Degree Distribution Optimization in Historical Graphs

Alexandros Spitalas, Charilaos Kapeletiotis, and Kostas Tsichlas

University of Patras, Rion, Patra, Greece.
a.spitalas@upatras.gr,kapeletiotis@ceid.upatras.gr,ktsichlas@ceid.upatras.gr

Abstract. In recent years, maintaining the history of graphs has become increasingly imperative due to the emergence of related applications in several fields, like health services, social interactions, and map guidance. Historical graphs focus on being able to store and query the whole graph evolution and not just the latest instance. Various frameworks have been used to store these graphs and query them. In this paper, we look at how an evolving historical graph can be stored in the distributed SQL database CockroachDB focusing on OLAP queries. In particular, we study an OLAP query (time-interval degree distribution) as to how it can be better executed in a distributed environment to improve its efficiency. To this end, we provide different approaches for this particular query that have different trade-offs between local computation and network latency. Finally, we provide experimental evidence concerning these trade-offs.

Keywords: temporal graphs · graph management systems · query engine · CockroachDB

1 Introduction

In recent years there has been a rapid increase in time-evolving networks, producing a considerable amount of data. Networks, such as citation networks, traffic networks, and social networks are, naturally represented as graphs, and they are usually dynamic. For example, in a citation network, new nodes and edges are constantly added due to the publication of new papers. An important challenge that arises in these time-evolving networks is the efficient management of their history to be able to reason about its whole evolution and not only about its latest state. This allows us to answer queries such as "What is the average connectivity of author X in the citation network between 2010 and 2015?".

There have been a lot of systems developed since 2016 for historical graph management. Most of them are distributed since evolving large graphs are extremely demanding concerning space usage and query/update time. A rather outdated related survey can be found in [10]. They focus mainly on the models used for temporal graphs and the techniques available to query them. Another recent survey is [2] (2021) which analyzes graph streaming systems, where the differences and similarities between graph streaming systems and historical graph systems are explicitly given. In general, graph streaming systems tend to use snapshots as the stable (latest) instance of the graph, since it may be the case that recent updates have not been registered. However, in principle, these snapshots may be stored and allow for historical queries as well. Some graph streaming systems explicitly - although it is controversial to what extent - support historical queries on such (small number) snapshots of their evolution.

1.1 Previous Work

In our previous work [5] we have constructed a vertex-centric storage optimal model for storing historical graphs, while in [4] and [8], we have created proof-of-concept prototypes of HiNode storage model. The implementations use as a storage backend Cassandra and MongoDB, both are distributed NoSQL databases, the first being a wide-column store while the later a document-based database. In the above works, we used three Snapshot-based datasets to benchmark their performance in storing and querying Historical Graphs using HiNode model.

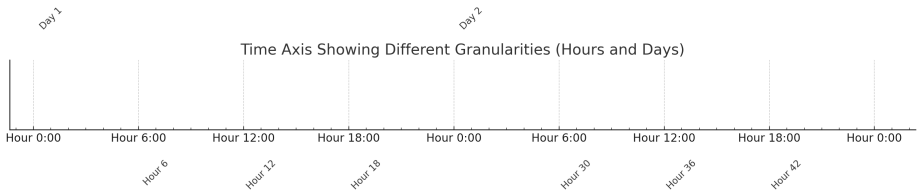


Fig. 1: Observing Different Granularities, in Days, Hours, or groups of 6 Hours

About historical graph queries, defining what constitutes a historical graph query has been studied by some researchers and can be categorized into time-dependent and time-independent algorithms. For certain algorithmic problems, new directions have emerged by incorporating the dimension of time. For example, in [6], algorithms were implemented for historical shortest path, earliest arrival time, and shortest path fastest path, while for temporal metrics, they provided temporal betweenness and temporal closeness. Similarly, in [9], the shortest path problem was divided into the Earliest Arrival path problem, Latest Departure path problem, and Shortest Duration Path problem. Lastly, [3, 7] discuss granularities in time instances. In our research, we need to define historical One Hop and degree distribution queries alongside the granularity of time instances as used in these queries.

1.2 Our Contribution

In the current work, we have implemented the vertex-centric HiNode model using the Distributed-SQL database CockroachDB as one more possible storage system that can be used for HiNode model. The new implementation is both distributed and supports strongly-consistent ACID transactions. We compare the behavior and performance of the new storage system in temporal graph tasks, compared to the MongoDB implementation. Moving one step further, we propose one new Algorithm to optimize the Historical Degree Distribution query. The proposed algorithm, manages to be more distributed and have less memory requirements in the master node. As a result having also better performance in a distributed environment, especially in in bigger datasets or tasks.

The rest of the paper is structured as follows. In Section 2 we show how we store the historical graph in CockroachDB. In Section 3 we discuss the different approaches to answering the query, and in Section 4 we provide our experimental findings. Finally, we conclude in Section 5.

2 CockroachDB-Distributed SQL implementation

In this implementation, we aim to embed a Distributed SQL database into the HiNode model. These databases are designed from the ground up, to operate as a single distributed relational database, which enables the database data to be replicated to all nodes in the cluster. The utilization of a Distributed SQL database ensures full support for ACID transactions. Furthermore, non-relational databases such as MongoDB, offer schema validation. However, the database itself does not impose referential integrity or strict restrictions on data types, often leading to data integrity issues. Based on these remarks, a distributed SQL database such as CockroachDB, appears to be a reasonable option, particularly in circumstances where the overall reliability of the data across the system is of vital importance.

As expected, this implementation diverges from previous ones, regarding the models that were constructed, which were reconfigured to be consistent with

the relational logic of the database. Table 1 shows the structure of the SingleTable model. One variation on previous implementations of this model that has emerged is that if we want to store multiple values for an attribute, they will be stored in nested JSONB format. In addition, since CockroachDB does not currently support the storage of multiple JSONBs in an array/list, the following modification was considered to be more appropriate: each record in the table represents one edge of the graph. The above structure resulted in the primary key of this table, being the row id set by the database. As a result, in CockroachDB, the SingleTable Model is edge-centric, and one record does not store the whole history of the node as in [5]. Regarding the MultipleTable model,

Record	
id	STRING
start	STRING
end	STRING
label	STRING
attributes	JSONB
edge	JSONB

Table 1: Structure of the SingleTable model

(Table 2), it was deemed more appropriate for each attribute related to a vertex to be stored in a single table, as opposed to the logic of the previous implementations, where each attribute was stored in a separate table. Our goal is to eliminate the obligation for each dataset to adhere to a specific schema.

vertices	
vid (pkey)	STRING
vstart (pkey)	STRING
vend	STRING
vlabel	STRING

attributes	
aid	STRING
alabel	STRING
attribute	JSONB

edges	
sourceid (pkey)	STRING
targetid (pkey)	STRING
estart (pkey)	STRING
eend	STRING
label	STRING
weight	STRING

Table 2: Structure of the MultipleTable

To meet the performance of the required stab queries, columns concerning the time instances were indexed. Regarding the SingleTable model generalized

inverted indexes were utilized, through which the JSONB fields were indexed, as well as in the MultipleTable model secondary indexes were applied.

3 Historical Query Approaches

In snapshot-based graphs, time instances are finite and well-defined, with each time instance straightforwardly defined as a snapshot of the graph. In Time-based graphs, theoretically, we have an infinite number of time instances, resulting in different approaches for storing and querying them. For example, if object time is measured in milliseconds, we can reduce granularity by storing seconds instead, thereby reducing detail and losing some information. One issue with this approach is whether to store an object that is alive for a fraction of a second; this can be handled in various ways. Some prefer to store it, if it is alive for even a millisecond, effectively taking the union of every involved time instance. Others invoke the law of the majority, counting it as alive only if it is alive for most instances within the hyper-time instance. Another approach assigns a weight to the hyper-time instance based on the percentage of time the object is alive. While all methods lose some information, they simplify storing and querying. Alternatively, one could store values without reducing granularity and delay this issue until query time. This lazy approach, though potentially less optimal for storage and querying, preserves all object details and defers the granularity decision to the query level.

We now proceed with defining historical queries. A general definition could be queries that involve more than one-time instance, regardless of their handling. Historical queries can be distinguished in at least two ways. Some involve multiple time instances but request results for each instance independently, such as querying degree distribution in every instance over a time period. Others also involve multiple instances but integrate time as part of the query, such as querying changes in degrees across neighborhoods or the causality of these changes over time. The first type can be handled independently for each period or more efficiently by leveraging the graph's structure in the database. The second type cannot be handled independently; it can be approached as an evolving graph in a streaming environment or by utilizing the full history stored in historical graphs for increased accuracy or efficiency. Additionally, we can delay granularity choices until query time.

For benchmarking, we test the first type of queries, such as querying degree distribution and one hop over a time period, requesting results for each time instance within this period. For time instances, we reduce granularity to yearly and use a union for the aliveness of objects.

3.1 Optimized Degree Distribution for Historical Graphs

Regarding OLAP queries, we have focused on the Historical Degree Distribution Query, how to optimize it, and how this can be applied in multiple SELECT queries in a Historical Graph Database. For the Historical Degree Distribution

Query, given a time interval (estart, eend), we outline the process of calculating the degree distribution at every time instance within this period. This can be broken down into two problems: determining the query to be executed in the database and handling the incoming data afterward.

The straightforward way to implement the Degree Distribution Query is to first retrieve all edges available between the period (estart, eend), then sum the appropriate edges during the intervals they are alive, and finally interpolate those results to find the degree distribution at each time instance. This approach is illustrated in Listing 1.1.

```
SELECT
  sourceid, estart, eend
FROM edges
WHERE
  DATE(eend) >= ts AND DATE(estart) <= te
```

Listing 1.1: Previous approach

```
 row     sourceid      estart       eend
-------------------------------------------------------
row 1 ->     1,      2010-12-21, 2011-03-20
row 2 ->     1,      2010-12-21, 2011-04-21
row 3 ->     1,      2010-12-24, 2012-12-31
row 4 ->     1,      2010-02-04, 2010-12-25
row 5 ->     2,      2010-02-04, 2012-12-31
```

However, using the Listing 1.1, most computational parts of the operation will be executed on one master node, with only a portion distributed across other nodes. To improve this, we had to find ways to reduce memory usage on individual nodes and divide the work among multiple workers. In Listing 1.2, we implemented a new approach for the query by first grouping the edges based on the source vertex and the interval they are alive. At a later stage, we count the different edges a source vertex has during the corresponding intervals. This can also be implemented using a Thinking Like a Vertex (TLV) technique, where each node counts its own edges at different time intervals and returns the result. The issue arising here is overlapping intervals in the results, as these cannot be handled internally within the select query. The above will be handled in the master node, although it is a much smaller task than the one from Listing 1.1.

```
SELECT
  sourceid,
  COUNT(targetid),
  EXTRACT(YEAR FROM DATE(estart))::int AS start,
  least(EXTRACT(YEAR FROM DATE(eend))::int, EXTRACT(YEAR FROM
     DATE(te))::int)::int AS end
FROM edges
WHERE DATE(eend) >= ts AND DATE(estart) <= te
GROUP BY
  sourceid,
  EXTRACT(YEAR FROM DATE(estart)),
```

```
  EXTRACT(YEAR FROM DATE(eend))
ORDER BY sourceid ASC
```

Listing 1.2: Present approach

```
 row     sourceid  count    start   end
 ------------------------------------------------------------
 row 1 ->     1,       2,     2010,   2011
 row 2 ->     1,       1,     2010,   2012
 row 3 ->     1,       1,     2010,   2010
 row 4 ->     2,       1,     2010,   2012
```

Algorithm 1 Fetch All: Degree Distribution using query 1.1

```
1: while rows.hasNext() do                                  ▷ full table scan
2:     id, degree, start, end ← row.id, row.degree, row.start, row.end
3:     for year from start to end do
4:         vertexDistribution[id][year]+ +          ▷ main difference with Algorithm 2
5:     end for
6: end while

7: for each v in vertexDistribution do
8:     for year, degree in v do
9:         degreeDistribution[year][degree]+ +
10:    end for
11: end for
```

Algorithm 2 More Memo: Degree Distribution using query 1.2

```
1: while rows.hasNext() do                                  ▷ reduced rows
2:     id, degree, start, end ← row.id, row.degree, row.start, row.end
3:     for year from start to end do
4:         vertexDistribution[id][year] += degree
5:     end for
6: end while

7: for each v in vertexDistribution do
8:     for year, degree in v do
9:         degreeDistribution[year][degree]+ +
10:    end for
11: end for
```

Algorithm 1 (Fetch All) is the only one that utilizes Listing 1.1. The data returned from the database are stored in a map, which maintains information about all the nodes of the graph. This map is then traversed to calculate the

Algorithm 3 Less Memo: Optimized Degree Distribution using query 1.2

```
1: previd ← NULL
2: while rows.hasNext() do                          ▷ reduced rows and memory
3:     newid, degree, start, end ← row.id, row.degree, row.start, row.end
4:     if newid ≠ previd  &&  previd ≠ NULL then
5:         for k, v in vertexDistribution do
6:             degreeDistr[k][v] + +
7:         end for
8:         vertexDistribution ← empty map[int]int
9:     end if

10:    for year from start to end do
11:        vertexDistribution[year] += degree
12:    end for

13:    previd ← newid
14: end while
```

degree distribution. Likewise, Algorithm 2 (More Memo) is algorithmically no different from the previous one, the only variation being that Algorithm 2 uses Listing 1.2 and is therefore adapted to it. However the above approach leads to excess memory allocation at runtime, hence we implemented Algorithm 3 (Less Memo) which is optimized with regard to the memory utilization and uses Listing 1.2. The main idea is that the degree distribution can be progressively computed. In this approach, the map that stores the data returned from the database maintains information only about the current node. Note that to accomplish this, the returned data from the database have to be sorted.

4 Experimental Results

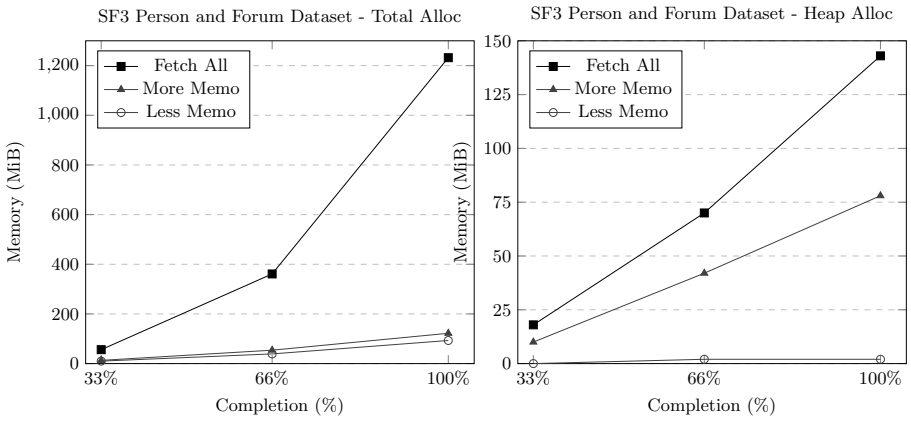
We benchmark the new CockroachDB implementation using a historical graph dataset generated with LDBC [1], adjusted to be event-based, and used in a streaming environment during dataset insertion. From this, we created three types of datasets, all featuring Persons as nodes and Person-Knows-Person relationships, with the largest dataset also including Forums and Forum-has-Member-Person relationships. In these datasets, we aim to test four aspects of the database: transaction throughput, memory usage, global query efficiency and distributed performance. To assess transaction throughput, we measure the time taken to insert each dataset into the database, including insertions, modifications, and deletions (without actually changing or deleting objects but retaining their history). For global queries, we test the Historical Degree Distribution by executing it six times, averaging the times while excluding the first execution to avoid a cold start (so taking the average of five executions). We also test the distributed capabilities of CockroachDB, while also testing our proposed Algorithms for Historical Degree Distribution Query. Finally, we compare the

memory usage of the resulting database for each dataset. Most experiments are omitted from this section and can be found in A

Dataset Statistics:

- SF3
  - Total number of nodes: 25870
  - Total number of edges: 668430
- SF10
  - Total number of nodes: 60800
  - Total number of edges: 2304951
- SF3 Person and Forum
  - Total number of nodes: 285499
  - Total number of edges: 10499492

4.1 Memory Requirements Experiments



(a) Total Alloc (b) Heap Alloc

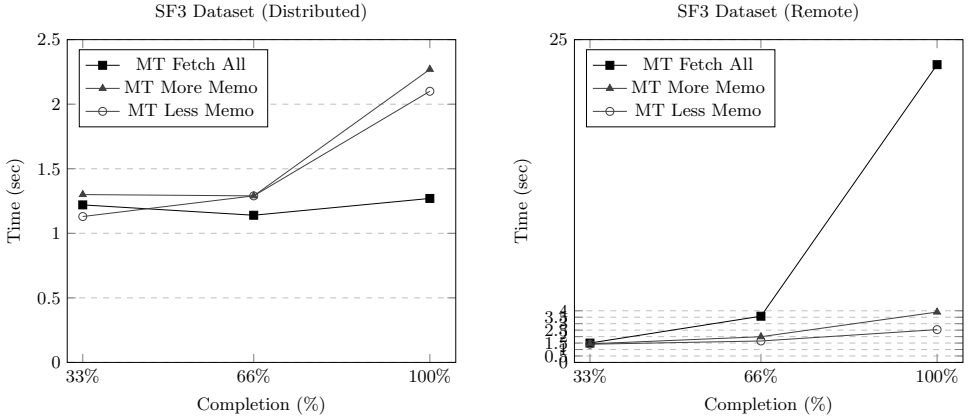
Fig. 2: SF3 Person and Forum Dataset

Regarding the memory requirements of the previously discussed algorithms, the MT and ST algorithms yield identical results. Therefore, to reduce complexity, we compare the results of the MT model exclusively in Figures 12, 13 and 2. The figures also distinguish between total memory requirements and heap memory requirements for each algorithm. Notable observations include:

- Across all three datasets, the Total Alloc Less Memo algorithm proves superior, offering an 18% to 25% reduction in time compared to the More Memo algorithm, and a 62.5% to 92.5% reduction in time compared to the Fetch All algorithm. This equates to up to 13.25 times less memory usage.

– When comparing Heap Alloc results, the More Memo algorithm reduces memory usage by 14% to 20%, 25% to 36.84%, and 40% to 45.4% in the SF3, SF10, and SF3 Person and Forum datasets, respectively, compared to Fetch All. The Less Memo algorithm further improves these reductions, achieving a 57.14% to 80%, 89.4% to 89.66%, and 97.14% to 98.6% reduction in memory requirements compared to Fetch All across all three datasets.
– The Less Memo algorithm not only reduces heap memory requirements but also enhances its performance on that matter as the dataset size increases.
– The analysis indicates that both More Memo and Less Memo consistently reduce memory usage compared to Fetch All. Less Memo generally achieves the highest percentage reduction across all datasets and years, demonstrating the most significant memory savings.

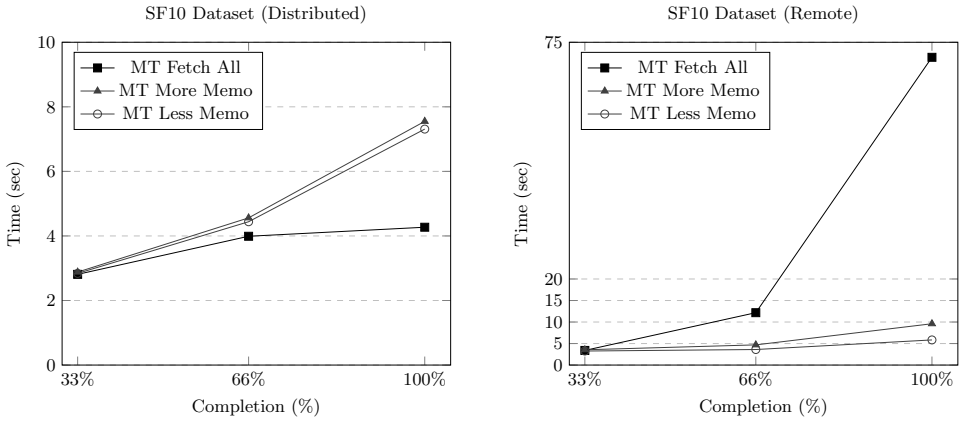
4.2 Distributed Experiments



(a) SF3 Dataset (Distributed) (b) SF3 Dataset (Remote)

Fig. 3: Comparison of Models and Algorithms for SF3 Dataset in distributed and remote clusters

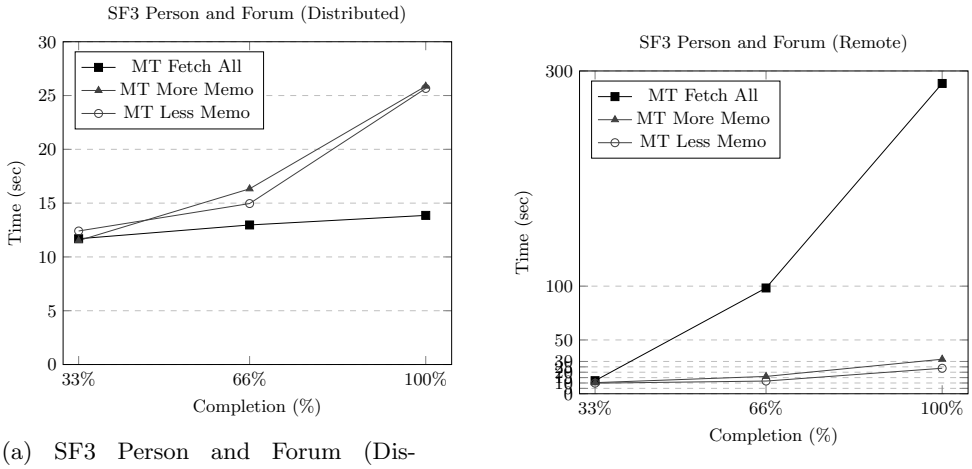
In Figures 3, 4 and 5, we compare the three algorithms of Degree Distribution query in a Distributed environment using CockroachDB. The cluster consists of 4 nodes, all in the same LAN with 10Gbps connection between them and minimized latency. We try two types of experiments, named Distributed and Remote Distributed, the basic difference between them is the location of the user that queries the Cluster. In the first one, the user is in the same LAN with the cluster, while in the second the user is in a remote location increasing the impact of I/Os. We reach the following conclusions from the experimental results:



(a) SF10 Dataset (Distributed)

(b) SF10 Dataset (Remote)

Fig. 4: Comparison of Models and Algorithms for SF10 Dataset in distributed and remote clusters



(a) SF3 Person and Forum (Distributed)

(b) SF3 Person and Forum (Remote)

Fig. 5: Comparison of Models and Algorithms for SF3 Person and Forum Dataset in distributed and remote clusters

- Comparing the MT More Memo and the MT Less Memo Algorithms we can observe the following. From Figures 3a, 4a and 5a, in a Distributed environment they have similar performance with less than 10% difference. On the contrary, when they are queried from a Remote location, we can observe in Figures 3b, 4b and 5b, that MT Less Memo is the best-performing model reducing the time from 26%, up to 39% depending the dataset and the percentage used in the query.
- Comparing the Fetch All Algorithm to the other two, we can observe an immense difference due to the impacts of I/Os. In the Figures 3a, 4a and 5a, depicting the Distributed experiments, we note that the best performing Algorithm is Fetch All, when used in 66% and 100% of the graph, managing to have up to 45% less time compared to More Memo and Less Memo Algorithms. Alternatively, in Figures 3b, 4b and 5b with the Remote Distributed experiments, we observe that the best-performing model is Less Memo, with a massive reduction of time up to 91% when used in the 100% of the graph.
- The best-performing model overall depends mostly on the impact of I/Os. In use-cases, that I/Os are valueless and the CPU power of each node in the cluster is similar to the machine making the query, then Fetch All is the Algorithm of choice. On the contrary, when I/Os are expensive, or the nodes of the cluster are significantly better than the machine querying, or when it's avoided to run expensive queries locally (e.g., in IoT applications), the Algorithm of choice is Less Memo.

5 Conclusions

In this paper, we provide a new implementation of the HiNode system using the Distributed-SQL Database CockroachDB and propose a new algorithm to query Historical Degree Distribution in a node-centric Historical Graph. For the first part, the new approach is best suited for applications where data reliability is crucial. For the second part, the new approach distributes query tasks and minimizes communication costs, making it ideal for applications where I/Os are expensive, CPU-intensive operations shouldn't run locally, memory specifications are strict, or distributed machines have significantly more performance than the local machine. We intend to extend the implementation of HiNode into a system named t-MAGMA while supporting multiple storage-backend choices for different applications. We also aim to improve the performance optimization of queries and generalize them for various types of queries.

Acknowledgments. This research was supported by the Hellenic Foundation for Research and Innovation (H.F.R.I.) under the "2nd Call for H.F.R.I. Research Projects to support Faculty Members & Researchers HFRI PhD Fellowship grant".

A Appendix A

A.1 System specifications

The following measurements were made on the same machine, the characteristics of which will be listed below. In addition, the queries for these benchmarks are performed on a single-node cluster, rather than a distributed one.

CPU	AMD Ryzen™ 5 7600 6 Cores 3.8 GHz Base Clock
RAM	Kingston Fury Beast 16GB DDR5 6000 MT/s
SSD	Kingston KC3000 PCIe 4.0 NVMe M.2 7,000MB/s Read, 6,000MB/s Write

A.2 Transaction Performance

As shown in Table 3, MongoDB clearly demonstrates significantly better performance in importing data. The three datasets used, SF3, SF10, and SF3 Person and Forum are respectively 75.5 MB, 245.1 MB, and 1.1 GB. For the given datasets MongoDB, compared to this specific implementation in CockroachDB, on the SingleTable model demonstrated respectively 91%, 90%, and 82% faster data insertion. As for the MultipleTable model, the insertion in MongoDB is 91%, 90%, and 91% faster. This happens as CockroachDB has to maintain SQL and ACID properties also with much replication, while MongoDB is a NoSQL database more focused on performance.

Database	Model	SF3	SF10	SF3 Person and Forum
CockroachDB	SingleTable	107 min	310 min	1773 min
	MultipleTable	23 min	77 min	258 min
MongoDB	SingleTable	9 min	31 min	311 min
	MultipleTable	2 min	7 min	23 min

Table 3: Data insertion

A.3 Comparing Historical Degree Distribution Query in MongoDB and CockroachDB

To guarantee consistency in the comparison between CockroachDB and MongoDB, the algorithm used for CockroachDB (Algorithm 1) is the same algorithm used in the corresponding implementation with MongoDB. Additionally, the query that was utilized is as similar as possible, considering the architectural differences of the databases. In the following, Algorithm 1 which uses Query 1

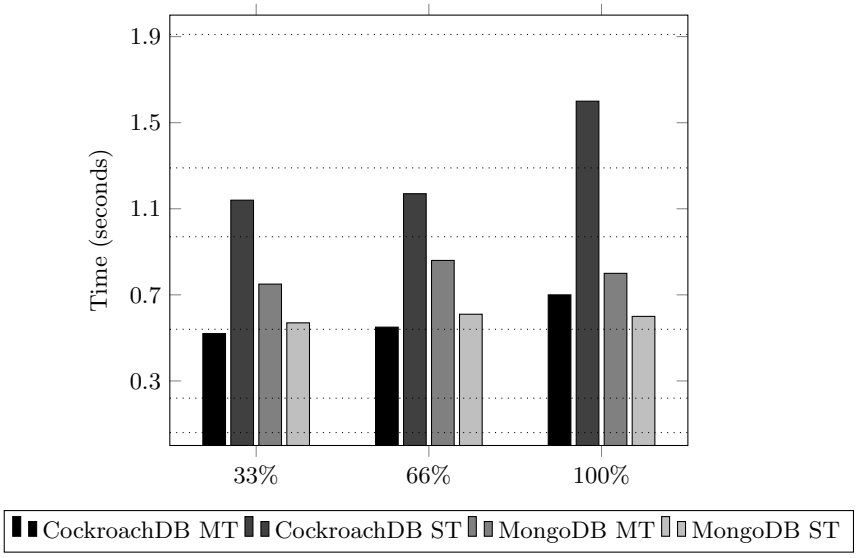


Fig. 6: Degree Distribution - SF3

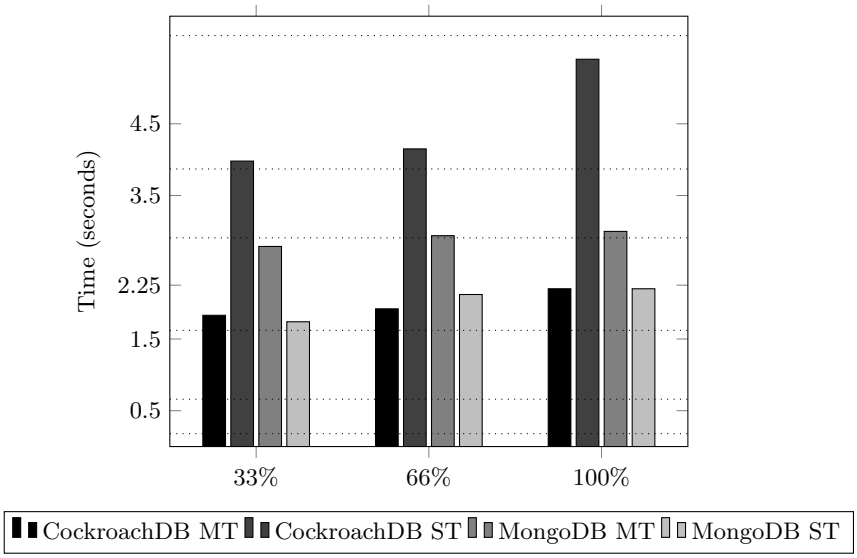


Fig. 7: Degree Distribution - SF10

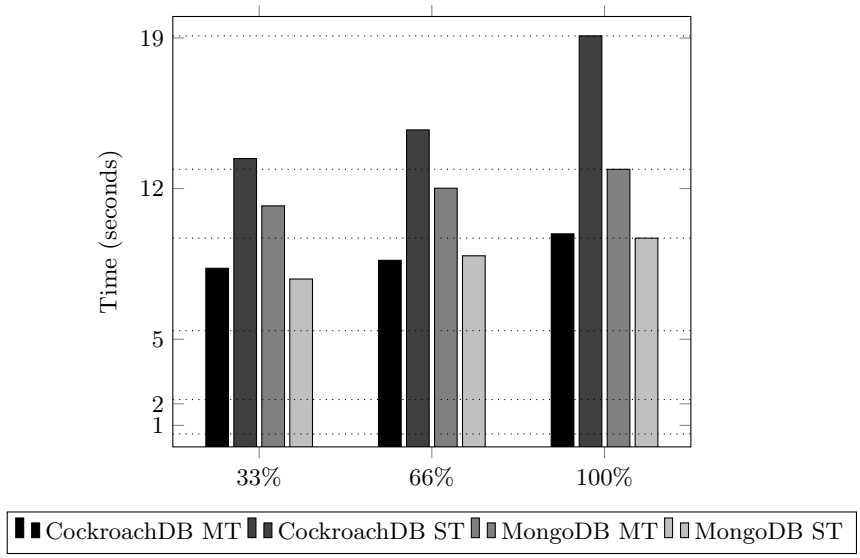


Fig. 8: Degree Distribution - SF3 Person and Forum

is named Fetch All, More Memo is Algorithm 2 which uses Query 2 and Less Memo is Algorithm 3 which also uses Query 2.

From Figures 6, 7 and 8, we compare MongoDB and CockroachDB for querying Historical Degree Distributions. In general, we can conclude the following:

- The best-performing MT model in all datasets and for all percentages of the graph is in CockroachDB, with a percentage difference between 12.5% - 36.05% in SF3, 26.67% - 34.41% in SF10, and 23.26% - 27.85% in SF3 Person and Forum compared to the corresponding MT model in MongoDB.
- The best-performing ST model in all datasets and for all percentages of the graph is in MongoDB, with a percentage difference between 47.86% - 62.5% in SF3, 48.92% - 59.26% in SF10, and 39.69% - 49.21% in SF3 Person and Forum compared to the corresponding ST model in CockroachDB.
- These differences are due to the varying advantages and performance characteristics of each database's structure like the different modeling described in 2. Although, observing solely MongoDB there are conflicting results compared to the [8] or [**?**] where MT was the best performing model. The reason is that MongoDB has a predefined allocated space for each record (document), which gives advantage an to the ST model compared to MT while the size of the document is increased, consequently, while there is an increase in the number of edges or attributes in the nodes as happens in the current dataset. Consequently, the ST model performs best in MongoDB, while the MT model performs best in CockroachDB at the currently tested dataset.
- The best-performing model overall depends on the percentage of the graph used and the dataset. For SF3, CockroachDB is the best-performing model

for 33% and 66% of the graph, with an 8.87% and 9.84% percentage difference to MongoDB ST, respectively. However, for querying 100% of the graph, MongoDB ST is the best-performing model, with a 14.29% decrease in time compared to CockroachDB. For SF10, MongoDB is the best-performing for 33% of the graph, and CockroachDB for 66% of the graph, but the performance increase is less than 10% in each case. Lastly, for SF3 Person and Forum, results are similar to SF10; MongoDB ST outperforms CockroachDB MT in 33% and 100% of the graph, while CockroachDB MT is better for 66% of the graph, with performance differences less than 10% in each case.

A.4 Comparing Different Algorithms of Historical Degree Distribution

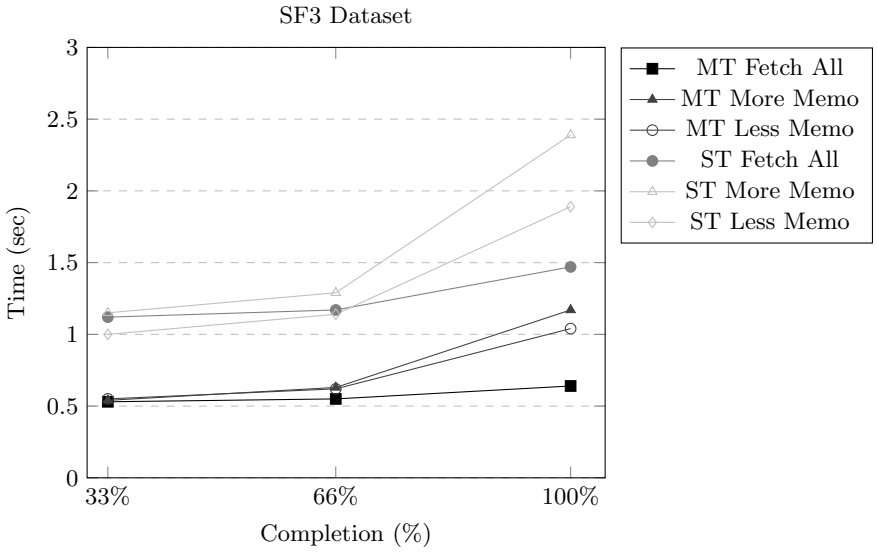


Fig. 9: Comparison of Models and Algorithms for SF3 Dataset

In Figures 9, 10 and 11, we compare the running times of the three developed algorithms for Historical Degree Distribution, both in the ST and MT models, with all algorithms implemented in CockroachDB. Across all three datasets, there are minimal changes in the ranking of the algorithms. Notable observations include:

– The best-performing model is MT Fetch All, with MT More Memo and MT Less Memo almost overlapping for the 2nd and 3rd positions. For 33% of the graph, the performance of these three algorithms is nearly identical.

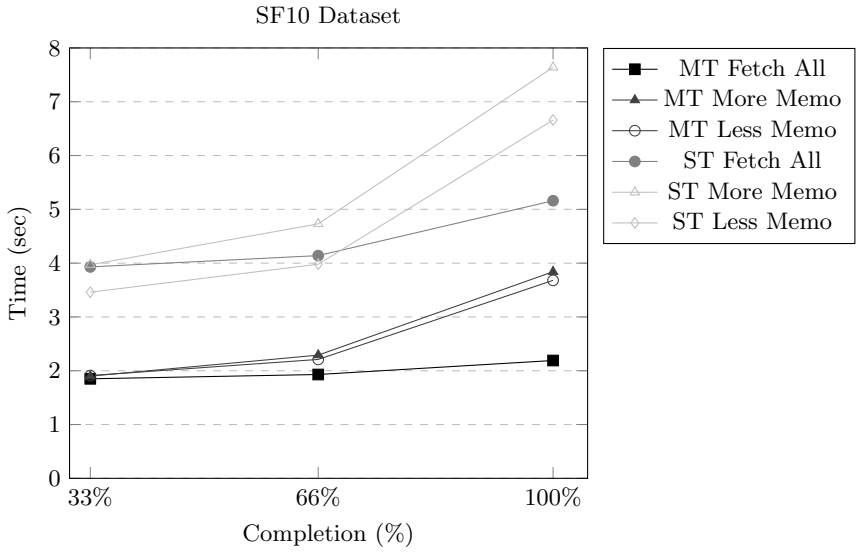


Fig. 10: Comparison of Models and Algorithms for SF10 Dataset

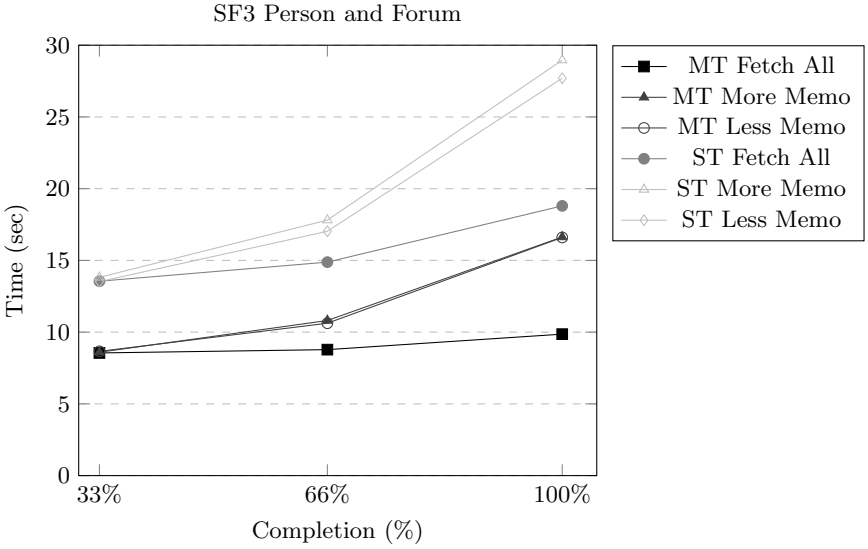
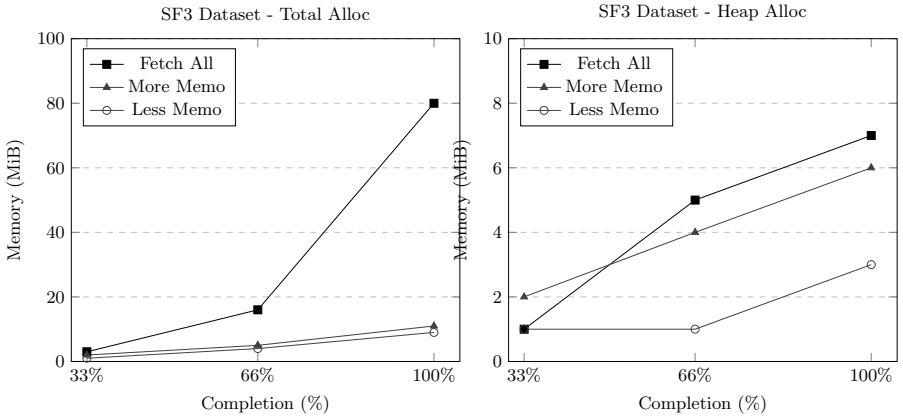


Fig. 11: Comparison of Models and Algorithms for SF3 Person and Forum Dataset

- The percentage difference between MT Less Memo and MT Fetch All for 66% and 100% of the graph is as follows: for SF3, it is 12.5% and 81.54%; for SF10, it is 19.17% and 73.78%; and for SF3 Person and Forum, it is 23.14% and 65.88%. This indicates a reduction in the percentage-wise difference as the datasets get larger while querying 100% of the graph.
- The only change in ranking across the three datasets occurs in SF3 Person and Forum, where ST Fetch All surpasses ST Less Memo at 33% and 66% of the graph.
- In the ST model, ST Less Memo outperforms the ST More Memo algorithm, making it the best-performing ST model for 33% and 66% of the graph in SF3 and SF10.
- It should be noted that these results are from experiments conducted on a single-node machine, and therefore, the new algorithms do not benefit from distributed workload capabilities.

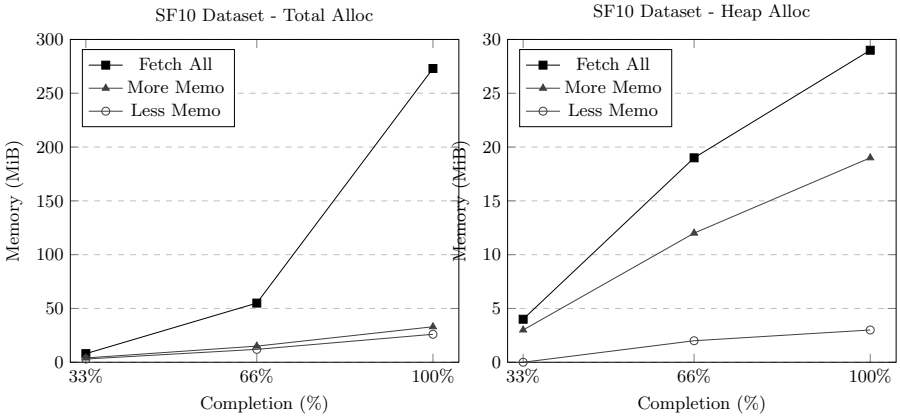


(a) Total Alloc (b) Heap Alloc

Fig. 12: SF3 Dataset

Acknowledgment

"This research was supported by the Hellenic Foundation for Research and Innovation (H.F.R.I.) under the "2nd Call for H.F.R.I. Research Projects to support Faculty Members & Researchers" (Project Number: 3480). "



(a) Total Alloc (b) Heap Alloc

Fig. 13: SF10 Dataset

References

1. Angles, R., Antal, J.B., Averbuch, A., Boncz, P.A., Erling, O., Gubichev, A., Haprian, V., Kaufmann, M., Larriba-Pey, J., Martínez-Bazan, N., Marton, J., Paradies, M., Pham, M., Prat-Pérez, A., Spasic, M., Steer, B.A., Szárnyas, G., Waudby, J.: The LDBC Social Network Benchmark. CoRR **abs/2001.02299** (2020), http://arxiv.org/abs/2001.02299
2. Besta, M., Fischer, M., Kalavri, V., Kapralov, M., Hoefler, T.: Practice of streaming processing of dynamic graphs: Concepts, models, and systems (2021)
3. Debrouvier, A., Parodi, E., Perazzo, M., Soliani, V., Vaisman, A.: A model and query language for temporal graph databases. The VLDB Journal pp. 1–34 (2021)
4. Kosmatopoulos, A., Gounaris, A., Tsichlas, K.: Hinode: implementing a vertex-centric modelling approach to maintaining historical graph data. Computing **101**(12), 1885–1908 (2019). https://doi.org/10.1007/s00607-019-00715-6, https://doi.org/10.1007/s00607-019-00715-6
5. Kosmatopoulos, A., Tsichlas, K., Gounaris, A., Sioutas, S., Pitoura, E.: Hinode: an asymptotically space-optimal storage model for historical queries on graphs. Distributed Parallel Databases **35**(3-4), 249–285 (2017). https://doi.org/10.1007/s10619-017-7207-z, https://doi.org/10.1007/s10619-017-7207-z
6. Lightenberg, W., Pei, Y., Fletcher, G., Pechenizkiy, M.: Tink: A temporal graph analytics library for apache flink. In: Companion Proceedings of the The Web Conference 2018. pp. 71–72 (2018)
7. Orlando, D., Ormachea, J.: Temporal graph visualizer (2020)
8. Spitalas, A., Gounaris, A., Tsichlas, K., kosmatpoulos, A.: Investigation of database models for evolving graphs. In: Combi, C., Eder, J., Reynolds, M. (eds.) 28th International Symposium on Temporal Representation and Reasoning, TIME 2021, September 27-29, 2021, Klagenfurt, Austria. LIPIcs, vol. 206, pp. 6:1–6:13. Schloss Dagstuhl - Leibniz-

Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.TIME.2021.6, https://doi.org/10.4230/LIPIcs.TIME.2021.6

9. Wang, Y., Yuan, Y., Ma, Y., Wang, G.: Time-dependent graphs: Definitions, applications, and algorithms. Data Science and Engineering **4**(4), 352–366 (Dec 2019). https://doi.org/10.1007/s41019-019-00105-0, https://doi.org/10.1007/s41019-019-00105-0
10. Zaki, A., Attia, M., Hegazy, D., Amin, S.: Comprehensive survey on dynamic graph models. International Journal of Advanced Computer Science and Applications **7**(2), 573–582 (2016)