

Cloud Computing and Big Data

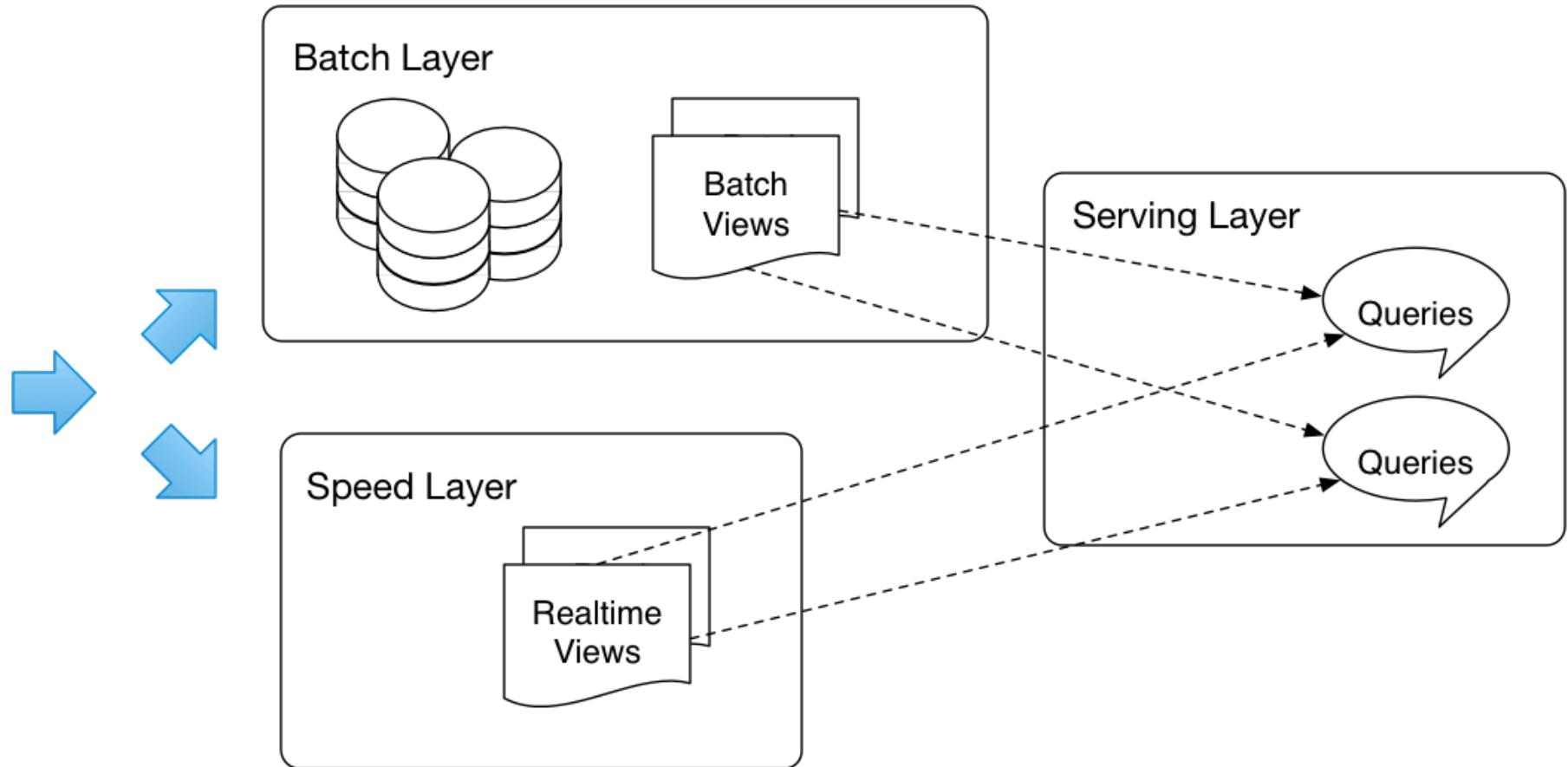
Realtime Big Data

Oxford University
Software Engineering
Programme
July 2018



© Paul Fremantle 2015. This work is licensed under a Creative Commons
Attribution-NonCommercial-ShareAlike 4.0 International License
See <http://creativecommons.org/licenses/by-nc-sa/4.0/>

Recap on the Lambda Architecture



Streaming

- **Continuous data flow**
 - “Unbounded streams of data”
- **Usually uses a message distribution system**
 - JMS
 - Apache Kafka
 - MQTT
 - Etc
- **An unbounded set of events with time**
 - $\langle t_1, E_1 \rangle, \langle t_2, E_2 \rangle, \dots, \langle t_n, E_n \rangle, \dots$

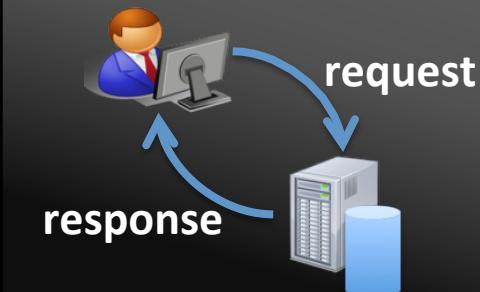
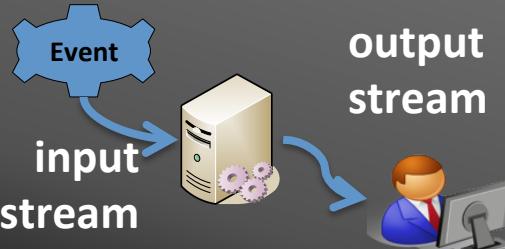


Stream processing categorization

- **Simple event processing**
 - Working on an event at a time
 - e.g. filter out all events where the wind speed > 50 mph
- **Event stream processing**
 - Time-based processing of a single stream of events
 - Average wind speed over the last hour compared to the average over the last day
- **Complex Event Processing**
 - Correlation of events across different streams
 - Emergency calls correlated with wind speed in real time

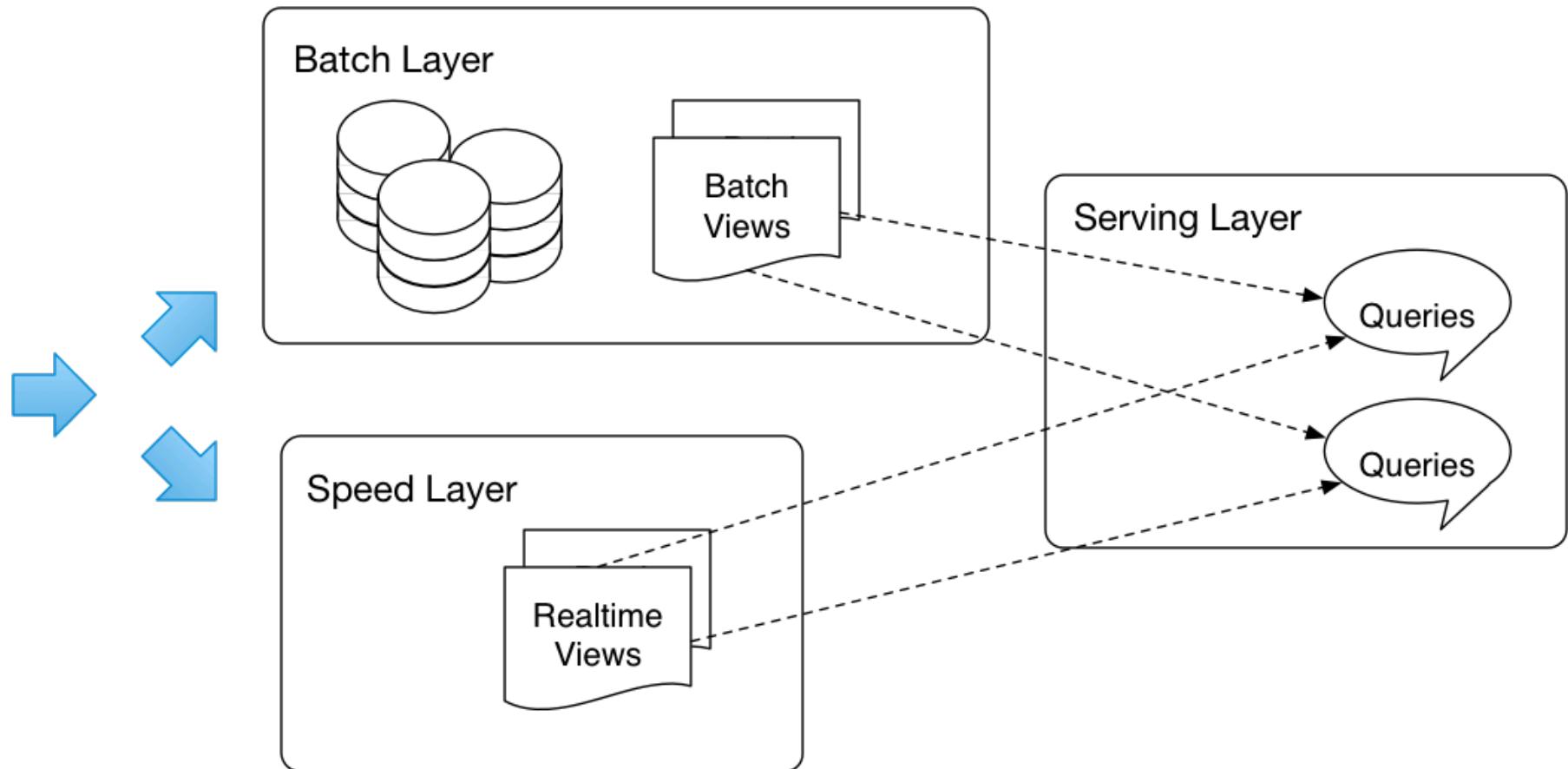


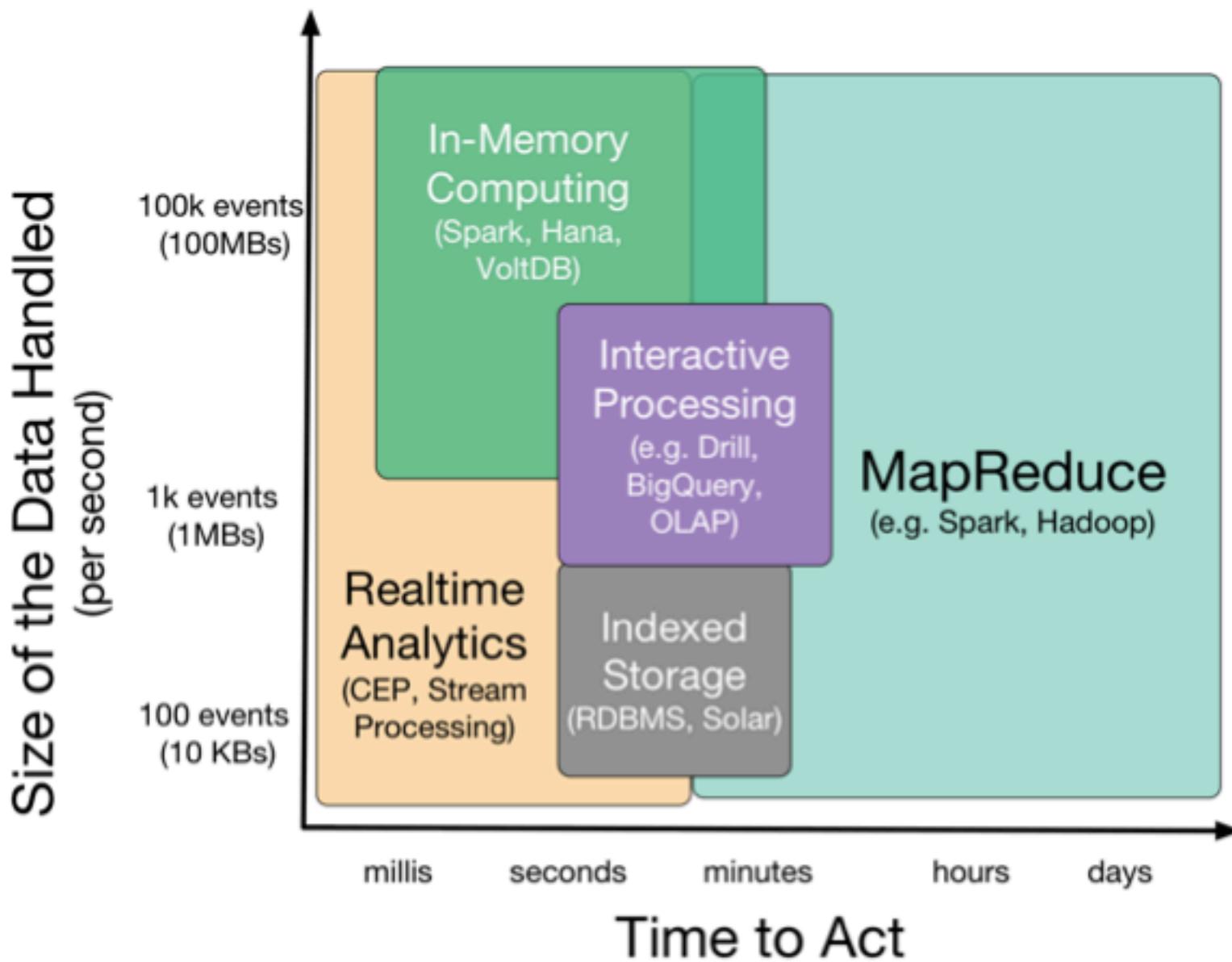
Comparing Databases with Real-Time systems

	Database Applications	Event-driven Applications
Query Paradigm	Ad-hoc queries or requests	Continuous standing queries
Latency	Seconds, hours, days	Milliseconds or less
Data Rate	Hundreds of events/sec	Tens of thousands of events/sec or more
		



Lambda Architecture





Approaches to Streaming

- Pure streaming
 - Each event is processed as it comes in
- Micro-batch
 - Small batches of events are processed
 - Typically trades flexibility for performance
- Shared nothing
 - You can process events on any system in the cluster
- Stateful / Partitioned
 - The event must be processed on a system that has the correct state in memory



Data distribution

- You need to get the events to the processing systems



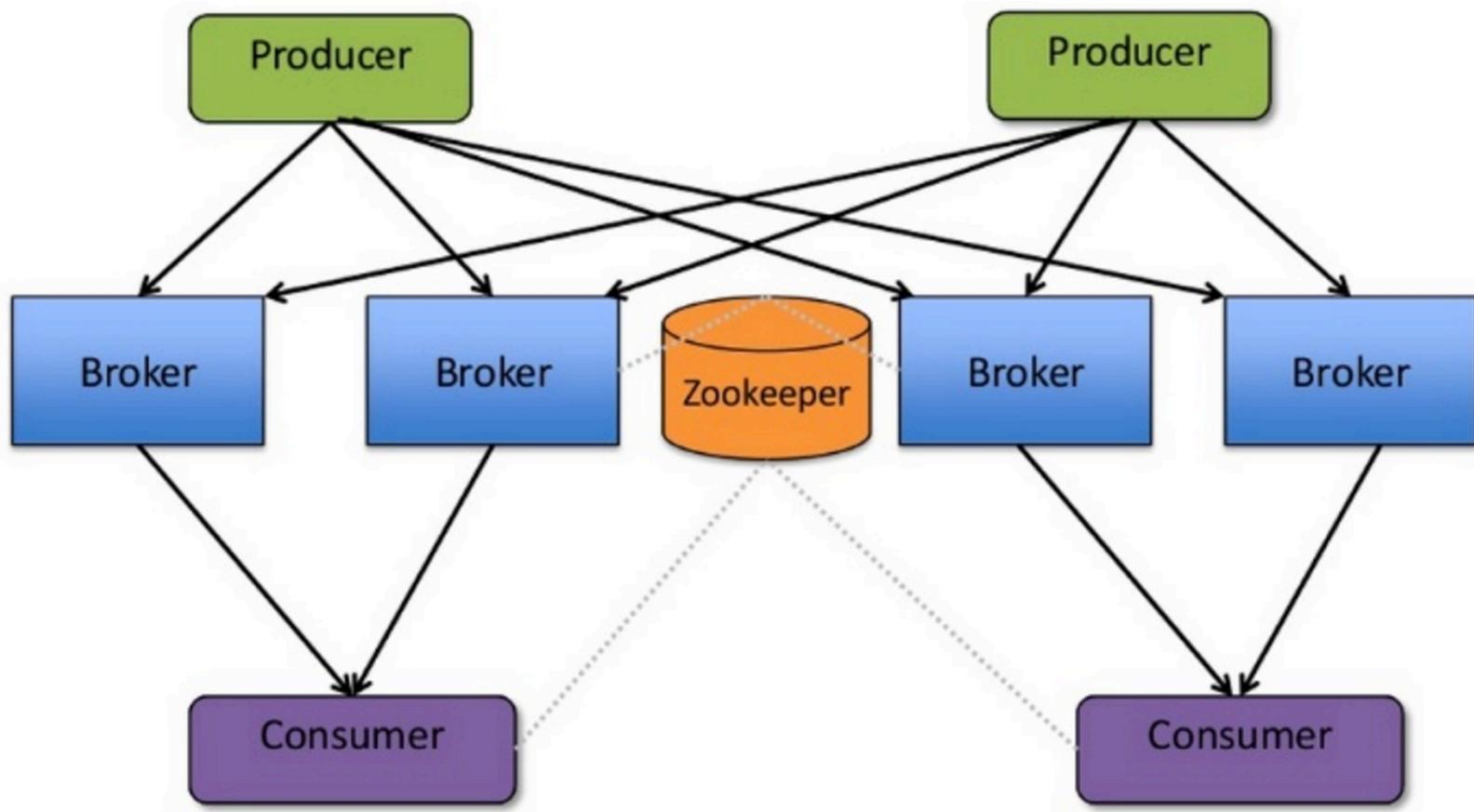
© Paul Fremantle 2015. This work is licensed under a Creative Commons
Attribution-NonCommercial-ShareAlike 4.0 International License
See <http://creativecommons.org/licenses/by-nc-sa/4.0/>

MQTT

- Very simple, lightweight, fast
- No built in support for clustering / big-data
 - But can make up for it by being very fast
- Used a lot in IoT



Apache Kafka



© Paul Fremantle 2015. This work is licensed under a Creative Commons
Attribution-NonCommercial-ShareAlike 4.0 International License
See <http://creativecommons.org/licenses/by-nc-sa/4.0/>

Source: <http://www.slideshare.net/charmalloc/>

Kafka

- Many of the approaches we've seen:
 - Partitioning
 - Multiple brokers
 - Elastically scalable
 - Supports clusters of co-ordinated consumers
 - Automatic re-election of leaders



Kafka exactly-once semantics



Mathias Verraes

@mathiasverraes

Follow

There are only two hard problems in distributed systems:
2. Exactly-once delivery 1.
Guaranteed order of messages
2. Exactly-once delivery

RETWEETS LIKES

6,775 **4,727**



10:40 AM - 14 Aug 2015

69

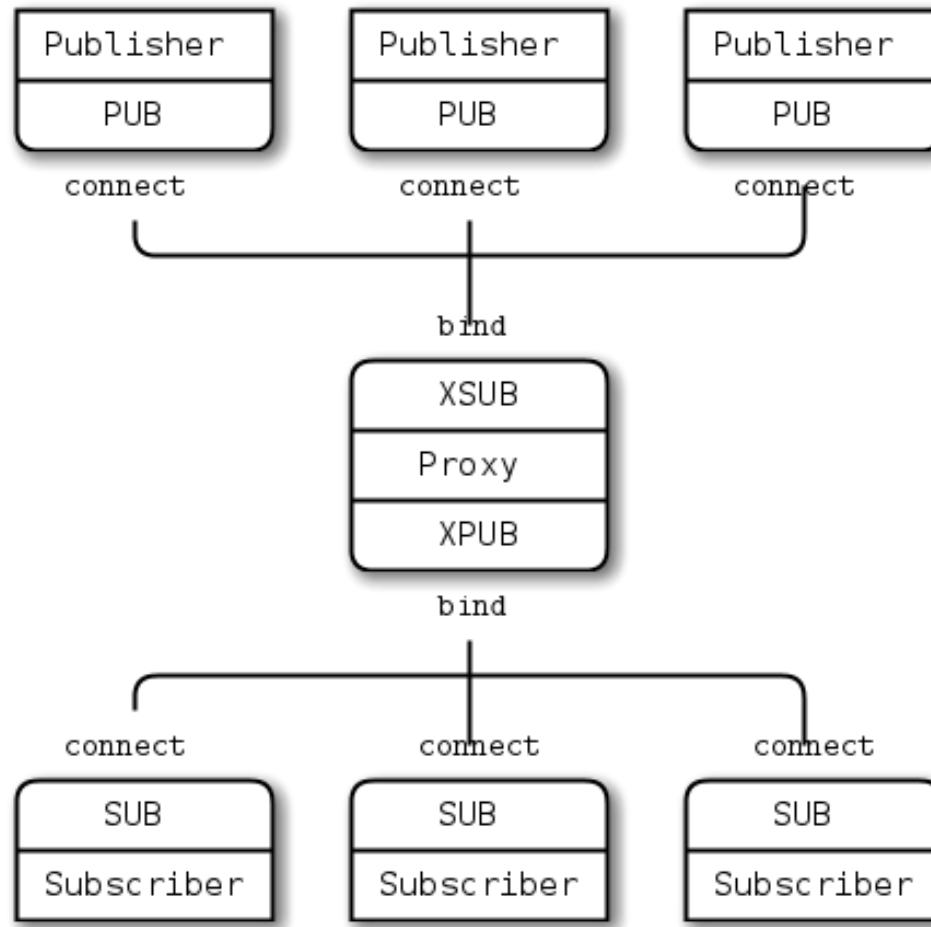
6.8K

4.7K



© Paul Fremantle 2015. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License
See <http://creativecommons.org/licenses/by-nc-sa/4.0/>

ZeroMQ

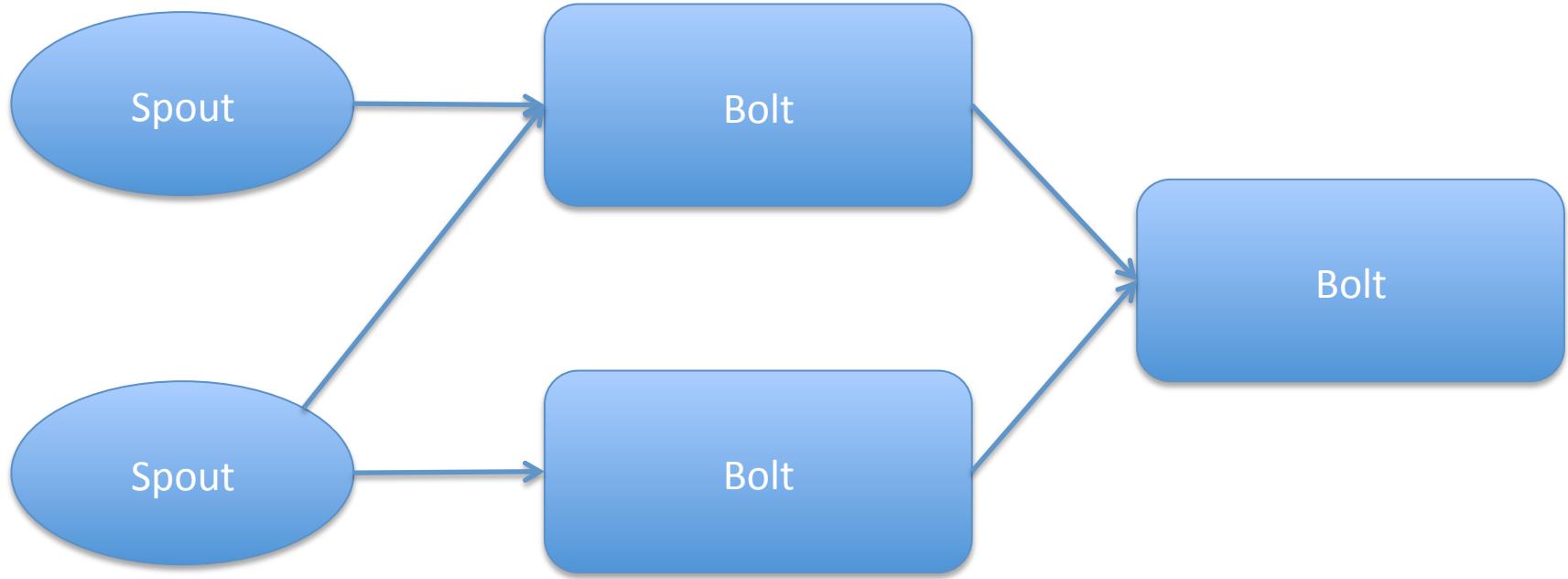


Processing the data



© Paul Fremantle 2015. This work is licensed under a Creative Commons
Attribution-NonCommercial-ShareAlike 4.0 International License
See <http://creativecommons.org/licenses/by-nc-sa/4.0/>

Apache Storm



Note: another DAG



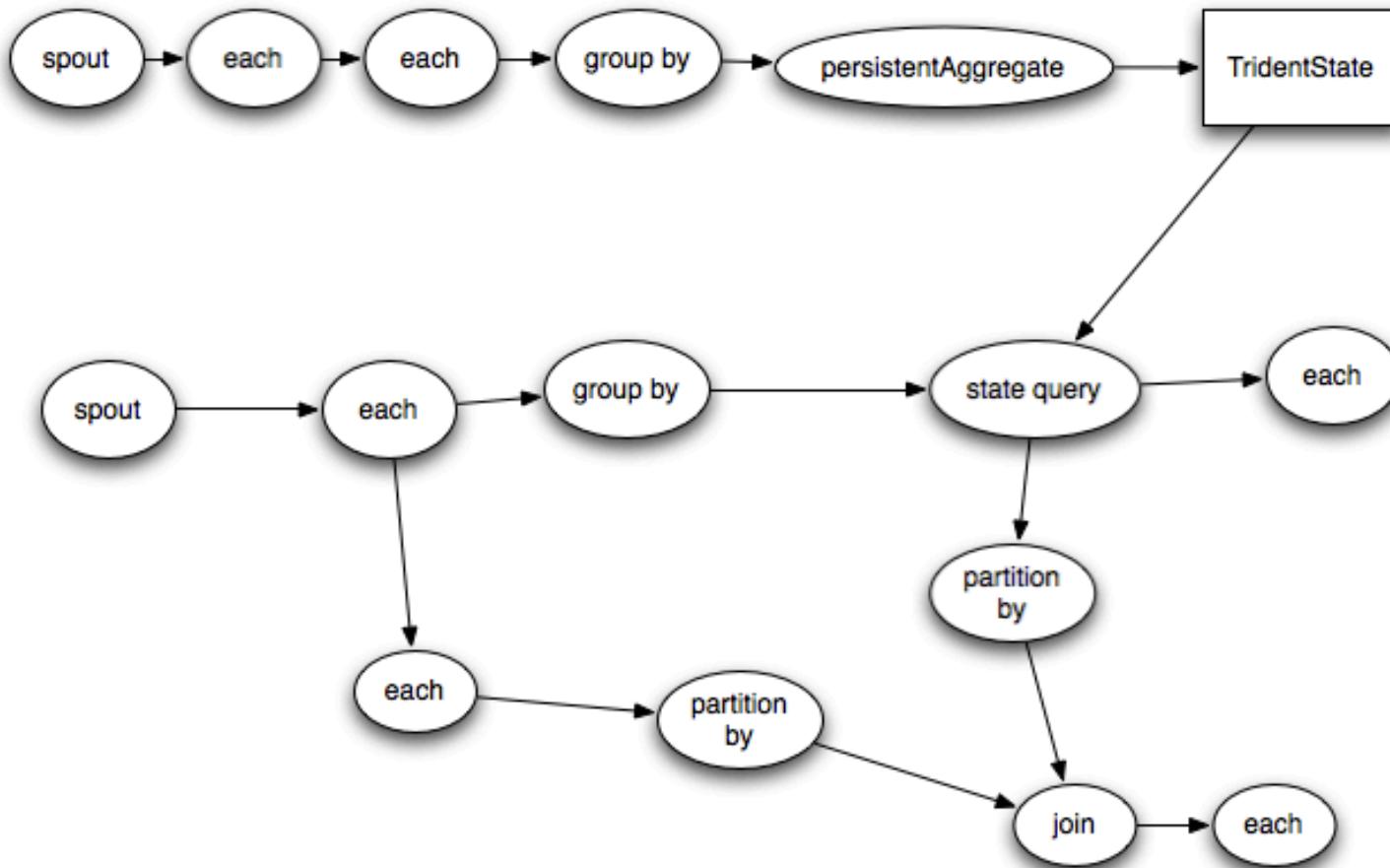
© Paul Fremantle 2015. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License
See <http://creativecommons.org/licenses/by-nc-sa/4.0/>

Apache Storm

- Originally developed by BackType
 - Nathan Marz
- Acquired by Twitter
- Open Sourced and then donated to Apache
- Became a top level project in 2014
 - <http://storm.apache.org>



Apache Storm Trident (micro-batch)

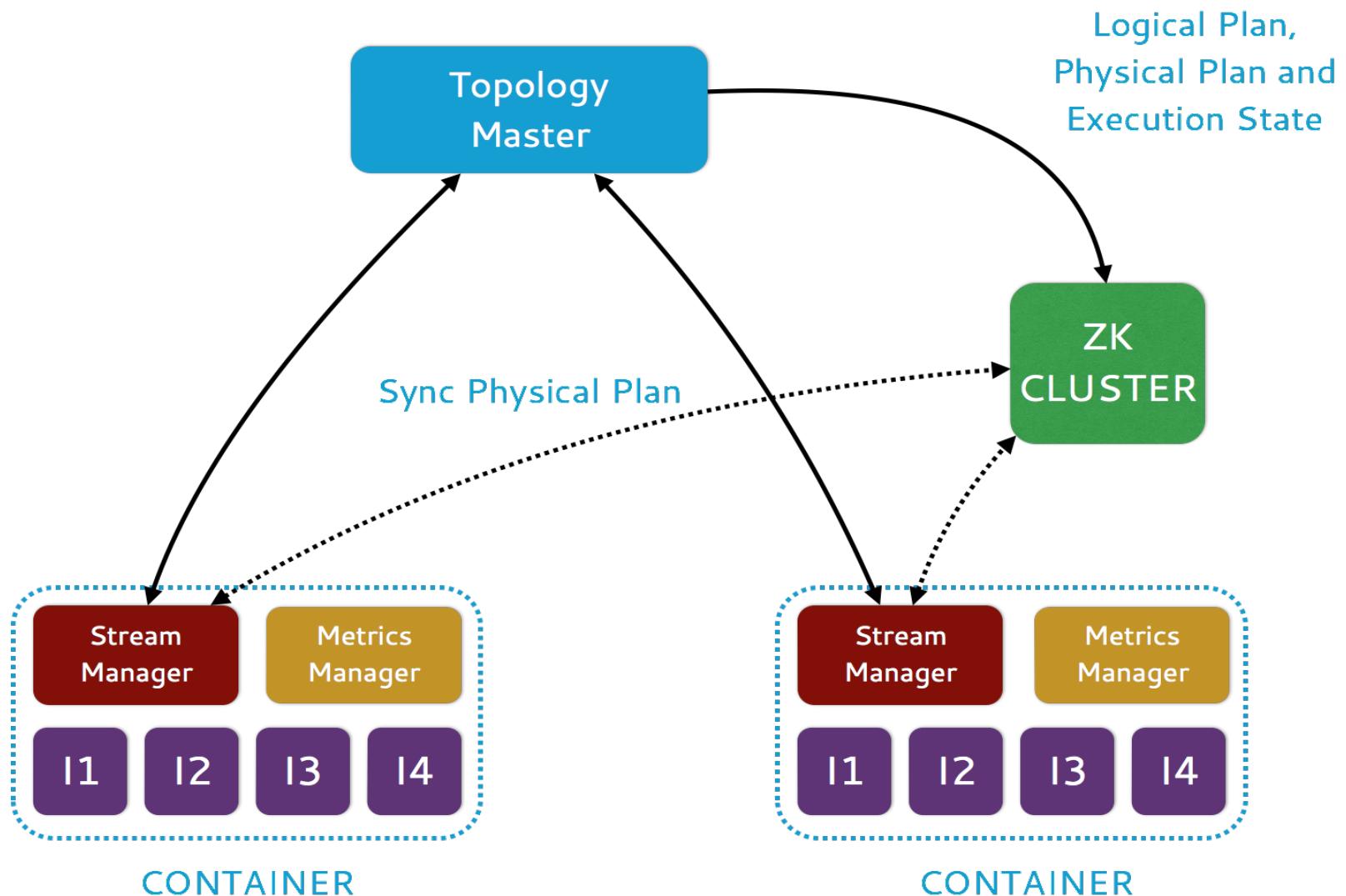


Storm vs Spark Streaming

- “Classic” Storm has no counterpart in Spark
 - Spouts and Bolts
 - Event by event processing
- Trident and Streaming both offer micro-batch models
 - More performant but less flexible
- Storm is more flexible for pure streaming systems
- Spark offers a much more unified programming model for Batch and Streaming



Heron



Heron: Key Features

- Fully API compatible with Apache Storm
- Task isolation
- Developer productivity
- Ease of manageability
- Use of mainstream languages C++/Java/Python



Heron

- In production at Twitter for >2 years
- Going into production at Microsoft, WeChat
- Donation to CNCF

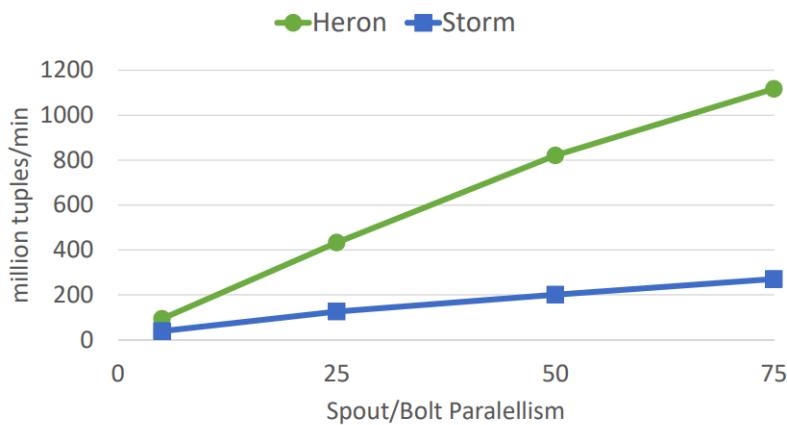


Fig. 2. Throughput with acks

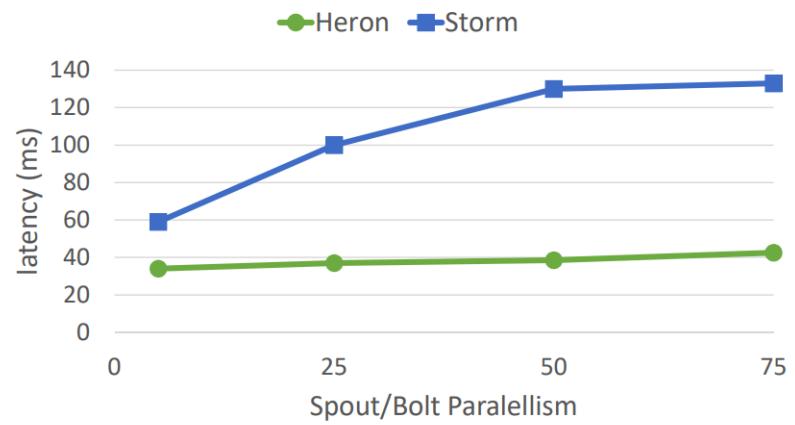
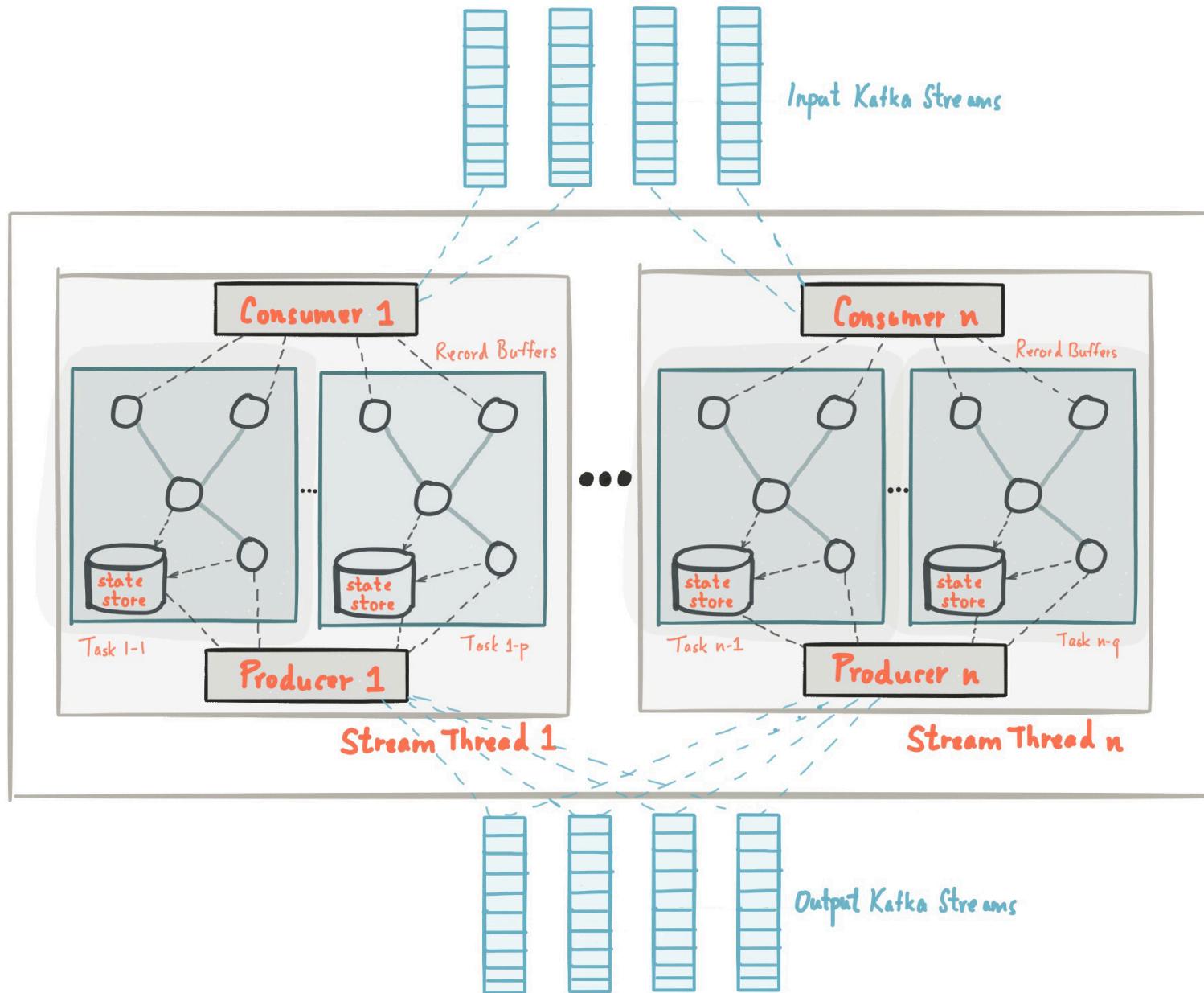


Fig. 3. End-to-end latency with acks



Kafka Streams



Kafka Streams

- Event-at-a-time processing (not microbatch) with millisecond latency
- Stateful processing including distributed joins and aggregations
- A convenient DSL
- Windowing with out-of-order data using a DataFlow-like model
- Distributed processing and fault-tolerance with fast failover
- Reprocessing capabilities so you can recalculate output when your code changes
- No-downtime rolling deployments

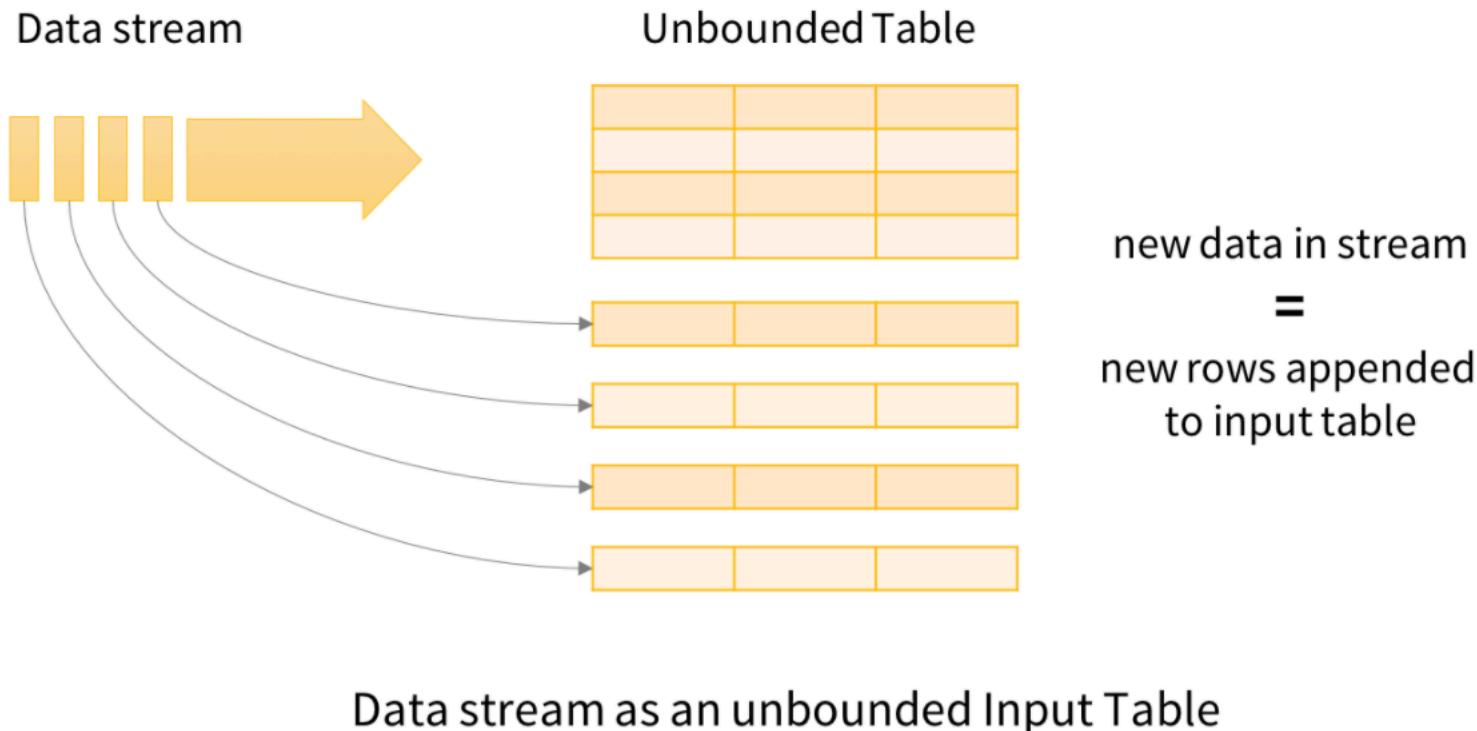


Apache Spark Streaming



Structured Streams in Spark

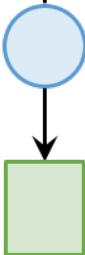
- Since Spark 2.0, there is a much better approach



Input Table



User Query



Result Table



Spark SQL Planner



System Time

Input Table



Incremental Query



Result Table



Output Update Mode

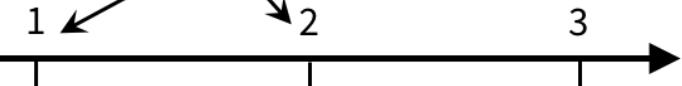


Triggers

1

2

3



data up
to t = 1

data up
to t = 2

data up
to t = 3

result up
to t = 1

result up
to t = 2

result up
to t = 3

rows
updated
at t = 2

rows
updated
at t = 3

User's batch-like
query on input table

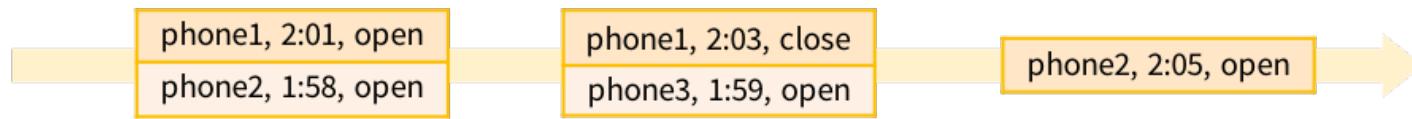
Incremental execution on streaming data

Structured Streaming Processing Model

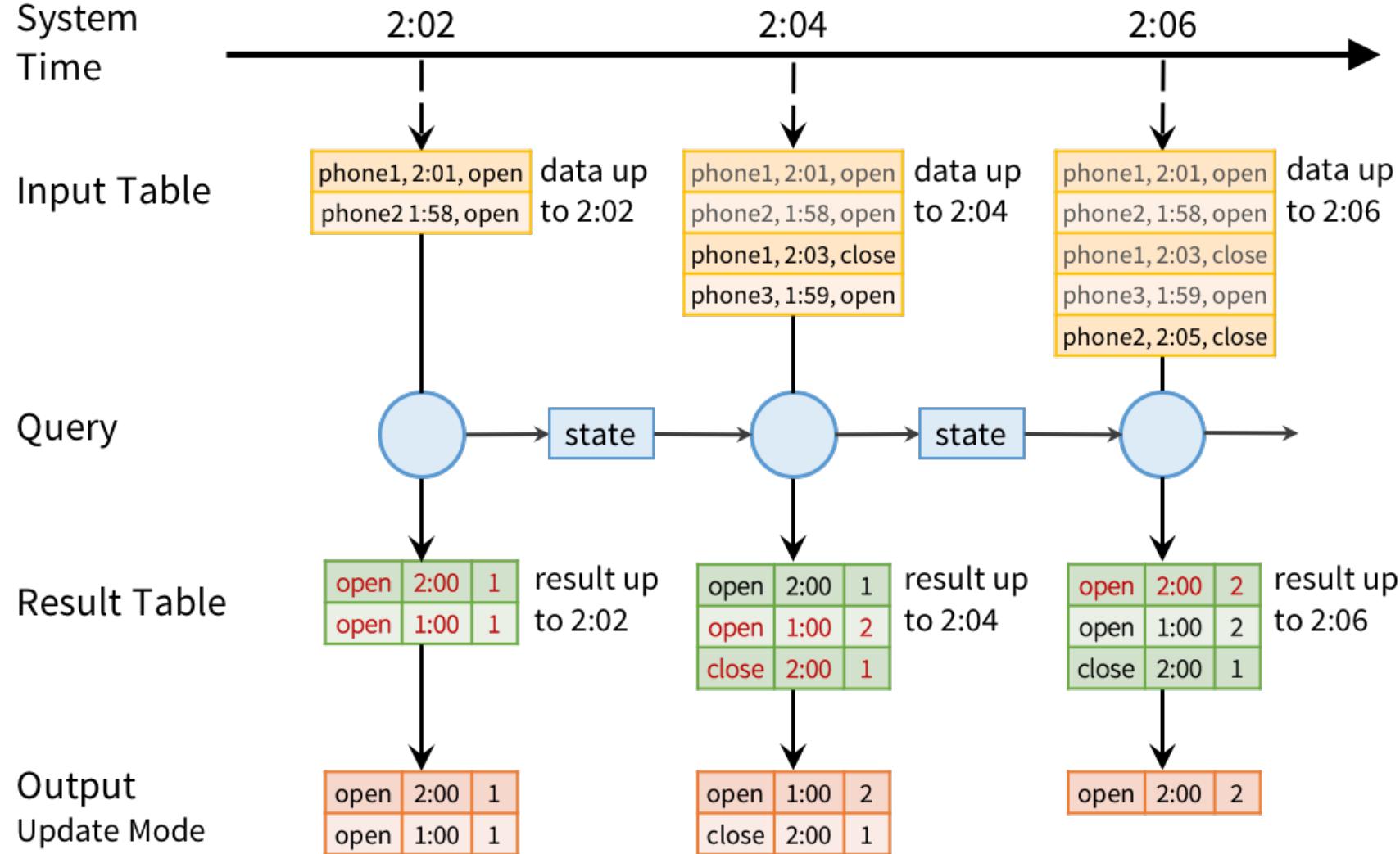
Users express queries using a batch API; Spark incrementalizes them to run on streams



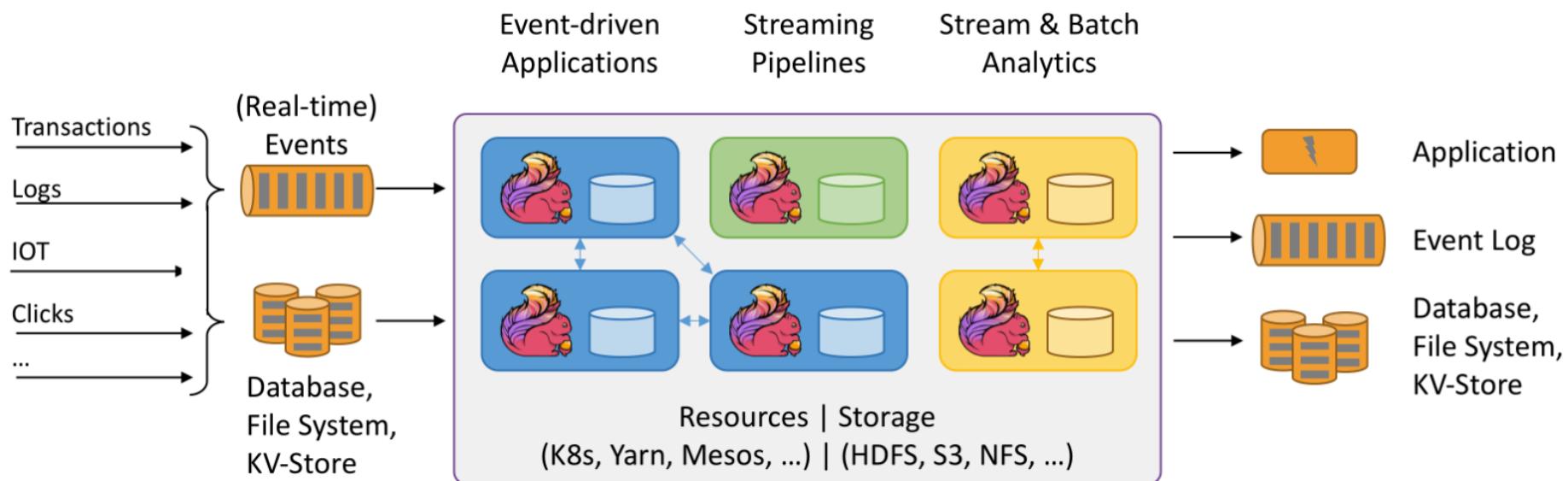
Arriving Records



System Time



Apache Flink



Siddhi

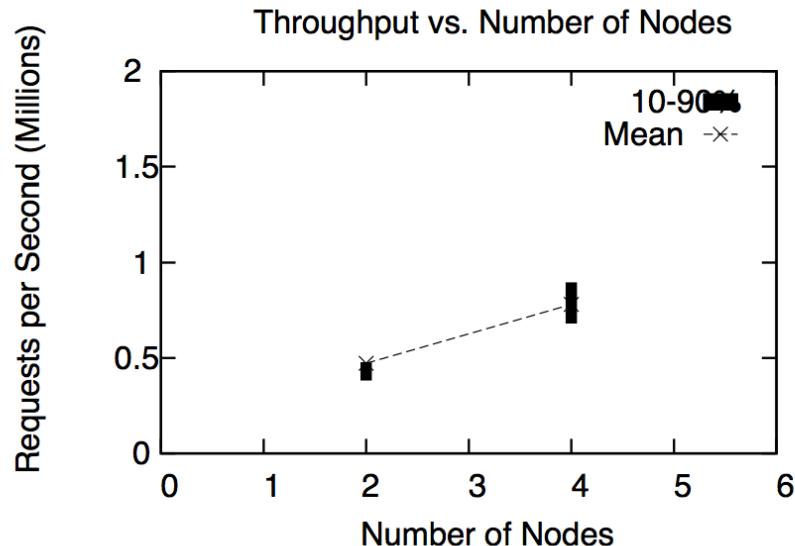
- A stateful query model
- SQL-like language for querying streams of data
 - Extended with **windows**
 - Time, Event count, batches
 - Partitioned
 - Based on data in the events
 - Pattern matching
 - A then B then C within window



© Paul Fremantle 2015. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License
See <http://creativecommons.org/licenses/by-nc-sa/4.0/>

Siddhi

- Apache Licensed Open Source on Github
 - <https://github.com/wso2/siddhi/>
- Pluggable into Storm, Spark and Kafka Streams
- Supports millions of events/sec
- [http://freo.me/DEBS Siddhi](http://freo.me/DEBS_Siddhi)

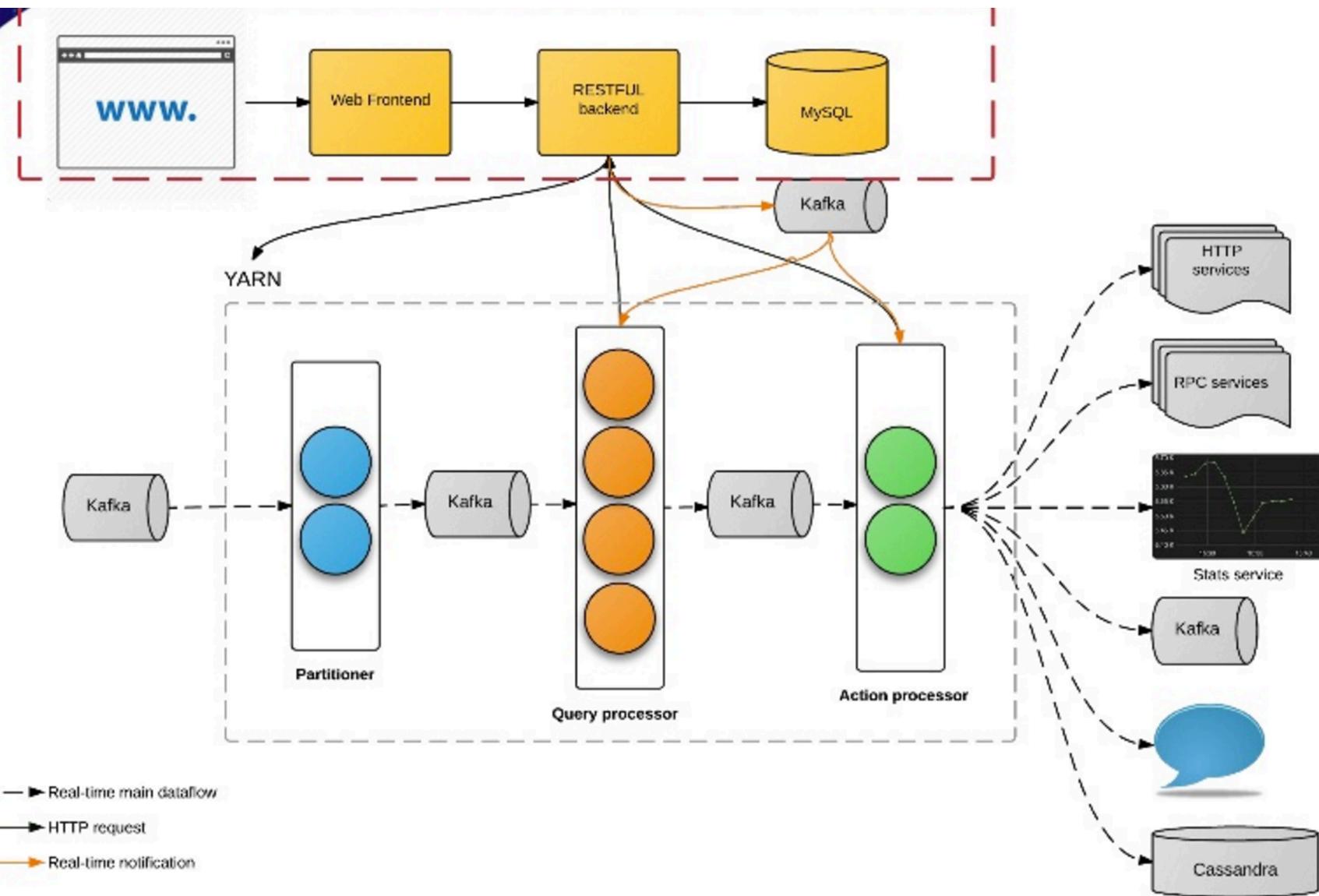


SiddhiQL

```
FROM login_stream#window.time(10 min)
SELECT ip,
       count(ip) as loginCount,
       cityId
GROUP BY ip
HAVING loginCount > 10
INSERT INTO login_attemp_repeatedly_stream;
```



Siddhi at Uber



Siddhi at Uber

- 100+ production apps
- 30 billion messages / day
- Fraud, anomaly detection
- Marketing, promotion
- Monitoring, feedback
- Real time analytics and visualization

<https://freo.me/siddhi-uber>



© Paul Fremantle 2015. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License
See <http://creativecommons.org/licenses/by-nc-sa/4.0/>

Long-running aggregation

```
define aggregation SweetProductionAggregation
from SweetProductionStream
select name, sum(amount) as totalAmount
group by name
aggregate every hour...year
```

```
from GetTotalSweetProductionStream as b join SweetProductionAggregation as a
on a.name == b.name
within b.duration
per b.interval
select a.AGG_TIMESTAMP, a.name, a.totalAmount
insert into HourlyProductionStream;
```



Streaming in Ballerina

```
function initRealtimeRequestCounter() {  
  
    stream<RequestCount> requestCountStream;  
  
    // Whenever the `requestCountStream` stream receives an event from the streaming rules defined in the `forever` block,  
    // the `printRequestCount` function is invoked.  
    requestCountStream.subscribe(printRequestCount);  
  
    // Gather all the events coming in to the `requestStream` for five seconds, group them by the host, count the number  
    // of requests per host, and check if the count is more than six. If yes, publish the output (host and the count) to  
    // the `requestCountStream` stream as an alert. This `forever` block is executed once, when initializing the service.  
    // The processing happens asynchronously each time the `requestStream` receives an event.  
    forever {  
        from requestStream  
        window timeBatch(10000)  
        select host, count(host) as count  
        group by host  
        having count > 6  
        => (RequestCount[] counts) {  
            // `counts` is the output of the streaming rules and is published to the `requestCountStream`.  
            // The `select` clause should match the structure of the `RequestCount` record.  
            requestCountStream.publish(counts);  
        }  
    }  
}
```

Summary

- **Realtime processing is hard**
 - Requires large memory and state
 - The lambda architecture splits the problem into batch and realtime challenges
- **Multiple approaches:**
 - Pure Streaming
 - Micro-batch
 - CEP



Questions?



© Paul Fremantle 2015. This work is licensed under a Creative Commons
Attribution-NonCommercial-ShareAlike 4.0 International License
See <http://creativecommons.org/licenses/by-nc-sa/4.0/>