Faculty of Science and Technology
Department of Computer Science

BacklinkDB: A Purpose-Built Backlink Database Management System

Marius Løvold Jørgensen

INF-3981 Master’s Thesis in Computer Science - February 2023

Abstract

In order to compile a list of all the backlinks for a given webpage, we need knowledge about all the outgoing links on the web. Traversing the web and storing all the backlink data in a database allows us to efficiently retrieve the list of backlinks for a web page on demand. However, the web consists of billions of backlinks which translates to terabytes of data. As the web is continuously evolving, the database needs to be rebuilt periodically in order for it to closely resemble the current state of the web.

This thesis presents BacklinkDB, a purpose-built database management system designed for managing a backlink database. Using a series of in-memory hash indices allows for high insert throughput when building the database. The backlink data for a given domain is stored together in sections throughout the database file. This allows for the requested backlink data to be easily located. With a simple SQL-inspired query language, the users can both insert and retrieve backlink data.

The evaluation shows that building a purpose-built database management system allows us to make the trade-offs between which performance metrics that is important. In this thesis, we will focus on creating a scalable backlink database management system with high insert performance.

Acknowledgements

First, I want to thank my supervisor Weihai Yu for his feedback and guidance in this project. I would also like to thank my friends and family for their support throughout my time as a student at UiT.

Contents

List of Figures

List of Tables

List of Abbreviations

ACID Atomicity, Consistency, Isolation and Durability

ACM Association for Computing Machinery

ANSI American National Standards Institute

BI Business Intelligence

CPU Central Processing Unit

DBMS Database Management System

GHz Gigahertz

GiB Gibibyte

GNU GNU’s not Unix

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

IANA Internet Assigned Numbers Authority

LTS Long-Term Support

MiB Mebibyte

ORM Object–Relational Mapping

OS Operating System

RAM Random-Access Memory

RDBMS Relational Database Management System

SPARC Standards Planning And Requirements Committee

SQL Structured Query Language

TCP Transmission Control Protocol

TLD Top-Level Domain

UiT University of Tromsø

URL Uniform Resource Locator

WARC Web ARChive

List of Listings

1 Introduction

Search engine companies collect and organize data from the world wide web to later help people find what they are looking for. Querying a search engine returns a curated list of links which is ordered based on what the search engine perceives as most relevant. Web searching can be divided into three categories of intent. Informational, navigational, and transactional search queries. Approximately 10% of all searches are transactional and navigational[5]. Transactional and navigational searches are valuable for businesses that have an online presence. Research has shown that businesses ranking high in search engines for relevant search terms will attract more customers[1, 10].

The implementation details of the most used search engines are a well-kept secret, but analysis of the search engine results page has shown that the biggest search engine, Google, utilizes the PageRank algorithm[8] when ranking webpages [7]. The PageRank algorithm ranks webpages by calculating the quality and quantity of all the incoming links (also known as backlinks).

To compile a list of all the backlinks for a given webpage, we need knowledge about all the outgoing links on the web. This is can be found by traversing the web using a web crawler. The web consists of billions of webpages[4], which translates to terabytes of backlink data. Access to backlink data is beneficial for people who want insight into who is linking to a specific webpage and businesses who want to analyze their competitors.

In this thesis, we will describe BacklinkDB, a purpose-built database management system for managing a backlink database. We will analyze the purpose-built approach by comparing it with two other popular database management systems, and discuss if BacklinkDB is a feasible alternative.

1.1 Thesis Statement

Access to backlink data can help reverse engineer the search engine results page. However, the web consists of billions of backlinks. Storing this data requires a system that can organize and efficiently serve terabytes of backlink data.

This thesis aims to investigate the potential of creating a purpose-built database management system as a scalable and cost-effective approach for managing a backlink database.

The thesis will investigate if the system is feasible by comparing the implementation of the proposed design to other database management systems also configured for storing backlink data.

1.2 Scope, Assumptions, and Limitations

Because the web is constantly evolving, the backlink data does too. Organizations like Common Crawl[1] crawl, organizes, and publish a full dataset consisting of all the indexable webpages each month. Assuming that the data stored in the backlink database will be outdated after a short period, implies that the database needs to be rebuilt often in order for it to always resemble the current state of the web. Because the backlink database needs to be rebuilt periodically, the design of BacklinkDB will not cover the *update* and *delete* operations for database records.

The focus of this thesis is on creating a scalable system for storing and retrieving backlink data. Therefore, the thesis will not focus on the handling of different types of failures. However, how this functionality could be integrated into the proposed design is discussed in section 8.4.1.

1. https://commoncrawl.org/

A common strategy for scaling a database is to partition the data over a distributed system. The thesis will only focus on the scalability of a single running instance of the database. A short discussion on how BacklinkDB can be converted to a distributed data store is found in section 8.1.7.

1.3 Method and Approach

In the final report of the Task Force on the Core of Computer Science by the ACM Education Board, a framework for representing scientific work within computing is described[2]. The framework consists of three main paradigms.

Theory: Consists of four steps rooted in mathematics. (1) Characterize objects of study (definition). (2) Hypothesize possible relationships among them (theorem). (3) Determine whether the relationships are true (proof). (4) Interpret results.

Abstraction (modeling): Based on four experimental methods which are followed in the investigation of a phenomenon. (1) Form a hypothesis. (2) Construct a model and make a prediction. (3) Design an experiment and collect data. (4) Analyze results.

Design: Consists of four steps rooted in engineering and consists of four steps followed in the construction of a system (or device) to solve a given problem. (1) State requirements. (2) State specifications. (3) Design and implement the system. (4) Test the system.

This thesis follows the steps of the *Design* paradigm. The requirements and specifications are outlined. A design is presented using the three abstraction levels of the Three-Schema Architecture, along with an implementation of the design. The system is then tested for correctness before being compared against the performance of other systems configured to solve the same problem.

1.4 Organization

Structure of this thesis:

Chapter 2: Background covers the technical background for this thesis.

Chapter 3: Design outlines the requirements for a backlink database and describes the design of BacklinkDB using the Three-Schema Architecture.

Chapter 4: Implementation provides implementation details.

Chapter 5: Experiments covers details about the motivation and the setup for the experiments.

Chapter 6: Results presents the experiment results.

Chapter 7: Evaluation evaluates the three backlink databases based on the results.

Chapter 8: Discussion and Future Work discusses the design choices, shortcomings, and potential improvements.

Chapter 9: Conclusion summary of the thesis findings.

2 Background

This chapter covers the technical background for this thesis. Section 2.1 explains the concept of websites and webpages. Section 2.2 introduces the fundamentals of the PageRank algorithm. Section 2.3 outlines the adjacency list data structure. Section 2.4 covers the basics of a web crawler. Section 2.5 gives an introduction to database management systems and the different types of databases relevant to this thesis. Section 2.6 describes the three abstraction levels of the Three-Schema Architecture.

2.1 Website and Webpages

A webpage is a web document accessible via an URL. A website consists of one or multiple webpages hosted under the same domain address.

2.1.1 URL - Uniform Resource Locator

An URL specifies the address of a webpage. Outlined in figure 2.1 is the different components that make up an URL. The *protocol*, *subdomain*, *TLD*, and *path* are required in an URL for it to be valid, while the *query string* and *fragment* are optional components that different web technologies can utilize for additional functionality.

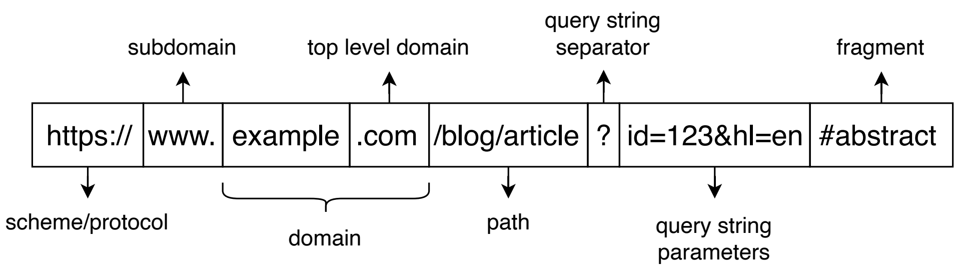


Figure 2.1: The components of an URL.

2.2 PageRank

PageRank[8] is an algorithm designed for ranking webpages. PageRank introduces the concept of a web surfer who traverses the web by randomly clicking successive links. Using the model of the random web surfer, a distribution of probabilities is created to represent how likely the surfer is to visit each of the webpages.

The algorithm can compute the distribution of probabilities for a large number of webpages effectively. PageRank was used as the foundation in the first version of the Google search engine, and it is still to this day a dominant factor for ranking webpages[7] in the Google search engine.

A backlink is an inbound (or external) link from one webpage to another. Figure 2.2 illustrates the perceived importance of a small set of websites where some of them link to each other. The perceived importance is calculated using the backlink data with the PageRank algorithm.

2.3 Adjacency List

An adjacency list is a data structure that can be used to effectively store graph data. The data structure consists of an array that is combined with multiple linked lists. An example of an adjacency list and its graph representation is illustrated in figure 2.3.

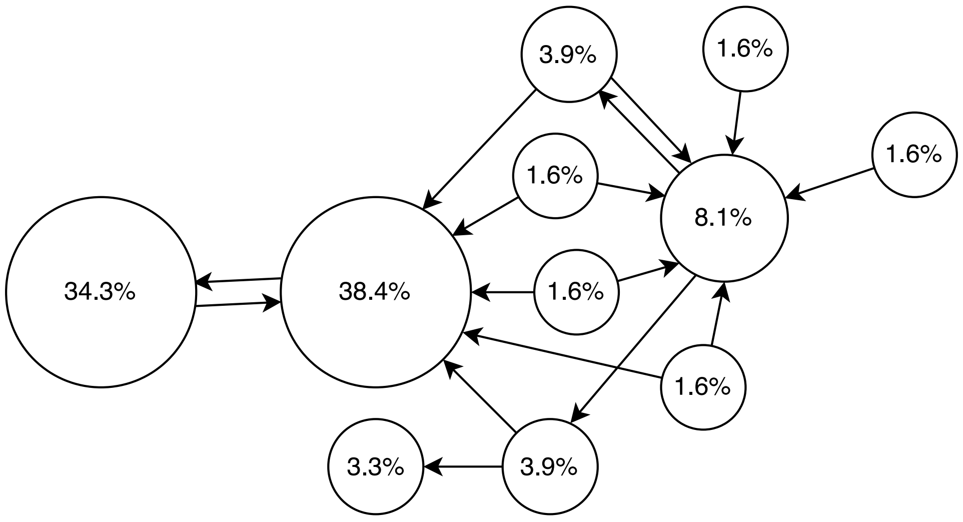


Figure 2.2: Example of the PageRank score for a small web of websites.

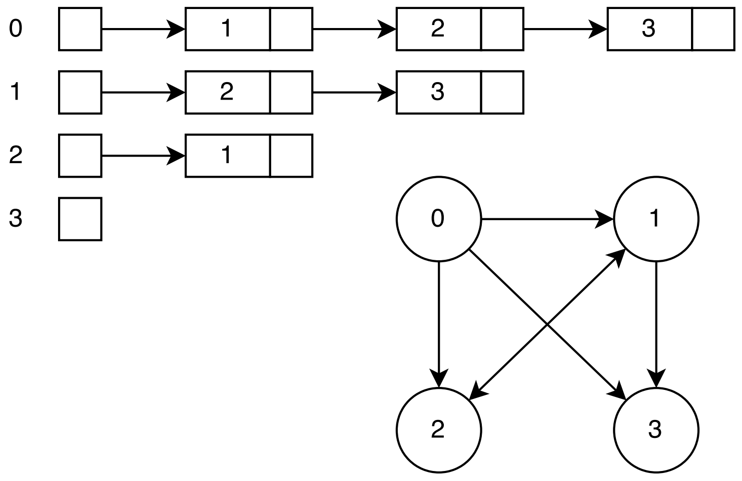


Figure 2.3: Example of an adjacency list along with its graph representation.

2.4 Web Crawler

A web crawler is a program that traverses the web, creates indices, and downloads the contents of webpages. A set of seed URLs is used as a starting point to initiate a crawl. After the seed URLs are crawled, a new set of URLs will have been discovered and the crawlers start to traverse the web until all the indexable webpages have been crawled. It is common for web crawlers to store the crawl data using the Web ARChive (WARC) file format.

2.5 DBMS - Database Management System

Database Management System (DBMS) is a type of software that manages the database. DBMSes presents an interface to execute operations that result in the manipulation or retrieval of data from the database. These operations are performed using a query language. Different query languages exist for different types of databases.

2.5.1 Database Index

A database index is a data structure that reduces the cost of processing queries. Index entries combine a search key with a pointer. The pointer stores the location on a database file where the corresponding data to the search key is located. There are two types of indices:

Ordered indices

An ordered index stores the values of the search key in sorted order. Using ordered indices allows for fast random access as an entry can be found using binary search. Binary search has an average time complexity of $O\ log(n)$.

Hash indices

A hash index uses a hash function to distribute the search keys across a range of buckets. Hash indices are commonly used for in-memory database indices. Using a hash function allows for quick look-ups as the average time complexity is $O(1)$.

2.5.2 Transactional Databases

A transactional database ensures that the validity of the data will be protected despite a failure or crash. Atomicity, Consistency, Isolation and Durability (ACID) is a set of properties that guarantees validity in a transactional database.

Atomicity - The atomicity property requires each transaction to be atomic. This is done by bundling together the set of database operations in a transaction to create a unit. Either all of the operations in the unit are executed or non of them are. This ensures that database will never enter an inconsistent state where some of the data is partially updated.

Consistency - The consistency property requires that a transaction takes the database from one valid state to another. And that the latest updated values are always returned.

Isolation - In a database that processes transactions concurrently, the isolation property requires that the result of the concurrently executed transactions is the same as if the transactions were to be executed in sequential order.

Durability - The durability property states that when a transaction is committed, a crash or failure cannot revert the committed transaction.

2.5.3 Analytical Databases

An analytical database focuses on storing vast amounts of data that is later used in analytical services and applications. Analytical databases are often used by data scientists and analysts to perform Business Intelligence (BI) processes. These processes often consist of aggregating large amounts of data. The workload of an analytical database consists of data-intensive read-only queries and batch inserts.

2.5.4 Relational Databases

In a relational database, data is stored using predefined relations in tables. Multiple tables can be joined, often using a primary and foreign key. In most Relational Database Management System (RDBMS) the Structured Query Language (SQL) is used.

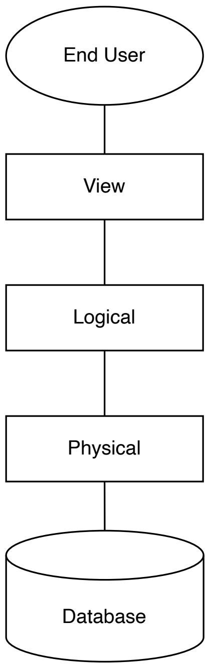


Figure 2.4: Illustration of the Three-Schema Architecture.

2.5.5 Graph Databases

Graph databases manage nodes and edges. The data entities are stored in the node while the edges between the node represent relationships. Relationships between nodes are described using a type and direction. Graph databases are often found in social networking applications to store information on people and defined their relationships with each other.

2.6 Three-Schema Architecture

The Three-Schema Architecture[9] (also known as the ANSI-SPARC Architecture) presents a standard for design an DBMS using three levels of abstraction. Figure 2.4 illustrates an overview of the different components of the architecture.

External Level - The external level consists of the different views of data that are presented to the end user.

Conceptual Level - The conceptual level describes the structure of the data stored in the database. Constraints and relationships of data are defined on this level.

Internal Level - The internal level describes how the database files are structured. This layer outlines the physical representation of the data on the disk.

2.6.1 Data Independence

The Three-Schema Architecture presents the idea of data independence between the layers. Changes to the DBMS in the internal layers should not require changes to be made in the schema of the conceptual level. And changes to the implementation of the conceptual level should not affect the data presented in the external level.

3 Design

The following chapter describes the design of BacklinkDB. This chapter covers the design of the overall DBMS structure and its components. The focus of the design is to create a resource-efficient and scalable system for managing a backlink database. Section 3.1 outlines the functional requirements. Section 3.2 presents an structural overview of the systems design. Section 3.3, 3.4 and 3.5 describes the design using the abstractions levels of the Three-Schema Architecture.

3.1 Requirements

This section outlines a set of the core functional requirements a DBMS managing a backlink database must be able to fulfill.

3.1.1 Functional Requirements

- A client must be able to connect to the DBMS and be able to execute queries.
- The DBMS must be able to insert backlink records in bulk, directly from a file.

- A client must be able to query the DBMS for a list of all the backlinks to a website given a domain name.
- A client must be able to query the DBMS for a list of all the backlinks to a webpage given an URL.

3.2 Overview - Database System Structure

Figure 3.1 presents a structural overview of the DBMS architecture. The database engine consists of multiple connected modules that have different responsibilities. In this chapter, these modules will be described in more detail.

3.3 External Level

The external level presents details about the interface a user of the DBMS will interact with. A simple SQL inspired query language is designed for interacting with BacklinkDB.

3.3.1 Query Language

BacklinkDB features a simple query language with support for three different queries in order to support the functionalities of inserting and retrieving backlinks.

SELECT EXACT

The `SELECT EXACT <URL>` statement retrieves and returns a list of all the backlinks for the given URL.

SELECT DOMAIN

The `SELECT DOMAIN <domain>` statement retrieves and returns a list of all the backlinks for a given domain.

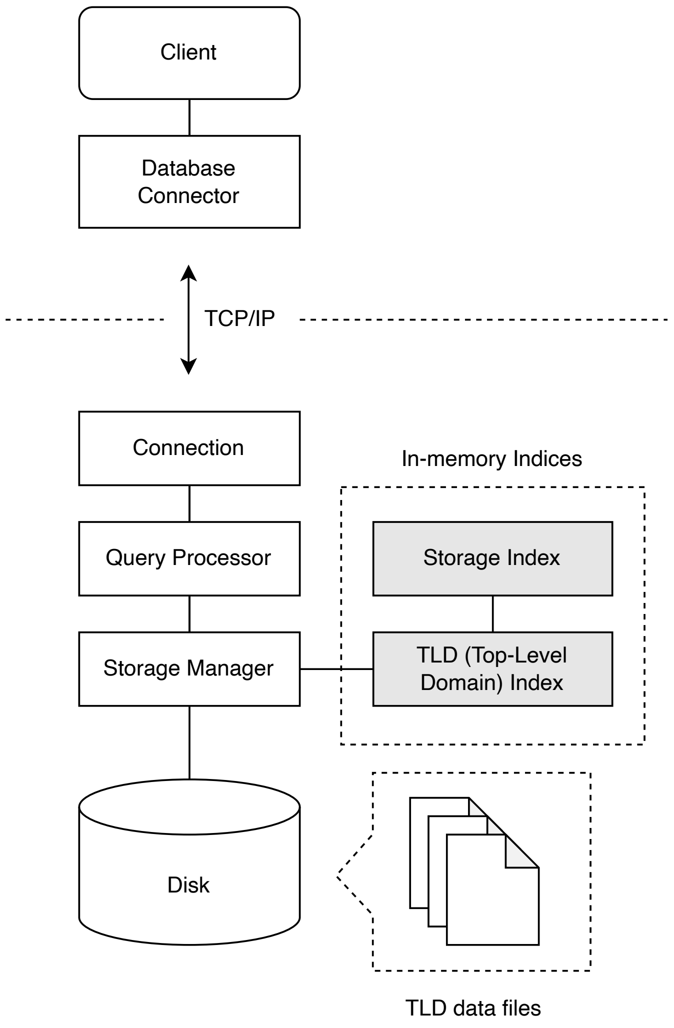


Figure 3.1: Overview of the BacklinkDB system structure.

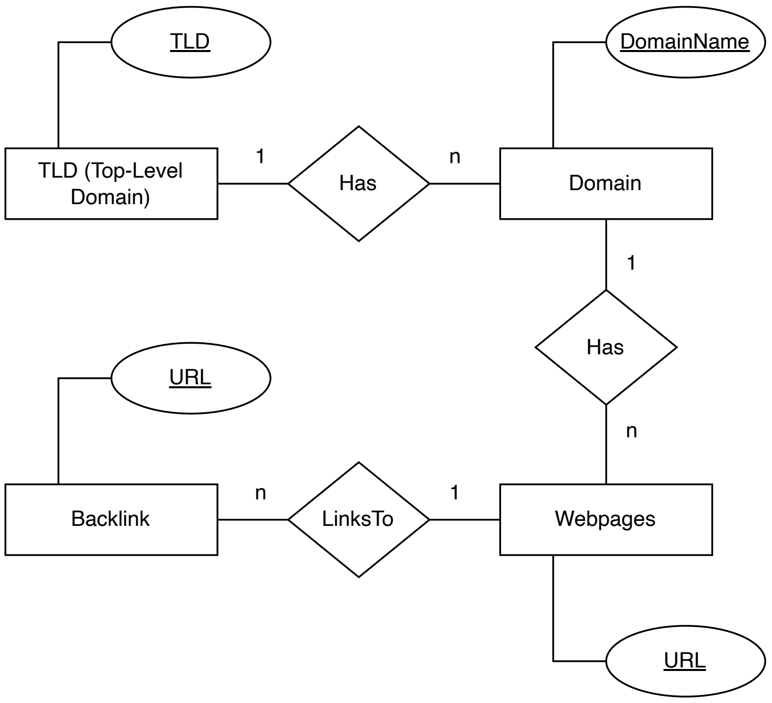


Figure 3.2: Entity–Relationship Model of the data stored in BacklinkDB.

LOAD

The `LOAD <filename>` statement iterates and inserts all of the backlinks from the given filename into the database.

3.4 Conceptual Level

Figure 3.2 illustrated the conceptual design of BacklinkDB using an Entity–Relationship Model. The figure defines what type of data is stored in the database and their relations.

3.5 Internal Level

This section will describe the internal abstraction level of BacklinkDB. The internal level describes the physical details of how the database manages the data on disk and in memory.

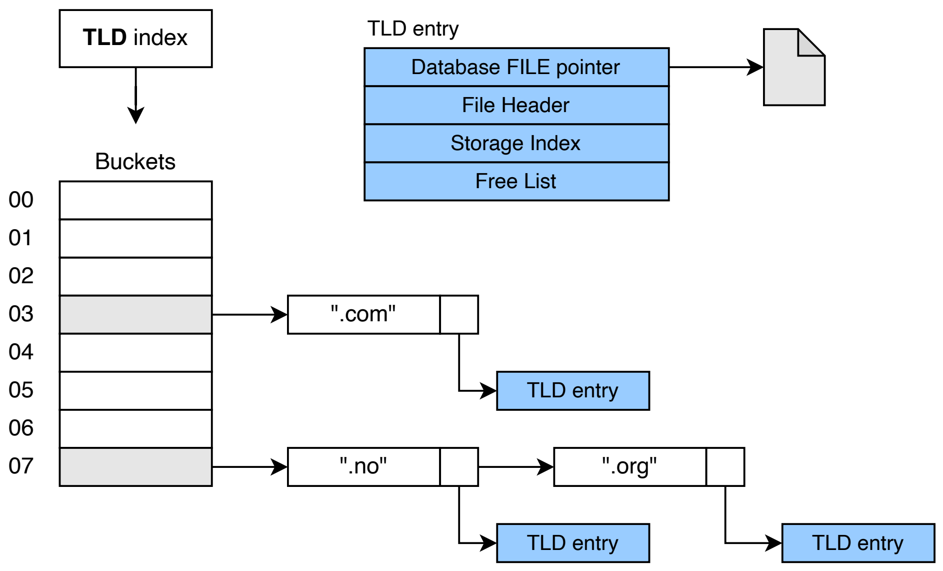


Figure 3.3: Overview of the TLD index structure.

3.5.1 Storage Manager

The storage manager's main purpose is to write and read the data from and to the disk. Communication with the disk is done through the Operating System (OS) file abstraction. By using the OS file abstraction, the underlying block structure can be ignored. However, the storage manager creates its own abstraction of blocks inside the data files in order to efficiently access data. The database consists of multiple data files, one for each TLD. The storage manager uses 4096-byte size blocks to read and write data to and from the data files.

3.5.2 Top-Level Domain (TLD) Index

The database keeps all of the backlinks for each separate TLD in different data files. Additional information about the data is stored during runtime. This information is stored in a TLD-entry. For efficient retrieval of the TLD-entry, a hash table is used. Using a hash table allows for high performance when querying and inserting data with an average time complexity for both operations of $O(1)$. Illustrated in figure 3.3 is an overview of the different components in the index. Each entry contains a pointer to the data file, an index for the data itself, and metadata for the data, such as the current number of links and domains. Each entry also contains a *free list* which is used to keep track of the unused sections in the data file.

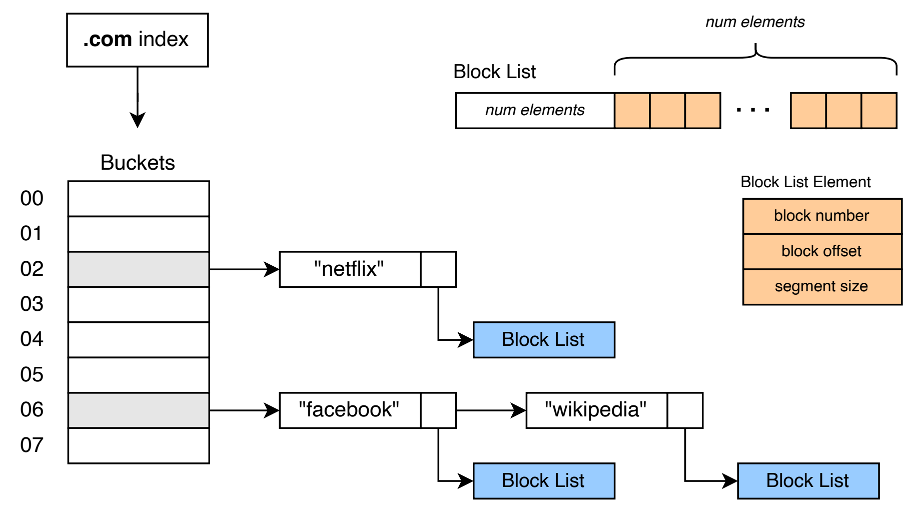


Figure 3.4: Overview of the storage index and block list structure using the *.com* TLD.

3.5.3 Data files

There is one data file for each of the TLDs. By naming the file using the TLD, it can easily be located in a directory. Each of the data files consists of a file header followed by a series of 4096-byte size blocks containing the backlink data. The file header keeps track of the current number of domains, backlinks, and blocks.

3.5.4 Storage Index

As illustrated in figure 3.3, each TLD-entry has its own storage index. The storage index keeps track of the different domains and what blocks the backlink to that domain is stored on. The storage index also uses a hash table because of its high performance on search and insertion. Figure 3.4 illustrates the different components in the storage index.

3.5.5 Block List

The block list contains the location of all the blocks that store the backlinks to a given domain. The structure of a block list is highlighted in orange in figure 3.4. A block list consists of a header, followed by multiple block list elements. The header stores the total number of elements, while each element stores a reference to a segment using the block number, block offset, and segment size. Representing the block list using this structure allows for it to be efficiently stored in memory.

3.5.6 Free List

As illustrated in figure 3.3, each TLD-entry has its own free list. The free list lives in the memory at runtime and keeps track of all the unused block segments in the data file. A free list entry contains the block number, segment offset, and segment size for the unused space in the data file. This data structure is used to effectively locate unused space when inserting backlink records.

3.5.7 Disk Block Structure

Each block is divided into 16 equal size 256-byte segment slots. The first slot of the block is assigned to the disk block header. The header holds information about what segment slots on the block are occupied. A segment dynamically resizes when more records are inserted. The segment's initial size is equal to 1 slot (256 bytes) and the maximum segment size is 15 slots (3850 bytes). When the maximum size of a segment is reached, a new segment at another disk block is allocated.

Block Header

The block header holds information about which segment slots on the block are occupied. This is done by storing the offset and the size (number of slots occupied) for each of the segments located on the block. Figure 3.5 illustrates the structure of the block header and its relation to the data on the block when some of the slots are occupied. The information stored in the block header is essential for the creation of the database indices and free list when the DBMS is initializing. Unoccupied segments (illustrated in white in on figure 3.5) are stored as entries in the free list during the runtime.

Block Segment

A block segment stores all or part of the backlink data for a specific domain. The link data is structured using variable-length tuples. The tuple consists of the *destination page* and the *source site*. In a link record, the *source site* represents the URL of the backlink, and the *destination page* contains the path of the block segments domain to which the *source site* is linking to.

URLs are variable-length records, which means that the segment needs to store additional information on how much space the *destination page* and the *source site* are occupying. This is done using slots. The section of slots for a block segment is located right after the segment header, as illustrated in figure 3.6.

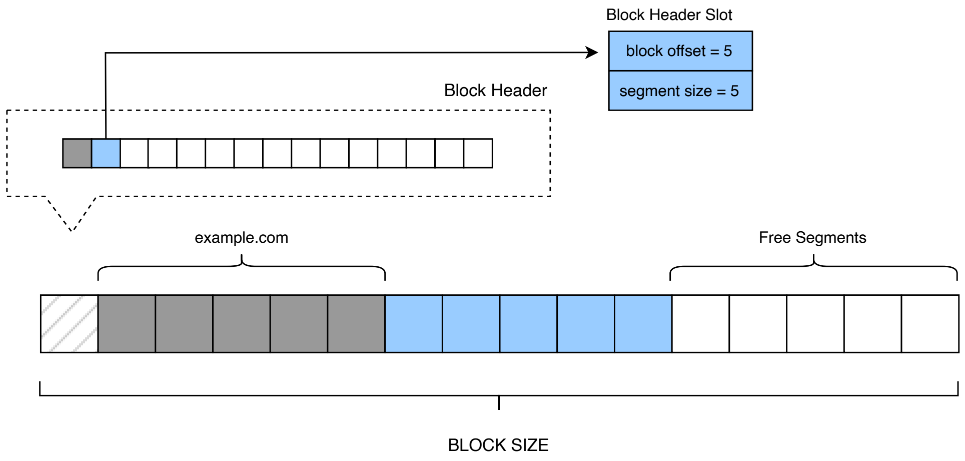


Figure 3.5: Illustration of the segment structure. The *block header* keeps track of the different segments located on the block.

These slots describe the offset and size for each of the link records stored in the segment. When inserting backlinks into the segment, the slots are appended, while the link record itself is added at the end of the free space.

The slots store information about the size and location of the link record. The link record itself gives more information about how to extract the data. Since both the *destination page* and *source site* is variable length, the header of the record contains information about what part of the record is occupied for each of the two values.

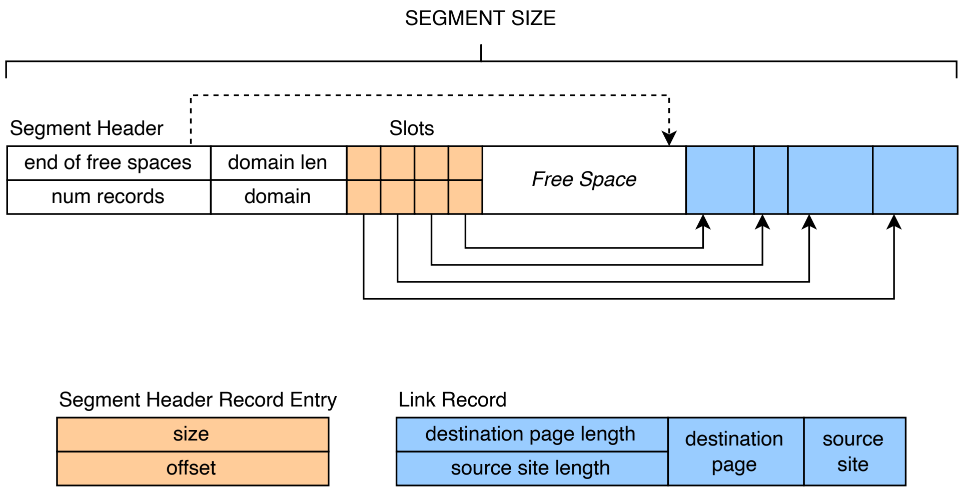


Figure 3.6: Structure of the content in a block segment and the link record that is stored within.

4 Implementation

The following chapter describes some of the implementation details of the design choices outlined in the previous chapter. Section 4.1 describes the choice of programming language BacklinkDB is implemented in. Section 4.2 outlines what test of the implementation was performed. Section 4.3 specifies the implementation details of the hash indices. Section 4.4 describes an internal mechanism implemented to prevent invalid backlink data to be inserted into the database. Section 4.5 describes the implementation detail of how block segments are managed when records are inserted. Section 4.6 describes how the retrieval of backlink data was implemented. Section 4.7 outlines the implementation details of the Transmission Control Protocol (TCP) socket server in the connection module.

4.1 Language choice

BacklinkDB was implemented in the C programming language and compiled using the GNU Compiler Collection[1] version 11.3.0. The C programming language was chosen because of its memory allocation and management features. Having fine-grained control over how memory is managed is beneficial when working with data at a low level.

1. https://gcc.gnu.org/

4.2 Testing

Verifying that the database stores and retrieves data correctly are done using Big-Bang Integration testing. Big-Bang Integration testing combines all the modules in the database, before running a series of database queries. Inserting a large amount of backlink data, before retrieving and comparing it with the inserted data ensures that non of the data is lost and that the DBMS functions as expected.

Utilizing the dynamic memory allocation and management functionality of C can result in memory errors and memory leaks. To prevent that memory errors occurs, the Big-Bang Integration test is run using Valgrind[2]. Valgrind is a tool that detects memory leaks and errors.

4.3 TLD Index and Storage Index

The TLD Index and Storage Index are implemented using a hash table with closed addressing (open hashing). Separate chaining is implemented using singly linked lists where one item is stored on each list element. The load factor of the hash table is set to 0.75 and the total size is doubled when rehashing the table. Jenkins hash function[6] is used for producing the hashes in the table. The hash function is non-cryptographic and is designed to uniformly distribute values.

4.4 Valid TLDs

In order to prevent invalid URLs to trigger the creation of new TLD data files, a hash table containing the values of all the valid TLDs is used. A lookup is done whenever an insert is performed to quickly determine if the destination domain is valid. If the destination TLD is not valid, the insert operation for the backlink is discarded. The list of valid TLDs published by Internet Assigned Numbers Authority (IANA)[3] is used to populate the table.

2. https://valgrind.org/
3. https://data.iana.org/TLD/tlds-alpha-by-domain.txt

4.5 Inserting links

The inserting backlink functionality is implemented by first extracting the TLD from the destination domain. Then, the correct TLD-entry from the TLD index is located and retrieved. The TLD-entry points to the Storage Index that is used to locate the Block List for the destination domain. The last element in the block list points to the last added segment containing the backlinks to the destination domain. If full, the current segment is resized or a new segment is allocated using an entry from the Free List. If the segment has available space, the link is inserted directly into that segment. After the link data is inserted into the segment, the block and file header metadata are updated before being flushed. After the data is flushed, the indices are updated.

4.5.1 Requesting a new segment

When a backlink is being inserted, but the allocated space for the destination domain cannot store the backlink record, a new segment is requested. If the Free List contains a segment that satisfies the requested segment size, the new segment is found and the Free List is updated. However, if the Free List does not contain a viable segment, the TLD data file is expanded by one block (4096 bytes). The requested segment is pulled from the newly allocated block, while the remaining segments are inserted into the Free List. After the data file has expanded, the file header is updated and flushed so that the data file has the correct number of total disk blocks stored in the header.

4.5.2 Reallocating segments

When a backlink is inserted into a segment that is full, but the segment itself does not occupy the 15-segment slot limit, the segment is reallocated. Reallocating a segment expands the number of segment slots a segment occupies. If the expanding segment position has a following segment that is free, the segment can expand into that segment.

If the expanding segment does not have a following free segment on the same disk block, the segment needs to be moved to another part of the data file where the new segment can fit. The new segment location is found either in the Free List or by expanding the data file. When the segment moves location, the block header is updated and the location of the newly free segment is stored in the Free List.

When a segment is expanded, the segment needs restructuring. All the link records need to be shifted to the end of the new segment and their respective slots offset needs to be updated. After the link records are moved, the end of the free space pointer also needs to be updated so it is pointing to the start of the first link records location.

4.6 Querying backlinks

When the storage manager is requested to retrieve all the backlinks for a given URL, the TLD of the URL is extracted. The storage index is found by doing a lookup in the TLD index. By hashing the domain in the URL, the Block List can be retrieved from the storage index. Iterating over the Block List gives us information about the location of all the block segments in the data file that contains the backlinks to the given domain. When loading these segments into memory, all the backlinks to the domain are located. By iterating over all the backlinks to the domain, the backlinks for the query request can be filtered before returning.

4.7 Connection

After initialization of the database, BacklinkDB starts a TCP socket server listening for connecting clients. When a client connects, the DBMS and client start communicating using a simple protocol. After the connection is established, the server awaits a query to process. If the query is of type `LOAD`, the DBMS will load the backlinks of the given file into the database. However, if the query is of type `SELECT`, the server will retrieve the requested data using the storage manager and copy it to a buffer. The data is then transmitted over TCP using 1024-byte chunks until the buffer is empty. Before sending the backlink data, the server notifies the client of how many total bytes to expect. This is how the client knows when to stop receiving the 1024-byte chunks.

5 Experiments

This chapter outlines five experiments testing BacklinkDB and benchmarks its performance against SQLite and Neo4j. The goal of the experiments is to produce data to evaluate the design and implementation of BacklinkDB.

Section 5.1 details the setup of the experiments. Section 5.2 outlines how the insert throughput for the three backlink databases is measured. Section 5.3 outlines how the search throughput for the backlink databases is measured. Section 5.4 describes the experiment for measuring total database size. Section 5.5 outlines a more in-depth experiment for analyzing the scalability of BacklinkDB. Section 5.6 describes the experiment of measuring fragmentation in BacklinkDB. Section 5.7 describes how the profiling of BacklinkDB is conducted.

5.1 Setup

5.1.1 Technical Specifications

OS	Ubuntu 22.04.1 LTS (GNU/Linux 5.4.0-137-generic x86_64)
CPU	Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHZ
RAM	128 GiB

5.1.2 Software

SQLite	v. 3.37.0
Neo4j	v. 5.4.0
Python	v. 3.10.6

5.1.3 Data

The link data used in the experiments is downloaded from the Common Crawls website[1]. Common Crawl is a non-profit organization that periodically crawls the web and publicizes data. For the experiments described in this chapter, data from the August 2022 crawl[2] is used.

Data prepossessing

Common Crawl provides data on all the indexable webpages. This data is provided in a series of WARC files found in their public repository. Common Crawl also provide WAT files which are produced by processing the WARC files and extracting the metadata for each webpage. The WAT files contain a list of all the outgoing links for each of the webpages.

All external links from the WAT file are extracted to their own *link* file so that they can be directly inserted into a database. Each link is stored on a separate line in the file using spaces to separate the *source domain*, *source path*, *destination domain*, and *destination path*. All the backlinks containing URLs longer than 2048 characters are discarded. A link file is created for each of the WAT files. These link files contain all the information needed to build a backlink database.

5.1.4 BacklinkDB

Setting up the BacklinkDB requires a client-side to the *connection* module described in section 4.7. This is implemented using Python's socket module[3].

1. https://commoncrawl.org/
2. https://commoncrawl.org/2022/08/august-2022-crawl-archive-now-available/
3. https://docs.python.org/3/library/socket.html

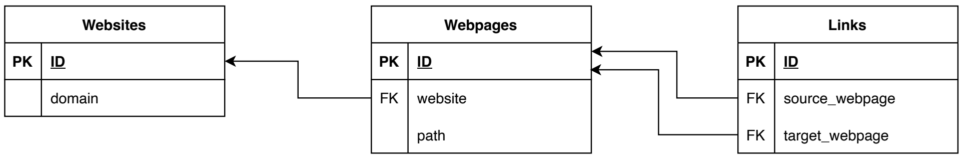


Figure 5.1: Relational backlink database schema.

5.1.5 SQLite

SQLite[4] is an open-source embedded transactional RDBMS written in the C programming language. SQLite is serverless, reading and writing data directly to the file system. As a result of the serverless design, SQLite has low complexity, high performance, and high portability.

The schema for creating the backlink database in SQLite is shown in figure 5.1. One table for the website, one table for the webpages, and one table for the links between two webpages. Querying the SQLite database is performed using Python's Object–Relational Mapping (ORM) package peewee[5] version 3.15.1.

5.1.6 Neo4j

Neo4j[6] is an open-source transactional graph database management system written in Java. The database is designed for enterprise applications featuring support for complex relations and queries.

The node and edge structure for the backlink graph database is illustrated in figure 5.2. Each webpage is represented by a node, and the backlink relation is defined using an edge between two nodes. The node stores information about the domain and path of the webpage. Querying the Neo4j backlink database is done using the neo4j[7] Python driver version 5.4.0.

5.2 Insert Throughput

A fully operational backlink database stores billions of links and has to rebuild periodically. This experiment intends to compare the scalability of the three

4. https://sqlite.org/index.html
5. https://pypi.org/project/peewee/
6. https://neo4j.com/
7. https://pypi.org/project/neo4j/

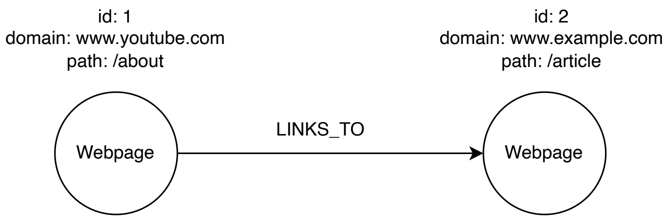


Figure 5.2: Graph backlink database.

different backlink databases by inserting different amounts of backlinks while measuring the throughput.

5.2.1 BacklinkDB

The links are inserted using the `LOAD <filename>` statement described in section 3.3.1.

5.2.2 SQLite

SQLite does not support creating relations when inserting records directly from a file. Therefore, each backlink is inserted separately using *peewee*. When inserting a backlink, the source and destination domain is inserted into the website's table before the webpages are inserted into the webpage table. When both the websites and webpages exist in the database, the link is created as outlined in listing 5.1.

Listing 5.1: Inserting a backlink in to the SQLite database.

```
def insert_link(source_domain, source_path, target_domain, target_path):

    source_website = Websites.get(Websites.domain == source_domain)
    source_webpage = Webpages.get(Webpages.website == source_website,
        Webpages.path == source_path)

    target_website = Websites.get(Websites.domain == target_domain)
    target_webpage = Webpages.get(Webpages.website == target_website,
        Webpages.path == target_path)

    Links.insert(source_page=source_webpage, target_page=target_webpage)
        .execute()
```

5.2.3 Neo4j

Inserting is done using the `LOAD CSV` Cypher command as outlined in listing 5.2. The `LOAD CSV` Cypher command is a feature for inserting large amounts of data at once. The subquery `CALL { ... } IN TRANSACTIONS` is used to commit and clear the memory buffer for every 1000 rows inserted. This prevents Neo4j from running out of memory.

Listing 5.2: Cypher command for loading the link data into the Neo4j backlink database.

```
LOAD CSV WITH HEADERS FROM file:///<filename> AS row
FIELDTERMINATOR " "
CALL {
    MERGE (source_page:WEBPAGE {path: coalesce(row.source_page, ""),
    domain: row.source_domain})
    MERGE (target_page:WEBPAGE {path: coalesce(row.target_page, ""),
    domain: row.target_domain})
    MERGE (source_page)-[:LINKS_TO]->(target_page)
} IN TRANSACTIONS;
```

5.3 Querying Backlinks

A fully operational backlink database stores terabytes of data. As the database grows, it is important that all the backlinks for a given URL can efficiently be retrieved. This experiment is intended to test how the backlink database's search performance is affected by the different amounts of backlinks stored in the database. The experiment measures the average throughput when querying for the same 10,000 backlinks five times. The backlinks used in this experiment are a shuffled subset of the already inserted links in order to prevent the insert order to affect the experiment results.

5.3.1 BacklinkDB

The backlinks for a given URL are retrieved using the `SELECT EXACT <url>` statement described in section 3.3.1.

5.3.2 SQLite

Retrieving the backlinks for a given URL is done as outlined in listing 5.3. The query uses two join operations in order to produce the list of backlinks for the given URL.

Listing 5.3: Retrieve all the backlinks for a given URL.

```
def get_backlinks_for_webpage(domain, path):

    # Get target webpage
    target_website = Websites.get(Websites.domain == domain)
    target_webpage = Webpages.get(Webpages.website == target_website,
        Webpages.path == path)

    backlinks = (
        Links.select(Websites.domain, Webpages.path)
        .join(Webpages, on=(Webpages.id == Links.source_page))
        .join(Websites, on=(Websites.id == Webpages.website))
        .where(Links.target_page == target_webpage)
    )

    return backlinks
```

5.3.3 Neo4j

The Cypher outlined in listing 5.4 is used to retrieve a list of all the backlinks for the given URL.

Listing 5.4: Cypher for retrieving all the backlinks for a given URL.

```
backlink_records = session.run(
    "MATCH (external_page:WEBPAGE)"
    "-[:LINKS_TO]->"
    "(webpage:WEBPAGE {domain: $domain, path: $path})"
    "RETURN external_page",
    domain=destination_domain, path=destination_path
)
```

5.4 Space Utilization

How each of the databases stores the backlink data will have an impact on the total size of the database. This experiment is intended to measure the size of the database when different amounts of backlinks are stored. The results will give an indication of how space efficient each of the backlink databases is.

5.5 BacklinkDB Scalability

The BacklinkDB scalability benchmark is designed to analyse how BacklinkDB performs when storing up to 4,000,000 backlinks. Doing analysis on the amount and distribution of the free space will give a more in-depth understanding of how BacklinkDB scales.

5.6 BacklinkDB Fragmentation

The design of BacklinkDB introduces fragmentation as records are inserted. Analyzing the current level of fragmentation occurring in the database is important when evaluating performance. This data can also be used when designing new iterations of the DBMS. Inserting 4,000,000 backlinks and measuring the amount of fragmentation with the domain that has the highest number of backlinks will give insight into how effective BacklinkDB's storage manager is.

5.7 BacklinkDB Profiling

Profiling the execution of the program will give insight into where the Central Processing Unit (CPU) spends most of the time. The results of the profiling will give insight into how efficiently the design and implementation of BacklinkDB is, as hot spots can be identified.

Callgrind[8] is a call-graph generating cache and branch prediction profiler that records instruction statistics on the given executable. In this experiment, we will profile the insert of 20,000 backlinks before retrieving all the backlinks for 10,000 of the links inserted.

8. https://valgrind.org/docs/manual/cl-manual.html

6 Results

This chapter presents the results from the experiments described in the previous chapter. Section 6.1 presents the insert throughput experiment results. Section 6.2 presents the search throughput experiment results. Section 6.3 presents the analysis of the three backlink databases. Section 6.4 presents a more in-depth analysis of BacklinkDB's performance. Section 6.5 presents a fragmentation analysis. Section 6.6 presents the profiling results for BacklinkDB.

6.1 Insert Throughput

Executing the experiment described in section 5.2 produced the results shown in table 6.1, 6.2 and 6.3. The experiments are performed using five different datasets of 100,000, 200,000, 300,000, 400,000, and 500,000 backlinks. Each dataset is inserted five times in order to calculate the average, standard deviation, and relative standard deviation. Figure 6.1 shows the insert throughput results from the three backlink databases using an error plot.

6.2 Querying Backlinks

Executing the experiment described in section 5.3 produced the results shown in table 6.4, 6.5 and 6.6. The experiments are performed using five different

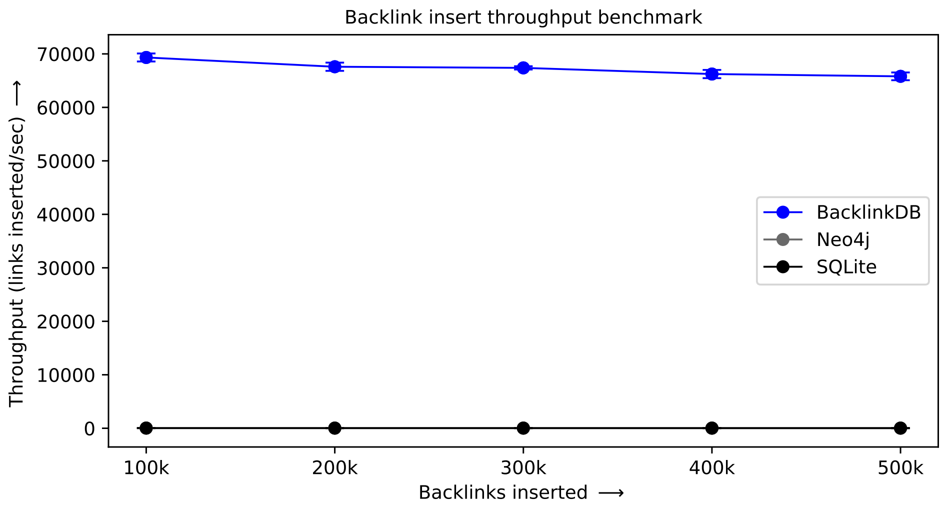


Figure 6.1: Insert throughput results from BacklinkDB, SQLite, and Neo4j.

Backlinks inserted	Avg. throughput	Standard deviation
100,000	69,332 links/sec	794 (1.08%)
200,000	67,598 links/sec	768 (1.14%)
300,000	67,386 links/sec	308 (0.46%)
400,000	66,235 links/sec	770 (1.16%)
500,000	65,806 links/sec	725 (1.10%)

Table 6.1: BacklinkDB - Throughput results from inserting five different amounts of backlink data five times.

Backlinks inserted	Avg. throughput	Standard deviation
100,000	34.25 links/sec	0.37 (1.08%)
200,000	33.21 links/sec	0.38 (1.13%)
300,000	33.49 links/sec	0.28 (0.85%)
400,000	31.80 links/sec	0.58 (1.83%)
500,000	31.60 links/sec	0.66 (2.08%)

Table 6.2: SQLite - Throughput results from inserting five different amounts of backlink data five times.

Backlinks inserted	Avg. throughput	Standard deviation
100,000	16.61 links/sec	0.13 (0.79%)
200,000	8.62 links/sec	0.10 (1.16%)
300,000	5.59 links/sec	0.37 (6.58%)
400,000	3.85 links/sec	0.67 (17.47%)
500,000	3.02 links/sec	0.17 (5.62%)

Table 6.3: Neo4j - Throughput results from inserting five different amounts of backlink data five times.

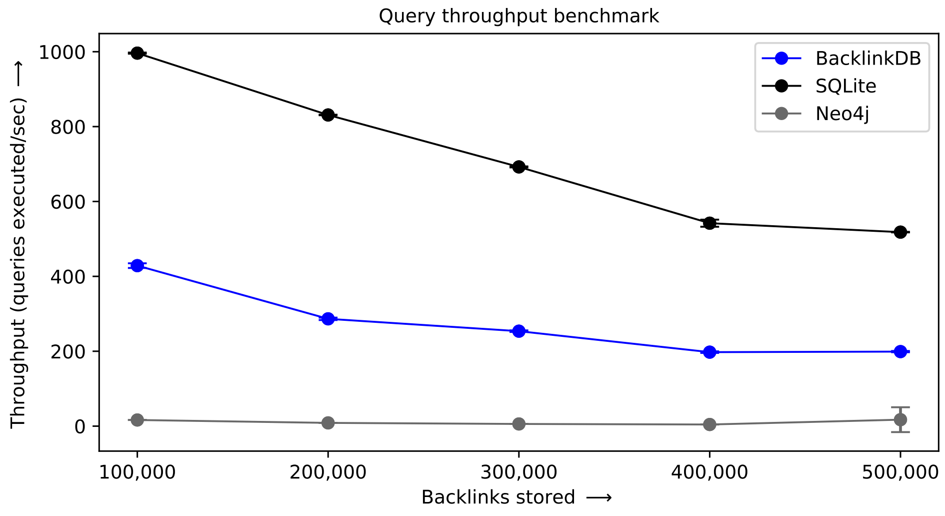


Figure 6.2: Throughput results from querying the three backlink databases managed by BacklinkDB, SQLite, and Neo4j.

datasets stored in the database. The different datasets contain 100,000, 200,000, 300,000, 400,000, and 500,000 backlinks. All the backlinks for 10,000 URLs are retrieved five times in order to calculate the average, standard deviation, and relative standard deviation. Figure 6.2 shows the search throughput results from the three backlink databases using an error plot.

6.3 Space Utilization

The results after executing the experiment described in section 5.4 with five different backlink datasets are shown in table 6.7. The size is found by measuring all the database files from each of the backlink databases after insertion.

Backlinks in database	Avg. throughput	Standard deviation
100,000	428.64 links/sec	6.24 (1.46%)
200,000	286.58 links/sec	3.25 (1.13%)
300,000	253.62 links/sec	2.08 (0.82%)
400,000	197.60 links/sec	2.12 (1.07%)
500,000	199.05 links/sec	1.67 (0.84%)

Table 6.4: BacklinkDB - Throughput results from retrieving all the backlinks from 10,000 different URLs five times.

Backlinks in database	Avg. throughput	Standard deviation
100,000	996.22 links/sec	1.52 (0.15%)
200,000	831.05 links/sec	0.74 (0.09%)
300,000	692.33 links/sec	1.35 (0.20%)
400,000	542.02 links/sec	9.49 (1.75%)
500,000	518.14 links/sec	0.74 (0.14%)

Table 6.5: SQLite - Throughput results from retrieving all the backlinks from 10,000 different URLs five times.

Backlinks in database	Avg. throughput	Standard deviation
100,000	16.39 links/sec	0.07 (0.41%)
200,000	8.66 links/sec	0.02 (0.20%)
300,000	5.80 links/sec	0.02 (0.36%)
400,000	4.35 links/sec	0.01 (0.13%)
500,000	3.84 links/sec	0.01 (0.35%)

Table 6.6: Neo4j - Throughput results from retrieving all the backlinks from 10,000 different URLs five times.

Backlinks stored	BacklinkDB	SQLite	Neo4j
100,000 (15.53 MiB)	20.34 MiB	20.13 MiB	15.41 MiB
200,000 (28.66 MiB)	40.35 MiB	41.70 MiB	30.48 MiB
300,000 (43.40 MiB)	60.56 MiB	63.20 MiB	45.44 MiB
400,000 (58.10 MiB)	80.03 MiB	84.40 MiB	61.35 MiB
500,000 (72.54 MiB)	98.87 MiB	104.88 MiB	74.95 MiB

Table 6.7: The backlink database's total size.

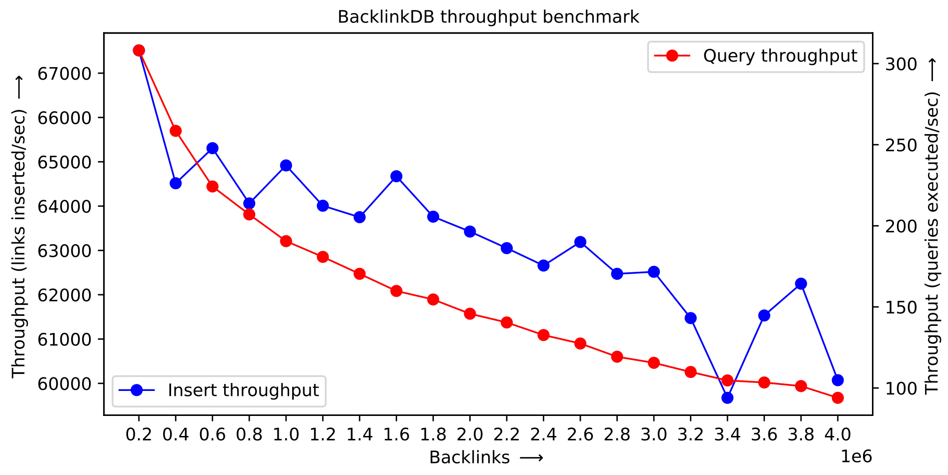


Figure 6.3: BacklinkDB throughput benchmark results. Insert and search throughput when inserting/storing 200,000 to 4,000,000 backlinks. Increments of 200,000 are used.

6.4 BacklinkDB Scalability

Figure 6.3 shows the throughput from inserting 200,000 to 4,000,000 backlinks using increments of 200,000. After each insert, the query throughput is measured by retrieving all the backlinks for 10,000 random URLs using the `SELECT EXACT` statement.

Figure 6.4 shows how the database size increases from 200,000 to 4,000,000 backlinks stored. The free space (shown in black) indicates how much of the database file is not utilized.

When storing 4,000,000 backlinks (586.81 MiB), BacklinkDB's database occupied 704.43 MiB of disk space. This is an increase of 20% in disk space. Measuring the total memory footprint for all the in-memory indices accumulated to 42.17 MiB. This translates to 5,99% of the database's size.

Figure 6.5 shows the block distribution sorted by the amount of free space when 4,000,000 backlinks are stored in BacklinkDB.

6.5 BacklinkDB Fragmentation

In a database with 4,000,000 backlinks, the results from executing the experiment described in section 5.6 shows that *www.facebook.com* was the domain

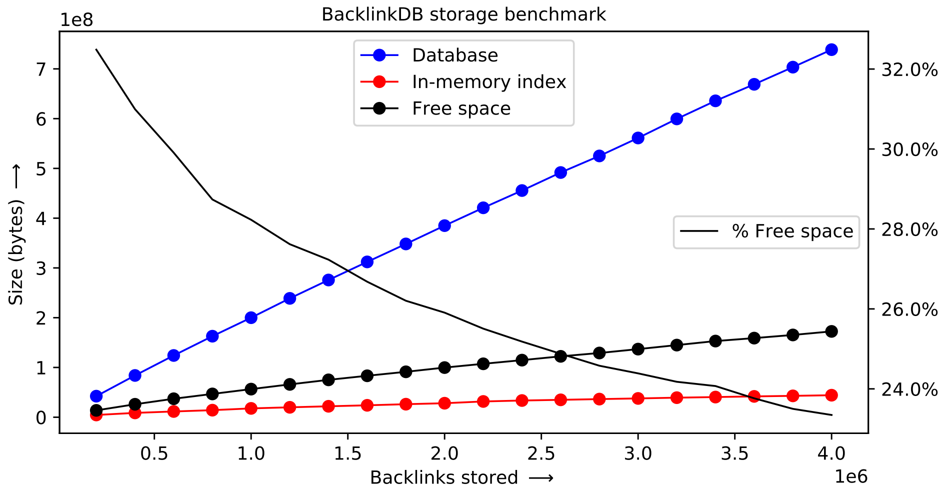


Figure 6.4: BacklinkDB storage benchmark. Database storage analysis when storing between 200,000 and 4,000,000 backlinks. Increments of 200,000 are used.

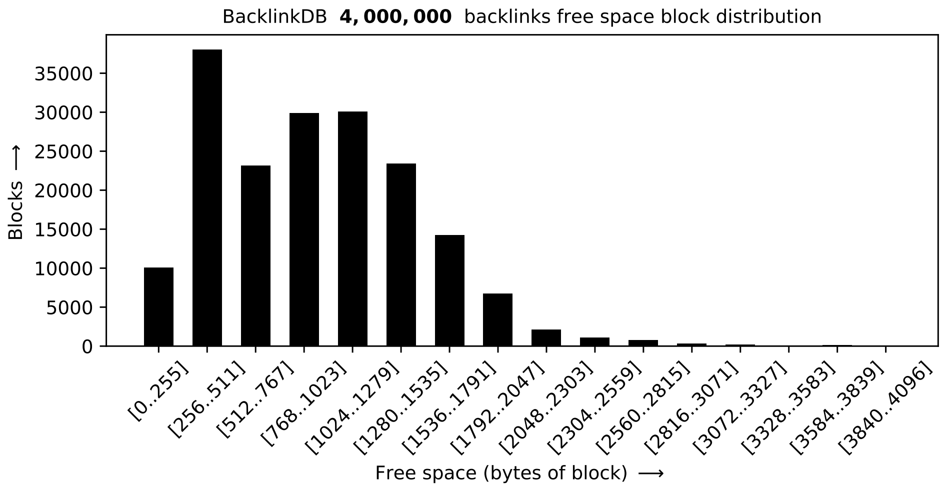


Figure 6.5: Block distribution is sorted by the amount of free space when 4,000,000 backlinks (586.81 MiB) are stored in the database.

Domain	Backlinks
https://www.facebook.com	175,429
https://twitter.com	144,785
https://www.blogger.com	126,390
https://www.instagram.com	93,629
https://www.youtube.com	68,374

Table 6.8: Descending list of the five most back-linked domains in a backlink database consisting of 4,000,000 backlinks.

with the most amount of backlinks. Table 6.8 shows a list of the five domains with the most amount of backlinks.

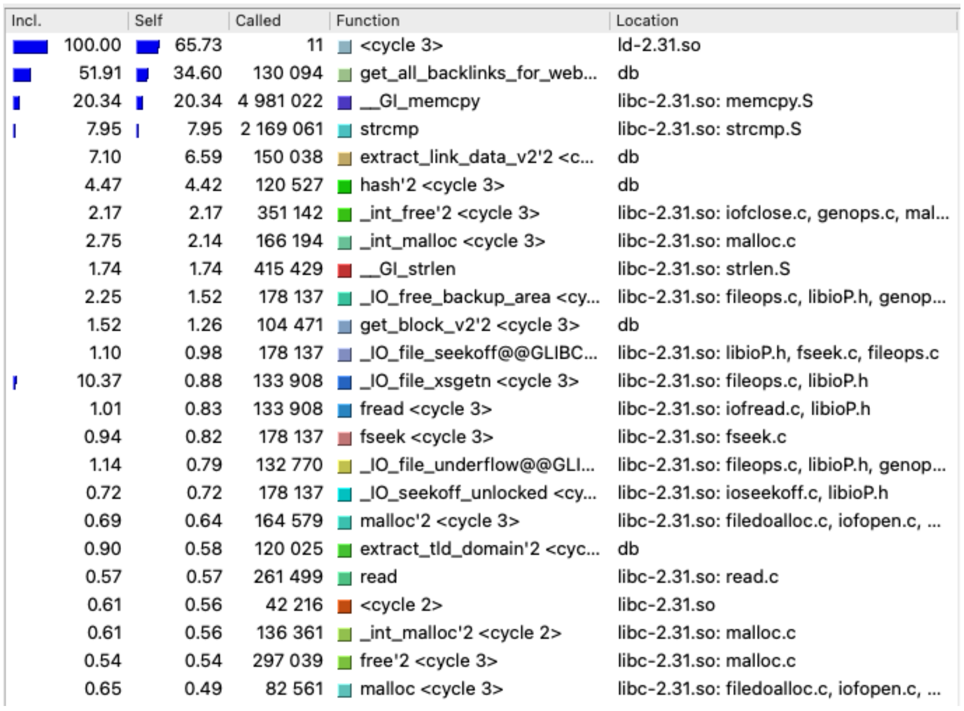
Analyzing the block list for the *www.facebook.com* domain showed that the backlinks are distributed over 22,978 different segments. The distribution of block that the backlinks are stored on showed that there is on average 4.51 blocks between each block containing *www.facebook.com* backlinks.

6.6 BacklinkDB Profiling

This section presents the results from the profiling experiment described in section 5.7. Inserting 20,000 backlinks before retrieving all the backlinks for 10,000 URLs was performed using the Callgrind tool. Figure 6.6 shows a screenshot of the call graph using the Callgrind visualizer: Kcachegrind[1].

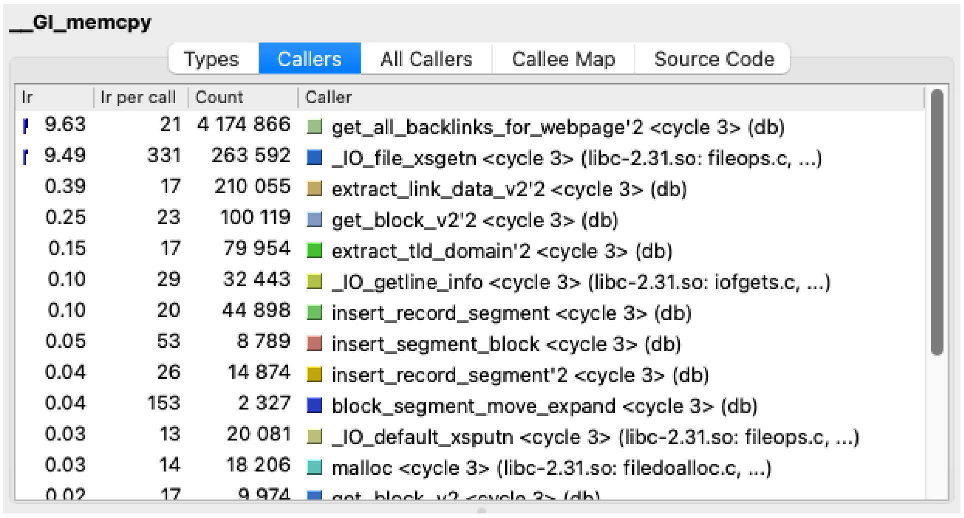
The *Incl.* (Inclusive time) column shows the total amount of time spent in the given function. The *Self* column displays the total amount of time spent in the function, not including the time called from that function. And the *Called* column displays the total number of times the function is called. Figure 6.7 lists the functions that spent most time calling the *memcpy* function.

1. https://kcachegrind.github.io/html/Home.html



Incl.	Self	Called	Function	Location
100.00	65.73	11	<cycle 3>	ld-2.31.so
51.91	34.60	130 094	get_all_backlinks_for_web...	db
20.34	20.34	4 981 022	__GI_memcpy	libc-2.31.so: memcpy.S
7.95	7.95	2 169 061	strcmp	libc-2.31.so: strcmp.S
7.10	6.59	150 038	extract_link_data_v2'2 <c...	db
4.47	4.42	120 527	hash'2 <cycle 3>	db
2.17	2.17	351 142	_int_free'2 <cycle 3>	libc-2.31.so: iofclose.c, genops.c, mal...
2.75	2.14	166 194	_int_malloc <cycle 3>	libc-2.31.so: malloc.c
1.74	1.74	415 429	__GI_strlen	libc-2.31.so: strlen.S
2.25	1.52	178 137	_IO_free_backup_area <cy...	libc-2.31.so: fileops.c, libioP.h, genop...
1.52	1.26	104 471	get_block_v2'2 <cycle 3>	db
1.10	0.98	178 137	_IO_file_seekoff@@GLIBC...	libc-2.31.so: libioP.h, fseek.c, fileops.c
10.37	0.88	133 908	_IO_file_xsgetn <cycle 3>	libc-2.31.so: fileops.c, libioP.h
1.01	0.83	133 908	fread <cycle 3>	libc-2.31.so: iofread.c, libioP.h
0.94	0.82	178 137	fseek <cycle 3>	libc-2.31.so: fseek.c
1.14	0.79	132 770	_IO_file_underflow@@GLI...	libc-2.31.so: fileops.c, libioP.h, genop...
0.72	0.72	178 137	_IO_seekoff_unlocked <cy...	libc-2.31.so: ioseekoff.c, libioP.h
0.69	0.64	164 579	malloc'2 <cycle 3>	libc-2.31.so: filedoalloc.c, iofopen.c, ...
0.90	0.58	120 025	extract_tld_domain'2 <cyc...	db
0.57	0.57	261 499	read	libc-2.31.so: read.c
0.61	0.56	42 216	<cycle 2>	libc-2.31.so
0.61	0.56	136 361	_int_malloc'2 <cycle 2>	libc-2.31.so: malloc.c
0.54	0.54	297 039	free'2 <cycle 3>	libc-2.31.so: malloc.c
0.65	0.49	82 561	malloc <cycle 3>	libc-2.31.so: filedoalloc.c, iofopen.c, ...

Figure 6.6: Descending order of the functions that BacklinkDB spent the most time in.



__GI_memcpy

Types | Callers | All Callers | Callee Map | Source Code

Ir	Ir per call	Count	Caller
9.63	21	4 174 866	get_all_backlinks_for_webpage'2 <cycle 3> (db)
9.49	331	263 592	_IO_file_xsgetn <cycle 3> (libc-2.31.so: fileops.c, ...)
0.39	17	210 055	extract_link_data_v2'2 <cycle 3> (db)
0.25	23	100 119	get_block_v2'2 <cycle 3> (db)
0.15	17	79 954	extract_tld_domain'2 <cycle 3> (db)
0.10	29	32 443	_IO_getline_info <cycle 3> (libc-2.31.so: iofgets.c, ...)
0.10	20	44 898	insert_record_segment <cycle 3> (db)
0.05	53	8 789	insert_segment_block <cycle 3> (db)
0.04	26	14 874	insert_record_segment'2 <cycle 3> (db)
0.04	153	2 327	block_segment_move_expand <cycle 3> (db)
0.03	13	20 081	_IO_default_xsputn <cycle 3> (libc-2.31.so: fileops.c, ...)
0.03	14	18 206	malloc <cycle 3> (libc-2.31.so: filedoalloc.c, ...)
0.02	17	9 974	get_block_v2 <cycle 3> (db)

Figure 6.7: List of the functions that spent most time calling the *memcpy* function.

7 Evaluation

This chapter presents a short evaluation of the results presented in the previous chapter. Section 7.1 evaluates the results from the insert throughput experiment. Section 7.2 evaluates the results from the search throughput experiment. Section 7.3 compares the results from the database space analysis. Section 7.4 evaluates BacklinkDB's scalability. Section 7.5 evaluates the fragmentation in BacklinkDB. Section 7.6 evaluates the profiling results of BacklinkDB.

7.1 Insert Throughput

The results presented in section 6.1 show that BacklinkDB has a major advantage compared to SQLite and Neo4j when inserting backlinks directly from the file. At 500,000 inserts BacklinkDB averaged 69,332 inserts/sec, while SQLite and Neo4j averaged 31.60 and 3.02 inserts/sec. This shows that BacklinkDB has 2,194 times higher throughput at 500,000 inserts compared to SQLite.

SQLite did not see a major drop in throughput as the dataset increased in size. However, the results for Neo4j show a drastic decrease in performance for every 100,000 increments in size. This shows that Neo4j has the least scalable backlink database after evaluating the insert performance.

7.2 Querying Backlinks

The results from experiment 5.3 show that SQLite has a 2.32 times higher average throughput compared to BacklinkDB when retrieving backlinks from a backlink database with 500,000 records. Neo4j suffered the same scalability problems as mentioned in the previous section with a rapid decline in throughput as the datasets increased in size.

7.3 Space Utilization

The accumulated size of all the database files for each of the three backlink databases represented in table 6.7 shows that Neo4j has the best utilization of space. When storing 72.54 MiB of backlink data, the Neo4j database needs 3.32% more storage to represent the data as nodes and edges. With the same dataset, BacklinkDB and SQLite use 36.30% and 44.60% more space when the dataset is stored in the database.

7.4 BacklinkDB Scalability

When analyzing the insert and search throughput from 200,000 to 4,000,000 backlinks, the BacklinkDB's performance decreases. This performance decrease is shown in figure 6.3. The insert throughput still is above 60,000 links inserted/sec, but the search throughput falls just below 100 queries/sec. This shows that BacklinkDB implementation is more scalable when it comes to inserting backlinks compared to doing backlinks retrieval.

The database storage analysis shown in figure 6.4 shows that as the backlink database grows, the free space percentage decreases. This is an indication that the database gets more space efficient as the database grows. The figure shows that the free space percentage converges. This indicates that a majority decrease in the free space percentage will happen at the start of the populating of a full-scale backlink database.

A more detailed analysis of the free space distributed is illustrated in figure 6.5. This figure shows that at 4,000,000 backlinks stored, the majority of free space is relatively smaller chunks distributed on the majority of the blocks in the data files.

7.5 BacklinkDB Fragmentation

Analyzing the fragmentation of the most backlinked domain when storing 4,000,000 backlinks resulted in a block list containing 22,978 elements. With a total of 175,429 backlinks, this translates to an average of 7.63 backlink records stored in each block segment. With an average of 4.51 blocks between each of the block segments for the domain, indicates that a high level of fragmentation occurs in the implementation of BacklinkDB. Section 8.1.1 and 8.1.2 contains further discussions on how defragmentation in BacklinkDB can be achieved.

7.6 BakclinkDB Profiling

The majority of the execution time is spent in the *get_all_backlinks_for_webpage* function as shown in the call graph in figure 6.6. This function retrieves all the backlinks for a given URL, successively placing them in a dynamically allocated block of memory. The call graph also shows that during the execution of the program, a lot of the time is spent copying memory using the *memcpy* function. *memcpy* is a standard C library function for copying a chunk of memory to another specified location. Figure 6.7 shows that during the execution, the function *get_all_backlinks_for_webpage* called *memcpy* 4,174,866 times. This shows that the *memcpy* calls introduce a hot spot. This issue is discussed further in section 8.4.2.

8 Discussion and Future Work

In this chapter, we will discuss BacklinkDB as a solution for managing a backlink database. This chapter will also discuss BacklinkDB's shortcomings and present possible improvements to the design. Section 8.1 discusses possible optimization improvements to BacklinkDB's design. Section 8.2 contains a short discussion comparing BacklinkDB to SQLite. Section 8.3 contains a short discussion comparing BacklinkDB to Neo4j. Section 8.4 discusses performance and failure handling. Section 8.5 discusses the advantages and disadvantages of building a purpose-built backlink DBMS.

8.1 Optimizations

When building a purpose-built DBMS from scratch, the goal is to create a system that is more efficient at managing the data compared to other existing off-the-shelf solutions. In a backlink database, we are not dealing with a complex data model, but we are dealing with a lot of data. Scalability and performance become the main focus when we are creating a system that is going to store terabytes of data. This section will discuss some of the shortcomings and improvements of the design presented in this thesis.

8.1.1 Adjacency List

Storing the backlinks on disk using an adjacency list will drastically decrease the total size of the database and reduce the search time. In the current design, each backlink record is defined using a source URL and a destination URL. This implies that if there are more than two backlinks for the same URL, the destination URL is redundantly stored $(n - 1)$ times in the database. Using an adjacency list will eliminate the redundant copies of the destination URLs.

The search time will be reduced when using an adjacency list as there is no need to inspect all the destination URLs when querying for backlinks. The search will stop when the adjacency list array entry for the given URL is found. This will still present a worst-case time complexity of $O(n)$, but by sorting the backlinks, a binary search can be used, reducing the worst-case time complexity when searching for all the backlinks for a given URL to $O(log\ n)$.

Adding the adjacency list functionality would require some redesign of the segment structure, as the backlink data for a single URL can't exceed the maximum size of a block segment. Allowing adjacency list elements to overflow into other segments would require additional metadata about the records to be stored, but could potentially reduce the overall size of the database significantly.

Storing the backlinks using a sorted adjacency list will decrease the insert performance. Sorting destination URLs when inserting backlinks will introduce more overhead, potentially shifting existing backlink records when an inserting operation is performed. Adding the sorted adjacency introduces the trade-off between less insert performance and higher search throughput.

8.1.2 Defragmentation

With the current design of BacklinkDB, the distribution of the disk blocks storing backlinks for a given domain can be sparse as shown in section 6.5. This is because the storage manager allocates a new block at the end of the data file when inserting backlinks. Having a block list containing multiple segments of data that are not stored consecutively on the data file requires multiple consecutive systems calls to read the data. By defragmenting the data files would reduce the total number of system calls required when retrieving backlinks. This would also drastically reduce the memory of the storage index, as the block list would only contain a defined range of segments on the data file, instead of a list of all the different segments.

In section 6.5 the *www.facebook.com* domain had its backlinks distributed over 22,978 different blocks. This translates to a block list of 22,978 elements defining a block number, a block offset, and a segment size. By adding the defragmentation functionality to BacklinkDB, the backlink data for *www.facebook.com* could be referenced by a range of blocks on the data file, reducing the size of the in-memory storage index.

8.1.3 Simple backlink record compression

All webpages use the Hypertext Transfer Protocol (HTTP) or Hypertext Transfer Protocol Secure (HTTPS) to transfer the contents of the webpage. These protocols are found at the start of every URL with the `http://` and `https://` prefix. For more efficient space utilization, the protocol can be represented using one uniquely identifiable byte value instead of 7-8 bytes, for each backlink. This simple compression scheme will have an measurable impact on the total database size when storing billions of backlinks. The string manipulation of prefixing 7-8 bytes of characters should not have a major impact on performance for each backlink retrieved.

8.1.4 Memory Fragmentation

As described in section 4.5 and 4.6, when inserting and retrieving backlinks, both operations include doing a lookup in the TLD index and the storage index. The implementation details of the storage index and TLD index can have an impact on the performance. Because both the *search key* and its *value* point to separate locations in memory, a cache hit is not guaranteed when reading the *value* after the *search key* is located.

A possible increase in performance could be achieved by adding a custom memory allocator to BacklinkDB's system structure. When the CPU accesses a value from an address in memory, the CPU loads a chunk of memory into the cache. A custom allocator can optimize the placement of the *search key* and its *value* so that it is located next to each other in the memory address space. This will increase the probability of cache hits occurring when doing lookups in the TLD and storage index.

8.1.5 Buffer Pool

Caching disk blocks in memory can allow for faster reads in some instances. The design presented in this thesis does not present a system explicitly for buffering the blocks in which the backlink data resides. The data is loaded from the disk on demand by the *storage manager*.

Adding caching to BacklinkDB would be beneficial if the backlinks for some of the URLs and domains are requested more frequently than others. Implementing a page cache system will increase the read performance for that particular subset of domains and URLs. While a caching system would boost the performance of some queries, it will require BacklinkDB to consume more memory at runtime. A trade-off decision need to be made between the amount of memory BacklinkDB will consume and the percentage of queries that will have an increase in response time.

8.1.6 Multiple connections

The benefit of having a server that manages the connected clients is that BacklinkDB potentially can support multiple client connections simultaneously. This can be achieved by continuously iterating over all the connections and processing the queries as they are received.

8.1.7 Distributed Data Store

An advantage of BacklinkDB's design is that the backlink for a given domain is located using the hash value of that domain. This introduces the opportunity of adding distributed data storage functionality without making any major changes to the core database design.

Partitioning the data by having multiple distributed instances of BacklinkDB that each is responsible for a range of hash values would increase both the throughput of insert and the throughput when retrieving backlinks. While a distributed design can increase overall performance, a distributed data store presents new challenges and trade-offs that are described in the CAP theorem[3]. CAP presents the three guarantees: *consistency*, *availability*, and *partition tolerance*, and the theorem states that in a distributed data store, you can only ensure two of the three guarantees.

8.2 BacklinkDB vs SQLite

The results presented in this thesis show that SQLite outperformed BacklinkDB when searching for and retrieving backlinks. Even though SQLite needed to perform two join operations when processing the retrieval of backlinks for a given URL, it has more than double the throughput compared to BacklinkDB. However, when inserting data, BacklinkDB outperformed SQLite.

Neo4j and BacklinkDB use sockets to connect to the database while SQLite writes and reads directly to the file system. Not having to communicate over sockets allows SQLite to have less communication overhead as database queries are invoked directly from a library.

8.3 BacklinkDB vs Neo4j

Neo4j's throughput performance was significantly lower compared to BacklinkDB and SQLite. Because the world wide web can be represented as a graph, it is natural to store link data in a graph database. However, when dealing with large amounts of backlink data, Neo4j did not appear to be a scalable system for managing a backlink database.

Neo4j stores records in linked lists on the disk, this provides the benefits of having much better space utilization as outlined in section 6.3. However, this alone does not make Neo4j a feasible DBMS for managing a backlink database.

8.4 BacklinkDB

Because of the amount of data needed to be stored in an operational backlink database, a high insert throughput is necessary. BacklinkDB has demonstrated that it has the potential of sustaining high throughput as the database scales. Section 8.1.7 presents a discussion on how BacklinkDB can operate as a distributed data store, which will not only increase the insert throughput but the throughput when retrieving backlinks as well.

Designed for only storing backlink data, BacklinkDB utilizes the space better than SQLite. Section 8.1.1 discusses how to add database normalization using adjacency lists. This optimization should further improve the space utilization of BacklinkDB.

8.4.1 Failure handling - Transactions

Neo4j and SQLite both support ACID transactions. Supporting these properties ensures the validity and consistency of the database in the event of a failure. BacklinkDB does not support ACID transactions. As this thesis aims to explore the benefits of building a purpose-built DBMS, support for ACID transactions was not implemented. In the benchmark results presented in section 6.1 BacklinkDB is compared against two transactional databases. Introducing ACID transactions to BacklinkDB would introduce extra overhead to each backlink inserted.

Support for transactional queries in BacklinkDB would introduce additional precautionary steps when a backlink is inserted. BacklinkDB is designed to be space efficient, meaning that the database will reorganize the segments as backlink records are inserted into the database (section 4.5.2). Because of this optimization, extra logging for each of the steps in the reallocating process is necessary in order for the database to be able to recover after a failure that happens during a reallocation.

Because the workload of a backlink database resembles an analytical database more than a transactional database, supporting transactions may introduce unnecessary overhead. The backlink data are inserted from files which means that if BacklinkDB has a failure while the backlink database is building, the backlink database can be rebuilt without any permanent loss of data.

8.4.2 Profiling Results

The profiling results presented in chapter 6.6 shows that BacklinDB spends a lot of the execution time copying memory using the *memcpy* function. This is an interesting finding because the bottleneck in most DBMSes often occurs as they are accessing the disk. Reducing the overall use of the *memcpy* function should be prioritized in future versions in order to improve the overall performance.

8.4.3 Space Utilization

Dividing blocks into segments results in additional complexity and overhead. The benefits of assigning a chunk of disk space to a set of backlinks for a specific domain allow for efficient lookup when combined with a hash index. The reason for sectioning a block into segments is so that we can reduce the total amount of free space.

Without the concept of block segments, entire 4096-byte size blocks are allocated each time a domain needs more space to store backlinks. One backlink record will almost never occupy the full size of a block, leaving a major section of the newly allocated block as free space. This free space can only be used to store the backlinks for that specific domain. The web consists of hundreds of millions of domains. Not accounting for this issue could result in a relatively high percentage of free space.

8.5 Purpose-Built Backlink DBMS

After conducting the experiments and evaluating the implementation of the design, BacklinkDB reveals both advantages and disadvantages. The previous sections in this chapter show that the underlying architecture can be adapted to overcome the shortcomings and potential challenges that will occur when scaling.

This thesis shows that when managing large amounts of data, using a purpose-built DBMS can be beneficial. With a purpose-built design, we have the ability to choose what performance metrics to focus on. BacklinkDB showed high insert performance and medium search throughput and space efficiency. Previously presented in this chapter are some potential improvements that will increase the search throughput and space efficiency. However, most of these improvements will introduce a lower insert performance.

9 Conclusion

This thesis presented BacklinkDB, a purpose-built database management system. The interface of BacklinkDB allows users to do a batch insert directly from files. Support for listing all the backlinks for a given webpage is achieved through a simple SQL-inspired query language.

Analysis of BacklinkDB's performance shows that a purpose-built DBMS can deliver a significantly higher throughput when inserting backlinks compared to SQLite and Neo4j. The search throughput for BacklinkDB is approximately 39% of that what SQLite achieved. However, this thesis presents design improvements that will increase BacklinkDB's search performance. Neo4j is the best DBMS of the three when it comes to utilizing disk space. Using techniques such as adjacency lists to store records on disk, BacklinkDB can potentially achieve better disk utilization.

Evaluation of the BacklinkDB show that building a purpose-built DBMS for managing a backlink database will have better resource utilization at scale compared to a off-the-shelf database management system.

Bibliography

[1] Albert Bifet, Carlos Castillo, Paul-Alexandru Chirita, and Ingmar Weber. An analysis of factors used in search engine ranking. In *AIRWeb*, pages 48–57, 2005.

[2] D. E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, Paul R. Young, and Peter J. Denning. Computing as a discipline. *Commun. ACM*, 32(1):9–23, jan 1989.

[3] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, jun 2002.

[4] Antonio Gulli and Alessio Signorini. The indexable web is more than 11.5 billion pages. In *Special interest tracks and posters of the 14th international conference on World Wide Web*, pages 902–903, 2005.

[5] Bernard J Jansen, Danielle L Booth, and Amanda Spink. Determining the informational, navigational, and transactional intent of web queries. *Information Processing & Management*, 44(3):1251–1266, 2008.

[6] Bob Jenkins. A new hash function for hash table lookup. *Dr. Dobb's Journal*, 1997.

[7] Cheng-Jye Luh, Sheng-An Yang, and Ting-Li Dean Huang. Estimating google's search engine ranking function from a search engine optimization perspective. *Online Information Review*, 40(2):239–255, 2016.

[8] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[9] American National Standards Institute. Standards Planning and Requirements Committee. Study Group on Data Base Management Systems. *Interim Report: ANSI/X3/SPARC Study Group on Data Base Management*

Systems. ACM, 1975.

[10] Ravi Sen. Optimal search engine marketing strategy. *International Journal of Electronic Commerce*, 10(1):9–25, 2005.