A Paradigm for Scalable, Transactional, and Efficient Spatial Indexes

by

Ning Gao

B.A., Zhejiang University, 2012

M.S., University of Colorado at Boulder, 2016

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2018

This thesis entitled:
A Paradigm for Scalable, Transactional, and Efficient Spatial Indexes
written by Ning Gao
has been approved for the Department of Computer Science

_____
Dirk Grunwald

_____
Prof. Qin Lv

Date _____

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Gao, Ning (Ph.D., Computer Science)

A Paradigm for Scalable, Transactional, and Efficient Spatial Indexes

Thesis directed by Prof. Dirk Grunwald

With large volumes of geo-tagged data collected in various applications, spatial query processing becomes essential. Query engines depend on efficient indexes to expedite processing. There are three main challenges: scaling out to accommodate large volumes of spatial data, supporting transactional primitives for strong consistency guarantees, and adapting to highly dynamic workloads. **This thesis proposes a paradigm for scalable, transactional, and efficient spatial indexes to significantly reduce development efforts in designing and comparing multiple spatial indexes.**

This thesis first introduces a distributed and transactional key value store called DTranx to persist the spatial indexes. DTranx follows the SEDA architecture to exploit high concurrency in multi-core environments and it adopts a hybrid of optimistic concurrency control and two-phase commit protocols to narrow down the critical sections of distributed locking during transaction commits. Moreover, DTranx integrates a persistent memory based write-ahead log to reduce durability overhead and combines a garbage collection mechanism without affecting normal transactions. To maintain high throughput for search workloads when databases are constantly updated, snapshot transactions are introduced.

Then, a paradigm is presented with a set of intuitive APIs and a Mempool runtime to reduce development efforts. Mempool transparently synchronizes local states of data structures with DTranx and it handles two critical tasks: address translation and transparent server synchronization, of which the latter includes transaction construction and data synchronization. Furthermore, a dynamic partitioning strategy is integrated into DTranx to generate partitioning and replication plans that reduce inter-server communications and balance resource usage.

Lastly, single-threaded data structures BTree and RTree are converted into distributed ver-

sions within two weeks. The BTree and RTree achieve 253.07 kops/sec and 77.83 kops/sec throughput respectively for pure search operations in a 25-server cluster.

Dedication

To all of families and friends.

Acknowledgements

I'm thankful to my thesis advisor Dirk Grunwald who has guided me through difficulties. Without his guidance and inspiration, I would not have been able to complete this dissertation. Dirk is a very smart and knowledgeble scholar and keeps a humble attitude to listen to others. Besides being a well informed advisor, Dirk is patient and encouraging. His metaphor of research as the Japanese dorodango inspired me to persevere and perfect my work. Thanks are also extended to my committee members: Eric Keller, Qin Lv, Shivakant Mishra, and Sriram Shivakant for their precious feedback.

I would like to thank my colleagues for insightful research discussion. This includes Hansu Gu, Chenyu Zheng, Tian Lei, Bryan Dixon, Blake Caldwell, Guohui Ding, Zhang Liu, Yuru Li, Mohammad Hashemi, Zhiyuan Liu, Mahnaz Roshanaei, and Shuo Zhang. They created a joyful and pleasant environment in our system lab. I wish the best luck to their future careers.

I have been fortunate to meet a lot of friends in Boulder. I thank Xinyang Zhou, Ke Ma, Yuanzhe Zhang, Zhanan Zou, Simeng Chen, Hang Yin, Tong Shen, Mengyuan Wang, Xizheng Ma, Feichi Feng, Haojie Hang, Xinshuo Yang, Ze Chen. I've enjoyed sharing life moments, climbing, snowboarding, hiking, and running with them. Especially, Xinyang Zhou, Ke Ma, Yuanzhe Zhang, and I started our friendship since 2012 and they helped me grow, kept me accompany, and shared life moments. Without them, my graduate life would never be like the same.

Finally, I would like to express my deep appreciation to my families. My parents Eryun Gao and Minjuan Lin have always been there to encourage and support me during difficult times. My wife Yuan Sui stood besides me throughout my graduate life and she has been both a caring

wife and a great friend. She has brought me happiness, encouragement, and love that makes me stronger.

Contents

Chapter

Tables

Table

Figures

Figure

Chapter 1

Introduction

1.1 Motivation

As geo-sensors are widely available and the volumes of geo-tagged data are collected in an unprecedented pace, numerous applications arise in demand of an indexing service to handle various spatial queries. For example, since the FCC opened unlicensed spectrum for public usage, a spectrum registration system, which regulates and coordinates the spectrum usage, needs to answer the question: whether a spectrum range A in location X collides with any existing hotspot. A spatial index that handles range queries is required. Another example is the Uber taxi service that collects drivers' and passengers' location information and searches for nearest drivers or passengers. Uber depends on a spatial index for nearest neighbor queries. Services like these have boomed during the last decade, necessitating spatial indexes to deal with various spatial queries. Since the 1970s, researchers have exploited multidimensional access methods, such as RTree, KD-Tree, etc. However, current applications call for a more scalable solution. For instance, Uber claimed[1] that it reached 15 million daily rides, 3 million drivers, and 75 million riders. Besides the large numbers of daily rides, these Uber drivers and riders frequently updated their real-time locations, causing high volumes of both query and update requests to the Uber database.

Spatial distributed data structures were heavily explored to scale the multidimensional access methods. There are two categories: monolithic indexes and hierarchical indexes. Monolithic indexes store the spatial indexes in distributed storage systems, such as HBase and Cassandra, to scale out.

[1] According to the statistics published in https://www.uber.com/newsroom/company-info/.

For example, MD-HBase [128] persisted a KD-Tree in HBase and spatial queries are processed in any MD-HBase server in parallel. Hierarchical indexes divide the spatial index structures into partitions and distribute the partitions across the servers. For instance, SD-RTree [49] proposed a distributed RTree and DiST [120] introduced a distributed KD-Tree. However, neither monolithic nor hierarchical indexes offer transactional primitives, which are notoriously costly in databases. Without transactional support, the aforementioned spectrum registration system might accept multiple concurrent requests, of which the covered regions overlap with one another. Furthermore, spatial systems naturally generate dynamic workloads, undermining load balances. If a subset of the cluster contains all the heavily accessed data, the overall throughput would be limited by these servers. In the worst case where all requests need to access a particular data item, the throughput of a cluster could be less than that of one server.

Thesis Statement: This thesis attempts to build scalable, transactional, and efficient spatial indexes and provide a paradigm such that researchers are able to design and compare various spatial indexes with minimum development efforts.

It aims to address all the challenges above. First, the paradigm enables researchers to convert central multidimensional access methods to distributed spatial indexes efficiently such that the converted distributed spatial indexes can be compared and combined to handle various kinds of spatial queries. Second, transactional primitives are provided for the developers to reason their program correctness and the internal design guarantees high throughput and low latency. Lastly, data partitioning are adapted dynamically to the current workloads to achieve better scalability.

1.2 System Overview

We follow the client-server model and offload scalability and transaction burdens to a distributed storage system. The overall architecture is shown in Figure 1.1. Clients offer a runtime and a library to facilitate the constructions of various spatial data structures and servers run a distributed key value store that supports high scalability and serializable transactions. The client-server model brings two benefits. First, the separation of the data structure logic from the storage

system enables us to leverage techniques from distributed databases to achieve low latency, high throughput, and linear scalability. Second, it simplifies DDS implementations by hiding distributed system complexities, such as data placements.

Client To minimize developing efforts, we introduce an intuitive abstraction and a Mempool runtime to consistently synchronize data between local machines and servers transparently. Developers decompose data structures into basic data unit that we call data bricks. For example, the data bricks of binary trees are tree nodes. Mempool calculates a distinct address for each data brick automatically. With transparent server synchronization and address translation, the conversion from single-threaded data structures to distributed versions becomes effortless while the transactional interface ensures consistency.

Server A distributed key value store is deployed on the servers. We implement DTranx instead of document databases or graph databases for two reasons. First, the mapping of data bricks to keys is straightforward, reducing address translation overhead. Second, distributed key value stores have been explored extensively in the last decade and numerous partitioning algorithms have been proposed, such as consistent hashing and content addressable network. Not only does DTranx implement efficient distributed transactions, it also integrates a dynamic partitioning strategy for load balance.

1.3 Contributions

The contributions of the thesis are summarized into four aspects as follows.

Transactional KV Store We build a scalable and transactional key value store from scratch. First, Staged Event-Driven Architecture(SEDA) is adopted to increase the system concurrency and an optimal binding from threads to physical cores is searched. On top of that, lock free queues connect SEDA stages as the communication channels to reduce the contention among threads. Second, A hybrid commit protocol that combines optimistic concurrency control and two phase commit is customized to achieve efficient serializable transactions. Moreover, we avoid deadlocks and livelocks with a customized locking mechanism. Third, A modular Write-Ahead

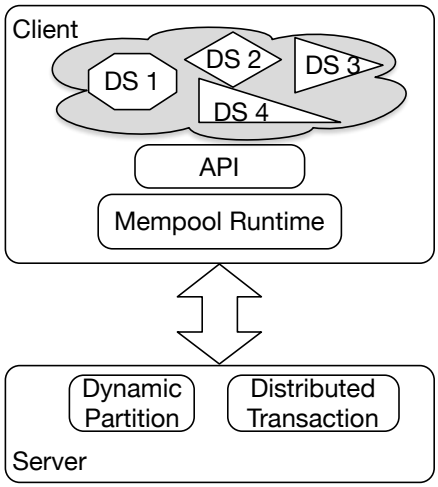


Figure 1.1: Client-Server Model

Log(WAL) based on Non-Volatile Memory cuts down the durability costs considerably, from which most distributed ACID-compliant systems suffer; And, a garbage collection mechanism is integrated to reclaim log space without affecting transaction performance. Lastly, snapshot read transactions are proposed to bypass distributed commit protocols. It maintains high throughput even when update transactions are constantly modifying the databases.

Paradigm We design the Mempool runtime and a set of intuitive APIs on the client side to reduce development efforts. It enables researchers to examine various spatial indexes in a short period while ensuring effective performance. After the internal logic of spatial indexes is implemented, the conversion to distributed versions only requires developers to write a few callback functions and wrap spatial query functions that call for consistent accesses with transactional primitives. This simplicity results from the Mempool runtime, which transparently synchronizes local states of data structures with DTranx. Mempool handles two critical tasks: address translation and transparent server synchronization. Mempool assigns addresses to data bricks such that the remote servers know where to store or fetch them. Transparent server synchronization includes automatic transaction construction and data synchronization. Particularly, Mempool is capable of capturing data brick reads, creations, and updates automatically and constructing transactions

transparently before committing to DTranx. It speeds up the conversion process significantly since developers do not have to keep track of the accessed data items in transactions. Furthermore, Mempool maintains a local cache of deserialized objects to lower the serialization overhead.

Dynamic Partitioning Strategy We propose a dynamic partitioning strategy into the transactional KV store to reduce inter-server communication and balance resource usage. Traditional key value stores do not yield good performance when workloads display non-uniform access patterns. For example, the server storing the root of a binary tree is visited by every request. We adopt a hybrid strategy that combines hash and adaptable partitioning. Transactions are represented as a graph structure and a partitioning algorithm called METIS [65] is applied to minimize edge cuts. It brings two benefits. First, data that are jointly accessed in transactions are stored together such that no 2PC is necessary. Second, data loads are effectively replicated and balanced across the cluster. In addition, we introduce a greedy algorithm to minimize data migrations and adjust 2PC to enable online database migration while still serving strongly consistent transactions.

BTree & RTree We successfully convert two single-threaded data structures to distributed ones within ten days: BTree and RTree. For pure search workloads in a 25-server cluster, our BTree and RTree achieves 253.07 kops/sec and 77.83 kops/sec respectively. STI-BT, a highly optimized distributed BTree shows ≈22 kops/sec in a 60-server cluster. For workloads with 70% search, 15% insert, and 15% remove operations in a 25-server cluster, our BTree and RTree support 100.13 kops/sec and 35.36 kops/sec throughput respectively. Tango achieves 20k ops/sec in a 18-server cluster for cross-partition transactions on a Zookeeper tree.

1.4 Thesis Organization

The remaining chapters are organized as follows. Chapter 2 introduces the background of distributed spatial access methods and the common architecture. Chapter 3 describes a scalable and transactional key value store, called DTranx, which adopts persistent memory for log designs. Chapter 4 then presents a paradigm for building and testing various scalable and transactional spatial structures based on DTranx. Chapter 5 reviews the past research on distributed spatial access

methods in detail and discusses techniques applied. Chapter 6 discusses the strength and weakness of DDSBrick in greater depth, comparing it with peer systems and analyzing the limitations. Finally, Chapter 7 summarizes the thesis.

Chapter 2

Background

2.1 History

Figure 2.1 shows the history of distributed spatial access methosds(DSAM) research. DSAM is mainly based on Multidimensional Access Methods(MAMs) and Scalable Distributed Data Structures(SDDSs). The first DSAM papers are MR-Tree [90] and k-RP*s [102] where MR-Tree proposes to store a global RTree in the master server and k-RP*s decentralizes a global RTree by gradually splitting.

2.1.1 MAMs: Multidimensional Access Methods

Gaede et al. [61] and Samet et al. [143] give a comprehensive review on MAMs. Based on the object types, there are two classes of methods: Point Access Methods(PAMs) and Spatial Access

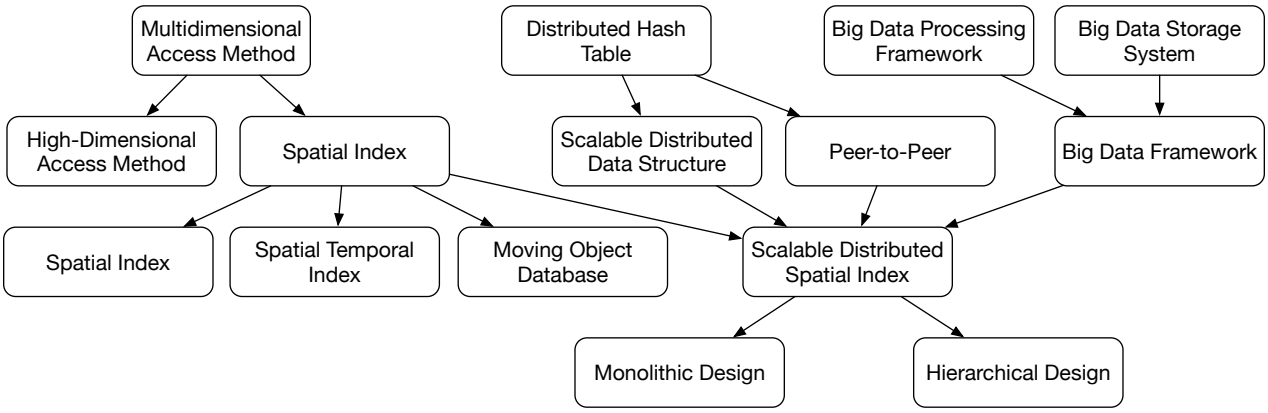


Figure 2.1: Distributed spatial access methods history.

Methods(SAMs) where the former searches for points and the latter handles extended objects, such as rectangles. Another categorization depends on the decomposition method. MAMs with data-dependent decomposition, such as cell tree [70], choose the partitioning hyperplane based on the current objects while MAMs with data-independent decomposition apply regular decomposition[1] , such as QuadTree [142] and PMR-quadtree [125]. On the other hand, decomposition methods can be either disjoint or overlapping based on whether the regions of child nodes are disjoint or overlapping. Overall, there are many criteria to define optimality for the MAMs and the choice of MAMs mainly depend on the application requirements.

Within MAMs, researchers that focus on spatial temporal access methods add temporal attributes in the access methods to support temporal queries. Based on the reviews from [115], [112], [127], and [141], the majority of spatial temporal access methods are R-Tree variations, such as MV3R-Tree [154] and TPR-Tree [160]. And, spatial temporal access methods usually do not handle deletions since old data are queried in temporal databases. Bohm et al. [27] conducts an overview on high-dimensional index structures where the high number of dimensions renders normal MAMs useless. Illarri et al. [82] introduces location dependent query processing in moving object databases(MODs), such as LUR-Tree [92]. The challenge is to process frequent updates efficiently. However, the majority of the proposed work depend on either of the two assumptions: trajectory being a mathematical function, queries being continuously monitoring a static space while objects are moving. This paper aims at literature work on distributed spatial indexes where the observers are not known as *a priori*.

2.1.2 SDDS: Scalable Distributed Data Structures

As distributed hash tables(DHTs) became heavily explored in 1990s, scalable distributed data structures attracted researchers' attention. The main focus was to extend DHTs to efficiently support complex query processing, such as range queries. There are generally two ap-

[1] Regular decomposition partitions the space by recursively halving it across the various dimensions instead of permitting the partitioning lines to vary

proaches: over-DHT indexing and overlay-dependent indexing. Over-DHT indexing rely only on the "put/get/lookup" interfaces and can utilize any DHTs. Overlay-dependent indexing entails development of locality-preserving overlays, which will be covered in 2.1.3. Examples of SDDSs include LH* [103], RP* [104], and LIGHT [153].

2.1.3 P2P: Peer-To-Peer Systems

P2P systems implemented indexes to query for object locations and consistent hashing [86] has been widely used. When P2P systems support key value searching, the indexes could be DHTs, such as Pastry [140], Chord [151], and P-Grid[7]. When P2P systems support range queries on keys, the indexes could be a tree structure over DHTs, such as LIGHT [153] or BATON [83]. One example of P2P systems supporting multi-dimensional queries is CAN [137].

2.1.4 NoSQL Databases & Big Data Computing Framework

Eldawy et al. [54] and Nam et al. [121] surveyed multidimensional indexes based on distributed storage systems. Eldawy et al. covers spatial databases that rely on big data frameworks, such as HBase [2] and MapReduce [43]. It summarizes three different approaches for spatial databases: on-top, from-scratch, and built-in. The on-top approach implements algorithms of query logic or MAMs construction in the big data processing system. For example, Zhang et al. [179] implements MapReduce programs to scan the input records for range queries and k-nearest neighbor queries. From-scratch approach designs the spatial databases from scratch and do not construct distributed MAMs. Examples are Paradise [44] and SciDB [152]. Built-in approach either applies space filling curves to reduce multidimensional data to one-dimensional keys or constructs central MAMs before storing to the NoSQL databases. MD-HBase [128] applies Z-ordering to order the multidimensional objects before a Quad-Tree or KD-Tree index is created where the index structures are stored in the underlying key value store. However, we do not follow its categorization since it focuses on spatial databases, which includes query engines, languages etc.

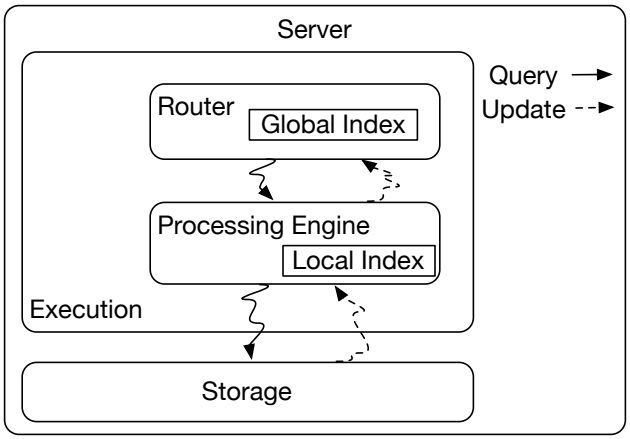


Figure 2.2: Architecture

2.1.5 Summary

From 1970s to 2000s, researchers have been explored multidimensional access methods, SDDSs, P2P systems, and distributed storage systems, which lays the foundation for building scalable distributed spatial indexes. Multidimensional access methods provide various data structure designs to support different data types and query types. SDDSs and P2P systems offer partitioning and load balance techniques to achieve linear scalability. Lastly, distributed storage systems lend persistence support with possibly transactional interfaces.

2.2 Architecture

We conduct this survey on distributed spatial index research from the last two decades. Specifically, our focus is spatial access methods as opposed to high dimensional access methods, distributed data structures as opposed to central single server structures and index design as opposed to GIS system or database designs. A high level design is illustrated in Figure 2.2.The execution module handles updates and queries by utilizing the spatial index structures. The spatial index structures can be either monolithic or hierarchical where monolithic indexes assembles single server indexes and hierarchical indexes divide the whole indexes into two layers: global index and local index. When servers receive a request, the router searches through the global index and

decides whether to forward the request or process it locally. The storage module persist the indexes and it could be either local storage systems, such as LevelDB, or distributed storage systems, such as HBase.

2.2.1 Execution Module

There are three major design challenges in the execution module: processing engine, router, and load balance. The processing engine handles three subproblems: query type support, filtering, and parallelism. Query support means what kind of queries the distributed spatial index supports, such as point queries, range queries, and KNN queries, etc. Gaede et al. [61] summarized all the spatial query types. Query support usually depends on the spatial AMs adopted. For example, RTree supports range queries and Voronoi diagram supports nearest neighbor searching. And, the spatial AMs also affect what spatial objects are supported. For example, KD-Tree only stores point data while RTree stores rectangular data. Filtering means how the processing engine filters out regions which expected objects are not in. For example, RTree with bloom filters, such as BR-Tree [80], reduces false positive rates. Parallelism aims to lower processing latencies. Some researchers utilizes the distributed computing engines like Mapreduce and Spark to parallelize query processing. Examples are Traverse [174] and [12].

The router maintains the whole or partial global index as the routing table. The critical problem is how to use less routing information to route within fewer steps. Less routing information means less cost for database updates and fewer steps means more efficient query processing. The most popular approach is to adopt the P2P overlay network that commonly needs O(logN) links to achieve average routing step bound as O(logN) where N is the number of the servers.

Load balance guarantees high scalability. If the workloads display static hot spots, the execution module can achieve load balance by offloading the highly loaded servers to lightly loaded ones. However, The problem becomes complicated when the hot spots frequently change, in which case index distribution should be adapted to dynamically adjust the loads.

2.2.2 Storage Module

Storage module is responsible for index construction and maintenance. Index construction can be either insertion gradually like SD-RTree [49] or batch construction like EDMI [186]. Index maintenance keeps the indexes consistent, which consists of consistency between global indexes and local indexes and consistency among replications. For hierarchical indexes, lazy updates are adopted in BR-Tree [80] and ATree [134] such that local index changes are lazily propagated to global indexes. The laziness could lead to false positives and false negatives. When local indexes shrink, lazy updates results in false positives where global indexes direct queries to servers without expected data. When local indexes expand, lazy updates causes incorrectness. For example, a search query might return a partial result set since it skips servers that contain the expected data. For consistency among replications, researchers choose different replication factor. Generally, indexes are stored in master-slave mode, partial-replication mode, and full-replication mode. In master-slave design, indexes are stored in a designated master server and requests are forwarded to slave servers for further processing. Partial-replication design stores part of the indexes in each server and proposes a routing algorithm to forward requests to corresponding servers. For example, P2P technologies overlay a structured network, mapping points, rectangles, and cubes to servers. Full-replication design simply replicates the global index in all servers such that each server is able to process requests locally.

2.3 Summary

This chapter reviews multidimensional access methods, scalable distributed data structures, peer-to-peer systems, and big data systems from which distributed spatial indexes inherit various techniques. Then, a high-level general architecture that all the distributed spatial indexes have followed is presented. The main challenges are summarized as four aspects: query support, routing, load balance, and distributed storage.

Chapter 3

DTranx

3.1 Introduction

There are numerous storage management systems, such as distributed RDBMS, NoSQL database, distributed file systems, and transactional key value stores. These systems offer different levels of transactional and data schema support. Distributed RDBMS provides strict data schema and full ACID properties with the price of low availability and efficiency. NoSQL databases and distributed file systems, such as Cassandra [93] and GFS [66], are scalable and highly available, but they often lack consistency support. Transactional key value stores, such as BigTable [32], sacrifice data schema flexibility, but they offer higher availability, superior performance, and better scalability.

In this paper we present DTranx, a SEDA-based distributed transactional key value store with persistent memory log. DTranx follows the SEDA[167] architecture to exploit the high concurrency in multi-core environments. SEDA organizes the software in a network of stages where stages contain both the application logic and communication channels. DTranx adopts lock free queues as the communication channels to reduce contention among threads. In addition, DTranx binds threads to physical cores to minimize the context switch overhead.

Unlike most existing key value stores, DTranx is fully ACID compliant supporting serializability. To serialize concurrent transactions, we adopt a hybrid of Optimistic Concurrency Control(OCC) and Two-Phase Commit(2PC) to narrow down the critical section of distribute locking to the commit time and enables parallel validation for high scalability. Furthermore, we avoid dead-

locks and livelocks with a customized locking mechanism where transactions are aborted if shared lock requests are rejected and the exclusive lock requests are blocked for a timeout if not granted immediately. However, if the data is exclusively locked when the new exclusive lock requests come, the new requests are rejected immediately.

Moreover, DTranx integrates a modular Write-Ahead Log (WAL) which can be configured to use conventional SATA SSDs or Non-Volatile Memory(NVM) [163] technologies. Applying NVM in the WAL considerably cuts down the durability cost that most ACID-compliant systems suffer. A state transition mechanism to garbage collect(GC) WALs is also developed to reclaim the logs of the completed transactions. The garbage collection process does not affect normal transactions since old logs and the current appending log are not in the same file.

In summary, the contributions of DTranx are as follows:

- Adopting SEDA concurrent architecture and employing the optimal core binding strategy;
- Customizing a hybrid commit protocol combining optimistic concurrency control and two-phase commit and introducing a locking mechanism to avoid deadlocks and livelocks;
- Adopting NVM using the Linux `pmem` library in the WAL of the distributed transactional system to reduce the persistence overhead and to offer durability;
- And, designing a state transition based garbage collection mechanism to efficiently reclaim increasing log space without affecting normal transactions.

3.2 Background

Staged Event-Driven Architecture SEDA is a highly concurrent architecture, consisting of a network of event-driven stages connected by queues. A stage is an independent software module that manages a shared resource. For example, the lock service in transactional systems is a stage that maintains the locking information and handles lock requests. As shown in Figure3.1, a stage is composed of an incoming event queue, a handler, and a thread pool. Besides the three core

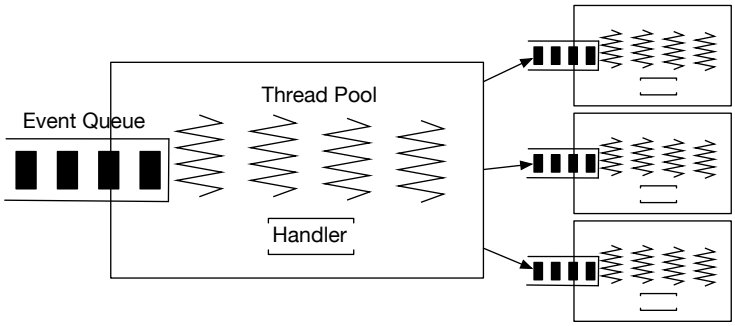


Figure 3.1: SEDA architecture

elements, SEDA adds a controller to adjust the thread pool size dynamically. The event handler sends events to another stage by invoking the enqueue operation on the incoming event queue of that stage. SEDA brings four benefits. First, it offers modularity and independent load management. Second, it facilitates debugging and performance analysis, which has always been a tough task for multi-threaded programs. Third, it optimizes the overall system performance by dynamically adjusting resource allocations, such as thread numbers among stages. Fourth, it enables batch request processing. For example, a database stage could write multiple keys at a time. Note that SEDA requires nonblocking design of the event handler.

Optimistic Concurrency Control Concurrent control is the coordination of multiple concurrent accesses to the database and Philip et. al [24] decomposed it into two major subproblems: read-write synchronization and write-write synchronization. There are pessimistic and optimistic approaches towards both subproblems. The pessimistic approach assumes the probability of access conflicts to be high and decides whether to restart at the start of transactions, such as two-phase locking. The optimistic approach assumes the probability of access conflicts to be low and decides whether to restart at the end of transactions. Specifically, Optimistic Concurrency Control(OCC) [74] consists of three phases: read, validation, and write. During the read phase, transactions read databases and store updated data in the buffer. Then, databases check whether the current transaction is in conflict with any concurrent operations. Finally, if it passes the validation phase, the current transaction proceeds to update the database states.

Two-Phase Commit Two-Phase Commit(2PC) is a classic commit protocol in the distributed environment that guarantees agreement among servers on the commit results. Moreover, once the agreement is reached, the commit results are persisted even if the servers fail. There are two roles for the servers: coordinator and participant. During the first phase, coordinators initiate 2PC by sending ***prepare*** messages to participants and participants either accept or reject the ***prepare*** messages. In the second phase, coordinators send out ***commit*** messages if all participants accept the ***prepare*** messages and ***abort*** messages, otherwise. Both coordinators and participants write Write-Ahead Log(WAL) to persist volatile states, such that the commit decisions for recovery purposes.

3.3 Design

DTranx adopts the SEDA architecture to reach the optimal performance in each server and achieves serializability by combining OCC and 2PC protocols. Furthermore, it introduces a NVM-based WAL design with a garbage collection mechanism to effectively and efficiently reclaim logs.

3.3.1 Architecture Overview

As shown in Figure 3.2, DTranx follows the SEDA [167] design and invents three categories of stages: ***Service***, ***Internal***,and ***Daemon***. ***Service*** stages handle Remote Procedure Calls(RPCs). For example, *ClientService* accepts transaction commit requests from clients and *TranxService* processes 2PC requests from peer servers. ***Internal*** stages manage local shared resources. For example, *LockService* maintains locking states and *WAL* writes logs to persistent storage. ***Daemon*** stages run background tasks. For example, *GC* periodically reclaims logs and *TranxAck* sends commit results from coordinators to participants.

To further exploit concurrency in SEDA, DTranx adjusts each of the stage components. First, DTranx removes the dynamic control of the thread pools but statically assigns thread numbers for each stage, after which threads are bound to physical CPU cores. We found out that one thread for

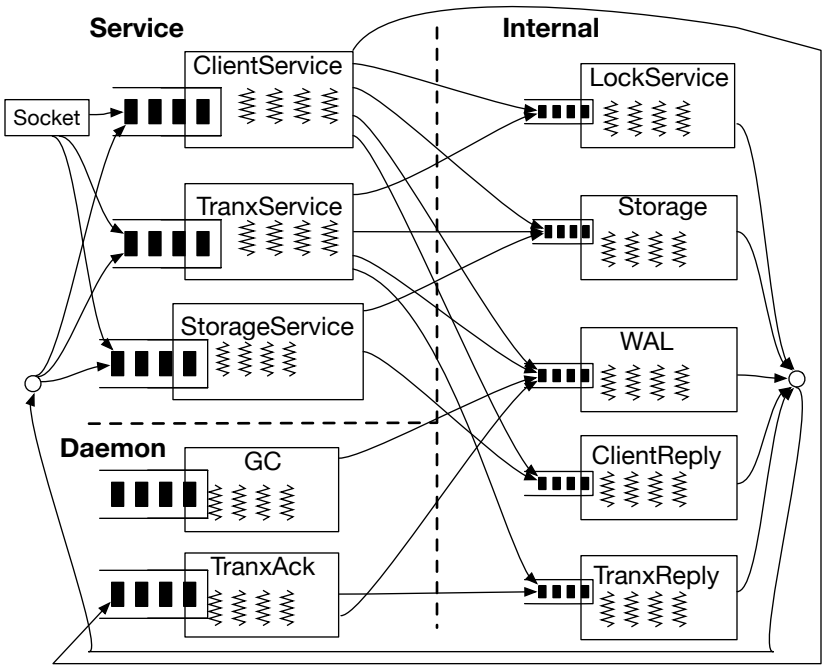


Figure 3.2: DTranx architecture.

each stage yields better performance when context switching is rare than that of multiple threads for each stage when context switching happens frequently. However, dynamic control of the thread pools is enabled in certain stages where handlers might be blocking. For example, *Storage* launches multiple threads to handle I/O requests which involve blocking system calls. Besides the core bindings for DTranx threads, kernel Interrupt Request(IRQ) threads are bound to CPU cores as well since I/O throughput is severely affected otherwise.

Second, DTranx adopts lock free queues as the incoming event queues such that the enqueue/dequeue operations on the queues are nonblocking and it achieves high throughput without compromising consistency. The lock free queues utilize atomic primitives to reserve a spot and then proceed to read/write in non-critical sections. In addition, multiple queues are created in each lock free queue to spread loads.

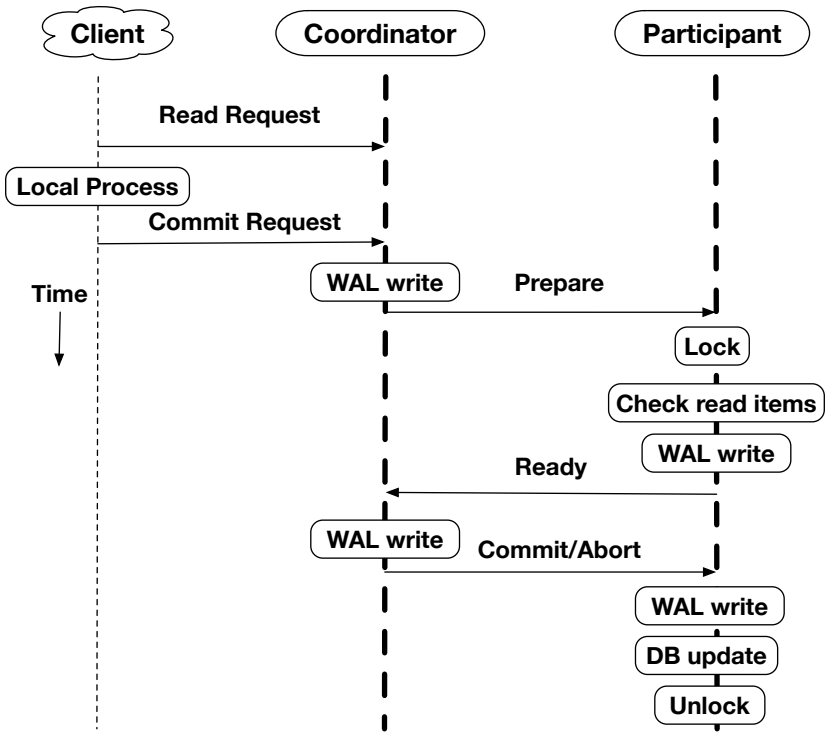


Figure 3.3: Commit Protocol.

Third, DTranx reduces queue element construction and destruction costs by pushing element pointers , instead of the element itself into the lock free queues and allocating an element pool to store destructed elements. For example, ***Service*** stages get elements from the pool when new requests come and ***Internal*** stages put elements to the pool when requests are completed.

3.3.2 Serializability

DTranx combines OCC and 2PC protocols to guarantee serializability, following Alexander's [156] hybrid OCC scheme that embedded lock acquisition and validation in the 2PC. The main benefit compared to distributed Two-Phase Locking(2PL) is that locks are only held during the commit time and DTranx employs parallel validation for better scalability. The detailed protocol flows are shown in Figure 3.3. Additionally, if all data items in a transaction are stored in the same server, 2PC is converted to One-Phase Commit(1PC) to reduce latencies.

Initially (Stage 1), transactions read data without locks and clients keep track of the read

items, read item versions, and write items in the local buffer. At commit time (Stage 2), clients choose a server as the coordinator and send it the transactions. During the first phase, coordinators initiate 2PC by first sending prepare messages to participants. Then, participants lock both the read and write items, check the read item versions, write WAL logs and reply to the coordinator. During the second phase, coordinators wait for responses from all participants, then decides whether to commit or abort, and notifies all participants of the agreed result. However, if any participant aborts in the first phase, the coordinator immediately sends out abort messages without waiting for all replies. Finally, participants write WAL logs, update database states, and unlock all relevant data.

Proof of Serializability

Assumption: Two phase locking(2PL) ensures serializability, see proof in [64].

Method: We reduce the hybrid OCC to 2PL. Using action abbreviations L (Locking), C (Checking), U (Unlock), R (Read), W (Write) and object abbreviations r (read items), w (write items). Concatenated action and object symbols represent tasks, e.g., Lr means "lock read items". The sequencing abbreviation "-" binds two actions and enforces an "execute before" local order and $\rightarrow$ binds two tasks and enforces an "execute before" distributed order. Our transactions can thus be represented as R $\rightarrow$ Lrw-Cr $\rightarrow$ W-Urw. The Cr action validates the read items. If any read item has been changed after it was read, the transaction aborts, releasing all locks. If not, our successful transaction is equivalent to Lr-R $\rightarrow$ Lw-Cr $\rightarrow$ W-Urw, thus Lr-R $\rightarrow$ Lw $\rightarrow$ W-Urw. In this way, all locking actions precede all unlocking actions, which is 2PL. Unlocking after committing to the database avoids cascading rollbacks. For successful transactions, the serialization point is the moment when all the write locks are granted.

Deadlock Common deadlock avoidance methods are timeout, wait-for graph, ordered locking and timestamps with wait-die or wait-wound mechanisms. SiloR [183] avoids deadlocks by enforcing a global order on the locking sequences, necessitating multiple round trips in distributed environments. The wait-for graph introduces too much network traffic and timestamps method requires a global synchronized clock, which will become the bottleneck or single point of failure.

Our deadlock avoidance method aborts transactions immediately if read locks are not granted and waits for a configurable time period(e.g. 50ms) before aborting write lock requests. However, if the data is currently exclusively locked, the write lock request is aborted immediately. If a transaction is aborted since write locks are not granted, DTranx retries committing it after an exponential timeout. If a transaction is aborted since read locks are not granted, DTranx restarts it immediately. This is because read lock request denial indicates there are concurrent transactions updating the same item and retrying committing will fail again.

Livelock Write starvation rarely happens since write lock requests are blocked for a short fixed period and exponential backoff technique is adopted to reduce the probability of lock conflicts when the transactions are retried. The same goes for read starvation since the number of read items are usually much larger than write items.

3.3.3 Persistent Memory Based Log

In distributed systems, the logging module plays a critical role in failure recovery. WAL is the persistent copy of the volatile states that are subject to failures due to power outage and kernel hanging. However, persisting WAL to the durable storage results in long latencies. With the advances of the NVM technologies, the performance gap between in-memory and persistent storage accesses is narrowing. Thus, we propose a WAL design based on NVM and introduce a garbage collection mechanism to effectively and efficiently reclaim the limited NVM space.

3.3.3.1 Log Design

The logging module is designed in three vertical layers: NVM library, LogManager and TranxLog. The NVM library provides the basic interface to persistent memory to create files, read and write data. LogManager structures the log into a list of log files, calculates checksums, and supports block read/write operations. Lastly, TranxLog offers high level abstractions for distributed transaction logs and presents a continuous and append-only log.

We use Intel's NVM [3] library to manipulate memory mapping for log files in persistent

memory. After log files are mapped to the memory space, writes are immediately durable after being flushed from cache to memory(e.g. using clflush in the x86 instruction set). Two adjustments of the NVM library are made. First, a read pointer is added to the original NVM library to provide an on-demand read interface. Second, internal write locks are disabled since only one thread is launched in the *WAL* stage, thus no race conditions.

On top of NVM library, LogManager organizes the logs in a list structure such that logs of variable sizes are supported. To reduce I/O system calls and reach higher throughput, reads and writes are block based and logs in the same block are buffered in memory. In addition, checksums are calculated and written for each block to detect data corruption.

TranxLog serializes transaction logs such as *CommitLog* that records commit states for coordinators and *ReadyLog* that records ready states for participants. Then, TranxLog separates WALs into files whose names are set to their creation timestamps. Thus, the file with the smallest timestamp is the oldest one, with which the garbage collector starts. On the other hand, reclaiming old log files does not interfere with current transactions since current transactions are appending logs to new log files.

3.3.3.2 Garbage Collection

Since the WAL is written as transactions are committed, its size would increase indefinitely if DTranx does not reclaim the WAL of complete transactions. The WAL for transactions that reach consensus are not required during recovery. Therefore, we introduce a state transition based garbage collection mechanism to identify unnecessary logs without performance hiccups. In particular, each transaction is assigned with a unique TranxID that combines the ServerID and a local monotonically increasing 64-bit integer. Since ServerIDs are assigned as the server indexes in the group membership stored in the replicated state machine Raft [130], TranxIDs are guaranteed to be distinct among servers. Moreover, each server keeps updating the largest committed TranxID, LC_TranxID, where transactions with TranxIDs less than LC_TranxID have reached consensus. Then, each server broadcasts its LC_TranxID and stores the LC_TranxIDs from other servers in a

fixed-size GC log. The benefits of the GC log are twofold: it is fixed size space usage and it enables WAL reclamation.

The state transition flow is illustrated in Figure 3.4. On the one hand, each transaction has a state to represent the current stages in the 2PC and each server has a *GC* state to record completed transactions where LC_TranxIDs are calculated. On the other hand, there are volatile and nonvolatile states where the nonvolatile states are durable copies of the volatile ones. For example, WALs are the nonvolatile copies of 2PC states including *Start*, *Prepare*, *Ready*, *Commit*, and *Abort*. GCLog is the nonvolatile state persisting the *GC* state. Although both WALs and GCLog are persistent copies of the volatile states, their orders of updating volatile and nonvolatile states differ. For WALs, nonvolatile 2PC states are updated after WALs are written in order for the transactions to be recoverable. For the GCLog, *GC* state is updated before GCLog for two reasons. First, the history can be replayed as long as WALs are not reclaimed yet. Second, accumulating in-memory *GC* states and writing to the GCLog in batch is more I/O efficient. For coordinators, GCThread periodically collects the volatile *GC* states and updates the GCLog, after which WALs containing only completed transactions are reclaimed and the LC_TranxIDs are broadcasted to all the other servers. For the participants, GCBroadcast thread passively receives the broadcasted LC_TranxID, updates the local *GC* state and GCLog, and then reclaims WALs.

Not only does the state transition help to reclaim WALs, it is also utilized to clean the aborted transaction IDs in the lock service, which are referenced to avoid faulty re-lock situations. For example, after a participant receives the prepare request of transactionA and its volatile state is checked to be *Start*, an abort request of transactionA arrives, changing the volatile state to *Abort*. It is possible that abort requests come before prepare requests are done since the coordinator immediately sends out abort requests if any participants aborts. Note that these two requests are processed concurrently. Then, the prepare requests lock the data items and these locks will never be unlocked. Nonetheless, the committed transaction IDs are not stored in the lock service since coordinators only send out commit requests after all participants agree to commit, in which case it is impossible that prepare and commit requests are processed concurrently.

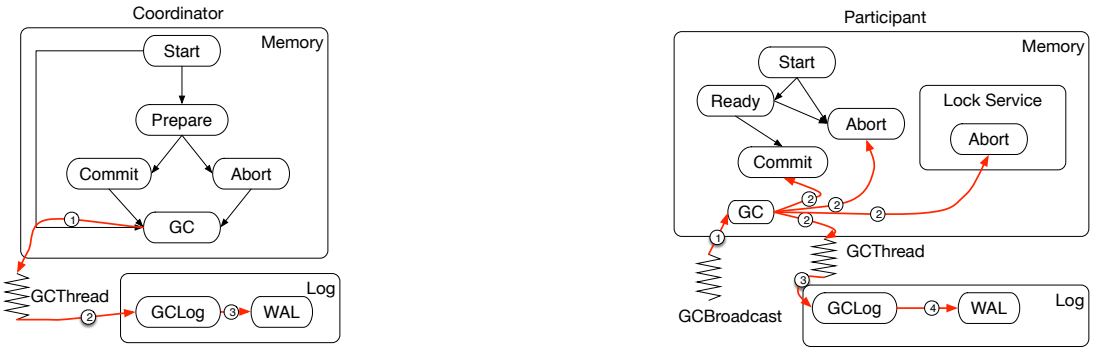


Figure 3.4: Coordinator and Participant state transition. Black arrows show the state transitions and red arrows show the ordered steps of garbage collection.

3.4 Implementation

In this section, we explain the implementation details of DTranx and also the major optimization techniques used to improve DTranx performance.

3.4.1 Cache

DTranx enables client side cache to avoid excessive network traffic. The caching policy works as follows: (1) The data cache is updated if read or commit requests succeed. (2) The data cache is invalidated if commit requests fail. In addition, DTranx servers piggyback the updated data in the response to failed commit requests such that the clients can update the local cache and read requests in the retrying transaction can read from the local cache.

On the other hand, DTranx enables the server side database cache to serve read requests in lower latency and it adopts the write-through strategy for durability.

3.4.2 Exactly-Once RPC

There are three different RPC calls corresponding to the three stages in Figure 3.2: Read requests from clients to servers; Commit requests from clients to servers; and, Transaction requests

from servers to servers. Duplicate processing of RPCs would lead to system failures in certain cases. For example, if DTranx servers process a duplicate prepare request after the corresponding commit request is done, the locking service would lock the data items and future transactions would not be able to update these data. Therefore, DTranx should guarantee to process each RPC exactly once.

First, we guarantee at least once delivery by resending messages on the sender side if no responses are received within a timeout. We build the RPC protocol based on the ZeroMQ library, which automatically resends messages if they are lost. In addition, DTranx implements the retrying mechanism itself when no responses are received since at least once delivery in ZeroMQ does not indicate at least once delivery in DTranx. For example, if servers are restarted after the ZeroMQ library receives a message but before the DTranx system detects the message, ZeroMQ does not retry the message and the message is lost.

Second, we guarantee at most once processing by blocking duplicate messages. we assign distinct IDs for each RPC message and receivers record the IDs of completed messages. Read requests are never blocked since they are idempotent. For Commit requests, each message has a clientID and messageID where the clientID is distinct for each TCP connection and the messageID is monotonically increasing for each client. ClientIDs are assigned by the ZeroMQ library when the connection is established. For Transaction requests, each message has a unique transaction ID(TranxID) and a message type. TranxID is the concatenation of the distinct coordinator server ID and a monotonically increasing integer. And, there are four message types corresponding to the four Transaction requests in Figure 4.5: *Prepare*, *Ready*, *Commit*, and *Abort*. With at least once delivery and at most once processing, each message is processed exactly once.

3.4.3 LevelDB

We choose levelDB[4] as the local database implementation since it is lightweight and efficient compared to multi-version KV stores. To validate the read items during the OCC commit, DTranx keeps a version number for each key value pair by storing the combination of the real value and a version number as the value in levelDB. The real value and the version number are separated

by a special delimiter, such as #. When clients send read requests, servers interpret the values retrieved from levelDB and returns the value and the version number to the clients. When clients send Commit requests, servers increment the corresponding version numbers by 1 if transactions commit.

3.4.4 Fault Recovery

As the cluster size increases, the probability of server failures will increase considerably. For example, if the aggregated MTBF (Mean Time Between Failure) of a server is 1 year including disk failures, network failures etc., then in a cluster of 100 servers, there is a server failure every 3 to 4 days on average. DTranx triggers the recovery process in two stages: local recovery and global recovery. Local recovery reapplies local logs by updating databases if transactions commit and lock data items if no agreement has been reached. In addition, DTranx fills the TranxID space with aborted transactions. Missing transactions are possible when servers crash immediately after read item checking fails in coordinators. Global recovery repairs transactions of which commit results can not be decided unilaterally. It is initiated after local recovery to inquire transaction states from other involved servers. Specially, if the coordinator is in *Prepare* state and all participants are in *Ready* states, neither committing nor aborting violates distributed consensus. DTranx chooses to abort them such that the clients can assume the transaction failure if no responses are received.

On the other hand, DTranx starts service stages in Figure 3.2 after local recovery such that the changes from completed transactions are applied and in-memory states of ongoing transactions are stored. However, the order between service stage startup and global recovery does not matter and DTranx chooses to starts service stages before global recovery to reduce the service downtime.

3.4.5 Optimizations

In order to achieve better performance, multiple optimization techniques are applied. The most significant techniques are listed below.

- **Delayed In-Memory Reclamation** DTranx reclaims the volatile *Commit*/*Abort* states

in participants when the servers are under light loads to avoid performance hiccups.

- **Batch Ack Phase** DTranx delays the second phase(Ack phase) of 2PC when coordinators send transaction commit results. We delegate the Ack phase to a separate stage, TranxAck in Figure 3.2, to reduce the transaction latency and offload the high processing demand of the ClientService stage.

- **Core Bindings** We manually analyze the queue size for each stage and bind the threads to physical cores in an optimal way. The best core binding strategy yields almost 6 times higher throughput than the worst. In the future, we plan to explore how to automate the core bindings to attain the best performance based on the number of CPU cores available.

3.5 Evaluation

Our tests are run on a Cloudlab [138] cluster with 36 machines. Each machine has dual Intel E5-2660 v3 with 20 2.6GHz cores, 130GB RAM, 480GB solid state disk at 6GB/sec, and 10Gbps Ethernet card. We emulate the NVM by enabling DAX support in Linux to create a PM-aware environment. This DRAM based emulation is adopted since current persistent memory latency is comparable to DRAM and NVM was not available on Cloudlab yet. For example, STT-RAM [17] achieves ∼10ns write latency compared to 50ns DRAM latency. NVM throughput is also far beyond the current usage as shown in Figure 3.7b. To generate workloads, we use Yahoo Cloud Serving Benchmark(YCSB) [35] C++ version and add DTranx and HyperDex support. YCSB clients are running in separate servers from the cluster which accommodates the DTranx system. Each experiment is repeated three times, and the average values are used.

3.5.1 Experiment Details

We evaluate DTranx with a database of 120 million key value pairs in the 36-node cluster. Test data keys are generated as integers from 1 to 120 million and values are 100 bytes of random characters. Transactions are categorized into read and update transactions. Read transactions only

contain read items and update transactions contain one write item. The total number of read/write items in one single transaction is uniformly distributed between 1 and 3.

3.5.2 Transaction

In Figure 3.5, DTranx is compared with Hyperdex Warp [55] that supports distributed transactions. Only successful transactions are counted in the throughput metric. DTranx shows approximately 30% higher throughput than Hyperdex and DTranx degrades slowly as the percentage of update transactions increases. Moreover, DTranx maintains high commit success rates. For example, DTranx reaches 99.65% success rate for 50% read workloads. On the other hand, Hyperdex shows high throughput but the software is unstable and periodically fails due to internal assertion errors, leading to low success rates. For example, several servers crashed during the 95% read workload, causing 58.96% success rates and 275.72k ops/sec throughput. To remedy the crash effect, we restarted the servers manually after each run. There are three reasons why DTranx outperforms Hyperdex. First, DTranx follows the highly concurrent SEDA architecture with lock free queues and stages are bound to physical cores, utilizing all CPU power and avoiding context switching overhead. Second, DTranx integrates the NVM based log that bypasses system calls like sync/fsync, reducing log persistence latencies. Third, DTranx applies various optimization techniques, such as an allocated element pool, batch ack phase, and optimal core binding strategy. Furthermore, strace [5] reveals that Hyperdex does not synchronize data to physical storage devices immediately after write log calls. While the Hyperdex paper supports fault recovery by replication, that version of the software is not publicly available. Lastly, the average latency for DTranx is below 2ms when the throughput is 50% of the maximum and it increases to 10ms when the throughput reaches the maximum.

3.5.3 Scalability

In this experiment, scalability tests are run against cluster of 3, 9, 18 and 36 servers. Corresponding to the cluster size, 10, 30, 60, 120 million keys are inserted into DTranx. As shown

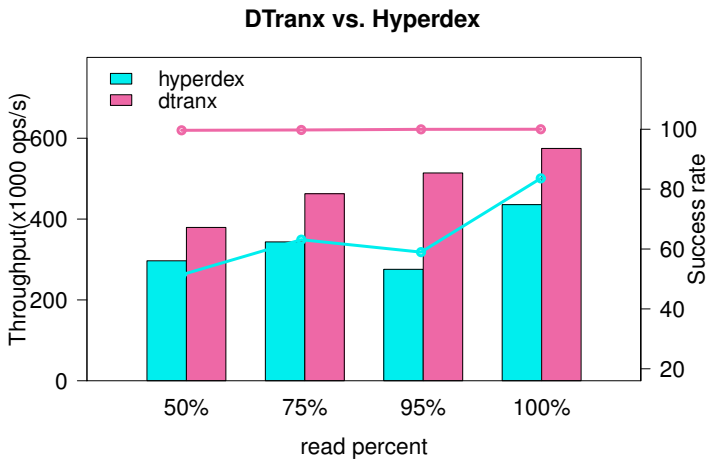


Figure 3.5: Distributed transactions. The horizontal axis represents the percentage of read transactions while the vertical axis shows the throughput on the left and commit success rates on the right.

in Figure 3.6, the throughput shows linear increases as more nodes are involved. For example, with pure read workloads, throughput reaches 574.76k reqs/sec with 36 nodes. In addition, workloads with various mixture of read and update transactions are benchmarked. Even with 50% read workloads and 50% update workloads, the throughput is 60% to 85% of that with pure read workloads. The high scalability of DTranx results from our efficient hybrid commit protocol design that minimizes the critical section of distributed locking and reduces the 2PC to 1PC whenever possible.

3.5.4 Persistent Memory Based Log

Two experiments are conducted to validate the effectiveness and efficiency of the NVM based log. Both experiments are run with 36 servers and 95% read transactions. In Figure 3.7a, the instant throughput is plotted with and without GC. The GC process doesn't affect normal transactions when WALs are GC'ed every 10 seconds since the reclamation of volatile states that affects normal transactions is delayed until servers are in light loads. The system with SSD log shows 19k ops/sec on average, which is 30 times slower than that with NVM log. In Figure 3.7b, we measure the

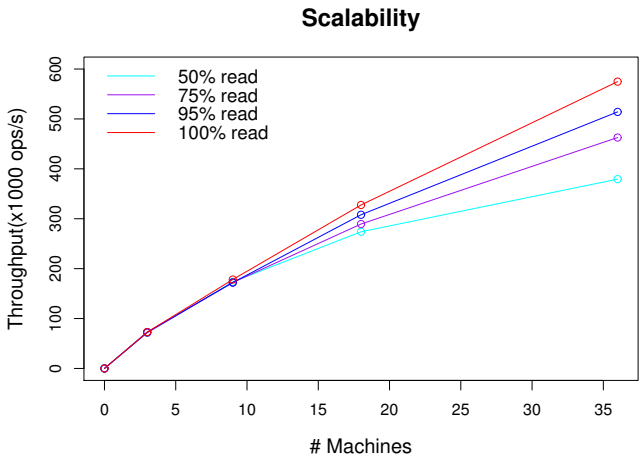
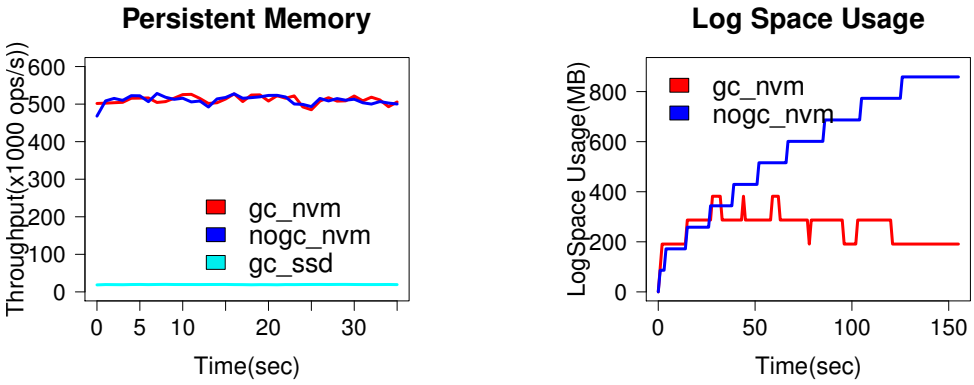


Figure 3.6: DTranx Scalability. Four workloads are run, namely 50%, 75%, 95%, and 100% read workloads.



(a) throughput

(b) log space usage

Figure 3.7: The left plot shows the instant throughput with and without GC for both NVM and SSD. The right plot shows the space usage with and without GC for NVM. Specifically, gc_nvm means the system with GC enabled and NVM based log; nogc_nvm is the system with NVM based log but GC disabled; gc_ssd is the system with GC enabled and SSD based log.

space over time with and without GC to show the GC efficiency. The logs are NVM files of 100MB size so that the space usage changes in units of 100MB. The GC mechanism successfully keeps log space usage low since DTranx reclaims the transactions from WALs much faster than it writes them. After 120 seconds, tests are complete and the log usage with GC converges to 200 MB(one for GCLog and one for the current WAL).

3.6 Related Work

DTranx is a highly concurrent and transactional KV store that integrates various techniques from concurrent programming, database, and NVM fields.

Distributed Transaction. Transaction research are heavily explored in the database field and we investigated both classic and state-of-the-art methods to guide the DTranx design. Spanner [36] was a globally consistent and efficient key value storage system, which required atomic clocks to be installed on each server and its two-phase locking approach limited concurrency. Yang [182] introduced transaction chains to obtain both serializable transactions and low latency but required that read and write items to be known as a priori, similar to Granola [37]. Calvin [157] designed a deterministic locking protocol to eliminate distributed commit protocols , but it enforced a global synchronization of transaction orders. SiloR [183] used OCC but required only exclusive locks on write items, while both shared and exclusive locks are requested in DTranx. SiloR would require two successful rounds of operations (exclusive lock, followed by read); in a distributed system, these two rounds would have significant latency due to RPC calls, but SiloR was implemented on a single computer and did not use RPCs. GMU [136] avoided read only transaction aborts by guaranteeing the Extended Update Serializability (EUS) isolation level, where read only transactions might observe snapshots from different linearizations of update transactions. This might work for some applications but as a fully ACID compliant KV store, DTranx enforces strong isolation. Other approaches that added serializability to snapshot isolation such as [172], [29] required a central server for validation in distributed environment.

We explored various distributed transaction designs and chose the system that combined OCC and 2PC since it yielded great performance and guaranteed strong consistency without special hardware such as atomic clocks.

Hardware-assisted Transactional System. With hardware advances like NVM, RDMA, software designs adopting these technologies show tremendous performance growth. NVM devices, such as PCM(Phase Change Memory), 3D XPoint, emerge and significantly reduce persistence

overhead. Tianzheng et al. [163] designed a scalable log leveraging NVM to support distributed logging where they focused on alleviating the contention bottleneck with passive group commit. According to our experiments, distributed transactional systems based on NVM logs yield such high throughput that the bottleneck resides in CPU processing. Hence, we focus on building an efficient GC supported NVM log. METRADB [113] was a middle layer that provided key value interfaces for applications and hided the complexities of using NVML to facilitate application development. However, we aim at providing an efficient garbage collection mechanism. On the other hand, HERD [85] focused on building key value services in memory using RDMA to reduce network round trips but lacked fault recovery over server failures. In the future, we plan to explore RDMA for lower latency RPCs.

3.7 Summary

We propose a fully ACID compliant transactional and scalable key value store that utilizes non-volatile memory based log with an effective and efficient garbage collection mechanism. To exploit the multi-core machines, we adapt the SEDA architecture with lock free queues and apply an optimal core binding strategy. Moreover, DTranx combines OCC and 2PC to move the locks to the commit time and employ parallel validation for better scalability. Evaluations show that DTranx offers higher throughput than the state-of-the-art system, Hyperdex, and DTranx also displays high scalability for various workloads. In the future, we plan to explore how to automate the core bindings to attain the best performance based on the number of CPU cores available.

With DTranx handling server side scalability and transaction challenges, next chapter illustrates a paradigm that transparently converts local data structure operations to remote transactions in DTranx.

Chapter 4

DDSBrick

4.1 Introduction

As the volumes of data increase in a rapid pace, there is a proliferation of big data frameworks such as Hadoop [168], Graphlab [105], and Spark [176] as well as distributed databases, such as Cassandra [93]. Despite being widely adopted to build scalable systems, most of the big data frameworks support coarse-grained data processing while fine-grained manipulations require delicate designs of customizing multiple distributed protocols. For example, Spark introduced distributed read-only objects called RDD, supporting only coarse-grained operators, such as map and reduce. Spanner [36] exploited Paxos state machines to manage transactions and Hyperdex [56] achieved consistency among replications by value dependent chaining, both of which involved vast design efforts. On the other hand, distributed key value stores support fine-grained operations and high scalability but offer low level abstractions. Since 1990s, researchers have exploited *distributed data structures*(DDSs) when single-threaded data structures yield low throughput. For example, the maximum throughput for a single-threaded R*-Tree [23] is 3.18 kops/sec. DDSBrick based RTree offers 35.36 kops/sec in a 25-server cluster and the throughput would continue to increase if more servers are added. Even though the throughput gain partly results from the larger size of the dataset used in DDSBrick based RTree, the difference would not have been so significant if it is not for the dynamic partitioning strategy of DDSBrick, which keeps the root nodes from becoming the bottleneck. DDS is a promising candidate to fill the gap between performance and programming abstractions. It is a higher level and expressive abstraction for scalable fine-grained systems. For

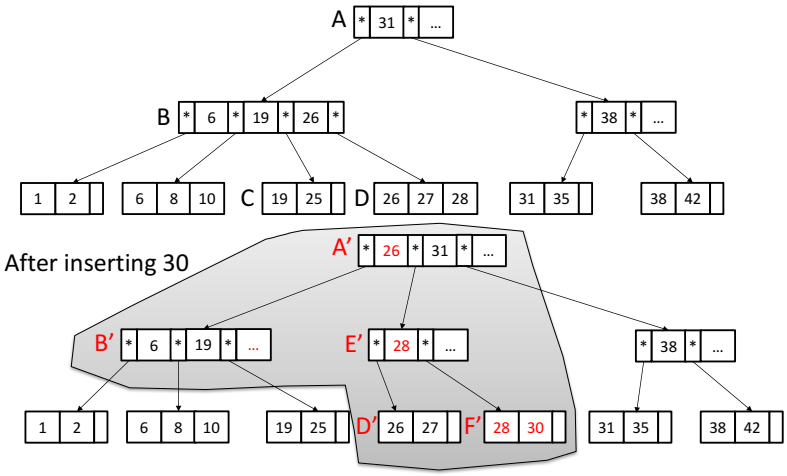


Figure 4.1: B+Tree insertion. The number of children varies from 2 to 4 except the root node. Red nodes are updated/inserted and the shaded area indicates the transaction boundary within which all nodes are updated/inserted atomically.

instance, Boxwood [111] used a BTree service as the fundamental storage infrastructure, simplifying the NFS file service design. Similarly, DDSBrick is intended to do the same for various kinds of data structures.

Despite the potential benefits of the DDS abstraction, designing and implementing error-prone and efficient DDSs take tremendous efforts. Specially, consistent accesses are necessary conditions for correctness. Figure 4.1 shows B+Tree states before and after an insertion operation. During the insertion, leaf node D overflows and the splitting process propagates from the bottom up, resulting in 5 node updates in total. Without consistent and atomic guarantees, a concurrent search operation of value 27 might go through A->B'->C, thus missing the update. Other problems related to the DDS design are data placement and data replication. In Figure 4.1, root node A is accessed by every operation while the leaf nodes, such as C are seldom accessed, necessitating a sophisticated replication strategy to spread loads for A and minimize replication costs for C. In general, there are three major challenges: first, it is error-prone to tailor and combine low level distributed protocols, such as consensus; second, it is non-trivial to implement efficient protocols from scratch, such as two-phase commit(2PC); third, diverse workload patterns require adaptable

partitioning and replication strategies to reach higher scalability. Researchers from different groups have implemented customized DDSs for special-purpose applications. Diegues et al. implemented STI-BT [46], an efficient B+Tree with extended update serializability. Minuet [150] is a distributed BTree that supports writable clones. Each of the DDSs above took months to design, implement, test, and tune for optimal performance.

Therefore, we propose DDSBrick, a paradigm for building scalable and transactional DDSs. DDSBrick reduces development efforts by taking care of distributed data placements and consistent accesses while providing intuitive programming interfaces. DDSBrick enables researchers to experiment on multiple data structures for various applications in a short period while ensuring effective performance. It separates specific data structure designs from distributed system complexities by injecting Mempool and adopting a distributed key value(KV) store as the backend storage system. Mempool transparently synchronizes local states of data structures with the distributed KV store such that application designers need only focus on the interactions between local data structures and Mempool. Mempool handles two tasks: address translation from local memory to storage systems and server synchronization.

Moreover, our KV store supports snapshot read transactions that maintain high throughput even when update transactions are constantly modifying the data structures. One scenario of using snapshot reads is to search for nearby taxis in a dynamically updated taxi geolocation database. The snapshot of taxi geolocations 5 seconds ago suffices for many applications, such as Uber. By bypassing expensive 2PC protocols, the KV store offers efficient snapshot operations without interfering with transactional operations. Our RTree snapshot search throughput is 3.17 times faster than that of transactional search when there are 6.82k update operations per second.

We also integrate a dynamic partitioning strategy into the KV store to reduce inter-server communication and balance resource usage. Traditional key value stores do not yield good performance when workloads display non-uniform access patterns. For example, the server storing the root of a binary tree is visited by every request. The KV store adopts a hybrid strategy that combines hash and adaptable partitioning. DDSBrick represents transactions with a graph struc-

ture and applies a partitioning algorithm called METIS [65] to minimize edge cuts. It brings two benefits. First, data that are jointly accessed in transactions are stored together such that no 2PC is necessary. Second, data loads are effectively replicated and balanced across the cluster. In addition, we introduce a greedy algorithm to minimize data migrations and adjust 2PC to enable online database migration while still serving strongly consistent transactions.

In summary, our contributions are as follows:

- Introducing DDSBrick, a paradigm to build scalable and transactional distributed data structures;
- Providing snapshot read transactions to bypass distributed transaction overhead;
- Integrating a dynamic partitioning strategy to balance resource usage based on distinct workload patterns;
- Implementing two practical DDSs, BTree and RTree, that exhibit high performance and high scalability; and
- Seamlessly synthesizing the techniques from database and distributed system fields to one piece of software.

4.2 Overview

DDSBrick follows the client-server model where clients run data structure algorithms and servers host a distributed storage system, such as RamCloud [132] and Tango [21]. Servers store the data structures and provide the transactional interfaces while application developers implement the DDSs based on a client library, which synchronizes local data structures with servers. The client-server model brings two benefits. First, the separation of the data structure logic from the storage system enables us to leverage techniques from distributed databases to achieve low latency, high throughput, and linear scalability. Second, it simplifies DDS implementations by hiding distributed system complexities, such as data placements.

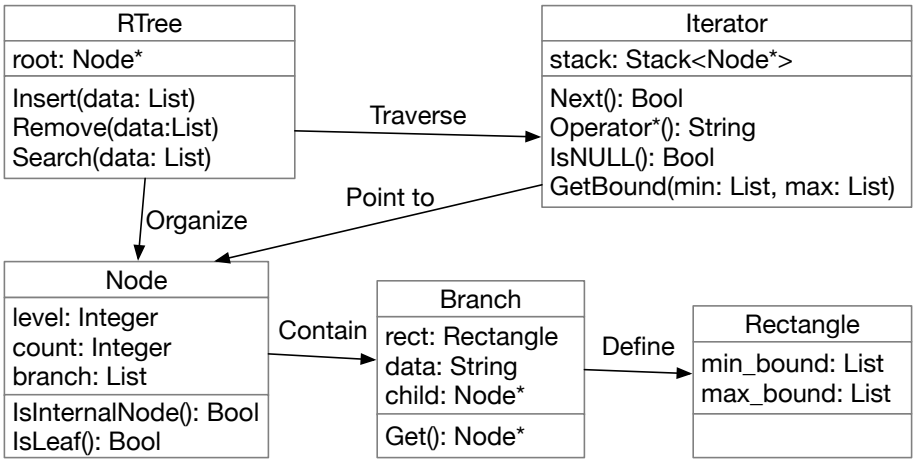


Figure 4.2: RTree Class Diagram. UML class diagram for RTree implementation.

On the client side, developers are required to decompose data structures into basic data units called data bricks. For example, the data bricks of binary trees are tree nodes and the data bricks of hash tables are key ranges of a fixed length. As shown in Figure 4.2, a RTree consists of Nodes, i.e. data bricks, and the algorithm defines how Nodes are organized to support operations, such as Insert, Remove, and Search. In this paper, we assume that the local data structure designs follow Figure 4.2 since the divisibility of data structures into data bricks is a prerequisite for DDSs. To minimize developing efforts, DDSBrick introduces Mempool to move data bricks between local machines and servers transparently. In addition, Mempool calculates a distinct address for each data brick automatically. With transparent server synchronization and address translation, the conversion from single-threaded data structures to distributed versions becomes effortless.

On the server side, DDSBrick depends on a scalable and transactional key value store called DTranx [63] where each data brick is stored as a key value pair. DTranx servers accept read requests and commit requests of which the former reads from DTranx optimistically and the latter initiates the serialization process. We choose DTranx instead of document databases or graph databases since the mapping of data bricks to keys is more straightforward, reducing address translation overhead; And, there are more partitioning algorithms available in KV stores, such as consistent hashing and content addressable network [137]. Moreover, DTranx adopted non-volatile memory(NVM)

Table 4.1: DDSBrick programming interfaces.

API	
Access	*New*: allocates new bricks. *Delete*: deletes old bricks. *GetBrick*: retrieves bricks from Mempool or DTranx. *GetRoot*: retrieves root from Mempool or DTranx. *AddUpdate*: notifies Mempool of a modified item.
Transaction	*Commit*: commits transactions. *Abort*: aborts transactions.
Hook	*(De-)Serialize*: serialization. *CopyBrick*: clones a data brick. *IsBrickDifferent*: checks if two bricks are different. *ResetPointerKey*: resets memory pointers in bricks.

to minimize database logging overhead. On top of DTranx, DDSBrick introduces snapshot read transactions and integrates a dynamic partitioning strategy. Snapshot read transactions offer high throughput when there are ongoing update transactions and the dynamic partitioning strategy generates data placement rules to balance load effectively. The dynamic partitioning strategy prevents the whole system from being bounded by slower and more heavy-loaded machines. Since DDSBrick is able to generate nearly optimal data placement rules according to history workloads, we do not expose interfaces to clients for customized data placements.

4.3 Paradigm

On the client side, Mempool is designed to handle two critical tasks: address translation and transparent server synchronization, of which the latter further includes transaction construction and data synchronization. Mempool presents an in-memory transactional KV store to the data structures and defines a set of interfaces.

4.3.1 Programming Interface

The APIs are divided into three categories: *Access*, *Transaction*, and *Hook*, as shown in Table 4.1. *Access* APIs are memory allocation functions and brick retrieval functions that bridge local memory and DTranx. For example, *GetBrick* fetches data bricks, de-serializes them, and

returns an in-memory instance; *Transaction* APIs are transaction primitives to commit changes consistently. Client operations, such as BTree insert and RTree search should be guarded with transaction primitives; *Hook* APIs are callback functions to be implemented in data bricks. For example, Mempool calls *Serialize* to convert in-memory data bricks to byte strings when committing transactions. We illustrate how to convert the local RTree in Figure 4.2 to a distributed RTree in four steps.

Step 1: Define persistent pointers

To persist the in-memory pointers in data bricks, developers define an extra string field(persistent pointers) for each in-memory pointer. Then, the persistent pointers are serialized and stored in DTranx. For the RTree example, a string field is defined in Branch class for child nodes. When a data brick is retrieved from DTranx, its persistent pointers are fetched while the in-memory pointers are initialized to NULL. The NULL value indicates that the corresponding data bricks are not in local memory. As more data bricks are referenced and retrieved, the in-memory pointers are assigned gradually. Whenever the in-memory pointers are changed, persistent pointers are updated accordingly.

Step 2: Intercept data brick accesses

Data brick accesses consist of reads, creations, and updates. Updates are discussed in Section 4.3.2. Developers define read methods and creation methods in data bricks to return neighbor bricks and newly allocated bricks. For example, *GetChild* and *NewLeaf* are defined in the Node and Branch class. Then, the read and creation methods call *Access* APIs, such as *GetBrick* to communicate with DTranx. For example, read methods receive persistent pointers as parameters and call *GetBrick* if the corresponding in-memory pointers are NULL. Finally, developers replace data brick reads and creations in data structure implementations with the read and creation methods. For example, RTree calls *GetChild* instead of referencing the child member in Branch. Since *Access* APIs are monitored by Mempool, data brick reads and creations are intercepted.

Step 3: Wrap client operations with transactions

Client operations, such as RTree REMOVE should be wrapped in transaction primitives.

In Algorithm 1, RTree REMOVE calls *Commit* after local erase is complete. If *Commit* succeeds, REMOVE returns; If *Commit* fails, REMOVE retries; If any exceptions are thrown when invariants are violated, *Abort* is called. Clients may observe violated invariants since single-threaded data structures might assume certain invariants that are not true in concurrent environments. For example, optimistically read bricks might be partially modified by another committing transaction in DTranx. Developers are required to implement the invariant checking in the read methods defined in Step 2. For example, *GetChild* in Node checks if its level is larger than that of the fetched child. However, invariant checking is disabled if the in-memory pointer is not NULL, in which case *GetBrick* is not called. Moreover, transactions are early aborted in case of invariant violation, thus preventing 2PC from being triggered.

Algorithm 1 Distributed RTree REMOVE. Lines starting with # are added for distributed RTree.

```
 1: procedure REMOVE(key)
 2:     #root ← MEMPOOL.GETROOT()
 3:     result ← True
 4:     while True do
 5:         try
 6:             iter ← FIND_UNIQUE(key, root)
 7:             if iter.node == Null then
 8:                 result ← False
 9:             else
10:                 result ← True
11:                 erase(iter)
12:             end if
13:             if #MEMPOOL.COMMIT( ) then break
14:             end if
15:             #root ← MEMPOOL.GETROOT()
16:         catch #Util::InConsistencyException
17:             #MEMPOOL.ABORT( )
18:             #root ← MEMPOOL.GETROOT()
19:         end try
20:     end while
21:     return result
22: end procedure
```

Step 4: Implement Hook functions inside data bricks

Mempool depends on user-defined *Hook* APIs to serialize data bricks and detect updated bricks in transactions. The first set of *Hook* APIs, namely *Serialization* and *Deserialization*, are called when Mempool retrieves or updates data bricks. *Serialization* writes all fields including per-

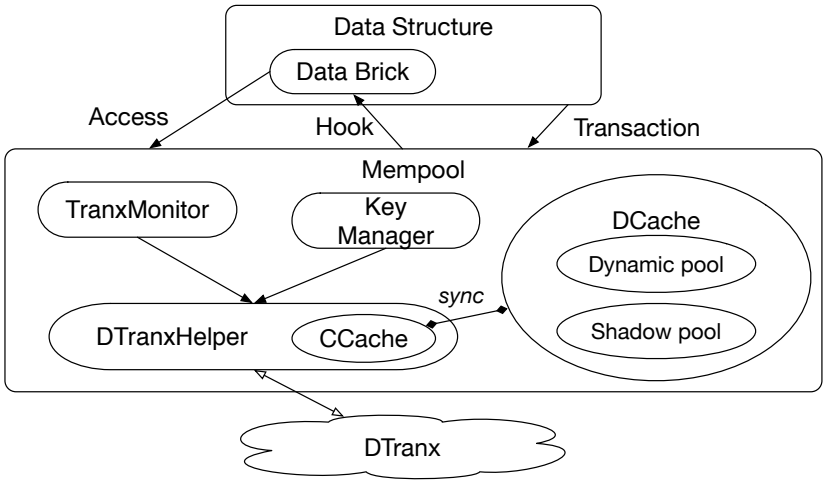


Figure 4.3: Mempool design. It shows the internal modules of Mempool and the interactions among these modules.

sistent pointers. The second set, consisting of *CopyBrick*, *IsBrickDifferent*, and *ResetPointerKey*, assist Mempool in constructing transactions. The *Hook* APIs are discussed in detail in Section 4.3.2.

4.3.2 Mempool Design

Mempool addresses three main challenges: address translation, transaction construction, and data synchronization. First, data bricks are assigned with distinct keys by concurrent clients. Second, data brick reads, creations, and updates are detected and transactions are constructed automatically. Third, data bricks are synchronized to DTranx. Correspondingly, Mempool contains three modules as depicted in Figure 4.3: KeyManager, TranxMonitor, and DTranxHelper. KeyManager manages the key space and generates unique keys for new data bricks. KeyManager follows RFC4122 standards [133] to generate unique keys and a special key is hard coded

in Mempool to handle metadata information, such as RTree root key. TranxMonitor constructs transactions. And, DTranxHelper provides the interface to DTranx.

TranxMonitor is capable of capturing data brick reads, creations, and updates. As mentioned in Section 4.3.1, data brick reads and creations are intercepted in *GetBrick* and *New*. Data brick update detection relies on a Mempool shadow method, which maintains two memory pools: a shadow pool and a dynamic pool. As shown in Figure 4.3, the shadow pool is a shadow copy of the data structures before the current transaction changes any bricks while the dynamic pool is actively read and modified. Both pools are empty initially. When data bricks are retrieved from DTranx, Mempool copies them to both pools and returns the copy in the dynamic pool to the data structures. With the shadow technique, Mempool captures data brick updates by comparing the bricks in the dynamic pool to the ones in the shadow pool. During commit time, Mempool compares the two data bricks by calling *IsBrickDifferent* in *Hook* APIs. However, the shadow method leads to constant copying operations. Programmers might choose to call *AddUpdate* manually whenever bricks are changed to bypass the shadow pool copying. The trade-off of programming simplicity and low overhead is left to the users. Above all, the overhead is on the client side and the scalability of DTranx is not compromised.

DTranxHelper incorporates a caching mechanism to obviate the invalidation of the whole pools. Otherwise, pools are invalidated between transactions and data bricks are fetched, deserialized, and replicated in each transaction. In Figure 4.3, CCache is the client side cache maintained by the DTranx client library and DCache is the deserialized Mempool. CCache and DCache are kept synchronized since the versions of optimistically read bricks, which are compared in 2PC, are only stored in CCache. Similar to the CCache policy [63], the DCache policy works as follows. If read requests succeed, DCache inserts the read bricks; If commit requests succeed, newly allocated bricks are deleted in DCache since the version numbers of new bricks do not exist in CCache; If commit requests fail, DCache invalidates all read bricks in failed transactions. Between transactions, there are two tasks. the dynamic pool is synchronized to the shadow pool to remain consistent. And, Mempool calls *ResetPointerKey* in *Hook* APIs to reset in-memory pointers to NULL such

that *GetBrick* is called in later transactions, capturing data brick reads.

4.4 Key Value Store

While DTranx offers transactional key value accesses with high performance, it fails to scale for workloads with hot spots. We integrate a dynamic partitioning strategy to address the hot spot issue and introduce *snapshot read transactions*(SRTs) that boost read transaction throughput when there are update operations. Furthermore, we customize a 2PC variant in Section 4.4.2.4 to ensure consistency while DTranx is updated and migrated concurrently.

4.4.1 Snapshot Read Transaction

While serializable transactions simplify application development, it is expensive to run 2PC protocol that involves remote procedure calls and write ahead log(WAL). SRTs provide consistent views to DDSs and reduce lock contention by skipping 2PC. We first discretize time into epochs during which there is at most one version for each data item. Normal transactions operate on the data in the active epoch while SRTs operate on data in old epochs. Since data in old epochs are read only and consistent, read-write and write-write conflicts avoidance is unnecessary between SRTs and normal transactions.

Database snapshots are created by *snapshot creation transactions*(SCTs). A special data item called *Epoch* is added to the database and replicated in all servers to denote the current epoch number. *Epoch* is also embedded into the version numbers to distinguish different snapshots and enable old snapshots reclamation. To resolve the conflicts between normal transactions and SCTs, normal transactions acquire shared locks on *Epoch* and SCTs acquire exclusive locks on it. The moment when SCTs obtain locks in every server is the serialization point after which copy-on-write technique is applied. SCTs increment *Epoch* but do not reclaim the old snapshots since the reclamation process could take long time when normal transactions are blocked. Instead, the old snapshots are reclaimed when database read operations find more than N versions. N is configured to reach the balance between disk usage and the number of old snapshots stored. For example,

N being 2 leads to one snapshot stored. Furthermore, if a data item is not updated during the current epoch, databases will not contain the item with the current epoch number. SCTs avoid deep-copying by inheriting the old data versions implicitly. For example, if *Epoch* is 5 and data item A contains versions in epoch 2 and 3, the version in epoch 3 is also interpreted as the versions in epoch 4 and 5.

Livelocks are impossible since exclusive lock requests from SCTs are blocked indefinitely while shared lock requests from normal transactions are non-blocking. During the snapshot creation, the normal transactions that hold shared locks on *Epoch* continue to completion while the normal transactions yet to request locks are aborted. We avoid deadlocks from multiple concurrent SCTs by allowing one SCT at a time. SCTs are initiated only at the master server and new SCTs are not started until the previous one commits. The master server is selected based on its rank in the group membership that is stored in a replicated state machine [130].

4.4.2 Dynamic Partitioning

Our dynamic partitioning strategy combines hashing and graph partitioning algorithms and takes replication into account. We do not offer high availability but apply replication for better performance. Our approach customizes data placement rules based on data access patterns. As a result, each data item has its own replication factor and data that are frequently accessed together are placed in the same servers. This fine-grained data placement method achieves well balanced resource usage while incurring little mapping data that record the data placement rules. To further reduce the mapping data size, data that are involved in less than a configured number of transactions are filtered out since they are seldom accessed.

Initially, data are assigned to servers based on their hash value. Then, distributed transactions are recorded, mapped to a graph structure, and input into METIS [65] to generate new placement rules. Finally, the new placement rules are installed in all servers, followed by the database reorganization.

4.4.2.1 DTranx Modification

There are two modifications to the original DTranx system: mapping data management and replicated data locking. First, mapping data is fine-grained data placement rules. For example, the mapping data for the RTree root is {X,Y,Z}, indicating that the RTree root is replicated in Server X, Y, and Z. We insert a "mapping read" step in coordinators during 2PC, as shown in Figure 4.5. In addition, a mapping cache is created on DTranx clients to route read and commit requests. The mapping cache is updated when either read or commit requests are forwarded, in which case DTranx servers piggyback the new mapping data in the reply messages. Second, we adjust the locking service to guarantee the consistency among replicated copies. Let's denote N as the replication factor for data D, S as the number of locks required for shared locking, X as the number of locks for exclusive locking. The following two restrictions are necessary for strong consistency: $S + X > N$ and $2X > N$. In DDSBrick, they are configured as $S = 1$ and $X = N$ to reduce the locking costs for hot spots, such as RTree root.

4.4.2.2 Graph Construction

Past transactions in WALs are collected and broadcast to the master server before DTranx reclaims the WALs. When the number of past transactions exceeds the configured threshold, the master server initiates the dynamic partitioning process. In the following, we use "dynamic partitioning" and "repartitioning" interchangeably.

In Figure 4.4, there are two kinds of vertices: replication vertex and central vertex. Replication vertices are created per transaction per data item and the vertices that represent the same data item are connected to the central vertices. Similarly, there are two kinds of edges: replication edge and transaction edge. Replication edges connect replication vertices to central vertices while transaction edges connect replication vertices of different data items in the same transactions. Edge weights are set based on the edge kind: replication edges weights are the update costs, meaning the number of transactions that update the data items; transaction edges are configured as the

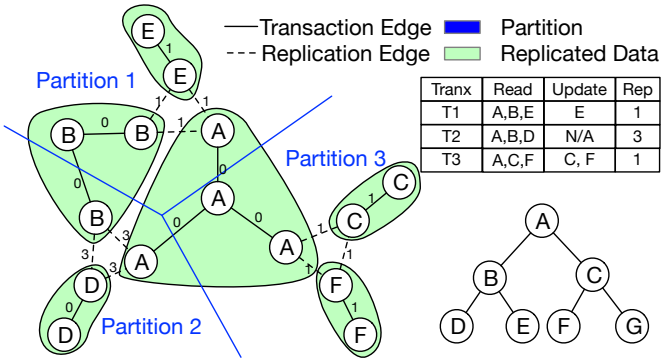


Figure 4.4: Transaction graph representation. It is a binary search tree on the bottom right and the transactions on the top right are the input for the graph generated on the left.

number of transactions that co-access the data items. For example, T2 is executed for 3 times and the transaction edge weights between {A,B}, {A,D}, and {B,D} are 3.

After the graph is constructed, the METIS algorithm partitions the graph structure, minimizing the edge cuts between partitions. It has four indications. First, frequently read data are replicated since the replication edge weights are low while frequently updated data are not replicated. This effect distributes read request loads and reduces exclusive lock overhead. Second, data that have been accessed together in transactions are stored in the same server such that no distributed commit protocol is necessary. Third, METIS generates a fairly balanced partitioning strategy. Lastly, METIS supports configurable partition numbers, supporting cluster membership changes. However, the running time of METIS grows substantially as the graph expands. To reduce the graph size, past transactions are sampled to balance the partition effectiveness and runtime costs.

4.4.2.3 Greedy Optimization

Different mappings of the partition IDs from the METIS algorithm to servers result in different data migration sizes. In the worst case, every key value pair is moved. Worse still, the mapping data size would be enormous. We designed a greedy algorithm to minimize the migration traffic as shown in Algorithm 2. *oldMapping* is a hashmap from data keys to a list of ServerIDs where date

currently reside and *newMapping* is a hashmap from data keys to a list of new partitionIDs from METIS. The output of Algorithm 2 is a mapping from partitionIDs to ServerIDs. The core idea of the greedy algorithm is to match servers in the *oldMapping* with those in the *newMapping* that share the most data items.

Initially, servers in the *oldMapping* are sorted by the number of keys they contain. Server X with the most keys are chosen and for each server Y in the *newMapping*, the number of keys shared by X and Y is calculated. Then, the new server Y' that shares the most keys with X is chosen to map to X.

Algorithm 2 Optimize Data Migration

```
1: procedure OPTIMIZE(oldMapping, newMapping)
2:     resultMapping ← {}
3:     serverBags ← {}
4:     serverOrder_old ← Ø
5:     for key ∈ oldMapping do
6:         for server ∈ oldMapping[key] do
7:             serverBags[server].add(key)
8:         end for
9:     end for
10:    serverOrder_old ← OrderServer(serverBags)
11:    while !serverOrder_old.empty() do
12:        srv_old ← serverOrder_old.front()
13:        serverOrder_old.pop()
14:        allKeys ← serverBags[srv_old]
15:        serverFreq_new ← {}
16:        serverOrder_new ← Ø
17:        for key ∈ allKeys do
18:            for server ∈ newMapping[key] do
19:                serverFreq_new[server] + +
20:            end for
21:        end for
22:        srv_new ← FindBest(serverFreq_new)
23:        resultMapping[srv_new] ← srv_old
24:    end while
25:    return resultMapping
26: end procedure
```

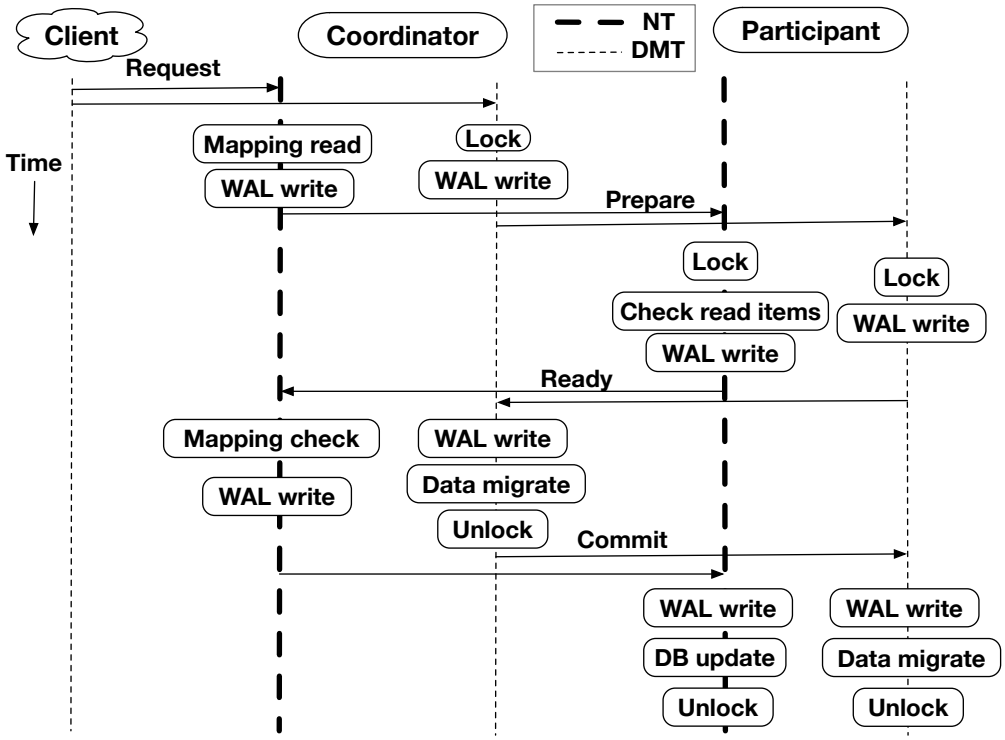


Figure 4.5: 2PC for NT and DMT

For S servers and K keys, the time complexity is

$$T(K,S) = \begin{cases} O(K) & K \gg S, \\ O(SK + SlogS + S(SK + SlogS)) & \text{otherwise.} \end{cases}$$

The memory complexity is

$$M(K,S) = \begin{cases} O(K) & K \gg S, \\ O(SK + S) & \text{otherwise.} \end{cases}$$

In real scenarios, K is much greater than S, in which case, both the time and memory complexities are O(K).

4.4.2.4 Data Migration Transaction

To install the new placement rules and migrate the database consistently, we customize a variant of 2PC. Let's denote normal transactions as NT and data migration transactions as DMT.

As shown in Figure 4.5, NT obtains the participant list from the mapping data and checks if they change during 2PC. DMT follows 2PC and replaces the database update with data migration. The serializability challenge is decomposed into three conflict cases: NT with NT, DMT with DMT, and NT with DMT.

Case 1: DMT-DMT Conflict

We prevent DMT-DMT conflicts by allowing one DMT at a time. Similar to SCTs in Section 4.4.1, the master server initiates the DMT and guarantees that DMTs are not started until the previous one completes.

Case 2: NT-DMT Conflict

Suppose that a NT "T1" updates DataA in ServerX and a DMT "T2" migrates DataA from ServerX to ServerY. T1 updates DataA in ServerX while T2 migrates the old version of DataA to ServerY, causing lost updates. We prevent NT-DMT conflicts by capturing read-write and write-write conflicts. Specifically, the database and the mapping data share the locking service and DMTs request exclusive locks on the corresponding data. Therefore, either T1 or T2 would abort due to write-write conflicts in the previous example.

Case 3: NT-NT Conflict

NT-NT conflicts happen when there is a concurrent DMT since NTs read mapping data before locks are granted. In Figure 4.6, data A is migrated from server X to server Y and Z. T1 requests exclusive locks in X where old mapping data is referenced, while T2 requests exclusive locks in Y and Z where the new mapping data is referenced. These two conflicting transactions are able to access the same data concurrently. Even though servers in the new mapping are storing consistent copies, clients with outdated mapping data might read A in X optimistically. To prevent NT-NT conflicts, mapping data for all items in NTs are checked after locks are granted in Figure 4.5. This checking guarantees that no intervening DMTs are running.

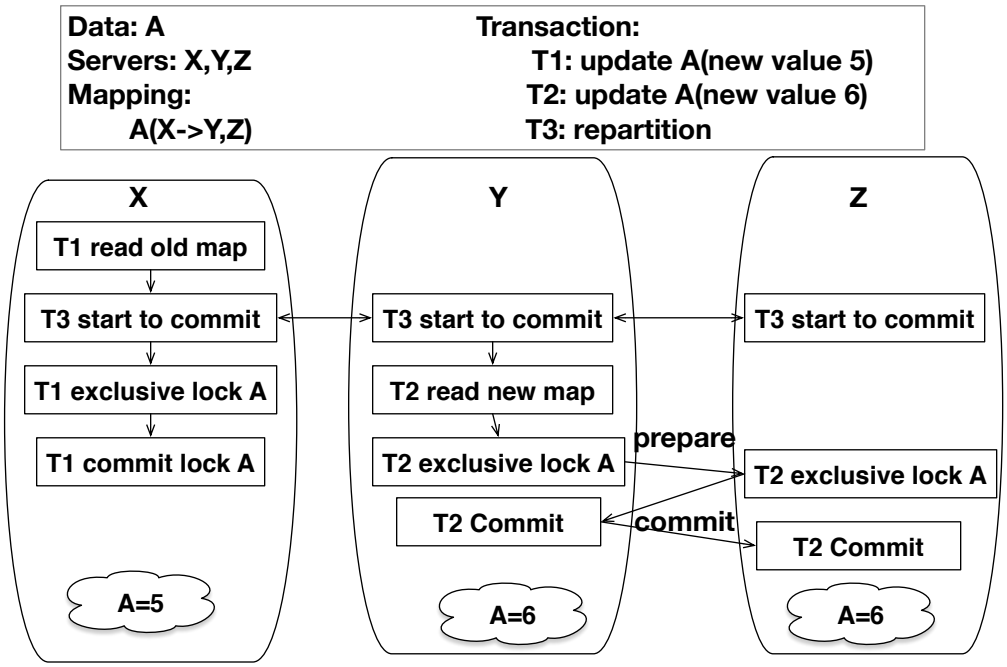


Figure 4.6: Repartition failure scenario. Arrows represent a happens-before relationship.

4.5 Evaluation

Our benchmark tests are run on Cloudlab [138] where each machine is equipped with 2 Intel E5-2660 v3 CPU(each with 10 2.6GHz cores), 130GB RAM, 480GB SSD at 6GB/sec, and 10Gbps Ethernet card. DAX support in Linux is enabled to emulate NVM. We used DRAM based emulated NVM since NVM was not available during the tests and current NVM yields latency comparable to DRAM. For example, STT-RAM [17] achieves ∼10ns write latency compared to 50ns for DRAM. In addition, current NVM offers far more throughput than what DTranx requires.

Distributed Data Structure we have transformed both an in-memory BTree [6] and RTree [71] to distributed versions. To generate workloads, we run the YCSB C++ version [35] and added BTree and RTree support. The max numbers of children for both BTree and RTree

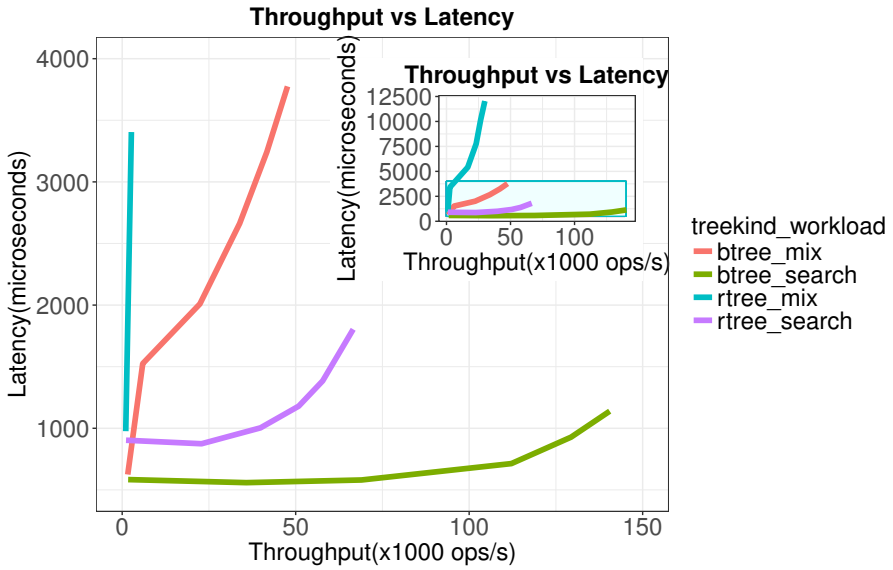


Figure 4.7: Throughput and Latency. There are four types of workloads: BTree search operations, BTree mix operations, RTree search operations, and RTree mix operations.

are configured as 30. YCSB clients are run in separate machines from the cloud servers that accommodate DTranx.

Workloads BTree and RTree are initialized with 1, 3, 6, and 8.5 million data items in 3, 9, 18, and 25-server clusters respectively. BTree stores 64-bit integers that are randomly generated from a uniform distribution function between -X to +X where X is 100, 300, 600, and 850 million for the different cluster sizes above. RTree stores three-dimensional cubes which are constructed with a coordinate of one vertex and a side length. Similar to BTree, the coordinates are randomly generated from a uniform distribution function. The side length is randomly generated from a uniform distribution function between -10 and +10. we choose the uniform distribution instead of zipfian since the uniform distribution of workloads to BTree or RTree is translated to a nonuniform distribution of workloads to DTranx. For all the following tests, the initial results during cache population are discarded and the average of 3 runs are reported with variances.

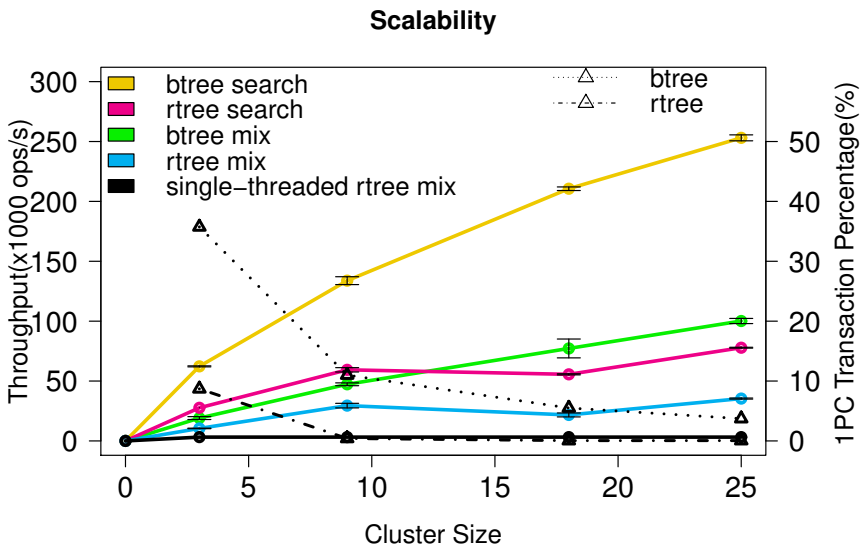


Figure 4.8: Scalability. Cluster sizes are 3, 9, 18, and 25. The right vertical axis is the percentage of distributed transactions that are committed in one server. The bottom baseline is the throughput for a single-threaded rtree.

4.5.1 BTree & RTree

In this section, we evaluate the throughput and latency of the distributed BTree and RTree as well as their scalability. Workloads are categorized into search ones and mix ones of which the former means search operations and the latter means 70% search, 15% remove, and 15% insert operations.

Throughput vs. Latency In Figure 4.7, throughput and latency are recorded in a 9-server cluster as we vary the number of client threads. The latency increases slowly in the beginning and soars up when the throughput approaches the maximum since more clients introduce higher contention in DTranx. However, there is a turning point for mix workloads when the throughput is low. The turning point is when the number of client threads changes from one to five where exclusive lock contention takes effect. Nonetheless, RTree mix workloads yields 7.77ms latency when the throughput hits 23.19 kops/sec. On the other hand, search workloads generally yield higher throughput and lower latency compared to mix workloads since they request non-contentious

shared locks. Compared with RTree, BTree yields lower latency for the same workloads since the overlapping regions in RTree leads to more data items accessed in transactions, thus more servers involved in 2PC. In the future, we plan to examine the RTree splitting algorithm to reduce the internal nodes overlapping.

Scalability In Figure 4.8, the throughput of BTree search workloads and RTree mix workloads reaches 253.07 kops/sec and 35.36 kops/sec respectively on a 25-server cluster. Both of them show linear scalability since the dynamic partitioning strategy spreads hot spot loads. However, RTree performance degrades from the 9-server cluster to the 18-server cluster. By inspecting the number of servers involved in RTree transactions, we found out that 7.19 servers in the 18-server cluster are involved on average in a single transaction compared to 4.4 servers in the 9-server cluster. The impact from more servers involved in transactions outweighs the benefits of extra servers alleviating the loads. In the 25-server cluster, RTree transactions involve 8.12 servers where the benefit of 7 more servers overshadows the impact of 0.93 more servers involved in transactions. On the other hand, the one-phase commit(1PC) transaction percentage decreases as the cluster size grows since database are spread out across more servers. And, BTree shows more 1PC transactions than RTree since BTree operations access fewer data items. Finally, to validate that better programmability is not achieved at the sacrifice of performance, we compared the performance of BTree read transactions and key-value read transactions to DTranx. Given that both BTree and key-value read transactions involve 5 data items on average(it is a 5-layer BTree), the throughputs differ within 8%. Otherwise speaking, DDSBrick-based BTree fully utilizes the underlying KV store.

Table 4.2: Tango vs. DDSBrick

Throughput	TangoMap	DDSBrick RTree	DDSBrick BTree
8/9-server	25k (est.)	29.48k	47.29k
18/25-server	25k (est.)	35.36k	100.13k

Tango Although there are no open source DDSs to compare with, we conduct a qualitative analysis between DDSBrick and Tango [21] in Table 4.2. The transaction throughput for a single

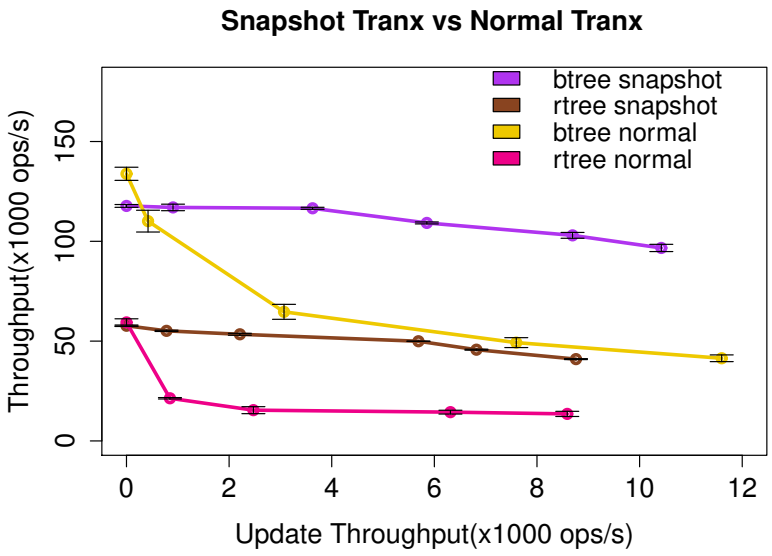


Figure 4.9: Snapshot throughput.

TangoMap object is around 15 kops/sec and 25 kops/sec in a 8-server cluster when keys are selected with zipfian and uniform distribution. DDSBrick RTree throughput for mix workloads reaches 29.48 kops/sec in a 9-server cluster. First, a fair comparison should be between TangoMap with zipfian distribution and DDSBrick RTree with uniform distribution since the latter generates highly skewed workloads to the backend storage. Second, 25 kops/sec is the maximum throughput for a single TangoMap since system wide throughput is limited by the speed at which a single client can play the logs. Third, even if data are partitioned across multiple TangoMap and transactions are constructed among the TangoMaps, the cross-partition throughput reaches around 25 kops/sec in a 18-server cluster, compared with 35.36 kops/sec for DDSBrick RTree in 25-server cluster. Tango depends on the creation of more partitions to further improve the overall performance, but the throughput for transactions across partitions is low. Based on this analysis, DDSBrick offers slightly better performance and it effectively alleviates development efforts for building DDSs.

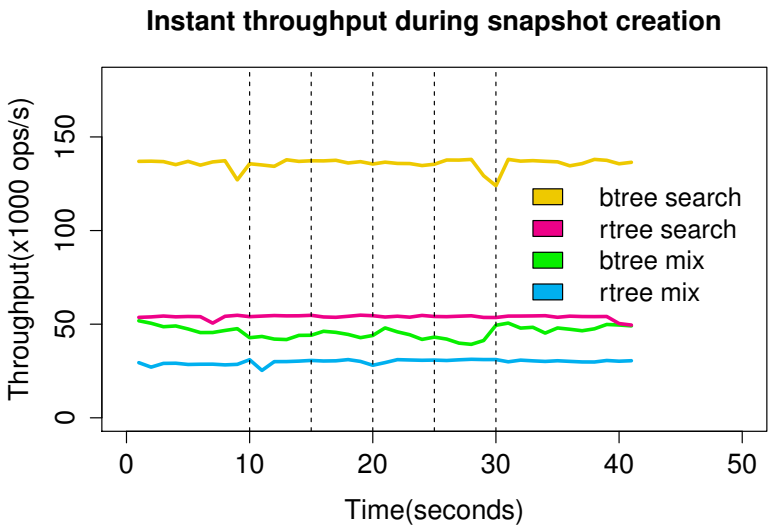


Figure 4.10: Snapshot creation. The vertical dotted lines are when new snapshots are created.

4.5.2 Snapshot

To confirm the efficiency of SRTs, we run a background update workload, consisting of 50% remove and 50% insert operations, in a 9-server cluster. In Figure 4.9, both SRT and normal read transaction throughputs are measured as different update workloads are running. Normal read throughput is severely affected since update transactions are contending for locks while SRT throughput is lightly affected as no locks are requested. Nonetheless, SRTs call for database reads, causing a slight performance drop. Furthermore, SRTs do not outperform normal read transactions when there are no concurrent update transactions since clients disable the local cache such that SRTs can get the most recent snapshot. For example, the throughput for BTree normal transactions(137.53 kops/sec) is 16.82% higher than that of BTree SRTs(117.73 kops/sec), implying 588.65 kops/sec optimistic reads(it is a 5-layer BTree).

On the other hand, we measure the instantaneous throughput as new snapshots are continuously created. In Figure 4.10, both search and mix workloads are run for BTree and RTree where snapshots are created every 5 seconds. Neither BTree nor RTree throughput is affected since the snapshot creation merely updates *Epoch* and old snapshots are not immediately cleaned. The

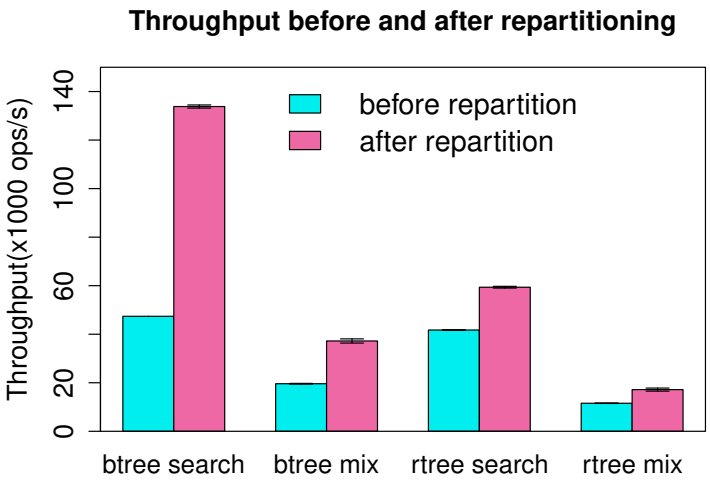


Figure 4.11: Throughput before and after DMT.

throughput for BTree mix workloads oscillates more than others since the number of client threads spawned during the tests exceeds the balancing point where the servers start to saturate. Since the instant throughput is fairly stable, we did not tune the client thread number.

4.5.3 Dynamic Partitioning

To evaluate if our dynamic partitioning strategy boosts performance, we run three tests: the first test compares BTree and RTree throughput when dynamic partitioning is enabled with the ones when it is disabled; the second test examines the service request numbers on the servers to find out why the dynamic partitioning increases throughput; the last test studies how DMTs affect ongoing transactions. For all three tests, both BTree and RTree are initialized with 3 million data items in a 9-server cluster and both search and mix workloads are run.

In Figure 4.11, Both BTree and RTree benefit from dynamic partitioning for search and mix workloads since data are replicated based on past transactions and loads are balanced across the cluster. BTree search throughput after database re-organization is 2.8 times higher than the one before. Moreover, the throughput gain of search workloads is greater than that of mix workloads

Figure 4.12: Server side requests metrics. We normalize the request numbers by dividing them by the average number of ClientService requests. And, data are sorted in increasing order such that the number of requests is the lowest in the server with ID 1.

since the former benefits from both data replication and data affinity while the latter suffers from data replication.

Figure 4.12 shows the number of service requests and 1PC transactions. ClientService requests are commit and abort requests from clients; TranxService requests are prepare, commit, and abort messages among servers; StorageService requests are optimistic data reads from clients. First, all service requests and 1PC transactions are well balanced across the cluster after repartitioning. For example, the maximum number of BTree TranxService requests is 3.47 times of the minimum before the repartitioning and 1.04 afterwards; Second, the number of RTree TranxService

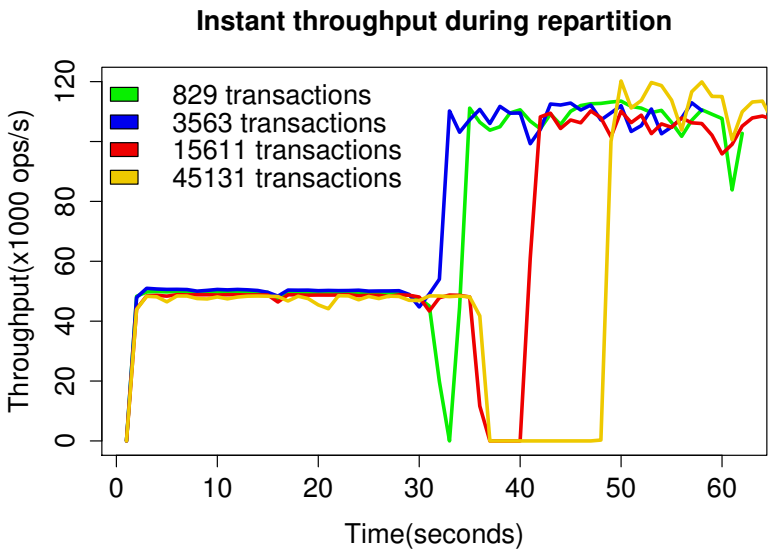


Figure 4.13: This figure shows how the repartitioning affects ongoing transactions for BTree. The number of transactions means how many history transactions are applied for the new partitioning strategy.

requests is larger than that of BTree after repartitioning since RTree children are overlapped and transactions involve more servers. It also explains why BTree throughput is higher than RTree; Third, the percentage of 1PC transactions for BTree increases by 10.88% compared to 0.31% increase for RTree during repartitioning. This increase contributes to the larger throughput increase of BTree than that of RTree; Fourth, the fact that the number of StorageService requests drops after repartitioning is coincidental. We happened to run more transactions after repartitioning and later transactions read data from the client cache. Lastly, ClientService requests are always balanced since clients choose servers randomly when 1PC is not possible.

Figure 4.13 shows how data migration affects ongoing transactions. The more transactions are input to METIS, the longer it takes to complete the transition since more data are relocated. For example, it took ≈ 10 seconds for 45131 transactions while it took less than a second for 3563 transactions. However, the throughput gain is similar for different numbers of history transactions. Fewer transactions are preferred to alleviate the negative impact on ongoing transactions. Repar-

titioning for 829 transactions took longer than that for 3563 transactions since the throughput metrics are measured once every second and the repartitioning time for both 829 and 3563 transactions is so short that random factors, such as garbage collection cast a greater impact on normal transactions.

4.5.4 Summary

We have learned the following lessons as we built DDSBrick and implemented data structures on top of the paradigm. First, there is a trade-off between automation and efficiency. If clients choose to notify DDSBrick of update items(less automation), DDSBrick can disable Mempool shadow copies(more efficiency). Second, cache pays off. For example, DCache reduces the serialization and memory copy overhead. And, CCache not only lowers the read and commit miss rates but also decreases the number of read requests to the servers. Third, partitioning strategy is critical for DDS scalability. On the other hand, DDSs, such as index trees show strong access biases where the top layer nodes are read more frequently than the bottom ones. Even for flat structured DDSs, such as distributed hash tables, they are likely to display hot spots in real world scenarios. METIS is able to adjust replication factors and balances loads on a fine level of granularity without incurring considerable mapping data.

4.6 Related Work

This paper presents a paradigm for building distributed data structures in a scalable and transactional manner while maintaining high throughput. We draw from past work in databases, scalable systems, and distributed transactional systems [114, 87, 68, 26, 172, 8, 36].

Distributed Data Structure. DDS was first introduced by Steven [68] where a distributed hash table was implemented for Internet services. The hash table was scalable, consistent, and fault tolerant, but it did not support transactional updates to multiple elements. Marcos **et al.** [9] developed a new paradigm called Sinfonia to simplify distributed system designs. By hiding the complexities of concurrency control and server failure recovery, Sinfonia's mini-transactions enabled

System \ Feature	Transaction	Load Balance	Snapshot	Paradigm
Sinfonia	Yes	No	No	Yes
Schism	Yes	Fine	No	No
DDSBrick	Yes	Fine	Yes	Yes
HashTable(Steven)	No	Hash	No	No
Minuet	Yes	No	Yes	No
Tango	Yes	Fine	No	Yes
STI-BT	Yes	Coarse	No	No

Figure 4.14: Feature supports for current DDS systems.

efficient and consistent access to data. However, the assumption that transaction data items are known *a priori* does not hold for many data structures. For example, B-tree [8] gradually picks up read and update items as the tree is traversed. Minuet [150], based on Sinfonia, built a consistent and efficient distributed B-Tree with dynamic transactions, but it did not handle workload hot spots. Recently, Tango [21] was introduced as a DDS paradigm that relied on a fault tolerant and durable distributed log, Corfu [20]. If the sequencer maintained offsets for large numbers of object streams, it would become the performance bottleneck.

Current DDS systems are compared in Figure 4.14. DDSBrick supports transactions, dynamic load balancing, snapshot operations, and a general paradigm for DDSs. STI-BT [46], a scalable and transactional index for distributed key value stores, divided a distributed BTree into a top layer tree and several bottom layer sub-trees where the top layer tree was fully replicated and the bottom layer sub-trees were partially replicated. Even though this static replication strategy outperformed hashing, it failed to dynamically adjust the sub-tree partial replications. Similar coarse grained partitioning DDSs are [50] [180].

Transactional Data Structures In the past decade, concurrent data structures are proposed to allow concurrent accesses without sacrificing performance. Examples are [79], [75]. However, the main drawback is the lack of transactional accesses to multiple elements in the data structures, such as atomically reading and writing two elements. Recently, TxCF-Tree[77] was

proposed to allow efficient transactional operations to multiple elements by separating structural operations, such as re-balancing from semantic ones. It minimized the interference between structural operations and the critical path of semantic operations but required a housekeeping thread to process the structural operations. TxCF-Tree optimized the housekeeping thread in a multi-core single machine, but it would introduce considerable overhead in distributed environments. Additionally, imposing the separation of structural operations from semantic ones left the design of the DDS paradigm interface specific to certain tree structures.

Data Placement Partition strategy plays a critical role in scalability since it balances resource usage and distribute loads. RAMCloud [132] stored the database in memory to lower latency but its hashing-based data placement did not suffice to handle hot spot keys. Schism [42] proposed to transform transactions to graphs and applied graph partitioning algorithms but it did not support online migration. Turcu [159] extended Schism to support independent transactions [37] and designed a machine-learning based mechanism for routing transactions. However, the mapping data size was small enough to fit into the client side cache and training and pushing the classifiers to clients resulted in overhead. DDSBrick combines the hashing and graph partitioning algorithms to generate data placement rules and keeps a client side mapping data cache for transaction routing. Additionally, DDSBrick designs a greedy algorithm to minimize data migrations that Schism lacks and adjusts 2PC to avoid interruptions to ongoing operations during data migration by resolving potential NT-NT, NT-DMT, DMT-DMT conflicts.

4.7 Summary

We have described the design of DDSBrick, a paradigm for scalable, transactional, and efficient distributed data structures. DDSBrick enables efficient conversion from local, non-concurrent data structures to distributed versions. We have shown that DDSBrick is simple to use and the converted data structures yield great performance and scalability. Currently, we are implementing a variety of data structures and exploiting hardware advances, such as RDMA to boost performance.

The next chapter reviews distributed spatial access methods during the last two decades. It

summarizes the most critical challenges and discusses the techniques proposed in these systems.

Chapter 5

Related Work and Discussion

5.1 Distributed Spatial Access Methods

We categorize the past work into two groups shown in Figure 5.1: monolithic indexes and hierarchical indexes. Monolithic indexes are structurally similar to local indexes but physically stored in a distributed manner. Hierarchical indexes use a global index to divide the space coarsely into regions and build a local index for each region.

5.1.1 Monolithic Indexes

Monolithic distributed spatial indexes are further divided into three groups: full replication index, central index, and space filling curve index. Full replication indexes replicate the spatial indexes in every server to alleviate query costs. Central indexes depend on distributed storage systems, such as NoSQL databases and distributed file systems, for scalability. Space filling curve indexes linearize multidimensional keys into one dimensional keys and process queries with range queries in one dimensional space.

5.1.1.1 Full Replication Index

Both GPR-Tree [60] and Lai et al. [148] replicate the RTree index in every server such that insertions and deletions are sent to all servers. With the index fully replicated, any server can process queries independently. Compared to GPR-Tree, Lai et al. [148] distributes loads when idle servers are detected. Lai et al. [148] divides the global space with a static partition function and

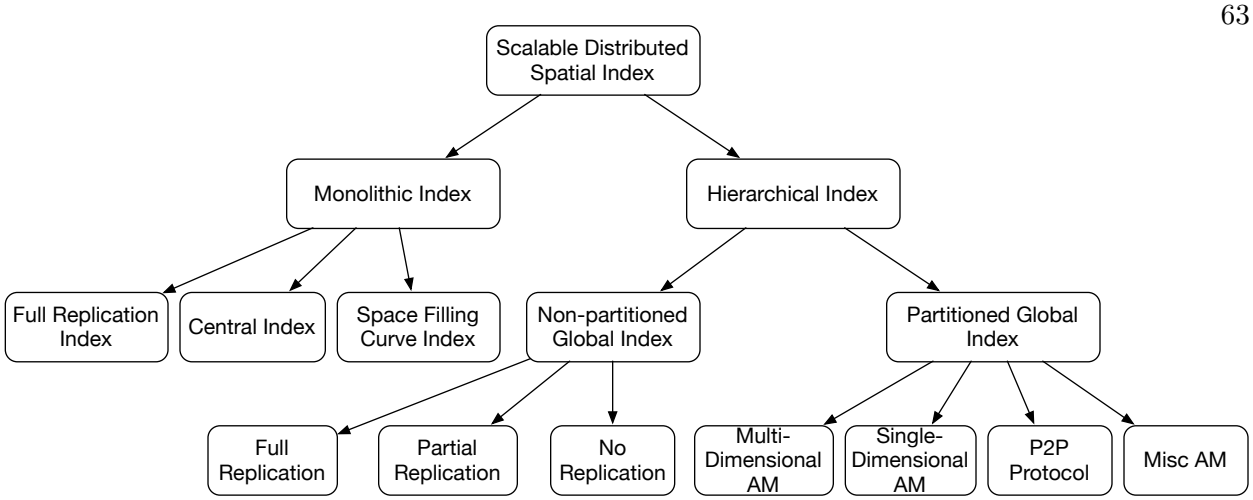


Figure 5.1: Literature Category

assigns each server with one region. A query is forwarded to the server whose assigned region intercepts with the query region and the forwarded server processes the query in its local RTrees. If the number of RTree nodes to be searched exceeds a threshold, part of the potential RTree nodes are forwarded to idle servers for further processing. [148] takes advantage of the fully replicated global index to dynamically distribute loads to achieve higher throughput. SIDI [126] partitions the global space using grid structures and replicate the partition tree in all servers. SIDI divides the global space into a grid of n*n uniform cells and aggregates the cells into zones such that there are more cells in sparse zones based on a sampled dataset. Then, zones are distributed to servers based on each server's capacity. It might scale well by partitioning data space according to density but its dependence on the stable spatial distribution renders SIDI inflexible for data whose spatial distribution frequently changes.

A full replication index works best for query dominant workloads since every server can process queries independently. Dynamic data distribution with frequent hot spot changes would lead to high update costs due to the necessity of broadcasting each update. Furthermore, load distribution technique in [148] utilizes all idle servers for queries with large regions.

5.1.1.2 Central Index

Central indexes are stored in distributed storage systems, such as HDFS or HBase. Traverse [174], Akdogan et al. [12], and Liao et al. [101] store the spatial indexes in HDFS and CCIndex [187] uses HBase. Both Traverse and Liao et al. [101] construct RTrees but Traverse stores each layer of the RTree in one HDFS file while Liao et al. [101] stores the whole RTree in one HDFS file and aligns the tree nodes with the file blocks in HDFS. The difference results from how indexes are constructed. Traverse employs packing algorithms [99] to build RTree from bottom up. On each layer of the RTree, one Mapreduce job is launched to generate the index and output to one HDFS file, which is the input to the next Mapreduce job. The reason for one HDFS file for the whole index in [101] is its configuration of large fan-out where the size of the internal tree nodes is small. Akdogan et al. [12] applies a Voronoi diagram [129] to partition the global space since Voronoi diagrams facilitate nearest neighbor searching. CCIndex builds an inverted table for each searching attribute and chooses one of the inverted tables during query processing, utilizing only one querying criterion. CCIndex chooses the inverted table that incurs minimal covered regions, filtering out as many regions as possible. Strictly, this strategy is not a multidimensional solution and it yields more false positives than multidimensional indexes that include all querying criteria. In terms of query engine design, central indexes launch either a procedural program, such as [101] or a Mapreduce program, such as Traverse and [12], to process queries. For highly selective queries, a procedural query engine is favored since the overhead to start a Mapreduce program might overshadow the benefits of increased parallelism. Paradase [107] builds a PMI index for range queries and a OII index for trajectory queries and stores both indexes in a GFS-style [67] storage. PMI is a partition tree that divides the whole space such that each partition stores data of approximately equal size. OII is an inverted index that tracks the trajectory for each spatial object. Paradase launches Mapreduce jobs for queries and utilizes one of the indexes based on query types.

All of these central indexes assume query dominant workloads and unload the system scalability onto distributed storage systems. The main decision choices are index storage and query

Table 5.1: Monolithic Designs with Space Filling Curves

Systems	Database	Linearization	Index Structure
MD-HBase [128]	HBase	Z-Order	KD-Tree
Fox2013 [59]	Accumulo	Z-Order	N/A
ST-HBase [110]	HBase	Z-Order	KD-Tree
Pyro [100]	HBase	Moore Curve	N/A
D8-Tree [41]	Cassandra	Z-Order	8-ary tree
KR$^+$-Index [164]	Cassandra	Hilbert Curve	R+-Tree
[98]	HBase	GeoHash [1]	N/A
HGrid [73]	HBase	Z-Order	Regular Grid

engine. Queries can be processed either in procedural programs or by distributed query engines, such as Mapreduce and Spark [175]. Index storage depends on how the indexes are built initially and how the query engine retrieves the indexes.

5.1.1.3 Space Filling Curve Index

In this category, papers adopt linearization techniques to map multidimensional data to one dimensional data and store spatial objects in key value stores. Some construct index structures to reduce false positive rates and support more kinds of queries. Table 5.1 lists all the related work.

MD-HBase [128] and ST-HBase [110] adopt KD-Trees with regular decomposition to filter out false positives. MD-HBase persists the KD-Tree in HBase and eliminates false positives by recursively sub-dividing the query regions, which emulates the way the KD-Tree partitions the space. However, MD-HBase does not handle deletions and it requires transactional primitives in HBase. ST-HBase assumes the KD-Tree fits in memory and it designs a Term Cluster Based Inverted Spatial Index(TCbISI) to support term based queries. Fox et al. [59] and Pyro [100] do not create indexes. Fox et al. [59] constructs bloom filters to filter out false positives and it integrates the time attribute in the linearized keys, supporting only spatial temporal queries. Pyro proposes a dynamic programming algorithm called Adaptive Aggregation Algorithm(A3) to generate a query plan that decides between unnecessarily reading more data blocks and issuing more disk seeks. In order to support moving hotspots, Pyro designs a group based replica placement

to preserve data locality when regions are split. D8-Tree [41] focuses on data-thinning queries by employing de-normalization, sacrificing storage space for performance. D8-Tree is a balanced 8-ary tree stored in Cassandra where each tree node contains the top k elements of the relative subregion. The top k elements contained in lower levels are guaranteed to include the elements contained in higher levels. By replicating the elements on different levels, D8-Tree enables users to navigate with different levels of visualized details efficiently. KR^+-Index [164] builds a R+-Tree and creates a mapping table from Hilbert Curve values of fixed precision grids to R+-Tree [146] leaf nodes. It uses R+Tree to partition the spatial database into regions and configure the fan-out to meet the trade-off between false positive and the number of sub-queries. The regions are split if they overflow but the top layer R+-Tree nodes are not stored or maintained. The query engine refers to the persistent mapping table to find the rectangles that match the queries. Lee et al. [98] applies geohash to address the spatial objects in HBase. Specifically, spatial objects are translated into a set of geohashes with the same precision and these geohashes are stored as rowkeys in HBase. The query engine translates the query region into geohash ranges and forwards the range queries to HBase. HGrid [73] combines Z-Order Curve and a regular grid index to linearize the spatial points. It first decompose the global space into tiles with Z-values and each tile is further divided into grids based on regular grid structures. The linearized key stored in HBase is the concatenation of the Z-value and the regular grid row index. The column name is the contatenation of the regular grid column index and the object ID. Similar to [98], HGrid translates query ranges to the key ranges in HBase for query processing. However, HGrid statically configures the granularity of the Z-Order value and regular grid index and it does not handle hotspots like Pyro does.

To summarize, space filling curve based approaches linearize multidimensional data to one dimensional data and uses the linearized key in key value stores. Figure 5.2 shows two kinds of space filling curves in the two-dimensional space: Z-Order Curve and Hilbert Curve. Moore Curve is a variation of Hilbert Curve and GeoHash is a variation of Z-Order Curve. Z-Order Curve has the property of prefix matching if the linearized key is constructed by interleaving the bits in x-axis and y-axis while Hilbert Curve keeps better data locality. Overall, there are three optimizations for space

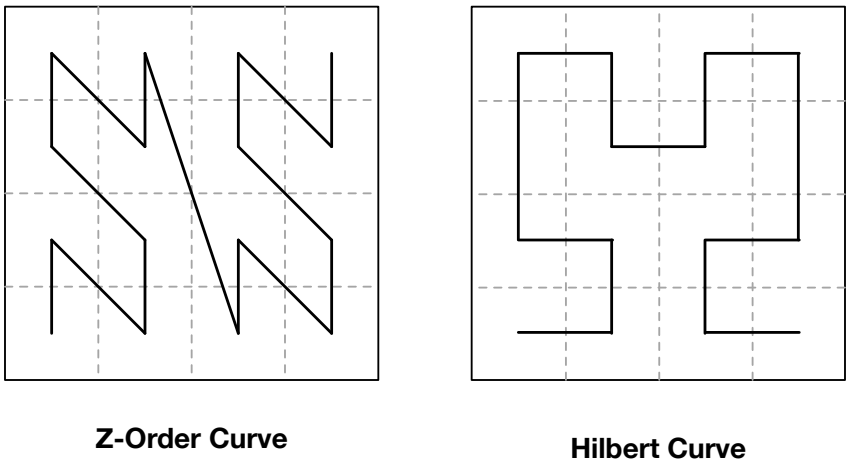


Figure 5.2: Space Filling Curves

filling curve based indexes. First, multidimensional data structures can be constructed to minimize false positive rates. Second, algorithms can be designed in query planning to generate range queries to reduce latency, such as Pyro's A3 algorithm. Third, storage specific optimizations are possible, such as pre-splitting in HBase. MD-HBase, ST-HBase, and Pyro share the common design of storing data in HBase and adopting a pre-split technique. Pre-split technique is to proactively split a region to pay the region split costs when the cluster is lightly loaded. MD-HBase pre-splits regions once the split patterns are learned from historical data. Pyro does pre-splitting to preserve data locality within regions. ST-HBase pre-splits for the inverted index table to increase concurrency when the system boots up.

5.1.2 Hierarchical Indexes

Compared to Monolithic indexes, hierarchical indexes could achieve better scalability because of their data dependent global partitioning. Hierarchical index designers are faced with consistency challenges between global indexes and local indexes when database updates are frequent. We categorize hierarchical indexes based on whether the global indexes are partitioned.

5.1.2.1 Non-partitioned Global Index

Non-partitioned global index papers store the whole global index in a set of servers, called masters. The master servers could be one server, a subset of the cluster, or the whole cluster.

Full Replication In this category, global indexes are replicated in every server but inconsistency might exist among the copies. The benefit of the full replication is that every server is capable of processing and routing the queries. However, the overhead of keeping the global index updated and consistent is high.

HQT* [88] stores spatial points in a linked list called buckets, which are stored locally in servers. A Quad-Tree is built to index the buckets using regular decomposition. Each server keeps a subset of the buckets and a copy of the Quad-Tree that might be outdated. HQT* only updates the local Quad-Tree when queries are forwarded by other servers. When servers are overloaded, an algorithm called Dissection Splitting divides the buckets into two sets and minimizes the number of buckets to be moved. BR-Tree [80] builds RTrees in each server with bloom filters and then the MBRs of the root nodes are published to all servers. The global index is organized as an array of MBRs with corresponding bloom filters. To reduce the global index update costs, BR-Tree periodically broadcast update messages when the local changes reach a predefined threshold. HQT* and BR-Tree differ in the index construction where hQT* incrementally inserts spatial points into buckets and the global index is expanding as the buckets are split while BR-Tree builds the local index in batch and constructs the global index afterwards. SHAHED [53] targets at a different application where metric data are collected periodically from sensors that are uniformly distributed around the globe. Since the sensors have static locations, SHAHED partitions the global space into a uniform grid and builds a stock Quad-Tree which is shared and used as a template. The collected sensor data are sorted and stored based on their Z-Order value and a lookup table that maps the Z-Order value to the position in the sorted list is stored in each server. Overall, each server keeps a copy of the global grid index, stock Quad-Tree, and the lookup table, all of which the query engine depends on.

Partial Replication Alternatively, global indexes are replicated in a configured number of servers. The cost of keeping global indexes consistent is relatively low since they are stored in fewer servers than full replication approaches. However, query requests being forwarded to servers with global indexes leads to load imbalance.

ATree [134], EEMINC [181], and Nam et al. [119] designate a set of master servers to store global indexes. ATree builds RTrees and bloom filters in each server and then selects and distributes RTree nodes to be stored in an array in master servers. Similarly, EEMINC builds KD-Trees in local servers and then selects KD-Tree nodes that are organized as a RTree in master servers. Both ATree and EEMINC design a cost-based algorithm to select the local indexes to be published as global indexes and both determine whether to publish a parent node or its children nodes by calculating the volume difference. Compared to ATree, EEMINC further accounts for the time cost of updating the global index structures in its cost model. Nam et al. [119] briefly introduces a replication protocol for the global index. It does not enforce strong consistency among the replicated global indexes. For example, replica A might process update requests X and Y in a different order from replica B. Concurrent search requests could potentially obtain different results if different replicas are referenced.

The following papers share two common designs. First, global indexes are stored in distributed storage systems, such as HDFS and HBase. Second, global index construction happens before local indexes. ABR-Tree [185] stores the global index in HBase to handle replications so that any server can read a copy of the global index from HBase and route the queries. However, the global index is a BTree on an ascending attribute, such as time. The dependence on the existence of an ascending attribute causes ABR-Tree to work only for append-only spatial temporal databases. EDMI [186] partitions the space into subspaces by building an adaptive KD-Tree on a sampled dataset and launches a Mapreduce program to build a Z-Order Prefix RTree(ZPR-Tree) for each subspace. Both the KD-Tree and ZPR-Trees are stored in HBase and data locality are preserved since spatial objects are sorted by their Z-order values, similar to STR packing [99].

ScalaGiST [106] presents a scalable generalized search tree, inspired by classical Generalized

Search Tree(GiST) [78]. ScalaGiST samples a data set and partitions the samples into several subspaces, after which a Mapreduce job is launched to construct a local RTree for each subspace. Then, the master server reads the root nodes of all RTrees and builds a global RTree using the root nodes as its leaf nodes. Both the global RTree and local RTrees are persisted in HDFS. Whitman et al. [169] builds the global index by constructing a PMR-QuadTree on a sampled dataset and stores the partition list in the Hive metastore. Then, a Mapreduce job is launched to build local PMR-QuadTrees in all partitions and these trees are stored in map files in HDFS. Hadoop-GIS [11] builds a grid based global index that recursively splits a tile if it contains too much data. The global index is stored in HDFS and the local indexes are built on demand in memory. It integrates a Mapreduce based query engine and a procedural program query engine to process highly selective and computing intensive queries respectively. Aji et al. [10] extends Hadoop-GIS to support nearest neighbor search and spatial joins. Ma et al. [108] splits the global space with a Region Split Tree, a combination of grid index and tree index. Region Split Tree splits overwhelmed regions into grids and stores the addresses of the new grids in the index. The Region Split Tree and local indexes are both stored in HBase. Similar to EDMI and ScalaGiST, Ma et al. [108] relies on the Mapreduce system to build local indexes when the data in regions become stable.

To sum up, many papers store the global indexes in distributed storage systems such that all servers can read them and forward queries. However, EDMI, Whitman et al. [169], and Hadoop-GIS assume the databases to be static and the global indexes are not changed over time. If the databases are dynamically changed, the consistency costs are comparable to full replication approaches since every server is storing an in-memory copy of the global indexes. ABR-Tree, ScalaGiST, and Ma et al. [108] handle better for dynamic databases. ABR-Tree assumes an append-only spatial temporal database. ScalaGiST keeps the new data in a separate tree and periodically merge the tree to the original index. Ma et al. [108] splits grids dynamically for overwhelmed regions. Second, a sampling method is introduced when the global space is partitioned, which works best for static databases. Examples are EDMI, ScalaGiST, and [169]. Third, Mapreduce programs help to parallelize the local index building. Lastly, ATree and EEMINC proposes a cost model to select local indexes to be

published as global indexes and guarantees that the local indexes selected from a local server cover the whole area of that server. This cost model method reduces false positive rates by providing more detailed MBR information while keeping the publishing costs small.

No Replication When the global indexes are not replicated, query processing tend to be costly but updates to the global indexes are more cost effective. The global indexes are stored in one designated master server.

MR-Tree [90] sorts spatial objects on their Hilbert values of the centers and assigns the objects to servers in a round-robin manner, declustering objects that are close in space. After the local objects are organized as chunks similar to RTree leaf nodes, the MBRs are sent to the master server where a top level RTree is built. Instead of addressing the bottleneck in the master server, MR-Tree focuses on selecting the optimal chunk size since a small chunk size results in activating several servers even for small queries and a huge chunk size limits parallelism. UQE-Index [109] takes the time attribute to build a B+-Tree as the global index. For the data in the current time interval, UQE-Index uses either QuadTree or KDTree to divide the spatial objects into several subspaces, which are stored as regions in HBase. When the current time interval ends, UQE-Index builds a RTree for each subspace. Since the spatial objects in each subspace during the current interval are not indexed, UQE-Index supports high update rates. SpatialHadoop [52] builds a RTree on a sampled dataset and generates a partition list based on the RTree boundaries. Then, it launches a Mapreduce job to sort the original data according to the partition list and constructs a local RTree for each partition. Finally, the master server builds an in-memory global index on top of the local RTrees. SpatialHadoop resembles ScalaGiST except that SpatialHadoop stores the global index in a master server. Both MR-Tree and SpatialHadoop optimize query performance and dynamic databases are not well supported. UQE-Index is a append-only spatial temporal index, like ABR-Tree.

5.1.2.2 Partitioned Global Index

Partitioned global index papers divide the global index structures into small pieces and distribute them to servers such that client requests are routed to the destination server where the expected local indexes reside. The critical problems to be addressed are local index selection for publishing, query routing, and load balance. The local index selection algorithm affects query performance and global index update costs since the more the local indexes are published, the lower the query false positive rates are and the higher the index update overhead is. Query routing algorithm directly determines the average number of query forwarding. It consists of two subproblems: what to maintain in the routing table and how to manage the global indexes. Load balance affects the system scalability and highly skewed resource usage could severely degrade system throughput.

Single Dimensional AMs In this category, papers adopt single dimensional AMs as the partitioned global index. To support multidimensional queries, a dimension reduction method is necessary, such as space filling curves. Another important problem is load balance since the linearized space is not filled uniformly.

QT-Chord [47] builds a MX-CIF QuadTree in each local server and publishes the local index nodes as global indexes in an overlay network, called Chord. QT-Chord assigns a code value to each index node based on the path from the root node to the index node such that the codes for parent nodes are prefixes of the codes of their children. To reduce the index maintenance cost, QT-Chord select a subset of the local index nodes by a mapping function [153] that maps the index node codes to DHT keys. The mapping function reduces the global index size by 75%. Similar to QT-Chord, Squid [144] also deploys the Chord network but its dimension reduction technique is the Hilbert Curves. Squid generates Hilbert Curve values for all spatial points and assigns the points to servers in the Chord network ring without hashing the Hilbert Curve values. The purpose of not hashing is to preserve data locality and speed up query pruning but it necessitates a load balancing step. DHR-Trees [166] builds a Hilbert RTree on spatial objects and employs P-Tree [38] to organize the servers by their smallest Hilbert Curve values. Each server stores the left-most root-to-leaf path of

Table 5.2: Hierarchical Designs with Partitioned Global Index

Systems	Global Index	Index Selection	Routing	Load Balance
QT-Chord [47]	Chord	Mapping function in QuadTree	P2P	Consistent hashing
Squid [144]	Chord	Hilbert Curve map	P2P	locality preserving hashing
DHR-Trees [166]	P-Tree	Hilbert Curve map	P2P	N/A
ZNet [147]	SkipGraph	Z-Order Curve map	P2P	offload to neighbors
SkipTree [14]	SkipNet	Whole Region MBR	P2P	leave-rejoin
SkipIndex [178]	SkipGraph	Whole Region MBR	P2P	offload to neighbors
Skip-Webs [18]	Skip-Webs	Whole Region MBR	P2P	N/A
RT-CAN [162]	C^2 [31]	Cost Model	P2P	N/A
k-RP*s [102]	KD-Tree	Whole Region MBR	Tree traversal	N/A
DiST [120]	KD-Tree	Whole Region MBR	Tree traversal	N/A
MIDAS [158]	KD-Tree	Whole Region MBR	Tree traversal	Virtual server balance
RAQ [124]	Partition Tree	Whole Region MBR	Tree Traversal	N/A
SD-RTree [49]	RTree	Whole Region MBR	Balanced Tree traversal with overlapping coverage	N/A
VBI-Tree [84]	RTree	Whole Region MBR	Balanced Tree traversal	N/A
P2PR-Tree [116]	RTree	Whole Region MBR	Tree Traversal	N/A

its independent Hilbert R-Tree as the routing information. ZNet [147] applies the Z-Order Curve to map multidimensional objects to single dimensional points and then relies on SkipGraph [19] for routing. Compared to Squid and DHR-Trees where the global space is statically partitioned since the partitioning level needs to be decided beforehand, ZNet partitions the space dynamically by covering continuous zones at the same partition level in each server. Moreover, each ZNet server estimates the average loads of its neighbors by sampling to periodically balance loads with its neighbors. SkipTree [14] and SkipIndex [178] uses a partition tree to decompose the global space where each leaf corresponds to a region. Each server is assigned with a region and all servers are organized in the SkipNet [76] or SkipGraph. The linearization method in SkipTree and SkipIndex

is to order the leaf regions in the partition tree such that servers know how to forward queries. However, SkipTree and SkipIndex differ in load balancing techniques where SkipTree removes the overloaded server before adding it again and SkipIndex migrate data to neighboring servers. Skip-Webs [18] organizes the global space with quadtree/octree in multiple layers, similar to SkipGraph and SkipNet. By referencing the pointers in different layers, the number of routing steps is bounded by O(logN). Moreover, Skip-webs provides the definition for range-determined link structures and the proof of a set-halving lemma on QuadTrees to support the theoretical bound for routing steps. With the QuadTree structure, Skip-Webs orders the servers and route queries accordingly.

All the papers rely on the P2P overlay network and enforce a total order among the servers. Although the underlying P2P systems guarantee O(logN) bound for routing, query costs are not necessarily bounded. Since Chord does not support range queries, QT-Chord yields higher costs for multidimensional range queries. However, Squid applies the Hilbert Curve values directly on the Chord ring to support range queries, reducing query costs. The approaches that do not guarantee uniform distribution require load balance and they either removes a server and adds it back or offloads to neighbor servers. QT-Chord, based on consistent hashing [86], does not require load balancing.

Multi-Dimensional AMs When the partitioned global index supports multidimensional queries intrinsically, there is no need for dimension reductions. The main challenges are balancing loads and reducing routing costs.

RT-CAN [162] adopts the C^2 [31] overlay, which extends CAN network and supports multi-dimensional queries intrinsically. Besides a mapping function to map R-Tree nodes to CAN nodes, RT-CAN proposes a cost model based on statistics to select local R-Tree nodes to publish. The index selection mechanism aims to balance index maintenance and query costs such that the overall performance for the current workloads is improved. When new server joins, RT-CAN splits the server with the largest volume to balance the loads. k-RP*s [102] stores spatial points in buckets and builds a paged KD-Tree to index the buckets. Each server stores a bucket and pages in the KD-Tree are distributed across the servers. Each split of bucket creates a new page in the KD-Tree

and KD-Tree pages are split in a way to equalize the probability that either page splits again. K-RP*s keeps a local client image of the global index, which might be outdated and are updated during the routing. The query engine traverses the global KD-Tree to find the target spatial points. However, since the KD-Tree is not kept balanced, the number of average network hops are O(N) in the worst case where N is the number of servers. DiST [120] adopts KD-Tree as the global index and stores partial global index in each server. The partial global indexes are updated when queries are further forwarded by the server that the local server forwards to. Although the index updates are piggybacked to reduce network traffic, the update costs become higher for larger clusters. DiST is targeted at workloads where range query performance is more important than scalability to large clusters. Similar to k-RP*s, DiST does not keep the KD-Tree balanced and a skewed global tree leads to the worst query performance. MIDAS [158] manages the space with a KD-Tree and each leaf node is a virtual peer that stores the spatial points in its rectangle. A physical server can be responsible for several virtual peers and each virtual peer stores the path from the root to its leaf node and a routing table of outer subtrees[1] . MIDAS balances loads by migrating virtual peers among physical servers but it does not guarantee the global tree balance. It merely proves that the expected depth of the distributed KD-Tree when n servers join on an initially empty overlay is O(logN). RAQ [124] designs a partition tree, similar to SkipTree and SkipIndex, and assigns each leaf node to a physical server. Each server contains a link to any of the servers in the sibling substrees on each layer of the partition tree. During query processing, RAQ forwards requests based on the links within O(logN) steps where N is the number of the servers. However, if the partition tree becomes skewed, the routing costs can be O(N) in the worst case. SD-RTree [49] organizes the whole space in a RTree and creates a routing table that assembles a binary RTree. To reduce the network hops for queries, SD-RTree inserts overlapping coverage information that records the overlapping regions between the current nodes and their outer subtrees. The overlapping region maintenance costs are negligible only if the RTree embedding space is fully covered. SD-RTree also maintains a balanced RTree by rotating unbalanced subtrees to bound the routing steps. SD-

[1] Outer subtrees of tree node A are the trees rooted at the sibling nodes of A or the ancestors of A.

RTreeII [51] optimizes SD-RTree by introducing a more flexible allocation protocol such that node splits can be dynamically accommodated when physical resources are scarce. VBI-Tree [84] designs a binary tree structure as the global index, supporting RTree, MTree [34], etc. Similar to SD-RTree, VBI-Tree maintains tree balance by rotation. Like BATON [83], each routing node stores pointers to a selected set of nodes on the same layer of the global tree and all the node information from the root to the current leaf node. P2PR-Tree [116] performs the first two layers of partitioning to statically divide the global space into blocks and groups. Each group is managed with the a RTree and each leaf node in the RTree corresponds to a server. Every server keeps track of the node information from the root to its leaf node and gradually collect routing information as more queries are forwarded and processed. However, P2PR-Tree does not keep RTrees balanced and it does not support queries across groups.

Overall, systems with partitioned multidimensional global indexes have to deal with load balancing and routing tasks. There are two approaches for load balance. One method migrates data from overloaded servers to their neighbors, such as VBI-Tree and SD-RTreeII. Another method is to balance the virtual servers among physical servers, such as MIDAS. Query routing costs depend on what to maintain in the routing tables and how to manage the global index. SD-RTree stores the overlapping coverage to reduce query cost and MIDAS2012 stores all the outer sibling information and all the node information on the path from the root node to its leaf node. And, VBI-Tree2006 stores pointers to some nodes on the same layer of the global tree and all the parent information to the root node. For structures like RTree where regions of nodes overlap with others, the overlapping coverage information in SD-RTree avoids unnecessary traversals, reducing network traffic. On the other hand, global tree balance determines query costs. SD-RTree and VBI-Tree maintains global tree balance to guarantee O(logN) query costs while k-RP*s, DiST, MIDAS, etc. does not. Furthermore, keeping a global index image on the client side minimizes the routing costs even though the image might be outdated. Examples are k-RP*s, DiST, and SD-RTree.

P2P protocols CAN [137], MAAN [30], and Mercury [25] are P2P protocols that support multidimensional queries intrinsically. CAN builds an overlay network by hashing keys to points in

the multidimensional space where the servers occupy. Each CAN server stores its neighbors in the multidimensional space and routes requests to its neighbors. A new server is assigned to a random point and the old server containing this point splits its zone in half, retaining half and handling the other half to the new server. The virtual coordinate space resembles an adaptive KD-Tree with regular decomposition but it stores neighbor links instead of parents and children links. MAAN and Mercury adopt the Chord-like network to organize the spatial databases on each dimension and both apply a locality preserving hashing to support range queries. MAAN adopts a more strict hashing algorithm called uniform locality preserving hashing to maintain load balance but uniform locality preserving hashing is only possible when the distribution function of the spatial points is continuous and monotonically increasing, and is known in advance. Mercury handles load balance by monitoring the loads to select the lightly loaded server and uses leave-rejoin protocol. Moreover, both MAAN and Mercury handle multidimensional queries by processing single dimensional range queries except that MAAN intersect results from range queries on all the dimensions in the query while Mercury collects and filters on the results from one single dimensional range query. By maintaining approximate histograms of data distribution, Mercury is able to find the most selective dimension in the query to minimize the number of servers to be contacted. The tradeoff is made by QT-Chord, MAAN, and Mercury to favor range query support in sacrifice of uniform distribution.

Miscellaneous AMs Wu et al. [170] and Ganesan et al. [62] applies a few index structures instead of a specific one. Wu et al. [170] depends on the overlay network to support range queries, such as CAN, BATON, and P-Ring [39]. Similar to RT-CAN, the mapping function from the global indexes to servers is implemented in the overlay network. However, the local index selection lacks details on the random walk model and bayesian network model for update activity prediction. Ganesan et al. [62] introduces two data structures SCRAP and MURK. SCRAP maps multi-dimensional data to the single dimension using Z-Order Curve or Hilbert-Curve and then deploys SkipGraph for routing in the one dimensional space. To balance loads, SCRAP adjusts the partition boundary between two servers managing neighboring ranges. MURK partitions the global space using a KD-Tree, similar to CAN, and adds skip pointers to reduce routing costs.

As mentioned earlier, three problems are addressed in partitioned global index designs: Local index selection, query routing, and load balance. Static local index selection chooses the MBR of the whole region in the local server and dynamic local index selection applies a cost-model to dynamically balance index update cost and query filtering effectiveness. P2P techniques are heavily utilized in query routing since it supports decentralized and highly dynamic environment. When range queries are supported by the adopted P2P system, papers either replaces consistent hashing with locality preserving hashing or simply modify the query engine to support data enumeration, like QT-Chord. For papers utilizing multidimensional data structures instead of P2P system, query routing module needs to decide what to maintain in the routing table and how to manage the global index. Routing tables commonly include all the node information that is on the path from the root to the leaf node. Particularly, for data structures where regions of nodes might overlap, keeping the overlapping coverage avoids unnecessary tree traversals. Global index management, such as tree balance is critical for query performance. Keeping the global tree balanced guarantees O(logN) forwarding steps by paying the balancing costs, which is commonly solved by the rotation technique. Lastly, load balance is necessary to scale out to large clusters when the data is in non-uniform distribution. Three approaches are presented: offloading to neighbor servers, leave-rejoin protocol, and migrating virtual servers among physical servers.

5.1.3 Discussion

Figure 5.3 lists all the problems and proposed solutions have been detailed in the last section. First, **Monolithic indexes vs. Hierarchical indexes**. Monolithic indexes handle query dominant workloads well since every server has access to the indexes to process requests independently. Loads are relatively balanced as long as requests are routed to servers in a balanced manner. However, if the query selectivity varies a lot, which means the standard deviation of query resource consumption is high, even distribution of queries does not suffice for load balance. Lai et al. [148] proposes a parallel search algorithm to migrate queries to idle servers. Moreover, there are still some monolithic indexes, especially space filling curve indexes, that handle database updates, such

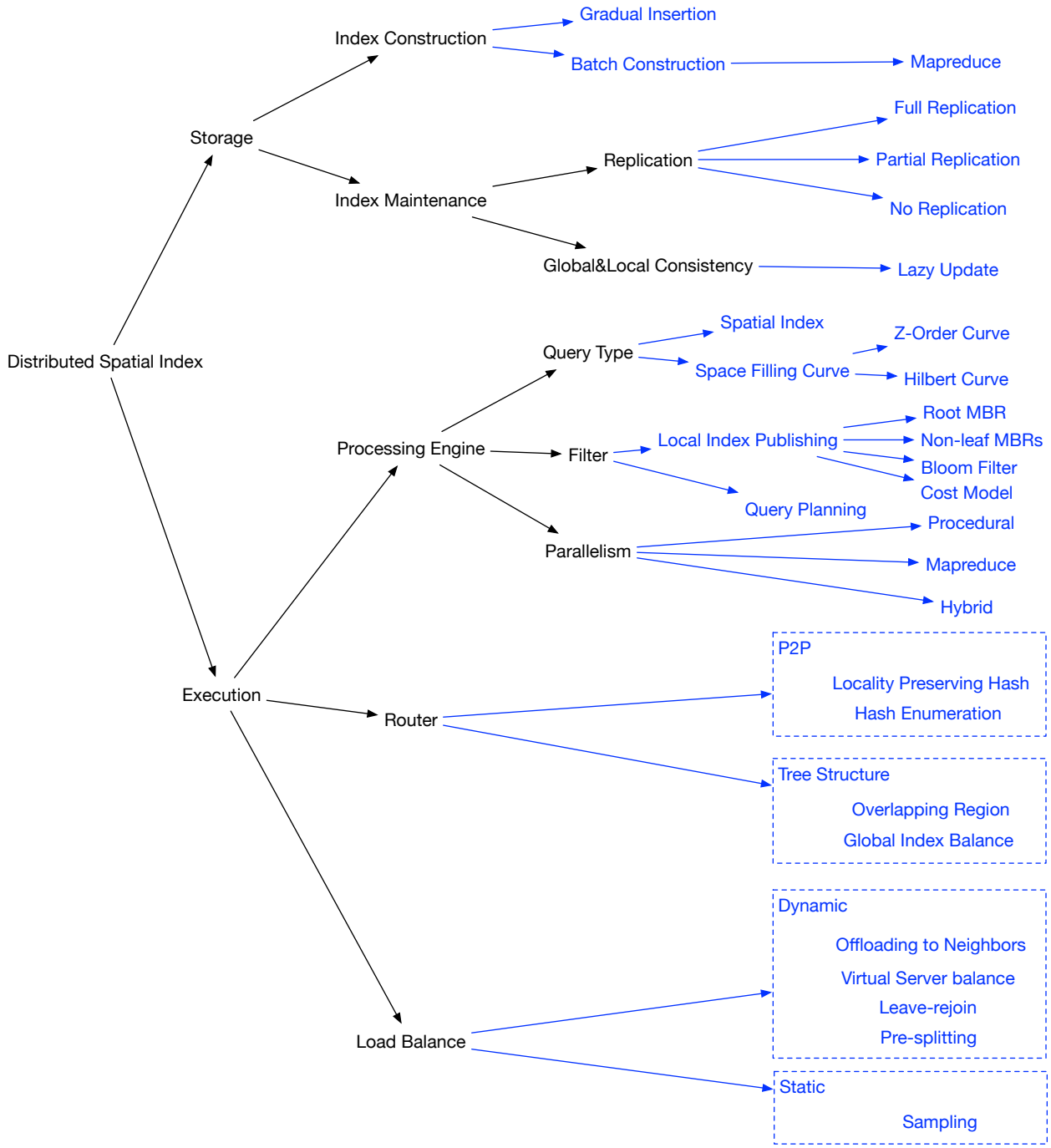


Figure 5.3: Problems: Black represent the problems and blue represent solutions.

as Pyro and MD-HBase. Hierarchical indexes limit database update costs within local indexes to increase parallelism but the servers storing global indexes could potentially become the bottleneck.

Second, **Non-partitioned hierarchical indexes vs. Partitioned hierarchical indexes**. Non-partitioned hierarchical indexes usually require fewer forwarding steps for request processing but the master servers storing global indexes could be a bottleneck. By choosing different replication factors, non-partitioned hierarchical indexes reaches a tradeoff between the update costs and read parallelism of the global indexes. Partitioned hierarchical indexes is a fully decentralized system and supports dynamic cluster environment but the routing costs are O(logN) at best where N is the number of servers. Furthermore, global index update costs could be considerable since partitioned hierarchical indexes usually store multiple servers' information in their routing tables. In the following, we will discuss two most debated design choices.

5.1.3.1 Local Index Selection

Local index selection is the process of selecting local indexes to be published as global indexes. Local index selection is specific to hierarchical indexes and it helps improve data filtering, thus fewer false positives. The more local indexes selected, the lower false positive rate is and the higher update costs are. When more local indexes are selected, the global index is able to filter out more servers but there is higher chance that local index updates propagates to global indexes. Moreover, local index selection should conform to the completeness and uniqueness properties, defined in RT-CAN [162]. The completeness property means that all spatial objects should intercept with at least one of the selected index nodes. The uniqueness property means that from all the nodes selected from one server, either a node can be selected only if all of its ancestors are not selected. The completeness property guarantees correctness and the unique property avoids unnecessary selections. The selection can be either static or dynamic. Static selection chooses one of the following three: the root node, all non-leaf nodes, or a bloom filter and dynamic selection is commonly a cost based model. The selection method decides the tradeoff between low false positive rates and low resource consumption.

Static selection. The root node selection and all non-leaf nodes selection are two extreme cases and favors lower update costs and lower false positives respectively. For example, VBI-

Tree [84] and MIDAS [158] adopt the root node selection and MR-Tree [90] selects all the leaf nodes. Bloom filters reduces false positives by taking some extra space to store hashing values of points. It merely works for exact match queries because of the hashing matching. BR-Tree [80], ATree [134], and Fox et al. [59] appends the bloom filters to the selected index nodes and BR-Tree proposes a count based bloom filters to support deletions. Specially, QT-Chord [47] selects the local indexes by a naming function that removes the grid index ending bits if the last two bits are the same.

Dynamic selection. The cost based model relies on a benefit comparison function to decide whether to choose a parent node or its children. Papers differ in the comparison function design. On the one hand, ATree calculates the difference of the parent node MBR and the aggregate MBR of all its children and decides to select the children when the MBR difference exceeds a threshold. Otherwise, the parent node is selected by default. This structural cost model is cost effective and often yields a more balanced selection plan than the static selection. On the other hand, the comparison function can also be designed based on query processing cost and index maintenance cost. RT-CAN collects the routing costs and index update costs from the history queries to generate the comparison function and Wu et al. [170] uses random walk model and the bayesian network model to predict update activities.

5.1.3.2 Load Balance

Similar to local index selection, load balance can be handled either statically or dynamically. The static method partitions a sampled dataset and adopts the partitioning plan for the whole database. SpatialHadoop [52] and Whitman et al. [169] store the partitioning plans in HDFS and build the local indexes in parallel for each partition. The dynamic method migrates data from heavily loaded servers to lightly loaded ones periodically and it relies on a monitoring module to collect load information. The first dynamic method is “offloading to neighbors” where each server collects load information from neighbors and balance loads only with neighbors, such as Squid [144]. The second dynamic method is “virtual server balance” where each server houses

multiple virtual servers and one or more of the virtual servers are moved from overloaded physical servers to lightly loaded ones, such as MIDAS [158]. The third dynamic method is "leave-rejoin" based on P2P systems where lightly loaded servers are chosen to leave and rejoin the P2P network, such as Mercury [25]. Since the joining algorithm chooses the heavily loaded servers to split its data between itself and the new server, the leave-rejoin technique balances loads. The last dynamic method is "pre-splitting" where regions are split when they tend to become congested in the near future. This technique is popularly adopted in HBase systems such as MD-HBase [128] and ST-HBase [110]. However, the algorithm of choosing the region for pre-splitting has not been fully examined and requires further exploration.

5.1.3.3 Other considerations

Index Construction If the index is built when database is empty, the index will be gradually updated as spatial objects are inserted, such as SD-RTree [49]. If the index is built when there are initial data in the database, there is a need for an algorithm to construct the indexes in batch. To parallelize the construction process, the first approach is the sampled method where a preliminary partitioning plan is generated based on the sampled data. Then, local indexes are built in parallel for each subspace of the partitioning plan and a final global index is created on top of the local index MBRs. Examples are EDMI, [169], ScalaGiST, and SpatialHadoop. The second approach recursively divides the global space if the subspace exceeds a threshold. For example, Ma et al. [108] splits the overwhelmed grids recursively. The last category applies for append-only spatial temporal indexes, such as ABR-Tree, where the top level index is a one dimensional access method on the time attribute.

Space filling curve Linearization technique has been heavily explored in space filling curve indexes and partitioned hierarchical indexes with single dimensional global indexes. Both two indexes replies on either Z-Order curve or Hilbert Curve to transform a multidimensional data to a single dimensional data. The dimension reduction enables researchers to utilize current single dimensional systems, such as distributed key value stores and P2P systems, for query supports and

system scalability. The critical problem for dimension reduction is how to improve query processing such that false positive rate is low. Pyro [100] proposes the adaptive aggregation algorithm to balance the storage seek time and false positives. MD-HBase [128] eliminates false positives by recursively sub-dividing the query regions. When the single dimensional systems do not support range queries intrinsically, such as Chord, a locality preserving hashing is adopted. Examples of locality preserving hashing are Squid [144] and MAAN [30] where MAAN uses a more strict hashing algorithm called uniform locality preserving hashing.

Global & Local consistency The local index updates are usually lazily updated as the global index since distributed transactions introduces high latency and resource usage. Non-partitioned hierarchical indexes update global indexes when the local index change exceeds a threshold, such as ATree. Partitioned hierarchical indexes update global indexes when requests are further forwarded, such as DiST [120]. Even if the local index change does not affect global indexes, it might still be beneficial to update the global index after it reselects the local indexes, such as EEMINC. Furthermore, lazy update technique is also applied in keeping replications consistency for non-partitioned hierarchical indexes. hQT* [88] updates global indexes when requests are forwarded and BR-Tree updates global indexes when the local change exceeds a threshold.

Parallelism The processing engine processes requests by traversing the spatial indexes and possibly forwarding to potential servers. This procedural approach yields low latency for highly selective queries. One example is Whitman et al. [169]. However, less selective queries might return so many results that the parallelism benefits overshadows the overhead of launching the parallel programs or the communication costs. Traverse,Akdogan et al. [12], and SpatialHadoop launch Mapreduce programs to parallelize query processing. Lai et al. [148] distributes queries when the local processing engine returns many potential subtrees and there is idle servers. A hybrid query engine with both procedural and parallel processing, such as Hadoop-GIS [11], can process both queries efficiently but there is no paper that addresses how to choose the two engines.

Table 5.3: Performance

Systems	Design	Cloud Size	Throughput	Latency
[101]	Monolithic Index	9 servers	N/A	40ms for %0.01 selectivity range query
MD-HBase	Monolithic Index	8 servers	125 kops/sec for insert throughput	800ms for 0.01% selectivity range queries
ST-HBase	Monolithic Index	16 servers	6k ops/sec for insert throughput	45ms for range queries
KR$^+$-Tree	Monolithic Index	10 servers	310 kops/sec for insert throughput	N/A
Pyro	Monolithic Index	34 servers	N/A	119ms for 100m*100m range queries
ScalaGiST	Hierarchical Non-partitioned Global Index	30 servers	N/A	100ms for %0.4 selectivity range query
BR-Tree	Hierarchical Non-partitioned Global Index	30 servers	2 kops/sec range queries	60ms for range queries
EDMI	Hierarchical Non-partitioned Global Index	8 servers	N/A	200ms for 0.001% selectivity range queries
[108]	Hierarchical Non-partitioned Global Index	8 servers	5.2 kops/sec for insert throughput	200ms for 0.0001% selectivity range queries
RT-CAN	Hierarchical Partitioned Global Index	32 servers	3.5 kops/sec for 0.01% selectivity range queries; 7.5 kops/sec for 20% insert queries	N/A
QT-Chord	Hierarchical Partitioned Global Index	32 servers	4.5 kops/sec for range queries	N/A

5.1.4 Performance Analysis

Table 5.3 lists the performance of a subset of the reviewed systems, which conduct experiments on non-simulated environments. All systems except MD-HBase and KR$^+$-Index achieve less than 10 kops/sec throughput. MD-HBase and KR$^+$-Index share the same design choice that the non-leaf nodes in KD-Tree or R$^+$-Tree are not maintained. Leaf nodes are directly stored and split if necessary while a grid index is applied for query processing. Workloads with large numbers of insertions result in more indirection times due to the split process, thus eventually worse query performance. On the other hand, query latencies range from 40ms to 800ms. Liao et al. [101] yields 40ms for query processing since it configures a large tree fan-out and stores the non-leaf nodes in memory. The query engine of [101] processes the range queries in memory and only needs to retrieve the leaf nodes in HDFS. To sum up, past distributed spatial access methods reach less than 10k ops/sec throughput and more than 40ms latencies. MD-HBase and KR$^+$-Index yield higher throughput at the cost of lower query performance. The last but not the least is that none of the access methods support transactional accesses.

5.2 Strength and weakness of DDSBrick

The previous section covers the past works on distributed spatial access methods. DDSBrick simplifies the design of scalable and transactional spatial indexes and offers high performance in case of dynamic workloads and frequent hot spot changes. It achieves scalability by adapting the partitioning and replication strategy to the current workloads. Nonetheless, some questions remain: why would DDSBrick based spatial indexes outperform traditional distributed spatial access methods? Does DDSBrick offer better programming interfaces than other paradigms like Tango? Are there limitations in DDSBrick and how could they possibly be overcome?

In this section, we discuss the these remaining questions respectively. First, we compare DDSBrick with DSAMs and show that DDSBrick handles replication and partitioning in finer granularity to achieve better performance. Second, we compare DDSBrick with Tango in great

depth and show that DDSBrick is able to express wider ranges of data structures than those implemented in the thesis. The dynamic partitioning strategy boosts performance for tree-like structures, as all spatial indexes are. Third, we list several limitations in DDSBrick and proposes potential solutions and directions to explore in the future.

5.2.1 DSAMs vs. DDSBrick

The previous section reviews all the distributed spatial access methods and categorize DSAMs into monolithic indexes and hierarchical indexes. The following is the comparison between DDSBrick and DSAMs.

Monolithic Indexes vs. DDSBrick Monolithic indexes manage index structures logically as a whole and persist them in distributed storage systems. The query engine is separated from the persistence layer such that queries can be processed in any server. DDSBrick also stores indexes in a distributed storage system but the query engine is coalesced with the storage system. There are two major differences between them: storage mapping and service distribution. Monolithic indexes map indexes to files or keys in a coarse-grained manner. For example, Traverse [174] stores each layer of a RTree in one HDFS file. DDSBrick maps each tree node to a key in DTranx. Fine-grained mapping enables flexible replication and partitioning configuration such that the storage systems are well tuned for dynamic workloads. Specially, space filling curve indexes map each spatial object to a key based on the linearization curves and these indexes with fine-grained mapping, such as Pyro, handle workload hot spots. However, space filling curve indexes do not handle databases with frequent updates well. On the other hand, monolithic indexes and DDSBrick distribute requests differently. Monolithic indexes simply route requests evenly across the servers while DDSBrick optimizes load balance by employing a workload-aware data partitioning strategy before queries are routed to servers that store the accessed data locally. Similarly, the request distribution strategy in monolithic indexes results in high network costs for update frequent workloads.

Hierarchical Indexes vs. DDSBrick Hierarchical indexes partition the space on two layers such that the updates in the bottom level index do not necessarily propagate to the upper level

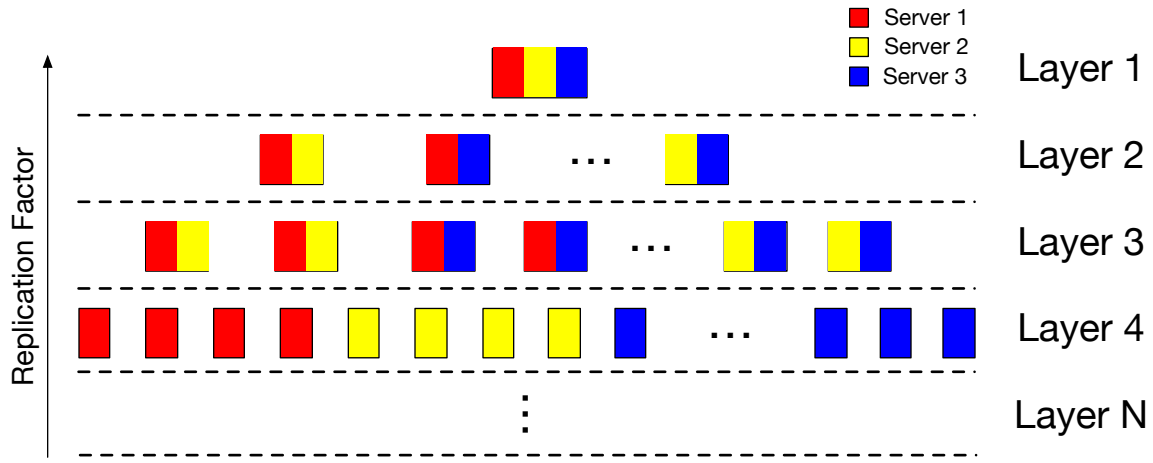


Figure 5.4: Hierarchical Indexes: objects with multiple color blocks are replicated in each corresponding server.

index. Compared to monolithic indexes, hierarchical indexes minimizes update costs and enables flexible load balance. DDSBrick can be viewed as a hierarchical index where each layer of the index tree is a layer in the hierarchy and DDSBrick can adjust the number of layers by configuring the tree fan-out. The design of more layers in the hierarchy is a double-edged sword. On the positive side, more layers allow for more flexible replication and partitioning configuration. On the negative side, it requires much of the system resource to manage the fine-grained replication and partitioning. DDSBrick proposes a dynamic partitioning strategy that integrates a graph partitioning algorithm called METIS to generate a key to server mapping with dynamic replication factors. This dynamic partitioning strategy achieves balanced loads for all kinds of workloads and the database repartitioning is a low latency process. Figure 5.4 shows a typical DDSBrick index. The top layer indexes are more likely to be replicated than those on the bottom layers. The higher replication factor on top layers help to distribute query requests and the lower replication factor on bottom layers conduces to lower update costs.

Overall, both monolithic indexes and DDSBrick persist indexes in distribute storage systems but they differ in storage mapping and service distribution. DDSBrick is able to dynamically adjust replication and partitioning strategy for changing hot spots and frequently updated databases.

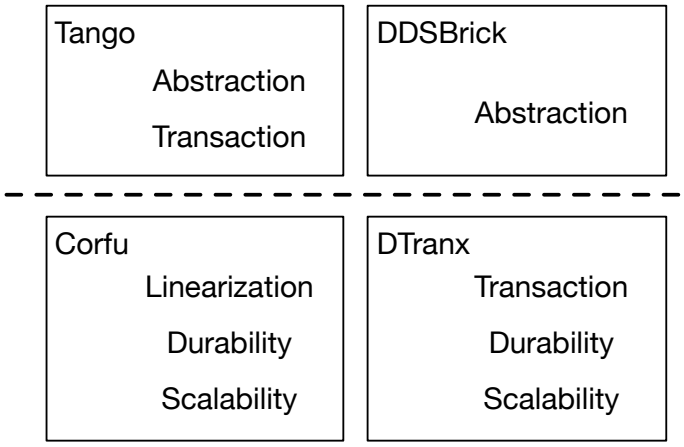


Figure 5.5: Tango vs. DDSBrick

Hierarchical indexes and DDSBrick share the hierarchical design but they differ in the number of layers in the hierarchy. DDSBrick takes advantage of the flexibility from more layers and manages the fine-grained replication and partitioning by a dynamic partitioning strategy. Furthermore, DDSBrick supports transactional index accesses and integrates snapshot transactions to optimize the performance for query workloads that do not require strong consistency. As far as we know, there are transactional distributed data structure designs and distributed spatial data structure designs but there is no distributed transactional spatial data structures in the past.

5.2.2 Tango vs. DDSBrick

Tango provides application developers with an abstraction of a replicated, in-memory data structure based on a distributed shared log, Corfu. Similar to DDSBrick, Tango offloads scalability and transaction burden to a distributed storage system. As shown in Figure 5.5, Corfu offers an ordered, durable, and scalable log and DTranx offers a transactional, durable, and scalable key value store. The choice of a shared log and a key value store leads to three distinct designs: programming model, transaction, and scalability.

Programming Model Tango runtime provides two APIs *update_helper* and *query_helper* and requires the data structure to implement an *apply* Hook function. An example of Tango

integer is shown in Algorithm 3. Both Tango and DDSBrick requires serialization functions to be implemented by the developers and *Transaction* APIs to be called in the external interfaces of the data structures. However, DDSBrick calls for extra functions, such as *Access* APIs in Section 4.3.1, to map tree nodes to keys in DTranx. Tango does not require the mapping since it does not partition the data structure internally.

Algorithm 3 TangoRegister class definition

```
 1: class TangoRegister
 2:     int oid;
 3:     TangoRuntime *T;
 4:     int state;
 5:     procedure APPLY(void *x)
 6:         state ← *(int*)X;
 7:     end procedure
 8:     procedure WRITEREGISTER(int newstate)
 9:         T->UPDATE_HELPER(&newstate, sizeof(int), oid)
10:     end procedure
11:     procedure READREGISTER
12:         T->QUERY_HELPER(oid)
13:         return state;
14:     end procedure
15: endclass
```

Transaction Although both Tango and DDSBrick support serializable transactions and snapshot read transactions and both implement optimistic concurrency control, their designs differ significantly. Tango implements transactions by reserving a position in the ordered log, which is basically an ordered database update list. A transaction only commits successfully if none of its read items are changed during the transaction. DDSBrick launches the 2PC protocol for transaction commit while the concurrency control method is optimistically as well. The difference between reserving a globally ordered log position and 2PC is the level of concurrency and costs. Tango enforces the ordering for every pair of transactions whether or not two transactions are in conflict and DDSBrick enforces orders for transactions when they are in conflict, increasing concurrency. In terms of costs, DDSBrick has the network overhead during 2PC and Tango also pays the network costs when reading the logs from Corfu disks.

Scalability Tango separates the object views from the persistent logs and clients instantiate the in-memory views of the data structures by applying all the relevant logs. If a distributed

data structure is implemented as a single Tango object, throughput for pure update workloads is high but read throughput is mainly bounded by a single client that needs to catch up with all the updates to serve read requests. If a distributed data structure is implemented as multiple Tango objects, all the objects should be instantiated in every server since Tango does not allow a client to execute transactions involving remote reads. The throughput is bounded by the sequencer which maintains the last K offsets for each stream ID. When the number of streams is hundreds of thousands or millions, the sequencer would be swamped.

5.3 Summary

This chapter reviews all DASMs and compares DDSBrick with distributed spatial indexes and another paradigm called Tango for building distributed data structures. Compared to DSAMs, DDSBrick divides the spatial indexes into more layers to manage replication and partitioning flexibly. Moreover, DDSBrick supports transactional accesses, which DSAMs lack. Tango and DDSBrick differ significantly in three aspects: programming model, transaction, and scalability. Tango is targeted for metadata services instead of distributed spatial indexes.

Chapter 6

Conclusion and Future Work

This thesis proposes a new paradigm for scalable, transactional, and efficient spatial indexes. It follows the client-server model where servers offer scalable and transactional storage and clients host a Mempool runtime and a set of intuitive APIs to reduce development efforts. By separating the distributed storage from data structure logic, it enables researchers to design and compare various distributed spatial indexes with little efforts.

This chapter reviews the main contributions of the thesis and discusses its limitations and potential future work.

6.1 Thesis Summary

Transaction DTranx combines optimistic concurrency control and two-phase commit to optimize the performance for serializable transactions. On top of that, snapshot read transactions are introduced for applications that can tolerate a little outdated database. Snapshot read transactions yield high throughput since they bypass distributed commit protocols. Moreover, a persistent-memory based write-ahead log system is designed to lower durability costs and a garbage collection mechanism is brought up to reclaim logs without affecting normal transactions.

Paradigm A Mempool runtime with a set of intuitive APIs are designed on the client side. It enables researchers to experiment on multiple data structures for various applications in a short period. Mempool transparently synchronizes local states of data structures with DTranx such that application designers need only focus on the interactions between local data structures

and Mempool. In 4.3.1, a detailed step-by-step explanation is illustrated to convert a local single-threaded data structure to a distributed one.

Scalability DTranx achieves linear scalability by adopting a dynamic partitioning strategy that combines hashing and adaptive partitioning. The adaptive partitioning balances loads for highly dynamic workloads, such as frequently changing hotspots, and utilizes a graph partitioning algorithm METIS to achieve two goals. First, data that are jointly accessed in transactions are stored together such that no 2PC is necessary. Second, data loads are effectively replicated and balanced across the cluster. Besides the two benefits, DTranx introduces a greedy algorithm to minimize data migrations and adjusts 2PC to enable online database migration while still serving strongly consistent transactions. Last but no the least, this hybrid partitioning method requires little mapping data to be stored since the majority of the data would use the hashing partitioning.

Evaluation We have implemented BTree and RTree in DDSBrick to show the performance. Specifically, BTree search workloads reach 253.07 kops/sec and RTree search workloads reach 77.83 kops/sec. Second, BTree snapshot read throughput is 2.33 the throughput of BTree normal read transactions when there are concurrent update transactions. Third, BTree search throughput after database re-organization is 2.8 times higher than the one before. Lastly, neither snapshot creation transactions nor dynamic migration transactions affect the performance considerably.

6.2 Limitations and Future work

Data Structures DDSBrick assumes that the data structures are divisible into basic

Table 6.1: Data structures mapped to DDSBrick

Data Structure	Data Brick Option 1	Data Brick Option 2
BTree/RTree	Tree node	Subtree
Array/Linked List	One element	A set of elements
Heap	Tree node	Subtree
Trie	Trie node	Subtrie
HashTable/HashMap	One key	A set of keys
Graph	Vertex	Subgraph

bricks and a finer grained division offers more flexibility to distribute bricks and scale out. Current data structures that have been implemented in DDSBrick are trees and tree-like structures are the majority of all data structures. Table 6.1 lists the common data structures and their data brick definitions. Each data structure can be divided in multiple ways where Option 1 is the fine grained division and Option 2 is coarse grained. Application developers can adjust the granularity by configuring the set size of the bricks. Future efforts can be paid to examine the effects of different brick granularities.

Overlapping Regions As shown in 4.5, RTree throughput is approximately 3 times higher than BTree since RTree operations involve higher numbers of bricks in transactions. The main cause is the RTree's overlapping regions. RTree allows children nodes to have overlapped regions in order to reduce update costs and support spatial objects with non-zero volumes. We can further explore data structures with non-overlapping decompositions, such as R^+-Tree [146] and PMR-QuadTree [125]. Another direction is to examine RTree splitting algorithms to reduce internal node overlapping.

Fault Tolerance DDSBrick depends on the METIS algorithm to generate both partitioning and replication plans. Since the access patterns to bricks differ significantly, each brick is assigned with a customized replication factor. In Figure 5.4, the root brick is replicated in every server while the leaf bricks are not replicated. METIS produces a replication plan that assigns high replication factors for read intensive bricks. It inevitably leads to some bricks to have no replications, thus low availability. A hybrid approach combining both METIS and static replications for leaf bricks can be explored to provide high availability.

Mapping data structure Maintenance Currently, DDSBrick requires that each server store a copy of the mapping data structures, which record the partition and replication plans. The mapping data are kept strongly consistent as explained in 4.4.2. Since the DMT transactions are run infrequently, the consistency costs are negligible. The concern is the growing size of the mapping data structure as the databases are expanding. By inspecting the mapping data size of a BTree with 30 millions spatial objects, the mapping data is less than 5MB. This is because

the majority of the bricks are not showing strong access patterns that hashing based partitioning suffices for load balance. If application developers design a data structure of which all the bricks show strong access patterns and workload hot spots are frequently changing, there will be a need for a better structure for the mapping data, such as compression.

Bibliography

[1] Geohas. http://geohash.org/.

[2] Hbase. http://hbase.apache.org.

[3] Intel NVML. https://pmem.io/.

[4] Leveldb – a fast and lightweight key/value database library by google http://code.google.com/p/leveldb/.

[5] Linux strace. https://strace.io/.

[6] Google Btree, December 2011.

[7] Karl Aberer. P-grid: A self-organizing access structure for p2p information systems. In Carlo Batini, Fausto Giunchiglia, Paolo Giorgini, and Massimo Mecella, editors, Cooperative Information Systems, pages 179–194, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[8] Marcos K. Aguilera, Wojciech Golab, and Mehul A. Shah. A practical scalable distributed b-tree. PVLDB, 1(1):598–609, 2008.

[9] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In SOSP, pages 159–174, 2007.

[10] Ablimit Aji, Fusheng Wang, and Joel H. Saltz. Towards building a high performance spatial query system for large scale medical imaging data. In Proceedings of the 20th International Conference on Advances in Geographic Information Systems, SIGSPATIAL '12, pages 309–318, New York, NY, USA, 2012. ACM.

[11] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel Saltz. Hadoop gis: A high performance spatial data warehousing system over mapreduce. Proc. VLDB Endow., 6(11):1009–1020, August 2013.

[12] A. Akdogan, U. Demiryurek, F. Banaei-Kashani, and C. Shahabi. Voronoi-based geospatial query processing with mapreduce. In 2010 IEEE Second International Conference on Cloud Computing Technology and Science, pages 9–16, Nov 2010.

[13] A. Akdogan, C. Shahabi, and U. Demiryurek. D-toss: A distributed throwaway spatial index structure for dynamic location data. IEEE Transactions on Knowledge and Data Engineering, 28(9):2334–2348, Sept 2016.

[14] Saeed Alaei, Mohammad Toossi, and Mohammad Ghodsi. Skiptree: A scalable range-queryable distributed data structure for multidimensional data. In Xiaotie Deng and Ding-Zhu Du, editors, Algorithms and Computation, pages 298–307, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[15] Mohamed Aly, Mario Munich, and Pietro Perona. Aly et al.: Distributed kd-trees for retrieval from large image collections1 distributed kd-trees for retrieval from very large image collections, 2011.

[16] A. Andrzejak and Zhichen Xu. Scalable, efficient range queries for grid information services. In Proceedings. Second International Conference on Peer-to-Peer Computing,, pages 33–40, 2002.

[17] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, Alexander Driskill-Smith, and Mohamad Krounbi. Spin-transfer torque magnetic random access memory (stt-mram). J. Emerg. Technol. Comput. Syst., 9(2):13:1–13:35, May 2013.

[18] Lars Arge, David Eppstein, and Michael T. Goodrich. Skip-webs: Efficient distributed data structures for multi-dimensional data sets. In Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing, PODC '05, pages 69–76, New York, NY, USA, 2005. ACM.

[19] James Aspnes and Gauri Shah. Skip graphs. ACM Trans. Algorithms, 3(4), November 2007.

[20] Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. Corfu: A distributed shared log. ACM Trans. Comput. Syst., 31(4):10:1–10:24, 2013.

[21] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In SOSP, pages 325–340, 2013.

[22] G. I. Baylor. Up, up and away. Proc. Roy. Soc., London A, 294:456–475, 1959.

[23] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, SIGMOD '90, pages 322–331, New York, NY, USA, 1990. ACM.

[24] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. ACM Comput. Surv., 13(2):185–221, June 1981.

[25] Ashwin R. Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: Supporting scalable multi-attribute range queries. In Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM '04, pages 353–366, New York, NY, USA, 2004. ACM.

[26] Carsten Binnig, Stefan Hildenbrand, Franz Färber, Donald Kossmann, Juchang Lee, and Norman May. Distributed snapshot isolation: Global transactions pay globally, local transactions pay locally. PVLDB, 23(6):987–1011, 2014.

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

[28] E. Bongers and J. Pouwelse. A survey of P2P multidimensional indexing structures. ArXiv e-prints, July 2015.

[29] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable isolation for snapshot databases. In SIGMOD, pages 729–738, 2008.

[30] Min Cai, Martin Frank, Jinbo Chen, and Pedro Szekely. Maan: A multi-attribute addressable network for grid information services. In Proceedings of the 4th International Workshop on Grid Computing, GRID '03, pages 184–, Washington, DC, USA, 2003. IEEE Computer Society.

[31] Wenyuan Cai, Shuigeng Zhou, Linhao Xu, Weining Qian, and Aoying Zhou. C 2: A new overlay network based on can and chord. In Minglu Li, Xian-He Sun, Qian-ni Deng, and Jun Ni, editors, Grid and Cooperative Computing, pages 42–50, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[32] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[33] R. Cheng, Y. Xia, Sunil Prabhakar, and Rahul Shah. Change tolerant indexing for constantly evolving data. In 21st International Conference on Data Engineering (ICDE'05), pages 391–402, April 2005.

[34] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In Proceedings of the 23rd International Conference on Very Large Data Bases, VLDB '97, pages 426–435, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[35] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In SOCC, pages 143–154, 2010.

[36] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. ACM Trans. Comput. Syst., 31(3):8:1–8:22, 2013.

[37] James Cowling and Barbara Liskov. Granola: Low-overhead distributed transaction coordination. In USENIX ATC, pages 223–235, 2012.

[38] Adina Crainiceanu, Prakash Linga, Johannes Gehrke, and Jayavel Shanmugasundaram. Querying peer-to-peer networks using p-trees. In Proceedings of the 7th International Workshop on the Web and Databases: Colocated with ACM SIGMOD/PODS 2004, WebDB '04, pages 25–30, New York, NY, USA, 2004. ACM.

[39] Adina Crainiceanu, Prakash Linga, Ashwin Machanavajjhala, Johannes Gehrke, and Jayavel Shanmugasundaram. P-ring: An efficient and robust p2p range index structure. In Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, SIGMOD '07, pages 223–234, New York, NY, USA, 2007. ACM.

[40] M. E. Crow. Aerodynamic sound emission as a singular perturbation problem. Stud. Appl. Math., 29:21–44, 1968.

[41] Cesare Cugnasco, Yolanda Becerra, Jordi Torres, and Eduard Ayguadé. D8-tree: A de-normalized approach for multidimensional data analysis on key-value databases. In Proceedings of the 17th International Conference on Distributed Computing and Networking, ICDCN '16, pages 18:1–18:10, New York, NY, USA, 2016. ACM.

[42] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: A workload-driven approach to database replication and partitioning. PVLDB, 3(1-2):48–57, 2010.

[43] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[44] David J. DeWitt, Navin Kabra, Jun Luo, Jignesh M. Patel, and Jie-Bing Yu. Client-server paradise. In Proceedings of the 20th International Conference on Very Large Data Bases, VLDB '94, pages 558–569, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[45] Adriano Di Pasquale and EnricoZ Nardelli. Distributed searching of k-dimensional data with almost constant costs. In Július Štuller, Jaroslav Pokorný, Bernhard Thalheim, and Yoshifumi Masunaga, editors, Current Issues in Databases and Information Systems, pages 239–250, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[46] N. Diegues and P. Romano. STI-BT: A scalable transactional index. In ICDCS, pages 104–113, 2014.

[47] Linlin Ding, Baiyou Qiao, Guoren Wang, and Chen Chen. An efficient quad-tree based index structure for cloud data management. In Proceedings of the 12th International Conference on Web-age Information Management, WAIM'11, pages 238–250, Berlin, Heidelberg, 2011. Springer-Verlag.

[48] Julian D. Dole. Perturbation Methods in Applied Mathematics. Winsdell Publishing Company, 1967.

[49] C. du Mouza, W. Litwin, and P. Rigaux. SD-Rtree: A Scalable Distributed Rtree. In IEEE International Conference on Data Engineering, pages 296–305, April 2007.

[50] C. du Mouza, W. Litwin, and P. Rigaux. SDR-tree: a Scalable Distributed Rtree. In ICDE, pages 296–305, 2007.

[51] Cédric du Mouza, Witold Litwin, and Philippe Rigaux. Dynamic storage balancing in a distributed spatial index. In Proceedings of the 15th Annual ACM International Symposium on Advances in Geographic Information Systems, GIS '07, pages 5:1–5:8, New York, NY, USA, 2007. ACM.

[52] A. Eldawy and M. F. Mokbel. Spatialhadoop: A mapreduce framework for spatial data. In 2015 IEEE 31st International Conference on Data Engineering, pages 1352–1363, April 2015.

[53] A. Eldawy, M. F. Mokbel, S. Alharthi, A. Alzaidy, K. Tarek, and S. Ghani. Shahed: A mapreduce-based system for querying and visualizing spatio-temporal satellite data. In 2015 IEEE 31st International Conference on Data Engineering, pages 1585–1596, April 2015.

[54] Ahmed Eldawy and Mohamed F. Mokbel. The era of big spatial data: A survey. Found. Trends databases, 6(3-4):163–273, December 2016.

[55] Robert Escriva and Emin Gun Sirer. The design and implementation of the warp transactional filesystem. In NSDI, pages 469–483, 2016.

[56] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Hyperdex: A distributed, searchable key-value store. SIGCOMM Comput. Commun. Rev., 42(4):25–36, 2012.

[57] J. S. Fabnis, H. J. Giblet, and H. McDormand. Navier-stokes analysis of solid rocket motor internal flow. J. Prop. and Power, 2:157–164, 1980.

[58] Fabrizio Falchi, Claudio Gennaro, and Pavel Zezula. Nearest neighbor search in metric spaces through content-addressable networks. Information Processing & Management, 44(1):411 – 429, 2008. Evaluation of Interactive Information Retrieval Systems.

[59] A. Fox, C. Eichelberger, J. Hughes, and S. Lyon. Spatio-temporal indexing in non-relational distributed databases. In 2013 IEEE International Conference on Big Data, pages 291–299, Oct 2013.

[60] Xiaodong Fu, Dingxing Wang, Weimin Zheng, and Meiming Sheng. Gpr-tree: a global parallel index structure for multiattribute declustering on cluster of workstations. In Proceedings. Advances in Parallel and Distributed Computing, pages 300–306, Mar 1997.

[61] Volker Gaede and Oliver Günther. Multidimensional Access Methods. ACM Comput. Surv., 30(2):170–231, June 1998.

[62] Prasanna Ganesan, Beverly Yang, and Hector Garcia-Molina. One torus to rule them all: Multi-dimensional queries in p2p systems. In Proceedings of the 7th International Workshop on the Web and Databases: Colocated with ACM SIGMOD/PODS 2004, WebDB '04, pages 19–24, New York, NY, USA, 2004. ACM.

[63] N. Gao, Z. Liu, and D. Grunwald. DTranx: A SEDA-based Distributed and Transactional Key Value Store with Persistent Memory Log. ArXiv e-prints, November 2017.

[64] Hector Garcia-Molina. Database systems: the complete book. Pearson Education India, 2008.

[65] Vipin Kumar George Karypis. A fast and highly quality multilevel scheme for partitioning irregular graphs. SIAM Journal on Scientific Computing, 20(1):359—392, 1999.

[66] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[67] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[68] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. Scalable, distributed data structures for internet service construction. In OSDI, pages 319–332, 2000.

[69] F. Guillot and Z. Javalon. Acoustic boundary layers in propellant rocket motors. J. Prop. and Power, 5:331–339, 1989.

[70] O. Gunther. The design of the cell tree: an object-oriented index structure for geometric databases. In [1989] Proceedings. Fifth International Conference on Data Engineering, pages 598–605, Feb 1989.

[71] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In SIGMOD, pages 47–57, 1984.

[72] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM.

[73] D. Han and E. Stroulia. Hgrid: A data model for large geospatial data sets in hbase. In 2013 IEEE Sixth International Conference on Cloud Computing, pages 910–917, June 2013.

[74] Theo Härder. Observations on optimistic concurrency control schemes. Inf. Syst., 9(2):111–120, November 1984.

[75] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In Proceedings of the 15th International Conference on Distributed Computing, DISC '01, pages 300–314, London, UK, UK, 2001. Springer-Verlag.

[76] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: A scalable overlay network with practical locality properties. In Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4, USITS'03, pages 9–9, Berkeley, CA, USA, 2003. USENIX Association.

[77] Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. Transactional interference-less balanced tree. In Proceedings of the 29th International Symposium on Distributed Computing - Volume 9363, DISC 2015, pages 325–340, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[78] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In Proceedings of the 21th International Conference on Very Large Data Bases, VLDB '95, pages 562–573, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[79] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In Proceedings of the 23rd International Conference on Distributed Computing Systems, ICDCS '03, pages 522–, Washington, DC, USA, 2003. IEEE Computer Society.

[80] Y. Hua, B. Xiao, and J. Wang. BR-Tree: A Scalable Prototype for Supporting Multiple Queries of Multidimensional Data. IEEE Transactions on Computers, 58(12):1585–1598, Dec 2009.

[81] IBM. SPSS Statistics. download from vendor site, 2012. version 21.

[82] Sergio Ilarri, Eduardo Mena, and Arantza Illarramendi. Location-dependent Query Processing: Where We Are and Where We Are Heading. ACM Comput. Surv., 42(3):12:1–12:73, March 2010.

[83] H. V. Jagadish, Beng Chin Ooi, and Quang Hieu Vu. Baton: A balanced tree structure for peer-to-peer networks. In Proceedings of the 31st International Conference on Very Large Data Bases, VLDB '05, pages 661–672. VLDB Endowment, 2005.

[84] H. V. Jagadish, Beng Chin Ooi, Quang Hieu Vu, Rong Zhang, and Aoying Zhou. Vbi-tree: A peer-to-peer framework for supporting multi-dimensional indexing schemes. In 22nd International Conference on Data Engineering (ICDE'06), pages 34–34, April 2006.

[85] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA efficiently for key-value services. SIGCOMM Comput. Commun. Rev., 44(4):295–306, 2014.

[86] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[87] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In STOC, pages 654–663, 1997.

[88] Jonas S. Karlsson. hqt*: A scalable distributed data structure for high-performance spatial accesses. In In Foundations of Data Organization and Algorithms (FODO, pages 37–46, 1998.

[89] Mohammad Kolahdouzan and Cyrus Shahabi. Voronoi-based k nearest neighbor search for spatial network databases. In Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30, VLDB '04, pages 840–851. VLDB Endowment, 2004.

[90] Nikos Koudas, Christos Faloutsos, and Ibrahim Kamel. Declustering spatial databases on a multi-computer architecture. In Peter Apers, Mokrane Bouzeghoub, and Georges Gardarin, editors, Advances in Database Technology — EDBT '96, pages 592–614, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[91] Vassil Kriakov, Alex Delis, and George Kollios. Management of highly dynamic multidimensional data in a cluster of workstations. In Elisa Bertino, Stavros Christodoulakis, Dimitris Plexousakis, Vassilis Christophides, Manolis Koubarakis, Klemens Böhm, and Elena Ferrari, editors, Advances in Database Technology - EDBT 2004, pages 748–764, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[92] Dongseop Kwon, Sangjun Lee, and Sukho Lee. Indexing the current positions of moving objects using the lazy update r-tree. In Proceedings Third International Conference on Mobile Data Management MDM 2002, pages 113–120, Jan 2002.

[93] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. SIGOPS Oper. Syst. Rev., 44(2):35–40, 2010.

[94] Henry Lao. Linear Acoustic Processes in Rocket Engines. PhD thesis, University of Colorado at Boulder, 1979.

[95] Q. Lao, M. N. Cassoy, and K. Kirkpatrick. Acoustically generated vorticity from internal flow. J. Fluid Mechanics, 2:122–133, 1996.

[96] Q. Lao, D. R. Kassoy, and K. Kirkkopru. Nonlinear acoustic processes in rocket engines. J. Fluid Mechanics, 3:245–261, 1997.

[97] J. K. Lawder and P. J. H. King. Using Space-Filling Curves for Multi-dimensional Indexing, pages 20–35. 2000.

[98] Kisung Lee, Raghu K. Ganti, Mudhakar Srivatsa, and Ling Liu. Efficient spatial query processing for big data. In Proceedings of the 22Nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL '14, pages 469–472, New York, NY, USA, 2014. ACM.

[99] S. T. Leutenegger, M. A. Lopez, and J. Edgington. Str: a simple and efficient algorithm for r-tree packing. In Proceedings 13th International Conference on Data Engineering, pages 497–506, Apr 1997.

[100] Shen Li, Shaohan Hu, Raghu Ganti, Mudhakar Srivatsa, and Tarek Abdelzaher. Pyro: A Spatial-temporal Big-data Storage System. In Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference, pages 97–109, 2015.

[101] H. Liao, J. Han, and J. Fang. Multi-dimensional index on hadoop distributed file system. In 2010 IEEE Fifth International Conference on Networking, Architecture, and Storage, pages 240–249, July 2010.

[102] W. Litwin and M. A. Neimat. k-rp*s: a scalable distributed data structure for high-performance multi-attribute access. In Fourth International Conference on Parallel and Distributed Information Systems, pages 120–131, Dec 1996.

[103] Witold Litwin, Marie-Anna Neimat, and Donovan A. Schneider. Lh*—a scalable, distributed data structure. ACM Trans. Database Syst., 21(4):480–525, December 1996.

[104] Witold Litwin, Marie-Anne Neimat, and Donovan A. Schneider. Rp*: A family of order preserving scalable distributed data structures. In Proceedings of the 20th International Conference on Very Large Data Bases, VLDB '94, pages 342–353, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[105] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. PVLDB, 5(8):716–727, 2012.

[106] Peng Lu, Gang Chen, Beng Chin Ooi, Hoang Tam Vo, and Sai Wu. Scalagist: Scalable generalized search trees for mapreduce systems [innovative systems paper]. Proc. VLDB Endow., 7(14):1797–1808, October 2014.

[107] Qiang Ma, Bin Yang, Weining Qian, and Aoying Zhou. Query processing of massive trajectory data based on mapreduce. In Proceedings of the First International Workshop on Cloud Data Management, CloudDB '09, pages 9–16, New York, NY, USA, 2009. ACM.

[108] Youzhong Ma, Xiaofeng Meng, Shaoya Wang, Weisong Hu, Xu Han, and Yu Zhang. An efficient index method for multi-dimensional query in cloud environment. In Weizhong Qiang, Xianghan Zheng, and Ching-Hsien Hsu, editors, Cloud Computing and Big Data, pages 307–318, Cham, 2015. Springer International Publishing.

[109] Youzhong Ma, Jia Rao, Weisong Hu, Xiaofeng Meng, Xu Han, Yu Zhang, Yunpeng Chai, and Chunqiu Liu. An Efficient Index for Massive IOT Data in Cloud Environment. In ACM International Conference on Information and Knowledge Management, pages 2129–2133, 2012.

[110] Youzhong Ma, Yu Zhang, and Xiaofeng Meng. St-hbase: A scalable data management system for massive geo-tagged objects. In Proceedings of the 14th International Conference on Web-Age Information Management, WAIM'13, pages 155–166, Berlin, Heidelberg, 2013. Springer-Verlag.

[111] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In OSDI, pages 8–8, 2004.

[112] Krassimir Markov, Krassimira Ivanova, Ilia Mitov, and Stefan Karastanev. Advance of the access methods. 2, 01 2008.

[113] Leonardo Marmol, Jorge Guerra, and Marcos K. Aguilera. Non-volatile memory through customized key-value stores. In USENIX HotStorage, pages 101–105, 2016.

[114] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter 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.

[115] Mohamed F. Mokbel, Thanaa M. Ghanem, and Walid G. Aref. Spatio-temporal access methods. IEEE Data Eng. Bull., 26:40–49, 2003.

[116] Anirban Mondal, Yi Lifu, and Masaru Kitsuregawa. P2pr-tree: An r-tree-based spatial index for peer-to-peer environments. In Wolfgang Lindner, Marco Mesiti, Can Türker, Yannis Tzitzikas, and Athena I. Vakali, editors, Current Trends in Database Technology - EDBT 2004 Workshops, pages 516–525, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[117] F. C. Mulick. Rotational axisymmetric mean flow and damping of acoustic waves in a solid propellant. AIAA J., 3:1062–1063, 1964.

[118] F. C. Mulick. Stability of four-dimensional motions in a combustion chamber. Comb. Sci. Tech., 19:99–124, 1981.

[119] B. Nam and A. Sussman. Spatial indexing of distributed multidimensional datasets. In CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005., volume 2, pages 743–750 Vol. 2, May 2005.

[120] Beomseok Nam and A. Sussman. Dist: fully decentralized indexing for querying distributed multidimensional datasets. In Proceedings 20th IEEE International Parallel Distributed Processing Symposium, pages 10 pp.–, April 2006.

[121] Beomseok Nam and Alan Sussman. Analyzing design choices for distributed multidimensional indexing. J. Supercomput., 59(3):1552–1576, March 2012.

[122] Enrico Nardelli. Distributed k-d trees. In In Proceedings 16th Conference of Chilean Computer Science Society (SCCC96, pages 142–154, 1996.

[123] Enrico Nardelli, Fabio Barillari, and Massimo Pepe. Distributed searching of multi-dimensional data: A performance evaluation study. Journal of Parallel and Distributed Computing, 49(1):111 – 134, 1998.

[124] Hamid Nazerzadeh and Mohammad Ghodsi. Raq: A range-queriable distributed data structure. In Peter Vojtáš, Mária Bieliková, Bernadette Charron-Bost, and Ondrej Sýkora, editors, SOFSEM 2005: Theory and Practice of Computer Science, pages 269–277, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[125] Randal C. Nelson and Hanan Samet. A population analysis for hierarchical data structures. In Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data, SIGMOD '87, pages 270–277, New York, NY, USA, 1987. ACM.

[126] Duc Hai Nguyen, Khue Doan, and Tran Vu Pham. Sidi: A scalable in-memory density-based index for spatial databases. In Proceedings of the ACM International Workshop on Data-Intensive Distributed Computing, DIDC '16, pages 45–52, New York, NY, USA, 2016. ACM.

[127] Long-Van Nguyen-Dinh, Walid G. Aref, and Mohamed F. Mokbel. Spatio-temporal access methods: Part 2 (2003 - 2010). IEEE Data Eng. Bull., 33:46–55, 2010.

[128] S. Nishimura, S. Das, D. Agrawal, and A. E. Abbadi. Md-hbase: A scalable multi-dimensional data infrastructure for location aware services. In 2011 IEEE 12th International Conference on Mobile Data Management, volume 1, pages 7–16, June 2011.

[129] Atsuyuki Okabe, Barry Boots, and Kokichi Sugihara. Spatial Tessellations: Concepts and Applications of Voronoi Diagrams. John Wiley & Sons, Inc., New York, NY, USA, 1992.

[130] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In USENIX ATC, pages 305–319, 2014.

[131] Aris M. Ouksel and Gianluca Moro. G-grid: A class of scalable and self-organizing data structures for multi-dimensional querying and content routing in p2p networks. In Gianluca Moro, Claudio Sartori, and Munindar P. Singh, editors, Agents and Peer-to-Peer Computing, pages 123–137, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[132] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The RAMCloud storage system. ACM Trans. Comput. Syst., 33(3):7:1–7:55, 2015.

[133] M. Mealling P. Leach and R. Salz. A universally unique identifier (uuid) urn namespace. RFC 4122, RFC Editor, July 2005.

[134] Andreas Papadopoulos and Dimitrios Katsaros. A-Tree: Distributed Indexing of Multidimensional Data for Cloud Computing Environments. 2011 IEEE Third International Conference on Cloud Computing Technology and Science, pages 407–414, 2011.

[135] Jignesh Patel, JieBing Yu, Navin Kabra, Kristin Tufte, Biswadeep Nag, Josef Burger, Nancy Hall, Karthikeyan Ramasamy, Roger Lueder, Curt Ellmann, Jim Kupsch, Shelly Guo, Johan Larson, David De Witt, and Jeffrey Naughton. Building a scaleable geo-spatial dbms: Technology, implementation, and evaluation. In Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data, SIGMOD '97, pages 336–347, New York, NY, USA, 1997. ACM.

[136] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In ICDCS, pages 455–465, 2012.

[137] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM '01, pages 161–172, New York, NY, USA, 2001. ACM.

[138] Robert Ricci, Eric Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. USENIX ;login:, 39(6), 2014.

[139] R. S. Richards and A. M. Brown. Coupling between acoustic velocity oscillations and solid propellant combustion. J. Prop. and Power, 5:828–837, 1982.

[140] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In Rachid Guerraoui, editor, Middleware 2001, pages 329–350, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[141] Betty Salzberg and Vassilis J. Tsotras. Comparison of access methods for time-evolving data. ACM Comput. Surv., 31(2):158–221, June 1999.

[142] Hanan Samet. The quadtree and related hierarchical data structures. ACM Comput. Surv., 16(2):187–260, June 1984.

[143] Hanan Samet. Algorithms and theory of computation handbook. chapter Multidimensional Data Structures for Spatial Applications, pages 6–6. Chapman & Hall/CRC, 2010.

[144] C. Schmidt and M. Parashar. Flexible information discovery in decentralized distributed systems. In High Performance Distributed Computing, 2003. Proceedings. 12th IEEE International Symposium on, pages 226–235, June 2003.

[145] B. Schnitzer and S. T. Leutenegger. Master-client r-trees: a new parallel r-tree architecture. In Proceedings. Eleventh International Conference on Scientific and Statistical Database Management, pages 68–77, Aug 1999.

[146] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In Proceedings of the 13th International Conference on Very Large Data Bases, VLDB '87, pages 507–518, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers Inc.

[147] Yanfeng Shu, Beng Chin Ooi, Kian-Lee Tan, and Aoying Zhou. Supporting multi-dimensional range queries in peer-to-peer systems. In Fifth IEEE International Conference on Peer-to-Peer Computing (P2P'05), pages 173–180, Aug 2005.

[148] Lai Shuhua, Zhu Fenghua, and Sun Yongqiang. A design of parallel r-tree on cluster of workstations. In Subhash Bhalla, editor, Databases in Networked Information Systems, pages 119–133, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[149] T. M. Smitty, R. L. Coach, and F. B. Höndra. Unsteady flow in simulated solid rocket motors. In 16st Aerospace Sciences Meeting, number 0112 in 78. AIAA, 1978.

[150] Benjamin Sowell, Wojciech Golab, and Mehul A. Shah. Minuet: A scalable distributed multiversion b-tree. PVLDB, 5(9):884–895, 2012.

[151] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[152] M. Stonebraker, P. Brown, D. Zhang, and J. Becla. Scidb: A database management system for applications with complex analytics. Computing in Science Engineering, 15(3):54–62, May 2013.

[153] Y. Tang, S. Zhou, and J. Xu. Light: A query-efficient yet low-maintenance indexing scheme over dhts. IEEE Transactions on Knowledge and Data Engineering, 22(1):59–75, Jan 2010.

[154] Yufei Tao and Dimitris Papadias. Mv3r-tree: A spatio-temporal access method for timestamp and interval queries. In Proceedings of the 27th International Conference on Very Large Data Bases, VLDB '01, pages 431–440, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[155] Joseph D. Taum. Investigation of flow turning phenomenon. In 20th Aerospace Sciences Meeting, number 0297 in 82. AIAA, 1982.

[156] A. Thomasian. Distributed optimistic concurrency control methods for high-performance transaction processing. IEEE Transactions on Knowledge and Data Engineering, 10(1):173–189, 1998.

[157] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In SIGMOD, pages 1–12, 2012.

[158] George Tsatsanifos, Dimitris Sacharidis, and Timos Sellis. Index-based query processing on distributed multidimensional data. GeoInformatica, 17(3):489–519, Jul 2013.

[159] A. Turcu, R. Palmieri, B. Ravindran, and S. Hirve. Automated data partitioning for highly scalable and strongly consistent transactions. IEEE Transactions on Parallel and Distributed Systems, 27(1):106–118, Jan 2016.

[160] Simonas Šaltenis, Christian S. Jensen, Scott T. Leutenegger, and Mario A. Lopez. Indexing the positions of continuously moving objects. In Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, SIGMOD '00, pages 331–342, New York, NY, USA, 2000. ACM.

[161] Fusheng. Wang, Jun. Kong, Jingjing. Gao, Lee. Cooper, Tahsin. Kurc, Zhengwen. Zhou, David. Adler, Cristobal. Vergara-Niedermayr, Bryan. Katigbak, Daniel. Brat, and Joel. Saltz. A high-performance spatial database based approach for pathology imaging algorithm evaluation. Journal of Pathology Informatics, 4(1):5, 2013.

[162] Jinbao Wang, Sai Wu, Hong Gao, Jianzhong Li, and Beng Chin Ooi. Indexing multidimensional data in a cloud system. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10, pages 591–602, New York, NY, USA, 2010. ACM.

[163] Tianzheng Wang and Ryan Johnson. Scalable logging through emerging non-volatile memory. PVLDB, 7(10):865–876, 2014.

[164] Ling-Yin Wei, Ya-Ting Hsu, Wen-Chih Peng, and Wang-Chien Lee. Indexing spatial data in cloud data managements. Pervasive Mob. Comput., 15(C):48–61, December 2014.

[165] Ling-Yin Wei, Ya-Ting Hsu, Wen-Chih Peng, and Wang-Chien Lee. Indexing spatial data in cloud data managements. Pervasive Mob. Comput., 15(C):48–61, December 2014.

[166] X. Wei and K. Sezaki. Dhr-trees: A distributed multidimensional indexing structure for p2p systems. In 2006 Fifth International Symposium on Parallel and Distributed Computing, pages 281–290, July 2006.

[167] Matt Welsh, David Culler, and Eric Brewer. SEDA: An architecture for well-conditioned, scalable internet services. SIGOPS Oper. Syst. Rev., 35(5):230–243, 2001.

[168] Tom White. Hadoop: The Definitive Guide. O'Reilly Media, Inc., Sebastopol, CA, 2009.

[169] Randall T. Whitman, Michael B. Park, Sarah M. Ambrose, and Erik G. Hoel. Spatial indexing and analytics on hadoop. In Proceedings of the 22Nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL '14, pages 73–82, New York, NY, USA, 2014. ACM.

[170] Sai Wu and Kun-Lung Wu. An indexing framework for efficient retrieval on the cloud. 32:75–82, 01 2009.

[171] Xiaopeng Xiong, M. F. Mokbel, and W. G. Aref. Lugrid: Update-tolerant grid-based indexing for moving objects. In 7th International Conference on Mobile Data Management (MDM'06), pages 13–13, May 2006.

[172] Maysam Yabandeh and Daniel Gómez Ferro. A critique of snapshot isolation. In EuroSys, pages 155–168, 2012.

[173] Fan Yang, Jinfeng Li, and James Cheng. Husky: Towards a more efficient and expressive distributed computing framework. Proc. VLDB Endow., 9(5):420–431, January 2016.

[174] Hung-chih Yang and D. Stott Parker. Traverse: Simplified indexing on large map-reduce-merge clusters. In Xiaofang Zhou, Haruo Yokota, Ke Deng, and Qing Liu, editors, Database Systems for Advanced Applications, pages 308–322, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[175] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), pages 15–28, San Jose, CA, 2012. USENIX.

[176] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In USENIX HotCloud, pages 10–10, 2010.

[177] Robert A. Zeddini. Injection-induced flows in porous-walled ducts. AIAA Journal, 14:766–773, 1981.

[178] Chi Zhang, Arvind Krishnamurthy, and Randolph Y. Wang. Skipindex: Towards a scalable peer-to-peer index service for high dimensional data. 2004.

[179] S. Zhang, J. Han, Z. Liu, K. Wang, and S. Feng. Spatial queries evaluation with mapreduce. In 2009 Eighth International Conference on Grid and Cooperative Computing, pages 287–292, Aug 2009.

[180] Xiangyu Zhang, Jing Ai, Zhongyuan Wang, Jiaheng Lu, and Xiaofeng Meng. An efficient multi-dimensional index for cloud data management. In CloudDB, pages 17–24, 2009.

[181] Xiangyu Zhang, Jing Ai, Zhongyuan Wang, Jiaheng Lu, and Xiaofeng Meng. An efficient multi-dimensional index for cloud data management. In Proceedings of the First International Workshop on Cloud Data Management, CloudDB '09, pages 17–24, New York, NY, USA, 2009. ACM.

[182] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In SOSP, pages 276–291, 2013.

[183] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. Fast databases with fast durability and recovery through multicore parallelism. In OSDI, pages 465–477, 2014.

[184] Y. Zhong, J. Han, T. Zhang, Z. Li, J. Fang, and G. Chen. Towards parallel spatial query processing for big spatial data. In 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum, pages 2085–2094, May 2012.

[185] Xin Zhou, Hui Li, Xiao Zhang, Shan Wang, Yanyu Ma, Keyan Liu, Ming Zhu, and Menglin Huang. ABR-Tree: An Efficient Distributed Multidimensional Indexing Approach for Massive Data. In Algorithms and Architectures for Parallel Processing, pages 781–790, Cham, 2015. Springer International Publishing.

[186] Xin Zhou, Xiao Zhang, Yanhao Wang, Rui Li, and Shan Wang. Efficient distributed multi-dimensional index for big data management. In Jianyong Wang, Hui Xiong, Yoshiharu Ishikawa, Jianliang Xu, and Junfeng Zhou, editors, Web-Age Information Management, pages 130–141, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[187] Yongqiang Zou, Jia Liu, Shicai Wang, Li Zha, and Zhiwei Xu. Ccindex: A complemental clustering index on distributed ordered tables for multi-dimensional range queries. In Chen Ding, Zhiyuan Shao, and Ran Zheng, editors, Network and Parallel Computing, pages 247–261, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.