

High Performance Document Store in Rust

By
Ishaan Aggarwal

Advisor: Dr. Chris Pollett
Committee : Dr. Robert Chun
Mr. Akshay Kajale

Agenda

- Problem Statement & Introduction
- Important Concepts
- System Architecture & Work Flow
- Individual Components & Technology Choices
- Applications & Dataset
- Experiments & Results
- Challenges & Learnings
- Conclusion

Problem Statement

- There is a lack of database systems which focus more on high read speeds.
- We aimed to build a high performance database system which:
 - Ensures fast read speeds
 - Can manage high volume of data
 - Allows indexing on the data
 - Can be used as a backend engine for variety of data retrieval focused applications
 - Memory efficient

Why solve this problem?

Introduction

- The volume of data being produced and consumed daily has increased quite aggressively in recent times.
- In just one internet minute -
 - 4 million Google searches are performed
 - 7 hundred thousand hours-worth of videos are consumed on Netflix
 - Snapchat creates around 2 million snaps
 - Users upload over 4.5 million videos on Youtube
- Essential services like banking, social media, government records, etc. require error-free storage and processing of data.

Introduction

- Answer to all of this? - High performance database systems.
- There are many kinds of databases based on how structured the data is and how the data will be queried.
- Most of the databases are built for general purposes where reads and writes are equally important.
- Systems focusing on data retrieval or heavy querying of data require database systems which allow fast read speeds.
- In this project, we aim to build such a database system and show its functionalities with respect to Search Engines.

We need to understand
Document Stores

Document Stores - Background

- Document stores are data storage systems that facilitate storing, retrieving, and managing document type records.
- This kind of databases are an important part of NoSQL database systems because the documents are semi-structured data.
- They offer the ability to query or update based on the underlying structure of the document through APIs or a query/update language.

Document Stores & Search Engines

- Many modern search engines have a similar architecture to NoSQL databases.
- Many search engines can also serve as a document store in their own right.
- Their indexes and query processing are dispersed over multiple servers.
- Document stores are frequently used to store unstructured data, or data that can be saved in a variety of structures, such as social network posts or web pages, etc.

Document Stores - Existing options

- MongoDB
- CouchDB
- OrientDB
- MarkLogic
- DocumentDB
- CosmosDB

Why not choose out of these?

- Are general purpose databases
- Don't focus wholly on fast reads thus sacrificing speed
- Some of them are paid

- It is fun to write your own custom database as per your requirements!

Understanding the input

Web Crawling

- Web crawlers are automatic programs that monitor recognized websites for changes and follow links to new websites and pages, indexing whatever content they come across.
- A collection of URLs is the starting point for a web crawler.
- They read the text on the web pages and index it.
- Add newly discovered links to the queue of indexed sites.
- This allows search engines to return search results from index pages, fast.

WARC Files

- A WARC (Web ARChive) is a container file standard for storing web content in its original context.
- WARCs are produced by crawlers, proxies, and other utilities that retrieve files from a live web server.
- It is a multi-purpose container file:
 - It can house other files
 - Concatenates several files into one digital object
 - Wraps around other files like the PDF and MP3 above, along with some additional information and formatting
 - Container to files that are native to the web

WARC Files

- WARC files contain metadata about the collection and arrangement of the sites' media so that they can be read and represented in live web browsing experiences like they were at the time of their collection.
- These are maintained by the International Internet Preservation Consortium (IIPC) which is an organization established to coordinate efforts to preserve internet content for the future.
- The WARC file format standard was published by the International Organization for Standardization (ISO) committee on technical interoperability as ISO 28500.

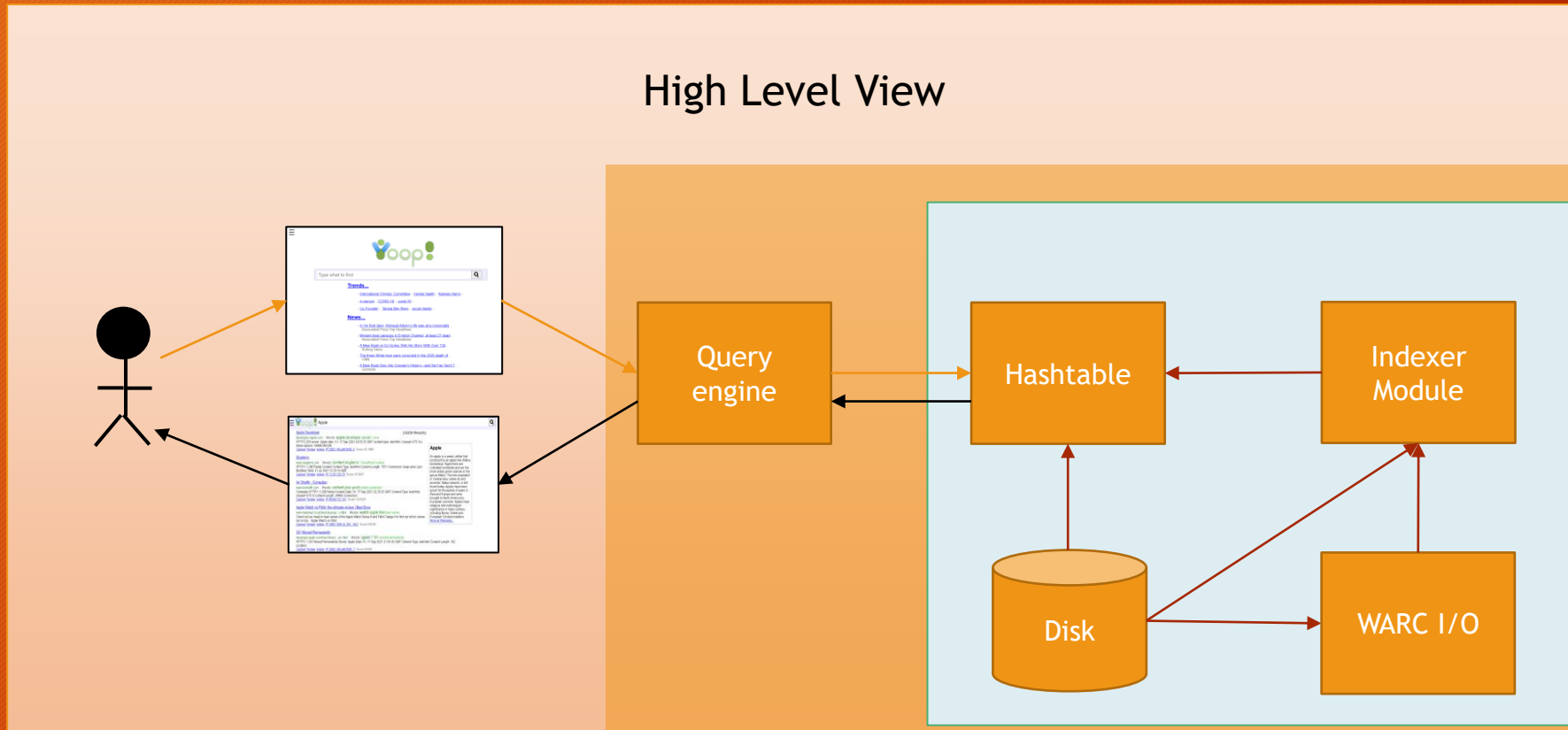
CDX Files

- CDX files are used to index records in WARC files.
- Generally, WARC files are large in size, approximating a couple of gigabytes or more, so having an index file helps in quick reads.
- CDX file contains a header line specifying the format of all subsequent lines.
- We can read a CDX index line and then read the subsequent offset specified to retrieve the WARC record in a WARC file quickly.

Design

System Architecture & Work Flow

High Level View



Components - Query Engine

The Query Engine serves the following purposes:

- Keeps running as an HTTP server.
- Receives the search query
- Processes it to format understandable by Indexer Module
- Processes the response from Hashtable
- Sends it back to the application which sent the query

Technology Choice - GraphQL

Below are the advantages of using GraphQL-

- No over-fetching and under-fetching
- Saves time and bandwidth
- Allows custom schema
- Good for fetching nested data
- Supports fetching data from different types of source APIs
- Fits our use case well and provides extensibility in future

Other option we explored was an ODBC based driver which required a separate HTTP(s) based server and was difficult to integrate with Rust.

Components - Hashtable

The Hashtable serves the following purposes:

- Component which does the actual storing of data and is the essence of this database system
- Fast read access of values (web pages) based on keys
- Worst-case run time complexity of read operation in linear Hashtable is $O(\text{constant})$
- Has no intelligence, does whatever it is told to do by the indexer module and hence, can be used as a pluggable component

Technology Choice - Linear Hashing

- Key-value pairs are stored in an array with slots pointing to their corresponding, value-storing linked lists, also called buckets.
- Linear hashing is a dynamic hashtable algorithm where the buckets, as well as the bucket array can grow as new key-value pairs are inserted.
- Without linear hashing, in case of collisions, the linked list could grow to a long length and effectively, read can start becoming a lengthy operation.

Technology Choice - Linear Hashing

- In linear hashing, a threshold is defined for the length of the linked lists (or size of a bucket).
- If that threshold is reached, bucket is split into new bucket and the values in it are re-hashed to newly split buckets, based on LSB comparisons.
- Causes linear write times if one starts with two buckets because a lot of splitting and re-hashing occurs.
- Ensures limited length of linked list thus, ensuring fast, $O(\text{constant})$ read times.
- Is space efficient because at each split, only 1 extra bucket is created, keeping extra space consumed to a minimum.

Components - Indexer Module

The Indexer Module serves the following purposes:

- A utility which compresses and indexes the WARC records in a CDX record like format, before storing them in the Hashtable.
- Also decompresses the records after their retrieval from Hashtable, so that they can be sent as a response.
- Needs to be provided with the format in which indexing is required.
- Gets this information from Query Engine, based on the schema.

Components - WARC I/O Utility

- Data needs to be present in the linear hashtable for them to be accessed for a query.
- Data is the web pages which are produced by web crawlers and stored in WARC files.
- WARC I/O utility, as the name suggests, serves the purpose of reading and writing WARC files.
- Write functionality is not used in our application as of now, but it's an additional feature.

Components - WARC I/O Utility

This utility has the following functionalities:

- Read WARC records
- Read records from a compressed WARC file in GZIP format
- Read records based on a filter
- Write WARC records in existing or new files
- Write records to a compressed WARC file in GZIP format
- Write records based on a filter

Language Choice - Rust

The whole project has been written in Rust because of the following reasons:

- Rust is becoming the to-go choice wherever performance and memory efficiency is required, some resources claiming that it matches, if not exceed, C in terms of performance.
- Uses LLVM to generate assembly level code, hence the speed.
- Good for applications which require security and memory safety.
- Strictly-typed language which detects memory leaks and unsafe code at compile-time.
- Borrow checker to check lifetime of variables at compile-time and hence eliminating the need of garbage collection.

Experiments and Results

Hardware Specification

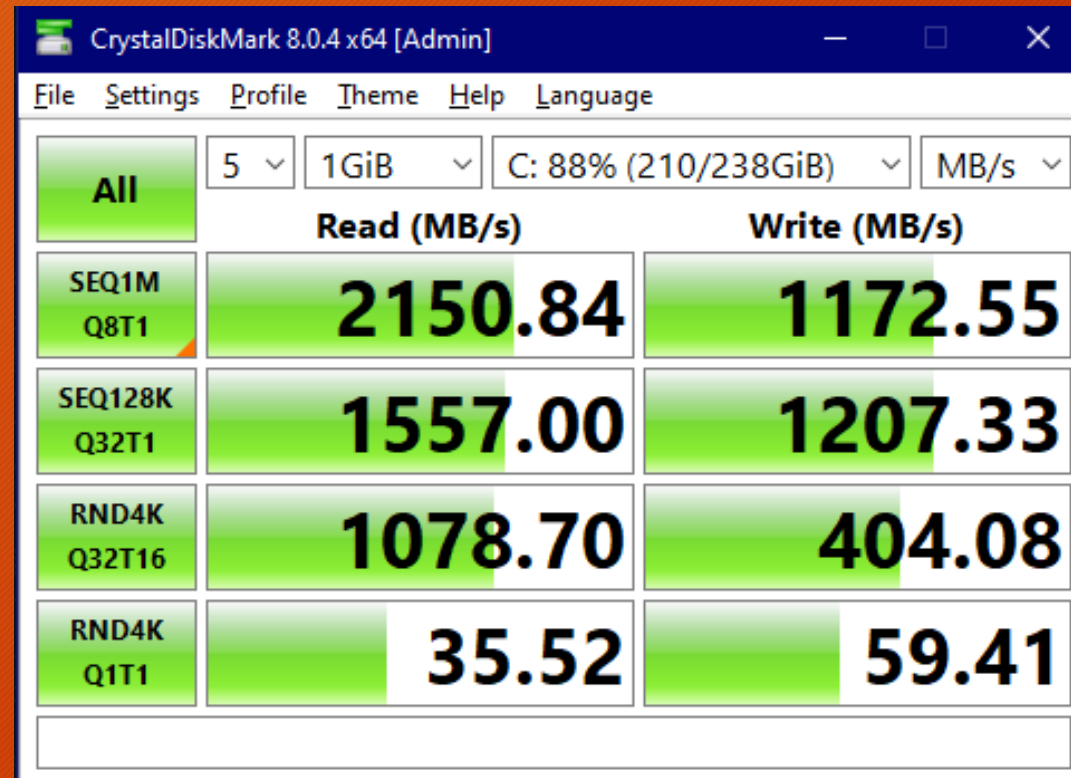
We performed various tests on the application using a machine with below mentioned specification:

- Model - Dell Latitude 3410
- CPU - Intel® Core™ i5-10210U @ 1.6GHz (8 CPUs) ~2.1 GHz
- RAM - 8192MB

```
System Information
Current Date/Time: Friday, December 10, 2021, 7:11:47 AM
Computer Name: 226LATITUDE3410
Operating System: Windows 10 Enterprise 64-bit (10.0, Build 19042)
Language: English (Regional Setting: English)
System Manufacturer: Dell Inc.
System Model: Latitude 3410
BIOS: 1.5.2
Processor: Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz (8 CPUs), ~2.1GHz
Memory: 8192MB RAM
Page file: 10037MB used, 30683MB available
DirectX Version: DirectX 12
```

Hardware Configuration

- Disk Type - Solid State Drive



The screenshot shows the CrystalDiskMark 8.0.4 x64 [Admin] application window. The interface includes a menu bar (File, Settings, Profile, Theme, Help, Language) and a control panel with dropdowns for 'All', '5', '1GiB', 'C: 88% (210/238GiB)', and 'MB/s'. The main display area shows performance results for five different test profiles, each with a green progress bar and numerical values for Read (MB/s) and Write (MB/s).

	Read (MB/s)	Write (MB/s)
All		
SEQ1M Q8T1	2150.84	1172.55
SEQ128K Q32T1	1557.00	1207.33
RND4K Q32T16	1078.70	404.08
RND4K Q1T1	35.52	59.41

Experiments & Dataset

Following slides will show experiments on two major components:

- Linear Hashtable - dataset is string key-value pairs ranging from 4 bytes to 16 bytes.
 - Time taken for insertion of various number of records
 - Time taken for retrieval of various number of records
- WARC I/O utility - dataset will be compressed (GZIP) WARC files
 - Time taken to read various number of records
 - Time taken to write some number of WARC files

These will also be compared against other similar implementations.

Experiments & Results - Linear Hashtable

Number of Insertions	Time (seconds) - 4 bytes	Time (seconds) - 8 bytes	Time (seconds) - 16 bytes	Time (seconds) - 32 bytes
10,000	19.5812	13.7196	10.7140	11.0392
20,000	40.5785	29.5620	29.2961	29.1158
50,000	135.2164	115.9305	110.8547	111.2843
100,000	294.1901	286.4809	290.2886	295.3691

- Rows represent different number of insertions
- Columns represent, in bytes, the size of keys and values that were inserted and the time taken

Experiments & Results - Linear Hashtable

Number of Retrievals	Time (seconds) - 4 bytes	Time (seconds) - 8 bytes	Time (seconds) - 16 bytes	Time (seconds) - 32 bytes
10,000	8.4741	6.0003	4.0419	4.2534
20,000	22.6415	18.2303	11.5651	11.5278
50,000	77.5042	53.4573	50.5842	55.6155
100,000	103.5246	91.2837	92.4166	91.2974

- Rows represent different number of retrievals
- Columns represent, in bytes, the size of keys and values that were inserted and the time taken

Experiments & Results - Linear Hashtable

Number of Insertions	Time (seconds) – Rust	Time (seconds) – JS
10,000	19.5812	45.0361
20,000	40.5785	101.1892
50,000	135.2164	296.7893
100,000	294.1901	687.6312

- Rows represent different number of insertions
- Columns represent comparison between java script and rust implementation starting with 2 initial buckets

Experiments & Results - Linear Hashtable

Number of Insertions	Time (seconds) – (2 buckets)	Time (seconds) – (256 buckets)	Time (seconds) – (1024 buckets)
10,000	19.5812	6.1263	6.0912
20,000	40.5785	14.3429	15.2482
50,000	135.2164	43.1485	41.6578
100,000	294.1901	195.3374	156.7210

- Rows represent different number of insertions of key-values of 32 bytes
- Columns represent the time taken for insertions when the number of initial buckets is increased

Experiments & Results - WARC Reads

Number of records parsed	Time (seconds) - Rust
10,000	71.5156
20,000	148.0942
50,000	432.8688

- Rows represent different number of WARC records parsed; each record is of size ~20KB

Challenges

- Rust is a relatively newer and less popular language in comparison to Java, C++, Python, JavaScript, etc.
- Steep learning curve
- Not many experts in this language
- Integration support available for established technologies but nearly no support for custom implementations
- Lack of libraries in Rust which are easily found in older languages
- Strongly typed vs Loosely typed languages - Porting indexer module from PHP to Rust was challenging

Learnings & Growth

Technical:

- Understood linear hashing mechanism, indexing in databases, and query engines
- Rust
- WARC files

Personal

- Troubleshooting
- Finding multiple solutions to one problem
- Perseverance

Conclusion

- We have built a document store which can support applications focusing on data retrieval.
- We have built a document store which is a high performing, durable, and memory efficient database system which can support fast reads.
- Our Linear Hashtable implemented using Rust, which is the core of this fast database, performs better than other similar implementations.
- Our WARC I/O utility in Rust performs similar to current Python implementation.

References

- <https://samrat.me/posts/2017-11-04-kvstore-linear-hashing/>
- <https://stackoverflow.blog/2020/01/20/what-is-rust-and-why-is-it-so-popular/>
- <https://dzone.com/articles/why-and-when-to-use-graphql-1>
- <https://blog.logrocket.com/how-to-create-a-graphql-server-in-rust/>
- <https://iipc.github.io/warc-specifications/specifications/cdx-format/cdx-2015/>

Thank You

Questions are welcome!