Bachelor's Thesis

Efficient Property Path Evaluation within the QLever Query Engine

Florian Kramer

September 23, 2019

Supervisor: Prof. Dr. Hannah Bast
Examiner: Prof. Dr. Hannah Bast

DECLARATION

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I also hereby declare that my thesis has not been prepared for another examination or assignment, either in its entirety or excerpts thereof.

Place, date

Signature

Contents

1. Abstract **1**

2. Introduction **1**

3. Related Work **2**

4. Theoretical Basis **3**

4.1. RDF Graph . . . 3

4.2. SPARQL . . . 5

4.3. QLever . . . 8

4.4. Transitive Closure . . . 10

4.5. Recursive Descent Parsing . . . 10

4.6. Depth First Search . . . 11

5. The Algorithm **11**

5.1. Limitations . . . 11

5.2. Parsing . . . 12

5.3. Transformation into ExecutionTree . . . 12

5.3.1. Representation of Results . . . 13

5.3.2. Transitive Operations . . . 14

5.3.3. Correctness . . . 15

5.3.4. Complexity . . . 16

5.3.5. Memory Usage . . . 17

5.3.6. Advantages and Disadvantages . . . 17

5.3.7. Other Transitive Closure Algorithms . . . 18

5.3.8. Restrictions . . . 18

6. Optimizations **20**

6.1. Fixed Right Side . . . 20

6.2. Join With A Small Result . . . 20

6.3. Speedup . . . 21

7. Evaluation **21**

7.1. The Slow Queries . . . 22

7.2. The Medium Queries 23
7.3. The Fast Queries 25
7.4. Summary 26

8. Future Work 26

A. Evaluation Results 28

B. Evaluation Queries 29

1. Abstract

This work is an implementation and evaluation of the property path feature from the SPARQL query language for the QLever query engine. This allows for searching for pairs of entities that are connected by a given path in a knowledge base. An example of such a query is:

```
SELECT ?scientist ?class WHERE {
  ?scientist <http://example/is-a>/<http://example/is-a>* ?class
}
```

This work also contains a comparison against Blazegraph in the form of an evaluation on a set of 39 property paths. On average the implementation presented here was about twice as fast as Blazegraph on the evaluation queries.

2. Introduction

When searching for patterns in graphs with labeled nodes and edges matching specific groups of paths can often be useful. One language that allows for writing queries against such graphs is SPARQL[1]. The simple most pattern that can be matched using SPARQL is two arbitrary nodes connected by a specific edge (also called a predicate). For example:

```
SELECT ?a ?b WHERE {
  ?a <http://example/is-a> ?b
}
```

matches all pairs of nodes $?a$ and $?b$ which are connected by the `is-a` predicate. In this example, ?a and ?b are variables. SPARQL also supports matching chains of predicates in the form of property paths. The example of a property path given in the Abstract (`<http://example/is-a>/<http://example/is-a>*`) matches any chain of predicates beginning with the `<http://example/is-a>` predicate, and then being composed of 0 or more repetitions of the same predicate.

The QLever query engine is a query engine for SPARQL queries [2]. As of this writing, it only supports a subset of the SPARQL standard though. To add support for property paths to the QLever engine three problems needed to be solved:

1. Parsing of property paths
2. Conversion to QLever's internal execution tree structure
3. Supporting transitive path operations

Parsing the input sanitizes it and makes it manageable. Converting the parse result into a QLever execution tree allows for integrating property paths with the rest of the operations QLever supports. The 'transitive path' operations are a subset of the operations of property paths which cannot be mapped to QLever's existing operations. This will be discussed in more detail in section 5.

Large parts of the features required by property paths could be implemented using existing features of QLever. The transitive operations required special attention though, as there was no feature in QLever that could reasonably compute a transitive path. To solve this, a new operation implementing the algorithm presented in this work was added to QLever. Using this additional operation the parsed predicate path can be transformed into a QLever execution tree, which can then be used together with the rest of QLever's features.

The resulting implementation can answer most small queries from the evaluation in under a second even when running on large datasets (such as Wikidata [3]). For the complete test set of queries used in the evaluation, the average execution time of predicate path-based queries was about two times lower for QLever than it was for Blazegraph [4].

3. Related Work

Given the highly specific nature of this work (focussing on the implementation of a specific feature of SPARQL), there is little work that is directly related to this.

[2] presents QLever, an efficient query engine for SPARQL and text. It explains the use

of ids to represent entities and the way QLever's index is structure. It also compares QLevers speed to that of three other SPARQL query engines.

[5] presents improvements to Tarjan's algorithm in the form of two optimized algorithms for computing transitive closures.

[6] presents a set of algorithms for transitive closures that can be adapted for path problems and one-sided recursions.

4. Theoretical Basis

This chapter contains various definitions and algorithms that are important for the implementation of property paths.

4.1. RDF Graph

RDF Graphs store the data that can be queried using SPARQL. They are defined in a W3C recommendation [7]. An RDF Graph (knowledge base) is a directed graph with labeled nodes and edges. The most common way of representing a knowledge base is a triple based notation. For example

```
<Albert_Einstein> <is-a> <Scientist>
<Albert_Einstein> <Date_of_Birth> "1879-03-14"^^xsd::date
<Scientist> <is-a> <Occupation>
```

In the example, every row is a single triple. The example represents a knowledge base with four nodes and three edges. Each of the triples is composed of a subject, a predicate and an object (in this order). Both the subject as well as the object represent a node in the graph while the predicate represents a directed, labeled edge from the subject to the object.

The values in between `<` and `>` are IRIs. IRIs are defined similarly to URIs, but allow for a wider variety of characters than a URI. Their full definition can be found in RFC

3987 [8]. For the sake of brevity and readability not all of the example IRIs used in this work rfc3987 compliant, as they do no begin with a protocol specification. A correct IRI would be `<http://example/AlbertEinstein>`.

`"1879-03-14"^^xsd::date` is a literal. A literal is composed of a utf-8 string, a data type IRI and, if the data type is that of a langString, a language tag . In the example `xsd::date` forms the data type IRI. The standard also allows for literals without a data type IRI, which then defaults to being a langString. An example of this would be `"Albert Einstein"@de` which would be the German name of Albert Einstein.

In the example above every node has a value. RDF graphs can also contain blank nodes. These are simply defined as not being an IRI or a literal. Beyond that their specific form is implementation-dependent. They can be used as structural elements in the graph to express more complex relations. For example:

```
<http://example/A> <http://example/height> _:b0
_:b0 <http://example/value> "1.73"
_:b0 <http://example/date> "2019-03-07"
```

uses a blank node (`_:b0`) to represent a height of `A` at a given date. Blank nodes are defined locally to the file they are in (when merging two RDF graphs with blank nodes two blank nodes from different graphs with the same name are still different nodes in the result graph).

The subject of all triples has to be an IRI or a blank node. The predicate has to be an IRI. The object can be an IRI, a literal or a blank node.

The knowledge base primarily used for the evaluation and testing of this work is Wikidata [3]. Wikidata is a general-purpose knowledge base that is publicly available on their website. They also offer their own SPARQL query service [9] based upon the Blazegraph engine [4]. At the time of writing, Wikidata contained about 58 million data items [3]. Wikidata is a free and open Wikimedia project which allows everybody to create new entries.

Wikidata uses numeric ids as part of the IRIs that identify nodes within the knowledge

base. `Q5`, for example, is the entry for a human (the full IRI is `<http://www.wikidata.org/entity/Q5>`).

Most wikidata IRIs begin with one of a small set of prefixes. All of these prefixes begin with `http://www.wikidata.org`. An example of a triple from wikidata is:

```
<http://www.wikidata.org/entity/Q42>
<http://www.wikidata.org/prop/direct/P31>
<http://www.wikidata.org/entity/Q5>
```

When using human-readable names instead of URLs this is equal to

```
Douglas Adams
instance of
human
```

4.2. SPARQL

The SPARQL query language is a W3C recommendation [1]. It is a language designed for interacting with knowledge bases. For this work a subset of SPARQL that allows for querying knowledge bases is sufficient. The full language also supports updating values and more.

To query knowledge bases, SPARQL defines the select query. A very simple select query would be:

```
SELECT ?a ?b ?child WHERE {
  ?a <is-a> ?b .
  ?a <Children> ?child
}
```

This query demonstrates the two most important parts of a select query, the selected variables (`?a ?b ?child`) and the pattern the query wants to match (`?a <is-a> ?b . ?a <Children> ?child`). The matched patterns are all triples of nodes `?a, ?b, ?child` where

- A triple `?a <is-a> ?b` exists
- A triple with the same value for `?a` of the form `?a <Children> ?child` exists.

The result of the query is a table with three columns (one for each entry of the result triples) and a row for every combination of triples that match the above requirements. A possible result on a knowledge base about Albert Einstein would be:

?a	?b	?child
<Albert_Einstein>	<Scientist>	<Eduard_Einstein>
<Albert_Einstein>	<Scientist>	<Hans_Albert_Einstein>
<Albert_Einstein>	<Scientist>	<Lieserl_Einstein>

The result can be computed by first finding all triples with the `<is-a>` predicate and all triples with the `<Children>` predicate and then joining these two together on the column for the variable `?a`. The result of the join is defined as containing a row for every combination of rows from the first and second table where the entries of all join columns equal.

To allow for querying for more complex dependencies between a subject and an object SPARQL 1.1 added support for property paths. With this new addition, the predicate in a triple in a query is replaced with a property path. A property path describes a chain of predicates. A property path is made up of a set of IRIs that have been combined using the following operators:

Operator	Function
`^p`	The inverse of path p (object and subject are swapped)
`p1 / p2`	The sequence of p1 and p2
`p1	p2`	Either p1 or p2
`p*`	p zero or more times
`p+`	p one or more times
`p?`	p zero or one times
`(p)`	Evaluate p first, then the rest
`!iri` or `!(iri1|...|irin)`	Any but the given IRIs
`!^iri` or `!(^iri1|...|^irin)`	Any but the given IRIs inverted
`!(iri1|...|irik|^irij|..|^irin)`	Any but the given IRIs (with some inverted)

With a precedence of:

1. `!` Negation
2. `()` Brackets
3. `*`, `?` and `+` Transitive operations
4. `^` Inversion
5. `/` Sequence
6. `|` Alternative

Using this syntax we can, for example, recursively query for all types Albert Einstein has:

```
SELECT ?type WHERE {
  <Albert_Einstein> <is-a>+ ?type
}
```

The standard already remarks, that the `|`, `/` and `^` operator can be translated to other SPARQL expressions, making them just a shorthand way of writing more complex queries:

Property Path	Equivalent SPARQL
?a <b>|<c> ?d	{?a <b> ?d} UNION {?a <c> ?d}
?a <b>/<c> ?d	?a <b> ?tmp1 . ?tmp1 <c> ?d
?a ^<b> ?c	?c <b> ?a

The UNION operation takes the results of two graph patterns and combines them by appending one result to the other. If one of the results contains a variable the other does not have the entries for the missing variable will be unbound (empty) in the part of the union result that originated from the other result. For example, the union of the following two tables would be:

?a	?b
<1>	<2>
<1>	<3>

UNION

?a	?c
<1>	<4>
<2>	<6>

=

?a	?b	?c
<1>	<2>	
<1>	<3>	
<1>		<4>
<2>		<6>

The remaining operators are not simply shorthand ways of writing complex SPARQL expressions, but need a different approach for implementing them. This will be looked at in more detail in chapter 5.

4.3. QLever

QLever [2] is a query engine for searching a knowledge base with an optional text corpus. It provides a SPARQL endpoint through a web interface. The source code is freely available on github [10].

To setup QLever with a given knowledge base, a preprocessing step is required. For this QLever provides an executable that takes a turtle or nt file and builds an index from that. One of the important steps during index creating is the assembly of a mapping from the words used in the index (e.g. <www.wikidata.org/wiki/Q42>) to numeric ids (e.g. 21323). When answering a query QLever then operates on these ids until the query

result is being transformed into json for sending to the client, during which the ids are resolved into strings.

To answer a simple SPARQL Query such as

```
SELECT ?a ?b ?h WHERE {
  ?a <is-a> ?b .
  ?a <Height> ?h
}
```

QLever initially creates a set of seeds, one from each triple in the query. Each of these seeds can then be read from the index, where it is stored as a table with one column for every variable in the triple. For example, the index contains a table for the `<is-a>` predicate which contains the numeric ids for all pairs of nodes in the knowledge base that are connected with an `<is-a>` edge.

In a second step, an optimized order of joins of the seeds is determined. Then the resulting join tree is evaluated and the result sent to the client.

For queries containing UNION such as

```
SELECT ?a ?b ?h WHERE {
  { ?a <is-a> ?b }
  UNION
  { ?a <Occupation> ?b }
  ?a <Height> ?h
}
```

QLever optimizes the two graph patterns of the union separately. It then creates a seed for the optimization of the entire query from a Union operation over those two previously optimized execution trees.

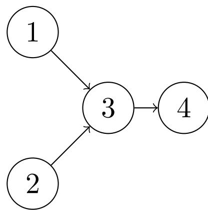
Internally, the result of the optimization is an execution tree. The nodes of this tree are small operations (such as Join, Union or Filter) that QLever can compute. The leaves of the tree are index scans, which read data from the index. The result of every operation is a table of ids which contains one column for every variable bound at this point. A bound variable is one for which values have already been found. These intermediate

results are then stored in a cache to speed up future queries. Once the final operation is finished the columns of the table the user has selected are mapped from the id space back into the input space and then turned into json (or alternatively csv or tsv).

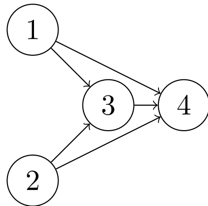
4.4. Transitive Closure

The transitive closure of a directed, acyclic graph G is a graph G' with an edge from vertex i to j if and only if (iff) there is a path in G from i to j.

For example the transitive closure of this Graph:



is



4.5. Recursive Descent Parsing

Recursive descent parsing is an old and well-known parsing technique in which a function is declared for every nonterminal of the grammar. These functions then call each other much in the same way the rules of the grammar are defined in terms of each other. This approach can be used to parse non-left recursive, context-free grammars.

For example the grammar:

```
S = v | v '+' S | '(' M ')'
M = v | v '*' M | '(' S ')'
v = [0-9]+
```

which accepts algebraic expressions containing positive numbers, addition, multiplication and requires very explicit bracket placement is a non left recursive, context free grammar that can be parsed using recursive descent parsing.

4.6. Depth First Search

Depth first search is a very commonly used technique for traversing graphs. Starting from a source node every non-marked child is visited. When a node is visited it is marked and its children are in turn visited.

5. The Algorithm

This chapter describes the main contribution to QLever in the form of an algorithm that computes the transitive operators.

5.1. Limitations

For the purpose of this work, the negation operator (`!`) for property paths was not implemented. Given the size of the data QLever targets (the entirety of wikidata) a negation would quickly result in a very large intermediate result. Even `!<http://schema.org/name>`, not the name predicate, is still almost the entirety of wikidata. Assuming wikidata to have around 11,000,000,000 triples and three times 64 bit per triple (which is the size of QLever's internal representation) the result would use about 245GiB.

Path	::=	`PathAlternative`
PathAlternative	::=	`PathSequence ( '|' PathSequence )*`
PathSequence	::=	`PathEltOrInverse ( '/' PathEltOrInverse )*`
PathElt	::=	`PathPrimary PathMod?`
PathEltOrInverse	::=	`PathElt	'^' PathElt`
PathMod	::=	`'?'	'*'	'+'`
PathPrimary	::=	`iri	'a'	'!' PathNegatedPropertySet	'(' Path ')'`
PathNegatedPropertySet	::=	`PathOneInPropertySet |`
		`'(' ( PathOneInPropertySet ( '|' PathOneInPropertySet )* )?`
PathOneInPropertySet	::=	`iri	'a'	'^' (iri	'a')`

Table 1: The grammar for property paths [1]

5.2. Parsing

The first step in implementing property paths is parsing. A recursive descent parser was used for the grammar. As there may not be any whitespace in a property path tokenization is easy. The input can be split on any of the operators (such as `|` or `/`). The result of the parsing is a tree representing the structure and precedence of operations of the property path. Every node of the tree represents an operation in the property path (such as a sequence or an inverse operator). The leaves of the tree are IRIs.

Including this parser for property paths in QLever's parser was reasonably straightforward. Any valid value for a predicate in a triple is also a valid property path. The parser knows when the next token should be a property path. Whenever that is the case everything up to the next whitespace can be consumed, and then interpreted as a property path. This nicely separates the property path parsing from the parsing of the rest of the query.

5.3. Transformation into ExecutionTree

The strategy for evaluating the property path is to take the tree generated by the parser and transform it into an execution tree, which can then be used as a seed in

the optimization of the query. This is similar to the approach taken for UNION, as the path is initially optimized separately and then added as a seed into the optimization of the containing query.

For the sequence, alternative and inverse operators the conversion to more basic SPARQL expressions offers a straightforward transformation. The only task for these then is to ensure that transformation works with the remainder of the operations.

The remaining operators (`?`, `+` and `*`) require a more specialized solution. As the existing set of QLever's Operations cannot compute the result of these operators a new operation is required. The semantics of all three operators are very similar:

- `?` Use the path zero to one times
- `+` Use the path at least one time
- `*` Use the path at least zero times

All three of the operators repeat the given path, the only difference is the maximum and minimum number of repetitions they allow for. Thus all three can be implemented using a single algorithm that takes these maxima and minima as a parameter. This algorithm can then be implemented as an operation within QLever, allowing it to be part of the execution tree generated by the optimizer. Due to the strong resemblance between this algorithm and computing transitive closures of graphs these three operators will be referred to as "transitive operators".

5.3.1. Representation of Results

Internally QLever represents intermediate results as tables of ids. To allow for simple implementations of the sequence, alternative and inversion operators the same representation is a good starting point for property paths. In this representation, the result of an operation is always fully materialized. A property path in this representation is a table with two columns. One column for the start node of the path, the other for the end node. Each row then represents two nodes connected by an instance of the given path. For example, for the path `<Occupation>` the result could look like this:

`<Albert_Einstein>`	`<Scientist>`
`<Wolfgang_Amadeus_Mozart>`	`<Composer>`

The result for the slightly more complex path `<Occupation>/<is-a>?` looks much the same:

`<Albert_Einstein>`	`<Scientist>`
`<Albert_Einstein>`	`<Occupation>`
`<Wolfgang_Amadeus_Mozart>`	`<Composer>`
`<Wolfgang_Amadeus_Mozart>`	`<Occupation>`

This representation can store the result of any path expression for a given dataset. If one of the ends of the path is not a variable but a fixed value, that value can simply be used in the table.

5.3.2. Transitive Operations

Thus the algorithm for the transitive operators gets a table with two columns as an input and returns a table with two columns. The input table can be interpreted as a set of edges of a directed graph. Every row in the table represents one edge and contains the start and end node of that edge. The nodes of that graph are then a subset of the nodes of the knowledge base. The edges can be, but don't have to be a subset of the edges of the knowledge base. The algorithm to compute the result of the transitive operation is then as follows:

- Build a hash map from every node to its neighbors. Use a hash set (a hash map without values) to store the neighbors. While building the hash map create a list of unique nodes from the left column of the input. The uniqueness can be determined by checking if the node is already a key in the hash map.
- For every node in the list of unique nodes from the left column of the input:
  - Run a dfs from that node in the graph using the hash map. Store the marked nodes of the dfs in a hash set.
  - For every node reached by the dfs check the depth against the min and max

- If the depth is within the constraints add a row to the result. The row contains the start node and the node reached by the dfs.
- If discovering a new node with the dfs would exceed the maximum depth, ignore that node and do not explore it.

In the special cases that the left or right end of the transitive operation is a specific entity, such as in this query:

```
SELECT ?a WHERE {
  ?a <is-a>+ <Albert_Einstein>
}
```

some additional steps are required. If the left side is a single entity the list of source nodes created when building the hash map contains only that entity. The hash map itself is still built fully though. This way only paths starting from that entity are considered. If the right side is fixed (such as in the example given above) only results terminating in the given node are added to the result.

For a minimum of 0 and a maximum of infinity (or the largest number that makes sense for the platform) and with an acyclic input graph this computes the transitive closure of the graph. The output can be interpreted as a graph again, with edges from each node to every node that could be reached from there in the input graph.

5.3.3. Correctness

The SPARQL standard defines the * and + operators through reachability [1]. Thus a pair of nodes a, b from the knowledgebase should be in the result exactly once if there is at least one path created from the edges in the input that starts at a and ends in b.

The proposed algorithm uses the construction of the hash map to build a list of unique nodes from those in the left column of the input. This ensures a source node is never processed twice. A visited node is marked by the algorithm and never visited a second time. Thus every pair of nodes a, b is considered as a candidate for the output at most

once, and no duplicate results will be contained in the output.

A dfs is also run from every unique node in the input, and an unconstrained dfs reaches all reachable nodes, so without a distance maximum or minimum the output will be complete.

If the maximum distance is limited the limit has to be 1 (SPARQL doesn't allow for other limits). With a maximum distance of 1 only direct neighbors of a node should be included in the output. In that case, the algorithm will simply iterate all children of the source node a and include all of them in the output. As a node is only marked once it has been visited (and thus has been included in the output), and no node beyond the maximum path length is visited, no pair of nodes that should be in the output but are not included by the algorithm, can exist. If the minimum distance is 1, which is the only other possible limitation the full dfs is run without constraints, so the result has to be complete. If the minimum distance is 0 all paths of length 0 are ignored, as they will be handled differently later on.

5.3.4. Complexity

For the analysis of the complexity, the hash map and set implementations are assumed to have a worst-case run time of $O(n)$ for both insertions as well as lookups and an amortized run time of $O(1)$ for both insertions as well as lookups.

In the worst-case, building the hash map takes $O(e^3)$, with e being the number of rows of the input table. For every row in the input one key-value pair has to be inserted into the hash map. In the worst-case, all keys are hashed to the same value, and thus a lookup takes $O(e)$. The insertion into the hash set then again takes $O(e)$ in the worst-case. As there are e insertions the run time of the building of the hash map is $O(e^3)$. When assuming amortized costs of $O(1)$ for lookups and insertions into the hash map and hash sets, the amortized run time is $O(e)$.

The second step of the algorithm is a dfs for every node in our graph. Let n be the number of nodes in the input graph. In the worst-case every dfs requires a lookup in the hash table for every node and every lookup takes $O(n)$. For every edge, the marked

state of the node at the other end also has to be checked, which is another lookup in a hash set and thus also $O(n)$. That yields a worst-case run time of $O(n^2 + en)$. The amortized cost is then $O(n + e)$.

Given that there are at most e rows in the input table the dfs is run at most e times. The worst-case run time for all dfs is then $O(en^2 + e^2 n)$, the amortized run time for all dfs is $O(en + e^2)$.

The run time for the algorithm as a whole is then $O(e^2 + en + e^2) = O(en + e^2)$ using the amortized hash map and set run time and $O(e^3 + en^2 + e^2 n)$ using the worst-case run time.

5.3.5. Memory Usage

Storing the input takes $O(e)$ bytes. The hash map then takes an additional $O(e)$ bytes. The list of starting nodes that is built together with the hash map also takes $O(e)$ bytes. The dfs has to store every node it visited (at most e) as well as what child it is currently processing for every node along the path it is currently considering. Thus, the dfs takes at most $O(d(G) + e)$ bytes, where $d(G)$ is the length of the longest cycle free path in the input graph. In the worst-case (The graph forming one long cycle) this is $O(e)$. The memory usage is thus linear in the size of the input ($O(e)$).

5.3.6. Advantages and Disadvantages

Using dfs from every node has a comparatively small memory footprint (linear in the input in the worst-case). It is also trivial to limit the valid depth for results, as the dfs stores the depth of the node it is currently processing. The algorithm can also easily be parallelized as every unique input node is processed independently of all others.

One problem with the method is, that many results may be computed several times. For example, in a graph that forms a single long chain an algorithm that can reuse subresults only has to consider one additional edge per node and can otherwise rely on previous results.

5.3.7. Other Transitive Closure Algorithms

There are a variety of different approaches to computing transitive closures. Following are some select algorithms as well as some arguments as to why they might be better or worse for this problem:

Warren's Algorithm [11] Warren's Algorithm performs two passes on the adjacency matrix of a graph to transform it into the adjacency graph of the transitive closure. There are two potential problems when trying to compute the transitive operations using Warren's algorithm. The first is memory usage. As long as the matrix is sparse this is not a problem, and would in practice likely not be a limitation. Supporting the limit on minimum distance could be solved by storing distance information in the matrix. This makes Warren's algorithm a potential candidate for an alternative implementation of the transitive operations.

CR_TC [5] Is based upon Trajan's algorithm. The basic idea is to detect strongly connected components in the input. There is a path from every node in a strongly connected component to every other node in that component, which allows for reusing these nodes and what can be reached from them. The algorithm can already deal with cycles, which is an important requirement for the transitive operators. The case of a maximum distance of one would need to be handled separately, as the algorithm does not support maximum distance limitations of any kind. Beyond that, the algorithm could definitely be used to solve the problem, and provides interesting possiblities for future work on speeding the transitive operations up. Wether the reuse of strongly connected components helps strongly depends on the input to the algorithm. A predicate like `is-a` or `Occupation` tends to be shaped more like a set of trees, while something like `connected-to` could definitely contain strongly connected components of size larger than one.

5.3.8. Restrictions

The addition of a transitive operation has one significant problem when it comes to the way QLever stores its results internally. Given the query:

```
SELECT ?a ?b WHERE {
  ?a <is-a>* ?b .
}
```

the empty path (`<is-a>` zero times) is a valid result for the property path. Any node is connected to itself with a path of length 0 though, so the result would have to contain every node in the knowledge base connected to itself. As QLever fully materializes all results this would require about 88GiB for wikidata and be very slow to process further. To avoid this we check if the empty path is a valid result of a parsed property path. For this, the minimum length of a path matching an expression needs to be calculated.

The `|`, `/` and `^` operators cannot by themselves lead to an empty path. The `+` operator has a minimum distance of one and can also not match the empty path. Thus, only `*` and `?` can introduce an empty path. When combining operators the following rules apply:

| Can be empty if one of subpaths can be empty

/ Can be empty if all of the subpaths can be empty

^ Can be empty if and only if its subpath can be empty

+ Can be empty if and only if its subpath can be empty

***** Can always be empty

? Can always be empty

Using these rules one can determine if the root operation of the tree can be empty. In that case, the computation of the result is simply aborted.

The handling of empty paths within the other operators requires some care. For the sequence operator, empty subpaths need to be resolved into a union. This is done by generating a union over all combinations of sequences with empty subpaths missing. For example `<a>/<b>*/<c>` becomes `(<a>/<b>+/<c>)|(<a>/<c>)` For all other operators empty subpaths can simply be ignored, as the flag attached to every node of the tree

indicating whether it is empty is sufficient.

6. Optimizations

The algorithm as described so far is always run on the entire input, even if the query only needs a small amount of the result. There are several simple optimizations which reduce the number of results computed in certain situations.

6.1. Fixed Right Side

If the right side of the transitive operation is a single entity, as in this query:

```
SELECT ?a WHERE {
  ?a <is-a>+ <Occupation>
}
```

computing the transitive operation for the entire input is unnecessary. Instead, the edges of the input are inverted (so the hash map maps from the target to the source of every edge). Then only a single dfs from the single node on the right side has to be computed. When writing the result pairs the order of their elements then has to be inverted. If we find a pair of entities `<a>, <b>` that are connected within the reversed graph and the constraints of the operation, then the pair `<b>, <a>` is connected by the inverse of the path connecting `<a>` and `<b>` in the reversed graph. As inverting the path does not change its length the inverted path still fulfills all distance constraints.

6.2. Join With A Small Result

If the transitive operation is part of a sequence it has a left or right neighbor it will be joined with. It might also later be joined with the execution tree representing another part of the query. If that other result is smaller than the input of the transitive operation it is faster to compute the transitive operation not for its entire input, but only for the

entries in the neighboring result. To do this, the other result is computed first and then sorted on the join column (if it is not already sorted on that column). Then the hash map is built as per usual, but the list of distinct nodes in the input is not created. Instead, the algorithm iterates over the join column of the other result and runs a dfs for every entry in there. To maintain join semantics, if an entry occurs twice in the neighbor the result for that entry needs to appear twice in the output. As the neighbor's result is sorted, equal entries always form a block, and the dfs only has to be run for the first entry of the block. For later entries, the result for the first entry can simply be copied.

If the neighbor is joined with the right result column the edge hash map has to be inverted, and the results also have to be inverted, similar to a fixed right side. With the same arguments as presented for a fixed right side, this will still yield a correct result.

6.3. Speedup

During the evaluation, these two optimization lead to a speedup of about 9 seconds per query on average. Especially queries with a fixed right side saw significant speedups.

7. Evaluation

For the evaluation QLever's and Blazegraph's performance has been compared. All evaluation queries consist of only a single property path. These 38 property paths were extracted from Wikidata's set of example queries [12]. The knowledgebase used is a subset of wikidata that only contains the data relevant to these queries. A subset of wikidata was used, as it was easier to work with, and tests with a larger dataset did not display any changes in the query time of the queries. The evaluation was run on

Endpoint	Mean [ms]	Standard Deviation	Maximum [ms]	Minimum [ms]
QLever	9941.88	34876.11	206620.71	14.37
Blazegraph	22189.19	65235.05	241000.00	29.91

Table 2: Aggregated Query Times

an Intel Xeon E5640 running Ubuntu 18.04 with 94G of RAM. Queries where run with a timeout of 240 seconds. QLever's internal cache was cleared before every query while Blazegraph does not appear to use any internal caching, and no additional caching was used. Queries that did run into the timeout are assumed to have a query time of 241 seconds. The full results of the evaluation can be found in appendix A, the property paths forming the queries in appendix B.

Table 2 contains aggregate data on the evaluation results. There are several observations that can be made about the data.

- Blazegraph's maximum query time indicates at least one timeout, while QLever never timed out.
- The standard deviation for Blazegraph is almost twice as high as that of QLever. In the full results, there are a lot of queries for which Blazegraph was very fast, and some for which it was very slow compared to QLever.
- On average QLever was more than twice as fast as Blazegraph. Given the high standard deviation of Blazegraph's results, and looking at the full results this does not indicate that QLever is always faster than Blazegraph though.

For a better understanding of QLever's and Blazegraph's strengths, I've split the evaluation results into groups based upon the longest query time for each query.

7.1. The Slow Queries

This category contains all queries that took at least 30 seconds. When examining the queries, their input sizes and result sizes, the reasons behind the long query times are obvious:

Query	Result Size
wdt:P2789	66,792
Q19 (wdt:P2789+)	101,808,568
wdt:P131	7,854,809
Q29 (wdt:P131+)	29,475,512

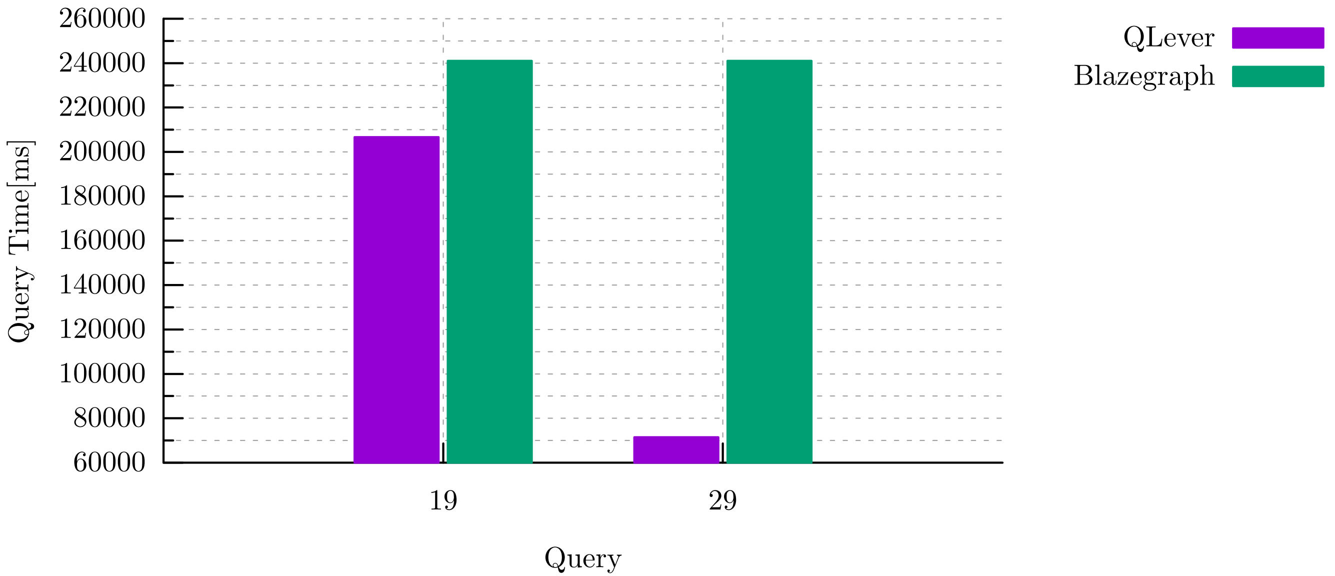


Figure 1: The Slow Queries

The table shows bot the input relations size (`wdt:P2789` and `wdt:P131`) for Q19 and Q29 as well as their result sizes. Both of these queries produce significantly more results than any query of the evaluation query set. Of note is, that the factor of the query time between Q29 and Q19 roughly matches the output size (about 70s vs 210s and about 29 million vs 101 million results). This could indicate, that the increased run time of Q19 is solely due to the larger output and that the increased connectivity of P2789 does not significantly increase the query time for QLever.

Blazegraph timed out on both queries, making it impossible to tell if it also performs better on the larger but less connected input.

7.2. The Medium Queries

This section contains all queries that took between 2 and 30 seconds. Of note in this subset of the queries is Query 25, in which QLever is significantly slower than Blazegraph, as well as the greatly varying query times of Blazegraph for queries 0, 6, 20, 21 and 33 compared to QLever's comparatively constant query times.

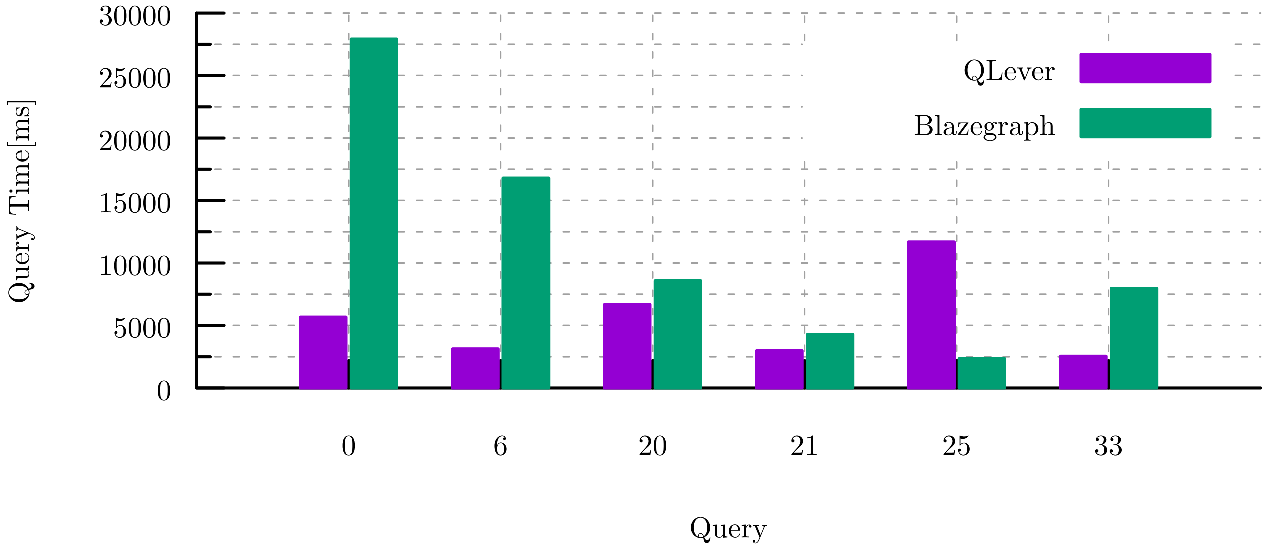


Figure 2: The Medium Queries

A possible reason for QLever's long computation time in relation to Blazegraph for Query 25 could be the large size of wdt:P31 and wdt:P279 compared to the output size. wdt:P31 has 52,739,893 and wdt:P279 2,270,780 entries. The result size of the query is only 60,057. While QLever will only do a single dst per transitive operation, as the object of the triple is a single entity it still has to build the entire edge hash map. It is possible, that Blazegraph avoids a computation on the entire input.

As for the other Queries: Queries 0 and 6 return large results (over 4 and 2 million respectively). Query 33 also has a result size of over 1 million. Query 21 also fits into the trend of being output size-dependent, as it has about half as many results as Query 33. QLever has already demonstrated with the slowest queries that it is faster for very large results.

While query 20 does not return a very large result (about 400000 rows) it does contain two large predicates, wdt:P31 and wdt:P279. That makes it very similar to Query 25, with the main difference being that no transitive operations are required on wdt:P31, which would then explain why Blazegraph and QLever have similar execution times for this query, instead of QLever being significantly slower.

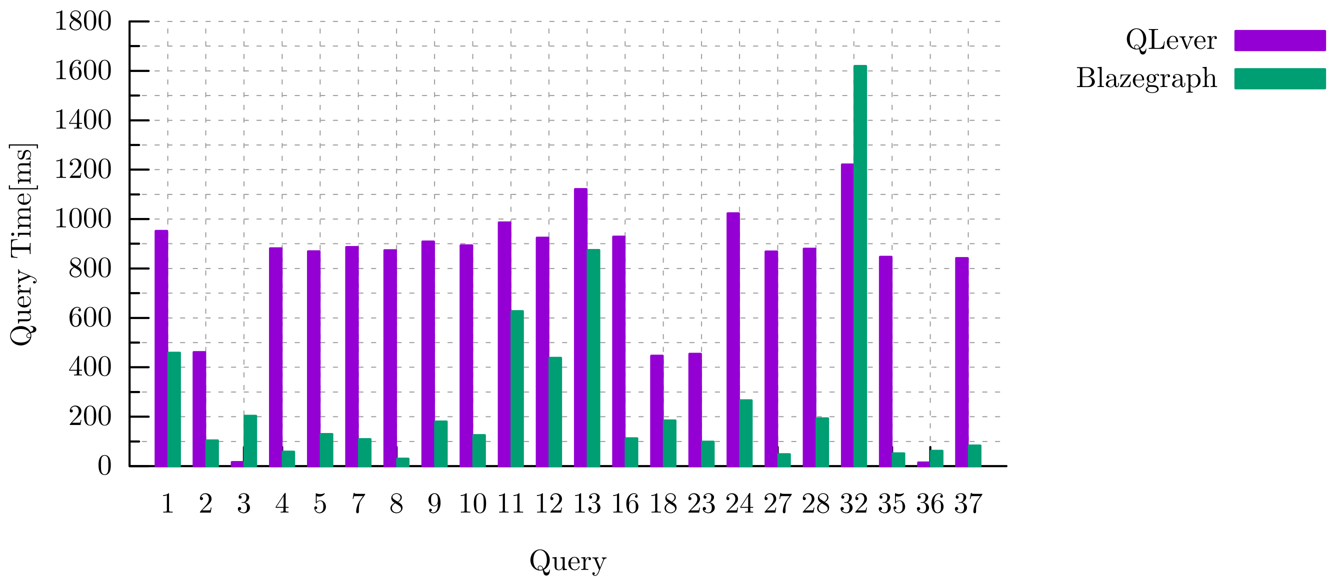


Figure 3: The Fast Queries

7.3. The Fast Queries

The fast queries are all queries that took less than 2 seconds to compute. In this category, Blazegraph shines and beats QLever for most queries. The average result length of these queries is just about 50,000 results and most query times are under 1 second for both systems.

Of note are queries 3, 32 and 36, all three of which show extremely short or long query times. Query 3 has two relatively small predicates and a fixed right side, which make it very fast to answer for both systems. Query 32 is interesting, as it contains both wdt:P31 as well as wdt:P279, very similar to Query 20. The execution time of Query 32 is much faster than that of Query 20 despite similar result sizes. A possible reason for this could be disk caches speeding up the loading of the two large predicates. Query 36 is composed of two very small predicates (one with under 100,000 entries, the other with under 10,000) which explains the speeds both systems manage on that query.

7.4. Summary

In summary, while Blazegraph is faster for queries with small results QLever scales much better for queries with larger results. For queries with very small inputs and result sizes, QLever is faster than Blazegraph, but both systems are very fast. Blazegraph appears to handle large inputs with small outputs better than QLever.

8. Future Work

There are several areas that stand out and in which QLever's property path support can be improved further:

- Supporting empty paths. This may be achievable by adding a flag to result tables that indicates if the table also contains the result of an empty path and then adding support for dealing with that flag to all other operations of QLever.
- Improving the speed of the transitive operation for small results from large inputs. Given the evaluation and Blazegraph's performance on these types of queries this is clearly possible, but I am not certain what the best strategy for improving the performance of QLever in this area would be.
- Supporting the not operator (`!`). This would mainly be useful for small knowledge bases, but could be implemented naively by iterating the entire index.
- Reusing subresults during the computation of the transitive operators. This might be accomplished by using the algorithm presented in [5] which uses strongly connected components. Depending on the shape of the input this might not lead to any significant improvements though.

References

[1] *W3C SPARQL 1.1.* URL: https://www.w3.org/TR/sparql11-query/.

[2] Hannah Bast and Björn Buchhold. "Qlever: A query engine for efficient sparql+ text search". In: *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management.* ACM. 2017, pp. 647–656.

[3] *Wikidata.* URL: https://www.wikidata.org/wiki/Wikidata:Main_Page.

[4] *Blazegraph.* URL: https://www.blazegraph.com/.

[5] Esko Nuutila. "Efficient transitive closure computation in large digraphs." In: (1998).

[6] Yannis E Ioannidis, Raghu Ramakrishnan, et al. "Efficient Transitive Closure Algorithms." In: *VLDB.* Vol. 88. 1988, pp. 382–394.

[7] *W3C RDF Primer.* URL: https://www.w3.org/TR/rdf11-concepts/.

[8] *RFC 3987.* URL: https://www.ietf.org/rfc/rfc3987.txt.

[9] *Wikidata Query Service.* URL: https://query.wikidata.org/.

[10] *QLever on Github.* URL: https://github.com/ad-freiburg/QLever.

[11] Henry S Warren Jr. "A modification of Warshall's algorithm for the transitive closure of binary relations". In: *Communications of the ACM* 18.4 (1975), pp. 218–220.

[12] *Wikidata SPARQL examples.* URL: https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/queries/examples.

A. Evaluation Results

Query	Results	QLever[ms]	Blazegraph[ms]
0	4966312	5663.413737900555	27912.98624803312
1	87815	952.15202588588	459.12213204428554
2	456	461.41138509847224	104.08312687650323
3	13708	16.424980014562607	203.27932899817824
4	3386	881.1869989149272	58.00669593736529
5	15842	869.3391859997064	129.9372969660908
6	2428903	3129.150436958298	16787.93105785735
7	10484	886.8803838267922	109.06786215491593
8	684	873.8095711451024	29.90942820906639
9	42199	908.8986669667065	180.47489784657955
10	23762	893.3967659249902	125.40710810571909
11	156608	986.1412858590484	627.201174851507
12	43473	924.7465180233121	437.76497105136514
13	266539	1121.5581190772355	874.2559358943254
14	935	11498.850631993264	48.92121907323599
15	4346	9625.69120991975	51.834324141964316
16	28832	928.9849980268627	112.68989089876413
17	21580987	21866.57116585411	241000
18	1980	446.4293448254466	185.14790991321206
19	101808568	206620.71331194602	241000
20	400727	6679.897459922358	8590.046549914405
21	848611	2989.0268170274794	4285.13348987326
22	157072	2197.793420171365	954.3136600404978
23	55	454.5198909472674	98.57554105110466
24	3673	1023.1200440321118	266.0502400249243
25	60057	11680.870834970847	2348.625988001004
26	605946	1361.5771820768714	14580.615665996447
27	6	868.8861350528896	48.20933612063527
28	15842	880.1459870301187	193.05791286751628
29	29475512	71419.56851794384	241000

30	9444	584.9689280148596	20448.568735970184
31	369257	941.9329529628158	2383.353666868061
32	390543	1221.3121470995247	1619.976585963741
33	1636254	2537.3869580216706	7968.457262963057
34	1193515	1691.299317870289	7769.262378802523
35	1933	847.1002841833979	51.947460044175386
36	6952	14.37153504230082	61.779591953381896
37	6472	841.7855598963797	83.05321191437542

B. Evaluation Queries

These queries use the following prefixes:

```
PREFIX p: <http://www.wikidata.org/prop/>
PREFIX psn: <http://www.wikidata.org/prop/statement/value-normalized/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
PREFIX wikibase: <http://wikiba.se/ontology#>
```

<table>
<tr><th>Query</th><th>SPARQL</th></tr>
<tr><td>0</td><td>?item (wdt:P31)/((wdt:P279)*) wd:Q5</td></tr>
<tr><td>1</td><td>?person (wdt:P31)/((wdt:P279)*) wd:Q95074</td></tr>
<tr><td>2</td><td>?item (wdt:P279)+ wd:Q37144</td></tr>
<tr><td>3</td><td>?composer (wdt:P19)|(wdt:P20) wd:Q1741</td></tr>
<tr><td>4</td><td>?object (wdt:P31)/((wdt:P279)?) wd:Q46169</td></tr>
<tr><td>5</td><td>?university (wdt:P31)/((wdt:P279)*) wd:Q3918</td></tr>
<tr><td>6</td><td>?item (wdt:P31)/((wdt:P279)*) wd:Q486972</td></tr>
<tr><td>7</td><td>?object (wdt:P31)/((wdt:P279)*) wd:Q159719</td></tr>
<tr><td>8</td><td>?place (wdt:P31)/((wdt:P279)?) wd:Q474</td></tr>
<tr><td>9</td><td>?site (wdt:P31)/((wdt:P279)*) wd:Q839954</td></tr>
</table>

10	?subj (wdt:P31)/((wdt:P279)*) wd:Q23413
11	?item (wdt:P31)/((wdt:P279)*) wd:Q1370598
12	?item (wdt:P31)/((wdt:P279)*) wd:Q33506
13	?item (wdt:P31)/((wdt:P279)*) wd:Q23397
14	?person (wdt:P166)/((wdt:P31)?) wd:Q7191
15	?person (wdt:P166)/((wdt:P31)?) wd:Q19020
16	?city (wdt:P31)/((wdt:P279)*) wd:Q515
17	?item (wdt:P31)/((wdt:P279)*) wd:Q191067
18	?subclass_settlement (wdt:P279)+ wd:Q486972
19	?connection (wdt:P2789)+ ?city
20	?item (wdt:P31)/((wdt:P279)*) wd:Q4022
21	?river (wdt:P31)/((wdt:P279)*) wd:Q355304
22	?island (wdt:P31)/((wdt:P279)*) wd:Q23442
23	?disease (wdt:P279)+ wd:Q504775
24	?ff (wdt:P31)/((wdt:P279)*) wd:Q235557
25	?compound ((wdt:P279)+)|((wdt:P31)+) wd:Q421948
26	?human (wdt:P106)/((wdt:P279)*) wd:Q901
27	?award (wdt:P31)/((wdt:P279)*) wd:Q7191
28	?u (wdt:P31)/((wdt:P279)*) wd:Q3918
29	?u (wdt:P131)+ ?state
30	?animal (wdt:P31)/(wdt:P31) wd:Q16521
31	?item (wdt:P106)/((wdt:P279)*) wd:Q639669
32	?item (wdt:P31)/((wdt:P279)*) wd:Q3305213
33	?item (wdt:P31)/((wdt:P279)*) wd:Q838948
34	?women (wdt:P106)/((wdt:P279)*) wd:Q483501
35	?brewery (wdt:P31)/((wdt:P279)*) wd:Q131734
36	?work (wdt:P37)|(wdt:P103) ?language
37	?language (wdt:P31)/((wdt:P279)*) wd:Q7275