Easy Freshness with Pequod Cache Joins

Authors:
Bryan Kate, Eddie Kohler, Michael S. Kester - Harvard University
Neha Narula, Yandong Mao, Robert Morris - MIT/CSAIL

Presented by:
Paweł Zięcik - University of Warsaw

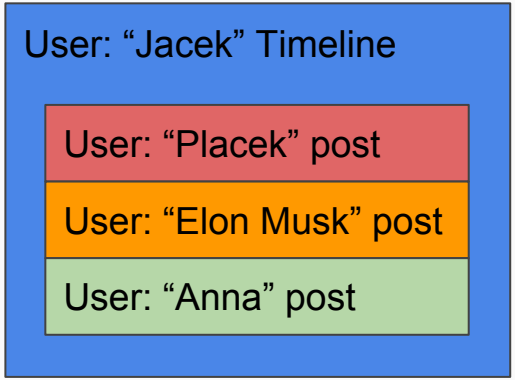
Agenda

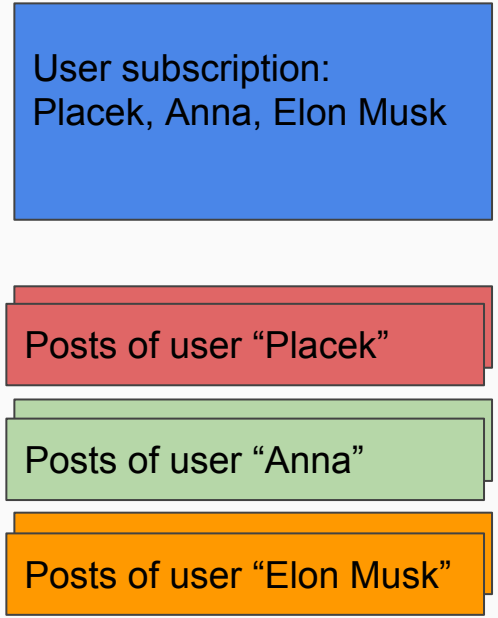
1. Problem and motivation
2. Solution design
3. Implementation
4. Evaluation

Problem and motivation

Problem - Twitter like system

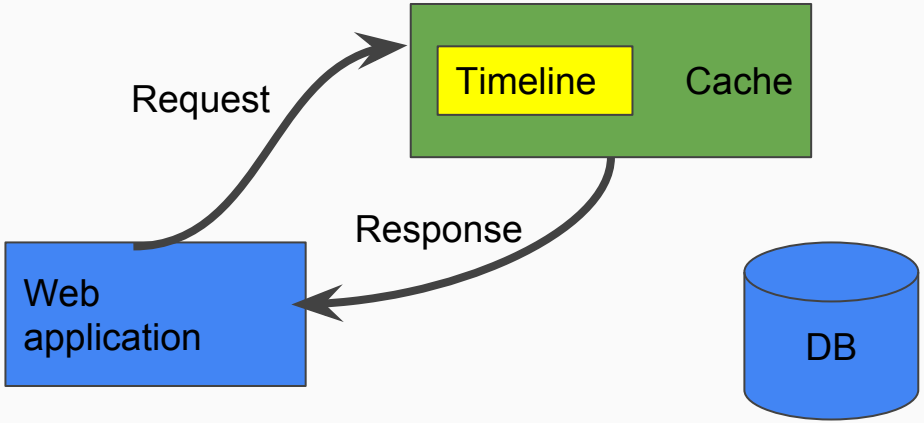
- Many timeline checks
- Construction of timeline is expensive
- We want to use caching for performance reason





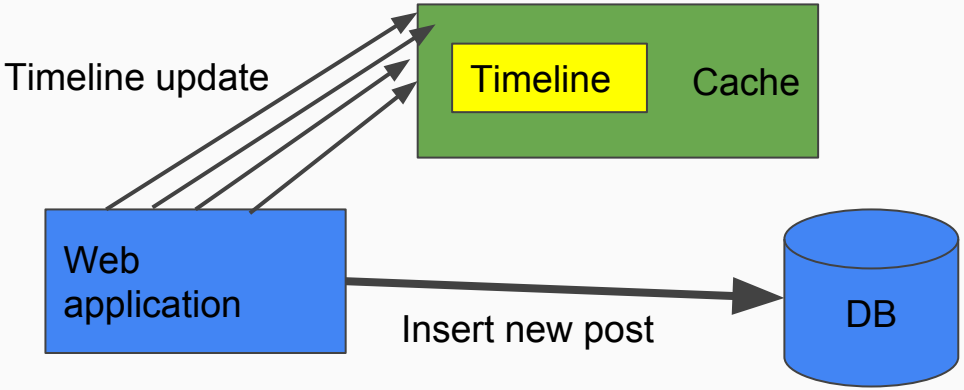
Performance of key-value cache

- Cached timeline increases performance



Performance of key-value cache

- When a new post arrives many cached timelines has to be updated
- Maintenance of timeline freshness may be challenging



Programmability of database

- Easy to write
- Maybe expensive to compute

```
SELECT post.time, post.author post.content
  FROM post JOIN sub
    WHERE sub.follows = post.author
      AND sub.user = “Jacek”
      AND sub.time >= 100
  ORDER BY post.time
```

Materialized view

- Query result saved in database
- Much better performance -> second query is inexpensive
- Still significantly worse performance comparing to caching

```
CREATE MATERIALIZED VIEW tline AS
SELECT sub.user, post.time, post.author, post.content
  FROM post JOIN sub
    WHERE sub.follows = post.author

SELECT * FROM tline
  WHERE tline.user = ‘Jacek’ AND tline.time >= 100
  ORDER BY tline.time
```

Solution

Pequod

Pequod is a distributed cache system with two key features

- **Performance** of key-value cache
  - Support operations: get, put, scan, **join**
- **Programmability** of relational database
  - Easy to program to keep freshness of data
  - Achieved by cache joins resembling database materialized views

Cache joins - KV materialized views

- Pequod is a key-value cache it understands get, put, scan operations
- How to store relations to create materialized view?
- Store post and subscription tables in cache
- Store some column information in keys

```
“post|<author>|<time>” -> <post_content>

“sub|<user>|<poster>”  -> “”
```

Cache joins

```
CREATE MATERIALIZED VIEW tline AS
SELECT sub.user, post.time, post.author, post.content
  FROM post JOIN sub
    WHERE sub.follows = post.author
```

```
tline|<user>|<time>|<author> =
  check sub|<user>|<author>
  copy post|<author>|<time>;
```

Cache joins

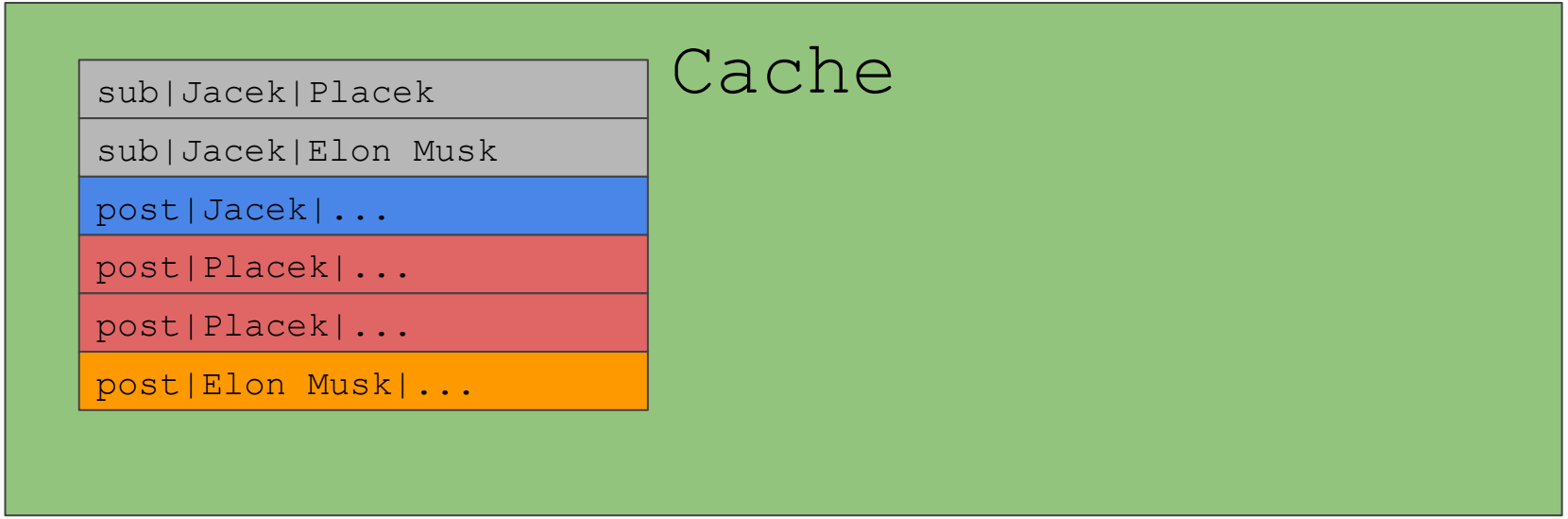
- Once we defined the cache join we can fetch timeline with scan operation

```
scan(tline|Jacek|100, tline|Jacek*)
```

- Order of columns in key is important

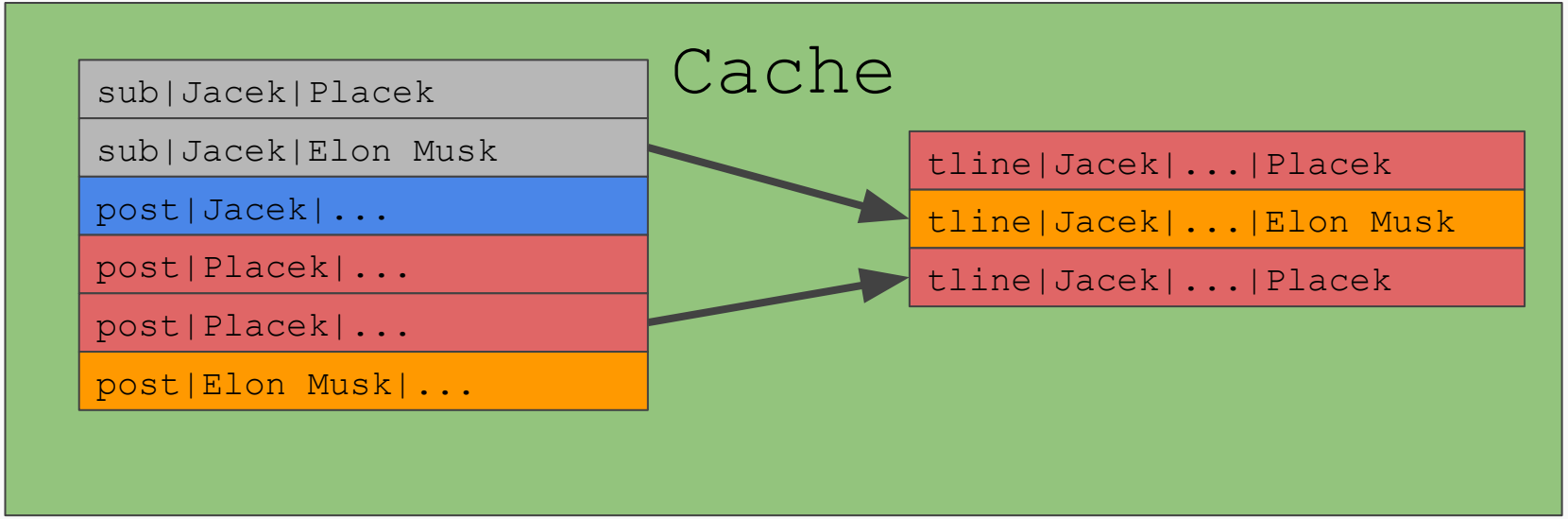
Cache joins

- Timeline request arrives `scan(tline|Jacek|100, tline|Jacek*)`
- Cache does not contain materialized tline



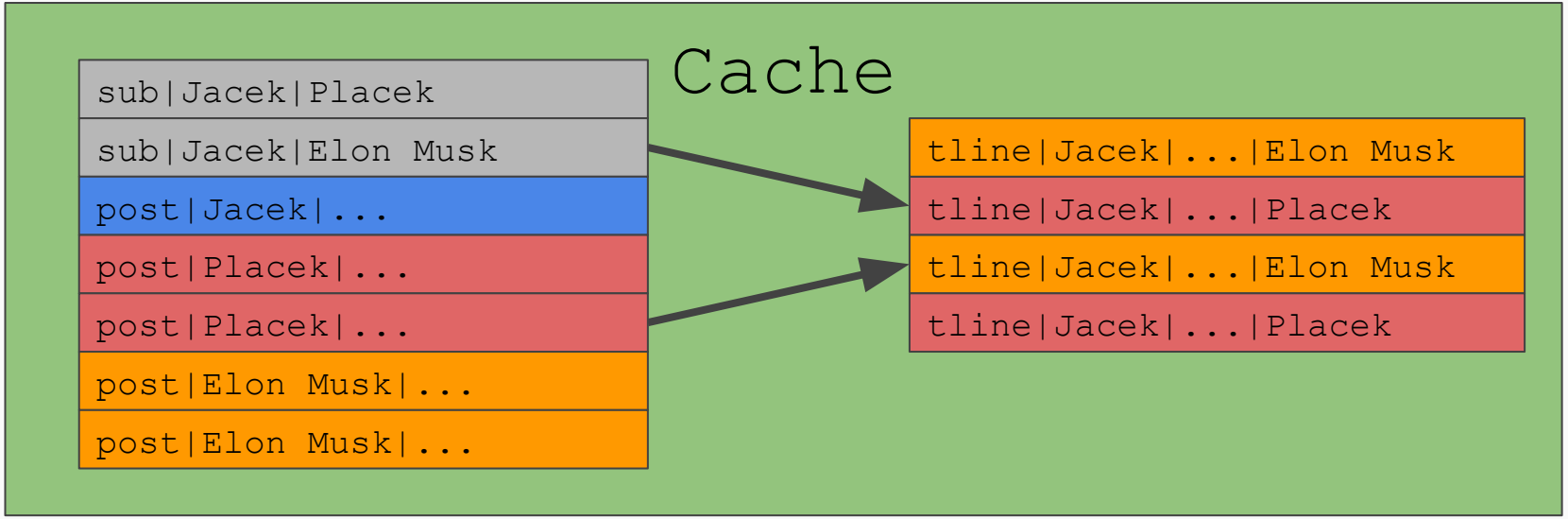
Cache joins

- Cache join is performed and requested part of timeline table is materialized



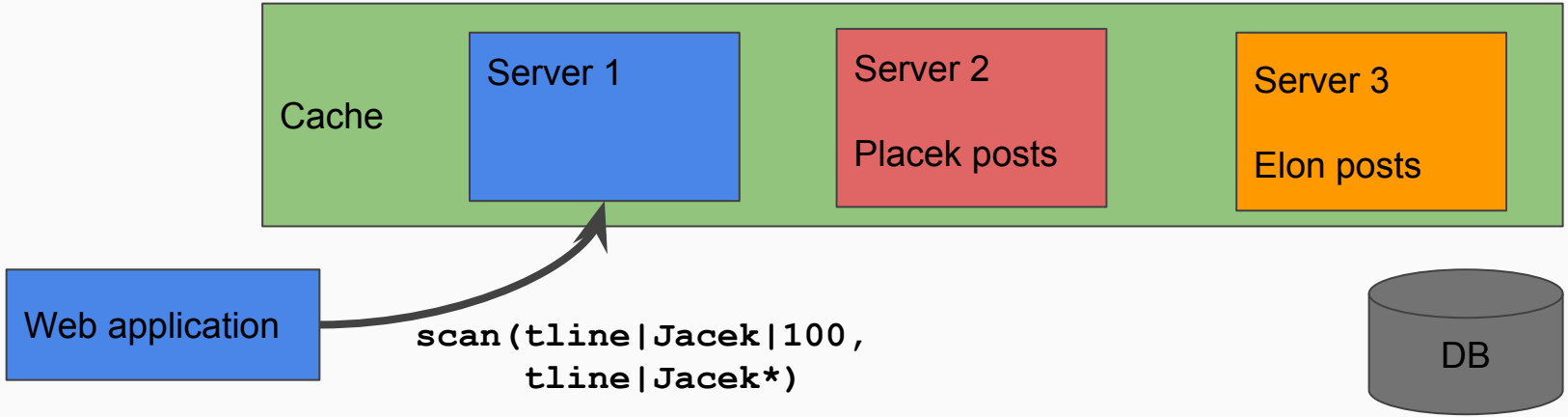
Eager update

- When new post arrives eager update to tline table is performed
- Lazy updates also supported



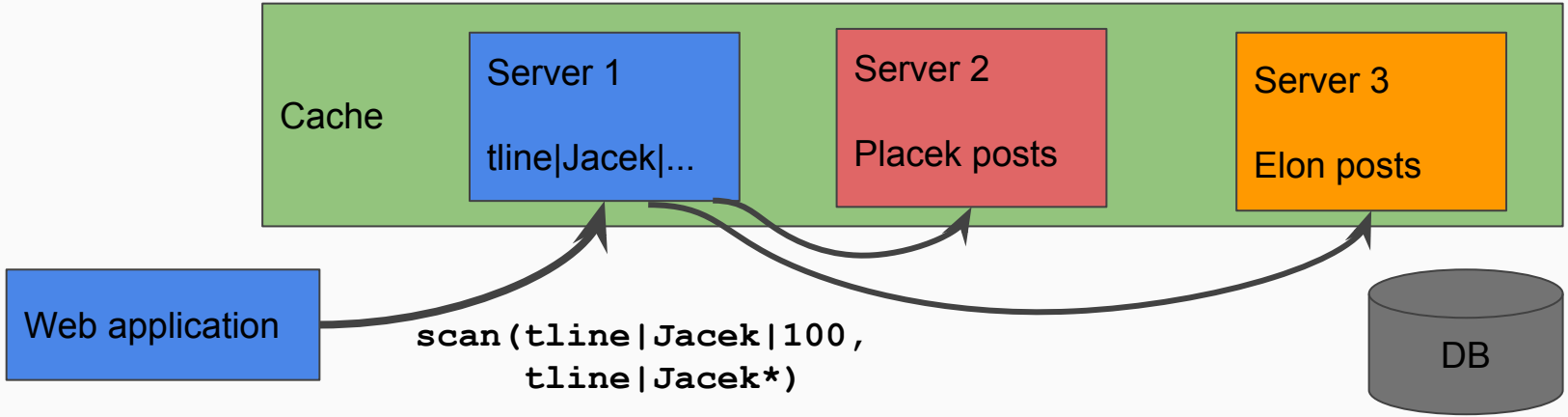
Distributed deployment (read)

- Timeline request arrives



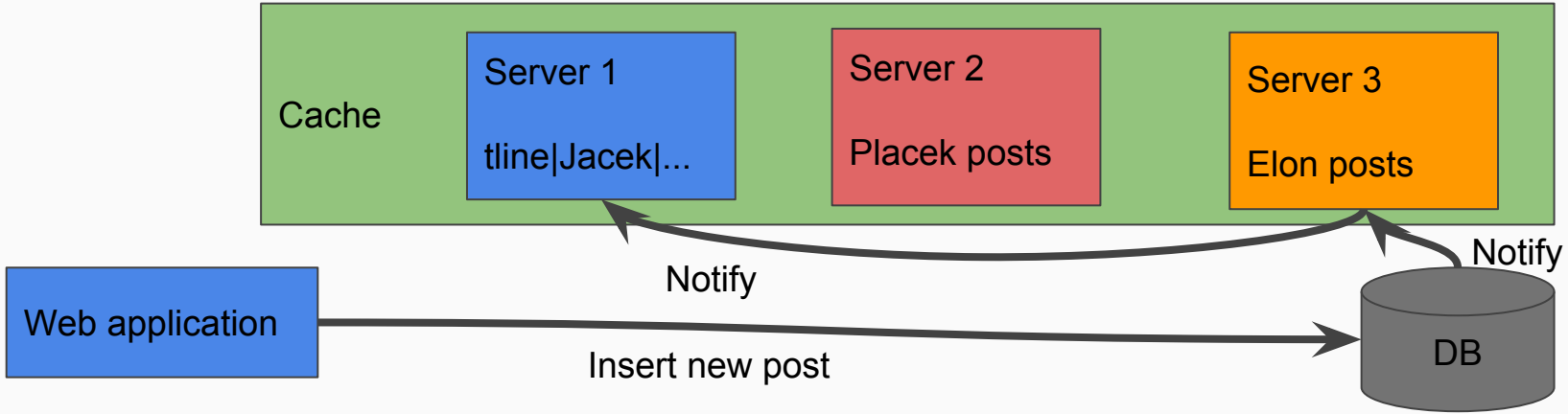
Distributed deployment (read)

- Timeline request arrives
- Based data is requested from other servers to materialized tline table



Distributed deployment (write)

- New post arrives
- Insert to database and notify relevant servers to update tline table



Implementation

Data storage

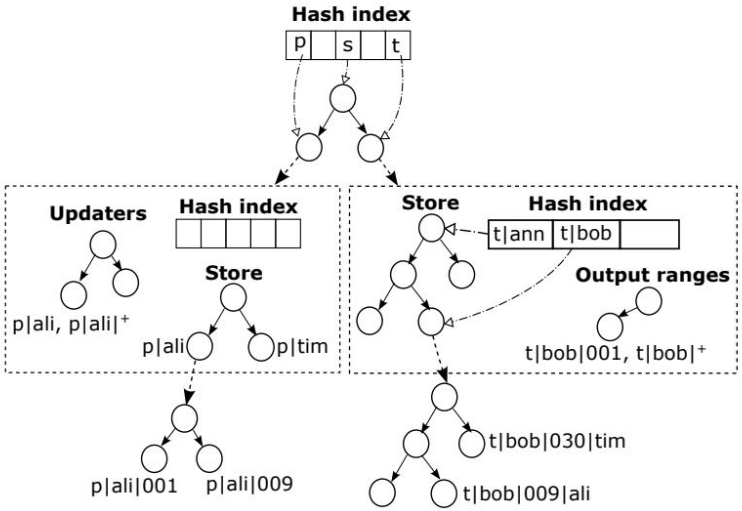


Figure 6: Pequod internal data structure for Twip. The logical store divides into tables (the rectangle layer) and, when appropriate, subtables (the lowest layer).

- Data stored in red-black trees
- Multilevel vs single level trees
  - 1.55x performance improvement
  - 1.17x higher memory consumption

Cache join query execution

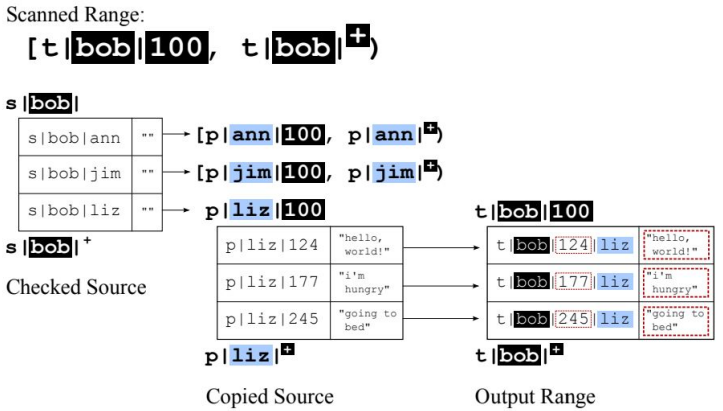


Figure 4: Example query execution of the timeline join. The scanned range provides context used when scanning source ranges. The keys and values in the output key range comprise elements of the original scan range and both source ranges.

- Execution is logically an iteration over source elements
- Yield a key-value for matching tuple
- Move selection operators as early as possible

Keeping freshness

- We have two auxiliary data structures to update cache joins
- Join status range
  - Information whether given join output range is valid (fresh)
  - `Js1 = [tline|Jacek|100, tline|Jacek*) -> {isValid : Bool}`
- Updaters
  - Maps source range to tuple of cache join, slot set, join status range
  - `[post|Elon|100, post|Elon|*) -> {tline, {user->Jacek}, js1}`

Keeping freshness

- We store updaters in an interval tree
- The entry is modified
- If entry key belongs to updater source range, the incremental update is triggered

Evaluation

Evaluation

- Data based on 2009 Twitter social graph
- Ratios of check, subscribe, post events are 100:10:1
- Database omitted in experiments

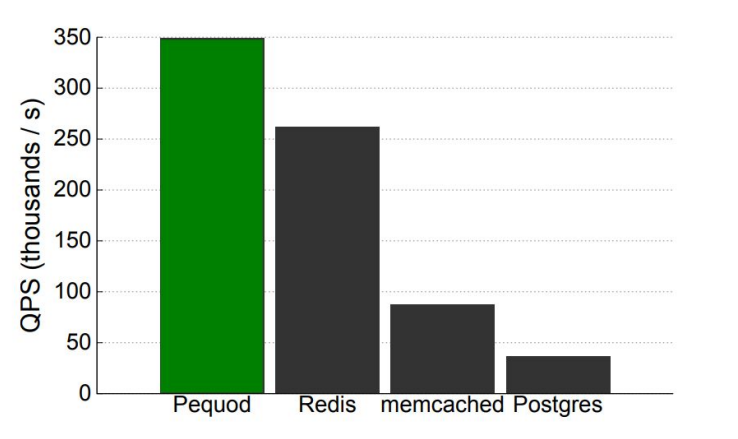
Performance evaluation

Do cache joins have key-value cache performance?

- Key-value caches
  - Redis
  - Memcached
- Pequod
  - With cache joins
  - Without cache joins (Pequod client)
- In memory DB-as-cache
  - Postgres

Evaluation

- Performance of key-value cache



System	Runtime
Pequod	197.06 s (1.00x)
Redis	262.62 s (1.33x)
Client Pequod	323.29 s (1.64x)
memcached	784.43 s (3.98x)
PostgreSQL	1882.78 s (9.55x)

Figure 7: Time to process a Twip experiment to completion using Pequod and related systems. Smaller numbers are better.

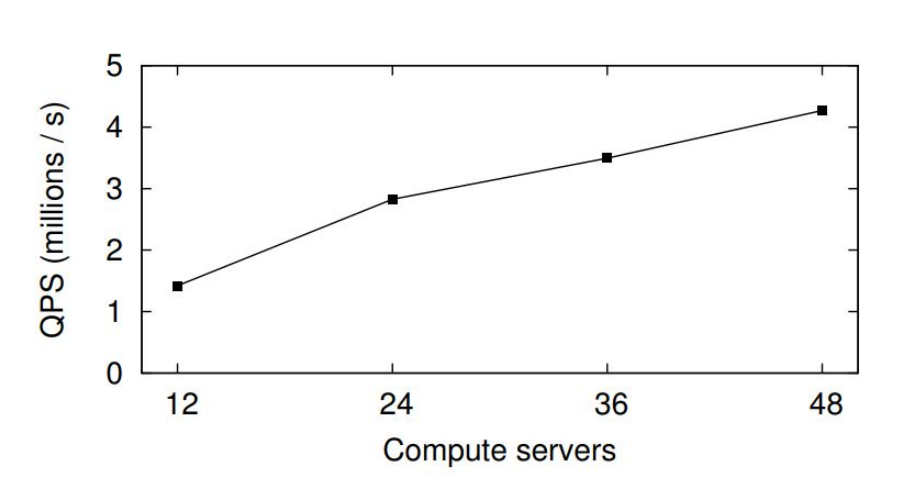
Scalability evaluation

Does Pequod scales well?

- Cluster on Amazon EC2
- Two tier deployment
  - Posts and subscriptions data on “base” servers
  - Timeline storing and constructing on “compute” servers

Scalability evaluation

- Increasing number of servers 4x improves performance 3x



Scalability evaluation (overhead)

- Increase in fraction of inter-server communication
  - 10% -> 16%
- Memory consumption on compute servers
  - 1.2 TB -> 1.5 TB (duplicate data)
- Memory consumption on base servers
  - 290 GB -> 297 GB (subscription metadata)

Conclusion

Pequod cache joins

- Performance of key-value cache
- Programmability of database materialized views

Bibliography

- Easy Freshness with Pequod Cache Joins Bryan Kate, Eddie Kohler, and Michael S. Kester, Harvard University; Neha Narula, Yandong Mao, and Robert Morris, MIT/CSAIL
- https://www.usenix.org/sites/default/files/conference/protected-files/nsdi14_slides_kate.pdf

Questions?