
DATABASE TECHNOLOGY
2IMD10

Project Report

Group 32

Quartile 3: 2019-2020

Lucas Giordano	1517317 (UTJ2ZKSLE)
Wessel van Lierop	0866126 (UTHT4MSTE)
Karina Mankevic	1036163 (UTEMALSE8)
Valeriya Prokopova	1033287 (UTS3VMV7H)

Abstract

Cardinality estimation is an important concept in the field of query optimization for databases. In this document, we introduce a cardinality estimator we implemented in the context of project **quickSilver** using RDF database. This estimator is built upon synopsis-based methods found in previous work on the subject. Using this approach, we are able to lift the assumption of containment. This assumption is commonly made in relational databases, but is not applicable to the RDF databases found in the project. We analyze the results provided by our implementation, concluding that it does indeed outperform implementations that operate under the containment assumption.

When evaluating database queries, we would like to do so in a Smart, Quick, and Frugal manner. In the second part of this document, we use these criteria to guide development of a query evaluation module within the context of project quickSilver. We find a query evaluation plan, utilizing the cardinality estimator we developed in the first part in addition to techniques such as k -path indices, dynamic programming, and different join algorithms.

Table of Contents

1	Introduction	1
2	Context and Motivation	1
3	Plan	2
4	Progress Discussion	2
4.1	Implementation Basis	2
4.2	Naive Estimator	3
4.2.1	Results	5
4.3	Estimator using SYN1 and SYN2	5
4.3.1	Kleene star	6
4.3.2	Results	7
4.4	Estimator using SYN1, SYN2 and SYN3	7
4.4.1	Results	10
5	Part 1: Conclusion	11
6	Part 2: Strategy	12
6.1	Smart	12
6.2	Quick	12
6.3	Frugal	12
6.4	Implementation order	12
7	Step-by-Step Plan	12
8	Literature Survey	13
9	Progress	13
9.1	Index	13
9.1.1	1-path index	14
9.1.2	Edge type and intermediate data-structure	14
9.2	Physical Operators	14
9.2.1	Abstract Class	14
9.2.2	Pipelining	15
9.3	Main Physical Operators	15
9.3.1	IndexLookUp	15
9.3.2	KleeneStar	15
9.3.3	MergeJoin	16
9.3.4	Leaderboard	16
9.4	Duplicate removal	16
9.5	Custom Index	17
9.6	Logical Optimization	18
9.6.1	Unbound queries	18
9.6.2	Bound queries	19
9.6.3	Cost Model	20
9.7	Cardinality estimation	21
9.7.1	Improved estimation	21
9.7.2	Cardinality estimation for bushy trees	21
9.7.3	Leaderboard point	21
9.8	Pruning non-joining edges in MergeJoin	22
9.9	Pipelining and IndexJoin	23
9.9.1	Introduction to IndexJoin	23

9.9.2	Rechability queries & Random-Access Index	24
9.10	2-path index for $l_1 > l_2 >$ queries	24
9.11	Other attempted improvements	25
9.11.1	Adjacency list as intermediate data-structure	25
9.11.2	Cached Kleene Star	25
10	Results	25
10.1	Leaderboard progression	25
10.2	Possible improvements	25
11	Part 2: Conclusion	26

1 Introduction

When a user queries a database, they cannot be expected to write queries that are efficient to process. However, efficiency of query processing is essential for overall database performance. Hence, we would like the system to choose a query evaluation plan which is the most efficient out of many different possible strategies for processing a given query. This process is called query optimization.

Cardinality estimation plays a central role in query optimization. The time it takes to process a query plan depends on the size (or cardinality) of the intermediate results produced by the query. Generally, it is not possible to compute exact cardinalities without actually executing the query. As such, these cardinalities must be approximated in order to estimate the cost of a query plan. This is done using statistical methods such as sampling and histograms. The problem of estimating sizes accurately is called cardinality estimation [1].

While we would like to be able to choose the optimal query evaluation plan for any given query, achieving this can prove difficult in practice. As mentioned, the only way to be sure of the cardinalities involved in a query plan is to execute said plan. The problem with that approach is the possibility for unfortunately-chosen query plans to take a long time to evaluate, which can take a toll on database performance. In line with this idea, the objective of cardinality estimation is not to find the optimal query plan in every instance, but to avoid the worst plans [2]. Of course, in order to actually benefit, this determination should take significantly less time than the execution of the query itself. As such, a considerable part of the challenge in cardinality estimation is to find a balance between speed and accuracy.

In this document, we describe how we apply cardinality estimation in the context of project **quickSilver**. The project involves graph databases where information about the edges between vertices is stored in the form of triples of the form subject-predicate-object. Such databases are commonly known as triplestores (or RDF stores). The differences between relational databases and triplestores influence decisions and assumptions we make in our implementation of a cardinality estimator. In RDF systems, the cardinality estimator is an essential component. However, designing such an estimator is a challenging task, in large part because queries over graph databases involve many joins [3]. Join estimation is one of the main sources of inaccuracy. The assumptions that estimation formulas are based on are unlikely to be fully correct in practice, which causes errors in the resulting estimates. Furthermore, these errors propagate through sequences of multiple joins, further compounding the problem. This document describes how we confront this challenge in our implementation.

2 Context and Motivation

While cardinality estimation in relational databases is well-documented in literature, in project **quickSilver** we are dealing with specifically RDF databases. Work on this subject is much more limited. Combined with the timeframe imposed by the project, this lead us to focus mainly on three specific works that we deemed especially relevant in our literature research.

G. Cormode et al. describe the concept of synopses in the context of cardinality estimation in general [4]. They categorize synopses into samples, histograms, wavelets and sketches. This work contains broadly useful information regarding the use of synopses.

T. Neumann and G. Moerkotte introduce a novel synopsis with the concept of characteristic sets, as well as methods of cardinality estimation that center on this synopsis [5]. The suggested methods are tailored towards RDF databases such as the ones used in project **quickSilver**.

The work of N. Yakovets describes **Waveguide**, a query optimizer for triplestores based on regular path queries [6]. Since there is a close match between the types of databases Waveguide is designed for and the ones in project **quickSilver**, this work provides highly relevant information for our purposes. Indeed, our cardinality estimator implementation is based largely on the synopses suggested in the section on cardinality estimation in that work.

3 Plan

To estimate cardinalities without perfect information about the (intermediate) query results involved, we make assumptions from which we can build a statistically sound estimator. Firstly, we assume independence between the data distributions of individual attributes in a relation. Furthermore, we assume uniformity, under which each distinct value for an attribute occurs equally often. In relational databases, an additional assumption is usually made. When joining two tables, the set of values in the join column with the smaller column cardinality is assumed to be a subset of the set of values in the join column with the higher column cardinality. This assumption is known as the containment assumption. Based on these three assumptions, we can construct a formula for the result size estimation of a join operation.

While containment is a reasonable assumption in relational databases, where tables are usually joined on primary keys, it is not useful in RDF databases. For this reason, in the context of project **quickSilver**, we attempt to improve our estimations by lifting the containment assumption. To this end, we implement an approach based on the synopses and methods put forward by N.Yakovets [6]. The notion of "forward" and "backward" edges used in the **quickSilver** graphs and queries was expected to pose a challenge, as we needed to construct additional synopses to account for this property. Furthermore, regular path queries on the **quickSilver** databases can include Kleene-star sub-paths, for which it is difficult to perform accurate cardinality estimation.

As a first step, we implement a basic cardinality estimator using the natural-join estimation formula built upon the assumptions of independence, uniformity, and containment. We refer to this initial version as our naive estimator. Using this as a baseline, we attempt to improve our results by lifting the assumption of containment in subsequent versions.

In our second version, we implement estimation based on the synopses referred to by Yakovets as SYN1 and SYN2. This improves our cardinality estimates by lifting the assumption of containment, as this assumption is unreasonable in the context of the RDF stores used in **quickSilver**.

In the third version, we further tailor the estimator to the specific type of database queries found in project **quickSilver**. In particular, we attempt to obtain better estimations for regular path queries with alternating backward and forward edges. To this end, we construct a third synopsis to store data relevant to such queries. In combination with the information captured in the other two synopses, we can achieve accurate cardinality estimations for any combination of forward and backward edges.

In future work, we would like to estimate cardinalities for regular paths involving Kleene-star operations more accurately. Furthermore we could attempt to find ways to lift the other two assumptions: independence and uniformity. The approach of Neumann and Moerkotte based on the concept of characteristic sets is built to (partially) lift the independence assumption, so an implementation of their ideas could be useful in this regard.

4 Progress Discussion

This section contains a discussion of the progress made towards the implementation of a cardinality estimator. Every stage introduces additional improvements that lead to increasingly accurate estimations.

4.1 Implementation Basis

First, we describe the foundational ideas our cardinality estimator implementation is built upon.

We represent a given query as a simple regular path query, which is a triple $(?s, \langle p \rangle, ?t)$. Here, $?s$ and $?t$ are source and target variables respectively, and $\langle p \rangle$ is a simple regular path expression. Path expression $\langle p \rangle$ is either a label with an associated operation or a concatenation (represented by a $"/"$) of multiple such labels and operations. Labels in a graph are represented as non-negative integers coming from a contiguous domain $[0, \dots, |L|)$. The operation comes from the set of operations $\{ ">",$

"<", ">", "+"}, where the former two operations denote respectively forward and backward edges and the latter represents the Kleene star. [7].

We consider a query as a tree of operations. Every leaf in the tree is a simple regular path expression: a label together with ">", "<" or "+" operation. If the node is not a leaf, then it is a concatenation "/" operation.

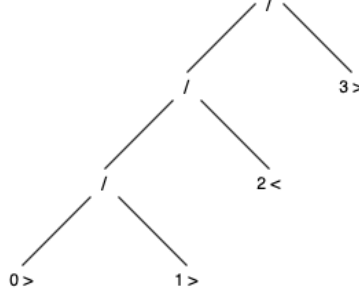


Figure 1: Example of a query tree

We recursively calculate the cardinality estimation of the tree. If a node is a leaf, we calculate the size approximation of a regular path expression stored in the leaf. For an inner node, which represents a concatenation of, we recursively estimate cardinalities of the left and right subtrees before estimate the result size of the concatenation between them. Left- and right-deep trees have useful properties with regard to the synopses we use in our implementation: one side of a concatenation is always a leaf containing a label-operation combination for which our synopses contain accurate statistics [6]. In our implementation, we opt to use a left-deep tree. The way we perform estimation of the result of each operation depends on the implementation version. Detailed descriptions of these versions and synopses can be found in the Progress Discussion section.

In order to express estimations for queries and intermediate results, we use a data structure **cardStat** that has three members: **noOut**, **noPaths** and **noIn**. **noPaths** indicates the number of (s, t) pairs in the evaluation result, **noOut** is the number of distinct values of s and **noIn** denotes the number of distinct values of t in the evaluation result [7].

4.2 Naive Estimator

Our first leaderboard entry was a basic implementation of a cardinality estimator based on three assumptions: independence, uniformity, and containment. This design is described as the classical approach in the work of Yakovets, as well as in the presentations accompanying project **quickSilver** [6, 2].

For this version of the cardinality estimator we gather the following statistical information:

- **numPairs** is the total number of unique pairs (s, t) ;
- **numLabels** is the number of distinct labels in the graph;
- **distinctFromNodes** is the number of distinct s nodes in the graph;
- **distinctToNodes** is the number of distinct t nodes in the graph.

We calculate **cardStat** result differently depending on the data stored in the tree leaf. More specifically, the implementation varies depending on the operation $\{ ">", "<", "+" \}$, but not the label itself. In this implementation, specific labels have no influence due to the uniformity assumption. Under this assumption, we can approximate the number of entries per label by simply dividing the total number of triples by the number of distinct labels. Thus, distinguishing only by the operation, we consider the three cases given below.

1. Forward operation (" $>$ ")

- (a) **noOut**: $\text{distinctFromNodes} / \text{numLabels}$
 - (b) **noPath**: $\text{numPairs} / \text{numLabels}$
 - (c) **noIn**: $\text{distinctToNodes} / \text{numLabels}$
2. Backward operation (" $<$ ")
- (a) **noOut**: $\text{distinctToNodes} / \text{numLabels}$
 - (b) **noPath**: $\text{numPairs} / \text{numLabels}$
 - (c) **noIn**: $\text{distinctFromNodes} / \text{numLabels}$
3. Kleene star operation (" $+$ ") For this version of implementation, a (very) rough approximation of " $+$ " is taken to be exactly the same as " $>$ " operation.

If a node is not a leaf, it contains a concatenation "/" operation. In this case, we are interested in approximating the concatenation results of the left and right subtrees. Consider the definition of "/":

$$\text{If } p = p_1/p_2 \text{ then } p(G) = \{(s,t) \mid \exists m \in N (s,m) \in p_1(G) \wedge (m,t) \in p_2(G)\} \text{ [7]}$$

From the definition, we can observe that concatenation of two subtrees is similar to a natural join on a common attribute. Hence, we can consider the formula for the result size estimation of a join:

$$|R \bowtie S| \approx \min\{T_R \frac{T_S}{V(S,Y)}, T_S \frac{T_R}{V(R,Y)}\}$$

This formula is described step by step in the work of N. Yakovets [6], and presented in the lecture slides [2]. It is based on the three assumptions mentioned before: independence, uniformity, and containment. In this formula, T_R is the size of R and T_S is the size of S . $V(R,Y)$ is defined as the number of distinct values appearing in attribute Y of relation R . Similarly, $V(S,Y)$ is defined as the number of distinct values appearing in attribute Y of relation S . On the one hand, each tuple of R joins approximately $\frac{T_S}{V(S,Y)}$ tuples of S ; on the other hand, each tuple of S joins approximately $\frac{T_R}{V(R,Y)}$ tuples of R . In order to minimize the final contribution, the smallest result is taken.

When applying this formula to our specific situation, R is the result of the left subtree, S is the result of the right subtree. Accordingly, T_R is the size of R and T_S is the size of S . $V(R,Y)$ in our case corresponds to the **noIn** statistics of the result of the left subtree. Similarly, $V(S,Y)$ corresponds to the **noOut** statistics of the result of the right subtree. To sum up, when applied to our case, the formula for cardinality estimation of concatenation "/" becomes as follows:

$$\text{noPaths} \approx \min\{\text{left.noPaths} \cdot \frac{\text{right.noPaths}}{\text{right.noOut}}, \text{right.noPaths} \cdot \frac{\text{left.noPaths}}{\text{left.noIn}}\}$$

Note that the given above formula only evaluates the size of the concatenation operation; that is, it produces the result for **noPaths**. However, the **cardStat** structure also contains **noOut** and **noIn**. To find these values, we apply a reduction factor. We again assume uniformity, meaning that the numbers of unique **s** and **t** decrease proportionally to the change in **noPaths**. Reduction factor (RF) and the remaining statistics are computed as follows:

$$\begin{aligned} RF &= \frac{\text{noPaths}_{\text{new}}}{\text{noPaths}_{\text{old}}} \\ \text{noOut}_{\text{new}} &= RF \cdot \text{noOut}_{\text{old}} \\ \text{noIn}_{\text{new}} &= RF \cdot \text{noIn}_{\text{old}} \end{aligned}$$

Whenever we estimate the cardinality of a query, we assume that source and target variables are always free. If this is the case for the query, we return the result of our estimation. Otherwise, if either s or t are constants, we scale our estimation by multiplying the result by reduction factor $\frac{1}{\text{distinctFromNodes}}$ or $\frac{1}{\text{distinctToNodes}}$, respectively. When the source is bound, we set **noOut** to be 1, since the only possible value for the source node is the node defined in the query as a constant. Similarly, when the target is bound, **noIn** is set to 1.

4.2.1 Results

Based on the leaderboard results, this method of cardinality estimation has shown to be much more accurate on the synthetic database than on the real database. The relative position on the leaderboard, as shown on Figure 2, indicates that this approach was reasonable for a first implementation.

user_id	acc_syn	acc_real	prep_time	est_time	peak_memory	score
h.a.melchers	8.68	49.70099999999999	19710.15	0.12496000000000002	24.07	151.88714199999998
koen	159.71	54.039	608.5400000000001	0.34630000000000005	24.81	293.2758
andrei.agaronian	159.71	54.039	887.784	0.33919	25.51	294.253622
j.w.f.m.c.v.d.looi	4.34	224.359	25221.581	0.3818000000000001	24.6	502.95594100000005
a.zioul	227.74	439.633	184.971	3.791889999999999	24.07	1132.0193489999997
j.s.reinders	3.93	2350.536	1348.961	0.21877000000000002	24.07	4730.464715
r.wichertjes	4.09	3304.0029999999997	21962.706000000002	14.02667	24.67	6661.53404
k.mankevic	68.63	3427.1870000000001	127.53599999999997	0.06916	24.07	6947.215368000001
r.m.jonker	276.71	3748.024	7295.107999999999	0.17843999999999996	25.27	7805.358796
l.a.castelijns	332.81	1999.5839999999996	0.0	18136.83109	24.07	7983.414217999999
j.e.v.bolta	2831.31	2837.8779999999997	1068.154	0.1026	24.07	8532.224674
s.e.d.vegt	49.43	6466.112	68606.836000000001	3.8358800000000004	24.89	13075.918012
jasperstam1	2013.39	5659.012	596.4409999999999	0.5290999999999999	24.07	13356.186260999999
b.m.mevius	33.28	7312.4820000000001	10026.078	103.78224999999999	25.43	14714.456528000004
m.h.hagedoorn	7048.73	4905.182999999999	49.74	390.5102	24.52	16961.76778
j.l.g.schols	23601.66	1100.7810000000002	180.709	0.14842000000000002	7.25	25810.682393000003
m.v.d.horst	23601.66	1114.367	2527.819	0.13072	7.25	25840.197963
s.a.tanja	21533.48	3999.0200000000004	6808.7649999999985	0.1519	25.27	29563.629145000003
g.c.v.wordragen	13.44	21357.730000000003	99908.92799999999	0.19552	161.09	42989.938032000005
david.tuin	28312.75	28329.445999999996	133.684	2.1367399999999996	24.07	84996.273032
i.geenen	28649.19	28658.484999999997	0.0	0.00834	24.07	85990.231668
hush	28649.19	28658.484999999997	0.0	0.008570000000000001	24.07	85990.231714
l.donders	28649.19	28658.484999999997	0.0	0.00903	24.07	85990.231806
e.e.g.a.samuels	28649.19	28658.484999999997	0.0	0.009120000000000003	24.07	85990.231824
t.uitedwillegen	28649.19	28658.484999999997	0.0	0.009610000000000002	24.07	85990.23192199999
s.veretennikov	28649.19	28658.484999999997	0.0	0.00966	24.07	85990.231932
c.a.smits	3.18	1580725.6609999998	92.375	0.21383999999999995	24.07	3161478.707143

Figure 2: Results of naive estimator (presented by k.mankevic)

4.3 Estimator using SYN1 and SYN2

For the second version of our cardinality estimator, we summarize properties of the subject database in synopses SYN1 and SYN2 as described by Yakovets [6] and also used in [8]. We introduce these synopses in order to improve accuracy of cardinality estimations of concatenations of the form $T_{r/l_1/l_2}$, which are all node pairs (s, t) where a path between them conforms to concatenation $r/l_1/l_2$. Here, l_1, l_2 are edge labels and r is some regular expression. Comparing to the naive estimator, here we try to lift the assumption of containment. [6].

For SYN1, we gather the following statistics for each label l in the graph:

- **out** is the total number of nodes that have outgoing edge labeled with l ;
- **in** is the total number of nodes that have incoming edge labeled with l ;
- **path** is the total number of paths labeled with l .

For SYN2, we gather the following statistics for each unordered pair of labels (l_1, l_2) :

- **middle** is the total number of nodes number that have incoming edge labeled l_1 and outgoing edge labeled l_2 ;
- **in** is the total number number of nodes that have incoming path labeled with l_1/l_2 ;
- **two** is the total number of paths labeled l_2 from nodes in middle to nodes in **in**.

From here, we can estimate the cardinality estimation of $T_{r/l_1/l_2}$ [6].

$$|T_{r/l_1/l_2}| = |T_{r/l_1}| \cdot \frac{l_1/l_2.\#two}{l_1.in}$$

$$d(s, T_{r/l_1/l_2}) = d(s, T_{r/l_1}) \cdot \frac{l_1/l_2.middle}{l_1.in}$$

$$d(o, T_{r/l_1/l_2}) = d(o, T_{r/l_1} \cdot \frac{l_1/l_2.in}{l_1.in})$$

When applied to our case, the formulas become as follows:

$$noPaths = left.noPaths \cdot \frac{syn2.two}{syn1.in}$$

$$noOut = left.noOut \cdot \frac{syn2.middle}{syn1.in}$$

$$noIn = left.noIn \cdot \frac{syn2.in}{syn1.in}$$

Similarly to the naive estimator we assume that source and target variable are always free. For the cases where s or t are bound, we scale by reduction factors as previously described in 4.2.

4.3.1 Kleene star

Estimating the cardinality of transitive closure is a difficult problem. This makes estimation for the Kleene-star operation challenging. The basic idea we had was based on the following observation

$$TC(G) = \bigcup_{l=1}^{\infty} P_l(G)$$

Where $P_l(G)$ is simply the set of nodes in the graph G reachable by a path of length l . If $(u, v) \in P_2(G)$, then $\exists n_1, n_2 \in N$ such that $(u, n_1), (n_1, n_2), (n_2, v) \in G$.

We can implement this formula directly up to a threshold T for cardinality estimation (recursively):

$$|TC_T(G)| = |TC_{T-1}(G) \cup concat(TC_{T-1}(G), P_1(G))|$$

The union operation is from [2]:

$$|A \cup B| = \begin{cases} |A| + \frac{|B|}{2} & \text{if } |A| > |B| \\ |B| + \frac{|A|}{2} & \text{else} \end{cases}$$

This estimation has considerable issues. Firstly, it is unclear which threshold T should be used. Our implementation uses $T = 3$ arbitrarily. We considered computing the average path length for each label and use that as threshold. However, such a calculation could be computationally expensive, and the result would be used with a less-than-perfect union estimation. We decided that the accuracy gain was unlikely to be worth the speed loss. Another problem is that the union estimation does not take care of duplicates, which could be quite important in graph data bases (u can reach v through different paths of various lengths). Furthermore, at this stage of development there were other areas where relatively large improvements could be made. For these reasons, we decided to defer more accurate Kleene-star estimation to future work.

4.3.2 Results

The implementation based on the SYN1 and SYN2 produced the following result as can be seen on the leaderboard on [Figure 3](#).

user_id	acc_syn	acc_real	prep_time	est_time	peak_memory	score
g.c.v.wordragen	8.53	12.953	19639.820000000003	0.1649	26.45	80.5588
a.zioul	16.6	20.830000000000002	11956.476999999999	0.18208	24.07	94.322893
h.a.melchers	8.68	49.709999999999994	649.2049999999998	0.1222200000000001	26.09	134.86364899999998
j.s.reinders	3.92	20.259	67547.403	0.2219	24.07	136.099783
s.e.d.vegt	30.78	44.242000000000004	113.86500000000001	5.135480000000001	24.07	144.474961
koen	4.84	72.99	3670.1140000000005	35.60559000000001	28.8	190.411232
thorn.prive	5.29	114.747	1543.8050000000003	115.11583999999996	28.0	287.35097299999995
giordano3102lucas	3.18	215.613	817.6759999999999	0.10565	24.07	459.31480600000003
andrei.agaronian	4.96	213.706	3553.3199999999997	35.609179999999995	28.81	471.8571559999999
j.w.f.m.c.v.d.looij	4.34	224.315	8533.149000000001	0.4235499999999999	24.6	486.18785899999995
r.wichertjes	18.22	224.89900000000003	130643.86799999999	0.30944	24.67	623.3937559999999
b.enache	1.01	1.1099999999999999	0.0	3087.19764	24.07	644.73952800000001
m.padhi	49.06	475.946	89.84300000000002	0.5693599999999999	24.37	1025.525715
l.a.castelijns	358.0	389.767	0.0	718.3079099999999	24.07	1305.265582
n.g.m.uphoff	24.04	1209.5490000000002	184.19099999999997	0.6958000000000001	24.37	2467.8313510000003
r.m.jonker	276.71	3748.024	7295.107999999999	0.17843999999999996	25.27	7805.358796
j.e.v.bolta	2831.31	2837.8779999999997	1068.154	0.1026	24.07	8532.224674
jasperstam1	2013.39	5659.012	621.137	0.53707	24.07	13356.212550999999
b.m.mevius	33.28	7312.4820000000001	10026.078	103.78224999999999	25.43	14714.456528000004
m.h.hagedoorn	5831.71	5140.764	49.95	391.93201	24.52	16216.194352000002
m.j.j.mandemakers	4.26	8786.357	677.674	40.099390000000001	24.07	17609.741551999996
k.mankevic	21319.59	271.349	1378.662	0.13228	24.07	21887.763118
nolmoonen	21615.36	311.64500000000001	14437.452000000001	0.18227	24.07	22277.193906
r.taki	21357.65	1157.194	169.368	0.20356	24.07	23696.31808
s.a.tanja	21600.95	1232.731	7070.865000000001	0.20307999999999998	25.27	24098.793481
i.geenen	28649.19	28658.484999999997	0.0	0.00834	24.07	85990.231668
l.donders	28649.19	28658.484999999997	0.0	0.00903	24.07	85990.231806
e.e.g.a.samuels	28649.19	28658.484999999997	0.0	0.009120000000000003	24.07	85990.231824

Figure 3: Results of SYN1, SYN2 estimator (presented by k.mankevic)

We can see that current cardinality estimator's score went down compared to the naive estimator discussed in section 4.2. The reason why the overall score dropped is due to significant decrease in accuracy on synthetic workload. However, it must be noticed that the accuracy improved from 3427.1 to 207.0 on the real workload. It is hard to say whether this version improved the quality of the estimator because the results vary.

It could be the case that the current version performed worse because of some implementation-specific details. For example, the estimation of Kleene-star operation is still roughly approximated (same as ">" operation). Additionally, as we found out only later, the graph databases contain duplicates, which were not accounted for in the current version. We do not know the exact amount of duplicates, but this fact could seriously worsen the performance. Duplicates are only taken care of in the next implementation version.

4.4 Estimator using SYN1, SYN2 and SYN3

The combination of SYN1 and SYN2 gives relevant statistics for all queries involving a sequence of either only forward edges (with operation $>$) or only backward edges. As soon as we started mixing $<$ and $>$ operations in the same query we encountered the limitations of these synopses. In an attempt to find solutions to our newfound problems, we researched *characteristic sets* [9] as suggested in the project description. The star join referred to in the relevant work is related to cardinality estimation for the types of queries we encounter here. However, since we were already working with synopses entirely unrelated to characteristic sets, we decided to try to find a solution within the framework we were currently working in. This lead us to the idea of constructing a third synopsis (SYN3) to handle paths with alternating operations. As the construction of SYN3 is based on the same premises used by Yakovets [6] to build SYN2, we use the same notation and argumentation similar to those found in his work.

We start by observing that performing a $<$ operation can be understood as reversing the edges of the graph. This means that a query of the form $l1 > /l1 <$ looks for path $l1/\tilde{l}2$ in the graph, where \tilde{l} is the

reversed edge of l . For any given edge (s, l, t) , we can construct its reverse has follows: (t, l, s) . After this observation, we realize that the result of $l1 > /l1 <$ is

$$\{(s, t) \mid \exists m \in N \text{ such that } (s, l1, m) \text{ and } (t, l2, m) \text{ both } \in G\}$$

This means we are not joining on middle nodes that have incoming edge $l1$ and outgoing $l2$, but rather on middle nodes that have two incoming edges labeled $l1$ and $l2$. This idea is summarized in figure 4.

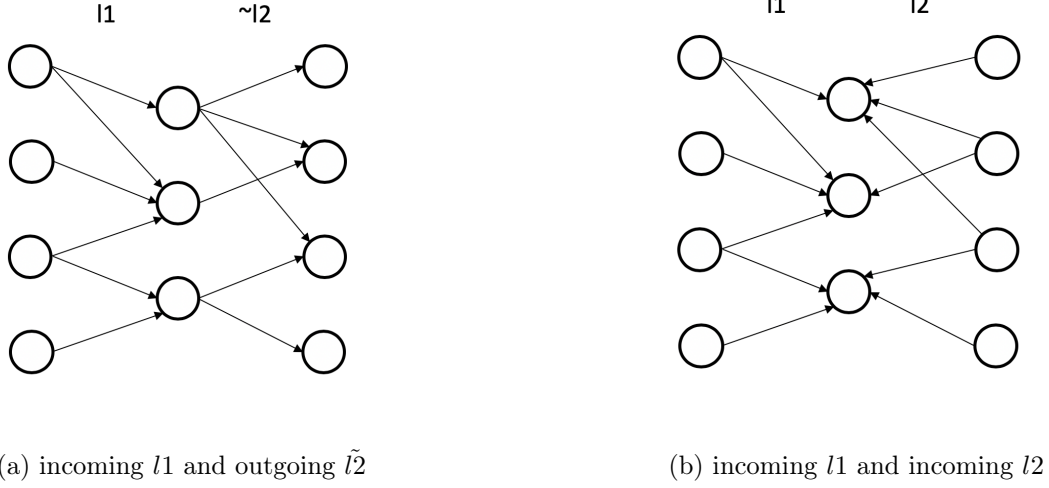


Figure 4: Equivalent figures

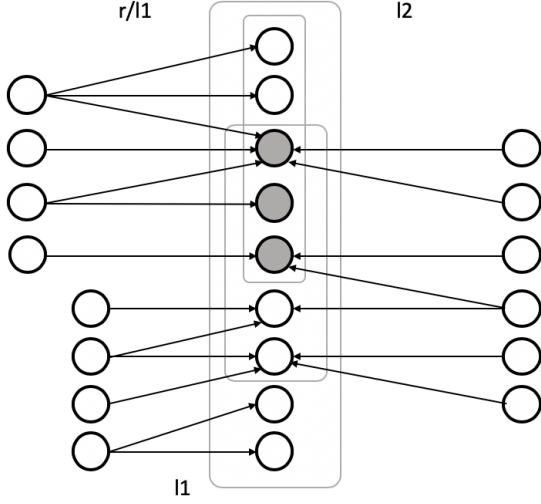
Since this construction does not match up completely with the one SYN2 is built upon, we cannot use only the existing synopses to estimate cardinalities for such queries. For this reason, we store in SYN3 the following statistics for each unordered pair of labels $(l1, l2)$:

- **in** is the total number of nodes in G which have incoming edge labeled $l1$ and incoming edge labeled $l2$
- **twoIn** is the number of paths from nodes in $syn1.out$ to nodes in $syn3.in$
- **out** is the number of nodes in G which have outgoing edge labeled $l1$ and outgoing edge labeled $l2$
- **twoOut** is the number of paths from nodes in $syn1.in$ to nodes in $syn3.out$

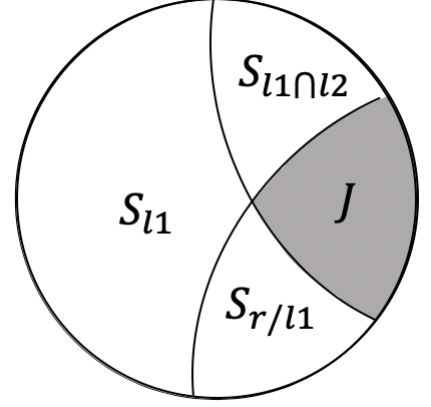
We will start by applying formula (6.8) from the work of Yakovets [6]. For two arbitrary tables T_1 and T_2 , we can estimate the size of the join of those tables on the common attribute $T_1.o = T_2.s$, using uniformity as follows:

$$|T_1 \bowtie_{T_1.o=T_2.s} T_2| = |J_{T_1, T_2}| \cdot \frac{|T_1|}{V(T_1, o)} \cdot \frac{|T_2|}{V(T_2, s)}$$

We try to estimate the size of the join set J_{T_1, T_2} in our query case $(*, l1 > /l2 <, *)$. Let $T_1 = T_{r/l1}$ where $T_{r/l1}$ refers to a table of node pairs (s, o) such that path between s and o in G that conforms to regular expression $r/l1$ and let $T_2 = T_{l2}$ where T_{l2} is a table all node pairs in G such that a path between them conforms to $l2$. Then $T_{r/l1} \bowtie_{T_1.o=T_2.o} T_{l2}$ would produce the result of the query $T_{r/l1/\tilde{l}2}$ that we need.



(a) Join $T_{r/l1} \bowtie_{T1.o=T2.o} T_{l2}$



(b) Partition of join candidate nodes

Figure 5: Estimating join cardinality using synopsis 3

Using SYN1 and SYN3, we can estimate the cardinality of $T_{l1/\tilde{l}2}$. First, let S_{l1} be the set of all nodes having incoming edge $l1$ and $S_{r/l1}$ the set of nodes that have incoming path $r/l1$. We also define a set $S_{l1 \cap l2}$ as the set of all nodes having both incoming edge $l1$ and incoming edge $l2$. This is illustrated in figure 5. Using those notations, analogously to the construction of SYN2, the join set J becomes $S_{l1 \cap l2} \cap S_{r/l1}$. The next step is to compute $E(|J|)$. In order to do so, we need to compute $P[x \in J]$ for a given $x \in S_{l1}$. Using the independence assumption, we have:

$$P[x \in J] = P[x \in S_{l1 \cap l2} \cap S_{r/l1}] = P[x \in S_{l1 \cap l2}] \cdot P[x \in S_{r/l1}] = \frac{|S_{l1 \cap l2}|}{|S_{l1}|} \cdot \frac{|S_{r/l1}|}{|S_{l1}|}$$

We can now assume that the random variable $|J| \sim B(|S_{l1}|, P[x \in J])$ (Bernoulli distribution) since by the uniformity assumption all nodes $x \in S_{l1}$ have the same probability to belong to $|J|$ and the trials are assumed to be independent. We now have everything we need to compute $E(|J|)$.

$$E(|J|) = |S_{l1}| \cdot P[x \in J] = \frac{|S_{l1 \cap l2}| \cdot |S_{r/l1}|}{|S_{l1}|}$$

We now approximate $|J|$ by its expected value and plug this last equation into equation (6.8) to obtain our cardinality estimation.

$$|T_{r/l1/\tilde{l}2}| = |T_{r/l1} \bowtie_{T1.o=T2.o} T_{l2}| = \frac{|S_{l1 \cap l2}| \cdot |S_{r/l1}|}{|S_{l1}|} \cdot \frac{|T_{r/l1}|}{V(T_{r/l1}, o)} \cdot \frac{|T_{l1/\tilde{l}2}|}{V(T_{l1/\tilde{l}2}, o)}$$

Observe that $|S_{l1}| = \text{Syn1.in}$, $|S_{l1 \cap l2}| = \text{Syn3.in}$ and $|S_{r/l1}| = V(T_{r/l1}, o)$. This observation allows us to compute the estimation using our synopsis statistics:

$$|T_{r/l1/\tilde{l}2}| = |T_{r/l1}| \cdot \frac{\text{Syn3.in}}{\text{Syn1.in}} \cdot \frac{|T_{l1/\tilde{l}1}|}{V(T_{l1/\tilde{l}1}, o)} = |T_{r/l1}| \cdot \frac{\text{Syn3.twoIn}}{\text{Syn1.in}}$$

Until now, we only described estimation for the case of a query having the form $(*, l1 > /l2 <, *)$, but the case $(*, l1 < /l2 >, *)$ is similar. Indeed, instead of considering the nodes that have both incoming

edges $l1$ and $l2$, we should consider those who have both outgoing edges $l1$ and $l2$. Following a similar argument, we obtain the following:

$$|T_{r/\tilde{l}_1/l_2}| = |T_{r/\tilde{l}_1}| \cdot \frac{Syn3.out}{Syn1.out} \cdot \frac{|T_{l_2/\tilde{l}_1}|}{V(T_{l_2/\tilde{l}_1}, s)} = |T_{r/\tilde{l}_1}| \cdot \frac{Syn3.twoOut}{Syn1.out}$$

4.4.1 Results

Looking at the leaderboard results after implementation of SYN3, we see a few interesting differences compared to our previous version. First and foremost, estimation accuracy on the synthetic data set has improved substantially. Indeed, at the time of submission, our `acc_syn` score was among the best on the leaderboard. While we attribute most of this improvement to the addition of SYN3, we also fixed implementation issues mentioned in 4.3.2, which presumably had an impact on performance as well. Another difference can be seen in the preparation time. It seems that the statistics in SYN3, which are gathered during preparation, are more computationally expensive to extract than those in the other two synopses. By contrast, our estimation time is very low. At the time of submission, our estimations are the fastest on the leaderboard by a decent margin. As we have no insight in the estimator implementations of other groups, we can only hypothesize about the reason for this difference. Our assumption is that they use entirely different approaches and/or synopses, involving more expensive calculations for the nodes of their tree of operations. Interestingly, our accuracy for the real dataset improved only slightly between versions. Unfortunately, at present we do not know the exact reason for this relatively small gain. Putting aside the metrics used in the leaderboard, we found during testing that the estimations for paths with alternating operators were still not as good as the ones for paths consisting of either only forward or only backward edges. This is an area which could be improved in future versions.

user_id	acc_syn	acc_real	prep_time	est_time	peak_memory	score
ugne.ciziute	3.92	20.249000000000002	7308.521	0.2282000000000001	24.07	75.842161
g.c.v.wordragen	8.67	26.437	2722.123	0.1470800000000002	26.53	90.825539
a.zioual	14.54	20.87	11780.949999999999	0.1873099999999998	24.07	92.16841200000002
h.a.melchers	8.68	49.70099999999999	657.276	0.15542	26.09	134.86036
s.e.d.vegt	30.78	44.242000000000004	118.993	5.239649999999999	24.07	144.500923
y.j.a.scheepers	12.02	59.23699999999999	30180.719999999998	0.1798600000000002	24.07	184.78069199999996
koen	4.84	72.99	3670.1140000000005	35.60559000000001	28.8	190.411232
noimoonen	12.07	59.515	44713.11	0.2173199999999999	23.05	198.906574
d.m.serban	1.03	2.0279999999999996	0.0	1095.6547099999998	24.07	248.28694199999998
t.e.w.bertens	12.07	59.513	53410.116	0.20303	65.79	250.336722
jasperstam1	9.1	108.524	776.796	5.934560000000001	24.25	252.361708
n.g.m.uphoff	13.33	111.38499999999999	2994.6859999999997	1.7401600000000002	26.67	266.112718
thorn.prive	5.29	114.747	1543.8050000000003	115.11583999999996	28.0	287.35097299999995
r.m.jonker	19.58	122.38499999999999	2765.782	0.2596900000000003	24.72	291.88771999999994
s.a.tanja	19.58	122.38499999999999	2814.9	0.2456199999999998	24.72	291.934024
lx.leixue	92.31	97.35900000000001	155.71200000000005	0.19109	24.44	311.66193
r.wichertjes	5.28	139.448	35967.939000000006	0.25967	24.67	344.865873
j.s.reinders	12.81	150.613	7145.25	0.6401799999999999	26.6	347.909286
david.tuin	1.34	173.052	18334.614999999998	3.13115	31.24	397.644845
j.l.g.schols	64.8	165.393	8609.850999999999	0.26306	24.37	428.618463
m.v.d.horst	64.8	165.393	9856.258	0.4634500000000003	24.37	429.90494800000005
andrei.agaronian	4.96	213.706	3553.3199999999997	35.609179999999995	28.81	471.8571559999999
giordano3102lucas	3.14	216.878	14827.107000000002	0.08565	24.07	475.810237
j.w.f.m.c.v.d.looij	4.34	224.315	8533.149000000001	0.4235499999999999	24.6	486.18785899999995
w.j.v.leeuwen	13.7	230.786	16868.828999999998	0.3703900000000005	24.07	516.284907
s.smeters	3.46	250.43099999999998	27680.569	0.2677800000000001	24.07	556.126125
wouternuijten	82.67	230.425	27852.12	0.3834800000000015	24.07	595.518816
k.mankevic	54.4	280.681	1428.913	0.1506600000000002	24.07	641.2910449999999
b.enache	1.01	1.1099999999999999	0.0	3087.19764	24.07	644.7395280000001
l.g.t.v.schooten	420.6	33.916	52.119	693.36981	24.57	651.7280810000001
r.taki	96.71	275.009	2945.6420000000007	0.49419	24.07	673.8424800000001
robvangastel	61.65	275.32300000000004	83814.367	53.376839999999994	28.1	734.8857350000001
j.e.v.bolta	33.49	342.92099999999994	1474.0600000000002	22.164360000000002	24.23	749.4689319999999
m.padhi	26.61	1326.344	184.773	1.1242700000000003	24.37	2704.077627

Figure 6: Results of SYN1, SYN2, SYN3 estimator (presented by giordano3102lucas)

5 Part 1: Conclusion

We implemented a cardinality estimator for the RDF databases that are the subject of project **quickSilver**. This estimator was iteratively improved through addition of synopses that allowed us to lift the containment assumption, one of three assumptions commonly made in cardinality estimation for relational databases. At the time of writing, our implementation had some interesting properties compared to others' implementations of cardinality estimators for **quickSilver**. We obtained relatively good accuracy on a synthetic data set, as well as top-tier estimation speeds. However, our estimation speed was somewhat offset by a relatively long preparation time.

In future work, we will aim to improve cardinality estimation for the Kleene-star operation. Furthermore, we will attempt to further improve accuracy of estimations involving paths with alternating operators, which is currently not at the level of our other estimations. In order to further increase accuracy, we could search for ways to lift the assumptions of independence and uniformity. As the characteristic set approach [9] manages to deal with independence to great results, this is an especially promising area of improvement.

6 Part 2: Strategy

In the first part of the project we worked on the problem of cardinality estimation. In part two of the project we will focus our attention on improving other components of the query pipeline. We categorize the improvements we will make according to the criteria laid out in the assignment description. As such, each improvement is described as making our implementation more Smart, Quick and/or Frugal.

6.1 Smart

For the Smart aspect of implementation, we will mostly aim at improving the query evaluation time by choosing the cheapest evaluation plan. The most significant idea in this category is logical optimization through the use of a dynamic programming algorithm to calculate cheap plans for unbound queries, where neither the source or target variable in the query is bound. For queries that do involve bound variables, we can push down the logical selection operator to the leaves of the query tree, leading to a specific plan structure to handle these types of queries. We also make use of an IndexJoin algorithm that is smart in the sense that it is able to sort its join results in small batches in a pipelined way, which helps distribute the cost of evaluation where possible.

6.2 Quick

The Quick aspect focuses on improving evaluation time by performing physical evaluations of query plans faster. To this end, we will construct an index data structure to facilitate quick lookups of edges. Furthermore, we will attempt to find fast join algorithms suited to the problem at hand. Finally, we will implement some degree of multithreading to further accelerate evaluation.

6.3 Frugal

Regarding the Frugal aspect, we are concerned with making our implementation memory efficient. In order to do this we will make our index data structure as lean and efficient as possible, only storing necessary information. In addition, we will make use of pipelining in order to store fewer intermediate results. We can further reduce memory consumption by removing duplicates from these results.

6.4 Implementation order

First, we will implement the index, the lookup functionality associated with it, and a join algorithm. This will form the core of our approach to the project. We will also start on the infrastructure for pipelining as we do this, since we will build upon this base during the rest of our implementation. In order to make this pipelining possible, we need to implement multithreading as well. When we have a working implementation of the physical operators, we will start coding the dynamic programming algorithm to handle the logical optimization. With both the physical and logical sides intact, we can start incrementally enhancing our implementation by optimizing index and operators, as well as improving evaluation plan construction. These improvements will be made in no particular order, being mostly dependent on the results we obtain and realizations we come to as we work on the project.

7 Step-by-Step Plan

In this section we will present the step-by-step plan we followed during the second part of the project.

As a first step, an initial literature survey was conducted. During this phase, much of the information regarding possible approaches was collected.

After the initial literature research, we knew we wanted to implement some techniques, such as pipelining, for which it was difficult to build upon the given basic implementation. We decided to change the architecture of the project to give us more control over our algorithms and data structures.

Next, we implemented an index. This index is the foundation data structure of the project, and all following parts of the implementation make use of it.

We then started working on the basic physical operators: IndexLookUp and KleeneStar. For our first join algorithm, we followed the guidelines proposed by the teacher during the lectures and arrived at merge join.

At this point we had a working basis for the project, which was naturally followed by logical optimization. In parallel with logical optimization, we were working on modifying the frugal side of an already implemented index data structure.

During the final stages of the project, we reflected on our current implementation. After having iteratively worked on the MergeJoin algorithm, we reached a point where it was difficult to improve further. There was also the issue that MergeJoin is not really compatible with the pipelined approach we envisioned. We decided to explore different join algorithms to help us tackle the large cardinalities found while evaluating queries in practice. We landed on IndexJoin, a fully pipelined physical operator. In general, during the project, we followed an iterative approach of development, going back and forth from the research part to the implementation part, trying different techniques and improvements.

8 Literature Survey

Our first step was to gather information about RDF database engines in general. We thus started by exploring the advised literature for our project[10][11].

Query optimization is a process that generally consists of three main steps, as described in the Database System Concepts book [1] and the slides [2]. First, given a query expression, all equivalent expressions should be generated using equivalence rules. Second, the results are annotated for further generation of alternative plans for each expression. Finally, the cost of each evaluation plan is estimated and the cheapest plan is selected. This procedure is a starting point for understanding the query optimization in this project. In addition, the thesis of G.Mak [8] describes the life of a query in detail, including query optimization with respect to the Telepath graph database. This source is especially valuable for our project, as it gave us a solid understanding of how the query optimization process described in the book can be applied to a graph database in practice. Furthermore, the thesis explains the implementation of a dynamic programming algorithm for generation of physical plans step by step, which turned out to be very useful for building the algorithm for quickSilver.

Where query processing is concerned, our first attempt at a merge join algorithm came from[1]. We further improved it based on the constraints of our project, as will be explained later in this report. The architecture of the project, based on physical plans and operators, was also inspired by[8]. We took some ideas from[12] to understand how queries containing a Kleene star could be evaluated. We derived our pipelined IndexJoin from the idea of a pipelined hash join[13].

Our index is based on a $k - path$ index[14] for $k = 1$ and a subset of all $k = 2$ paths. The thesis[15] on path indexing gave us a good starting point for an understanding of how this could be implemented efficiently.

9 Progress

In this section we demonstrate the implementation process for the second part of the project. We will present techniques in the order of which they were implemented. At each stage we discuss the results achieved after the technique was introduced.

9.1 Index

The first improvement we decided to work on implementing was a problem-specific index to facilitate quick lookups for the specific types of queries we deal with in project quickSilver. We are working in

the context of graph databases, where we are interested in paths consisting of edges. In the Mak thesis on the subject of the Telepath project, we found an interesting concept for a purpose-built solution: k -path indexing.

9.1.1 1-path index

We use the definition of a k -path provided by Mak. Informally, we can say a k -path is a path of length k in a graph written as a sequence of edge labels. By indexing the results of path queries using the label sequence as a key, we can avoid traversing the graph to obtain query results. Theoretically, if we had a complete index for every value of k , we could answer every query by probing the index with the corresponding path and returning the result. Of course, constructing such an index is a resource-intensive task. We quickly encounter a combinatorial explosion of label sequences as we increase k . On top of that, since k is a path length, its maximum value grows with graph size. Clearly, we will not be able to answer all queries through direct index probing. Instead, we will get sub-paths of a query path from the index, after which we join them using a join algorithm. Initially, when we looked at the leaderboard score function, we saw a large multiplier applied to memory usage. Furthermore, during the lectures, the importance of a low memory footprint was emphasized. Another concern with precomputing a large index is that it might take a long time to construct. We note, however, that the scoring function is relatively lenient regarding precomputation time. Since storing a full index for $k = 2$ already seemed memory-intensive, we chose to use a 1-path index. Functionally, a 1-path index is just a different way to store the graph edges. This way of storing edges offers some efficiency compared to the adjacency list found in the basic implementation provided to us in project quickSilver. The adjacency list for a label allocates space in its data structure for each node in the graph, even if a node has no outgoing edges with the corresponding label. On the other hand, a k -path index only uses space for paths (i.e. edges) that actually exist in the graph.

Our first implementation of the index consisted of two arrays of ordered maps (a data structure in the C++ standard library), each with two maps for every label in the graph. One of these maps stores the edge with the source as the key and target as the value. The other stores the edge in reverse, where the target is the key and the source the value. With this construction, we can query the index for both regular and reverse edges, as both types are encountered in the path queries found in project quickSilver. This allows us to return iterators for bound queries (i.e. where either the source or the target variable of the query is bound) in logarithmic time. Another benefit of this path index is that there is no need to explicitly associate store the label for an edge. This information is implicit from the array index instead.

9.1.2 Edge type and intermediate data-structure

To facilitate pipelining, we decided to represent the intermediate graph data structure as a simple flow of edges ($struct \rightarrow Edge\{Node, Node\}$) produced by a child operator and consumed by the parent operator. If we were dealing with a large database schema with more columns than a 3-integer RDF store, it would make sense to store the data in a data-store, only passing pointers to the data (rather than the data itself) from children to their parent operators. However, since a pointer is a 64 bit integer in modern architectures, and both source and target also require 64 bits ($2 \cdot 32$ integers), we can simply directly pass edges instead.

9.2 Physical Operators

9.2.1 Abstract Class

PhysicalOperator is an abstract class which provides an interface to all physical operators used in this project, such as MergeJoin or KleeneStar. A Physical Operator has a left child and a right child, and it is used to represent a physical plan which is constructed for query evaluation. Internal nodes in our logical evaluation plan are handled with MergeJoin or IndexJoin. As for the leaves, we have Kleene star and IndexLookup, which are operators that probe the index.

9.2.2 Pipelining

As stated in earlier, we needed to build a custom architecture to be able to implement pipelining. Our implementation uses a pull-based approach, where the parent operator must call the *produceNextEdge* method on its children to obtain the results from their output queues. To achieve the desired behaviour, we stored in each *PhysicalOperator* an output queue which is concurrent and synchronous (blocking). Thus, the call of *produceNextEdge* can put the parent thread in a sleeping state if the output queue of the requested child is empty, which can occur when the child is working on producing a result. Since the results are not sent as a complete table, we need a way for the child to inform the parent that it will not produce any new edge. To do this, we send a special edge as a signal: $END_EDGE = Edge\{NONE, NONE\}$, where $NONE = UINT32_MAX$.

The concurrent blocking queue was implemented under a class named *BlockingQueue* using concurrency primitives such as condition variables and mutexes. Furthermore, we tried to reduce the costly signaling operations as much as possible by only waking up the parent thread if the number of items in the queue is higher than a given threshold. The size limit parameter of the queue controls the amount of memory we are willing to allocate to each physical operator. It thus acts as a parameter to control the trade-off between evaluation time and memory.

We summarize our first attempt at this abstract class in algorithm 1.

Algorithm 1 Physical Operator abstract class

```
1 class PhysicalOperator {
2     PhysicalOperator* left;
3     PhysicalOperator* right;
4     ResultSorted defaultResultSorted;
5     ...
6     virtual cardStat eval() = 0;
7     virtual void evalPipeline(ResultSorted resultSorted) = 0;
8     virtual Edge produceNextEdge() = 0;
9     cardStat getCardinality();
10    ...
11 }
```

9.3 Main Physical Operators

In our initial implementation, we used three physical operators. The *IndexLookUp* and *KleeneStar* operators dealt with one-label operations, which are found in the leaves of the query tree. Those operators, that probe the index, are required to be able to produce results sorted on either source or target efficiently. For concatenation, we used only a single join algorithm, which was an implementation of Merge Join.

9.3.1 IndexLookUp

The *IndexLookUp* operator simply probes the index to find all edges with a requested combination of source, target, and label. The operator works for both bound and unbound source and target variables. Furthermore, *IndexLookUp* can deal with reversed edges as well. The result edges found by *IndexLookUp* are pipelined into an output queue, where the parent operator (i.e. a *MergeJoin*) can pull them from.

9.3.2 KleeneStar

The *KleeneStar* operator is used to deal with queries that include the Kleene star. It constructs the transitive closure for a specific label: all paths consisting of only edges with this label from a given source node to all reachable target nodes. It finds these paths using depth-first search (DFS). For

prep time	0
eval time (syn)	414
eval time (real)	2106.0
graph loading time	2821
peak memory	20.07
score	2923.47

Table 1: Leaderboard score: basic physical operators

bound queries, we only have to do a single DFS from whichever variable is bound, be it the source or the target node. For unbound queries, we have to execute a search for each source node that has an outgoing edge with the label in question. This can quickly lead to large intermediate results. As such, pipelining is essential in keeping the KleeneStar operator somewhat efficient while still being able to produce sorted results in batches. Since each DFS will output all the relevant edges for the source node the search was started from, we naturally output our results into the output queue in a source-sorted format. We can also be sure that once all results of a specific DFS have been output, the operator will produce no more edges with the source node that DFS was started from. This aids in useful pipelining, as MergeJoin requires its inputs to be sorted.

9.3.3 MergeJoin

In this implementation, our only join algorithm was merge join. The corresponding MergeJoin operator pulls intermediate result edges from the output queues of its children, which can be either leaves or other concatenations. It performs a natural join of these two input relations using the well-known merge join algorithm[1]. The results of this operator need to be sorted in order to be used as an input for another MergeJoin one level higher in the tree. We have to apply damming here, because we cannot sort the join result before all the edges in it are known. Unfortunately, this means we cannot make use of pipelining to its fullest extent if we have MergeJoin as our only join algorithm.

9.3.4 Leaderboard

Our first results on the leaderboard were not really good, see table 1, especially given the fact that the basic implementation (<https://dbtech.avantgraph.io/hush/quicksilver>) has a score of 3300. However, in the following sections we will of course improve considerably this score.

9.4 Duplicate removal

As we were far from satisfied with our leaderboard results, we began looking into ways to improve. We realized fairly early on that since we were counting distinct paths, any duplicates encountered in the final result, and by extension in the intermediate results, were superfluous. As it turns out, natural joins tend to produce many duplicates in the specific case of a graph database. As such, we started looking into ways to remove duplicates from intermediate results, in order to reduce both our memory footprint and the size of the joins that had to be executed. That is where we got the idea of duplicates removal. The C++ standard library offers various tools that help with this task:

- `ordered_map` (set)
- `unordered_map` (hash table)
- sorting followed by single-pass duplicate removal

After trying each option, we were surprised to find that not only the set, but also the hash table approach ran into trouble efficiently handling high-cardinality (intermediate) results. The sorting approach, although we hypothesized that it would be worse than the use of hash tables, turned out to be the fastest by a significant margin. However, we required the final output to be sorted either on source (respectively target) to be able to use a hash table for the *noIn* (respectively *noOut*) field in

prep time	0
eval time (syn)	370
eval time (real)	1582
graph loading time	2821
peak memory	9.17
score	2138.2

Table 2: Leaderboard score: Duplicate removal

the final cardStat. This combination of sorting/hashing gave us both the fastest evaluation time and the lowest memory footprint.

This relatively minor code change in our project led to major improvements on the leaderboard, as you can see in Table 2.

9.5 Custom Index

Because of the nature of our index, which is constructed in full before any queries are answered, we do not need to implement insertion or deletion capabilities. This means that it is possible to use a more bare-bones data structure to represent it. In that sense, the maps offered by the C++ standard library are suboptimal. To reduce the memory overhead created by the tree-depth of nodes in the map, we decided to flatten the operator tree so that all nodes are on the same level, allowing us to leave the depth out entirely. This reduces the number of pointers that need to be sorted, and thus the overall memory consumption. Moreover, we want to avoid needlessly storing the source for each consecutive edge having the same source, which is possible since the edges are sorted. Our implementation defines two structs: *Header* and *LabelIndex* (see algorithm 2).

Algorithm 2 Data-structures index

```

1 typedef struct {
2     Node source;
3     uint16_t nbTargets;
4     uint32_t indexEdges;
5 } Header;
6 typedef struct {
7     Header* headers;
8     uint32_t nbHeaders;
9     Node* edges;
10    uint32_t nbEdges;
11 } LabelIndex;

```

The first struct represents a given source and encodes the number of targets associated with it, as well as the index in the edges array where we can find its associated targets. The second is a data structure containing an array of headers (named *headers*), sorted by source, along with a continuous array of targets (*edges*) for the sources in the headers. Using a *LabelIndex* for each label, we can fully encode all the edges of a graph. The next step is to construct such an index for both regular and reversed edges. C arrays are used for efficiency in these structs. An example of this index can be found in figure 7. Having a custom index allows us to define a custom iterator that can directly reverse the edges, which simplifies the implementations of the *IndexLookUp* operator and the construction of synopses for cardinality estimation. For bound queries, we can simply perform a binary search in the *LabelIndex.headers* array to find the source we are looking for. Since we have complete control over the data-structure, we leave ourselves the possibility to use further compression methods in case memory ever became the bottleneck in our evaluation process. However, even in our final solution, our score is still bounded by the evaluation time, making such measures unnecessary.

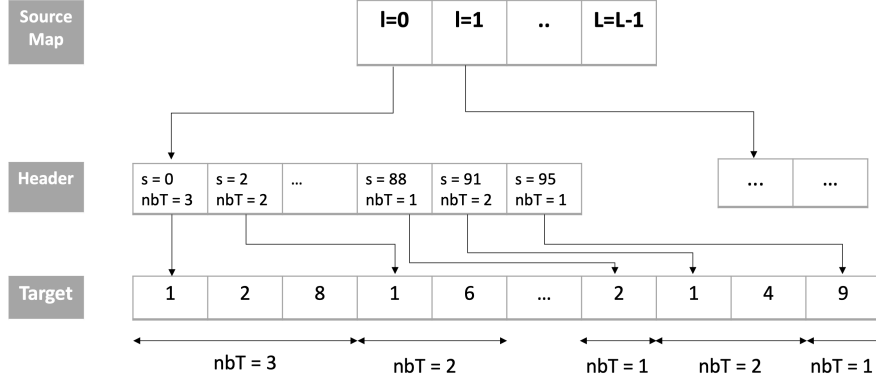


Figure 7: Custom index example

This special index allows us to reduce the memory usage on the leaderboard by 2 MB, which is a significant improvement.

9.6 Logical Optimization

A query evaluation plan describes a sequence of operations that can be performed to evaluate a query. One query can have multiple possible evaluation plans, each with their own evaluation cost. Our aim is to choose a plan with the lowest evaluation cost in order to improve the evaluation time. However, it is also important to be quick in calculating these costs; if calculating evaluation costs takes longer than the evaluation itself for an average query, cost calculation is of limited usefulness. We adopt two approaches: one for bound and one for unbound queries. For unbound queries, we first flatten the tree, and then perform cost-based join order selection. For queries with bound source or target, we construct special evaluation plans explained in more detail later, while pushing down the selection operator.

9.6.1 Unbound queries

Initially, a query is represented as a left-deep tree. However, this might not be the cheapest plan. In order to consider all possible plans, we need to allow representing a query as a bushy tree. To do that, we first need to flatten the left-deep tree that we already have. This is a useful intermediate step for sub-tree generation. Flattening, in this context, means modifying the tree such that its depth is 1. This is easy to do in our case, since the way the queries are constructed leads to all internal nodes being concatenation operations. Consider the example in Figure 8.

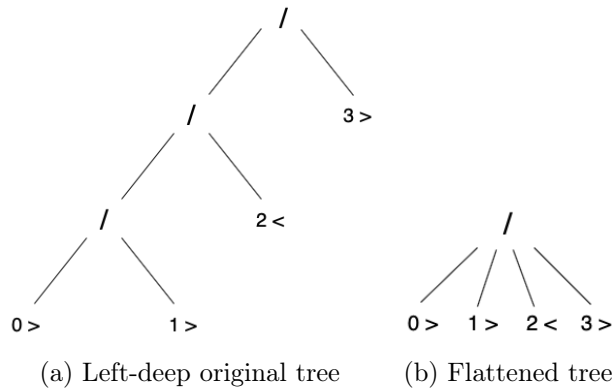


Figure 8: Flattening left-deep tree

As soon as the tree is flattened, we can generate bushy tree plans. As mentioned, there can be multiple

evaluation plans for the same query and these plans have different evaluation costs. Consider the example in figure [Figure 9](#), where two different bushy plans are represented for the flattened tree in [Figure 8b](#).

Because we are concerned with the time it takes to determine which of all the possible plans has the lowest cost, we want to calculate costs in an efficient manner. For this reason, we use a dynamic programming algorithm for join order optimization. This is a cost-based algorithm that generates evaluation plans in order of increasing size of the sub-tree as was presented by [8]. For each sub-tree of size n it stores the cheapest evaluation plan in a hash map. This is done by recursively combining two cheapest physical plans of size k and $n-k$ in order to get the plan for size n . As a result, when the algorithm terminates, we have computed the cheapest evaluation plan for the complete query.

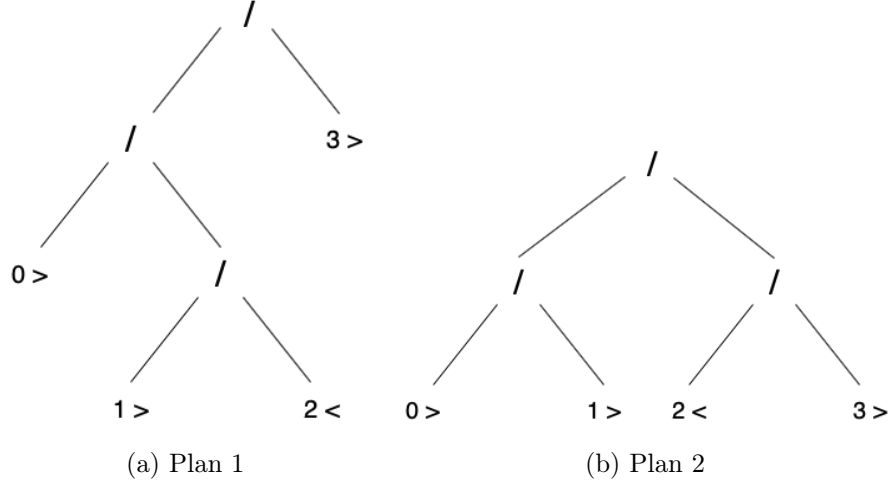


Figure 9: Possible bushy plans for flattened tree in [Figure 8b](#)

9.6.2 Bound queries

Queries are said to be bound when either its source or target variable is a constant. Note, that source and target cannot both be constants simultaneously in the context of this project. Thus, we consider only two special cases for bound queries. The underlying idea for dealing with bound queries is based on the heuristic of pushing down selection operator as far as possible.

Let us first consider a source-bound query. With a bound source, when the query is parsed into a path tree, the selection operator can be directly applied to its leftmost child. In doing so, we get fewer results after merging the left-most child. As a consequence, a good strategy is the construction of a left-deep query evaluation plan, because this construction results in smaller intermediate results on every level of the tree. An example of a left-deep query tree can be seen in [Figure 10a](#).

Analogously, with a bound target variable, the selection operator can be directly applied to the right-most child of the query tree. In this case, constructing a right-deep query evaluation plan is our best option. An example of a right-deep query tree can be seen in [Figure 10b](#).

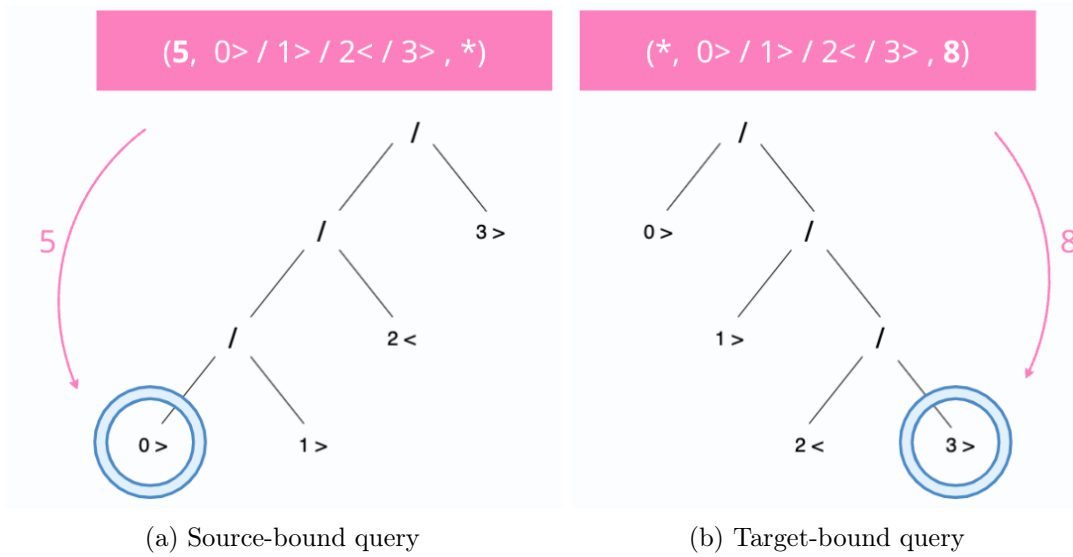


Figure 10: Pushing down selection operator for bound queries

This approach to bound queries is adopted because it applies the idea that it is not worth it to find the optimal evaluation plan if the cost analysis itself takes as much time as the average query. In this project, since we know that the left- or right-deep evaluation trees are good enough to consistently avoid the worst plans for bound queries, we build them directly instead of applying the more costly dynamic programming algorithm.

9.6.3 Cost Model

Our cost model for logical optimization was based on the cardinality estimation of Part 1 as well as on the lecture slides on query processing[16] and query optimization[17]. Moreover, the Telepath project[8] also gave us good insights about how to choose a good cost model. The physical operator interface was thus augmented by a purely virtual method *cost* returning an integer. We want to take into account both the preparation time, such as sorting for MergeJoin, and the intermediate result size when deciding on which physical plan we should pick to evaluate our query. For our basic operators we have:

- **IndexLookup** \rightarrow *cost* = 1 since our index allows to always retrieve an iterator on the requested edges in constant time.
- **KleeneStar** \rightarrow *cost* = *getCardinality()*. Indeed, some computation must be performed to build the transitive closure, and all edges have to be output at some point. The cost can thus simply be defined as its estimated cardinality.

The cost of a merge join is more complex and can be found in algorithm 3. It first computes the intermediate result cost of both the left and the right children. Then, the cost of the merge join operator itself is added to it. From the lecture slides[16], we recall that this is the sum of the size of the left and the right relation, in addition to the potential cost of having to sort the left and right relation first.

Algorithm 3 Merge Join: cost

```
1 uint32_t MergeJoin::cost() {
2     uint32_t intermediateCost = left->cost() + right->cost();
3     uint32_t operatorCost = 0;
4     uint32_t leftCard = left->getCardinality().noPaths;
5     uint32_t rightCard = right->getCardinality().noPaths;
6     operatorCost = leftCard + rightCard; //cost without sorting;
7     //add cost to sort left result
8     if (!(left->defaultResultSorted == TARGET_SORTED ||
9           left->defaultResultSorted == ANY)) {
10        operatorCost += leftCard * (uint32_t)std::log(leftCard);
11    }
12    //add cost to sort right results
13    if (!(right->defaultResultSorted == SOURCE_SORTED ||
14          right->defaultResultSorted == ANY)) {
15        operatorCost += rightCard * (uint32_t)std::log(rightCard);
16    }
17    return operatorCost + intermediateCost;
18 }
```

9.7 Cardinality estimation

In this section, we describe additional improvements we made to the cardinality estimator we implemented in Part 1.

9.7.1 Improved estimation

While we were testing our logical estimation, we found out that our cardinality estimation needed to be improved. We revisited the math we developed in synopsis 3. We figured out that the estimation was greatly improved if we used the same statistics as in synopsis 2, but computed on a graph with reversed edges either on the first edge of the path ($<>$) or the second ($><$). We redefined the first case as the new synopsis 3 and the second as synopsis 4. Moreover, since we used a pseudo-merge-join algorithm to estimate *two*, *middle* and *in*, we can still compute the final number of paths for each join (with possible duplicate edges in the result) and store it in the corresponding synopsis. We also extended synopses 2, 3, and 4 with another statistic: *noPaths*. This allows a perfectly accurate prediction of all queries of size 2 (requiring one join). This will decrease the error in the following recursive formula:

$$noPaths = left.noPaths \cdot \frac{syn_{2,3,4}.in}{l_1.in} \quad (9.1)$$

9.7.2 Cardinality estimation for bushy trees

Since we only dealt with left deep trees in our previous work, the generalisation to bushy trees is not immediately clear. However, we should note that the final result cardinality of a query does not depend on the order in which the joins are performed. This means that we can convert the bushy plan into a left deep plan to estimate its cardinality.

9.7.3 Leaderboard point

With our logical optimization working well, we pushed our code to the leaderboard, achieving the score in Table 3. We can see that the evaluation times on both synthetic and real data sets benefit from this improvement, as we expected.

prep time	956
eval time (syn)	192
eval time (real)	1424
graph loading time	2621
peak memory	7.46
score	1769.7

Table 3: Leaderboard score: Logical Optimization

9.8 Pruning non-joining edges in MergeJoin

An observation that we can make from the basic implementation provided with the project is that the adjacency list representation of the graph allows to directly access from a given left edge (s, m) all edges that we know are going to join (m, t) to produce the resulting edges (s, t) just by accessing $left.adj[m]$. However, in the traditional merge join algorithm, we have to skip all the edges with no matching join attribute for both the left and the right children. An idea for improvement is to reduce the number of edges that we have to consider in this skipping phase, allowing better pruning. We can achieve this by augmenting the interface of our physical operator with a method *skip()* (see algorithm 4).

Algorithm 4 Physical Operator: skip method

```

1 class PhysicalOperator {
2     PhysicalOperator* left;
3     PhysicalOperator* right;
4     ...
5     virtual void evalPipeline(ResultSorted resultSorted) = 0;
6     virtual Edge produceNextEdge() = 0;
7     ...
8     virtual void skip(Node until) = 0;
9     ...
10 }
```

Since all of our current physical operators work on sorted data, this can be implemented as e.g. a binary search. Our MergeJoin algorithm becomes 5. The index already provides an easy way to skip edges (iterate on headers without considering the targets), so we only have to update the iterator.

Algorithm 5 Merge Join: skip method

```
1 Edge r = right->produceNextEdge();
2 Edge l = left->produceNextEdge();
3 std::vector<Edge> setL;
4 while(r!=END_EDGE && l!=END_EDGE){
5     while (l != END_EDGE && l.target < r.source) {
6         //advance until we have a joining edge
7         left->skip(r.source);
8         l = left->produceNextEdge();
9     }
10    setL.clear();
11    Edge tl = l;
12    bool done = false;
13    while (!done && l!=END_EDGE){
14        //setL contains all edges in Left that have the same target value
15        if (tl.target == l.target){
16            setL.push_back(l);
17            l = left->produceNextEdge();
18        }
19        else done = true;
20    }
21    while (r!=END_EDGE && r.source < tl.target) {
22        //advance until we have a joining edge
23        right->skip(tl.target);
24        r = right->produceNextEdge();
25    }
26    while (!(r == END_EDGE) && tl.target == r.source){
27        for(auto tl : setL){
28            res.push_back(Edge{tl.source, r.target});
29        }
30        r = right->produceNextEdge();
31    }
32 }
```

9.9 Pipelining and IndexJoin

9.9.1 Introduction to IndexJoin

Using the built-in profiling tool of CLion, we figured out that for large cardinality queries we spent most of our time sorting the data in the merge join. Moreover, we cannot pipeline the computation due to the sorting requirement of the merge join algorithm. This brings us to the idea of batch sorting. We initially tried to build a multi-threaded merge join that sorts part of the output in other threads and then combine them in the *produceNextEdge* method (while removing the duplicates). However, as this did not lead to any particular improvement so we decided to give up on this approach.

Running into a drought of ideas, we came back to do some literature research on pipelined join algorithms, and found a paper introducing a pipelined version of HashJoin[13]. From our discussion in section 9.4, we still believe that hash tables provided by the C++ standard library are not worth using for large cardinality queries, so we do not want to swap to hash joins for our project implementation. Nevertheless, the ideas in the paper can be used to create an algorithm that does not use a hash function at all. The key observation is that in an RDF graph database, the hash function of the hash join will result in left edges having the target equal to the source of the right edges. In the context

prep time	941
eval time (syn)	143
eval time (real)	920
graph loading time	2593
peak memory	3.79
score	1143.3

Table 4: Leaderboard score: IndexJoin version 1

prep time	951
eval time (syn)	132
eval time (real)	620
graph loading time	2606
peak memory	4.09
score	838.3

Table 5: Leaderboard score: IndexJoin version 2

of a left-deep tree where the right child of a join algorithm is always going to be an index lookup, we can store the edges coming from the right child into an adjacency list such that indexing this list by the target of a left-child edge will immediately give all the corresponding targets, without using any hash function. This means that if the right-child edges are produced in a source-sorted order, we can simply produce all the edges of the left child having the same source s , store them in an array, then query the adjacency list representation of the right results, and finally output all joined edges into an output array. This output array can now be sorted and passed over for duplicate removal, since we know that the left child will not produce any more edges with source s . We have thus achieved a batch-sorting pipelined join algorithm, which we simply refer to as *IndexJoin*. Locally, we figured out that this algorithm works faster than MergeJoin for large cardinality queries (more than 100 000 tuples), leading us to update the logical optimization to take this property into account. We build a left deep tree consisting of only IndexJoin operators whenever the estimated cardinality is bigger than 100 000 edges. This implementation led to significant improvements on the leaderboard, reducing our previous score by almost 600 points, as can be seen in Table 4. Note that the memory footprint is now one of the best on the leaderboard. Moreover, this approach also allows us to compute the final cardStat incrementally while we are outputting the results.

9.9.2 Reachability queries & Random-Access Index

We can further improve the algorithm by not storing the right relation in memory, and instead relying on the index to find all the nodes reachable by a given source. However, since our index answers reachability queries of length 1 in $\log_2(N_l)$, where N_l is the number of source nodes associated to label l , we would really benefit from a random access index for those queries with a $O(1)$ complexity. We thus updated our index to allow random access. This was done by storing an array of size V , where V is the number of vertices in the graph, for each label. This array acts as a hash table where the value is the *indexEdge* field of the header associated with the source key. This considerably improved our evaluation time at the small price of storing the hash table in memory. The results can be seen in Table 5.

9.10 2-path index for $l_1 > l_2 >$ queries

When we were first designing the index, we considered increasing k in the $k - path - index$ approach, since it was shown[14][15][8] to lead to significantly faster evaluation time. However, we initially gave up this idea due to its large memory requirement. We revisited it with the following observation in mind: by looking at the query files we were provided, we saw that a large proportion of the queries actually involve a join of the form $l_1 > l_2 >$. This means that we actually do not have to build a complete $2 - path - index$ for all combinations of queries $>>$, $<<$, $<>$, $><$ to achieve improvements

prep time	940
eval time (syn)	120
eval time (real)	567
graph loading time	3504
peak memory	6.01
score	812.6

Table 6: Leaderboard score: 2-path index

at runtime. We thus decided to augment our index with a $2 - path$ index for $l_1 > l_2 >$ queries. This does incur memory costs, but we believe it to have a noticeable effect on evaluation time. The resulting score can be found in Table 6. The logical optimization has been changed to take this into account, but we believe that there are still improvements to be made in that side of the project.

9.11 Other attempted improvements

9.11.1 Adjacency list as intermediate data-structure

As we mentioned in our presentation, we started an almost completely new project from scratch to change our intermediate data structure to an adjacency list representation (code can be found in the *adj - intermediate - data - structure* branch). However, even for small cardinality queries, we could not achieve satisfying results, leading us to abandon this approach.

9.11.2 Cached Kleene Star

Recall that our current implementation of Kleene star simply runs a DFS for each source node to allow pipelined batch-sorted results. We tried to cache some intermediate results in the computation to avoid going through all the depth-first searches. However, this didn't lead to noticeable improvements, so we decided to keep the version without caching.

10 Results

In this section, we will present the progression of our leaderboard scores. We will also mention possible improvements to our project that we thought of, but did not implement due to time constraints.

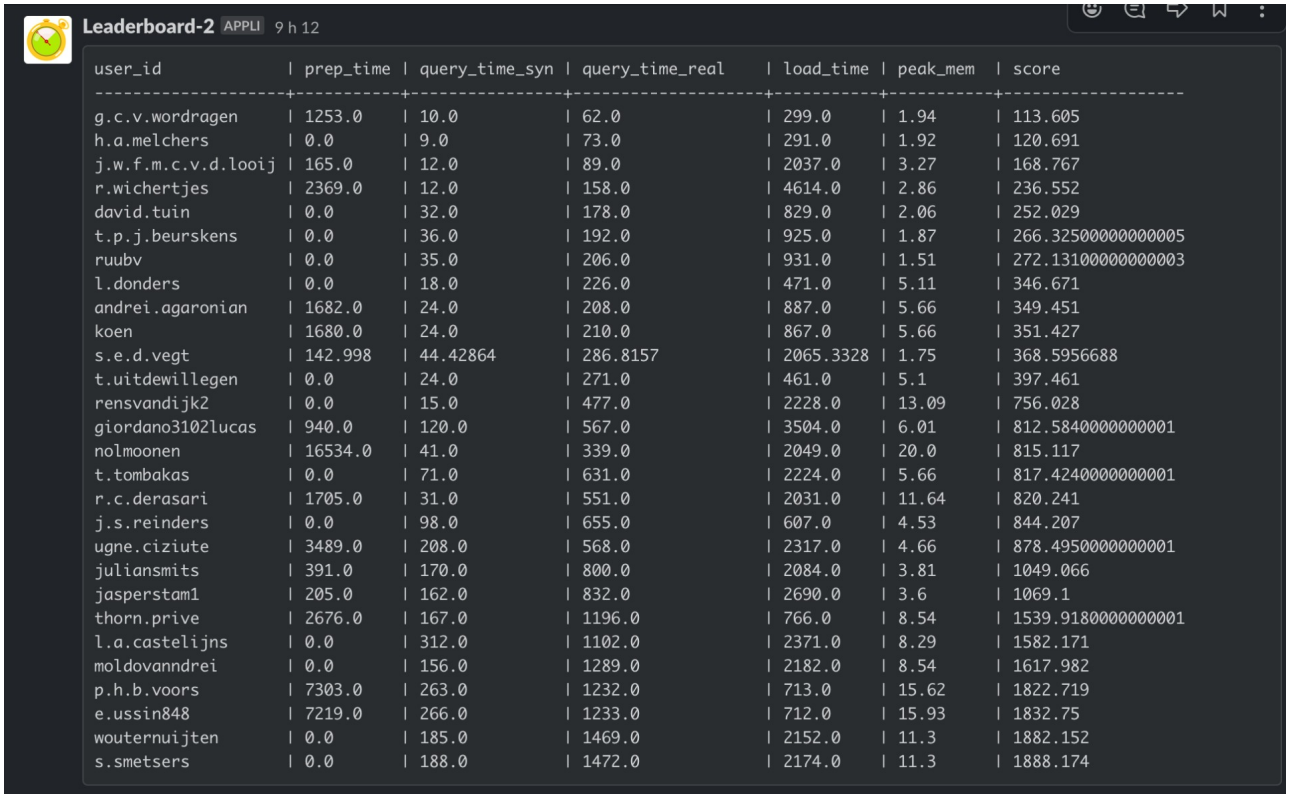
10.1 Leaderboard progression

Although we showed our leaderboard progression throughout this report, we would like to briefly summarize it here.

The first milestone we reached was the implementation of the initial index and the first versions of the physical operators. We entered the leaderboard with a score of 2923. After implementing duplicate removal, we improved our score to 2138. Fixing our cardinality estimation and implementing logical optimization allowed us to reach a score of 1769. Finally, the addition of IndexJoin considerably lowered our score, getting us to a final score of 838. Our final leaderboard ranking can be seen in Figure 11.

10.2 Possible improvements

In our video presentation, we mentioned that we were going to explore the smart transitive closure algorithm presented during the lectures. We did look at it, but we could not find an easy way to couple it with the sorting requirement of our join algorithms. However, we focused more on the evaluation of large cardinality queries, since they clearly represented the bottleneck in our evaluation pipeline. For this reason, the implementation of the smart transitive closure cannot be found in our project. Given more time, we think that it could have been valuable to further explore this aspect.



user_id	prep_time	query_time_syn	query_time_real	load_time	peak_mem	score
g.c.v.wordragen	1253.0	10.0	62.0	299.0	1.94	113.605
h.a.melchers	0.0	9.0	73.0	291.0	1.92	120.691
j.w.f.m.c.v.d.looi	165.0	12.0	89.0	2037.0	3.27	168.767
r.wichertjes	2369.0	12.0	158.0	4614.0	2.86	236.552
david.tuin	0.0	32.0	178.0	829.0	2.06	252.029
t.p.j.beurskens	0.0	36.0	192.0	925.0	1.87	266.3250000000005
ruubv	0.0	35.0	206.0	931.0	1.51	272.1310000000003
l.donders	0.0	18.0	226.0	471.0	5.11	346.671
andrei.agaronian	1682.0	24.0	208.0	887.0	5.66	349.451
koen	1680.0	24.0	210.0	867.0	5.66	351.427
s.e.d.vegt	142.998	44.42864	286.8157	2065.3328	1.75	368.5956688
t.uitdewillegen	0.0	24.0	271.0	461.0	5.1	397.461
rensvandijk2	0.0	15.0	477.0	2228.0	13.09	756.028
giordano3102lucas	940.0	120.0	567.0	3504.0	6.01	812.5840000000001
nolmoonen	16534.0	41.0	339.0	2049.0	20.0	815.117
t.tombakas	0.0	71.0	631.0	2224.0	5.66	817.4240000000001
r.c.derasari	1705.0	31.0	551.0	2031.0	11.64	820.241
j.s.reinders	0.0	98.0	655.0	607.0	4.53	844.207
ugne.ciziute	3489.0	208.0	568.0	2317.0	4.66	878.4950000000001
juliansmits	391.0	170.0	800.0	2084.0	3.81	1049.066
jasperstam1	205.0	162.0	832.0	2690.0	3.6	1069.1
thorn.prive	2676.0	167.0	1196.0	766.0	8.54	1539.9180000000001
l.a.castelijns	0.0	312.0	1102.0	2371.0	8.29	1582.171
moldovanandrei	0.0	156.0	1289.0	2182.0	8.54	1617.982
p.h.b.voors	7303.0	263.0	1232.0	713.0	15.62	1822.719
e.ussin848	7219.0	266.0	1233.0	712.0	15.93	1832.75
wouternuijten	0.0	185.0	1469.0	2152.0	11.3	1882.152
s.smetsters	0.0	188.0	1472.0	2174.0	11.3	1888.174

Figure 11: Final leaderboard submission and ranking

Given some of the other scores on the leaderboard (especially on the synthetic data set), we believe that MergeJoin should either be improved or abandoned, but unfortunately we were not able to implement a different algorithm leading to a faster evaluation. Given the benefits of our pipelined Index-Join, we think that this could have been used more extensively in our project.

The last improvement we made, based on the 2-path index for queries $l_1 > l_2$, allowed us to improve by a few points on the leaderboard, but we believe that we could have incorporated it in a better way in our project. Currently, it is only used for the IndexJoin once per plan. However, it should also be used in all MergeJoin-based plans and in the Kleene star operator where possible. However, we have to completely refine our logical optimizer in order to do so. Since we do not store the reverse adjacency list of this 2-path index, we cannot efficiently produce e.g. edges sorted on target, a downside that considerably limits its usage. The lack of time during the exam weeks did not allow us to fully complete this task.

11 Part 2: Conclusion

In the second part of the project, we focused on implementing query evaluation in a Smart, Quick, and Frugal way. Starting from the basic implementation that was provided to us, we changed the architecture and added some of our own custom data structures. These data structures allowed us more control over our project. We proceeded to build our custom index, the basic physical operators, and a merge join algorithm, while attempting to pipeline results as much as we could. After that, we implemented a dynamic programming algorithm that does most of the work on the logical planning side of the project. With the important parts in place, we iteratively improved the separate components as we came up with new strategies and ideas, sometimes revisiting older ideas with new perspectives. Although we are not at the very top of the leaderboard, we learned a lot, and we feel like we obtained satisfactory results.

References

- [1] A. Silberschatz, H.F.Korth, S.Sudarshan. *Database System Concepts*. McGraw-Hill Education, New York, NY, USA, 7th edition, 2020.
- [2] N.Yakovets. TU/e - 2IMD10 - Lecture 1. https://canvas.tue.nl/courses/11460/files/2021780?module_item_id=163121.
- [3] G. Stefanoni, B. Motik, E. Kostylev. Estimating the cardinality of conjunctive queries over rdf data using graph summarisation, 2018.
- [4] G. Cormode et al. Synopses for massive data: Samples, histograms, wavelets, sketches, 2012.
- [5] T. Neumann, G. Moerkotte. Characteristic Sets: Accurate Cardinality Estimation for RDF Queries with Multiple Joins. 2011.
- [6] N.Yakovets. Optimization of regular path queries in graph databases, 2016.
- [7] N. Yakovets. quickSilver Course Project. <https://www.win.tue.nl/~hush/qs/>, 2020.
- [8] Giedo Mak. Telepath: A path-index based graph database engine, 2017.
- [9] A. Gubichev et al. Exploiting the query structure for efficient join ordering in sparql queries. <https://openproceedings.org/2014/conf/edbt/Gubichev014.pdf>, 2014.
- [10] Andrey Gubichev and Thomas Neumann. Path query processing on very large rdf graphs. 01 2011.
- [11] Katja Hose, Ralf Schenkel, Martin Theobald, and Gerhard Weikum. *Database Foundations for Scalable RDF Processing*, pages 202–249. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
- [12] Angela Bonifati, G.H.L. Fletcher, Hannes Voigt, and N. Yakovets. *Querying graphs*. Morgan & Claypool Publishers, 2018.
- [13] Philip Garcia and Henry Korth. Pipelined hash-join on multithreaded architectures. page 1, 01 2007.
- [14] George Fletcher, J. Peters, and Alexandra Poulouvasilis. *Efficient regular path query evaluation using path indexes*. 03 2016.
- [15] J. M. Sumrall. Path indexing for efficient path query processing in graph databases, 2015.
- [16] N. Yakovets. lecture 4 : query processing. 03 2020.
- [17] N. Yakovets. lecture 5 : query optimization. 03 2020.