Radboud University Nijmegen

Faculty of Science

Improving Entity Linking Systems With DuckDB

Computing Science Bachelor's Thesis

Author:
Ege Sari

Supervisor:
Arjen P. de Vries

Second reader:
Hannes Mühleisen

January 2023

Abstract

Entity linking systems are complex applications that have an important place in information retrieval systems. Because of its complexity, entity linking is performed by third-party toolkits. While most entity linking toolkits do not use database management systems, there are some toolkits that use the advantage of them. REL is one of these toolkits and uses SQLite, which has a row-based architecture. An alternative for SQLite may be DuckDB. DuckDB has a column-based architecture and has a potential design for zero-copy data transfer in the future. The comparison between the performances of DuckDB and SQLite is made based on the experiments using the advantageous features of DuckDB. DuckDB allows users to manipulate the query optimizer to choose index join over hash join. In addition to this, a novel text compression method FSST is available on DuckDB. The results of the experiments show that SQLite still outperforms DuckDB.

Contents

1 Introduction **3**

2 Related Work **4**

3 Experiments **5**
3.1 Changing SQLite to DuckDB 5
3.1.1 Experiment 1 5
3.1.2 Experiment 2 7
3.2 Comparison of Hash Join and Index Join 10
3.2.1 Experiment 3 10
3.3 Comparison of Enhanced DuckDB and SQLite 11
3.3.1 Experiment 4 11
3.3.2 Experiment 5 13

4 Conclusion **15**

5 Future Work **16**

References **17**

1 Introduction

In order to improve quality and operationalization, DataOps and DevOps are widely used across the industry. Even though the growth in Machine Learning (ML) applications is gigantic, there is a need for a methodology similar to DataOps and DevOps, which will help with the complexities of ML engineering. MLOps is one of the approaches needed to be developed for this purpose [7].

Optimization in large dataset scanning is an essential component of MLOps, especially for NLP applications and recommender systems. Entity linking systems are one of the potentials for NLP applications. The importance of the entity linking systems lies in its massive place in information retrieval systems: From document ranking [21] to entity retrieval [11] and from the query recommendation [20] to knowledge base population [3].

In entity linking systems, the entity mentions are matched with their corresponding knowledge. These systems look up the candidate entities and return the best candidate after disambiguation [9]. A common approach for the linking operation is called wikification, where the Wikipedia pages are the target knowledge base and the mentions are matched with these pages [19, 6].

The entity linking operation is performed by third-party toolkits because of the complexity of the entity linking systems [13]. DBpedia Spotlight [17], GENRE [15], or TAGME [16] are some examples of these third-party toolkits. Yet most of the known toolkits do not meet the requirements in most fields. This deficiency includes maintenance problems [18], aimed at only short texts and/or efficiency problems for long texts [10], and dependency on external sources [5].

REL is a third-party entity linking toolkit developed at Radboud University in order to overcome the deficiencies of other toolkits. It uses the standard entity linking architecture which is based on the three-steps pipeline. In the first step of the pipeline, mention detection, the text spans that possibly can be linked to an entity, called mentions, are detected. The second step, candidate selection, is about choosing candidate entities that the mentions can be linked to. The third and last step, entity disambiguation, is responsible for choosing the most corresponding entity to the mention and linking each other [2, 1]. As an example, these three steps can be applied to a small sentence such as "Obama will visit Merkel." In the first step, the mentions "Obama" and "Merkel" are chosen. In the second step, the candidate entities for "Obama" would be "Barack Obama" and "Obama (a planaria kind)". In the last step, the best entity "Barack Obama" is chosen and the mention is linked to this entity. In the second step of this pipeline, in the candidate selection step, the candidates are stored in a database of probabilities $p(e|m)$ where e is the entity and m is the mention [13, 8]. In the next step of the pipeline, in the entity disambiguation step, the queries are made on this database to extract the best matching entity.

Most entity linking toolkits do not use a database management system. Yet in recent years, there are toolkits developed using database management systems. Nordlys can be given as an example, it uses MongoDB as a database management system [10]. Using a database management system is important for the entity linking toolkit because it increases the speed of data retrieval by using its own features like query optimizer and other efficiency features. The efficiency of the entity linking toolkits is directly influenced by the efficiency of the database management system. The main topic of this thesis is to study the time efficiency of third-party entity linking toolkits and to consider improv-

ing their usage of database management systems behind the toolkit. Possible ways of improvement are discussed, experimented and interpreted. REL is used as the use case example and all the experiments are carried out using REL. But it must be noted that REL is also a developing toolkit, which means the first two experiments are tested on different versions. Because of this, the experiments described in Section 3.1 are done in the older version of REL whereas the experiments described in Section 3.2 and Section 3.3 are done in the newer version.

In its current version, REL uses SQLite. SQLite is a twenty-two years old OLTP database engine. It is an embedded database management system, which means it is a kind of library rather than a standalone app. It can "be embedded" in an application and can be run in a separate host process. Also, SQLite has a row-wise storage architecture [12]. On the other hand, DuckDB is an embedded OLAP database management system that uses column-wise storage. It has been introduced in 2019. DuckDB may perform comparable behavior with SQLite for small datasets. But, because of its architecture, DuckDB will perform better especially for larger datasets, while SQLite will begin to not tolerate it because of its row-based execution model. Although DuckDB does not have a server process or client protocol interface, it is accessed using a C/C++ API. Also, DuckDB allows applications that previously used SQLite to use DuckDB through re-linking or library overloading by an SQLite compatibility layer [14]. It has huge potential, especially for machine learning and NLP applications. For these kinds of applications, SQLite first creates a local copy of the data, sends them to the ML/NLP libraries, and does the operations over there. However, DuckDB has the aim to be an analytical database to deal with edge computing scenarios like the previous example [14]. By its architecture, theoretically, DuckDB may allow the ML/NLP library to its data, and continue to hold the data in its own data tables, without the need of creating a local copy. This will give the benefit of zero-copy operations for ML/NLP applications.

These benefits and potential advantages of DuckDB may be used in entity linking systems. Our main hypothesis is using DuckDB would increase the efficiency of the backend performance and therefore totally increase the efficiency of the entity linking system itself.

In the next sections, the ways to improve REL are discussed and experimented with. The first and main way to improve is the implementation of DuckDB to REL as the database management system. After the initial implementation, the important features of DuckDB such as persistent index and pragma for forcing index join are used and compared to SQLite performance. We use also the FSST string compression method. Briefly, FSST(Fast Static Symbol Table) is a compression method used for strings in databases. It provides fast and efficient string compression by replacing strings with numbers using a symbol table [4]. Our main hypothesis is that using DuckDB and its advantageous features such as forcing index join and enabling FSST may improve the efficiency of entity linking systems.

2 Related Work

In August 2022, Chris Kamphuis and his fellow researchers published a paper [13]. The main topic of the paper is improving REL. They have discussed how to enhance the REL entity linking toolkit by proposing improvement methods and interfering with experiment results. One of the improvement methods was converting the database management system of REL, SQLite, to DuckDB. But the experiment results showed

that SQLite was still outperforming DuckDB. Although the version of DuckDB is not known at that time, it is certain that it was different from the version we used to run experiments which is 0.6.0. In addition to this, in this work, the only improvement including DuckDB is replacing SQLite with DuckDB. On the other hand, in this thesis, not only the backend is changed, but also the advantages of DuckDB are going to be tested and compared. Furthermore, algorithms that are both compatible with DuckDB and believed to improve the efficiency of entity linking systems like FSST are going to have experimented.

3 Experiments

In this section, the experiments are presented with their methodologies and results discussed. There are five experiments with four hypotheses. The first two experiments are to illustrate the runtime comparison based on the database choice for REL DuckDB and SQLite. The third experiment is focused on the performance of DuckDB using its feature and comparison with plain DuckDB. The fourth experiment focused on again the comparison of the performance of SQLite and DuckDB, but this time DuckDB configuration is modified to control the choice of join operator. The last and fifth experiment is focused on using FSST and index join forcing feature together. Experimental setup and the results are available online on GitHub[1].

3.1 Changing SQLite to DuckDB

The following two experiments are about testing the efficiency of DuckDB adapted to REL.

3.1.1 Experiment 1

The first experiment is about making a comparison between DuckDB and SQLite based on the performance of REL when using that database as the backend.

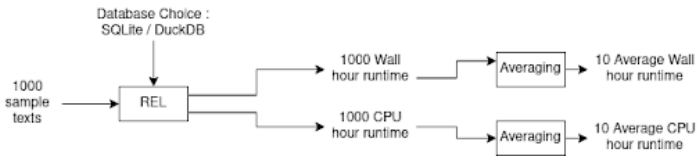
Methodology

The methodology is simple. REL is loaded with one thousand sample texts one by one. The sample of texts is chosen from the first five thousand documents of MS Marco V2 collection. Each sample text contains different words, different sentences and has different sizes. Then for each sample, the runtime is measured in both wall time and CPU time by recording the start time and end time. The difference between the start and end time is regarded as the runtime.

After the measurement is done, the comparison is made in two methods. The first method is comparing each sample. The second method aggregates the data, for instance by looking at average runtimes. It separates the result set into ten groups with a group size of one hundred. Then the average value of these groups is calculated. At the end of this method, we have ten results; each result represents an average value of one hundred consecutive results. The second method aims for a more refined result.

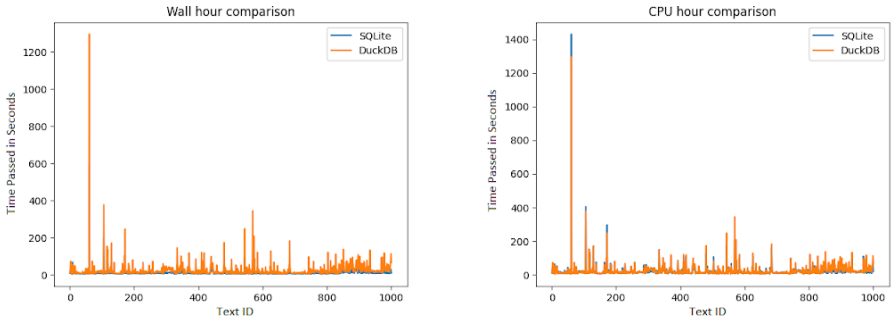
In both methodologies, we can say which database is more efficient directly based on the test results. The illustration of the methodology can be seen in the diagram below:

[1] https://gitlab.science.ru.nl/esari/rel-improvement



Results

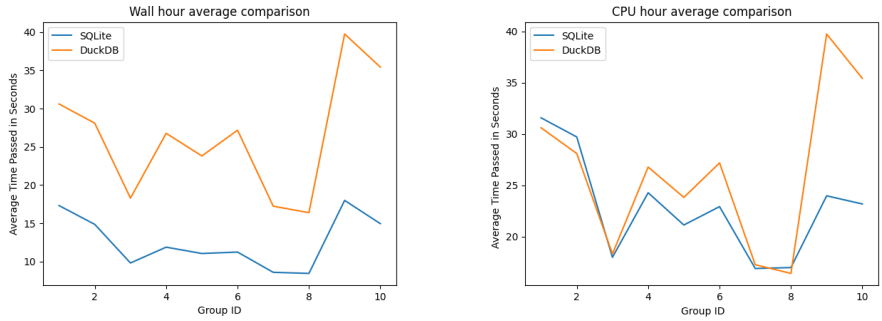
The results without averaging suggest that SQLite performs better than DuckDB. According to the comparison based on wall hour, it is clear that DuckDB is inefficient compared to SQLite which is obvious in Figure 1a. The comparison based on CPU hours suggests that there are some points DuckDB performs better. Yet these points are still a minority as can be seen in Figure 1b.



(a) Wall hour comparison (b) CPU hour comparison

Figure 1: Runtime Comparisons of SQLite and DuckDB for Experiment 1

But since these results are not so refined and may be not accurate, we have used averaging. But again the results do not change. The results clearly indicate that the wall hour runtime of DuckDB is higher than that of SQLite. Also, the comparison based on the CPU hour runtime shows again SQLite has better efficiency than DuckDB in general. The results can be seen in Figure 2a and Figure 2b.



(a) Wall hour average comparison (b) CPU hour average comparison

Figure 2: Average Runtime Comparisons of SQLite and DuckDB Experiment 1

3.1.2 Experiment 2

The second experiment is again about comparing DuckDB and SQLite for REL. Since the first experiment cost a lot of time, a more efficient approach was developed using micro-benchmarking. This experiment tests the SQL queries used in REL and measures the time, rather than measuring the end-to-end runtime of REL. This experiment is done several times since it takes less time. The results do not differ. This means that the runtime difference between SQLite performance and DuckDB performance can not be explained by the overhead of calling the database engine from its host.

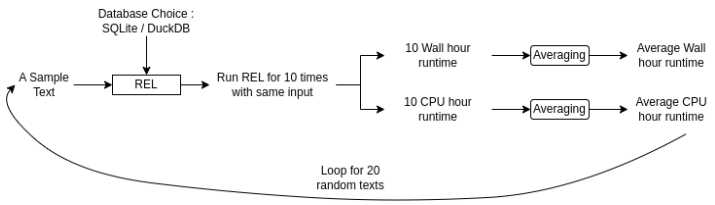
Methodology

The methodology is based on measuring the queries. For each query that can be run on the database for REL, we have measured the runtime. The runtime is measured in both wall time and CPU time, again by recording the start time and end time. The difference between the start and end time is regarded as the runtime for that query.

The main motivation for making a new methodology different from the Experiment 1 is the time efficiency of the experiment. The previous experiment took too much time and it does not offer refined results. Also, we wanted to make sure that the difference between the SQLite performance and DuckDB performance is caused by the difference in the query efficiencies rather than the overhead of calling the database engine.

Then twenty texts from the first five thousand documents of MS Marco V2 collection were chosen randomly. Each text is run on REL ten times and for each query, the runtime is measured. After that, the mean value of the runtimes is extracted from the ten results for every twenty texts. In the end, for each query type, we have a group of twenty values. And each value represents an average runtime value of that query of ten times run text sample.

By this methodology, we can explain which database is more efficient for running queries that are used in REL. Hence we can explain which database is more efficient for REL in an indirect way based on the experiment results. An illustration of the methodology can be seen in the diagram below:



Results

While this experimentation was done, in REL there were six functions that are running queries on the selected database. We have observed that only two query functions are used actively when running REL, at least for our case. They are `lookup()` and `llookup_wik()`. The former begins a transaction and executes a SELECT query in that transaction. The latter one directly executes a SELECT query. The runtime results are the time that has passed from the query execution started until it ended.

As we have mentioned before the results do not differ from the previous experiment. For the `lookup()` function, SQLite performs better than DuckDB performs. In detail,

SQLite is always more efficient compared to DuckDB based on the CPU hour runtime. Also, it seems faster than DuckDB except for a few cases for wall hour runtime. The figures Figure 3 and Figure 4 illustrate the performance comparison.

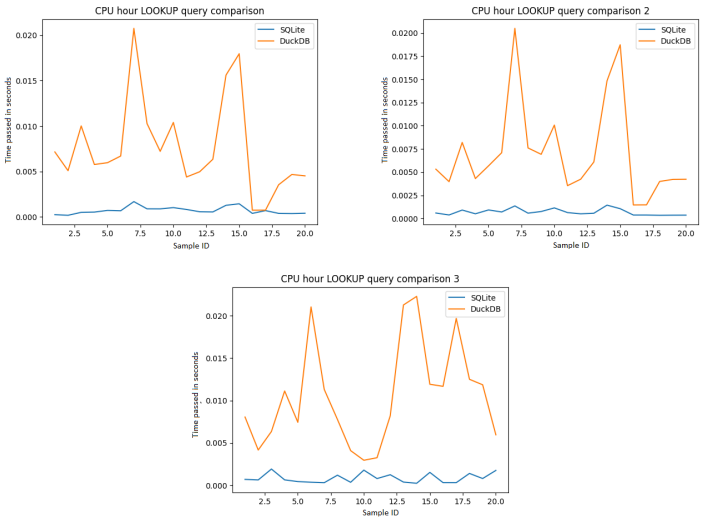


Figure 3: CPU hour comparisons of `lookup()` for Experiment 2

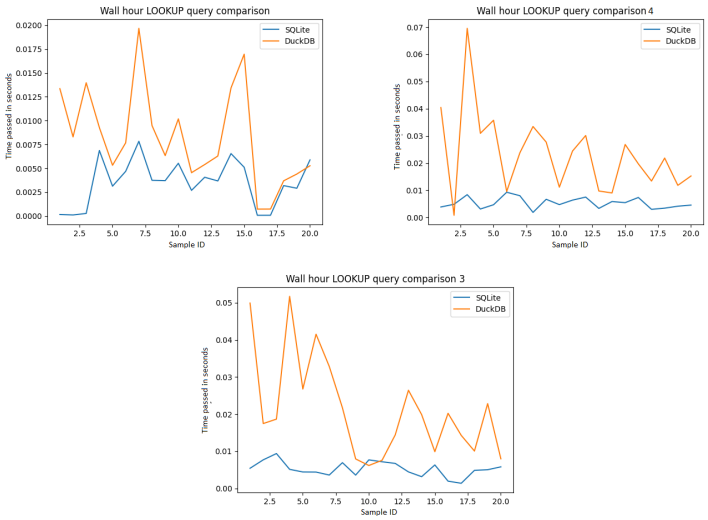


Figure 4: Wall hour comparisons of `lookup()` for Experiment 2

For the lookupwik() function, the difference between DuckDB and SQLite performance is evident. SQLite always performs better than DuckDB. The figures Figure 5 and Figure 6 illustrate the performance comparison.

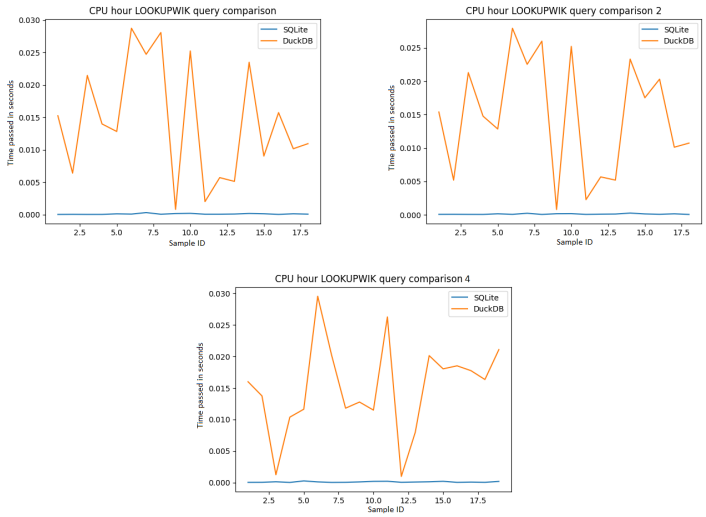


Figure 5: CPU hour comparisons of `lookup_wik()` for Experiment 2

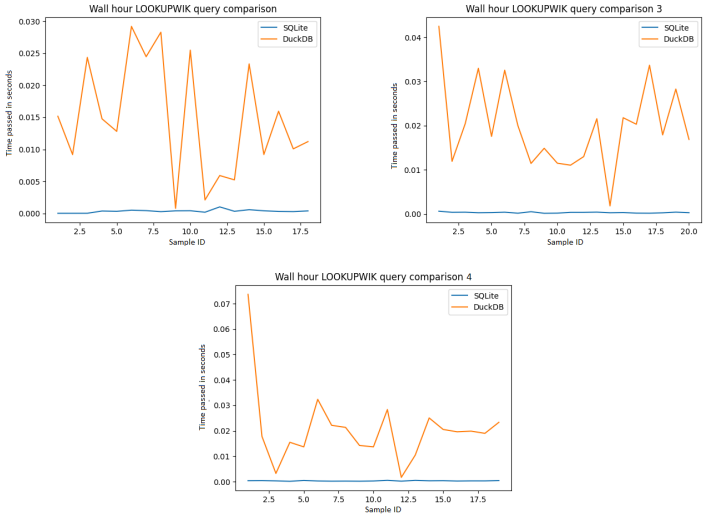


Figure 6: Wall hour comparisons of `lookup_wik()` for Experiment 2

3.2 Comparison of Hash Join and Index Join

3.2.1 Experiment 3

As can be seen in the previous results, deploying DuckDB instead of SQLite in the entity linking system is not enough. Furthermore, SQLite still performs better. Yet we can try to improve the performance of DuckDB.

Joining is essential for database operations. In REL, we join the tables based on the given lookup word. There are two join categories: server-side joins and client-side joins. In server-side join, all the data is gathered and the join operation is done once. In client-side join, the join operation is executed partially by the client and rest is handled at the server-side. Server-side join is more efficient compared to client-side join since the amount of data transmitted is decreased and hence the time cost is decreased.

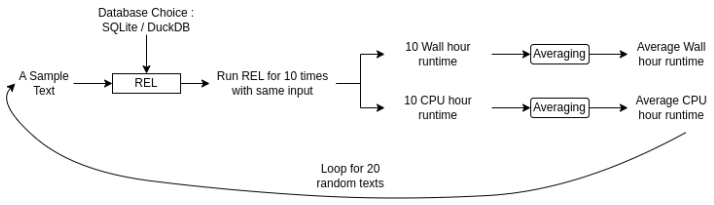
When we started to do this experiment, REL was updated. In the previous version, the words are queried one by one and for each time join operation is applied. In the novel version, the lookup can be done for multiple words, and the join operation is applied once. So we can say the current version of REL uses a server-side join model.

When we inspect the query optimization plan of DuckDB for the queries used in the entity linking system, we observe that a hash join is used. Hash joins are considered better based on their algorithmic complexity. But in REL, we use queries for comparison between strings. When the chosen string has low selectivity, it will increase the latency and decrease the efficiency. Rather than using hash joins, we can force DuckDB to use index joins. Typically, nested loop joins, are not preferred because of their complexity. We can use the persistent index feature of DuckDB to use index join, an optimized version of nested loop joins. In this way, we can improve the query speed and observe better performance compared to the hash join.

In this experiment, we will observe the performance of entity linking systems using DuckDB. We will make a comparison between using hash joins and index joins in the query optimization.

Methodology

The methodology will be the same as in Experiment 2. But this time rather than choosing DuckDB or SQLite, the preference would be made between plain DuckDB and DuckDB with its feature. But again there will be twenty randomly chosen texts from the first five thousand documents of MS Marco V2 collection for each run. Each text will be taken as input and run ten times. The results will be the average wall hour and CPU hour runtime for each query function for the given text. The illustration of the methodology is given below:



Results

As mentioned before, while this experimentation was done, REL was updated. In the previous version, REL makes a lookup query for each word. One member of the REL team discovered that there can be one lookup done for several words, which will lead to a decrease in the total lookup. Hence the overall time would be lower. Indeed in the benchmark results, REL became faster 25%. The approved pull request can be seen on GitHub[2]. Again, in REL there were six functions that are running queries on the selected database. But there was a new function called `lookup_many()`. We have observed that three query functions are used actively when running REL, at least for our case. They are `lookup()`, `lookup_many()`, and `lookup_wik()`.

The results of the experiments are plotted. The blue line represents the runtimes where the natural join operator is chosen by DuckDB query optimizer. The orange line represents the runtimes where the query optimizer is forced to choose index join when it is possible. Even though the experiment is done three times for the accuracy, the runtime variance is low. Hence a set of random results is chosen. The graph in Figure 7 illustrates the experimental results for the `lookup()` query, the graph in Figure 8 illustrates the experimental results for the `lookup_many()` query, and the graph in Figure 9 illustrates the experimental results for the `lookup_wik()` query.

As can be seen in both CPU and wall hour runtimes, enforcing index join is either better or the same. From these graphs, we infere two things. First, as can be seen, at some points of the runtime graphs the runtimes are the same. This may seem a contradiction with our hypothesis, suggesting index join is better than hash join. When we inspect the query optimization plan in DuckDB, we see that DuckDB does not always use hash join. DuckDB chooses index join when the estimated selectivity of the string is low if the index is provided. Second, indeed we can say that index join has a better overall performance when we can use persistent indexes as we can using DuckDB. Hence we should get the benefit of persistent indexes and use the pragma to force query optimization using index joins when it is possible in entity linking systems.

3.3 Comparison of Enhanced DuckDB and SQLite

In the last two experiments, we would like to compare DuckDB and SQLite but now DuckDB's advantageous features are used.

3.3.1 Experiment 4

In this experiment, we would like to compare SQLite and DuckDB by using its persistent index and index join force features. Our hypothesis is DuckDB may be faster than SQLite if we use the pragma for forcing index join when it is possible. Again the experiment will be based on the measurements of runtimes of REL using either DuckDB or SQLite.

Methodology

For this experiment, the methodology will be the same as Experiment 2 and Experiment 3. There are twenty randomly chosen texts. Each text will be taken as input and run ten times. Again we do not choose one thousand samples because of the same reason, it takes too much time and do not give details on the query efficiency of the chosen database system. The results will be the average wall hour and CPU hour runtime for

[2]https://github.com/informagi/REL/pull/127

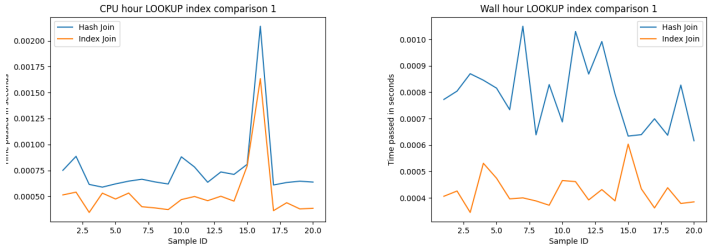


Figure 7: CPU and Wall hour comparisons of `lookup()` for Experiment 3

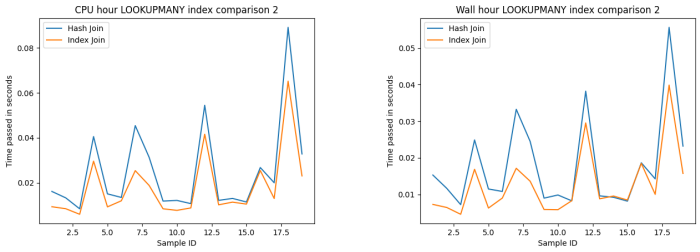


Figure 8: CPU and Wall hour comparisons of `lookup_many()` for Experiment 3

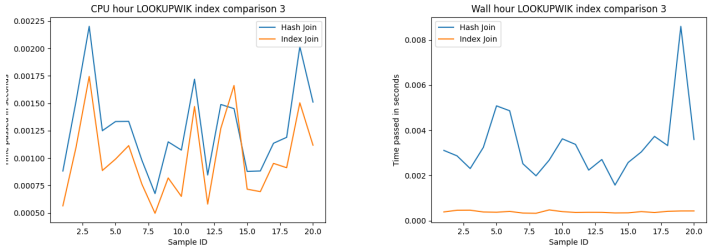


Figure 9: CPU and Wall hour comparisons of `lookup_wik()` for Experiment 3

each query function for the given text. Again there are three query functions namely `lookup()`, `lookup_many()`, and `lookup_wik()`.

Results

The results of the experiments are plotted. The blue line represents the runtimes when using SQLite. The orange line represents the runtimes where using DuckDB with index join when it is possible is forced. Even though the experiment is done three times for accuracy, the results do not differ. Hence a set of random results is chosen. The graph in Figure 10 illustrates the experiment results for the `lookup()` query. The graph in Figure 11 illustrates the experiment results for the `lookup_many()` query. The graph in Figure 12 illustrates the experiment results for the `lookup_wik()` query.

In the plots, it can be seen that SQLite performs better than DuckDB by having lower runtimes. A detailed inference can be done. Even though we use persistent indexes and pragma to force the query optimizer to use the index join when it is possible, still SQLite has better efficiency compared to DuckDB. Hence our hypothesis at the beginning of the experiment failed. The only explanation behind this failure is, indeed the current version of DuckDB offers a (slightly) lower runtime performance than SQLite.

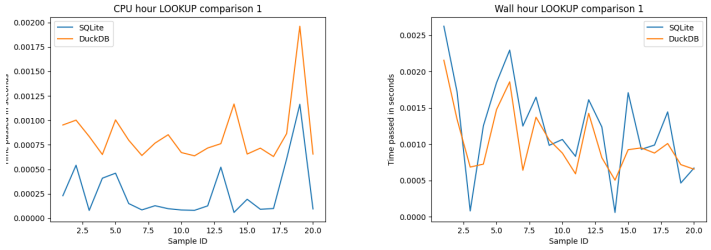


Figure 10: CPU and Wall hour comparisons of `lookup()` for Experiment 4

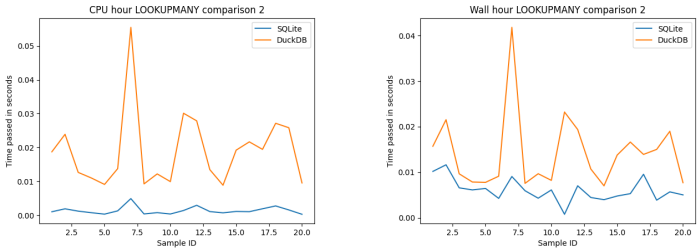


Figure 11: CPU and Wall hour comparisons of `lookup_many()` for Experiment 4

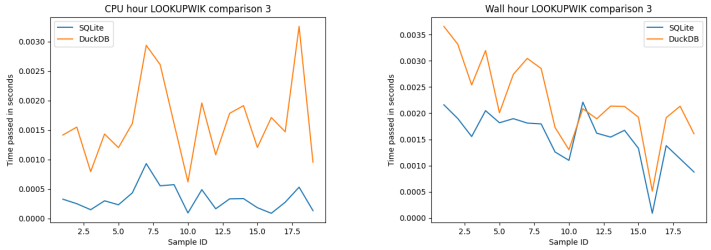


Figure 12: CPU and Wall hour comparisons of `lookup_wik()` for Experiment 4

3.3.2 Experiment 5

In the last experiment, we would like to compare SQLite and DuckDB by using the FSST feature with persistent index and index join force features. As mentioned before FSST is a novel text compression method that offers faster string compression [4]. Our hypothesis is DuckDB may be faster than SQLite if we set enable FSST vectors for text compression. Also, we will use pragma to force index join. This experiment may be seen as an extension of the previous experiment. Again the experiment will be based on the measurements of runtimes of REL using either DuckDB or SQLite.

Methodology

For this experiment, the methodology will be the same as Experiment 2, Experiment 3, and Experiment 4. There are twenty randomly chosen texts from the first five thousand documents of MS Marco V2 collection. Each text will be taken as input and run ten times. Again we do not choose one thousand samples because of the same reason, it takes too much time and does not give details on the query efficiency of the chosen database system. The results will be the average wall hour and CPU hour runtime for each query function for the given text. Again there are three query functions namely `lookup()`, `lookup_many()`, and `lookup_wik()`.

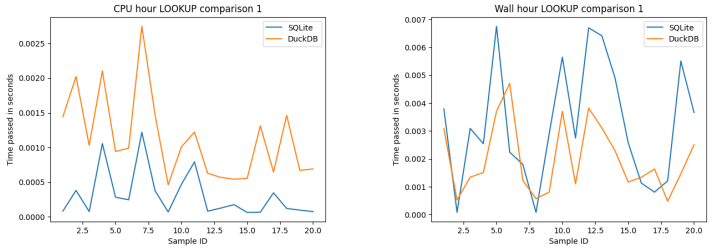


Figure 13: CPU and Wall hour comparisons of `lookup()` for Experiment 5

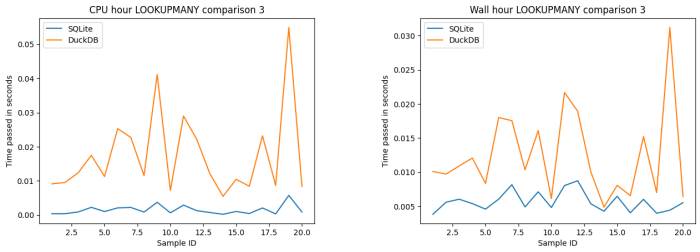


Figure 14: CPU and Wall hour comparisons of `lookup_many()` for Experiment 5

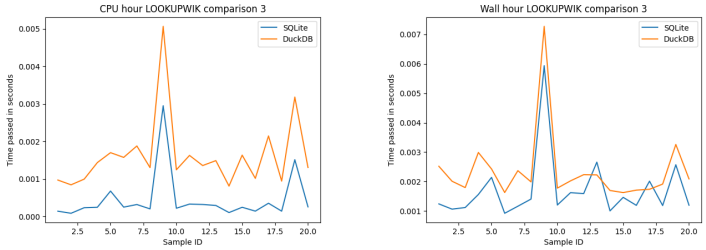


Figure 15: CPU and Wall hour comparisons of `lookup_wik()` for Experiment 5

Results

The results of the experiments are plotted. The blue line represents the runtimes when using SQLite. The orange line represents the runtimes where using DuckDB with FSST and index join are enabled. Even though the experiment is done three times for accuracy, the conclusion is the same. The graphs in Figure 13, 14 and 15 illustrates the experiment results for the `lookup()`, `lookup_many()`, and `lookup_wik()` queries, respectively.

In the graphs, it can be seen that interestingly DuckDB performs better for the wall hour runtime for `lookup()` function. But for other functions and runtimes, this does not apply, again SQLite performs better. Hence based on the graphic data, we can say that SQLite performs better when we enable FSST text compression and force index join when it is possible on DuckDB.

For this specific experiment, it is important to acknowledge that we are not sure if we have enabled FSST correctly. Following by the documentation of DuckDB, we believe we have enabled FSST text compression. On the other hand, this may be not the optimal way to use FSST on entity linking systems. Rather than only enabling FSST on the DuckDB side, we may have to change the architecture of REL.

4 Conclusion

At the beginning of this thesis, the main hypothesis was that using DuckDB as the database management system for the third-party entity linking toolkits would be more efficient. The main reason for this is DuckDB is an OLAP database management system which means it uses columnar storage architecture. Using an OLAP database management system is more accurate for entity linking systems rather than using OLTP database management systems such as SQLite.

For this thesis, the example entity linking toolkit is REL(Radboud Entity Linker). It uses SQLite as the main database management system. Our main hypothesis is replacing SQLite with DuckDB would reduce the runtime and increase the overall efficiency. In addition to this, DuckDB carries a potential for zero-copy data transfer thanks to its architecture mentioned in previous sections.

To prove that we have run four experiments. The first two experiments are focused on the comparison of SQLite and DuckDB performances. In these two experiments, we fed REL with randomly chosen sample texts and measured the runtimes separately while using SQLite or DuckDB as the database management system.

The third experiment is focused on the advantages of DuckDB that we can use to outperform SQLite. There are two advantageous features of DuckDB: Persistent indexes and pragma to force the query optimizer to use index join when it is possible. The methodology was the same as the previous experiment, we fed REL with randomly chosen sample texts and measured the runtimes separately while using DuckDB with or without its features.

The last two experiments are focused on again the comparison of SQLite and DuckDB performances. But this time, as differing from the first two experiments, we have used DuckDB with its advantageous features such as enabling FSST text compressions and forcing index join.

We have inferred that clearly DuckDB is slower than SQLite based on wall time and CPU time runtimes from Experiment 1 and Experiment 2. The reason behind this is indeed simple. SQLite has been being developed for the last twenty years, while DuckDB is a new innovative solution. Indeed using plain DuckDB as a replacement for SQLite may not be a way to improve the efficiency, at least not with this version of DuckDB.

But in Experiment 3, it can be inferred that using index joins with persistent indexes is better than using hash joins for DuckDB. In addition to this, we have seen that sometimes the query optimizer of DuckDB also chooses index joins, or using index joins is not always possible. Using index joins is better than using hash joins, for entity linking systems. Because in most entity linking systems, we run queries based on the string values. Hash joins may create an increase in runtime if the selectivity of the string is low. And most of the time, the selectivity of the string is low. Hence using index join over hash join provides better performance.

Although DuckDB is run with pragma that forces query optimizer to use index joins when it is possible, SQLite performs still better as can be seen in Experiment 4. Regarding the results from the first two experiments, where the query optimizer of DuckDB chooses hash joins mostly, we can say that DuckDB does not perform better than SQLite. The only explanation for that is indeed DuckDB is not more efficient

compared to SQLite.

Lastly, SQLite has a better performance when we use FSST text compression on DuckDB with index joins. This inference can be seen in the results from the last experiment Experiment 5.

In conclusion, SQLite (still) outperforms DuckDB even though we use the advantages of DuckDB. But still, there may be more improvements and research to make on this topic. These possible improvements are discussed in the next section.

5 Future Work

In this section, further improvements are discussed. The most improvement may be done in the optimization of entity linking toolkits for FSST on DuckDB. We believe adapting the entity linking toolkits for FSST text compression would decrease the total runtime and hence increase efficiency. Since DuckDB has already adapted the FSST algorithm, using DuckDB becomes advantageous for this operation.

As mentioned before, DuckDB has great potential for zero-copy data transfer. This may be implemented in real, especially on entity linking systems. This would increase the performance of DuckDB and it may outperform SQLite.

It can be noticed that parallel computing techniques are not used in the experiments with DuckDB. On the other hand, users are allowed to change the degree of parallelism while using DuckDB. The same experiments we have done in this thesis may be done in order to find an optimal degree of parallelism for DuckDB and this could lead to the optimal usage of DuckDB for entity linking systems and lead to better runtime performance than SQLite.

In this thesis, REL is used as the example entity linking toolkit for the experiments. Yet, REL is designed for individual texts or files [13, 1]. As mentioned in the previous sections, an advanced version of REL, REBL is introduced. In this advanced version, REL is modified to handle multiple documents to be tagged simultaneously. REL was the use case for our experimentation since it has the common three-step pipeline, wikification, and handling individual documents like most entity linking toolkits. The previous version of DuckDB is implemented in REBL and tested its runtime. Unfortunately, SQLite was still faster than DuckDB. Now, it is highly recommended to run these experiments on REBL with the current or future versions of DuckDB. The synergy of batch processing and columnar storage of DuckDB may improve the overall efficiency.

References

[1] Hasibi F. Dercksen-K. Balog K. de Vries A. P. an Hulst, J. M. Rel: An entity linker standing on the shoulders of giants. *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 2197–2200, July 2020.

[2] Krisztian Balog. *Entity-Oriented Search*. Springer, 2018.

[3] Ramampiaro H. Takhirov N. Nørvåg-K. Balog, K. Multi-step Classification Approaches to Cumulative Citation Recommendation. *Proceedings of the 10th Conference on Open Research Areas in Information Retrieval*, pages 121–128, May 2013.

[4] Neumann T. Leis V. Boncz, P. FSST: fast random access string compression. *Proceedings of the VLDB Endowment*, 13(12):2649—-2661, August 2020.

[5] Ferragina P. Ciaramita M. Rüd-S. Schütze H. Cornolti, M. SMAPH: A Piggyback Approach for Entity-Linking in Web Queries. *ACM Transactions on Information Systems*, (13):1–42, January 2019.

[6] I.H. Witten D. Milne. Learning to link with Wikipedia. *Proceedings of the 17th Conference on Information and Knowledge Management*, pages 509–518, October 2008.

[7] Salama K. et al. *Practitioners Guide to MLOps*. Google, May 2021.

[8] Hofmann T. Ganea, O. Deep joint entity disambiguation with local neural attention. *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 2619—-2629, 2017.

[9] Radford W. Nothman J. Honnibal-M. Curran J. R. Hachey, B. Artificial Intelligence, Wikipedia and Semi-Structured Resources Evaluating Entity Linking with Wikipedia. *Artificial Intelligence*, 194:130—-150, January 2013.

[10] Balog K. Garigliotti D. Zhang-S. Hasibi, F. Nordlys: A Toolkit for Entity-Oriented and Semantic Search. *Proceedings of the 40th International ACM SIGIR conference on research and development in Information Retrieval*, pages 1289—-1292, August 2017.

[11] K. Bratsberg S. Hasibi F., Balog. Exploiting Entity Linking in Queries for Entity Retrieval. *Proceedings of the 2016 ACM International Conference on the Theory of Information Retrieval*, pages 209—-218, September 2016.

[12] R. Hipp. Database File Format. *https://www.sqlite.org/*, 2019.

[13] Hasibi F. Lin J. Vries-A.P. de Kamphuis, C.F.H. REBL: Entity Linking at Scale. *Ceur Workshop Proceedings, (2022)DESIRES 2022 – 3rd International Conference on Design of Experimental Search Information REtrieval Systems*, 30–31:1–8, August 2022.

[14] Raasveldt M. Mühleisen, H. DuckDB: an Embeddable Analytical Database. *Proceedings of the 2019 International Conference on Management of Data*, pages 1981—-1984, June 2019.

[15] S. Riedel-F. Petroni N. D. Cao, G. Izacard. Autoregressive Entity Retrieval. *International Conference on Learning Representations*, 2021.

[16] U. Scaiella P. Ferragina. TAGME: On-the-Fly Annotation of Short Text Fragments (by Wikipedia Entities). *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, pages 1625–1628, October 2010.

[17] A. García-Silva C. Bizer P. N. Mendes, M. Jakob. DBpedia Spotlight: Shedding Light on the Web of Documents . *Proceedings of the 7th International Conference on Semantic Systems*, pages 1–8, September 2011.

[18] Blanco R. Mehdad Y. Stent-A. Thadani K. Pappu, A. Lightweight Multilingual Entity Extraction and Linking. *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pages 365–374, February 2017.

[19] A. Csomai R. Mihalcea. Wikify!: Linking documents to encyclopedic knowledge. *Proceedings of the 16th Conference on Information and Knowledge Management*, pages 233—-242, 2007.

[20] Meij E. de Rijke E. Reinanda, R. Mining, Ranking and Recommending Entity Aspects. *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 263—-272, August 2015.

[21] Callan J. Liu T. Xiong, C. Word-Entity Duet Representations for Document Ranking. *Proceedings of the 40th International ACM SIGIR conference on research and development in Information Retrieval*, pages 763–772, August 2017.