ECOOP'25
Bergen

TEXAS
The University of Texas at Austin

In-memory Object Graph Stores


Aditya Thimmaiah


Zijian Yi

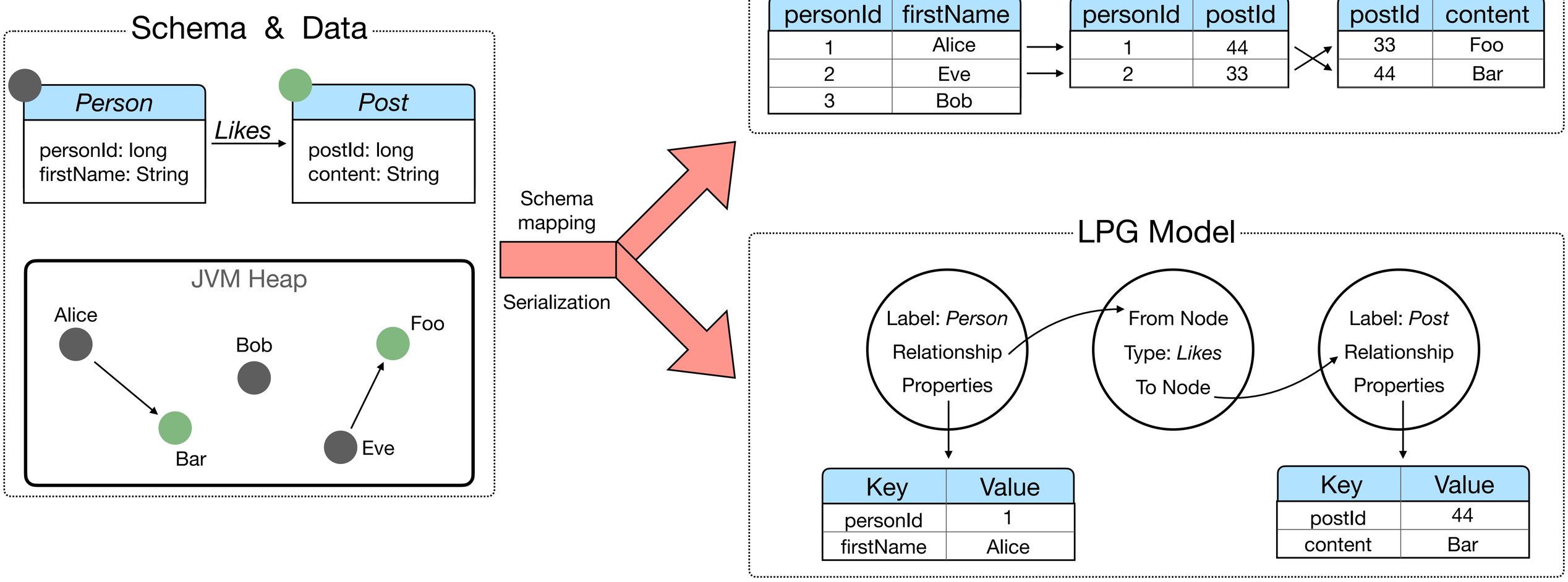

Joseph Kenis


Christopher Rossbach


Milos Gligoric

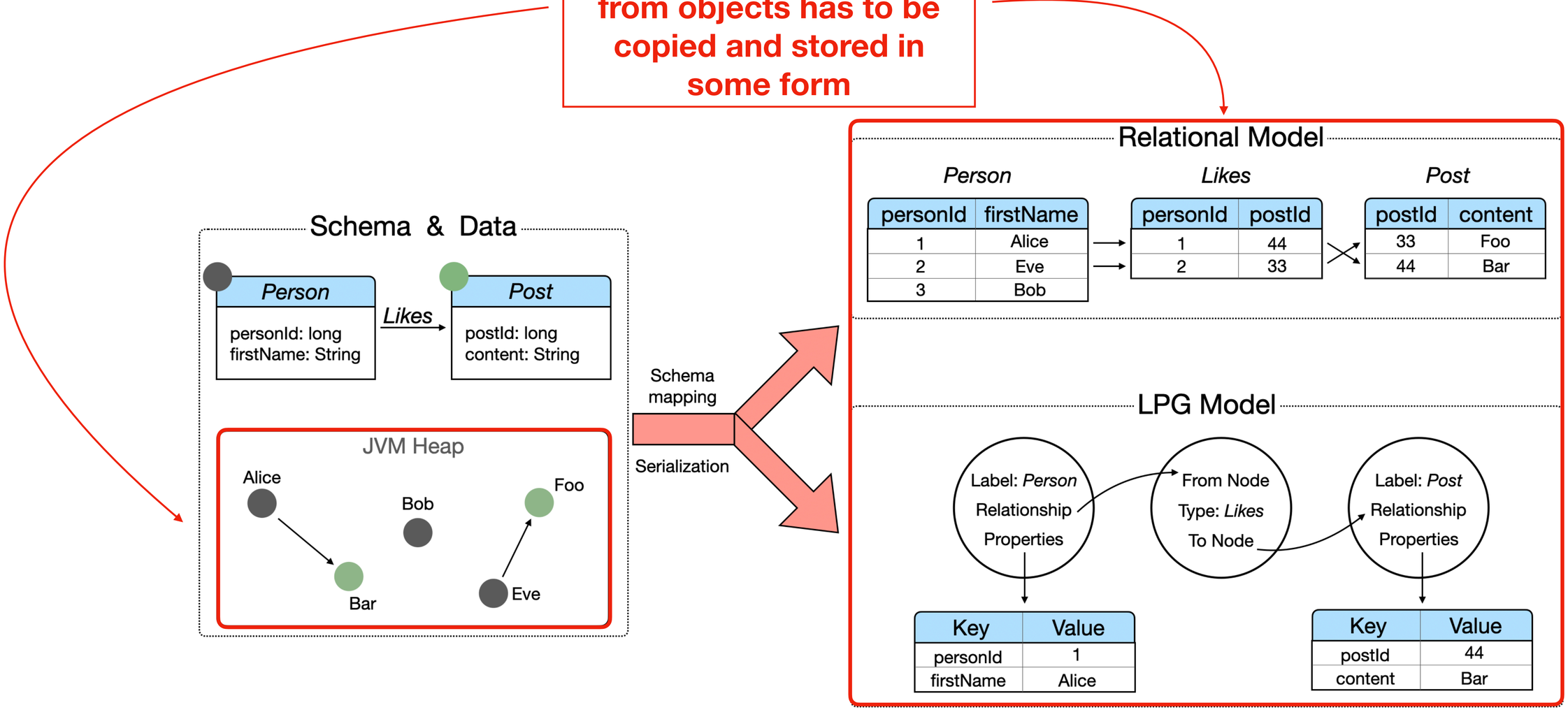
CCF-2107291
CCF-2217696
CCF-2313027
CCF-2403036

Motivation


Schema & Data
Person
personId: long
firstName: String
Likes
Post
postId: long
content: String
JVM Heap
Alice
Bob
Foo
Bar
Eve
Schema mapping
Serialization
Relational Model
Person
personId
firstName
1
Alice
2
Eve
3
Bob
Likes
personId
postId
1
44
2
33
Post
postId
content
33
Foo
44
Bar
LPG Model
Label: Person
Relationship
Properties
From Node
Type: Likes
To Node
Label: Post
Relationship
Properties
Key
Value
personId
1
firstName
Alice
Key
Value
postId
44
content
Bar

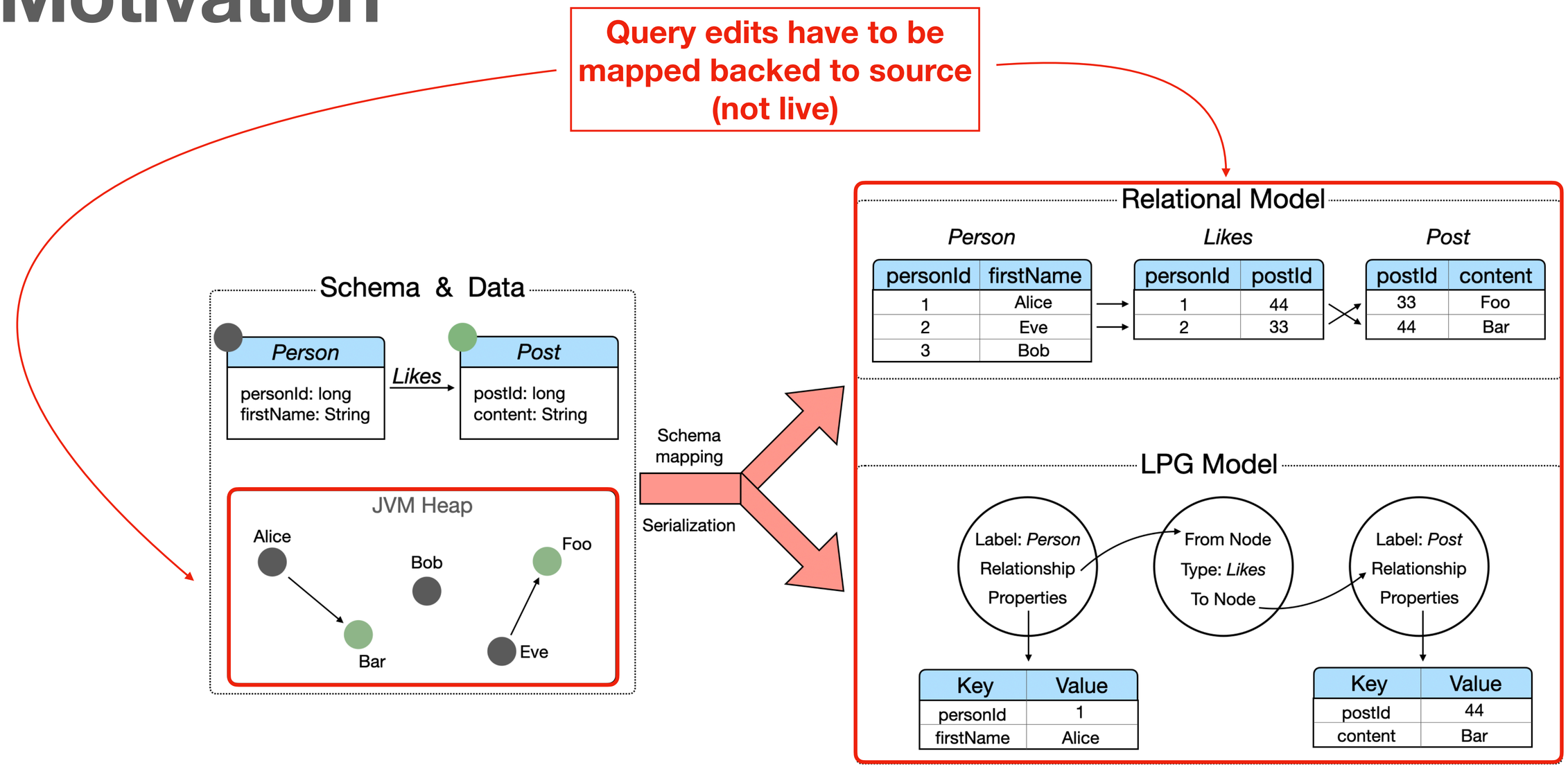
Motivation

Disconnect between storage models — data from objects has to be copied and stored in some form



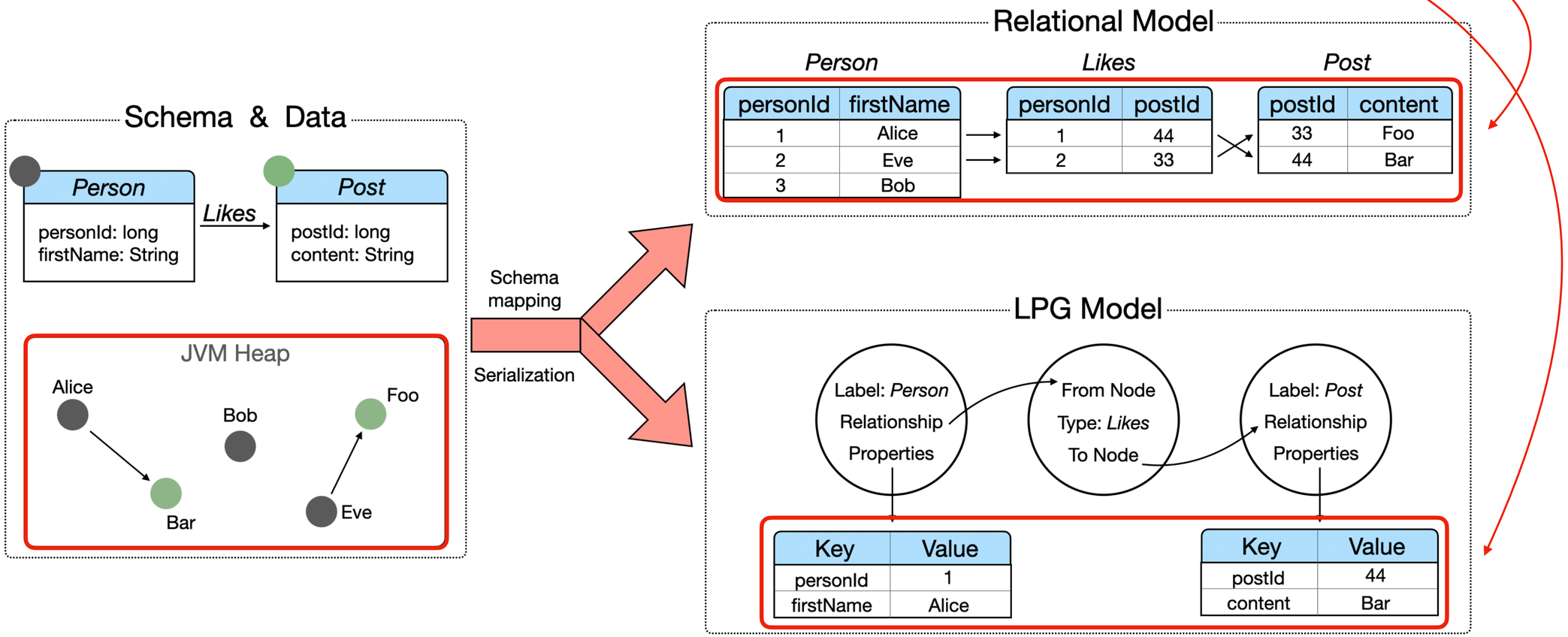
Motivation

Query edits have to be mapped backed to source (not live)



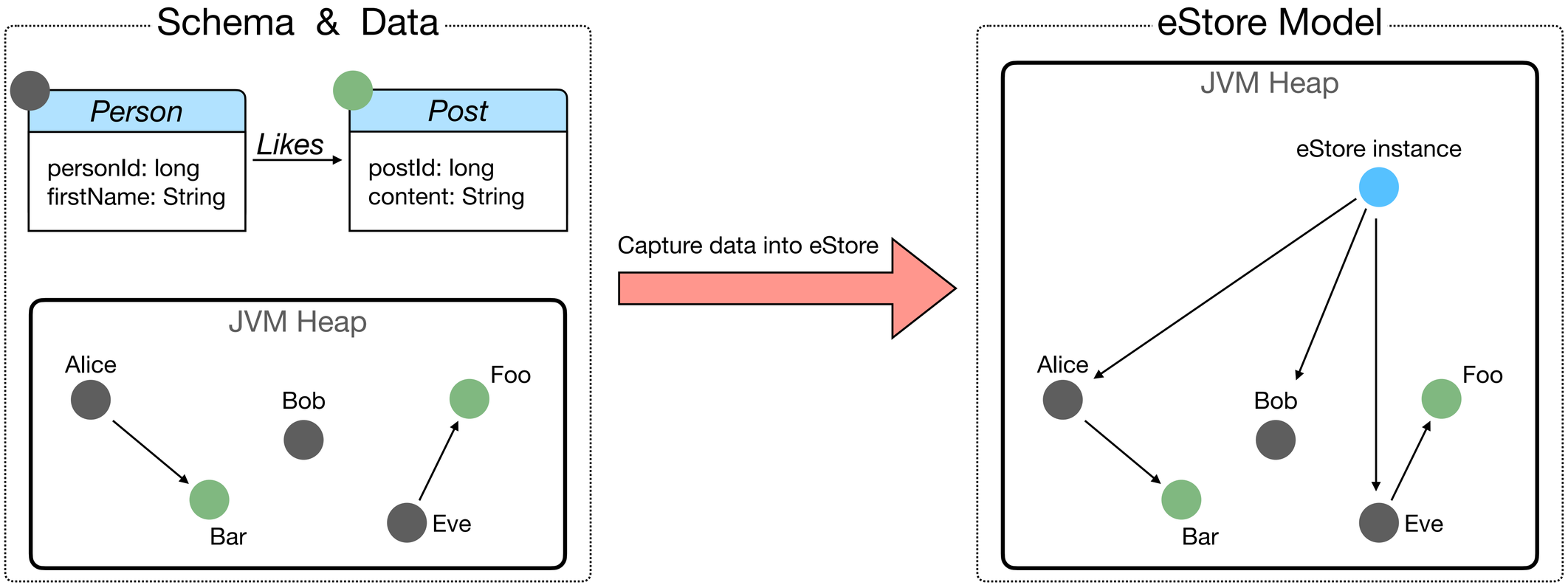
Motivation

Heavy abstraction and bloat — not ideal for quick setup and testing environments



Motivation

eStore is motivated by a simple observation: *your heap is already a richly connected graph—stop copying it, and start querying it directly.*



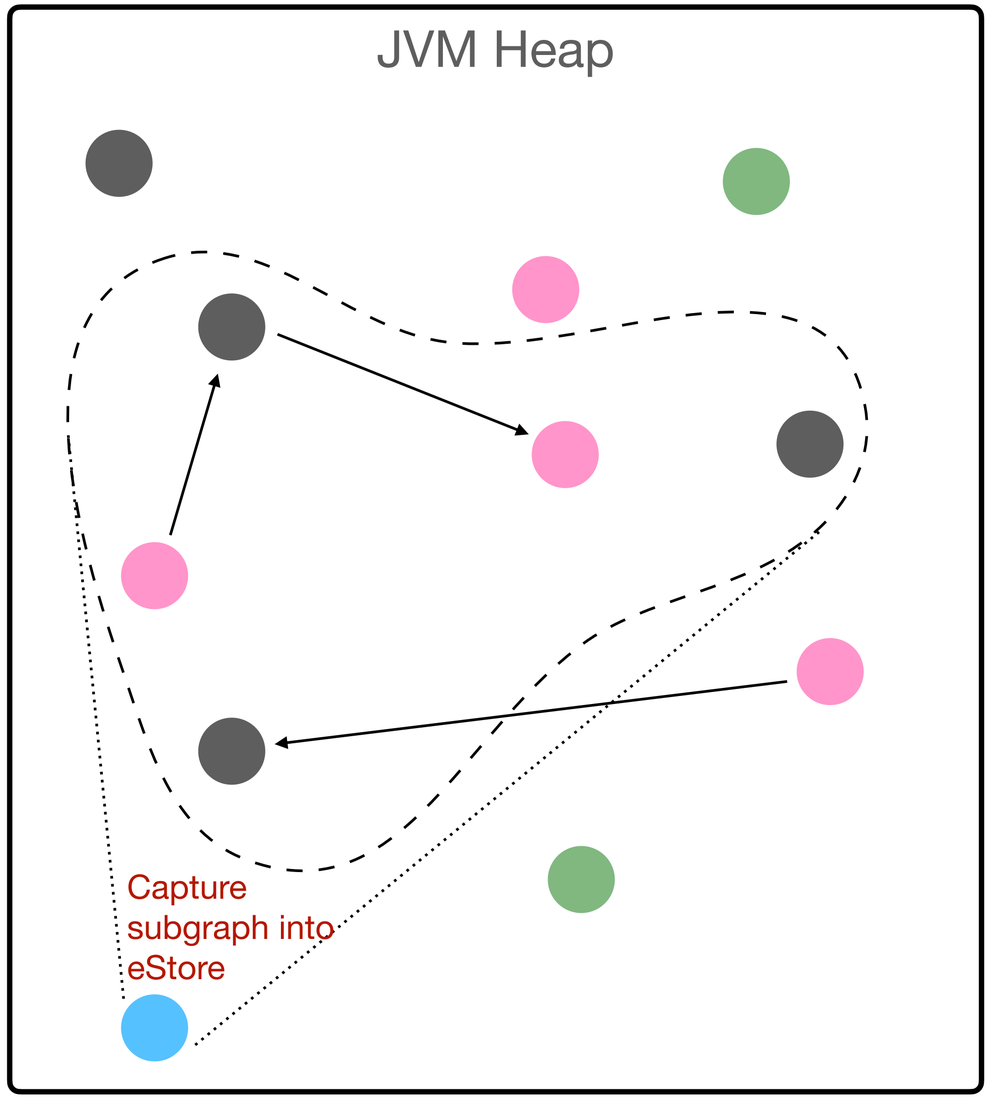
Our Contributions

- eStore: An in-memory object graph store that allows declarative queries over live objects and interoperability between imperative code and stored object
- Epsilon storage model: Allows storing objects with zero abstraction
- Compile-time query rewriting into imperative code to reduce runtime cost
- Formalization of the epsilon storage model
- Evaluation
  - Demonstrate robustness and lightweight design of eStore by evaluating on the LDBC SNB benchmark
  - Demonstrate eStore superior performance over existing heap querying frameworks (OGO)
  - Demonstrate the performance improvement with compile-time query rewriting

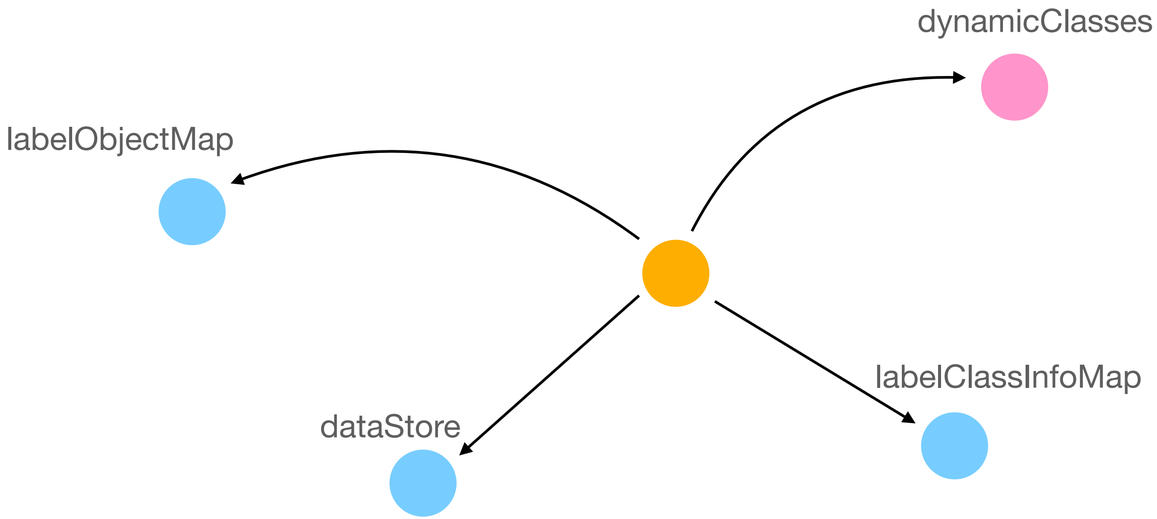
Insight behind eStore

- We see the heap of the JVM as a graph whose subgraphs can be captured into a graph object store and queried
  - Objects as nodes
  - References as edges
- We use expressive declarative graph query language Cypher to query the graph object store
- Example:

```
1| public void testAcyclicity(){
2|    List<Long> list = new LinkedList<Long>();
3|    … //Add nodes to the list
4|    EpsilonStore db = new EpsilonStore("db");
5|    db.captureAll(list);
6|    assertTrue(db.query(
7|      "MATCH (n:`LinkedList$Node`)-[:next*]->(n)"
8|   +" RETURN COUNT(n) = 0”).getBoolean(0));
9|}
```



Example | Creating eStore instance




```
1| EpsilonStore db = new EpsilonStore ("dbname");
```

Example | Creating eStore instance

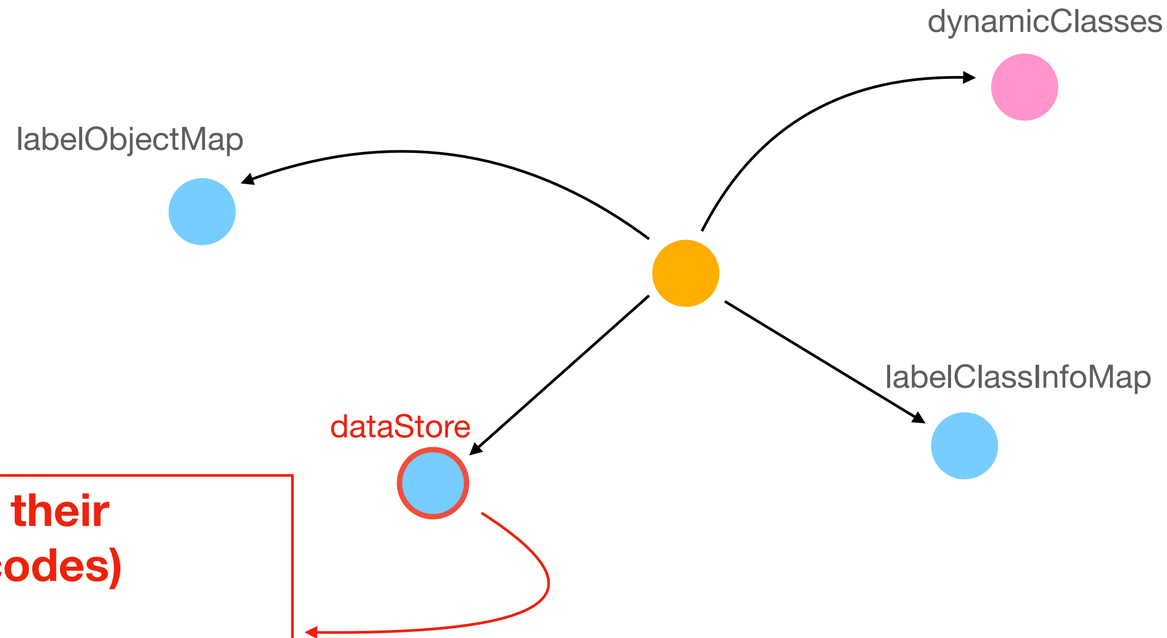
Stores objects by their class names (labels)

=>

Optimizes queries by restricting scans to stored objects belonging to label

```
1| EpsilonStore db = new EpsilonStore ("dbname");
```

Example | Creating eStore instance



Stores objects by their unique ID's (hashcodes)

=>

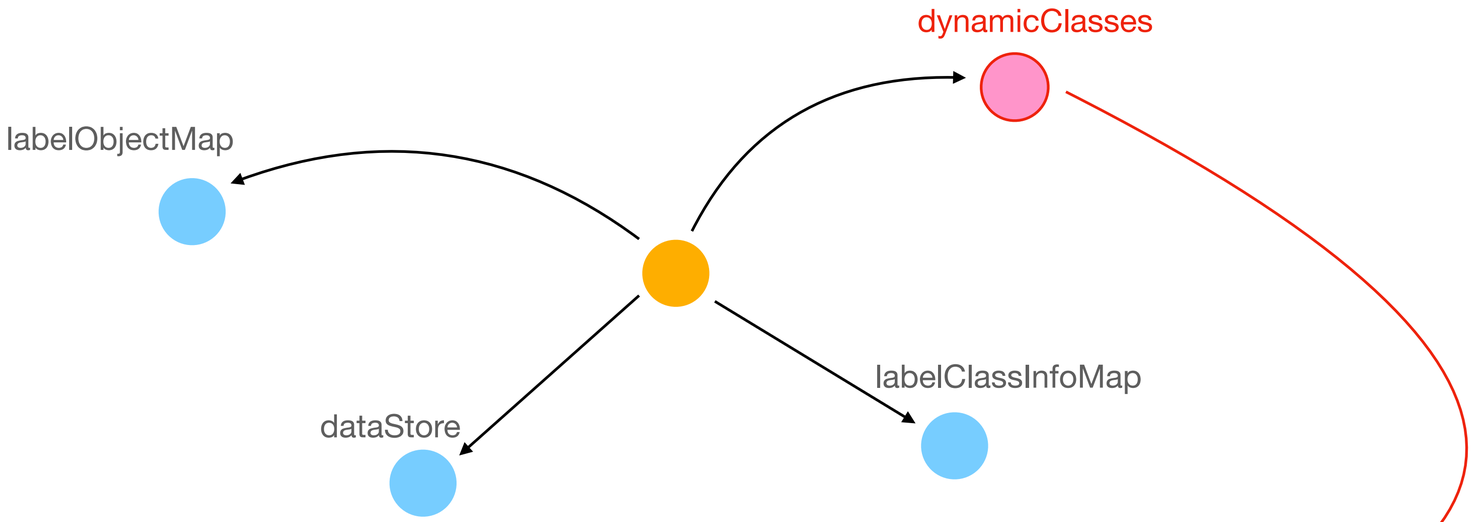
Optimizes queries by restricting scans to object with specified ID

```
1| EpsilonStore db = new EpsilonStore ("dbname");
```

LEGEND

- Class instance
- ArrayList instance
- HashMap instance
- eStore instance
- Post instance
- Person instance
- instanceof relation
- Key
- HashMap instance
- List element relation

Example | Creating eStore instance



Stores classes created dynamically (runtime)

=>

Supports querying without pre-defined/ explicit schema

```
1| EpsilonStore db = new EpsilonStore ("dbname");
```

Example | Creating eStore instance

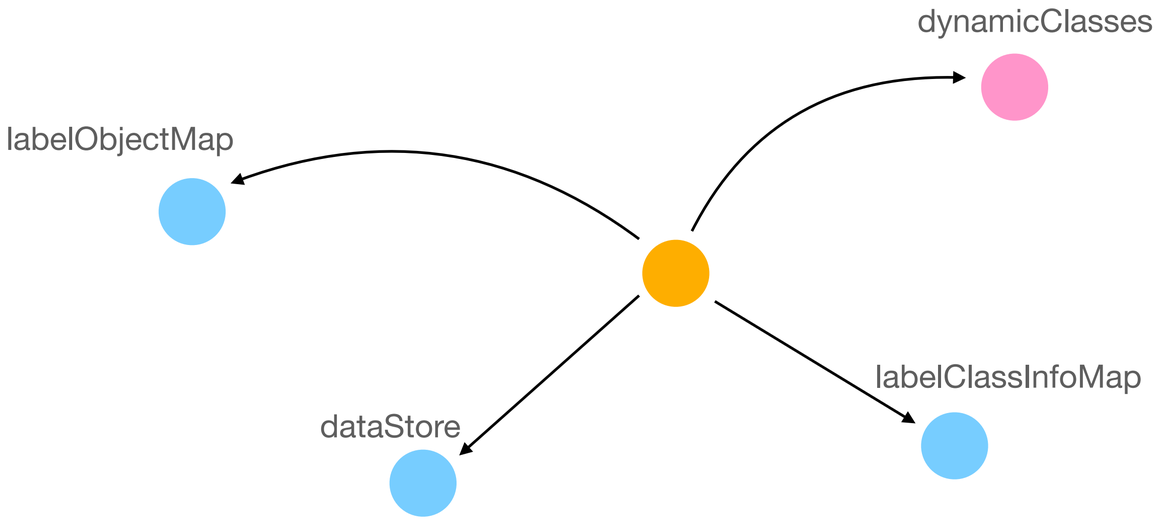
Maps classes of stored objects to their meta prop objects (ClassInfo)

=>

Optimizing queries containing node properties by caching class field names and types

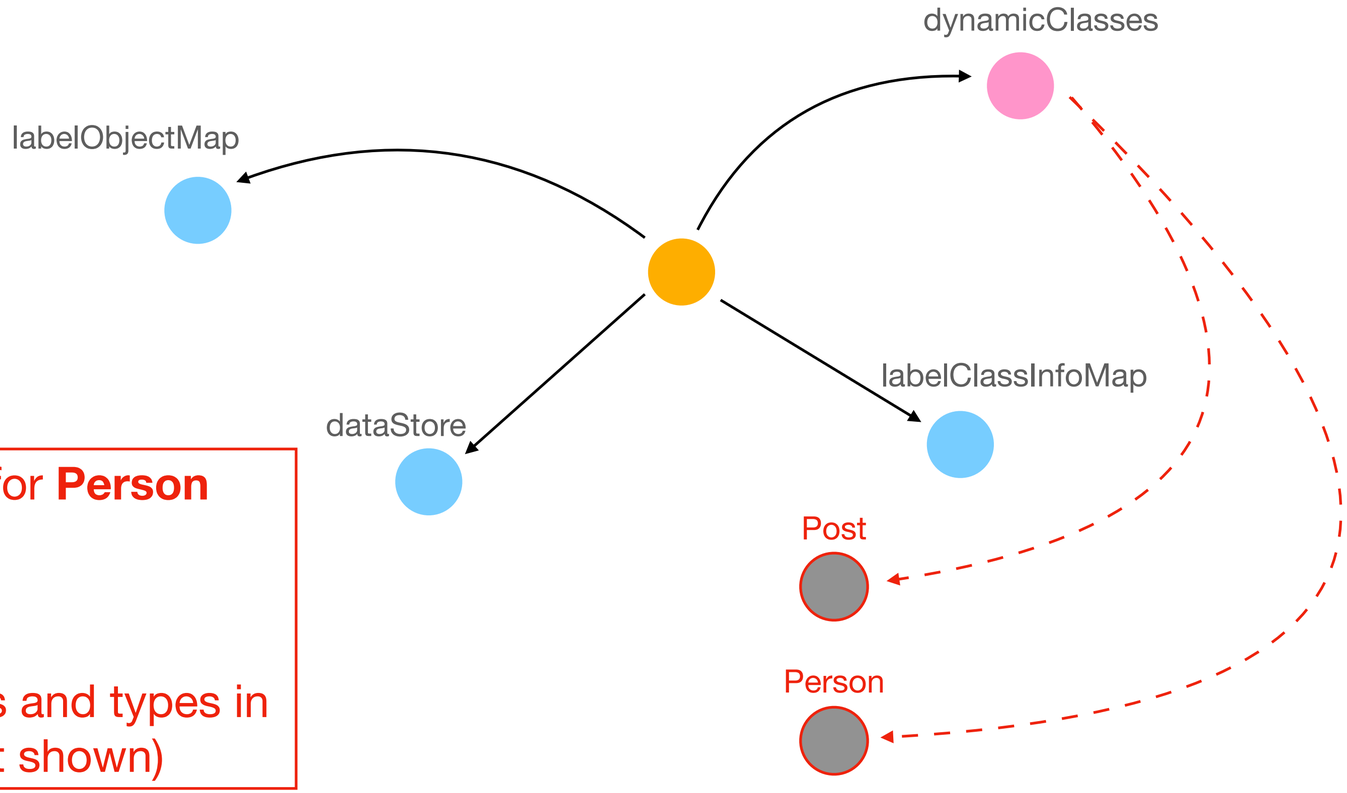
```
1| EpsilonStore db = new EpsilonStore ("dbname");
```

Example | Adding data without pre-defined schema




```
1| db.query(
2| "CREATE (m:`Person` {id: 112, firstName: ‘Eve’}"
3|+"-[r:LIKES]->"
4|+"(n:`Post` {id: 481, content: ‘About Databases’})"
5|);
```

Example | Adding data without pre-defined schema

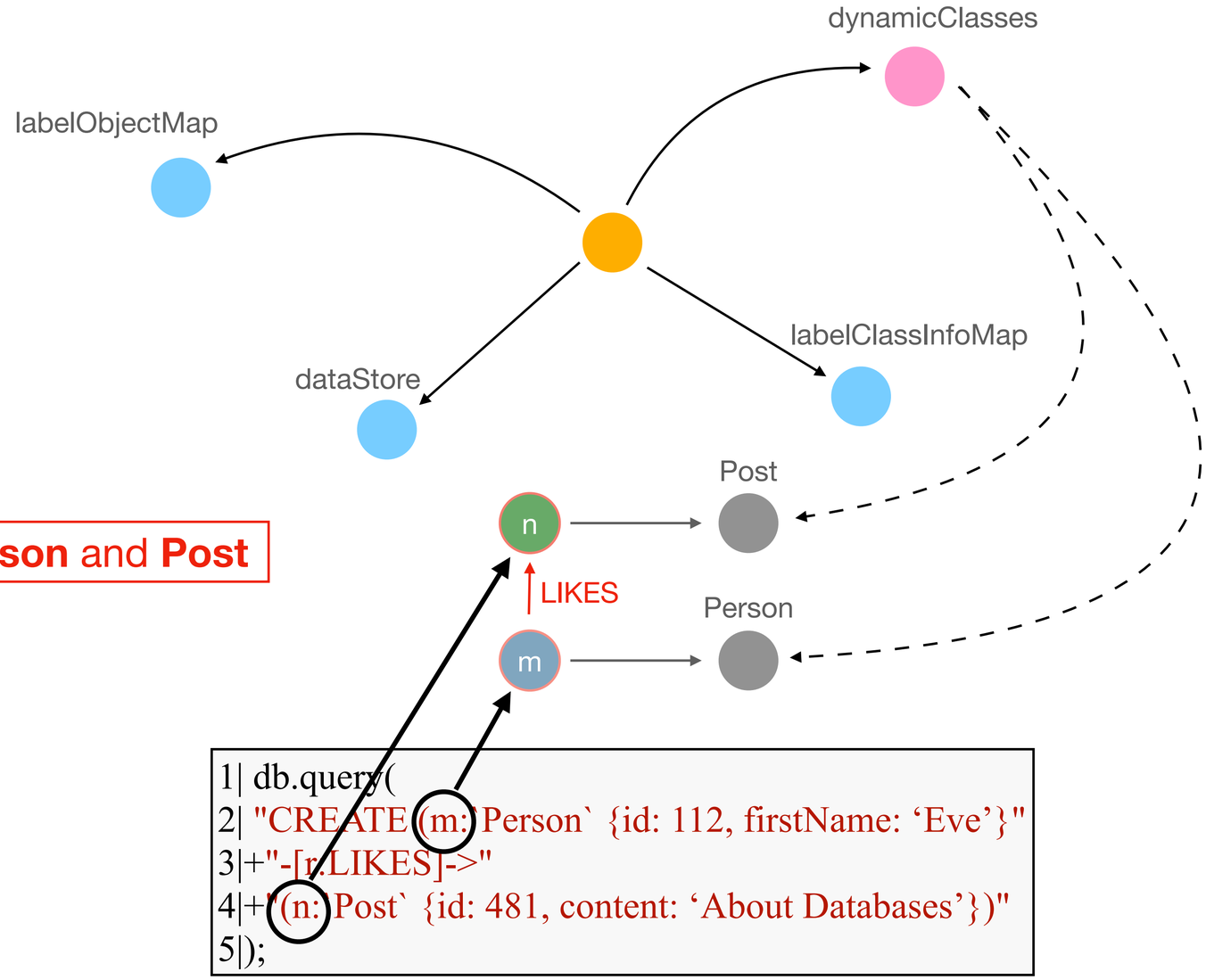


1. Runtime class creation for **Person** and **Post** using **ASM**
2. Store created classes in **dynamicClasses**
3. Cache class field names and types in **labelClassInfoMap** (not shown)

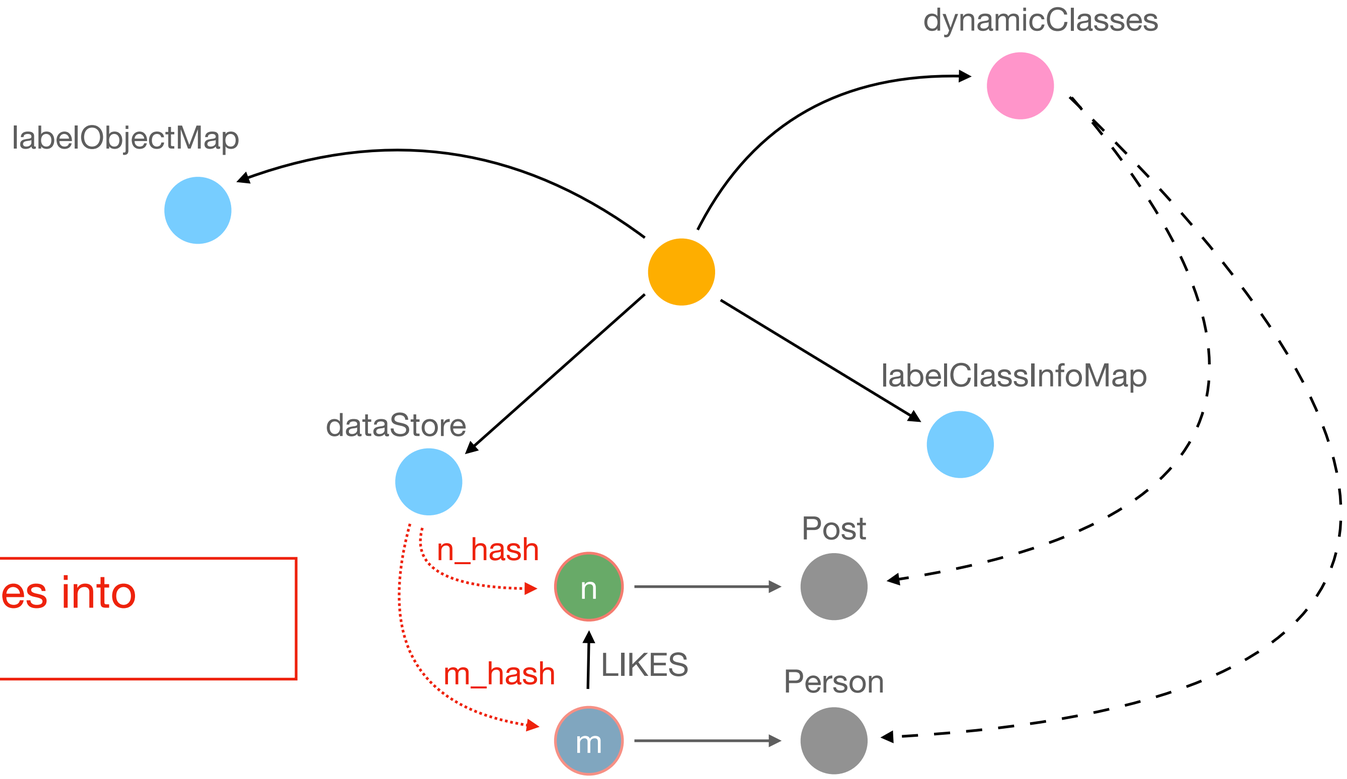
```
1| db.query(
2| "CREATE (m:`Person` {id: 112, firstName: ‘Eve’}"
3|+"-[r:LIKES]->"
4|+"(n:`Post` {id: 481, content: ‘About Databases’})"
5|);
```

Example | Adding data without pre-defined schema

1. Create instances of **Person** and **Post**



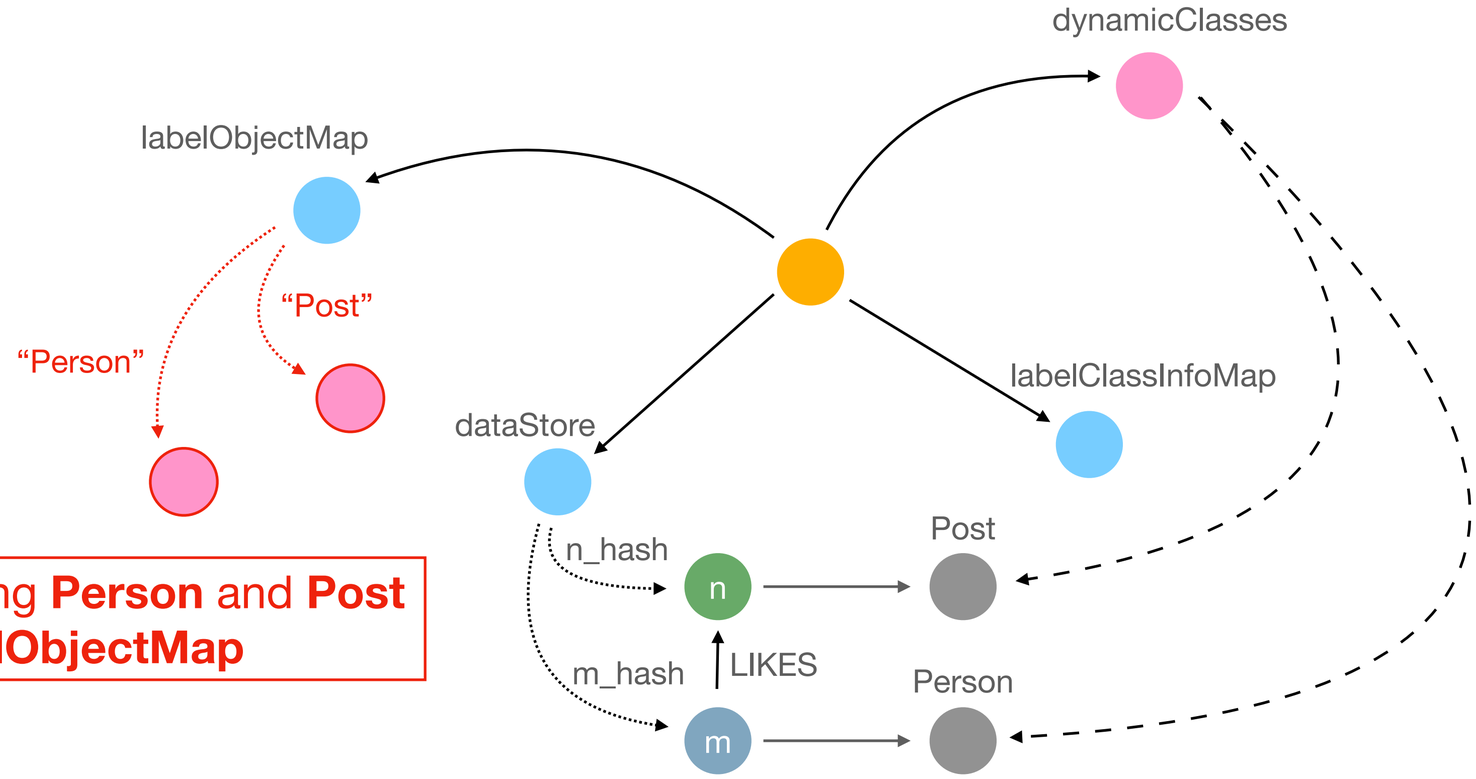
Example | Adding data without pre-defined schema



1. Capture created instances into **datastore**

```
1| db.query(
2| "CREATE (m:`Person` {id: 112, firstName: 'Eve'}"
3|+"-[r:LIKES]->"
4|+"(n:`Post` {id: 481, content: 'About Databases'})"
5|);
```

Example | Adding data without pre-defined schema



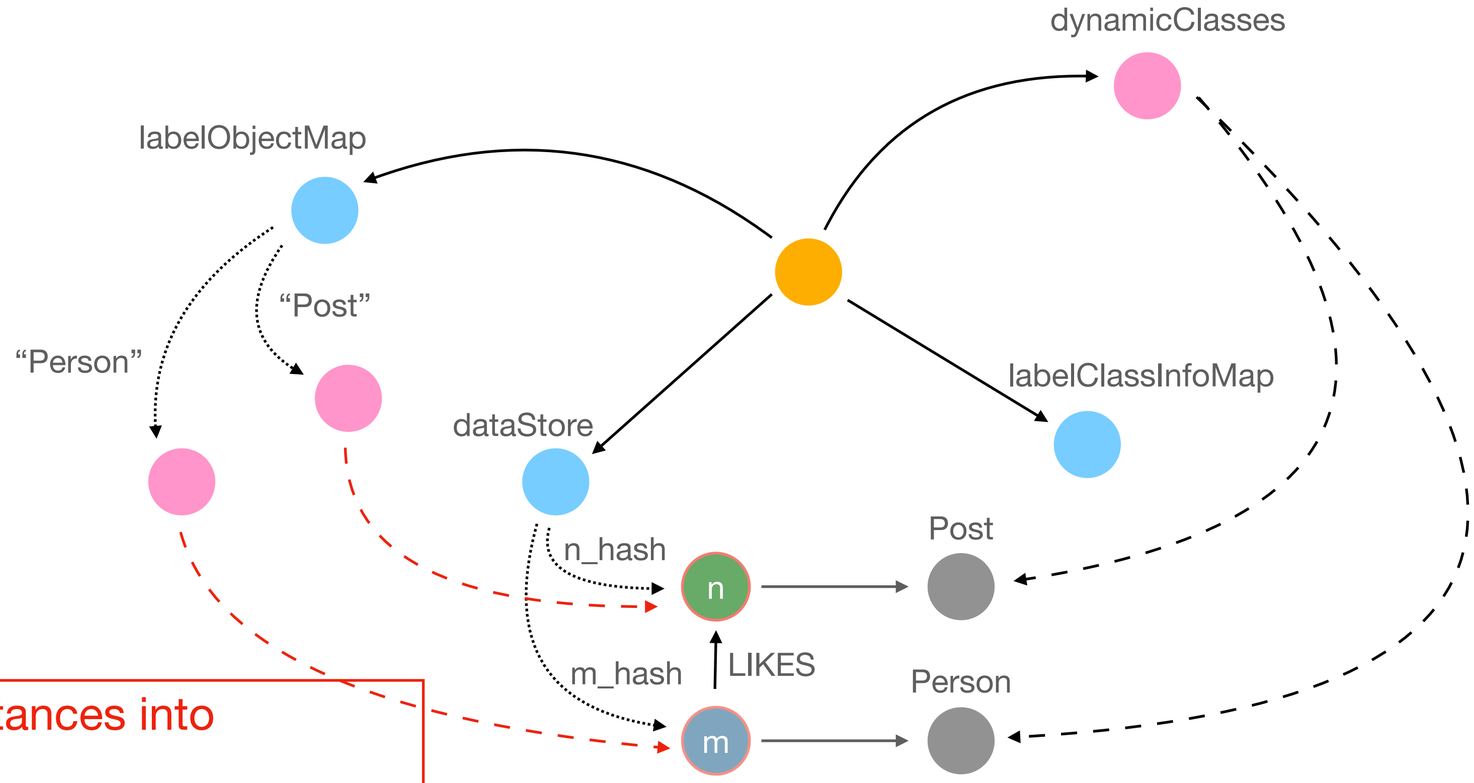
1. Add lists for storing **Person** and **Post** instances to **labelObjectMap**

```
1| db.query(
2| "CREATE (m:`Person` {id: 112, firstName: 'Eve'}"
3|+"-[r:LIKES]->"
4|+"(n:`Post` {id: 481, content: 'About Databases'})"
5|);
```

LEGEND

- Class instance
- ArrayList instance
- HashMap instance
- eStore instance
- Post instance
- Person instance
- instanceof relation
- Key
- HashMap instance
- List element relation

Example | Adding data without pre-defined schema



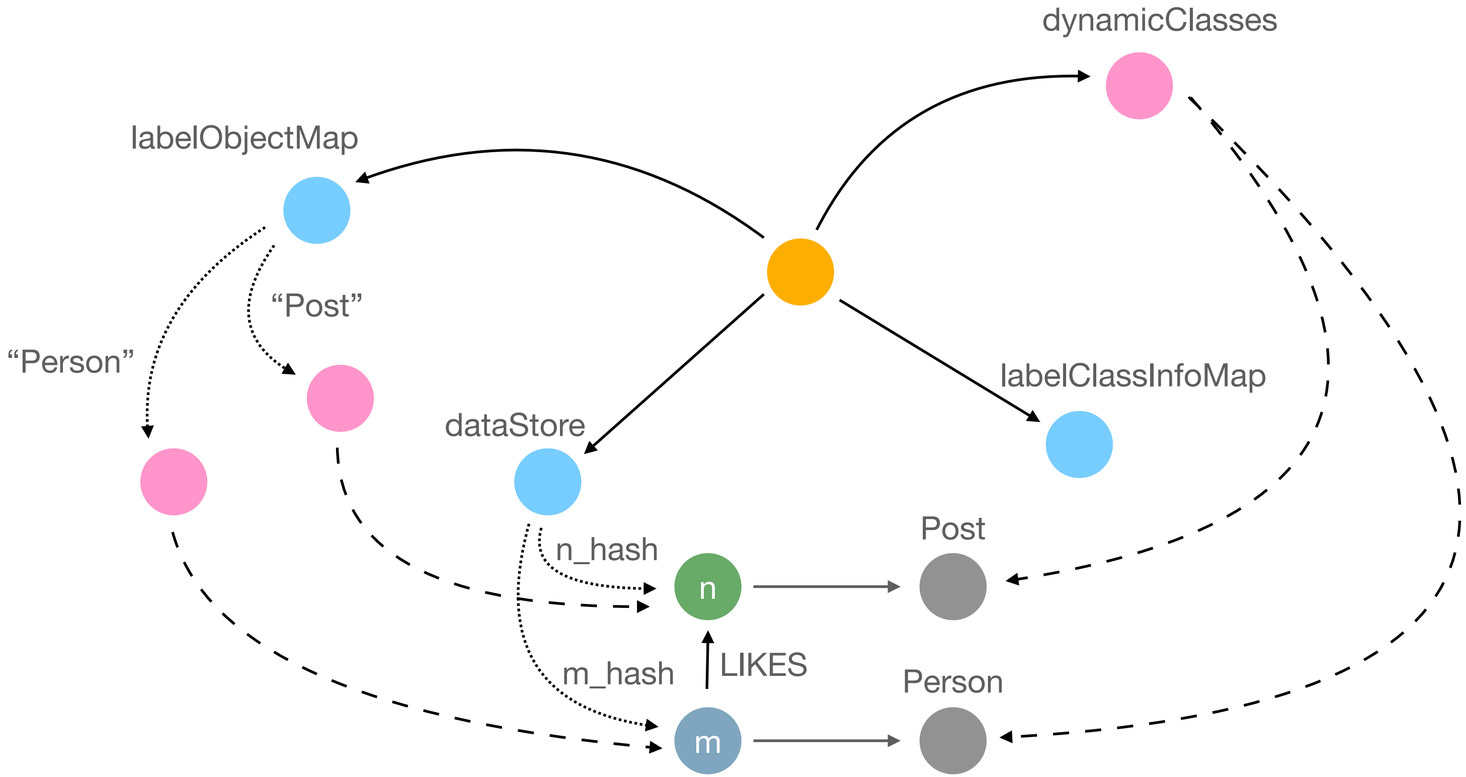
1. Insert created instances into **labelObjectMap**

```
1| db.query(
2| "CREATE (m:`Person` {id: 112, firstName: 'Eve'}"
3|+"-[r:LIKES]->"
4|+"(n:`Post` {id: 481, content: 'About Databases'})"
5|);
```

LEGEND

- Class instance
- ArrayList instance
- HashMap instance
- eStore instance
- Post instance
- Person instance
- instanceof relation
- Key
- HashMap instance
- List element relation

Example | Deleting edges

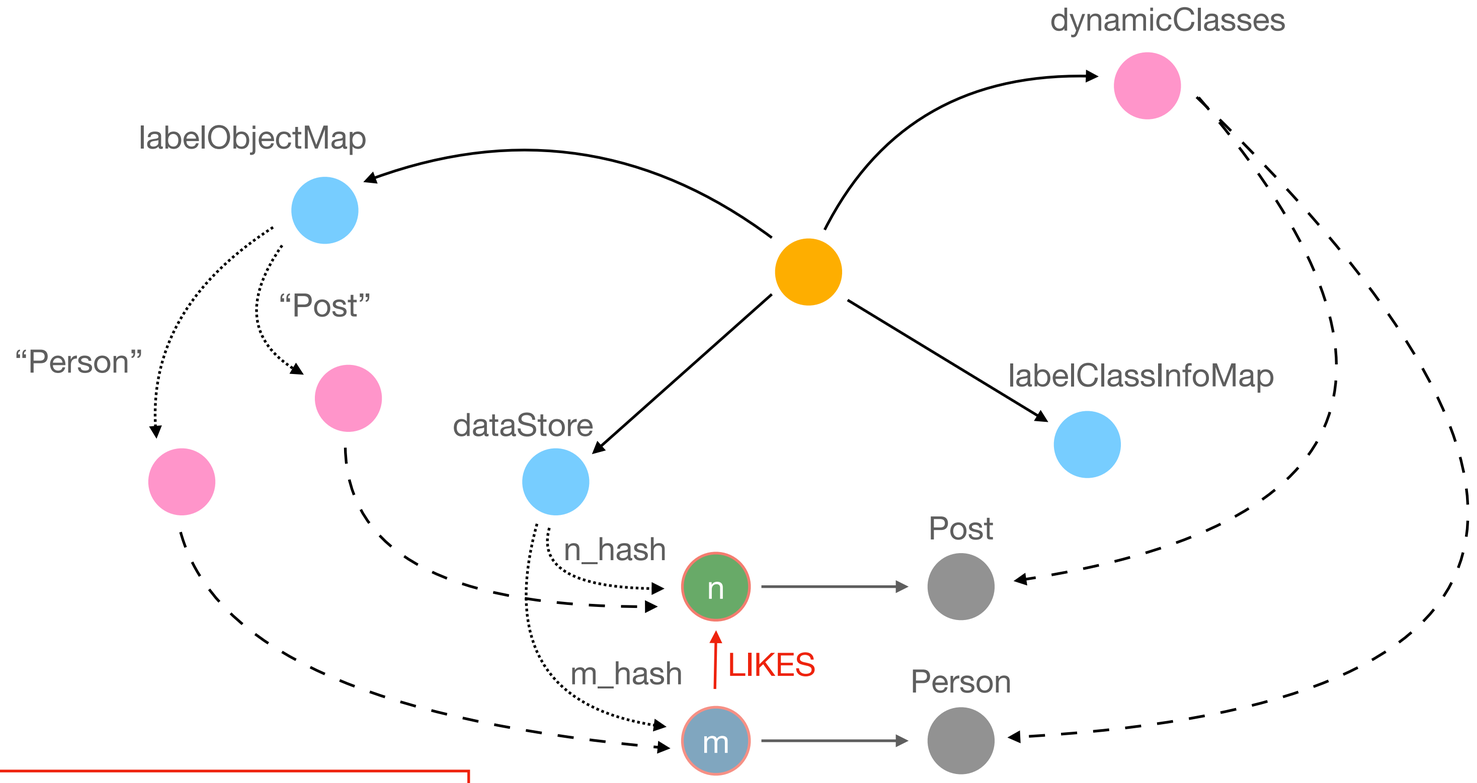



```
1| db.query(
2| "MATCH (m:`Person` {id: 112})"
3| +"-[r:LIKES]->"
4| +"(n:`Post` {id: 481}) "
5| +"DELETE r"
6| );
```

LEGEND

- Class instance
- ArrayList instance
- HashMap instance
- eStore instance
- Post instance
- Person instance
- instanceof relation
- Key
- HashMap instance
- List element relation

Example | Deleting edges



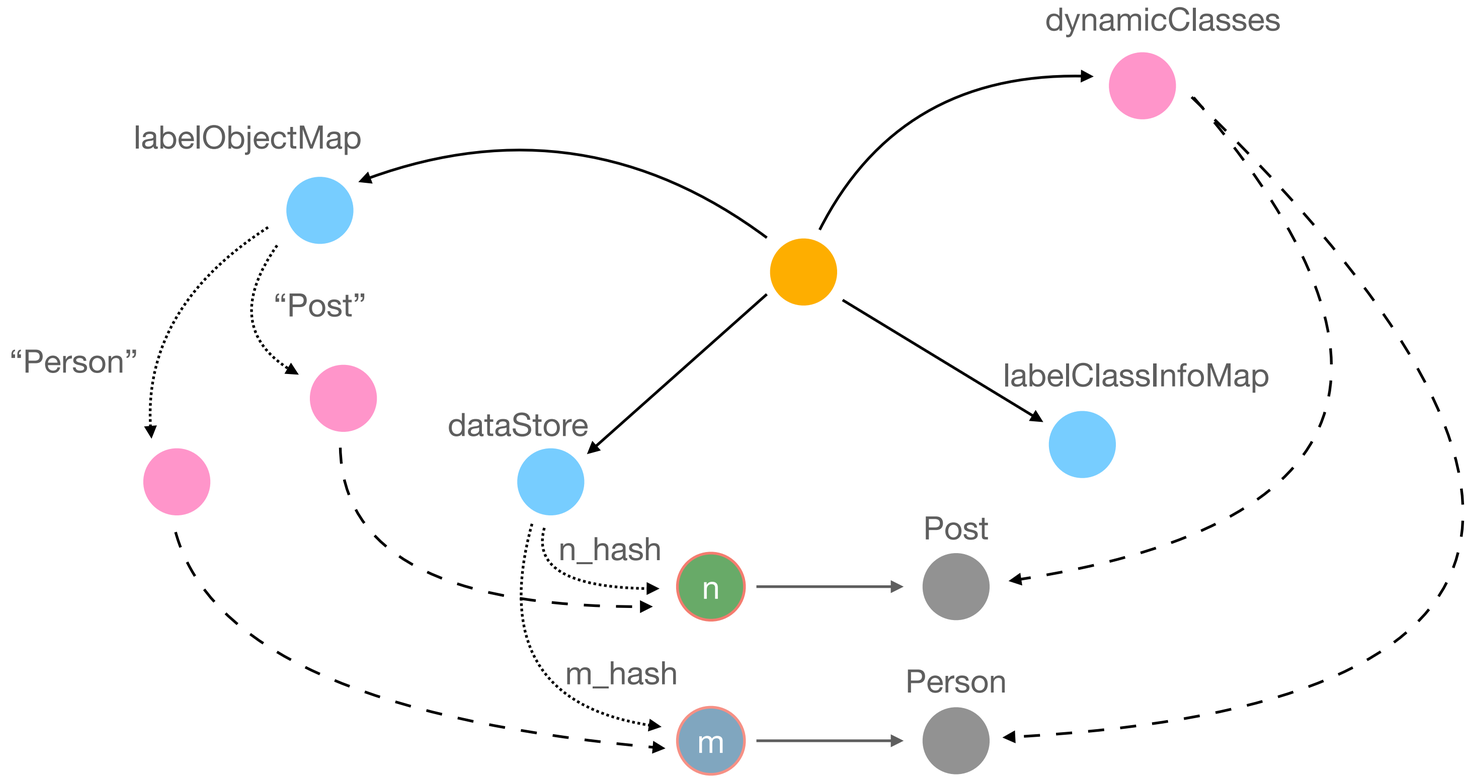
1. Match the pattern
   (m:`Person` …)-[r:LIKES]->(n:`Post` …)

```
1| db.query(
2| "MATCH (m:`Person` {id: 112})"
3| +"-[r:LIKES]->"
4| +"(n:`Post` {id: 481}) "
5| +"DELETE r"
6| );
```

LEGEND

- Class instance
- ArrayList instance
- HashMap instance
- eStore instance
- Post instance
- Person instance
- instanceof relation
- Key
- HashMap instance
- List element relation

Example | Deleting edges



1. Delete edge **LIKES** (r)

```
1| db.query(
2| "MATCH (m:`Person` {id: 112})"
3| +"-[r:LIKES]->"
4| +"(n:`Post` {id: 481}) "
5| +"DELETE r"
6| );
```

LEGEND

- Class instance
- ArrayList instance
- HashMap instance
- eStore instance
- Post instance
- Person instance
- instanceof relation
- Key
- HashMap instance
- List element relation

Results | LDBC SNB Benchmark

- **Benchmark setup**: We compare end-to-end and component latencies for Neo4j (server & in-memory) versus eStore on 9 LDBC SNB queries across scale factors 0.1–10.

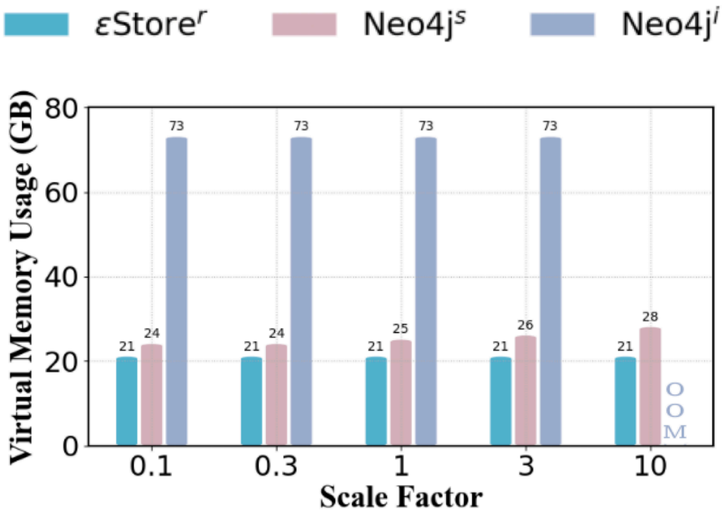
- eStore completes every run within the 63 GB cap and delivers 5–13 × lower latency than Neo4j, while Neo4j^{i} hits OOM on all scale factor 10 queries.

Query	SF	Neo4j^{s}					Neo4j^{i}					eStore^{r}				
		t_{Pa}	t_{Pl}	t_{Ex}	t_{Bd}	T_{tot}	t_{Pa}	t_{Pl}	t_{Ex}	t_{Bd}	T_{tot}	t_{Pa}	t_{Pl}	t_{Ex}	t_{Bd}	T_{tot}
Q^{A}_{SNB}	0.1	11	29	0	40	**46**	1	33	3	37	**38**	0	0	0	0	**0**
	0.3	11	28	0	39	**44**	1	33	5	39	**40**	0	1	1	2	**3**
	1	10	24	0	34	**38**	0	31	24	55	**55**	0	0	1	1	**1**
	3	12	24	0	36	**40**	2	32	40	75	**76**	0	1	4	4	**5**
	10	11	23	0	35	**39**			OOM			0	0	4	5	**5**
Q^{B}_{SNB}	0.1	10	24	0	34	**39**	0	32	2	35	**36**	0	0	0	0	**0**
	0.3	12	27	0	39	**43**	0	33	5	38	**39**	0	0	1	1	**2**
	1	11	26	0	37	**41**	1	34	9	44	**45**	0	0	0	1	**1**
	3	10	23	0	33	**37**	0	33	52	86	**87**	0	1	4	5	**6**
	10	11	24	0	35	**39**			OOM			0	0	4	4	**4**
Q^{C}_{SNB}	0.1	11	25	0	36	**41**	0	32	2	34	**35**	0	0	1	1	**1**
	0.3	10	27	0	37	**41**	0	32	4	37	**37**	0	0	3	3	**4**
	1	10	24	0	34	**38**	0	30	18	49	**50**	0	0	3	3	**3**
	3	10	24	0	34	**38**	0	31	21	52	**53**	0	0	8	8	**9**
	10	11	25	0	36	**41**			OOM			0	0	19	19	**19**
Q^{D}_{SNB}	0.1	16	30	0	46	**51**	1	34	0	36	**36**	0	0	0	0	**1**
	0.3	15	30	0	45	**50**	1	34	0	35	**36**	0	1	2	2	**4**
	1	13	25	0	38	**43**	0	34	0	34	**35**	0	0	1	1	**1**
	3	16	26	0	43	**47**	3	34	0	37	**38**	0	1	2	3	**4**
	10	15	26	0	42	**46**			OOM			0	0	4	4	**5**
Q^{E}_{SNB}	0.1	10	23	0	33	**38**	1	30	0	31	**31**	0	0	7	7	**7**
	0.3	11	21	0	32	**36**	2	34	0	36	**37**	0	0	29	29	**30**
	1	11	22	0	34	**38**	4	29	0	33	**33**	0	0	114	115	**115**
	3	11	20	0	31	**36**	6	29	0	36	**37**	0	1	319	320	**321**
	10	11	20	0	31	**35**			OOM			0	0	866	866	**867**
Q^{F}_{SNB}	0.1	10	15	29	54	**58**	0	24	44	67	**68**	0	0	5	6	**6**
	0.3	10	16	66	92	**97**	1	25	107	133	**134**	0	1	15	16	**16**
	1	9	16	212	237	**241**	1	24	286	311	**312**	0	0	35	35	**35**
	3	8	16	530	554	**559**	0	23	754	777	**778**	0	0	87	87	**88**
	10	11	18	1531	1559	**1564**			OOM			0	0	258	258	**259**
Q^{G}_{SNB}	0.1	9	16	31	57	**61**	2	26	59	88	**88**	0	0	6	6	**7**
	0.3	9	16	102	128	**133**	0	24	206	230	**231**	0	0	30	31	**31**
	1	10	15	423	448	**453**	1	24	728	753	**754**	0	0	74	75	**75**
	3	9	15	1298	1322	**1326**	3	25	2484	2511	**2512**	0	0	244	244	**245**
	10	10	15	4393	4418	**4422**			OOM			0	0	915	915	**915**
Q^{H}_{SNB}	0.1	10	16	3	29	**33**	0	24	7	31	**32**	0	0	1	1	**1**
	0.3	10	16	6	33	**38**	0	24	16	40	**41**	0	1	4	4	**5**
	1	9	16	18	44	**48**	0	23	40	63	**63**	0	0	3	3	**3**
	3	9	16	45	70	**74**	0	23	105	128	**129**	0	1	10	11	**11**
	10	10	16	112	137	**143**			OOM			0	0	22	22	**22**
Q^{I}_{SNB}	0.1	9	17	2	28	**32**	0	24	3	27	**27**	0	0	0	0	**0**
	0.3	10	16	3	29	**34**	0	24	6	30	**31**	0	1	1	2	**2**
	1	9	16	8	33	**38**	0	24	14	38	**39**	0	0	1	1	**1**
	3	10	16	14	39	**43**	0	24	33	57	**57**	0	1	4	5	**5**
	10	9	15	40	64	**69**			OOM			0	0	6	6	**7**

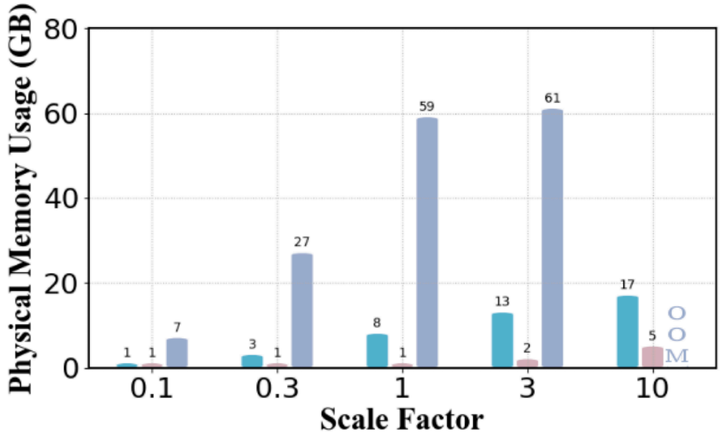
$t_{Pa} = Parse\ time, t_{Pl} = Planning\ time, t_{Ex} = Execution\ time, t_{Bd} = Breakdown\ time, t_{tot} = End - to - end\ time$

Results | LDBC SNB Benchmark

- **Physical memory efficiency**: Even at SF = 10, eStore peaks at 17 GB—nearly 4× lower than Neo4j^{i} (61 GB) and well within system limits.
- **Graceful scaling**: eStore's physical usage rises predictably (1 → 17 GB) with data size, avoiding sudden spikes.
- **Take-away**: eStore delivers in-memory performance while maintaining a far leaner and more predictable memory profile than Neo4j's in-memory configuration.



(a) SNB virtual memory usage in GB.



(b) SNB physical memory usage in GB.

Results | Reimplementing Methods

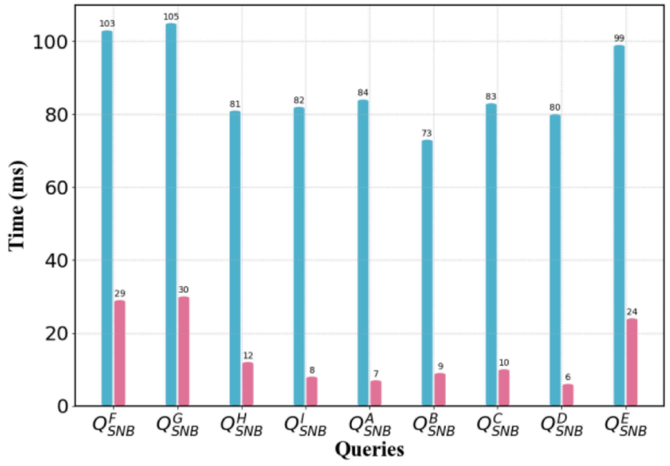
- **Benchmark setup**: measured end-to-end execution times for re-implementing data-structure methods in OGO vs. eStore.
- **Scalability**: OGO^{Neo} and OGO^{Mem} time out (> 1min) on larger workloads.
- **Performance**: eStore runs roughly 10× faster than OGO across all workloads.

Data-structure	#Elements	OGO^{Neo}	OGO^{Mem}	$\epsilon Store^{r}$	$\epsilon Store^{u}$
JCF v17.0					
ArrayList	10^2	2231	264	89	81
	10^3	8169	486	97	90
	10^4	40261	888	133	116
	10^5	TO	TO	166	162
ArrayDeque	10^2	1632	399	88	86
	10^3	3377	497	105	89
	10^4	41301	1540	128	120
	10^5	TO	TO	156	166
HashMap	10^2	2391	315	78	77
	10^3	9479	516	88	100
	10^4	TO	2753	136	126
	10^5	TO	TO	356	394
LinkedList	10^2	24831	420	85	79
	10^3	TO	TO	83	88
	10^4	TO	TO	125	133
	10^5	TO	TO	232	216
Guava v32.1.3-jre					
ArrayTable	10^2	1711	358	82	76
	10^3	2742	415	94	90
	10^4	23298	1031	125	140
	10^5	TO	TO	237	228
Eclipse v11.1.0					
UnifiedSet	10^2	1592	514	78	76
	10^3	2916	595	87	90
	10^4	55464	3373	158	149
	10^5	TO	TO	223	239
UnifiedMap	10^2	1997	489	88	83
	10^3	5079	865	102	109
	10^4	TO	TO	111	113
	10^5	TO	TO	210	200
FastList	10^2	1637	453	80	77
	10^3	2750	540	89	87
	10^4	40243	1255	130	143
	10^5	TO	TO	172	161
ArrayStack	10^2	1820	411	78	82
	10^3	3538	741	87	84
	10^4	TO	2461	127	129
	10^5	TO	TO	153	153

Results | Compile-time Code Generation

- **Benchmark setup**: LDBC SNB end-to-end times for eStore with vs. without compile-time rewriting.
- **Sequential run** (queries executed back-to-back): compile-time rewriting delivers a substantial speed-up on the first query.
- **Standalone run** (each query executed in isolation): compile-time rewriting accelerates every query.

(a) Sequence.



(b) Standalone.

Conclusions

- **Epsilon storage model**: queries live objects with virtually no abstraction overhead.
- **eStore**: graph-based object store that blends declarative queries with imperative code.
- **Compile-time query rewriting**: cuts runtime latency by an order of magnitude.
- **Benchmark results**: eStore surpasses production graph databases on LDBC SNB, making it a strong testing-environment substitute.
- **Robustness & versatility**: proven on LDBC SNB and by re-implementing methods in large-scale open-source projects.
- **Plus more**: comprehensive API design and formal foundations detailed in the paper.

Contact : auditt@utexas.edu

Supplementary