UNIVERSITY OF CALIFORNIA,
IRVINE

Spatial Indexing in the Era of Social Media.

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Sattam Alsubaiee

Dissertation Committee:
Professor Chen Li, Chair
Professor Michael J. Carey
Professor Sharad Mehrotra

2014

DEDICATION

To my beloved, precious, and dearest parents, Mubark and Huda, for their endless love, guidance, and encouragement; without them I would not be the person that I am today. To my siblings: Sultan, Salamah, Omran, Reem, and Abdullah for their continuous support, responsibility, and humor. To Jawaher, my loving wife, for standing besides me and taking care of me and my children; I am grateful for all her love and sacrifices that made me succeed. To the jewels of my life, my children, Huda and Mubark, for filling my life with love, joy, and happiness.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I am very grateful to my advisers Professor Chen Li and Professor Michael J. Carey for shaping me into an independent researcher.

Professor Li has taught me how to identify new interesting problems and how to approach them. He has spent tremendous time and effort in teaching me how to technically write papers and then how to do great presentations about them. Chen's help and guidance was an important factor in my success.

Professor Carey has inspired me to pick and solve practical problems that can have useful impact on user experience. His guidance in how to design and build different components of a big system was invaluable. He has taught me how to understand a system behavior by isolating variables, analyzing results, and drawing conclusions.

I also would like to thank Professor Sharad Mehrotra for joining my dissertation committee. I was motivated by his passion and enthusiasm.

I thank Vinayak Borkar and Dr. Alexander Behm for their valuable suggestions and discussions during my thesis work. I also thank Zachary Heilbron and Young-Seok Kim for their fruitful collaboration that resulted in producing the forth chapter of my thesis.

I would like to thank my co-authors and also my other teammates in the Flamingo and AsterixDB projects for their discussions and code reviews, especially, Abdullah Alamoudi, Yingyi Bu, Raman Grover, and Dr. Shengyue Ji.

I thank my mentors during my internship at HP Labs Dr. Goetz Graefe, Dr. Harumi A. Kuno, and Wey Guy for enriching my academic knowledge with industry-level experience.

I would like to thank my friends Hashem Alayed, Osama Alfalah, Dr. Mubarak Alhajri, Dr. Abdulaziz Alhussien, Dr. Mishari Almishari, Dr. Majed Alresaini, Abdulrahman Alzaid, and Mohammed Alzaid for their support and humor, and special thanks to my friends Yasser Altowim and Hotham Altwaijry for their moral support and all the fun times that I spent with them during the program.

I would like to thank KACST for providing me with a generous Graduate Study Fellowship during my PhD work. The work reported in this thesis has also been supported by a UC Discovery grant, by NSF IIS awards 0910989 and 0844574, and by NSF CNS awards 1305430 and 1059436. In addition, the AsterixDB project has benefited from generous industrial support from Amazon, eBay, Facebook, Google, HTC, Microsoft, Oracle Labs, and Yahoo.

Finally, I thank ACM and VLDB Endowment for permissions to include materials from "Supporting Location-Based Approximate-Keyword Queries" (ACM SIGSPATIAL GIS 2010) and "Storage Management in AsterixDB" (VLDB 2014) into my thesis.

CURRICULUM VITAE

Sattam Alsubaiee

EDUCATION

Doctor of Philosophy in Information and Computer Science — **2014**
University of California, Irvine — *Irvine, California*

Master of Science in Computer Science — **2008**
University of Southern California — *Los Angeles, California*

Bachelor of Science in Computer Science — **2005**
King Saud University — *Riyadh, Saudi Arabia*

SELECTED HONORS AND AWARDS

Ph.D. Fellowship Award — **2008-2014**
King Abdulaziz City for Science and Technology (KACST)

Excellent Demo Award — **2010**
International Conference on Database Systems for Advanced Applications (DASFAA)

Master Fellowship Award — **2006-2008**
King Abdulaziz City for Science and Technology (KACST)

PUBLICATIONS

AsterixDB: A Scalable, Open Source BDMS **2014**
International Conference on Very Large Databases (VLDB)

Storage Management in AsterixDB **2014**
International Conference on Very Large Databases (VLDB)

ASTERIX: An Open Source System for "Big Data" Management and Analysis (Demo) **2012**
International Conference on Very Large Databases (VLDB)

ASTERIX: Scalable Warehouse-Style Web Data Integration **2012**
International Workshop on Information Integration on the Web (IIWeb)
co-located with the ACM Special Interest Group on Management of Data (SIGMOD)

Supporting Location-Based Approximate-Keyword Queries **2010**
ACM International Conference on Advances in Geographic Information Systems (GIS)

Fuzzy Keyword Search on Spatial Data (Demo) **2010**
International Conference on Database Systems for Advanced Applications (DASFAA)

ABSTRACT OF THE DISSERTATION

Spatial Indexing in the Era of Social Media.

By

Sattam Alsubaiee

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 2014

Professor Chen Li, Chair

The rapid adoption of smart phones and the social media boom has increased the interest in location-based services. A new set of applications and popular online services that utilize users' locations have been created, and many ordinary people are increasingly interacting with these services on a daily basis through their smart phones, tablets, cameras, etc., where most of those gadgets come equipped with GPS sensors. The new complex features provided by those applications and the scale of the massive data handled by them impose new and interesting challenges for spatial databases. In this thesis, we present spatial indexing and query processing techniques in response to some of these challenges.

First, we study how to support approximate keyword search on spatial data. There are many popular websites that support keyword search on their spatial data, such as business listings and photos. In these systems, users may experience difficulties finding the entities they are looking for if they do not know their exact spelling, such as the name of a restaurant. We develop three algorithms for constructing a specialized index that can answer location-based approximate keyword queries, successively improving the time and space efficiency by exploiting the textual and spatial properties of the data. We experimentally demonstrate the efficiency of our techniques on real, large datasets.

Second, we introduce a framework for converting an in-place update, disk-based data struc-

ture to a deferred-update, append-only data structure. We show that converting an R-tree index (and other non-totally ordered index) to an LSM index is non-trivial if the resultant index is expected to have performant read and write operations. Our framework enables the "LSM-ification" of any kind of index structure that supports certain primitive operations, enabling the index to ingest data efficiently. We have implemented our framework in the context of the AsterixDB system as a way to extend both the R-tree and the inverted keyword index to LSM-based indexes. Our results have shown that using an LSM-based version of the R-tree can significantly outperform its conventional counterpart for *both* ingestion and query speed.

Third, we study how to optimize the performance of query workloads that favor recent data. There are many use cases where users of a database system are mostly interested in querying recent data. We propose a solution that exploits the natural partitioning property that LSM-based indexes provide for its components, allowing us to filter out many components when answering queries. Our solution is generalizable to any LSM-based index structure including LSM R-trees, and has been implemented in the context of the AsterixDB system. Our experiments show that we can reduce query times by up to 99% for selective range predicates.

Chapter 1

Introduction

Research in spatial databases has been an active area in the last few decades, mainly driven by applications such as Computer Aided Design (CAD), Multimedia Information System (MMIS), NASA's Earth Observation System (EOS), and more importantly Geographical Information System (GIS). Spatial query processing has become a significant niche market and important enough such that many database management systems (DBMS) provide native support for it. With the evolving world of social media and the mobile devices boom, however, location-based applications are now becoming more important than ever. A new set of applications and popular online services that utilize users' locations have been created, and many ordinary people are increasingly interacting with these services on a daily basis through their smart phones, tablets, cameras, etc., where most of those gadgets come equipped with GPS sensors. Recent studies show that smart phone penetration has gone well beyond 50% in the US [46], and this trend is expected to continue in the near future. Many social networks, such as Facebook, Twitter, and Google+, allow users to geo-tag their statuses and tweets. In fact, there are popular social networks, e.g., Foursquare, where their core business depends on users sharing their location using their mobile devices. The new complex features provided by those applications and the scale of the massive data handled

by the applications impose new and interesting challenges for spatial databases.

In this thesis, we present spatial indexing and query processing techniques in response to some of these challenges. In particular, we study three different problems and provide an efficient solution for each one of them.

Location-based approximate keyword search: Many websites support keyword search on their spatial data, such as business listings, social network posts, and photos. They accept queries consisting of two conditions: a set of keywords and a spatial location. The goal is to find objects with these keywords close to the location. In these systems, inconsistencies and errors can exist in both queries and the data. For instance, a user might be looking for a restaurant called `Alouette` close to `New York`. The website returns listings close to `New York` that have the keyword `Alouette`. Sometimes, users may not know the exact spelling of the entities they are looking for. For example, a user could have heard of the restaurant by word-of-mouth but is unfamiliar with its exact spelling, and issues the following query with a typo: `Aloette` close to `New York`. Similarly, errors can exist in the data as well. For instance, websites such as Twitter and Flickr support location-based keyword search. Since the posts in those websites are created by users, the textual information corresponding to a post may have spelling errors. To bridge the gap between queries and data with possible errors, it is important to support approximate keyword search on spatial data. In the first part of this thesis, we study how to answer such queries efficiently. We focus on a natural index structure that augments a tree-based spatial index, such as R-tree, with capabilities for approximate keyword search. We systematically study how to efficiently combine these two types of indexes and how to search the resulting index to find answers. We develop three algorithms for constructing the index, successively improving the time and space efficiency by exploiting the textual and spatial properties of the data. We experimentally demonstrate the efficiency of our techniques on real, large datasets.

A general purpose LSM-based indexing framework: Social networks, online commu-

nities, mobile devices, and instant messaging applications are generating digital information at an increasing rate. As an example, Twitter's users post more than 500 million tweets per day [43]. With the increasing adoption of smart phones, this rate is expected to rise even higher. Most conventional database index structures such as B^+-trees use in-place writes to perform updates and have been widely used in traditional relational database systems due to their low read latency. However, today's emerging applications require handling append-intensive workloads where the performance of in-place write index structures may not be acceptable. For this reason, nowadays many NoSQL (a.k.a. key-value) databases adopt the Log-Structured Merge (LSM) Tree [37] as their underlying storage structure. The LSM-tree is able to ingest data at a high rate by temporarily batching the incoming records into a main memory buffer, and then sequentially flushing the buffer to disk, eliminating costly random disk writes. Since much of today's social data is geo-tagged, many applications desire the ability to issue spatial queries against geo-tagged data or even analyze it for useful insights. Most conventional spatial indexes, such as the R-tree, also use in-place writes; therefore, they cannot keep up with the high ingestion rate required by many applications. Consequently, it is natural to extend the R-tree (or any other in-place index structure) to use the same technique as the LSM-tree. Unfortunately, it turns out that the extension of those indexes lacking a total order among their entries, such as R-tree, is challenging. In the second part of this thesis, we discuss these challenges and propose a generic framework to extend conventional index structures to be LSM-based indexes. Our framework can utilize existing non-LSM indexes by behaving as a wrapper. Thus, we can avoid designing and implementing new specialized indexes from scratch, saving development time. We have implemented our framework in the context of the AsterixDB system [7, 5, 14, 8, 6] as a way to extend both R-tree and inverted keyword index to LSM-based indexes. Furthermore, for the special case of the R-tree, we provide an additional possible technique to extend the R-tree to an LSM-based index. We present initial performance evaluation results of AsterixDB focusing on the LSM-based storage system, and also experimentally compare the two versions of the

LSM-based R-tree.

Filter-based LSM index acceleration: Large volumes of events and social data can be aggregated and analyzed to derive knowledge valuable to businesses, governments, and society. Although older historical data can be useful to infer interesting patterns, in many cases more recent or real-time data analysis is more important to the decision making process. For example, consider the case where the CEO of a smart phone company such as Apple wants to know how consumers are reacting to a new release of the iPhone in a certain geographic area. Such information is clearly valuable to the decision-making process of the company. The increasing availability and popularity of social data and event data make such information more readily available and more real time. We can derive such knowledge by doing a *spatial aggregation* on Twitter data as follows: We can formulate a query to find the tweets mentioning "iPhone" that have originated from Los Angeles in the *last* one hour, group them on a spatial grid, and compute the number of such tweets in each cell in the grid. By doing this time-based spatial analysis, the smart phone business managers can easily have a good understanding of the market and thus can make important, correct, and critical decisions. With the social-media data explosion, however, answering near real-time queries efficiently can be challenging. In the third part of this thesis, we study how to efficiently answer queries that are targeting recent data within very large data sets. We propose a solution that exploits the natural partitioning property that LSM-based indexes provide for its components, allowing us to filter out many components when answering queries. Our solution is generalizable to any LSM-based index structure, and can be applied not just on temporal fields (e.g., based on recency), but also can be applied to any "time correlated fields" such as Universally Unique Identifiers (UUIDs), user-provided integer ids, etc. We have implemented and experimentally evaluated the solution in the context of AsterixDB.

Chapter 2

Preliminaries and Related Work

In this chapter, we introduce basic concepts and existing technologies related to our work. We start by providing a brief explanation of the Log Structured Merge (LSM) tree, followed by a high-level introduction to the AsterixDB Big Data Management System. We then discuss the related work in the area of spatial-textual indexing, log structured storage, and the usage of partitioning for effective filtering power.

2.1 LSM-Tree

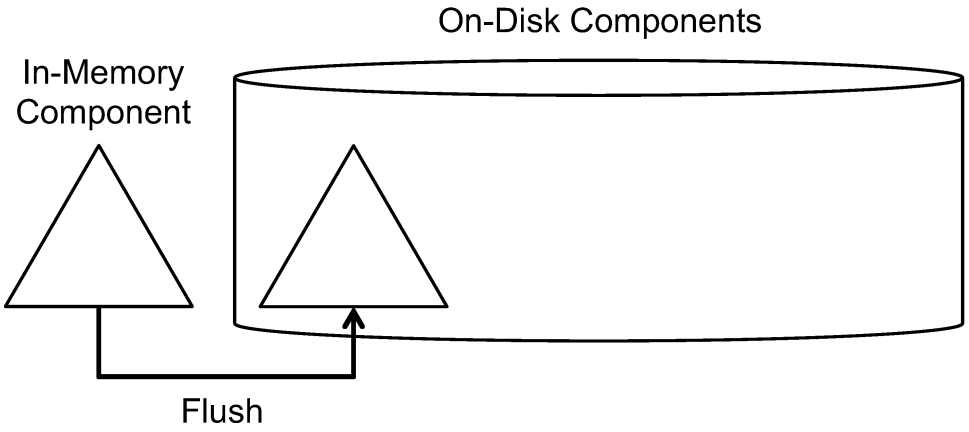
An LSM-tree [37] is an ordered, persistent index structure that supports typical operations such as insert, delete, and query. It is optimized for frequent or high-volume updates. By first batching updates in memory, the LSM-tree amortizes the cost of an update by converting what would have been a disk seek into some portion of a sequential I/O.

Entries being inserted into an LSM-tree are initially placed into a component of the index that resides in main memory—an *in-memory component*. When the space occupancy of the in-memory component exceeds a specified threshold, entries are *flushed* to disk. As entries

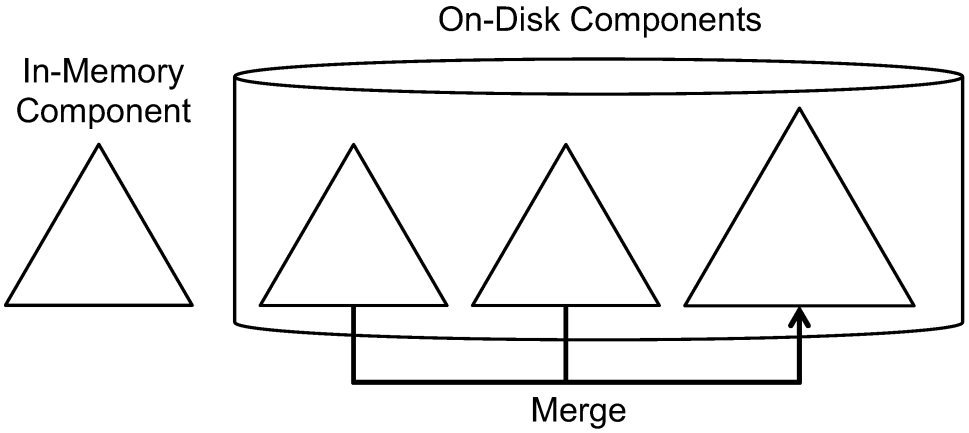
accumulate on disk, the entries are periodically merged together subject to a *merge policy* that decides when and what to merge. In practice, two different variations of flush and merge are used. Block-based, "rolling merges" (described in [37]) periodically migrate blocks of entries from newer components (including the in-memory component) to older components that reside on disk—*disk components*—while maintaining a fixed number of components. On the other hand, component-based flushes migrate an entire component's worth of entries to disk, forming a new disk component, such that disk components are ordered based on their freshness, so component-based merges combine the entries from a sequence of disk components together to form a new disk component. Popular NoSQL systems commonly employ the component-based variation, where in-memory and disk components are usually called *memtables* and *SSTables*, respectively (e.g., [17]). Throughout the rest of this thesis, a reference to an LSM-tree implies a reference to a component-based LSM-tree. Figure 2.1 shows these component-level LSM operations. As the case in [37], a component is often another instance of a disk-inspired structure such as a B^+-tree. However, it is possible to use other index structures whose operations are semantically equivalent (e.g., Skip List [39]) for the in-memory component.

Disk components of an LSM-tree are immutable. Modifications (e.g., updates and deletes) of existing entries are handled by inserting control entries into the in-memory component. A delete (or "anti-matter") entry, for instance, carries a flag marking it as a delete, while an insert can be represented simply as the new entry itself. Control entries with identical keys must be *reconciled* during searches and merges by either annihilating older entries in the case of a delete, or by replacing older entries with a new entry in the case of an update. During merges, older deleted entries may be safely ignored when forming a new component, effectively removing them from the index.

Entries in an LSM-tree are scattered throughout the sequence of components, which requires range scans to be applied to all of the components. As entries are fetched from the



(a) Flushing a full in-memory component to disk.



(b) Merging multiple on-disk components into a single component.

Figure 2.1: Basic LSM operations: flush and merge.

components, the reconciliation described above is performed. A natural design for an LSM-tree range scan cursor that facilitates reconciliation is a heap of sub-cursors sorted on ⟨key, component number⟩, where each sub-cursor operates on a single component. This design temporally groups entries with identical keys, easing reconciliation.

Point lookups in unique indexes can be further optimized. Given key uniqueness in an LSM-tree, cursors can access the components one-by-one, from newest to oldest (i.e., in component number order), allowing for early termination as soon as the key is found. Enforcing key uniqueness, however, increases the cost of an insertion since an additional integrity check is now required: the index must first be searched for the key that is being inserted. This is in

contrast to the typical usage of LSM-trees in popular NoSQL systems, where the semantics of insert usually mean "insert if not exists, else update" (a.k.a. "upsert"), where primary key existence is not considered an error. Throughout the rest of the thesis, references to *insert* will assume the semantics "insert if not exists, else error if the key exists". As suggested in [42], a Bloom filter can be maintained in main memory for each disk component to reduce the chance of performing unnecessary disk I/O during point lookups, thereby decreasing the cost of performing the integrity check and point lookups in general.

Compared to a B^+-tree, the LSM-tree offers superior write throughput at the expense of reads and scans [27, 37]. As the number of disk components increases, search performance degrades since more disk components must be accessed. It is therefore desirable to keep few disk components by periodically merging multiple components into fewer, larger components in order to maintain acceptable search time. We refer interested readers to [41] for a recent study of advanced LSM merging strategies.

2.2 AsterixDB

The AsterixDB project [1, 8, 14] started in 2009 at UC Irvine. The goal at the outset was to design and implement a highly scalable platform for information storage, search, and analytics. By combining and extending ideas from semistructured data management, parallel database systems, and first-generation data-intensive computing platforms (MapReduce and Hadoop), AsterixDB was envisioned to be a parallel, semistructured information management system with the ability to ingest, store, index, query, analyze, and publish very large quantities of semistructured data. AsterixDB is well-suited to handle use cases ranging all the way from rigid, relation-like data collections, whose types are well understood and invariant, to flexible and more complex data, where little is known a priori and the instances in data collections are highly variant and self-describing.

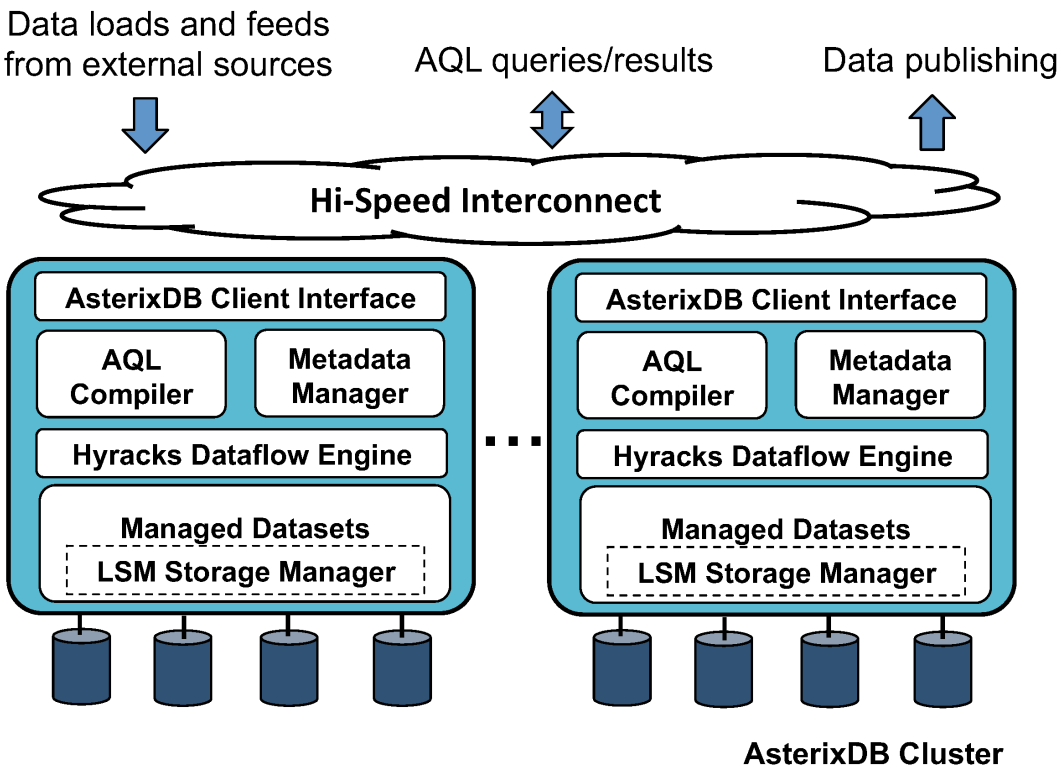


Figure 2.2: AsterixDB system architecture.

Figure 2.2 provides an overview of an AsterixDB cluster and how the software components of AsterixDB map to a node in a shared-nothing cluster. The bottom-most layer provides storage facilities for managed datasets based on LSM-trees, which can be targets of ingestion. Datasets are managed by AsterixDB as partitioned LSM-based B^+-trees with optional LSM-based secondary indexes. Further up the stack lies a data-parallel runtime called Hyracks [16]. It sits at roughly the same level that Hadoop does in implementations of other high-level languages such as Pig [36], Hive [12] or Jaql [26]. The topmost layer of AsterixDB is a parallel DBMS, with a full, flexible data model (ADM) and query language (AQL) for describing, querying, and analyzing data. AQL is comparable to languages such as Pig, Hive, or Jaql, but ADM and AQL support both native storage and indexing of data as well as access to external data (e.g., in HDFS). As part of the AQL compiler, there exists a layer called Algebricks, a model-agnostic, algebraic "virtual machine" for optimizing parallel queries. Algebricks is the target for AQL compilation, but it can also be the target for

other declarative data languages. ADM includes support for both spatial and textual data, as does the AQL query language.

2.3 Related Work

In this section, we touch upon existing work relevant to spatial-textual indexing, log structured storage, and data partitioning that is used by different systems for effective filtering power.

Spatial keyword search: There have been a number of studies on answering spatial queries for exact matching of keywords [50, 44, 19, 25, 22, 20, 49]. Chen et al. [19] used separate indices for the spatial and textual information. Zhou et al. [50] suggested a hybrid index that combines spatial and inverted indexes, where they use an R*-tree [13] and build inverted indexes for the keywords at the leaf nodes, or use an inverted index to store the keywords and build an R*-tree for each keyword. These techniques do not simultaneously use the spatial and the textual information for pruning. Other studies [25, 22, 20, 49] used the textual and spatial information conjunctively. They augment a tree-based spatial index with textual information in each node. Our work in Chapter 3 complements these studies by allowing approximate as well as exact keyword matching.

Approximate string search: We refer to the problem of conjunctive keyword search with relaxed keywords as *approximate keyword search*. An important subproblem of approximate *keyword* search is that of approximate *string* search, defined as follows. Given a collection of strings, find those that are similar to a given query string. There are two main families of approaches to answer such queries. (1) Trie-based method: The string collection is stored as a trie (or a suffix tree). To answer an approximate string query, the trie is traversed and the subtrees that cannot be similar to the query are pruned. Popular pruning techniques use

an NFA or a dynamic programming matrix with backtracking. We refer the reader to [34] for more details. (2) Inverted-index method: Its main idea is to decompose each string in the collection to small overlapping substrings (called "grams") and build an inverted index on these grams. More details on gram-based indexing and search algorithms can be found in [24, 30, 34].

Spatial approximate-keyword search: Yao et al. [48] proposed a structure called MHR-tree to answer spatial approximate-keyword queries. They enhance an R-tree [23] with min-wise signatures at each node to compactly represent the union of the grams contained in objects of that subtree. They then use the concept of set resemblance between the signatures and the query strings to prune branches in the tree. The main advantage of this approach is that its index size does not require a lot of space since the min-wise signatures are very small. However, the method could miss query answers due to the probabilistic nature of the signatures, while the approach that we will present in Chapter 3 can guarantee to find all true answers. In Section 3.6, we compare these two solutions experimentally.

Log structured storage: Utilizing memory to support efficient write-heavy workloads has been suggested by the log-structured file system (LFS) [40]. Inspired by the LSM-tree [37], different variants of log-structured B^+-trees have been introduced in the literature and some have been deployed commercially [2, 3, 4, 17]. The bLSM-tree [41] uses advanced scheduling algorithms to provide predictable write performance. LogBase [18] has adopted log-only storage, where the logs are the actual data repository. The Bkd-tree [38] utilizes an kd-tree to index multi-dimensional point data to provide efficient data ingestion for spatial workloads.

A partitioned exponential file [27] offers a generic data structure for write-intensive workloads that can be customized for different types of data. Our work presented in Chapter 4 differs in three ways:

1) We provide a more general approach for converting existing index structures to LSM-based index structures that avoids building specialized indexes from scratch, saving index structure design and development time.

2) We have designed and implemented a framework to enforce the ACID properties across multiple heterogeneous LSM indexes (interested reader can refer to [7] for more details).

3) We provide a full implementation of the work in AsterixDB as an open-source software package.

Utilizing LSM structures for efficient query processing: Muth et al. [33] have introduced an index structure called LHAM for transaction-time temporal data. They used an LSM-like structure to index record versions in two dimensions: one dimension is given by the conventional record key and the other by the timestamp of the record version. The record version is obtained at the time of insertion, representing the transaction timestamp. The outcome is that the data is partitioned into successive components based on the timestamps of the record versions. Queries with temporal predicates only need to access those components that satisfy the predicates and the remaining components can be skipped. HBase uses a similar idea to LHAM by using the LSM-tree to store incoming records and maintaining a timestamp with each record for versioning purposes. Each LSM component is then tagged with the minimum and maximum timestamps of the records contained in the component. When answering a query with temporal predicates, HBase can leverage the minimum and maximum timestamp filters to only access those relevant components, reducing query time.

Our work presented in Chapter 5 differs than LHAM and HBase in the following ways:

1) We do not require the filters to be used with transaction timestamps. Instead, when creating a dataset, the user is allowed to create a filter on any totally-ordered datatype field of the record (e.g., integer, double, datetime, UUID, etc.).

2) We apply the filters not only on disk components of the primary index, but also on the disk components of the secondary indexes, allowing significant additional pruning power.

3) We study the effect of using different merge policies and out-of-order records on query performance when using our generalized filters and experimentally show their effect.

Chapter 3

Supporting Location-Based Approximate-Keyword Queries

3.1 Introduction

As described in Chapter 1, an increasing number of websites support location-based keyword search on their data objects such as business listings and photos. They accept queries consisting of two conditions: a set of keywords and a spatial location. The goal is to find objects with these keywords close to the location. Such a query is called a *spatial-keyword* query [25]. For instance, there are several local-search websites, such as Bing Maps, Google Maps, Yahoo! Local, and Yelp.

In these systems, inconsistencies and errors can exist in user queries and data. For instance, a user might be looking for a restaurant called `Suteishi` close to `New York`. The following is the corresponding query:

Q_1: (`Suteishi Restaurant`) *near* (`New York`).

The website will respond by returning listings close to `New York` that have the keyword `Suteishi`. Sometimes, users may not know the exact spelling of the entities they are looking for. For instance, a user could have heard of the restaurant by word-of-mouth but be unfamiliar with its exact spelling, and might issue the following query with a typo:

$$Q_1'\text{: (}\texttt{Sutishi Restaurant}\text{) } \textit{near} \text{ (}\texttt{New York}\text{).}$$

Errors can exist in data as well. For instance, Flickr supports location-based photo search.[1] Consider a query that asks for photos of `Alcatraz` close to `San Francisco`:

$$Q_2\text{: (}\texttt{Alcatraz}\text{) } \textit{near} \text{ (}\texttt{San Francisco}\text{).}$$

Since Flickr photos are manually uploaded and tagged by users, the title or the description of a photo may have spelling errors. Query Q_2 may not be able to find a photo with the mistyped title `Alkatraz`.

Finding relevant answers to such queries is important. Unfortunately, most existing location-based search engines currently will not provide correct answers to the query Q_1' even with a single typo (as of August 1, 2014). One simple solution to this problem on a spatial dataset is to build a collection of keywords from the dataset. For a mistyped keyword, we find words from the collection that are similar to the keyword. We use these similar keywords to suggest another query or find objects with these keywords. The main drawback of this approach is that it does not consider the location condition when relaxing the mistyped keyword.

In this chapter, we study how to support approximate keyword search on spatial data. Answering such queries efficiently is critical to these websites to achieve a high query throughput to serve many concurrent users. Although there are studies on both approximate keyword search and location-based search, the problem of supporting both types of search simultaneously has received little attention. To answer such queries, a natural index structure is to

[1] http://www.flickr.com/map

augment a tree-based spatial index with approximate-string indexes such as a gram-based inverted index or a trie-based index. The main focus of this work is a systematic study of how to efficiently combine these two types of indexes and how to search the resulting index (called LBAK-tree) to find answers.

We develop three algorithms in this context. First, in Section 3.3, we show a basic algorithm that selects a fixed level of nodes in the spatial tree to store approximate-string indexes. (A brief description of a system prototypes using this approach is presented in [11].) Second, in Section 3.4, we develop a cost-based algorithm that judiciously selects a set of nodes in the tree to store approximate indexes. Our cost model utilizes the spatial distribution of objects within each node to build the index structure with a space budget. Third, in Section 3.5 we continue improving the solution by exploiting the frequency distribution of keywords. We show how to further reduce the index size without sacrificing query time. We have conducted a comprehensive experimental study, presented in Section 3.6, to evaluate the proposed techniques. Our results, which has also appeared in [9], show the efficiency and scalability of these optimizations.

3.2 Problem Formulation

The problem of approximate keyword search on spatial data can be formulated as follows. Consider a collection of spatial objects $o_1, \ldots, o_n$, each having a textual description (a set of keywords) and a location. A spatial approximate-keyword query $Q = \langle Q_s, Q_t \rangle$ consists of two conditions: a spatial condition Q_s such as a rectangle or a circle, and an approximate keyword condition Q_t having a set of k pairs $\{\langle w_1, \delta_1 \rangle, \langle w_2, \delta_2 \rangle, \ldots, \langle w_k, \delta_k \rangle\}$, each representing a keyword w_i with an associated similarity threshold δ_i. The similarity thresholds refer to a similarity measure such as edit distance, Jaccard, etc., which could be different for each keyword. Our goal is to find all objects in the collection that are within the spatial region Q_s

and satisfy the approximate keyword condition Q_t. We focus on conjunctive approximate keyword queries; thus, an object satisfies the approximate keyword condition if for each keyword w_i in Q_t, the object has a keyword in its description whose similarity to w_i is within the corresponding threshold δ_i. We mostly discuss range queries but our solutions can be easily adapted to support nearest-neighbor queries.

Before proceeding, we briefly review the basics of answering queries with spatial conditions, and answering approximate string queries. Queries with a spatial condition are typically supported by a tree-based index such as an R*-tree, KD-tree, Quad-tree, etc. We assume a tree-based spatial index as a basis for our work, and we often use an R*-tree as a representative tree-based spatial access method. To support the approximate keyword condition, we use an index for approximate string search. Most trie-based indexes are specific to edit distance and its variants, and do not support other similarity measures such as Jaccard. An inverted-index approach [30] supports a family of similarity metrics, such as edit distance, Jaccard, etc. The two methods differ in their performance characteristics, whose specifics are independent of our proposed framework. We focus on the inverted-index approach, though we can as well choose a trie-based approach.

3.3 Basic Index and Search

In this section, we introduce a basic index structure and the corresponding search algorithm for answering location-based approximate-keyword (LBAK) queries.

3.3.1 The LBAK-Tree

The main idea of the basic index (called "LBAK-tree") is to augment a tree-based spatial index with capabilities for approximate string search and keyword search. We use approxi-

mate string search to identify for each query keyword those strings that are similar. Once we have identified similar keywords, we use the keyword-search capability to prune search paths. For example, Figure 3.1 shows an LBAK-tree, e.g., an R*-tree that has been enhanced to support spatial approximate-keyword queries. To support keyword search we choose some nodes to store the union of keywords contained in the objects of their subtree. To support approximate string search, we select nodes in the tree to build approximate string indexes (called "approximate indexes" hereafter) on their stored keywords. In this example and later discussions, we use a gram-based inverted index to perform approximate string search, but our solutions naturally extend to other types of approximate string indexes.

Fixed-level solution: A simple way to choose nodes to place approximate indexes is the following. We choose one level L for which we construct approximate indexes at each node. The challenge is to choose a level L that provides good query performance. Notice that at each tree node, its stored keywords are the *union* of keywords of the objects in its subtree. If multiple objects have the same keyword, this keyword is stored only once in their common ancestors. There is a trade off between the average query time and the size of the approximate indexes. As we move L down the tree, the total number of keywords on which we need to build approximate indexes increases. Thus the total size of the approximate indexes will increase. Meanwhile, the performance of finding similar keywords from an approximate index is related to the size of the index. Typically the smaller the index is, the faster its lookups are. On the other hand, doing many approximate-keyword lookups on small indexes may increase the total running time as well.

3.3.2 Search Algorithm

Here, we discuss the search procedure for answering spatial approximate-keyword queries. We classify the LBAK-tree nodes into three categories:

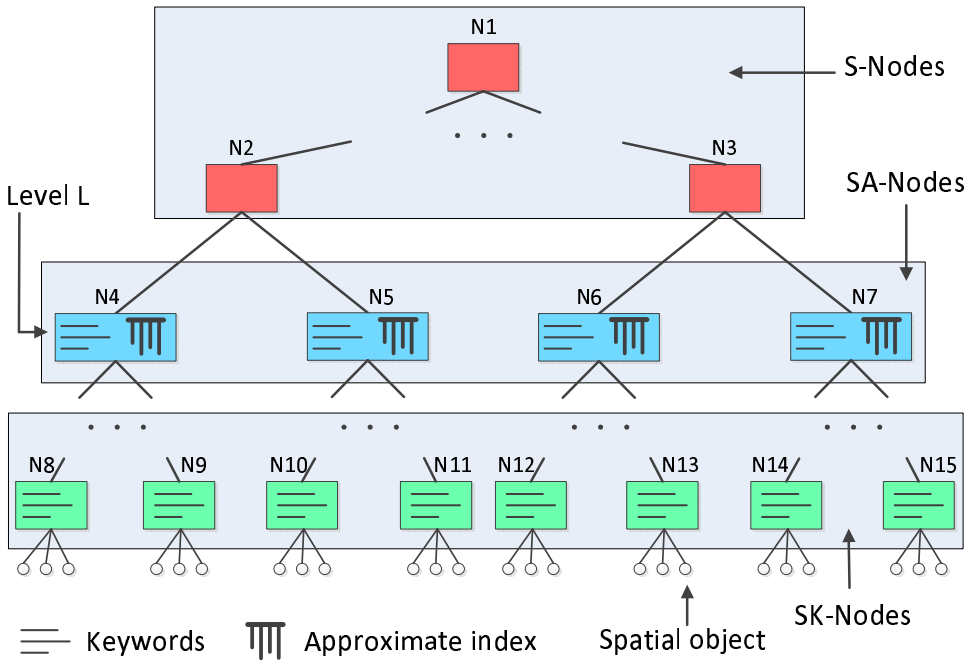


Figure 3.1: The LBAK-tree with approximate indexes at one level, and nodes enhanced with keywords.

- *S-Nodes* do not store any textual information such as keywords or approximate indexes, and can only be used for pruning based on the spatial condition.
- *SA-Nodes* store the union of keywords of their subtree, and an approximate index on those keywords. We use SA-Nodes to find similar keywords, and prune subtrees with the spatial and approximate keyword conditions.
- *SK-Nodes* store the union of keywords of their subtree, and prune with the spatial condition and its keywords. Note that we must have previously identified the relevant similar keywords once we reach an SK-Node.

Algorithm outline: Let Q be a query with a spatial condition Q_s and an approximate-keyword condition Q_t with k keywords. The algorithm traverses the tree top-down and performs the following actions at a tree node depending on the node type. At an S-Node, the algorithm only relies on the spatial information of the node to decide which children to traverse. Once the algorithm reaches an SA-Node, it uses the node's approximate index

to find similar keywords. For each keyword w_i in Q_t, the algorithm maintains a set of keywords C_i that are similar to w_i according to its similarity threshold δ_i. Assuming we use the AND-semantics, a node is pruned if one of the query's C_i sets is empty. Otherwise, we propagate the list $C = C_1, \ldots, C_k$ downward and use it for further pruning. In particular, at each SK-Node n, for each keyword w_i in Q_t, the algorithm intersects its similar keywords C_i propagated from the parent with the stored keywords of n. The node n can be pruned if one of the similar keyword-sets C_i has an empty intersection with the node's keywords. Otherwise, the algorithm passes those intersection sets of similar keywords to n's children for further traversal. At a leaf node, the algorithm adds to the answer set all the node's objects that satisfy the condition Q_s and have a non-empty intersection between their keywords and each of the propagated similar-keyword sets from the query.

Pseudo-code: Let us examine the pseudo-code for LBAKSearch in Algorithm 1. Note that the algorithm invokes several helper procedures, e.g., InitSimilarKeywordSets, ProcSNode, etc., defined in Algorithms 2, 3, 4, and 5. In later sections, we will override these helper procedures, but the procedure in Algorithm 1 remains the same.

The input of Algorithm 1 is a query $Q = \langle Q_s, Q_t \rangle$ and an LBAK-tree root r. We initialize a list C of k similar-keyword sets (line 2), where k is the number of keywords in Q_t. We maintain a stack S to traverse the tree. Initially, we push the root r and the list C to the stack (line 4). We start traversing the tree by popping the pair (n, C) from the stack (line 6). If n is not a leaf node, then all n's children that satisfy the spatial condition will be investigated (lines 7-9). Depending on the type of the node we invoke a helper procedure to process it (lines 10-18):

For an S-Node (lines 11-12), we only rely on the spatial condition to do pruning, and push the pair (n_i, C) to the stack S (within Algorithm 3). For an SA-Node (lines 13-14), we use its approximate index to find similar keywords for each query keyword as shown in Algorithm 4. We call GetSimKwds (w_i, δ_i) to get w_i's similar keywords, for $i = 1,\ldots,k$, and store them in

Algorithm 1: LBAKSearch

Input : A query $Q = \langle Q_s, Q_t \rangle$, with Q_t having k pairs $\{\langle w_1, \delta_1 \rangle, \ldots, \langle w_k, \delta_k \rangle\}$ associating a keyword w_i with its similarity threshold δ_i;
An LBAK-tree root r;

Output: A set R of all objects satisfying Q_s and Q_t;

```
1  Result set R ← ∅;
2  C ← InitSimilarKeywordSets(r, Q_t);
3  Initialize an empty stack S;
4  S.push (r, C);
5  while S ≠ ∅ do
6      (n, C) ← S.pop();
7      if n is not a leaf node then
8          foreach child n_i of n do
9              if n_i does not satisfy Q_s then continue;
10             switch n_i.type do
11                 case S-Node:
12                     S ← ProcSNode(n_i, Q_t, C, S);
13                 case SA-Node:
14                     S ← ProcSANode(n_i, Q_t, C, S);
15                 case SK-Node:
16                     S ← ProcSKNode(n_i, Q_t, C, S);
17             endsw
18         end
19     else // leaf node
20         foreach object o_i of n do
21             if o_i satisfies Q_s and Q_t then
22                 R.add(o_i);
23             end
24         end
25     end
26 end
27 return R
```

Algorithm 2: InitSimilarKeywordSets

Input : Root r of an LBAK-tree;
$Q_t = \{\langle w_1, \delta_1 \rangle, \ldots, \langle w_k, \delta_k \rangle\}$

Output: A list of similar-keyword sets $C = C_1, \ldots, C_k$;

```
1  C ← {∅, ∅, ..., ∅} // k empty sets
2  return C
```

Algorithm 3: ProcSNode

Input : S-Node n;
$Q_t = \{\langle w_1, \delta_1\rangle, \ldots, \langle w_k, \delta_k\rangle\}$;
Similar-keyword sets $C = C_1, \ldots, C_k$;
Stack S;
Output: Stack S;

1 S.push (n, C);
2 **return** S

Algorithm 4: ProcSANode

Input : SA-Node n;
$Q_t = \{\langle w_1, \delta_1\rangle, \ldots, \langle w_k, \delta_k\rangle\}$;
Similar-keyword sets $C = C_1, \ldots, C_k$;
Stack S;
Output: Stack S;

1 **for** *i=1* **to** k **do**
2 | $C_i \leftarrow n$.GetSimKwds(w_i, δ_i);
3 **end**
4 **if** *all* C_i*'s* $\neq \emptyset$ **then**
5 | S.push (n, C);
6 **end**
7 **return** S

Algorithm 5: ProcSKNode

Input : SK-Node n;
$Q_t = \{\langle w_1, \delta_1\rangle, \ldots, \langle w_k, \delta_k\rangle\}$;
Similar-keyword sets $C = C_1, \ldots, C_k$;
Stack S;
Output: Stack S;

1 **for** *i=1* **to** k **do**
2 | $G_i \leftarrow n$.keywords $\cap\, C_i$;
3 **end**
4 **if** *all* G_i*'s* $\neq \emptyset$ **then**
5 | $G \leftarrow G_1, G_2, \ldots, G_k$;
6 | S.push (n, G);
7 **end**
8 **return** S

w_i's corresponding similar-keyword set C_i. If at least one similar keyword is found for each keyword in Q_t, then the pair (n_i, C) is pushed to the stack S for future investigation. For an SK-Node (lines 15-16), we compute the intersection G_i between n_i's keywords and the similar-keyword set C_i. If all the intersection sets are not empty, then the pair (n_i, G) is pushed to S to be examined later (Algorithm 5). Finally, when we reach a leaf node, we add its objects that satisfy the two conditions Q_t and Q_s to the results (lines 20-26).

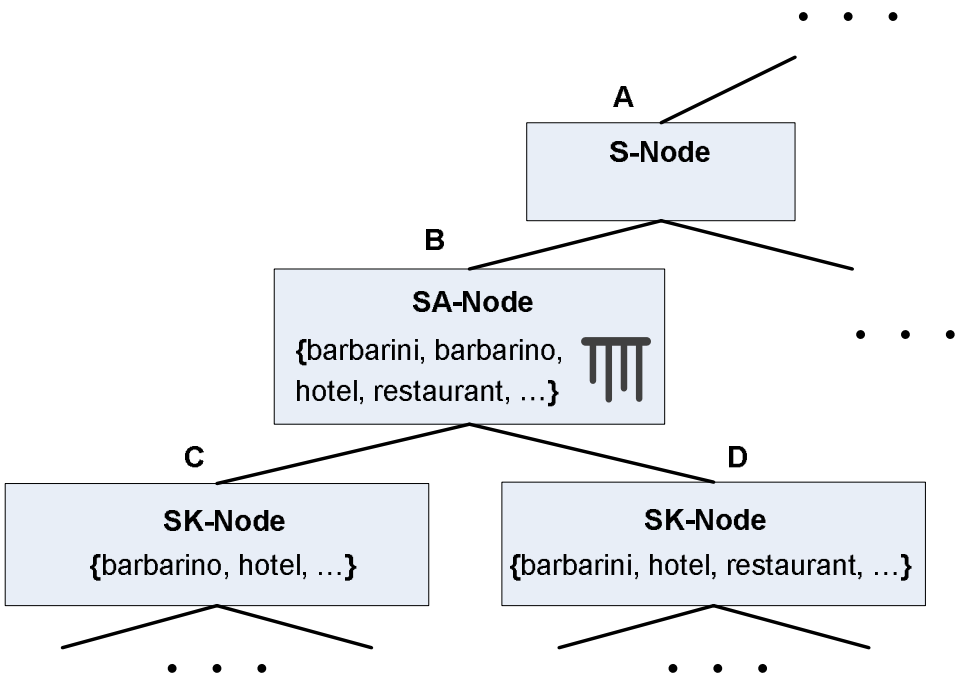


Figure 3.2: Exemplary portion of an LBAK-tree with approximate indexes at a fixed level.

Example: Figure 3.2 shows a portion of an LBAK-tree. We want to find objects in `New York` with keywords similar to `barbarene` and `resturant`, expressed as: $Q = \langle\{\texttt{New York}\}; \{\langle\texttt{barbarene}, 2\rangle, \langle\texttt{resturant}, 2\rangle\}\rangle$. Notice that the query keywords have typos. We use edit distance as the similarity measure, and 2 as the similarity threshold for both keywords. Let us assume that nodes A, B, C, and D satisfy the spatial condition `New York`. Throughout the traversal of the tree, we always check the spatial condition. We focus on how to utilize the approximate indexes and stored keywords to prune irrelevant subtrees. At the S-Node A, we only rely on the spatial condition for pruning. When we reach the

SA-Node B, we probe its approximate index to find keywords similar to `barbarene` and `resturant` according to the edit-distance threshold of 2. We can find two keywords similar to `barbarene` (namely, `barbarini` and `barbarino`), and one keyword similar to `resturant` (namely, `restaurant`). Once we visit the SK-Nodes C and D, we intersect their stored keywords with {`barbarini`, `barbarino`} and {`restaurant`}, respectively. Clearly, we can prune node C because it does not have the keyword `restaurant`. Since node D has the keywords `barbarini` and `restaurant`, we traverse its children. We repeat the process until we find the answers.

3.4 Placing Approximate Indexes at Variable Levels

In this section we study how to improve the solution described in Section 3.3 based on the following observations. First, our experiments showed that around 90% of the query time is spent on approximate-index lookups. Therefore, optimizing the placement of approximate indexes in the tree can greatly improve the average query time.

Second, the fixed level L is chosen without considering the local spatial distribution of the objects within each node. In many datasets, the spatial distribution of objects is skewed, and nodes at the same level can greatly vary in size. For instance, consider a sparse node n_1 in an R*-tree, e.g., a desert area in Nevada, where very likely a query overlaps with only few of n_1's children. When traversing the tree through n_1, it is better to rely only on the very selective spatial condition for pruning without considering its textual information. Thus, we prefer to build the approximate indexes at the descendants of n_1, where the local pruning power of the spatial condition becomes weaker. Our hope is that a query will only probe a few, small approximate indexes below n_1, if any. On the other hand, consider a dense node n_2, e.g., one representing New York city, where a query region is likely to overlap with many of n_2's children. If we build approximate indexes for n_2's children, a query will likely need to

probe all of them, which would be more expensive than probing one big approximate index at n_2. Therefore, we would rather build a single approximate index at n_2 to avoid the cost of many approximate-index lookups.

Our new method presented in this section allows the approximate indexes to appear at different levels in the tree as shown in Figure 3.3. The new method reduces both the overall space cost of the index and the average query time.

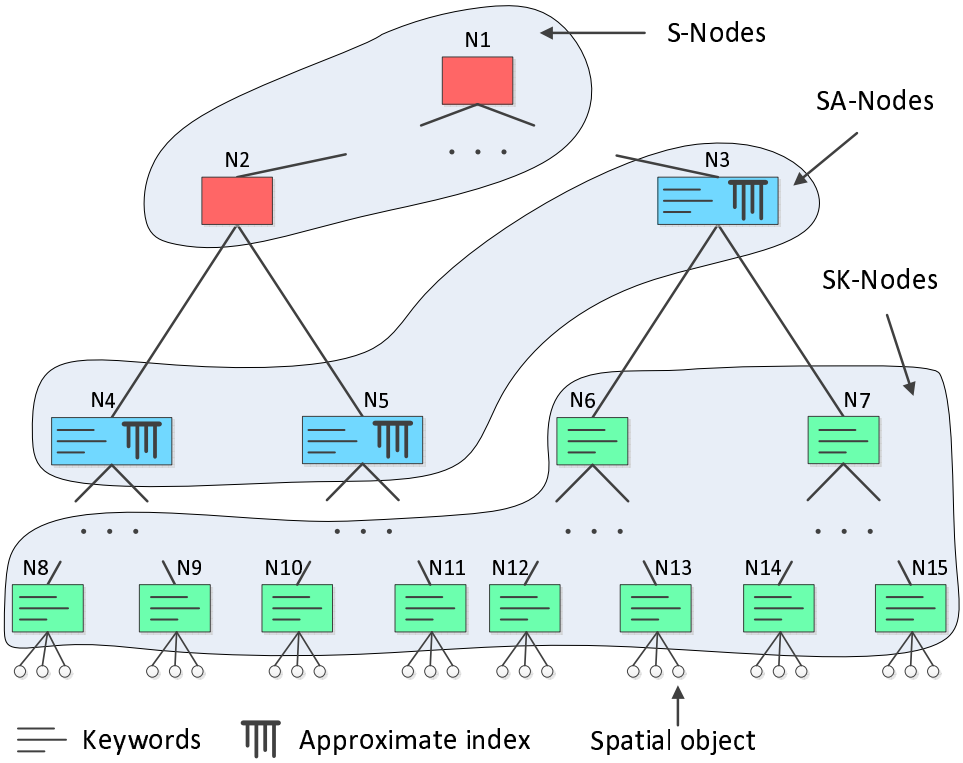


Figure 3.3: The LBAK-tree utilizing the local spatial distribution of objects.

3.4.1 Selecting Nodes for Approximate Indexes

Our problem is finding the optimal set of nodes that should have approximate indexes. It can be formulated as the following optimization problem: "Given an R*-tree and a space budget, choose nodes from the tree to store approximate indexes, such that the average query time of a given workload is minimized." We can show that this problem is NP-hard

by a reduction from the Knapsack problem to this problem, where each approximate index has a value (the average query time improvement that we obtain by building the index) and a weight (the space cost of the index).

Algorithm outline: We develop a greedy algorithm, called SelectSANodes, that traverses the tree top-down and tries to push approximate indexes down the most promising paths. The algorithm maintains a priority queue of nodes to be visited. The priority of a node n is the benefit of building multiple approximate indexes at its children as compared to building a single index at n. A detailed explanation of how to compute this benefit is presented in Section 3.4.2. For each visited node n, if the benefit of building multiple approximate indexes at n's children is negative, then the algorithm selects n to be an SA-Node, and it will not traverse its children. If the algorithm reaches a leaf node, it immediately selects the leaf to be an SA-Node. The algorithm terminates when the space budget is exhausted or there is no more benefit to push approximate indexes down the tree.

Pseudo-code: We now walk through the pseudo-code of SelectSANodes shown in Algorithm 6. We use W_n to denote the set of stored keywords at node n. The input of the algorithm is an R*-tree root r and a space budget S. We assume that the space budget is large enough to at least build an approximate index on the root's keywords W_r. We maintain a priority queue H of nodes to visit, ordering the nodes descending by their benefit. Initially, we call GetBenefit (r) to obtain the benefit of storing the approximate indexes at r's children (line 3), and push r with its benefit to the queue (line 4). The algorithm starts traversing the tree by popping the pair with the highest benefit to (n, B) (line 6). Next, we call GetSpaceCost (W_{n_i}) to compute the cost of building multiple approximate indexes at n's children (lines 7-10). We choose to build a single approximate index at n, and adjust the space budget S accordingly if one of the following is true: n is a leaf node, there is not enough space to build multiple approximate indexes at n's children, or there is no benefit of building multiple approximate indexes at the children (lines 11-13). Otherwise, we push n's

children to the queue for further evaluation (lines 14-19).

Algorithm 6: SelectSANodes

Input : An R*-tree root r;
A space budget S;
Output: A set of nodes R to build approximate indexes on;

```
1  Result set R ← ∅;
2  Priority Queue H ← ∅;
3  B ← GetBenefit(r);
4  H.push (r, B);
5  while H ≠ ∅ do
6  |  (n, B) ← H.pop();
7  |  S_c ← ∅; // space cost of children
8  |  foreach child n_i of n do
9  |  |  S_c ← S_c + GetSpaceCost(W_{n_i});
10 |  end
11 |  if n is a leaf node or S <= S_c or B <= 0 then
12 |  |  R.add(n);
13 |  |  S ← S− GetSpaceCost(W_n);
14 |  else
15 |  |  foreach child n_i of n do
16 |  |  |  B_i ← GetBenefit(n_i);
17 |  |  |  H.push (n_i, B_i);
18 |  |  end
19 |  end
20 end
21 return R
```

The search algorithm for answering queries based on an index built with SelectSANodes is the same as the one for the fixed-level solution, i.e., Algorithm 1 with the helper Algorithms 2, 3, 4, and 5.

3.4.2 Estimating Benefits

In this section, we develop a cost model for algorithm SelectSANodes to estimate the benefit of pushing approximate indexes down the tree based on the expected size overhead and lookup-time improvement. Recall that we use the benefits to order the nodes in the priority

queue.

Benefit of a node: Given a node n and its m children $n_1, \ldots, n_m$, the benefit of n, denoted as $b(n)$, is determined by the ratio of the lookup-time difference of a single approximate index at n and multiple approximate indexes at n's children, to their size difference. The following equation expresses this idea:

$$b(n) = \frac{pTime(n) - cTime(n)}{|pSpace(n) - cSpace(n)|}, \tag{3.1}$$

where "pTime" is the average query time of probing an approximate index at the parent, "cTime" denotes this time if the indexes were built at the children, and "pSpace" and "cSpace" are the corresponding space costs of the indexes.

Let W_n denote the set of stored keywords at node n, $s(W_n)$ denote the size of an approximate index on keywords W_n, and $t(W_n)$ denote the average lookup time for that approximate index. We define the benefit of node n as follows:

$$b(n) = \frac{p(n) * t(W_n) - \sum_{i=1}^{m} p(n_i) * t(W_{n_i})}{|s(W_n) - \sum_{i=1}^{m} s(W_{n_i})|}. \tag{3.2}$$

We weight the lookup time for a node n by $p(n)$, which denotes the probability of n satisfying the spatial condition of a query in a workload.

Space cost of an approximate index: Since the main space cost of a gram-based approximate index is the inverted lists, we focus on estimating their size.

Suppose we use q-grams in the approximate indexes for a positive integer q. Each keyword w to be inserted into the approximate index yields $|w| - q + 1$ q-grams. Let $G(w)$ be the bag of grams generated for w. For every gram in $G(w)$, we insert one element into its corresponding

inverted list. Let σ be the size of each inverted-list element. The keyword w contributes at most $(|w| - q + 1) * \sigma$ bytes to the size of the inverted lists. Note that this estimate is an upper bound on w's size contribution because we remove duplicate string ids from inverted lists. Thus, we estimate the size of an approximate index on a set of keywords W as:

$$s(W) = |W| * (\lambda - q + 1) * \sigma, \tag{3.3}$$

where λ is the average keyword length of a particular dataset.

Lookup time of an approximate index: The lookup time of an approximate index is a function of its size. We have experimentally determined that the cost of querying an approximate index is linear in the number of keywords. Table 3.1 shows the experimental data underlying this conclusion. We built four approximate indexes of different sizes (first column), and ran a workload of queries against them to determine the average lookup time (second column). In the third column we computed the slope of the line going through the point represented by the first row and the other corresponding point. Since the slope is roughly constant for different index sizes, we conclude that the average approximate-index lookup time grows linearly with its size. Thus, we can estimate the average lookup time of an approximate index on W keywords as:

$$t(W) = \beta * |W| + \alpha, \tag{3.4}$$

the slope β and the intercept α are implementation dependent and can be determined experimentally.

Size	Time (ms)	Slope
1	0.02	-
10000	0.207	0.000019
1M	22.253	0.000022
10M	210.152	0.000021

Table 3.1: Experimental data showing linear growth in lookup-time with size of an approximate index.

3.5 Exploiting Frequency Distribution of Keywords

Objects in a spatial dataset can share the same keyword. For example, many objects can have the keyword `hotel` in their textual descriptions. Moreover, very often some keywords appear more frequently than others, and their frequency distribution is skewed. For instance, in a dataset about business listings, a keyword `restaurant` is more likely to appear frequently in objects than `consulate`. In Section 3.4, we presented a cost-based solution that selects a set of nodes to store the approximate indexes. In this section, we further improve our cost-based index-construction procedure by exploiting the skewed distribution of keywords. The following enhancements can further reduce both the space cost of the approximate indexes and the average query time.

3.5.1 Index Construction

Intuitively, we want to reduce the number of keywords in the approximate indexes. We achieve this by removing frequent keywords from sibling nodes, and placing them in their common parent instead. As a consequence, approximate indexes on frequent keywords can now appear in any S-Node above an SA-Node. To further clarify, as shown in Figure 3.4, SA-Nodes contain approximate indexes on infrequent keywords, while some S-Nodes above an SA-Node hold approximate indexes on frequent keywords.

A keyword is considered frequent in a node n if the fraction of n's children having that

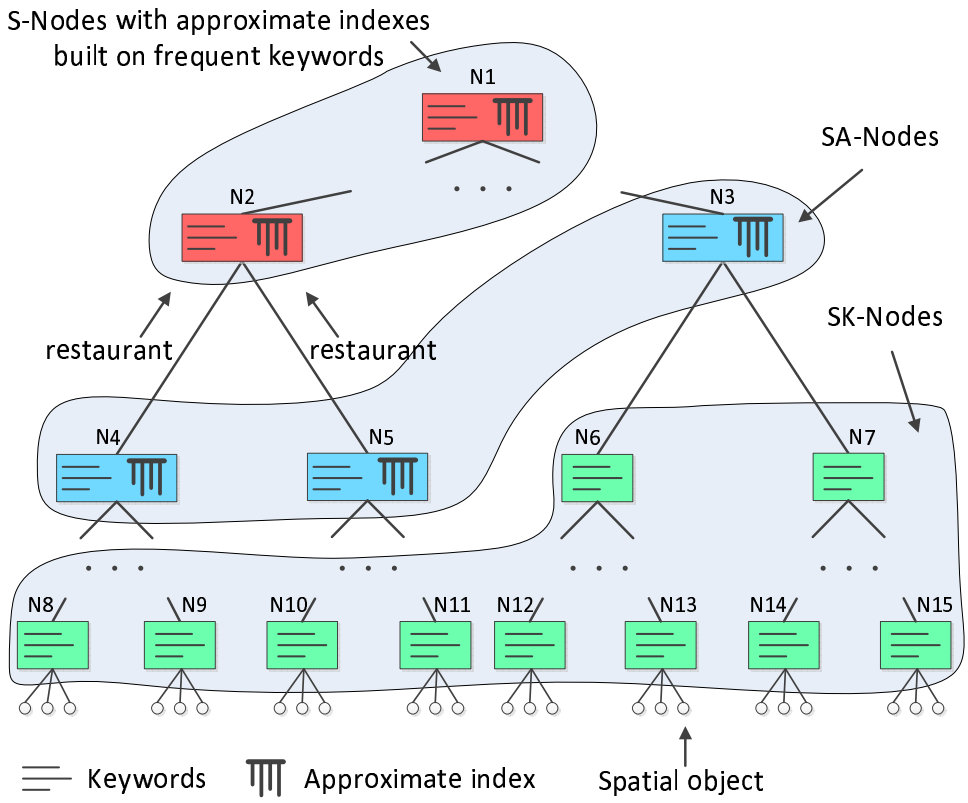


Figure 3.4: Improved LBAK-tree exploiting the skewed frequency distribution of keywords.

keyword exceeds a certain threshold, denoted by ω. For example, if $\omega = 0.9$, then a keyword is frequent if it appears in at least 90% of n's children. A small ω decreases the space cost of the approximate indexes. On the other hand, the average query time may increase because we could visit false-positive nodes, since not all of n's children actually contain the frequent keywords. Those false-positive paths will be pruned at SK-Nodes.

Algorithm outline: To discover the frequent keywords in the tree, we maintain for each node n two sets of keywords: a set of infrequent keywords W_n, and a set of frequent keywords F_n. We identify the frequent and infrequent keywords of the node n by examining its children. We say a child n_i has a particular keyword if the keyword occurs in W_{n_i} or F_{n_i}. If a keyword w is frequent at node n, then we remove w from the appropriate keyword sets in all of n's children. In this way, we ensure that popular keywords in a subtree only appear in the root of that subtree. The propagation of frequent and infrequent keywords

is performed bottom-up until the keyword sets of all nodes have been filled. The next step is to choose nodes to build approximate indexes on. We use a procedure similar to SelectSANodes from Section 3.4 but with a modified benefit function that distinguishes the frequent and infrequent keywords. Since the index-construction procedure requires only minor modifications, we omit its pseudo-code.

Benefit of a node: The new benefit of a node n is determined by the space and time cost of building an approximate index on $W_n \cup F_n$, versus building multiple indexes at n's children excluding the frequent keywords at n. We use the following variations for pTime, cTime, pSpace, and cSpace in the benefit function of Equation 3.1:

$$\begin{aligned} pTime(n) &= p(n) * t(W_n \cup F_n) \\ cTime(n) &= p(n) * t(F_n) + \sum_{i=1}^{m} p(n_i) * t((W_{n_i} \cup F_{n_i}) - F_n) \\ pSpace(n) &= s(W_n \cup F_n) \\ cSpace(n) &= s(F_n) + \sum_{i=1}^{m} s((W_{n_i} \cup F_{n_i}) - F_n) \end{aligned} \tag{3.5}$$

As before, we use $p(n)$ to denote the probability of n satisfying the spatial condition of any query in a workload.

3.5.2 Search Algorithm

Answering approximate-keyword queries on the improved index follows the common search Algorithm 1, with different implementations for the helper procedures InitSimilarKeywordSets, ProcSNode, and ProcSANode. We first probe the approximate index (if any) at the root node to get the similar query keywords that are frequent at the root (Algorithm 7). During the tree

Algorithm 7: InitSimilarKeywordSets

Input/Output: Same as Algorithm 2

```
1 C ← {∅, ∅, ..., ∅} // k empty sets
2 for i=1 to k do
3 |  C[i] ← r.GetSimKwds(w_i,δ_i);
4 end
5 return C
```

traversal, when we encounter an S-Node with an approximate index (on frequent keywords), we probe it for similar keywords (Algorithm 8). Note that at such an S-Node, we cannot prune subtrees based the textual condition, because we cannot guarantee to have gathered all similar keywords yet (we have only gathered those similar keywords that are frequent). At an SK-Node we probe its approximate index, and possibly prune its subtree based on the textual information (Algorithm 9).

Algorithm 8: ProcSNode

Input/Output: Same as Algorithm 3

```
1 for i=1 to k do
2 |  G_i ← C_i ∪ n_i.GetSimKwds(w_i)
3 end
4 G ← G_1, G_2, ..., G_k;
5 S.push (n_i, G);
6 return S
```

Algorithm 9: ProcSANode

Input/Output: Same as Algorithm 4

```
1 for i=1 to k do
2 |  G_i ← C_i ∪ n_i.GetSimKwds(w_i,δ_i)
3 end
4 if all G_i's ≠ ∅ then
5 |  G ← G_1, G_2, ..., G_k;
6 |  S.push (n_i, G);
7 end
8 return S
```

3.5.3 Incremental Maintenance of Indexes

We discuss how to incrementally maintain the proposed indexes in the presence of insertions and deletions. Insertion of new objects into an LBAK-tree proceeds as follows. We first insert the new object into a leaf according to the standard R*-tree insertion procedure (assuming no node-splits for now). We then propagate the keywords of the new object bottom-up. At an SK-Node we add the new keywords to its stored set of keywords. At an SA-Node we add the keywords to its approximate index. For the approach exploiting keyword frequencies, we must pay attention to add the new keywords to the appropriate keyword set, i.e., the one with frequent or infrequent keywords. Additionally, at an S-Node we check its children for new frequent keywords, and add them to its approximate index. We deal with R*-tree node-splits as follows. For the two new nodes generated by the split, we recompute their stored set of keywords (frequent and infrequent) by examining their children. If the nodes are SA-nodes, then we delete their approximate indexes, and give them a special "split" marker. After we have propagated all new keywords up to the root, we re-traverse the tree and rebuild approximate indexes at those nodes with special markers (and remove the markers). Note that the R*-tree may cause re-insertion of objects, causing multiple splits in different branches. Therefore, the second re-traversal of the tree is necessary if a split occurred. Deletions can be dealt with in a similar fashion. Before deleting a keyword at a particular node, we must examine all its children to ensure *none* of them have that keyword (or if the deletion causes a frequent keyword to become infrequent).

If many updates significantly change the distribution of keywords, we can reevaluate for each SA-Node whether pushing their approximate index up or down the tree would yield a benefit (according to the appropriate benefit function). We can use a similar procedure to deal with changing workloads, but must additionally modify the intersection probabilities ($p(n)$) in the benefit function to reflect the new workload.

3.6 Experiments

In this section, we present our experimental results on the proposed techniques. We evaluated the following variations: the fixed-level (FL) approach from Section 3.3 (e.g., "FL-0" means the approximate indexes are at the root level), the variable-level approach (VL) from Section 3.4, and the variable-level approach exploiting keyword-frequencies (VLF) from Section 3.5. In this chapter, we used the Flamingo Package [10] as our approximate string-search solution. We also compare our solutions with the MHR-tree [48]. We conducted all experiments on a machine with a four-core Intel Xeon E5520 2.26Ghz processor and 12GB of RAM, running a Ubuntu operating system. We implemented all algorithms (including the R*-tree) in C++ and compiled them with GCC using the "−O3" flag. We stored the index structures in main memory in order to achieve the goal of a high query throughput, as required by many online search engines. If not stated otherwise, we use a keyword-frequency threshold of $\omega = 1$ for algorithm VLF, and a branching factor of 40 for the R*-tree.

Datasets: We used two real, large datasets. The first one was a multimedia metadata collection extracted from Flickr pages, called "CoPhIR Test Collection" [15]. We processed the dataset to extract the photos taken in the U.S. based on their latitude and longitude values. Moreover, we used the keywords in the title, description, and tags of a photo as its textual attribute. The average number of keywords per object was 13.5. The final dataset had about 3.75 million objects. The total raw data size was about 500MB. We refer to this dataset as CoPhIR. The second dataset contained 20.4 million business listings in the U.S., obtained from Florida International University.[2] Each business listing had a longitude and latitude value, and a descriptive name consisting of three keywords on average. The total raw data size was about 4GB. We refer to this dataset as Business.

Queries: We generated a workload of 10,000 queries for each dataset as follows. We ran-

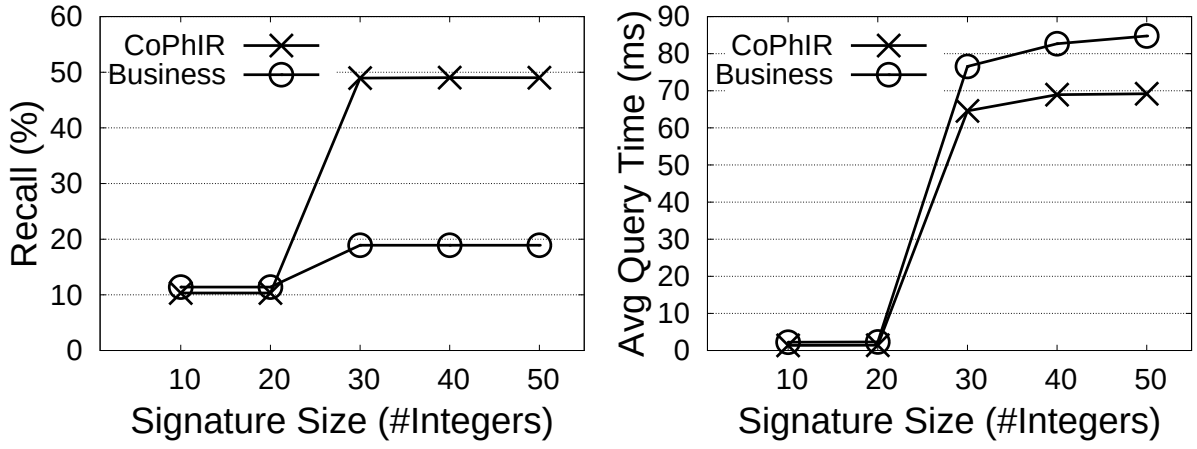
[2] http://n0.cs.fiu.edu/ihmc.com.forms.html

domly selected objects from the dataset and used their coordinates as the query-window center. We then created a 30km-by-30km query window around that center to reflect a spatial condition. For the approximate keyword condition, we randomly chose two of the keywords of the randomly chosen object. For most of the experiments we used a normalized edit-distance function and a similarity threshold of 0.8.

3.6.1 Comparison with MHR-Tree

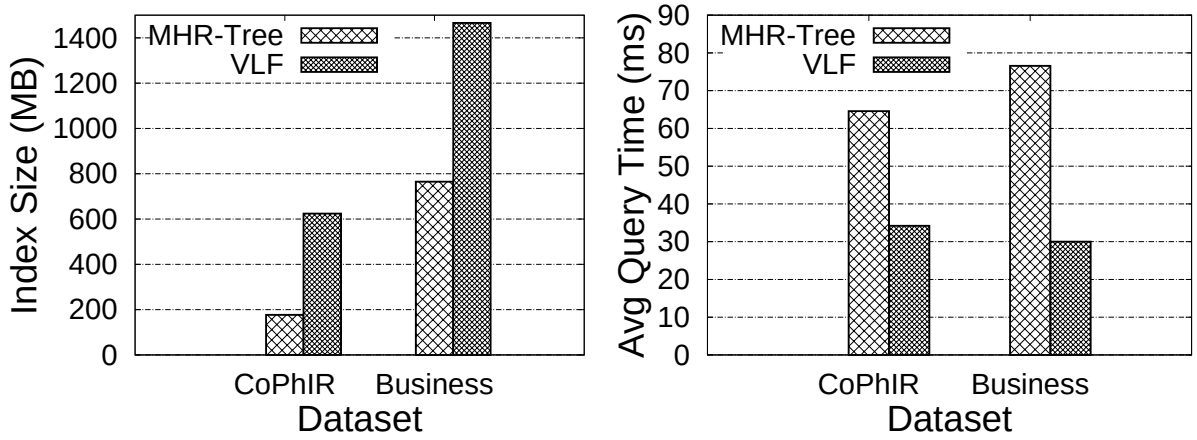
We first compared our LBAK-tree constructed using the VLF algorithm with the MHR-tree [48]. We used an edit distance threshold of 2 for both approaches. The main difference between these two approaches is that the MHR-tree uses a probabilistic signature scheme to represent textual information in tree nodes, whereas the LBAK-tree uses approximate indexes and keywords for that purpose. Since the min-wise signatures in the MHR-tree are probabilistic in nature, this approach could miss some answers. On the other hand, the size overhead of these signatures is very small. Figure 3.5 shows the main issues with MHR-tree. First, the recall of MHR-tree shown in Figure 3.5(a) were constantly below 50%, which may not be acceptable for many applications. Second, as we increased the signature size in order to achieve a higher recall, the query time also increased (Figure 3.5(b)). The reason is that the pruning power of the min-wise signatures is limited, and the approximate keyword condition is validated at the leaf level, leading to many edit-distance computations as the recall increases.

In Figure 3.6 we compare VLF with MHR-tree using 30 signatures for each tree node. Clearly, the merit of MHR-tree lies in a comparably small index size, due to the compact signatures. However, we see that VLF significantly outperformed MHR-tree in terms of query time. Note that as shown in Figure 3.5(b), the query time of MHR-tree will likely increase if we give it more space.



(a) Recall (b) Query Time

Figure 3.5: Recall and query time of MHR-tree with increasing signature size.



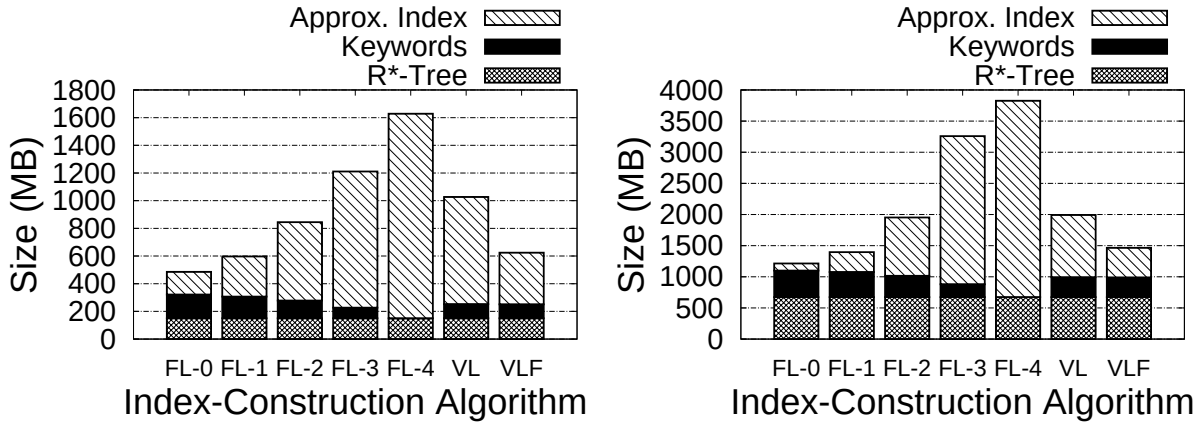
(a) Index Size (b) Query Time

Figure 3.6: Comparison of VLF with MHR-tree using 30 signatures for each node.

3.6.2 Index Size and Query Time

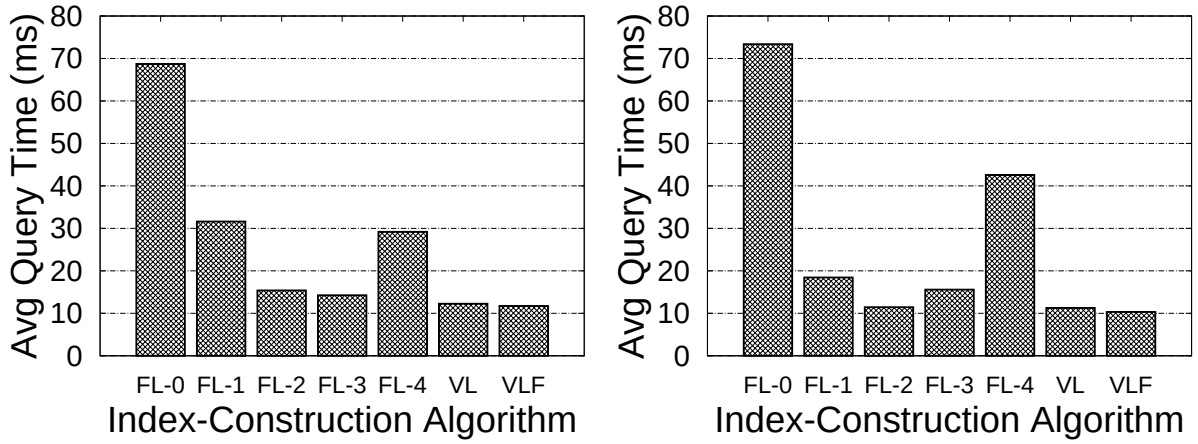
Figure 3.7 shows the sizes of the individual index components for various construction algorithms. Figure 3.8 shows the corresponding query times. For the algorithms VL and VLF, we report the minimum index size required to achieve the best query performance.

The fixed-level solutions FL-0 to FL-4 show a clear size-trend in Figure 3.7: as we pushed the approximate indexes down the tree, their space requirement increased because of redundant keywords in adjacent nodes. On the other hand, the query times decreased because we



(a) CoPhIR (b) Business

Figure 3.7: Sizes of index components.



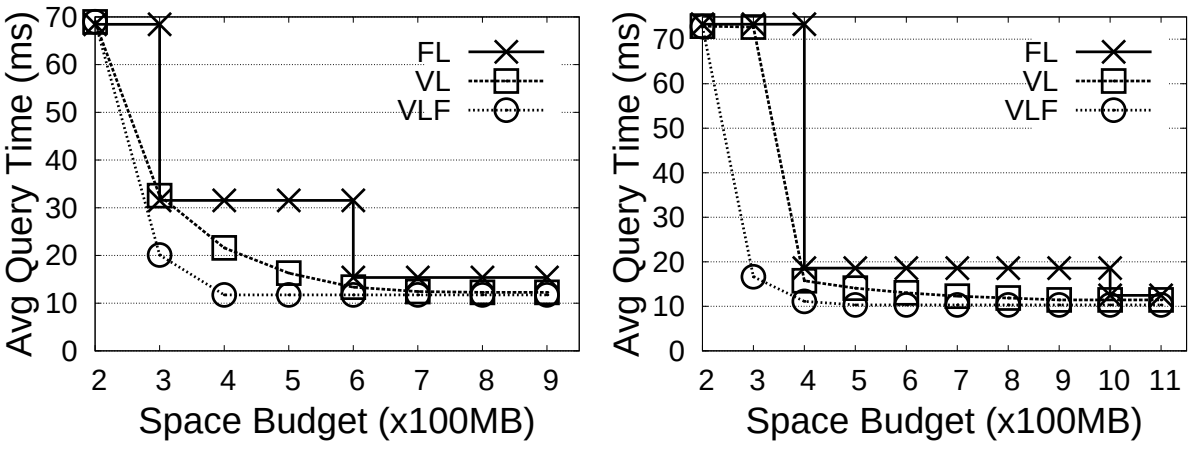
(a) CoPhIR (b) Business

Figure 3.8: Query time by index-construction method.

probed few, smaller indexes rather than one bigger index. But the query time eventually increased when pushing the indexes further down (e.g., FL-3 in Figure 3.8(b)), again, because of keyword redundancies. The space overhead for the approximate indexes for algorithm VL was large compared to the R*-tree and keywords, but the query time was good. Compared to VL, algorithm VLF pushed frequent keywords up the tree to reduce the space by half without sacrificing query performance.

3.6.3 Space Budget

In Figure 3.9 we show the effect on query performance when giving our index-construction methods FL, VL, and VLF a space budget for building approximate indexes.



(a) CoPhIR (b) Business

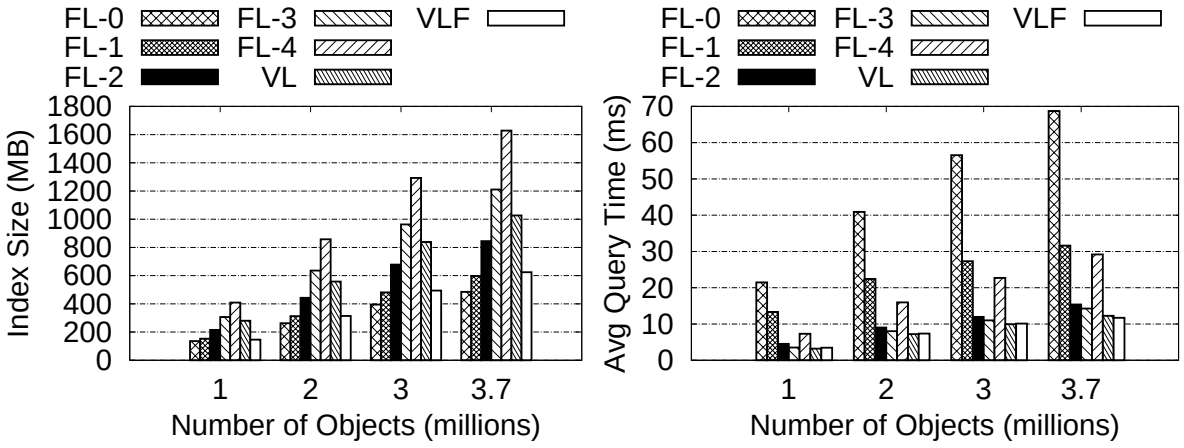
Figure 3.9: Query time with increasing space budget for approximate indexes.

The fixed-level solution, FL, exhibited "jumps" when given enough space to push down the approximate indexes one more level, improving the query time. Note that at even higher space budgets not shown in the figure, the query time of FL will eventually increase, due to the cost of probing many, smaller approximate indexes at a lower level. VL's and VLF's curves are smoother than FL's because they have more flexibility in placing the approximate indexes. VL's curve meets FL's curve at some points because their performance is limited by redundant keywords residing in many approximate indexes. This observation is supported by VLF outperforming both LF and VL at the points where LF and VL meet. In summary, VLF offers good query performance at significantly less space than the other two methods.

3.6.4 Scalability

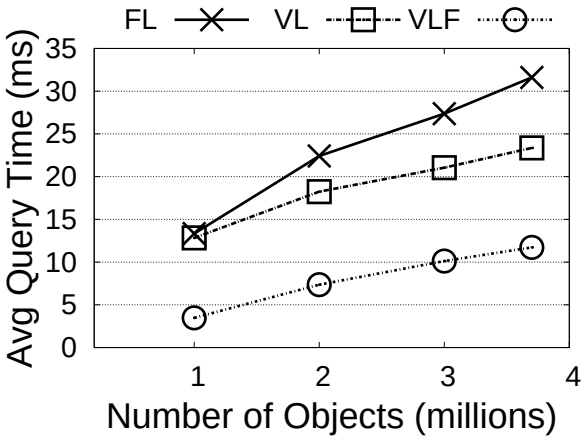
We changed the number of objects indexed by different LBAK-tree variants. Figures 3.10(a) and 3.11(a) show the total index sizes. For algorithms VL and VLF, we report the minimum

index size required to achieve the best query performance. Figures 3.10(b) and 3.11(b) show the corresponding best query times. To emphasize the merit of VLF, we created Figures 3.10(c) and 3.11(c) as follows. We determined the minimum index size for VLF to achieve the best query time, and used that size as a space budget for FL and VL. As a result, FL's level could vary from point to point.



(a) Total Index Size.

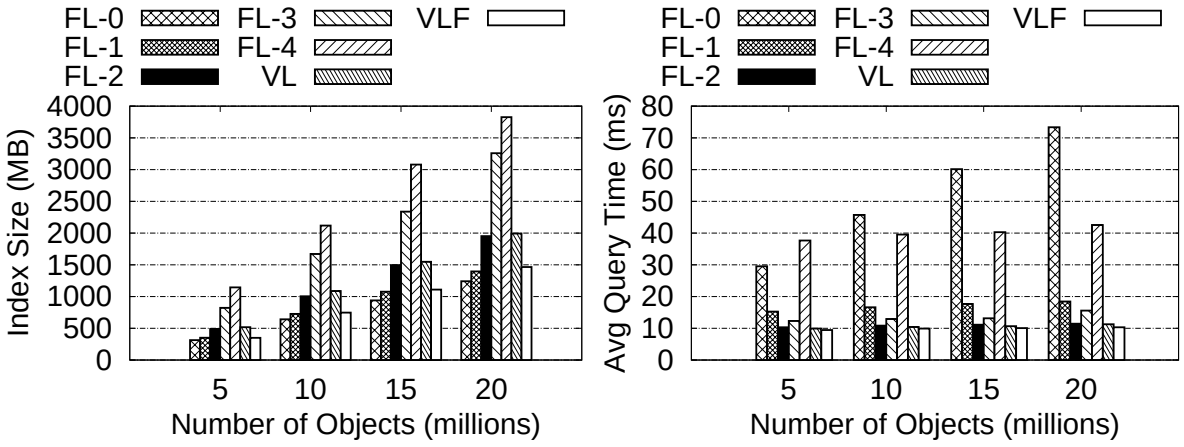
(b) Query Time.



(c) Query Time vs. VLF.

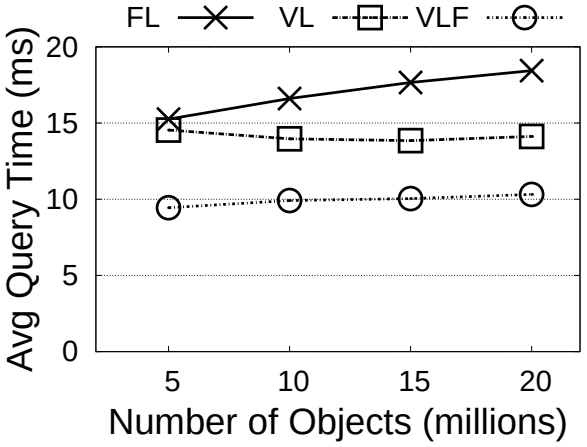
Figure 3.10: Index size and query time with varying numbers of indexed objects on CoPhIR.

For both datasets, we observe a linear trend for the index size and query time (Figures 3.10 and 3.11(a,b)). The fixed-level approaches show a space-time tradeoff with the level. As we pushed the approximate indexes down the tree the index size increased because the number of duplicate occurrences of keywords in approximate indexes increased. However, the query



(a) Total Index Size.

(b) Query Time.



(c) Query Time vs. VLF.

Figure 3.11: Index size and query time with varying number of indexed objects on Business.

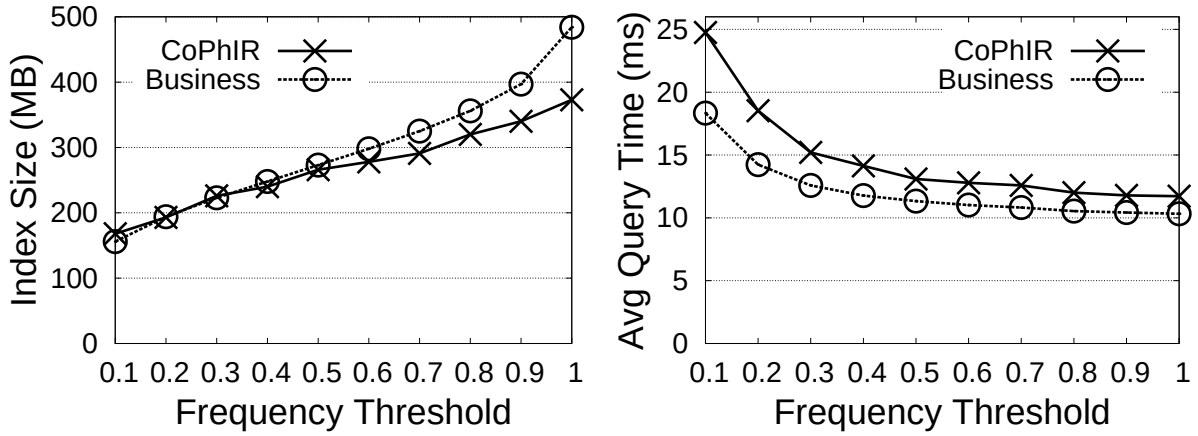
time improved until FL-2 for 3.11(b) and FL-3 for 3.10(b), but then sharply degraded at lower levels. The reason is that the cost of probing multiple smaller approximate indexes became higher than probing a few larger ones. The effect of duplicate keywords can also be observed by comparing VL and VLF. VLF consistently performed better both in space and time, because it effectively minimized the redundancy in keywords. Notice that space and time increased more rapidly on the CoPhIR dataset than on the Business dataset. The main reason is that the objects in the CoPhIR dataset had, on average, a higher number of keywords in their description. The longer textual description contributed to the index size, and increased the chance of intersecting with query keywords. In comparison, the Business

dataset was sparse in the textual dimension, making queries highly selective and rendering their performance insensitive to the number of indexed objects.

Finally, let us examine Figures 3.10(c) and 3.11(c). Here, the difference between VL and VLF is clearer due to the smaller scale of the Y-axis (as compared to the other figures). We see that when given the same amount of memory, VLF clearly outperformed the other methods, though they could achieve a comparable performance with more memory (Figure 3.9).

3.6.5 Keyword-Frequency Threshold

Algorithm VLF uses a threshold ω to decide whether a keyword is frequent or not (Section 3.5). Intuitively, a threshold of $\omega = 0$ (i.e., every keyword is considered to be frequent) would produce an LBAK-tree with one approximate index at the root. On the other hand, a threshold of $\omega > 1$ (no keyword is frequent) would produce an LBAK-tree similar to the one generated by VL. We ran experiments with varying ω values, the results of which are shown in Figure 3.12.



(a) Approximate-Index Size

(b) Query Time

Figure 3.12: Effect of keyword-frequency threshold.

We observe a clear space-time tradeoff with the keyword-frequency threshold. As we increased the threshold, we pushed more keywords to lower levels in the tree, causing space

overhead due to infrequent keywords being duplicated at multiple nodes. On the other hand, increasing the threshold decreased the query time, because of the following two effects: (1) we traversed fewer false-positive branches that did not actually have a keyword deemed frequent at its parent, and (2) the total cost of probing a few smaller indexes at a node's children could be less than probing one big index at the node itself. The VLF algorithm will try to push indexes only down these beneficial paths.

3.7 Conclusions

In this chapter we have presented an index structure called LBAK-tree to answer location-based approximate-keyword queries. We showed how to combine approximate indexes efficiently with a tree-based spatial index. We developed a cost-based algorithm that selects tree nodes to store approximate indexes. Moreover, we improved the techniques to exploit the frequency distribution of keywords, further reducing the index size and query time. Finally, we conducted extensive experiments to show the efficiency of our techniques.

Chapter 4

A General Purpose LSM-Based Indexing Framework

4.1 Introduction

One implicit assumption that we made when designing our solution in the previous chapter is that queries are much more frequent than inserts. For many workloads, such as those handled by business-listing search engines, this can be a valid assumption. For instance, Forbes [45] reported that approximately half a million small businesses get started each month in the US. Ingesting this small amount of data can be easily handled by conventional spatial index structures. One popular choice for a spatial index structure that has been adopted by many database systems is R-tree, mainly because it is a universal index that can support different geometric shapes (such as points, circles, lines, polygons, etc.), and also because of its low read latency.

As we mentioned in Chapter 1, there is a new set of emerging applications that involve much more insert-intensive workloads. For example, Facebook has reported that the average

number of content items shared daily as of May 2013 is 4.75 billion [21], while Twitter users are posting around 500 million tweets daily [43], and those rates are expected to rise exponentially. Using a conventional index structure such as B^+-tree to ingest this massive amount of data has shown to be ineffective because of its usage of in-place writes to perform updates, resulting in costly random writes to disk. Consequently, popular NoSQL systems such as [2, 3, 4, 17, 41] have adopted LSM-trees as their storage structure to replace the conventional B^+-tree. LSM-trees amortize the cost of writes by batching updates in memory before writing them to disk, thus avoiding random writes. This benefit comes at the cost of sacrificing read efficiency, but, as shown in [41], these inefficiencies can be mostly mitigated.

Since much of today's newly generated data is geo-tagged, efficient methods for ingesting and searching rapidly arriving spatial data are also required. However, similar to B^+-tree, R-tree does not scale well for insert-intensive workloads since it is also an update-in-place structure. Therefore, it is desirable to have an LSM version of R-tree that can efficiently handle spatial data.

In this chapter, we present the storage system implemented in the AsterixDB system. Within AsterixDB's storage system is a general framework for converting conventional indexes to LSM-based indexes, allowing higher data-ingestion rates. We show that converting the R-tree (and other indexes without a total order among their entries) to an LSM index is non-trivial if the resultant index is expected to have performant read and write operations. The framework acts as a coordinating wrapper for existing, non-LSM indexes so that designing and implementing specialized indexes from scratch can be avoided, saving design and development time. We also describe the concurrency and transaction key ideas that can enforce the ACID properties across multiple heterogeneous LSM indexes.

The rest of the chapter is organized as follows. First, we describe the framework for converting conventional index structures to LSM indexes. Then we explain some of the concurrency and transaction related techniques that we use to enforce the ACID properties across multiple

heterogeneous LSM indexes. Finally, we present performance evaluation results of AsterixDB focusing on the LSM-based storage system, including a detailed performance study for its spatial LSM indexes.

4.2 Secondary Index LSM-ification

This section describes a generic framework for converting a class of indexes (e.g., conventional B^+-trees, R-trees, and inverted indexes) with certain, basic operations into LSM-based secondary indexes. The framework provides a coordinating wrapper that orchestrates the creation and destruction of LSM components and the delegation of operations to the appropriate components as needed. Using the original index's implementation as a component, we can avoid building specialized index structures from scratch while enabling the advantages of an LSM index.

Applying the key ideas behind LSM-trees to index structures other than B^+-trees turns out to be non-trivial and must be done carefully in order to achieve a high-write throughput. In particular, we start by explaining why it is a challenging process to reconciliate index entries when "LSM-ifying" indexes such as the R-tree and inverted index.

4.2.1 Reconciliation Challenges

Entries in a log-structured data structure are descriptions of the state for a particular key. Since modification operations such as inserts and deletes on a log-structured data structure always produce a new state (entry) for a particular key, there must be a process for determining the correct state of the key amongst the existing states. This is called *reconciliation*. We define the correct state of the key to be the most recent state since that will provide the usual semantics expected of index operations.

The process of reconciliation was briefly outlined for LSM B$^+$-trees in Section 2.1, whereby entries corresponding to a single key were grouped by virtue of being returned in the native, key order of the underlying index. However, this luxury is only possible when the underlying index return entries grouped by their key. To explore the difficulties of reconciliation for an index lacking this grouping property, such as R-tree, we analyze typical operations on an LSM index.

Search and merge operations: When performing a range scan on an LSM index, each component must be searched using the same predicate since matching entries may be distributed across the components. Each entry returned as the product of searching a component must be reconciled since it may or may not be the correct state for a particular key. Recall that in the LSM B$^+$-tree, this reconciliation is simple: in a manner reminiscent of the merge step of merge-sort, we simply collect the next entry from each sub-cursor that has a matching key, retaining only the most recent version of the entry and discarding the others. In general, since the entries being returned from an LSM index's sub-cursors may not be totally ordered (e.g., from an R-tree sub-cursor), this approach cannot be used. One possible approach could be to perform a search for every key returned from a sub-cursor on all newer components, but this would be prohibitively expensive since it may incur multiple expensive, multi-path searches in an R-tree or a multitude—one for each token—of searches in an inverted index. Alternatively, all entries can be returned and then sorted using any comparator that guarantees that exactly equal values come together (e.g., byte-based sorting).

Insert and delete operations: Since modification operations on an LSM index produce entries in the in-memory component, it is possible to introduce multiple states for a particular key in a single component. The existence of multiple states in a single component could prohibit the possibility of determining the most-recent state unless additional information is carried in the entries. One example of such information is a sequence number. But, if the overhead of storing sequence numbers with each entry is to be avoided, reconciliation should

be performed at modification time to ensure that at most one state per key resides in a single component. In a B$^+$-tree component, this is simple: when modifying (e.g., deleting) an entry, the previous version of the entry will be discovered when traversing the tree. Unfortunately, this is not true, for example, in an R-tree, since the location of an entry being inserted is not uniquely determined. One possible solution is to perform a search to find any identical keys before each modification, but doing so can greatly reduce the ingestion rate of an LSM R-tree since repeated multi-path, spatial searches will incur overhead. Furthermore, this issue is exacerbated when LSM-ifying an inverted index. Keys in an inverted index are usually $\langle$token, id$\rangle$ pairs. Thus, there exists a state for every token inserted into the index. In order to remove a document from the inverted index, the document will need to be retokenized and a delete entry will need to be added for every token that was produced, making both the delete operation and reconciliation during search and merge operations exorbitantly expensive.

4.2.2 Efficient Reconciliation

The issue of efficiently reconciling can be viewed equivalently as the problem of invalidating a single entry. We can achieve this by maintaining data and control entries separately, in two different structures. Specifically, new incoming data entries can be batched into the underlying in-memory index, while control entries, representing deleted entries, are batched into an in-memory *B$^+$-tree*, called the *deleted-key B$^+$-tree*. The deleted-key B$^+$-tree is essentially a delete-list that stores the primary keys of the deleted entries. We choose to use a B$^+$-tree, but any data structure that provides similar operations in an efficient manner could be used.

Storing control entries separately avoids expensive multi-path searches for control entries, since only the delete-list needs to be searched. For the inverted index, it also avoids expensive tokenization and token control entry insertion for delete operations, since only a single entry needs to be made in the deleted-key B$^+$-tree.

An LSM index employing this approach maintains two data structures in memory: the original index structure and the deleted-key B^+-tree. Both structures are tightly coupled and will always be flushed to disk together, yielding disk components consisting of the original index structure and a deleted-key B^+-tree. As such, we refer to the pair of data structures as a single component. Figure 4.1 shows a secondary LSM R-tree index employing the above design, alongside a secondary LSM B^+-tree that does not use the above design. We omit the use of the above design for LSM B^+-trees since the B^+-tree naturally groups entries, serving as an optimization. Each index has one in-memory component and one disk component. Note that the deleted-key B^+-trees are only used to validate entries through point lookups. Thus, as an optimization, for every disk instance of the deleted-key B^+-tree, a Bloom filter containing the entries of the tree is maintained in memory to reduce the need to access pages of the deleted-key B^+-tree on disk. Index operations on this new structure are explained in Section 4.2.4.1.

4.2.3 Imposing a Linear Order

For a certain class of indexes lacking a total order, it is possible to impose a linear order on the index entries. For example, a Z-order curve or Hilbert curve can be used to impose a linear order on the index entries of an R-tree [28]. This is especially useful for bulk-loading since the ordering can be applied during flushes and merges of LSM components. By doing so, a hybrid approach to efficient reconciliation becomes possible. In this hybrid strategy, incoming inserts and deletes are still maintained in two different in-memory data structures as explained above. However, when flushing the in-memory component to disk, the data and control (anti-matter) entries of the in-memory index structure and the deleted-key B^+-tree can be sorted based on the ordering criterion (e.g., Hilbert curve) and merged to form a single disk component that consists of the single, original index structure. This hybrid approach can benefit the performance of the LSM index in many aspects. First, the sorted

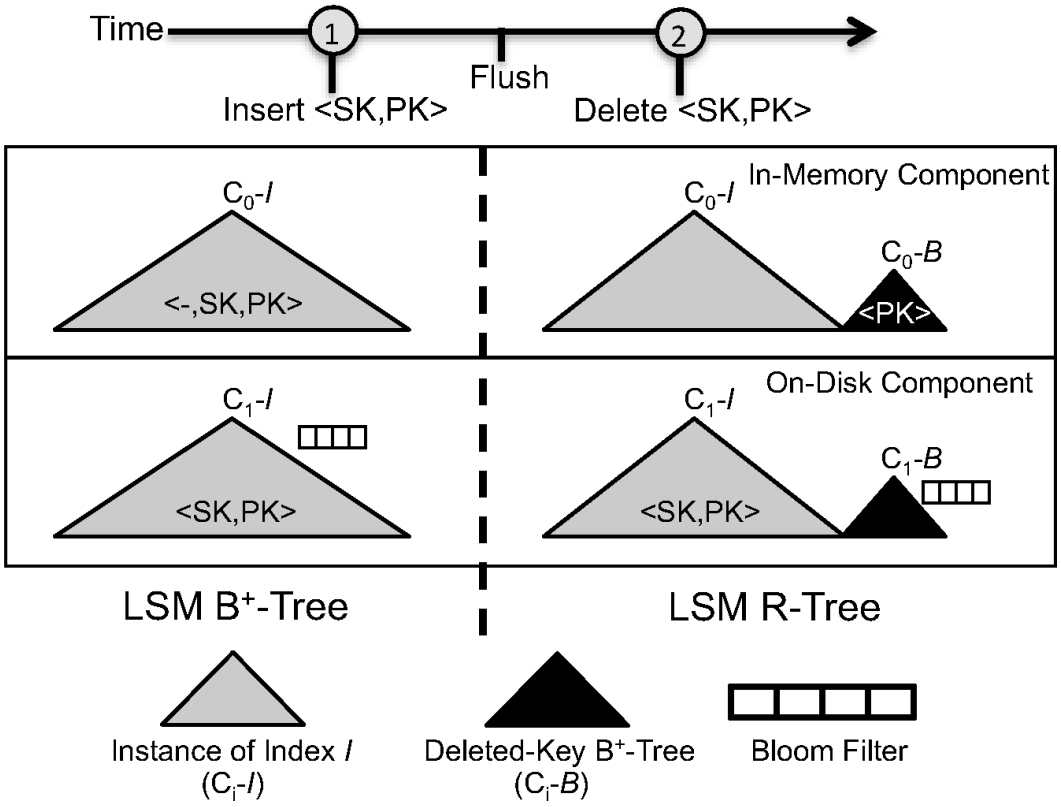


Figure 4.1: The final state of insertion, flushing, then deletion applied to a secondary LSM R-tree and a secondary LSM B$^+$-tree. Both indexes are storing entries of the form $\langle SK, PK \rangle$, where SK is a secondary key and PK is the associated primary key. The LSM R-tree handles deletion by inserting the primary keys of the deleted entries in its deleted-key B$^+$-tree, while the LSM B$^+$-tree handles it by inserting a control entry, denoted by $\langle -, SK, PK \rangle$, into its memory component.

mini-cursors design can now be used to search the disk components of the index, allowing deleted entries to be ignored on the fly. Second, since the participating disk components are already ordered based on the sorting criteria, the merging process is simple and efficient: scan the component's leaves and output the sorted entries into a new disk component. Third, the newly-formed, merged component will retain the original ordering. For the R-tree, retaining the ordering improves the performance of searches and generally produces a higher quality index as shown in [28].

4.2.4 LSM Generalization

Based on the above design, we can provide a generic framework to "LSM-ify" the secondary indexes of a system. Let entries be of the form $e = \langle SK, PK \rangle$ where SK is a secondary key and PK is the associated primary key. The framework requires that the following primitive operations be supported by every secondary, non-LSM index that is to be converted into an LSM index:

1. Bulk-load: Given a stream of entries $e_1, \ldots, e_m$, the bulk load operation creates a single disk component of the index. The bulk load operation is used for two reasons: to flush an in-memory component of the index into a new disk component and to merge multiple disk components into a single disk component.
2. Insert: This operation inserts a given entry e into the index.
3. Delete: The delete operation removes a given entry e from the index.

With those primitive operations, we describe the basic operations of the proposed general LSM index design and of the LSM B$^+$-tree, which are both implemented in AsterixDB.

4.2.4.1 General LSM Index Operations

Insert and delete operations: Since entries in the deleted-key B$^+$-tree refer only to disk components, inserted entries never need to be reconciled with deleted entries upon insertion. Thus, an insertion is performed by inserting an entry $e = \langle SK, PK \rangle$ into the in-memory index without the need to search for control entries in the in-memory deleted-key B$^+$-tree. To complement this, deletes are performed by deleting the given entry directly from the index structure and adding a control entry $e' = \langle PK \rangle$ to the in-memory deleted-key B$^+$-tree.

Search and merge operations: When answering a search query, all components of the LSM index must be examined. An entry $e = \langle SK, PK \rangle$ is part of the result set if it satisfies two conditions:

1. SK satisfies the query predicate, and
2. There does not exist a control entry $e' = \langle PK \rangle$ in the deleted-key B$^+$-tree of a newer component than e's component.

When merging components of an LSM index, the deleted-key B$^+$-trees of the participating components are also searched in the same manner to keep deleted entries out of the merged component.

4.2.4.2 LSM B$^+$-Tree Operations

Insert and delete operations: Insert operations are preceded by a search to check if an insert entry exists with the same key. If an insert entry already exists, an error will be thrown. Otherwise, the entry $e = \langle SK, PK \rangle$ will be added to the in-memory component. Delete, on the other hand, simply inserts an anti-matter entry $e' = \langle -, SK, PK \rangle$ into the in-memory component. If the in-memory component contains an entry that has the same key as the key being inserted or deleted, then the existing entry is simply replaced with the new entry (except, if the existing entry is an insert entry and the operation to be performed is an insert, which will throw an error, as mentioned above, to enforce primary key uniqueness).

Search and merge operations: When answering a range query, all components of the LSM B$^+$-tree must be examined. An entry $e = \langle SK, PK \rangle$ in a component is part of the result set if it satisfies two conditions:

1. SK satisfies the query predicate, and

2. There does not exist a more recent control entry
   $e' = \langle -, SK, PK \rangle$.

A point lookup query can be optimized to search components sequentially from newest to oldest until a match is found.

Similarly, when merging multiple components of an LSM B$^+$-tree, the participating components are searched as in the range query to avoid putting deleted entries into the merged component.

4.2.5 Current Implementation of Indexes in AsterixDB

Data in AsterixDB is partitioned using hash-based partitioning on the dataset's primary key. All of the dataset's secondary indexes are local as in most shared-nothing parallel databases. Thus, secondary index partitions refer only to data in the local primary index partition. AsterixDB currently supports LSM-based B$^+$-trees, R-trees, inverted keyword, and inverted ngram secondary indexes. Note that we implemented two versions for the R-tree: one that keeps anti-matter entries inside the disk R-trees (AMLSM R-tree), and another that uses a deleted-key B$^+$-tree with every disk component for maintaining control entries (LSM R-tree). We choose the LSM R-tree to be the default spatial index in AsterixDB, but we intend to switch to the AMLSM R-tree as a result of the performance evaluations performed for this chapter.

In AsterixDB, secondary index lookups are routed to all of a dataset's partitions since matching data could be in any partition. These lookups occur in parallel and primary keys are the result of these lookups. The resulting primary keys are then used to lookup the base data from the primary index, sorting the keys first to access the primary index in an efficient manner. Inserting and deleting entries from a dataset requires that each of the datasets'

indexes be mutated, since AsterixDB maintains consistency across all indexes of a dataset. The primary index is always updated first, followed by updating the secondary indexes, if any.

In the current release (Release 0.8.6), AsterixDB provides three different merge policies that can be configured per dataset: *constant*, *prefix*, and *no-merge*. The constant policy merges disk components when the number of components reaches some constant number k, which can be configured by the user. While the prefix policy (the default for AsterixDB) relies on component sizes *and* the number of components to decide which components to merge. Specifically, it works by first trying to identify the smallest ordered (oldest to newest) sequence of components such that the sequence does not contain a single component that exceeds some threshold size M and that either the sum of the component's sizes exceeds M or the number of components in the sequence exceeds another threshold C. If such a sequence of components exist, then all there components are merged to form a single component. Finally, the no-merge policy simply never merges disk components. AsterixDB also provides a DML statement that allows compacting a dataset and all its indexes by merging all their disk components.

4.3 Transactions Across Multiple LSM Indexes

In this section, we describe the key ideas that are used by AsterixDB to implement transactions. We refer the interested reader to [7] for additional details, such as about transaction support in AsterixDB.

4.3.1 Providing Record-Level ACIDity

AsterixDB supports record-level, ACID transactions across multiple heterogeneous LSM indexes in a dataset. Transactions begin and terminate implicitly for each record inserted, deleted, or searched while a given DML statement is being executed. This is similar to the level of transaction support found in today's NoSQL stores. Since AsterixDB supports secondary indexes, the implication of this transactional guarantee is that all the secondary indexes of a dataset are consistent with the primary index.

AsterixDB does not support multi-statement transactions. In fact, a DML statement that involves multiple records can itself involve multiple independent record-level transactions. A consequence of this is that, when a DML statement attempts to insert 1000 records, it is possible that the first 800 records could end up being committed while the remaining 200 records fail to be inserted. This situation could happen, for example, if a duplicate key exception occurs as the 801st insertion is attempted. If this happens, AsterixDB will report the error as the result of the offending DML insert statement and the application logic above will need to take the appropriate actions needed to assess the resulting state and to clean up and/or continue as appropriate.

AsterixDB utilizes a no-steal/no-force buffer management policy and write-ahead-logging (WAL) to implement a recovery technique that is based on LSM disk component shadowing and index-level logical logging. Whenever a new disk component is created, through a flush or merge operation, a *validity* bit is atomically set in a metadata page of the component to indicate that the operation has completed successfully. During crash recovery, any disk component with an unset validity bit is considered invalid and removed, ensuring that the data in the index is physically consistent. By using a no-steal policy, only committed operations from in-memory components are selectively replayed during crash recovery and there is no need for an undo phase, unlike the repeating history step in ARIES [32].

4.3.2 Concurrency Considerations

AsterixDB's concurrency control is based on two-phase locking (2PL) and follows the latch protocols described in ARIES/KVL [31] for the B$^+$-tree and GiST [29] for the R-tree. Transaction locks are only acquired on primary keys when accessing a primary index. Locks are never acquired when accessing a secondary index, which could lead to inconsistencies when reading entries of a secondary index that are being altered concurrently. To prevent these inconsistencies, secondary index lookups are always validated when fetching the corresponding records from the primary index.

Keys are only locked during insert, delete, and search operations. Flushing an in-memory component and merging disk components do not set locks on the entries of the components, nor do they generate log records.

4.3.3 An Example

Here we provide a detailed example that depicts the lifecycle of inserted and deleted records, including the locking behavior, when using our indexing framework.

As shown in Figure 4.2, there is a dataset with records consisting of three fields: *Id* (an integer key for the primary LSM B$^+$-tree, named *PIdx*), *Loc* (a two-dimensional point key for the secondary LSM R-tree, named *SIdx*), and *Name*. A component of *PIdx* is denoted as PC_i, where PC_0 represents the in-memory component and PC_1 represents the oldest disk component. A component of *SIdx* consists of an R-tree and a deleted-key B$^+$-tree, each of which is denoted as SC_i and BC_i as is done for the primary index. Older disk components have smaller values of i.

The timeline shows a sequence of five consecutive operations and their side-effects (e.g., flush and merge), where each operation is executed by a single transaction, starting from T1 to

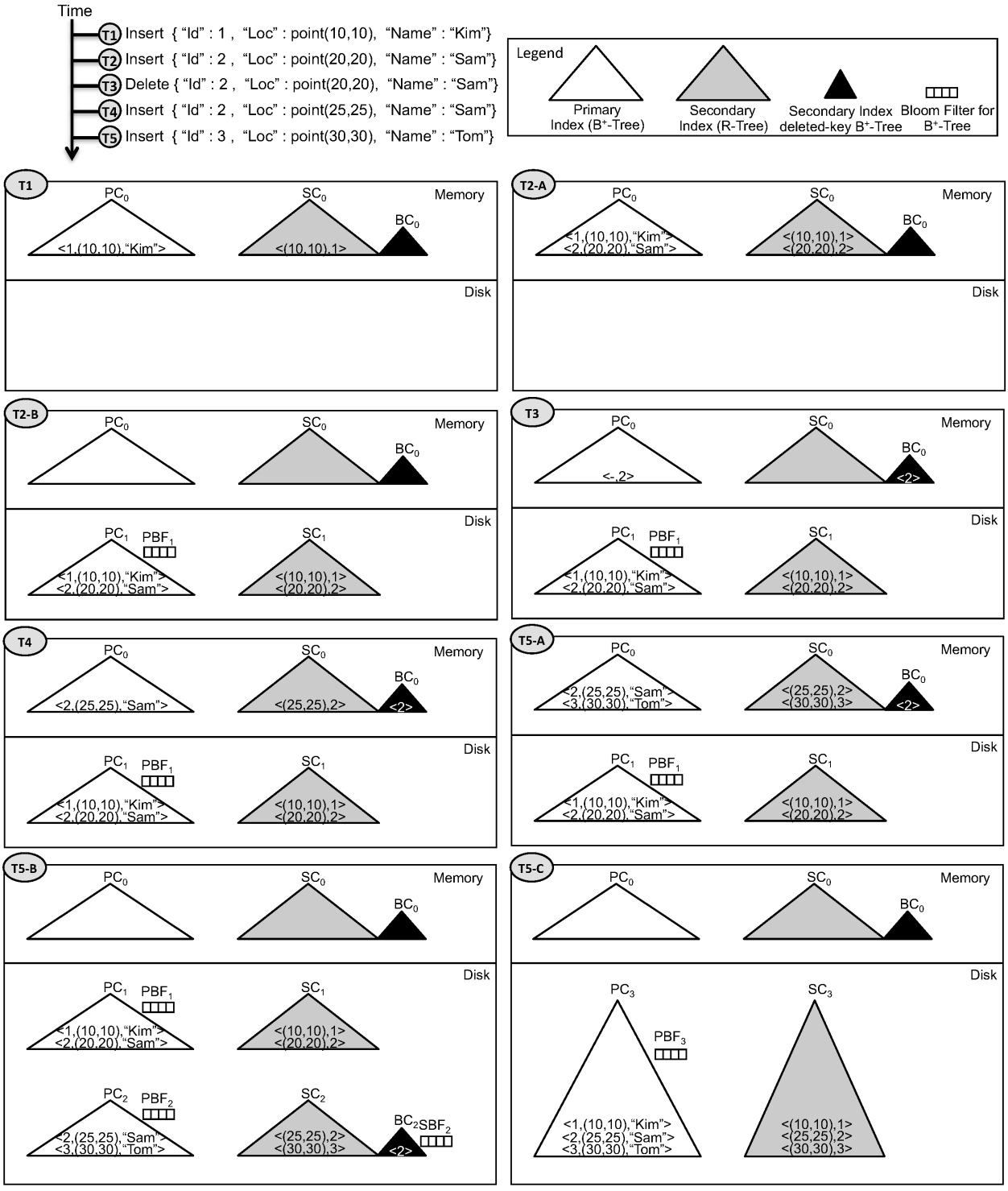


Figure 4.2: An example showing the lifecycle of inserted and deleted records when using our LSM indexing framework.

T5. Each panel in Figure 4.2 represents the final states of the indexes after the operation or side-effect is performed. For expository purposes, we use a flush policy that flushes in-

memory components when the number of in-memory entries is two and a merge policy that merges disk components of an index when the number of disk components is two.

The initial state of the indexes is empty before T1 is executed. Transaction T1 inserts the first record as follows. First, to enforce the primary key uniqueness, T1 must ensure that the key $\langle 1 \rangle$ does not exist in the primary index by performing a search operation. If no match is found, an exclusive lock (X-lock) is acquired on the key.

Next, the entry $\langle 1, (10, 10), \text{“}Kim\text{”} \rangle$ is inserted into the in-memory component of the primary index, PC_0, followed by the insertion of the secondary index entry $\langle (10, 10), 1 \rangle$ into the in-memory component of the secondary index, SC_0 (for which no additional locking is required). The result is shown in the right-hand side of panel T1. The IDs *PIdx* and *SIdx* are the IDs of the primary and the secondary indexes, respectively. At this point T1 is committed and all its locks (the X-lock on key $\langle 1 \rangle$) are released.

The second record $\langle 2, (20, 20), \text{“}Sam\text{”} \rangle$ is inserted by T2 in the same way as in T1. The initial outcome of executing T2 is shown in the panel marked as T2-A. Since the memory budget for the memory components of PC_0 and SC_0 has been exhausted, both components are now flushed to disk, as shown in T2-B (also notice the Bloom filter that has been created for the flushed B^+-tree).

Now, a delete operation is issued by T3 to delete records with coordinates that fall within a radius of 5 degrees from a point with coordinates $\langle 22, 22 \rangle$. The delete operation is transformed to two consecutive operations: 1) find all records satisfying the delete condition and 2) delete the qualified records. In this example, AsterixDB's query optimizer will choose the R-tree as the access method to quickly identify all qualified records. Each retrieved entry must be validated through a primary index lookup. In AsterixDB, validation starts only after all the candidate qualified results are retrieved from the secondary index. This barrier-style validation is used to avoid complex solutions that would involve verifying that the entries in

a secondary index page are still valid when consecutively acquiring and releasing the page's latch during a search operation. This validation solution is analogous to *revalidation after unconditional locking* in [31]. In AsterixDB, the barrier-style validation comes for free since the qualified candidate results from any secondary index are sorted (which is a blocking operation) on the primary key before probing the primary index.

As shown in T3 of Figure 4.2, the entry $\langle(20, 20), 2\rangle$ in SC_1 is going to be retrieved by the delete, followed by a primary index lookup for key $\langle 2\rangle$, where transaction T3 acquires a shared lock (S-lock) on the primary key. Since the qualified record has been identified, now it must be deleted as follows. First, the S-lock on primary key $\langle 2\rangle$ is upgraded to an X-lock. Then, the control entry $\langle -, 2\rangle$ is inserted into PC_0. Notice that the control entry in the primary index includes only the delete flag and the key without the associated payload. After that, the entry $\langle 2\rangle$ is inserted into BC_0.

Next, transaction T4 inserts a record $\langle 2, (25, 25), \text{“}Sam\text{”}\rangle$. Its primary key is equal to the deleted record key in the third operation, but the point now has different location coordinates. When the corresponding entry is inserted into PC_0, the control entry in PC_0 is replaced by the new record as shown in the figure. In contrast, the entry $\langle 25, 25, 2\rangle$ is inserted into SC_0 without deleting the entry $\langle 2\rangle$ from BC_0 (recall that entries in BC_0 only refer to disk components). The locking actions performed on behalf of T4 will be similar to previous transactions.

Finally, transaction T5 inserts a record $\langle 3, (30, 30), \text{“}Tom\text{”}\rangle$ as shown in T5-A, which triggers a flush operation as shown in T5-B. Based on the merge policy, the two disk components are now merged into one as shown in T5-C. In PC_3, the old entry $\langle 2, (20, 20), \text{“}Sam\text{”}\rangle$ in PC_1 has been superseded by the recent entry $\langle 2, (25, 25), \text{“}Sam\text{”}\rangle$ in PC_2 and PBF_3 has been created for the new disk component PC_3. Component SC_3 does not have an entry $\langle 20, 20, 2\rangle$, as it was removed by the control entry in BC_2. Lastly, component BC_2 and its associated Bloom filter were removed since all deleted keys were consolidated during the merge.

4.4 Experimental Evaluation

This section shows results from an experimental evaluation of AsterixDB's storage engine. Section 4.4.1 evaluates the system as a whole while Section 4.4.2 provides a micro-benchmark that evaluates the "LSM-ification" framework from the perspective of an R-tree.

4.4.1 AsterixDB's Storage System

The following experiments demonstrate the scalability of AsterixDB's data-ingestion while varying the number of nodes in the cluster and the number of indexes being used. We show the performance impacts that different merge policies have for data ingestion and queries. In addition, we show the performance of range queries for different operating regions when using the default merge policy of AsterixDB, namely the prefix policy.

To evaluate the performance of data ingestion, we used a feature of AsterixDB called *data feeds*, which is a mechanism for having data continuously arrive into AsterixDB from external sources and to have that data incrementally populate a managed dataset and its associated indexes. We mimicked an external data source, Twitter, by synthetically generating tweets that resemble actual tweets. The synthetically generated tweets have fields such as user, message (the tweet itself), sending time, sender location, and other relevant fields. The tweets are generated in the Asterix Data Model (ADM) format with monotonically increasing 64-bit integer keys. The average size of a tweet was 1KB.

For each experiment, we used two sets of machines. The first set of machines was used to generate the synthetic tweets which are sent over the network to the second set of machines (the AsterixDB cluster) for ingestion. We empirically determined the minimum number of machines that can saturate a single-machine AsterixDB instance, in terms of transactions per second (TPS), and then used this number to scale the number of data-generation machines

when conducting a multi-machine AsterixDB experiment. The second set of machines is a cluster of eight IBM machines used to host an AsterixDB instance, each with a 4-core Xeon 2.27 GHz CPU, 12GB of main memory, and four locally attached 10,000 rpm SATA drives. Of the available 12GB, AsterixDB is given 6GB while the remaining free memory is locked in order to disable the OS's file system buffer cache. In each participating machine, we dedicated one disk to be used by the transaction log manager for writing its log records, while the three remaining disks are dedicated as data storage disks. The three storage disks are used as separate partitions of the tweets dataset by AsterixDB, hosting their associated indexes. Thus, a dataset in an 8-machine AsterixDB instance has 24 partitions.

Currently in AsterixDB, all partitions of a dataset and its indexes in a machine share the same memory budget for in-memory components (divided equally across the indexes). This implies that when the memory component of the index in a partition is declared to be full, all memory components of the indexes for the same dataset in that machine are also declared to be full. Therefore, multiple consecutive flush requests are sent to the I/O scheduler for flushing the memory components of that dataset and its indexes for all the partitions located on the machine.

Table 4.1 shows the configuration parameters used throughout the experiments in 4.4.1, including the threshold values of each merge policy described in Section 4.2.5. Unless otherwise specified, we used the prefix merge policy and the length of each experiment was 20 minutes, starting from an empty dataset. Finally, in all experiments, we used two memory components per index (i.e., double buffering) to avoid stalling ingestion during flushes.

4.4.1.1 Scalability

Figure 4.3 shows the average ingestion TPS as the number of nodes in the cluster is varied with different combinations of secondary indexes. In particular, Figure 4.3(a) shows the TPS

Parameter	Value
Memory given for a dataset and its indexes in a machine	1GB
Data page size	128KB
Disk buffer cache size	3GB
Bloom filter target false positive rate	1%
Memory allocated for buffering log records (log tail)	16MB
Max component size of prefix merge policy	1GB
Max component count of prefix merge policy	5
Max component count of constant merge policy	3

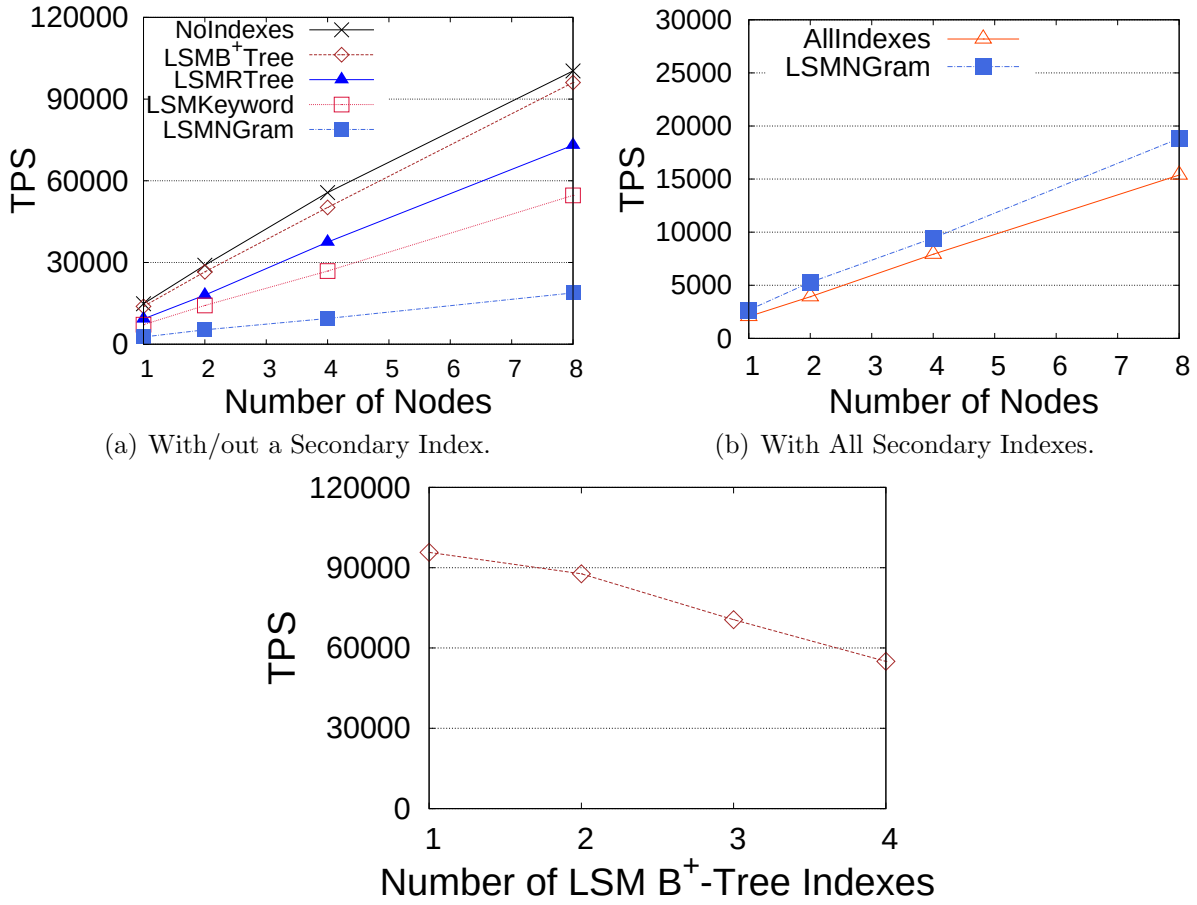
Table 4.1: Settings used throughout these experiments.

when ingesting tweets into a dataset that has no secondary indexes (*NoIndexes*), or has one of a single, secondary LSM B$^+$-tree, LSM R-tree, LSM inverted keyword, or LSM inverted ngram index (*LSMB$^+$Tree*, *LSMRTree*, *LSMKeyword*, and *LSMNGram*, respectively).

As the number of nodes and offered workload increase, we observe near-linear increase in ingestion throughput. A dataset with no secondary indexes always achieves the highest TPS for the obvious reason that there is no overhead of maintaining secondary indexes. Adding a secondary LSM B$^+$-tree to the dataset reduces its ingestion rate, and the reduction is higher as we add more complex secondary indexes such as an LSM R-tree or an LSM inverted index. Clearly, the LSM ngram index is the most expensive index of the four secondary indexes since it requires gram-tokenizing all of the strings in a tweet followed by inserting the resulting ⟨token, id⟩ pairs into the index.

Figure 4.3(b) shows the TPS for a dataset with all of the four supported secondary indexes in AsterixDB. The results show that the upper bound on TPS is determined by the least upper bound of all of the indexes (here the *LSMNGram*).

Figure 4.3(c) shows the effect of adding additional secondary indexes to a dataset in a cluster of 8 machines. When an additional secondary LSM B$^+$-tree was added to the dataset, ingestion throughput dropped near-linearly due to the overhead of maintaining consistency between the primary and the secondary indexes.



(a) With/out a Secondary Index.

(b) With All Secondary Indexes.

(c) Multiple Secondary LSM B$^+$-Trees.

Figure 4.3: Data-ingestion throughput when varying number of nodes and using different types of secondary indexes.

4.4.1.2 Effects of Merge Policies

This section shows how three different merge policies–prefix, constant, and no-merge–affect ingestion throughput. In Table 4.2, we report the TPS and its ratio to the prefix policy for each merge policy. In addition, we show the average number of flush and merge operations that were performed per partition.

Regardless of the merge policy, the average flush count per partition is always proportional to the TPS. This is due to the fact that given the same amount of data to ingest and the same in-memory component size, a higher TPS indicates the memory component is always

filled faster, incurring more flush operations.

Using the prefix merge policy yields the highest TPS for two primary reasons. First, it avoids merging large components by ignoring those that are larger than the size threshold $M = 1\text{GB}$. In addition, it always tries to find the *smallest* sequence of components, whose total size is larger than M, to be considered for a merge. Second, the policy avoids stacking too many small components through the use of the predefined count threshold, $C = 5$. When the policy fails to find a sequence of components based on size consideration, it still considers merging small components when their count exceeds C. Reducing the number of small disk components improves the performance of both inserts (by reducing the cost of enforcing the primary key uniqueness constraint) and search queries. Recall that to ensure primary key uniqueness, each insert into the primary index must be preceded by searching all components of the index for a duplicate. Thus, as more disk components are accumulated, the cost of maintaining uniqueness increases since more and more components must be searched.

When the constant policy is used, we observe that the TPS is lower than the TPS achieved by the prefix policy. This is because the constant policy never allows the number of disk components to exceed three, resulting in fewer large disk components after merging. Also without considering a component's size, the constant policy will merge small components with large components, resulting in all previously ingested data being read and written to/from disk, so as more data is ingested, merges become more costly.

Merge Policy	**TPS**	**TPS Ratio to Prefix**	**Flush Count**	**Merge Count**
Prefix	102,232	1.00	47	12
Constant	77,774	0.76	35	17
No-Merge	78,101	0.76	35	0

Table 4.2: Merge policies effect on ingestion throughput.

4.4.1.3 Range Query Performance

In this experiment, we ingested a total of 490GB worth of synthetic tweets into a dataset with a secondary LSM B^+-tree index on a randomly generated, 32-bit integer field, where both the primary and secondary indexes are using the prefix merge policy. We queried the dataset in three different operating regions: 1) during ingestion, 2) post-ingestion, and 3) post-ingestion and post-compaction, where each index's disk components are manually compacted (merged) to form a single component per index. The experiment started by ingesting 420GB worth of tweets. Then, during ingestion, random range queries are submitted sequentially to AsterixDB. Once data ingestion ended (490GB worth of tweets were ingested), 2000 random range queries were submitted sequentially. The dataset was then compacted so that every index had exactly one disk component. After that, another 2000 random range queries were submitted sequentially. All queries had range predicates on the secondary key. Thus, the secondary index is always searched first, followed by probing the primary index for every entry returned from the secondary index. In addition, the queries were generated randomly such that the result sets have an equal probability to have a cardinality of 10, 100, 1000, or 10000 records.

Table 4.3 shows the results of this experiment. Out of all three operating regions, querying the dataset while it is ingesting data produces the slowest response time for the obvious reason that the system resources (CPU and disk) are being contended for by both inserts and queries. We also observed that the performance of the queries was worse when there was an ongoing merge, as merges were both CPU and I/O intensive operations.

Once ingestion was over, the query performance improved by a large margin due to reduced resource contention. On the other hand, surprisingly, the performance of queries after the final compaction improved by only a small margin. The reason for the comparable performance is two fold. First, before probing a primary index, the entries are sorted based on

Avg. Result Cardinality	Avg. Response Time While Ingesting	Avg. Response Time After Ingestion	Avg. Response Time After Compaction
10	1023	139	138
100	1185	191	184
1000	2846	634	500
10000	11647	3747	3381

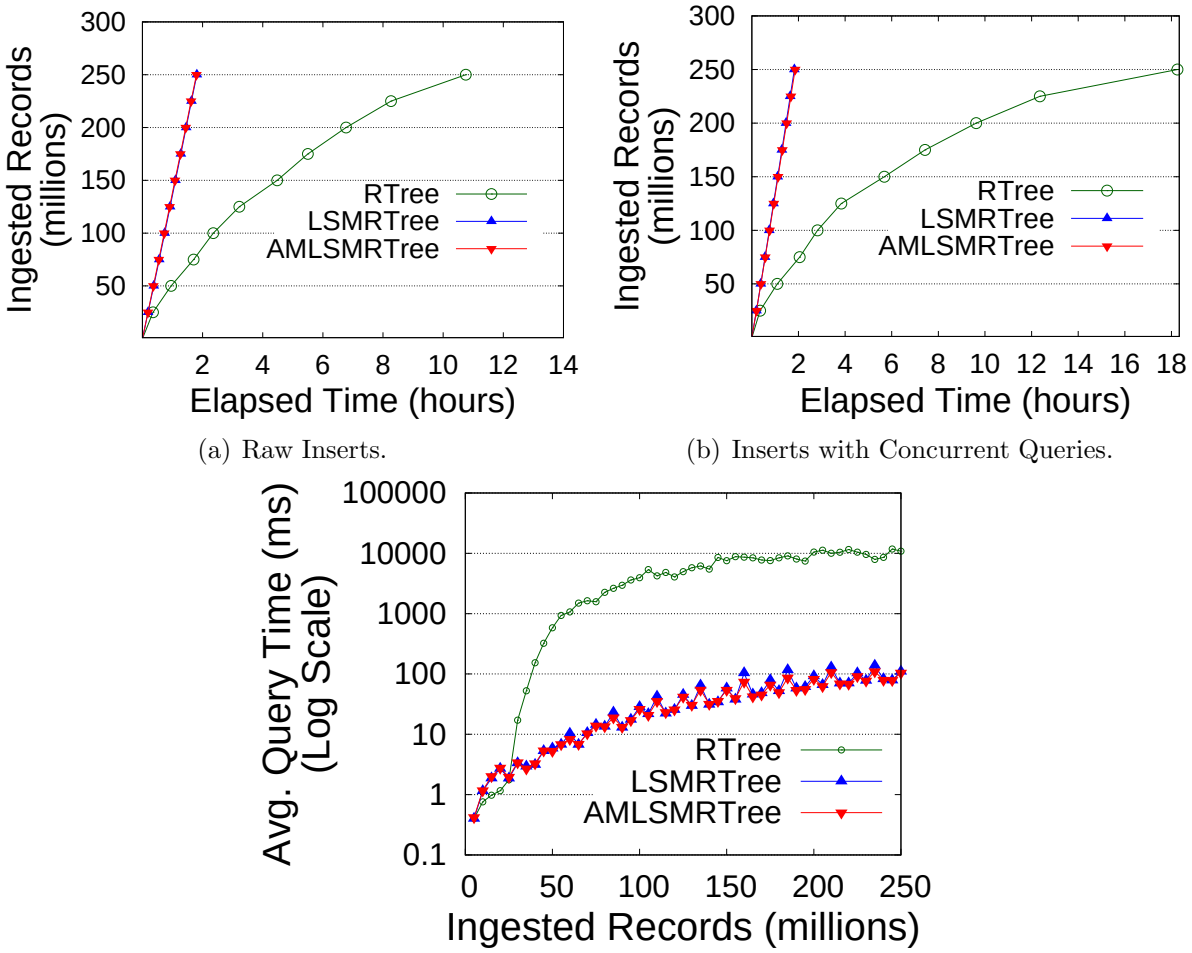
Table 4.3: Range query performance (in milliseconds).

the primary key; therefore, good cache locality is achieved when accessing the primary index, mitigating the negative effects of having more disk components. Second, each primary index probe is a point lookup that makes use of the Bloom filters on the primary-index disk components, greatly reducing the chance of unnecessary I/O.

Finally, for all operating regions, there is a considerable overhead for small range queries, which is caused by the Hyracks [16] job initialization. For each request, a new job is created and executed. As job creation includes the translation and optimization of an AQL program and execution includes the distribution and start of the job on all cluster nodes, this overhead can be significant for small jobs/queries. We tried an AQL query that does not touch persistent data to estimate this overhead. AsterixDB took an average of 43ms to execute our "no-op" query. We plan on fixing this next.

4.4.2 LSM-ification Framework

Next we show the detailed performance characteristics of a secondary index implemented using the "LSM-ification" framework. To do so, We compare an LSM R-tree (*LSMRTree*), an AMLSM R-tree (*AMLSMRTree*), and a conventional R-tree [23] (*RTree*). For both LSM indexes, the Hilbert curve was used to order the entries within each disk component. The following experiments were performed as micro-benchmarks where the indexes were accessed directly, bypassing compilation, job setup, transactions, and other code paths that are in-



(a) Raw Inserts.

(b) Inserts with Concurrent Queries.

(c) Queries during Ingestion.

Figure 4.4: Ingestion and query performance of the R-tree, LSM R-tree, and AMLSM R-tree.

curred by AQL statements submitted to AsterixDB.

We first empirically determined that the best ingestion and query performance for the LSM indexes can be obtained when their in-memory and disk components page sizes are 0.5KB and 32KB, respectively. On the other hand, a 2KB page size yielded, by far, the best ingestion performance for the R-tree. However, larger page sizes such as 16KB yielded much better query performance, but performed poorly for ingestion. We decided to use a 2KB page size for the R-tree since we are focusing on ingestion-intensive workloads. All experiments in this section used a single machine and utilize a single disk. All records and queries are sent from a client residing on a different machine. The machine configurations are the same as

the experiments in 4.4.1. The R-tree used a 1.5GB LRU buffer cache, while each of the two LSM indexes used a 1GB LRU buffer cache for caching disk component pages and 0.5GB for their in-memory components. In addition, we used the prefix merge policy for both LSM indexes.

These micro-experiments employed data inspired by a real dataset. The source dataset contained event occurrence data that had time and location stamps, but the location information available in the source data was just at the (*city, state*) level. To convert this data into a more GPS-like spatial dataset for use in our micro-experiments, we synthetically augmented it as follows: we geotagged all of the US records (of which there were approximately 120 million) and ended up with roughly 4500 unique data points. We then used that point data to generate records following the frequency distribution of the provided dataset, but we shifted the points by adding random values in the range of [-0.25,0.25] to the latitude and longitude values as a way to mimic a more realistic fined-grained spatial dataset. The final dataset thus has approximately 4500 clusters, each with many distinct data points. The size of each record is 40 bytes (four double values representing each record's bottom-left and upper-right corners and a 64-bit integer representing a monotonically-increasing primary key). Note that since the spatial location being indexed is a point, the minimum bounding rectangle (MBR) for the point has corners with identical data (the point itself). It would be possible to optimize space in this case, but that is not currently done.

We generated query MBRs that provided result sets with similar cardinalities when querying each index as follows. Since many of the clusters are overlapping, the frequency distribution of each cluster was pre-computed based on the percentage of the overlapping area. For example, if two clusters overlap each other by 20% of their area, the frequency distribution of each cluster is incremented by 20% of the (original) distribution of the other cluster in order to compensate. To create an MBR for a query, a cluster is first chosen at random. Then, the size of the MBR is determined based on the pre-computed frequency distribution of the

chosen cluster such that the size will be relatively small if the cluster is highly populated and vice-versa. The size was restricted so as not to exceed the cluster size, avoiding the creation of queries that may span clusters, which may lead to very large result sets (this occurs when the number of records in the cluster is less than the requested result cardinality). Based on the MBR's size, we choose the MBR center randomly within the bounds of the cluster such that the MBR is fully contained inside the cluster. The average result set size of the queries was 2361 with 250 million records ingested into an index.

4.4.2.1 Ingestion Performance

Raw inserts: Figure 4.4(a) shows the raw insert performance when ingesting a total of 250 million records from a single stream. The R-tree took roughly 11 hours to ingest all of the data. Both LSM indexes were able to ingest the same amount of data in less than two hours. In the figure, the LSM indexes' curves are on top of each other because they behave similarly when the workload did not contain deletes (i.e., append-only workloads), as expected.

We also observe from Figure 4.4(a) that the LSM indexes maintained the same ingestion rate throughout the ingestion period. The first reason for this is that random disk I/O was avoided by LSM indexes. Second, in contrast to a primary index, a secondary index did not enforce key uniqueness, avoiding the overhead of checking for duplicate keys. Third, the prefix merge policy bounded the merge cost by ignoring disk components that were larger than a specified threshold (1GB in our experiments). On the other hand, the R-tree suffered from performance degradation over time. As more data is ingested, the cost of performing in-place index writes becomes increasingly expensive since the height of the tree grows without bound.

Inserts with concurrent queries: Similar to the previous experiment, a stream of 250 million sequential insert operations was sent to the indexes. In addition, a second, concur-

rent stream of queries were submitted sequentially during ingestion. Figure 4.4(b) shows the performance of these queries. For both LSM indexes, the effect was very minimal, mainly because their inserts are mostly CPU bound while queries are I/O bound. Therefore, queries were not contending with inserts for resources. On the other hand, the R-tree's ingestion ability was negatively affected, with elapsed time up from 11 hours to 18 hours. The main reason was that in-place inserts and queries were both I/O bound, causing resource contention between the two operations.

4.4.2.2 Spatial Range Query Performance

Query performance after ingestion: Here, again, 250 million records were ingested into each of the three indexes. Then, 1000 queries were submitted sequentially to each index. After that, the two LSM indexes were each compacted into a single disk component. Then, the same 1000 queries were again submitted to the LSM indexes. Before compaction, each LSM index had 12 disk components, resulting from 78 flush and 19 merge operations, for a total size of 11.3GB. After compacting the disk components, the size of the single component was 11.3GB in both LSM indexes. The final R-tree index size was 17.6GB. The difference in size was due to the fact that the LSM indexes fully packed the pages of on-disk components since they were immutable, resulting in better space utilization. Table 4.4 shows the results of this experiment.

After ingestion, the average query response time for the basic R-tree was 25 and 29 times slower than that of the LSM R-tree and the AMLSM R-tree, respectively. The LSM indexes' query times were much better since they incurred fewer buffer cache misses compared to the R-tree, which can be attributed to: 1) the LSM R-tree and the AMLSM R-tree used the Hilbert curve to order their entries in every disk component, which improves the clustering quality of the R-trees' entries; 2) the pages of the LSM disk components were fully utilized; and 3) the disk page size of the LSM indexes was larger than the R-tree page size. The

LSM R-tree performed slightly better than the AMLSM R-tree because the AMLSM R-tree returned records sorted based on the Hilbert order to reconcile index entries. Since there were no deleted records in this experiment, the time spent to maintain the Hilbert order while answering queries was wasted. On the other hand, the Bloom filters associated with every deleted-key B^+-tree in the LSM R-tree handled this special case (an empty Bloom filter) in an efficient manner. After compacting the LSM indexes, their performance dramatically improved since spatially-adjacent entries from different disk components were packed into a single component. Thus, the LSM indexes had to access fewer disk pages when answering a query.

Index Name	**After Ingestion**		**After Compaction**	
	Avg. Response Time	Avg. Cache Misses	Avg. Response Time	Avg. Cache Misses
RTree	2543.2	560.6	-	-
LSMRTree	86.7	24.7	17.5	8.1
AMLSMRTree	100.8	24.7	16.5	8.1

Table 4.4: Spatial range query performance (in milliseconds) and number of buffer cache misses after ingestion and compaction.

Query performance during ingestion: In this experiment, a stream of 250 million inserts were again sent concurrently with another stream that submitted queries sequentially during the ingestion. Figure 4.4(c) shows the corresponding query performance. Notice that the Y axis is using a log scale. Each data point represents the average query time of all the queries that were submitted since the last data point. When the index had less than 25 million records, the R-tree query performance was slightly better because all its pages were cached in memory, and because queries submitted to the LSM indexes contended with flush and merge operations (there were 9 flushes and 2 merges during the ingestion of the first 25 million records). When the R-tree had ingested around 30 million records, pages from the buffer cache started to be evicted, which led to significant performance degradation. Both LSM

indexes provided comparable query performance throughout the ingestion process. Their query times show some variance (the sawtooth shape) due to resource contention of ongoing merge operations.

Effect of deletion: Finally, we also studied the impact that deletion has on query performance for the LSM R-tree and AMLSM R-tree. Again, we used a single stream that inserted 250 million records with a modification that now causes a 1% chance to delete an existing record instead of inserting a new record. Therefore, by the end of the experiments, the stream had submitted roughly 2.5 million delete operations to each index. We also used a second, concurrent stream that sent queries sequentially during the ingestion process.

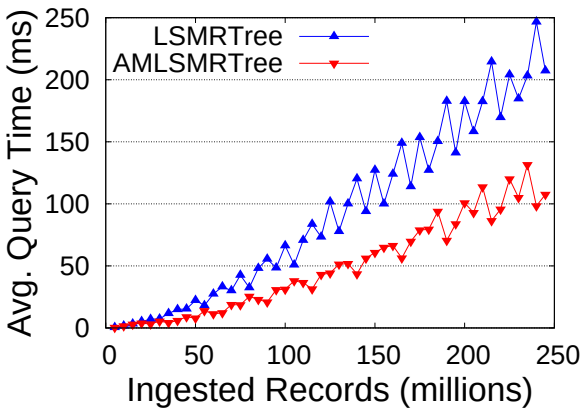


Figure 4.5: Effect of deletion on query performance.

Figure 4.5 shows the performance of queries in this case. Again, each data point represents the average query time of all the queries submitted since the last data point. The AMLSM R-tree consistently provided faster query response time than the LSM R-tree, almost by a factor of two, due to the handling of entry reconciliation. The AMLSM R-tree reconciliated entries "on the fly" as it accessed the entries based on Hilbert order from different disk components. On the other hand, the LSM R-tree reconciliated entries by probing all of the Bloom filters that were associated with the deleted-key B^+-trees of the newer components, and possibly the deleted-key B^+-trees themselves, which was more costly than reconciliation of entries in the AMLSM R-tree.

4.5 Conclusions

In this chapter, we have presented the storage engine implemented in the AsterixDB system. We described the framework in AsterixDB that leverages existing implementations of conventional indexes (e.g., the R-tree) to convert them to LSM-based indexes, which allowed us to avoid building specialized index structures from scratch and enables the advantages that an LSM index provides. Further, we explained how AsterixDB enforces the ACID properties across multiple heterogeneous LSM indexes. We also discussed the challenges that a system like AsterixDB faces for managing its disk and memory resources when dealing with many LSM-based indexes. Finally, we shared results from a preliminary evaluation of AsterixDB's storage engine that shows its performance characteristics in different settings and we presented the results for a set of micro-benchmarks to evaluate the "LSM-ification" framework.

Chapter 5

Filter-Based LSM Index Acceleration

5.1 Introduction

Not a long time ago, using traditional data warehouse systems for data analytics was the norm, and the technology was only accessible to big companies capable of writing big checks. The exponential growth of social-networking data, however, combined with open source software platforms, has motivated smaller companies and organizations to collect and store huge amounts of data on a daily basis as well. Thanks to the recent technological advances in the Big Data space, those companies are now able to use Hadoop-based solutions to analyze the data and gain valuable insights that can help them sustain their businesses. With the evolving world of social media, however, it has quickly become evident that not all data analytics problems can be solved efficiently with Hadoop, particularly due to its nature as a batch processing system. Location-based advertising, credit card fraud analytics, and recommendation systems are examples of applications that require real-time responses for which the speed of answering such queries in Hadoop is not sufficient. Therefore, new data platforms such as NoSQL systems and stream-processing systems have emerged to handle

such use cases (e.g., MongoDB, HBase, Cassandra, BigTable, Spark, Storm, etc.). In many cases, companies have ended up using Hadoop-based solutions for long-running tasks, mostly for analyzing historical data, while using index-based (e.g., LSM-tree based NoSQL stores) and streaming-based solutions for short-running tasks that are more time-critical for their businesses.

One class of queries that require real-time answers and have wide range of use cases are query workloads that favor recent data. As an example, let us consider a spatial aggregation query similar to the one mentioned in Chapter 1. Suppose the campaign manager for a US presidential candidate wants to know how potential voters are currently reacting to the candidates in a certain geographic area. A useful piece of information is the level of voters' interest in other rivals, as this is clearly valuable to the decision-making process of the campaign. We can formulate a spatial aggregation query to find tweets mentioning the names of other rivals that have been posted within the *last* day, group them on a spatial grid structure, and compute the number of such tweets for each cell in the grid. By doing this time-based spatial analysis, the campaign staff can gain an understanding of the current public opinion and make informed decisions such as broadcasting more political ads in certain areas.

As another example for query workloads that favor recent data, consider power-consumption monitoring systems. This is a real use case shared with us by a local company that provides energy management solutions. In their setting, there are thousands of sensors installed in different buildings (e.g., on a university campus) to monitor energy consumption. Sensor readings are continuously collected and never thrown away. The users (campus facility administrators) are then provided with a GUI to monitor energy consumption by showing them different charts representing aggregated sensor readings in different buildings. The users must specify the desired time resolution for their queries (e.g., last hour, last four hours, last eight hours, last 12 hours, last 24 hours, last week, last month, last quarter, last

year, and all data). Based on what we heard from their engineers, users mostly perform analytics on recent data and rarely on old data; in particular, their users tend to be most interested in the last 12 and 24 hour time resolutions. Their engineers determined that their initial solution, based on a relational database system, would probably not scale well, so they have resorted to pre-computing the aggregation values as they ingest the sensor readings into a popular key-value store. Although they can obtain fast responses with their new approach, their solution still suffers from being write-intensive since they pre-compute aggregations for different time resolutions and write them into different tables. In addition, by using a key-value store, they have lost the advantages of using a declarative query language and they would consider moving back to a system that provides such a language (e.g., AsterixDB), as long as their time-based aggregation queries can run fast and the system is production-ready and reliable.

In this chapter, inspired by such use cases, we study how to answer queries on recent data efficiently in the context of AsterixDB. The AsterixDB project has taken a different approach than other systems combining a wide range of capabilities in a single unified system in order to provide better manageability, functionality, and performance, as opposed to gluing together multiple independent solutions. As described in the previous chapter, one novel feature of AsterixDB is wholly adopting LSM-trees as the underlying technology for all of its internal data storage and indexing. In this chapter, we will further exploit this feature. We show how to utilize LSM structures to provide near real-time responses to query workloads that favor recent data.

The rest of this chapter is organized as follows. First, we review the Asterix Data Model (ADM) and the Asterix Query Language (AQL). Next, we describe a few approaches to solve the problem and discuss the limitations of each approach. Then, we explain a new solution, which exploits the multi-component nature of LSM-based indexes to provide near real-time performance for queries on recent data. Finally, we present results from an experimental

study that we conducted to evaluate the solution.

5.2 User Model: AsterixDB Query Language

In this section we describe the user model of AsterixDB [5], consisting of its data model and query language targeting semistructured data. We will illustrate the features of ADM and AQL via a simple scenario where we want to store, index, and query social data from Twitter.

5.2.1 Asterix Data Model

The Asterix Data Model (ADM) is based on ideas borrowed from JSON extended with additional primitive types as well as type constructors borrowed from object databases [35]. Figure 5.1 illustrates the user of ADM to model a Twitter messages dataset, which we will use as our running dataset example throughput this chapter.

To begin, we create a new dataverse called `SocialData`, and use it. A dataverse in AsterixDB is analogous to a database in the relational world. After that, we create the `TwitterUserType` and `TweetType` types to model Twitter users and messages, respectively. Notice that the `TwitterUserType` is an open type, signifying that the instances of this type will conform to its specification but are allowed to contain arbitrary additional fields that may vary from one instance to the next. On the other hand, the `TweetType` is a closed type, meaning that each of its instances must conform to its specification and cannot contain extra fields. The "?" symbol in the `sender-location` field means that the presence of this field is optional. The `referred-topics` field is a collection of primitive string values, and the user information is represented as a nested record of another type. Finally, we create a dataset called `Tweets` that stores data instances conforming to the `TweetType`

```
create dataverse SocialData;
use dataverse SocialData;

create type TwitterUserType as open {
    screen-name: string,
    lang: string,
    friends_count: int32,
    statuses_count: int32,
    name: string,
    followers_count: int32
};

create type TweetType as closed {
    tweetid: string,
    user: TwitterUserType,
    sender-location: point?,
    send-time: datetime,
    referred-topics: {{ string }},
    message-text: string
};

create dataset Tweets(TweetType) primary key tweetid ;
```

Figure 5.1: Data Definition Language (DDL) to create a dataset of tweets in AsterixDB.

data type (as an AsterixDB dataset is loosely analogous to a table in the relational world). The dataset's primary key is the `tweetid` field.

Figure 5.2 shows two DML statements to insert two data instances into the `Tweets` dataset. Each data instance is created by using the record constructor {}. Similarly, the data values of the complex fields `user`, `sender-location`, `send-time`, and `referred-topics` are created by using their corresponding function constructors; namely {}, `point`, `datetime`, and {{}}, respectively. As can be seen in the first insert statement, the `sender-location` is missing from the data instance, which is acceptable since it was declared as an optional field. Also notice that in the second insert statement, we have added an additional field, `age`, inside the nested data instance `user`, which is also acceptable since the `TwitterUserType` was created as an open type.

```
insert into dataset Tweets {
"tweetid": "1",
"user":{"screen-name": "JohnSmith",
    "lang": "en",
    "friends_count": 39331,
    "statuses_count": 654,
    "name": "John Smith",
    "followers_count": 65409},
"send-time": datetime("2014-04-26T10:10:00Z"),
"referred-topics": {{"phone","customization"}},
"message-text": "I like my phone, its customization is cool."
}

insert into dataset Tweets {
"tweetid": "1000",
"user":{"screen-name": "WilliamSilverman",
    "lang": "en",
    "friends_count": 2765,
    "statuses_count": 126,
    "name": "William Silverman",
    "followers_count": 872,
    "age":51},
"sender-location":point("41.44,61.65"),
"send-time": datetime("2014-05-21T11:11:01Z"),
"referred-topics": {{"basketball"}},
"message-text": "Going to play basketball this afternoon."
}
```

Figure 5.2: Inserting two data instances into the `Tweets` dataset.

5.2.2 Asterix Query Language

The Asterix Query Language (AQL) is a declarative query language that borrows its core ideas from XQuery [47]. In the following, we provide some basic query examples that show some of the capabilities of AQL.

5.2.2.1 Simple Selection

The query shown in Figure 5.3 conceptually iterates over the dataset `Tweets` and returns all the users with the name "John Smith". As can be seen, AQL uses "." syntax to access

fields.

```
for $tweet in dataset Tweets
where $tweet.user.name = 'John Smith'
return $tweet
```

Figure 5.3: An example of a simple selection query in AsterixDB.

5.2.2.2 Group-by, Order-by, and Limit

The query shown in Figure 5.4 iterates over the tweets and their referred (i.e., referred-to) topics, grouping them based on the referred topics, and counts the number of tweets having each referred topic. It then sorts the results based on the count of referred topics, and finally returns the three most frequently referred topics in the `Tweets` dataset.

```
for $tweet in dataset Tweets
for $referredTopic in $tweet.referred-topics
group by $topic := $referredTopic with $tweet
order by count($tweet)
limit 3
return {"Topic": $topic, "Occurrence": count($tweet)}
```

Figure 5.4: An example of a query with group by, order by, and limit in AsterixDB.

5.2.2.3 Join

Suppose we have another dataset called `FacebookMessages` that stores Facebook users' posts. The query shown in Figure 5.5 joins the `Tweets` and `FacebookMessages` datasets and returns all users who have accounts in both Twitter and Facebook (for expository purposes, we assume the user names are the same across social networks and also unique).

```
for $tweet in dataset Tweets
for $message in dataset FacebookMessages
where $tweet.user.name = $message.user.name
return {"User name": $tweet.user.name}
```

Figure 5.5: An example of a join query in AsterixDB.

5.2.3 Spatial Support in AsterixDB

As we mentioned in Section 2.2, ADM provides native support for spatial data types. In particular, it supports the following two-dimensional types: point, line, rectangle, closed polygon, and circle. In Figure 5.1, the `sender-location` in `TweetType` is of the `point` type. In addition, AsterixDB has several built-in spatial functions, including:

- *spatial-intersection*: Given two geometry objects of a spatial data type (point, line, etc.), this function outputs a boolean value to indicate whether or not the two objects intersect.
- *spatial-distance*: Given two points, the function returns the Euclidean distance between them.
- *spatial-area*: Given a geometry object, this function returns its area.
- *spatial-cell*: This function determines which grid cell a point-of-interest belongs to. It receives the location of the point, the origin of a bounding rectangle, and the latitude and longitude increments to specify the resolution of the grid.

Figure 5.6 shows an example of how a user of AsterixDB can create an LSM-based R-tree index on the spatial attribute `sender-location`.

Currently, all spatial data types are indexable, and the optimizer is able to use the spatial index to answer selection queries that use the spatial-intersect function on an indexed field.

```
create index locationIndex on Tweets(sender-location) type rtree;
```

Figure 5.6: An example showing how to create an LSM-based R-tree index in AsterixDB.

Figure 5.7 shows a spatial selection query in AsterixDB. The query can utilize the R-tree index over the `sender-location` field to returns all the tweets that are within the specified polygon.

```
for $tweet in dataset Tweets
let $polygon := create-polygon([40.0,79.87,41.0,80.0,45.0,75.5,39.0,72.0])
where spatial-intersect($tweet.sender-location, $polygon)
return $tweet
```

Figure 5.7: An example of a spatial selection query utilizing a secondary LSM R-tree index in AsterixDB.

In addition, AsterixDB supports index nested-loops spatial join queries. Figure 5.8 shows a relevant example query; this query goes through the set of all tweets and, for each one, uses a nested query to pair it with a bag of tweets sent from nearby locations.

```
for $tweet1 in dataset Tweets
let $circle := create-circle($tweet1.sender-location, 0.25)
return {
"tweetid1": $tweet1.tweetid,
"nearby-tweets": for $tweet2 in dataset Tweets
                 where spatial-intersect($tweet2.sender-location, $circle)
                 return {"tweetid2":$tweet2.tweetid}
}
```

Figure 5.8: An example of a spatial join query that can be optimized to use an LSM R-tree index (if exists) in AsterixDB.

5.3 Answering Queries on Recent Data

In this section, we present alternative approaches to optimize query workloads that favor recent data (called "recency queries" hereafter) in a system like AsterixDB, and we discuss

the tradeoffs and limitations of each approach. We will use the spatial aggregation use-case mentioned in Section 5.1 to drive our discussion.

5.3.1 Spatial Aggregation Use Case

We have added capabilities for doing spatial aggregation in AsterixDB due to their importance. Such a query typically specifies a grid structure including a spatial range and a resolution and asks for a density distribution (histogram) of data within the grid. The query may optionally include further conditions such as a time interval and keywords and do an aggregation on the data records satisfying these additional conditions. A spatial aggregation query logically partitions data records into groups (based on grid cell membership) and applies an aggregation function to all records in each group. Figure 5.9 shows a color-coded density grid on the map that visualizes the results of a typical spatial aggregation query using the Google Maps API.

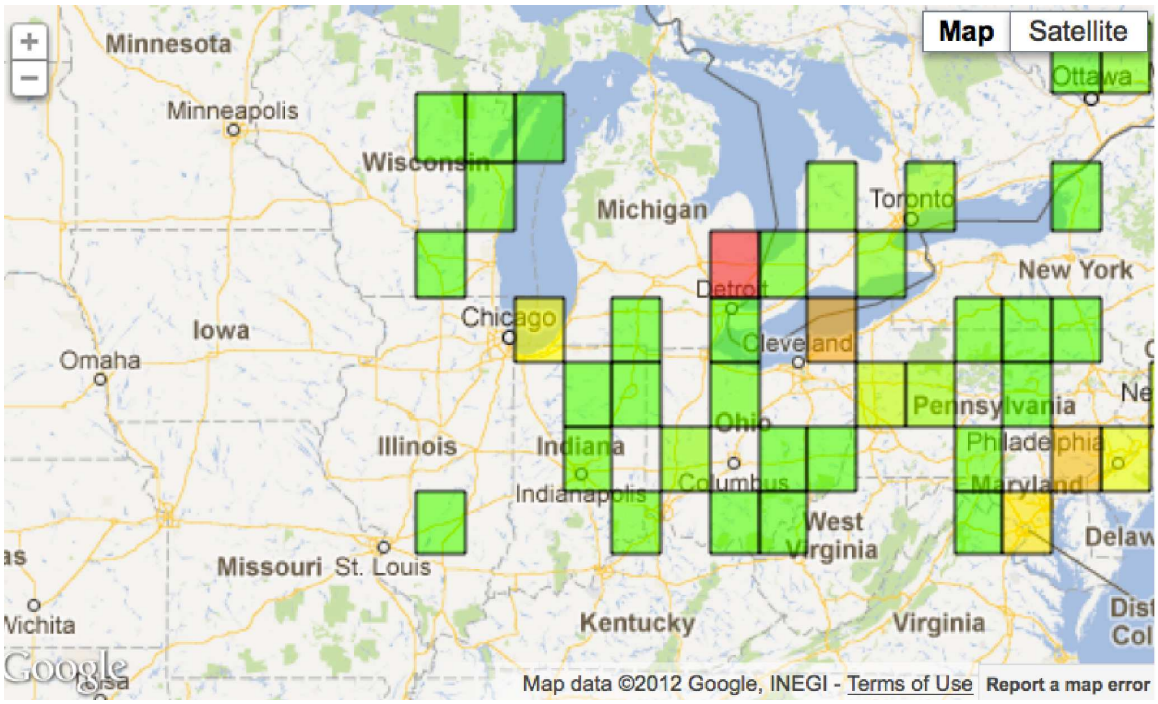


Figure 5.9: A visualization of the results of a spatial aggregation query. The color of each cell indicates the tweet count.

Since our focus in this chapter is optimizing recency queries, we will discuss possible approaches to optimize this query when it includes a temporal predicate that only returns recent data records and filters out the old ones.

Suppose we are just a few days away from the US presidential election that is going to be held on November 8th, 2016. A campaign manager could use a GUI to submit a spatial aggregation query to AsterixDB, such as the query shown in Figure 5.10, to see how potential voters are reacting to the candidate "Hillary Clinton", *recently*, in the swing state of Ohio. This query spatially aggregates election-related tweets. It starts by constraining tweets to a bounding rectangle inside Ohio, a datetime window of the last day, and containing the referred topic "Hillary Clinton". The `spatial-cell` function determines which grid cell each tweet belongs to. This function receives the location of the tweet, the origin of the bounding rectangle, and the latitude and longitude increments that specify the resolution of the grid. It returns the cell (represented by a rectangle) that the tweet belongs to. Those tweets are then grouped according to their containing grid cells. Finally the `count` function is applied to each group of tweets to return the final answer in the form of pairs of cells and the number of tweets (that satisfy the predicates) in the corresponding cell.

5.3.2 Alternative Runtime Execution Strategies

In this section, we present possible runtime execution strategies for recency queries. We discuss the alternative query plans and their tradeoffs and limitations.

A query that is submitted to AsterixDB undergoes different stages before it is executed [5]. First, the query is parsed and compiled to generate a logical plan. Then, the rule-based query optimizer (Algebricks) will take over and attempt to optimize the logical plan by using many rewriting rules. The final optimized logical plan is then converted to a Hyracks job and handed over to the Hyracks runtime system for execution.

```
for $tweet in dataset Tweets
let $searchReferredTopic := "Hillary Clinton"
let $leftBottom := create-point(38.52,-84.78)
let $rightTop := create-point(41.94,-80.48)
let $latResolution := 3.0
let $longResolution := 3.0
let $region := create-rectangle($leftBottom,$rightTop)
where spatial-intersect($tweet.sender-location, $region)
and $tweet.send-time > datetime("2016-11-05T00:00:00Z")
and (some $referredTopic in $tweet.referred-topics
  satisfies ($referredTopic = $searchReferredTopic))
group by $cell := spatial-cell($tweet.sender-location,
  $leftBottom, $latResolution, $longResolution)
  with $tweet
return { "Cell": $cell, "NumTweets": count($tweet) }
```

Figure 5.10: A spatial aggregation query over tweets that were generated by Ohio state users, close to the US presidential election in 2016, containing the referred topic "Hillary Clinton".

All the optimized logical query plans that we will discuss for the spatial aggregation query are similar to each other in the sense that their execution involves performing two consecutive steps:

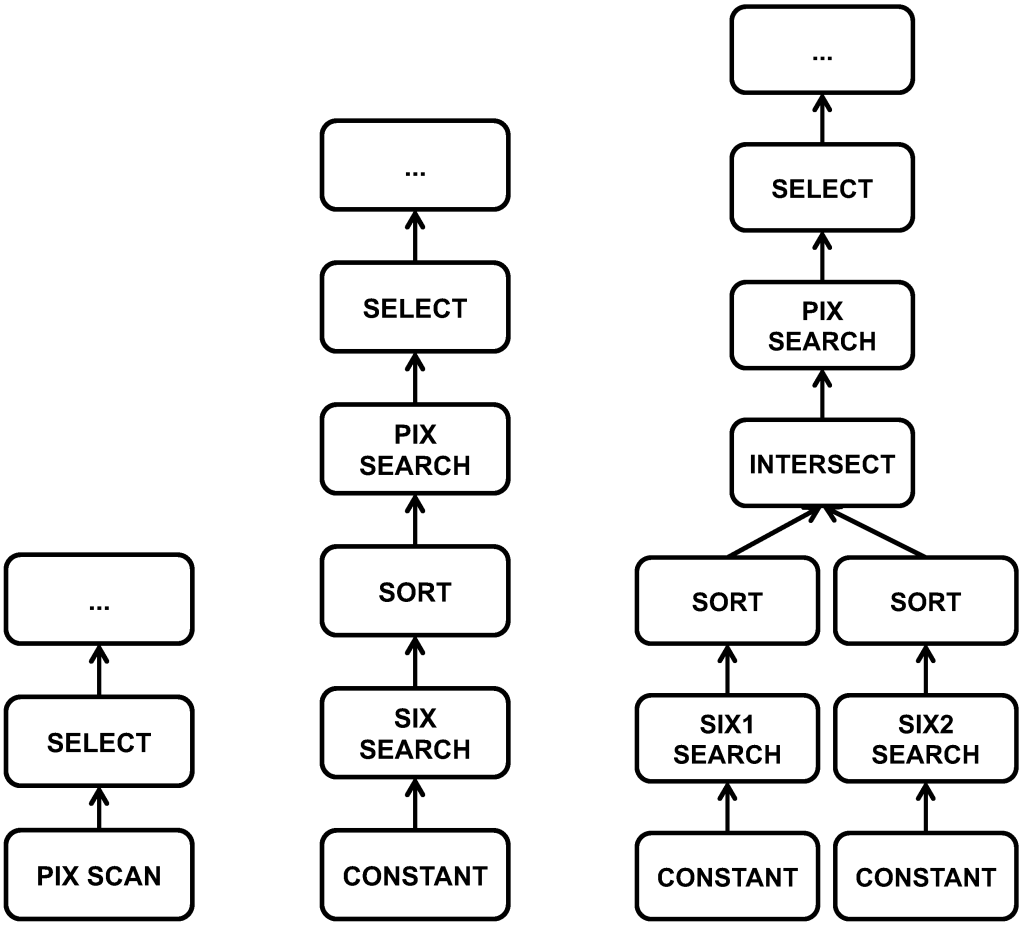
1) Fetching all data records that match the query predicates, and

2) Partitioning data records into groups and applying an aggregation function to all records in each group.

Our goal here is optimizing the first step, i.e., reducing the required time to fetch all records that satisfy the query predicates, as it turns out that this step tend to dominate the query execution time (and also not all recency queries involve performing aggregation).

In the case that there are no applicable secondary indexes, the only possible AsterixDB plan for executing this query would be to perform a full scan over the `Tweets` dataset. Figure 5.11(a) shows a simplified snippet of the optimized logical plan for this execution strategy. This strategy would be a good choice if the query predicates are not very selective.

However, since the query has many potentially selective predicates on different fields, it would make sense to have a secondary index on one or more of the fields (e.g., an R-tree on the `sender-location` field or a B^+-tree on the `send-time` field). Figure 5.11(b) shows the query plan for an index-based strategy in AsterixDB. The plan starts by feeding the secondary index with the applicable query predicate (shown in the plan as "CONSTANT") to fetch all the qualified primary keys. The resulting primary keys are then used to lookup the base data from the primary index, sorting them based on their keys first in order to access the primary index in an efficient manner.



(a) Scan-Based Plan. (b) Single Index Plan. (c) Multi Index Plan.

Figure 5.11: Different selection-query plans. Each box represents a logical operator in the query plan. "PIX" stands for Primary Index, and "SIX" for Secondary Index. The select operator contains all the predicates in the query.

There are two drawbacks, however, when using such an index-based strategy for answering a multi-predicate query.

1) It is challenging to decide which index to use without having a cost-based optimizer that can estimate the selectivity of each predicate. For instance, would using an R-tree on the `sender-location` field outperforms using a B^+-tree on the `send-time` field, or vice versa?

2) Searching a secondary index using its corresponding predicate is performed irrespective of the other predicates. In some cases, this could cause too many entries to be retrieved from the secondary index, followed by probing the primary index for each entry, to find out that many of the base records are actually do not satisfy the other query predicates. This can lead to unacceptable performance for queries that ought to have real-time responses.

Figure 5.11(c) shows another possible execution strategy, which is to maintain multiple secondary indexes (e.g., an R-tree on the `sender-location` field and a B^+-tree on the `send-time` field) and use all the applicable indexes to fetch the lists of qualifying primary keys. Then, an intersection operation between those lists can be performed to obtain the primary keys of the records that satisfy both the corresponding query predicates. The resultant primary keys are then used to lookup the base records from the primary index. Performing the intersection at this early stage can reduce the number of needed primary index lookups, improving the total query time. However, there are several possible drawbacks for this strategy as well:

1) It is not clear whether using all the secondary indexes will always yield a good performance. For example, in the case when there is one or more unselective predicates, searching the corresponding secondary indexes may require a considerable amount of

time, which could negatively impact the total query time, again emphasizing the need for a cost-based optimizer to handle such cases well.

2) As shown in Section 4.4.1.1, for each additional secondary index that is added to the dataset, the ingestion throughput will decrease due to the overhead of maintaining all of the secondary indexes. For ingestion-intensive workloads (such as the workloads that AsterixDB is targeting), this can be an important factor that must be considered.

Note that the query plans shown in Figure 5.11(a) and Figure 5.11(b) are already supported in AsterixDB, but not the multi index-based plan shown in Figure 5.11(c). We leave an experimental study of the multi index-based plan as possible future work.

5.4 LSM-based Filters for Accelerating Queries

In this section, we propose a solution that leverages the structure of an LSM-based index to accelerate queries. Our solution works best with monotonically increasing sequence of values, e.g., time-correlated fields such as UUID and datetime fields, making it a perfect fit for providing near real-time performance for recency queries such as the spatial aggregation query of Figure 5.10.

5.4.1 Basic Idea

Since an LSM-based index naturally partitions data into multiple disk components, it is possible, when answering certain queries, to exploit partitioning to only access some components and safely filter out the remaining components, thus reducing query times. This can be achieved by augmenting each LSM disk component with additional information (called a "filter" hereafter) about one or more other fields of the record (called the filter's key here-

after). Consequently, the index can filter out entries based on two dimensions, one based on the original index key(s), and the second based on the filter's key.[1]

To support filtering, each LSM disk component has an associated filter record that maintains the minimum and maximum filter key values for the records contained in the component. Then, when accessing the index to answer a query, the index lookup operation can first leverage the associated filter records to prune components that do not match the filter's predicate. Then, the normal search on the index need only be performed for those components that survive the filtering process. Notice that filters can be used with both primary and secondary indexes for additional pruning power.

One of the main use cases for LSM-based filters is to use them to index time-correlated fields or monotonically increasing sequences, such as datetime fields. In this case, the filters on the disk components are most likely to have disjoint minimum and maximum values, making them very effective for pruning. In the following, we provide two examples that show the benefits of using filters with LSM indexes.

Example 1: Suppose that users are commonly interested in retrieving recent tweets from the `Tweets` dataset that were posted from specific users based on their sending time (e.g., tweets posted by "John Smith" in the last 24 hours). Figure 5.12 shows the AQL query for this example. We can create a filter on the `send-time` field of the primary index, and utilize the components' filter records to quickly prune components that do not match the temporal predicate.

Example 2: Consider the spatial aggregation query, shown in Figure 5.10. We can maintain a secondary LSM R-tree index on the `sender-location` field and create a filter on the `send-time` field of the primary and secondary indexes. When answering the spatial aggregation query, we can first access the LSM R-tree, using the index components' filter

[1]The current release of AsterixDB (Release 0.8.6) supports the creation of single-field filters. Allowing multi-field filters is a straightforward extension and it is planed for a future work.

```
for $tweet in dataset Tweets
where $tweet.send-time > datetime("2014-08-14T00:00:00Z")
and $tweet.user.name = 'John Smith'
return $tweet
```

Figure 5.12: A query that returns all the tweets posted by "John Smith" in the last 24 hours (assuming the current date is August 15th, 2014).

records to quickly filter out those components that do not match the temporal predicate of the query, and then search the remaining R-tree components that survive the filtering process. Similarly, when using the resulting primary keys to probe the primary index, the lookup operation can filter out all older disk components of the index and only probe those components that satisfy the temporal predicate.

Clearly from the above examples, using LSM-based filters can help improve the performance of queries by pruning as many records as possible in an early stage. They also can improve the ingestion performance by not maintaining a secondary index in some scenarios (as in the first example). In addition, they can potentially improve the ingestion performance under concurrent queries by reducing the contention on system resources (CPU and I/O) due to component filtering.

5.4.2 LSM-based Filters in AsterixDB

In this section, we discuss the implementation of LSM-based filters in AsterixDB. We start by describing how to enable filters for a dataset at the logical level. We then explain the changes that we had to make to the system's internals to incorporate filters in AsterixDB's indexes.

5.4.2.1 User Model

We have added support for LSM-based filters to all of AsterixDB's index types. To enable the use of filters, the user must specify the filter's key when creating a dataset, as shown in Figure 5.13. Filters can be created on any totally ordered datatype (i.e., any field that can be indexed using a B^+-tree), such as integers, doubles, floats, UUIDs, datetimes, etc.

```
create dataset Tweets(TweetType) primary key tweetid with filter on send-time;
```

Figure 5.13: Creating a dataset of tweets with a filter on the `send-time` field.

The name of the filter's key field is persisted in the "dataset" dataset (which is the metadata dataset that stores the details of each dataset in an AsterixDB instance) so that DML operations against the dataset can recognize the existence of filters and can update them or utilize them accordingly. Creating a dataset with a filter in AsterixDB implies that the primary and all secondary indexes of that dataset will maintain filters on their disk components. Once a filtered dataset is created, the user can use the dataset normally (just like any other dataset). AsterixDB will automatically maintain the filters and will leverage them to efficiently answer queries whenever possible (i.e., when a query has predicates on the filter's key).

5.4.2.2 Maintaining the Filters

In an LSM index, there are three possible methods where a new disk component is created: through a flush, merge, or fresh bulk-load operation. To guarantee the correctness of a dataset's filters, each of these operations must ensure that the filter associated with a new disk component has the minimum and maximum filter key values of the records contained therein. In the following, we show how each of those operations maintains these values efficiently:

1. Flush: Each index incrementally maintains the minimum and maximum filter key values for the records contained in its in-memory component. Those values must be updated, as needed, with each insert and delete operation against the index. When the in-memory component is flushed to disk, its filter record is also flushed with it to disk.

2. Merge: The minimum and maximum values for the resulting component's filter record are taken from the smallest and largest values of all components that are participating in the merge.

3. Bulk-load: Each added entry simply updates the filter record's minimum and maximum values as needed.

Notice that the leaf node entries in secondary indexes in AsterixDB are of the form $e = \langle SK, PK \rangle$, where SK is the secondary key and PK is the associated primary key. Although this form is still maintained in the presence of filters, we had to change the insert, delete, and load plans for indexes with filters so that the filter key values are passed to the index with each record for the purpose of updating its components' filter records.

5.4.2.3 Query Processing

We have added a new rewrite rule to the AsterixDB optimizer that checks whether or not a query can be optimized to use filters. The rule first checks to see if the queried dataset has a filter. If so, then it analyzes the query in the hope of finding an applicable query predicate. In particular, the rule searches for predicates on the filter's key that use one of the following operators: >, >=, <, <=, =. If it finds such predicates, it modifies the plan so that the filter's predicates are passed to all the indexes of the dataset that appear in the query plan (in conjunction with other predicates that are normally used to search the indexes).

At runtime, the filter's predicates will be used to prune all the components that do not match them. Note that the implementation of AsterixDB's LSM indexes was designed from the start to allow each index operation (e.g., search) to choose the components that it needs to perform its logic against, and this is where component filtering takes place. In other words, the search cursors for LSM indexes did not have to be changed in order to support filters. Once the components not matching the filter's predicates have been filtered out, then a normal search on the index is performed for the remaining components.

5.4.2.4 Effect of Merge Policies

Merge policies play an important role with respect to the ingestion and query performance of LSM indexes. This role is even more important for LSM-based filters to be effective. Here we discuss the tradeoffs of the various AsterixDB merge policies and introduce a new merge policy that further improve query times when using filters.

To understand the effect of merge policies on filters, let us revisit a similar concept that has been heavily used in many database systems, which is table partitioning. In these systems, partitioning is mainly used to mitigate the impact of table scans. A table can be divided into smaller partitions based on a partitioning key. Queries with predicates on the partitioning key can then prune those partitions that do not satisfy the predicates, thus reducing query times. Similarly, when using filters, each LSM disk component is essentially treated as a partition. Thus, the number and size of disk components can greatly impact query performance.

Generally speaking, having very few disk components will reduce a filter's pruning power for highly selective queries on the filter's key. On the other hand, less selective queries on the filter's key will perform best when the number of disk components is minimal. As mentioned in Section 4.2.5, the current release (Release 0.8.6) of AsterixDB supports three

merge policies. The constant merge policy tries to keep as few disk components as possible by constantly merging all the disk components into a single disk component. This has been shown in Section 4.4 to have a negative impact on the ingestion performance since merges are CPU and I/O intensive operations. In addition, the constant merge policy will also have a negative impact on the performance of queries when using filters since the chances of accessing all the disk components becomes very high, eliminating the benefits of filters.

In contrast, when used in conjunction with filters, AsterixDB's no-merge policy can provide excellent performance for selective queries on the filter's key; this is due to the high chance of pruning many of the components. However, similar to the constant merge policy, the no-merge policy has proven to provide bad ingestion performance due to the time spent searching the many disk components before each primary key insert to enforce key uniqueness, making the no-merge a bad choice for write-intensive workloads.

The AsterixDB prefix merge policy, which relies on component sizes and the number of components to decide which components to merge, has proven to provide excellent performance for both ingestion and queries. However, when evaluating our filtering solution with the prefix policy, we observed a behavior that can reduce filter effectiveness. In particular, we noticed that under the prefix merge policy, the disk components of a secondary index tend to be constantly merged into a single component. This is because the prefix policy relies on a single size parameter for all of the indexes of a dataset. This parameter is typically chosen based on the sizes of the disk components of the primary index (e.g., 1GB in our experiments), which tend to be much larger than the sizes of the secondary indexes' disk components. This difference caused the prefix merge policy to behave similarly to the constant merge policy (i.e., relatively poorly) when applied to secondary indexes. Consequently, the effectiveness of filters on secondary indexes was greatly reduced under the prefix-merge policy, but they were still effective when probing the primary index.

Based on this behavior, we developed a new merge policy, an improved version of the prefix

policy, called the correlated-prefix policy. The basic idea of this policy is that it delegates the decision of merging the disk components of *all* the indexes in a dataset to the primary index. When the policy decides that the primary index needs to be merged (using the same decision criteria as for the prefix policy), then it will issue successive merge requests to the I/O scheduler on behalf of all other indexes associated with the same dataset. The end result is that secondary indexes will always have the same number of disk components as their primary index under the correlated-prefix merge policy. This has improved query performance, as we will see in Section 5.5.5, since disk components of secondary indexes now have a much better chance of being pruned.

5.5 Experiments

This section presents the results of an experimental evaluation of the filtering solution designed and implemented in AsterixDB. We focus on query workloads that target recent data within very large datasets. We show the effect of using filters with different workload and physical design settings, e.g., when using a dataset with or without different secondary indexes types, signifying the applicability and effectiveness of using filters with any LSM index structure. We also present how the different merge policies can impact the filter's pruning power. In addition, we experimentally show the effect of records' arrival order on recency queries.

5.5.1 Data Generation

As in the previous chapter, we used AsterixDB's data feeds to populate a dataset using streams of synthetic tweets, with an average tweet size of 1KB. The generated tweets conform to the type definition shown in Figure 5.14, which is a slightly modified version of the type

definition shown in Figure 5.1. There are two differences between those two type definitions. First, at the time of conducting our experiments, support for the creation of an index on nested records, e.g., on the field `user.name`, was a feature that was still under development in AsterixDB. Thus, as a temporary solution, we added the `userid` field of type `int32` to the `TweetType` to create a secondary index on this field for some of our experiments. Second, we changed the type of `tweetid` to `int64` to simplify the creation of unique primary keys.

```
create type TwitterUserType as open {
    screen-name: string,
    lang: string,
    friends_count: int32,
    statuses_count: int32,
    name: string,
    followers_count: int32
};

create type TweetType as closed {
    tweetid: int64,
    user: TwitterUserType,
    sender-location: point?,
    send-time: datetime,
    referred-topics: {{ string }},
    message-text: string,
    userid: int32
};
```

Figure 5.14: Tweet type definition used in our experiments.

In the following we describe how we generated those fields of a tweet that can have an impact on the selectivity of query predicates and, as a result, on query times:

1) `tweetid`: We generated tweets with monotonically increasing ids.

2) `userid`: We chose user ids randomly with replacement from the range [0-4999].

3) `sender-location`: Although it was an optional field, all the generated tweets had

locations that were generated randomly from a distribution similar to the one described in Section 4.4.2.

4) `send-time`: Each tweet was populated with the exact time of actual tweet generation.

The remaining fields were generated with random content since they were not used in the predicates of the queries that we issued.

5.5.2 Machine and Parameter Configurations

We used two machines to conduct the experiments. The first machine was used to generate the synthetic tweets which were sent over the network to the second machine that hosted a single-machine AsterixDB instance for ingestion. The latter was an IBM machine with a 4-core Xeon 2.27 GHz CPU, 12GB of main memory, and four locally attached 10,000 rpm SATA drives. We dedicated one disk for use by the transaction log manager for writing log records, with the three remaining disks dedicated as data storage disks for separate partitions of the `Tweets` dataset and their associated secondary index partitions. We used the same parameters as shown in Table 4.1 to configure the datasets. Unless otherwise specified, we also used AsterixDB's prefix merge policy to manage the disk components of all the dataset's indexes.

5.5.3 Query Generation

For all of our experiments, we ingested a total of 70GB worth of synthetic tweets into the targeted dataset. We then stopped data ingestion and issued queries with multiple predicates on different fields, with one of the predicates being based on time-recency to mimic a query workload that is targeting recent data. Since the ingestion had already stopped, the recency

predicate was chosen based on the largest ingested `send-time` value. For example, if the largest ingested `send-time` value was `datetime("2014-04-26T10:10:00Z")`, and the goal was to retrieve all tweets ingested in the last 1 minute, then the corresponding query predicate would be "retrieve all tweets with `send-time` values that are larger than `datetime("2014-04-26T10:09:00Z")`".

We experimented with multiple recency predicates with varying selectivities ranging from a predicate that only matches the most recent data (e.g., the last 8 seconds) all the way to a predicate that matches all of the ingested data. To do that, we chose an initial time unit (e.g., 8 seconds) that served as the smallest recency predicate width (e.g., all tweets ingested in the last 8 seconds). We then multiplied this time unit by different factors to increase the window size of the recency predicate. For instance, for those experiments that used an initial time unit of 8 seconds, we used the following time-unit multipliers: 1, 6, 42, 180, 540, and 2160. Thus, the corresponding recency predicates that we tried were: last 8 seconds, last 48 seconds, last 5.6 minutes, last 24 minutes, last 72 minutes, and last 288 minutes. The multipliers used were inspired by the user interface for the power monitoring company use case that we mentioned at the start of this chapter. Unless otherwise specified, we have used an initial time unit of 8 seconds for all the experiments. The query times that we show are averaged across 200 random queries using different time unit factors.

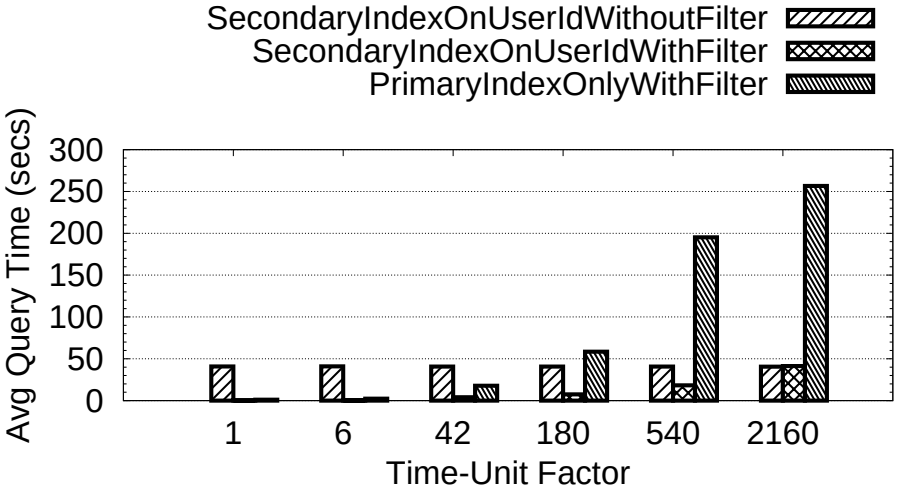
We plotted the results using histogram charts, where the X axis represents the different time-unit factors and the Y axis represents the average query time. Since the histograms have extremely high and low query time values, we plotted each chart twice, with linear and log scales.

5.5.4 Query Performance

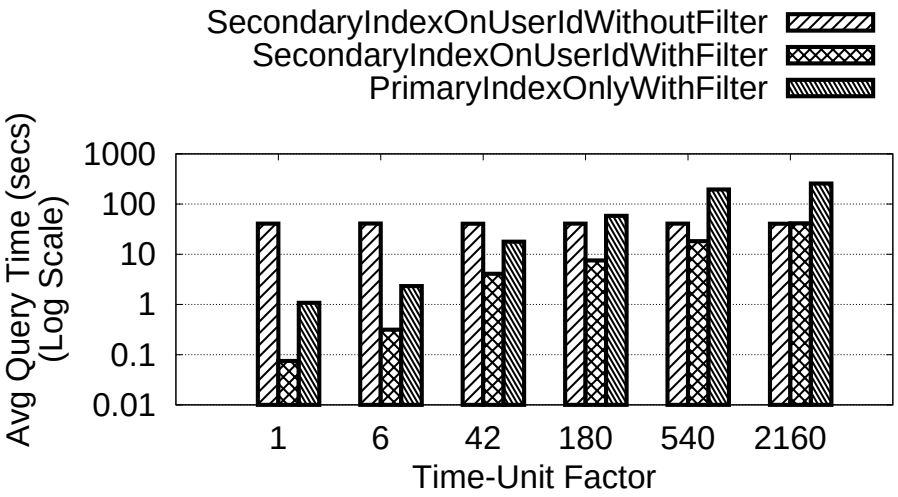
Figure 5.15 shows the average query time for running 200 random queries, similar to the one shown in Figure 5.16, against a dataset with no secondary indexes but with a filter on the `send-time` field, against a dataset with a secondary LSM B^+-tree on the field `userid`, and against a dataset with both a secondary LSM B^+-tree on the field `userid` and a filter on the field `send-time`.

The dataset that had no filter on `send-time` always provided the same average query time no matter how selective the recency predicate was. This is because the recency predicate was only used as a post-processing filtering step after fetching the candidate entries from the secondary index on the `userid` predicate and then using those entries to probe the primary index.

Both datasets that had filters on `send-time` benefited greatly from filters when the recency predicate was selective due to the pruning power of filters. For instance, for the time-unit factor one, all disk components of the corresponding indexes were pruned, resulting in searching only the in-memory component of the index. However, as we decreased the selectivity of the recency predicate by using larger time-unit factors, the effectiveness of filters decreased. This is an expected behavior since the less selective a recency predicate is, the more disk components that satisfy the predicate and thus must be searched. That being said, even for the extreme case when the recency predicate is unselective, filters do not hurt query performance and provide the same performance as if the filters did not exist. This can be seen when using the time-unit factor 2160, where all the records satisfy the recency predicate. For the dataset that had a filter but no secondary index, the performance was identical to the performance of a dataset scan. For the dataset that had both a secondary index and a filter, the unselective time-unit query performance was identical to the performance of using a secondary index on `userid` without a filter.



(a) Linear Scale.



(b) Log Scale.

Figure 5.15: Average query time when using a dataset with a secondary LSM B$^+$-tree on `userid`, a dataset with a filter, and a dataset with both a secondary LSM B$^+$-tree on `userid` and a filter on `send-time`.

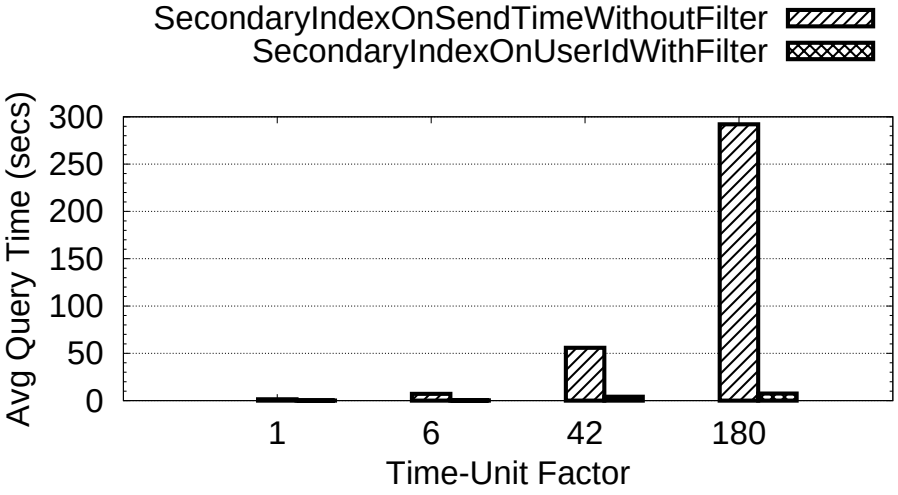
Next, Figure 5.17 shows the performance of queries when using a dataset with a secondary B$^+$-tree on the field `send-time` compared to a dataset that was using both a secondary LSM B$^+$-tree on the field `userid` and a filter on the field `send-time`. Clearly the dataset with both a secondary index and a filter outperformed the dataset with a secondary index on the `send-time` for all the time-unit factors. We do not show larger time-unit factors since their query times against the dataset without a filter were so high that we decided to

```
let $count := count (
    for $tweet in dataset Tweets
    where $tweet.send-time > datetime("2014-06-21T07:49:39.349Z")
    and $tweet.userid = 3319
    return $tweet
)
return {"NumTweets": $count}
```

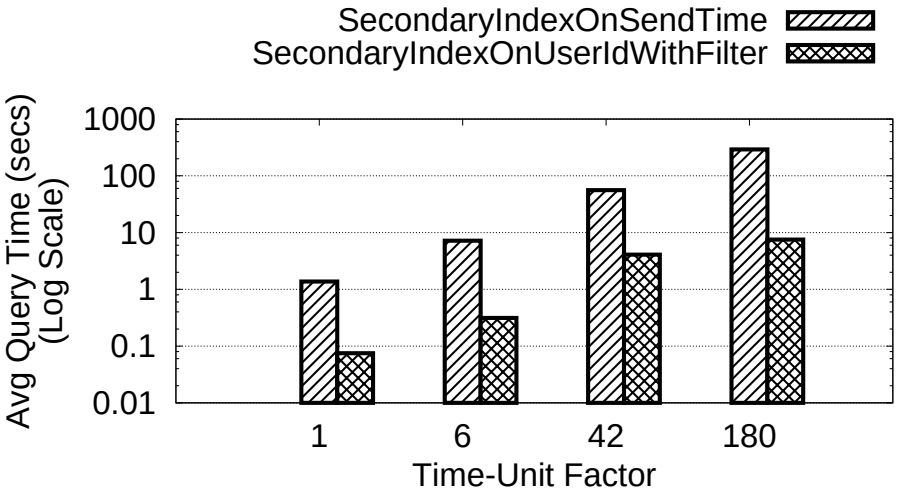
Figure 5.16: An example of an aggregation query that we used in our experiments to measure the effectiveness of filters against different datasets.

stop the experiment and repeat it without those large time-unit factors. The reason why the dataset with both a secondary index and a filter outperformed the other dataset is because it used both query predicates to prune a larger number of index entries at the secondary index, resulting in much fewer primary index probes.

Finally, Figure 5.18 shows the average query time for spatial aggregation queries, similar to the one shown in Figure 5.19, against two datasets that are both using a secondary LSM R-tree on the field `sender-location`; one of them was also utilizing a filter on the field `send-time`. In order to generate random spatial predicates, we used the same approach that we presented in Section 4.4.2. The results here confirm our previous findings. A dataset that is not using a filter will always provide similar query times regardless of the selectivity of the recency predicate. On the other hand, the filters are very effective when used to answer selective queries, and their effectiveness decreases as the selectivity decreases. Note that the slight variation in the query times for the different time-unit factors for the dataset without a filter is due to the heuristic approach that was used to generate the spatial predicates, as it does not always provide the exact requested selectivity but was good enough to sever our purposes here.



(a) Linear Scale.

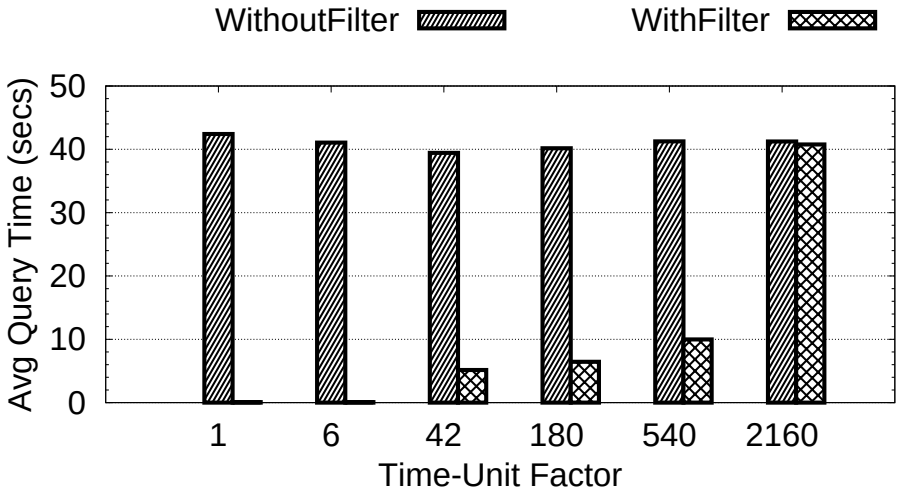


(b) Log Scale.

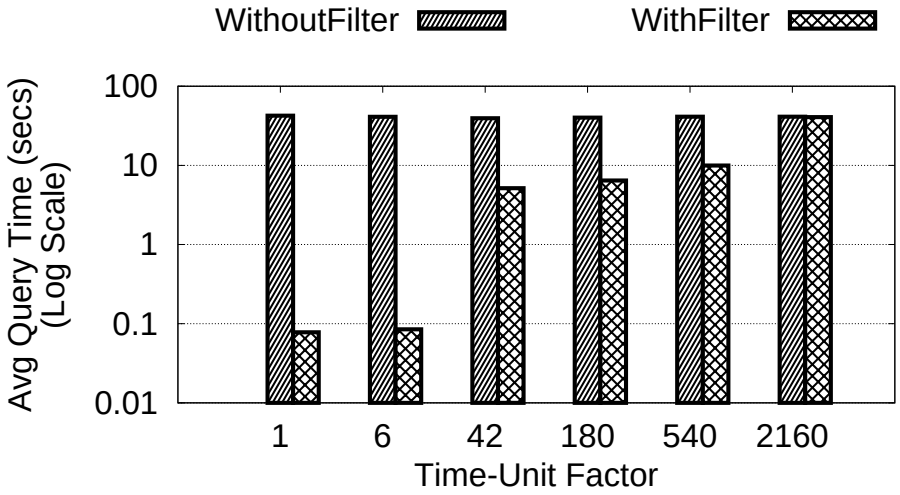
Figure 5.17: Average query time when using a dataset with a secondary LSM B$^+$-tree on `send-time` and a dataset with both a secondary LSM B$^+$-tree on `userid` and a filter on `send-time`.

5.5.5 Impact of Merge Policies

Next, we show the impact of using different merge policies with filters. Each dataset had a secondary LSM B$^+$-tree on `userid`. Since the handled TPS when using a different merge policy is widely varied (the total time for ingesting the 70GB worth of tweets was 110 minutes and 26 hours for the prefix and no-merge policies, respectively), the selectivity of recency



(a) Linear Scale.



(b) Log Scale.

Figure 5.18: Average time to answer spatial aggregation queries when using a dataset with a secondary LSM R-tree on `sender-location` and a dataset with both a secondary LSM R-tree on `sender-location` and a filter on `send-time`.

queries of the same time-unit factor, as a result, is also widely varied. Thus, we added a new field called `integer-send-time` to the `TweetType` definition and used it as the filter's key (it was populated with monotonically increasing integer values). This way, we were able to completely isolate the impact of TPS variance on the selectivity of queries, allowing for an accurate comparison between the different merge policies. We also used an initial time unit of 32,000 for all of the experiments in this section. (As an example, for

```
for $tweet in dataset Tweets
where $tweet.send-time > datetime("2014-07-03T08:16:27.620Z")
and spatial-intersect($tweet.sender-location, create-rectangle(
create-point(42.73,-80.84), create-point(43.23,-80.34)))
group by $c := spatial-cell($tweet.sender-location, create-point(42.73,-80.84),
0.1, 0.1)
with $tweet
let $num := count($tweet)
return { "Cell": $c, "NumTweets": $num }
```

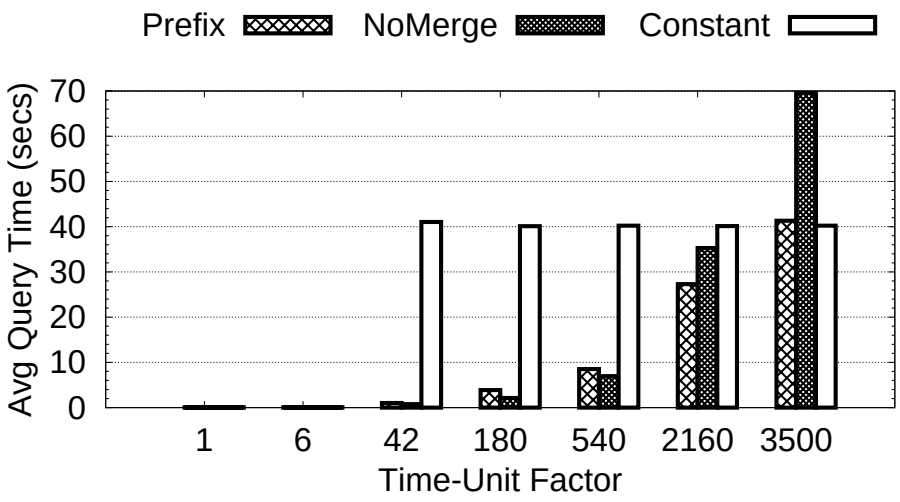
Figure 5.19: An example of a spatial aggregation query that we used in our experiments to measure the effectiveness of filters when combined with an LSM R-tree.

the time-unit factor 540, the corresponding recency predicate would ask for all tweets with `integer-send-time` values that are 540 x 32000 = 17280000 smaller than the largest ingested `integer-send-time` value.)

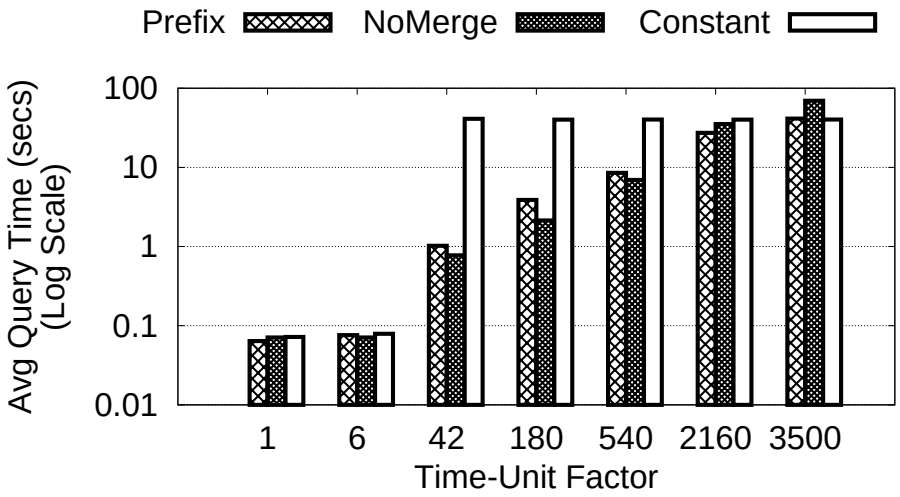
Figure 5.20 shows the average query time for answering recency queries, similar to the one shown in Figure 5.21, for datasets that were using different merge polices. All the datasets provided identical average query time for the smaller time-unit factors, 1 and 6, because all the disk components were pruned by the filters since they did not match the recency predicates.

Clearly, the performance differences are huge when the recency predicates are less selective and when, as a result, disk components must be accessed. The constant merge policy always provided the same high average query time for the larger factors. This is because there was a single disk component for both the primary and secondary indexes at the end of the ingestion, and thus, the pruning power of the filters were lost.

Both the no-merge and prefix policy provided query times that are relative to the number of accessed disk components. For the middle time-unit factors (i.e., 42, 180, and 540), the prefix policy slightly suffered from having only three disk components for the secondary index, reducing the effectiveness of its filters. The index had only three disk components



(a) Linear Scale.



(b) Log Scale.

Figure 5.20: Average query time when using constant, no-merge, and prefix merge policies on a dataset with both a secondary LSM B$^+$-tree on `userid` and a filter on `integer-send-time`.

since, as described in Section 5.4.2.4, this policy tends to behave similar to the constant policy when using large values for its size parameter. On the other hand, in the case of the no-merge policy, the queries had to touch fewer secondary index entries due to the many disk components that were pruned, and as a result there were much fewer primary index probes. However, as the selectivity of the recency predicate decreased, the no-merge policy had to pay for accessing many smaller disk components. This can be seen clearly in the case of the

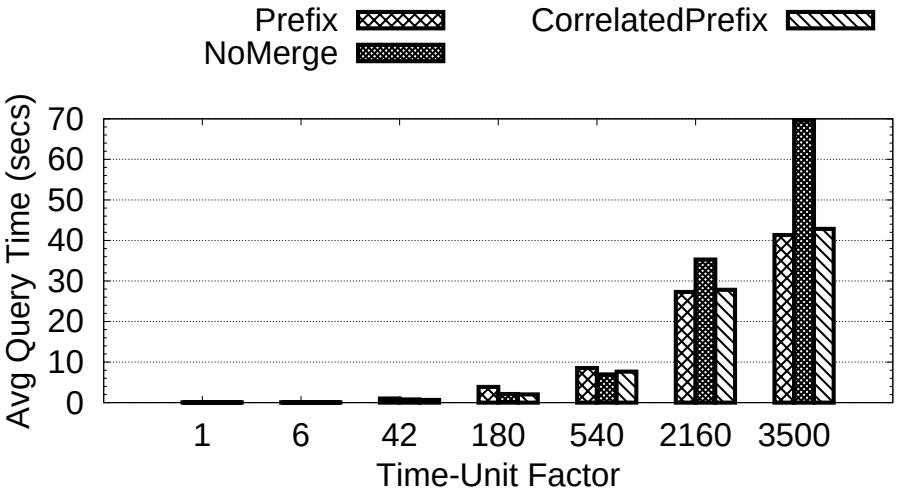
```
let $count := count (
    for $tweet in dataset Tweets
    where $tweet.integer-send-time > 105477969
    and $tweet.userid = 3319
    return $tweet
)
return {"NumTweets": $count}
```

Figure 5.21: An example of an aggregation query that we used in our experiments to compare the impact of different merge polices on filters.

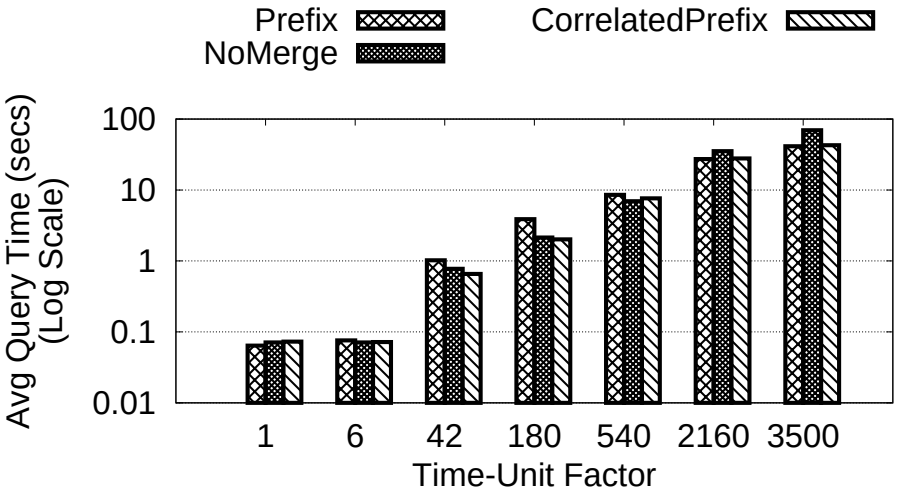
largest two time-unit factors, where 158 and 242 out of 242 disk components (for both the secondary and primary indexes) had to be accessed for the 2160 and 3150 time-unit factors, respectively.

Figure 5.22 shows how the new correlated-prefix merge policy combines the good characteristics from both the prefix and no-merge polices. For the middle time-unit factors (i.e., 42, 180, and 540), the correlated-prefix policy behaved similar to the no-merge policy in the sense that it benefited from having multiple disk components for the secondary index. For the larger time-unit factors, 2160 and 3500, it behaved in a manner similar (with minor overhead) to the prefix policy in the sense that it does not have too many smaller disk components (there were 22 secondary index disk components).

Figure 5.23 shows how the LSM R-tree behaved when using the correlated-prefix policy. The results confirm our findings; under the correlated-prefix policy, the selective recency predicates benefited from having multiple smaller R-tree components. Clearly, the performance differences here are larger compared to those for secondary LSM B^+-tree. This is due to the fact that searching an R-tree is more expensive than searching a B^+-tree. For very large time-unit factors, however, the queries did not benefit from the filters. In the case, having very few disk components can help query performance, as can be seen in the case of the prefix policy for the time-unit factor 3500.



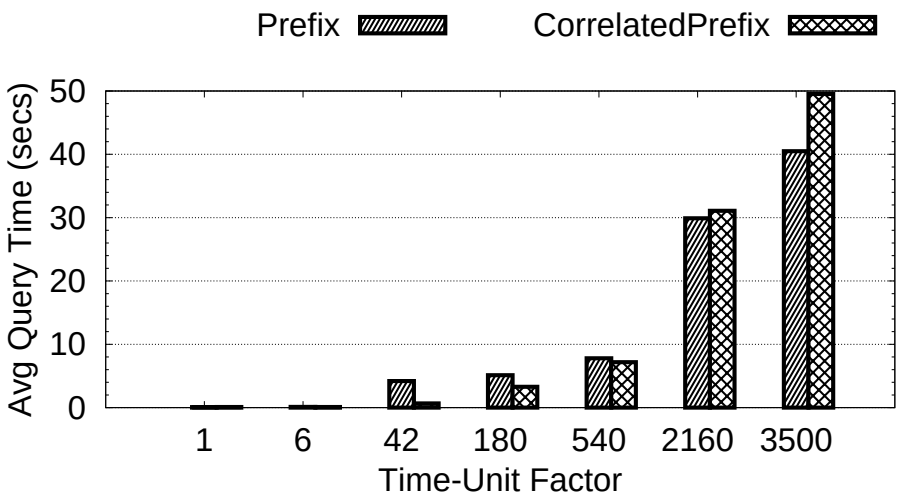
(a) Linear Scale.



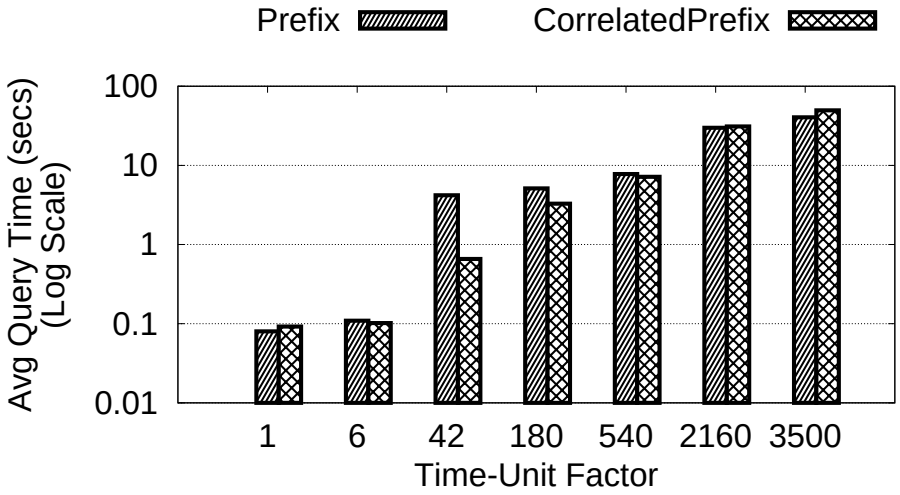
(b) Log Scale.

Figure 5.22: Average query time when using the correlated-prefix compared to the no-merge and prefix policies on a dataset with both a secondary LSM B$^+$-tree on `userid` and a filter on `integer-send-time`.

In addition to the query performance results, it is worth mentioning that the correlated-prefix policy helped to reduce the ingestion time by around 10%. This resulted from its reducing the number of secondary index merge operations, and, as a result, reducing the system's CPU and I/O contention.



(a) Linear Scale.



(b) Log Scale.

Figure 5.23: Average query time for answering spatial aggregation queries when using the prefix and correlated-prefix merge policies on a dataset with both a secondary LSM R-tree on `sender-location` and a filter on `send-time-int`.

5.5.6 Effect of Out of Order Records

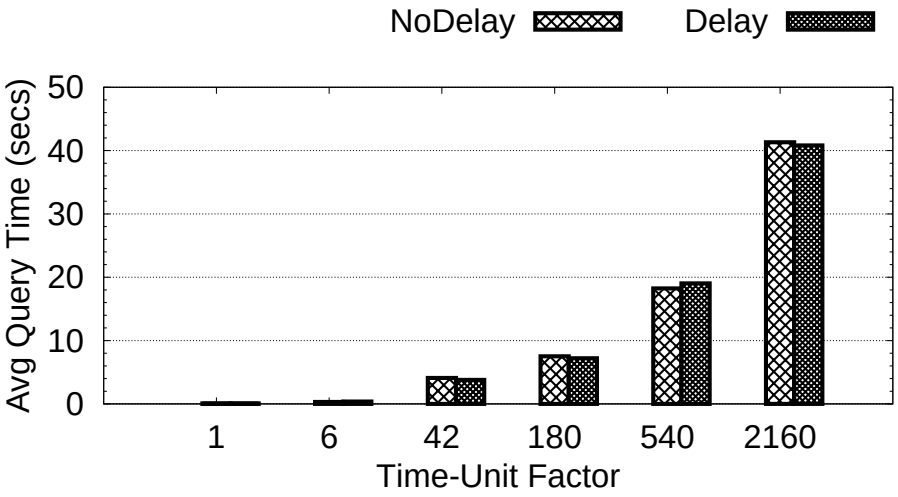
In many applications, data records, e.g., sensor readings, are sent over a network to the target dataset for ingestion. Records can arrive out-of-order due to network delays or faulty sensors. Here we provide experimental results to investigate whether such order variations have any impact on the effectiveness of filters. To do that, we have randomly deducted a

value that ranges from 0 to 60,000,000 milliseconds (10 minutes) from the `send-time` value of each tweet (note that the total ingestion time was around 110 minutes). The dataset had a secondary LSM B$^+$-tree on `userid` and a filter on `send-time` and the queries that we used are again similar to the one shown in Figure 5.16.

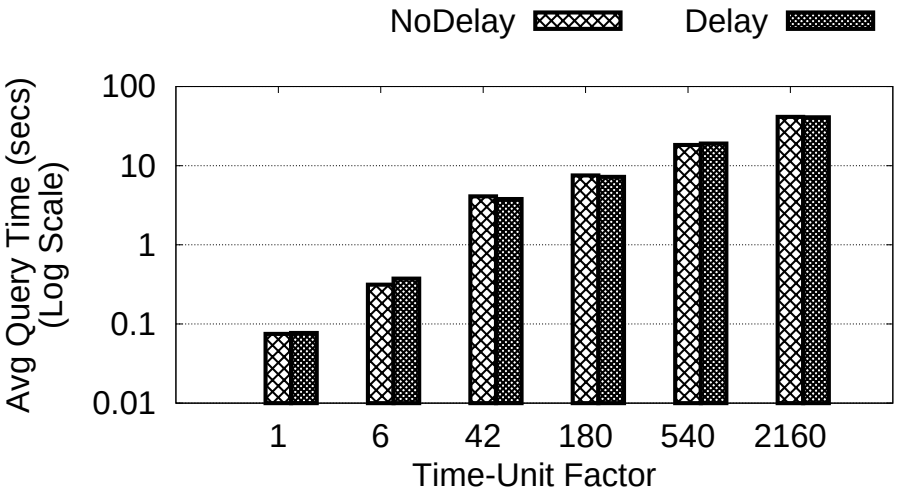
Figure 5.24 shows the result. Clearly, recency queries were not effected by delayed records. The reason is because the delayed records only effect the minimum values of the filters. However, recency predicates (which use "larger than" semantics) are only compared with the maximum values of the filters. This implies that, if a delayed record satisfies a recency query, the disk component where the record would have landed (if it was not delayed) plus all the newer disk components will be accessed anyway. In other words, no matter which disk component a delayed record lands in, a recency query will always access the same number of disk components. Thus, the query times will not be affected.

5.6 Conclusions

In this chapter, we have presented an efficient solution for answering query workloads that favor recent data, allowing the AsterixDB system to provide near real-time performance for recency queries. We have designed and implemented filters for all of AsterixDB's index types, including LSM B$^+$-tree, LSM R-tree, and LSM inverted indexes for textual data. Our solution exploits the fact that LSM indexes partition their data into successive disk components based on their freshness. By maintaining filters on disk components of the LSM indexes, we were able to reduce recency query response times by up to 99% (for very selective queries that only access the in-memory component of the index). In addition, our solution's experimental evaluation included experiments to examine the impact of merge policies, including the new correlated-prefix policy, and out-of-order records on the performance of filters.



(a) Linear Scale.



(b) Log Scale.

Figure 5.24: Effect of out-of-order records on filters.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The era of social media is upon us. A wealth of new data, mostly in unstructured format, is being generated at unprecedented rate (especially due to the increasing adoption rate of smart phones), and much of this data contains rich data types such as locations and timestamps in addition to textual content. Consequently, new applications and features have been rapidly appearing, and as a result new requirements and expectations must be met. Those requirements have created challenging new problems for traditional database systems, including spatial databases, due to new location-based services that are being born out of this social-media movement. In this thesis, we have responded to some of the challenges that are facing spatial databases by developing several spatial indexing techniques.

In Chapter 3 we studied how to efficiently answer location-based approximate-keyword queries. Data and queries of location-based search engines could have errors and typos, and thus it could be hard for users to find the entities they are searching for. We developed a solution that combines a tree-based index structure, such as the R-tree, with approximate-

keyword capabilities for answering both approximate and exact queries. In particular, we developed three algorithms, where each one builds on the previous algorithm, successively improving the time and space efficiency by exploiting the textual and spatial properties of the data. Our experiments have shown the effectiveness of our solution in returning all relevant answers efficiently.

In Chapter 4 we focused on developing a spatial index structure that can ingest fast data efficiently. First, we showed that converting an R-tree index (and other non-totally ordered indexes) to an LSM index is non-trivial if the resultant index is expected to have performant read and write operations. We then presented an indexing framework to enable the "LSM-ification" of any kind of index structure that supports certain primitive operations, enabling the index to ingest data efficiently. Our framework has been used to "'LSM-ify" all the indexes of AsterixDB, making it an all-LSM storage engine. We also presented some of the key ideas of the concurrency and transaction techniques that we developed for AsterixDB's indexes. Our experimental results have shown that an LSM-based version of the R-tree can significantly outperform its conventional counterpart for both ingestion and query speed.

In Chapter 5 we described how we can exploit AsterixDB's LSM-based indexes to provide near-real time performance for query workloads that favor recent data within very large datasets. Use cases for recency queries include applications such as real-time data analysis, spatial aggregation queries, etc. In many of those applications, providing real-time or near real-time query performance is critical to the core functionality of the application. Our solution utilizes the natural temporal partitioning property that LSM indexes provide, and does so by installing filter records on the disk components of the index. Those filters can be used at query time to prune all components that do not satisfy the applicable query predicates. Our experiments have shown that applying filters on both primary and secondary indexes can reduce recency query times by up to 99%, making them very effective for answering real-time query workloads.

6.2 Future Work

In Chapter 4 we have shown that merge policies play an important factor in the efficiency of ingestion and queries for LSM indexes. In AsterixDB, a single merge policy is chosen per dataset, and all the associated indexes will use the same policy to merge their disk components. One possible future direction is to allow each secondary index to use its own merge policy. Having a special merge policy per index could be useful for certain complex indexes, such as the LSM R-tree, where a size-based merge policy might not be always the right answer. Instead, it would be possible to introduce new merge policies specific to the LSM R-tree, policies that exploits the spatial properties of the index entries, to allow for better partitioning of spatial records to improve data ingestion and query performance.

Another possible related direction could be to add a new maintenance operation to the LSM indexes of AsterixDB, called *partition*, which is the inverse of the merge operation. Given a partitioning policy, the partition operation would receive as input n disk components and produces m disk components. The goal of this operation would be to cluster the index entries based on a criteria other than size or number of components. Thus, during query times, it would be possible to avoid accessing all the disk components and as a result improve the query speed.

In Chapter 5 we have suggested the possibility of using a multi-index search plan for answering multi-predicate queries (e.g., recency queries). Using multiple secondary indexes to answer such queries can improve their speed. However, as we maintain more secondary indexes with a dataset, the ingestion throughput decreases due to the overhead of maintaining consistency between the primary and secondary indexes. Thus, it is unclear whether having multiple secondary indexes is cost-effective for ingestion-intensive workloads (such as the workloads that AsterixDB is targeting); this direction deserves a future experimental study.

Another study that deserves to be looked at in the future is to experimentally compare the

filters that we introduced in Chapter 5 with those implemented in HBase. In addition, it is also interesting to compare the filters with the solution that the power management company has implemented, mentioned in Section 5.1, which is pre-computing common aggregations on the fly. This experimental comparison should measure the performance of both data ingestion and query speed for the two solutions and study the corresponding tradeoff.

Bibliography

[1] AsterixDB. http://asterixdb.ics.uci.edu/.

[2] Cassandra. http://cassandra.apache.org/.

[3] HBase. http://hbase.apache.org/.

[4] LevelDB. https://code.google.com/p/leveldb/.

[5] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. R. Borkar, Y. Bu, M. J. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y.-S. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. J. Tsotras, R. Vernica, J. Wen, and T. Westmann. Asterixdb: A scalable, open source bdms. *VLDB*, 2014.

[6] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. R. Borkar, Y. Bu, M. J. Carey, R. Grover, Z. Heilbron, Y.-S. Kim, C. Li, N. Onose, P. Pirzadeh, R. Vernica, and J. Wen. Asterix: An open source system for Big Data management and analysis. *PVLDB*, 5(12):1898–1901, 2012.

[7] S. Alsubaiee, A. Behm, V. Borkar, Z. Heilbron, Y.-S. Kim, M. J. Carey, M. Dreseler, and C. Li. Storage management in asterixdb. *VLDB*, 2014.

[8] S. Alsubaiee, A. Behm, R. Grover, R. Vernica, V. Borkar, M. J. Carey, and C. Li. Asterix: scalable warehouse-style web data integration. In *IIWeb*, 2012.

[9] S. Alsubaiee, A. Behm, and C. Li. Supporting location-based approximate-keyword queries. In *ACM SIGSPATIAL GIS*, 2010.

[10] S. Alsubaiee, A. Behm, R. Vernica, S. Ji, J. Lu, L. Jin, Y. Lu, and C. Li. UCI Flamingo Package 4.0, 2010.

[11] S. Alsubaiee and C. Li. Fuzzy keyword search on spatial data (demo). In *DASFAA*, 2010.

[12] Apache Hive, http://hadoop.apache.org/hive.

[13] N. Beckmann, H. P. Begel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *SIGMOD*, 1990.

[14] A. Behm, A. Behm, V. R. Borkar, M. J. Carey, R. Grover, C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou, and V. J. Tsotras. Asterix: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29(3), 2011.

[15] P. Bolettieri, A. Esuli, F. Falchi, C. Lucchese, R. Perego, T. Piccioli, and F. Rabitti. CoPhIR: a test collection for content-based image retrieval. *CoRR*, 2009.

[16] V. R. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, 2011.

[17] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. W. M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM TOCS.*, 26(2), 2008.

[18] S. Chen, P. B. Gibbons, M. Kozuch, and T. C. Mowry. Log-based architectures: using multicore to help software behave correctly. *ACM SIGOPS Oper. Syst. Rev.*, 45(1), 2011.

[19] Y.-Y. Chen, T. Suel, and A. Markowetz. Efficient query processing in geographic web search engines. In *SIGMOD*, 2006.

[20] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB*, 2(1), 2009.

[21] Facebook. Facebook's growth in the past year. https://www.facebook.com/media/set/?set=a.10151908376636729.1073741825.20531316728.

[22] I. D. Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *ICDE*, 2008.

[23] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, 1984.

[24] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava. Fast indexes and algorithms for set similarity selection queries. In *ICDE*, 2008.

[25] R. Hariharan, B. Hore, C. Li, and S. Mehrotra. Processing spatial-keyword (SK) queries in geographic information retrieval (GIR) systems. In *SSDBM*, 2007.

[26] Jaql, http://www.jaql.org.

[27] C. Jermaine, E. Omiecinski, and W. G. Yee. The partitioned exponential file for database storage management. *The VLDB Journal.*, 16(4), 2007.

[28] I. Kamel and C. Faloutsos. On packing R-trees. In *CIKM*, 1993.

[29] M. Kornacker, C. Mohan, and J. M. Hellerstein. Concurrency and recovery in generalized search trees. In *SIGMOD*, 1997.

[30] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.

[31] C. Mohan. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on b-tree indexes. In *VLDB*, 1990.

[32] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS.*, 17(1), 1992.

[33] P. Muth, P. E. O'Neil, A. Pick, and G. Weikum. Design, implementation, and performance of the lham log-structured history data access method. In A. Gupta, O. Shmueli, and J. Widom, editors, *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 452–463. Morgan Kaufmann, 1998.

[34] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[35] Object database management systems. http://www.odbms.org/odmg/.

[36] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD*, 2008.

[37] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4), 1996.

[38] O. Procopiuc, P. K. Agarwal, L. Arge, and J. S. Vitter. Bkd-tree: A dynamic scalable kd-tree. In *SSTD*, 2003.

[39] W. Pugh. Skip Lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6), 1990.

[40] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *SOSP*, 1991.

[41] R. Sears and R. Ramakrishnan. bLSM: a general purpose log structured merge tree. In *SIGMOD*, 2012.

[42] D. G. Severance and G. M. Lohman. Differential files: Their application to the maintenance of large databases. *ACM TODS.*, 1(3), 1976.

[43] Twitter Blog. New Tweets per second record, and how!, August 2013. https://blog.twitter.com/2013/new-tweets-per-second-record-and-how.

[44] S. Vaid, C. B. Jones, H. Joho, and M. Sanderson. Spatio-textual indexing for geographical search on the web. In *SSTD*, 2005.

[45] www.forbes.com. 16 Surprising Statistics About Small Businesses, september 2013. http://www.forbes.com/sites/jasonnazar/2013/09/09/16-surprising-statistics-about-small-businesses/.

[46] www.forbes.com. More Than Half Of Us Have Smartphones, Giving Apple And Google Much To Smile About, June 2013. http://www.forbes.com/sites/markrogowsky/2013/06/06/more-than-half-of-us-have-smartphones-giving-apple-and-google-much-to-smile-about/.

[47] XQuery 1.0: An XML query language. http://www.w3.org/TR/xquery/.

[48] B. Yao, F. Li, M. Hadjieleftheriou, and K. Hou. Approximate string search in spatial databases. In *ICDE*, 2010.

[49] D. Zhang, B. C. Ooi, and A. K. H. Tung. Locating mapped resources in web 2.0. In *ICDE*, 2010.

[50] Y. Zhou, X. Xie, C. Wang, Y. Gong, and W.-Y. Ma. Hybrid index structures for location-based web search. In *CIKM*, 2005.