Porting QLever to WebAssembly using CMake, Conan and Emscripten

Zamir Çaçi-Schätzle

Matriculation Number: 4742858

Bachelor's Thesis

Examiner

Prof. Dr. Hannah Bast

Supervisor

Robin Textor-Falconi

University of Freiburg

Department of Computer Science

February 27, 2026

Declaration

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

Place, Date	Signature

Contents

1 Introduction **6**

1.1 Compiling C/C++ code 6

1.2 Emscripten and the WebAssembly code format 7

1.3 Make, CMake and external libraries 7

2 The main problem and possible solutions **9**

2.1 Choosing a solution 10

3 Conan package manager **11**

3.1 Libraries and dependencies 11

3.2 Conan profiles 11

3.3 Cross-building 12

3.4 The host profile 12

3.4.1 Software libraries and their components 13

3.4.2 Emscriptens compiler settings 13

3.5 Conanfile.txt 14

4 Choosing the emsdk **15**

5 Adjustments to the QLever source code **15**

5.1 Macros 15

5.1.1 Macro example 15

5.1.2 Defining a helper macro 16

5.1.3 EXPECT_DEATH and ASSERT_DEATH 17

5.2 The build commands 17

5.2.1 Note regarding memory consumption 18

6 Evaluation of the compiled output **19**

7 Adjusting the code for the Web **20**

7.1 Generate wasm and js for libraries 20

7.2 Writing some glue code 21

7.2.1 Web IDL 21

7.2.2 Embind 21

7.2.3 Virtual file system 24

8 Executing the code on the web **25**

9 Evaluation **25**

10 Related work **26**

10.1 Ammo.js 26

10.2 d3wasm 26

10.3 Qt WebAssembly 26

11 Future work **27**

11.1 Using a newer version of the emsdk 27

11.2 Testing the compiled output 27

11.3 buildIndex 27

12 Acknowledgements **28**

13 Figure sources **29**

Abstract

QLever [2] is a graph database that can efficiently load very large datasets, model that data with RDF and query these models using the SPARQL language. Recent improvements [3] regarding the memory consumption of QLever have enabled the possibility to run QLever on devices with lower RAM. In this thesis we present the compilation of the QLever code using Emscripten [4], a complete compiler toolchain for C/C++ code. We will use Conan, a package and dependency manager for C/C++ libraries, in order to tackle the difficulties that arise when we port C/C++ code into a different programming language. We will then use Emscripten bindings to declare the C/C++ classes that we want to instantiate in JavaScript. Emscripten emits WebAssembly and JavaScript code. WebAssembly is a low-level code format which is designed for efficient code execution in the web. The emitted JavaScript code acts as an interface between the C/C++ world and JavaScript. We will use that interface in order to load and query prebuilt indexes in a browser.

1 Introduction

Before we dive deeper into the idea behind Emscripten and WebAssembly, we will first give a brief reminder how C/C++ source code is transformed into executable machine code.

1.1 Compiling C/C++ code

The process of taking C source code (and C++ code with minor differences) and turning it into executable machine code can be split into multiple phases :

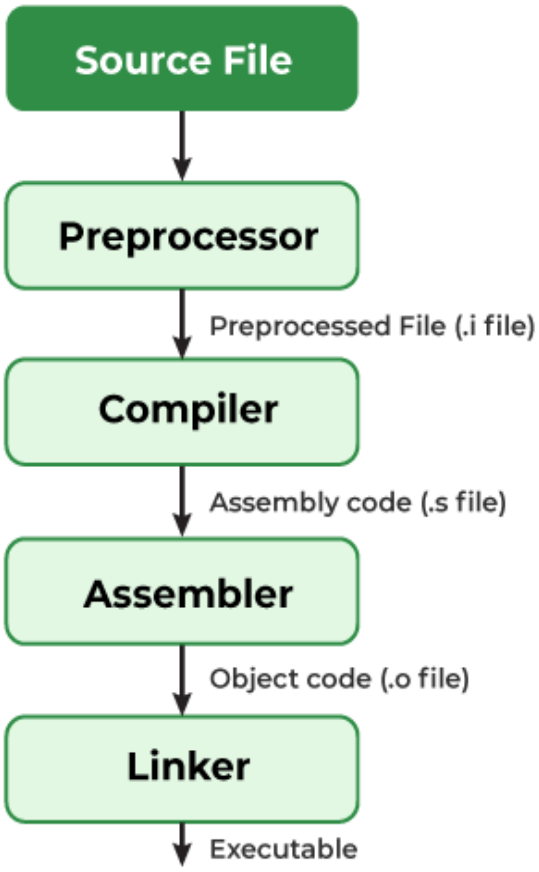

Figure 1: Compilation in C (Source: geeksforgeeks)

1. The source file gets preprocessed.
    - Comments are removed.
    - Macros are expanded.
    - The content of included files is copied into the source code.
    - Conditional compilation occurs.
2. The preprocessed file gets compiled into assembly code.
3. The assembly code gets translated into machine code by an assembler. Machine code is a sequence of instructions that a processor can execute.
4. The compiler does the previous steps for multiple files. It then links the machine code that 'belongs together'. This can be a header file, where a function is declared, and a c/cpp file, where the corresponding function is implemented. The compiler also links static libraries in this phase.

Clang is one compiler that can be used to transform C/C++ source code to machine code.

1.2 Emscripten and the WebAssembly code format

Emscripten is a complete compiler toolchain that can port C/C++ code to WebAssembly. It uses clang in order to produce LLVM (Low Level Virtual Machine) assembly code. Emscripten then takes that LLVM assembly code and produces JavaScript and WebAssembly code from it. (It is therefore able to port other languages to WebAssembly provided they are in LLVM or can be compiled into LLVM.) In order to do the latter it needs WebAssembly libraries. One challenge of this project revolves around the integration of the WebAssembly libraries in the build process. The WebAssembly libraries are located in a different directory of the operating system than their C/C++ pendants. Every tool of the build process of QLever needs to know where its required resources are located. In the next section we will introduce another tool that is already part of the build process of QLever.

1.3 Make, CMake and external libraries

Another important tool of the QLever build process is CMake. Compiling a small C/C++ project can usually be done with a couple of commands from the terminal.

Whereas each command represents a step mentioned in 1.1. However, when a project grows in size, it can be convenient to split the project into multiple directories to keep a good overview. Each directory could represent a group of code that belongs together. But then each group of code requires different compile commands. In that case it is convenient to use something like *make* with a Makefile. A Makefile contains information such like the compile commands for the files and their dependencies. A Makefile can also be used to bundle multiple compile commands to one single command, like *make compile*. There is another advantage of using a Makefile with *make*. Let's assume we changed something in one of the files. Now we want to build the project again, but we don't want to recompile the complete project. The Makefile knows all the dependencies and therefore only recompiles the files that are influenced by the changes. This saves a lot of time in development. Furthermore, if a project grows into an even bigger size, it can be convenient to write multiple Makefiles in order to organize the project. CMake offers a comfortable way to manage these Makefiles by automatically generating them instead of having to write and manage them individually. We can do that by writing multiple so called *CMakeLists.txt*. One CMakeLists.txt file acts as the main file. It contains all the information and settings for the complete project, such as compiler settings, build type, required libraries, the main executable targets and many more. The other CMakeLists.txt, one for each subdirectory of the project, contain a collection of commands where we can simply declare targets and libraries with *add_executable()* and *add_library()* commands.
Going back to the main CMakeLists.txt, one way how CMake gathers the required libraries is through the *find_package()* command. Let's look at the following line from the main CMakeLists.txt of QLever:

find package(ICU 60 REQUIRED COMPONENTS uc i18n)

With this command we tell CMake the following:

- We need the library ICU for our project.
- The version of ICU needs to be atleast 60.0.
- We need the additional components uc and i18n from ICU.

2 The main problem and possible solutions

We tried to compile QLever with Emscripten and used Emscriptens integration of CMake in order to do so. Unfortunately, the following problem occured:
The *find_package* command of CMake wasn't able to find the libraries ICU and Boost. As stated before in the linking phase of building the machine code, the build now additionaly requires WebAssembly libraries as well. However, the WebAssembly libraries are stored in a different directory of the operating system than their C/C++ pendants. This caused CMake not to find the required libraries, even though Emscripten has its own integration with CMake. We considered the following two approaches in order to solve this issue.

1. Use Conan, a package and dependency tool that is able to create an enviroment where the files that CMake and Emscripten require are available to use.

2. Manage the location and the paths of the required libraries manually. We would write some CMake code in order to list the locations CMake has to look into to pick up the libraries.

2.1 Choosing a solution

The advantage of the first approach is that Conan does already have integrations for CMake and Emscripten which ease the compatibility of these three tools. Conan is also already being used in the MacOS build for QLever. We know therefore that atleast CMake and Conan will most likely work together when building the QLever code. The disadvantage of using Conan compared to the other approach is that we would additionaly have to look into Conan and its integrations for CMake and Emscripten. This would require a lot more work to do (compared to the second approach) and it could still be possible that the conflict between CMake and Emscripten is not resolved. We also considered a third aspect which significantly influenced our decision. Let's assume we decided for the second approach. We would have looked up the locations of the required WebAssembly libraries and written some CMake code in order to redirect it to the right location. However, even if this approach worked out, it would come with different problems. First, maintaining the software: Let's assume QLever gets some additional functionality which requires new libraries or persisting dependencies change. What if again CMake has difficulties finding the libraries? We'd have to write CMake code once more in order to fix the problem. Second, different operating systems: We want to build QLever on an ubuntu distribution with version 24.04. What if someone else wants to build QLever with Emscripten on a different version? Or what if someone wants to build QLever on a different operating system, e.g. macOS? It's quite likely that these alternative builds would also suffer from the same problem, especially the latter one. These would require additional efforts when the setting changes. As a conclusion, we think that even if the second approach would be the simpler short-term solution with less effort, it would be a less elegant, not very sustainable and very hard-coded solution susceptible to changes. Therefore we ultimately decided to give Conan a try.

In the next section we will give a brief introduction to Conan and elaborate on our efforts in order to connect the tools successfully.

3 Conan package manager

3.1 Libraries and dependencies

Conan is an open source package and dependency managament tool for C/C++ code. Let's say u are using OpenSSL, a software toolkit for the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols. OpenSSL itself depends on Zlib, a sofware library for data compression. When managing a project with Conan and OpenSSL is rquested with CMake or any other build tool, it is not neccessary to keep track of OpenSSL's dependencies. In this case Conan identifies OpenSSL as part of the project and also gets the library Zlib. Whereas 'get' means it downloads the libraries from conan remote center, a repository with many open source tools and libraries already built as binaries for different operating systems.

3.2 Conan profiles

Conan uses profiles in order to specify the settings of the build enviroment. This includes things like the operating system, the compiler and the compiler version that will be used in the build process. It is also the place where we declare the build type (e.g. Release or Debug), additional tools that need to be installed (such as Emscripten or GoogleTest) and flags that can be passed on a specific part of the build or globally.
Conan can create a basic build profile very quickly with a simple:

conan profile detect

This generates a basic build profile named default. On an ubuntu 24.04 distribution, the generated build profile might look like this:

```
1  [settings]
2  arch=x86_64
3  build_type=Release
4  compiler=gcc
5  compiler.cppstd=gnu17
6  compiler.libcxx=libstdc++11
7  compiler.version=13
8  os=Linux
9
```

Figure 2: Default build profile generated with 'conan profile detect' (Source: own)

3.3 Cross-building

When compiling the QLever project to executable machine code with a compiler like gcc or clang, the default build profile is sufficient when using Conan as the package manager. Our build however is a bit more complex. We are using Emscripten as the compiler. Emscripten takes C/C++ code as input, compiles it with clang to machine code, and then produces JavaScript and WebAssembly code as output. This is called 'cross-building'. When cross-building with Conan, we need another profile in addition to the default build profile.

3.4 The host profile

This second profile is called a host profile. This host profile will contain all the neccessary settings for managing components of libraries. The host profile acts as an **interface between Conan and Emscripten**. Various Emscripten compile settings can be set here. E.g. we can choose if the WebAssembly code should be in 32- or 64-bit. We chose the 64-bit format for this thesis because a lot of pointers in the QLever code use the 64-bit format. This saves us some potential errors in the build process and when testing the output.
These are the basic settings for the host profile that we used for our build:

```
[settings]
arch=wasm64
build_type=Release
compiler=emcc
compiler.cppstd=20
compiler.libcxx=libc++
# emsdk 3.1.73 is the latest version provided by the Conan package manager.
compiler.version=3.1.73
os=Emscripten
compiler.threads=posix

[tool_requires]
emsdk/3.1.73

```

Figure 3: Basic settings for our host profile (Source: own)

3.4.1 Software libraries and their components

Many software libraries have components. Components of libraries are additional libraries that can be used together with the base library. They can be thought of like add-ons for the base library that contain optional functionality. By default, Conan has a specific setting for each of the libraries it offers via conan center. This means, that it takes some components by default and leaves others out. Conan offers an easy way to change the components setting for a library.
Let's take QLever for example. QLever uses the library ICU, a set of widely used C/C++ and Java libraries providing Unicode and Globalization support for software applications. QLever also requires the components 'uc' and 'i18n' of ICU. We can see here that it takes the components 'uc', 'i18n' and 'icuio' by default. However, we don't need 'icuio' for QLever. We can tell Conan not to take 'icuio' by simply adding the following line in the [**options**] category of our host profile:

```
14
15  [options]
16  icu/*:with_icuio=False
17
18  boost/*:without_atomic=True
19  boost/*:without_charconv=True
```

Figure 4: Disabling components in the [options] menu (Source: own)

The * means that we apply this setting to all versions of the library ICU that come with Conan. In addition to ICU, we remove 24 unneccessary components of the library Boost with a slightly different syntax. Removing these components is not neccessary but we also don't want to bloat our output with code that we don't intend to use.

3.4.2 Emscriptens compiler settings

Lastly, Emscripten offers a way to customize the compile settings for the build. We can write these settings into the [**conf**] section of the host profile and Conan will

propagate these flags to all compile commands from Emscripten. It is also the place where we can specify that we also use ninja to speed up the build.
We need to tell Conan which libraries to download from conan center in order to provide them to CMake. We can do that with a conanfile.txt.

3.5 Conanfile.txt

The conanfile.txt is another important aspect of the build using Conan. As the ending of the file name suggests it is a simple text file. The developer team of Conan recommends using a conanfile.py for more complex build structures (for example, if you need multiple versions of the same library) because it offers more ways to customize the build process. Since we are only requesting 4 libraries, a conanfile.txt is sufficient. Besides that, most other dependencies of QLever are integrated using fetchcontent from CMake.
The conanfile.txt has a [requires] section. This requires section acts as an **interface between Conan and CMake**. All the neccessary libraries that are requested with the find_package(...) command in the CMakeLists.txt can be listed here. When running the conan install command, Conan will then download all the listed libraries and their respective dependencies as prebuild binaries from conan center. We used the conanfile.txt that was already being used for the macOS build of QLever because we have the same dependencies:

```
1  [requires]
2  boost/1.81.0
3  icu/76.1
4  openssl/3.1.1
5  zstd/1.5.5
6
7  [generators]
8  CMakeDeps
9  CMakeToolchain
10 VirtualRunEnv
11
```

Figure 5: The conanfile.txt (Source: own)

In the next section we will elaborate a problem that occured regarding Emscripten.

4 Choosing the emsdk

Conan provides two options regarding the emsdk (Emscripten software development kit).
We can either use the emsdk from Conan. Or we can use a local installation of the emsdk. The configuration with the emsdk from Conan is immediately compatible. But it comes with a small problem. The newest version of the emsdk from Conan is only 3.1.73. The newest version of the emsdk at the time of writing this thesis is about 5.0.0. That is quite a difference. For this reason we initially wanted to use a local and up-to-date version instead. However, because of similar issues like the ones in section 2.1, we still decided to go with the old 3.1.73 version from Conan.
After deciding which emsdk to use, we were finally able to start building the QLever code. In the next section we will discuss the compile errors that occured and how we solved them.

5 Adjustments to the QLever source code

5.1 Macros

During the compilation, an issue regarding a macro occured. The macro wasn't expanded properly and this led to a wrong result.

5.1.1 Macro example

Macros are placeholders that the preprocessor substitutes with code in the preprocessing phase.
Here's a simple C++ program that explains what a macro is:

```
#include <iostream>

// Define a macro to calculate the sum of two numbers
#define SUM(x,y) (x + y)

int main() {
  int x = 7;
  int y = 8;
  int result = SUM(x,y); // Preprocessor expands this to (x + y)
  std::cout « „Sum of“ « x « „and“ « y « „equals“ « result;
  return 0;
}
```

Figure 6: Macro example (Source: own)

This outputs: Sum of 7 and 8 equals 15
In the main function, we have the equation:

$$\text{int result} = \text{SUM(x,y)}$$

As we briefly mentioned in 1.1, during the preprocessing phase, the preprocessor substitutes macros with another term. In this case, SUM(x,y) is getting replaced with (x + y), the term that we defined.

5.1.2 Defining a helper macro

Our problem occured with a macro being inside another macro. We will use the following example to explain the issue and our implementation in order to solve this issue. In this example, the macro TEST1 represents the issue and macro TEST2 represents our solution.

```
30
31
32  #include <iostream>
33
34  #define AD_STRINGIFY(x) #x
35  #define TEST(x) std::cout << (x) << std::endl;
36  #define TEST1(x) TEST(#x)
37  #define TEST2(x) TEST(AD_STRINGIFY(x))
38  #define MACRO_THINGY value
39
40
41  int main() {
42      TEST1(MACRO_THINGY); // -> prints MACRO_THINGY
43      TEST2(MACRO_THINGY); // -> prints value
44  }
45
46
```

Figure 7: Macro helper function (Source: own)

The problem that we ran into can be best described with TEST1. TEST1 consists of the following macros:

1. MACRO_THINGY, which should be replaced with value
2. TEST1(x) itself which should be replaced with TEST(#x)
3. #x is a macro that substitutes #x with the string of x
4. TEST(x) which should get replaced with the term that prints the string

Like the comment in main behind TEST1 suggests, the evaluation of these nested macros returns the name of the macro that we wanted to replace, not the actual value. This is due to the order of macro expansions. The expansion of macro #x (3.) occurs before the expansion of MACRO_THINGY (1.). In order to solve this issue, we defined an additional macro AD_STRINGIFY(x). This new macro replaces the input x with the macro #x which then replaces x with the string of x. This new macro prevents the # macro from being expanded before MACRO_THINGY is getting expanded. It basically delays the turn when the expansion of #x occurs. This is best described by TEST2. The main difference to TEST1 lies in the expansion of TEST2(x) to TEST(AD_STRINGIFY(x)). This way the substitution of #x to the string of x can not happen yet. This leads to MACRO_THINGY being replaced with value before the stringification happens. That gives us the desired result.

5.1.3 EXPECT_DEATH and ASSERT_DEATH

Multiple files issued errors during the compilation due to the definition of the macros EXPECT_DEATH and ASSERT_DEATH not being detected. We were able to fix this issue by replacing them with EXPECT_DEATH_IF_SUPPORTED and ASSERT_DEATH_IF_SUPPORTED respectively.
`https://google.github.io/googletest/reference/assertions.html`

5.2 The build commands

In this section we will elaborate on the commands in order to build the QLever source code with CMake, Conan and Emscripten. First, we need to get all the libraries for CMake and Emscripten, and Emscripten itself. We do this with Conan by running the following conan install command:

conan install . -pr:b default -pr:h conanprofiles/emscripten.profile -s build_type=Release -b missing -of build

We give Conan the build profile and the host profile that we set up precisely for this build. We also tell Conan that we want the Release versions of the libraries. It's important that the Conanfile.txt is located in the same directory where this command is executed, because all the required libraries are listed there. Otherwise the dot needs to be changed to the path to the Conanfile. This command also creates a folder with the name build. Besides the libraries, this directory will also contain external repositories that we requested through the fetchcontent_Declare() command from CMake (things like the googletest framework). We then move into the build directory and run the following cmake command:

cmake -S .. -B . **-DCMAKE_TOOLCHAIN_FILE=conan_toolchain.cmake** -DCMAKE_BUILD_TYPE=Release ... -GNinja

The important part is the specification of the **conan_toolchain.cmake** file. We already mentioned that the Conanfile.txt and the host profile act as interfaces between CMake, Conan and Emscripten. All these settings need to be translated into CMake language in order for CMake to properly integrate these specifications. This is exactly what the conan_toolchain.cmake file is there for. The other main difference are the additional flags that we give CMake. These enable the exclusion of the targets that we don't want to build which we explained in 5.2. After that, we finally run

cmake –build .

in order to start the build.

5.2.1 Note regarding memory consumption

We used a Linux distribution inside a Windows distribution through wsl (windows subsystem for linux). Our machine had 16GB of total RAM. This means that the wsl can only use 8GB of RAM by default. This led to the build freezing on targets that require more RAM. We solved this issue by creating a file named **.wslconfig** in which we wrote **MEMORY=12GB**.

6 Evaluation of the compiled output

After some minor adjustments to the source code of QLever we were able to successfully compile it to WebAssembly. The next question that arises is: How good does the output work? Using the googletest framework, the answer is a bit disappointing. More than 50 percent of the tests failed. From the tests that failed, more than 80 percent of them failed due to comparison issues. It was mostly because 4 byte values where being compared to 8 byte values. We assume that with some minor adjustments, a large portion of the compiled output will work correctly afterwards. The fact that the code compiled successfully implies that it functions somewhat properly. But we need to also mention this:
When we were trying to compile the QLever code to WebAssembly, one of the compilation errors had to do with the Boost library. We were able to solve it by adding this line to the [**conf**] section of the host profile:

boost/*:tools.build:cxxflags=['-sMEMORY64=1']

Even though we set the flag globally, the issue was fixed only when we set the flag explicitly for boost as well. Another problem during the second part of this work was also solved similarly. We activated exceptions globally in the profile (-fexceptions):

```
44
45  [conf]
46  tools.cmake.cmaketoolchain:generator=Ninja
47  tools.build:exelinkflags=['-fexceptions', '-sALLOW_MEMORY_GROWTH
48  tools.build:sharedlinkflags=['-fexceptions', '-sALLOW_MEMORY_GRO
49  boost/*:tools.build:cxxflags=['-sMEMORY64=1']
50
```

Figure 8: global flags in the host profile (Source: own)

But the exceptions were not present for that target during execution on the web. Only when we also explicitly set them in CMake for the target were they working properly. This leads to the assumption that even more issues can be fixed by explicitly setting flags for targets. We were unfortunately not able to further analyze this issue.

7 Adjusting the code for the Web

7.1 Generate wasm and js for libraries

We are now able to create wasm and js files. For our next step we need the Qlever.cpp library. It contains all the neccessary classes and methods that we want to call from JavaScript. However, Emscripten only creates .wasm and .js files for C/C++ code that has a main function. In order to create the .wasm and .js for a library we need to take the Qlever.o file and rerun the command like it is described in the quick example here: Our approach is slightly different than described there because we are using CMake and Ninja.
First we have the normal qlever target with an added STATIC:

```
24
25  add_library(qlever STATIC Qlever.cpp)
26  target_compile_options(qlever PRIVATE -fexceptions)
27  qlever_target_link_libraries(qlever parser engine util index absl::strings)
28
```

Figure 9: STATIC qlever target (Source: own)

And then we add an additional target qleverwasm:

```
2
3
4   add_executable(qleverwasm dummy.cpp)
5
6   target_compile_options(qleverwasm PRIVATE -fexceptions)
7   set_target_properties(qleverwasm PROPERTIES
8       OUTPUT_NAME "multithreadoutput11"
9       SUFFIX ".js")
10
11  target_link_options(qleverwasm PRIVATE
12      "-sMODULARIZE=1" "-sEXPORT_NAME=MyModule" "-sENVIRONMENT=web,worker" "-sEXPORT_ES6=0"
13      "-sALLOW_MEMORY_GROWTH=0" "-sUSE_PTHREADS=1" "-sSAFE_HEAP=0" "-sDEFAULT_PTHREAD_STACK_SIZE=16777216"
14      "-sPTHREAD_POOL_SIZE=6" "-sPROXY_TO_PTHREAD=1" "-sPTHREADS_DEBUG=1" "-sASSERTIONS=2"
15      "-sDISABLE_EXCEPTION_CATCHING=0" "-sEXPORTED_RUNTIME_METHODS=['FS']" "-fexceptions"
16      "--no-entry"
17      )
18
19  target_link_libraries(qleverwasm PRIVATE embind "-Wl,--whole-archive" qlever "-Wl,--no-whole-archive")
20
21
22
```

Figure 10: (Source: own)

The only relevant link option for now is the last one: **”–no-entry”** We need it for source files without a main function.

For the commands: We do **conan install** and generate the build files with cmake like we usually do. And then, above the build directory, we run

emmake ninja -C build qleverwasm

This will build only what is needed for qlever and qleverwasm (which is still almost the full build).
This will emit the qlever library as a JavaScript file and a wasm module.

7.2 Writing some glue code

We now have the JavaScript file with the wasm module. However, we are still not able to create instances of the objects that we want to use. We need some kind of glue code which translates the structures into JavaScript language. Emscripten provides two options in order to achieve this.

7.2.1 Web IDL

Web IDL is an interface definition language that can be used to describe interfaces that are intended to be implemented in web browsers. See here: Some example code can be seen here We would write all the classes, methods and attributes in IDL language. However, there are multiple problems regarding IDL. First, IDL doesn't support enum classes introduced in C++ 11 standard. It supports only enums, the older variant. Second, IDL doesn't support C++ standard templates like std::vector or std::optional. Which is bad, because QLever uses them often. A possible workaround would be to implement the templates from scratch in order to satisfy the compiler and then use something like typedef in order to create the objects. But that would be an unreasonable amount of effort and we don't know for sure if the outcome would work. That's why we decided to go with Embind.

7.2.2 Embind

Another way to write the glue code is to use the EMSCRIPTEN_BINDINGS block. Similar to Web IDL, we write the code, that we want to use, in an intermediate language. To give an example, let's look at a simplified version of the struct CommonConfig:

```
namespace qlever {

// The common configuration shared by the index building and query execution.
struct CommonConfig {
  // The basename of all files that QLever will write as part of the index
  // building.
  std::string baseName_;
}
}


using namespace emscripten;

// Binding code
EMSCRIPTEN_BINDINGS(module) {

  class_<qlever::CommonConfig>("CommonConfig")
    .constructor<>()
    .property("baseName", &qlever::CommonConfig::baseName_);
}




// In JavaScript
var config = new MyModule["CommonConfig"]();
config.baseName = 'hello';
```

Figure 11: Simplified struct from Qlever.cpp (Source: own)

With this simple logic we can treat the Qlever classes from C++ as they were normal JavaScript objects. And even better, Embind supports std::vector (register_vector) and std::optional (register_optional), which WebIDL does not. Now that we are able to instantiate the Qlever code in JavaScript, let's talk briefly about N-Triples and RDF. We want to take a N-Triple file (or any other RDF format) and build an index of that file. A short and simplified explanation: A N-Triple file is a file which contains a collection of triples divided by a dot. The entries of the triples are ordered in the form: subject, predicate and object. Each entry is uniqely identified by e.g. an URL. A subject can be related (predicate) to other subjects. The predicate defines how they are related to each other. A subjects attributes can further be described by its relations to objects. Building an index of such a file means extracting these relations and creating a model of that information which can then be queried using a query language like SPARQL [1]. This way we can retrieve information from that model. (End of short explanation)
For the implementation: We will basically reimplement the code from LibQLeverExample.cpp in JavaScript. First we need a configuration. This configuration gets one (or multiple) file(s), a filetype and a name for the index. This configuration will be given to the buildIndex method in order to produce index files. Then the

configuration gets passed to an engine configurator object. After that the engine configurator itself will be passed to a *qlever* instance. On that qlever instance we can finally run a query. Here is a complete overview of our bindings:

```
230
231  using namespace emscripten;
232
233  // Binding code
234  EMSCRIPTEN_BINDINGS(module) {
235    enum_<qlever::Filetype>("Filetype")
236      .value("Turtle", qlever::Filetype::Turtle)
237      .value("NQuad", qlever::Filetype::NQuad);
238
239    class_<qlever::InputFileSpecification>("InputFileSpecification")
240      .constructor<>()
241      .property("filename", &qlever::InputFileSpecification::filename_)
242      .property("filetype", &qlever::InputFileSpecification::filetype_)
243      .property("defaultGraph", &qlever::InputFileSpecification::defaultGraph_);
244    register_optional<std::string>();
245
246    class_<qlever::CommonConfig>("CommonConfig")
247      .constructor<>()
248      .property("baseName", &qlever::CommonConfig::baseName_);
249
250    class_<qlever::IndexBuilderConfig, base<qlever::CommonConfig>>("IndexBuilderConfig")
251      .constructor<>()
252      .function("validate", &qlever::IndexBuilderConfig::validate)
253      .property("inputFiles", &qlever::IndexBuilderConfig::inputFiles_);
254    register_vector<qlever::InputFileSpecification>("InputFileSpecificationVector");
255
256    class_<qlever::EngineConfig>("EngineConfig")
257      .constructor<qlever::IndexBuilderConfig>();
258
259    class_<qlever::Qlever>("Qlever")
260      .constructor<const qlever::EngineConfig&>()
261      .class_function("buildIndex", &qlever::Qlever::buildIndex)
262      .function("query", select_overload<std::string(std::string, ad_utility::MediaType) const>(&qlever::Qlever::query));
263
264    enum_<ad_utility::MediaType>("MediaType")
265    .value("textPlain", ad_utility::MediaType::textPlain)
266    .value("json", ad_utility::MediaType::json)
267    .value("sparqlJson", ad_utility::MediaType::sparqlJson)
268    .value("sparqlXml", ad_utility::MediaType::sparqlXml)
269    .value("qleverJson", ad_utility::MediaType::qleverJson)
270    .value("tsv", ad_utility::MediaType::tsv)
271    .value("csv", ad_utility::MediaType::csv)
272    .value("csv", ad_utility::MediaType::csv)
273    ;
274  }
```

Figure 12: Complete set of Emscripten bindings in Qlever.cpp (Source: own)

And this is the logic in JavaScript:

```
// create the configuration and an input file type for it
const config = new MyModule["IndexBuilderConfig"]();
const vec = new MyModule["InputFileSpecificationVector"]();
const fileSpec = new MyModule["InputFileSpecification"]();

// assign a file to the configuration which we want to build an index from
fileSpec.filename = "housemd.nq";
fileSpec.filetype = MyModule.Filetype.NQuad;
config.baseName = "HouseIndex";

vec.push_back(fileSpec);
config.inputFiles = vec;

// create placeholders for the class instantiations
var engineConfig;
var qlever;
// This is a simple query that selects all persons from the data model
var basicQuery = 'PREFIX schema: <http://schema.org/> SELECT *{ ?person a schema:Person}';
var queryResult;

// build an index from config
MyModule.Qlever.buildIndex(config);

// create an engine config with the modified config
engineConfig = new MyModule.EngineConfig(config);

// assign the engine to qlever and run the query
qlever = new MyModule.Qlever(engineConfig);
queryResult = qlever.query(basicQuery, MyModule.MediaType.qleverJson)
```

Figure 13: From config to query in JavaScript (Source: own)

7.2.3 Virtual file system

WebAssembly does not use a physical filesystem. It simulates a virtual filesystem instead. Therefore we need to fetch the files that we want to use in our code and write them into the virtual filesystem. We can achieve that with the following logic (using housemd.nq as en example):

```
async function fetch_house() {
    // 🔹 Fetch RDF file from public/
    const response = await fetch("/housemd.nq");
    const buffer = await response.arrayBuffer();
    if (!response.ok) {
        throw new Error("Failed to fetch housemd.nq");
    }
    // 🔹 Write housemd dataset into Emscripten virtual FS
    MyModule.FS.writeFile(
        "housemd.nq",
        new Uint8Array(buffer)
    );
}
```

Figure 14: Write a file into Emscriptens virtual filesystem (Source: own)

8 Executing the code on the web

We tried to run the code with a small dataset with 8 subjects. Unfortunately, the buildIndex method was not finishing properly. The program just froze and we could not find the cause of that. That's why we decided to prebuilt the index and fetch all the created files into the filesystem. This worked out flawlessly for the small dataset. We ran a simple query that returns all 8 subjects. This took under 10ms on average (done 3 times). Then we ran a different query that returns all pairs of subjects that know each other and that removes duplicates. This took roughly 160ms on average (done 3 times). We then ran the same configuration with the native build for comparison. This took only 30ms on average (done 3 times). This implies that the port to WebAssembly takes quite a hit in performance.
We also tried a prebuilt index of the olympics dataset, which has a size of 330MB. The query did not finish. The UI froze like when we tried to build the index.

9 Evaluation

We have proven that it is possible to run QLever on the web using Emscripten with the Emscripten bindings if we prebuilt the index and fetch all the produced files. We have also seen that the performance on the web is acceptable but significantly worse compared to the performance on the native build. That is even on very small datasets. That is however somewhat expected because WebAssembly does not use a physical file system. We also observed that, even on these small datasets, QLever uses almost 10 GB of memory for the execution of the query. As a final conclusion, even though the performance on the web is significantly lower compared to the native build, the possibility to port the complete QLever code to WebAssembly with some minor additions in the code is still a respectable option.

10 Related work

10.1 Ammo.js

Ammo.js is the direct port of the Bullet physics engine using Emscripten. It's C++ source code consists of basic variable types which enables the possibility to bind the objects using WebIDL without any adjustments of the source code.

10.2 d3wasm

d3wasm is a port of the Doom 3 video game to WebAssembly using Emscripten.

10.3 Qt WebAssembly

Qt is a multi cross-platform framework which lets you run C++ and Python code on many different devices and operating systems. Qt WebAssembly is the port of Qt with Emscripten. With Qt WebAssembly you can run applications on the web that don't require full access to host device capabilities.

11 Future work

There are a couple of things that can be done to further improve this project.

11.1 Using a newer version of the emsdk

As we already mentioned in section 4, using a local installation of the emsdk with Conan has the issue that CMake's find_package() command isn't able to locate the correct directories for the WebAssembly libraries. Later on during our work we discovered that we can actually run the find_package command in debug mode. Then we discovered, that the collection of paths was not including the build directory. However that is the place (by default) where Conan stores the files that CMake uses in order to generate the Makefiles. A simple solution would therefore be to tell CMake the path to the build folder. We already tested this and it worked. Earlier in section 2 where we discussed a very similar issue, we said that hardcoding paths is not very sustainable. In this case however this is totally fine because the build folder contains exactly the files that are needed for the build. That's why we think this is a solid solution. But we also observed that the newer versions of Emscripten are stricter regarding templates when we tried to compile the code with them.

11.2 Testing the compiled output

We already mentioned that we tested the compiled output using the googletest framework. A lot of the tests failed due to different byte formats being compared. We assume that the MEMORY64 flag is not being distributed to all targets. We already solved multiple issues by setting them explicitly.

11.3 buildIndex

It is desirable to build the index on the web instead of using prebuilt ones. We have already seen that queries can be executed on the web using small datasets. We are confident that it is possible to build an index for datasets of similar size.

12 Acknowledgements

I want to thank Prof. Dr. Hannah Bast for her excellent lectures which further motivated me to become a good teacher.
I want to thank Johannes Kalmbach for his help during the project.
I especially want to thank Robin Textor-Falconi for his continuous support during this project which significantly improved the quality of this work.

I want to thank my family for their immense support over the years which enabled me to pursue my bachelor's degree.
I especially want to thank my wife who has been the most important person in my life for more than 12 years and who has helped me become a far better version of myself.

13 Figure sources

1. Compilation process in C

References

[1] C. B. Aranda et al. , *"SPARQL 1.1 Overview",* Mar. 2013.

[2] H. Bast and B. Buchhold. "QLever: A Query Engine for Efficient SPARQL+Text Search," in: *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, in CIKM '17*. Singapore: Singapore: Association for Computing Machinery, 2017, pp. 647–656. https://doi.org/10.1145/3132847.3132921.

[3] Robin Textor-Falconi. "Lazy Evaluation of SPARQL Queries with Caching". MA thesis. University of Freiburg, 2024.

[4] Alon Zakai. "Emscripten: an LLVM-to-JavaScript compiler". In: *OOPSLA '11: Proc. of the ACM internat. conference companion on Object oriented programming systems languages and applications companion*. 2011, pp. 301–312. https://doi.org/10.1145/2048147.2048224.