Semester Project: C++ Benchmarking Framework for Consistent Hashing Algorithms in Non-peer-to-peer Contexts

Supervisor: *Amos Brocco* *Student:* *Roberto Vicario*
Information Systems and Networking Institute, Department of Innovative Technologies
University of Applied Sciences and Arts of Southern Switzerland

Abstract

Within the scope of a research project, conducted under the guidance of Massimo Coluzzi, a *framework in Java* [7] was developed for benchmarking state-of-the-art consistent hashing algorithms. In order to explore the performance of these algorithms using different languages, I was tasked with creating a version in C++. This version currently encompasses only a partial selection of the technologies considered in the ISIN framework. However, the tool has been designed adaptively to easily implement new algorithms and benchmarks. The software remains consistent with the behavior of its Java counterpart. It is designed to operate via a command-line interface and to save benchmark results in a CSV file, utilizing YAML files for parameter configuration.

Keywords *algorithms, anchor, consistent-hashing, cpp, csv, dx, jump, maglev, memento, power, ring, yaml*

1 Introduction

In this paper, we delve into the development and application of a benchmarking tool tailored for evaluating consistent hashing algorithms. Rather than a step-by-step breakdown, we embark on a journey through the landscape of current algorithms and tools, exploring their nuances and implications. Central to our exploration are the metrics employed to gauge algorithmic efficacy, interwoven with the intricacies of computation and analysis. With a focus on both practicality and theoretical depth, we unveil the implementation process of our tool, aiming to provide a holistic understanding of its mechanics. Lastly, we navigate through the results obtained, contemplating avenues for future refinement and expansion of the tool's capabilities.

2 State-of-the-art

The ISIN Institute at SUPSI has created a robust Java framework designed to generate benchmarks for state-of-the-art consistent hashing algorithms. This framework computes different benchmarks using different algorithms and hash functions. It allows for the adjustment of various aspects of algorithm behavior through a configuration YAML file. Parameters are passed through the command line, enabling modifications to aspects like the number of initialization nodes or the iteration count. Throughout execution, metrics such as the mean are calculated for each benchmark, and the related results are stored in a CSV file. This tool serves as a comprehensive benchmarking framework for distributed scenarios, representing one of the first of its type in the domain of open-source software.

2.1 Consistent Hashing Algorithms

These algorithms play a crucial role in distributed systems and represent some of the most famous methods in state-of-the-art of consistent hashing:

- *Ring by Karger et al. (1997)* [8]: It optimizes resource usage, and scale seamlessly with network expansion.
- *Jump by Lamping et al. (2014)* [9]: It is the most suited for data storage scenarios due to its requirement of sequentially numbered buckets, making it less ideal for distributed web caching.
- *Maglev by Eisenbud et al. (2016)* [6]: It used for high packet processing performance to manage substantial and growing traffic.
- *Anchor by Mendelson et al. (2020)* [11]: It is a robust and scalable hashing algorithm crucial for networking applications like maintaining TCP flow connection affinity.
- *Dx by Dong et al. (2021)* [5]: It emerges as a breakthrough in consistent hashing, critical for data routing and load balancing in various domains like distributed databases and peer-to-peer networks.
- *Power by Leu et al. (2023)* [10]: It minimizes key remapping during bucket count alterations, crucial for load balancing, distributed caching, and key-value stores.
- *Memento by Coluzzi et al. (2023)* [4]: It aims to evenly distribute data across nodes with optimal performance and minimal memory footprint.

The Java framework has essentially incorporated the same algorithms, though there remains the possibility of implementing new ones in both cases.

3 Benchmarking

As outlined in *”A survey and comparison of consistent hashing algorithms” by Coluzzi et al. (2023)* [3], here is a concise overview of the benchmarks utilized:

- *Balance*: The ability of the algorithm to spread the keys evenly across the cluster nodes.
- *Initialization Time*: The time the algorithm requires to initialize its internal structure.
- *Lookup Time*: The time the algorithm needs to find the node a given key belongs to.
- *Memory Usage*: The amount of memory the algorithm uses to store its internal structure.
- *Monotonicity*: The ability of the algorithm to move the minimum amount of resources when the cluster scales.
- *Resize Balance*: The ability of the algorithm to keep its balance after adding or removing nodes.
- *Resize Time*: The time the algorithm requires to reorganize its internal structure after adding or removing nodes.

To assess the performance of each algorithm, we rely on these benchmarks, which primarily gauge the behavior and efficiency of a consistent hashing algorithm.

3.1 Hash Functions

Currently, the framework exclusively utilizes *CRC32* for each algorithm to hash the keys. To incorporate additional functions, one simply needs to uncomment the loop section within the benchmark routine in the code. In the referenced framework, the available hash functions for benchmark computation include *CRC32, MD5, Murmur, and XXHash*.

3.2 Metrics

For each iteration will be executed a certain number of runs, as reported in the configuration file. Subsequently, will be saved into results file the *mean, the variance, and the standard deviation* computed on each run.

4 Framework

My project involved constructing a C++ framework with features comparable to its Java equivalent. The *C++ framework* [2] is presently accessible on GitHub and is distributed under the GNU General Public License version 3.

4.1 Implementation

This subsection aims to explain the key elements of the framework, providing insight into its functionality. For project configuration, we employed *CMake, vcpkg, and Ninja*. The system implementation comprised `Main.cpp` and `Routine.hpp` to orchestrate benchmarking processes. *YAML* configuration facilitated the setup of routines, with results saved to a *CSV* file via `HandlerImpl.hpp`.

Initially, configuring the setup services is essential to kickstart the project. For convenience, two bash scripts, `repo.sh` and `cmake.sh` were added in the root directory to streamline this process. Once configured, simply navigate to the build directory and execute the *Ninja* command. Detailed instructions are available in the corresponding repository documentation.

Following the initial setup, the project was organized into various directories. The primary application entry point resides in the `Main.cpp` file at the root level. Algorithms are implemented within the `Algorithms/` directory, while hash functions are located within a `misc/` folder within `Algorithms/`. Benchmarks implementations are located in the corresponding `Benchmarks/` folder, accompanied by the `Routine.hpp` file, which manages flow control in conjunction with `Main.cpp`.

The benchmark routine's execution configuration is managed through YAML files located in the `configs/` folder. Another crucial component for implementation is the *Handler*, which is invoked by the core orchestrator to manage CSV file creation and updates. The Handler's implementation is stored in the `Handler/` folder.

4.2 Configuration

The format of the configuration file is described in detail in the `configs/template.yaml` file. The tool will use the `configs/default.yaml` file that represents the default configuration if no configuration file is provided.

4.3 Flow Control

Figure 1 shows a UML sequence diagram to explain how the benchmark routine procedure works.

We will call *Core* as the entity which handle the main flow in the `Main.cpp` and `Routine.hpp` files within the code; *Core* is responsible for orchestrating the benchmarking process. The *Core* interacts with the *Engine* boundary, which corresponds to the algorithm implementation, which performs the computations based the algorithm logic. Finally, the system utilizes *YAML* and *CSV* files for configuration and data storage, respectively.

The benchmark routine follows a nested loop structure. Initially, the *Core* entity requests the *YAML* configuration, which includes algorithms, benchmarks, hash functions, and initialization nodes. Then, it iterates through each combination of these parameters.

Within the nested loops, the *Core* entity sends the current combination of parameters to the *Engine* to compute the benchmarks using the current algorithm. After receiving the results, it updates metrics in the *CSV* file.

This routine repeats for all possible combinations of parameters specified in the *YAML* configuration, for the specified number of iterations. Finally, the *CSV* file is updated with the computed metrics, *the mean, the variance, and the standard deviation*, and the stream is flushed and closed.

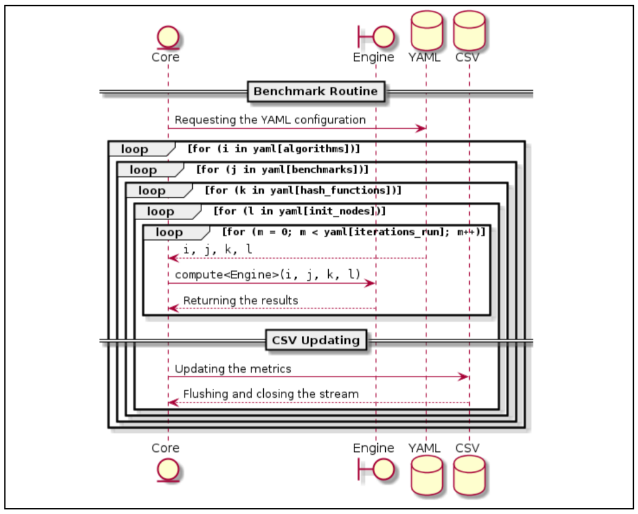
Core
Engine
YAML
CSV
Benchmark Routine
Requesting the YAML configuration
loop [for (i in yaml[algorithms])]
loop [for (j in yaml[benchmarks])]
loop [for (k in yaml[hash_functions])]
loop [for (l in yaml[init_nodes])]
loop [for (m = 0; m < yaml[iterations_run]; m++)]
i, j, k, l
compute<Engine>(i, j, k, l)
Returning the results
CSV Updating
Updating the metrics
Flushing and closing the stream
Core
Engine
YAML
CSV

Figure 1: Exploring the flow control of the benchmark routine.

4.4 Comparing Languages Features

This section aims to analyze Java and C++ features in order to justify the decision to implement the same framework in another language. The evaluation will be based on benchmark criteria, focusing specifically on the time and memory performance of both languages.

Time Features Java relies on JIT compilation, which translates bytecode to native machine code at runtime. While JIT compilation can lead to improved performance after warm-up, it may introduce variability in benchmark results due to factors like optimization thresholds and compilation overhead. C++ code is typically compiled ahead of time (AOT), allowing for predictable performance characteristics and minimizing runtime overhead associated with dynamic compilation.

Memory Features Java's garbage collection automatically manages memory allocation and deallocation, simplifying memory management for developers. However, garbage collection pauses may introduce non-deterministic behavior and overhead, impacting memory-related benchmarks. C++ allows for manual memory management using `new` and `delete` operators, offering more control over memory allocation and deallocation. While manual management can reduce overhead and improve determinism, it requires careful attention to avoid memory leaks and segmentation faults.

4.5 New Features

To implement new features is important to follow the procedures for adding new algorithms and benchmarks to the system. Follow the instructions below to seamlessly integrate your custom algorithms and benchmarks into the existing framework.

Adding New Algorithms Insert the algorithm name into any configuration file located in `configs/`. Implement your algorithm in `Algorithms/your_algo/`. Keep in mind that the system employs C++ templates to integrate the algorithms into the loop. Integrate a new execution routine into `Main.cpp`. Append a new `else if` branch and incorporate your engine using: `execute<YourEngine>("your_algo", handler, yaml);`

Adding New Benchmarks Insert the benchmark name into any configuration file located in `configs/`. Implement the benchmark in `Benchmarks/`. Create a function named `computeYourBenchmark` within it, accepting parameters `string algorithm` and `uint32_t initNodes`. Note that the system utilizes C++ templates for benchmark integration into the loop. Integrate a new benchmark routine into `Benchmarks/Routine.hpp`. Append a new `else if` branch and incorporate your engine.

5 Results

Upon thorough examination of all benchmarks, we assessed every algorithm according to various metrics and categorized them into three performing groups: best, average, and worst. These classifications are illustrated in tables 1 and 2. The evaluation is conducted using a simulated cluster with 100000 nodes.

Among the time metrics analyzed, the algorithm demonstrating the best performance is *Power*, followed with good results by *Anchor*, *Dx*, and *Jump*. Concerning memory usage, these four algorithms exhibit more or less similar efficient performance.

This is logical because *Power* boasts an expected time complexity of $\mathcal{O}(1)$ for key lookup, regardless of the number of buckets, and hash values are computed in real-time. It is touted as faster than well-known consistent hash algorithms, which typically have a lookup time complexity of $\mathcal{O}(\log n)$. [10]. On the other hand, *Jump* eschews internal data structures, resulting in minimal memory usage and unmatched speed across all metrics. However, it comes with the notable limitation that only the last inserted node can be removed. In contrast, *Anchor* and *Dx* tackle this limitation by employing an internal data structure to track the cluster's nodes. This approach slightly slows down initialization and resizing but ensures similar performance in lookup operations [3].

Best	Average	Worst
Power	Anchor Dx Jump	Memento

Table 1: Ranking the algorithms time performance of the algorithms.

Best	Average	Worst
Power Jump	Anchor Dx	Memento

Table 2: Ranking the algorithms memory usage of the algorithms.

Last benchmarks examinated are balance, resize balance and monotonicity. I esaminated these benchmark as one single group because measure the algorithm's capacity to efficiently utilize resources, maintain balance, and adapt seamlessly to changes in cluster size while minimizing overhead. Figure 2 illustrates the progression of balance benchmarks for each initialization node value. Regarding performance ranking, *Power* emerges as the most effective, but even the others demonstrate a good balance. The results for the resize balance are similar to those of balance, maintaining the precedent ranking. Comparing these benchmarks to those in the Java framework, we find a parallel ranking, with the strong difference that within the Java tool the *Power* algorithm is not implemented yet. In the case of monotonicity, all algorithms yield identical results, warranting classification as equally proficient, mirroring the rankings observed in the Java framework.

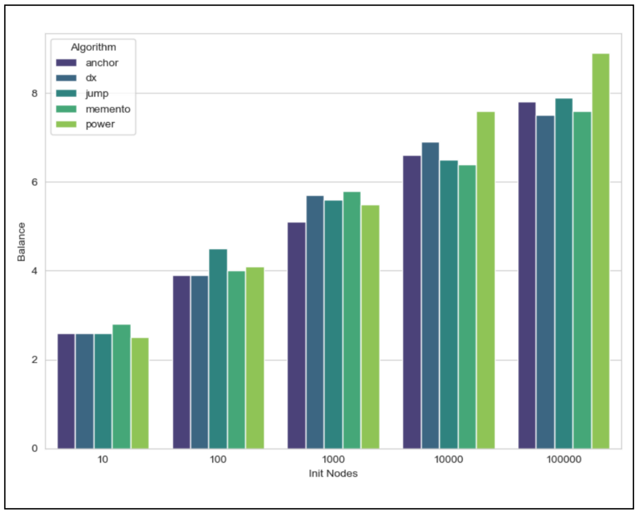

Figure 2: Evaluating the balance of each algorithm on initialization nodes change.

6 Conclusion

This paper provides a comprehensive view of the semester project I conducted, focusing on the development of a C++ benchmarking framework tailored for consistent hashing algorithms within non-peer-to-peer contexts.

The aim is to offer a thorough overview of the project, for future works and researchers with an interest in the subject. I delved into theoretical concepts that aided my studying throughout the semester, explaining how the framework could produce benchmarks effectively.

Currently, only a few algorithms remain unimplemented within this project, which presents an opportunity to compare new solutions with the existing ones. Although the *Maglev* and *Ring* algorithms have been partially implemented, they are not fully integrated into the system and are hosted in dedicated branches for future works. It could be useful to also implement the *Multi-probe by Appleton et al. (2015)* [1] and *Rendezvous by Thaler et al. (1998)* [12] algorithms to facilitate a more comprehensive comparison than what was achieved in the *research* of my supervisors [3].

References

[1] Ben Appleton and Michael O'Reilly. "Multi-probe consistent hashing". In: *arXiv preprint arXiv:1505.00062* (2015).

[2] Amos Brocco and Roberto Vicario. *cpp-consistent-hashing-algorithms*. Version 0.1.0. DOI: `https://github.com/slashdotted/cpp-consistent-hashing-algorithms`. URL: `https://github.com/slashdotted/cpp-consistent-hashing-algorithms`.

[3] Massimo Coluzzi et al. "A survey and comparison of consistent hashing algorithms". In: *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2023, pp. 346–348.

[4] Massimo Coluzzi et al. "MementoHash: A Stateful, Minimal Memory, Best Performing Consistent Hash Algorithm". In: *IEEE/ACM Transactions on Networking* (2024).

[5] Chao Dong, Fang Wang, and Dan Feng. "Dxhash: A scalable consistent hash based on the pseudo-random sequence". In: *arXiv preprint arXiv:2107.07930* (2021).

[6] Daniel E Eisenbud et al. “Maglev: A fast and reliable software network load balancer.” In: *Nsdi*. Vol. 16. 2016, pp. 523–535.

[7] Institute of Information Systems Networking – Department of Innovative Technologies – University of Applied Sciences Arts of Southern Switzerland. *java-consistent-hashing-algorithms*. 2021. URL: `https://github.com/SUPSI-DTI-ISIN/java-consistent-hashing-algorithms.git`.

[8] David Karger et al. “Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web”. In: *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. 1997, pp. 654–663.

[9] John Lamping and Eric Veach. “A fast, minimal memory, consistent hash algorithm”. In: *arXiv preprint arXiv:1406.2294* (2014).

[10] Eric Leu. “Fast Consistent Hashing in Constant Time”. In: *arXiv preprint arXiv:2307.12448* (2023).

[11] Gal Mendelson et al. “Anchorhash: A scalable consistent hash”. In: *IEEE/ACM Transactions on Networking* 29.2 (2020), pp. 517–528.

[12] David G Thaler and Chinya V Ravishankar. “Using name-based mappings to increase hit rates”. In: *IEEE/ACM Transactions on networking* 6.1 (1998), pp. 1–14.