A cross-technology benchmark for incremental graph queries

Georg Hinkel, Antonio Garcia-Dominguez, René Schöne, Artur Boronat, Massimo Tisi, Théo Le Calvar, Frederic Jouault, József Marton, Tamás Nyíri, János Benjamin Antal, Márton Elekes, Gábor Szárnyas

Presenter: Gábor Szárnyas (CWI Amsterdam)

Software and Systems Modeling 2022 | MODELS 2022 J1 track

TTC 2018 “Social media” case

News | Your solutions ▾ | Programme | Calls ▾ | Aims and scope | People | History ▾

11th Transformation Tool Contest

A contest for users and developers of transformation tools.

Part of the Software Technologies: Applications and Foundations (STAF) federated conferences.

Hosted at the IRIT in Toulouse, France on Friday **29 June 2018**.

"Social media" case

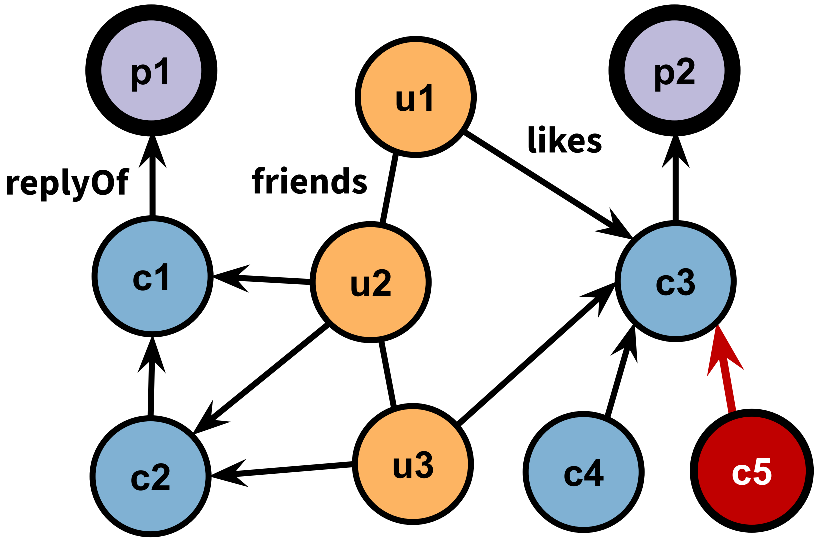
Data	Social network graph
Queries	Score posts and comments
Changes	New entities are inserted
Goal	Keep query results up-to-date

Scoring posts

Score = 10 × #comments + #likes

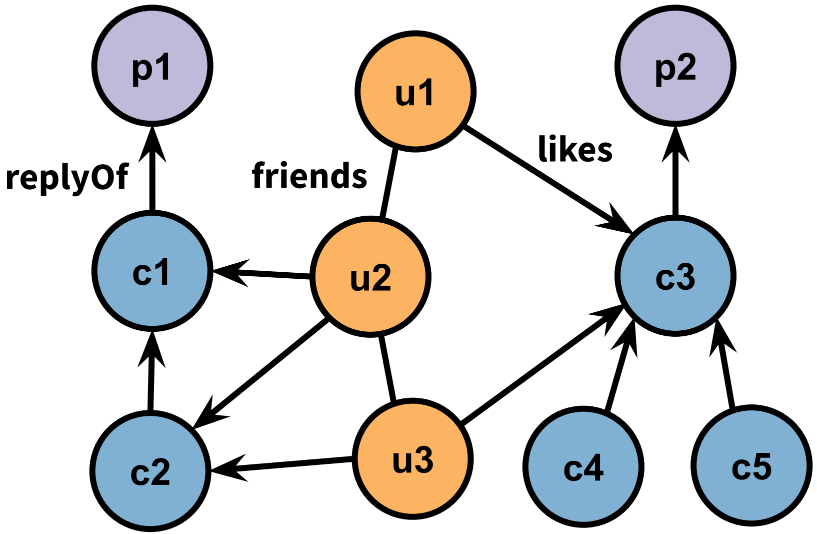
Score: 23

~~Score: 22~~
Score: 32



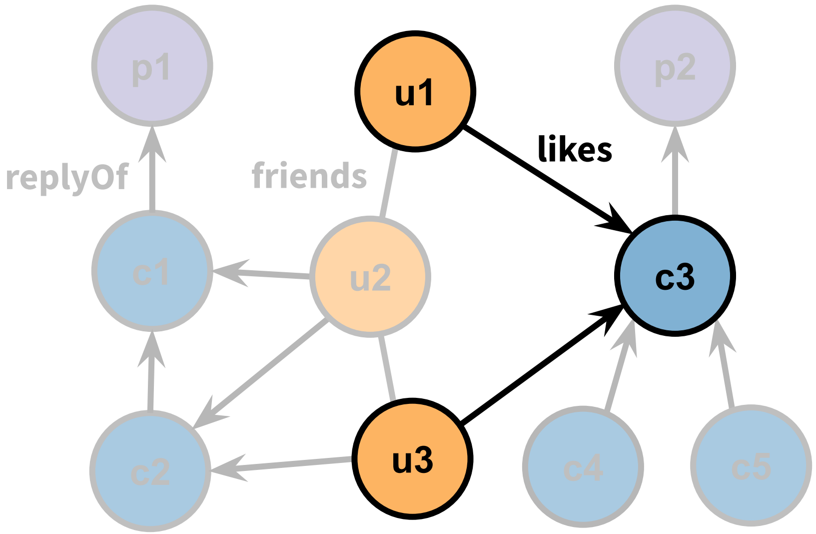
Scoring comments

For each comment, find connected components of users who liked the comment



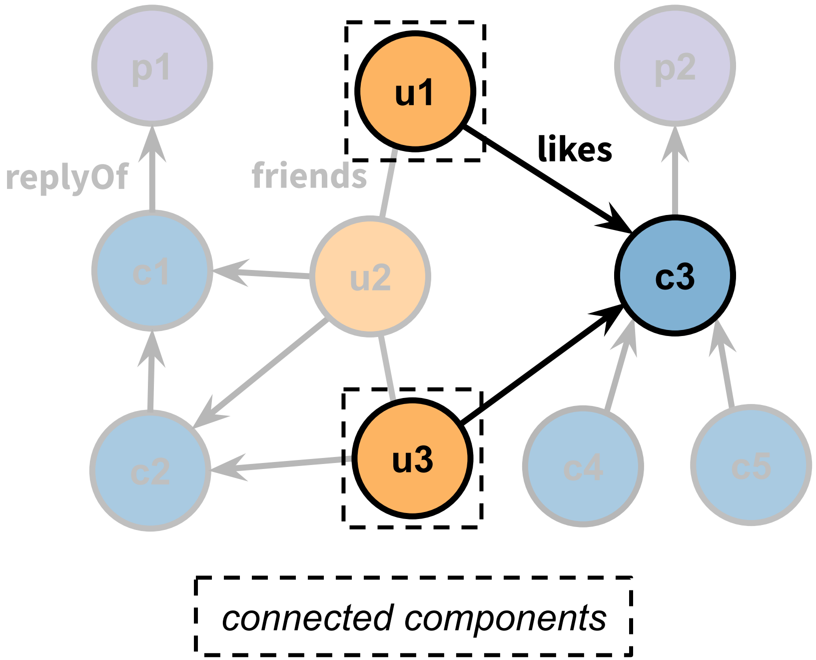
Scoring comments

Score = Σ (component size)^2



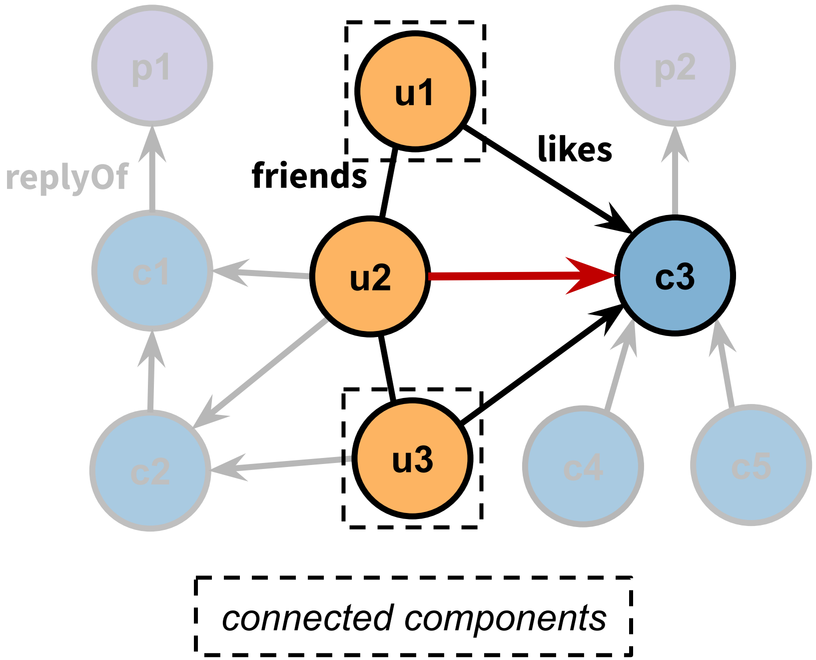
Scoring comments

Score = Σ (component size)^2 = 1^2 + 1^2 = 2



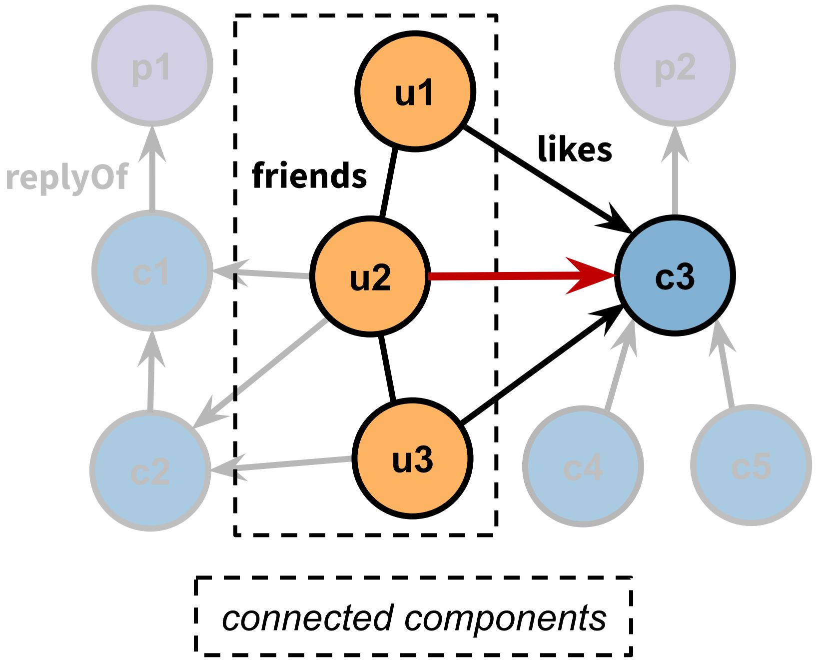
Scoring comments

Score = Σ (component size)^2 = $1^2 + 1^2 = 2$



Scoring comments

Score = Σ (component size)^2 = $3^2 = 9$



Solutions

Solution	Data model	Variants
Active Operations Framework	EMF	1
ATL	EMF	2
Hawk	EMF	3
JastAdd	EMF	[illegible]
Xtend	EMF	[illegible]
YAMTL	EMF	3
NMF	NMF	2
Differential Dataflow	relational	1
PostgreSQL	relational	2
Neo4j	graph	2
GraphBLAS	matrix	2

Most solutions use the Eclipse Modeling Framework

21 solutions in total

DBMSs

Non-incremental query formulation

Examples of the how the initial query evaluation is formulated in:

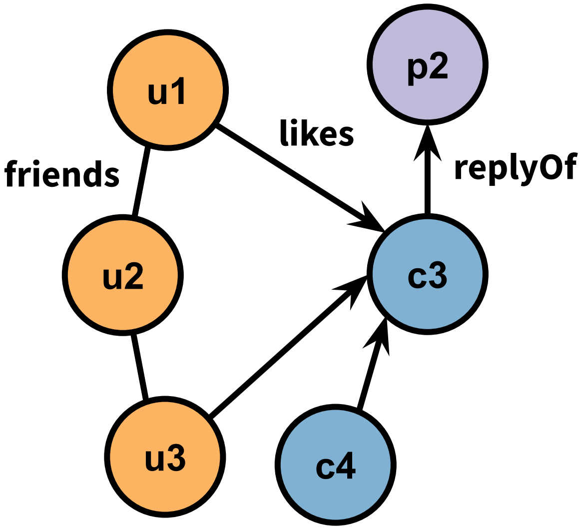
- NMF
- Neo4j
- PostgreSQL

Note: Implementations can be quite complex – this is a “programming contest”

Scoring posts

Score = 10 × #comments + #likes

Traversing the Submission tree



NMF

```
post.Descendants()
```

Neo4j (Cypher)

```
MATCH (p:Post)
OPTIONAL MATCH (p)<-[:REPLY_OF*]-(c:Comment)
```

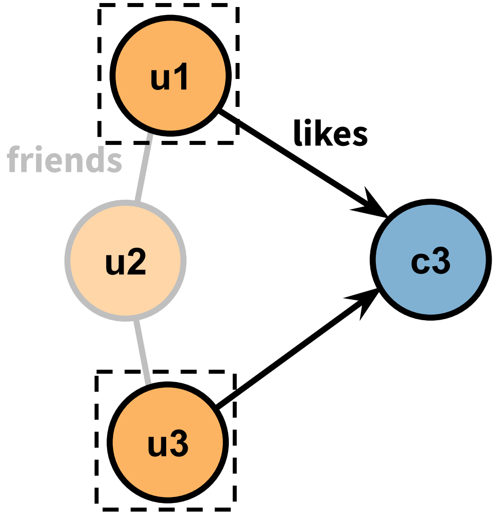
PostgreSQL (SQL:1999)

```
WITH RECURSIVE
  comments_with_ancestors(id, ancestorid) AS (
        SELECT c.id, c.parentid AS ancestorid
          FROM comments c
      UNION
        SELECT cr.id, c.parentid AS ancestorid
          FROM comments_with_ancestors cr
             , comments c
         WHERE cr.ancestorid = c.id)
```

Scoring comments

Score = Σ (component size)²

Finding connected components of Users



NMF: Tarjan's algorithm

```
let layering = Layering<IUser>.CreateLayers(
  comment.LikedBy,
  u => u.Friends.Intersect(comment.LikedBy))
let score = layering.Sum(l => Square(l.Count))
```

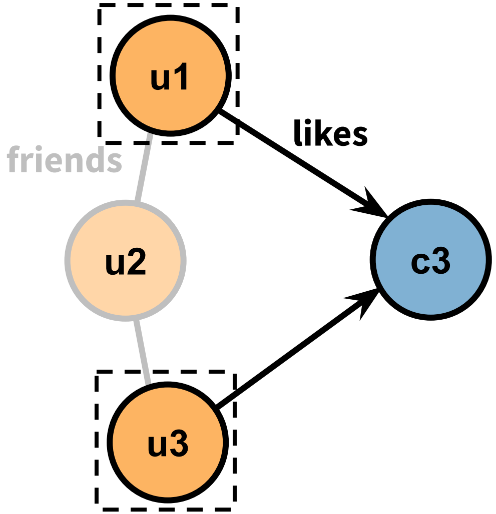
Neo4j: User-defined function

```
MATCH (c:Comment) WHERE (c)<-[:LIKES]-(:User)
CALL gds.wcc.stream({
  nodeQuery:
    "MATCH (c:Comment)<-[:LIKES]-(u:User)
     WHERE id(c)=" + id(c) + "
     RETURN id(u) AS id",
  relationshipQuery:
    "MATCH (u1:User)<-[:FRIENDS]->(u2:User)
     RETURN id(u1) AS source, id(u2) AS target",
  validateRelationships: false
})
YIELD componentId
...
```

Scoring comments

Score = Σ (component size)^2

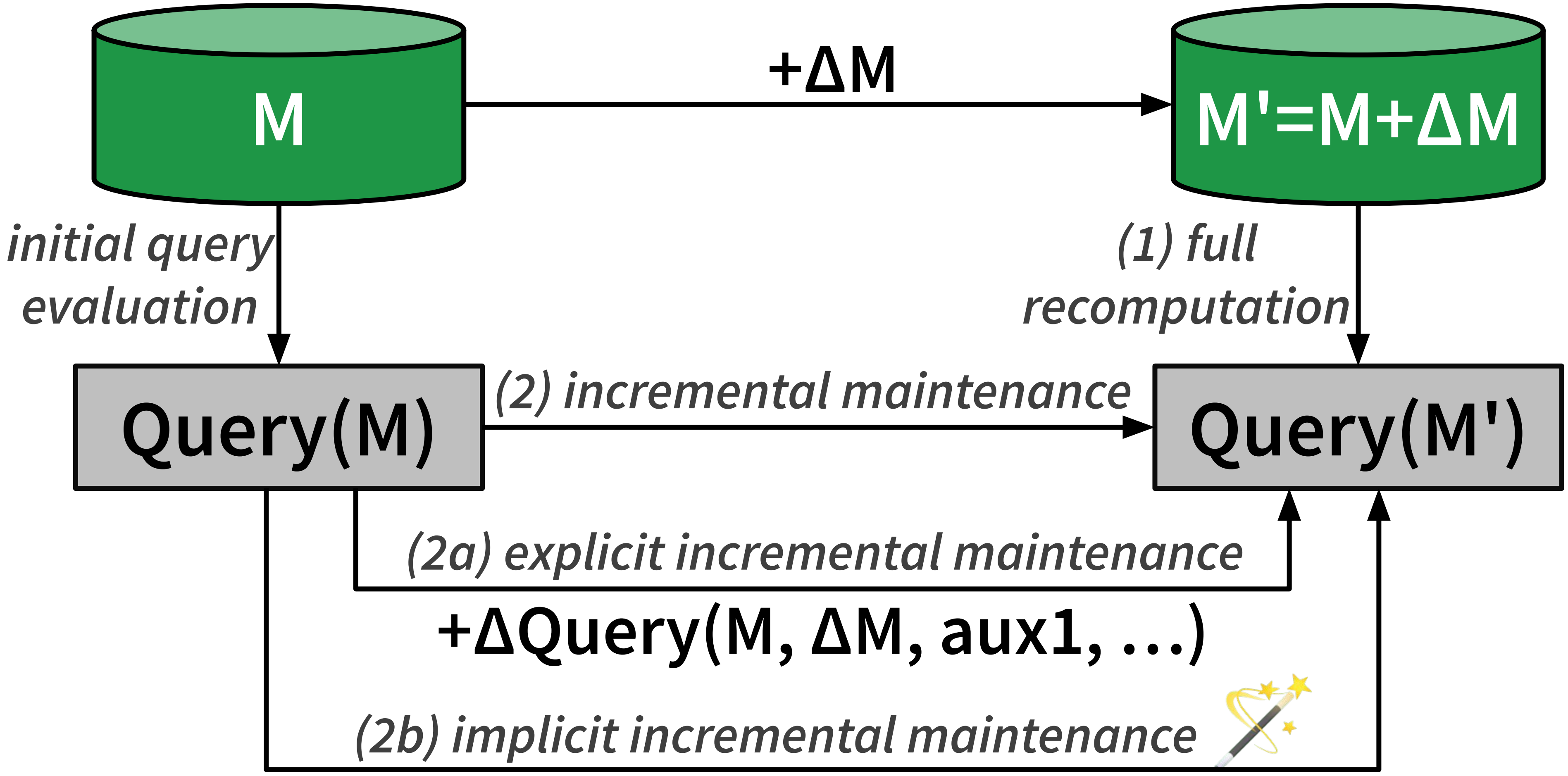
Finding connected components of Users



PostgreSQL: Simplified SQL query

```
WITH RECURSIVE
  comment_friends(commentid, user1id, user2id) AS ( ... ),
  comment_friends_closed(commentid, head_userid, tail_userid) AS
    SELECT l.commentid
         , l.userid AS head_userid, l.userid AS tail_userid
      FROM likes l
     UNION
    SELECT cfc.commentid, cfc.head_userid, f.user2id AS tail_userid
      FROM comment_friends_closed cfc, comment_friends f
     WHERE cfc.tail_userid = f.user1id
       AND cfc.commentid = f.commentid
), comment_components AS (
    SELECT commentid, head_userid AS userid
         , min(tail_userid) AS componentid
      FROM comment_friends_closed
     GROUP BY commentid, head_userid
), comment_component_sizes AS (
    SELECT cc.commentid, cc.componentid, count(*) AS component_size
      FROM comment_components cc
     GROUP BY cc.commentid, cc.componentid
)
SELECT c.id AS commentid
     , coalesce( sum( power(ccs.component_size, 2) ), 0) AS score
  FROM comments c
  LEFT JOIN comment_component_sizes ccs ON (ccs.commentid = c.id)
 GROUP BY c.id, c.ts
...
```

Incremental maintenance

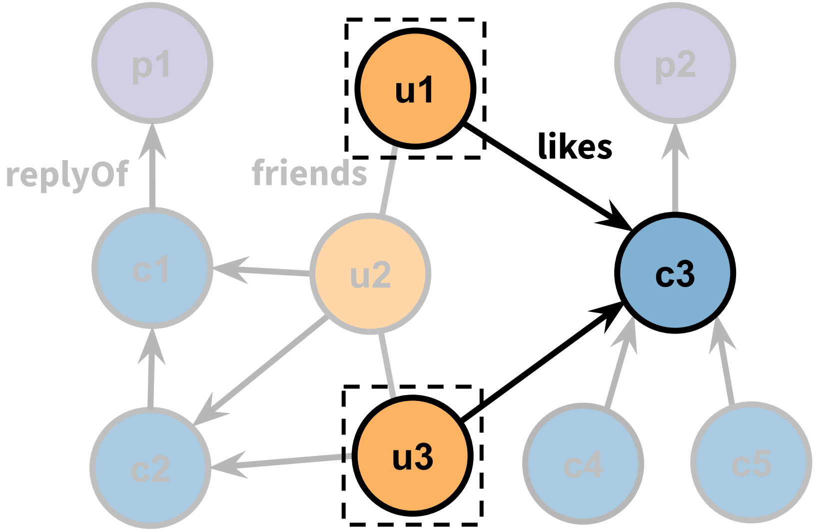

M
+ΔM
M'=M+ΔM
initial query evaluation
(1) full recomputation
Query(M)
(2) incremental maintenance
Query(M')
(2a) explicit incremental maintenance
+ΔQuery(M, ΔM, aux1, ...)
(2b) implicit incremental maintenance

Solution	Data model	Explicitly incremental	Implicitly incremental
Xtend	EMF	–	–
Hawk	EMF	+	–
PostgreSQL	relational	+	–
Neo4j	graph	+	–
GraphBLAS	matrix	+	–
Active Operations Framework	EMF	–	+
ATL	EMF	–	+
JastAdd	EMF	–	+
NMF	NMF	–	+
Differential Dataflow	relational	–	+
YAMTL	EMF	+	+

Scoring comments

Score = Σ (component size)^2

Finding connected components of Users



Incremental evaluation

The **granularity** of the incremental maintenance has a big effect on performance:

- New “likes” edge → recalculate only for the the affected single comment
- New “knows” edges → recalculate for each affected comments
- Reusing existing connected components?

Scoring comments: SQL incremental

```sql
INSERT INTO comment_friends (status,
        commentid, user1id, user2id)
    SELECT 'B' AS status
         , l1.commentid, f.user1id,
               f.user2id
      FROM likes l1, likes l2
         , friends f
     WHERE l1.userid = f.user1id
       AND f.user2id = l2.userid
       AND l1.commentid = l2.commentid;
```

Listing 36 Initialization phase for the Incremental PostgreSQL solution for Q2, initializing the *comment_friends* relation.

```sql
INSERT INTO comment_friends (status,
        commentid, user1id, user2id)
    SELECT 'D' AS status
         , l1.commentid, f.user1id,
               f.user2id
      FROM likes_d l1, likes l2
         , friends f
     WHERE l1.userid = f.user1id
       AND f.user2id = l2.userid
       AND l1.commentid = l2.commentid
  UNION ALL
    SELECT 'D' AS status
         , l1.commentid, f.user1id,
               f.user2id
      FROM likes_b l1, likes l2
         , friends_d f
     WHERE l1.userid = f.user1id
       AND f.user2id = l2.userid
       AND l1.commentid = l2.commentid
  UNION ALL
    SELECT 'D' AS status
         , l1.commentid, f.user1id,
               f.user2id
      FROM likes_b l1, likes_d l2
         , friends_b f
     WHERE l1.userid = f.user1id
       AND f.user2id = l2.userid
       AND l1.commentid = l2.commentid;
```

Listing 37 SQL maintenance phase for the Incremental PostgreSQL solution for Q2: updating the *comment_friends* relation.

```sql
WITH RECURSIVE
         comment_friends_closed_init(
    commentid, head_userid,
            tail_userid) AS (
-- transitive closure
         (reachability-only, no path is
         recorded)
-- of friendship-subgraphs defined by
         comment likes

    -- start with the users that liked
            a specific comment.
    -- They are the nodes of the
            projected users graph for a
            comment
    SELECT l.commentid, l.userid AS
            head_userid, l.userid AS
            tail_userid
      FROM likes l
  UNION
    -- expand the closure with the
            edges of the projected
            graph,
    -- which is stored in
            comment_friends table
    SELECT cfc.commentid,
            cfc.head_userid, f.user2id
            as tail_userid
      FROM comment_friends_closed_init
            cfc
         , comment_friends f
     WHERE cfc.tail_userid = f.user1id
       AND cfc.commentid = f.commentid
)
INSERT INTO
         comment_friends_closed(commentid,
         head_userid, tail_userid)
    select commentid, head_userid,
            tail_userid
      from comment_friends_closed_init w
          left join
                    comment_friends_closed
                    q using (commentid,
                    head_userid,
                    tail_userid)
     where q.commentid IS NULL;
```

Listing 38 SQL initialization phase for the Incremental PostgreSQL solution for Q2: initializing the *comment_friends* relation's closure.

```sql
WITH comment_components AS (
    SELECT commentid, head_userid AS
            userid
         , min(tail_userid) AS
               componentid
      FROM comment_friends_closed
     GROUP BY commentid, head_userid
)
, comment_component_sizes AS (
    SELECT cc.commentid,
            cc.componentid, count(*) AS
            component_size
      FROM comment_components cc
     GROUP BY cc.commentid,
            cc.componentid
)
-- consider all comments including
      those without likes
SELECT c.id AS commentid
     , coalesce( sum(
            power(ccs.component_size,
            2) ), 0) AS score
  FROM comments c left join
          comment_component_sizes ccs
          on (ccs.commentid = c.id)
 GROUP BY c.id, c.ts
 ORDER BY sum(
          power(ccs.component_size, 2) )
          DESC NULLS LAST
        , c.ts DESC LIMIT 3;
```

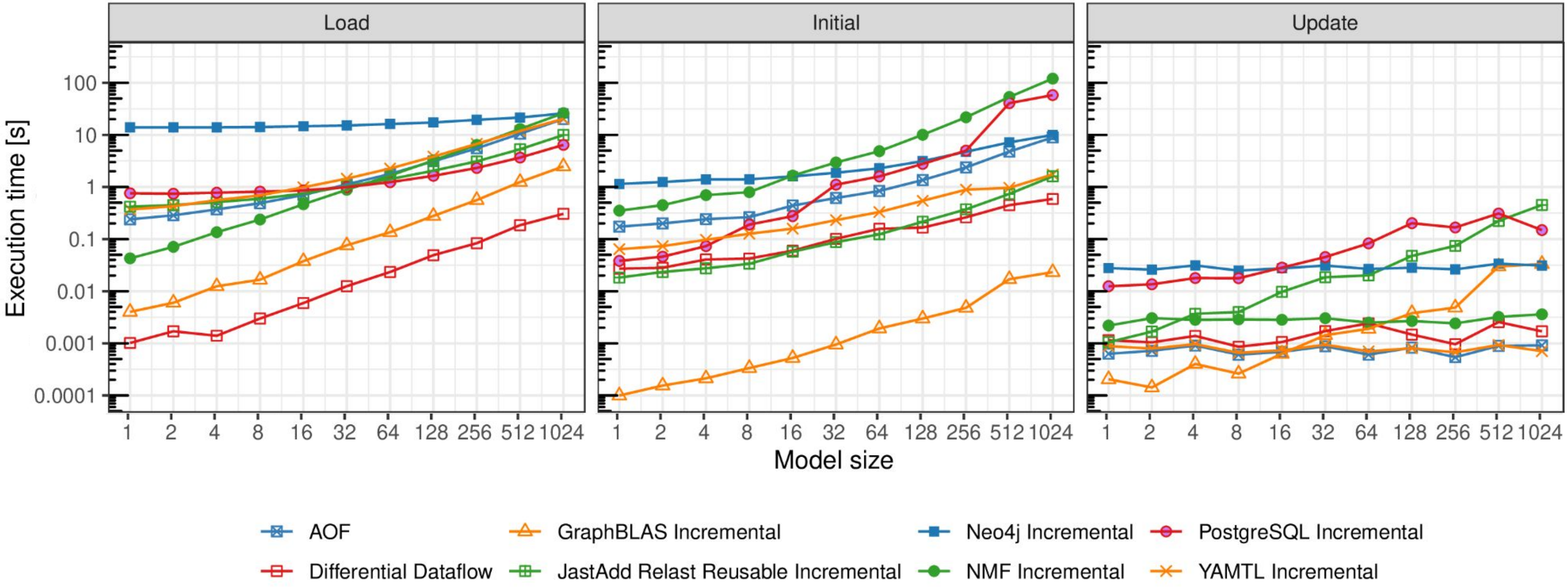
Listing 40 SQL result retrieval phase for the Incremental PostgreSQL solution for Q2.

```sql
WITH RECURSIVE -- note: though not the 1st query is
            the recursive one, the RECURSIVE keyword
            needs to be at the beginning
comment_friends_closed_stage0 AS (
    -- in order to maintain the transitive closure in
            comment_friends_closed
    -- we build on the transitive closure built so
            far and the new likes.
    -- We need the new likes because users that liked
            a specific comment
    -- are the nodes of the projected users graph for
            a comment
    SELECT commentid, head_userid, tail_userid
      FROM comment_friends_closed
  UNION ALL
    SELECT l.commentid, l.userid AS head_userid,
            l.userid AS tail_userid
      FROM likes_d l
)
, comment_friends_closed_stage1(commentid,
         head_userid, tail_userid) AS (
-- the transitive closure computed so far
         (reachability-only, no path is recorded)
-- is expanded by paths built from the new friendships
    SELECT commentid, head_userid, tail_userid
      FROM comment_friends_closed_stage0
  UNION
    SELECT cfc.commentid, cfc.head_userid, f.user2id
            as tail_userid
      FROM comment_friends_closed_stage1 cfc
         , comment_friends_d f
     WHERE cfc.tail_userid = f.user1id
       AND cfc.commentid = f.commentid
)
, comment_friends_closed_stage2 AS (
-- transitive closure having the new friendships is
         then expanded using the
-- previous transitive closure stage
      SELECT distinct cfc.commentid, cfc.head_userid,
              r.tail_userid
        FROM comment_friends_closed_stage1 cfc
             inner join comment_friends_closed r on
                       (cfc.tail_userid =
                       r.head_userid AND
                       cfc.commentid = r.commentid)
             -- LEFT JOIN and WHERE ... IS NULL is
                       the antijoin
             -- used to eliminate edges already
                       present in the previous closure
             -- this is to prevent unnecessary
                       CONFLICTs in the INSERT
                       statement below.
             left join comment_friends_closed s0 on
                       (cfc.commentid = s0.commentid
                       AND cfc.head_userid =
                       s0.head_userid AND
                       cfc.tail_userid =
                       s0.tail_userid)
       WHERE s0.commentid IS NULL
    UNION
      SELECT commentid, head_userid, tail_userid
        FROM comment_friends_closed_stage1
)
INSERT INTO comment_friends_closed(commentid,
         head_userid, tail_userid)
    select commentid, head_userid, tail_userid
      from comment_friends_closed_stage2 w
           left join comment_friends_closed q using
                   (commentid, head_userid,
                   tail_userid)
     where q.commentid IS NULL
ON CONFLICT DO NOTHING;
```

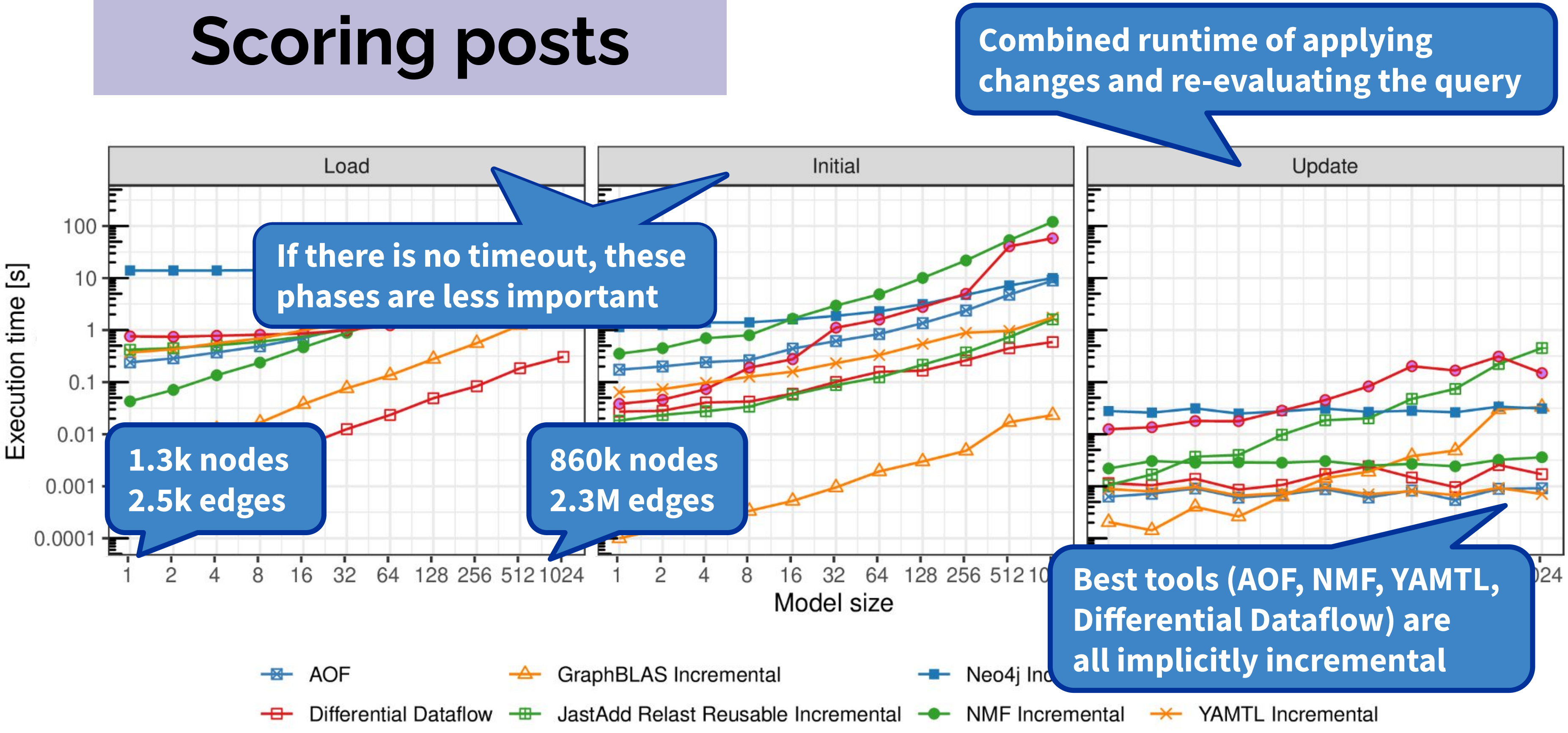
Listing 39 SQL maintenance phase for the Incremental PostgreSQL solution for Q2: updating the *comment_friends* relation's closure.

Results and findings

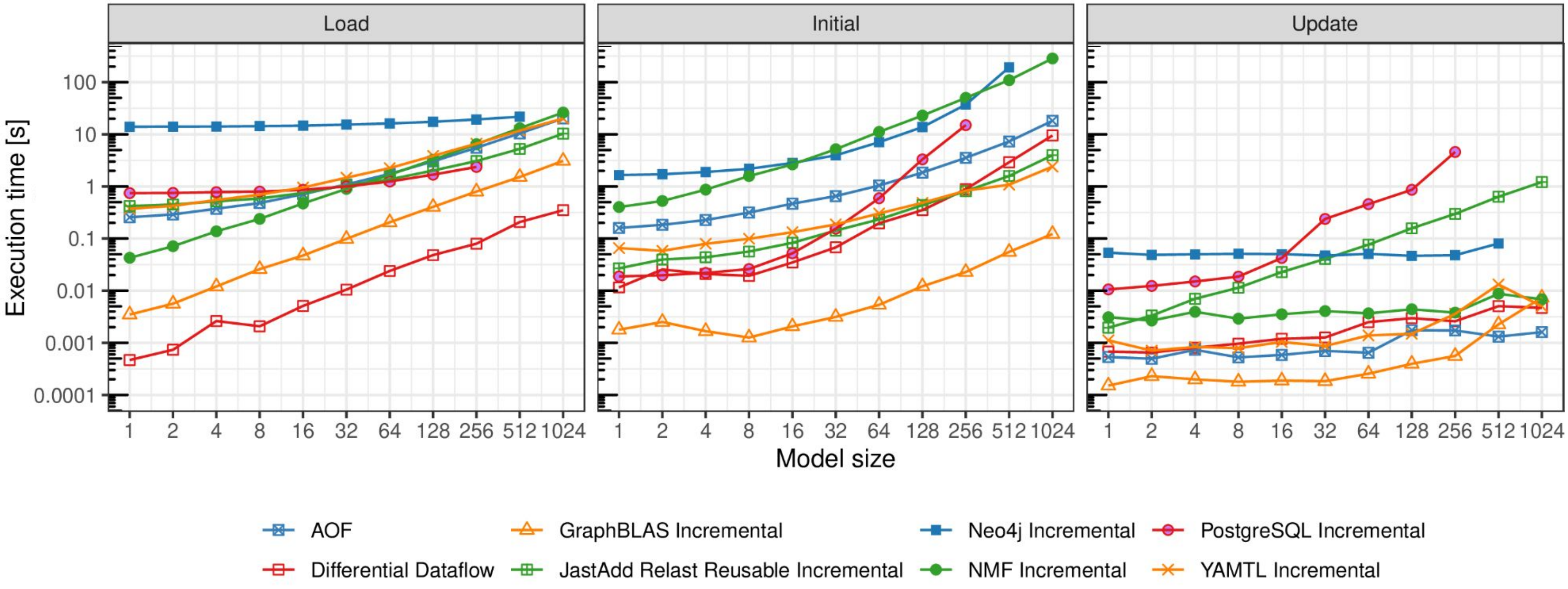
Scoring posts



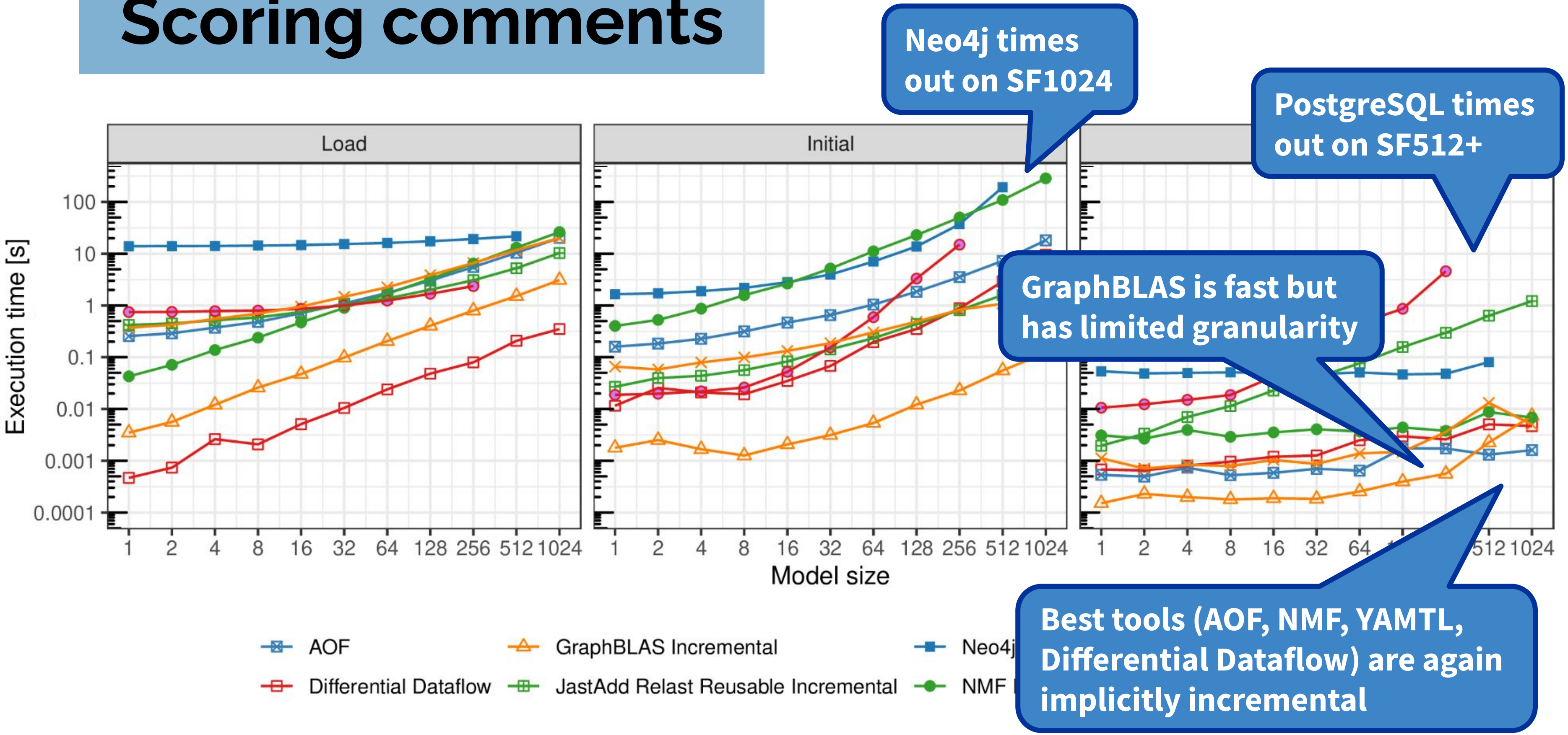
Scoring posts



Scoring comments



Scoring comments



Findings

1. Implicitly incremental tools are superior
2. Lacklustre performance from databases
3. Parallelization is not supported by EMF tools and databases
4. User-defined functions are important
5. Fair benchmarking and reproducibility are challenging

The Linked Data Benchmark Council

LDBC

The Linked Data Benchmark Council (LDBC) is a non-profit organization founded in 2012 with members from academia and industry. Its goals are:

1. Defining graph processing benchmarks
2. Facilitating fair competition
3. Accelerating the adoption of ISO GQL and SQL/PGQ

LDBC members

20 companies and organizations, including:

LDBC benchmarks

Social Network Benchmark:

- The data of the TTC 2018 “Social Media” case is a subset of the SNB.

The SNB has new and updated workloads:

- analytical: Business Intelligence workload v1
- transactional: Interactive workload v2

Rigorous auditing process that takes system costs (license and ownership) into account.

Conclusion

Conclusion

The TTC 2018 “Social Media” case:

- A cross-technology benchmark
- for incremental graph queries

Findings:

- Two simple graph queries can be challenging to formulate even non-incrementally
- DBMSs have performance issues for graph queries
- Explicit incremental evaluation is difficult
- Implicit incremental tools are superior
- → “retrofitted” incrementality has limited benefits

Ω

Finding: Lack of parallelization

Parallelization is paramount today: even laptop CPUs have 8–16 cores.

The initial evaluation is trivially parallelizable for both queries.

Observation:

- Only NMF, Differential Dataflow, and GraphBLAS support parallelization.
- EMF tools and databases (Neo4j, PostgreSQL) lack parallelization.

Finding: Importance of user-defined functions

Some computations are difficult to express in a declarative language, e.g. the connected components algorithm

User-defined functions (UDFs) can be used to express these computations

- Common among EMF tools – Java/Xtend code operating on the EMF model
- Database systems like Spark/Databricks and Snowflake support Java UDFs

Incremental maintenance of UDFs is difficult

Finding: Fair benchmarking is difficult

We are comparing very different systems:

- EMF tools
- Neo4j – graph DBMS
- PostgreSQL – relational DBMS
- GraphBLAS – concurrent sparse linear algebra library written in C
- Differential Dataflow – dataflow library written in Rust

Reproducibility is also difficult:

- dockerized execution
- extensive CI tests
- benchmarking in standard cloud VMs

Limitations of the benchmark

Limitations

No delete operations

- Adding them to the data generator is difficult (GRADES-NDA'20 paper)

No (unweighted) shortest path queries

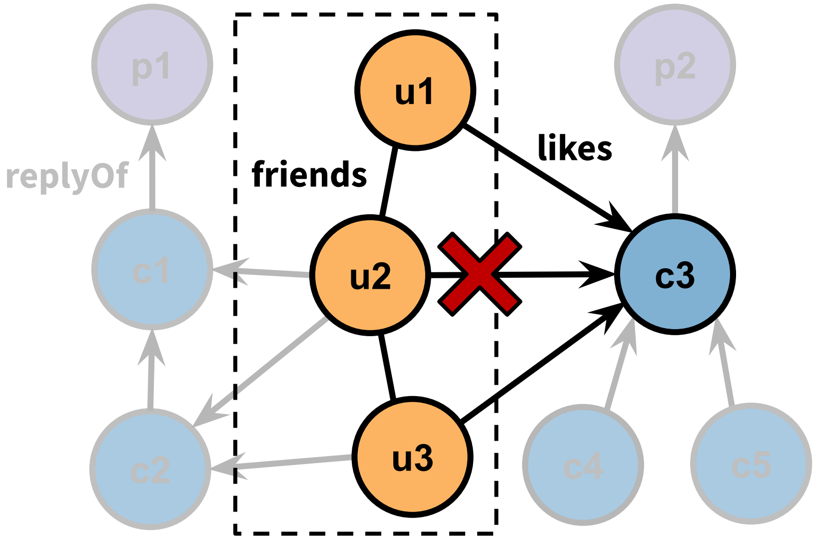
- Another important graph kernel, also challenging with deletes

See examples in the next slides.

Connected components with delete operations

Scoring comments

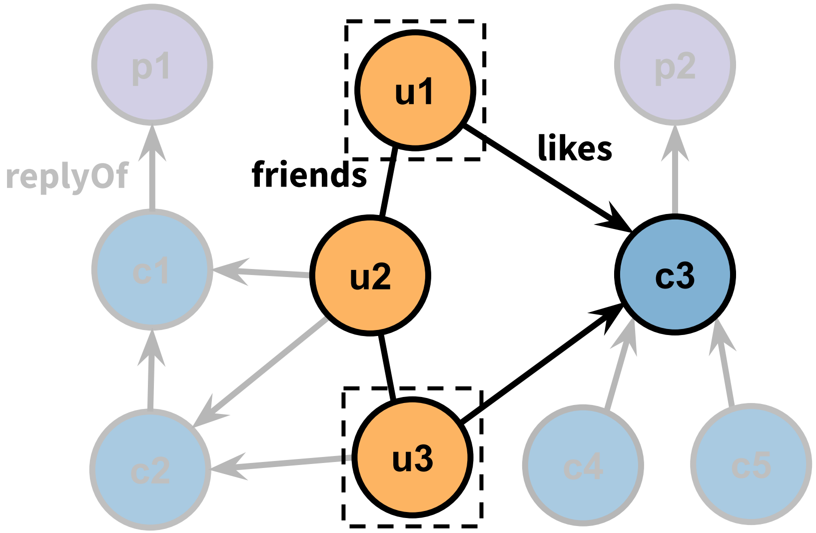
Score = Σ (component size)^2 = 3^2 = 9



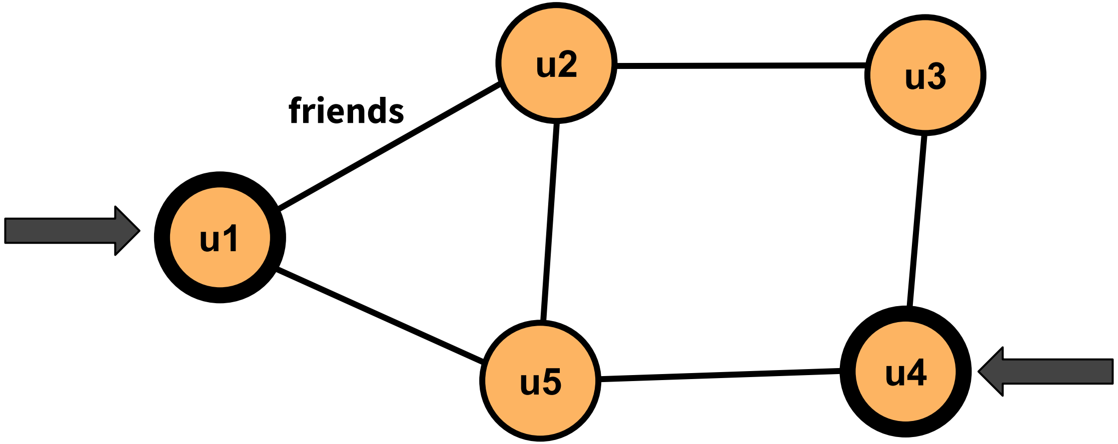
Connected components with delete operations

Scoring comments

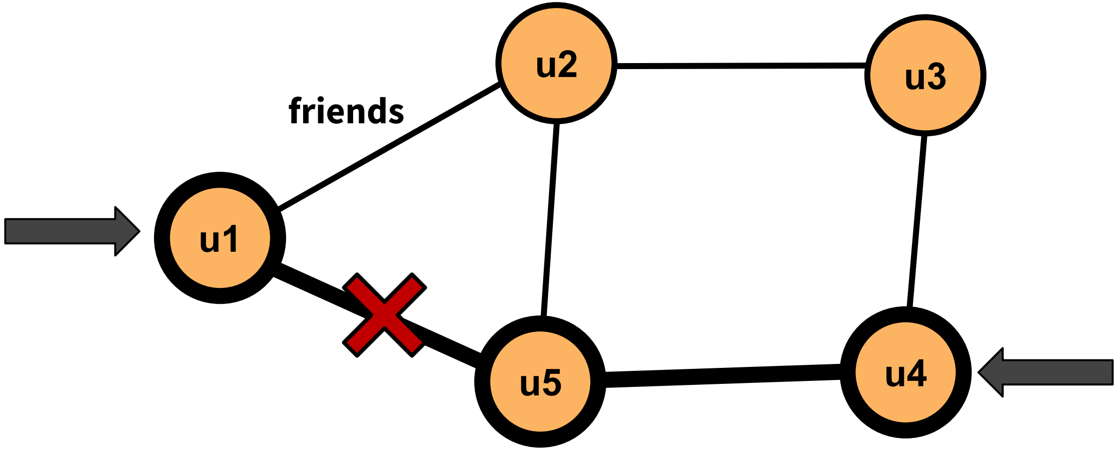
Score = Σ (component size)^2 = $1^2 + 1^2 = 2$



Shortest paths with delete operations

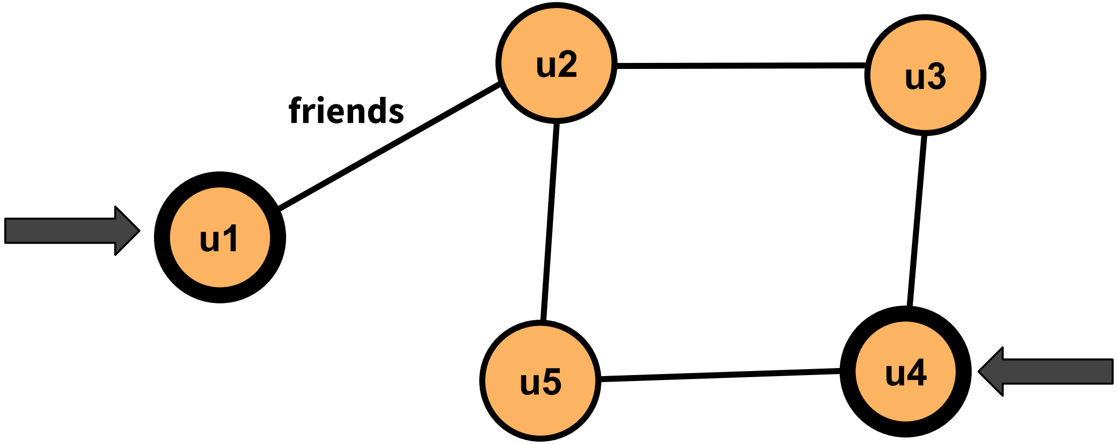


Shortest paths with delete operations



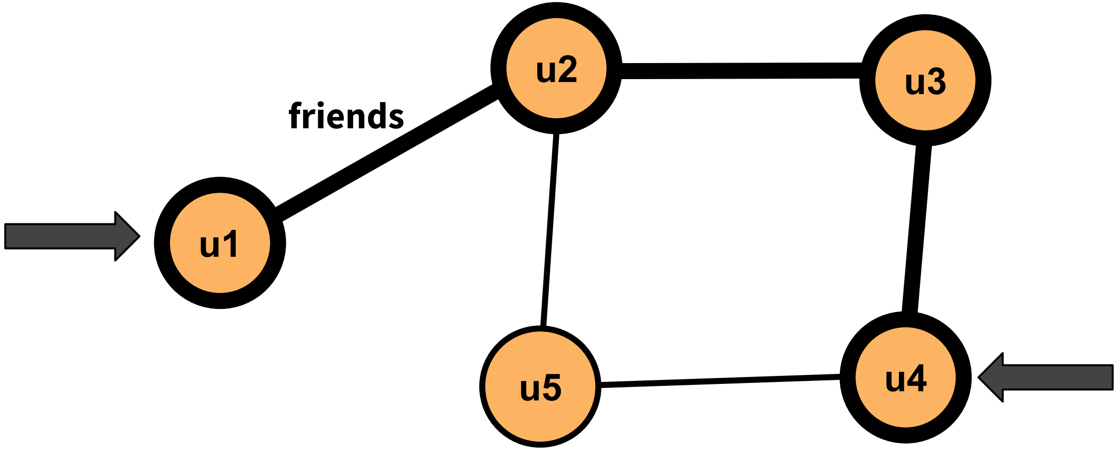
Shortest path: [u1, u5, u4]

Shortest paths with delete operations



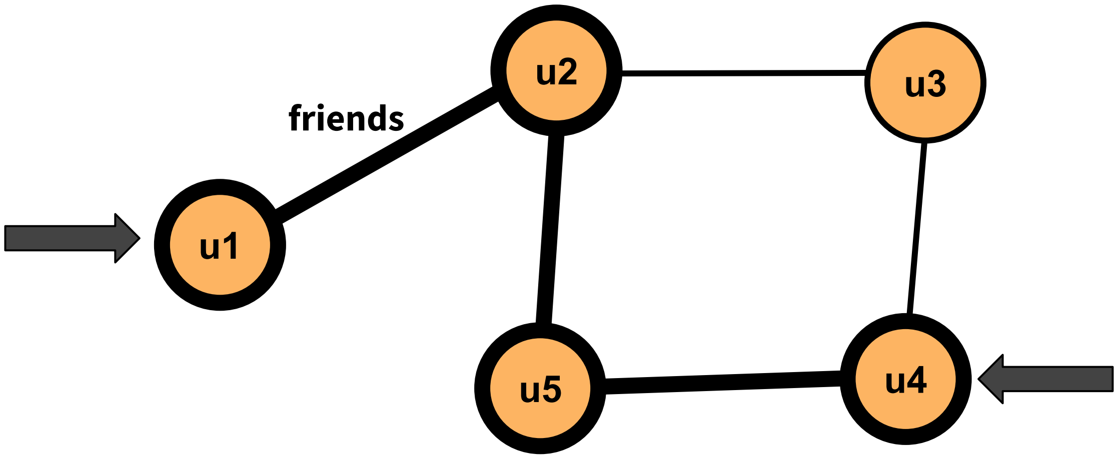
Shortest path: ?

Shortest paths with delete operations



Shortest path: [u1, u2, u3, u4]

Shortest paths with delete operations



Shortest path: [u1, u2, u3, u4]
[u1, u2, u5, u4]

Ideas for incremental evaluation // Future work

Ideas for incremental evaluation

The ideas in the following slides could work **if all inserts are added one-by-one** and there are not too many inserts. (There aren't, see the table with the model sizes.)

IIRC none of the solutions in the paper used this: they all went for a bulk insertion followed by a single recomputation step.

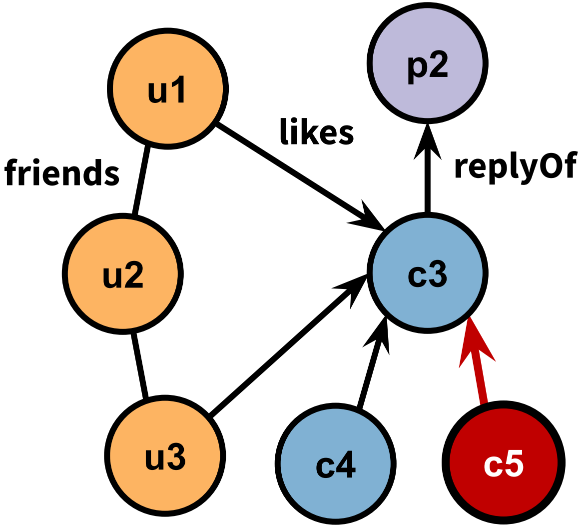
With a client-server setup, doing operations in bulk likely makes sense. Performing an individual maintenance operation per insert is likely expensive. With a read-oriented system (e.g. column store), it makes sense to perform the inserts in bulk.

Still, it would be interesting to give this a go with an embeddable database (e.g. DuckDB, Neo4j) or a system which provides an option to write stored procedure (like Oracle's PL/SQL).

Scoring posts

Score = 10 × #comments + #likes

Traversing the Submission tree



Trick: For each Comment, store its root Post. When inserting a new child Comment, it should get its parent's root Post.

This works because the subgraph is a tree and there are no cut-and-link operations.

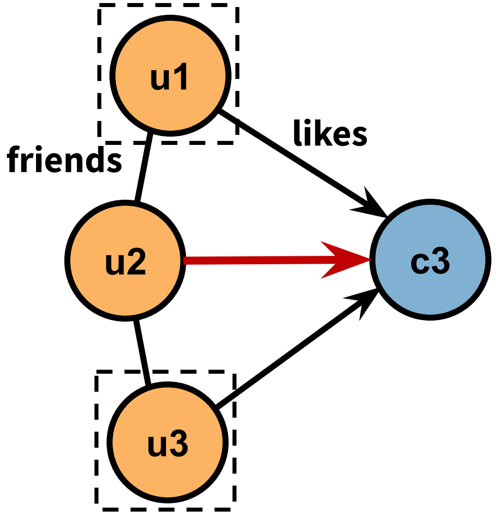
Trick: Upon adding a new "likes" or a new "friends" edge, connected components can only be merged together.

This works because there are no delete operations.

Scoring comments

Score = Σ (component size)^2

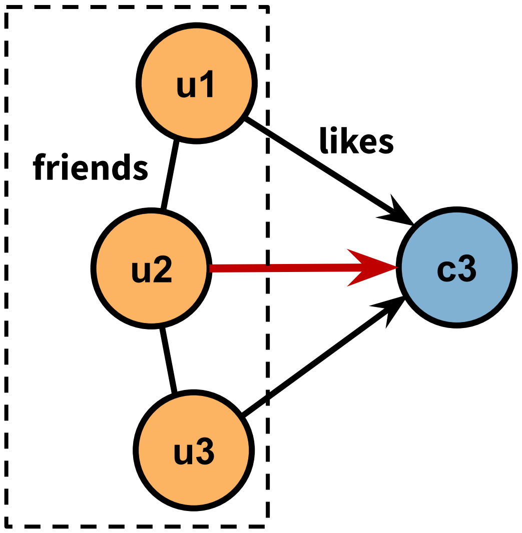
Finding connected components of Users



Scoring comments

Score = Σ (component size)^2

Finding connected components of Users



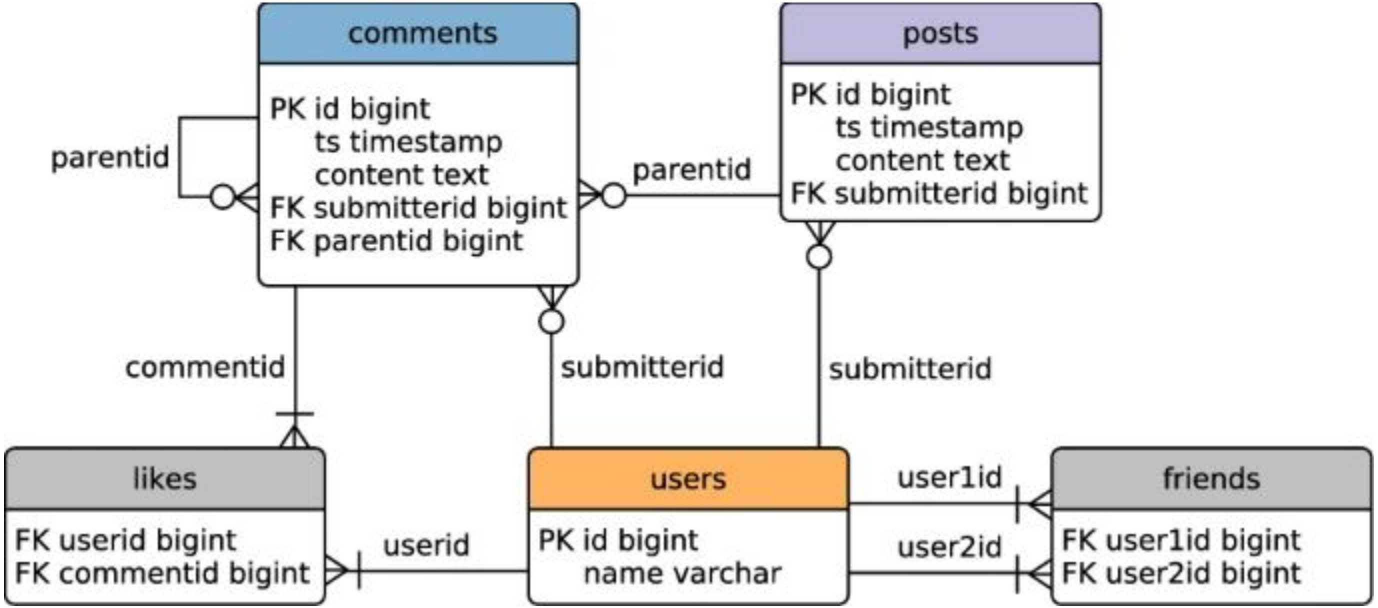
Trick: Upon adding a new "likes" or a new "friends" edge, connected components can only be merged together.

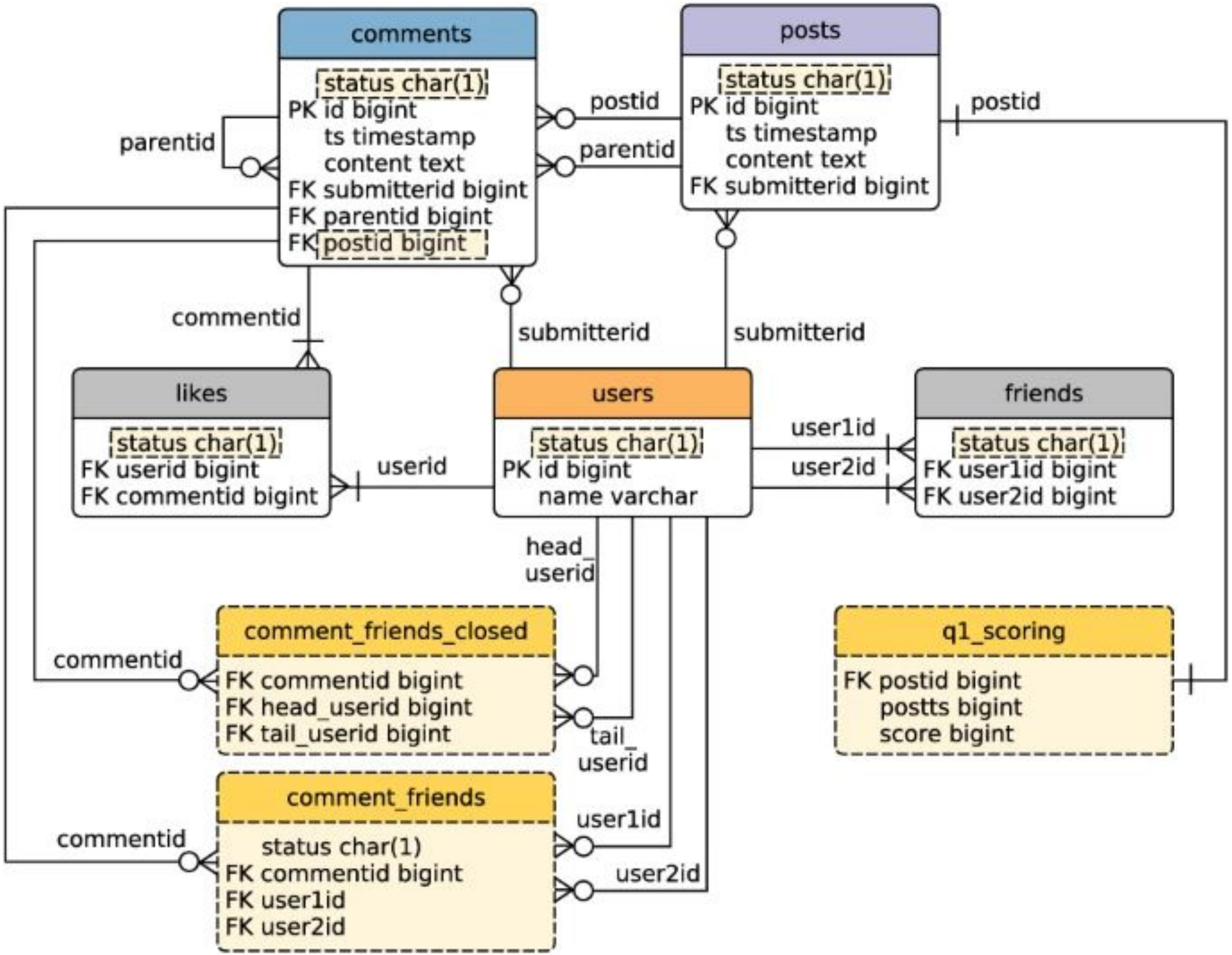
This works because there are no delete operations.

Potential extensions to the slide deck

- ~~concrete result slides~~
- ~~details on SQL/PGQ~~ ~~*WONTFIX*~~
- ~~model sizes~~
- ~~incremental query formulation~~
- interesting findings
- more info on concrete tools
  - mention of DD & videos
  - mention of GraphBLAS
  - Hawk, NMF, YAMTL, etc.
- anything on DuckDB/DuckPGQ as a potential tool for Q1
- ~~complaining that most MDE tools are single-threaded~~
- incremental tricks explained…

Incremental query formulation


comments
PK id bigint
ts timestamp
content text
FK submitterid bigint
FK parentid bigint
posts
PK id bigint
ts timestamp
content text
FK submitterid bigint
parentid
parentid
commentid
submitterid
submitterid
likes
FK userid bigint
FK commentid bigint
userid
users
PK id bigint
name varchar
user1id
user2id
friends
FK user1id bigint
FK user2id bigint


comments
status char(1)
PK id bigint
ts timestamp
content text
FK submitterid bigint
FK parentid bigint
FK postid bigint
posts
status char(1)
PK id bigint
ts timestamp
content text
FK submitterid bigint
postid
parentid
postid
parentid
commentid
submitterid
submitterid
likes
status char(1)
FK userid bigint
FK commentid bigint
userid
users
status char(1)
PK id bigint
name varchar
user1id
user2id
friends
status char(1)
FK user1id bigint
FK user2id bigint
head_
userid
comment_friends_closed
FK commentid bigint
FK head_userid bigint
FK tail_userid bigint
commentid
tail_
userid
q1_scoring
FK postid bigint
postts bigint
score bigint
comment_friends
status char(1)
FK commentid bigint
FK user1id
FK user2id
commentid
user1id
user2id

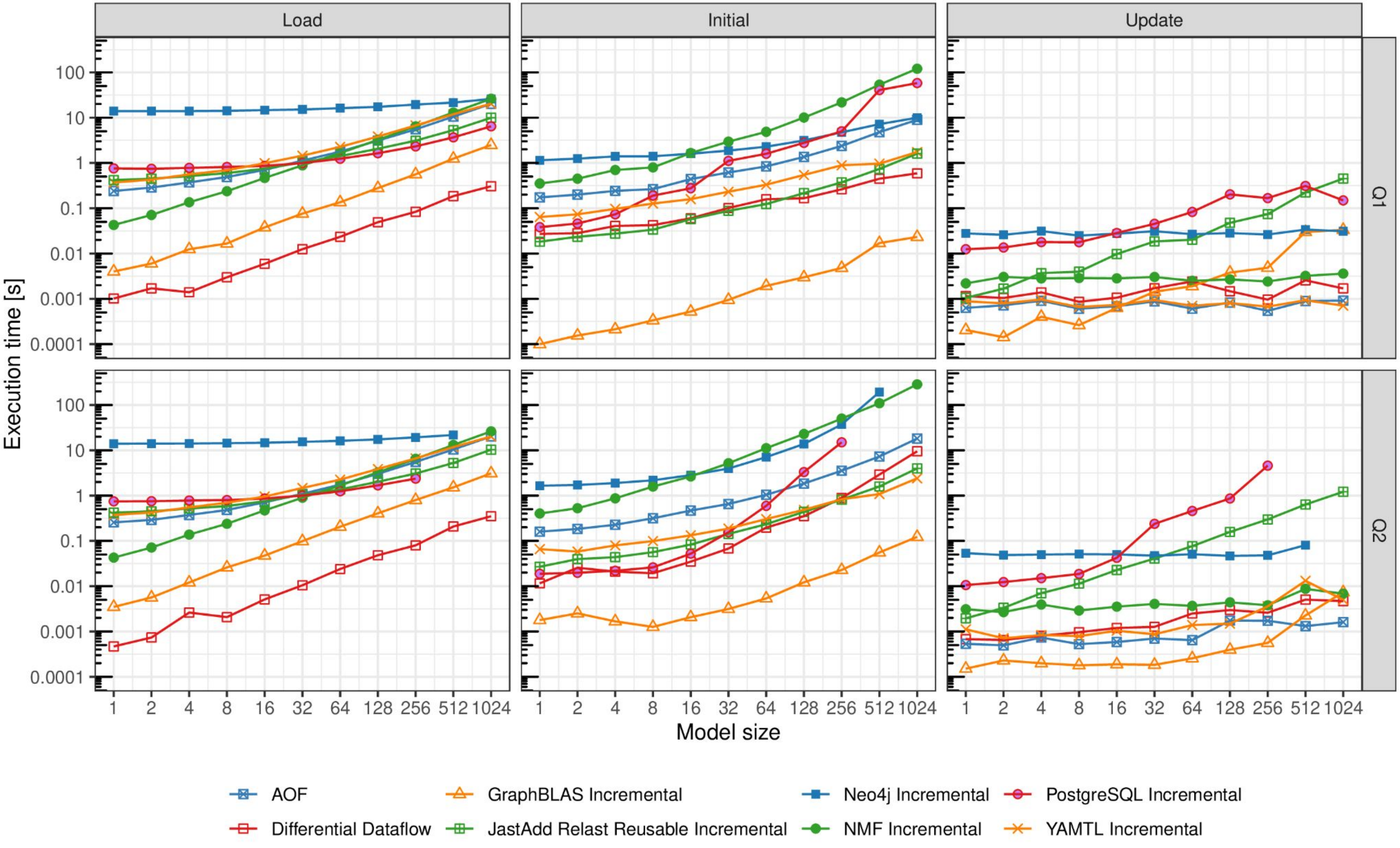
Incremental view maintenance

Categories:

- **Non-incremental:** query is recomputed each time
- **Implicitly implemental:** the maintenance is done automatically by the system
- **Explicitly incremental:** the query developer manually incrementalizes the query (poor man's view maintenance, can be retrofitted to existing systems)

Studied in depth in database research
…but most research focused on equijoins
…maybe anti- and outer joins

Transitive reachability (tree queries, connected components, etc.) are less studied.



Load
Initial
Update
Q1
Q2
Execution time [s]
Model size
100
10
1
0.1
0.01
0.001
0.0001
1
2
4
8
16
32
64
128
256
512
1024
AOF
Differential Dataflow
GraphBLAS Incremental
JastAdd Relast Reusable Incremental
Neo4j Incremental
NMF Incremental
PostgreSQL Incremental
YAMTL Incremental

All tools

(1) AOF
(2) ATL (3) + Incremental
(4) Differential Dataflow
(5) GraphBLAS (6) + Incremental
(7) Hawk (8) + IU (9) + IUQ
(10) JastAdd
(11) Neo4j (12) + Incremental
(13) NMF (14) + Incremental
(15) PostgreSQL (16) + Incremental
(17) Xtend
(18) YAMTL (19) + II (20) + EI

Most tools use the EMF data model

(1) Active Operations Framework (AOF)

(2) ATL (3) + Incremental

(7) Hawk (8) + IU (9) + IUQ

(10) JastAdd

(17) Xtend

(18) YAMTL (19) + II (20) + EI

NMF:

(13) NMF (14) + Incremental

Relational:

(4) Differential Dataflow

(15) PostgreSQL (16) + Incremental

Property graph:

(11) Neo4j (12) + Incremental

Matrix:

(5) GraphBLAS (6) + Incremental

Draft

a very small benchmark suite, just two queries and a few transformations

already highlights numerous usability and performance characteristics of systems !!

e.g. why don’t MDE tools use relational DBMSs

discuss the two queries briefly

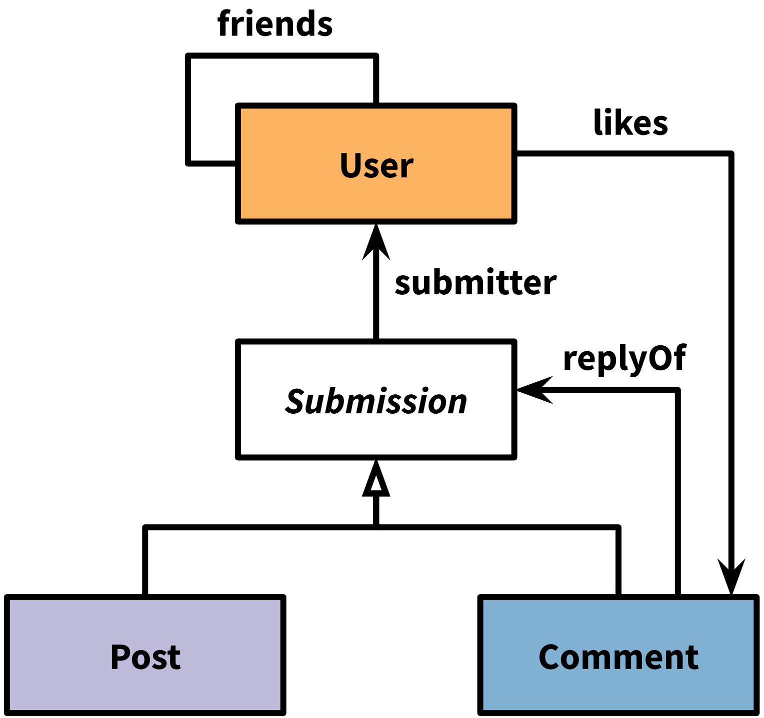
present a few solutions (e.g. Postgres/SQL; Neo4j/Cypher; MDE tools; differential dataflow, refer to Frank McSherry’s video)

FMS videos

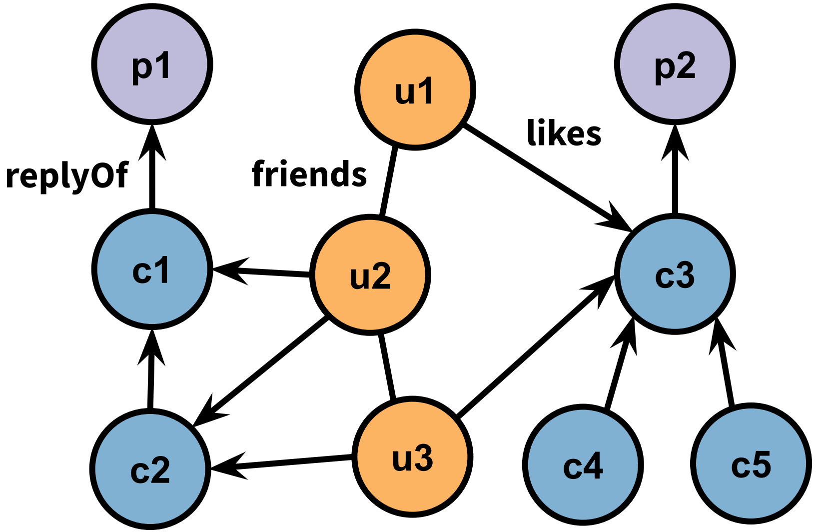
Live coding differential dataflow:

- https://www.youtube.com/watch?v=W6TKxS_pWr0
- https://www.youtube.com/watch?v=83rG471bmw8
- https://www.youtube.com/watch?v=uZ23MnpujNA

Schema



Instance



Differential dataflow: CC computation

```
likes                   // node    label   comment
    .filter(|_| false)
    .map(|(user, comm)| ((user.clone(), comm), user))
    .iterate(|labels| {

        let knows = knows.enter(&labels.scope());
        let likes = likes.enter(&labels.scope());

        labels
            .map(|((node, comment), label)| (node, (label, comment)))
            .join_map(&knows, |_node, (label, comment), dest| ((dest.clone(), comment.clone()), label.clone()))
            .concat(&likes.map(|(user, comm)| ((user.clone(), comm), user)))
            .reduce(|_key, input, output| {
                // only produce output, if `input` contains `_key.0`
                if input.iter().any(|(label,_wgt)| *label == &_key.0) {
                    output.push((input[0].0.clone(), 1));
                }
            })
    });
```

Table 2 Classification of approaches for Q2, sorted by tool and incrementality

Solution	Incremental conn. components	Algorithm	Sorting
AOF	⊗	Breadth-first traversal	Incremental
ATL	○	Depth-first traversal	Full
ATL incremental	⊗	Breadth-first traversal	Incremental
Differential Dataflow	⊗	Fixed-point label propagation	Full
GraphBLAS	○	FastSV	Offline top-x
GraphBLAS incremental	⊘	FastSV on overestimation + merge	Online top-x
Hawk	○	Tarjan	Full
Hawk (IU)	○	Tarjan	Full
Hawk (IUQ)	○	Tarjan	Online top-x
JastAdd	⊘	Depth-first/Kosaraju	Offline top-x
Neo4j	○	Union-find variant	Offline top-x
Neo4j incremental	⊗	Breadth-first traversal	Incremental
NMF reference	○	Tarjan	Full
NMF incremental	⊗	Edge changes	Incremental
PostgreSQL	○	Breadth-first traversal	Online top-x
PostgreSQL incremental	⊘	Overestimation + breadth-first traversal	Online top-x
Xtend	○	Tarjan	Online top-x
YAMTL-B	○	Weighted quick-union-find with path compression	Online top-x
YAMTL-II	⊗	Weighted quick-union-find with path compression	Online top-x
YAMTL-EI	⊗	Weighted quick-union-find with path compression	Online top-x

Notation—⊗ yes; ⊘ to some extent; ○ no. Overestimation means that all connected components are re-computed that might be affected by the changes

Table 1 Classification of approaches for Q1, ordered by solution name and support for incrementality

Solution	Incrementality	Sorting
AOF	⊗	Incremental
ATL	○	Full
ATL incremental	⊗	Incremental
Differential Dataflow	⊗	Full
GraphBLAS	○	Offline top-x
GraphBLAS incremental	⊗	Offline top-x
Hawk	⊘	Full
Hawk (IU)	⊘	Full
Hawk (IUQ)	⊘	Online top-x
JastAdd	⊘	Offline top-x
Neo4j	○	Offline top-x
Neo4j incremental	⊗	Incremental
NMF reference	○	Full
NMF incremental	⊗	Incremental
PostgreSQL	○	Online top-x
PostgreSQL incremental	⊗	Online top-x
Xtend	○	Online top-x
YAMTL-B	○	Online top-x
YAMTL-II	⊘	Online top-x
YAMTL-EI	⊗	Online top-x

Notation: ⊗ yes; ⊘ to some extent; ○ no

Table 4 Comparison of the tools used in the paper

Tool	Version	Data model	Engine	Solution	Decl.	Batch	Implicit	Explicit	DB	MV	Parallel
AOF	v201806	EMF	Java 8	Xtend	⊘	○	⊗	○	○	○	○
ATL	3.8.0	EMF	Java 8	ATL	⊗	⊗	○	○	○	○	○
ATL Incremental	v201904	EMF	Xtend	ATL/AOF	⊗	○	⊗	○	○	○	○
Differential Dataflow	0.11.0	relations	Rust	Rust	○	○	⊗	○	○	○	⊗
GraphBLAS	4.0.3	matrices	C	C++	○	⊗	○	⊗	○	○	⊗
Hawk	2.1.0	EMF	Java 8	EOL	⊘	⊗	⊘	⊗	⊗	⊘	○
JastAdd	2.3.5	EMF	Java 11	Java 11	○	⊗	⊗	○	○	○	○
Neo4j	4.2.4	property graph	Java 11	Java 11	⊗	⊗	○	⊗	⊗	⊗	○
NMF	2.0.169	NMF	C#	C#	⊗	⊗	⊗	○	○	○	⊗
PostgreSQL	12.4	relations	C	Java 11/SQL	⊗	⊗	○	⊗	⊗	⊗	○
Xtend	2.20.0	EMF	Java 8	Xtend	⊗	⊗	○	○	○	○	⊛
YAMTL	0.1.5	EMF	Java 11	Xtend	⊘	⊗	⊗	⊗	○	○	○

Data Model: the data model exposed to the user. *Engine:* programming language used to implement the engine (model transformation engine, database query engine, etc.), *Solution:* programming language and query language (if applicable) used to implement the solution, *Decl.:* the solution specified the queries using a declarative query language, *Batch:* only batch mode is supported, *Implicit:* implicit incrementalization is supported, *Explicit:* a solution with explicit incrementalization was implemented, *DB:* database-backed, i.e. the tool persists the model on disk after each transaction, *MV:* materialized views, *Parallel:* parallelization is supported. *Notation*—⊗ yes; ⊘ to some extent; ○ no; ⊛ yes, using Java 8 streams

Model sizes for each scale factor

Table 7 Model sizes for each scale factor: number of nodes and edges, number of changes

Type\scale factor	1	2	4	8	16	32	64	128	256	512	1024
Comments	640	1064	2315	5056	9220	18,872	39,212	76,735	148,470	273,418	540,905
Posts	554	889	1845	2270	5518	10,929	18,083	37,228	74,668	167,299	314,510
Users	80	118	190	204	394	595	781	1158	1678	2606	3699
Total number of nodes	1274	2071	4350	7530	15,132	30,396	58,076	115,121	224,816	443,323	859,114
friends	53	102	262	298	904	1827	2752	5695	11,118	24,387	45,386
replyTo	640	1064	2315	5056	9220	18,872	39,212	76,735	148,470	273,418	540,905
likes	6	24	66	129	572	1598	4770	13,374	36,815	102,276	268,432
submitter	1194	1953	4160	7326	14,738	29,801	57,295	113,963	223,138	440,717	855,415
Total number of edges	2533	4207	9118	17,865	34,654	70,970	143,241	286,502	568,011	1,114,216	2,251,043
Total number of changes	67	120	132	104	110	117	68	86	45	112	74