Bachelor's Thesis

A Macro-Benchmarking Library for the QLever SPARQL Engine

Andre Schlegel

June 20, 2024

Submitted to the University of Freiburg
Institute for computer science at the University of Freiburg
Chair of Algorithms and Data Structures

universität freiburg

University of Freiburg
Institute for computer science at the University of Freiburg
Chair of Algorithms and Data Structures

Author Andre Schlegel,
Matriculation Number: 4515013

Editing Time March 20, 2024 - June 20, 2024

Examiners Prof. Dr. Hannah Bast,
Institute for computer science at the University of Freiburg
Chair of Algorithms and Data Structures

Supervisor Johannes Kalmbach,
Institute for computer science at the University of Freiburg
Chair of Algorithms and Data Structures

Declaration I hereby declare, that I am the sole author and composer of this 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 hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, Date

Signature

Abstract

Comparing execution times of different algorithm implementations can be time and work intensive. This time and work is needed to cover every possible, generalized situation for the different algorithm implementations, otherwise, no true statements can be made about the execution times of different algorithm implementations. To make covering every possible, generalized situation for the different algorithm implementations easier, I've built an internal macro-benchmarking library for the, as named on the project page of [1], "QLever SPARQL engine". The internal macro-benchmarking library for the QLever SPARQL engine offers a simple method for measuring the execution time of computer software in seconds, multiple ways to organize measured execution times, metadata support, and support for user-defined runtime configuration options, which should simplify the generation of measured execution times.

Contents

Figure Index

Table Index

Code Index

Chapter 1

Introduction

Reducing the execution times for queries to the QLever SPARQL engine requires reducing the execution times of algorithm implementations in the QLever SPARQL engine. Reducing the execution time of an algorithm implementation in the QLever SPARQL engine, is done by either changing the used algorithm, or by changing the implementation of the algorithm.

It is not always apparent if the changed implementation or changed algorithm always has a reduced execution time, only has a reduced execution time in specific situations, or never has a reduced execution time. To tell if, and when, the execution time is reduced, the execution time must be measured in every possible, generalized situation the changed implementation or changed algorithm may encounter. Such measurements are done with a set of benchmarks. Source [2] defines a "benchmark" as, "a computer program that measures the ... speed of computer software ... ". In this thesis, a benchmark measures the execution time of computer software. A set of benchmarks is called a benchmark suite. A benchmark suite should contain at least one benchmark for every possible, generalized situation the changed implementation or changed algorithm may encounter. However, creating a benchmark suite with at least one benchmark for every possible, generalized situation the changed implementation or changed algorithm may encounter takes a lot of time.

Generating a benchmark suite with at least one benchmark for every possible, generalized situation the changed implementation or changed algorithm may encounter via an algorithm is not possible. Because generation via an algorithm is not possible, I've tried to reduce the construction time by building an internal macro-benchmarking library for the QLever SPARQL engine.

A macro benchmark is designed to measure long execution times. A long execution time, for example, would be one second. A long execution time tends

to correlate to a large workload. Source [3] defines “workload” as, “the amount of work to be done, especially by a ... machine in a period of time ... ”. In this thesis, the machine referenced is always a computer.

Currently, my internal macro-benchmarking library for the QLever SPARQL engine enables its user to measure execution time in seconds, organize macro benchmarks, define configuration option for use at runtime, and define metadata. The organized macro benchmark measurements, defined runtime configuration options, and defined metadata can be displayed in a human-readable format. The organized macro benchmark measurements and defined metadata can also be exported to a “JSON” file [4].

Third-party benchmarking libraries could be used for benchmark suite construction instead of my internal macro-benchmarking library for the QLever SPARQL engine. However, all third-party benchmarking libraries that I could find are micro benchmarking libraries. A micro benchmarking library is designed to measure very short execution times, often in the range of milliseconds, as accurately as possible. The QLever SPARQL engine has a focus on short execution times for big workloads. However, in the context of the QLever SPARQL engine, a short execution time for a big workload can mean multiple seconds to multiple days. Multiple seconds to multiple days are not a good fit for micro benchmarks, who are designed for execution times in the range of milliseconds. Multiple seconds to multiple days are better measured with a macro benchmarks, who are designed for long execution times in the range of one second and longer. Therefore, when writing a benchmark for an implemented algorithm in the QLever SPARQL engine with the same focus as the QLever SPARQL engine, a short execution time of multiple seconds to multiple days for a big workload, a macro benchmark is a better fit than a micro benchmark.

Chapter 2

Key Ideas

In this section, I will walk through all the functionality my internal macro-benchmarking library for the QLever SPARQL engine provides.

First off, a few short definitions:

- Benchmark: Source [2] defines a "benchmark" as, "a computer program that measures the ... speed of computer software ... ". In this thesis, a benchmark measures the execution time of computer software.
- Benchmark suite: Set of benchmarks.
- Metadata: Source [5] defines "metadata" as, "information that is given to describe or help you use other information ... ".
- Workload: Source [3] defines "workload" as, "the amount of work to be done, especially by a ... machine in a period of time ... ". In this thesis, the machine referenced is always a computer.

The functionality of my internal macro-benchmarking library for the QLever SPARQL engine can be split into three sections:

1. Section 2.1: Creating one, or more, macro benchmark suites.
2. Section 2.2: Defining runtime configuration options.
3. Section 2.3: Executing one, or more, macro benchmark suites.

For every section, I will go through a UML use case diagram overview and explain the UML use case diagram overview.

2.1 Macro Benchmark Suite Creation

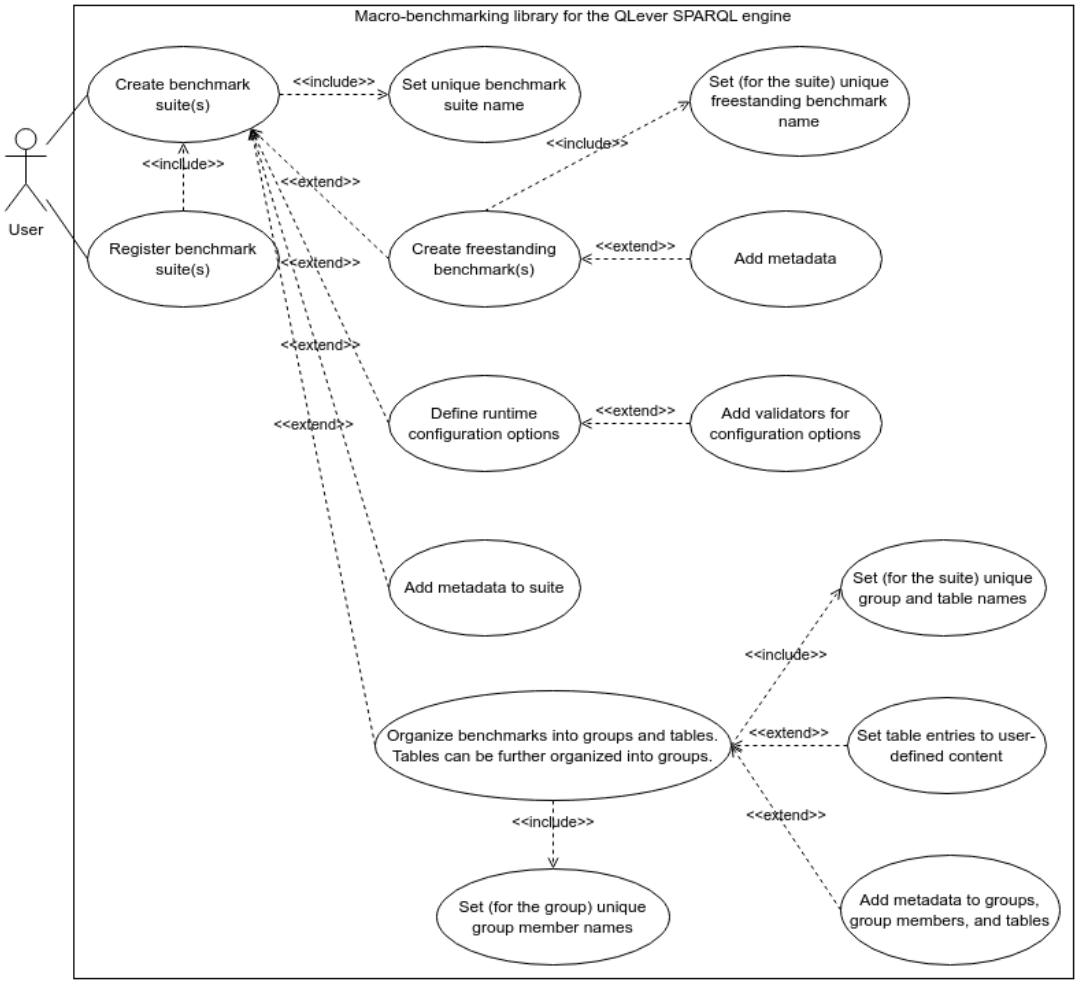

Figure 2.1: UML use case diagram overview for macro benchmark suite creation.

In this section, I will go through the creation of macro benchmark suites inside my internal macro-benchmarking library for the QLever SPARQL engine.

Firstly, a user creates zero or more empty macro benchmark suites. Every macro benchmark suite has to be uniquely named by the user. This unique name is used to differentiate between different macro benchmark suites when executing one or more macro benchmark suites. For more information about the execution of macro benchmark suites see section 2.3.

Inside a macro benchmark suite, a user can create zero or more freestanding macro benchmarks. Freestanding macro benchmarks are normal macro benchmarks, that are not organized in any structure. Every freestanding macro bench-

mark has to be named by the user. The name of a freestanding macro benchmark must be unique inside the macro benchmark suite containing the freestanding macro benchmark. Metadata can be added to a freestanding macro benchmark.

Inside a macro benchmark suite, runtime configuration options can be defined. If a user created more than one macro benchmark suite, all the macro benchmark suites need to have the same runtime configuration options. For more information see section 2.2.

Inside a macro benchmark suite, metadata can be added to the macro benchmark suite itself. This metadata normally describes general information. General information is information, that is shared by all macro benchmarks in the macro benchmark suite.

Inside a macro benchmark suite, a user can create zero or more table for the organization of macro benchmarks. Every entry in a table is either a macro benchmark measurement or set by a user. An entry set by a user can be a boolean, a string, or a rational number. Every table has to be named by the user. The name of a table must be unique inside the macro benchmark suite containing the table. Metadata can be added to a table. The number of columns in a table must be set when creating a table and can not be changed, as are the names of the columns in a table. However, the number of rows in a table can be set when creating a table and can be changed after creating the table. Names for the rows in a table are optional. Names for the rows in a table can be set when creating a table and can be changed after creating the table.

Inside a macro benchmark suite, a user can create zero or more groups for the organization of macro benchmarks and tables. Every group has to be named by the user. The name of a group must be unique inside the macro benchmark suite containing the group. Metadata can be added to a group. A group is a set of group members. A group member can be a macro benchmark or a table. Every group member has to be named by the user. The name of a group member must be unique inside the group containing the group member. Metadata can be added a to a group member.

After a user creates zero or more empty macro benchmark suites and defines the inside of the macro benchmark suites, the user can register none, only a part, or all of the created macro benchmark suites. Only registered macro benchmark suites will be executed later. For more information see section 2.3.

2.2 Defining Runtime Configuration Options

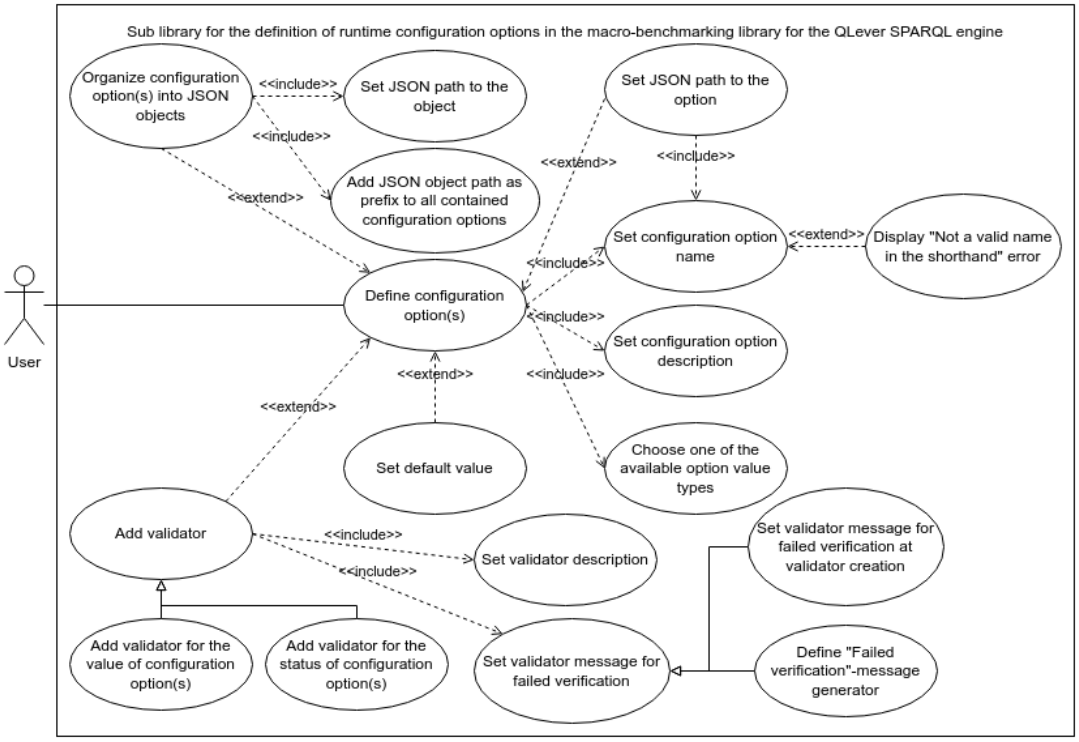

Figure 2.2: UML use case diagram overview for runtime configuration option definition.

In this section, I will go through the process of defining runtime configuration options inside my internal macro-benchmarking library, for the QLever SPARQL engine.

Runtime configuration options are, in the context of my internal macro-benchmarking library for the QLever SPARQL engine, variables inside a compiled program, that can be set by a user at runtime from outside the compiled program.

Runtime configuration options are organized within a recursive “JSON” object structure [4]. A recursive JSON object structure is a JSON object, that can contain only key value pairs with the value a recursive JSON object structure or a runtime configuration option. A user can set the runtime configuration options at runtime either by passing a JSON file, or by passing a JSON-style shorthand string via the CLI. For more information about setting runtime configuration options at runtime see section 2.3.

A user can define zero, or more, runtime configuration options. When defining a runtime configuration option, a user must:

- Define an array of strings describing the position of the runtime configuration option in the recursive JSON object structure. I call this array of strings a JSON path. A JSON path must contain at least one string. Iteratively using the JSON path strings as keys inside the recursive JSON object structure, will point to the runtime configuration option. However, the JSON path to a runtime configuration option is not allowed to be a prefix of an JSON path to a different runtime configuration option. The last string of the JSON path to a runtime configuration option will be used as the name for the runtime configuration option.
- Write a description for the runtime configuration option. The description is later used in the generated documentation. For more information about the generated documentation see the section 2.3.
- Choose one of the available types for the value of the runtime configuration option. The available types for the value of the runtime configuration option are boolean, string, number, array of boolean, array of string, or array of number. The runtime configuration option can only be set to a value of the type chosen for the runtime configuration option.

Optionally, a user can give a default value for the runtime configuration option. When the runtime configuration option is not set at runtime and a default value for the runtime configuration option was given, the runtime configuration option will be set to the default value. When no default value for the runtime configuration option was given, the runtime configuration option must be set at runtime.

A user can create zero, or more, JSON objects inside the recursive JSON object structure. Runtime configuration options can be defined inside those JSON objects. JSON objects inside the recursive JSON object structure are used to organize runtime configuration options. When creating a JSON object inside the recursive JSON object structure, the user must give an array of strings describing the position of the JSON object in the recursive JSON object structure. I call this array of strings a JSON path. A JSON path must contain at least one string. The JSON path to the JSON object is added as a prefix to the JSON paths of all the runtime configuration options contained inside the JSON object.

After defining the runtime configuration options, the user can add zero or more validators. A validator, in the context of my internal macro-benchmarking library for the QLever SPARQL engine, checks the validity of the status of a runtime configuration option at runtime or checks the validity of the value a

runtime configuration option was set to at runtime. A validator can check the validity of one, or more, runtime configuration options at runtime.

When defining a validator, a user must:

- Give the runtime configuration options, a validator checks the validity of at runtime, to the validator.
- Write a description for the validator. The description is later used in the generated documentation. For more information about the generated documentation see section 2.3.
- Write a failure message. The failure message is shown when the validator checks the validity of runtime configuration options and the runtime configuration options do not pass. For more information see section 2.3. The failure message can be a fixed message or a message generated by the validator. A message generated by the validator can include more information. For example: Has the runtime configuration option a default value and was the value of the runtime configuration option set at runtime?

2.3 Macro Benchmark Suite Execution

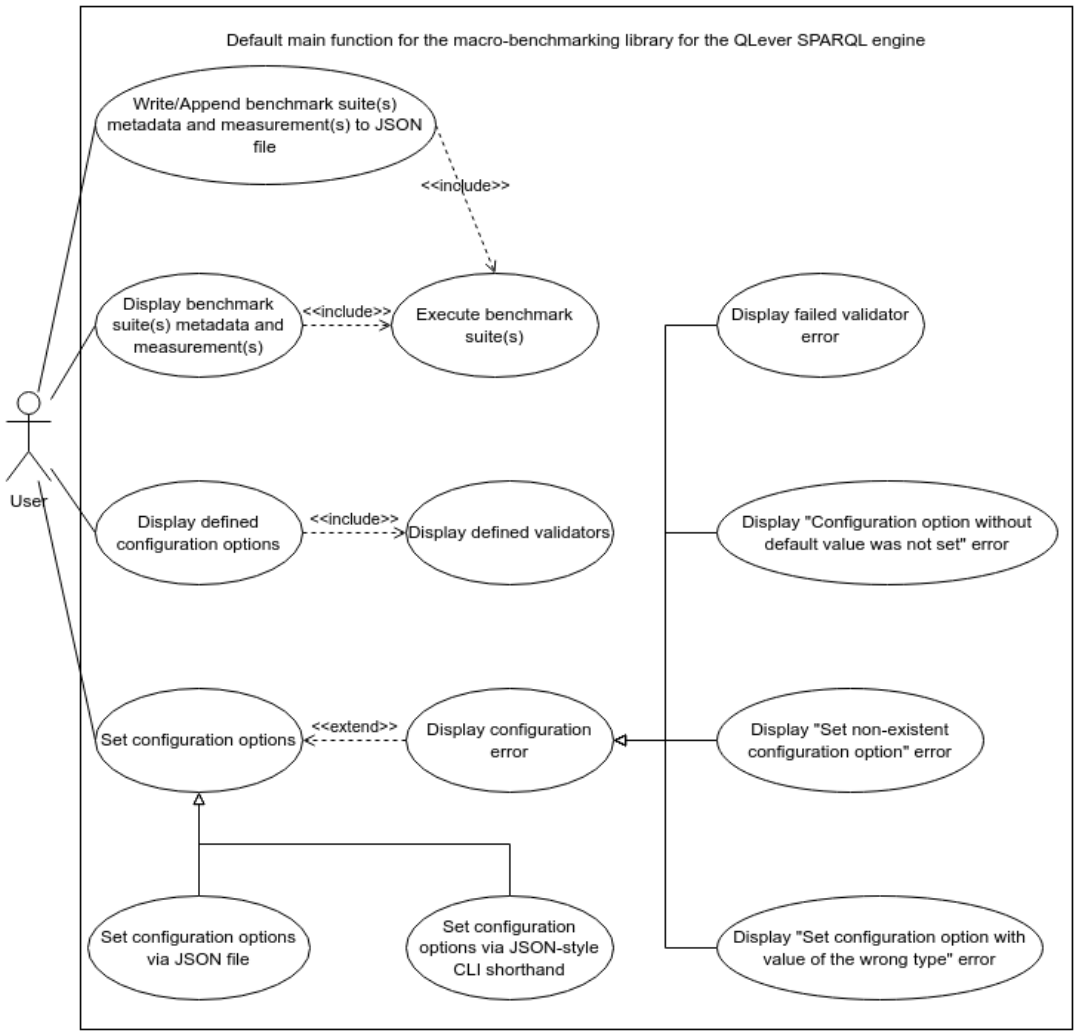

Figure 2.3: UML use case diagram overview for macro benchmark suite execution.

In this section, I will go through the functionality of the current default main function for my internal macro-benchmarking library for the QLever SPARQL engine.

Firstly, when macro benchmarks are executed, than every macro benchmark in a registered macro benchmark suite is executed once. Only executing a subset of all macro benchmarks in registered macro benchmark suites is not supported. Macro Benchmarks in not registered macro benchmark suites can never be executed.

The execution of macro benchmarks only happens if a user chose to write,

and optionally append, registered macro benchmark suites metadata and measurements to a "JSON" file, or if a user chose to display the registered macro benchmark suites metadata and measurements [4]. While executing the macro benchmarks, a message will be displayed before and after executing a macro benchmark. This message informs the user about the progress of the macro benchmark executions.

A user can choose to display the registered macro benchmark suites metadata and measurements. After executing all macro benchmarks in registered macro benchmark suites, the registered macro benchmark suites will be displayed in a list.

Every registered macro benchmark suit will be displayed as:

- Macro benchmark suite name.
- Macro benchmark suite metadata.
- What values the runtime configuration options of the registered macro benchmark suite were set to.
- A list of all the runtime configuration options of the registered macro benchmark suite and all the validator for runtime configuration options of the registered macro benchmark suite.

  Every runtime configuration option of the registered macro benchmark suite is displayed with:
  - The runtime configuration option name.
  - The runtime configuration option value type.
  - The value the runtime configuration option was set to.
  - The runtime configuration option description.
  - If a runtime configuration option default value was given, the runtime configuration option default value.

  If a runtime configuration option of the registered macro benchmark suite is contained inside a JSON object, the following additional information will be displayed:
  - The runtime configuration option is contained inside a JSON object.
  - The name of the JSON object containing the runtime configuration option.

  Every validator for runtime configuration options of the registered macro benchmark suite is displayed. A validator is displayed together the validator description.
- A list of the freestanding macro benchmarks in the registered macro bench-

mark suite. Every freestanding macro benchmark is displayed with the measured execution time in seconds and the freestanding macro benchmark metadata, if freestanding macro benchmark metadata was defined.

- A list of the tables in the registered macro benchmark suite. If the metadata of a table was defined, the table is displayed with the metadata of the table. The content of a table is displayed row for row and separated by line breaks. Above the displayed rows of a table, the tables column names are displayed.
- A list of the groups in the registered macro benchmark suite. For every group the group metadata is displayed, if the group metadata was defined. For every group all group members are displayed in a list. The macro benchmarks in a group are displayed the same way as the freestanding macro benchmarks. The tables in a group are displayed the same way as the entries of the list of tables in the registered macro benchmark suite.

A user can choose to write the registered macro benchmark suites metadata and measurements to a JSON file. If wanted, the registered macro benchmark suites metadata and measurements can be appended to the JSON file. If the user does not want the registered macro benchmark suites metadata and measurements to be appended to the JSON file, the content of the JSON file will be overwritten. The new content of the JSON file will contain JSON representations of all registered macro benchmark suites.

The JSON representation of a registered macro benchmark suite contains:

- The registered macro benchmark suite name
- The registered macro benchmark suite metadata.
- The JSON representations of all the freestanding macro benchmarks in the registered macro benchmark suite.
- The JSON representations of all the groups in the registered macro benchmark suite.
- The JSON representations of all the tables in the registered macro benchmark suite.

The JSON representation of a freestanding macro benchmark contains:

- The name of the freestanding macro benchmark.
- The metadata of the freestanding macro benchmark.
- The execution time measured by the freestanding macro benchmark.

The JSON representation of a table contains:

- The name of the table.
- The metadata of the table.

- The JSON representations of the content of the table.

The JSON representation of a group contains:

- The name of the group.
- The metadata of the group.
- The JSON representations of all the group members of the group.

A user can display all runtime configuration options of all registered macro benchmark suites and all the validator for runtime configuration options of all the registered macro benchmark suites. Displaying all runtime configuration options of all registered macro benchmark suites and all the validator for runtime configuration options of all the registered macro benchmark suites does not execute any macro benchmarks of the registered macro benchmark suites. The runtime configuration options of all registered macro benchmark suites and all the validator for runtime configuration options of all the registered macro benchmark suites are displayed in as a list.

A runtime configuration option is displayed with:

- The runtime configuration option name.
- The runtime configuration option value type.
- The value the runtime configuration option was set to.
- The runtime configuration option description.
- If a runtime configuration option default value was given, the runtime configuration option default value.

If a runtime configuration option is contained inside a JSON object, the following additional information will be displayed:

- The runtime configuration option is contained inside a JSON object.
- The name of the JSON object containing the runtime configuration option.

Every validator for runtime configuration options of the registered macro benchmark suite is displayed. A validator is displayed together the validator description.

As a reminder:

- Runtime configuration options are, in the context of my internal macro-benchmarking library for the QLever SPARQL engine, variables inside a compiled program, that can be set by a user at runtime from outside the compiled program.
- Runtime configuration options are organized within a recursive JSON object structure. A recursive JSON object structure is a JSON object, that can contain only key value pairs with the value a recursive JSON object structure

or a runtime configuration option.

A user can set the values of runtime configuration options at runtime either by passing a JSON file, or by passing a JSON-style shorthand string. Setting the values of runtime configuration options at runtime, does not execute any macro benchmarks of the registered macro benchmark suites.

When a user sets the values of runtime configuration options at runtime by passing a JSON file, the JSON file must contain a modified version of the recursive JSON object structure containing the runtime configuration options. I call this modified version of the recursive JSON object structure containing the runtime configuration options the runtime configuration.

The runtime configuration contains no runtime configuration options. Instead of runtime configuration options, the runtime configuration contains values for runtime configuration options. A runtime configuration option at the same position in the recursive JSON object structure as a value in the runtime configuration will be set to this value.

When a user sets the values of runtime configuration options at runtime by passing a JSON-style shorthand string, the JSON-style shorthand string describes a runtime configuration, see the previous paragraph, in a different syntax. The syntax used in the JSON-style shorthand string is slightly different from JSON. The syntax used in the JSON-style shorthand string was designed for use with the CLI. For more information about the syntax used in the JSON-style shorthand string see the relevant paragraph in chapter 3.

Setting the values of runtime configuration options at runtime, can fail. If syntax error are ignored, the reason for the failure when trying to set the values of runtime configuration options at runtime must be one of the following:

- A validator checked the validity of one or more runtime configuration options at runtime and the runtime configuration options failed the validity check. In this case, the failure message of the validator is displayed, and the names and values of all configuration options who were checked for validity by the validator are displayed.
- The user did not set the value of a configuration option that has no default value. The configuration option who was not set will be displayed. A configuration option without a default value must always be set. Remember, runtime configuration options are, in the context of my internal macro-benchmarking library for the QLever SPARQL engine, variables inside a compiled program, that can be set by a user at runtime from outside the

compiled program. A variable without a value can lead to hard to identify errors in algorithms.

- The user tried to set the value of a non-existent configuration option. Trying to set the value of a non-existent configuration option, tends to be the symptom of a bigger error or bigger misunderstanding.
- The user tried to set the value of a configuration option to a value with a type different than the defined type of the value of the configuration option. The configuration option whose value the user tried to set to a value with a type different than the defined type of the value of the configuration option will be displayed. Remember, runtime configuration options are, in the context of my internal macro-benchmarking library for the QLever SPARQL engine, variables inside a compiled program, that can be set by a user at runtime from outside the compiled program. A variable with a type different than the expected type can lead to hard to identify errors in algorithms.

Chapter 3

Implementation

In this chapter, I will give a broad overview about the implementation of my internal macro-benchmarking library for the QLever SPARQL engine.

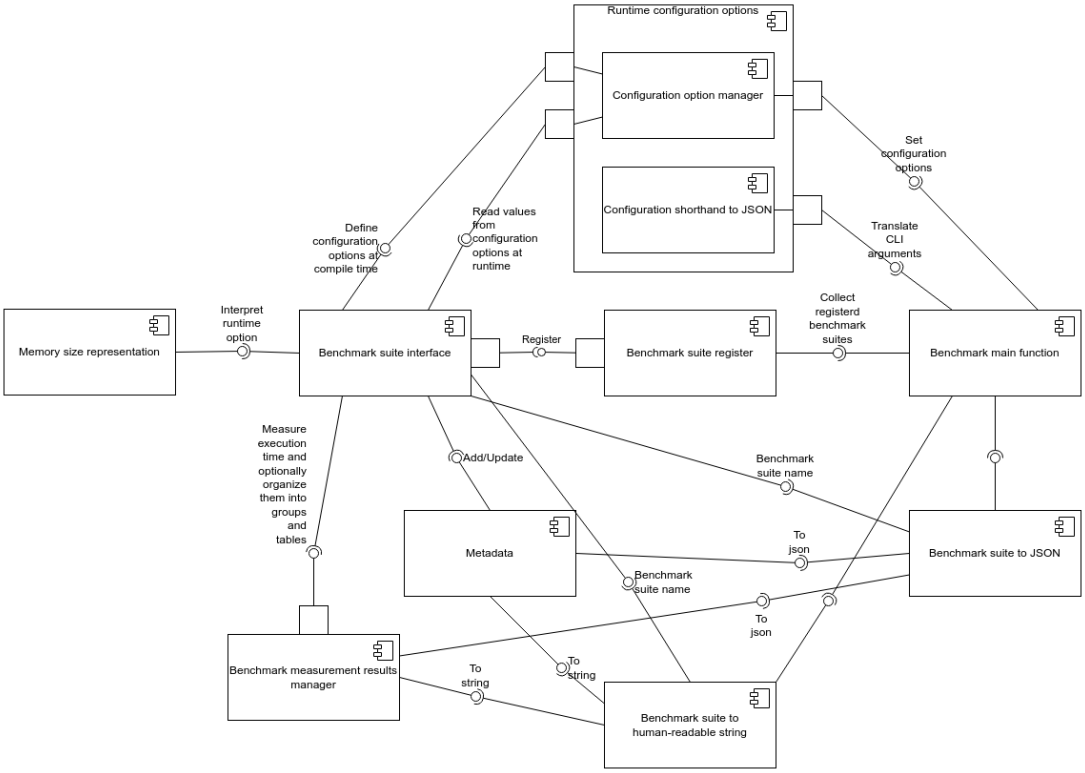

Figure 3.1: UML component diagram overview for my internal macro-benchmarking library for the QLever SPARQL engine.

In order to define a macro benchmark suite, a user must implement the benchmark suite interface. The benchmark suite interface holds a metadata object. Metadata The metadata object held by the macro benchmark suite normally describes general information. General information is information, that

is shared by all macro benchmarks in the macro benchmark suite. The metadata object held by the macro benchmark suite can be accessed via getter.

The metadata class itself wraps a third-party, so called on the project page of source [6], “Nlohmann-JSON”, “JSON” as defined by source [4], “JSON” object [6]. However, only key value pairs can be added to the third-party Nlohmann-JSON JSON object wrapped by the metadata class. Key value pairs added to the third-party Nlohmann-JSON JSON object wrapped by the metadata class can never be deleted, but the value in a key value pair can be updated.

Listing 3.1: Minimized example usage of the runtime configuration options of an example benchmark suite interface implementation. The code is paraphrased from the file “`BenchmarkExamples.cpp`” in source [1].

```
1  class ExampleBenchmarkSuiteImplementation : public ↩
       BenchmarkInterface {
2   protected:
3    // The values the runtime configuration options were set to will ↩
         be found here.
4    std::string dateString_;
5    int numberOfStreetSigns_;
6    std::vector<bool> wonOnTryX_;
7    float balanceOnStevesSavingAccount_;
8
9   public:
10   ConfigOptions() {
11     ad_utility::ConfigManager& manager = getConfigManager();
12
13     // Simple runtime configuration options with default values.
14     manager.addOption("date", "The current date.", &dateString_, ↩
         "22.3.2023"s);
15     manager.addOption("coin-flip-try", "The number of succesful ↩
         coin flips.",
16                       &wonOnTryX_, {false, false, false, false, false});
17     auto numSigns =
18         manager.addOption("num-signs", "The number of street signs.",
19                           &numberOfStreetSigns_, 10000);
20
21     // A simple validator that checks the value of a configuration ↩
         option.
```

```
22      manager.addValidator([](const int& num) { return num >= 0; },
23                           "The number of street signs must be at least ↩
                                 0!",
24                           "Negative numbers, or floating point ↩
                                 numbers, are not "
25                           "allowed for the configuration option ↩
                                 \"num-signs\".",
26                           numSigns);
27
28      // Sub manager can be used to organize runtime configuration ↩
            options better.
29      // They are basically just a path prefix for all the runtime ↩
            configuration options
30      // inside it.
31      // Note: I've called sub manager
32      // "JSON objects inside the recursive JSON object structure" ↩
            inside this thesis.
33      ad_utility::ConfigManager& subManager{
34          manager.addSubManager({"accounts"s, "personal"s})};
35      subManager.addOption("steve"s, "Steves saving account balance.",
36                           &balanceOnStevesSavingAccount_, -41.9f);
37    }
38  };
```

Often a user will want to limit memory usage of a macro benchmark suite. For example, when the macro benchmark suite is executed on different computer hardware with more or less available memory. I've created an explicit type for memory size representation to easier communicate such memory usage limits inside code. This explicit type for memory size representation is called "`MemorySize`" [1]. `MemorySize` supports interpreting strings, comparisons between values of `MemorySize`, conversion to string, and conversion to, and from, most memory size units like, for example, bytes, kilobytes, and megabytes. By using `MemorySize` in conjunction with runtime configuration options, a user can easily set different memory usage limits at runtime.

As a reminder, a user can create freestanding macro benchmarks, tables for the organization of macro benchmarks, and groups for the organization of macro benchmarks and tables. For more information see section 2.1.

A user creates freestanding macro benchmarks, tables for the organization of macro benchmarks, and groups for the organization of macro benchmarks and tables, using a macro benchmark measurement results manager. A macro benchmark measurement results manager manages instances of classes representing freestanding macro benchmarks, tables for the organization of macro benchmarks, and groups for the organization of macro benchmarks and tables. In the benchmark suite interface, a user creates a macro benchmark measurement results manager by implementing an abstract function that returns a macro benchmark measurement results manager.

Listing 3.2: Minimized example usage of benchmark measurement results manager inside a example benchmark suite interface implementation. The code is paraphrased from the file "`BenchmarkExamples.cpp`" in source [1].

```
1  class ExampleBenchmarkSuiteImplementation : public ↩
       BenchmarkInterface {
2   public:
3    BenchmarkResults runAllBenchmarks(){
4      BenchmarkResults results{};
5
6      /*
7      The functions for measuring the execution time of functions ↩
           only take
8      lambdas, which will be called without any arguments. Simply ↩
           wrap the actual
9      function call you want to measure.
10     */
11     auto& dummyFunctionToMeasure = [](){
12       // Do whatever you might want to measure.
13     };
14
15     // In order to recognise later, which time belongs to which ↩
           macro benchmark,
16     // most of the functions concerning the measurement of ↩
           functions, or their
17     // organization, require an identifier. A.k.a. a name.
18     const std::string identifier = "Some identifier";
19
20     // Create a freestanding macro benchmark.
```

```
21      results.addMeasurement(identifier, dummyFunctionToMeasure);
22
23      /*
24      Create an empty table with a number of rows and columns.
25      The number of columns can not be changed after creation, but ↩
            the number of rows can.
26      Important: The row names aren't saved in a seperate container, ↩
            but INSIDE the
27      first column of the table.
28      */
29      auto& table = results.addTable(identifier, {"rowName1", ↩
            "rowName2", "etc."},
30        {"Column for row names", "columnName1", "columnName2", ↩
              "etc."});
31
32      // You can add measurements to the table as entries, but you ↩
            can also
33      // read and set entries.
34      table.addMeasurement(0, 2, dummyFunctionToMeasure); // Row 0, ↩
            column 1.
35      table.setEntry(0, 1, "A custom entry");
36      table.getEntry(0, 2); // The measured time of the dummy function.
37
38      // Replacing a row name.
39      table.setEntry(0, 0, "rowName1++");
40
41      /*
42      Creates an empty group. Groups and tables can be added to ↩
            better organize
43      them.
44      */
45      auto& group = results.addGroup(identifier);
46      // Add a macro benchmark.
47      group.addMeasurement(identifier, dummyFunctionToMeasure);
48      // Add a table.
49      group.addTable(identifier, {"rowName1", "rowName2", "etc."},
50        {"Column for row names", "columnName1", "columnName2", ↩
              "etc."});
```

```
51
52      return results;
53    }
54  };
```

After defining a macro benchmark suite, and the content of the macro benchmark suite, a user must register the implemented benchmark suite interface of the defined macro benchmark suite.

An implemented benchmark suite interface is registered using the benchmark suite register. Every instance of the benchmark suite register shares the same internal array of registered, implemented benchmark suite interfaces. Creating a global instance of "`BenchmarkRegister`" and passing an instance of the implemented benchmark suite interface, will add the instance of the implemented benchmark suite interface to the internal array of registered, implemented benchmark suite interfaces [1]. To make the registering of implemented benchmark suite interfaces easier, a macro is available. This macro performs all steps needed to register the implemented benchmark suite interface.

Finally, the benchmark main function, a collection of functions, manages the registered macro benchmark suites. The benchmark main function uses the third-party library "Boost" to translate the CLI arguments generated by the user [7]. After translating the CLI arguments generated by the user, the main function uses various helper functions to execute the translated CLI arguments.

If a CLI argument to display all runtime configuration options of all registered macro benchmark suites and all the validator for runtime configuration options of all the registered macro benchmark suites was given, the benchmark main function will display all runtime configuration options of all registered macro benchmark suites and all the validator for runtime configuration options of all the registered macro benchmark suites. For the display of all runtime configuration options of all registered macro benchmark suites, the benchmark main function uses a helper function in the benchmark main function. For the display of all validators for runtime configuration options of all the registered macro benchmark suites, the benchmark main function uses a helper function in the configuration option manager.

As a reminder:

- Runtime configuration options are organized within a recursive "JSON" object structure [4]. A recursive JSON object structure is a JSON object,

that can contain only key value pairs with the value a recursive JSON object structure or a runtime configuration option.

- A user can set the values of runtime configuration options at runtime either by passing a JSON file, or by passing a JSON-style shorthand string.
- When a user sets the values of runtime configuration options at runtime by passing a JSON file, the JSON file must contain a modified version of the recursive JSON object structure for configuration options. I call this modified version of the recursive JSON object structure for configuration options the runtime configuration. The runtime configuration contains no configuration options. Instead of configuration options, the runtime configuration contains values for configuration options. A configuration option at the same position in the recursive JSON object structure as a value in the runtime configuration will be set to this value.
- When a user sets the values of runtime configuration options at runtime by passing a JSON-style shorthand string, the JSON-style shorthand string describes a runtime configuration, see the previous list item, in a syntax slightly different from JSON. The syntax used in the JSON-style shorthand string was designed for use with the CLI.

The syntax of a JSON-style shorthand string is the normal JSON syntax with four differences:

1. Only a runtime configuration can be described.
2. No line breaks.
3. The key of any key value pair is not surrounded by quotation marks. Furthermore, a key can only contain letters, numbers, the dash symbol, and the underscore symbol.
4. The outermost JSON object in the runtime configuration is not surrounded by curly brackets. All other JSON objects than the outermost JSON object are still surrounded by curly brackets.

If a CLI argument for setting the values of runtime configuration option was given, the benchmark main function, using a helper class, that translates the configuration shorthand to JSON, and the configuration option manager, sets the values of the runtime configuration options of all registered macro benchmark suites.

If a CLI argument to display the registered macro benchmark suites metadata and measurements was given, the benchmark main function will execute all macro benchmarks in registered macro benchmark suites and display the registered macro

benchmark suites, using multiple helper functions, in a list. The collection of multiple helper functions for displaying the registered macro benchmark suites is called ”Benchmark suite to human-readable string” in figure 3.1.

If a CLI argument to write, and optionally append, the registered macro benchmark suites metadata and measurements to a given JSON file was given, the benchmark main function will execute all benchmarks in registered macro benchmark suites, translate the registered macro benchmark suites, using multiple helper functions, to JSON, and write the JSON to the given JSON file. The collection of multiple helper functions for translating the registered macro benchmark suites to JSON is called ”Benchmark suite to JSON” in figure 3.1.

Chapter 4

Example Macro Benchmark Suite for Join Operation Algorithms

In this chapter, I first explain the join operator, see section 4.1, that is used inside QLever SPARQL engine. Then I explain the join operator algorithm currently used by the QLever SPARQL engine, see section 4.2.1, followed by an alternative join operator algorithm, see section 4.2.2.

In section 4.3, I explain the structure of my macro benchmark suites used for the comparison of the implementation of the two join operator algorithms explained in section 4.2. Section 4.4 lists the conditions under which my macro benchmark suites were executed and how they were executed.

Finally, section 4.5 evaluates the macro benchmark measurements generated by the executions, as described in section 4.4, of my macro benchmark suites.

4.1 Join Operation

In this section, I will explain the join operator.

The "`IdTable`" class is defined as, according the internal QLever SPARQL engine documentation, "The `IdTable` class is QLever's central data structure." [1]. A `IdTable` class instance can be thought of as a table. More information about the `IdTable` class is not needed for my thesis.

One of the supported operations for the `IdTable` class is the join operator. The join operator for `IdTable` class instances behaves the same as the join operator for normal tables with one exception. The join operator for `IdTable` class instances does not create a new `IdTable` class instance. Instead, the join operator for `IdTable` class instances adds rows to an existing `IdTable` class instance.

The join operator for normal tables requires two tables and a column index number for each of the two given tables. As of now, I will refer to the two normal tables required for the join operator as the first input table and second input table. Similarly, I will refer to the column in the first input table at the column index number position for the first input table as the first join column and the column in the second input table at the column index number position for the second input table as the second join column.

The join operator for normal tables creates a new normal table. As of now, I will refer to the table created by the join operator for normal tables as the result table. For every row in the Cartesian product of the first and second input table, where the value of the first join column is equal to the value of the second join column, the result table contains the Cartesian product row with the second join column removed.

A few example for the join operator:

Table 4.1: Example input table A

6	4	1
6	4	3
2	4	9
2	2	3
9	1	9

Table 4.2: Example input table B

9	1	9	3
5	3	5	3
8	4	9	3

Table 4.3: Join operation result table for table 4.1 with 2 as the column index number position for the first input table and table 4.2 with 2 as the column index number position for the second input table.

6	4	1	8	9	3
6	4	3	8	9	3
2	4	9	8	9	3
9	1	9	9	9	3

Table 4.4: Join operation result table for table 4.1 with 3 as the column index number position for the first input table and table 4.2 with 1 as the column index number position for the second input table.

2	4	9	1	9	3
9	1	9	1	9	3

4.2 Algorithms

The execution of the "`IdTable`" class join operator can be implemented with multiple different algorithms [1]. In this section, I will broadly explain the current algorithm used by the QLever SPARQL engine, see section 4.2.1, for the execution of the `IdTable` class join operator, followed by a broad explanation for an alternate algorithm, see section 4.2.2, for the execution of the `IdTable` class join operator.

First, three definitions shared by the explanations in the section 4.2.1 and section 4.2.2:

1. The first of the two `IdTable` instances required by the `IdTable` class join operator is referred to as the left input table. The second of the two `IdTable` instances required by the `IdTable` class join operator is referred to as the right input table. The column in the left input table at the column index number position for the left input table is referred to as the left join column. Vice versa for the right input table.
2. Of the two `IdTable` instances required by the `IdTable` class join operator, the `IdTable` instance with the biggest number of rows is referred to as the bigger input table. The `IdTable` instance with the smallest number of rows is referred to as the smaller input table. If the two `IdTable` instances required by the `IdTable` class join operator have the same number of rows, then one of the two `IdTable` instances required by the `IdTable` class join operator will be randomly designated as the bigger input table and the remaining `IdTable` instance will the smaller table. Furthermore, the column in the smaller input table at the column index number position for the smaller input table is referred to as the smaller join column. Vice versa for the bigger input table.
3. The `IdTable` instance the `IdTable` class join operator adds row to is referred to as the result table.

4.2.1 Merge and Galloping Join

In this section, I will broadly explain the currently used QLever SPARQL engine algorithm for the "`IdTable`" class join operator execution [1], that can be found inside the function "`join`" inside the file "`Join.cpp`" inside the QLever SPARQL engine [1].

Currently, the QLever SPARQL engine algorithm for the `IdTable` class join operator execution chooses between two algorithms for the `IdTable` class join operator execution.

Before the QLever SPARQL engine algorithm for the `IdTable` class join operator execution chooses between two algorithms for the `IdTable` class join operator execution, if the left or right input table has zero rows the QLever SPARQL engine algorithm for the `IdTable` class join operator execution is done.

The two algorithms QLever SPARQL engine chooses between are the merge join algorithm and the galloping join algorithm. Both merge join algorithm and the galloping join algorithm require the left input table to be sorted by the content of the left join column and the right input table to be sorted by the content of the right join column. Both merge join algorithm and the galloping join algorithm add rows sorted by the left join column to the result table.

When the number of rows in the bigger input table divided by the number of rows in the smaller input table is less than or equal to 1000, the merge join algorithm is chosen.

I will now explain a simplified version of the merge join algorithm. For a more detailed explanation of the merge join algorithm see the pseudocode in listing 4.1.

The following variables are passed to the simplified version of the merge join algorithm:

- The left input table.
- The left join column index number.
- The right input table.
- The right join column index number.
- The result table.

The simplified version of the merge join algorithm walks through the left join column and the right join column in parallel. I will refer to the position of the simplified version of the merge join in the left joint column as the current left join column entry and to the position of the simplified version of the merge join in the right joint column as the current right join column entry. The current left join

column entry starts with the join column in the first row of the left input table. The current right join column entry starts with the join column in the first row of the right input table.

The walk of the simplified version of the merge join algorithm through the left join column and the right join column in parallel behaves as follows:

- If the content of the current left join column entry is smaller than the content of the current right join column entry, the simplified version of the merge join algorithm takes one step in the left join column.
- If the content of the current left join column entry is bigger than the content of the current right join column entry, the simplified version of the merge join algorithm takes one step in the right join column.
- If the content of the current left join column entry is equal to the content of the current right join column entry, the simplified version of the merge join algorithm stops the walk through the left join column and the right join column. The simplified version of the merge join algorithm finds the range in the left join column with the same content as the current left join column entry and the range in the right join column with the same content as the current right join column entry. I will refer to the range in the left join column with the same content as the current left join column entry as the left equality range and to the range in the right join column with the same content as the current right join column entry as the right equality range. The simplified version of the merge join algorithm then creates the new rows for the result table from the Cross product of the left equality and right equality range. After creating the new rows for the result table the simplified version of the merge join algorithm takes steps in the left join column until outside the left equality range, steps in the right join column until outside the right equality range, and starts the walk through the left join column and the right join column again.

Listing 4.1: Pseudocode explanation for the merge join algorithm. Based on the code of function "`zipperJoinWithUndef`", that implements the merge join algorithm, in QLever SPARQL engine [1].

```
1 def mergeJoin(leftInputTable, leftJoinColumnIndex, ↩
      rightInputTable, rightJoinColumnIndex, resultTable):
2     # First row of both input tables.
3     leftRowIdx, rightRowIdx = 0
```

```
4       while leftRowIdx < leftInputTable.numRows() and rightRowIdx < ↩
            rightInputTable.numRows():
5           # Search through both input tables until rows with equal ↩
                join column content can be found.
6           while leftInputTable[leftRowIdx][leftJoinColumnIndex] < ↩
                rightInputTable[rightRowIdx][rightJoinColumnIndex]:
7               leftRowIdx += 1
8               if leftRowIdx >= leftInputTable.numRows():
9                   return
10          while leftInputTable[leftRowIdx][leftJoinColumnIndex] > ↩
                rightInputTable[rightRowIdx][rightJoinColumnIndex]:
11              rightRowIdx += 1
12              if rightRowIdx >= rightInputTable.numRows():
13                  return
14
15          # Find the range in the input table rows where the ↩
                respective join column content is equal.
16          endSameLeftRowIdx = leftRowIdx
17          while endSameLeftRowIdx < leftInputTable.numRows() and ↩
                leftInputTable[leftRowIdx][leftJoinColumnIndex] == ↩
                leftInputTable[endSameLeftRowIdx][leftJoinColumnIndex]:
18              endSameLeftRowIdx += 1
19          endSameRightRowIdx = rightRowIdx
20          while endSameRightRowIdx < rightInputTable.numRows() and ↩
                rightInputTable[RightRowIdx][rightJoinColumnIndex] == ↩
                rightInputTable[endSameRightRowIdx][rightJoinColumnIndex]:
21              endSameRightRowIdx += 1
22
23          # Append new rows the result table.
24          for leftRowIdx in range(leftRowIdx, endSameLeftIdx):
25              for innerRightRowIdx in range(rightRowIdx, ↩
                    endSameRightIdx):
26                  # Append the row of right input table without the ↩
                        right join column to the row of the left input ↩
                        table.
27                  resultTable.append(combineRows(leftInputTable[leftRowIdx], ↩
                        rightInputTable[innerRightRowIdx], ↩
                        rightJoinColumnIndex))
```

```
28          leftRowIdx = endSameLeftRowIdx
29          rightRowIdx = endSameRightRowIdx
```

When the number of rows in the bigger input table divided by the number of rows in the smaller input table is bigger than 1000, the galloping join algorithm is chosen.

I will now explain a simplified version of the galloping join algorithm. For a more detailed explanation of the galloping join algorithm see the pseudocode in listing 4.2.

The following variables are passed to the simplified version of the galloping join algorithm:

- The smaller input table
- The smaller join column index number.
- The bigger input table.
- The bigger join column index number.
- Which of the smaller and bigger input table is the left table.
- The result table.

The simplified version of the galloping join algorithm walks through the smaller join column and the bigger join column in parallel. I will refer to the position of the simplified version of the galloping join in the smaller joint column as the current smaller join column entry and to the position of the simplified version of the galloping join in the bigger joint column as the current bigger join column entry. The current smaller join column entry starts with the join column in the first row of the smaller input table. The current bigger join column entry starts with the join column in the first row of the bigger input table.

The walk of the simplified version of the galloping join algorithm through the smaller join column and the bigger join column in parallel behaves as follows:

1. If the content of the current smaller join column entry is smaller than the content of the current bigger join column entry, the simplified version of the galloping join algorithm takes one step in the smaller join column.
2. If the content of the current smaller join column entry is not smaller than the content of the current bigger join column entry, the simplified version of the galloping join algorithm stops the walk through the left join column and the right join column. The simplified version of the galloping join algorithm searches for the first occurrence of the content of the current smaller join column entry in the bigger join column, starting at the current bigger join

column entry, via binary search. The current bigger join column entry is set to the first occurrence of the content of the current smaller join column entry in the bigger join column. The simplified version of the galloping join algorithm finds the range in the left join column with the same content as the current left join column entry and the range in the right join column with the same content as the current right join column entry. I will refer to the range in the left join column with the same content as the current left join column entry as the left equality range and to the range in the right join column with the same content as the current right join column entry as the right equality range. The simplified version of the galloping join algorithm then creates the new rows for the result table from the Cross product of the left equality and right equality range. After creating the new rows for the result table the simplified version of the galloping join algorithm takes steps in the left join column until outside the left equality range, steps in the right join column until outside the right equality range, and starts the walk through the left join column and the right join column again.

Listing 4.2: Pseudocode explanation for the galloping join algorithm. Based on the code of function "`gallopingJoin`", that implements the galloping join algorithm, in QLever SPARQL engine [1].

```
1  # Approximate the lower and upper bound of the range in `column` ↩
       in which the first instance of `searchedValue` can be found ↩
       using a variant of the exponential search.
2  def exponentialSearch(column, startIdx, searchedValue):
3      step = 1
4      lowerIdx, entryIdx = startIdx
5      while column[entryIdx]< searchedValue:
6          lowerIdx = entryIdx
7          entryIdx += step
8          step *= 2
9          if (entryIdx >= column.size()):
10             return lowerIdx, column.size()
11     # It might be, that `entryIdx` points to the first element ↩
           that is >= searchedValue. I have to add one to the second ↩
           element, s.t. the second element is a guaranteed upper bound.
12     return lowerIdx, entryIdx + 1
13
```

```
14
15 def gallopingJoin(smallerInputTable, smallerJoinColumnIndex, biggerInputTable, biggerJoinColumnIndex, resultTable, leftInputTableIsBigger):
16     # First row of both input tables.
17     smallerRowIdx, biggerRowIdx = 0
18     while smallerRowIdx < smallerInputTable.numRows() and biggerRowIdx < biggerInputTable.numRows():
19         while smallerInputTable[smallerRowIdx][smallerJoinColumnIndex] < biggerInputTable[biggerRowIdx][biggerJoinColumnIndex]:
20             smallerRowIdx += 1
21             if smallerRowIdx >= smallerInputTable.numRows():
22                 return
23
24         # Use a variant of the exponential search to approximate the lower and upper bound of the range in the bigger join column in which the first instance of `smallerInputTable[smallerRowIdx][smallerJoinColumnIndex]` can be found.
25         lowerIdx, upperIdx = exponentialSearch(biggerInputTable.column(biggerJoinColumnIndex), biggerRowIdx, smallerInputTable[smallerRowIdx][smallerJoinColumnIndex])
26
27         # Binary search for the first occurrence of `smallerInputTable[smallerRowIdx][smallerJoinColumnIndex]` in the range in the bigger join column in which the first instance of `smallerInputTable[smallerRowIdx][smallerJoinColumnIndex]` can be found.
28         biggerRowIdx = findFirstOccurenceWithBinarySearch(lowerIdx, upperIdx, biggerInputTable.column(biggerJoinColumnIndex), smallerInputTable[smallerRowIdx][smallerJoinColumnIndex])
29         if (biggerRowIdx >= biggerInputTable.numRows()) {
30             return
31         }
```

```
32
33          # Find the range in the input table rows where the ↩
                respective join column content is equal.
34          endSameSmallerRowIdx = smallerRowIdx
35          while endSameSmallerRowIdx < smallerInputTable.numRows() ↩
                and ↩
                smallerInputTable[smallerRowIdx][smallerJoinColumnIndex] ↩
                == ↩
                smallerInputTable[endSameSmallerRowIdx][smallerJoinColumnIndex]:
36              endSameSmallerRowIdx += 1
37          endSameBiggerRowIdx = biggerRowIdx
38          while endSameBiggerRowIdx < biggerInputTable.numRows() and ↩
                biggerInputTable[biggerRowIdx][biggerJoinColumnIndex] ↩
                == ↩
                biggerInputTable[endSameBiggerRowIdx][biggerJoinColumnIndex]:
39              endSameBiggerRowIdx += 1
40
41          # Append new rows the result table.
42          for smallerRowIdx in range(smallerRowIdx, endSamesmallerIdx):
43              for innerbiggerRowIdx in range(biggerRowIdx, ↩
                    endSamebiggerIdx):
44                  # Append the row of right input table without the ↩
                        right join column to the row of the left input ↩
                        table.
45                  resultTable.append(combineRows(smallerInputTable[smallerRowIdx], ↩
                        smallerJoinColumnIndex, ↩
                        biggerInputTable[innerbiggerRowIdx], ↩
                        biggerJoinColumnIndex, leftInputTableIsBigger))
46          smallerRowIdx = endSameSmallerRowIdx
47          biggerRowIdx = endSameBiggerRowIdx
```

4.2.2 Hash Join

In this section, I will broadly explain an alternative algorithm for the "`IdTable`" class join operator execution [1], that can be found inside the function "`hashJoinImpl`" inside the file "`Join.cpp`" inside the QLever SPARQL engine [1].

As an alternative for the merge join algorithm and galloping join algorithm,

currently used by the QLever SPARQL engine, see section 4.2.1, I have implemented the so-called hash join algorithm in the QLever SPARQL engine. Currently, the hash join algorithm is not yet integrated in the program flow of the QLever SPARQL engine.

Unlike the merge join algorithm and galloping join algorithm, currently used by the QLever SPARQL engine, see section 4.2.1, the hash join algorithm does not require the left input table to be sorted by the content of the left join column and the right input table to be sorted by the content of the right join column. However, the rows added to the result table by the hash join algorithm are also not sorted by the left or right join column.

I will now explain a simplified version of the hash join algorithm. For a more detailed explanation of the hash join algorithm see the pseudocode in listing 4.3.

The following variables are passed to the simplified version of the hash join algorithm:

- The smaller input table
- The smaller join column index number.
- The bigger input table.
- The bigger join column index number.
- Which of the smaller and bigger input table is the left table.
- The result table.

First, the simplified version of the hash join algorithm appends every row of the smaller input table to a list in a hash map with the content of the smaller join column for that row of the smaller input table as the key of the hash map. I will refer to the hash map containing lists of all smaller input tables rows as the smaller input table hash map.

After inserting every row of the smaller input table into the smaller input table hash map, the simplified version of the hash join algorithm walks through the bigger input table rows. I will refer to the current row of the bigger input table as the current row. If the content of the bigger join column in the current row is not a key for an entry in the smaller input table hash map, the current row is skipped. If the content of the bigger join column in the current row is a key for an entry in the smaller input table hash map, the simplified version of the hash join algorithm creates the new rows for the result table from the Cross product of the current row and the list of rows in the smaller input table hash map entry for whom the content of the bigger join column in the current row is the key.

Listing 4.3: Pseudocode explanation for the hash join algorithm. Based on the code of function "`hashJoinImpl`", that implements the galloping join algorithm, in QLever SPARQL engine [1].

```
1  def hashJoin(smallerInputTable, smallerJoinColumnIndex, ↩
       biggerInputTable, biggerJoinColumnIndex, resultTable, ↩
       leftInputTableIsBigger):
2      if smallerInputTable.numRows() == 0 or ↩
           biggerInputTable.numRows() == 0:
3          return
4
5      # Put the rows of the smaller input table in a hash map. The ↩
           keys of hash map entries are the smaller join column entries.
6      Hashmap hashMap
7      for row in smallerInputTable:
8          hashMap[row[smallerJoinColumnIndex]].append(row)
9
10     # Walk through the bigger input table and append new rows to ↩
           the result table.
11     for biggerInputTableRow in biggerInputTable:
12         # Skip, if there are no rows in the smaller input table for ↩
               the content of the bigger join column entry.
13         if ↩
               hashMap[biggerInputTableRow[biggerJoinColumnIndex]].size() ↩
               == 0:
14             continue;
15
16         # Append new rows the result table.
17         for smallerInputTableRow in ↩
               hashMap[biggerInputTableRow[biggerJoinColumnIndex]]:
18             # Append the row of right input table without the right ↩
                   join column to the row of the left input table.
19             resultTable.append(combineRows(smallerInputTableRow, ↩
                   smallerJoinColumnIndex, biggerInputTableRow, ↩
                   biggerJoinColumnIndex, leftInputTableIsBigger))
```

4.3 Join Algorithm Macro Benchmark Suites Structure

In this section, I will explain the structure of my macro benchmark suites, that were used to compare the execution time of the implementation for the merge and galloping join algorithm, see section 4.2.1, with the execution time of the implementation for the hash join algorithm, see section 4.2.2, in different situation and under different workloads. As of now, I will refer to my macro benchmark suites that were used to compare the execution time of the implementation for the merge and galloping join algorithm with the execution time of the implementation for the hash join algorithm as my join benchmark suites.

First, a few general reminders:

- Benchmark: Source [2] defines a "benchmark" as, "a computer program that measures the ... speed of computer software ... ". In this thesis, a benchmark measures the execution time of computer software.
- Benchmark suite: Set of benchmarks.

Of the two "`IdTable`" instances required by the algorithms in section 4.2 the `IdTable` instance with the biggest number of rows is referred to as the bigger input table [1]. The `IdTable` instance with the smallest number of rows is referred to as the smaller input table. The column in the smaller input table at the column index number position for the smaller input table is referred to as the smaller join column. Vice versa for the bigger input table.

Inside my join benchmark suites, I always pass the smaller input table as the first parameter to the implementations of the algorithms in section 4.2 and the bigger input table as the second parameter to the implementations of the algorithms in section 4.2. I do this to cover every possible, generalized situation the implementations of the algorithms in section 4.2 may encounter efficiently. Generalized situations in which the bigger input table is passed as the first parameter to the implementations of the algorithms in section 4.2 can be ignored, because the execution time does not change noticeable compared to generalized situation in which the smaller input table is passed as the first parameter to the implementations of the algorithms in section 4.2 and the bigger input table is passed as the second parameter to the implementations of the algorithms in section 4.2,

The content of the smaller and bigger input table entries are randomly

generated natural numbers. Covering every possible, generalized situation for the content of the smaller and bigger input table entries inside my join benchmark suite is not possible. Furthermore, only the amount and distribution of unique content in entries inside the smaller and bigger join column affect the execution time of the algorithms in section 4.2, not the content of an entry in the smaller or bigger join column itself. The number of rows in the smaller input table and the number of columns in the smaller input table are not set to a randomly generated natural number. Vice versa for the bigger input table.

Every macro benchmark suite in my join benchmark suites organizes its macro benchmarks with tables. For a reminder about tables see section 2.1. I will refer to a table for the organization of macro benchmarks in my join benchmark suites as macro benchmark tables. Every macro benchmark table in my join benchmark suites follows the same structure. I will refer to the structure followed by every macro benchmark table in my join benchmark suites as the join benchmark table structure.

Every row in a macro benchmark table following the join benchmark table structure has its own smaller input table and bigger input table. The content of the smaller and bigger input table of a row in a macro benchmark table following the join benchmark table structure is randomly generated at the creation of that row.

In a row in a macro benchmark table following the join benchmark table structure there are seven columns:

1. A variable for the generation of the smaller and bigger input table that is changing with every row. For example, the amount of rows in the smaller input table.
2. The macro benchmark measured execution time of sorting the smaller input table by the content of the smaller join column and the bigger input table by the content of the bigger join column with the internal sorting function in the QLever SPARQL engine. The sorting of the smaller input table by the content of the smaller join column and the bigger input table by the content of the bigger join column is always done.
3. The macro benchmark measured execution time of the implementation of the merge and galloping join algorithm, see section 4.2.1.
4. The sum of two previous list entries.
5. The macro benchmark measured execution time of the implementation of the hash join algorithm, see section 4.2.2.

6. Number of rows in the result of the join operation with the smaller and bigger input table.
7. List entry 4 divided by list entry 5. The value in this column is called the hash join speedup.

   The hash join algorithm does not require the smaller input table to be sorted by the content of the smaller join column and the bigger input table to be sorted by the content of the bigger join column. Because the merge and galloping join algorithm requires the smaller input table to be sorted by the content of the smaller join column and the bigger input table to be sorted by the content of the bigger join column, the execution time of sorting the smaller input table by the content of the smaller join column and the bigger input table by the content of the bigger join column should be added to the execution time of the merge and galloping join algorithm. Otherwise, any comparison of the execution time of the merge and galloping join algorithm with the execution time of the hash join algorithm would be missing important information.

   Therefore, the hash join speedup truly shows how much faster the macro benchmark measured execution time of the implementation of the hash join algorithm is than the macro benchmark measured execution time of the implementation of the merge and galloping join algorithm.

The amount of rows in a macro benchmark table following my join benchmark table structure is dependent on which macro benchmark suite of the join benchmark suites the macro benchmark table following the join benchmark table structure is contained in.

Before I list the macro benchmark suites in my join benchmark suites, I will define the value named row ratio. The row ratio is defined as the number of rows in the bigger input table divided by the number of rows in the smaller input table. In the join benchmark suites, the number of rows in the bigger input table are set to the result of multiplying the number of rows in the smaller input table with the row ratio. I decided to set number of rows in the bigger input table to the result of multiplying the number of rows in the smaller input table with the row ratio, because the evaluation in section 4.5 focuses on the row ratio.

There are five macro benchmark suites in the join benchmark suites:

1. The macro benchmark suite, name quoted from the file containing my join benchmark suites in [1], “Benchmarktables, where the tables are the same size and both just get more rows.” In this macro benchmark suite, the

smaller and bigger input table always have the same amount of rows. In the macro benchmark tables of this macro benchmark suite, the amount of rows in the smaller and bigger input table grows with every macro benchmark row.

2. The macro benchmark suite, name quoted from the file containing my join benchmark suites in [1], "Benchmarktables, where the smaller table grows and the ratio between tables stays the same." Every macro benchmark table in this macro benchmark suite has its own row ratio assigned. In the macro benchmark tables of this macro benchmark suite, the amount of rows in the smaller input table grows with every macro benchmark row.
3. The macro benchmark suite, name quoted from the file containing my join benchmark suites in [1], "Benchmarktables, where the smaller table stays at the same amount of rows and the bigger tables keeps getting bigger." Every macro benchmark table in this macro benchmark suite has its own number of rows for the smaller input table assigned. In the macro benchmark tables of this macro benchmark suite, the row ratio grows with every macro benchmark row.
4. The macro benchmark suite, name quoted from the file containing my join benchmark suites in [1], "Benchmarktables, where only the sample size ratio changes." As a reminder, inside my join benchmark suites, the content of the smaller and bigger join column entries are randomly generated natural numbers. As said before, only the amount and distribution of unique content in entries inside the smaller and bigger join column affect the execution time of the algorithms in section 4.2, not the content of an entry in the smaller or bigger join column itself.

   The random natural number generation for the content of the smaller join column entries generates the content for an entry in the smaller join column by choosing a random natural number from a set of natural numbers with a discrete uniform distribution. I will refer to set of natural numbers used in the random natural number generation for the content of the smaller join column entries as the smaller random number set. Normally, the size of the smaller random number set is equal to the amount of rows in the smaller input table. The random natural number generation for the content of the bigger join column entries generates the content for an entry in the bigger join column by choosing a random natural number from a set of natural numbers with a discrete uniform distribution. I will refer to set of natural

numbers used in the random natural number generation for the content of the bigger join column entries as the bigger random number set. Normally, the size of the bigger random number set is equal to the amount of rows in the bigger input table.

The smaller and bigger random number set do not share any elements. Instead, after the random natural number generation for the content of the smaller and bigger join column entries, the content of some randomly choose smaller join column entries are overwritten by the content of randomly choose bigger join column entries.

A "sample size ratio", quoted from [1], adjusts the size of the smaller or bigger random number sets. A sample size ratio is a positive floating point number. There are two sample size ratios in my join benchmark suites. One sample size ratio for the smaller random number set and one sample size ratio for the bigger random number set.

The size of the smaller random number set is set to the number of rows in the smaller input table multiplied by the sample size ratio for the smaller number set. The size of the bigger random number set is set to the number of rows in the bigger input table multiplied by the sample size ratio for the bigger number set.

Normally, the sample size ratio for the smaller random number set is used for every smaller input table in the join benchmark suites. Vice versa for the sample size ratio for the bigger random number set. However, every macro benchmark table in this macro benchmark suite has its own sample size ratio for the smaller random number set assigned. In the macro benchmark tables of this macro benchmark suite, the sample size ratio for the bigger random number set grows with every macro benchmark row.

5. The macro benchmark suite, name quoted from the file containing my join benchmark suites in [1], "Benchmarktables, where the tables are the same size and both just get more rows." Every macro benchmark table in this macro benchmark suite has its own number of rows for the bigger input table assigned. In the macro benchmark tables of this macro benchmark suite, the amount of rows in the smaller input table grows with every macro benchmark row.

Quick reminder, every macro benchmark table in my join benchmark suites follows the same structure. I refer to the structure followed by every macro benchmark table in my join benchmark suites as the join benchmark table structure.

Every row in a macro benchmark table following the join benchmark table structure has its own smaller input table and bigger input table. Every row in a macro benchmark table following the join benchmark table structure has seven columns. One of the seven columns is the macro benchmark measured execution time for sorting the smaller input table by the content of the smaller join column and the bigger input table by the content of the bigger join column.

Every macro benchmark table in the five macro benchmark suites inside the join benchmark suites is repeated four times:

- At the point in time a macro benchmark measures the execution time of sorting the smaller input table by the content of the smaller join column and sorting the bigger input table by the content of the bigger join column, the smaller and bigger input table are already sorted. The smaller and bigger input table are sorted with the internal sorting function in QLever SPARQL engine.
- At the point in time a macro benchmark measures the execution time of sorting the smaller input table by the content of the smaller join column and sorting the bigger input table by the content of the bigger join column, the smaller input table is already sorted and the bigger input table is unsorted. The smaller and bigger input table are sorted with the internal sorting function in QLever SPARQL engine.
- At the point in time a macro benchmark measures the execution time of sorting the smaller input table by the content of the smaller join column and sorting the bigger input table by the content of the bigger join column, the smaller input table is unsorted and the bigger input table is already sorted. The smaller and bigger input table are sorted with the internal sorting function in QLever SPARQL engine.
- At the point in time a macro benchmark measures the execution time of sorting the smaller input table by the content of the smaller join column and sorting the bigger input table by the content of the bigger join column, the smaller and bigger input table are unsorted. The smaller and bigger input table are sorted with the internal sorting function in QLever SPARQL engine.

In order to make the join benchmark suites more flexible after compilation, I've defined multiple runtime configuration options. As a reminder, runtime configuration options are, in the context of my internal macro-benchmarking library for the QLever SPARQL engine, variables inside a compiled program, that

can be set by a user at runtime from outside the compiled program.

The defined runtime configuration options for the join benchmark suites are:

- As said in source [1] the amount of columns for the smaller input table in benchmark tables. Has a default value of 20.
- As said in source [1] the amount of columns for the bigger input table in benchmark tables. Has a default value of 20.
- As said in source [1] the minimum amount of rows for the smaller input table in benchmark tables. Has a default value of 10 000.
- As said in source [1] the minimum amount of rows for the bigger input table in benchmark tables. Has a default value of 100 000.
- As said in source [1] the maximum amount of rows for the bigger input table in benchmark tables. Has a default value of 10 000 000.
- The chance for the content of a randomly choose smaller join column entries to be overwritten by the content of a randomly choose bigger join column entry. Has a default value of 50%. For more context, see the previous explanation for the macro benchmark suite, name quoted from the file containing my join benchmark suites in [1], "Benchmarktables, where only the sample size ratio changes."
- The sample size ratio for the smaller random number set. Has a default value of one. For more context, see the previous explanation for the macro benchmark suite, name quoted from the file containing my join benchmark suites in [1], "Benchmarktables, where only the sample size ratio changes."
- The sample size ratio for the bigger random number set. Has a default value of one. For more context, see the previous explanation for the macro benchmark suite, name quoted from the file containing my join benchmark suites in [1], "Benchmarktables, where only the sample size ratio changes."
- The seed for the generation of seeds for the random natural number generators. If the user does not set the seed for the generation of seeds for the random natural number generators at runtime, the seed for the generation of seeds is set to a random non-deterministic value. For more context, see the previous explanation for the macro benchmark suite, name quoted from the file containing my join benchmark suites in [1], "Benchmarktables, where only the sample size ratio changes."
- As said in source [1] the minimum row ratio between the smaller and the bigger input table in the macro benchmark suites "Benchmarktables, where the smaller table grows and the ratio between tables stays the same." and

"Benchmarktables, where the smaller table stays at the same amount of rows and the bigger tables keeps getting bigger." Has a default value of one.

- As said in source [1] the maximum row ratio between the smaller and the bigger input table in the macro benchmark suite, name quoted from the file containing my join benchmark suites in [1], "Benchmarktables, where the smaller table grows and the ratio between tables stays the same." Has a default value of 1 000.
- As said in source [1] the max amount of memory that any "`IdTable`" instance takes up. If the user sets the max amount of memory at runtime, the configuration option for the maximum amount of rows for the bigger input table is ignored. Otherwise, the amount of memory that any `IdTable` instance can take up is not limited.
- As said in source [1] the max amount of time, in seconds, that any macro benchmark execution time measurement can take up. If the user does not set the max amount of time at runtime, the amount of time, in seconds, that any macro benchmark execution time measurement can take up is not limited.
- The sample size ratios used in the macro benchmark suite, name quoted from the file containing my join benchmark suites in [1], "Benchmarktables, where only the sample size ratio changes." The sample size ratios from this runtime configuration option are not used anywhere else. The default value for this runtime configuration option is a list of positive floating point numbers. Has a default value of 0.1, 1 and 10.

4.4 Sample Specifications

I refer to my macro benchmark suites that were used to compare the execution time of the implementation for the merge and galloping join algorithm, see section 4.2.1, with the execution time of the implementation for the hash join algorithm, see section 4.2.2, as my join benchmark suites. In this section, I will the list the hardware, see section 4.4.1, I executed my join benchmark suites on and the values, see section 4.4.2, used for the join benchmark suites runtime configuration options.

4.4.1 Hardware

I executed my join benchmark suites on the following hardware:

- AMD Ryzen 7 3700X 8-Core Processor
- RAM 130 GiB

4.4.2 Values for Runtime Configuration Options

As explained in section 4.5, I executed my join benchmark suites multiple times. For every execution of my join benchmark suit I only set two of the available runtime configuration options for my join benchmark suites at runtime.

The first of two runtime configuration options I set is the max amount of memory that any "`IdTable`" instance takes up [1]. I set the max amount of memory that any `IdTable` instance takes up to 20 GB.

The second of two runtime configuration options I set is the seed for the generation of seeds for the random natural number generators. I will refer to the seed for the generation of seeds for the random natural number generators simply as the seed. However, I do not set the seed to the same value for every execution of my join benchmark suit. See table 4.5 for what value the seed was set to for what execution of my join benchmark suites.

Table 4.5: Values for the seed for the generation of seeds for the random natural number generators.

Join benchmark suit execution	Value for the seed
1 to 20	146081905
21 to 40	193901340
41 to 60	288613237
61 to 80	155923003
81 to 100	4648133

The remaining defined runtime configuration options for the join benchmark suites, see the list in section 4.3, are not set and use their default values if they have a default value.

4.5 Evaluating Join Algorithm Macro Benchmark Suite Results

I refer to my macro benchmark suites that were used to compare the execution time of the implementation for the merge and galloping join algorithm, see section 4.2.1, with the execution time of the implementation for the hash join algorithm, see section 4.2.2, as my join benchmark suites. Note, that a macro benchmark suite can never be used to compare different algorithms. A benchmark suite can only compare the implementations of different algorithms. In this section, I evaluate the macro benchmark execution time measurements from executing my join benchmark suites multiple times. As of now, I will refer to the macro benchmark execution time measurements from executing my join benchmark suites once, simply as a macro benchmark measurement.

A few small reminders:

- Benchmark: Source [2] defines a "benchmark" as, "a computer program that measures the ... speed of computer software ... ". In this thesis, a benchmark measures the execution time of computer software.
- Workload: Source [3] defines "workload" as, "the amount of work to be done, especially by a ... machine in a period of time ... ". In this thesis, the machine referenced is always a computer.

As mentioned in section 4.3, the content of the entries in "`IdTable`" instances in the join benchmark suites are randomly generated natural numbers [1]. Using a runtime configuration option, a user can set the seed for the generation of seeds for the random natural number generators in the join benchmark suites. I will refer to the seed for the generation of seeds for the random natural number generators simply as the benchmark seed.

As said in section 4.4.2 I executed my join benchmark suites 100 times. I do not use the same benchmark seed for every execution of my join benchmark suites. However, I, also, did not set the benchmark seed to a unique value for every execution of my join benchmark suites. See table 4.5 for what value the benchmark seed was set to for what execution of my join benchmark suites.

I executed my join benchmark suites with different values for the benchmark seed, in order to be able to make general statements based on the macro benchmark measurements about the implementations of the merge and galloping join algorithm, see section 4.2.1, and the hash join algorithm, see section 4.2.2. For

this paragraph I will refer to the merge and galloping join algorithm and the hash join algorithm as the join algorithms. When evaluating macro benchmark measurements from multiple join benchmark suites executions with the same value for the benchmark seed, any statements about the implementations of the join algorithms are not general statements. When evaluating macro benchmark measurements from multiple join benchmark suites executions with the same value for the benchmark seed, the macro benchmark measurements can only give information about the implementations of the join algorithms when executed with the `IdTable` instances generated for this value of the benchmark seed, as input. Remember, the content of the entries in `IdTable` instances in the join benchmark suites are randomly generated natural numbers. Executing my join benchmark suites with different values for the benchmark seed, the macro benchmark measurements can still only give information about the implementations of join algorithms when executed with the `IdTable` instances generated for these values for the benchmark seed, as input. However, the more different values for the benchmark seed I use for the join benchmark suites executions, the more general any statements based on the macro benchmark measurements is.

As explained in section 4.3, every macro benchmark suite in my join benchmark suites organizes its macro benchmarks with tables, see section 2.1. I will refer to a table for the organization of macro benchmarks in my join benchmark suites as macro benchmark tables. Every macro benchmark table in my join benchmark suites follows the same structure. I will refer to the structure followed by every macro benchmark table in my join benchmark suites as the join benchmark table structure. Every row of a macro benchmark table following the join benchmark table structure has its own two `IdTable` instances and follows the same structure, see section 4.3.

I will refer to a row of a macro benchmark table following the join benchmark table structure as a macro benchmark row.

As a reminder, see the explanation in section 4.3, every macro benchmark row contains a hash join speedup. The hash join speedup of a macro benchmark row is the result of the calculation 4.1 below. The sentence “Execution time merge and galloping join” in calculation 4.1 is an abbreviation of the sentence “The macro benchmark rows macro benchmark measured execution time of the implementation of the merge and galloping join algorithm”. The sentence “Execution time sorting `IdTable` instances” in calculation 4.1 is an abbreviation of the sentence “The macro benchmark rows macro benchmark measured execution

time of sorting the `IdTable` instances of the macro benchmark row with the internal sorting function in QLever SPARQL engine". For more information on the sorting of the `IdTable` instances of the macro benchmark row see section 4.3. The sentence "Execution time hash join" in calculation 4.1 is an abbreviation of the sentence "The macro benchmark rows macro benchmark measured execution time of the implementation of the hash join algorithm".

$$\frac{\text{Execution time merge and galloping join} + \text{Execution time sorting } \texttt{IdTable} \text{ instances}}{\text{Execution time hash join}} \tag{4.1}$$

As explained in section 4.3 the hash join speedup truly shows how much faster the macro benchmark measured execution time of the implementation of the hash join algorithm is than the macro benchmark measured execution time of the implementation of the merge and galloping join algorithm.

When the internal measurement tool used by my macro-benchmarking library for the QLever SPARQL engine measures the execution time of computer software, the internal measurement tool used by my macro-benchmarking library for the QLever SPARQL engine does not stop measuring the execution time of computer software the moment the computer software finishes. As of now, I will refer to the internal measurement tool used by my macro-benchmarking library for the QLever SPARQL engine simply as the internal measurement tool. The internal measurement tool can not stop measuring the execution time of computer software the moment the computer software finishes because of overhead in the computer software and the computer temporarily switching to the execution of different computer software. The time difference between the execution time of computer software and the execution time of computer software measured by internal measurement tool is called noise.

I executed my join benchmark suites multiple times with the same value for the benchmark seed, in order to identify measurements in the macro benchmark measurements with too much noise. I define a measurement in the macro benchmark measurements as having too much noise, when, in the worst case, over 90% of the measurement in the macro benchmark measurements is noise. Different macro benchmarks have different minimum and maximum bounds for noise. Hardware also affects minimum and maximum bounds for noise. In order to identify noise, execution time measurements from multiple executions of the same macro benchmark must be compared.

My algorithm for the identification of macro benchmark measurements with too much noise does not catch all macro benchmark measurements with too much noise. However, my algorithm also has barely any false identifications. My algorithm executes the following steps for every set of all macro benchmark measurements from the same macro benchmark:

1. Identify the smallest and biggest macro benchmark measurement.
2. Designate the absolute different between the smallest and biggest macro benchmark measurement as the max noise value. The max noise value is the biggest possible value for the noise in the macro benchmark. The noise in the macro benchmark must be smaller than, or equal to, the max noise value.
3. If a macro benchmark measurement is smaller than, or equal to, the max noise value, the macro benchmark measurement has too much noise.

When every macro benchmark measurement in a macro benchmark row has too much noise, I delete the macro benchmark row. It is not possible to extract any meaningful information from a macro benchmark row where every macro benchmark measurement has too much noise. If at least one macro benchmark measurement in a macro benchmark row does not have too much noise, the difference between a macro benchmark measurement in the macro benchmark row with not too much noise and macro benchmark measurement in the macro benchmark row with too much noise should be big enough for the hash join speedup in the macro benchmark row to hold usable information. After all, if one value is very small and another value very big, any changes in the very small value would not be noticeable in any division of the two values.

Every row of a macro benchmark table in my join benchmark suites has its own two `IdTable` instances. Of the two `IdTable` instances the `IdTable` instance with the biggest number of rows is referred to as the bigger input table. The `IdTable` instance with the smallest number of rows is referred to as the smaller input table. Inside my join benchmark suites, I always pass the smaller input table as the first parameter to the implementations of the algorithms in section 4.2 and the bigger input table as the second parameter to the implementations of the algorithms in section 4.2.

The column in the smaller input table at the column index number position for the smaller input table is referred to as the smaller join column. Vice versa for the bigger input table. The smaller input table is sorted, when the smaller input table is sorted by the content of the smaller join column. Vice versa for the

bigger input table.

At the point in time a macro benchmark in a macro benchmark row measures the execution time of sorting the smaller input table, see section 4.3, the smaller input table is either already sorted, or unsorted. Vice versa for the bigger input table. For this list entry I will refer to the at the point in time a macro benchmark measures the execution time of sorting the smaller and bigger input table as the sorting time point. The sorting status of the smaller and bigger input table at the sorting time point is called the sorting configuration.

I refer to the macro benchmark measurements from executing my join benchmark suites 100 times and without all the macro benchmark rows where all measurements have too much noise as the noiseless macro benchmark measurements. I refer to a set of all the executions of a macro benchmark row in the noiseless macro benchmark measurements as the noiseless macro benchmark row executions. I refer to a subset of a noiseless macro benchmark row executions containing all executions of the macro benchmark row with the same value for the benchmark seed and the same sorting configuration as a repeated noiseless macro benchmark row.

Sometimes, when executing a macro benchmark there is a spike in noise. A spike in noise can be caused by things like a temporary increase of the workload for the computer, higher priority processes on the computer requiring attention by the computer, or similar situations. The execution time measured by a macro benchmark with a spike in noise can be far bigger than any execution time measured by the same macro benchmark without a spike in noise.

On the opposite side, sometimes, when executing a macro benchmark there is a drop in noise. A drop in noise can be caused by things like a temporary decrease of the workload for the computer, higher priority processes on the computer closing, or similar situations. The execution time measured by a macro benchmark with a drop in noise can be far smaller than any execution time measured by the same macro benchmark without a drop in noise.

A macro benchmark row containing macro benchmarks with a spike or drop in noise, in a repeated noiseless macro benchmark row can contain a hash join speedup far bigger, or far smaller, than any of the other hash join speedups in the remaining macro benchmark rows in the same repeated noiseless macro benchmark row. I call the hash join speedup in macro benchmark row in a repeated noiseless macro benchmark row that is far bigger, or far smaller, than any of the other hash join speedups in the remaining macro benchmarks rows in the same repeated

noiseless macro benchmark row an outlier hash join speedup.

When trying to approximate the mean of the hash join of a repeated noiseless macro benchmark row, an outlier hash join speedup in the same repeated noiseless macro benchmark row can distort the value of the approximated mean of the hash join of the repeated noiseless macro benchmark row away from the true mean of the hash join of the repeated noiseless macro benchmark row. Because I want to approximate the mean of the hash join of multiple macro benchmark rows later for the evaluation of the macro benchmark execution time measurements from executing my join benchmark suites multiple times, I try to identify and remove macro benchmark rows, containing outlier hash join speedups, from repeated noiseless macro benchmark rows.

My algorithm for the identification of outlier hash join speedups in repeated noiseless macro benchmark rows does not catch all outlier hash join speedups in repeated noiseless macro benchmark rows. My algorithm uses the so called "1.5 IQR rule" [8].

As explained in [8], the 1.5 IQR rule is used for the definition of outliers in statistical analysis. Given a list of numbers, the 1.5 IQR rule defines a lower and upper bound. The 1.5 IQR rule defines any entry of the given list of numbers outside the defined lower and upper bound as an outlier of the given list of numbers.

My algorithm for the identification of outlier hash join speedups in repeated noiseless macro benchmark rows executes the following steps for every repeated noiseless macro benchmark row:

1. Collect all hash join speedups from the macro benchmark rows inside the repeated noiseless macro benchmark row.
2. Calculate the lower and upper bound defined by the 1.5 IQR rule for the collected hash join speedups.
3. Go through every macro benchmark rows inside the repeated noiseless macro benchmark row. If the hash join speedup of a macro benchmark row is outside the lower and upper bound defined by the 1.5 IQR rule for the collected hash join speedups, the hash join speedup of the macro benchmark row is an outlier hash join speedup.

When a macro benchmark row in a repeated noiseless macro benchmark rows contains an outlier hash join speedups, I delete the macro benchmark row.

I refer to the noiseless macro benchmark measurements without all the macro benchmark rows containing outlier hash join speedups as the filtered macro bench-

mark measurements.

Before evaluating the filtered macro benchmark measurements, I need to define a few terms:

- The row ratio is defined as the number of rows in the bigger input table divided by the number of rows in the smaller input table. In the join benchmark suites, the number of rows in the bigger input table are set to the result of multiplying the number of rows in the smaller input table with the row ratio.
- Sorting merge and galloping join algorithm refers to an algorithm. Sorting merge and galloping algorithm first sorts the smaller and the bigger input table with the internal sorting function in QLever SPARQL engine. After sorting the smaller and bigger input table, the sorting merge and galloping algorithm executes the merge and galloping algorithm with the sorted smaller and bigger input table.

For the evaluation of the filtered macro benchmark measurements I focus on the macro benchmark rows of the filtered macro benchmark measurements. Remember, my join benchmark suites were executed 100 times. I refer to a set of all the executions of a macro benchmark row in the filtered macro benchmark measurements as the filtered macro benchmark row executions. I refer to a subset of a filtered macro benchmark row executions containing all executions of the macro benchmark row with the same value for the benchmark seed and the same sorting configuration as a repeated filtered macro benchmark row. For any repeated filtered macro benchmark row I further focus on the row ratio, the sorting configuration, and the mean of the hash join speedup.

The hash join speedup shows, as explained in section 4.3, how much faster the implementation of the hash join algorithm truly is than the implementation of the merge and galloping join algorithm. If the hash join speedup is bigger than one, then the implementation of the hash join algorithm is faster than the implementation of the sorting merge and galloping join algorithm. If the hash join speedup is smaller than one, then the implementation of the hash join algorithm is slower than the implementation of the sorting merge and galloping join algorithm

The focus on the sorting configuration it to observe the impact on the hash join speedup by the execution time of sorting the smaller and bigger input table.

I choose the focus on the row ratio, because a metric to decide between the algorithms in section 4.2 was needed. A quick look at the generated diagrams from the filtered macro benchmark measurements, see figure 4.1, figure 4.2, figure 4.3

and figure 4.4, shows, that the row ratio allows for a clear decision between the algorithms in section 4.2.

Now, on the actual evaluation of the filtered macro benchmark measurements. I created four diagrams, see figure 4.1, figure 4.2, figure 4.3 and figure 4.4, for the evaluation of the filtered macro benchmark measurements.

Every diagram for the evaluation of the filtered macro benchmark measurements has the same type of value on the vertical axis. The type of value on the vertical axis is the approximated mean of the hash join speedups inside a repeated filtered macro benchmark row. Remember, the hash join speedup of a macro benchmark row can be found as an entry in the same macro benchmark row. Furthermore, the scale of the vertical is logarithmic with base ten.

Before I explain my algorithm for the approximation of the mean of the hash join speedups inside a repeated filtered macro benchmark row, I need to give a broad overview over the "confidence interval" formula [9]. For more than a broad overview see [9].

The calculated mean of, for example, multiple executions of the same benchmark can be quite far off the actual mean of the same benchmark. For example, let the actual mean of imaginary benchmark A be one second. I execute benchmark A five times. Because my computer was under a big workload while I executed the benchmark A five times, every execution time was measured as longer than two seconds by benchmark A. The calculated mean of benchmark A is, therefore, around two seconds and not near the actual mean of benchmark A.

As explained in [9], the confidence interval formula approximates inclusive boundaries for the true mean of a set of values. However, the result of the confidence interval formula is not guaranteed to be always correct. The chance of the result of the confidence interval formula being correct can be adjusted. The chance of the result of the confidence interval formula being correct is called, according to [9], the "confidence level". The higher the confidence level the broader the approximation of the inclusive boundaries for the true mean by the confidence interval formula. A confidence level of 100% is not possible.

For the approximation of the mean of the hash join speedups inside a repeated filtered macro benchmark row, I use a rather simple algorithm. The steps of the rather simple algorithm for a repeated filtered macro benchmark row are:

1. If there is only one macro benchmark row inside the repeated filtered macro benchmark row, this algorithm finishes and returns no values. There are not enough values for an approximation with the confidence interval formula.

2. I approximate the inclusive boundaries for the true mean of the hash join speedups inside the repeated filtered macro benchmark row with the confidence interval formula. I use a 99% confidence level for the confidence interval formula. I will refer to the result of this step as the confidence interval boundaries.
3. If the lower bound of the confidence interval boundaries is less than zero and the upper bound of the confidence interval boundaries bigger than one, this algorithm finishes and returns no values.

   For a lower bound of the confidence interval boundaries less than zero and an upper bound of the confidence interval boundaries bigger than one the true mean of the hash join speedups inside the repeated filtered macro benchmark row must be in $(0, \text{Upper bound of the confidence interval boundaries}]$. A true mean in $(0, \text{Upper bound of the confidence interval boundaries}]$ gives no information if the implementation of the hash join algorithm is faster than the implementation of the sorting merge and galloping algorithm. A true mean in $(0, \text{Upper bound of the confidence interval boundaries}]$ only gives the information that the implementation of the hash join algorithm can be faster than the implementation of the sorting merge and galloping algorithm, because the upper bound of the confidence interval boundaries is bigger than one, and that the implementation of the hash join algorithm can be infinitely slower than the implementation of the sorting merge and galloping algorithm. This information is useless for comparing the execution time of the implementation of the hash join algorithm with the execution time of the implementation of the sorting merge and galloping algorithm.
4. If the lower bound of the confidence interval boundaries is less than zero, the lower bound of the confidence interval boundaries is set to 0.01. Because the true mean of the hash join speedups inside the repeated filtered macro benchmark row can never be negative, a very small value is more true to reality than a negative value. It does not matter if the true mean of the hash join speedups inside the repeated filtered macro benchmark row is smaller than 0.01. A true mean of the hash join speedups inside the repeated filtered macro benchmark row, with a lower bound of 0.01 already shows that the implementation of the hash join algorithm can be much slower than the implementation of the sorting merge and galloping algorithm.
5. The confidence interval boundaries are returned.

Note, that two values are returned by the algorithm above. Therefore, every

repeated filtered macro benchmark row delivers two approximations of the mean of the hash join speedups inside the repeated filtered macro benchmark row.

I will now explain and interpret the four diagrams I created for the evaluation of the filtered macro benchmark measurements.

Figure 4.1 takes all repeated filtered macro benchmark rows. The horizontal axis of Figure 4.1 sorts all repeated filtered macro benchmark rows by their sorting configuration.

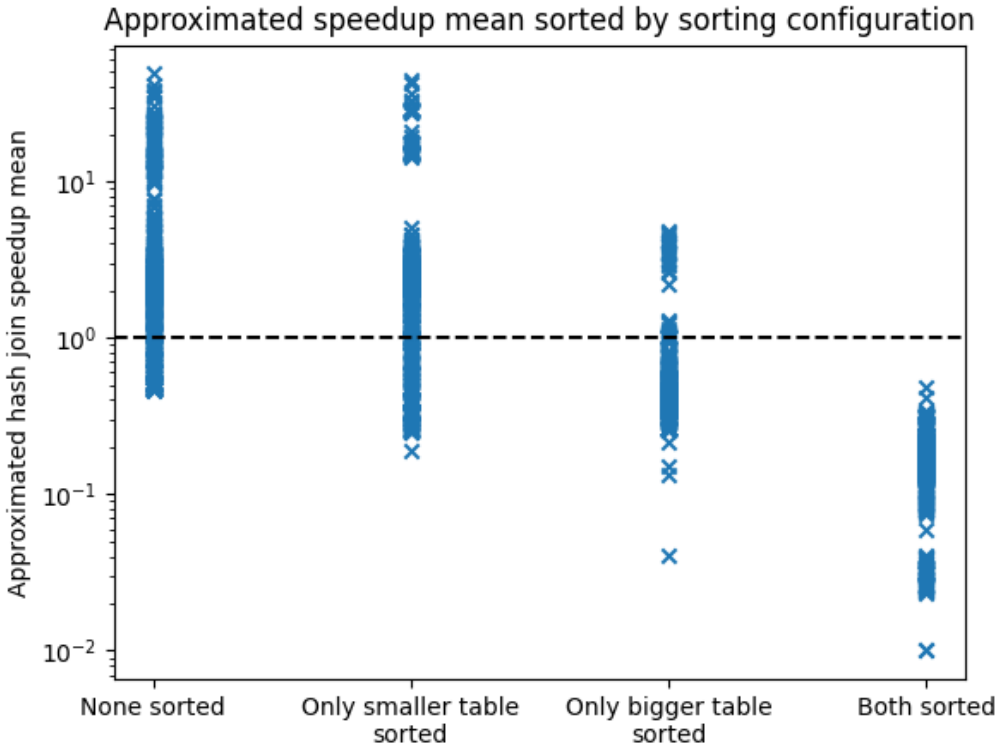

Figure 4.1: All repeated filtered macro benchmark rows sorted by their sorting configuration.

As I can see, when the sorting configuration is both the smaller and bigger input table unsorted, the implementation of the hash join algorithm is always significantly slower than the implementation of the sorting merge and galloping join algorithm. All the other sorting configurations can have both the implementation of the hash join algorithm being faster than the implementation of the sorting merge and galloping join algorithm and the implementation of the hash join algorithm being slower than the implementation of the sorting merge and galloping join algorithm.

Figure 4.2 takes all repeated filtered macro benchmark rows with the sorting configuration where the smaller and bigger input table are not sorted. The horizontal axis of Figure 4.2 sorts all repeated filtered macro benchmark rows by their row ratio.

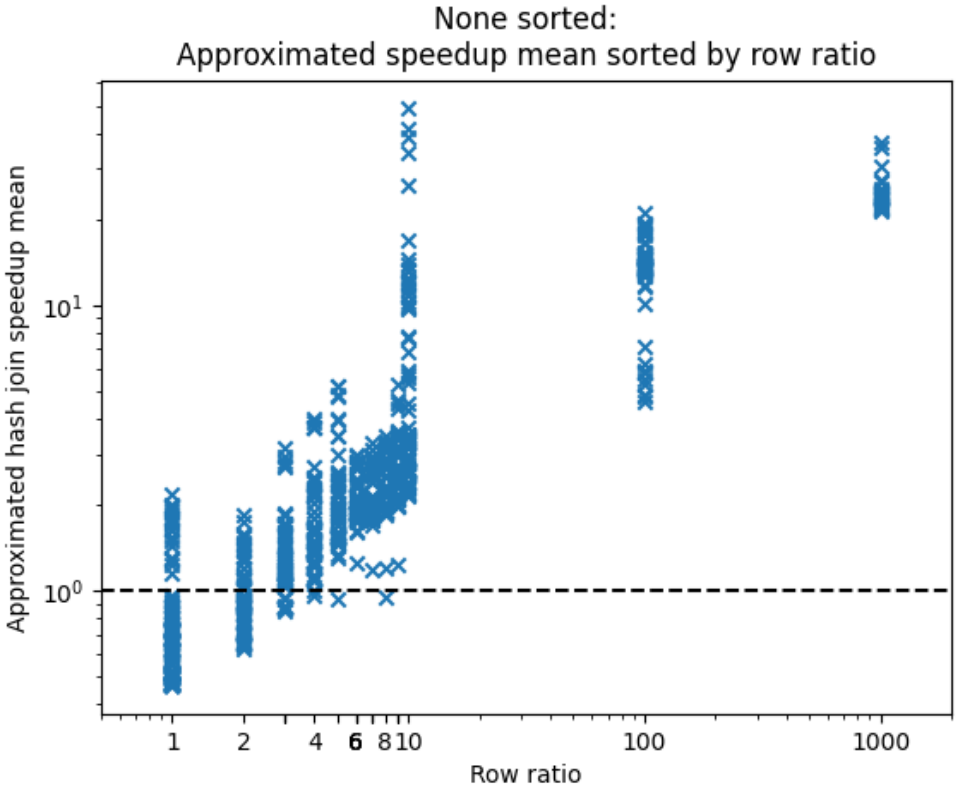

Figure 4.2: All repeated filtered macro benchmark rows where both the smaller and bigger input table are sorted, sorted by their row ratio.

As I can see, when the sorting configuration is both the smaller and bigger input table unsorted and the row ratio is bigger than three, the implementation of the hash join algorithm is always faster than, or about as fast as, the implementation of the sorting merge and galloping join algorithm.

Figure 4.3 takes all repeated filtered macro benchmark rows with the sorting configuration where the smaller input table is sorted and the bigger input table is not sorted. The horizontal axis of Figure 4.3 sorts all repeated filtered macro benchmark rows by their row ratio.

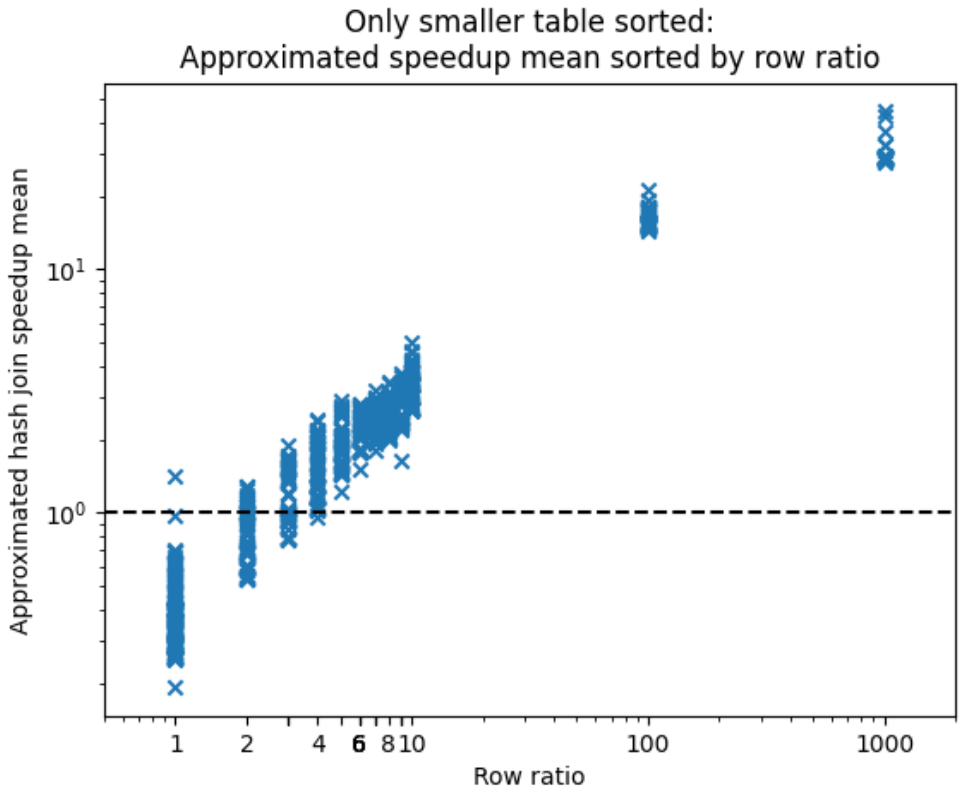

Figure 4.3: All repeated filtered macro benchmark rows where the smaller input table is sorted, and the bigger input table is not sorted, sorted by their row ratio.

As I can see, when the sorting configuration is the smaller input table is sorted, the bigger input table is not sorted, and the row ratio is bigger than three, the implementation of the hash join algorithm is always faster than, or about as fast as, the implementation of the sorting merge and galloping join algorithm.

Figure 4.4 takes all repeated filtered macro benchmark rows with the sorting configuration where the smaller input table is not sorted and the bigger input table is sorted. The horizontal axis of Figure 4.4 sorts all repeated filtered macro benchmark rows by their row ratio.

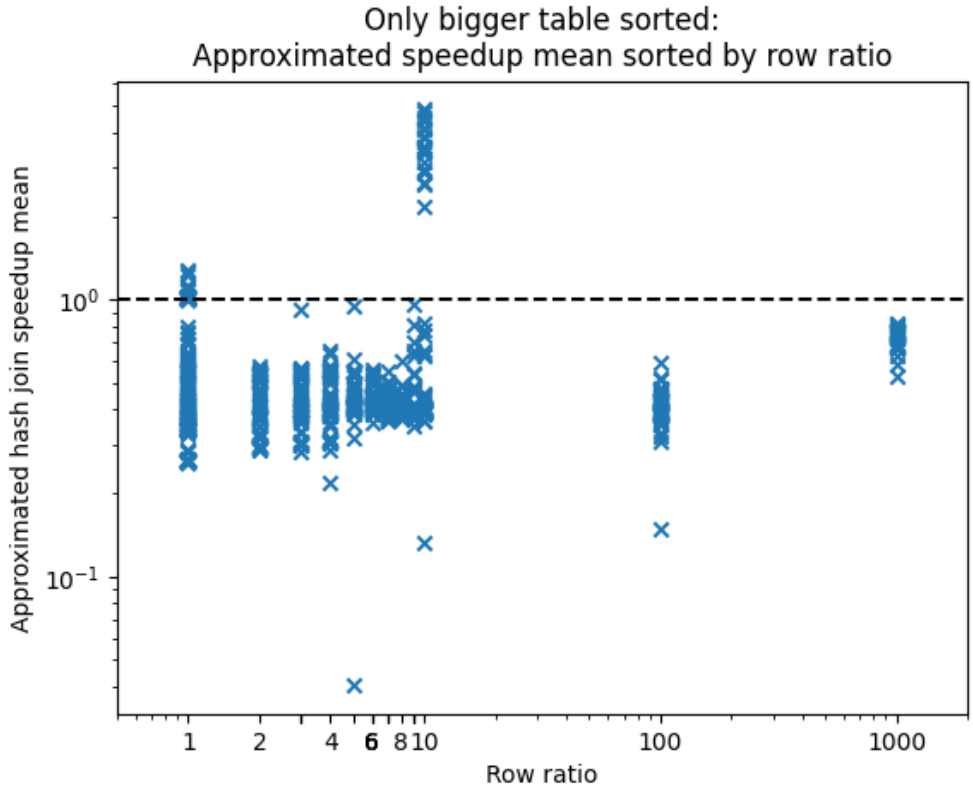

Figure 4.4: All repeated filtered macro benchmark rows where the smaller input table is not sorted, and the bigger input table is sorted, sorted by their row ratio.

As I can see, when the sorting configuration is the smaller input table is not sorted and the bigger input table is sorted, the implementation of the hash join algorithm is always slower than, or about as fast as, the implementation of the sorting merge and galloping join algorithm. There are outliers where the implementation of the hash join algorithm is faster than the implementation of the sorting merge and galloping join algorithm. However, those outliers are too unreliable.

If I combine the conclusions drawn from figure 4.1, figure 4.2, figure 4.3 and figure 4.4, then there exists a combination of sorting configurations and a minimum row ratio where the implementation of the hash join algorithm is always faster than, or about as fast as, the implementation of the sorting merge and galloping join algorithm. When the the bigger input table is not sorted, the smaller input table is either sorted, or not sorted, and the row ratio is bigger than three, than the implementation of the hash join algorithm is always faster than, or about as fast as, the implementation of the sorting merge and galloping join

algorithm.

Chapter 5

Conclusion

I have created a relatively easy to use internal macro-benchmarking library for the QLever SPARQL engine with a focus on organizing macro-benchmarks and processing runtime configuration options.

I have compared the features of my macro-benchmarking library for the QLever SPARQL engine with the features of nine different third-party micro-benchmarking libraries. I was unable to find any third-party macro-benchmarking libraries for the ‘C++‘-language. The nine different micro-benchmarking libraries are:

- “Picobench” [10]
- “Nanobench” [11]
- “Catch2” [12]
- “Sltbench” [13]
- “Folly benchmark component” [14]
- “CppBenchmark” [15]
- “Hyperfine” [16]
- “Celero” [17]
- “Google benchmark” [18]

As a reminder, the QLever SPARQL engine has a focus on short execution times for big workloads. In the context of the QLever SPARQL engine, a short execution time for a big workload can mean multiple seconds to multiple days. Micro benchmarks are designed for execution times in the range of milliseconds. Multiple seconds to multiple days are better measured with a macro benchmarks, who are designed for long execution times in the range of one second and longer.

Following from this information, when working with execution times longer than one second, my macro-benchmarking library for the QLever SPARQL engine tends to be the better choice than any of the nine different micro-benchmarking

libraries. Furthermore, my macro-benchmarking library for the QLever SPARQL engine offers better organization of the benchmarks in the form of groups and tables, and offers runtime configuration option support. Any of the nine different third-party micro-benchmarking libraries supports, at most, the organization of benchmarks into groups. Furthermore, eight of the nine different third-party micro-benchmarking libraries do not support runtime configuration option in any way. Only "Sltbench" support runtime configuration options, as far as I could find [13]. However, the Sltbench runtime configuration options support seems to be barely documented and seems to only support basic value passing, based on some example benchmarks I could find in the documentation of Sltbench.

When working with execution times shorter than one second, any of the nine different micro-benchmarking libraries are a better choice than my macro-benchmarking library for the QLever SPARQL engine. The measurement tool used by my macro-benchmarking library for the QLever SPARQL engine is too inaccurate to use for execution times shorter than one second. Any of the nine different micro-benchmarking libraries are more accurate when measuring execution times shorter than one second. Any of the nine different micro-benchmarking libraries are more accurate when measuring execution times shorter than one second, because their time measurement tools are more accurate, and they implement algorithms to reduce measurement errors. Some of the nine different micro-benchmarking libraries also support more measurement metrics than execution time.

While my macro-benchmarking library for the QLever SPARQL engine has its own niche in macro benchmarks, there is room for improvement:

- Currently, the value of a runtime configuration option must be a boolean, string, number, array of boolean, array of string, or array of number. Explicit support for creating a more complex value by interpreting a "JSON" structure, like a JSON object, would allow a user more flexibility [4]. Being able to create a more complex value by interpreting a JSON structure, could also lead to a more informative runtime configuration option documentation.
- Currently, the benchmark main function executes every macro benchmark in a registered macro benchmark suite exactly once. Most algorithms for the reduction of execution time measurement errors require multiple execution time measurements from the repeated execution of the same benchmark. Extending the benchmark main function to execute every macro benchmark in a registered macro benchmark suite a given number of times, would allow

an easy creation of repeated macro benchmark measurements. The repeated macro benchmark measurements could then be written to a JSON file. If a user requires more accurate execution measurements, the user could then use the JSON file of repeated macro benchmark measurements in conjunction with algorithms for the reduction of execution time measurement errors to reduce the execution time measurement error.

While I was working on an extensive example macro benchmark for my bachelor thesis, using my macro-benchmarking library for the QLever SPARQL engine, I identified three, for now, problems with the ideas behind my macro-benchmarking library for the QLever SPARQL engine.

Firstly, extracting the information from the content of the JSON file that contains the registered macro benchmark suites metadata and measurements, and that is generated by the benchmark main function, is more complex than anticipated. As of now, I will refer to the JSON file that contains the registered macro benchmark suites metadata and measurements, and that is generated by the benchmark main function, as the benchmark JSON file. The content of the benchmark JSON file is not missing any registered macro benchmark suites metadata or measurements. The JSON structure in the benchmark JSON file is complex enough, that a human can not comprehend all content of the benchmark JSON file by reading it. Purely interpreting the JSON structure of the benchmark JSON file with a program, takes, in my experience, over an hour of work. Reworking the structure of the benchmark JSON file to be simpler, could lessen the time needed to interpret, with a program, the JSON structure of the benchmark JSON file.

Secondly, my macro-benchmarking library for the QLever SPARQL engine does not support the repetition of macro benchmarks. Originally, I thought that the fluctuation of macro benchmarks execution time measurements would be ignorable. In micro benchmarks, the fluctuation of execution time measurements is in the range of, at most, “centiseconds” [19]. According to source [19] centisecond is 0.01 seconds. Centiseconds are ignorable in macro benchmark execution time measurements, where the focus is on execution time measurements longer than one second. However, a look at the repeated measurement of my example macro benchmark for my bachelor thesis, using my macro-benchmarking library for the QLever SPARQL engine, has shown fluctuation of execution time measurements in macro benchmarks, that are in the range of seconds. Fluctuations in the range of seconds are not ignorable in macro benchmark execution time measurements.

However, when comparing two macro benchmarks whose difference in execution time is multiple seconds, fluctuations in the range of seconds are ignorable. After all, fluctuations in the range of seconds do not change which execution time is faster, when comparing execution times with a difference of multiple seconds.

The fluctuations of macro benchmark execution time measurements can not be removed. However, there are multiple algorithms, that approximate the true average of a macro benchmark execution time measurement if given a set of the same macro benchmark execution time measurement repeated multiple times. As of now, I will refer to algorithms that approximate the true average of a measurement if given a set of the same measurement repeated multiple times as average measurement approximation algorithms. Average measurement approximation algorithms can be found in micro benchmarking system, like the nine different micro-benchmarking libraries I have mentioned before. A user could use average measurement approximation algorithms easier if my macro-benchmarking library for the QLever SPARQL engine supported the repetition of macro benchmarks. Alternatively, average measurement approximation algorithms could be directly integrated into my macro-benchmarking library for the QLever SPARQL engine. However, as said at the end of the previous paragraph, when comparing two macro benchmarks whose difference in execution time is multiple seconds, the execution times can be compared without further processing. Executing a macro benchmark multiple times in order to use an average measurement approximation algorithm would only waste time, when comparing two macro benchmarks whose difference in execution time is multiple seconds. As a reminder, an execution of a macro benchmark can also take multiple seconds to multiple days. Repeated execution of a macro benchmark could take over a week, even if repeated execution is not needed. If any average measurement approximation algorithms are directly integrated into my macro-benchmarking library for the QLever SPARQL engine, then, the usage of any integrated average measurement approximation algorithms must be configurable.

Thirdly, when the internal measurement tool used by my macro-benchmarking library for the QLever SPARQL engine measures the execution time of computer software, the internal measurement tool used by my macro-benchmarking library for the QLever SPARQL engine does not stop measuring the execution time of computer software the moment the computer software finishes. As of now, I will refer to the internal measurement tool used by my macro-benchmarking library for the QLever SPARQL engine simply as the internal measurement tool. The

internal measurement tool can not stop measuring the execution time of computer software the moment the computer software finishes because of overhead in the computer software and the computer temporarily switching to the execution of different computer software. The time difference between the execution time of computer software and the execution time of computer software measured by internal measurement tool is called noise.

It is possible for a macro benchmark execution time measurement to nearly entirely consist of noise. As of now, I will refer to a macro benchmark execution time measurement that nearly entirely consist of noise as a noisy macro benchmark measurement. Noisy macro benchmark happen when the execution time of the measured computer software is too short, often less than 0.1 seconds. A noisy macro benchmark measurement is useless to the user, because, the noisy macro benchmark measurement does not say anything about the macro benchmark. However, identifying a noisy macro benchmark measurement is harder than I anticipated. Currently, my macro-benchmarking library for the QLever SPARQL engine has no support for filtering out noisy macro benchmark measurements, because, I have not found a way to identify a noisy macro benchmark measurement using only a single macro benchmark execution. Furthermore, even with repeated macro benchmark executions, I have yet to find a reliable way to identify a noisy macro benchmark measurement, that is capable of identifying every noisy macro benchmark measurement.

Theoretically, the problem of noisy macro benchmark measurements could be solved by the user. The user could manually compare repeated macro benchmark executions. If the execution time measurements from repeated macro benchmark executions look like noisy macro benchmark measurements, a user could either delete the macro benchmark, or adjust the workload for the macro benchmark until the execution time measurements have an acceptable level of noise. Source [3] defines “workload” as, “the amount of work to be done, especially by a ... machine in a period of time ... ”. In this thesis, the machine referenced is always a computer. However, giving the responsibility of solving the noisy macro benchmark measurement problem to the user, has a few problems. The main problem is that noisy macro benchmark measurements are not easy to identify. Different macro benchmarks have different minimum and maximum bounds for noise. Hardware also affects minimum and maximum bounds for noise. A user would have to examine repeated macro benchmark measurements for noisy macro benchmark behavior. However, noisy macro benchmark behavior is hard to identify when the minimum

and maximum bounds for noise are not known. Additionally, the examination of multiple repeated macro benchmark measurements could take a lot of time, and mental energy of the user, when there are a lot of macro benchmarks. A user may not wish to spend his time examining multiple repeated macro benchmark measurements, every time, when he wants to evaluate his macro benchmarks.

Glossary

Benchmark Source [2] defines a “benchmark” as, “a computer program that measures the ... speed of computer software ... ”. In this thesis, a benchmark measures the execution time of computer software.

Benchmark suite Set of benchmarks.

Metadata Source [5] defines “metadata” as, “information that is given to describe or help you use other information ... ”.

Workload Source [3] defines “workload” as, “the amount of work to be done, especially by a ... machine in a period of time ... ”. In this thesis, the machine referenced is always a computer.

Bibliography

[1] Chair of Algorithms and Data Structures of the institute for computer science at the University of Freiburg, *QLever SPARQL engine*, version commit 57ba939b32ad30f3c4802f2e6050aedc1b13b2d9, Jun. 7, 2024. [Online]. Available: `https://github.com/ad-freiburg/qlever` (visited on 06/07/2024).

[2] "BENCHMARK | English meaning - Cambridge Dictionary." (), [Online]. Available: `https://dictionary.cambridge.org/dictionary/english/benchmark` (visited on 06/16/2024).

[3] "WORKLOAD | English meaning - Cambridge Dictionary." (), [Online]. Available: `https://dictionary.cambridge.org/dictionary/english/workload` (visited on 06/16/2024).

[4] Wikipedia contributors. "JSON — Wikipedia, The Free Encyclopedia." (Jun. 14, 2024), [Online]. Available: `https://en.wikipedia.org/w/index.php?title=JSON&oldid=1229014619` (visited on 06/20/2024).

[5] "METADATA | English meaning - Cambridge Dictionary." (), [Online]. Available: `https://dictionary.cambridge.org/dictionary/english/metadata` (visited on 06/16/2024).

[6] *Nlohmann-JSON*, version v3.11.3, Nov. 28, 2023. [Online]. Available: `https://github.com/nlohmann/json` (visited on 06/01/2024).

[7] *Boost*, version 1.81, Dec. 14, 2022. [Online]. Available: `https://www.boost.org/` (visited on 06/01/2024).

[8] S. Chaudhary. "Why 1.5 Is Used in the IQR Rule for Outlier Detection." (Jan. 24, 2024), [Online]. Available: `https://builtin.com/articles/1-5-iqr-rule` (visited on 06/15/2024).

[9] R. Bevans. "Understanding Confidence Intervals | Easy Examples & Formulas." (Aug. 7, 2020), [Online]. Available: `https://www.scribbr.com/statistics/confidence-interval/` (visited on 06/12/2024).

[10] *Picobench*, version 2.07, Mar. 6, 2024. [Online]. Available: https://github.com/iboB/picobench (visited on 05/31/2024).

[11] *Nanobench*, version 4.3.11, Feb. 16, 2023. [Online]. Available: https://nanobench.ankerl.com/ (visited on 05/31/2024).

[12] *Catch2*, version 3.6.0, May 5, 2024. [Online]. Available: https://github.com/catchorg/Catch2 (visited on 05/31/2024).

[13] *Sltbench*, version r-2.4.0, Aug. 23, 2020. [Online]. Available: https://github.com/ivafanas/sltbench?tab=readme-ov-file (visited on 05/31/2024).

[14] *Folly benchmark component*, version v2024.05.06.00, May 6, 2024. [Online]. Available: https://github.com/facebook/folly (visited on 05/31/2024).

[15] *CppBenchmark*, version 1.0.4.0, Sep. 27, 2023. [Online]. Available: https://github.com/chronoxor/CppBenchmark (visited on 05/31/2024).

[16] *Hyperfine*, version v.1.18.0, Oct. 5, 2023. [Online]. Available: https://github.com/sharkdp/hyperfine (visited on 05/31/2024).

[17] *Celero*, version v2.8.5, Dec. 26, 2022. [Online]. Available: https://github.com/DigitalInBlue/Celero (visited on 05/31/2024).

[18] *Google benchmark*, version v1.8.3, Aug. 31, 2023. [Online]. Available: https://github.com/google/benchmark (visited on 05/31/2024).

[19] Wikipedia contributors. "Orders of magnitude (time) — Wikipedia, The Free Encyclopedia." (Jun. 17, 2024), [Online]. Available: https://en.wikipedia.org/w/index.php?title=Orders_of_magnitude_(time)&oldid=1229602925 (visited on 06/20/2024).