Imperial College London

LSDS
Large-Scale Distributed Systems Group

Stateful Distributed Dataflow Graphs: Imperative Big Data Programming for the Masses

Peter Pietzuch
prp@doc.ic.ac.uk

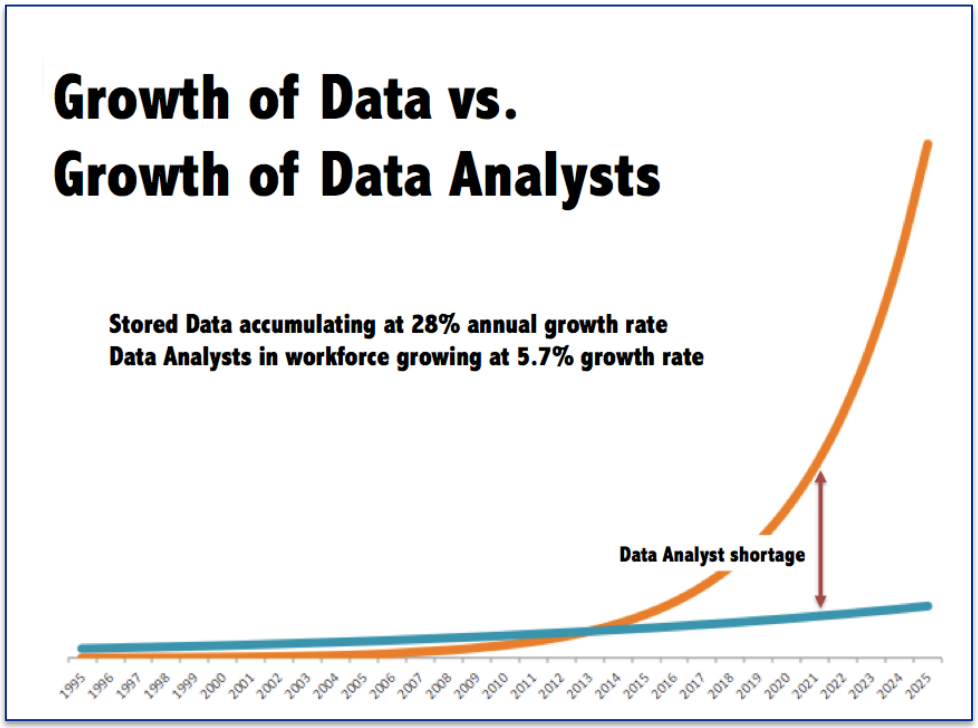
Large-Scale Distributed Systems Group
Department of Computing, Imperial College London
http://lsds.doc.ic.ac.uk

EIT Digital Summer School on Cloud and Big Data 2015 – Stockholm, Sweden

Growth of Big Data Analytics

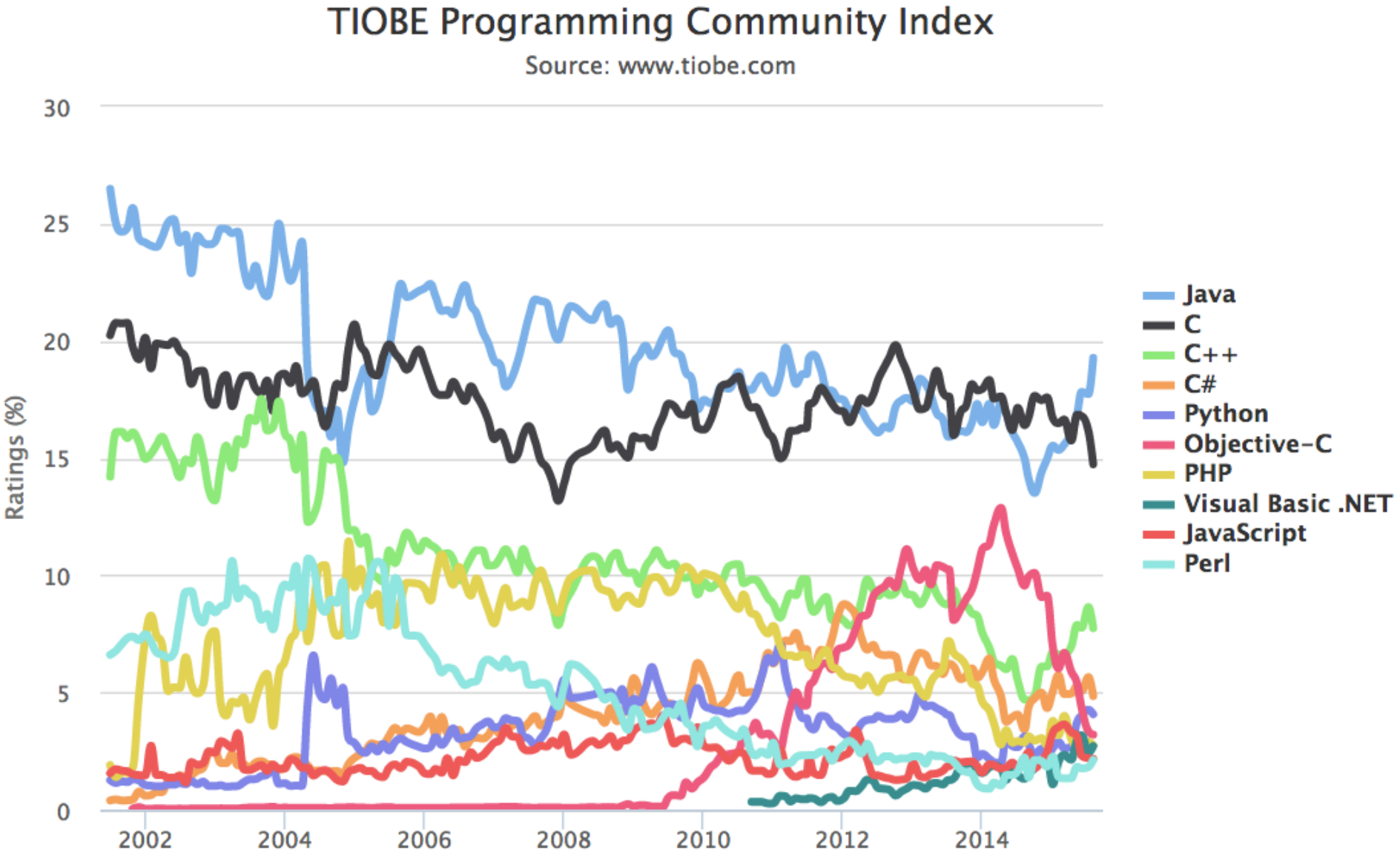
Big Data Analytics: gaining value from data

- Web analytics, fraud detection, system management, networking monitoring, business dashboard, …



Need to enable more users to perform data analytics

Programming Language Popularity



Programming Models For Big Data?

Distributed dataflow frameworks tend to favour functional, declarative programming models

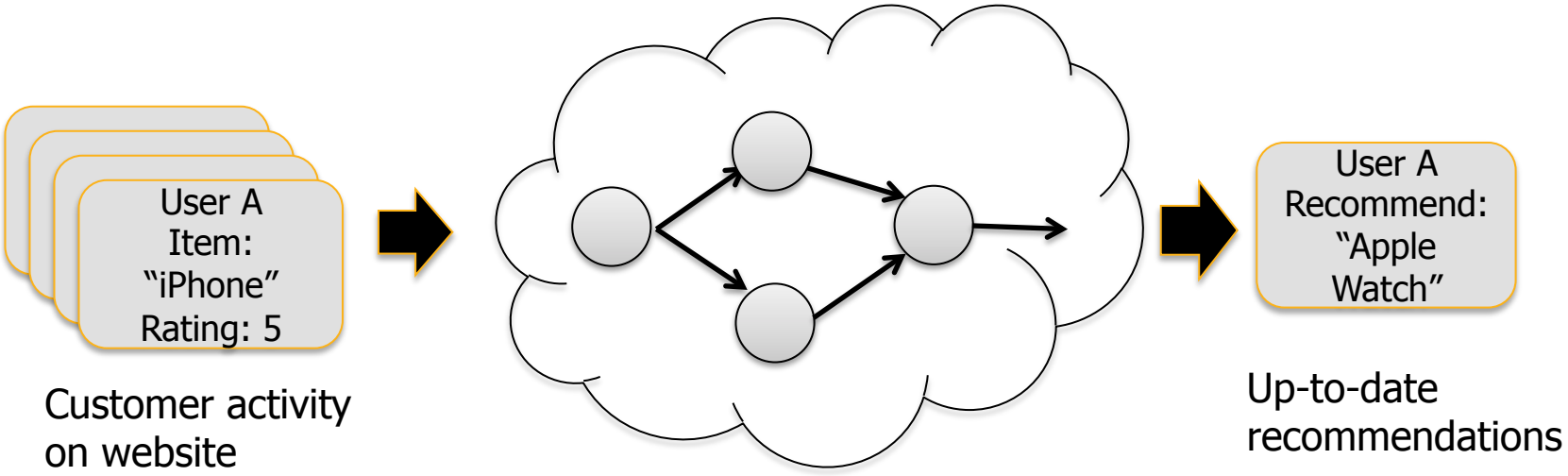
- MapReduce, SQL, PIG, DryadLINQ, Spark, …
- Facilitates consistency and fault tolerance issues

Domain experts tend to write imperative programs

- Java, Matlab, C++, R, Python, Fortran, …

Example: Recommender Systems

Recommendations based on past user behaviour through collaborative filtering (cf. Netflix, Amazon, …):



Customer activity on website

Up-to-date recommendations

Distributed dataflow graph

(eg MapReduce, Hadoop, Spark, Dryad, Naiad, …)

Exploits data-parallelism on cluster of machines

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

Vector getRec(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();
```

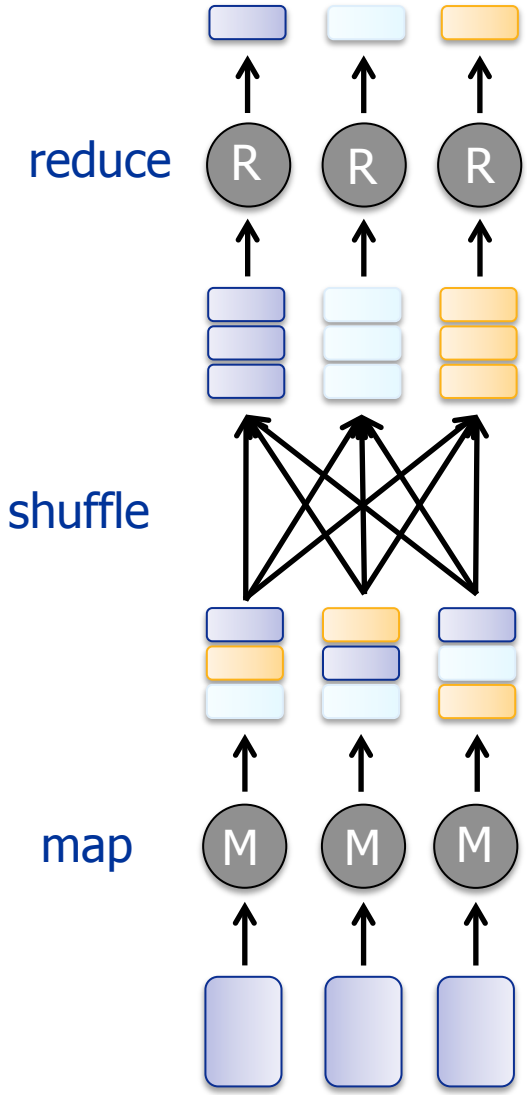
Collaborative Filtering in Spark (Scala)

```
// 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 data immutable

No fine-grained model updates

Stateless MapReduce Model



Data model: (key, value) pairs

Two processing functions:

map(k_1,v_1) → list(k_2,v_2)

reduce(k_2, list(v_2)) → list (v_3)

Benefits:

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

Big Data Programming for the Masses

Our goals:

Imperative **Java programming model** for big data apps

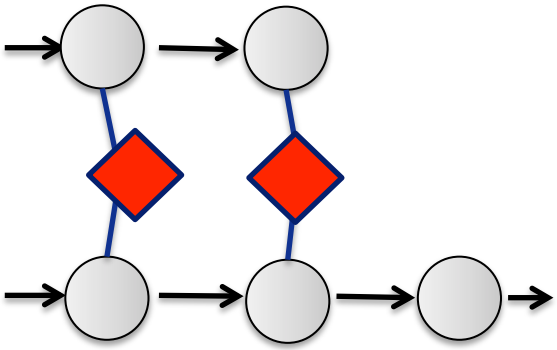
High throughput through **data-parallel execution** on cluster

Fault tolerance against node failures

System	Mutable State	Large State	Low Latency	Iteration
MapReduce	No	n/a	No	No
Spark	No	n/a	No	Yes
Storm	No	n/a	Yes	No
Naiad	Yes	No	Yes	Yes

Stateful Dataflow Graphs (SDGs)

1



2

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

Program.java

Static program analysis

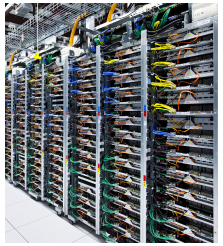
Data-parallel **Stateful Dataflow Graph (SDG)**

3

SEEP distributed dataflow framework

Cluster

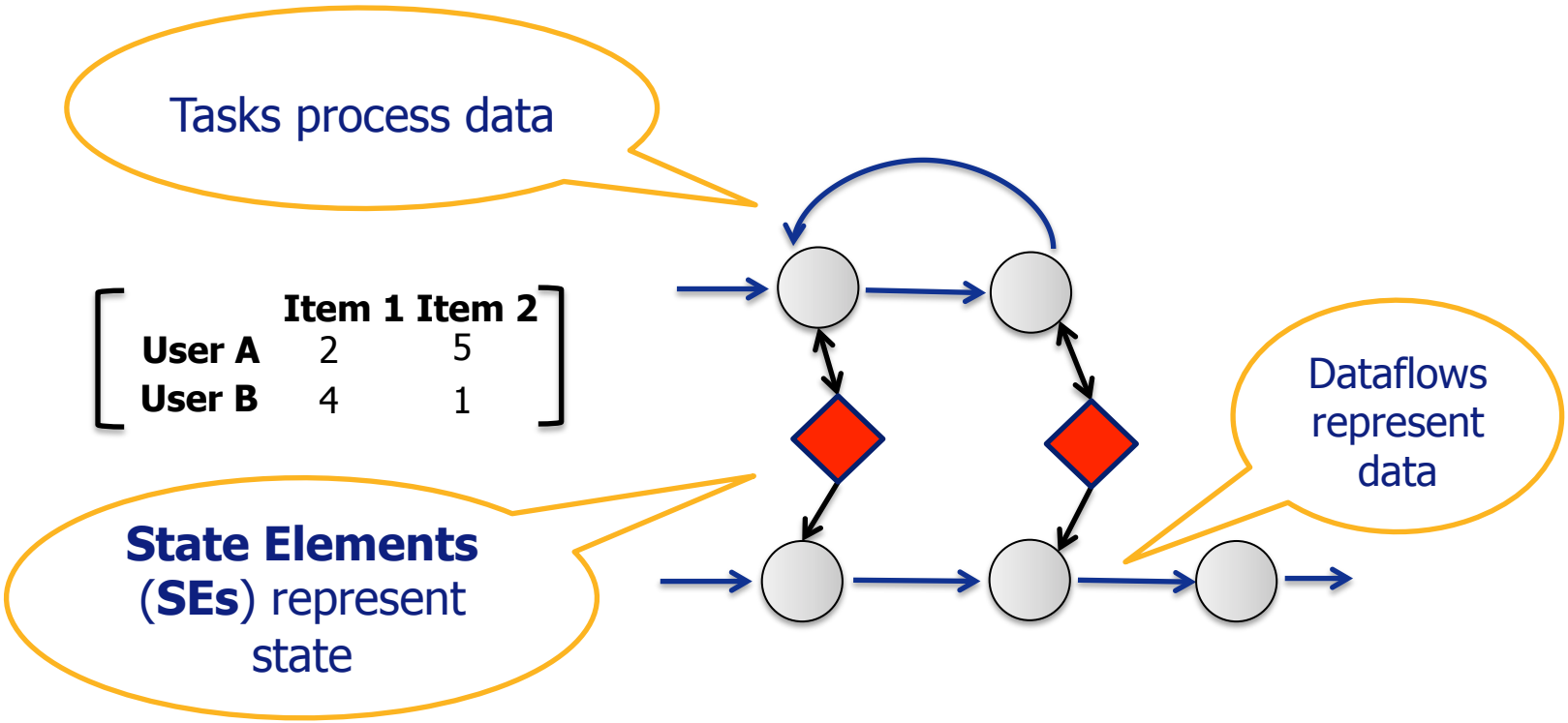
Dynamic scale out & checkpoint-based fault tolerance



4

Experimental evaluation results

State as First Class Citizen



Tasks have access to arbitrary state

State elements (SEs) represent in-memory data structures

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

Challenges with Large State

Mutable state leads to concise algorithms but complicates **scaling** and **fault tolerance**

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

Big Data problem: **Matrices become large**

State will not fit into single node

Challenge: Handling of distributed state?

Distributed Mutable State

State Elements support two abstractions for distributed mutable state:

Partitioned SEs:

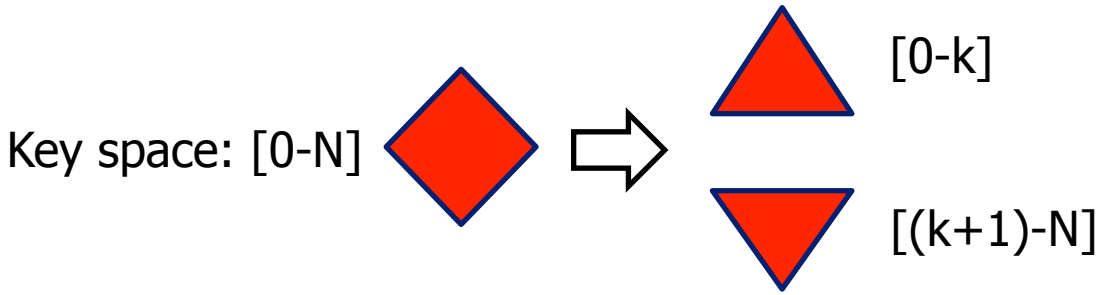
Tasks access **partitioned** state by key

Partial SEs:

Tasks can access **replicated** state

(I) Partitioned State Elements

Partitioned SE split into disjoint partitions

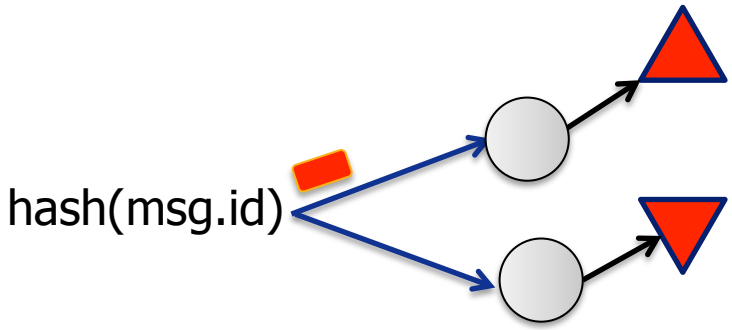


Access by key

User-Item matrix (UI)

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

State partitioned according to **partitioning key**

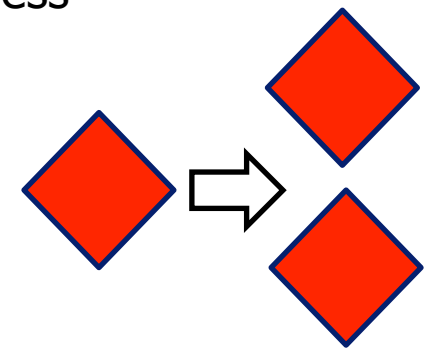


Dataflow routed according to **hash** function

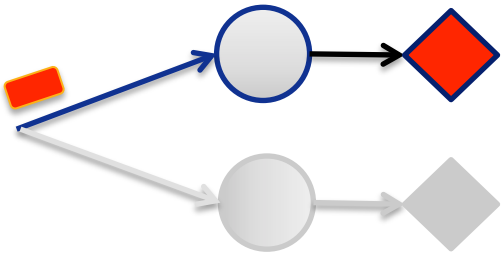
(II) Partial State Elements

Partial SEs are replicated (when partitioning is impossible)

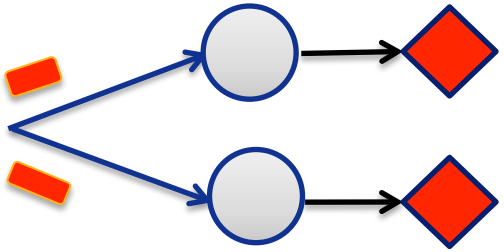
- Tasks have local access



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



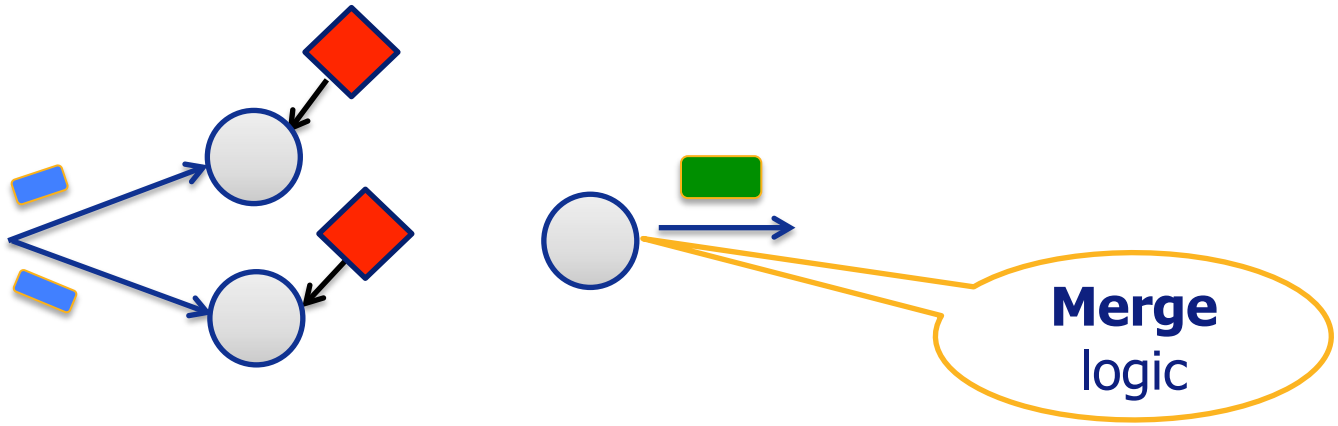
Local access:
Data sent to one



Global access:
Data sent to all

State Synchronisation with Partial SEs

Reading all partial SE instances results in set of partial values

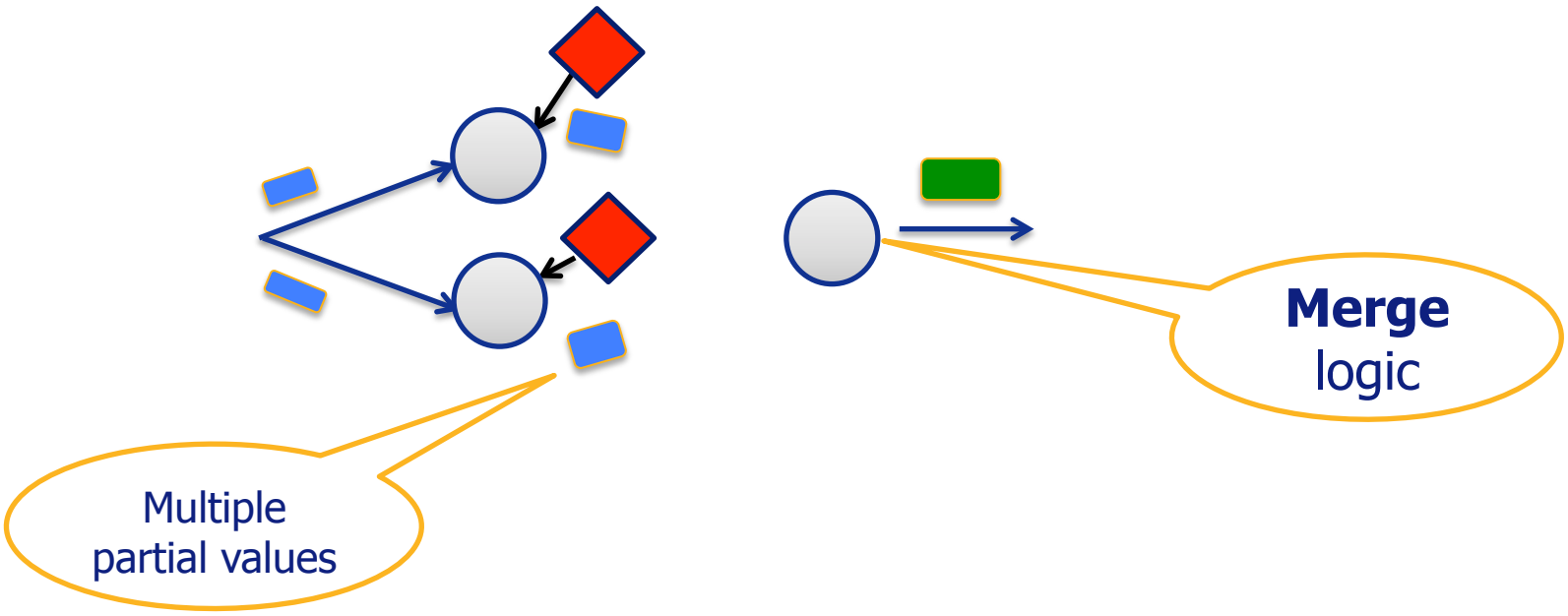


Requires application-specific **merge logic**

- Merge task reconciles state and updates partial SEs

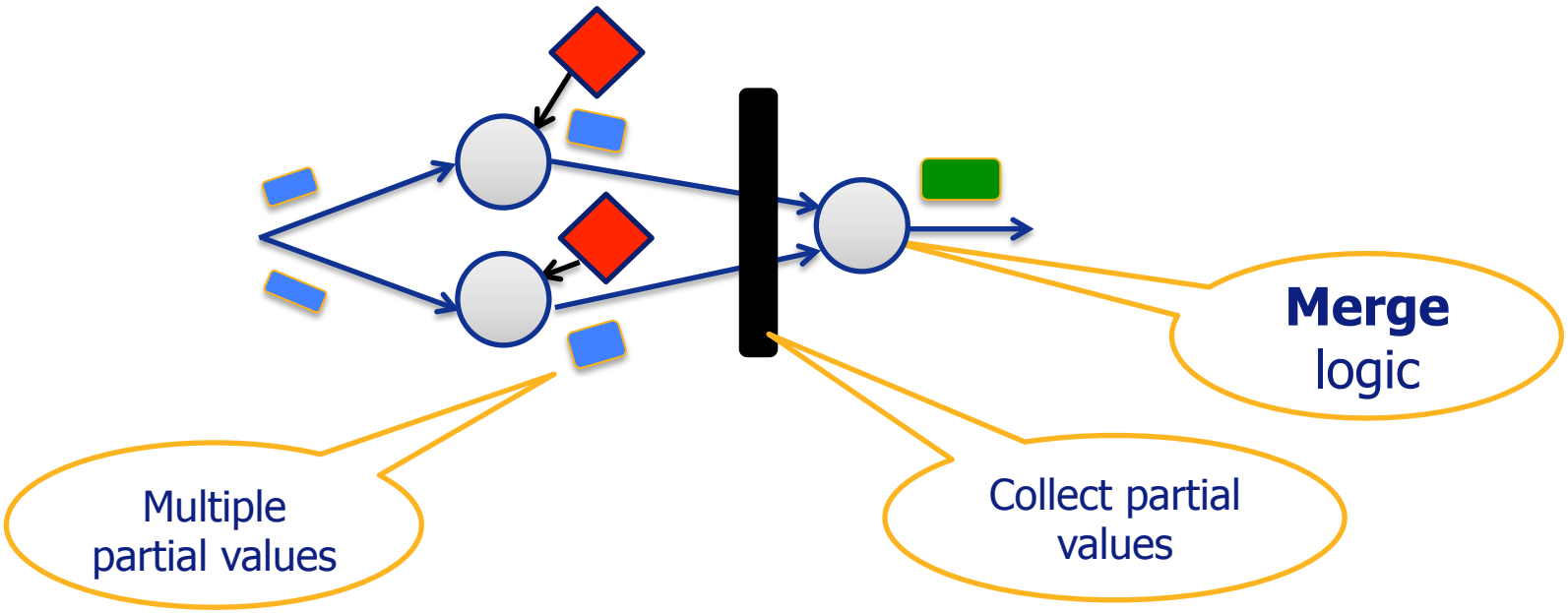
State Synchronisation with Partial SEs

Reading all partial SE instances results in set of partial values



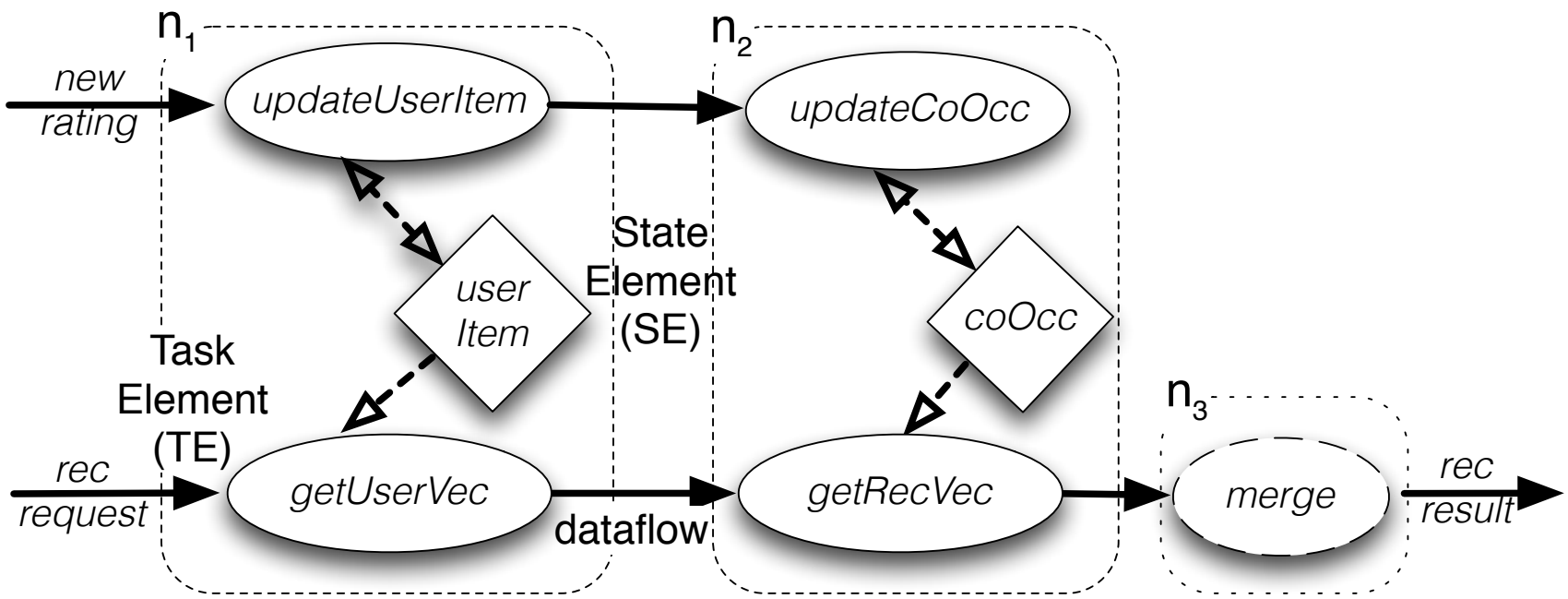
State Synchronisation with Partial SEs

Reading all partial SE instances results in set of partial values

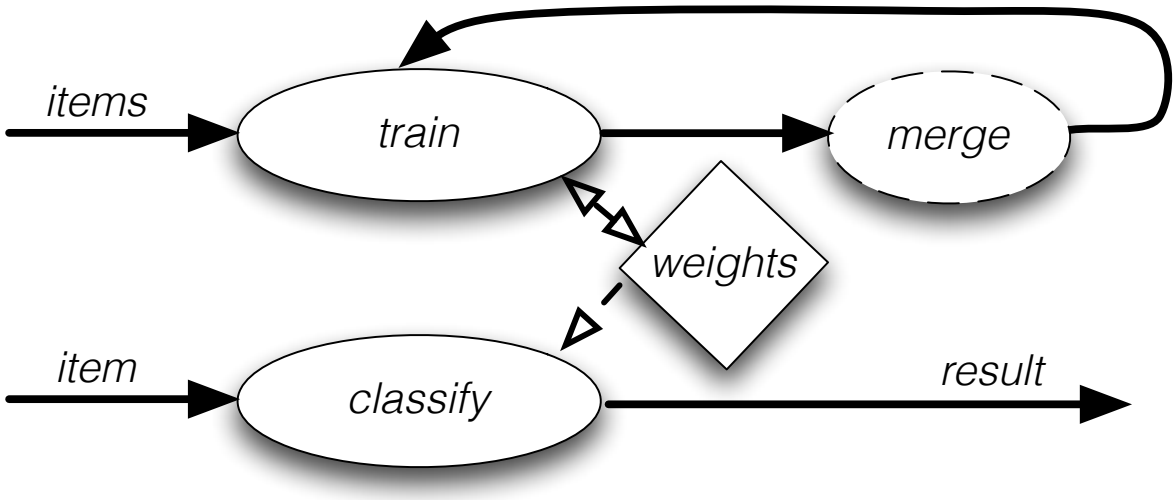


Barrier collects partial state

SDG for Collaborative Filtering

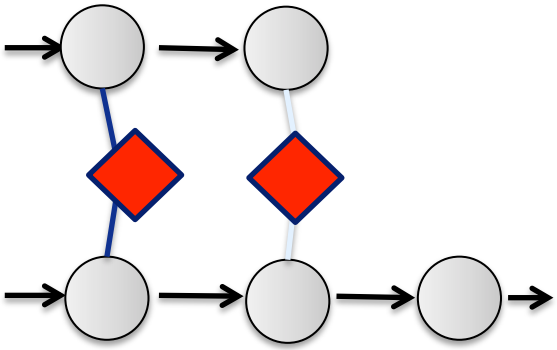


SDG for Logistic Regression



Requires support for iteration

Stateful Dataflow Graphs (SDGs)



2

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

Program.java

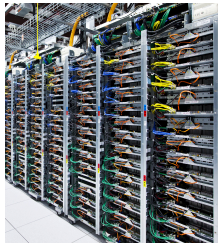
Static program analysis

Data-parallel **Stateful Dataflow Graph (SDG)**

SEEP distributed dataflow framework

Cluster

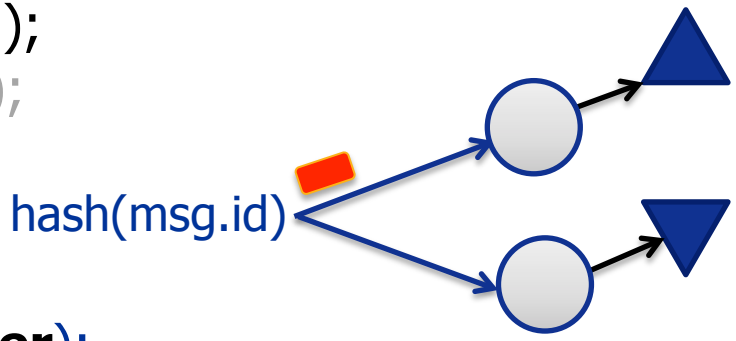
Dynamic scale out & checkpoint-based fault tolerance



Partitioned State Annotation

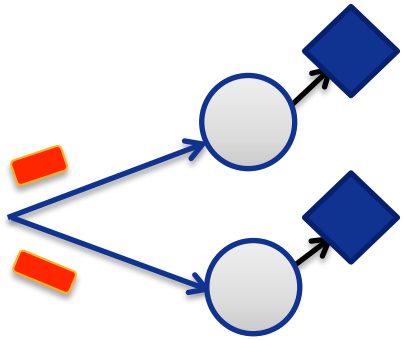
@Partition field annotation indicates **partitioned** state

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



Partial State and Global Annotations

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

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

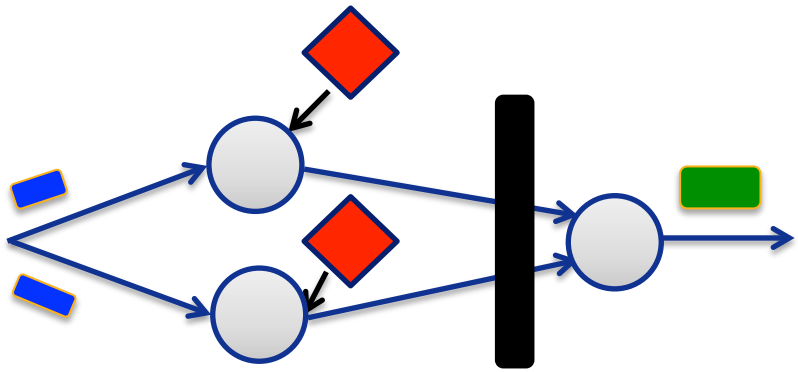


@Partial field annotation indicates **partial state**

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

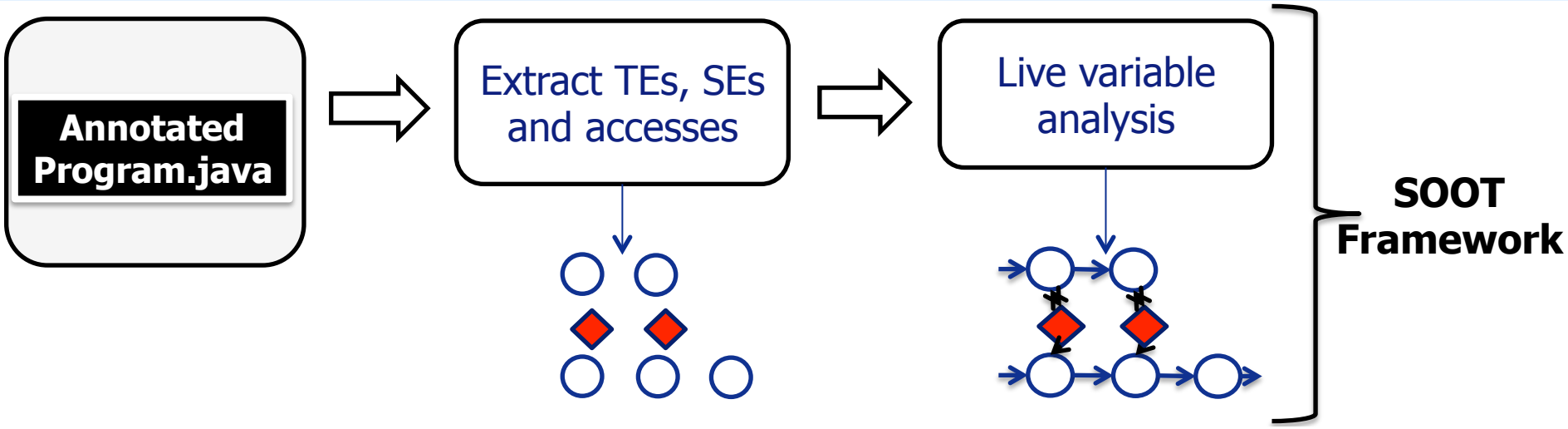
Partial and Collection Annotation

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

Vector getRec(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){
  /*...*/
}
```

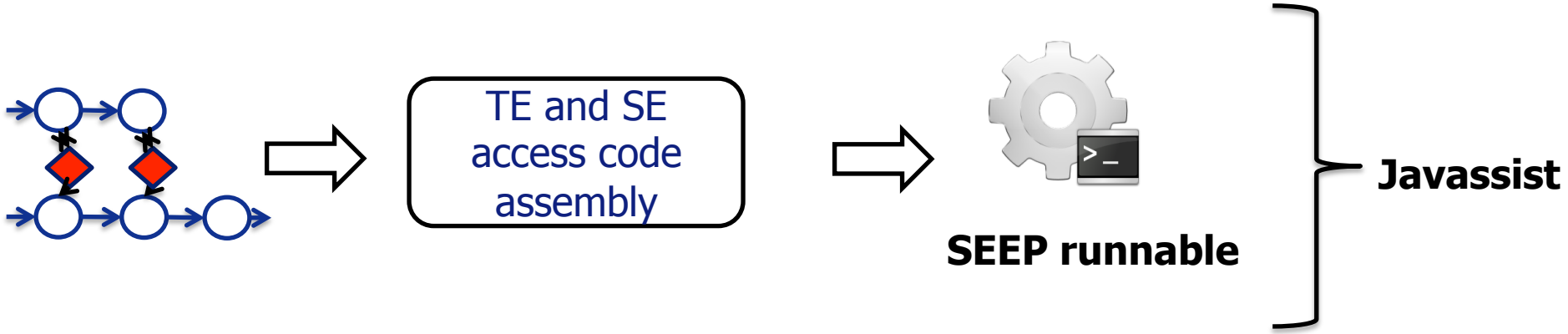


@Collection annotation indicates **merge logic**

Java2SDG: Translation Process

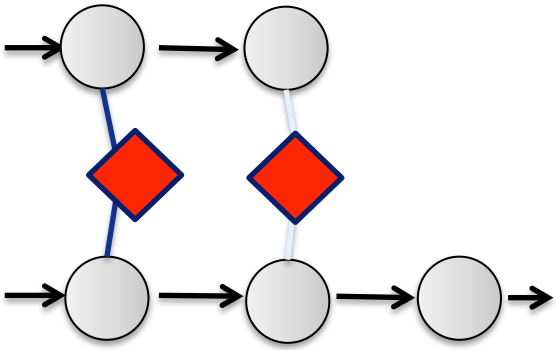


Extract **state** and **state access patterns** through static code analysis



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

Stateful Dataflow Graphs (SDGs)



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

Program.java

Static program analysis

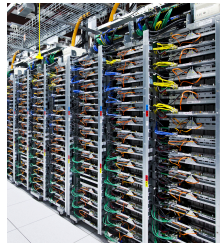
Data-parallel **Stateful Dataflow Graph (SDG)**

3

SEEP distributed dataflow framework

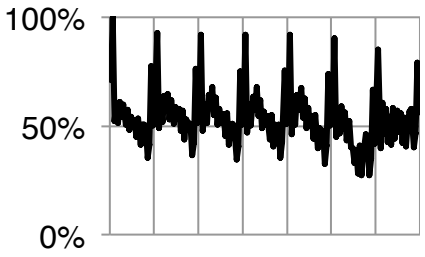
Cluster

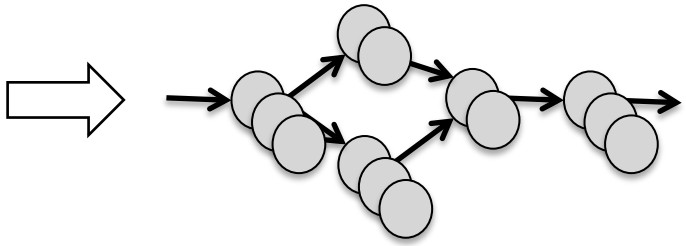
Dynamic scale out & checkpoint-based fault tolerance

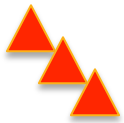


Scale Out and Fault Tolerance for SDGs

High/bursty input rates ➔ Exploit **data-parallelism**

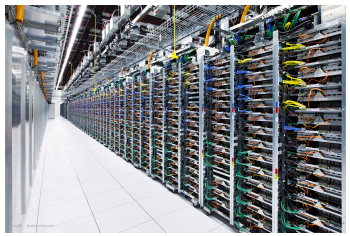


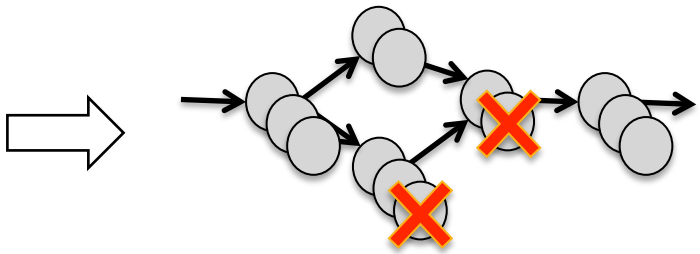




Partitioning of state

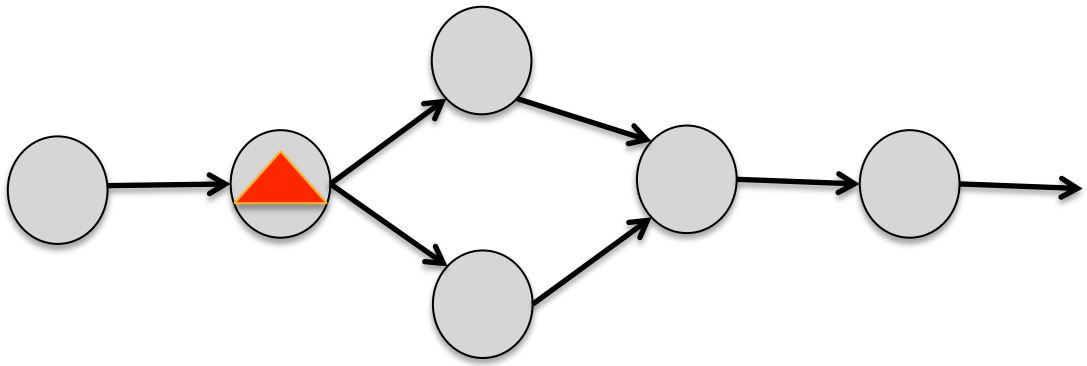
Large scale deployment ➔ Handle node **failures**





Loss of state after node failure

Dataflow Framework Managing State



- Expose state as external entity to be managed by the distributed dataflow framework

Framework has state management primitives to:

- Backup and recover state elements
- Partition state elements

Integrated mechanism for scale out and failure recovery

- Node recovery and scale out with state support

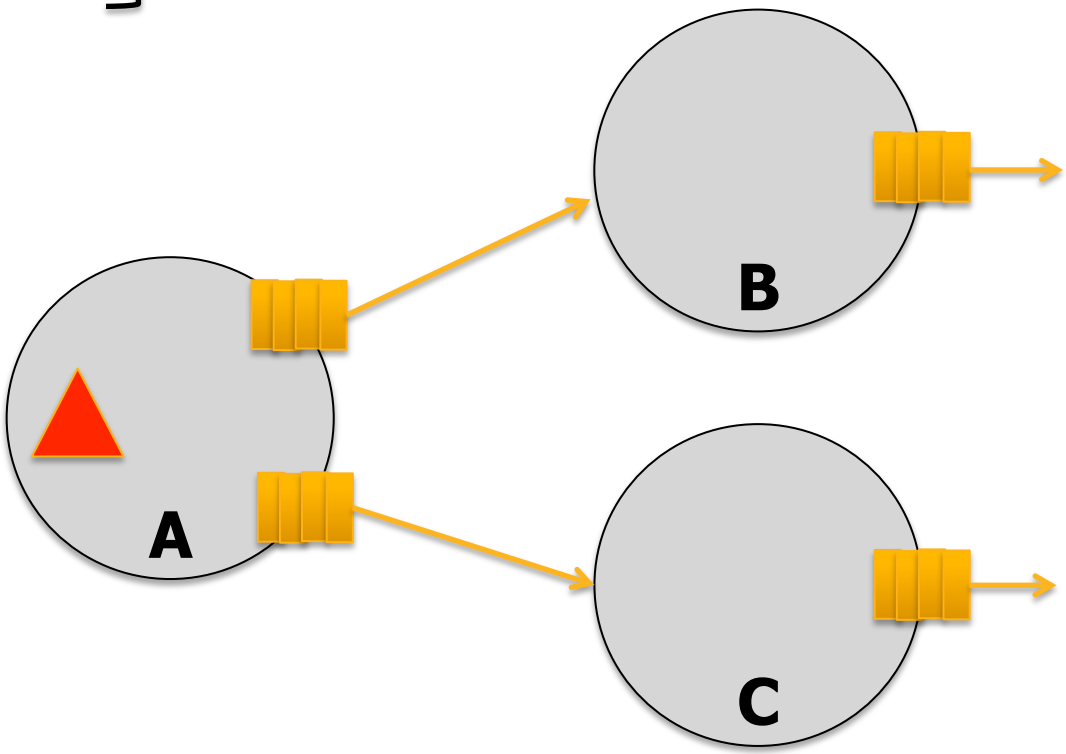
What is State?

Processing state

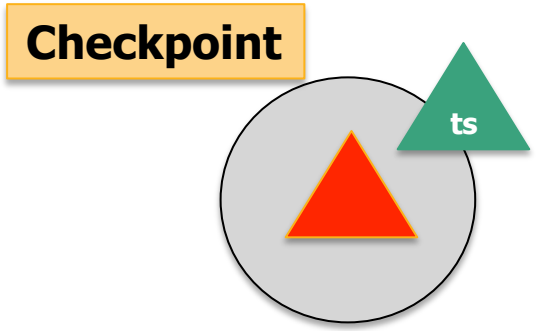
	Item 1	Item 2
User A	2	5
User B	4	1

Buffer state

Data ts1	Data ts2	Data ts3	Data ts4

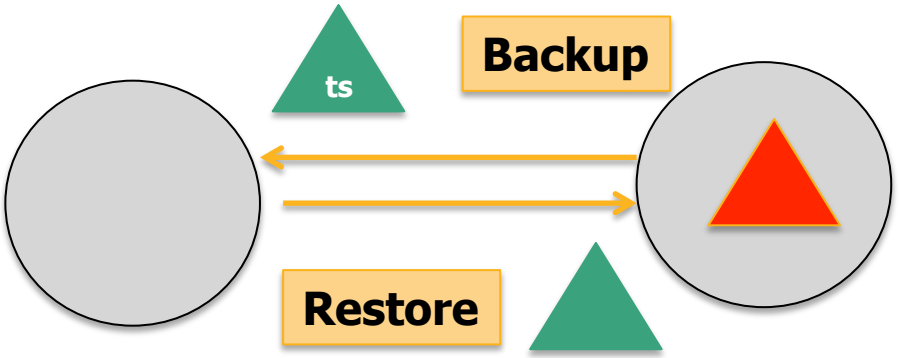


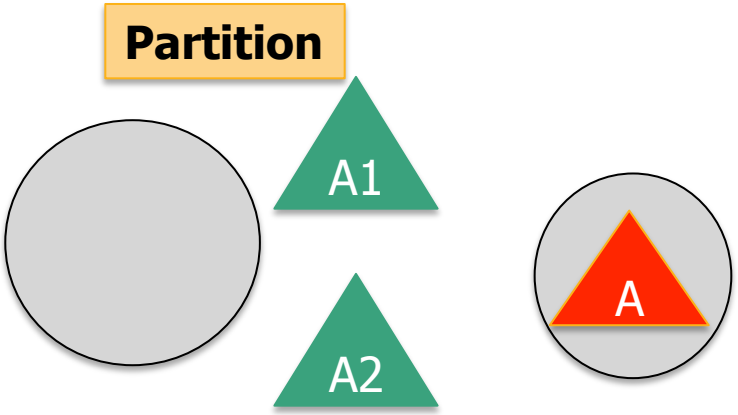
State Management Primitives



- Makes state available to framework
- Attaches **last processed data timestamp**

- Moves copy of state from one node to another



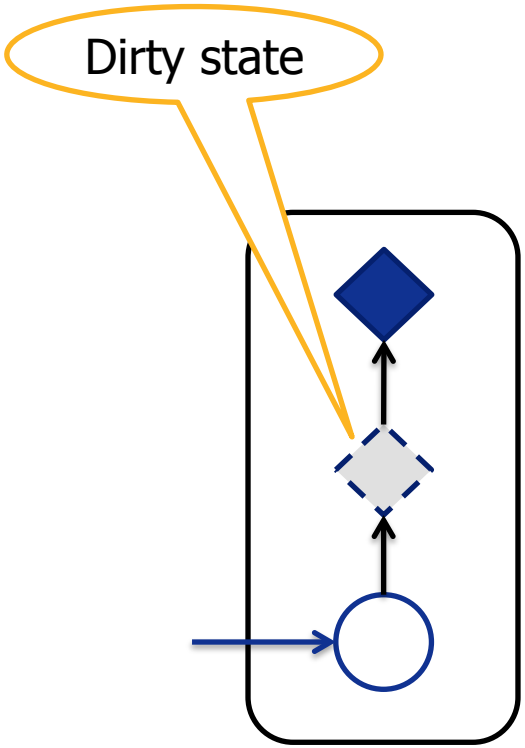


- Splits state to scale out tasks

State Primitive: Checkpointing

Challenge: Efficient checkpointing of large state in Java?

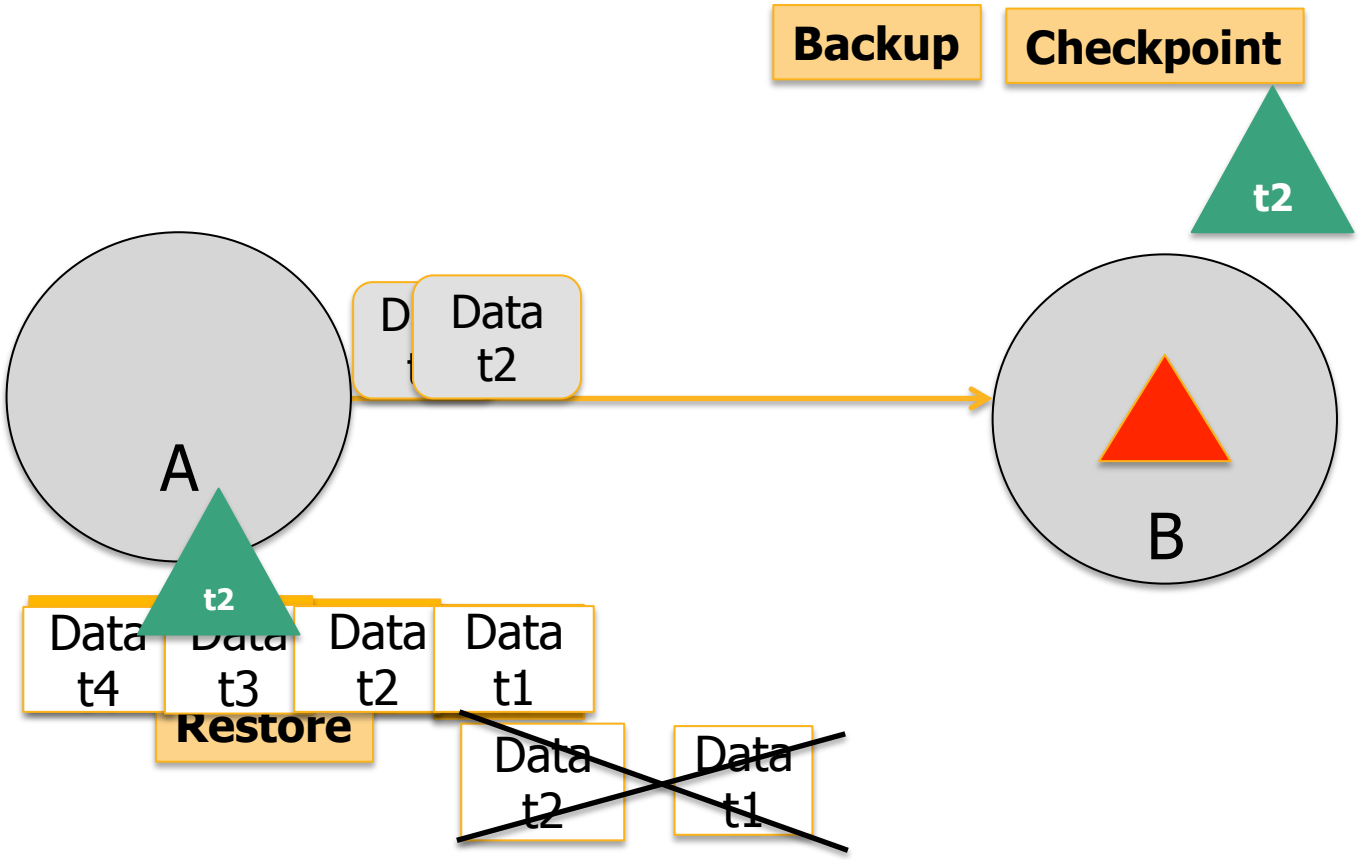
- No updates allowed while state is being checkpointed
- Checkpointing state should not impact data processing path



Asynchronous, lock-free checkpointing

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

State Primitives: Backup and Restore

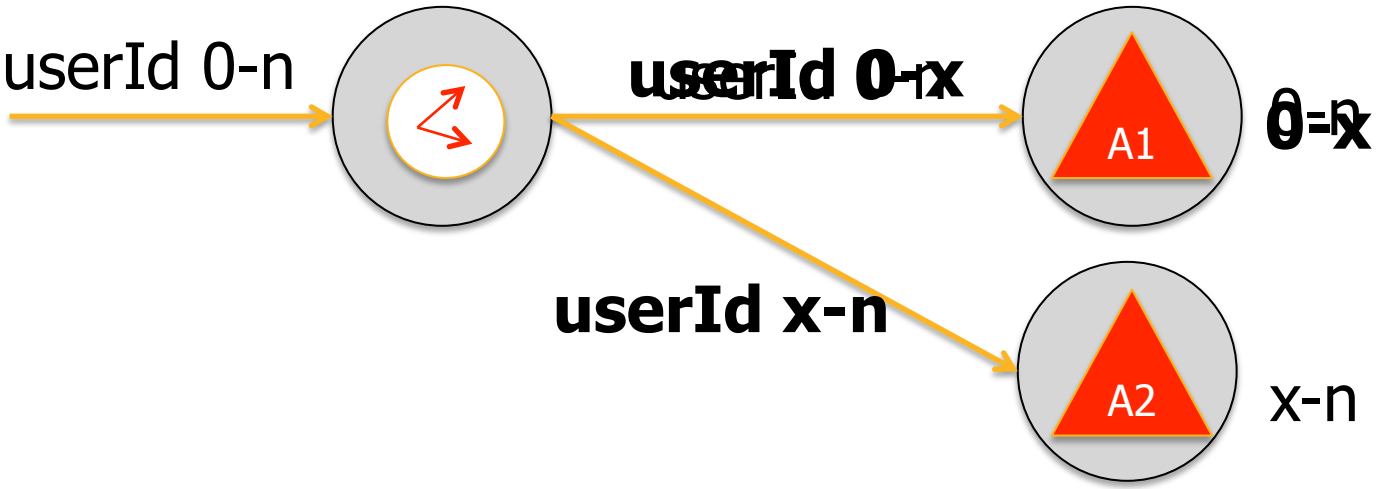


State Primitives: Partition

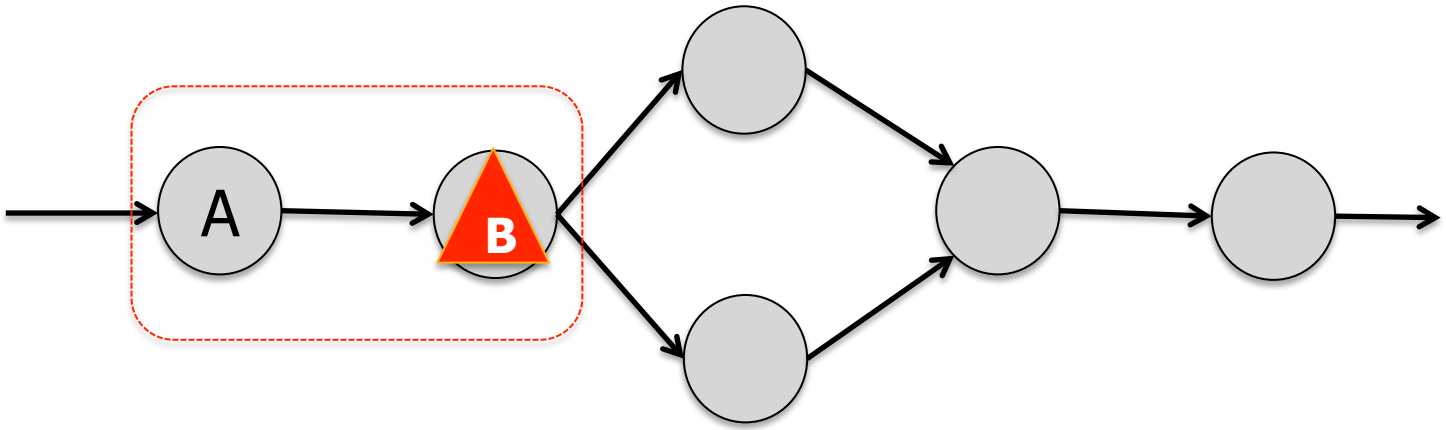
Processing state modeled as (key, value) dictionary

State partitioned according to **key** k

- Same key used to partition streams



Failure Recovery and Scale Out



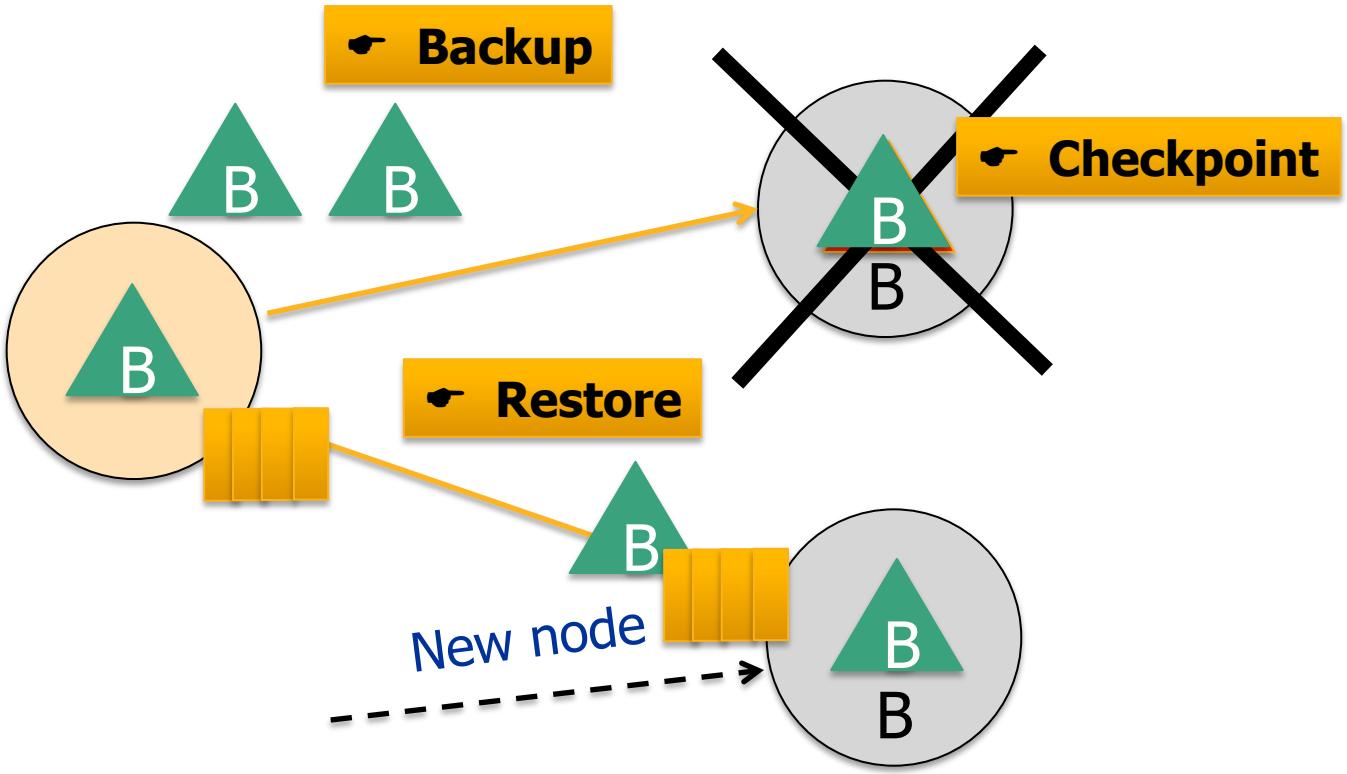
Two cases:

- Node B **fails ➔ Recover**
- Node B becomes **bottleneck ➔ Scale out**

Recovering Failed Nodes

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

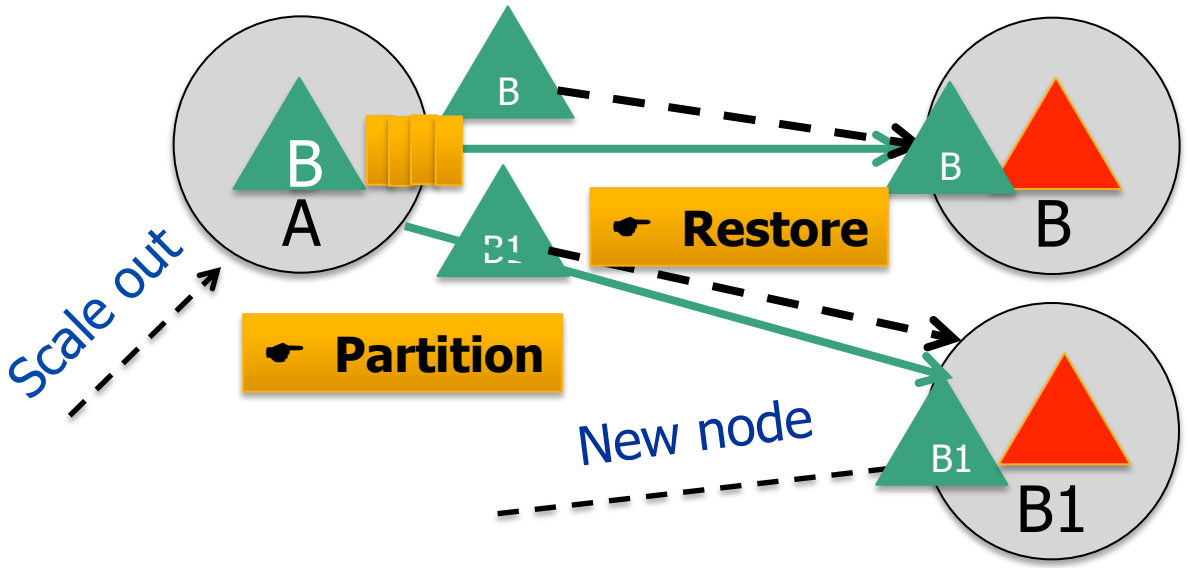
Use backed up state to recover quickly



State restored and unprocessed data replayed from buffer

Scaling Out Tasks

Finally, upstream node replays unprocessed data to update checkpointed state

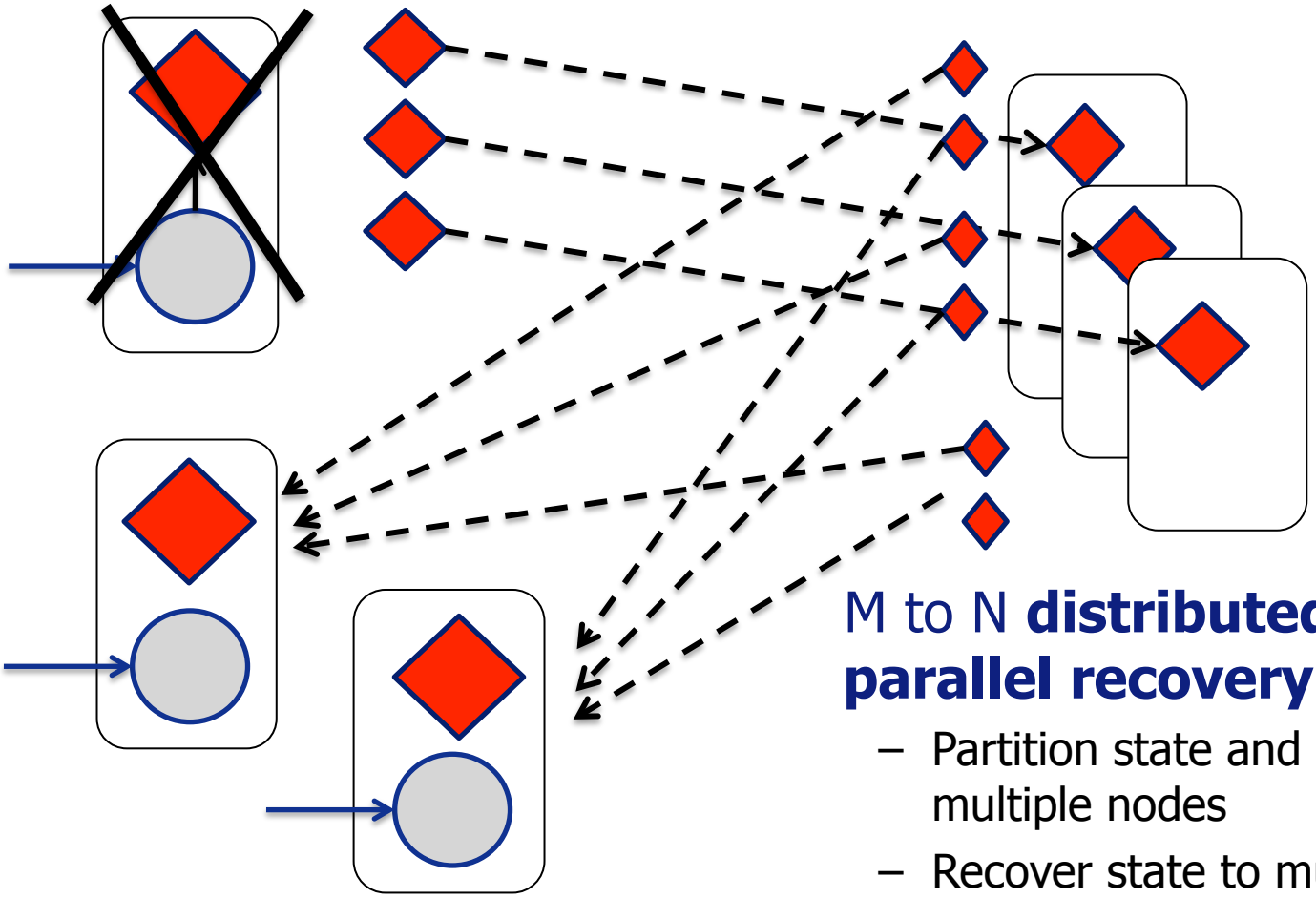


For scale out, backup node already has state elements to be parallelised

Distributed M-to-N Backup/Recovery

Challenge: Fast recovery?

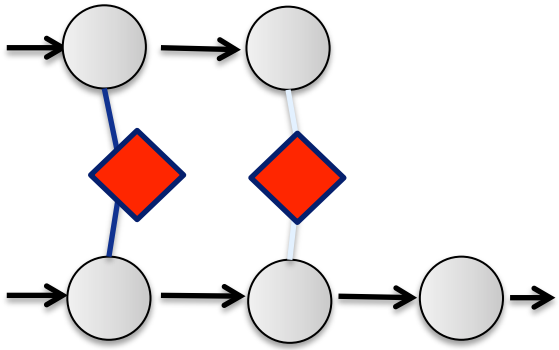
- Backups large and cannot be stored in memory
- Large writes to disk through network have high cost



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

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

Stateful Dataflow Graphs (SDGs)



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

SEEP distributed dataflow framework

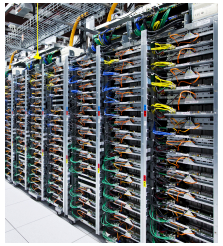
Program.java

Static program analysis

Data-parallel **Stateful Dataflow Graph (SDG)**

Dynamic scale out & checkpoint-based fault tolerance

Cluster

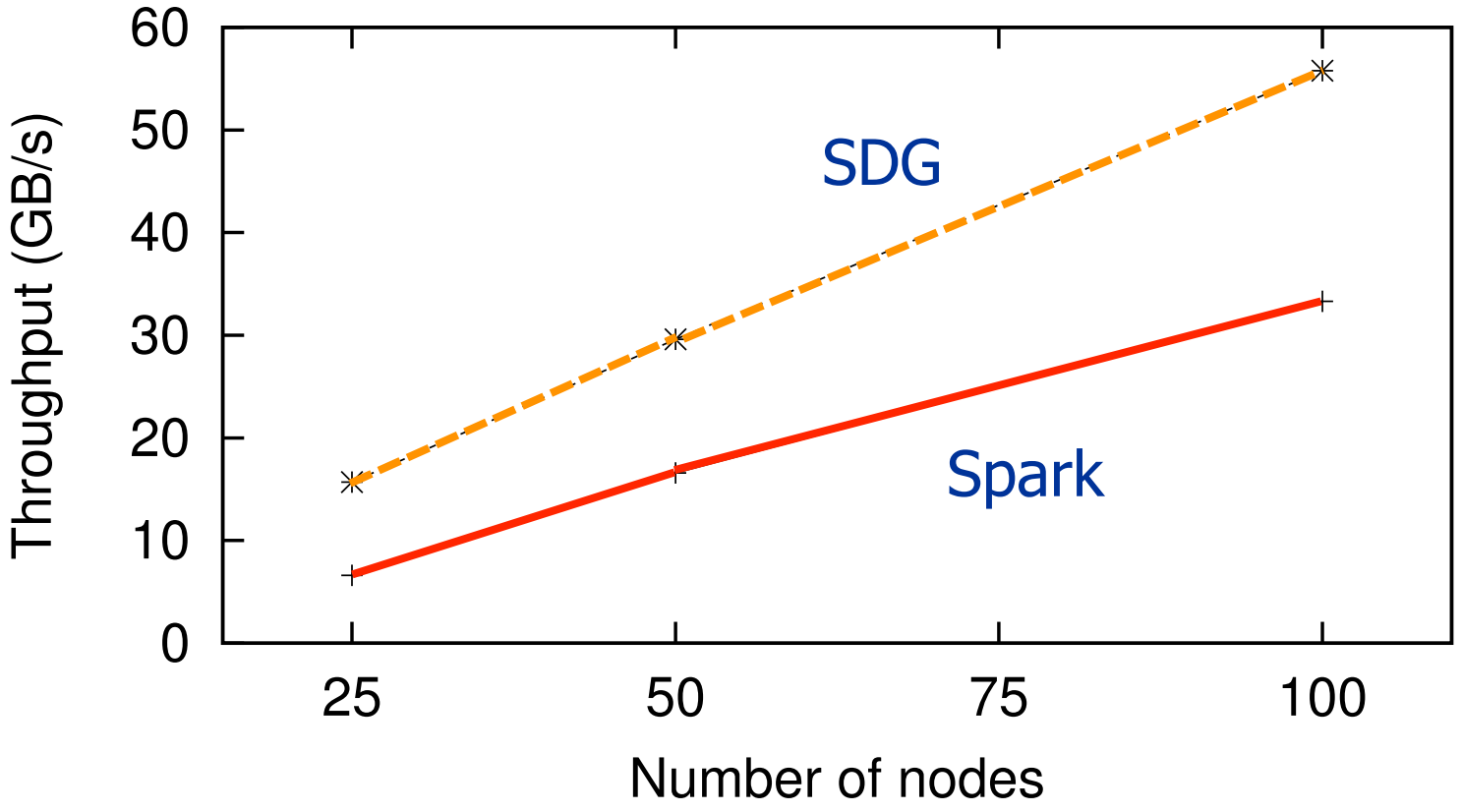


4 Experimental evaluation results

Throughput: Logistic Regression

100 GB training dataset for classification

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

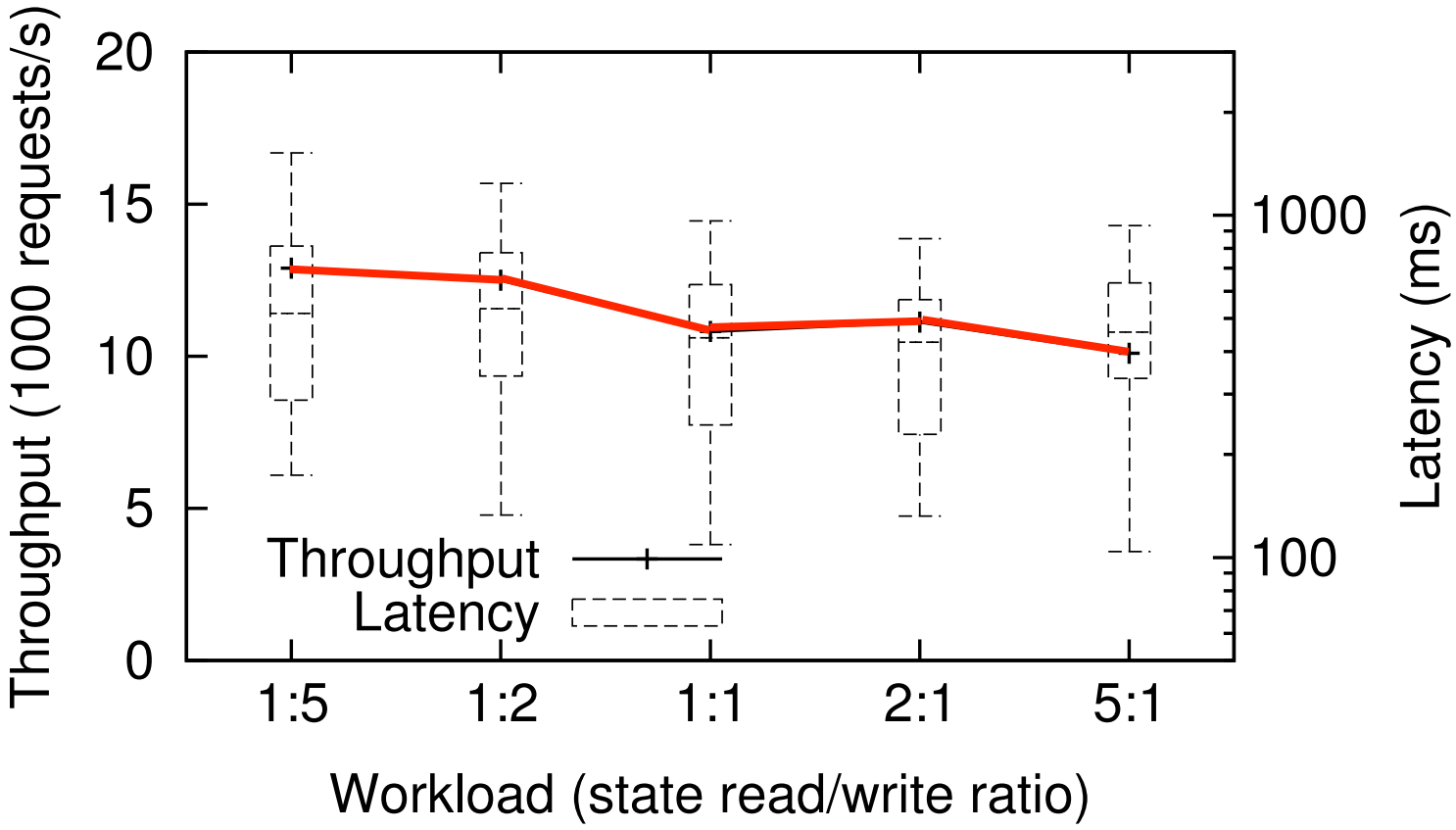


SDGs have comparable throughput to Spark despite mutable state

Mutable State Access: Collaborative Filtering

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

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

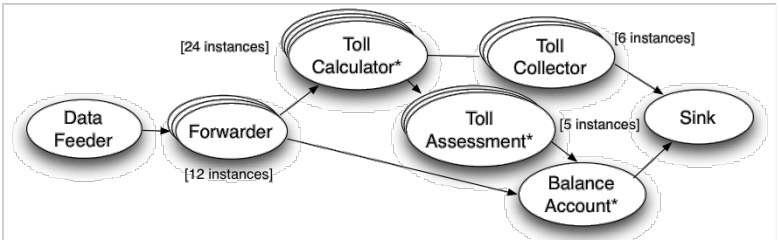


SDGs serve fresh results over large mutable state

Elasticity: Linear Road Benchmark

Linear Road Benchmark [VLDB'04]

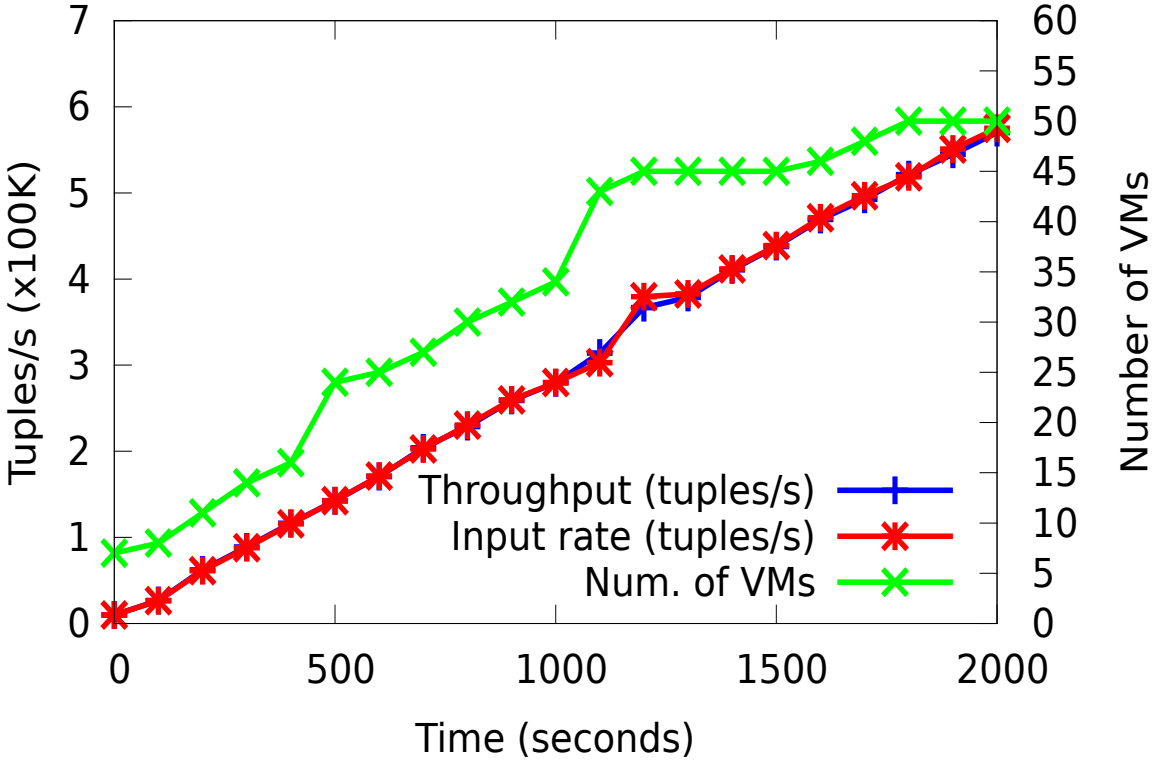
- Network of toll roads of size L
- Input rate increases over time
- SLA: results < 5 secs



Deployed on Amazon EC2 (c1 & m1 xlarge instances)

Scales to L=350 with 60 VMs

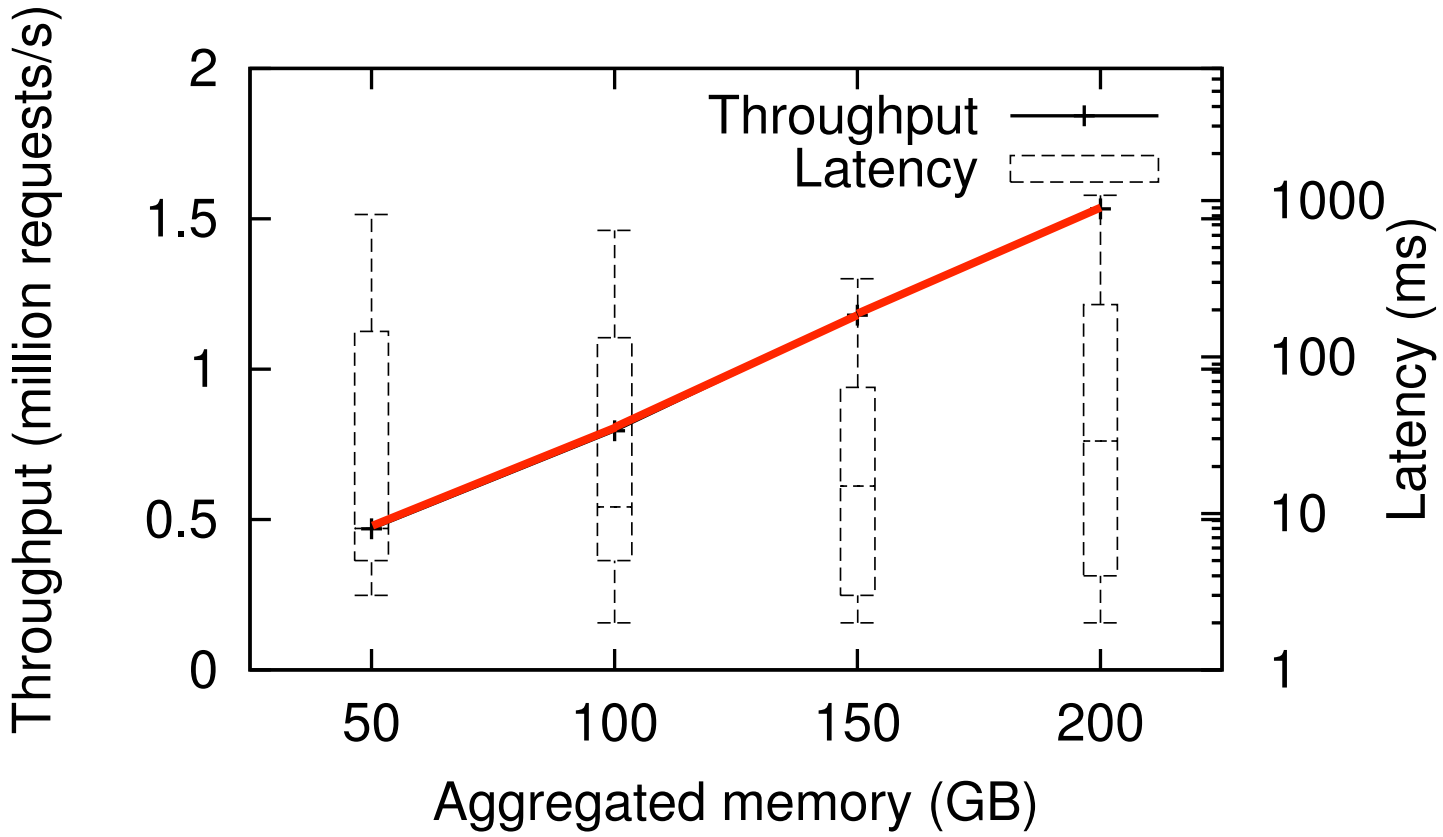
L=512 highest reported result in literature [VLDB'12]



SDGs can scale dynamically based on workload

Large State Size: Key/Value Store

Increase state size in distributed key/value store



SDGs can support online services with mutable state

Summary

Programming models for Big Data matter

- Logic increasingly pushed into bespoke APIs
- Existing models do not support fine-grained mutable state

Stateful Dataflow Graphs support mutable state

- Automatic translation of annotated Java programs to SDGs
- SDGs introduce new challenges in terms of parallelism and failure recovery
- Automatic state partitioning and checkpoint-based recovery

SEEP available on GitHub: https://github.com/lsds/Seep/

Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch, **"Integrating Scale Out and Fault Tolerance in Stream Processing using Operator State Management"**, **SIGMOD'13**

Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch, **"Making State Explicit for Imperative Big Data Processing"**, **USENIX ATC'14**

Thank you! Any Questions?

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