Imperial College London

Department of Computing

Drinking From The Fire Hose: Scalable Stream Processing Systems

Peter Pietzuch
prp@doc.ic.ac.uk

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

Cambridge MPhil – November 2014

The Data Deluge

1200 Exabytes (billion GBs) created in 2010 alone

- Increased from 150 Exabytes in 2005

Many new sources of data become available

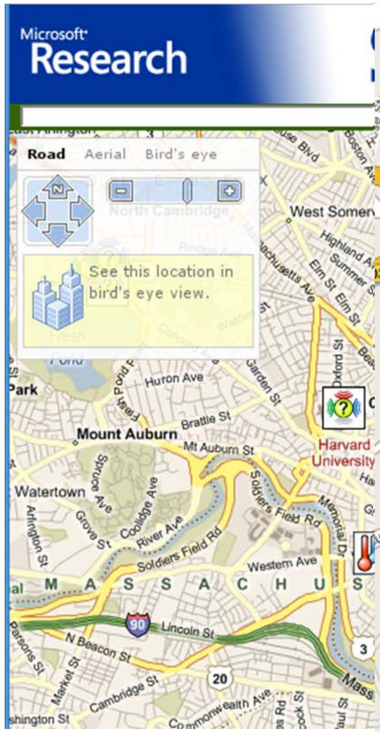
- Sensors, mobile devices
- Web feeds, social networking
- Cameras
- Databases
- Scientific instruments

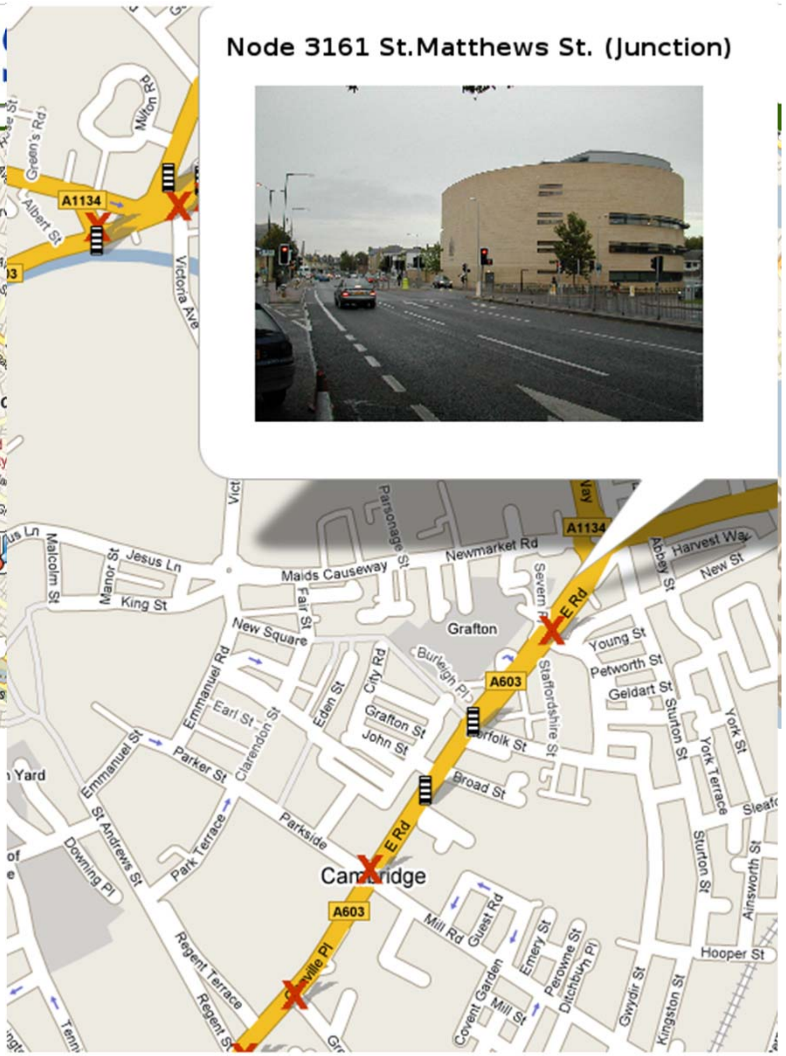
☛ **How can we make sense of all data ?**

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

Real Time Traffic Monitoring

Instrumenting country's transportation infrastructure





Time-EACM
(Cambridge)

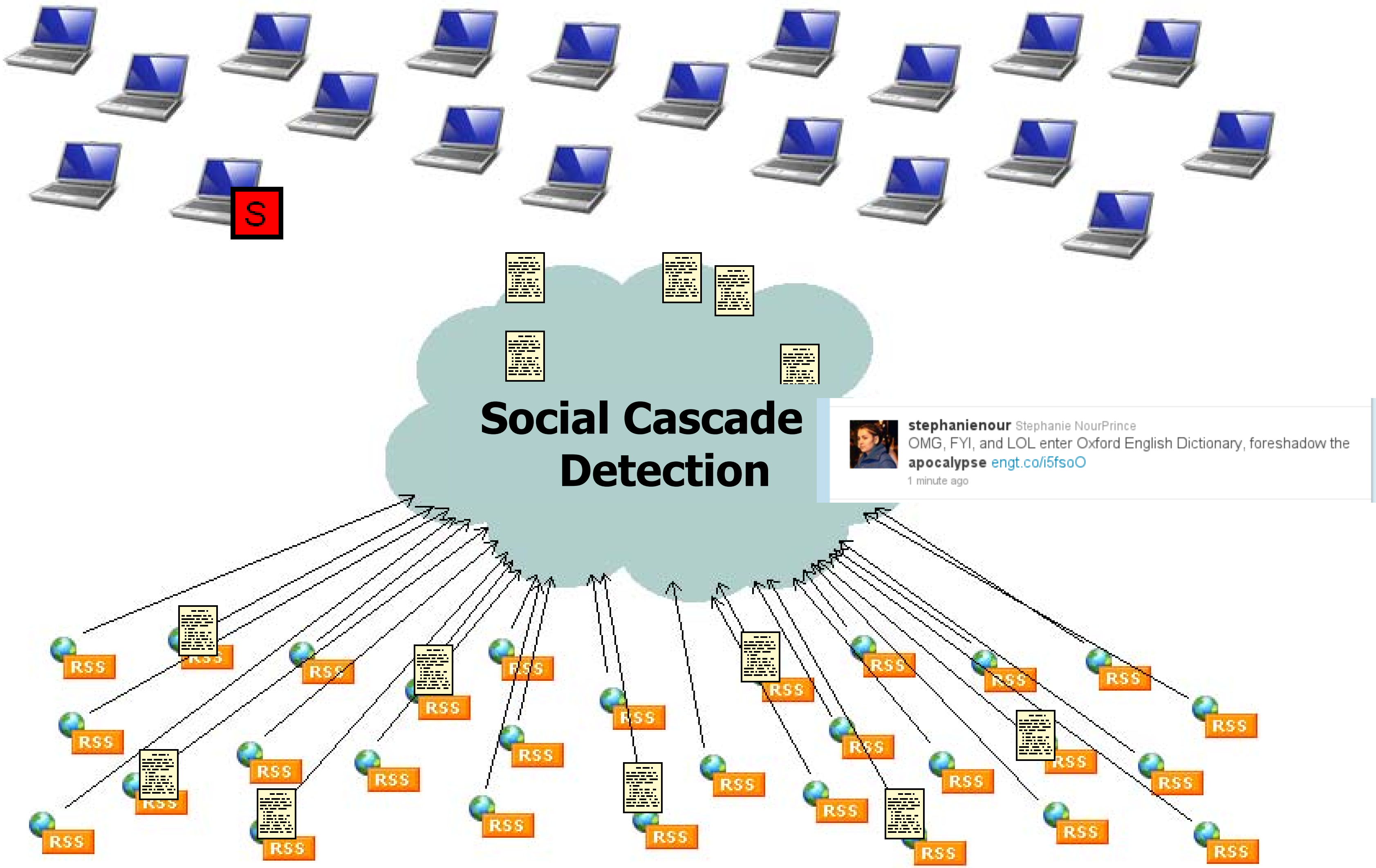
Many parties interested in data

- Road authorities, traffic planners, emergency services, commuters
- But access not everything: Privacy

High-level queries

- "What is the best time/route for my commute through central London between 7-8am?"

Web/Social Feed Mining



Detection and reaction to social cascades

Fraud Detection

How to detect identity fraud as it happens?

Illegal use of mobile phone, credit card, etc.

- Offline: avoid aggravating customer
- Online: detect and intervene

Huge volume of call records

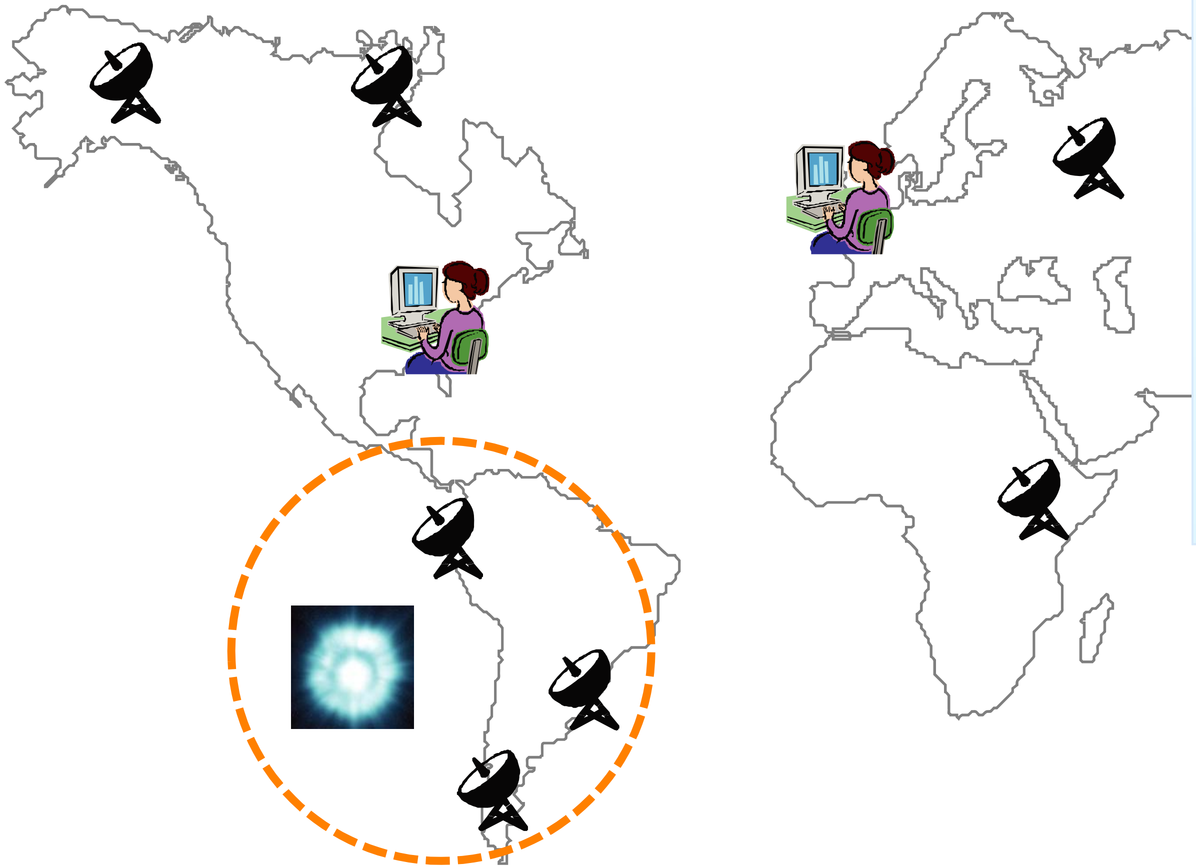
More sophisticated forms of fraud

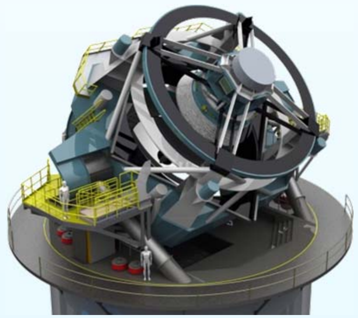
- e.g. insider trading

Supervision of laws and regulations

- e.g. Sabanes-Oxley, real-time risk analysis

Astronomic Data Processing





Large Synoptic Survey Telescope (LSST)

- Generates 1.28 Petabytes per year

Analysing transient cosmic events: γ-ray bursts

Stream Processing to the Rescue!

☛ Process data streams on-the-fly without storage

Stream data rates can be high

- High resource requirements for processing (clusters, data centres)

Processing stream data has real-time aspect

- Latency of data processing matters
- Must be able to react to events as they occur

Traditional Databases (Boring)

D

Qu

es

Data Stream Processing System

result stream

- Indexing?

Overview

Why Stream Processing?

Stream Processing Models

- Streams, windows, operators

Stream Processing Systems

- Distributed Stream Processing
- Scalable Stream Processing with Distributed Dataflows
- Stateful dataflow graphs for stream processing

Stream Processing

Need to define

1. Data model for streams

2. Processing (query) model for streams

Data Stream

"A **data stream** is a real-time, continuous, ordered (implicitly by arrival time or explicitly by timestamp) **sequence of items**. It is impossible to control the order in which items arrive, nor is it feasible to locally store a stream in its entirety."
[Golab & Ozsu (SIGMOD 2003)]

Relational model for stream structure?

- Can't represent audio/video data
- Can't represent analogue measurements

Relational Data Stream Model

Streams consist of infinite sequence of tuples

- Tuples often have associated time stamp
  - e.g. arrival time, time of reading, ...

Tuples have fixed relational schema

- Set of attributes

id = 27182
temp = 24 C
rain = 20mm

sensor output

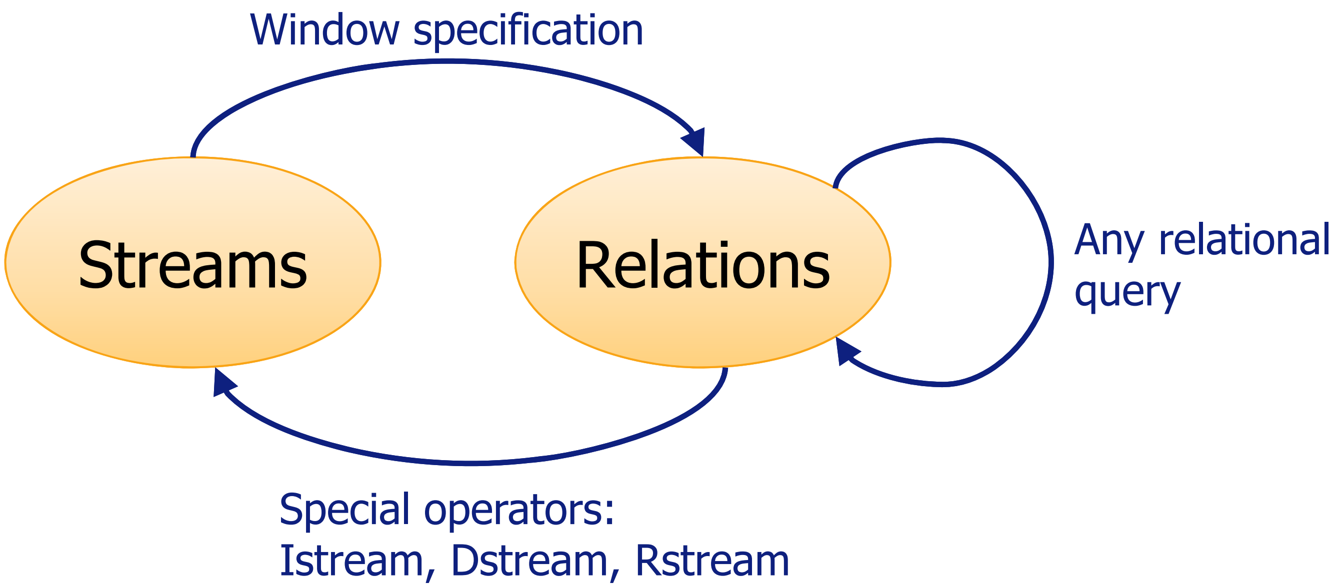
`Sensors(id, temp, rain)`

t_1	t_2	t_3	t_4	...					
id temp rain	id temp rain	id temp rain	id temp rain	id temp rain	id temp rain	id temp rain	id temp rain	id temp rain	id temp rain

Sensors data stream

time

Stream Relational Model



Window converts stream to dynamic relation

- Similar to maintaining view
- Use regular relational algebra operators on tuples
- Can combine streams and relations in single query

Sliding Window I

How many tuples should we process each time?

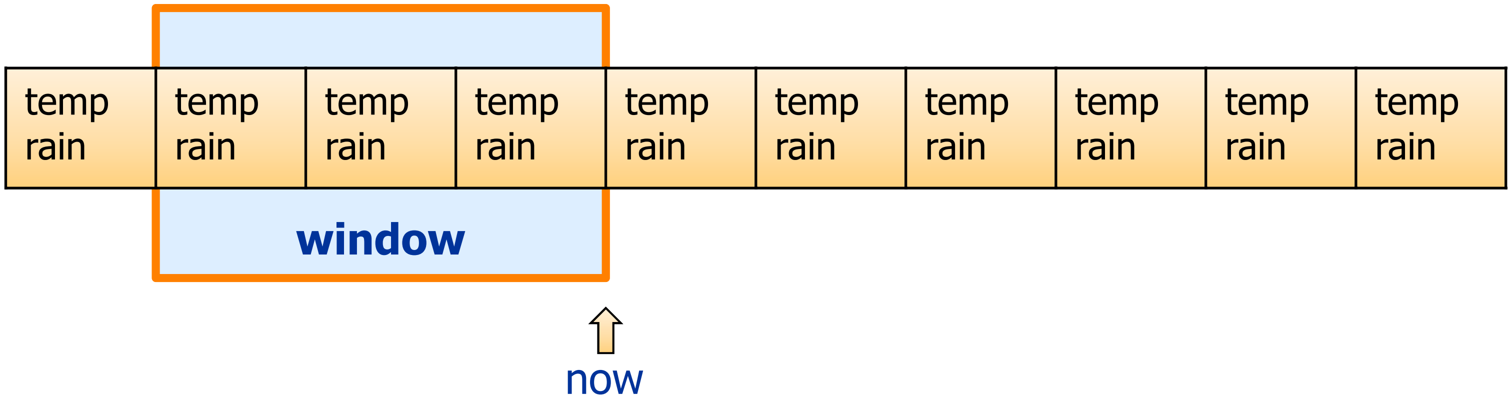
Process tuples in window-sized batches

Time-based window with size τ at current time t

[t - τ : t]	`Sensors [Range τ seconds]`
[t : t]	`Sensors [Now]`

Count-based window with size n:

last n tuples	`Sensors [Rows n]`



Sliding Window II

How often should we evaluate the window?

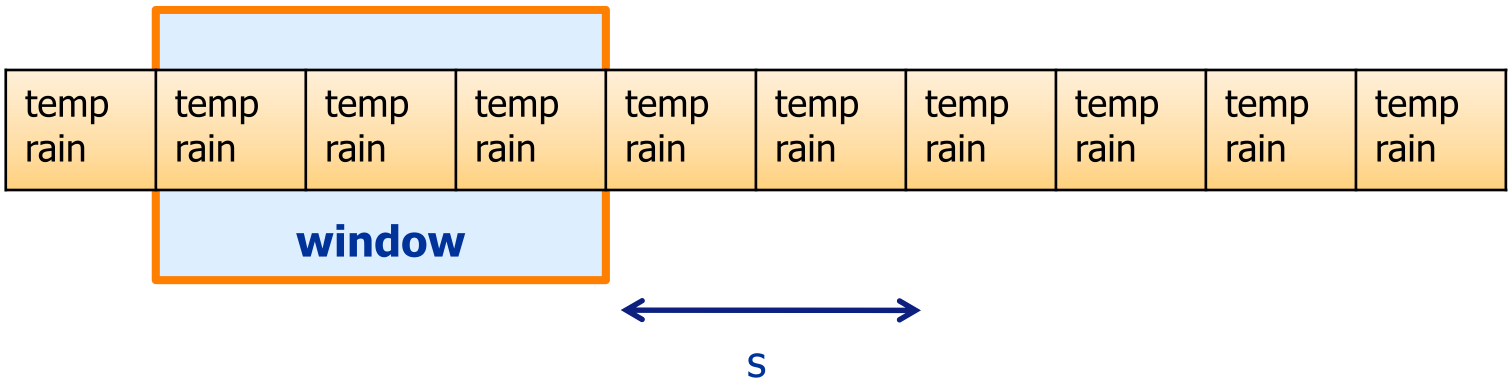
1. Output new result tuples as soon as available
   - Difficult to implement efficiently

2. Slide window by s seconds (or m tuples)

```
Sensors [Slide s seconds]
```

Sliding window: $s < T$

Tumbling window: $s = T$



Continuous Query Language (CQL)

Based on SQL with streaming constructs

- Tuple- and time-based windows
- Sampling primitives

```
SELECT temp
FROM Sensors [Range 1 hour]
WHERE temp > 42;
```

```
SELECT *
FROM S1 [Rows 1000],
     S2 [Range 2 mins]
WHERE S1.A = S2.A
   AND S1.A > 42;
```

Apart from that regular SQL syntax

Join Processing

Naturally supports joins over windows

```
SELECT *
FROM S1, S2
WHERE S1.a = S2.b;
```

Only meaningful with window specification for streams

- Otherwise requires unbounded state!

Sensors(time, id, temp, rain) **Faulty(time, id)**

```
SELECT S.id, S.rain
FROM Sensors [Rows 10] as S, Faulty [Range 1 day] as F
WHERE S.rain > 10 AND F.id != S.id;
```

Converting Relations ➔ Streams

Define mapping from relation back to stream

- Assumes discrete, monotonically increasing timestamps $\tau, \tau+1, \tau+2, \tau+3, \ldots$

Istream(R)

- Stream of all tuples (r, τ) where $r \in R$ at time τ but $r \notin R$ at time $\tau-1$

Dstream(R)

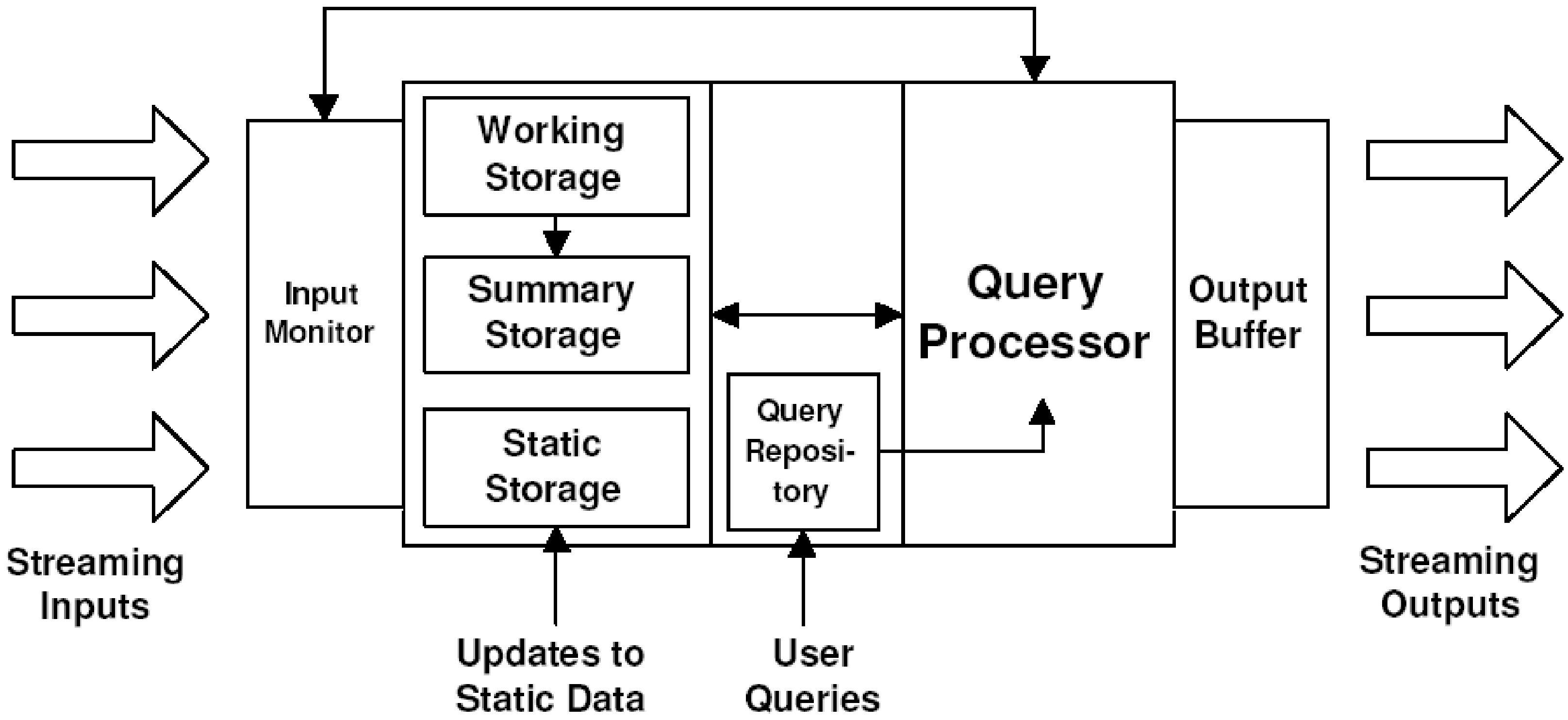
- Stream of all tuples (r, τ) where $r \in R$ at time $\tau-1$ but $r \notin R$ at time τ

Rstream(R)

- Stream of all tuples (r, τ) where $r \in R$ at time τ

Stream Processing Systems

General DSPS Architecture



Source: Golab & Ozsu 2003

Stream Query Execution

Continuous queries are long-running
➔ properties of base streams may change

- Tuple distribution, arrival characteristics, query load, available CPU, memory and disk resources, system conditions, ...

Solution: Use **adaptive query plans**

- Monitor system conditions
- Re-optimise query plans at run-time

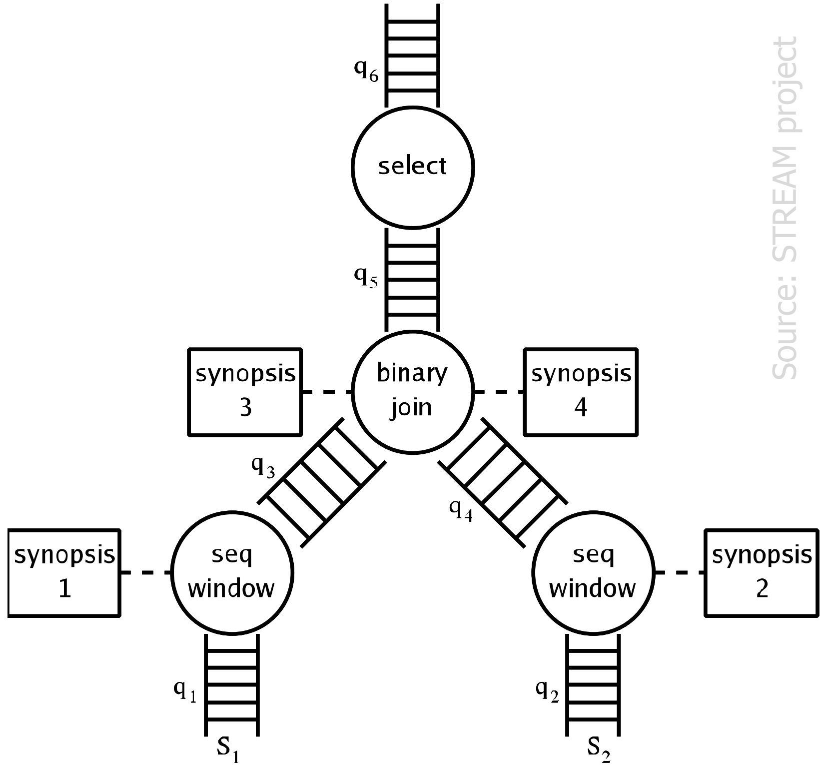
DBMS didn’t quite have this problem...

Query Plan Execution

Executed query plans include:

– **Operators**
– **Queues** between operators
– **State**/"Synposis" (windows, ...)
– **Base streams**

```
SELECT *
FROM S1 [Rows 1000],
     S2 [Range 2 mins]
WHERE S1.A = S2.A
   AND S1.A > 42;
```



Source: STREAM project

Challenges

– State may get large (e.g. large windows)

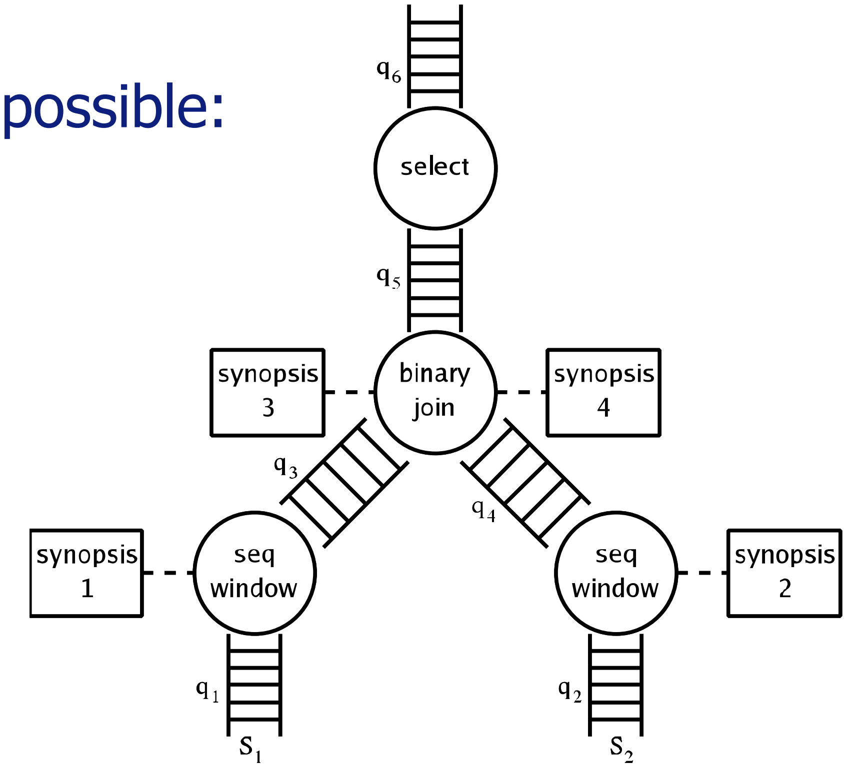
Operator Scheduling

Need scheduler to invoke operators (for time slice)

- Scheduling must be adaptive

Different scheduling disciplines possible:

1. Round-robin
2. Minimise queue length
3. Minimise tuple delay
4. Combination of the above



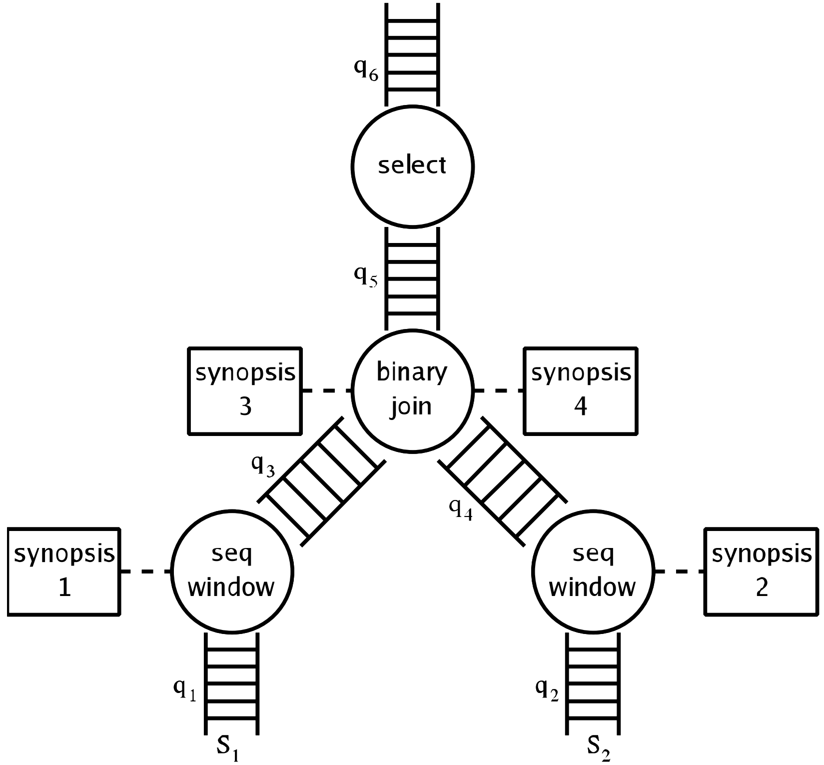
Load Shedding

DSMS must handle overload:
Tuples arrive faster than processing rate

Two options when overloaded:

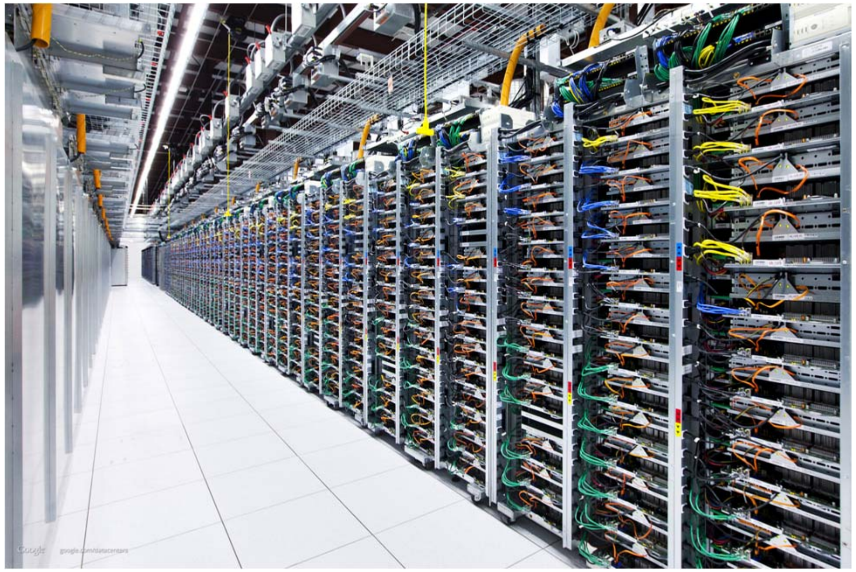
1. **Load shedding**: Drop tuples
   - Much research on deciding which tuples to drop: c.f. result correctness and resource relief
   - e.g. sample tuples from stream

2. **Approximate processing**: Replace operators with approximate processing
   - Saves resources



Scalable Stream Processing

Big Data Centres + Big Data



Google: 20 data centre locations

- over 1 million servers
- 260 Megawatts (0.01% of global energy)
- 4.2 billion searches per day (2011)
- Exabytes (10^{18}) of storage

Assumptions:

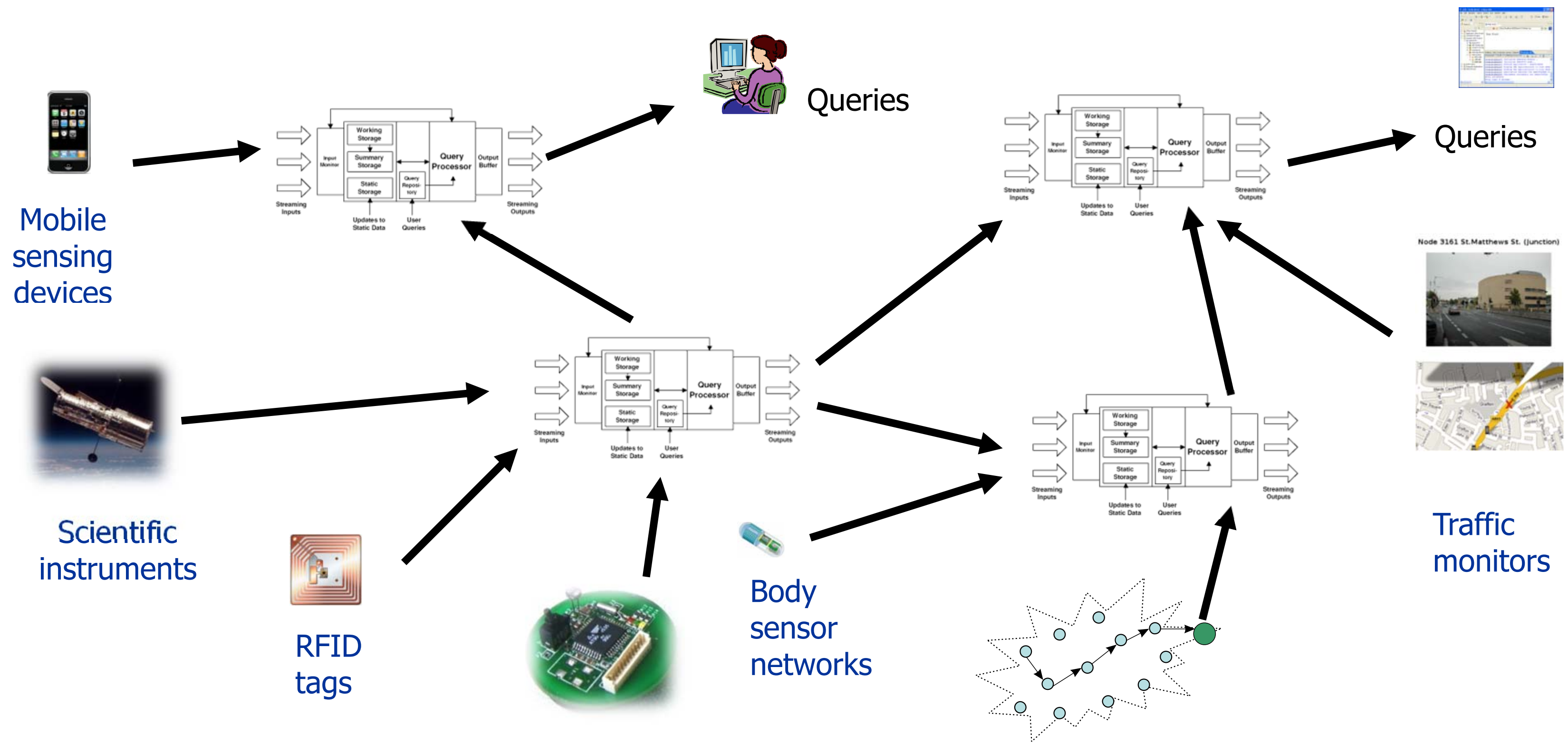
- **Scale out** and not scale up
  - Commodity servers with local disks
  - Data-parallelism is king
- Software designed for **failure**

Platforms for stream processing?

Distributed Stream Processing

Interconnect multiple DSPSs with network

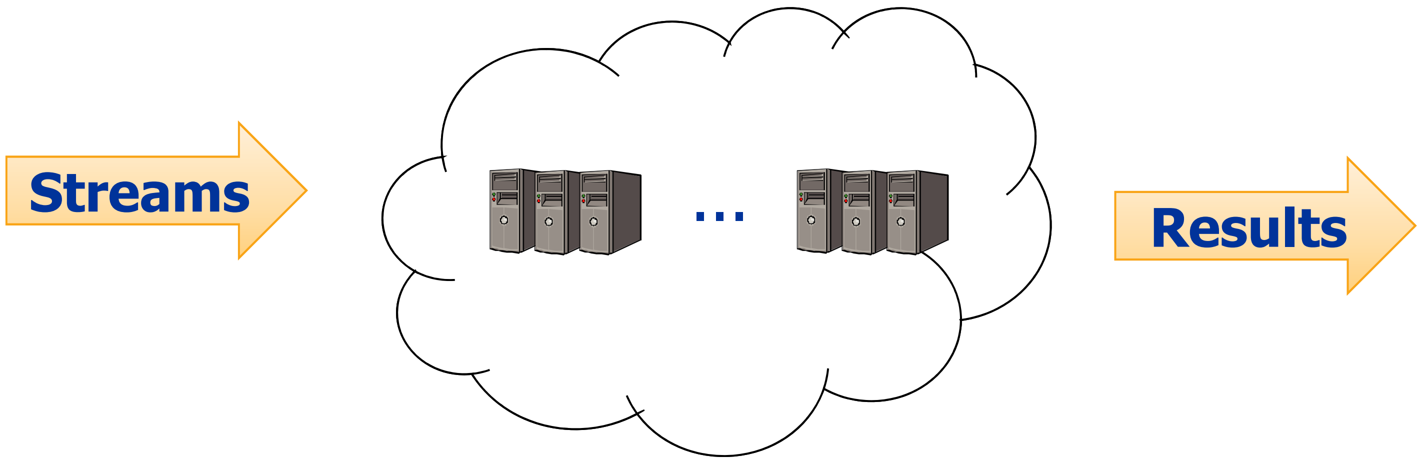
– Better scalability, handles geographically distributed stream sources



Stream Processing in the Cloud

Clouds provide virtually infinite pools of resources

– Fast and cheap access to new machines for operators



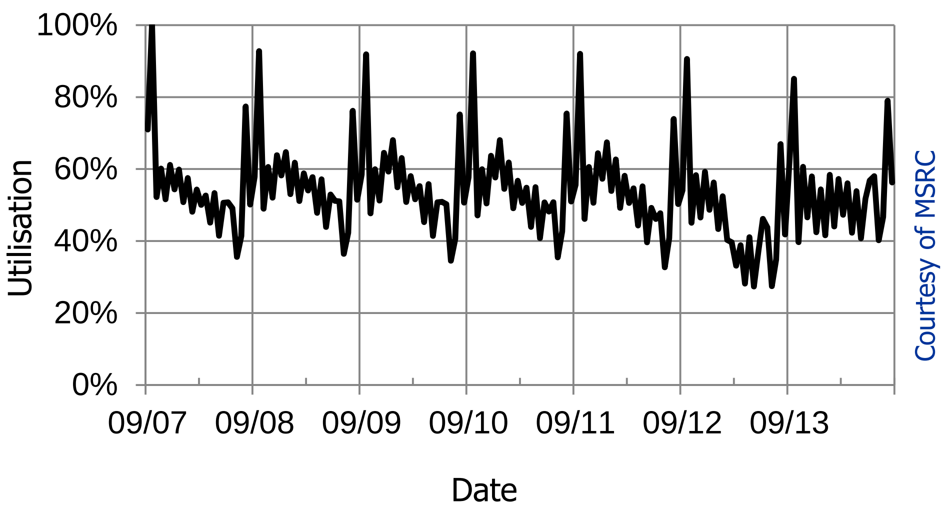
n virtual machines in cloud data centre

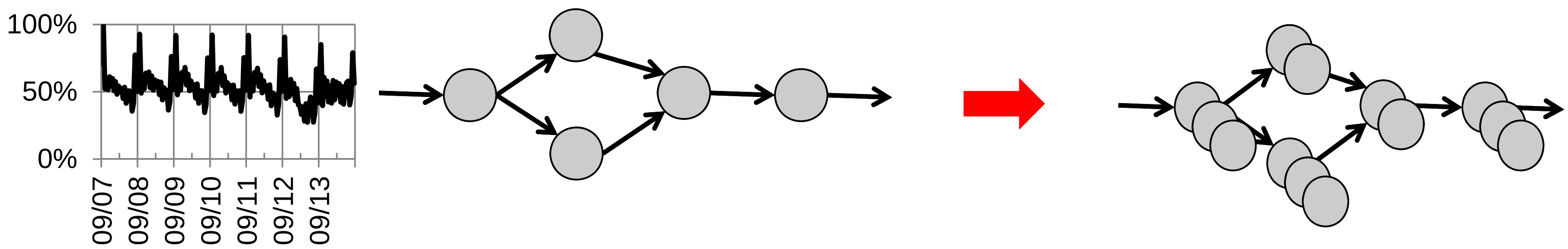
☛ How do you decide on the optimal number of VMs?

– Needlessly overprovisioning system is expense
– Using too few nodes leads to poor performance

Challenge 1: Elastic Data-Parallel Processing

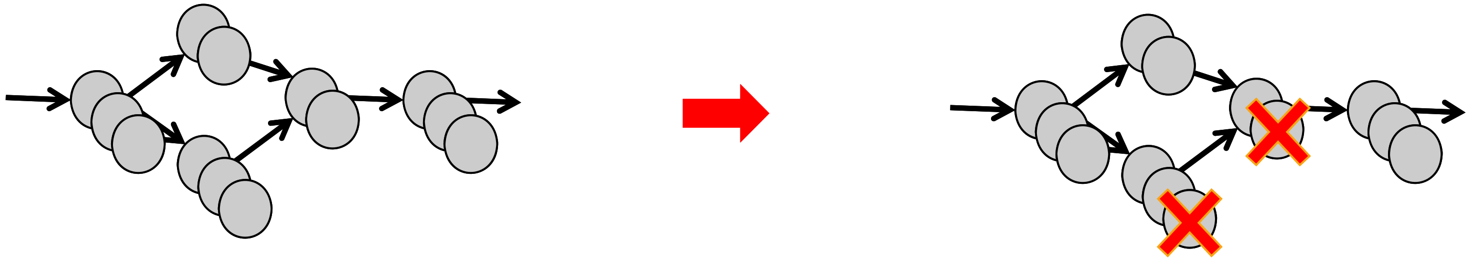
Typical stream processing workloads are bursty





High + **bursty** input rates ➔ Detect **bottleneck** + **parallelise**

Challenge 2: Fault-Tolerant Processing



Large scale deployment ➔ Handle node **failures**

Failure is a common occurrence

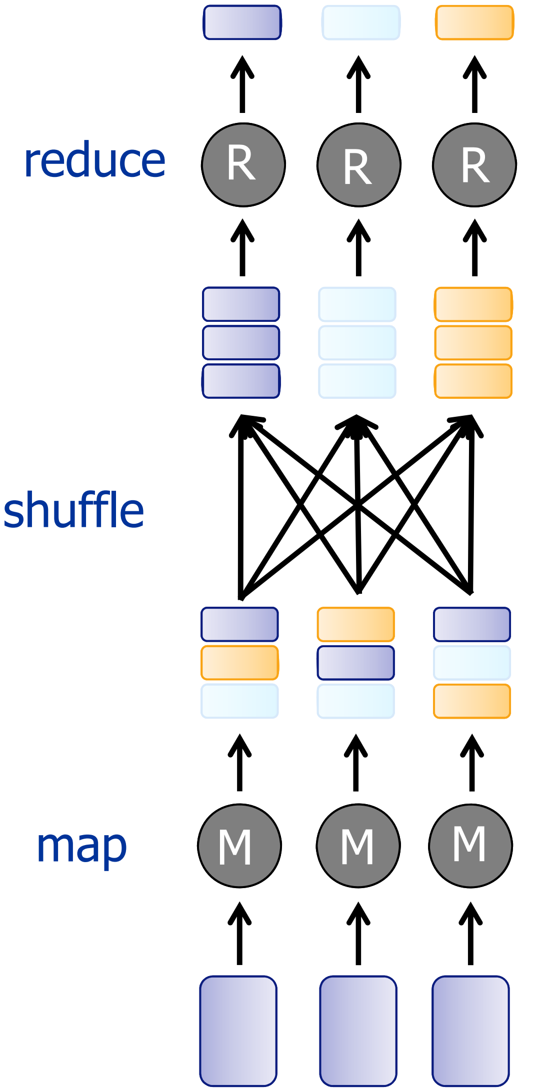
– Active fault-tolerance requires 2x resources

– Passive fault-tolerance leads to long recovery times

Google,
USENIX
OSDI'04

MapReduce: Distributed Dataflow

Sanjay Ghemawat   Jeff Dean



partitioned data on
distributed file system

Data model: (key, value) pairs

Two processing functions:

$map(k_1,v_1) \rightarrow list(k_2,v_2)$

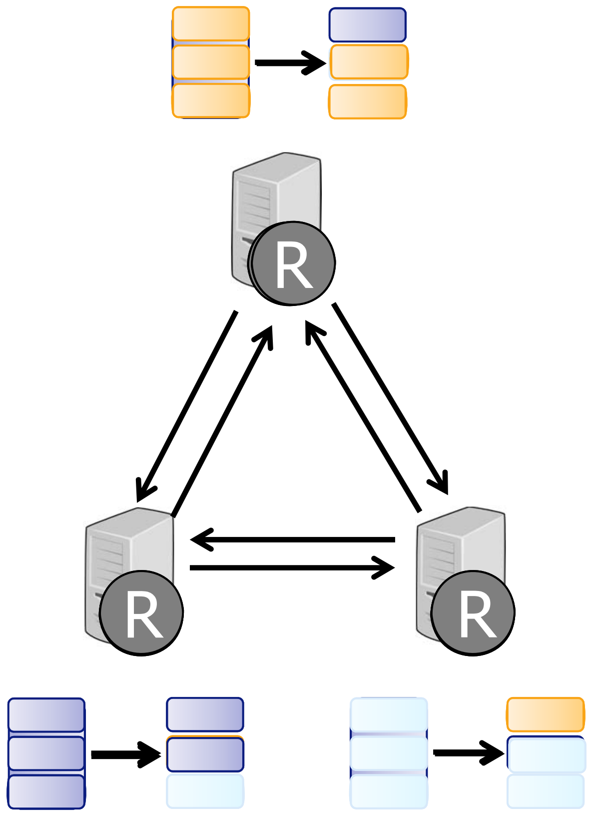
$reduce(k_2, list(v_2)) \rightarrow list\ (v_3)$

Benefits:

– Simple programming model
– Transparent parallelisation
– Fault-tolerant processing

$2 billion market revenue (2013)

MapReduce Execution Model

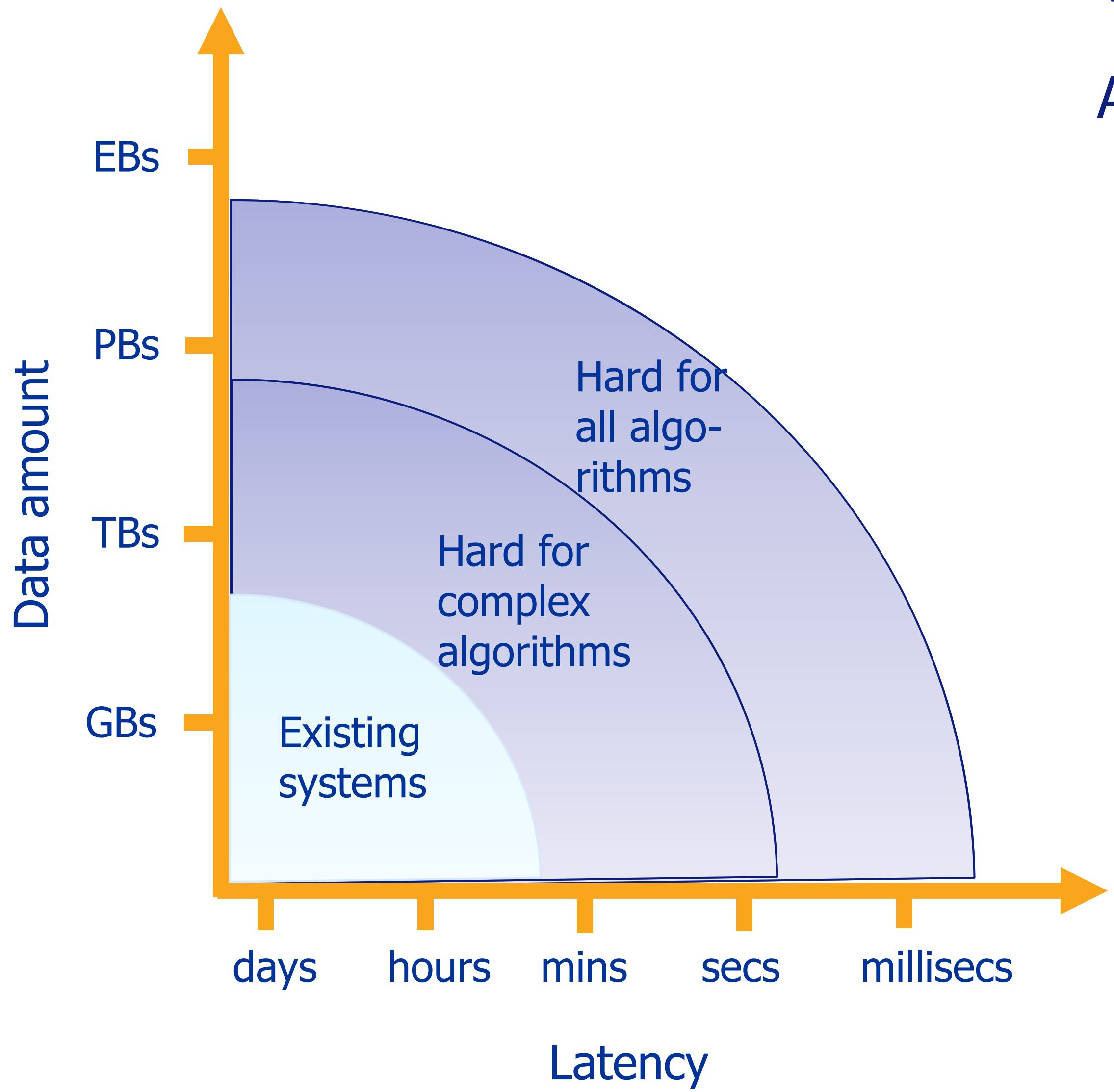


Map/reduce tasks scheduled across cluster nodes

Intermediate results persisted to local disks

- Restart failed tasks on another node
- Distributed file systems contains replicated data

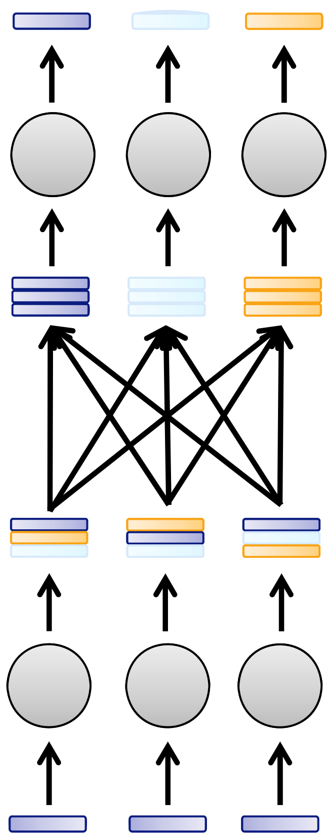
Design Space for Big Data Systems



Volume and Velocity

Algorithmic complexity
- Arbitrary data transformation
- Iterative algorithms
- Large state as part of computation

Spark: Micro-Batching



RDD as discretised stream

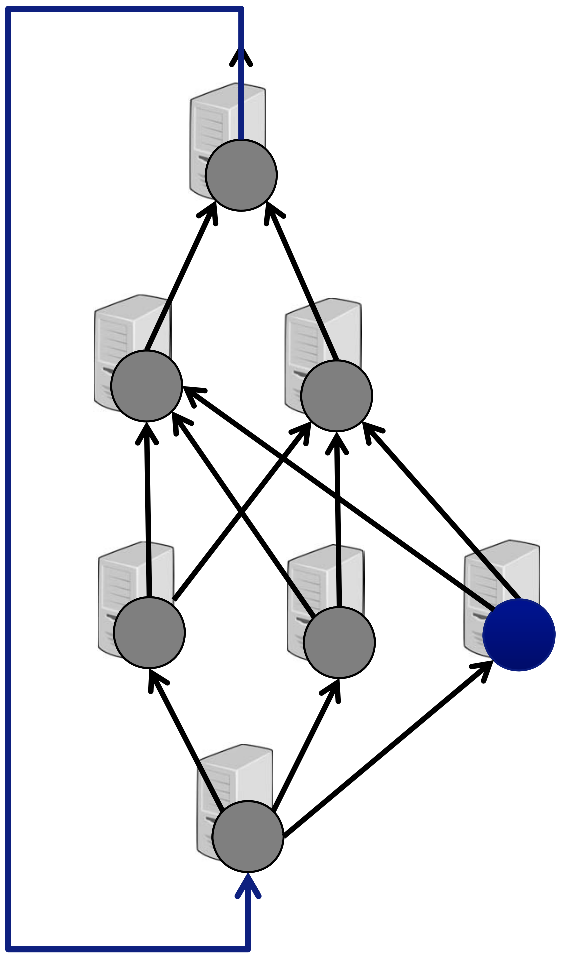
Idea:
Reduce size of data partitons to produce up-to-date, incremental results

Micro-batching for data

- Window-based task semantics
- Parallel recomputation of RDDs

Challenge:
Need to control scheduling overhead

SEEP: Pipelined Dataflows



Idea:
Materialise dataflow graph to avoid scheduling overhead

Challenges:

1. Support for iteration
2. Resource allocation of tasks to nodes
3. Failure recovery

Cycles in graph for iteration

Dynamic scale out of tasks

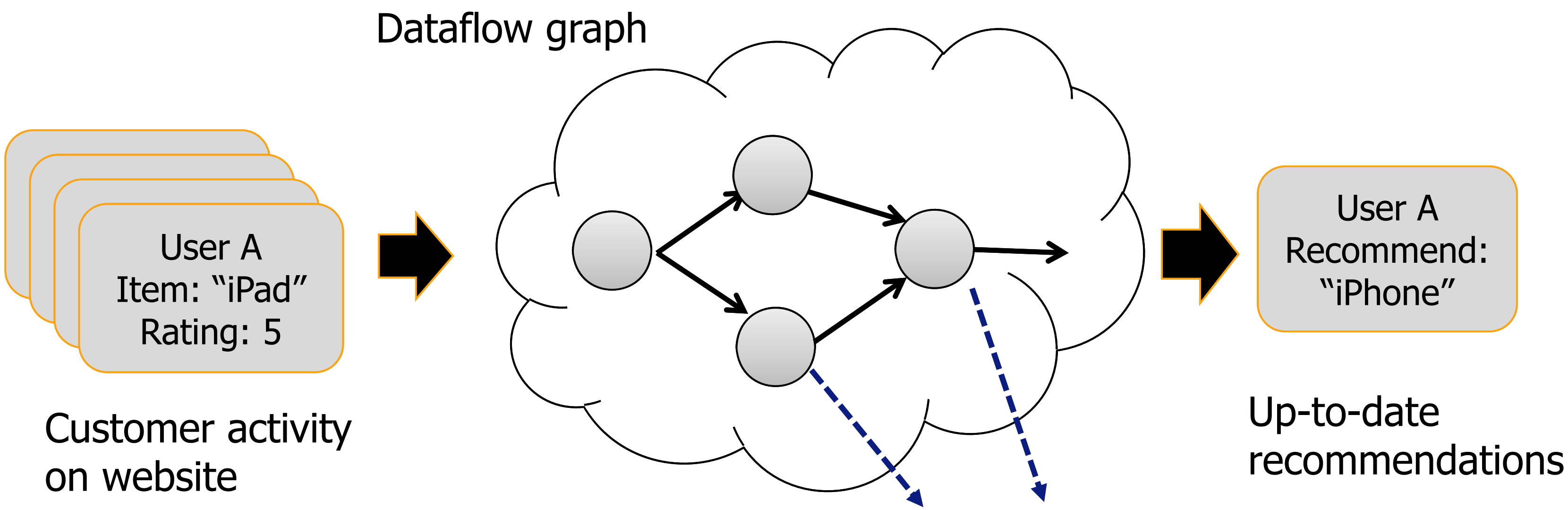
- Identify bottleneck task at runtime
- Transform dataflow graph to parallelise task

Checkpoint-based recovery

- Asynchronous checkpointing of intermediate data to other nodes

What about Processing State?

Online collaborative filtering:



User-item matrix

	Item 1	Item 2
User A	2	5
User B	4	1

GBs to TBs in size

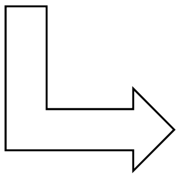
SDG: Imperative Programming Model

```
Matrix userItem = new Matrix();
Matrix coOcc = new Matrix();

void addRating(int user, int item, int rating) {
    userItem.setElement(user, item, rating);
    updateCoOccurrence(coOcc, userItem);
}

Vector getRecommendation(int user) {
    Vector userRow = userItem.getRow(user);
    Vector userRec = coOcc.multiply(userRow);
    return userRec;
}
```

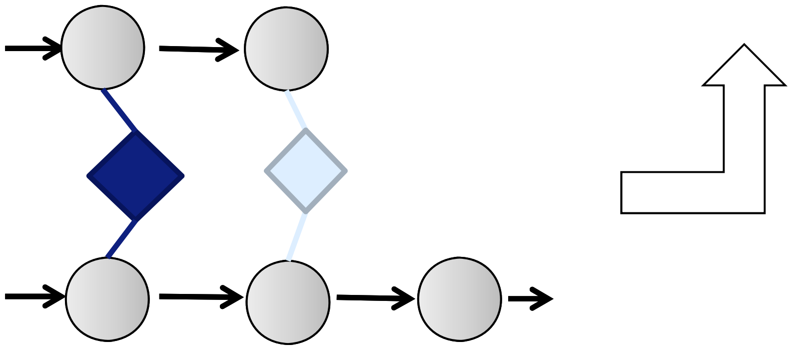
SEEP cluster

Annotated Java program
(@Partitioned, @Partial, @Global, …)

Static program analysis

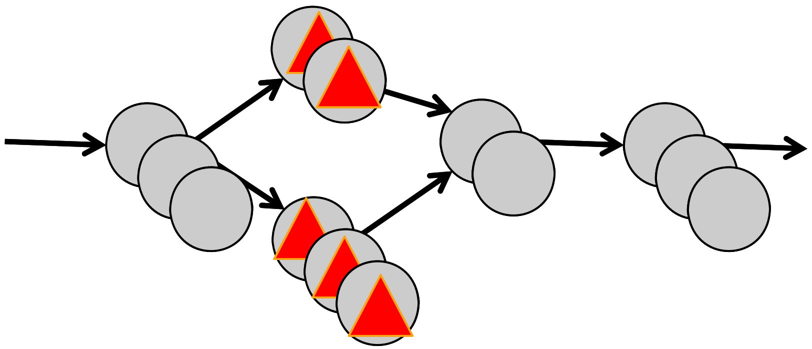


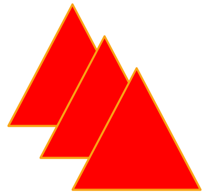
SDG



State Complicates Things…

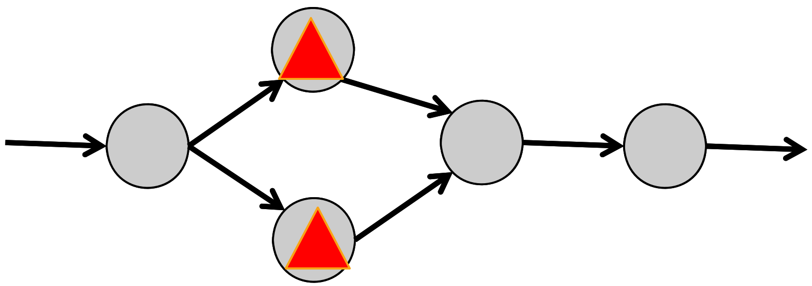
1. Dynamic scale out impacts state

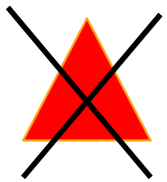




Partitioning of state

2. Recovery from failures





Loss of state after node failure

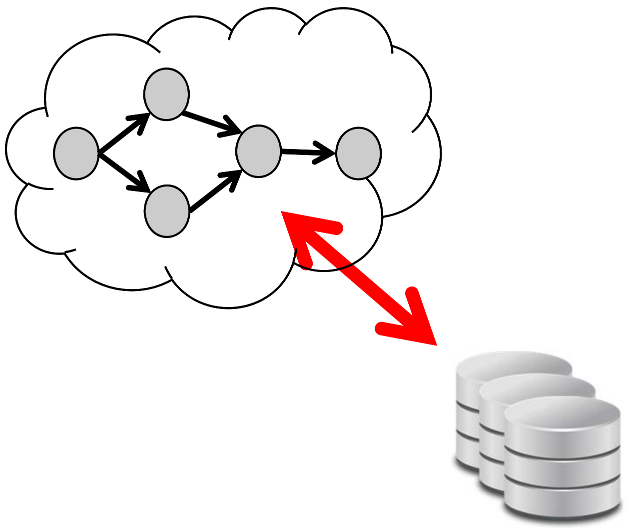
Current Approaches for Stateful Processing

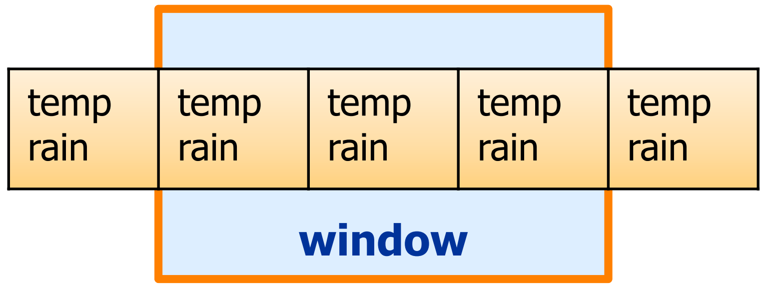
Stateless stream processing systems (eg Yahoo S4, Twitter Storm, …)

- **Developers manage state**
- Typically combine with external system to store state (eg Cassandra)
- Design complexity

Relational stream processing systems (eg Borealis, Stream)

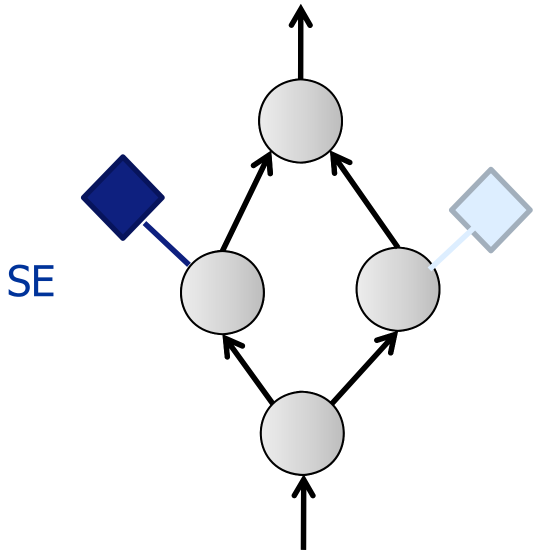
- State is **window** over stream
- No support for arbitrary state
- Hard to realise complex ML algorithms





SDG: Stateful Dataflow Graphs

$$\begin{array}{cc} & \begin{array}{cc}\textbf{Item 1} & \textbf{Item 2}\end{array} \\ \begin{array}{c}\textbf{User A}\\ \textbf{User B}\end{array} & \begin{bmatrix} 2 & 5 \\ 4 & 1 \end{bmatrix}\end{array}$$



Idea:
Add state to dataflow graph

Challenge:
Handing of distributed state

State elements (SEs) represent in-memory data structures

- SEs are mutable
- Tasks have local access to SEs
- SEs can be shared between tasks

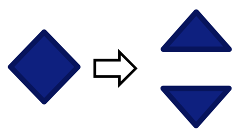
Asynchronous checkpointing for recovery

SDG: Distributed State Elements

SEs can be:

Partitioned SE

Key space: [0-n]



[0-k]

[(k+1)-n]

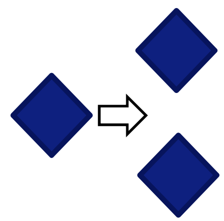
SE can be partitioned according to partitioning key

User-item matrix

Access by key ⇨

	Item 1	Item 2
User A	2	5
User B	4	1

Partial SE

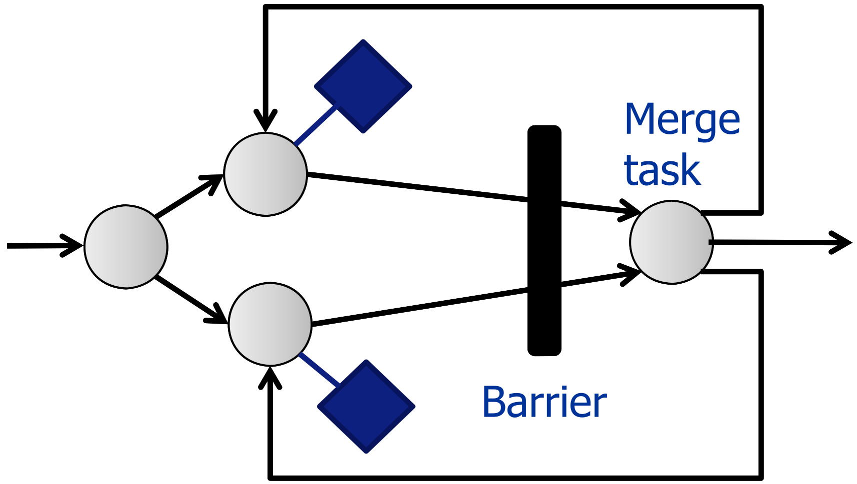


Tasks require global access to SE

– SE cannot be partitioned, but must be replicated

SDGs: State Synchronisation with Partial SEs

Need to synchronise state of partial SEs



Explicit state reconcilation through merge tasks

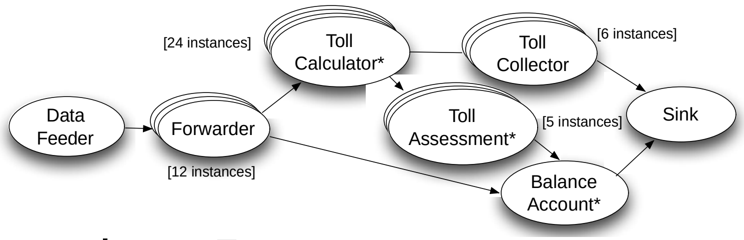
- Barrier collects partial state
- Merge task reconciles state and updates partial SEs

Experimental Evaluation

SEEP: Scalability on Amazon EC2

Linear Road Benchmark [VLDB'04]

- Network of toll roads of size L
- Input rate increases over time
- Dataflow graph with 5 operators; SLA: results < 5 secs

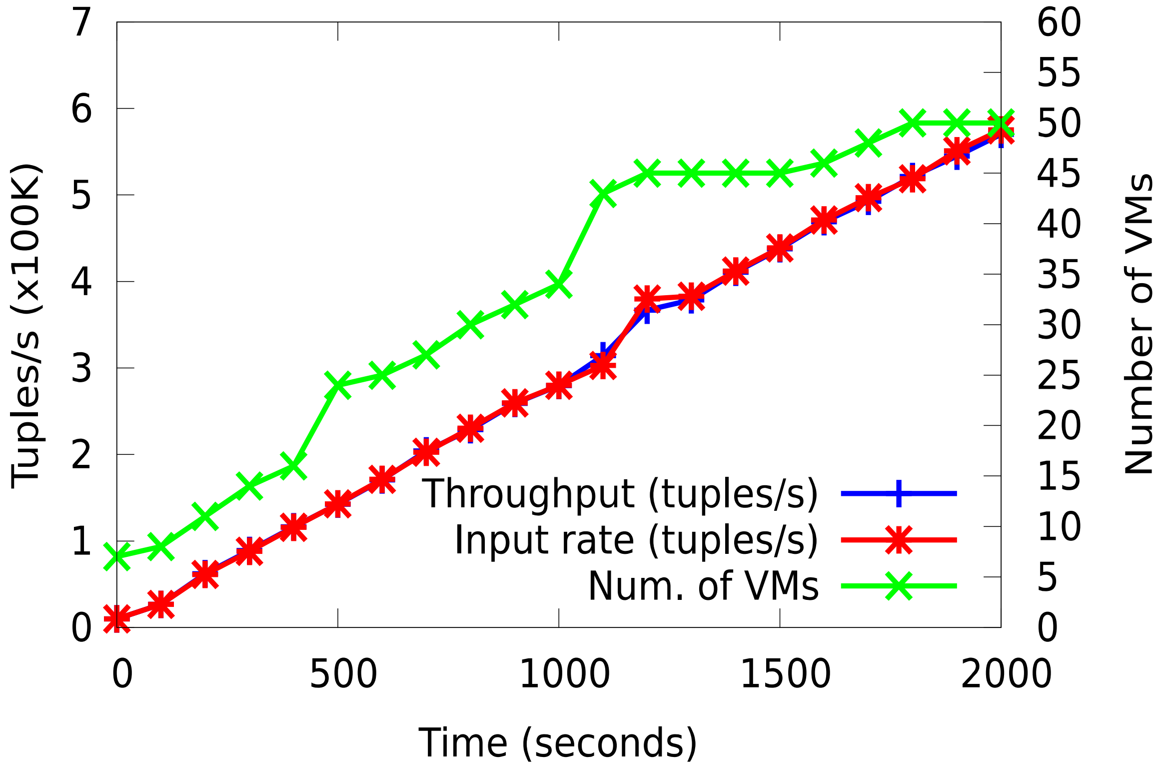


SEEP deployed on Amazon EC2

- Scales to 60 VMs (small instances with 2GB RAM)

Achieves L=350

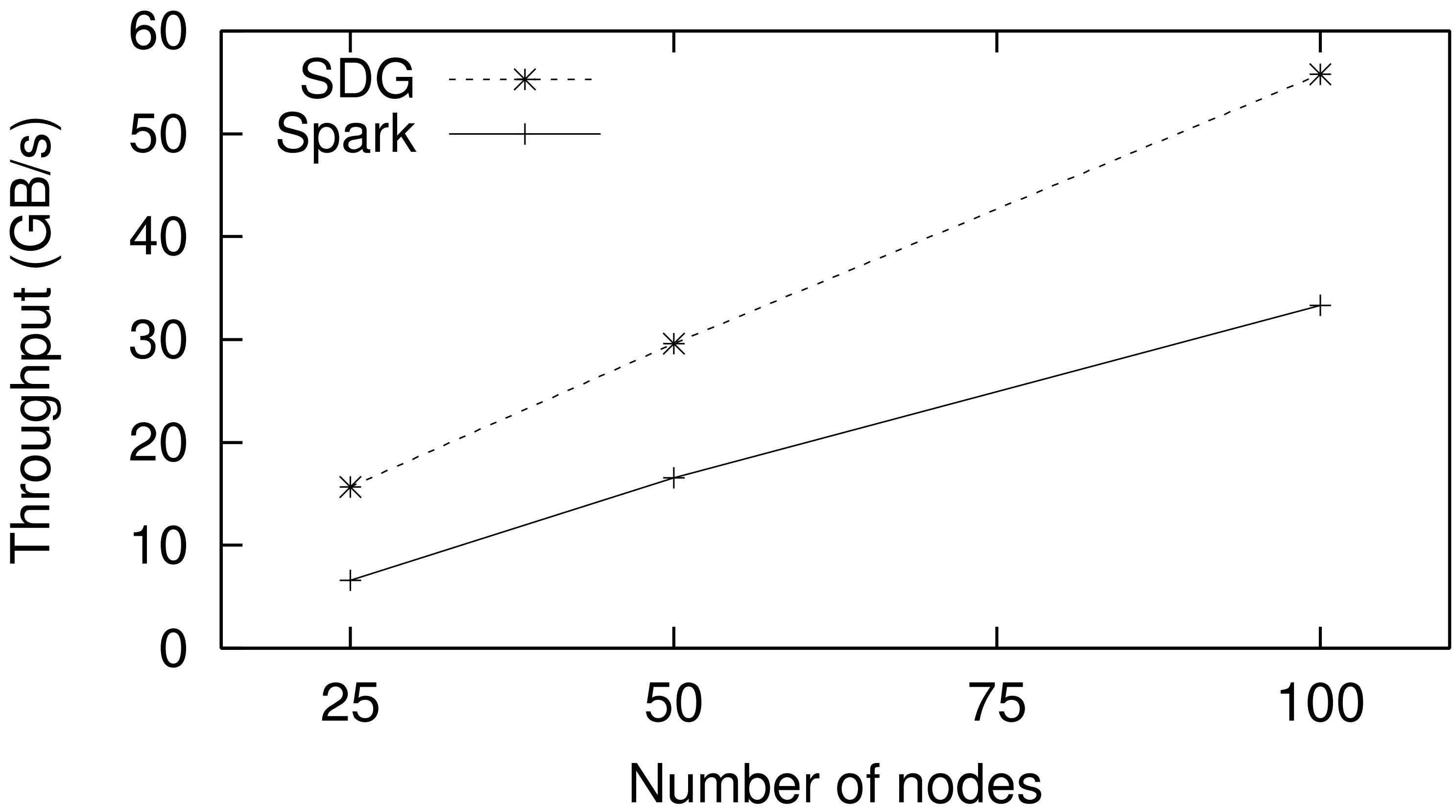
- L=512 highest reported result in literature [VLDB'12]



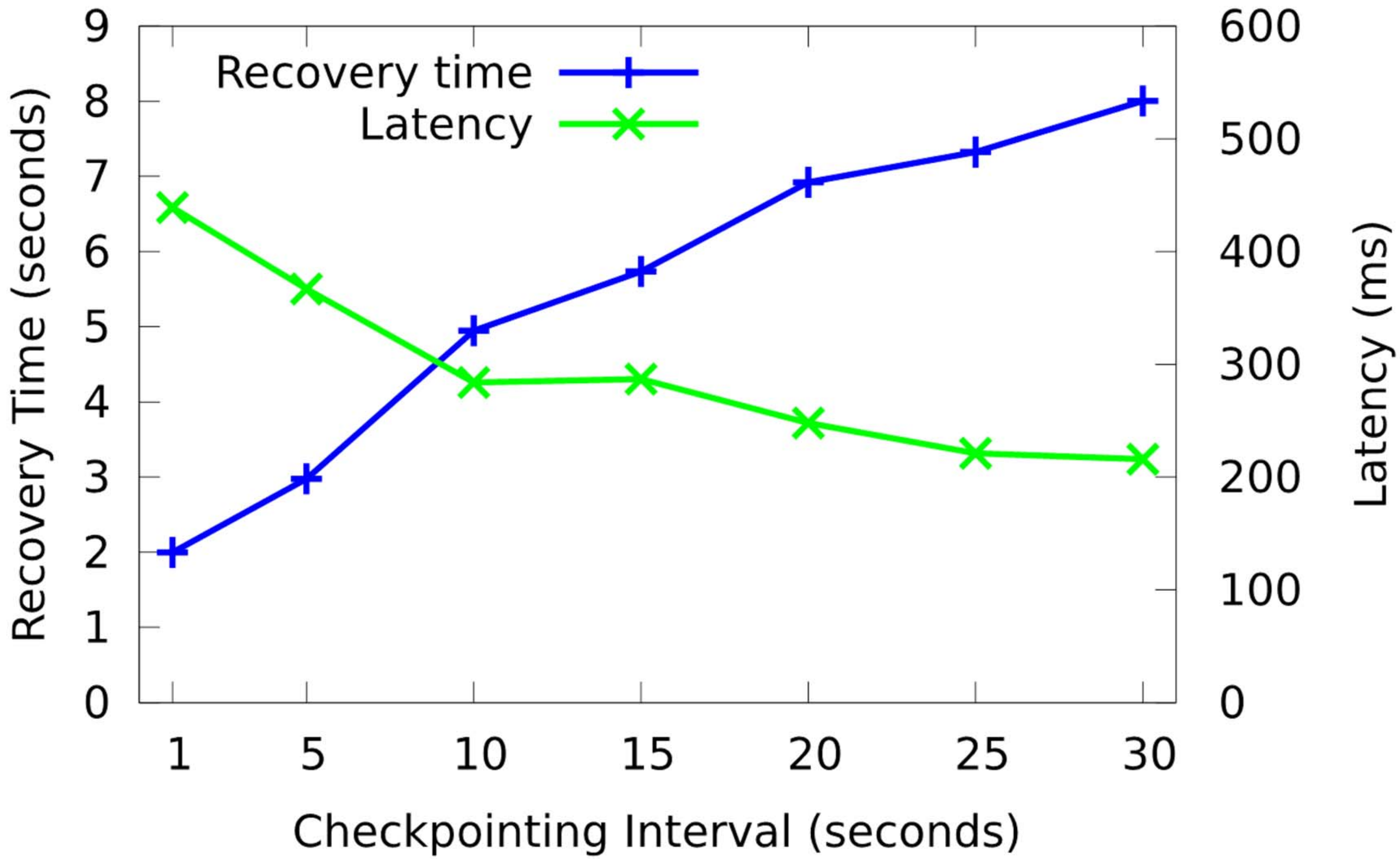
Performance of SEEP

Logistic regression

- Deployed on Amazon EC2 (“m1.xlarge” VMs with 4 vCPUs and 16 GB RAM)
- 100 GB dataset



Overhead of Checkpointing



☛ **Tradeoff between latency and recovery time**

Related Work

Scalable stream processing systems

- **Twitter Storm, Yahoo S4, Nokia Dempsey, Apache Samza**
  Exploit operator parallelism mainly for stateless queries

Distributed dataflow systems

- **MapReduce, Dryad, Spark, Apache Flink, Naiad, SEEP**
  Shared nothing data-parallel processing on clusters

Elasticity in stream processing

- **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

Summary

Stream processing grows in importance

- Handling the data deluge
- Enables real-time response and decision making

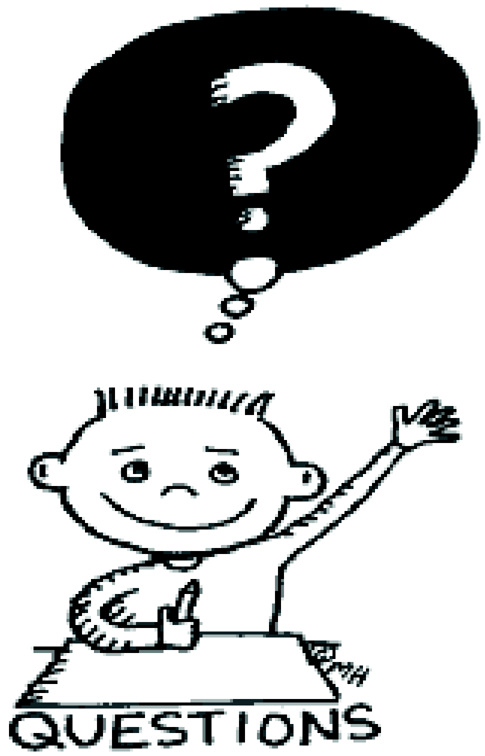
Principled models to express stream processing semantics

- Window-based declarative query languages
- What is the right programming model for machine learning?

Stateful distributed dataflows for stream processing

- High stream rates require data-parallel processing
- Fault-tolerant support for state important for many algorithms
- Convergence of batch and stream processing

Thank You! Any Questions?



Peter Pietzuch
<prp@doc.ic.ac.uk>
http://lsds.doc.ic.ac.uk