Migrating Monolithic Applications to Microservices with CARGO

Vikram Nitin
Columbia University

Shubhi Asthana
IBM Research

Baishakhi Ray
Columbia University

Rahul Krishna
IBM Research

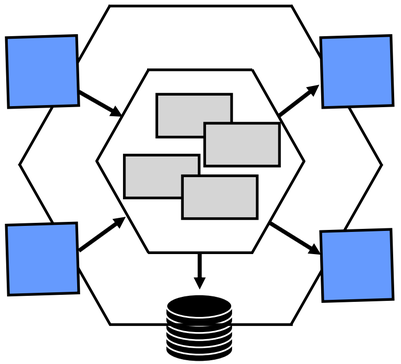
IBM Research

https://arxiv.org/abs/2207.11784

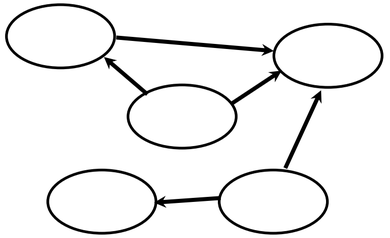
CS@CU

Automated monolith decomposition tools

Monolithic Application

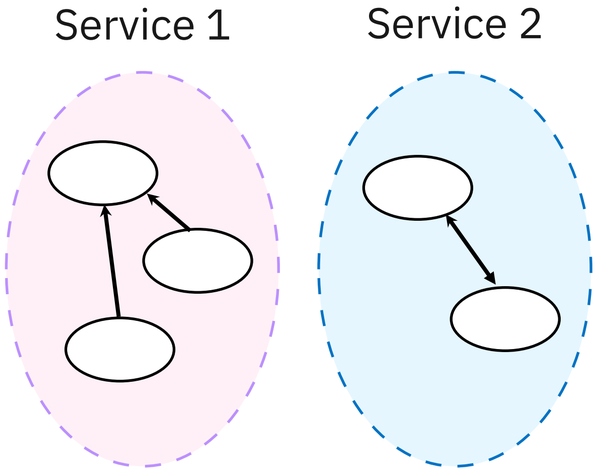


Static and/or Dynamic Analysis



Call / Control-Flow Graph

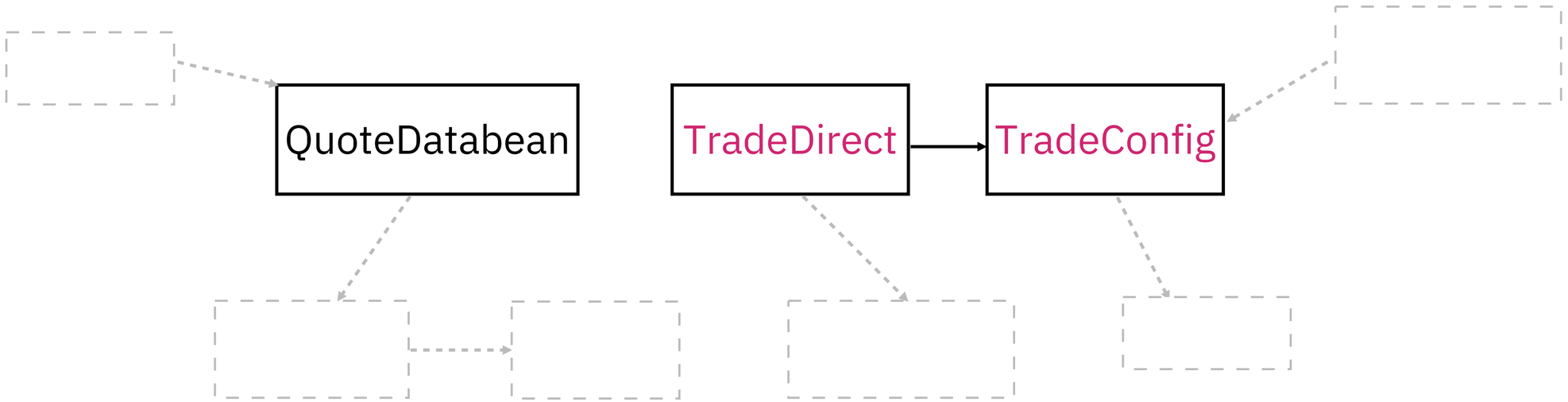
Automated Microservice Partitioning



Microservice Recommendations

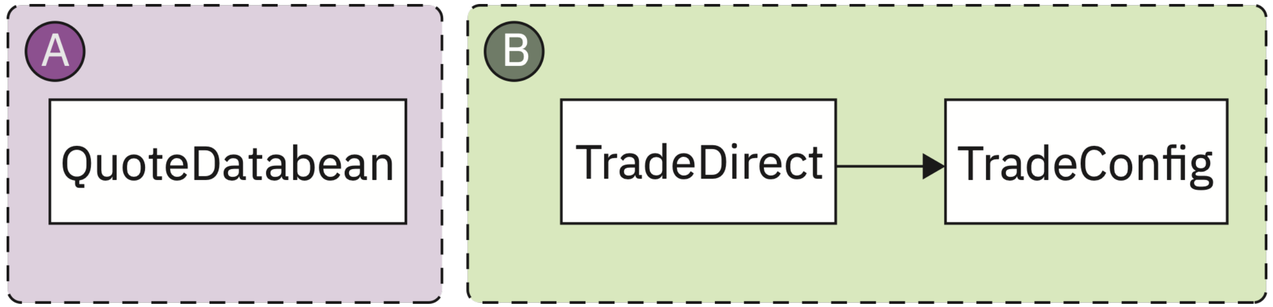
The Problem

A real example from a benchmark application, DAYTRADER. This is a portion of a call-graph, and there is a *call edge* between the classes `TradeDirect` and `TradeConfig`.



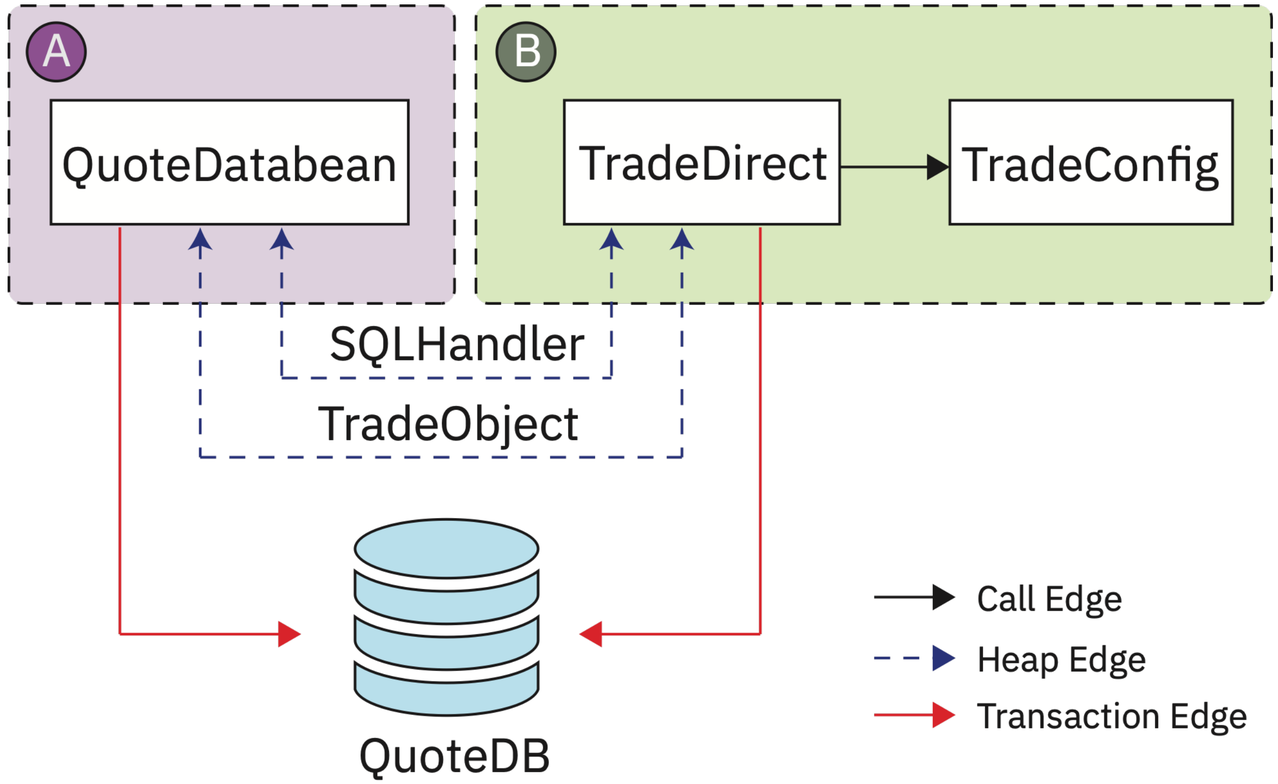
The Problem

In this commonly recommended partitioning, `QuoteDatabean` and `TradeDirect` lie in different partitions.



The Problem

But if we look beyond the call graph, we find this...

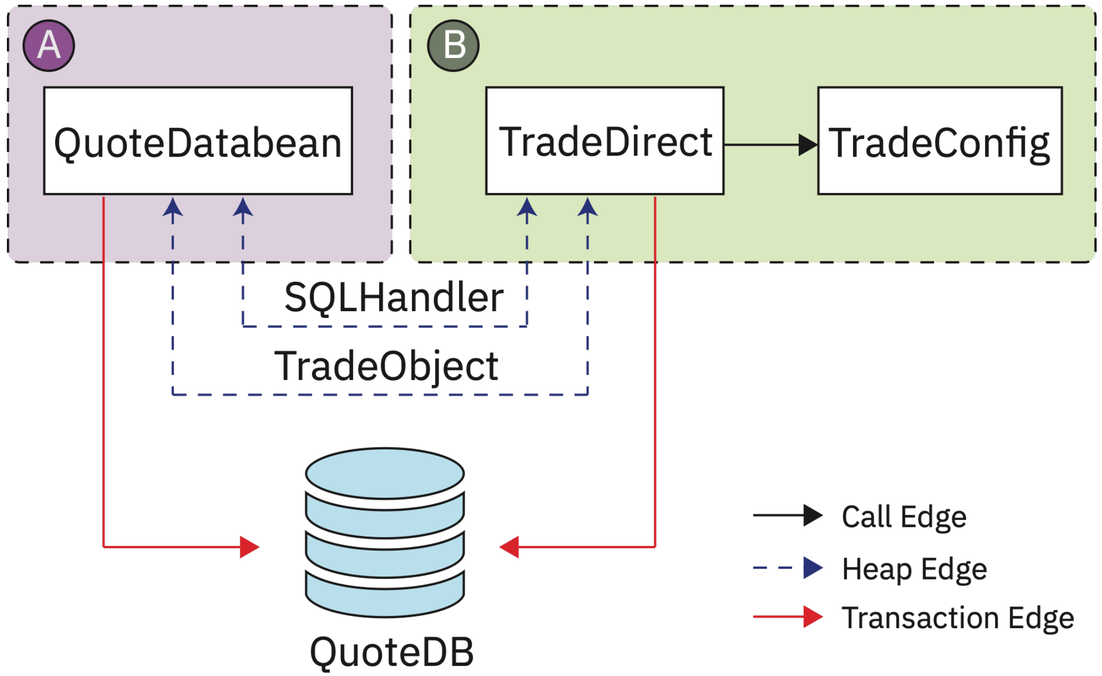


The Problem

Implementing this partitioning scheme can be challenging…

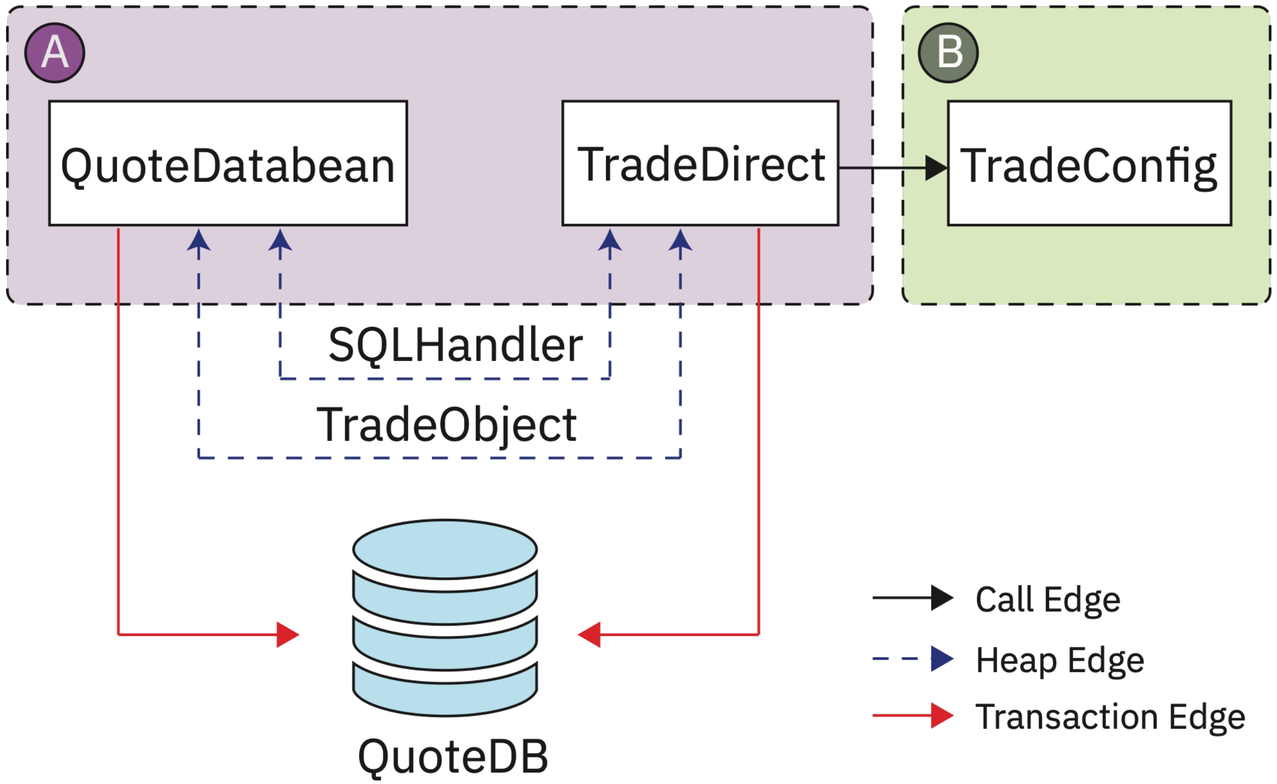
1. **Distributed monolith:** The two classes `QuoteDatabean` and `TradeConfig` are tightly coupled (access shared heap objects)

2. **Distributed transaction:** `QuoteDatabean` and `TradeConfig` write to the same DB.



The Problem

In reality, QuoteDatabean and TradeDirect are tightly coupled!
Our algorithm, CARGO, groups them in the same partition



How do we get better partitions?

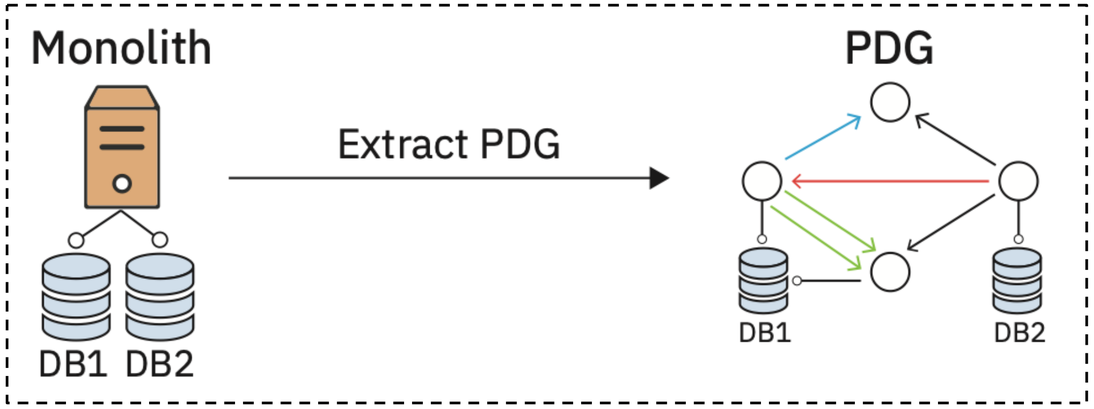
Our algorithm CARGO, uses the following key ideas:

1. **Complete and precise static analysis:** Capture many types of dependencies between classes and build a program dependency graph (PDG)
2. **Explicitly model transactions:** Database transactions are added as edges in the PDG.
3. **Detect communities in the PDG:** We present a novel community detection algorithm to assign partitions to nodes in the PDG.

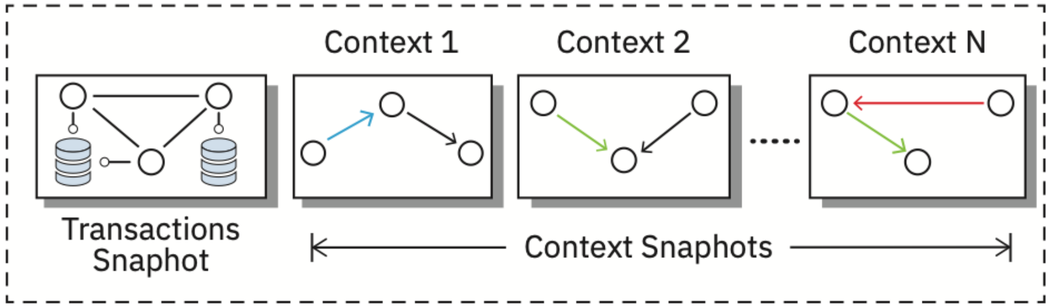
CARGO

CARGO: Overview

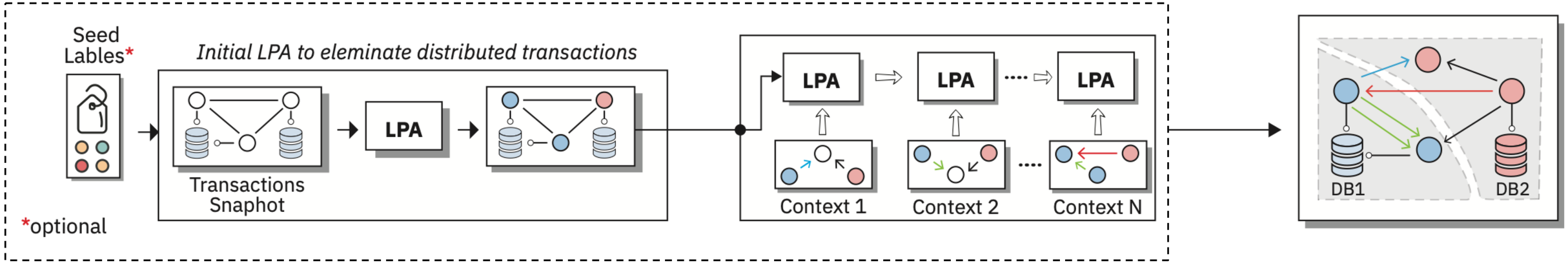
Step I: Build *context-sensitive program dependency graph*



Step II: Extract *context snapshots*



Step III: **Deploy** *Context-sensitive Label Propagation*



Microservice Boundaries

Step I: Context Sensitive PDG

Extract a Context-sensitive Program Dependency Graph

Context-Sensitivity

```
1   void main(Object[] args) {
2     A a1 = new A();
3     Object v1 = a1.foo(new Object());
4
5     A a2 = new A();
6     Object v2 = a2.foo(new Object());
7   }
```

```
1   class A {
2     Object foo(Object v) {
3       B b = new B();
4       return b.bar(v);
5     }
6   }
```

```
1   class B {
2     Object bar(Object v) {
3       . . .
4     }
5   }
6
```

Context-Sensitivity

Context-insensitive Analysis

main()

```
1   void main(Object[] args) {
2     A a1 = new A(); // A/1
3     Object v1 = a1.foo(new Object());
4
5     A a2 = new A(); // A/2
6     Object v2 = a2.foo(new Object());
7   }
```

```
1   class A {
2     Object foo(Object v) {
3       B b = new B();
4       return b.bar(v);
5     }
6   }
```

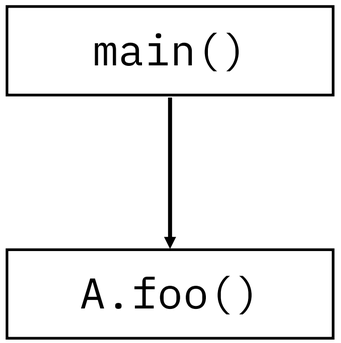
```
1   class B {
2     Object bar(Object v) {
3       . . .
4     }
5   }
6
```

Context-sensitive Analysis

[Φ, Φ] main()

Context-Sensitivity

Context-insensitive Analysis

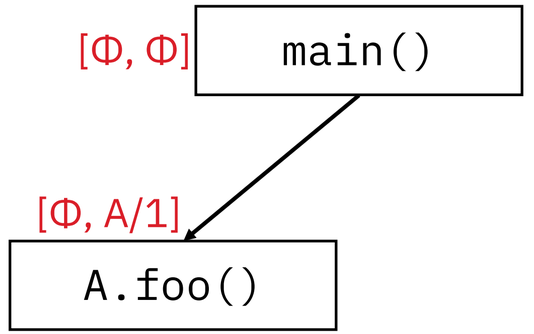



```
1   void main(Object[] args) {
2     A a1 = new A(); // A/1
3     Object v1 = a1.foo(new Object());
4
5     A a2 = new A(); // A/2
6     Object v2 = a2.foo(new Object());
7   }
```

```
1   class A {
2     Object foo(Object v) {
3       B b = new B();
4       return b.bar(v);
5     }
6   }
```

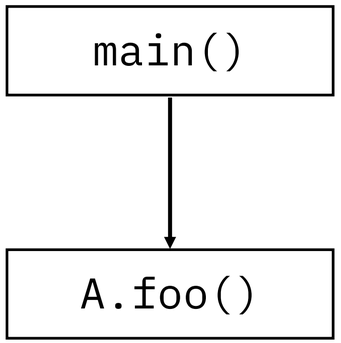
```
1   class B {
2     Object bar(Object v) {
3       . . .
4     }
5   }
6
```

Context-sensitive Analysis



Context-Sensitivity

Context-insensitive Analysis

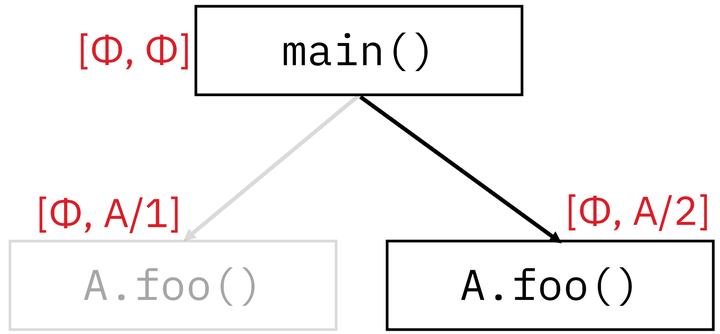



```
1   void main(Object[] args) {
2     A a1 = new A(); // A/1
3     Object v1 = a1.foo(new Object());
4
5     A a2 = new A(); // A/2
6     Object v2 = a2.foo(new Object());
7   }
```

```
1   class A {
2     Object foo(Object v) {
3       B b = new B();
4       return b.bar(v);
5     }
6   }
```

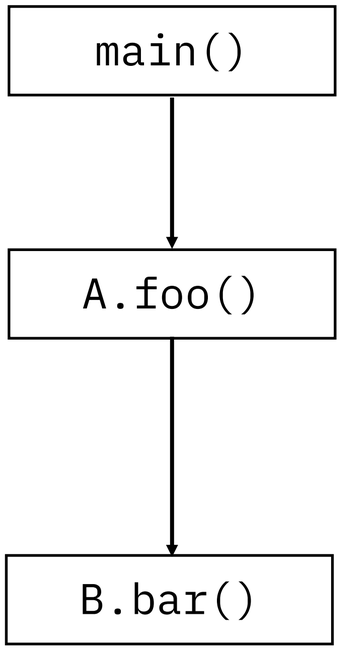
```
1   class B {
2     Object bar(Object v) {
3       . . .
4     }
5   }
6
```

Context-sensitive Analysis



Context-Sensitivity

Context-insensitive Analysis

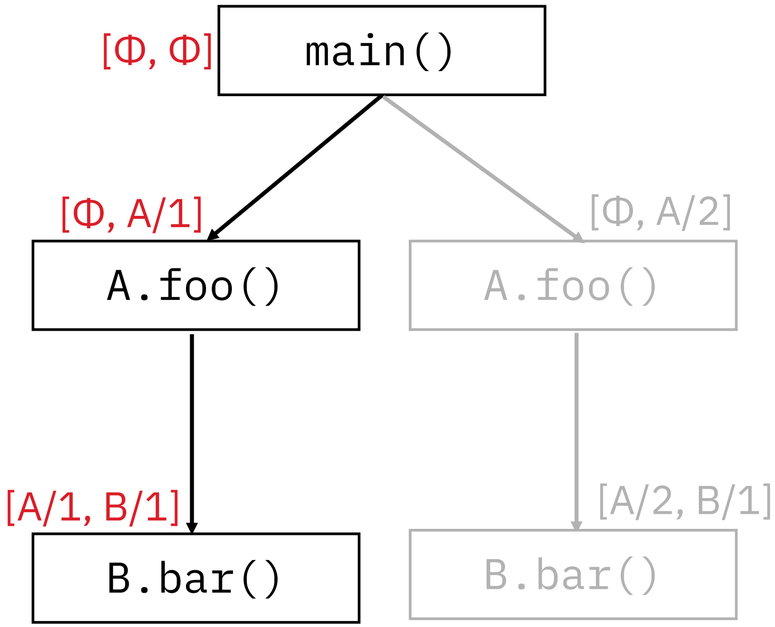



```
1   void main(Object[] args) {
2     A a1 = new A(); // A/1
3     Object v1 = a1.foo(new Object());
4
5     A a2 = new A(); // A/2
6     Object v2 = a2.foo(new Object());
7   }
```

```
1   class A {
2     Object foo(Object v) {
3       B b = new B(); // B/1
4       return b.bar(v);
5     }
6   }
```

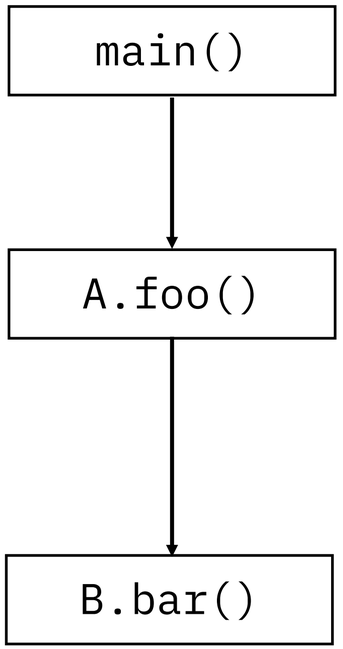
```
1   class B {
2     Object bar(Object v) {
3       . . .
4     }
5   }
6
```

Context-sensitive Analysis



Context-Sensitivity

Context-insensitive Analysis

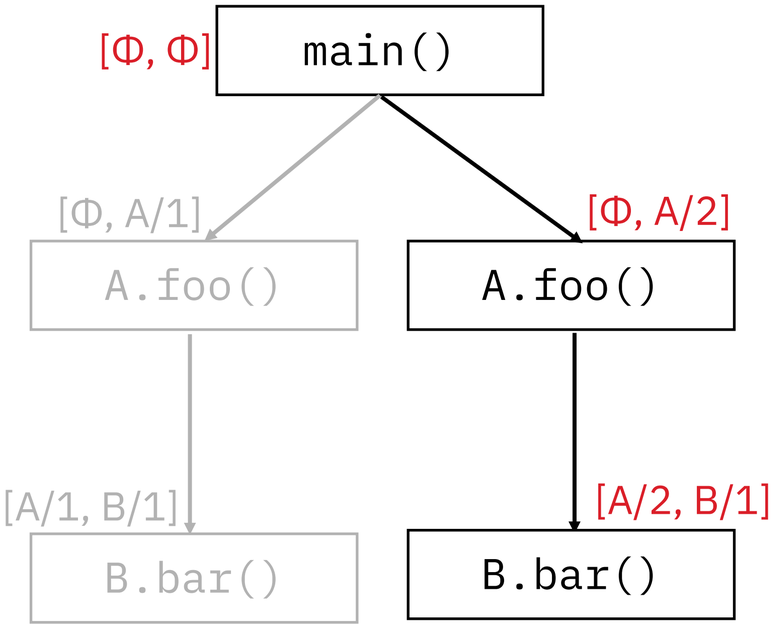



```
1   void main(Object[] args) {
2     A a1 = new A(); // A/1
3     Object v1 = a1.foo(new Object());
4
5     A a2 = new A(); // A/2
6     Object v2 = a2.foo(new Object());
7   }
```

```
1   class A {
2     Object foo(Object v) {
3       B b = new B(); // B/1
4       return b.bar(v);
5     }
6   }
```

```
1   class B {
2     Object bar(Object v) {
3       . . .
4     }
5   }
6
```

Context-sensitive Analysis

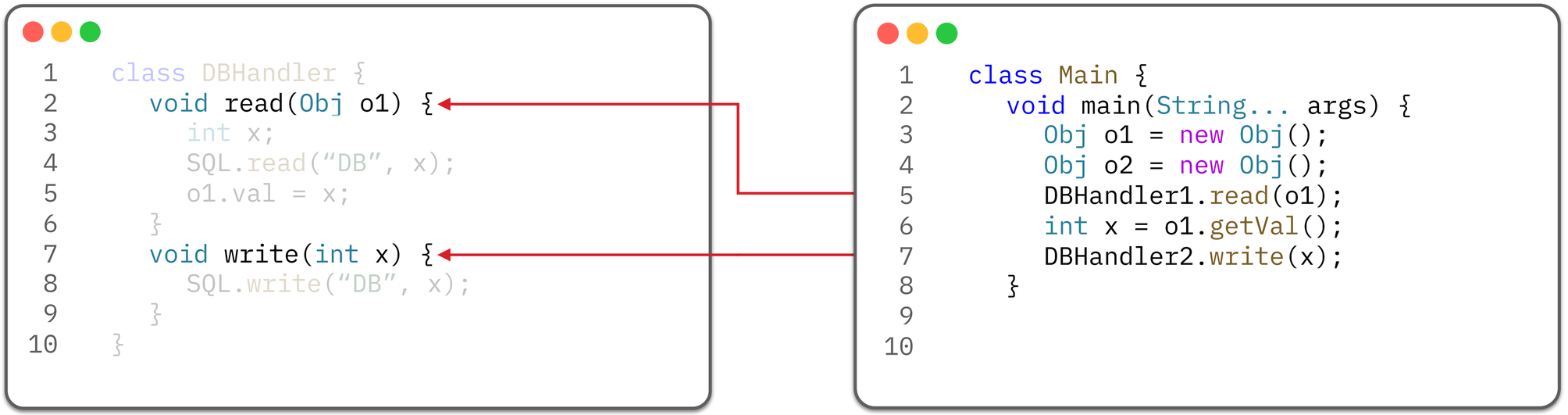


Building a Program Dependency Graph

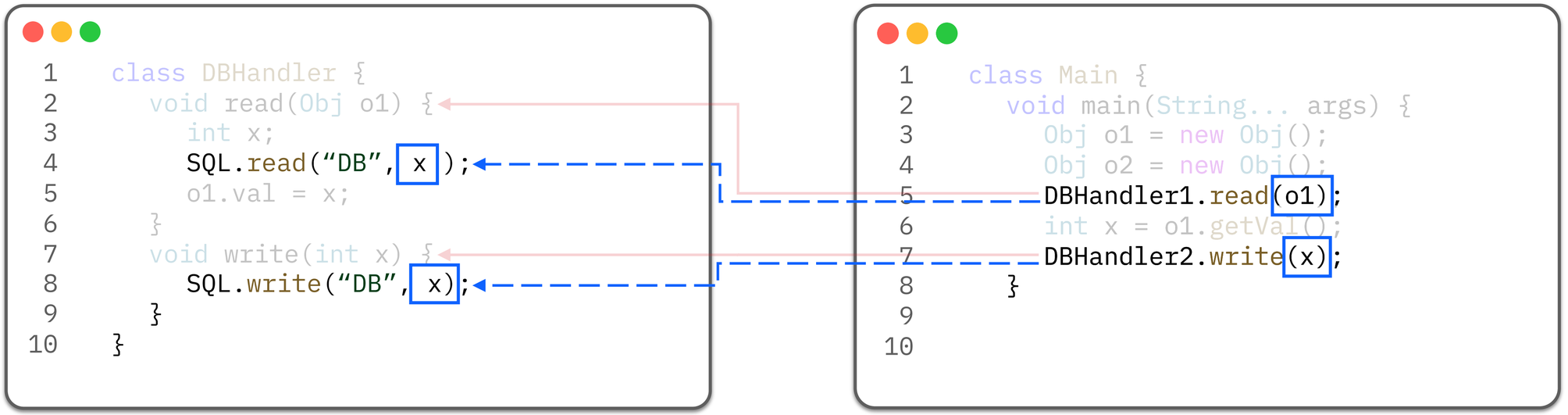
```
1    class DBHandler {
2       void read(Obj o1) {
3          int x;
4          SQL.read(“DB”, x);
5          o1.val = x;
6       }
7       void write(int x) {
8          SQL.write(“DB”, x);
9       }
10   }
```

```
1    class Main {
2       void main(String... args) {
3          Obj o1 = new Obj();
4          Obj o2 = new Obj();
5          DBHandler1.read(o1);
6          int x = o1.getVal();
7          DBHandler2.write(x);
8       }
9
10
```

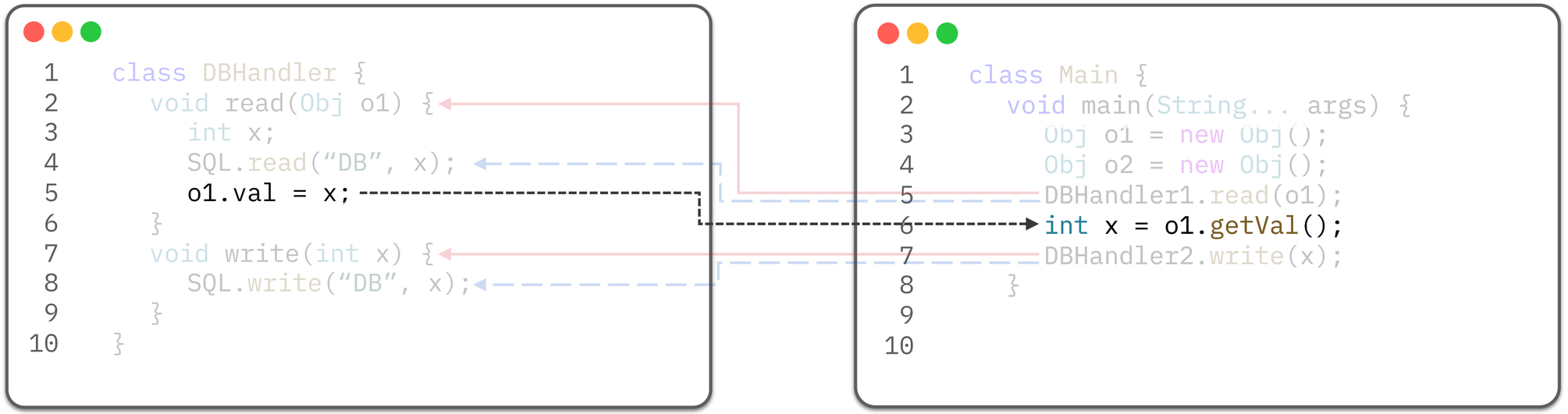
Building a Program Dependency Graph



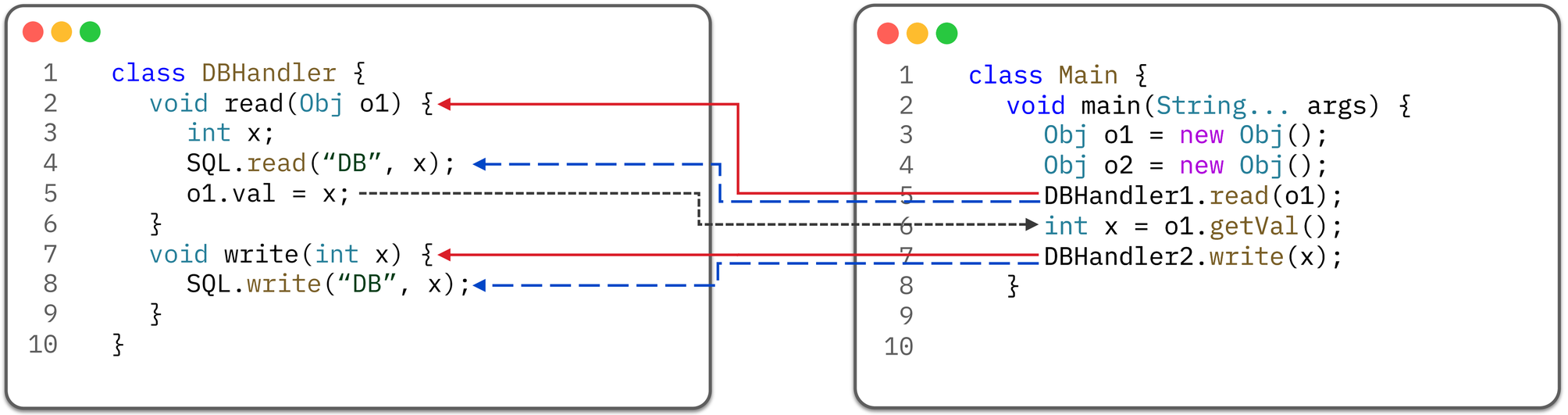
Building a Program Dependency Graph



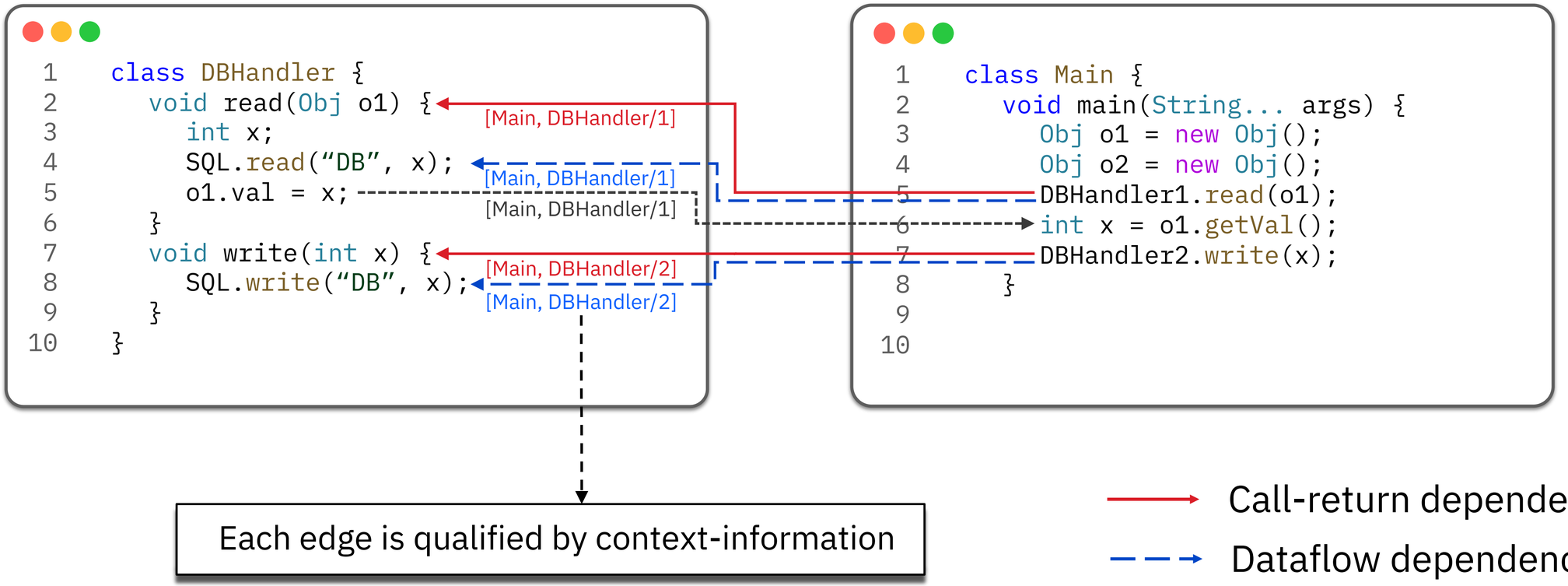
Building a Program Dependency Graph



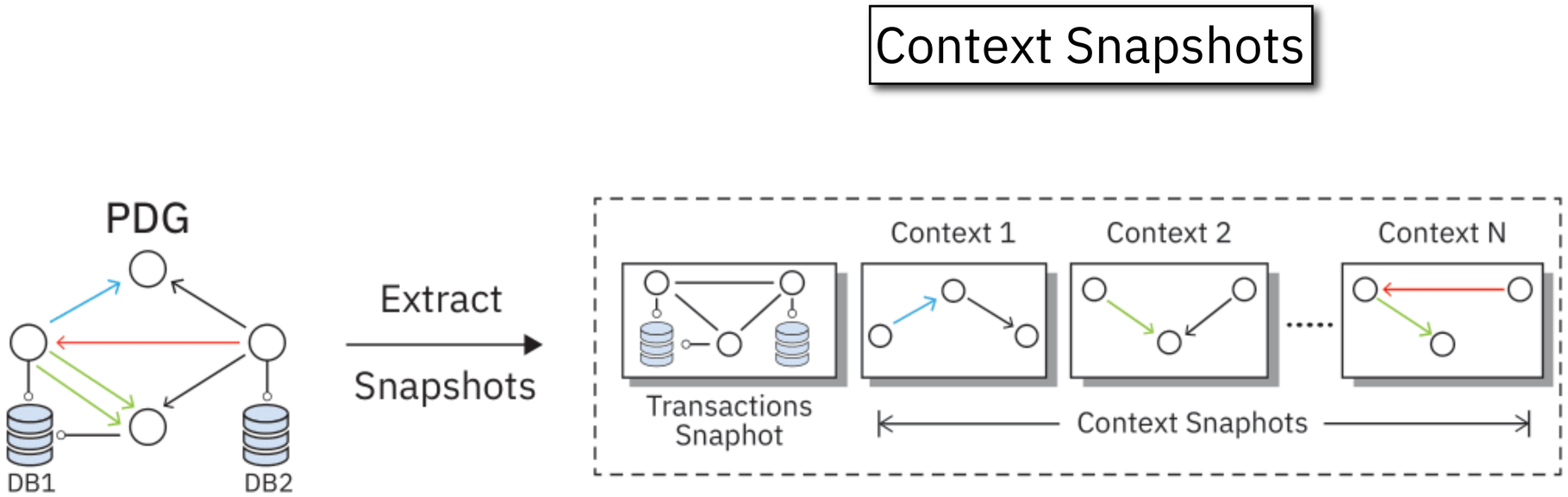
Building a Program Dependency Graph



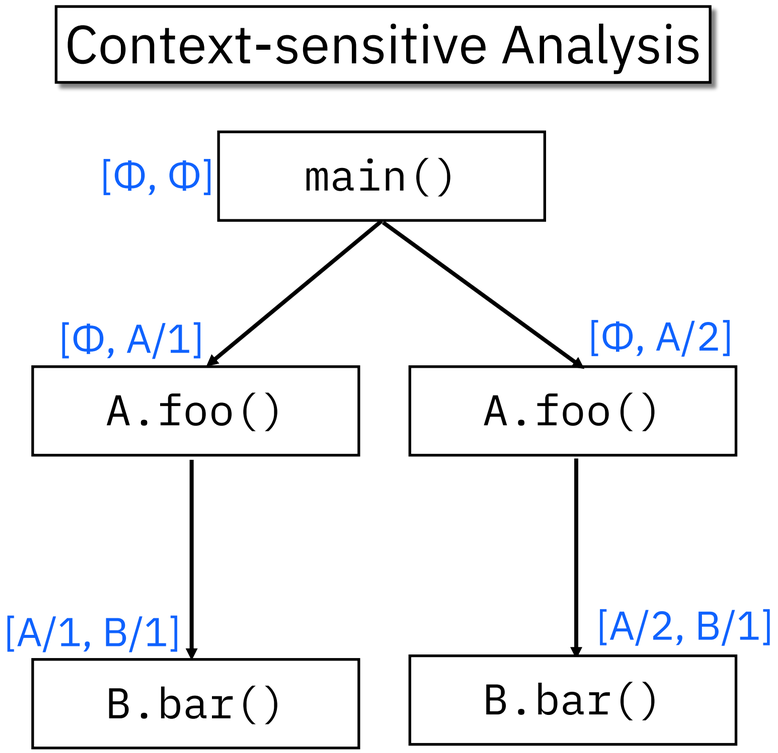
Building a Program Dependency Graph



Step II: Isolating Context Snapshots



Recap: Context-sensitive Analysis

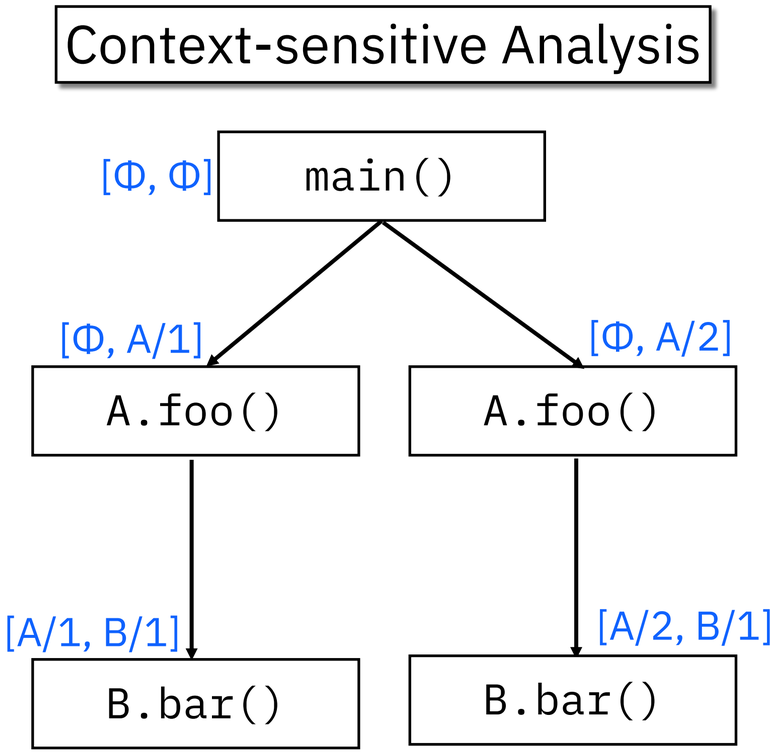



```
1   void main(Object[] args) {
2     A a1 = new A();
3     Object v1 = a1.foo(new Object());
4
5     A a2 = new A();
6     Object v2 = a2.foo(new Object());
7   }
```

```
1   class A {
2     Object foo(Object v) {
3       B b = new B();
4       return b.bar(v);
5     }
6   }
```

```
1   class B {
2     Object bar(Object v) {
3       . . .
4     }
5   }
6
```

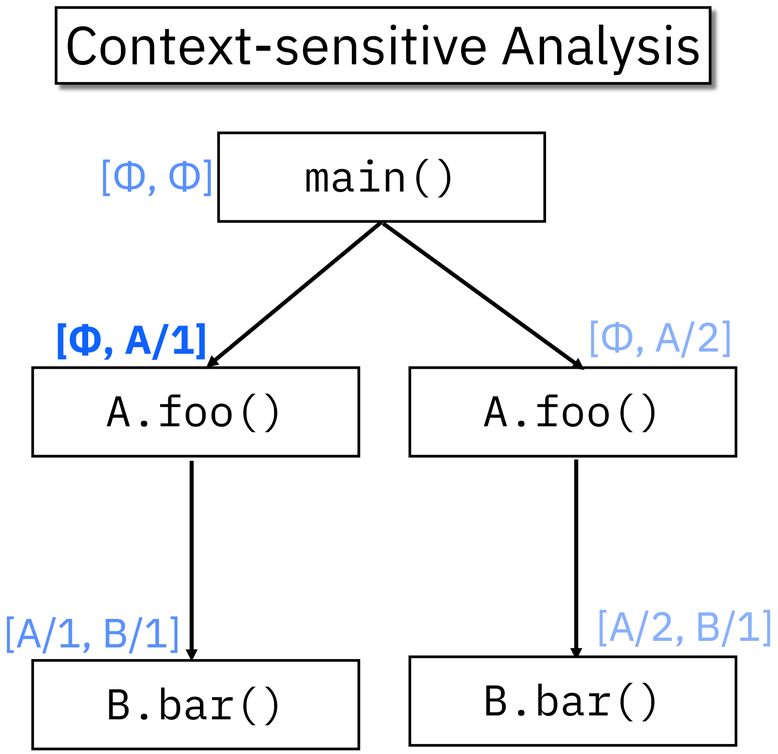
Context Snapshots



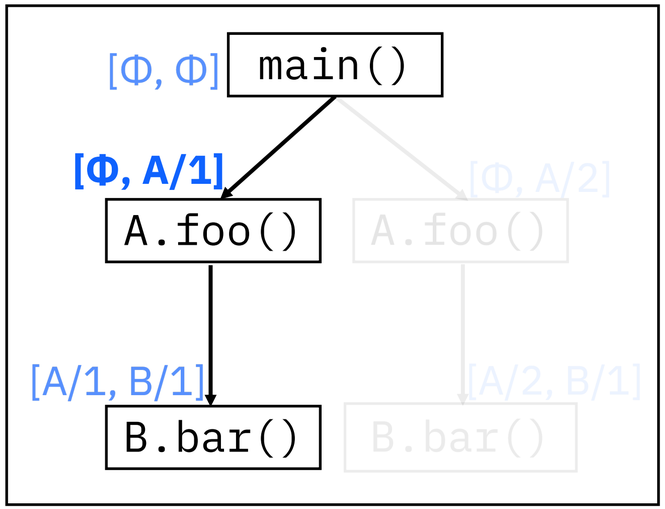
The context-sensitive PDG is a superposition of all possible contexts.

At any time, `A.foo()` and `B.bar()` can exist in any one context *but not* both simultaneously.

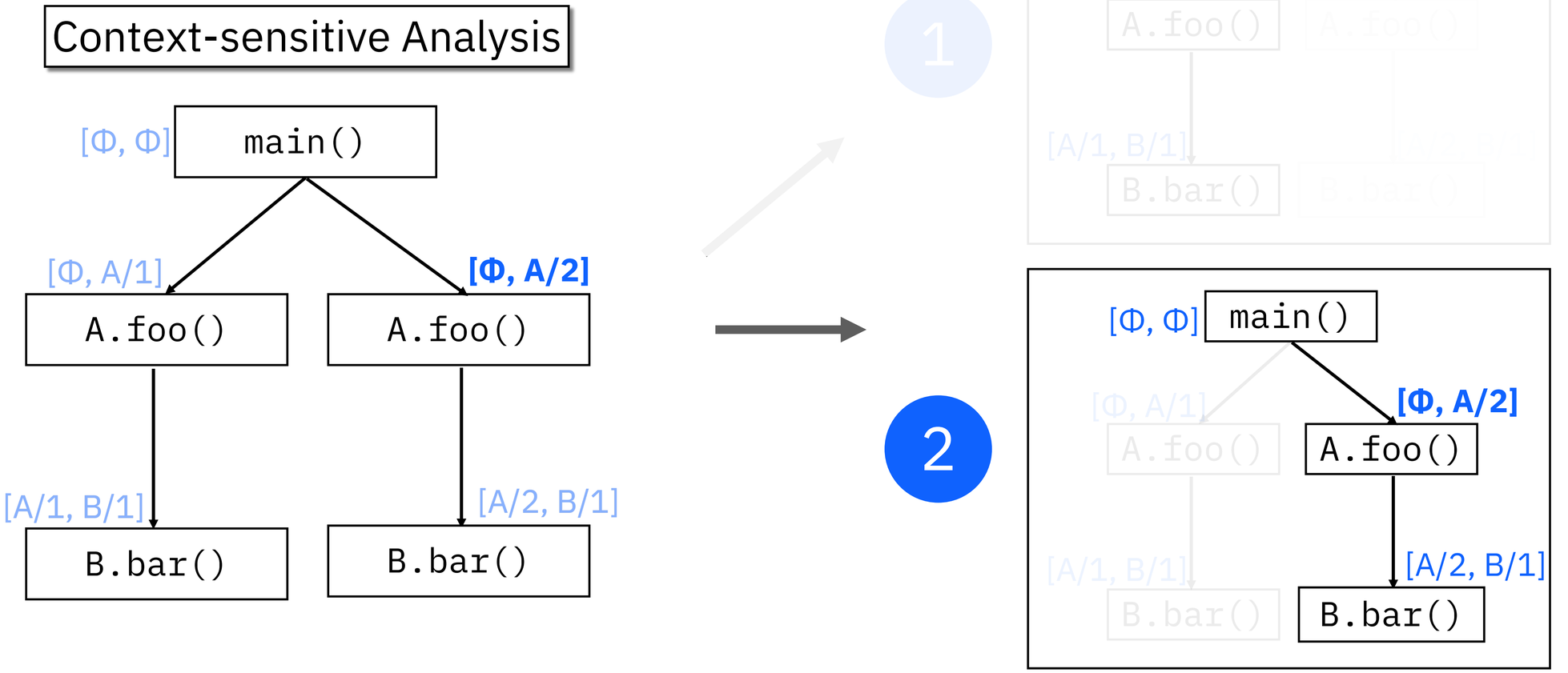
Context Snapshots



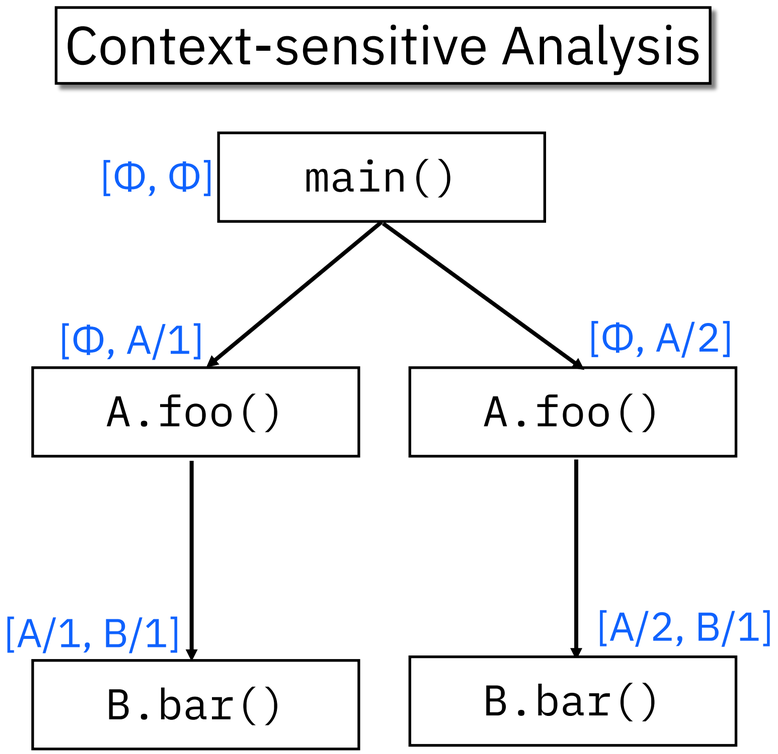
1

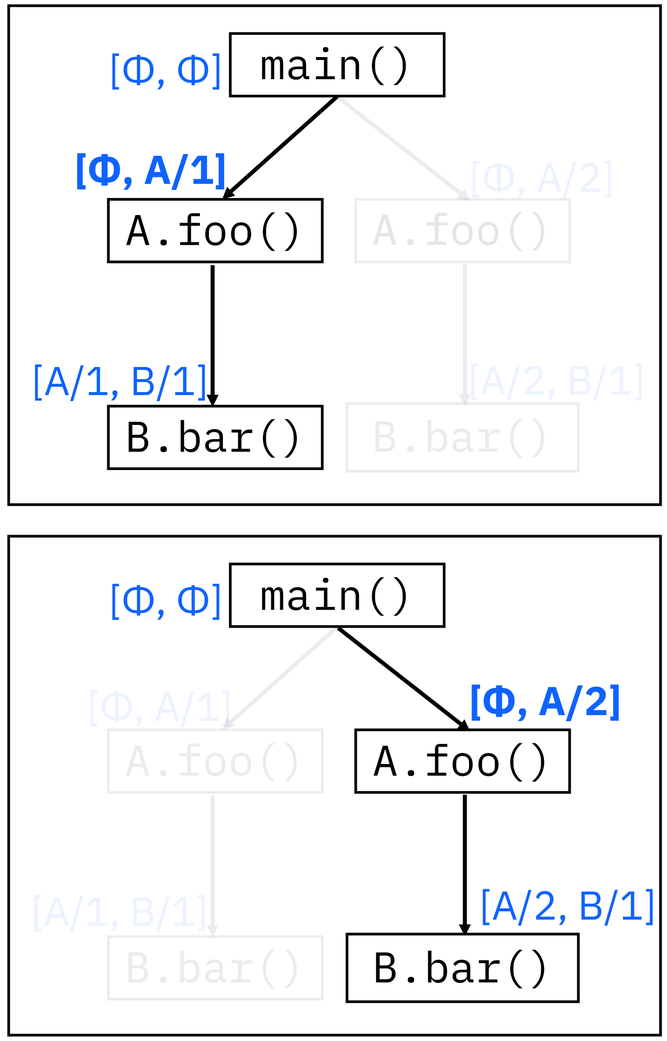


Context Snapshots

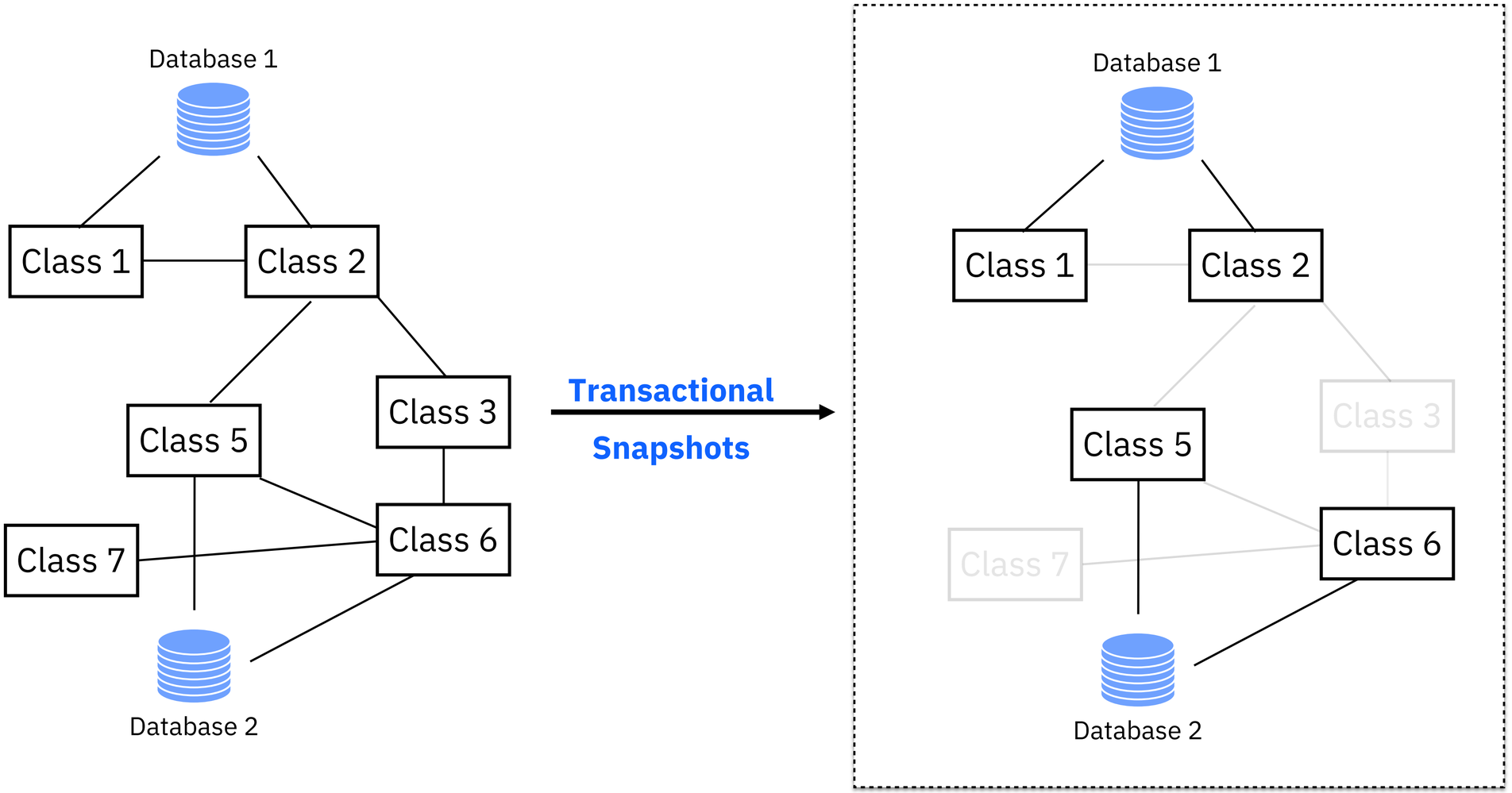


Context Snapshots





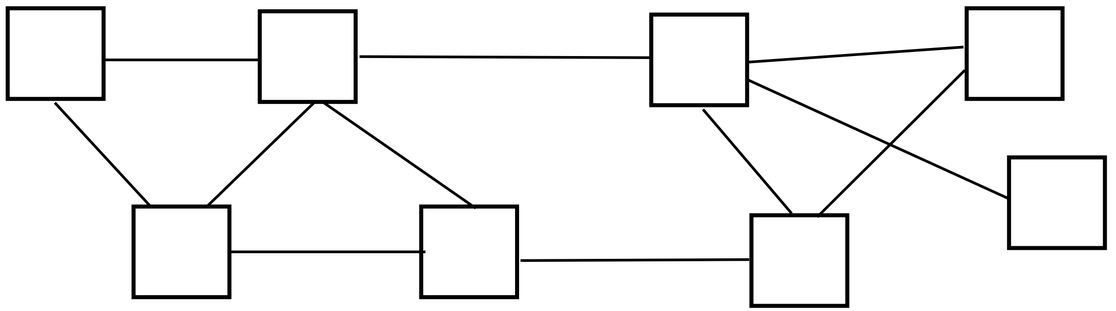
Transactional Snapshot



Step III: Context-Sensitive Label Propagation

Context-sensitive Label Propagation

Label propagation is an algorithm to detect communities of nodes in a graph

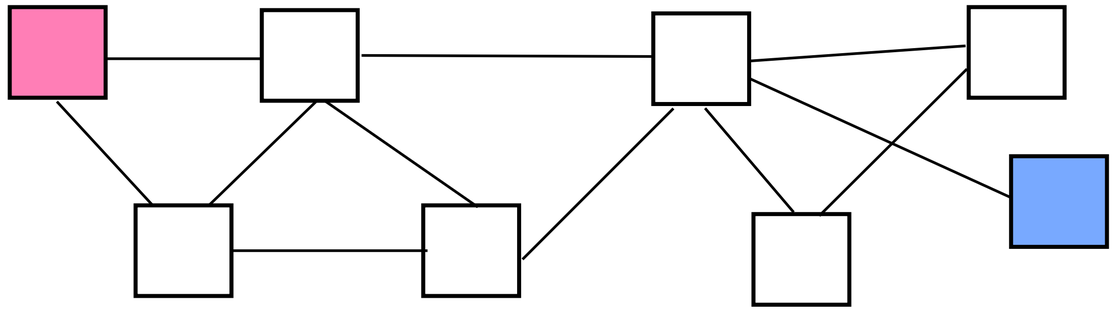


Label Propagation Algorithm

Initial State

We start with some initial assignment of labels to nodes

Magenta and Blue are the two categories of label



Label Propagation Algorithm

Initial State

We start with some initial assignment of labels to nodes

Magenta and Blue are the two categories of label

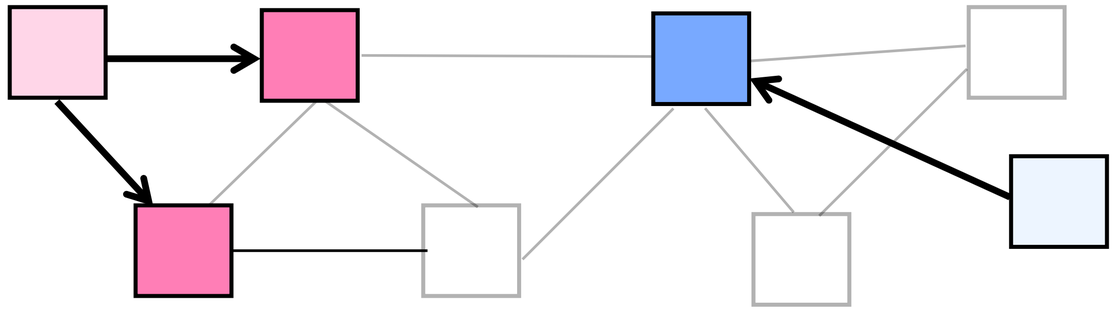
a) Random (unsupervised)

b) From the output of another algorithm (semi-supervised)

Label Propagation Algorithm

Pass 1

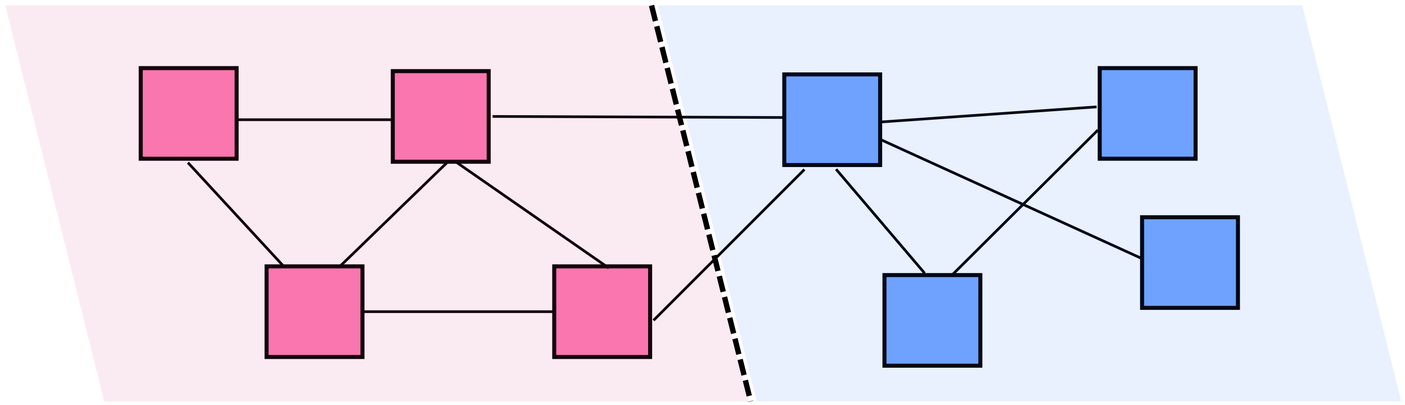
Each node is assigned the majority label of its neighbors



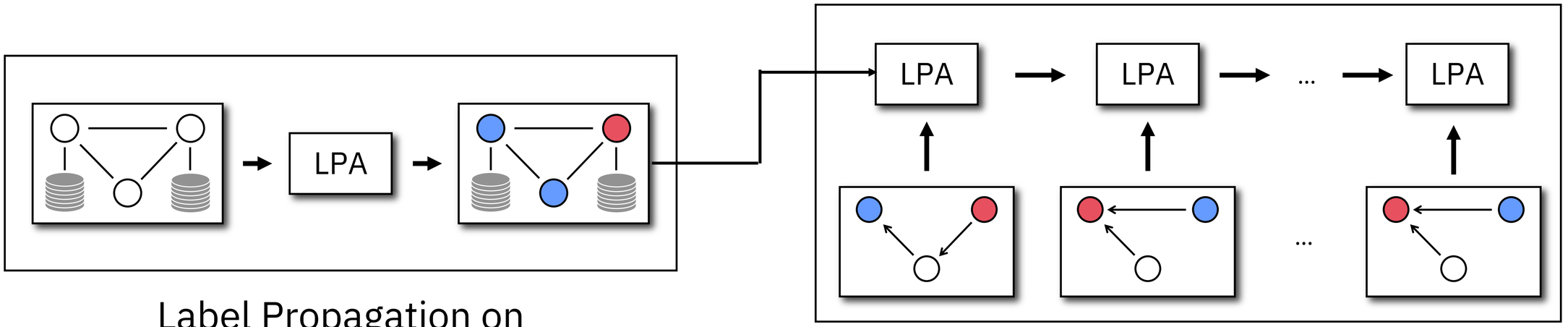
Label Propagation Algorithm

Final

Repeat until convergence (no more node updates)



Context-sensitive Label Propagation

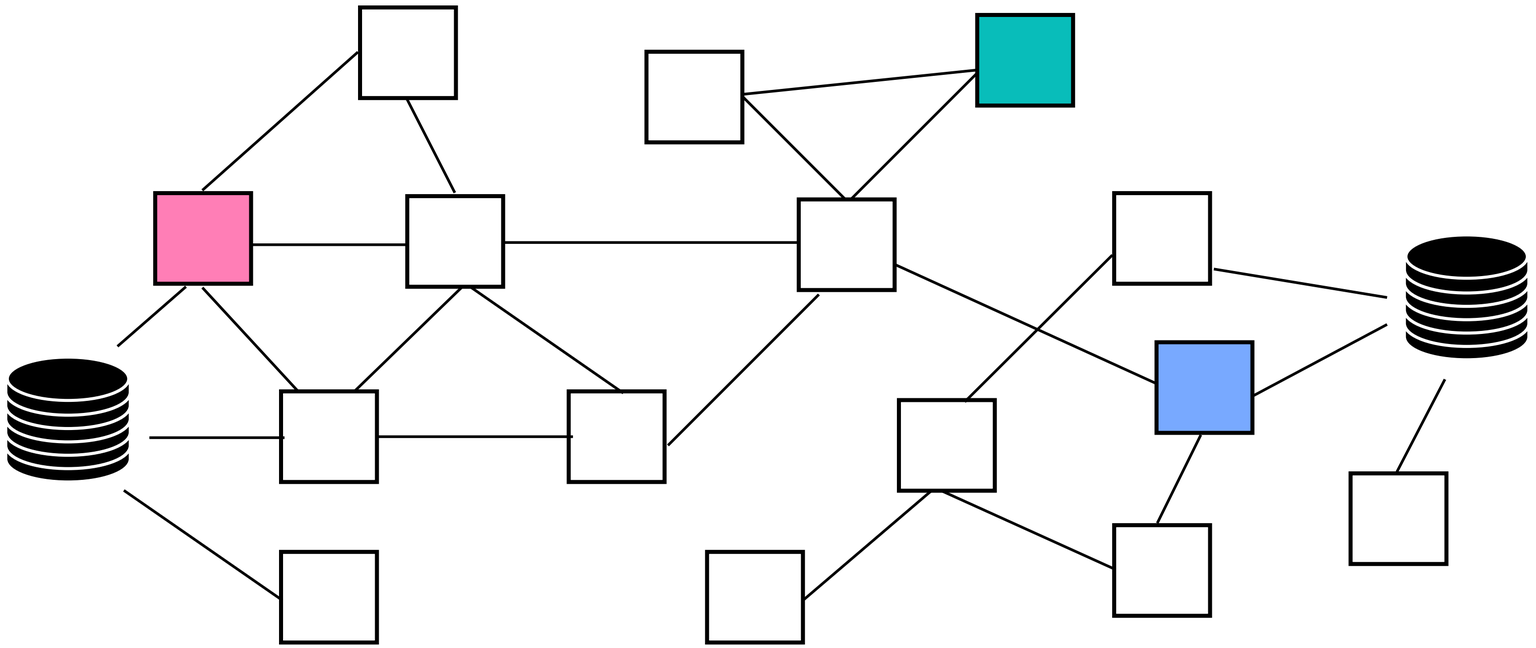


Label Propagation on
Transactional Snapshot

Label Propagation on
Context Snapshots

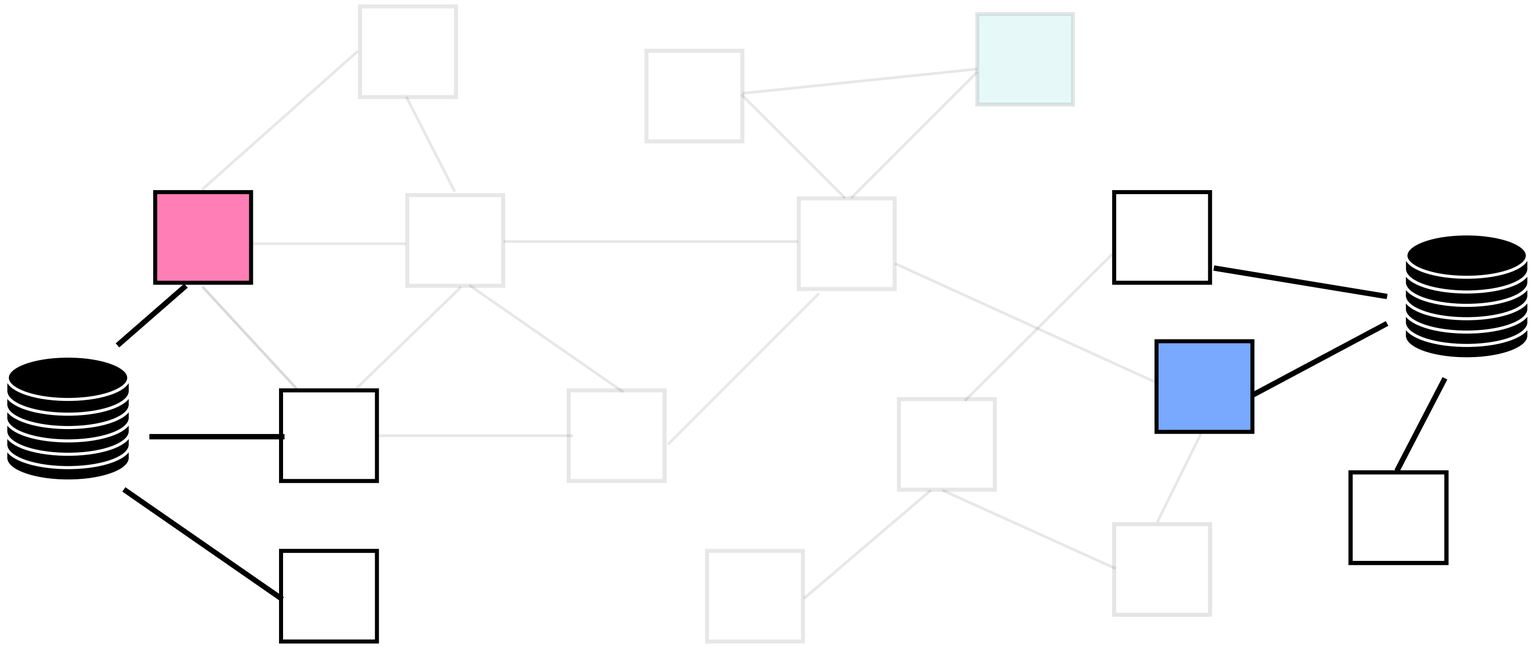
Context-sensitive Label Propagation

Initialize labels



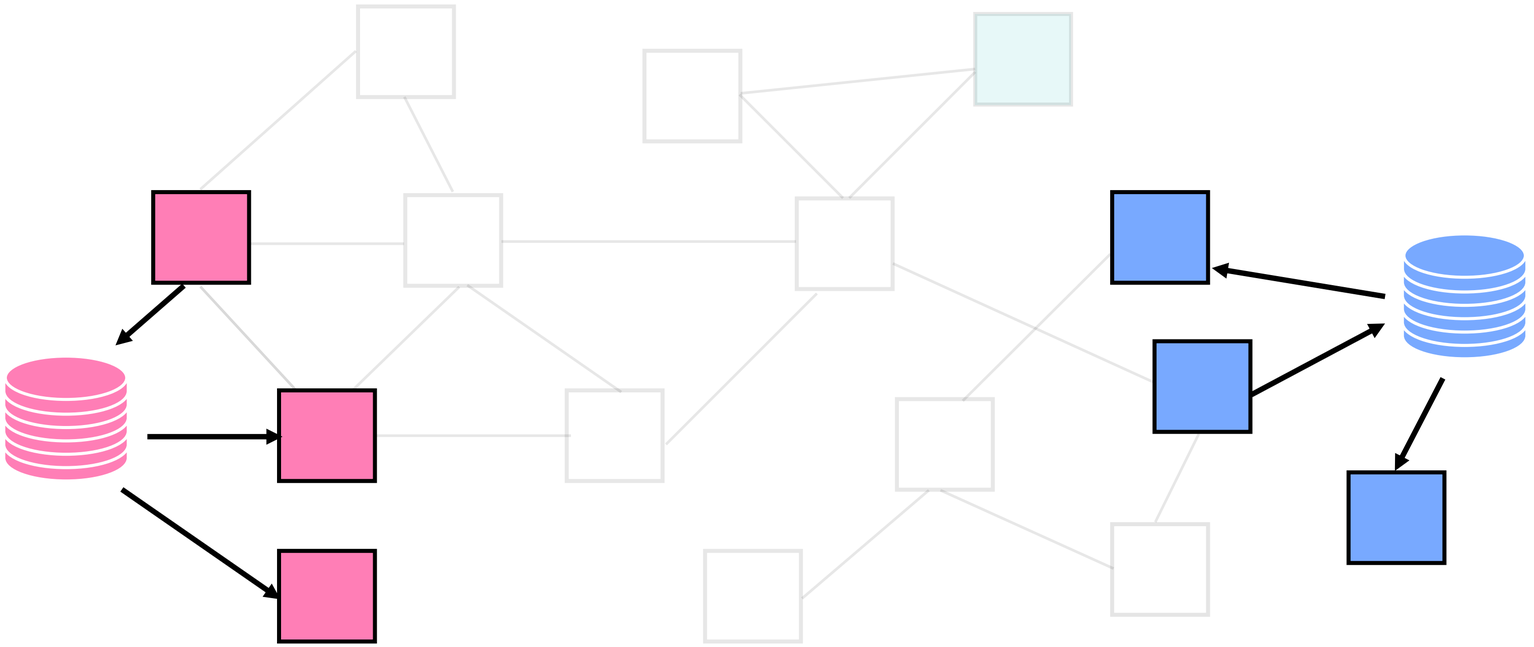
Context-sensitive Label Propagation

Transactional Snapshot



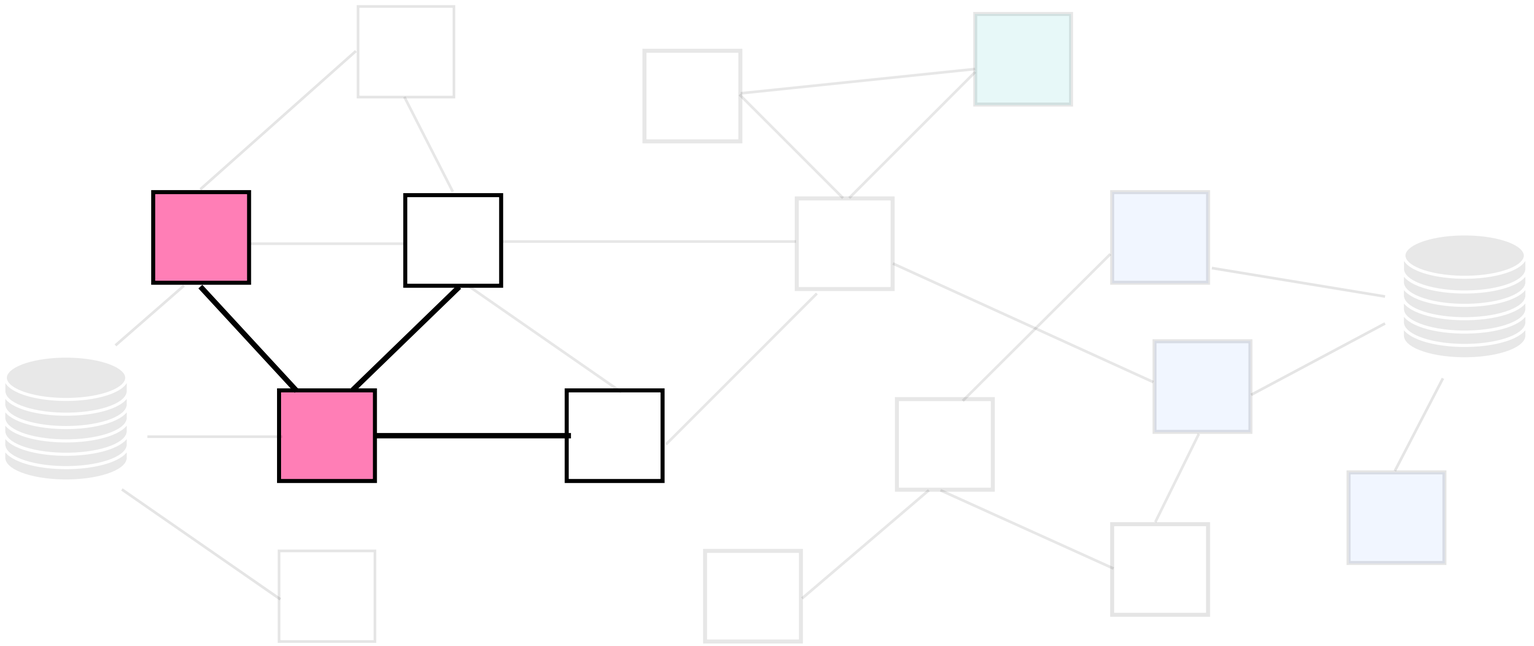
Context-sensitive Label Propagation

Propagate Labels



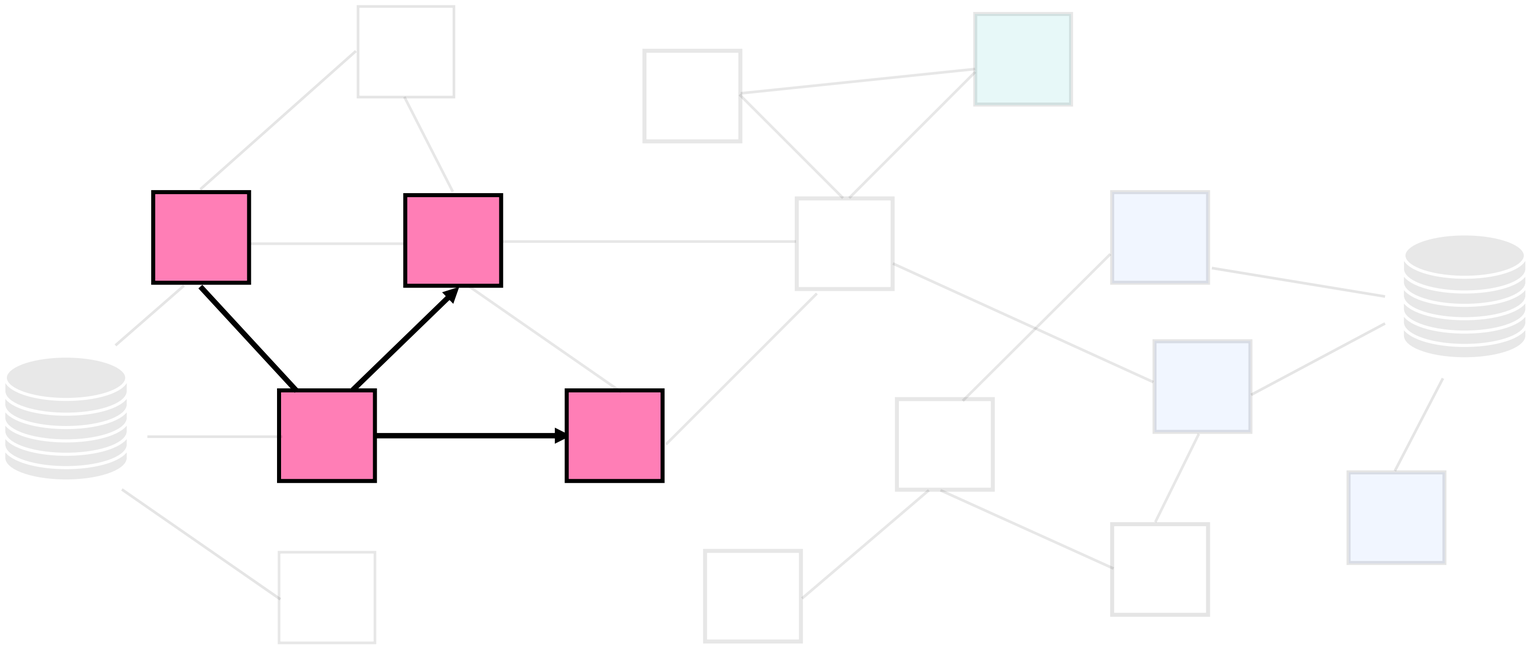
Context-sensitive Label Propagation

Context Snapshot 1



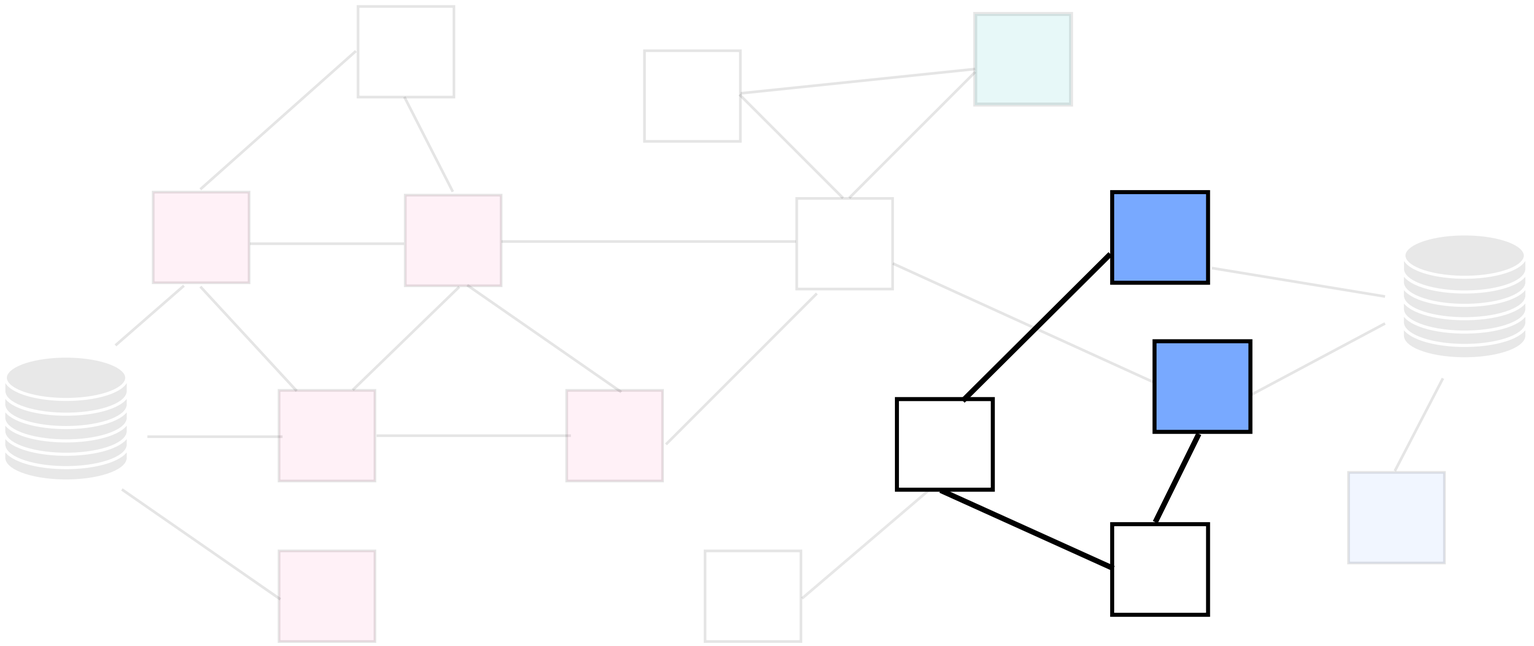
Context-sensitive Label Propagation

Propagate Labels



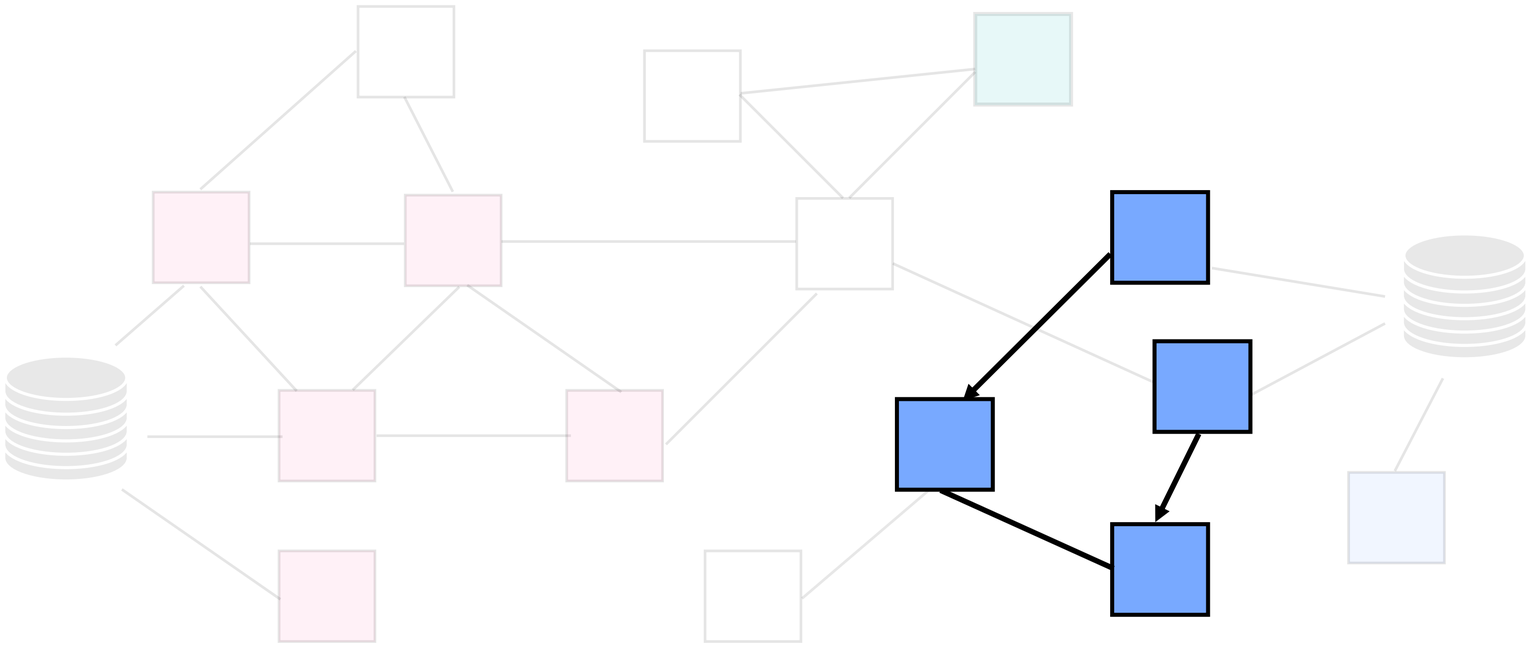
Context-sensitive Label Propagation

Context Snapshot 2



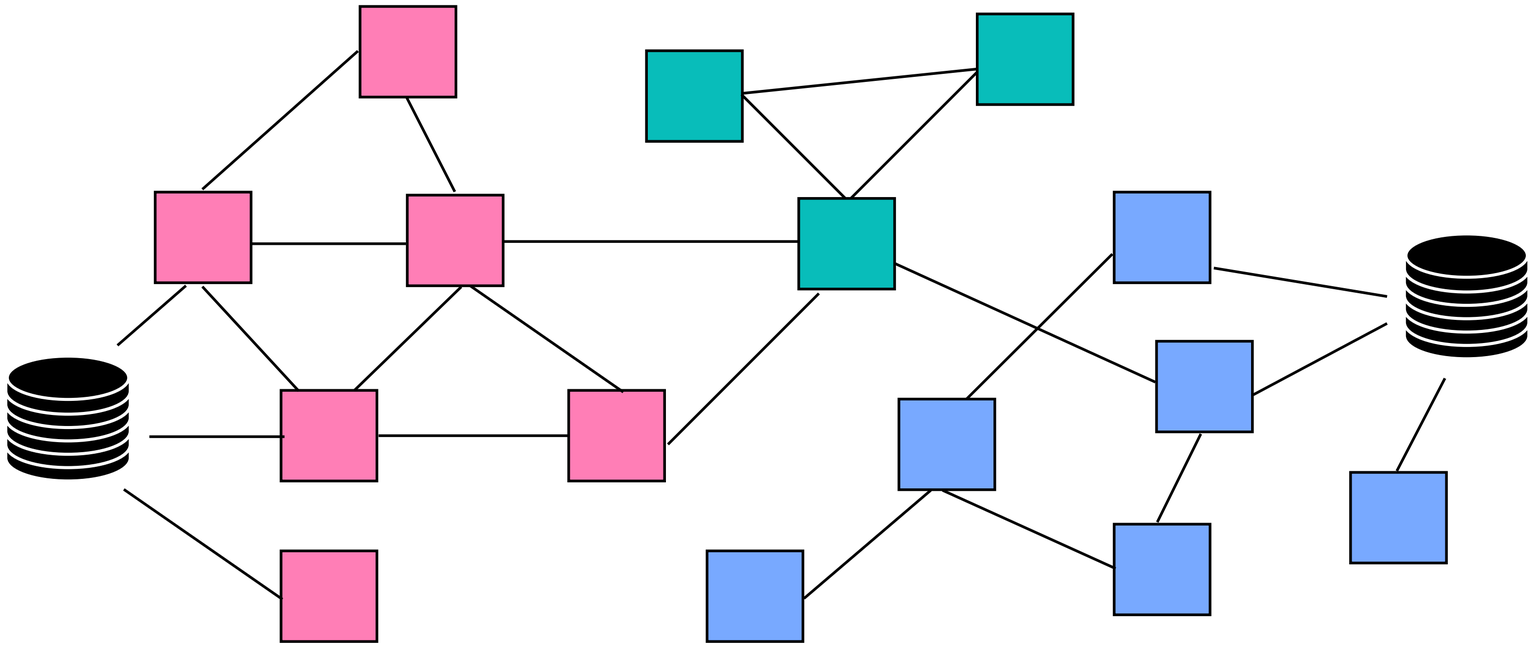
Context-sensitive Label Propagation

Propagate Labels

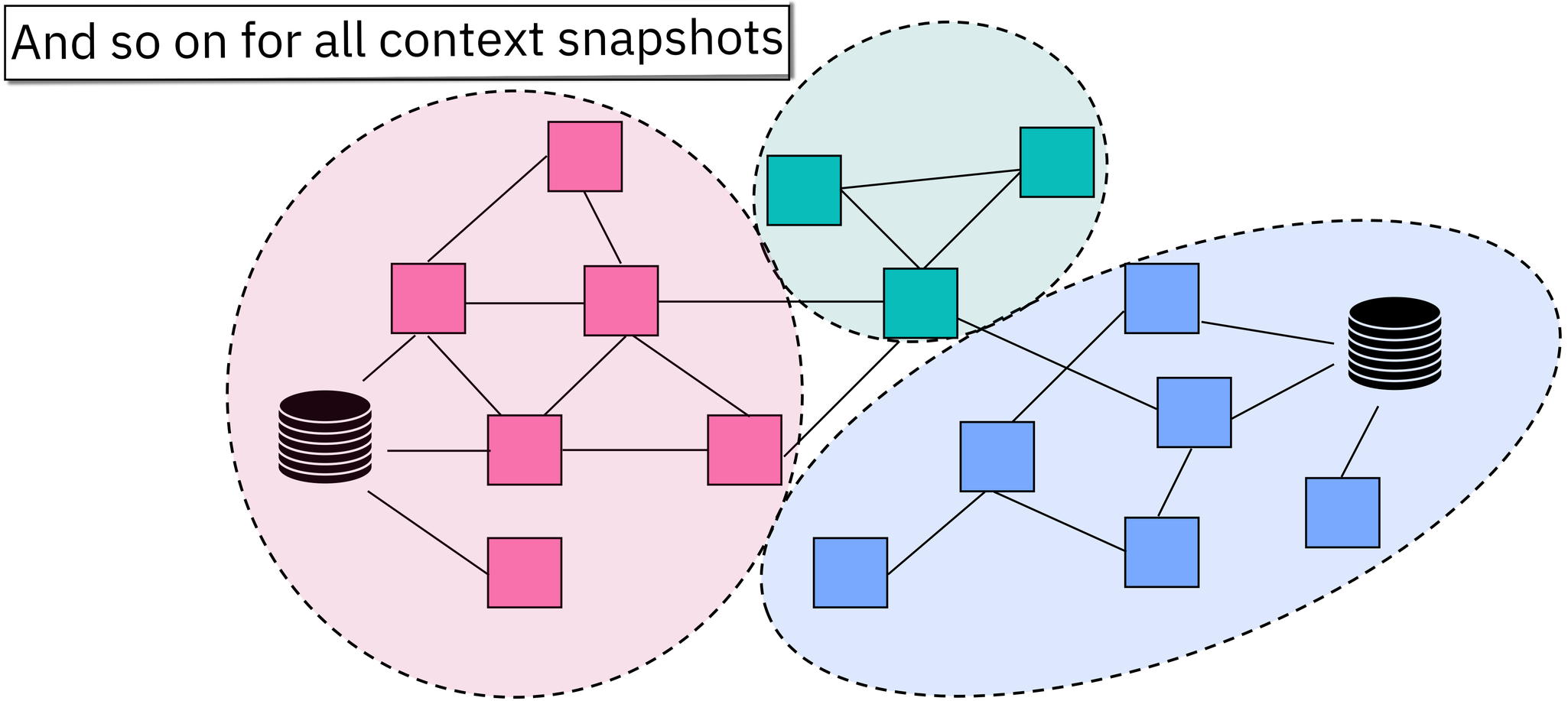


Context-sensitive Label Propagation

And so on for all context snapshots

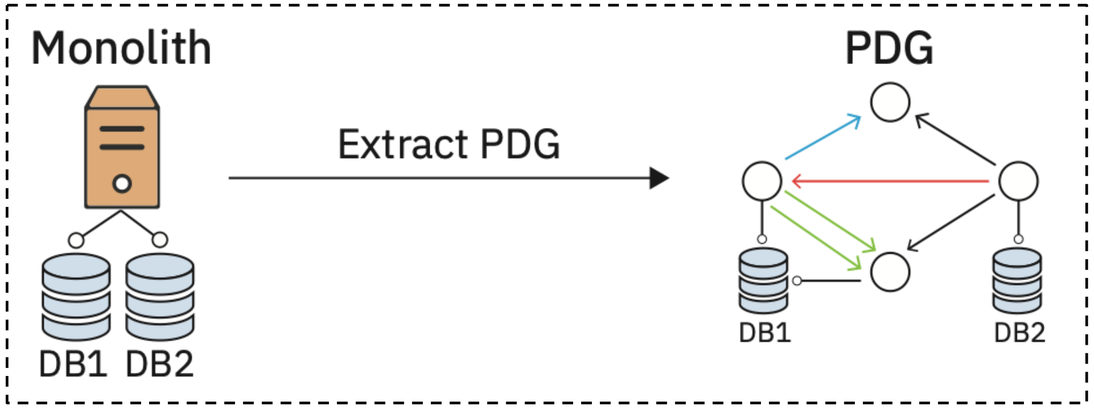


Context-sensitive Label Propagation

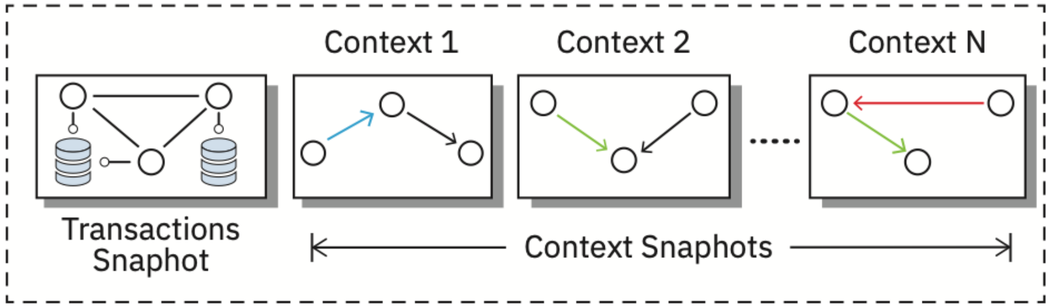


CARGO: Summary

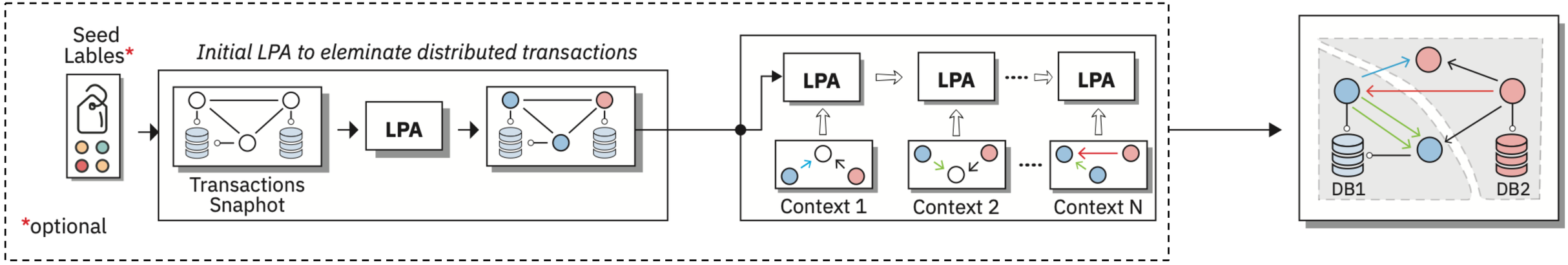
Step I: Build *context-sensitive program dependency graph*



Step II: Extract *context snapshots*



Step III: **Deploy** *Context-sensitive Label Propagation*



Microservice Boundaries

Evaluation

Evaluation Setup - Benchmark Applications

Application	Description	Java Framework	# Classes	# SQL Tables
Daytrader	Trading application	Java EE 8, Websphere	109	6
Plants	Online plant shopping	Java EE 7, Websphere	33	.
AcmeAir	Website of a fictitious airline	Openliberty, Websphere eXtreme	66	.
JPetStore	Online pet supply store	Spring, Springboot	37	.
Proprietary1	Proprietary app	.	82	.

Evaluation Setup - Baseline Approaches

Approach	Summary
Mono2Micro*	Dynamic call traces and hierarchical clustering.
CoGCN	A Graph Neural Network and K-Means on a static call graph
FoSCI	Genetic Search-based algorithm on dynamic execution traces
MEM	A Minimum-Spanning Tree based Clustering Algorithm on a graph. Edit-history and semantics are used to define coupling

** Enterprise scale decomposition tool*

Evaluation Setup - “refining” partitions

CARGO can be used to refine the partitions produced by other approaches.

❑ Denoted by “++” suffix.

E.g., *Mono2Micro++ denotes running CARGO with initial partition labels produced by Mono2Micro.*

Original Approach	Refined with CARGO
Mono2Micro	Mono2Micro++
CoGCN	CoGCN++
FoSCI	FoSCI++
MEM	MEM++

Research Questions

RQ-1 Effectiveness in remediating distributed transactions

RQ-2 Latency and Throughput improvements resulting from refined microservice partitions

RQ-3 Quality of microservice partition architectural metrics

RQ-1 Distributed database transactions

To minimize distributed transactions, we would like, to the extent possible, for each database table to be accessed from one microservice partition only

Evaluation

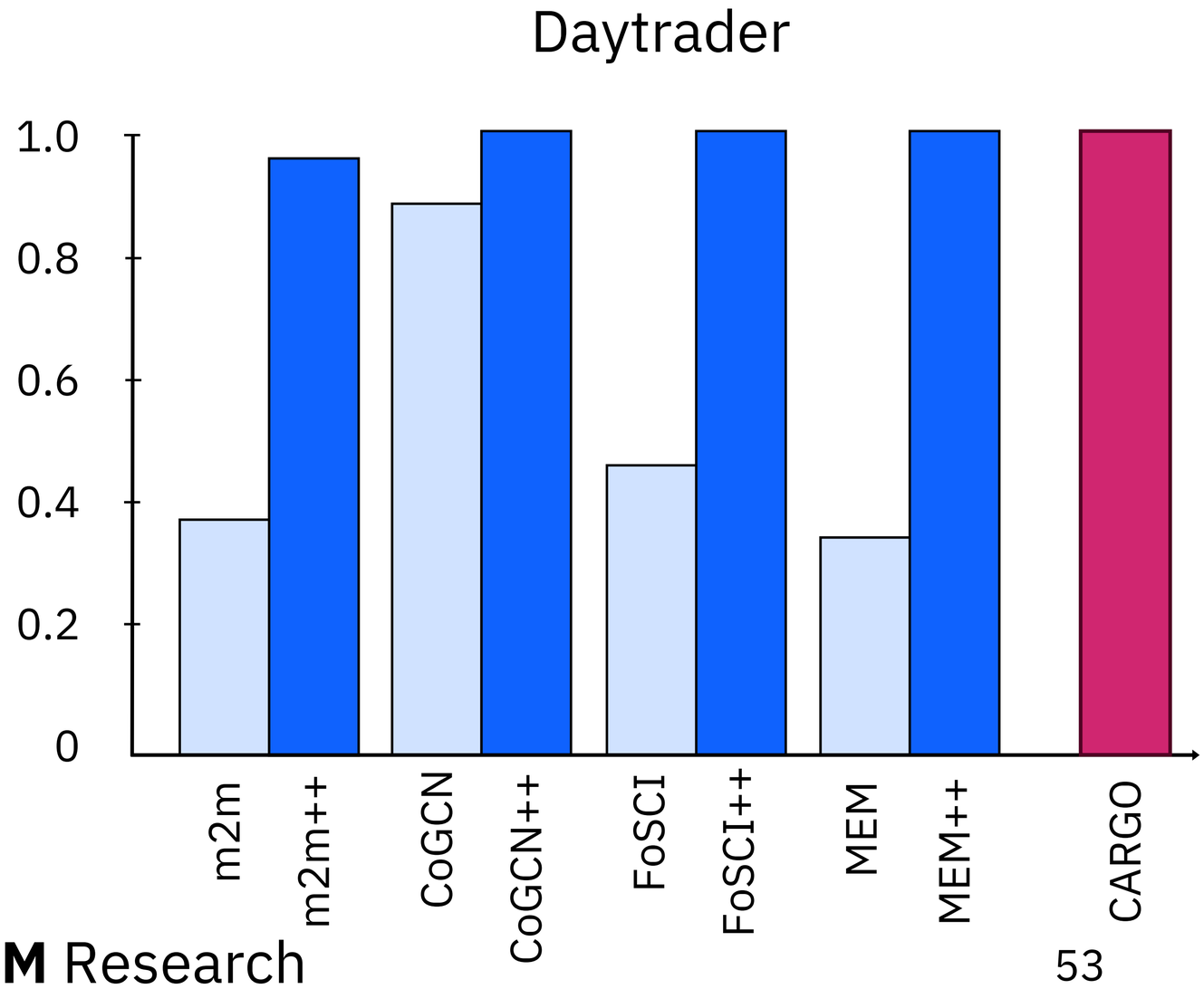
- Use Transactional Purity (TXP) to measure the tendency of a database table to be accessed by multiple microservices

Entropy

$$TXP = 1 - \sum_{i=0}^{K} p_i \cdot log\left[\frac{1}{p_i}\right]$$

- Lower purity indicates accesses from more microservices
- Higher purity indicates accesses from less microservices

RQ-1 Distributed database transactions



Summary

- For each of the 4 baselines, the refined partitions have higher transactional purity.
- FoSCI++ and MEM++ have transactional purity of 1.0, (i.e., no distributed transactions after repartitioning).
- CARGO (unsupervised) natively achieves transactional purity of 1.0

++ implies refinement with CARGO

Research Questions

RQ-1 Effectiveness in remediating distributed transactions

RQ-2 Latency and Throughput improvements resulting from refined microservice partitions

RQ-3 Quality of microservice partition architectural metrics

RQ-2 Runtime Performance Improvements

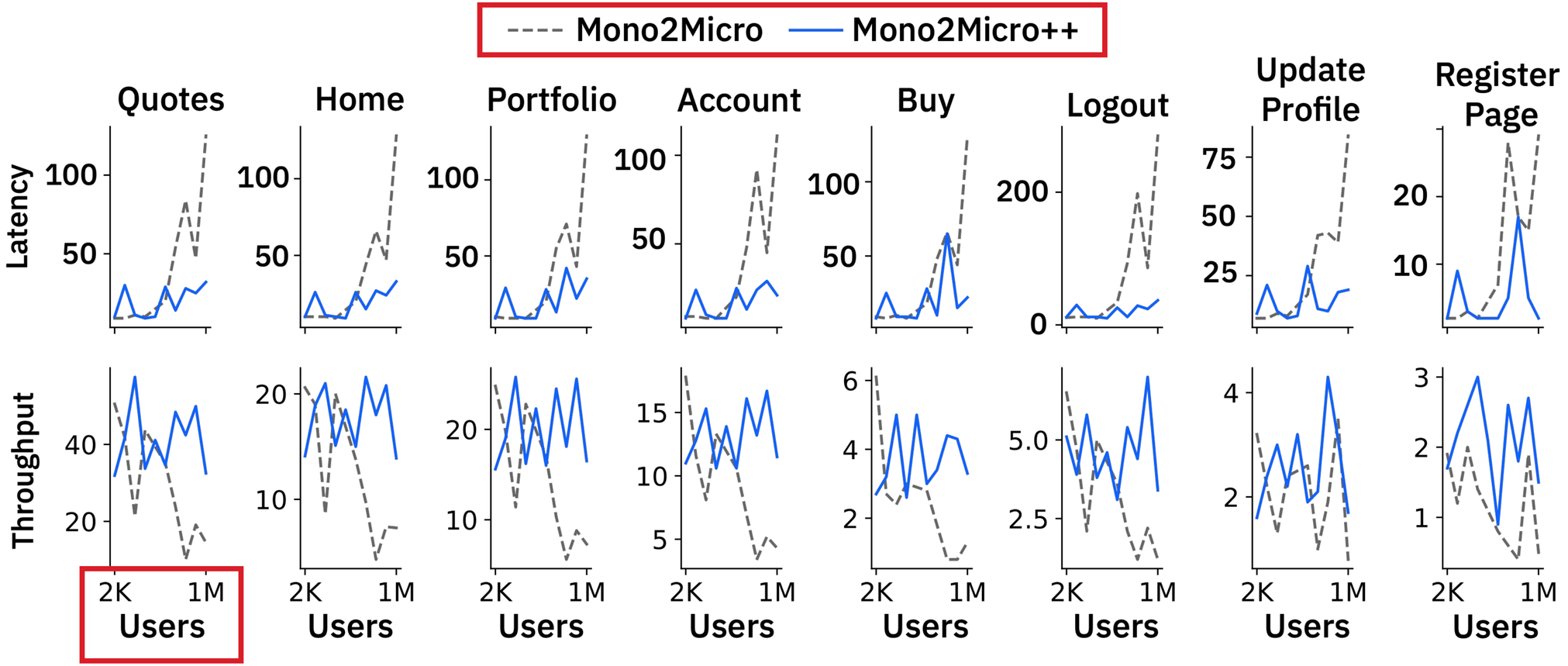
Minimizing distributed transactions can offer significant runtime benefits in terms of reduced latency and improved throughput.

Evaluation

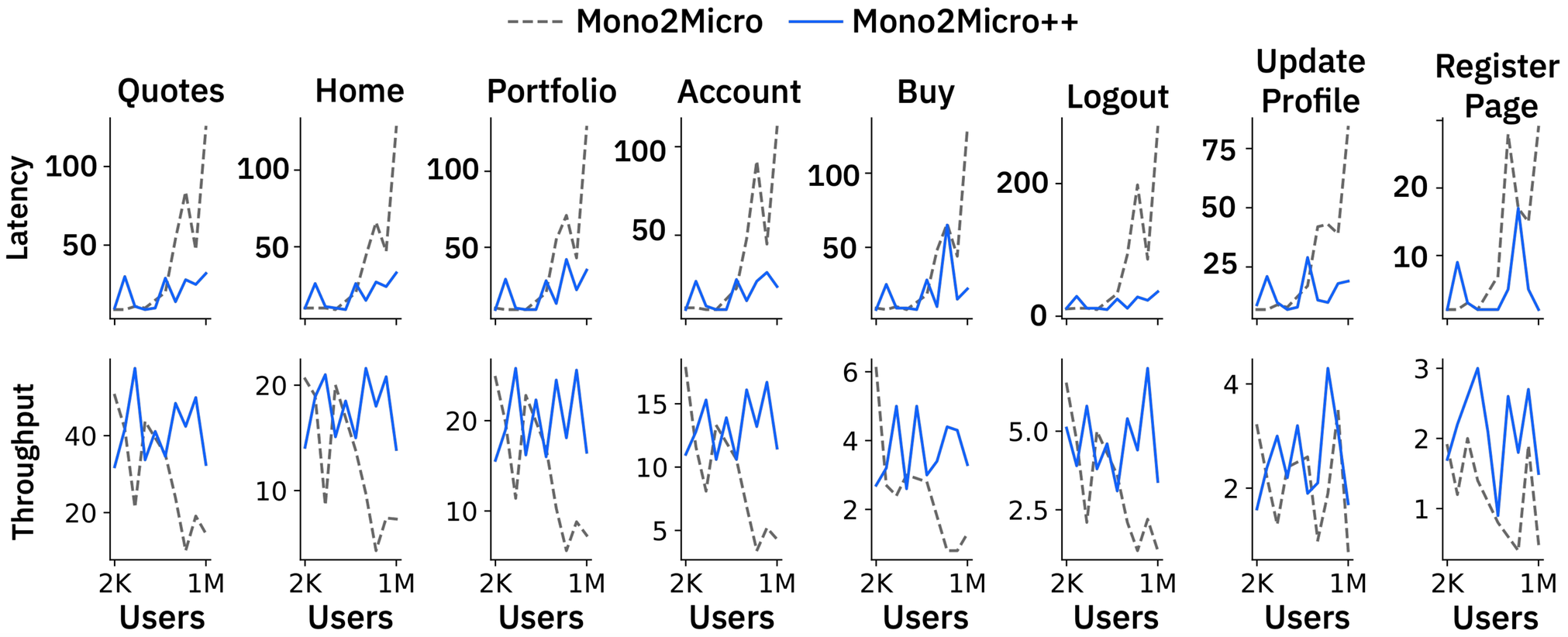
- ❑ Deploy two variants of the applications:
  1. Application with original partitioning (Mono2Micro) [1]
  2. Application with partitions refined with CARGO (aka. Mono2Micro++) [2]
- ❑ Compare two runtime performance metrics:
  1. Latency: Time between reception and completion of a request (milliseconds)
  2. Throughput: Number of successful requests honored per unit time (requests/second)

[1] https://github.com/vikramnitin9/tackle-data-gravity-insights/tree/main/RQ2/daytrader_apps/daytrader_cargo
[2] https://github.com/vikramnitin9/tackle-data-gravity-insights/tree/main/RQ2/daytrader_apps/daytrader_mono2micro

RQ-2 Runtime Performance Improvements



RQ-2 Runtime Performance Improvements



Significant improvements in latency and throughput. Repartitioned application has 11% lower latency and 120% higher throughput on average across use cases compare to the original deployment.

Research Questions

RQ-1 Effectiveness in remediating distributed transactions

RQ-2 Latency and Throughput improvements resulting from refined microservice partitions

RQ-3 Quality of microservice partition architectural metrics

RQ-3 Partitions and their Architectural Quality

METRIC		DESCRIPTION
Coupling	▽	Average Coupling among partitions
Cohesion	△	Average cohesion within a partition
BCP	▽	Purity of Business use cases per partition.
ICP	▽	Inter-partition call volume

▽ Lower is better △ Higher is better

RQ-3 Partitions and their Architectural Quality

Coupling ▽

	Mono2Micro	Mono2Micro++	CARGO
Daytrader	0.78	0.02	0.01
Plants	0.31	0.04	0.05
AcmeAir	0.58	0.04	0.03
JPetstore	0.77	0.03	0.03
Proprietary	0.42	0.03	0.04
Win/Tie/Loss	5/0/0		

Cohesion △

	COGCN	COGCN++	CARGO
Daytrader	0.37	0.61	0.71
Plants	0.39	0.46	0.6*
AcmeAir	0.21	0.32	0.96
JPetstore	0.20	0.24	0.94
Proprietary	0.69	0.73	0.75*
Win/Tie/Loss	5/0/0		

*Mono2Micro++ performs slightly better than CARGO

CARGO improves the partitioning quality (reduced coupling and increased cohesion) of other approaches and works equally well in unsupervised mode.

RQ-3 Partitions and their Architectural Quality

BCP ▽

	Mono2Micro	Mono2Micro++	CARGO
Daytrader	2.31	2.57	1.31
Plants	1.68	2.20	1.79
AcmeAir	1.29	1.48	1.75
JPetstore	2.25	2.35	2.87
Proprietary	1.53	1.23	1.55
Win/Tie/Loss	0/0/5		

CARGO performs poorly on BCP. The definition of BCP depends heavily on the quality of the generated business use cases, which Mono2Micro has access to but we do not.

Summary

- Existing automated approaches miss key code and/or transactional dependencies
- We present CARGO, which uses (a) precise static analysis, (b) explicit modeling of database transactions, and (c) a novel community detection algorithm
- Compared to existing approaches, CARGO (a) reduces distributed transactions, (b) achieves better latency and throughput, (c) achieves better performance on architectural metrics