UNIVERSITY OF CALIFORNIA,
IRVINE

Transactional and Spatial Query Processing in the Big Data Era

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Young-Seok Kim

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

2016

DEDICATION

To my family.

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to acknowledge my advisors Chen Li and Michael Carey for guiding me through this work and shaping me into an independent researcher. Chen has always showed me how to examine, dissect, and develop research problems by asking right questions which are seemingly obvious but actually fundamental questions to find out and understand the core problems. Mike has been a conscientious and passionate mentor and collaborator. His unceasing and positive encouragement, thoughtful advice, and humorous anecdotes made my day enjoyable. I would like to thank Professor Sharad Mehrotra for joining my doctoral committee. His knowledge and enthusiasm have helped me to improve myself.

I am also grateful to the many people having contributed to AsterixDB project, especially students from UCI including Vinayak Borkar, Pouria Pirzadeh, Alexander Behm, Sattam Alsubaiee, Raman Grover, Yingyi Bu, Zachary Heilbron, Jianfeng Jia, Taewoo Kim, Abdullah Alamoudi, and Murtadha Hubail for our fruitful team work to design, implement, and maintain AsterixDB.

I would like to thank Samsung for providing me with a generous Samsung Fellowship during my PhD work. Also, the AsterixDB project has been supported by an initial UC Discovery grant, by NSF IIS awards 0910989, 0910859, 0910820, and 0844574, and by NSF CNS awards 1305430 and 1059436. The project has also enjoyed industrial support from Amazon, eBay, Facebook, Google, HTC, Microsoft, Oracle Labs, Infosys, and Yahoo!

I also thank TU Berlin and Odej Kao for providing my access to their cluster machines for the experiments in the comparative study of spatial indexes.

CURRICULUM VITAE

Young-Seok Kim

EDUCATION

Doctor of Philosophy in Information and Computer Science **2016**
University of California, Irvine *Irvine, California*

Master of Science in Computer Science **2004**
Korea Advanced Institute of Science and Technology *Daejeon, Korea*

Bachelor of Science in Computer Engineering **2002**
Dongguk University *Seoul, Korea*

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)

Adaptive Logging for Mobile Device **2010**
International Conference on Very Large Databases (VLDB)

Design and Implementation of an OGSI-Compliant Grid Broker Service **2004**
IEEE International Symposium on Cluster Computing and the Grid (CCGRID)

ABSTRACT OF THE DISSERTATION

Transactional and Spatial Query Processing in the Big Data Era

By

Young-Seok Kim

Doctor of Philosophy in Computer Science

University of California, Irvine, 2016

Professor Chen Li, Chair

Over the past decade, the proliferation of mobile devices has generated a variety of data at an unprecedented rate. The trend will be further accelerated by the advent of the Internet-of-Things era. Such data include signals, texts, photos, and videos tagged with date, time, and geo coordinates. The data are structured, semi-structured, or unstructured. Data-processing systems that aim to ingest, store, index, and analyze Big Data must deal with such data efficiently. In response, we have developed Apache AsterixDB, a parallel, semi-structured information management platform, that provides the ability to ingest, store, index, query, and analyze mass quantities of data.

The key contributions of this thesis fall in two major parts. First, in order to store and index newly generated data and make them queryable in a timely manner, a record-level transaction model was designed and implemented in AsterixDB based on the read-committed isolation level. Second, due to the importance of efficient query processing for such dynamic geo-tagged data, we implemented five variants of representative, disk-resident spatial indexing methods on top of the Log-Structured Merge-tree-based (LSM) storage layer in AsterixDB and evaluated their pros and cons in light of the dynamic characteristics of geo-tagged Big Data.

Chapter 1

Introduction

1.1 Motivation

Over the past decade, the proliferation of mobile devices has generated a variety of data at an unprecedented rate. Such data include signals, texts, photos, and videos that are tagged with date, time, and geo coordinates. The data are structured, semi-structured, or not structured at all. The big success of major social media services such as Facebook, Twitter, Foursquare, Flickr, Instagram, Youtube, Vimeo, Google Photos, etc, have been spurring such data generation. The trend will be further accelerated by the advent of the Internet-of-Things (IoT) era, where literally everything could involve a variety of sensors and generate such data.

Data-processing systems that aim to ingest, store, index, and analyze Big Data must deal with such data efficiently. In response, we have developed Apache AsterixDB, a parallel, semi-structured Big Data Management System (BDMS), that provides the ability to ingest, store, index, query, and analyze mass quantities of data.

During the course of developing Apache AsterixDB, in order to store and index such big and fast data, we decided to support a Log-Structured Merge-tree-based (LSM) storage layer due to its superior insert performance versus traditional in-place-update storage structures. However, in order to work as a full-fledged data-processing system, it is critical to provide certain transactional guarantees. Naturally, we had the following question that motivated the first part of our thesis study.

- What is a proper transaction model for storing, indexing, and querying such big and fast data in a timely manner?

Next, due to the value of analyzing big and fast geo-tagged data coming from mobile devices and IoT sensors, it also became critical to store, index, and query such geo-tagged data in a timely manner for data-processing systems. Therefore, considering the importance of query processing for dynamic geo-tagged data and the availability of LSM B-tree[1], LSM R-tree, and LSM inverted indexes in the LSM storage layer in AsterixDB, we had the following question that motivated the second and third parts of our study.

- Among three of the most popular disk-resident index structures, namely B-tree, R-tree, and inverted index, which is the most appropriate choice as a foundation for spatial indexing, especially for dynamic geo-tagged data, in the context of an LSM storage layer?

1.2 Thesis Preview

The thesis consists of three major parts, each of which is now previewed briefly.

[1] In this thesis, a B-tree and an LSM B-tree represents a B^+-tree and LSM B^+-tree, respectively.

1.2.1 Record-Level Transactions in AsterixDB

AsterixDB supports *record-level*, ACID transactions across multiple heterogeneous LSM indexes for a dataset. AsterixDB does not support multi-statement transactions, and, in fact, a DML (Data Manipulation Language) statement that involves multiple records will itself involve multiple independent record-level transactions. Transactions begin and terminate implicitly for each record that is inserted, deleted, or searched while a given DML statement is being executed. This fine granularity (i.e., record) of the transaction model enables newly updated records to be committed as soon as possible regardless of the number of records updated in a batch. Also, based on the *read-committed* isolation level [13, 6], AsterixDB allows queries to read records that are new or updated even after the queries have begun. In addition, a deadlock-free locking protocol has been achieved by leveraging the record-level nature of the transaction model, which made it possible to avoid the hold-and-wait situation that is a necessary condition to form a deadlock.

In the first part of this thesis, we describe the details of AsterixDB's record-level transactions, including how logging and recovery work and how concurrency control is supported. Also, we explain the detailed workflow of multi-record operations including insert, delete, select and join queries. In addition, we discuss a set of expected but possibly undesired situations that can arise under the read-committed isolation level during those operations.

1.2.2 Comparative Study of LSM Spatial Indexes for Point Data

For the second part of this thesis study, among a set of representative, disk-resident spatial indexing methods that have been adopted by major SQL and NoSQL systems, we implement five variants of these methods in the form of LSM spatial indexes in the context of AsterixDB in order to evaluate their pros and cons in light of the dynamic characteristics of geo-tagged *point* data. Three of the indexes are implemented using the LSM B-tree index, one is implemented using the LSM inverted index, and the last one is the LSM R-tree index itself. We then evaluate these five LSM spatial indexes based on real-world geo-tagged point data. The evaluation covers a broad spectrum of workloads—from a “load once, query many” case without incremental inserts to a case where continuous concurrent insertions are mixed with concurrent queries—in order to capture both static and dynamic workload characteristics. We focus solely on point data in this part of our study because point data is one of the most important data types given the amount of such data generated by today’s social-media-service applications in mobile devices.

1.2.3 Comparative Study of LSM Spatial Indexes for Non-Point Data

In the third part of the thesis, we extend the comparative study of LSM spatial indexes to evaluate their performance for dynamic *non-point* data. For this study, we first narrow down the scope of the compared indexes to include only two indexes, an LSM R-tree spatial index and an LSM B-tree-based spatial index, by excluding spatial indexes that would show obviously inferior performance based on the study results for point data and for which there are no known efficient methods to support dynamic non-point data. We describe several issues that arise with non-point data in the LSM B-tree-based spatial index and explain how to tackle the issues. We evaluate the two indexes using the similar workloads used for the evaluation with point data except for using two different datasets that have reasonably large-sized non-point data objects.

Based on these two comparative spatial index studies, we select an overall winner index among all of the compared indexes in Chapter 6.

Chapter 2

Foundations

In this chapter, we explain the foundations of our study. First, we describe the main idea of the LSM-tree and its operations. We then provide an overview of Apache AsterixDB. After that, we describe briefly how its storage layer enables the conversion of non-LSM indexes into LSM indexes and provide more details of LSM indexes such as LSM B-tree, LSM R-tree, and LSM inverted index in AsterixDB.

2.1 LSM-trees

An LSM-tree [53] is an ordered, persistent index structure that supports typical operations such as insert, delete, and search. 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 several disk seeks 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—called an *in-memory component*. When the space occupancy of the in-memory component exceeds a specified threshold, entries are sequentially *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 [53]) 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. In contrast, 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, and 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 sometimes called *memtables* and *SSTables*, respectively (e.g., in [21]). Throughout the rest of this thesis, a reference to an LSM-tree implies a reference to a component-based LSM-tree. As is the case in [53], each component is usually a B-tree. However, it would be possible to use other index structures whose operations are semantically equivalent (e.g., Skip Lists [57]) for the in-memory component. Figure 2.1 shows a life cycle of a component-based LSM-tree.

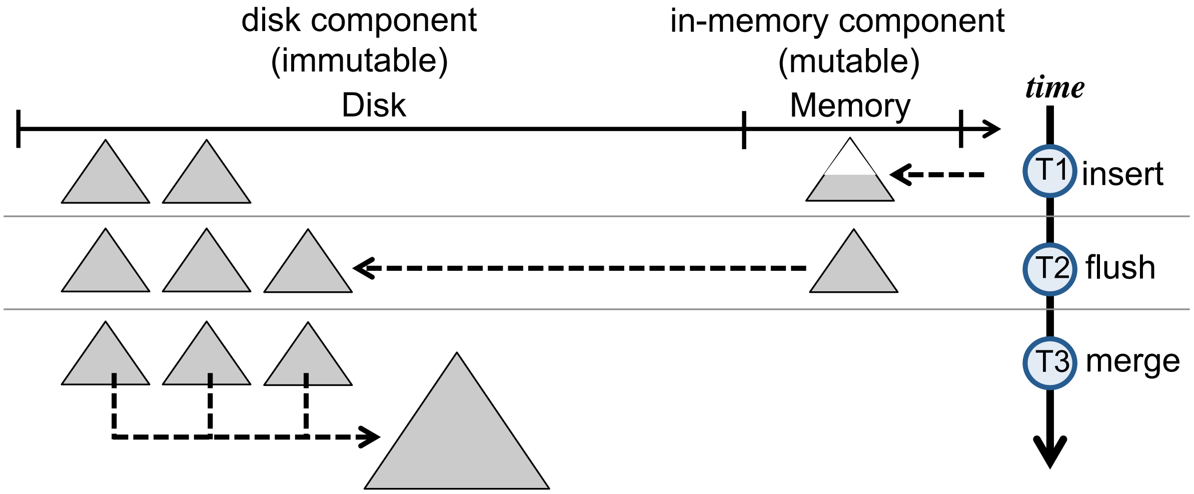


Figure 2.1: Life cycle of a component-based LSM-tree

Disk components of an LSM-tree are immutable. Modifications (e.g., updates and deletes) of existing entries are thus 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. 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, previously deleted entries may be safely ignored (skipped) when forming a new component, effectively removing them from the index.

The entries in an LSM-tree are scattered throughout its sequence of components, which requires range scans to involve all of the components. As entries are fetched from the components, the reconciliation process 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 $\langle key, \textit{component number} \rangle$, where each sub-cursor operates on a single component. This design temporally groups entries with identical keys, enabling merge-based 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"), consequently checking the primary key uniqueness is not required due to the upsert semantic. 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 [61], a Bloom filter can be maintained for each disk component to reduce the chance of performing unnecessary searches during point lookups, thereby decreasing the I/O cost of performing the integrity check and point lookups in general.

Compared to a B-tree, the LSM-tree offers superior write throughput at some potential expense in reads and scans [41, 53]. As the number of disk components increases, search performance degrades since more disk components must be accessed for reads and scans. It is therefore desirable to keep relatively few disk components by periodically merging multiple disk components into fewer, larger components in order to maintain acceptable search performance. We refer interested readers to [60] for a study of advanced LSM merging strategies.

2.2 Apache AsterixDB

Apache AsterixDB [2, 8] is a parallel, semi-structured information management platform that provides the ability to ingest, store, index, query, and analyze mass quantities of data.

AsterixDB supports use-cases ranging from rigid, relation-like data collections, whose types are predefined and invariant, to more flexible and complex data, where little is known a priori and the data instances are variant and self-describing. For this purpose, AsterixDB has a flexible data model called ADM (Asterix Data Model) that is a superset of JSON and a declarative query language called AQL (Asterix Query Language) comparable to languages such as Pig [52], Hive [11], and Jaql [40]. In addition, a rich set of primitive data types is provided, including support for spatial, temporal, and textual data. Also, it supports partitioned LSM-tree-based data storage and indexing options such as B^+-trees, R-trees, and inverted indexes [9]. Besides, it supports *Data Feeds* [35], a mechanism for having data continuously arrive into AsterixDB from external sources and to have that data incrementally populate datasets[1] and their associated indexes. Together with record-level transactions, feeds support fast ingestion of newly arriving data and make them queryable in a timely manner.

[1] A dataset is analogous to a table in a traditional relational DBMS.

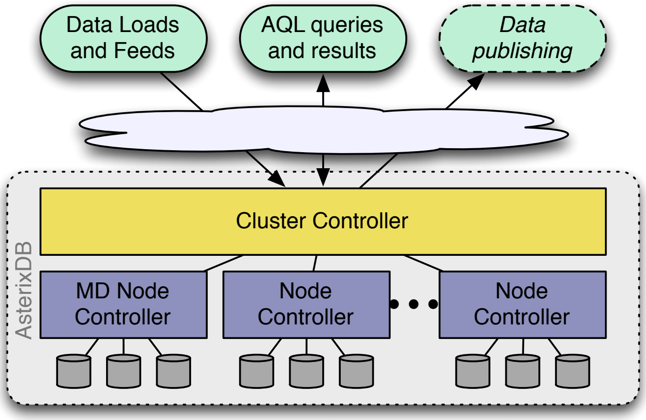


Figure 2.2: Logical deployment of an AsterixDB instance

Furthermore, node failures caused by software and/or hardware problems are remedied by log-based recovery, LSM-tree-based shadowing, and/or log and data replication in multiple nodes.

An AsterixDB instance consists of a single Cluster Controller (CC) and one or more Node Controllers (NC), where one of NCs maintains the metadata of the managed data in the instance. A user request written in AQL or SQL++[2] goes to the CC. The CC compiles the user request and generates an optimized parallel execution plan, consulting with the metadata, and the plan describes a DAG (Directed Acyclic Graph) of physical operators required to execute the request. The plan is then sent to all involved NCs. Each NC generates the DAG of physical operators according to the description in the plan. Finally, the DAG of physical operators starts operating on the data provided from the user request and/or the storage in each node. Figure 2.2 shows a logical deployment of an AsterixDB instance, where the leftmost NC (denoted as the "MD Node Controller") maintains the metadata and the dotted Data publishing path [20] indicates an ongoing research effort to add support for continuous queries and notifications.

A dataset in AsterixDB consists of a primary index plus zero or more secondary indexes,

[2] SQL++ is a semi-structured data model and query language. See [54] for more details

where all records in the primary index are horizontally partitioned by its primary key across all NCs and entries in the secondary indexes are local to the records in the primary index partitions.

The full software stack of AsterixDB consists of three layers from top to bottom: AsterixDB, Algebricks, and Hyracks. The AsterixDB layer implements ADM and AQL, where AQL queries are parsed and analyzed to produce an initial logical query plan. The logical query plan is then fed into Algebricks for rewriting.

Algebricks [17] is a query compiler that separates language-specific and data model-dependent aspects from a more general query compiler backend that can generate executable data-parallel programs for shared-nothing clusters. Given an initial logical plan, Algebricks performs logical rewriting heuristically using logical rewrite rules and then performs physical rewriting, which translates the optimized logical plan to an Algebricks physical plan by selecting physical operators for every logical operation in the plan. Both logical and physical rewritings are rule-based and configured by selecting the set of rules to execute during the optimization phases. Also, Algebricks presents hooks to provide Algebricks with system-specific metadata such as available access methods. The resulting physical plan is a *job* (a DAG of physical operators) specification, which is fed into Hyracks for execution.

Hyracks [18] is a push-based data-parallel runtime execution engine. The Hyracks master node instructs a set of worker nodes to execute clones of an input job in parallel and orchestrates the job's execution. Each worker node generates the DAG of physical operators and executes the operators in parallel while adhering to the blocking requirements of the operators based on data dependencies. Hyracks implements its key building blocks as libraries such as its operator library, connector library, storage library, and HDFS utilities. The storage library includes a number of native storage and indexing mechanisms, such as B^+-Tree, R-Tree, inverted index, and their LSM-based counterparts.

2.3 LSM Storage Layer in AsterixDB

The LSM storage layer is a key foundation throughout our thesis study. In this section, we explain the LSM storage layer in AsterixDB in more detail.

AsterixDB's storage layer [9] provides a general framework for converting a class of indexes (including conventional B-trees, R-trees, and inverted indexes) with certain basic operations such as insert, delete, and bulkload into LSM-based indexes. The framework behaves as 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, building specialized index structures from scratch can be avoided while enabling the advantages provided by an LSM-based index organization.

Figure 2.3 shows an example of LSM index operations. The timeline in the figure shows a sequence of operations. In Chapter 2.3.5, we will describe each operation's effect in detail including the initial empty state of each LSM index components and the resulting state after all of the operations are performed. Before doing so, we first explain the logic of each LSM index operations.

2.3.1 Primary LSM B-tree

As shown in Figure 2.3, the in-memory component of the primary LSM B-tree consists of a single B-tree, which stores entries of $\langle pk, record \rangle$ pairs and orders them by the primary key (denoted as "pk"). Similarly, a disk component consists of a B-tree with an associated Bloom filter. The filter is used to avoid unnecessary disk I/Os when checking primary-key uniqueness during insertions.

An insert operation inserts an entry $e = \langle pk, record \rangle$ into the in-memory component if

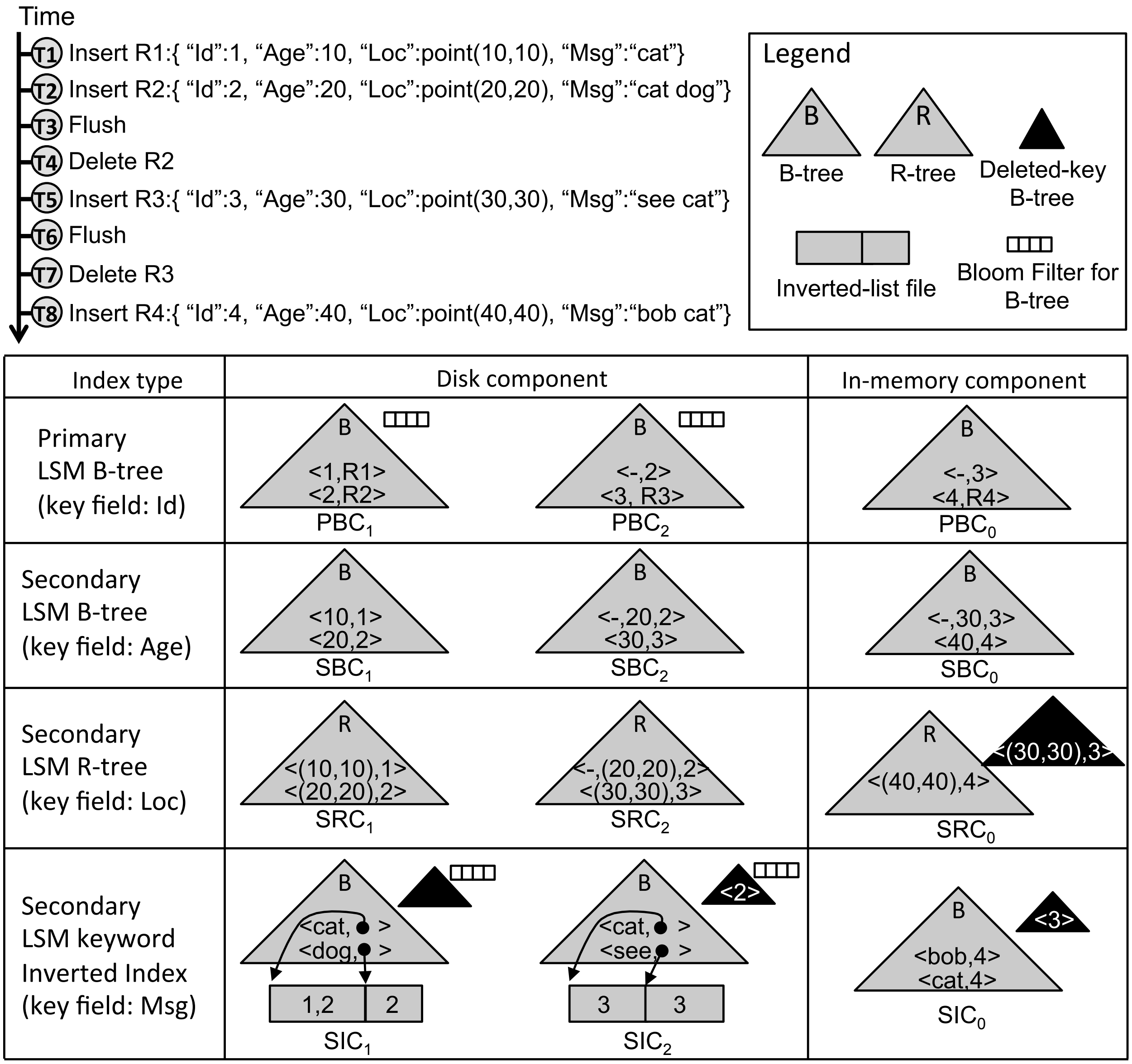

Time
T1 Insert R1:{ “Id”:1, “Age”:10, “Loc”:point(10,10), “Msg”:“cat”}
T2 Insert R2:{ “Id”:2, “Age”:20, “Loc”:point(20,20), “Msg”:“cat dog”}
T3 Flush
T4 Delete R2
T5 Insert R3:{ “Id”:3, “Age”:30, “Loc”:point(30,30), “Msg”:“see cat”}
T6 Flush
T7 Delete R3
T8 Insert R4:{ “Id”:4, “Age”:40, “Loc”:point(40,40), “Msg”:“bob cat”}
Legend
B
B-tree
R
R-tree
Deleted-key B-tree
Inverted-list file
Bloom Filter for B-tree
Index type
Disk component
In-memory component
Primary LSM B-tree (key field: Id)
<1,R1> <2,R2>
PBC1
<-,2> <3, R3>
PBC2
<-,3> <4,R4>
PBC0
Secondary LSM B-tree (key field: Age)
<10,1> <20,2>
SBC1
<-,20,2> <30,3>
SBC2
<-,30,3> <40,4>
SBC0
Secondary LSM R-tree (key field: Loc)
<(10,10),1> <(20,20),2>
SRC1
<-,(20,20),2> <(30,30),3>
SRC2
<(40,40),4>
<(30,30),3>
SRC0
Secondary LSM keyword Inverted Index (key field: Msg)
<cat, > <dog, >
1,2
2
SIC1
<2>
<cat, > <see >
3
3
SIC2
<bob,4> <cat,4>
<3>
SIC0

Figure 2.3: An example of LSM index operations

a corresponding anti-matter entry $e' = \langle -, pk \rangle$ does not already exist in the component. Otherwise, e' is replaced with e.

A delete operation inserts an anti-matter entry, $e' = \langle -, pk \rangle$ into the in-memory component if the entry e does not exist in the in-memory component. Otherwise, e is replaced with e'. A flush operation creates a new disk component with a Bloom filter by bulk-loading an on-disk B-tree instance with all of the entries in the in-memory component.

A search operation creates a range-scan cursor with a given predicate. The cursor consists of a heap of sub-cursors on the components per the design described in Section 2.1. The heap facilitates the reconciliation of multiple versions of an entry identified by the same primary key by making them come out together from the heap. Thus, the range cursor can return only the latest version of the entry from the heap if the latest version is not anti-matter. Otherwise, it is ignored.

A component-merge operation retrieves all reconciled entries from the disk components, merged by the search operation, and creates a new disk component by bulk-loading those entries. The merged disk components are then removed once there are no longer any ongoing readers using them.

2.3.2 Secondary LSM B-tree

The differences between the primary LSM B-tree and a secondary LSM B-tree are 1) a secondary disk component has no Bloom filter, 2) an entry e in the secondary LSM B-tree is $\langle sk, pk \rangle$ and the corresponding anti-matter entry e' is $\langle -, sk, pk \rangle$, where sk represents a secondary key, and 3) an insert operation in the secondary LSM B-tree does not check for key uniqueness. Except for these differences, the secondary index works in exactly the same manner as the primary one. It is worth mentioning that since Bloom filter supports only key lookups, it is only used for the primary-key-lookup purpose in LSM indexes in AsterixDB.

2.3.3 Secondary LSM R-tree

An in-memory component of a secondary LSM R-tree consists of a traditional R-tree along with a deleted-key B-tree that captures deleted entries. The deleted-key B-tree is useful when the merge-based reconciliation (described in Section 2.1) is not available due to the lack of total order among entries in the in-memory component [9]. An LSM R-tree's disk component is simply a variant of an R-tree, where it orders indexed entries using a Hilbert curve when loading the tree.

An insert operation for a secondary LSM R-tree works in a way similar to a secondary LSM B-tree except that it follows the traditional R-tree insert logic.

A delete operation removes an entry $e = \langle sk, pk \rangle$ from the in-memory R-tree component if e exists. Then, instead of inserting an anti-matter entry into the R-tree, it inserts e into the deleted-key B-tree. This e entry in the deleted-key B-tree plays a role of a sentinel by preventing any entries of e coming out of an older component's R-tree from being returned.

A flush operation creates a new disk component as follows. The entries in the in-memory component R-tree are sorted using a Hilbert curve. Also, the entries in the associated deleted-key B-tree are sorted using the same Hilbert curve. During both sortings, entries are compared relatively based on the Hilbert curve. Then, a new disk component is created by merging and bulk-loading the two sets of sorted entries. During this process, an entry e from the deleted-key B-tree becomes an anti-matter entry e' in the new disk component if an identical entry e does not appear in the sorted R-tree list. (Otherwise, e from the deleted-key B-tree is ignored.) Note that an entry e can appear both in the R-tree and the deleted-key B-tree of the same in-memory component if the deletion of e is followed by the insertion of an identical entry e in the same in-memory component.

A search operation creates a range-scan cursor consisting of a heap of sub-cursors only on disk components. It gets its reconciled entries in a similar manner as the LSM B-tree search operation except that the reconciled entries from the heap must be checked against the in-memory component's deleted-key B-tree, and only the surviving entries are returned from the range cursor at the end.

A merge operation gets its reconciled entries just as the search operation does, but does so without checking the in-memory component's deleted-key B-tree since a merge operation only merges disk components.

2.3.4 Secondary LSM Inverted Index

An in-memory component of a secondary LSM inverted index consists of a B-tree plus a deleted-key B-tree. The B-tree stores $\langle token, pk \rangle$ entries ordered by the token and the primary key, where multiple $\langle token, pk \rangle$ entries can be created from a given secondary indexed field value. For example, as shown in Figure 2.3, when the "see cat" value of the record R3 is indexed in an keyword inverted index, ⟨ "see", 3 ⟩ and ⟨ "cat", 3 ⟩ entries are created and indexed, where "see" and "cat" are the keyword tokens and 3 is the primary key. The deleted-key B-tree captures deleted entries by recording their primary keys. A disk component consists of a B-tree, a deleted-key B-tree, a Bloom filter on the latter, and an inverted-list file. The disk component B-tree stores ⟨*token*, *inverted-list pointer*⟩ entries ordered by tokens, where each inverted-list is a list of primary keys whose records include the associated token in their secondary indexed field values. The inverted-list file stores multiple such inverted-lists. The disk component's deleted-key B-tree behaves in the same way as for the in-memory component, but it additionally has a Bloom filter to avoid unnecessary disk I/Os.

Given a secondary index field value, an insert operation adds one or more $\langle token, pk \rangle$ entries into an in-memory component, where the logic of inserting an entry is the same as for the secondary LSM B-tree.

A delete operation removes one or more ⟨*token*, *pk*⟩ entries from the in-memory B-tree component, if any, and then inserts a *representative* entry, ⟨*pk*⟩, into the deleted-key B-tree. If disk components were to have antimatter tuples instead of having a deleted-key B-tree like the LSM R-tree does, the deleted-key B-tree would need to have all ⟨*token*, *pk*⟩ entries–which could be many–versus having a single $\langle pk \rangle$ entry. This is why AsterixDB uses the deleted-key B-tree instead of having antimatter tuples in disk components of inverted indexes.

A flush operation creates a new disk component as follows. While reading all the entries in the in-memory component, it creates a set of ⟨*token*, *a pointer of the pk list*⟩ pairs and the *pk lists* on the fly, and the pairs are inserted into the disk component B-tree and the pk lists are inserted into the inverted-list file.

A search operation on an LSM inverted index creates a range cursor consisting of sub-cursors on all the components, and processes each sub-cursor one at a time. Each entry coming out from a sub-cursor must probe the deleted-key B-trees of all newer components in order not to return deleted entries.

A merge operation obtains its correctly reconciled entries in the same way that search operations do, and it uses them to create a new disk component by merging and bulk-loading the entries into the new component.

2.3.5 An Example of LSM Index Operations

Figure 2.3 shows how all these indexes work with a small example scenario that we can now discuss in greater detail. The timeline in the figure shows a sequence of operations. In this example, there is a dataset with records consisting of four fields: Id (an integer key for the primary LSM B-tree), Age (an integer key for the secondary LSM B-tree), Loc (a two-dimensional point key for the secondary LSM R-tree), and Msg (a string field on which an LSM inverted index is created). The components of the primary LSM B-tree, the secondary LSM B-tree, the secondary LSM R-tree, and the secondary LSM inverted index are denoted as PBC_i, SBC_i, SRC_i, and SIC_i, respectively, where the subscript number of a disk component in the figure represents a component-creation timestamp. Older disk components have smaller values of i, and in-memory components have the special subscript 0. For expository purposes, we assume that after every two entries are inserted into the primary LSM B-tree's in-memory component, all indexes' in-memory components are flushed and new disk components are created.

The initial state of the indexes is empty. After two records R1 and R2 are inserted, the state of each index is as shown in Figure 2.4. Each index's in-memory component contains its corresponding entries from the associated field values in the records. For example, the primary index's in-memory component, PBC_0, contains two entries, where each entry consists of a $\langle pk, record \rangle$ pair. The secondary LSM B-tree's in-memory component, SBC_0, also contains two entries, where each entry consists of an $\langle sk, pk \rangle$ pair, and similarly for the R-tree. However, the inverted index's SIC_0 contains three entries, each consisting of a $\langle token, pk \rangle$ pair, since R2's Msg field value generates two keyword tokens,"cat" and "dog". Also, since there are no deleted records yet, the deleted-key B-trees (in black) are empty.

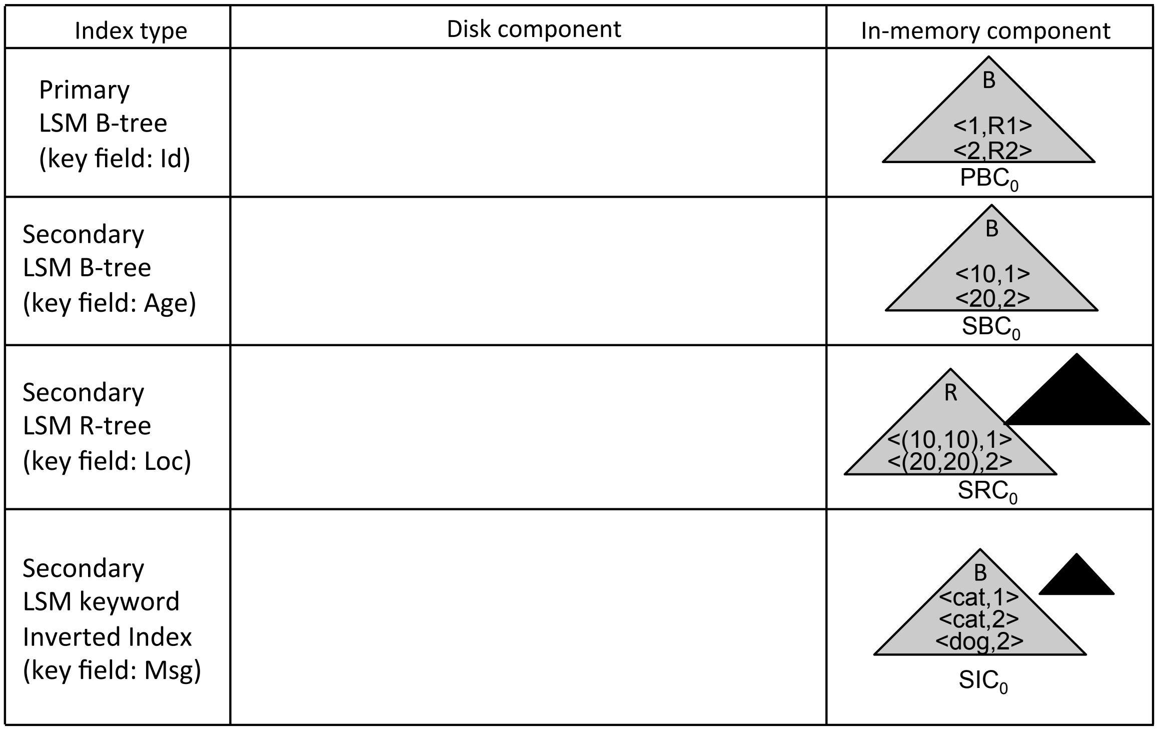

Index type
Disk component
In-memory component
Primary LSM B-tree (key field: Id)
B
<1,R1>
<2,R2>
PBC0
Secondary LSM B-tree (key field: Age)
B
<10,1>
<20,2>
SBC0
Secondary LSM R-tree (key field: Loc)
R
<(10,10),1>
<(20,20),2>
SRC0
Secondary LSM keyword Inverted Index (key field: Msg)
B
<cat,1>
<cat,2>
<dog,2>
SIC0

Figure 2.4: Indexes after inserting two records, R1 and R2

After the first two records are inserted, suppose those in-memory components are flushed. After the first flush operation for each in-memory component, the state of each index is shown in Figure 2.5. Note that when the entries in the inverted index's in-memory component are flushed and a new disk component is created, ⟨*token*, *pointer to a pk list*⟩ pairs are put into the corresponding B-tree in SIC_1 and the *pk lists* (i.e., the inverted-lists) are put into the inverted-list file in SIC_1. Also, Bloom filters are created for PBC_1 and for SIC_1's deleted-key B-tree.

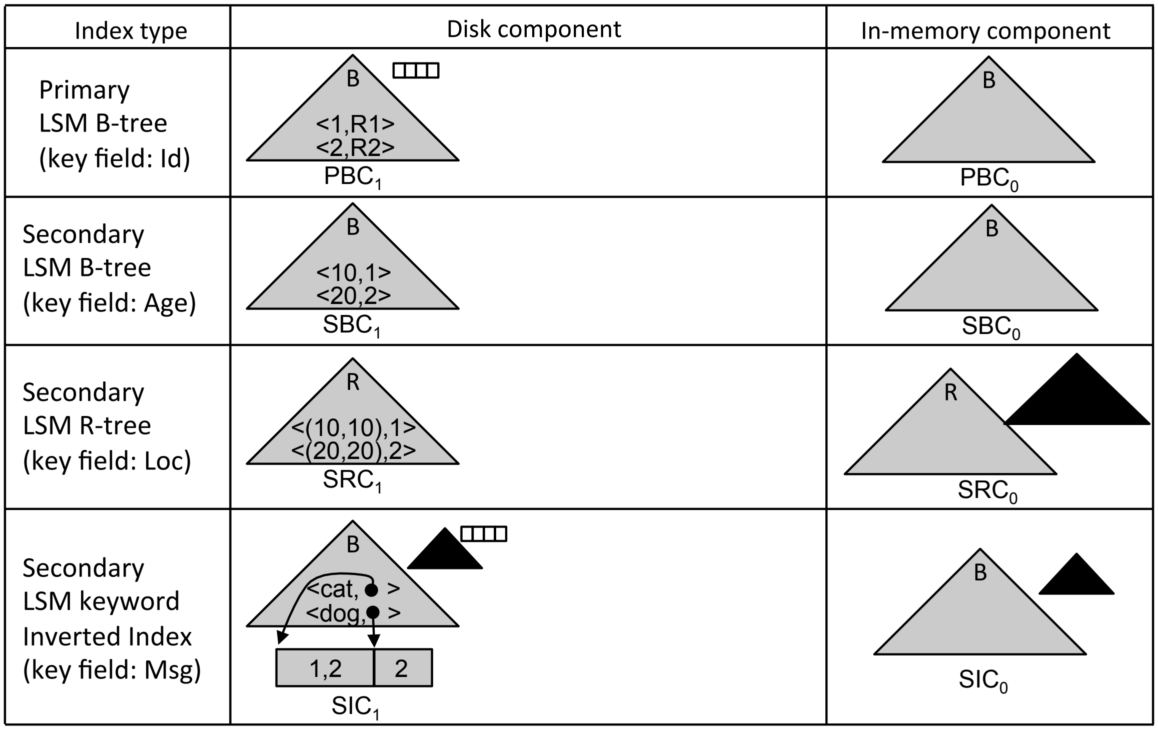

Index type
Disk component
In-memory component
Primary LSM B-tree (key field: Id)
B
<1,R1>
<2,R2>
PBC1
B
PBC0
Secondary LSM B-tree (key field: Age)
B
<10,1>
<20,2>
SBC1
B
SBC0
Secondary LSM R-tree (key field: Loc)
R
<(10,10),1>
<(20,20),2>
SRC1
R
SRC0
Secondary LSM keyword Inverted Index (key field: Msg)
B
<cat, ● >
<dog, ● >
1,2
2
SIC1
B
SIC0

Figure 2.5: Indexes after the first flush of the in-memory components

Suppose now that record R2 is deleted and record R3 is inserted. After these two operations, the state of each index is shown in Figure 2.6. As a result of deleting record R2, PBC_0 contains the anti-matter tuple ⟨-,2⟩. Similarly, SBC_0 contains the anti-matter tuple ⟨-,20,2⟩. In contrast, SRC_0 and SIC_0 record their deleted entries by making entries in their associated deleted-key B-trees.

After that, suppose those new in-memory components are flushed again. After the second

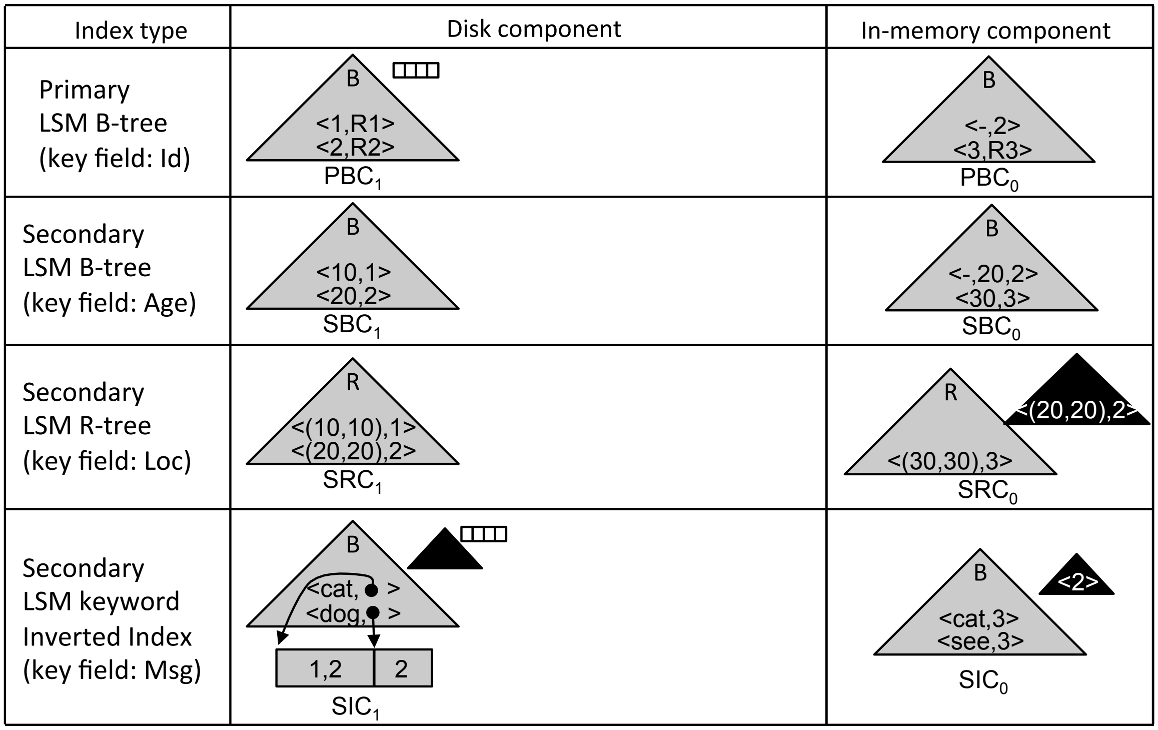


Figure 2.6: Indexes after deleting R2 and inserting R3

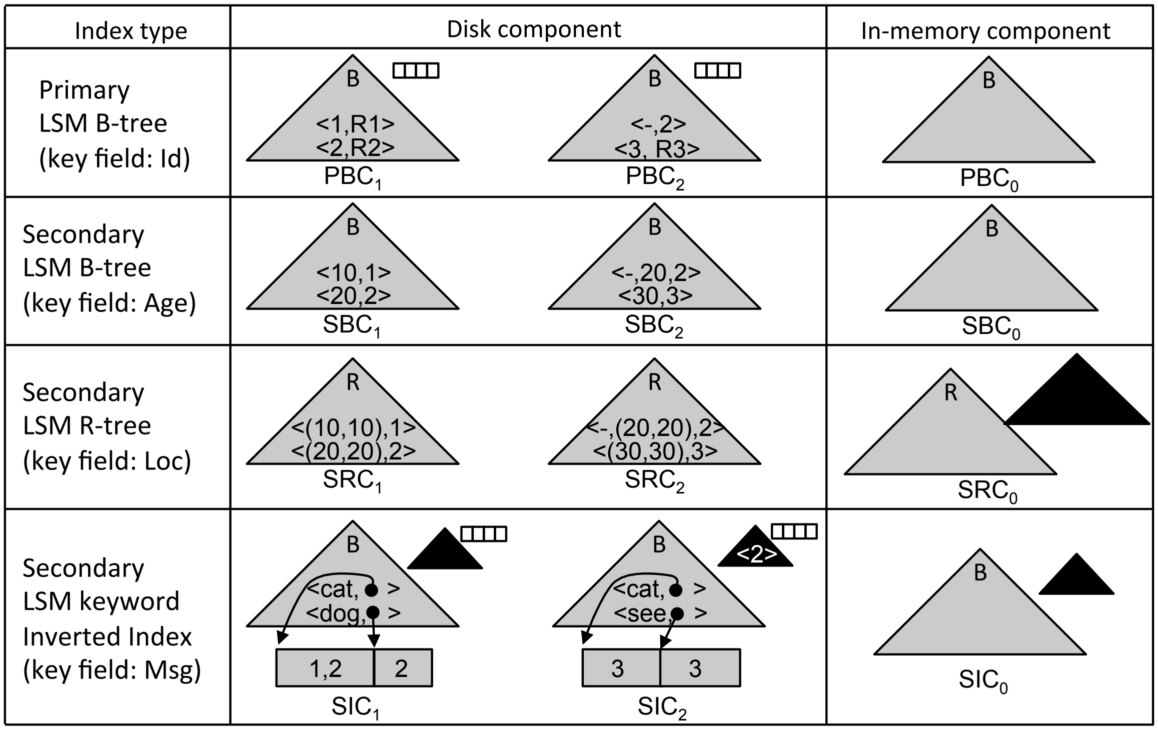


Figure 2.7: Indexes after the second flush of the in-memory components

flush operation for each in-memory component, the state of each index is shown in Figure 2.7. Note that the deleted entry ⟨(20,20),2⟩ recorded in SRC_0's deleted-key B-tree now becomes an anti-matter tuple ⟨-,(20,20),2⟩ in the newly created disk component, SRC_2. In constrast, the inverted index's deleted entry ⟨2⟩ is still recorded using a deleted-key B-tree in the newly created disk component, SIC_2.

Lastly, after record R3 is deleted and record R4 is inserted, the state of each index (before the next flush) is depicted in Figure 2.3.

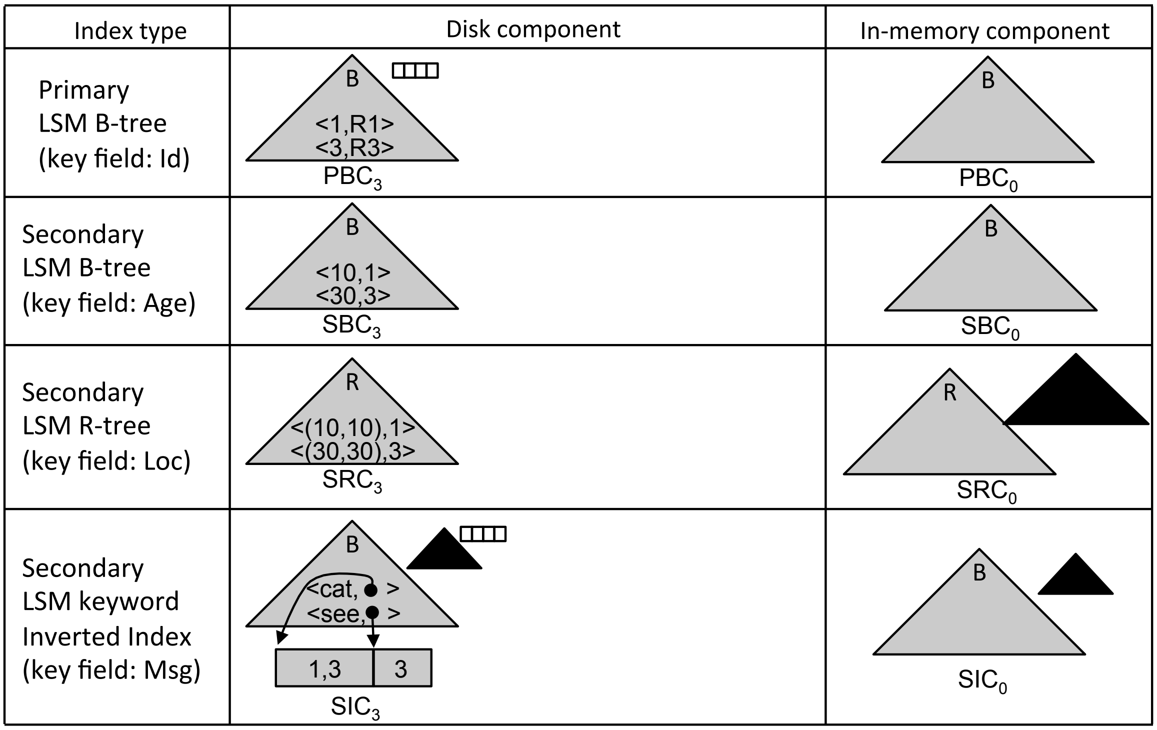

Index type
Disk component
In-memory component
Primary LSM B-tree (key field: Id)
B
<1,R1>
<3,R3>
PBC3
B
PBC0
Secondary LSM B-tree (key field: Age)
B
<10,1>
<30,3>
SBC3
B
SBC0
Secondary LSM R-tree (key field: Loc)
R
<(10,10),1>
<(30,30),3>
SRC3
R
SRC0
Secondary LSM keyword Inverted Index (key field: Msg)
B
<cat, ● >
<see, ● >
1,3
3
SIC3
B
SIC0

Figure 2.8: Indexes after the merge operation

We conclude this section by explaining a merge operation example. Suppose that the merge policy in the example scenario triggers a merge operation whenever there are two disk components. This means that when the second disk components of the indexes were created by the second flush operation (which was triggered by inserting record R3 at time T5), a merge operation will be triggered. This merge operation will merge the two disk components of each index into a new disk component for each index. The resulting indexes are shown in Figure 2.8. During the merge operation, all anti-matter tuples or deleted entries will be reconciled with their corresponding matter entries. Consequently, all of the entries related to the deleted records will now be physically removed from the resulting indexes.

Chapter 3

Record-Level Transactions in AsterixDB

3.1 Introduction

In order to store and index big and fast data generated from mobile devices and IoT (Internet-of-Things) sensors, we designed and implemented a Log-Structured Merge-tree-based (LSM) storage layer due to its superior insert performance as compared to traditional in-place-update storage structures. However, in order to support a full-fledged data-processing system, it was critical to provide certain transactional guarantees without degrading the systems's performance.

A transaction in a database management system (DBMS) is a sequence of operations that read and write shared data as a single logical unit of work that transforms one state of the database into another state. A transaction has four properties: atomicity, consistency, isolation, and durability (ACID) [15, 34].

- Atomicity: A transaction must be an atomic unit of work; either all of its data modifications are performed or none of them are performed.

- Consistency: A transaction is a correct transformation of the state. The resulting state of the actions taken as a group does not violate any of the integrity constraints associated with the state. (This property requires that the transaction itself must be a correct program.)

- Isolation: Even though multiple transactions can execute concurrently, it appears to each transaction, T, that each of the other transactions executed either before T or after T, but not both.

- Durability: Once a transaction completes successfully, its changes to the database state will survive subsequent system failures.

Let us take a look at a simple example that reveals potential issues related to transactions when big and fast data is stored and indexed into a traditional DBMS and the data in the DBMS is concurrently queried.

Suppose that data producers provide records into a DBMS and that data readers consume the data from the DBMS. The data producers may provide 1 record at a time, 1,000 records at a time, 1 million records at a time, or even 1 billion records at a time, based on the producers' needs or constraints (e.g., power consumption). Further, suppose that the readers want to consume the data as soon as possible when there is newly produced data, regardless of the number of records inserted in a batch.

There are several issues that can prevent the producers and readers from storing and reading the data in a timely manner. The first issue is that the readers are traditionally not allowed to read uncommitted records. Thus, if all records in a batch belong to a single transaction, the readers must wait until all those records are inserted and committed. As the number of records in a batch increases, the readers will need to wait longer. The second issue is that when the readers send a query to the DBMS, the DBMS traditionally allows the readers to read committed data as of the time that the query started in order to guarantee the isolation property. This means that if new records are being inserted and committed between the time when the query starts and the time when it ends, the query may not be permitted to read those new records inserted and committed during the query is processed. Moreover, if there are subsequent queries belonging to the same transaction as the previous query, those queries may not be permitted to read those new records, either.

DBMSs supporting concurrent transactions provide different *isolation levels* [13, 6] to allow programmers to trade off transactional guarantees for a potential performance gain [33, 37, 14]. ANSI SQL-92 [10] defines isolation levels in terms of the three phenomena known as dirty reads, non-repeatable reads, and phantom reads.

- Dirty reads: A dirty read occurs when a transaction reads data that has not yet been committed. For example, suppose transaction 1 updates a row. Transaction 2 reads the updated row before transaction 1 commits the update. If transaction 1 rolls back the change, transaction 2 will have read data that is considered to have never existed.
- Non-repeatable reads: A non-repeatable read occurs when a transaction reads the same row twice but gets different values. For example, suppose transaction 1 reads a row. Transaction 2 updates that row and commits the update. If transaction 1 rereads the row, it retrieves different row values.
- Phantom reads: A phantom is a row that matches the search criteria but is not initially seen. For example, suppose transaction 1 reads a set of rows that satisfy some search criteria. Transaction 2 generates a new row (through either an update or an insert) that matches the search criteria for transaction 1. If transaction 1 reexecutes the statement that reads the rows, it gets a different set of rows.

Isolation Level	**Dirty read**	**Non-repeatable read**	**Phantom read**
Read uncommitted	Possible	Possible	Possible
Read committed	Not Possible	Possible	Possible
Repeatable read	Not Possible	Not Possible	Possible
Serializable	Not Possible	Not Possible	Not Possible

Table 3.1: Isolation levels

Table 3.1 shows the isolation levels defined by the phenomena. Transactions running at the *serializable* isolation level are guaranteed that the result of executing the transactions is equivalent to some serial execution of the transactions, which prevents the three phenomena. Due to the coordination overhead and interference among such concurrently executed transactions when using serializability to avoid the phenomena, the serializable isolation level is often replaced in pratical applications with a lesser isolation level, such as repeatable read, read committed, or even read uncommitted, as long as the offered guarantees are good enough for the application's purpose. It is interesting to note that recently proliferating NoSQL systems have a tendency to provide very primitive transactional guarantees by supporting only small atomic operations such as reads and writes for a single ⟨key, value⟩ pair as a transaction.

AsterixDB has chosen to support *record-level* transactions that begin and terminate implicitly for each record inserted, deleted, or searched while a given DML statement is being executed. This fine granularity (i.e., record) of the AsterixDB transaction model enables newly added records to be committed as soon as possible regardless of the number of records inserted in a "batch". Also, based on the *read-committed* isolation level, AsterixDB allows queries to read new records that are inserted even after multi-record queries are started.

Considering that major traditional SQL systems provide options for users to pick an appropriate isolation level for their applications, no single isolation level may fit all applications. We believe that the design for record-level transactions based on the read-committed isolation level in AsterixDB is an appropriate way to provide transactional guarantees to applications that must deal with big and fast data with an acceptable performance so that the applications can write and read such data in timely manner. In this chapter, we explain the details of the record-level transactions in AsterixDB.

The rest of this chapter is organized as follows. First, we explain the concept of record-level transactions in more detail. Then, we describe how logging and recovery in AsterixDB work, followed by our concurrency control scheme. Ater that, we explain the detailed storage and execution-level workflow of transactional operations with examples. Finally, we discuss related work and provide conclusions.

3.2 Transaction Model

AsterixDB supports local, record-level transactions across a primary and any number of secondary LSM indexes in a dataset based on the read-committed isolation level.

Record-level transactions begin and terminate implicitly for each record inserted, deleted, or searched while a given AQL statement is being executed. This is similar to the level of transaction support found in today's NoSQL stores. However, 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. This means that due to the atomicity and durability of the record-level transactions, the effect of an inserted record, for example, happens in "all or nothing" manner through the primary index and all secondary indexes, if any, even if there are system failures. The details of the recovery from system failures are explained in Section 3.3.

Record-level transactions are not distributed transactions that span multiple nodes or multiple storage partitions in a node. This is because a dataset in AsterixDB consists of a primary and any number of secondary LSM indexes, all records in the primary index are horizontally partitioned by their primary keys, and the entries in the secondary indexes are local to the records in the primary index partitions, as described in Section 2.2.

AsterixDB does not support multi-statement transactions. In fact, a single AQL statement that involves multiple records can itself involve multiple independent record-level transactions. A consequence of this design is that, when an AQL 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 AQL insert statement and the application logic above will need to take the appropriate action(s) to assess the resulting state and to clean up and/or continue as appropriate.

Lastly, due to the read-committed isolation level, queries may see undesired situations resulting from the allowance of non-repeatable reads and potential phantom reads. More details of AsterixDB's concurrency control are discussed in Section 3.4.

3.3 Logging and Recovery

In this section, we explain that how the atomicity and durability of record-level transactions are guaranteed in AsterixDB.

In general, records stored and indexed in a DBMS are usually organized in pages. A page is an I/O unit to be efficiently written and read to/from underlying nonvolatile storage. When there are no concurrent transactions running, the atomicity and the durability of a transaction in the DBMS are guaranteed essentially by keeping the before and after state of the records affected by the transaction's operations and applying the records' state selectively according to the transaction's final state, either committed or not. More specifically, the insert/delete/update operations in a transaction first capture the before and after images of the affected records in log records written to log pages. Only then are the affected records in the original pages updated. When a transaction is committed, a commit log record for the transaction is generated and all log records and the commit log record for the transaction are made durable (in that order). The transaction is then considered as committed. Since all the log records are made durable when a transaction is committed, even if the system has a crash, the result of the committed transaction's operations can be recaptured by applying the effect of the operations captured in the log records. Also, when a transaction needs to be aborted, the effect of its operations can be canceled using the before-state of the affected records in the log records. This is how a typical log-based recovery scheme such as ARIES [51] works to guarantee the atomicity and durability of a transaction. For a shadowing scheme, copies of the original pages including records to be affected by a transaction are created, and the copied pages are then updated. When the transaction commits, the original pages are invalidated and the copied pages become authoritative. See [49] for more details of the shadow-paging approach.

When concurrent transactions are allowed, the recovery scheme depends greatly on the concurrency-control scheme being used, which in general needs to make sure that a record modified by one transaction, T1, should not be modified by another transaction, T2, until T1 is committed or aborted. Otherwise, the record can end up in an *unrecoverable* state, for example, when T1 aborts after T2 committed.

The rest of the section describes the details of logging and recovery in AsterixDB.

3.3.1 No-Steal Policy

Buffer-management-policy choices such as *steal*, *no-steal*, *force*, and *no-force* in a DBMS are subtle details that are important to the implementation of a recovery scheme [36]. If a page modified by a transaction is allowed to be written to the permanent database on nonvolatile storage before the transaction commits, then the steal policy is said to be followed by the buffer manager. Otherwise, a no-steal policy is said to be in effect. Steal implies that during normal or restart rollback, some undo work might have to be performed on the nonvolatile storage version of the database. If a transaction is not allowed to commit until all of the pages that it has modified are written to the permanent version of the database, then a force policy is said to be in effect. Otherwise, a no-force policy is said to be in effect. With a force policy, during restart recovery, no redo work will be necessary for committed transactions. However, during normal processing (not during crash recovery), the steal policy provides more flexibility in managing limited buffer resources and the no-force policy incurs less overhead during commit time.

AsterixDB supports a no-steal and no-force buffer management policy. With the no-steal policy, we trade off the flexibility of buffer management for the simplified recovery behavior which avoids undo work. However, due to the short life-cycles of record-level transactions in AsterixDB, the lost flexibility is not as significant as it would be in a traditional transaction processing system.

As described in Section 2.3, AsterixDB has a wholly-adopted LSM-based approach in its index storage layer. As updates fill the in-memory components of LSM indexes, memory pressure grows. This pressure must eventually be released by flushing an in-memory component to disk. In order to maintain the no-steal policy, AsterixDB prevents the flushing of an in-memory component until all uncommitted transactions that have modified the component are completed. Further, the system prevents new transactions from entering an in-memory component that is "full" until it has been flushed, reset, and is able to accommodate them. This is done by maintaining a reference count on each in-memory component that is incremented and decremented when a transaction enters and exits a component, respectively. A transaction enters a component before it performs an operation on the component. Then, when the transaction is committed, the transaction exits the component. A reference count of zero implies that there are no active transactions in the component, hence it is safe to flush. As an optimization, AsterixDB maintains two reference counts: one for write transactions and one for read transactions. Read transactions are always permitted to enter an in-memory component, regardless of whether the component is full or not (except for a very brief duration when the component is reset). Write transactions, however, continue to follow the stricter rules that were described when using a single reference count.

3.3.2 Index-Level Logical Logging

Updating a record may involve changes to multiple data pages, e.g., a page of a base table (i.e., a primary index in AsterixDB) that includes the record as well as the associated secondary index pages. These changes to data pages generate log record(s) for the atomicity and durability of the transaction. There are three types of logging: physical logging, logical logging, and physiological logging [34]. Physical logging stores the affected data pages' changes into physical log records, where a physical log record includes a page id and the before and after images of the changes in the page. Multiple physical log records could be generated from a single update operation that affects multiple pages. In contrast, logical logging stores an operation type, the key of the record affected, and the record itself into a logical log record. Redo and undo can be done by replaying the operation stored in the log record or performing the inverse operation of the operation, respectively. Thus, logical logging is better than the physical logging approach since the size of the log records generated from a single update operation is much smaller.

Despite its space advantage, the logical logging approach has a fundamental issue, i.e., an *action-consistent state* of the persistent database must be ensured after the system starts from a crash and before recovery based on the logical log records begins [34]. The action-consistent state of the persistent database represents that, even though an update operation incurs updates to multiple pages, the updated pages must be in effect atomically after the system starts from a crash and before recovery begins. Without an action-consistent state, the correctness of the logical undo and redo operations based on the logical log records cannot be guaranteed.

A compromise between logical and physical logging is also possible. This design is called the physical-to-a-page/logical-within-a-page approach, or simply physiological logging. Physiological logging stores a page id, an operation type in the page, and the corresponding record in the page into a physiological log records. Thus, even though it is logical within a page, multiple log records can still be generated from the multiple physical pages updated by a single update operation.

AsterixDB employs *index-level logical logging* based on the the WAL (Write-Ahead-Log) protocol. Index-level logical logging means that a single update operation in an LSM index generates a single logical log record. This also means that inserting a record into a dataset in AsterixDB generates as many log records as the number of primary and secondary indexes on the dataset. This index-level logical logging approach is enabled by the no-steal policy and atomic creation of LSM index disk components, which together ensure action-consistent state in each disk component of an LSM index. More specifically, due to the no-steal policy, when an in-memory component is flushed to disk, it is guaranteed that all committed transactions' operations are in effect in the in-memory component. AsterixDB employs atomic creation of LSM index disk components; when a new disk component is created from either a flush or a merge operation, the disk component is installed with a validity bit at the end of its creation. A disk component with the validity bit set guarantees that all transactions' operations are in effect in the component, and disk components without the validity bit can be simply deleted when the system restarts from the crash. The WAL protocol is maintained by the no-steal policy and by flushing all log records up to the commit record to disk before a transaction is marked as committed.

Lastly, when a log record is created and placed into the log tail (log buffers in main memory) of a log file, the *componentLSN* of the in-memory component is updated with the LSN of the log record. The componentLSN plays a similar role as a pageLSN in ARIES [51].

3.3.3 Checkpointing

In general, in order to reduce the volume of log records to be read during crash recovery and to recycle disk space used for log files, checkpoints are taken. AsterixDB takes checkpoints periodically and asynchronously for those purposes. For a checkpoint, an independent checkpoint thread periodically checks the difference between the system's current *low water mark* (i.e., the LSN where recovery must begin from) and the *nextLSN* (i.e., the LSN of the next log record that will be created). If the difference exceeds a certain threshold, a checkpoint operation is triggered in order to move the current low water mark forward to the next low water mark. The targeted next low water mark is the sum of the current low water mark LSN and the threshold size. However, if the smallest LSN among the componentLSNs of the in-memory components in the system is less than the next water mark LSN, flush operations are scheduled to flush the in-memory components whose componentLSNs are less than the next low water mark. These flush operations are required to make all committed transactions' effects in in-memory components durable before the checkpoint operation discards the corresponding log records. Once this has been ensured, then the checkpoint operation is performed by the checkpoint thread as follows.

First, the checkpoint thread creates a new checkpoint file in a special directory, where the new checkpoint file contains the targeted low water mark LSN, i.e., a new current low water mark. After that, it removes the previous checkpoint file. Lastly, it also removes all log files whose last log record's LSN is less than the targeted low water mark LSN.

3.3.4 Abort Processing

In general, aborting a transaction has to cancel the effect of the transaction. Since AsterixDB does not provide a user-level interface to abort the ongoing transactions created by a job serving an AQL query, an abort operation can only be triggered when an erroneous situation happens, e.g., due to a duplicate key exception thrown from inserting a record with a duplicate primary key. When an abort is triggered while a job is being executed, the ongoing record-level transactions belonging to the job are identified by reading from the first log record to the last log record created from the job; when the encountered transactions do not have corresponding commit log records, undo operations are performed based on each ongoing transactions' logical log records. In this way, the system stops running erroneous jobs as early as possible instead of allowing ongoing transactions (belonging to the same job) to proceed further.

AsterixDB does not maintain a typical transaction-level table with an entry for each active transaction. Instead, it maintains a *transactor* table, which maintains an entry for each *job*, where an entry includes necessary information for a job such as the firstLSN and the lastLSN created from (the transactions belonging to) the job and the job status (running or aborted). Since a job that can face an abort situation may include both committed record-level transactions and non-committed record-level transactions, unlike ARIES, an abort operation in AsterixDB does not read log records from the lastLSN of the transactor backward by following the *previousLSN* of each log record and performing undo operations. Also, unlike ARIES, no additional log records (e.g., CLRs in ARIES) are created during an abort operation since there is no partial effect of an operation in each index, which is due to the provision of an action-consistent state in each index (as described in Section 3.3.2).

3.3.5 Crash Recovery

The crash recovery logic in AsterixDB is greatly simplified by the use of the no-steal policy and the atomic creation of an LSM index's disk components (which is called *disk component shadowing*). Crash recovery consists of two parts. The first part ensures the action-consistent state of each LSM index's disk components. Any disk component with an unset validity bit is considered to be invalid and is removed, thus ensuring that the data in each index is physically consistent. The second part ensures that all committed transactions' effects that had not been made durable when the system crashed are made durable now based on two-phase log-based recovery. The first phase analyzes winner transactions by reading log records from the low water mark (stored in the checkpoint file) to the end of the log file. The second phase selectively replays only the winner transactions' operations. Index operation *idempotence* is guaranteed by comparing the LSNs of log records with an *indexLSN*, an LSN that indicates the last operation that was applied to the index, i.e., the largest componentLSN of all componentLSNs of the index. Only if a log record's LSN is greater than the indexLSN, is the redo operation for the log record performed.

3.4 Concurrency Control

In general, concurrency control ensures the isolation property of transactions and consequently ensures the atomicity of the concurrently running transactions by preventing data objects in the persistent shared database from being modified concurrently by multiple transactions. AsterixDB provides concurrency control based on locking and latch protocols described in ARIES/KVL [50] for the B-tree and GiST [44] for the R-tree. In this section, we describe the details of concurrency control in AsterixDB.

3.4.1 Read-Committed Isolation Level

AsterixDB's concurrency control supports the read-committed isolation level based on locking, where reader transactions and writer transactions have different locking durations[1]. Writer transactions follow the *strict two-phase locking* (2PL) protocol. Thus, all of the exclusive locks acquired by writer transactions are held until the tranactions commit. Reader transactions acquire very short duration locks called *instant* locks, where an instant lock is released as soon as the lock is granted. Instant locks are used to ensure that all data items read by reader transactions are committed, thereby achieving the read-committed isolation level.

Generally, concurrent transactions running at the read-committed isolation level may face the phenomena known as non-repeatable reads and phantom reads. These phenomena may happen when records read by a given transaction are reread by the same transaction during the transaction's lifetime. However, if the records are only read once during the transaction's lifetime, these phenomena are not possible by definition described in Section 3.1. The possibility of the non-repeatable reads and phantom reads for record-level transactions in AsterixDB may seem unclear at first glance. However, even a record-level transaction may see a record more than once. For example, a delete operation with a predicate will first search for the qualified records and then deletes the records; a qualified record is thus read twice, once for the search and once for the delete. Also, during a self-join operation, the records being joined may be read twice. Thus, it is possible to see these phenomena even for record-level transactions. As a consequence of the phenomena, for the delete example, a record that was qualified during the search could be modified and become a non-qualified record or have been deleted by another transaction before the delete is actually performed. Similarly, for the join case, a record the first time could have been modified before the

[1] Since AsterixDB does not support multi-statement transactions and an AQL in AsterixDB is either read-only query or non-read only query, it can be known that whether the transactions created from a query is read-only or not when the transactions begin.

record is read for the second time. More details of these undesired consequences under the read-committed isolation level will be discussed in Section 3.5.

3.4.2 Deadlock-Free Locking Protocol

A deadlock is a cycle of transactions that all are waiting for locks to be released. A necessary condition for a deadlock to happen is a “Hold-and-Wait” situation; in such a cycle, each transaction must wait for a lock while it is holding at least one other lock. For record-level transactions in AsterixDB, a record is the only (one) resource to be locked during a given transaction’s lifetime. Based on the use of record-level transactions, and given that there is no lock upgrade on any resource in any situation in AsterixDB, it is thus possible for transactions to avoid the “Hold-and-Wait” situation. Therefore, a deadlock-free locking protocol is achievable in AsterixDB.

In traditional transaction processing systems, a single transaction can be processed by multiple threads. The threads processing a transaction mark their operations using the transaction id. For example, when a lock is requested by one of its threads, the thread provides the transaction id as well as the resource id to be locked. Based on a waits-for-graph in which a vertex represents a transaction id and a directed edge from a vertex $T1$ to another vertex $T2$ is added if $T1$ is waiting for a resource held by $T2$, a lock manager can detect a deadlock if the waits-for-graph includes a cycle [38].

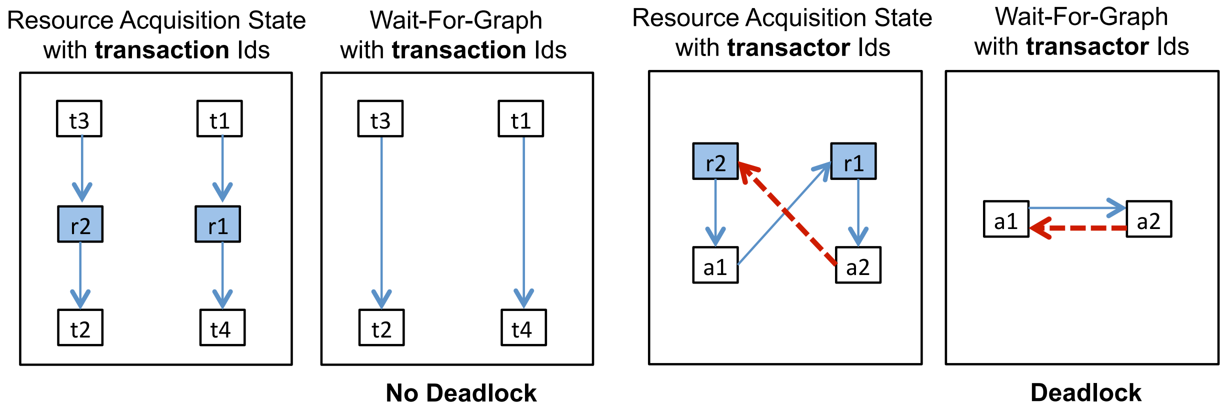
We will use the term *transactor* to denote a thread that works on processing transactions. In a traditional transaction processing system, it is a natural choice to make a transactor serve a single transaction until that transaction is completed (instead of having it serve multiple transactions in an interleaved manner). In contrast, in AsterixDB, it is a more natural choice to have a transactor serve multiple record-level transactions in an interleaved manner due to the following behavior in the AsterixDB parallel runtime architecture. For efficiency, a set of data records to be inserted by a given AQL statement into a primary index and any secondary indexes are conveyed together in a frame that moves from one index operator to the next one. The frame first reaches the primary index operator, where the records, i.e., the $\langle pk$, *data record*$\rangle$ pairs in the frame, are inserted into the primary index. Subsequently, the frame flows to the first secondary index operator and the $\langle sk, pk\rangle$ pairs from the frame are inserted into the first secondary index. Then, it goes to the next secondary index operator, and so on. After the frame passes through all of the index operators, it reaches a commit operator. The commit operator then creates a commit log record for each transaction, i.e., for each data record in the frame. After the commit log records are flushed to disk, locks held by the corresponding transactions are released.

The design choice of having a transactor serve multiple record-level transactions in an interleaved manner leads to an issue in terms of deadlock detection. Consider, for instance, the following case:

- A transactor $a1$ waits for a lock on a record $r1$ for a transaction $t1$ while holding a lock on a record $r2$ for a different record-level transaction $t2$.
- A transactor $a2$ waits for a lock on the record $r2$ for a transaction $t3$ while holding a lock on the record $r1$ for yet another transaction $t4$.

If *transaction* ids are used to construct a waits-for-graph, as shown in Figure 3.1(a), there is no apparent deadlock. However, if *transactor* ids are used in the waits-for-graph, as shown in Figure 3.1(b), a deadlock appears in the graph. This is due to the fact that when a transactor waits, all record-level transactions served by the transactor also effectively wait. Moreover, unless a system can provide as many threads as the number of records being operated on in the system at any point in time, this situation is not easily avoidable.

In order to deal with this issue, we have devised a deadlock-free frame processing and locking


Resource Acquisition State with **transaction** Ids
Wait-For-Graph with **transaction** Ids
t3
t1
r2
r1
t2
t4
t3
t1
t2
t4
No Deadlock
Resource Acquisition State with **transactor** Ids
Wait-For-Graph with **transactor** Ids
r2
r1
a1
a2
a1
a2
Deadlock

(a) In theory

(b) In reality

Figure 3.1: Wait-For-Graph in theory vs. reality

protocol for use in AsterixDB. The intuition is as follows. If a transactor can somehow avoid the “Hold-and-Wait” situation, there can be no deadlock involving that transactor. In order to avoid “Hold-and-Wait”, we need to enforce a protocol in which a transactor may either hold locks without waiting or else wait without holding any locks. Considering the push-based flow of frames of records through a job pipeline described above, such a protocol can be enforced as shown in Algorithm 1. Addressing a snippet of a primary index operator that inserts or deletes a set of entries from a frame that was pushed from the previous operator, this algorithm describes how to make a transactor wait for a lock to be granted without holding any other locks if a lock is not granted immediately upon request.

With this approach, when the primary index operator receives a frame from the previous operator, it tries for each entry in the frame to acquire a lock for the entry using the entry's recordId (lines 1–2). The *tryLock()* function does not wait if the requested lock is not granted immediately. If the try fails, all entries for which locks have already been obtained are prematurely pushed on to the next operator in lines 3–4. (This enables those entries to reach the commit operator, where corresponding commit log records will be created, and where the locks held for the entries will be released once their commits have been flushed to disk.) Then, the primary index operator creates a special *WAIT* log record in line 5 and

Algorithm 1: Deadlock-free frame processing and locking protocol

```
    Input: a frame pushed from the previous operator
 1  for each data record entry in the input frame do
 2      result = tryLock(entry.recordId);
 3      if result == FAILURE then
 4          push all input frame entries for which locks have already been acquired on
            to the next operator;
 5          create a WAIT log record;
 6          wait until the LogFlusher thread has flushed this log record and provided a
            notification;
 7          lock(entry.recordId);
 8      end
 9      create an UPDATE log record;
10      insert (or delete) the data record entry;
11  end
```

waits until the *LogFlusher* thread flushes the log and gives a corresponding notification back in line 6. Receipt of this notification, which will be generated whenever the LogFlusher sees a WAIT log record in the log tail and has flushed it, guarantees that all locks held by this transactor have been released (since log records are written to the log tail in append-only manner) and have been flushed by the LogFlusher thread in a first-in-first-out manner. After that, in line 7, the primary index operator requests the same lock again, but in a blocking manner this time, which makes the operator wait until the lock is granted. This blocking lock request cannot cause a deadlock since the transactor no longer holds any other locks at this moment. Once the lock request is granted either by tryLock() or lock(), an *UPDATE* log record for the current data record is created and the record is inserted (or deleted) in lines 9–10.

Figure 3.2 illustrates the operation of this algorithm using an example. There are two threads, a transactor thread in light gray and a LogFlusher thread in black. The timeline shows the operations (denoted by Ti) performed by the two threads. The Ti in light gray represents the transactor's operation and Ti in black represents the LogFlusher's operation. The order between two of the operations, namely T5 and T6, is not deterministic, but all of

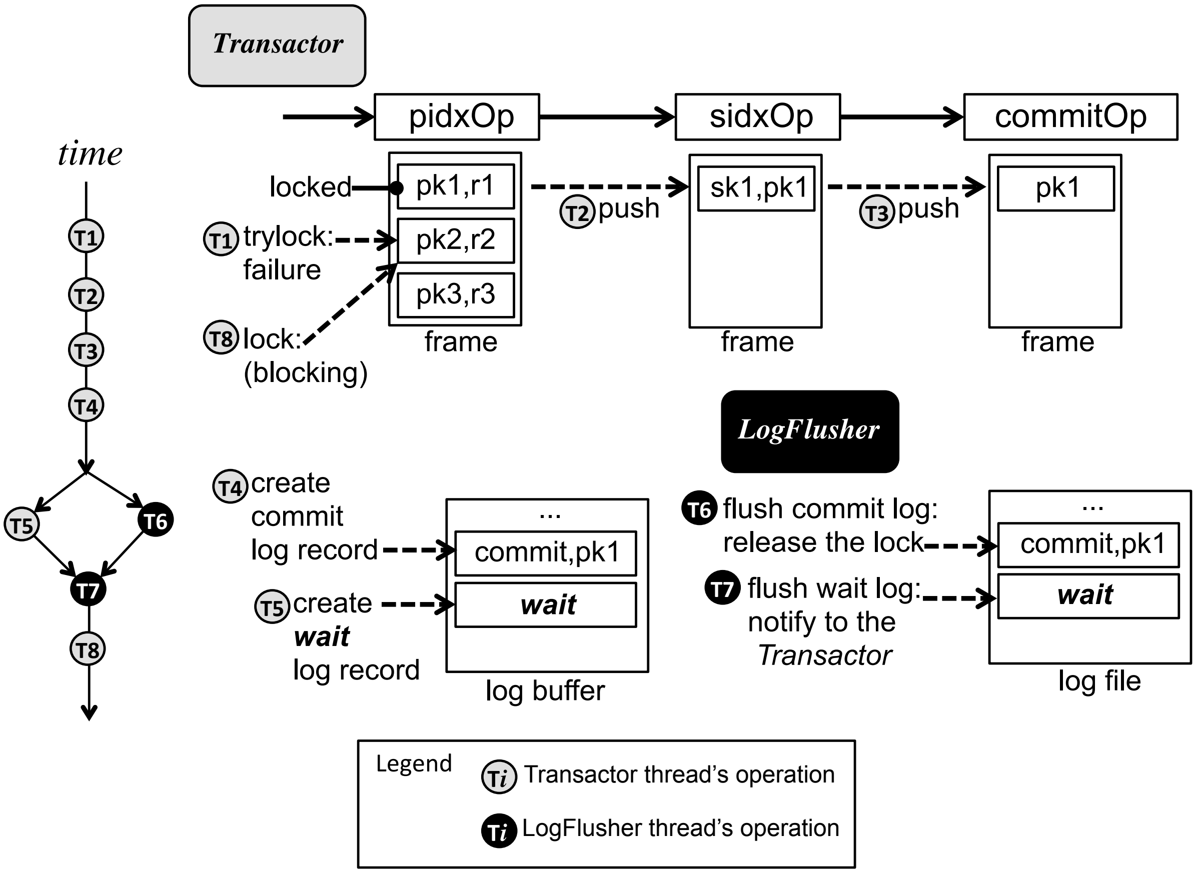


Figure 3.2: An example of deadlock-free frame processing and locking protocol

the other operations are totally ordered.

Suppose that a frame that has reached the primary index operator, denoted as "pidxOp" in the figure, contains three entries. Also, suppose that the entry ⟨pk1,r1⟩ at the top of the frame was locked already, the corresponding UPDATE log record was created, and the entry was already inserted into the primary index. Then, when operation T1 was performed by the transactor to insert the second entry in the frame, the trylock operation failed. Following Algorithm 1, the first entry (which was already locked) will be pushed (T2) to the next secondary index operator, denoted as "sidxOp", and the pushed entry is inserted into the secondary index. After that, the entry is pushed (T3) to the next commit operator, denoted as "commitOp", and the corresponding commit log record is created in the log buffer in memory (T4). Then, the transactor thread creates a corresponding WAIT log record (T5) based on line 5 of Algorithm 1. Concurrently, with this thread now waiting, the LogFlusher thread flushes the commit log record in the log buffer to the log file on disk and releases the corresponding lock for the record (T6). Also, the LogFlusher has flushed the WAIT log record and thus sends a notification to the transactor (T7). Finally, the transactor can safely request the lock on the second entry (T8).

3.4.3 Supporting Index-Only-Scan Queries

In AsterixDB, locks are acquired only when accessing primary indexes. Locks are not acquired when accessing secondary indexes. This primary-index-only locking design facilitates supporting a variety of secondary index structures in terms of concurrency control since index-structure-specific aspects (such as the *revalidation after unconditional locking* described in ARIES/KVL [50]) can be ignored for locking. However, it could lead to potential inconsistency when reading the entries of a secondary index if the corresponding records of the primary index are being altered concurrently. To prevent such inconsistency, the entries retrieved from the secondary index are always validated in AsterixDB by fetching the corresponding records from the primary index after obtaining locks for the records. As a consequence of this validation process, however, primary-index-only locking disables *index-only-scan queries*, where an index-only-scan query can be solely evaluated by searching or scanning a *covering index* that includes all of the fields that appear in a query.

In order to avoid this limitation, we devised a simple but effective method to support index-only-scan queries without losing the secondary-index-structure-agnostic locking behavior. The key idea is as follows. While processing an index-only-scan query, when a qualified entry is obtained from the secondary index, a lock is requested for the entry based on its primary key. If the lock is granted, the entry can be returned without the aforementioned validation step in the primary index. Otherwise, the entry must still go through the final validation step.

For the implementation, when the query optimizer in AsterixDB determines that a given query can be an index-only-scan query, an additional execution path is added right after the secondary index scan operator; the additional path tries to acquire a lock for each qualified entry from the secondary index. If the lock request is granted, the entry is returned without the validation step in the primary index. Otherwise, the entry follows the path for the validation process in the primary index. We will describe the detailed execution plan further with an example in Section 3.5.2.

3.5 Transactional Operation Flow in AsterixDB

In this section, we explain in more detail the flow of transactional operations with example AQL queries that perform inserting, searching, and deleting records. We will show job pipelines (i.e., DAGs of operators) for such queries and explain the process of acquiring and releasing locks as well as the process of creating and flushing log records. Also, we discuss further the consistency-related consequences of running concurrent transactions under the read-committed isolation level.

To begin with, let us define a simple example *dataset* which is used throughout this section; an AsterixDB dataset is analogous to a table in an RDBMS. The corresponding AQL DDL (data definition language) statements are shown in Figure 3.3.

The DDL statements first create a MyMusic *dataverse* (which is analogous to a "database" in an RDBMS) and then proceed to use it. The SongType *datatype* is then created, where a datatype specifies record field names and their types and also can specify optionally whether more undeclared fields may appear (using **as open**) or not (using **as closed**). After that, the Song dataset is created, where a record in the dataset must conform to the SongType, and

```
create dataverse MyMusic;
use dataverse MyMusic;

create type SongType as open {
    id: int64,
    title: string,
    release-year: int32
}

create dataset Song(SongType) primary key id;
create index idxReleaseYear on Song(release-year);
```

Figure 3.3: DDL statements to create an example dataset with a secondary index

the id field is declared to be the primary key field for the dataset. In addition, a secondary B-tree (default) index is created on the release-year field.

Using this example dataverse, the rest of the section provides more detailed view of transaction processing in AsterixDB by focusing on the detailed flow of insert, search, and delete operations for a series of example AQL updates and queries. Recall that when an AQL query is submitted to an AsterixDB instance, a job specification for the query is ultimately generated by the compiler, the corresponding job pipeline is instantiated as a DAG of operators, and then during execution, a group of entries in a frame will be processed by one operator and then passed to the next operator in a push-based manner until all input frames have been consumed by the last operator. Also, for update operations, the entries in the primary index are updated first and then the rest of the secondary indexes, if any, are updated one by one.

3.5.1 Inserts

Figure 3.4 shows an AQL query, AQ1[2], to insert a record. Also, Figure 3.5 shows a typical job pipeline for an insert operation resulting from an AQL query such as AQ1. As shown in

[2] AQ represents AsterixDB Query.

```
insert into dataset Song(
{ "id": 1991, "title": "Jude", "release-year": 2000 }
);
```

Figure 3.4: AQ1 to insert a record

Figure 3.5, in order to insert a record into a dataset, the record is pushed frame by frame from the first operator at the bottom up through the last operator at the top with appropriate fields considering each operator's input entry format. The figure provides a simplified version of the actual job pipeline in order to focus on transaction processing, omitting unnecessary details such as how to create a record from the given user input, how to get a primary key or a secondary key from the record, etc. Also, in the figure, the frames that carry groups of entries are not shown. Instead, entries with appropriate fields are used to denote the data flowing between the operators.

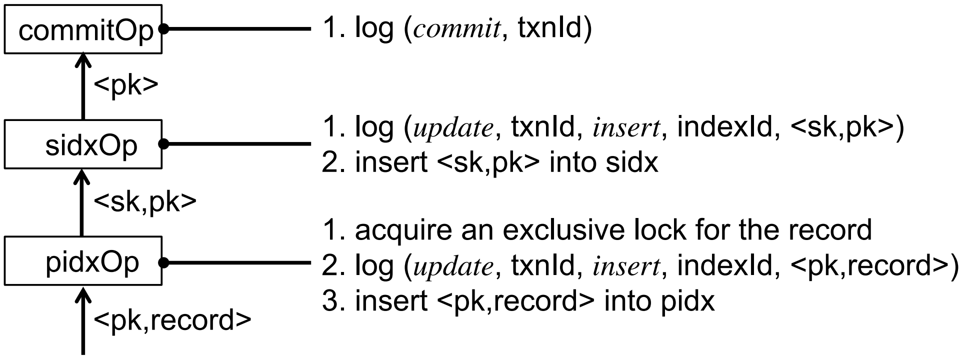

commitOp
1. log (*commit*, txnId)
<pk>
sidxOp
1. log (*update*, txnId, *insert*, indexId, <sk,pk>)
2. insert <sk,pk> into sidx
<sk,pk>
pidxOp
1. acquire an exclusive lock for the record
2. log (*update*, txnId, *insert*, indexId, <pk,record>)
3. insert <pk,record> into pidx
<pk,record>

Figure 3.5: Job pipeline for inserting a record

Now, let us explain how a record is inserted into a dataset. The first operator, denoted as "pidxOp", represents the primary index operator which gets an input frame that contains a set of $\langle pk, record\rangle$ pairs. For the example AQ1, there will only be one entry in the input frame. To insert the input entry into the primary index, the pidxOp goes through three steps denoted as 1–3 in the figure. First, it acquires an exclusive lock on the record, using its recordId, by following the deadlock-free locking protocol described in Algorithm 1 (in Section 3.4.2). The recordId is a $\langle datasetId, pkHashValue\rangle$ pair, where the pkHashValue represents a hash value of a primary key. (Since AsterixDB maintains locking information in main memory, we use a hash value of a primary key to reduce memory consumption instead of the entire primary key, which could be large.) After the lock is acquired, a logical log record for the entry to be inserted is written to the log buffer (i.e., the tail of the log file) in main memory. The log record includes a log record type (*update* in the example), a transactionId (denoted as txnId), an operation type (*insert* in the example), an indexId, and an entry ($\langle pk, record \rangle$). A transactionId is a $\langle jobId, recordId \rangle$ pair, i.e., a $\langle jobId, datasetId, pkHashValue \rangle$ triplet (since recordId is a $\langle datasetId, pkHashValue \rangle$ pair). With this triplet, we can identify which specific record-level transaction is working on which specific record belonging to a specific dataset. Also, an indexId is required to indicate which index in the dataset is affected by this operation since AsterixDB supports index-level logical logging. With this information in the log record, redo or undo can be easily done by replaying the operation or applying its inverse operation, respectively. After the log is written, the entry is inserted into the primary index and then the output of the operator, i.e., a frame including $\langle sk, pk \rangle$ pairs, is pushed to the secondary index operator denoted as "sidxOp".

The secondary index operator first writes a log record and then inserts the entry $\langle sk, pk \rangle$ into the secondary index. Unlike the primary index operator, the secondary index operator does not acquire any locks since the necessary lock was already acquired during the primary index operation and is still held according to the strict 2PL protocol. (Latching is used, of course, to ensure index operation consistency.) Then, the output of the operator, i.e., a frame including $\langle sk, pk \rangle$, is pushed to the commit operator denoted as "commitOp". In general, the commit operator writes a commit log record for each entry in the input frame. For AQ1, only one commit log record is written to the log tail.

All of the operations of the job pipeline in Figure 3.5 will be executed sequentially by a single thread. However, when there are multiple partitions of a dataset in a node and multiple records are inserted, each partition will have its own job pipeline executed by a different thread, so insert operations are executed in parallel by as many threads as there are dataset partitions. In addition to the job pipeline thread, there is another thread involved called the LogFlusher. The LogFlusher thread asynchronously flushes log records in the log tail to the log file on disk as soon as it recognizes the arrival of newly written log records. Instead of relying on a certain explicit group commit timer, it performs a group commit in a self-throttled manner based on the time spent on flushing the currently accumulated log records to the log file. If a small volume of log records arrives at the log tail, the thread can flush those log records quickly and then can flush the next batch of log records accumulated in the mean time as soon as possible. In contrast, if a large volume of log records arrives at the log tail, the thread will spend more time on flushing them due to the large volume while more log records are being accumulated and the volume of log records in the next batch will increase as well. Thus, the overall flush throughput can increase. Also, if the type of any of the flushed log records is commit, the thread also releases the locks held by the corresponding transaction based on the transactionIds in the commit log records. Note that we make sure that the strict 2PL protocol is followed for exclusive locks during the insert operation by acquiring an exclusive lock before a record is inserted and releasing it after the commit log record is flushed to disk.

3.5.2 Select Queries

In this section, we first explain how a non-index-only-scan select query is processed and then we explain how an index-only-scan select query is processed via the scheme described in Section 3.4.3.

Figure 3.6 shows a select query, AQ2, that finds songs whose release-year is less than 2016 and whose title is “Jude”, and returns the songs’ ids. As the title predicate is not *covered* using the idxReleaseYear index alone, AQ2 is a non-index-only-scan query.

```
for $x in dataset Song
where $x.release-year < 2016 and $x.title = "Jude"
return $x.id;
```

Figure 3.6: AQ2 to select records (non-index-only-scan)

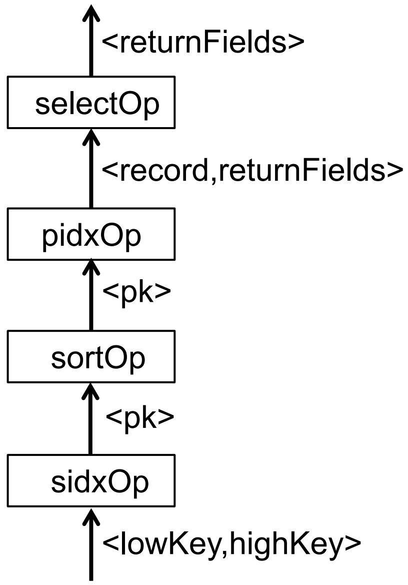


Figure 3.7: Job pipeline for a non-index-only-scan select query

Figure 3.7 shows a typical job pipeline for a non-index-only-scan query such as AQ2. First, the predicate covered by a chosen secondary index is converted into a key range from a low key to a high key, where the inclusion of the boundary key value depends on the specified predicate. (An equality condition ends up having the same value for the low key and the high key and the key range is inclusive.) Then, a frame that specifies the key range is pushed to the secondary index operator (sidxOp). This operator retrieves qualified entries from the key range and the entries are pushed (frame by frame) to the next operator. For a non-index-only-scan query, the secondary index operator does not acquire any locks for the qualified entries from the secondary index. (This is based on the primary-index-only locking design.) The next operator (sortOp) orders the input primary keys so that the primary index operator will have a better buffer behavior during performing its lookups for each primary key. For each input primary key, the primary index operator (pidxOp) acquires a shared-mode *instant* lock on a record retrieved from the index, and outputs the record to an output frame, where the record in the output frame is a copy of the record from the primary index. Note that this means that there are no locks held for records in the output frame since each instant lock is released as soon as it is granted[3]. Thus, records in the primary index could now be modified by other concurrent transactions. However, even if the records in the primary index are modified before the current search query is over, the record copies in the output frame are not affected by the concurrent modifications, and the rest of the current search query job pipeline can continue working based on the copied records. Due to the primary-index-only locking design, it is possible for the secondary key in an entry read from the secondary index to have been changed before the corresponding record is read from the primary index. However, the job pipeline still sees only committed records from the primary index due to the instant lock, and such a possible concurrent modification by another transaction does not affect the records in the job pipeline since they are copied versions. This is addressed by having the next operator in the flow, i.e., "selectOp", which makes sure that the predicate in the query still holds for the records copied from the primary index, and any non-qualified records will be discarded. In addition, any residual predicates not covered by the secondary index (e.g., the title predicate in AQ2) are also evaluated in the selectOp operator. Finally, a set of fields (which appeared in the return clause of the AQL) from the qualified records are returned.

```
for $x in dataset Song
where $x.release-year < 2016
return $x.id;
```

Figure 3.8: AQ3 to select records (index-only-scan)

Figure 3.8 shows a similar, but index-only-scan query, AQ3, where all referenced fields, i.e., the release-year and id fields in the query, are covered by the idxReleaseYear index.

[3] Again, the instant lock is used to ensure that all data read is committed.

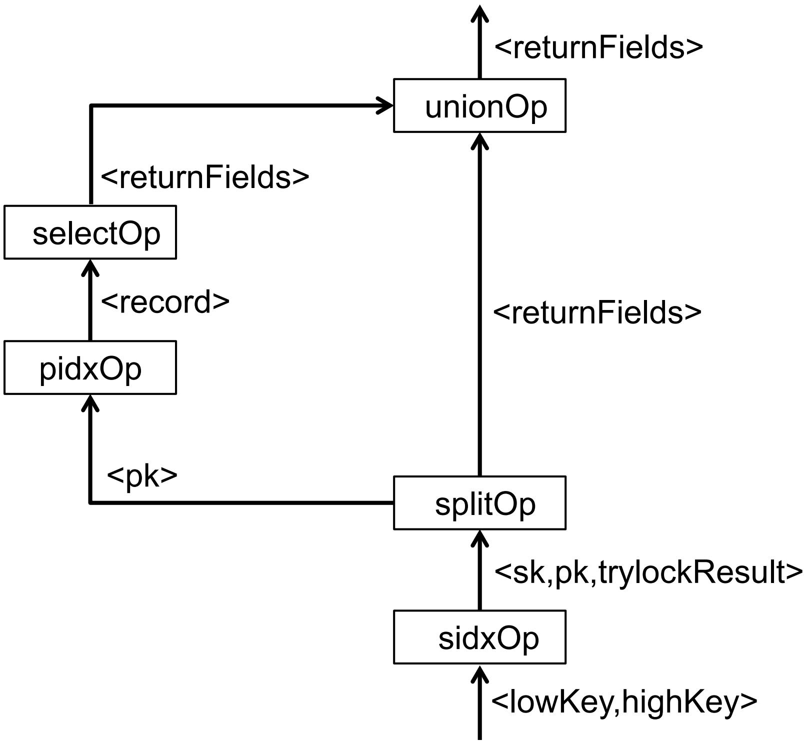


Figure 3.9: Job pipeline for an index-only-scan select query

Figure 3.9 shows a typical job pipeline for an index-only-scan query such as AQ3. Similar to the non-index-only-scan query, a predicate covered by a secondary index is converted into a key range, and the secondary index operator then searches qualified entries using the key range. However, unlike the non-index-only-scan query, the secondary index operator *tries* (without blocking) to acquire a shared-mode instant lock using the primary keys for each of the qualified entries, and each entry in an output frame includes the result of whether the lock was successfully acquired or not (as shown between the sidxOp and splitOp in the figure). Based on the result, the splitOp pushes successfully locked entries directly to the unionOp, but pushes the failed entries to the pidxOp on the left path. The left path, starting with the pidxOp followed by the selectOp, works exactly the same as the corresponding two operators did in the non-index-only-scan query. However, there is no sort operator between the sidxOp and pidxOp on the left path since the number of entries on this path will be usually very small, so any sorting effect would be minimal. Lastly, the unionOp outputs the entries coming from both execution paths.

In summary, the (hopefully many) entries that are successfully locked during the secondary index search avoid the costly validation steps that require accessing the primary index, and the number of failed entries that take the left path should be very small since the fine-granularity (record-level) transactions' short lifetimes make the chances of concurrent modifications very rare. Thus, while there is still a possibility to access the primary index, the two-execution-path job pipeline should make such queries work almost as efficiently as index-only-scan queries.

It should be noted that during this select-query execution, there is a type of undesired situation that may happen due to the read-committed isolation level. When the secondary index operator retrieves qualified entries from the secondary index, as shown in Figures 3.6 and 3.8, a qualified entry (such as the entry $\langle 2000, 1991 \rangle$ inserted by AQ1) that is retrieved from the secondary index could be changed concurrently in such a way that the new secondary key value in the modified entry (say $\langle 2002, 1991 \rangle$) still satisfies the given predicate (release-year less than 2016), causing the modified entry to be retrieved again by the index operator. Consequently, it is possible for multiple entries stemming from a single record to be retrieved and returned.

3.5.3 Deletes

Figure 3.10 shows an AQL query, AQ4, to delete records based on a predicate. The query deletes songs whose release-year is less than 2016.

```
delete $x from dataset Song
where $x.release-year < 2016;
```

Figure 3.10: AQ4 to delete records

Figure 3.11 shows a typical job pipeline for a delete operation such as AQ4. The job pipeline consists of a search phase followed by a delete phase. The search phase finds qualified records to be deleted in a way very similar to a typical non-index-only-scan query. The delete phase then deletes these found records in a manner similar to a typical insert operation. The job pipeline shown in Figure 3.11 indeed has the same operators that both the search and insert job pipelines have.

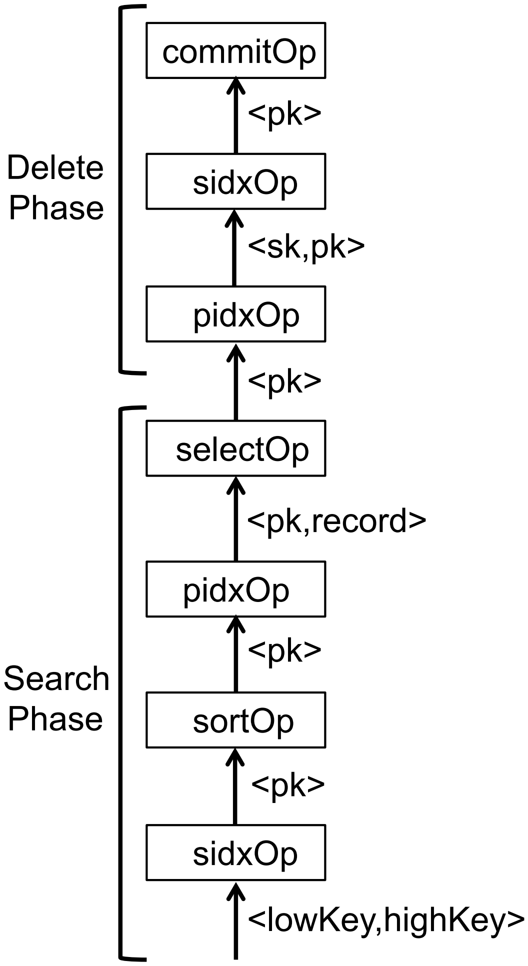


Figure 3.11: Job pipeline for deleting records

In the search phase, to search for the records to be deleted, a predicate covered by the index is converted into a key range. Then, the initial sidxOp searches for qualified entries based on the key range and the resulting entries are sorted by the sortOp. After that, the pidxOp reads the corresponding records while acquiring shared-mode *instant* locks to ensure that it reads committed records. Next, the selectOp re-evaluates the predicate over the records read from the primary index. After this search phase, the delete phase starts. To delete records, for each record, the primary index operator acquires an exclusive lock (following the deadlock-free locking protocol describe in Section 3.4.2) and then it writes a log record. After that, the record is deleted from the primary index. Next, for each secondary index entry to be deleted, the sidxOp creates a log record and then deletes the corresponding entry from the secondary index. Lastly, the commitOp writes a commit log record for each input entry. The LogFlusher thread flushes the commit log records and releases corresponding exclusive locks.

Due to the use of instant locks during the search phase, the records found during the search operation by a transaction T1 could possibly be modified and committed by another concurrent transaction before the delete phase of T1 begins. The following two possible situations can be caused by such a concurrent modification by another transaction.

- Case 1: A record identified to be deleted by T1 has already been deleted by another transaction after the search phase and before the delete phase of T1.
- Case 2: A record to be deleted by T1 has been deleted but then inserted (or has been updated) by another transaction after the search phase and before the delete phase of T1.

For the first case, the effect of attempting to delete such a non-existing record simply becomes a no-op. For the second case, however, the newly updated record will still be deleted by T1 even if the search predicate is no longer be satisfied by the updated record. These are both possible consequences of the read-committed isolation-level.

3.5.4 Self-Join Queries

Figure 3.12 shows a self-join query, AQ5, that finds songs that were released in the same year as songs whose title is “Jude” and returns these songs.

```
for $x in dataset Song
for $y in dataset Song
where $x.title = "Jude" and
      $x.release-year = $y.release-year and
      $x.id != $y.id
return $y
```

Figure 3.12: AQ5 to do a self-join

Figure 3.13 shows a job pipeline for a typical self-join query such as AQ5. Since this is a join query, there are two separate paths at the bottom that then are merged into one path at the joinOp. The left primary index operator in the job scans all entries from the primary index of a dataset while acquiring a shared-mode *instant* lock for each entry that it reads.

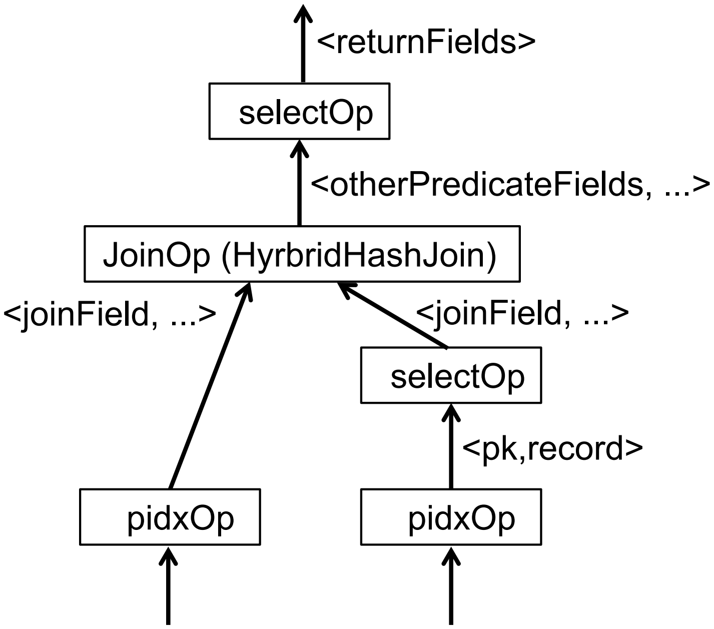


Figure 3.13: Job pipeline for a self-join query

Similarly, the right primary index operator scans all entries from the same primary index while acquiring a shared-mode instant lock for each entry. In this example, due to the title predicate in AQ5, there is also a select operator on the right path that pushes only qualified entries to the join operator. Then, the join operator pushes matched entries to the next operator, i.e., another selectOp, which evaluates a residual predicate, i.e., $x.id != $y.id.

Note that in this self-join query, the same dataset is scanned twice, once on the left path and once on the right path. Also, due to the use of instant locks, a record scanned from the left path may possibly be modified and committed by another concurrent transaction before the record is scanned on the right path. Suppose, for instance, that AQ5 does not have the predicate $x.id != $y.id, and a job of AQ5 reads from the left path a song whose id, title, and release-year are 1990, "Jude", and 2010, respectively. Suppose further that another concurrent transaction has modified the song's release-year to 2016 and has committed before the job of AQ5 reads the song from the right path. Consequently, the song will not be returned as the query's result. This is also an expected, but undesirable situation that can stem from the read-committed isolation level.

3.5.5 Summary

To summarize, we have explained the detailed transactional dataflow for record-level transactions using several AQL examples including insert, delete, select, and join queries. Also, we have discussed several possible but undesired consequences related to the read-committed isolation level. Though we have covered relatively simple operations in regard to these consequences, in general, if a query's job pipeline reads a record more than once in the pipeline, the value of the record may change from one read to the next read due to the use of only instant locks, and thus such consequences may happen.

3.6 Related Work

Many DBMSs offer several different isolation levels [33, 13, 6, 5] to allow programmers to trade off transactional guarantees for potential gains in performance [33, 37, 14]. Some transaction processing systems support the serializable isolation level even for multi-node distributed transactions. Spanner, Google's geo-scale distributed database system [24], and Calvin [62] both belong to this category. Traditional relational DBMSs such as Oracle, MS SQL Server, IBM DB2, and others, allow users to choose an appropriate isolation level ranging from the strongest (serializable) to weaker isolation levels such as repeatable-read, read-committed, read-uncommitted, and their variations. In contrast, most of the more recently appearing NoSQL systems [19, 21, 23, 26, 45, 16, 3] have opted for faster and more scalable operations, but fewer transactional guarantees; most support atomic read or write operations for a single key/value pair under eventual consistency [63]. The design choice of supporting local, record-level transactions in AsterixDB provides for ACID transactions for individual records at the read-committed isolation level. This is similar to NoSQL systems in terms of the granularity of a transaction, but, unlike typical NoSQL systems, it ensures consistency between a primary index and any number of secondary indexes for each dataset.

There is a long history of work in the logging and recovery area as well. One of the most influential recovery schemes is ARIES [51]. Also, in order to reduce logging overheads during normal processing and redo and undo costs during recovery or abort, physical, logical, physiological logging methods have been explored. Previous works by Lomet et al. [47, 48] provide a recovery method based on logical logging, but their method is not based on pure logical logging. It is based on a combination of logical logging in the transaction component layer and physical logging in the data component layer, where physical logging guarantees action consistency for indexes so that logical logging can work correctly based on action consistency when a recovery operation is required. Our work differs in its approach to maintaining action consistency. AsterixDB maintains action consistency through the use of a no-steal buffer management policy and disk component shadowing.

Lastly, many deadlock solutions such as prevention, avoidance, and a variety of detect-and-resolve strategies have been explored [27, 25, 7]. Our deadlock-free protocol falls into one of the well-known prevention strategies, namely hold-and-wait is simply not allowed in any situation. In AsterixDB, we were able to avoid the hold-and-wait situation by leveraging the record-level transaction model plus employing a frame-splitting technique which can effectively release all of the locks held for records when a "wait" is required to acquire a lock.

3.7 Conclusions

In this chapter, we have explained record-level transactions in AsterixDB. These fine-granularity transactions enable records to be modified and committed as soon as possible by reducing the amount of work to be done in a "transaction", thus making the data queryable in a timely manner. Unlike typical NoSQL systems, AsterixDB maintains consistency between a primary index and any number of secondary indexes in a dataset. All secondary indexes are partition-local to the primary index, so record-level transactions are always local transactions and do not suffer from the two-phase commit (2PC) overhead that would otherwise be required for the atomicity of distributed transactions.

In addition, we have described AsterixDB's novel recovery method based on the use of a no-steal/no-force buffer management policy, index-level logical logging, and LSM disk component shadowing. The no-steal policy and disk component shadowing together provide action consistency for each index's logical operations, so a single logical log record generated from a single logical operation in an index can correctly capture all necessary information for the corresponding redo and undo operations. Also, due to the no-steal policy and disk-component shadowing, crash recovery is greatly simplified—during recovery, any disk components without their validity bit being set are simply removed and then the winner transactions' effects that were not made durable before the system crashed are selectively replayed.

We have also presented the lock-based concurrency control mechanism used in AsterixDB. It supports record-level transactions at the read-committed isolation level by employing strict 2PL for write transactions but letting read transactions employ instant locks. In addition, we have described the primary-index-only-locking strategy used in AsterixDB and its ability to still support index-only-scan queries involving secondary indexes. We also described how AsterixDB can achieve deadlock-free locking by avoiding the hold-and-wait situations that are necessary to form a deadlock.

Lastly, we have described in detail the flow of transactional operations including insert, delete, select, and join queries. The descriptions included a job pipeline (a DAG of operators) for each of the queries and described the detailed processing steps such as when locks are acquired and released, when log records are created and flushed, when indexes are accessed and updated, etc. Also, we highlighted the expected but undesired situations that can occur, hopefully rarely, with the read-committed isolation level in use during those operations.

Chapter 4

Comparative Study of LSM Spatial Indexes for Dynamic Point Data

4.1 Introduction

There have been many studies in the spatial data processing area, from proposing new spatial index data structures to evaluating and analyzing those proposed indexes [59, 31]. A large portion of the studies, however, have dealt with static data workloads in the sense that once data are loaded into a spatial index, the index becomes read-only without having incremental insertions. On one hand, the static data workloads make sense for applications such as car GPS navigation systems since the number of spatial objects added to the map daily, monthly, or even yearly, is relatively small. However, this is not the case for today's social media services, for which it is critical to add newly generated data in a timely manner and make them queryable. In addition, many of the past studies have counted only I/Os without including CPU cost, and most have considered only the index access cost instead of measuring the overall system's end-to-end processing cost, which includes not only the index-access cost, but also other possible costs such as query compilation, parallel job scheduling, logging, locking, etc. Even though I/O cost is one of the most expensive costs in general, CPU cost, especially when dealing with spatial data, is not negligible.

A few recent studies [41, 9] have gone further and have included dynamic workloads that reflect continuous incremental inserts and concurrent queries. However, these studies did not measure end-to-end time in the context of a full-function data-processing system.

Considering all of these limitations, it is now the right time to revisit existing spatial indexing methods in order to evaluate their pros and cons in light of a high frequency of insertions as well as concurrent queries in this Big Data era. The contributions of this chapter are as follows:

1) Among representative, disk-resident spatial indexing methods that have been adopted by major SQL and NoSQL data processing systems such as Oracle, IBM DB2, MS SQL Server, MySQL, PostgreSQL, MongoDB, Lucen/Solr, and Lucen/Elastic Search, we have implemented five variants of these methods in the form of Log-Structured-Merge (LSM) spatial indexes in the context of Apache AsterixDB. This means that the five implemented indexes are not only spatial indexes, but also LSM-trees, which provide superior handling of a high frequency of insertions by having deferred-update, append-only data structure features. Due to AsterixDB's wholly adopted LSM-tree storage layer, called its "LSM-ification" framework [9], each implemented index in AsterixDB can effectively behave as an LSM-tree almost "for free". Also, due to the common use of the AsterixDB code base, these five indexes are compared as fairly as they can be in terms of the overall system's end-to-end processing cost.

2) Among the five implemented indexes, three are based on B-trees[1], one on R-trees, and the other on inverted indexes, which are three of the most popular disk-resident indexes. With

[1] In this chapter, the term B-tree represents a B^{+}-tree.

this setup, we can answer an interesting question: Among these three indexes, which one is the most appropriate as a choice to support spatial indexing? In other words, if built-in indexes such as B-tree and inverted index are used to implement spatial indexing, can they be as efficient as or even superior to R-trees? The results of this study can provide valuable insights for practitioners and developers who are interested in using and/or implementing a good spatial index, considering what they have in their systems, by knowing the pros and cons of the candidate index types.

3) We focus on real-world geo-tagged point data in our evaluation and cover a broad spectrum of workloads—from a “load once, query many” case without incremental inserts to a case where continuous concurrent insertions are mixed with concurrent queries—in order to capture both static and dynamic workload characteristics. We focus solely on point data in this chapter because point data is one of the most important data types considering the amount of such data coming from modern Big Data applications.

The rest of the chapter is organized as follows. First, we describe the details of the five spatial indexes. Next, we present our evaluation strategy and results. After that, we further discuss closely related studies. Finally, we provide conclusions.

4.2 Five LSM Spatial Indexes

This section describes the five spatial indexes named as follows:

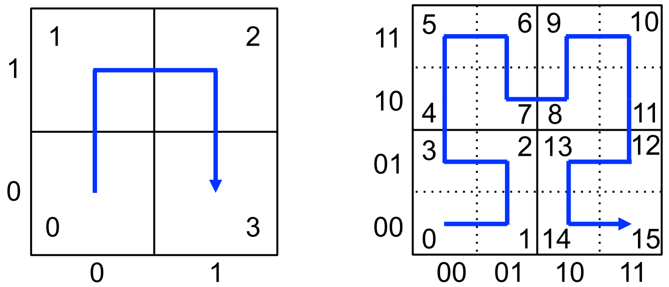
- R-tree
- Dynamic Hilbert B-tree (DHB-tree)
- Dynamic Hilbert Value B-tree (DHVB-tree)
- Static Hilbert B-tree (SHB-tree)

- Spatial Inverted File (SIF)

The DHB-tree, DHVB-tree, and SHB-tree indexes are each implemented using AsterixDB's LSM secondary B-tree index as a basis. The SIF index is implemented based on the LSM secondary inverted index. The R-tree is the existing LSM secondary R-tree index of AsterixDB except with an index-size optimization of storing a point consisting of x/y coordinates as a point instead of as a minimum bounding rectangle (MBR) in a leaf node. Since we already explained the LSM secondary R-tree index in Section 2.3.3, we focus on the other four indexes in this section.

4.2.1 DHB-tree and DHVB-tree

Details of indexing and querying two dimensional spatial point objects using a B-tree with space-filling curves such as Hilbert and Z curves are described in previous studies [55, 39, 29, 46]. Our DHB-tree and DHVB-tree were implemented using the main ideas of the study [46]; here we review the key ideas such as how to store two dimensional point objects in a traditional B-tree and how to search for points overlapping with a given query region. We use the Hilbert curve since it is considered to be superior to other space-filling curves [39].



(a) First-order curve (b) Second-order curve

Figure 4.1: Hilbert curve example

A Hilbert curve is a fractal space-filling curve. First and second *order* curves are shown in Figure 4.1. The first-order curve divides a square space into four sub-squares. The second-order curve divides each sub-square into another four sub-squares. The order represents the level of recursive divisions and also the number of bits required to cover each axis of the square space. Higher-order curves have a higher resolution power for distinguishing two different points. Each sub-square has a sequence number along the curve. The level of the space dictates the resolution power, and two points in the space will have the same sequence number if they belong to the same sub-square at the lowest level.

Since the Hilbert curve is a fractal, the way to draw the curve can be captured in a state-transition diagram that essentially provides a mapping from (x,y) coordinates to a sequence number along the curve and vice versa. Figure 4.2 shows the state diagram on the left. On the right, it shows how to map a point with (x,y) coordinates, (10,01), to its Hilbert sequence number, 1101, using the state diagram, where all numbers are binary. Initially, the state is 0. First, by taking a bit value from each coordinate and appending two numbers in x and then y, coordinate 10 can be mapped. State 0 in the diagram maps the coordinate 10 to a sequence number 11. Then, the state transits to state 2 by following the associated arrow. The next two bits of the sequence number can be obtained in a similar manner. The final sequence number is obtained by combining them together. Once a sequence number for a given point object is computed, a traditional B-tree can be used to store the point object just like any other one-dimensional data, and objects close in space will tend to be close in their Hilbert numbering as well.

Now, we explain the main idea of how a spatial range query is processed using the B-tree index, where the query searches for points overlapping with a given region which could be any shapes such as rectangle, circle, or polygon. First, the MBR of the given region is computed. Second, note that the MBR's boundary intersects with a set of sub-squares at the lowest level of the Hilbert space. Among the intersecting sub-squares, the sub-square with the smallest sequence number can be found. The intuition is as follows. Once an MBR is given, we can determine which quadrants (or sub-squares) at the first level are overlapping

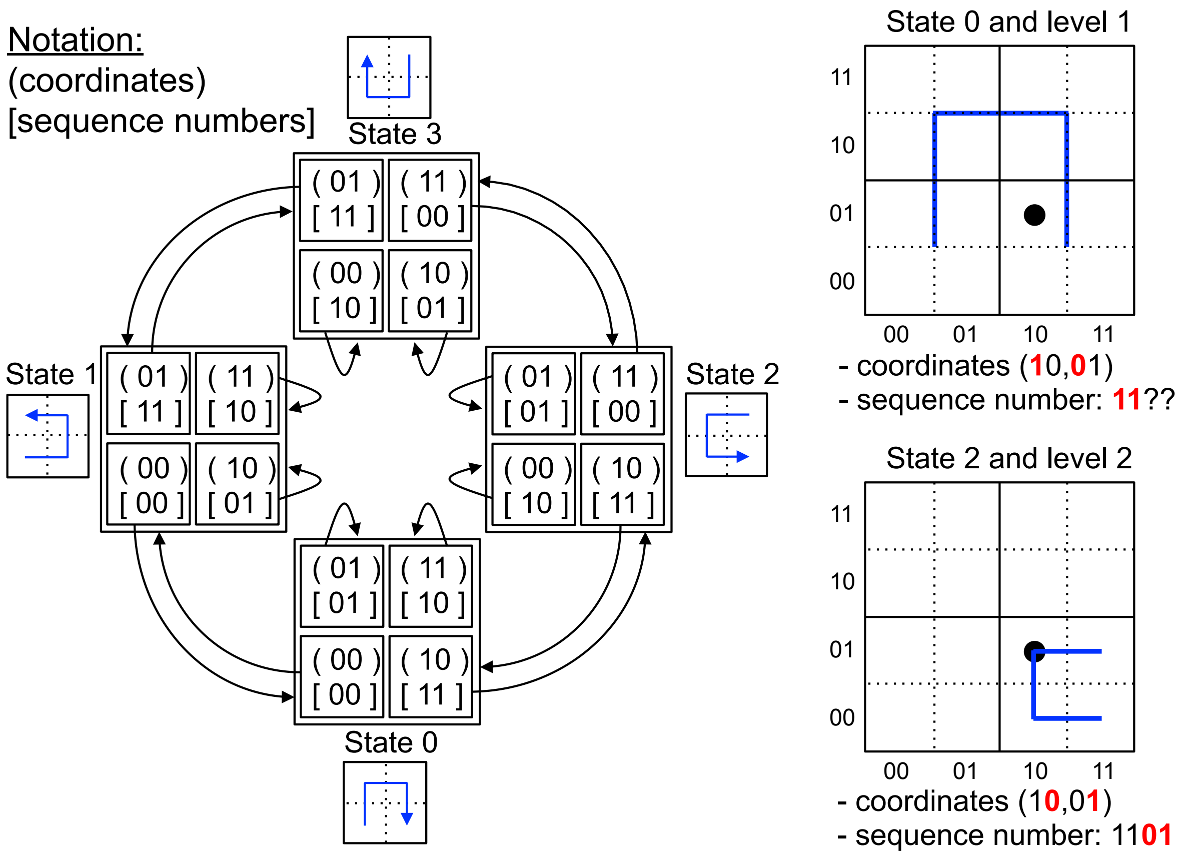


Figure 4.2: Mapping point coordinates to a Hilbert sequence number using Hilbert curve state diagram

with the MBR. Also, we can tell which sub-square has the smallest Hilbert sequence number among them at the level using the state diagram. We can then repeat this step with the sub-square by reducing the MBR into the region that overlaps with the sub-square until we reach the lowest level. Third, a B-tree range search is conducted using the resulting sub-square's sequence number as a low key. The range search cursor returns probed entries until the cursor goes out of the MBR region, which is determined as soon as a probed index entry does not overlap with the MBR. Then, the next range search is conducted with a sub-square whose sequence number is the smallest, but greater than the latest probed out-of-region entry's sequence number. The search process terminates when there is no such boundary sub-square left. False positives introduced by using an MBR over the actual query region are discarded at the post-processing step. See [46] for further details of computing boundary sub-squares for a given MBR.

In general, both DHB-tree and DHVB-tree indexes can store point objects and support range queries as described above. However, there is a key difference between them in terms of their ways of comparing two points. One way is to compute full Hilbert sequence numbers for the two given points and then compare the resulting sequence numbers. Another way is to compare two points relatively, without getting their full sequence numbers as follows: Recall how we map a given point's coordinates to a sequence number shown in Figure 4.2. As soon as the state diagram outputs two different sequence numbers for the two compared points' coordinates, while we are moving from the most significant bit to least significant bit in each coordinate, the comparison can stop and we can tell the comparison order. This relative comparison method can be cheaper than the absolute sequence number comparison method if the two points lie far away from each other. The DHB-tree adopts the relative-comparison method, which implies that stored entries in a DHB-tree index have coordinates and primary key fields, but no stored sequence number. Also, this means that whenever a comparison is needed, it will go through the relative comparison steps again. The DHVB-tree instead adopts the absolute-comparison method, so all entries in the DHVB-tree index have not only coordinates and primary key fields but also a stored Hilbert sequence number. (See Table 4.2 for the difference between an entry in a DHB-tree index and an entry in a DHVB-tree.) This is why there is "V" in its name, DHVB-tree, where the V represents that the value of Hilbert sequence number is stored in it. The performance tradeoffs between these two methods will be studied in the experiments.

4.2.2 SHB-tree

In the SHB-tree indexing method, a two-dimensional space is statically decomposed into a k-level $2^n \times 2^n$ grid hierarchy, where k and n are numbers chosen and fixed when the index

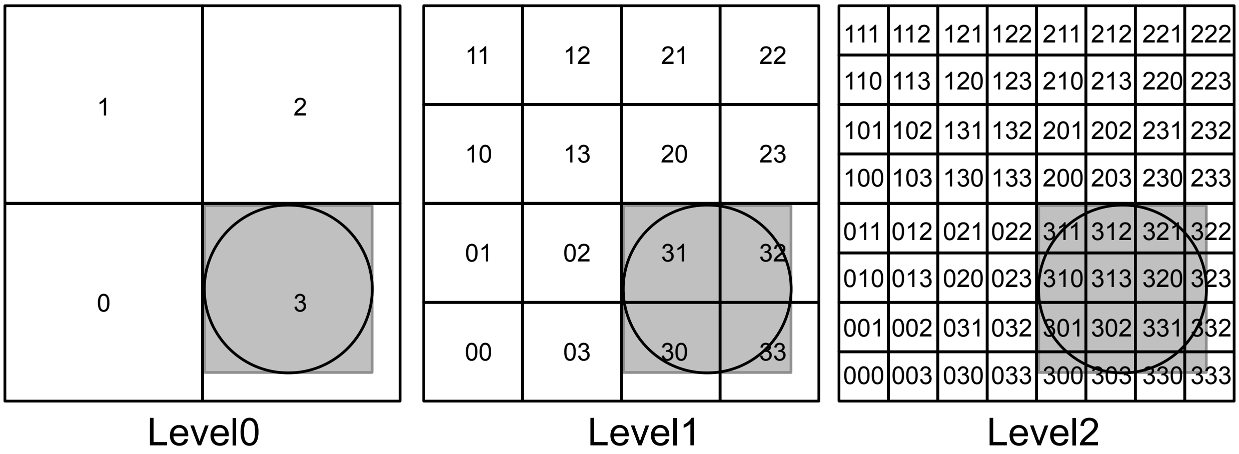


Figure 4.3: An example 3-level 2×2 grid hierarchy and cell numbers at each level

is first created. The top level (*level* 0) has $2^n \times 2^n$ cells, and each successive level further decomposes a cell at the previous level into $2^n \times 2^n$ sub-cells. Also, the top-level cells are numbered in a linear fashion by using a Hilbert curve, and so are the sub-cells belonging to a parent cell at the previous level with prepending their ascendant's cell numbers. Figure 4.3 shows an example of a 3-level 2×2 grid hierarchy and the cell numbers at each level. Since the top level has four cells and each cell has four sub-cells, they use the sequence numbers of the first-order Hilbert curve shown in Figure 4.1(a).

In order to store a spatial (non-point) object in the SHB-tree index, the MBR of the object is decomposed into a set of cells, each of which either overlaps with the MBR or is completely covered by the MBR according to the following *covering rule*. If the MBR completely covers a cell at a level, that cell is said to be *covered* by the MBR. A covered cell is included in the result set and is not decomposed at lower levels. The rest of the overlapping cells, if any, are further decomposed at the next level. This rule applies at all levels of the grid hierarchy except that the overlapping cells at the lowest level are included in the result set without further decomposition. A point object (which is our sole focus in this chapter) is always mapped into a cell at the lowest level. Once such a set of cells are computed for an object being inserted, each cell number as a key is inserted into the underlying B-tree, where a cell number also includes the level number of the cell at the end. Thus, each entry in an SHB-tree index consists of a ⟨*cell number*, *primary key*⟩ pair.

When a query region is given, a set of cell numbers for the MBR of the query region are computed (as detailed above) and used to search matching cell numbers of the indexed objects. Searching the set of cell numbers of the MBR over the index can be optimized when there are consecutive cell numbers by forming a range search with those consecutive cell numbers. This optimization can effectively reduce the number of index searches. False positives caused by 1) the MBR of the query region, 2) the MBR of stored spatial object, and 3) the overlapping, non-covered cells at the lowest level are all discarded during a post-processing step, where the first cause is not relevant to a rectangle region query and the second one is not relevant to a point object. In general, a finer granularity of cells at the lowest level can reduce the number of false positives by the third cause, but it may also increase the number of index searches that result from processing more cells for a given query region's MBR.

To illustrate, in Figure 4.3, the circle represents an original query region and the gray rectangle represents its MBR. This MBR will first be mapped into the following cell numbers: 31 at level 1, and 300, 301, 302, 303, 320, 321, 322, 323, 330, 331, 332, and 333 at level 2. Then, consecutive cell numbers can be mapped into ranges. For example, the cell numbers 300, 301, 302, and 303 can form a range from 300∼303. Also, we can easily obtain the ranges of the lowest-level cell numbers corresponding to a given intermediate cell number since the grid hierarchy is static. This fact can be exploited to determine consecutiveness between two cell numbers at the lowest level, even when the cells belong to two different parent cells (such as 303 and 310 in the figure). In the example, the MBR ends up forming a range from 300∼333. Thus, there can be a single B-tree range search for this example range query.

This approach to spatial indexing is particularly interesting since it has been used for the spatial index support in MS SQL Server [30]. Also, the linear Quadtree [32] in Oracle

Spatial [42] works in a similar manner except that the number of sub-cells in a cell is fixed to four and the cells are numbered in the same manner that a quadtree numbers its cells.

Before moving on, it is worth contrasting the key difference between the SHB-tree and DHB-tree (or DHVB-tree) approaches. Conceptually, both the SHB-tree and DHB-tree need to identify a set of ranges overlapping with a given query region. However, the way to identify the ranges is different. The SHB-tree can *compute* the relevant ranges for a given region without considering the current entries in the underlying B-tree. In constrast, the DHB-tree identifies the ranges *on the fly* while retrieving current entries in the underlying B-tree as described in Section 4.2.1.

4.2.3 SIF

SIF provides a way to support spatial indexing based on the use of (keyword) inverted index. Its main idea is similar to the SHB-tree approach except that the SIF method uses an inverted index. This means that a spatial object is mapped to a set of cell numbers which are then stored in an inverted index as a set of tokens (essentially spatial keywords). Similarly, a region query goes through the mapping step to obtain a set of cell numbers and then searches these cell numbers just like a set of keyword tokens are searched for in an inverted index. This inverted-index-based spatial indexing approach was adapted from the spatial-keyword inverted file (SKIF) index [43], and a survey regarding spatial-keyword query processing can be found in [22]. SKIF can support location-based text search queries with an inverted file capable of indexing and searching both textual and spatial data in a similar, integrated manner by proposing a distance measure called *spatial tf-idf*. Essentially, this is enabled by partitioning a spatial region into a set of predefined grid cells and dealing with the resulting cell identifiers as spatial tokens.

In general, a traditional inverted index supports searching for exact matches for a given string *token*, where a set of string tokens can be generated from a given query string. Also, an inverted index may provide richer search types, such as prefix search or range search, according to the underlying data structure, such as trie or B-tree, used to implement the index. In our study, we chose to use the LSM inverted index in AsterixDB as a black box to implement the SIF index. As a result, our SIF index only supports exact match. (Note that this implies more overhead in a SIF index than in an SHB-tree due to the lack of range search support in the SIF index.) As described in Section 4.2.2, a range search in the SHB-tree index can reduce the number of searches when there are contiguous cell numbers resulting from a given query region. Nonetheless, the following related optimization is available for the SIF index to reduce the number of searches over the underlying LSM inverted index: a set of child cell numbers can be replaced with their common parent cell number if these child cells completely cover the parent cell. Note that this optimization requires each whole spatial object to be stored at all levels. For example, suppose that a point object is mapped into cell number 312 at level 2 in Figure 4.3. The point object will be mapped into not only cell 312 at level 2, but also into cell 3 at level 0 and cell 31 at level 1. Then, all three cell numbers will be stored in the SIF index. By doing this, when the query region in the figure is given and an exact match with cell number 3 at level 0 is processed, the point object can be found at level 0. However, note that this optimization comes with an increased index size caused by storing a point object at all levels in the grid hierarchy. The increased index size can degrade this indexing method's performance, as will be studied in the experiments.

4.3 Evaluation Plan

We can now proceed to study the performance of the five Big Data spatial indexing alternatives. In this section, we describe our evaluation strategy.

4.3.1 AsterixDB Setup

For the evaluation, we used an 8-node cluster to host an AsterixDB instance, with each node running CentOS Linux with a Quadcore Intel Xeon CPU E3-1230 V2 3.30GHz, 16GB RAM, 1 GBit Ethernet NIC, and three locally attached 7,200 rpm SATA drives setup as RAID 0. Of the available 16GB, AsterixDB was given 7GB. The rest of the physical memory was locked to prevent the underlying OS from utilizing it except that 2.5GB was left for the OS. In each node, a dataset was stored in four separate partitions in order to provide more parallelism. In total, a dataset in the 8-node AsterixDB instance was horizontally partitioned into 32 partitions based on primary keys.

In AsterixDB, the secondary indexes of a dataset are local to the primary index, i.e., they are located in the same node as the data being indexed. All partitions of the primary and secondary indexes of a dataset in a node share the same memory budget for their in-memory components. This means that once the memory budget of a given dataset is exceeded, each in-memory component belonging to the dataset is scheduled for a flush operation. The scheduled flushes are conducted asynchronously [9]. AsterixDB uses two memory components per index (i.e., double buffering) to avoid stalling incoming inserts during flushes. As the number of disk components is increased, merge operations are also triggered by a merge policy. We used AsterixDB's default merge policy, called the *prefix merge policy*, which relies on components' sizes *and* the number of components to decide which components to merge. It works by first identifying the smallest ordered (oldest to newest) sequence of components such that the sequence does not contain a single component that exceeds a threshold size M but where either the sum of the components' sizes exceeds M or the number of components in the sequence reaches another threshold C. If such a sequence of components exists, the components in the sequence are merged into a single component and then the originating sequence of components is deleted.

Table 4.1 shows the AsterixDB configuration parameters used for our experiments, including the chosen threshold values for the prefix merge policy.

Parameter	**Value**
Memory budget for in-memory components per dataset	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 mergable component size of prefix merge policy (M)	1GB
Max component count of prefix merge policy (C)	5

Table 4.1: Settings used throughout these experiments

We implemented the DHVB-tree index by using 64-bit Hilbert sequence numbers and the same resolution power was given to the DHB-tree index. Also, we configured each of the SHB-tree and SIF index to have a 6-level $2^4 \times 2^4$ grid hierarchy. A cell's size at the bottom level is around 2 meters × 1 meter assuming 2m as one unit for the x axis and 1m as the unit size for the y axis. Table 4.2 shows the details of each index entry's fields with their sizes at the logical level. (The physical implementation level includes more bytes for metadata.) In the table, a point field consists of two double values that represent x/y coordinates. The DHVB-tree index's entry includes both a Hilbert sequence number and the original point, as the original point is still required to check whether the point overlaps with the MBR of a query region during range searches over the underlying LSM B-tree. This is because the Hilbert sequence number cannot be used directly for that purpose unless the sequence number is mapped back to the original coordinates. (We want to avoid paying that computation cost for the same reason that the Hilbert sequence number is computed upfront, stored in the index, and then used during searches.) Also, each index's entry includes the original point in order to support *index-only scan* queries (i.e., when an index can be a *covering index*) when a query includes predicates that can be solely evaluated using the data in the secondary index. AsterixDB supports index-only scan queries for any type of secondary indexes in general.

The cell number field for the SHB-tree and SIF indexes occupies 7 bytes, including 6 for a 1-byte cell number at each level and one byte for a level indicator. For a point object in the SHB-tree index, the level indicator always has the bottom-level indicator value.

Index	**Fields of an index entry (size in bytes)**
DHB-tree	point (16), pk (8)
DHVB-tree	Hilbert sequence number (8), point (16), pk (8)
R-tree	point (16), pk (8)
SHB-tree	cell number (7), point (16), pk (8)
SIF	cell number (7), pk (8), point (16)

Table 4.2: Each index entry's fields with their sizes in bytes

4.3.2 Spatial Dataset

We obtained real-world GPS point data from *OpenStreetMap* [4], which includes more than 2.7 billion points. Because the obtained data itself included only points represented as a latitude and a longitude, we generated synthetic tweets using the obtained point data for this evaluation. More specifically, we uniformly sampled 1.6 billion points out of the 2.7 billion points and augmented them with synthetic tweet data. The tweets have fields such as user, message, sending time, sender location, etc, where the sender location field was filled with the GPS point. The tweets were generated in the Asterix Data Model (ADM) format with monotonically increasing 64-bit integer keys. The average size of a tweet was 1KB.

4.3.3 Workload

We used two types of workloads: one is static and the other is dynamic. The static one has no incremental inserts once the data are loaded. In contrast, the dynamic one has continuous data arrivals while also having concurrent queries. For queries, we included both select and join queries, all of which use a secondary spatial index as an access path. Also, the queries fall into two categories. One is an index-only-scan query that only accesses a secondary spatial index. The other is a non-index-only-scan query that searches the secondary spatial index first and then accesses the primary index due to a query predicate that is not covered by the secondary index alone. (Note that we did not evaluate workloads with frequent updates and deletes since the important characteristics of the data workloads from mobile devices and sensors are insert-intensive and append-only.) The detailed flows of the workloads are described below.

4.3.3.1 Static Workload

For the static workload, we pre-generated 1.6 billion tweet records amounting to 1.6TB in eight tweet files, each of which includes 200 million records. The files are placed on the 8-node AsterixDB cluster, one file per node. Also, we prepared a query seed tweet file that includes 10,000 uniformly sampled records from the 1.6 billion tweet records and placed this file on one of the nodes. This file was used to ensure that generated queries had at least one actual answer. Besides the AsterixDB cluster, we used another client node to drive the static workload. All operations in the workload were executed using AsterixDB's REST API. The overall workload consists of the following steps:

Step 1: Load the 1.6 billion tweets into the Tweet dataset's primary index. Also, load the file of 10,000 seed tweets into the QuerySeedTweet dataset's primary index.

Step 2: Create a secondary spatial index of a given type (DHB-tree, DHVB-tree, R-tree, SHB-tree, or SIF) on the Tweet dataset's sender location field.

Step 3: Issue a total record count query. The query scans the Tweet dataset's primary index to minimize the OS file system's buffer cache effect for the next queries.

Step 4: Issue 5,500 circle region select queries sequentially by alternating the radius from

0.00001, 0.0001, 0.001, 0.01, to 0.1 degree which corresponds to around 1m, 10m, 100m, 1km, 10km, respectively. Each such query searches all points overlapping with the given circle region and returns the number of points. The center of the circle in each query is obtained from a randomly chosen record in the query seed tweet file. The first 500 queries are used for cache warm-up, so their performance numbers are not included in the reported results.

Step 5: Issue 200 circle region join queries with varying radii from 0.00001, 0.0001, 0.001, to 0.01 degree. Each query is an index-nested-loop-join-based count query. More specifically, each query first obtains 100 join seed tweets from the QuerySeedTweet dataset; the obtained tweets' sender locations are used as center points to create 100 circles. These 100 circles are then used as query regions to search for overlapping points in a spatial index. Finally, the number of overlapping points is returned.

The 0.1-degree case was excluded for the joins since the 0.01-degree join query already may span more than 300km^2 with 100 1km-radius circles.

Overall, Step 1 is executed once and Steps 2–5 are repeated for each type of spatial index, where Step 2 creates a secondary index after dropping an existing secondary spatial index, if any. Also, the complete set of steps are run twice, once for the index-only scan queries and once for the non-index-only scan queries. The non-index-only scan queries include an additional range predicate on an integer field not covered by the secondary spatial index. The result counts from both types of queries are made to be the same by making the additional predicate actually filter nothing out.

It is worth mentioning how non-index-only-scan queries are processed in AsterixDB. Queries that use a non-covering secondary index go through the following steps: 1) search for all qualified candidate entries from the secondary index, 2) sort the candidate entries based on the primary key, 3) search for each primary key after an instant lock[2] is acquired, and 4) do post-processing to remove any false positives from the secondary index results and evaluate any predicates not covered by the secondary index. Unlike a traditional DBMS, during the secondary index search, no locks are acquired. Instead, all necessary locks are acquired during the primary index lookup as detailed in Chapter 3. Due to the primary-index-only-locking behavior, there could be a pontential inconsistency between the primary index and a secondary index, so the primary index lookup with locking and the post-processing steps are always performed, regardless of whether the secondary index can generate false positives or not. The sorting by primary key is for getting better cache behavior during the primary index lookup.

In contrast, recall from Chapter 3 that index-only-scan queries do not include the sorting or primary index lookup steps. Also, post-processing was required only if the secondary index lookup results may include false positives. For index-only-scan queries, locks are acquired during the secondary index search, but the locks are not instant and they must thus be released when the query is over.

4.3.3.2 Dynamic Workload 1

Dynamic workload 1 studies the data ingestion scalability of each index in terms of the supportable number of inserts per second (IPS) by varying the number of nodes hosting the AsterixDB instance. We used two sets of nodes, one set for the AsterixDB instance as described in the static workload and another for tweet-generator clients. Similar to the static workload, using the pre-generated tweet files as source files, a client generates and sends tweets to the AsterixDB instance over the network using a feature of AsterixDB called *data feeds* [35], which is a mechanism for having data continuously arrive into AsterixDB from

[2] An instant lock is released as soon as the lock is acquired. This is possible due to the record-level transaction semantics in AsterixDB; see Chapter 3 for more details.

external sources and for having the data incrementally populate a managed dataset and its associated indexes. Inserting records through a data feed instead of insert statements avoids repeated query compilation and remote query launching overheads. A tweet-generator client in a node was forced to be fast enough to saturate a single-node AsterixDB instance in terms of IPS. As the number of AsterixDB instance nodes increased, the number of tweet-generator client nodes was increased accordingly.

In dynamic workload 1, after a Tweet dataset is created with a secondary spatial index on the sender location field, each tweet-generator inserts tweets for 1 hour. These operations are repeated for each index. There is no query for this workload, so it simply addresses the question of ingestion capacity.

4.3.3.3 Dynamic Workload 2

Dynamic workload 2 includes both data ingestion and concurrent queries. In this workload, 8 tweet-generator clients generate tweets and ingest them into an 8-node AsterixDB instance for 1 hour. At the same time, 8 query-generator clients generate spatial range queries in the same manner as described in the static workload's Step 4 and submit them to the AsterixDB instance. Both the tweet and query generators used the pre-generated tweet files as source files to obtain the tweets to be inserted and coordinates to be queried, respectively. We use another 8 nodes for the 8 tweet generators and 8 query generators, with a tweet generator and query generator per node. In addition, in order to study the trend of the query response times as the tweet generator's ingestion rate is varied, we make the tweet generator clients sleep for 1 millisecond after every ingestion of 1, 10, 100, or 1000 record(s), respectively. The complete workload is run twice, once with index-only-scan queries and once with non-index-only-scan queries, as was done for the static workload.

It is worth contrasting the implication towards LSM components between the static workload and the dynamic workload 2. For the static workload, there is only one disk component for each index (in each storage partition) and the disk component contains all of the corresponding entries for the loaded records. In constrast, for the dynamic workload 2, queries and data ingestion happens concurrently from an empty database. This means that while the queries are being processed, in-memory components of each index are populated and flushed by the concurrent data ingestion, and also the disk components (created from the flushes) are merged continuously according to the merge policy. Thus, the number of components and the number of entries that queries need to search will be quite different for the two workloads. Also, based on these aspects, we can tell that the dynamic workload 2 is more appropriate for evaluating LSM spatial indexes' performance by including the dynamic nature of LSM indexes.

4.4 Experiments and Results

We now proceed to study the characteristics of the different spatial indexes under the various workloads of interest.

4.4.1 Results of Static Workload

4.4.1.1 Index Size and Creation Time

	pidx	**dhbtree**	**dhvbtree**	**rtree**	**shbtree**	**sif**
Index size (GB)	1001.53	46.24	59.66	47.74	62.64	353.70
Index creation time (minutes)	85.02	25.74	18.02	17.08	17.14	46.60

Table 4.3: Index size and creation time

Table 4.3 shows the primary index (denoted as "pidx") size after loading 1.6 billion records and the elapsed time for the loading. Also, it shows the elapsed times for creating each type of spatial secondary index on the sender location field of 1.6 billion records and the corresponding index size. Among the secondary indexes, the R-tree and SHB-tree indexes are similar and faster than the others for loading. The SHB-tree index is bigger than the R-tree index due to the entry-size difference shown in Table 4.2. The DHVB-tree index is faster than the DHB-tree index. The SIF index is the slowest. The explanation for these results is as follows.

First, the SIF index stores a point object 6 times, once for each level of the 6-level grid hierarchy, so its load time is much slower than the others. Next, once the DHVB-tree index computes a Hilbert sequence number for each sender location during index creation, which involves mainly sorting all entries and bulk-loading them, the sequence number is reused until the sorting is over. In contrast, the DHB-tree index compares pairs of entries relatively. This relative comparison must be repeated until the end of sorting, and there is nothing to reuse for the comparisons. Thus, the DHB-tree index involves more overhead than the DHVB-tree index in terms of its sorting cost during index creation.

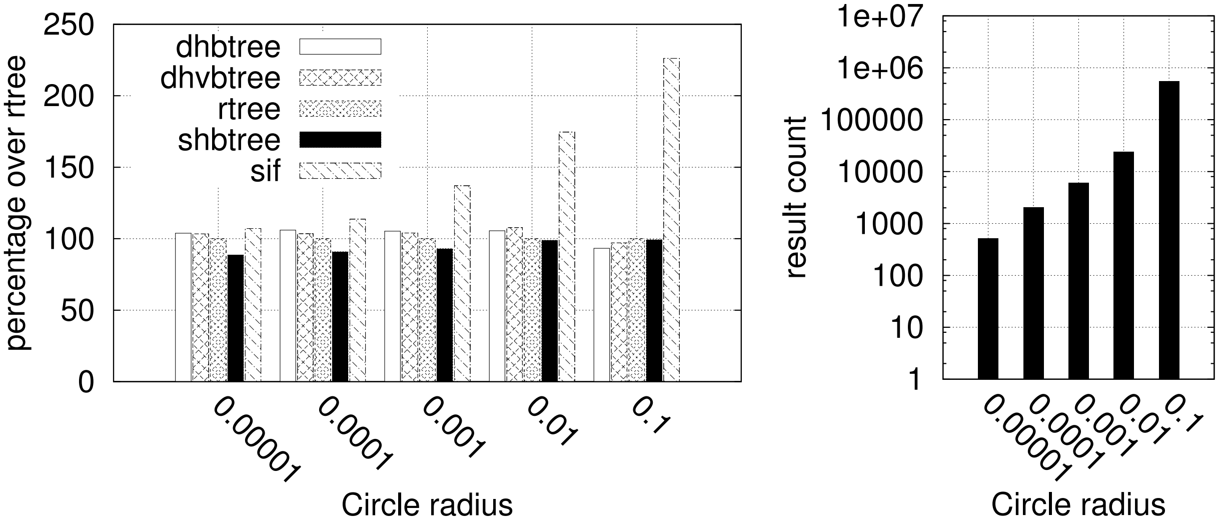
The SHB-tree index has an advantage over the DHVB-tree index, as computing a cell number for a given point object in the SHB-tree index is simpler than computing a Hilbert sequence number in the DHVB-tree index. In the SHB-tree index, the number of child-cells of a cell at a level is pre-defined when the index is created, and it is also small enough to create a static mapping table that is a two-dimensional array; each entry in the array is a pre-computed Hilbert sequence number. The x/y coordinates of a point object identifies an entry by using the coordinates as the array indexes of the entry. Thus, instead of going through a series of bit comparisons using the Hilbert state diagram as the DHVB-tree index does, the SHB-tree index can simply use the mapping table to look up a Hilbert sequence number at each level and repeat the lookups as many times as the number of levels.

4.4.1.2 Index-Only-Scan Select Query

The results of the index-only-scan select queries described at Step 4 in the static workload are discussed next.

Circle radius Query type	0.00001	0.0001	0.001	0.01	0.1
Index-only-scan select	97	94	94	101	297
Index-only-scan join	902	963	1000	1528	n/a
Non-index-only-scan select	193	239	429	1439	4981
Non-index-only-scan join	4316	7172	11268	11997	n/a

Table 4.4: R-tree's query response time (in milliseconds)



(a) Select response time % over R-tree

(b) Select result count

Figure 4.4: Static workload's *index-only-scan* select query's response time (as percentage over R-tree) and result count

Figure 4.4(a) shows, for each type of spatial index, the query-response-time percentage over the query response time of the R-tree index. The corresponding R-tree index's query response times are shown in Table 4.4. Figure 4.4(b) shows the queries' result counts. Also, the profile information such as the false-positive ratio, operators' elapsed times, and cache-miss count is shown in Figures 4.5–4.7, respectively. The false-positive ratio shown in Figure

4.5 represents $\frac{A}{B}$, where B is the number of qualified records from a spatial index search and A is the number of records discarded from the set of qualified records. Figure 4.6 details the elapsed times spent on each query runtime operator[3] involved in queries with different circle radii; the timings for query compilation and remote query launching (which took around 50 milliseconds together) are not shown. In the figures, the spatial index search operator is denoted as `SIDX_SEARCH`. The assign operator appearing on the SIF index represents the time spent on obtaining the set of cell numbers for the given query region's MBR. `TXN_JOB_COMMIT` represents lock-releasing overhead when a query is over. Figure 4.7 shows the cache-miss counts (from AsterixDB's buffer cache) caused mainly by the spatial-index accesses.

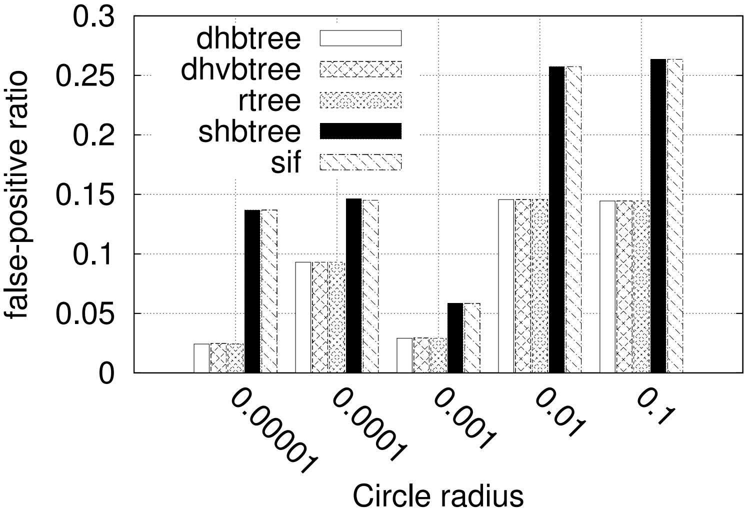


Figure 4.5: Static workload's *index-only-scan* select query's false-positive ratio

Overall, regardless of the radius size, all indexes except SIF provide similar query response times, though the SHB-tree index is around 10 to 15 percent faster for the small circle radii, 0.00001 to 0.001. The SHB-tree index's smaller response times for small radii are mainly due to the smaller cache-miss counts, which can be attributed to its simpler index-range-scan mechanism than the DHB-tree, DHVB-tree, and R-tree indexes (as described in Section 4.2) even though the SHB-tree index size is bigger. Also, a traditional R-tree index search

[3] We mention only important operators here regarding query response times. See [18, 8] for more detailed descriptions of query plans and the runtime operators involved in AsterixDB query plans.

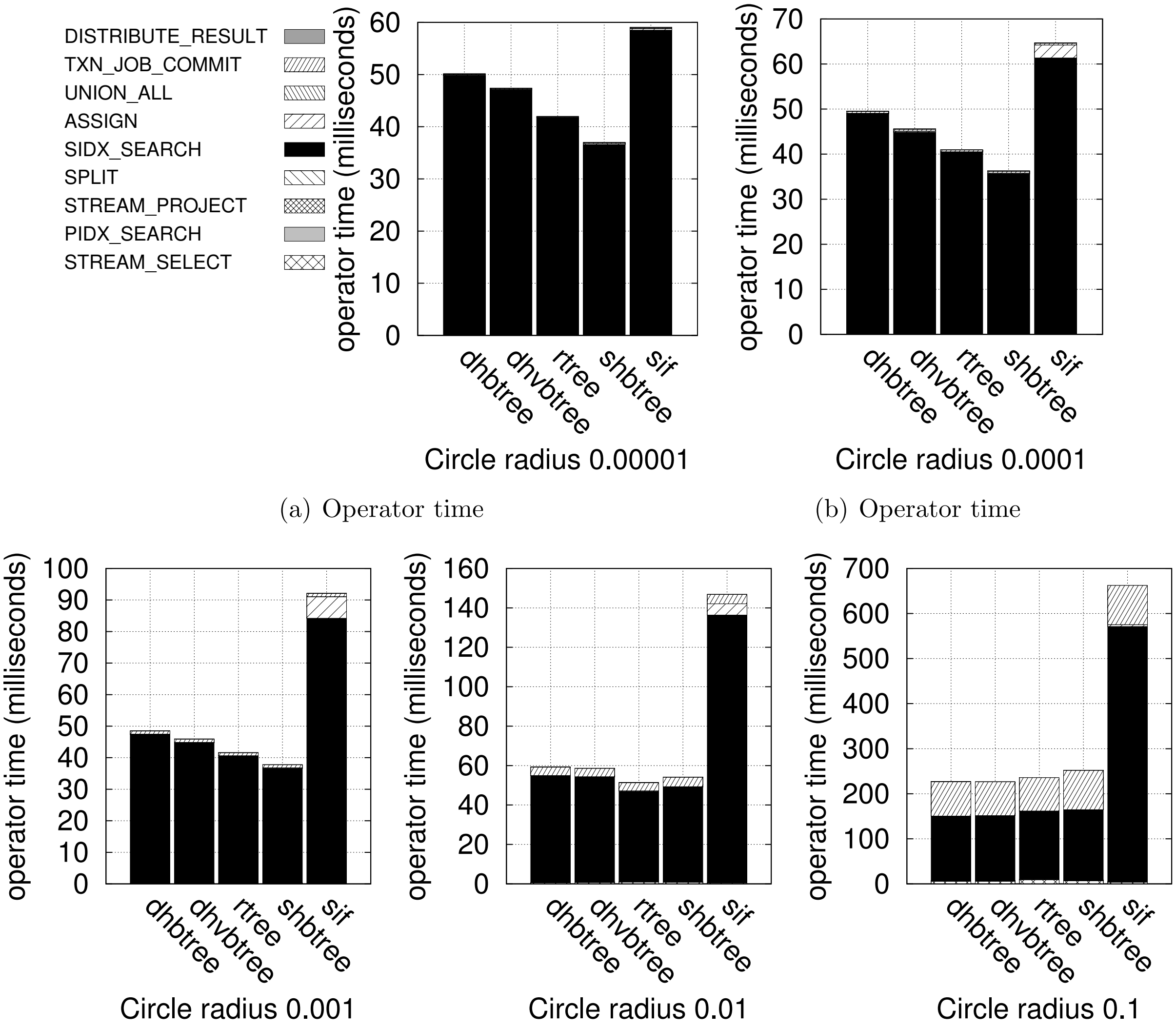
involves scanning all of the entries in each node during traversals down the tree and checking whether these entries overlap with the query region's MBR. In contrast, a more efficient binary search is available during a traditional B-tree index search. These SHB-tree index advantages are washed out for large circle radii, 0.01 and 0.1.

One reason for the wash-out effect for larger radii is that a large query region's MBR is partitioned into a larger number of cells, so more searches are required. Another reason is that there is more overhead related to lock and unlock operations for more false-positive entries, especially when result counts are larger. As shown in Figure 4.6(e), the unlocking overhead (`TXN_JOB_COMMIT`) becomes significant when the result count gets larger due to the large radius. The cell-based indexes, the SHB-tree and SIF indexes, generate more false positives caused by the cells at the bottom level of the grid hierarchy as described in Section 4.2.2.

Lastly, the SIF index shows the worst performance regardless of the radius size. This is due to its lack of range-search support, which causes more exact-match searches, and the lack of prefix-search support, which requires storing a point object at each level of the grid hierarchy as described in Section 4.2.3. Nonetheless, the SIF index can provide a way to support a spatial indexing method when only an inverted index is available in a given system.

4.4.1.3 Index-Only-Scan Join Query

The query-response-time percentages over the R-tree index and the result counts for the index-only-scan join queries described at Step 5 are shown in Figures 4.8(a) and 4.8(b), respectively. Also, the profile information such as the false-positive ratio, operators' elapsed times, and cache-miss count is shown in Figures 4.9–4.11, respectively. In the join operations, the outer table's (QuerySeedTweet) primary-index-search operator is denoted as `OUTER_PIDX_SEARCH` and the inner table's (Tweet) primary and secondary indexes are



(a) Operator time

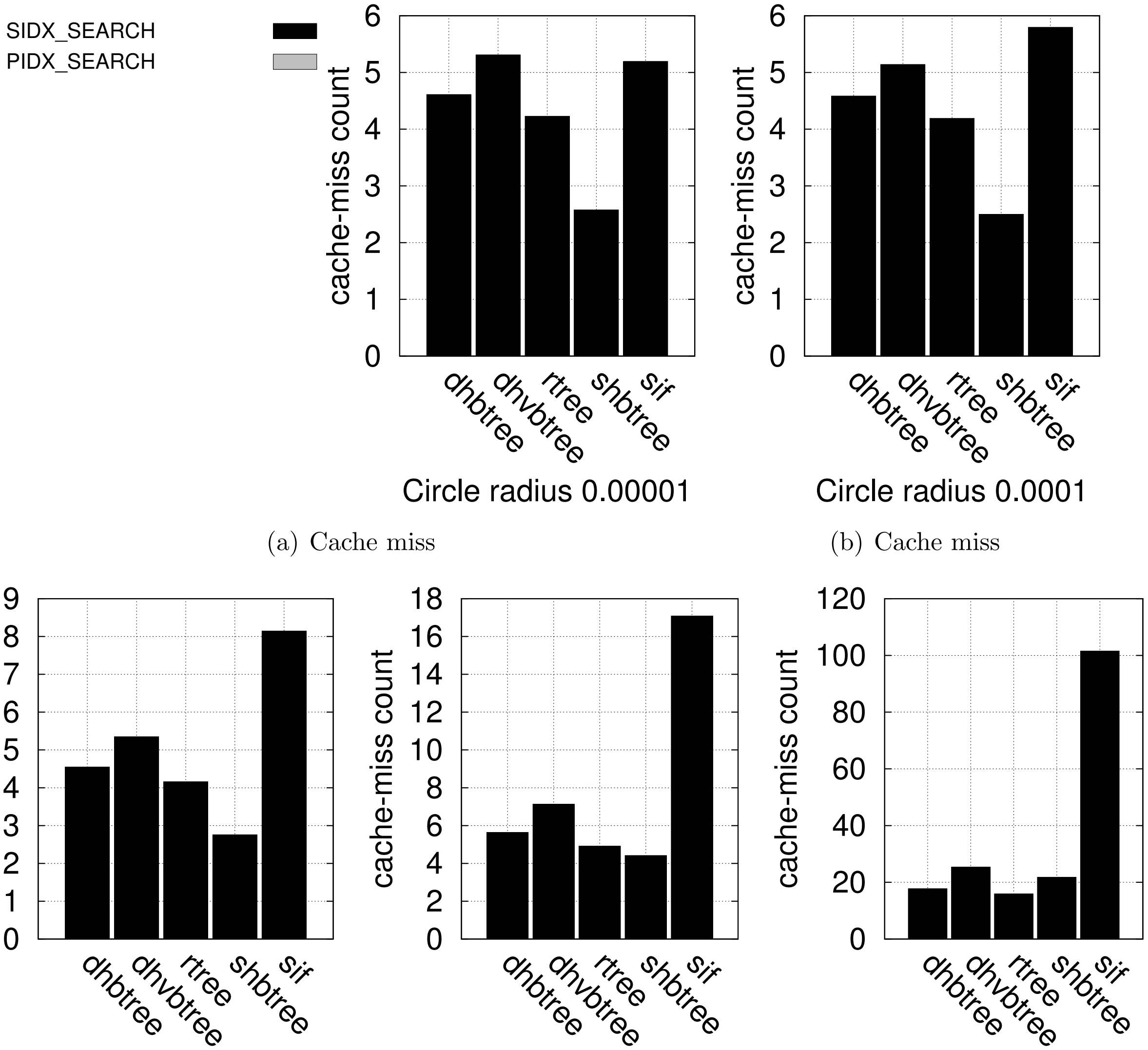
(b) Operator time

(c) Operator time

(d) Operator time

(e) Operator time

Figure 4.6: Static workload's *index-only-scan* select query's operator time



(a) Cache miss

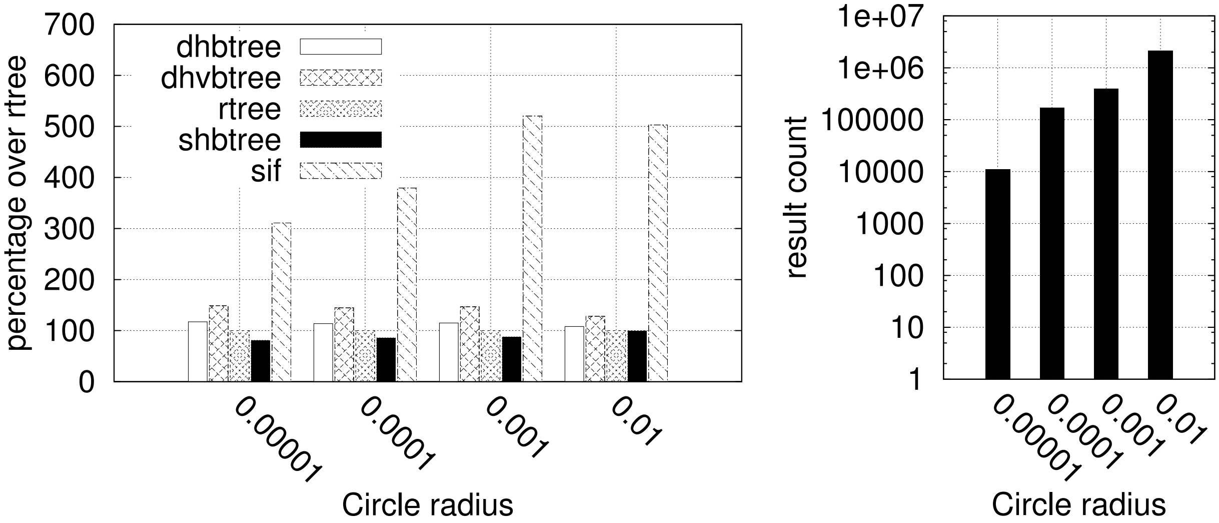
(b) Cache miss

(c) Cache miss

(d) Cache miss

(e) Cache miss

Figure 4.7: Static workload's *index-only-scan* select query's cache-miss count



(a) Join response time % over R-tree

(b) Join result count

Figure 4.8: Static workload's *index-only-scan* join query's response time (as percentage over R-tree) and result count

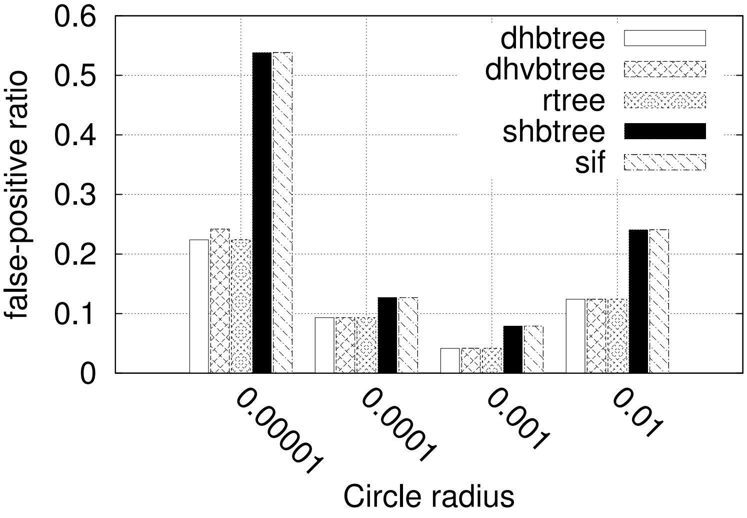
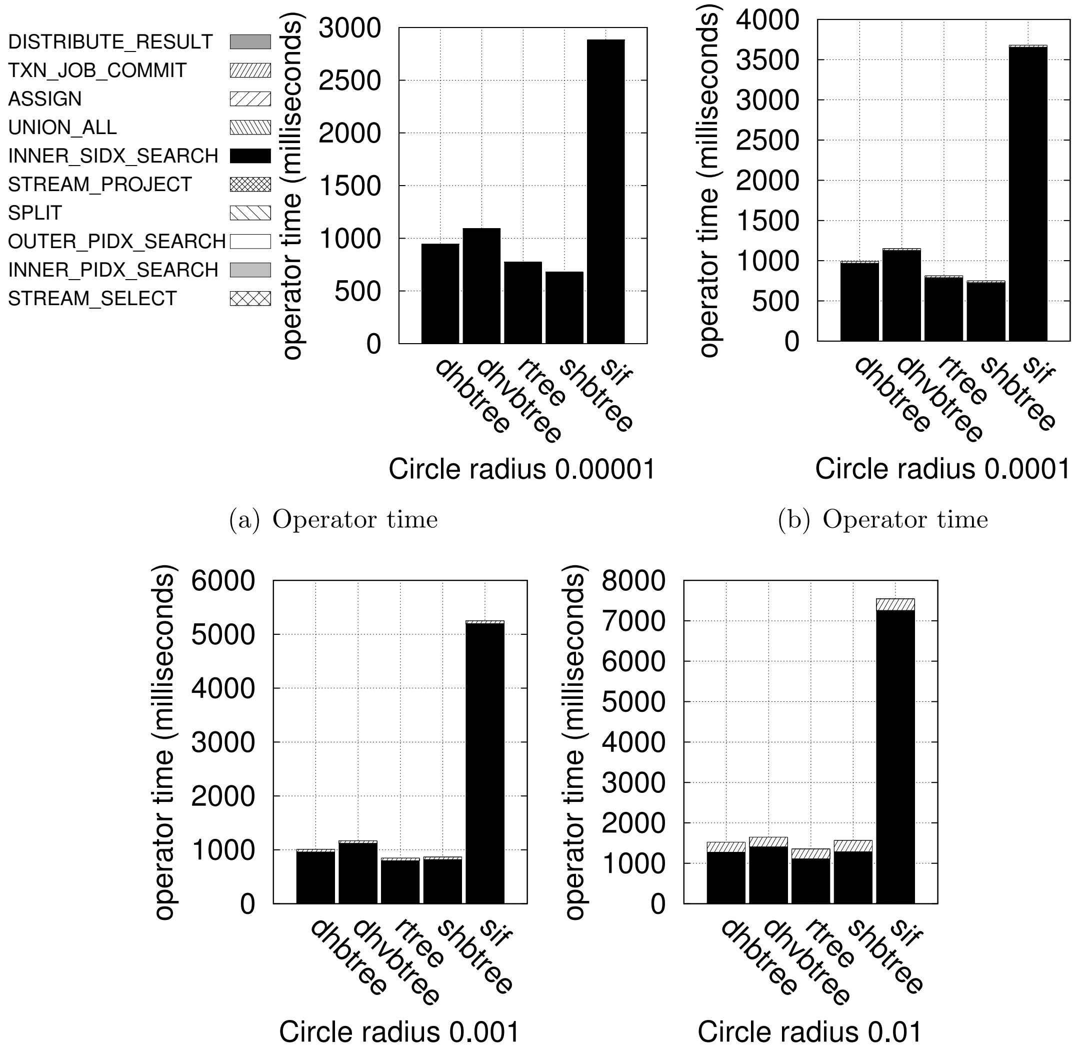


Figure 4.9: Static workload's *index-only-scan* join query's false-positive ratio

denoted as INNER_PIDX_SEARCH and INNER_SIDX_SEARCH, respectively.



(a) Operator time

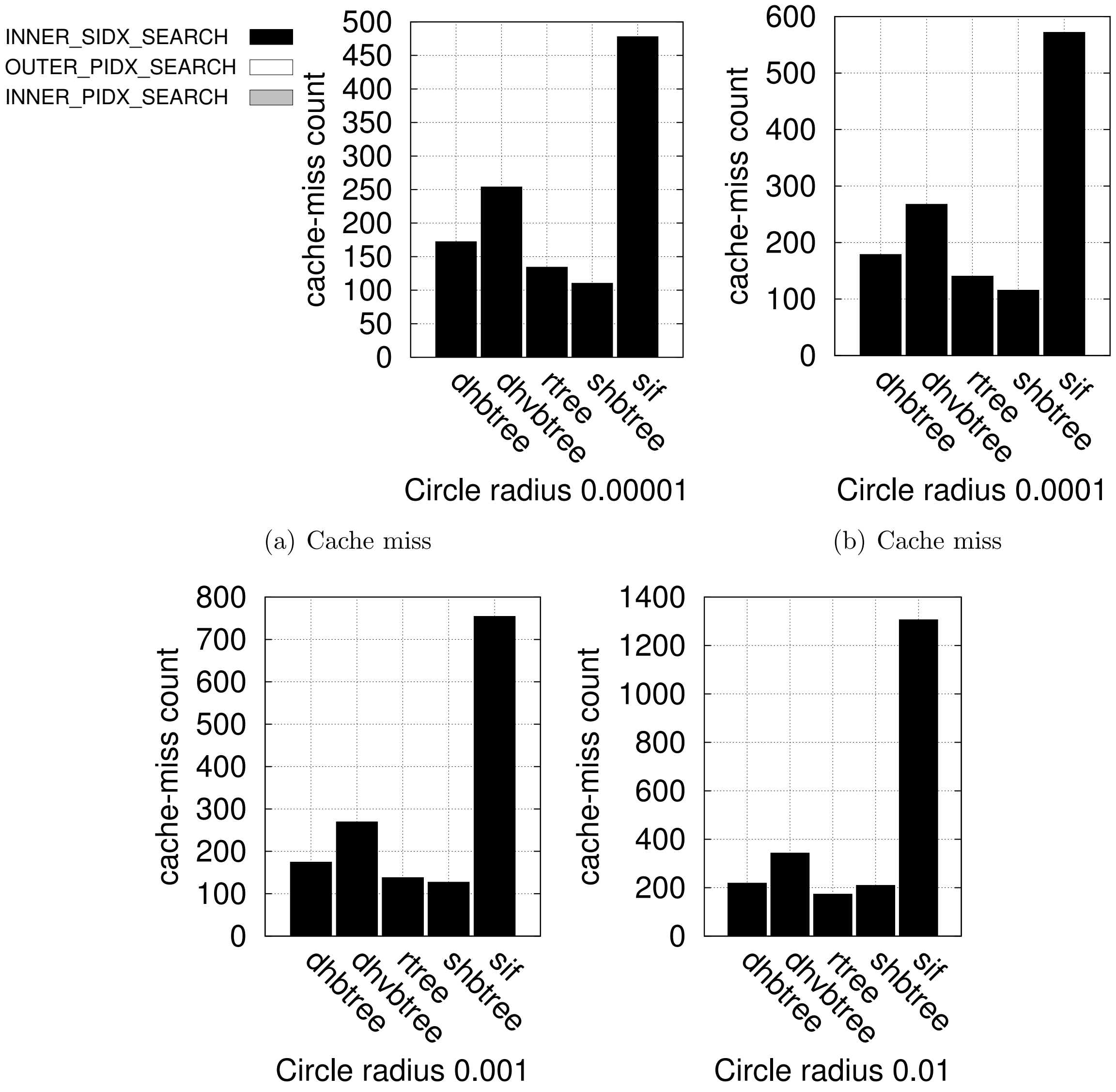
(b) Operator time

(c) Operator time

(d) Operator time

Figure 4.10: Static workload's *index-only-scan* join query's operator time

Overall, these results show a trend similar to the index-only-scan select-query results. Since a join query essentially leads to 100 range queries from the index-search's perspective, and the compilation and remote-query-launching overhead is relatively much smaller for a join query than a shorter select query, the performance gaps seen among the indexes for the select queries are magnified for the join queries. Consequently, the query-response-time differences caused by cache misses become more prominent. Lastly, the outer table having only 10,000 records allows it to be cached completely, so its corresponding primary-index-search operator does not cause any significant cache misses.



(a) Cache miss

(b) Cache miss

(c) Cache miss

(d) Cache miss

Figure 4.11: Static workload's *index-only-scan* join query's cache-miss count

4.4.1.4 Non-Index-Only-Scan Select Query

Figure 4.12 shows the query-response-time percentages over the R-tree index. Also, the profile information such as the operators' elapsed times and cache-miss count is shown in

Figures 4.13 and 4.14, respectively, where the black bars and gray bars represent the measured values for the secondary spatial index and the primary index, respectively. We omit the result counts and the false-positive ratios here since they are exactly the same as for the index-only-scan select-query case.

In general, for the non-index-only-scan queries, the primary-index-search cost becomes the dominant factor as the radius gets larger. Also, due to the use of instant locks, there is no lock-release overhead for the non-index-only-scan queries. It is notable that as the circle radius gets larger, the SHB-tree and SIF indexes' query response times become larger as shown in Figure 4.12. This is due to their larger numbers of false positives, which cause more I/Os due to reading more primary-index pages. This overhead is seen in the operator-time chart shown in Figure 4.13(e) and the cache-miss-count chart shown in Figure 4.14(e).

4.4.1.5 Non-Index-Only-Scan Join Query

Figure 4.15 shows the results of the non-index-only-scan join query in terms of the query-response-time percentages over the R-tree index. Also, the profile information such as the

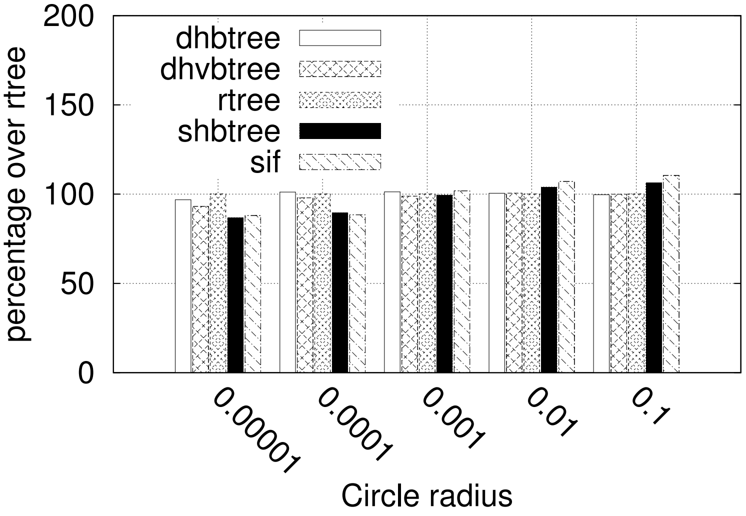
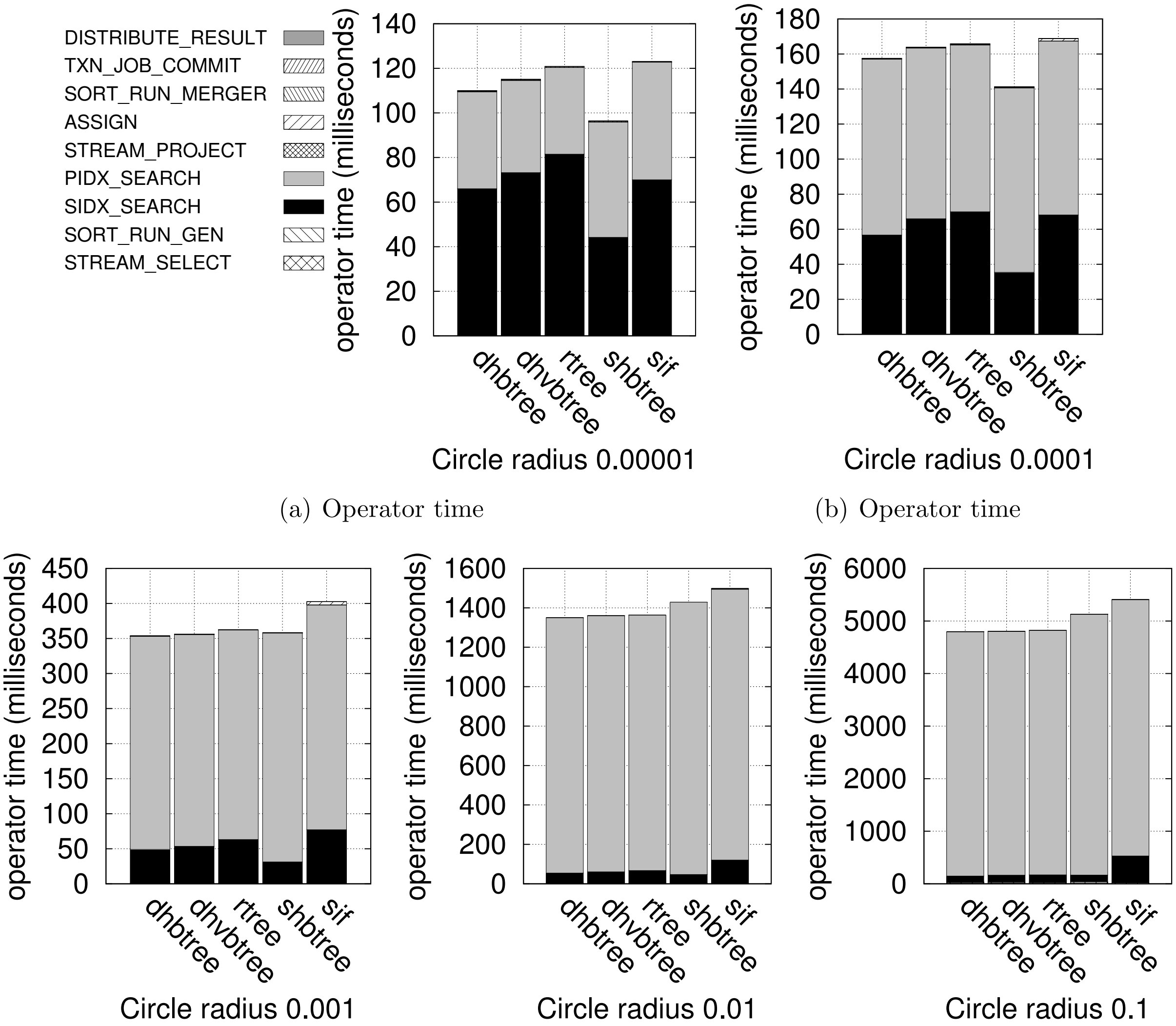


Figure 4.12: Static workload's *non-index-only-scan* select query's response time (as percentage over R-tree)



(a) Operator time

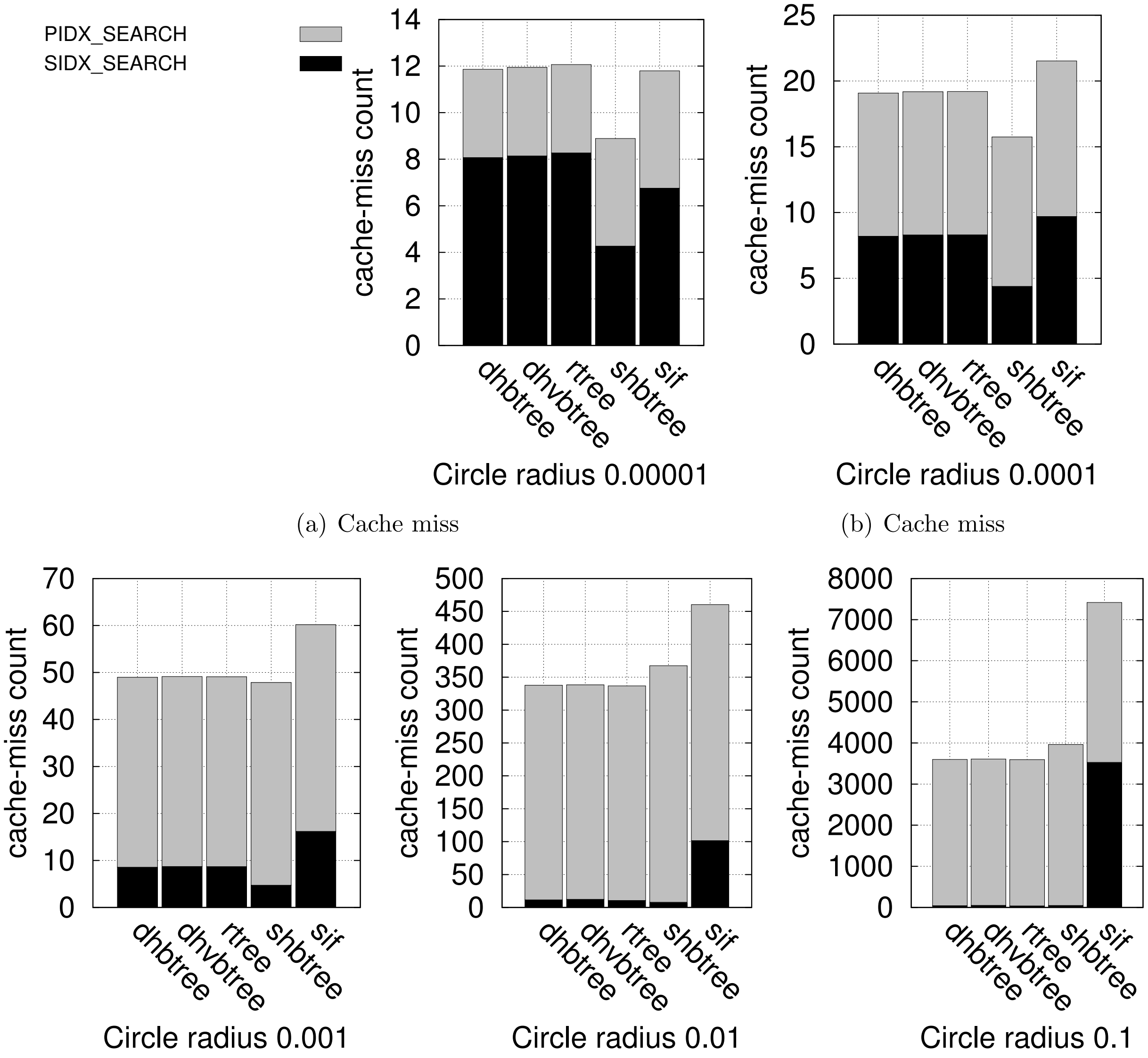
(b) Operator time

(c) Operator time

(d) Operator time

(e) Operator time

Figure 4.13: Static workload's *non-index-only-scan* select query's operator time



(a) Cache miss

(b) Cache miss

(c) Cache miss

(d) Cache miss

(e) Cache miss

Figure 4.14: Static workload's *non-index-only-scan* select query's cache-miss count

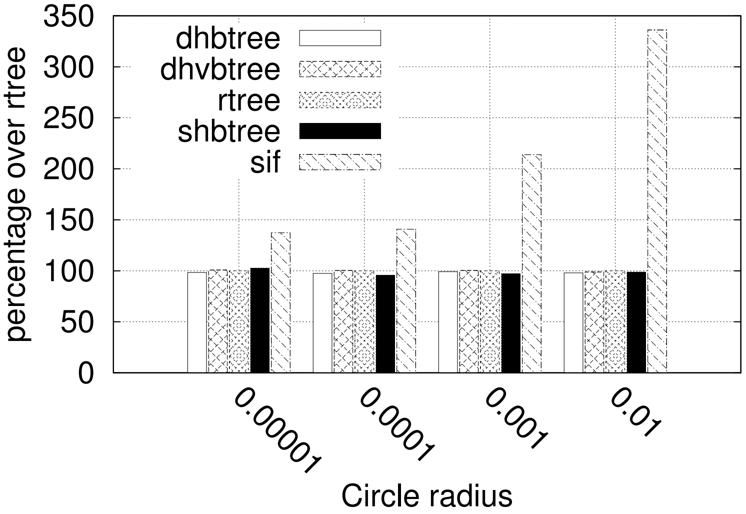


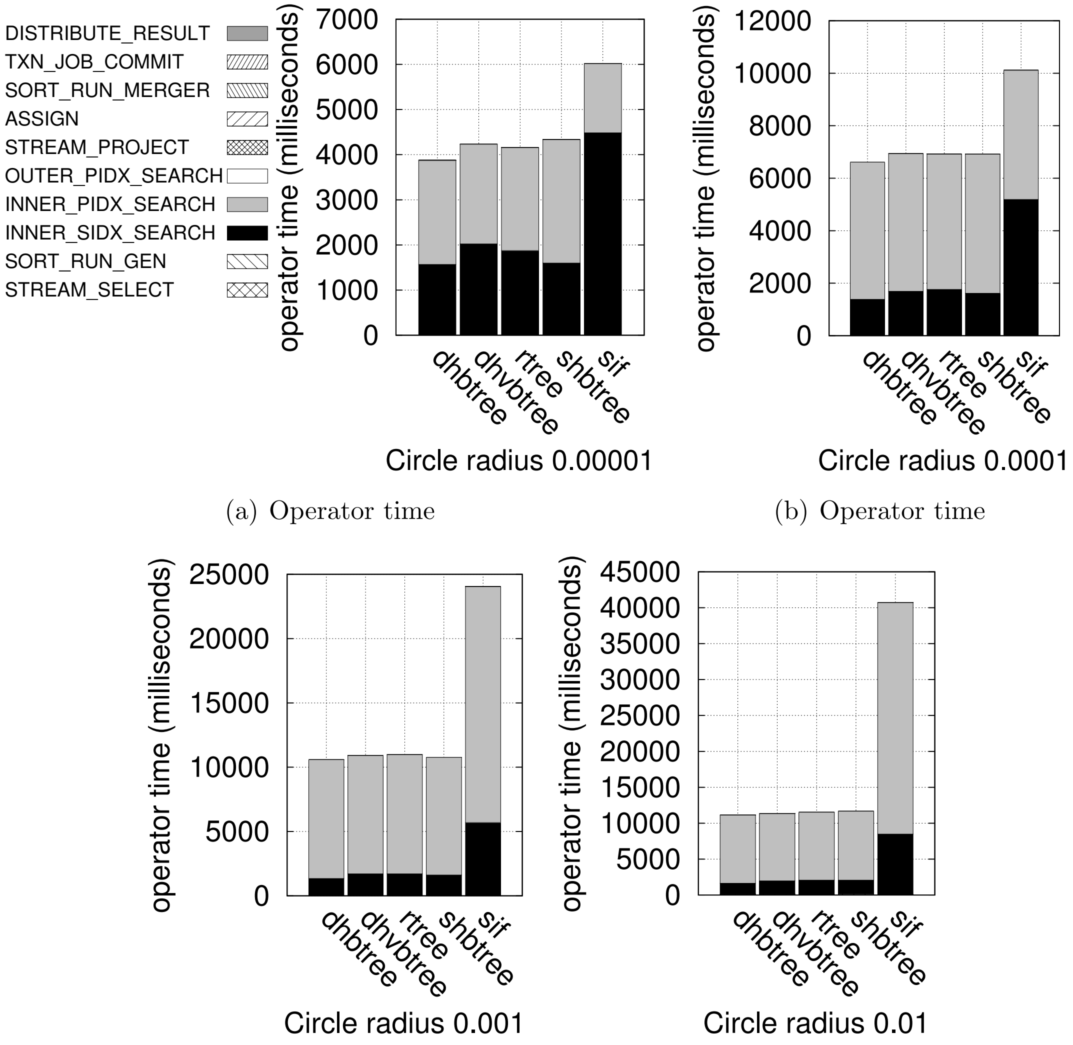
Figure 4.15: Static workload's *non-index-only-scan* join query's response time (as percentage over R-tree)

operators' elapsed times and cache-miss count is shown in Figures 4.16 and 4.17, respectively, where the black and gray bars represent the inner table's secondary spatial index and the inner table's primary index, respectively. The trend in these charts is very similar to those seen in the non-index-only-scan select-query case.

4.4.2 Results of Dynamic Workload 1

Dynamic workload 1 tests the scalability of each type of spatial index in terms of its IPS (inserts per second) performance for one hour of data ingestion. The results are shown in Figure 4.18. Figure 4.18(a) shows the IPS while varying the number of nodes in the cluster. As the number of nodes increases, the overall IPS increases linearly for all the indexes. The primary and secondary index sizes resulting from one hour of ingestion are shown in Figures 4.18(b) and 4.18(c), respectively. In addition, Figure 4.18(d) shows a series of current IPSs measured every five seconds during the one-hour ingestion from a single node of the cluster.

The DHVB-tree and SHB-tree indexes outperform the other indexes in terms of their IPS. The DHVB-tree index outperforms the DHB-tree index due to the DHB-tree index's relative-



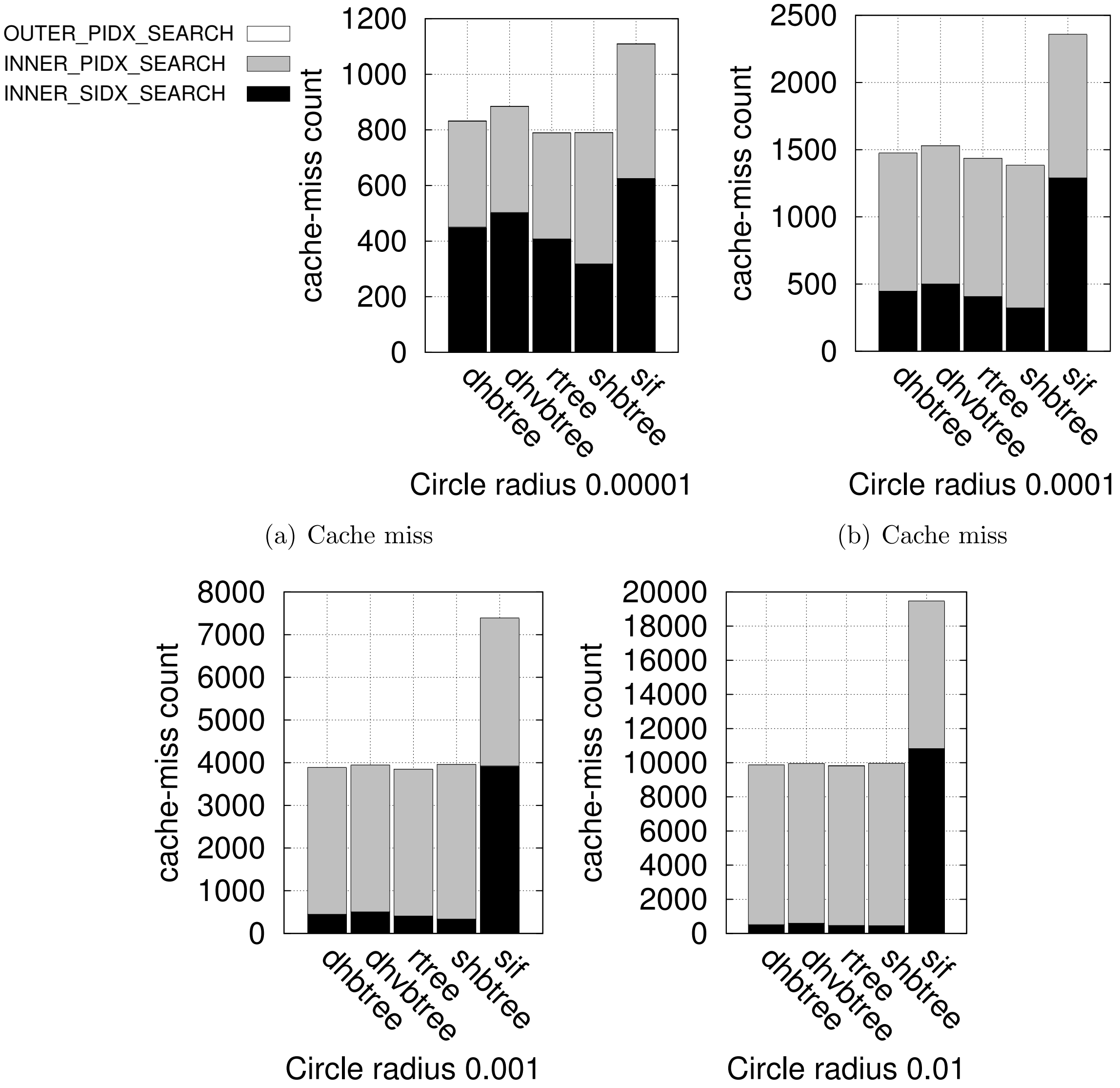
(a) Operator time

(b) Operator time

(c) Operator time

(d) Operator time

Figure 4.16: Static workload's *non-index-only-scan* join query's operator time



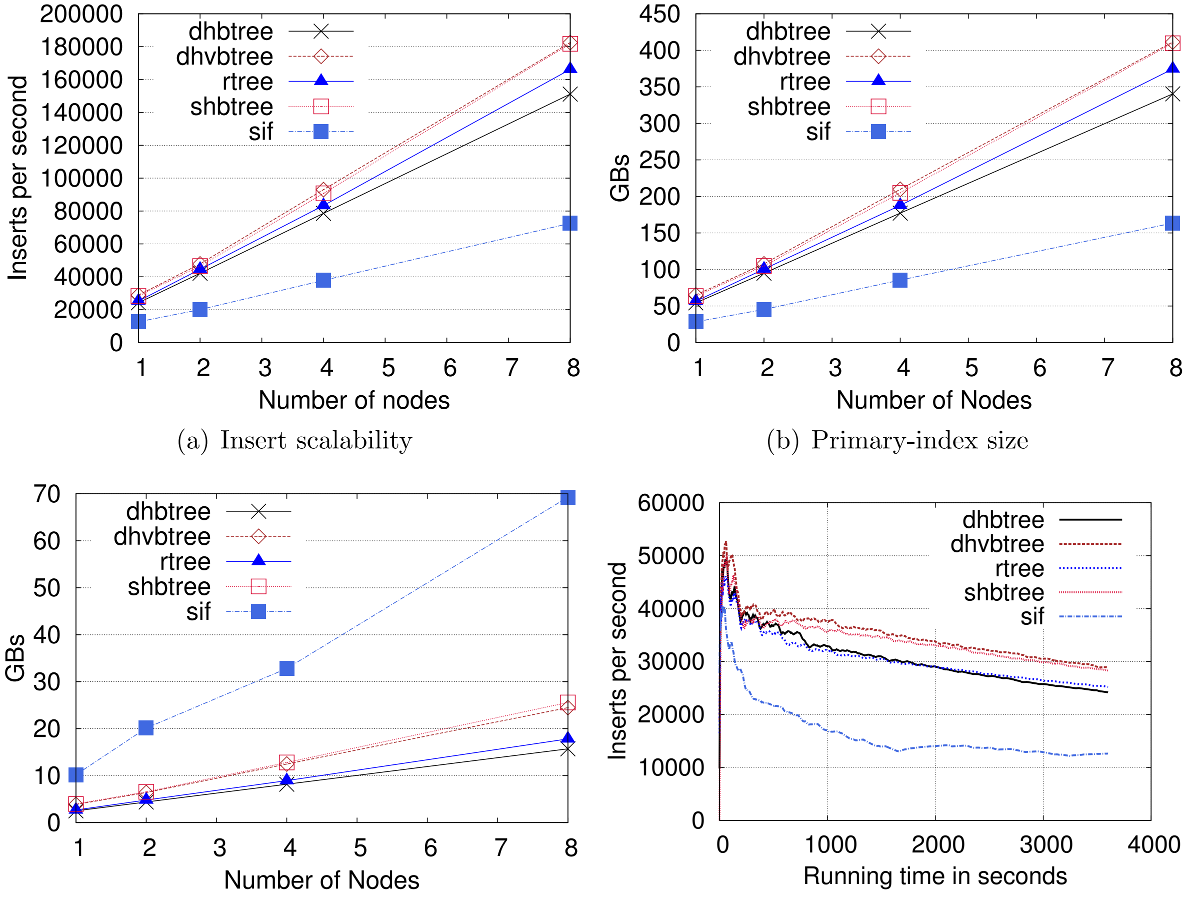
(a) Cache miss

(b) Cache miss

(c) Cache miss

(d) Cache miss

Figure 4.17: Static workload's *non-index-only-scan* join query's cache-miss count



(a) Insert scalability

(b) Primary-index size

(c) Spatial-index size

(d) IPS every five seconds

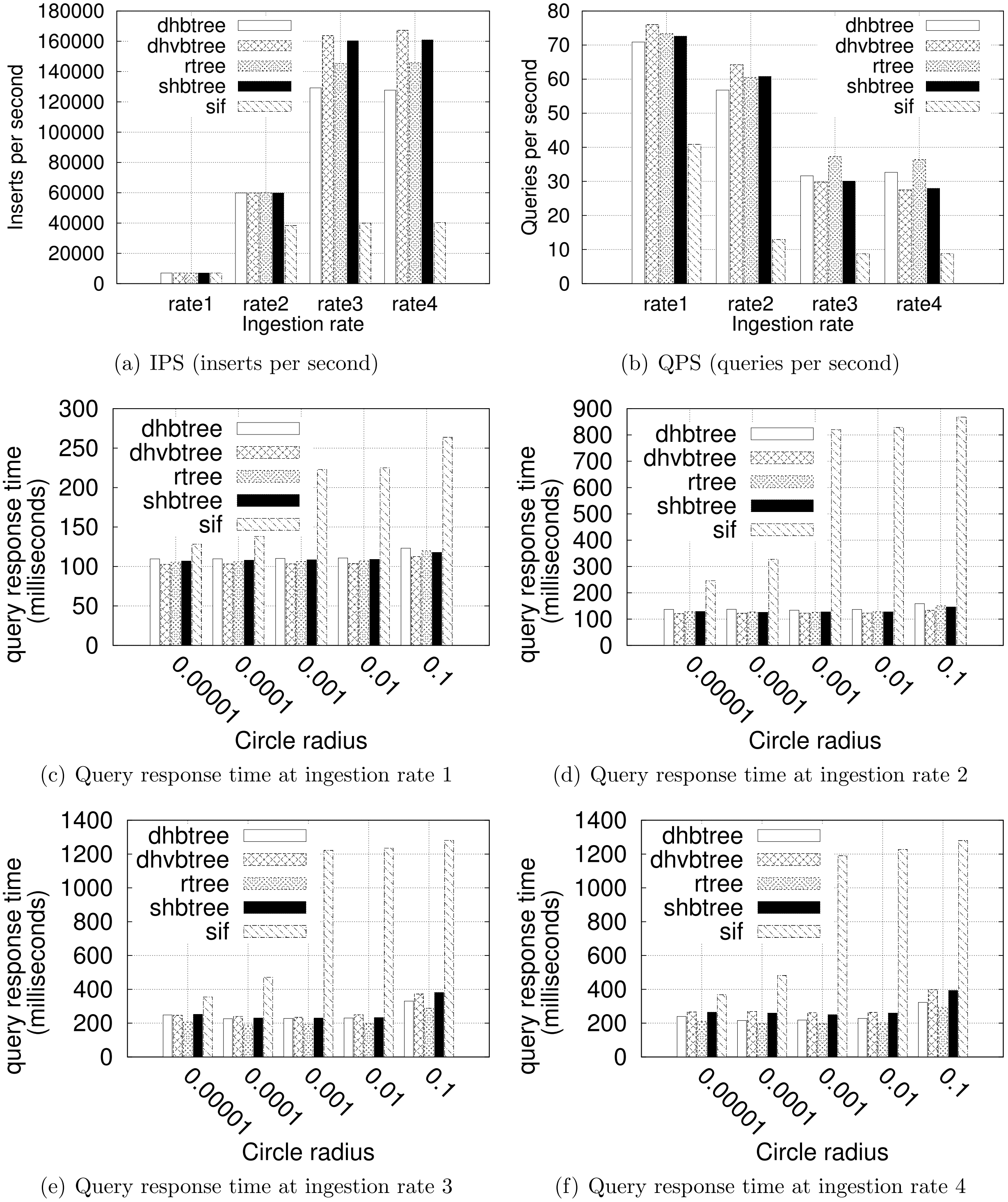
Figure 4.18: Dynamic workload 1

comparison overhead when inserting entries into the DHB-tree index. It also outperforms the R-tree index due to the R-tree index's relative-comparison overhead while flushing the in-memory component's entries, where the process of flushing consists of Hilbert-sorting and bulk-loading the entries. Also, the R-tree index's incremental insert logic includes more overhead than that of a traditional B-tree index due to the effort for forming better MBRs in internal nodes. Again, SIF is the slowest of the indexes due to the toll of six inserts for each point object, one at each level. Figure 4.18(d) reflects the impact of LSM merge operations. In general, the merge-operation overhead increases as time goes by since the components' sizes increase until they exceed the threshold M of the prefix merge policy.

4.4.3 Results of Dynamic Workload 2

Dynamic workload 2 studies each index type's ability to ingest and query concurrently while varying the ingestion rate. Figure 4.19 shows the results of the concurrent index-only-scan query case for this workload, where rate 1, rate 2, rate 3, and rate 4 represent making tweet generators sleep for 1 millisecond after every 1, 10, 100, and 1000 generated record(s), respectively. Figure 4.19(a) shows the IPS for each spatial index for each ingestion rate after one hour of ingestion while concurrent queries are being processed. Figure 4.19(b) shows the corresponding QPS (queries per second). Figures 4.20(c)–4.20(f) show the average query response times for each ingestion rate.

First, as the offered ingestion rate goes up, IPS also goes up, but it begins to saturate at rate 3 and goes up just slightly at rate 4 for all indexes except the SIF index. (Data that is not ingested is discarded by setting the feed policy to discard excess data.) In contrast, the corresponding QPS is the highest at rate 1 and then goes down. This is due to the limited resources in the cluster being shared by both the ingestion and query workloads. Overall, the DHVB-tree index shows the best IPS at the all offered ingestion rates. It also shows the



(a) IPS (inserts per second)

(b) QPS (queries per second)

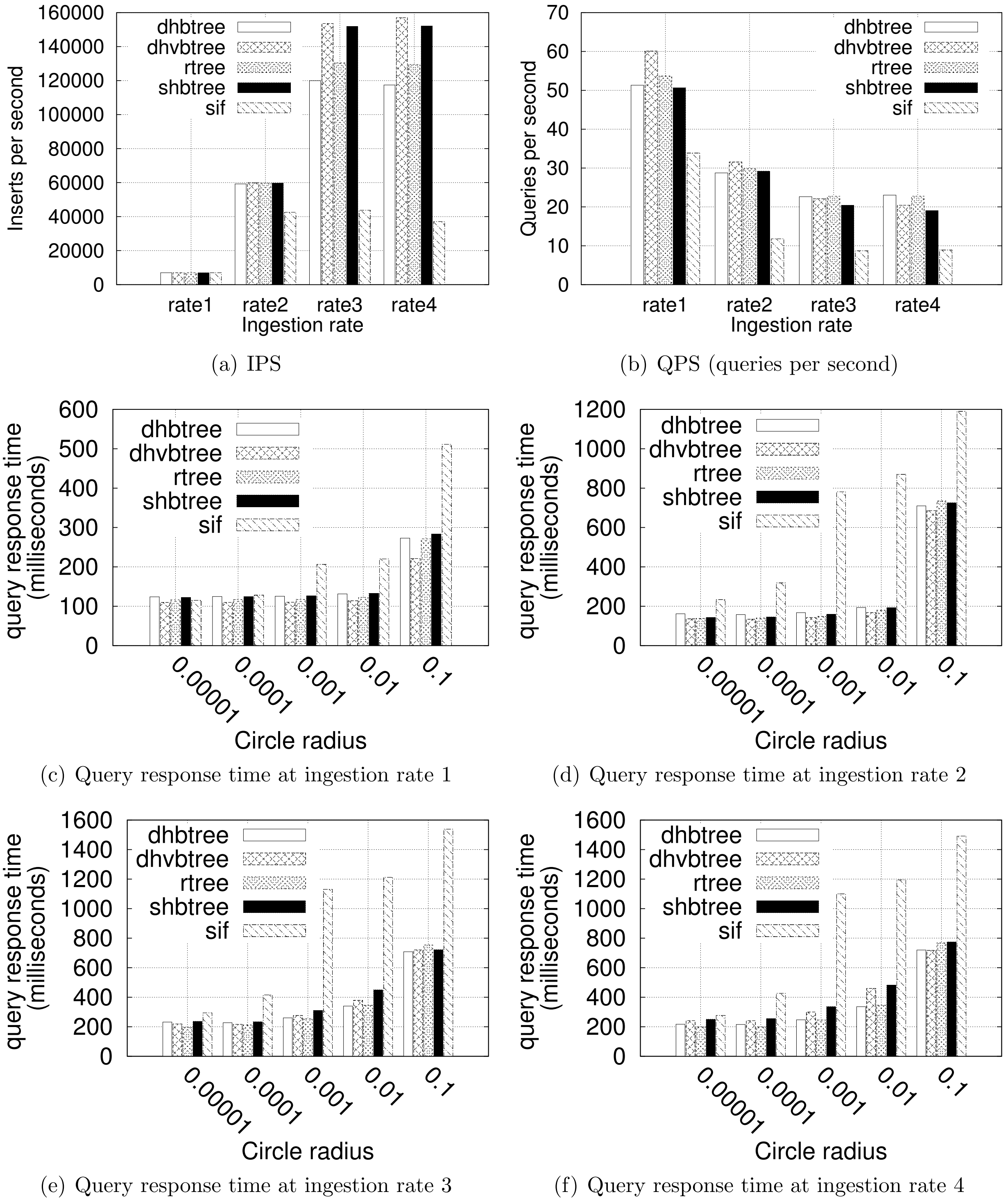
(c) Query response time at ingestion rate 1

(d) Query response time at ingestion rate 2

(e) Query response time at ingestion rate 3

(f) Query response time at ingestion rate 4

Figure 4.19: Results of dynamic workload 2 for the *index-only-scan* query case, where the rate 1, rate 2, rate 3, and rate 4 represent making tweet generators sleep for 1 millisecond after every 1, 10, 100, and 1000 generated record(s), respectively.



(a) IPS

(b) QPS (queries per second)

(c) Query response time at ingestion rate 1

(d) Query response time at ingestion rate 2

(e) Query response time at ingestion rate 3

(f) Query response time at ingestion rate 4

Figure 4.20: Results of dynamic workload 2 for the *non-index-only-scan* query case, where the rate 1, rate 2, rate 3, and rate 4 represent making tweet generators sleep for 1 millisecond after every 1, 10, 100, and 1000 generated record(s), respectively.

best QPS for the slower ingestion rates, while the R-tree index shows the best QPS for the higher ingestion rates.

As shown in Figure 4.20, the results of the concurrent non-index-only-scan query case for dynamic workload 2 shows the same overall trend as the index-only-scan query case. The difference here is that the measured IPSs and QPSs are smaller and the gaps in the QPS chart are smaller, and the overall query response times are larger. This is mainly due to the added primary index search overhead.

4.4.4 Effect of Varying Memory Budget

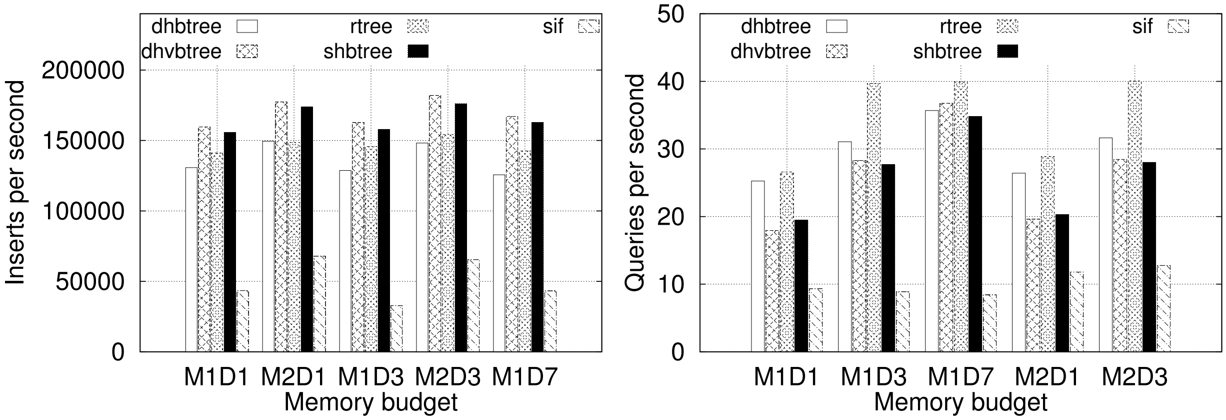
For completeness, we conducted more experiments by varying the per-node memory budget of the in-memory components and the disk buffer cache size in order to see the effects of changing these memory budgets on the ingestion and query performance of the compared spatial indexes. At the same time, we analyzed additional LSM index metrics such as the number of flushes, merges, disk components, and write amplification [60].

Settings	Memory budget for in-memory components per dataset (in GB)	Memory budget for disk buffer cache size (in GB)
M1D1	1	1
M1D3	1	3
M1D7	1	7
M2D1	2	1
M2D3	2	3

Table 4.5: Memory budget settings for in-memory components per dataset (the number after "M") and disk buffer cache size (the number after "D") in GB

As shown in Table 4.5, we conducted experiments using 5 different pairs of memory budgets for in-memory components and disk buffer cache sizes. For instance, the configuration denoted as "M1D3" represents a 1GB budget for the in-memory components per dataset and 3GB for the disk buffer cache size. M1D3 was the default setup used in the previous experiments.

Figure 4.21 shows the result of varying the memory budget on the concurrent ingestion and query performance using dynamic workload 2. The offered ingestion rate was ingestion rate 3, where the tweet generators sleep for 1 millisecond after every 1000 tweets.



(a) IPS

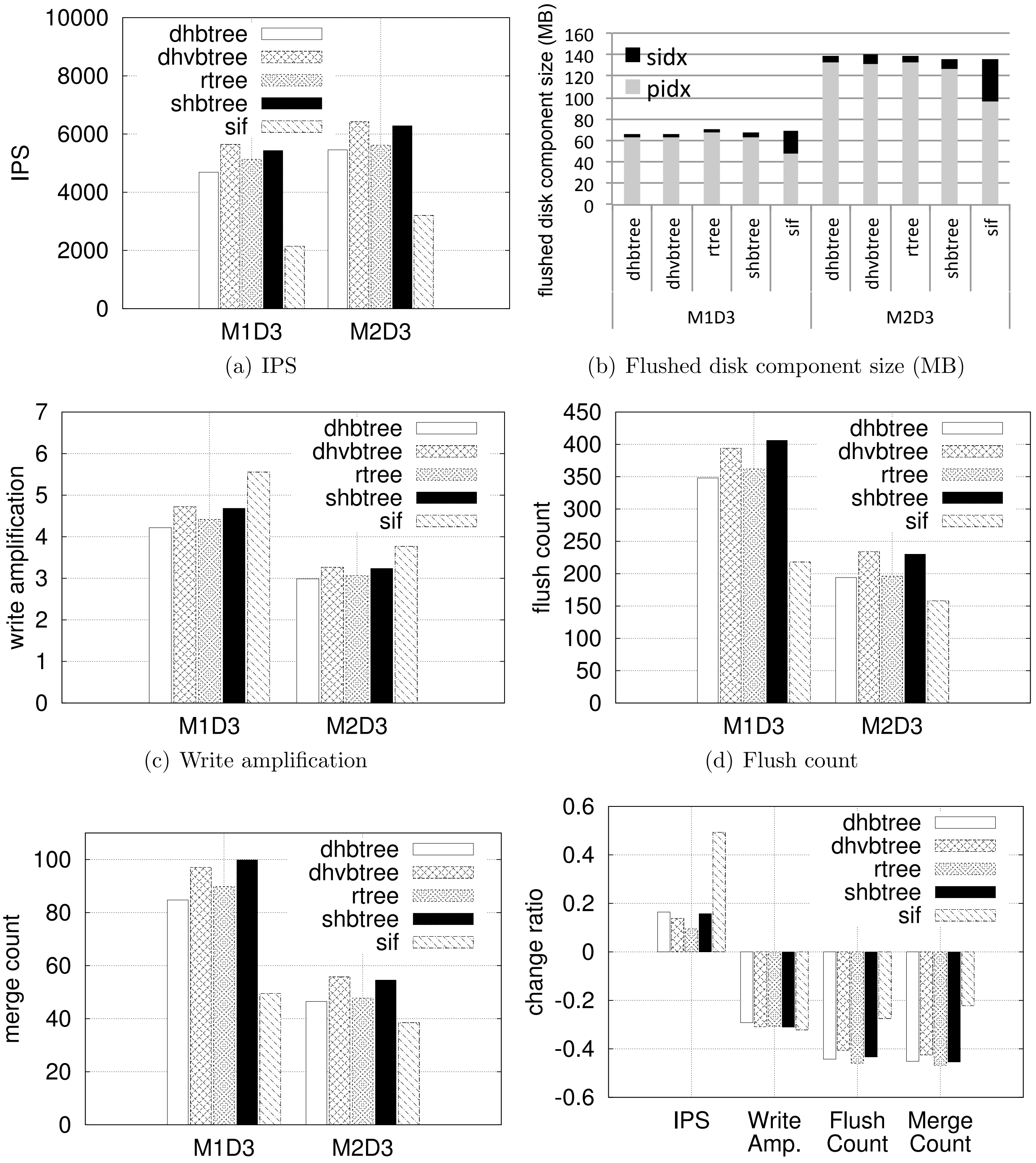
(b) QPS

Figure 4.21: Performance of concurrent ingestion and queries in terms of IPS and QPS while varying memory budgets

Overall, as the in-memory component's memory budget and disk buffer cache size were increased, the inserts per second (IPS) and queries per second (QPS) were improved, respectively. Ingestion performance is mainly affected by the in-memory component's memory budget, while query performance is mainly affected by the disk buffer cache size, as one would expect, with bigger being better in each case. The degree of the effect varies from one underlying index structure to another.

We have also analyzed LSM index metrics such as the number of flushes, merges, and disk components, and also *write amplification* (to be explained shortly), in order to explain the relationship between these metrics and ingestion performance. We used the ingestion-only workload (dynamic workload 1) while changing the memory budget of the in-memory components from M1D3 to M2D3.

Figure 4.22 shows the results, including the corresponding LSM index metrics. All numbers in the charts are the average values for a single partition (among 32 partitions throughout



(a) IPS

(b) Flushed disk component size (MB)

(c) Write amplification

(d) Flush count

(e) Merge count

(f) Change ratio

Figure 4.22: IPS and profiled LSM index metrics from a single partition

the 8-node cluster) as opposed to aggregated total numbers from all partitions.

Figure 4.22(a) shows the IPS after one hour of ingestion. The pattern of the results is almost the same as that seen in Figure 4.21(a) except that the y-scale is based on a single partition. Figure 4.22(b) shows the (stacked) sizes of the flushed disk components for each secondary index and its associated primary index. Note that flushed disk components are components that have not been merged with other disk components yet. Thus, their sizes are determined by the configured parameter, i.e., the memory budget for in-memory components per dataset. For example, for the M1D3 case, the allocated 1GB budget is shared by 4 pairs of primary and secondary index memory components from the 4 partitions in a node, with one pair per partition. Also, when the in-memory component's capacity is reached and it is thus being flushed, in order to prevent ingestion from stalling, AsterixDB uses a "double-buffering" technique by having two in-memory components. In the end, for the M1D3 case, the sum of a primary and a secondary index's flushed component's size was around 64MB. However, depending on the secondary index type, the portions between the primary index and the secondary index vary. For the M2D3 case, the combined size was roughly doubled.

The *write amplification* [60] metric captures the cost of LSM writes as a ratio of the actual amount of data written to disk to the optimal amount of data to be written. For example, if 100 records are inserted and flushed to a disk component, and then the disk components are merged two times before the end of the experiment, the write amplification for these 100 records is roughly 3 since they were written 3 times (1 for flush, 2 for merges). Figure 4.22(c) shows the overall write amplification of the ingested records during the one hour of ingestion. In addition, Figures 4.22(d) and 4.22(e) show the corresponding flush and merge operations' occurrence counts. Each flush operation represents a new disk-component creation, but there are not as many components as the number of flush operations due to the merge operations. In our experiments, the prefix merge policy (described in Section 4.3.1) triggers a merge operation when there are 5 mergable disk components, where "mergable disk components" means that their sizes are each less than 1GB. Usually every 4 flush operations in each index's partition will trigger a merge operation since there is an existing mergable disk component created from the previous merge operation. This behavior is reflected in Figures 4.22(d) and 4.22(e).

In summary, Figure 4.22(f) shows the change ratio of the aforementioned metrics when moving from M1D3 to M2D3. Overall, as the memory budget increases from 1GB to 2GB, all IPS increase and the write amplification and flush/merge counts decrease. Even though IPS increased, the IPS-change ratio varies depending on the index type, and it is less than the absolute change ratio of those LSM metrics. This is because there are still other factors impacting the IPS in this experimental setup. For example, in a given node, having 4 data-storage partitions and 1 transaction-log-storage partition over a RAID-0 logical disk consisting of 3 physical disks may blur the impact of the write amplification of the ingested data since there could be more contention in using the physical disks among concurrent threads running on those logical partitions. The size of the transaction logs during ingestion is another aspect that is essentially orthogonal to the memory-budget increase. Also, the read cost during the merge operation is another important factor.

4.4.5 Summary of the Results

An overall summary of the results of this chapter is captured in Figure 4.23. The figure shows each index's rank for each experimental case including its percentage of the corresponding measure over the baseline R-tree index. For example, for the ingestion-only case, the "dhvb-tree(110)" entry represents that the DHVB-tree index is ranked in first place in terms of IPS, where "110" is its percentage over the IPS of the R-tree index, i.e., 10% more IPS than the R-tree index. Also, two indexes fall into the same rank if the difference between them is less than 1 percent. For the query-only cases, a percentage value represents an average of

rank	Loading		Query only				Ingestion only	Ingestion + Query							
	Index size	Index creation time	Index-only-scan-query response time		Non-index-only-scan query response time		IPS (max ingestion rate)	With index-only-scan query				With non-index-only-scan query			
								Slow ingestion rate (rate1 and rate2)		Fast ingestion rate (rate3 and rate4)		Slow ingestion rate (rate1 and rate2)		Fast ingestion rate (rate3 and rate4)	
			Select	Join	Select	Join		IPS	QPS	IPS	QPS	IPS	QPS	IPS	QPS
1	dhbtree (97)	rtree, shbtree	shbtree (94)	shbtree (88)	shbtree (97)	dhbtree (98)	dhvbtree (110)	rtree, dhbtree, dhvbtree, shbtree	dhvbtree (105)	dhvbtree (114)	rtree	rtree, dhbtree, dhvbtree, shbtree	dhvbtree (109)	dhvbtree (120)	rtree, dhbtree
2	rtree		rtree	rtree	dhvbtree (98)	shbtree (99)	shbtree (109)		rtree, shbtree	shbtree (110)	dhbtree (87)		rtree	shbtree (117)	
3	dhvbtree (125)	dhvbtree (106)	dhbtree, dhvbtree (103)	dbhtree (113)	sif(99)	rtree, dhvbtree	rtree			rtree	shbtree (79)		dhbtree, shbtree (96)	rtree	dhvbtree (93)
4	shbtree (131)	dhbtree (151)		dhvbtree (142)	rtree, dhbtree		dbhtree (91)		dhbtree (95)	dhbtree (88)	dhvbtree (78)			dhbtree (91)	shbtree (87)
5	sif(741)	sif(273)	sif(152)	sif(428)		sif(207)	sif(44)	sif(82)	sif(39)	sif(28)	sif(24)	sif(86)	sif(51)	sif(31)	sif(39)

Figure 4.23: Summary of the experimental results showing each index's rank for each experimental case including its percentage of the corresponding measure over the baseline R-tree index.

the percentages over all radius cases.

Overall, except for the SIF index, there is neither one clear winner nor a clear loser considering both the ingestion and query performance. The query-performance differences for the most part were not large in the setting of a real end-to-end system; this is especially true for large-circle non-index-only-scan queries, where the required primary-index lookups were costly. Nonetheless, if we had to pick a winning index based on the evaluation results with a focus on balancing concurrent ingestion and querying (since this is a very important workload in the Big Data era), the DHVB-tree index could be the winner if ingestion has a higher priority, and the R-tree index could be the winner if queries have a higher priority. We believe that these experimental results can be useful as a valuable guide for practitioners seeking to use or implement spatial indexing methods in their system considering their pros and cons.

4.5 Related Work

A comprehensive survey [31] covers a broad spectrum of multidimensional access methods by classifying them into two main categories, 1) point-access methods and 2) spatial-access methods, based on whether the indexed data are point or not. That survey also includes various comparative studies' results among those methods. Most of the studies, however, measured only I/Os or included CPU time only for index-access cost with static workloads, as opposed to measuring end-to-end performance using concurrent dynamic workloads based on a full-fledged system.

An experimental study from Oracle [42] compared a linear Quad-tree [32] and an R-tree to evaluate whether B-tree-based spatial indexes can be as effective as a specialized spatial tree, i.e., an R-tree index. Their answer was twofold. First, the R-tree was found to be better for searching since the Quadtree tended to return more false positives than the R-tree. Second, the Quadtree was found to be better for insert/delete operations since their B-tree-based linear Quadtree was simpler than the R-tree index, which suffers from the overhead involved in forming good MBRs. Our study's results from dynamic workloads 1 and 2, more specifically, our comparative results between the LSM-based R-tree and LSM-based SHB-tree indexes, are consistent with these results in terms of IPS, QPS, and the end-to-end query response time. However, the Oracle results are not consistent with the results of our static workload's index-only-scan query case; in our experiments, the SHB-tree index outperformed the R-tree index since the false-positive-processing cost (that did not involve accessing a primary index for the index-only-scan query) was not significant.

The MS SQL Server spatial index uses a B-tree for spatial indexing, but no comparative evaluation of alternatives was included in [30]. The Bkd-tree [56], an extension of the K-D-B-tree index [58], which made the kd-tree [12] disk-resident, maintains multiple disk kd-trees instead of one, similar to what the LSM-tree does, in order to provide I/O efficient update performance. Also, the partitioned exponential (PE) file [41] was designed for intense update loads and concurrent, analytic queries, and it can deal with multidimensional data. The work in [41] included evaluation results between the PE file and a variant of an R-tree. The PE file was shown to outperform the R-tree variant in terms of IPS and QPS. Lastly, our own AsterixDB storage management experiments [9] showed the superiority of an LSM-ified R-tree over a traditional in-place-update R-tree in terms of both insert and query performance due to the former's leveraging of the well organized MBRs resulting from a Hilbert curve based ordering of indexed point objects.

4.6 Conclusions

This study has explored a set of representative, disk-resident spatial indexing methods that have been adopted by major SQL and NoSQL systems. We implemented five alternative variants of these methods in the form of LSM-based spatial indexes in Apache AsterixDB—DHB-tree, DHVB-tree, R-tree, SHB-tree, and SIF—in order to evaluate their pros and cons in light of the dynamic characteristics of high-volume, geo-tagged point data. We have empirically examined the performance of these five spatial indexes in terms of their query-response time, inserts per second, and queries per second using three types of workloads: a query-only workload, an insert-only workload, and a concurrent insert-and-query workload. Based on the observed performance results, we discussed the pros and cons of these five indexes including the effect of their different underlying bases, i.e., their B-tree, R-tree, and inverted index foundations.

Overall, except for the SIF index, there is neither one clear winner nor a clear loser considering both ingestion and query performance. The query performance differences for the most part were not large in the setting of a real end-to-end system; this is especially true for large-circle non-index-only-scan queries, where primary index lookups to fetch the query result records were costly and dominant. Nonetheless, if we had to pick a winning index based on our evaluation results with a focus on balancing concurrent ingestion and querying (a very important workload in the Big Data era), the DHVB-tree index could be the winner if ingestion gets a higher priority, and the R-tree index could be the winner if queries get a higher priority.

Chapter 5

Evaluation of LSM Spatial Indexes For Dynamic Non-Point Data

5.1 Introduction

In this chapter, we extend the comparative study of LSM spatial indexes to include dynamic non-point data [1] (i.e., polygons) in order to see how these spatial indexes perform with such data. For this study, we evaluate only two of the previously studied index types, the R-tree and the SHB-tree, for the following reasons. First, to the best of our knowledge, there is no (efficient) method to use the DHB-tree or the DHVB-tree indexes for indexing and searching non-point data. Thus, they were excluded. Second, as shown in the previous chapter, due to the inferior performance of the SIF index versus the SHB-tree index, the SIF index is also excluded.

The rest of the chapter is organized as follows. First, we describe necessary changes to the

[1] Non-point data objects are objects with a spatial extent. In contrast, point data objects are objects without a spatial extent [31].

SHB-tree index for indexing and searching non-point data. Then, we present the evaluation plan and our experimental results. Finally, we provide conclusions.

5.2 Supporting Non-Point Data in the SHB-tree

In order to index and search non-point spatial objects, the SHB-tree index needs to deal with the following issues. The first issue is that a non-point spatial object can be decomposed into one or more cells in the grid hierarchy since its non-point area may overlap with one or more cells. The overlapping cell numbers can be indexed as multiple entries in the underlying LSM B-tree index, where each entry is a pair consisting of $\langle$*secondary key*, *primary key*$\rangle$ and the primary keys in the entries are identical. The decomposition works exactly just as a given query area is decomposed, as described in Section 4.2.2. Due to the possibility of multiple entries from a non-point spatial object, when a query is processed, and duplicated entries identified by a primary key must be discarded in order to return the correct cardinality of matching entries.

The second issue is that the overlapping cells for a given spatial object may lie at different levels of the grid hierarchy since the spatial object may *cover* a complete cell in upper grid levels. In order to search such covered upper level cells correctly, the decomposition process for a given query area, which was described in Section 4.2.2, should be modified properly. To begin with, a set of overlapping cell numbers are computed and then any contiguous cell numbers are combined to form a range of cell numbers in order to reduce the number of B-tree index searches. This is the same process that was described in Section 4.2.2, where all cell numbers were represented using bottom-level cell numbers. This is due to the fact that point objects are always mapped into bottom-level cells. However, for non-point objects, which can be mapped into non-bottom-level cells, all ascendant cell numbers for those bottom-level cell numbers must also be searched.

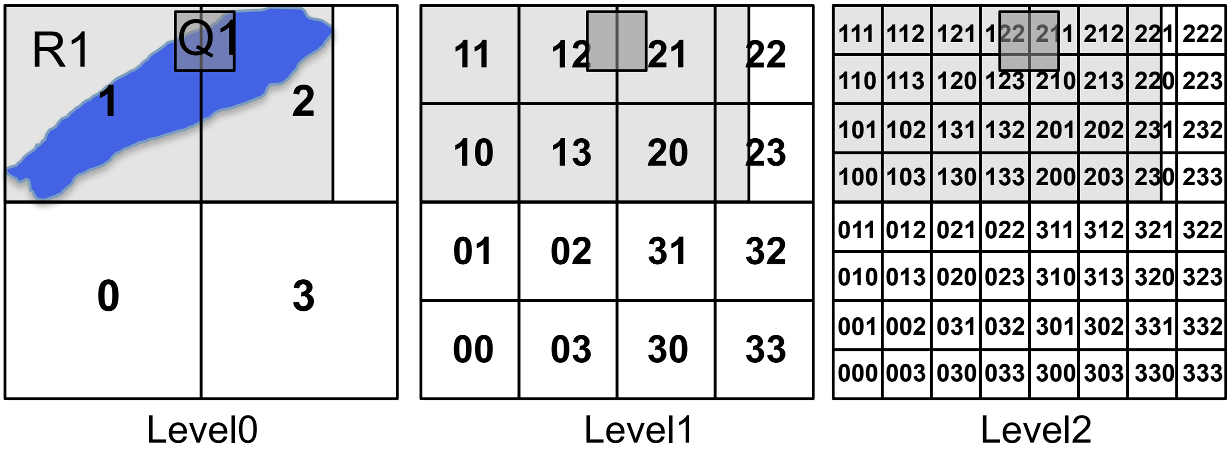


Figure 5.1: An example 3-level 2×2 grid hierarchy and cell numbers at each level

We can illustrate the aforementioned issues with an example. Figure 5.1 shows a 3-level 2×2 grid hierarchy and the cell numbers at each level. Suppose we want to find lakes that overlap with a certain city. The polygon surrounded by a rectangle R1 in light gray is a lake. R1, which represents the MBR of the lake object, is to be stored in an SHB-tree index. Further suppose that the small rectangle in dark gray, Q1, represents the MBR of the relevant city polygon.

In order to index the lake object, R1 is mapped into cell numbers 100 at level 0, 200 and 210 at level 1, and 220, 221, 230, and 231 at level 2. These cell numbers will be represented as 1000, 2001, 2101, 2202, 2212, 2302, and 2312 by appending their level number at the end of the cell number. Then, this set of entries is stored into the underlying LSM B-tree index, where an entry consists of a cell number (as a secondary key) and R1's primary key.

In order to search for overlapping objects in the SHB-tree index for the given city query area, we map Q1 into cell numbers 122, 123, 210, and 211 at level 2. Together with the level number, these cell numbers will be represented as 1222, 1232, 2102, and 2112. Then, in order to reduce the number of searches, contiguous cell numbers are combined to form a range. Thus, the following two cell-number search ranges are obtained: [1222–1232] and [2102–2112]. However, note that if we naively search for matching entries in the SHB-tree index using just these two ranges, none of the R1 entries will be found even though R1's cell 1000 at level 0 overlaps with Q1's cells 1222 and 1232 at level 2, and another R1 cell, 2101 at level 1 overlaps with other Q1 cells, 2102 and 2112 at level 2. This is because when R1 is indexed, the R1 cells overlapping with the query region are captured at non-bottom levels, level 0 and 1, while the query region cells are captured at the bottom level, level 2. Due to the difference of captured levels between the indexed object and the query region, they do not match each other during the index search.

To solve this problem of overlapping non-point objects that are captured at non-bottom levels, all of the ascendants cell numbers of the bottom-level cell numbers (or ranges) for a query region should be searched as well. In the example above, for the range [1222–1232], the ascendant cell numbers, 1000 at level 0 and 1201 at level 1 should be searched as well. As a result, R1's cell 1000 will be matched. Also, for the range [2102–2112], the ascendant cell numbers 2000 at level 0 and 2101 at level 1 should be searched, and thus another R1 cell, 2101 at level 1, will be matched. Note that among the two matched entries, only one should be returned after removing duplicated matches for an identical object; otherwise, the index will end up returning an incorrect number of matched entries and the cardinality of the final results would be wrong.

5.3 Evaluation Plan and Results

In this section, we explain the evaluation plan and experimental results. For the evaluation, we use the same basic experimental environment described in Chapter 4.3, including the AsterixDB instance setup, the tweet-generator clients, the query-generator clients, and the types of workloads, except that here the location fields in the experimental tweet dataset involve non-point spatial objects.

First, we describe the non-point spatial object datasets used for the evaluation. Then, we provide preliminary experimental results and use them to tune the performance of the SHB-tree index. After that, we provide the comparative experimental results for each workload.

5.3.1 Spatial Non-Point Dataset

For the evaluation, we used two types of synthetic tweet datasets that we generated in the same manner as described in Chapter 4.3.2 except that the sender location fields now contain non-point (polygonal) spatial objects. In order to synthetically model reasonable collections of real-world spatial objects, we mimicked size distributions for two types of collections of spatial objects: houses and lakes.

For the House-Tweet dataset, we generated 1.6 billion synthetic rectangular houses using the sampled 1.6 billion real-world GPS point data from *OpenStreetMap* [4] as the centroids of the houses' rectangles. The house sizes were generated based on the distribution of house sizes reported from the American Housing Survey for the Unites States [1]. The resulting house sizes followed a Gaussian distribution with a mean of 2000 square feet and a standard deviation of 1000 square feet. For the Lake-Tweet dataset, we generated 1.6 billion rectangular lakes, again using the GPS point data as the centroids of the rectangles. Their sizes were generated based on a Zipf distribution of the world's lake sizes [28]. The minimum size of a lake was set to 0.001 km^2, the maximum to 80000 km^2, and the slope (the value of the exponent characterizing the Zipf distribution) was set to 4.

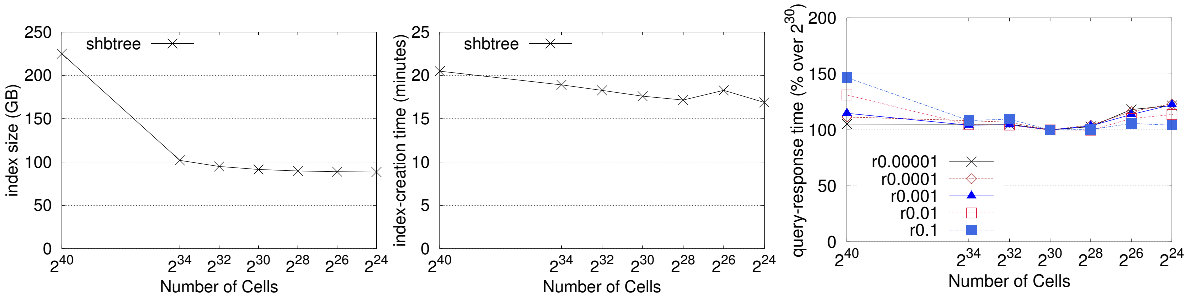
Note that the reason why we use these two two datasets in this evaluation is not because we want to compare the two types of indexes specifically with the houses and lakes as the spatial objects, but because we want to compare the alternate indexes with different characteristics of non-point spatial objects whose sizes follow very different but representative distributions, Gaussian and Zipf, that can be commonly observed in the real world and whose object sizes are large enough to require treatment as non-point spatial objects (as opposed to point objects).

5.3.2 Preliminary Experiments for Tuning the SHB-tree

Unlike a point object, a non-point object can be mapped into multiple cells, which can undermine the performance of the SHB-tree index since many entries need to be inserted into the index for a non-point object and the search overhead during queries can increase accordingly. If the size of the cells in the grid hierarchy is increased, the number of mapped cells for a non-point object will decrease, but the number of false positives can increase during queries. Thus, it is critical to find a reasonable sweet spot for the cell size in the grid hierarchy such that both the number of mapped cells and the number of false positives are minimized as much as possible.

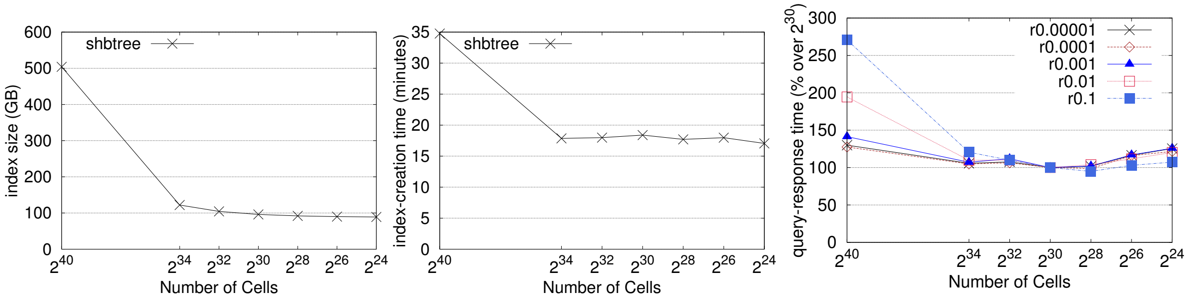
The implemented SHB-tree index is configured here to have a 6-level grid hierarchy, and when the index is created initially, users can set the number of cells at each grid level by choosing one of three options: "l" (low), "m" (medium), "h" (high), where "l", "m", and "h" denote a 4×4 grid (16 cells), an 8×8 grid (64 cells), and a 16×16 grid (256 cells), respectively. For example, if a user configures an SHB-tree index with a 6-level grid hierarchy to have the setting (h,h,h,h,h,h), the top level will decompose the given space into a 16×16 grid, the second level decomposes each parent cell into a 16×16 grid, and so on. Thus, there will be 2^{48} cells at the bottom grid level in this case.

We empirically identified the sweet spot for the number of SHB grid cells by executing a set of preliminary experiments while varying the number of cells at each grid level as follows: (h,h,h,h,l,l), (l,l,m,m,m,h), (l,l,l,l,h,h), (l,l,l,l,h,m), (l,l,l,l,m,m), (l,l,l,l,l,m), and (l,l,l,l,l,l), where these options cause the corresponding bottom level grids to have 2^{40}, 2^{34}, 2^{32}, 2^{30}, 2^{28}, 2^{26}, and 2^{24} cells, respectively. For the tuning experiments, the static workload (described in Chapter 4.3.3.1) was used.



(a) Index size (b) Index-creation time (c) Query-response time

Figure 5.2: The effects of varying the number of cells in the SHB-tree index with the House-Tweet dataset



(a) Index size (b) Index-creation time (c) Query-response time

Figure 5.3: The effects of varying the number of cells in the SHB-tree index with the Lake-Tweet dataset

We first tuned the SHB-tree index for the House-Tweet dataset. Figure 5.2 shows the effects of varying the number of cells in the SHB-tree index on the index size, index-creation time, and query-response time. The query-response time graph shows the query response times as a percentage relative to the query response times with the index setting 2^{30}, and the queries are index-only-scan queries. The x-axis of the graphs shows the number of cells in the bottom-level grid resulting from the aforementioned option settings. Figure 5.3 shows the same types of graphs for the Lake-Tweet dataset.

Overall, as the number of bottom-level cells is decreased, the index size and index-creation time decrease. More specifically, changing the index setting from 2^{40} to 2^{34} reduces the index size and index-creation time dramatically and the rest of the changes show much smaller margins. The query-response time also shows a similar trend until the number of cells reaches 2^{30}. All of these trends can be attributed to the fact that the number of cells overlapping with non-point spatial objects tends to decrease as the number of cells in the grid hierarchy decreases. More interestingly, a larger number of cells in the grid hierarchy means not only a larger number of overlapping cells for non-point objects to be stored but also a potentially larger number of overlapping cells for each of the given query regions, which makes matters worse for the index by causing more searches. In Figures 5.2(c) and 5.3(c), we can see that the larger radius queries' response times are far more sensitive than the smaller radius queries when the number of cells in the grid hierarchy is set to 2^{40}. This can be attributed to the aforementioned larger number of cells overlapping with both the stored non-point objects and the query regions.

Beyond the 2^{30} setting, we see in both figures that the query-response times increase as the number of cell is decreased further. This is likely due to the increased number of false positives. Also, very interestingly, the smaller radius queries' response times seem to be more sensitive to the false positives than the larger ones, as is shown for the 2^{24} setting in Figures 5.2(c) and 5.3(c).

An overall comparison between the graphs in Figure 5.2 and Figure 5.3 also constrasts the sensitivities of the two datasets to varying the number of cells in the grid hierarchy. The Lake-Tweet dataset of the Zipf distribution tends to show more negatively sensitive results when the number of cells is increased.

These experimental results confirm that picking a proper setting for the number of cells in an SHB-tree index is critical to its performance. Based on these initial experimental results, we chose to use 2^{30} cells, i.e., option (l,l,l,l,h,m), for both datasets in the main experiments that now follow.

5.3.3 Static Workload with the House-Tweet Dataset

5.3.3.1 Index Size and Creation Time

	pidx	**rtree**	**shbtree**
Index size (GB)	1024.83	74.59	91.43
Index creation time (minutes)	107.25	16.99	16.30

Table 5.1: Index size and creation time for the house-tweet dataset

Table 5.1 shows the primary index (denoted as "pidx") size after loading 1.6 billion records and the elapsed time for the loading. Also, it shows the elapsed times for creating each type of spatial secondary index on the sender location field (which has rectangle values) of 1.6 billion records and the corresponding index sizes. The R-tree index-creation time is a bit larger than that of the SHB-tree index even though the SHB-tree index is notably bigger than the R-tree index. This may be attributed to the proper setting of the number of cells in the grid hierarchy (from the preliminary experiments) and to the R-tree index's relative-comparison behavior based on a Hilbert curve during its bulk-loading, which seems to have more overhead than the SHB-tree index's bulk-loading. Also, the entry-size differences shown back in Table 4.2 affect the index size difference; the point field in each entry is replaced here with an MBR field whose size is 32 bytes for non-point data.

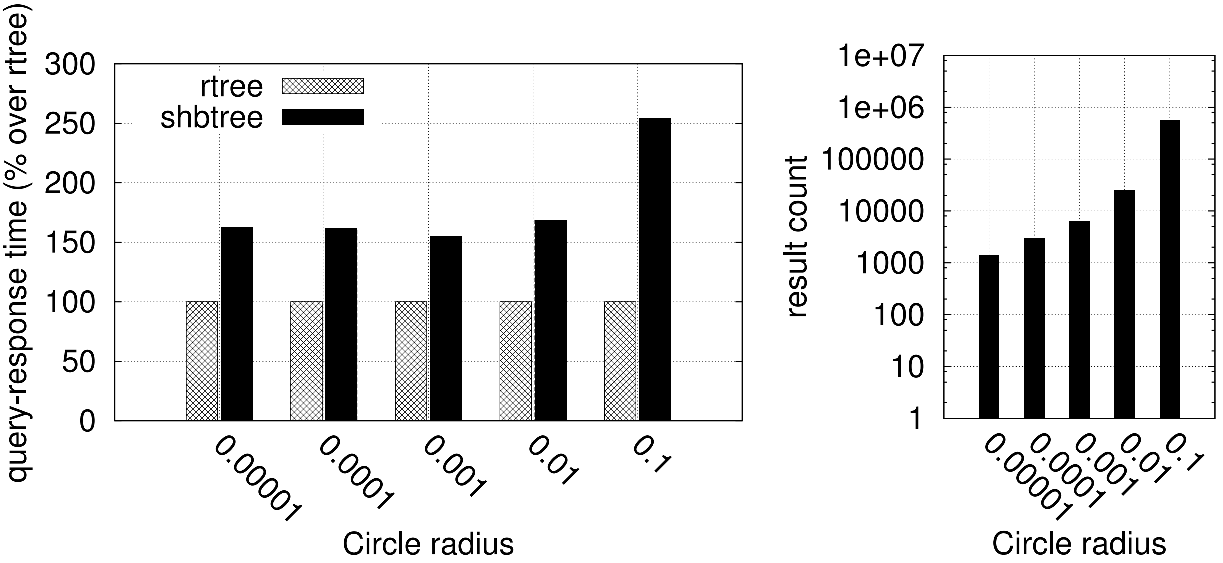
In addition, from Table 5.1 and Table 4.3, we can see that the size of each index (as shown in Table 5.1) built with the rectangle data from the House-Tweet dataset is about 25.6 GB larger than the size of each corresponding index (as shown in Table 4.3) built with the point data in Chapter 4. 25.6 GB is the difference between the 1.6 billion rectangle values' sizes and the 1.6 billion point values' sizes.

Circle radius **Query type**	**0.00001**	**0.0001**	**0.001**	**0.01**	**0.1**
Index-only-scan select	102	99	102	109	223
Index-only-scan join	1567	1582	1584	1966	n/a
Non-index-only-scan select	235	242	405	1367	4661
Non-index-only-scan join	7011	7772	10808	11451	n/a

Table 5.2: R-tree's query-response time (in milliseconds)

5.3.3.2 Index-Only-Scan Select Query

The performance results for the static workload's index-only-scan select queries with the House-Tweet dataset are discussed next.



(a) Select response time % over R-tree

(b) Select result count

Figure 5.4: Static workload's *index-only-scan* select query's response time percentage over R-tree and result count

Figure 5.4(a) shows the query-response times as a percentage over the query-response times of R-tree. The corresponding R-tree query-response times are shown in Table 5.2. Figure 5.4(b) shows the query result counts. Profile information such as the false-positive ratio, operators' elapsed times, and cache-miss count is shown in Figures 5.5–5.7. As described in Chapter 4.4.1.2, the false-positive ratio shown in Figure 5.5 represents $\frac{A}{B}$, where B is

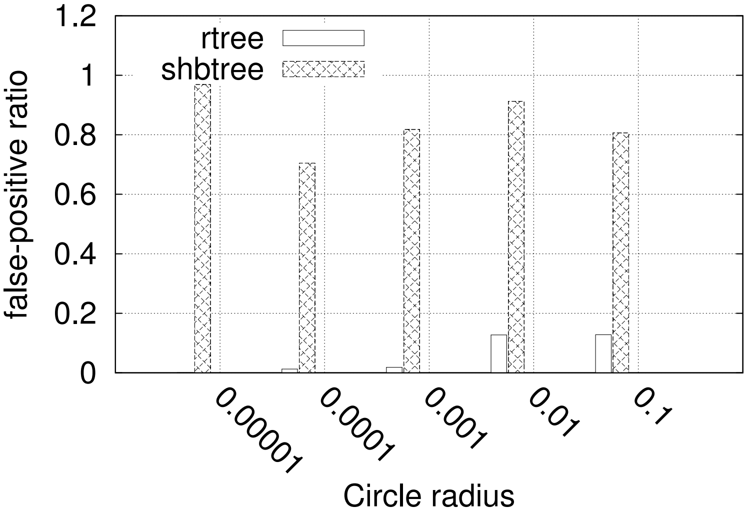
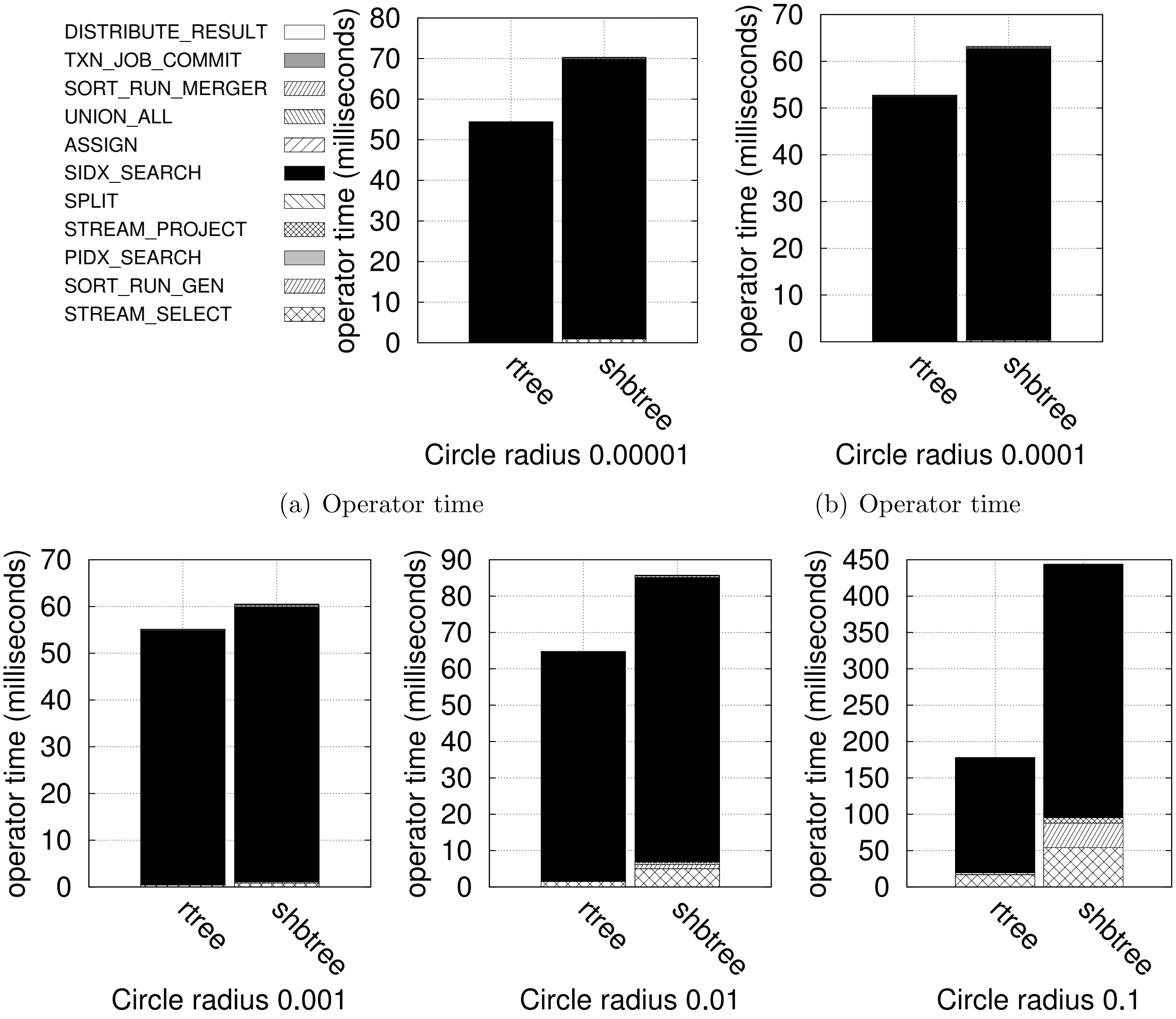


Figure 5.5: Static workload's *index-only-scan* select query's false positive ratio

the number of qualified records from a spatial index search and A is the number of records discarded from among these qualified records. Figure 5.6 details the elapsed times spent on each query runtime operator involved in queries with different circle radii; the query compilation and remote query launching overheads (which again took around 50 milliseconds together) are not shown. In the figures, the spatial index search operator is denoted as `SIDX_SEARCH` and the symbol `TXN_JOB_COMMIT` represents the lock-releasing overhead when a query is over. One implementation change[2] from Chapter 4 to Chapter 5 is that the shared-mode locks here are always *instant* locks regardless of whether the query is an index-only-scan query or not. Thus, unlike the charts shown in Chapter 4, there is no lock-releasing overhead here when a query is over. Also, in order to remove duplicated entries returned from the SHB-tree index, where the duplication may occur if a house rectangle overlaps with multiple cells as described in Section 5.2, a distinct operation based on sorting is applied. In the figures, `SORT_RUN_GEN` and `SORT_RUN_MERGER` capture the SHB-tree index overhead of sorting for this distinct operation. The R-tree index does not have this overhead. Figure 5.7 shows the cache-miss counts (from AsterixDB's buffer cache) caused

[2] We have changed AsterixDB's implementation to use instant locks for all read operations in the following reasons. The first reason is the severe lock-release overhead for index-only-scan queries, which was observed from the experiments in Chapter 4. The second reason is to achieve the deadlock-free locking protocol that was decribed in Chapter 3.



(a) Operator time

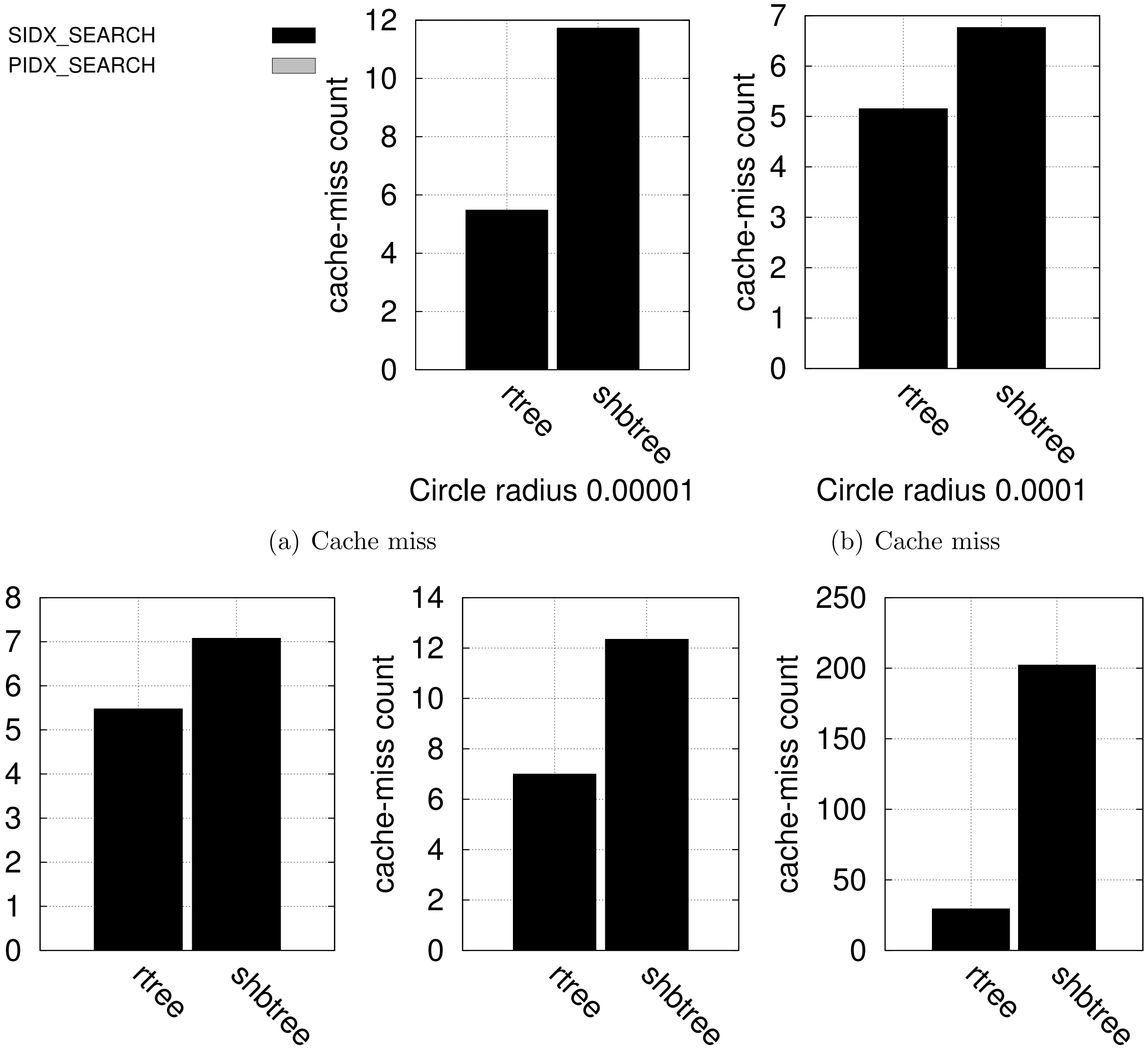
(b) Operator time

(c) Operator time

(d) Operator time

(e) Operator time

Figure 5.6: Static workload's *index-only-scan* select query's operator time



(a) Cache miss

(b) Cache miss

(c) Cache miss

(d) Cache miss

(e) Cache miss

Figure 5.7: Static workload's *index-only-scan* select query's cache-miss count

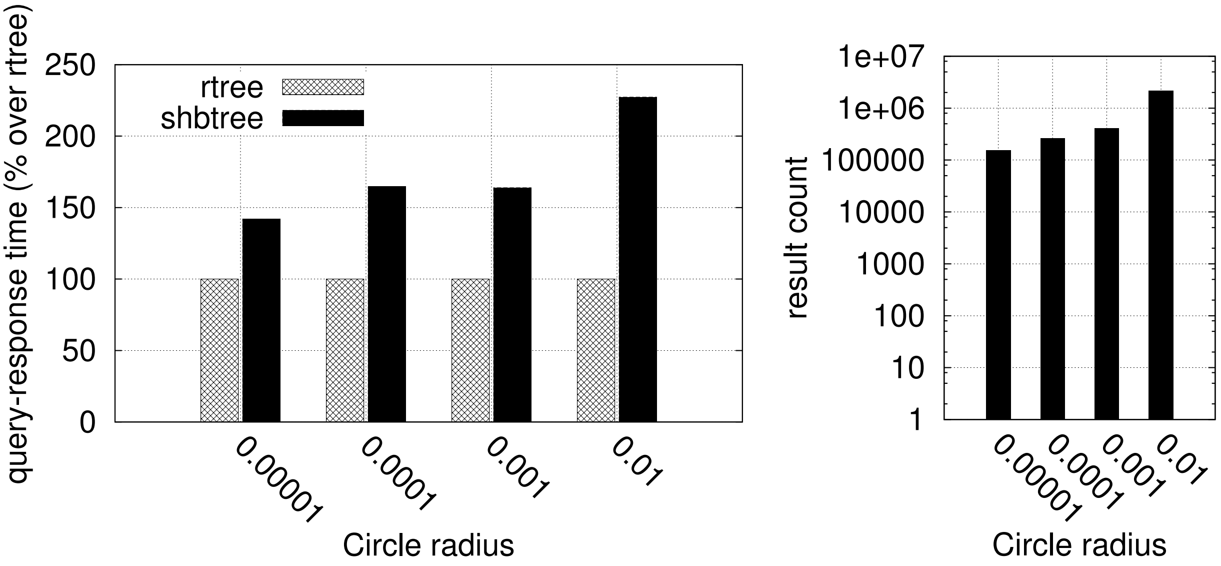
mainly by the spatial-index accesses.

Overall, we see that the R-tree index outperforms the SHB-tree index here regardless of the query radius size. Based on the profiled information, the false-positive ratio shown in Figure 5.4(a) is a dominant factor that dictates the performance gap between the R-tree index and the SHB-tree index. The SHB-tree index shows a much higher false-positive ratio than the R-tree index. This high false-positive ratio in the SHB-tree index must be caused by reducing the number of cells from 2^{48} in Chapter **??** to 2^{30} here, which increased the avarege number of objects mapped into a single cell. Even though we empirically found a cell-number configuration from the preliminary experiments in Section 5.3.2 to achieve better performance for the SHB-tree index, still the best configuration we picked may not show the SHB-tree index performance which may compete with the R-tree index due to the much higher false-positive ratio caused by changing the SHB-tree index setting in order to have less overlapping cells for a non-point object.

5.3.3.3 Index-Only-Scan Join Query

The query-response times as a percentage over the R-tree and the result count for the index-only-scan join queries are shown in Figures 5.8(a) and 5.8(b), respectively. Also, the profile information such as the false-positive ratio, operators' elapsed times, and cache-miss count is shown in Figures 5.9–5.11. In the join operations, the outer table's (QuerySeedTweet) primary-index search operator is denoted as `OUTER_PIDX_SEARCH`, and the inner table's (Tweet) primary and secondary indexes are denoted as `INNER_PIDX_SEARCH` and `INNER_SIDX_SEARCH`, respectively.

Overall, due to the SHB-tree index's significantly higher false-positive ratio, its performance is again seem to be inferior than the R-tree index. These results show a trend similar to the index-only-scan select query results except that the numbers shown in the profile information



(a) Join response time % over R-tree

(b) Join result count

Figure 5.8: Static workload's *index-only-scan* join query's response time percentage over R-tree and result count

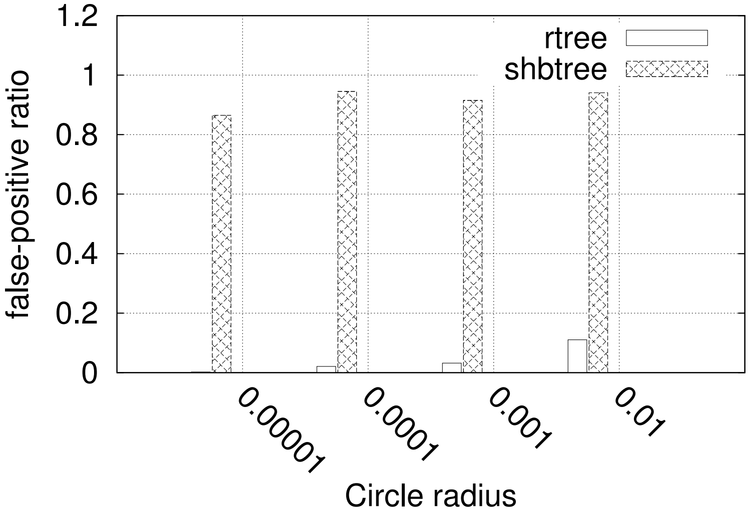
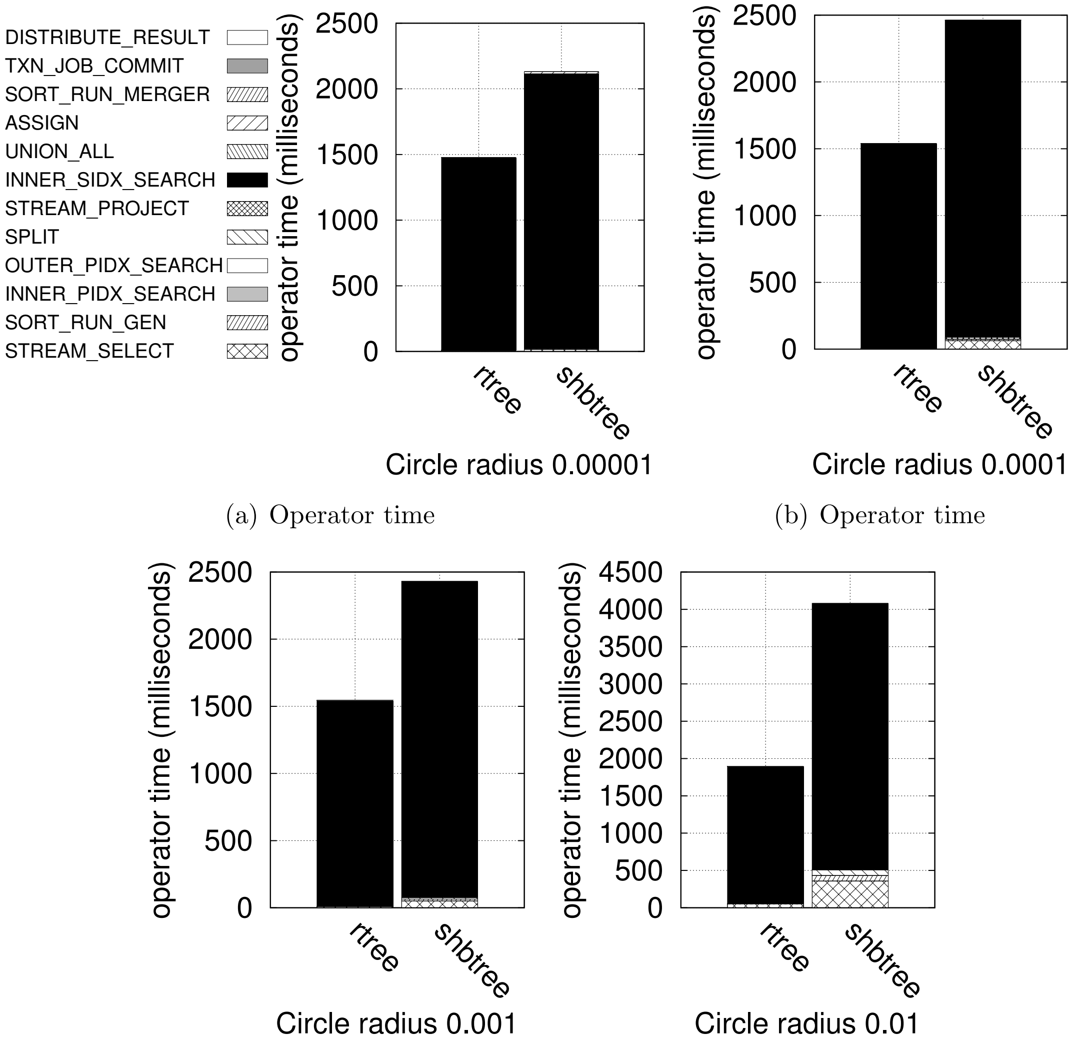


Figure 5.9: Static workload's *index-only-scan* join query's false positive ratio



(a) Operator time

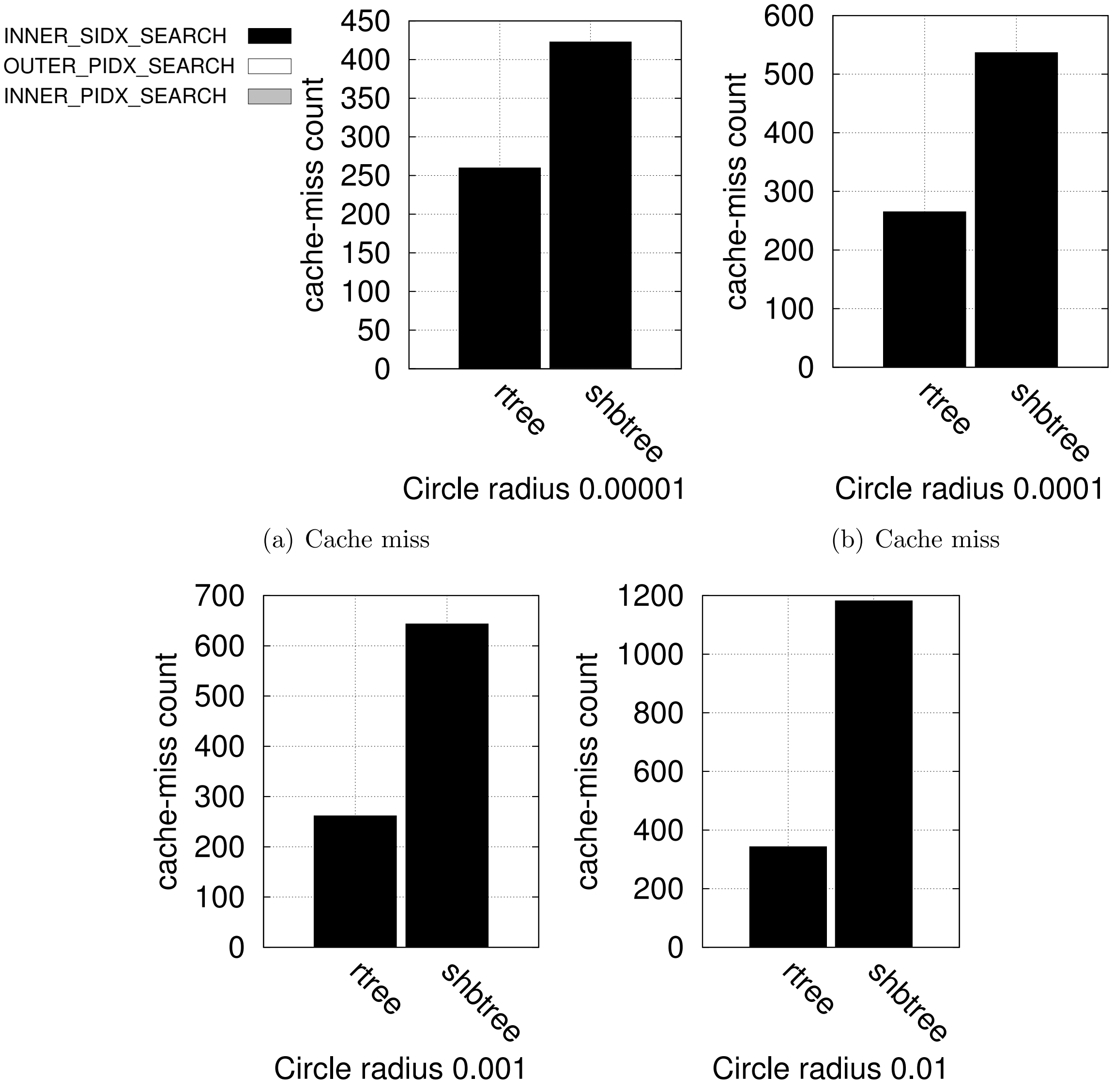
(b) Operator time

(c) Operator time

(d) Operator time

Figure 5.10: Static workload's *index-only-scan* join query's operator time

charts for the select queries are magnified for the join queries since a join query essentially leads to 100 range queries from an index-searching perspective.



(a) Cache miss

(b) Cache miss

(c) Cache miss

(d) Cache miss

Figure 5.11: Static workload's *index-only-scan* join query's cache-miss count

5.3.3.4 Non-Index-Only-Scan Select Query

The results of the non-index-only-scan select query are discussed next. Figure 5.12 shows the query-response times as a percentage over the R-tree. Also, their profile information such as the operators' elapsed times and cache-miss count is shown in Figures 5.13 and 5.14, where the black bars and gray bars represent measured values for the secondary spatial index and the primary index, respectively. We omit the result counts and false-positive ratios since they are exactly the same as in the index-only-scan select query case.

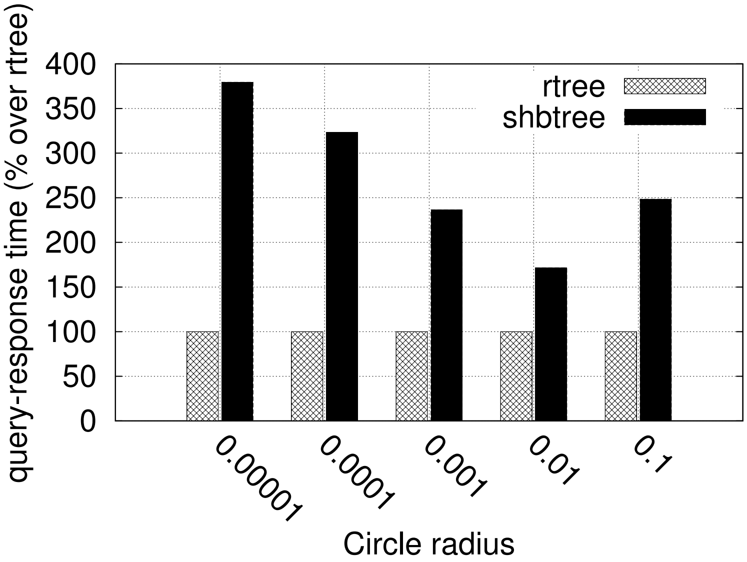
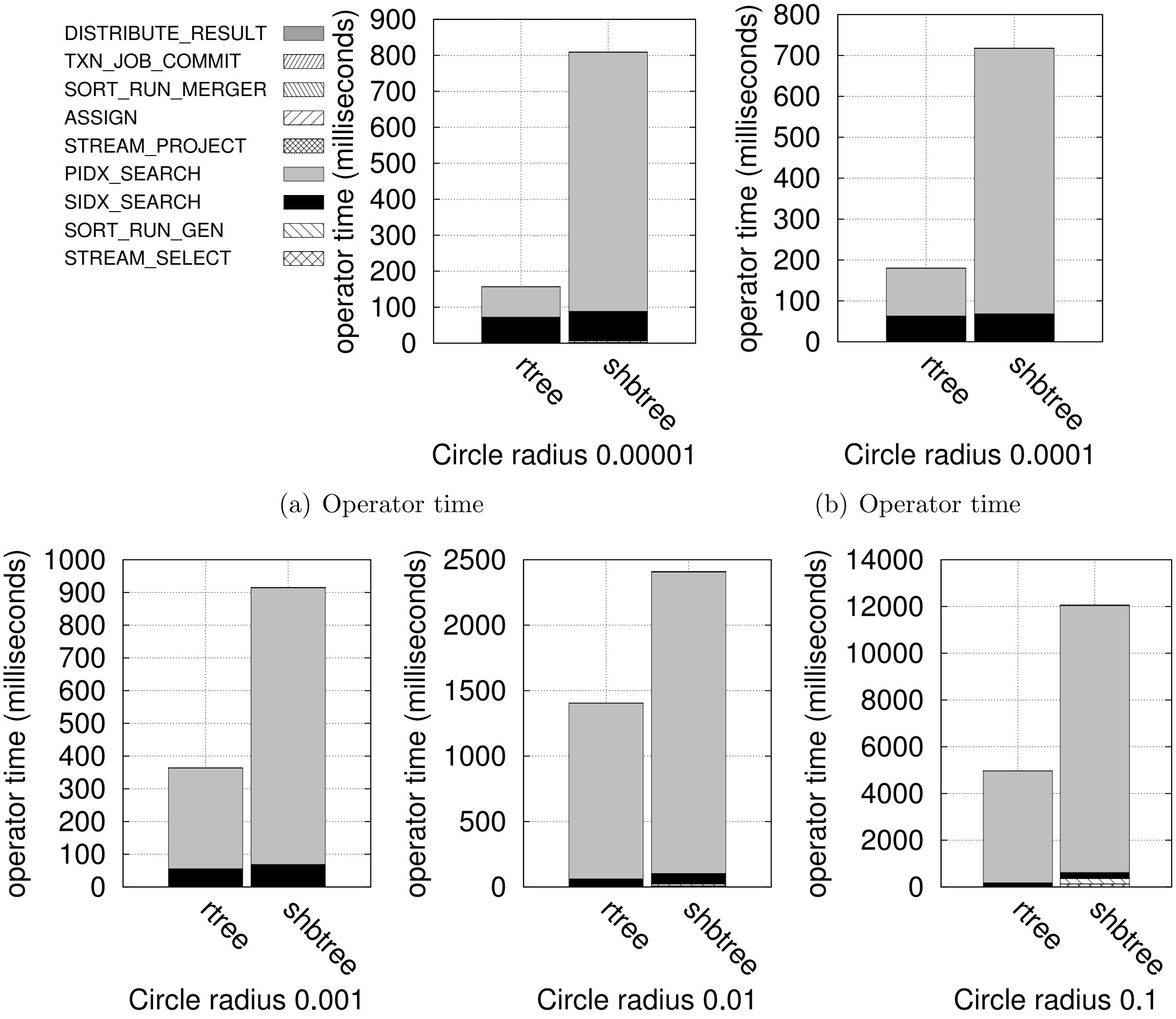


Figure 5.12: Static workload's *non-index-only-scan* select query's response time percentage over R-tree

In general, for the non-index-only-scan queries, the primary-index search cost is the dominant factor, as shown in Figures 5.13 and 5.14. Overall, the R-tree index far outperforms the SHB-tree index as shown in Figure 5.12. This is due to the SHB-tree index's higher false-positive ratio (shown in Figure 5.5). For the non-index-only-scan queries, more false positives means more primary index lookups, and thus even more costs. As the circle radius size gets larger, the gap of the false-positive ratio between the two indexes tends to decrease as shown in Figure 5.5. This trend is reflected in the cache-miss count, which dictates the profiled operator time, and consequently in the relative query-response times as shown in Figure 5.12.



(a) Operator time

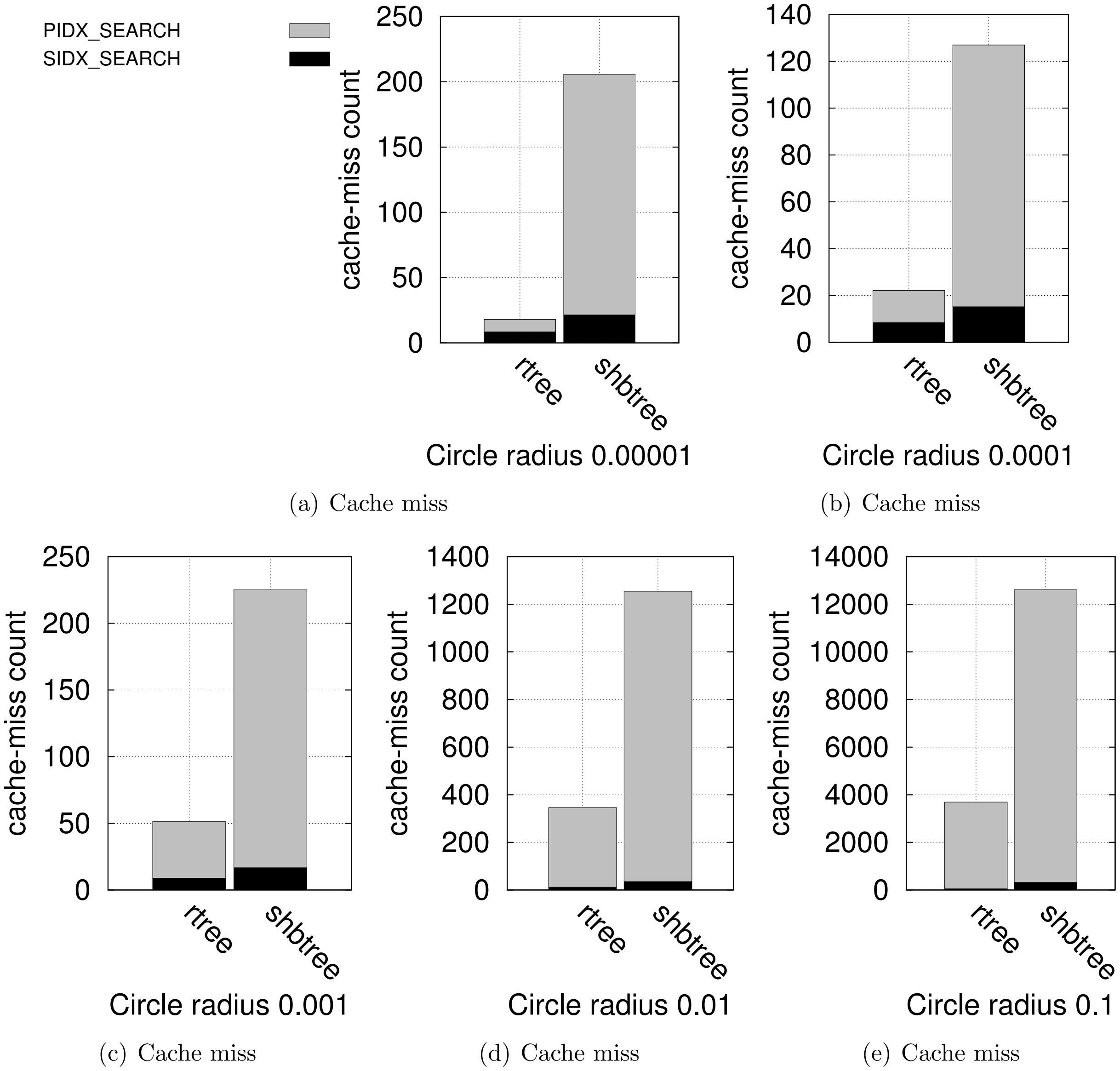
(b) Operator time

(c) Operator time

(d) Operator time

(e) Operator time

Figure 5.13: Static workload's *non-index-only-scan* select query's operator time



(a) Cache miss

(b) Cache miss

(c) Cache miss

(d) Cache miss

(e) Cache miss

Figure 5.14: Static workload's *non-index-only-scan* select query's cache-miss count

5.3.3.5 Non-Index-Only-Scan Join Query

The results of the non-index-only-scan join query are discussed next. Figure 5.15 shows the query-response time as a percentage over the R-tree. Also, the profile information such as the operators' elapsed times and the cache-miss count is shown in Figures 5.16 and 5.17, where the black bars and the gray bars represent the inner table's secondary spatial index and the inner table's primary index, respectively. The trend shown in the charts is similar to the non-index-only-scan select query case, i.e., the false-positive ratio affects the cache-miss count, which dictates the profiled operators' time and the query-response time in the end. Thus, due to the higher false-positive ratio, the SHB-tree's query-response times are larger than those of the R-tree index.

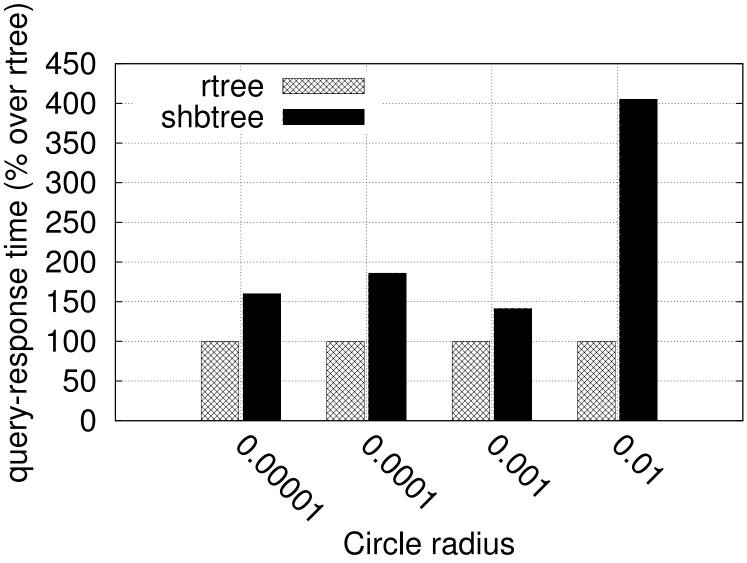
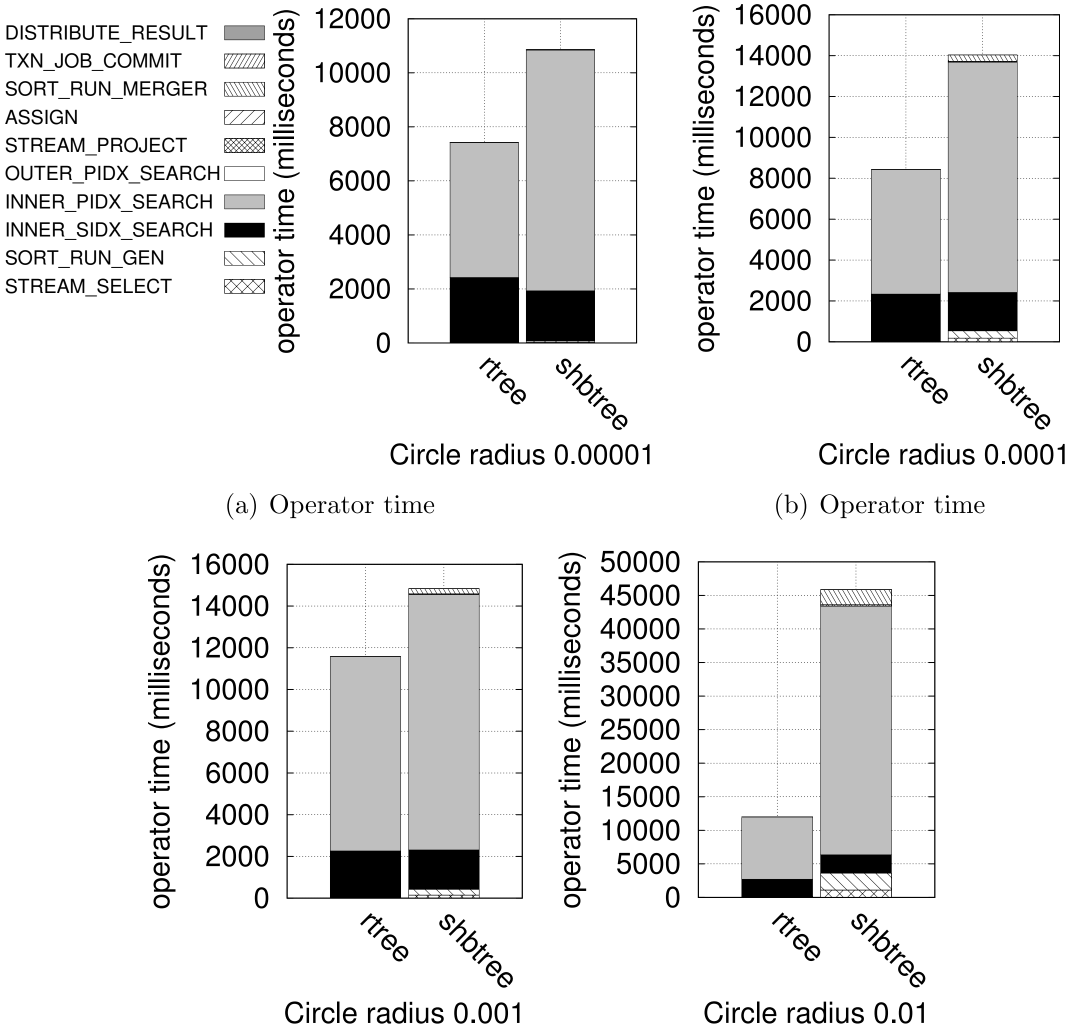


Figure 5.15: Static workload's *non-index-only-scan* join query's response time percentage over R-tree

5.3.4 Dynamic Workload 1 with House-Tweet Dataset

Dynamic workload 1 tests the scalability of each index in terms of IPS (inserts per second) for one hour of data ingestion. Figure 5.18(a) shows the IPS results while varying the number



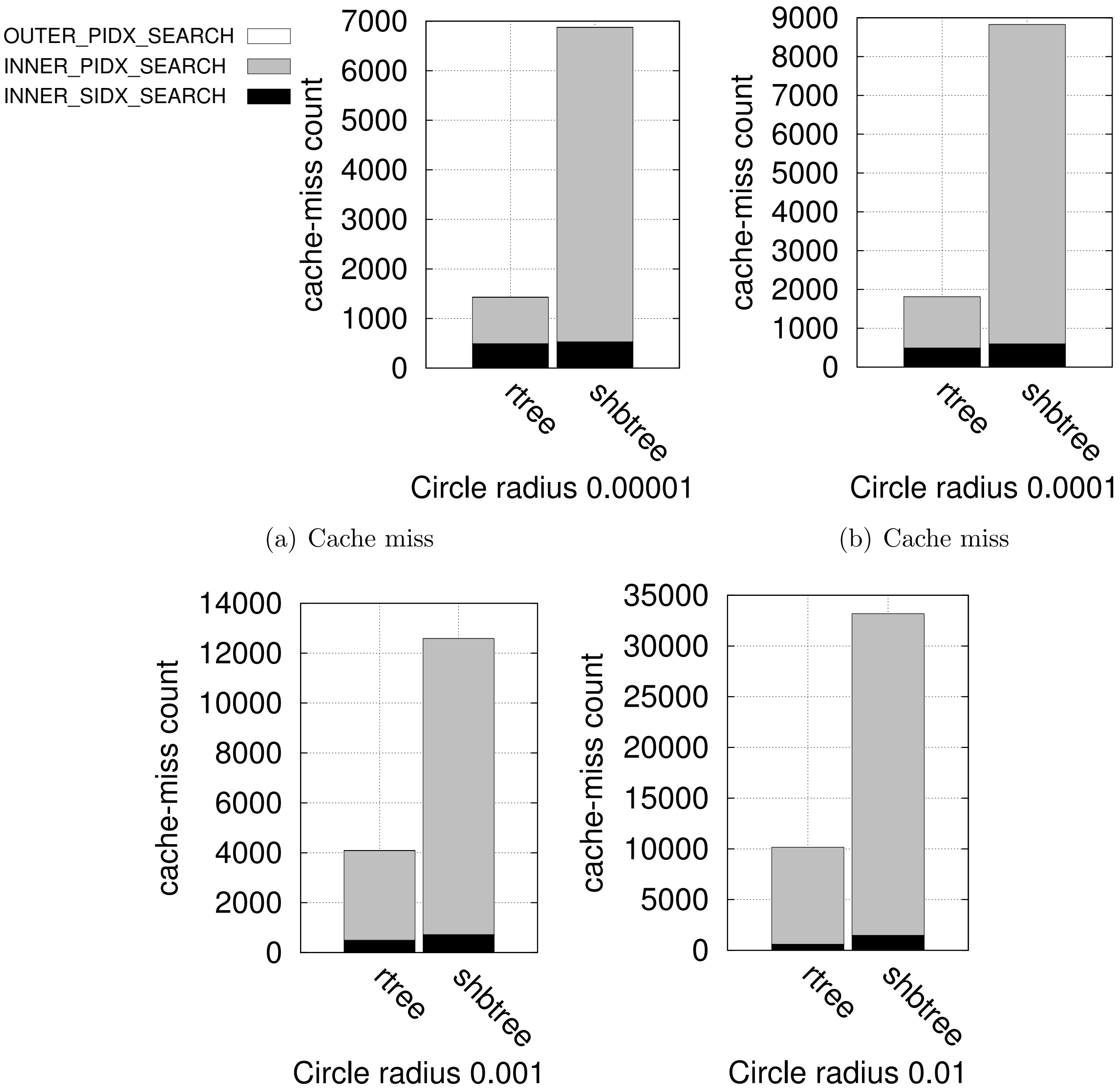
(a) Operator time

(b) Operator time

(c) Operator time

(d) Operator time

Figure 5.16: Static workload's *non-index-only-scan* join query's operator time



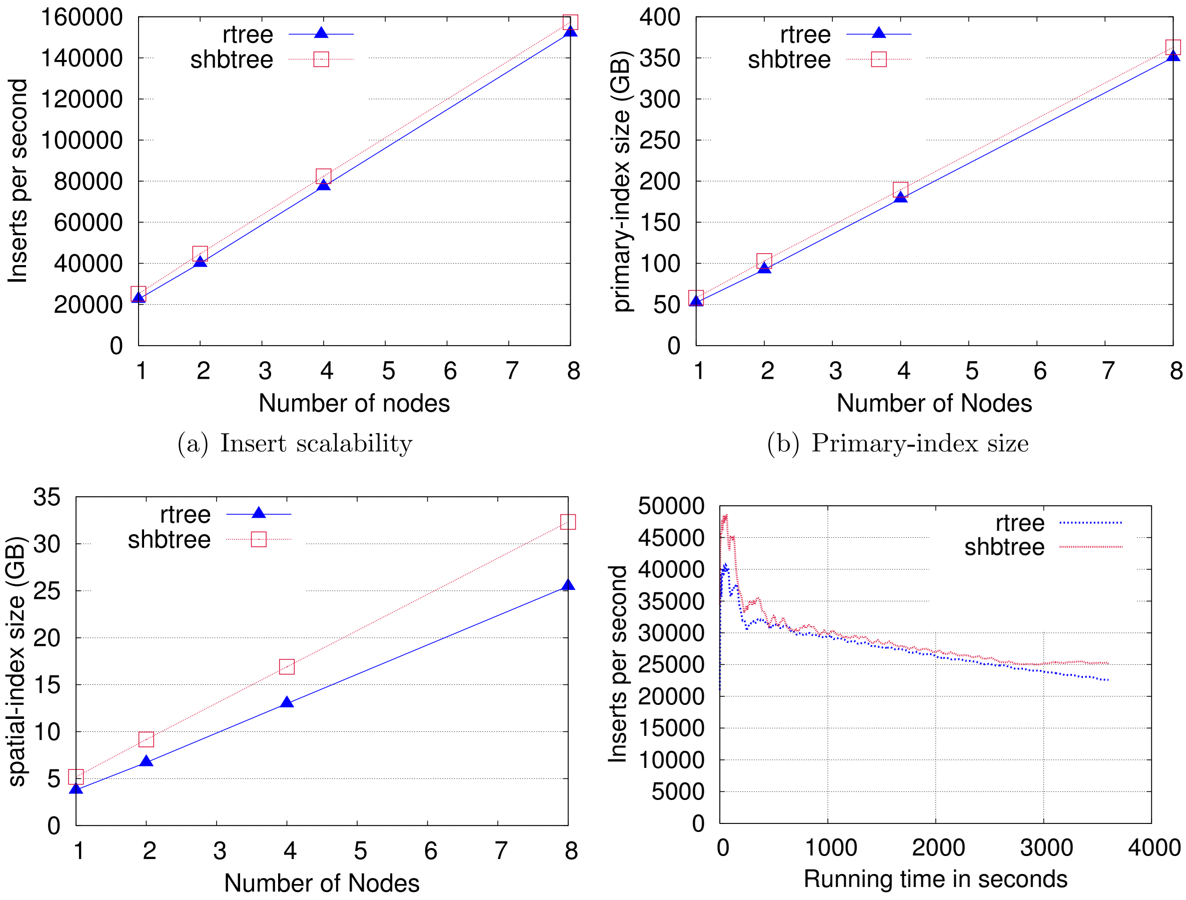
(a) Cache miss

(b) Cache miss

(c) Cache miss

(d) Cache miss

Figure 5.17: Static workload's *non-index-only-scan* join query's cache-miss count



(a) Insert scalability

(b) Primary-index size

(c) Spatial-index size

(d) IPS every five seconds

Figure 5.18: Dynamic workload 1

of nodes. As the number of nodes increases, the IPS rate increases linearly for both types of indexes. The primary and secondary index sizes resulting from one hour of ingestion are shown in Figures 5.18(b) and 5.18(c), respectively. In addition, Figure 5.18(d) shows a series of instantaneous IPS values measured every five seconds during the one-hour ingestion from a single node.

The SHB-tree index outperforms the R-tree index in terms of IPS. This result is quite similar to the results for the point objects shown in Figure 4.18. The difference here is that due to the rectangle objects, the overall IPS is slightly smaller and the index sizes are larger than for the point objects.

5.3.5 Dynamic Workload 2 with the House-Tweet Dataset

Dynamic workload 2 tests the indexes' ability to ingest and query concurrently for each index while the ingestion rate is varied. Figure 5.19 shows the results of the index-only-scan query case for this workload, where rate 1, rate 2, rate 3, and rate 4 represent making the tweet generators sleep for 1 millisecond after every 1, 10, 100, and 1000 generated record(s), respectively. Figure 5.19(a) shows the IPS results for each spatial index for each ingestion rate after one hour of ingestion while concurrent queries are being processed. Figure 5.19(b) shows the corresponding QPS (queries per second) results.

First, as the ingestion rate goes up, IPS also goes up, but it starts to saturate at rate 3 and goes up just slightly at rate 4 for both indexes. In contrast, the corresponding QPS is the highest at rate 1 and then goes down. As for point data, this is due to the limited resources in the cluster being shared by both the ingestion and query workloads.

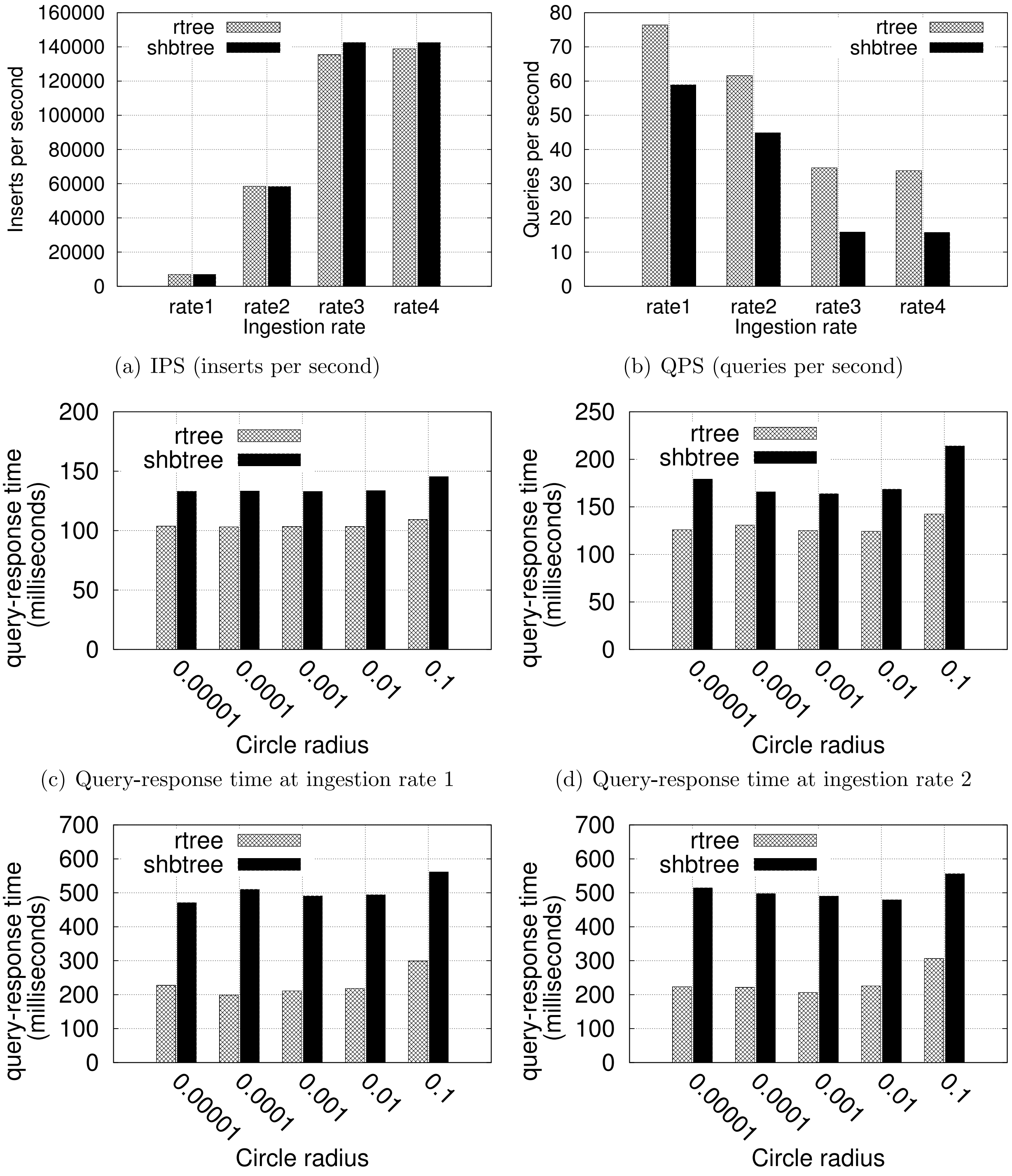
Also, as shown in Figure 5.20, the results of the non-index-only-scan query case for dynamic workload 2 shows the same overall trend as the index-only-scan query case. The difference here is that the measured IPSs and QPSs are smaller and the gaps in the QPS chart are smaller; the query-response times are larger. This is mainly due to the primary-index search overhead.

5.3.6 Static Workload with the Lake-Tweet Dataset

	pidx	**rtree**	**shbtree**
Index size (GB)	1024.84	74.59	96.08
Index creation time (minutes)	108.26	20.30	17.99

Table 5.3: Index size and creation time for the lake-tweet dataset

To explore a different spatial object size distribution, Table 5.3 shows the primary index (denoted as pidx) size after loading 1.6 billion Laek-Tweet records and the elapsed time



(a) IPS (inserts per second)

(b) QPS (queries per second)

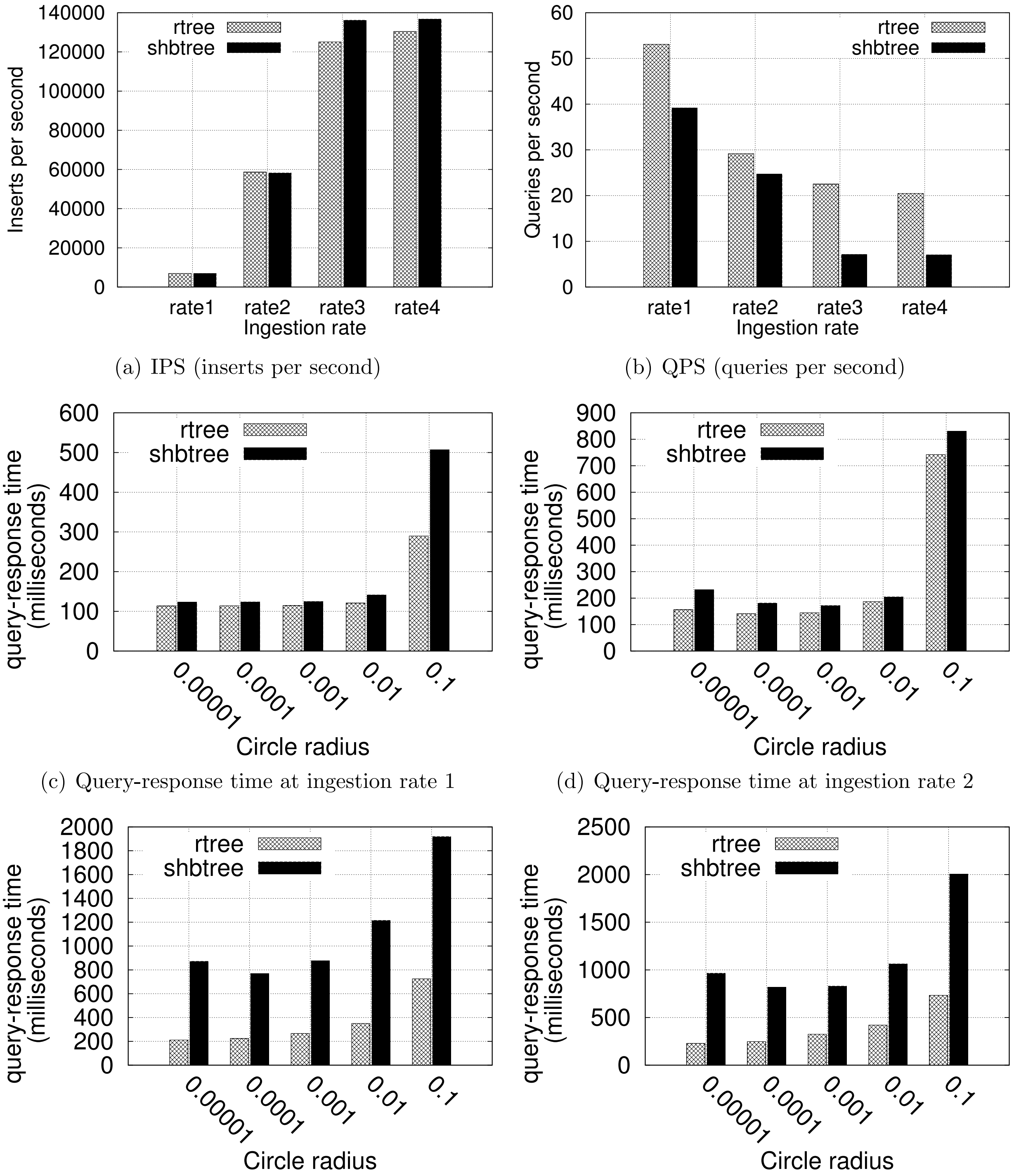
(c) Query-response time at ingestion rate 1

(d) Query-response time at ingestion rate 2

(e) Query-response time at ingestion rate 3

(f) Query-response time at ingestion rate 4

Figure 5.19: Results of dynamic workload 2 for the *index-only-scan* query case, where rate 1, rate 2, rate 3, and rate 4 represent making tweet generators sleep for 1 millisecond after every 1, 10, 100, and 1000 generated record(s), respectively.



(a) IPS (inserts per second)

(b) QPS (queries per second)

(c) Query-response time at ingestion rate 1

(d) Query-response time at ingestion rate 2

(e) Query-response time at ingestion rate 3

(f) Query-response time at ingestion rate 4

Figure 5.20: Results of dynamic workload 2 for the *non-index-only-scan* query case, where rate 1, rate 2, rate 3, and rate 4 represent making tweet generators sleep for 1 millisecond after every 1, 10, 100, and 1000 generated record(s), respectively.

for the loading. Also, it shows the elapsed times for creating each spatial secondary index on the sender location field (which has rectangular values) of 1.6 billion records and the corresponding index sizes. Replacing the House-Tweet dataset with the Lake-Tweet dataset increased the SHB-tree index size from 91GB to 96GB, but their primary index size and the R-tree index size remained almost the same, as can be seen in Table 5.1 and Table 5.3. This can be attributed to the lake rectangles' size, which is generated based on the Zipf distribution as described in Section 5.3.1. This may have made a small portion of large lake rectangles overlap with more cells such that the number of secondary keys (i.e., cell numbers) to be inserted into the SHB-tree index increased.

As shown in the charts in the rest of the section, the overall relative performance results based on the Lake-Tweet dataset between the R-tree index and the SHB-tree index are similar to the ones with the House-Tweet dataset due to the SHB-tree index's high false-positive ratio. We present the result charts based on the Lake-Tweet dataset experiments without further detailed explanations.

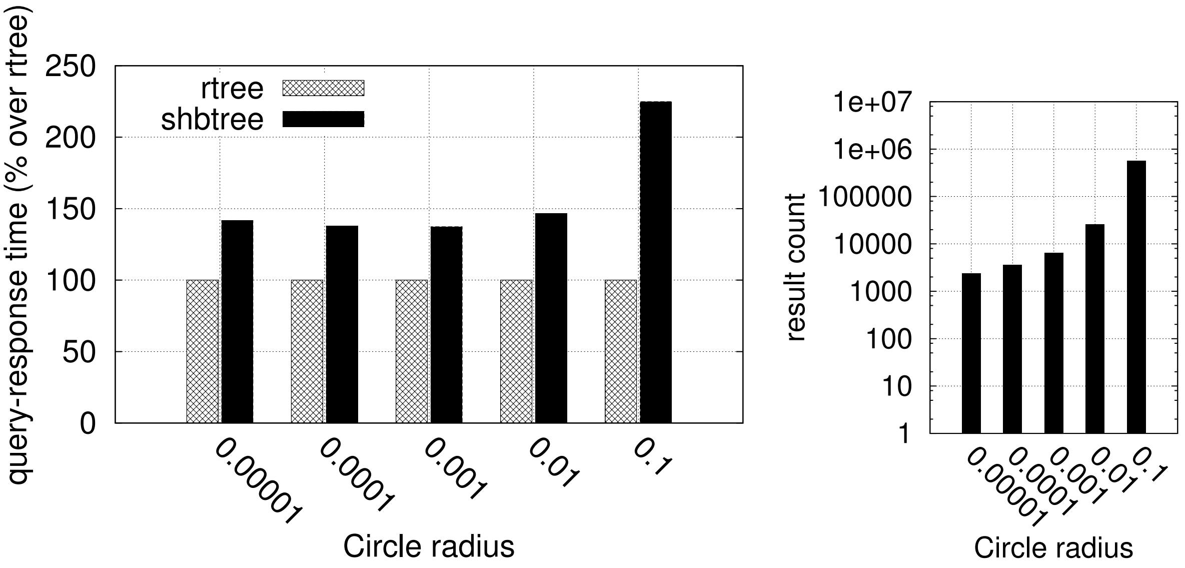
Circle radius **Query type**	**0.00001**	**0.0001**	**0.001**	**0.01**	**0.1**
Index-only-scan select	107	106	106	115	237
Index-only-scan join	1678	1719	1747	2153	n/a
Non-index-only-scan select	283	270	438	1469	5072
Non-index-only-scan join	8991	9383	11628	12293	n/a

Table 5.4: R-tree's query-response time (in milliseconds)

5.3.6.1 Index-Only-Scan Select Query

Figure 5.21(a) shows the query-response time as a percentage over the query-response time of the R-tree index. The corresponding R-tree's query-response times are shown in Table 5.4. Figure 5.21(b) shows the queries' result counts. Their profile information, such as the false-positive ratio, operators' elapsed times, and cache-miss count, is shown in Figures

5.22–5.24.



(a) Select response time % over R-tree

(b) Select result count

Figure 5.21: Static workload's *index-only-scan* select query's response time percentage over R-tree and result count

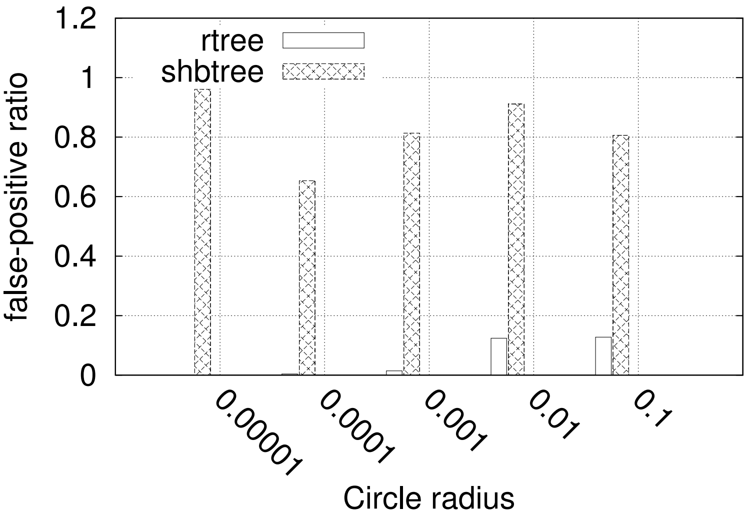
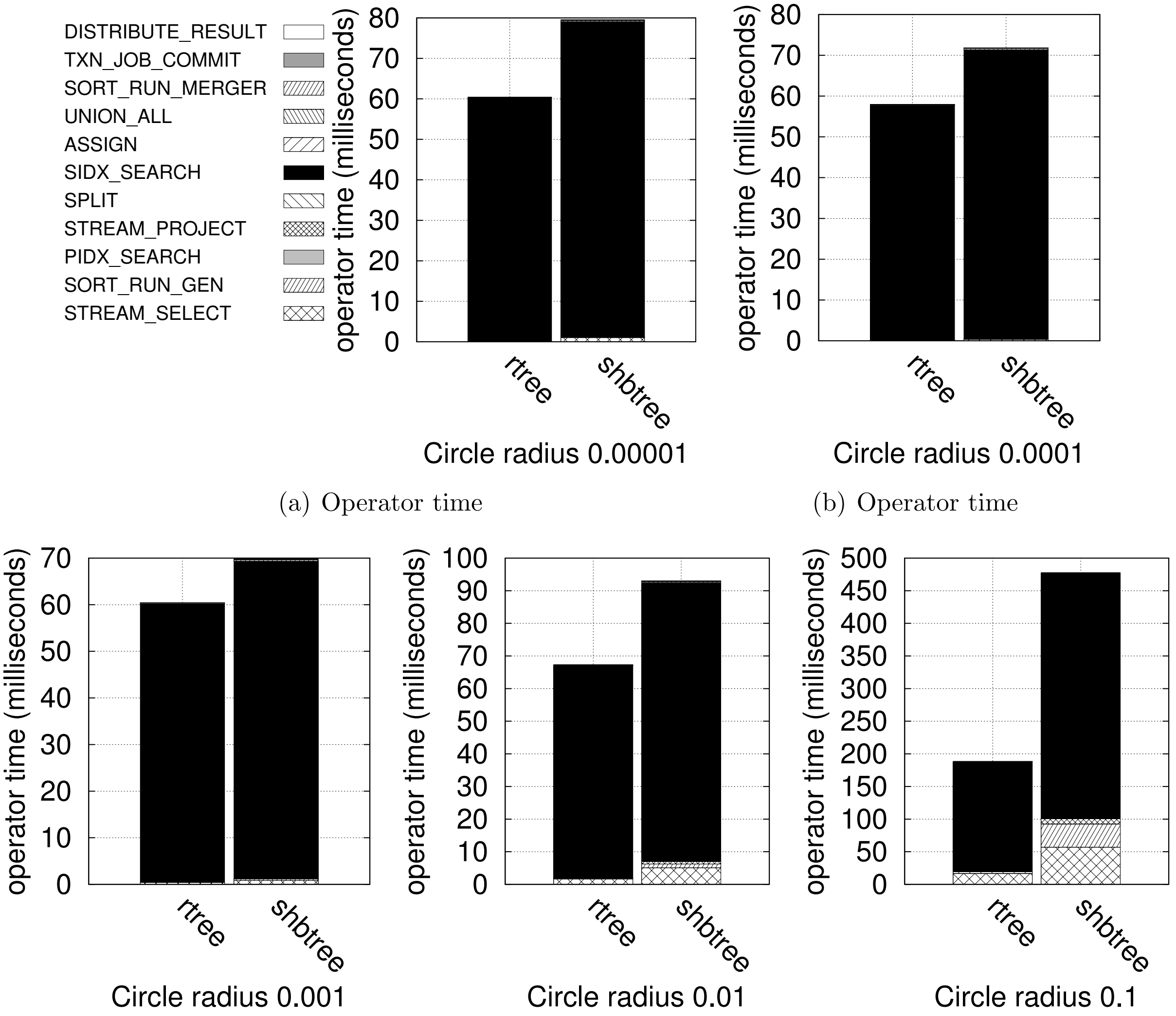


Figure 5.22: Static workload's *index-only-scan* select query's false positive ratio

5.3.6.2 Index-Only-Scan Join Query

The query-response times as a percentage over the R-tree index and the result counts for the index-only-scan join queries are shown in Figures 5.25(a) and 5.25(b). Also, the pro-



(a) Operator time

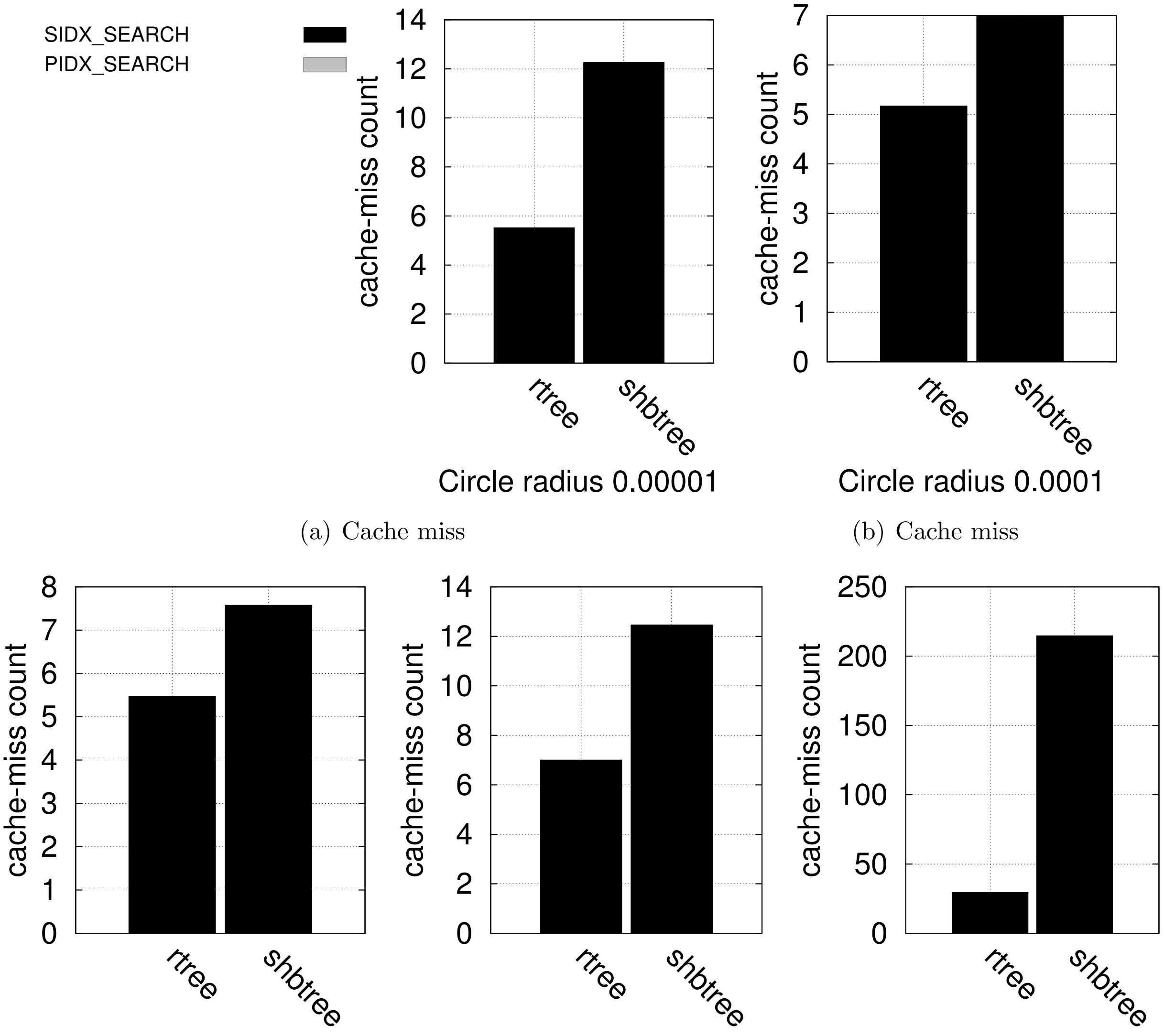
(b) Operator time

(c) Operator time

(d) Operator time

(e) Operator time

Figure 5.23: Static workload's *index-only-scan* select query's operator time



(a) Cache miss

(b) Cache miss

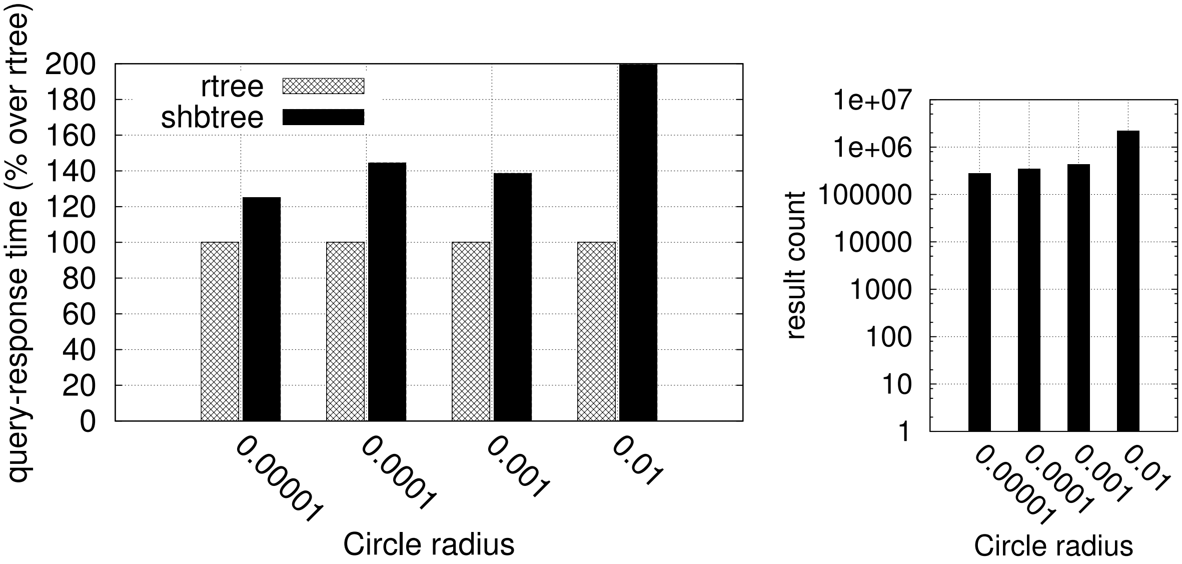
(c) Cache miss

(d) Cache miss

(e) Cache miss

Figure 5.24: Static workload's *index-only-scan* select query's cache-miss count

file information, such as the false-positive ratio, operators' elapsed times, and cache-miss count, is shown in Figures 5.26–5.28. In the join operations, the outer table's (QuerySeedTweet) primary-index search operator is denoted as `OUTER_PIDX_SEARCH`, and the inner table's (Tweet) primary and secondary indexes are denoted as `INNER_PIDX_SEARCH` and `INNER_SIDX_SEARCH`, respectively.



(a) Join response time % over R-tree

(b) Join result count

Figure 5.25: Static workload's *index-only-scan* join query's response time percentage over R-tree and result count

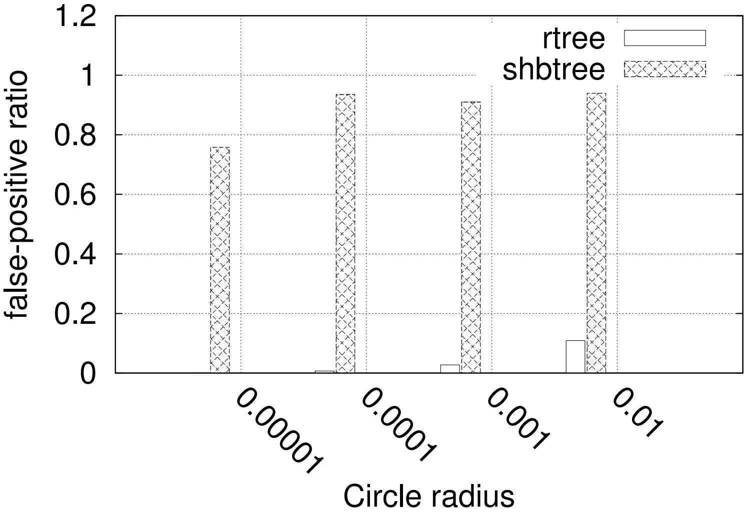
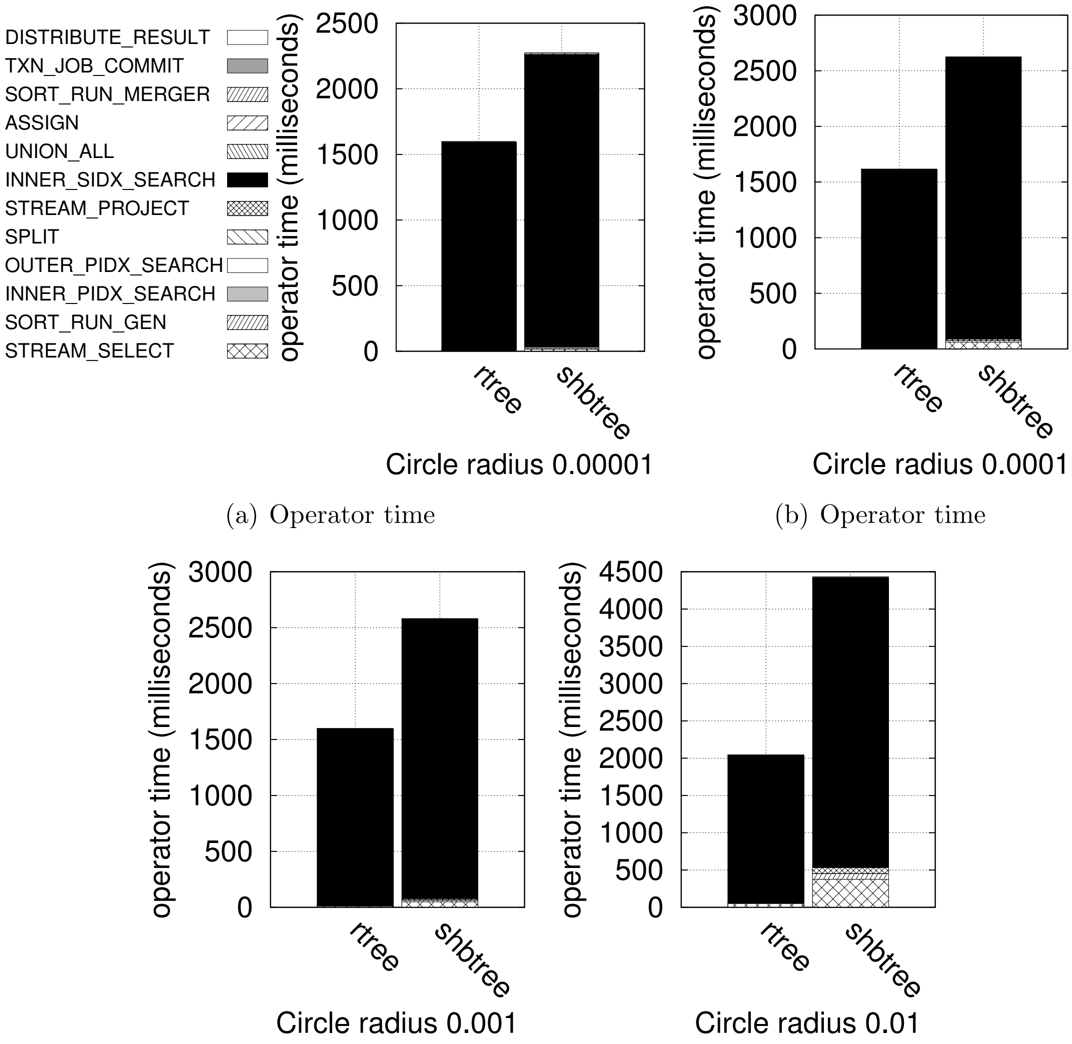


Figure 5.26: Static workload's *index-only-scan* join query's false positive ratio



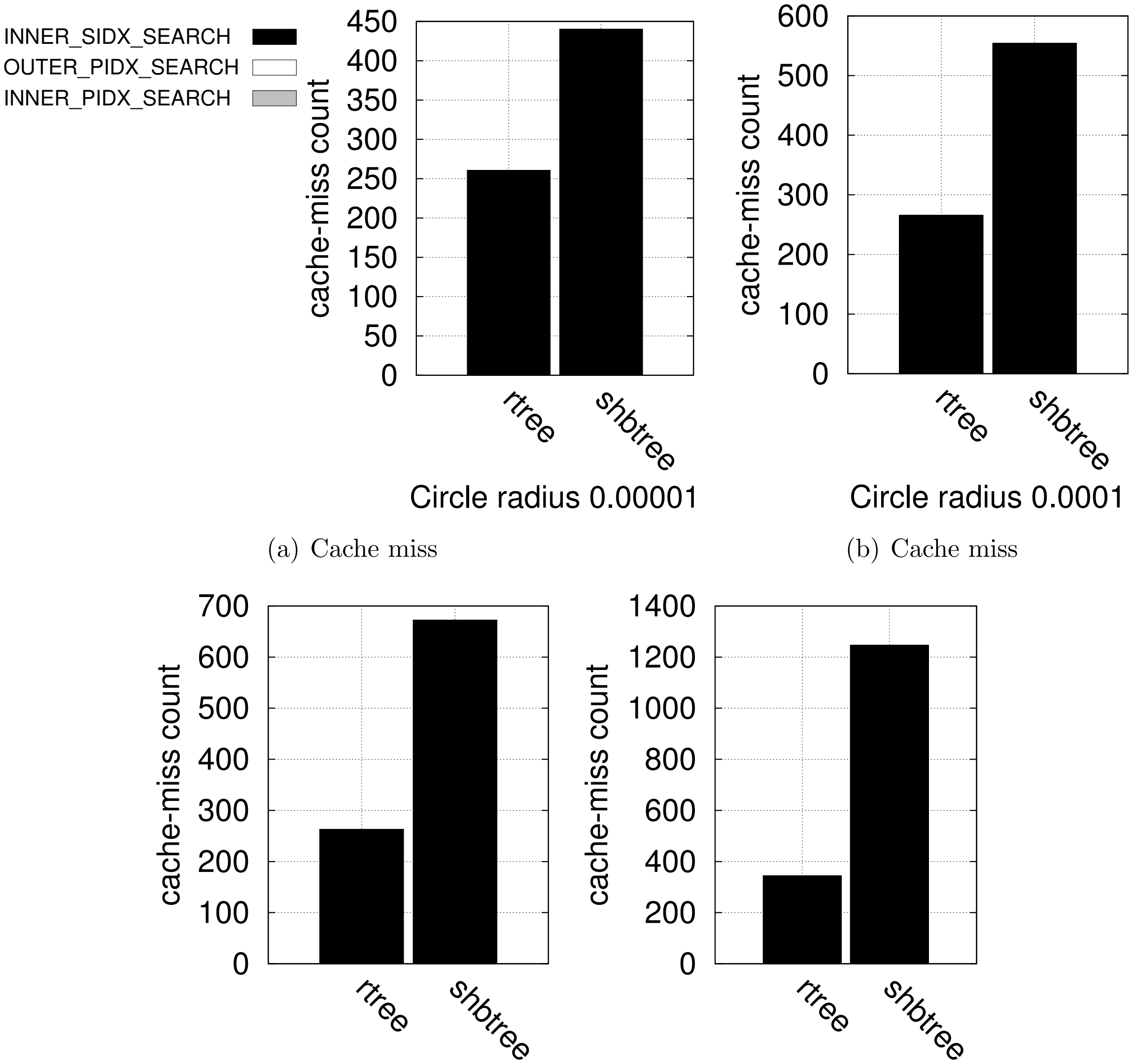
(a) Operator time

(b) Operator time

(c) Operator time

(d) Operator time

Figure 5.27: Static workload's *index-only-scan* join query's operator time



(a) Cache miss

(b) Cache miss

(c) Cache miss

(d) Cache miss

Figure 5.28: Static workload's *index-only-scan* join query's cache-miss count

5.3.6.3 Non-Index-Only-Scan Select Query

Figure 5.29 shows the query-response time as percentage over the R-tree. Also, the profile information such as the operators' elapsed times and the cache-miss count is shown in Figures 5.30 and 5.31, where the black bars and gray bars represent measured values for the secondary spatial index and the primary index, respectively. We again omit the result counts and false-positive ratios here since they are exactly the same as in the index-only-scan select query case.

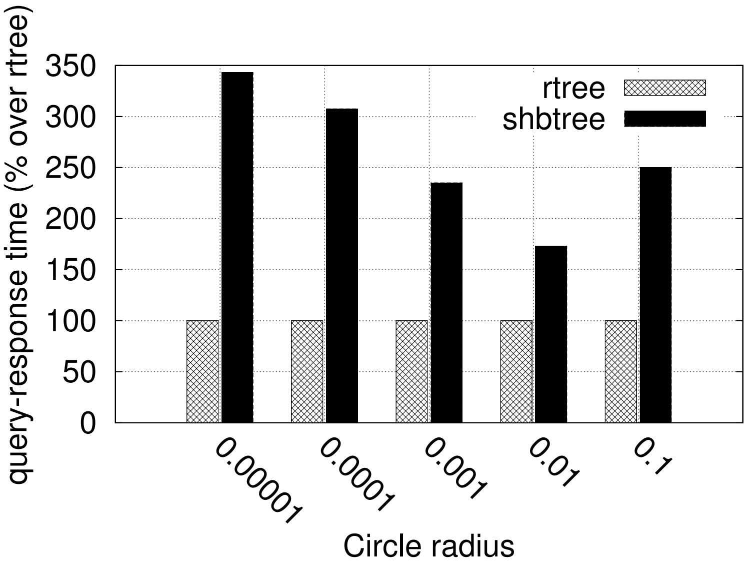
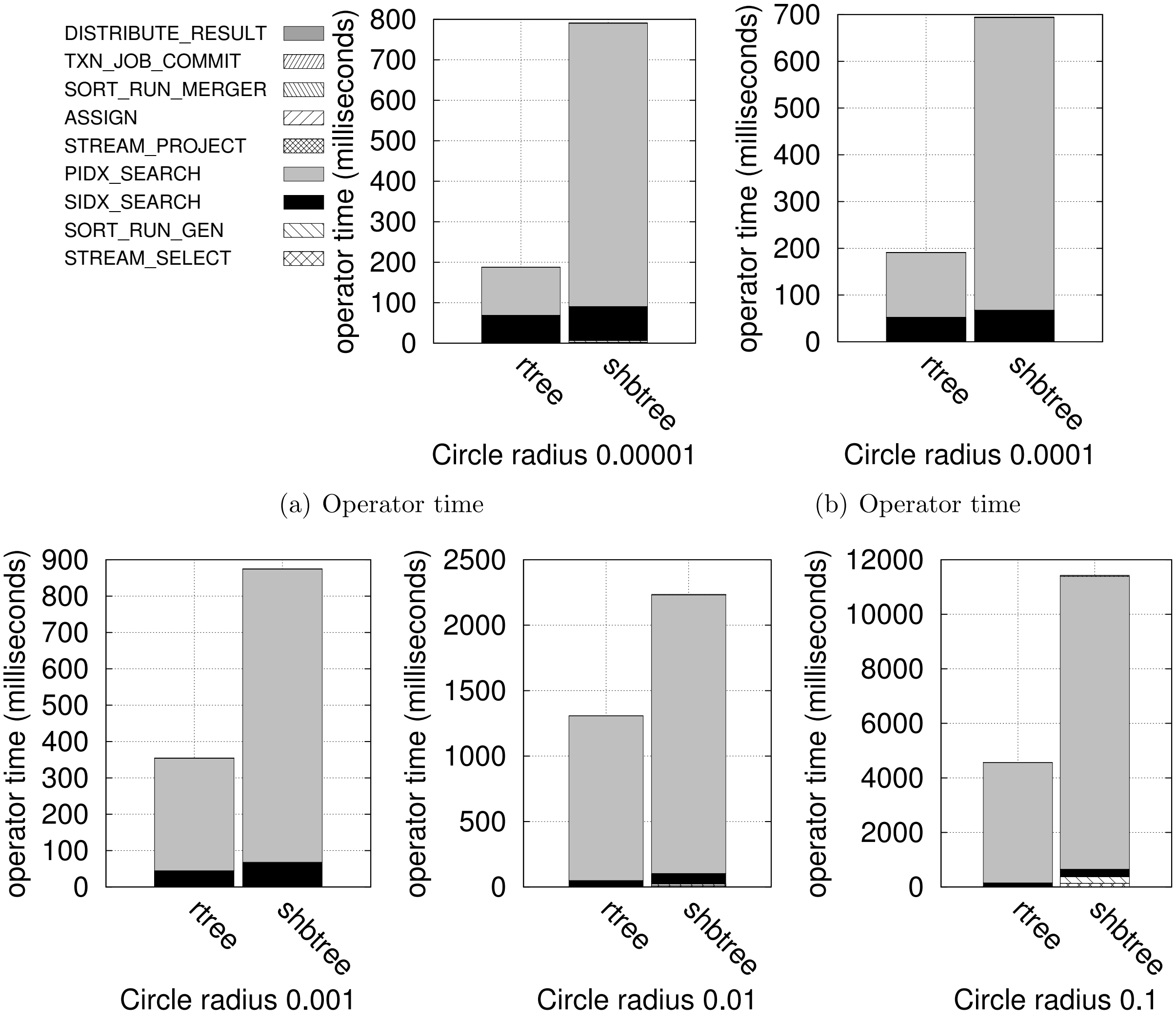


Figure 5.29: Static workload's *non-index-only-scan* select query's response time percentage over R-tree

5.3.6.4 Non-Index-Only-Scan Join Query

The results of the non-index-only-scan join query are provided next. Figure 5.32 shows the query-response time as a percentage over the R-tree. Also, the profile information such as the operators' elapsed times and the cache-miss count is shown in Figures 5.33 and 5.34, where the black bars and the gray bars represent the inner table's secondary spatial index and the inner table's primary index, respectively.



(a) Operator time

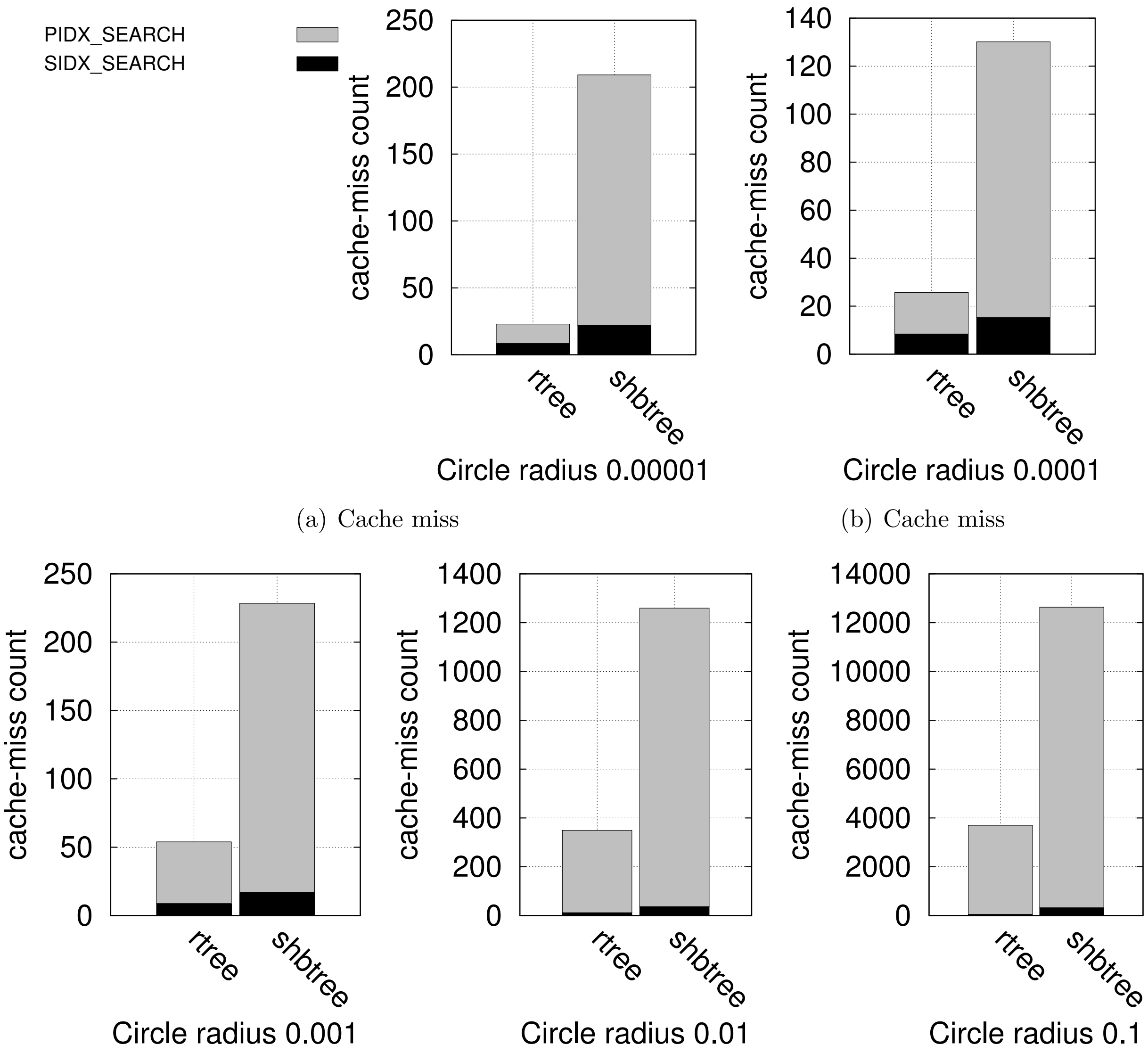
(b) Operator time

(c) Operator time

(d) Operator time

(e) Operator time

Figure 5.30: Static workload's *non-index-only-scan* select query's operator time



(a) Cache miss

(b) Cache miss

(c) Cache miss

(d) Cache miss

(e) Cache miss

Figure 5.31: Static workload's *non-index-only-scan* select query's cache-miss count

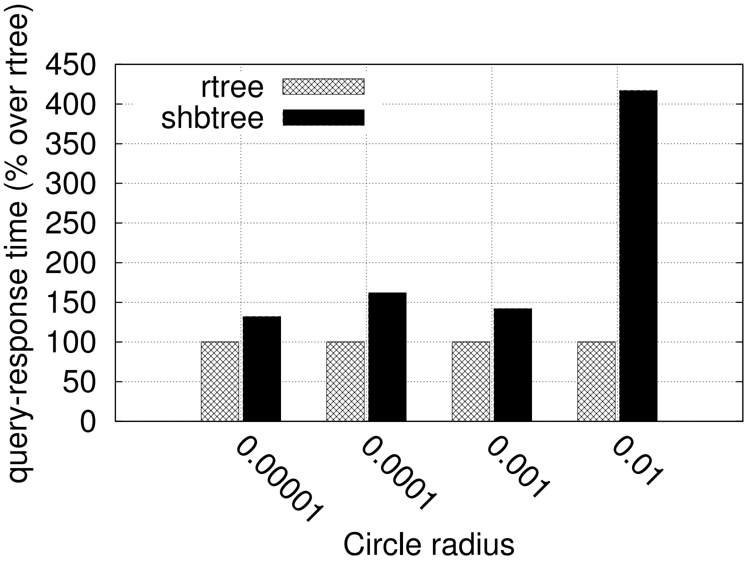


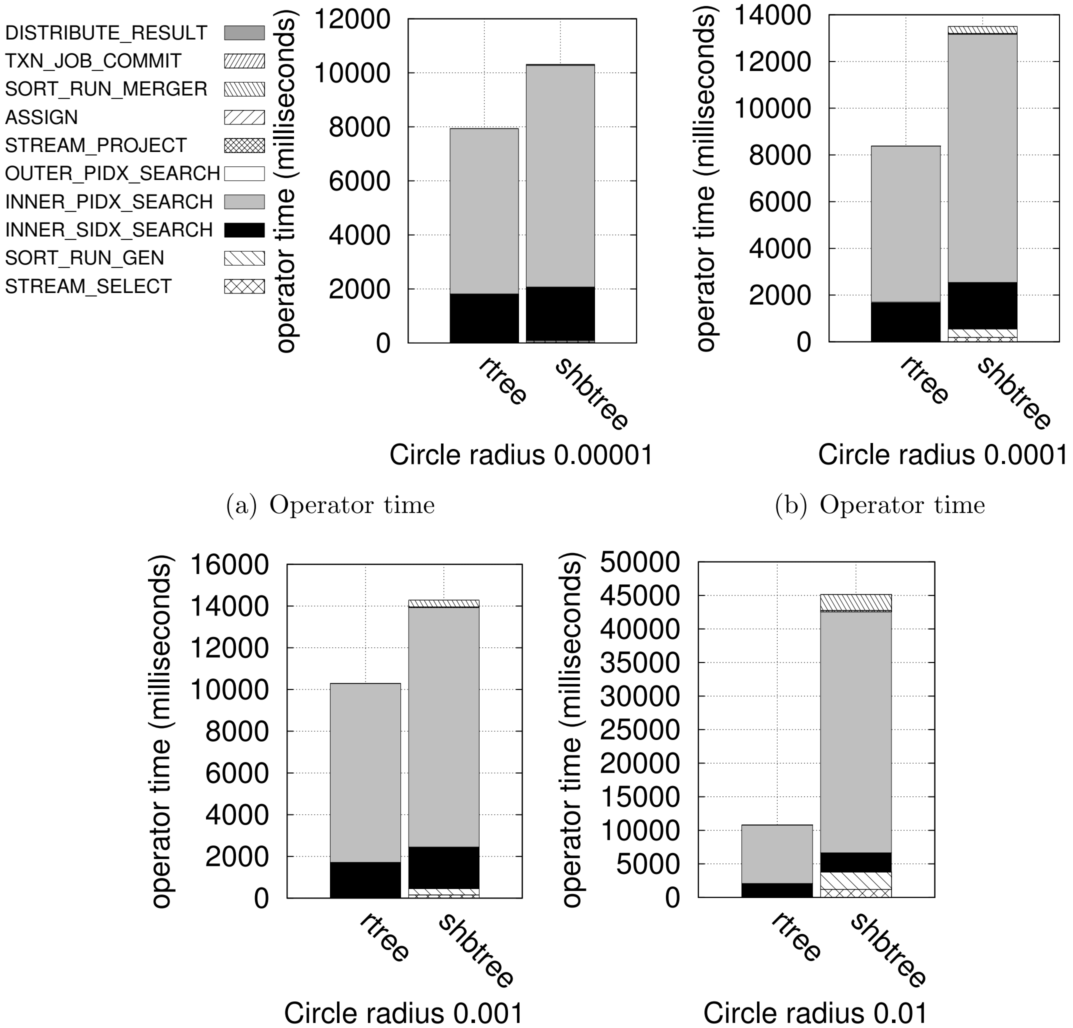
Figure 5.32: Static workload's *non-index-only-scan* join query's response time percentage over R-tree

5.3.7 Dynamic Workload 1 with the Lake-Tweet Dataset

Dynamic workload 1 tests the scalability of each index in terms of IPS (inserts per second) for one hour of data ingestion. Figure 5.35(a) shows the IPS results while varying the number of nodes. As the number of nodes increases, the IPS rate increases linearly for both types of indexes. The primary and secondary index sizes resulting from one hour of ingestion are shown in Figures 5.35(b) and 5.35(c), respectively. In addition, Figure 5.35(d) shows a series of instantaneous IPSs measured every five seconds during the one-hour ingestion from a single node.

5.3.8 Dynamic Workload 2 with the Lake-Tweet Dataset

Dynamic workload 2 tests the indexes' ability to ingest and query concurrently while the ingestion rate is varied. Figure 5.36 shows the results of the index-only-scan query case for this workload, where rate 1, rate 2, rate 3, and rate 4 represent making the tweet generators sleep for 1 millisecond after every 1, 10, 100, and 1000 generated record(s), respectively.



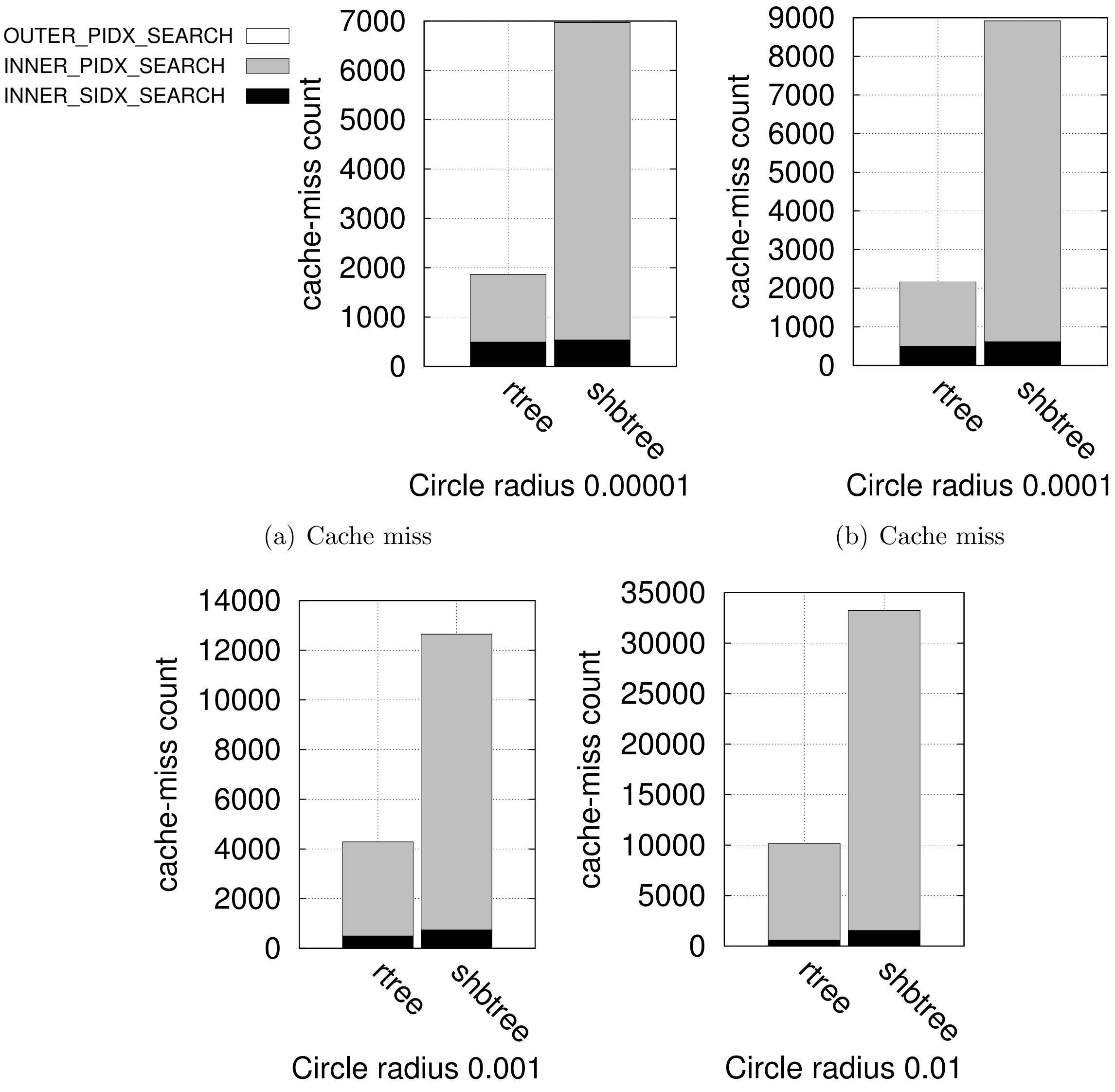
(a) Operator time

(b) Operator time

(c) Operator time

(d) Operator time

Figure 5.33: Static workload's *non-index-only-scan* join query's operator time



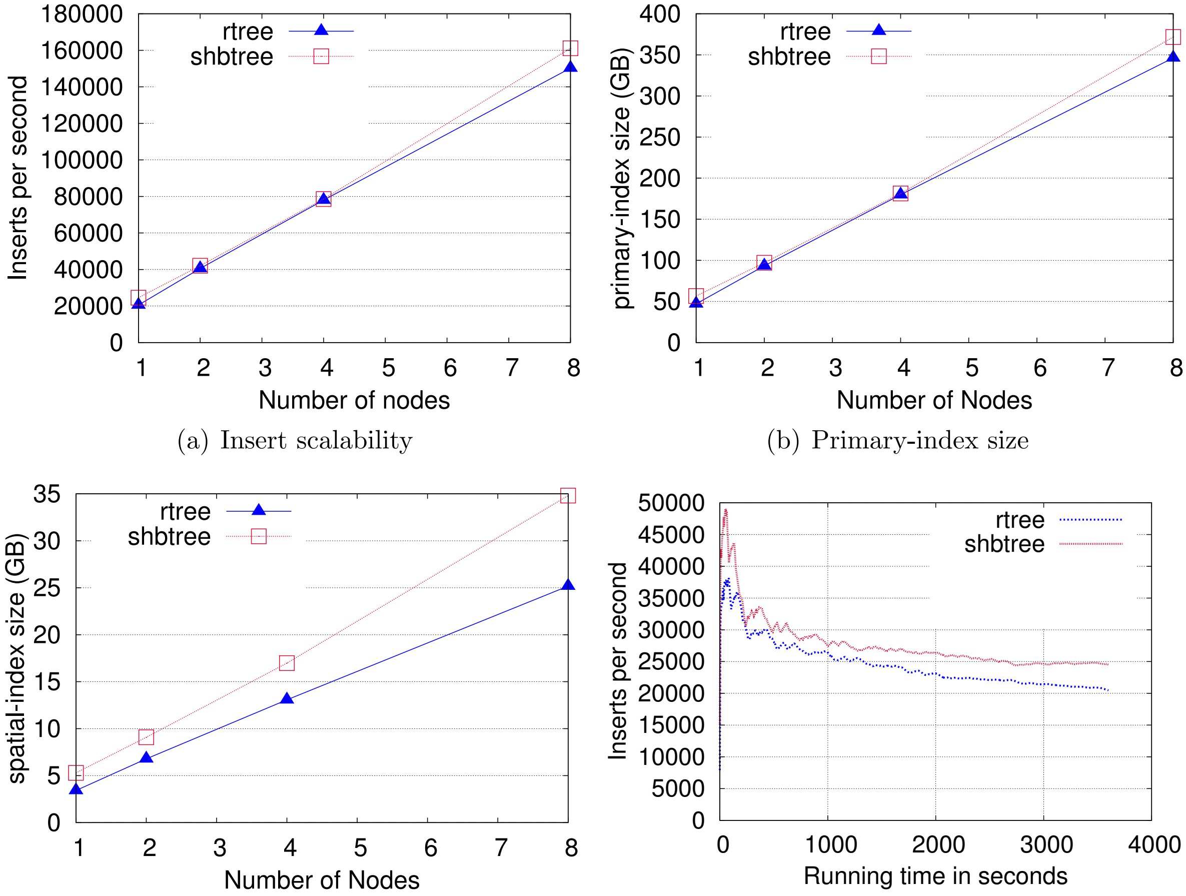
(a) Cache miss

(b) Cache miss

(c) Cache miss

(d) Cache miss

Figure 5.34: Static workload's *non-index-only-scan* join query's cache-miss count



(a) Insert scalability

(b) Primary-index size

(c) Spatial-index size

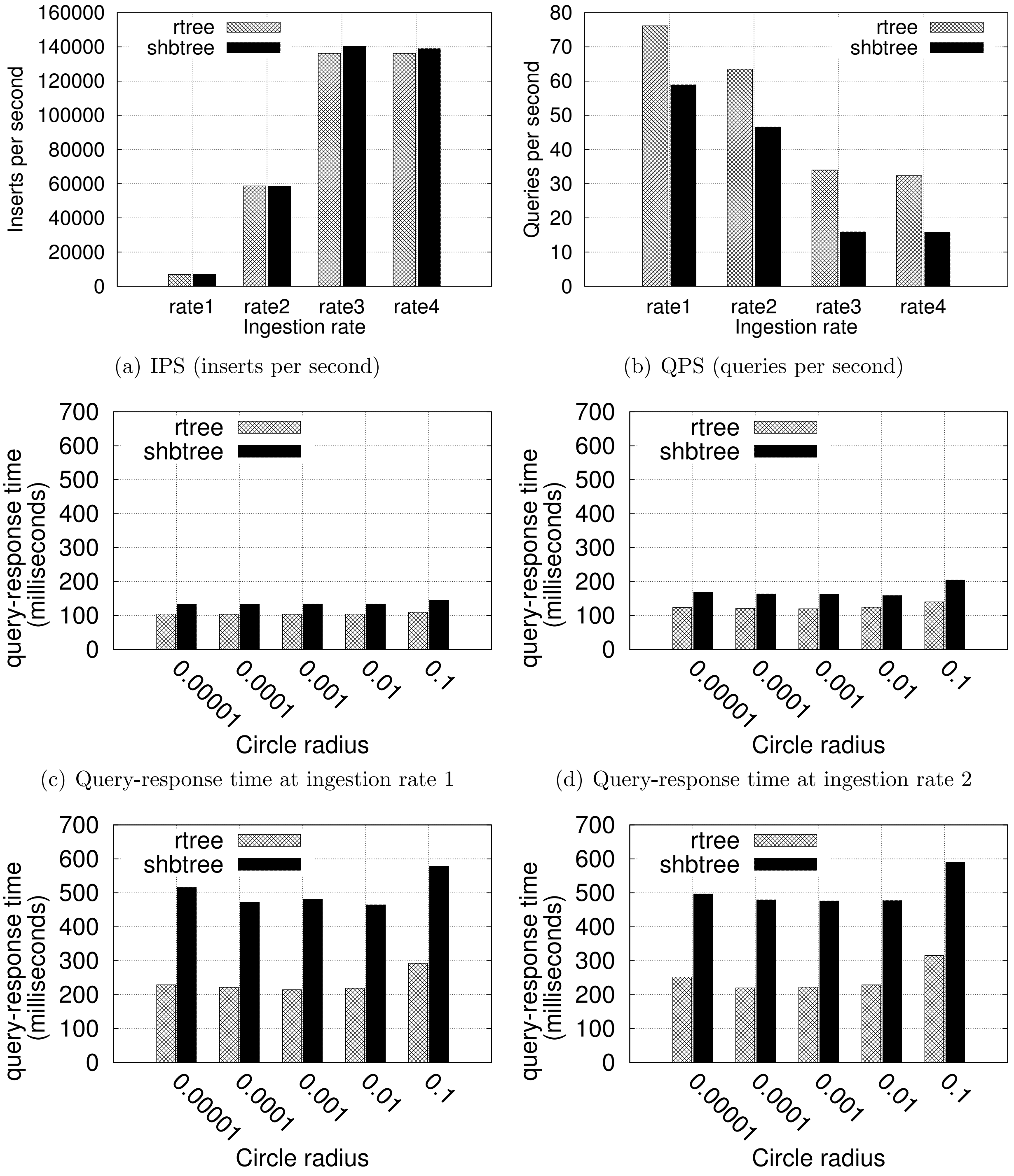
(d) IPS every five seconds

Figure 5.35: Dynamic workload 1

Figure 5.36(a) shows the IPS results for both spatial indexes for each ingestion rate after one hour of ingestion while concurrent queries are being processed. Figure 5.36(b) shows the corresponding QPS (queries per second) results. Similarly, Figure 5.37 shows the results of the non-index-only-scan query case for this workload.

5.4 Related Work

A recent study [64] proposed a mapping-based spatial indexing scheme called Size Separation Indexing (SSI) which is a variant of the space-driven spatial index. The key idea of SSI is as follows. SSI indexes separate non-point spatial objects into multiple separate grids based on the objects' sizes. Each SSI grid decomposes a given space into a set of cells, and each cell is numbered with a space filling curve of its own order. Thus, the numbers of cells in each grid are different. With this setup, SSI then stores the centroids of the spatial objects as keys in the index without including the objects' shapes. To process a spatial query with this setup, a given query area is expanded to include all overlapping centroids of the indexed objects to avoid *false negatives*; the query is evaluated once for each grid and the query area is expanded differently for each grid considering the min and max size of the objects that can be captured in each grid. The study in [64] compared the proposed scheme with other existing spatial indexing methods including R*-tree. However, the study only dealt with static workloads, i.e., once the data were loaded, there were only queries without incremental updates in the workloads. In addition, since the proposed indexing scheme exploits the size distribution of the spatial objects in order to separate the objects effectively in multiple separate grids, the size distribution of the spatial objects needs to be available and considered before the index is built. Thus, while interesting, this approach seems not to be easily adaptable for incrementally updated dynamic data once the index is initially built.



(a) IPS (inserts per second)

(b) QPS (queries per second)

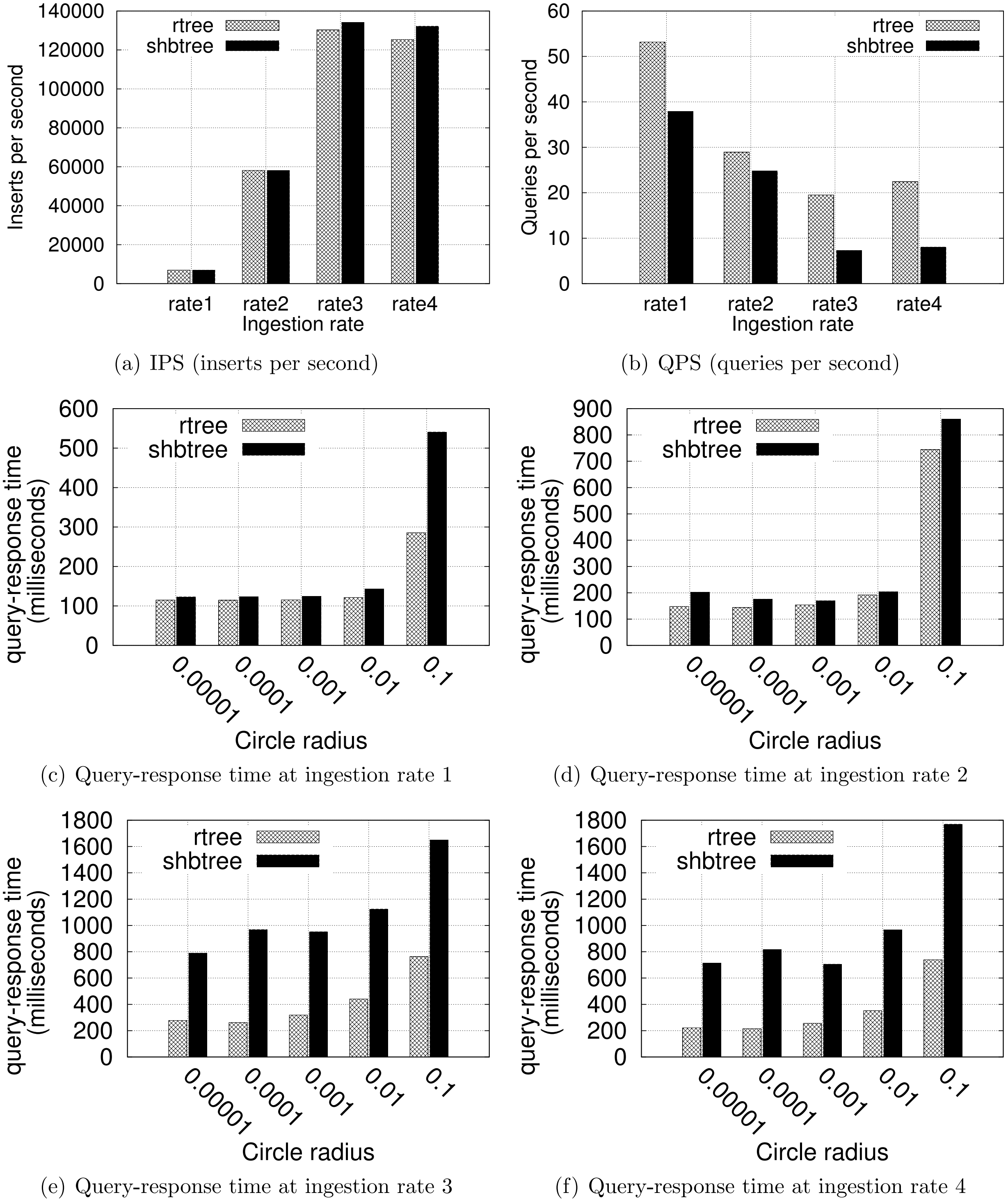
(c) Query-response time at ingestion rate 1

(d) Query-response time at ingestion rate 2

(e) Query-response time at ingestion rate 3

(f) Query-response time at ingestion rate 4

Figure 5.36: Results of dynamic workload 2 for the *index-only-scan* query case, where rate 1, rate 2, rate 3, and rate 4 represent making tweet generators sleep for 1 millisecond after every 1, 10, 100, and 1000 generated record(s), respectively.



(a) IPS (inserts per second)

(b) QPS (queries per second)

(c) Query-response time at ingestion rate 1

(d) Query-response time at ingestion rate 2

(e) Query-response time at ingestion rate 3

(f) Query-response time at ingestion rate 4

Figure 5.37: Results of dynamic workload 2 for the *non-index-only-scan* query case, where rate 1, rate 2, rate 3, and rate 4 represent making tweet generators sleep for 1 millisecond after every 1, 10, 100, and 1000 generated record(s), respectively.

5.5 Conclusions

In this chapter, we have extended Chapter 4's comparative study of LSM spatial indexes to evaluate their performance for dynamic non-point data. For this extended study, the LSM R-tree index and the LSM SHB-tree index were included. First, we explained how the SHB-tree index should be modified to support non-point data. We then implemented the modified SHB-tree index and empirically compared the alternatives with two different characteristics of datasets that had non-point data whose sizes followed two very representative distributions, Gaussian and Zipf distributions, that can be commonly observed in the real world and whose sizes are reasonably large enough to be treated as non-point spatial objects. Considering this purpose, one dataset was generated to include rectangular spatial objects whose sizes followed the Gaussian distribution of house sizes in the US and another dataset included rectangles whose sizes followed the Zipf distribution of the world's lake sizes. The experiments included three types of workloads—a query-only workload, an insert-only workload, and a concurrent insert-and-query workload—to evaluate the LSM spatial indexes's performance for both static and dynamic non-point data use cases.

In the experiments, we first conducted preliminary experiments for the SHB-tree index to find a sweet spot for the number of cells in the grid hierarchy so that the SHB-tree can have fewer overlapping cells for non-point objects and fewer false positives for a given query region. Although the SHB-tree index was configured to use the empirically found sweet spot, due to much higher false-positive ratio, the SHB-tree index showed inferior query performance relative to the R-tree index in terms of both the query-response time and queries per second metrics. For the ingestion performance, the SHB-tree index outperformed the R-tree index, but this performance difference was very marginal. Overall, based on these experimental results, we can conclude that the LSM R-tree index is the winner for dealing with dynamic non-point data. Moreover, given its solid showing in Chapter 4 as well, it seems that the LSM R-tree index would be "the" index to choose—if one wanted to choose just one—for inclusion in a Big Data storage system.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Over the past decade, the proliferation of mobile devices has generated a variety of data at an unprecedented rate. Such data include signals, texts, photos, and videos that are tagged with date, time, and geo coordinates. The data are structured, semi-structured, or not structured at all. The big successes of major social media services such as Facebook, Twitter, Foursquare, Flickr, Instagram, Youtube, Vimeo, Google Photos, have been spurring such data generation. This trend will be further accelerated by the advent of the Internet-of-Things (IoT) era, where literally everything could involve a variety of sensors and generate such data.

Data processing systems that aim to ingest, store, index, and analyze Big Data must deal with such data efficiently. In response, we have developed Apache AsterixDB, a parallel, semi-structured Big Data Management System (BDMS), that provides the ability to ingest, store, index, query, and analyze mass quantities of data. AsterixDB employs a wholly-adopted LSM storage layer due to LSM's superior insert performance as compared to traditional in-place update storage structures.

In this thesis, we have developed a record-level transaction model to support the storage and indexing of such big and fast data as quickly as possible and to make them queryable in a timely manner with certain ACID guarantees. Also, considering the value of exploiting geo-tagged data, it is also critical to store, index, and query geo-tagged data in a timely manner. We have thus done a thorough empirical comparative study of LSM spatial indexes in order to evaluate their pros and cons in light of the dynamic characteristics of high volume geo-tagged data. Our contributions are summarized as follows.

In Chapter 3, we have explained the design for record-level transactions in AsterixDB including how logging and recovery work and how concurrency control is supported. These fine-granularity transactions enable records to be modified and committed as soon as possible by reducing the amount of work to be done in each transaction, thus making them queryable in a timely manner. Also, based on the read-committed isolation level, AsterixDB allows queries to read records that are new or updated even after the queries have begun. Unlike typical NoSQL systems, AsterixDB maintains the consistency between a primary index and any number of secondary indexes for a dataset. All secondary indexes are partition-local to the primary index, so all record-level transactions are local transactions that do not suffer from the two phase commit (2PC) overhead that would be required for the atomicity of distributed transactions. In addition, a deadlock-free locking protocol has been achieved by leveraging the record-level nature of the transaction model, which made it possible to avoid the hold-and-wait situation that is a necessary condition to form a deadlock. Moreover, we have provided an effective solution to support index-only-scan queries in AsterixDB.

Considering that major traditional SQL systems provide options for users to pick an appropriate isolation level for their applications, no single isolation level may fit all database applications. We believe that the design for record-level transactions based on the read-committed isolation level in AsterixDB is an appropriate way to provide transactional guarantees to applications that must deal with big and fast data with acceptable performance so that the applications can write and query such data in timely manner.

Chapter 4 explored a set of representative, disk-resident spatial indexing methods that have been adopted by major SQL and NoSQL systems. We implemented five alternative variants of these methods in the form of LSM-based spatial indexes in Apache AsterixDB—DHB-tree, DHVB-tree, R-tree, SHB-tree, and SIF—in order to evaluate their pros and cons in light of the dynamic characteristics of high-volume, geo-tagged point data. We have empirically examined the performance of these five spatial indexes in terms of their query response time, inserts per second, and queries per second using three types of workloads: a query-only workload, an insert-only workload, and a concurrent insert-and-query workload. Based on the performance results, we discussed the pros and cons of the five indexes including the effects of their different underlying bases, i.e., B-tree, R-tree, and inverted index foundations.

Overall, except for the SIF index, there is neither one clear winner nor a clear loser when considering both data ingestion and query performance. The query-performance differences for the most part were not large in the setting of a real end-to-end system; this was especially true for large-circle non-index-only-scan queries, where the primary index lookups were costly and dominant. Nonetheless, if we had to choose a winning index based on the evaluation results with a focus on balancing concurrent fast ingestion and querying (a very important workload in the Big Data era), the DHVB-tree index could be the winner if ingestion has a higher priority, and the R-tree index could be the winner if queries has a higher priority in a given application setting.

In Chapter 5, we extended the comparative study of LSM spatial indexes to evaluate their performance for dynamic non-point (polygon) data. For this extended study, the LSM R-tree index and the LSM SHB-tree index were considered. First, we explained how the SHB-tree index can be modified to support non-point data. We then implemented the modified SHB-tree index and empirically compared the two index types with two different datasets. One dataset included rectangular spatial objects whose sizes followed the Gaussian distribution of US house sizes, and another dataset included rectangles whose sizes followed the Zipf distribution of the world's lake sizes. The experiments again included three types of workloads—a query-only workload, an insert-only workload, and a concurrent insert-and-query workload—to evaluate the LSM spatial indexes' performance for both static and dynamic non-point data. In the evaluation, we first conducted experiments for the SHB-tree index to find a sweet spot for the number of cells in the grid hierarchy so that the SHB-tree can balance the number of overlapping cells for non-point objects against the false positives for a given query region. Even though the SHB-tree index was configured to use the resulting sweet spot configuration, due to a much higher false-positive ratio, the SHB-tree index showed notably inferior performance as compared to the R-tree index for both of the workloads studied.

Based on the evaluation results from Chapters 4 and 5, if we had to choose a single overall winning index, the LSM R-tree index would be the winner for the following reasons: First, the R-tree can deal with both dynamic point and non-point spatial objects seamlessly. Second, when concurrent fast ingestion and querying workloads were used, it outperformed all of the other indexes in terms of query performance for point data by 20%, and it outperformed the SHB-tree index for non-point data by more than 100%, both with reasonable ingestion performance. Last, but by no means least, the R-tree index does not require a parameter tuning effort like that required for structures like the SHB-tree index.

Based on this study's results, we have decided to support the LSM R-tree index as "the" spatial index in Apache AsterixDB. We believe that these study results should also be very useful as a valuable guide for other practitioners wishing to use or implement spatial indexing methods in their Big Data systems while considering their pros and cons.

6.2 Future Work

In Chapter 3, we have discussed certain expected but undesired situations that can arise with concurrent workloads due to the read-committed isolation level in AsterixDB. In order to remove such possible undesired situations at the read-committed isolation level, and also considering the LSM storage layer's append-only nature in AsterixDB, it would be interesting to study how the current lock-based concurrency control scheme in AsterixDB could be reshaped to support a multi-version concurrency control scheme by leveraging the system's LSM nature. Also, it could be interesting for AsterixDB to be enhanced to allow users to choose from among several isolation levels considering their applications' purposes, similar to what traditional SQL systems provide.

We saw in Chapter 4 that the LSM R-tree index may not outperform LSM B-tree based spatial indexes in terms of ingestion performance due to the overheads of forming high quality MBRs in the in-memory R-tree component and of sorting the spatial objects based on the Hilbert curve during component flush and merge operations. Considering this, it is tempting to seek ways to improve the LSM R-tree index's ingestion performance so that it can beat other indexes in terms of both its query *and* ingestion speeds. One possible direction could be to compromise on forming high quality MBRs in the in-memory R-tree component since the MBRs of in-memory components are reorganized anyway during the first flush operation.

In Chapters 4 and 5, we have evaluated five candidate LSM spatial indexes in the context of AsterixDB, and as a result, we have decided to support only the LSM R-tree index as the spatial index in AsterixDB. It would now be interesting to compare the performance of spatial query processing based on the LSM R-tree index in AsterixDB with the spatial query performance of other SQL and NoSQL systems under concurrent ingestion and query workloads.

Bibliography

[1] American housing survey for the united states: 2011. https://www.census.gov/content/dam/Census/programs-surveys/ahs/data/2011/h150-11.pdf.

[2] Apache AsterixDB. https://asterixdb.apache.org/.

[3] Mongodb. https://www.mongodb.com/.

[4] OpenStreetMap's bulk GPS point data. https://blog.openstreetmap.org/2012/04/01/bulk-gps-point-data/.

[5] A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* PhD thesis, Cambridge, MA, USA, 1999.

[6] A. Adya, B. Liskov, and P. E. O'Neil. Generalized isolation level definitions. In *ICDE*, pages 67–78, 2000.

[7] R. Agrawal, M. J. Carey, and L. W. McVoy. The performance of alternative strategies for dealing with deadlocks in database management systems. *IEEE Trans. Software Eng.*, 13(12):1348–1363, 1987.

[8] 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. 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. *PVLDB*, 7(14):1905–1916, 2014.

[9] S. Alsubaiee, A. Behm, V. R. Borkar, Z. Heilbron, Y. Kim, M. J. Carey, M. Dreseler, and C. Li. Storage management in AsterixDB. *PVLDB*, 7(10):841–852, 2014.

[10] American National Standards Institute. *American national standard for information systems: database language — SQL: ANSI X3.135-1992.* American National Standards Institute, 1992.

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

[12] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[13] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995.*, pages 1–10, 1995.

[14] P. A. Bernstein and S. Das. Rethinking eventual consistency. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 923–928, 2013.

[15] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[16] D. Borkar, R. Mayuram, G. Sangudi, and M. Carey. Have your data and query it too: From key-value caching to big data management. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 239–251, 2016.

[17] V. R. Borkar, Y. Bu, E. P. C. Jr., N. Onose, T. Westmann, P. Pirzadeh, M. J. Carey, and V. J. Tsotras. Algebricks: a data model-agnostic compiler backend for big data languages. In *Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC 2015, Kohala Coast, Hawaii, USA, August 27-29, 2015*, pages 422–433, 2015.

[18] 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 Conference*, pages 1151–1162, 2011.

[19] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: facebook's distributed data store for the social graph. In *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, pages 49–60, 2013.

[20] M. J. Carey, S. Jacobs, and V. J. Tsotras. Breaking BAD: a data serving vision for big active data. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, DEBS '16, Irvine, CA, USA, June 20 - 24, 2016*, pages 181–186, 2016.

[21] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[22] L. Chen et al. Spatial keyword query processing: An experimental evaluation. *PVLDB*, 6(3):217–228, 2013.

[23] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.

[24] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8, 2013.

[25] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[26] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.

[27] E. W. Dijkstra. Cooperating sequential processes, technical report ewd-123. Technical report, 1965.

[28] J. A. Downing, Y. T. Prairie, J. J. Cole, C. M. Duarte, L. J. Tranvik, R. G. Striegl, W. H. McDowell, P. Kortelainen, N. F. Caraco, J. M. Melack, and J. J. Middelburg. The global abundance and size distribution of lakes, ponds, and impoundments. *Limnology and Oceanography*, 51(5):2388–2397, 2006.

[29] C. Faloutsos and Y. Rong. DOT: A spatial access method using fractals. In *ICDE Conference*, pages 152–159, 1991.

[30] Y. Fang et al. Spatial indexing in Microsoft SQL Server 2008. In *SIGMOD Conference*, pages 1207–1216, 2008.

[31] V. Gaede and O. Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.

[32] I. Gargantini. An effective way to represent quadtrees. *Commun. ACM*, 25(12):905–910, 1982.

[33] J. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In *IFIP Working Conference on Modelling in Data Base Management Systems*, pages 365–394, 1976.

[34] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[35] R. Grover and M. J. Carey. Data ingestion in AsterixDB. In *EDBT Conference*, pages 605–616, 2015.

[36] T. Härder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.

[37] P. Helland. Life beyond distributed transactions: an apostate's opinion. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, pages 132–141, 2007.

[38] R. C. Holt. Some deadlock properties of computer systems. *ACM Comput. Surv.*, 4(3):179–196, 1972.

[39] H. V. Jagadish. Linear clustering of objects with multiple atributes. In *SIGMOD Conference*, pages 332–342, 1990.

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

[41] C. M. Jermaine, E. Omiecinski, and W. G. Yee. The partitioned exponential file for database storage management. *VLDB J.*, 16(4):417–437, 2007.

[42] K. V. R. Kanth, S. Ravada, and D. Abugov. Quadtree and R-tree indexes in oracle spatial: a comparison using GIS data. In *SIGMOD Conference*, pages 546–557, 2002.

[43] A. Khodaei, C. Shahabi, and C. Li. Hybrid indexing and seamless ranking of spatial and textual features of web documents. In *DEXA Conference*, pages 450–466, 2010.

[44] M. Kornacker, C. Mohan, and J. M. Hellerstein. Concurrency and recovery in generalized search trees. In *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA.*, pages 62–72, 1997.

[45] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *Operating Systems Review*, 44(2):35–40, 2010.

[46] J. K. Lawder. *The Application of Space-Filling Curves to the Storage and Retrieval of Multi-dimensional Data.* PhD thesis, Birkbeck College, University of London, 2000.

[47] D. B. Lomet, A. Fekete, G. Weikum, and M. J. Zwilling. Unbundling transaction services in the cloud. In *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*, 2009.

[48] D. B. Lomet, K. Tzoumas, and M. J. Zwilling. Implementing performance competitive logical recovery. *PVLDB*, 4(7):430–439, 2011.

[49] R. A. Lorie. Physical integrity in a large segmented database. *ACM Trans. Database Syst.*, 2(1):91–104, 1977.

[50] C. Mohan. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on b-tree indexes. In *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings.*, pages 392–405, 1990.

[51] C. Mohan, D. J. Haderle, B. G. Lindsay, H. Pirahesh, and P. M. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[52] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD Conference*, pages 1099–1110, 2008.

[53] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.*, 33(4):351–385, 1996.

[54] K. W. Ong, Y. Papakonstantinou, and R. Vernoux. The SQL++ semi-structured data model and query language: A capabilities survey of sql-on-hadoop, nosql and newsql databases. *CoRR*, abs/1405.3631, 2014.

[55] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *PODS Conference*, pages 181–190, 1984.

[56] O. Procopiuc, P. K. Agarwal, L. Arge, and J. S. Vitter. Bkd-Tree: A dynamic scalable kd-tree. In *SSTD Conference*, pages 46–65, 2003.

[57] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[58] J. T. Robinson. The k-d-b-tree: A search structure for large multidimensional dynamic indexes. In *SIGMOD Conference*, pages 10–18, 1981.

[59] H. Samet. *Foundations of Multidimensional and Metric Data Structures.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[60] R. Sears and R. Ramakrishnan. bLSM: a general purpose log structured merge tree. In *SIGMOD Conference*, pages 217–228, 2012.

[61] D. G. Severance and G. M. Lohman. Differential files: Their application to the maintenance of large databases. *ACM Trans. Database Syst.*, 1(3):256–267, 1976.

[62] A. Thomson, T. Diamond, S. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 1–12, 2012.

[63] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.

[64] R. Zhang, J. Qi, M. Stradling, and J. Huang. Towards a painless index for spatial objects. *ACM Trans. Database Syst.*, 39(3):19:1–19:42, 2014.