Imperial College London

LSDS
Large-Scale Distributed Systems Group

Trends in Scalable Stream Processing:

Parallelism & Programmability

Peter Pietzuch
prp@imperial.ac.uk

Large-Scale Distributed Systems Group
Department of Computing, Imperial College London
http://lsds.doc.ic.ac.uk

EBSIS Summer School 2016 – Sinaia, Romania

Imperial College London

Focus on science, engineering, medicine and business

- Located in South Kensington near Hyde Park
- About 15,000 students

Imperial College London

Who Am I?

Peter Pietzuch

PhD, Distributed Systems group, **University of Cambridge**

Post-doc, Systems Research group, **Harvard University**

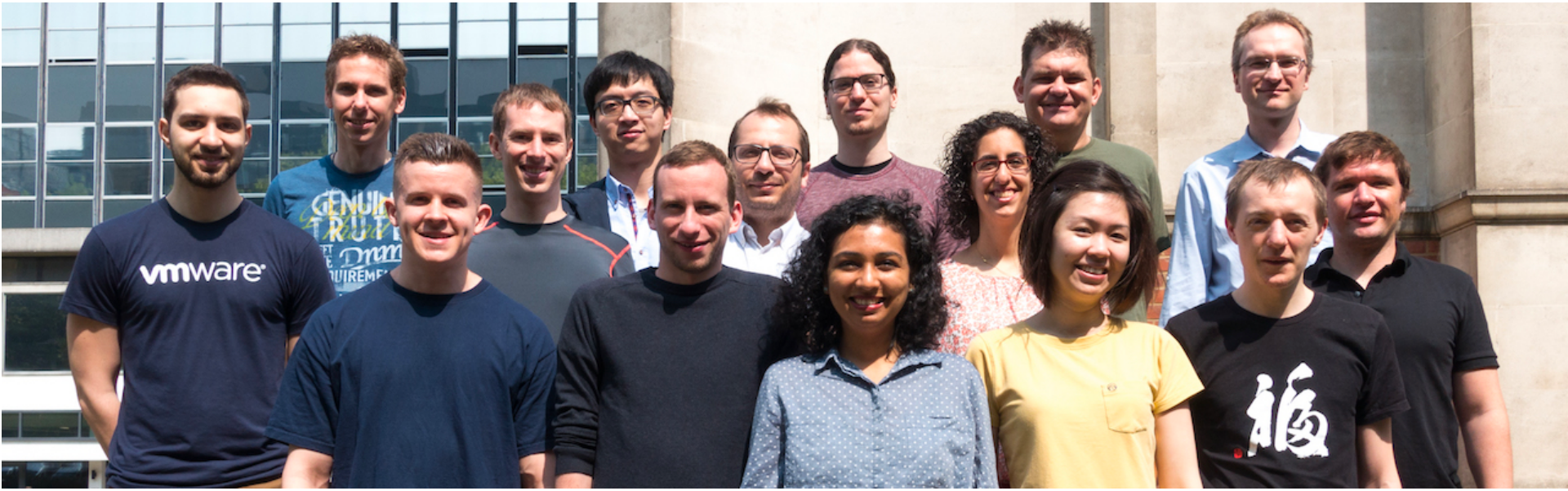
Reader (Associate Professor)
Department of Computing, **Imperial College London**

- Joined Imperial 8 years ago
- Head, Large-Scale Distributed Systems (LSDS) Group

Large-Scale Distributed Systems (LSDS) Group

Currently 15 members
(8 post-docs, 7 PhD students)

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

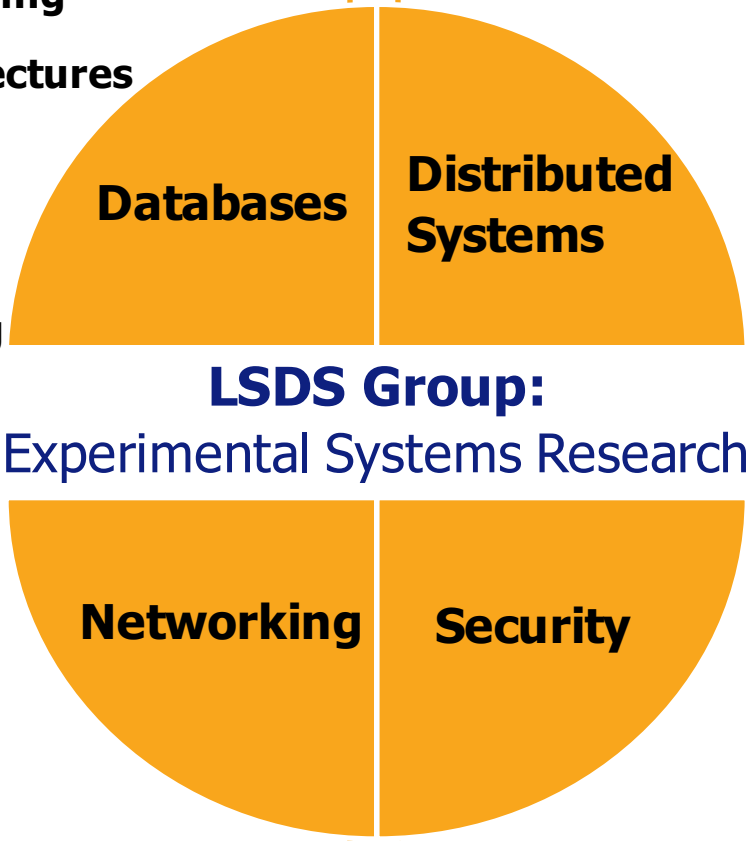


LSDS Mission Statement:
"To support the design and engineering
of scalable, robust and secure distributed applications"

Past and Present LSDS Research

- **Distributed dataflow systems** [ICDE'16, ATC'14, SIGMOD'13]
- **Multicore data processing** [SIGMOD'16, VLDB'14]
- **Heterogeneous architectures** [SIGMOD'16]
- **Stream processing** [SIGMOD'16, CIDR'15, ICDE'11]
- **Scalable machine learning**
- Complex event processing

- Cloud computing
- Multi-data-centre support
- **Edge computing** [TMC, MobiSys'15]
- **Resource allocation**
- Data centre management
- Publish/subscribe systems
- Middleware



LSDS Group:
Experimental Systems Research

- Data centre networking
- **In-network processing** [CoNEXT'14, ATC'16]
- **Protocol checking** [TSE'14]
- MANET data processing
- Overlay networks
- Content distribution

- **Cloud security** [CCS'15, ESORICS'16]
- **Information Flow Control** [ICDE'14, Middleware'11, ATC'10]
- **Web vulnerabilities** [WebApps'11]
- **Hardware trust**
- Browser security

Event Data Is Everywhere

More data created than ever

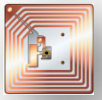
- Generated **2.5 Exabytes** (billion GBs) each day in 2015

Many new **sources of event data** become available


Internet services, web sites


Social feeds


IoT devices


RFID tags


Cameras


Mobile devices


Scientific instruments


Data repositories

Storage and networking costs become cheaper

- **Hard drive cost per GB** dropped from **$8.93** (2000) to **$0.03** (2014)

☛ Many applications want to exploit these events in real-time…

Intelligent Urban Transport

Instrumentation of urban transport

- Induction loops to measure traffic flow
- Video surveillance of hot spots
- Sensors in public transport

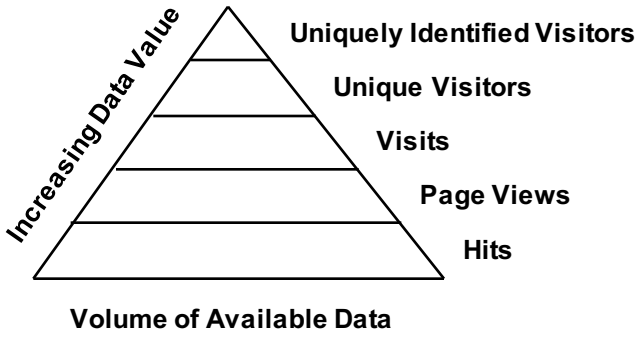
Potential queries

- How to detect traffic congestion and road closures?
- How to explain the cause of congestion (public event, emergency)?
- How to react accordingly (eg by adapting traffic light schedules)?

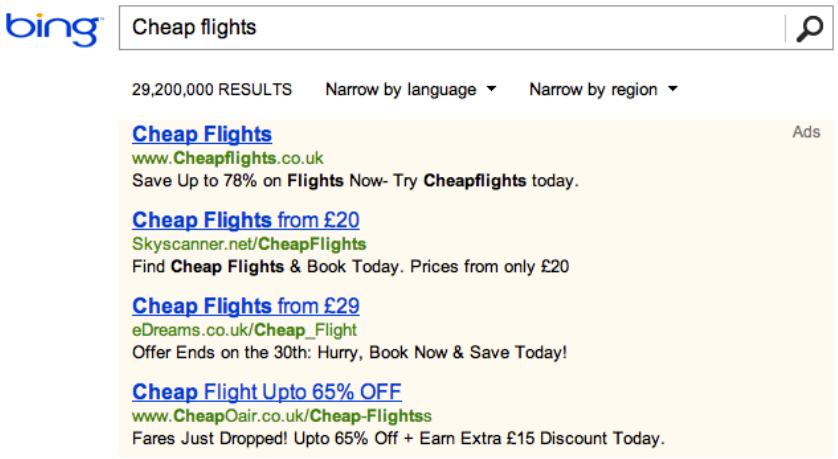
Real-Time Web Analytics

Potential queries

- How to uniquely identify web site visitors?
- How to maximize user experience with relevant content?
- How to analyse "click paths" to trace most common user routes?

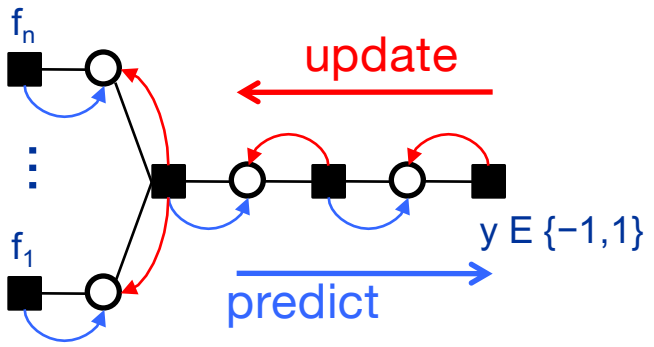


Example: **Online predictions for adverts** to serve on search engines

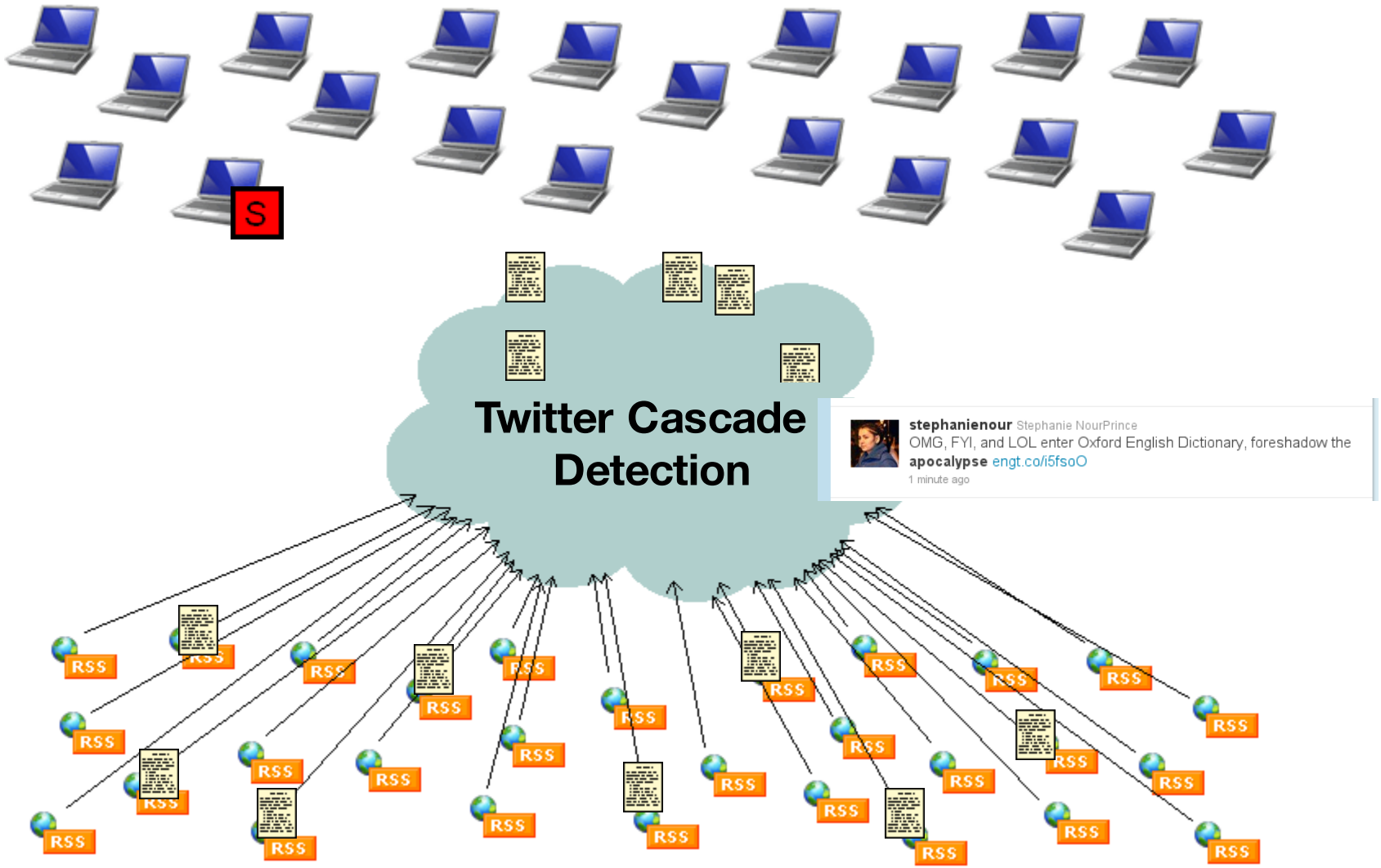


Solution: **AdPredictor**

- Bayesian learning algorithm ranks ads according to click probabilities



Social Data Mining

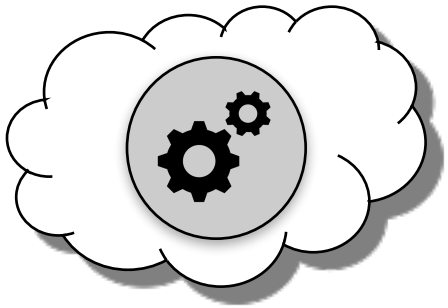


Applications Follow An Event-Based Model

Event producers

Loose coupling
(results in scalable design)

Event streams

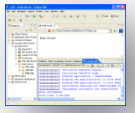


Event/stream processing system

Subscriptions, Queries

Results

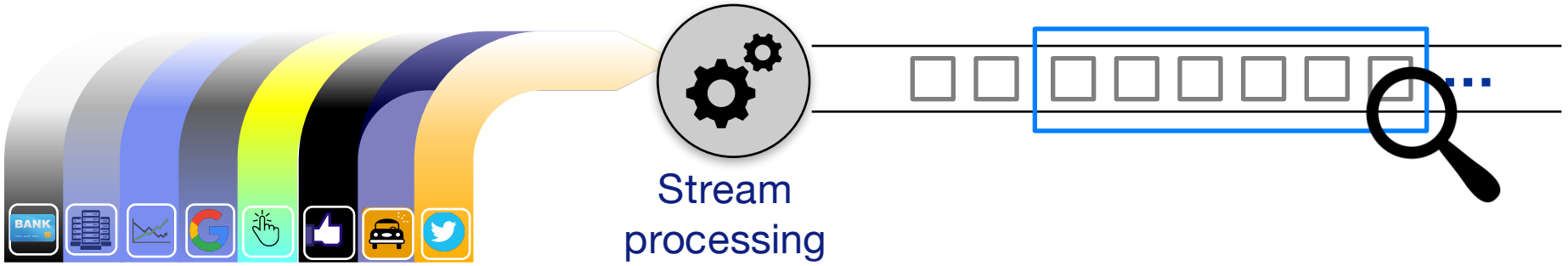
Users



Applications

Event consumers

Challenge 1: Performance Matters!

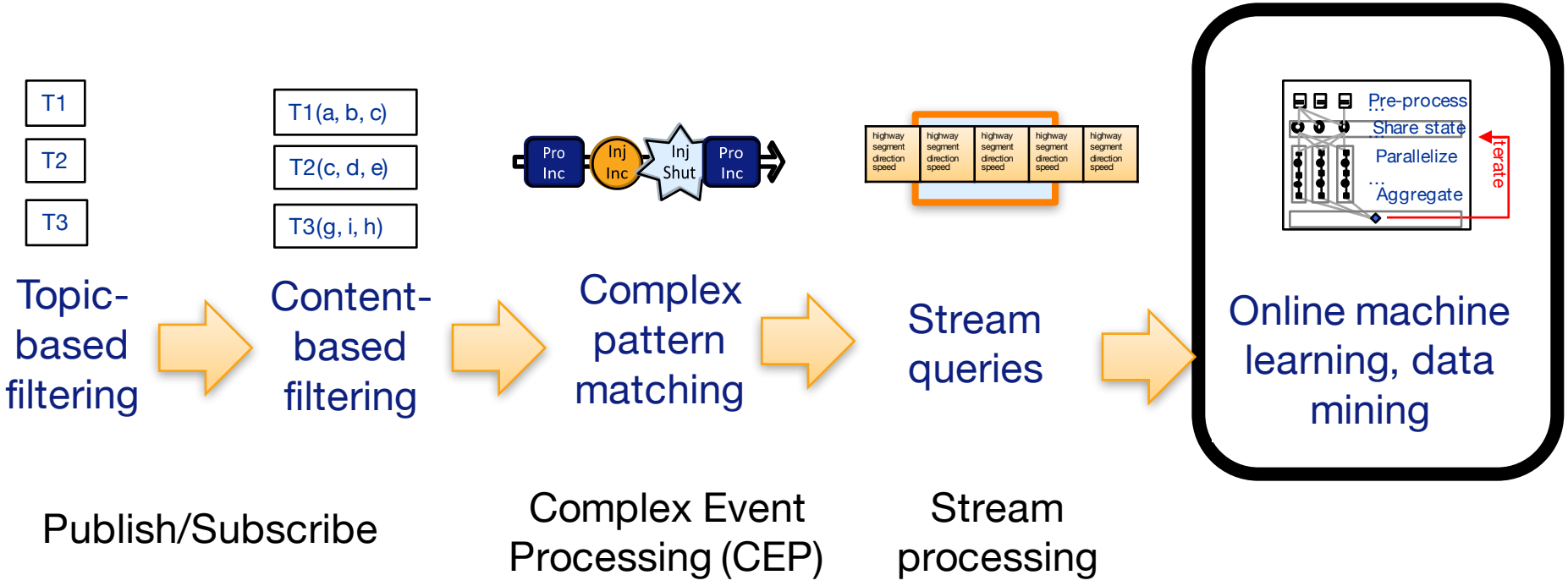


High-throughput streams

Low-latency results

Facebook Insights:	Aggregates 9 GB/s	< 10 sec latency
Feedzai:	40K credit card transactions/s	< 25 ms latency
Google Zeitgeist:	40K user queries/s (1 sec windows)	< 1 ms latency
NovaSparks:	150M trade options/s	< 1 ms latency

Challenge 2: Programmability Matters!



Roadmap

Introduction to Stream Processing Systems

Challenge 1: Performance
How to exploit **parallelism on modern hardware** independently of processing semantics?

Challenge 2: Programmability
How to support **online machine learning algorithms** over stream data?

Conclusions

What Is An Event?

An **event** is a happening of interests. An **event type** is a specification of a set of events of the same structure and semantics.
[Etzion and Niblett (2011)]

Events can have fixed **relational schema**

- Payload of event is a set of attributes

highway = M25
segment = 42
direction = north
speed = 85

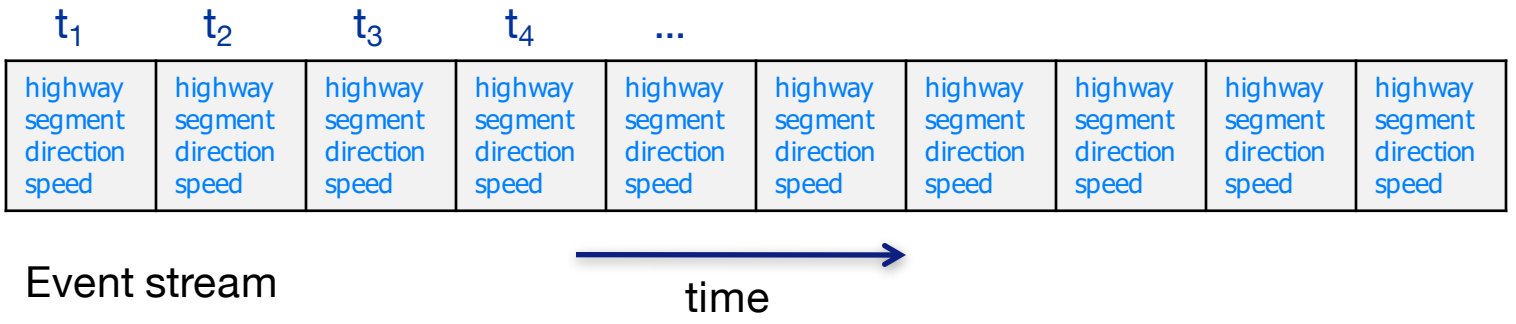
Vehicle speed data

Vehicles(highway, segment, direction, speed)

What Is An Event Stream?

Event stream is an **infinite sequence of event tuples**

– Assume associated **timestamp** (eg time of reading, time of arrival, …)

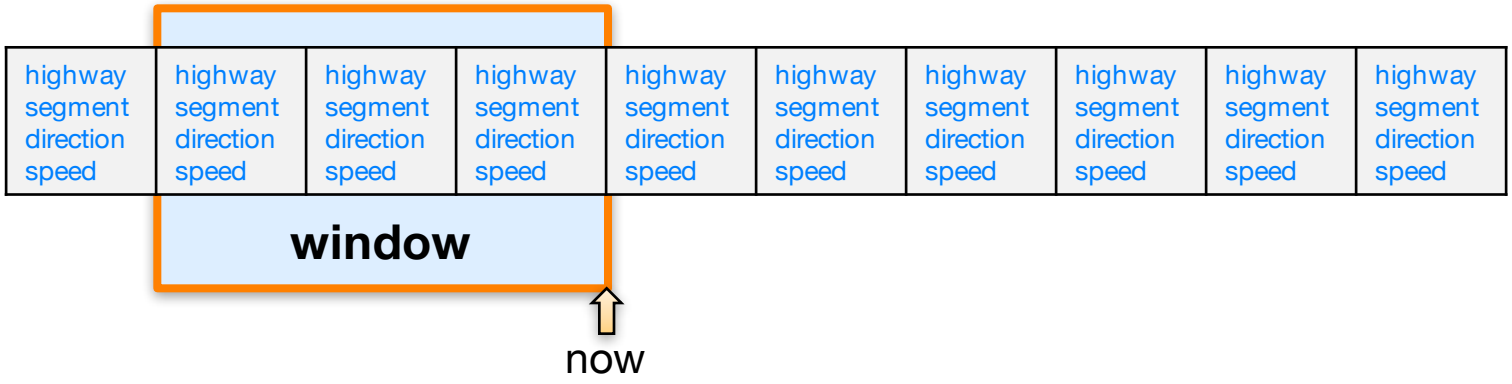


☛ But we have an infinite amount of data to process…

How Many Tuples To Process?

Windows defined **finite set of tuples** for processing

- Process events in window-sized batches



Time-based window with size τ at current time t

[t - τ : t] **Vehicles[Range τ seconds]**

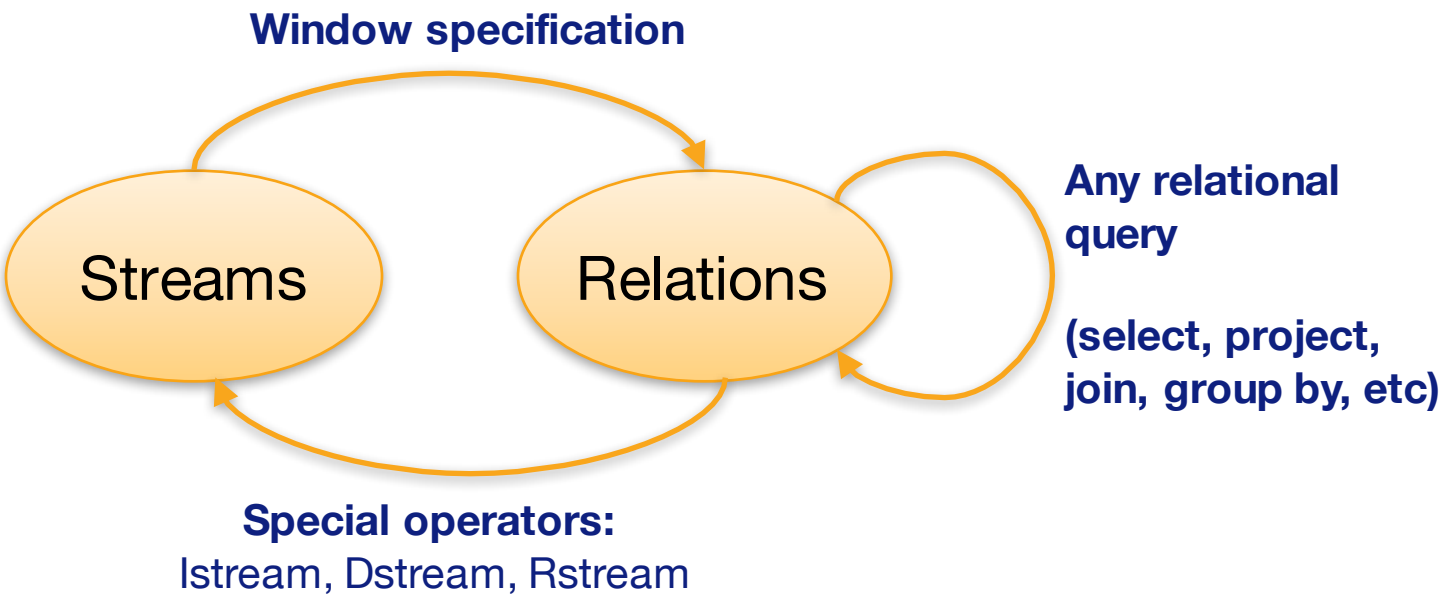
Count-based window with size n:

last n tuples **Vehicles[Rows n]**

How To Define Event Queries?

Window converts **event stream** to **dynamic relation** (database table)

- Similar to maintaining **database view**
- Use regular relational algebra operators on tuples



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

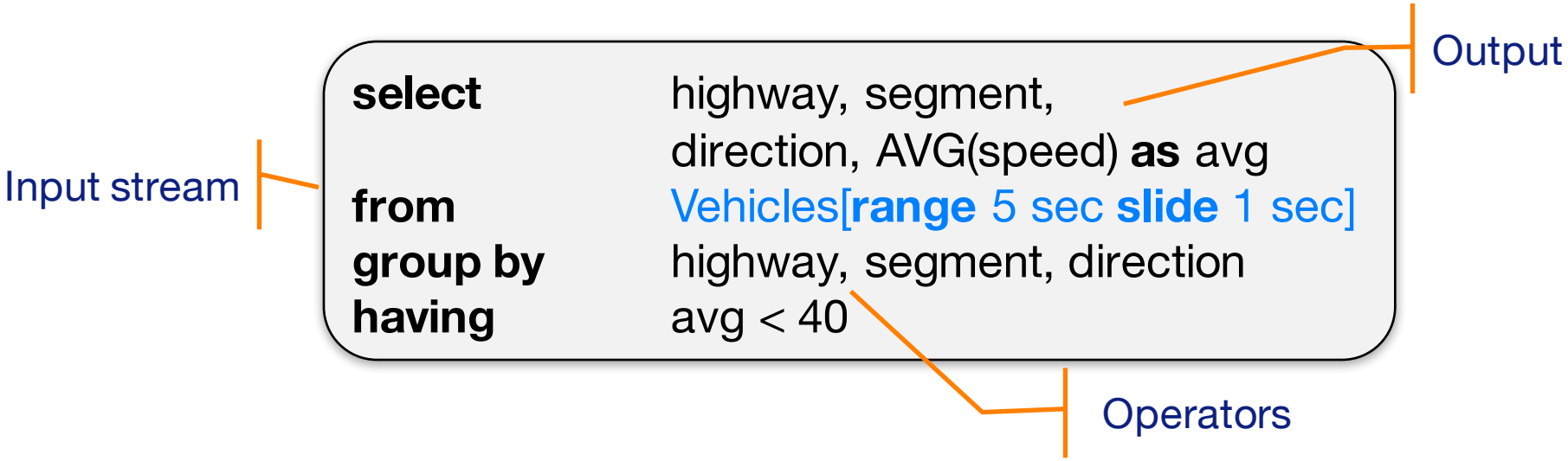
CQL: SQL-Based Declarative Queries

CQL provides well-defined semantics for stream queries

- Based on well-defined relational algebra (select, project, join, …)

Example: Identify slow moving traffic on highway

- Find highway segments with average speed below 40 km/h



☛ Principled way to define stream processing semantics…

Roadmap

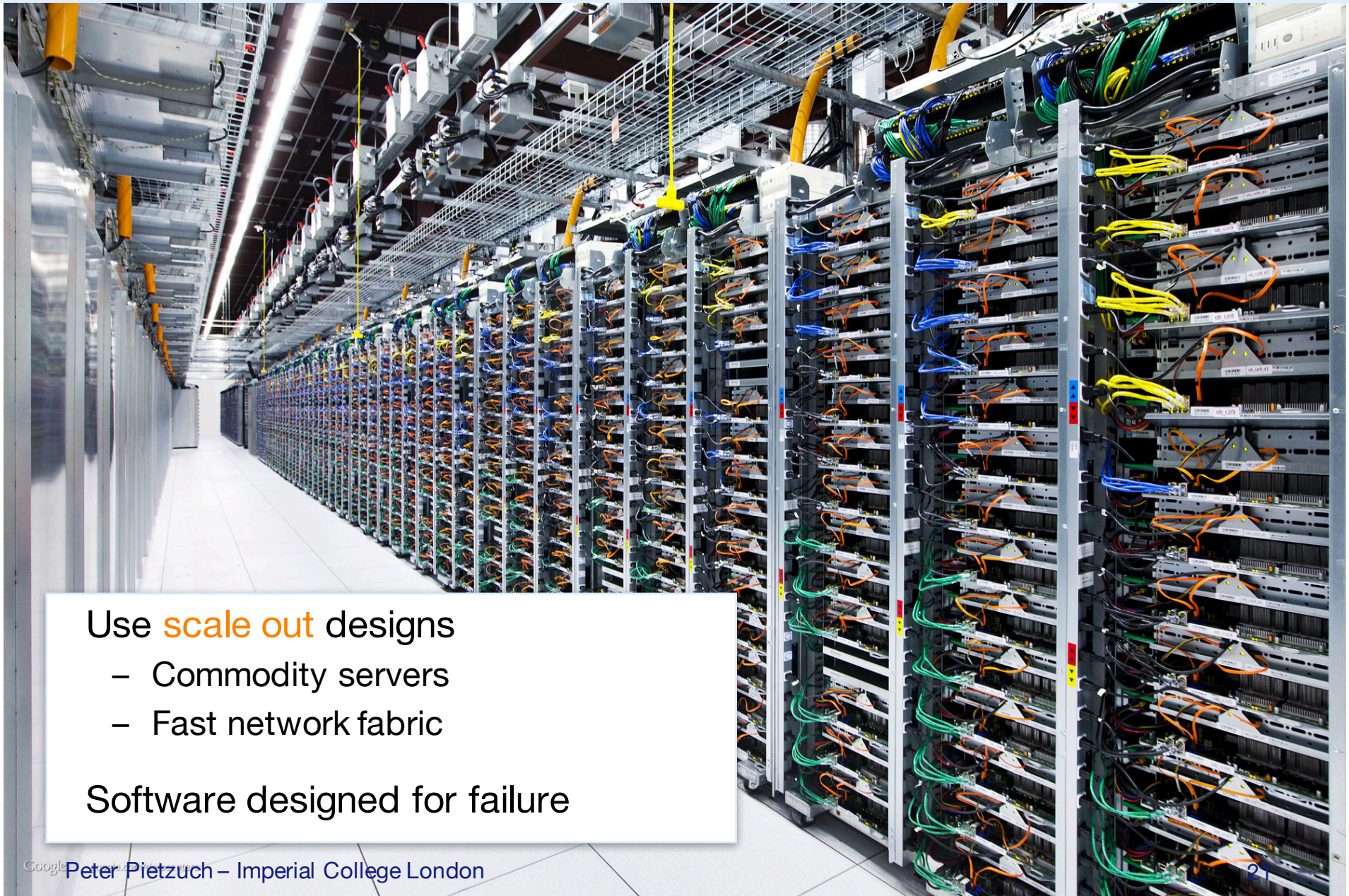
Introduction to Stream Processing Systems

Challenge 1: Performance
How to exploit **parallelism on modern hardware** independently of processing semantics?

Challenge 2: Programmability
How to support **online machine learning algorithms** over stream data?

Conclusions

How To Scale Big Data Systems?



Use scale out designs

- Commodity servers
- Fast network fabric

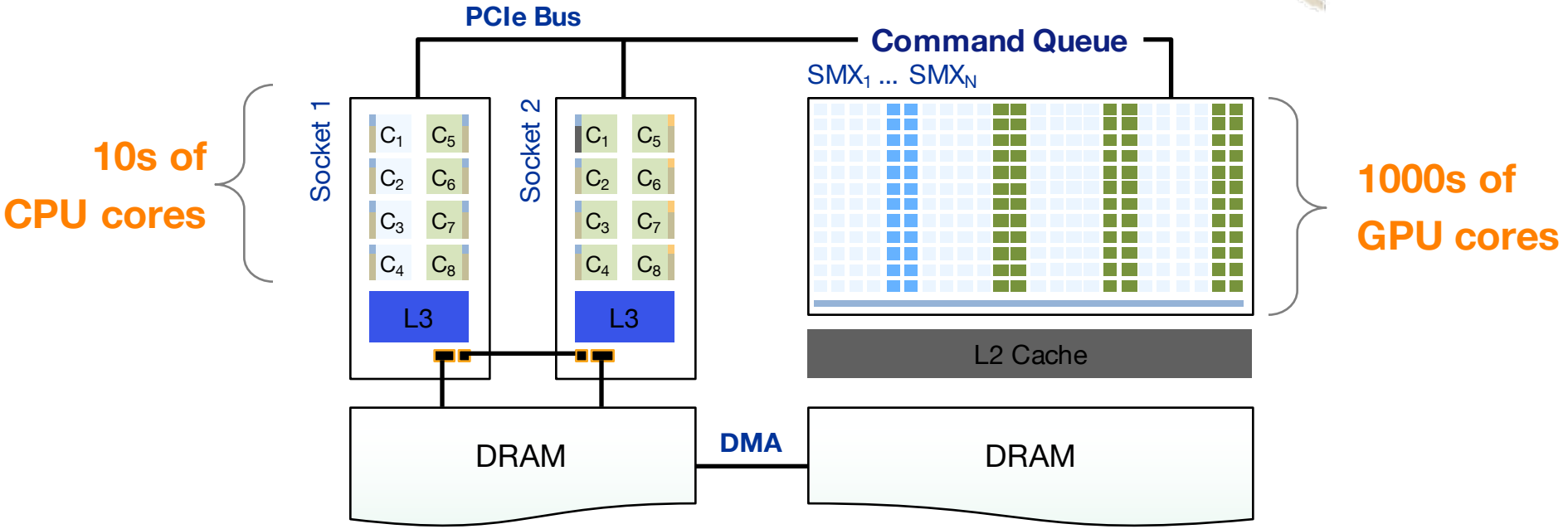
Software designed for failure

But Must Also Exploit Parallel Hardware

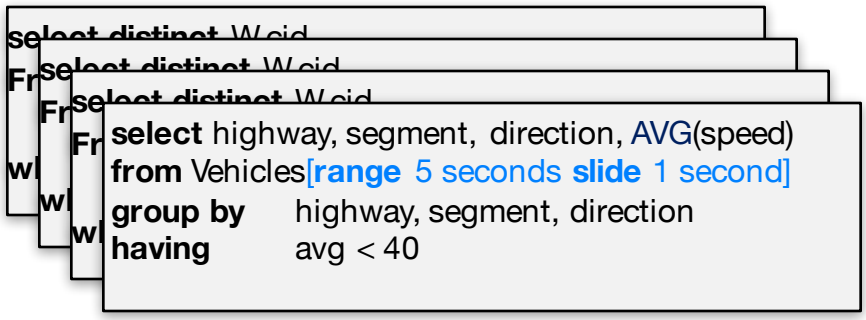
Servers have many **parallel CPU cores**

Servers with **GPUs** common

- GPU have even more specialised cores



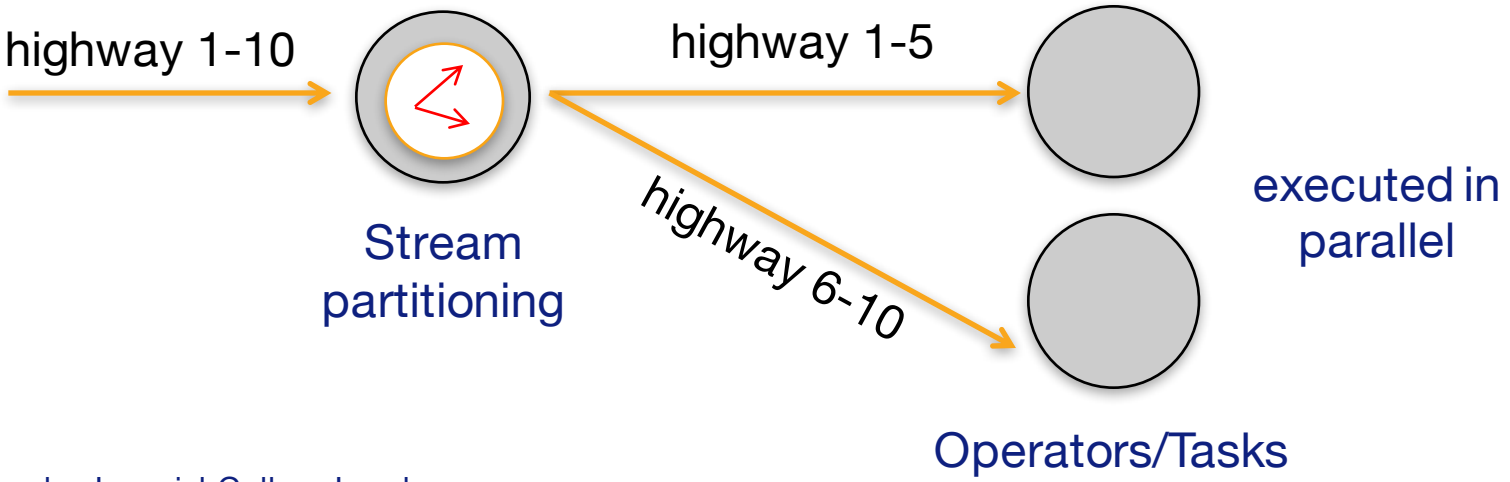
Task Parallelism Vs Data Parallelism



Task parallelism:
Multiple queries

```
select highway, segment, direction, AVG(speed)
from Vehicles[range 5 seconds slide 1 second]
group by     highway, segment, direction
having       avg < 40
```

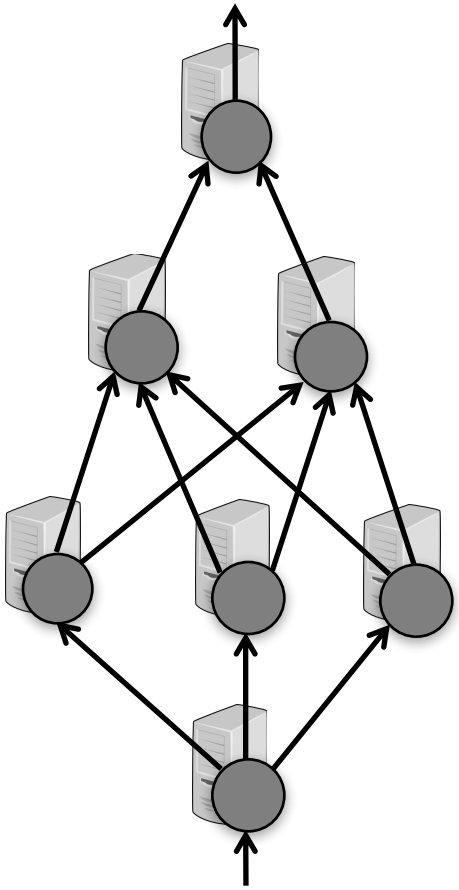
Data parallelism:
Single query



Apache Storm: Dataflow Graphs

parallelism degree 2

parallelism degree 3



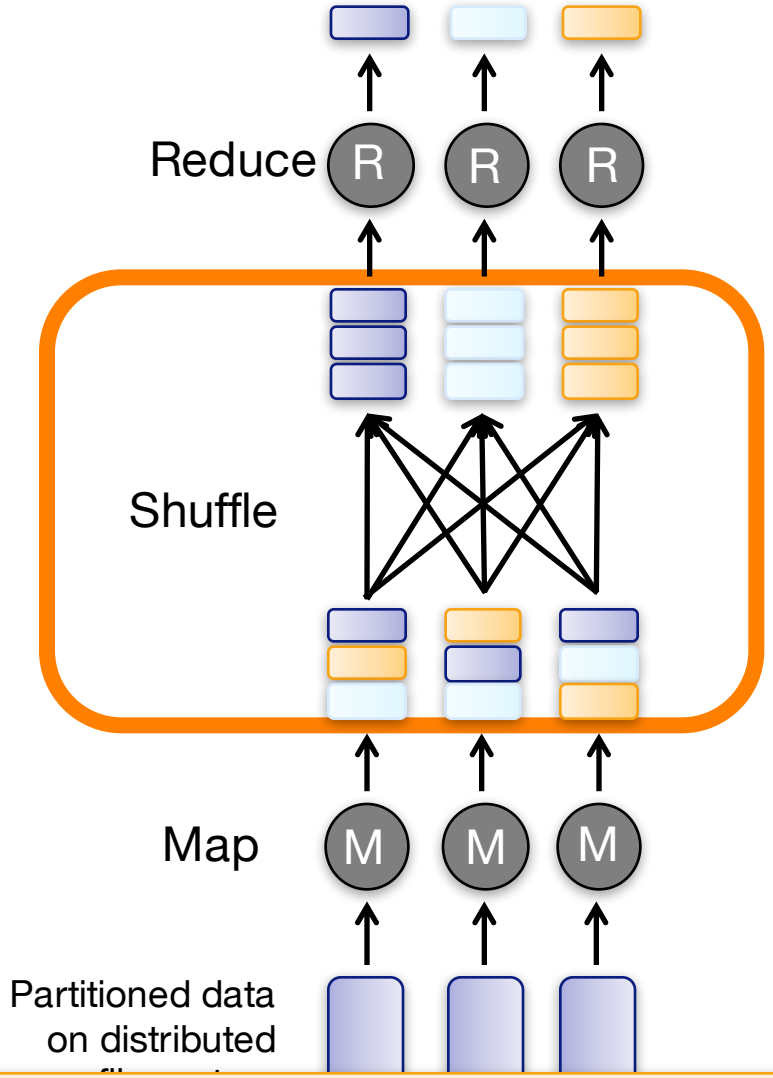
Idea:
Execute event operators as data-parallel **tasks**

Task organised as **dataflow graph**

Many systems do this, e.g. Apache Storm, Apache Flink, Google Dataflow, …

☛ But must manually assign tasks to nodes…

Use Apache Hadoop For Stream Processing?



MapReduce model

- Data model: (key, value) pairs
- Two processing functions:

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

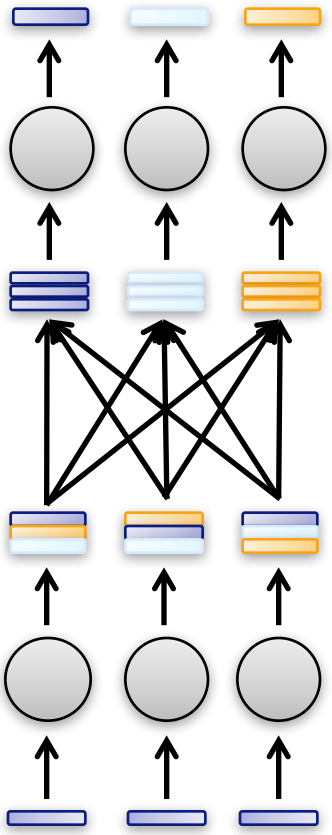
$reduce(k_2, list(v_2)) \rightarrow list\ (v_3)$

Benefits

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

☛ Shuffle phase introduces synchronisation barrier (batch processing)

Apache Spark: Micro-Batching



Stream, divided into micro-batches

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

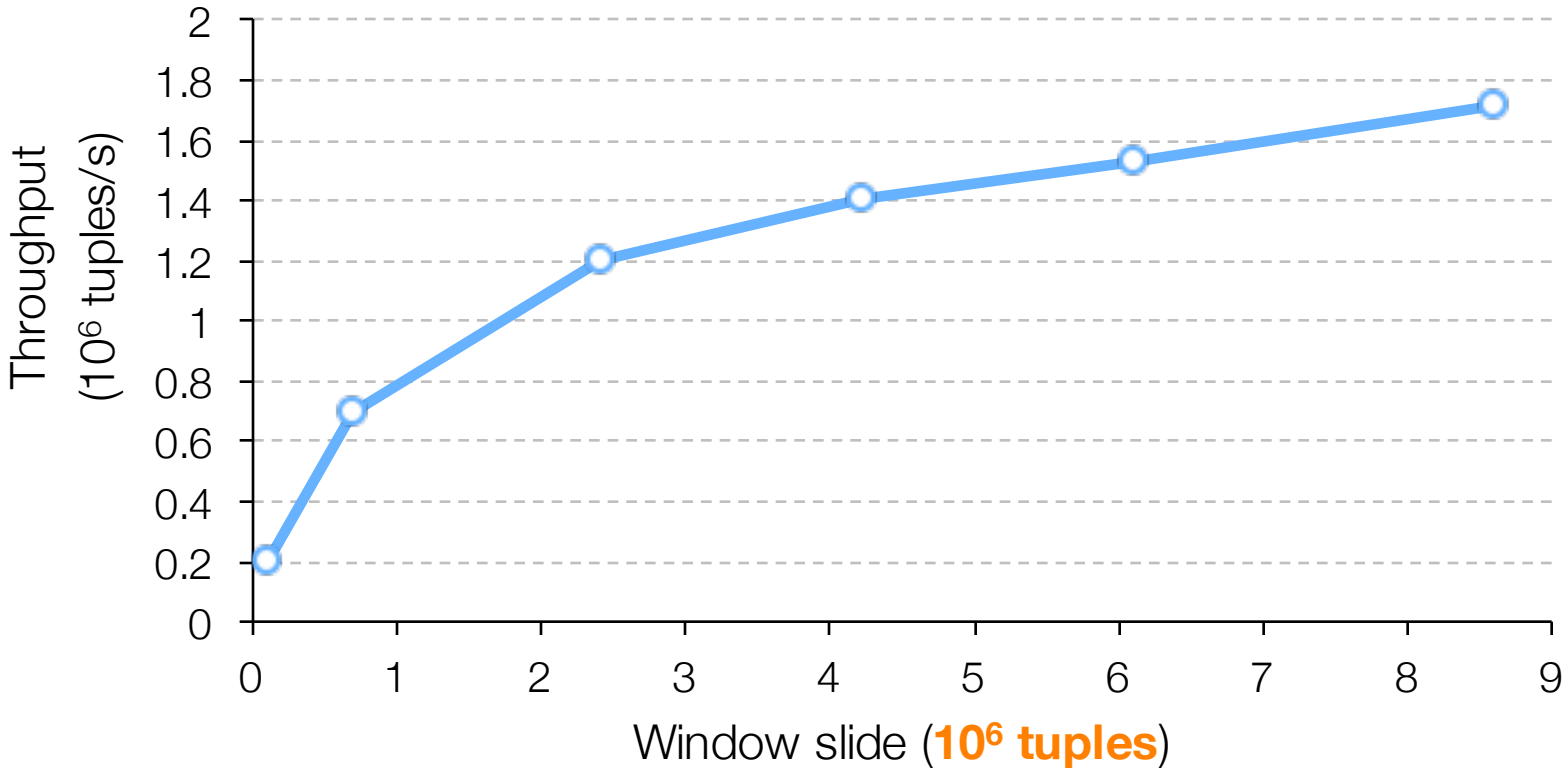
Micro-batching for stream data

- Tasks operate on micro-batch partitions
- Results produced with low latency

☛ Interaction of query windows and micro-batches?

Spark: Small Slides Result In Low Throughout

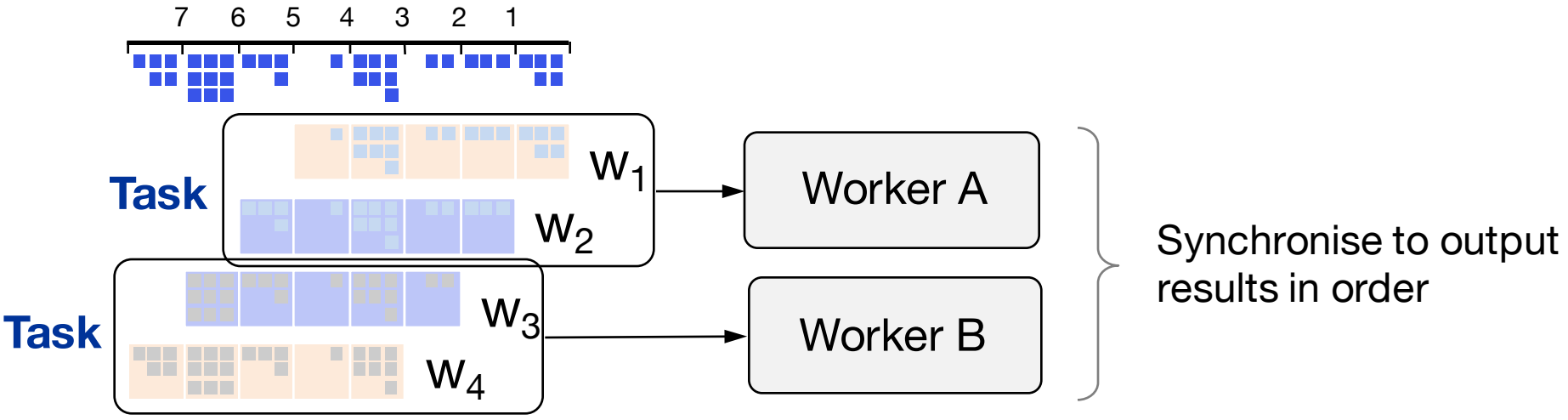
select AVG(S.1) **from** S [**rows** 1024 **slide** *x*]



☛ Want to avoid coupling performance with query definition

How To Parallelise Sliding Windows?

```
select      highway, segment, direction, AVG(speed) as avg
from        Vehicles[range 5 seconds slide 1 second]
group by    highway, segment, direction
having      avg < 40
```

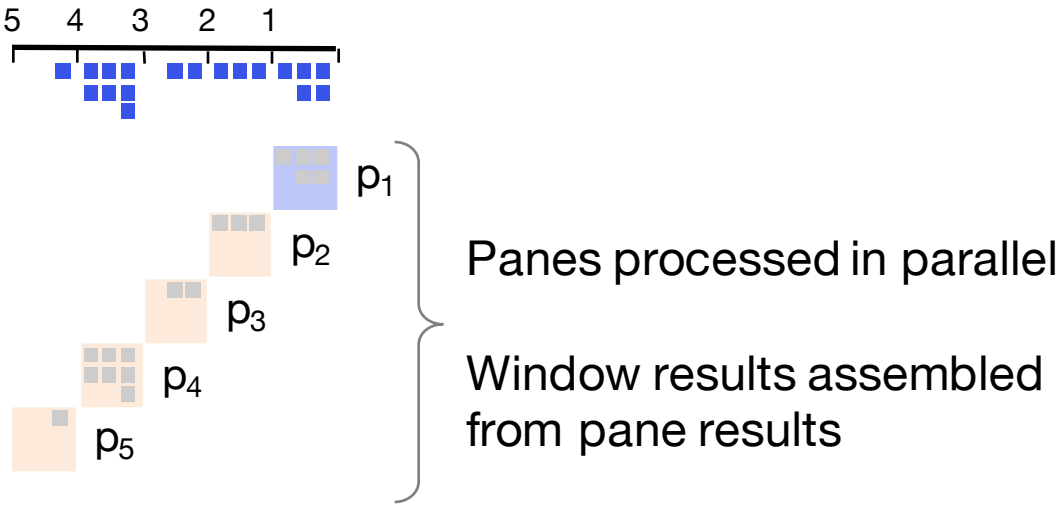


☛ Leads to **redundant computation**

Avoiding Redundant Computation

Use **panes** to remove window overlap between tasks

- Smallest unit of parallelism without data dependencies between windows



Apache Spark uses **panes** for **micro-batches** with windowed queries

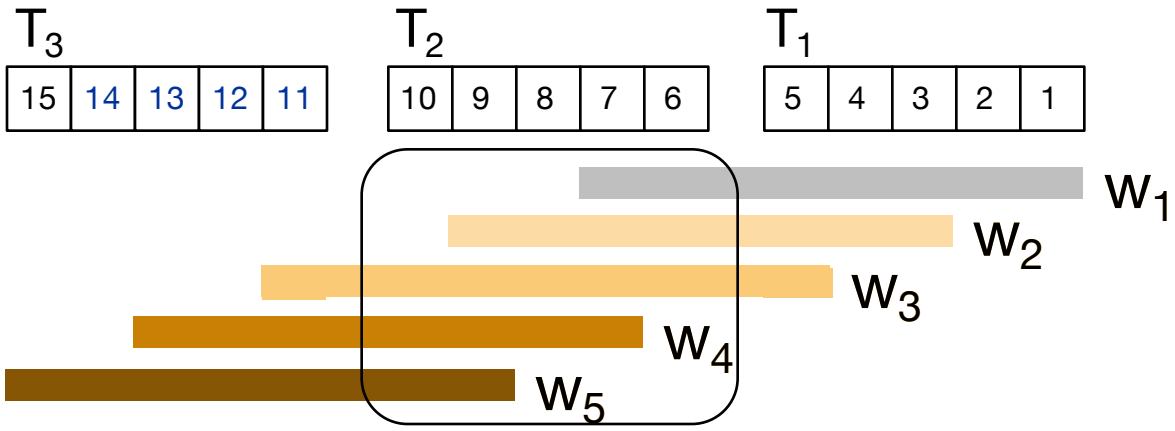
- **Micro-batch size** limited by pane size

- **Window slide** limited by minimum micro-batch size (~500 ms)

SABER: Window Fragment Model

Idea: **Decouple task size from window size/slide**

- e.g. 5 tuples/task, window size 7 rows, slide 2 rows



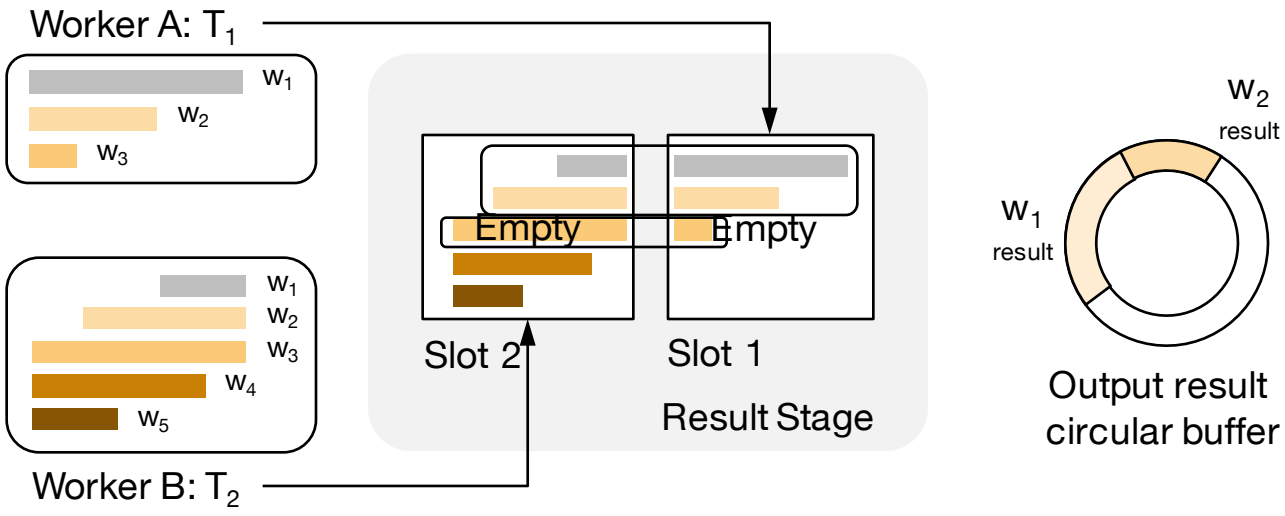
Task contains one or more **window fragments**

- Closing/pending/opening windows in T_2
- Workers process fragments incrementally

Merging Window Fragment Results

Idea: **Decouple task size from window size/slide**

- Assemble window fragment results
- Output them in correct order

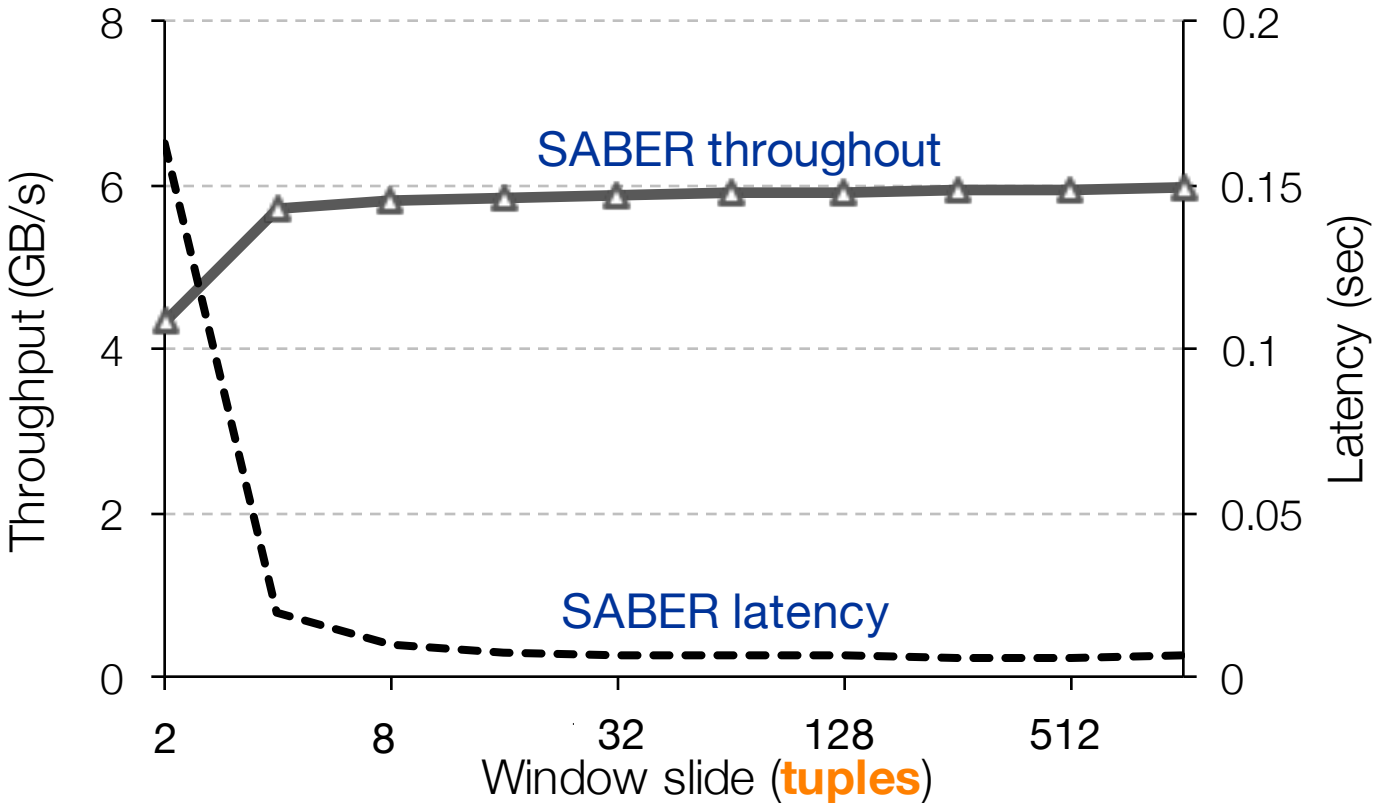


Worker A stores T_1 results, merges with T_2 results, and forwards complete windows downstream

Worker A stores T_1 results, merges with T_2 results (nothing to forward)

Evaluation: SABER Window Performance

select AVG(S.1) **from** S [**rows** 1024 **slide** ***x***]



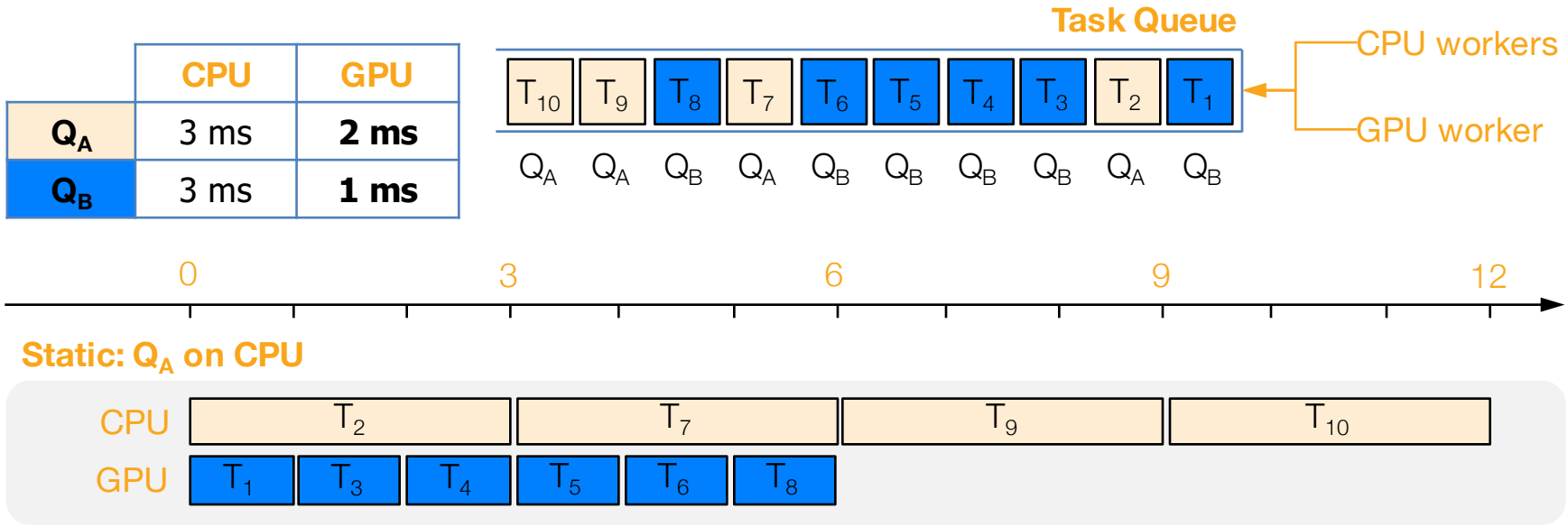
☛ Performance with windowed queries remains predictable

When to use a GPU for a CQL operator?

Statically schedule queries on CPU/GPU based on cost model?

– Cost model depends on operators, windows, input

	CPU	GPU
Q_A	3 ms	**2 ms**
Q_B	3 ms	**1 ms**



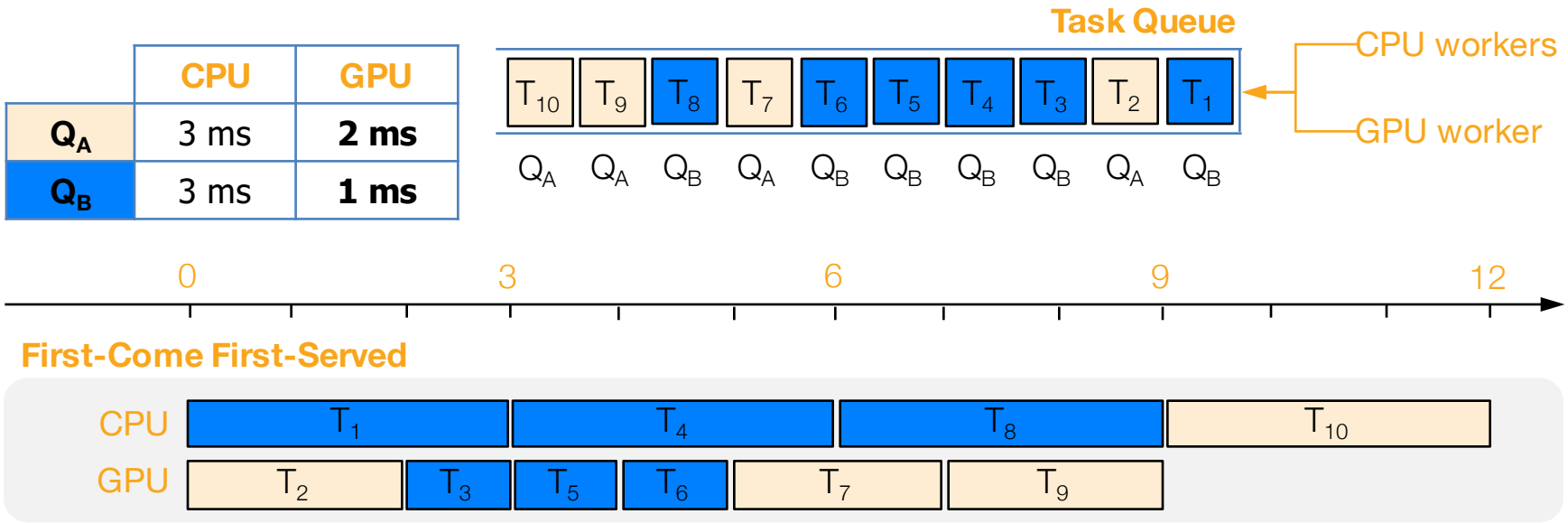
☛ Static scheduling under-utilises processors

SABER's Hybrid Stream Processing Model

Idea: Enable tasks to run on both processors

- Scheduler assigns tasks to idle processors

	CPU	GPU
Q_A	3 ms	**2 ms**
Q_B	3 ms	**1 ms**



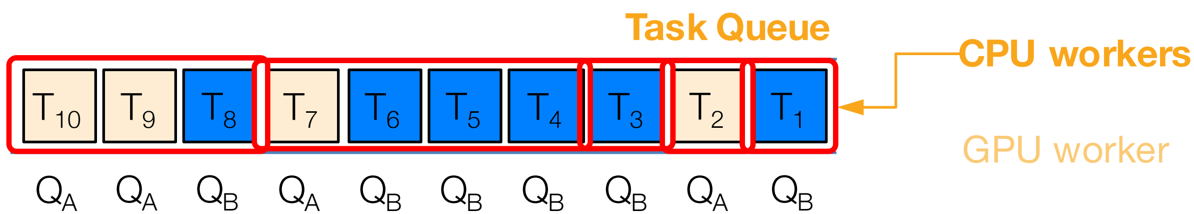
☛ FCFS ignores effectiveness of processor for given task

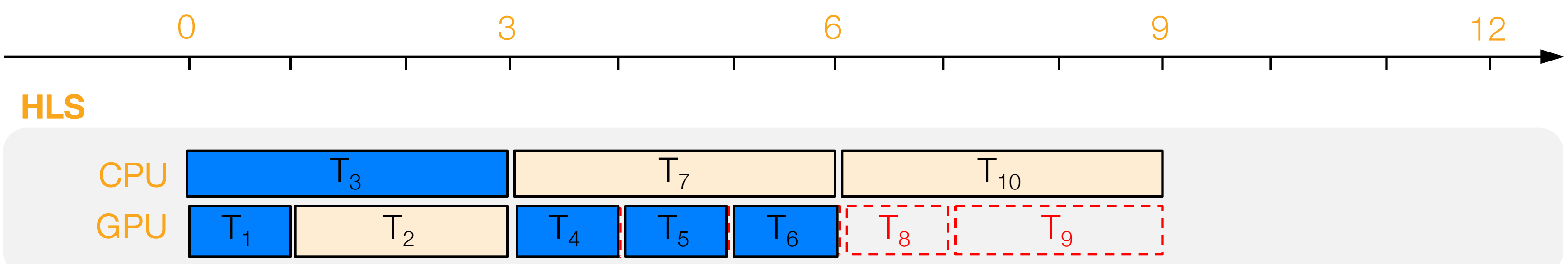
Heterogeneous Look-Ahead Scheduler (HLS)

Idea: Idle processor skips tasks that could be executed faster by another processor

- Decision based on observed query task throughput

	CPU	GPU
Q_A	3 ms	**2 ms**
Q_B	3 ms	**1 ms**



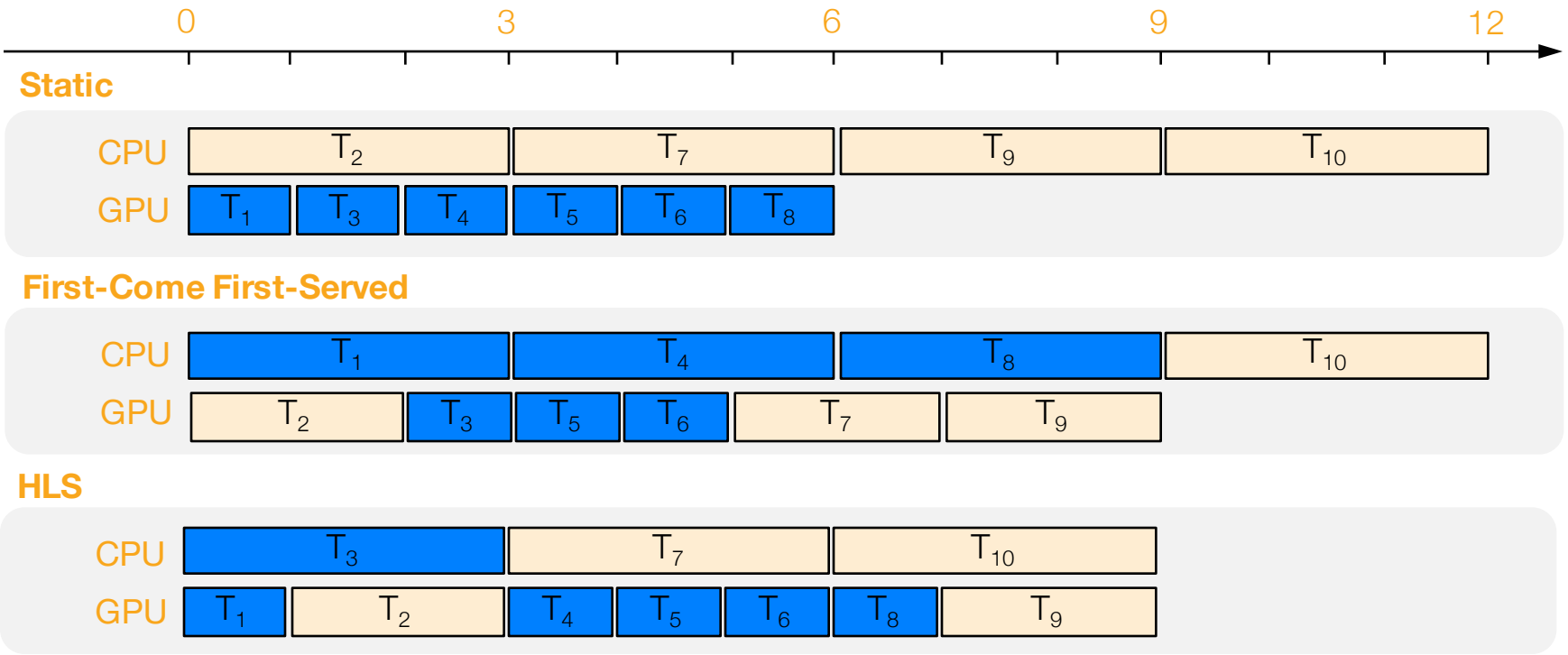


[illegible] 3 ms of work for GPU

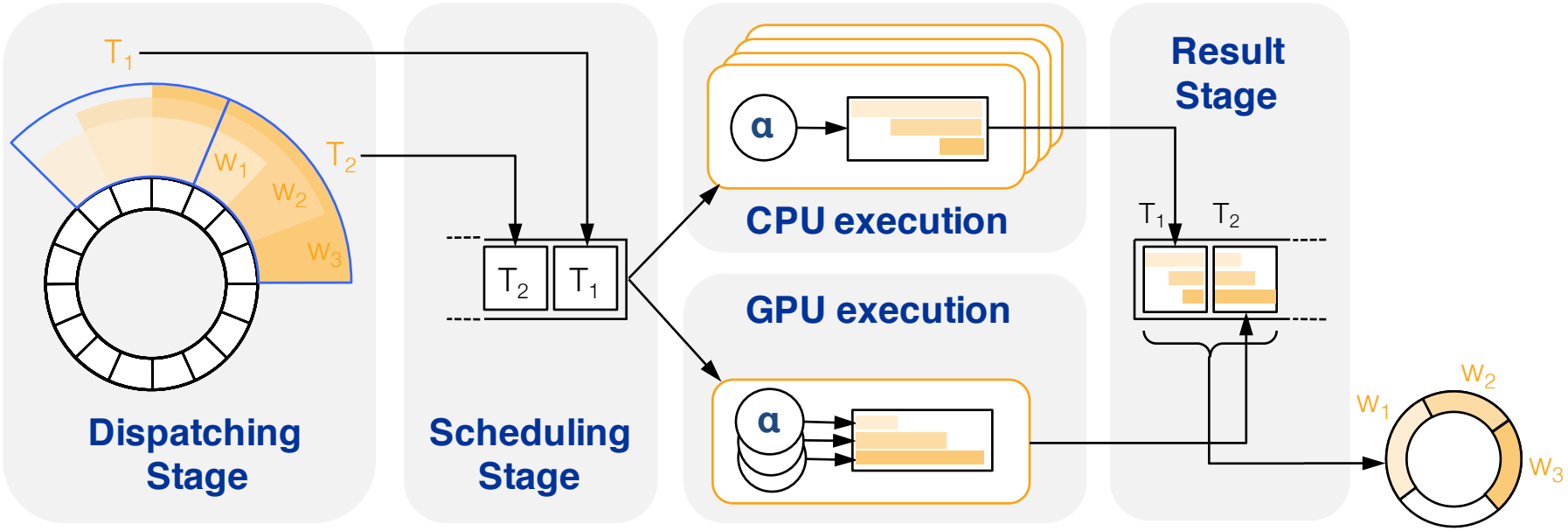
CPU and GPU Contribute Proportionally

HLS gives aggregate throughput of all processors

- CPU executes both Q_A and Q_B tasks



SABER Architecture



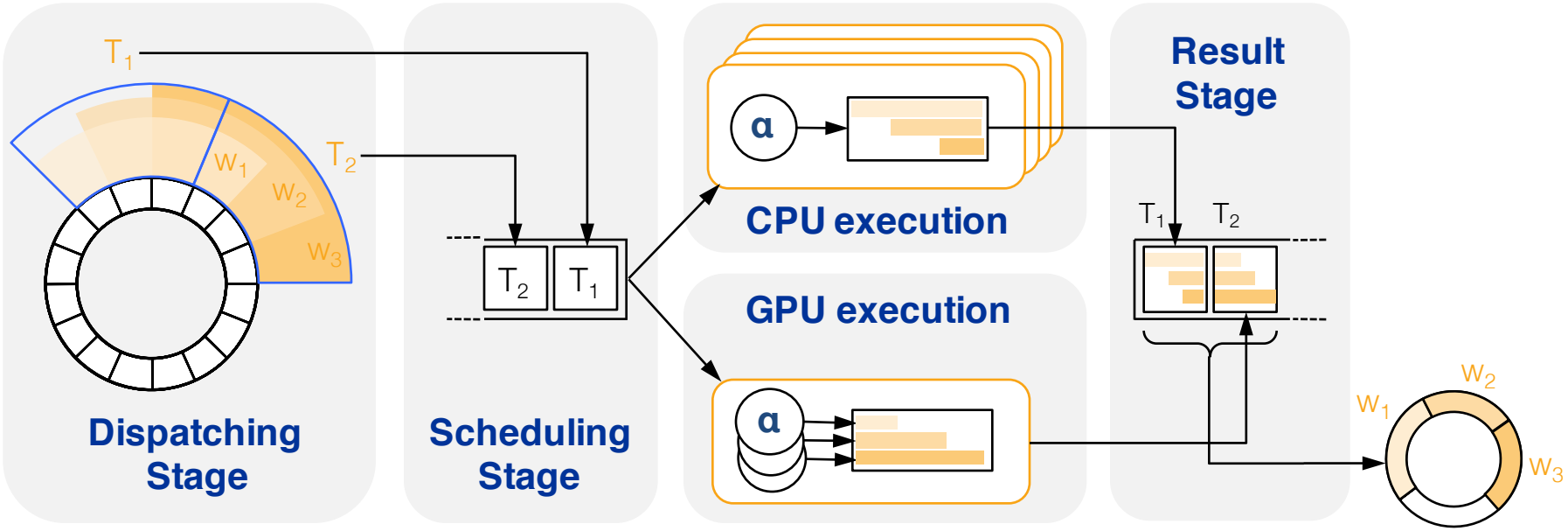
Window computation delayed until task execution → more parallelism

All query tasks added to system-wide lock-free queue

GPU parallelises window fragment result computation

CPU performs incremental window computations instead

SABER Architecture

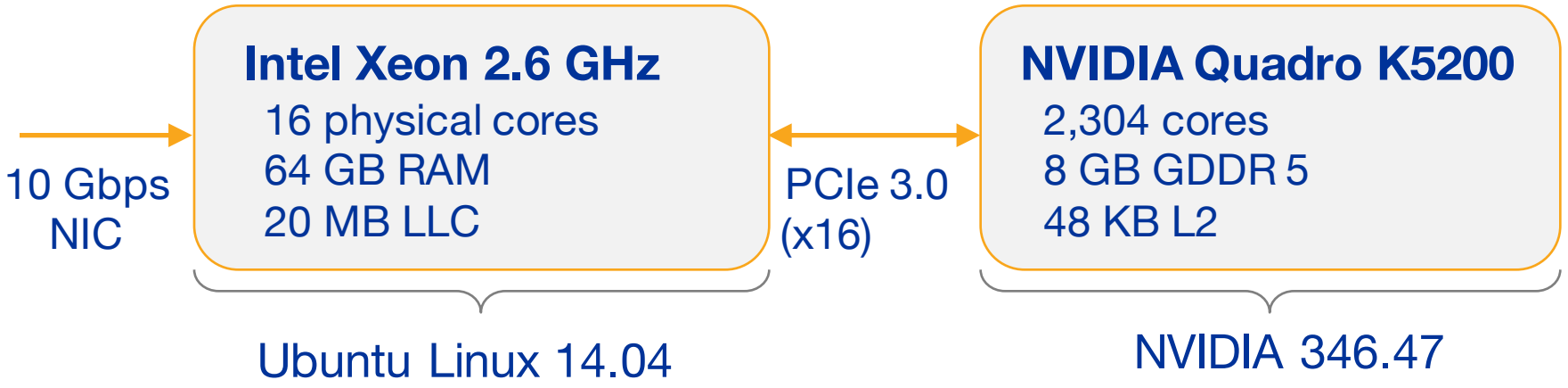


Implemented in Java (15K LOC), C (2K) and OpenCL (2K)

Supports projection, selection, aggregation (w/group-by) and join over time- and count-based windows

Available on Github: **https://github.com/lsds/saber**

Evaluation: Set-up & Workloads

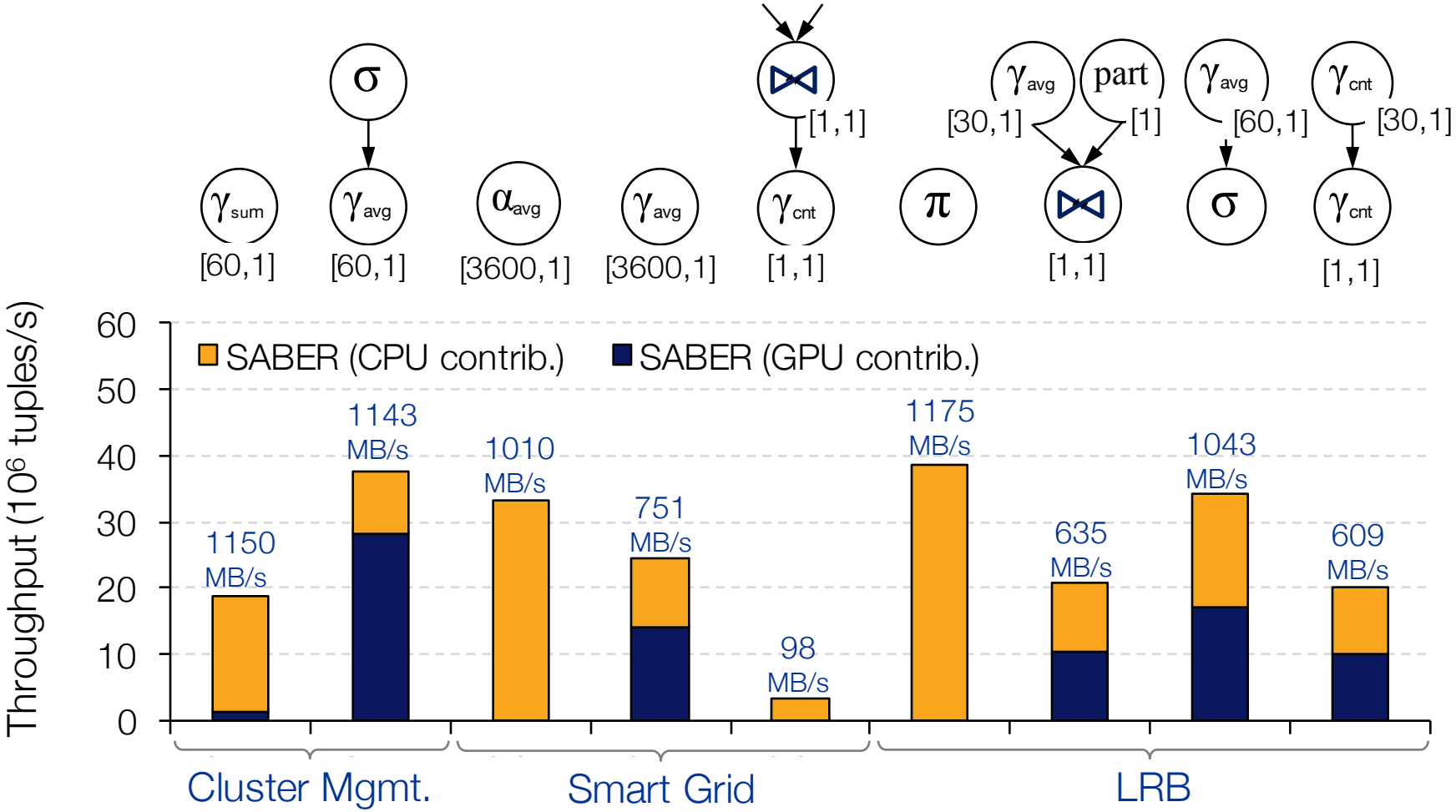


Google cluster data: jobs events from Google infrastructure

SmartGrid measurements: plug measurements from houses

Linear Road Benchmark: car positions and speed on highway

Is Hybrid Stream Processing Effective?



Roadmap

Introduction to Stream Processing Systems

Challenge 1: Performance
How to exploit **parallelism on modern hardware** independently of processing semantics?

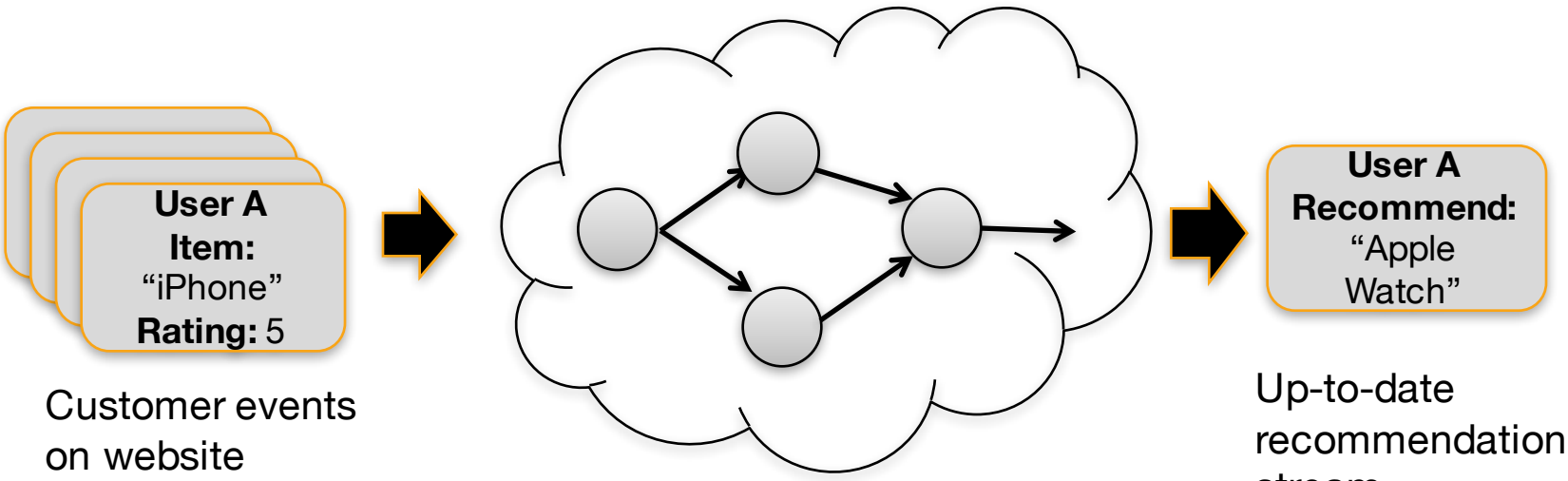
Challenge 2: Programmability
How to support **online machine learning algorithms** over stream data?

Conclusions

Supporting Online Machine Learning

Online recommender system

- Recommendations based on past user ratings
- Eg based on **collaborative filtering** (cf Netflix, Amazon, …)



Customer events on website

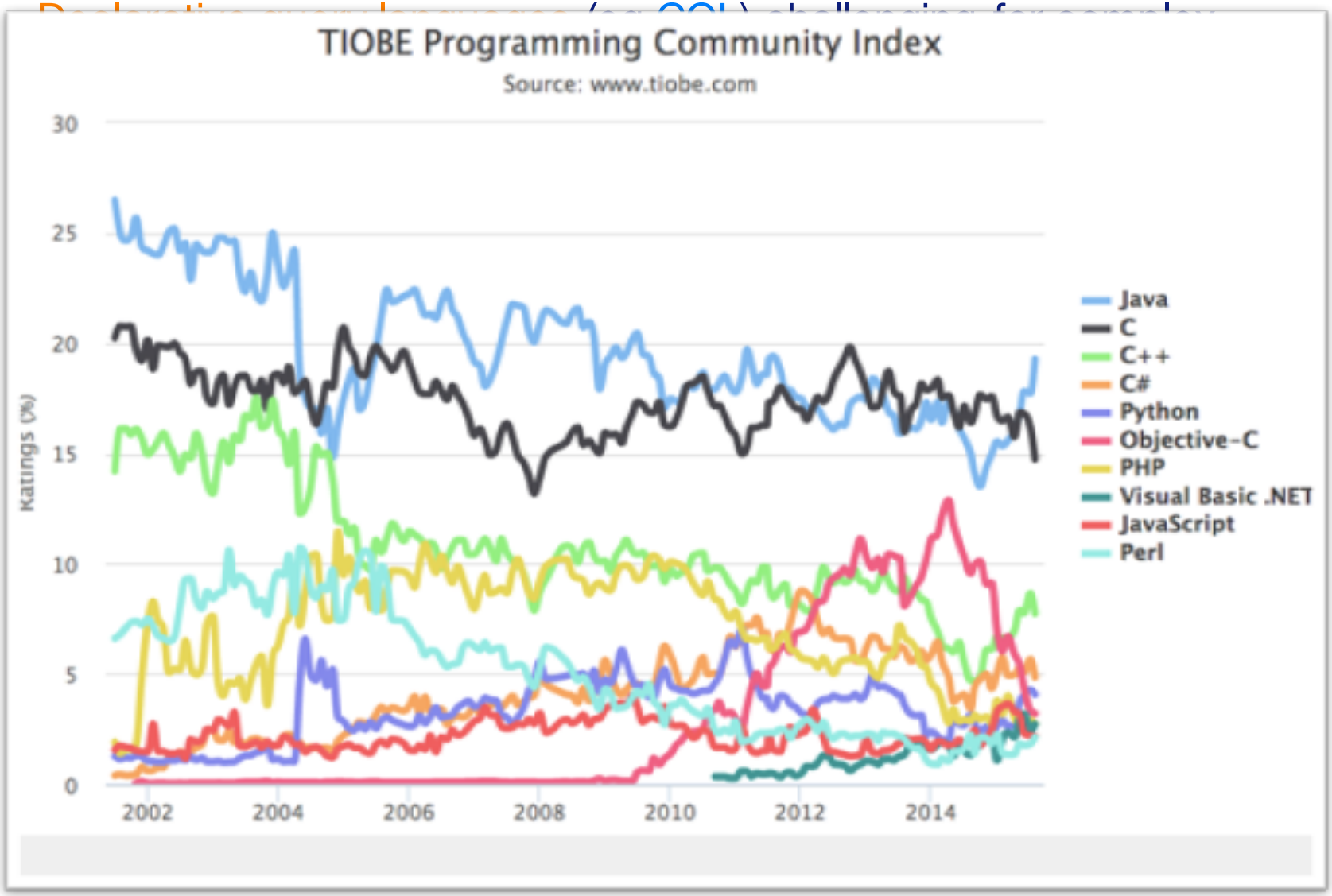
Executed as dataflow graph

(eg Storm, Spark, Flink, …)

Up-to-date recommendation stream

☛ What programming abstraction to use to specify the algorithm?

Programming Models For Stream Processing?



Online Collaborative Filtering In Java

Update with new ratings

	Item-A	Item-B
User-A	4	5
User-B	0	5

User-Item matrix (**UI**)

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

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

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

Multiply for recommendation

User-B	1	2

x

	Item-A	Item-B
Item-A	1	1
Item-B	1	2

Co-Occurrence matrix (**CO**)

Collaborative Filtering In Spark (Java)

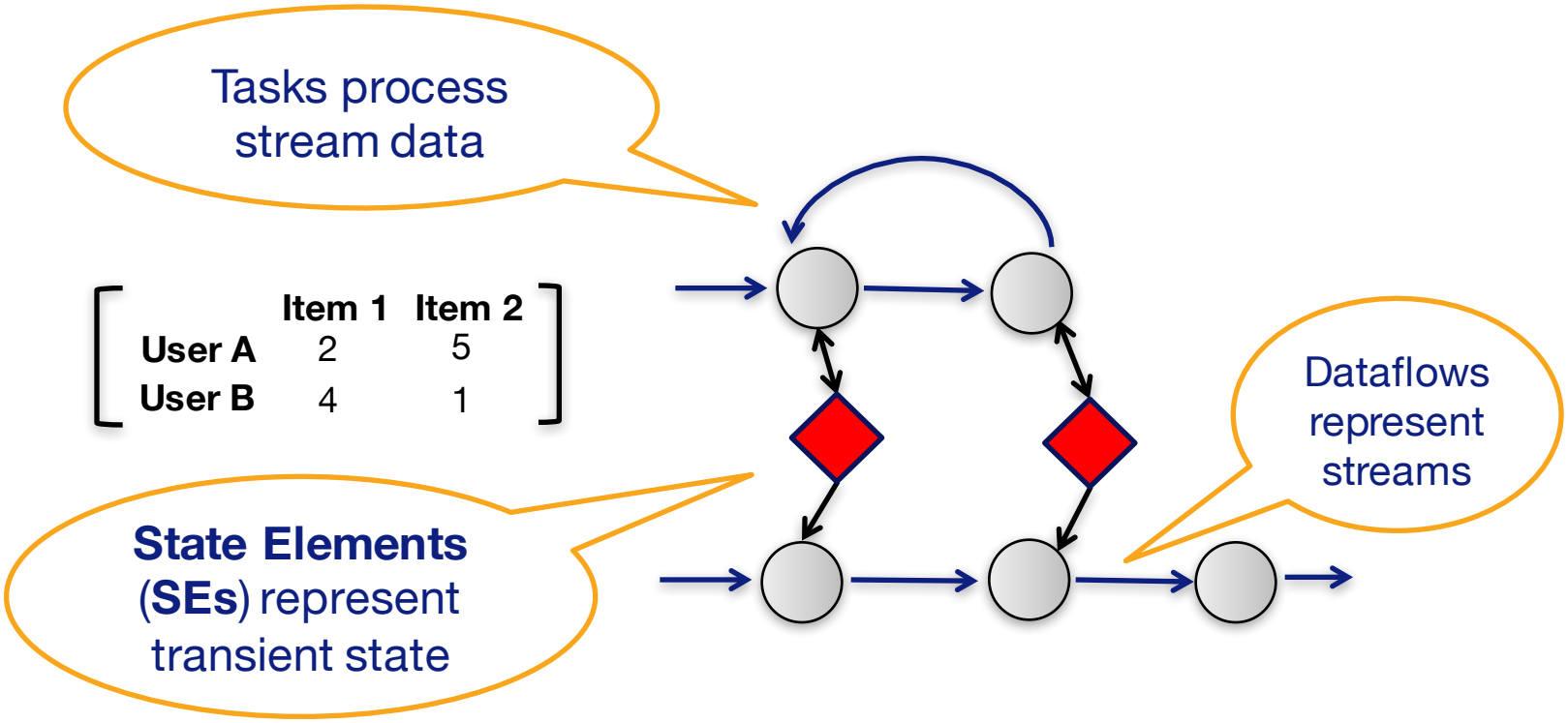
```
// Build the recommendation model using ALS
int rank = 10;
int numIterations = 20;
MatrixFactorizationModel model = ALS.train(JavaRDD.toRDD(ratings), rank, numIterations, 0.01);

// Evaluate the model on rating data
JavaRDD<Tuple2<Object, Object>> userProducts = ratings.map(
  new Function<Rating, Tuple2<Object, Object>>() {
    public Tuple2<Object, Object> call(Rating r) {
      return new Tuple2<Object, Object>(r.user(), r.product());
    }
  }
);
JavaPairRDD<Tuple2<Integer, Integer>, Double> predictions = JavaPairRDD.fromJavaRDD(
  model.predict(JavaRDD.toRDD(userProducts)).toJavaRDD().map(
    new Function<Rating, Tuple2<Tuple2<Integer, Integer>, Double>>() {
      public Tuple2<Tuple2<Integer, Integer>, Double> call(Rating r){
        return new Tuple2<Tuple2<Integer, Integer>, Double>(
          new Tuple2<Integer, Integer>(r.user(), r.product()), r.rating());
      }
    }
));
JavaRDD<Tuple2<Double, Double>> ratesAndPreds = JavaPairRDD.fromJavaRDD(ratings.map(
    new Function<Rating, Tuple2<Tuple2<Integer, Integer>, Double>>() {
      public Tuple2<Tuple2<Integer, Integer>, Double> call(Rating r){
        return new Tuple2<Tuple2<Integer, Integer>, Double>(
          new Tuple2<Integer, Integer>(r.user(), r.product()), r.rating());
      }
    }
)).join(predictions).values();
```

Collaborative Filtering In Spark (Scala)

```
// Build the recommendation model using ALS
val rank = 10
val numIterations = 20
val model = ALS.train(ratings, rank, numIterations, 0.01)

// Evaluate the model on rating data
val usersProducts = ratings.map {
  case Rating(user, product, rate) => (user, product)
}
val predictions =
  model.predict(usersProducts).map {
    case Rating(user, product, rate) => ((user, product), rate)
  }
val ratesAndPreds = ratings.map {
  case Rating(user, product, rate) => ((user, product), rate)
}.join(predictions)
```

☛ All event data is immutable, no fine-grained model updates

Processing State As First Class Citizen



State elements (SEs) are mutable in-memory data structures

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

Challenges With Large Processing State

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

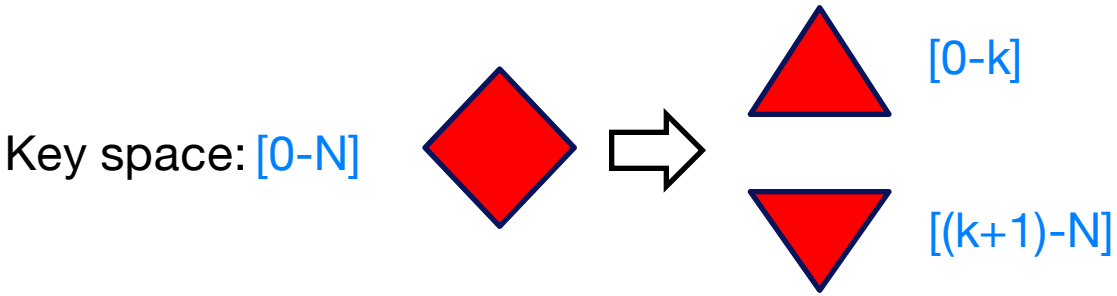
Big Data problem: **Matrices become large**

State will not fit into single node

☛ How to handle distributed state in a scalable fashion?

1. Partitioned State Elements

Idea: **Partitioned SEs** are split into disjoint partitions

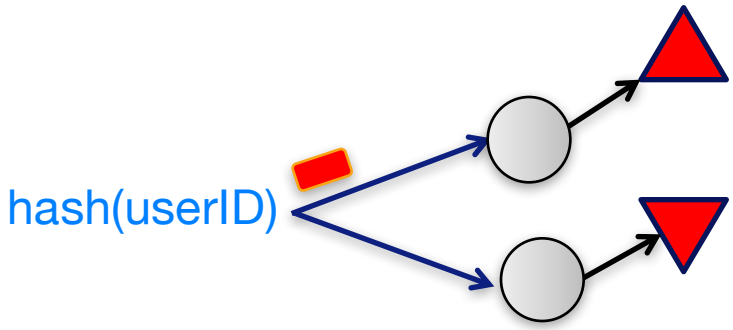


User-Item matrix (UI)

Access by key

	Item-A	Item-B
User-A	4	5
User-B	0	5

State **partitioned** according to partitioning key



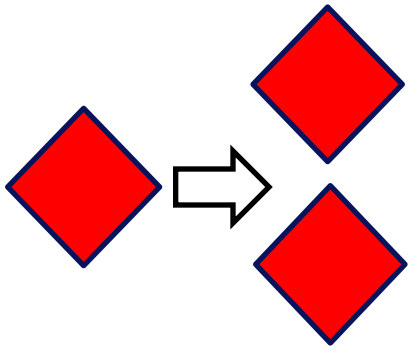
Dataflow **routed** according to hash function

2. Partial State Elements

Partial SEs are replicated (when partitioning is not possible)

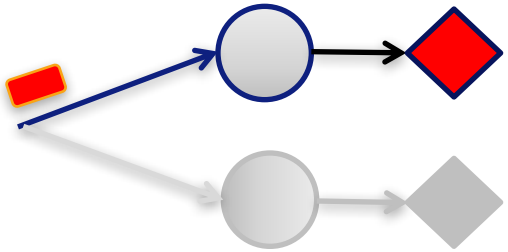
Co-Occurrence matrix (**CO**)

	Item-A	Item-B
Item-A	1	1
Item-B	1	2

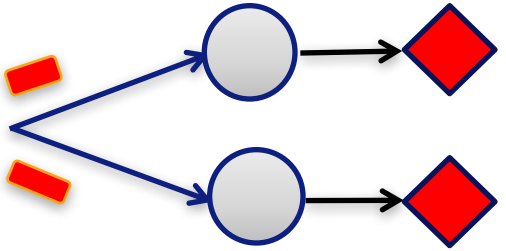


– Replicas kept **weakly consistent**

Access to partial SEs either **local** or **global**



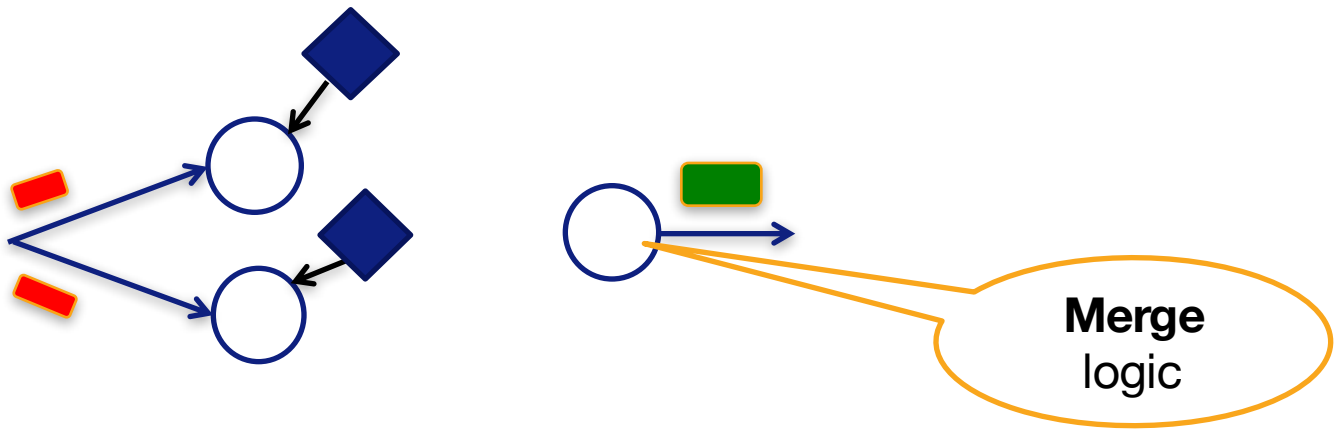
Local access:
Tuples sent to one



Global access:
Tuples sent to all

State Synchronisation with Partial SEs

Reading all partial SE instances results in set of partial values

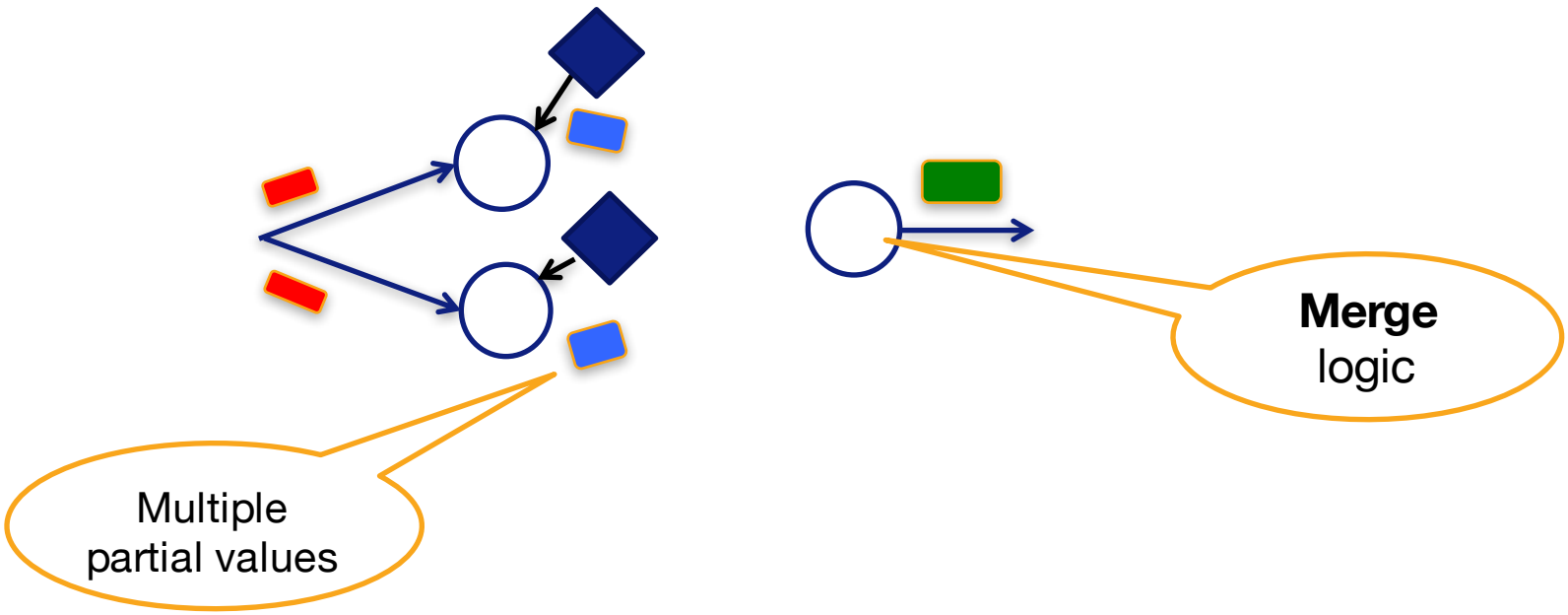


Requires application-specific merge logic

- Merge task reconciles state and updates partial SEs

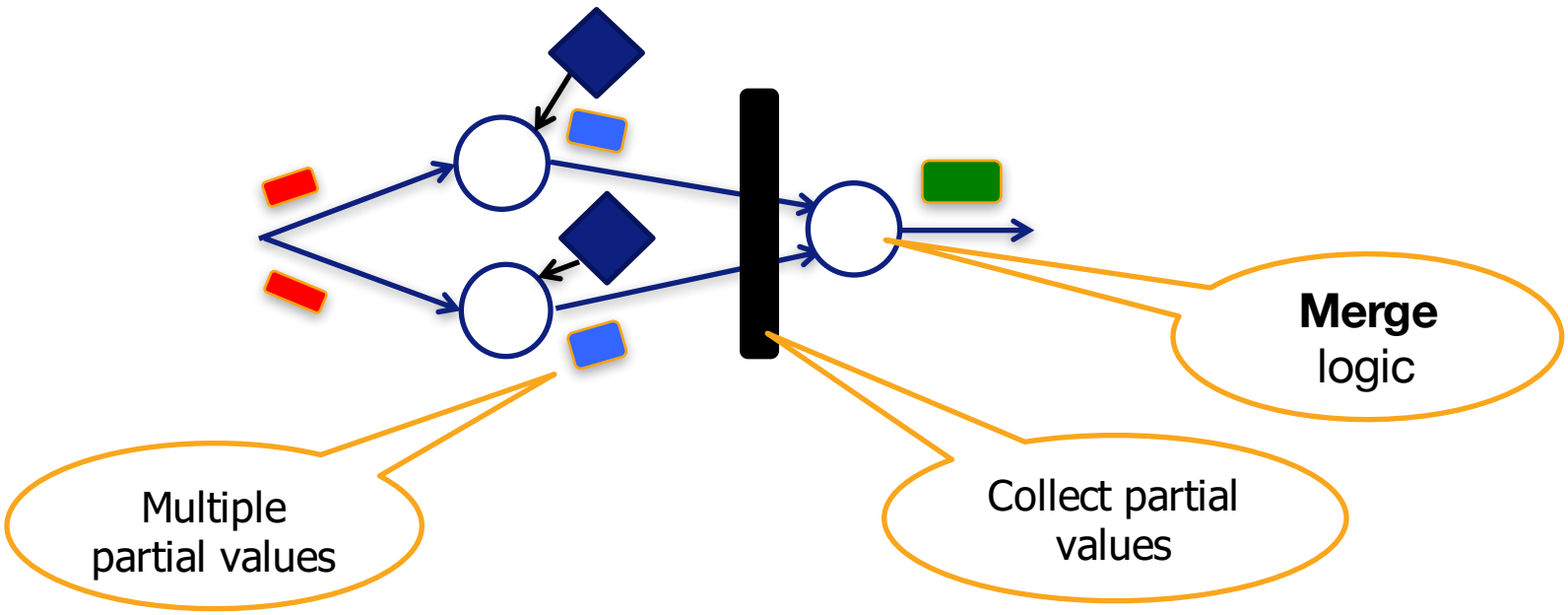
State Synchronisation with Partial SEs

Reading all partial SE instances results in set of partial values



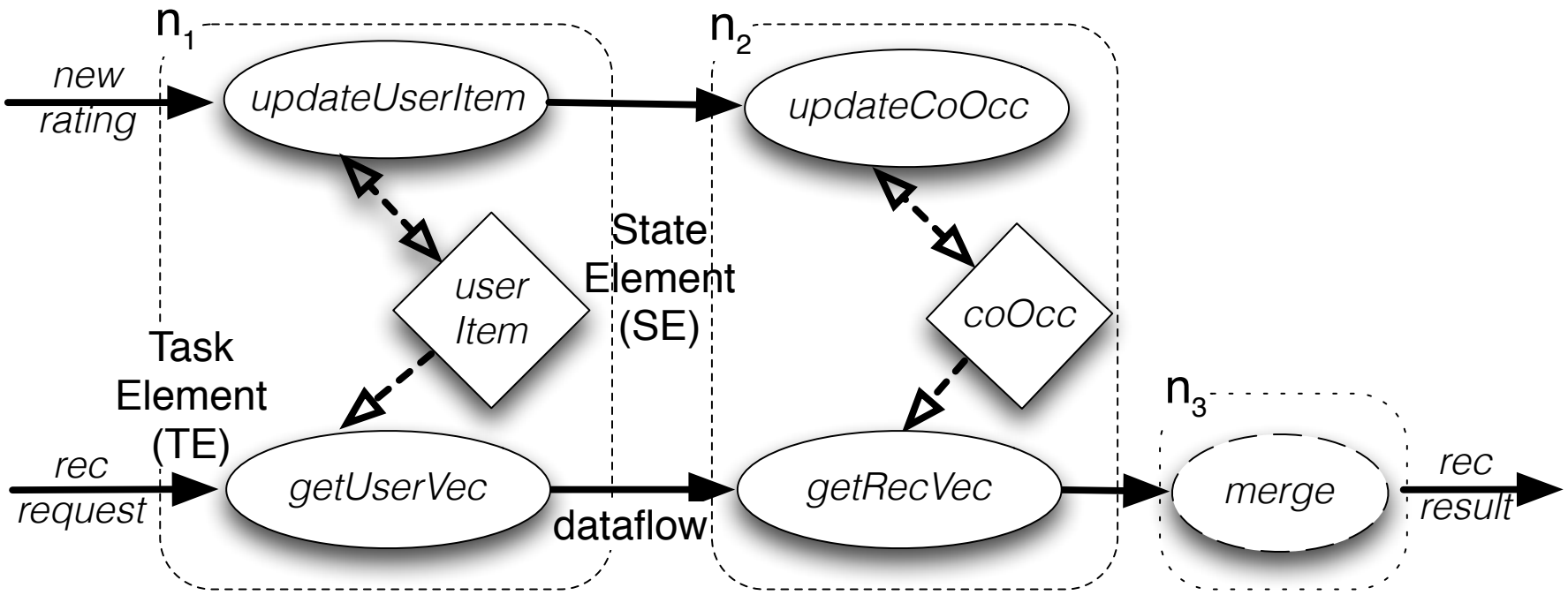
State Synchronisation with Partial SEs

Reading all partial SE instances results in set of partial values



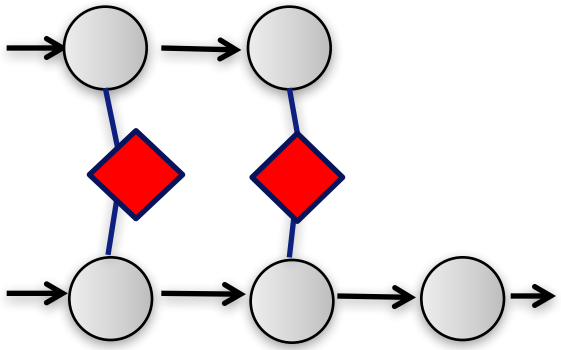
Barrier collects partial state

SDG for Collaborative Filtering



☛ Note that this combines **batch** and **stream** processing in single model

Scalable & Elastic Event Processing (SEEP)



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

Program.java

Static program analysis

Data-parallel Stateful Dataflow Graph (SDG)

SEEP distributed dataflow framework

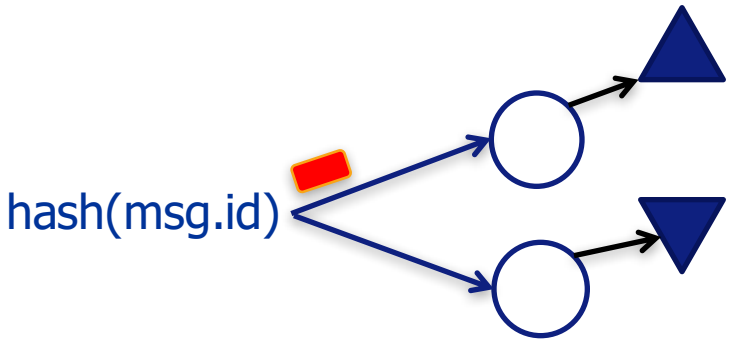
Cluster

Translation & checkpoint-based fault tolerance

Partitioned Java Annotation

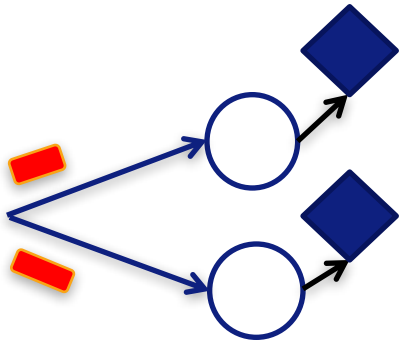
@Partition field annotation indicates partitioned state

```
@Partitioned Matrix userItem = new SeepMatrix();
Matrix coOcc = new Matrix();

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

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



Partial State and Global Annotations

```
@Partitioned Matrix userItem = new SeepMatrix();
@Partial Matrix coOcc = new SeepMatrix();

void processRatingEvent(int user, int item, int rating) {
  userItem.setElement(user, item, rating);
  updateCoOccurrence(@Global coOcc, userItem);
}
```

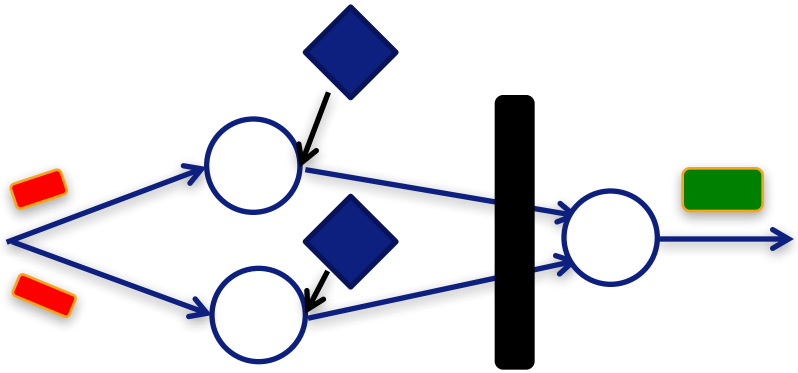


@Partial field annotation indicates **partial state**

@Global annotates variable to indicate **access to all partial instances**

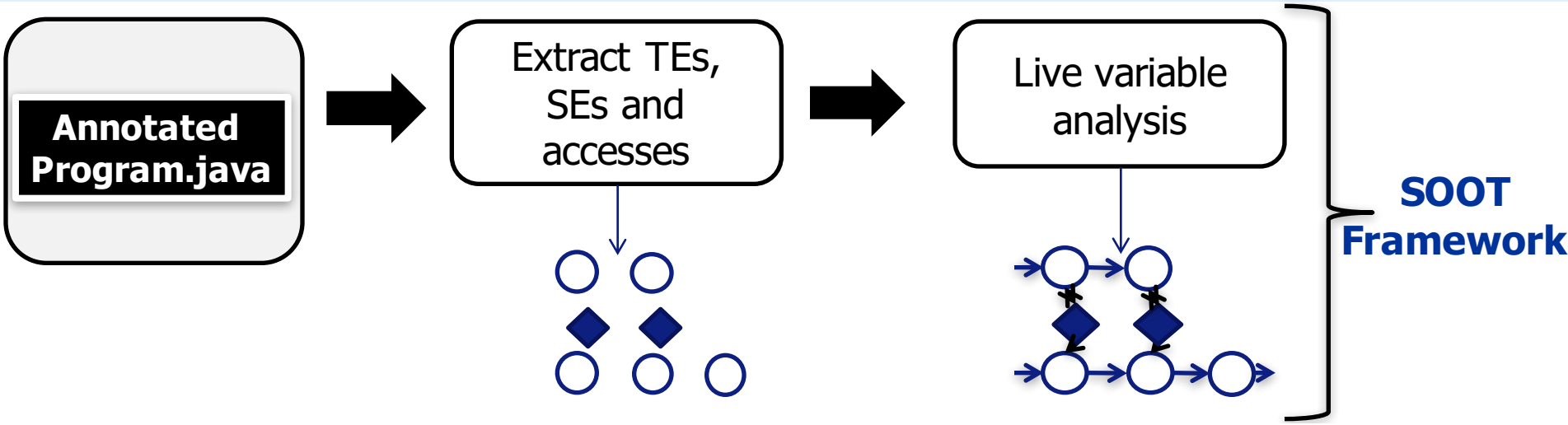
Partial and Collection Annotation

```
@Partitioned Matrix userItem = new SeepMatrix();
@Partial Matrix coOcc = new SeepMatrix();

Vector processRecEvent(int user) {
  Vector userRow = userItem.getRow(user);
  @Partial Vector puRec = @Global coOcc.multiply(userRow);
  Vector userRec = merge(puRec);
  return userRec;
}

Vector merge(@Collection Vector[] v){
  /*…*/
}
```

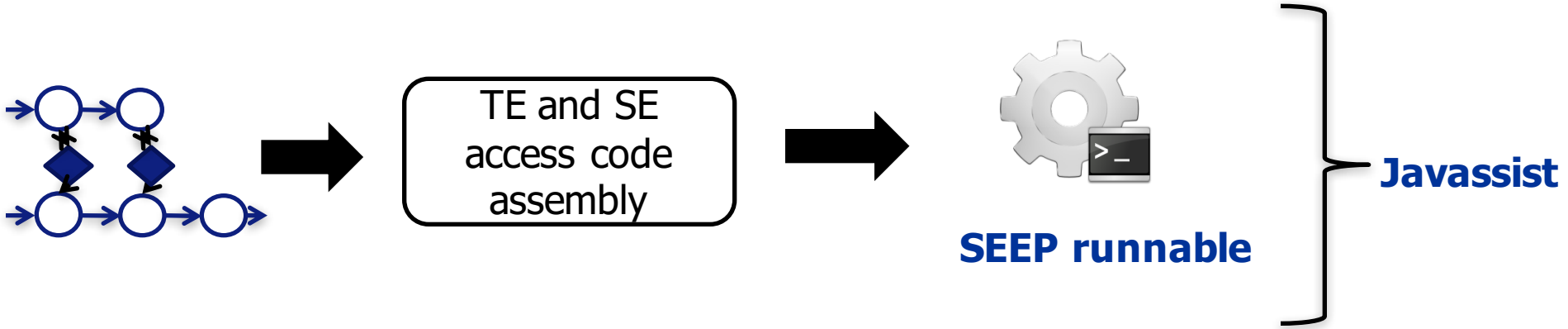


@Collection annotation indicates **merge logic**

Java2SDG: Translation Process

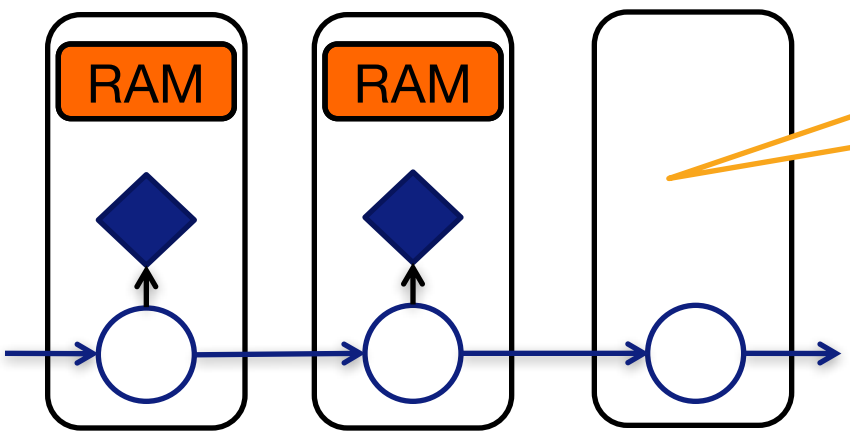


Extract **state** and **state access patterns** through static code analysis



Generation of **runnable code** using TE and SE connections

Fault Tolerance With State



Physical deployment of SDG

Task elements access local **in-memory** state

Node failures may lead to **state loss**

Checkpointing state

No updates allowed while state is being checkpointed

Checkpointing state should not impact data processing path

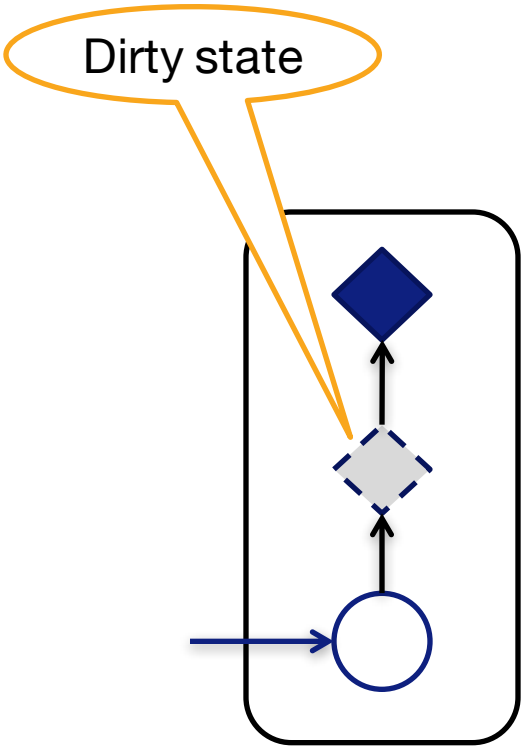
State backup

Backups large and cannot be stored in memory

Large writes to disk through network have high cost

Checkpointing Support for SDGs

Challenge: **Efficient checkpointing** of large state in Java?



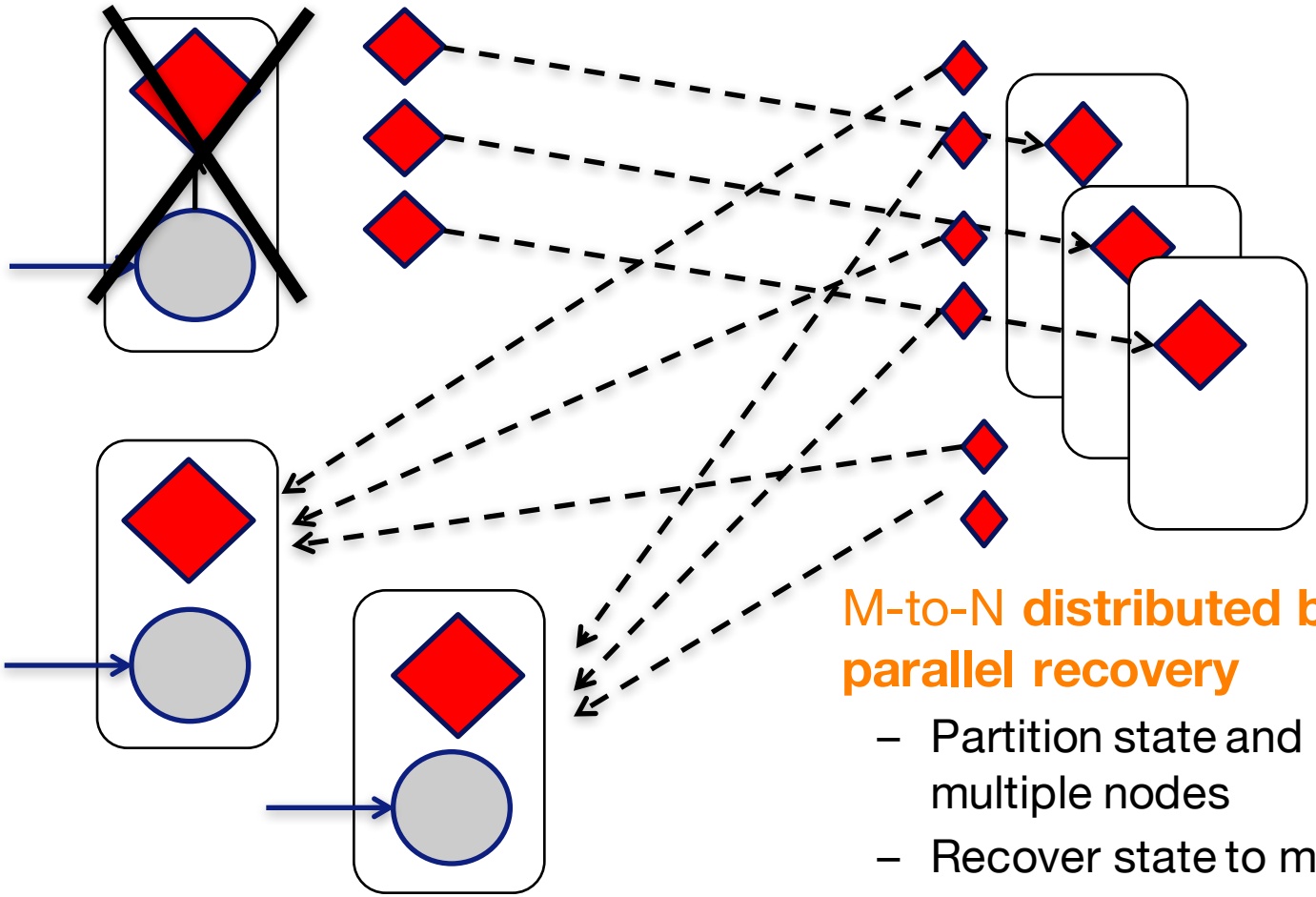
Asynchronous, lock-free checkpointing

1. Freeze mutable state for checkpointing
2. Dirty state supports updates concurrently
3. Reconcile dirty state

Distributed M-to-N Backup/Recovery

Challenge: **Fast recovery?**

- Backups large and cannot be stored in memory
- Large writes to disk through network have high cost



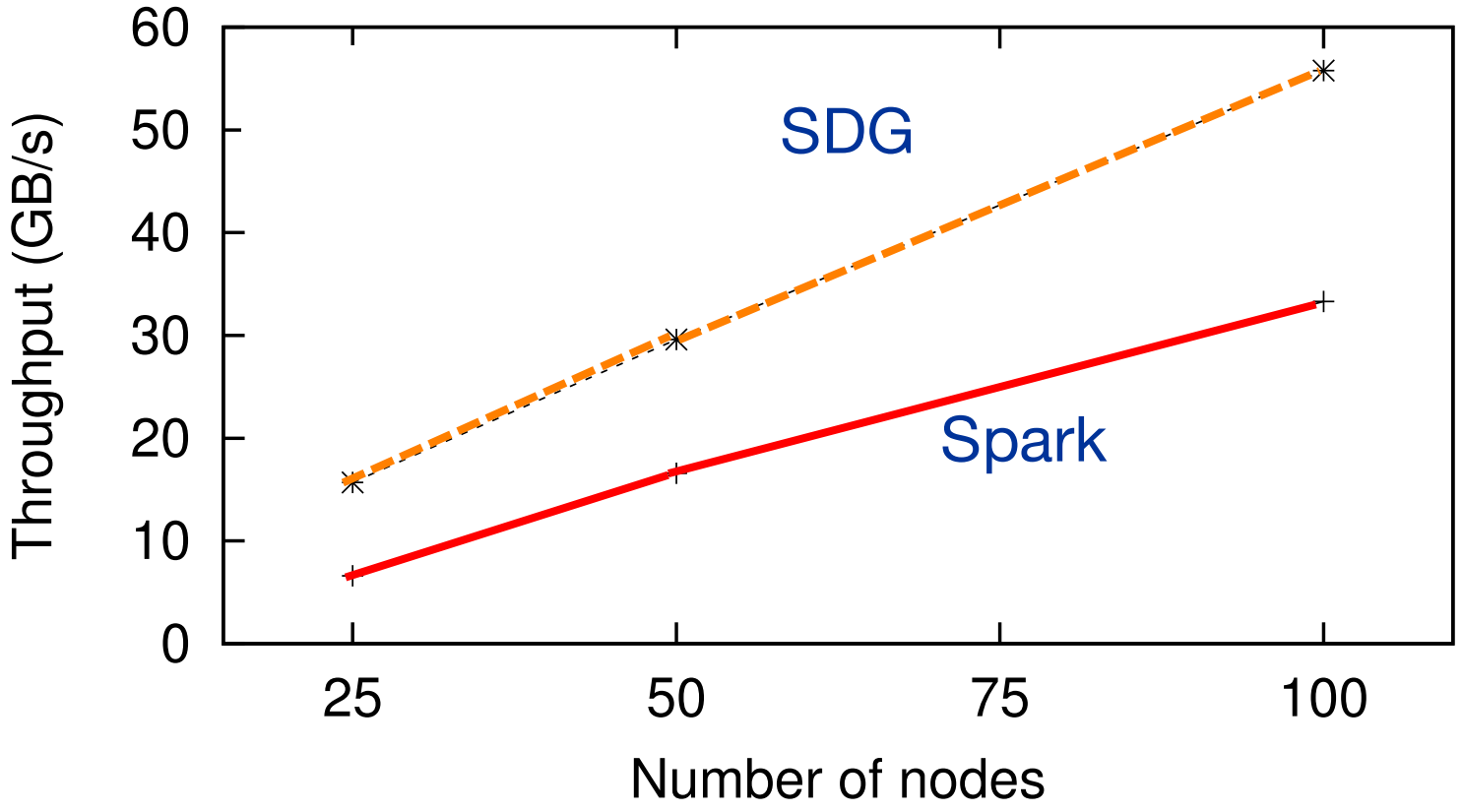
M-to-N **distributed backup** and **parallel recovery**

- Partition state and backup to multiple nodes
- Recover state to multiple nodes

Evaluation: Spark Comparison

Online logistic regression with 100 GB training dataset

Deployed on Amazon EC2 ("m1.xlarge" VMs with 4 vCPUs and 16 GB RAM)

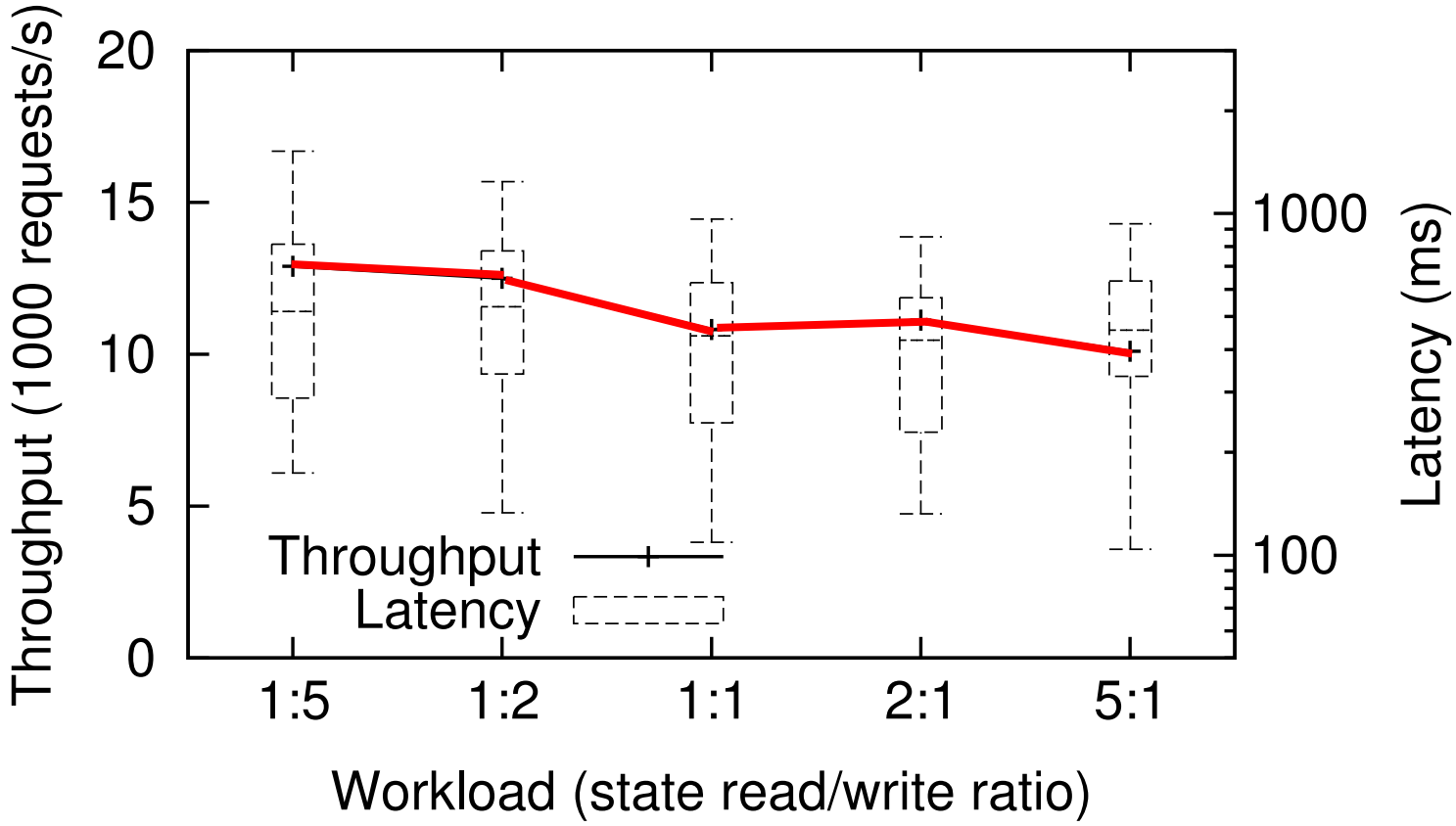


☛ SDG has comparable throughput to Spark despite mutable state

Evaluation: Mutable State Access

Collaborative filtering, while changing read/write ratio (add/getRating)

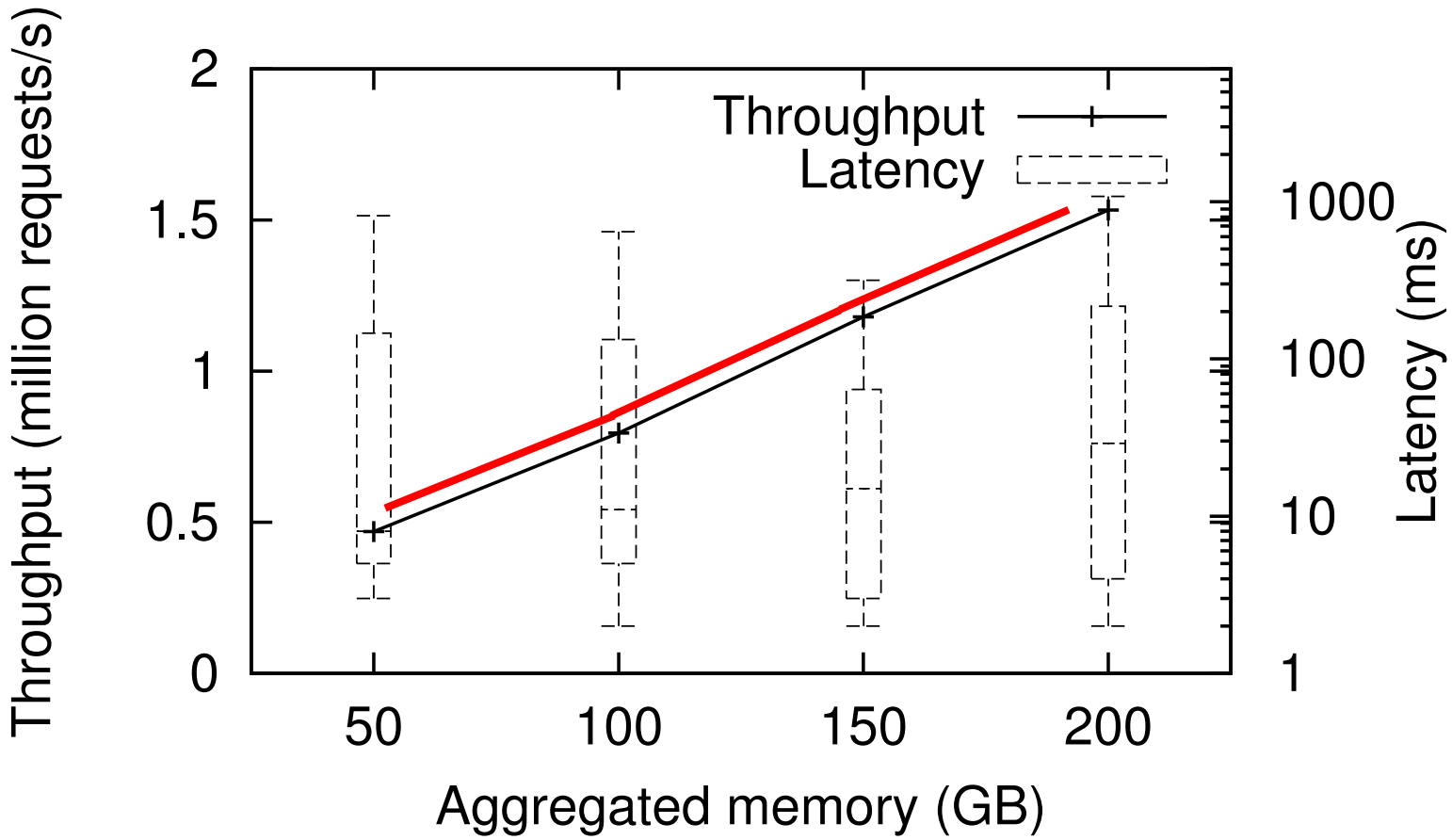
Private cluster (4-core 3.4 GHz Intel Xeon servers with 8 GB RAM)



☛ Higher read workload impacts performance due to state reconciliation

Evaluation: Large State Size

Increase state size in distributed key/value store



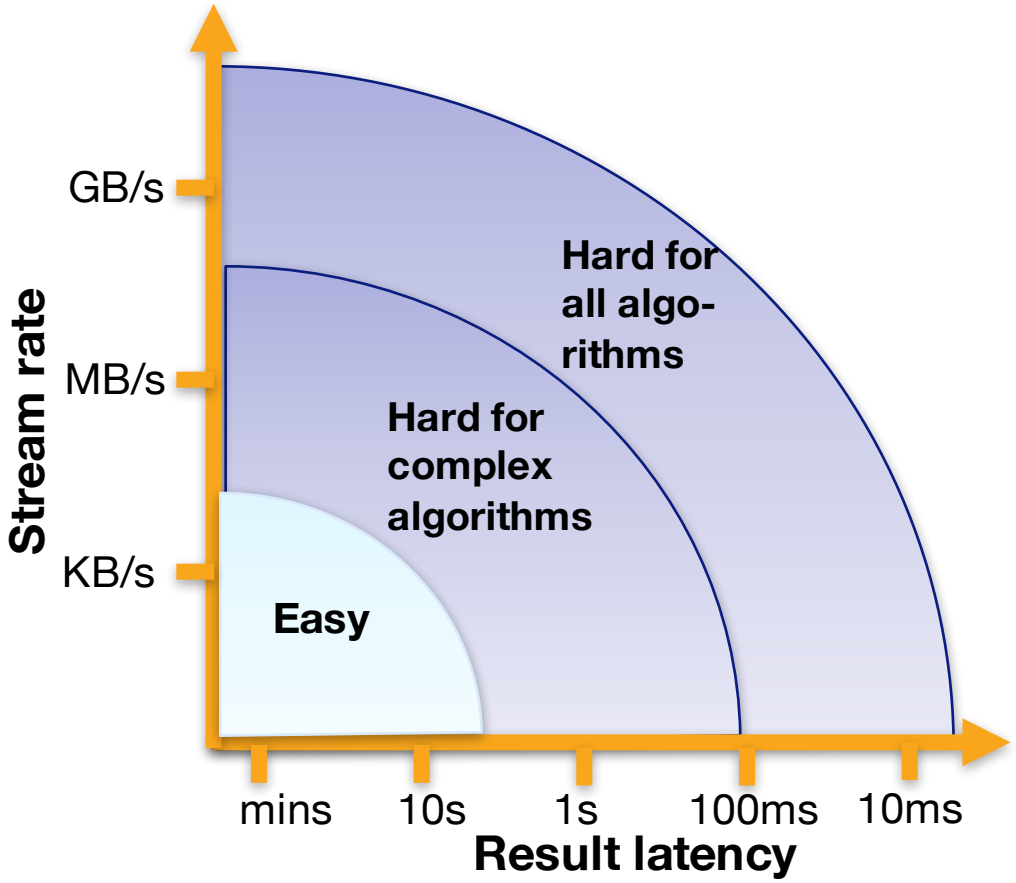
☛ SDGs can support online applications with mutable state

Conclusions I

Stream processing is a crucial part of many data processing stacks

- Many applications and services require real-time view of data streams
- Batch processing models increasingly replaced by stream processing

Interesting tension between **performance** and **algorithmic complexity**



Conclusions II

1. Modern **parallel hardware** (multicore CPUs/GPUs) raises challenges
   - New event-based system designs must exploit **data parallelism**
   - But must not couple performance with **processing semantics**

☛ **SABER: Principled window handling in parallel stream processing**

2. **Online machine learning** over events is killer application
   - Complex streaming applications require **expressive programming models**
   - Want to offer **natural programming abstractions** to users

☛ **SDG: Stateful stream processing for machine learning**

Thank you! Any Questions?

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