Bachelor's Thesis Computing Science

Radboud University Nijmegen

Dynamic Full-Text Indexing on Lakehouse Storage

Author:
Sam Struthers
s1091108

First supervisor/assessor:
prof. dr. ir. A.P. de Vries

Second assessor:
prof. dr. ir. D. Hiemstra

Daily supervisor:
G.A.W. Hendriksen

June 2, 2026

Abstract

Maintaining full-text search indexes over large, evolving web-crawl collections is a fundamental challenge in information retrieval. Handling standard takedown requests traditionally requires fully reindexing the corpus, which consumes significant computational resources and demands considerable investment in hardware. We build on the idea to use a database within a lakehouse infrastructure, specifically DuckLake, for implementing full-text ranking models. This infrastructure manages metadata automatically and provides full ACID compliance, enabling dynamic index updates without requiring full file rewrites.

To demonstrate feasibility, we designed an index schema replicating a standard information retrieval setup and implemented a disjunctive Okapi BM25 ranking algorithm using pure SQL. We explore index usage under updates by simulating sequential document deletions, relying on DuckLake's merge-on-read strategy to process the modifications.

Empirical evaluation demonstrated that query time displays a linear increase relative to the deletion percentage, heavily influenced by the I/O overhead of reading delete files. To mitigate this, we implemented a storage optimisation and compaction strategy using data file rewrites. This periodic maintenance effectively resets the accumulated read overhead, transforming the query performance profile into a manageable "sawtooth" pattern and keeping the query performance closer to its optimum.

We conclude that the increased flexibility of a database, paired with straightforward storage compaction, makes the lakehouse infrastructure a suitable setup for implementing ranking models using SQL under updates, without significant performance drawbacks.

Contents

1 Introduction **3**

2 Preliminaries **5**
2.1 Data Lakes, Data Warehouses, and the Lakehouse Paradigm . 5
2.2 Apache Parquet 6
2.3 Apache Iceberg 6
2.4 DuckLake 7

3 Related Work **10**
3.1 Okapi BM25 for full-text search 10
3.2 Ranking Models in Column-Store Databases 11
3.3 Conjunctive vs. Disjunctive Query Evaluation 11
3.4 Updating Inverted Indexes 12

4 Index Maintenance using Lakehouse Infrastructure **13**
4.1 Overview 13
4.2 Indexing 14
4.2.1 Creating index tables 14
4.2.2 Managing the Index with DuckLake 16
4.3 Full-Text Search (FTS) Strategy 17
4.3.1 Ranking models using SQL 18
4.4 Dynamic Index Implementation 18
4.4.1 Deleting documents from the index 19
4.4.2 Inserting documents into the index 20
4.4.3 Modifying pre-existing documents 22
4.5 Index Maintenance 22
4.6 Storage Optimisation and Compaction 23

5 Evaluation and Results **24**
5.1 BM25 full-text search complexity analysis 24
5.2 Empirical evaluation (no optimisation) 26
5.3 Empirical evaluation (storage optimisation and compaction) . 28

6 Conclusions **31**
6.1 Summary of Contributions 31
6.2 Limitations 31
6.3 Future Work 32

Chapter 1

Introduction

Information retrieval researchers and practitioners have long relied on specialised, custom-built data structures for document ranking [1]. While databases are defined as a comprehensive collection of related data organised for convenient access, they are not the conventional approach to document ranking. Many traditional databases are optimised for structured, transactional workloads and rely on proprietary storage formats that require rewriting files in order to apply updates. This makes them suitable for applications operating on moderate amounts of data, but document ranking for web-crawl data quickly exceeds the scale these architectures were designed for. Furthermore, traditional database architectures are not designed to handle the volume of unstructured data present in web-crawl corpora.

In research systems for web retrieval, web-indexes have traditionally been treated as static structures, which fails to reflect the ever-changing nature of the internet. Modern indexes must be flexible and dynamic to accurately represent web-data. For example, takedown requests arise regularly when entities update their websites or wish to be removed from an index. The conventional approach is to fully reindex the corpus without the affected data, which is a time-consuming and computationally expensive operation.

We propose implementing ranking models using a database within a lakehouse infrastructure, specifically DuckLake. This infrastructure allows data to be stored externally in standard, open-source file formats like Parquet. Metadata is automatically managed, providing full ACID compliance while enabling data updates and index modifications without requiring full file rewrites. We ask: can DuckLake's SQL-based lakehouse infrastructure support practical full-text ranking and dynamic index maintenance on web-crawl data without full reindexing? The primary focus of this thesis is to demonstrate the feasibility of this approach through a series of experiments. We argue that the flexibility of a database, combined with the relative ease of optimising performance through storage compaction, makes this a suitable setup for implementing full-text ranking models using SQL. The full imple-

mentation is publicly available at [2].

The remainder of this thesis is structured as follows. Chapter 2 introduces the background concepts forming the base of this work, covering the lakehouse paradigm, Apache Parquet, Apache Iceberg, and DuckLake. Chapter 3 reviews the related work on full-text ranking models, query evaluation strategies, and dynamic index maintenance that motivate our approach. Chapter 4 describes the design and implementation of the dynamic index and its associated maintenance operations. Chapter 5 presents a theoretical analysis of full-text search and the empirical evaluation — analysing query performance under sustained deletions and the effect of periodic storage compaction. Chapter 6 concludes and outlines directions for future work.

Chapter 2

Preliminaries

2.1 Data Lakes, Data Warehouses, and the Lakehouse Paradigm

Data lakes are designed to store large volumes of data in cheap object storage; the cheap storage enables the "schema-on-read" approach as it is affordable to store data without processing it, and thus only imposing the structure when querying the data. Any kind of data in any format (structured, semi-structured, unstructured) can be stored without enforcing a schema.

A data warehouse stores highly structured, processed data. In contrast to a data lake, a data warehouse uses a "schema-on-write" approach — where data is transformed and structured before being stored, adhering to a predefined schema. The highly organised schema ensures consistency and reliability, enabling fully ACID compliant transactions. This comes at the cost of flexibility and scalability: warehouses store data in proprietary formats and require considerable preprocessing before ingestion, making them poorly suited for web-crawl data at scale.

A data lakehouse is a modern architecture that combines the scalability and flexibility of a data lake with the structured performance and reliability of a data warehouse [3]. By utilising both the "schema-on-read" and "schema-on-write" approaches, a lakehouse supports raw data ingestion for flexibility, and structured datasets for traditional analytics. It allows the storage, management and analysis of structured, semi-structured and unstructured data in a single system.

Dynamic index maintenance requires a unified storage infrastructure that can accommodate both the raw web-crawl data and the structured index tables. The lakehouse model is particularly well-suited to this because it provides ACID-compliant modifications without requiring full file rewrites — a capability enabled by a metadata layer over standard open formats, which is precisely the approach DuckLake takes.

2.2 Apache Parquet

Apache Parquet is an open source, column-oriented data file format designed for efficient data storage and retrieval [4]. The columnar format means that each column is stored separately on disk, rather than row-by-row as in traditional CSV, JSON, or transactional databases. This makes it especially suited for analytical workloads like BM25 scoring, which primarily accesses only a few columns — such as `termid`, `docid`, and `tf` — when processing data. When analysing large datasets such as a web crawl index, this is essential because the computational costs of loading the full dataset into memory at runtime are extensive; columnar storage means we can load only the columns required for computation.

Due to the constraints of the columnar layout and sequential compression and encoding, Parquet files are immutable [4]. This is a significant drawback as it means a full file rewrite is required in order to update the source data, making Parquet typically only suited to static datasets. In the context of index maintenance, this immutability is the core constraint the system must address. DuckLake's metadata layer is designed to work around this limitation.

2.3 Apache Iceberg

Before open table formats were introduced, managing a data lake was an entirely manual process — engineers had to track which files contained which data, handle concurrent writes themselves, and implement custom snapshot and versioning logic. Apache Iceberg was developed at Netflix to address this at scale, introducing table-level ACID guarantees on top of raw object storage [5].

Iceberg achieves this through a layered metadata chain, illustrated in Figure 2.1. Each table is tracked through a hierarchy of files: a root metadata file points to a manifest list, which in turn points to a set of manifest files, which finally point to the underlying data files [3]. The catalog database stores only a pointer to the current root metadata file. As datasets grow and changes accumulate, this file-based chain becomes increasingly unwieldy — each new snapshot appends a new root metadata file, manifest list, and set of manifest files to the hierarchy.

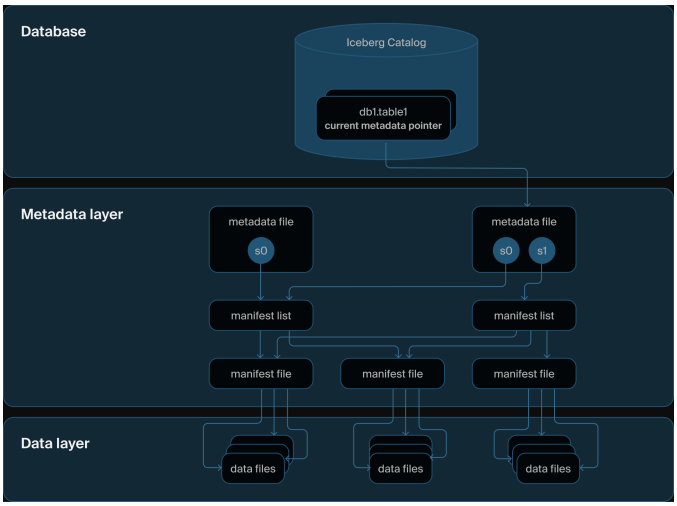

Figure 2.1: Iceberg Catalog Architecture [3]

As snapshots accumulate, the overhead of navigating this chain grows proportionally, and the catalog architecture becomes difficult to maintain at scale. This motivates the DuckLake approach, which replaces the file-based hierarchy with a single SQL database.

2.4 DuckLake

DuckLake addresses the limitations of file-based catalog systems by consolidating all metadata into a single SQL database [3]. Rather than maintaining a chain of JSON and Avro metadata files as Iceberg does, DuckLake stores all table and snapshot metadata in a multi-table relational catalog — simplifying the architecture while retaining full ACID compliance.

DuckLake offers lakehouse functionality for a data lake containing data in open formats such as Parquet, using a single SQL database to contain all metadata — in contrast to the more complex file-based systems seen by its competitors. Transactions in DuckLake are carried out through SQL queries, whilst being fully ACID-compliant. Its infrastructure may therefore also be deployed for index maintenance, due to its inherent flexibility when modifying data stored in DuckLake. Parquet files cannot be modified in place [4]; but through DuckLake's efficient metadata management, modifications are stored as SQL transactions in separate data files and a merge-on-read strategy is implemented when retrieving the data. This ensures that the changes are being applied to the original data when information is being retrieved.

Instead of using metadata files to track data files and modifications, DuckLake uses a single, multi-table database to store all metadata. This is called the metadata catalog (Figure 2.2). It simplifies the architecture by

unifying all the metadata in a single location, and allows interactions with the data through standard SQL operations.

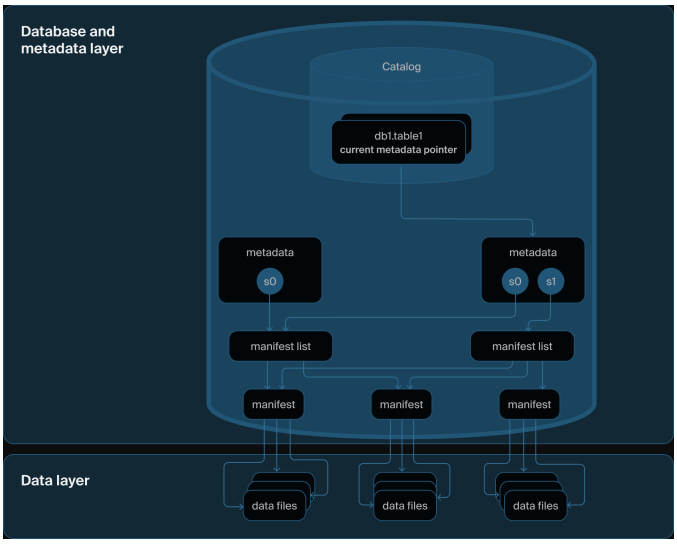

Figure 2.2: DuckLake Architecture [3]

Figure 2.3 details the structure of the catalog database, and how it interacts with the data layer in order to keep track of each data file's metadata.

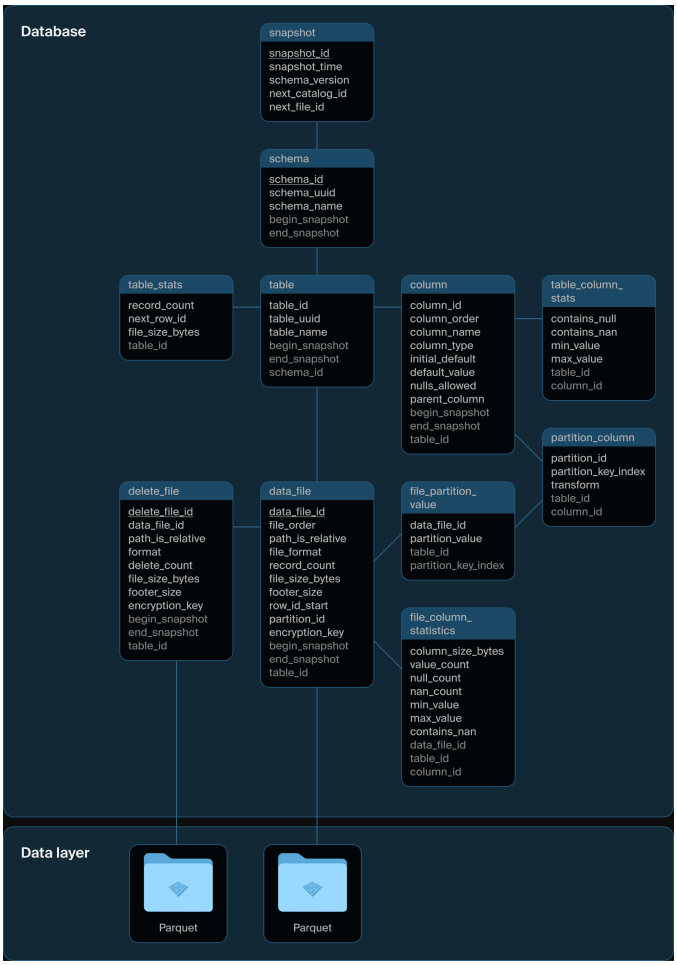

Figure 2.3: DuckLake Schema [3]

For this thesis, DuckLake's SQL interface and merge-on-read strategy make it particularly well-suited to implementing and dynamically maintaining a full-text index on web-crawl data — providing the transactional guarantees of a warehouse over the open-format storage of a lake.

Chapter 3

Related Work

3.1 Okapi BM25 for full-text search

Full-Text Search (FTS) is the primary query mechanism used in web search engines. Queries can consist of multiple terms, and the goal is to return the most relevant documents ranked by relevance. The standard approach to FTS is to use the classic BM25 formula. Okapi BM25 was first introduced in 1994 by Robertson at the TREC conference [6] and developed further to produce the classic BM25 algorithm [7] which is a widely accepted standard for FTS. BM25 works by ranking a set of documents based on the query terms appearing in each document, regardless of their proximity within the document. BM25 represents a family of scoring functions with slightly different components and parameters, but for this project we will be using one of the more common instantiations of the function.

Given a query Q, containing terms $q_1, \ldots, q_n$, the BM25 score of a document D is computed as follows:

$$score(D, Q) = \sum_{i=1}^{n} IDF(q_i) \cdot \frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{avgdl})} \tag{3.1}$$

where $f(q_i, D)$ is the number of times a term q_i appears in a document D, $|D|$ is the number of terms in document D (document length), and $avgdl$ is the average document length in the corpus from which the documents are drawn.

k_1 and b are free parameters. Given the scope of this project, we chose values from the range that Robertson deemed "reasonably good in many circumstances" [7] ($0.5 < b < 0.8$ and $1.2 < k_1 < 2$) and we do not perform parameter optimisation, neither do we explore the accuracy of search results.

The classic BM25 term-weighting and document scoring function proposed by Robertson defines the IDF weighting as follows:

$$IDF(q_i) = \log \frac{N - n(q_i) + 0.5}{n(q_i) + 0.5} \tag{3.2}$$

where N is the total number of documents in the corpus and $n(q_i)$ is the number of documents containing q_i. This formula can cause some issues to arise. If a term appears in more than half of the documents then the value inside the log becomes less than 1 resulting in a negative IDF. Whilst this is theoretically correct as seeing a very common term actually reduces the probability of relevance compared to not seeing it, it can break search ranking as a document could end up with a negative overall score. In our project we use a commonly implemented adaptation to smooth the score. We do this by adding 1 inside the log to ensure that all IDFs are positive. IDF is hence computed as follows:

$$IDF(q_i) = \log(\frac{N - n(q_i) + 0.5}{n(q_i) + 0.5} + 1) \tag{3.3}$$

The smoothed IDF formula is used throughout the implementation described in Chapter 4.

3.2 Ranking Models in Column-Store Databases

A common assumption in IR is that document ranking is a task best suited outside of databases, implemented in custom data structures. Mühleisen et al. [1] challenged this by demonstrating that classic IR ranking models, including BM25, can be implemented as SQL queries on a column-store database (MonetDB), achieving competitive performance with specialised IR systems. In this thesis we take the same SQL-based approach but extend it to dynamic indexes stored in a lakehouse infrastructure. Mühleisen et al. proved it works for static indexes; this thesis shows that its functionality can be extended to dynamic index maintenance as well.

3.3 Conjunctive vs. Disjunctive Query Evaluation

Asadi & Lin [8] explored the effectiveness/efficiency tradeoffs between conjunctive and disjunctive BM25 query evaluation. Their results demonstrated that conjunctive query evaluation was substantially faster than disjunctive, whilst only producing a modest drop in retrieval effectiveness. Disjunctive evaluation is exhaustive but expensive, scoring every document containing at least one query term; conjunctive evaluation prunes the candidate set aggressively at the cost of recall. This thesis deliberately uses disjunctive query evaluation to stress-test the infrastructure under a heavier computational load. Queries are generated by randomly sampling terms from the index and combining them to form a varying bag-of-words query string.

3.4 Updating Inverted Indexes

The problem of maintaining a dynamic inverted index has been studied extensively in the classical IR literature. Zobel and Moffat [9] provide a comprehensive survey of inverted file structures for text search engines, covering both static index construction and the challenges of dynamic index maintenance under document insertions and deletions.

Büttcher and Clarke [10] investigated this problem for retrieval systems with a high query load. They identify two principal update strategies: re-merge approaches, which periodically rewrite the index to incorporate changes, and in-place update strategies, which modify posting lists directly. They proposed a Hybrid Immediate Merge strategy combining both, reducing total index update overhead by up to 73% compared to a pure re-merge baseline, whilst preserving query performance.

In this thesis we address the same problem of dynamic index maintenance, but through a different mechanism. Rather than managing posting list contiguity at the index level, we leverage DuckLake's merge-on-read storage strategy, where updates are expressed as delete markers on immutable Parquet data files. These delete records accumulate between compaction events; at each compaction, the data files are rewritten to physically remove deleted rows. This is functionally analogous to the re-merge strategy described by Büttcher and Clarke, but handled entirely at the storage layer.

Chapter 4

Index Maintenance using Lakehouse Infrastructure

4.1 Overview

Index maintenance involves regularly rebuilding or reorganising database indexes; in the context of web-indexes including regular updates ensuring the information in the index is correct. Although web-indexes in academic research have traditionally been treated as static structures, this does not reflect the ever-changing nature of the internet. This means that an index should be flexible and dynamic in order to more accurately represent web-data. Takedown requests are a common occurrence which can happen when an entity, whose web-data has been crawled and indexed, makes changes to their website, or no longer wishes for their data to be reachable, and thus requests for it to be removed from the index. The conventional approach is to reindex the corpus without the affected data [3], which is a time-consuming and computationally expensive operation given the scale of a web-crawl corpus.

In contrast, DuckLake's merge-on-read strategy enables us to make simple SQL transactions when interacting with the metadata catalog in order to remove data from a file. This means that we can simply identify the information related to the data in the takedown request and delete it from associated tables and statistics. DuckLake then stores these transactions in separate data files, tracked in the catalog, which get merged to the original data when the data is being used. Clients retrieve information through the DuckLake client by sending SQL queries. This means that as soon as the index or data files have been modified or updated the changes will be instantly visible to the clients using the data. DuckLake is optimised for merge-on-read efficiency making it an ideal candidate for the experimentation of using a lakehouse infrastructure for index maintenance.

4.2 Indexing

The scope of this project is targeted towards the index of a web-crawl data, which will be stored in an open format, namely Parquet files. Sample web-crawl data was imported into DuckLake to reproduce a conventional scenario; namely, full-text search on unstructured data stored in a data lake. This scenario requires an index that classic ranking algorithms can be used on.

The index schema used for this project (Figure 4.1) was designed in accordance with the requirements of the classic BM25 ranking algorithm for full-text search.

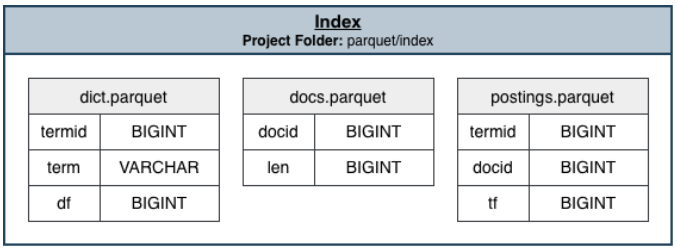

Figure 4.1: Index Schema

4.2.1 Creating index tables

`my_ducklake.data` is imported from the web-crawl files into DuckLake, which stores it in its internal format (managed Parquet files) in a designated location. This location can either be local or external object storage; for the purposes of this project we store the data locally.

Exploiting the nature of DuckLake allows us to implement an indexing algorithm using pure SQL; interacting with the data tables by simply using SQL queries, without needing imperative code or intermediate file exports. This makes the described implementations in this report generalisable, and the rational behind the query semantics more comprehensible. The complete implementation is available at [2].

The first step of indexing the web-crawl data is to extract the terms from each document. To do this we first use a basic regex parser (Listing 4.1) to remove any whitespace and punctuation; separating the terms based on adjacent alphabetical characters. The scope of this project is to demonstrate the performance and feasibility of using a lakehouse infrastructure for index maintenance and thus this simple method for term extraction is suitable to this end. There are many more complex methods for term extraction which may provide more accuracy but as term extraction was not the focus of the project a more primitive method was chosen for simplicity.

```
1 CREATE OR REPLACE TEMP VIEW v_token_stream AS
2     SELECT
3         docid,
```

```
        UNNEST(regexp_extract_all(lower(content), '[a-z]+')) AS term
    FROM my_ducklake.data
```

Listing 4.1: Regex content parsing

Listing 4.1 provides a function mapping $docid \rightarrow \{term\}$. Each *term* string will be assigned a distinct `termid` — for every extracted *term* in the corpus, no two `termid`'s can be assigned to the same *term*. The mapping of $termid \rightarrow term$ is stored in a single row in the `dict` table, where `df` is incremented based on how many distinct documents a term appears in. Listing 4.2 details the SQL used to carry out this operation directly within the DuckLake schema.

```
CREATE OR REPLACE TABLE my_ducklake.dict AS
    SELECT
        row_number() OVER (ORDER BY term) AS termid,
        term,
        COUNT(DISTINCT docid) AS df
    FROM v_token_stream
    GROUP BY term;
```

Listing 4.2: Building dictionary

After all documents have been processed and each term has been added to the dictionary, the table is fully materialized within DuckLake. Note that this full rebuild is only required for initial indexing of the data as updates/modifications to the index do not require full reindexing when using DuckLake. To improve query performance through the classic BM25 algorithm, we pre-compute and store the length of each document in the `docs` table. Listing 4.3 details the SQL used to carry out this procedure. The length of a document refers to the amount of terms a document contains.

```
CREATE OR REPLACE TABLE my_ducklake.docs AS
    SELECT
        docid,
        COUNT(*) AS len
    FROM v_token_stream
    GROUP BY docid;
```

Listing 4.3: Building docs

When identifying which documents are relevant to a query we must be able to identify what document contains a given `termid`. For this we create an inverted index mapping $termid \rightarrow \{docid\}$. This is called a postings table and will be stored as `my_ducklake.postings` where each `(termid,docid)` pair is distinct. Creating the postings table requires a join on the dict table created in Listing 4.2 and the `v_token_stream`, which is the temp view created to store the stream of extracted terms from the content in `my_ducklake.data`. We join on `term`, which we can use to determine the

set of `termid`'s contained in each document. We also compute the term frequency for each `termid` in a `docid`, as this is a statistic used in the BM25 score calculation defined in Equation 3.1. Listing 4.4 outlines the SQL used to carry out this procedure.

```
CREATE OR REPLACE TABLE my_ducklake.postings AS
    SELECT
        d.termid,
        t.docid,
        COUNT(*) AS tf
    FROM v_token_stream t
    JOIN my_ducklake.dict d ON t.term = d.term
    GROUP BY d.termid, t.docid;
```

Listing 4.4: Building Postings

This produces a document-level inverted index: posting lists store term frequency alongside each document identifier, but no word positions. Posting lists are not frequency-sorted, as the disjunctive evaluation strategy used in this project requires exhaustive scoring of all matching documents, making early termination inapplicable.

4.2.2 Managing the Index with DuckLake

We now have an index stored in the format outlined in Figure 4.1, consisting of three tables (`dict`, `docs`, `postings`) residing directly within the `my_ducklake` catalog. Unlike traditional approaches where files are manually exported and imported, DuckLake manages the underlying storage automatically.

DuckLake persists these tables as managed Parquet files in the configured data directory, but abstracts this complexity away. It acts as an interface enabling interaction with the index through standard SQL. Any changes or modifications to the tables through a DuckLake SQL transaction will be tracked and managed by the engine.

When data is modified, DuckLake employs a merge-on-read strategy. Instead of mutating existing Parquet files in place, it writes lightweight delete files that store pointers to invalidated rows.

Updates are similarly handled as a logical deletion followed by an insertion of new records. At query time, DuckLake efficiently merges these delete files with the original data to present a consistent view — allowing us to interact with the tables as single mutable entities while avoiding the high I/O cost of constant file rewrites.

For index maintenance, this means that any client wishing to access the index can attach to DuckLake, and then interact with the index through the DuckLake interface. This ensures that when changes are made, DuckLake is able to provide lakehouse features by keeping track of the metadata as-

sociated to the index files, ensuring that consumers accessing the index will always receive a consistent view of the data.

4.3 Full-Text Search (FTS) Strategy

We implemented the disjunctive BM25 in pure SQL to reduce the amount of transactions to DuckLake, reducing extraneous factors that may have affected query performance, such as latency, to ensure that we got the most accurate results possible. The SQL implementation is provided below.

```
WITH corpus_stats AS (
  SELECT
    COUNT(*)::DOUBLE AS N,
    AVG(d.len)::DOUBLE AS avgdl
  FROM my_ducklake.docs AS d
),

term_hits AS (
  SELECT
    p.docid    AS docid,
    p.termid   AS termid,
    p.tf::DOUBLE AS tf,
    d.len::DOUBLE AS len
  FROM my_ducklake.postings AS p
  JOIN my_ducklake.docs AS d ON d.docid = p.docid
  WHERE p.termid IN (SELECT termid FROM query_terms)
),

idf_table AS (
  SELECT
    dict.termid AS termid,
    ln((stats.N - dict.df + 0.5) / (dict.df + 0.5)) AS idf
  FROM my_ducklake.dict AS dict, corpus_stats AS stats
  WHERE dict.termid IN (SELECT termid FROM query_terms)
),

scored AS (
  SELECT
    th.docid AS docid,
    SUM(
      idf_table.idf *
      ((? + 1) * th.tf) /
      (th.tf + ? * (1 - ? + ? * (th.len / stats.avgdl)))
    ) AS score
  FROM term_hits AS th
  JOIN idf_table ON idf_table.termid = th.termid
  CROSS JOIN corpus_stats AS stats
  GROUP BY th.docid
```

```
)

SELECT docid, score
FROM scored
ORDER BY score DESC
LIMIT ?
```

Listing 4.5: Disjunctive BM25

4.3.1 Ranking models using SQL

Implementing ranking models using SQL has been explored in the past when operating on column stores [1]. DuckLake's interface provides a formal and theoretically sound framework in which we can interact with data, namely using SQL. For rapid prototyping this is particularly advantageous as researchers can explore new scoring functions and features by simply issuing SQL queries, and researchers are forced to be precise about query semantics, which may be especially useful when complex query operators are introduced in document ranking.

Note that the SQL implementation uses the natural logarithm (`ln`) rather than log. As with BM25 generally, the choice of logarithm base affects only score magnitude and not document ranking order [7]. Any alternative base can be obtained by dividing by ln(base).

In our project we implemented Okapi BM25 as an SQL query, but our approach can easily be extended to other ranking functions. Our experiments focused on disjunctive query evaluation where the document must contain at least one query term, but previous work [8] has shown that conjunctive query evaluation yields comparable results to disjunctive query evaluation, yet is faster. We chose to use disjunctive query evaluation to increase the computational load on the DuckLake infrastructure to test the limitations more extensively.

4.4 Dynamic Index Implementation

Dynamic indexing is the idea that we are able to dynamically update an index without having to reindex the whole data set. Many scenarios lead to indexes requiring an update, such as takedown requests, data modification, or data insertion. The common approach to updating an index is to fully reindex the data set, however this can be time consuming and computationally expensive. Taking advantage of the flexibility provided by the lakehouse infrastructure, we demonstrate a fully dynamic index maintenance system incorporating support for ACID properties of transactions with multiple concurrent readers and writers.

4.4.1 Deleting documents from the index

We first demonstrate how using the index schema (Figure 4.1) stored in a lakehouse infrastructure can be used to remove all pertinent information related to a takedown request from the index. In the index schema (Figure 4.1), a single document can be identified by `docid`. The content of a document consists of a series of terms, which are assigned a `termid` when indexed and the relation between `termid`'s and `docid` are stored in the `postings` table. When a `term` is assigned a `termid`, the assignment is stored in the `dict` table along with document frequency which measures how many documents contain a term. When processing a delete we must therefore adjust this document frequency for all terms which are in the deleted document. We can do this by subtracting 1 from the document frequency. Once all of the `termid`'s have been processed and the document frequencies for each have been adjusted we remove all rows where $df = 0$ from the `dict` table. We can then remove all the rows with the deleted documents `docid`, and finally the row in `docs` table with the deleted documents `docid`. Once these changes have been processed the index will no longer contain any data related to the deleted document.

The following is the SQL for deleting a document and its related data from an index stored in DuckLake:

```
BEGIN;

DROP TABLE IF EXISTS touched_termids;

CREATE TEMP TABLE touched_termids(termid BIGINT);

INSERT INTO touched_termids
SELECT DISTINCT termid FROM my_ducklake.postings WHERE docid = ?;

UPDATE my_ducklake.dict
SET df = CASE WHEN df > 0 THEN df - 1 ELSE 0 END
WHERE termid IN (SELECT termid FROM touched_termids);

DELETE FROM my_ducklake.dict
WHERE df = 0
  AND termid IN (SELECT termid FROM touched_termids);

DELETE FROM my_ducklake.postings WHERE docid = ? ;
DELETE FROM my_ducklake.docs    WHERE docid = ? ;
DELETE FROM my_ducklake.data    WHERE docid = ? ;

DROP TABLE IF EXISTS touched_termids;

COMMIT;
```

Listing 4.6: SQL for document deletion

4.4.2 Inserting documents into the index

The insert function can be used following a delete to provide the ability to modify a documents content and update the index, or it can be used to insert an entirely new document. When used for document modification we specify a `docid` in order to keep it consistent, whilst for new documents a new `docid` will be generated.

To insert a document into the pre-existing index we stage the input into a suitable format where we can use SQL to run computations. We use temporary tables for this. After the input is staged in temporary tables we resolve the `docid` and document length to prepare the row which will be inserted into `docs` table which is a prerequisite for generating the rows for the other tables as we need to have the `docid`. We use the same regex method of term extraction as used in the initial indexing to ensure that the insertion is consistent with the other documents in the index. Next we tokenize the terms in the new document and count them for the term frequency statistic which will be used in the `postings` table. Following this we upsert the `dict` table. To do this we do a left join from the temporary `doc_terms` table and the pre-existing `dict` matching on the `term` so we can identify the `termid` of any pre-existing terms in the index. We also temporarily store the row numbers from the partition of terms which are not in the index which we will later use to compute a new unique `termid` for them.

To compute the final `termid` we use the following:

```
1    COALESCE(termid, (SELECT max_id FROM base) + rn)
```

Listing 4.7: Computing unique `termid`

This will use the pre-existing `termid` if the term is already in the dictionary, or generate a new unique `termid` for terms not yet in the index. New `termid`'s are computed by taking the current maximum `termid` from the dictionary and adding each new term's row number to it. Since the row numbers are assigned per-partition over only the unmatched terms, this guarantees that every newly generated `termid` is distinct and does not collide with any existing entry.

Now that the new values required for a merge are staged, we issue a MERGE statement against the DuckLake dictionary. Whenever there is a MATCHED term we increment the document frequency of that row by one, and if there are no matching terms, we insert the new `termid` and `term` into the dictionary with a `df` of 1.

All of the required values and statistics for inserting new rows relating to the document are already staged and ready so we can directly insert into `docs`, `postings` and `data`.

The following is the SQL for the insert function. It has been broken down into the steps, as described above, for convenience.

```
-- 1. Stage input
CREATE TEMP TABLE input_stage(docid BIGINT, content TEXT);
INSERT INTO input_stage VALUES (docid, doc);
----------------------------------

-- 2. Resolve DocID and Length
CREATE TEMP TABLE target_doc AS
SELECT
    COALESCE(i.docid, (SELECT COALESCE(MAX(d.docid), 0) + 1 FROM
        my_ducklake.docs d)) AS docid,
    i.content,
    len(regexp_extract_all(lower(i.content), '[a-z]+')) AS doc_len
FROM input_stage i;

-- 3. Tokenize and count TF
CREATE TEMP TABLE doc_terms AS
WITH raw_tokens AS (
    SELECT UNNEST(regexp_extract_all(lower(content), '[a-z]+')) AS
        term
    FROM input_stage
)
SELECT term, COUNT(*) AS tf
FROM raw_tokens
GROUP BY term;
----------------------------------

-- 4. Upsert dictionary (merging with current index)
WITH base AS (SELECT COALESCE(MAX(termid), 0) AS max_id FROM
    my_ducklake.dict),
annotated AS (
    SELECT
        dt.term,
        d.termid,
        ROW_NUMBER() OVER (PARTITION BY (d.termid IS NULL) ORDER BY
            dt.term) AS rn
    FROM doc_terms dt
    LEFT JOIN my_ducklake.dict d ON dt.term = d.term
),
source AS (
    SELECT
        term,
        COALESCE(termid, (SELECT max_id FROM base) + rn) AS
            final_termid
    FROM annotated
)
MERGE INTO my_ducklake.dict AS tgt
USING source
ON (tgt.term = source.term)
WHEN MATCHED THEN
```

```
45      UPDATE SET df = tgt.df + 1
46  WHEN NOT MATCHED THEN
47      INSERT (termid, term, df) VALUES (source.final_termid, source.
            term, 1);
48  ----------------------------------
49
50  -- 5. Insert Docs
51  INSERT INTO my_ducklake.docs (docid, len)
52  SELECT docid, doc_len FROM target_doc;
53  ----------------------------------
54
55  -- 6. Insert Postings
56  INSERT INTO my_ducklake.postings (termid, docid, tf)
57  SELECT d.termid, td.docid, dt.tf
58  FROM doc_terms dt
59  JOIN target_doc td ON 1=1
60  JOIN my_ducklake.dict d ON dt.term = d.term;
61  ----------------------------------
62
63  -- 7. Insert Content
64  INSERT INTO my_ducklake.data (docid, content)
65  SELECT docid, content FROM target_doc;
```

Listing 4.8: SQL for document insertion

4.4.3 Modifying pre-existing documents

To modify a document from the index we first delete the document, ensuring that there will be no incorrect information stored on the index. We can then insert the modified document back into the index after specifying the `docid`.

4.5 Index Maintenance

The scope of this thesis focuses only on the process of data deletion from an index stored in DuckLake.

When data is deleted from a table that is stored on DuckLake such as `dict`, `docs`, and `postings`, DuckLake will create a delete file containing the row ids of rows that are deleted. Each data file will have its own delete file if any deletes are present for this data file. The meta-data table pertaining to delete files can be seen in Figure 2.3, in the `ducklake_delete_file` table. This table is what keeps track of the location of the delete files and which data files they relate to. When we run a query on DuckLake we first read the data file, then, if one exists, we read the delete file pertaining to the data file and merge them before returning the results. This is called the merge-on-read strategy. Although DuckLake optimises this merge-on-read process [3], when we encounter heavily deleted files this means that we must

read all original rows and all of the deleted rows, which can have an impact on performance when running BM25 queries.

4.6 Storage Optimisation and Compaction

As demonstrated in Chapter 5, heavily deleted indexes exhibit significant performance degradation when no optimisation is carried out. Due to the nature of a web-search index, it is unlikely that significant amounts of data will be deleted at any given time, given that typical takedown requests operate primarily on a single domain or document. As a web-index often contains a large number of documents, it is unlikely that any single takedown request will impact query latency. Although a single takedown request will not have a significant impact on performance, we do expect to see performance degradation over time as numerous takedown requests are processed. For this reason we implement a strategy to maintain the index, ensuring that it consistently performs near its optimal.

As the main contributor to performance degradation is the fragmentation of the data being distributed across data files and delete files, we introduce an index maintenance function which will rewrite the data files, merging with the delete files, and then checkpoint DuckLake to cleanup any expired metadata files, and perform general maintenance.

The SQL for the index maintenance is as follows:

```
1 CALL ducklake_rewrite_data_files('my_ducklake', delete_threshold =>
    0.01);
2 CHECKPOINT;
```

Listing 4.9: Index Maintenance SQL

Here, `delete_threshold => 0.01` instructs DuckLake to rewrite only those data files in which more than 1% of rows have been marked as deleted, avoiding unnecessary rewrites of files that have not accumulated significant delete overhead.

Rewriting the data files by merging them with the delete files, reduces the time complexity of running a BM25 full-text query back to its optimal state. As we no longer have any delete files to read the time complexity for a full-text query can be defined as follows:

$$\mathrm{T}_{\text{total}} = O(\underbrace{\text{Read Data File}}_{\text{Fixed Cost}} + \underbrace{\text{Merge}}_{FixedCost} + \underbrace{\text{Calculate Score}}_{\propto N_{live}}) \tag{4.1}$$

Checkpointing DuckLake after running the data rewrite will flush inlined data, remove expired snapshots, merge adjacent files, cleanup old files, and delete orphaned files. This returns DuckLake to its optimal state ensuring there is no unnecessary data or files being stored.

Chapter 5

Evaluation and Results

5.1 BM25 full-text search complexity analysis

When running a BM25 full-text query on a dynamic index, the time complexity of data retrieval using DuckLake's merge-on-read strategy is comprised of two primary costs, namely, the I/O cost where the engine must find every data file and delete file associated with the table segment currently being processed and read these files from disk into memory. Then there is the CPU cost where the engine starts reading the main data file and for every single row it will check if the row ID is in the delete file, and discards it immediately if it is. The second cost which contributes to the overall query time is the CPU cost of calculating the BM25 score. This is proportional to the number of rows we are running the BM25 scoring function on.

The total time complexity can be defined as follows:

$$\mathrm{T}_{\text{total}} = O(\underbrace{\text{Read Data File}}_{\text{Fixed Cost}} + \underbrace{\text{Read Delete File}}_{\propto N_{deleted}} + \underbrace{\text{Merge}}_{\text{Fixed Cost}} + \underbrace{\text{Calculate Score}}_{\propto N_{live}}) \tag{5.1}$$

where

`Read Data File` is the fixed cost of reading the entire original data file. This cost never changes, unless the data files are rewritten.

`Read Delete File` is the cost of gathering and reading all the delete files. This cost is directly proportional to the number of deleted rows.

`Merge` is the CPU cost of merging the original data file with the delete files on-read. The cost is constant and depends on the total number of rows in the original data files, as it must check every row against the delete files during the merge-on-read.

`Calculate Score` is the computational cost of running the BM25 scoring function. This cost is proportional to the number of live rows (rows that have not been deleted).

The I/O read cost of the BM25 scoring algorithm (Listing 4.5), which increases with the number of postings and delete records, is the following:

```
-- line 5
-- cost: read docs + read docs delete
FROM my_ducklake.docs AS d

-- line 14
-- cost: read postings + postings delete + read docs + read docs
    delete
-- This is the largest contributor to the I/O cost
FROM my_ducklake.postings AS p
JOIN my_ducklake.docs AS d ON d.docid = p.docid

-- line 23
-- cost: read dict + read dict delete
FROM my_ducklake.dict AS dict, corpus_stats AS stats
```

Listing 5.1: I/O Read Cost from listing 4.5

The scoring cost of the BM25 scoring algorithm (Listing 4.5), which decreases as the proportion of deleted documents increases, is the following:

```
-- line 27 to 39
scored AS (
  SELECT
    ...
    SUM(
      idf_table.idf *
      ((? + 1) * th.tf) /
      (th.tf + ? * (1 - ? + ? * (th.len / stats.avgdl)))
    ) AS score
  FROM term_hits AS th
  ...
)
```

Listing 5.2: Scoring Cost from listing 4.5

As this contains divisions, multiplications and `SUM`'s it is CPU-intensive. However, this logic only runs on the surviving rows after the merge-on-read, meaning the scoring cost decreases as the proportion of deleted documents increases.

Let $x \in [0, 100]$ represent the deletion percentage, defined as the proportion of deleted entries relative to the total index size. Let $T_{total}(x) = T_r(x) + T_s(x)$ be the total query latency, where $T_r(x)$ is the **read latency** (strictly increasing; $\Delta T_r > 0$) and $T_s(x)$ is the **scoring latency** (strictly decreasing; $\Delta T_s < 0$).

5.2 Empirical evaluation (no optimisation)

To test the significance of the performance degradation as the percentage of the index being deleted increases, we run performance testing on 100 queries and take the average query time for each. We then delete 10000 rows and then repeat the same 100 queries whilst storing the percentage of the original index that has been deleted as well as the average query time.

Dataset

The experiments use the OWS Curlie 2025 test collection[1] developed by the Open Web Search project [11, 12]. The full collection contains approximately 20 million documents; for this work, the first 457,685 documents were used, distributed across 11 Parquet files (approximately 1.4 GB on disk). Each document includes full-text content, structured metadata (title, description, URL components, and Curlie topic labels), and WARC crawl metadata, with documents crawled between July 2024 and July 2025. The BM25 index is built over the `main_content` field of these documents.

Hardware Specifications

All performance testing was conducted on a MacBook Pro with an M2 Pro chip, 16 GB of RAM, running macOS Tahoe 26.2.

The limitations of using a personal laptop for the testing limited the amount of queries that we could run in a reasonable amount of time. A delete batch of 10000 was also required to allow the algorithm to be run in under an hour. We tested the impact of varying the size of the delete batch and found no significant variations and so a delete batch of 10000 was found to be an optimal balance of time and granularity.

Test parameters

We implemented performance testing functionality into `dynamic_index.py` [2], accessible via the `perf-test` sub-parser (`python dynamic_index.py perf-test ...`). Each run evaluated 100 disjunctive queries, returning the top 10 results, with a delete batch size of 10,000 rows and no maintenance interval (checkpoint disabled). The index was reset to a clean state before each run.

Results

The results are shown in Figure 5.1.

[1] `https://dashboard.ows.eu/corpora/b510baa6-ebd2-11f0-8c43-02a47ca5d9fd`

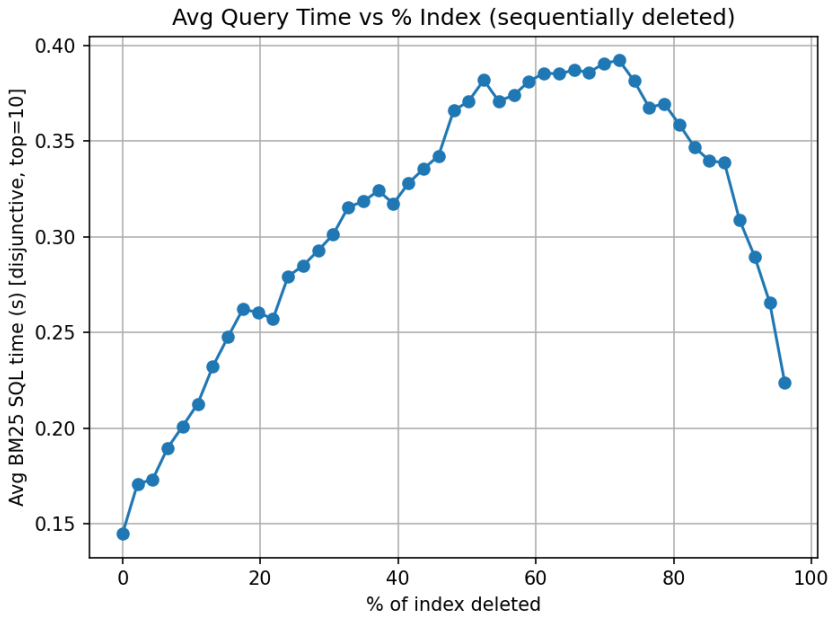

Figure 5.1: BM25 Query performance on heavily deleted index

Discussion

The results demonstrate that query time displays a linear increase relative to the deletion percentage in the interval $0\% \rightarrow 60\%$. The trend reaches an inflection point at approximately 65% (the peak), beyond which the query time shifts to a linear decrease. This behaviour suggests there is a trade-off between the overhead of processing delete files in the merge-on-read and the computational savings of scoring fewer documents.

Phase 1 ($0 \rightarrow 60\%$): Read-Dominated Growth. The marginal cost of reading additional delete records exceeds the marginal savings gained from scoring fewer rows:

$$\Delta T_r(x) > |\Delta T_s(x)|$$

Consequently, $\Delta T_{total}(x) > 0$, resulting in a net increase in query latency.

Phase 2 ($60 \rightarrow 70\%$): Equilibrium. The system reaches a critical point where the rate of I/O overhead growth is exactly offset by the rate of computational savings:

$$\Delta T_r(x) \approx |\Delta T_s(x)|$$

At this stage, $\Delta T_{total}(x) \approx 0$, observed as the local maximum or plateau of the latency curve.

Phase 3 ($70 \rightarrow 100\%$): Scoring-Dominated Decay. The computational savings from skipping the expensive scoring step finally outweigh the linear cost of reading the delete files:

$$\Delta T_r(x) < |\Delta T_s(x)|$$

Here, $\Delta T_{total}(x) < 0$, causing a rapid decline in total query time.

5.3 Empirical evaluation (storage optimisation and compaction)

To test the impact of rewriting the data files and checkpointing, we repeated the same test conducted in Section 5.2, however this time we ran index maintenance (Listing 4.9) at set delete percentage intervals. The maintenance interval was varied across runs from 10% to 50% in increments of 10%, with hardware specifications and all other parameters kept identical to those in Section 5.2.

Test parameters

All six runs share the same base configuration: 100 disjunctive queries, top 10 results, a delete batch of 10,000 rows, and a full index reset before each run to ensure a clean, controlled starting state. Runs 2–6 reuse the same set of 100 queries generated in Run 1, allowing a direct comparison across maintenance intervals. The only parameter varied across runs is the maintenance interval (checkpoint percentage), as shown in Table 5.1.

Run	Checkpoint %	Re-use queries
Run 1	None (no maintenance)	—
Run 2	10%	Yes
Run 3	20%	Yes
Run 4	30%	Yes
Run 5	40%	Yes
Run 6	50%	Yes

Table 5.1: Varied parameters across the six evaluation runs.

Results (Storage optimisation and compaction)

The combined results across all six runs are shown in Figure 5.2.

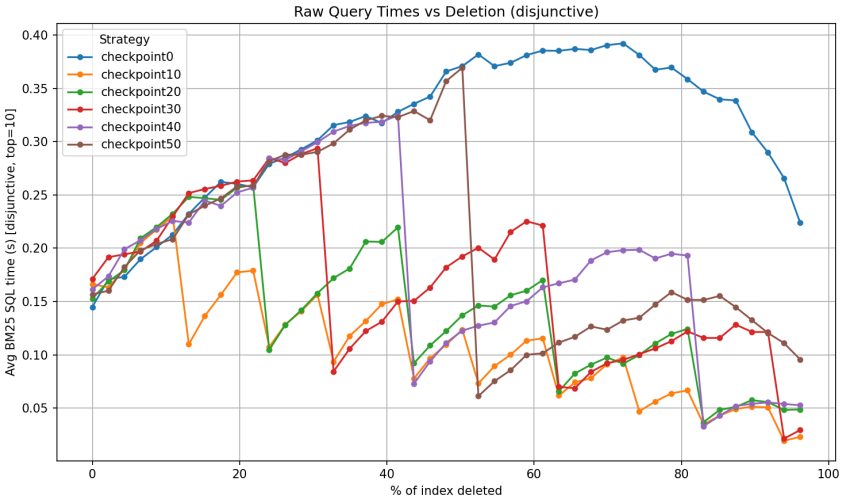

Figure 5.2: BM25 Query performance with data file rewrites and checkpointing every N %

Discussion

The combined results shown in Figure 5.2 demonstrate that implementing periodic storage optimisation transforms the query performance profile from a single continuous curve into a "sawtooth" pattern. Each maintenance strategy initially follows the same linear increase observed in the unoptimised test (Run 1), but once the maintenance interval is reached the optimised tests diverge rapidly, experiencing a significant drop below the previous baseline before beginning a new, shallower ascent.

Let x_c represent the checkpoint interval (e.g., 10%), with $T_{total}(x)$ governed by the periodic application of the compaction procedure `ducklake_rewrite_data_files`.

Phase 1 (Inter-Checkpoint): Accumulation. Between maintenance intervals ($x \in [kx_c, (k+1)x_c)$), the system behaves identically to the unoptimised baseline. The marginal cost of reading accumulating delete flags exceeds the savings from scoring fewer rows:

$$\Delta T_r(x) > |\Delta T_s(x)|$$

Consequently, $\Delta T_{total}(x) > 0$, causing the linear "sawtooth" rise in latency.

Phase 2 (At Checkpoint): Compaction Reset. At $x = kx_c$, the compaction process triggers a discrete event where the data files are rewritten. This physically removes deleted rows, eliminating the read overhead (T_r) associated with dead space, and results in an instantaneous vertical drop in latency, effectively resetting the accumulated "delete tax".

Phase 3 (Global Trend): Baseline Decay. Unlike the unoptimised curve, the local minima of the optimised runs follow a strictly decreasing trend. The post-compaction latency approaches a theoretical minimum defined by the volume of remaining data: at 80% deletion, the system is scanning a file physically 80% smaller than the original baseline.

As web-search indexes typically operate on large volumes of data, the deleted percentage of the index will rarely increase rapidly, meaning there is ample opportunity to perform periodic index maintenance before significant performance degradation is observed.

The cumulative query cost over the index lifetime is shown in Figure 5.3. Without maintenance, the total query time accumulates to approximately 2350 s by the time 95% of the index has been deleted, compared to only 500 s for the 10% interval strategy — a reduction of nearly 80%. This demonstrates that whilst the per-query degradation may appear modest in isolation, the aggregate query cost over the full index lifetime is significant.

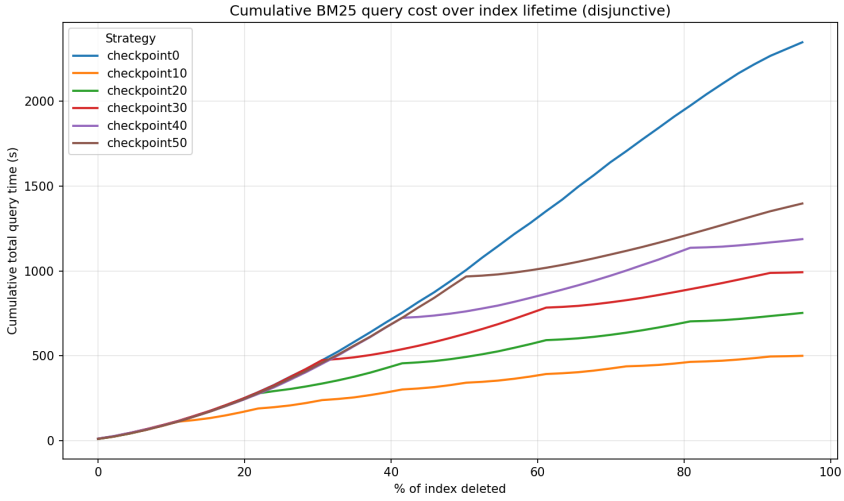

Figure 5.3: Cumulative BM25 query cost over the index lifetime for each maintenance strategy

Chapter 6

Conclusions

In this thesis, we have argued analytically and demonstrated empirically that SQL-based lakehouse infrastructure can support practical full-text ranking and dynamic index maintenance on web-crawl data without reindexing. Furthermore, whilst the results of testing BM25 query performance highlight significant performance degradation as the percentage of the index that is marked as deleted increases, we demonstrate how implementing simple periodic storage optimisation transforms the query performance profile, keeping the query performance closer to its optimum.

6.1 Summary of Contributions

We have designed and built an index schema (`dict`, `docs`, `postings`) stored in DuckLake, with BM25 executed entirely in SQL against this schema. We have implemented a fully dynamic index supporting insert, delete, and modify operations without requiring full reindexing, leveraging DuckLake's merge-on-read strategy to apply changes without rewriting data files. Our empirical evaluation demonstrated a predictable, analytically explainable performance profile under sustained deletions, and showed that periodic compaction successfully resets accumulated delete overhead, producing a sawtooth performance curve with a declining floor.

6.2 Limitations

There are several limitations to the methodology used in this thesis. The testing was conducted on a single MacBook Pro M2, meaning our conclusions are about relative trends rather than absolute performance. Furthermore, testing was conducted on a single-node setup; distributed object storage with concurrent readers and writers was not evaluated. For indexing, we utilised a simple regex parser for term extraction; while sufficient for our setup, optimising or measuring the precise accuracy of this extraction was

out of scope, which would matter if retrieval effectiveness were being strictly evaluated. During our performance evaluation, bag-of-words queries were used to ensure reproducibility, but this may not accurately reflect real query distributions. Finally, the maintenance evaluation only benchmarked data deletion, meaning insertion and modification were not empirically tested.

6.3 Future Work

Further experimentation could be carried out for data insertion and modification. This would complete the full dynamic index lifecycle evaluation. This project aimed to demonstrate the feasibility of using a database for ranking in a lakehouse infrastructure, and so testing was carried out on a range of maintenance thresholds, over a full incremental index deletion. This is not a realistic scenario that would occur typically and instead aimed to demonstrate the performance implications of this infrastructure across all cases. This means that future work could be carried out to find a more efficient way to maintain an index in a lakehouse, such as finding more optimal maintenance thresholds to trigger the compaction events. This could involve implementing adaptive compaction thresholds based on a dynamic cost-benefit analysis rather than fixed intervals. Finally, future research could compare the retrieval effectiveness of the SQL-based BM25 implementation on DuckLake against a reference implementation on standard benchmarks, such as TREC or MS MARCO [13].

Bibliography

[1] H. Mühleisen, T. Samar, J. Lin, and A. De Vries, "Old dogs are great at new tricks: Column stores for ir prototyping," in *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*. Gold Coast Queensland Australia: ACM, Jul. 2014, pp. 863–866.

[2] S. Struthers, "Dynamic index DuckLake," https://github.com/s4mstruthers/dynamic-index-ducklake, 2025, accessed: 2025.

[3] M. Raasveldt and H. Mühleisen, "The DuckLake Manifesto: SQL as a Lakehouse Format," DuckDB Labs, Tech. Rep., 2025.

[4] Apache Software Foundation, "Apache Parquet format specification," https://github.com/apache/parquet-format, 2013, accessed: 2025.

[5] ——, "Apache Iceberg table format specification," https://iceberg.apache.org/spec/, accessed: 2025.

[6] S. Robertson, S. Walker, S. Jones, M. Hanco c kBeaulieu, and M. Gatford, "Okapi at TREC-3," in *Proceedings of The Third Text REtrieval Conference (TREC-3)*, ser. NIST Special Publication, vol. 500–225. National Institute of Standards and Technology, 1994, pp. 109–126.

[7] S. Robertson and H. Zaragoza, "The Probabilistic Relevance Framework: BM25 and Beyond," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009.

[8] N. Asadi and J. Lin, "Effectiveness/efficiency tradeoffs for candidate generation in multi-stage retrieval architectures," in *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*. Dublin Ireland: ACM, Jul. 2013, pp. 997–1000.

[9] J. Zobel and A. Moffat, "Inverted files for text search engines," *ACM Computing Surveys*, vol. 38, no. 2, 2006.

[10] S. Büttcher and C. L. A. Clarke, "Hybrid index maintenance for contiguous inverted lists," *Information Retrieval*, 2008.

[11] M. Granitzer, S. Voigt, N. A. Fathima, M. Golasowski, C. Guetl, T. Hecking, G. Hendriksen, D. Hiemstra, J. Martinovič, J. Mitrović, I. Mlakar, S. Moiras, A. Nussbaumer, P. Öster, M. Potthast, M. Senčar Srdič, S. Megi, K. Slaninová, B. Stein, A. P. de Vries, V. Vondrák, A. Wagner, and S. Zerhoudi, "Impact and development of an Open Web Index for open web search," *Journal of the Association for Information Science and Technology*, Aug. 2023. [Online]. Available: https://asistdl.onlinelibrary.wiley.com/doi/full/10.1002/asi.24818

[12] G. Hendriksen, M. Dinzinger, S. M. Farzana, N. A. Fathima, M. Fröbe, S. Schmidt, S. Zerhoudi, M. Granitzer, M. Hagen, D. Hiemstra, M. Potthast, and B. Stein, "The Open Web Index: Crawling and Indexing the Web for Public Use," in *Advances in Information Retrieval (ECIR 2024)*, ser. Lecture Notes in Computer Science. Springer, Mar. 2024, pp. 130–143. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-031-56069-9_10

[13] T. Nguyen, M. Rosenberg, X. Song, J. Gao, S. Tiwary, R. Majumder, and L. Deng, "MS MARCO: A Human Generated MAchine Reading COmprehension Dataset," in *Advances in Neural Information Processing Systems Workshop on Cognitive Computation*, 2016.