Effective Search in Distributed Environments

Research Thesis

In Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

Naama Kraus

Submitted to the Senate of the Technion – Israel Institute of Technology

Av 5777 Haifa August 2017

The research thesis was done under the supervision of Prof. Idit Keidar and Dr. David Carmel in the Viterbi Faculty of Electrical Engineering, Technion – Israel Institute of Technology, Haifa, Israel.

The generous financial help of the Technion and the Hasso Plattner Institut is gratefully acknowledged.

This research thesis is dedicated to the memory of my brother in law, Dr. Kobi Kraus, a brilliant and promising researcher, and a person of wisdom and truth.

Acknowledgment

I would like to sincerely thank the many people who made this research possible.

- I was fortunate to have two great advisors: Prof. Idit Keidar and Dr. David Carmel. I wish to thank Idit and David for inspiring me, teaching me how to conduct professional research, giving me the freedom to explore, while providing guidance when needed. I thank you for your support and encouragement at hard times, and the push to always move on. It was my pleasure to work closely with top researchers, and most importantly, with great people. This period of my life will always be with me, and I deeply thank you for making it so significant.
- I wish to thank Prof. Oren Kurland, Prof. Bracha Shapira, and Prof. Yuval Cassuto for their useful feedback on this thesis.
- I would like to thank our research group's team members for the fun times together. I enjoyed our group meetings, the lunch times talks, and the professional discussions. Your technical feedback, advice, and support were always valuable. In particular, I would like to thank Kfir Lev-Ari and Noam Shalev for discussions and encouragements. And most importantly, I thank Sasha Spiegelman, my office-mate and best partner along these years.
- I wish to thank Orly Babad-Tamir and Danit Cohen for their administrative support. Thank you for efficiently assisting in any need that arises, and always doing this with a smile.
- My deep thanks to my parents, Sara and Melech Westreich, for their continuous support and encouragement. Thanks for teaching me to always strive for knowledge and wisdom.
- I wish to thank my parents in law, Leah and Jehoshua Kraus, for the kind support. I am grateful for the assistance, which enabled me to invest time and resources in my research.
- To my beloved children, Mattan, Noa and Yuval, who gave me the light and happiness in a way that only children can.
- I thank my three wonderful brothers, and my extended family for their interest and encouragement.
- Last though most important – there are not enough words out there that can express my gratefulness to Shraga Kraus, my husband and my best friend. Your continuous push, support, and belief in me made all this possible. I am more than fortunate to have you always with me.

List of Publications

1. Naama Kraus, David Carmel, Idit Keidar, "Fishing in the Stream: Similarity Search over Endless Data", *2017 IEEE International Conference on Big Data*.

2. Naama Kraus, David Carmel, Idit Keidar, Meni Orenbach, "NearBucket-LSH: Efficient Similarity Search in P2P Networks", *Similarity Search and Applications - 9th International Conference SISAP 2016*.

3. Naama Kraus, David Carmel, Idit Keidar, "Tail-Tolerant Distributed Search", *Submitted for publication*.

Table of Contents

Abstract **1**

List of Acronyms **3**

List of Symbols **5**

1 Introduction **7**

2 Background **11**

2.1 Exact and Approximate Search 11

2.2 Similarity Search 12

2.3 Locality Sensitive Hashing 12

3 Tail-Tolerant Distributed Search **15**

3.1 Related Work 17

3.2 Model and Problem Definition 18

3.2.1 Distributed Search 18

3.2.2 The Impact of High Tail Latency 20

3.2.3 Search Quality Metrics 20

3.3 Tail-Tolerant DiS 21

3.3.1 Replication 21

3.3.2 Repartition 23

3.4 Analysis 24

3.4.1 Success Probability Formulation 24

3.4.2 Replication 25

3.4.3 Repartition 27

3.5 Empirical Study 28

3.5.1 Methodology 29

3.5.2 Selection Schemes for Replication 30

3.5.3 Replication vs. Repartition 35

3.6 Summary 35

4 Similarity Search over Endless Data **37**
4.1 Similarity Search over Data-Streams 40
4.1.1 SSDS 40
4.2 Stream-LSH 41
4.2.1 Stream-LSH 41
4.2.2 Retention Policies 41
4.2.3 Dynamic Popularity 43
4.3 Analysis 43
4.3.1 Index Size and Number of Retained Copies 43
4.3.2 Success Probability 44
4.4 Empirical Study 51
4.4.1 Methodology 51
4.4.2 Retention Policies 52
4.4.3 Quality-Sensitivity 52
4.4.4 Dynamic Popularity 54
4.5 Related Work 55
4.6 Summary 57

5 NearBucket-LSH: Efficient Similarity Search in P2P Networks **59**
5.1 Model and Problem Definition 61
5.1.1 User Similarity Search in OSN 61
5.1.2 P2P OSN and CAN 61
5.2 Background and Previous Work 62
5.3 Algorithm 62
5.3.1 CAN-based LSH 63
5.3.2 NearBucket-LSH 64
5.4 Analysis 65
5.4.1 Success Probability Formulation 65
5.4.2 Success Probability Comparison 66
5.5 Evaluation 68
5.5.1 Methodology 70
5.5.2 Search Quality Results 71
5.6 Summary 71

6 Conclusion **73**

Bibliography **75**

List of Figures

2.1 Locality Sensitive Hashing (LSH) illustration. 14

3.1 Distributed search (DiS) architecture. 19
3.2 Shard selection methods under Replication. 22
3.3 Random and CRCS SP distributions 31
3.4 Recall as a function of miss probability. 32
3.5 Non-uniform SP distributions. 33
3.6 Recall for three query sets. 34
3.7 Recall as a function of the number of selected shards. 35
3.8 Recall with Replication and Repartition 36

4.1 Threshold vs. Smooth probability of SP. 38
4.2 Expected number of copies of an item. 44
4.3 SP according to similarity and age. 46
4.4 CSP by age radius. 47
4.5 CSP of quality-sensitive Stream-LSH. 48
4.6 DynaPop effect of parameters 50
4.7 SP as a function of expected popularity 50
4.8 Recall comparison of the three retention policies. 53
4.9 Recall of quality-sensitive Smooth. 54
4.10 Recall by popularity radius of DynaPop. 55

5.1 Analytical SP for a constant number of hash functions. 67
5.2 Analytical SP as a function of network cost. 69
5.3 Recall as a function of network cost. 72

List of Tables

3.1 Success probability of different shard selections. 23
3.2 Scores of shard replicas in rSmartRed. 23
3.3 Dataset statistics after pre-processing. 29

4.1 Train and test statistics. 51
4.2 DynaPop item and interest streams statistics. 54

5.1 Summary of costs of similarity search in CAN-based LSH. 68

Abstract

In this thesis, we tackle search effectiveness of distributed search (DiS), and in particular explore tradeoffs between search quality and other considerations. We explore three scenarios: tail-tolerant distributed search, similarity search over endless data-streams, and network-efficient similarity search in peer-to-peer networks. Distributed search engines typically allow missing some of the search results for various reasons, which degrades search quality. For example, in the scenarios that we explored, search quality degrades due to late responses, capacity limitations, and limited network bandwidth. We propose algorithms that improve search quality by better exploiting the infrastructure's resources, namely, index redundancy, space capacity, and network cost. We achieve improvements by considering the internals of the search algorithms in use, rather than using them as black boxes.

We evaluate our algorithms both theoretically and empirically, by formulating a DiS algorithm's *success probability* and measuring its empirical recall. The success probability of a DiS algorithm captures the probability that it finds a query's search result. For measuring recall, we consider centralized search as our ground truth, and measure an algorithm's recall with respect to the search results of a given centralized search algorithm. We measure recall by conducting empirical evaluations using real-world datasets. We compare our algorithms to prior art and show, both theoretically and empirically, that they increase search quality in the three scenarios that we examined.

The topics covered in this thesis are summarized as follows.

Tail-tolerant distributed search. We introduce a novel approach for constructing and searching a distributed index with redundancy when some query results are missed due to high tail latency. We propose *rSmartRed*, an optimal strategy for selecting the number of node replicas to search over at runtime, which considers each node's likelihood to contain results that are relevant to the query, as well the probability to miss its results due to high latency. In addition, when feasible, we propose to replace Replication with *Repartition*, which constructs independent index instances instead of exact copies. Our tail-tolerant distributed search improves search effectiveness when results are omitted due to a high latency compared to naïvely using Replication as a black box.

Similarity search over endless data-streams. We present *Stream-LSH*, a similarity search algorithm that uses a bounded index for indexing unbounded data. We propose a randomized policy for dynamically maintaining items in the index, which takes into account items' age, quality, and popularity attributes. We show that Stream-LSH better exploits capacity resources which improves search effectiveness compared to prior art.

Efficient similarity search in peer-to-peer networks. We present *NearBucket-LSH*, an effective algorithm for similarity search in large-scale distributed online social networks organized as peer-to-peer overlays. As communication is a dominant consideration in distributed systems, we focus on minimizing the network cost while guaranteeing good search quality. We decrease the network cost by considering the internals of the similarity search and the peer-to-peer architecture, and harnessing their properties to our needs.

List of Acronyms

CRCS	central rank based collection selection
DiS	Distributed Search
LSH	Locality Sensitive Hashing
P2P	peer-to-peer
ReDDE	relevant document distribution estimation
SP	success probability
CSI	centralized sample index
CSP	cumulative success probability
TF	term frequency
IDF	inverse document frequency

List of Symbols

A	search algorithm
$\mathcal{D}$	document collection
D_j	shard
d	document
d_q	document relevant to query q
f	miss probability
$\mathcal{G}$	family of LSH hash functions
g	a concatenation of k LSH hash functions
H_i	LSH hash table
h	LSH hash function
I_q	query's result set
$\hat{I}_q$	query's approximate result set
k	LSH concatenation size
L	number of LSH hash tables
p	retention factor
q	query
r	redundancy level
sim	similarity function
t	number of shards to select
u	insertion factor
γ	CRCS result set size
θ	angle between vectors
μ	stream items arrival rate
ρ	interest probability

Chapter 1

Introduction

Search has become a popular feature, ubiquitously used by both end users and applications. Users use commercial search engines on a daily basis, searching for data that satisfies their information needs. Applications, such as recommendation applications [14], use search as a basic primitive inside their algorithms.

In this thesis, we focus on *distributed search (DiS)*, which is the common implementation of today's search in large scale, commercial search engines [21, 30, 27, 76, 40, 51, 94, 53]. DiS distributes the search service among up to thousands of nodes, commonly located within a datacenter. We also consider DiS implementations over a *peer-to-peer network (P2P)* [26, 37, 71, 66], which are fully decentralized and extremely scalable distributed systems.

The holy grail of search algorithms is providing high *search quality* to clients. The main aspect of search is the ability to retrieve search results that are relevant to the query. But modern search systems are interested in data attributes beyond relevancy, such as temporal attributes [55, 31]. These are in particular of interest to applications that search over streams of social and news data [33]. In such contexts, we extend search quality to consider age, quality, and popularity attributes of the data.

Although search quality is of top importance to clients, returning the entire result list of a query is sometimes inefficient, costly, or infeasible. Therefore, DiS systems commonly use *approximate search*, which allows missing some of the search results, at the cost of degrading search quality. For instance, searching the entire dataset may be costly in terms of the network cost [19] or load on the nodes [74], and so DiS often searches only a subset of the nodes, potentially missing some the query's results [30]. Another reason for returning approximate results is missing responses: waiting for responses from slow nodes entails a high search latency, which implies direct loss in revenue [53, 25, 81]. Therefore, DiS systems typically ignore late responses [51, 40] and thus miss some of the query's results. A third example arises when searching over an unbounded data stream, where indexing the entire data is infeasible due to capacity limitations, and thus, only a subset of the documents are indexed and searched [83].

In this thesis, we tackle a number of fundamental tradeoffs between the search system's available resources and the search quality it provides. We consider three domains, and address search quality degradation due to late responses, capacity limitations, and limited network bandwidth. We improve search quality in these scenarios by better utilizing the system's resources.

In Chapter 3, we address tail-tolerant distributed search. Large-scale DiS systems commonly encounter the *high tail latency* phenomenon, in which responses from nodes sometimes take excessively long to arrive. In order to provide a timely response, DiS typically sets strict timeouts and ignores late responses [51, 40], which degrades search quality. A common approach for mitigating response misses is to use *Replication*, which constructs and searches multiple copies of the index partition [21, 1, 8, 53, 94, 51, 88, 40, 48, 42, 30].

We observe that in the context of DiS, searching multiple replicas can be wasteful [88, 53, 94, 51, 40], most notably when response misses are infrequent. We introduce two improvements over prior art: 1) *rSmartRed*, an optimal algorithm for searching over a replicated index, and 2) *Repartition*, a randomized alternative to Replication which constructs non-exact partition copies. Our tail-tolerant DiS decreases the resource waste induced by Replication and thus improves search quality.

In Chapter 4, we address similarity search over endless data streams such as social posts and online news. Here, keeping and searching the entire data is not feasible, as the data is unbounded while capacity resources are bounded [73, 83, 70, 65]. We devise *Stream-LSH*, a time-sensitive similarity search algorithm for data-streams, which bounds the index size in expectation. Stream-LSH considers the age, quality, and dynamic popularity attributes of the data, and better exploits capacity resources in a way that improves search quality.

In Chapter 5, we address efficient similarity search in peer-to-peer networks. Here, communication cost is a dominant factor thus searching the entire P2P network is inefficient [50, 19]. We present *NearBucket-LSH*, a network-efficient similarity search algorithm, which harnesses the P2P network architecture in order to increase search quality for a given communication cost.

We measure search quality of a DiS algorithm by comparing its results to the results of a centralized search, which searches the entire data. In particular, we consider centralized search as our ground truth [74, 64, 42]. We evaluate our algorithms using the following theoretical and empirical search quality metrics [64]:

Success probability analytically formulates the probability that a DiS algorithm finds a particular document that is relevant to a given query. A document is considered relevant to the query, if it appears in the query's top-m results according to the centralized search algorithm.

Recall empirically measures the fraction of documents that a DiS algorithm retrieves from a query's top-m results of the centralized search.

We provide analyses of our algorithms as well as empirical studies using real-world datasets and show that they improve search quality compared to prior art.

Chapter 2

Background

In this chapter, we provide background that is required throughout this thesis. We define *exact search* and *approximate search*, and the search quality metrics that we use. We then overview *similarity search* and *Locality Sensitive Hashing (LSH)*.

2.1 Exact and Approximate Search

The goal of a search system is to retrieve relevant documents to a given query from a given document collection $\mathcal{D}$ [67]. Documents and queries are typically represented by weighted vectors in some high d-dimensional vector space $V = (\mathbb{R}_0^+)^d$. Typically, a search system consists of two basic primitives:

Indexing pre-processes $\mathcal{D}$ and indexes the documents into a persistent data structure called *inverted index*, which we refer to shortly as *index* in this thesis.

Query processing uses the index to retrieve a list of documents ranked according to their estimated relevance to the query.

Given a query q, an *exact search* algorithm A searches the entire document collection $\mathcal{D}$, and returns a ranked list of documents, $I_q \subseteq \mathcal{D}$. In this thesis, we consider centralized search, which is deployed on a single machine and searches the entire document collection, as our ground truth exact search.

Exhaustively searching over the entire document collection is sometimes inefficient, costly, or infeasible. Thus, Given a query q, an *approximate search* algorithm, $\hat{A}$, searches a subset of $\mathcal{D}$, returning an approximation $\hat{I}_q \subseteq \mathcal{D}$ of I_q.

Approximate search induces degradation in search quality, as some of q's results (i.e., documents in I_q) are missing from $\hat{I}_q$. We define a theoretical and an empirical search quality metrics for measuring the quality of approximate search algorithms.

Quality Metrics We theoretically analyze the quality of an approximate search algorithm through its *success probability (SP)*:

Definition 2.1.1 (Success Probability)**.** *Given a query q and a unique document $d_q \in I_q$, $SP(\hat{A}, q)$ is the probability over success and failure that $\hat{A}$ successfully finds d_q.*

We empirically measure the search quality of an approximate search algorithm $\hat{A}$ using the *recall* metric. Recall measures the fraction of search results that $\hat{A}$ retrieves, with respect to the search results of an exact search algorithm A:

Definition 2.1.2 (Recall)**.** *The recall of an approximate search algorithm $\hat{A}$ for query q is defined by*

$$Recall(\hat{A}, q) \triangleq \frac{|\hat{I}_q \cap I_q|}{|I_q|}.$$

Note that $Recall(\hat{A}, q) \in [0, 1]$. We empirically measure the search quality of $\hat{A}$, $Recall(\hat{A})$, by averaging $Recall(\hat{A}, q)$ over all queries q.

2.2 Similarity Search

Given a subset of vectors, $U \subseteq V$, where $V = (\mathbb{R}_0^+)^d$ is some high d-dimensional vector space, similarity search is the task of finding vectors in U that are similar to some query vector $q \in V$ [35].

Similarity search is based on a *similarity function*, which measures the similarity between two vectors, $u, v \in V$ [35]:

Definition 2.2.1 (similarity function)**.** *A similarity function $sim : V \times V \to [0, 1]$ is a function such that $\forall u, v \in V, sim(u, v) = sim(v, u)$ and $sim(v, v) = 1$.*

The similarity function returns a *similarity value* within the range $[0, 1]$, where 1 denotes perfect similarity, and 0 denotes no similarity. We say that u is *s-similar* to v if $sim(u, v) = s$.

A commonly used similarity function for textual data is *angular similarity* [83, 73], which is closely related to cosine similarity [32, 35]. The angular similarity between two vectors $u, v \in V$ is defined as:

$$sim(u, v) = 1 - \frac{\theta(u, v)}{\pi}, \tag{2.1}$$

where $\theta(u, v) = arccos(\frac{u \cdot v}{\|u\| \cdot \|v\|})$ is the angle between u and v.

2.3 Locality Sensitive Hashing

Locality Sensitive Hashing (LSH) [52, 47] is a widely used approximate similarity search algorithm for high-dimensional spaces, with sub-linear search time complexity. LSH limits the search to vectors that are likely to be similar to the query vector instead of linearly searching over all the vectors. This reduces the search time complexity at the cost of missing similar vectors with some probability.

LSH uses hash functions that map a vector in the high dimensional input space $(\mathbb{R}_0^+)^d$ into a representation in a lower dimension $k << d$, so that the hashes of similar vectors are likely to collide. LSH executes a pre-processing (index building) stage, where it assigns vectors into buckets according to their hash values. Then, given a query vector, the similarity search algorithm computes its hashes and searches vectors in the corresponding buckets. The LSH algorithm is parametrized by k and L, where k is the hashed domain's dimension, and L is the number of hash functions used, as explained below. Formally [32]: a *locality sensitive hashing* with similarity function *sim* is a distribution on a family $\mathcal{H}$ of hash functions on a collection of vectors, $h : V \rightarrow \{0, 1\}$, such that for two vectors u, v,

$$Pr_{h \in \mathcal{H}} \left[h(u) = h(v) \right] = sim(u, v). \tag{2.2}$$

In order to increase the probability that similar vectors are mapped to the same bucket, the algorithm defines a family $\mathcal{G}$ of hash functions, where each $g(v) \in \mathcal{G}$ is a concatenation of k functions chosen randomly and independently from $\mathcal{H}$. In the case of angular similarity, $g : V \rightarrow \{0, 1\}^k$, i.e., g hashes v into a binary *sketch vector*, which encodes v in a lower dimension k. For two vectors u, v,

$$Pr_{g \in \mathcal{G}} \left[g(u) = g(v) \right] = (sim(u, v))^k, \tag{2.3}$$

for any randomly selected $g \in \mathcal{G}$. The larger k is, the higher the precision.

Figure 2.1 illustrates LSH.

In order to mitigate the probability to miss similar items, the LSH algorithm selects L functions randomly and independently from $\mathcal{G}$. The item vectors are now replicated in L hash tables H_i, $1 \leq i \leq L$. Upon query, search is performed in L buckets. This increases the recall at the cost of additional storage and processing.

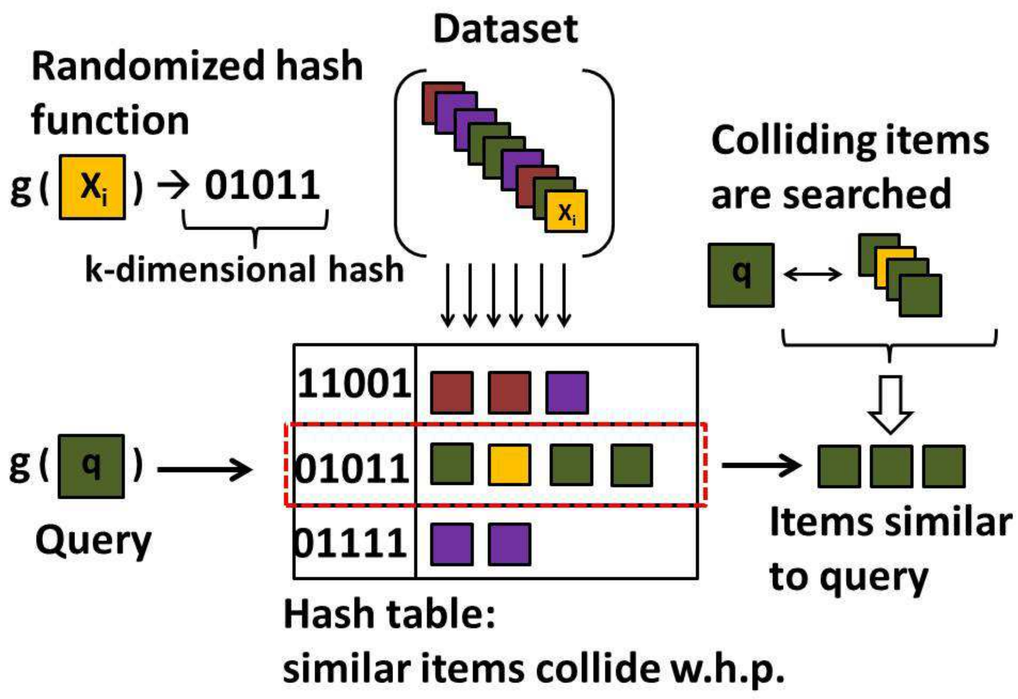


Figure 2.1: Locality Sensitive Hashing (LSH) illustration.

Chapter 3

Tail-Tolerant Distributed Search

Commercial search engines serve tens of thousands of online queries a second, covering corpora with billions of documents. In order to scale with the massive data, a search service is typically deployed on a cluster of nodes, which jointly implement *distributed search (DiS)* [29, 21, 30, 27, 76, 53, 94, 51, 40]. A common DiS approach is to partition the documents into subsets called *shards*, where each shard is assigned to a node in which it is locally indexed and searched [21, 30, 27, 40]. In order to reduce the computational cost, it is common to employ *approximate search* [30, 74, 42, 29, 53, 94, 51, 40], whereby queries are sent to only a subset of the shards that are deemed most likely to satisfy the query.

Search providers nowadays aim to deliver results to the client within a few hundreds of milliseconds, expecting processing times of tens of milliseconds from the back-end search service [2, 5, 51, 40]. Unfortunately, responses from nodes may sometimes take excessively long to arrive; this occurs due to many reasons, including network or server load, background processing, misconfiguration, crashes, etc. This phenomenon is called the *tail latency* problem – the problem that tail (high percentile) latencies are much larger than the average latency [53, 94, 51, 40]. Given that an increase in search response time directly implies loss in revenue [53, 25, 81], search engines typically set strict timeouts and ignore late responses [51, 40]. Consequently, some of the relevant results are missed, entailing degradation of search quality [53, 51, 40, 42].

Because in large data centers high tail latencies are the norm [53, 40, 94, 51, 88], commercial search engines often deploy shards over a replicated storage layer, which, to guarantee timely responses, directs queries to all replicas of the requested shard [21, 1, 8, 53, 88, 40, 48, 42, 30]. Although Replication is the standard approach to building fault-tolerant services [18], we observe that in the context of search it is not ideal. This is because the challenges in the two cases are different: whereas classical fault-tolerance is concerned with services that are either available or unavailable, in search, the *quality* of the results is of essence. In this context, accessing multiple replicas of the same shard can be wasteful [88, 53, 40], most notably when result misses are infrequent; a better use of resources could be accessing additional shards instead of additional copies of the same shard.

In this chapter we consider tail-tolerance [40] as a first class citizen in distributed search design. We study the problem of *tail-tolerant distributed search*, whose goal is to maximize search quality when responses are missed due to high tail latencies. We suggest two improvements over the standard approach. First, we present *rSmartRed*, an optimal selection scheme for replicated distributed search indexes. Given a query, rSmartRed considers each shard's probability to satisfy the query, as well as the probability to miss the shard's results, in determining the number of replicas to select per shard. Second, when feasible, we propose to employ *Repartition*, an alternative to Replication that reduces the waste due to searching redundant shards. Repartition randomly constructs independent partitions of the index instead of exact copies.

Following the seminal work of Lv et al. [64], we use *success probability* and *recall* metrics for measuring search quality. We analyze the success probability to find a document relevant to a given query using different DiS algorithms; we prove that rSmartRed's selection scheme is optimal for Replication, and that Repartition improves over Replication for any selection scheme. We confirm our analysis by conducting an empirical study using the Reuters RCV1 and Livejournal real-world datasets. Our experiments show that rSmartRed achieves higher recall than techniques used today. We further show the superiority of Repartition over Replication when excessive latencies are infrequent.

3.1 Related Work

Fault-tolerant distributed systems have been extensively explored in the literature and standard textbooks, e.g., [18]. This line of work typically considers a binary availability model, where the service is either available or unavailable, whereas distributed search's availability can be captured by a finer-grain notion of search quality.

A wealth of prior art explores distributed search (refer to [29, 30] for a comprehensive overview), where the index is partitioned into shards and distributed among multiple nodes. At runtime, a *broker* handling the query distributes it to nodes, awaits their responses, aggregates the responses, and sends the query result to the user. The broker may employ search over all shards, but to avoid excessive computation, it is more common to use approximate search [29, 30, 57, 47, 53], which selects a subset of the shards to search over. The selection of shards to search over is either random [30, 53], or based on the estimated likelihood of the shards to contain results that are relevant to the query [29, 47].

Most existing academic work on distributed search does not consider tail-tolerance. Nevertheless, high tail latency is a serious problem in practice, and so industrial solutions must take it into consideration [40, 1, 8, 53, 94, 51]. The ubiquitous approach to dealing with high tail latencies is truncating the tail, namely, responding to the user without waiting for responses from all nodes [40, 53, 94, 51, 88]. The rationale behind this approach is that "returning good results quickly is better than returning the best results slowly" [40]. To compensate for the omitted responses, it is common to use redundancy in the form of Replication [21, 1, 8, 53, 88, 40, 48, 42, 30]. Note that commercial search engines [40, 53, 94, 51] apply engineering decisions (typically architecture-specific) and other optimizations to reduce the tail latency of the search workflow; such strategies are orthogonal to our research, and are not sufficient by themselves [40, 88]. Replication complements these optimizations, and is the focus of this work.

Prior art observed that Replication incurs resource waste due to duplicate search operations [88, 53, 40]. Commercial search engines [40, 53] decrease this waste by combining two techniques. First, they only re-issue a search request for slow shards, and second, they cancel ongoing duplicate requests upon learning that they are not likely to contribute much to the search quality. Although these strategies were shown to be useful to some extent, the approach of simultaneously sending multiple copies of each request is still commonly used despite the waste it incurs [88, 40]. In this paper, we tackle Replication's inherent waste using two improvements: First, we propose a simple optimal algorithm for adjusting the replication level for each shard when processing a given query. Second, we propose an alternative approach to redundancy, which further decreases waste.

3.2 Model and Problem Definition

In this section, we present the *distributed search (DiS)* model that we consider. We then present the problem of search quality degradation due to high tail latency in DiS, and the search quality metrics that we use.

3.2.1 Distributed Search

Centralized search, deployed on a single machine, does not scale with the size of the data [21, 30], and so is not used in practice for searching large data collections. We consider a DiS algorithm which approximates a given centralized search algorithm, as we further detail. Distributed search scales the search service by distributing indexing and query processing over multiple nodes [29, 21, 30, 27, 76, 53, 94, 51, 40]. We consider the common approximate search approach for distributed query processing [30, 74, 42, 29, 53, 94, 51, 40], which in order to reduce computational costs, submits each query to only a selected subset of the nodes. Note that in order to avoid missing important (e.g., popular) results, search engines commonly dedicate certain nodes to storing important documents, which are always searched [30, 27]; we consider here an approximate search over the rest of the nodes.

We assume a centralized search algorithm is given as a black box. We assume a cluster of $n > 1$ nodes connected by a fast network. We consider the common *document-based* approach to DiS, whereby each node holds and performs search on some subset of $\mathcal{D}$ [21, 30, 27, 40]. To implement DiS, one needs to address two aspects: partitioning and shard selection. Figure 3.1 illustrates DiS architecture, which we further detail in the next paragraphs.

Partitioning At the indexing stage, DiS applies a *partitioning scheme* to partition $\mathcal{D}$ into a set of n pairwise disjoint *shards* $D_j \subset \mathcal{D}$, $\mathcal{D} = \bigcup_{j=1}^{n} D_j$. Each shard D_j is assigned to a separate node where it is locally indexed. One common approach is similarity-based partitioning [30, 29, 74, 57], e.g., LSH [52, 47], which constructs shards of similar documents. LSH randomly selects a hash function that maps each document $d \in \mathcal{D}$ into its corresponding shard, where the probability that two documents are mapped to the same shard grows with their similarity [32, 35]. Note that LSH is a natural choice for distributed partitioning, since parallelizing it is straightforward because hashing a document does not require any information about other documents.

Shard selection A broker module handles clients' search requests [30]. At runtime, the broker accepts an input query and submits the query to the nodes, each of which locally searches its shard using the given centralized search algorithm, and returns to the broker a ranked list of results that it finds most relevant to the query. Note that all shards apply the same ranking function hence their ranks are comparable. The broker collects and

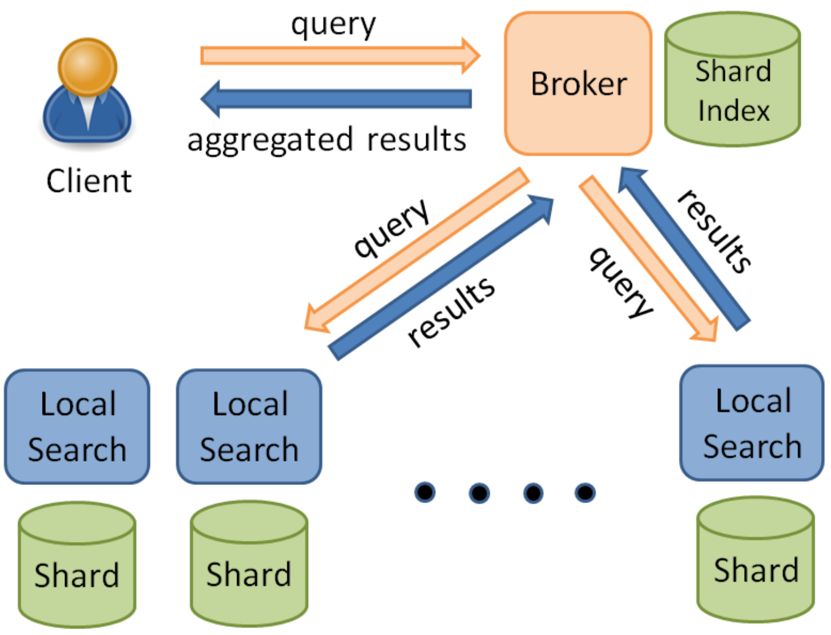


Figure 3.1: Illustration of distributed search (DiS) architecture supporting approximate search. At indexing time, DiS partitions the document collection into shards, which are distributed among the nodes. At runtime, a broker module serves clients' queries: for each query, the broker first selects a subset of the shards to search over. Each node searches its shard locally and returns its search results to the broker. Finally, the broker aggregates the nodes' results and returns them to the client.

merges the shards' results and returns a final result list to the caller, which approximates the results of the centralized search.

The broker uses a *shard selection scheme* in order to select a subset of $t \leq n$ shards to send the query to. Typically, the selection scheme uses a *shard index*, which holds the partition's meta-data. More specifically, it maps shard identifiers to the nodes where they reside, and optionally maintains some compact representation of each shard's content. The shard index is constructed during the indexing stage, it is typically centralized and replicated for availability.

Given a query, the shard selection scheme approximates a probability distribution over the shards, associating with each shard D_j the estimated probability that it contains a relevant document to the query; the latter is called the shard's *success probability* for the query. A document is considered relevant to the query, if it appears in the query's top-m results according to the centralized search algorithm [74, 64, 42].

Shard selection may use the simple *Random* [30, 53] approach, which does not employ a shard representation, and randomly selects shards independently of the input query. Note that Random induces a uniform success probability distribution over the shards. A more effective approach is to select shards that are deemed most likely to satisfy the query (refer to [30] for an overview of selection methods). One popular method is ReDDE [79], which represents a shard using a random sample of its content. At the indexing stage, ReDDE randomly samples documents from each shard and indexes them into the shard index[1]. At shard selection time, given a query q, ReDDE retrieves from the shard index a set of documents that are most relevant to q, and based on them, selects t shards that are most likely to contain documents relevant to q. In our experiments, we use CRCS Linear [78] to approximate the success probability distribution over the shards, which we detail in Section 3.5.

3.2.2 The Impact of High Tail Latency

We now extend DiS to consider high tail latency. In order to provide search results in a timely manner (typically a search latency of few hundreds of milliseconds [2, 5, 40, 51]), the broker waits for responses from nodes up to a fixed timeout that is given to DiS as a parameter [51, 40]. The broker collects results from the nodes that respond on time, and drops the results of the slow nodes [53, 51, 40]. A node may fail to return its results with the desired latency due to various reasons: E.g., it may be temporarily down due to hardware or software problems, be overloaded by other queries, or lose messages due to network failures or loads [40]. We assume that each node fails to respond on time with some *miss probability*. For simplicity, we assume that each node fails to respond independently of other shards, and that the miss probability is common to all nodes. We denote the miss probability by f. When a node's response is skipped, some of the relevant results may be missing from the final result set, which entails degradation of search quality [53, 51, 40, 42]. Note that the search quality of DiS with approximate search is typically lower than that of centralized search even without misses, as the search is restricted to a subset of the collection. Result misses due to high tail latencies further degrade search quality; our goal is to ameliorate this.

3.2.3 Search Quality Metrics

We analyze the quality of a DiS algorithm A through its *success probability*: Given a query q and a document $d_q \in \mathcal{D}$ relevant to q, $SP(A, q)$ is the probability that A finds d_q.

We empirically measure the search quality of a DiS algorithm A by comparing its results with those of a centralized search, which has full access to all documents [74, 64, 42], and uses the same ranking function as A. More specifically, let $S_C^m(q)$ be the top-m search results for query q according to the centralized search, and let $S_A^m(q)$ be the top-m results of a DiS algorithm A. We measure the search quality of A for query q by the *recall* it achieves relative to a centralized system:

$$Recall@m(q) \triangleq \frac{|S_C^m(q) \cap S_A^m(q)|}{|S_C^m(q)|}.$$

[1] In ReDDE, the shard index is commonly called a centralized sample index (CSI).

Note that $Recall@m(q) \in [0, 1]$. We measure the search quality of a DiS algorithm A, $Recall@m$, by averaging $Recall@m(q)$ over all queries.

3.3 Tail-Tolerant DiS

Existing DiS systems mitigate search quality degradation using Replication [21, 1, 8, 53, 88, 40, 48, 42, 30]. We propose two improvements to currently used approaches: rSmartRed, an optimal shard selection scheme for Replication, and Repartition, an alternative method to redundancy which improves over Replication.

3.3.1 Replication

Given a redundancy level configuration parameter $r > 1$ and a partition $\{D_1, \ldots, D_n\}$, Replication constructs r identical copies of that partition. Large-scale search systems usually deploy their service over multiple data centers and use dozens of replicas [21], whereas smaller-scale search systems that run in a single data center typically use a few copies per shard, e.g., $r = 3$ [1, 8]. A shard selection scheme that uses Replication needs to take an additional aspect into account: Besides identifying the shards most likely to satisfy the query, it needs to also decide how many replicas of each shard to contact. We discuss three approaches for doing so. We assume that all approaches are given a fixed budget of tr shards to select out of all nr shards.

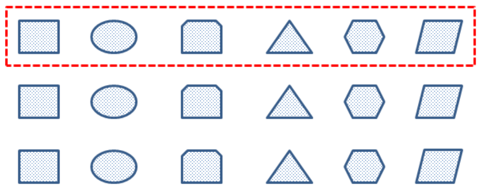
Existing shard selection approaches

Two main approaches are used for shard selection today. First, in some cases, redundancy is used only for load-balancing and not for mitigating result misses [30, 27], yielding an approach we call "no redundancy", denoted *NoRed*. NoRed selects all tr shards from a single partition without replicas ($tr \leq n$), and the broker directs user queries to different index partitions. Figure 3.2(a) illustrates NoRed.

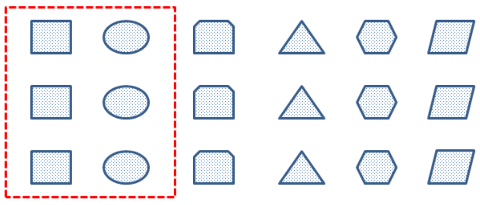
In other cases, a "full redundancy", denoted *rFullRed*, approach is used [48, 42, 30, 21, 54, 1, 8, 40, 53, 88]. Given a query, the broker selects t out of n shards of the original partition, and replicates its selection by contacting all r replicas of each selected shard. Figure 3.2(b) illustrates rFullRed. This approach arises when shard selection and redundancy are two separate abstraction layers, and the search algorithm uses replicated storage as a black box.

Optimal shard selection

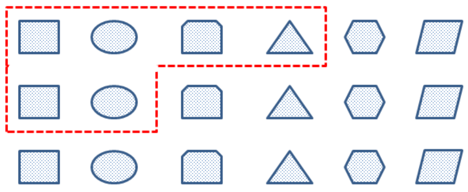
We next open up the black box and integrate shard selection and redundancy. We present an optimal approach for Replication that we call rSmartRed. Given a replicated index and a particular shard selection scheme over a single partition, rSmartRed maximizes the probability to find relevant documents with respect to the given shard selection scheme.



(a) NoRed



(b) rFullRed



(c) rSmartRed

Figure 3.2: Illustration of the three shard selection methods under Replication. A row represents a partition of the collection into 6 shards. At runtime, a total of 6 shards (dashed lines) are selected. NoRed selects shards from a single partition without replicas; rFullRed contacts all 3 replicas of each shard it selects; rSmartRed adjusts the number of selected shard replicas such that search quality is maximized.

Our method considers both the miss probability and the success probability distribution (induced by the shard selection scheme) when selecting shard replicas.

To give an intuition why existing approaches are not optimal, consider the following example: the dataset is partitioned into 5 shards, each shard has two replicas ($r = 2$), and the broker selects $tr = 2$ shards per query. For some query q, D_1's success probability is 0.8, and D_2's success probability is 0.1. The success probability of the rest of the shards is smaller. Clearly, D_1 should be selected at least once. There are two alternatives for selecting the second shard: D_1's replica or D_2. If D_1's two replicas are selected, a relevant document d_q is found if it is stored in D_1, and at least one of D_1's replicas does not fail to respond, which happens with probability $0.8(1 - f^2)$. If D_1 and D_2 are selected, d_q is found if it is stored in either D_1 or D_2, and the shard that contains it does not fail to respond. As D_1 and D_2 are disjoint, this happens with probability $(0.8 + 0.1)(1 - f)$. Table 3.1 depicts the success probabilities of the two selection alternatives for two values of f. As the table demonstrates, the selection that maximizes the success probability depends on the value of f. For $f = 0.05$, selecting D_1 and D_2 is preferable, whereas for $f = 0.2$, selecting the two replicas of D_1 is preferable.

	Two replicas of D_1	D_1 and D_2
$f = 0.05$	0.8	0.85
$f = 0.2$	0.77	0.72

Table 3.1: Success probability of different shard selections (columns) for different miss probabilities (rows) when selecting a total of $tr = 2$ shards under Replication. When $f = 0.05$, it is preferable to select the top two shards from the same partition, whereas when $f = 0.2$, it is preferable to select two replicas of the highest ranked shard.

Our rSmartRed algorithm considers f and an estimated distribution of the shard success probabilities. Given a query q, we denote by $p_q(j)$ the estimated success probability of shard D_j. Given r replicas of the partition, we assign a score of $f^{i-1}p_q(j)$ to the ith replica of shard D_j as depicted in Table 3.2. We then select the tr shard replicas with the highest scores. Figure 3.2(c) shows an example selection of rSmartRed. In Section 3.4, we prove that if the estimations are accurate, then rSmartRed maximizes the probability to find relevant documents with any given number of selections. Note that rSmartRed is more likely to select multiple replicas of a shard as the shard's success probability increases and as f increases.

	D_1	$\dots$	D_n
Replica 1	$p_q(1)$	$\dots$	$p_q(n)$
Replica 2	$p_q(1)f$	$\dots$	$p_q(n)f$
$\vdots$	$\vdots$		$\vdots$
Replica r	$p_q(1)f^{r-1}$		$p_q(n)f^{r-1}$

Table 3.2: Scores of shard replicas in rSmartRed. rSmartRed selects the tr shard replicas with the highest scores.

The following observation follows immediately from the algorithm:

Observation 1. *For each i, the t_i shards that rSmartRed selects from partition i are the t_i top-scored shards in that partition according to success probability.*

3.3.2 Repartition

We propose Repartition, a new approach for constructing a redundant index for tail-tolerant DiS. Like Replication, Repartition constructs r partitions of $\mathcal{D}$, each consisting of n pairwise disjoint shards, and each of the nr shards is assigned to a separate node. However, unlike Replication, Repartition does not replicate the partition into r partition copies. Instead, Repartition constructs r independent, non-identical partitions of $\mathcal{D}$. Repartition uses a randomized partitioning scheme and applies it r times independently to construct such r partitions. One may use LSH [52, 47] for implementing Repartition by randomly and independently selecting r hash functions. We propose two shard selection schemes for Repartition: *pTop* and *pSmartRed*. Note that NoRed is trivially applicable to Repartition as well.

pTop As partitions are independent, a natural shard selection scheme for Replication selects the t top-scored shards from each partition independently, where a shard's score is its estimated success probability. We call this selection scheme pTop. Note that like rFullRed, pTop selects the same number of shards (t) from each partition.

pSmartRed Our second selection, pSmartRed, imitates rSmartRed, and works as follows: pSmartRed first arbitrarily selects one of the partitions of $\mathcal{D}$, and computes rSmartRed's shard selection over r replicas of $\mathcal{D}$. Recall that rSmartRed selects t_i shards from each partition replica i. pSmartRed then selects the t_i top-scored shards from each partition i of the re-partitioned index according to the success probability distribution of the shards in that partition. Therefore, pSmartRed preserves the number of shards that rSmartRed selects from each partition. For example, pSmartRed applies rSmartRed's selection that is illustrated in Figure 3.2(c), by selecting the four top-scored shards from one partition and the two top-scored shards from the second.

As we show both analytically and in our empirical study, Repartition improves over Replication. On the other hand, creating and maintaining the index are more costly with Repartition. Another limitation of Repartition is that it is not applicable when the partitioning is given by a third party and cannot be altered.

3.4 Analysis

In this section, we analytically study the success probability [64] to retrieve a document relevant to a query when searching a tail-tolerant distributed index. We provide closed-form analysis of the success probability under Replication, and prove that rSmartRed is the optimal selection. We also prove that for any shard selection scheme for Replication, there exists a shard selection scheme for Repartition with a larger or equal success probability.

3.4.1 Success Probability Formulation

Consider a tail-tolerant DiS algorithm $A(r,t)$ with redundancy r and tr shards selected per query. Consider a query q. Although multiple documents may be relevant to q, for the sake of the analysis, we consider exactly one document $d_q \in \mathcal{D}$ that is relevant to q. For query q, the shard selection scheme induces a probability distribution $p_q : \{1, \ldots, n\} \to [0,1]$, where $p_q(j)$ is the probability that d_q is stored in shard D_j (in practice, shard selection schemes such as ReDDE approximate this distribution). Since $\{D_1, \ldots, D_n\}$ is a partition, $\sum_{j=1}^{n} p_q(j) = 1$. We denote by $SP(q, f, A(r,t))$ the probability that $A(r,t)$ finds d_q when processing query q under miss probability f. SP is called $A(r,t)$'s success probability. Henceforth we fix a query q and remove q from our notations.

3.4.2 Replication

Consider a replicated DiS algorithm $A_R(r,t)$. We denote by $S_i \subseteq \{D_1, \ldots, D_n\}$ the set of shards for which $A_R(r,t)$ selects at least $i \geq 1$ replicas. For example, if three replicas of D_7 are selected, then $D_7 \in S_1, S_2, S_3$. Note that $\sum_{i=1}^{r} |S_i| = tr$. In addition,

$$S_r \subseteq S_{r-1} \subseteq \ldots \subset S_1. \tag{3.1}$$

We denote by $SP(f, S_i)$ the probability that d_q is found when accessing the shards in S_i. This occurs if d_q is stored in one of the shards in S_i and that shard does not fail to respond. Since shards in S_i are disjoint:

$$SP(f, S_i) = (1-f) \sum_{D_j \in S_i} p(j). \tag{3.2}$$

We denote by $SP(f, S_i^j)$ the probability that d_q is found when accessing the shards in $\{S_i, \ldots, S_j\}$. By definition, A_R's success probability equals $SP(f, S_1^r)$. The following lemma formulates $SP(f, S_1^r)$:

Lemma 1.

$$SP(f, S_1^r) = (1-f) \left(\sum_{D_j \in S_1} p(j) + \ldots \sum_{D_j \in S_r} f^{r-1} p(j) \right).$$

Proof. We prove by induction on r.

Base ($r = 1$): Follows directly from Equation 3.2.

Step: we assume for $r-1$:

$$SP(f, S_1^{r-1}) = (1-f) \left(\sum_{D_j \in S_1} p(j) + \ldots + \sum_{D_j \in S_{r-1}} f^{r-2} p(j) \right). \tag{3.3}$$

As searching the shards in $\{S_1, \ldots, S_r\}$ is the union of the events of 1) searching the shards in $\{S_1, \ldots, S_{r-1}\}$ and 2) searching the shards in S_r, then according to the probability of a union of events[2],

$$SP(f,S_1^r) = SP(f,S_1^{r-1}) + SP(f,S_r) - SP(f,S_r)SP((f,S_1^{r-1})|(f,S_r)), \tag{3.4}$$

where $SP((f,S_1^{r-1}|(f,S_r))$ denotes the conditional probability to find d_q when searching the shards in $\{S_1,\dots,S_{r-1}\}$, given that d_q is found when searching the shards in S_r. Let $D_j \in S_r$ be the shard that contains d_q. Due to containment (Equation (3.1)), $D_j \in S_i$, $1 \le i \le r-1$. Hence, d_q is found if at least one of those $r-1$ shards does not fail to respond, which happens with probability $1-f^{r-1}$. Thus,

$$SP((f,S_1^{r-1})|(f,S_r)) = 1 - f^{r-1}. \tag{3.5}$$

By Equations 3.4 and 3.5:

$$SP(f,S_1^r) = SP(f,S_1^{r-1}) + SP(f,S_r) - SP(f,S_r)(1-f^{r-1}). \tag{3.6}$$

And by Equations 3.2, 3.3:

$$\begin{aligned} SP(f,S_1^r) &= (1-f)(\sum_{D_j\in S_1} p(j) + \dots + \sum_{D_j\in S_{r-1}} f^{r-2}p(j)) \\ &+(1-f)\sum_{D_j\in S_r} p(j) - (1-f^{r-1})(1-f)\sum_{D_j\in S_r} p(j) \\ &= (1-f)\left(\sum_{D_j\in S_1} p(j) + \dots + \sum_{D_j\in S_r} f^{r-1}p(j)\right). \end{aligned}$$

□

Optimal selection

Theorem 1. *For a given $1 \le tr \le nr$, rSmartRed selects tr shards such that SP is maximized.*

Proof. According to Lemma 1, SP is maximized when $\sum_{D_j\in S_1} p(j) + \dots + \sum_{D_j\in S_r} f^{r-1}p(j)$ is maximized. Selecting the tr shards with the largest score values, $f^{i-1}p(j)$, maximizes the sum hence the success probability. □

rSmartRed's selection depends on the miss probability and the shard success probability distribution as formulated in Lemma 1. When f is high and the distribution is skewed, i.e., few shards have a high success probability, the optimal selection is likely to select those shards' replicas and tends towards the rFullRed method. When the success

[2] $Pr(A\cup B) = Pr(A) + Pr(B) - Pr(B)Pr(A|B)$.

probability distribution is close to uniform, or when f is low, the optimal selection is more likely to select additional shards of the partition, hence it tends towards the NoRed method. Note that $f^{i-1}p_j$ exponentially decreases as i increases, hence the effectiveness of selecting additional replicas of a shard decreases as more of its replicas are selected.

3.4.3 Repartition

Theorem 2. *Consider a Repartition in which all partitions have the same probability distribution for a given query q. Then, for every shard selection used with Replication of one of these partitions, there exists a shard selection for Repartition with a larger or equal success probability for q.*

Proof. Consider a Repartition consisting of r independent partitions of $\mathcal{D}$, and a Replication algorithm $A_R(r,t)$ employing r replicas of one of these partitions. Consider $A_R(r,t)$'s shard selection, $S_1,\ldots,S_r$, which selects $t_i \triangleq |S_i|$ top-scored shards from each partition replica i. We construct a Repartition algorithm $A_P(r,t)$ to select the t_i top-scored shards for each independent partition i, according to success probability. I.e., $A_P(r,t)$ preserves the number of shards that $A_R(r,t)$ selects per each partition. We denote $A_P(r,t)$'s selection by $S'_1,\ldots,S'_r$. We prove that

$$SP_P(f,\bigcup_{i=1}^{r} S'_i) \geq SP_R(f,\bigcup_{i=1}^{r} S_i). \tag{3.7}$$

by induction on r.

Base ($r=1$): According to Equation 3.2, and since the distributions are equal for all partitions: $SP_P(f,S'_1) = SP_R(f,S_1)$.

Step: we assume for $r-1$:

$$SP_P(f,\bigcup_{i=1}^{r-1} S'_i) \geq SP_R(f,\bigcup_{i=1}^{r-1} S_i). \tag{3.8}$$

We compute for r: as $\bigcup_{i=1}^{r} S'_i = (\bigcup_{i=1}^{r-1} S'_i) \cup S'_r$, and as the partitions are independent, then according to the probability of a union of independent events:

$$\begin{aligned} SP_P(f,\bigcup_{i=1}^{r} S'_i) &= SP_P(f,\bigcup_{i=1}^{r-1} S'_i) + SP_P(f,S'_r) \\ &- SP_P(f,S'_r)SP_P(f,\bigcup_{i=1}^{r-1} S'_i). \end{aligned} \tag{3.9}$$

$SP_P(f,\bigcup_{i=1}^{r-1} S'_i)$ denotes the probability that we found d_q when searching $\bigcup_{i=1}^{r-1} S'_i$. Denote by η the probability that d_q is stored in $\bigcup_{i=1}^{r-1} S'_i$. For each S'_i, $1 \leq i \leq r-1$, d_q is either stored in one of the shards in S'_i or it is not. Hence, d_q is stored in u shards in $\bigcup_{i=1}^{r-1} S'_i$, where $0 \leq u \leq r-1$, and is found if at least one of those shards does not fail to respond which happens with probability $1 - f^u$. Thus,

$$SP_P(f, \bigcup_{i=1}^{r-1} S'_i) = \eta(1 - f^u). \tag{3.10}$$

Since both $A_R(r,t)$ and $A_P(r,t)$ select t_r top-scored shards from their r-th partition, and since success probabilities are equal for all partitions, then according to Equation 3.2,

$$SP_P(f, S'_r) = SP_R(f, S_r). \tag{3.11}$$

Substituting Equations 3.10 and 3.11 in Equation 3.9 we get:

$$SP_P(f, \bigcup_{i=1}^{r} S'_i) = SP_P(f, \bigcup_{i=1}^{r-1} S'_i) + SP_R(f, S_r)$$
$$-SP_R(f, S_r)\eta(1 - f^u).$$

According to the induction assumption in Equation 3.8, and since $0 \leq \eta \leq 1$ and $1 - f^u \leq 1 - f^r$, it follows that

$$SP_P(f, \bigcup_{i=1}^{r} S'_i) \geq SP_R(f, \bigcup_{i=1}^{r-1} S_i) + SP_R(f, S_r)$$
$$- SP_R(f, S_r)(1 - f^r) \overset{(3.6)}{=} SP_R(f, \bigcup_{i=1}^{r} S_i). \tag{3.12}$$

□

Note that pSmartRed preserves the number of shards that rSmartRed selects per each partition, and thus according to Equation 3.7, pSmartRed's success probability is equal or greater than rSmartRed's. Nevertheless, although rSmartRed is optimal for Replication, this does not imply that pSmartRed is optimal for Repartition. Note further that Theorem 2 holds under the assumption that all partitions' probability distributions are the same. In practice this assumption does not necessarily hold, but our experiments show that Repartition is advantageous nevertheless.

3.5 Empirical Study

We empirically evaluate our tail-tolerant distributed search using two real-world datasets. We measure search quality using the recall metric and demonstrate the superiority of rSmartRed over NoRed and rFullRed, and the improvement that Repartition suggests over Replication. Our empirical study confirms our analysis.

3.5.1 Methodology

Datasets

- Reuters RCV1 [6] consists of news articles spanning a year. We represent each news article by its title and first paragraph. We parse the Reuters datasets using conventional methods, including stop-word removal and stemming [67].
- Livej [92] was crawled from the Livejournal [3] free online community by the Stanford SNAP Project [9]. In Livejournal, the corpus consists of users who join blogs that reflect their topics of interest. We represent a user by a document where the document's terms correspond to the blogs he joined, filtering out users with no topics of interest.

Table 3.3 summarizes the dataset statistics after pre-processing.

	Number of Documents	Number of Terms
Reuters	779,913	96,513
Livej	1,147,948	664,414

Table 3.3: Dataset statistics after pre-processing.

Experiment setup We use Lucene [4] 4.3.0 search library as our indexing and retrieval infrastructure. We weight document terms according to Lucene's TF-IDF function, where $TF(term)$ is the square root of the term's frequency in the document, and $IDF(term) = ln(\frac{N_d}{N_{term}+1}) + 1$, where N_d is the total number of documents, and N_{term} is the number of items containing the term. We score documents with Lucene's default similarity function, which implements a variant of the cosine-based retrieval model.

Tail-tolerant DiS simulator We use the Tarsos-LSH Java implementation [10] of cosine-based LSH for the partitioning. LSH provides a configuration parameter k, which controls the number of shards in the partition. We partition the data into $n = 32$ shards by setting k to 5. We simulate the DiS on a single machine by maintaining a separate index for each shard, as well as for the shard index. We set $r = 3$ in all experiments.

We construct the centralized shard index by sampling documents from every shard with a configured sampling probability. Given a query, we first search the shard index and retrieve a result set of top γ documents; we set $\gamma = 500$ in our experiments (for all queries). We then score the shards based on the result set according to CRCS Linear [78]: The score $S(D)$ of shard D is defined as $S(D) \triangleq \sum_{d \in R_D} S(d)$, where R_D is the subset of the results sampled from shard D, and $S(d) = \gamma - j$, where $1 \leq j \leq \gamma$ is the rank of d in the result set. We normalize CRCS's scores, $\hat{S}(D) \triangleq \frac{S(D)}{\sum_{D'} S(D')}$, in order to produce the shard success probability distribution that CRCS induces.

We simulate distributed query processing as follows: Given a query, we retrieve the top 100 results of each shard. In order to simulate misses of results, we drop the results of each shard with probability f. We union the results of the responsive nodes and omit duplicates (duplicates exist due to redundancy), which yields a result set R of unique documents. Since all shards apply the same scoring function, we rank the documents in R according to their scores and return the top-scored 100 documents in R.

We examine two shard success probability distributions: 1) a uniform success probability distribution that we produce using the Random shard selection, and 2) a skewed success probability distribution that we produce using CRCS with sampling probability of 0.4[3]. Figure 3.3 illustrates the average estimated success probability of the five top-scored shards for the LiveJ and the Reuters datasets produced as follows: For each query in the evaluation set, we estimate the success probabilities of the query's top five shards. We then average each of these five success probabilities over all queries. As the figure illustrates, the Random shard selection induces a uniform success probability distribution, which is identical for both datasets (Figure 3.3(a)). CRCS induces a skewed success probability distribution (Figures 3.3(b) and 3.3(c)), and in particular the most skewed one in LiveJ (Figure 3.3(b)).

Evaluation We evaluate search quality by measuring the average recall@100 over an evaluation set of queries (see Section 3.2.3). For Reuters, we use 200 Trec topics [7] as our evaluation set. For LiveJ, we construct an evaluation set by randomly sampling 1,000 documents from the dataset. We confirm statistical significance using paired-ttest with 5% significance level.

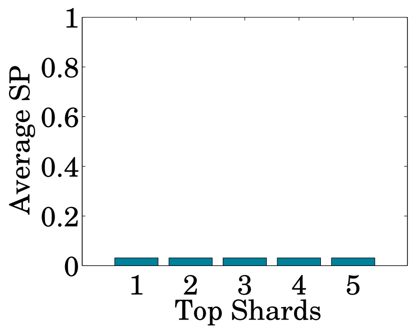
3.5.2 Selection Schemes for Replication

We study rFullRed, NoRed, and rSmartRed selection schemes for Replication and show the superiority of rSmartRed over the other two.

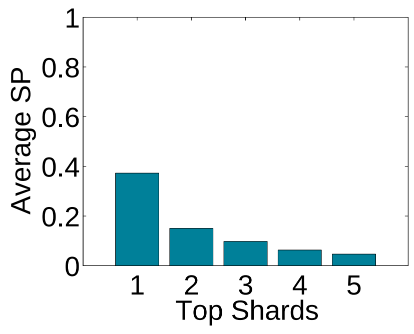
Effect of miss probability and success probability distribution We evaluate recall as a function of f, $0 \leq f \leq 0.5$ (we discuss here relatively high f values that are not necessarily realistic for the purpose of the demonstration; we later zoom-in on lower f values). We fix $t = 5$, i.e., we select $tr = 15$ shards, which are about half of the number of shards in one partition (32). Figure 3.4 presents our results for the Random and the CRCS-based success probability distributions (Random's results are very similar for both datasets, and so, we present here LiveJ's results only).

For both success probability distributions and both datasets, rFullRed achieves relatively stable recall for all miss probabilities. This is since rFullRed uses the maximal number of replicas (r) possible for all shards it selects, which makes the content it selects available regardless of f. However, the recall rFullRed achieves is significantly lower for

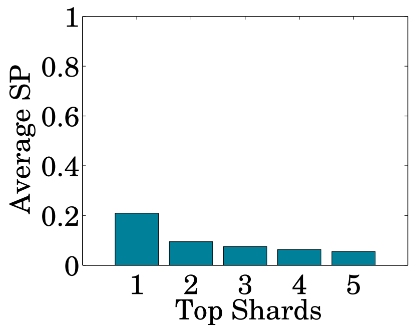
[3]In order to produce a highly skewed success probability using CRCS, for the purpose of demonstration, we use an extremely high sampling probability of 0.4.



(a) Random



(b) LiveJ CRCS



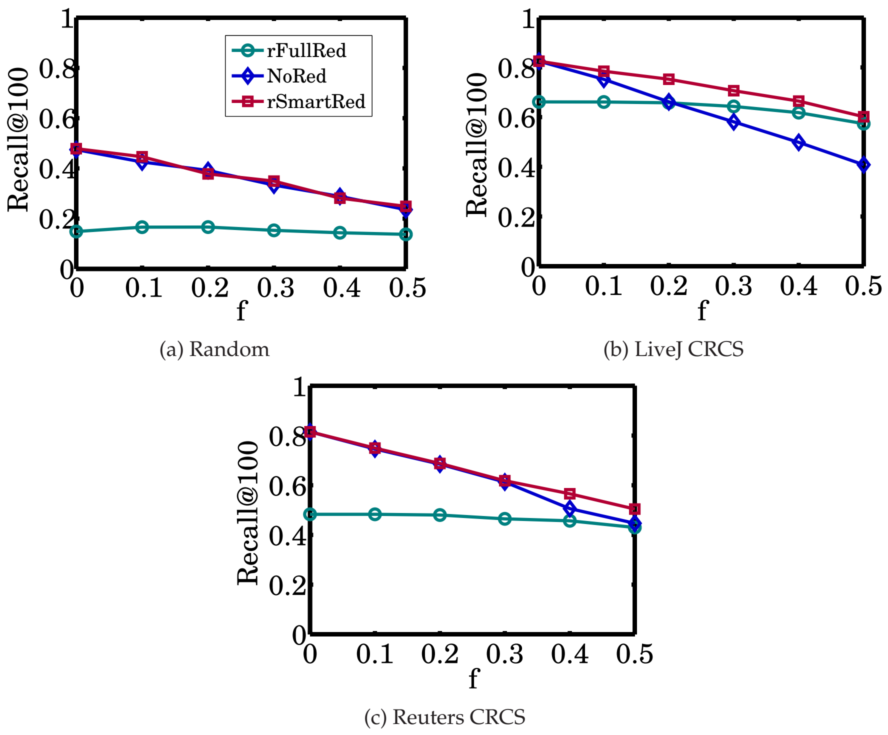
(c) Reuters CRCS

Figure 3.3: Random and CRCS success probability distributions over the Reuters and LiveJ datasets.

low miss probabilities compared to NoRed. This is since searching a large number of replicas is wasteful when misses are infrequent; it is more beneficial to select additional shards by decreasing the number of shard replicas. At the other extreme, NoRed selects a single replica of each shard. It achieves higher recall compared to rFullRed when miss probability is low, as it searches more distinct shards. However, when miss probability increases, NoRed's recall decreases. This tradeoff is most pronounced in LiveJ when the success probability distribution is highly skewed (Figure 3.4(b)), where NoRed's recall drops below the recall of rFullRed for f values that exceed 0.2. This is since in the case of a skewed success probability distribution, the responsiveness of top-scored shards is crucial, hence contacting their replicas is beneficial. When the success probability distribution is uniform (Figurs 3.4(a)), NoRed's recall becomes close to that of rFullRed for $f = 0.5$. This is since in this case, when a shard fails to respond, contacting one of its replicas (rFullRed) or contacting another shard in the partition (NoRed) contributes similarly to the success probability.

As expected, for both distributions, for all values of f, rSmartRed's recall is at least as good as those of rFullRed and NoRed. When the success probability distribution is close to uniform (Figure 3.4(a)), rSmartRed and NoRed behave similarly, since not much is gained from redundancy. But when the success probability distribution is skewed

(Figures 3.4(b) and 3.4(c)), as is common for many queries, rSmartRed outperforms both rFullRed and NoRed by adjusting its selection to the miss probability. For example, as demonstrated in Figure 3.4(b), rSmartRed achieves a statistically significant higher recall then rFullRed for $f < 0.2$ and a statistically significant higher recall than NoRed for $f > 0.2$.



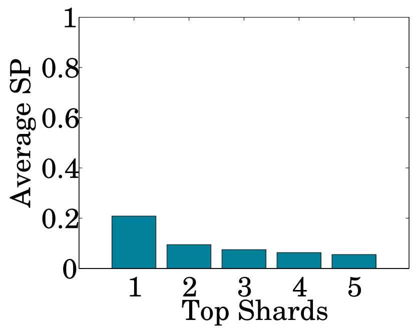
(a) Random (b) LiveJ CRCS (c) Reuters CRCS

Figure 3.4: Recall@100 for the three shard selection schemes for Replication as a function of miss probability f. rSmartRed outperforms both rFullRed and NoRed in all scenarios.

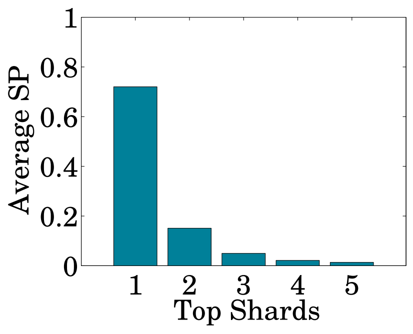
We next zoom in on smaller f values, $0 \leq f \leq 0.2$. We examine three CRCS-based (non-uniform) success probability distributions for LiveJ by considering the following three sets of queries:

- *Whole* consists of all queries in LiveJ's evaluation set.
- *Skewed* consists of queries in LiveJ's evaluation set for which the success probability of the top shard is greater than 0.5; 26.3% of queries are in this category.
- *MostSkewed* consists of queries in LiveJ's evaluation set for which the success probability of the top shard is greater than 0.8; only 0.092% of queries are in this category.

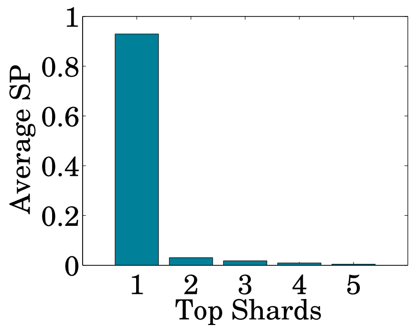
Figure 3.5 illustrates the average estimated success probability of the five top-scored shards for each of the query sets.



(a) Whole query set



(b) Skewed query set



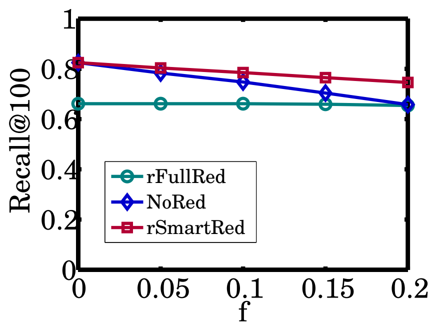
(c) MostSkewed query set

Figure 3.5: Three non-uniform success probability distributions for LiveJ with CRCS-based shard selection, which correspond to three different query sets.

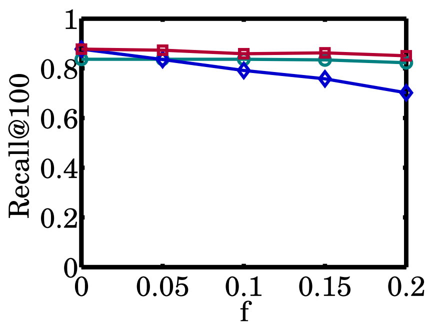
We compare the recall of the three selection schemes over each of the three query sets and present our results in Figure 3.6. Figure 3.6(a) depicts recall measured over the Whole query set. As we have previously seen, NoRed outperforms rFullRed for $0 \leq f \leq 0.2$, however, NoRed's recall decreases as f increases until it reaches rFullRed's recall for $f = 0.2$. As rSmartRed is optimal, it outperforms both.

Next, in Figure 3.6(b), we depict recall for the Skewed query set, which has a more skewed success probability distribution than Whole's. Here too, NoRed's recall is higher than rFullRed's for low miss probabilities, but it drops below rFullRed at a smaller f value (0.05). This is since here, the responsiveness of the top shards is more crucial due to their higher success probability. Hence, Replication becomes valuable for lower miss probability values than in the previous case. As before, rSmartRed selects the number of replicas in an optimal manner and so outperforms both rFullRed and NoRed.

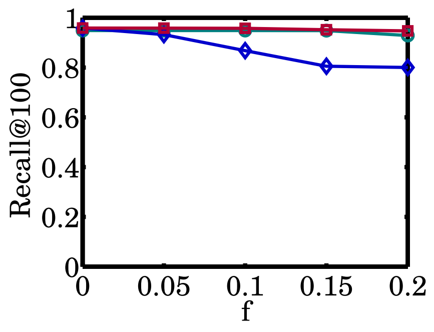
Finally, Figure 3.6(c) examines the recall over the MostSkewed query set, which has the most skewed success probability distribution. In this extreme case, the average success probability of the top shard is 0.92, hence searching a single shard – the top one – is crucial. As both rFullRed and NoRed select the top shard due to its high success probability (Observation 1), they both achieve high recall (0.96) when misses are infrequent ($f \leq 0.05$). When the miss probability increases ($f > 0.05$), NoRed's recall drops below rFullRed's recall as NoRed does not employ redundancy. For all f values that we examined, rSmartRed is optimal.



(a) Whole query set



(b) Skewed query set



(c) MostSkewed query set

Figure 3.6: LiveJ Recall@100 for the three shard selection schemes for Replication as a function of miss probability f for three different query sets, each inducing a different success probability distribution. rSmartRed outperforms both rFullRed and NoRed in all scenarios.

Effect of the number of selected shards We wrap up the study of Replication by varying tr, i.e., the number of selected shards. We experiment with $t \in \{3, 5, 8, 10\}$ values. As we fix $r = 3$, this yields up to 30 shards, which is almost the number of shards in the partition: $n = 32$. We fix $f = 0.1$. Figure 3.7 depicts our results for the LiveJ dataset (we omit Reuters results, which do not provide additional insight). For all selection schemes, the recall increases with the number of selected shards tr, as expected. Second, for all tr values that we examine, rSmartRed's recall is equal to or greater than the recall of NoRed and rFullRed, which confirms our theory.

Searching shard replicas is useless in case of a uniform success probability distribution. Indeed, in this case, rFullRed performs worse than NoRed and rSmartRed (Figure 3.7(a)). rFullRed's inferiority becomes more pronounced as tr increases, since the number of replicas that it wastefully selects increases. When the distribution is highly skewed (Figure 3.7(b)), few shards have high success probabilities, hence searching additional shards becomes unproductive at some point. Indeed, we observe a diminishing returns in NoRed's recall when *tr* increases. In contrast, rSmartRed continues to improve by selecting replicas of high probability shards.

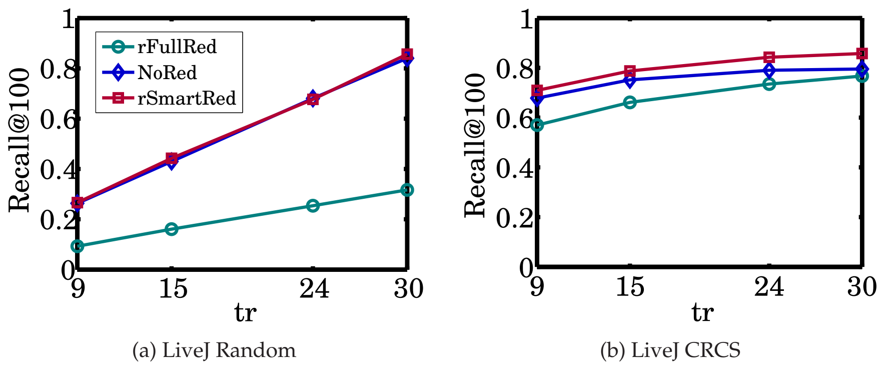


Figure 3.7: LiveJ Recall@100 for the three shard selection schemes for Replication as a function of the number of selected shards (*tr*). rSmartRed outperforms both rFullRed and NoRed in all scenarios.

3.5.3 Replication vs. Repartition

We move on to compare between Replication and Repartition as shown in Figure 3.8. NoRed is identical for both redundancy methods, hence we omit it from the comparison. As real deployments attempt to maintain a low miss probability, we experiment with $0 \leq f \leq 0.2$. Additionally, as real deployments attempt to use good predictors for shard selection, we experiment with the CRCS success probability distribution.

Figure 3.8(a) depicts recall as a function of miss probability for a fixed $t = 5$. According to pSmartRed's specification, rSmartRed and pSmartRed select the same number of shards per partition. pSmartRed achieves a statistically significant higher recall than rSmartRed thanks to using Repartition, which confirms our analysis. Similarly, rFullRed and pTop select the same number of shards per partition, t. Here as well, pTop achieves a statistically significant improvement over rFullRed for the same reason. Overall, Repartition achieves a statistically significant higher recall than Replication for low miss probabilities and skewed success probabilities, which reflects an important practical use case for real deployments. In Figure 3.8(b) we fix $f = 0.1$ and vary tr. We observe Repartition's superiority for all examined tr values.

3.6 Summary

In this chapter, we tackled search quality degradation in DiS due to high tail latency. We observed that using Replication can be wasteful due to searching identical shard copies,

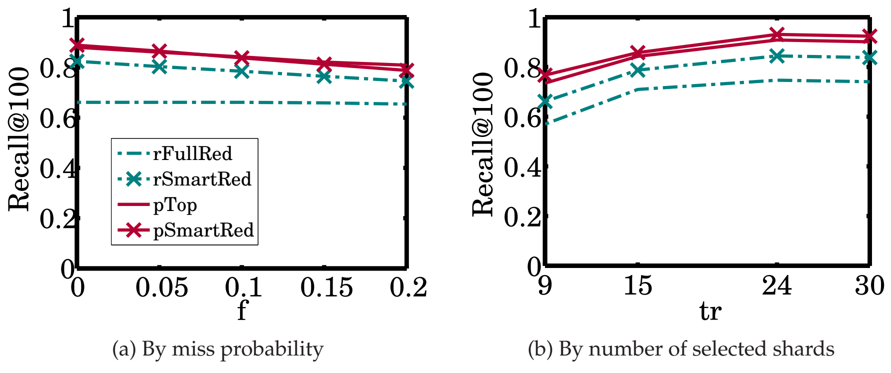


Figure 3.8: LiveJ Recall@100 with Replication and Repartition for a skewed success probability distribution. Repartition outperforms Replication.

and proposed two improvements: rSmartRed and Repartition. We presented two shard selection schemes for Replication that are in use today: NoRed, which does not use redundancy, i.e. does not contact shard replicas, and rFullRed which contacts all replicas of each selected shard. We showed that NoRed is beneficial when miss probability is low, however, when miss probability increases, the search quality of NoRed degrades. In this case, searching multiple shard copies becomes beneficial, and the search quality of rFullRed improves over NoRed's. In particular, we showed that this trade-off is most pronounced when the shard success probability is skewed. We presented rSmartRed, an optimal shard selection scheme for Replication, which considers both the miss probability and the shard success probability when selecting shard copies. rSmartRed selects shards and their number of replicas by applying a simple scoring function over the shard replicas. We showed that rSmartRed outperforms NoRed and rFullRed, hence, rSmartRed is appealing for improving search quality in replication-based tail-tolerant DiS.

We next proposed Repartition, an alternative to Replication, which reduces Repartition's waste by constructing non-exact partition copies. We showed that Repartition improves over Replication when the miss probability is low and the shard success probability is skewed, which reflects an important practical use case for real deployments. The drawback of Repartition is its higher construction and maintenance cost compared to Replication. In addition, Repartition is not applicable when the partitioning is provided by a third party and can not be altered, whereas Replication is applicable in such cases as well.

Chapter 4

Similarity Search over Endless Data

Users today are exposed to massive volumes of information arriving in endless data streams: hundreds of millions of content items are generated daily by billions of users through widespread social media platforms [83, 85, 11]; fresh news headlines from different sources around the world are aggregated and spread by online news services [61, 38]. In this era of information explosion it has become crucial to 'fish in the stream', namely, identify stream content that will be of interest to a given user. Indeed, search and recommendation services that find such content are ubiquitously offered by major content providers [38, 41, 61, 34, 36].

A fundamental building block for search and recommendation applications is *similarity search*, an algorithmic primitive for finding similar content to a queried item [80, 17]. For example, a user reading a news item or a blog post can be offered similar items to enrich his reading experience [87]. In the context of streams, many works have observed that applications ought to take into account temporal metrics in addition to similarity [41, 61, 60, 55, 62, 85, 49, 73, 83, 70]. Nevertheless, the similarity search primitive has not been extended to handle endless data-streams. To this end, we introduce here the problem of *similarity search over data streams (SSDS)*.

In order to efficiently retrieve such content at runtime, an SSDS algorithm needs to maintain an *index* of streamed data. The challenge, however, is that the stream is unbounded, whereas physical space capacity cannot grow without bound; this limitation is particularly acute when the index resides in RAM for fast retrieval [83, 65]. A key aspect of an SSDS algorithm is therefore its *retention policy*, which continuously determines which data items to retain in the index and which to forget. The goal is to retain items that best satisfy the needs of users of stream-based applications.

We present *Stream-LSH*, an SSDS algorithm based on *Locality Sensitive Hashing (LSH)*, which is a widely used randomized similarity search technique for massive high dimensional datasets [47]. LSH builds a hash-based index with some redundancy in order to increase recall, and Stream-LSH further takes into account quality, age, and dynamic popularity in determining an item's level of redundancy.

A straightforward approach for bounding the index size is to focus on the freshest items. Thus, when indexing an endless stream, one can bound the index size by eliminating the oldest items from the index once its size exceeds a certain threshold. We refer to this retention policy as *Threshold*. Although such an approach has been effectively used for detecting new stories [83] and streaming similarity self-join [70], it is less ideal for search and recommendations, where old items are known to be valuable to users [58, 87, 24].

We suggest instead *Smooth* – a randomized retention policy that gradually eliminates index entries over time. Since there is redundancy in the index, items do not disappear from it at once. Instead, an item's representation in the index decreases with its age. Figure 4.1 illustrates the probability to find a similar item with the two retention policies using the same space capacity. In this example, the index size suffices for Threshold to retain items for 20 days. We see that Threshold is likely to find fresh similar items, but fails to find items older than 20. Using the same space capacity, Smooth finds similar items for a longer time period with a gradually decaying probability; this comes at the cost of a lower probability to find very fresh items. We further show that Smooth ex-

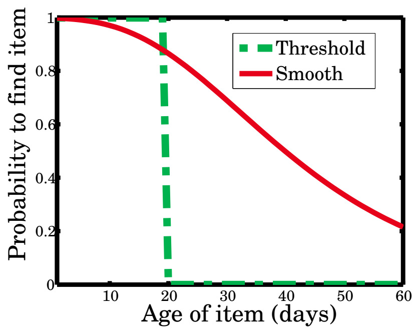


Figure 4.1: Probability of successful retrieval of similar items as a function of their age with Threshold and Smooth retention policies in example settings.

ploits capacity resources more efficiently so that the average recall is larger than with Threshold.

We extend Stream-LSH to consider additional data characteristics beyond age. First, our Stream-LSH algorithm considers items' query-independent quality, and adjusts an item's redundancy in the index based on its quality. This is in contrast to the standard LSH, which indexes the same number of copies for all items regardless of their quality. Second, we present the *DynaPop* extension to Stream-LSH, which considers items' dynamic popularity. DynaPop gets as input a stream of user interests in items, such as retweets or clickthrough information, and re-indexes in Stream-LSH items of interest; thus, it has Stream-LSH dynamically adjust items' redundancy to reflect their popularity.

To analyze Stream-LSH with different retention policies, we formulate the theoretical *success probability (SP)* metric of an SSDS algorithm when seeking items within given similarity, age, quality, and popularity radii. Our results show that Smooth increases the probability to find similar and high quality items compared to Threshold, when using the same space capacity. We show that our quality-sensitive approach is appealing for similarity search applications that handle large amounts of low quality data, such as user-generated social data [15, 23, 28], since it increases the probability to find high-quality items. Finally, we show that using DynaPop, Stream-LSH is likely to find popular items that are similar to the query, while also retrieving similar items that are not highly popular albeit with lower probability. Retrieving similar items from the tail of the popularity distribution in addition to the most popular ones is beneficial for applications such as query auto-completion [20] and product recommendation [93]. We validate our theoretical results empirically on several real-world stream datasets using the recall metric.

4.1 Similarity Search over Data-Streams

We extend the problem of similarity search defined in Section 2.2, and define similarity search over unbounded data-streams. Our stream similarity search is time-sensitive and quality-aware, and so we also define a recall metric that takes these aspects into account.

4.1.1 SSDS

SSDS considers an unbounded *item stream* $U \subseteq V$ arriving over an infinite time period, divided into discrete *ticks*. The (finite) time unit represented by a tick is specified by the application, e.g., 30 minutes or 1 day. On every time tick, 0 or more new items arrive in the stream, and the *age* of a stream item is the number of time units that elapsed since its arrival. Note that each item in U appears only once at the time it is created. Each item is associated with a query-independent quality score, which is specified by a given weighting function $quality : V \to [0,1]$.

Similarity search over data-streams An SSDS algorithm's input consists of a *query* vector $q \in V$ and a three-dimensional radius, $(R_{sim}, R_{age}, R_{quality})$, of similarity, age, and quality radii, respectively. An *exact SSDS algorithm* returns a unique ideal result set

$$\begin{aligned}&Ideal(q, R_{sim}, R_{age}, R_{quality}) \triangleq \\ &\quad \{v \in U | sim(q,v) \geq R_{sim} \wedge age(v) \leq R_{age} \wedge \\ &\quad quality(v) \geq R_{quality}\}.\end{aligned}$$

An (approximate) *SSDS algorithm* A returns
a subset $Appx(A, q, R_{sim}, R_{age}, R_{quality})$ of q's ideal result set.

Recall

Definition 4.1.1 (recall at radius). *The* recall at radius *of algorithm A for query q and radius $(R_{sim}, R_{age}, R_{quality})$ is*

$$\text{Recall}(A, R_{sim}, R_{age}, R_{quality})(q) \triangleq \frac{|Appx(A, q, R_{sim}, R_{age}, R_{quality})|}{|Ideal(q, R_{sim}, R_{age}, R_{quality})|}.$$

The recall at radius $\text{Recall}(A, R_{sim}, R_{age}, R_{quality})$ *of A is the mean recall over the query set Q.*

Dynamic popularity We consider a second unbounded stream I which consists of items from the item stream U and arrives in parallel to U. We call I the *interest stream*. The arrival of an item at some time tick in I signals interest in the item at that point in time. Note that an item may appear multiple times in the interest stream.

We capture an item's *dynamic popularity* by a weighted aggregation of the number of times it appears in the interest stream, where weights decay exponentially with time [59]:

Let $t_0, \ldots, t_n$ denote time ticks since the starting time t_0, and the current time t_n. The indicator $a_i(x)$ is 1 if item x appears in the interest stream at time t_i and is 0 otherwise. A parameter $0 < \alpha < 1$ denotes the *interest decay*, which controls the weight of the interest history and is common to all items.

Definition 4.1.2 (item popularity)**.** *The function $pop : U \to [0, 1]$ assigns a popularity score pop(x) to an item $x \in U$:*

$$pop(x) \triangleq (1 - \alpha) \sum_{i=0}^{n} a_i(x) \alpha^{(n-i)}.$$

Given an assignment of popularity scores to items, we are interested in the retrieval of items within a popularity radius $R_{pop} \in [0, 1]$, i.e., with a popularity score that is not lower than R_{pop}. We define recall in a similar manner to the previous definitions.

4.2 Stream-LSH

Stream-LSH is an extension of Locality Sensitive Hashing overviewed in Section 2.3 for unbounded data-streams, augmented with age, quality, and dynamic popularity dimensions. Stream-LSH consists of a retention policy that defines which items are retained in the index and which are eliminated as new items arrive.

4.2.1 Stream-LSH

Stream-LSH, presented in Algorithm 1, extends LSH's indexing procedure to operate on an unbounded data-stream. Every time tick, Stream-LSH accepts a set of newly arriving items in the item stream U and indexes each item into its LSH buckets. Stream-LSH selects an item's initial redundancy according to its quality: it indexes the item into each bucket with a probability that equals its quality, independently of other buckets. In addition, in order to bound the index size, in each time tick, Stream-LSH eliminates items from the index according to the retention policy it uses. Note that the two operations – indexing new items and eliminating old ones – are independent, and indexing and elimination work independently in each bucket.

4.2.2 Retention Policies

We first describe the Threshold [83, 70] and Bucket [73] policies, which have been used by prior art in other contexts. Then, we describe our randomized Smooth policy that gradually eliminates item's copies from the index as a function of its age.

Threshold and Bucket

The Threshold retention policy [83, 70] presented in Algorithm 2 sets a limit T_{size} on table size, and eliminates the oldest items from all L tables once the size limit is exceeded. Note

Algorithm 1 Stream-LSH

```
1: On every time tick $t$ do:
2: foreach $H_i \in HashTables$ do
3:     foreach $item \in items(t)$ do
4:         ▷ Hash to bucket $B_i$
5:         $B_i \leftarrow g_i(item)$
6:         ▷ Quality-based indexing
7:         with probability $quality(item)$, $B_i$.ADD($item$)
8:         ▷ Elimination by retention policy
9:         $H_i$.ELIMINATE()
10:    end foreach
11: end foreach
```

that with Threshold, the number of copies of an item in the index does not vary with age.

Algorithm 2 Threshold retention policy

```
1: function H.ELIMINATE
2:     remove $|H|$ - $T_{size}$ oldest items in table
3: end function
```

The Bucket retention policy [73] given in Algorithm 3 sets a limit B_{size} on bucket size (rather than on table size), and eliminates the oldest items in each bucket once its size limit is exceeded. Note that with Bucket, the number of copies of an item in the index varies with age, since each bucket is maintained independently. The probability of an item to be eliminated from a bucket depends on the data distribution, i.e., on the probability that newly arriving items will be mapped to that item's bucket.

Algorithm 3 Bucket retention policy

```
1: function H.ELIMINATE
2:     foreach $B_i \in H$ do
3:         remove $|B_i|$ - $B_{size}$ oldest items in bucket
4:     end foreach
5: end function
```

Smooth

In Algorithm 4 we present *Smooth*, our randomized retention policy that gradually eliminates item copies from the index as a function of their age. Smooth accepts as a parameter a *retention factor* p, $0 < p < 1$. Upon a time tick, Smooth eliminates each existing item copy from its bucket with probability $1 - p$, independently of the elimination of other items. The number of buckets an item is indexed into thus exponentially decays over time. As we show in Section 4.3.1, Smooth entails an expected bounded index size that is a function of p.

Algorithm 4 Smooth retention policy

```
1: function H.ELIMINATE
2:     foreach item in $H$ do
3:         with probability $1 - p$, remove item from $H$
4:     end foreach
5: end function
```

Although as described in Algorithm 4 Smooth entails a linear scan of all items in all hash tables at each time unit, it can be implemented efficiently by randomly and uniformly selecting a fraction of $1 - p$ of the items to eliminate from each table.

4.2.3 Dynamic Popularity

DynaPop extends Stream-LSH indexing procedure to dynamically re-index items based on signals of user interests, as reflected by the interest stream I. Here, an item's redundancy increases as the interest in it increases. At each time tick, DynaPop re-indexes an item that arrives in I into each of its buckets with probability $quality(x)u$ independently of other buckets; the *insertion factor*, $0 < u < 1$, is a parameter to the algorithm. Note that in this context, an item's quality may also change dynamically over time. At each time tick, the current quality value is considered.

4.3 Analysis

In this section, we analyze Stream-LSH's index size and prove that it maintains a bounded index. We additionally analyze the probability that Stream-LSH finds items within similarity, age, quality, and popularity radii.

4.3.1 Index Size and Number of Retained Copies

Index size We first analyze the expected index size using Smooth with a retention factor p assuming that a constant number of new items μ arrive at each time unit, and that their mean quality is ϕ. Consider one hash table, and denote time ticks as $t_0, \ldots, t_n$. At time t_0, Smooth stores $\mu\phi$ items in the hash table in expectation. A ratio $1 - p$ of these $\mu\phi$ items are removed at every time tick, and thus the expected number of items that arrive at t_0 and survive elimination until t_n is $p^n\mu\phi$. It follows that the expected number of items in the table at any given time during the processing of an infinite stream is $\sum_{i=0}^{\infty} p^i\mu\phi = \frac{\mu\phi}{1-p}$. The retention process is performed independently in each of the L hash tables, therefore,

Proposition 1. *If μ new items with mean quality ϕ arrive at each time unit, the expected size of an index with L hash tables using Smooth with retention factor p is* $\frac{\mu\phi}{1-p}L$.

Next, assume that the arrival rate is not constant, but the number of new items that arrive at each time unit is bounded by μ^*, which is a reasonable assumption in practical systems. Further note that ϕ is bounded by 1. Thus, at most μ^* items are indexed into each hash table at each time unit. The number of items that arrive at t_0 and survive elimination until t_n is therefore bounded by $\frac{\mu^*}{1-p}L$.

Number of retained copies Next, we analyze the evolution of an item's number of copies in the index as a function of its age and quality according to Threshold and Smooth. We omit Bucket from our analysis due to its dependency on the data distribution. We examine Threshold and Smooth when using the same index size: $20\mu\phi L$ in expectation, and so we set $T_{size} = 20\mu\phi$ for Threshold, $p = 0.95$ for Smooth (Proposition 1), and $L = 15$ for both.

Let x be some item. Threshold retains $quality(x)L$ copies of x in expectation if $age(x) < 20$, and zero copies otherwise. Smooth retains $quality(x)p^{age(x)}L$ copies of x in expectation. Figure 4.2 illustrates the number of index copies of an item with quality 1 (solid line), and of an item with quality 0.5 (dashed line), as a function of the item's age. Due to its quality-based indexing, Stream-LSH (for both Threshold and Smooth) retains a smaller number of copies of low quality items compared to high quality ones. Additionally, as Smooth's name suggests, it smoothly decays an item's number of copies in contrast with Threshold. This difference between the two policies impacts their effectiveness as we analyze in Section 4.3.2.

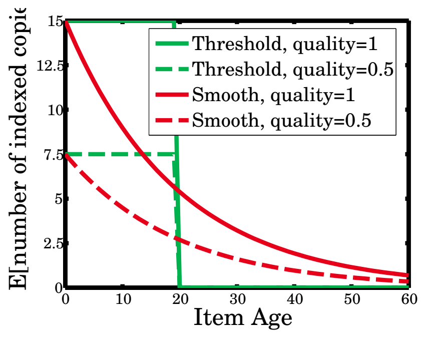


Figure 4.2: Expected number of copies of an item in the index as a function of its age, for items with quality values of 1 and 0.5.

4.3.2 Success Probability

Success probability quantifies the probability of an algorithm to find some item x given a query q [64]. For an LSH algorithm A with parameters k and L, we denote this probability by $SP(A(k, L))$. We sometimes omit k, L where obvious from the context.

According to LSH theory for angular similarity [32], the probability of an $LSH(k, L)$ algorithm to find x that is s-similar to q in bucket $g_i(q)$ equals s^k, under a random selection of $g_i \in \mathcal{G}$. LSH searches in L buckets $g_1(q), \cdots, g_L(q)$ independently, thus, $LSH(k, L)$ finds x with probability $1 - \left(1 - s^k\right)^L$.

Retention Policies

We analyze Stream-LSH success probability when using Threshold and Smooth. We do not analyze Bucket, as its behavior depends on the data distribution, which we do not model.

Success probability We denote by $SP(A(k, L), s, a, z)$ the probability of algorithm $A(k, L)$ to find an item x for query q, s.t. $sim(q, x) = s$, $age(x) = a$, and $quality(x) = z$.

Stream-LSH indexes a newly arriving item x into each bucket $g_i(x)$ independently with probability z. For a constant arrival rate μ, a mean quality ϕ, and a size limit T_{size}, Threshold eliminates items that reach age $T_{age} = \frac{T_{size}}{\mu\phi}$. Thus,

$$SP(\text{Threshold}(k, L), s, a, z) = \begin{cases} 1 - (1 - s^k z)^L, & \text{if } a < T_{age} \\ 0, & \text{otherwise} \end{cases} \tag{4.1}$$

Smooth retains an item x in the index with probability $p^{age(x)}$, thus,

$$SP(Smooth(k, L), s, a, z) = 1 - (1 - p^a s^k z)^L. \tag{4.2}$$

Numerical illustration We compare the success probabilities of Threshold and Smooth. In order to achieve a fair comparison, we fix k, L, and the index size. Given that our treatment of quality is orthogonal to the retention policies, our example ignores quality, and so we assume $quality(x) = 1$ for all items x. For the purpose of the illustration, we select a configuration where $k = 10$ and $L = 15$; we set $T_{size} = 20\mu$ and $p = 0.95$ yielding a common index size for both policies (Proposition 1).

Figure 4.3 illustrates as 'heat maps' the success probabilities of Threshold and Smooth for this configuration. The x axis denotes similarity values s, and the y axis denotes age values a. Figure 4.3(a) depicts $SP(\text{Threshold}(10, 15), s, a, 1)$, while figure 4.3(b) depicts $SP(\text{Smooth}(10, 15), s, a, 1)$.

As Threshold completely eliminates all item's copies that reach age 20, the success probability for $a \geq 20$ is 0 (colored white). The success probability of newer items behaves according to standard LSH, i.e., the more similar an item is to the query (s is closer to 1), the higher the success probability (color tends towards red). Fixing an s value, the success probability remains constant as a increases, since the number of buckets an item is indexed into remains constant. With Smooth, on the other hand, for a fixed s value, the success probability gradually decays as a increases. Smooth retains items for a longer time period than Threshold, and thus the success probability is non-zero for items older than 20.

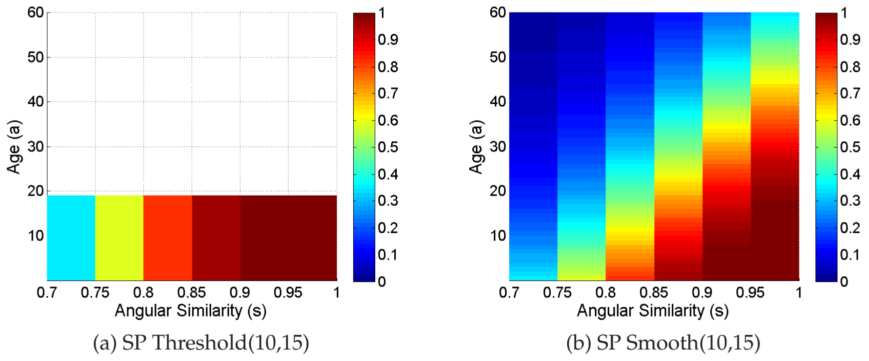


Figure 4.3: Success probability to find an item according to similarity and age for a common index size.

Cumulative success probability We next formulate the *cumulative success probability (CSP)* over similarity, age, and quality radii. Given a query q, a similarity radius R_{sim}, an age radius R_{age}, and a quality radius $R_{quality}$, CSP quantifies an algorithm's probability to find an item x for which $sim(q,x) \in [R_{sim}, 1]$, $age(x) \in [0, R_{age}]$, and $quality(x) \in [R_{quality}, 1]$. CSP is the expected SP over all choices of $s \in [R_{sim}, 1]$, $a \in [0, R_{age}]$, and $z \in [0, R_{quality}]$ and is given by the following equation:

$$CSP(A, R_{sim}, R_{age}, R_{quality}) = \\ \int_{z=R_{quality}}^{1} \int_{a=0}^{R_{age}} \int_{s=R_{sim}}^{1} \frac{f(s,a,z)SP(A,s,a,z)}{\psi(R_{sim}, R_{age}, R_{quality})} dsdadz,$$

where $f(s,a,z)$ denotes the joint probability density function of similarity s, age a, and quality z,
and

$$\psi(R_{sim}, R_{age}, R_{quality}) = \\ \int_{z=R_{quality}}^{1} \int_{a=0}^{R_{age}} \int_{s=R_{sim}}^{1} f(s,a,z)dsdadz,$$

is a normalization factor.

Threshold keeps items up to age $\frac{1}{1-p}$; from (4.1) and (4.2):

$$CSP(Threshold(k,L), R_{sim}, R_{age}, R_{quality}) = \\ \int_{z=R_{quality}}^{1} \int_{a=0}^{\frac{1}{1-p}} \int_{s=R_{sim}}^{1} \frac{f(s,a,z)(1-(1-s^k z)^L)}{\psi(R_{sim}, R_{age}, R_{quality})} dsdadz$$

and

$$CSP(Smooth(k,L), R_{sim}, R_{age}, R_{quality}) = \int_{z=R_{quality}}^{1} \int_{a=0}^{R_{age}} \int_{s=R_{sim}}^{1} \frac{f(s,a,z)(1-(1-p^a s^k z)^L)}{\psi(R_{sim}, R_{age}, R_{quality})} dsdadz$$

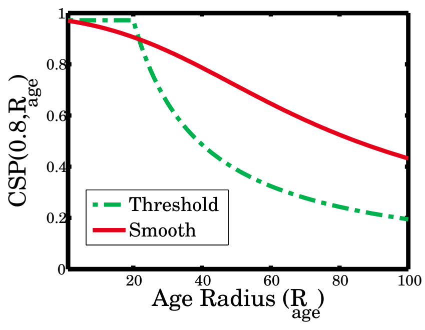
Numerical illustration We compare $CSP(A, R_{sim}, R_{age}, R_{quality})$ of Stream-LSH with Threshold and Smooth. We pose the following assumptions: we focus on the effect of the retention policy and hence we assume a constant quality function $\forall x \; quality(x) = 1$. In general, the items' similarity distribution is data-dependent, here, we assume a uniform distribution. We consider a discrete age distribution because time is partitioned into discrete time ticks. We assume a constant number of items arriving at each time unit, hence items' age is distributed uniformly. Last, we assume that similarity and age are independent. Under these assumptions we get:

$$CSP(Threshold(k,L), R_{sim}, R_{age}) = \sum_{a=0}^{\frac{1}{1-p}} \int_{s=R_{sim}}^{1} \frac{(1-(1-s^k)^L)}{R_{age}(1-R_{sim})} ds,$$

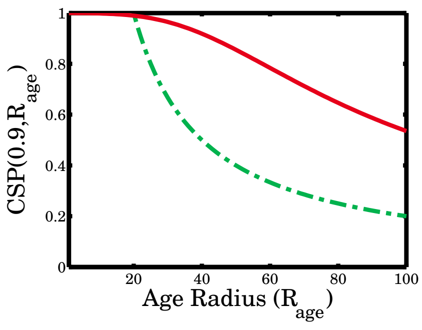
and

$$CSP(Smooth(k,L), R_{sim}, R_{age}) = \sum_{a=0}^{R_{age}} \int_{s=R_{sim}}^{1} \frac{(1-(1-p^a s^k)^L)}{R_{age}(1-R_{sim})} ds.$$

For the purpose of the illustration, we select the same configuration as above: $k = 10$, $L = 15$, $T_{size} = 20\mu$, and $p = 0.95$. Figures 4.4(a) and 4.4(b) depict CSP for fixed R_{sim} values 0.8 and 0.9, respectively, and a varying age radius R_{age}. The graphs show a freshness-similarity tradeoff between Threshold and Smooth. Smooth has a better CSP for high age radii (R_{age} exceeds 20), for any R_{sim} value. This comes at the cost of a decreased CSP for similarity radius 0.8 when $R_{age} \leq 20$.



(a) $R_{sim} = 0.8$



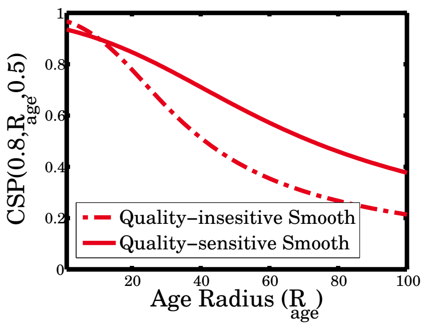
(b) $R_{sim} = 0.9$

Figure 4.4: Cumulative success probability of Threshold and Smooth by age radius for a common index size.

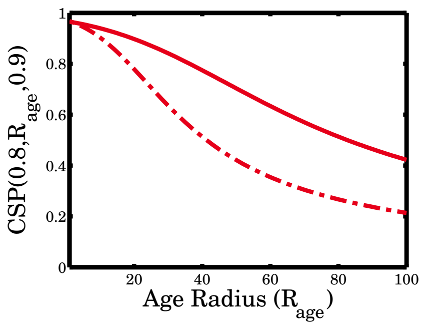
Quality-Sensitivity

Some applications, most notably social media ones, commonly handle content of varying quality, and in particular, large amounts of low quality user-generated content [15, 23, 28]. We show that for such applications, quality-sensitive indexing is expected to be attractive, as it can better utilize space for improving the CSP of high quality items. We compare Stream-LSH's quality-sensitive indexing, which indexes an item with redundancy that is proportional to its quality, to a quality-insensitive Stream-LSH, which indexes L copies of each item regardless of its quality. We use the Smooth retention policy and the same index size for both variants.

Assuming an average quality of 0.5, the expected number of newly indexed items per table is μ according to quality-insensitive indexing, and 0.5μ according to quality-sensitive indexing. To obtain a common index size of $10\mu L$, we use $p = 0.9$ in the quality-insensitive algorithm and $p = 0.95$ in the quality-sensitive one (Proposition 1). Figure 4.5 illustrates the two Stream-LSH variants for $R_{sim} = 0.8$ and varying R_{age} radii. In Figure 4.5(a), we examine items above the mean quality ($R_{quality} = 0.5$), and in Figure 4.5(b) we examine high-quality items ($R_{quality} = 0.9$). Both figures show that quality-sensitive indexing increases the CSP compared to the quality-insensitive variant. The improvement is even more marked for high-quality items ($R_{quality} = 0.9$).



(a) $R_{quality} = 0.5$



(b) $R_{quality} = 0.9$

Figure 4.5: Cumulative success probability comparison of quality-sensitive and quality-insensitive Stream-LSH by age radius for a common index size.

Dynamic Popularity

We next analyze an item's success probability according to Stream-LSH when using DynaPop and the Smooth retention policy. We first formulate the *bucket probability (SB)* to find an item in a bucket it is hashed to. We then use SB to formulate an item's success probability.

For the sake of the analysis, we assume that the interest in an item does not vary over time. At each time t_i, x is included in the interest stream with some probability ρ_x, which we call the item's *interest probability*. That is, for all t_i, x appears with probability

ρ_x. According to Definition 4.1.2 and due to the linearity of expectation:

$$E(pop(x)) = (1-\alpha)\sum_{i=0}^{n} E(a_i(x))\alpha^{(n-i)},$$

which is a geometrical series that converges to $E(a_i(x))$ when $n \to \infty$. As our stream is infinite and $E(a_i(x)) = \rho_x$:

$$E(pop(x)) = \rho_x. \tag{4.3}$$

When clear from the context, we omit x and denote ρ for brevity.

Bucket probability We denote by $SB(p, u, \rho, z, n)$ the probability that x is stored in its bucket at time t_n, where u and p are the insertion and retention factors, respectively, $quality(x) = z$, and ρ is x's interest probability. We denote by E_i, $0 \le i \le n$, the event that x is inserted to its bucket at time t_i, and survives elimination until time t_n, but is not selected for insertion to its bucket at any subsequent time t_j, $i < j \le n$. Then

$$Pr(E_i) = zu\rho p^{(n-i)}(1-zu\rho)^{n-i} = zu\rho[p(1-zu\rho)]^{n-i}$$

Since $SB(p, u, \rho, z, n) = Pr(\bigcup_{i=0}^{n} E_i)$ and $\bigcup_{i=0}^{n} E_i$ is a union of pairwise disjoint events, it follows that

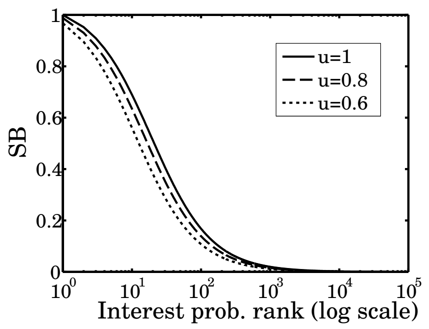
$$SB(p, u, \rho, z, n) = \sum_{i=0}^{n} zu\rho[p(1-zu\rho)]^{n-i}.$$

$SB(p, u, \rho, z, n)$ is a geometric series that converges to $\frac{zu\rho}{1-p(1-zu\rho)}$ when $n \to \infty$. Our interest stream is infinite, thus:

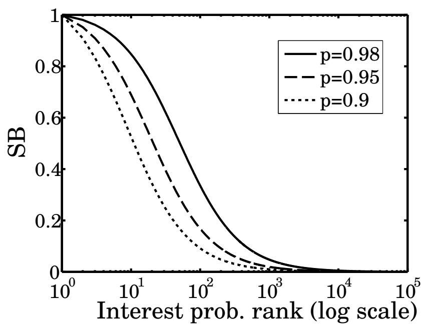
Proposition 2. *Given an item x, the probability $SB(p, u, \rho, z)$ to find item x in its bucket when using Stream-LSH with DynaPop and the Smooth retention policy is $\frac{zu\rho}{1-p(1-zu\rho)}$, where u and p are the algorithm's insertion and retention factors respectively, $quality(x) = z$, and ρ is x's interest probability.*

Numerical illustration We illustrate SB for an interest probability that follows a Zipf distribution (typical in social phenomena [33]) which implies a small number of very popular items and a long tail of rare ones. We consider a Zipf distribution where the r-th ranked item x has an interest probability $\rho_x = 1/r$. We set the quality z to 1.

Figure 4.6 illustrates the SB according to interest probability rank, for different values of u and p. In Figure 4.6(a) we fix $p = 0.95$ and examine the effect of the insertion probability u. The graphs illustrate that increasing u increases SB most notably for popular items. In Figure 4.6(b) we fix $u = 1$ and examine the effect of the retention probability p. The graphs illustrate that when p increases, DynaPop retains additional items of lower popularity.



(a) Insertion factor u



(b) Retention factor p

Figure 4.6: Effect of parameters on the probability to find an item in its bucket according to DynaPop.

Success probability We denote by $SP(DynaPop(k, L), s, w, z)$ the probability of Stream-LSH with DynaPop and the Smooth retention policy to find an item x for query q, s.t. $sim(q, x) = s$, $E(pop(x)) = w$, and $quality(x) = z$. By applying Proposition 2 and as $w = \rho$ (Equation 4.3):

$$SP(DynaPop(k, L), s, w, z) = 1 - (1 - \frac{zuw}{1 - p(1 - zuw)} s^k)^L \tag{4.4}$$

The cumulative success probability is computed similarly to the cumulative success probability analysis in Section 4.3.2, and in particular depends on the distribution of w.

Numerical illustration Figure 4.7 depicts $SP(DynaPop(k, L), s, w, z)$ as a function of w's rank. We illustrate SP for three s values: 0.7, 0.8, and 0.9. We fix $k = 10$, $L = 15$, $z = 1$, $p = 0.95$ and $u = 1$. As the graphs show, an item's success probability increases as

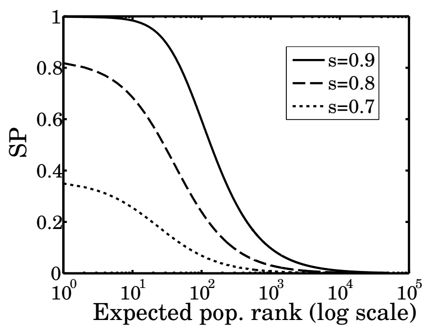


Figure 4.7: Success probability to find an item as a function of its expected popularity according to DynaPop.

its similarity to the query increases. Additionally, an item's success probability increases as its expected popularity increases.

4.4 Empirical Study

We conduct an empirical study of Stream-LSH using real world stream datasets and evaluate its effectiveness using the recall metric.

4.4.1 Methodology

External libraries We use Apache Lucene 4.3.0 [4] search library for the indexing and retrieval infrastructure. For retrieval, we override Lucene's default similarity function by implementing angular similarity according to Equation 2.1. For the LSH family of functions, we use TarsosLSH [10].

Datasets We use Reuters RCV1 [6] news dataset and Twitter [91, 70] social dataset. In both datasets, each item is associated with a timestamp denoting its arrival time. The Reuters dataset consists of news items from August 1996 to August 1997, and the Twitter dataset consists of Tweets collected in June 2009. These datasets do not contain quality information and so we assume $quality(x) = 1$ for all items. In order to evaluate quality-sensitivity, we use a smaller Twitter dataset [12], denoted TwitterNas, consisting of a stream of Nasdaq related Tweets spanning 97 days from March 10th to June 15th 2016. TwitterNas contains number of followers of Tweets authors, which we use for assigning quality scores to Tweets (see Section 4.4.3). In all datasets, we represent an item as a (sparse) vector whose dimension is the number of unique terms in the entire dataset, and each vector entry corresponds to a unique term, weighted according to Lucene's TF-IDF formula.

Train and test We partition each dataset into (disjoint) train and test sets. The train set is the prefix of the item stream up to a tick that we consider to be the current time. The test set is the remainder of the dataset, which was not previously seen by the Stream-LSH algorithm. We randomly sample an evaluation set Q of 3,000 items from the test set and compute recall over Q according to the given radii. Table 4.1 summarizes the train and test statistics.

		Train		Test	
	Time unit	Num. items	Num. ticks	Num. items	Num. ticks
Reuters	Day	756,927	343	22,986	10
Twitter	10 Minutes	18,224,293	2,705	42,296	10
TwitterNas	Day	275,946	92	18,831	5

Table 4.1: Train and test statistics.

4.4.2 Retention Policies

We evaluate the recall of the three retention policies for the Reuters and Twitter datasets as a function of age. As the retention aspect of our algorithm is orthogonal to the quality-sensitive indexing aspect, we assume here that $quality(x) = 1$ for all items. In order to achieve a fair comparison, we use $k = 10$ and $L = 15$ for the three retention policies, and configure them to use approximately the same index size: We set $T_{size} = 45{,}000$ and $B_{size} = 45$ in Reuters; $T_{size} = 180{,}000$ and $B_{size} = 177$ in Twitter; $p = 0.95$ in both datasets.

Figure 4.8 depicts our recall results for Reuters in the top row, and Twitter in the bottom row. Our goal is to retrieve items that are similar to the query, hence we focus on R_{sim} values 0.8, and 0.9. As we are also interested in the retrieval of items that are not highly fresh, we evaluate recall over varying age radii values.

When considering $R_{sim} = 0.8$ (leftmost column) there is a tradeoff between Threshold and Smooth: when focusing on the highly fresh items ($R_{age} < 20$), Threshold's recall is slightly larger than Smooth's. Indeed, Threshold is effective when only the retrieval of the highly fresh items is desired. However, Smooth outperforms Threshold when the age radius increases to include also less fresh items. For example, in Reuters, Smooth achieves a recall of 0.69 for items that are at least 0.8-similar to the query and are not older than age 50, and Threshold achieves a lower recall of 0.42. Bucket's recall is higher than Threshold's for ages that exceed 20, as unlike Threshold, Bucket does not eliminate items at once. Yet, Smooth outperforms Bucket when increasing the age radius due to applying an explicit gradual elimination over all items. When increasing the similarity radius to $R_{sim} = 0.9$ (leftmost column), the advantage of Smooth over Threshold becomes pronounced. For example, in Twitter, Smooth achieves a recall of 0.97 for items that are not older than age 50, whereas Threshold only achieves a recall of 0.7.

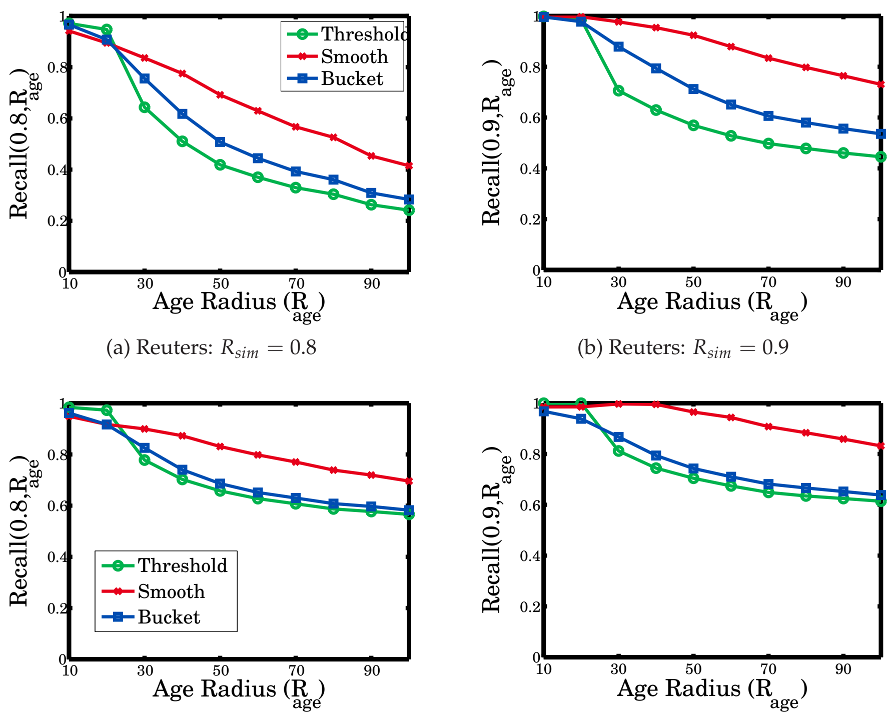
4.4.3 Quality-Sensitivity

We move on to evaluating Stream-LSH's quality-sensitive approach. We experiment with the TwitterNas dataset, which contains for each Tweet x the number of followers of its author representing its authority, and denoted $T_f(x)$. We define the following quality scoring function:

$$quality(x) = log_2(1 + min(1, T_f(x)/N_f)),$$

where N_f is a configurable normalization factor. In our experiments, we set $N_f = 5{,}000$ (15% of the authors have more than 5,000 followers). Applying $quality(x)$ on TwitterNas entails an average quality score of 0.33.

We experiment with quality-sensitive and quality-insensitive variants of Smooth, with $k = 10$ and $L = 15$. In order to conduct a fair comparison, we set retention factors that entail approximately the same index size for both variants. More specifically, we set $p = 0.9$ for the quality-insensitive variant, which results in an index size of 636,290 items in our experiment, and $p = 0.97$ for the quality-sensitive variant which results in



(a) Reuters: $R_{sim} = 0.8$

(b) Reuters: $R_{sim} = 0.9$

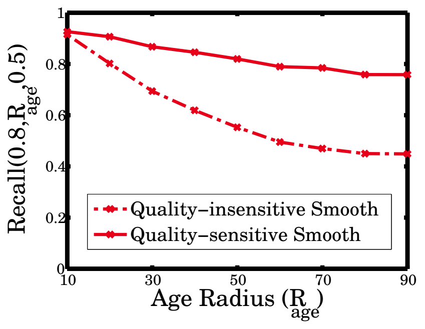
(c) Twitter: $R_{sim} = 0.8$

(d) Twitter: $R_{sim} = 0.9$

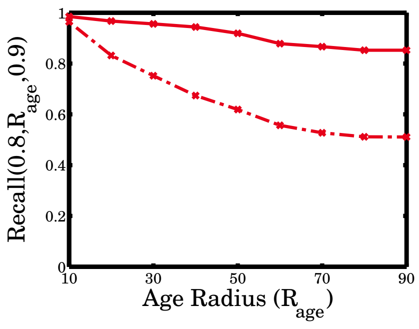
Figure 4.8: Recall comparison by age radius of the three retention policies using approximately the same index size.

an index size of 590,818 items in our experiment. Recall that quality-sensitive indexing is more compact, which enables a slower removal of item copies. This is reflected by a larger p value in the quality-insensitive case. We experiment with the same radii as in our analysis: we fix $R_{sim} = 0.8$, and experiment with $R_{quality} = 0.5$, and $R_{quality} = 0.9$ over varying age values. Figure 4.9 depicts the recall achieved by the two Smooth variants as a function of the age radius.

The graphs demonstrate that for both $R_{quality}$ values, the quality-sensitive approach significantly outperforms the quality-insensitive approach when searching for similar items ($R_{sim} = 0.8$) over all age radii values that we examined. This is since the quality-sensitive approach better exploits the space resources for high quality items. The advantage of quality-sensitive indexing increases as the age of high-quality items increases, which is an advantage when the retrieval of items that are not necessarily the most fresh ones is desired. For example, considering $R_{quality} = 0.5$ (Figure 4.9(a)) and $R_{age} = 30$, the recall achieved by quality-insensitive Smooth is 0.7, whereas the recall achieved by quality-sensitive Smooth is 0.88. When considering $R_{age} = 90$, the recall of quality-insensitive Smooth is 0.45, whereas the recall of quality-sensitive Smooth is 0.76. A similar trend is observed for $R_{quality} = 0.9$ (Figure 4.9(b)). Note that the graphs of $R_{quality} = 0.9$ and $R_{quality} = 0.5$ differ due to the different distributions of similarity,



(a) $R_{quality} = 0.5$



(b) $R_{quality} = 0.9$

Figure 4.9: Recall comparison of quality-insensitive and quality-sensitive Smooth using approximately the same index size.

age, and quality, when considering different radii values in real data. As we noted in our analysis, the advantage of the quality-sensitive approaches is most pronounced when there exists a large amount of low quality items in the dataset. Indeed, in our setting, 73% of the items are assigned a quality value below 0.5. In such cases, using quality-sensitive Stream-LSH is expected to be appealing for similarity-search stream applications.

4.4.4 Dynamic Popularity

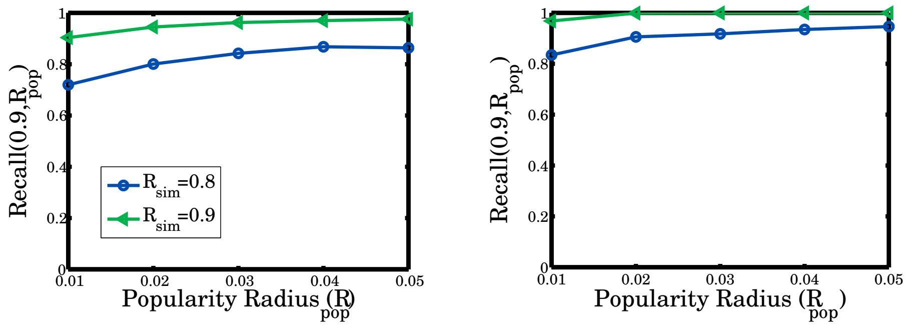
We wrap up by studying Stream-LSH when using DynaPop and the Smooth retention policy. We experiment with $u = 0.95$, $p = 0.95$. As our datasets do not contain temporal interest information, we simulate an interest stream I by considering query results as signals of interests in items [33], as follows: We use the first 75% items in the train set as the item stream U. We construct a query set Q^* by randomly sampling each item from the remaining 25% of the train set with probability 0.1. For each query $q \in Q^*$, we retrieve its top 10 most similar items in U and include them in the interest stream I at q's timestamp t_q, as well as at their original arrival times in U. Table 4.2 summarizes the item and interest stream statistics. We compute popularity scores at the current time according to Definition 4.1.2 with $\alpha = 0.95$.

	Item stream		Interest stream	
	Num. items	Num. ticks	Num. items	Num. ticks
Reuters	540,882	252	226,890	95
Twitter	13,124,853	2,000	4,267,518	1,500

Table 4.2: DynaPop item and interest streams statistics.

Figure 4.10 depicts recall as a function of R_{pop} for similarity radii 0.8 and 0.9. For both datasets and similarity radii, the recall increases as the popularity radius increases. DynaPop provides high recall when searching for the most popular items in the dataset: For example, in the Reuter's dataset (Figure 4.10(a)), for $R_{sim} = 0.8$ (blue curve) and

$R_{pop} = 0.05$ (capturing the 3.5% most popular items in the data set), the recall is 0.86. When increasing the similarity radius to $R_{sim} = 0.9$ (green curve), the recall increases and is 0.97. DynaPop's recall is lower when we also search for less popular items: In the Reuter's dataset, for $R_{sim} = 0.8$ and $R_{pop} = 0.01$ (capturing the 24% most popular items in the data set), the recall is 0.72. When increasing the similarity radius to $R_{sim} = 0.9$, the recall is 0.9. Overall, DynaPop achieves good recall for popular items that are similar to the query while also retrieving similar items that are less popular albeit with lower recall; the latter is beneficial for applications such as query auto-completion [20] and product recommendation [93].



(a) Reuters (b) Twitter

Figure 4.10: Recall by popularity radius of Stream-LSH when using DynaPop with the Smooth retention policy.

4.5 Related Work

Previous work on recommendation over streamed content [45, 38, 61, 56, 60, 62, 28] focused on using temporal information for increasing the relevance of recommended items. Stream recommendation algorithms extend techniques originally designed for static data, such as content-based and collaborative-filtering [14], and apply them to streamed data by taking into account the temporal characteristics of stream generation and consumption within the algorithm internals. In the context of search, many works extend ranking methods to consider temporal aspects of the data, (see [55] for a survey), and quality features such as a social post's length or the author's influence [85]. However, these search and recommendation works do not tackle the challenge of bounding the capacity of their underlying indexing data-structures. Rather, they assume an index of the entire stream with temporal information is given. Our work is thus complementary to these efforts in the sense that we offer a retention policy that may be used within their similarity search building block.

TI [33] and LSII [89] improve realtime indexing of stream data using a policy that determines which items to index online and which to defer to a later batch indexing stage. Both assume unbounded storage and are thus complementary to our work. In addition, the TI focuses on highly popular queries, whereas we also address the tail of the popularity distribution. LSII addresses the tail, however, it assumes exact search while we focus on approximate search, which is the common approach in similarity search [47].

A few previous works have addressed bounding the underlying index size in the context of stream processing [73, 83, 70, 65]. Two papers [73, 83] have focused on *first story detection*, which detects new stories that were not previously seen. Both use LSH as we do. Petrović et al. [73] maintain buckets of similar stories, which are used in realtime for detecting new stories using similarity search. In order to bound the index, they define a limit on the number of stories kept within a bucket, and eliminate the oldest stories when the limit is reached. We call this retention policy Bucket. Sundaram et al. [83]'s primary goal is to parallelize LSH, in order to support high-throughput data streaming. They bound the index size using a retention policy we call Threshold, by eliminating the oldest items when the entire index exceeds a given space limit. Both papers focus on the first story detection application, while our work focuses on the similarity search primitive. Their retention policies are well-suited for first story detection, where the index is searched in order to determine whether any recent matching result exists (indicating that the story is not the first), and are less adequate for similarity search, where multiple relevant results to an arbitrary query are targeted. We evaluate their retention policies in our Stream-LSH algorithm and find that our Smooth policy provides much better results in our context.

Morales and Gionis propose *streaming similarity self-join* (SSSJ) [70], a primitive that finds pairs of similar items within an unbounded data stream. Similarly to us, SSSJ needs to bound its underlying search index. Our work differs however in several aspects: First, we study a different search primitive, namely, similarity search, which searches for items similar to an arbitrary input query rather than retrieving pairs of similar items from the stream. Second, SSSJ only retrieves items that are not older than a given age limit. It thus bounds the index using a variant of Threshold. In contrast, we do not assume that an age limit on all queries is known a priory. In this context, we propose Smooth, which better fits our setting as we show in our evaluation. Third, we tackle approximate similarity search whereas SSSJ searches for an exact set of similar pairs.

Magdy et al. [65] propose a search solution over stream data with bounded storage, which increases the recall of tail queries. Their work differs from ours in the retrieval model, more specifically, they assume the ranking function is static and query-independent, e.g., ranking items by their age. Each item's score is known a priori for all queries, and can be used to decide at indexing time which items to retain in the index. This approach is less suitable to similarity search, where scores are query-dependent and only known at runtime.

In addition, we note that the aforementioned works on bounded-index stream processing [73, 83, 70, 65] do not take into account quality and dynamic popularity as we do.

Several papers have focused on improving the space complexity of LSH via alternative search algorithms [46, 86, 82], via decreasing the number of tables used at the cost of executing more queries [72], or by searching more buckets [64]. Unlike Stream-LSH, these works consider static (finite) data rather than a stream.

4.6 Summary

In this chapter, we presented Stream-LSH, an SSDS algorithm which considers quality, age, and popularity attributes in addition to similarity. For the retention, we proposed Smooth, which gradually removes item copies from the index, rather than removing all item's copies at once when index size limit is exceeded. We showed that Smooth bounds the index size, and that it increases recall compared to other approaches, by better exploiting space capacity. We observed that Smooth poses the following tradeoff: it increases recall when increasing the age radius compared to other approaches, at the cost of decreasing the recall of highly fresh items. Hence, Smooth is less appealing for applications that focus on highly fresh items, but is beneficial when the retrieval of items of varying age is valuable.

Stream-LSH also considers items' quality, and adjusts items' redundancy accordingly. We showed that our quality-sensitive Stream-LSH increases the recall of high-quality items, which is beneficial to social applications that handle content of varying quality. Finally, Stream-LSH dynamically adjusts items' redundancy according to the dynamic interest in them. It achieves good recall for popular items while also retrieving items that are less popular albeit with lower recall. This is beneficial to recommendation applications, which take item's dynamic popularity into account in their recommendation algorithm.

Chapter 5

NearBucket-LSH: Efficient Similarity Search in P2P Networks

Online Social Networks (OSNs) have become popular interaction platforms that serve hundreds of millions of users. In order to meet scale requirements, commercial OSNs are implemented over a distributed cloud infrastructure. An alternative paradigm is a *Peer-to-Peer (P2P)* OSN (e.g., [26, 37, 71, 66]), which offers increased scalability, as well as user privacy, and avoids control by a single authority.

OSN users expose profiles that reflect their sets of *interests*. The interest profile may be provided explicitly by the user, or mined implicitly from her content and activity [69, 95, 77, 13]. User *similarity search* is the task of effectively finding OSN users similar to a user query based on common interests. It is used for many applications including recommending new friends [68, 90], as well as for recommending content based on preferences of similar users [14].

A similarity search algorithm in P2P OSNs faces several challenges: The algorithm should be decentralized in order to fit the P2P architecture. As network cost is a dominant consideration in P2P networks, the algorithm should be network-efficient, while preserving a good search quality. Furthermore, the similarity search should cope with the dynamic nature of OSNs: users join or leave, and users dynamically modify their interest profile. We present a similarity search algorithm in P2P OSNs that meets these requirements.

We base our algorithm on *Locality Sensitive Hashing (LSH)* [47] (see Section 5.2), which is a widespread randomized method for efficient similarity search in high-dimensional spaces. LSH hashes an OSN user (based on her interest profile) into a succinct representation, where the hash values of similar users collide *with high probability (w.h.p.)*. At a pre-processing stage, LSH maps users into collections of objects called *buckets* based on common hashes. Upon receiving a query, LSH limits the search to buckets to which the query is mapped; these contain similar users w.h.p. LSH improves search time complexity at the cost of search quality, as the search is approximate and may miss similar users. We follow here a variant of LSH, called MultiProb-LSH [64], which increases search quality by additionally searching *near buckets*, i.e., buckets similar to the query's bucket.

We present *NearBucket-LSH*, which integrates LSH into a P2P architecture. For our P2P overlay we use *Content Addressable Network (CAN)* [75], which is a good fit for a distributed LSH implementation, as we later show. We use CAN to dynamically map and store LSH buckets within nodes, and refresh bucket contents once in a while in order to adjust to changes in the data. Upon search, we use CAN to locate the buckets to search in.

In P2P settings, searching additional buckets entails contacting additional nodes, which is a network-costly operation. We improve the network-efficiency when searching near buckets by exploiting the internals of CAN: We observe that in CAN, near buckets reside in a bucket's neighboring nodes, and thus contacting them incurs a low network cost. We further eliminate this network cost by caching near buckets in each CAN node.

We analytically study NearBucket-LSH for the cosine similarity metric. We first prove that for any fixed number, k, NearBucket-LSH's choice of k near buckets to search in is optimal. We next compare NearBucket-LSH to LSH, as well as to Layered-LSH [50, 19], a previously suggested LSH variant for distributed systems, which also searches near buckets with the goal of reducing network cost. Our analysis shows that NearBucket-LSH achieves better success probability for a given network cost than the other two approaches. We next provide an empirical evaluation of our algorithm using three real world OSN datasets. Our experiments demonstrate that the cache-based NearBucket-LSH provides the greatest search quality for a given network cost, compared to LSH and Layered-LSH.

5.1 Model and Problem Definition

5.1.1 User Similarity Search in OSN

Each OSN user exposes an *interest profile* (either explicitly or implicitly), which we represent as a non-negative weighted feature vector in a high d-dimensional vector space $V = (\mathbb{R}_0^+)^d$. We use v_i to denote the i-th entry of vector v (corresponding to the i-th interest feature). The interests-weighting scheme may be arbitrary.

We consider an *m-similarity search* algorithm which accepts as an input a *query* vector $q \in V$. It returns a unique *ideal result set* of m user vectors that are most similar to q, according to the given similarity function. An *approximate m-similarity search* algorithm trades-off efficiency with accuracy. Given a query q, it returns an *approximate result set* of m user vectors, which may differ from q's ideal result set.

We use here the cosine similarity function, which is used in the context of similarity between OSN users [16]. The similarity between two vectors $u, v \in V$ is defined as the cosine of the angle between them:

$$sim_{\cos}(u, v) = \frac{u \cdot v}{\|u\| \cdot \|v\|}. \tag{5.1}$$

Yet, LSH was not defined for cosine similarity, but rather, for the closely related angular similarity (Definition 2.1). As the angular and cosine similarities are closely related, we can similarly analyze LSH for cosine similarity [32, 35].

5.1.2 P2P OSN and CAN

P2P networks are distributed systems organized as overlay networks with no central management. Nodes (also called *peers*) are autonomous entities that may join or leave at any time; content is distributed among the participating nodes. P2P networks provide massive scalability, fault tolerance, privacy, anonymity, and load balancing (see [63] for a survey). We consider a P2P Online Social Network [26, 37, 71, 66], in which users' content is distributed among nodes. Any node in the P2P OSN may initiate a similarity search query. Typical OSNs include hundreds of millions of users, and millions of interest features. We consider a dynamic data model, in which users join or leave the OSN and existing users update their interest profiles. We assume the update rate is several orders of magnitude lower than the query rate ($5 - 10$ orders of magnitude, depending on the specific application).

In our algorithm, we use CAN [75] as our overlay, which naturally fits a distributed LSH implementation, as we later show. CAN implements a self-organizing P2P network representing a virtual c-dimensional Cartesian coordinate space on a c-torus. The Cartesian space is dynamically partitioned into *zones*, which are distributed among CAN nodes. CAN implements a *Distributed Hash Table (DHT)* abstraction, which provides a distributed *lookup* operation that accepts a vector as key, and returns a node that owns the zone to which the vector belongs. Each node maintains a table of *neighbors*, which are nodes that own zones adjacent to its own. These tables are used for routing messages within CAN.

5.2 Background and Previous Work

MultiProb-LSH [64] is an extension to LSH (overviewed in Section 2.3), which improves the recall of an LSH algorithm with parameters k and L. MultiProb-LSH observes that *near buckets*, which are buckets that slightly differ from the query's *exact bucket* $g(q)$, have a high probability to contain vectors similar to the query. Searching in such near buckets yields additional similar results w.h.p., which increases LSH's recall. MultiProb-LSH was introduced in the context of the l_p norm; here, we apply its principles to cosine-based LSH, in the context of P2P OSN.

In P2P networks, buckets are distributed over the overlay nodes. Contacting a near bucket involves performing a DHT lookup of its node, which incurs high network cost. Prior art [50, 19] suggests *Layered-LSH*, which maps buckets to nodes using a second LSH, such that near buckets are assigned to the same node w.h.p. Queries now access a single node holding the desired buckets, which reduces the network cost. In Section 5.4.1, we show that in the case of cosine similarity, Layered-LSH is equivalent to the basic LSH for an appropriate choice of k, incurring the same network cost as the basic LSH. In this chapter, we show that our NearBucket-LSH algorithm improves recall for a given network cost compared to LSH and Layered-LSH, and is therefore more network-efficient compared to prior art.

As LSH is a widespread similarity search algorithm, commonly used in real deployments, we focus here on LSH-based methods. In particular, we compare our algorithm to prior approaches that use LSH [50, 19]. We note that other P2P similarity search methods have been proposed [22], in particular, Falchi et al. [43] use CAN as their overlay. However, these methods are not based on LSH, which is the focus of our work.

In addition to distributed solutions, there are also parallel LSH variants, e.g. [84]. However, these do not focus on improving network-efficiency, which is not of essence in a parallel setting.

5.3 Algorithm

Our algorithm is based on locality sensitive hashing, reviewed in Section 5.2 above. In order to implement our P2P user similarity search, we construct a dedicated overlay above the CAN infrastructure. We distribute LSH buckets of user vectors among the overlay nodes, occasionally refresh their content to adjust for changes, and route search queries to the appropriate buckets, as described in Section 5.3.1. In Section 5.3.2, we extend this basic approach to also search in near buckets.

5.3.1 CAN-based LSH

The Overlay We use a k-dimensional CAN (i.e., $c = k$) to store and lookup LSH buckets in a decentralized manner. For simplicity, we assume that $N = 2^k$, where N denotes the number of CAN nodes. Note that our overlay may be formed by a subset of the OSN nodes, but for simplicity of the description, we assume all OSN nodes participate in the overlay. Each CAN node owns the zone of a single k-dimensional binary vector v representing some LSH sketch vector, and maintains the bucket of user vectors that are mapped to v by some hash function $g \in \mathcal{G}$. We name such a node the *bucket node* of v. The bucket node provides a local similarity search facility over its locally stored user vectors. The local search time is typically proportional to the searched bucket size [47]. The internal bucket data-structure and local search implementation are orthogonal to this research.

Each CAN node in our overlay has k neighbors; the i-th neighbor of node v owns a vector u that differs from v in the i-th entry only. Routing a message from node v to one of its neighbors requires a single hop, i.e., a single message. Routing a message from an arbitrary source node v to an arbitrary target node u, entails modifying the binary vector entries that differ between u and v. Two vectors of length k, differ in $k/2$ entries, and thus, the expected path length is $k/2$ hops[1].

The L hash functions $g = \{g_1, \cdots, g_L\}$ are randomly selected from $\mathcal{G}$ a priori. They are given to the distributed algorithm as a configuration parameter, and are known to all bucket nodes. CAN supports multiple hash functions [75], which we use for supporting multiple g_i's and mapping each user vector into L bucket nodes.

Bucket Maintenance Our algorithm constructs and refreshes the buckets continuously, in a decentralized manner. Thus, each bucket node stores soft state that is regularly refreshed. Each user periodically re-hashes its vector using LSH into L sketch vectors in $\{1, 0\}^k$. It then performs DHT lookups to locate the corresponding bucket nodes, and sends them the fresh user vector. Note that the user vector may or may not have changed since the previous update message.

We do not construct buckets a priori. Rather, bucket construction is triggered by vector update messages. A CAN node becomes an active bucket node when it first receives a notification of some user vector. Since user vectors change dynamically, their hashes change accordingly. Obsolete vectors that are not refreshed for a certain predefined length of time are garbage-collected from bucket nodes.

Query Processing Each P2P node may trigger an m-similarity search request for an input query q. The similarity search follows the LSH algorithm [47], using our overlay. The initiating node, denoted n, activates the function Query in Algorithm 5: It hashes q into L sketch vectors according to the pre-defined g_i functions, looks-up L corresponding

[1] Note that in a general c-dimensional CAN of N nodes, the expected routing length is $c/4 \left(N^{1/c}\right)$ [75], which equals $k/2$ for $c = k$ and $N = 2^k$.

bucket nodes n_i using CAN, and sends m-similarity search requests with the input query q to all L bucket nodes in parallel. Each bucket node locally performs an m-similarity search (function SIMSEARCH in Algorithm 5), and sends back a set of up to m results, associated with their similarity values. Node n receives L result sets, which it merges and sorts according to the similarity values. It then returns a final m-result set to the caller.

Algorithm 5 Distributed LSH Algorithm

```
1:  function QUERY($q$)                                   ▷ At the query node
2:      pforeach $g_i \in g$ do                           ▷ A parallel foreach
3:          $v_i \leftarrow g_i(q)$
4:          $n_i \leftarrow DHT$.LOOKUP($v_i$)            ▷ Lookup bucket node
5:          $n_i$.SENDREQ($SimSearch, q, n$)              ▷ Send request
6:      end pforeach
7:      $hits \leftarrow$ collect results from bucket nodes
8:      return top $m$ hits                               ▷ Rank and return top $m$
9:  end function

10: function SIMSEARCH($q, n$)                            ▷ Query $q$ from $n$
11:     $res \leftarrow Bucket$.LOCALSIMSEARCH($q$)       ▷ Local search
12:     $n$.SENDRES($res$)                                ▷ Send back result
13: end function
```

5.3.2 NearBucket-LSH

Given a query q and some hash function $g \in \mathcal{G}$, the basic LSH algorithm searches in the exact bucket $g(q)$. NearBucket-LSH extends LSH to also search in near buckets that differ from $g(q)$ in exactly one vector entry, i.e., one bit is flipped. As we analytically show in Section 5.4, searching in near buckets increases the probability to find similar users.

Contacting a neighbor costs a single message, for a total of kL messages per query. We further eliminate these additional messages by caching k near buckets at each CAN node. In order to maintain fresh caches, each node periodically sends its bucket to its neighbors. The cache requires an additional storage of size kB at each node, where B is the average bucket size. Note that our cache is only used for storing near buckets.

A CAN node maintains a table of k neighbors that differ from it in exactly one entry, which are also the neighbors that hold the desired near buckets. Given a query q, NearBucket-LSH uses a query function similar to the function QUERY in Algorithm 5: to contact the L exact bucket nodes using CAN. But here, the sent request is SimSearchNB. Once such a request reaches some exact bucket node, it activates the function SIMSEARCHNB in Algorithm 6: The node first performs a local similarity search in its own bucket (line 2 in Algorithm 6). Then for each of its k neighbor nodes n_j,

$j = \{1, \cdots, k\}$, it checks if that node's bucket is cached locally. If it is, it searches it, and if not, it forwards the query to that node. In case messages are forwarded, bucket nodes perform local m-similarity searches of query q in parallel, and each returns a result set to the initiating node n.

Algorithm 6 Distributed NearBucket-LSH Algorithm

```
1:  function SIMSEARCHNB(q, n)                              ▷ Query $q$ from $n$
2:      $res \leftarrow Bucket$.LOCALSIMSEARCH($q$)             ▷ Local search
3:      $n$.SENDRES($res$)                                   ▷ Send back result
4:      pforeach $j \in \{1, \cdots, k\}$ do                  ▷ A parallel foreach
5:          $n_j \leftarrow Neighbors.j$                      ▷ Extract the $j$-th neighbor
6:          if $Bucket_j.isCached$ then
7:              $res \leftarrow Bucket_j$.LOCALSIMSEARCH($q$)   ▷ Local search
8:              $n$.SENDRES($res$)                           ▷ Send back result
9:          else
10:             $n_j$.SENDREQ($SimSearch, q, n$)             ▷ Forward request
11:         end if
12:     end pforeach
13: end function
```

It is possible to cache all k near buckets or any subset of them. For the purpose of the analysis and evaluation in the next sections, we refer to the following two extremes: we name NB-LSH a NearBucket-LSH that does not use caching at all, and CNB-LSH a NearBucket-LSH that caches all k near buckets.

5.4 Analysis

We theoretically analyze an algorithm's capability of retrieving similar objects, and show the superiority of NearBucket-LSH to successfully retrieve similar objects for a given network cost.

5.4.1 Success Probability Formulation

The basic building block in our analysis is the *success probability* [64] of an algorithm A to find object y that has a similarity value s to query object q, under a random selection of $g \in \mathcal{G}$. We denote this success probability by $SP(A, s)$.

LSH. Let $LSH(k, L)$ denote the angular-LSH algorithm with parameters k and L, and let s denote the angular similarity between query q and searched object y. According to the LSH theory [32], for a randomly selected $h \in \mathcal{H}$:

$$Pr_{h\in\mathcal{H}}\left[h(q) = h(y)\right] = s, \text{ and } Pr_{h\in\mathcal{H}}\left[h(q) \neq h(y)\right] = (1 - s). \tag{5.2}$$

$LSH(k, L)$ searches in L exact buckets independently, thus, it finds y in any of these buckets with probability:

Proposition 3.

$$SP(LSH(k, L), s) = 1 - \left(1 - s^k\right)^L.$$

NearBucket-LSH. We define *b-near buckets* to be buckets that differ from an exact bucket in $0 \leq b \leq k$ entries (note that a 0-near bucket is an exact bucket). The success probability of finding y in a b-near bucket of $g(q)$ is:

$$s^{k-b}(1-s)^b. \tag{5.3}$$

As our vectors are non-negative, their angular similarities s satisfy that $s \in [0.5, 1]$. This implies that $\forall s, (1-s) \leq s$, and therefore, for $0 \leq b_1 < b_2 \leq k$, $s^{k-b_2}(1-s)^{b_2} \leq s^{k-b_1}(1-s)^{b_1}$, thus:

Proposition 4. *The success probability when searching in a b_1-near bucket is greater or equal to the success probability when searching in a b_2-near bucket, for any $0 \leq b_1 < b_2 \leq k$. Hence, NearBucket-LSH's selection of k 1-near buckets is optimal, with respect to any other k buckets selected for search, in addition to the exact bucket.*

The exact bucket and its near buckets are disjoint, as an object is mapped to exactly one bucket according to a specific g. NearBucket-LSH searches in L exact buckets each along with its k 1-near buckets. Thus,

Proposition 5.

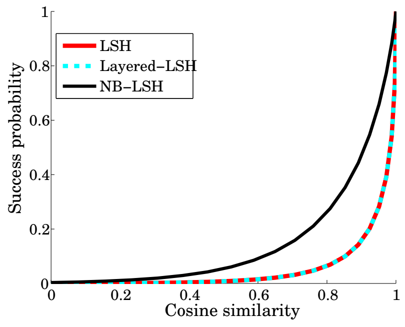
$$SP(\textit{NearBucket-LSH}(k, L), s) = 1 - (1 - (s^k + ks^{k-1}(1-s)))^L.$$

Layered-LSH. We show that for the angular similarity, Layered-LSH is equivalent to the basic LSH. Layered-LSH maps near buckets to the same node w.h.p., which can be achieved by using Hamming-based LSH [47, 35] as follows. Let g_{ang} be the angular-LSH used for mapping vectors to buckets. By definition, g_{ang} is a concatenation of h_i angular-LSH functions. Let g_{ham} be the Hamming-LSH used for mapping buckets to nodes. Hamming-based LSH hashes a binary vector to another binary vector of a lower dimension k, by randomly and independently selecting k entries of the input vector. In our case, this resorts to randomly and independently selecting k entries from $g_{ang}(v)$, each of which corresponds to some $h_i \in \mathcal{H}$. We get that $g_{ham}(g_{ang}(v))$ maps v to a node according to k randomly selected $h \in \mathcal{H}$ functions, which is equivalent to using the angular-LSH with parameter k.

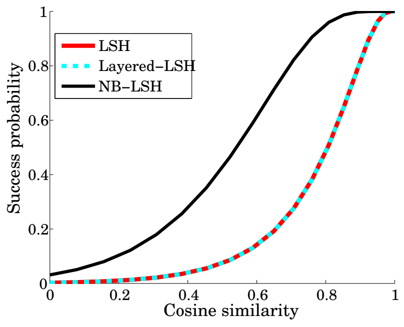
5.4.2 Success Probability Comparison

We use Propositions 3 and 5 to compare the success probabilities of LSH, Layered-LSH, and NearBucket-LSH. We compute an algorithm's success probability as a function of the cosine similarity between the query and the searched object[2]. As Layered-LSH is equivalent to LSH, we refer to both as LSH in this discussion. For the purpose of the demonstration, we present graphs for selected k and L values. Note however that we observed the same trend for other k and L values; we omit the respective graphs from this text.

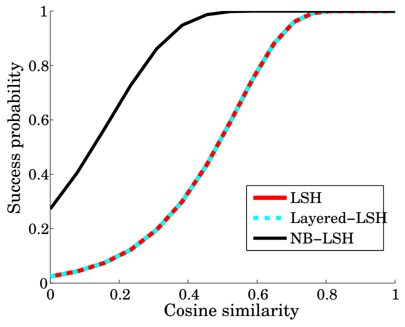
Constant Number of Hash Functions. We first examine the effect of searching near buckets on the success probability, hence, we compare LSH and NearBucket-LSH for a constant L. Figure 5.1 depicts their success probabilities for $k = 12$ and for increasing L values of 1, 10, and 100. As the graphs demonstrate, the success probability of NearBucket-LSH is greater than or equal to the success probability of LSH for all similarities, for a constant L.



(a) $L = 1$



(b) $L = 10$



(c) $L = 100$

Figure 5.1: Analytical success probability as a function of L ($k = 12$). NearBucket-LSH guarantees a greater or equal success probability compared to LSH and Layered-LSH, as it searches in more buckets (namely, near buckets). The gap increases as L increases.

Network Efficiency. As we have seen, for a constant L, NearBucket-LSH increases the success probability of LSH at the cost of contacting additional buckets. We proceed to analyzing the success probability as a function of the network cost. We measure the

[2]We transform cosine similarity into angular similarity and then apply the success probability formulas.

network cost by the average number of messages per query. We distinguish between the cached (CNB-LSH) and non-cached (NB-LSH) versions of NearBucket-LSH.

The first column of Table 5.1 summarizes the number of bucket nodes contacted (and searched) by each of the algorithms, for given k and L parameters. Looking up an exact bucket node requires an average of $k/2$ routing hops, and contacting a neighbor node costs one message. The second column in Table 5.1 summarizes the average number of messages per query, for given k and L parameters.

	Number of nodes contacted per query	Average number of messages per query	Number of vectors stored in a node	Number of vectors searched per query
LSH	L	$\frac{1}{2}kL$	B	LB
Layered-LSH	L	$\frac{1}{2}kL$	B	LB
NB-LSH	L(1+k)	$1\frac{1}{2}kL$	B	$L(k+1)B$
CNB-LSH	L	$\frac{1}{2}kL$	$(k+1)B$	$L(k+1)B$

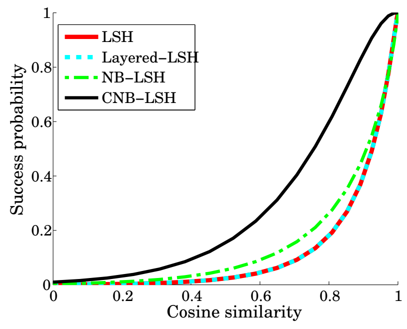
Table 5.1: Summary of costs of similarity search in CAN-based LSH variants for given k, L LSH parameters.

Figure 5.2 depicts success probability for $k = 12$ and an increasing network costs of 18, 180, and 1800 average number of messages. The graphs illustrate that, thanks to the low network cost of searching near buckets, NearBucket-LSH, (and more notably CNB-LSH), improves LSH's success probability for all similarity values, for a constant average number of messages. Note that one could further extend NearBucket-LSH to search in near buckets that differ from the query's bucket in more than one entry. The success probability of such buckets decreases (Proposition 4), whereas the network cost in NB-LSH and the storage cost in CNB-LSH increases compared to 1-near buckets. Thus, searching additional buckets is expected to be less effective.

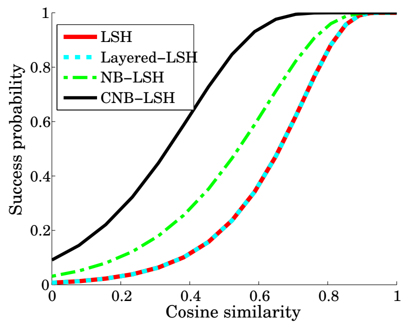
Other Considerations. Our work focuses on minimizing the network cost, which is a dominant cost in P2P networks. For completeness, we present in the third and fourth columns of Table 5.1 other costs which tradeoff with network-efficiency. We denote the average bucket size by B. In terms of storage capacity, NB-LSH preserves the same space complexity as LSH and Layered-LSH. CNB-LSH increases the space complexity due to caching, while being more network-efficient than NB-LSH. Both NearBucket-LSH variants search over a larger number of vectors than LSH, implying more processing work per query. As our algorithm searches the buckets in parallel, and the average bucket size is equal in all algorithms, this does not affect the query latency.

5.5 Evaluation

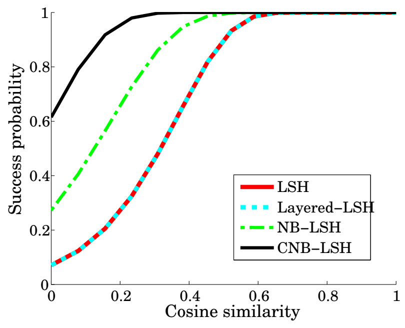
We empirically evaluate our algorithm on three real world OSN datasets of varying sizes, and demonstrate the superiority of CNB-LSH over other approaches.



(a) network cost: 18



(b) network cost: 180



(c) network cost: 1800

Figure 5.2: Analytical success probability as a function of network cost for $k = 12$. NB-LSH exploits the low lookup cost of near buckets in CAN, and increases LSH's and Layered-LSH's success probability for a given network cost. CNB-LSH further saves messages by caching near buckets, and achieves the greatest success probability for a given network cost.

5.5.1 Methodology

Datasets. We use three real-world publically-available datasets of OSNs [92]:

- *DBLP* [39], the computer science bibliography database: Authors are users, and venues are interests. We use a crawl of 13,477 interests, and 260,998 users that have at least one interest.
- *LiveJournal* [3] blogging-based OSN: Users publish blogs and form interest groups, which users can join. The LiveJournal crawl consists of 664,414 such groups, which we consider as user interests. There are 1,147,948 users with at least one interest.
- *Friendster* [44] online gaming network: Similarly to LiveJournal, Friendster allows users to form interest groups, which we consider as interests. The dataset consists of 1,620,991 interest groups, and 7,944,949 users with at least one interest.

All datasets contain anonymous user ids and interest information. We filtered out users having no interest.

Parameters. We set $k = 10$ in DBLP, $k = 12$ in LiveJournal and $k = 15$ in Friendster. We follow previous art [19, 50] that uses k values between 10 and 20, and bucket sizes of a few hundreds [47]. Thus, we have 1,024 buckets in DBLP, 4,096 in LiveJournal, and 32,768 in Friendster. The average bucket size is approximately 250 vectors in all datasets. We set m, the number of search results, to 10.

Creating Sketch Vectors. We construct users' weighted interest vectors according to the dataset at hand. We weight each interest I based on its inverse frequency in user vectors [13]: $w(I) = ln(\frac{N_u}{N_I+1}) + 1$, where N_u denotes the total number of users, and N_I denotes the number of users having interest I. The user vector entry v_i is zero or $w(I)$ according to whether the user is associated with specific interest I. We use TarsosLSH's [10] for mapping vectors into LSH buckets.

Simulator. We implement a simulator of our CAN-based overlay using Apache Lucene 4.3.0 [4] centralized search index. We simulate distributing user vectors in bucket nodes by indexing vectors by their hash values (sketch vectors). The hash is then used for looking up a specific bucket node, and local similarity search is performed by limiting the search to the selected bucket (using Lucene's Filter mechanism). We additionally use Lucene to compute the ideal result set of a given query, by executing the query over the whole dataset. We score results according to the cosine similarity.

Evaluation Set. We construct a query set of 3,000 randomly sampled users. For each query q, we retrieve its ideal result set, as well as the result sets according to the algorithms we compare. For each dataset, we measure recall and precision over the query set in use.

Recall. (at m) is defined as follows [64]:

Definition 5.5.1 (recall at m)**.** *Given a query q, let $I_m(q)$ denote its ideal m-result set. Let $A_m(q)$ denote the approximate m-result set of q returned by some algorithm A. An algorithm's recall for query q is the fraction of results from q's m-ideal result set that are returned by A:*

$$recall@m(A, q) = \frac{|A_m(q) \cap I_m(q)|}{|I_m(q)|}. \tag{5.4}$$

An algorithm's recall, recall@m (A), is the mean of the queries' recall averaged over a query set Q.

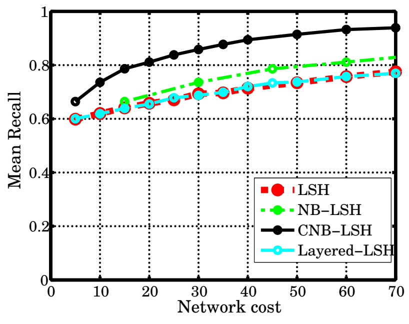
5.5.2 Search Quality Results

Figure 5.3 illustrates our experimental results as a function of network cost. As in Section 5.4.2, we measure the network cost by the average number of messages per query according to Table 5.1. We increase the network cost by gradually increasing L, which increases search quality for all datasets as expected. We use larger values of k for larger datasets in order to preserve a common average bucket size. This ensures that local search takes the same time, and the cache sizes are identical. The larger k is, the lower the success probability is, thus, we expect a decrease in search quality when the dataset size increases, which is indeed demonstrated in the graphs.

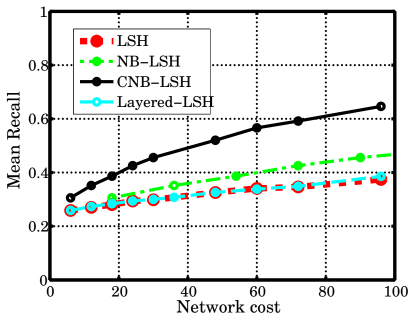
The three datasets show a similar trend. Layered-LSH's recall equals that of the basic LSH as expected. NearBucket-LSH (both cached and non-cached) demonstrates an increase in recall compared to LSH and Layered-LSH, which is achieved by searching in additional near buckets stored at neighboring nodes or the node itself. CNB-LSH improves recall significantly, for example in LiveJournal, it achieves a 0.59 recall using 72 queries, compared to a recall of 0.35 for LSH. In all cases, NB-LSH is between LSH and CNB-LSH.

5.6 Summary

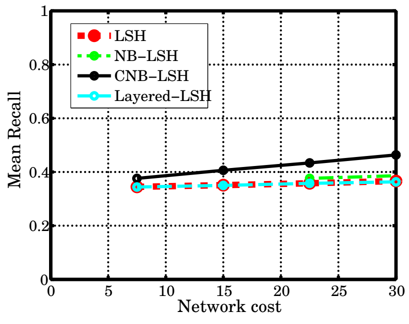
In this chapter, we presented a network-efficient, LSH-based, similarity search algorithm in P2P OSNs. We showed that in P2P OSNs, searching near buckets increases recall at the cost of sending additional messages over the network. We presented NearBucket-LSH, which uses CAN as its overlay, and exploits CAN's architecture for decreasing the network cost. Our experiments demonstrate that NearBucket-LSH achieves a higher recall compared to the basic LSH for a given network cost. As our algorithm is tailored to CAN, its drawback is that it is restricted to using CAN as its overlay. We additionally proposed a cached version of NearBucket-LSH, which further decreases the network cost by caching near-buckets at neighboring nodes, at the cost of increasing storage capacity. Overall, NearBucket-LSH is beneficial when reducing the network cost is of top consideration, which is a typical case in P2P networks.



(a) DBLP



(b) LiveJournal



(c) Friendster

Figure 5.3: Recall as a function of the average number of messages per query, for three real world datasets: DBLP, LiveJournal, and Friendster ($k = 10$, $k = 12$, $k = 15$, respectively). For all datasets, CNB-LSH provides the greatest recall as a function of the network cost.

Chapter 6

Conclusion

In this thesis, we explored search effectiveness in DiS, focusing on the tradeoff between search quality and other considerations. We improved search quality in three scenarios that we examined, by better exploiting the underlying system's resources. It would be of interest for future work to examine other scenarios in which this tradeoff arises, and further improve the search quality of DiS.

We studied tail-tolerant DiS, which is crucial for real-world commercial search engines. We observed that tail-tolerant DiS is amenable to a non-binary availability model based on degradation in search quality. We showed that in this context, Replication is not ideal for mitigating result misses, as searching exact shard copies can be wasteful. We introduced two strategies that better fit tail-tolerant DiS. First, we proposed to consider miss probability as well as each shard's probability to satisfy the query for selecting shards. We devised rSmartRed, an optimal shard selection scheme for Replication. Second, we proposed Repartition, an alternative approach for applying redundancy. Repartition constructs independent index partitions instead of exact copies, which improves search quality over Replication in practical scenarios.

Our work considers a static index during query processing. It would be interesting for future work to explore tail-tolerance at indexing stage when using a dynamic index. It would also be of interest to consider tail-tolerance when assuming other DiS models, including other index partitioning approaches, or DiS over multiple, distant datacenters. Finally, it would be of interest to consider redundancy approaches that are more space-compact than Replication.

We introduced the problem of similarity search over endless data-streams, which faces the challenge of indexing unbounded data. We proposed Stream-LSH, an SSDS algorithm that uses a retention policy to bound the index size. We showed that our Smooth retention policy increases recall of similar items compared to methods proposed by prior art. In addition, our Stream-LSH indexing procedure is quality-sensitive, and is extensible to dynamically retain items according to their popularity.

While our work focuses on similarity search, our approach may prove useful in future work, for addressing space constraints in other stream-based search and recommendation primitives. Our work considers similarity, age, quality, and dynamic popularity attributes. It may be of interest for future work to consider additional attributes and complex relations among them.

We presented NearBucket-LSH, a network-efficient LSH algorithm for P2P OSNs, which provides good search quality. We first analytically showed that, for cosine similarity, our choice of searched near buckets is optimal, that is, near buckets that differ in a single entry from the query's bucket are more likely to contain similar vectors than other near buckets. We then showed that one may dramatically lower the additional network cost for searching in these buckets by exploiting CAN's internal structure and judicious caching.

Our proposed overlay focuses on angular-LSH, which fits OSN similarity search. It would be of an interest to extend our overlay to support other LSH families such as l_p-LSH.

Bibliography

[1] Announcing replicated elasticsearch clusters on aws. https://qbox.io/blog/announcing-replicated-elasticsearch-clusters.

[2] Google wants a 200 ms page load time?! https://www.searchenginenews.com/sample/update/entry/you-have-200-ms-to-load-your-page.

[3] Livejournal. http://www.livejournal.com/.

[4] Lucene. http://lucene.apache.org/core/.

[5] A provider-side view of web search response time. https://www.microsoft.com/en-us/research/publication/a-provider-side-view-of-web-search-response-time/.

[6] Reuters rcv1. http://www.daviddlewis.com/resources/testcollections/rcv1/.

[7] Reuters trec topics. http://trec.nist.gov/data/filtering/T11filter_T2002-filt-topics.txt.

[8] Solr index replication. https://cwiki.apache.org/confluence/display/solr/Index+Replication.

[9] Stanford snap project. http://snap.stanford.edu/.

[10] Tarsos-lsh. https://github.com/JorenSix/TarsosLSH.

[11] The top 20 valuable facebook statistics. https://zephoria.com/top-15-valuable-facebook-statistics/.

[12] Twitter nasdaq. http://followthehashtag.com/datasets/nasdaq-100-companies-free-twitter-dataset/.

[13] L. A. Adamic and E. Adar. Friends and neighbors on the web. *SOCIAL NETWORKS*, 25:211–230, 2001.

[14] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE transactions on knowledge and data engineering*, 17(6):734–749, 2005.

[15] E. Agichtein, C. Castillo, D. Donato, A. Gionis, and G. Mishne. Finding high-quality content in social media. WSDM '08, pages 183–194, 2008.

[16] A. Anderson, D. Huttenlocher, J. Kleinberg, and J. Leskovec. Effects of user similarity in social media. WSDM '12, pages 703–712, 2012.

[17] A. Andoni. *Nearest Neighbor Search: the Old, the New, and the Impossible*. PhD thesis, Massachusetts Institute of Technology, 2009.

[18] H. Attiya, A. Bar-Noy, and D. Dolev. Distributed computing: Fundamentals, simulations, and advanced topics. In *2nd Edition*, 2004.

[19] B. Bahmani, A. Goel, and R. Shinde. Efficient distributed locality sensitive hashing. In *CIKM '12*, pages 2174–2178, 2012.

[20] Z. Bar-Yossef and N. Kraus. Context-sensitive query auto-completion. WWW '11, pages 107–116, 2011.

[21] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[22] M. Batko, D. Novak, F. Falchi, and P. Zezula. Scalability comparison of peer-to-peer similarity search structures. *Future Generation Comp. Syst.*, pages 834–848, 2008.

[23] H. Becker, M. Naaman, and L. Gravano. Selecting quality twitter content for events. ICWSM11, 2011.

[24] F. R. Bentley, J. J. Kaye, D. A. Shamma, and J. A. Guerra-Gomez. The 32 days of christmas: Understanding temporal intent in image search queries. CHI '16, pages 5710–5714, 2016.

[25] J. Brutlag. Speed matters for google web search. `http://www.isaacsunyer.com/wp-content/uploads/2009/09/test_velocidad_google.pdf`, 2009.

[26] S. Buchegger, D. Schiöberg, L. H. Vu, and A. Datta. PeerSoN: P2P social networking - early experiences and insights. In *SNS '09*, pages 46–52, March 31, 2009.

[27] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin. Earlybird: Real-time search at twitter. ICDE '12, pages 1360–1369, 2012.

[28] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin. Earlybird: Real-time search at twitter. ICDE '12, pages 1360–1369, 2012.

[29] J. Callan. Distributed information retrieval. In *Advances in Information Retrieval*, pages 127–150. Kluwer Academic Publishers, 2000.

[30] B. B. Cambazoglu and R. Baeza-Yates. *Advanced Topics in Information Retrieval*, chapter Scalability Challenges in Web Search Engines, pages 27–50. 2011.

[31] R. Campos, G. Dias, A. M. Jorge, and A. Jatowt. Survey of temporal information retrieval and related applications. *ACM Comput. Surv.*, pages 15:1–15:41, 2014.

[32] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC '02*, pages 380–388, 2002.

[33] C. Chen, F. Li, B. C. Ooi, and S. Wu. Ti: An efficient indexing mechanism for real-time search on tweets. SIGMOD '11, pages 649–660, 2011.

[34] J. Chen, R. Nairn, and E. Chi. Speak little and well: Recommending conversations in online social streams. CHI '11, pages 217–226, 2011.

[35] F. Chierichetti and R. Kumar. LSH-preserving functions and their applications. In *SODA '12*, pages 1078–1094, 2012.

[36] M. Curtiss, I. Becker, T. Bosman, S. Doroshenko, L. Grijincu, T. Jackson, S. Kunnatur, S. Lassen, P. Pronin, S. Sankar, G. Shen, G. Woss, C. Yang, and N. Zhang. Unicorn: A system for searching the social graph. *Proc. VLDB Endow.*, pages 1150–1161, 2013.

[37] L. A. Cutillo, R. Molva, and M. Önen. Safebook: A distributed privacy preserving online social network. In *WOWMOM*, pages 1–3, 2011.

[38] A. S. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: Scalable online collaborative filtering. WWW '07, pages 271–280, 2007.

[39] DBLP. http://www.informatik.uni-trier.de/ ley/db/.

[40] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013.

[41] F. Diaz. Integration of news content into web results. WSDM '09, pages 182–191, 2009.

[42] S. Ding, S. Gollapudi, S. Ieong, K. Kenthapadi, and A. Ntoulas. Indexing strategies for graceful degradation of search quality. SIGIR '11, pages 575–584, 2011.

[43] F. Falchi, C. Gennaro, and P. Zezula. A content-addressable network for similarity search in metric spaces. In *DBISP2P'05*, pages 98–110, 2005.

[44] Friendster. http://www.friendster.com/.

[45] E. Gabrilovich, S. Dumais, and E. Horvitz. Newsjunkie: Providing personalized newsfeeds via analysis of information novelty. WWW '04, pages 482–490, 2004.

[46] J. Gan, J. Feng, Q. Fang, and W. Ng. Locality-sensitive hashing scheme based on dynamic collision counting. SIGMOD '12, pages 541–552, 2012.

[47] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB '99*, pages 518–529, 1999.

[48] C. Gómez-Pantoja, M. Marín, V. G. Costa, and C. Bonacic. An evaluation of fault-tolerant query processing for web search engines. In *Euro-Par2011*, pages 393–404, 2011.

[49] I. Guy, T. Steier, M. Barnea, I. Ronen, and T. Daniel. Swimming against the streamz: Search and analytics over the enterprise activity stream. CIKM '12, pages 1587–1591, 2012.

[50] P. Haghani, S. Michel, and K. Aberer. Distributed similarity search in high dimensions using locality sensitive hashing. In *EDBT '09*, pages 744–755, 2009.

[51] Y. He, S. Elnikety, J. Larus, and C. Yan. Zeta: Scheduling interactive services with partial execution. SoCC '12, pages 1–14, 2012.

[52] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. STOC '98, pages 604–613, 1998.

[53] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up distributed request-response workflows. SIGCOMM '13, pages 219–230, 2013.

[54] F. P. Junqueira, V. Leroy, and M. Morel. Reactive index replication for distributed search engines. SIGIR '12, pages 831–840, 2012.

[55] N. Kanhabua, R. Blanco, and K. Nørvåg. Temporal information retrieval. *Foundations and Trends in Information Retrieval*, pages 91–208, 2015.

[56] M. Kompan and M. Bielikova. Content-based news recommendation. In *E-Commerce and Web Technologies*, pages 61–72. 2010.

[57] A. Kulkarni and J. Callan. Selective search: Efficient and effective search of large textual collections. *ACM Trans. Inf. Syst.*, 33(4):1–33, 2015.

[58] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? WWW '10, pages 591–600, 2010.

[59] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of Massive Datasets, 2nd Ed*. Cambridge University Press, 2014.

[60] L. Li, D. Wang, T. Li, D. Knox, and B. Padmanabhan. Scene: A scalable two-stage personalized news recommendation system. SIGIR '11, pages 125–134, 2011.

[61] J. Liu, P. Dolan, and E. R. Pedersen. Personalized news recommendation based on click behavior. IUI '10, pages 31–40, 2010.

[62] M. Lu, Z. Qin, Y. Cao, Z. Liu, and M. Wang. Scalable news recommendation using multi-dimensional similarity and jaccard-kmeans clustering. *J. Syst. Softw.*, pages 242–251, 2014.

[63] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys and Tutorials*, 7:72–93, 2005.

[64] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: Efficient indexing for high-dimensional similarity search. In *VLDB '07*, pages 950–961, 2007.

[65] A. Magdy, R. Alghamdi, and M. F. Mokbel. On main-memory flushing in microblogs data management systems. ICDE '16, pages 445–456, 2016.

[66] M. Mani, A.-M. Nguyen, and N. Crespi. Scope: A prototype for spontaneous p2p social networking. In *PerCom Workshops*, pages 220–225, 2010.

[67] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. 2008.

[68] M. McPherson, L. Smith-Lovin, and J. M. Cook. Birds of a feather: Homophily in social networks. *Annual Review of Sociology*, pages 415–444, 2001.

[69] A. Mislove, B. Viswanath, K. P. Gummadi, and P. Druschel. You are who you know: inferring user profiles in online social networks. WSDM '10, pages 251–260, 2010.

[70] G. D. F. Morales and A. Gionis. Streaming similarity self-join. *PVLDB*, 9(10):792–803, 2016.

[71] R. Narendula, T. G. Papaioannou, and K. Aberer. Towards the realization of decentralized online social networks: An empirical study. In *ICDCS Workshops*, pages 155–162, 2012.

[72] R. Panigrahy. Entropy based nearest neighbor search in high dimensions. In *SODA '06*, pages 1186–1195, 2006.

[73] S. Petrović, M. Osborne, and V. Lavrenko. Streaming first story detection with application to twitter. HLT '10, pages 181–189, 2010.

[74] D. Puppin, F. Silvestri, R. Perego, and R. Baeza-Yates. Tuning the capacity of search engines: Load-driven routing and incremental caching to reduce and balance the load. *ACM Trans. Inf. Syst.*, 28(2):5:1–5:36, 2010.

[75] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM '01*, pages 161–172, New York, NY, USA, 2001.

[76] K. M. Risvik, T. Chilimbi, H. Tan, K. Kalyanaraman, and C. Anderson. Maguro, a system for indexing and searching over very large text collections. WSDM '13, pages 727–736, 2013.

[77] N. K. Sharma, S. Ghosh, F. Benevenuto, N. Ganguly, and K. Gummadi. Inferring who-is-who in the twitter social network. WOSN '12, pages 55–60, 2012.

[78] M. Shokouhi. Central-rank-based collection selection in uncooperative distributed information retrieval. In *Proceedings of the 29th European Conference on IR Research*, ECIR'07, pages 160–172, 2007.

[79] L. Si and J. Callan. Relevant document distribution estimation method for resource selection. SIGIR '03, pages 298–305, 2003.

[80] M. Slaney and M. Casey. Locality-sensitive hashing for finding nearest neighbors. *Signal Processing Magazine, IEEE*, pages 128–131, 2008.

[81] S. Souders. Velocity and the bottom line. http://radar.oreilly.com/2009/07/velocity-making-your-site-fast.html, 2009.

[82] Y. Sun, W. Wang, J. Qin, Y. Zhang, and X. Lin. Srs: Solving c-approximate nearest neighbor queries in high dimensional euclidean space with a tiny index. *Proc. VLDB Endow.*, pages 1–12, 2014.

[83] N. Sundaram, A. Turmukhametova, N. Satish, T. Mostak, P. Indyk, S. Madden, and P. Dubey. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *Proc. VLDB Endow.*, 6(14):1930–1941, Sept. 2013.

[84] N. Sundaram, A. Turmukhametova, N. Satish, T. Mostak, P. Indyk, S. Madden, and P. Dubey. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *Proc. VLDB Endow.*, pages 1930–1941, 2013.

[85] K. Tao, F. Abel, C. Hauff, and G.-J. Houben. Twinder: A search engine for twitter streams. ICWE'12, pages 153–168, 2012.

[86] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. *ACM Trans. Database Syst.*, pages 20:1–20:46, 2010.

[87] J. Teevan, D. Ramage, and M. R. Morris. #twittersearch: A comparison of microblog search and web search. WSDM '11, pages 35–44, 2011.

[88] A. Vulimiri, O. Michel, P. B. Godfrey, and S. Shenker. More is less: Reducing latency via redundancy. HotNets-12, pages 13–18, 2012.

[89] L. Wu, W. Lin, X. Xiao, and Y. Xu. LSII: an indexing structure for exact real-time search on microblogs. ICDE '12, pages 482–493, 2013.

[90] R. Xiang, J. Neville, and M. Rogati. Modeling relationship strength in online social networks. WWW '10, pages 981–990, 2010.

[91] J. Yang and J. Leskovec. Patterns of temporal variation in online media. WSDM '11, pages 177–186, 2011.

[92] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. In *MDS '12*, pages 3:1–3:8, 2012.

[93] H. Yin, B. Cui, J. Li, J. Yao, and C. Chen. Challenging the long tail recommendation. *Proc. VLDB Endow.*, pages 896–907, 2012.

[94] J.-M. Yun, Y. He, S. Elnikety, and S. Ren. Optimal aggregation policy for reducing tail latency of web search. SIGIR '15, pages 63–72, 2015.

[95] E. Zheleva and L. Getoor. To join or not to join: the illusion of privacy in social networks with mixed public and private user profiles. In *WWW*, pages 531–540, 2009.

Hebrew Section

במחקר זה אנו עוסקים במספר איזונים (tradeoffs) מרכזיים בין משאבי המערכת הזמינים לאיכות החיפוש שהמערכת מספקת. אנו עוסקים בשלושה תחומים, ובפרט, בפגיעה באיכות החיפוש בשל שרתים איטיים, מגבלות של קיבולת מקום האחסון ורחב התקשורת. אנו משפרים את איכות החיפוש על ידי שימוש מושכל במשאבי המערכת במקרים אלו אנו מודדים את איכות החיפוש של האלגוריתמים בהם אנו עוסקים על ידי שני מדדים המקובלים בספרות:

Success Probability – מבטא באופן אנאליטי את ההסתברות שאלגוריתם חיפוש מקורב מאחזר מסמך מסוים הרלבנטי לשאילתה נתונה.

Recall – מודד באופן אמפירי את החלק של המסמכים הרלבנטיים שאלגוריתם חיפוש מקורב מחזיר מתוך סך כל המסמכים הרלבנטיים לשאילתה.

המחקר נעשה בפקולטה להנדסת חשמל על שם ויטרבי, הטכניון – מכון טכנולוגי לישראל.

תקציר

חיפוש באוסף של מסמכי טקסט הינו רכיב תוכנה פופולרי, הנמצא בשימוש נפוץ על ידי משתמשי קצה וכן על ידי אפליקציות. משתמשי קצה נעזרים במנועי חיפוש (כגון גוגל, יהאו ובינג) על בסיס יומיומי, כדי לחפש אינפורמציה המספקת את צרכי המידע שלהם. אפליקציות, כמו למשל מערכות המלצה, משתמשות בחיפוש כרכיב בסיסי באלגוריתמים שלהם.

במחקר זה אנו עוסקים בחיפוש מבוזר, שהינו המימוש הנפוץ ביותר כיום למערכות חיפוש מסחריות. מערכות אלו פועלות בסביבה של גדלים גבוהים, הן מבחינת כמויות המידע שהן מחפשות עליהן, והן מבחינת קצב השאילתות שהן משרתות. החיפוש המבוזר מבזר את שירות החיפוש על עד אלפי שרתים הנמצאים לרב במרכזי מידע (datacenters). במחקר זה אנו עוסקים בנוסף במימוש של מערכות חיפוש מבוזרות מעל מערכת *Peer-to-Peer (P2P)*. מערכת P2P הינה מערכת מחשבים המבוזרת באופן מוחלט, כלומר, ללא ניהול מרכזי כלל.

ה"גביע הקדוש" של אלגוריתמי חיפוש הינו סיפוק תוצאות באיכות גבוהה. פן מרכזי של *איכות החיפוש* הוא היכולת לאחזר תוצאות חיפוש שהן רלבנטיות לשאילתה. אך איכות החיפוש אינו האספקט היחידי הנלקח בחשבון. מערכות חיפוש מודרניות לוקחות בחשבון תכונות נוספות של המידע מלבד הרלבנטיות לשאילתה. לדוגמה, תכונות טמפורליות, כלומר תלויות זמן. תכונות אלו הינן בפרט בעלות חשיבות לאפליקציות המבצעות חיפוש על זרם של נתונים, לדוגמה, מידע המתפרסם ברשתות חברתיות וכן חדשות מקוונות. בהקשר זה, אנו מרחיבים את מושג איכות החיפוש ולוקחים בחשבון תכונות נוספות של המידע. כגון, הזמן בו הגיע המידע, איכות המידע, וכן מידת הפופולריות הדינמית שלו.

למרות שאיכות החיפוש הינה מוטיב מרכזי עבור משתמשים, אחזור של כל תוצאות החיפוש עבור השאילתה הוא לעתים לא יעיל, יקר מבחינת השימוש במשאבי המערכת, או אפילו בלתי אפשרי. לכן, מערכות חיפוש מבוזרות משתמשות לרב ב*חיפוש מקורב*, המאפשר אחזור של רק חלק מתוצאות החיפוש במחיר של פגיעה באיכות החיפוש. לדוגמה, חיפוש על כל מאגר המידע עשוי להיות יקר מבחינת מחיר התקשורת או העומס על השרתים. במקרים אלו מערכות חיפוש מבוזרות מבצעות לרב חיפוש רק על חלק מהשרתים ועשויות לאחזר רק חלק מתוצאות החיפוש. סיבה נוספת לאחזור של תוצאות מקורבות הינה אובדן של חלק מהתשובות: במערכות שרתים מבוזרות חלק מהשרתים עשויים להיות איטיים באופן משמעותי יותר מהאחרים. מערכת חיפוש המחכה לתשובות של שרתים איטיים תספק את תוצאות החיפוש הסופיות למשתמש תוך זמן ארוך, מה שיפגע ברווחיה. לכן, לרב, מערכות חיפוש מבוזרות מתעלמות משרתים איטיים וכך מאבדות לעתים חלק מתוצאות החיפוש. דוגמה שלישית הינה חיפוש במאגר מידע לא חסום – בזרם אינסופי של נתונים. במקרה זה, אחסון של כל המידע בשרתים אינו אפשרי עקב מגבלות של קיבולת מקום. לפיכך, רק חלק מהמידע מאוחסן בכל זמן נתון, מה שמאפשר חיפוש על חלק זה בלבד.

המחקר נעשה בהנחיית פרופ׳ עדית קידר וד״ר דוד כרמל בפקולטה להנדסת חשמל על שם ויטרבי בטכניון – מכון טכנולוגי לישראל.

תודתי נתונה לטכניון – מכון טכנולוגי לישראל ול-Hasso Plattner Institut בגרמניה על התמיכה הכספית הנדיבה בהשתלמותי.

עבודת מחקר זו מוקדשת לזכרו של גיסי, ד״ר קובי קראוס - חוקר מבריק ומבטיח, איש חכמה ואמת.

חיפוש אפקטיבי בסביבות מבוזרות

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
דוקטור לפילוסופיה

נעמה קראוס

הוגש לסנט הטכניון – מכון טכנולוגי לישראל

אב תשע״ז חיפה אוגוסט 2017