Database Systems Implementation

Hilary Term 2022

Mini-project

Candidate Number: 1032514

FHS Mathematics and Computer Science

1 RDF Indexing Algorithm

This section contains my answers to Question 1. Section 1.1 answers Part (a); section 1.2 answers Part (b); and section 1.3 answers Part (c).

1.1 RDF Indexing Data-structure Overview

This section is about the RDF indexing data structure introduced in [1]. Here is more of a discussion of the algorithm, whereas more of my implementation-specific details can be found in section 2.6.

This data structure augments a naïve 'triples-table' storage with *three* linked list structures, as well as providing six indices over that data, to allow efficient answers to all eight possible triple pattern types[1].

Each triple t is augmented with pointers $N_{\text{sp}}, N_{\text{p}}, N_{\text{op}}$, which are all implemented as offsets in the triples table. The former forms a linked list over all triples with a given subject s, grouped by their predicate p; likewise for the pointers N_{op}. The pointer N_{p} forms a linked list over all triples with a given predicate p.

The six indices are denoted as $I_{\text{s}}, I_{\text{p}}, I_{\text{o}}, I_{\text{sp}}, I_{\text{op}}, I_{\text{spo}}$. Each index is implemented as a hash map, except the first three which are implemented as arrays indexed by the encoded subject/predicate/object. The first three indices give the start of the linked lists constructed via pointers $N_{\text{sp}}, N_{\text{p}}, N_{\text{op}}$ for a given subject/predicate/object, respectively. The indices I_{sp} and I_{op} give pointers to the specific *predicate groups* within the N_{sp} and N_{op} pointers, respectively. Finally, I_{spo} maps each triple to its offset in this table (of course, if there exists a triple with a given subject, predicate and object, there exists exactly one match in the table). In addition to storing pointers, the I_{s} and I_{o} indices keep track of the *sizes* of the lists they point to.

Notably absent is an I_{so} index; this is not used because it is practically the same size as the entire table. Very few subject-object pairs match lots of predicates.

1.2 The 'Add' Function

Initially, to perform duplicate removal, we look up a new triple in the I_{spo} index to see if it already exists. If so, we reject it and make no changes.

Now, assume the triple $\langle spo \rangle$ does not already exist in the database. Insert the triple to the table; suppose the index of this insertion is i.

Set the predicate pointer N_p of the newly-inserted triple as $N_p := I_{\text{p}}[p]$ and update the predicate index as $I_{\text{p}}[p] := i$. Increment the sizes stored at $I_{\text{s}}[s]$ and $I_{\text{o}}[o]$.

We now want to do similar updates with the subject and object, but we also have the I_{sp} and I_{op} indices to maintain. So, to begin with, similarly to before, set $N_{\text{sp}} := I_{\text{sp}}[s, p]$ and $N_{\text{op}} := I_{\text{op}}[o, p]$. In effect, we are inserting this triple to the 'front' of its sp-group and op-group. If the current subject-predicate pair has not been inserted before, instead we set $N_{\text{sp}} := I_{\text{s}}[s]$, and likewise for the object. Moreover, if the subject has never been inserted before, we instead set N_{sp} to some NULL value.

It is possible that the current subject-predicate pair exists in the DB already, but some other triple with the same subject (but with necessarily distinct predicate) points to the same destination as N_{sp}. This is bad because our pointers no longer form a linked list. We need to find this triple and update its pointer to point to i, the index of the new triple. I describe how to do this in section 2.6 in amortised O(1), and defer discussion of this issue to that section; but in essence, I solve it using another level of indirection, allowing N_{sp} to either hold a pointer to the table or an iterator in the I_{sp} index. The issue is symmetric for objects as well as subjects.

The final issue is how to update $I_{\text{s}}[s]$ (and likewise symmetrically for objects). We only want to update this to i when i's predicate is that of the foremost 'predicate group' for its given subject. This is certainly true if the subject-predicate pair did not exist previously, meaning its insertion into I_{sp} created a new key in the hash map. It is also true if the predicate of the element pointed to by $I_{\text{s}}[s]$ is equal to p. These are the only two cases where the new triple i is put to the front of its s-list, and hence the only cases where we update $I_{\text{s}}[s] := i$.

[1] There are three values in a triple (obviously), and each of which can either be variable or known, giving $2^3 = 8$ possible types of triple pattern.

1.3 The 'Evaluate' Function

As mentioned, there are only eight types of triple pattern. This is few enough that one can brute-force, to find the best plan for each one.

- If no parameters are known, we just perform a full table scan. If some of the variables are equal, e.g. if the triple is `?X ?Y ?X`, this requires additional postprocessing, but is still implemented with a full table scan.
- If all parameters are known and there are no variables, we just check the I_{spo} index for the presence of this tuple, and yield it if it exists.
- If the subject is the only variable, we use the I_{op} index and N_{op} pointers to find the first element of the linked list, and the next element, respectively. We stop when we encounter an element with a different predicate. Each element of the linked list gives us a value to bind for the subject variable.
- If the object is the only variable, the evaluation is symmetric.
- If the subject and predicate are the only variables, then we use the I_{o} index to find the first element of the linked list, and use the N_{op} pointers to find the remainder of the linked list. We do not stop if we encounter a new predicate; we only stop once we are at the end of the list. Each element of the list gives us values to bind for the subject and predicate variables; if these variables happen to be equal, we have to make an additional post-processing check to ensure their values are equal.
- The case where the object and predicate are the only variables is symmetric.
- If the predicate is the only known value, we use the I_{p} index to find the first element of the linked list, and use the N_{p} pointers to seek through this list. We bind the subjects and objects encountered to the two variables. Again, if these are the same variable (when the triple pattern is of the form `?X p ?X`), we have to make a check before allowing each triple to be yielded.
- If the predicate is the only variable, things are a tiny bit more complicated, because we don't have an I_{so} index. Instead, we lookup the subject and predicate in their respective I_{s} and I_{o} indices, and pick the one with the smaller size (recall that we store their sizes as well as their pointers). What follows then is a scan over the selected linked list, testing whether the object is equal to the input object (in the case of an I_{s} scan; for an I_{o} scan we test the subject) to ensure the yielded triple is valid.

2 Documentation

2.1 Overview

This section contains documentation of the main parts of the program (particularly where they are relevant to Question 2 in the exam paper), as well as compilation notes, and command line usage. For information on the code repository structure (i.e. what to find in each file), please consult the `README`.

In particular, section 2.6 is relevant to Question 2(a); section 2.6.2 is relevant to Question 2(b); section 2.8 is relevant to Question 2(c); section 2.9 is relevant to Question 2(d); the parser for SPARQL queries, for Question 2(e), can be found in `dbsi_query.cpp` but is otherwise unremarkable here; and finally, the command line interface is implemented in `dbsi_project.cpp` with comments on usage in section 2.3.

2.2 Compilation

This project is compiled using CMake. I have done so in such a way as to make compilation of this project essentially the same as compilation of the Database Systems Implementation course practicals. The `README` has some more details on compilation, but it should work as you would compile any other CMake project. **Please note that this project makes extensive use of C++17 features.** CMake should already detect and require this. No external or third party libraries are required.

2.3 Command Line Usage

In addition to the 'interactive'-style usage required in the exam specification, I have added some additional command line options, as well as a 'non-interactive' mode of usage for the program. Running `./dbsi_project -h` prints help info, which is a summary of what is contained in this section.

To use interactive mode, simply run the program with no arguments:

```
tr01[~/dbsi/dbsi_project_build/dbsi_project]$ ./dbsi_project
LOAD LUBM-001-mat.ttl
Loaded 137933 triples in 2706ms.
SELECT ?X WHERE { ?X ?X ?X }
----------
?X
----------
0 results obtained in 31ms (= 0ms planning + 31ms evaluation).
QUIT
Exiting...
```

Non-interactive mode is launched if the `-i` or `-f` options are present as command line arguments. The former provides a query input, and the latter specifies a file from which the program is expected to read query input(s). For example,

```
dbsi_project -i "LOAD␣family_guy.ttl" -f "/path/to/my/family/guy/queries.txt"
```

would load the *Family Guy* Turtle file, and then execute the queries in the text file at the given path.

Moreover, providing `-L` logs the join-planner's output, so you can see what type of query is actually being evaluated. To help make the program output easier to copy-paste into a spreadsheet, the `-P` option should be used when profiling `COUNT` queries (it just makes the output less verbose). If you are using any of these flags, they must appear before any/all instances of `-i` and `-f`.

2.4 Philosophy on Correctness

This section, which is orthogonal to database implementation, briefly comments on how I have tackled error-checking throughout the project. In `dbsi_assert.h` I have defined a set of macros and preprocessor definitions which are used in pretty much every `.cpp` file.

Where I write `DBSI_CHECK_PRECOND(expr)`, I mean "this condition should hold if you are using this class/function properly". `DBSI_CHECK_POSTCOND(expr)` means "this condition should hold if this function

I am relying on is implementing its advertised behaviour". `DBSI_CHECK_INVARIANT(expr)` means "this condition should hold if I have implemented this function as I intended to".

These are all disabled by default, however by commenting out `#define DBSI_DISABLE_ALL_CHECKS` in `dbsi_assert.h` you can perform these checks (and you can even toggle which ones to check). These are generally accompanied with a significant performance hit.

The main idea is that **in a release build, none of these should ever fail.** For errors in query processing and file loading, which cannot be ruled out even in a release build, these are handled either by using `std::optional` and/or logging a message to `std::cerr`, and failing gracefully. In particular, the program should never exit automatically or stop functioning without you closing the program, even in the case of an error.

2.5 Iterators

To keep query execution from using an amount of memory dependent on the database size, the iterator paradigm must be used throughout, whenever dealing with data. The templated-interface `IIterator<T>` in `dbsi_iterator.h` describes how iterators over a type `T` work in this project.

2.6 The RDF Index

2.6.1 Deviations from the Paper

The RDF index follows the approach of [1], described in section 1, almost entirely. However I did make some minor changes. Namely, the paper does not give explicit details on how triples are added to the database, so the solution I settled on required making some minor changes to keep additions amortised O(1).

The changes made were that the subject, predicate and object (I_s, I_p and I_o) indices are now hash maps, rather than arrays. This is so that we can store the `std::unordered_map`'s *iterators*, without insertions invalidating them.

Then, the N_{sp} and N_{op} pointers have also been changed. They hold exactly one of two types of values. Either an index to the main table (as they do in the paper), *or* they may hold an *iterator* from the respective I_{sp} and I_{op} indices. In this instance, the iterators can essentially be thought of as pointers.

The reasons why this is relevant is best illustrated by an example. Consider inserting the following (encoded) triples in order:

```
(1, 2, 3)
(1, 4, 3)
(1, 5, 3)
(1, 2, 5)
```

After inserting the first three triples, the I_s index maps subject 1 to the third triple. The N_{sp} pointers form a linked list from the third triple, to the second, to the first. When it comes to inserting the fourth triple, there are two reasonable places to insert it: either just *before the first* triple in the group $I_{\text{sp}}[1, 2]$, or just *after the last*. Doing the former requires scanning the index to find the triple which has N_{sp} pointing to this element, which, because it necessarily has a distinct predicate, is not easy to find (it would require scanning I_s); doing the latter requires scanning the entire group of triples with this subject-predicate pair which, while cheaper, is still not O(1). It depends on the sizes of the subject-predicate groups.

To fix this issue, I did the following. Whenever the N_{sp} pointer maps to a triple with a distinct predicate, it does not store that triple's index, but it instead stores its *iterator* in the I_{sp} index. This is equivalent to storing its predicate, but this is faster, because when it comes time to follow the N_{sp} pointer, we do not need to search the index but simply dereference the iterator. When new values are inserted to the I_{sp} index which alter this value, it doesn't matter because the iterator is not invalidated and still works.

For more details, see `RDFIndex::add`, which has extensive comments, including extending the above discussion to rules on when to update the I_s index. Of course, all of the above applies equally to objects as it does to subjects. Moreover, I implemented a function `RDFIndex::check_integrity` which tests the entire structural integrity of this datastructure, which is enabled only in debug mode.

2.6.2 Access-planning and evaluation

The `RDFIndex`'s implementation of `EVALUATE` is quite short. However this is only because most of the work has been offloaded to one of its subclasses, `RDFIndex::IndexIterator`. When you evaluate a triple pattern at the RDF index, it first looks at the structure of the triple pattern, and then gives instructions to the `IndexIterator`, which then follows these instructions during execution. Therefore, this iterator objects acts as the 'engine' for evaluating SPARQL queries.

The RDF index's `plan_pattern` function returns `std::pair<IndexType, EvaluationType>` which (i) says how to find the first triple, and (ii) how to go from one triple to the next. The RDF index handles (i) and therefore provides the iterator with a starting point computed from these indices. This way, the iterator doesn't need to concern itself with anything other than the triples table. Moreover, for (ii), the RDF index passes the evaluation type to the iterator, which tells the iterator what to do every time it finishes processing a triple (when `->next()` is called): use one of the linked-list pointers N_{p}, N_{sp}, N_{op}, or simply increment the index, or simply terminate after returning a single triple.

2.7 Nested Loop Join

While naturally a recursive procedure, nested loop join (the class `NestedLoopJoinIterator` found in the file `dbsi_nlj.cpp`) has been implemented here using a stack of 'sub-iterators', one for each triple pattern in the join. This stack of iterators is always either empty (when the join has finished), or full (when the join is in progress). When any of the iterators in this stack becomes invalidated, it is removed, and then the iterator at the next level up of the recursion is stepped (via `->next()`) and then this updates the current variable set, which then allows the RDF index to create another iterator. See the function `NestedLoopJoinIterator::update_iterators` for more details.

I have designed this class in a way which makes it agnostic to *how* the RDF index implements each triple pattern; it always queries the `RDFIndex` for new iterators executing a given triple pattern, and these iterators don't expose *how* they are achieving their triple-pattern-matching.

2.8 Greedy Join Optimisation

This functionality, implemented as `dbsi::joins::greedy_join_order_opt` in `dbsi_nlj.cpp`, operates on a list of triple patterns, before they are evaluated. It implements the same method given in the exam specifications, also found in [2].

2.9 Turtle File Loading

Loading Turtle files is done by constructing an instance of a class `TurtleTripleIterator` which creates an iterator interface over a Turtle file, loading it in such a way that no more than one triple is held in memory (by this class) at any one time. If the iterator encounters an error, it stops processing queries and marks itself as 'done' to the outside world, but logs an error message to `std::cerr`. While only used for parsing *files*, this takes an arbitrary `std::istream` as input, so it could technically also be used for loading triples from `std::cin` as well.

Its output is produced in decoded format, where it is then passed to the dictionary to encode it. This is done using a handy function called `autoencode`, which takes as input (i) the dictionary, and (ii) an iterator over non-coded triples, and returns an iterator over automatically-encoded triples. Along with the corresponding function `autodecode`, this makes it very easy to go back-and-forth between encoded and decoded versions of iterators.

Machine	CPU Type	Number of Cores	Clock Speed (MHz)	L1/L2/L3 Cache Sizes	RAM
CS Dept. Lab Machine TR01	Intel(R) Xeon(R) CPU E5-2660 v3	4	2600	128KiB / 1MiB / 100MiB	32 GB
My Desktop	AMD Ryzen 3 1300X	4	4000	384KB / 2MB / 8MB	8 GB

Figure 1: Device statistics.

3 Benchmarking and Results

3.1 Machines Tested

I tested all experiments on two machines. My desktop, a Windows machine, and the CS department lab machines. Statistics for these machines can be found in fig. 1.

My desktop has both an SSD and HDD. I tested both to see if there was much of a difference to load times and, surprisingly, there was not. All results thereafter were done on the HDD.

3.2 Test Protocol

To test the performance of the program, I would first ensure that there are no unnecessary programs running on the device. Then, I would invoke the executable (in non-interactive mode, providing all arguments up-front), which will then load the right data file and then execute queries 1-14 in turn. Before running the tests, the queries were modified by replacing `SELECT` with `COUNT` (but were not changed in any other way). This is to eliminate overhead associated with printing the results.

This was executed for the small data file (`LUBM-001-mat.ttl`), followed by the medium one (`LUMB-010-mat.ttl`), followed by the large one (`LUBM-100-mat.ttl`). This overall process is then repeated five times. An example command which executes each query on the small dataset is the following:

```
./dbsi_project -P -i "LOAD␣../../LUBM-001-mat.ttl" -f "../../queries/q01.txt" -f "
    ../../queries/q02.txt" -f "../../queries/q03.txt" -f "../../queries/q04.txt" -f "
    ../../queries/q05.txt" -f "../../queries/q06.txt" -f "../../queries/q07.txt" -f "
    ../../queries/q08.txt" -f "../../queries/q09.txt" -f "../../queries/q10.txt" -f "
    ../../queries/q11.txt" -f "../../queries/q12.txt" -f "../../queries/q13.txt" -f "
    ../../queries/q14.txt"
```

During the development process, I noticed that all commands (including difficult queries, and loading the large file) never took longer than around 15 minutes. As a result, I deemed it unnecessary to set a timeout limit, because I expected everything to finish in a reasonable amount of time. Indeed, this turned out to be OK.

3.3 Benchmarking Experiment Results

The results of the benchmarking experiments can be found in fig. 2. We clearly see that larger datasets have substantially slower load and execution times, roughly proportional to the 10x size increase between the small, medium and large datasets (`LUBM-001-mat.ttl`, `LUBM-010-mat.ttl` and `LUBM-100-mat.ttl`, respectively). There is a large gap in performance between the lab machines and my desktop, however the lab machines are still quite responsive on the small dataset.

Interestingly, I observed an almost tenfold increase in performance on my desktop compared to the lab machines, even though my desktop is far from 10x more powerful than the lab machines. Since 8GB RAM is large enough to hold even the largest dataset, and the clock speed is only approximately twice as powerful, I can only conclude that most of the difference is due to the larger L1 and L2 caches.

The slowest operation of all shown in this table is the loading of the large dataset on the lab machine; this takes about six minutes. The shortest operations are queries 12 and 13 on the small dataset, which are completed in under a millisecond on my desktop.

3.4 Discussion

This section contains a brief description and analysis of the performance on each query. The join type selected for each query can be found in fig. 3, where each triple pattern is represented as a three-letter string with 'V' denoting 'Variable', and 'S', 'P' and 'O' denoting 'subject', 'predicate' and 'object', respectively. For example, 'VVV' requires a full table scan, but 'SPO' just requires a single lookup to the SPO-index to check whether that triple exists in the database.

To see what join the program will use, you do not need to load any data. Using the `-L` option on the command line will print the join selected for any query you give it. For example, to obtain the join type for query 2, you would run the following:

```
dbsi_project -L -f "E:/dbsi/queries/q02.txt"
```

which outputs

```
        --> NLJ over patterns with (conditional) types VPO SPV SPO SPV SPO SPO
0 results obtained in 0ms (= 0ms planning + 0ms evaluation).
```

This means that the nested loop join starts with an outermost triple pattern of type `VPO`, which then binds whichever variable is in the subject-position. The next triple pattern in the loop is `SPV`, *after* binding the variable from the first triple pattern. The rest follows similarly. Join types will generally end with a bunch of `SPO`s, because by that point most variables are typically bound with values.

3.4.1 Query 1

This query was a pair of VPO triple patterns. As a result, both two join orders result in a join of type 'VPO SPO'. This involves first using the OP-index, then scanning all valid subjects by following the N_{op} linked list pointers. Each subject produced by doing this is then looked-up in the SPO-index, with different predicate and object, to check it exists, and if so, is included in the output.

3.4.2 Query 2

This query involved a 6-way join. Its 'shape' is a triangle of 3 joins, each sharing exactly one variable with the others (XY, YZ and XZ), and then being joined with two other triple pattern each (XY with X and Y; XZ with X and Z; YZ with Y and Z). The selected join order was 'VPO SPV SPO SPV SPO SPO', which means that it selected one of the 1-variable triple patterns to evaluate first, then pushed this value into one of the 2-variable triple patterns (binding two of three variables). It follows this with an 'SPO', which means it must be checking against one of the other 1-variable triple patterns. Finally, it looks up one of the remaining two 2-variable triple patterns to bind a value for the third variable. There are two relations left by this point, which are then simple 'SPO's.

3.4.3 Query 3

Functionally the same as query 1.

3.4.4 Query 4

This query involves a 5-way join. First, the program evaluates one of the 'VPO' triple patterns (which has `?X` as its subject-variable). Whichever value it gets, it checks against the other 'VPO' triple pattern, which by this point becomes an 'SPV' because we have bound a value for `?X`. The remaining three triple patterns are 'SPV', but the variable is different in each case, so it is essentially a cross product (after substituting in `?X`).

3.4.5 Query 5

Functionally the same as query 1.

3.4.6 Query 6

This is just a simple 'VPO' type.

3.4.7 Query 7

This query consists of a 4-way join. It is one of the more expensive queries in this sample. The first two triple pattern types selected, after substituting variables, are 'VPO SPV'. Since `?Y` is the only variable to appear in the object position, this means it first finds values for `?X`. There are then two triple patterns it can choose from to find a value for `?Y`. After doing this, there are two final 'SPO' triple patterns to check the triples exist.

3.4.8 Query 8

This query consists of a 5-way join. The query planner chooses first the 'VPO' triple pattern only involving `?X`, then using this value evaluates the triple pattern involving both `?X` and `?Y`. Then, knowing this value for `?Y`, performs two 'SPO' checks to see if that value for `?Y` was valid. Finally, given the value of `?X`, an 'SPV' triple pattern is evaluated to get all of the possible values for `?Z`.

It is promising that the query planner leaves this `?Z` variable until last, because it means we don't need to do work finding it if there is no valid corresponding `?Y` value. However we aren't saving too much, because the predicate (looks like) it is selecting the email address `?Z` for the corresponding student `?X`, which there is likely at most one of.

3.4.9 Query 9

This query consists of a 6-way join. It follows the same triangular structure as query 2, and the planner selects the same join type. However, across the board, it is cheaper to evaluate (by a factor of 2 in some cases), whilst also producing more results. Therefore the difference in performance between the two must be down to the selectivities of the given triple patterns.

3.4.10 Query 10

Functionally the same as query 1. It is probably more expensive than query 1 because it produces more tuples; query 1 is essentially "get all graduate-student-taking-graduate-course" pairs, and query 10 is a relaxation of this to "get all *student*-taking-graduate-course" pairs.

3.4.11 Query 11

Functionally the same as query 1. But is, in general, cheaper than query 1. This is because the cardinalities involved are much smaller: query 1 is about "students" but query 11 is about "research groups", of which there are fewer.

3.4.12 Query 12

Very similar to query 7. However, this query is much cheaper than query 7 (which happens to be the second most expensive overall). This is again because the cardinalities involved are very different. Query 7 is about students but query 12 is about departmental chairs.

3.4.13 Query 13

Very similar to query 1. While it does not produce many results, it is still more expensive than query 1. This is possibly because the query planner gets the two triple patterns the wrong way around. The first triple pattern in query 13 "checks that `?X` is a person", and the second "checks that `?X` is an alumnus of a specific, given university". It is almost surely the case that every alumnus is already a person, so the first triple pattern is probably redundant (although the query optimiser cannot know this). But for sure, there are more people than alumni, so the second triple pattern is much more selective. Swapping their order would therefore be better.

3.4.14 Query 14

Functionally the same as query 6, and with a broadly similar execution cost, too.

3.5 Performance Profiling Results

In addition to the benchmarking experiments, I also ran my application through a profiler to see where the bottlenecks were. This allowed me to make several small tweaks to improve performance by about threefold after I'd implemented everything.

The current bottlenecks are primarily loading (particularly the function `dbsi::parse_resource` in `dbsi_parse_helper.cpp`), and the remaining functions where the program spends most of its time are from `std::unordered_map`. I used the results of early profiling experiments to optimise the loading process significantly, for example by *not* using `in >> std::ws` to skip whitespace but instead implementing my own, and also reducing memory allocations during string construction with a `static std::vector<char>` instead of a `std::stringstream`.

I have decided to not try optimising the `std::unordered_map`, because I am relying on the C++ standard library to have an efficient implementation of a hash map upon which I am unlikely to improve by writing my own. The only argument to be made is about the choice of the hash function for the various `std::unordered_map` data structures used in the RDF index. I did experiment with some more 'optimal' hash functions (in the sense of 'better bit scrambling'), however these hash functions are called so often that anything not completely trivial is too slow, even if there are more even hash bucket distributions. I settled on the following hash function for a `size_t sub, pred, obj` triple, which can be found in `dbsi_types.h`:

```
(sub << (2 * sizeof(size_t) / 3))
^ (pred << (sizeof(size_t) / 3))
^ obj
```

The results of my profiling tests can be found in fig. 4. In particular it illustrates what is mentioned above, that the most expensive functions are `dbsi::parse_resource`, which reads a resource from the input file stream and checks for errors (e.g. a missing angular bracket when reading an IRI), and `dbsi::RDFIndex::add`, which is the function for adding an encoded triple to the index.

References

[1] Boris Motik et al. "Parallel materialisation of datalog programs in centralised, main-memory RDF systems". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 28. 1. 2014.

[2] Petros Tsialiamanis et al. "Heuristics-based query optimisation for SPARQL". In: *Proceedings of the 15th International Conference on Extending Database Technology*. 2012, pp. 324–335.

Query	TR01 (Small)	TR01 (Medium)	TR01 (Large)	My Desktop (Small)	My Desktop (Medium)	My Desktop (Large)
Load	2711.8 +/- 30.5	35108.8 +/- 315.1	377544 +/- 3489.4	818.5 +/- 3.3	10738.5 +/- 45.7	128208 +/- 2852.4
1	24.8 +/- 0.4	313.8 +/- 1.9	3328.8 +/- 22.8	2 +/- 0	30 +/- 0	482 +/- 58.3
2	1269.8 +/- 15.7	16240.4 +/- 84	171084.8 +/- 3834.8	122.5 +/- 1.1	1580.5 +/- 11.3	16786 +/- 45.7
3	78.2 +/- 2.3	998.4 +/- 13.8	10759.4 +/- 416	6.5 +/- 0.5	92.5 +/- 0.5	1208 +/- 53.5
4	10 +/- 0	78.6 +/- 0.9	795.8 +/- 17.6	1 +/- 0	7.5 +/- 0.5	80.5 +/- 1.2
5	113.4 +/- 2.9	1397.4 +/- 11.7	14670.2 +/- 234.2	10 +/- 0.4	128 +/- 0.5	1488 +/- 53
6	44.8 +/- 0.4	565.6 +/- 6.8	6055.8 +/- 95.7	8 +/- 0.4	113.5 +/- 0.5	1473 +/- 69.3
7	841.6 +/- 23.8	10224.2 +/- 105	109306.4 +/- 2368.4	93 +/- 0.4	1149 +/- 7.6	12433 +/- 85.7
8	698.6 +/- 6.7	4876.8 +/- 27.2	49471.6 +/- 721.6	91.5 +/- 0.5	566 +/- 4.2	5591.5 +/- 11.8
9	640.2 +/- 11	7614 +/- 99.2	81375 +/- 1233.5	80 +/- 0	963 +/- 2.3	10492 +/- 70.7
10	103 +/- 1.9	1312.6 +/- 39.6	13958.2 +/- 333.4	9 +/- 0	122 +/- 1.2	1326.5 +/- 20.2
11	4 +/- 0	38.4 +/- 0.9	396.2 +/- 5.4	0 +/- 0	3 +/- 0	37 +/- 0.4
12	1 +/- 0	12 +/- 0	131.8 +/- 3.3	0 +/- 0	1 +/- 0	17 +/- 0
13	109.2 +/- 2.3	1385.2 +/- 8.8	14889.6 +/- 352.8	9 +/- 0.4	128.5 +/- 0.7	1368 +/- 3
14	34.2 +/- 1.6	430.8 +/- 2.9	4560.6 +/- 55.8	6 +/- 0	83.5 +/- 0.5	885 +/- 4

Figure 2: Results of the benchmarking tests, for Lab Machine TR01 and my desktop, for loading the datasets and for queries 1-14. All times displayed in milliseconds, in the format 'avg +/- std'.

Query	Join Type
1	VPO SPO
2	VPO SPV SPO SPV SPO SPO
3	VPO SPO
4	VPO SPO SPV SPV SPV
5	VPO SPO
6	VPO
7	VPO SPV SPO SPO
8	VPO SPV SPO SPO SPV
9	VPO SPV SPO SPV SPO SPO
10	VPO SPO
11	VPO SPO
12	VPO SPV SPO SPO
13	VPO SPO
14	VPO

Figure 3: The type of join selected for each query. Note: this includes variable substitutions, so if a variable is bound in an earlier part of the join, it appears as known in its later appearances. This makes the join types representative of exactly what the program is doing when it's executing them.

Name	Inclusive (s)	Exclusive (s)	Inclusive (percent)	Exclusive (percent)
dbsi::parse_resource	1.689335	11.843606	6.934746	48.618186
dbsi::RDFIndex::add	1.116094	4.091761	4.581583	16.796742
dbsi::Dictionary::encode	0.064048	1.982312	0.262918	8.137421
dbsi::TurtleTripleIterator::read_triple	0.038015	8.57591	0.156052	35.204243
dbsi::TurtleTripleIterator::valid	0.03299	0.03299	0.135424	0.135424
dbsi::encode	0.025036	2.007348	0.102773	8.240195
dbsi::autoencode::2::AutoencodingTripleIterator::current	0.023985	2.416582	0.098459	9.920107
QueryApplication::operator()	0.018968	19.401138	0.077864	79.641972
dbsi::TurtleTripleIterator::current	0.012031	0.14412	0.049387	0.591615
dbsi::autodecode	0.011986	3.946018	0.049203	16.198465
dbsi::autoencode::2::AutoencodingTripleIterator::next	0.006	0.006	0.02463	0.02463
dbsi::autoencode::2::AutoencodingTripleIterator::valid	0.003	0.003	0.012315	0.012315
dbsi::TurtleTripleIterator::next	0.001985	0.324898	0.008148	1.333711
dbsi::RDFIndex:: RDFIndex	0	0.327189	0	1.343116
main	0	19.751311	0	81.079438

Figure 4: A subset of the profiling results. 'Inclusive' denotes the time spent in the function but not in child calls; 'exclusive' includes child calls. For example, the main function has almost no inclusive time, but its exclusive time is the largest, because obviously, the program spends almost all of its execution time in a child call of the main function. I have only included functions from the profiling output that I have written (so, excluding functions from the standard library and the kernel). These are then sorted according to their inclusive time.