Making State Explicit for Imperative Big Data Processing

Raul Castro Fernandez
Imperial College London
rc3011@doc.ic.ac.uk

Matteo Migliavacca
University of Kent
mm53@kent.ac.uk

Eva Kalyvianaki
City University London
evangelia.kalyvianaki.1@city.ac.uk

Peter Pietzuch
Imperial College London
prp@doc.ic.ac.uk

USENIX Annutal Technical Conference 2014

Mutable State in a Recommender System

```
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;
}
```

User-Item matrix (**UI**)

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

Update with new ratings

Co-Occurrence matrix (**CO**)

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

User-B	1	2

x

Multiply for recommendation

Challenges When Executing with Big Data

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

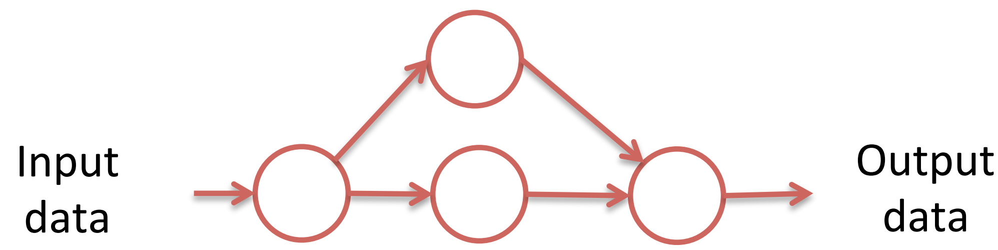
```
Matrix userItem = new Matrix();
Matrix coOcc = new Matrix();
```

Big Data Problem:
Matrices become large

> **Cannot lose state after failure**

> **Need to manage state to support data-parallelism**

Using Current Distributed Dataflow Frameworks



> No mutable state **simplifies fault tolerance**

> **MapReduce**: Map and Reduce tasks

> **Storm:** No support for state

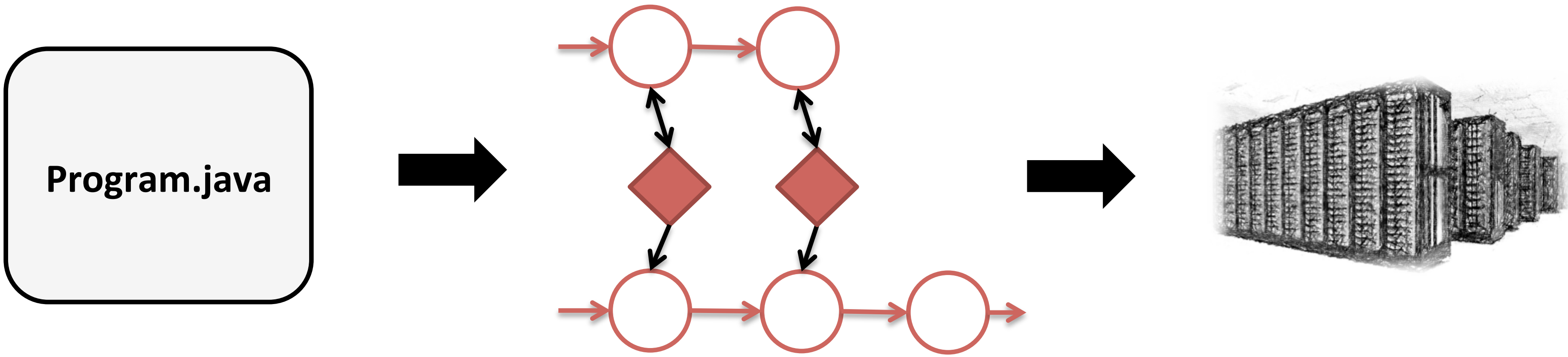
> **Spark:** Immutable RDDs

Imperative Big Data Processing

> Programming distributed dataflow graphs **requires learning new programming models**

Our Goal:
Run Java **programs with mutable state** but with **performance** and **fault tolerance** of **distributed dataflow systems**

Stateful Dataflow Graphs: From Imperative Programs to Distributed Dataflows



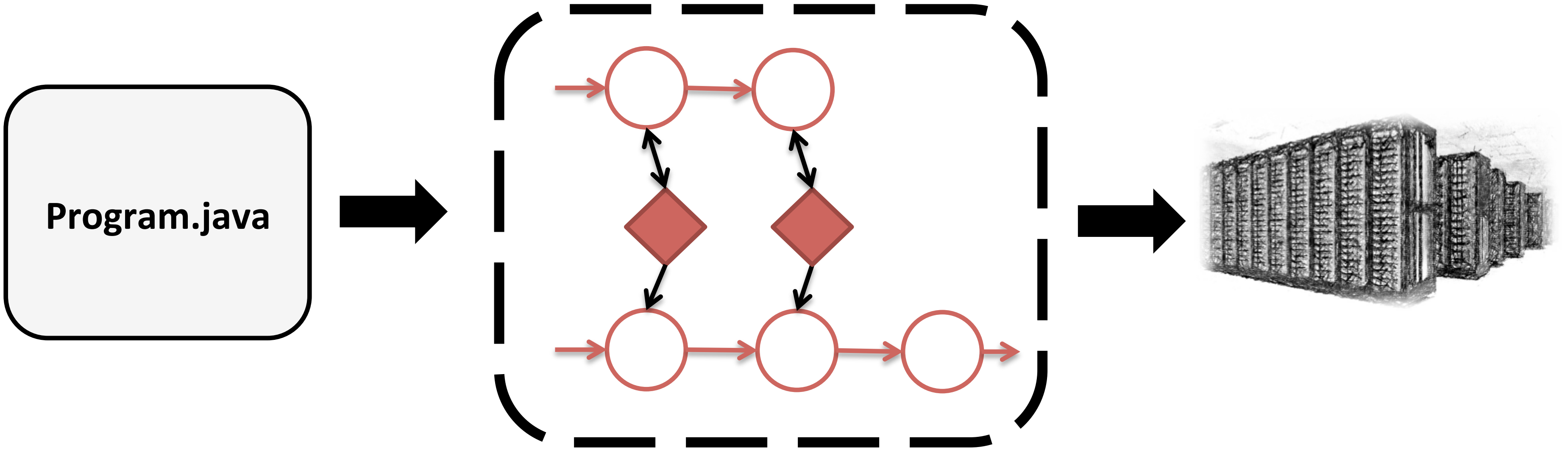
SDGs: Stateful Dataflow Graphs

> Mutable distributed state in dataflow graphs

> @Annotations help with translation from Java to SDGs

> Checkpoint-based fault tolerance recovers mutable state after failure

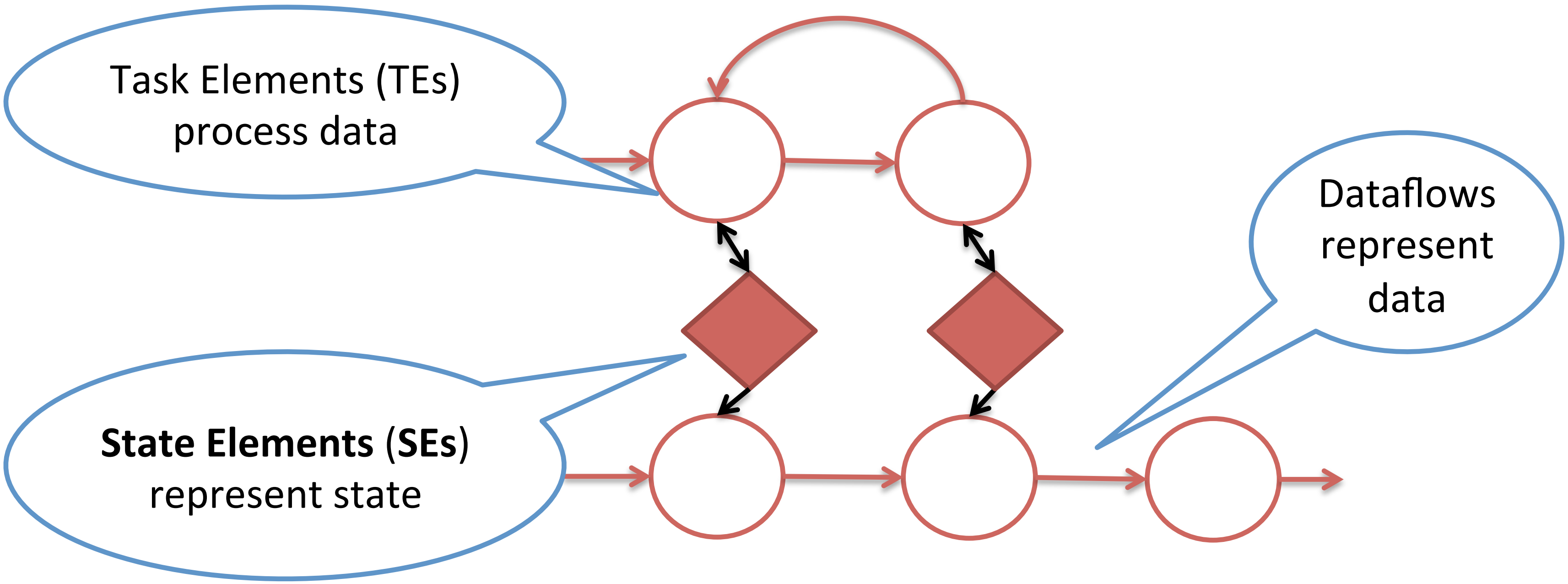
Outline



- **SDG: Stateful Dataflow Graphs**
- Handling distributed state in SDGs
- Translating Java programs to SDGs
- Checkpoint-based fault tolerance for SDGs
- Experimental evaluation

SDG: Data, State and Computation

> SDGs separate **data and state** to allow **data** and **pipeline parallelism**



> Task Elements have **local access** to State Elements

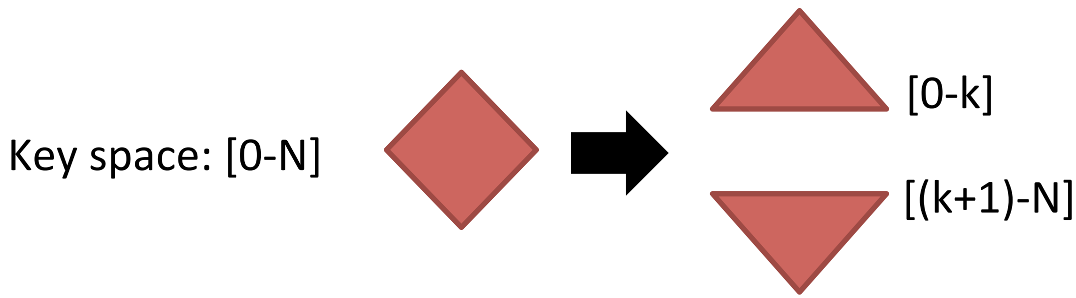
Distributed Mutable State

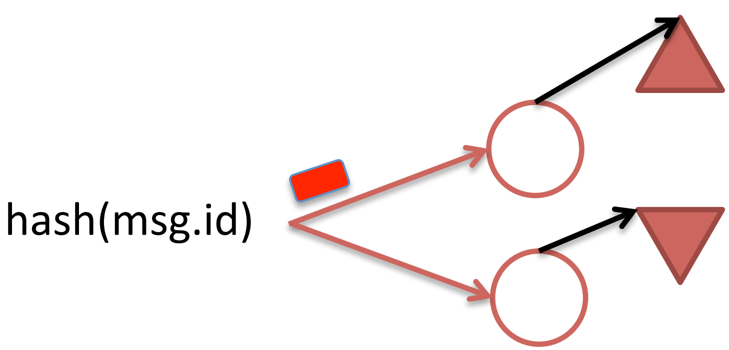
State Elements support two abstractions for **distributed mutable state**

- **Partitioned SEs:** task elements always access state by key
- **Partial SEs:** task elements can access complete state

Distributed Mutable State: Partitioned SEs

> **Partitioned** SEs split into disjoint partitions





Dataflow routed according to **hash** function

User-Item matrix (***UI***)

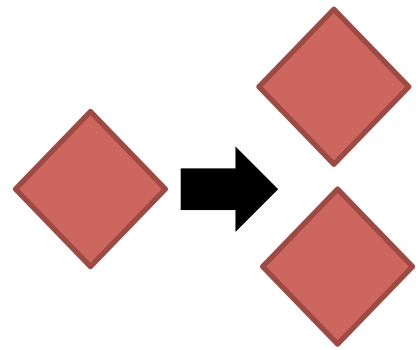
Access by key

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

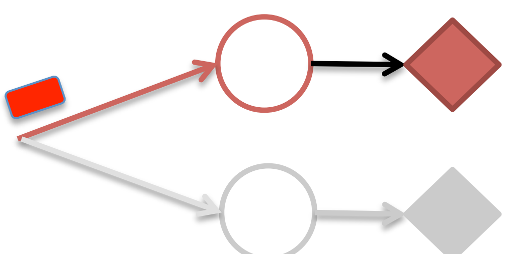
State partitioned according to **partitioning key**

Distributed Mutable State: Partial SEs

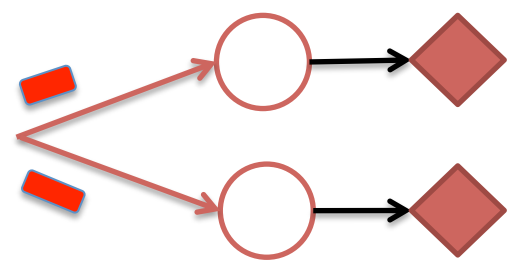
> **Partial** SE gives nodes local state instances



> **Partial** SE access by Tes can be *local* or *global*



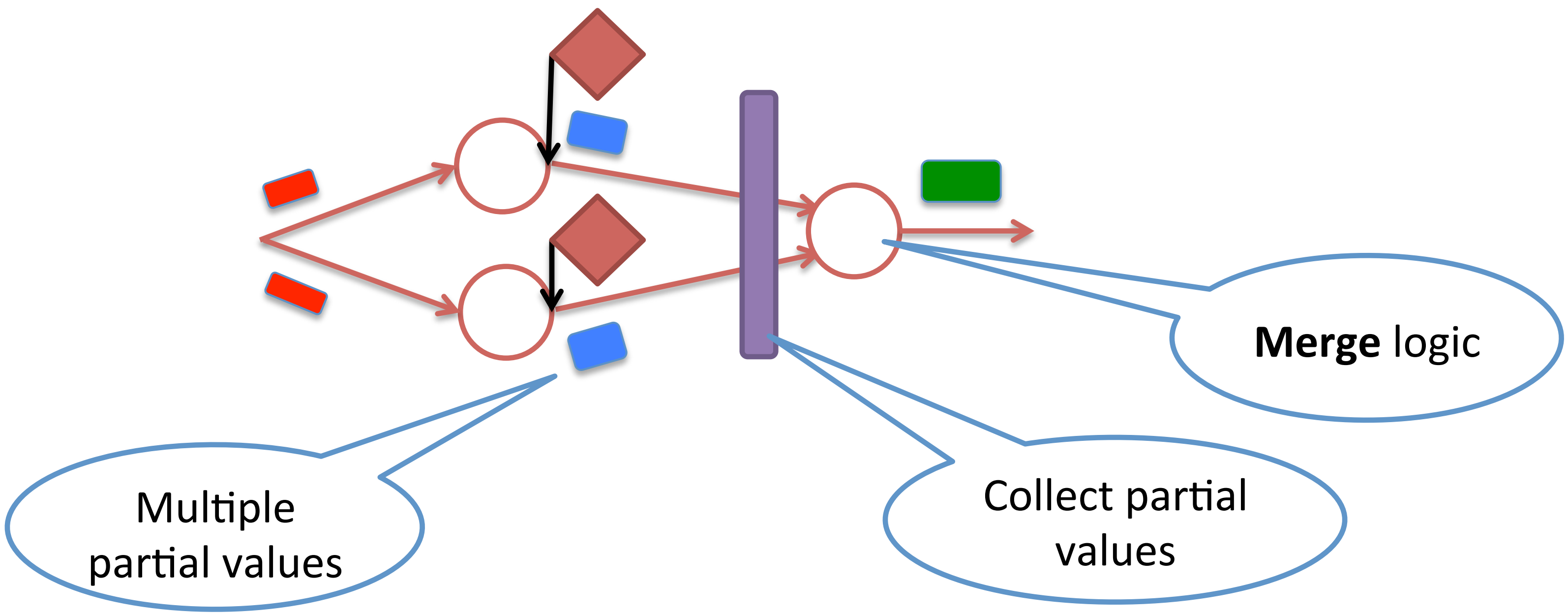
Local access:
Data sent to one



Global access:
Data sent to all

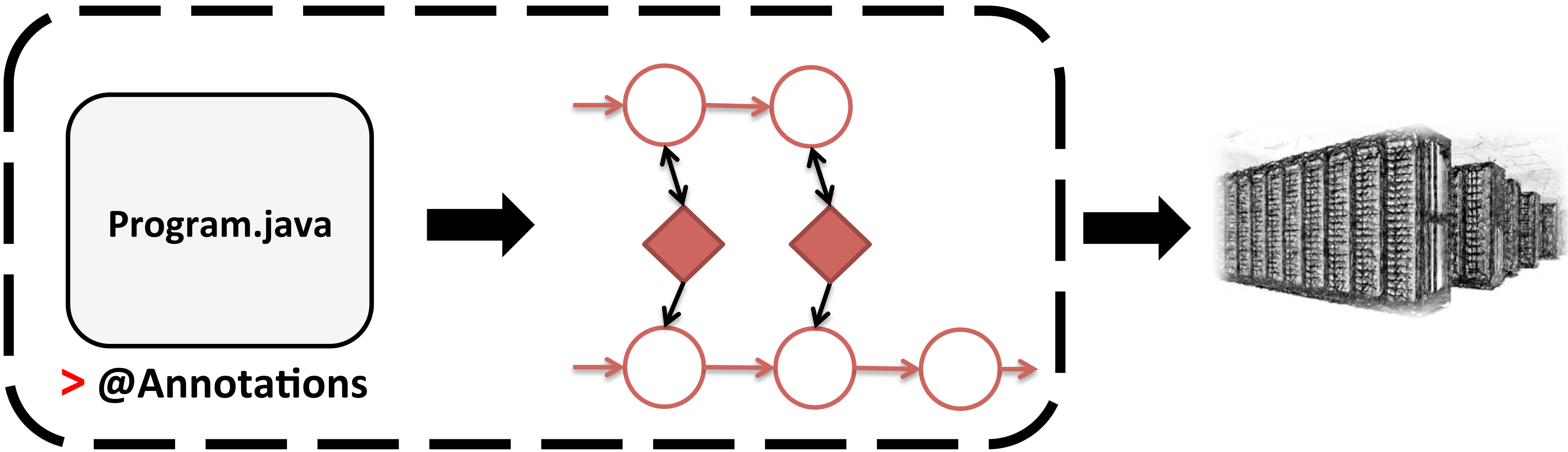
Merging Distributed Mutable State

> Reading all partial SE instances results in set of **partial** values



> Requires application-specific merge logic

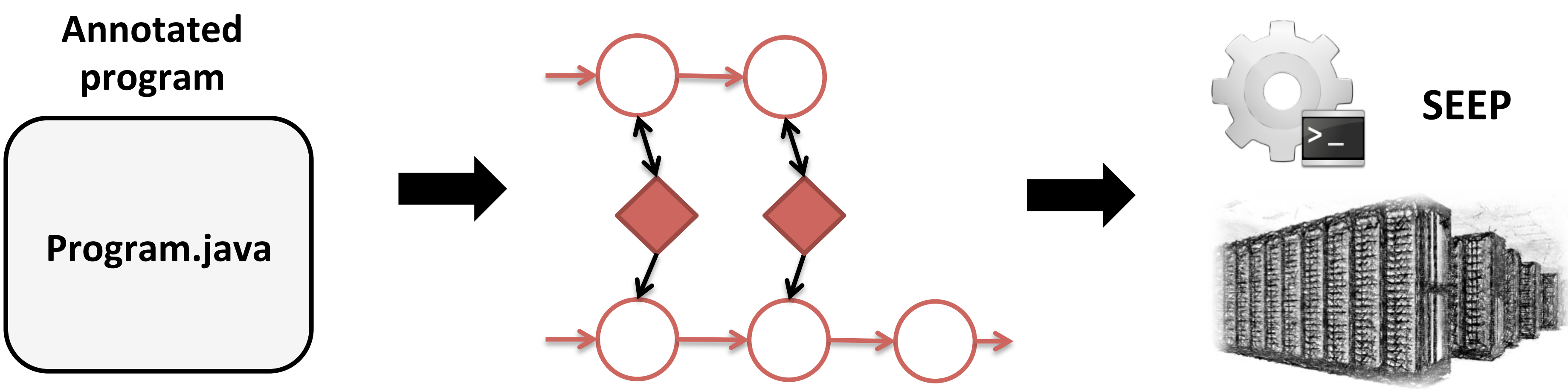
Outline



- SDG: Stateful Dataflow Graphs
- Handling distributed state in SDGs
- **Translating Java programs to SDGs**
- Checkpoint-based fault tolerance for SDGs
- Experimental evaluation

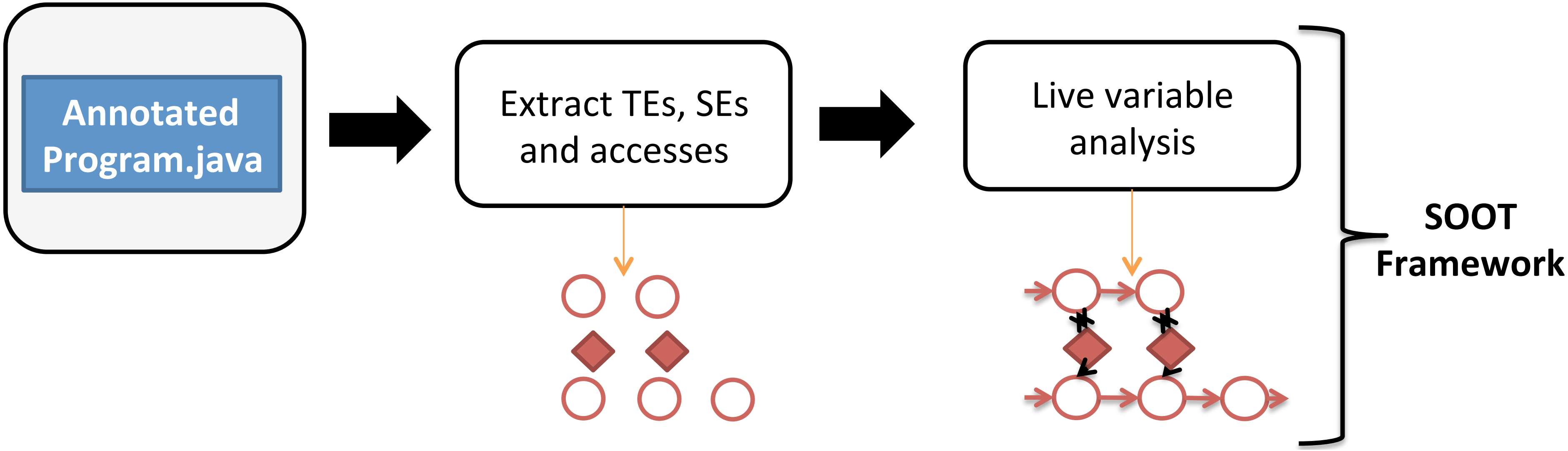
From Imperative Code to Execution

- Translation occurs in two stages:
  - *Static code analysis*: From Java to SDG
  - *Bytecode rewriting*: From SDG to *SEEP* [SIGMOD'13]

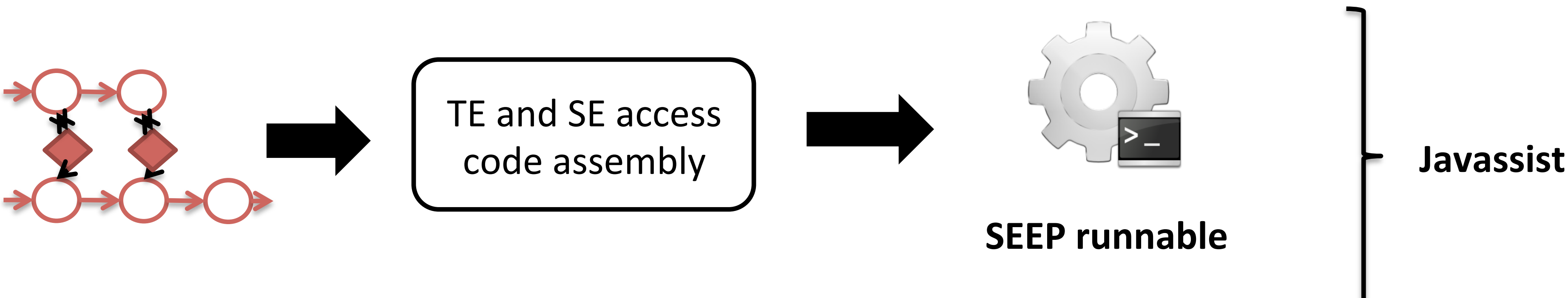


> SEEP: **data-parallel processing platform**

Translation Process



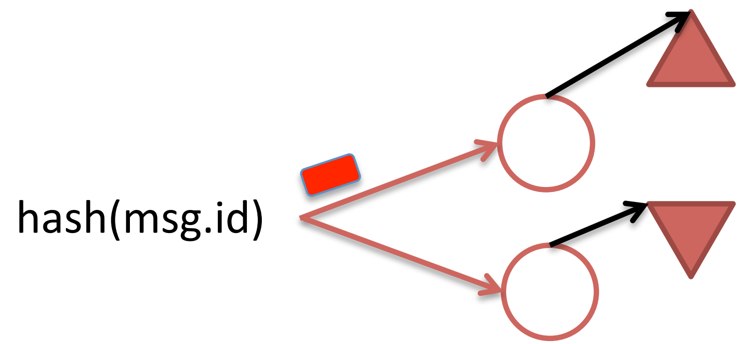
> Extract **state** and **state access patterns** through static code analysis



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

Partitioned State Annotation

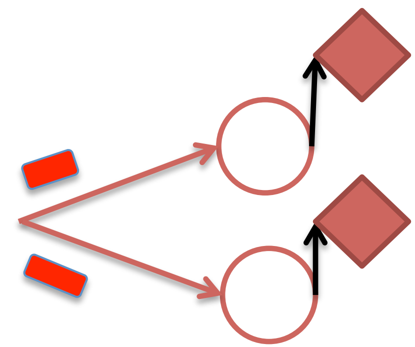
```
@Partitioned Matrix userItem = new SeepMatrix();
Matrix coOcc = new Matrix();

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;
}
```



> **@Partition field annotation** indicates *partitioned* state

Partial State and Global Annotations

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

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

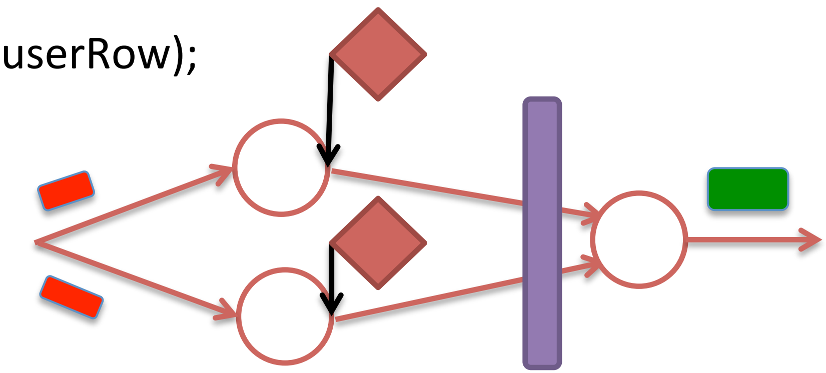


> **@Partial field annotation** indicates *partial* state

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

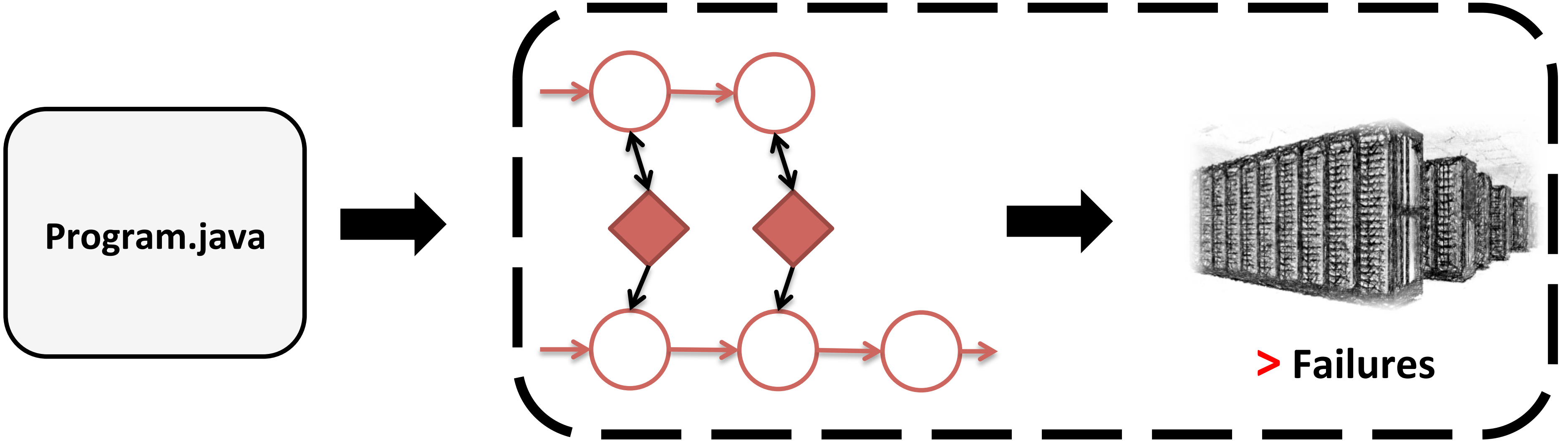
Partial and Collection Annotations

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

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){
  /*…*/
}
```



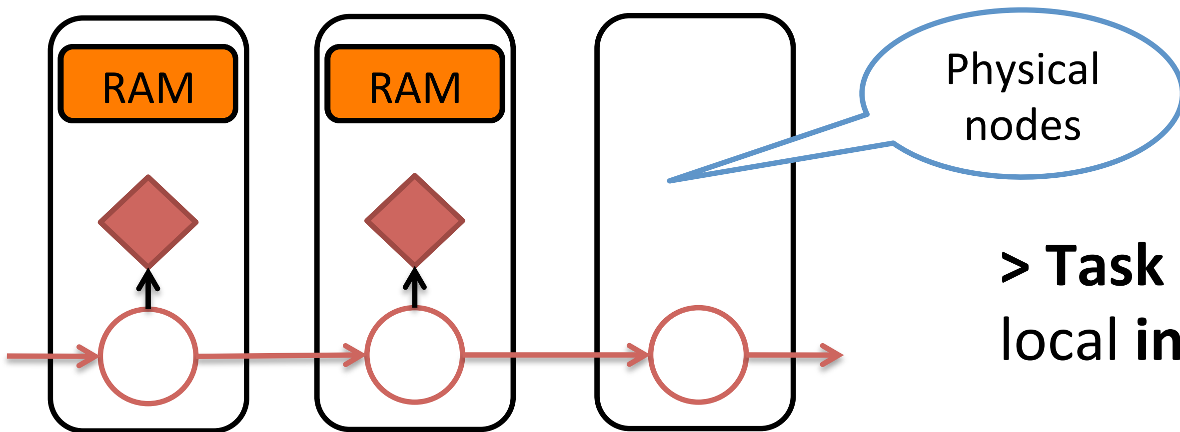
> **@Collection** annotation indicates **merge logic**

Outline



- SDG: Stateful Dataflow Graphs
- Handling distributed state in SDGs
- Translating Java programs to SDGs
- **Checkpoint-Based fault tolerance for SDGs**
- Experimental evaluation

Challenges of Making SDGs Fault Tolerant



Physical deployment of SDG

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

> Node failures may lead to **state loss**

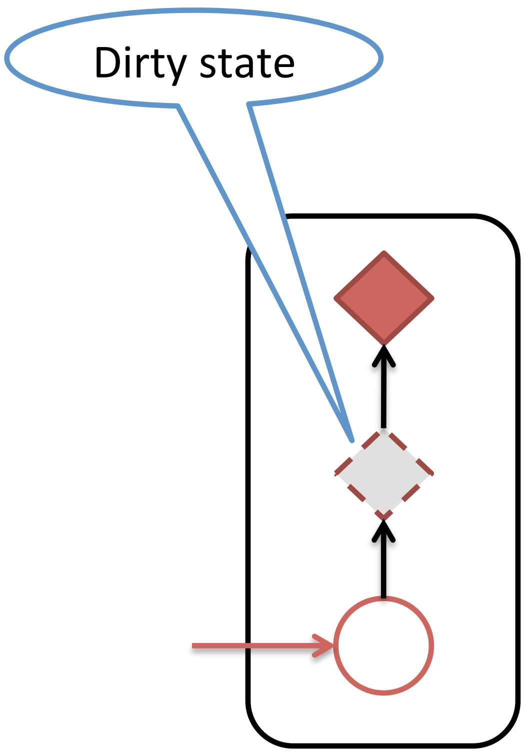
Checkpointing State

- 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

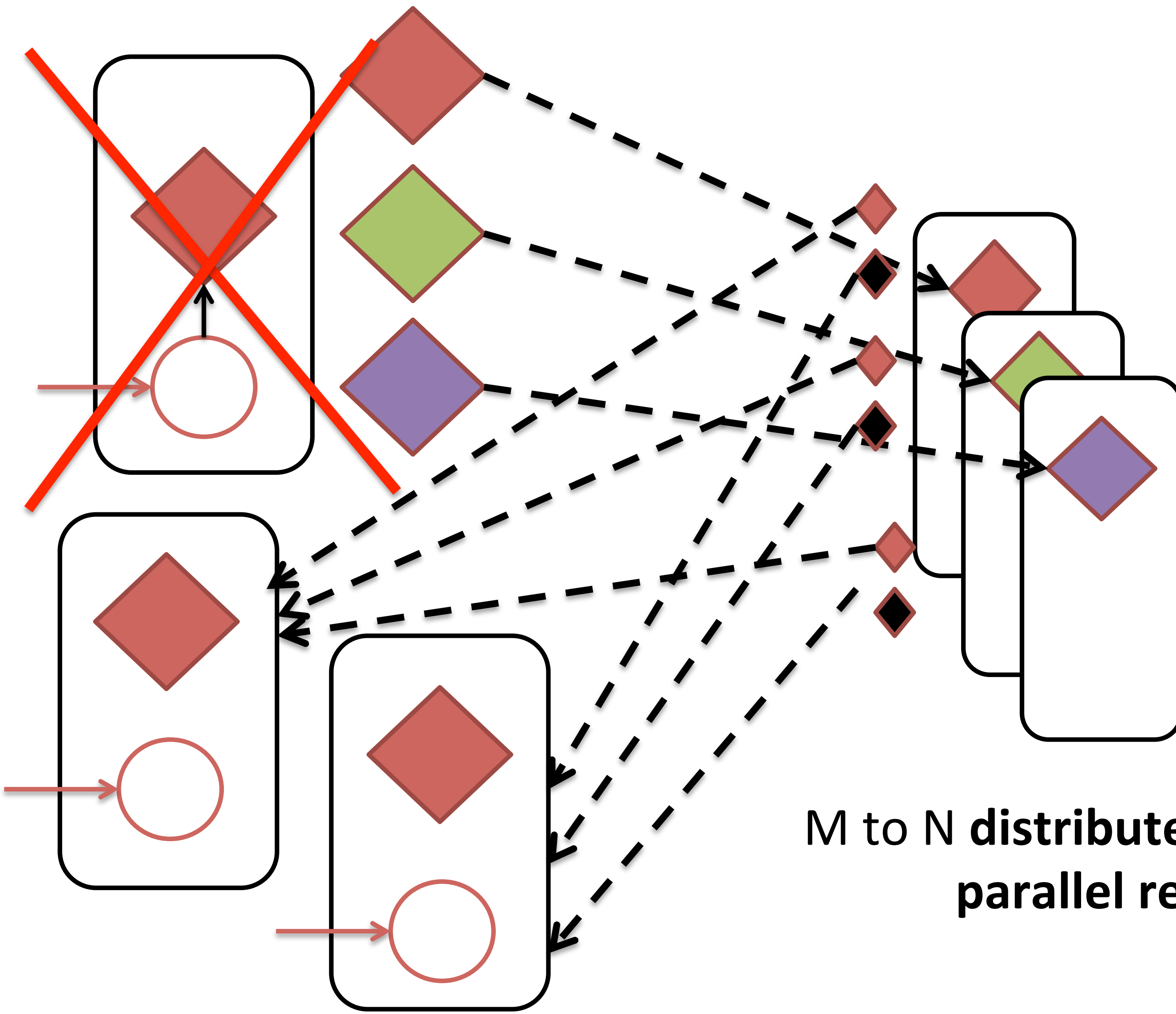
Checkpoint Mechanism for Fault Tolerance



Asynchronous, lock-free checkpointing

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

Distributed M to N Checkpoint Backup



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

Evaluation of SDG Performance

How does mutable state impact performance?

How efficient are translated SDGs?

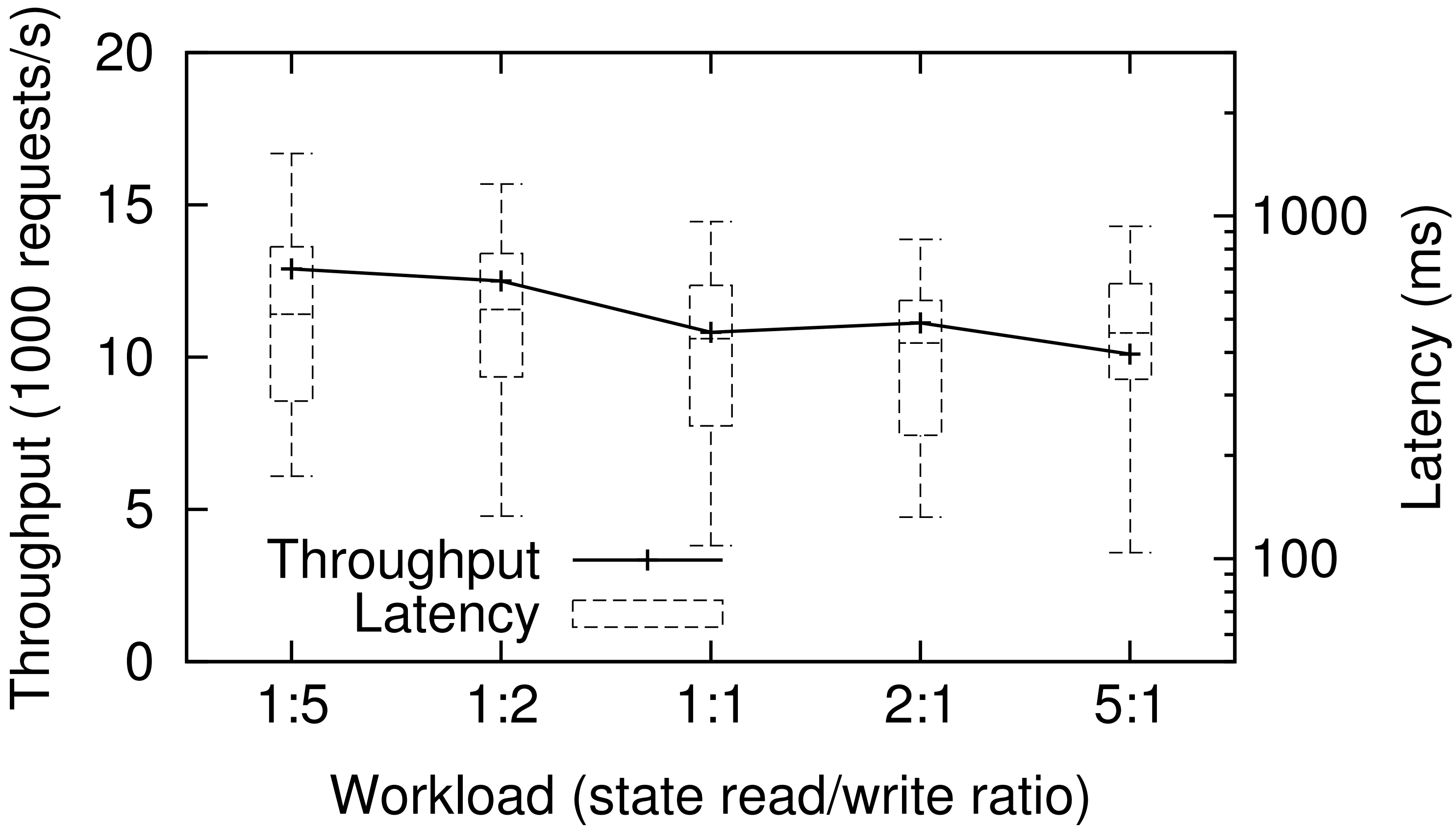
What is the throughput/latency trade-off?

Experimental set-up:

- Amazon EC2 (c1 and m1 xlarge instances)
- Private cluster (4-core 3.4 GHz Intel Xeon servers with 8 GB RAM)
- Sun Java 7, Ubuntu 12.04, Linux kernel 3.10

Processing with Large Mutable State

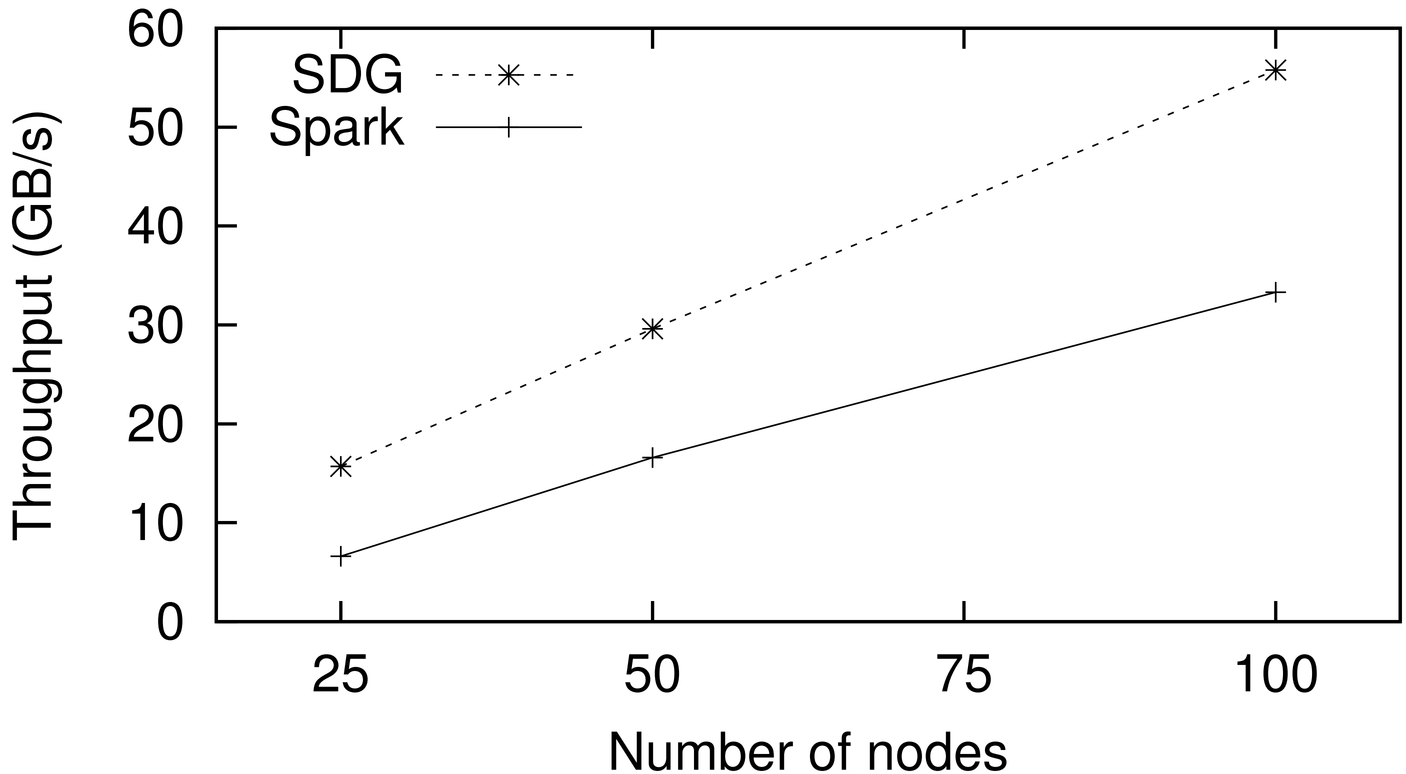
> addRating and getRec functions from recommender algorithm, while changing read/write ratio



Combines batch and online processing to serve fresh results over **large mutable state**

Efficiency of Translated SDG

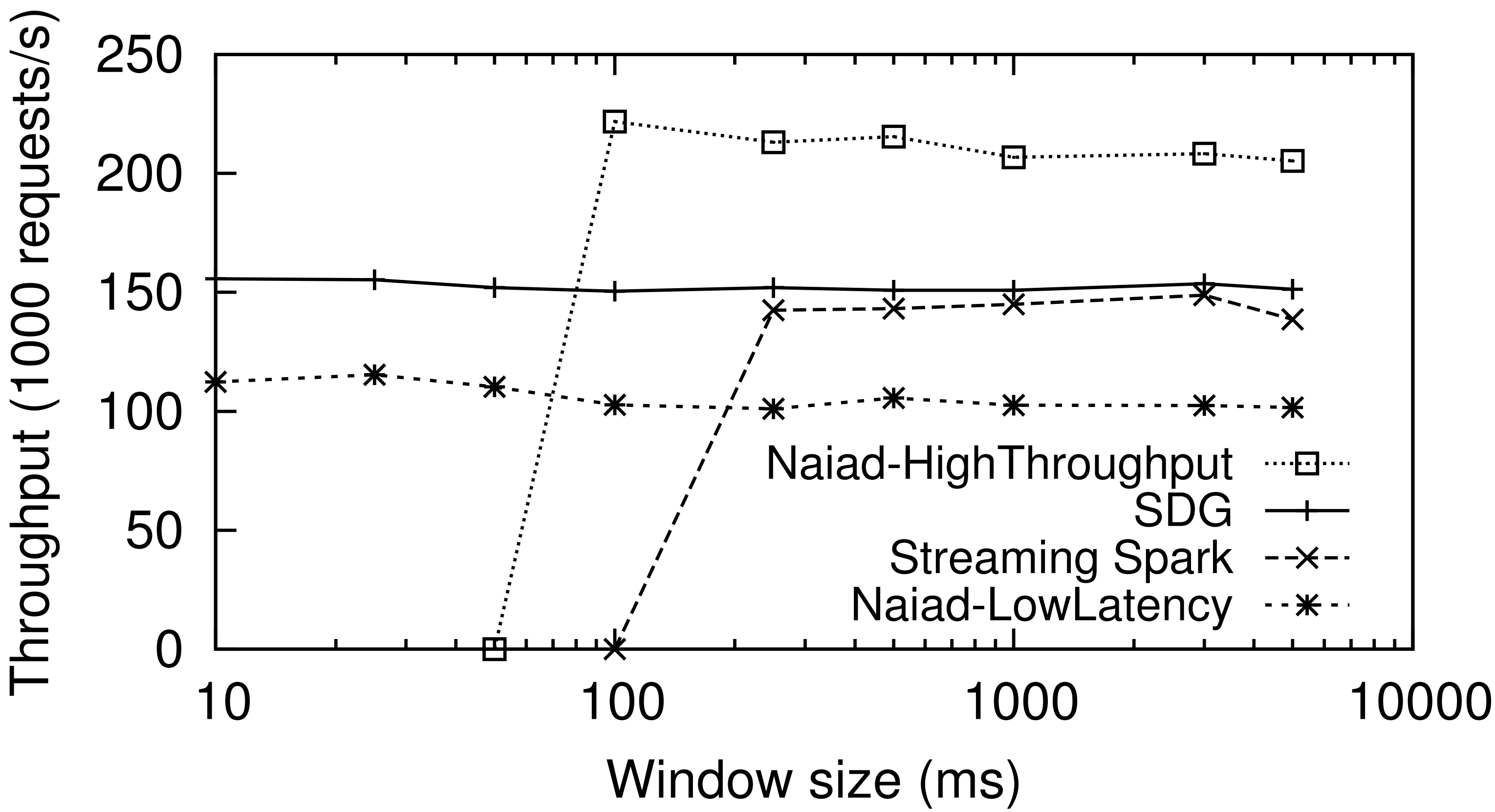
> Batch-oriented, iterative logistic regression



Translated SDG achieves performance similar to non-mutable dataflow

Latency/Throughput Tradeoff

> Streaming word count query, reporting counts over windows



SDGs achieve high throughput while mainting low latency

Summary

Running Java programs with the **performance of current distributed dataflow frameworks**

SDG: Stateful Dataflow Graphs

- Abstractions for distributed **mutable state**
- **Annotations** to disambiguate types of distributed state and state access
- Checkpoint-based **fault tolerance** mechanism

https://github.com/lsds/Seep/

Thank you!
Any Questions?

Raul Castro Fernandez
rc3011@doc.ic.ac.uk