Master's Thesis

Enabling Automated Setup and Evaluation of SPARQL Engines through Unified Benchmarking Infrastructure

Tanmay Garg

Matriculation number: 5453150

Supervisor and First reviewer

Prof. Dr. Hannah Bast

Second reviewer

Prof. Dr. Christian Schindelhauer

Albert-Ludwigs-University of Freiburg
Faculty of Engineering
Department of Computer Science
Chair for Algorithms and Data Structures

October 11th, 2025

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 hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Freiburg im Breisgau, 11.10.2025

Place, Date

Signature

Abstract

Knowledge graphs provide a structured way to represent the growing volume of web data by organizing information into entities and the relationships between them. They are typically expressed in the Resource Description Framework (RDF), where knowledge is stored as subject-predicate-object triples that capture facts about the world. SPARQL engines enable efficient management and querying of these graphs, but they vary widely in their architecture, storage models, indexing strategies, and query optimization. This variation makes it essential to identify engines that deliver both strong current performance and effective scalability as data and query demands grow.

In this thesis, we present a unified and extensible framework for benchmarking and evaluating seven SPARQL engines: QLever, Virtuoso, Blazegraph, Apache Jena, Oxigraph, GraphDB, and MillenniumDB. The framework allows setup of SPARQL endpoints for all supported engines using uniform, simple commands that work with both containerized environments and pre-installed native binaries, without having to worry about internal engine-specific details. A single configuration file, the Qleverfile, ensures consistent and reproducible setup across engines and datasets. The framework supports execution of benchmark queries with structured output that records runtimes, result sizes, and benchmark metadata. We also develop an interactive web application that automatically consumes the benchmark results, enabling comparisons across engines at both aggregate and per-query levels.

We use the framework to evaluate the seven engines on various synthetic and real-world benchmarks at three scales: small (∼50M triples), medium (∼500M triples), and large (∼8B triples). The experiments show how the framework simplifies setup, ensures reproducibility, and provides actionable insights into performance, scalability, and correctness of SPARQL engines. All software and materials to reproduce the evaluation are publicly available at `https://github.com/ad-freiburg/sparql-engine-evaluation-tanmay`.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

The rapid growth of data on the web has created a strong need for structured ways to represent and query information. Knowledge graphs address this need through a graph-based data model designed to capture and convey real-world knowledge [1]. They organize data into entities, such as people, places, or concepts, and the relationships that link them. The standard for modeling knowledge graphs is the Resource Description Framework (RDF) [2]. RDF represents information as subject-predicate-object triples, where consistent identifiers are used for the same entities. The subject and object refer to entities (or resources), while the predicate specifies the relationship between them. For example, to represent that Lionel Messi is a footballer that plays for Argentina as `<subject>` - `<predicate>` - `<object>` triples:

`<Lionel_Messi>` - `<has_profession>` - `<Footballer>`
`<Lionel_Messi>` - `<player_of>` - `<Argentina_National_Team>`

Now, we can add more triples:

`<Lionel_Messi>` - `<player_of>` - `<FC_Barcelona>`
`<FC_Barcelona>` - `<plays_in_league>` - `<La_Liga>`

When combined, these triples create a small knowledge graph shown in Figure 1.

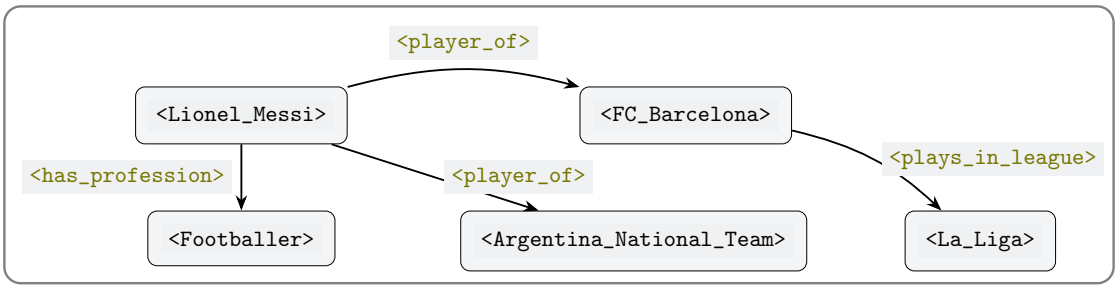

Figure 1: A toy knowledge graph illustrating RDF triples about Lionel Messi, his profession, national team, club, and related league.

To query and extract information from such graphs, the World Wide Web Consortium (W3C) standardized the SPARQL query language [3]. SPARQL allows users to specify patterns of triples with variables and retrieve the matching results (Listing 1). In practice, SPARQL supports everything from simple lookups to complex graph traversals.

Listing 1 SPARQL query to get all football players that have played for both Argentina National Team and FC Barcelona, with Messi being one of the results.

```
SELECT ?player
WHERE {
  ?player <has_profession> <Footballer> .
  ?player <player_of> <Argentina_National_Team> .
  ?player <player_of> <FC_Barcelona> .
}
```

Knowledge graphs have grown into one of the most important methods for storing structured data at web scale. The largest publicly available knowledge graphs with a coherent schema such as Yago-4 [4], Wikidata [5] and UniProt [6] can easily have billions of these triples. The need to process complex SPARQL queries over such massive datasets has led to the proliferation of a wide variety of SPARQL engines [7]. A SPARQL engine (also called RDF Triplestore) is simply a system that supports the storing and indexing of RDF data and processing of SPARQL queries on it. Indexing here refers to building efficient data structures that allow the engine to quickly find relevant triples without scanning the entire dataset.

However, not all engines are built in the same way. They differ in their internal architecture, how they store data, create indexes, and optimize query execution[1]. These design choices have a major impact on performance and scalability. For instance, Oxigraph [8] stands out for its simplicity and very fast data loading, but it produces large indexes and offers little query optimization, which can slow down complex queries. Blazegraph [9], by contrast, can handle large datasets and heavy workloads, but it is less user-friendly and often requires careful configuration to perform well. Virtuoso [10] takes yet another approach, offering powerful query optimization, but scaling it to massive knowledge graphs often demands substantial hardware resources. These differences show that no single engine performs best in every situation. The optimal choice depends on the dataset size, query workload, and scalability needs.

[1]Query optimization refers to selecting the most efficient way to execute a query, for example by reordering query steps or making use of indexes to reduce unnecessary work.

Because of these performance and scalability differences, it is important to evaluate multiple SPARQL engines for each use case.

1.1 Motivation

Consider the case of a research group or an engineering team at a company that needs to select a SPARQL engine for querying a large knowledge graph. Their goal is to identify an engine that offers good query performance today and can continue to meet future demands as the knowledge graph and query workload grows. Or, consider the case of researchers/developers who work on SPARQL engines and must test their systems against existing ones and against earlier versions of their own work. For them, the goal is to provide competitive evaluation, and a way to test whether modifications improve or harm performance and scalability. In both scenarios, the evaluation process and the challenges faced are broadly the same. It involves several steps that together determine how well SPARQL engines perform and scale in practice.

A natural first step would be to consult the academic literature, where several benchmark studies have already compared the key SPARQL engines and published the results. Benchmarks in this context are programs designed to assess the speed and efficiency of engines when executing a range of SPARQL queries. A recent survey lists several such benchmarks [11], and these will be discussed in more detail in Chapter 2. Although these benchmarks provide valuable insights, relying solely on them is problematic. The results of these benchmarks often differ due to heterogeneity of datasets, differences in benchmark design and query workloads, and variations in the underlying hardware and software environments. Moreover, published benchmarks quickly become outdated as newer versions of engines are released with feature changes and optimizations.

For this reason, users must attempt to reproduce the benchmark results themselves. This typically involves downloading, installing, and configuring multiple SPARQL engines. However, this process is rarely straightforward. Some engines provide binary distributions, while others must be compiled from source. Each engine comes with its own configuration parameters and quirks, many of which significantly affect indexing and query performance. For example, Blazegraph and Virtuoso rely on custom configuration files that, in their default form, yield very poor indexing and querying performance once datasets exceed a few hundred million triples. Achieving competitive performance on medium and large-scale knowledge graphs requires extensive manual tuning of these configuration files. Since such details are often poorly documented, users are forced to rely on trial-and-error, prior

expertise, or community knowledge. Ensuring a fair and comparable setup across engines therefore requires considerable expertise and effort.

Once the engines have been set up and configured, the next challenge is running the benchmarks in a standardized and reproducible manner. Existing benchmarks differ in many ways when it comes to ease of use. Some benchmarks provide full frameworks with a way to query and measure performance, while others consist only of raw query files, leaving the burden of implementation to the user. To obtain a reliable picture of performance and scalability, several benchmarks at various scales need to be run across multiple engines. But coordinating these different benchmarks across multiple engines is not straightforward. It requires careful setup of datasets, query execution, and measurement for each case.

Even when benchmarks are successfully executed, analyzing the results presents additional difficulties. Existing benchmark tools output aggregate performance metrics and per-query results as static tables. These formats require further manual processing to extract insights. There is limited support for side-by-side comparison of engines across different benchmarks and queries. Moreover, testing correctness is just as important as speed. An engine might be significantly faster than other engines, but the result size might be different for the same query. Identifying such discrepancies and determining whether an engine produces correct results requires additional effort, which is rarely supported by existing benchmarking frameworks.

In summary, whether the goal is to select the best engine for a use case or to test an engine under development, the evaluation process faces the same challenges. These challenges show the need for better tools and methods that make SPARQL benchmarking more reproducible, reliable, and easier to manage.

1.2 Contributions

The challenges of evaluating SPARQL engines highlight the need for tools that simplify setup, ensure reproducibility, and support meaningful comparison across systems. This thesis addresses these gaps by presenting a unified and extensible framework for the setup and evaluation of SPARQL engines with minimal manual intervention.

The work builds on QLever-control [12], a Python package that provides a straightforward command-line interface for automated setup and endpoint management of QLever. QLever [13] is a high-performance SPARQL engine developed at the University of Freiburg for efficient querying of massive knowledge graphs. Both QLever and Qlever-control are free, and open-source software.

This thesis extends QLever-control into a unified and extensible benchmarking framework for multiple SPARQL engines. The main contributions are as follows:

Automatic SPARQL Endpoint Creation Across Engines

The framework generalizes the workflow of QLever-control to support seven SPARQL engines: QLever [13], Virtuoso [10], MillenniumDB [14], GraphDB [15], Blazegraph [9], Apache Jena [16], and Oxigraph [8]. Uniform and automated setup for each SPARQL engine is provided through a set of engine-specific wrapper scripts included in the QLever-control package, denoted as `<qengine>` [2]. Each engine-specific wrapper script automates dataset retrieval, indexing, and endpoint startup, allowing users to set up SPARQL engines without needing to know the internal commands or configuration details of each engine (Listing 2). The configuration for each engine follows the same unified format defined by the **Qleverfile**, ensuring consistency and reproducibility across engines and datasets. The framework supports container-based setup using Docker or Podman, as well as using pre-installed native binaries for each engine. The design is extensible, allowing new SPARQL engines to be added with minimal effort.

Listing 2 Setting up and managing a SPARQL endpoint for all supported engines

```
# Generate engine-specific Qleverfile for the benchmark
<qengine> setup-config <benchmark>
# SPARQL endpoint setup
<qengine> get-data       # download the dataset
<qengine> index          # Build index data-structures for this dataset
<qengine> start          # Start the engine server using the index
<qengine> log            # tail server logs
<qengine> query          # run a SPARQL query against the endpoint
<qengine> status         # show all running endpoint instances
<qengine> stop           # stop a running endpoint
```

[2] `<qengine>` denotes the engine-specific wrapper scripts: `qlever`, `qvirtuoso`, `qmdb`, `qgraphdb`, `qblazegraph`, `qjena`, and `qoxigraph`.

Running Benchmark Queries

A new `benchmark-queries` command, integrated with the engine-specific wrapper scripts, is introduced to execute SPARQL queries from standardized YAML files. The YAML query format allows us to specify the benchmark name, description and scale, in addition to the queries. This information is used by the web application (see next contribution) to display benchmark-specific context and organize the results more effectively. The `benchmark-queries` command runs the input queries against the selected engine's SPARQL endpoint and produces a structured YAML results file containing runtimes, result sizes, timeouts, and other execution details. By default, it works seamlessly with all supported engines, but it can also target arbitrary SPARQL endpoints, enabling evaluation of engines not included in the framework. This combination of standardized input and structured output ensures consistent, reproducible, and easily analyzable benchmarking data across engines and datasets.

Interactive Evaluation Web Application

A single, non-engine-specific `serve-evaluation-app` command is introduced that automatically consumes the YAML benchmark results generated by `benchmark-queries` and serves the web application (web-app). The web-app provides an interactive interface for exploring benchmarking results instead of static tables. Users can quickly view aggregate performance metrics to compare overall performance across engines and benchmarks. In addition, the web-app supports side-by-side, per-query evaluation, showing runtimes and result sizes for detailed inspection of individual queries. This design allows both high-level and fine-grained analysis without manual processing of results.

Finally, we use the unified benchmarking framework to evaluate the seven supported SPARQL engines: QLever, Virtuoso, MillenniumDB, GraphDB, Blazegraph, Jena, and Oxigraph. The evaluation runs multiple synthetic and real-world benchmarks, comparing engine performance, scalability, as well as index build time and index size across three dataset scales: small (∼50 million triples), medium (∼500 million triples), and large (∼8 billion triples). This evaluation is not intended as a comprehensive performance study of the engines. Instead, it illustrates how the framework can execute a wide range of benchmarks at multiple scales, providing actionable insights into engine behavior; see the discussion in Section 5.2. The experiments also highlight the ease of setting up endpoints, running benchmarks, and analyzing results through the interactive evaluation web application. All materials needed to reproduce these evaluations are available at `https://github.com/ad-freiburg/sparql-engine-evaluation-tanmay`.

1.3 Limitations

Although this work presents a unified framework for evaluating SPARQL engines, it is important to recognize its limitations.

The framework currently supports the setup of only seven engines: QLever, Virtuoso, Blazegraph, Apache Jena Fuseki, Oxigraph, GraphDB, and MillenniumDB. Many other systems, including commercial and distributed SPARQL engines, are not yet supported. The task of adding support for new engines is simplified by the easy-to-extend nature of the framework, but still requires knowledge of the target engine's configuration and commands.

Another limitation lies in the engine setup process. The framework fully automates container-based setups for all supported engines, so that users do not have to manually download binaries or source code. However, the installation of native binaries is not automated and must be handled by the user. Once native binaries are available, indexing and startup can still be managed through the framework, but the installation itself is outside its scope.

The framework supports running any benchmark, provided that the queries work with the dataset. However, queries must first be converted into the standardized TSV or YAML format required by the framework. This preprocessing step is trivial, since the only information needed is a short description and the full SPARQL query text for each query. Still, it introduces a small amount of extra work for users who want to integrate new benchmarks.

This work focuses on sequential query performance, since it is the most fundamental and comparable aspect of SPARQL benchmarking. Other important factors such as reasoning, update handling, federation, concurrent query evaluation, and full SPARQL 1.1 compliance are not systematically evaluated. We consider this out of scope for now.

Finally, the framework provides structured outputs and visualization tools, but correctness checks are limited to comparing result sizes. Subtle semantic differences in query answers are not detected automatically. Such differences would require manual inspection to confirm full consistency across all engines.

1.4 Outline

The remainder of this thesis is organized as follows:

Chapter 2: **Related Work** - This chapter reviews existing literature on benchmarking approaches, and RDF query evaluation frameworks. It highlights key studies and identifies gaps that motivate the development of the proposed framework.

Chapter 3: **Building on a strong foundation: QLever-control** - This chapter explains why QLever-control serves as a good foundation for creating the unified, extensible benchmarking framework proposed in this thesis.

Chapter 4: **Approach and Implementation** - This chapter describes the architecture of the proposed framework, and the design choices that makes the framework easy-to-extend. It details how QLever-control was extended to support multiple SPARQL engines. It also explains the benchmark execution and evaluation web application in detail.

Chapter 5: **Evaluation** - This chapter presents experiments using several existing benchmarks at three different scales. It demonstrates how the framework allows performance and scalability comparison across all supported SPARQL engines and benchmarks, and provides analysis of the results.

Chapter 6: **Conclusion** - This chapter concludes the thesis, highlighting the contributions and significance of the work. It also discusses potential directions for future improvements and extensions to the benchmarking framework.

2 Related Work

In this chapter, we review existing SPARQL benchmarks and benchmarking frameworks for evaluating SPARQL engines. A benchmark typically combines three elements: an RDF dataset, a set of SPARQL queries, and performance metrics for evaluation. A variety of approaches exist, ranging from synthetic benchmarks that generate controlled datasets and queries, to real-world benchmarks that reflect authentic usage patterns. Then, there are frameworks and engines that provide the infrastructure for executing and managing these benchmarks. This chapter reviews the main work in each of these areas and discusses their strengths and limitations in the context of SPARQL engine evaluation.

2.1 Synthetic benchmarks

Synthetic benchmarks generate datasets and queries in a controlled way, often based on data generators and predefined query templates. They are widely used to test scalability because dataset size can be increased systematically, and experiments can be reproduced consistently across engines. Some well-known synthetic benchmarks are:

Berlin SPARQL Benchmark (BSBM) [17] models an e-commerce use case with products, offers, vendors, and consumers. BSBM generates synthetic product data along with queries for search, navigation, and comparison. Three query mixes of varying complexity are provided to measure engine performance against different use cases.

Lehigh University Benchmark (LUBM) [18] is based on a university domain ontology, and focuses on evaluating reasoning, query performance and scalability. It generates synthetic university datasets of arbitrary size, together with queries that test reasoning and data retrieval at scale.

Waterloo SPARQL Diversity Test Suite (Watdiv) [19] emphasizes query diversity. It generates scalable datasets and query workloads that vary in shape and selectivity, stressing different aspects of query evaluation. WatDiv is often used to study how engines behave under a wide range of query patterns.

SPARQL Performance Benchmark (SP^{2}Bench) [20] is based on the DBLP bibliographic dataset. It produces realistic bibliographic data and a diverse query workload that covers a variety of SPARQL operators and RDF access patterns. SP^{2}Bench aims at a comprehensive performance evaluation of engines by assessing the generality of optimization approaches implemented.

Together, these benchmarks have been widely adopted to evaluate SPARQL engines under controlled conditions. They highlight scalability and performance differences, but they may not fully reflect the complexity of real-world data and query workloads.

2.2 Real-world benchmarks

Real-world benchmarks don't need data-generators as they are based on a real-world dataset. The schemas of real-world datasets are often much more complex compared to synthetic datasets owing to their collaborative nature. Some well-known real-world benchmarks are:

Wikidata Graph Pattern Benchmark (WGPB) [21] is a benchmark that aims to stress complex basic graph patterns and worst-case-optimal join strategies on real-world Wikidata Knowledge graph. WGPB defines 17 abstract basic graph patterns and instantiates 50 queries per pattern using random walks in a filtered Wikidata-truthy graph.

WDBench [22] is also a Wikidata-based benchmark constructed from real query logs, with an emphasis on harder cases such as queries that time out on the public endpoint. WDBench focuses on query features that are common to SPARQL and graph databases: basic graph patterns, optional graph patterns, path patterns, navigational graph patterns.

FEASIBLE [23] is a feature-based benchmark generation framework designed to create customized SPARQL benchmarks from real query logs or synthetic query sets. It can generate benchmarks for any dataset that has available query logs. FEASIBLE also includes a query selection component that considers structural query features such as join vertices, triple pattern selectivity, and expected result size.

Mu-bench [24] is a microbenchmarking framework designed to generate customized SPARQL benchmarks from real-world query logs. It uses clustering algorithms to select diverse query subsets based on user-defined criteria for specific SPARQL features.

Sparqloscope [25] is a generic benchmark that automatically generates comprehensive benchmarks for any given RDF dataset. It produces around 100 carefully crafted queries

covering most SPARQL 1.1 features and various SPARQL functions for numerical values, strings, dates, language filters, etc. It aims to evaluate relevant features in isolation and as concisely as possible.

Real-world benchmarks complement the scalability benefits of synthetic ones by focusing on authenticity and reveal how engines behave under practical conditions.

2.3 Benchmarking frameworks and Query engines

There exists several benchmarking frameworks and query engines that provide an independent infrastructure for executing the benchmarks discussed above and collecting performance data. They are therefore complementary to benchmarks and help enable systematic evaluation. These include:

IGUANA [26] is a SPARQL benchmark execution framework which can measure the performance of engines during data loading, data updates as well as under different loads and parallel requests. It takes as input a benchmark, a dataset, and optionally update operations, and then evaluates the behavior of an engine on any given benchmark in a holistic way.

Comunica [27] is a modular SPARQL query engine designed for federated query evaluation. It can query across heterogeneous sources, including SPARQL endpoints, data dumps, and Triple Pattern Fragment (TPF) interfaces. Its modular design allows researchers to experiment with different query strategies while relying on a common framework.

These systems extend the scope of benchmarks by offering practical ways to execute them and to study query performance across diverse environments.

2.4 Discussion

Most benchmarks discussed in Section 2.1 and Section 2.2 provide only a set of predefined queries, leaving the execution and analysis of results to the user. This limitation can be partially addressed by benchmarking frameworks such as IGUANA or Comunica, which support the execution of queries for arbitrary benchmarks. However, even when using these frameworks or when benchmarks themselves support query execution, as in BSBM, LUBM, WGPB, and WDBench, the results are typically stored in static formats such as XML or CSV. These formats require significant additional processing to extract

meaningful insights or to perform side-by-side comparisons across engines. Furthermore, there is no unified mechanism to automatically compare performance across multiple benchmarks, making it difficult to identify patterns, trends, or relative strengths and weaknesses of different SPARQL engines across query types and workloads.

The unified framework developed in this thesis directly addresses these issues. It automates SPARQL engine setup using Docker or Podman, removing the need for users to download or compile engine-specific code. The `benchmark-queries` command then plays the role of a query engine by executing any benchmark workload, whether synthetic or real-world, against arbitrary SPARQL endpoints. The benchmark results are automatically consumed by the evaluation web application, which provides aggregate performance metrics and detailed per-query comparisons across benchmarks and engines.

This combination makes benchmarking reproducible, extensible, and much more accessible. In fact, Sparqloscope, one of the recent real-world benchmarks, relies on the `benchmark-queries` command and web application developed in this thesis to present its results, demonstrating the practical value and applicability of this work.

3 Building on a strong foundation: QLever-control

Building a unified framework for benchmarking multiple SPARQL engines is a significant challenge. The engines are fundamentally heterogeneous, each with distinct requirements for installation, configuration, and execution. Creating a framework to automate these diverse processes from scratch would be an immense and error-prone undertaking. Moreover, providing support for both native binaries and containerized environments, and implementing a benchmarking utility adds further layers of complexity and substantial development effort.

A more practical and effective solution is to build on a solid existing foundation. QLever and its companion tool, QLever-control, provide such a foundation. QLever-control is a Python package that automates endpoint-creation for QLever. With simple and easy-to-remember commands such as `setup-config`, `get-data`, `index`, and `start`, users can set up a complete SPARQL endpoint for a dataset with minimal effort, without worrying about low-level engine details. Each command in QLever-control is implemented as a separate Python class, which makes the system highly modular. This design allows commands to be extended through inheritance, enabling support for other engines with minimal code duplication. The framework supports both native binaries and containerized execution for QLever. It also provides the `example-queries` command that executes some predefined queries and prints query runtime statistics, which forms a natural starting point for a full-featured benchmarking utility.

By leveraging the existing modularity and automation of QLever-control, we can extend its functionality to multiple engines. This chapter presents a detailed overview of QLever-control and the `example-queries` command. This will show how this robust foundation acts as a blueprint to make a unified, extensible, and reproducible benchmarking framework possible.

3.1 QLever-control

QLever [13] is a SPARQL engine developed at the chair of Algorithms and Data Structures at the University of Freiburg. QLever can efficiently index and query very large knowledge graphs with over 100 billion triples on a single standard PC or server.

QLever-control [12] is a companion tool for QLever. It provides a Python script to create QLever SPARQL endpoints for any RDF dataset. The script is simple and intuitive to use. It provides detailed help text for all the commands and arguments on the command line. It also supports context-sensitive autocompletion for all commands and options. When using container systems such as Docker or Podman, the user doesn't even have to download the QLever code and the script downloads the image for the user. However, if QLever is built from source on the machine, the script also supports the use of native `IndexBuilderMain` and `ServerMain` binaries that QLever uses.

The script allows users to control all things QLever does, with all the configuration in one place, the so-called Qleverfile. The script comes with a number of example Qleverfiles for a wide-range of datasets, which makes it very easy to get started and also helps users to write their own Qleverfile for their own data. If the user doesn't want to use or override some parameters of the Qleverfile, they can specify the arguments directly on the command line as well.

Let us consider the example of creating a SPARQL endpoint for the *120 years of Olympics* dataset. This dataset is one of the nineteen that QLever supports out-of-the-box at the time of writing.

QLever-control is distributed via PyPI[1] and can be installed with the standard command: `pip install qlever`

After installation, the next step is to create a directory in which the index data will be stored. All dataset-specific commands for the Olympics dataset will then be executed from within this directory.

```
mkdir olympics && cd olympics   # Create and enter directory
qlever setup-config olympics    # Generate Qleverfile for Olympics
```

The command above generates a Qleverfile in the olympics directory. This file contains all the necessary configuration for working with the Olympics dataset. We will take

[1]The Python Package Index (PyPI) is a repository of software for the Python programming language.

a closer look at the Qleverfile for olympics dataset in Section 3.2. But working with out-of-box Qleverfiles is made extremely easy, and we don't have to touch the Qleverfile in this case to set up the SPARQL endpoint.

Setting up a SPARQL endpoint after generating the Qleverfile takes just a few commands:

```
qlever get-data                # Download the dataset
qlever index                   # Build index data structures
qlever start                   # Start the QLever server
qlever query                   # Launch an example query
qlever stop                    # Stop the QLever server
```

Each command above can also be appended with a `--show` argument. This shows the full command line that is being used to execute the given command. That way, users can also learn, on the side, how QLever works. For example:

```
qlever start --show
```

The output of this command when working with native binaries will look like:

```
nohup ServerMain -i olympics -j 8 -p 7019 -m 5G -c 2G -e 1G -k 200 -s 30s >
↪  olympics.server-log.txt 2>&1 &
```

Based on these observations, it becomes clear that QLever-control makes the process of setting up a SPARQL endpoint remarkably simple. For knowledge graphs with an existing Qleverfile, the entire setup can be completed in just a handful of commands. This not only reduces manual effort but also makes endpoint deployment accessible to a much broader audience. Setting up an endpoint for a custom knowledge graph is just as easy. At the heart of this simplicity lies the Qleverfile, which encapsulates all necessary configuration details in a single, shareable format.

3.2 Qleverfile

A central element of the QLever-control framework is the Qleverfile. Qleverfile is a lightweight, human-readable configuration file that specifies all parameters required to set

up and serve a SPARQL endpoint for QLever. At its core, it is a plain-text configuration file that consists of key-value pairs organized in logical sections. This design replaces the need for lengthy setup instructions, custom shell scripts, or error-prone manual procedures. Instead, it provides a single, standardized, and transparent way of describing how a SPARQL endpoint should be deployed. With this in place, setting up an endpoint requires only a few generic commands as we saw in Section 3.1

What makes Qleverfile especially powerful is not only its simplicity, but also its contribution to reproducibility. Sharing a knowledge graph setup becomes as easy as sharing its Qleverfile. Anyone with QLever-control installed can use the file to recreate the exact same endpoint, ensuring consistent experimental conditions across machines and research groups.

To further reduce barriers, QLever-control provides nineteen ready-to-use Qleverfiles. These span datasets of vastly different sizes: from just a few million triples to a hundred billion triples. This gives the users immediate, real-world examples of how to configure SPARQL engines for diverse scales. Such references not only accelerate initial experimentation but also serve as starting points for adapting setups to new knowledge graphs.

We will now examine the structure of a Qleverfile in detail. Using the Olympics Qleverfile as an example, we illustrate the most important sections and options of the configuration.

```
[data]
NAME              = olympics
BASE_URL          = https://github.com/wallscope/olympics-rdf
GET_DATA_CMD      = curl -sLo olympics.zip -C -
↪  ${BASE_URL}/raw/master/data/olympics-nt-nodup.zip && unzip -q -o
↪  olympics.zip && rm olympics.zip
DESCRIPTION       = 120 Years of Olympics, data from ${BASE_URL}
```

The `[data]` section defines the dataset that we wish to serve. At a minimum, every dataset should be assigned a clear and descriptive `NAME`. If we want the command `qlever get-data` to automatically retrieve or prepare the dataset, we must specify a `GET_DATA_CMD`. Otherwise, the input files need to be generated, downloaded, or provided manually. Each dataset should also include a concise `DESCRIPTION`, ideally noting its origin and date of retrieval.

```
[index]
INPUT_FILES     = olympics.nt
CAT_INPUT_FILES = cat ${INPUT_FILES}
```

The `[index]` section specifies input files and indexing settings. The format for `INPUT_FILES` should be such that `ls ${INPUT_FILES}` lists all input files. The parameter `CAT_INPUT_FILES` defines how these files are concatenated into a single input stream (for example, using `cat` for plain text files or `zcat` for compressed ones).

```
[server]
PORT               = 7019
TIMEOUT            = 30s
MEMORY_FOR_QUERIES = 5G
CACHE_MAX_SIZE     = 2G
```

The `[server]` section configures the SPARQL server itself. The parameter `PORT` determines on which port the server accepts queries. The `TIMEOUT` parameter configures the maximum time a query is allowed to run before it is terminated. All the other parameters are specific to the ServerMain binary of QLever.

```
[runtime]
SYSTEM = docker
IMAGE  = docker.io/adfreiburg/qlever:latest
```

The `[runtime]` section determines the execution environment for QLever. By setting `SYSTEM = docker` (or podman), the engine runs inside a container environment, which ensures consistency across machines, as the required image is pulled automatically. Alternatively, `SYSTEM = native` directs QLever to use locally compiled binaries such as `IndexBuilderMain` and `ServerMain`, which must be accessible in our environment's PATH.

3.3 The `example-queries` command

One of the key features of QLever-control is the `example-queries` command, which allows users to run predefined queries against a SPARQL endpoint and inspect their performance. While it was originally designed to run the example queries shown in the QLever demonstration interface, it also doubles as a lightweight benchmarking tool by measuring both execution time and result size.

The `example-queries` command can be invoked directly from the command line and supports detailed help and context-sensitive autocompletion.

```
qlever example-queries
```

By default, it reads the host and port from the Qleverfile to form the default QLever SPARQL endpoint. Queries for the specified `[data] NAME` option in Qleverfile (i.e. an out-of-box dataset) are automatically fetched from a remote API maintained for the QLever UI client. Both the SPARQL endpoint and the queries can be overridden using command-line arguments. For example, a custom endpoint can be specified using `--sparql-endpoint`, or queries can be provided in TSV format using `--get-queries-cmd`. This allows `example-queries` to run any set of queries against an arbitrary SPARQL endpoint, not just QLever.

The command offers a rich set of options for flexibility. Users can run a subset of queries (by ID or regular expression), select the output format (JSON, TSV, CSV, or Turtle), fetch only result sizes instead of full results, or limit the number of returned results.

From a benchmarking perspective, `example-queries` is particularly valuable because it provides a reproducible mechanism for evaluating query performance. Before the work in this thesis, the command only printed runtime statistics to the console. By extending it to save structured output, it becomes possible to systematically record execution times and result sizes across different endpoints or engine versions. This capability forms the foundation for automated benchmarking workflows, which can then be generalized to other engines using the same Qleverfile configuration model.

4 Approach and Implementation

In this thesis, we are trying to answer a simple but challenging research question: which SPARQL engine should a user choose that not only performs well on the current dataset and workload, but also scales effectively as data and query demands grow? Answering it is far from trivial, as discussed in Section 1.1. Each engine comes with its own installation steps, configuration quirks, and command-line interface, making setup slow and error-prone. Running queries often requires manual scripts, and achieving a consistent and comprehensive comparison across engines is difficult without an automated framework and clear visualization tools.

What if evaluating multiple SPARQL engines over large knowledge graphs was as easy as running just a few simple commands without ever having to worry about the internal details? In the previous Chapter 3, we saw how QLever-control already achieves this for QLever. A single `qlever setup-config` command generates a Qleverfile that captures all the necessary configuration. Users only edit the most relevant options, and all the internal complexity is handled automatically. A few simple commands such as `qlever get-data`, `qlever index`, and `qlever start` are all what's needed to set up the SPARQL endpoint.

What if, we can extend this same functionality to other SPARQL engines and set up a SPARQL endpoint for all the engines using uniform commands? The `setup-config` command can give us a Qleverfile tailored to the needs of the engine in question. We would need to change only the relevant sections, just like before. And then a familiar workflow of `get-data`, `index`, and `start` can set up the SPARQL engine for the given dataset. The work detailed in this chapter will achieve this functionality for all the six additional open-source engines in question.

To go one step further, what if we could also evaluate all the SPARQL engines in a fully systematic way? The `example-queries` command from Section 3.3 could be extended into a true benchmarking utility that works uniformly across engines. With just a single command per engine, we could run the same set of benchmark queries and store the results in a shared directory. These results could then be automatically consumed and presented in a clean, interactive web application. This application would visualize query

runtimes, result sizes, and aggregate metrics in well-organized tables, enabling side-by-side comparisons across datasets and engines. What was once a tedious and error-prone process would become reproducible, shareable, and easy to extend.

The following sections detail the implementation of this unified benchmarking framework. We begin by detailing how the QLever-control Python package was extended to support SPARQL endpoint setup for other engines. We provide a broad overview of how engine-specific commands were implemented and how engine-specific quirks were handled. Then, we take a closer look at how the `benchmark-queries` command replaces the existing `example-queries` command to allow uniform benchmarking of all the engines. Finally, we will discuss the `serve-evaluation-app` command that automatically launches the evaluation web application and how the web application consumes the results and presents them interactively, making it simple to analyze performance and scalability across different SPARQL engines.

4.1 Extending QLever-control for Multi-Engine Support

This section describes how QLever-control was extended from supporting a single engine (QLever) to supporting multiple engines. The guiding principles for this redesign were rooted in core object-oriented software engineering ideas, particularly Separation of Concerns (SoC) and Don't Repeat Yourself (DRY). SoC guaranteed that all engine-specific logic remained isolated. DRY ensured that common functionality was implemented once and reused across all engines. Together, these principles helped create a clean structure that both preserved the original simplicity of QLever-control and made it future-proof for additional engines.

Directory Structure

The source code of QLever-Control is organized in a modular way. Figure 2 shows the original single-engine directory tree, while Figure 3 illustrates the extended structure for multiple engines.

The original layout (see Figure 2) consisted of a single `src/qlever` directory containing the core logic of the application. The `qleverfile.py` file defined all supported command-line arguments. The `Qleverfiles` directory stored template Qleverfiles specifying datasets and configuration parameters. The `commands` subdirectory contained the implementations of all user-facing commands such as `get-data`, `index`, and `start`. All these commands

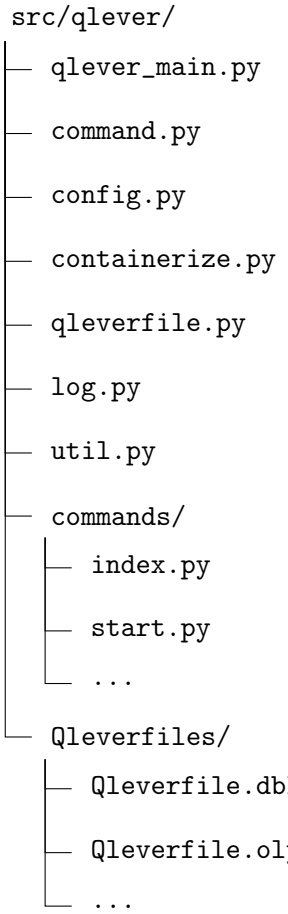

Figure 2: Original QLever-only directory structure.

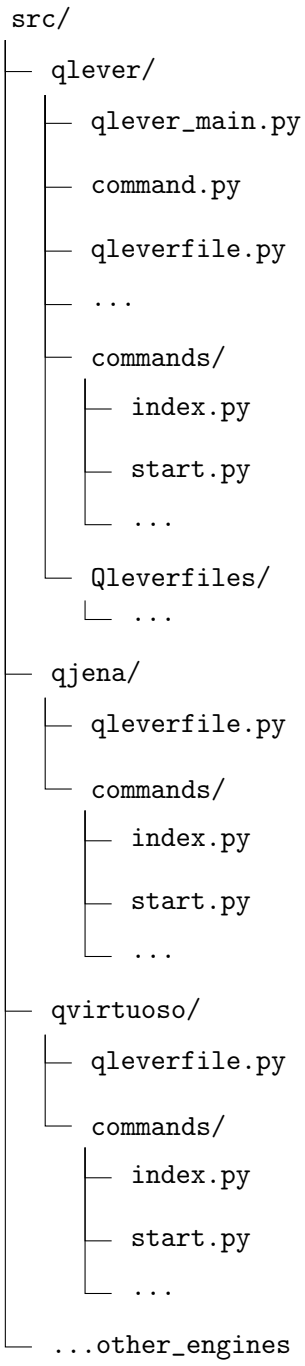

Figure 3: Extended multi-engine directory structure.

implemented the abstract interface defined in `src/qlever/command.py`. The commands also specify which Qleverfile arguments are relevant for their execution.

The extended multi-engine layout (see Figure 3) introduces one directory per engine: `src/<qengine>/`. `<qengine>` here refers to the engine-specific wrapper script used to invoke the engine commands. All the core application logic still resides in the `src/qlever` directory. Each new engine directory contains its own `qleverfile.py`, which defines the new engine-specific command-line arguments. These arguments either supplement or override the command-line arguments specified in `src/qlever/qleverfile.py`. Each engine directory also has its own `commands/` subdirectory for the corresponding command implementations. The template Qleverfiles remain in their original directory and are simply adapted by each engine during setup. This design keeps all engine-specific code localized, ensuring clear separation of concerns.

All command classes for the new engines follow the same abstract interface defined in `src/qlever/command.py`. Whenever a new engine's command shares substantial functionality with an existing QLever command, the concrete QLever implementation serves as a base class. Only the differing functionality is overridden, which significantly reduces code duplication and adheres to the DRY principle. This is discussed in more detail in Section 4.2.

Execution Flow

Despite the structural changes, the execution flow remains uniform across all engines. Each engine is invoked via a dedicated wrapper script (e.g., `qlever`, `qjena`, or `qvirtuoso`), all of which call the common entry point `src/qlever/qlever_main.py`. Script names are constructed by prefixing the engine name with **q**, avoiding conflicts with binaries of the underlying engines that may already be installed. At runtime, the script name is extracted and mapped to the corresponding directory in `src/`.

Using the script name as an identifier, QLever-control dynamically discovers the relevant commands in the `src/<qengine>/commands/` directory and instantiates them. The command-line options from `src/qlever/qleverfile.py` are then merged with the engine-specific arguments from `src/<qengine>/qleverfile.py`. From this combined set, only the options relevant to the chosen command are parsed, after which the command is executed.

This modular flow ensures that adding a new engine requires no changes to the central logic. A new engine only needs to provide its own arguments and command implementa-

tions within its directory. As a result, QLever-control was extended from supporting a single engine to handling multiple SPARQL engines with minimal modification to the original codebase.

4.2 Implementation of Engine-Specific Commands

In the previous section, we described how QLever-control was extended to support multiple SPARQL engines by modifying the directory structure and execution flow. While this establishes the framework for multi-engine support, it is equally important to understand how the individual commands for each engine are implemented.

This section provides an overview of the common approaches used to implement engine-specific commands. We categorize the commands into three groups:

- **Engine-agnostic commands**, which can be reused across all engines without modification.
- **Derived commands**, which inherit from existing QLever (or other engine) implementations and override only the parts that differ.
- **Non-reusable commands**, which must be developed from scratch because they address functionality unique to a given engine.

This provides a clear mental model for adding support for additional engines and commands in the future.

4.2.1 Engine-agnostic commands

Some commands in the multi-engine framework are designed to be completely engine-agnostic. These commands can be reused across all supported engines without modification, as their functionality does not depend on engine-specific behavior. These commands include:

- `get-data`: Command to download/retrieve the dataset.
- `log`: Command to show the last lines of the server log file and follow it.

The `get-data` command simply executes the bash command (`GET_DATA_CMD`) defined in the `[data]` section of the Qleverfile using Python's `subprocess` module. This allows the framework to retrieve or prepare datasets independent of the underlying SPARQL engine.

The `log` command uses Python's `subprocess` module along with the Unix `tail` utility to display and follow the last lines of the server log file. As long as the engines can be configured to generate the server log with the same filename i.e. `<NAME>.server-log.txt`, the `log` command operates without engine-specific customization. `<NAME>` here refers to the `NAME` option specified in `[data]` section of the Qleverfile.

Whenever the engine-agnostic behavior is sufficient for a command, we simply create a symbolic link to the command defined in `src/qlever/commands/` rather than duplicating code.

4.2.2 Derived commands

Some commands behave almost identically across all engines, differing only in small details such as default engine-specific parameters. For these cases, it is sufficient to reuse an existing implementation (from QLever or another engine) as a base class and extend it only where necessary. This avoids code duplication and reduces maintenance overhead. The following commands fall into this category:

- `benchmark-queries`: Run the given benchmark queries and show their processing times and result sizes (optionally save the results in a file).
- `query`: executes a single SPARQL query against the endpoint.
- `status`: Show index/server processes running on the machine.
- `stop`: terminates the server process if running.
- `setup-config`: generates a Qleverfile in the current working directory.

The `query` and `benchmark-queries` commands share the same core functionality of sending queries to the SPARQL endpoint. The only difference across engines lies in the default endpoint URL. For example, QLever exposes its endpoint at `<host>:<port>`, while Oxigraph uses `<host>:<port>/query`. By using inheritance, the QLever implementation can be reused directly, with only the `sparql_endpoint` argument replaced in each engine-specific subclass before calling the QLever superclass implementation. Listing 3 shows Oxigraph's `QueryCommand` class implementation.

A similar pattern applies to the `status` and `stop` commands, both of which rely on a command-line regular expression (`cmdline_regex`) to identify the running server process. Again, the QLever implementation can serve as the base, with engine-specific subclasses overriding only the `cmdline_regex` used to match the process name.

Listing 3 Python implementation of `query` command for Oxigraph

```python
from qlever.commands.query import QueryCommand as QleverQueryCommand

# Query Command implementation for Oxigraph
class QueryCommand(QleverQueryCommand):

    def execute(self, args) -> bool:
        # Override QLever's default endpoint format of host_name:port
        args.sparql_endpoint = f"{args.host_name}:{args.port}/query"
        return super().execute(args)
```

The `setup-config` command

The `setup-config` command requires slightly more adaptation. For QLever, the command simply copies the appropriate Qleverfile from `src/qlever/Qleverfiles` into the user's working directory. For the new engines, however, the Qleverfiles must be adapted to account for engine-specific parameters. To avoid duplicating logic across engines, a shared implementation was created. All engines reuse the same code for reading the template Qleverfile, preserving the common parameters, and writing the adapted configuration into the user's working directory. The only method that needs to be overridden is the one responsible for adding engine-specific parameters. A full implementation of this design is provided in Oxigraph's `setup-config` command, while the other engines simply inherit from it and override the adaptation method.

To make working with these engines as seamless as possible, several new command-line arguments were introduced. In particular, the parameters `--total-index-memory` and `--total-server-memory` allow the overridable adaptation method to directly compute suitable default memory settings for the index and server processes of the respective engine. This ensures that the generated Qleverfile contains sensible defaults without requiring users to understand the complex engine-specific memory configurations. In addition, the arguments `--port`, `--timeout`, and `--system` provide users a convenient way to overwrite the default values of the most common parameters. Together, these extensions mean that in most cases users do not need to modify the generated Qleverfile manually, reducing setup complexity and improving the overall usability of the framework.

As an example, running the following command

```
qjena setup-config olympics --total-index-memory 32G --total-server-memory
↪  16G --port 1234 --timeout 180s --system native
```

will generate a Qleverfile with the following sections:

```
[index]
INPUT_FILES = olympics.nt
JVM_ARGS = -Xms32G -Xmx32G
THREADS = 11

[server]
PORT = 1234
JVM_ARGS = -Xms16G -Xmx16G
TIMEOUT = 180s

[runtime]
SYSTEM = native
```

Here, the memory parameters are automatically configured using the user-provided values. The number of threads is derived automatically from the number of CPU cores on the user's machine. In this example, the system has 12 cores, so the configuration assigns 11 threads. This follows the recommendation of Jena's index binary `tdb2.xloader` [1] that advises using one fewer thread than the total number of cores as an initial setting. This automatic tuning of the Qleverfile ensures near-optimal performance without requiring the user to tune the configuration manually. The memory-related arguments for all new engines are explained in detail in Section 4.3.

4.2.3 Non-reusable commands

Unlike the derived commands, some commands cannot be reused or lightly adapted across engines. Instead, they require dedicated implementations tailored to each engine's specific binaries and runtime behavior. The most important commands in this category are:

- `index`: Build the index for a given RDF dataset.
- `start`: Launch the server for the engine.

Both `index` and `start` commands are tightly coupled to the internal mechanics of each engine. Therefore, they naturally vary from one engine to another and cannot be generalized through simple inheritance or argument adjustments.

It is not feasible to detail the full workings of these commands for each engine here.

[1] https://jena.apache.org/documentation/tdb/tdb-xloader.html

However, both commands still follow a broadly similar blueprint as shown in Listing 4. The engine-specific parameters and quirks that these commands must handle will be covered in Section 4.3.

Listing 4 Execution flow for `index` and `start` commands

```
Construct engine-specific command line using the input arguments:
   - Use the appropriate index/server binary for the selected engine
   - If system != native:
        - Wrap the command in container execution (docker/podman)

Handle show-only mode:
   - If --show flag is set:
        - Display the constructed command line to the user
        - Exit without execution

Perform validation checks:
   - If system == native:
        - Verify that the index/system binary is installed and accessible
   - For the index command:
        - Check that the required input files exist in the current directory
        - Check if a previous index does not exist already
   - For the start command:
        - Check that an index is present in the current directory
        - Verify that no server is already running at the specified port

Execute the command:
   - If all checks succeed, launch the command line process
   - Log success or error messages appropriately
```

4.3 Engine-specific parameters and quirks

In the previous section, we described how common design patterns and abstractions allowed most commands to be reused or extended across engines. This section introduces the six new engines that were added to QLever-control and details the concrete adaptations required to ensure that the commands function correctly for each of them. This provides a comprehensive picture of the practical adjustments needed to support multiple SPARQL engines within the unified framework.

To integrate the six additional engines into the extended framework, we followed a uniform pattern as outlined in Section 4.1. It was sufficient to create a new `src/<qengine>`

directory, provide engine-specific arguments in `qleverfile.py`, and implement the required commands inside the `commands` subdirectory. The `<qengine>` here refers to the wrapper script that was used to execute the engine commands (e.g. `qoxigraph` for Oxigraph).

Since many `qleverfile.py` arguments are self-explanatory and similar across engines, we highlight only the most interesting ones in this section. Each engine's `index` and `start` commands support the `--extra-args` option, which accepts a free-form string of additional arguments passed directly to the underlying indexing and server binaries. This feature allows advanced users to specify configuration options that are not exposed in the Qleverfile. The complete and up-to-date list of arguments can be inspected directly in the code repository[2].

4.3.1 Oxigraph (`qoxigraph`)

Oxigraph is a lightweight SPARQL engine based on the RocksDB key-value store that fully implements the SPARQL 1.1 standard. Oxigraph is written in Rust and is in heavy development at the time of writing. It relies on a single binary, `oxigraph`, to perform both indexing and serving tasks. This design greatly simplifies command construction.

As discussed in Section 4.2, the `setup-config` command is fully implemented for Oxigraph, with other engines inheriting from this implementation. One peculiarity of Oxigraph is that its binary does not support any form of memory-based configuration. As a result, the options `--total-index-memory` and `--total-server-memory` are not available for Oxigraph. This keeps its configuration simpler but also more limited.

The arguments defined in `qleverfile.py` cover both indexing and serving tasks. The most interesting ones of these are:

- `--ulimit`: raises the maximum number of files that can be opened simultaneously. This is particularly important when indexing many large input files, where the command automatically sets the limit to 500,000 if the dataset exceeds 10 GB. Oxigraph index process can sometimes fail if this limit is not set high enough.
- `--lenient`: attempt to keep loading even if the data file is invalid (default: `False`).

The `index` command stores each engine's index files in a dedicated subdirectory named `<NAME>_index`, where `<NAME>` corresponds to the value of the `NAME` option in the `[data]` section of the Qleverfile. This index directory name is then used by the `start` command

[2] `https://github.com/ad-freiburg/sparql-engine-evaluation-tanmay`

to start the server. This gives us the ability to link each server process directly to its configuration name, which greatly benefits the `status` and `stop` commands. For instance, if `qoxigraph` instances for both `olympics` and `dblp` datasets are running, the `status` command will list two processes, identifiable by `olympics_index` and `dblp_index`. The `stop` command is also then able to terminate the correct process by matching the index directory name, leaving the other unaffected.

Native timeout support was only introduced to Oxigraph in version 0.5. To save the user from worrying about version-specific timeout details, the `start` command automatically inspects the output of `oxigraph serve --help` at runtime and adds the timeout option, if supported.

4.3.2 Apache Jena (`qjena`)

Apache Jena (a project of the Apache Software Foundation since 2012) is a widely used SPARQL engine implemented in Java for building Semantic Web and Linked Data applications. It integrates with TDB2, which can be used as a high-performance RDF store on a single machine. While TDB2 supports multiple loaders, we adopt `tdb2.xloader` as the standard indexing binary in our framework. The reason is that `tdb2.xloader` is designed as a robust bulk loader optimized for large datasets, prioritizing stability and reliability. Unlike other TDB2 loaders that trade off stability for performance on smaller datasets, `tdb2.xloader` provides a consistent and predictable loading process across dataset sizes. This makes it an ideal choice as the single supported indexer within our unified interface. Apache Jena Fuseki, which uses the `fuseki-server` binary, is a SPARQL server and can run as a standalone server.

A custom Dockerfile is defined for Jena. The image tagged `adfreiburg/qjena` uses `openjdk:21-jdk-slim` as the base image, downloads the latest Apache Jena and Fuseki packages, installs the binaries and adds them to the `PATH`, and sets `bash` as the entrypoint. When using containerized execution, if the `adfreiburg/qjena` image is not found locally, it will be built automatically the first time the `index` command is invoked.

The most interesting `qleverfile.py` arguments for Jena are:

- `--threads`: sets the number of threads for indexing, with the recommended setting being the number of CPU cores minus one, as advised by Jena's own documentation.
- `--jvm-args`: specifies Java Virtual Machine options for both index and server processes (e.g., heap size).

- `--extra-env-args`: allows additional java-related environment variables to be passed to the underlying Jena index and server binaries.

The `JVM_ARGS` option is automatically set by the `setup-config` command for the Qleverfile to reflect the total index and server memory limits chosen by the user, ensuring consistent resource allocation across runs.

Just like in Oxigraph, the `index` command creates a dedicated subdirectory named `<NAME>_index`, which is then used to start the server. This allows the `status` and `stop` commands to reliably identify and manage Jena processes by configuration name.

4.3.3 Blazegraph (`qblazegraph`)

Blazegraph is a Java-based SPARQL engine best known for powering the official Wikidata query service. Despite its widespread adoption in the past, it is now considered "abandonware", with its last public release being version 2_1_6_RC (released in Feb 2020). Unlike other engines that ship with separate indexing and server binaries, Blazegraph relies on a single `blazegraph.jar` file, which can be used both for bulk loading and serving queries.

We also define a custom Dockerfile for Blazegraph. It uses `openjdk:21-jdk-slim` as the base image, downloads `blazegraph.jar`, and ends with bash as the entrypoint. When the `index` command is run for the first time in a containerized setup, the framework automatically checks for the `adfreiburg/qblazegraph` image, builds it if missing, and then proceeds with the indexing.

Just like in Jena, the most relevant `qleverfile.py` argument for Blazegraph is:

- `--jvm-args`: specifies JVM options such as heap size for both indexing and serving processes. This is critical for performance on large datasets, where sufficient heap allocation prevents memory-related failures. As with Jena, the `setup-config` command automatically maps the user's total index and server memory inputs to the correct `--jvm-args` settings in the Qleverfile.

Blazegraph relies on two configuration files: `RWStore.properties` and `web.xml` for configuring bulk loader and server settings respectively. In our implementation, the `setup-config` command behaves as usual, but additionally places both configuration files in the current working directory. The provided `RWStore.properties` file is already tuned for fast bulk loading with sensible defaults, though users can adjust it manually before running `index`. The `web.xml` file defines runtime options for the server, such as query timeout and read-only mode. However, these options in the `web.xml` don't need to

be modified. When the `start` command is called, the framework automatically updates these options to match those specified in the Qleverfile. This makes sure that users don't have to manually edit these Blazegraph-specific configuration files and can simply start a SPARQL endpoint by running `qblazegraph index` followed by `qblazegraph start`.

Unlike Oxigraph and Jena, Blazegraph does not produce a dedicated index subdirectory but instead stores all data in a single `blazegraph.jnl` file. To work around this limitation, the framework relies on the `web.xml` configuration file, which is renamed during the `start` command to include the configuration name. For example, if the user runs `qblazegraph start` for the olympics dataset, the framework will prepare a file named `olympics.web.xml` with the correct parameters and use it to launch the Blazegraph server. This ensures that multiple Blazegraph instances are tied to their configuration name. Consequently, the `status` and `stop` commands can reliably identify and manage these processes just as they do for Oxigraph and Jena.

4.3.4 MillenniumDB (`qmdb`)

MillenniumDB is a graph oriented database management system developed by the Millennium Institute for Foundational Research on Data (IMFD). It is implemented in C++ and designed for high-performance query execution over RDF and property graph data. Like Oxigraph, it also relies on a single binary, `mdb`, to perform both indexing and serving tasks.

The official MillenniumDB image available on Docker Hub is outdated, whereas the Dockerfile maintained in their GitHub repository is actively updated. To prevent users from having to manually clone the repository when working in container environments, the framework directly builds the Dockerfile from the GitHub URL and tags the resulting image as `adfreiburg/qmdb`.

The most interesting `qleverfile.py` arguments for MillenniumDB are:

- `--buffer-strings`, `--buffer-tensors`: control the buffer sizes used during data import. The `setup-config` command automatically divides the total indexing memory equally between the two.
- `--threads`: sets the number of worker threads for query evaluation (fixed to 2 by default in our framework).

- `--strings-static`, `--strings-dynamic`, `--versioned-buffer`, `--unversioned-buffer`:
  These determine the buffer sizes used for the server process. Their values are automatically derived from the total memory specified during `setup-config`, with the majority of the memory allocated to versioned buffer.

Just like in Oxigraph and Jena, the `index` command stores the MillenniumDB index in a dedicated subdirectory named `<NAME>_index`, which is then used to start the server. This allows the `status` and `stop` commands to reliably identify and manage MillenniumDB processes by configuration name.

4.3.5 Virtuoso (`qvirtuoso`)

Virtuoso, developed by Open-Link Software, is a mature and feature-rich SPARQL engine that is written in C and comes in both open-source and commercial editions. In this work, we focus exclusively on the open-source version. Virtuoso makes use of two primary binaries: `isql` for data loading and `virtuoso-t` for starting and running the server.

A notable quirk of Virtuoso is that the server must already be running in order for the data to be loaded. We try to account for this behavior in our `index` implementation by doing the following. The command first starts the server and polls it until it is ready. It then proceeds with data loading through `isql`, and finally shuts the server down. After indexing, the `start` command can be used to relaunch the server with the built index, ensuring a consistent and reproducible workflow.

Another distinctive feature of Virtuoso is its reliance on an INI-style configuration file, `virtuoso.ini`, which exposes a wide range of tunable settings. To simplify usage, we implement support for the most important configuration options directly in `qleverfile.py`. When the user runs the `index` or `start` command, the parameters are taken from the Qleverfile and automatically inserted into `virtuoso.ini`.

The most relevant arguments exposed through `qleverfile.py` are:

- `--isql-port`: the port used by Virtuoso's ISQL client (default: `1111`).
- `--num-parallel-loaders`: controls parallelization during data loading, with the recommended maximum being approximately the number of CPU cores divided by 2.5.
- `--free-memory-gb`: specifies the free system memory allocated for Virtuoso buffers, which is used to derive `NumberOfBuffers` and `MaxDirtyBuffers` options.

- `--max-query-memory`: sets the memory available to the query processor.

Both `FREE_MEMORY_GB` and `MAX_QUERY_MEMORY` options are automatically calculated for the Qleverfile during the `setup-config` step, based on the memory limits specified by the user.

Like Blazegraph, Virtuoso does not produce a dedicated index subdirectory but instead stores all data in a single `virtuoso.db` file. Similar to the approach taken for Blazegraph, the framework renames the `virtuoso.ini` to include the configuration name before starting the server i.e. `<NAME>.virtuoso.ini`. This ensures that multiple Virtuoso instances are tied to their configuration names. Consequently, the `status` and `stop` commands can reliably identify and manage these processes just as they do for other engines.

Virtuoso also provides the ability to sequentially load data into an existing `virtuoso.db` index. To support this, we introduce the `--extend-existing-index` option for the `index` command. With this flag, the user can repeatedly update the `[index] INPUT_FILES` entry in the Qleverfile and load new data into the same index. This feature is especially useful when working with massive datasets such as Wikidata, where a single bulk load may fail without committing progress. The incremental loading of the dataset makes it possible to complete very large data imports reliably without restarting from scratch.

4.3.6 GraphDB (`qgraphdb`)

GraphDB, developed by Ontotext, is a highly efficient, scalable and robust graph database with RDF and SPARQL support. Written in Java, it comes with both free and enterprise editions. In this work, we focus on the free edition. GraphDB provides two key binaries: `importrdf` for bulk data loading and `graphdb` for running the SPARQL server.

Typically, GraphDB exposes an interactive console through which index and server settings can be configured prior to data loading. To maintain consistency with the unified workflow of other engines, the framework bypasses the console and instead downloads GraphDB's internal `config.ttl` configuration file automatically. This file is then overwritten with parameters taken from the Qleverfile when the `index` command is executed.

By default, GraphDB stores all datasets in a common location and exposes all of them on port 7200. Our framework overrides this behavior by generating each index in the current working directory and binding the server to the port specified in the Qleverfile,

ensuring complete separation of multiple datasets. The index subdirectory is again named `<NAME>_index`, which is then used by `start`. This ensures that the `status` and `stop` commands can reliably identify and manage GraphDB processes by configuration name.

One notable quirk of GraphDB is that even its free edition requires a license file. Users must generate this license on the GraphDB website, after which it can be supplied to the framework. To accommodate this, the `start` command provides an additional argument, `--license-file-path`, which specifies the location of the required license file.

The most relevant `qleverfile.py` arguments for GraphDB are:

- `--threads`: sets the number of RDF parsers to use during indexing.
- `--entity-index-size`: controls the initial size of the entity hash table, where larger sizes reduce collisions and improve retrieval performance.
- `--jvm-args`: specifies JVM arguments for both index and server processes (e.g., heap size).
- `--ruleset`: selects the entailment ruleset to apply (e.g., `rdfs`, `owl-horst`, `owl2-rl`), which determines the applied semantics (default = "empty").
- `--heap-size-gb`: sets the Java minimum and maximum heap size (`-Xms` and `-Xmx`) for the GraphDB server.

The `ENTITY_INDEX_SIZE`, `JVM_ARGS` and `HEAP_SIZE_GB` arguments are automatically calculated for the Qleverfile during the `setup-config` step, based on the memory limits specified by the user.

4.4 The `benchmark-queries` command

The `benchmark-queries` command extends and generalizes the functionality of the earlier `example-queries` command described in Section 3.3. The `example-queries` command was tied to QLever's demo interface and focused on running predefined example queries or reading simple TSV files that contained a description and a query. The new command was designed explicitly with benchmarking in mind. It retains all the core features of executing SPARQL queries against a given endpoint, measuring runtime, and recording result sizes, but introduces several important enhancements.

The most important change is support for benchmark queries in YAML format, in addition to the existing TSV format. YAML files allow attaching rich metadata such as benchmark name, description, scale and per-query descriptions. These can later be used in

the evaluation web application to provide more information about the benchmark. Let's say we have the benchmark queries for the DBLP [28] dataset. A snippet of benchmark queries YAML file for it is shown in Listing 5.

Listing 5 Example YAML benchmark queries file snippet for the DBLP dataset

```yaml
name: DBLP
description: Sparqloscope benchmark queries generated for DBLP dataset
scale: 525000000       # Approx number of triples
queries:
  - name: join-2-small-large
    description: JOIN of a small and a large predicate
    query: |
      PREFIX dblp: <https://dblp.org/rdf/schema#>
      SELECT (COUNT(*) AS ?count)
      WHERE {
        ?s dblp:formerStreamTitle ?o1 .
        ?s rdf:type ?o2
      }
  - name: join-2-large-small
    description: JOIN of a large and a small predicate
    query: |
      PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
      PREFIX dblp: <https://dblp.org/rdf/schema#>
      SELECT (COUNT(*) AS ?count)
      WHERE {
        ?s rdf:type ?o1 .
        ?s dblp:formerStreamTitle ?o2
      }
```

A second extension is giving the user the option to store the benchmark results in a structured YAML file. To achieve this, two arguments were added to the `benchmark-queries` command:

- `--result-file`: Base name used for the result YAML file, which should be of the form `<dataset>.<engine>`, e.g., `dblp.qlever`.
- `--results-dir`: The directory where the YAML result file would be saved.

Recall from Section 4.2.2 that, when calling `benchmark-queries` through an engine's wrapper script (e.g., `qoxigraph benchmark-queries`), the SPARQL endpoint is already preconfigured for that engine. This means users do not need to manually pass `--sparql-endpoint`, simplifying benchmark execution. By combining this with the `--result-file` and `--results-dir` options, the same queries for a benchmark can be executed against multiple engines, and their results collected in a common results directory. To keep result

files compact but still informative, only the first few responses per query are stored, configurable via `--max-results-output-file` argument. For example, the result file `dblp.oxigraph.results.yaml` would hold Oxigraph's results for the DBLP benchmark, while `dblp.jena.results.yaml` holds Jena's. These files can then later be consumed by the evaluation web application, which is covered in Section 4.5.

The result YAML file for a given benchmark and engine looks similar to the queries YAML file in Listing 5. In addition to the benchmark name, description, and scale, it also stores the timeout value in seconds specified for the benchmark. For each query, we also store the runtime in seconds, the result size, and the first few results (limited by `--max-results-output-file`).

To bolster the robustness of the benchmarking framework, we implemented the `--restart-on-hang` argument. This flag, when supplied to the `benchmark-queries` command, addresses issues where an engine either crashes or hangs indefinitely after the timeout period. In such events, the system automatically restarts the engine's server, notes the restart in the results YAML file, and continues execution with the next query. This prevents a single problematic query from blocking an entire benchmarking run. It is especially useful for engines such as Oxigraph and Blazegraph. Oxigraph does not support query timeouts for versions before v0.5 and may run indefinitely on difficult queries. Similarly, while Blazegraph has a global timeout, certain queries have been observed to run beyond the configured limit. With `--restart-on-hang`, benchmarking sessions complete reliably without manual intervention.

4.5 The Evaluation Web Application

When the same `--results-dir` argument is used across engines when executing the `benchmark-queries` command, the resulting YAML files accumulate in a shared directory. The purpose of the evaluation app is to consume these files, transform them into a single structured JSON representation, and serve them to a lightweight web application for interactive analysis.

This section has two parts. First, we describe the new `serve-evaluation-app` command, which automates the transformation of YAML files and serving of the web application. Then, we present the web application itself, which enables comparisons of engine performance across benchmarks.

4.5.1 The `serve-evaluation-app` command

The `serve-evaluation-app` command automates three tasks:

1. Reading all benchmark result files in the specified directory.
2. Converting the structured YAML files into one JSON representation.
3. Starting a web server that provides the data to the evaluation frontend.

Listing 6 Launching the `serve-evaluation-app` command

```
qlever serve-evaluation-app --results-dir /path/to/eval/results
```

By default, the web server is served on `localhost` at port 8000. These defaults can be overridden with the `--host` and `--port` arguments. An additional optional argument, `--title-overview-page`, sets the title of the overview page of the web app (default: **SPARQL Engine Performance Evaluation**).

Each benchmark result YAML file contributes query-level statistics (runtime, result size, partial results, success or failure). From these values, a set of aggregate performance metrics for each engine and benchmark are computed:

- **Arithmetic mean runtime**: the average runtime across all successful queries. Failed queries are penalized with a runtime of timeout × 2.
- **Geometric mean runtime (P=2)**: the geometric mean of runtimes, which reduces the effect of extreme outliers. Failed queries are penalized with a runtime of timeout × 2.
- **Geometric mean runtime (P=10)**: the geometric mean of runtimes, with failed queries penalized with a runtime of timeout × 10.
- **Median runtime**: the median of all runtimes, representing the typical performance. Failed queries are penalized with a runtime of timeout × 2.
- **Failure rate**: the percentage of queries that did not produce a valid result within the timeout.
- **Runtime distribution**: the percentage of queries falling into predefined categories such as under 1 second, between 1-5 seconds, and over 5 seconds.

These metrics provide a compact yet comprehensive summary of an engine's performance on a benchmark. The raw per-query runtimes and result sizes are preserved as well.

The processed data is then merged into a single JSON object with two top-level keys:

- **performance_data**: a mapping from benchmark to engine. Each engine contains the aggregate metrics data listed above as well as detailed query-level entries for the given benchmark.
- **additional_data**: benchmark metadata, such as benchmark name, description, and scale.

The Python `http.server` is used to start the web server and the JSON data is provided to the evaluation frontend at `/yaml_data`. After launching the command, the URL where the web app is served is displayed to the user.

4.5.2 The web application

The evaluation web application makes benchmarking results easy to explore, compare, and share. The web-app is implemented using HTML, CSS, plain JavaScript and Bootstrap v5.3 for additional styling. This technology stack was chosen for its simplicity, robustness, and longevity. Plain JavaScript ensures that the app is lightweight and fast, and runs anywhere a browser does. Bootstrap provides a modern, responsive design that adapts to different screen sizes, ensuring usability on desktops, laptops, or tablets. Further, Ag Grid (Community Edition) powers the interactive tables and adds advanced functionality such as sorting, filtering, and exporting. Together, these technologies create a lightweight, future-proof application that researchers can rely on for reproducible evaluations.

The main entry point of the web application is the **Overview Page** (Figure 4). On this page, the aggregate performance metrics of the SPARQL engines are displayed for each benchmark. The table header shows the benchmark name and a small info button, which can be tapped to reveal details about the benchmark. The benchmark name and description correspond to the top-level `NAME` and `DESCRIPTION` keys in the benchmark-queries YAML file (Listing 5). The tables are sorted by benchmark scale in ascending order, where the scale corresponds to the `SCALE` key in the same YAML listing. In each table, the first column lists the engines, and the remaining columns show aggregate metrics for that benchmark. The aggregate metrics used are described in Section 4.5.1.

Thanks to Ag Grid, users can sort and filter metrics by clicking on the column headers, quickly spotting engines that perform better or worse under different benchmarks. Each table can also be exported as a TSV file, which makes it easy to carry results into external analysis workflows.

DBLP ⓘ TSV **Compare Results**

Engine	G.Mean (P=2)	G.Mean (P=10)	Median (P=2)	A. Mean (P=2)	Failed
QLever	0.29 s	0.29 s	0.24 s	1.76 s	0.00 %
Blazegraph	11.79 s	14.84 s	29.64 s	80.64 s	14.29 %
Graphdb	9.49 s	10.40 s	23.09 s	55.68 s	5.71 %
Virtuoso	0.82 s	0.86 s	1.39 s	14.68 s	2.86 %
Mdb	2.06 s	2.10 s	3.63 s	15.14 s	0.95 %
Jena	23.00 s	35.87 s	55.01 s	127.35 s	27.62 %
Oxigraph	19.18 s	31.32 s	31.29 s	130.22 s	30.48 %

Figure 4: Example aggregate metrics benchmark table for DBLP from the overview page of the web application. Aggregate metrics: <=1s, (1s, 5s], >5s are not shown here.

Clicking on an engine row in any benchmark table takes the user to a **Details Page** (Figure 5) for that specific engine and benchmark. The default **Query runtimes tab** lists individual query runtimes and result sizes, giving a clear picture of how the engine performs on each query.

Details - QLever (DBLP)

Query runtimes | Full query | Execution tree | Query result

SPARQL Query	Runtime (s)	Result Size
join-2-small-large	0.02 s	1 [2,000]
join-2-large-small	0.01 s	1 [2,000]
join-2-large-large	0.31 s	1 [54,513,886]
filter-many-results	0.92 s	1 [23,919,950]
optional-join-large-small	0.57 s	1 [118,749,867]
transitive-path-plus	0.07 s	1 [300,322]
number of objects	0.01 s	1 [119,949,931]

Figure 5: Details page (query runtimes tab) for QLever on the DBLP benchmark. Only a subset of queries is shown here.

Clicking on a query row fills the other tabs with information about that query. The **Full query tab** shows the query description and complete SPARQL query text. The

Query result tab shows the query response or the failure message, limited to the `--max-results-output-file` number of results set in `benchmark-queries`. The Execution tree tab is available only for QLever when queries are executed using the special `application/qlever-results+json` format, which provides extra metadata along with the result. A query execution tree is a structured view of how the engine evaluates a query internally, showing the operators and their order. This offers valuable insights into query planning and execution strategy of QLever.

From the overview page, users can also click the Compare Results button to open the **Evaluation page** (Figure 6) for a benchmark. The Evaluation page provides the most comprehensive per-query comparison of SPARQL engines. It shows a table where queries form the rows and engines form the columns. Each cell reports the runtime of a query on a specific engine. Failures are highlighted in red, and the fastest runtime for a given query is shown in green. This makes it immediately clear which engines perform the best for which queries. Clicking on a query cell reveals a tooltip that shows the query description and full SPARQL query, with an option to copy the query text. Similarly, clicking on a failed query cell displays the failure reason.

The evaluation page includes several customization features (as seen in Figure 6) that enhance the comparison process:

- Users can toggle which engines are displayed, making it easy to focus only on the engines of interest.
- Aggregate metrics, such as arithmetic mean, geometric mean, percent of failed queries and median, can be pinned to the top of the table. This provides a constant high-level reference while scrolling through detailed query results.
- Engines can be reordered left to right based on aggregate metrics, either ascending or descending. This helps spot overall performance leaders more quickly. Thanks to Ag Grid, the engines can also be manually dragged and reordered to fit the user's needs.
- Users can choose whether to display result sizes as small, muted text below each runtime. This adds context to performance results without cluttering the main view.

Warnings provide additional insight during evaluation. When an engine's result size for a query differs from the majority, a warning symbol (⚠) appears in that engine's cell. If no consensus exists among engines, the warning symbol is displayed in the query row. The tooltip specifies how the result size deviates from the majority. This consensus-based

DBLP Evaluation results

Show Engines: ☑ QLever ☑ Oxigraph ☑ Virtuoso ☑ Mdb ☐ Jena

Order Engines: Geometric Mean (P=2) ↑

TSV | Show Aggregate Metrics | Show Result Size

Query	QLever	Virtuoso	Mdb	Oxigraph
Geom. Mean (P=2)	**0.29 s**	**0.82 s**	**2.06 s**	**19.18 s**
Failed Queries	**0.00 %**	**2.86 %**	**0.95 %**	**30.48 %**
join-2-small-large	0.02 s	0.07 s	0.08 s	0.28 s
join-2-large-small	0.01 s	0.01 s	0.01 s	⟳timeout
join-2-large-large	0.31 s	1.37 s	6.09 s	⟳timeout
filter-many-results ⚠	0.92 s	1.4 s	3.63 s	⟳timeout
optional-join-large-small	0.57 s	⚠ 2.6 s	41.75 s	⟳timeout
transitive-path-plus	0.06 s	failed	0.01 s	0.01 s
number of objects	0.01 s	timeout	timeout	⟳timeout

Figure 6: Evaluation page of the DBLP benchmark comparing QLever, Virtuoso, MillenniumDB, Oxigraph, and Jena (Jena column hidden for demonstration). Aggregate metrics are shown as bold pinned rows on top. Engines are ordered left to right by increasing geometric mean runtime ($P = 2$). Best per-query runtimes are shown in green, and failures or timeouts in red. ⚠ marks result size deviations: next to an engine if its result size differs from the majority, and next to a query if no result size consensus among the engines. ⟳ indicates that the server was restarted. Only a subset of queries and aggregate metrics is shown.

feedback supports not only performance assessment but also correctness checking, which is often neglected in benchmarks.

If the `benchmark-queries` command was run with the `--restart-on-hang` option, a restart symbol (⟳) is shown for the query that triggered a server restart. The tooltip explains the reason, which can be either a server crash caused by the query or a lack of response beyond the timeout plus 30 s.

A QLever-specific bonus

For a benchmark, when at least two QLever runs include execution tree information, a **Compare Execution Trees** button appears on both the details page and the evaluation page. Clicking on this button takes the user to the **Compare Execution Trees page**. This page allows users to select two different QLever versions from dropdown menus and view their execution trees side by side for the same benchmark query. Hovering over

the query name at the top reveals the full SPARQL query as a tooltip. The trees are automatically scaled to fit on the screen as much as possible, but users can freely adjust the zoom level with + and - buttons. Dragging the cursor to move within an execution tree is also supported. To make comparison easier, a synchronized scroll and zoom feature ensures that moving or zooming one tree also applies the same adjustment to the other. The execution trees comparison functionality is particularly valuable for QLever developers, as it provides a visual way to detect regressions or confirm optimizations between different QLever versions.

5 Evaluation

In this chapter, we use our unified benchmarking framework to evaluate the seven SPARQL engines discussed in Section 4.3. For the evaluation, we run multiple synthetic and real-world benchmarks (from Chapter 2) and compare the performance and scalability of engines at three dataset scales: small (ca. 50 million triples), medium (ca. 500 million triples), and large (ca. 8 billion triples).

It is important to note that the purpose of this evaluation is not to provide a complete performance study of all engines. Instead, it is designed to show how our framework can be used to quickly set up multiple engines, run a wide variety of benchmarks at multiple scales, and compare performance in the evaluation web application. This makes it possible to gather actionable insights into performance and scalability of various SPARQL engines. The full materials needed to reproduce our evaluation can be found at `https://github.com/ad-freiburg/sparql-engine-evaluation-tanmay`.

The next section describes the experimental setup, including engine versions, datasets, benchmarks, and configuration settings.

5.1 Experimental setup

All evaluations were run on a machine running Ubuntu 24.04 LTS, equipped with an AMD Ryzen 9 5900X CPU (12 cores, 24 threads, 3.7 GHz), 128 GiB of DDR4 memory and with 3.6 TB NVMe SSD storage.

5.1.1 Datasets, Benchmarks, and settings (regarding memory and timeout)

To provide a comprehensive evaluation, we use a combination of well-established synthetic and real-world datasets across three distinct scales. For the small and medium-scale experiments, we use synthetic datasets generated with the official data generators for SP^{2}Bench and Watdiv benchmarks. This approach allows for controlled and reproducible

experiments at approximately 50 million and 500 million triples, respectively. We reuse the SP^{2}Bench 50-million-triple index to generate benchmark queries with the Sparqloscope benchmark.

For the medium-scale evaluation using Sparqloscope, we use the DBLP dataset. The DBLP dataset contains quality-checked and manually curated bibliographic information on major computer science journals and proceedings. The specific dump used for our experiments is from September 1st, 2025 [29].

For large-scale evaluation using Sparqloscope and WDBench benchmarks, we employ the Wikidata-truthy dataset. The Wikidata Truthy dataset is a subset of Wikidata, a collaboratively edited knowledge graph hosted by the Wikimedia Foundation. The so-called "truthy" dump only contains direct values of best-rank statements. This dataset represents a massive, heterogeneous, and complex real-world knowledge graph. For this evaluation, we use the version dated June 13th, 2025. Before indexing, we preprocess the dataset by removing all `geo:wktLiteral` instances. This step avoids subtle result size and rounding errors caused by the eight-byte internal representation of geographic literals. The dataset is then split into multiple parts, each containing approximately 100 million triples, to improve indexing performance and prevent out-of-memory errors for Java-based engines.

Table 1 provides an overview of the datasets, benchmarks, and total triples at the three distinct scales.

Triples	**Dataset**	**Benchmark**
50,000,869	SP^{2}Bench data-generator synthetic dataset	SP^{2}Bench v1.1
54,493,332	Watdiv data-generator synthetic dataset	Watdiv v0.6
50,000,869	SP^{2}Bench data-generator synthetic dataset	Sparqloscope
500,000,869	SP^{2}Bench data-generator synthetic dataset	SP^{2}Bench v1.1
546,041,900	Watdiv data-generator synthetic dataset	Watdiv v0.6
524,632,117	DBLP (01.09.2025)	Sparqloscope
7,969,161,598	Wikidata-truthy (13.06.2025)	Sparqloscope
7,969,161,598	Wikidata-truthy (13.06.2025)	WDBench

Table 1: Datasets and benchmarks used in the evaluation. Horizontal separators visually indicate the three dataset scales: small (∼50M triples), medium (∼500M triples), and large (∼8B triples).

The chosen benchmarks are designed to evaluate distinct aspects of SPARQL engine performance, providing a comprehensive view of their capabilities. SP^{2}Bench, Watdiv, Sparqloscope and WDBench are explained in more detail in Chapter 2.

To ensure consistent evaluation across scales, the following memory and timeout settings were applied:

- Small scale (50M triples): 16 GiB RAM, 60 seconds timeout.
- Medium scale (500M triples): 32 GiB RAM, 180 seconds timeout.
- Large scale (8B triples): 64 GiB RAM, 300 seconds timeout.

These configurations provide sufficient resources for engines to complete queries while controlling the maximum runtime for failing queries.

5.1.2 SPARQL Engine Setup and Benchmarking Workflow

All the seven SPARQL engines were set up using their native binaries for the best possible performance. Our evaluation uses QLever at commit baa8421, Virtuoso at version 7.2.15, MillenniumDB at commit 0a41c3b, GraphDB at version 11.0.0, Blazegraph at version 2.1.6 RC, Apache Jena at version 5.5.0 and Oxigraph at version 0.4.11.

The extended benchmarking framework made the setup process uniform and straightforward across all engines. The SPARQL endpoint for each engine was set up using the corresponding wrapper script[1], denoted below as `<qengine>`. At any given time, only one engine's SPARQL endpoint was active to ensure clean and isolated measurements. Before starting experiments for another engine, the currently running server was stopped

The `setup-config` command generates a Qleverfile for the given engine and benchmark. The `setup-config` command used the total index and server memory values, as well as the timeout, specified for each benchmark and scale in Section 5.1.1:

```
<qengine> setup-config <benchmark_name>
                       --total-index-memory <index_mem> \
                       --total-server-memory <server_mem> \
                       --port <PORT> --timeout <timeout> \
                       --system native
```

[1] `<qengine>` denotes the engine-specific wrapper scripts: `qlever`, `qvirtuoso`, `qmdb`, `qgraphdb`, `qblazegraph`, `qjena`, and `qoxigraph`.

Once the configuration file is generated, the benchmark dataset is fetched, indexed, and the SPARQL endpoint is started with:

```
<qengine> get-data
<qengine> index
<qengine> start
```

There are a few exceptions to this pattern. For e.g. `qoxigraph` takes no memory based arguments for the `setup-config` command. And `qgraphdb start` requires a `--license-file-path` argument. However, these exceptions are minimal and do not affect the general workflow of setting up a SPARQL endpoint for the engines.

To run the benchmark, each engine was first warmed up with a single query:

```
<qengine> query  # SELECT * WHERE {?s ?p ?o} LIMIT 10
```

Then, the full benchmark suite for each engine was executed with:

```
<qengine> benchmark-queries --queries-yml <yml_file> \
                            --result-file <benchmark.engine> \
                            --results-dir <result_dir>
```

For all benchmarks except Sparqloscope, the option `--download-or-count count` was also used. This wraps each query to return only its result size rather than downloading the full result set, reducing network and serialization overhead and making the measurements more representative of query execution performance. Sparqloscope was run without this option, since most of its queries already return counts by default and are specifically designed to test fine-grained query behavior in isolation. The `benchmark-queries` command runs all benchmark queries sequentially and stores the resulting YAML file in the specified results directory.

Finally, the results can be visualized interactively via the evaluation web-app. The following command launches the web app at `localhost:8000/www`:

```
<qengine> serve-evaluation-app --results-dir <result_dir>
```

[1] `<qengine>` denotes the engine-specific wrapper scripts: `qlever`, `qvirtuoso`, `qmdb`, `qgraphdb`, `qblazegraph`, `qjena`, and `qoxigraph`.

5.2 Results

In this section, we present the indexing performance and query performance evaluation of the SPARQL engines across the three dataset scales. The purpose here is not to establish a definitive ranking of engines, but to identify performance and scalability trends across benchmarks and dataset scales. Therefore, in the query performance result tables in this section, we focus on two key aggregate metrics: geometric mean runtime (s), and the percentage of failed queries. For failed queries, a penalty of twice the timeout value was applied when computing the geometric mean, ensuring that instability directly impacts the aggregate results. Full results with all aggregate metrics, as well as detailed per-query engine comparisons, are available in the evaluation web application (see `https://github.com/ad-freiburg/sparql-engine-evaluation-tanmay`).

We present the evaluation results separately for the three dataset scales: small (ca. 50M triples), medium (ca. 500M triples), and large (ca. 8B triples). For each scale, we discuss the indexing performance and query performance of individual engines based on aggregate metrics, highlighting how their performance changes as the dataset size increases and how this affects scalability.

5.2.1 Small scale (ca. 50 million triples)

On the small-scale datasets (Table 2), QLever is the clear winner in both indexing time and size, achieving the fastest indexing and the smallest indexes. MillenniumDB and Virtuoso alternate for second place depending on the dataset, showing competitive performance. Oxigraph loads data quickly, but its index size is extremely large and can exceed QLever's by more than a factor of five. The three Java-based engines (GraphDB, Blazegraph, and Jena) are slower and produce larger indexes, with GraphDB consistently outperforming the other two.

Based on the geometric mean and failed query rates in Table 3, Virtuoso emerges as the clear winner on SP^{2}Bench, with MillenniumDB and QLever following. However, this picture is misleading for MillenniumDB. While its aggregate metrics appear competitive, a closer inspection of the per-query results (see evaluation web application) reveals that it produced incorrect result sizes for four queries, giving empty results quickly which makes its aggregate metrics look strong. These errors undermine MillenniumDB's apparent strong aggregate results, bringing its failure rate to 43 %. This makes its effective performance about the same as Jena's and much worse than GraphDB and Blazegraph.

Dataset	Metric	QLV	VRT	MDB	GDB	BLZ	JNA	OXI
SP^2Bench	Index time	72 s	164 s	96 s	221 s	286 s	398 s	103 s
	Index size	1.6 G	4.2 G	2.8 G	8.3 G	6.2 G	6.9 G	11.0 G
Watdiv	Index time	89 s	138 s	100 s	173 s	202 s	411 s	100 s
	Index size	1.5 G	1.5 G	2.4 G	2.8 G	3.2 G	4.5 G	9.9 G

Table 2: Indexing performance for the seven engines (QLV: QLever, VRT: Virtuoso, MDB: MillenniumDB, GDB: GraphDB, BLZ: Blazegraph, JNA: Apache Jena, OXI: Oxigraph) on the SP^2Bench (ca. 50M triples), and Watdiv (ca. 55M triples) datasets. Indexing time is shown in seconds and index size in GiB. Best values (lowest time and smallest size) are highlighted in blue.

Benchmark	Metric	QLV	VRT	MDB	GDB	BLZ	JNA	OXI
SP^2Bench	G. Mean	1.17 s	0.69 s	1.80 s*	2.57 s	2.96 s	3.78 s	29.46 s
	Failed	21.4 %	7.1 %	14.3 %*	28.6 %	21.4 %	35.7 %	57.1 %
Watdiv (100 queries)	G. Mean	0.04 s	0.04 s	0.04 s	0.08 s	0.13 s	0.14 s	0.45 s
	Failed	1.0 %	1.0 %	6.0 %	6.0 %	6.0 %	8.0 %	10.0 %
Sparqloscope SP^2Bench	G. Mean	0.08 s	0.15 s	0.39 s	1.87 s	2.50 s	5.55 s	6.64 s
	Failed	0.0 %	0.0 %	0.0 %	0.0 %	3.3 %	20.0 %	31.1 %

Table 3: Query performance evaluation for the seven engines (QLV: QLever, VRT: Virtuoso, MDB: MillenniumDB, GDB: GraphDB, BLZ: Blazegraph, JNA: Apache Jena, OXI: Oxigraph) on the SP2Bench (ca. 50M triples), Watdiv (ca. 55M triples), and Sparqloscope SP^2Bench (ca. 50M triples) benchmarks. The best geometric mean and failure rate are highlighted in blue. Failed queries are assigned a value of (timeout = 60) $\times$ 2 = 120 s for computing geometric mean and median. * indicates that the reported performance is worse than shown; see the section text for details.

Oxigraph performs the worst on the benchmark, with over half of the queries failing (57 %). Overall for SP^2Bench, Virtuoso is clearly the most reliable, while QLever, Blazegraph, and GraphDB also demonstrate reasonable robustness despite higher failure rates. These results reflect the nature of SP^2Bench benchmark itself. Its benchmark queries with high result cardinalities and complex structures expose weaknesses in query optimization strategies even at the small scale, clearly separating engines with more robust optimizers from those that struggle.

Watdiv presents a contrasting picture. Its queries are lighter but span a wide range of structural patterns and join selectivities, allowing it to expose how engines adapt to structural diversity. MillenniumDB, Virtuoso, and QLever lead in terms of geometric mean runtime, indicating efficient query execution across a variety of query structures. However, MillenniumDB's excellent runtime comes with slightly higher failure rates, which is similar to GraphDB and Blazegraph. Virtuoso and QLever combine speed with

reliability, reinforcing their consistent top-tier performance. Apache Jena and Oxigraph lag behind again, exhibiting both slower runtimes and higher failure rates. Overall at the small scale, all engines perform reliably well on the Watdiv Benchmark.

Sparqloscope is designed for fine-grained analysis where individual query results are most meaningful, yet aggregate metrics still reveal important trends. On the Sparqloscope benchmark at the small scale, QLever, Virtuoso, and MillenniumDB clearly lead, achieving geometric means below 0.4 s with no failures, demonstrating highly efficient query execution across a broad range of SPARQL features. GraphDB and Blazegraph also perform reliably, with almost no failed queries, but their geometric means are noticeably higher at 1.87 s and 2.50 s, respectively. In contrast, Apache Jena and Oxigraph struggle, showing very high failure rates even at this modest scale. Jena fails mainly on `OPTIONAL` and `MINUS` queries, while Oxigraph additionally fails on `EXISTS` and `UNION` constraint queries. At this small scale, Sparqloscope clearly singles out Jena and Oxigraph as the worst performers, exposing specific weaknesses in their handling of complex SPARQL constructs and highlighting the areas where their query processing could be improved.

Across the small-scale benchmarks, Virtuoso and QLever consistently achieve top-tier performance, combining low runtimes with minimal failures. MillenniumDB performs well in terms of speed but exhibits higher failure rates on SP^2Bench, showing some query optimization gaps. GraphDB and Blazegraph are always middle of the pack and deliver reliable but slower performance. Apache Jena and Oxigraph consistently struggle, with highest failure rates and slowest performance on all three benchmarks. Sparqloscope also shows that they are clearly lacking when it comes to complex SPARQL constructs.

NOTE: For Oxigraph, the version used in these experiments (v0.4.11) lacked native timeout handling. Queries exceeding the timeout limit were only terminated after an additional 30 seconds when the server was restarted, negatively affecting benchmark performance. The current release v0.5 has since added proper timeout support.

5.2.2 Medium scale (ca. 500 million triples)

On the medium-scale datasets (Table 4), the indexing time and size trends from the small scale largely persist. QLever again leads with the fastest indexing times and compact indexes, with Virtuoso and MillenniumDB closely following. For Blazegraph (marked with * in Table 4), index time and index size results were obtained using a special, bulk-loading-optimized `RWStore.properties` configuration that is included in our framework. This configuration allows the system to load data in a highly efficient manner, leading to

substantially smaller index sizes and faster indexing times compared to the default setup. In contrast, loading via SPARQL `UPDATE` operations or using a non-optimized default properties file results in significantly larger indexes and much longer loading times, often up to double the values reported here.

Dataset	Metric	QLV	VRT	MDB	GDB	BLZ	JNA	OXI
SP^{2}Bench	Index time	12.1 m	31.4 m	17.6 m	32.8 m	73.9 m*	93.1 m	22.4 m
	Index size	15.3 G	42.0 G	28.0 G	51.0 G	56.0 G*	70.0 G	100.0 G
Watdiv	Index time	16.5 m	24.7 m	17.9 m	28.7 m	92.9 m*	96.7 m	22.2 m
	Index size	17.2 G	17.0 G	24.0 G	28.0 G	31.0 G*	46.0 G	102.0 G
DBLP	Index time	10.3 m	12.4 m	14.7 m	25.4 m	40.3 m*	70.6 m	13.3 m
	Index size	9.4 G	14.0 G	20.0 G	35.0 G	34.0 G*	54.0 G	80.0 G

Table 4: Indexing performance for the seven engines (QLV: QLever, VRT: Virtuoso, MDB: MillenniumDB, GDB: GraphDB, BLZ: Blazegraph, JNA: Apache Jena, OXI: Oxigraph) on the SP^{2}Bench (ca. 500M triples), Watdiv (ca. 550M triples), and DBLP (ca. 525M triples) datasets. Indexing time is shown in minutes and index size in GiB. Best values (lowest time and smallest size) are highlighted in blue. * indicates that additional special steps or problem workarounds were required to obtain the reported indexing measurements; see the section text for details.

Benchmark	Metric	QLV	VRT	MDB	GDB	BLZ	JNA	OXI
SP^{2}Bench	G. Mean	7.89 s	2.93 s	9.96 s*	14.61 s	15.55 s	21.64 s	86.31 s
	Failed	35.7 %	14.3 %	14.3 %*	42.9 %	42.9 %	50.0 %	64.3 %
Watdiv	G. Mean	0.13 s	0.10 s	0.15 s	0.35 s	0.73 s	0.58 s	2.17 s
(100 queries)	Failed	3.0 %	5.0 %	9.0 %	9.0 %	9.0 %	10.0 %	10.0 %
Sparqloscope	G. Mean	0.29 s	0.82 s	2.06 s	9.49 s	11.79 s	23.00 s	18.14 s
DBLP	Failed	0.0 %	2.9 %	1.0 %	5.7 %	14.3 %	27.6 %	30.5 %

Table 5: Query performance evaluation for the seven engines (QLV: QLever, VRT: Virtuoso, MDB: MillenniumDB, GDB: GraphDB, BLZ: Blazegraph, JNA: Apache Jena, OXI: Oxigraph) on the SP^{2}Bench (ca. 500 million triples), Watdiv (ca. 550 million triples), and DBLP Sparqloscope (ca. 525 million triples) benchmarks. The best geometric mean and failed queries % are highlighted in blue. All the failed queries are assigned a value of (timeout: 180 s) * 2 = 360 s for the computation of geometric mean and median. * indicates that the reported metric is worse than shown; see the section text for details.

At the medium scale (Table 5), SP^{2}Bench further stresses the engines, highlighting scalability challenges. Virtuoso remains the clear winner, maintaining the lowest failure rates and geometric mean. This demonstrates that its mature query optimizing techniques

effectively handles high-cardinality queries with increasing scale. QLever experiences a notable performance degradation. Its geometric mean increases several times compared to the small scale, largely due to the failure rate going up from 21 % to 36 %. One particularly demanding query even causes the QLever server to crash. This highlights, how QLever's query planner hasn't covered all the optimization cases yet and lags behind Virtuoso. MillenniumDB again shows strong, but misleading aggregate metrics. It produces wrong result size for the same four queries as the small scale, which raises its effective failure rate to 43 %. This brings its performance on par with GraphDB and Blazegraph, whose runtimes and failure rates also worsen with scale. Oxigraph and Jena exhibit severe scalability limitations, with more than half the queries failing.

For Watdiv on the medium scale, failure rates and geometric means increase slightly compared to the small scale, but the changes are moderate. QLever maintains the lowest failure rate at 5 %, but its geometric mean rises to 0.13 s, slightly above Virtuoso's 0.10 s. This can be attributed to QLever incurring a fixed overhead for each query (most due to the query planner) which shows for the small and medium scale, but is negligible at large scale. MillenniumDB has similar failure rates when compared to the three java-based engines, but much better geometric mean. The three Java-based engines perform reliably well with increasing scale here, but remain clearly behind the leaders in both speed and reliability. Oxigraph remains the worst performer with the worst geometric mean of 2.17 s. Only a hundred randomly selected Watdiv queries from the official workload were evaluated here. A greater number of stress query templates would likely increase the challenge and further differentiate engine performance.

On the medium-scale Sparqloscope benchmark, performance differences between the engines become more pronounced. QLever is the clear scalability winner here with no failures and a geometric mean of only 0.29 s. Virtuoso, and MillenniumDB are close behind, maintaining low failure rates and geometric means as well. In contrast, the Java-based engines and Oxigraph show a clear performance gap. Blazegraph experiences the most severe degradation. Its failure rate rises sharply from 3.3 % at the small scale to 14.3 %. With increasing scale, Blazegraph shows poor performance across the board, with high failure rates on `MINUS` and `EXISTS` queries. For some of these queries, the server failed to terminate within the timeout and had to be restarted. GraphDB's performance also deteriorates, with worse performance on most SPARQL constructs and failures on `EXISTS` queries. Jena and Oxigraph remain the weakest overall, with failure rates around 30 %, though Oxigraph performs somewhat better than Jena, highlighting Jena's worse scalability.

With increasing scale, Virtuoso and QLever continue to be the top performers across all the 3 benchmarks. But, SP^{2}Bench manages to expose some query optimization gaps for QLever at the medium scale, with one query even managing to crash the QLever server. MillenniumDB maintains fast runtimes and consistently places third across all three benchmarks. GraphDB and Blazegraph move further into the middle tier, showing clear signs of strain as runtimes rise and failures increase with scale. Blazegraph especially struggles at the medium scale, showing worse scalability when compared to GraphDB. Apache Jena and Oxigraph continue to be last, with failure rates climbing sharply and scalability issues becoming more apparent. Overall, even at the modest medium scale of 500 million triples, a clear gulf emerges between the top engines (Virtuoso, QLever, MillenniumDB) and the weaker group of Java-based engines plus Oxigraph.

5.2.3 Large scale (ca. 8 billion triples)

The large-scale evaluation (Table 7) uses the Wikidata-truthy dataset and two real-world benchmarks: Sparqloscope and WDBench. Oxigraph is excluded from this evaluation due to excessive index size, missing query optimizations, and lack of timeout support in version 0.4.11.

Dataset	Metric	QLV	VRT	MDB	GDB	BLZ	JNA	OXI
Wiki-truthy	Index time	3.5 h	13.8 h*	4.2 h	>20 h†	22.6 h*	21.0 h	×
	Index size	149 G	373 G	317 G	453 G	500 G*	684 G	×

Table 6: Indexing performance for the seven engines (QLV: QLever, VRT: Virtuoso, MDB: MillenniumDB, GDB: GraphDB, BLZ: Blazegraph, JNA: Apache Jena, OXI: Oxigraph) on the Wikidata Truthy dataset (ca. 8B triples). Indexing time is shown in hours and index size in GiB. Best values (lowest time and smallest size) are highlighted in blue. †GraphDB was restarted from a checkpoint after a crash and the reported time is approximate. * indicates that additional special steps or problem workarounds were required to obtain the reported indexing measurements; see the section text for details. Oxigraph was excluded from this scale and is shown as ×.

On the large-scale Wikidata-truthy dataset (Table 6), QLever remains the undisputed winner, achieving the fastest indexing time and a compact index of 149.4 GiB. Unlike the small and medium-scale datasets, MillenniumDB now clearly secures second place, with much faster indexing and a smaller index than Virtuoso. Virtuoso lags behind with longer loading times and a larger index of 373 GiB. As shown in Table 6, the indexing process for Virtuoso (marked with *) required special handling. In particular, the data was loaded in incremental chunks and committed progressively to avoid extreme slowdowns

Benchmark	Metric	QLV	VRT	MDB	GDB	BLZ	JNA	OXI
Sparqloscope	G. Mean	3.94 s	14.99 s	35.34 s	115.62 s	103.05 s	165.33 s	×
Wiki-truthy	Failed	2.9 %	15.2 %	40.0 %	61.9 %	57.1 %	67.6 %	×
WDBench	G. Mean	1.39 s	8.33 s*	3.54 s	12.81 s	47.25 s	93.18 s	×
(100 queries)	Failed	7.0 %	24.0 %*	12.0 %	23.0 %	33.0 %	47.0 %	×

Table 7: Query performance evaluation for the seven engines (QLV: QLever, VRT: Virtuoso, MDB: MillenniumDB, GDB: GraphDB, BLZ: Blazegraph, JNA: Apache Jena, OXI: Oxigraph) on the Wikidata Truthy (ca. 8 billion triples) benchmarks created using Sparqloscope and WDBench. The best geometric mean and failed queries % are highlighted in blue. All the failed queries are assigned a value of (timeout: 300 s) * 2 = 600 s for the computation of geometric mean and median. At the large scale, only the Wikidata-truthy index was built for each engine, so the synthetic benchmarks SP^{2}Bench and Watdiv are not applicable. * Indicates that the reported performance is worse than shown; see the section text for details. Oxigraph was excluded from this scale and is shown as ×.

or crashes and to ensure the process completed successfully. The Java-based engines (GraphDB, Blazegraph, and Jena) are far slower, produce massive indexes, and take over 20 hours to complete. As discussed earlier in Section 5.2.2, the index time and size values for Blazegraph (marked with *) reflect the use of the bulk-loading-optimized `RWStore.properties` configuration. Without this optimization, both the indexing time and index size would be considerably higher. Oxigraph is excluded due to potential excessive index size at this large scale. Overall, the large-scale results starkly highlight QLever's superior scalability and efficiency, while the gulf to the other engines, particularly Virtuoso and the Java-based engines, becomes much clearer.

On Sparqloscope, the gulf between engines widens dramatically compared to the medium scale. QLever cements itself as the scalability champion, with a geometric mean rising only modestly (from 0.29 s to 3.94 s) and the lowest failure rate of just 2.9 %. This stability highlights the efficiency of its join algorithms, compressed index blocks, and targeted optimizations. Virtuoso follows, with stronger results than most competitors, especially on `OPTIONAL`, `MINUS`, and `EXISTS` queries where its relational database foundation pays off. However, it scales less gracefully, as its geometric mean grows by nearly a factor of 20 (0.8 s to 15 s), and its failures rise to 15 %. Virtuoso especially struggles on property path queries, where it consistently fails. MillenniumDB's performance degrades considerably at the large scale, with its failure rate jumping from 1 % to 40 %. The Java-based engines all collapse under the large-scale workload. GraphDB, Blazegraph, and Jena, which had already struggled at medium scale, now fail on more than 60 % of queries and slow down by an order of magnitude, becoming effectively unusable. Compared to Sparqloscope at medium scale, the results at large scale reveal how scalability magnifies differences.

Engines that seemed only "weaker" at medium scale become practically infeasible at large scale.

WDBench also paints a similar picture, but introduces an unexpected outcome. While QLever again leads the pack, MillenniumDB surpasses Virtuoso on this benchmark, marking one of the few cases where Virtuoso does not occupy a top position. Virtuoso's weakness is explained by its poor handling of property path and navigational graph pattern queries, where it frequently fails, times out, or returns incorrect result sizes. For at least ten queries from these two categories, it returns result sizes that differ significantly from all other engines that complete the queries successfully. One query also manages to crash the Virtuoso server, requiring a restart to continue the evaluation. If these ten queries are counted as failed, Virtuoso's failure rate increases to 34 %, placing it at the same level as Blazegraph. Since Virtuoso is generally faster than Blazegraph on the queries it completes, its overall performance would then be worse than GraphDB but still better than Blazegraph. In contrast, MillenniumDB proves surprisingly strong on WDBench, suggesting that its query engine is more resilient for complex, heterogeneous graph workloads. GraphDB performs reasonably well and is much more reliable and faster compared to the other two java engines. Blazegraph and Jena perform quite poorly and are slower by an order of magnitude and face substantially higher failure rates.

Taken together, the large-scale results show that only a few engines remain viable at billion-triple scale. QLever is the most scalable and reliable engine, providing consistent speed, low failure rates, and robustness across query types. Virtuoso performs well on many SPARQL constructs and benefits from its relational design. However, it repeatedly fails on property path and navigational graph pattern queries and is therefore unsuitable for workloads that depend on these features. MillenniumDB handles property path and navigational queries more robustly, which explains its stronger results on WDBench. But, Sparqloscope shows that MillenniumDB performs worse compared to QLever and Virtuoso across most other SPARQL constructs. The three Java-based engines, GraphDB, Blazegraph, and Apache Jena, exhibit high failure rates and slow runtimes at this scale and are effectively impractical for billion-scale deployments. Overall, Virtuoso and MillenniumDB have specific strengths but remain a considerable step below QLever in overall performance and reliability at this large 8-billion-triple scale.

6 Conclusion

In this thesis, we have presented a unified framework for benchmarking SPARQL engines that simplifies and standardizes the evaluation process. By extending QLever-control to support seven SPARQL engines (QLever, Virtuoso, MillenniumDB, GraphDB, Blazegraph, Apache Jena, and Oxigraph), we have enabled users to set up endpoints, and run benchmarks without having to learn the internal commands or configuration details of each engine. Datasets can be retrieved and indexed automatically, benchmarks can be executed in a uniform way, and results can be explored through an interactive evaluation application instead of static tables.

Through an evaluation across multiple synthetic and real-world benchmarks, we have showed how the framework supports comparisons of query runtimes, result size correctness, index build times, and index sizes at different dataset scales. The evaluation revealed many interesting strengths and weaknesses of the engines. Users can use these insights to identify engines that not only meet their current needs but can also scale to larger workloads. Researchers and developers who work with SPARQL engines benefit from a reproducible environment in which they can easily test their systems against established baselines, validate improvements, and demonstrate competitive performance across multiple benchmarks.

With this work, we have taken a step toward making SPARQL benchmarking more reproducible, reliable, and accessible. All software, benchmarks and results are publicly available on `https://github.com/ad-freiburg/sparql-engine-evaluation-tanmay`.

The next section discusses directions for future work that can further extend the scope and applicability of the framework.

6.1 Future Work

The framework presented in this thesis is designed to be extensible, and several directions can further increase its utility and usability.

Support for More Engines: The framework is designed to be very easy-to-extend and Section 4.1, and Section 4.2 can be used as guidance to add support for new engines. Future work will expand coverage by integrating additional engines, ensuring broader applicability for both users and researchers.

Automated Collection of Index Statistics: Currently, QLever-control provides an `index-stats` command that reports index size and index time for QLever. For the other engines, these values were retrieved manually during our evaluation in Section 5.2. A useful extension would be to implement an `index-stats` command for each supported engine so that index size and index time can be collected automatically.

Integration of Index Statistics into the Web Application: Once index statistics are gathered automatically, they can be consumed directly by the evaluation web application. This would enable users to explore and compare not only query runtimes and result sizes, but also index times and storage requirements in a unified interface.

Guidance for Native Installation: The framework currently supports automated container-based setups, while native binary installation is left to the user. To make this easier, a new command could be added for each engine. This command would print clear, engine-specific instructions for downloading, installing, and placing the required binaries on the system path.

Continuous Integration and Regression Testing: A valuable extension would be to integrate the framework with continuous integration (CI) pipelines such as GitHub Actions. This would allow automatic benchmarking to be triggered whenever a new version of an engine is pushed to its repository. Such integration would enable regression testing, ensuring that new releases do not degrade query performance, scalability, or correctness. Over time, this could also build a historical record of performance trends across engine versions, providing useful insights for developers.

7 Acknowledgments

I would like to express my sincere gratitude to Prof. Dr. Hannah Bast for her continuous guidance, insightful discussions, and valuable suggestions throughout the course of this thesis. Her mentorship and encouragement have been essential to the successful completion of this work.

I want to thank Prof. Dr. Christian Schindelhauer who agreed to act as a second reviewer for this thesis.

I am also thankful to Johannes Kalmbach for supervising the initial project that inspired and evolved into this thesis, and for sharing thoughtful ideas that helped shape its direction.

I would like to thank my friends Rudradeep, Sankha, and Mohit for taking the time to proofread my work and offering valuable comments and suggestions for improvement.

Finally, I am deeply grateful to my girlfriend Ambre and my parents for their constant support and encouragement throughout my master's studies.

Bibliography

[1] A. Hogan, E. Blomqvist, M. Cochez, C. D'amato, G. D. Melo, C. Gutierrez, S. Kirrane, J. E. L. Gayo, R. Navigli, S. Neumaier, A.-C. N. Ngomo, A. Polleres, S. M. Rashid, A. Rula, L. Schmelzeisen, J. Sequeda, S. Staab, and A. Zimmermann, "Knowledge graphs," *ACM Computing Surveys*, vol. 54, no. 4, p. 1–37, Jul. 2021. [Online]. Available: http://dx.doi.org/10.1145/3447772

[2] W. R. W. Group, "Resource description framework (rdf): Concepts and abstract syntax," *W3C Recommendation*, 2014. [Online]. Available: https://www.w3.org/TR/rdf11-concepts/

[3] E. Prud'hommeaux, S. Harris, and A. Seaborne, "Sparql query language for rdf," *W3C Recommendation*, 2013. [Online]. Available: https://www.w3.org/TR/sparql11-query/

[4] T. Pellissier Tanon, G. Weikum, and F. Suchanek, "Yago 4: A reason-able knowledge base," in *The Semantic Web: 17th International Conference, ESWC 2020, Heraklion, Crete, Greece, May 31–June 4, 2020, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2020, p. 583–596. [Online]. Available: https://doi.org/10.1007/978-3-030-49461-2_34

[5] C. to Wikimedia, "Wikidata rdf dump format," Jul 2025. [Online]. Available: https://www.mediawiki.org/wiki/Wikibase/Indexing/RDF_Dump_Format

[6] T. U. Consortium, "Uniprot: the universal protein knowledgebase in 2025," *Nucleic Acids Research*, vol. 53, no. D1, pp. D609–D617, 11 2024. [Online]. Available: https://doi.org/10.1093/nar/gkae1010

[7] W. Ali, M. Saleem, B. Yao, A. Hogan, and A. N. Ngomo, "A survey of RDF stores & SPARQL engines for querying knowledge graphs," *VLDB J.*, vol. 31, no. 3, pp. 1–26, 2022.

[8] T. Pellissier Tanon *et al.*, "Oxigraph/oxigraph: Sparql graph database." [Online]. Available: https://github.com/oxigraph/oxigraph

[9] B. B. Thompson, M. Personick, and M. Cutcher, "The bigdata® rdf graph database," in *Linked Data Management*, A. Harth, K. Hose, and R. Schenkel, Eds. Chapman and Hall/CRC, 2014, pp. 193–237. [Online]. Available: http://www.crcnetbase.com/doi/abs/10.1201/b16859-12

[10] O. Erling, "Virtuoso, a Hybrid RDBMS/Graph Column Store," *IEEE Data Eng. Bull.*, vol. 35, no. 1, pp. 3–8, 2012.

[11] M. Saleem, G. Szárnyas, F. Conrads, S. A. C. Bukhari, Q. Mehmood, and A. N. Ngomo, "How Representative Is a SPARQL Benchmark? An Analysis of RDF Triplestore Benchmarks," in *WWW*. ACM, 2019, pp. 1623–1633.

[12] H. Bast *et al.*, "Qlever-control," https://github.com/ad-freiburg/qlever-control.

[13] 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*, ser. CIKM '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 647–656. [Online]. Available: https://doi.org/10.1145/3132847.3132921

[14] D. Vrgoc, C. Rojas, R. Angles, M. Arenas, D. Arroyuelo, C. Buil-Aranda, A. Hogan, G. Navarro, C. Riveros, and J. Romero, "MillenniumDB: A Persistent, Open-Source, Graph Database," *CoRR*, vol. abs/2111.01540, 2021.

[15] "Ontotext: Graphdb." [Online]. Available: https://graphdb.ontotext.com/

[16] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson, "Jena: implementing the semantic web recommendations," in *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers & Posters*, ser. WWW Alt. '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 74–83. [Online]. Available: https://doi.org/10.1145/1013367.1013381

[17] C. Bizer and A. Schultz, "The Berlin SPARQL Benchmark," *Int. J. Semantic Web Inf. Syst.*, vol. 5, no. 2, pp. 1–24, 2009. [Online]. Available: http://wbsg.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark

[18] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A benchmark for OWL knowledge base systems," *J. Web Semant.*, vol. 3, no. 2-3, pp. 158–182, 2005. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=aeedd9f5c0fe4ca3e850cfbf7425e3f312e30cf7

[19] G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee, "Diversified Stress Testing of RDF Data Management Systems," in *ISWC (1)*, ser. Lecture Notes in

Computer Science, vol. 8796. Springer, 2014, pp. 197–212. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-11964-9_13

[20] M. Schmidt, T. Hornung, M. Meier, C. Pinkel, and G. Lausen, "SP^{2}Bench: A SPARQL Performance Benchmark," in *Semantic Web Information Management*. Springer, 2009, pp. 371–393. [Online]. Available: https://arxiv.org/abs/0806.4627

[21] A. Hogan, C. Riveros, C. Rojas, and A. Soto, "A Worst-Case Optimal Join Algorithm for SPARQL," in *ISWC (1)*, ser. Lecture Notes in Computer Science, vol. 11778. Springer, 2019, pp. 258–275. [Online]. Available: https://aidanhogan.com/docs/SPARQL_worst_case_optimal.pdf

[22] R. Angles, C. Buil-Aranda, A. Hogan, C. Rojas, and D. Vrgoc, "WDBench: A Wikidata Graph Query Benchmark," in *ISWC*, ser. Lecture Notes in Computer Science, vol. 13489. Springer, 2022, pp. 714–731. [Online]. Available: https://iswc2022.semanticweb.org/wp-content/uploads/2022/11/978-3-031-19433-7_41.pdf

[23] M. Saleem, Q. Mehmood, and A.-C. Ngonga Ngomo, "Feasible: A feature-based sparql benchmark generation framework," in *The Semantic Web - ISWC 2015*, M. Arenas, O. Corcho, E. Simperl, M. Strohmaier, M. d'Aquin, K. Srinivas, P. Groth, M. Dumontier, J. Heflin, K. Thirunarayan, K. Thirunarayan, and S. Staab, Eds. Cham: Springer International Publishing, 2015, pp. 52–69. [Online]. Available: https://doi.org/10.1007/978-3-319-25007-6_4

[24] M. Saleem, A. Akhter, S. Vahdati, and A.-C. Ngonga Ngomo, "mu-bench: Real-world micro benchmarking for sparql query processing over knowledge graphs," in *Proceedings of the 11th International Joint Conference on Knowledge Graphs*, ser. IJCKG '22. New York, NY, USA: Association for Computing Machinery, 2023, p. 39–47. [Online]. Available: https://doi.org/10.1145/3579051.3579054

[25] H. Bast, J. Kalmbach, C. Ullinger, and R. Textor-Falconi, "Sparqloscope: A generic benchmark for comprehensive performance evaluation of sparql engines," 2025. [Online]. Available: https://ad-publications.cs.uni-freiburg.de/ISWC_sparqloscope_BKTU_2025.pdf

[26] F. Conrads, J. Lehmann, M. Saleem, M. Morsey, and A.-C. Ngonga Ngomo, "Iguana: A generic framework for benchmarking the read-write performance of triple stores," in *The Semantic Web – ISWC 2017*, C. d'Amato, M. Fernandez, V. Tamma, F. Lecue, P. Cudré-Mauroux, J. Sequeda, C. Lange, and J. Heflin, Eds. Cham: Springer International Publishing, 2017, pp. 48–65. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-68204-4_5

[27] R. Taelman, J. Van Herwegen, M. Vander Sande, and R. Verborgh, "Comunica: A modular sparql query engine for the web," in *The Semantic Web – ISWC 2018*, D. Vrandečić, K. Bontcheva, M. C. Suárez-Figueroa, V. Presutti, I. Celino, M. Sabou, L.-A. Kaffee, and E. Simperl, Eds. Cham: Springer International Publishing, 2018, pp. 239–255. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-00668-6_15

[28] M. R. Ackermann, H. Bast, B. M. Beckermann, J. Kalmbach, P. Neises, and S. Ollinger, "The dblp Knowledge Graph and SPARQL Endpoint," *TGDK*, vol. 2, no. 2, pp. 3:1–3:23, 2024.

[29] dblp Team, "dblp computer science bibliography – Monthly Snapshot RDF/N-Triple Release of September 2025," September 2025. [Online]. Available: https://doi.org/10.4230/dblp.rdf.ntriples.2025-09-01