The Australian National University
2600 ACT | Canberra | Australia

Australian National University

School of Computing

College of Engineering, Computing and Cybernetics (CECC)

Hybrid Memory-Aware Query Caching for Real-Time Search

— 12 pt research project (S1/S2 2023)

A report submitted for the course
COMP3770, Project Work in Computing

By:
Chethin Weerakkody

Supervisor:
Dr. Shoaib Akram

November 2026

Declaration:

I declare that this work:

- upholds the principles of academic integrity, as defined in the University Academic Misconduct Rules;
- is original, except where collaboration (for example group work) has been authorised in writing by the course convener in the class summary and/or Wattle site;
- is produced for the purposes of this assessment task and has not been submitted for assessment in any other context, except where authorised in writing by the course convener;
- gives appropriate acknowledgement of the ideas, scholarship and intellectual property of others insofar as these have been used;
- in no part involves copying, cheating, collusion, fabrication, plagiarism or recycling.

November, Chethin Weerakkody

Abstract

The demand for real-time search and analytics over short, textual data from microblogging websites have surged, driven by the growth of social media datasets. Real-time indices stress the DRAM capacity of a single server, where even the fastest SSDs cannot satisfy response time constraints demanded by real-time search. Intel Optane DC Persistant Memory (PMEM) is a promising new memory technology that demonstrates similar performance characteristics to traditional DRAM. To mitigate DRAM pressure, one solution is to augment the DRAM capacity with a denser storage technology, serving as a second tier. Although hybrid DRAM/PMEM systems deliver promising performance, there are some performance trade-offs compared to a DRAM exclusive system. This study explores query caching for real-time search, building upon SPIRIT, a real-time engine for text inversion and search. SPIRIT utilizes a dynamically growing inverted index over two-tiered DRAM/PMEM memory. In this work we deliver a real-time cache that bridges the performance gap between hybrid memory and DRAM exclusive systems. We explore the possibility of caching results in PMEM directly to avoid extra DRAM strain. We find different parameters that influence caching performance based on memory hardware. This research contributes to enhancing the efficiency of hybrid memory-based real-time caching.

Table of Contents

1 Introduction **1**

2 Background and Related Work **3**
2.1 Multi-Tiered Memory and Intel Optane DC PMEM . . . 3
2.2 Real-Time Search . . . 3
2.3 Real-Time Search with SPIRIT . . . 4
2.4 Related Work - Caching for Real-Time Search . . . 5

3 Design and Implementation **6**
3.1 Principles . . . 6
3.2 Overview . . . 7
3.3 Core Data Structures . . . 7
3.4 Memory Management . . . 9
3.5 Real-time Caching Policy . . . 9
3.6 Controlling Fragmentation . . . 10
3.7 Query Evaluators and RT-Cache . . . 11
3.8 Eviction . . . 12

4 Evaluation **13**
4.1 Hardware . . . 13
4.2 Datasets and Workloads . . . 13
4.3 Default Parameters and Measurement Metrics . . . 14
4.4 Results . . . 14
4.4.1 Repetition Probability . . . 14
4.4.2 Cache Capacity and Partitioning . . . 16
4.4.3 Index Size . . . 18
4.4.4 Multiple Executors and Workload Complexity . . . 19
4.4.5 RT-Cache and DRAM only index . . . 20

5 Concluding Remarks **21**
5.1 Conclusion . . . 21
5.2 Future Work . . . 21

Bibliography **23**

Chapter 1

Introduction

It is crucial for numerous enterprises, including LinkedIn, Meta, and X (Twitter), to enable fast real-time search over social media content (Auradkar et al., 2012; Lin et al., 2012). Social media content is actively queried for relevance, similar to static web content, or implicitly by the platform to populate a user's home feed. These queries run concurrently with an indexing engine that builds indices in real-time, coping with a massive volume of data, e.g., 500 million Tweets per day for Twitter (Twitter, 2023). As a result real-time indexing significantly stresses the CPU and memory hardware of the system (Lin et al., 2012; Akram, 2021).

To offer the speed required for real-time operations, it is required to place the entire index on Dynamic Random Access Memory (DRAM) (Zhuang et al., 2014; Gupta et al., 2023; Xie et al., 2018). Even the fastest Solid-State Drives (SSDs) cannot keep up with the response time requirements we want for real-time search. The growth of social media content has led to an expansion in index sizes, placing a strain on DRAM capacity. However, DRAM capacity doesn't scale as fast as the demand (Mutlu and Subramanian, 2014; Handy, 2018; Gupta et al., 2023; Berg et al., 2020; McAllister et al., 2021; Chen et al., 2020; Abulila et al., 2019; Huang et al., 2015; Weiner et al., 2022). As a result, significant financial investments have been allocated to either increasing DRAM capacity in servers or placing portions of the index in SSDs which tend to give significantly worse performance.

Previous work introduces SPIRIT, a real-time engine for text inversion and search (Akram and Atkinson, 2022). SPIRIT addresses the strain on DRAM by augmenting DRAM capacity with a more promising denser storage technology — Intel Optane DC Persistant Memory (PMEM). Optane is a high-performance, non-volatile memory solution designed to deliver enhanced data storage and processing capabilities. While comparable to DRAM in performance, Optane introduces unique trade-offs (Akram, 2021). Although SPIRIT delivers high performance ingestion and fast query evaluation

demonstrating that PMEM serves as a viable complement to DRAM, hybrid memory indices give slightly worse performance compared to an all DRAM index. Our work aims to bridge this performance gap and enhance real-time search performance by enabling query caching in PMEM.

Caching of query results is a well-established approach to speeding up query evaluation of long-running popular queries. However, real-time search encounters specific hurdles in query caching: (1) frequent updates to the index due to fresh data ingestion often result in inconsistent caches, and (2) variable-sized cache lines (i.e., query results) contribute to significant fragmentation upon eviction (Bortnikov et al., 2011). In this study, we look at query caching for real-time text inversion across multi-tiered memory. Our focus is on establishing a cache layered on top of the SPIRIT platform. In addition to fundamental issues relating to caches (organization, replacement, etc), we explore the design space of query caching for SPIRIT and tackle novel questions: (1) can we expect an overall speedup by caching in slow tier memory, (2) how do the distinctive trade-offs of PMEM impact the decision between using PMEM or DRAM for caching, (3) determining the optimal moments to utilize the cache versus direct computation over the index, and (4) strategies for handling index updates.

Chapter 2

Background and Related Work

2.1 Multi-Tiered Memory and Intel Optane DC PMEM

Hybrid (multi-tiered) memory is the combination of fast DRAM and a high-capacity yet slower secondary byte-addressable memory. For our purposes, we consider Persistent Memory (PMEM) as the second tier, with Intel Optane DC PMEM serving as a real-world PMEM prototype.

Intel Optane DC PMEM leverages 3D XPoint technology, a phase-change random access memory technology (Lee et al., 2010). Similar to DRAM, Optane connects to the main memory bus as a non-volatile DIMM (NVDIMM). Optane supports upto 512 GB of storage per DIMM, this is twice the maximum density of current DRAM DIMMs (Intel, 2019). Optane promises latency levels 10 times lower than NVMe SSDs, although they have 2–3 times higher latency than regular DRAM (Yang et al., 2020).

While Optane presents promising features, including high bandwidth and low read/write latencies, integrating it as the secondary tier memory in a hybrid system results in overall slower performance compared to an exclusive DRAM system (Akram and Atkinson, 2022). This is due to the distinct trade-offs associated with Optane. The bandwidth of persistent memory may become a bottleneck, especially under write-intensive workloads. Additionally, Optane demonstrates even poorer performance in random accesses (Yang et al., 2020), and unlike DRAM, it is prone to wear and tear over time due to extensive write volumes.

2.2 Real-Time Search

Real-time search engines are engines that can provide immediate results as soon as new data becomes available. This is different to traditional search engines (batch indexed) that have a delay between indexing and the ability to query new information (Wilson,

2022). A real-time search engine must handle a high rate of document ingestion, so low indexing latency is crucial (Tonozzi and Daniliuc, 2020; Lin et al., 2012). Updates to the index should be readily visible to query evaluators almost immediately after ingestion and queries should be resolved swiftly. Therefore, these engines experience significant number of concurrent reads and writes to the index. Consequently, creating demands on the ingestion portion of a search engine.

With these reasons, real-time indices are commonly placed exclusively in DRAM (Zhuang et al., 2014; Gupta et al., 2023; Xie et al., 2018). However, we see current state of the art using a hybrid memory setup for real-time indices (Akram and Atkinson, 2022). Documents are initially ingested into DRAM, which ensures immediate visibility. Later, a gradual transfer of documents into PMEM is carried out to remove the strain on DRAM. We focus our work on SPIRIT, a search engine built on this principle (Akram and Atkinson, 2022).

2.3 Real-Time Search with SPIRIT

SPIRIT is an in-house engine for **S**calable, **P**ersistent, **I**n-memory, **R**eal-time, **T**ext **I**nversion over large multicore servers. The core principle of SPIRIT involves maintaining a segmented index over a pre-allocated hybrid DRAM/PMEM heap. The implementation of SPIRIT has two main components: (1) an ingest pipeline and (2) a pool of query evaluators.

The ingest pipeline ingests documents into segments. These segments provide necessary data structures for storing postings related to different words in documents. Initial ingestion takes place in DRAM because creating new segments in PMEM results in indexing latency (Akram, 2021). Once a segment reaches a threshold size, it becomes immutable and undergoes a transformation from a write-optimized segment (WOS) format to a read-optimized (ROS) format. To allow continuous ingestion a new WOS is created in DRAM. The immutable ROS resident in DRAM is synchronously (and directly) copied into PMEM. SPIRIT retains segments on the DRAM heap until it exhausts DRAM capacity. When SPIRIT eventually removes segments from DRAM, the engraved segments in PMEM seamlessly become visible to query evaluators. This allows low-latency query evaluation for the most recent results, which are often the most critical (Busch et al., 2012). The fundamental concept of SPIRIT's index aligns with the log-structured merge (LSM) tree O'Neil et al. (1996).

Operating independently from the indexing pipeline, a pool of query evaluator threads are dedicated to resolving queries. SPIRIT 's query evaluators support AND queries of one or more terms. The SPIRIT queries return results in binary format to reduce IO bottleneck (Akram and Atkinson, 2022). The query evaluators have complete visibility over the entire on-heap index. The query evaluators evaluates results segment at a time, accessing segments from most to least recent. This allows most recent result to appear first in the query response. A lightweight protocol switches the implementation of the

query evaluation algorithm for each evaluator based on the segment placement (DRAM WOS or PMEM ROS) and query type (multi-term or single term). Other details are found in an unpublished paper Akram and Atkinson (2022).

2.4 Related Work - Caching for Real-Time Search

Caching in Real-Time search faces a key challenge: concurrent index updates may render cached results outdated due to continuous changes in the underlying data. Ensuring up-to-date query results is critical when using a Real-Time cache. A solution for fixing an inconsistent cache is to remove outdated entries. Identifying the right moment to invalidate cached data poses a challenge. If done too early, there's a risk of inefficiently fetching data too frequently and wasting resources. On the other hand, delaying the invalidation poses the risk of providing users with outdated data, leading to inaccurate results and sub-optimal performance (Yldiz, 2023).

Bai and Junqueira (2012) propose integrating a Cache Invalidation Predictor (CIP) within the Real-Time cache. The CIP functions as a heuristic-based predictor, identifying outdated entries. These outdated entries could either be evicted (Bai and Junqueira, 2012) or undergo lazy updating to align with the most recent updates in the index. Bortnikov et al. (2011) proposes a CIP that runs in parallel to the Real-Time cache, marking entries as stale. In contrast, cache invalidation could be event based, where cache reads/writes trigger the CIP marking entries as invalid. Cache invalidation uses related metadata as heuristics to determine staleness of entries. A common heuristic used in CIPs are time-to-live or expiration values (Sazoglu et al., 2013).

In contrast to invalidating entries, an alternative approach involves preserving sufficient metadata that allows the creation of a consistent result. This ensures that when there's a cache hit on stale data, the cached information isn't completely lost. Implementing this strategy requires having a mechanism capable of clearly indicating what is cached and what is new information from the updated index. This allows query evaluators to retrieve new information while retaining the old cached data. Approaches like the latter are more commonly used when there is no need to remove existing information from a query result. Currently the engine we focus on SPIRIT does not support deletion, so this method is a suitable candidate for a Real-Time caching policy.

Outside of the fundamental challenge of cache consistency with the index, other Real-Time caching challenges include high likely hood of fragmentation. When a query is initially cached, there may be a substantial amount of data to store. However, in subsequent calls to the same query, the addition of new data may be minimal, resulting in variable-sized cache entries in memory. This variability can give rise to fragmentation, which has the potential to impact read performance adversely. Managing this fragmentation becomes essential for maintaining the efficiency of the Real-Time cache.

No research exists related to real-time caching in PMEM. Our work contributes RT-Cache, and an evaluation of parameters that affect real-time caching in PMEM.

Chapter 3

Design and Implementation

In this section, we describe the design and implementation of our Real-Time Cache (RT-Cache).

3.1 Principles

The design of RT-Cache must follow these principles:

1. **Seamless interface with SPIRIT's query evaluators** (P1): The implementation should integrate with SPIRIT's existing query evaluators to achieve a notable speedup across various workloads. The design of the cache should not change based on changes in SPIRIT.
2. **Maintain Real-Time search with Cache** (P2): Indexed data should be readily searchable. The cache should not slow down querying in any edge case. To prevent likely cache misses, the system should prioritize reading the newest segments directly from the index rather than relying solely on the cache.
3. **Keep Cache consistent with index** (P3): The utilization of the cache should yield consistent results as if the cache were not employed, with the exception of permissible deviations in the newest ingested documents
4. **Use a pre-allocated heap** (P4): RT-Cache should operate on a preallocated, on-heap DRAM or PMEM memory. Purposely avoid external memory allocators and block storage accesses.
5. **Easily adaptable to different memory technologies** (P5): The design should exhibit the flexibility to switch between Persistent Memory (PMEM) and Dynamic Random Access Memory (DRAM) as the underlying hardware.

3.2 Overview

SPIRIT initialises RT-Cache on startup by allocating a heap on the required memory hardware. The query evaluators of SPIRIT adhere to a cache-first policy, initiating a cache lookup when evaluating a query. In the event of a cache miss, the system reverts to a default SPIRIT lookup. On a cache hit, all stored results are retrieved from the cache. RT-Cache contains metadata which guides the query evaluator to identify results not present in the cache, prompting a subsequent default SPIRIT lookup to obtain the latest results from newly ingested documents.

SPIRIT has already optimized single-term query evaluation extensively, however, it encounters challenges in resolving intersections for multi-term queries (Akram and Atkinson, 2022). In an effort to enhance SPIRIT's performance, we decide to exclusively store multi-term query results in the cache. Regardless of a cache hit or miss, all multiterm query results are added to the cache. It's probable that a cache hit won't yield the most current results. In such instances, any new segments containing relevant results are also added into the cache. LRU is employed to evict queries if the cache reaches capacity when adding new entries.

3.3 Core Data Structures

We use a number of key data structures to construct RT-Cache. The backbone of the cache is a fixed-size hash table consisting of `CacheBucket` structs. These cache buckets are responsible for storing metadata related to the queries and results being cached. Each cache bucket points to a linked list of `CacheBlock` struct, here, results from queries are stored. Figure 3.1 shows the overall structure of the cache.

Cache buckets contain metadata related to the cached query and stored results. We store the total size of the stored results, which is used to retrieve the results from the cache. The buckets store the query in term form and the number of terms in the query, providing the means to identify the appropriate cache bucket incase a collision in the hash table. The cache bucket stores the last seen segment ID, which informs query evaluators about the extent to which the index has been cached for a specific query, facilitating the retrieval of consistent results if the cache is employed. To manage eviction, cache buckets retain a sense of recency within the cache. Each bucket has pointers to the next and previous buckets, establishing an order based on recency where the first bucket in the list is considered the most recent, and the last is regarded as the least recent.

Cache blocks store the result of intersection of corresponding terms in the query. They also store information about the size of the stored result specific to the block, which allows retrieval. SPIRIT has a segmented index, ensuring that a document is exclusively associated with a single segment (Akram and Atkinson, 2022). We uphold this principle in the design of our cache blocks, guaranteeing that results for a given query from a specific document are held within a single cache block. This characteristic is a fundamental aspect of our caching policy, which is discussed in detail in Section

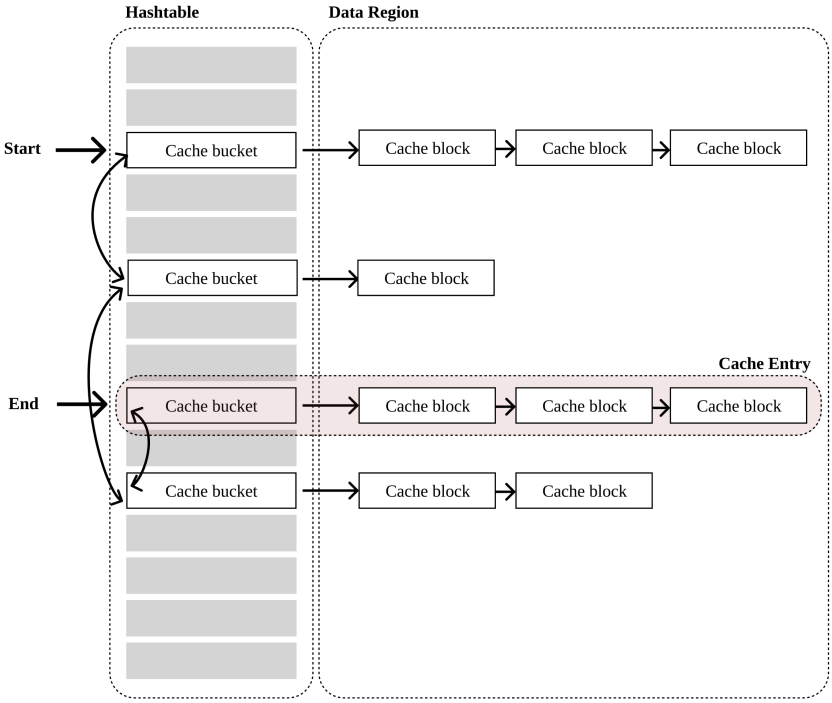


Figure 3.1: An illustration of the structure of the cache showing the fundamental data structures employed in creating the cache.

3.5. Real-time caching allows already cached queries to be re-cached with newer results. This feature is facilitated by the linked list of cache blocks. If a query is cached more than once, the new results are stored in a new cache block and appended to the linked list.

The allocated space for the cache is divided between the **hash table** and the **data blocks**. The dbj2 hash algorithm is applied to hash each term in a query, and the sum of these hashes is used as an index into the hash table. The hash table uses an open addressing scheme to avoid collisions. Since the cache requires eviction, we need to implement deletion in our open addressing scheme. To address this, each cache bucket maintains a status tag, serving as a notification for the lookup algorithms about its deletion.

To prevent long probing chains of deleted buckets, we have implemented an optimization. In this approach, buckets following extended chains of deleted buckets are relocated to the first deleted bucket. This optimization aims to eliminate the overhead for the subsequent retrieval process.

3.4 Memory Management

In SPIRIT all memory is pre-allocated. Pre-allocation is purposely avoiding any external allocator calls during operation, which is critical in controlling tail latency. To maintain consistency and control over tail latencies, the heap in RT-Cache will also be pre-allocated, hence the principle **P4**. According to the research focus we need to be able to evaluate the appropriateness of caching in Persistent Memory. To achieve this, there is a need to adapt the heap to reside on PMEM, hence the principle **P5**. We utilize an anonymous `mmap` for DRAM heap allocation and file-backed `mmap` for PMEM heap allocation.

To fulfill the principle **P4** and **P5** we have opted to use a custom memory allocator. To avoid impacting SPIRIT's performance by altering its heap directly, we have designated a separate heap for RT-Cache through another mapping. Our choice is a custom built Two-Level Segregated Fit (TLSF) Fixed Heap allocator (Kilrain, 2022). TLSF guarantees constant-time allocation and deallocation, which makes it suitable for real-time systems. We have modified the memory allocator to allow the heap location to be either in PMEM or DRAM, before executing SPIRIT.

As discussed above, we need to partition the heap between the hash table and data stored. Each cache bucket is approx 350 bytes, allowing ∼3000 cache buckets in a 1 MB hash table. The most optimal partition between hash table and data will be determined later in section 4.

3.5 Real-time Caching Policy

RT-Cache must align with the real-time characteristics of SPIRIT. SPIRIT maintains a segmented index, placing newly added documents into the newest segment. When a segment reaches its capacity it is committed and a new segment is created. RT-Cache caches queries as it encounters them. However, if the same query resurfaces after some time, the cached results may no longer align with the latest results due to continuous document ingestion.

To satisfy **P3** principle, guaranteeing consistency with the latest information considering the ongoing ingestion of documents, we need to implement a policy in RT-Cache. We must store metadata in the cache bucket related to query such that the query evaluator can clearly discern the contents of the cache and distinguish between what is present and what is not. Once the query evaluator identifies what is not in the cache, it needs to perform a lookup over the segments that have not been cached and append the extra results accordingly. However, a challenge arises with the newest segment, as caching values from it before it is committed can lead to problems. In subsequent encounters with the query, that segment might have changed before it was committed. To address this, we consider two potential solutions.

One approach is to store information such as the last doc ID, to pinpoint precisely

where in the ingestion process we cached up to. This allows the query evaluator to do a default SPIRIT lookup and only append results that follow the specific document ID. Alternatively, we may choose not to cache the latest segment at all and only store the last-seen segment ID as metadata. This decision ensures that results in the cache have been committed and remain unaltered, effectively sidestepping the aforementioned problem.

We choose the latter approach because the first option involves significant bookkeeping and introduces unnecessary overhead. The second approach, is preferred as the cost of always having to look up over one segment is minimal and more manageable. Since we are doing a default SPIRIT lookup on the newest segments we ensure **P2** principle.

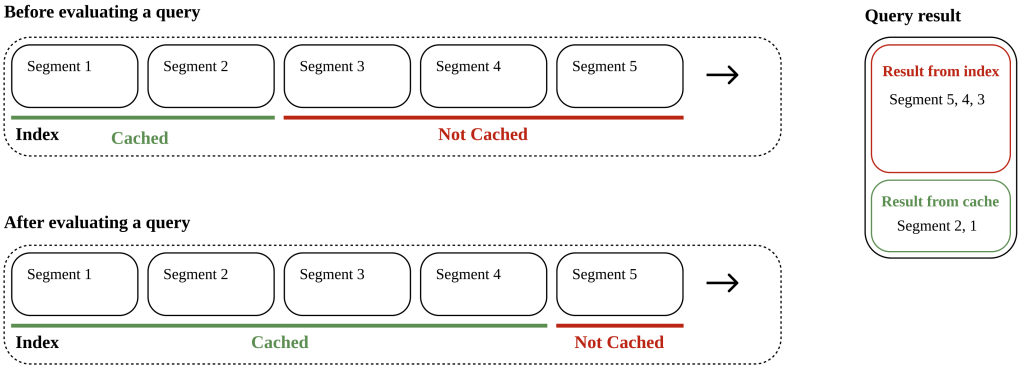


Figure 3.2: An illustration showing which segments gets cached after a query has run (bottom), and just before it has run (top). Query results always prepend results from index to results from cache to maintain relevancy ordering.

3.6 Controlling Fragmentation

We have established that Real-Time caching has a high likelihood of fragmentation (Section 2.4). Considering the chosen caching policy as detailed in Section 3.5, we add new data onto the cache whenever a repeating query is updated. So, we enable adding variable sized cache entries out of order, which gives rise to fragmentation. A popular solution to fix fragmentation is to lazily de-fragment the underlying data structure. However this is a write intensive operation. We know that PMEM suffers under write intensive workloads so adopting such a method for removing fragmentation would likely cause the performance of the whole system to degrade.

In RT-Cache we use a preventative measure towards fragmentation. When we follow our real-time caching policy we do not cache the most recent segment. This allows new documents to collect before being considered to be cached. When a query requires caching it is expected that the number of results to be cached is large enough that

the variable sized cache entries would not create a noticeable trade off. However, this assumption does not hold true for low complexity terms.

The greatest performance drop would be observed in very frequent low complexity terms. Terms that would have almost no results but are frequent enough to reserve a cache block each time they are seen. To counter this, we implement an optimisation where a cache block isn't initialised if the number of results to be cached is 0. Instead, we update the related cache bucket with the last seen segment ID to suggest that the query was cached. This allows future query evaluators to know that the new segments are cached without allocating extra blocks in the cache entry.

This approach will remove the worst possible cases of fragmentation. However, there is room for improvement with our method. We do not have a solution for low complexity queries that have very little non zero results. In our evaluation we determine whether it is worth caching low complexity queries.

3.7 Query Evaluators and RT-Cache

SPIRIT's query evaluators run in parallel with the ingest pipeline, processing each query sequentially. These evaluators remain idle until queries are received. Upon receiving queries, the evaluators engage in computation by extracting terms and iterating over segments from newest to earliest. This involves reading each segment and getting postings for terms that match the query. SPIRIT query evaluators return queries in binary format, designed to prevent query IO from becoming a performance bottleneck. SPIRIT uses two different algorithms to evaluate single and multi-term queries. It heavily optimises single term queries by placing all postings related to a term in a contiguous block of memory making retrieval very fast. However, in the case of multi-term querying, SPIRIT traverses posting lists for each term until postings from the same document are located. This process lacks optimization. To address this, we choose to utilize RT-Cache exclusively to cache multi-term queries for SPIRIT.

To satisfy **P1** principle we have chosen a cache first policy. The query evaluator will initially do a cache lookup before evaluating the query. If the cache lookup is successful, it begins by comparing the last segment ID from the cached result with the current segment ID. This comparison determines the number of additional segments that need to be examined. Based on this, a default SPIRIT evaluation is conducted on the newest segments. This is achieved by iterating from the current segment down to the last segment not present in the cache. The results from both the newest segments and the cached results are combined and then returned to the user. For unsuccessful cache lookups, a default SPIRIT evaluation is conducted.

Before moving on to evaluate the next query, we need to add the recently processed query to the cache. Considering that we combine the results from the cache with the SPIRIT results before returning them to the user, we choose to store the raw binary results in the cache as well. This approach eliminates the need for additional computation to adapt

the cache results to the output format.

When adding into the cache, if the original cache lookup was unsuccessful, a new cache bucket associated with this query must be created. The results are copied into a cache block and linked to the cache bucket. However, it is essential to refrain from copying results linked to the current segment, following our real-time caching policy. In instances where the cache lookup is successful, the cache needs to include any additional results from newer segments. The query evaluators are adjusted to keep track of new results discovered during the evaluation phase, which are later utilized to adjust the cache. Similar to the previously mentioned case, these results should exclude those from the most recent segment. The process involves copying the results into a cache block, which is then appended to the relevant cache block list for the query. The last seen segment ID tag is updated to match the current segment ID. This ensures that any subsequent queries for the same cached query yields updated results.

3.8 Eviction

We use LRU as our eviction policy. Cache buckets are ordered based on recency. This is achieved by maintaining a doubly linked list of cache buckets, the first in the list is the most recent while the last in the list is the least recent. When adding a new query we append the cache bucket to the start of the list. If the cache reaches capacity either due to not having enough entries in the hash table or not having enough space for data blocks it will choose the bucket at the end of the list for eviction. RT-Cache evicts the entire entry rather than a few data blocks from the entry. Not removing the entire query requires extra care when handling the metadata for that cache entry because we would be deleting some parts of the cached index over others. Whereas when evicting the whole entry no care needs to be taken related to the meta data which makes it a fast process.

We do not explore the effect of cache eviction against performance of Real-Time caching. However RT-Cache provides a framework where more complicated eviction policies can be evaluated.

Chapter 4

Evaluation

4.1 Hardware

We conducted experiments on a Dell PowerEdge R740 server equipped with two sockets, running the Ubuntu 18.04.1 Linux OS (5.4.0 kernel). Each CPU in this server is an Intel Xeon Gold 6252N, operating at a clock speed of 2.3 GHz with 48 physical cores (96 logical cores), along with a 36 MB L3 cache.

Each of the integrated memory controllers (iMC) supports six memory channels. Each channel is connected to a 32 GB Micron DDR4 DIMM (400 GB total DRAM). Additionally, each channel attaches to a 128 GB Intel Optane NVDIMM (1.5 TB total PMEM).

4.2 Datasets and Workloads

For our dataset preparation and ingestion process, we obtain Wikipedia's English corpus from the Luceneutil website (McCandless, 2022). This corpus provides a set of line-terminated files, each of which is 1 KB in size. Our dataset closely mirrors real-world scenarios, as real-time search is commonly employed on platforms like social media, which feature a high volume of short posts. We evaluate SPIRIT using different document sizes, one million and 10 million documents (default).

When evaluating query performance, we need to include query terms that appear in the source corpus at various frequencies. Frequently occurring words stress the read performance due to the need to process a large number of results, whereas rarer words terminate search early. To achieve this we use terms sourced from the `topTerms20120502.txt` file accessible on the Luceneutil website (McCandless, 2022). These terms are categorized into three levels: low, medium, and high complexity. We notice that SPIRIT is efficient at handling single-word queries, however, faces challenges when resolving intersections

for multi term queries. To assess the potential performance improvement in SPIRIT, we conduct experiments using RT-Cache with multi-word queries, specifically focusing on HH, MM, and LL queries.

In evaluating RT-Cache, it becomes evident that the traces proposed have a notable lack of repetition. Where utilization of the cache offers little to no discernible improvement over cache-disabled configurations. To investigate whether integrating the cache has any performance benefits compared to the default SPIRIT configuration, we deemed it necessary to create synthetic workloads that allow us to control the extent of query repetition. To facilitate this, we developed a workload generator with two key parameters: (1) the likelihood of a query repeating and (2) the maximum number of other queries between two repeating queries. We used the generator to create workloads including HH, MM, and LL queries, each with varying repetition probabilities. Through these controlled experiments, we aim to assess the potential performance enhancements created by RT-Cache in the context of multi-term queries.

4.3 Default Parameters and Measurement Metrics

Unless otherwise stated, all tests are conducted in real-time mode where the index is built from 10 M documents. Query evaluation begins concurrently after 20% of the index is built. Query evaluation ends immediately after the index is fully established, and in all cases there are enough queries to maintain query evaluation while the index is building. The workload used for query evaluation is a HH (High complexity two-word query) trace with a repetition probability of 15% and a maximum of 100 queries between any two repeating queries. We use one instance of SPIRIT with one query thread for evaluation. We set SPIRIT to ingest into a hybrid DRAM/PMEM index where 5 GB is the upper bound for index in DRAM. We use Queries Per Second (QPS) normalised to a baseline with cache-disabled as the default measurement of performance. We measure cache hit rate and the proportion of segments read from cache vs index to help evaluate our results.

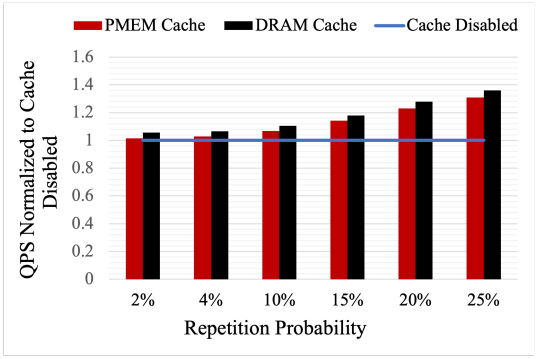
4.4 Results

With our evaluation we aim to answer the following two key questions:

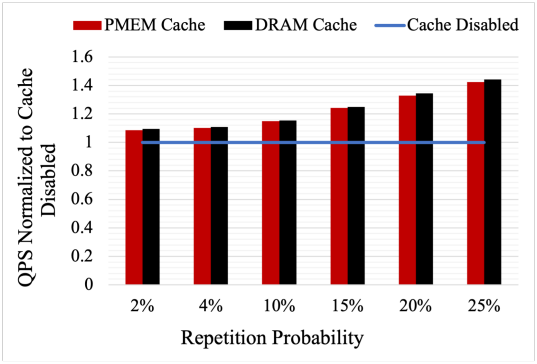
- Can the performance of a real-time search engine be improved by caching results using slow tier memory (PMEM)?
- What are the distinctive trade-offs of PMEM that impact the usability of a cache in PMEM vs DRAM.

4.4.1 Repetition Probability

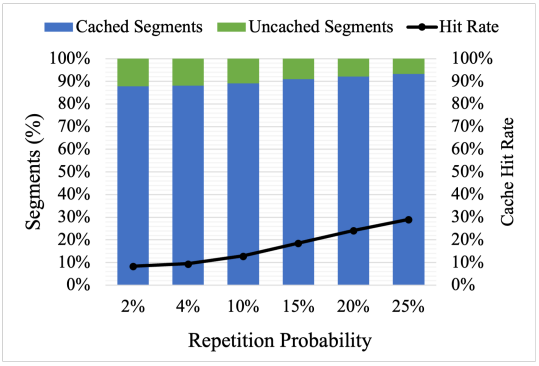
To begin, we first evaluate the performance of RT-Cache under different workloads. Figure 4.1 reports the effect of repetition probability in query workloads on the perfor-



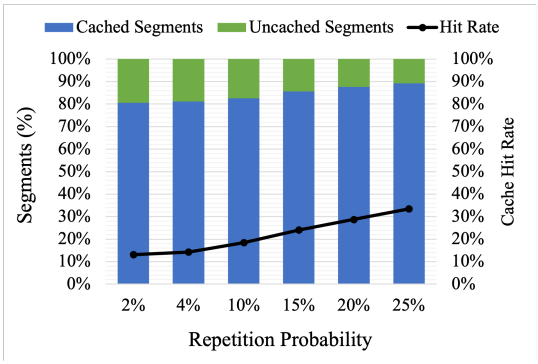
(a) 10 M documents ingested



(b) 1 M documents ingested



(c) 10 M documents ingested



(d) 1 M documents ingested

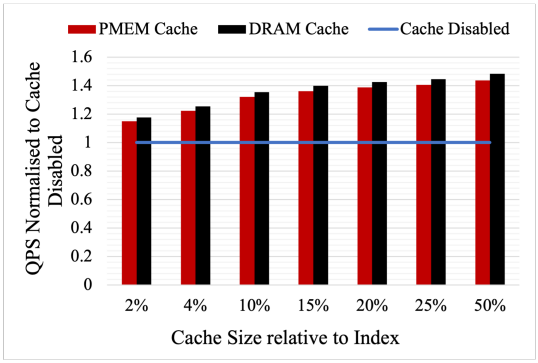
Figure 4.1: Showing query workloads with different repetition probabilities against the performance of RT-Cache measured by Queries per Second (QPS) (top). Showing the cache-hit rate and the relative proportion between cached and uncached segments (bottom)

mance of SPIRIT using RT-Cache. Figure 4.1 shows that as the repetition probability increases, the measured QPS also increase. We can see that in both 4.1a and 4.1b the QPS measured for a cache placed in DRAM is always greater compared to that of placed in PMEM. Although the general trends are similar between different index sizes shown by Figure 4.1a and Figure 4.1b there are some key differences.

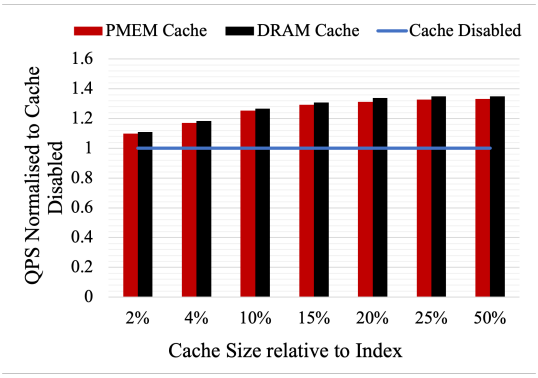
In Figure 4.1a we can see that the relative increase in performance (QPS) is quite small when the repetition probability is little; whereas in Figure 4.1b even with small repetition probabilities enabling the cache shows performance benefits. We can also see that the difference between performance of a DRAM cache against a PMEM cache is more significant in 4.1a compared to 4.1b. These observations should relate to the memory bandwidth limitation of PMEM; PMEM has a lower bandwidth compared to DRAM, which becomes critical when a large number of results need to be read (Intel, 2021). In Figure 4.1a we have a larger index compared to 4.1b so it is likely that the bandwidth bottleneck caused PMEM to be less effective.

Figures 4.1c and 4.1d show that as the repetition probability increases the cache hit rate increases, it also shows that the proportion of cached segments also increase. This explains the performance boost seen in Figures 4.1a and 4.1b. For a fair comparison between PMEM and DRAM caches, a repetition probability of 15% will be used to mock a real world use case when evaluating future metrics. With this we can conclude that the cache will only be useful if there is some repetition in the provided workloads.

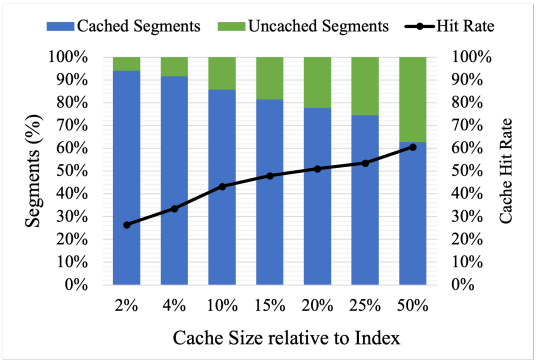
4.4.2 Cache Capacity and Partitioning



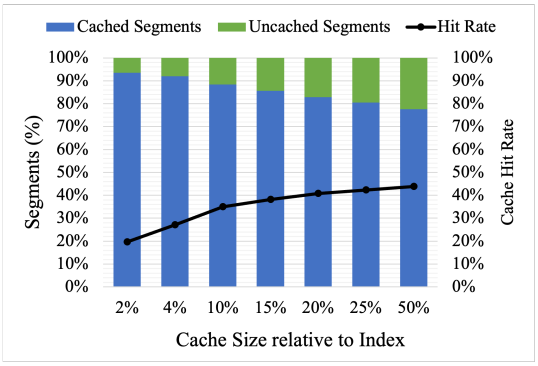
(a) 10 M documents ingested



(b) 1 M documents ingested



(c) 10 M documents ingested



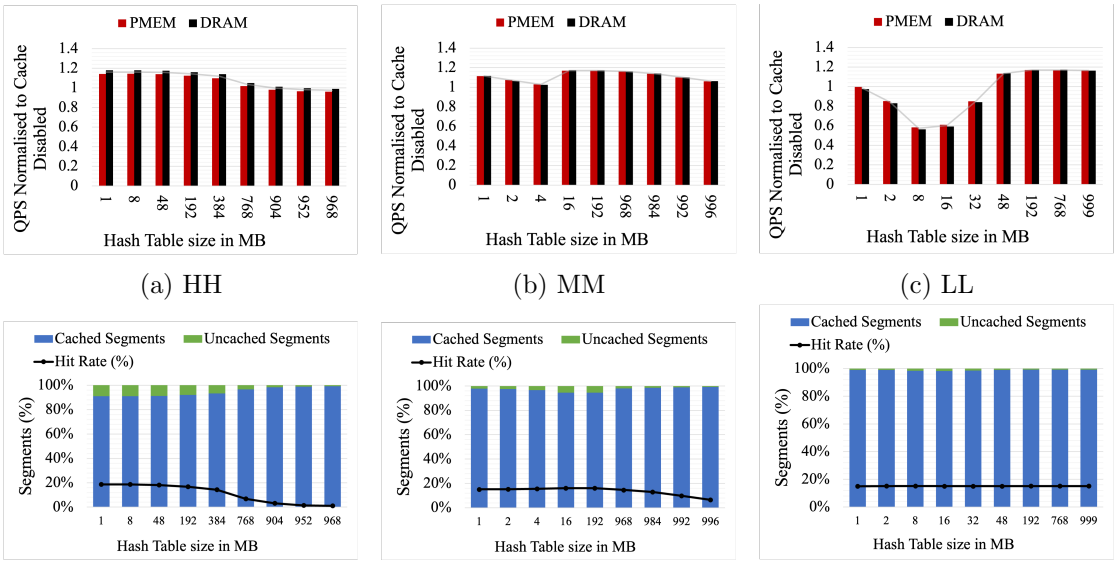
(d) 1 M documents ingested

Figure 4.2: Showing cache capacity relative to the index against the performance of RT-Cache measured by Queries per Second (QPS) (top). Showing the cache-hit rate and the relative proportion between cached and uncached segments (bottom)

In Figure 4.2 we explore the effect of cache capacity on performance of the cache. We find that as the cache capacity increases so does performance, however the larger the cache becomes the performance increase starts to plateau. We see that a cache that is 10% the size of the index provides performance boosts similar to a cache that is 50%+ the size of the index. Figure 4.2 shares similar features to Figure 4.1 like a noticeable performance difference between PMEM and DRAM caches when the index is larger.

Figure 4.2c and 4.2d show that as the cache size increases the cache hit-rate increases. However, it also shows that the proportion of uncached segments compared to cached segments increases as the capacity increases. As the cache size increases, it has the capacity to retain a greater number of queries. Some of these queries may have been cached a considerable time ago. Consequently, when such a query resurfaces, a substantial number of new segments must be searched creating the significant increase in uncached segments. This trend explains the plateauing nature of Figures 4.2a and 4.2b.

With this we can conclude that the cache only needs to be large enough to hold the most recent queries to achieve a reasonable speed up. It is also feasible to make a PMEM cache outperform a cache placed in DRAM by increasing the capacity.



(a) HH (b) MM (c) LL

(d) HH (e) MM (f) LL

Figure 4.3: Showing the effect on performance with varying hash table size (MB) in a 1000 MB cache when considering different workloads (top). Showing the cache-hit rate and the relative proportion between cached and uncached segments (bottom)

To explore how real-time caching is affected by other parameters, sensitivity studies were conducted. Figure 4.3 shows the effect of changing the hash table size and the unique trade-offs related to workload complexity. In Figure 4.3 we alter the hash table size (MB) of a cache that has a total capacity of 1000 MB.

In Figure 4.3a (HH) we see that increasing the hash table size has diminishing returns. Initially, the performance drop is not too obvious, but there is a noticeable drop in performance when the hash table is more than ∼50% of the total cache size. This is likely due to the data portion being capacity limited, so queries are being evicted prematurely before filling up the hash table. This is evident in Figure 4.3d where the

cache-hit rate drops significantly. For HH workloads, a 1 MB - 999 MB partition works most ideally since the majority of space is required to store the results for each query.

We see a valley like trend in Figures 4.3b and 4.3c. That is when the hash table increases in size the relative performance decreases initially, but the performance increases after a certain ratio. This is most noticeable in LL queries from Figure 4.3c. This effect is unnoticeable in Figure 4.3a (HH) but can be seen from the raw data. As the size of a hash table increases, the cost of cache misses will increase. This is because when the cache is full, a cache miss will have iterated through every entry before determining that the query is not in the cache. For low complexity queries this trade-off takes a noticeable performance hit. Whereas in high complexity queries, they gain a significant advantage when there is a cache hit, so the performance bottleneck is left unnoticed.

In both Figures 4.3b and 4.3c after a certain hash table size is reached the performance jumps to 20% above the baseline. For MM queries from Figure 4.3b this is 16 MB and LL queries from Figure 4.3c this is 48 MB. This is attributed to a limitation in the artificial workloads used. The suggested hash table sizes can accommodate almost all queries that were run, meaning any repetition will be cached. Although there is a 15% chance a query will repeat in the next 100 queries it is still possible for the same query to show up thousands of queries away. These rogue duplicate queries are being stored in the large hash tables such that there is a performance benefit. Since the artificial workloads used aren't a perfect demonstration of real world use, the conclusion related to sensitivity is that it depends extensively on the workloads provided.

In Figure 4.3a the difference between PMEM and DRAM is quite clear, DRAM performs better than PMEM because of the better memory bandwidth. In Figure 4.3b this trend is less clearly shown since there is less data to retrieve. However, surprisingly, in Figure 4.3c PMEM has better performance compared to DRAM when the hash table is small; this trend was unexpected since PMEM should have worse latency and bandwidth compared to DRAM. Low-complexity queries are highly sensitive to external factors due to their efficiency. It is probable that placing RT-Cache and the index in DRAM introduced additional stress, leading to slightly poorer performance compared to RT-Cache in PMEM.

4.4.3 Index Size

From previous evaluations we found that the index size had an effect on the performance difference between PMEM and DRAM. Figure 4.4 shows the effect of index size on the performance of the cache. For each index we use a cache that is 10% the size of the index. Figure 4.4 shows that as the index size grows, the performance difference between RT-Cache placed in PMEM and DRAM becomes more obvious. This has been attributed to PMEM having a lower memory bandwidth compared to DRAM (Intel, 2021).

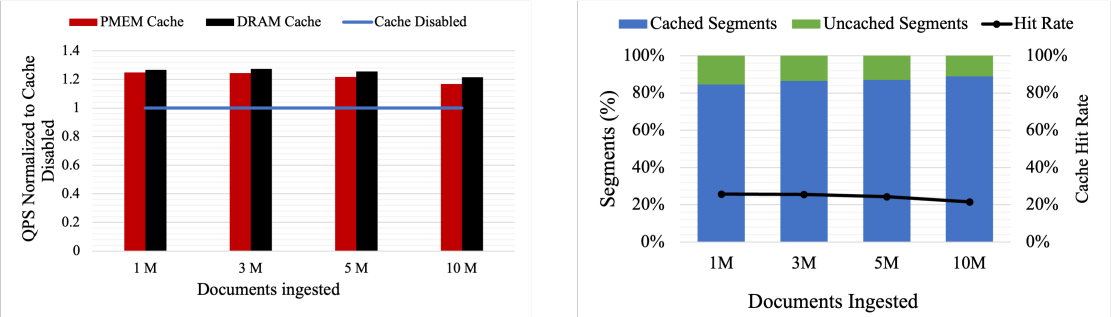


Figure 4.4: Showing index size ingested against the performance of RT-Cache measured by Queries per Second (QPS) (left). Showing the cache-hit rate and the relative proportion between cached and uncached segments (right)

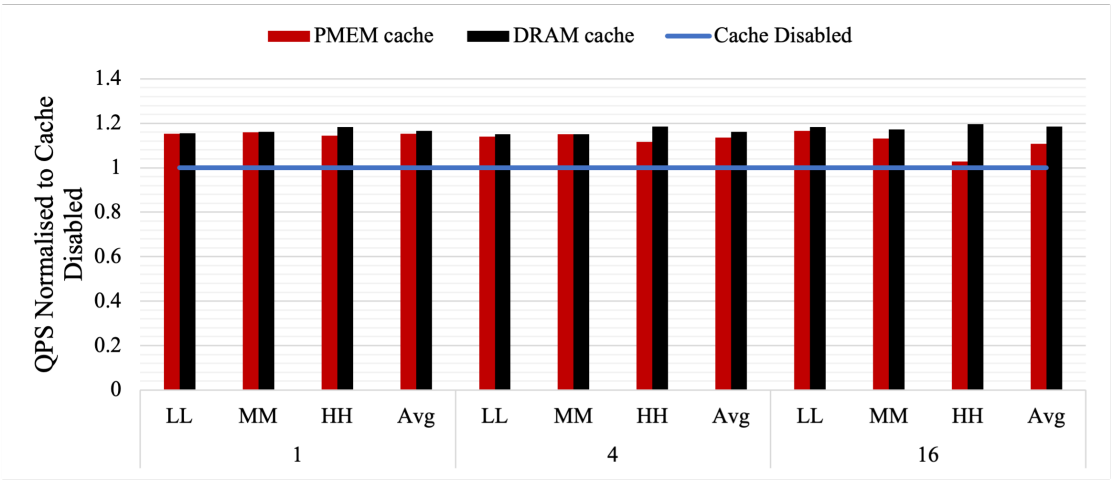


Figure 4.5: Showing the impact of multiple executors and different workloads on the performance of RT-Cache

4.4.4 Multiple Executors and Workload Complexity

To further analyse properties that may affect RT-Cache when placed on PMEM against DRAM, multiple executors were benchmarked. Figure 4.5 shows the effect of multiple concurrent executors of SPIRIT against the performance of one executor. In each test DRAM performs better compared to PMEM. In all three cases HH workloads always have the greatest difference between DRAM and PMEM performance. This is followed by MM workloads, and the least difference is shown by LL workloads. These observations relate to the memory bandwidth limitation of PMEM discussed in Section 4.4.3.

Figure 4.5 shows that as the number of executors increase the performance of a PMEM cache significantly worsens. This is clearly shown in the HH workload for 16 executors where the PMEM cache only provides a marginal boost in performance compared to

cache disabled. With more executors the memory hardware is being accessed more randomly, writes and reads may happen concurrently. Writes hurt PMEM, the bandwidth of PMEM depends significantly on the writes throughput (Fedorova et al., 2022). The Figure 4.5 shows that the access times for DRAM are not affected shown by consistent results across the different number of executors. In conclusion RT-Cache runs poorly when placed in PMEM when the hardware is shared across multiple processes.

4.4.5 RT-Cache and DRAM only index

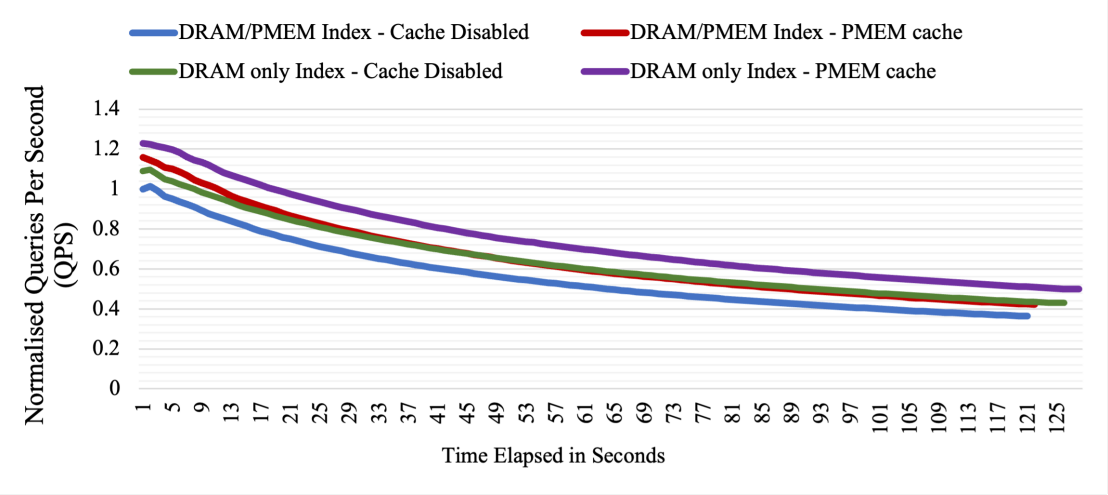


Figure 4.6: Showing the real-time effect of using the RT-Cache placed in PMEM when concurrently building the index in either hybrid DRAM/PMEM or exclusively DRAM.

Figure 4.6 shows the real-time use of RT-Cache in SPIRIT while concurrently ingesting 10 M documents. We have SPIRIT evaluating HH queries concurrently after 20% of the index is ingested. Figure 4.6 shows that as the index grows the QPS naturally decreases. We see that in cache disabled configurations using a DRAM only index performs better than a hybrid DRAM/PMEM index. These results match prior literature on SPIRIT however this trade-off is accepted since the goal was to find a DRAM alternative that does not significantly hit performance (Akram and Atkinson, 2022). Figure 4.6 shows that using RT-Cache with a hybrid DRAM/PMEM index provides comparable results to having SPIRIT create the index in DRAM exclusively. This shows that it is feasible to use real-time caching on PMEM and expect reasonable speedups.

Chapter 5

Concluding Remarks

5.1 Conclusion

In this work we present RT-Cache, a new take on real-time caching utilising Intel Optane DC Persistent Memory. We build RT-Cache on top of SPIRIT, a real-time search engine that utilizes two-tiered memory (DRAM/PMEM) for the index. We find that using RT-Cache with SPIRIT when ingesting documents to a hybrid DRAM/PMEM index can give performance gains close to a DRAM-only index with no caching. This demonstrates that caching in PMEM is a feasible alternative when expecting to remove DRAM strain in real-time search.

However, we find that the trade-offs of PMEM related to random accesses and write-intensive tasks reduce the overall applicability of RT-Cache in specific conditions: (1) when running high complexity queries with large results, (2) when having multiple concurrent instances of SPIRIT running and (3) when evaluating low complexity queries that are highly sensitive to memory latency. This suggests that RT-Cache requires future work related to optimising the specific limitations of PMEM.

5.2 Future Work

Our work successfully shows that real-time search results can be cached in PMEM so that an overall speedup is obtained. Although we find a use for real-time caching in SPIRIT the performance gain depends extensively on the workload. We found different workloads and configurations that made RT-Cache to have worse performance compared to not adopting it. We suggest future research to be done using real world query traces to correctly mimic proper workloads. One suggestion is to use YCSB query traces (Cooper, 2021). Although using RT-Cache allowed SPIRIT to gain a performance boost, it follows simple caching principles. RT-Cache uses a simple implementation of LRU for eviction.

Future work can explore the effect of different eviction techniques in real-time caches that are placed in PMEM. SPIRIT has the ability to restrict the number of query results. Our caching policy does not work as well if the total required number of results is found in the index, since the cached data would not be used. RT-Cache provides a framework where other real-time caching policies can be investigated.

RT-Cache does not address all limitations and unique trade-offs offered by Persistent memory nor does it explore intelligent caching solutions. There are a few optimisations that can be adopted on RT-Cache that can improve its overall performance.

- RT-Cache employs a cache first policy, this led to unnecessary cache lookups which created a noticeable performance hit in Figure 4.3. We can avoid such unnecessary cache lookups by utilising bloom filters to predict whether a query would be in the cache.
- We noticed that query complexity affects the performance boost differently in Section 4.4.2, and there is no common configuration that will boost performance in all query types. We can have multiple versions of RT-Cache to house different types of queries so that each will be optimised specifically for the type of query cached. We can implement a complexity predictor by using meta data related to result length.
- Our real-time caching policy requires an initial cache lookup to determine the last segment ID seen in the cache. However, in all cases the query evaluator will have to start a default SPIRIT lookup from the most recent segment and back. We can parallelise cache lookup and SPIRIT search so that when cache lookup terminates, SPIRIT would have already retrieved the most recent results.
- RT-Cache can only be placed exclusively on DRAM, or PMEM. We can explore caching performance when the pre-allocated heap is over hybrid memory. We can follow a SPIRIT like ingestion pipeline where we cache into DRAM and move results into PMEM.

Bibliography

Abulila, A.; Mailthody, V. S.; Qureshi, Z.; Huang, J.; Kim, N. S.; Xiong, J.; and Hwu, W.-m., 2019. Flatflash: Exploiting the byte-accessibility of ssds within a unified memory-storage hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19 (Providence, RI, USA, 2019), 971–985. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3297858.3304061. https://doi.org/10.1145/3297858.3304061. [Cited on page 1.]

Akram, S., 2021. Exploiting intel optane persistent memory for full text search. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2021 (Virtual, Canada, 2021), 80–93. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3459898.3463906. https://doi.org/10.1145/3459898.3463906. [Cited on pages 1 and 4.]

Akram, S. and Atkinson, A., 2022. Spirit: Scalable, persistent, in-memory real-time inversion of text. [Cited on pages 1, 3, 4, 5, 7, and 20.]

Auradkar, A.; Botev, C.; Das, S.; De Maagd, D.; Feinberg, A.; Ganti, P.; Gao, L.; Ghosh, B.; Gopalakrishna, K.; Harris, B.; Koshy, J.; Krawez, K.; Kreps, J.; Lu, S.; Nagaraj, S.; Narkhede, N.; Pachev, S.; Perisic, I.; Qiao, L.; Quiggle, T.; Rao, J.; Schulman, B.; Sebastian, A.; Seeliger, O.; Silberstein, A.; Shkolnik, B.; Soman, C.; Sumbaly, R.; Surlaker, K.; Topiwala, S.; Tran, C.; Varadarajan, B.; Westerman, J.; White, Z.; Zhang, D.; and Zhang, J., 2012. Data infrastructure at linkedin. In *2012 IEEE 28th International Conference on Data Engineering*, 1370–1381. doi:10.1109/ICDE.2012.147. [Cited on page 1.]

Bai, X. and Junqueira, F. P., 2012. Online result cache invalidation for real-time web search. In *Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. https://api.semanticscholar.org/CorpusID:12164977. [Cited on page 5.]

Berg, B.; Berger, D. S.; McAllister, S.; Grosof, I.; Gunasekar, S.; Lu, J.; Uhlar, M.; Carrig, J.; Beckmann, N.; Harchol-Balter, M.; and Ganger,

G. R., 2020. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 753–768. USENIX Association. https://www.usenix.org/conference/osdi20/presentation/berg. [Cited on page 1.]

Bortnikov, E.; Lempel, R.; and Vornovitsky, K., 2011. Caching for realtime search. In *Advances in Information Retrieval*, 104–116. Springer Berlin Heidelberg, Berlin, Heidelberg. [Cited on pages 2 and 5.]

Busch, M.; Gade, K.; Larson, B.; Lok, P.; Luckenbill, S.; and Lin, J., 2012. Earlybird: Real-time search at twitter. In *2012 IEEE 28th International Conference on Data Engineering*, 1360–1369. doi:10.1109/ICDE.2012.149. [Cited on page 4.]

Chen, Y.; Lu, Y.; Yang, F.; Wang, Q.; Wang, Y.; and Shu, J., 2020. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20 (Lausanne, Switzerland, 2020), 1077–1091. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3373376.3378515. https://doi.org/10.1145/3373376.3378515. [Cited on page 1.]

Cooper, B., Frank, 2021. Ycsb: Yahoo cloud servicing benchmark. https://github.com/brianfrankcooper/YCSB. [Cited on page 21.]

Fedorova, A.; Smith, K. A.; Bostic, K.; LoVerso, S.; Cahill, M.; and Gorrod, A., 2022. Writes hurt: Lessons in cache design for optane nvram. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22 (San Francisco, California, 2022), 110–125. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3542929.3563461. https://doi.org/10.1145/3542929.3563461. [Cited on page 20.]

Gupta, S.; Oh, Y.; Yan, L.; Sutherland, M. J.; Bhattacharjee, A.; Falsafi, B.; and Hsu, P., 2023. Astriflash: A flash-based system for online services. In *Proceedings of the High Performance Computer Architecture (HPCA)*. [Cited on pages 1 and 4.]

Handy, J., 2018. Emerging memories today: Why emerging memories are necessary. https://thememoryguy.com/. [Cited on page 1.]

Huang, J.; Badam, A.; Qureshi, M. K.; and Schwan, K., 2015. Unified address translation for memory-mapped ssds with flashmap. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15 (Portland, Oregon, 2015), 580–591. Association for Computing Machinery, New York, NY, USA. doi:10.1145/2749469.2750420. https://doi.org/10.1145/2749469.2750420. [Cited on page 1.]

Intel, 2019. Intel® optane™ technology: Memory or storage? https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/what-is-optane-technology-brief.pdf. [Cited on page 3.]

Intel, 2021. Intel® optane™ persistent memory – memory mode decision guide. https://www.intel.com/content/www/us/en/developer/articles/guide/intel-optane-persistent-memory-decision-guide.html. [Cited on pages 15 and 18.]

Kilrain, J., 2022. Htfh-rt-search-cache. https://github.com/EngineersBox/HTFH-RT-Search-Cache. [Cited on page 9.]

Lee, B. C.; Zhou, P.; Yang, J.; Zhang, Y.; Zhao, B.; Ipek, E.; Mutlu, O.; and Burger, D., 2010. Phase-change technology and the future of main memory. *IEEE Micro*, 30, 1 (Jan. 2010). doi:10.1109/MM.2010.24. [Cited on page 3.]

Lin, J.; Lok, P.; Larson, B.; Gade, K.; Luckenbill, S.; and Busch, M., 2012. Earlybird: Real-time search at twitter. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. doi:10.1109/ICDE.2012.149. [Cited on pages 1 and 4.]

McAllister, S.; Berg, B.; Tutuncu-Macias, J.; Yang, J.; Gunasekar, S.; Lu, J.; Berger, D. S.; Beckmann, N.; and Ganger, G. R., 2021. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21 (Virtual Event, Germany, 2021), 243–262. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3477132.3483568. https://doi.org/10.1145/3477132.3483568. [Cited on page 1.]

McCandless, M., 2022. Lucene benchmarks over time. https://github.com/mikemccand/luceneutil. [Cited on page 13.]

Mutlu, O. and Subramanian, L., 2014. Research problems and opportunities in memory systems. *Supercomputing Frontiers and Innovations*, 1, 3 (Oct 2014). [Cited on page 1.]

O'Neil, P.; Cheng, E.; Gawlick, D.; and O'Neil, E., 1996. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33, 4 (jun 1996), 351–385. doi:10.1007/s002360050048. https://doi.org/10.1007/s002360050048. [Cited on page 4.]

Sazoglu, F. B.; Cambazoglu, B. B.; Ozcan, R.; Altingövde, I. S.; and Ulusoy, Ö., 2013. Strategies for setting time-to-live values in result caches. *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, (2013). https://api.semanticscholar.org/CorpusID:5821981. [Cited on page 5.]

Tonozzi, N. and Daniliuc, D., 2020. Reducing search indexing latency to one second. https://blog.twitter.com/. [Cited on page 4.]

Twitter, 2023. Twitter's recommendation algorithm. https://blog.twitter.com/engineering/en_us/topics/open-source/2023/twitter-recommendation-algorithm. [Cited on page 1.]

Weiner, J.; Agarwal, N.; Schatzberg, D.; Yang, L.; Wang, H.; Sanouillet, B.; Sharma, B.; Heo, T.; Jain, M.; Tang, C.; and Skarlatos, D., 2022. Tmo: Transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22 (Lausanne, Switzerland, 2022), 609–621. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3503222.3507731. https://doi.org/10.1145/3503222.3507731. [Cited on page 1.]

Wilson, C., 2022. The difference between real-time, near real-time, and batch processing in big data. https://www.precisely.com/blog/big-data/difference-between-real-time-near-real-time-batch-processing-big-data. [Cited on page 3.]

Xie, Z.; Cai, Q.; Chen, G.; Mao, R.; and Zhang, M., 2018. A comprehensive performance evaluation of modern in-memory indices. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 641–652. doi:10.1109/ICDE.2018.00064. [Cited on pages 1 and 4.]

Yang, J.; Kim, J.; Hoseinzadeh, M.; Izraelevitz, J.; and Swanson, S., 2020. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST)*. https://www.usenix.org/conference/fast20/presentation/yang. [Cited on page 3.]

Yldiz, S., 2023. Why cache invalidation is hard and how to solve it. https://levelup.gitconnected.com/why-cache-invalidation-is-hard-and-how-to-solve-it-7921ffe4f6eb. [Cited on page 5.]

Zhuang, Y.; Burstein, P.; Bennett, F.; Bezek, S.; Busch, M.; Dudte, C.; Fujii, K.; Garg, V.; Hardt, M.; Hurley, J.; Hook, A.; Johnson, N.; Larson, B.; Lewis, A.; Li, Z.; Lok, P.; Luckenbill, S.; Mishne, G.; Saraf, Y.; Shenoy, J.; Snider, T.; Tie, H.; Vallis, O.; Wang, J.; Wang, J.; Wang, L.; Wang, T.; Yan, B.; Youll, J.; Zeng, M.; and Zhao, K., 2014. Building a complete tweet index. https://blog.twitter.com/engineering/en_us/a/2014/building-a-complete-tweet-index. [Cited on pages 1 and 4.]