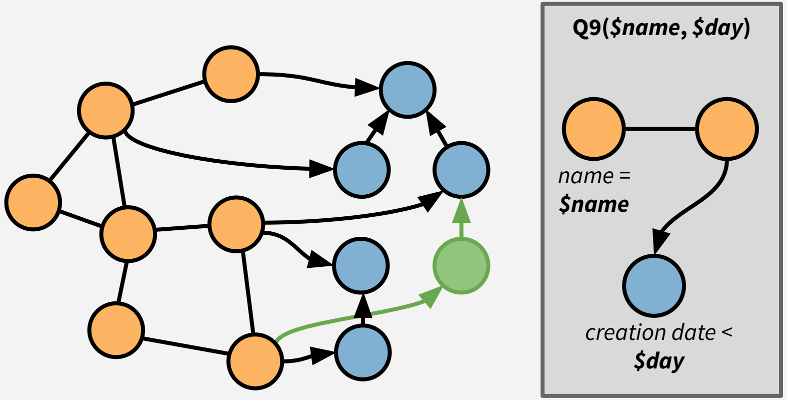
The LDBC Social Network Benchmark Interactive workload v2:

A transactional graph query benchmark with deep delete operations

David Püroja, Jack Waudby, Peter Boncz, **Gábor Szárnyas**

TPCTC | 2023-08-28 | Vancouver

SNB Interactive v1 (2015)

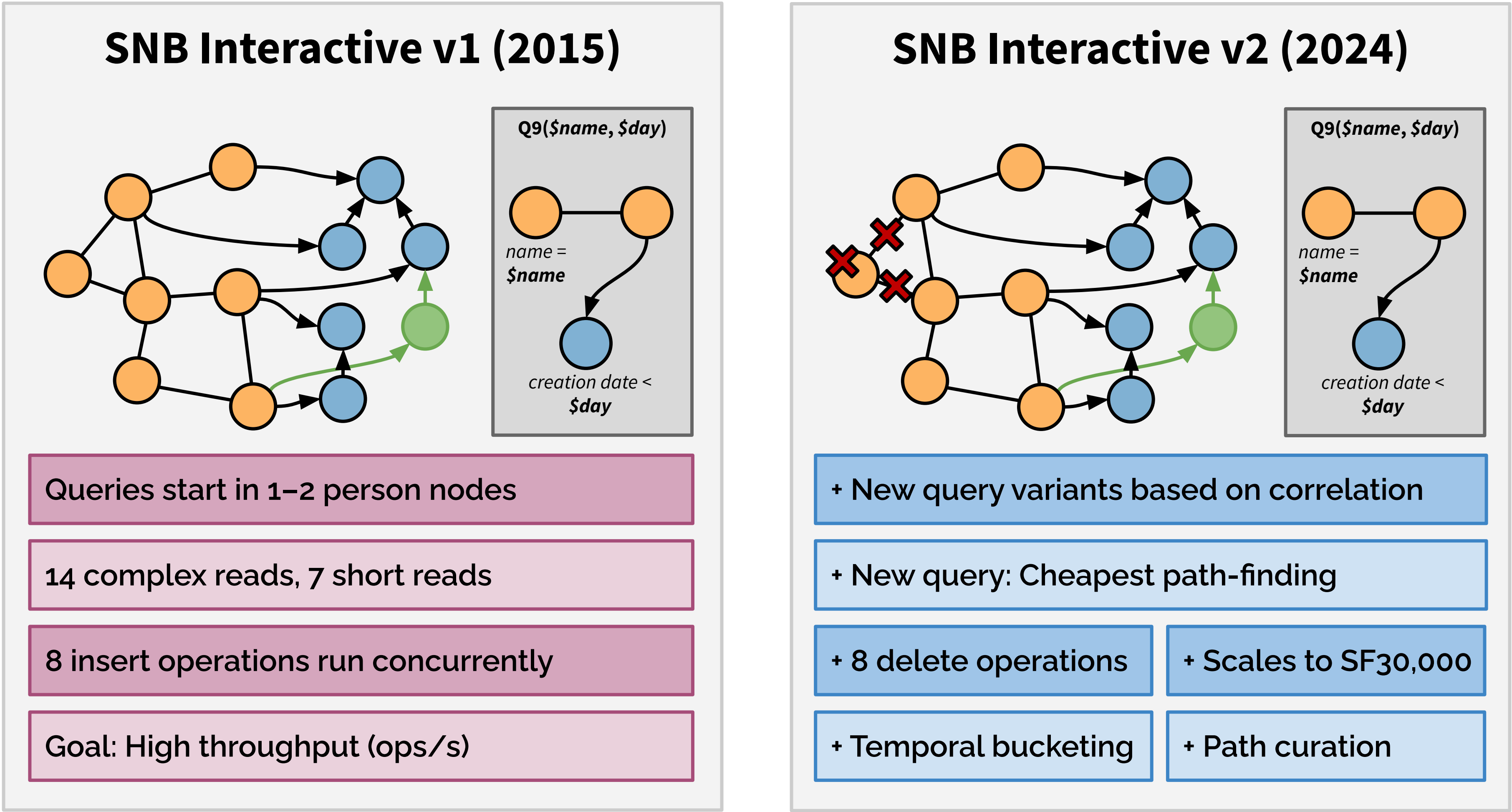


Queries start in 1–2 person nodes

14 complex reads, 7 short reads

8 insert operations run concurrently

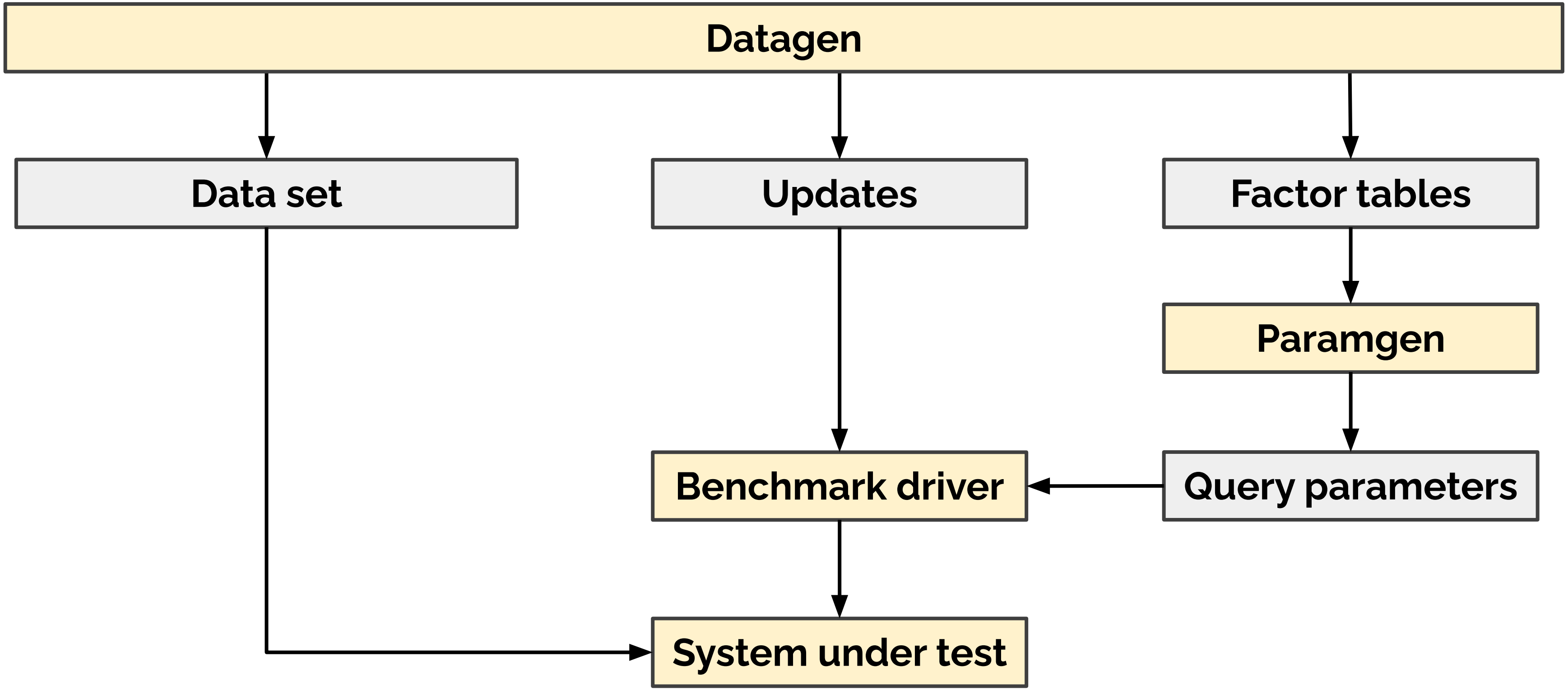
Goal: High throughput (ops/s)



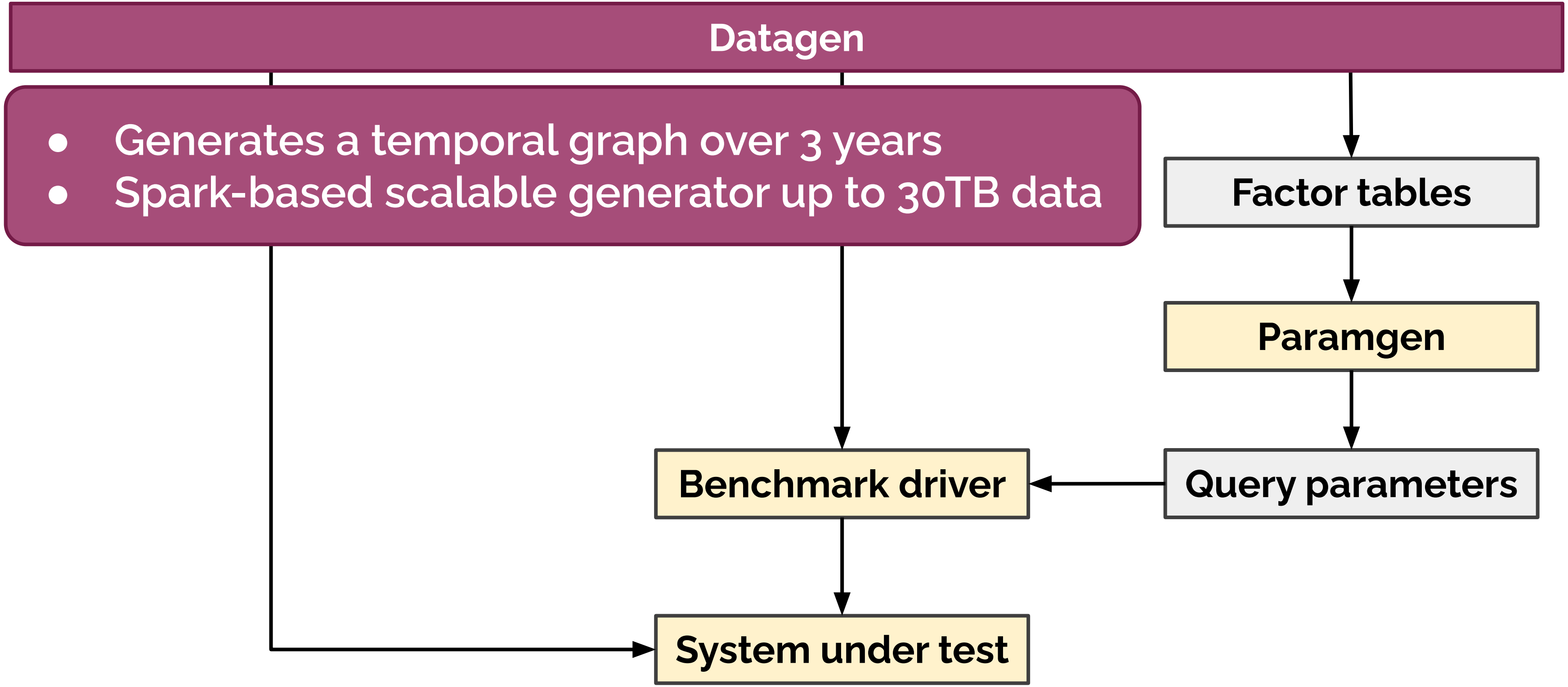
SNB Interactive v1 (2015)
Q9($name, $day)
name = $name
creation date < $day
Queries start in 1–2 person nodes
14 complex reads, 7 short reads
8 insert operations run concurrently
Goal: High throughput (ops/s)
SNB Interactive v2 (2024)
Q9($name, $day)
name = $name
creation date < $day
+ New query variants based on correlation
+ New query: Cheapest path-finding
+ 8 delete operations
+ Scales to SF30,000
+ Temporal bucketing
+ Path curation

Benchmark framework

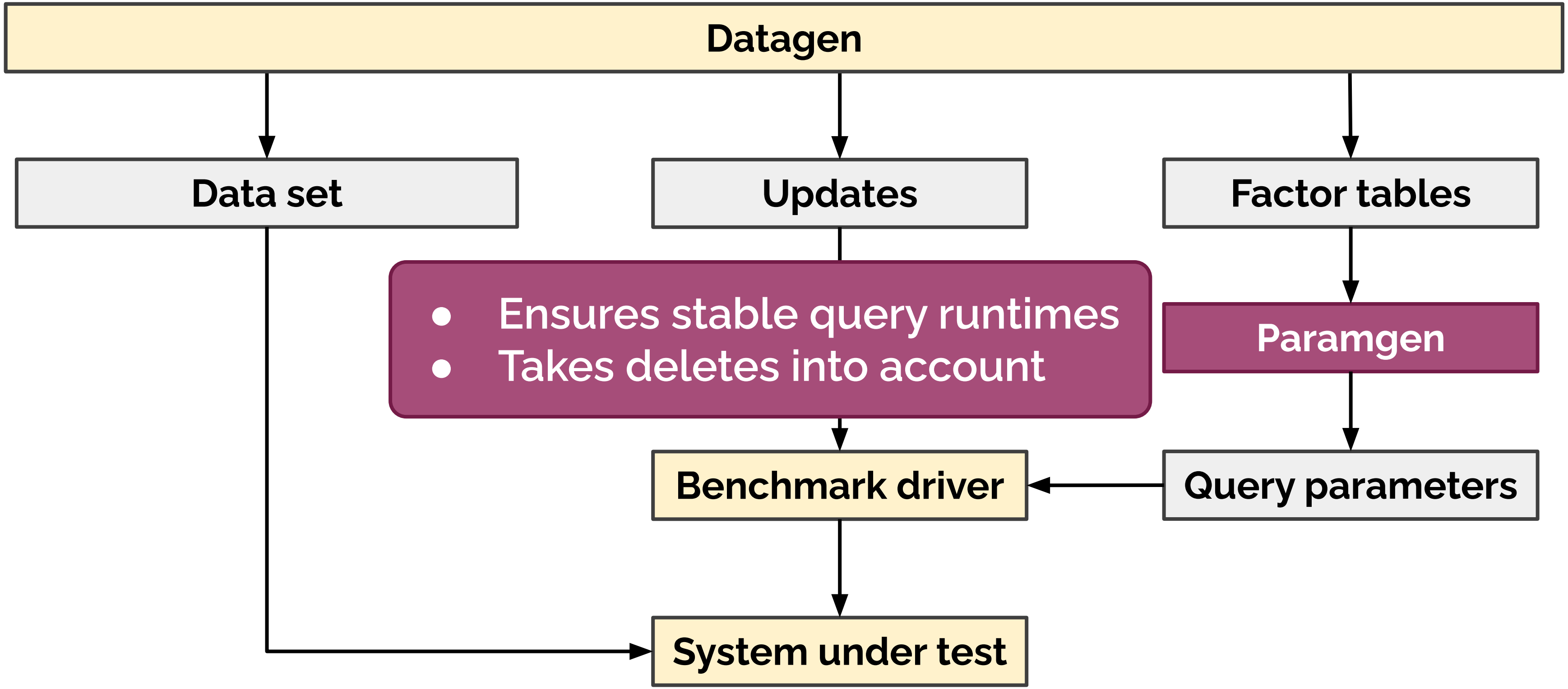
Benchmark workflow



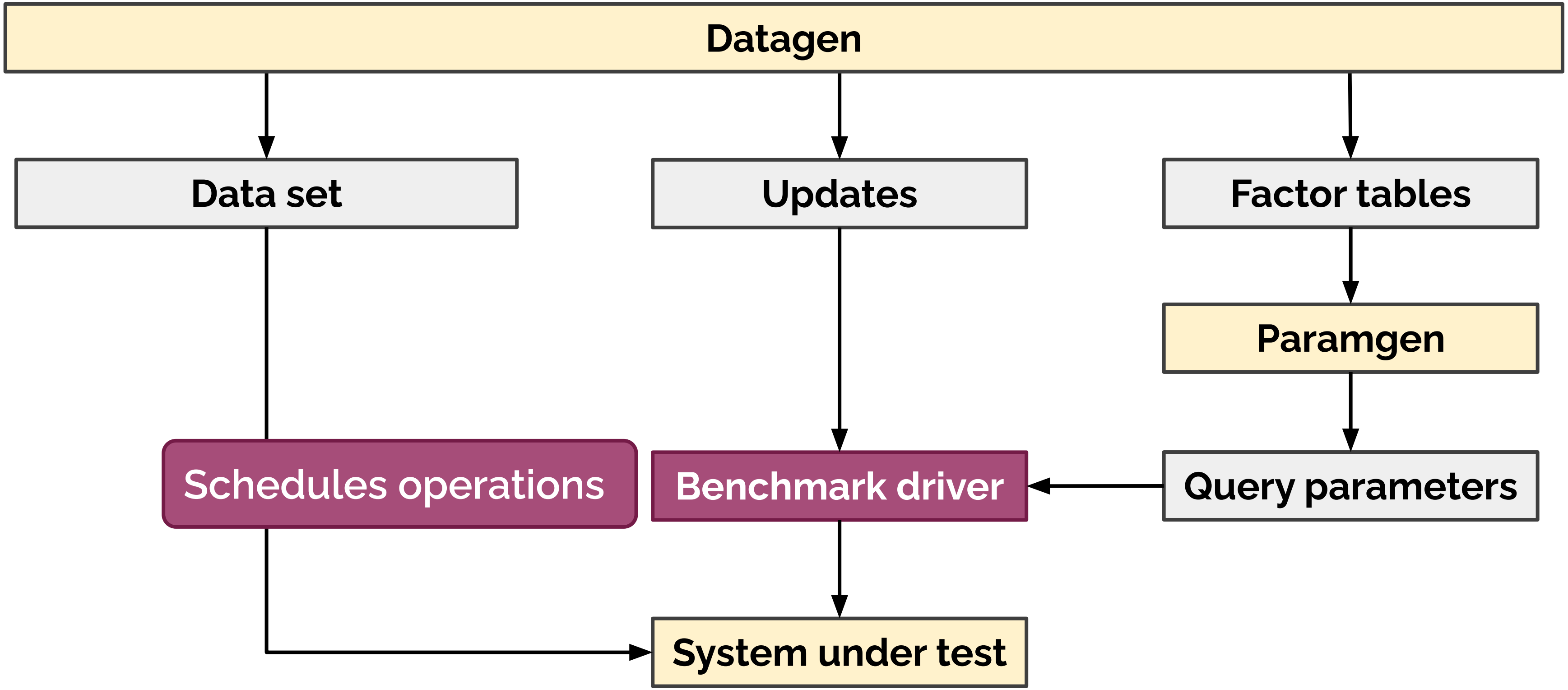
Benchmark workflow



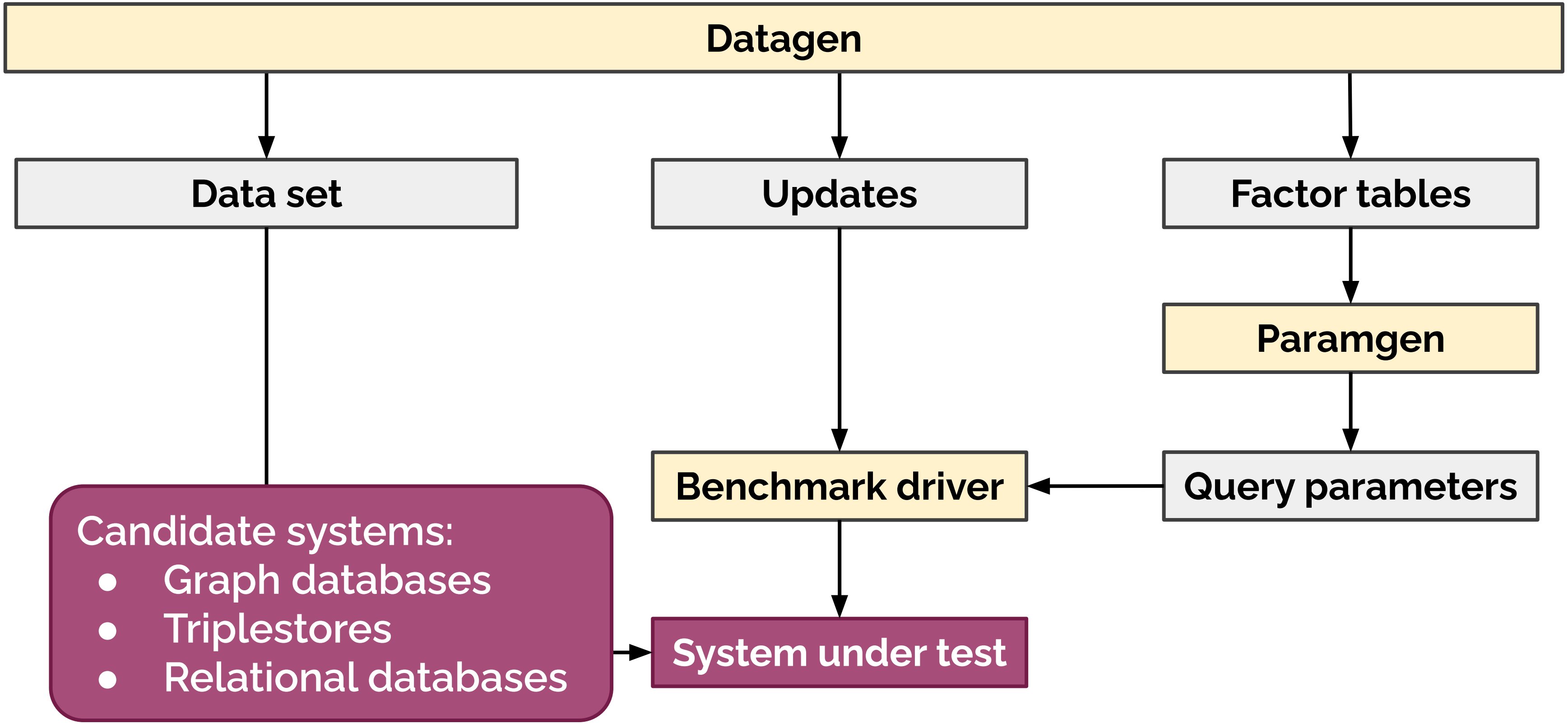
Benchmark workflow



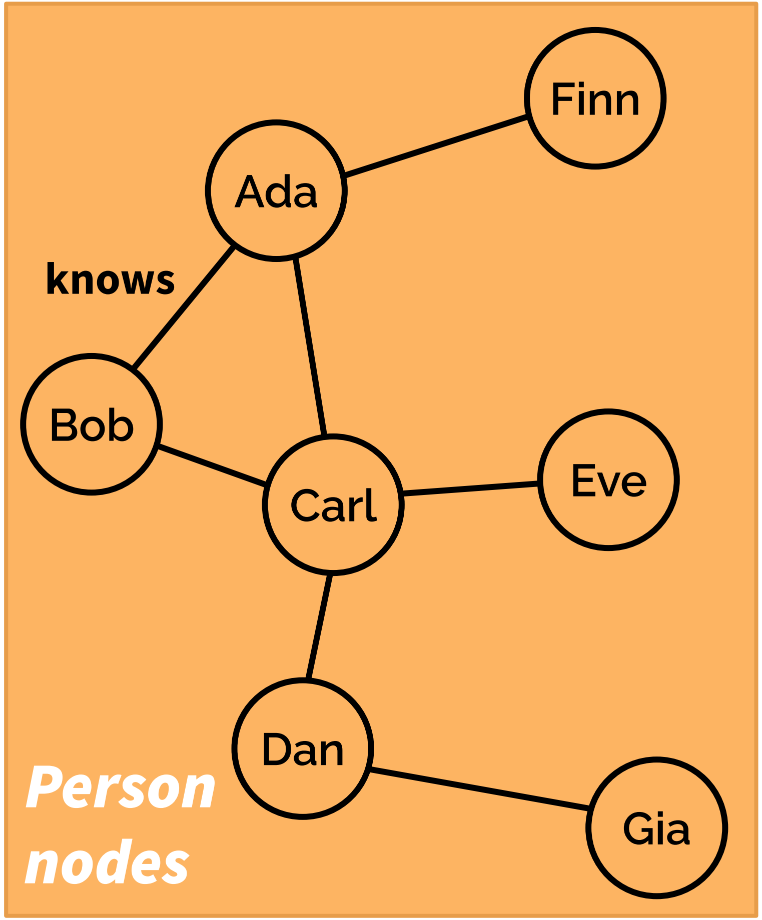
Benchmark workflow



Benchmark workflow



Data generator: Highlights

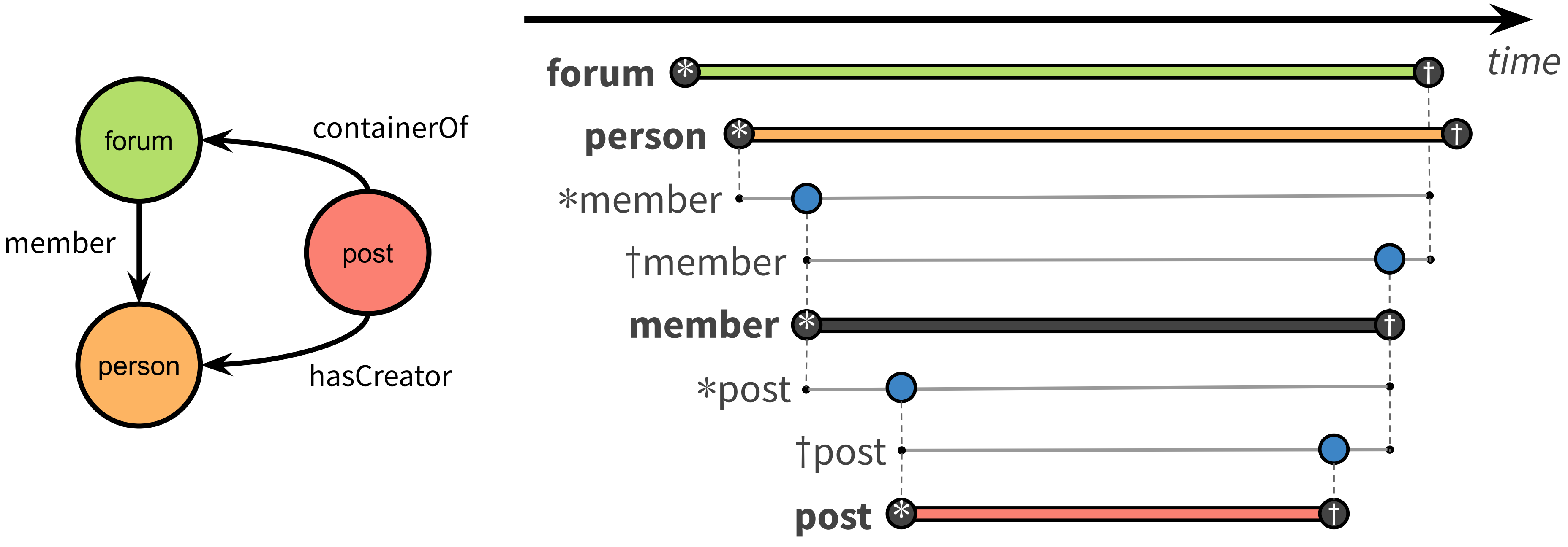


Person–knows–Person

- Degree distribution: Ugander et al. *"The Anatomy of the Facebook Social Graph"* (2011)
- Edges are added along 3 dimensions: university attendance, interests, random
- Deletes are implemented according to Lőrincz et al. *"Collapse of an online social network: Burning social capital to create it?"* (2019)

Generating deletes along dependencies: Lifespan management

The generator generates the entire temporal with creation dates * and deletion dates †



Factor table generation

Example: #comments for friends of friends

- numFoaFComments(p1, cnt) = count(knows(p1, p2) $\bowtie$ knows(p2, p3) $\bowtie$ hasCreator(p3, c))
  filter for unique values of p1, p2, p3

Joining three large tables would be very expensive, so we approximate it:

1. numFriendComments(p2, cnt) = count(knows(p2, p3) $\bowtie$ hasCreator(p3, c))
2. numFoaFComments(p1, cnt) = sum(knows(p1, p2) $\bowtie$ numFriendComments(p2, cnt))
   filtering is omitted

Operations

Workload mix

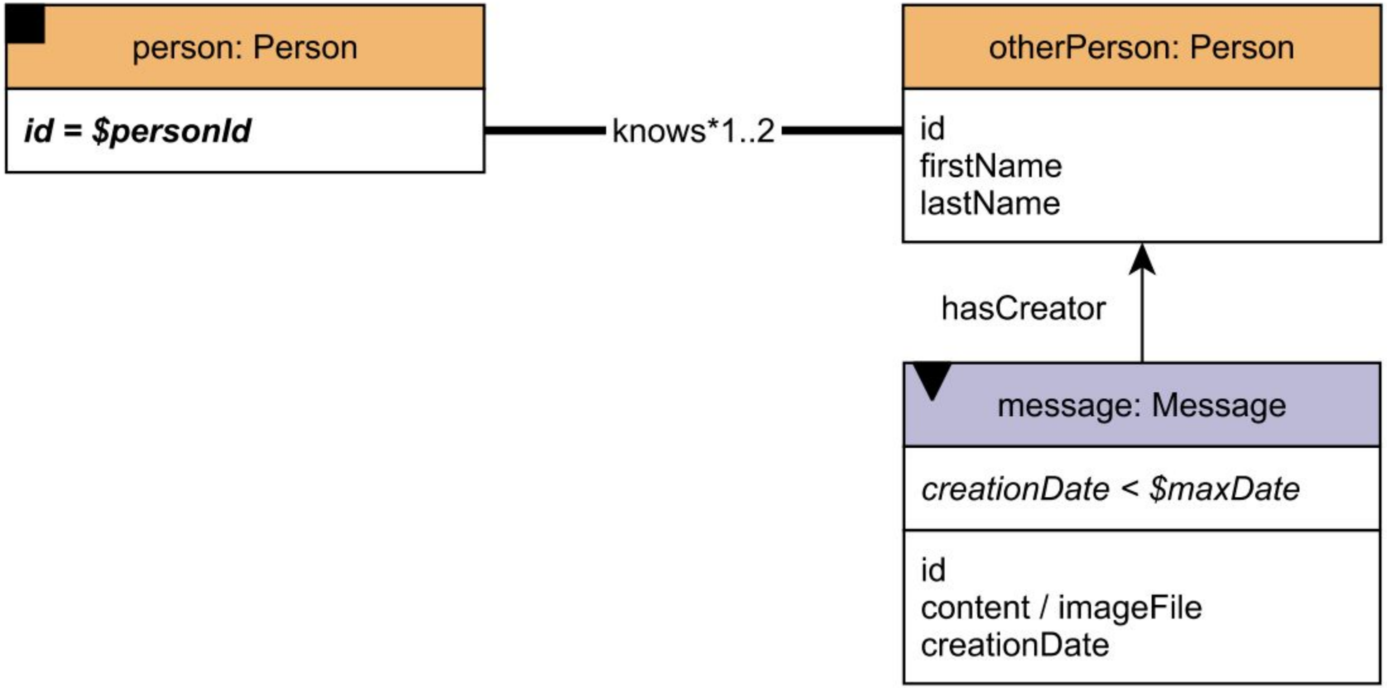
CR
complex reads
8%, 1–500 ms

SR
short reads
72%, 0.1–75 ms

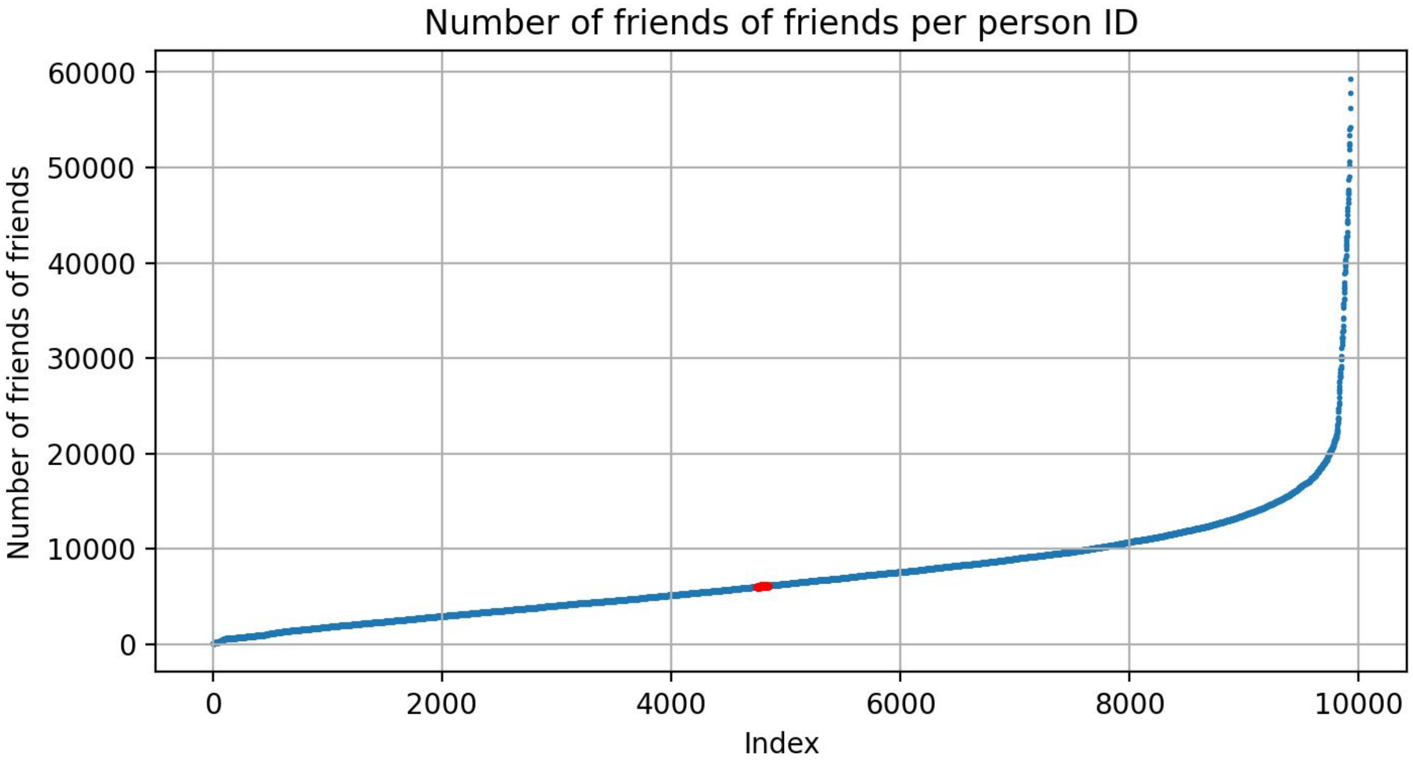
INS
inserts
20%, 0.1–100 ms

DEL
deletes
0.2%, ?? ms

Complex read Q9: Recent messages by F/FoaF

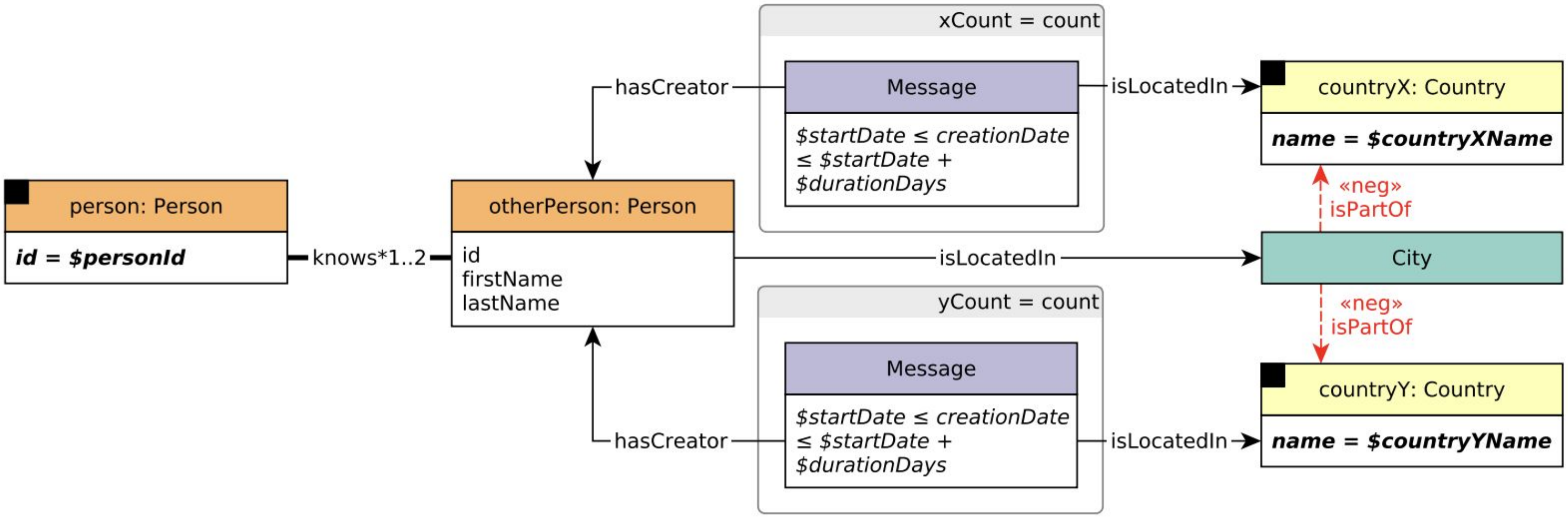


Q9 parameter selection: Window

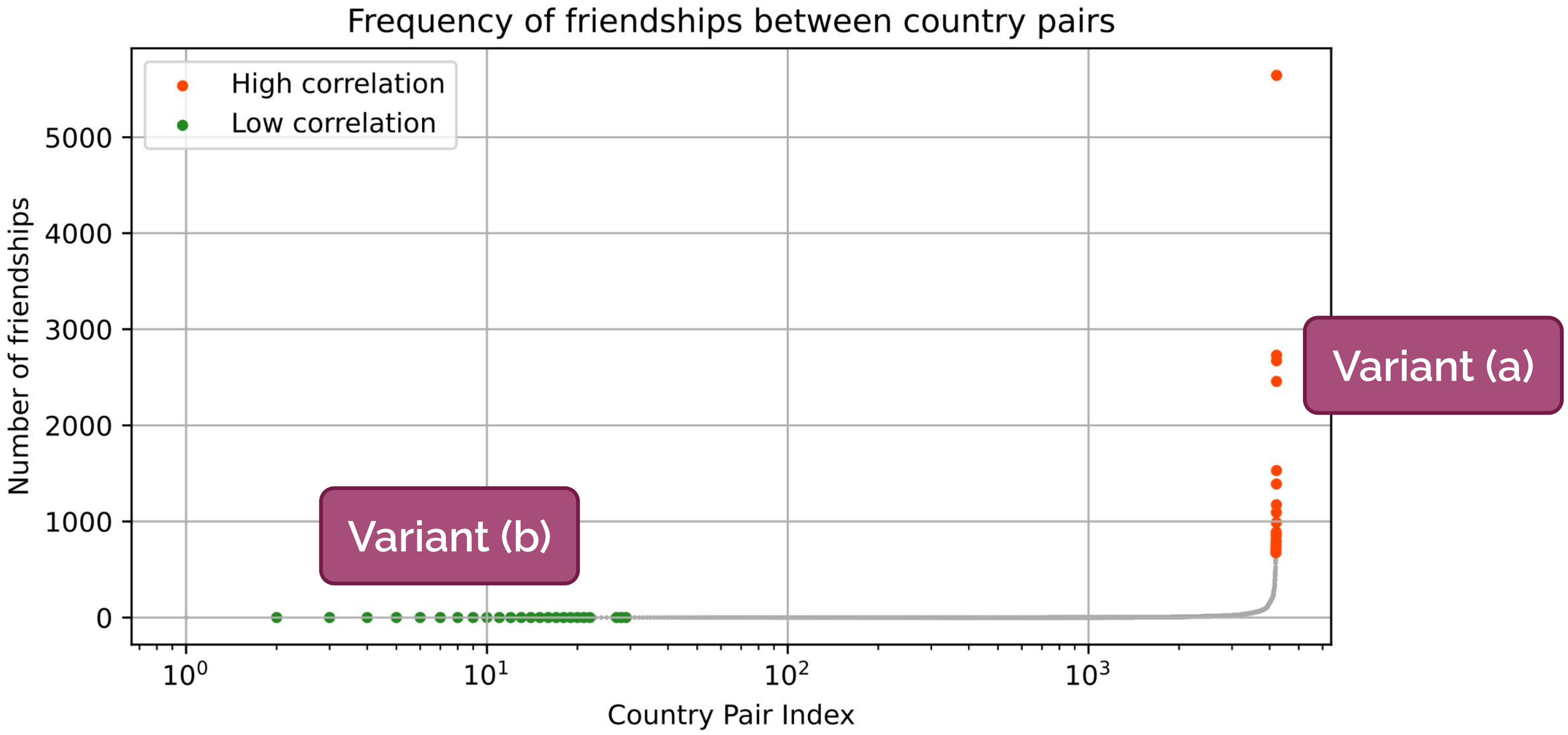


Complex read Q3: Travelling abroad

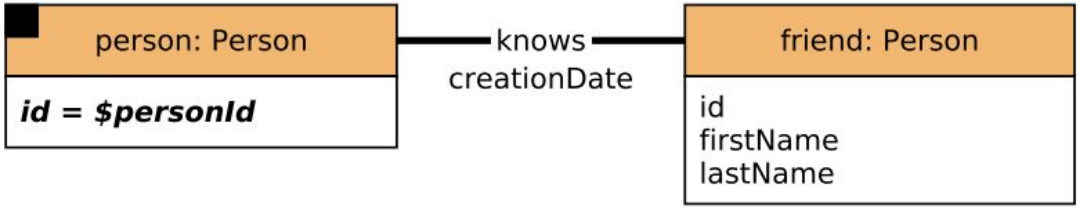
Friends and FoaFs that created Messages from given Countries but do not live there



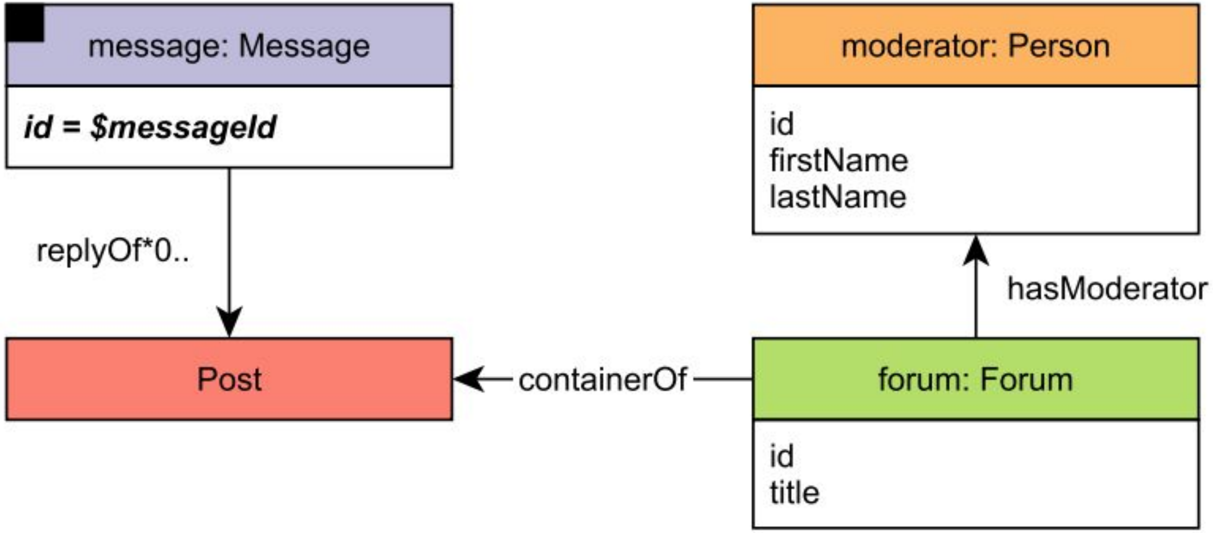
Complex read Q3: Travelling abroad



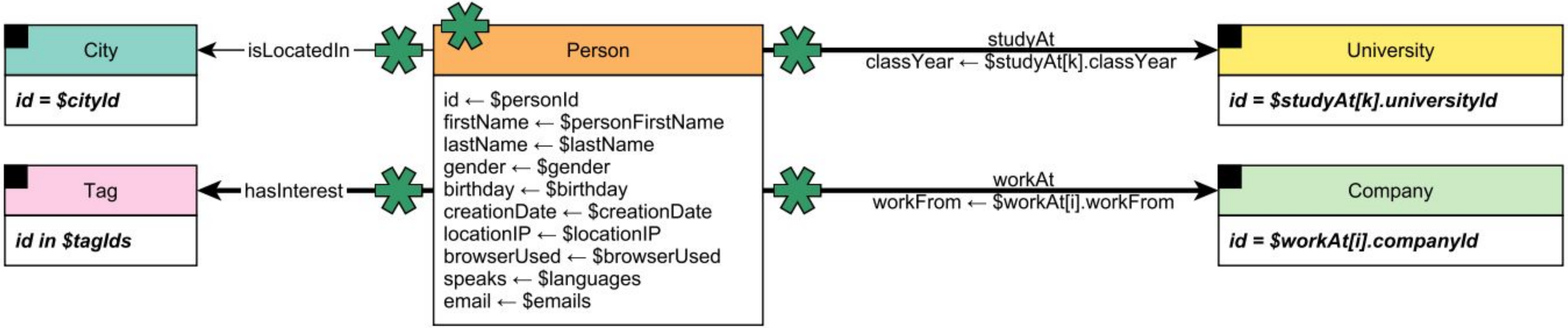
Short read Q3: Friends of a Person



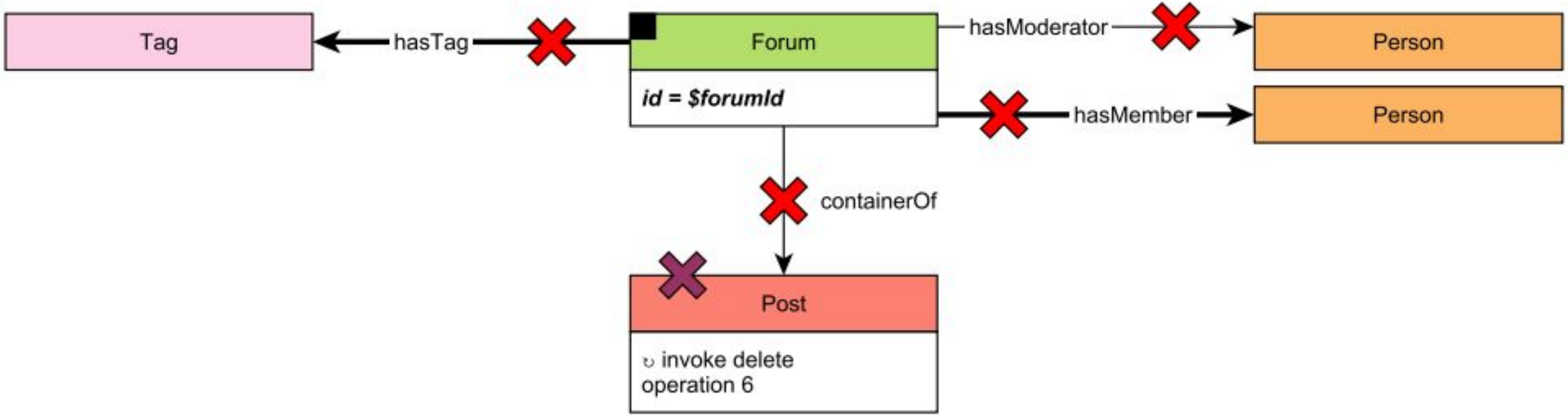
Short read Q6: Forum of a Message



Insert query INS1: Add Person

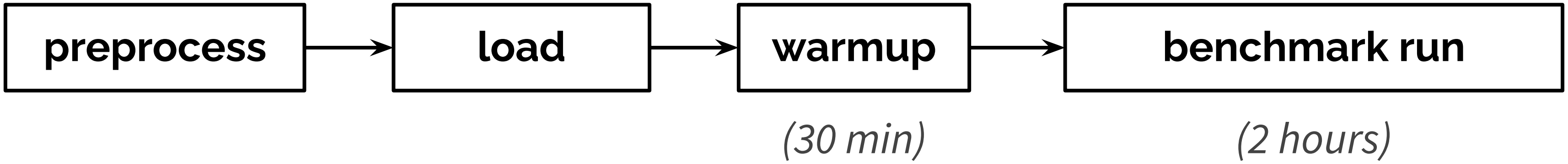


Delete query DEL4: Remove Forum



Scheduling

Benchmark execution



- Collect individual query runtimes
- Check 95% on-time requirement

Driver execution modes

The driver has 3 modes of operation, all start with the initial data set loaded.

1-2) Generate validation data set, Validate implementation

- single-threaded
- deterministic

3) Run benchmark

- multi-threaded
- calculates throughput
- pass/fail schedule

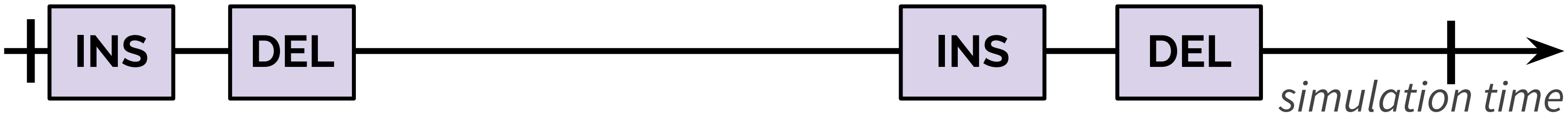
Scheduling operations: Theory

- **Updates:** replayed as they happen in the social network
- **Complex reads:** a given complex read query is scheduled for X update operations
- For each complex read instance, a sequence of **short reads** is triggered, short reads can trigger other short reads

	IS 1	IS 2	IS 3	IS 4	IS 5	IS 6	IS 7
IC 1	⊗	⊗	⊗				
IC 2	⊗	⊗	⊗	⊗	⊗	⊗	⊗
IC 3	⊗	⊗	⊗				
IC 7	⊗	⊗	⊗	⊗	⊗	⊗	⊗
IC 8	⊗	⊗	⊗	⊗	⊗	⊗	⊗
IC 9	⊗	⊗	⊗	⊗	⊗	⊗	⊗
IC 10	⊗	⊗	⊗				
IC 11	⊗	⊗	⊗				
IC 12	⊗	⊗	⊗				
IC 14	⊗	⊗	⊗				
IS 2	⊗	⊗	⊗	⊗	⊗	⊗	⊗
IS 3	⊗	⊗	⊗				
IS 5	⊗	⊗	⊗				
IS 6	⊗	⊗	⊗				
IS 7	⊗	⊗	⊗	⊗	⊗	⊗	⊗

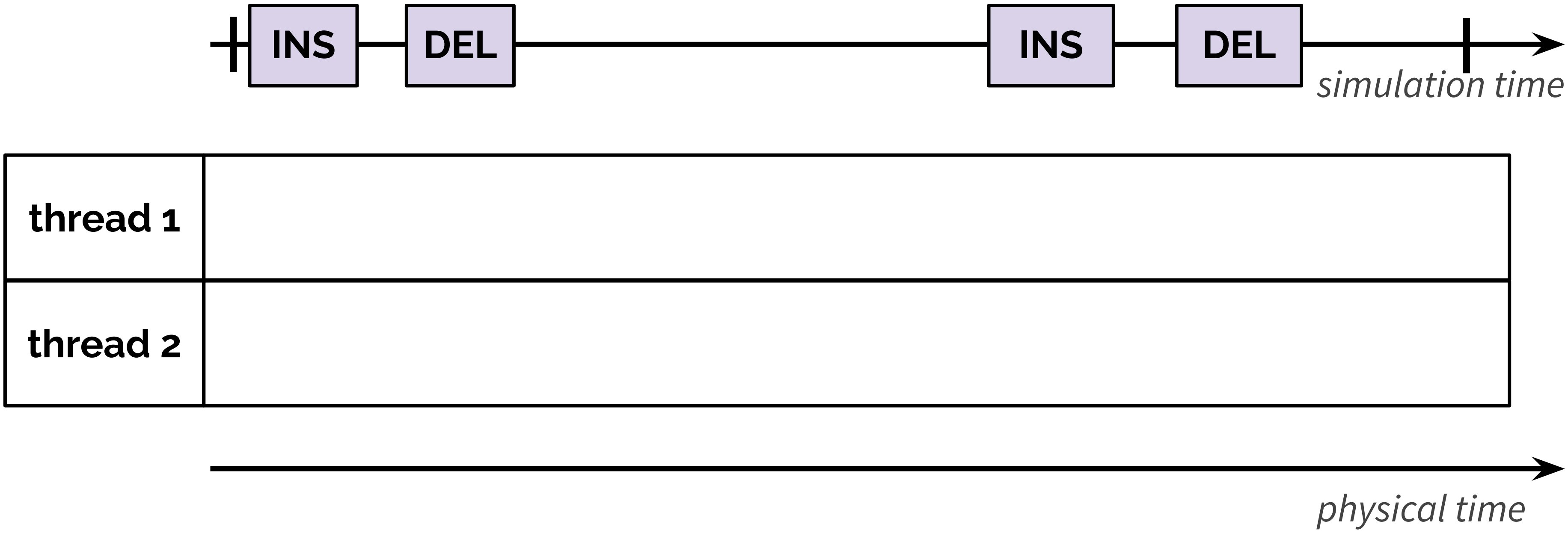
Scheduling operations: Example

Replay speed is determined by the TCR (total compression ratio)



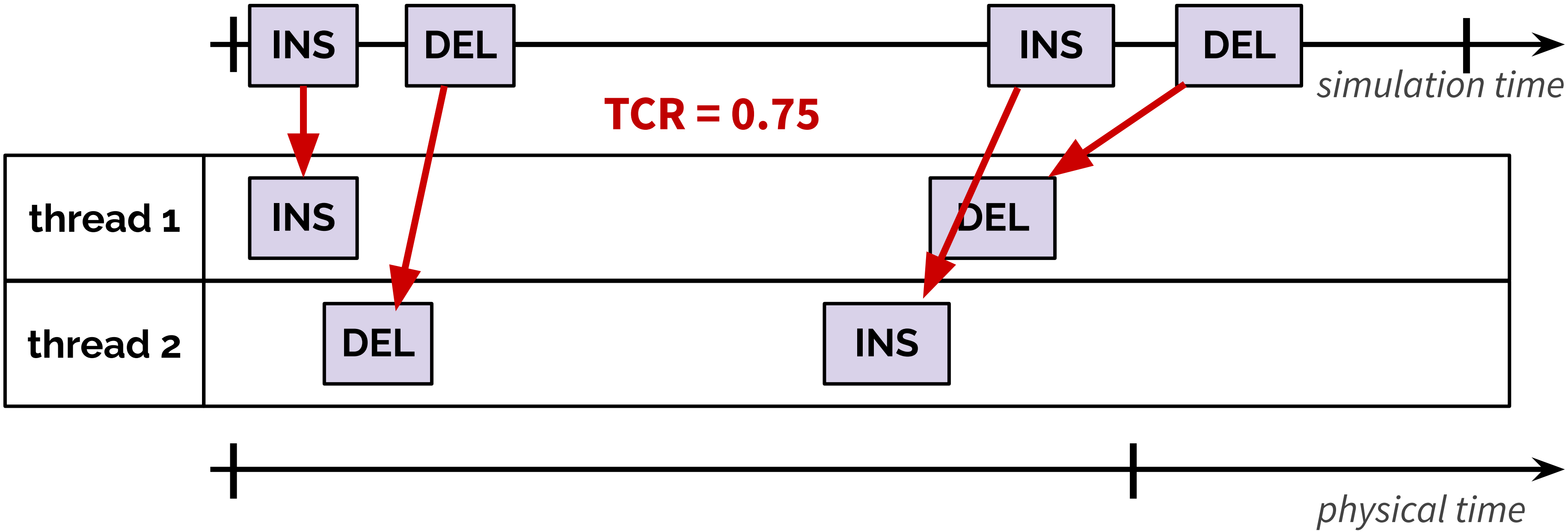
Scheduling operations: Example

Replay speed is determined by the TCR (total compression ratio)



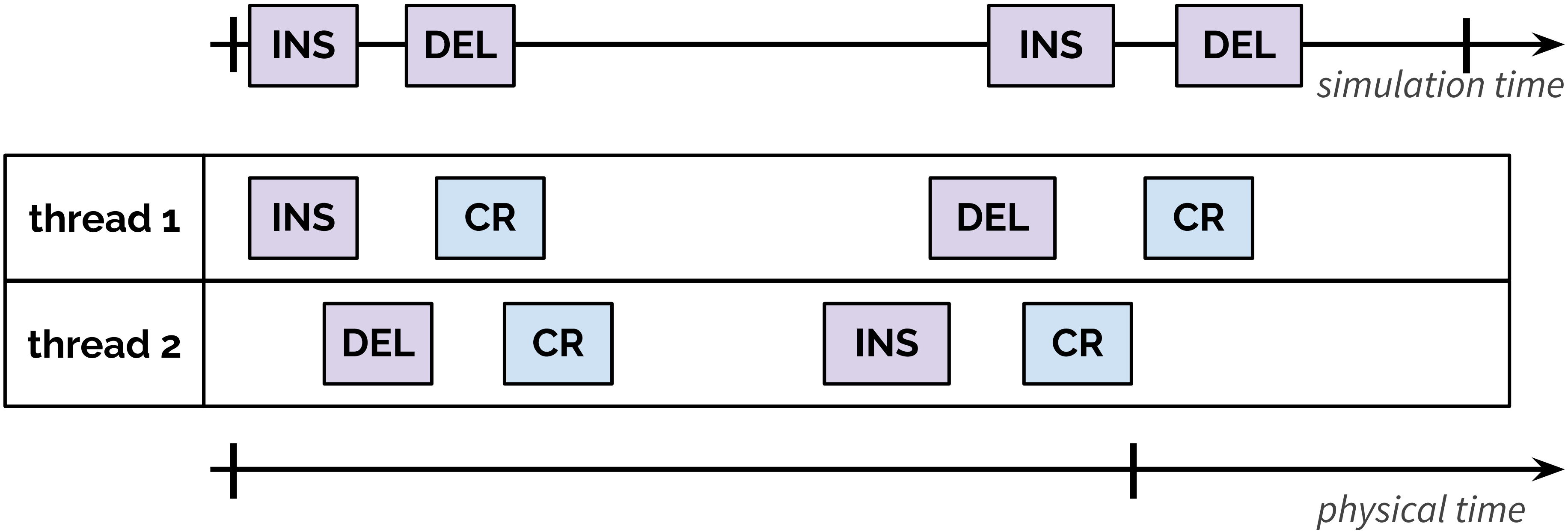
Scheduling operations: Example

Replay speed is determined by the TCR (total compression ratio)



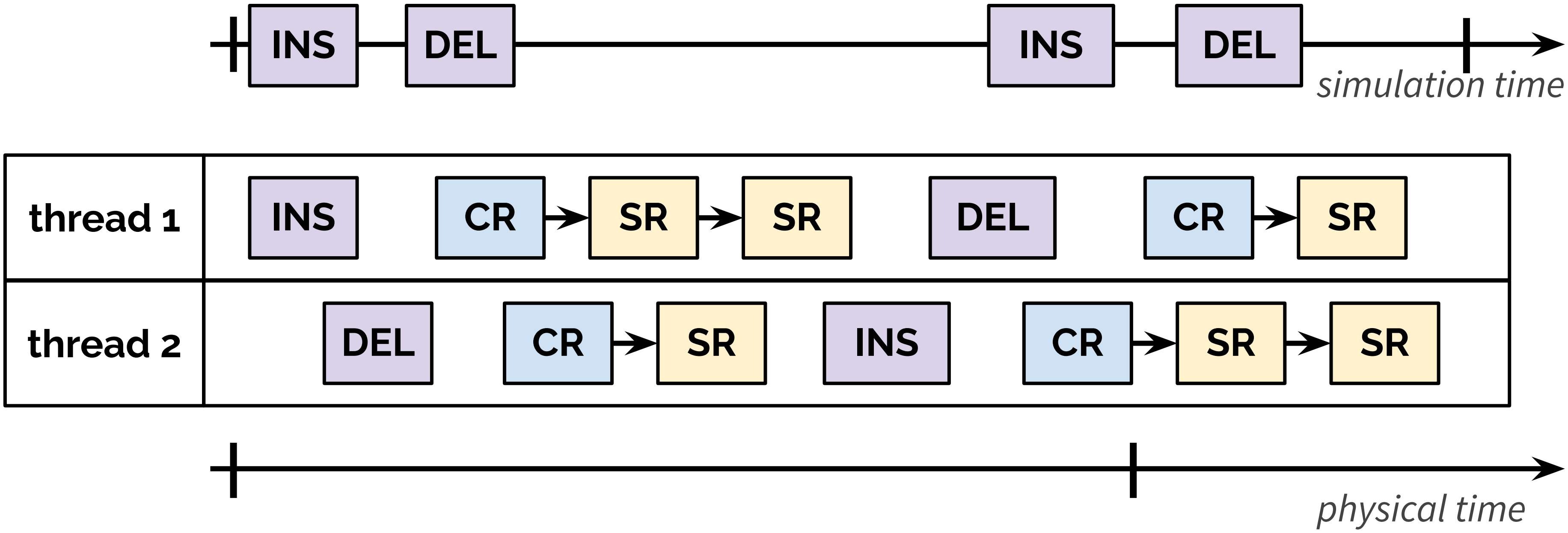
Scheduling operations: Example

Replay speed is determined by the TCR (total compression ratio)



Scheduling operations: Example

Replay speed is determined by the TCR (total compression ratio)

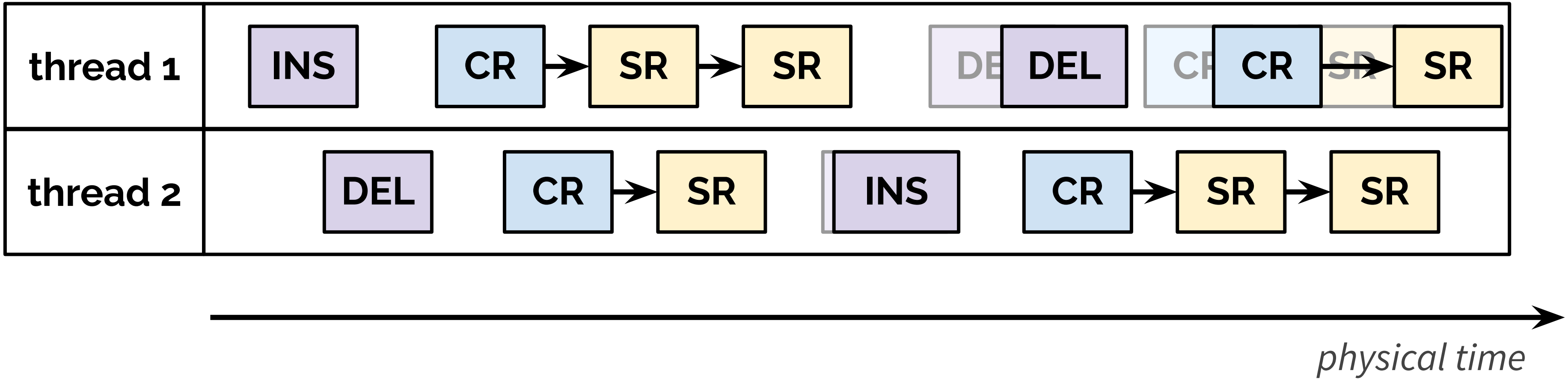


95% on-time requirement

In order to pass an audit, 95% of the executed queries must meet the following condition:

$$\textit{actual start time} - \textit{scheduled start time} < 1 \textit{ second}$$

If a run falls behind, it is no longer valid.



Scalability

Scaling up to SF30,000

Migrated from the Hadoop-based data generator to the Spark-based one

Scaling to large SFs gets super-exponentially more difficult

- more expensive: compute/storage costs, egress
- longer execution and transfer times
- things start to break more and more often
    - tools cannot load/process
    - connections drop
    - AWS disks corrupt
    - EMR jobs hang
    - availability zone out of instances
    - running out of disk/temp space
    - files get lost silently during transfer

Cheapest path-finding

Cheapest path query

“Cheapest path” = weighted shortest path (Dijkstra, Bellman–Ford)

Syntax in GQL and SQL/PGQ:

```
MATCH ANY CHEAPEST PATH p=
  (a:Person WHERE a.name='Bob')
  -[k:knows COST 1/k.interactionScore]->*
  (b:Person WHERE b.name='Eve')
```

The `ANY CHEAPEST PATH` clause is denoted as a *language opportunity*.

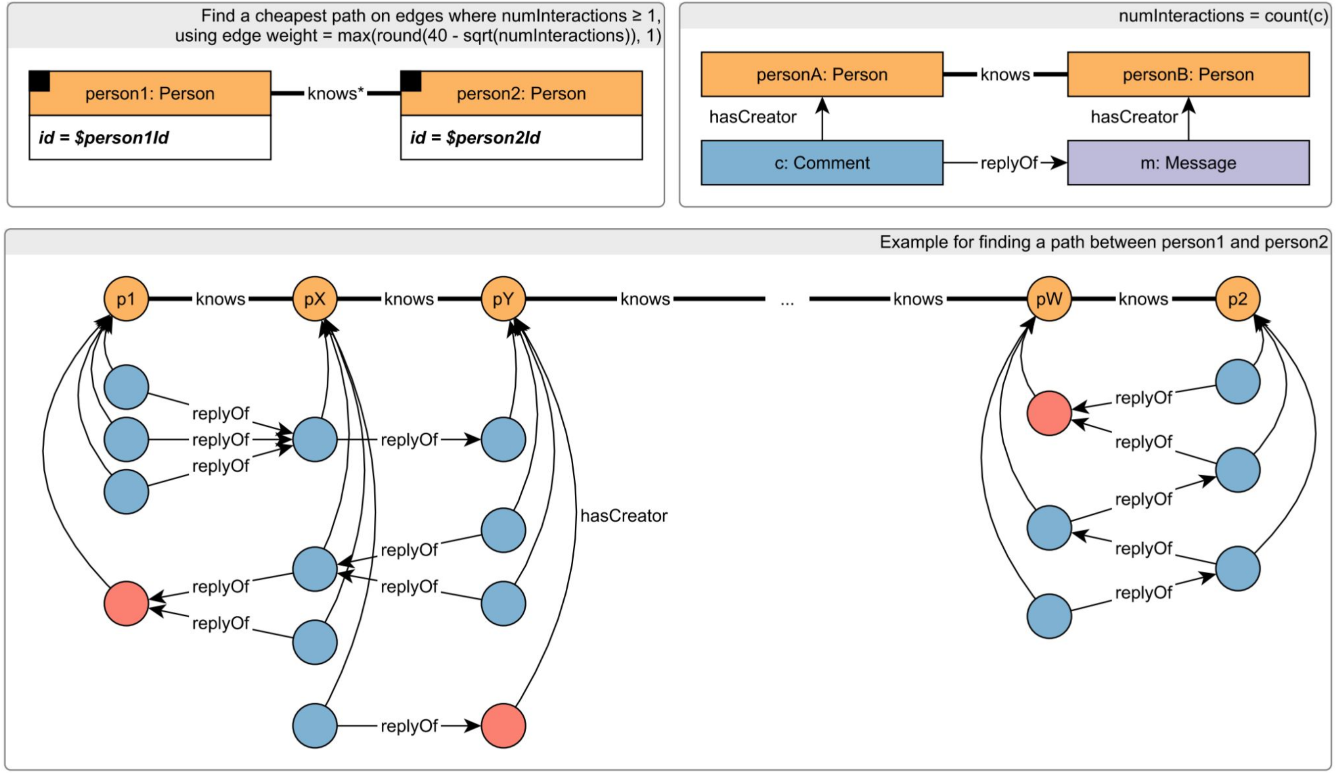
Cheapest path query

Difficult to express in SQL:1999 – long and cumbersome query, slow execution

But an important computational kernel: included in Interactive v2

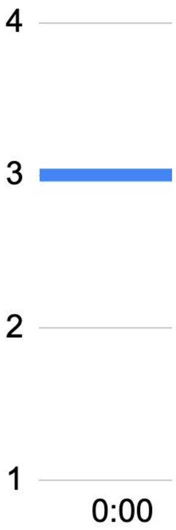
```
with recursive pathb(a, b, w) AS ( SELECT least(c.creatorpersonid, p.creatorpersonid) AS a, greatest(c.creatorpersonid,
p.creatorpersonid) AS b, greatest(round(40 - sqrt(count(*)::bigint, 1)  AS w FROM message c, message p WHERE
c.parentmessageid = p.id AND EXISTS (SELECT * FROM person_knows_person WHERE person1id = c.creatorpersonid AND person2id =
p.creatorpersonid) group by a, b), path(src, dst, w) AS ( SELECT a, b, w FROM pathb union all SELECT b, a, w FROM pathb ),
shorts(dir, gsrc, dst, prev, w, dead, iter) AS ( SELECT sdir, sgsrc, sdst, sdst, sw, sdead, siter FROM (VALUES (false,
:person1Id::bigint, :person1Id::bigint, 0::bigint, false, 0), (true, :person2Id::bigint, :person2Id::bigint, 0::bigint,
false, 0)) t(sdir, sgsrc, sdst, sw, sdead, siter) union all ( with ss AS (SELECT * FROM shorts), toExplore AS (SELECT *
FROM ss WHERE dead = false order by w limit 1000), newPoints(dir, gsrc, dst, prev, w, dead) AS (  SELECT e.dir, e.gsrc AS
gsrc, p.dst AS dst, p.src as prev, e.w + p.w AS w, false AS dead  FROM path p join toExplore e on (e.dst = p.src)  UNION
ALL  SELECT dir, gsrc, dst, prev, w, dead OR EXISTS (SELECT * FROM toExplore e WHERE e.dir = o.dir AND e.gsrc = o.gsrc AND
e.dst = o.dst) FROM ss o ), fullTable AS (  SELECT DISTINCT ON(dir, gsrc, dst) dir, gsrc, dst, prev, w, dead  FROM
newPoints  ORDER BY dir, gsrc, dst, w, dead, prev DESC ), found AS (SELECT min(l.w + r.w) AS wFROM fullTable l, fullTable
rWHERE l.dir = false AND r.dir = true AND l.dst = r.dst) SELECT dir, gsrc, dst, prev, w, dead OR (coalesce(t.w > (SELECT
f.w/2 FROM found f), false)), e.iter + 1 AS iter FROM fullTable t, (SELECT iter FROM toExplore limit 1) e) ), ss(dir, gsrc,
dst, prev, w, iter) AS (SELECT dir, gsrc, dst, prev, w, iter FROM shorts WHERE iter = (SELECT max(iter) FROM shorts)),
result(f, t, inter, w) AS ( SELECT l.gsrc, r.gsrc, l.dst, l.w + r.w FROM ss l, ss r WHERE l.dir = false AND r.dir = true
AND l.dst = r.dst ORDER BY l.w + r.w LIMIT 1), sp1(arr, cur) as ( SELECT ARRAY[inter]::bigint[], inter FROM result UNION
ALL SELECT array_prepend(ss.prev, sp1.arr), ss.prev FROM ss, sp1 WHERE ss.dir = false AND ss.dst = sp1.cur AND ss.prev <>
ss.dst), sp2(arr, cur) as ( SELECT (SELECT arr FROM sp1 WHERE cur = (SELECT f FROM result)), (SELECT inter FROM result)
UNION ALL SELECT array_append(sp2.arr, ss.prev), ss.prev FROM ss, sp2 WHERE ss.dir = true AND ss.dst = sp2.cur AND ss.prev
<> ss.dst) SELECT sp2.arr AS personIdsInPath, result.w AS pathWeight FROM result, sp2 WHERE sp2.cur = result.t;
```

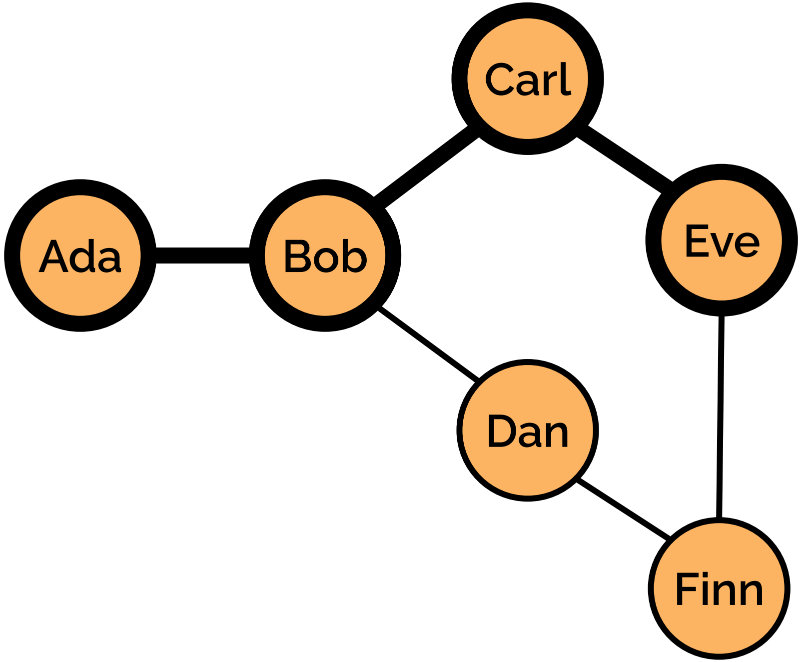
Cheapest path query: Q14 new version



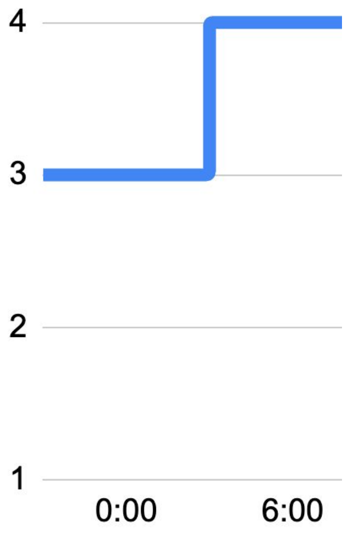
Path curation

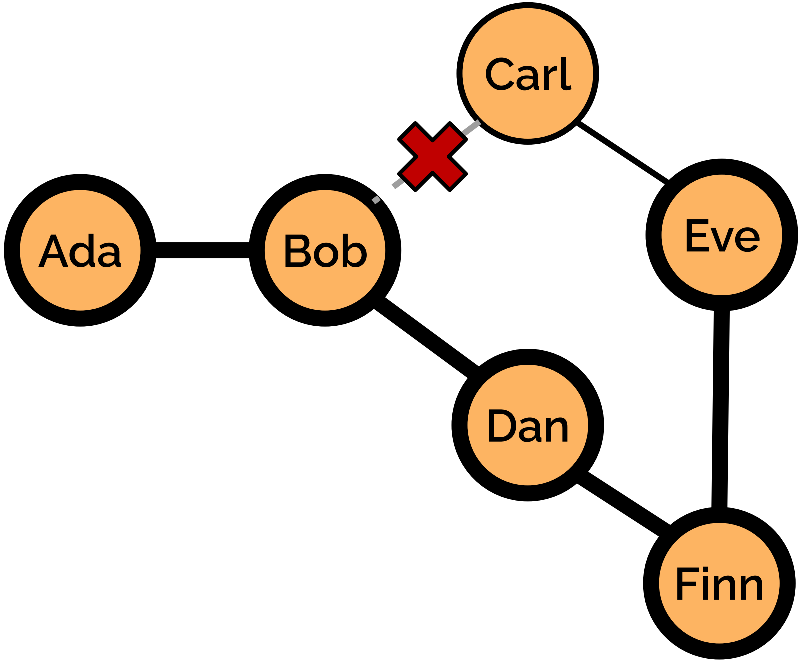
Shortest distance from “Ada” to “Eve”



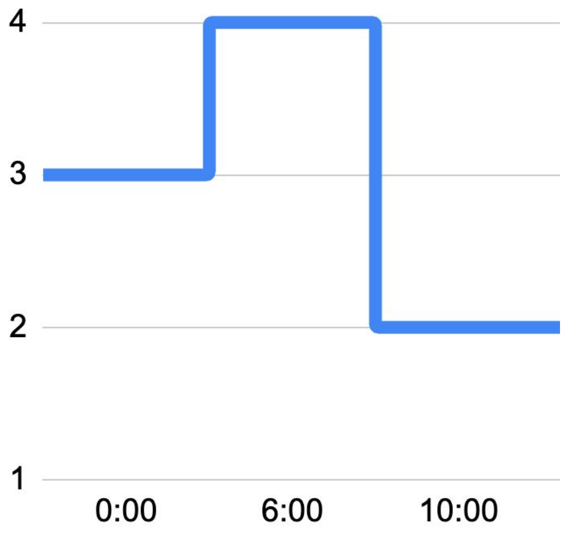


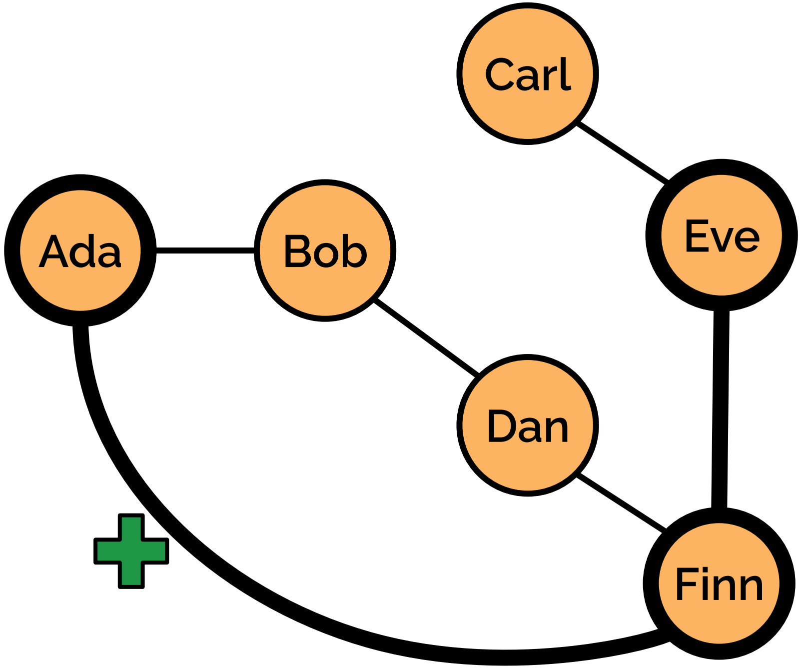
Shortest distance from “Ada” to “Eve”



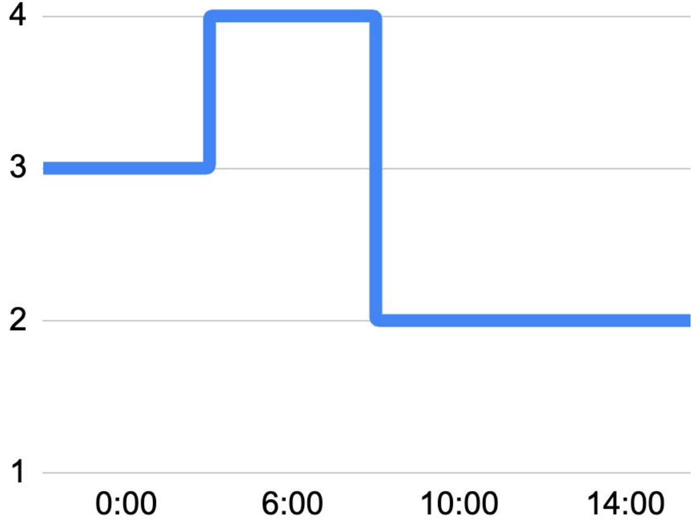


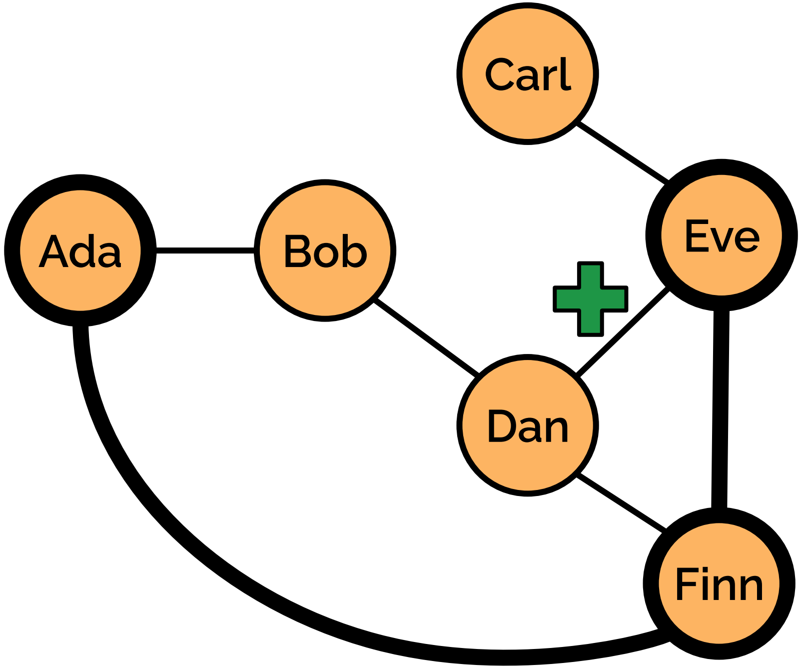
Shortest distance from “Ada” to “Eve”



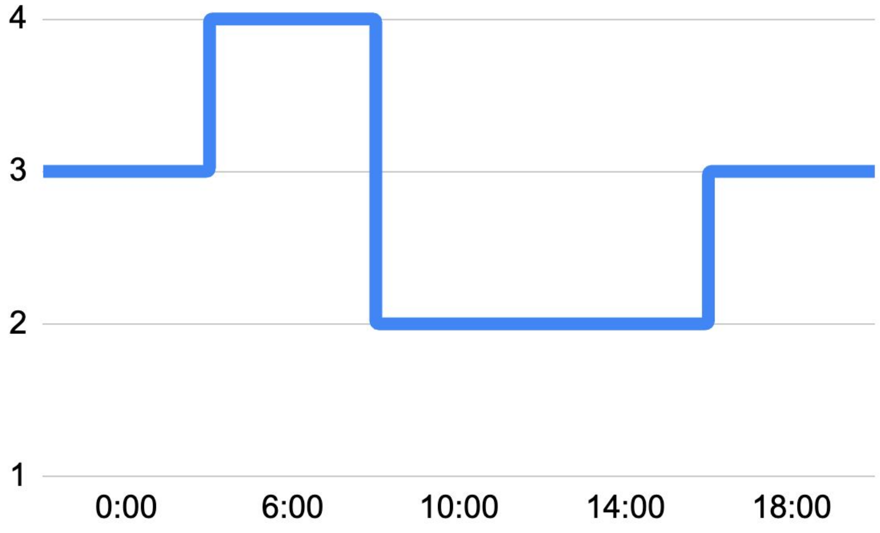


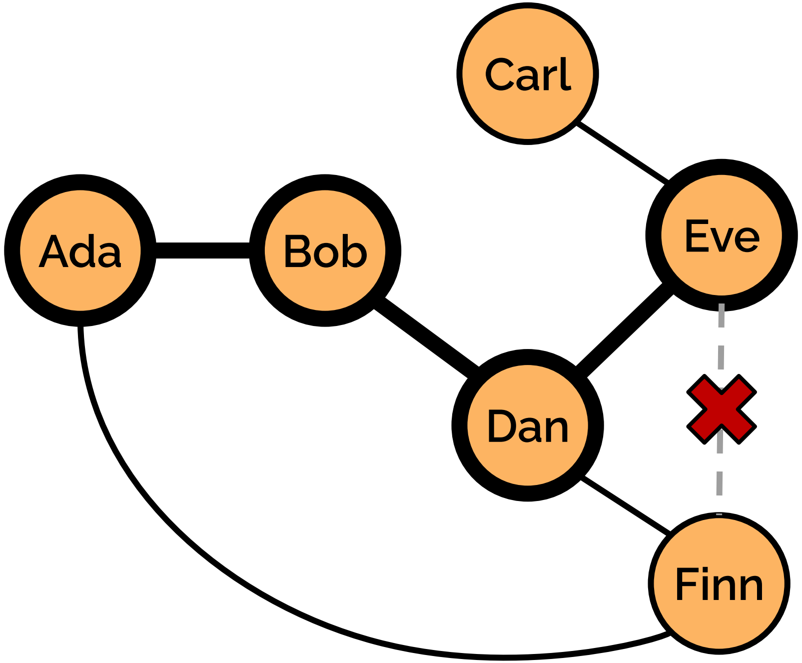
Shortest distance from “Ada” to “Eve”





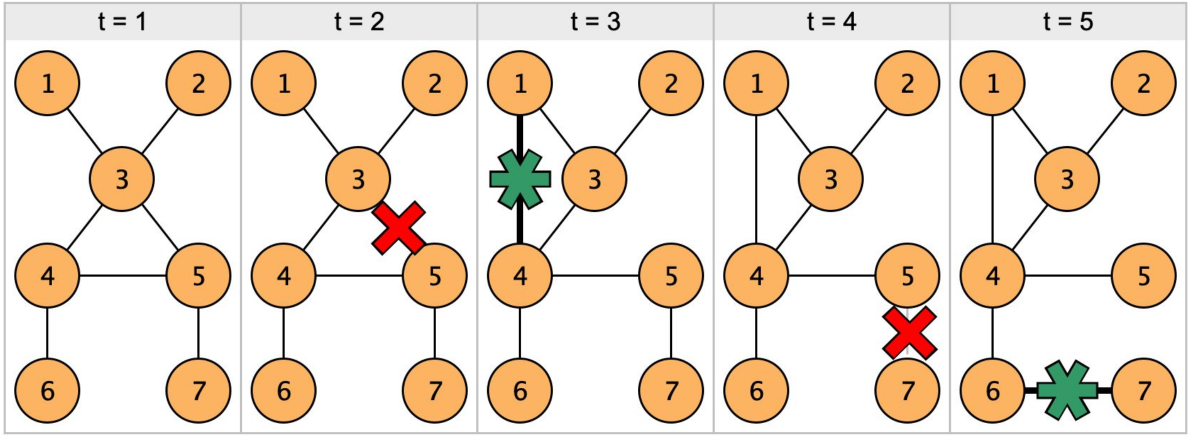
Shortest distance from “Ada” to “Eve”





The shortest path distance changes multiple times during the day.

Path curation with temporal bucketing

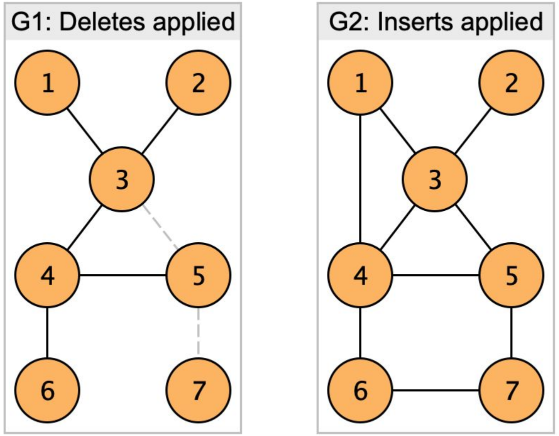


For each day, we construct:

G1 – deletes but no inserts, setting an *upper* bound

G2 – inserts but no deletes, setting a *lower* bound

lower ≤ actual length ≤ upper



Pairs of nodes yielding 3-hop paths in G1 and G2:

- ~~1 to 5~~
- ~~1 to 6~~
- ~~2 to 5~~
- 2 to 6

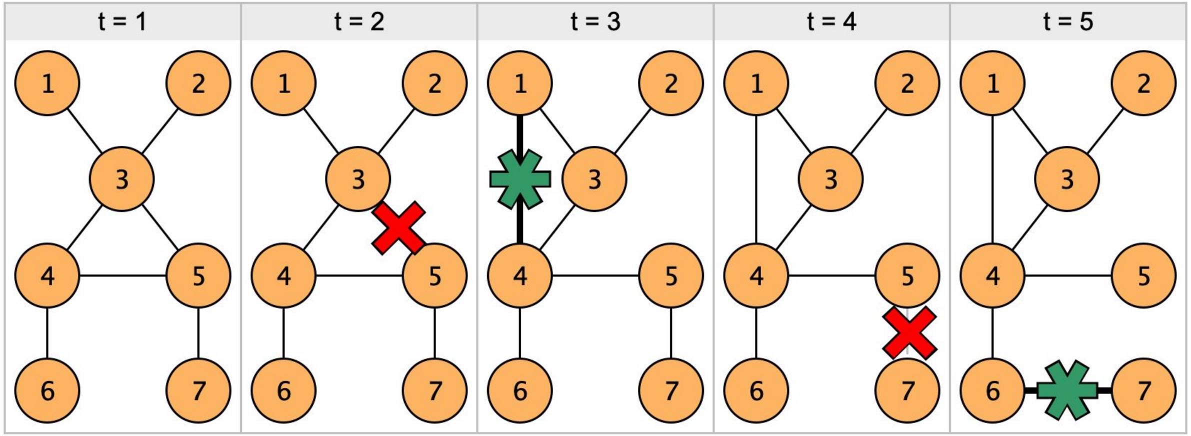
Path curation with temporal bucketing

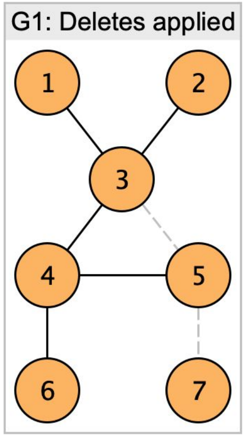
For each day, we construct:

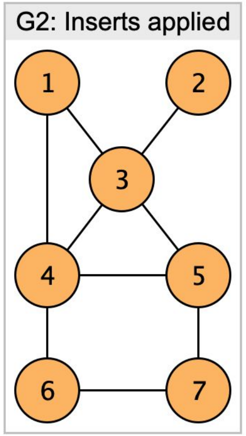
G1 – deletes but no inserts, setting an *upper* bound

G2 – inserts but no deletes, setting a *lower* bound

lower ≤ actual length ≤ upper







Connected components algorithm on G2

Pairs of nodes in different components are guaranteed to be unreachable that day

Is path curation alone sufficient?

Not yet:

- We also have to consider the degree distribution of the source–target nodes.

Actually:

- For “perfect” parameter curation, we would need to run the entire workload with many parameter candidates and only keep ones which showed a similar behaviour.

Summary

Implementations

system	data model	language
neo4j	graph	Cypher
PostgreSQL	relational	SQL
Microsoft® SQL Server®	relational	SQL + graph extension
UMBRA	relational	SQL

SNB Interactive v2

- A scalable, transactional database benchmark
- Interesting queries (correlated vs. anti-correlated, cheapest path finding)
- Deep delete operations
- State-of-the-art parameter selection
- Fine-tuning ongoing, to be released in 2024

Please reach out if you would like to implement the benchmark

LDBC

The graph & RDF
benchmark reference