UNIVERSITY OF CALIFORNIA SAN DIEGO

Query Processing of Sorted Lists on Modern Hardware

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Jianguo Wang

Committee in charge:

Professor Yannis Papakonstantinou, Chair
Professor Steven Swanson, Co-Chair
Professor Nuno Bandeira
Professor Pamela Cosman
Professor Alin Deutsch

2019

The dissertation of Jianguo Wang is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Co-Chair

Chair

University of California San Diego

2019

DEDICATION

To my family.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the help of many people.

I am grateful for my two excellent advisors, Yannis Papakonstantinou and Steven Swanson, for their guidance and support. They provided unique research perspectives to me: databases and modern hardware, which became the main theme of my research in the last few years. They taught me how to conduct good research, starting from choosing an interesting topic, justifying every design decision critically, evaluating the design thoroughly, all the way to writing a paper. Besides that, they also gave me professional advices of finding summer internships and full-time jobs.

I would like to thank Nuno Bandeira, who introduced a very interesting mass spectrometry project that nicely folded many of my prior works on sorted lists into solving a real-world problem. He also provided many insightful comments to improve performance and part of financial support during my studies at UCSD.

Special thanks go to two other committee members, Pamela Cosman and Alin Deutsch, for their precious time of reading the thesis and serving on the committee.

I thank Victor Vianu and Arun Kumar for their valuable comments to improve my work and presentation skills during the database seminar.

I am indebted to my industry mentors during my summer internships: Dongchul Park, Yang-Suk Kee, David Lomet, Justin Levandoski, Garret Swart, and Weiwei Gong. Thanks for their supervision and the opportunity to get involved in industrial systems.

I am also hugely appreciative to my co-authors: Yannis Papakonstantinou, Steven Swanson, Nuno Bandeira, Dongchul Park, Yang-Suk Kee, Chunbin Lin, Ruining He, Yuliang Li, Yang Liu, Moojin Chae, and Benjamin Pullman. Thanks for their collaboration, brainstorming, and encouragements that lead to fruitful research.

I also thank my friends: Chunbin Lin, Ruining He, Costas Zarifis, Xun Jiao, Yanqin Jin, Yuliang Li, Jiapeng Zhang, Xiaochu Liu, Pingfan Meng, Yang Liu, Andiry Xu, Lu Zhang, Jian

Yang, Benjamin Pullman, and Ying Lv. Life is meaningless without friends and I really enjoyed the time with them.

Finally, I would like to thank my parents, sister, and girlfriend, for their unconditional love and support. I dedicate this thesis to them.

Chapter 2 contains material from “MILC: Inverted List Compression in Memory” by Jianguo Wang, Chunbin Lin, Ruining He, Moojin Chae, Yannis Papakonstantinou, and Steven Swanson, which appeared in Proceedings of Very Large Data Bases Conference (VLDB), pages 853–864, 2017. The dissertation author was the primary investigator of the paper.

Chapter 3 contains material from “Evaluating List Intersection on SSDs for Parallel I/O Skipping” by Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson, which was submitted for publication. The dissertation author was the primary investigator of the paper.

Chapter 4 contains material from “Index-based, High-dimensional, Cosine Threshold Querying with Optimality Guarantees” by Yuliang Li, Jianguo Wang, Benjamin Pullman, Nuno Bandeira, and Yannis Papakonstantinou, which appeared in Proceedings of International Conference on Database Theory (ICDT), 2019. The dissertation author was the primary investigator of the paper.

VITA

06 / 2009	B. S. in Computer Science, Zhengzhou University
12 / 2012	M. Phil. in Computer Science, The Hong Kong Polytechnic University
03 / 2019	Ph. D. in Computer Science, University of California San Diego

PUBLICATIONS

Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, Steven Swanson. "Evaluating List Intersection on SSDs for Parallel I/O Skipping." Submitted, 2019.

Yuliang Li, **Jianguo Wang**, Benjamin Pullman, Nuno Bandeira, Yannis Papakonstantinou. "Index-based, High-dimensional, Cosine Threshold Querying with Optimality Guarantees." *Proceedings of International Conference on Database Theory (ICDT)*, 2019.

Yang Liu, **Jianguo Wang**, Steven Swanson. "Griffin: Uniting CPU and GPU in Information Retrieval Systems for Intra-Query Parallelism." *Proceedings of ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 327–337, 2018.

Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, Steven Swanson. "An Experimental Study of Bitmap Compression vs. Inverted List Compression." *Proceedings of ACM Conference on Management of Data (SIGMOD)*, pages 993–1008, 2017.

Jianguo Wang, Chunbin Lin, Ruining He, Moojin Chae, Yannis Papakonstantinou, Steven Swanson. "MILC: Inverted List Compression in Memory." *Proceedings of Very Large Data Bases Conference (VLDB)*, pages 853–864, 2017.

Jianguo Wang, Dongchul Park, Yang-Suk Kee, Yannis Papakonstantinou, Steven Swanson. "SSD In-Storage Computing for List Intersection." *Proceedings of International Workshop on Data Management on New Hardware (DaMoN)*, affiliated with SIGMOD, pages 1–7, 2016.

Jianguo Wang, Eric Lo, Man Lung Yiu, Jiancong Tong, Gang Wang, Xiaoguang Liu. "Cache Design of SSD-Based Search Engine Architectures: An Experimental Study." *ACM Transactions on Information Systems (TOIS)*, 32(4): 1–26, 2014.

Jianguo Wang, Eric Lo, Man Lung Yiu, Jiancong Tong, Gang Wang, Xiaoguang Liu. "The Impact of Solid State Drive on Search Engine Cache Management." *Proceedings of ACM Conference on Research and Development in Information Retrieval (SIGIR)*, pages 693–702, 2013.

Jianguo Wang, Eric Lo, Man Lung Yiu. "Identifying the Most Connected Vertices in Hidden Bipartite Graphs using Group Testing." *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 25(10): 2245-2256, 2013.

ABSTRACT OF THE DISSERTATION

Query Processing of Sorted Lists on Modern Hardware

by

Jianguo Wang

Doctor of Philosophy in Computer Science

University of California San Diego, 2019

Professor Yannis Papakonstantinou, Chair
Professor Steven Swanson, Co-Chair

A sorted list is a collection of ⟨ID, value⟩ pairs sorted either by ID (called an ID-sorted list) or value (called a value-sorted list). Although simple, it is surprisingly the fundamental structure in a wide range of data management systems, e.g., databases, search engines, graph systems, and high-dimensional similarity search systems. Existing query processing over sorted lists were mainly optimized for past hardware, e.g., slow CPUs, small DRAM, and slow disks. However, with the emergence of new hardware, e.g., fast multi-core CPUs, big DRAM, fast SSDs, we find that many of existing design decisions become obsolete.

Motivated by this, in this dissertation, we investigate how to best leverage the characteristics of emerging hardware to accelerate query processing over sorted lists. First, we studied the impact of big memory and modern CPUs to ID-sorted list compression. We designed a new compression called MILC, which is SIMD-aware, cache-aware, and most importantly can answer queries directly over compressed lists. Second, we studied the impact of fast SSDs to ID-sorted list intersection. We developed and evaluated SSD-aware intersection algorithms that can skip unnecessary data pages to leverage SSDs fast random accesses. Finally, we folded the proposed hardware-specific techniques into a real-world modern application – mass spectrometry search – that involves query processing on both ID-sorted lists and value-sorted lists. We also developed domain-specific and data-specific optimizations besides hardware-specific optimizations. Putting them together, our new mass spectrometry search system is significantly faster than state-of-the-art systems.

Chapter 1

Introduction

Recent years have witnessed the rapid development of hardware architecture ranging from compute, memory, and storage. In the compute side, CPU frequency has reached the physical limit and the trend is shifted towards designing CPUs with more cores, wider SIMD, more caches, and heterogenous processors (e.g., GPU). In the memory side, DRAM capacity has increased significantly such that many systems (including big data systems exemplified by Apache Spark[1]) become memory-resident. For example, Intel Xeon E7 v2 servers can support 6TB of DRAM.[2] In the storage side, SSDs (solid-state drives) have become the mainframe in the storage market. More importantly, the development of recent NVRAM (non-volatile memory) [XS16], e.g., Intel 3D XPoint,[3] is expected to dramatically change the landscape of memory and storage by providing near-DRAM performance and disk-like persistence.

As a result, software systems have to fully leverage the underlying hardware to improve performance, because different hardware may have completely different characteristics. As an example of HDD (hard disk drive) vs. SSD (solid-state drive), HDDs are known to be extremely slow in random accesses, thus HDD-oriented systems are designed to minimize random accesses

[1]https://spark.apache.org/

[2]https://www.alphr.com/news/enterprise/387196/intel-xeon-e7-v2-servers-support-6tb-of-ram

[3]https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html

ID	Value
10	0.5
28	0.9
48	0.4
64	0.6

(a) ID-sorted list

ID	Value
28	0.9
64	0.6
10	0.5
48	0.4

(b) Value-sorted list

Figure 1.1: An example of ID-sorted list and value-sorted list

(with more sequential accesses). However, SSDs have very fast random accesses and thus SSD-based systems should minimize both random and sequential I/Os. Besides performance improvement, monetary cost will be reduced as well, especially in the age of cloud computing where users pay for computing resources [AFG$^+$10].

In this thesis, we investigate a particular type of data structure – sorted lists – to understand how modern hardware changes the design spaces. A sorted list is a collection of $\langle$ID, value$\rangle$ pairs sorted either by ID (called an ID-sorted list) or value (called a value-sorted list), see Figure 1.1. Although simple, it is surprisingly the fundamental structure in a wide range of data management systems, e.g., databases, search engines, graph systems, and high-dimensional similarity search systems. Existing query processing over sorted lists were mainly optimized for past hardware, e.g., slow CPUs, small DRAM, and slow disks. However, with the emergence of new hardware, e.g., fast multi-core CPUs, big DRAM, fast SSDs, we find that many of existing design decisions become obsolete.

Motivated by this, in this thesis, we investigate how to best leverage the characteristics of emerging hardware to accelerate query processing over sorted lists.

First, we studied the impact of big memory and modern CPUs to ID-sorted list (a.k.a inverted list) compression (Chapter 2). Traditional compression algorithms were optimized for slow disks where the goal was to minimize space overhead to reduce expensive disk I/O time in disk-oriented systems. But the recent trend is shifted towards reducing query processing time because the underlying systems tend to be memory-resident. Although there are many highly optimized compression approaches in main memory, there is still a considerable performance gap between query processing over compressed lists and uncompressed lists. We aim to bridge this performance gap for the first time by proposing a new compression scheme, namely, MILC (memory inverted list compression). MILC relies on a series of techniques including offset-oriented fixed-bit encoding, dynamic partitioning, in-block compression, cache-aware optimization, and SIMD acceleration. We conduct experiments on three real-world datasets in information retrieval, databases, and graph analytics to demonstrate the high performance and low space overhead of MILC. We compare MILC with 12 recent compression algorithms and experimentally show that MILC improves the query performance by up to $13.2\times$ and reduces the space overhead by up to $4.7\times$.

Second, we studied the impact of fast SSDs to ID-sorted list intersection (Chapter 3). Traditional disk-based intersection algorithms were optimized for hard disk drives (HDDs) since HDDs have dominated the storage market for decades. In particular, those HDD-centric algorithms read every relevant list entirely to memory to minimize expensive random reads by performing sequential reads, although many entries in the list may be useless. Such a tradeoff makes perfect sense on HDDs because random reads are one to two orders of magnitude slower than sequential reads. However, fast solid state drives (SSDs) have changed this landscape by improving random I/O performance dramatically. More importantly, SSDs are manufactured with multiple flash channels to support parallel I/Os. As a result, the performance gap between random and sequential reads becomes very small ($1\times \sim 2\times$) on SSDs. This means that HDD-optimized intersection algorithms might not be suitable for SSDs because the total amount of data accessed is unnecessarily high. To understand the impact of SSDs to list intersection, we tune five existing in-memory intersection algorithms to be SSD-aware with the idea of parallel I/O skipping, and experimentally evaluate them on synthetic datasets and real datasets. We investigate the effect of skip pointers, bloom filters, parallelism, cache, compression, list size, list size ratio, intersection ratio, and the number of lists to the performance. Based on the results, we provide insights and lessons on how to design efficient SSD-optimized intersection algorithms.

Finally, we folded the proposed hardware-specific techniques into a real-world modern application – mass spectrometry search – that involves query processing on both ID-sorted lists and value-sorted lists (Chapter 4). A spectrum is essentially modeled as a multi-dimensional vector (implemented as a value-sorted list) and the problem is to return all database vectors having cosine similarity to a query vector exceeding a given threshold θ. These queries arise naturally in many applications, such as information retrieval, recommender systems, and mass spectrometry. Fagin's Threshold Algorithm (TA) provides an appropriate baseline index and algorithm for answering cosine threshold queries: an inverted index (i.e., a collection of ID-sorted lists) is built from the normalized database vectors during pre-processing; at query time, the algorithm traverses the index partially to gather a set of candidate vectors to be later verified for θ-similarity. We show optimizations, over the TA-inspired baseline, which minimize the number of database and index accesses. In particular, we provide a tight stopping condition to terminate early the index traversal by taking into account the normalized vector condition and devise an incremental maintenance algorithm for its fast computation. Then we show a traversal strategy exploiting a common data skewness condition which is shown to hold in real mass spectrometry datasets. Under the skewness assumption, the proposed traversal strategy has a strong, near-optimal performance guarantee. The proposed techniques are quite general since they can be applied to a large class of similarity functions beyond cosine. All the algorithms are optimized for modern CPUs, memory, and SSD storage to ensure fast performance. As a result, our new mass spectrometry search system is significantly faster than state-of-the-art systems.

Chapter 2

MILC: Inverted List Compression in Memory

2.1 Introduction

An inverted list is a sorted list of integers. Although simple, it is the standard structure in a wide range of applications. For instance, search engines usually rely on inverted lists to find relevant documents. Databases also heavily need inverted lists to accelerate SQL processing [BGGT09].

Inverted list compression is a topic that has been studied for 50 years due to its benefits in disk-oriented systems as well as recent memory-oriented systems. In disk-centric systems, compression can reduce expensive I/O time by shortening lists' sizes. Thus, list compression algorithms designed for disks (e.g., VB [TH72], Rice [RP71], Elias gamma [Eli75]) mainly focus on reducing space overhead. The CPU decompression overhead is negligible compared to the saved I/O time due to the giant performance gap between disk and CPU. In recent memory-oriented systems, compression is also beneficial because it makes the system accommodate much more data than the physical memory capacity. For example, 100GB's raw data can be pushed to a

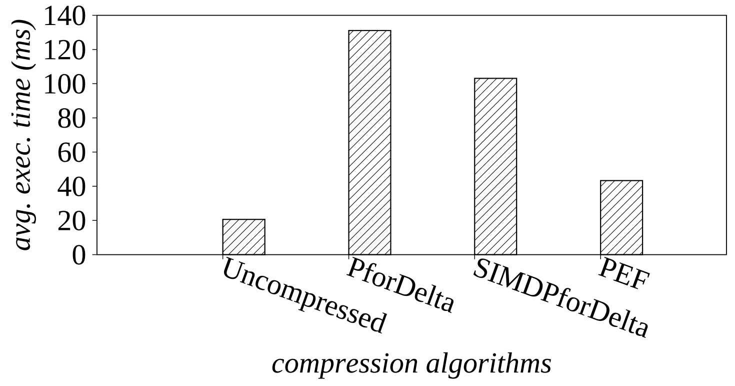


Figure 2.1: Executing queries over compressed and uncompressed lists

server with 32GB DRAM. This reduces the total cost of ownership (TCO) since main memory is still an expensive resource. As a result, many compression algorithms have been developed for in-memory inverted lists, e.g., PforDelta [ZHNB06], SIMDPforDelta [LB15b], and PEF [OV14].

Motivation. We observe that there is still a considerable performance gap for query processing over compressed lists (with state-of-the-art compression algorithms) versus uncompressed lists in memory. For example, Figure 2.1 shows the (average) execution time of running 1000 real-world search engine queries over 300GB data.[1] It shows that the performance gap is $2.1\times$ to $6.4\times$.

This raises an interesting question: *Is it possible to bridge this performance gap when operating on compressed data while still keeping low space overhead*? This work gives a positive answer to this question by proposing a new compression algorithm, namely, MILC (memory inverted list compression). Compared with uncompressed lists, MILC achieves a compression ratio up to $4.7\times$ and executes queries nearly as fast as that on uncompressed lists according to our experiments. Before diving into the technical descriptions, we define the problem first.

Problem statement. Given a sorted list L of n positive integers, the problem of inverted list compression is to compress L with as few as possible bits (smaller than the original list) while supporting query processing on compressed data as fast as possible. We mainly focus on

[1]We use the Web data described in Section 2.9.

supporting efficient membership testing – checking whether an element appears in a compressed list – because it is the core of many operations, e.g., intersection, union, difference, selection, join, successor finding, and top-k query processing.

Limitations of existing compression solutions. Existing compression algorithms for inverted lists, e.g., VB [TH72], Simple8b [AM10], GroupVB [Dea09], and PforDelta [ZHNB06], usually follow a golden rule that is to compute the differences (called *d-gaps*) between two consecutive integers (since all integers are sorted), and only encode the small d-gaps using fewer bits to save space. For example, let $L = \{8, 15, 20, 25, 35, 40, 52, 60, 65, 78, 90\}$, then existing solutions usually convert L to $L' = \{8, 7, 5, 5, 10, 5, 12, 8, 5, 13, 12\}$, where $L'[0] = L[0]$ and $L'[i] = L[i] - L[i-1]$ $(i \geq 1)$. But this is exactly why existing approaches cannot support membership testing efficiently: They have to decompress the entire list. Even with skip pointers as suggested in [MZ96], still, they need to decompress at least one block of data on the fly. Moreover, the decompression overhead is high because they need to traverse the data at least twice in order to recover the original values: (1) decode each individual d-gap; (2) calculate prefix sums. Some compression algorithms may need more rounds, e.g., PforDelta requires another round of traversal to recover the exception values. Another important drawback of d-gap-based compression algorithms is that they are unfriendly to SIMD (single instruction multiple data) due to the inherent data dependencies in computing prefix sums [LB15b]. Those compression algorithms that do not explicitly rely on d-gaps (such as EF [Eli74, Vig13], PEF [OV14]) also have problems in dealing with membership testing efficiently as we explain more in related work.

Challenges. It is challenging to design a new compression approach to achieve similar query performance as uncompressed lists while keeping low space overhead, considering the problem has been studied for many years. This particularly holds today because the underlying hardware (e.g., big memory, new processors with large CPU caches and wide SIMDs) has changed significantly. The compression format should also be compliant with CPU cache lines and SIMD instructions such that membership testing can be executed even more efficiently. To solve the problem, we need to break the traditional rule by abandoning d-gaps. This will increase the space overhead naturally. Therefore, we need to design new techniques to reduce space overhead while maintaining high query performance.

Technical overview. To address the above challenges, we develop a novel compression scheme MILC (memory inverted list compression) that achieves similar membership testing performance with uncompressed lists. The basic idea of MILC is that it partitions an input list into different blocks, then it applies offset-based (instead of delta-based) encoding and uses the same number of bits to encode all the elements within the block (Section 2.4). This is crucial to the success of MILC because it enables MILC to support membership testing directly on compressed data as we show in Section 2.4.

To further reduce space overhead and improve query performance, MILC employs four optimizations: (1) *Dynamic partitioning* (Section 2.5). It partitions a list into variable-sized blocks based on dynamic programming to minimize exception values in each block, i.e., elements in a block have low variance. This effectively reduces the space overhead because exceptions need more bits to represent, and all the other elements end up using the same high number of bits. (2) *In-block compression* (Section 2.6). MILC further splits every block into sub-blocks by smartly plugging in lightweight skip pointers to reduce the space overhead. (3) *Cache-aware optimization* (Section 2.7). MILC reorganizes data in a way that considers CPU cache line alignment. It can improve the performance of membership testing because CPU cache misses are reduced. (4) *SIMD acceleration* (Section 2.8). MILC also leverages SIMD for fast query processing.

Contribution. The main contribution of this work is a new compression scheme MILC that achieves similar membership testing performance with uncompressed lists while keeping low space overhead. MILC is tailored for modern computing hardware including big memory, fast CPU caches, and wide SIMD processing capabilities. To the best of our knowledge, this is the first inverted list compression algorithm that has such high query performance with low space overhead.

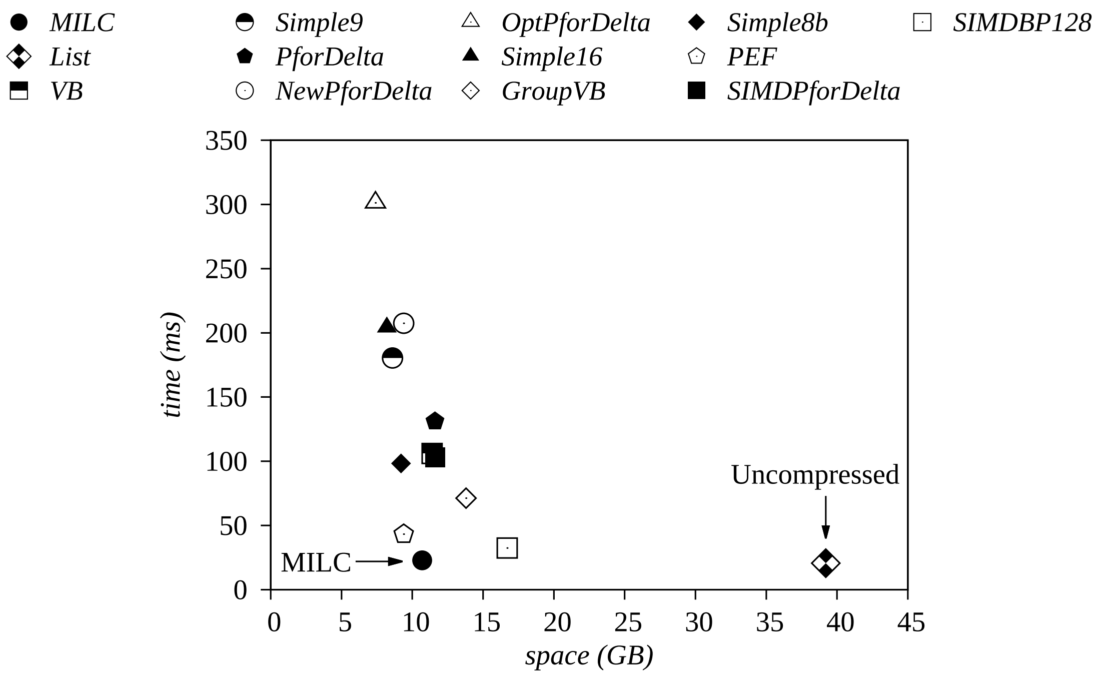


Figure 2.2: Experiments overview of MILC

Experimental overview. We conduct experiments on datasets from information retrieval, databases, and graph analytics to demonstrate the advantages of MILC with a spectrum of 12 compression algorithms in terms of query performance and space overhead. Figure 2.2 shows a preview on 300GB Web data in answering 1000 user queries,[2] and Section 2.9 describes more details. It shows that MILC runs faster than existing compression approaches and consumes low space also. Thus, MILC represents the best tradeoff for inverted list compression in main memory in terms of time and space.

2.2 Applications

In this section, we provide motivating applications that rely on inverted lists for efficient query processing. This means that a large range of applications can benefit from this work on inverted list compression.

[2]We report the membership testing time to measure the effectiveness of MILC as described in Section 2.9.

```
SELECT d_year, s_nation, p_category,
       sum(lo_revenue - lo_supplycost) as profit
FROM   date, customer, supplier, part, lineorder
WHERE  lo_custkey = c_custkey
       AND lo_suppkey = s_suppkey
       AND lo_partkey = p_partkey
       AND lo_orderdate = d_datekey
       AND c_region = 'AMERICA'
       AND s_region = 'AMERICA'
       AND d_year = 1997
       AND p_mfgr = 'MFGR#1'
```

(a) star schema join

```
SELECT d_year, s_nation, p_category,
       sum(lo_revenue - lo_supplycost) as profit
FROM   lineorderfull
WHERE  c_region = 'AMERICA'
       AND s_region = 'AMERICA'
       AND d_year = 1997
       AND p_mfgr = 'MFGR#1'
```

(b) intersection

Figure 2.3: An example of star schema join and intersection

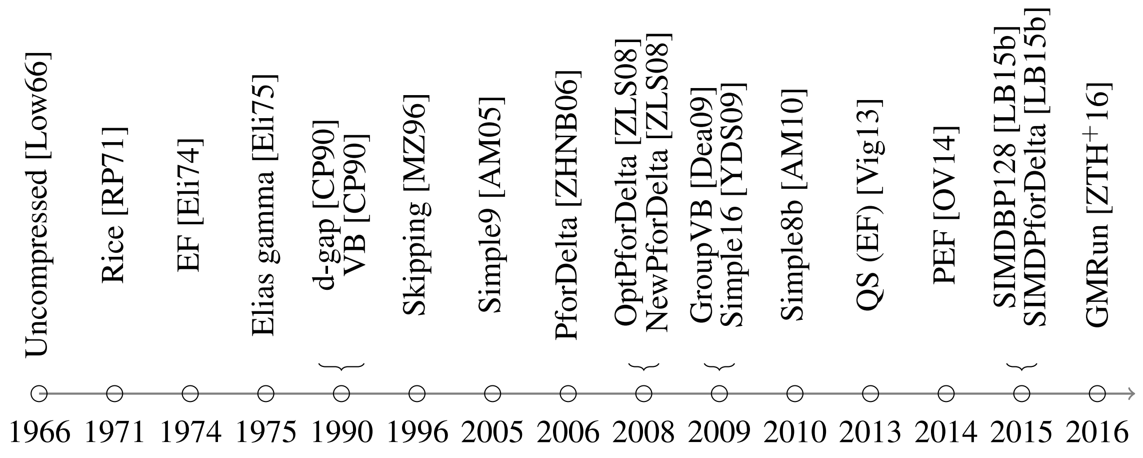


Figure 2.4: A brief history of representative inverted list compression approaches

2.2.1 Information retrieval

Information retrieval (IR) is a killer application of inverted lists to answer user queries with multiple terms [MRS08]. IR systems store an inverted list for each term all the documents that contain the term. Taking the intersection or union of the lists for a set of query terms identifies those documents that contain all or at least one of the terms.

2.2.2 Database query processing

Inverted lists are also helpful in SQL databases, especially if there is logically one huge table and the query involves many predicates, e.g., a conjunction of predicates as shown in Figure 2.3b. In this case, most databases would precompute a list of matching row IDs for each predicate to facilitate the conjunction (or intersection) query [RQH$^+$07].

Besides that, many *star schema joins* can also be framed as conjunctive queries as suggested in prior works [RQH$^+$07, BGGT09]. For instance, the star schema join in Figure 2.3a can be converted to the intersection query in Figure 2.3b where `lineorderfull` is a logical huge table that is created offline as follows [RQH$^+$07, BGGT09]: (1) `lineorderfull` = `lineorder` ⋈ `date` ⋈ `customer` ⋈ `supplier` ⋈ `part`; (2) add an additional *id* column to `lineorderfull`. Note that `lineorderfull` and `lineorder` have the same number of tuples due to many-to-one mapping.[3] Then the star schema join in Figure 2.3a can be reduced to the intersection query in Figure 2.3b as $L_1 \cap L_2 \cap L_3 \cap L_4$ where:

$L_1 = \pi_{id}\sigma_{c\_region='AMERICA'}$ (`lineorderfull`)

$L_2 = \pi_{id}\sigma_{s\_region='AMERICA'}$ (`lineorderfull`)

$L_3 = \pi_{id}\sigma_{d\_year=1997}$ (`lineorderfull`)

$L_4 = \pi_{id}\sigma_{p\_mfgr='MFGR\#1'}$ (`lineorderfull`)

Since all the lists are precomputed and stored in an index structure such as B-tree, then the query plan can be executed as follows: L_1 and L_2 are intersected first, then the results of $L_1 \cap L_2$

[3]The many-to-one mapping is from a foreign key in the fact table to the primary key in the dimension table.

are intersected with L_3, finally the results of $L_1 \cap L_2 \cap L_3$ are intersected with L_4.

2.2.3 Graph analytics

Graph databases represent another family of advocates of inverted lists. There are usually two types of inverted lists in graph databases: adjacency lists and association lists. An adjacency list is dedicated for a vertex to maintain all neighborhood vertices connected with it. An association list is dedicated for an object (e.g., a Facebook page) to keep all relevant associations where an association is specified by a source object, destination object, and association type (e.g., `tagged-in`, `likers`) [VAB+12]. Many queries over these graphs can be answered efficiently using inverted lists. For example, finding "Restaurants in San Francisco liked by Mike's friends" reduces to finding the intersection of the adjacency list of "Mike" and the association lists of "Restaurants" and "San Francisco"; discovering common friends among a group of people transforms to computing the intersection of several adjacency lists.

2.2.4 More applications

In addition, there are many other applications that heavily use inverted lists for fast query processing. For example, data integration systems build inverted lists for q-grams to find the most similar strings [JLFL14]. Data mining systems deploy inverted lists for fast data cube operations such as slicing, dicing, rolling up and drilling down [LKH+08, LHG04]. XML databases depend on inverted lists to find twig patterns efficiently [BKS02]. Key-value stores also organize data elements falling into the same bucket (hash collision) with a chained list, which is essentially an inverted list [DSL11].

2.3 Related Work

In this section, we take a retrospective look at the major inverted list compression algorithms developed so far. Figure 2.4 shows a brief history.

As mentioned in Section 2.1, the common wisdom of a decent inverted list compression algorithm is to compute the deltas (a.k.a *d-gaps*) between two consecutive integers first and only encode the d-gaps to save space.[4] To prevent from decompressing the entire list during query processing, it organizes those d-gaps into blocks (of say 128 elements per block[5]) and builds a skip pointer per block such that only a block of data needs to be decompressed. Today, most excellent compression methods exactly follow this convention, including PforDelta [ZHNB06] (and its descendants such as NewPforDelta [ZLS08] and OptPforDelta [YDS09]), VB [CP90], GroupVB [Dea09], Simple9 [AM05], Simple16 [YDS09], and Simple8b [AM10].

Among them, PforDelta is a mature algorithm that is commonly used because it has a good tradeoff between query execution time (or decompression speed) and space overhead [ZLS08, YDS09]. The basic idea is that it compresses a block of 128 d-gaps by choosing the smallest b in the block such that a majority of elements (say 90%) can be encoded in b bits (called *regular values*). It then encodes the 128 values by allocating 128 b-bit slots, plus some extra space at the end to store the values that cannot be represented in b bits (called *exceptions*). Each exception takes 32 bits while each regular value takes b bits. In order to indicate which slots are exceptions, it uses the unused b-bit slots from the preallocated 128 b-bit slots to construct a linked list, such that the b-bit slot of one exception stores the offset to the next exception. In the case where two exceptions are more than 2^b slots apart, it adds additional forced exceptions between the two slots. Besides PforDelta, there are many variations, e.g., NewPforDelta [ZLS08] and OptPforDelta [ZLS08]. For example, OptPforDelta was designed to reduce the space overhead of

[4]Early compression algorithms (before 1990) do not follow this rule and encode each element of a list individually, e.g., Rice [RP71] and Elias gamma [Eli75]. However, they are far worse than today's compression algorithms, e.g., PforDelta, in terms of both query execution time and space overhead. Thus, we ignore them in this work.

[5]The block size represents a tradeoff between space and time and several existing works suggest 128 as the block size [ZLS08, AM10].

PforDelta but at the expense of more decompression overhead.

However, PforDelta (as well as its variations) still takes considerable time to decompress a block of data, because it usually takes three phases for decompression: (1) It needs to copy the 128 b-bit values from the slots into an integer array via bit manipulations; (2) It then walks through the linked list of exceptions and copies their values into the corresponding array slots; (3) It also goes through the integer array again to perform prefix sums to recover the original values.

Recently, there is a resurgence of EF encoding [Vig13] which is not directly based on d-gaps. Actually, EF encoding was originally proposed in 1974 [Eli74], but it did not attract too much attention until 2013 when Vigna discovered that EF encoding can be competitive with PforDelta [Vig13]. It encodes a sequence of integers using a low-bit array and a high-bit array. The low-bit array stores the lower $b = \log \frac{U}{n}$ bits of each element contiguously where U is the maximum possible element and n is the number of elements in the list. The high-bit array then stores the remaining higher bits of each element as a sequence of unary-coded d-gaps. Later on, Giuseppe and Rossano improved it by leveraging the clustering property of a list, making it outperform PforDelta for some intersection queries but not union queries [OV14]. We call it PEF (Partitioned Elias Fano) in this work. The fundamental problem of EF encoding (and its descendants including PEF) is that query processing is still not as efficient as it can be due to two reasons: (1) It needs to *sequentially* go through every *bit* in the high-bit array until a match is found, which requires many bit manipulations; (2) After that, it also needs to *sequentially* examine 2^b possible ties in the lower-bit array which can be slow if b is large.

In the literature, there were also proposals about reordering document IDs for better compression ratio, e.g., [YDS09, ZTH$^+$16]. This is orthogonal to this work and we do not consider them in this work. Besides that, this work focuses on data compression for inverted lists, which is also orthogonal to data compression in databases [LWP16].

Currently, there is also a trend of leveraging SIMD to accelerate the decompression speed of existing compression methods, such as SIMDPforDelta [LB15b]. The main idea is to reorganize data elements in a way such that a single SIMD operation processes multiple elements. However, for d-gap based compression approaches, computing prefix sums usually cannot leverage SIMD efficiently because of the intrinsic data dependencies [LB15b].

Finally, we comment on the compression approaches used in Vertica [LFV$^+$12] and Brighthouse [SWES08] that are also relevant to this work. The main idea of both approaches is to partition a list into blocks and maintain metadata for each block to support query processing. However, MILC is significantly different from them in (1) how to compress the elements within a block; (2) how to partition the list into different blocks; and (3) how to store the metadata information.

2.4 Basic compression structure

In this section, we present the basic compression structure as a starting point of MILC.

Storage structure. MILC's basic structure follows the PforDelta compression algorithm in partitioning the list L into blocks but is different in compressing the data elements within a block. It splits L into $\lceil \frac{n}{m+1} \rceil$ partitions where $(m+1)$ is the size of each partition except the last partition if n is not divisible by $(m+1)$. The choice of m will be discussed later on. The first element (i.e., the minimal value) of each block serves as a skip pointer and all the skip pointers are stored in a *metadata block*. Thus, each partition except the last one contains exactly m elements, called a *data block*. The metadata block contains $\lceil \frac{n}{m+1} \rceil$ elements (skip pointers); each element points to a data block.

MILC stores a data block as follows. Suppose the block contains the following m elements: $\{a_0, a_1 ..., a_{m-1}\}$ and β is its skip pointer. MILC stores each element a_i as the difference between a_i and the skip pointer, i.e., $a_i - \beta$, instead of $a_i - a_{i-1}$ as in PforDelta [ZHNB06]. We call it offset-based encoding instead of delta-based encoding. So the maximum difference is $(a_{m-1} - \beta)$, which can be encoded in $b = \lceil \log(a_{m-1} - \beta + 1) \rceil$ bits. Indeed, every element in the same data block is represented in b bits – unlike PforDelta, MILC does not use exceptions. Different blocks may use different number of bits to represent their values. To save space, MILC fully utilizes the 32 bits of a word by packing as many values as possible and padding the residual bits of the word (if any) with the next value if possible.

MILC stores the metadata block in the same format as PforDelta. Each entry in the metadata block contains the metadata information of a data block including the start value (32 bits), offset (32 bits), and the number of bits b (8 bits) to encode the data block.

Example. As an example, Figure 2.5 depicts the structure and storage format of $L = \{120, 200, 270, 420, 820, 860, 1060, 1160, 1220, 1340, 1800, 1980, 2160, 2400\}$ consisting of 14 elements and $m = 4$. It stores the list as follows: (1) It divides L into $\lceil \frac{14}{4+1} \rceil = 3$ partitions where each partition (except the last one) has 5 elements: $\{120, 200, 270, 420, 820\}$; $\{860, 1060, 1160, 1220, 1340\}$; $\{1800, 1980, 2160, 2400\}$. (2) It extracts the first element from each partition and puts it to the metadata block: $\{120, 860, 1800\}$. As a result, the data blocks are: $\{200, 270, 420, 820\}$ (the skip pointer is 120), $\{1060, 1160, 1220, 1340\}$ (the skip pointer is 860), and $\{1980, 2160, 2400\}$ (the skip pointer is 1800). (3) It subtracts the skip pointer from each data block. For example, for the first data block (B_0), since its skip pointer is 120, then it is stored a sequence of values by subtracting 120, i.e., $\{80, 150, 300, 700\}$. (4) It determines the smallest b in each block such that all the elements can be encoded in b bits, e.g., for block B_0, the maximum number 700 can be encoded in 10 bits, thus, it uses 10 bits to represent every element in B_0. (5) It serializes each data block as compact as possible (Figure 2.5). For example, B_0 has four 10-bit elements, but only the first three elements can be entirely packed into a 32-bit word. The fourth 10-bit element needs to span two words: the lower 2 bits are stored in the current word and the higher 8 bits are stored in a new word. Then B_1 is stored immediately after B_0 by sharing the last word in B_0 without wasting a single bit as is shown in Figure 2.5.

Next, we discuss the choice of m. If m is large, then it needs more bits to encode the data blocks because each data block spans a wide range, thus the overall space tends to be high. On

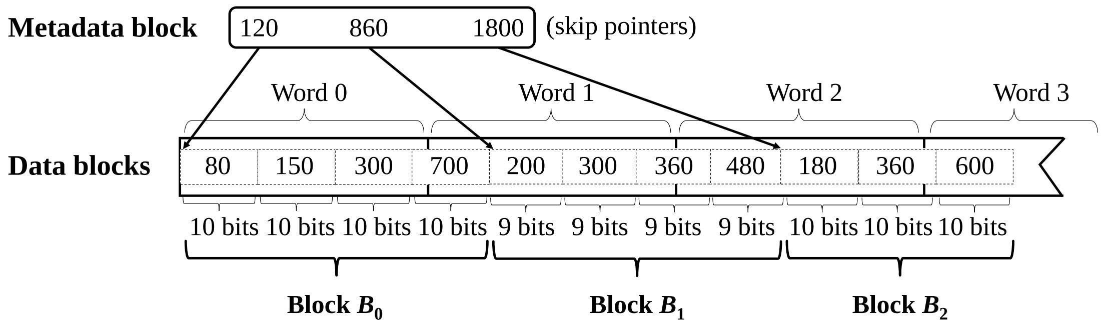


Figure 2.5: An example of storage format

the other hand, if *m* is small, then there will be more elements in the metadata block, which incurs high space overhead. Following the convention of PforDelta, we set *m* to 128 but other values are also possible. Later on in Section 2.5, we discuss the choice of *m* dynamically to minimize the overall space.

Supporting membership testing. MILC's storage structure supports membership testing over a compressed list directly without decompressing a whole block, because MILC uses fixed-bit encoding to represent each element in the block using the same number of bits while preserving the order. Let *e* be a search key, then it performs binary search in the metadata block and jumps to the potential data block and runs another binary search but using a new key $(e - \beta)$ where β is the skip pointer of the data block.

Next, we explain how to implement binary search within a data block (as it is trivial to perform binary search in the metadata block as it is uncompressed). The problem requires bit manipulations because each element takes *b* bits, which are not necessarily 8 bits – `byte` type, 16 bits – `short` type, or 32 bits – `int` type that are natively accessible by a programming language. Observe that the core of binary search is obtaining the *k*-th value because binary search needs to consistently compare the search key with the middle value within a search range. Conventionally on the integer array, it is `A[k]` to access the *k*-th value of an array `A`. But on the bit array, it requires a few bit manipulations to convert a *b*-bit value to a 32-bit value. For example in Figure 2.5, assume $b = 10$ and `A` be the compressed data blocks, then the first four values are:

```
1st:  (A[0] & 0X03FF)
2nd:  (A[0] >> 10) & 0X03FF
3rd:  (A[0] >> 20) & 0X03FF
4th:  (A[0] >> 30) | ((A[1] & 0X00FF) << 2)
```

Space overhead analysis. It is evident that the space overhead of the storage format is high compared with PforDelta. Let us roughly analyze how high it is by assuming the elements in a list are equally apart to facilitate the analysis. Let θ be the gap between two consecutive elements in a block, m be the block size (e.g., $m = 128$), p be the exception ratio (e.g., $p = 10\%$ [YDS09]), then PforDelta requires the following b bits to represent an element:

$$b = \lceil \log(\theta + 1) \rceil + 32 \times p \approx \log \theta + 3.2$$

Then for the basic compression structure, the gap now becomes $m \times \theta$. Thus, it requires the following b' bits to represent an element in the block:

$$b' = \lceil \log(m \times \theta + 1) \rceil \approx \log(128 \times \theta) = \log \theta + 7$$

That means the basic compression incurs $7 - 3.2 = 3.8$ more bits per element compared to PforDelta (but with much higher performance). Thus, in next sections, we present techniques to reduce the space overhead while keeping fast query performance.

Remark. It is worth noting that the basic structure of MILC presented in this section is based on PforDelta but with two important modifications. (1) Instead of computing deltas, MILC stores the offset values. (2) Instead of setting b such that a *majority* of elements are within 2^b, MILC determines b such that *all* elements are within 2^b, i.e., fixed-bit encoding. We also note that the basic structure of MILC is different from FOR [GRS98], which also applies fixed-bit encoding. FOR was designed for storing a page of non-sorted elements. Thus, it needs to decompress the entire page during query processing. However, MILC can support membership testing directly over compressed data without even decompressing a whole block.

2.5 Dynamic partitioning

In this section, we present a technique of dynamic partitioning to reduce the space overhead while keeping high query performance.

Why dynamic partitioning? The reason why the basic compression structure in Section 2.4 consumes much space is that it *evenly* partitions an input list into blocks (we call it *static partitioning*). So, if there are some exceptions[6] in the block, then all the elements within the block have to use the same high number of bits to represent. In other words, static partitioning is vulnerable to data skew. As an example, if a data block is $\{3, 8, 10, 15, 150\}$, then it requires 8 bits just because of 150 (an exception) while the other values actually only need 4 bits to represent. Thus, it could save a lot of space if we can *dynamically* split a list in a way that similar (or close) elements are stored together to minimize exceptions.[7]

Thus, the problem is: Given a sorted list L of integers, how to split L into blocks such that the overall space overhead is minimized? The representation of each individual block still follows the fixed-bit encoding (Section 2.4) in order to support membership testing efficiently.

Dynamic partitioning. We propose a partitioning scheme by converting the problem to a dynamic programming problem for minimizing the overall space overhead. Dynamic programming can model the space overhead of partitioning a list at different positions such that it picks up a partitioning strategy with the lowest space overhead. Let E_i be the space overhead of representing $L[0:i]$, then it splits $L[0:i]$ at the j-th ($j < i$) position: $L[0:j]$ and $L[j+1:i]$. Therefore, the space overhead of $L[0:i]$ is the summation of the space overhead of $L[0:j]$ and $L[j+1:i]$. Let $c(j,i)$ ($j \leq i$) be the space overhead of representing $L[j:i]$ and ℓ be the maximal

[6]A value is called an *exception* value if it is obviously larger than most other values in the block.

[7]Note that PforDelta does not have this issue because it uses different number of bits to represent regular elements and exceptions, but PforDelta cannot support membership testing directly on compressed data.

size of a block, then,

$$E_i = \min_{j=\max\{0,i-\ell\}}^{i-1} (E_j + c(j+1,i)) \tag{2.1}$$

Next, we analyze $c(j,i)$ used in Equation 2.1. Since the first element of the partition (i.e., $L[j]$) is stored in the metadata block as a skip pointer and the remaining values $L[j+1:i]$ are stored in a data block, then we compute the overhead of the two parts separately.

First of all, we analyze the space overhead of the skipping information (metadata block), which requires the following information per data block: (1) start value (32 bits), i.e., $L[j]$; (2) offset (32 bits) indicating where the data block starts from; (3) number of elements in the block (8 bits); (4) number of bits to encode the block (8 bits). Thus, the skipping information per data block needs $32+32+8+8=80$ bits.

Second, we consider the space overhead of the data block $L[j+1:i]$. Recall that each element in the block is stored as the difference between it and $L[j]$. Among them, the maximal gap is $L[i]-L[j]$, which requires $\lceil \log(L_i - L_j + 1) \rceil$ bits. And there are $(i-j)$ elements in the block, thus, it requires $\lceil \log(L_i - L_j + 1) \rceil \times (i-j)$ bits in total. Therefore, $c(j,i)$ can be computed as follows:

$$c(j,i) = \lceil \log(L_i - L_j + 1) \rceil \times (i-j) + 80 \tag{2.2}$$

Example. Figure 2.6 shows an example where L contains 256 elements and L=$\{4, \cdots,$ 120, 500, $\cdots$, 600, 605, $\cdots$, 900$\}$. Using the fixed-length partitioning (or static partitioning) with the block size being 128 (Figure 2.6a), then L is partitioned into two blocks and the last element in the first block is 600. For the first block, each element takes $\lceil \log(600-4+1) \rceil = 10$ bits. While the dynamic partitioning (Figure 2.6b) can determine that the first 108 elements are similar and thus group them together. As a result, each element in the first block requires only $\lceil \log(120-4+1) \rceil = 7$ bits. For each element in the second block, it takes 9 bits for both static and dynamic partitioning. As a result, static partitioning takes $10 \times 128 + 9 \times 128 = 2432$ bits while dynamic partitioning takes $7 \times 108 + 9 \times 148 = 2088$ bits.

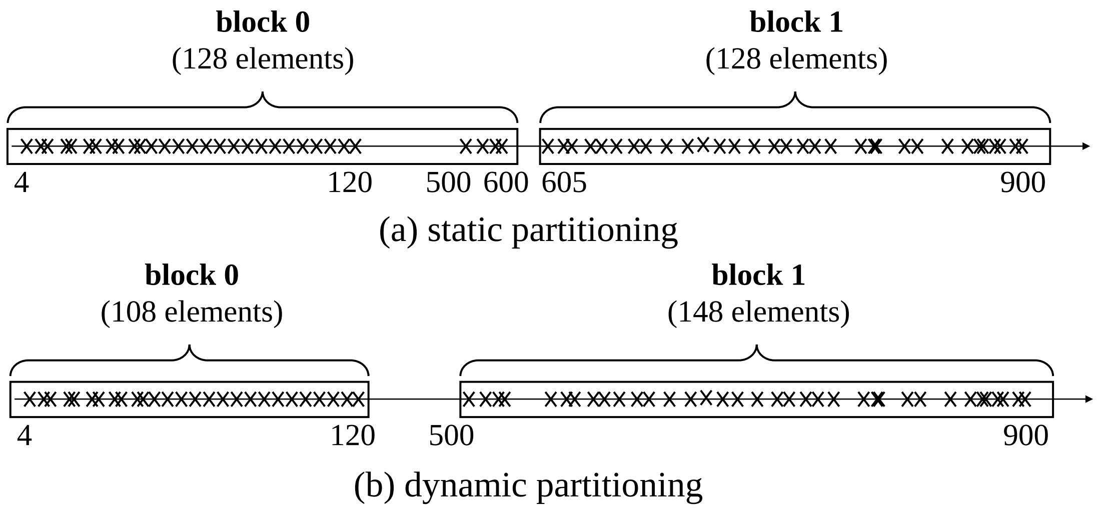


Figure 2.6: An example of dynamic partitioning

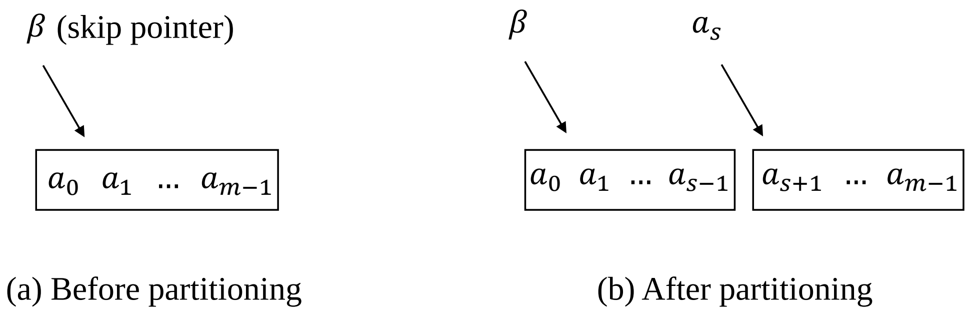


Figure 2.7: An example of illustrating the maximal size of a data block

Determining the maximal block size ℓ. The maximal group size ℓ is very important in MILC's dynamic partitioning scheme: if it is too small (say $\ell = 1$), then the optimal partitioning can be missed; if it is too large (say $\ell = |L|$), it takes too much time to find the optimal partitioning. In Theorem 1, we show that the maximal block size after dynamic partitioning is less than 2λ, where λ is the number of bits to maintain the skipping information per block (i.e., $\lambda = 80$). As a result, we set $\ell = 160$ in Equation 2.1. Note that Theorem 1 is also very useful in Section 2.6, in determining lightweight skip pointers.

Theorem 1 *A data block has at most 2λ elements after dynamic partitioning, where λ is the number of bits needed to store the skipping information per block.*

Proof 1 *We show that after partitioning, if a block still has more than 2λ elements, then we can always find a lower space cost by splitting the block into two parts, which contradicts with the optimality property achieved by dynamic programming. Without loss of generality, suppose a block contains m elements (Figure 2.7): $a_0, a_1, ..., a_{m-1}$ and β is the skip pointer of the block. We assume $m \geq 2\lambda$, next, we show that there always exists a lower cost by splitting the block into two.*

Before partitioning, the total number of bits X required is (Figure 2.7a):

$$X = \lceil \log(a_{m-1} - \beta + 1) \rceil \times m$$

Then we split the block into two parts by picking up the middle value $a[s]$ (where $s = \lfloor \frac{m}{2} \rfloor$) as a skip pointer. Therefore the partitions are $[0 : s-1]$ and $[s+1 : m-1]$. Then the total number of bits X' is (Figure 2.7b):

$$X' = \underbrace{\lceil \log(a_{s-1} - \beta + 1) \rceil \times s}_{\text{1st block}} + \underbrace{\lceil \log(a_{m-1} - a_s + 1) \rceil \times (m-1-s)}_{\text{2nd block}} + \underbrace{\lambda}_{\text{skip pointer } a_s}$$

Next we show $X' \leq X$ if $m \geq 2\lambda$ by introducing an intermediate variable Y:

$$Y = \lceil \log(a_{s-1} - \beta + 1) \rceil \times \frac{m}{2} + \lceil \log(a_{m-1} - a_s + 1) \rceil \times \frac{m}{2} + \lambda$$

Since $X' < Y$ no matter whether m is even or odd. Next, we show $Y \leq X$ if $m \geq 2\lambda$. $Y \leq X$ is equivalent to:

$$m \times (2\lceil \log(a_{m-1} - \beta + 1) \rceil - \lceil \log(a_{m-1} - a_s + 1) \rceil - \lceil \log(a_{s-1} - \beta + 1) \rceil) \geq 2\lambda$$

So, the remaining proof is to show $f = 2\lceil \log(a_{m-1} - \beta + 1) \rceil - \lceil \log(a_{m-1} - a_s + 1) \rceil -$

$\lceil \log(a_{s-1} - \beta + 1) \rceil \geq 1$ *since* $m \geq 2\lambda$. *We prove it by contradiction. Thus, we assume that* $f = 0$ *(as* $f \geq 0$ *and* f *is an integer).*

$$\lceil \log(a_{m-1} - \beta + 1) \rceil - \lceil \log(a_{m-1} - a_s + 1) \rceil = 0$$
$$\lceil \log(a_{m-1} - \beta + 1) \rceil - \lceil \log(a_{s-1} - \beta + 1) \rceil = 0$$

In order to hold:

$$\log(a_{m-1} - \beta + 1) - \log(a_{m-1} - a_s + 1) < 1 = \log 2$$
$$\log(a_{m-1} - \beta + 1) - \log(a_{s-1} - \beta + 1) < 1 = \log 2$$

That is,

$$\log \frac{a_{m-1} - \beta + 1}{a_{m-1} - a_s + 1} < \log 2$$
$$\log \frac{a_{m-1} - \beta + 1}{a_{s-1} - \beta + 1} < \log 2$$

That is,

$$a_{m-1} - \beta + 1 < 2(a_{m-1} - a_s + 1)$$
$$a_{m-1} - \beta + 1 < 2(a_{s-1} - \beta + 1)$$

Summing up the left sides gives $2(a_{m-1} - \beta + 1) < 2(a_{m-1} - a_s + 1 + a_{s-1} - \beta + 1)$*, i.e.,* $a_s - a_{s-1} < 1$*, which is a contradiction since any two consecutive numbers differ at least 1.*

Time complexity. Let n be the list size, then the time complexity of finding the optimal partitioning is $O(\ell n)$, which can be regarded as $O(n)$ since ℓ is a small constant ($\ell \leq 160$), as shown by Theorem 1.

Supporting membership testing. With dynamic partitioning, the structure supports membership testing efficiently in the same way as presented in Section 2.4, because a data block is still represented using fixed-bit encoding.

Remark. MILC is different from VSEncoding [SV10] because VSEncoding applies dynamic partitioning for PforDelta, i.e., it applies dynamic partitioning to a list of delta values. However, as we have shown in Section 2.4, this is exactly why membership testing is not supported efficiently due to the inevitable high decompression overhead. Besides that, VSEncoding chooses the maximal block size ℓ arbitrarily, which can either miss the optimal result or incur more preprocessing time. However, MILC solves the problem in an elegant way by proving in Theorem 1 that the maximal block will not exceed 2λ (which is 160).

2.6 In-block compression

In this section, we further reduce the space overhead of the compression from another angle while keeping fast query performance.

Why in-block compression? The dynamic partitioning groups similar elements to the same data block. However, all the elements in the same block have to use the number of bits based on the *maximal* element (i.e., the rightmost element) in the block in order to support fast search. But this on the other hand wastes some space for smaller elements. As an example in Figure 2.6b, after dynamic partitioning, the first block needs 7 bits to encode every element because the maximal value is 120. However, many smaller elements such as 10 and 20 do not necessarily need 7 bits. Therefore, in-block compression aims to use fewer bits to encode each element within a block to reduce the overall space overhead.

In-block compression structure. The main idea of in-block compression is to treat the elements in a data block as a micro inverted list and compress them using the approaches described in previous sections (with modifications) by splitting a block into sub-blocks. Before presenting the partitioning details, we answer the following question first: If partitioning a block into sub-blocks can reduce the overall space overhead, why previous dynamic partitioning (Section 2.5) – supposed to return a partitioning scheme with the *lowest* space cost – fails to capture such partitioning? That is because the overhead of maintaining a skip pointer within the

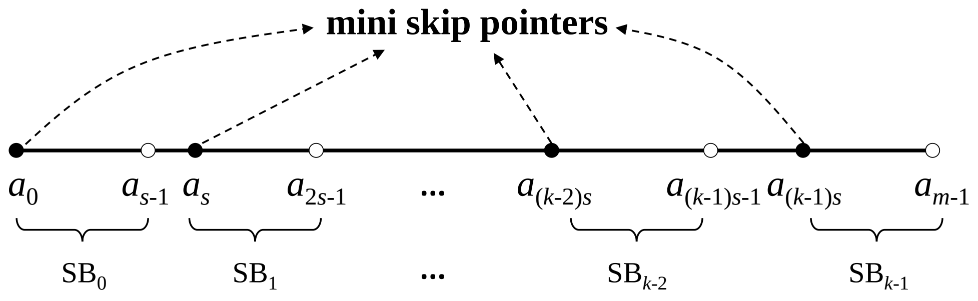


Figure 2.8: Split a data block into sub-blocks

block is *much smaller* than that outside the block. For example, it needs 80 bits to maintain a skip pointer outside the block as described in Section 2.5, but it only needs b (say 10) bits to maintain a skip pointer within the block (called a *mini* or *lightweight* skip pointer) as we explain below.

In particular, in-block compression applies the static partitioning method presented in Section 2.4 to *evenly* (except the last sub-block) split the elements into sub-blocks. Note that MILC does not apply the dynamic partitioning approach (Section 2.5) for in-block compression because that will incur more space as we explain later in the discussion part at the end of this section. Formally, suppose a block contains m elements (Figure 2.8): $\{a_0, a_1, \cdots, a_{m-1}\}$, and let k be the number of sub-blocks, then the in-block compression partitions the block into the following k sub-blocks: $\{a_0, a_1, ..., a_{s-1}\}$, $\{a_s, a_{s+1}, ..., a_{2s-1}\}$, ..., $\{a_{(k-2)s}, a_{(k-2)s+1}, ..., a_{(k-1)s-1}\}$, $\{a_{(k-1)s}, a_{(k-1)s+1}, ..., a_{m-1}\}$, where $s = \lfloor \frac{m}{k} \rfloor$. The first element of each sub-block serves as a mini skip pointer and all the mini skip pointers are stored together. Then, for every mini skip pointer in the block, it uses $\lceil \log(a_{m-1} - \beta + 1) \rceil$ bits where β is the skip pointer of the block. For every other element in the block, it uses the following number of bits b to encode:

$$b = \max\{\lceil \log(a_{s-1} - a_0 + 1) \rceil, \cdots, \lceil \log(a_{m-1} - a_{(k-1)s} + 1) \rceil\} \tag{2.3}$$

Note that without in-block compression, each element originally takes $b' = \lceil \log(a_{m-1} - \beta + 1) \rceil$ bits and $b' \geq b$.

Besides that, in-block compression needs to maintain an extra 16-bit global information for all the sub-blocks: number of bits for encoding the sub-blocks (8 bits) and number of sub-blocks k (8 bits).

Example. Figure 2.9 illustrates an example of a list L with two data blocks B_0 and B_1. Thus there are two skip pointers (stored in the format explained in Section 2.4) in the metadata block. Within each data block, it is further partitioned into sub-blocks. For example, the block B_0 consists of two sub-blocks and the block B_1 contains three sub-blocks. For all the sub-blocks within B_0, it uses the same b_0 bits to encode each element, which it originally requires b'_0 bits ($b'_0 \geq b_0$). For all the sub-blocks within B_1, it instead uses b_1 bits to represent each element. Figure 2.9 also highlights the 16-bit global information for each block B_0 and B_1.

Determining the optimal number of skip pointers. The next question is: *How many mini skip pointers to add for a data block*? We solve the problem by analyzing the relationship of the overall space overhead T_k with k in order to find the optimal k.

$$\begin{aligned} T_k = \max\{&\lceil\log(a_{s-1}-a_0+1)\rceil, \lceil\log(a_{2s-1}-a_s+1)\rceil, \cdots, \\ &\underbrace{\lceil\log(a_{m-1}-a_{(k-1)s}+1)\rceil\} \times (m-k)}_{\textit{sub-blocks}} \\ &+ \underbrace{\lceil\log(a_{m-1}-\beta+1)\rceil \times k}_{\textit{mini skip pointers}} + \underbrace{16}_{\textit{global information}} \end{aligned} \tag{2.4}$$

To find the optimal number k^*, we can enumerate all possible solutions to find which value leads to the minimal space overhead. Since we do not want a sub-block contain too few elements, say it should contain at least 4 elements. Then, we can search k from 2 to $m/4$. Thus,

$$k^* = \arg\min_{k=2}^{m/4} T_k \tag{2.5}$$

Time complexity. The time complexity of finding the optimal partitioning (off-line) is $\sum_{k=2}^{m/4} k = O(\frac{m^2}{32}) = O(800) = O(1)$ since $m \leq 160$ from Theorem 1.

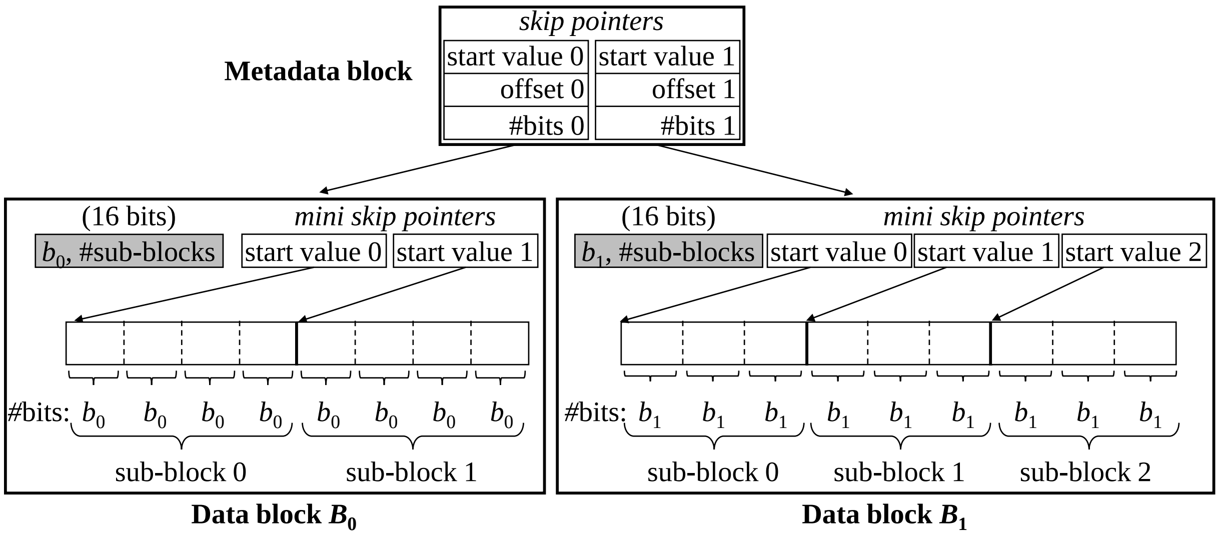


Figure 2.9: In-block compression

Supporting membership testing. It is a three-level structure where each level supports membership testing by using a revised key within a data block or a sub-block.

Discussion. We close this section by discussing two more questions: (1) Why not using dynamic partitioning to partition a block? (2) Can we further reduce the space overhead by partitioning a sub-block into sub-sub-blocks?

For the first question, it needs to maintain more skipping information to dynamically partition a data block into sub-blocks. The skipping information should at least contain: start value ($\lceil \log(a_{m-1} - \beta + 1) \rceil$ bits where β is the skip pointer of the block), number of elements (8 bits), offset (16 bits), and number of bits used to encode a sub-block (8 bits). Thus, a skip pointer needs ($\lceil \log(a_{m-1} - \beta + 1) \rceil + 32$) bits, which is much higher since the current solution only needs $\lceil \log(a_{m-1} - \beta + 1) \rceil$ bits. On the other hand, it may not save too much space overhead in the sub-blocks because all data elements in a data block are very similar.

For the second question, it may not reduce the overall space anymore. That is because there is an extra space overhead associated with each split, i.e., 16 bits as highlighted in Figure 2.9. However, when there are few elements and each element uses very few bits, it is extremely difficult to save 16 bits anymore with further partitioning because in-block compression works if and only if $(m-k)(b'-b) \geq 16$. For example, suppose a block contains 8 elements: $\{10, 20, 30, 40, 50, 60, 70, 80\}$. Without partitioning, it takes $7 \times 8 = 56$ bits. With two partitions, i.e., 10 and 50 are promoted as mini skip pointers (taking $7 \times 2 = 14$ bits). The two sub-blocks become (after subtracting the skip pointer): $\{10, 20, 30\}$ and $\{10, 20, 30\}$. They take $3 \times 5 + 3 \times 5 = 30$ bits. Together with the mini skip pointers, the overall space overhead is $30 + 14 = 44$ bits, saving $56 - 44 = 12$ bits, which is less than 16 bits. Thus, we do not recommend further partitioning anymore.

2.7 Cache-conscious compression

In this section, we further improve the layout of MILC such that it is more friendly to CPU cache lines for minimizing cache misses during membership testing.

What is cache-aware and why? Modern CPUs dedicate several layers of very fast caches (L1/L2/L3 cache) to alleviate the growing disparity between CPU clock speed and memory latency (a.k.a *memory wall*). Whenever a CPU instruction encounters a memory access, it first checks whether the accessed data resides in the caches. If yes, it accesses the data from the caches directly. Otherwise, a *cache line* (typically 64 bytes) of data is loaded from main memory to the caches. This will potentially evict other cache lines that are in the caches. Thus, the goal of the cache-conscious design is to reduce cache misses by ensuring that a cache line brought from memory is fully utilized before it is being evicted.

Cache-aware design. We explain how to turn the compression structure presented in the previous section (Section 2.6) into a cache-aware structure. We classify the membership testing into two categories: within a metadata block (storing uncompressed skip pointers) and within a data block (storing compressed data).

For the membership testing within a metadata block, it is essentially the conventional binary search over an array. Previous studies have investigated it [RR99, KCS$^+$10]. The main idea is to organize the elements into a B-tree structure with the node size being a CPU cache line (64 bytes). Note that the B-tree is materialized as an array using a level-order traversal manner without explicit storing any tree pointers, for saving space overhead. Thus, search can be efficiently executed by traversing the B-tree. However, there are two unique challenges in incorporating them into a fully functional compression structure: (1) The number of elements (i.e., skip pointers) may not form a perfect tree[8] but most previous studies made such an assumption to save space overhead by not explicitly storing the tree pointers. We observe that only a collection of $17^h - 1$ elements can be converted to a h-level perfect tree. That is because a cache line contains $64/4 = 16$ elements (i.e., 17 children), then the total number of elements in a h-level perfect tree is:

$$\underbrace{16}_{\text{level 1}} + \underbrace{16 \times 17}_{\text{level 2}} + \underbrace{16 \times 17^2}_{\text{level 3}} + \cdots + \underbrace{16 \times 17^{h-1}}_{\text{level } h} = 17^h - 1$$

However, there are many inverted lists and each inverted list has a different number of skip pointers that may not be $17^h - 1$. (2) Another unique challenge is how to find the corresponding data block after a skip pointer is located in the metadata block. This was not an issue for the non-cache-aware structure because the skip pointers and data blocks are stored in the same order. That is, a skip pointer and its data block have the same index number. However, if the skip pointers are stored in a cache-aware manner, the index numbers become completely different.

To solve the first challenge, we convert an array of sorted elements (i.e., skip pointers, non-cache-aware) to a complete tree instead of a perfect tree. A h-level complete tree [CLRS09] ensures that (1) only the last level is not full and all the elements in the last level are stored from left to right; (2) if the last level is removed, then it becomes a $(h-1)$-level perfect tree. We are not aware of any previous cache-aware designs having solved the problem, probably because they simply assumed the number of elements can form a perfect tree. But Schlegel et al. presented a solution in the SIMD area [SGL09] that can be extended to cache-aware designs.

[8]A perfect tree (i.e., balanced and full) has three requirements [CLRS09]: (1) every node has precisely k entries where k is the fanout; (2) every intermediate node has exactly $k+1$ children nodes; (3) every leaf node has the same depth.

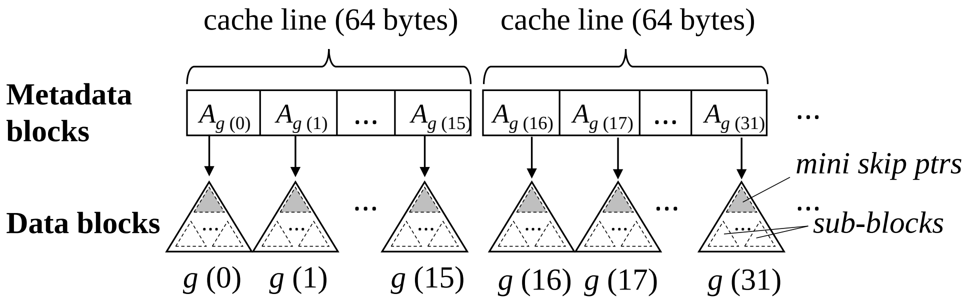


Figure 2.10: Cache-aware layout

The main idea is to determine for any element from the old non-cache-aware array the position in the new cache-aware array by developing a one-to-one mapping. Formally, let n be the number of elements (skip pointers), k be the number of elements that a cache line can accommodate ($k = 16$), H be the number of levels ($H = \lceil \log_k(n+1) \rceil$), i be the element position in the old non-cache-aware array, and $g_n(i)$ be the position in the new cache-aware array. Then,

$$g_n(i) = \begin{cases} f_H(i) & \text{if } i \leq f_H^*(n) \\ f_{H-1}(i - o_H^*(n) - 1) & \text{otherwise} \end{cases} \tag{2.6}$$

We omit the explanations and proofs of these equations due to space constraints and refer interested readers to [SGL09].

Next, we discuss how to tackle the second challenge. A simple solution is to compute a reverse mapping from the (new) position in the cache-aware array to the (old) position in the non-cache-aware array. However, that will take considerable time as the mapping has to be computed on the fly. MILC's solution is to change the storage of the data blocks such that they have the same order with their corresponding skip pointers. Figure 2.10 illustrates the design. The data blocks (as well as the skip pointers) are stored in the order of $g(0)$, $g(1)$, $g(2)$ and so on.

Next, we comment on the second type of membership testing that happens within a data block. It turns out that the existing design presented in the previous section is actually cache-aware. That is because each data block is organized as a two-level tree structure with the mini skip pointers being stored as the root node while the sub-blocks being stored as children nodes.

Supporting membership testing. The membership testing is executed efficiently by traversing an array of cache-aware skip pointers in the metadata block. Then it goes to the right data block to continue membership testing by using a revised key.

2.8 SIMD acceleration

In this section, we discuss how to further improve the performance of MILC by leveraging the SIMD instructions.

What is SIMD-aware and why? A SIMD instruction operates on a s-bit register where s depends on different processors. Typically, s is 128, but more recent processors also support 256-bit or 512-bit SIMD operations. In this work, we use 128-bit SIMD instructions for a fair comparison with existing works [LB15b]. The benefit of using SIMD instructions is to improve performance by processing multiple elements at a time.

Note that although compilers can automatically optimize some simple code with SIMD instructions, the optimizations are very limited, e.g., only for simple loops [Int12]. Thus, to fully exploit the SIMD instructions, we need to explicitly design a new storage structure that are amenable to SIMD.

SIMD-aware design. Recall in the previous discussions, the framework of MILC has two categories of blocks: metadata block and data blocks. The data elements in the metadata block are uncompressed while the data elements in the data blocks are compressed. Next, we discuss how to organize the data elements in both types of blocks into a SIMD-efficient structure.

For the elements (skip pointers) in the metadata block, they are stored as a contiguous sequence of cache lines. We store the data elements within the same cache line following the k-ary approach [SGL09]. In this way, a SIMD operation can be applied to quickly find out which child node to visit [SGL09].

For the elements in the data blocks, MILC basically follows the design in the previous section to support efficient membership testing while keeping a low space overhead. That is because a SIMD operation interacts with a s-bit SIMD register as a vector of *banks*, where a bank is a continuous section of b bits. For example, in SSE (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions), b is 8 (`byte` type), 16 (`short` type), or 32 (`int` type). However, for inverted list compression, each element can be encoded in arbitrary number of bits b (not necessarily 8, 16, or 32 bits). Note that, there are other options if the space overhead is not a problem, e.g., by rounding up b to 8, 16, and 32.

Supporting membership testing. Given a search key, the first level of uncompressed data (skip pointers) will be accessed in a SIMD-aware manner following [SGL09] to find out which block that potentially contains the key; then perform membership testing within the block.

2.9 Experiments

In this section, we experimentally compare MILC against state-of-the-art inverted list compression schemes in terms of membership testing time and space overhead.

2.9.1 Experimental setting

Experimental platform. We conduct experiments on a commodity machine (Intel i7-4770 quad-core 3.40 GHz CPU, 64GB DRAM) with Ubuntu 14.04 installed. The CPU's L1, L2, and L3 cache sizes are 32KB, 256KB, and 8MB. The CPU is based on Haswell microarchitecture which supports AVX2 instruction set. We use mavx2 optimization flag for the SIMD optimization. We implement MILC in C++ and compile the code using GCC 4.4.7 with O3 enabled.

Datasets. In this work, we use the datasets from information retrieval, databases, and graph analytics.

(1) *Web data*. It is a collection of 41 million Web documents (around 300GB) crawled in 2012.[9] It is a standard benchmark in the information retrieval community. We parse the documents and build inverted lists for each term. The query log contains 1000 real queries randomly selected from the TREC[10] 2005 and 2006 (efficiency track).

(2) *DB data*. It is a star schema benchmark (SSB),[11] which includes one fact table (LINEORDER) and four dimension tables (CUSTOMER, SUPPLIER, PART, and DATE). We set the scale factor as 10 so the number of rows in the fact table is around 60 million. We use the query described in Section 2.2.2 for evaluation. Each list is allocated for a predicate on a logical huge table as described in Section 2.2.2. The list sizes are 11916634, 12028431, 9098421, and 11997098. Note that MILC is also applicable to other SSB queries. We use one single query for evaluation due to space limitation.

(3) *Graph data*. It is the twitter dataset crawled in 2009, which consists of 52,579,682 vertices and 1,963,263,821 edges. The data is widely used in graph analytics. Each list is an adjacency list of a vertex. We follow the methodology in [CBB$^+$13] to evaluate the following query: "find out the common friends between a group of people". Note that other queries could also be applied. The list sizes are 423640, 507777, 526292, and 779957, respectively.

Competitors. We compare MILC with a wide range of recent compression approaches: Uncompressed, VB [TH72], PforDelta [ZHNB06], OptPforDelta [ZLS08], NewPforDelta [ZLS08], Simple8b [AM10], Simple9 [AM05], Simple16 [YDS09], GroupVB (a.k.a VarintGB) [Dea09], SIMDBP128 [LB15b], SIMDPforDelta [LB15b], PEF [OV14]. We implement PEF from scratch and implement the other above-mentioned compression algorithms based on the source codes provided from [LB15b]. For the uncompressed lists, we use conventional binary search. Note that we do not compare with some obviously low-performance encodings, such as Golomb, Rice, and Elias gamma. We also ignore those general purpose encoding schemes such as Snappy, LZ,

[9] http://www.lemurproject.org/clueweb12.php
[10] http://trec.nist.gov/
[11] http://www.cs.umb.edu/ poneil/StarSchemaB.PDF

LZ4, LZO, or gzip, because they are much slower than PforDelta [LB15b].

Evaluation methodology. In this work, we mainly use the following measurements for evaluation.

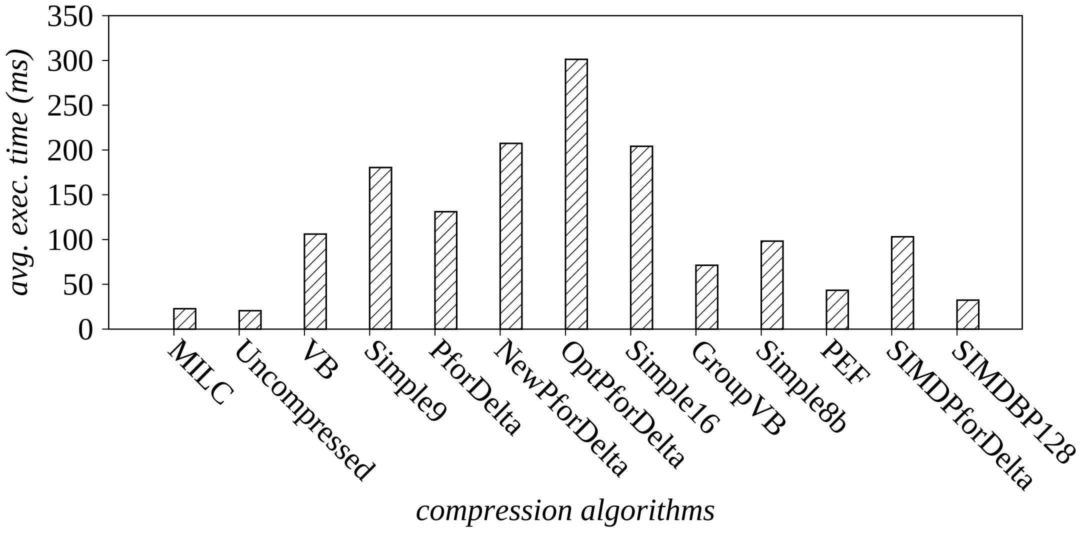
(1) *Execution time*. For each compression algorithm, we measure how fast it supports membership testing. In particular, we report the intersection time of each query since list intersection is essentially a series of membership testing operations to consistently find whether an element appears in a list. We use an SvS [CM10a] that has been widely used in practice including Lucene. Assuming there are k lists L_1, L_2, $\cdots$, L_k ($|L_1| \leq |L_2| \leq \cdots |L_k|$) that are compressed. SvS decompresses the shortest list L_1 first. Then for each element $e \in L_1$, SvS checks whether e appears in L_2 (i.e., membership testing). Note that SvS does not need to decompress the entire L_2 due to skip pointers and it only needs to decompress a block of data that potentially contains e for membership testing. Then the results of L_1 and L_2 will be intersected with L_3 and the process continues until L_k.

(2) *Space overhead*. We also measure the space overhead that a compression algorithm takes.

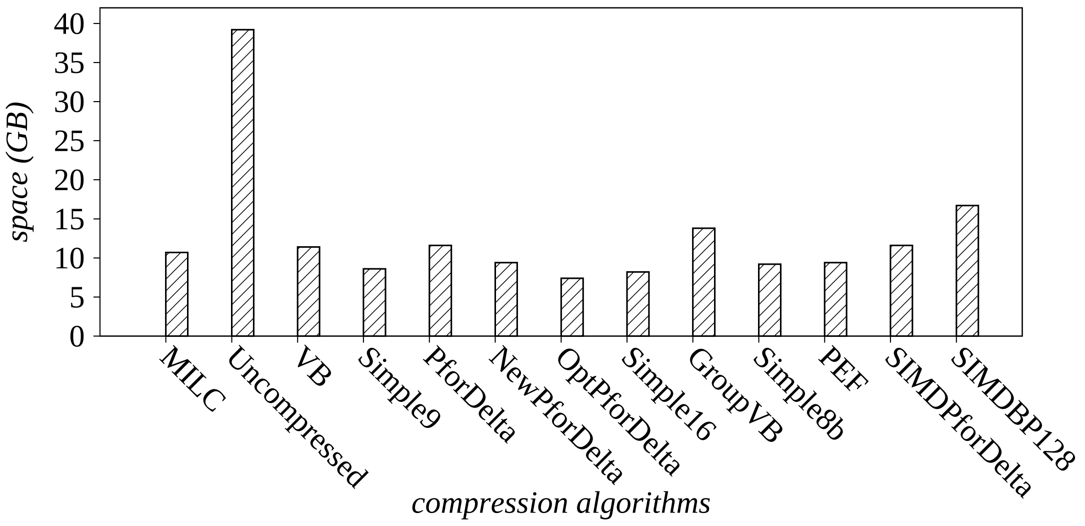
2.9.2 Experimental results

In this experiment, we compare MILC with existing compression approaches on the two datasets. Note that MILC incorporates all the optimizations presented in Section 2.4, Section 2.5, Section 2.6, Section 2.7, and Section 2.8.

Figure 2.11 compares the average execution time and space overhead of on Web data. The execution time is measured by the average time (ms) of running those queries. Figure 2.11 shows that, (1) Compared with uncompressed lists, MILC achieves almost the same query performance but with a 3.7× lower space overhead. The high performance is because MILC relies on fixed-bit encoding (instead of d-gaps as in most other compression algorithms) to support membership testing directly over compressed lists without decompressing even a whole block. MILC also relies on efficient architectural-aware data organizations such as cache-aware and SIMD-aware



(a) Execution time



(b) Space overhead

Figure 2.11: Comparing against existing compression approaches on Web data

optimizations to achieve high query performance. The small space overhead is that MILC applies dynamic partitioning to store similar elements together. MILC also leverages the novel in-block compression technique to further reduce the space overhead. Figure 2.11 shows that, query processing on compressed lists can be (sometimes) executed as fast as that on uncompressed lists while keeping a low space overhead at the same time. (2) Compared with PforDelta, MILC is 5.75$\times$ faster in execution time and 7.8% less in space overhead. The execution time saving is because PforDelta needs to decompress a whole block of data during membership testing because it is based on d-gaps but MILC does not need to do so. The space overhead saving is because PforDelta partitions a list statically while MILC partitions a list dynamically. And also, MILC applies in-block compression to further reduce the space overhead. (3) Compared with the variants of PforDelta, e.g., OptPforDelta and NewPforDelta, MILC has many advantages too. For example, MILC is 13.2$\times$ faster (in execution time) than OptPforDelta while only incurring 44% more space. MILC is 9.1$\times$ faster than NewPforDelta while only taking 13.8% more space. (4) Compared with Simple9 [AM05], Simple16 [YDS09], and Simple8b [AM10], MILC is 7.9$\times$, 9.0$\times$, and 4.3$\times$ faster but only consumes 24.4%, 30.4%, and 16.3% more space. (5) Compared with PEF [OV14], MILC is 1.9$\times$ faster in execution time, while only consuming 13.8% more space. (5) Compared with the other compression algorithms, e.g., VB, GroupVB, SIMDBP128, and SIMDPforDelta, MILC runs 1.4$\times$ to 4.6$\times$ faster in query processing and takes 6.5% to 56.1% less space also.

Figure 2.12 shows the results of evaluating MILC on DB data. It shows that, (1) Compared with uncompressed lists, MILC needs similar execution time but only requires 3.7$\times$ lower space overhead. (2) Compared with PforDelta, MILC is 5.65$\times$ faster in execution time but MILC takes less space overhead. (3) Compared with OptPforDelta, MILC is 10.3$\times$ faster in execution time but OptPforDelta only takes 38% less space overhead. (4) Compared with NewPforDelta, MILC is 8.2$\times$ faster in execution time but NewPforDelta only incurs 16% less space overhead. (5) Compared with Simple9 [AM05], Simple16 [YDS09], and Simple8b [AM10], MILC is 6.7$\times$, 8.1×, and 3.9× faster but only consumes 17.1%, 14.6%, and 14.3% more space. (6) Compared with PEF [OV14], MILC is 1.7× faster in execution time, while only consuming 22.7% more space. (7) Compared with the other compression algorithms, e.g., VB, GroupVB, SIMDBP128, and SIMDPforDelta, MILC is 1.22× to 3.5× faster in query processing and takes 2.6% to 68.5% less space also.

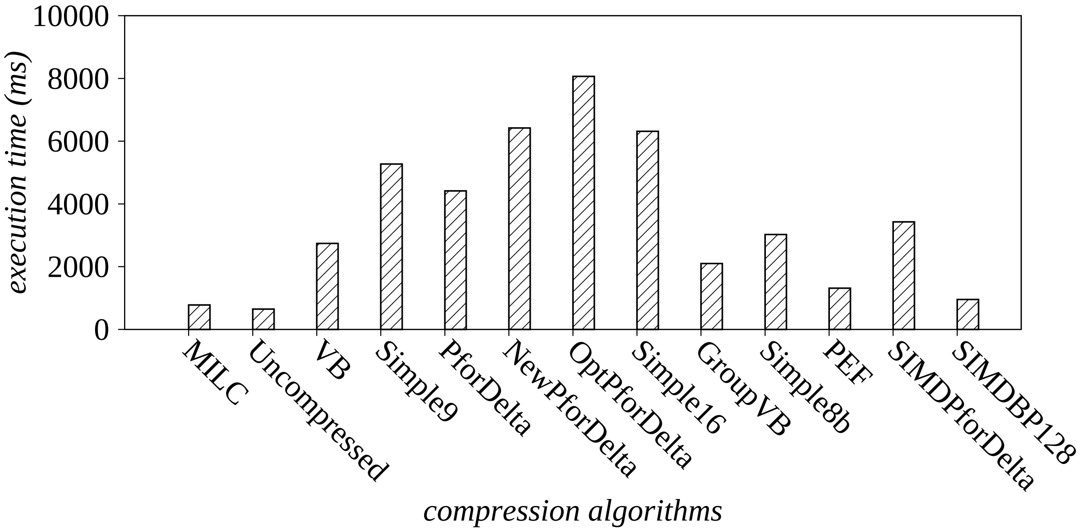
Figure 2.13 shows the results of evaluating MILC on Graph data. We omit the descriptions of the results since they are largely similar to Figure 2.11 and Figure 2.12.

Overall, MILC represents the best tradeoff for inverted list compression especially in main memory databases compared among a spectrum of 12 existing compression algorithms.

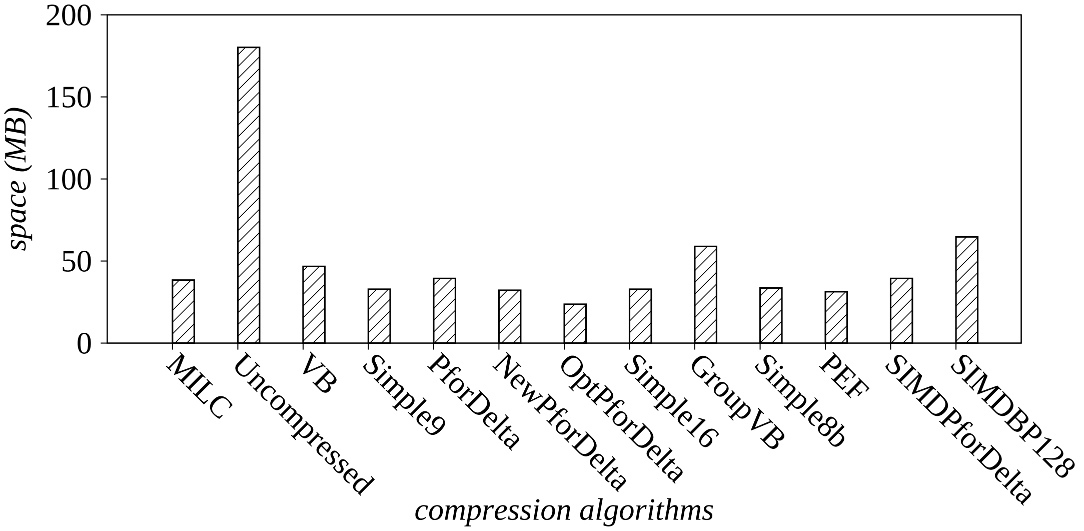
2.10 Chapter summary

In this chapter, we proposed a new compression approach MILC for encoding inverted lists in main memory. MILC is the first compression scheme that achieves similar query performance compared with uncompressed lists. Also, MILC is significantly faster than existing compression algorithms while keeping low space overhead.

Chapter 2 contains material from “MILC: Inverted List Compression in Memory” by Jianguo Wang, Chunbin Lin, Ruining He, Moojin Chae, Yannis Papakonstantinou, and Steven Swanson, which appeared in Proceedings of Very Large Data Bases Conference (VLDB), pages 853–864, 2017. The dissertation author was the primary investigator of the paper.

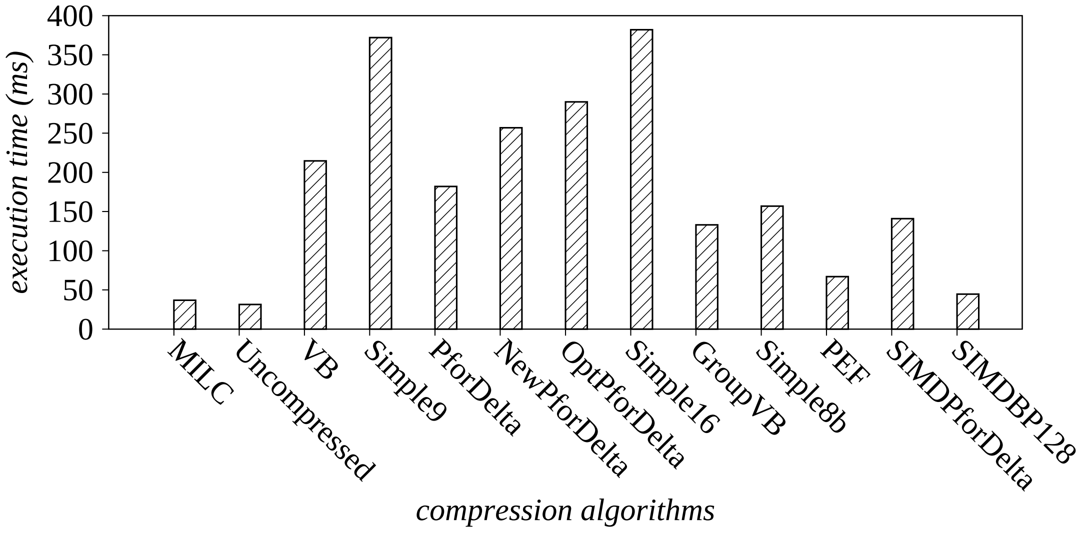


(a) Execution time

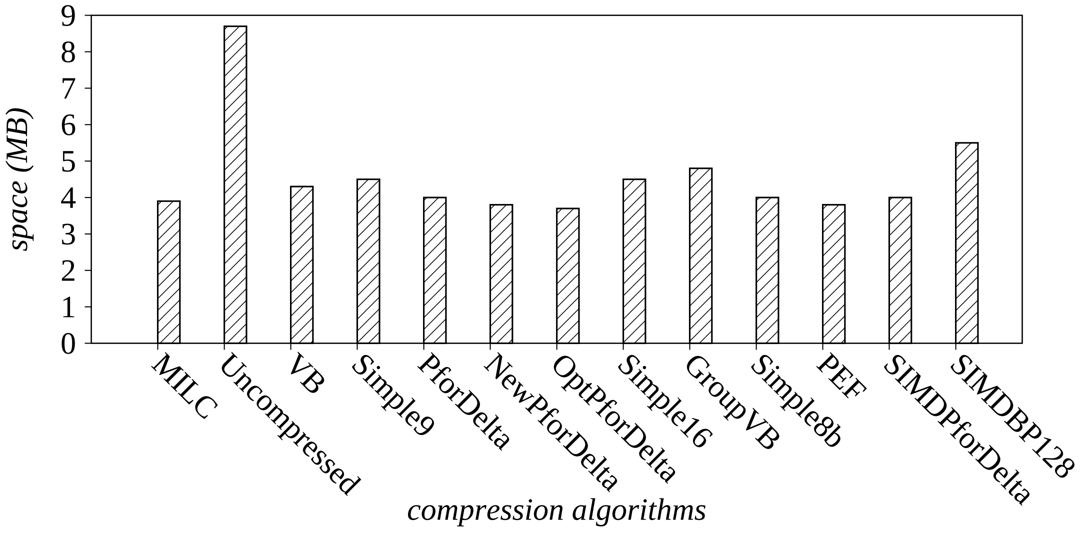


(b) Space overhead

Figure 2.12: Comparing against existing compression approaches on DB data



(a) Execution time



(b) Space overhead

Figure 2.13: Comparing against existing compression approaches on Graph data

Chapter 3

Evaluating List Intersection on SSDs for Parallel I/O Skipping

3.1 Introduction

List intersection is very important because it is heavily used in many applications such as information retrieval and database querying. For instance, finding documents that contain all the query terms needs the intersection of several inverted lists; evaluating conjunctive predicates of analytical queries in databases requires the intersection of several columns. In this work, we focus on disk-resident systems where all the lists cannot fit in main memory and therefore, at least partially, need to be stored on secondary storage. Since HDDs (hard disk drives) have been dominating the storage market for decades, existing disk-based intersection algorithms were mainly optimized for HDDs [DLOM00, BYSC05, CM10b, DK11]. Those algorithms aim to minimize the number of expensive random reads. They generally carry out list intersection using a two-phase process. (1) Read each list from disk to memory in its entirety, so that each list requires only one random read. This stage usually requires a single thread to access disk because an HDD has only one magnetic disk head to serve one I/O request simultaneously, thus, using multiple threads to read a list does not improve performance [GBN14]. We refer to it as "list-at-a-time single-threaded" I/O access pattern. (2) Perform the intersection in memory, which can use multiple threads to exploit multi-core CPUs. The two-phase paradigm is a perfect fit for HDDs, because random reads on HDDs are one to two orders of magnitude more expensive than sequential reads [THS+09, AS10], due to the extremely slow disk seeks.

Today, solid state drives (SSDs) have become an alternative secondary storage solution to HDDs in the storage market. Compared to HDDs, SSDs have many advantages such as low I/O latency, high I/O throughput, and low energy consumption [GCC+09]. As a result, SSDs are deployed in many large-scale systems, e.g., Oracle's 11g database, Amazon's Redshift database and Google's and Baidu's search engines [Ma10]. In particular, Baidu, the largest search engine in China, has changed HDDs to SSDs completely in its storage system [Ma10, WLY+13].

The landscape shifting from HDDs to SSDs has raised an interesting research question: *What is the impact of SSDs to list intersection algorithms*? There are two notable properties of SSDs when compared to HDDs that can change the algorithmic design decisions. (1) The first one is that random reads are fast enough to be comparable with sequential reads. On modern SSDs, random reads are only $1\times \sim 2\times$ slower than sequential reads [THS+09]. For instance, for the Samsung SSD[1] used in the experiments, the reported theoretical sequential read bandwidth is 550MB/s while the random read bandwidth is 400MB/s; thus, the gap is $1.375\times$. Moreover, the gap is even smaller if the I/O access pattern is not purely random because of the internal cache inside the SSD. (2) The second interesting property is that an SSD supports parallel I/O because it is manufactured to incorporate multiple flash channels [RPK+11]. Thus, it can serve multiple I/O requests simultaneously. In contrast, an HDD has only one disk head such that it can only serve a single I/O request at the same time.

With the two new properties mentioned above, we speculate existing HDD-centric list intersection algorithms might not work well on SSDs and we rethink SSD-aware intersection

[1]http://www.samsung.com/semiconductor/minisite/ssd/product/consumer/850pro.html

algorithms by explicitly leveraging the unique characteristics. A decent SSD-optimized intersection algorithm shall follow "page-at-a-time multi-threaded" I/O access pattern (instead of the HDD's "list-at-a-time single-threaded" access pattern):

(1) **Page-at-a-time**: access a list page by page for reducing unnecessary pages that do not contain intersection results to leverage the SSD's fast random reads. In this way, the optimization goal should be minimizing the number of pages accessed by skipping as many as possible of the pages that do not contain the intersection results. For example, all the white-color pages in Figure 3.1 can be skipped. Note that, skipping does not work well on HDDs because it introduces many expensive random reads that do not pay off (as we show later in experiments) since HDDs are slow in random reads.

(2) **Multi-threaded**: issue multiple page-sized I/O requests at the same time via multiple threads. We observe that using a single thread, the page-at-a-time I/O access pattern underutilizes the SSD's bandwidth seriously because there is only one page read at a time such that only one flash channel is active.[2] Meaning that even if the total amount of data accessed is reduced, the actual execution time may not be reduced as expected. Fortunately, modern SSDs support concurrent I/O requests with multiple flash channels. Thus, we shall explicitly use multiple threads in order to saturate the SSD's bandwidth if the underlying lists are accessed page-by-page. Note that it does not make sense to issue multiple threads to read a list on an HDD because it has only one disk head. Table 3.1 shows a summary of whether the disk bandwidth can be saturated using different I/O access patterns.

Following "page-at-a-time multi-threaded" I/O access pattern, then how to design efficient intersection algorithms for SSDs? There are many algorithmic design choices in terms of how to skip unnecessary pages efficiently and issue I/Os in a parallel way. In this work, we tune five

[2]Note that, the list-at-a-time I/O pattern (even single thread) on the SSD can saturate the I/O bandwidth because a list usually has many pages that can be stripped over multiple flash channels of the SSD (a.k.a internal parallelism), but it cannot skip unnecessary pages.

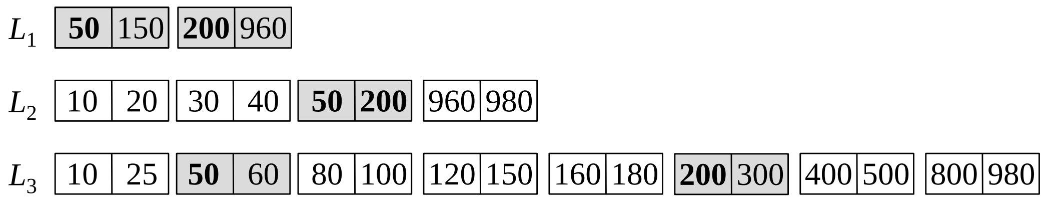


Figure 3.1: An example of list intersection on an SSD

existing in-memory intersection algorithms to be SSD-conscious based on parallel I/O skipping (via skip pointers and bloom filters), and experimentally evaluate them on both real data and synthetic data.

In particular, this work aims to answer the following questions experimentally for SSD-aware list intersection:

- What is the impact of skip pointers and how to use skip pointers efficiently?
- What is the impact of bloom filters and how to use bloom filters efficiently?
- What is the impact of parallelism?
- What is the impact of cache?
- What is the impact of compression?
- What is the impact of other important parameters including list size, list size ratio, intersection ratio, and the number of lists?

Contribution. In this work, we first extend five in-memory intersection algorithms to be SSD-aware with the idea of parallel I/O skipping, and then experimentally evaluate them on synthetic datasets and real datasets. We study the impact of skip pointers, bloom filters, parallelism, cache, compression, list size, list size ratio, intersection ratio, and the number of lists to the performance. The experimental results shed lights on how to design efficient SSD-centric list intersection algorithms.

Table 3.1: Saturating bandwidth with different I/O patterns on HDD and SSD

	Single-thread	**Multi-thread**
Page-at-a-time	no	no
List-at-a-time	yes	yes

(a) on HDD

	Single-thread	**Multi-thread**
Page-at-a-time	no	yes
List-at-a-time	yes	yes

(b) on SSD

3.2 Applications

In this section, we elaborate more on the applications of list intersection.

3.2.1 Information retrieval

Information retrieval (IR) is a killer application of list intersection to answer user queries with multiple terms [MRS08]. IR systems store an inverted list for each term all the documents that contain the term. Taking the intersection of the lists for a set of query terms identifies those documents that contains all the terms.

3.2.2 Database querying

List intersection is also useful in evaluating not only plain conjunction queries but also joins. For instance, in the star schema benchmark, one can issue the following query to compute the profits for each city in 1998 for products consumed by customers from "`UNITED STATES`" with brand "`MFGR#2221`" and suppliers of "`AMERICA`":

```
      SELECT S.city, SUM(O.revenue-O.supplycost) as profit
FROM   lineorder O, customer C, supplier S, date D, part P
WHERE  O.custkey = C.custkey AND O.orderdate = D.date
       AND O.partkey = P.partkey
       AND O.suppkey = S.suppkey
       AND C.nation = 'UNITED STATES'
       AND S.region = 'AMERICA'
       AND P.brand1 = 'MFGR#2221'
       AND D.year = 1998
       GROUP BY S.city
```

In order to answer queries efficiently, most databases precompute a list of matching row IDs for each predicate [RQH$^+$07], e.g., L_1 = {`O.rid | P.brand1 = 'MFGR#2221'`} and L_2 = {`O.rid | D.year = 1998`}. Then, a query can be executed efficiently by intersecting the precomputed lists of row IDs for all predicates involved in the query.

3.3 Related work

In this section, we describe the previous work: (1) list intersection on HDDs (Section 3.3.1); (2) list intersection in main memory (Section 3.3.2); (3) SSD-optimized design in IR and database querying (Section 3.3.3).

3.3.1 List Intersection on HDDs

To reduce the I/O cost for list intersection on HDDs, earlier work tried to avoid reading all the lists in their entirety [ZM06]. A solution mentioned in [ZM06] is to load the shortest list first, and intersect all the other lists in ascending order of their sizes. The algorithm terminates once the current intersection results are empty. In this way, the longer lists can be skipped. However, the algorithm ends up reading all the lists if the intersection results are not empty. Actually, for any list L, the algorithm either reads L entirely or does not read L at all. It does not skip any blocks in a single list (the goal of this work). That is because skipping introduces many random reads which are very expensive on HDDs.

Thus, if the lists are stored on HDDs, the best solution is to load the lists to memory in their entirety and carry out list intersection in main memory.

3.3.2 List intersection in memory

There is rich literature on the in-memory list intersection problem [CM10b, DLOM01, DLOM00, BK02, DK11], partially because all the lists of a query have to be loaded to memory first if they are stored on HDDs (as explained in Section 3.3.1), and partially because the entire inverted index is small enough to fit in memory for some applications.

We classify the existing algorithms into four categories: comparison-based (Section 3.3.2), hash-based (Section 3.3.2), partition-based (Section 3.3.2), and bitmap-based (Section 3.3.2). Table 3.2 provides an overview. We describe the existing algorithms using k sorted lists L_1, L_2, $\cdots$, L_k ($|L_1| \leq |L_2| \leq \cdots \leq |L_k|$).

Comparison-based intersection algorithms

The first comparison-based intersection algorithm is the sort-merge algorithm [HL72], which would access the entire lists if extended to SSDs.

To skip unnecessary comparisons, the SL (short-long) algorithm [CM10b] was proposed. For each element $e \in L_1$, SL checks whether e appears in all the other lists. If yes, e is a result. The presence of e on L_i can be obtained by invoking Member(L_i, e), which returns true if L_i contains e. In SL, Member(L_i, e) can be implemented using binary search, skip lists [Pug90] or treaps [BRM98]. We extend SL to SSDs, so that it skips pages.

The SvS algorithm [DLOM01] is a comparison-based algorithm, which is a variant of SL. SvS starts from the shortest list and intersects the other lists in ascending order of their sizes. In other words, L_1 and L_2 are intersected first, then the results of $L_1 \cap L_2$ are intersected with L_3. The process continues until L_k. SvS also relies on Member(L, e) to test whether e is in L. Thus, basically, SvS and SL are the same except that SvS needs to maintain a result buffer if the number

Table 3.2: A list of in-memory list intersection algorithms and the extensions to SSDs

Categories	In-memory intersection	Extensions to SSDs
Comparison-based	Sort-merge [HL72]	Load the entire lists
	SL [CM10b]	**Can skip pages**
	SvS [CM10b, DLOM01]	**Can skip pages**
	Zig-Zag [DLOM00, BK02]	**Can skip pages**
Hash-based	**Simple hash [Knu73]**	**Can skip pages**
	BPP [BPP07]	Load the entire lists
Partition-based	Divide & conquer [BYSC05]	Load the entire lists
	Group [DK11]	Load the entire lists
Bitmap-based	Bitmap [CM10b]	Load the entire bitmaps

of lists intersected exceeds two.

Another algorithm, called Zig-Zag (ZZ, a.k.a Adaptive), uses Successor(L,e) for skipping [DLOM00, BK02]. Successor(L,e) returns the smallest element in L that is greater than or equal to e. Every time an eliminator e is selected (initially e is the first element of L_1), then e is probed against the other lists in a round-robin fashion. For the current list L_i, ZZ checks whether e is the same as Successor(L_i,e). If yes, the occurrence counter of e is increased (e is a result if the counter reaches k). Otherwise, it updates e to Successor(L,e). ZZ terminates once e is invalid.

If L is compressed (e.g., using PforDelta [ZHNB06]), neither Member(L,e) nor Successor(L,e) can be implemented efficiently. To allow skipping, a general technique is to build an auxiliary data structure (e.g., skip list) over L [MZ96, ST07]. The original list L is then split into segments. The auxiliary data structure maintains for each segment the minimum/maximum element (uncompressed). Thus, Member(L,e) and Successor(L,e) can be implemented by routing to the desired segment and decompressing it individually.

Hash-based intersection algorithms

The naive hash-based intersection algorithm [Knu73] builds a hash table to implement Member(L,e). If extended to SSDs, we could build an external-memory hash table to support skipping. However, the hash table takes too much space. Moreover, it resembles SL [CM10b] when extended to SSDs because both of them route a search element to a page that potentially contains the element. Bille et al. suggested another way of using hash [BPP07]. Unfortunately, if extended to SSDs, it would not skip pages because all the elements have to be accessed at least once in the worst case.

Partition-based intersection algorithms

Baeza-Yates et al. proposed an algorithm for intersection based on the divide-and-conquer framework [BYSC05]. If extended to SSDs, it would load the entire lists to memory because every element has to be accessed at least once.

Ding and König proposed another partition-based algorithm [DK11], which splits a list into partitions based on the universal hash. Unfortunately, if extended to SSDs, it would also access the entire lists (see Theorem 3.3 of [DK11]).

Bitmap-based intersection algorithms

Up to this point, we have represented a list as a sorted array. The list can also be represented as a bitmap: set the i-th bit to 1 if and only if i is in the list. Then the intersection is computed using bitwise operations [CM10b]. If extended to SSDs, the *entire* bitmap of a list has to be loaded to memory.

3.3.3 SSD-optimized design in IR and DB

Several prior works have studied the impact of SSDs to IR systems. Huang and Xia discussed allocating the inverted index on SSDs and HDDs [HX11] to maximize query performance while minimizing operational cost. A similar issue was also discussed in [LLX+12]. The impact of SSDs on cache management was studied in [WLY+13, TWL13, WLY+14]. They found that existing cache policies optimized for HDDs do not work well on SSDs. The issue of inverted index maintenance on SSDs was studied in [LCL+12, JRSP14] by leveraging the fast random accesses of SSDs. However, all these works still used existing HDD-centric list intersection algorithms. This work, in contrast, focuses on experimenting SSD-aware intersection algorithms.

There are also a number of studies exploring the impact of SSDs to database join [SHWG08, DP09] since list intersection is a special case of join. But those studies focus on mitigating the slow random writes of SSDs, because join algorithms usually need to write data back to SSDs if the buffer is insufficient. However, the focus of this work is to leverage the performance of sequential reads and random reads (read-only).

3.4 Algorithms

Although there is no prior SSD-oriented intersection algorithm, a number of in-memory skipping-based intersection algorithms can be naturally extended to SSDs to skip unnecessary pages, as discussed in Section 4.10. In this section, we present the extended algorithms that will be experimented later on. Before that, we describe the problem and list structure first.

Problem statement. Consider k ($k \geq 2$) lists L_1, L_2, ..., L_k ($|L_1| \leq |L_2| \cdots \leq |L_k|$) stored on an SSD. Each list is stored in pages of a typical size, e.g., 4KB [THS+09, WLY+13]. The problem is to find the intersection of these lists, i.e., $\bigcap_{i=1}^{k} L_i$, while minimizing the total amount of data accessed in order to reduce the actual execution time.

We first focus on the intersection where all the lists are stored on the SSD initially in order to obtain clean results. Then, we evaluate the impact of cache where popular lists are cached in main memory.

List structure. We follow a typical setting in IR systems to represent and compress a list [MZ96, ZM06]. Each entry is a document ID d_i of 4 bytes before compression.[3] All document IDs in every list are sorted in ascending order. Depending on different systems, the lists can be compressed using a compression algorithm, e.g., PforDelta [ZHNB06] and

[3]All the evaluated algorithms are applicable to entries that contain other auxiliary information, e.g., document frequencies and positions.

SIMDPforDelta [WLPS17]. We also evaluate the impact of different compression schemes to the performance. If a list is compressed, we follow prior work [MZ96, ST07] to build skip pointers such that intersection only needs to examine those promising pages.

3.4.1 BL (Baseline)

We denote BL as the baseline algorithm that directly runs the existing HDD-centric algorithm on the SSD. This serves as a baseline when one uses SSDs to replace HDDs in their systems. In particular, BL reads each of the k lists in its entirety in a single thread and then runs multiple threads for in-memory intersection. Actually, it does not matter whether the whole list is accessed by using a single thread or multiple threads because both can saturate the I/O bandwidth as described in Table 3.1 as long as the *whole* list is accessed at a time.[4] Once the lists are loaded to memory, BL uses SL for the in-memory intersection to save memory accesses because it has high performance and widely used in practice [CM10b].

The advantage of BL is that it is simple and there is no need to change existing algorithms when replacing HDDs with SSDs. But the disadvantage is that BL can be slow because it reads many unnecessary pages.

3.4.2 SL (Short-Long)

As mentioned in Section 3.3.2, an important operation used in SL is $\mathsf{Member}(L,e)$, which checks whether element e is in list L as illustrated in Section 3.3.2. In main memory, it is usually implemented using binary search or skip lists. On SSDs, we implement $\mathsf{Member}(L,e)$ by pre-computing skip pointers [MZ96]: we store for each page the minimum[5] uncompressed ID (called a *skip pointer*). The skip pointers are small enough to be stored in memory since there is only one skip pointer per page. As an example consider the list L_3 in Figure 3.1, and assume each

[4]Note that parallelism makes a difference when the underlying list is accessed *page by page*.
[5]This can also be the maximum ID.

Algorithm 1: Short-long (SL) algorithm [CM07,CM10b]

```
Input: $k$ lists $L_1, L_2, ..., L_k$ on an SSD ($|L_1| \leq \cdots \leq |L_k|$)
Output: $L_1 \cap L_2 \cap \cdots \cap L_k$
1  result set $R \leftarrow \emptyset$;
2  for each element $e$ in each page of $L_1$ do
3      $flag \leftarrow$ true;
4      for $i \leftarrow 2$ to $k$ do
5          if Member$(L_i, e)$ is true then
6              continue;
7          else
8              $flag \leftarrow$ false;
9              break;
10     if $flag$ is true then
11         put $e$ to $R$;
12 return $R$;
```

page contains four elements. Figure 3.2 depicts the skip pointers: 10, 80, 160, and 400.

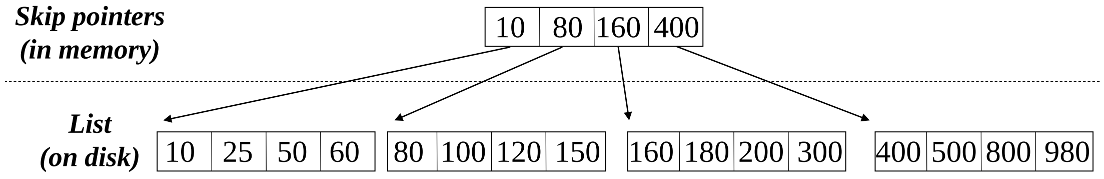


Figure 3.2: An example of the skip pointers of L_3 in Figure 3.1

Member(L, e) can be implemented efficiently by using skip pointers. A page that potentially contains e can be identified efficiently. Then, SL reads the page to verify whether e is actually in L. In this way, many unnecessary pages can be skipped. Algorithm 1 shows the extended SL algorithm. It reads the shortest list to memory (page by page) and skips over the longer lists via Member(L, e). Note that SL only requires a page of L_1 instead of the whole list L_1 to fit in memory if L_1 is very long. In this way, SL reduces memory footprint significantly compared to BL.

Note that SL is equivalent to SvS that intersects two lists at a time. Thus, we do not consider SvS in this work.

Algorithm 2: Zig-Zag (ZZ) [DLOM00, DLOM01]

```
Input: $k$ lists $L_1, L_2, ..., L_k$ on an SSD ($|L_1| \leq \cdots \leq |L_k|$)
Output: $L_1 \cap L_2 \cap \cdots \cap L_k$
1  result set $R \leftarrow \varnothing$;
2  set $e$ as the first element of $L_1$;
3  $i \leftarrow 2$;
4  while $e$ is not invalid do
5      let $L_i$ be the current list to examine;
6      $s \leftarrow$ Successor$(L_i, e)$;
7      if $s = e$ then
8          $counter \leftarrow counter + 1$;
9          if $counter = k$ then
10             put $e$ to $R$;
11             set $e$ to the next unprocessed element in $L_i$;
12     else
13         $e \leftarrow s$;
14     $i \leftarrow (i+1) \bmod k$;
15 return $R$;
```

3.4.3 ZZ (Zig-Zag)

The Zig-Zag (ZZ) intersection algorithm [DLOM00, BK02] adopts another way of using skip pointers for intersection as described in Section 3.3.2. ZZ implements Successor(L, e) to find the successor of e in L instead of implementing Member(L, e). When extended to SSDs, we can implement Successor(L, e) using skip pointers (shown in Figure 3.2) and only read the promising page to skip unnecessary pages, see Algorithm 2. The skip pointers are stored in memory.

3.4.4 SBF (Single-Bloom-Filter)

We observe that skip pointer based intersection algorithms, i.e., SL and ZZ still incur unnecessary page accesses:

- CASE 1: The skip pointers can only route element e to the page whose range covers e. However, e may still not exist in that page, in which case reading it is wasteful. As an

Algorithm 3: SBF (Single-Bloom-Filter)

```
   Input: $k$ lists $L_1, L_2, ..., L_k$ on an SSD ($|L_1| \leq \cdots \leq |L_k|$)
   Output: $L_1 \cap L_2 \cap \cdots \cap L_k$
1  result set $R \leftarrow \varnothing$;
2  for each element $e$ in each page of $L_1$ do
3      $flag \leftarrow$ true;
4      for $i \leftarrow 2$ to $k$ do
5          find the page $\mathcal{P}$ of $L_i$ whose range covers $e$ in skip pointers;
6          test whether $e \in \mathcal{P}$ using the bloom filter of $\mathcal{P}$;
7          if the bloom test returns false then
8              $flag \leftarrow$ false;
9              break;
10         else
11             read the page $\mathcal{P}$ from the SSD to memory;
12             if $e \notin \mathcal{P}$ then
13                 $flag \leftarrow$ false;
14                 break;
15     if $flag$ is true then
16         put $e$ to $R$;
17 return $R$;
```

Table 3.3: The storage of different types of data

Type	Storage
skip pointers	in main memory
bloom filters	in main memory
lists	on disk (SSD or HDD)

example, let $e = 200$, and a page contains four elements: $\{15, 100, 300, 500\}$. The page will be identified by the skip pointers since e falls in the range. However, e is not in the page.

- CASE 2: Even if e is in $L_1, ..., L_{i-1}$ ($2 \leq i \leq k$), it may not be in L_i, making the previous page accesses (containing e) from $L_1, \cdots, L_{i-1}$ wasteful. As an example, suppose e is in L_1 and L_2 but not in L_3, then the pages in L_1 and L_2 that contain e are unnecessarily accessed, since e is not a result.

To solve the problem, a natural idea is to use bloom filters [Blo70]. In particular, we build for each page a bloom filter to improve the SL algorithm. Note that ZZ cannot benefit from bloom filters because ZZ relies on Successor(L, e) instead of Member(L, e) while bloom filters can only be helpful for membership testing. We show two modifications of SL to leverage bloom filters, namely, SBF (single-bloom-filter) and MBF (multi-bloom-filter), see Algorithm 3 and Algorithm 4. For fast execution, we store all the bloom filters in main memory (as shown in Figure 3.3).

In SBF, whenever a target page $\mathcal{P}$ (whose range contains e) is identified by the skip pointers, instead of loading $\mathcal{P}$ immediately, SBF performs a bloom test between e and the bloom filter of $\mathcal{P}$. If e fails the bloom test (i.e., the bloom test returns false), there is no need to read $\mathcal{P}$ since e is definitely not in $\mathcal{P}$. Otherwise SBF reads it. SBF can skip the unnecessary pages introduced by CASE 1, but not CASE 2.

We analyze the false positive rate of reading a non-promising page. Before that, we introduce the concept of *reference count*.

Definition 1 (Reference count) *The reference count of a page $\mathcal{P}$ (in a list L) is the number of elements that are routed to $\mathcal{P}$ by the skip pointers of L.*

The reference count of a page is actually the number of bloom tests that must be performed against that page. Let s be the bloom filter size (in the number of bits per element), then the false positive rate of accessing an element in the page is 0.6185^s [Mit01]. Let c be the reference count, then the false positive rate of accessing the page is $1-(1-0.6185^s)^c$ because as long as one of the c bloom tests produces a false positive, the page has to be unnecessarily accessed. Thus, the effectiveness of SBF depends on the bloom filter size and the reference count. And the reference count depends on many factors as well, e.g., the intersection size, and the number of lists. Assuming all the elements in each list are distributed evenly, then the reference count c_i of a page in L_i can be computed as $c_i = \frac{|L_1 \cap L_2 \cap \cdots \cap L_{i-1}|}{|L_i|/b}$ where b is the number of elements that a page contains because every intersected result in the previous $(i-1)$ lists has to be probed on the i-th list.

Note that in SL, as long as the reference count $c \geq 1$, that page has to be accessed. But SBF can alleviate the situation because the false positive rate can be low for larger c (e.g., 2 to 5) as shown later in Figure 3.8 when s is high (e.g., $s = 8$ bits).

3.4.5 MBF (Multi-Bloom-Filter)

MBF (multi-bloom-filter) is another way of using bloom filters (stored in memory as mentioned in Figure 3.3), see Algorithm 4. It is different from SBF when the number of lists $k \geq 3$. In other words, MBF and SBF are the same when the number of lists is two. It has a filtering and a verification phase. (1) Filtering phase: if the bloom test of SBF returns true, instead of reading the target page immediately, MBF performs the bloom tests between e and all the other lists as well. In other words, in the filtering phase, MBF checks whether e (in L_1) passes the bloom tests of all the other lists $L_2, \cdots, L_k$. If not, there is no need to read any page because e is definitely not a result. If yes, MBF continues to the verification phase. (2) Verification phase: MBF reads the $(k-1)$ target pages from L_2 to L_k that passed the filtering phase. The MBF algorithm can skip the unnecessary pages introduced by both CASE 1 and CASE 2.

Remark. Skip pointers and bloom filters used in SL, ZZ, SBF, and MBF can reduce data movement, but also increase random reads. The tradeoff makes sense on SSDs where random reads are comparable to sequential reads. However, on HDDs, they cannot improve the performance due to the expensive random reads.

3.4.6 Parallel intersection algorithms

By default, all the algorithms (BL, SL, ZZ, SBF, and MBF) in this work are evaluated in multiple threads. We discuss in detail how to parallelize the algorithms. The key question is: how

```
Algorithm 4: MBF (Multi-Bloom-Filter)
    Input: $k$ lists $L_1, L_2, ..., L_k$ on an SSD ($|L_1| \leq \cdots \leq |L_k|$)
    Output: $L_1 \cap L_2 \cap \cdots \cap L_k$
 1  result set $R \leftarrow \varnothing$;
 2  for each element $e$ in each page of $L_1$ do
 3      $flag \leftarrow$ true;
        // phase 1:  filtering
 4      for $i \leftarrow 2$ to $k$ do
 5          find the page $\mathcal{P}$ of $L_i$ whose range covers $e$ in skip pointers;
 6          test whether $e \in \mathcal{P}$ using the bloom filter of $\mathcal{P}$;
 7          if the bloom test returns false then
 8              $flag \leftarrow$ false;
 9              break;
        // phase 2:  verification
10      if $flag$ is true then
11          for $i \leftarrow 2$ to $k$ do
12              read the page $\mathcal{P}$ (of $L_i$ whose range covers $e$) from the SSD to memory;
13              if $e \notin \mathcal{P}$ then
14                  $flag \leftarrow$ false;
15                  break;
16          if $flag$ is true then
17              put $e$ to $R$;
18  return $R$;
```

to partition the k lists for parallel intersection? We want to create many independent intersection tasks such that each thread works on one task individually. We choose the state-of-the-art partition strategy proposed in [TCJ11]. It works as follows. Assume there are N threads, then it partitions L_1 into N even sublists: L_1^1, L_1^2, $\cdots$, L_1^N. For each sublist L_1^i ($1 \leq i \leq N$), it performs binary search of $L_1^i[0]$ on L_j ($j \geq 2$) to partition L_j into N sublists. Thus, the N intersection tasks are: $(L_1^1 \cap L_2^1 \cap \cdots \cap L_k^1)$, $(L_1^2 \cap L_2^2 \cap \cdots \cap L_k^2)$, $\cdots$, $(L_1^N \cap L_2^N \cap \cdots \cap L_k^N)$. Every intersection task will be executed by the corresponding algorithm.

Remark. Parallel I/O is crucial to SSD-based list intersection, much more important than ever before. It is precisely and particularly suitable for SSDs *and* skipping-based intersection: (i) if lists are accessed entirely without skipping, it is not necessary to apply parallel intersection because sequentially loading a long list can already saturate the SSD's I/O bandwidth. Just because of skipping, parallelism becomes *the only way to fully utilize the SSD's I/O bandwidth*; (ii) if lists are stored on HDDs, parallelism cannot improve performance because HDDs only have one disk moving head that can serve at most one disk I/O simultaneously.

3.5 Results on synthetic data

In this section, we present experimental results on synthetic datasets to understand the impact of the key parameters to the performance.

Experimental settings. We conduct experiments on a commodity machine (Intel i7 3.10 GHz CPU, 4 physical cores, 8 hyper-threaded cores, 16GB DRAM) with Windows 8 installed. The experimental platform also includes an SSD (Samsung 850 Pro SSD 256GB[6]) and an HDD (Seagate HDD 2TB, 7200rpm[7]). All the algorithms are coded in C++. We use the Win32 native thread APIs to implement parallelism. By default, all the algorithms run at 32 threads in parallel. Although the CPU has 8 hyper-threaded cores, opening 32 threads can also improve performance because of frequent I/O operations.[8] Moreover, the page size is 4KB by default (in both HDD and SSD).

Evaluation metrics. We measure the performance of an algorithm in two aspects: actual execution time and the number of pages accessed. The execution time is averaged across three runs for accuracy.

Synthetic datasets. To study the effect of crucial parameters to the overall performance, we generate synthetic data by fixing all the other parameters when evaluating the effect of a particular parameter. Unless otherwise stated, we use the default settings shown in Table 3.4.

[6]http://www.samsung.com/semiconductor/minisite/ssd/product/consumer/850pro.html

[7]https://www.seagate.com/files/staticfiles/docs/pdf/datasheet/disc/barracuda-ds1737-1-1111us.pdf

[8]We do not open more than 32 threads because the SSD's I/O queue depth is 32, i.e., it accepts at most 32

Table 3.4: Default parameters used in synthetic data

Parameters	Default
number of threads	32
number of lists	2
list size	$\lvert L_1 \rvert = 10^4$, $\lvert L_2 \rvert = 10^7$
list size ratio	1000
intersection ratio	1% of $\lvert L_1 \rvert$
bloom filter size	4 bits (per element)
data distribution	uniform (from domain $[0, 2^{32} - 1]$)
cache	no
compression	no

Table 3.5: Number of pages for Figure 3.3

BL	ZZ	SL	SBF	MBF
11177	6553	6553	1514	1514

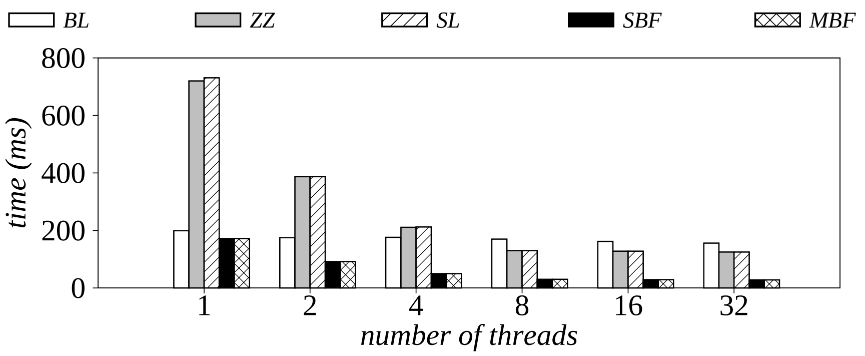
3.5.1 Effect of number of threads

We first examine the effect of parallelism, which is very important to the performance. We use the default settings (shown in Table 3.4) except we vary the number of threads from 1 to 32. Figure 3.3 shows the execution time of the five algorithms on both SSD and HDD. Table 3.5 shows the number of pages for each algorithm, which is irrelevant with the number of threads. There are many interesting results of this experiment.

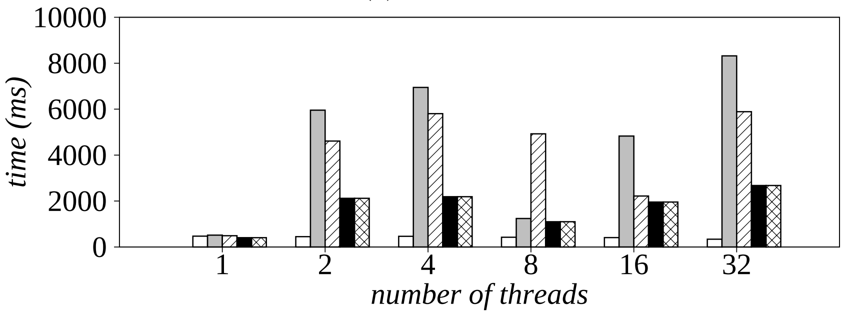
(1) For the baseline algorithm BL, Figure 3.3 demonstrates that multiple threads do not help much in reducing the execution time on both SSD and HDD. That is because BL loads the whole list from disk to memory using a single thread and executes intersection in memory where the I/O cost is absolutely the performance bottleneck. Note that for list-at-a-time I/O access pattern, using a single thread is able to saturate the disk bandwidth for both SSD and HDD.

(2) For SL, the results are very interesting. Let's first look at the results of SL on the SSD. Figure 3.3a shows that when the number of threads is 1, SL is even slower than BL although SL

outstanding I/O requests simultaneously.



(a) on SSD



(b) on HDD

Figure 3.3: Effect of number of threads

reads less data than the baseline BL (see Table 3.5). That is because BL can fully utilize the SSD's I/O bandwidth by reading a whole list at a time but SL cannot because it reads a page at a time in order to skip unnecessary pages. However, as the number of threads increases, the performance of SL becomes better while BL does not increase much. At the point when the number of threads is 8, SL runs 1.3X faster than BL. This confirms our conjecture that on SSDs, we should access a list by using "page-at-a-time multi-threaded" I/O access pattern instead of the conventional "list-at-a-time single-threaded" access pattern.

To make the results comprehensive, we also report the execution time on the HDD in Figure 3.3b. It shows a completely different picture with the SSD in the sense that SL always runs slower than the baseline BL no matter how many threads are used. More importantly, increasing the number of threads (on the HDD) can even make the performance slower, e.g., SL runs slower when the number of threads increases from 1 to 2. That is because of expensive random seeks introduced by multiple threads. Note that when the number of threads is 1, SL has similar performance to BL because those random seeks are short under a single thread and short seeks tend to have high performance as explained in [RW94]. However, for multiple threads, random seeks tend to be long due to the context switch between different threads.[9] This confirms that HDD-optimized intersection algorithms should read the whole list to memory instead of skipping pages.

This also explains that if we treat the SSD as a drop-off disk and apply an existing HDD-centric algorithm BL, it becomes sub-optimal. For example, the intersection time (BL) on the HDD is 474ms; if we only replace the HDD with the SSD without changing any algorithm, then the execution time (BL on the SSD) drops to 199ms due to SSD's fast I/O performance. However, if we optimize the intersection algorithm by leveraging SSD's unique properties, the execution time (SL on the SSD) can drop to 125ms, which makes a better use of the SSD.

(3) For ZZ, it reads the same number of pages with SL when the number of lists is two.[10] As a result, the execution time of ZZ is the same as SL on the SSD. But on the HDD, ZZ is slower than SL because ZZ introduces more back-and-forth random accesses.

(4) For SBF and MBF, they are basically the same when the number of lists is two. Thus, they have the same execution time and read the same amount of data. On the SSD, they can improve the performance of SL a lot because the intersection size is small. But on the HDD, bloom filters will not help because they introduce many random accesses by filtering out non-promising pages.

3.5.2 Effect of list size

Figure 3.4 shows the effect of list size on the SSD. We use two lists L_1 and L_2 ($|L_1| \leq |L_2|$). We vary $|L_2|$ from 1 million to 100 million and set $\frac{|L_2|}{|L_1|} = 1000$, thus, $|L_1|$ also increases. The

[9]Note that the results on the HDD do not monotonically change with the number of threads. That is because of the impact of disk I/O reordering if multiple I/O requests are issued, which highly depends on disk internals [RW94].

[10]But as we show later in Figure 3.7, ZZ reads more data than SL when the number of lists exceeds two.

rest parameters follow the default values shown in Table 3.4. We omit ZZ and MBF because for two-list intersection, ZZ is the same as SL and MBF is the same as SBF.

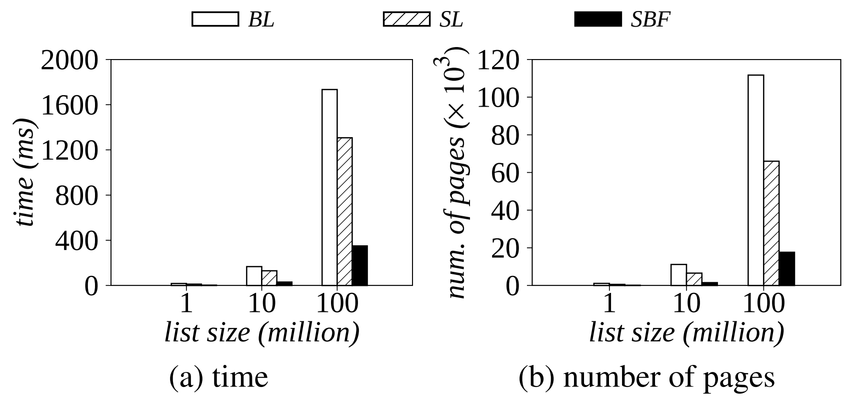


Figure 3.4: Effect of list size

Figure 3.4 reveals that the execution time and the number of page accesses increase as lists become longer. In particular, the execution time is roughly proportional to the number of page accesses because random reads and sequential reads on the SSD are roughly similar. It also shows that skipping-based intersection algorithms SL and SBF outperform BL.

3.5.3 Effect of list size ratio

We define the list size ratio w as $\frac{|L_2|}{|L_1|}$. We fix $|L_2| = 10^7$, set the intersection size at 1% of $|L_1|$, and vary w from 1 to 10^4. Figure 3.5 shows the results. For all the algorithms, the execution time and the number of pages decrease as w increases, because $|L_1|$ decreases.

Figure 3.5 demonstrates that when w is small (e.g., $w \leq 10$), all the algorithms fail to skip any pages, because many elements in L_1 are routed to the same page of L_2, making the page's reference count too high. Thus, many pages in L_2 become relevant. When $w \geq 100$, the skipping-based algorithms (e.g., SL and SBF) start to skip and their performance improves.

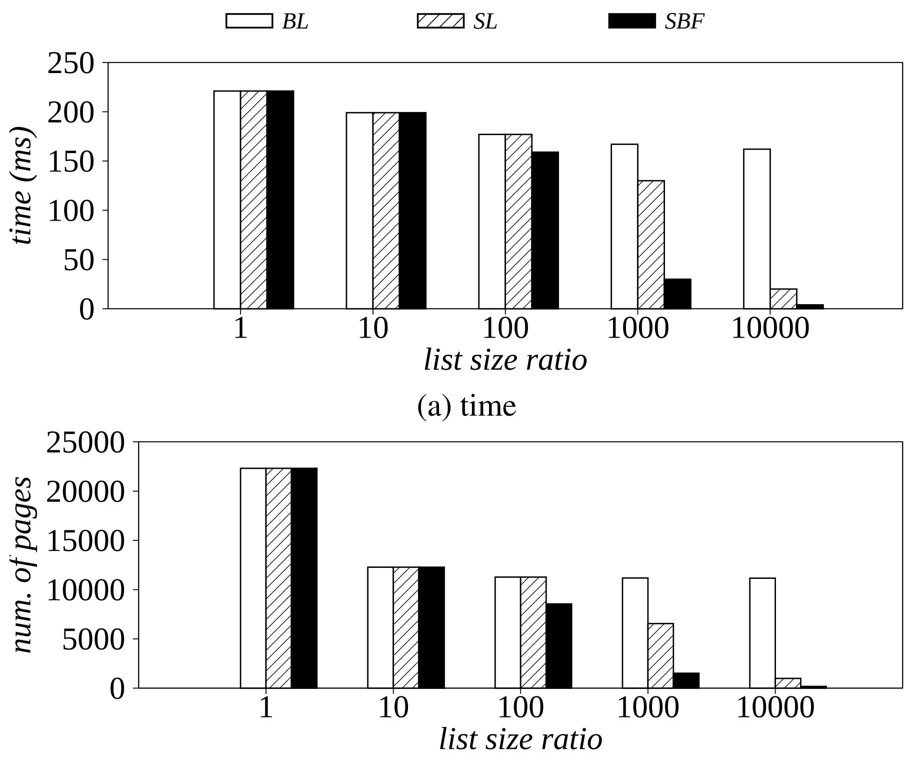
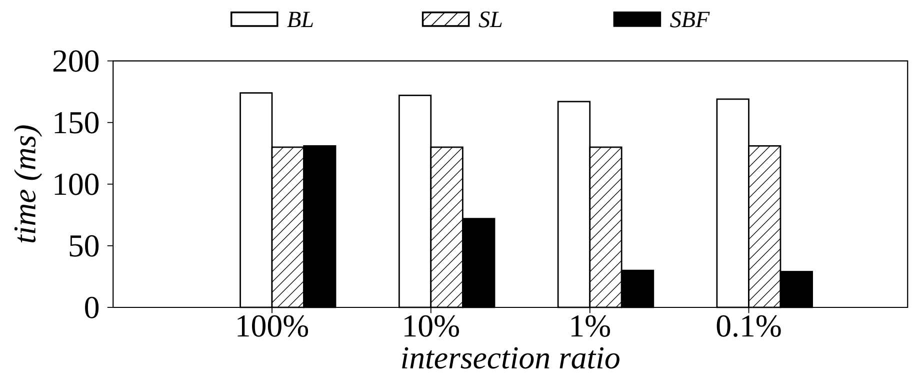
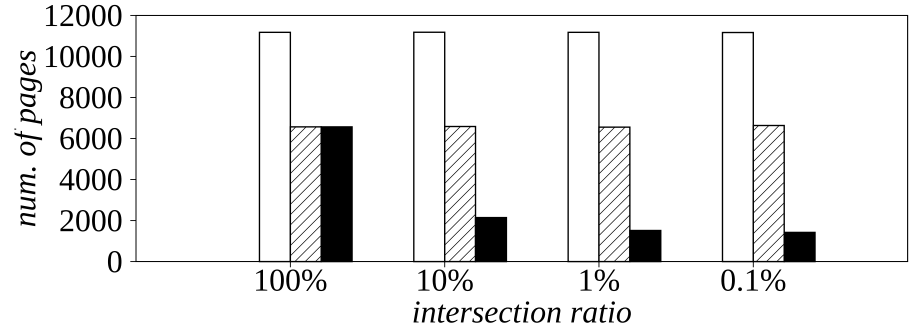


Figure 3.5: Effect of list size ratio



(a) time



(b) number of pages

Figure 3.6: Effect of intersection ratio

3.5.4 Effect of intersection ratio

The intersection ratio is also an important parameter. In this experiment, we fix $|L_2| = 10^7$, $|L_1| = \frac{|L_2|}{1000}$, and vary the intersection ratio r from 0.01% (of $|L_1|$) to 100% (of $|L_1|$). Figure 3.6 shows the results. It shows that BL and SL do not change much as r varies. For BL, that is because it experiences a bottleneck of reading all the lists, no matter what the intersection ratio is. For SL, as long as a page range in L_2 covers an element, that page has to be loaded regardless of whether the page really contains the element. That is because SL only maintains the range information via skip pointers. But the performance of SBF improves when r becomes smaller because bloom filters can rule out unnecessary pages.

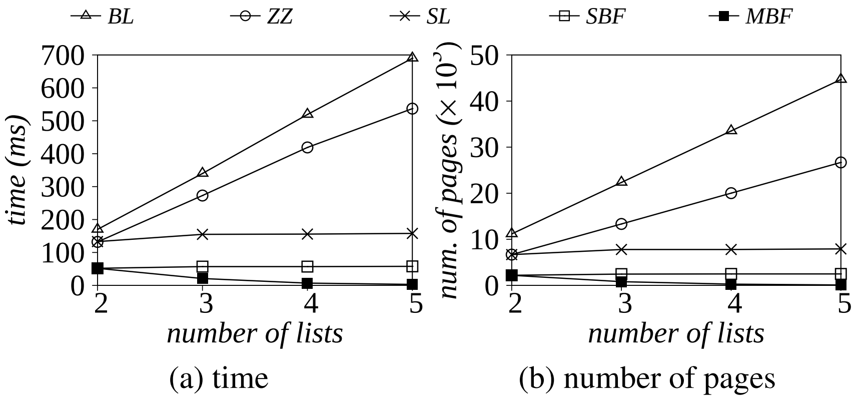


Figure 3.7: Effect of number of lists

3.5.5 Effect of number of lists

In this set of experiments, we examine the effect of the number of lists k (ranging from 2 to 5). We largely follow [DK11] to generate lists. In particular, we set $|L_1| = 10^4$ and $|L_2| = |L_3| = |L_4| = |L_5| = 10^7$ because in practice, short and long lists are usually mixed in a query. Moreover, we set the intersection size between L_i ($i \geq 2$) and L_1 as 10% of $|L_1|$. As a result, $|L_1 \cap L_2| = 1143$, $|L_1 \cap L_2 \cap L_3| = 113$, $|L_1 \cap L_2 \cap L_3 \cap L_4| = 18$, $|L_1 \cap L_2 \cap L_3 \cap L_4 \cap L_5| = 4$.

Figure 3.7 plots the results. It reveals that the skipping-based algorithms scale well with k due to efficient skipping. More importantly, the gap between BL and SL (or SBF, MBF) becomes larger as k increases. That is because SL, SBF, and MBF perform list intersection from short lists to long lists. Let L_i be the current list, then the intersection size of the previous lists is $|L_1 \cap L_2 \cap \ldots \cap L_{i-1}|$. Thus, as i increases, the gap between $|L_1 \cap L_2 \cap \ldots \cap L_{i-1}|$ and $|L_i|$ increases because the list size is increasing from L_1 to L_k, while the intersection size is getting smaller with more lists being intersected. According to Figure 3.5, the performance gap increases when the list size ratio increases. Thus, it becomes even more relevant to develop SSD-optimized algorithms when the intersection involves many lists.

Figure 3.7 also shows that MBF outperforms SBF because MBF can filter out more unnecessary pages. Another interesting result is that the number of pages accessed by MBF can even decrease when k increases. This is because the false positive rate of a page decreases when

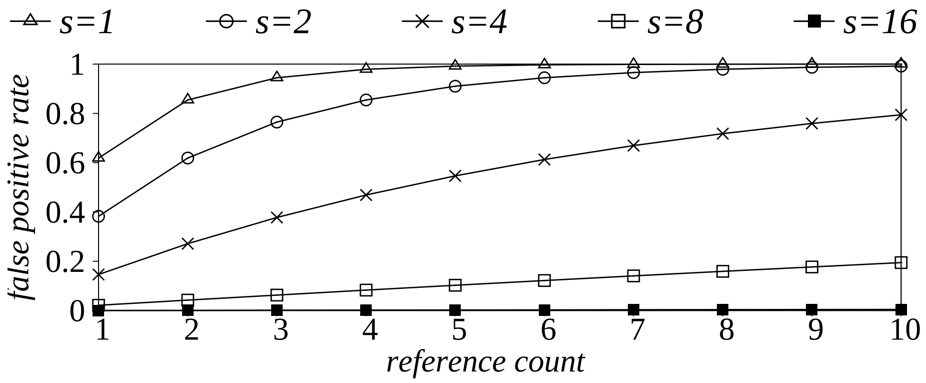


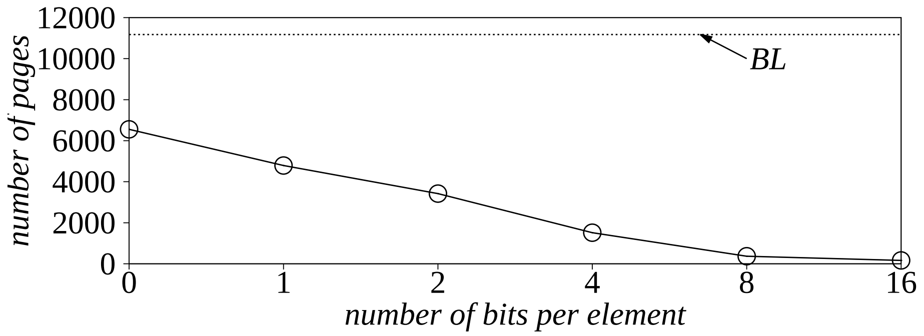
Figure 3.8: False positive rate in SBF

k becomes larger in MBF; thus, the bloom filters in MBF can in turn filter out many unnecessary pages. Figure 3.7 also confirms that SL is better than ZZ when $k \geq 3$. The intuition is that, consider at some point, ZZ and SL access the same element a in L_1. Then for ZZ, after it walks through one iteration (accessing $L_1 \rightarrow L_2 \rightarrow \cdots \rightarrow L_k \rightarrow L_1$) and returns to L_1 again, it will access the element that is in the same page of a with high probability if the lists are distributed uniformly. However, if a is not a result, ZZ wastes many I/Os compared to SL because SL can switch back to L_1 much earlier since only the intersection of $L_1 ... L_{i-1}$ are intersected with L_i.

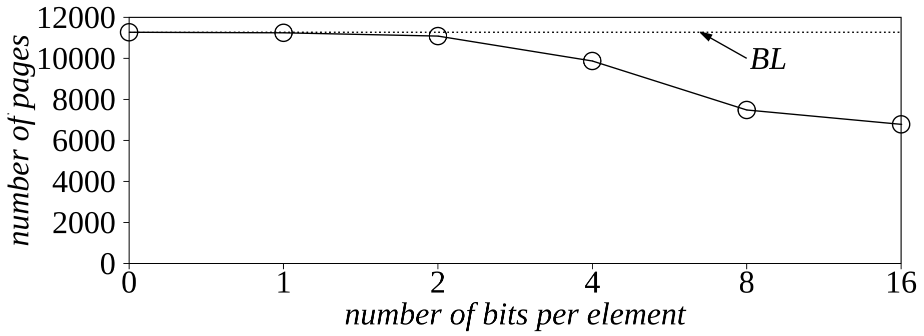
3.5.6 Effect of bloom filter effectiveness

Next, we evaluate SBF to understand the effectiveness of bloom filters. The effectiveness highly depends on the false positive rate, i.e., $f = 1 - (1 - 0.6185^s)^c$ where s is the bloom filter size (in bits per element) and c is the reference count of the page. Thus, we first plot f with varying s and c in Figure 3.8. It shows that f increases with both s and c. In particular, when s is 1, 2, or 4, f increases very quickly when c increases. Thus, bloom filters are effective only if the reference count is low unless the bloom filter size is high, say, $s = 16$ bits.

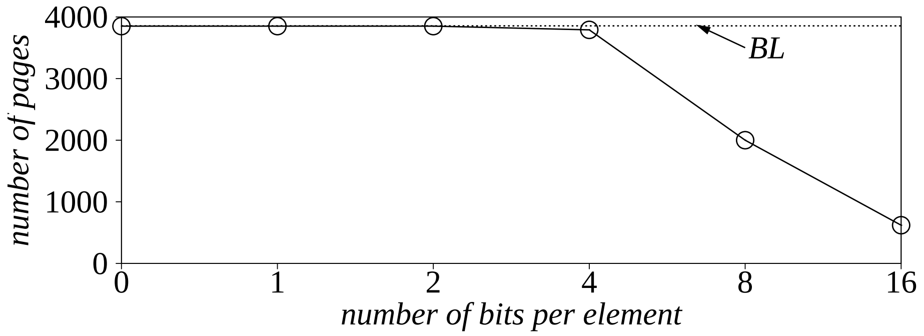
Then we evaluate the actual performance of SBF in Figure 3.9 with varying bloom filter size. We use three settings that have different list sizes and intersection sizes, i.e., they have different reference counts of data pages. In Figure 3.9a, $|L_1| = 10^4$, $|L_2| = 10^7$, and $|L_1 \cap L_2| =$



(a) $|L_1| = 10^4$, $|L_2| = 10^7$, intersection size = 100



(b) $|L_1| = 10^5$, $|L_2| = 10^7$, intersection size = 10000



(c) $|L_1| = 10^5$, $|L_2| = 10^7$, intersection size = 10000, compressed

Figure 3.9: Effect of bloom filter size in SBF

100; In Figure 3.9b, $|L_1| = 10^5$, $|L_2| = 10^7$, and $|L_1 \cap L_2| = 10000$; In Figure 3.9c, it has the same setting with Figure 3.9b unless the lists are compressed using SIMDPforDelta [LB15a], which performs the best as we see in Section 3.5.8. As a reference, we also show the baseline BL. Generally, Figure 3.9 shows that the performance improves when s increases. But the performance improvement is different. In particular, Figure 3.9a shows the best improvement when s increases while Figure 3.9c shows the worst improvement. That is because in Figure 3.9a, data pages' reference counts are much smaller than that in Figure 3.9c. Note that the reference count of a page in L_2 can be roughly estimated as the ratio between $|L_1|$ and the number of pages taken by L_2. Thus, they have a low false positive rate according to Figure 3.8.

3.5.7 Effect of zipf distribution

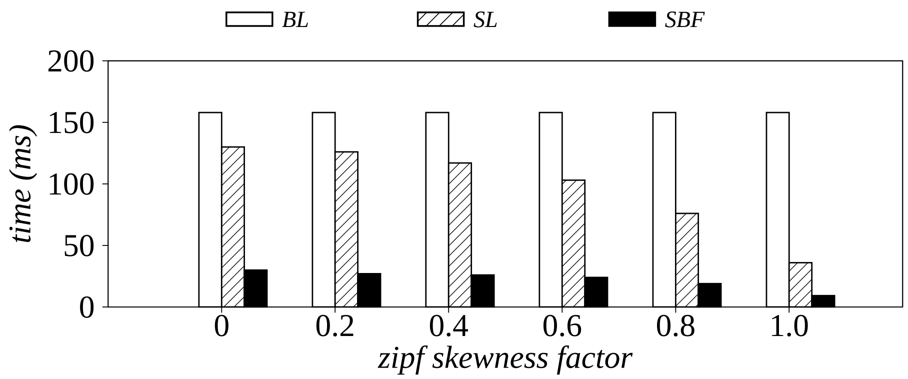
In the previous experiments, we assume lists are distributed uniformly. In the next experiment, we assume lists are distributed following the zipf distribution. In particular, the value i in a list is included with a probability of $\frac{1/i^f}{\Sigma_{j=1}^{d}(1/i^f)}$ where d is the domain size (2^{32}) and f is the skewness factor. Note that when $f = 0$, the zipf distribution is the same as uniform distribution. We use the default setting unless the lists are distributed differently.

Figure 3.10 shows the results. It shows that when f increases from 0 to 1, the performance of BL is not affected because it reads the whole lists. However, SL and SBF read less data as the lists become more skewed. That is because the intersection results tend to be concentrated on fewer pages if data is more skewed, which provides a good opportunity for data skipping.

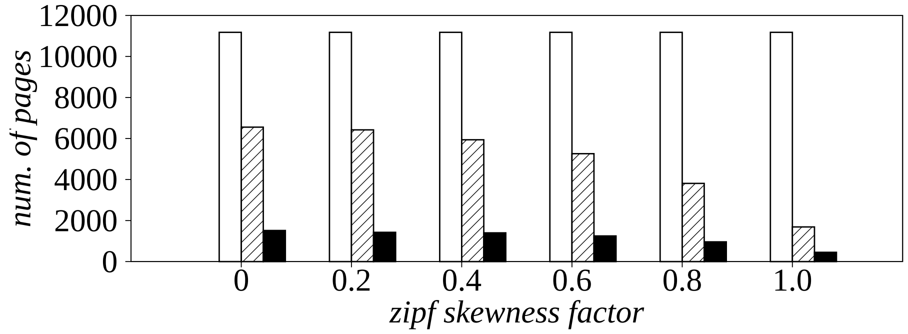
3.5.8 Effect of compression

Figure 3.11 compares BL, SL, and SBF when the lists are compressed using different compression algorithms. We use the default settings unless the lists are compressed. We evaluate the performance based on three best compression approaches, namely, SIMDPforDelta, SIMDPforDelta*, and SIMDBP128*, as they have high performance and low space overhead as reported in [WLPS17].

Figure 3.11 shows that all the algorithms become faster when compared to uncompressed lists because the lists take less space after compression. We see that the gap between BL and skipping-based algorithms (SL and SBF) becomes smaller on compressed data. That is because after compression, a data page contains more elements and it becomes more difficult to skip a



(a) time



(b) number of pages

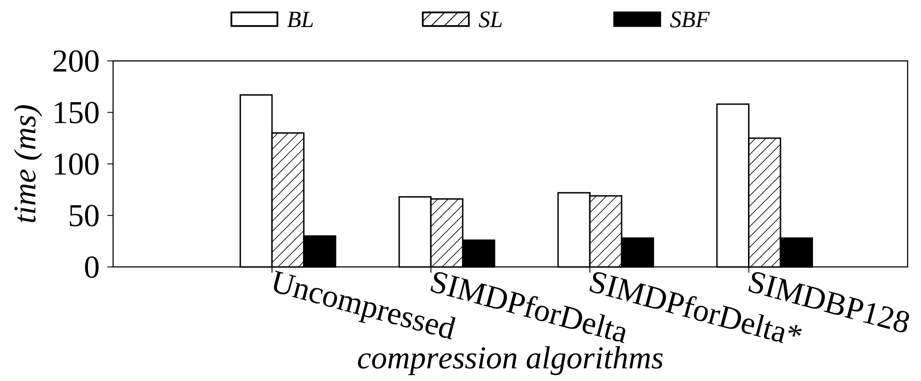
Figure 3.10: Effect of Zipf distribution

page unless the intersection size is very small. But still, SL and SBF perform better than BL.

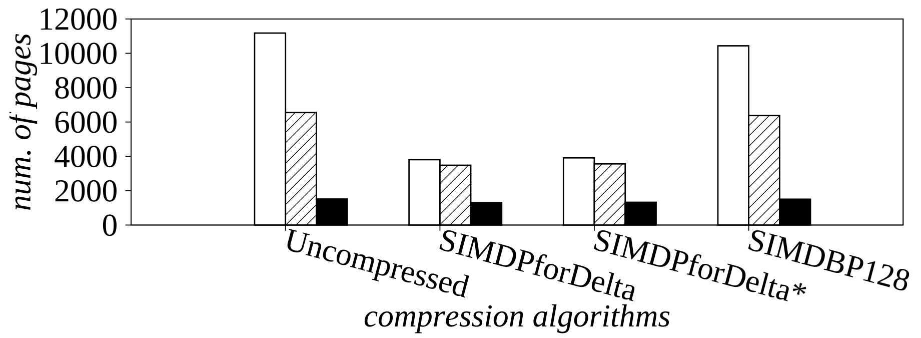
3.5.9 Effect of cache

Finally, we evaluate the effect of caching popular lists in memory. We use five lists generated in Section 3.5.5 and vary the number of lists cached, see Figure 3.12. Let *i* be the number of lists cached, it means the first *i* lists are cached in memory.

Figure 3.12 shows that even with cached lists, it makes even more sense for skipping on-disk lists. That is because the intersection size of the in-memory lists can be very small compared to the sizes of the on-disk lists since $|L_1| \leq ... \leq |L_k|$. Then the skipping-based algorithms can skip many unnecessary pages of the on-disk lists. Thus, as long as some lists are stored on the SSD, it becomes interesting to apply skipping-based intersection algorithms instead of using the



(a) time



(b) number of pages

Figure 3.11: Effect of compression

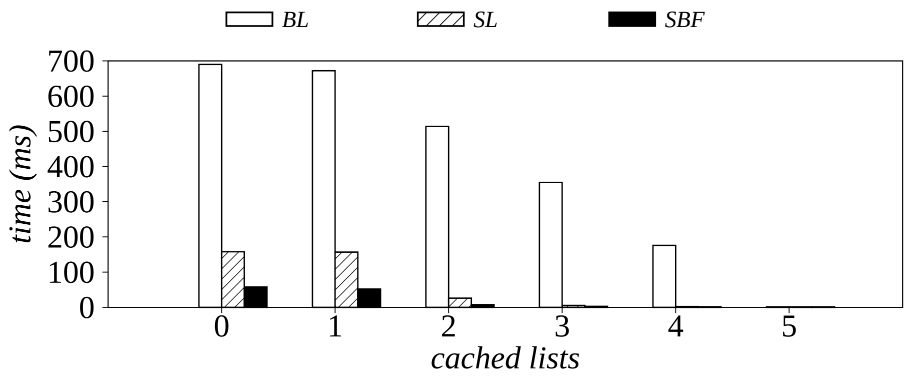
conventional BL algorithm.

3.6 Results on real data

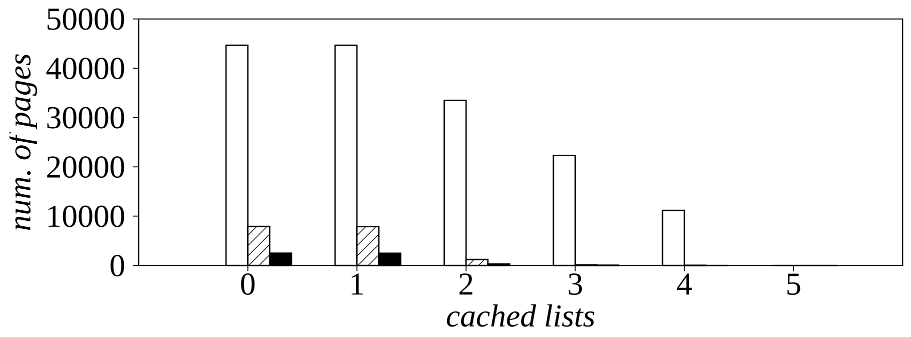
In this section, we present experimental results on two real datasets: IR data and DB data.

IR data. It is a standard benchmark in the information retrieval community. It includes 41 million Web pages crawled by CMU in 2012.[11] The total data size is around 300GB. The query log contains 131,654 real queries from the TREC 2005 and 2006 (efficiency track).[12] We parse the Web documents and build inverted lists for the terms. By default, we compress the lists using SIMDPforDelta. The compressed index size is 12.5GB. During experiments, we run the

[11]http://www.lemurproject.org/clueweb12.php/
[12]http://trec.nist.gov/



(a) time



(b) number of pages

Figure 3.12: Effect of cache

queries over the corresponding dataset. In particular, for each query that contains *k* query terms, we compute list intersection among those *k* inverted lists to report the average execution time and the number of page accesses. Table 3.6 shows the detailed data statistics including list size, intersection ratio, list size ratio, and the number of lists.

DB data. It is a star schema benchmark,[13] which includes one fact table (LINEORDER) and four dimension tables (CUSTOMER, SUPPLIER, PART, and DATE). We set the scale factor as 10 and pick up four queries (i.e., Q1.1, Q2.3, Q3.4, and Q4.3). For each predicate of a query, we precompute a list of rows that are valid for the predicate as described in Section 3.2. By default, the lists are not compressed, and the impact of compression is studied in Table 3.9.

Unless otherwise stated, we run each algorithm in 32 threads and set the bloom filter size as 4 bits per element in SBF and MBF.

[13]http://www.cs.umb.edu/ poneil/StarSchemaB.pdf

Table 3.6: IR data characteristics

List size	$\leq 10^3$	$10^3 \sim 10^4$	$10^4 \sim 10^5$	$10^5 \sim 10^6$	$10^6 \sim 10^7$	$> 10^7$
Percentage	3.9%	4.0%	12.3%	33.9%	40.5%	5.4%

(a) The distribution of query-involved list sizes

Intersection ratio	$\leq 0.01\%$	$0.01\% \sim 0.1\%$	$0.1\% \sim 1\%$	$1\% \sim 10\%$	$> 10\%$
Percentage	22.3%	1.9%	17.1%	37.3%	21.4%

(b) The distribution of intersection ratios (the intersection size over the size of the shortest list)

List size ratio	≤ 10	$10 \sim 10^2$	$10^2 \sim 10^3$	$10^3 \sim 10^4$	$> 10^4$
Percentage	30.1%	37.9%	18.1%	6.3%	7.6%

(c) The distribution of list size ratios (max size / min size)

Number of lists	2	3	4	5	6	7	>7
Percentage	26.6%	31.5%	22.9%	11.2%	4.5%	1.9%	1.4%

(d) The distribution of number of lists (query terms)

Table 3.7: DB data characteristics

Query	**Number of lists**	**List sizes**	**Intersection size**
Q1.1	3	8573056 16367104 29998976	1168750
Q2.3	2	59648 12000640	12036
Q3.4	5	239360 239872 240128 240512 713600	0
Q4.3	5	2398336 2400128 8566528 8569984 11999360	358

3.6.1 Overall results

We first consider the setting where all the lists are stored on the disk initially without caching any lists in memory. Table 3.8 shows the results on IR and DB of the five intersection algorithms. (1) Generally, skipping-based algorithms (i.e., ZZ, SL, SBF, MBF) read less amount

Table 3.8: Results on real data

	BL	ZZ	SL	SBF	MBF
IR	27	25	18	17	16
DB (Q1.1)	1098	1166	1171	1174	1178
DB (Q2.3)	254	289	291	223	223
DB (Q3.4)	31	38	14	9	3
DB (Q4.3)	594	793	481	342	142

(a) time on SSD (ms)

	BL	ZZ	SL	SBF	MBF
IR	100	314	245	241	237
DB (Q1.1)	2058	62806	63701	63910	63910
DB (Q2.3)	647	12256	11790	10572	10572
DB (Q3.4)	67	457	119	35	8
DB (Q4.3)	1405	48983	35316	22319	10354

(b) time on HDD (ms)

	BL	ZZ	SL	SBF	MBF
IR	1600	1442	1102	1061	1040
DB (Q1.1)	61317	61317	61317	61317	61317
DB (Q2.3)	13461	13312	13312	10201	10201
DB (Q3.4)	1870	1870	800	638	370
DB (Q4.3)	37875	37874	24005	17815	8776

(c) number of pages

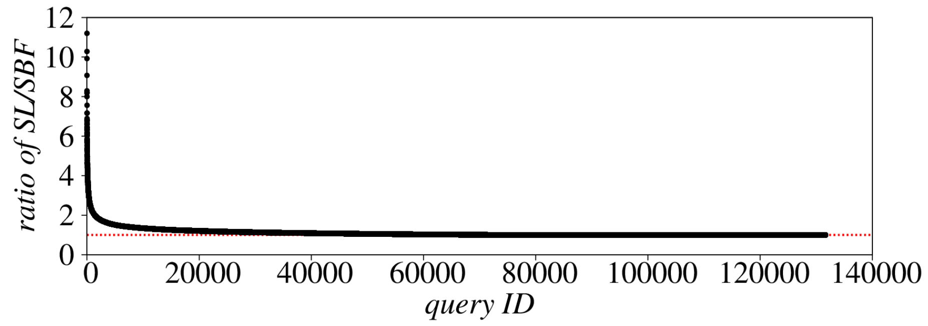
of data than the baseline BL,[14] and therefore they also run faster than BL on the SSD as reported in Table 3.8a. However, they run slower than BL on the HDD in Table 3.8b due to expensive random reads introduced by skipping. It explains why existing list intersection algorithms for HDDs prefer loading the entire list to memory to minimize the expensive random reads, i.e., "list-at-a-time single-threaded" I/O access pattern. This also confirms our conjecture that HDD-centric intersection algorithms become sub-optimal on SSDs. Thus, we shall deploy skipping-based parallel algorithms (i.e., "page-at-a-time multi-threaded" I/O access pattern) for intersection in SSD-resident systems. (2) On the SSD, ZZ performs worse than SL due to more accesses to longer lists, especially when the number of lists exceeds two. Thus, we should favor SL towards

[14]Note that when running Q1.1 on the DB dataset, all the algorithms read the same number of pages because the intersection size is too large that every data page contains a result.

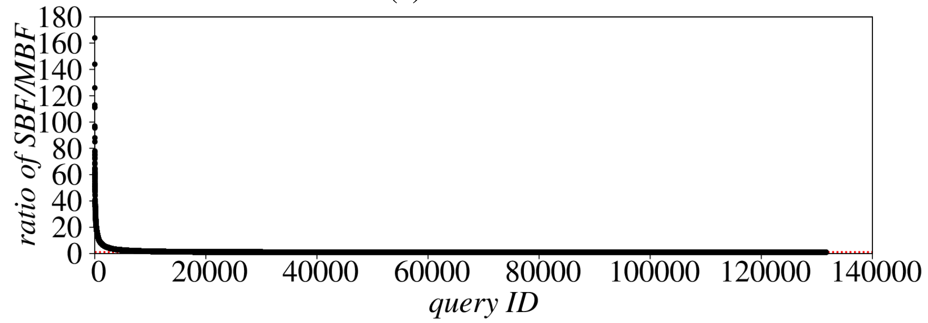
ZZ. (3) The efficiency of SBF and MBF varies because it depends on many parameters. For the IR data, both SBF and MBF read the similar number of pages as SL. We investigate the reasons next in Figure 3.13 and find that SBF and MBF are effective for some queries but not all. As a result, the average performance becomes similar to SL. We explain more in the next experiment. For Q1.1, SBF and MBF cannot reduce any page access because the intersection size is so large that every page contains the intersection result. For Q2.3, SBF and MBF access slightly less data than SL since the intersection size is pretty large. For Q3.4 and Q4.3, SBF and MBF outperform SL significantly because the intersection size is small and also the number of lists is high. (4) MBF outperforms SBF on Q3.4 and Q4.3 because the number of lists is high such that MBF has a smaller false positive rate than SBF since MBF only accesses a page when all the lists passed the bloom filtering tests.

3.6.2 Effect of bloom filters

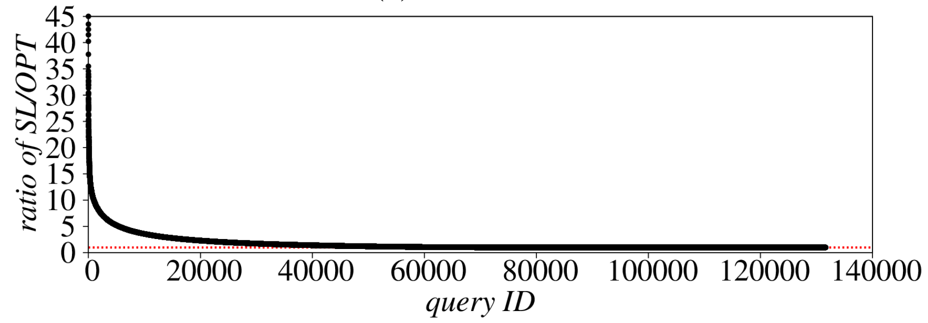
Next, we investigate why bloom filter based approaches, i.e., SBF and MBF, cannot improve much over SL on the SSD for the IR data. To understand why, we output for each query the number of page accesses incurred by SBF and MBF. Then we sort the queries in Figure 3.13 according to the ratio $r_1 = \frac{\text{SL}}{\text{SBF}}$ and $r_2 = \frac{\text{SBF}}{\text{MBF}}$ on the IR dataset. If $r_1 > 1$, it means SL reads more pages than SBF, i.e., SBF is better than SL. Similarly, if $r_2 > 1$, it means SBF reads more pages than MBF, i.e., MBF is better than SBF. Figure 3.13a shows that the distribution of r_1 is highly skewed in the sense that most queries have r_1 close to 1. For example, out of 131654 queries, only 1430 queries have $r_1 \geq 2$ and 3903 queries have $1.5 \leq r_1 < 2$. That is because the performance of SBF highly depends on the reference count of a page, i.e., how many elements routed to the page via skip pointers. If the reference count is high (e.g., ≥ 5), then the false positive rate is very high according to Figure 3.8. As for the relative performance between SBF and MBF, Figure 3.13b shows that the distribution of r_2 is also highly skewed, i.e., only a tiny portion of queries have high r_2. That is because MBF also works well when the reference count is low in order to achieve



(a) SL / SBF



(b) SBF / MBF



(c) SL / OPT

Figure 3.13: Relative performance of SL/SBF, SBF/MBF, and SL/OPT on IR data

a low false positive rate.

As a reference, we also plot the optimal number of pages (denoted as OPT) required for the intersection. It is surprising to see that SL works well on average; it is close to OPT. To see why, we store for each query the ratio of $\frac{\text{SL}}{\text{OPT}}$ and display in Figure 3.13c in a sorted order. It shows that, except for a small number of queries, SL runs as fast as OPT. That is because on the IR dataset, whenever a page is identified by SL, that page is very likely to have a true result on the two datasets, i.e., the reference count of those pages is 1.

Table 3.9: Effect of compression

	BL	ZZ	SL	SBF	MBF
Uncompressed	6961	5575	3761	3514	3412
SIMDPforDelta	1600	1442	1102	1061	1040
SIMDPforDelta*	1653	1488	1135	1092	1070
SIMDBP128	2311	2042	1513	1447	1416

(a) IR

	BL	ZZ	SL	SBF	MBF
Uncompressed	37875	37874	24005	17815	8776
SIMDPforDelta	7607	7067	6481	5391	3951
SIMDPforDelta*	7870	7870	6671	5521	4049
SIMDBP128	12924	12923	9989	7964	5100

(b) DB (Q4.3)

3.6.3 Effect of compression

Next, we show the impact of compression. Table 3.9 shows the number of page accesses on IR and DB (Q4.3) of the five intersection algorithms where the lists are stored using a different compression scheme. We pick up three best compression approaches, namely, SIMDPforDelta, SIMDPforDelta*, and SIMDBP128*, because they have high performance and low space overhead as reported in prior work [WLPS17].

The results show that compression plays an important role. The performance gap between skipping-based algorithms and the baseline BL is high on uncompressed lists. But when the lists are compressed, the gap becomes smaller. That is because if a list is compressed, then a page contains more elements. Thus, it becomes more difficult to skip a page during intersection The results also show an interesting result regarding different compression algorithms. As reported in [WLPS17], SIMDBP128* is the fastest for in-memory intersection. However, on disks (both SSD and HDD), SIMDBP128* is worse than SIMDPforDelta and SIMDPforDelta*. That is because SIMDBP128* consumes too much space overhead, incurring high I/O cost.

Table 3.10: Effect of cache

Index percentage	BL	ZZ	SL	SBF	MBF
0%	1600	1442	1102	1061	1040
12.5%	815	740	586	564	554
25%	485	443	350	336	331
50%	209	193	153	147	146

(a) IR

#cached lists	BL	ZZ	SL	SBF	MBF
0	37875	37874	24005	17815	8776
1	24482	24482	16000	9956	6975
2	21805	21805	13323	7279	4298
3	12240	12240	5052	3324	2537
4	2679	2679	2610	1239	1239

(b) DB (Q4.3)

3.6.4 Effect of cache

Next, we enable the cache to study the impact of caching in Table 3.10. For an intersection that involves *k* lists, if some of the lists are cached in memory, we do intersection between those in-memory lists first and then skip over on-disk lists by reading promising pages that contain the intersection results. Table 3.10a shows the number of page accesses of varying cached index in the IR data and Table 3.10b shows the number of page accesses of varying the number of cached lists in the DB data (for Q4.3). It shows that as the cached data increases, the gap between skipping-based intersection algorithms (i.e., ZZ, SL, SBF, MBF) and the baseline BL becomes smaller. That is because less amount of data are stored on disk such that the filtered pages become less. But we still see a clear performance improvement that skipping-based algorithms perform better than the BL, which indicates that as long as the entire inverted index is not cached in memory, it still makes sense to use skipping-based algorithms for SSD-based intersection.

3.7 Chapter summary

In this section, we summarize this chapter, provide lessons on how to design SSD-optimized intersection algorithms and then outline the future work.

3.7.1 Summary

In this chapter, we conducted a series of experiments to examine the impact of fast SSDs to list intersection algorithms. We showed that SSD-optimized intersection algorithms shall use "page-at-a-time multi-threaded" I/O access pattern instead of conventional "list-at-a-time single-threaded" pattern. In terms of the algorithmic design choices, we presented the impact of skip pointers, bloom filters, parallelism, cache, compression, list size, list size ratio, intersection ratio, and the number of lists to the performance. We summarize the main findings as follows:

1. Skip pointers play a critical role and SL efficiently leverages skip pointers for fast intersection (faster than ZZ). In most cases, it outperforms the baseline BL significantly on SSDs, especially when the list size skewness is high, the intersection size is small, and the number of lists is high.

2. The effectiveness of bloom filters varies significantly on different queries and datasets. From the high-level, it depends on the bloom filter size s and reference count c. For a moderate value s (e.g., s is 4 or 8), bloom filters are effective when c is in the range of say 2 to 5 because if $c = 1$, then SL can also perform well (but with less memory footprint than SBF and MBF); and if c is too large, then the false positive rate is high. The reference count further depends on other factors such as intersection size and the number of lists: it tends to be smaller for smaller intersection size and more lists.

3. Parallelism is very important to improve the performance of skipping-based intersection algorithms because it can fully utilize SSD's I/O bandwidth if the underlying I/O access

pattern is page-by-page. That is, page-level skipping must be combined with parallelism for high performance.

4. With cache, it still makes sense to skip unnecessary pages in a parallel manner unless all the lists are cached in main memory.

5. With compression, it is also beneficial for page-level skipping although it becomes more difficult when compared to skipping over uncompressed lists.

6. List size does not necessarily affect the effectiveness of skipping on SSDs.

7. List size ratio significantly affects the effectiveness of skipping on SSDs. The higher the ratio is, the more efficient the skipping becomes.

8. Intersection ratio does not necessarily affect the performance of skip pointer based algorithms such as SL and ZZ but it affects bloom filter based algorithms such as SBF and MBF.

9. The number of lists *k* can also affect the performance of skipping because it makes the list size ratio more skew as the intersection size shrinks quickly with top few lists being intersected. The higher the *k* is, the more efficient the skipping becomes.

3.7.2 Recommendations

Although the actual performance of different algorithms depends on many factors, generally we recommend SL (with multiple threads) as an SSD-optimized intersection algorithm. It has high performance, takes small memory footprint, and is also easy to be integrated into existing systems. In most cases, it is faster than the baseline BL. Even in the worst case when SL cannot skip any data pages, SL performs similar to BL.[15] For SBF and MBF, we do not provide strong

[15]The disadvantage of SL is the extra space overhead for the skip pointers. But they are significantly smaller than the original lists since each data page maintains only one skip pointer.

recommendations for SSD-based intersection because (1) their performance can vary significantly on different queries and datasets; (2) they may consume a high amount of memory to store bloom filters.

3.7.3 Lessons

We provide the lessons that people can learn from this work.

1. Although simply replacing HDDs by SSDs and directly running existing HDD-optimized intersection algorithms on the SSD can improve performance (since SSDs are faster than HDDs), SSDs are highly underutilized, because the design decisions are still made for HDDs. Thus, we strongly recommend practitioners to re-optimize their systems explicitly for SSDs.

2. Parallel I/O skipping, which previously cannot improve list intersection in HDD-resident systems, now becomes compelling in SSD-resident systems.

3. Do use parallelism on SSDs if data is accessed page-by-page, because that can significantly reduce the execution time.

4. It is still worthwhile to consider skipping on SSDs for intersection even when the lists are compressed or cached.

5. For HDD-based intersection, use BL and do not use skipping-based algorithms (e.g., SL, ZZ, SBF, and MBF).

Chapter 3 contains material from “Evaluating List Intersection on SSDs for Parallel I/O Skipping” by Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson, which was submitted for publication. The dissertation author was the primary investigator of the paper.

Chapter 4

Index-based High-dimensional Cosine Threshold Querying with Optimality Guarantees

4.1 Introduction

Given a database of vectors, a cosine threshold query asks for all database vectors with cosine similarity to a query vector above a given threshold. This problem arises in many applications including document retrieval [BCH$^+$03], image search [KG09], recommender systems [LCYM17] and mass spectrometry. For example, in mass spectrometry, billions of spectra are generated for the purpose of protein analysis [AM16, LDE$^+$07, WB13]. Each spectrum is a collection of key-value pairs where the key is the mass-to-charge ratio of an ion contained in the protein and the value is the intensity of the ion. Essentially, each spectrum is a high-dimensional, non-negative and sparse vector with $\sim$2000 dimensions where $\sim$100 coordinates are non-zero.

Cosine threshold queries play an important role in analyzing such spectra repositories. Example questions include "is the given spectrum similar to any spectrum in the database?", spectrum identification (when matching query spectra against reference spectra), or clustering (when matching pairs of unidentified spectra) or metadata queries (when searching for public datasets containing matching spectra, even if obtained from different types of samples). For such applications with a large vector database, it is critically important to process cosine threshold queries efficiently – this is the fundamental topic addressed in this work.

Definition 2 (Cosine Threshold Query) *Let $\mathcal{D}$ be a collection of high-dimensional, non-negative, unit vectors;* $\mathbf{q}$ *be a query vector;* θ *be a threshold* $0 < \theta \leq 1$. *Then the cosine threshold query returns the vector set* $\mathcal{R} = \{\mathbf{s} | \mathbf{s} \in \mathcal{D}, \texttt{cos}(\mathbf{q}, \mathbf{s}) \geq \theta\}$. *A vector* $\mathbf{s}$ *is called* θ-similar *to the query* $\mathbf{q}$ *if* $\texttt{cos}(\mathbf{q}, \mathbf{s}) \geq \theta$ *and the* score *of* $\mathbf{s}$ *is the value* $\texttt{cos}(\mathbf{q}, \mathbf{s})$ *when the query* $\mathbf{q}$ *is understood from the context.*

In the literature, cosine threshold querying is a case of the Cosine Similarity Search (CSS) area [TG16, AK14, LCYM17], where other aspects like approximate answers, top-k queries and similarity join are considered. Our work considers specifically CSS with exact, threshold and single-vector queries, which is the case of interest to many applications including mass spectrometry.

Because of the unit-vector assumption, the scoring function cos computes essentially the dot product $\mathbf{q} \cdot \mathbf{s}$. When the assumption is lifted, the problem is equivalent to *inner product threshold querying*, which is of interest in its own right. Related work on cosine and inner product similarity search is summarized in Section 4.10.

A TA-like baseline index and algorithm and its shortcomings. We show that the TA algorithm, by Fagin et al. [FLN01], provides an appropriate starting point for optimizing the processing of cosine threshold queries despite the following differences in the problem statement:

1. The TA's problem asks for the vectors with the top-k scores according to a monotonic query function $F(\mathbf{s})$. In contrast, the cosine threshold queries use $\texttt{cos}(\mathbf{q}, \mathbf{s})$, which can be viewed

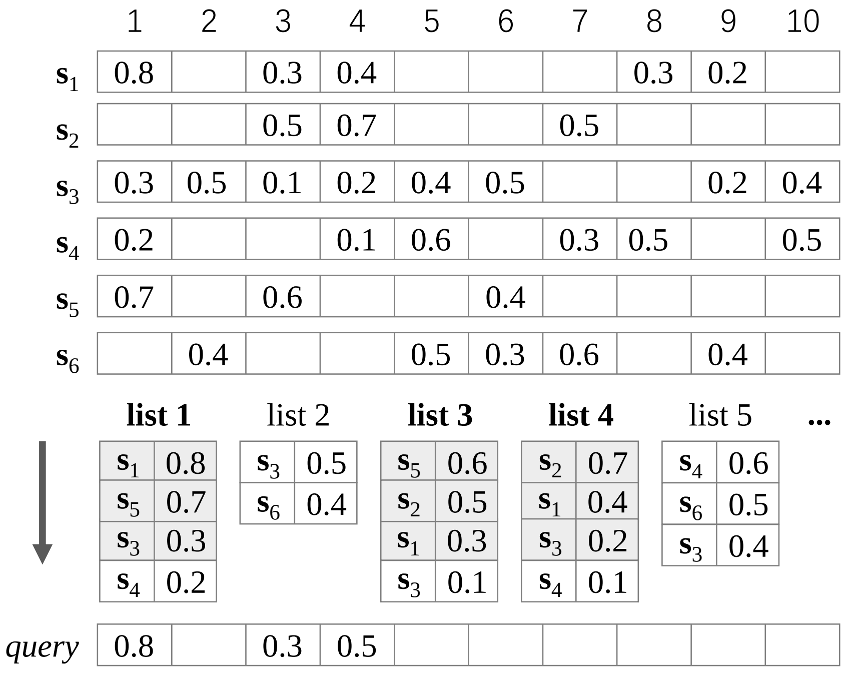
as a particular family of functions $F(\mathbf{s}) = \mathbf{s} \cdot \mathbf{q}$ parameterized by $\mathbf{q}$ when $|\mathbf{s}| = 1$. In addition, they return all $\mathbf{s}$ whose score exceeds the threshold θ.

2. The TA's problem poses no constraint on the data. In contrast, the cosine threshold queries assume a database of unit vectors. Note, the unit vector assumption does not really restrict the applicability of the proposed techniques, as we can indirectly produce unit vectors by precomputing the weight of each vector.

A *baseline* index and algorithm inspired by TA can answer cosine threshold queries exactly without a full scan of the vector database for each query. In addition, the baseline algorithm enjoys the same instance optimality guarantee as the original TA. This baseline is created as follows. First, identically to the TA, the baseline index consists of one sorted list for each of the d dimensions. In particular, the i-th sorted list has pairs $(\mathsf{ref}(\mathbf{s}), \mathbf{s}[i])$, where $\mathsf{ref}(\mathbf{s})$ is a reference to the vector $\mathbf{s}$ and $\mathbf{s}[i]$ is its value on the i-th dimension. The list is sorted in descending order of $\mathbf{s}[i]$.[1]

Next, the baseline, like the TA, proceeds into a *gathering phase* during which it collects a complete set of references to candidate result vectors. The TA shows that gathering can be achieved by reading the d sorted lists from top to bottom and terminating early when a *stopping condition* is finally satisfied. The condition guarantees that any vector that has not been seen yet has no chance of being in the query result. The baseline makes a straightforward change to the TA's stopping condition to adjust for the difference between the TA's top-k requirement and the threshold requirement of the cosine threshold queries. In particular, in each round the baseline algorithm has read the first b entries of each index. (Initially it is $b = 0$.) If it is the case that $\mathtt{cos}(\mathbf{q}, [L_1[b], \ldots, L_d[b]]) < \theta$ then it is guaranteed that the algorithm has already read (the references to) all the possible candidates and thus it is safe to terminate the gathering phase, see Figure 4.1 for an example. Every vector $\mathbf{s}$ that appeared in the j-th entry, $j < b$, of a list is a candidate.

[1]There is no need to include pairs with zero values in the list.



	1	2	3	4	5	6	7	8	9	10
$\mathbf{s}_1$	0.8		0.3	0.4				0.3	0.2	
$\mathbf{s}_2$			0.5	0.7			0.5			
$\mathbf{s}_3$	0.3	0.5	0.1	0.2	0.4	0.5			0.2	0.4
$\mathbf{s}_4$	0.2			0.1	0.6		0.3	0.5		0.5
$\mathbf{s}_5$	0.7		0.6			0.4				
$\mathbf{s}_6$		0.4			0.5	0.3	0.6		0.4	

list 1		list 2		**list 3**		**list 4**		list 5		**...**
$\mathbf{s}_1$	0.8	$\mathbf{s}_3$	0.5	$\mathbf{s}_5$	0.6	$\mathbf{s}_2$	0.7	$\mathbf{s}_4$	0.6	
$\mathbf{s}_5$	0.7	$\mathbf{s}_6$	0.4	$\mathbf{s}_2$	0.5	$\mathbf{s}_1$	0.4	$\mathbf{s}_6$	0.5	
$\mathbf{s}_3$	0.3			$\mathbf{s}_1$	0.3	$\mathbf{s}_3$	0.2	$\mathbf{s}_3$	0.4	
$\mathbf{s}_4$	0.2			$\mathbf{s}_3$	0.1	$\mathbf{s}_4$	0.1			

	1	2	3	4	5	6	7	8	9	10
query	0.8		0.3	0.5						

Figure 4.1: An example of cosine threshold query with six 10-dimensional vectors

In the next phase, called the *verification* phase, the baseline algorithm (again like TA) retrieves the candidate vectors from the database and checks which ones actually score above the threshold.

For inner product queries, the baseline algorithm's gathering phase benefits from the same $d \cdot \mathsf{OPT}$ instance optimality guarantee as the TA. Namely, the gathering phase will access at most $d \cdot \mathsf{OPT}$ entries, where OPT is the optimal number of entries accessed by any traversal algorithm for the particular input instance.

There is an obvious optimization: Only the k dimensions that have non-zero values in the query vector $\mathbf{q}$ should participate in query processing – this leads to a $k \cdot \mathsf{OPT}$ guarantee.[2] But even this guarantee loses its practical value when k is a large number. In the mass spectrometry scenario k is $\sim$100. In document similarity and image similarity cases it is even higher.

For cosine threshold queries, the $k \cdot \mathsf{OPT}$ guarantee no longer holds. The baseline fails to

[2]This optimization is equally applicable to the TA's problem: Scan only the lists that correspond to dimensions that actually affect the function F.

utilize the unit vector constraint to reach the stopping condition faster, resulting in an unbounded gap from OPT because of the unnecessary accesses (see remark in Section 4.4.1). [3] Furthermore, the baseline fails to utilize the skewing of the values in the vector's coordinates (both of the database's vectors and of the query vector) and the linearity of the similarity function. Intuitively, if the query's weight is concentrated on a few coordinates, the query processing should overweight the respective lists and may, thus, reach the stopping condition much faster than reading all relevant lists in tandem.

We retain the baseline's index and the gathering-verification structure which captures the family of TA-like algorithms. The algorithmic framework, which we refer to as Gathering-Verification, is reviewed and shown to be appropriate for cosine threshold queries in Section 4.2. Within the framework, we reconsider

1. *Traversal strategy optimization*: A *traversal strategy* determines the order in which the gathering phase proceeds in the lists. In particular, we allow the gathering phase to move deeper in some lists and less deep in others. For example, the gathering phase may have read at some point $b_1 = 106$ entries from the first list, $b_2 = 523$ entries from the second list, etc. Multiple traversal strategies are possible and, generally, each traversal strategy will reach the stopping condition with a different configuration of $[b_1, b_2, \ldots, b_n]$. The traversal strategy optimization problem asks that we efficiently identify a traversal path that minimizes the access cost $\sum_{i=1}^{d} b_i$. To enable such optimization, we will allow lightweight additions to the baseline index.

2. *Stopping condition optimization*: We reconsider the stopping condition so that it takes into account (a) the specifics of the cos function and (b) the unit vector constraint. Moreover, since the stopping condition is tested frequently during the gathering phase, it has to decide very fast whether the traversal can stop. Notice that the stopping condition optimization

[3] Notice, the unit vector constraint enables inference about the collective weight of the yet unseen coordinates of a vector.

problem is independent of the traversal strategy and its optimal solution is not dependent on skewness assumptions about the data.

Contributions and summary of results.

- We present a stopping condition for early termination of the index traversal (Section 4.4). We show that the stopping condition is *complete* and *tight*, informally meaning that (1) for any traversal strategy, the gathering phase will produce a candidate set containing all the vectors θ-similar to the query, and (2) the gathering terminates as soon as no more θ-similar vectors can be found (Theorem 4). In contrast, the stopping condition of the (TA-inspired) baseline is complete but not tight (Theorem 3). The proposed stopping condition takes into account that all database vectors are normalized and reduces the problem to solving a special quadratic program (Equation 4.4) that guarantees both completeness and tightness. While the new stopping condition prunes further the set of candidates, it can also be efficiently computed in $O(\log d)$ time using incremental maintenance techniques presented in Section 4.4.3.
- We introduce the *hull-based* traversal strategies that exploit the skewness of the data (Section 4.5). In particular, skewness implies that each sorted list L_i is "mostly convex", meaning that the shape of L_i is approximately the *lower convex hull* constructed from the set of points of L_i. This technique is quite general, as it can be extended to the class of *decomposable functions* which have the form

$$F(\mathbf{s}) = f_1(\mathbf{s}[1]) + \ldots + f_d(\mathbf{s}[d])$$

where each f_i is non-decreasing.[4] Consequently, we provide the following optimality guarantee for inner product threshold queries: The number of accesses executed by the gathering phase (i.e., $\sum_{i=1}^{d} b_i$) is at most $\mathsf{OPT} + c$ (Theorem 6 and Corollary 1), where OPT

[4]The inner product threshold problem is the special case where $f_i(\mathbf{s}[i]) = q_i \cdot \mathbf{s}[i]$.

Table 4.1: Summary of theoretical results

	Stopping Condition		Traversal Strategy	
	Baseline	*This work*	*Baseline*	*This work*
Inner Product	Tight		$d \cdot \mathsf{OPT}$	$\mathsf{OPT} + c$
Cosine	Not tight	Tight	NA	$\mathsf{OPT}(\theta - \varepsilon) + c$

is the number of accesses by the optimal strategy and c is the max distance between two vertices in the lower convex hull. Experiments show that in the mass spectrometry case, c is a very small fraction of the accesses.

- Despite the fact that cosine and its tight stopping condition are not decomposable, we show that the hull-based strategy can be adapted to cosine threshold queries by approximating the tight stopping condition with a carefully chosen decomposable function. We show that when the approximation is at most ε-away from the actual value, the access cost is at most $\mathsf{OPT}(\theta - \varepsilon) + c$ (Theorem 7) where $\mathsf{OPT}(\theta - \varepsilon)$ is the optimal access cost on the same query $\mathbf{q}$ with the threshold lowered by ε and c is a constant similar to the above decomposable cases. Experiments show that the adjustment ε is very small in practice, e.g., 0.1. We summarize these new results in Table 4.1.

4.2 Algorithmic framework

In this section, we present a Gathering-Verification algorithmic framework to facilitate optimizations in different components of an algorithm with a TA-like structure. We start with notations summarized in Table 4.2.

To support fast query processing, we build an index for the database vectors similar to the original TA. The basic index structure consists of a set of 1-dimensional sorted lists (a.k.a inverted lists in web search [BCH+03]) where each list corresponds to a vector dimension and contains vectors having non-zero values on that dimension, as mentioned earlier in Section 4.1.

Table 4.2: Notation

$\mathcal{D}$	the vector database
d	the number of dimensions
$\mathbf{s}$ (bold font)	a data vector
$\mathbf{q}$ (bold font)	a query vector
$\mathbf{s}[i]$ or s_i	the i-th dimensional value of $\mathbf{s}$
$\lvert\mathbf{s}\rvert$	the L1 norm of $\mathbf{s}$
$\lVert\mathbf{s}\rVert$	the L2 norm of $\mathbf{s}$
θ	the similarity threshold
$\mathtt{cos}(\mathbf{p},\mathbf{q})$	the cosine of two vectors $\mathbf{p}$ and $\mathbf{q}$
L_i	an inverted list built on the i-th dimension
$\mathbf{b} = (b_1,\ldots,b_d)$	a position vector
$L_i[b_i]$	the b_i-th value of L_i
$L[\mathbf{b}]$	the vector $(L_1[b_1],\ldots,L_d[b_d])$

Formally, for each dimension i, L_i is a list of pairs $\{(\mathsf{ref}(\mathbf{s}),\mathbf{s}[i]) \mid \mathbf{s} \in \mathcal{D} \wedge \mathbf{s}[i] > 0\}$ sorted in descending order of $\mathbf{s}[i]$ where $\mathsf{ref}(\mathbf{s})$ is a reference to the vector $\mathbf{s}$ and $\mathbf{s}[i]$ is its value on the i-th dimension. In the interest of brevity, we will often write $(\mathbf{s},\mathbf{s}[i])$ instead of $(\mathsf{ref}(\mathbf{s}),\mathbf{s}[i])$. As an example in Figure 4.1, the list L_1 is built for the first dimension and it includes 4 entries: $(\mathbf{s}_1,0.8)$, $(\mathbf{s}_5,0.7)$, $(\mathbf{s}_3,0.3)$, $(\mathbf{s}_4,0.2)$ because $\mathbf{s}_1$, $\mathbf{s}_5$, $\mathbf{s}_3$ and $\mathbf{s}_4$ have non-zero values on the first dimension.

Next, we show the Gathering-Verification framework (Algorithm 5) that operates on the index structure. The framework includes two phases: the gathering phase and the verification phase.

Gathering phase (line 1 to line 5). The goal of the gathering phase is to collect a complete set of candidate vectors while minimizing the number of accesses to the sorted lists. The algorithm maintains a *position vector* $\mathbf{b} = (b_1,\ldots,b_d)$ where each b_i indicates the current position in the inverted list L_i. Initially, the position vector $\mathbf{b}$ is $(0,\ldots,0)$. Then it traverses the lists according to a *traversal strategy* that determines the list (say L_i) to be accessed next (line 3). Then it advances the pointer b_i by 1 (line 4) and adds the vector $\mathbf{s}$ referenced in the entry $L_i[b_i]$ to a candidate pool $\mathcal{C}$ (line 5). The traversal strategy is usually stateful, which means that its decision is made based on information that has been observed up to position $\mathbf{b}$ and its past decisions. For example, a

Algorithm 5: Gathering-Verification Framework

```
  input  : ($\mathcal{D}$, $\{L_i\}_{1\le i\le d}$, $\mathbf{q}$, $\theta$)
  output : $\mathcal{R}$
  /* Gathering phase                                              */
1 Initialize $\mathbf{b} = (b_1,\ldots,b_d) = (0,\ldots,0)$;
  // $\varphi(\cdot)$ is the stopping condition
2 while $\varphi(\mathbf{b})$ = false do
      // $\mathcal{T}(\cdot)$ is the traversal strategy to determine which list to access
         next
3     $i \leftarrow \mathcal{T}(\mathbf{b})$;
4     $b_i \leftarrow b_i + 1$;
5     Put the vector $\mathbf{s}$ in $L_i[b_i]$ to the candidate pool $\mathcal{C}$;
  /* Verification phase                                           */
6 $\mathcal{R} \leftarrow \{\mathbf{s} | \mathbf{s} \in \mathcal{C} \wedge \cos(\mathbf{q},\mathbf{s}) \ge \theta\}$;
7 return $\mathcal{R}$;
```

strategy may decide that it will make the next 20 moves along dimension 6 and thus it needs state in order to remember that it has already committed to 20 moves on dimension 6.

The gathering phase terminates once a *stopping condition* is met. Intuitively, based on the information that has been observed in the index, the stopping condition checks if a complete set of candidates has already been found.

Next, we formally define stopping conditions and traversal strategies. As mentioned above, the input of the stopping condition and the traversal strategy is the information that has been observed up to position $\mathbf{b}$, which is formally defined as follows.

Definition 3 *Let* $\mathbf{b}$ *be a position vector on the inverted index* $\{L_i\}_{1\le i\le d}$ *of a database* $\mathcal{D}$. *The partial observation at* $\mathbf{b}$, *denoted as* $\mathcal{L}(\mathbf{b})$, *is a collection of lists* $\{\hat{L}_i\}_{1\le i\le d}$ *where for every* $1 \le i \le d$, $\hat{L}_i = [L_i[1],\ldots,L_i[b_i]]$.

Definition 4 *Let* $\mathcal{L}(\mathbf{b})$ *be a partial observation and* $\mathbf{q}$ *be a query with similarity threshold* θ. *A* ***stopping condition*** *is a boolean function* $\varphi(\mathcal{L}(\mathbf{b}),\mathbf{q},\theta)$ *and a* ***traversal strategy*** *is a function* $\mathcal{T}(\mathcal{L}(\mathbf{b}),\mathbf{q},\theta)$ *whose domain is* $[d]$[5]. *When clear from the context, we denote them simply by* $\varphi(\mathbf{b})$ *and* $\mathcal{T}(\mathbf{b})$ *respectively.*

[5] $[d]$ is the set $\{1,\ldots,d\}$

Verification phase (line 6). The verification phase examines each candidate vector $\mathbf{s}$ seen in the gathering phase to verify whether $\texttt{cos}(\mathbf{q},\mathbf{s}) \geq \theta$ by accessing the database. Various techniques [TG16, AK15, LCYM17] have been proposed to speed up this process. Essentially, instead of accessing all the d dimensions of each $\mathbf{s}$ and $\mathbf{q}$ to compute exactly the cosine similarity, these techniques decide θ-similarity by performing a partial scan of each candidate vector. We review these techniques, which we refer to as *partial verification*, in Section 4.6. Additionally, as a novel contribution, we show that in the presence of data skewness, partial verification can have a near-constant performance guarantee (Theorem 8) for each candidate.

4.3 Remark on the suitability of TA-like algorithms

One may wonder whether algorithms that start the gathering phase NOT from the top of the inverted lists may outperform the best TA-like algorithm. In particular, it appears tempting to start the gathering phase from the closest to q_i point in each inverted list and traverse towards the two ends of each list. Next, we prove why this idea can lead to poor performance. In particular, we prove that in a general setting, the computation of a tight and complete stopping condition (formally defined in Definition 5 and 6) becomes NP-HARD since it needs to take into account constraints from two pointers (forward and backward) for each inverted list. Furthermore, in applications like mass spectrometry, the data skewing leads to small savings from pruning the top area of each list, since the top area is sparsely populated - unlike the densely populated bottom area of each list. Thus it is not justified to use an expensive gathering phase algorithm for small savings.

As already reviewed in Section 4.1, the gathering phase stops when the cosine function score at the current position $\mathbf{b}$ is below the input threshold θ. Formally, the baseline stopping condition φ_{BL} is

$$\varphi_{\mathsf{BL}}(\mathbf{b}) = \left(\sum_{i=1}^{d} q_i \cdot L_i[b_i] < \theta\right). \tag{4.1}$$

To determine the order of accessing the inverted lists, TA advances all the pointers at equal rate. An obvious optimization is to move only the pointers whose dimension has non-zero values in $\mathbf{q}$. Let $\mathsf{nz} = \{i | i \in [d] \wedge q_i > 0\}$ be the list of all non-zero dimensions and let $m = |\mathsf{nz}|$. The baseline traversal strategy $\mathcal{T}_{\mathsf{BL}}$ is the following:

$$\mathcal{T}_{\mathsf{BL}}(\mathbf{b}) = \mathsf{nz}[\ |\mathbf{b}| \bmod m\ ], \tag{4.2}$$

so that each inverted list of a non-zero dimension is advanced at equal speed. By the classic result of [FLN01],

Theorem 2 *For inner product threshold queries (without the normalization constraint), the access cost of the* TA*-inspired baseline is at most* $m \cdot \mathsf{OPT}$ *where* m *is the query's number of non-zero dimensions and* OPT *is the optimal access cost.*

For cosine threshold queries, since the baseline condition φ_{BL} is not tight (Theorem 3), the same instance optimality bound does not hold.

Section 4.10.1 reviews additional prior work ideas [TG16, TGM15] that avoid traversing some top/bottom regions of the inverted index. Such ideas may provide additional optimizations to TA-like algorithms in variations and/or restrictions of the problem (e.g., a restriction that the threshold is very high) and thus they present future work opportunities in closely related problems.

4.4 Stopping condition

In this section, we introduce a fine-tuned stopping condition that satisfies the tight and complete requirements to early terminate the index traversal. These two requirements can be described informally as follows.

First, the stopping condition has to guarantee *completeness* (Definition 5), i.e. when the stopping condition φ holds on a position $\mathbf{b}$, the candidate set $\mathcal{C}$ must contain all the true results. Note that since the input of φ is the partial observation at $\mathbf{b}$, we must guarantee that for all possible databases $\mathcal{D}$ consistent with the partial observation $\mathcal{L}(\mathbf{b})$, the candidate set $\mathcal{C}$ contains all vectors in $\mathcal{D}$ that are θ-similar to the query $\mathbf{q}$. This is equivalent to require that if a unit vector $\mathbf{s}$ is found below position $\mathbf{b}$ (i.e. $\mathbf{s}$ does not appear above $\mathbf{b}$), then $\mathbf{s}$ is NOT θ-similar to $\mathbf{q}$. We formulate this as follows.

Definition 5 (Completeness) *Given a query* $\mathbf{q}$ *with threshold* θ*, a position vector* $\mathbf{b}$ *on index* $\{L_i\}_{1 \le i \le d}$ *is complete iff for every unit vector* $\mathbf{s}$*,* $\mathbf{s} < L[\mathbf{b}]$ *implies* $\mathbf{s} \cdot \mathbf{q} < \theta$*. A stopping condition* $\varphi(\cdot)$ *is complete iff for every* $\mathbf{b}$*,* $\varphi(\mathbf{b}) = \texttt{True}$ *implies that* $\mathbf{b}$ *is complete.*

The second requirement of the stopping condition is *tightness*. It is desirable that the algorithm terminates immediately once the candidate set $\mathcal{C}$ contains a complete set of candidates, such that no additional unnecessary access is made. This can reduce not only the number of index accesses but also the candidate set size, which in turn reduces the verification cost. This is formally defined as follows.

Definition 6 (Tightness) *A stopping condition* $\varphi(\cdot)$ *is tight iff for every complete position vector* $\mathbf{b}$*,* $\varphi(\mathbf{b}) = \texttt{True}$.

It is desirable that a stopping condition be both complete and tight. In Section 4.4.1, we first show that the baseline stopping condition is complete but not tight. Then in Section 4.4.2, we present a new stopping condition that is both complete and tight.

4.4.1 The baseline stopping condition is not tight

Theorem 3 *The* baseline *stopping condition*

$$\varphi_{\mathsf{BL}}(\mathbf{b}) = \left(\sum_{i=1}^{d} q_i \cdot L_i[b_i] < \theta \right) \tag{4.3}$$

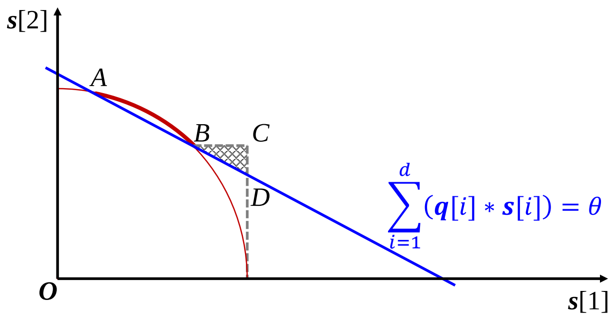


Figure 4.2: A 2-d example of φ_{BL}'s non-tightness

is complete but not tight.

Proof 2 *For every position vector* $\mathbf{b}$, $\varphi_{\mathsf{BL}}(\mathbf{b}) = \texttt{True}$ *implies* $\mathbf{q} \cdot L[\mathbf{b}] < \theta$. *So for every* $\mathbf{s} < L[\mathbf{b}]$, *we also have* $\mathbf{q} \cdot \mathbf{s} < \theta$ *so* φ_{BL} *is complete.*

To show the non-tightness, it is sufficient to show that for some position vector $\mathbf{b}$ *where* $\mathbf{b}$ *is complete,* $\varphi_{\mathsf{BL}}(\mathbf{b})$ *is* `False` *so the traversal continues.*

We illustrate a counterexample in Figure 4.2 with two dimensions (i.e., $d = 2$*). Given a query* $\boldsymbol{q}$*, all possible* θ*-similar vectors form a hyper-surface defining the set* $\mathsf{ans} = \{\mathbf{s} | \ \|\mathbf{s}\| = 1, \sum_{i=1}^{d} q_i \cdot s_i \geq \theta\}$*. In Figure 4.2,* $\|\mathbf{s}\| = 1$ *is the circular surface and* $\sum_{i=1}^{d} q_i \cdot s_i \geq \theta$ *is a half-plane so the set of points* ans *is the arc* $\widehat{AB}$*.*

By definition, a position vector $\mathbf{b}$ *is complete if the set* $\{\mathbf{s}|\mathbf{s} < L[\mathbf{b}]\}$ *contains no point in* ans*. A position vector* $\mathbf{b}$ *satisfies* φ_{BL} *iff the point* $L[\mathbf{b}]$ *is above the plane* $\sum_{i=1}^{d} q_i \cdot s_i = \theta$*. It is clear from Figure 4.2 that if the point* $L[\mathbf{b}]$ *locates at the region BCD, then* $\{\mathbf{s}|\mathbf{s} < L[\mathbf{b}]\}$ *contains no point in* $\widehat{AB}$ *and is above the half-plane* $\sum_{i=1}^{d} q_i \cdot s_i = \theta$*. There exists a database of 2-d vectors such that* $L[\mathbf{b}]$ *resides in the BCD region for some position* $\mathbf{b}$*, so the stopping condition* φ_{BL} *is not tight.*

Remark. The baseline stopping condition φ_{BL} is not tight because it does not take into account that all vectors in the database are unit vectors. In fact, one can show that φ_{BL} is tight and complete for inner product queries where the unit vector assumption is lifted. In addition, since φ_{BL} is not tight, any traversal strategy that works with φ_{BL} has no optimality guarantee in general since there can be a gap of arbitrary size between the stopping position by φ_{BL} and the one that is tight (i.e. there can be arbitrarily many points in the region *BCD*).

4.4.2 A tight stopping condition

To guarantee tightness, one can check at every snapshot during the traversal whether the current position vector $\mathbf{b}$ is complete and stop once the condition is true. However, directly testing the completeness is impractical since it is equivalent to testing whether there exists a real vector $\mathbf{s} = (s_1, \ldots, s_d)$ that satisfies the following following set of quadratic constraints:

$$\begin{cases} \sum_{i=1}^{d} s_i \cdot q_i \geq \theta \\ s_i \leq L_i[b_i], \ \forall\, i \in [d] \\ \sum_{i=1}^{d} s_i^2 = 1. \end{cases} \tag{4.4}$$

We denote by $\mathbf{C}(\mathbf{b})$ (or simply $\mathbf{C}$) the set of $\mathbb{R}^d$ points defined by the above constraints. The set $\mathbf{C}(\mathbf{b})$ is infeasible (i.e. there is no satisfying $\mathbf{s}$) if and only if $\mathbf{b}$ is complete, but directly testing the feasibility of $\mathbf{C}(\mathbf{b})$ requires an expensive call to a quadratic programming solver. Depending on the implementation, the running time can be exponential or of high-degree polynomial [BV04]. We address this challenge by deriving an equivalently strong stopping condition that guarantees tightness and is efficiently testable:

Theorem 4 *Let τ be the solution of the equation*

$$\sum_{i=1}^{d} \min\{q_i \cdot \tau, L_i[b_i]\}^2 = 1. \tag{4.5}$$

The following stopping condition is tight and complete:

$$\varphi_{\mathsf{TC}}(\mathbf{b}) = \left(\sum_{i=1}^{d} \min\{q_i \cdot \tau, L_i[b_i]\} \cdot q_i < \theta \right). \tag{4.6}$$

Proof 3 *The tight and complete stopping condition is obtained by applying the Karush-Kuhn-Tucker (KKT) conditions [KT14] for solving nonlinear programs. We first formulate the set of constraints in (4.4) as an optimization problem over* $\mathbf{s}$*:*

$$\text{maximize} \quad \sum_{i=1}^{d} s_i \cdot q_i \quad \text{subject to} \quad \begin{aligned} &\sum_{i=1}^{d} s_i^2 = 1, \\ &s_i \leq L_i[b_i],\, i \in [d] \end{aligned} \tag{4.7}$$

So checking whether $\mathbf{C}$ *is feasible is equivalent to verifying whether the maximal* $\sum_{i=1}^{d} s_i \cdot q_i$ *is at least* θ*. So it is sufficient to show that* $\sum_{i=1}^{d} s_i \cdot q_i$ *is maximized when* $s_i = \min\{q_i \cdot \tau, L_i[b_i]\}$ *as specified above.*

The KKT conditions of the above maximization problem specify a set of necessary conditions that the optimal $\mathbf{s}$ *needs to satisfy. More precisely, let*

$$L(\mathbf{s}, \mu, \lambda) = \sum_{i=1}^{d} s_i q_i - \sum_{i=1}^{d} \mu_i (L_i[b_i] - s_i) - \lambda \left(\sum_{i=1}^{d} s_i^2 - 1 \right)$$

be the Lagrangian of (4.7) where $\lambda \in \mathbb{R}$ *and* $\mu \in \mathbb{R}^d$ *are the Lagrange multipliers. Then,*

Lemma 1 (derived from KKT) *The optimal* $\mathbf{s}$ *in (4.7) satisfies the following conditions:*

$$\begin{aligned} &\nabla_{\mathbf{s}} L(\mathbf{s}, \mu, \lambda) = 0 && \textit{(Stationarity)} \\ &\mu_i \geq 0,\ \forall\, i \in [d] && \textit{(Dual feasibility)} \\ &\mu_i (L_i[b_i] - s_i) = 0,\ \forall\, i \in [d] && \textit{(Complementary slackness)} \end{aligned}$$

in addition to the constraints in (4.7) (called the Primal feasibility *conditions).*

By the Complementary slackness condition, for every i, if $\mu_i \neq 0$ *then* $s_i = L_i[b_i]$*. If* $\mu_i \neq 0$*, then from the Stationarity condition, we know that for every i,* $q_i + \mu_i - \lambda \cdot s_i = 0$ *so* $s_i = q_i/\lambda$*. Thus, the value of* s_i *is either* $L_i[b_i]$ *or* q_i/λ*.*

If $L_i[b_i] < q_i/\lambda$ *then since* $s_i \leq L_i[b_i]$*, the only possible case is* $s_i = L_i[b_i]$*. For the remaining* *dimensions, the objective function $\sum_{i=1}^{d} s_i \cdot q_i$ is maximized when each s_i is proportional to q_i, so $s_i = q_i/\lambda$. Combining these two cases, we have*

$$s_i = \min\{q_i/\lambda, L_i[b_i]\}.$$

Thus, for the λ that satisfies $\sum_{i=1}^{d} \min\{q_i/\lambda, L_i[b_i]\}^2 = 1$, the objective function $\sum_{i=1}^{d} s_i \cdot q_i$ is maximized when $s_i = \min\{q_i/\lambda,$
$L_i[b_i]\}$ for every i. The theorem is obtained by letting $\tau = 1/\lambda$.

Interpretation of φ_{TC}. The tight stopping condition φ_{TC} essentially provides an efficient way to compute the vector $\mathbf{s}$ below $L(\mathbf{b})$ with the maximum cosine similarity with the query $\mathbf{q}$. Let

$$\mathsf{MS}(L[\mathbf{b}]) = \sum_{i=1}^{d} \min\{q_i \cdot \tau, L_i[b_i]\} \cdot q_i \tag{4.8}$$

be the maximum similarity. At the beginning of the gathering phase, $b_i = 0$ for every i so $\mathsf{MS}(L[\mathbf{b}]) = 1$ as $\mathbf{s}$ is not constrained. The cosine score is maximized when $\mathbf{s} = \mathbf{q}$ where $\tau = 1$. During the gathering phase, as b_i increases, the upper bound $L_i[b_i]$ of each s_i decreases. When $L_i[b_i] < q_i$ for some i, s_i can no longer be q_i. Instead, s_i equals $L_i[b_i]$, the rest of $\mathbf{s}$ increases proportional to $\mathbf{q}$ and τ increases. As illustrated in Figure 4.3, during the traversal, the value of τ monotonically increases and the score $\mathbf{s}(L[\mathbf{b}])$ monotonically decreases. This is because the space for $\mathbf{s}$ becomes more constrained by $L(\mathbf{b})$ as the pointers move deeper in the inverted lists.

4.4.3 Efficient computation of φ_{TC} with incremental maintenance

Testing the tight and complete condition φ_{TC} requires solving τ in Equation (4.5), for which a direct application of the bisection method takes $O(d)$ time. Next, we provide a more efficient algorithm based on incremental maintenance which takes only $O(\log d)$ time for each test of φ_{TC}.

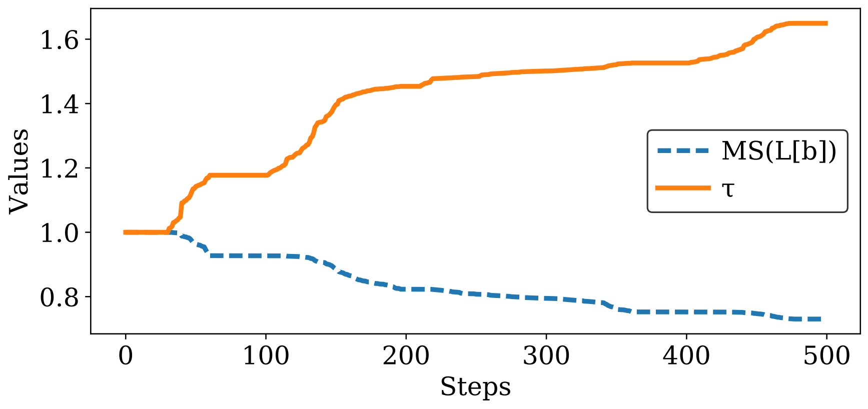


Figure 4.3: The distribution of τ and $\mathsf{MS}(L[\mathbf{b}])$

According to the proof of Theorem 4,

$$s_i = \begin{cases} L_i[b_i], & \tau \geq \dfrac{L_i[b_i]}{q_i}; \\ q_i \cdot \tau, & \text{otherwise.} \end{cases} \tag{4.9}$$

Wlog, suppose $\frac{L_1[b_1]}{q_1} \leq \cdots \leq \frac{L_d[b_d]}{q_d}$ and τ is in the range $[\frac{L_k[b_k]}{q_k}, \frac{L_{k+1}[b_{k+1}]}{q_{k+1}}]$ for some k. We have $s_i = L_i[b_i]$ for every $1 \leq i \leq k$ and $s_i = q_i \cdot \tau$ for $k < i \leq d$. So if we let $\mathsf{eval}(k, \tau)$ be the function

$$\mathsf{eval}(k,\tau) = \sum_{i=1}^{d} s_i^2 = \sum_{i=1}^{k} L_i[b_i]^2 + \sum_{i=k+1}^{d} q_i^2 \cdot \tau^2, \tag{4.10}$$

then for the largest k such that $\mathsf{eval}(k, L_k[b_k]/q_k) \leq 1$, τ can be computed by solving

$$\sum_{i=1}^{k} L_i[b_i]^2 + \sum_{i=k+1}^{d} q_i^2 \cdot \tau^2 = 1$$
$$\Rightarrow \quad \tau = \left(\frac{1 - \sum_{i=1}^{k} L_i[b_i]^2}{1 - \sum_{i=1}^{k} q_i^2} \right)^{1/2}.$$

Then, $\mathsf{MS}(L[\mathbf{b}])$ can be computed as follows:

$$\mathsf{MS}(L[\mathbf{b}]) = \sum_{i=1}^{k} L_i[b_i] \cdot q_i + (1 - \sum_{i=1}^{k} q_i^2) \cdot \tau \; . \tag{4.11}$$

Computing $\mathsf{MS}(L[\mathbf{b}])$ using the above approach directly requires that the $L_i[b_i]$'s are sorted in each step by $L_i[b_i]/q_i$, which requires $O(d \log d)$ time. However, this is still too expensive as the stopping condition is checked in every step. Fortunately, we show that $\mathsf{MS}(L[\mathbf{b}])$ can be incrementally maintained in $O(\log d)$ time as we describe below.

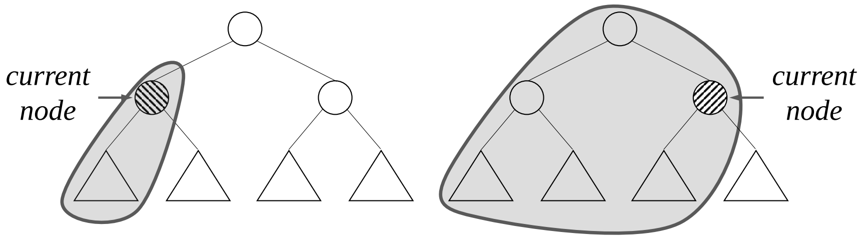
We use a binary search tree (BST) to maintain an order of the L_i's sorted by $L_i[b_i]/q_i$. The BST supports the following two operations:

- $\mathsf{update(i)}$: update $L_i[b_i] \to L_i[b_i + 1]$
- $\mathsf{compute()}$: return the value of $\mathsf{MS}(L[\mathbf{b}])$

The $\mathsf{compute()}$ operation essentially performs the binary search of finding the largest k mentioned above. To ensure $O(\log d)$ running time, we observe that from Equation (4.11) and (4.11), for any k, $\mathsf{MS}(L[\mathbf{b}])$ can be computed if $\sum_{i=1}^{k} L_i[b_i] \cdot q_i$, $\sum_{i=1}^{k} L_i[b_i]^2$, and $\sum_{i=1}^{k} q_i^2$ are available. Let T be the BST and each node in T is denoted as an integer n, meaning that the node represents the list L_{n}. We denote by $\mathsf{subtree(n)}$ the subtree of T rooted at node n, and maintain the following information for each node n:

- $\mathsf{n.key}$: $L_n[b_n]/q_n$,
- $\mathsf{n.LQ}$: $\sum_{i \in \mathsf{subtree(n)}} L_i[b_i] \cdot q_i$,
- $\mathsf{n.Q2}$: $\sum_{i \in \mathsf{subtree(n)}} q_i^2$ and
- $\mathsf{n.L2}$: $\sum_{i \in \mathsf{subtree(n)}} L_i[b_i]^2$.

Thus, whenever there is a move on the list L_i, the key of the node i (i.e., $L_i[b_i]/q_i$) will be updated. Then we can remove the node i from the tree and insert it again using the new key, which takes $O(\log d)$ time (with all the associated values being updated as well). To compute



(a) case 1 (b) case 2

Figure 4.4: An example of incremental maintenance

$\mathsf{MS}(L[\mathbf{b}])$, we need to handle two cases shown in Figure 4.4. In the first case, all the required information for computing $\mathsf{MS}(L[\mathbf{b}])$ are stored in the current node and its left subtree, while in the second case, the required information needs to be passed from the parent node to the current node to compute $\mathsf{MS}(L[\mathbf{b}])$. See Algorithm 6 for details.

```
Algorithm 6: compute(T)
1  $(\mathsf{LQ\_parent}, \mathsf{Q2\_parent}, \mathsf{L2\_parent}) \leftarrow (0,0,0)$;
2  $\mathsf{MS}(L[\mathbf{b}]) \leftarrow 1$ ;                    // $\mathsf{MS}(L[\mathbf{b}]) = 1$ if $\forall i\ \tau \leq L_i[b_i]/q_i$
3  $\mathsf{n} \leftarrow \mathsf{root}(T)$;
4  while $\mathsf{n} \neq \mathsf{null}$ do
5      $\mathsf{LQ} \leftarrow \mathsf{LQ\_parent} + \mathsf{n.left.LQ} + L_\mathsf{n}[b_\mathsf{n}] \cdot q_\mathsf{n}$;
6      $\mathsf{Q2} \leftarrow \mathsf{Q2\_parent} + \mathsf{n.left.Q2} + q_\mathsf{n}^2$;
7      $\mathsf{L2} \leftarrow \mathsf{L2\_parent} + \mathsf{n.left.L2} + L_\mathsf{n}[b_\mathsf{n}]^2$;
8      $\mathsf{f(n)} \leftarrow \mathsf{LQ} + (1 - \mathsf{Q2}) \cdot \mathsf{n.key}$;
9      if $\mathsf{f(n)} \leq 1$ then
10         $\tau \leftarrow ((1 - \mathsf{L2})/(1 - \mathsf{Q2}))^{1/2}$;
11         $\mathsf{MS}(L[\mathbf{b}]) \leftarrow \mathsf{LQ} + (1 - \mathsf{Q2}) \cdot \tau$;
12         $\mathsf{n} \leftarrow \mathsf{n.left}$;
13     else
14         $\mathsf{LQ\_parent} \leftarrow \mathsf{LQ\_parent} + \mathsf{n.left.LQ} + L_\mathsf{n}[b_\mathsf{n}] \cdot q_\mathsf{n}$;
15         $\mathsf{Q2\_parent} \leftarrow \mathsf{Q2\_parent} + \mathsf{n.left.Q2} + q_\mathsf{n}^2$;
16         $\mathsf{L2\_parent} \leftarrow \mathsf{L2\_parent} + \mathsf{n.left.L2} + L_\mathsf{n}[b_\mathsf{n}]^2$;
17         $\mathsf{n} \leftarrow \mathsf{n.right}$;
18 return $\mathsf{MS}(L[\mathbf{b}])$;
```

4.5 Near-optimal traversal strategy

Given the inverted lists index and a query, there can be many stopping positions that are both complete and tight. To optimize the performance, we need a traversal strategy that reaches one such position as fast as possible. Specifically, the goal is to design a traversal strategy $\mathcal{T}$ that minimizes $|\mathbf{b}| = \sum_{i=1}^{d} b_i$ where $\mathbf{b}$ is the first position vector satisfying the tight and complete stopping condition if $\mathcal{T}$ is followed. Minimizing $|\mathbf{b}|$ also reduces the number of collected candidates, which in turn reduces the cost of the verification phase. We call $|\mathbf{b}|$ the *access cost* of the strategy $\mathcal{T}$. Formally,

Definition 7 (Access Cost) *Given a traversal strategy $\mathcal{T}$, we denote by $\{\mathbf{b}_i\}_{i \geq 0}$ the sequence of position vectors obtained by following $\mathcal{T}$. The access cost of $\mathcal{T}$, denoted by $\mathsf{cost}(\mathcal{T})$, is the minimal k such that $\varphi_{\mathsf{TC}}(\mathbf{b}_k) = \mathtt{True}$. Note that $\mathsf{cost}(\mathcal{T})$ also equals $|\mathbf{b}_k|$.*

Definition 8 (Instance Optimality) *Given a database $\mathcal{D}$ with inverted lists $\{L_i\}_{1 \leq i \leq d}$, a query vector $\mathbf{q}$ and a threshold θ, the optimal access cost $\mathsf{OPT}(\mathcal{D}, \mathbf{q}, \theta)$ is the minimum $\sum_{i=1}^{d} b_i$ for position vectors $\mathbf{b}$ such that $\varphi_{\mathsf{TC}}(\mathbf{b}) = \mathtt{True}$. When it is clear from the context, we simply denote $\mathsf{OPT}(\mathcal{D}, \mathbf{q}, \theta)$ as $\mathsf{OPT}(\theta)$ or OPT.*

At a position $\mathbf{b}$, a traversal strategy makes its decision locally based on what has been observed in the inverted lists up to that point, so the capability of making globally optimal decisions is limited. As a result, traversal strategies are often designed as simple heuristics, such as the lockstep strategy in the baseline approach. The lockstep strategy has a $d \cdot \mathsf{OPT}$ near-optimal bound which is loose in the high-dimensionality setting.

In this section, we present a traversal strategy for cosine threshold queries with tighter near-optimal bound by taking into account that the index values are skewed in many realistic scenarios, such as mass spectrometry. We approach the (near-)optimal traversal strategy in two steps.

First, we consider the simplified case with the unit-vector constraint ignored so that the problem is reduced to inner product queries. We propose a general traversal strategy that relies on convex hulls pre-computed from the inverted lists during indexing. During the gathering phase, these convex hulls are accessed as auxiliary data during the traversal to provide information on the increase/decrease rate towards the stopping condition. The hull-based traversal strategy not only makes fast decisions (in $O(\log d)$ time) but is near-optimal (Corollary 1) under a reasonable assumption. In particular, we show that if the distance between any two consecutive convex hull vertices of the inverted lists is bounded by a constant c, the access cost of the strategy is at most $\mathsf{OPT} + c$. Experiments on real data show that this constant is small in practice.

The hull-based traversal strategy is quite general, as it applies to a large class of functions beyond inner product called the *decomposable functions*, which have the form $\sum_{i=1}^{d} f_i(s_i)$ where each f_i is a non-decreasing real function of a single dimension s_i. Obviously, for a fixed query $\mathbf{q}$, the inner product $\mathbf{q} \cdot \mathbf{s}$ is a special case of decomposable functions, where each $f_i(s_i) = q_i \cdot s_i$. We show that the near-optimality result for inner product queries can be generalized to any decomposable function (Theorem 6).

Next, in Section 4.5.5, we consider the cosine queries by taking the normalization constraint into account. Although the function $\mathsf{MS}(\cdot)$ used in the tight stopping condition φ_{TC} is not decomposable so the same technique cannot be directly applied, we show that the hull-based strategy can be adapted by approximating $\mathsf{MS}(\cdot)$ with a decomposable function. In addition, we show that with a properly chosen approximation, the hull-based strategy is near-optimal with a small adjustment to the input threshold θ, meaning that the access cost is bounded by $\mathsf{OPT}(\theta - \varepsilon) + c$ for a small ε (Theorem 7). Under the same experimental setting, we verify that ε is indeed small in practice.

4.5.1 Decomposable functions

We start with defining the decomposable functions for which the hull-based traversal strategies can be applied:

Definition 9 (Decomposable Function) *A decomposable function $F(\mathbf{s})$ is a d-dimensional real function where*

$$F(\mathbf{s}) = \sum_{i=1}^{d} f_i(s_i)$$

and each f_i is a non-decreasing real function.

Given a decomposable function F, the corresponding stopping condition is called a *decomposable condition*, which we define next.

Definition 10 (Decomposable condition) *A decomposable condition φ_F is a boolean function $\varphi_F(\mathbf{b}) = \big(F(L[\mathbf{b}]) < \theta\big)$ where F is a decomposable function and θ is a fixed threshold.*

When the unit vector constraint is lifted, the decomposable condition is tight and complete for any scoring function F and threshold θ. As a result, the goal of designing a traversal strategy for F is to have the access cost as close as possible to OPT when the stopping condition is φ_F.

4.5.2 The max-reduction traversal strategy

To illustrate the high-level idea of the hull-based approach, we start with a simple greedy traversal strategy called the *Max-Reduction* traversal strategy $\mathcal{T}_{\mathsf{MR}}(\cdot)$. The strategy works as follows: at each snapshot, move the pointer b_i on the inverted list L_i that results in the maximal reduction on the score $F(L[\mathbf{b}])$. Formally, we define

$$\begin{aligned}\mathcal{T}_{\mathsf{MR}}(\mathbf{b}) &= \underset{1\le i\le d}{\operatorname{argmax}}\,(F(L[\mathbf{b}]) - F(L[\mathbf{b}+\mathbf{1}_i])) \\ &= \underset{1\le i\le d}{\operatorname{argmax}}\,(f_i(L_i[b_i]) - f_i(L_i[b_i+1]))\end{aligned}$$

where $\mathbf{1}_i$ is the vector with 1 at dimension i and 0's else where. Such a strategy is reasonable since one would like $F(L[\mathbf{b}])$ to drop as fast as possible, so that once it is below θ, the stopping condition φ_F will be triggered and terminate the traversal.

It is obvious that there are instances where the max-reduction strategy can be far from optimal, but is it possible that it is optimal under some assumption? The answer is positive: if for every list L_i, the values of $f_i(L_i[b_i])$ are decreasing at decelerating rate, then we can prove that its access cost is optimal. We state this ideal assumption next.

Assumption 1 (Ideal Convexity) *For every inverted list L_i, let $\Delta_i[j] = f_i(L_i[j]) - f_i(L_i[j+1])$ for $0 \le j < |L_i|$.*[6] *The list L_i is ideally convex if the sequence Δ_i is non-increasing, i.e., $\Delta_i[j+1] \le \Delta_i[j]$ for every j. Equivalently, the piecewise linear function passing through the set of points $\{(j, f_i(L_i[j]))\}_{0\le j\le |L_i|}$ is convex for each i. A database $\mathcal{D}$ is ideally convex if every list L_i is ideally convex.*

An example of an inverted list satisfying the above assumption is shown in Figure 4.5(a). The max-reduction strategy $\mathcal{T}_{\mathsf{MR}}$ is optimal under the ideal convexity assumption:

Theorem 5 (Ideal Optimality) *Given a decomposable function F, for every ideally convex database $\mathcal{D}$ and every threshold θ, the access cost of $\mathcal{T}_{\mathsf{MR}}$ is exactly* OPT.

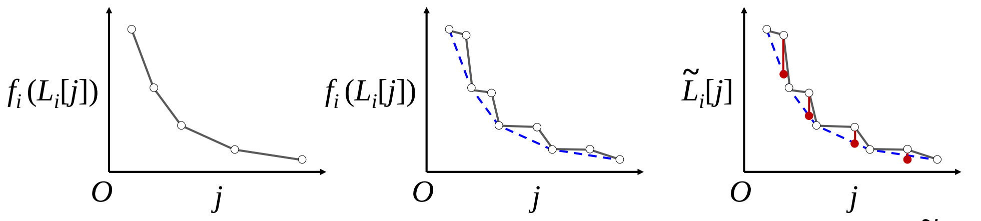
Proof 4 *Let $\{\mathbf{b}_t\}_{1\le t\le k}$ be the sequence of position vectors produced by the strategy $\mathcal{T}_{\mathsf{MR}}$.*

Since each Δ_i is non-increasing and the strategy $\mathcal{T}_{\mathsf{MR}}$ chooses the dimension i with the maximal $\Delta_i[b_i]$, then at each step t, the multiset $\{\Delta_i[j] | 1 \le i \le d, 0 \le j \le \mathbf{b}_t[i]\}$ contains the first t largest values of all the $\Delta_i[j]$'s from the multiset $\{\Delta_i[j] | 1 \le i \le d, 0 \le j < |L_i|\}$. Since the score $F(L[\mathbf{b}_t])$ equals

$$\sum_{i=1}^{d} f_i(L_i[0]) - \sum_{i=1}^{d} \sum_{j=1}^{\mathbf{b}_t[i]} \Delta_i[j] \,,$$

it follows that for each $\mathbf{b}_t$ of $\mathcal{T}_{\mathsf{MR}}$, the score $F(L[\mathbf{b}_t])$ is the lowest score possible for any position vector reachable in t steps. Thus, if the optimal access cost OPT *is t with an optimal stopping*

[6]Recall that $L_i[0] = 1$.



(a) Convex (b) Near-convex (c) Constructing $\widetilde{L}_i[j]$

Figure 4.5: Convexity and near-convexity

position $\mathbf{b}_{\mathsf{OPT}}$*, then* $\mathbf{b}_t$*, the* t*-th position of* $\mathcal{T}_{\mathsf{MR}}$*, satisfies that* $F(L[\mathbf{b}_t]) \le F(L[\mathbf{b}_{\mathsf{OPT}}]) < \theta$*. So* $\mathcal{T}_{\mathsf{MR}}$ *is optimal.*

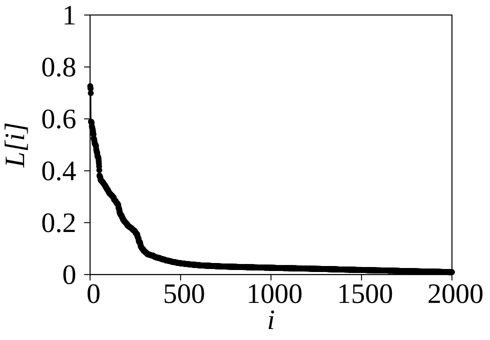
4.5.3 The hull-based traversal strategy

Theorem 5 provides a strong performance guarantee but the ideal convexity assumption is usually not true on real datasets. Without the ideal convexity assumption, the strategy suffers from the drawback of making locally optimal but globally suboptimal decisions. The pointer b_i to an inverted list L_i might never be moved if choosing the current b_i only results in a small decrease in the score $F(L[\mathbf{b}])$, but there is a much larger decrease several steps ahead. As a result, the $\mathcal{T}_{\mathsf{MR}}$ strategy has no performance guarantee in general.

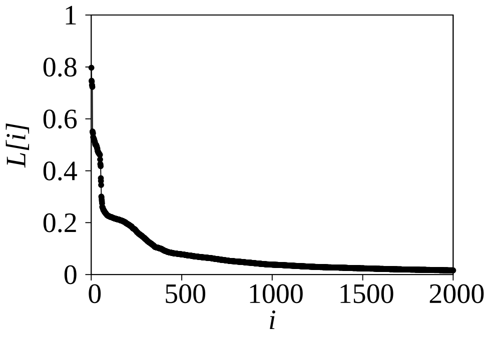
In most practical scenarios, we can bring the traversal strategy $\mathcal{T}_{\mathsf{MR}}$ to practicality by considering a relaxed version of Assumption 1. Informally, instead of assuming that each list $f_i(L_i)$ forms a convex piecewise linear function, we assume that $f_i(L_i)$ is "mostly" convex, meaning that if we compute the *lower convex hull* [DBCVKO08] of $f_i(L_i)$, the gap between any two consecutive vertices on the convex hull is small.[7] Intuitively, the relaxed assumption implies that the values at each list are decreasing at "approximately" decelerating speed. It allows list segments that do not follow the overall deceleration trend, as long as their lengths are bounded by a constant. We verified this property in the mass spectrometry dataset, as illustrated in Figure 4.6.

We formally state the assumption next.

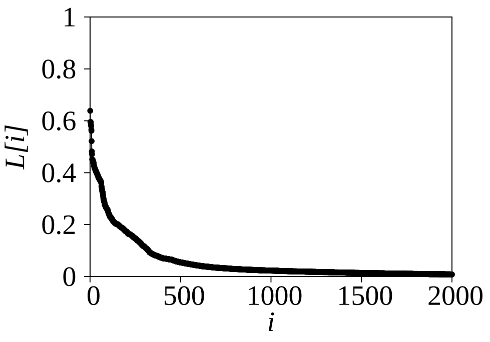
[7]We denote by $f_i(L_i)$ the list $[f_i(L_i[0]), f_i(L_i[1]), \ldots]$ for every L_i.



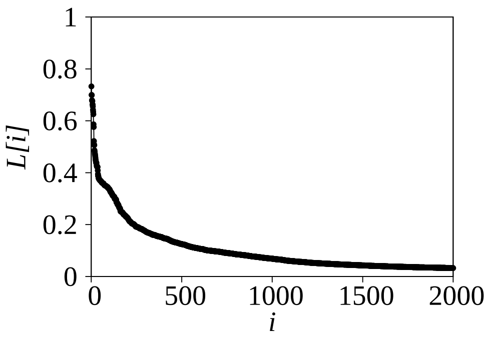
(a) list L_0



(b) list L_1



(c) list L_2



(d) list L_3

Figure 4.6: The skewed inverted lists in mass spectrometry

Assumption 2 (Near-Convexity) *For every inverted list L_i, let H_i be the lower convex hull of the set of 2-D points $\{(j, f_i(L_i[j]))\}_{0 \le j \le |L_i|}$ represented by a set of indices $\mathsf{H}_i = \{j_1, \dots, j_n\}$ where for each $1 \le k \le n$, $(j_k, f_i(L_i[j_k]))$ is a vertex of the convex hull. The list L_i is near-convex if for every k, $j_{k+1} - j_k$ is upper-bounded by some constant c. A database $\mathcal{D}$ is near-convex if every inverted list L_i is near-convex with the same constant c, which we refer to as the convexity constant.*

Example 1 *Intuitively, the near-convexity assumption captures the case where each $f_i(L_i)$ is decreasing with* approximately *decelerating speed, so the number of points between two convex hull vertices should be small. For example, when f_i is a linear function, the list L_i shown in Figure 4.5(b) is near-convex with convexity constant 2 since there is at most 1 point between each pair of consecutive vertices of the convex hull (dotted line). In the ideal case shown in Figure 4.5(a), the constant is 1 when the decrease between successive values is strictly decelerating.*

Imitating the max-reduction strategy, for every pair of consecutive indices j_k, j_{k+1} in H_i and for every index $j \in [j_k, j_{k+1})$, let $\tilde{\Delta}_i[j] = \dfrac{f_i(L_i[j_k]) - f_i(L_i[j_{k+1}])}{j_{k+1} - j_k}$. Since the $(j_k, f_i(L_i[j_k]))$'s are vertices of a lower convex hull, each sequence $\tilde{\Delta}_i$ is non-decreasing. Then the *hull-based* traversal strategy is simply defined as

$$\mathcal{T}_{\mathsf{HL}}(\mathbf{b}) = \underset{1 \le i \le d}{\operatorname{argmax}}(\tilde{\Delta}_i[b_i]). \tag{4.12}$$

Remark on data structures. In a practical implementation, to answer queries with scoring function F using the hull-based strategy, the lower convex hulls need to be ready before the traversal starts. If F is a general function unknown a priori, the convex hulls need to be computed online which is not practical. Fortunately, when F is the inner product $F(\mathbf{s}) = \mathbf{q} \cdot \mathbf{s}$ parameterized by the query $\mathbf{q}$, each convex hull H_i is exactly the convex hull for the points $\{(j, L_i[j])\}_{0 \le i \le |L_i|}$ from L_i. So by using the standard convex hull algorithm [DBCVKO08], H_i can be pre-computed in $O(|L_i|)$ time. Then the set of the convex hull vertices H_i can be stored as inverted lists and accessed for computing the $\tilde{\Delta}_i$'s during query processing. In the ideal case, H_i can be as large as $|L_i|$ but is much smaller in practice.

Moreover, during the traversal using the strategy $\mathcal{T}_{\mathsf{HL}}$, choosing the maximum $\tilde{\Delta}_i[b_i]$ at each step can be done in $O(\log d)$ time using a max heap. This satisfies the requirement that the traversal strategy is efficiently computable.

4.5.4 Near-optimality results

We next show that the hull-based strategy $\mathcal{T}_{\mathsf{HL}}$ is near-optimal under the near-convexity assumption.

Theorem 6 *Given a decomposable function F, for every near-convex database $\mathcal{D}$ and every threshold θ, the access cost of $\mathcal{T}_{\mathsf{HL}}$ is strictly less than $\mathsf{OPT} + c$ where c is the convexity constant.*

When the assumption holds with a small convexity constant, this near-optimality result provides a much tighter bound compared to the $d \cdot \mathsf{OPT}$ bound in the TA-inspired baseline. This is achieved under data assumption and by keeping the convex hulls as auxiliary data structure, so it does not contradict the lower bound results on the approximation ratio [FLN01]. Next, we formally prove the theorem.

Proof 5 *Let $\mathcal{B} = \{\mathbf{b}_i\}_{i \geq 0}$ be the sequence of position vectors generated by $\mathcal{T}_{\mathsf{HL}}$. We call a position vector $\mathbf{b}$ a* boundary position *if every b_i is the index of a vertex of the convex hull H_i. Namely, $b_i \in \mathsf{H}_i$ for every $i \in [d]$. Notice that if we break ties consistently during the traversal of $\mathcal{T}_{\mathsf{HL}}$, then in between every pair of consecutive boundary positions $\mathbf{b}$ and $\mathbf{b}'$ in $\mathcal{B}$, $\mathcal{T}_{\mathsf{HL}}(\mathbf{b})$ will always be the same index. We call the subsequence positions $\{\mathbf{b}_i\}_{l \leq i < r}$ of $\mathcal{B}$ where $\mathbf{b}_l = \mathbf{b}$ and $\mathbf{b}_r = \mathbf{b}'$ a* segment *with boundaries $(\mathbf{b}_l, \mathbf{b}_r)$.*

We show the result using the following lemma.

Lemma 2 *For every boundary position vector $\mathbf{b}$ generated by $\mathcal{T}_{\mathsf{HL}}$, we have $F(L[\mathbf{b}]) \leq F(L[\mathbf{b}^*])$ for every position vector $\mathbf{b}^*$ where $|\mathbf{b}^*| = |\mathbf{b}|$.*

Intuitively, the above lemma says that if the traversal of $\mathcal{T}_{\mathsf{HL}}$ reaches a boundary position $\mathbf{b}$, then the score $F(L[\mathbf{b}])$ is the minimal possible score obtained by any traversal sequence of at most $|\mathbf{b}|$ steps. Lemma 2 is sufficient for Theorem 6 because of the following. Suppose $\mathbf{b}_{\mathsf{stop}}$ is the stopping position in $\mathcal{B}$, which means that $\mathbf{b}_{\mathsf{stop}}$ is the first position in $\mathcal{B}$ that satisfies φ_F and the access cost is $|\mathbf{b}_{\mathsf{stop}}|$. Let $\{\mathbf{b}_i\}_{l \leq i < r}$ be the segment that contains $\mathbf{b}_{\mathsf{stop}}$. Given Lemma 2, Theorem 6 holds trivially if $\mathbf{b}_{\mathsf{stop}} = \mathbf{b}_l$. It remains to consider the case $\mathbf{b}_{\mathsf{stop}} \neq \mathbf{b}_l$. Since the traversal does not stop at $\mathbf{b}_l$, we have $F(L[\mathbf{b}_l]) \geq \theta$. By Lemma 2, $\mathbf{b}_l$ is the position with minimal $F(L[\cdot])$ obtained in $|\mathbf{b}_l|$ steps so $|\mathbf{b}_l| \leq \mathsf{OPT}$. Since $|\mathbf{b}_{\mathsf{stop}}| - |\mathbf{b}_l| < |\mathbf{b}_r| - |\mathbf{b}_l| \leq c$, we have that $|\mathbf{b}_{\mathsf{stop}}| < \mathsf{OPT} + c$. We illustrate this in Figure 4.7.

Now it remains to show Lemma 2. We construct a new collection of inverted lists $\{\tilde{L}_i\}_{1 \leq i \leq d}$ from the original lists as follows. For every i and every pair of consecutive indices j_k, j_{k+1} of H_i,

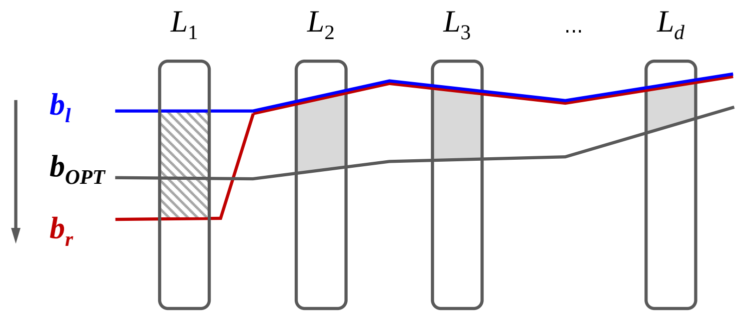


Figure 4.7: Illustration of Theorem 6

we assign

$$\tilde{L}_i[j] = f_i(L_i[j_k]) + (j - j_k) \cdot \tilde{\Delta}_i[j] \quad \textit{for every } j \in [j_k, j_{k+1}].$$

Intuitively, we construct each $\tilde{L}_i$ *by projecting the set of 2D points* $\{(i, f_i(L_i[j]))\}_{j \in [j_k, j_{k+1}]}$ *onto the line passing through the two boundary points with index* j_k *and* j_{k+1}*, which is essentially projecting the set of points onto the piecewise linear function defined by the convex hull vertices in* H_i *(See Figure 4.5(c) for an illustration). The new* $\{\tilde{L}_i\}_{1 \le i \le d}$ *satisfies the following properties.*

(i) By the construction of each convex hull H_i*, we have* $\tilde{L}_i[j] \le f_i(L_i[j])$ *for every* i *and* j.

(ii) For every boundary position $\mathbf{b}$*, we have* $\tilde{L}[\mathbf{b}] = F(L[\mathbf{b}])$ *since for every index* j *on a convex hull* H_i*,* $\tilde{L}[j] = f_i(L_i[j])$.

(iii) The collection $\{\tilde{L}_i\}_{1 \le i \le d}$ *is ideally convex*[8]*. In addition, the strategy* $\mathcal{T}_{\mathsf{HL}}$ *produces exactly the same sequence reduced by the max-reduction strategy* $\mathcal{T}_{\mathsf{MR}}$ *when* $\{\tilde{L}_i\}_{1 \le i \le d}$ *is given as the input. By the same analysis for Theorem 5, for every position vector* $\mathbf{b}$ *generated by* $\mathcal{T}_{\mathsf{HL}}$*,* $\tilde{L}[\mathbf{b}]$ *is minimal among all position vectors reached within* $|\mathbf{b}|$ *steps.*

Combining (ii) and (iii), for every boundary position vector $\mathbf{b}$ *generated by* $\mathcal{T}_{\mathsf{HL}}$ *and every* $\mathbf{b}^*$ *where* $|\mathbf{b}^*| = \mathbf{b}$*, we have* $F(L[\mathbf{b}]) = \tilde{L}[\mathbf{b}] \le \tilde{L}[\mathbf{b}^*]$*. Finally, by (i) and since* F *is non-decreasing, we have* $\tilde{L}[\mathbf{b}^*] \le F(L[\mathbf{b}^*])$ *so* $F(L[\mathbf{b}]) \le F(L[\mathbf{b}^*])$ *for every* $\mathbf{b}^*$.

[8] where each f_i of the decomposable function F is the identity function

Since the baseline stopping condition φ_{BL} is tight and complete for inner product queries, one immediate implication of Theorem 6 is that

Corollary 1 *(Informal) The hull-based strategy $\mathcal{T}_{\mathsf{HL}}$ for inner product queries is near-optimal.*

Verifying the assumption. We demonstrate the practical impact of the near-optimality result in real mass spectrometry datasets. The near-convexity assumption requires that the gap between any two consecutive convex hull vertices has bounded size, which is hard to achieve in general. According to the proof of Theorem 6, for a given query, the difference from the optimal access cost is at most the size of the gap between the two consecutive convex hull vertices containing the last move of the strategy (the $\mathbf{b}_l$ and $\mathbf{b}_r$ in Figure 4.2). The size of this gap can be much smaller than the global convexity constant c, so the overall precision can be much better in practice. We verify this by running a set of 1,000 real queries on the dataset[9]. The gap size is 163.04 in average, which takes only 1.3% of the overall access cost of traversing the indices. This indicates that the near-optimality guarantee holds in the mass spectrometry dataset.

4.5.5 The traversal strategy for cosine

Next, we consider traversal strategies which take into account the unit vector constraint posed by the cosine function, which means that the tight and complete stopping condition is φ_{TC} introduced in Section 4.4. However, since the scoring function MS in φ_{TC} is not decomposable, the hull-based technique cannot be directly applied. We adapt the technique by approximating the original MS with a decomposable function $\tilde{F}$. Without changing the stopping condition φ_{TC}, the hull-based strategy can then be applied with the convex hull indices constructed with the approximation $\tilde{F}$. In the rest of this section, we first generalize the result in Theorem 6 to scoring functions having decomposable approximations and show how the hull-based traversal strategy can be adapted. Next, we show a natural choice of the approximation for MS with practically

[9]https://proteomics2.ucsd.edu/ProteoSAFe/index.jsp

tight near-optimal bounds. Finally, we discuss data structures to support fast query processing using the traversal strategy.

We start with some additional definitions.

Definition 11 *A d-dimensional function F is decomposably approximable if there exists a decomposable function $\tilde{F}$, called the* decomposable approximation *of F, and two non-negative constants ε_1 and ε_2 such that $\tilde{F}(\mathbf{s}) - F(\mathbf{s}) \in [-\varepsilon_1, \varepsilon_2]$ for every vector $\mathbf{s}$.*

When applied to a decomposably approximable function F, the hull-based traversal strategy $\mathcal{T}_{\mathsf{HL}}$ is adapted by constructing the convex hull indices and the $\{\tilde{\Delta}_i\}_{1\leq i\leq d}$ using the approximation $\tilde{F}$. The following can be obtained by generalizing Theorem 6:

Theorem 7 *Given a function F approximable by a decomposable function $\tilde{F}$ with constants $(\varepsilon_1, \varepsilon_2)$, for every near-convex database $\mathcal{D}$ wrt $\tilde{F}$ and every threshold θ, the access cost of $\mathcal{T}_{\mathsf{HL}}$ is strictly less than $\mathsf{OPT}(\theta - \varepsilon_1 - \varepsilon_2) + c$ where c is the convexity constant.*

Proof 6 *Recall that $\mathbf{b}_l$ is the last boundary position generated by $\mathcal{T}_{\mathsf{HL}}$ that does not satisfy the tight stopping condition for F (which is φ_{TC} when F is MS) so $F(L[\mathbf{b}_l]) \geq \theta$. It is sufficient to show that for every vector $\mathbf{b}^*$ where $|\mathbf{b}^*| = |\mathbf{b}_l|$, $F(L[\mathbf{b}^*]) \geq \theta - \varepsilon_1 - \varepsilon_2$ so no traversal can stop within $|\mathbf{b}_l|$ steps, implying that the final access cost is no more than $|\mathbf{b}_l| + c$ which is bounded by $\mathsf{OPT}(\theta - \varepsilon_1 - \varepsilon_2) + c$.*

By Lemma 2, we know that for every such $\mathbf{b}^*$, $\tilde{F}(L[\mathbf{b}^*]) \geq \tilde{F}(L[\mathbf{b}_l])$. By definition of the approximation $\tilde{F}$, we know that $F(L[\mathbf{b}^*]) \geq \tilde{F}(L[\mathbf{b}^*]) - \varepsilon_1$ and $\tilde{F}(L[\mathbf{b}_l]) \geq F(L[\mathbf{b}_l]) - \varepsilon_2$. Combined together, for every $\mathbf{b}^*$ where $|\mathbf{b}^*| = |\mathbf{b}_l|$, we have

$$\begin{aligned} F(L[\mathbf{b}^*]) &\geq \tilde{F}(L[\mathbf{b}^*]) - \varepsilon_1 \geq \tilde{F}(L[\mathbf{b}_l]) - \varepsilon_1 \\ &\geq F(L[\mathbf{b}_l]) - \varepsilon_1 - \varepsilon_2 \geq \theta - \varepsilon_1 - \varepsilon_2. \end{aligned}$$

This completes the proof of Theorem 7.

Choosing the decomposable approximation. By Theorem 7, it is important to choose an approximation $\tilde{F}$ of MS with small ε_1 and ε_2 for a tight near-optimality result. By inspecting the formula (4.8) of MS, one reasonable choice of $\tilde{F}$ can be obtained by replacing the term τ with a fixed constant $\tilde{\tau}$. Formally, let

$$\tilde{F}(L[\mathbf{b}]) = \sum_{i=1}^{d} \min\{q_i \cdot \tilde{\tau}, L_i[b_i]\} \cdot q_i \tag{4.13}$$

be the decomposable approximation of MS where each component is a non-decreasing function $f_i(x) = \min\{q_i \cdot \tilde{\tau}, x\} \cdot q_i$ for $i \in [d]$.

Ideally, the approximation is tight if the constant $\tilde{\tau}$ is close to the final value of τ which is unknown in advance. Next, we argue that when $\tilde{\tau}$ is properly chosen, the approximation parameter $\varepsilon_1 + \varepsilon_2$ is very small.

For ε_1: A trivial upper bound of ε_1 is $\tilde{\tau} - 1$ since the initial value of τ is 1 and the gap $\tilde{F}(L[\mathbf{b}]) - \mathsf{MS}(L[\mathbf{b}])$ is maximized when $\mathbf{b} = \mathbf{0}$. This upper bound can be improved as follows. We notice that in the proof of Theorem 7, given $|\mathbf{b}^*| = |\mathbf{b}_l|$, we need to have $F(L[\mathbf{b}^*]) \geq \theta - \varepsilon_1 - \varepsilon_2$ for every such $\mathbf{b}^*$, which is equivalent to requiring that this property holds for the $\mathbf{b}^*$ that minimizes $F(L[\mathbf{b}^*])$ given $|\mathbf{b}^*| = |\mathbf{b}_l|$. This $\mathbf{b}^*$ satisfies that $F(L[\mathbf{b}^*]) \leq F(L[\mathbf{b}_l])$ and $F(L[\mathbf{b}_l])$ is known when the query is executed. This upper bound of $F(L[\mathbf{b}^*])$ implies a lower bound of τ at position $\mathbf{b}^*$, which also implies the following lower bound of $\mathsf{MS}(L[\mathbf{b}^*]) - \tilde{F}(L[\mathbf{b}^*])$:

$$(\dagger) \quad \mathsf{MS}(L[\mathbf{b}^*]) - \tilde{F}(L[\mathbf{b}^*]) \geq \min\{0, 1/\mathsf{MS}(L[\mathbf{b}_l]) - \tilde{\tau}\}.$$

The complete proof of $(\dagger)$ is provided in Lemma 3. Thus, when the query is given, ε_1 is at most $\max\{0, \tilde{\tau} - 1/\mathsf{MS}(L[\mathbf{b}_l])\}$.

For ε_2: In general, there is no upper bound for τ since it can be as large as $L_i[b_i]/q_i$ for some i. The gap $\mathsf{MS}(L[\mathbf{b}]) - \tilde{F}(L[\mathbf{b}])$ can be close to 1. However, the proof of Theorem 7 only requires ε_2 to be at least the difference between MS and $\tilde{F}$ at position $\mathbf{b}_l$. So ε_2 is upper-bounded by $\mathsf{MS}(L[\mathbf{b}_l]) - \tilde{F}(L[\mathbf{b}_l])$ for a given query.

Summarizing the above analysis, the approximation factor ε is determined by the following two factors:

1. how much the approximation $\tilde{F}$ is smaller than the scoring function MS at $\mathbf{b}_l$, the starting position of the last segment chosen during the traversal, and
2. how much $\tilde{F}$ is bigger than MS at the optimal position with exactly $|\mathbf{b}_l|$ steps that minimizes MS.

This yields the following upper bound of ε:

$$\varepsilon \leq \max\{0, \tilde{\tau} - 1/\mathsf{MS}(L[\mathbf{b}_l])\} + \mathsf{MS}(L[\mathbf{b}_l]) - \tilde{F}(L[\mathbf{b}_l]). \tag{4.14}$$

Lemma 3 *Let* $\mathbf{b}$ *be an arbitrary position vector and let* $\mathbf{b}^*$ *be the position vector such that*

$$\mathbf{b}^* = \underset{\mathbf{b}':|\mathbf{b}|=|\mathbf{b}'|}{\arg\min}\{\mathsf{MS}(L[\mathbf{b}'])\}.$$

Then

$$(\dagger) \quad \mathsf{MS}(L[\mathbf{b}^*]) - \tilde{F}(L[\mathbf{b}^*]) \geq \min\{0, 1/\mathsf{MS}(L[\mathbf{b}]) - \tilde{\tau}\}.$$

where $\tilde{F}$ *is the decomposable function where each component is* $f_i(x) = \min\{\tilde{\tau} \cdot q_i, x\} \cdot q_i$ *for constant* $\tilde{\tau}$ *and every* $1 \leq i \leq d$.

Proof 7 *Let* τ^* *be the value of* τ *at* $L[\mathbf{b}^*]$. *We consider two cases separately:* $\tau^* \geq \tilde{\tau}$ *and* $\tau^* < \tilde{\tau}$.

Case One: *When* $\tau^* \geq \tilde{\tau}$, *since each* f_i *increases as* $\tilde{\tau}$ *increases, we have*

$$f_i(x) = \min\{\tilde{\tau} \cdot q_i, x\} \cdot q_i \leq \min\{\tau^* \cdot q_i, x\} \cdot q_i$$

thus $\mathsf{MS}(L[\mathbf{b}^*]) \geq \tilde{F}(L[\mathbf{b}^*])$.

Case Two: *Suppose* $\tau^* < \tilde{\tau}$. *We show the following Lemma.*

Lemma 4 *For every position $\mathbf{b}$ and constant c, $\mathsf{MS}(L[\mathbf{b}]) < c$ implies that the τ at $L[\mathbf{b}]$ is at least $1/c$.*

Recall the notations $\mathsf{LQ} = \sum_{i:L_i[b_i]<\tau\cdot q_i} L_i[b_i]\cdot q_i$, $\mathsf{Q2} = \sum_{i:L_i[b_i]<\tau\cdot q_i} q_i^2$ and $\mathsf{L2} = \sum_{i:L_i[b_i]<\tau\cdot q_i} L_i[b_i]^2$. Then τ satisfies that

$$\mathsf{LQ} + (1-\mathsf{Q2})\cdot\tau < c \tag{4.15}$$

and

$$\mathsf{L2} + (1-\mathsf{Q2})\cdot\tau^2 = 1. \tag{4.16}$$

By rewriting Equation (4.16), we have $(1-\mathsf{Q2}) = (1-\mathsf{L2})/\tau^2$. Plug this into (4.15), we have $\mathsf{LQ} + (1-\mathsf{L2})/\tau < c$. Since $L_i[b_i] < \tau\cdot q_i$, we have $\mathsf{LQ} > \mathsf{L2}/\tau$ so $1/\tau < c$ which means $\tau > 1/c$. This completes the proof of Lemma 4.

We know that since $\mathbf{b}^*$ minimizes $\mathsf{MS}(L[\mathbf{b}'])$ among all $\mathbf{b}'$ with $|\mathbf{b}'| = |\mathbf{b}|$, we have $\mathsf{MS}(L[\mathbf{b}^*]) \leq \mathsf{MS}(L[\mathbf{b}])$. By Lemma 4, we have $\tau^* \geq 1/\mathsf{MS}(L[\mathbf{b}])$.

Now consider $\mathsf{MS}(L[\mathbf{b}^*]) - \tilde{F}(L[\mathbf{b}^*])$. Since $\tau^* < \tilde{\tau}$, $\mathsf{MS}(L[\mathbf{b}^*]) - \tilde{F}(L[\mathbf{b}^*])$ can be written as the sum of the following 3 terms:

- $\sum_{i:L_i[b_i]/q_i<\tau^*}(L_i[b_i]\cdot q_i - L_i[b_i]\cdot q_i)$*, which is always 0,*
- $\sum_{i:\tau^*\leq L_i[b_i]/q_i<\tilde{\tau}}(q_i^2\cdot\tau^* - L_i[b_i]\cdot q_i)$ *and*
- $\sum_{i:\tilde{\tau}\leq L_i[b_i]/q_i}(q_i^2\cdot\tau^* - q_i^2\cdot\tilde{\tau})$.

In the second term, since each $L_i[b_i]$ is at most $q_i\cdot\tilde{\tau}$, so this term is greater than or equal to $\sum_{i:\tau^*\leq L_i[b_i]/q_i<\tilde{\tau}}(q_i^2\cdot\tau^* - q_i^2\cdot\tilde{\tau})$. Adding them together, we have $\mathsf{MS}(L[\mathbf{b}^*]) - \tilde{F}(L[\mathbf{b}^*])$ is lower bounded by

$$\sum_{i:\tau^*\leq L_i[b_i]/q_i} q_i^2\cdot(\tau^* - \tilde{\tau})$$

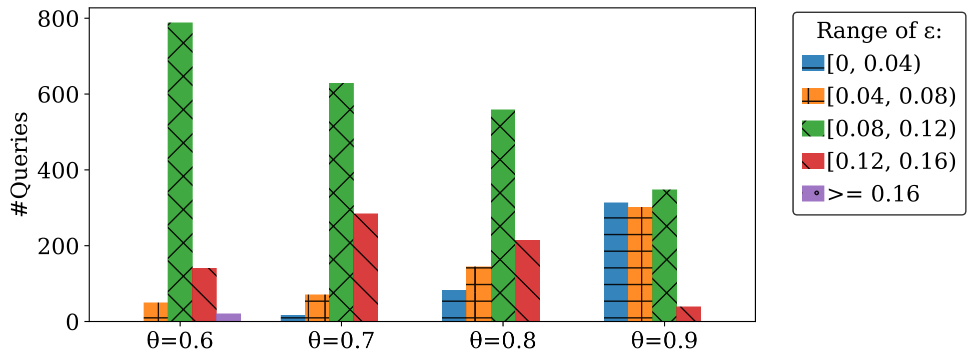


Figure 4.8: The distribution of ε

which is greater than or equal to $\tau^* - \tilde{\tau}$. *Since* $\tau^* \geq 1/\mathsf{MS}(L[\mathbf{b}])$, *we have* $\mathsf{MS}(L[\mathbf{b}^*]) - \tilde{F}(L[\mathbf{b}^*]) \geq 1/\mathsf{MS}(L[\mathbf{b}]) - \tilde{\tau}$. *Finally, combining case one and two together completes the proof of Lemma 3.*

Verifying the near-optimality. Next, we verify that the above upper bound of ε is small in practice. We ran the same set of queries as in Section 4.5.4 and show the distribution of ε's upper bounds in Figure 4.8. We set $\tilde{\tau} = 1/\theta$ for all queries so the first term of (4.14) becomes zero. Note that more aggressive pruning can yield better ε, but it is not done here for simplicity. Overall, the fraction of queries with an upper bound <0.12 (the sum of the first 3 bars for all θ) is 82.5% and the fraction of queries with $\varepsilon > 0.16$ is 0.5%. Similar to the case with inner product queries, the average of the convexity constant c is 193.39, which is only 4.8% of the overall access cost.

Remark on data structures. Similar to the inner product case, it is necessary that the convex hulls for $\mathcal{T}_{\mathsf{HL}}$ can be efficiently obtained without a full computation when a query comes in. For every $i \in [d]$, we let $\tilde{\mathsf{H}}_i$ be the convex hull for the i-th component f_i of $\tilde{F}$ and H_i be the convex hull constructed directly from the original inverted list L_i. Next, we show that each $\tilde{\mathsf{H}}_i$ can be efficiently obtained from H_i during query time so we only need to pre-compute the H_i's.

We observe that when $L_i[b_i] \geq q_i \cdot \tilde{\tau}$, $f_i(L_i[b_i])$ equals a fixed value $q_i^2 \cdot \tilde{\tau}$ otherwise is proportional to $L_i[b_i]$. As illustrated in Figure 4.9 (left), the list of values $\{f_i(L_i[j])\}_{j \geq 0}$ is essentially obtained by replacing the $L_i[j]$'s greater than $q_i \cdot \tilde{\tau}$ with $q_i \cdot \tilde{\tau}$.

The following can be shown using properties of convex hulls:

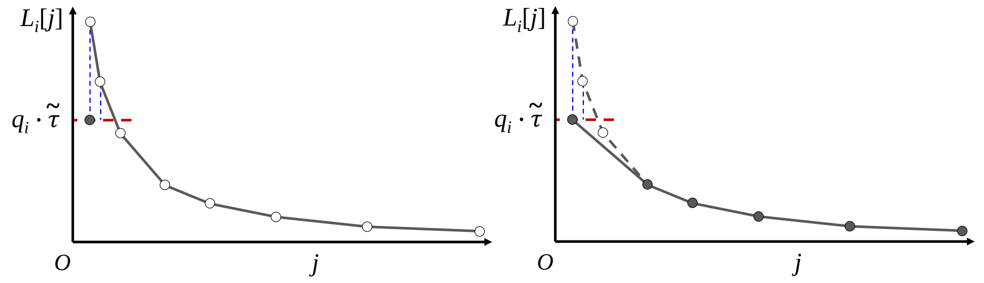


Figure 4.9: The construction of the convex hull $\tilde{\mathsf{H}}_i$

Lemma 5 *For every $i \in [d]$, the convex hull $\tilde{\mathsf{H}}_i$ is a subset of H_i where an index j_k of H_i is in $\tilde{\mathsf{H}}_i$ iff $k = 1$ or*

$$\frac{q_i \cdot \tilde{\tau} - L_i[j_k]}{j_k} \geq \frac{L_i[j_k] - L_i[j_{k+1}]}{j_{k+1} - j_k}. \tag{4.17}$$

Lemma 5 provides an efficient way to obtain each convex hull $\tilde{\mathsf{H}}_i$ from the pre-computed H_i's. When a query $\mathbf{q}$ is given, we perform a binary search on each H_i to find the first $j_k \in \mathsf{H}_i$ that satisfies (4.17). Then $\tilde{\mathsf{H}}_i$ is the set of indices $\{0, j_k, j_{k+1} \ldots\}$. We illustrate the construction in Figure 4.9 (right).

Suppose that the maximum size of all H_i is h. The computation of the $\tilde{\mathsf{H}}_i$'s adds an extra $O(d \log h)$ of overhead to the query processing time, which is insignificant in practice since h is likely to be much smaller than the size of the database.

4.6 The verification phase

Next, we discuss optimizations in the verification phase where each gathered candidate is tested for θ-similarity with the query. The naive approach of verification is to fully access all the non-zero entries of each candidate $\mathbf{s}$ to compute the exact similarity score $\texttt{cos}(\mathbf{q}, \mathbf{s})$ and compare it against θ, which takes $O(d)$ time. Various techniques have been proposed [TG16, AK15, LCYM17] to decide θ-similarity by leveraging only partial information about the candidate vector so the exact computation of $\texttt{cos}(\mathbf{q}, \mathbf{s})$ can be avoided. In this section, we revisit these existing techniques which we call *partial verification*. In addition, as a novel contribution, we show that in the presence of data skewness, partial verification can have a near-constant performance guarantee (Theorem 8).

Informally, while a vector $\mathbf{s}$ is scanned, based on what has been observed in $\mathbf{s}$ so far, it is possible to infer that

(1) the similarity score $\texttt{cos}(\mathbf{q},\mathbf{s})$ is certainly at least θ or

(2) the similarity score is certainly below θ.

In either case, we can stop without scanning the rest of $\mathbf{s}$ and return an accurate verification result. The problem is formally defined as follows:

Problem 1 (Partial Verification) *A partially observed vector* $\tilde{\mathbf{s}}$ *is a* d*-dimensional vector in* $(\mathbb{R}_{\geq 0} \cup \{\perp\})^d$*. Given a query* $\mathbf{q}$ *and a partially observed vector* $\tilde{\mathbf{s}}$*, compute whether for every vector* $\mathbf{s}$ *where* $\mathbf{s}[i] = \tilde{\mathbf{s}}[i]$ *for every* $\tilde{\mathbf{s}}[i] \neq \perp$*, it is* $\texttt{cos}(\mathbf{s},\mathbf{q}) \geq \theta$*.*

Intuitively, a partially observed vector $\tilde{\mathbf{s}}$ contains the entries of a candidate $\mathbf{s}$ either already observed during the gathering phase, or accessed for the actual values during the verification phase. The unobserved dimensions are replaced with a null value $\perp$. We say that a vector $\mathbf{s}$ is compatible with a partially observed one $\tilde{\mathbf{s}}$ if $\mathbf{s}[i] = \tilde{\mathbf{s}}[i]$ for every dimension i where $\tilde{\mathbf{s}}[i] \neq \perp$.

The partial verification problem can be solved by computing an upper and a lower bound of the cosine similarity between $\mathbf{s}$ and $\mathbf{q}$ when $\tilde{\mathbf{s}}$ is observed. We denote by $ub(\tilde{\mathbf{s}})$ and $lb(\tilde{\mathbf{s}})$ the upper bound and lower bound, so $ub(\tilde{\mathbf{s}}) = \max_{\mathbf{s}}\{\mathbf{s} \cdot \mathbf{q}\}$ and $lb(\tilde{\mathbf{s}}) = \min_{\mathbf{s}}\{\mathbf{s} \cdot \mathbf{q}\}$ where the maximum/minimum are taken over all $\mathbf{s}$ compatible with $\tilde{\mathbf{s}}$. By [TG16, AK15, LCYM17], the upper/lower bounds can be computed as follows:

Lemma 6 *Given a partially observed vector* $\tilde{\mathbf{s}}$ *and a query vector* $\mathbf{q}$*,*

$$ub(\tilde{\mathbf{s}}) = \sum_{\tilde{\mathbf{s}}[i]\neq\perp} \tilde{\mathbf{s}}[i]\cdot\mathbf{q}[i] + \sqrt{1-\sum_{\tilde{\mathbf{s}}[i]\neq\perp}\tilde{\mathbf{s}}[i]^2}\cdot\sqrt{1-\sum_{\tilde{\mathbf{s}}[i]\neq\perp}\mathbf{q}[i]^2}\ ,$$

and

$$lb(\tilde{\mathbf{s}}) = \sum_{\tilde{\mathbf{s}}[i]\neq\perp} \tilde{\mathbf{s}}[i]\cdot\mathbf{q}[i] + \sqrt{1-\sum_{\tilde{\mathbf{s}}[i]\neq\perp}\tilde{\mathbf{s}}[i]^2}\cdot\min_{\tilde{\mathbf{s}}[i]=\perp}\mathbf{q}[i]\ .$$

Example 2 *Figure 4.10 shows an example of computing the lower/upper bounds of a partially scanned vector* $\mathbf{s}$ *with a query* $\mathbf{q}$*. Assume the first three dimensions have been scanned, then the lower bound and upper bound are computed as follows:*

$$lb = 0.8\times 0 + 0.4\times 0.7 + 0.3\times 0.5 = 0.43$$
$$ub = lb + \sqrt{1-0.8^2-0.4^2-0.3^2}\cdot\sqrt{1-0.7^2-0.5^2} = 0.6$$

If the threshold θ *is 0.7, then it is certain that* $\mathbf{s}$ *is not* θ*-similar to* $\mathbf{q}$ *because* $0.6 < 0.7$*. The verification algorithm can stop and avoid the rest of the scan. Note that when* $\mathbf{q}$ *is sparse, most of the time we have* $lb(\tilde{\mathbf{s}}) = \sum_{\tilde{\mathbf{s}}[i]\neq\perp}\tilde{\mathbf{s}}[i]\cdot\mathbf{q}[i]$ *due to the existence of 0's.*

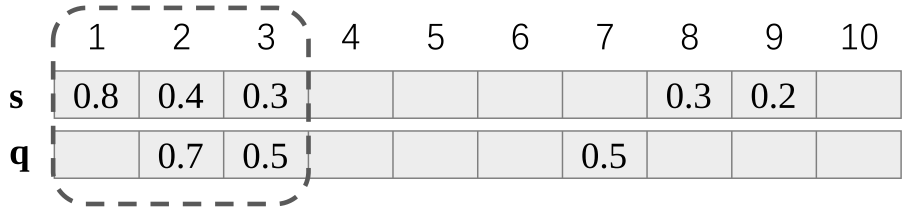


Figure 4.10: An example of the lower and upper bounds

Performance Guarantee. Next, we show that when the data is skewed, partial verification achieves a strong performance guarantee. We assume that each candidate $\mathbf{s}$ is stored in the database such that $\mathbf{s}[1]\geq\mathbf{s}[2]\geq\cdots\geq\mathbf{s}[d]$ and we scan $\mathbf{s}$ sequentially starting from $\mathbf{s}[1]$. We notice that partial verification saves data accesses when there is a gap between the true similarity score and the threshold θ. Intuitively, when this gap is close to 0, both the upper and lower bounds converge to θ as we scan $\mathbf{s}$ so we might not stop until the last element. When the gap is large and $\mathbf{s}$ is skewed, meaning that the first few values account for most of the candidate's weight, then the first few $\mathbf{s}[i] \cdot \mathbf{q}[i]$ terms can provide large enough information for the lower/upper bounds to decide θ-similarity. Formally,

Theorem 8 *Suppose that a vector* $\mathbf{s}$ *is skewed: there exists an integer* $k \leq d$ *and constant value* c *such that* $\sum_{i=1}^{k} \mathbf{s}[i]^2 \geq c$. *For every query* $(\mathbf{q}, \theta)$, *if* $|\mathtt{cos}(\mathbf{s}, \mathbf{q}) - \theta| \geq \sqrt{1-c}$, *then the number of accesses for verifying* $\mathbf{s}$ *is at most* k.

Equivalently, Theorem 8 says that if the true similarity is at least δ off the threshold θ (i.e. $|\mathtt{cos}(\mathbf{s}, \mathbf{q}) - \theta| \geq \delta$ for $\delta > 0$), then it is only necessary to access k entries of $\mathbf{s}$ with the smallest k that satisfies $\sum_{i=1}^{k} \mathbf{s}[i]^2 \geq 1 - \delta^2$. For example, if $\delta = 0.1$ and the first 20 entries of a candidate $\mathbf{s}$ account for ¿99% of $\sum_{i=1}^{d} \mathbf{s}[i]^2$, then it takes at most 20 accesses for verifying $\mathbf{s}$.

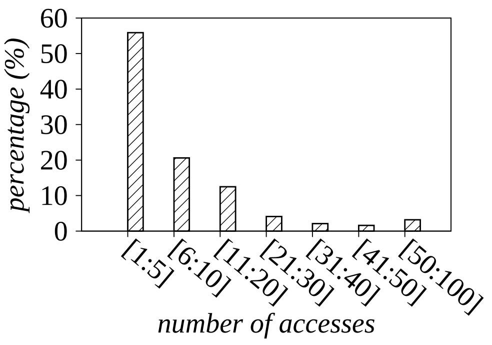
Proof 8 ***Case one:*** *We first consider the case where* $\mathtt{cos}(\mathbf{s}, \mathbf{q}) - \theta \geq \sqrt{1-c}$. *In this case, we need to show* $\sum_{i=1}^{k} \mathbf{s}[i] \cdot \mathbf{q}[i] \geq \theta$. *Since* $\mathtt{cos}(\mathbf{s}, \mathbf{q}) - \theta \geq \sqrt{1-c}$, *which means*

$$\sum_{i=1}^{k} \mathbf{s}[i] \cdot \mathbf{q}[i] + \sum_{i=k+1}^{d} \mathbf{s}[i] \cdot \mathbf{q}[i] \geq \theta + \sqrt{1-c}\ ,$$

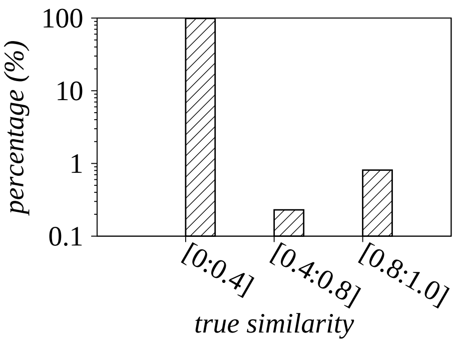
it suffices to show that $\sum_{i=k+1}^{d} \mathbf{s}[i] \cdot \mathbf{q}[i] \leq \sqrt{1-c}$. *This can be obtained by*

$$\begin{aligned}\sum_{i=k+1}^{d} \mathbf{s}[i] \cdot \mathbf{q}[i] &\leq \sqrt{\sum_{i=k+1}^{d} \mathbf{s}[i]^2} \cdot \sqrt{\sum_{i=k+1}^{d} \mathbf{q}[i]^2} \\ &= \sqrt{1 - \sum_{i=1}^{k} \mathbf{s}[i]^2} \cdot \sqrt{1 - \sum_{i=1}^{k} \mathbf{q}[i]^2} \\ &\leq \sqrt{1-c}\ .\end{aligned}$$

Case two: *We then consider the case where* $\mathtt{cos}(\mathbf{s}, \mathbf{q}) - \theta \leq -\sqrt{1-c}$. *In this case, we*



(a) number of accesses



(b) true similarities

Figure 4.11: The distribution of number of accesses and true similarities

need to show

$$\sum_{i=1}^{k} \mathbf{s}[i] \cdot \mathbf{q}[i] + \sqrt{1 - \sum_{i=1}^{k} \mathbf{s}[i]^2} \cdot \sqrt{1 - \sum_{i=1}^{k} \mathbf{q}[i]^2} \leq \theta \, .$$

The first term of the LHS is bounded by $\sum_{i=1}^{d} \mathbf{s}[i] \cdot \mathbf{q}[i] \leq \theta - \sqrt{1-c}$. *The second term of the LHS is bounded by* $\sqrt{1 - \sum_{i=1}^{k} \mathbf{s}[i]^2}$
$\leq \sqrt{1-c}$. *Adding together, the LHS is bounded by* θ.

Example 3 *Figure 4.11a shows the number of accesses for each candidate during the verification phase of a real query with a vector having 100 non-zero valuesand* $\theta = 0.6$. *The result shows that for most candidates, the number of accesses is much smaller than 100. In particular, 55.9% candidates need less than five accesses and 93.1% candidates need less than 30 accesses. This is because as shown in Figure 4.11b, only 0.23% of candidates have true similarity within* ± 0.2 *compared to* θ *(the range [0.4, 0.8]). The rest of the candidates, according to Theorem 8, can be verified in a small number of steps.*

4.7 System design and implementation

In this section, we present a mass spectrometry search system based on the techniques mentioned in prior sections.

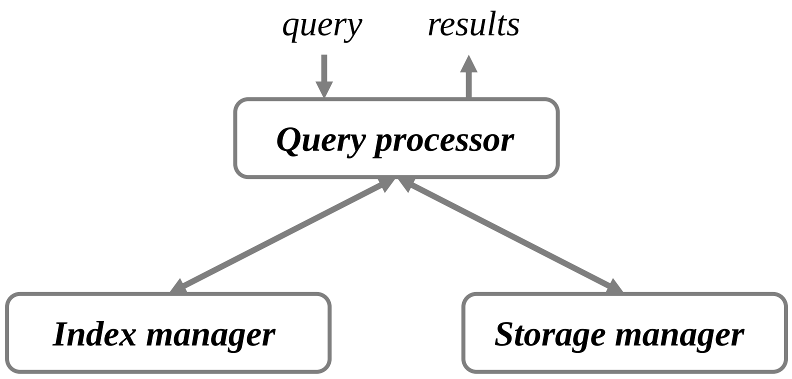


Figure 4.12: Mass spectrometry search system architecture

4.7.1 System architecture

The goal of the system is to find similar spectra for given a query spectrum. Figure 4.12 shows the system architecture that takes as input a user query (or a collection of queries) and returns the search results. There are three major components of the system:

- Query processor. It receives user queries and accesses the index manager and storage manager to support efficient query processing following Algorithm 5.
- Index manager. It builds and stores index (i.e., a collection of value-sorted lists) to support the gathering phase of the algorithm to return a collection of candidates.
- Storage manager. It stores and manages the actual spectrum. It supports the verification phases of the algorithm to verify whether a candidate is a true result.

4.7.2 Index manager

The index structure consists of a collection of value-sorted lists where each list is built for a dimension and is augmented with convex hull for fast query processing as illustrated in Section 4.2 and Section 4.5. In the implementation, instead of building the index for the whole dataset, we build it per *bucket* as explained next to significantly reduce the search space.

Before diving into the buckets, we describe a bit more on mass spectrometry. A spectrum s is associated with an extra attribute *peptide mass* s_p that measures the molecular weight of the spectrum. Observe that two spectra s and r are similar only if their peptide mass values are close enough, i.e., $|s_p - r_p| \leq \lambda$, where λ is a threshold parameter (e.g., 2).

As a result, we partition the whole spectrum dataset based on their peptide mass value and each bucket stores the spectra that share the same peptide mass. Thus, we build index per bucket and only search relevant buckets upon query processing. For example in Figure 4.13, assume the query has peptide mass 504 and $\lambda = 1$, then it only needs to search bucket 503, 504, and 505, which will dramatically reduce the search space.

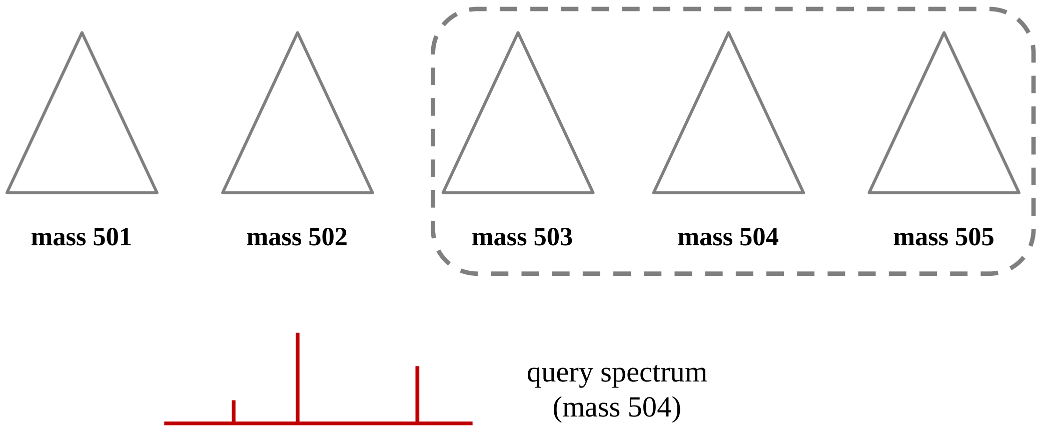


Figure 4.13: Index partitioning

The index manager supports both in-memory and on-disk accesses. In general, it stores the whole index in main memory. However, if the whole index cannot fit into memory, it stores bucket indexes in memory such that queries can be executed in a bucket at a time.

Besides that, the in-memory version index is optimized for CPU caches. Recall in TA, there is a randomly accessed list built on top of the sorted list to fetch the i-th dimensional value of a spectrum s. However, that design incurs too many CPU cache misses because the list is usually very long. Instead, we defer the access of $s[i]$ to the verification phase and sequentially scan all the peaks of the candidate s to reduce CPU cache misses.

4.7.3 Storage manager

As mentioned above, we partition the whole spectrum dataset into buckets based on peptide mass values. Thus, the storage manager maintains a collection of buckets where each bucket can be stored as a single file or multiple buckets share the same file to avoid opening too many files in the system.[10] The storage manager also maintains metadata to keep track of the routing information such that it can access the right buckets. Note that the design is very suitable for parallelism where each processor can process a single file without any synchronization overhead.

In terms of the storage within a bucket, we serialize each spectrum into bytes and store them sequentially on the disk. Since each spectrum may have different size, we also store metadata that remembers the files offset and size of the spectrum, see Figure 4.14. For a single spectrum, all the peaks are sorted in descending order based on the intensity to allow fast verification as explained in Section 4.6.

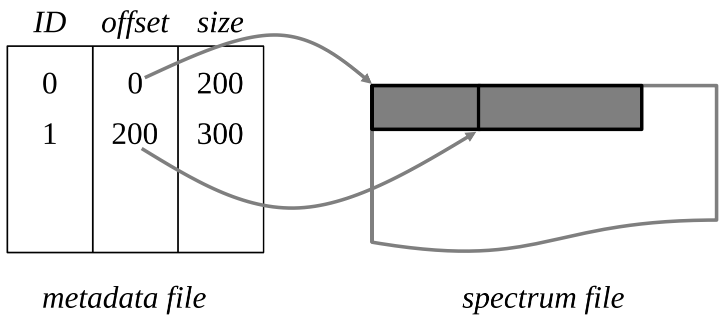


Figure 4.14: Spectrum storage

To reduce the space overhead, we also compress the spectrum. Let s_i be the i-th intensity and $\hat{s}_i$ be its discretized value, i.e., $\hat{s}_i = \frac{\mathsf{round}(2^b \cdot s_i)}{2^b}$, where b is the number of bits for compression (e.g., $b = 8$). Then $\hat{s}_i$ can be represented as an integer $\mathsf{round}(2^b \cdot s_i)$, which is stored in b bits. Note that $|s_i - \hat{s}_i| \leq \frac{1}{2^b} \cdot \frac{1}{2}$ for every i.

[10]Many operating systems have a limit of the number of maximal files opened, e.g., the default value of CentOS (used in the experiments) is 1024.

Given a query spectrum q (non-compressed) and a data spectrum s (compressed), then the error bound of the similarity is,

$$
\begin{aligned}
|\sum_{i=1}^{d}(q_i \cdot s_i) - \sum_{i=1}^{d}(q_i \cdot \hat{s}_i)| &= \sum_{i=1}^{d}(q_i \cdot |s_i - \hat{s}_i|) \\
&\leq \sum_{i=1}^{d}(q_i \cdot \frac{1}{2^{b+1}}) \\
&= \frac{1}{2^{b+1}} \sum_{i=1}^{d} q_i \\
&\leq \frac{1}{2^{b+1}} \sqrt{d \cdot \sum_{i=1}^{d} q_i^2} \\
&= \frac{\sqrt{d}}{2^{b+1}}
\end{aligned}
$$

As an example, Table 4.3 shows the error bound assuming d is 100. It shows that the error bound is very small.

Table 4.3: Error bound

b (bits)	absolute error (d = 100)
8	0.019531
9	0.009766
10	0.004883
11	0.002441
12	0.001221
13	0.00061
14	0.000305
15	0.000153
16	0.000076

Next, we compute the score lower bound and upper bound for verification. Note that:

$$\hat{s}_i - \varepsilon \leq s_i \leq \hat{s}_i + \varepsilon \text{ where } \varepsilon = \frac{1}{2^{b+1}}$$

$$\begin{aligned} lower\ bound &= \sum_{i=1}^{k}(q_i \cdot s_i) \\ &\geq \sum_{i=1}^{k}(q_i \cdot (\hat{s}_i - \varepsilon)) \\ &= \sum_{i=1}^{k}(q_i \cdot \hat{s}_i) - \varepsilon \sum_{i=1}^{k} q_i \end{aligned}$$

$$\begin{aligned} upper\ bound &= \sum_{i=1}^{k}(q_i \cdot s_i) + \sqrt{1 - \sum_{i=k+1}^{d} q_i^2} \cdot \sqrt{1 - \sum_{i=k+1}^{d} s_i^2} \\ &\leq \sum_{i=1}^{k}(q_i \cdot (\hat{s}_i + \varepsilon)) + \sqrt{1 - \sum_{i=k+1}^{d} q_i^2} \cdot \sqrt{1 - \sum_{i=k+1}^{d} (\hat{s}_i - \varepsilon)^2} \end{aligned}$$

4.7.4 Query processor

We implement the query processor within every bucket such that only relevant buckets are investigated to reduce the search space. However, we also add another optimization of clustering to make the lists within each cluster skew. We then develop a pruning condition such that a whole cluster of spectra can be pruned directly.

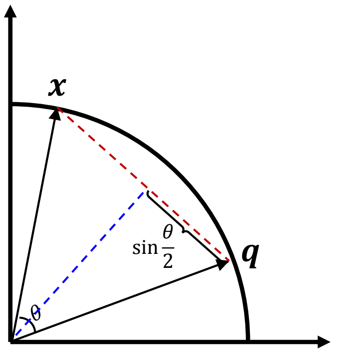


Figure 4.15: Relationship between cosine distance and Euclidean distance

We first show that on normalized vectors, the Euclidean distance and cosine similarity are equivalent. In particular, if the angle of two normalized vectors $q(q_1, q_2, ..., q_d)$ and $x(x_1, x_2, ..., x_d)$ is θ (i.e., their cosine similarity is $\cos\theta$), then their Euclidean distance $r_\theta = 2\sin(\theta/2)$. Figure 4.15

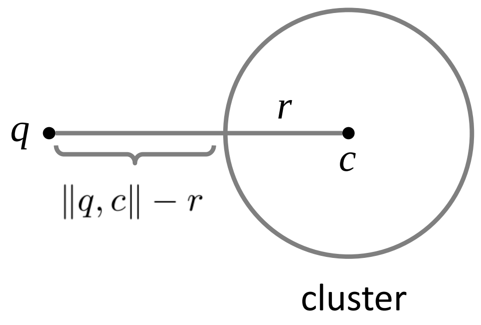


Figure 4.16: Illustration of clustering pruning

shows their relationship.

Thus, if $\|q,c\| - r > r_\theta$ where c is the center of a cluster, r is the radius of the cluster, and r_θ is the corresponding radius of the query threshold θ, then the whole cluster can be pruned directly, see Figure 4.16.

Next, we show the optimizations of batch query processing where a collection of queries are received. We first sort all the query spectra based on peptide mass values. In this way, those query spectra that share the same peptide mass value will be routed to the same bucket for searching. As a result, we only need to load the index and data of that bucket once to reduce disk I/O significantly.

Besides that, our techniques can also be extended to handle dimension tolerance. Let $\langle k_i, v_i \rangle$ be the i-th peak of a spectrum where k_i and v_i are float numbers, δ be the peak granularity (e.g., $\delta = 0.01$), ε be the dimension tolerance (e.g., $\varepsilon = 0.05$), then the dimensions that k_i matches range from $\frac{k_i}{\delta} - \frac{\varepsilon}{\delta}$ to $\frac{k_i}{\delta} + \frac{\varepsilon}{\delta}$. During query processing, we need to match k_i to all those relevant lists as well. A naive implementation would be merging the relevant lists off-line in the index building phase. However, it increases the space overhead. A space-efficient solution is to only store the convex hull of the merged lists and access the "virtual" merged list on-line.

4.8 Experiments

In this section, we present the experimental results.

4.8.1 Experimental setting

We conduct experiments on an Intel server (Xeon CPU E5-2667 v4 @ 3.20GHz, 12 physical cores, 250GB DRAM) with CentOS 6.6 installed. There is a 1TB SSD attached to the server. We implement the system using C++.

The dataset consists of a collection of 13.3 million spectra. We randomly pick up 1000 queries and evaluate the overall query time.

4.8.2 Experimental results

In the first experiment, we show the space overhead, see Table 4.4. The index size is slightly higher than the data size due to extra convex hull points built per list.

Table 4.4: Space overhead

number of spectra	13.3 million
data size	10GB
index size	12GB

Next, we compare our method with two other solutions: TA and WHOLE, where WHOLE means to scan all the whole lists within a bucket without pruning. All the algorithms are performed on a single bucket. We set the default query threshold θ as 0.6,[11] which is a commonly accepted parameter in the mass spectrometry domain. Figure 4.17 shows the results. It shows that our algorithm is 7.2$\times$ faster than the WHOLE algorithm that scans the whole lists; it is 2.8$\times$ faster than TA. The main reason is that our algorithm adopts an advanced stopping condition that terminates much earlier and a better traversal strategy that minimizes the index accesses.

[11]Note that we will vary the parameter θ later on.

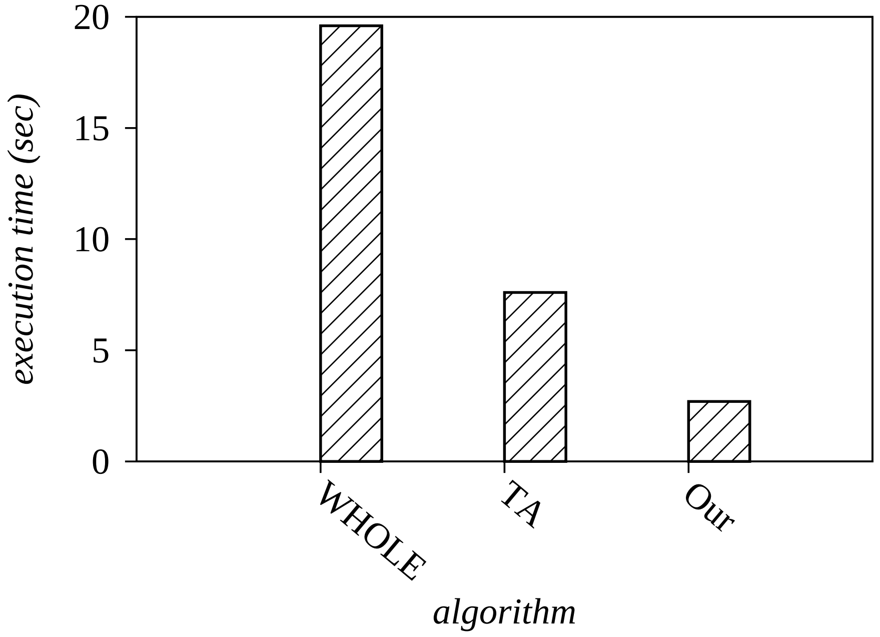
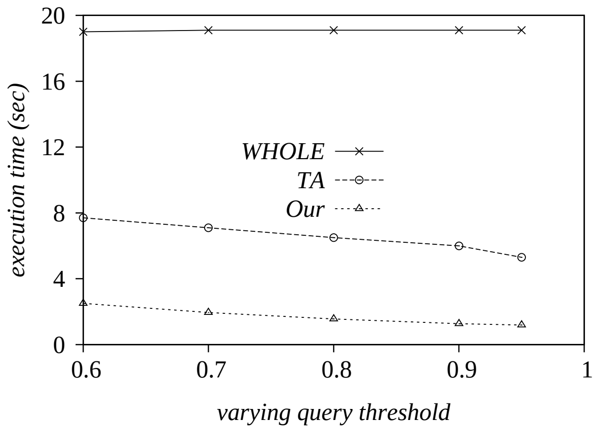


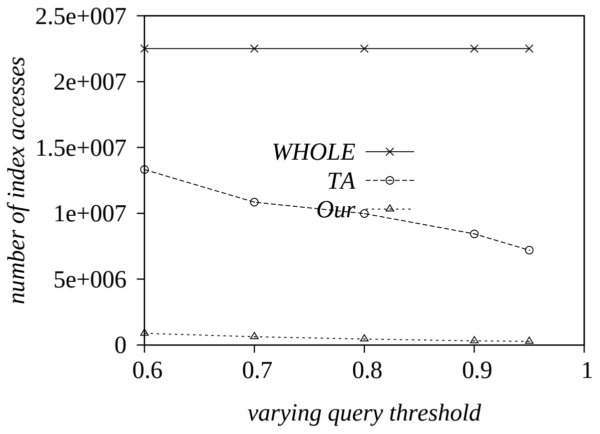
Figure 4.17: Query execution time

Table 4.5 shows the detailed time breakdown (i.e., gathering time and verification time) and statistics to understand our algorithm. The statistics include the number of index accesses, candidate size, and result size. In which, the number of index accesses is defined as $\sum_{i=1} b_i$ where b_i is the stopping position of list L_i. It shows that gathering phase is the performance bottleneck for all the three algorithms. Our algorithm can reduce both index accesses and number of candidates significantly when compared to other algorithms. Table 4.5 shows that the gathering time is roughly propositional to the number of index accesses and verification time is roughly propositional to the candidate set size. Besides that, it is also interesting to see from Table 4.5 that in our algorithm, the ratio of candidate set size and the final result set size is 427418 / 210310 = 2$\times$, meaning that our algorithm incurs at most twice of the index accesses when compared to the optimal algorithm that may not exist.

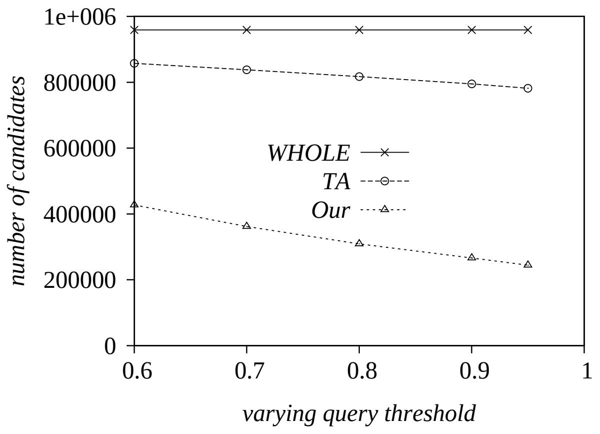
Next, we evaluate the algorithms with varying query threshold θ and report the query execution time, number of index accesses and number of candidates in Figure 4.18. It shows that for WHOLE, it is not sensitive to query threshold θ because it needs to access the whole lists



(a) query execution time



(b) number of index accesses



(c) number of candidates

Figure 4.18: Varying query threshold

Table 4.5: Query execution statistics

	WHOLE	TA	Our
Number of index accesses	22511856	13321642	884847
Candidate size	958836	857407	427418
Result size	210310	210310	210310
Gathering time (sec)	17.7	6.5	1.9
Verification time (sec)	1.3	1.2	0.6

regardless of the query threshold. However, for TA and our method, they become faster as the query threshold goes higher because of early termination. Thus, the performance gap increases as θ becomes bigger. For example, when $\theta = 0.95$, our algorithm is $16\times$ faster than WHOLE. And in all cases, our algorithm is significantly faster than TA.

4.9 Discussion

4.9.1 Traversal starting from the middle

Next, we consider a variant of the Gathering-Verification algorithm starting the traversal from the middle of the inverted lists instead of starting from the top like the classic TA. In the most general setting, the starting position can be anywhere in the inverted lists, and two pointers (upper and lower) are used to indicate the traversed range for each inverted list. At each iteration, in addition to choosing the list to be traversed, the traversal strategy also decides whether the upper or the lower pointer of the selected list needs to be moved. We can show that in this setting, deciding a tight and complete stopping condition is NP-HARD thus intractable.

We start with some definitions. A *configuration* of the traversal is a pair of position vector $(\mathbf{a}, \mathbf{b})$ where $\mathbf{a}$ and $\mathbf{b}$ are the upper and the lower position vector of the inverted lists $\{L_i\}_{i\in[d]}$ respectively. At each configuration, it is guaranteed that all vectors $\mathbf{s}$ satisfying $L_i[b_i] \le s_i \le L_i[a_i]$ for some dimension i have been collected in the candidate set. So the tight stopping condition needs to test whether all the remaining vector $\mathbf{s}$ can be θ-similar to the query $\mathbf{q}$. For each dimension i, the condition that s_i needs to satisfy is

$$s_i \geq L_i[a_i] \vee s_i \leq L_i[b_i] \ ,$$

which can be written into an equivalent quadratic form

$$(s_i - L_i[a_i]) \cdot (s_i - L_i[b_i]) \geq 0$$

since $L_i[a_i] \geq L_i[b_i]$.

Thus, the tight stopping condition is equivalent to the following quadratic program:

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{d} s_i \cdot q_i \\ \text{subject to} \quad & \sum_{i=1}^{d} s_i^2 = 1, \\ & (s_i - L_i[a_i]) \cdot (s_i - L_i[b_i]) \geq 0, \quad \text{for } i \in [d]. \end{aligned} \tag{4.18}$$

Let

$$F(\mathbf{s}, \mu, \lambda) = \sum_{i=1}^{d} s_i q_i + \sum_{i=1}^{d} \mu_i (s_i - L_i[a_i]) \cdot (s_i - L_i[b_i]) - \lambda \left(\sum_{i=1}^{d} s_i^2 - 1 \right)$$

be the Lagrangian of (4.18) where $\lambda \in \mathbb{R}$ and $\mu \in \mathbb{R}^d$ are the Lagrange multipliers. Then, the KKT conditions of (4.18) are

$$\begin{aligned} & \nabla_{\mathbf{s}} F(\mathbf{s}, \mu, \lambda) = 0 && \text{(Stationarity)} \\ & \mu_i \geq 0, \ \forall\, i \in [d] && \text{(Dual feasibility)} \\ & \mu_i (s_i - L_i[a_i])(s_i - L_i[b_i]) = 0, \ \forall\, i \in [d] && \text{(Complementary slackness)} \end{aligned}$$

in addition to the Primal feasibility in (4.18).

By the Complementary slackness condition, either $\mu_i = 0$ or $(s_i - L_i[a_i])(s_i - L_i[b_i]) = 0$. If $\mu_i = 0$, from the Stationarity condition, we have

$$q_i + \mu_i(s_i/2 - L_i[a_i] - L_i[b_i]) - \lambda s_i = 0$$

so $s_i = q_i/\lambda$.

If $\mu_i \neq 0$, then $(s_i - L_i[a_i])(s_i - L_i[b_i]) = 0$ so $s_i = L_i[a_i]$ or $s_i = L_i[b_i]$. Since $\mathbf{s} \cdot \mathbf{q}$ is maximized when s_i is proportional to q_i and $s_i \geq L_i[a_i] \vee s_i \leq L_i[b_i]$, the value of s_i is determined as follows:

- if $q_i/\lambda \in [L_i[b_i], L_i[a_i]]$, then s_i equals either the upper bound $L_i[a_i]$ or the lower bound $L_i[b_i]$;
- otherwise, $s_i = q_i/\lambda$.

Here, we can see that solving the quadratic program (4.18) is more difficult than the quadratic program (4.7) in Section 4.4.2 where the traversal starts from the top of the lists. Intuitively, when $s_i \neq q_i/\lambda$, unlike the solution for (4.7), there are still two possible options: the upper bound $L_i[a_i]$ and the lower bound $L_i[b_i]$, which lead to exponentially many combinations. One can easily show that even when λ is fixed, computing the optimal $\mathbf{s}$ is NP-HARD by reduction from the subset sum problem [Kar72]. This is because when λ is fixed, the s_i's that equal to q_i/λ are fixed. When $L_i[b_i] = 0$ for all i, checking whether $\sum_{i=1}^{d} s_i^2 = 1$ is essentially checking whether there exists a subset of $\{L_i[a_i]^2\}_{i \in [d]}$ whose sum is some fixed constant depending on λ.

4.9.2 Applying the proposed techniques to top-k cosine queries

We show that the proposed stopping condition and traversal strategy can also be applied to *top-k* cosine queries. In the top-k setting, each query is a pair $(\mathbf{q}, k)$ where $\mathbf{q}$ is a query vector and k is an integer parameter. A query $(\mathbf{q}, k)$ asks for the top k database vectors with the highest cosine similarity with the query vector $\mathbf{q}$.

The proposed techniques can be applied by adapting the classic structure of TA. The algorithm traverses the 1-d inverted lists and keeps track of the k-th highest similarity score among the gathered vectors. Note that to guarantee tightness, the exact similarity scores need to be computed online, which results in additional computation cost since the exact computation can be avoided for threshold queries using partial verification (Section 4.6). The stopping condition can be adapted as follows.

Theorem 9 *At a position vector* $\mathbf{b}$*, let* θ_k *be the k-th highest score among vectors on or above* $\mathbf{b}$*. Then the following stopping condition is tight and complete:*

$$\varphi_{\textit{top-k}}(\mathbf{b}) = \Big(\mathsf{MS}(L[\mathbf{b}]) < \theta_k\Big).$$

The score $\mathsf{MS}(L[\mathbf{b}])$ can be computed using the same $O(\log d)$ incremental maintenance algorithm in Section 4.4.3. The lower bound θ_k needs to be updated when a new candidate is gathered. Computing the similarity score takes $O(d)$ time and updating θ_k can be done in $O(\log k)$ using a binary heap.

The hull-based traversal strategy for inner product threshold queries can be directly applied to top-k inner product queries. The following near-optimality result can be shown.

Theorem 10 *For a top-k inner product query* $(\mathbf{q}, k)$*, the access cost of the hull-based traversal strategy* $\mathcal{T}_{\mathsf{HL}}$ *on a near-convex database* $\mathcal{D}$ *is at most* $\mathsf{OPT} + c$ *where c is the convexity constant of* $\mathcal{D}$*.*

The hull-based strategy can also be applied to top-k cosine queries. Recall that for cosine threshold queries, the hull-based traversal strategy operates on the convex hulls of a decomposable approximation $\tilde{F}$ of MS where $\tilde{F}(L[\mathbf{b}]) = \sum_{i=1}^{d} \min\{q_i \cdot \tilde{\tau}, L_i[b_i]\} \cdot q_i$. As discussed in Section 4.5.5, the choice of the constant $\tilde{\tau}$ is ideally the value τ of the optimal traversal path, which is dependent on the threshold θ. Therefore, for top-k cosine queries where the final threshold θ_k is unknown in advance, a bad choice of $\tilde{\tau}$ can lead to poor approximation. In a practical implementation, the constant $\tilde{\tau}$ can be made query-dependent and more carefully tuned. We leave these aspects for future work.

4.10 Related work

In this section, we present the related work.

4.10.1 Cosine similarity search

The cosine threshold querying studied in this work is a special case of the *cosine similarity search* (CSS) problem [BMS07, AK14, AK15] mentioned in Section 4.1. Thus, we first survey the techniques developed for CSS.

LSH. A widely used technique for cosine similarity search is locality-sensitive hash (LSH) [RU11, AIL$^+$15, HTY17, IM98, TYSK09]. The main idea of LSH is to partition the whole database into buckets using a series of hash functions such that similar vectors have high probability to be in the same bucket. However, LSH is designed for *approximate* query processing, meaning that it is not guaranteed to return all the true results. In contrast, this work focuses on exact query processing which returns all the results.

TA-family algorithms. Another technique for cosine similarity search is the family of TA-like algorithms. Those algorithms were originally designed for processing top-k ranking queries that find the top k objects ranked according to an aggregation function (see [IBS08] for a survey). We have summarized the classic TA algorithm [FLN01], presented a baseline algorithm inspired by it, and explained its shortcomings in Section 4.1. The Gathering-Verification framework introduced in Section 4.2 captures the typical structure of the TA-family when applied to our setting.

The variants of TA (e.g., [GBK00, BMS$^+$06, DPK08, BGM02]) can have poor or no performance guarantee for cosine threshold queries since they do not fully leverage the data skewness and the unit vector condition. For example, Güntzer et al. developed Quick-Combine [GBK00]. Instead of accessing all the lists in a lockstep strategy, it relies on a heuristic traversal strategy to access the list with the highest rate of changes to the ranking function in a fixed number of steps ahead. It was shown in [FLN03] that the algorithm is not instance optimal. Although the hull-based traversal strategy proposed in this work roughly follows the same idea, the number of steps to look ahead is variable and determined by the next convex hull vertex. Thus, for decomposable functions, the hull-based strategy makes globally optimal decisions and is near-optimal under the near-convexity assumption, while Quick-Combine has no performance guarantee because of the fixed step size even when the data is near-convex.

Bast et al. studied top-k ranked query processing with a different goal of minimizing the overall cost by scheduling the best sorted accesses and random accesses [BMS$^+$06]. However, when the number of dimensions is high, it requires a large amount of online computations to frequently solve a Knapsack problem, which can be slow in practice. Note that, [BMS$^+$06] only evaluated the number of sorted and random accesses in the experiments instead of the wall-clock time. Besides, it does not provide any theoretical guarantee, which is also applicable to [JP11]. Akbarinia et al. proposed BPA to improve TA, but the optimality ratio is the same as TA in the worst case [APV07]. Deshpande et al. solved a special case of top-k problem by assuming that the attributes are drawn from a small value space [DPK08]. Zhang et al. developed an algorithm targeting for a large number of lists [ZSH16] by merging lists into groups and then apply TA. However, it requires the ranking function to be distributive that does not hold for the cosine similarity function. Yu et al. solved top-k query processing in subspaces [YAY14] that are not applicable to general cosine threshold queries.

Some works considered non-monotonic ranking functions [ZHC$^+$06, XHC07]. For example, [ZHC$^+$06] focused on the combination of a boolean condition with a regular ranking function and [XHC07] assumed the ranking functions are lower-bounded. The cosine function has not been considered in this line of work.

COORD. Teflioudi et al. proposed the COORD algorithm based on inverted lists for CSS [TGM15, TG16]. The main idea is to scan the whole lists but with an optimization to prune irrelevant entries using upper/lower bounds of the cosine similarity with the query. Thus, instead of traversing the whole lists starting from the top, it scans only those entries within a feasible range. We can also apply such a pruning strategy to the Gathering-Verification framework by starting the gathering phase at the top of the feasible range. However, there is no optimality guarantee of the algorithm. Also the optimization only works for high thresholds (e.g., 0.95), which are not always the requirement. For example, a common and well-accepted threshold in mass spectrometry search is 0.6, which is a medium-sized threshold, making the effect of the pruning negligible.

Partial verification. Anastasiu and Karypis proposed a technique for fast verification of θ-similarity between two vectors [AK14] without a full scan of the two vectors. We can apply the same optimization to the verification phase of the Gathering-Verification framework. Additionally, we prove that it has a novel near-constant performance guarantee in the presence of data skewness. See Section 4.6.

Other variants. There are several studies focusing on cosine similarity join to find out all pairs of vectors from the database such that their similarity exceeds a given threshold [BMS07, AK14, AK15]. However, this work is different since the focus is comparing to a given query vector $\mathbf{q}$ rather than join. As a result, the techniques in [BMS07, AK14, AK15] are not directly applicable: (1) The inverted index is built online instead of offline, meaning that at least one full scan of the whole data is required, which is inefficient for search.[12] (2) The index in [BMS07, AK14, AK15] is built for a fixed query threshold, meaning that the index cannot be used for answering arbitrary query thresholds as concerned in this work. The theoretical aspects of similarity join were discussed recently in [APRS16, HTY17].

[12]The linear scan is feasible for join because the naive approach of join is quadratic.

4.10.2 Euclidean distance threshold queries

The cosine threshold queries can also be answered by techniques for distance threshold queries (the threshold variant of nearest neighbor search) in Euclidean space. This is because there is a one-to-one mapping between the cosine similarity θ and the Euclidean distance r for unit vectors, i.e., $r = 2\sin(\arccos(\theta)/2)$. Thus, finding vectors that are θ-similar to a query vector is equivalent to finding the vectors whose Euclidean distance is within r. Next, we review exact approaches for distance queries while leaving the discussion of approximate approaches in Section 4.10.7. There are four main types of techniques for exact approaches: tree-based indexing, pivot-based indexing, clustering, and dimensionality reduction.

Tree-based indexing. Several tree-based indexing techniques (such as R-tree, KD-tree, Cover-tree [BKL06]) were developed for range queries (so they can also be applied to distance queries), see [BBK01] for a survey. However, they are not scalable to high dimensions (say thousands of dimensions as studied in this work) due to the well known dimensionality curse issue [WSB98].

Pivot-based indexing. The main idea is to pre-compute the distances between data vectors and a set of selected pivot vectors. Then during query processing, use triangle inequalities to prune irrelevant vectors [CGZ+17, HKP01]. However, it does not scale in high-dimensional space as shown in [CGZ+17] since it requires a large space to store the pre-computed distances.

Clustering-based (or partitioning-based) methods. The main idea of clustering is to partition the database vectors into smaller clusters of vectors during indexing. Then during query processing, irrelevant clusters are pruned via the triangle inequality [Sam05, RR11]. Clustering is an optimization orthogonal to the proposed techniques, as they can be used to process vectors within each cluster to speed up the overall performance.

Dimensionality reduction. Since many techniques are affected negatively by the large number of dimensions, one potential solution is to apply dimensionality reduction (e.g. PCA, Johnson-Lindenstrauss) [LLW10, LC09, JLS86] before any search algorithm. However, this does not help much if the dimensions are not correlated and there are no hidden latent variables to be uncovered by dimensionality reduction. For example, in the mass spectrometry domain, each dimension represents a chemical compound/element and there is no physics justifying a correlation. We applied dimensionality reduction to the data vectors and turned out that only 4.3% of dimensions can be removed in order to preserve 99% of the distance.

4.10.3 Inner product search

The cosine threshold querying is also related to inner product search where vectors may not be unit vectors, otherwise, inner product search is equivalent to cosine similarity search.

Teflioudi et al. proposed the LEMP framework [TGM15, TG16] to solve the inner product search where each vector may not be normalized. The main idea is to partition the vectors into buckets according to vector lengths and then apply an existing cosine similarity search (CSS) algorithm to each bucket. This work provides an efficient way for CSS that can be integrated to the LEMP framework.

Li et al. developed an algorithm FEXIPRO [LCYM17] for inner product search in recommender systems where vectors might contain negative values. Since we focus on non-negative values in this work, the proposed techniques (such as length-based filtering and monotonicity reduction) are not directly applicable.

There are also tree-based indexing and techniques for inner product search [RG12,CGR13]. But they are not scalable to high dimensions [KSR17]. Another line of research is to leverage machine learning to solve the problem [ME16, FPW16, SLZ$^+$15]. However, they do not provide accurate answers. Besides, they do not have any theoretical guarantee.

4.10.4 CSS in hamming space

[ET16] and [QWX+18] studied cosine similarity search in the Hamming space where each vector contains only binary values. They proposed to store the binary vectors efficiently into multiple hash tables. However, the techniques proposed there cannot be applied since transforming real-valued vectors to binary vectors loses information, and thus correctness is not guaranteed.

4.10.5 Keyword search

This work is different from keyword search although the similarity function is (a variant of) the cosine function and the main technique is inverted index [MRS08]. There are two main differences: (1) keyword search generally involves a few query terms [WLH+17]; (2) keyword search tends to return Web pages that contain all the query terms [MRS08]. As a result, search engines (e.g., Apache Lucene) mainly sort the inverted lists by document IDs [BCH+03, DS11, CCG11] to facilitate boolean intersection. Thus, those algorithms cannot be directly applied to cosine threshold queries. Although there are cases where the inverted lists are sorted by document frequency (or score in general) that are similar to this work, they usually follow TA [DS11]. This work significantly improves TA for cosine threshold queries.

4.10.6 Mass spectrometry search

For mass spectrometry search, the state-of-the-art approach is to partition the spectrum database into buckets based on the spectrum mass (i.e., molecular weight) and only search the buckets that have the similar mass to the query [KLA+17]. Each bucket is scanned linearly to obtain the results. The proposed techniques in this work can be applied to each bucket and improve the performance dramatically.

Library search, wherein a spectrum is compared to a series of reference spectra to find the most similar match, providing a putative identification for the query spectrum [YMG+98]. Methods in library search often use cosine [LDE+07] or cosine-derived [WB13] functions to compute spectrum similarity. Many of the state-of-the-art library search algorithms, e.g., SpectraST [LDE+07], Pepitome [DCM+12], X!Hunter [CCFB06], and [WCB16] find candidates by doing a linear scan of peptides in the library that are within a given parent mass tolerance and computes a distance function against relevant spectra. This is likely because the libraries that are searched against have been small, but spectral libraries are getting larger, e.g., MassIVE-KB now has more than 2.1 million precursors. This work fundamentally improves on these, as it uses inverted lists with many optimizations to significantly reduce the candidate spectra from comparison. M-SPLIT used both a prefiltering and branch and bound technique to reduce the candidate set for each search [WPSK+10]. This work improves on both these by computing the similar spectra directly, while still applying the filtering.

Database search [EMY94] is where the experimental spectrum is compared to a host of theoretical spectra that are generated from a series of sequences. The theoretical spectra differ from experimental spectra in that they do not have intensities for each peak and contain all peaks which could be in the given peptide fragmentation. In other words, the values in the vectors are not important and the similarity function evaluates the number of shared dimensions that are irrelevant to the values. While the problem is different in practice than that described in this work, it is still interesting to consider optimizations. Dutta and Chen proposed an LSH-based technique which embeds both the experimental spectra and the theoretical spectra onto a higher dimensional space [DC07]. While this method performs well at the expense of low recall rate, our method is guaranteed to not lose any potential matches. There are also other methods optimizing this problem by using preindexing [THS+05, KLA+17] but none do this kind of preindexing for calculating matches between experimental spectra.

4.10.7 Approximate approaches

Besides LSH, there are many other approximate approaches for high-dimensional similarity search, e.g., graph-based methods [FWC17, DML11, WJZ14], product quantization [JDS11, AKLS15, Lem12], randomized KD-trees [SAH08], priority search k-means tree [ML14], rank cover tree [HN15], randomized algorithms [WSWR18], HD-index [ASKB18], and clustering-based methods [LCGMW02, CZC10].

4.10.8 Others

This work is also different from [LDLL18] because that work focused on similarity search based on set-oriented p-norm similarity, which is different from cosine similarity.

In the literature, there are skyline-based approaches [LCH12] to solve a special case of top-k ranked query processing where no any ranking function is specified. However, those approaches cannot be used to solve the mass spectrometry problem that only involves *unit vectors* such that all the vectors belong to skyline points.

4.11 Chapter summary

In this chapter, we proposed optimizations to the index-based, TA-like algorithms for answering the cosine threshold queries, which lies at the core of mass spectrometry. The novel techniques include a complete and tight stopping condition computable in $O(\log d)$ time and a family of convex hull-based traversal strategies with near-optimality guarantees for a larger class of decomposable functions beyond cosine. With these techniques, we show near-optimality first for inner-product threshold queries, then extend the result to the full cosine threshold queries using approximation. All these results are significant improvements compared to a baseline approach inspired by the classic TA algorithm. In addition, we have verified with experiments on real data the assumptions required by the near-optimality results.

Chapter 4 contains material from “Index-based, High-dimensional, Cosine Threshold Querying with Optimality Guarantees” by Yuliang Li, Jianguo Wang, Benjamin Pullman, Nuno Bandeira, and Yannis Papakonstantinou, which appeared in Proceedings of International Conference on Database Theory (ICDT), 2019. The dissertation author was the primary investigator of the paper.

Chapter 5

Conclusion and future work

5.1 Conclusion

The continued evolution of computing hardware and infrastructure imposes new challenges and opportunities to query engines. In this dissertation, we studied the impact of modern hardware to query processing of sorted lists (both ID-sorted and value-sorted). In particular, we investigated the how modern processors and big memory affect list compression (Chapter 2), how SSDs affect list intersection (Chapter 3), and then showed how to apply the knowledge of modern hardware to a real application – mass spectrometry search – that heavily relies on query processing of sorted lists (Chapter 4).

We conclude that directly running the existing algorithms and systems on modern hardware improves performance somewhat but misses important opportunities for further improvement. Thus, we shall revisit existing designs and rethink new designs by explicitly using the characteristics of emerging hardware.

5.2 Future work

There could be many interesting follow-ups from this thesis. Below we name a few.

- Efficient inverted list update on SSDs. Traditionally on HDDs, each inverted list is stored *continuously* to minimize the expensive random accesses [ZM06]. As a result, for the update, it incurs a lot of data movement to guarantee the continuity [LMZ05]. However, this work indicates that on SSDs, the inverted list is not required to be stored continuously since the entire list is not required during query processing. Thus, it is interesting to explore efficient update solutions with minimal data movement.

- Extension of list intersection to next-generation ultra-fast storage, e.g., Intel 3D XPoint SSDs and non-volatile memories (NVRAM) [XS16]. With new storage devices, the idea of skipping-on-disk is still relevant because they support fast random accesses and parallel I/O access. But the interesting thing is that the I/O is fast enough to be blurred with memory. For example, it is expected that the new generation NVRAM has similar performance with DRAM [APD15]. As a result, it is interesting to investigate whether it is still worthy to cache inverted list in memory or in general how to best utilize DRAM and NVRAM for efficient query processing.

- Extension of list intersection to hybrid storage systems that include both SSDs and HDDs. As we show in this work, SSDs and HDDs exhibit different properties such that they demand different intersection algorithms. Thus, it is interesting to investigate how to design algorithms that are applicable for heterogeneous storage devices.

- Extension of MILC. Currently, MILC is optimized for main memory. It can also be extended to other storage devices including NVRAM, SSDs [WLY+13, WLY+14], and HDDs. Another direction is to tailor MILC for supporting in-storage computing [WPK+16, WPPS16]. Besides that, it is also interesting to extend MILC to support more operations and queries, e.g., intersection, union, top-k query processing.

- Extension of techniques of threshold queries to support more types of queries in mass spectrometry area, e.g., cosine similarity with shifted peaks or simply with the number of common peaks [KLA$^+$17]. The underlying index structure should be similar but need more tweaks for high performance.

Bibliography

[AFG$^+$10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *CACM*, 53(4):50–58, 2010.

[AIL$^+$15] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. Practical and optimal lsh for angular distance. In *NIPS*, pages 1225–1233, 2015.

[AK14] David C. Anastasiu and George Karypis. L2AP: Fast cosine similarity search with prefix L-2 norm bounds. In *ICDE*, pages 784–795, 2014.

[AK15] David C. Anastasiu and George Karypis. PL2AP: Fast parallel cosine similarity search. In *IA3*, pages 8:1–8:8, 2015.

[AKLS15] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. Cache locality is not enough: High-performance nearest neighbor search with product quantization fast scan. *PVLDB*, 9(4):288–299, 2015.

[AM05] Vo Ngoc Anh and Alistair Moffat. Inverted index compression using word-aligned binary codes. *IR*, 8(1):151–166, 2005.

[AM10] Vo Ngoc Anh and Alistair Moffat. Index compression using 64-bit words. *SPE*, 40(2):131–147, 2010.

[AM16] Ruedi Aebersold and Matthias Mann. Mass-spectrometric exploration of proteome structure and function. *Nature*, 537:347–355, 2016.

[APD15] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *SIGMOD*, pages 707–722, 2015.

[APRS16] Thomas Dybdahl Ahle, Rasmus Pagh, Ilya Razenshteyn, and Francesco Silvestri. On the complexity of inner product similarity join. In *PODS*, pages 151–164, 2016.

[APV07] Reza Akbarinia, Esther Pacitti, and Patrick Valduriez. Best position algorithms for top-k queries. In *VLDB*, pages 495–506, 2007.

[AS10] D. G. Andersen and Steven Swanson. Rethinking flash in the data center. *IEEE Micro*, 30(4):52–54, 2010.

[ASKB18] Akhil Arora, Sakshi Sinha, Piyush Kumar, and Arnab Bhattacharya. HD-Index: Pushing the scalability-accuracy boundary for approximate knn search in high-dimensional spaces. *PVLDB*, 11(8):906–919, 2018.

[BBK01] Christian Böhm, Stefan Berchtold, and Daniel A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *CSUR*, 33(3):322–373, 2001.

[BCH$^+$03] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. In *CIKM*, pages 426–434, 2003.

[BGGT09] Truls A. Bjorklund, Nils Grimsmo, Johannes Gehrke, and Oystein Torbjornsen. Inverted indexes vs. bitmap indexes in decision support systems. In *CIKM*, pages 1509–1512, 2009.

[BGM02] Nicolas Bruno, Luis Gravano, and Amélie Marian. Evaluating top-k queries over web-accessible databases. In *ICDE*, pages 369–380, 2002.

[BK02] Jérémy Barbay and Claire Kenyon. Adaptive intersection and t-threshold problems. In *SODA*, pages 390–399, 2002.

[BKL06] Alina Beygelzimer, Sham Kakade, and John Langford. Cover trees for nearest neighbor. In *ICML*, pages 97–104, 2006.

[BKS02] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: Optimal xml pattern matching. In *SIGMOD*, pages 310–321, 2002.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7):422–426, 1970.

[BMS$^+$06] Holger Bast, Debapriyo Majumdar, Ralf Schenkel, Martin Theobald, and Gerhard Weikum. Io-top-k: Index-access optimized top-k query processing. In *VLDB*, pages 475–486, 2006.

[BMS07] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.

[BPP07] Philip Bille, Anna Pagh, and Rasmus Pagh. Fast evaluation of union-intersection expressions. In *ISAAC*, pages 739–750, 2007.

[BRM98] Guy E. Blelloch and Margaret Reid-Miller. Fast set operations using treaps. In *SPAA*, pages 16–26, 1998.

[BV04] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[BYSC05] Ricardo Baeza-Yates, Ro Salinger, and Santiago Chile. Experimental analysis of a fast intersection algorithm for sorted sequences. In *SPIRE*, pages 13–24, 2005.

[CBB$^+$13] Michael Curtiss, Iain Becker, Tudor Bosman, Sergey Doroshenko, Lucian Grijincu, Tom Jackson, Sandhya Kunnatur, Soren Lassen, Philip Pronin, Sriram Sankar, Guanghao Shen, Gintaras Woss, Chao Yang, and Ning Zhang. Unicorn: A system for searching the social graph. *PVLDB*, 6(11):1150–1161, 2013.

[CCFB06] R Craig, J C Cortens, D Fenyo, and R C Beavis. Using annotated peptide mass spectrum libraries for protein identification. *Journal of Proteome Research*, 5(8):1843–1849, 2006.

[CCG11] Kaushik Chakrabarti, Surajit Chaudhuri, and Venkatesh Ganti. Interval-based pruning for top-k processing over compressed lists. In *ICDE*, pages 709–720, 2011.

[CGR13] Ryan R. Curtin, Alexander G. Gray, and Parikshit Ram. Fast exact max-kernel search. In *SDM*, pages 1–9, 2013.

[CGZ$^+$17] Lu Chen, Yunjun Gao, Baihua Zheng, Christian S. Jensen, Hanyu Yang, and Keyu Yang. Pivot-based metric indexing. *PVLDB*, 10(10):1058–1069, 2017.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[CM07] J. Shane Culpepper and Alistair Moffat. Compact set representation for information retrieval. In *SPIRE*, pages 137–148, 2007.

[CM10a] J. Shane Culpepper and Alistair Moffat. Efficient set intersection for inverted indexing. *TOIS*, 29(1):1–25, 2010.

[CM10b] J. Shane Culpepper and Alistair Moffat. Efficient set intersection for inverted indexing. *TOIS*, 29(1):1–25, 2010.

[CP90] Douglas R. Cutting and Jan O. Pedersen. Optimizations for dynamic inverted index maintenance. In *SIGIR*, pages 405–411, 1990.

[CZC10] Bin Cui, Jiakui Zhao, and Gao Cong. ISIS: A new approach for efficient similarity search in sparse databases. In *DASFAA*, pages 231–245, 2010.

[DBCVKO08] Mark De Berg, Otfried Cheong, Marc Van Kreveld, and Mark Overmars. *Computational Geometry: Introduction*. Springer, 2008.

[DC07] Debojyoti Dutta and Ting Chen. Speeding up tandem mass spectrometry database search: metric embeddings and fast near neighbor search. *Bioinformatics*, 23(5):612–618, 2007.

[DCM$^+$12] Surendra Dasari, Matthew C Chambers, Misti A Martinez, Kristin L Carpenter, Amy-Joan L Ham, Lorenzo J Vega-Montoto, and David L Tabb. Pepitome: Evaluating improved spectral library search for identification complementarity and quality assessment. *Journal of Proteome Research*, 11(3):1686–95, 2012.

[Dea09] Jeffrey Dean. Challenges in building large-scale information retrieval systems: Invited talk. In *WSDM*, page 1, 2009.

[DK11] Bolin Ding and Arnd Christian König. Fast set intersection in memory. *PVLDB*, 4(4):255–266, 2011.

[DLOM00] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Adaptive set intersections, unions, and differences. In *SODA*, pages 743–752, 2000.

[DLOM01] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Experiments on adaptive set intersections for text retrieval systems. In *ALENEX*, pages 91–104, 2001.

[DML11] Wei Dong, Charikar Moses, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *WWW*, pages 577–586, 2011.

[DP09] Jaeyoung Do and Jignesh M. Patel. Join processing for flash SSDs: remembering past lessons. In *DaMoN*, pages 1–8, 2009.

[DPK08] Prasad M Deshpande, Deepak P, and Krishna Kummamuru. Efficient online top-k retrieval with arbitrary similarity measures. In *EDBT*, pages 356–367, 2008.

[DS11] Shuai Ding and Torsten Suel. Faster top-k document retrieval using block-max indexes. In *SIGIR*, pages 993–1002, 2011.

[DSL11] Biplob Debnath, Sudipta Sengupta, and Jin Li. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *SIGMOD*, pages 25–36, 2011.

[Eli74] Peter Elias. Efficient storage and retrieval by content and address of static files. *JACM*, 21(2):246–260, 1974.

[Eli75] P. Elias. Universal codeword sets and representations of the integers. *TOIT*, 21(2):194–203, 1975.

[EMY94] Jimmy K. Eng, Ashley L. McCormack, and John R. Yates. An approach to correlate tandem mass spectral data of peptides with amino acid sequences in a protein database. *Journal of the American Society for Mass Spectrometry*, 5(11):976–989, 1994.

[ET16] Sepehr Eghbali and Ladan Tahvildari. Cosine similarity search with multi index hashing. *CoRR*, abs/1610.00574, 2016.

[FLN01] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, 2001.

[FLN03] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *JCSS*, 66(4):614–656, 2003.

[FPW16] Marco Fraccaro, Ulrich Paquet, and Ole Winther. Indexable probabilistic matrix factorization for maximum inner product search. In *AAAI*, pages 1554–1560, 2016.

[FWC17] Cong Fu, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with navigating spreading-out graphs. *CoRR*, abs/1707.00143, 2017.

[GBK00] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kiebling. Optimizing multi-feature queries for image databases. In *VLDB*, pages 419–428, 2000.

[GBN14] Pedram Ghodsnia, Ivan T. Bowman, and Anisoara Nica. Parallel I/O aware query optimization. In *SIGMOD*, pages 349–360, 2014.

[GCC$^+$09] Laura M. Grupp, Adrian M. Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H. Siegel, and Jack K. Wolf. Characterizing flash memory: anomalies, observations, and applications. In *MICRO*, pages 24–33, 2009.

[GRS98] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing relations and indexes. In *ICDE*, pages 370–379, 1998.

[HKP01] Vagelis Hristidis, Nick Koudas, and Yannis Papakonstantinou. PREFER: A system for the efficient execution of multi-parametric ranked queries. In *SIGMOD*, pages 259–270, 2001.

[HL72] F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly-ordered sets. *SICOMP*, 1972.

[HN15] Michael E. Houle and Michael Nett. Rank-based similarity search: Reducing the dimensional dependence. *PAMI*, 37(1):136–150, 2015.

[HTY17] Xiao Hu, Yufei Tao, and Ke Yi. Output-optimal parallel algorithms for similarity joins. In *PODS*, pages 79–90, 2017.

[HX11] Bojun Huang and Zenglin Xia. Allocating inverted index into flash memory for search engines. In *WWW*, pages 61–62, 2011.

[IBS08] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A survey of top-k query processing techniques in relational database systems. *CSUR*, 40(4):1–58, 2008.

[IM98] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *ICDT*, pages 604–613, 1998.

[Int12] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 2012.

[JDS11] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *PAMI*, 33(1):117–128, 2011.

[JLFL14] Yu Jiang, Guoliang Li, Jianhua Feng, and Wen-Syan Li. String similarity joins: An experimental evaluation. *PVLDB*, 7(8):625–636, 2014.

[JLS86] William B. Johnson, Joram Lindenstrauss, and Gideon Schechtman. Extensions of lipschitz maps into banach spaces. *Israel Journal of Mathematics*, 54(2):129–138, 1986.

[JP11] Wen Jin and Jignesh M. Patel. Efficient and generic evaluation of ranked queries. In *SIGMOD*, pages 601–612, 2011.

[JRSP14] Wonmook Jung, Hongchan Roh, Mincheol Shin, and Sanghyun Park. Inverted index maintenance strategy for flashssds: Revitalization of in-place index update strategy. *Inf. Syst.*, 49:25–39, 2014.

[Kar72] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[KCS$^+$10] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. Fast: Fast architecture sensitive tree search on modern cpus and gpus. In *SIGMOD*, pages 339–350, 2010.

[KG09] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing for scalable image search. In *ICCV*, pages 2130–2137, 2009.

[KLA$^+$17] Andy T. Kong, Felipe V. Leprevost, Dmitry M. Avtonomov, Dattatreya Mellacheruvu, and Alexey I. Nesvizhskii. Msfragger: Ultrafast and comprehensive peptide identification in mass spectrometry-based proteomics. *Nature Methods*, 14:513–520, 2017.

[Knu73] Donald E. Knuth. *The Art of Computer Programming*. Addison Wesley Longman Publishing Co., 1973.

[KSR17] Omid Keivani, Kaushik Sinha, and Parikshit Ram. Improved maximum inner product search with better theoretical guarantees. In *IJCNN*, pages 2927–2934, 2017.

[KT14] Harold W Kuhn and Albert W Tucker. Nonlinear programming. In *Traces and Emergence of Nonlinear Programming*, pages 247–258. 2014.

[LB15a] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *SPE*, 45(1):1–29, 2015.

[LB15b] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *SPE*, 45(1):1–29, 2015.

[LC09] Xiang Lian and Lei Chen. General cost models for evaluating dimensionality reduction in high-dimensional spaces. *TKDE*, 21(10):1447–1460, 2009.

[LCGMW02] Chen Li, Edward Chang, Hector Garcia-Molina, and Gio Wiederhold. Clustering for approximate similarity search in high-dimensional spaces. *TKDE*, 14(4):792–808, 2002.

[LCH12] Jongwuk Lee, Hyunsouk Cho, and Seung-won Hwang. Efficient dual-resolution layer indexing for top-k queries. In *ICDE*, pages 1084–1095, 2012.

[LCL$^+$12] Ruixuan Li, Xuefan Chen, Chengzhou Li, Xiwu Gu, and Kunmei Wen. Efficient online index maintenance for SSD-based information retrieval systems. In *HPCC*, pages 262–269, 2012.

[LCYM17] Hui Li, Tsz Nam Chan, Man Lung Yiu, and Nikos Mamoulis. Fexipro: Fast and exact inner product retrieval in recommender systems. In *SIGMOD*, pages 835–850, 2017.

[LDE$^+$07] Henry Lam, Eric W. Deutsch, James S. Eddes, Jimmy K. Eng, Nichole King, Stephen E. Stein, and Ruedi Aebersold. Development and validation of a spectral library searching method for peptide identification from ms/ms. *Proteomics*, 7(5), 2007.

[LDLL18] Wenhai Li, Lingfeng Deng, Yang Li, and Chen Li. Zigzag: Supporting similarity queries on vector space models. In *SIGMOD*, 2018.

[Lem12] Victor Lempitsky. The inverted multi-index. In *CVPR*, pages 3069–3076, 2012.

[LFV$^+$12] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. The vertica analytic database: C-store 7 years later. *PVLDB*, 5(12):1790–1801, 2012.

[LHG04] Xiaolei Li, Jiawei Han, and Hector Gonzalez. High-dimensional olap: A minimal cubing approach. In *VLDB*, pages 528–539, 2004.

[LKH$^+$08] Eric Lo, Ben Kao, Wai-Shing Ho, Sau Dan Lee, Chun Kit Chui, and David W. Cheung. Olap on sequence data. In *SIGMOD*, pages 649–660, 2008.

[LLW10] Yi-Ching Liaw, Maw-Lin Leou, and Chien-Min Wu. Fast exact k nearest neighbors search using an orthogonal search tree. *PR*, 43(6):2351–2358, 2010.

[LLX$^+$12] Ruixuan Li, Chengzhou Li, Weijun Xiao, Hai Jin, Heng He, Xiwu Gu, Kunmei Wen, and Zhiyong Xu. An efficient SSD-based hybrid storage architecture for large-scale search engines. In *ICPP*, pages 450–459, 2012.

[LMZ05] Nicholas Lester, Alistair Moffat, and Justin Zobel. Fast on-line index construction by geometric partitioning. In *CIKM*, pages 776–783, 2005.

[Low66] Thomas C. Lowe. Design principles for an on-line information retrieval system. Technical report, University of Pennsylvania, 1966.

[LWP16] Chunbin Lin, Jianguo Wang, and Yannis Papakonstantinou. Data compression for analytics over large-scale in-memory column databases (summary paper). *CoRR*, abs/1606.09315, 2016.

[Ma10] Ruyue Ma. Baidu distributed database. In *System Architect Conference China*, 2010.

[ME16] Stephen Mussmann and Stefano Ermon. Learning and inference via maximum inner product search. In *ICML*, pages 2587–2596, 2016.

[Mit01] Michael Mitzenmacher. Compressed bloom filters. In *PODC*, pages 144–150, 2001.

[ML14] Marius Muja and David G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *PAMI*, 36(11):2227–2240, 2014.

[MRS08] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[MZ96] Alistair Moffat and Justin Zobel. Self-indexing inverted files for fast text retrieval. *TOIS*, 14(4):349–379, 1996.

[OV14] Giuseppe Ottaviano and Rossano Venturini. Partitioned elias-fano indexes. In *SIGIR*, pages 273–282, 2014.

[Pug90] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *CACM*, 33:668–676, 1990.

[QWX$^+$18] Jianbin Qin, Yaoshu Wang, Chuan Xiao, Wei Wang, Xuemin Lin, and Yoshiharu Ishikawa. GPH: Similarity search in hamming space. In *ICDE*, 2018.

[RG12] Parikshit Ram and Alexander G. Gray. Maximum inner-product search using cone trees. In *SIGKDD*, pages 931–939, 2012.

[RP71] R. Rice and J. Plaunt. Adaptive variable-length coding for efficient compression of spacecraft television data. *TOCT*, 19(6):889–897, 1971.

[RPK$^+$11] Hongchan Roh, Sanghyun Park, Sungho Kim, Mincheol Shin, and Sang-Won Lee. B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives. *PVLDB*, 5(4), 2011.

[RQH$^+$07] Vijayshankar Raman, Lin Qiao, Wei Han, Inderpal Narang, Ying-Lin Chen, Kou-Horng Yang, and Fen-Ling Ling. Lazy, adaptive rid-list intersection, and its application to index anding. In *SIGMOD*, pages 773–784, 2007.

[RR99] Jun Rao and Kenneth A. Ross. Cache conscious indexing for decision-support in main memory. In *VLDB*, pages 78–89, 1999.

[RR11] Sharadh Ramaswamy and Kenneth Rose. Adaptive cluster distance bounding for high-dimensional indexing. *TKDE*, 23(6):815–830, 2011.

[RU11] Anand Rajaraman and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2011.

[RW94] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–28, 1994.

[SAH08] Chanop Silpa-Anan and Richard Hartley. Optimised kd-trees for fast image descriptor matching. In *CVPR*, pages 1–8, 2008.

[Sam05] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers Inc., 2005.

[SGL09] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. K-ary search on modern processors. In *DaMoN*, pages 52–60, 2009.

[SHWG08] Mehul A. Shah, Stavros Harizopoulos, Janet L. Wiener, and Goetz Graefe. Fast scans and joins using flash drives. In *DaMoN*, pages 17–24, 2008.

[SLZ$^+$15] Fumin Shen, Wei Liu, Shaoting Zhang, Yang Yang, and Heng Tao Shen. Learning binary codes for maximum inner product search. In *ICCV*, pages 4148–4156, 2015.

[ST07] Peter Sanders and Frederik Transier. Intersection in integer inverted indices. In *ALENEX*, pages 71–83, 2007.

[SV10] Fabrizio Silvestri and Rossano Venturini. Vsencoding: Efficient coding and fast decoding of integer lists via dynamic programming. In *CIKM*, pages 1219–1228, 2010.

[SWES08] Dominik Ślęzak, Jakub Wróblewski, Victoria Eastwood, and Piotr Synak. Brighthouse: An analytic data warehouse for ad-hoc queries. *PVLDB*, 1(2):1337–1345, 2008.

[TCJ11] Shirish Tatikonda, B. Barla Cambazoglu, and Flavio P. Junqueira. Posting list intersection on multicore architectures. In *SIGIR*, pages 963–972, 2011.

[TG16] Christina Teflioudi and Rainer Gemulla. Exact and approximate maximum inner product search with lemp. *TODS*, 42(1):5:1–5:49, 2016.

[TGM15] Christina Teflioudi, Rainer Gemulla, and Olga Mykytiuk. Lemp: Fast retrieval of large entries in a matrix product. In *SIGMOD*, pages 107–122, 2015.

[TH72] L.H. Thiel and H.S. Heaps. Program design for retrospective searches on large data bases. *IPM*, 8(1):1 – 20, 1972.

[THS$^+$05] Wilfred H. Tang, Benjamin R. Halpern, Ignat V. Shilov, Sean L. Seymour, Sean P. Keating, Alex Loboda, Alpesh A. Patel, Daniel A. Schaeffer, and Lydia M. Nuwaysir. Discovering known and unanticipated protein modifications using ms/ms database searching. *Analytical Chemistry*, 77(13):3931–3946, 2005.

[THS$^+$09] Dimitris Tsirogiannis, Stavros Harizopoulos, Mehul A. Shah, Janet L. Wiener, and Goetz Graefe. Query processing techniques for solid state drives. In *SIGMOD*, pages 59–72, 2009.

[TWL13] Jiancong Tong, Gang Wang, and Xiaoguang Liu. Latency-aware strategy for static list caching in flash-based web search engines. In *CIKM*, pages 1209–1212, 2013.

[TYSK09] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *SIGMOD*, pages 563–576, 2009.

[VAB$^+$12] Venkateshwaran Venkataramani, Zach Amsden, Nathan Bronson, George Cabrera III, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Jeremy Hoon, Sachin Kulkarni, Nathan Lawrence, Mark Marchukov, Dmitri Petrov, and Lovro Puzar. Tao: How facebook serves the social graph. In *SIGMOD*, pages 791–792, 2012.

[Vig13] Sebastiano Vigna. Quasi-succinct indices. In *WSDM*, pages 83–92, 2013.

[WB13] Mingxun Wang and Nuno Bandeira. Spectral library generating function for assessing spectrum-spectrum match significance. *Journal of Proteome Research*, 12(9):3944–3951, 2013.

[WCB16] Mingxun Wang, Jeremy J. Carver, and Nuno Bandeira. Sharing and community curation of mass spectrometry data with global natural products social molecular networking. *Nature Biotechnology*, 34(8):828–837, 2016.

[WJZ14] Yubao Wu, Ruoming Jin, and Xiang Zhang. Fast and unified local search for random walk based k-nearest-neighbor query in large graphs. In *SIGMOD*, pages 1139–1150, 2014.

[WLH+17] Jianguo Wang, Chunbin Lin, Ruining He, Moojin Chae, Yannis Papakonstantinou, and Steven Swanson. MILC: Inverted list compression in memory. *PVLDB*, 10(8):853–864, 2017.

[WLPS17] Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. An experimental study of bitmap compression vs. inverted list compression. In *SIGMOD*, pages 993–1008, 2017.

[WLY+13] Jianguo Wang, Eric Lo, Man Lung Yiu, Jiancong Tong, Gang Wang, and Xiaoguang Liu. The impact of solid state drive on search engine cache management. In *SIGIR*, pages 693–702, 2013.

[WLY+14] Jianguo Wang, Eric Lo, Man Lung Yiu, Jiancong Tong, Gang Wang, and Xiaoguang Liu. Cache design of ssd-based search engine architectures: An experimental study. *TOIS*, 32(4):1–26, 2014.

[WPK+16] Jianguo Wang, Dongchul Park, Yang-Suk Kee, Yannis Papakonstantinou, and Steven Swanson. Ssd in-storage computing for list intersection. In *DaMoN*, pages 1–7, 2016.

[WPPS16] Jianguo Wang, Dongchul Park, Yannis Papakonstantinou, and Steven Swanson. Ssd in-storage computing for search engines. *TC*, 2016.

[WPSK+10] Jian Wang, Josué Pérez-Santiago, Jonathan E Katz, Parag Mallick, and Nuno Bandeira. Peptide identification from mixture tandem mass spectra. *MCP*, 9(7):1476–1485, 2010.

[WSB98] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, pages 194–205, 1998.

[WSWR18] Yiqiu Wang, Anshumali Shrivastava, Jonathan Wang, and Junghee Ryu. Randomized algorithms accelerated over cpu-gpu for ultra-high dimensional similarity search. In *SIGMOD*, 2018.

[XHC07] Dong Xin, Jiawei Han, and Kevin C. Chang. Progressive and selective merge: Computing top-k with ad-hoc ranking functions. In *SIGMOD*, pages 103–114, 2007.

[XS16] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *FAST*, pages 323–338, 2016.

[YAY14] Albert Yu, Pankaj K. Agarwal, and Jun Yang. Top-k preferences in high dimensions. In *ICDE*, pages 748–759, 2014.

[YDS09] Hao Yan, Shuai Ding, and Torsten Suel. Inverted index compression and query processing with optimized document ordering. In *WWW*, pages 401–410, 2009.

[YMG$^+$98] John R. Yates, Scott F. Morgan, Christine L. Gatlin, Patrick R. Griffin, and Jimmy K. Eng. Method to compare collision-induced dissociation spectra of peptides: Potential for library searching and subtractive analysis. *Analytical Chemistry*, 70(17):3557–3565, 1998.

[ZHC$^+$06] Zhen Zhang, Seung-won Hwang, Kevin Chen-Chuan Chang, Min Wang, Christian A. Lang, and Yuan-chi Chang. Boolean + ranking: Querying a database by k-constrained optimization. In *SIGMOD*, pages 359–370, 2006.

[ZHNB06] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar ram-cpu cache compression. In *ICDE*, 2006.

[ZLS08] Jiangong Zhang, Xiaohui Long, and Torsten Suel. Performance of compressed inverted list caching in search engines. In *WWW*, pages 387–396, 2008.

[ZM06] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *CSUR*, 38:1–56, 2006.

[ZSH16] Shile Zhang, Chao Sun, and Zhenying He. Listmerge: Accelerating top-k aggregation queries over large number of lists. In *DASFAA*, pages 67–81, 2016.

[ZTH$^+$16] Zhaohua Zhang, Jiancong Tong, Haibing Huang, Jin Liang, Tianlong Li, Rebecca J. Stones, Gang Wang, and Xiaoguang Liu. Leveraging context-free grammar for efficient inverted index compression. In *SIGIR*, pages 275–284, 2016.