Imperial College London

Department of Computing

Data Stream Processing in the Cloud

Evangelia Kalyvianaki
ekalyv@imperial.ac.uk

joint work with Raul Castro Fernandez, Marco Fiscato, Matteo Migliavacca and Peter Pietzuch

Large-Scale Distributed Systems Group
http://lsds.doc.ic.ac.uk

Doctoral School Day in Cloud Computing, Université catholique de Louvain – November 2012

The Data Deluge

Big data

- 150 Exabytes (billion GBs) in 2005 → 1200 Exabytes in 2010
- real-time big data analytics in UK £25 billions → £216 billions in 2012-17

Many new sources of data become available

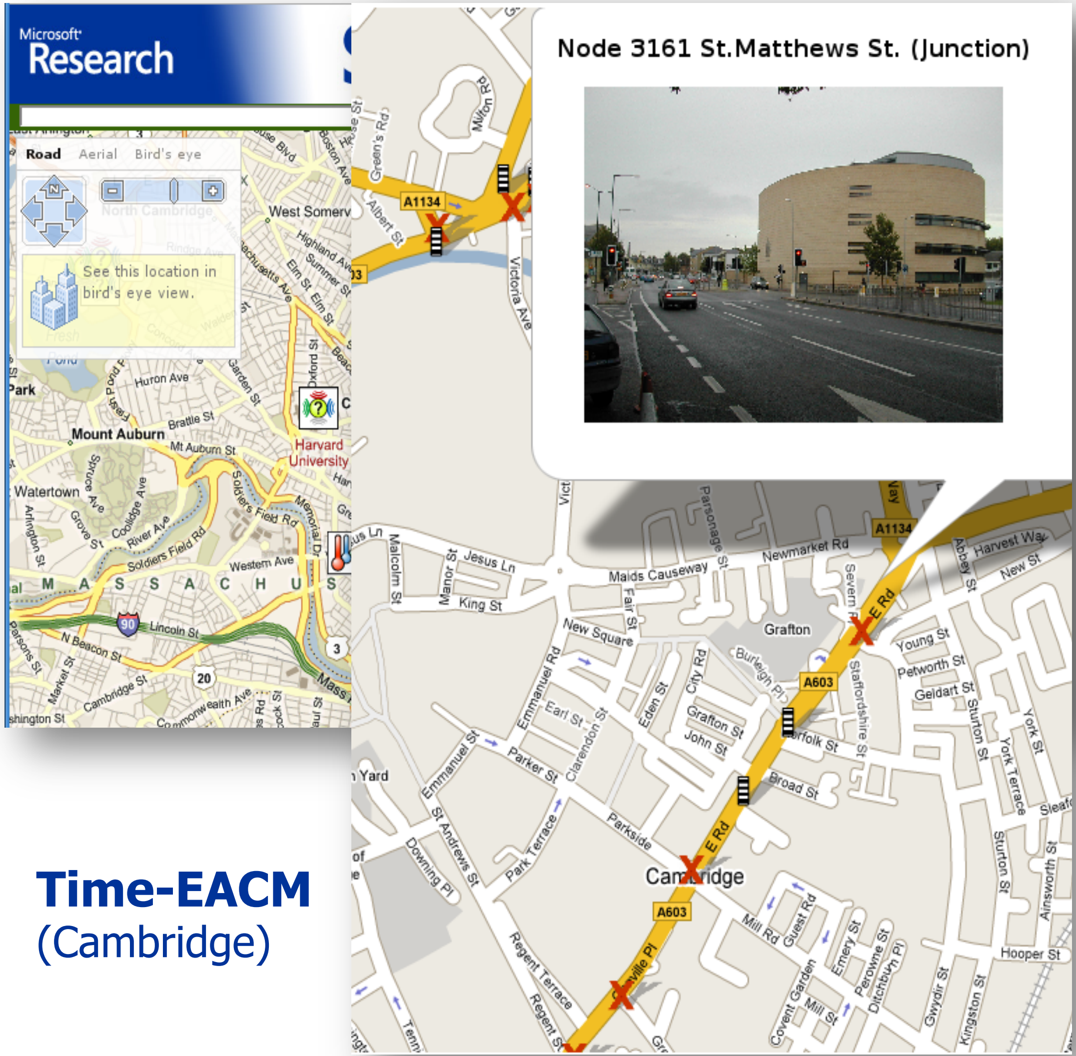
- Sensors, mobile devices
- Web feeds, social networking
- Cameras
- Scientific instruments

☛ **How can we make sense of all data ?**

- Most data is not interesting
- New data supersedes old data
- Challenge is not only storage but also querying

Real Time Traffic Monitoring

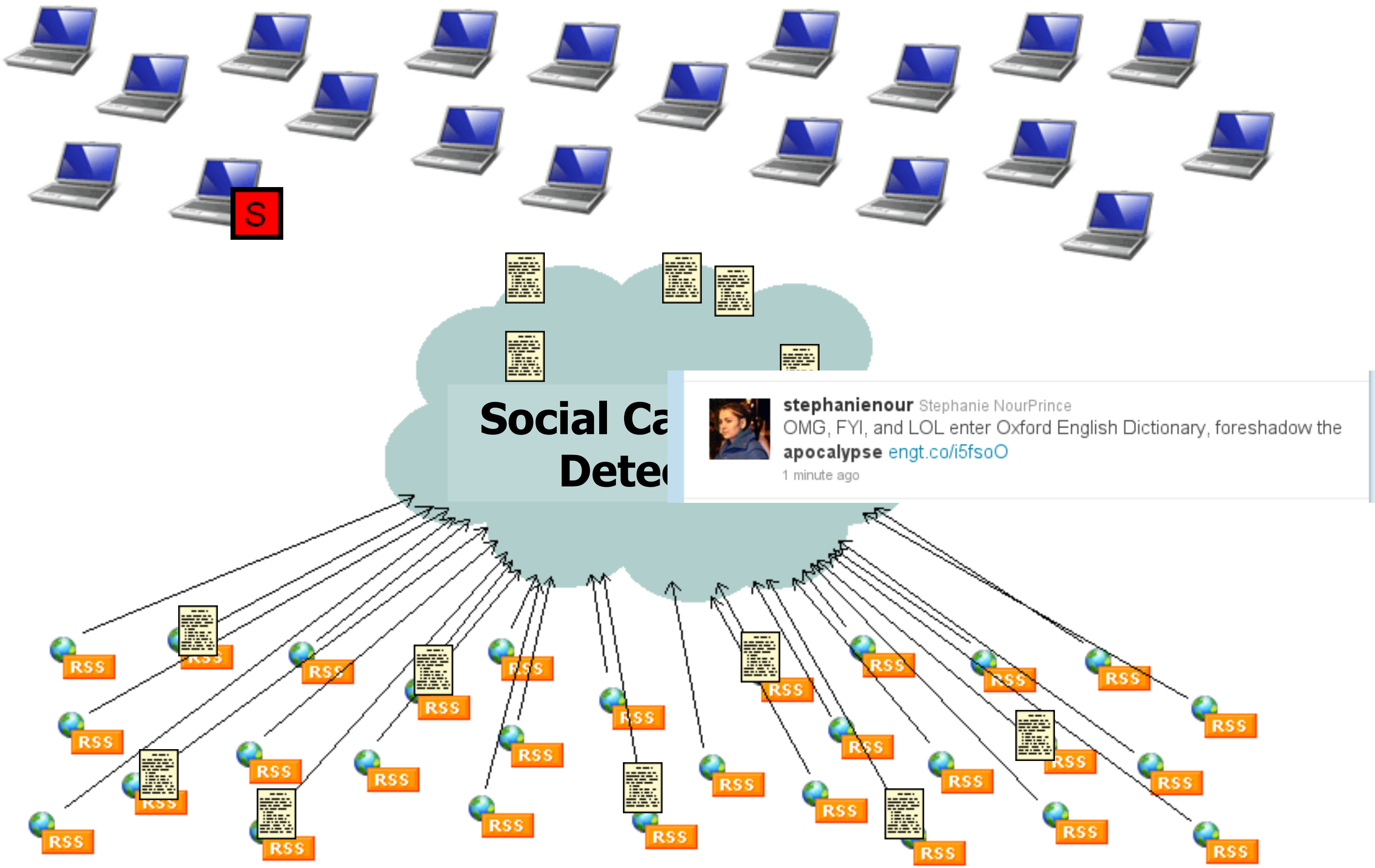
Instrumenting country's transportation infrastructure



Time-EACM
(Cambridge)

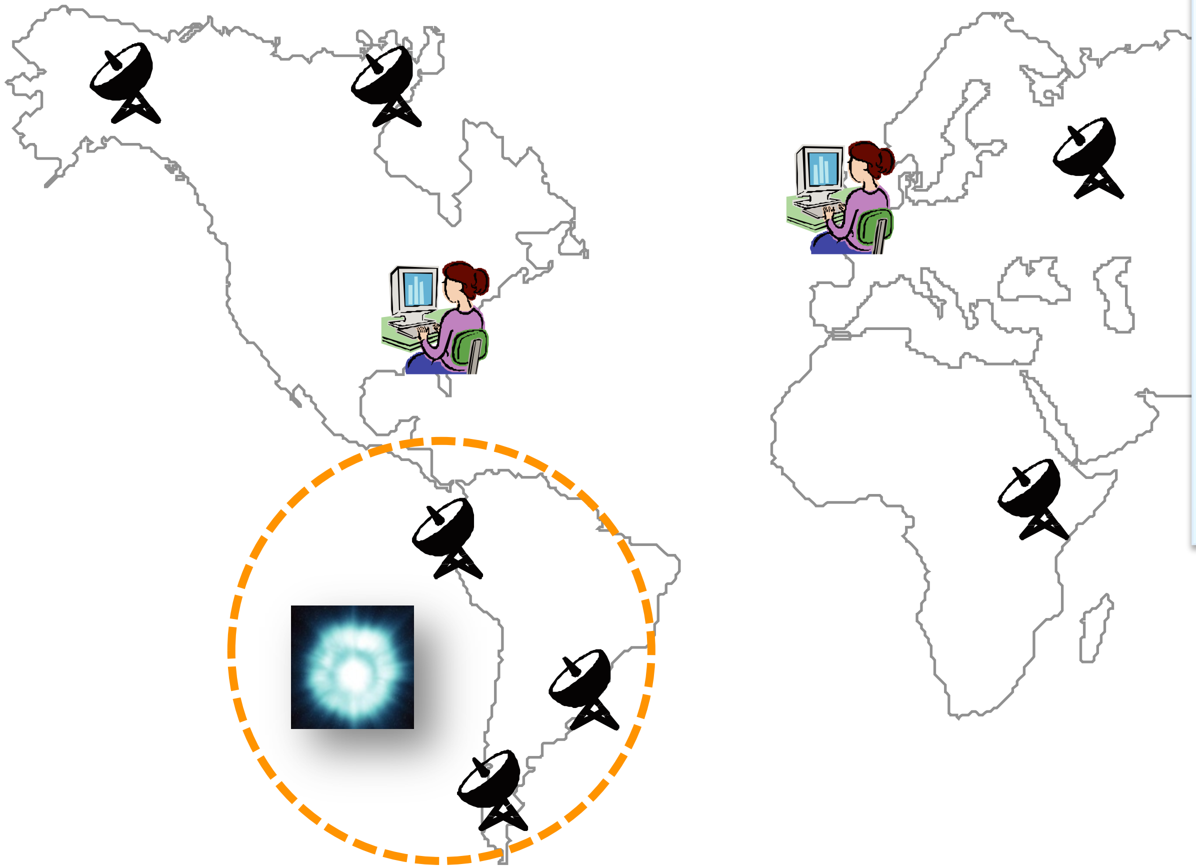
– Many parties interested in data
  - Road authorities
  - Traffic planners
  - Commuters
– High-level queries
  - "*What is the best time/route for my commute from London to Cambridge at 7-8am?*"

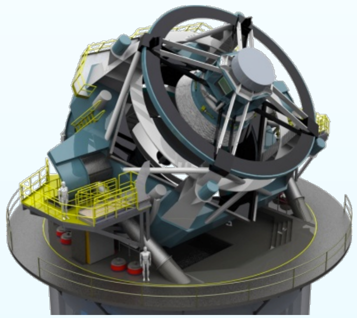
Web/Social Feed Mining



Detection and reaction to social cascades

Astronomic Data Processing





Large Synoptic Survey Telescope (LSST)

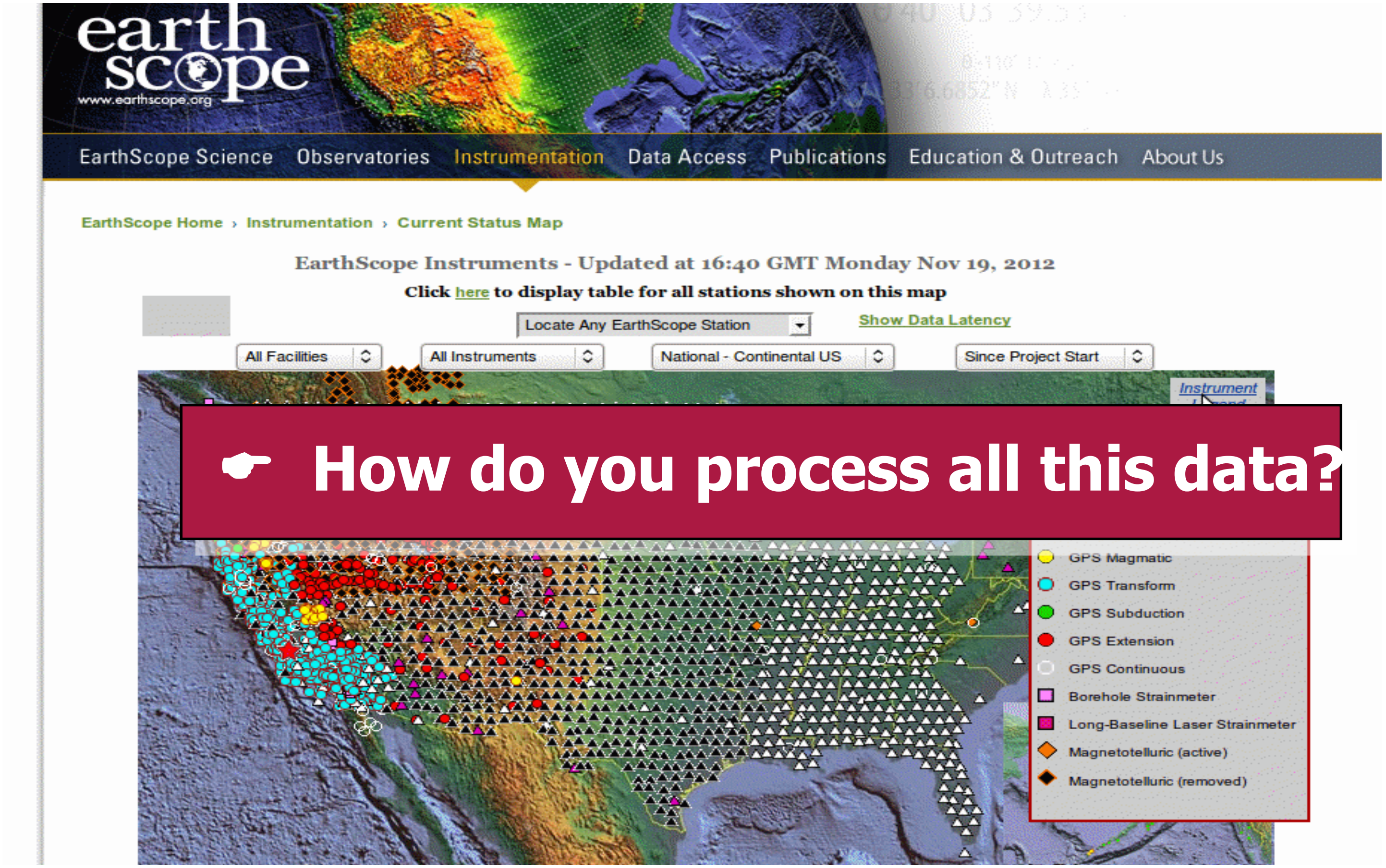
- Generates 1.28 Petabytes per year

Analysing transient cosmic events: γ-ray bursts

Global Sensor Applications: EarthScope

Using sensors to understand geological evolution

- Many sources: seismometers, GPS stations, …

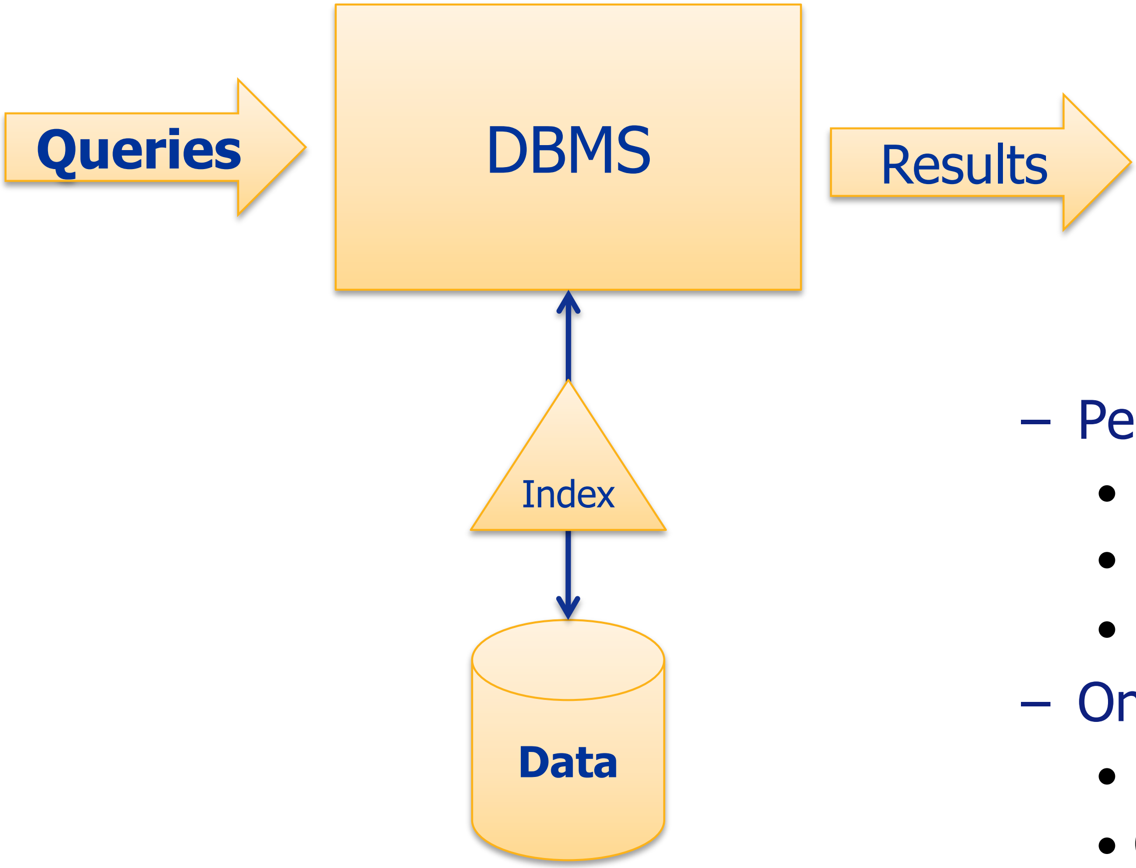


☛ **How do you process all this data?**

Traditional Databases

Database Management System (DBMS):

- Data relatively static but queries dynamic

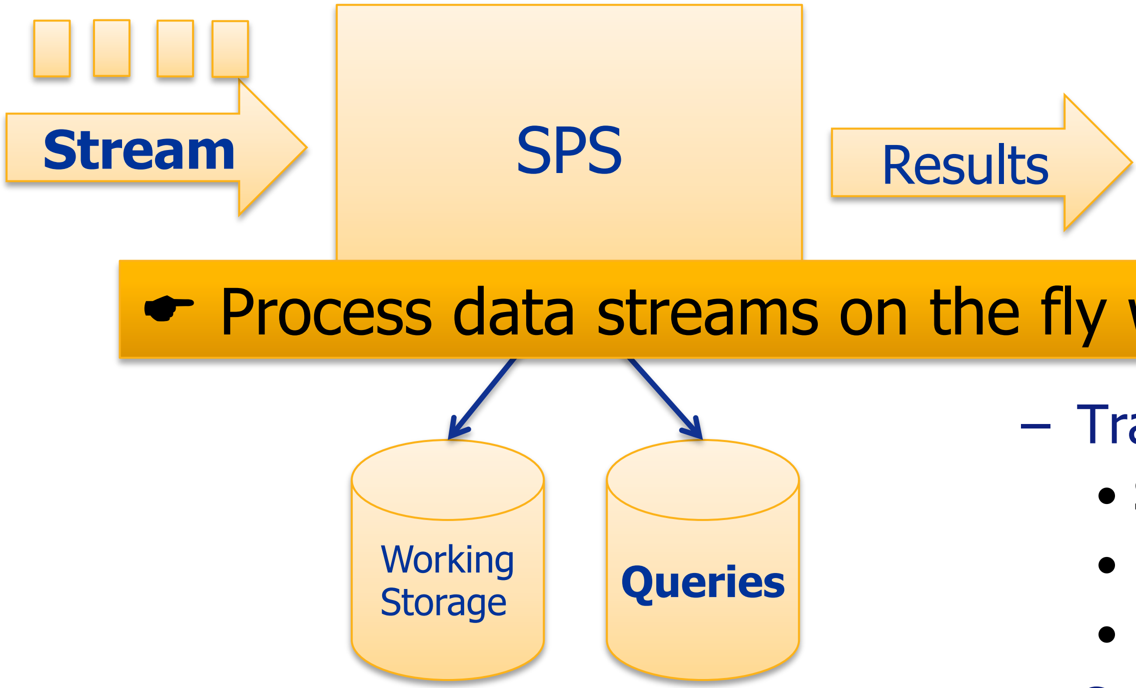


- Persistent relations
  - Radom access
  - Low update rate
  - Unbounded disk storage
- One-time queries
  - Finite query result
  - Queries exploit (static) indices

Data Stream Processing System

SPSs: Queries static but data dynamic

- Data represented as time-dependant **data stream**

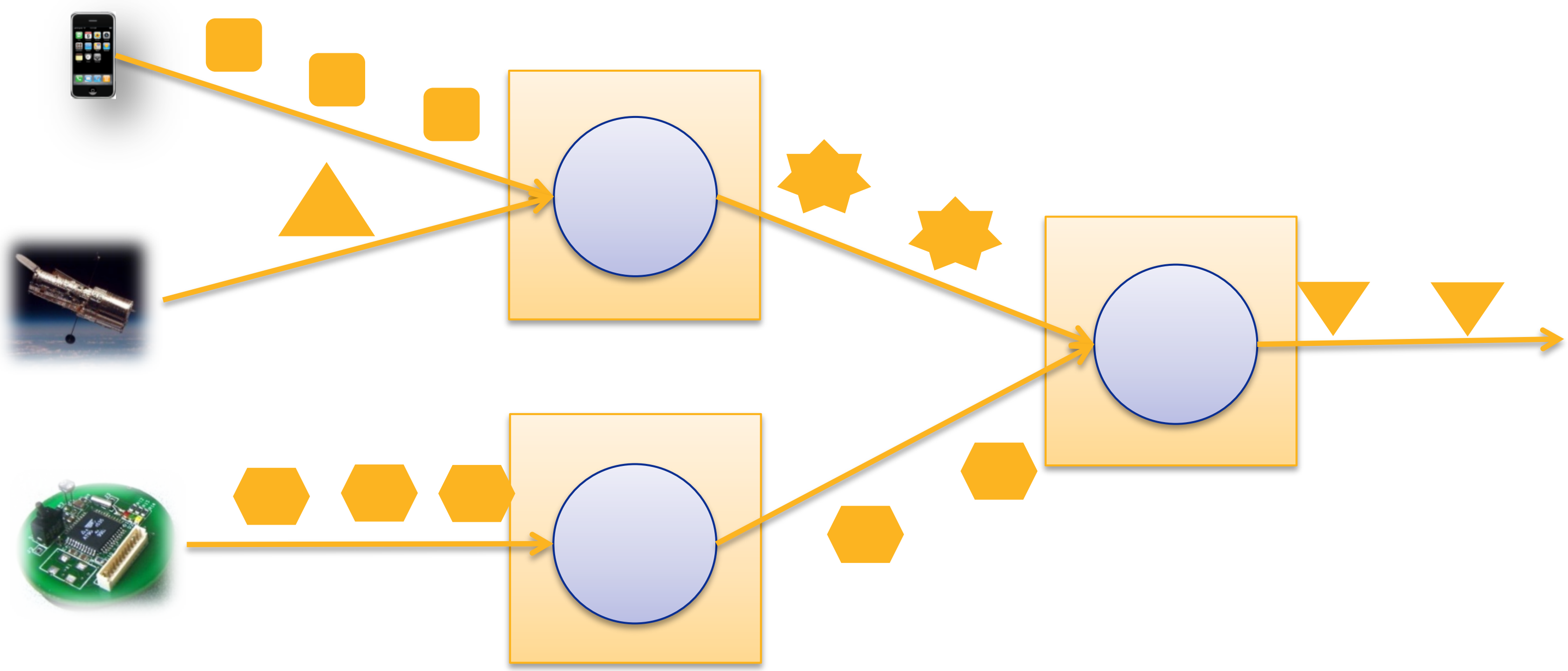


☛ Process data streams on the fly without storage

- Transient streams
  - Sequential access
  - Potentially high rate
  - Bounded main memory
- Continuous queries
  - Time-dependant res. stream
  - Indexing?

Data Stream Processing

☛ Process **tuple streams** on-the-fly by **operators**:



This talk is about ...

Data Stream Processing in the Cloud

Scalable and Fault-tolerance Stream Processing in the Cloud

- Increasing workload rates
- Stateful operators

Fair Stream Processing in Federated SPSs under Overload

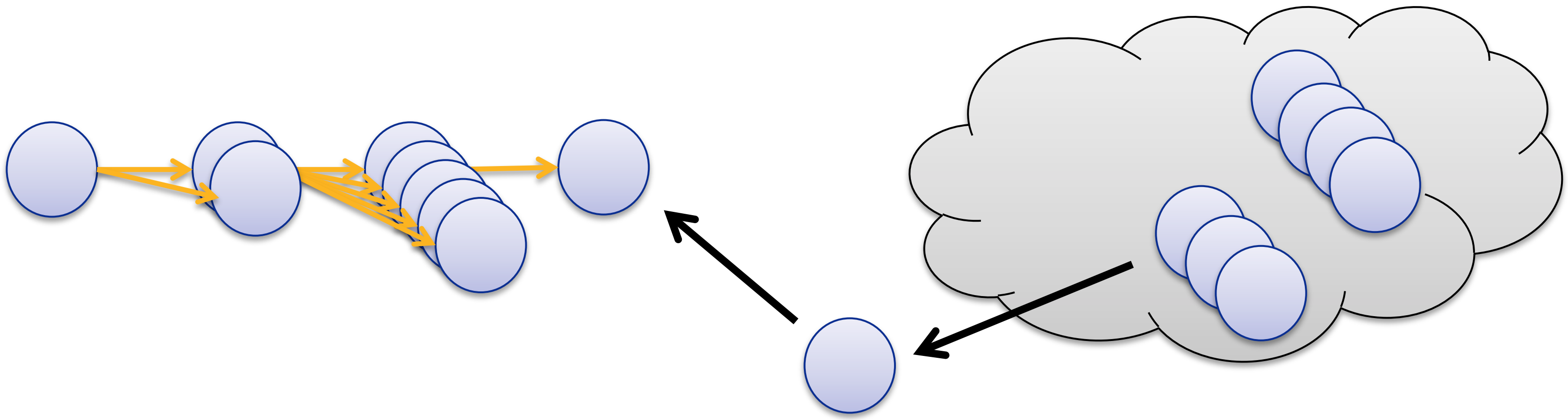
- Tuple shedding user-feedback metric
- Fair tuple shedding under overload

Scalable and Fault-tolerant Stream Processing in the Cloud

Stream Processing in the Cloud

Clouds provide virtually infinite pools of resources

- Fast and cheap access to new machines for operators



In a utility-based pricing model:

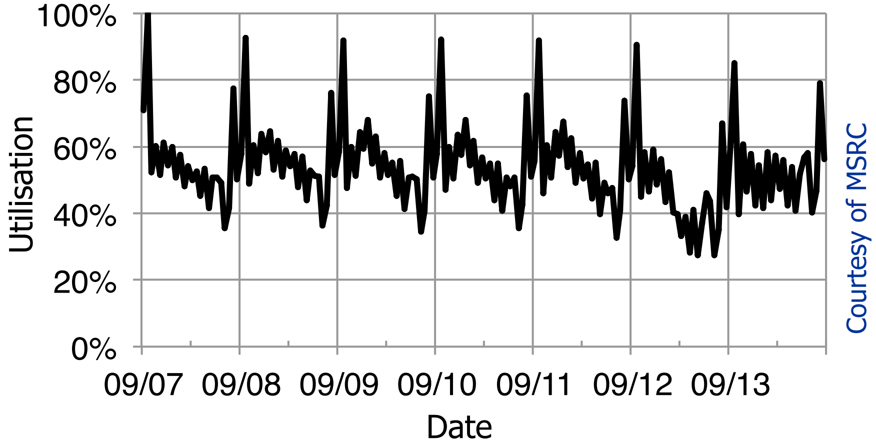
☛ How do you use the optimal number of resources?

- Needlessly over-provisioning system is expense
- Using too few resources leads to poor performance

Challenges in Cloud-Based Stream Processing

Intra-query parallelism

– Provisioning for workload peaks unnecessarily conservative



☛ **Dynamic scale out:** increase resources when peaks appear

Failure resilience

– Active fault-tolerance requires 2x resources
– Passive fault-tolerance leads to long recovery times

☛ **Hybrid fault-tolerance:** low resource overhead with fast recovery

☛ both mechanisms must support **stateful** operators

Operator State Management

Operator state:

- A summary of past tuples' processing, e.g. max result
- It cannot be lost, or stream results are affected

On **scale out**:

- Partition operator state correctly, maintaining consistency

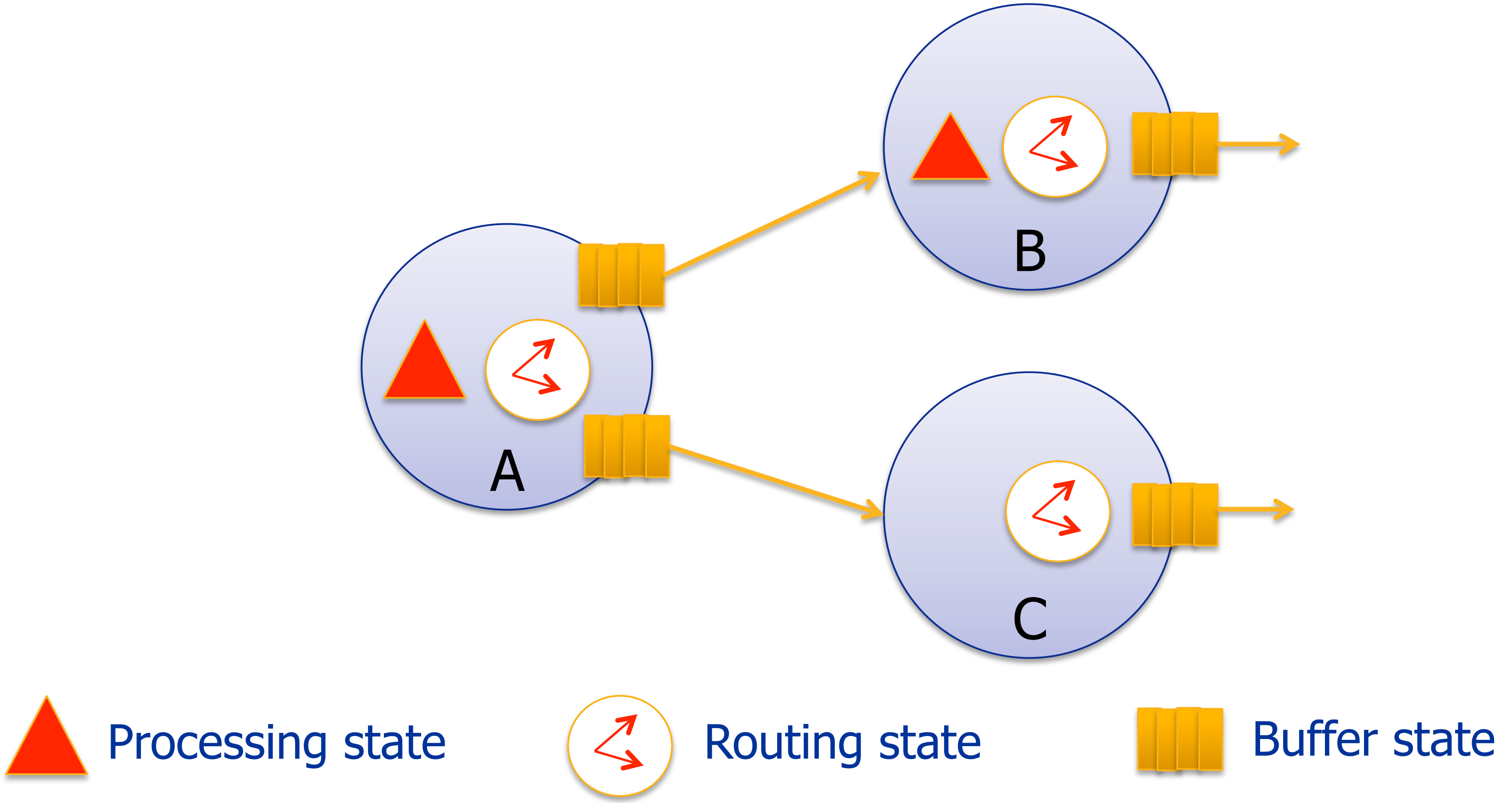
On **failure recovery**:

- Restore state of failed operator
- Define primitives for state management and build other mechanisms on top of them

☛ Make operator state an external entity that can be managed by the stream processing system

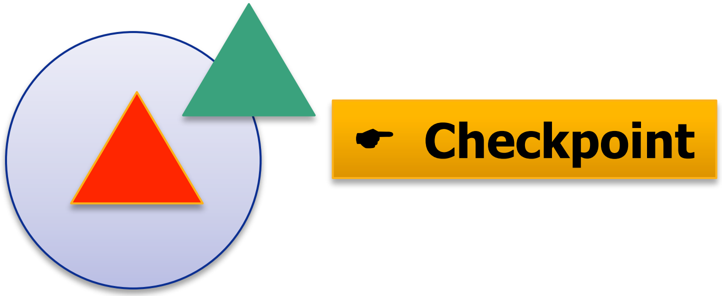
State Management

What is state in stream processing system?

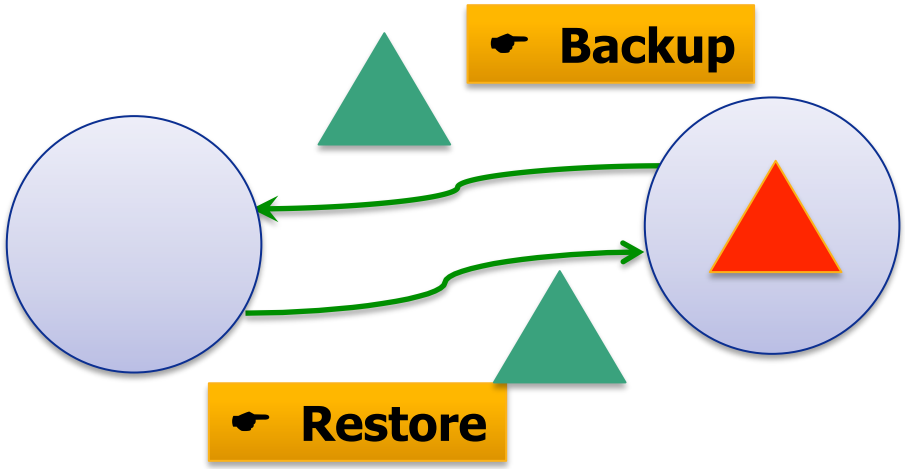


– Need to externalise processing state of operators

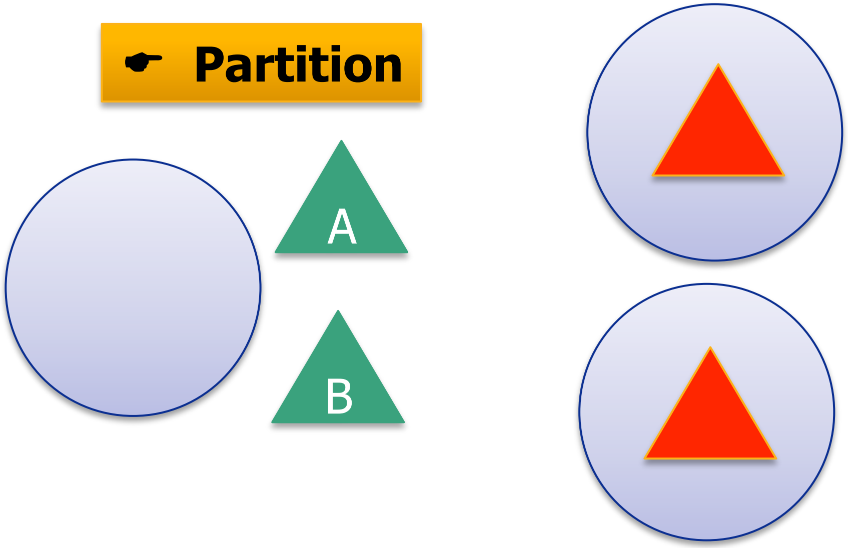
State Management Primitives



Takes snapshot of state and makes it externally available



Moves copy of state from one operator to another

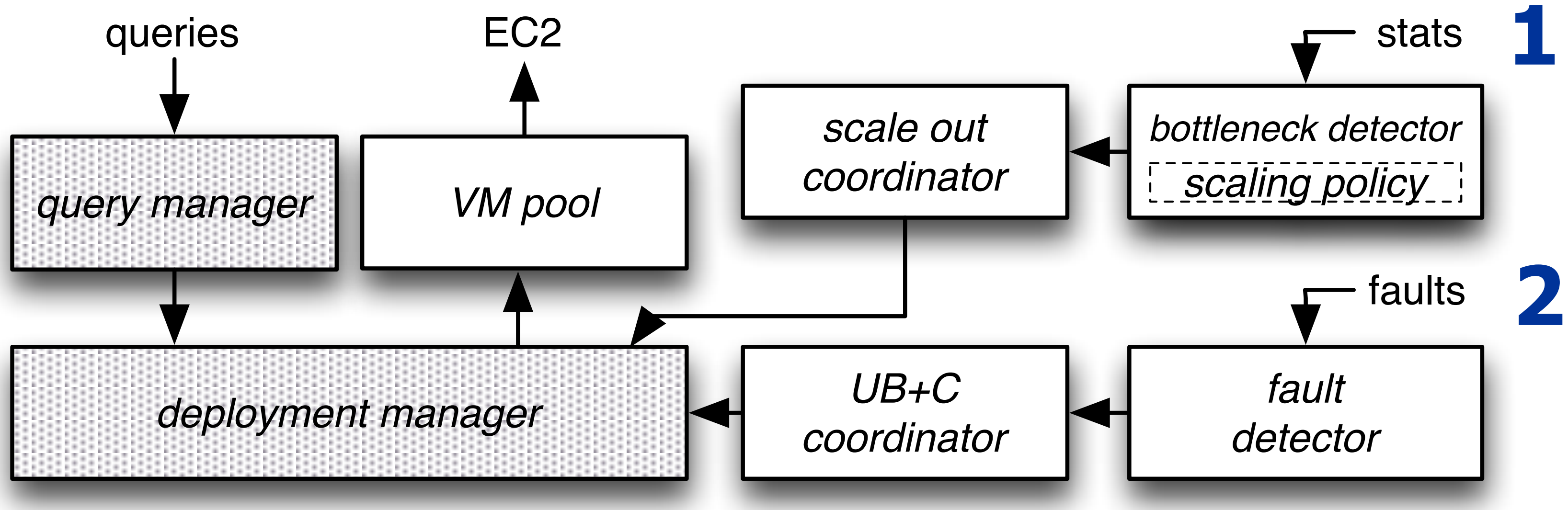


Splits state in a semantically correct fashion for parallel processing

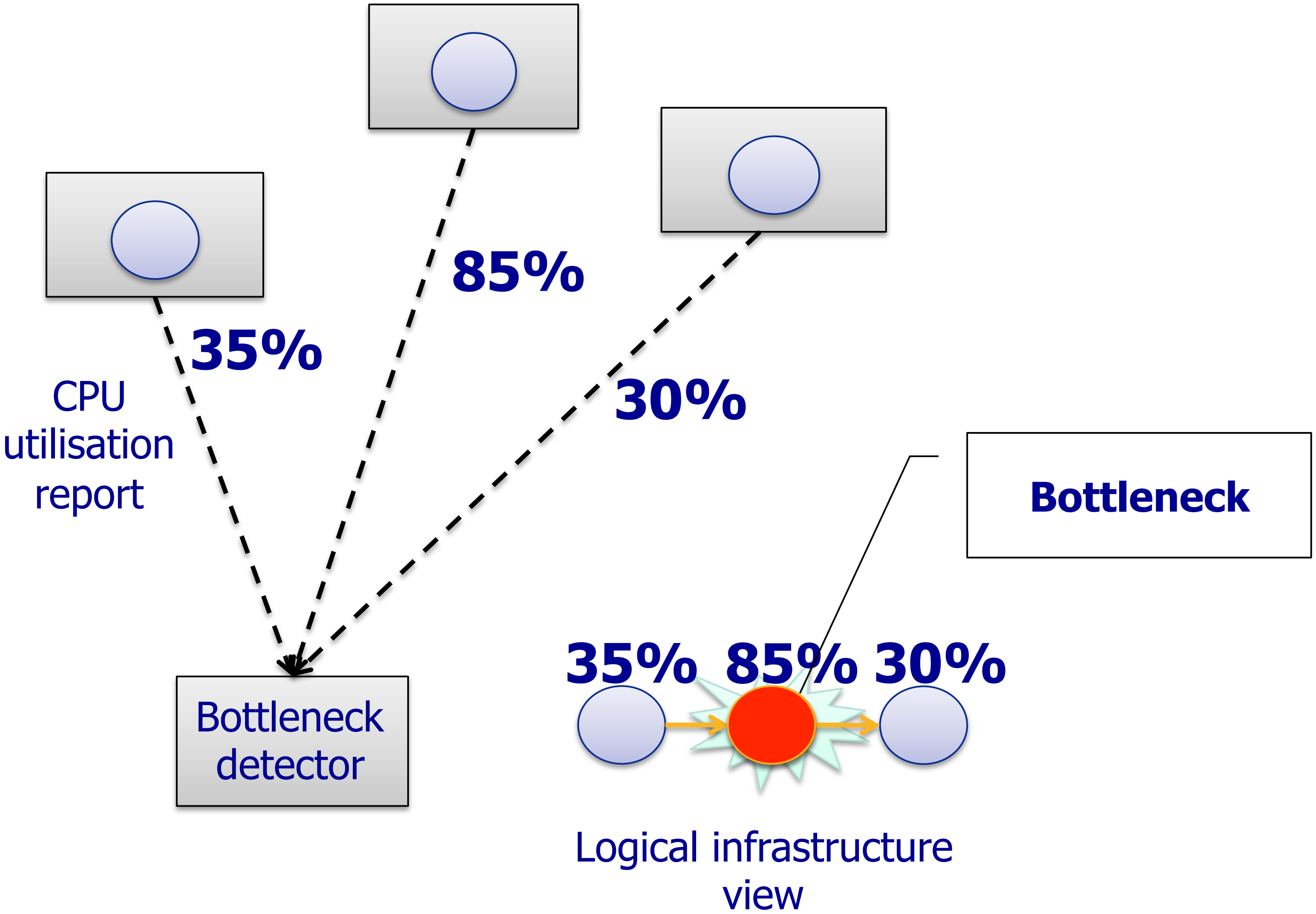
State Management in Action, SEEP

1. Dynamic Scale Out: Detect bottleneck, remove by adding new parallelised operator

2. Failure Recovery: Detect failure, replace with new operator

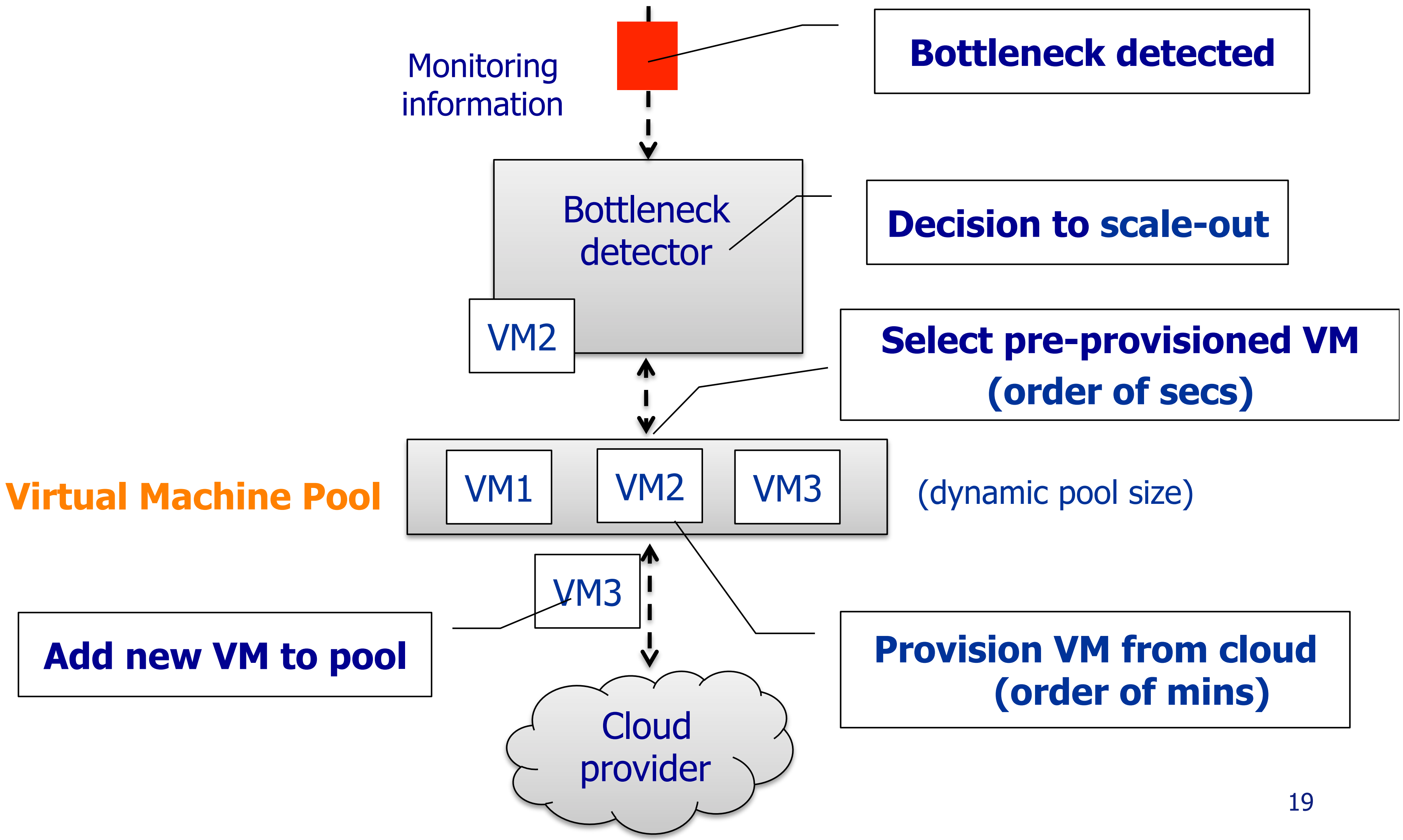


Dynamic Scale Out: Detecting bottlenecks



The VM Pool: Adding operators

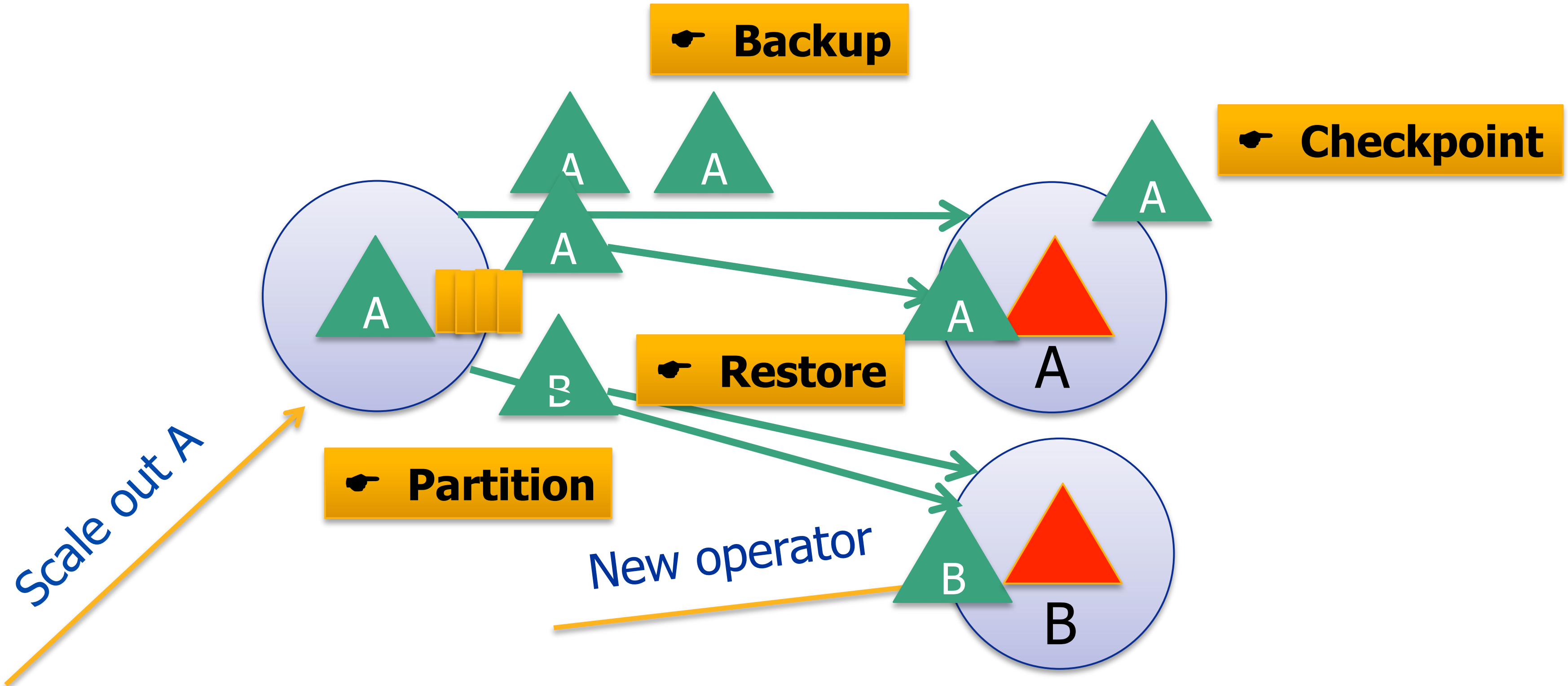
Problem: Allocating new VMs takes minutes...



Scaling Out Stateful Operators

Finally, upstream operators replay unprocessed tuples to update checkpointed state

Periodically, stateful operators checkpoint and back up state to designated **upstream backup node**



For scale out, backup node already has state of operator to be parallelised

State Partitioning

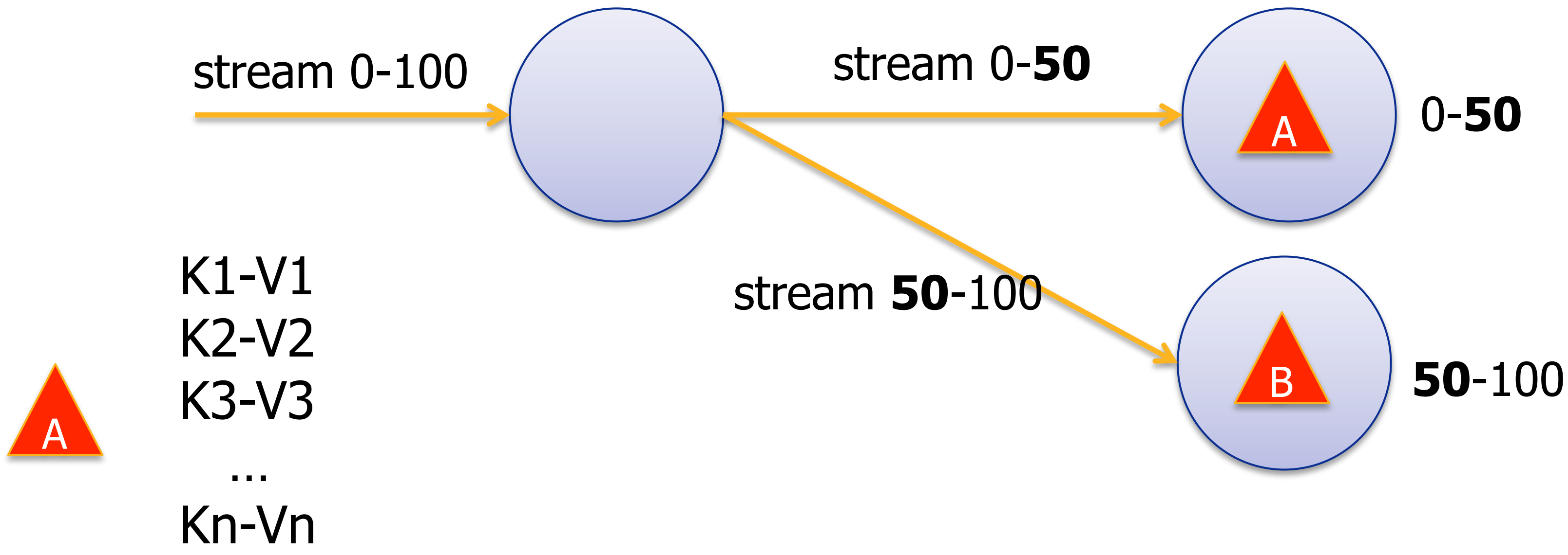
Processing state modeled as (key, value) dictionary

State partitioned according to key k of tuples

– Same key used to partition incoming streams

Tuples will be routed to correct operator

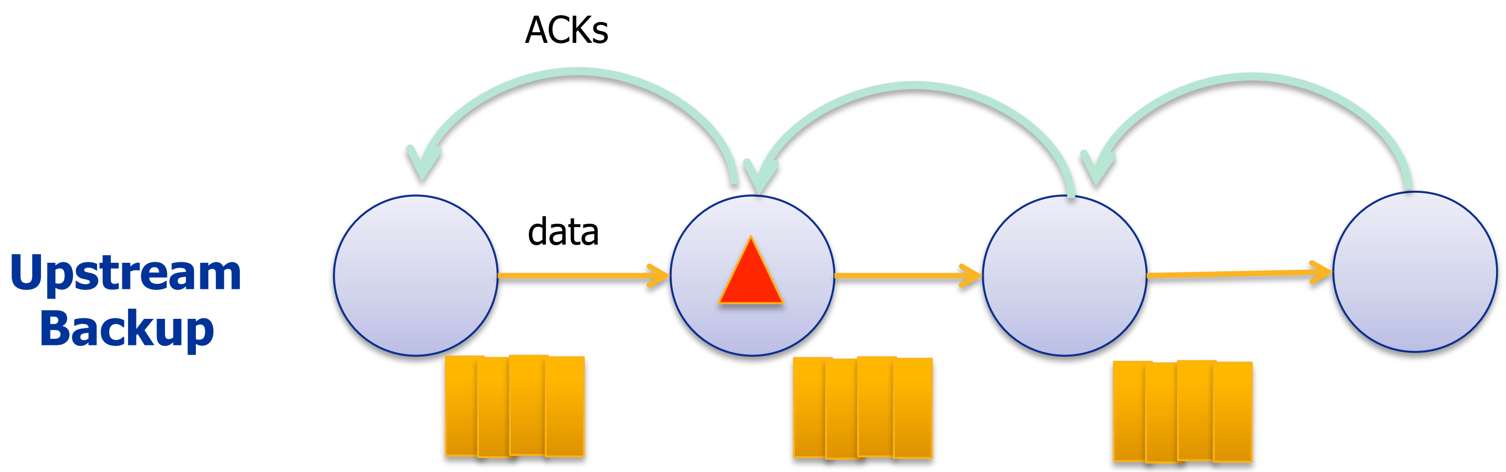
– x is splitting key that partitions state



Passive Fault-Tolerance Model

Recreate operator state by replaying tuples after failure

– Send acknowledgements upstream for tuples processed downstream



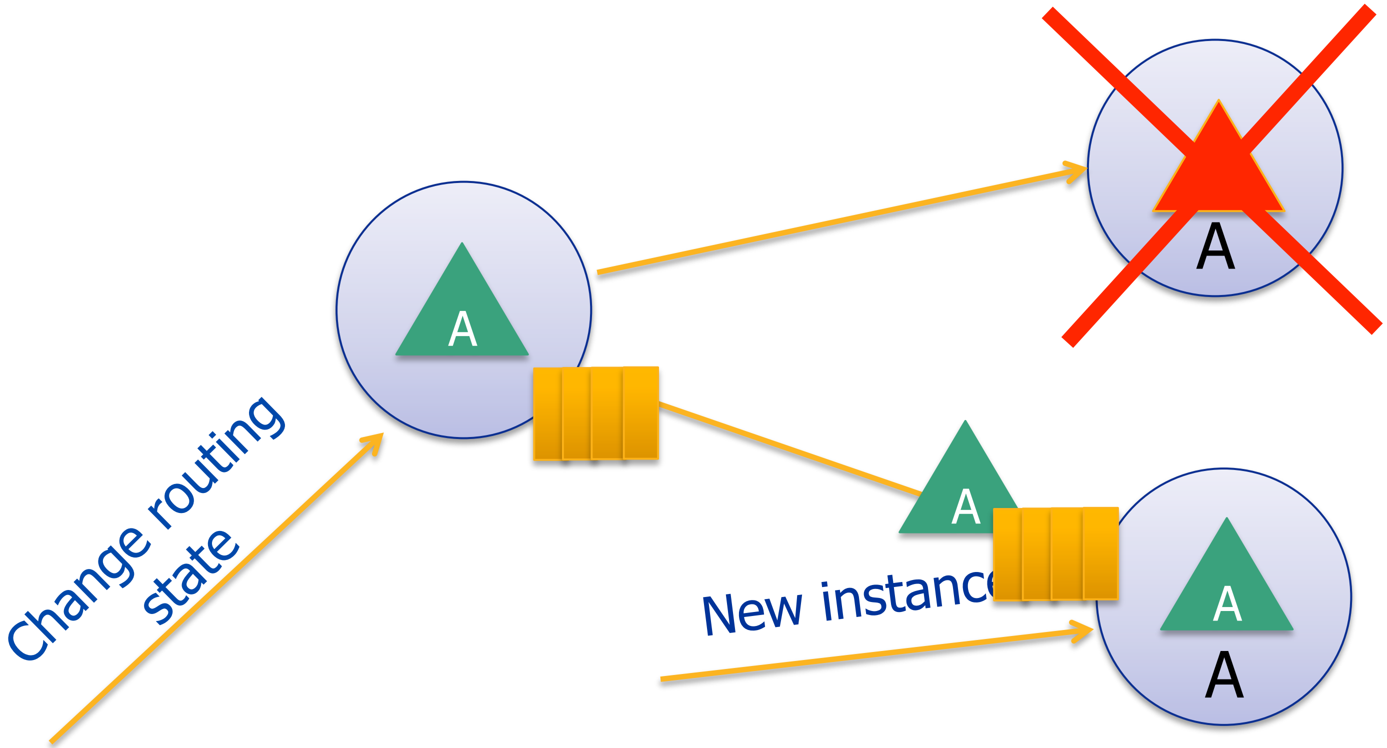
May result in long recovery times due to large buffers

– System is reprocessing streams after failure ➔ inefficient

Upstream Backup + Checkpointing

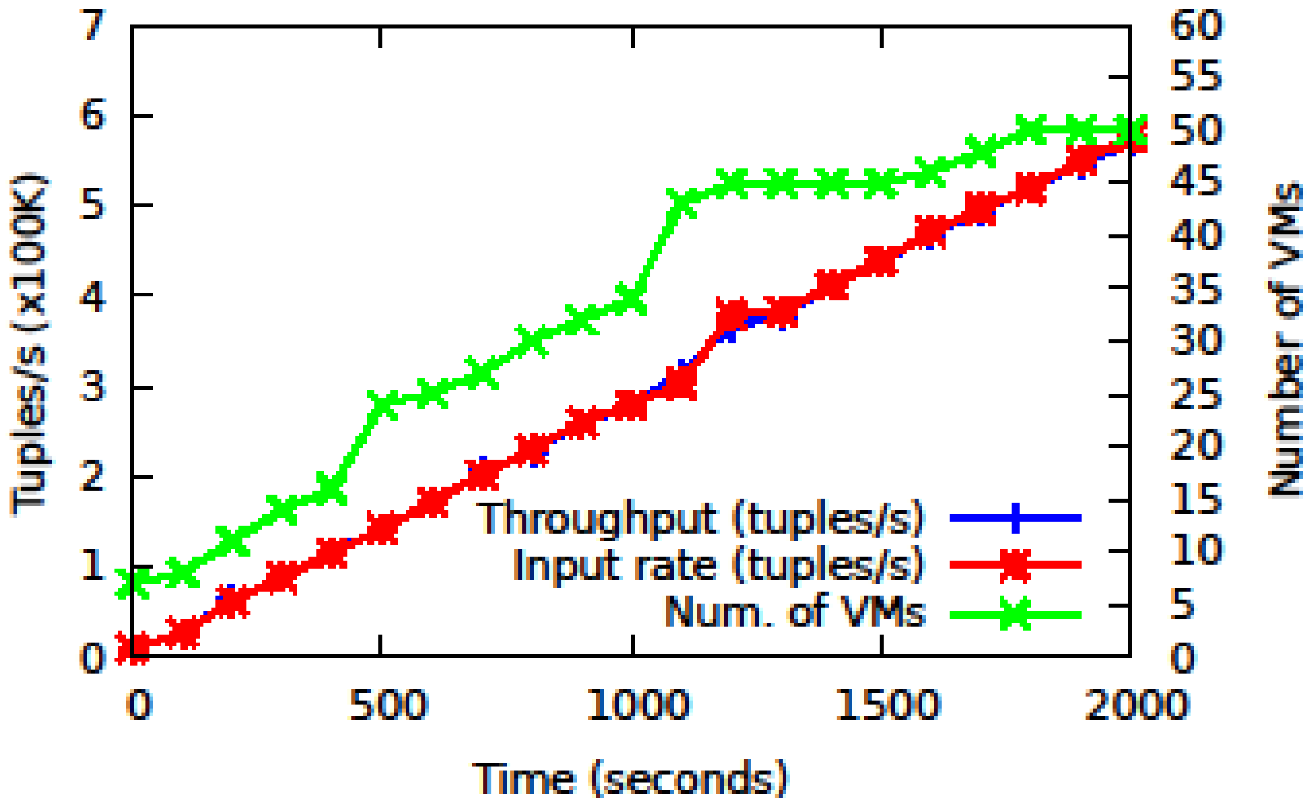
Benefit from state management primitives

- Use periodically backed up state on upstream node to recover faster



State is restored and unprocessed tuples are replayed from buffer

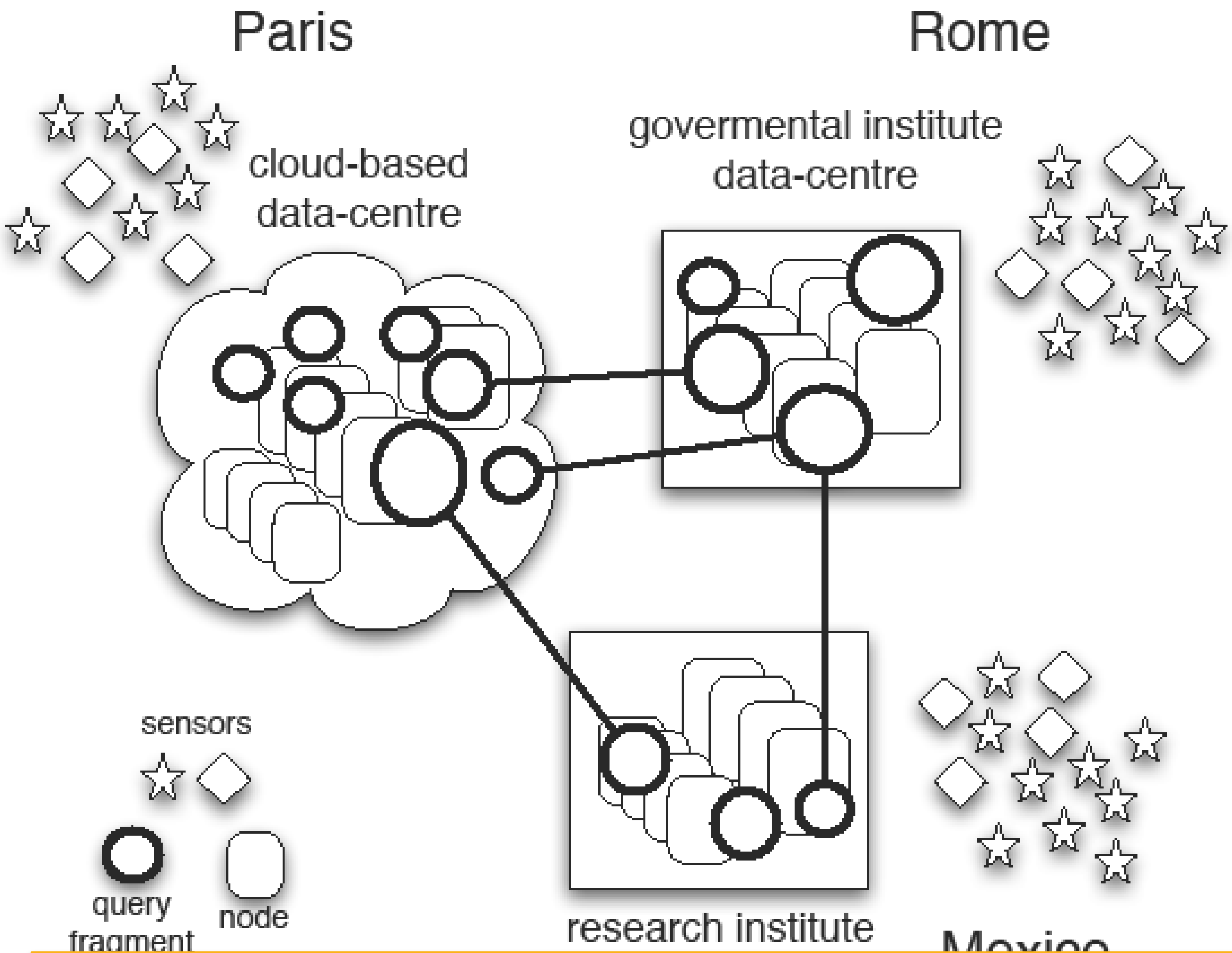
SEEP Evaluation



☛ SEEP scales out to increasing workload in the Linear Road Benchmark

THEMIS: Max-min Fairness in Federated Stream Processing under Overload

Federated Stream Processing System

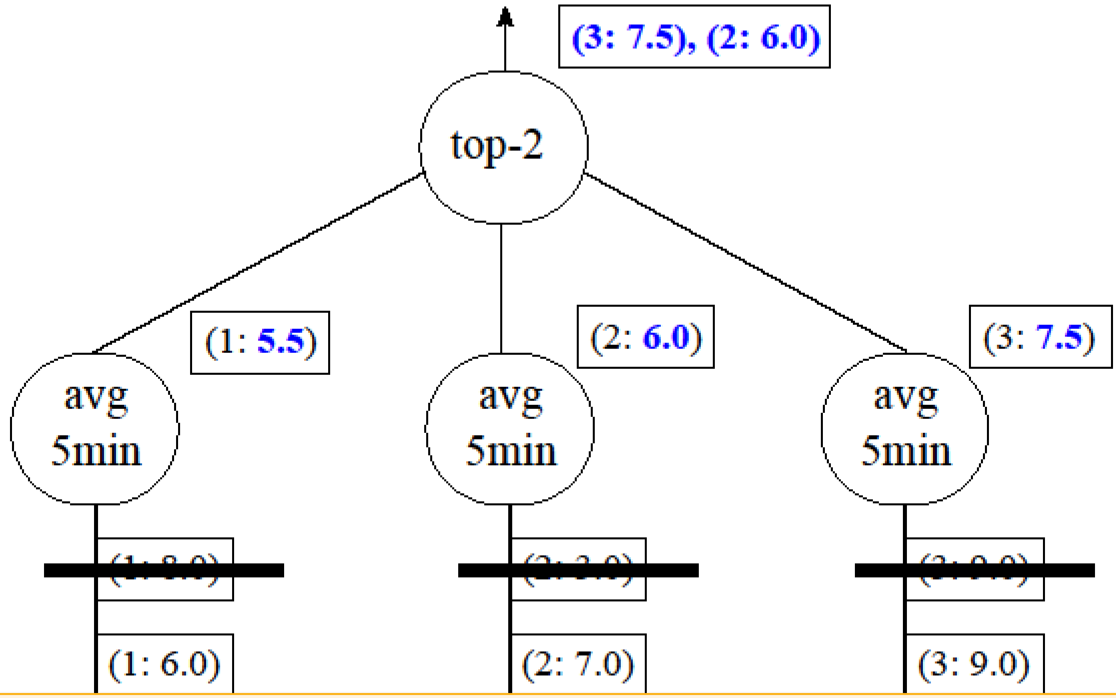


- We cannot scale out to additional resources
- Permanent resource, skewed overload conditions
- Tuple shedding

Tuple Load Shedding → discard data!

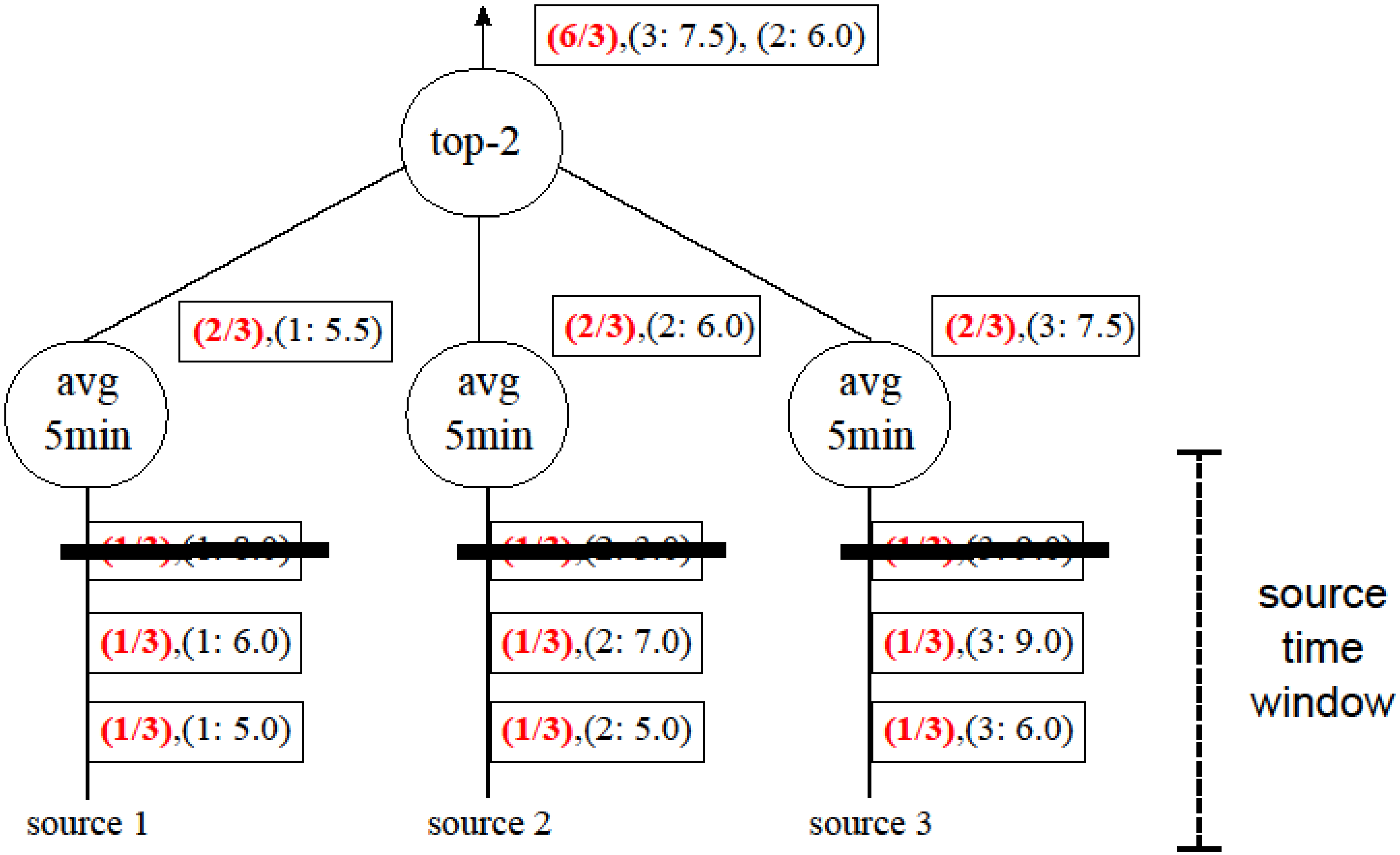
Query:

Which are the two rooms with the highest temperatures, every 5 minutes?



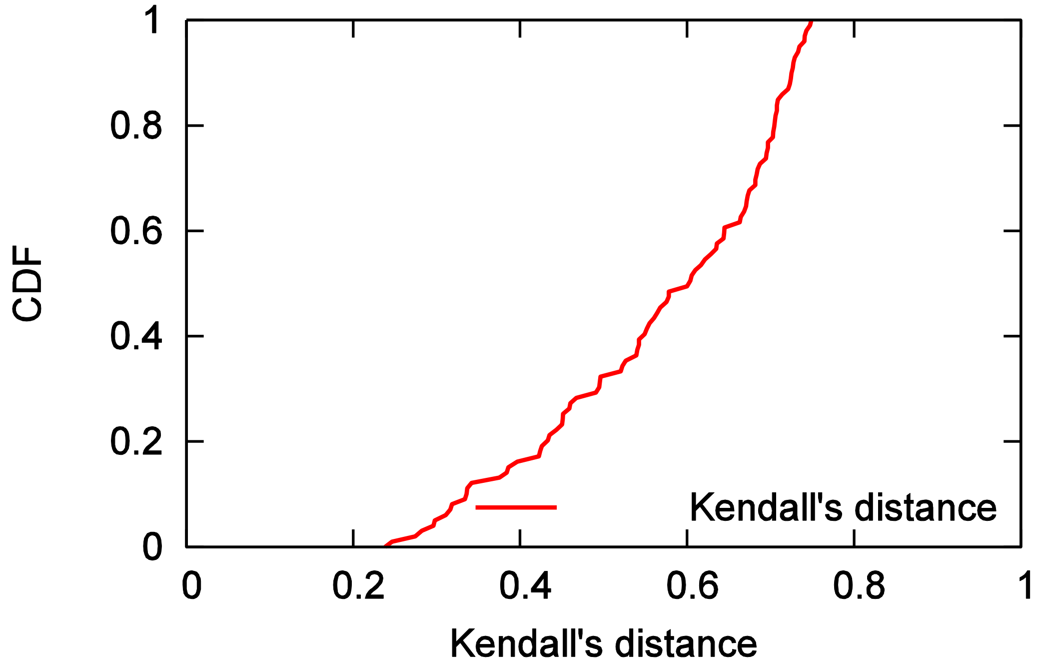
- Reduces resource footprint
- Useful only when feedback is provided to user
- Shedding is controlled for fair processing among queries

Source Information Content (SIC) metric



- SIC metric provides feedback on loss of source tuples
- SIC is query-independent

Unfair Processing in Federated SPSs



- 3 nodes, 100 top-5 queries
- Traces from 40 PlanetLab nodes
- *"Select the 5 nodes with the highest free CPU and at least 500MB of MEM every second"*
- Skewed query deployment

☛ Random shedding → a wide spread in processing quality

Fair Stream Processing in Federated-SPSs

G1: Query-independent processing metric → SIC

G2: Stream processing fairness → max-min SIC

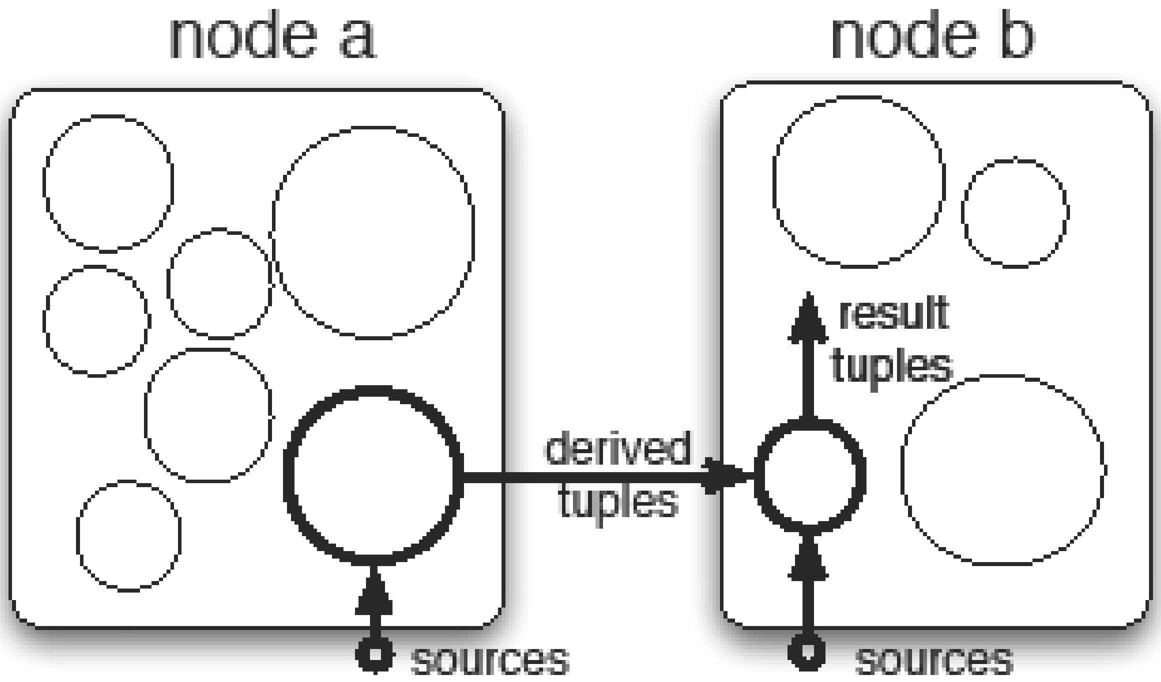
– Some queries are less/more overloaded than others

Max-min SIC Fairness:

The ordering of queries is max-min SIC fair if and if only an increase in the SIC value of a query must be at the expense of the decrease of the SIC value of an already smaller query.

G3: Decentralised fairness → sites are autonomous

Max-min Decentralised Fairness Challenges

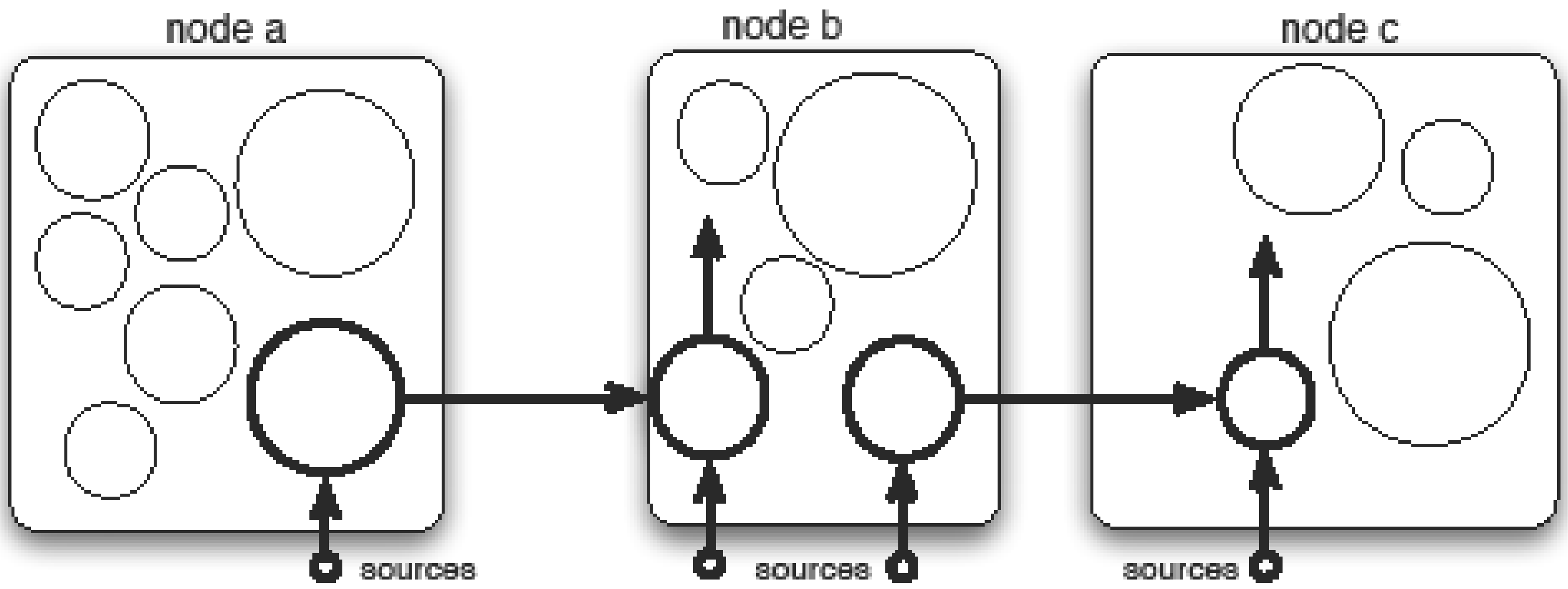


assume (node a) << (node b)

Research question:

how can we balance shedding so to maximise SIC values on (node a) queries?

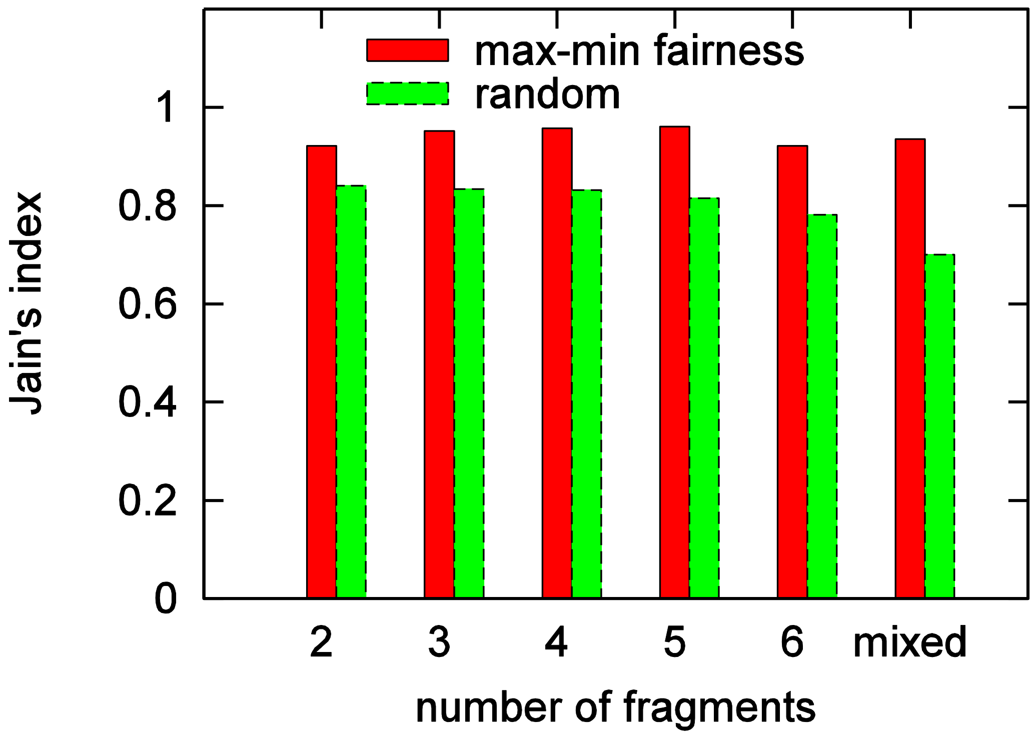
Max-min Decentralised Fairness Solution



Solution insights:

- Each node solves a max-min problem for its running queries
- Each node is updated on the result SIC value of its queries
  - → nodes take informed local decisions for global fairness
- Each node always sheds the least SIC tuples
  - → save on resources
- Solve a small problem at-a-time and iterate with feedback

THEMIS Evaluation



– 18 nodes, 2,000 fragments
– Mix workload: cov, top-5,avg

☛ THEMIS max-min fairness is always better than random

Conclusions

Data Stream Processing is efficient in the Cloud

- New challenges emerge from Cloud scalability
  - Scale out and fault-tolerance have to be integrated
- New problems arise because of distribution
  - Fairness in overload management requires feedback of processing

Future work -> Cloud is there but does not come cheap

- Large-scale management
- Competing requirements from multi-tenancy deployment
- Unknown changing workloads
- Pay-as-you-go model, is this the best?
- Minimise the cost for users, maximise Cloud providers’ revenue
- Novel architectural designs for data-centre management

Thank you! ekalyv@imperial.ac.uk

Experimental Evaluation

Goals

- Correlation of SIC metric with result correctness
- Effectiveness of the max-min fairness algorithm
- Scalability of the fairness algorithm
- Overhead of our shedder implementation

Prototype system: THEMIS

- Implemented in Java

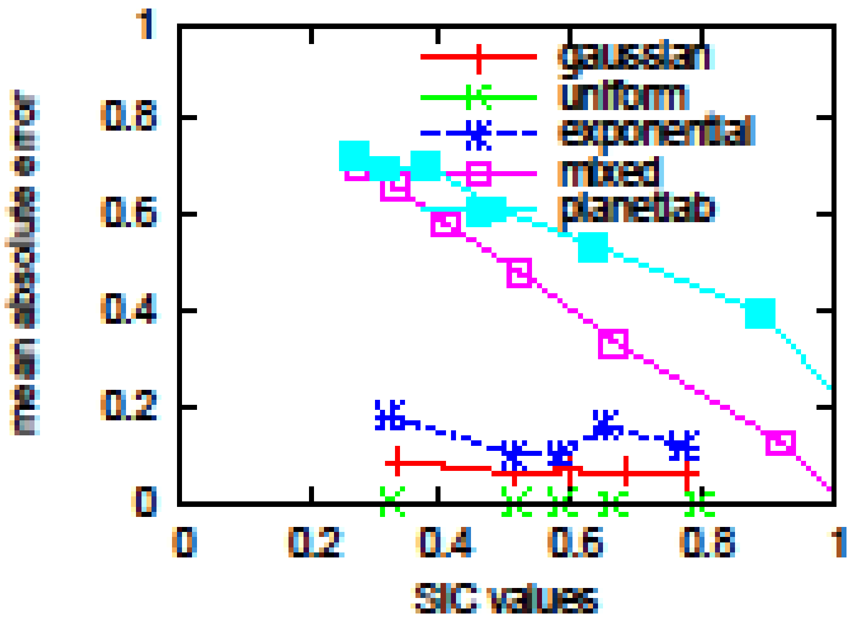
Workload

- Aggregate workload (max, count, avg)
- Complex workload (top-5, avg-all, covariance)
- Synthetic data (uniform, Gaussian, exponential)
- PlanetLab data (CPU and memory usags, 1month, 40 nodes)

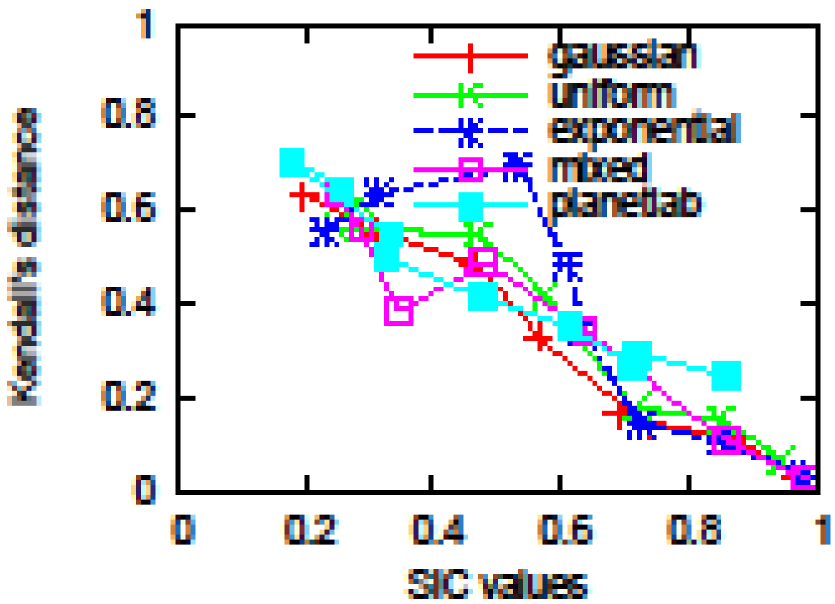
Deployment on local and Emulab (18 nodes) test-beds

THEMIS Evaluation

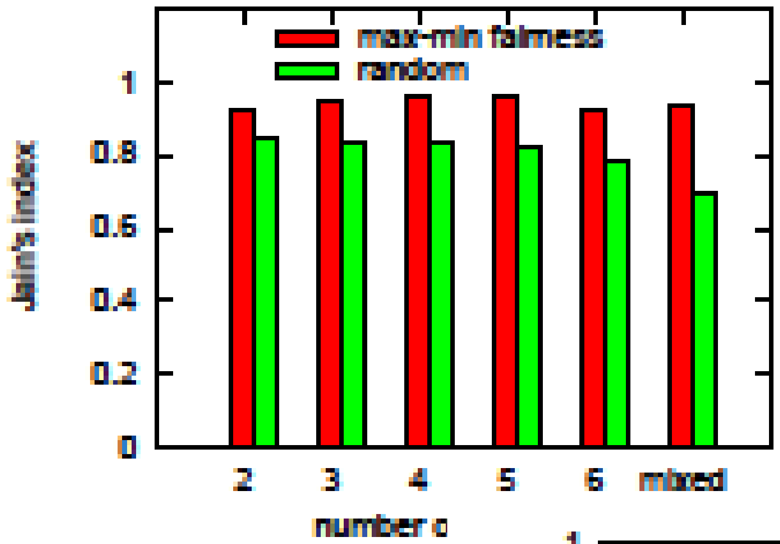
max query

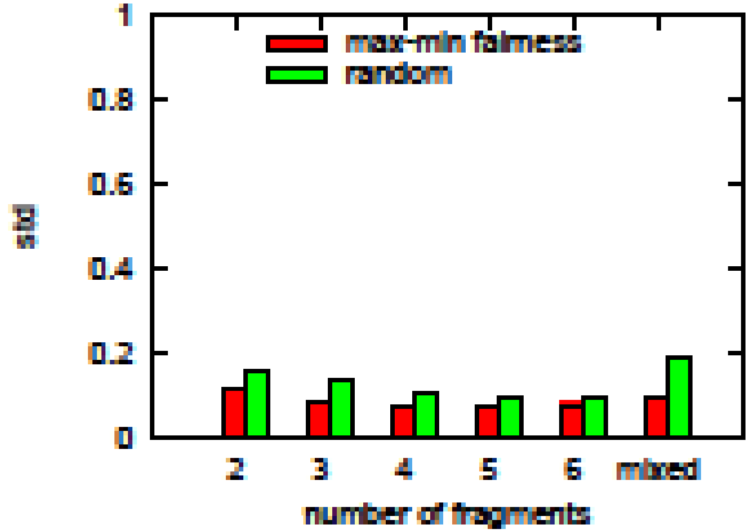


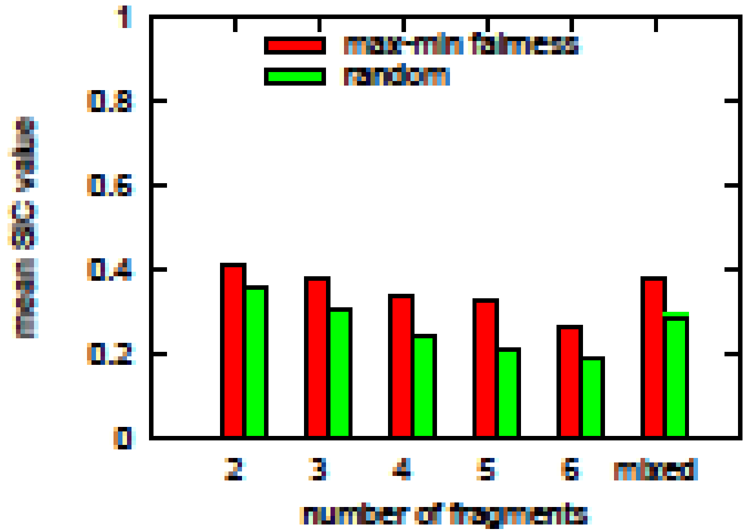
top-5 query



THEMIS Evaluation







Experimental Evaluation

Goals

- Investigate effectiveness of scale out mechanism
- Recovery time after failure using UBC
- Overhead of state management

Prototype system: Scalable and Elastic Event Processing (SEEP)

- Implemented in Java; Storm-like data flow model

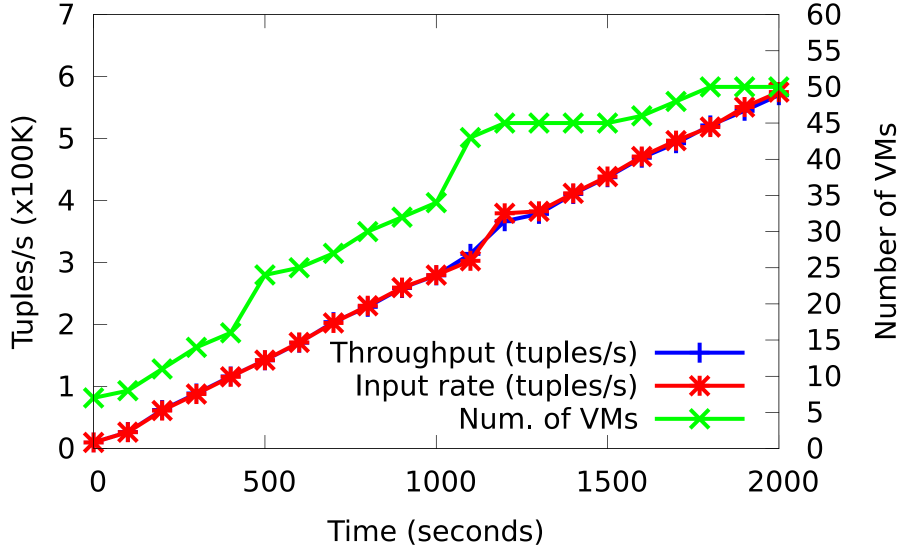
Sample queries + workload

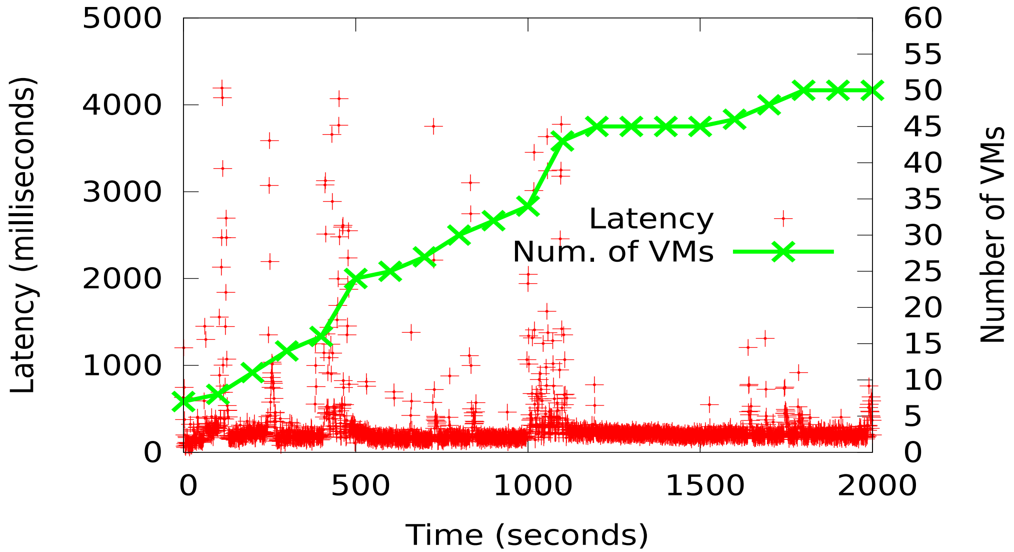
- **Linear Road Benchmark** (LRB) to evaluate scale out [VLDB'04]
  - Provides an increasing stream workload over time for given load factor
  - Query with 8 operators; SLA: results < 5 secs
- **Windowed word count query** to evaluate fault tolerance
  - Induce failure to observe performance impact

Deployment on Amazon AWS EC2

- Sources and sinks on high-memory double extra large instances
- Operators on small instances

Scale Out: LRB Workload





Scales to load factor L=350 with 60 VMs on Amazon EC2

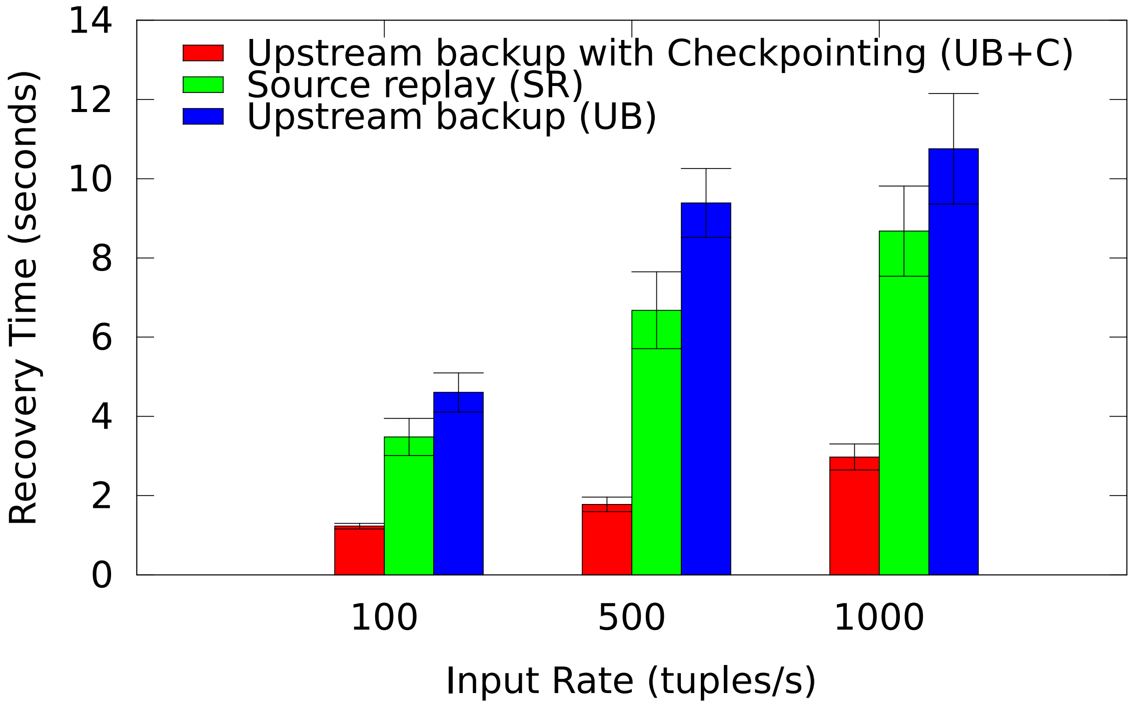
- Automated query parallelisation

L=512 highest report result [VLDB'12]

- Hand-crafted query on dedicated cluster

Scale out leads to latency peaks, but remains within LRB SLA

UB+C: Recovery Time

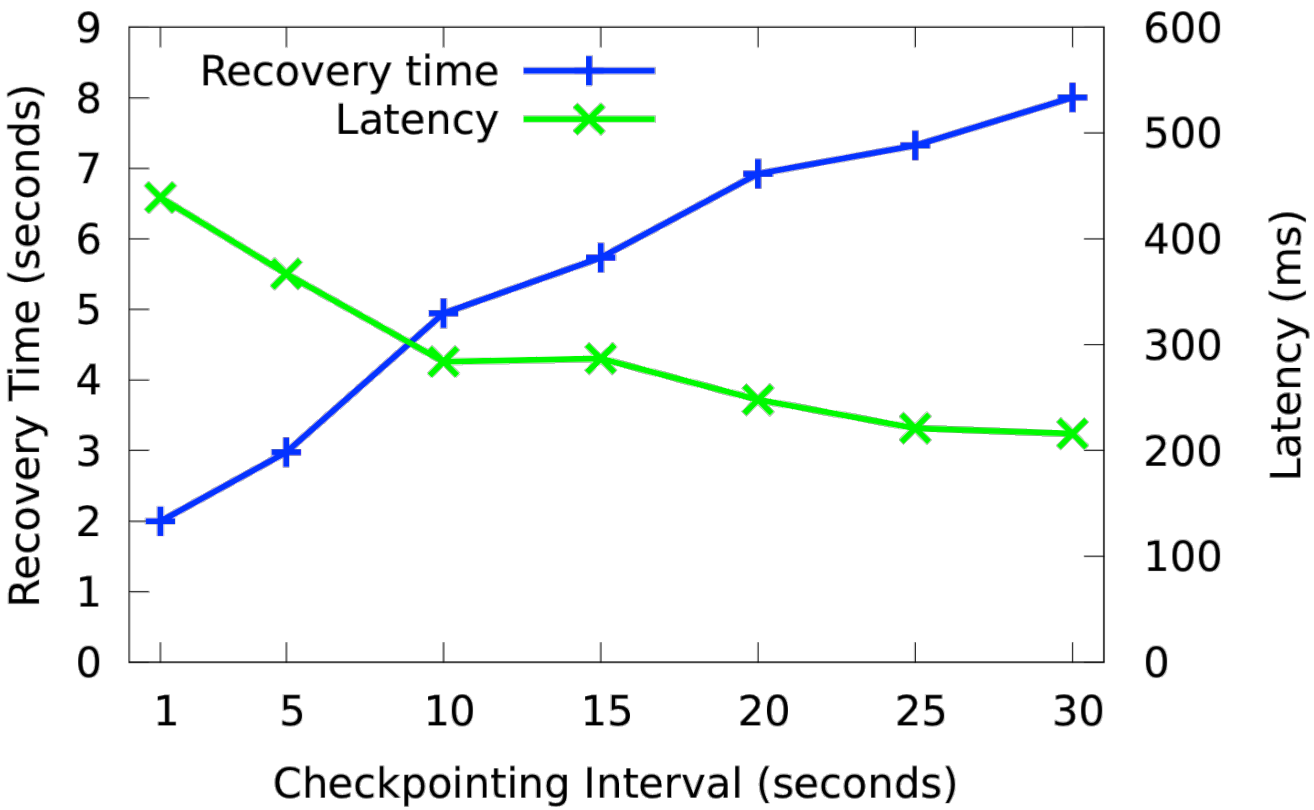


Source Replay:
Upstream Backup with tuples replayed by source only

State backed up every 5 seconds in UB+C

☛ UB+C achieves faster recovery, especially for fast stream rates

Tradeoff of Checkpointing Interval



☛ Shorter checkpointing interval leads to faster recovery times
But also incurs more overhead, impacting tuple processing latency

Related Work

Scalable stream processing systems

- **Twitter Storm, Yahoo S4, Nokia Dempsey**
  Exploit operator parallelism mainly for stateless queries
- **ParaSplit operator** [VLDB'12]
  Partition stream for intra-query parallelism

Support for elasticity

- **StreamCloud** [TPDS'12]
  Dynamic scale out/in for subset of relational stream operators
- **Esc** [ICCC'11]
  Dynamic support for stateless scale out

Resource-efficient fault tolerance models

- **Active Replication at (almost) no cost** [SRDS'11]
  Use under-utilized machines to run operator replicas
- **Discretized Streams** [HotCloud'12]
  Data is checkpointed and recovered in parallel in event of failure

Future Work

Support for full elasticity

- Add dynamic scale in mechanism
- Bottlenecks easier to detect than spare capacity

Cost-aware policies for elasticity

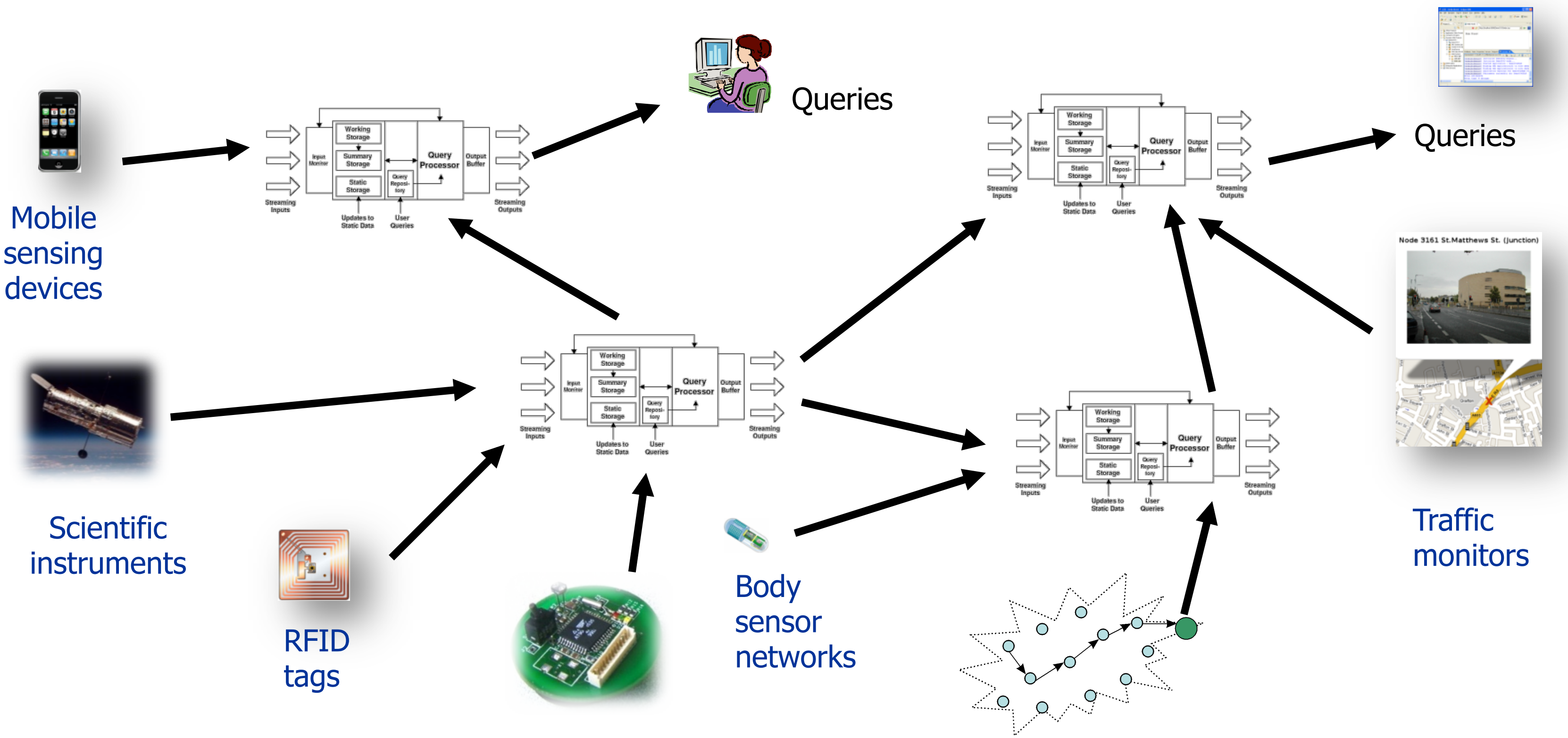
- Performance/cost tradeoff
- How to achieve user-provided SLAs

High-level query languages

- Integrated support for processing stream & historic data
- Programming models

Distributed DSPS

- Interconnect multiple DSPSs with network
  - Better scalability, handles geographically distributed stream sources



- Interconnect on LAN or Internet?
  - Different assumptions about time and failure models

Twitter Storm & Yahoo S4

Yahoo! S4 (http://incubator.apache.org/s4/)

- Java framework for implementing stream processing applications
- Hides stream "plumbing" from developers
- Uses Zookeeper for coordination

Twitter Storm (https://github.com/nathanmarz/storm)

- Focus on fault-tolerance: acknowledgement of processed tuples
- Spouts produce data; bolts process data
- Different mechanisms for stream partitioning and bolt parallelisation

This is just the beginning... lots of open challenges...