example Topology DAG**
  - **Spouts:** sources of streams
  - **Bolts:** UDF compute ops
  - Tasks mapped to worker processes and executors (threads)




```
Config conf = new Config();
conf.setNumWorkers(3);
topBuilder.setSpout("Spout1", new FooS1(), 2);
topBuilder.setBolt("Bolt1", new FooB1(), 3).shuffleGrouping("Spout1");
topBuilder.setBolt("Bolt2", new FooB2(), 2).shuffleGrouping("Spout1");
topBuilder.setBolt("Bolt3", new FooB3(), 2)
.shuffleGrouping("Bolt1").shuffleGrouping("Bolt2");
StormSubmitter.submitTopology(..., topBuilder.createTopology());
```

Example Twitter Heron

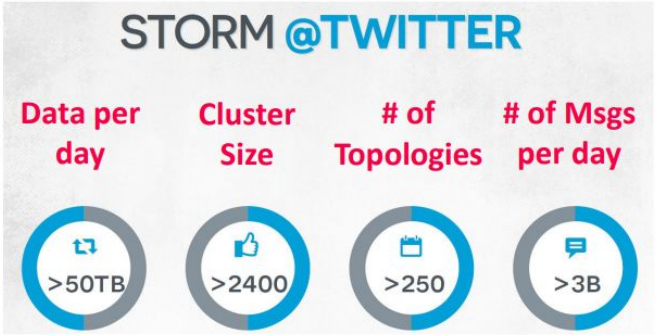


- **Motivation**
  - **Heavy use of Apache Storm** at Twitter
  - Issues: **debugging, performance**, shared **cluster resources**, back pressure mechanism


Twitter Heron: Stream Processing at Scale

Sanjeev Kulkarni et al:
Twitter Heron: Stream Processing at Scale.
SIGMOD 2015

Example Twitter Heron

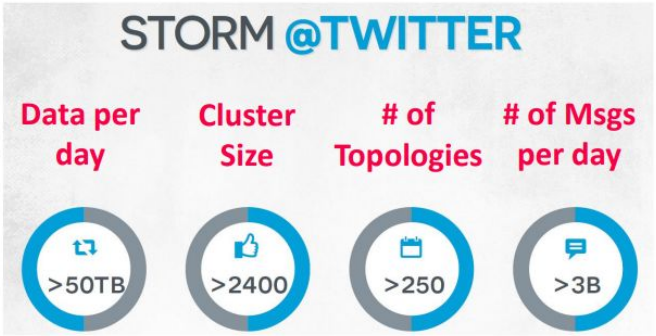


- **Motivation**
  - **Heavy use of Apache Storm** at Twitter
  - Issues: **debugging, performance**, shared **cluster resources**, back pressure mechanism
- **Twitter Heron**
  - API-compatible distributed streaming engine
  - **De-facto streaming engine at Twitter** since 2014


Twitter Heron: Stream Processing at Scale

Sanjeev Kulkarni et al: Twitter Heron: Stream Processing at Scale. **SIGMOD 2015**

Example Twitter Heron



- **Motivation**
  - **Heavy use of Apache Storm** at Twitter
  - Issues: **debugging, performance**, shared **cluster resources**, back pressure mechanism
- **Twitter Heron**
  - API-compatible distributed streaming engine
  - **De-facto streaming engine at Twitter** since 2014
- **Dhalion (Heron Extension)**
  - Automatically reconfigure Heron topologies to meet throughput SLO
- **Now back pressure implemented in Apache Storm 2.0 (May 2010)**


Twitter Heron: Stream Processing at Scale

Sanjeev Kulkarni et al: Twitter Heron: Stream Processing at Scale. **SIGMOD 2015**

Discretized Stream (Batch) Computation

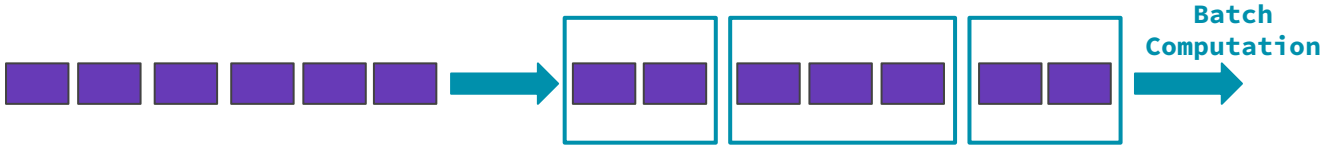
- **Motivation**
  - **Fault tolerance** (low overhead, fast recovery)
  - Combination w/ **distributed batch analytics**

Matei Zaharia et al: Discretized streams: fault-tolerant streaming computation at scale. SOSP 2013

Discretized Stream (Batch) Computation

Matei Zaharia et al: Discretized streams: fault-tolerant streaming computation at scale. SOSP 2013

- **Motivation**
  - **Fault tolerance** (low overhead, fast recovery)
  - Combination w/ **distributed batch analytics**
- **Discretized Streams (DStream)**
  - Batching of input tuples (100ms - 1s) based on ingest time.
  - Periodically run distributed jobs of stateless, deterministic tasks -> DStreams
  - State of all tasks materialized as RDDs, recovery via lineage



- **Criticism: High latency, required for batching**

Unified Batch/Streaming Engines

- **Apache Spark Streaming (Databricks)**
  - Micro-batch computation with exactly-once guarantee
  - Back-pressure and water mark mechanisms
  - Structured streaming via SQL (2.0), continuous streaming (2.3)
- **Apache Flink (Data Artisans, now Alibaba)**
  - Tuple-at-a-time with exactly-once guarantee
  - Back-pressure and water mark mechanisms
  - Batch processing viewed as special case of streaming

Unified Batch/Streaming Engines

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

- **Google Cloud Dataflow**
  - Tuple-at-a-time with exactly-once guarantee
  - MR FlumeJava MillWheel Dataflow
  - Google's fully managed batch and stream service
- **Apache Beam (API+SDK from Dataflow)**
  - Abstraction for Spark, Flink, Dataflow w/ common API, etc
  - Individual runners for the different runtime frameworks

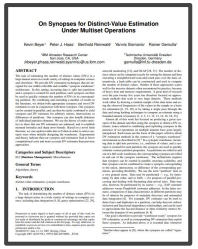
Data Stream Mining

Overview Stream Mining

- **Streaming Analysis Model**
  - Independent of actual storage model and processing system
  - Unbounded stream of data item **S = (s1, s2, …)**
  - Evaluate function **f(S)** as aggregate over stream or window of stream
  - Standing vs ad-hoc queries
- **Recap: Classification of Aggregates**
  - **Additive** aggregation functions (**SUM, COUNT**)
  - **Semi-additive** aggregation functions (**MIN, MAX**)
  - **Additively computable** aggregation functions (**AVG, STDDEV, VAR**)
  - **Aggregation functions (e.g SORTING) -> approximations**
- **Example:** Approximate # Distinct Items (e.g., **KMV**)

Stream Mining KMV

Beyer et al. On synopses for distinct-value estimation under multiset operations. **SIGMOD 2007**



- **Definition**
  - Estimate the number of unique elements (cardinality) in a large data stream.
  - KMV stores only a small, fixed-size subset of values derived from a hash function and uses that subset to make an accurate estimate.
- **How it Works:**
  - **Hashing:** Each stream element is passed through a hash function that generates a numeric value between **0 and 1**.
  - **Store** K Smallest Hashes: KMV keeps only the **K smallest hash values**.
  - **Estimate** Cardinality $\hat{N}$ using the formula:

$$\hat{N} \approx \frac{K-1}{h_{(K)}}$$

Stream Mining KMV

Data Stream: {A, B, C, A, D, B, E}

Hash Values:

$$h(A) = 0.15$$
$$h(B) = 0.45$$
$$h(C) = 0.22$$
$$h(D) = 0.70$$
$$h(E) = 0.10$$

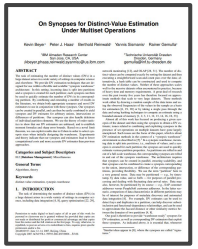
Select K = 3: The 3 smallest hash values are: 0.10, 0.15, 0.22.

Estimate:
The largest hash in this set is 0.22, so:

$$\hat{N} \approx \frac{K-1}{h_{(K)}} = \frac{3-1}{0.22} \approx 9.09 \text{ (estimated distinct values)}$$

Note: The real number of distinct values is 5 ({A, B, C, D, E}), and the difference comes from the small K. Increasing K improves accuracy.

Beyer et al.
On synopses for distinct-value estimation under multiset operations.
SIGMOD 2007


On Synopses for Distinct-Value Estimation Under Multiset Operations

Summary and Q&A

Summary and Q&A

- **Summary and Q&A**
  - Data Stream Processing
  - Distributed Stream Processing
  - Data Stream Mining
- **Next Lectures**
  - Distributed Machine Learning **[Jan 17]**
  - **Written Exam [Feb 07]**

Vielen Dank!