Graphflow: An Active Graph Database

Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, Semih Salihoglu
School of Computer Science, University of Waterloo
Waterloo, Ontario, Canada
chathura.kankanamge@uwaterloo.ca, s3sahu@uwaterloo.ca, amine.mhedhbi@uwaterloo.ca,
y522chen@uwaterloo.ca, semih.salihoglu@uwaterloo.ca

ABSTRACT

Many applications detect the emergence or deletion of certain subgraphs in their input graphs continuously. In order to evaluate such continuous subgraph queries, these applications resort to inefficient or highly specialized solutions because existing graph databases are passive systems that only support one-time subgraph queries. We demonstrate *Graphflow*, a prototype *active graph database* that evaluates general one-time and continuous subgraph queries. Graphflow supports the property graph data model and the Cypher++ query language, which extends Neo4j's declarative Cypher language with *subgraph-condition-action triggers*. At the core of Graphflow's query processor are two worst-case optimal join algorithms called Generic Join and our new Delta Generic Join algorithm for one-time and continuous subgraph queries, respectively.

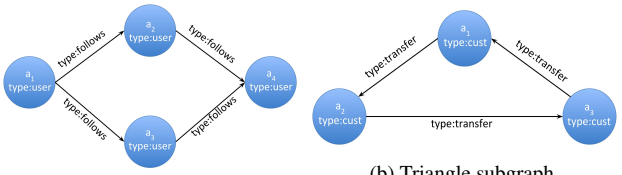
1. INTRODUCTION

Evaluating subgraph queries, i.e., finding instances of a given subgraph in a larger graph, is a fundamental computation performed by many applications that process graphs. Existing graph databases, such as Neo4j [5] and OrientDB [7] only support *one-time subgraph queries* that are evaluated on a snapshot of the graph until completion. However, many applications need to evaluate *continuous subgraph queries*, which detect the emergence or deletion of a given subgraph continuously. We give three examples.

Example 1: Twitter's MagicRecs recommendation application [4] continuously detects a *diamond* subgraph, shown in Figure 1a, in Twitter's who-follows-whom graph. The diamond subgraph consists of a user a_1 who follows two separate users a_2 and a_3, who both follow another user a_4. Once a diamond subgraph is detected, MagicRecs recommends user a_1 to follow user a_4, because two users that a_1 is following are interested in a_4.

Example 2: Consider a financial fraud detection application that has as input a transactions graph. In the graph, individual customers are vertices and an edge from two customers a_1 and a_2 indicates a transfer of money from a_1 to a_2. The application detects circular transactions, which may indicate money laundering activity, and reports them to a fraud detection specialist. This query would correspond to a continuous cycle query. The triangle query, shown in Figure 1b, is an example cycle query.

Example 3: Consider a datacenter monitoring application that has as input a job-host dependency graph, in which jobs and host machines are vertices. The hosts have a periodically updated `status` property that can be either `running` or `down`. An edge between two jobs j_1 and j_2 represents that j_1 depends on j_2 and an edge between a job j_1 and a host h_1 represents that j_1 runs on h_1. The monitoring application continuously detects and alerts the owners of jobs that are directly or indirectly running on a host that is `down`. This query would correspond to a continuous path query.



(a) Diamond subgraph. (b) Triangle subgraph.

Figure 1: Example subgraph queries.

Building these applications on top of existing *passive* graph databases would lead to inefficient solutions, such as periodically executing a one-time subgraph query and taking the differences in outputs. Alternatively, one can model graphs as relational tables, express subgraph queries as incrementally maintained views and use triggers of active relational database systems. Relational database triggers would simplify the development of these applications as the job of continuously detecting a subgraph is delegated to the database. However, using a relational data model and query language is less natural for graph applications than the graph-specific data models and languages of graph databases. The lack of continuous query support by existing graph databases has lead some applications to build efficient but highly specialized solutions. For example, MagicRecs is specialized only to detect the diamond subgraph. An active graph database that supports general continuous queries would be of immense use.

We demonstrate *Graphflow*, an active graph database that supports *subgraph-condition-action triggers* for this purpose. In Graphflow, applications express the subgraphs they want to detect continuously in a declarative fashion and an action they want to perform when the subgraph emerges or is deleted from the graph. Graphflow's user-facing components, data model, query language, as well as the system's data storage component is graph-specific. Internally, the system's query processor is based on a new worst-case join algorithm called *Generic Join* [6] and our new incremental view maintenance algorithm called *Delta Generic Join* [1].

2. GRAPHFLOW SYSTEM

Figure 2 shows the high-level architecture of Graphflow. Graphflow is a single node in-memory system implemented in Java. There are four main components of the system: (1) an in-memory property graph store; (2) the Cypher++ query language, which extends

SIGMOD'17, May 14-19, 2017, Chicago, IL, USA

DOI: http://dx.doi.org/10.1145/3035918.3056445

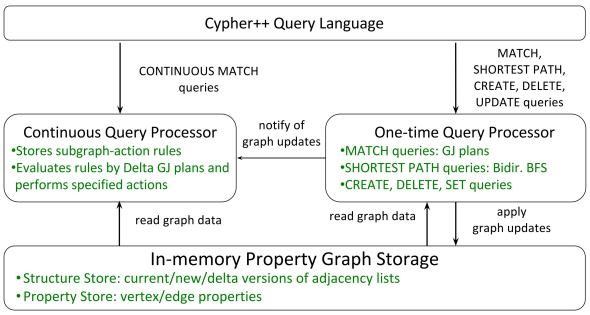


Figure 2: Graphflow architecture.

Neo4j's declarative Cypher language with subgraph-condition-action triggers; (3) a *One-time Query Processor* (`OQP`); and (4) a *Continuous Query Processor* (`CQP`). We next describe each component.

2.1 In-memory Property Graph Store

Graphflow's data model is a property graph, i.e., a labeled graph. Graphs are directed and vertices and edges can have arbitrary key-value properties on them. The graph is stored in two components.

Structure Store: Stores, for each vertex `u`, `u`'s incoming and outgoing adjacency lists stored as arrays. The adjacency lists are stored in sorted order of the outgoing neighbor IDs, which allows efficient intersections of two adjacency lists.

Property Store: Stores the key-value properties on vertices and edges. The keys can be arbitrary strings, and values can be integers, doubles, booleans, or strings.

2.2 One-time Subgraph Queries

One-time subgraph queries are expressed using original Cypher's MATCH clause. The following is the one-time query shown visually in Figure 1b that finds all of the triangles in the graph that are formed by `transfer` type edges:

MATCH a_1-[type='transfer']->a_2-[type='transfer']->a_3, a_3-[type='transfer']->a_1

The `type='transfer'` brackets specify a filter on the edges.

Any subgraph query Q on an input graph G can be seen in relational algebra as a multiway join on replicas of an `edges` table that contains all of the edges in G as (source ID, destination ID) tuples. For example, the triangle query, ignoring filters, is equivalent, in Datalog syntax, to the query:

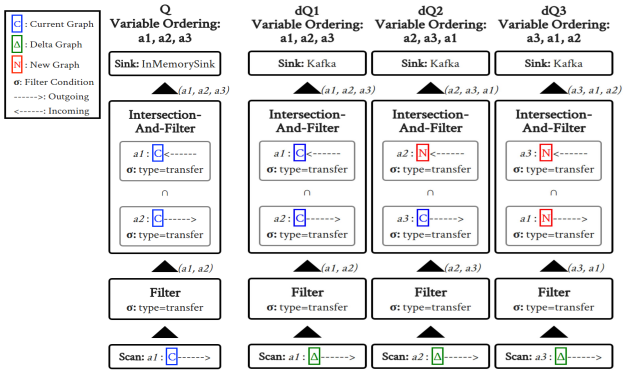
triangles (a_1, a_2, a_3) := $\text{edges}_1(a_1, a_2)$, $\text{edges}_2(a_2, a_3)$, $\text{edges}_3(a_3, a_1)$

We refer to the vertices in Q, e.g., a_1, a_2, and a_3, as *variables*. Graphflow evaluates the equivalent multiway join queries using Generic Join (GJ) plans, which we describe next.

2.2.1 Generic Join [6]

The traditional way of evaluating multiway join queries is using binary join plans, which join the input tables one at a time. This corresponds to evaluating subgraph queries by matching Q in G one *edge-at-a-time*. Recently, Ngo et. al [6] showed that for many queries any binary join plan can produce many more intermediate tuples than the maximum worst-case output of the query. Ngo et. al. then developed the first worst-case join algorithms, one of which is GJ. Unlike binary join plans, GJ evaluates queries using a *vertex-at-a-time* strategy. Let Q contain m variables, a_1, a_2, ..., a_m, and n edges. GJ consists of two steps:

- **Global Variable Ordering:** As a first step GJ orders the variables in Q arbitrarily. Any permutation of the variables can be picked. We assume for simplicity that a_1, ..., a_m is picked.



(a) GJ Plan. (b) DeltaGJ Plan.

Figure 3: Example GJ and DeltaGJ plans.

- **Iterative Prefix Subgraph Extension Step:** GJ iteratively evaluates a set of *prefix subqueries* Q_1, ..., Q_m in G, where Q_i is the subgraph query that includes only the first i variables in Q and only the `edges` tables between these variables in Q. At a high-level, GJ finds subgraphs in G that partially match Q and extends these subgraphs one vertex at a time. When computing Q_{i+1} from Q_i, GJ takes each $p = (a_1^*, a_2^*, ..., a_i^*)$ in Q_i, where a_i^* is the ID of a vertex in G. When extending p to tuples in Q_{i+1}, for each $\text{edges}_j(a_k, a_{i+1})$ table in Q_{i+1} where $a_k \in \{a_1, ..., a_i\}$, GJ takes vertex a_k^*'s outgoing adjacency list. Similarly for each $\text{edges}_j(a_{i+1}, a_k)$ table in Q_{i+1}, GJ takes vertex a_k^*'s incoming adjacency list. GJ then takes the Cartesian product of p with the intersection of these adjacency lists.

2.2.2 Generic Join Plans

Given a MATCH query Q, Graphflow constructs its logical GJ plan. The logical plan construction consists of picking an ordering of the variables in Q using heuristics, organizing the intersection operations that is consistent with the picked variable ordering, and deciding the placement of the filter operations in the query. Figure 3a shows an example plan for the triangle query in Graphflow's *Plan Viewer* (`PV`) interface. We describe the optimizations we have implemented in Graphflow for constructing GJ plans.

Sorted Adjacency Lists: As discussed in Section 2.1, Graphflow stores the adjacency lists in sorted order of neighbor IDs, and uses the *galloping* intersection technique [8] for fast intersections.

Variable Ordering Heuristics: We start with the highest degree variable in Q, breaking the ties arbitrarily. Then, we iteratively pick the variable that is adjacent to the highest number of already picked variables, breaking ties by the overall degree or arbitrarily if the degrees are equal. Although GJ's worst-case optimality is independent of the variable ordering, the ordering can have significant performance implications in practice, which we plan to demonstrate to the conference attendees.

Filter Placement: Any filters on a single vertex or edge in Q, e.g., `type='transfer'` on the edge (a_2, a_3), is executed during the intersection operation that involves the edge (a_2, a_3). This is done by the `Intersection-and-Filter` operation shown in Figure 3a. Any filter that accesses two or more properties is executed as a separate `Filter` operation that is pushed down to the first prefix subgraph it can be applied on.

2.3 Continuous Subgraph Queries

Continuous subgraph queries are expressed through subgraph-condition-action triggers using our CONTINUOUS MATCH clause extension to Cypher. The following is an example of a trigger that detects the directed triangles in a transactions graph and calls a

reportCircularTransfer UDF for each emerged triangle.

```
CONTINUOUS MATCH a1-[:type='transfer']->a2-[:type='transfer']->a3,
                 a3-[:type='transfer']->a1
ON EMERGENCE ACTION UDF reportCircularTransfer IN udf.jar
```

The other values for the ON clause are DELETION or ALL, which detect only the deletions or both the emergence and the deletions of the subgraph. Currently, the other ACTION value is FILE, which writes the subgraphs to a local file.

Updates, such as insertions or additions of edges, are one-time queries that arrive at OQP. Upon an update, OQP first temporarily modifies the in-memory graph store, which implicitly stores three different versions of the graph upon receiving an update:

- **Current**: Version of the graph prior to the update.
- **Delta**: Only the updates.
- **New**: New version of the graph after applying the updates.

OQP then notifies the CPQ of the updates, which checks for each registered trigger using DeltaGJ plans, if the subgraph specified in the trigger emerged or was deleted, and performs the action of the trigger. Once CQP is done detecting subgraphs, the updates become permanent in the graph. We next explain CQP's DeltaGJ plans.

2.3.1 Delta Generic Join Plans

DeltaGJ is a new incremental view maintenance algorithm we developed that evaluates a set of delta queries $dQ1$, ..., dQn for Q using GJ as the input tables change to maintain the result of Q. We note that, although the idea of delta queries is old and based on techniques from references [2, 3], our evaluation of dQi using GJ has performance implications that do not exist for existing IVM algorithms based on delta queries evaluated with binary join plans.

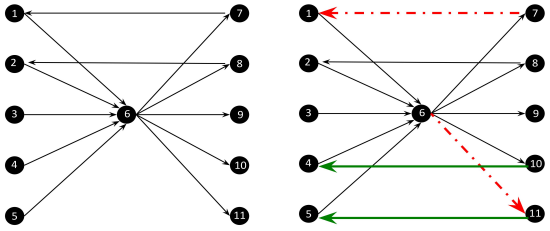
Upon an update, let c-edges, Δ-edges, and n-edges correspond to the edges in the Current, Delta and New graphs, respectively.[1] Figure 5 shows examples of these tables when the input graph in Figure 4a is updated to Figure 4b. The +/− signs indicate the inserted and deleted edges, which are also used to differentiate emerged and deleted subgraphs in the output. References [2, 3] have shown that the union of the following n queries gives the newly emerged tuples (or subgraphs) in Q due to updates.

$$dQ_1 := \Delta edges_1, c\text{-}edges_2, c\text{-}edges_3, ..., c\text{-}edges_n$$
$$dQ_2 := n\text{-}edges_1, \Delta edges_2, c\text{-}edges_3, ..., c\text{-}edges_n$$
$$...$$
$$dQ_n := n\text{-}edges_1, n\text{-}edges_2, n\text{-}edges_3, ..., \Delta edges_n$$

DeltaGJ evaluates each of these queries using GJ with the following important restriction. Note that in each dQ_i, there is exactly one copy of the Δ-edges table. Let a_i, a_j be the variables in that Δ-edges table. The variable ordering of GJ when evaluating dQ_i has to start with a_i, a_j or a_j, a_i. This corresponds to starting the evaluation from the vertices adjacent on the newly inserted or deleted edges when we match Q vertex at a time in G. As we show in our tech-report [1], when variables are ordered in this fashion, DeltaGJ is worst-case optimal under insertion-only workloads.

Graphflow's CQP produces the n logical plans for each query specified in each registered trigger. Figure 3b shows the three logical plans for the three delta queries when evaluating the triangle query. Notice that in each plan, the variables are ordered differently. Finally, note that for intersection operations, the plans include information about from which version of the graph (Current, Delta, or New) the adjacency lists should be read from.

[1]Updates are treated as the deletion of the previous version and the insertion of the new version of the edge or vertex.



(a) Input graph. (b) Updated input graph.

Figure 4: Example input graph and its updated version.

src	dst
1	2
1	6
2	6
2	8
3	6
4	6
5	6
6	7
6	8
6	9
6	10
6	11
7	1

(a) Current-edges.

src	dst	+/-
6	11	-
7	1	-
10	4	+
11	5	+

(b) Δedges.

src	dst
1	2
1	6
2	6
2	8
3	6
4	6
5	6
6	7
6	8
6	9
6	10
10	4
11	5

(c) New-edges.

Figure 5: Current-edges, Δedges, and New-edges tables.

Name	|V|	|E|	Description
Soc	75K	508K	Epinions.com who-trusts-whom graph
LJ	4.8K	69M	LiveJournal social graph
TW	42M	1.5B	Twitter who-follows-whom graph

Table 1: Graph datasets.

Graph	GF-Tr	Neo-Tr	GF-D	Neo-D	GF-P	Neo-P
Soc(4)	**0.2**	8	**0.8**	77	**0.4**	2.3
LJ(20)	**1.6**	57	**3.4**	150	**1.8**	24
TW(1000)	**7**	907	**18**	861	**57**	663

Table 2: One-time query experiments run-time results.

Graph	GF-Tr	Neo-Tr	GF-D	Neo-D	GF-P	Neo-P
Soc(4)	**101K**	1.1M	**5.5M**	14.5M	101K	101K
LJ(20)	**2.1M**	6.6M	**4.5M**	18.2M	2.1M	2.1M
TW(1000)	**1.2M**	3.5M	**2.3M**	3.5M	1.2M	1.2M

Table 3: One-time query experiments intermediate data results.

3. PERFORMANCE

We compared the performance of Graphflow against Neo4j on one-time queries and PostgreSQL delta query plans that use binary joins on continuous queries. Unlike Graphflow, both systems use edge-at-a-time strategies for evaluating subgraph queries. Table 1 shows the three real-world input graphs we used. We used a Linux machine with 256GB RAM and Intel E5-2670 processor. We note that our experiments cover a limited set of input graphs and queries. We leave an extensive performance study to future work.

For one-time queries, we used the triangle, diamond, and a simple path query outputting the two degree neighbors of each vertex. We added an integer group property, starting from 1, on the edges to make the queries faster. The maximum group value varied across experiments. We searched for only the subgraphs in which each edge had a group of 1. We ensured that Neo4j keeps the entire graph in its in-memory cache. For each query, we measured the query runtime and the intermediate number of subgraphs each system generates. Table 2 shows the results. In the table the Tr, D, and P suffixes indicate the triangle, diamond, and path queries, respectively. The numbers in parentheses indicate the number of groups we used in the input graph. As seen in the table, Graphflow generates up to 10.9x less intermediate data than Neo4j and is between 5.8x and 129x faster on our queries. We note that Graphflow generates less intermediate data than Neo4j as a consequence of its vertex-at-a-time strategy instead of Neo4j's edge-at-a-time

Exp.	GF-R	GF-T	GF-I	PSQL-R	PSQL-T	PSQL-I
Soc-Tr(4)	**0.008s**	**26K**	**2.6K**	1.2s	186	37.5K
Soc-D(4)	**0.13s**	**183K**	**42K**	3.3s	7K	245.1K
LJ-Tr(4)	**0.03s**	**21K**	**12.5K**	15s	42	96.5K
LJ-D(4)	**4.5s**	**125K**	**85.4K**	134s	447	723.8K

Table 4: Continuous query experiments.

strategy. This accounts for part of our runtime efficiency. However, our runtime efficiency is also due to two other important factors: (1) Graphflow is a prototype system, which is inherently more efficient as it supports fewer features; and (2) instead of Neo4j's linked lists storing Java objects, our graph store is backed by Java primitive type arrays, which are faster in lookups.

For continuous queries, we used the triangle and diamond queries. In PostgreSQL, we stored a `c-edges`, a Δ`-edges`, and an `n-edges` table. We put indices on both the `srcID` and `dstID` columns of each table. We used Linux's tmpfs file system to store PostgreSQL's data in memory. We loaded a random 20% of the edges in the graph to each system, using `c-edges` and `n-edges` tables in PostgresSQL. Then we issued a batch of updates containing `B` edges, 75% of which are random insertions from the remaining 80% of the original graph edges, and 25% of which are random deletions. In PostgreSQL we inserted these `B` edges to Δ`-edges` and applied them to `n-edges` tables (actually deleting the deleted edges). We let Graphflow's `CQP` to process the batch using DeltaGJ plans. In PostgresQL, we used manual stored procedures to evaluate the same delta queries, which in PostgreSQL are executed with binary join plans. We issued batches until 2M edges (or all remaining 80% edges) are inserted and measured the runtime and throughput, i.e., detected subgraphs per second, of each system and the intermediate data generated by each system. The runtimes omit data indexing time, which was more efficient in Graphflow. The results are shown in Table 4. We omit experiments with Twitter because PostgreSQL was taking a prohibitively long time to load and complete batches. The "R","T", and "I" suffixes indicate the average runtime, throughput, and intermediate data per batch, respectively. Batch sizes were 10K for `Soc` and 100K for `LJ`. Graphflow generates up to 14.4x less intermediate data than PostgreSQL's binary join plans and is between 25x to 500x faster on our queries. We note again that while our intermediate data generation is due to differences between DeltaGJ and binary join plans, our runtime efficiency is also due to other factors we outlined above.

4. DEMONSTRATION SCENARIOS

Our demonstration illustrates three aspects of Graphflow: (1) an overview of the system; (2) the ease of development of a Graphflow application; and (3) Graphflow's GJ and DeltaGJ plans.

4.1 Graphflow Overview

Our first scenario aims to give an overview of an active graph database and the continuous graph applications that can be built on it. Prior to the conference, we will prepare a graph database that contains as nodes, the authors of SIGMOD 2017 papers, and as edges, their Twitter followers. Nodes will have optional email addresses as a property. We will build two applications:

- **MagicRecs:** Continuously detects diamonds and sends follower recommendations as emails to the attendees who visit our demo.
- **SIGMOD Cliques:** Continuously finds a clique of 4 users and shows the *most recent SIGMOD cliques* on our monitor.

We will grow our preloaded database continuously by adding each attendee that visits our demo. We will insert the attendee's Twitter username and email address to a browser UI, which will add the attendee and her followers to Graphflow using Twitter's API. Upon this insertion, our application will send her an email based on the diamonds she participates in. If she is part of a 4-clique, our monitor will refresh to show her clique as the most recent SIGMOD clique. We will determine and ask her to follow certain users that lead to new diamonds that she is part of. She will get new recommendations soon after she follows those users.

Throughout our interaction, we will give the attendee an overview of our graph store, Cypher++ language, and query processors.

4.2 Graphflow Application Development

Our next scenario demonstrates the ease of application development with Graphflow. The attendee will develop from scratch the financial fraud detection application from Section 1. A typical development cycle will be as follows:

1. Using Graphflow's console, the attendee will create a transactions graph that contains customer nodes and transaction edges.
2. The attendee will write a UDF in Java that processes cycles. In the UDF, the attendee can choose among possible actions to take, including printing the cycle on the screen, sending emails or text messages to someone, or issuing further queries to Graphflow to delete a transaction in the cycle.
3. The attendee will finally write the Cypher++ trigger to detect the emergence of a cycle of a small size (e.g., 3 or 4) and as action call the UDF she implemented in the previous step.

Once the application is built, we will test the application as follows. Prior to the conference, we will build a UI that contains a synthetic transactions graph. Initially all of the edges will be colored gray indicating the edge has not been inserted to the database. When the attendee clicks on an edge, the edge will be added to the database and turn green. Once the attendee forms a cycle, she will observe that the action she implemented is taken.

4.3 GJ and DeltaGJ Plans

The goal of this scenario is to demonstrate Graphflow's GJ and DeltaGJ query plans. The attendee will write a one-time or continuous subgraph query and see Graphflow's default GJ or DeltaGJ plan for the query visually on `PV` (as in Figure 3). We will explain the different operators, graph versions, and variable orderings in the plan. To demonstrate the effects of variable orderings, the attendee will type a new ordering on `PV` that will override Graphflow's default plan and generate a new plan. The attendee will execute different plans on one of the real-world graphs from Table 1 by clicking a button on `PV`. For continuous queries, edges of the graph will be streamed at the background using a script we have prepared. The attendee will observe the performances of different plans through the statistics on `PV` about run-time, intermediate prefixes generated, and throughput. By executing a fixed plan on different scale inputs, the attendee will also test Graphflow's scalability.

5. REFERENCES

[1] K. Ammar, F. McSherry, and S. Salihoglu. Delta Generic Join. Technical report, University of Waterloo, 2016. https://cs.uwaterloo.ca/~ssalihog/papers/deltagj.pdf.

[2] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. *SIGMOD Rec.*, 15(2), 1986.

[3] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. *SIGMOD Rec.*, 22(2), 1993.

[4] P. Gupta, V. Satuluri, A. Grewal, S. Gurumurthy, V. Zhabiuk, Q. Li, and J. Lin. Real-time Twitter Recommendation: Online Motif Detection in Large Dynamic Graphs. *VLDB*, 7(13), 2014.

[5] Neo4j. https://neo4j.com/.

[6] H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: New developments in the theory of join algorithms. *SIGMOD Rec.*, 42(4), 2014.

[7] OrientDB. http://orientdb.com/orientdb/.

[8] T. L. Veldhuizen. Leapfrog Triejoin: A worst-case optimal join algorithm. *CoRR*, 2012.