Efficient Property Path Evaluation within the QLever Query Engine

Florian Kramer

February 28, 2020

Motivation

Implementation

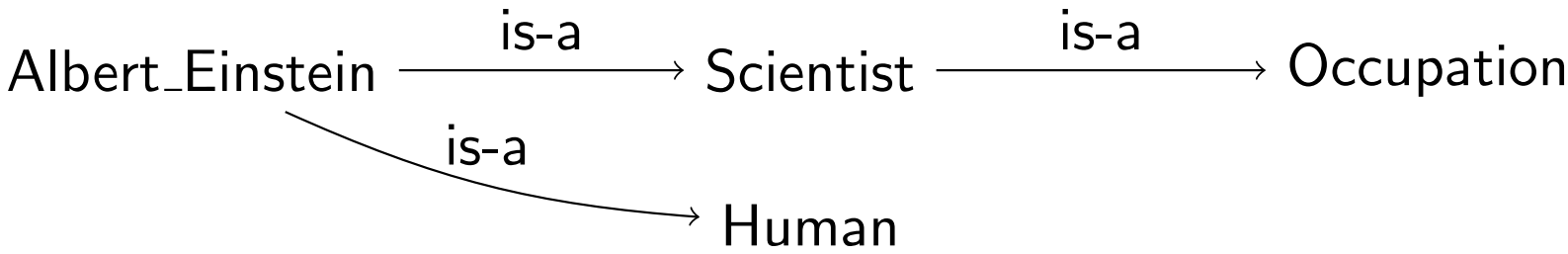
Evaluation

Motivation - Wikidata

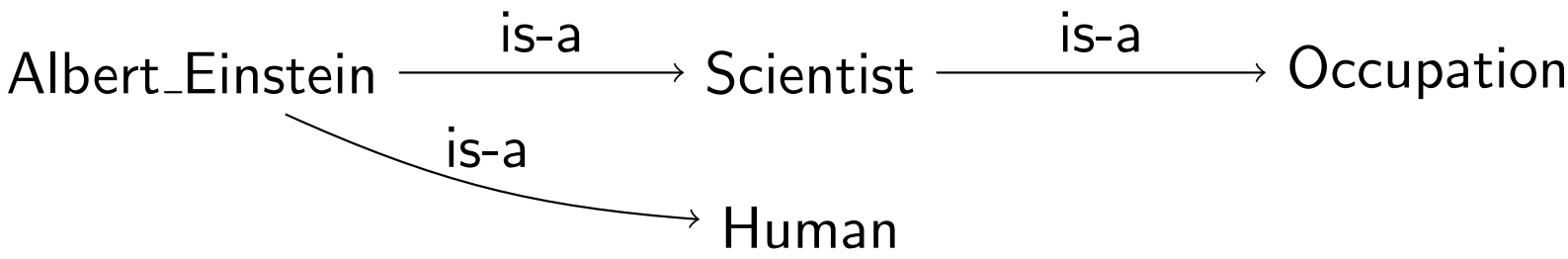
According to their website [1]:

- Free knowledge base storing structured data
- 77,613,715 data items
- Anyone can edit these
- It is a general knowledge base
- Created by the wikimedia foundation

Motivation - RDFGraph [5]



Motivation - RDFGraph [5]



subject	predicate	object
<Albert_Einstein>	<is-a>	<Scientist>
<Albert_Einstein>	<is-a>	<Human>
<Scientist>	<is-a>	<Occupation>

Motivation - SPARQL [6]

```
SELECT ?parent ?child ?bd WHERE {
  ?parent <Children> ?child .
  ?child <Date_Of_Birth> ?bd
}
```

Motivation - SPARQL [6]

```
SELECT ?parent ?child ?bd WHERE {
   ?parent <Children> ?child .
   ?child <Date_Of_Birth> ?bd
}
```

?scientist	*?child*	*?bd*
Albert Einstein	Eduard Einstein	28 July 1910
Albert Einstein	Hans Albert Einstein	14 May 1904
Albert Einstein	Liserl Einstein	1 January 1902

Motivation - QLever

- SPARQL Search Engine for knowledge bases

Motivation - QLever

- SPARQL Search Engine for knowledge bases
- Can work with very large datasets

Motivation - QLever

- SPARQL Search Engine for knowledge bases
- Can work with very large datasets
- Is on github [4]

Motivation - QLever

- SPARQL Search Engine for knowledge bases
- Can work with very large datasets
- Is on github [4]
- Supports only a subset of SPARQL

Motivation - Property Path

```
SELECT ?person ?class WHERE {
  ?person <Children>+ ?descendant
}
```

Property Path Operations [6]

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
`!p`	Any but the given IRIs (can be combined with | and /)

Questions?

Motivation

Implementation

Evaluation

Parsing

- Property paths are free of whitespace

Parsing

- Property paths are free of whitespace
- The operators can be used as delimiters

Parsing

- Property paths are free of whitespace
- The operators can be used as delimiters
- Recursive Descent parser

Parsing

- Property paths are free of whitespace
- The operators can be used as delimiters
- Recursive Descent parser
- Produces an AST which can then be processed further

Parsing

- Property paths are free of whitespace
- The operators can be used as delimiters
- Recursive Descent parser
- Produces an AST which can then be processed further
- Processes parantheses and ensures precedence of operations

Property Paths - Replacements

The sparql standard [6] defines these replacements:

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`

Negation

Operator	Function
`!p`	Everything that is not connected by p.

Negation

Operator	Function
`!p`	Everything that is not connected by p.

- Assume wikidata has about 11,000,000,000 triples

Negation

Operator	Function
`!p`	Everything that is not connected by p.

- Assume wikidata has about 11,000,000,000 triples
- QLever uses at least 2 times 64bit per result

Negation

Operator	Function
`!p`	Everything that is not connected by p.

- Assume wikidata has about 11,000,000,000 triples
- QLever uses at least 2 times 64bit per result
- No single relation in wikidata covers a majority of these triples

Negation

Operator	Function
`!p`	Everything that is not connected by p.

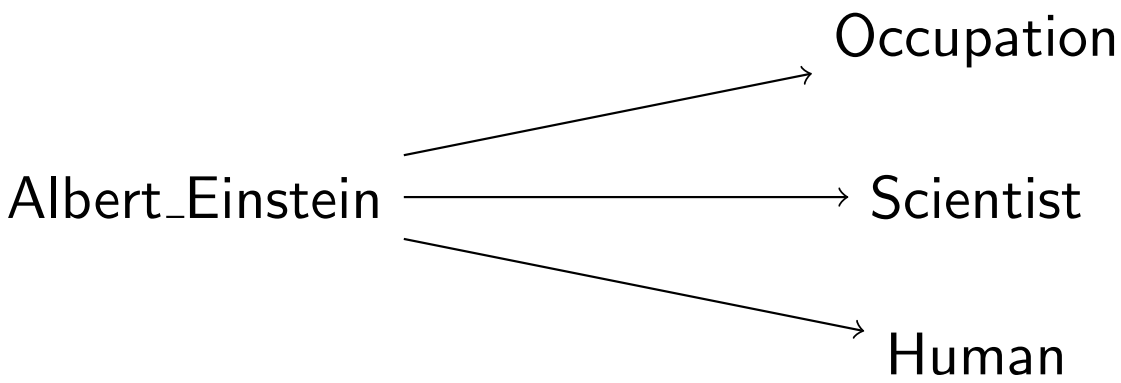
- Assume wikidata has about 11,000,000,000 triples
- QLever uses at least 2 times 64bit per result
- No single relation in wikidata covers a majority of these triples
- The result of a negation would then likely exceed 150 GiB

Transitive Operators

Operator	Function
p*	p zero or more times
p+	p one or more times
p?	p zero or one time

Result Representation

?a	?b
<Albert_Einstein>	<Occupation>
<Albert_Einstein>	<Scientist>
<Albert_Einstein>	<Human>



The Algorithm

- 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.

The Algorithm

- 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:

The Algorithm

- 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.

The Algorithm

- 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

The Algorithm

- 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.

The Algorithm

- 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.

Correctness

- The dfs will explore every reacheable node

Correctness

- The dfs will explore every reacheable node
- All results we write have a minimum distance of 1

Correctness

- The dfs will explore every reacheable node
- All results we write have a minimum distance of 1
- The algorithm works for a maximum distance of 1

Correctness

- The dfs will explore every reacheable node
- All results we write have a minimum distance of 1
- The algorithm works for a maximum distance of 1
- Every pair of nodes is considered at most once, so the output won't contain any duplicates

Amortized Complexity

Let e be the number of edges in the input. Let n be the number of distinct nodes in the input.

- Building the hash map is in $O(e)$

Amortized Complexity

Let e be the number of edges in the input. Let n be the number of distinct nodes in the input.

- Building the hash map is in $O(e)$
- The dfs is run $O(e)$ times

Amortized Complexity

Let e be the number of edges in the input. Let n be the number of distinct nodes in the input.

- Building the hash map is in $O(e)$
- The dfs is run $O(e)$ times
- A single dfs takes $O(e + n)$

Amortized Complexity

Let e be the number of edges in the input. Let n be the number of distinct nodes in the input.

- Building the hash map is in $O(e)$
- The dfs is run $O(e)$ times
- A single dfs takes $O(e + n)$
- The amortized run time is then $O(en + e^2)$

Empty Paths

- The ? and $*$ operator match the empty path. The algorithm for them doesn't support that though.

Empty Paths

- The ? and $*$ operator match the empty path. The algorithm for them doesn't support that though.
- The result of the empty path is every node connected to itself.

Empty Paths

- The ? and $*$ operator match the empty path. The algorithm for them doesn't support that though.
- The result of the empty path is every node connected to itself.
- Instead of materializing that, annotate operations that can produce an empty path.

Empty Paths

- The ? and $*$ operator match the empty path. The algorithm for them doesn't support that though.
- The result of the empty path is every node connected to itself.
- Instead of materializing that, annotate operations that can produce an empty path.
- Don't allow the empty path as a result of an entire predicate path.

Empty Paths

Operation	Can be empty
|	If all subpath can be empty
/	If any subpath can be empty
+	If the subpath can be empty

Operation	Handling
|	The annotation suffices
/	Union over all combinations of missing empty subpaths.
+	The annotation suffices

Integration in QLever

- QLever transforms every triple in a query into an operation tree with a single scan operation, internally called a seed.

Integration in QLever

- QLever transforms every triple in a query into an operation tree with a single scan operation, internally called a seed.
- We can now transform a triple with a property path into an execution tree

Integration in QLever

- QLever transforms every triple in a query into an operation tree with a single scan operation, internally called a seed.
- We can now transform a triple with a property path into an execution tree
- That execution tree is then optimized separately and then inserted as a seed into the optimization of the parent graph pattern

Optimizations

- Try to reduce the number of dfs

Optimizations

- Try to reduce the number of dfs
- Utilize joins with small results

Optimizations

- Try to reduce the number of dfs
- Utilize joins with small results
- Compute that result first, then do a dfs for every entry

Optimizations

- Try to reduce the number of dfs
- Utilize joins with small results
- Compute that result first, then do a dfs for every entry
- If the join is with the right side variable, invert the input paths and the results

Questions?

Motivation

Implementation

Evaluation

Evaluation

- Comparing QLever and Blazegraph [3]

Evaluation

- Comparing QLever and Blazegraph [3]
- Property paths from the wikidata example queries [1]

Evaluation

- Comparing QLever and Blazegraph [3]
- Property paths from the wikidata example queries [1]
- Run on the relevant part of Wikidata

Evaluation

- Comparing QLever and Blazegraph [3]
- Property paths from the wikidata example queries [1]
- Run on the relevant part of Wikidata
- The QLever cache is flushed before every query

Evaluation

- Comparing QLever and Blazegraph [3]
- Property paths from the wikidata example queries [1]
- Run on the relevant part of Wikidata
- The QLever cache is flushed before every query
- 240s timeout

Evaluation

- Comparing QLever and Blazegraph [3]
- Property paths from the wikidata example queries [1]
- Run on the relevant part of Wikidata
- The QLever cache is flushed before every query
- 240s timeout
- 38 queries composed of only a single property path each

Example Query

```
PREFIX wd: <http://www.wikidata.org/entity/>
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
SELECT ?item  WHERE {
  ?item (wdt:P31)/((wdt:P279)*) wd:Q5
}
```

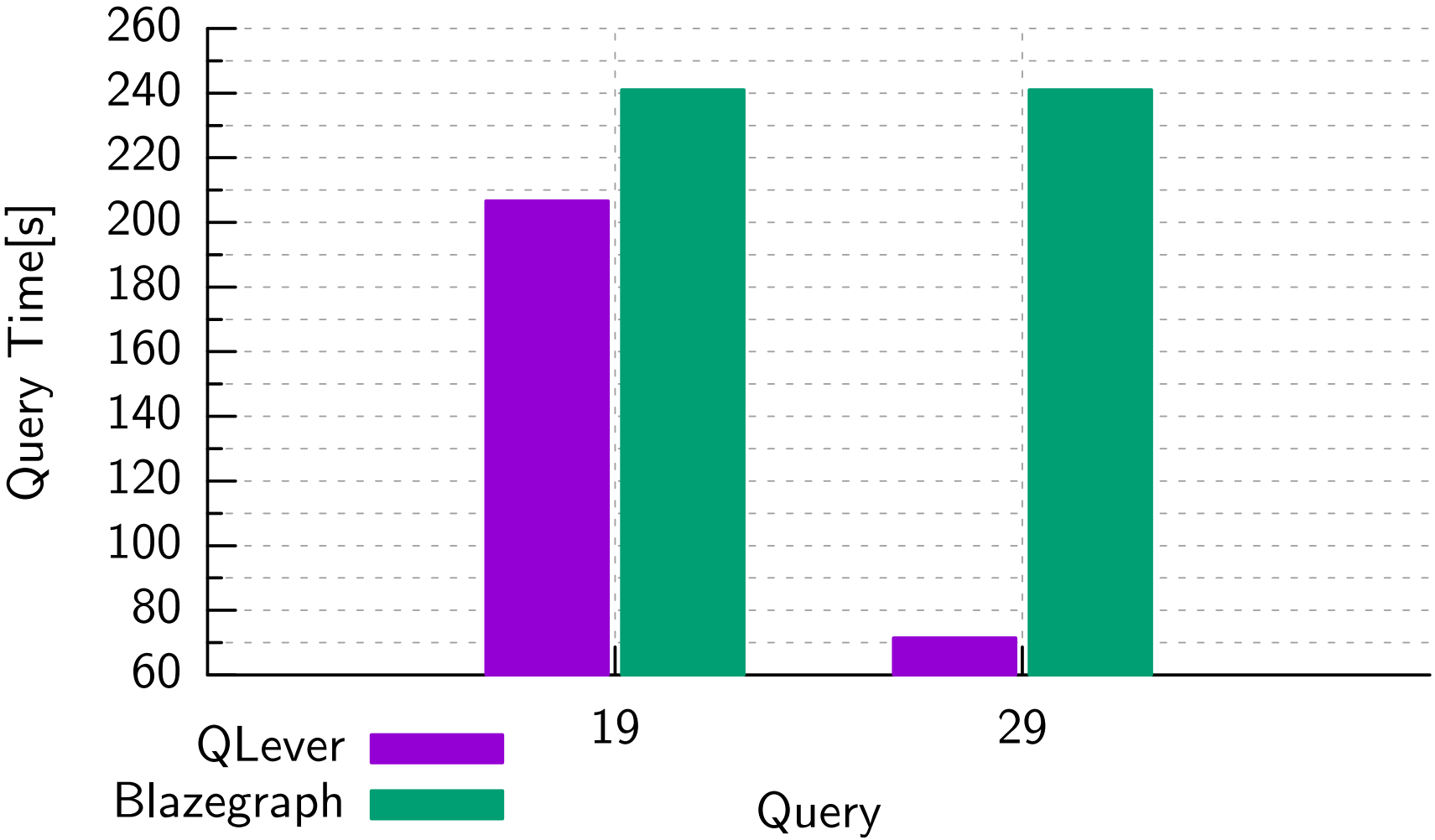
P31 instance of

P279 subclass of

Result Overview

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

Result Slow


260
240
220
200
180
160
140
120
100
80
60
Query Time[s]
19
29
Query
QLever
Blazegraph

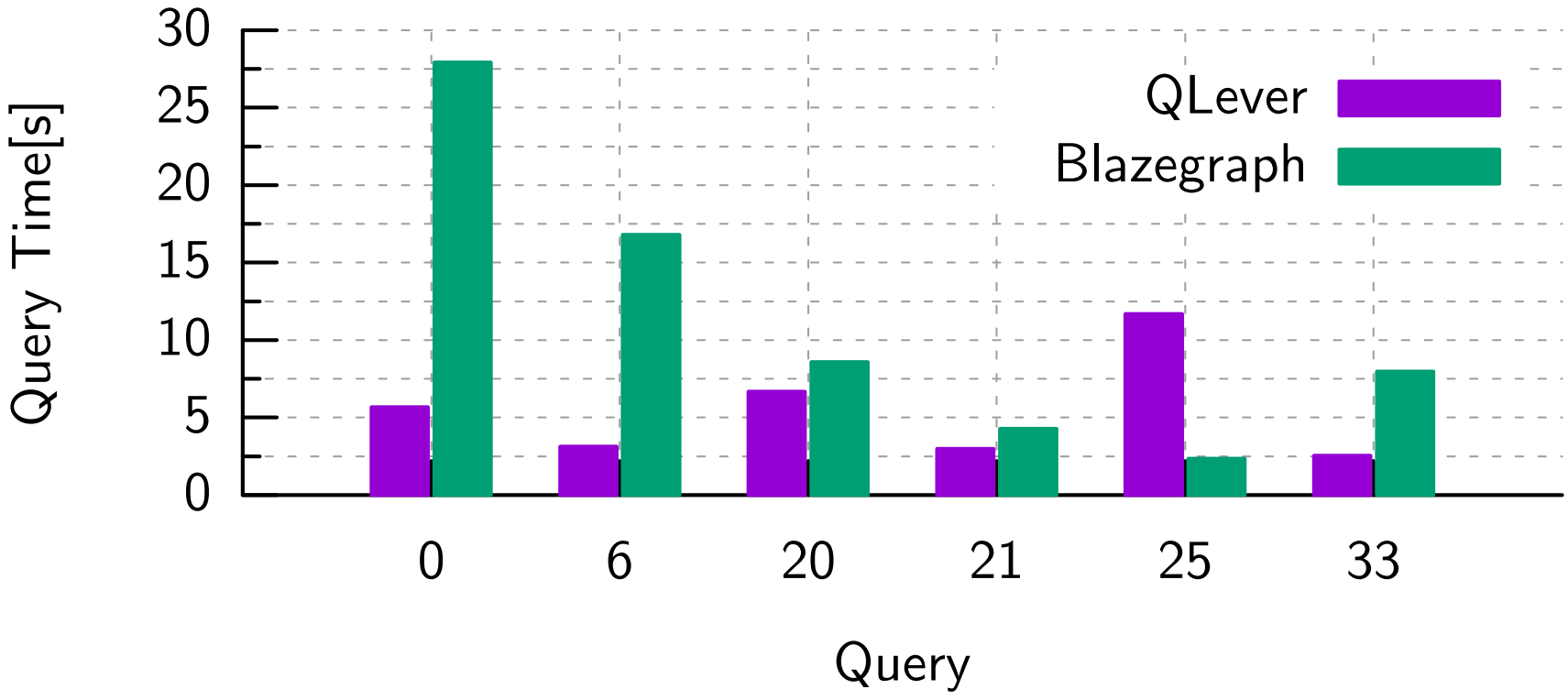
Result Slow

Query	SPARQL	Result Size
19	`?connection (wdt:P2789)+ ?city`	101,808,568
29	`?u (wdt:P131)+ ?state`	29,475,512

P2789 connects with

P2131 located in the administrative territorial entity

Result Medium



Query	0	6	20	21	25	33
Size [k]	4,966	2,428	400	848	60	1,636

Query 25

Query	SPARQL
25	`?compound wdt:P279+|wdt:P31+ wd:Q421948`

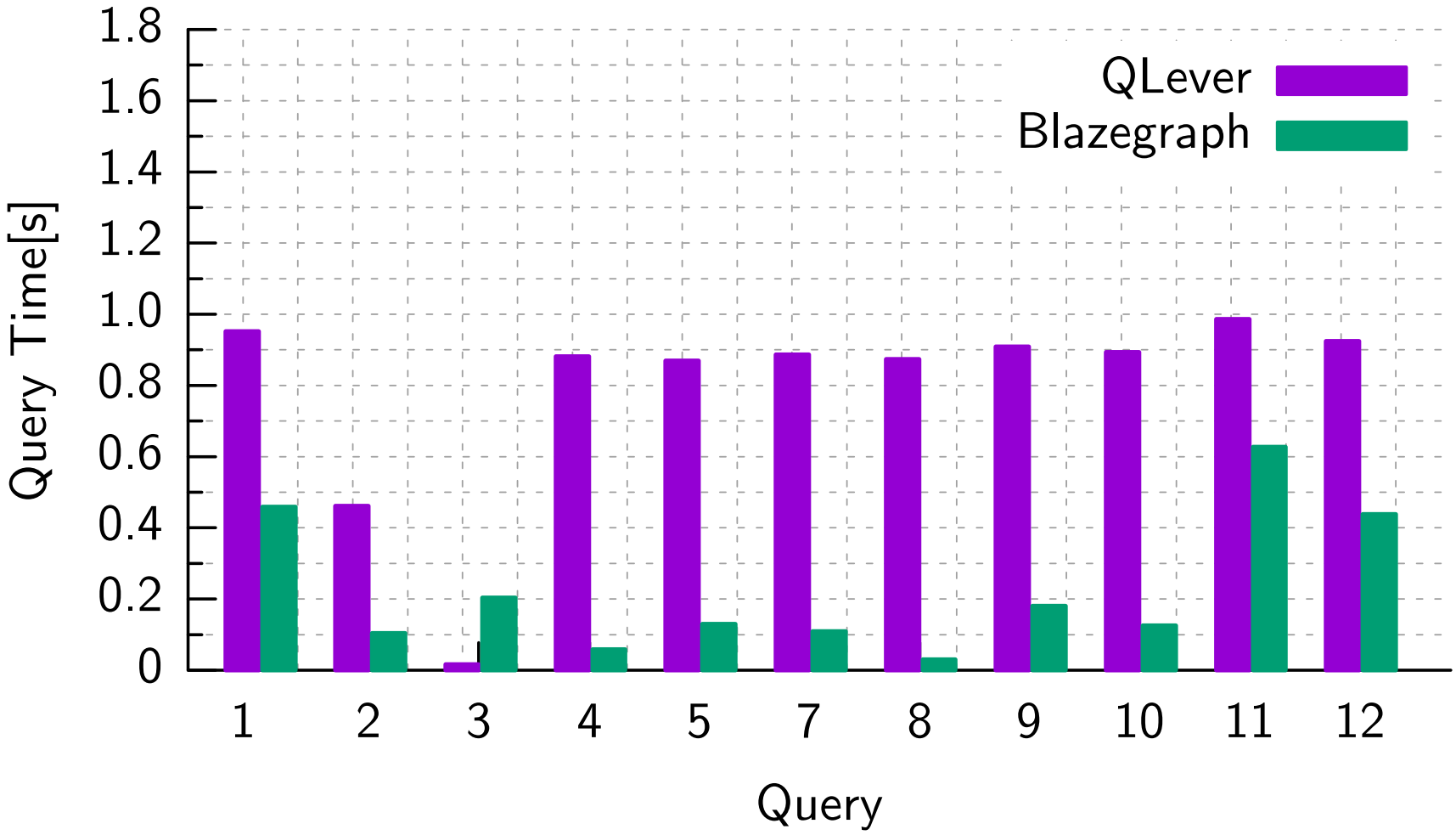
P31 instance of (52,739,893)

P279 subclass of (2,270,780)

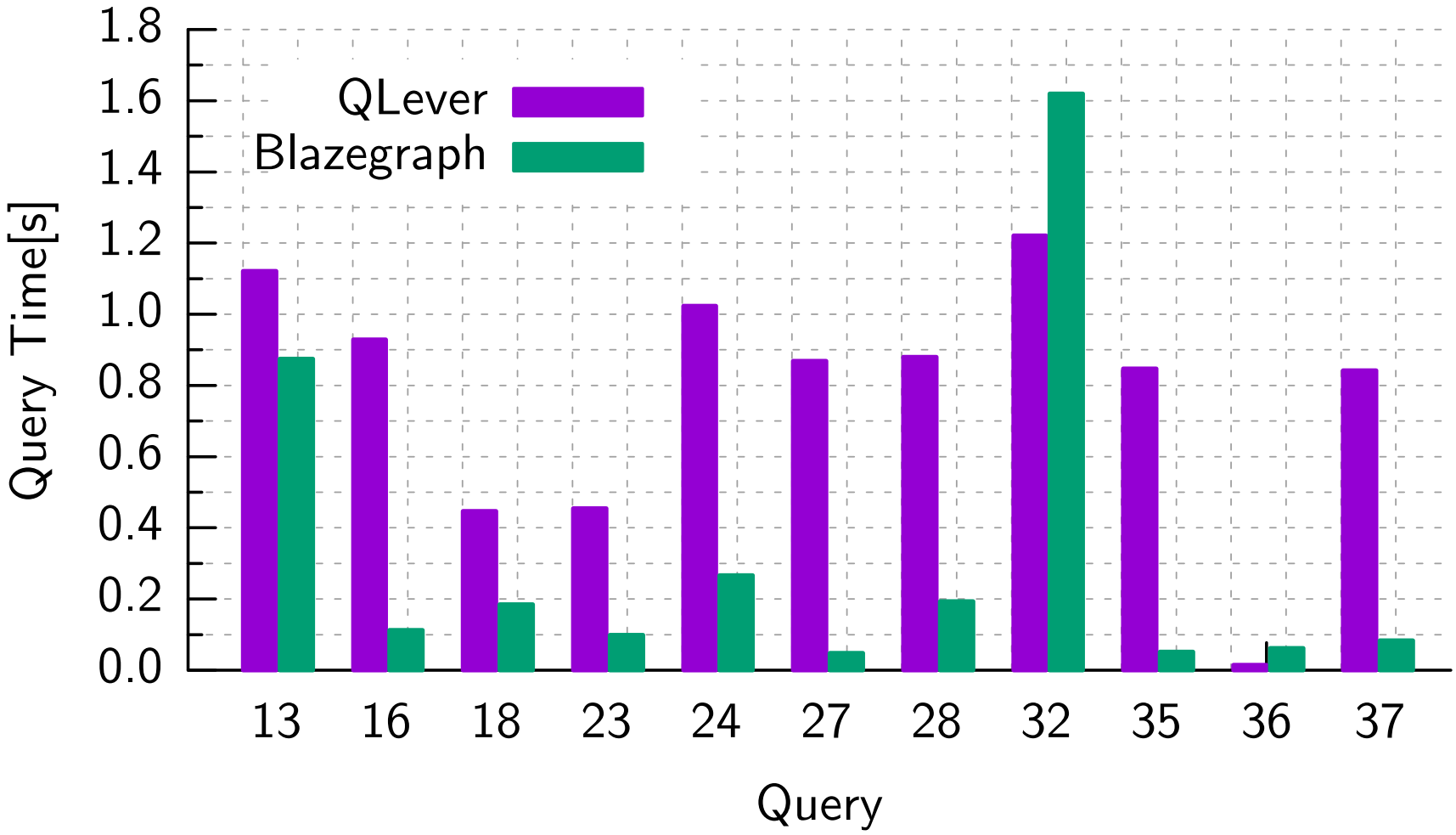
Result 60,057

- Qlever has to iterate both large relations
- The right side is fixed

Result Fast 1



Result Fast 2



Result Fast

Query	SPARQL	Result Size
3	`?c (wdt:P19)|(wdt:P20) wd:Q1741`	13,708
36	`?work (wdt:P37)|(wdt:P103) ?l`	6,952

P19 place of birth (2,344,764)

P20 place of death (890,695)

P37 official language (11,519)

P103 native language (89,015)

Bibliography I

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

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

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

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

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

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

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

Bibliography II

[8] 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.

[9] 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.

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

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

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

Questions?