WORQ-Tree: A New Data Structure for Dynamic High-Dimensional Semi-Group Orthogonal Range Searching

Xiaotian You, Raymond Chi-Wing Wong
The Hong Kong University of Science and Technology
{xyouaa,raywong}@cse.ust.hk

ABSTRACT

Semigroup orthogonal range searching, which is to aggregate the values in an orthogonal range, has been widely discussed for a long time. With the age of information coming, the data generating speed hits a new record everyday. It is important to have a write-efficient database system to store the data and make it possible to do semigroup orthogonal range searching with reasonable efficiency.

A linear-space high-dimensional data structure is important for such write-intensive workloads. In this paper, we proposed a new data structure named WORQ-tree of size $O(N)$ where N is the number of data records. Its amortized insertion I/O cost is $O(d \log_M \frac{N}{B} \log_2 \frac{N}{M}/B)$ and the worst-case query I/O cost is $O((N/B)^{\frac{2d-3}{2d-1}} \log_B \frac{N}{B} \log_2 \frac{N}{M})$, which is the best query I/O guarantee among all the $O(N)$-space multi-dimensional data structures, where M is the maximum number of blocks of records stored in memory, B is the number of records in a block and d is the number of dimensions in the records. To further balance the insertion and query cost, we propose a new lazy merge method, which adds tolerance of range overlap. With a given lazy constant K, we can make the amortized insertion I/O cost at least $\log K$ times lower by sacrificing the worst-case query I/O cost at most K times higher. We conducted various numerical experiments where the results show the insertion of our data structure outperforms other state-of-the-art high-dimensional data structures, which is at least 5.5 times faster than other baselines on HDD and 4.4 times faster on SSD in heavy insertion tests. The query time is also the shortest among the $O(N)$-space baselines, at least 1.4 times faster on HDD and at least 2.2 times faster on SSD. The lazy merge can make insertions of the data structure up to 110% faster than the original one.

PVLDB Reference Format:
Xiaotian You, Raymond Chi-Wing Wong. WORQ-Tree: A New Data Structure for Dynamic High-Dimensional Semi-Group Orthogonal Range Searching. PVLDB, 15(1): XXX-XXX, 2022.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/papercodepapercode/vldbworq.

Proceedings of the VLDB Endowment, Vol. 15, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

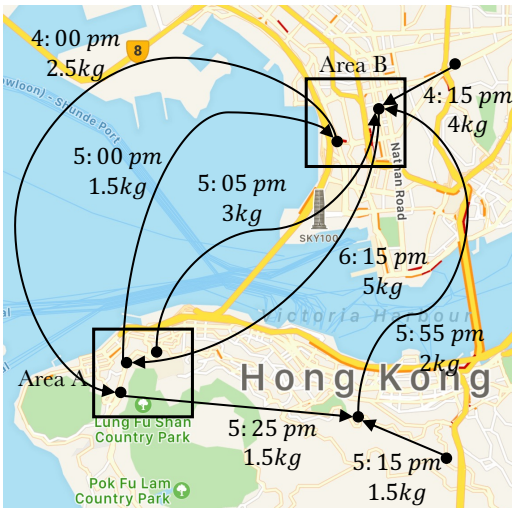


Figure 1: An illustration of courier information

T	$From$	To	w
4:00 pm	(22.3119, 114.1625)	(22.2784, 114.1307)	$2.5kg$
4:15 pm	(22.3253, 114.1711)	(22.3181, 114.1674)	$4kg$
5:00 pm	(22.2831, 114.1323)	(22.3119, 114.1625)	$1.5kg$
5:05 pm	(22.2852, 114.1374)	(22.3181, 114.1674)	$3kg$
5:15 pm	(22.2709, 114.1703)	(22.2744, 114.1686)	$1.5kg$
5:25 pm	(22.2784, 114.1307)	(22.2784, 114.1307)	$1.5kg$
5:55 pm	(22.2744, 114.1686)	(22.3181, 114.1674)	$2kg$
6:15 pm	(22.3181, 114.1674)	(22.2852, 114.1374)	$1kg$

Table 1: An example of courier information

1 INTRODUCTION

Over the last decades, oceans of data have been produced, stored, and analyzed. In addition, with the coming of new technologies such as 5G and IoT, we will have even more and more data, which will make a huge challenge to the efficiency of the database systems. Meanwhile, doing analytical work under a write-intensive data workload is a real-world demand in many areas. In 2020, the whole world suffered from COVID-19. Due to the unawareness of this new disease, it took a long time for most countries to understand the risk of the virus and realize the importance of active prevention. Suppose that we can collect the health data from the patients, such as heart rate, blood pressure and oxygen saturation. When we meet an unnamed infectious disease, we can estimate the number of suspected cases in an area based on the symptom. For other example, running an online shop, the marketing department needs to get a consumer profile for some stock-keeping units, and the courier needs a big picture of the delivery workload in their districts to balance the distribution of human power.

We take the courier data as a representative. In Alibaba's 11.11 shopping festival, Cainiao network [1] handled 2.32 billion delivery orders which handled 27k orders per second on average. A courier record can be formulated as $(T, From, To, w)$, where T denotes the time to ship, w denotes the weight of the parcel, $From = (From_x, From_y)$ and $To = (To_x, To_y)$ denote the latitude and longitude coordinates of the locations that we ship from and we ship to, respectively. An *area* is a 2-dimensional range in the form of $([From_{x\,min}, From_{x\,max}], [From_{y_{min}}, From_{y_{max}}])$. A location $(From_x, From_y)$ is in the area if $From_x \in [From_{x\,min}, From_{x\,max}]$ and $From_y \in [From_{y_{min}}, From_{y_{max}}]$. Figure 1 and Table 1 show an example. The courier may want to know the cumulative weights of parcels they shipped from 5:00 pm to 5:30 pm from Area A to Area B. $[5, 5.5]$ is the 1-dimensional query range of T to denote the time range from 5:00 pm to 5:30 pm. $([22.28, 22.29], [114.13, 114.14])$ is the 2-dimensional query range of $(From_x, From_y)$ with latitude and longitude to denote the Area A. $([22.31, 22.32], [114.16, 114.17])$ is the 2-dimensional query range of (To_x, To_y) to denote the Area B. Given a 5-dimensional query range r = ([5,5.5], [22.28, 22.29], [114.13, 114.14], [22.31, 22.32], [114.16, 114.17]) of $(T, From_x, From_y, To_x, To_y)$ and a sum aggregate function of w, we can obtain the aggregate result $4.5kg$, which is the total weight of parcels shipped from Area A to Area B in the time range. This kind of query is very common in real life and it may come from time to time, even when the data is still being updated. In this example, we can calculate the aggregate values by *semigroup orthogonal range searching*, which is defined in the following.

A *semigroup* is defined to be a set together with an aggregate function which takes inputs from this set. An element in the semigroup is defined to be a member in the set. In the above example, the parcel's weight w is in the real number set $\mathbb{R}$ and we are given a sum aggregate function $f(w_1, w_2) = w_1 + w_2$ for $\forall w_1, w_2 \in \mathbb{R}$. Therefore, the weight w is an element in a semigroup with a set $\mathbb{R}$ and a sum aggregate function. With the data of d-dimensional points, the orthogonal range searching is to find the points in a given d-dimensional rectangle. For example, we are given a 5-dimensional rectangular query range based on a time range, Area A and Area B in the above example. Semigroup orthogonal range searching is one of the important problems to discuss. In the problem, we are given a continuous input of high-dimensional points with their weights to store in a data structure where each weight is an element in a given semigroup. With the semigroup's aggregate function, our goal is to calculate the aggregate result of all the points located in a rectangular query range.

Studied by John C. McCallum [2], the average memory price is 6 USD per GB in 2017. Analyzed by Backblaze [3], the average disk price is 0.028 USD per GB in 2017. In a word, it is more than 200 times cheaper to store data on disk than in memory. Microsoft [31] is developing on-disk indexes to meet the needs for exabyte (10^6 TB) scale. In addition, Alibaba [25] has also developed an on-disk cloud database for real-time operations at PB (10^3 TB) scale. From 2009 to 2017, the disk price [3] decreased 75% from 0.11 USD to 0.028 USD while the memory price [2] decreased 50% from 12 USD to 6 USD. We observe that the disk will still be a cheaper device to store data in the near future. Based on this observation, we focus on how to solve this problem on disk. The number of I/Os costed by the solutions (I/O cost) is important for disks. To be more specific, we evaluate all the solutions based on write efficiency, query efficiency, and storage efficiency. The write efficiency is the average I/O cost of a single record insertion into a data structure. The query efficiency is the worst-case I/O cost in the data structure to answer a query. The space efficiency C is the number of blocks occupied by the data structure on disk.

There are a lot of researchers who have worked on this topic. In 1990, Chazalle [17] proved the tightest time complexity. He focused on a more fundamental problem which is called *dominance searching*. In this problem, their query is based on a dominant area of a query point instead of a query range. Chazalle proved that if C units of storage are available and N points are inserted, the query time is $\Omega((\log N/\log(2C/N))^{d-1})$ in the worst case, where d is the dimension of the points. However, this lower bound has its limits. It only works in *faithful semigroups*, which are semigroups with constraints defined in [17]. For example, if the aggregate function is $f(w_1, w_2) = (w_1 + w_2) \bmod 2$, the semigroup is not faithful. In addition, this lower bound is only provably tight when $C = \Omega(N \log^{d-1+\varepsilon} N)$ for any fixed $\varepsilon > 0$, which means it lacks space efficiency. In 2019, Afshani [9] proved that for any data structure that uses C units of storage, it has a query I/O bound $\Omega(N(\log N \log\log N)^{d-1}/C)$. When the storage $C = O(N)$, we have a chance to get a data structure with an $O((\log N \log\log N)^{d-1})$ query bound. This lower bound has two limits: The point distribution should be uniform, and the semigroup should be *idempotent*, which means for any element x in the semigroup, the aggregate function f will make $f(w, w) = w$ holds like $\mathrm{MAX}(w, w) = w$. In our study, we want a solution for all semigroups.

Although it is still an open problem that if we can find a linear-space data structure that can answer the semigroup orthogonal range searching problem in $O(polylog(N))$ I/Os, there is little hope of obtaining such a data structure. R-trees [21] and LSM R-trees [26] were proposed to support insertion and range query, but their query efficiency is not guaranteed. The Bkd-tree [30] was proposed to support fast insertion and the query I/O bound is $O((N/B)^{\frac{d-1}{d}} \log_2 \frac{N}{M})$, but its insertion I/O cost will increase fast when d comes large since it is $O(d^2 \log_M \frac{N}{B} \log_2 \frac{N}{M}/B)$, which means it lacks write efficiency.

As above mentioned, all the existing methods have weaknesses. The main weak point is that they cannot meet the modern requirements of write efficiency, query efficiency and space efficiency simultaneously. To tackle this problem, we propose a novel on-disk data structure named Write-Optimized Range-Query Tree (WORQ-tree). For high write efficiency, we adopt the same logarithmic method as the LSM-tree but a different inner design, which can help us obtain an $O(d \log_M \frac{N}{B} \log_2 \frac{N}{B}/B)$ write efficiency. For high query efficiency, we use a partition strategy and prove that by tuning up the partition scheme, we can obtain an $O((N/B)^{\frac{2d-3}{2d-1}} \log_B \frac{N}{B} \log_2 \frac{N}{M})$ query efficiency. Meanwhile, we also keep the $O(N)$ space cost, which means the data structure has high space efficiency. To support more intensive insertion, we propose a lazy-merge method based on our partition scheme. Given a lazy constant K, we can make the write efficiency at least $\log K$ times higher by sacrificing the query efficiency at most K times lower. By setting $K = \log\log N$, we can make the write efficiency become $O(d \log_M \frac{N}{B} \log_2 \frac{N}{B}/B/\log\log N)$ and the query efficiency $O((N/B)^{\frac{2d-3}{2d-1}} \log_B \frac{N}{B} \log_2 \frac{N}{M} \log_2 N)$.

The following shows our contributions:

- We are the first to study a write-optimized on-disk data structure for semigroup orthogonal range searching to analyze a huge amount of data while the previous studies in the problem were all focused on in-memory data structures.
- With $O(N)$ space usage, the amortized cost of insertion is $O(d \log_M \frac{N}{B} \log_2 \frac{N}{M}/B)$, which is better than other high-dimensional data structures consuming linear space such as Bkd-tree, LSM R-tree and PH-tree.
- We proved that by tuning up the partition scheme, the data structure can achieve $O((N/B)^{\frac{2d-3}{2d-1}} \log_B \frac{N}{B} \log_2 \frac{N}{M})$ query efficiency.
- We proposed a lazy-marge method based on the partition scheme with a lazy constant to accurately control the trade-off between write efficiency and query efficiency.
- We evaluate our data structure under various real-world data and synthetic data to show that our data structure is at least 5.5 times faster than other baselines on HDDs and 4.4 times faster on SSDs in heavy insertion tests.

The rest of the paper is organized as follows. In Section 2, we formally define the problem. Section 3 discusses the related work. In Section 4, we propose a novel and effective data stucture. Section 5 shows the experiments we conducted. Section 6 gives the conclusion.

2 PROBLEM DEFINITION

This section presents the problem of semigroup orthogonal range searching in a write-intensive database. Given a set S and an aggregate function $f : S \times S \to S$, $G = (S, f)$ is a *commutative semigroup* if and only if:

- $\forall w_1, w_2 \in S, f(w_1, w_2) \in S$;
- $\forall w_1, w_2 \in S, f(w_1, w_2) = f(w_2, w_1)$;
- $\forall w_1, w_2, w_3 \in S, f(w_1, f(w_2, w_3)) = f(f(w_1, w_2), w_3)$.

For example, let $\mathbb{Z}$ be the set of all integers and $\mathrm{MAX}(w_1, w_2)$ be the aggregate function to return the larger one in $\{w_1, w_2\}$, $G = (\mathbb{Z}, \mathrm{MAX})$ is a commutative semigroup, since $\forall w_1, w_2, w_3 \in \mathbb{Z}$, we have $\mathrm{MAX}(w_1, w_2) = \mathrm{MAX}(w_2, w_1) \in \mathbb{Z}$ and $\mathrm{MAX}(w_1, \mathrm{MAX}(w_2, w_3)) = \mathrm{MAX}(\mathrm{MAX}(w_1, w_2), w_3)$. The elements in the semigroup G is defined to the members in the set S. Since $f(w_1, f(w_2, w_3)) = f(f(w_1, w_2), w_3)$, we simply write it as $f(w_1, w_2, w_3)$. Similarly, $f(w_1, \cdots, w_l)$ is defined to the aggregate results of the l elements in the semigroup.

Given a commutative semigroup $G = (S, f)$ and N points $(x_1, x_2, \cdots, x_{d-1}, x_d)$ in $\mathbb{R}^d$ where each point has a weight $w \in S$, we need to build a data structure on disk to store the points and weights, where every *block* contains B points. Given a query range $r = (I_1, I_2, \cdots, I_{d-1}, I_d)$ where $I_i = [I_i^l, I_i^r]$ is a closed interval in $\mathbb{R}$, if for $\forall i \in [1, d]$, we have $x_i \in I_i$ i.e. $I_i^l \le x_i \le I_i^r$ for a point, then the point is in this query range. We need to obtain the aggregate value of all the points in the query range. The memory used to calculate this answer can only store M blocks. Here we denote our database as a set $D = \{(p, w(p)) \mid (p, w(p)) \text{ is inserted}\}$ where p is a d-dimensional point and $w(p)$ is an element in S. For a subset $Q = \{q_1, q_2, \cdots, q_l\} \subset D$ where $q_i = (p_i, w(p_i)) \in D$ for $\forall i \in [1, l]$, we simply write $f(w(p_1), w(p_2), \cdots, w(p_l))$ as $f(Q)$ and call it the aggregate value of Q. There are two operations for this database D:

(1) $Insert(p, w(p))$, where p is a d-dimensional point $(p.x_1, p.x_2, \cdots, p.x_d)$ which is the *key* of this record and $w(p)$ is an element in G. When the operation is done, this key-value pair will be stored in the database so that we have $(p, w(p)) \in D$.
(2) $Query(r)$, where r is a d-dimensional range $(I_1, I_2, \cdots, I_{d-1}, I_d)$. There exists a subset $Q \subset D$, where $\forall (p, w(p)) \in Q$, it holds that $x_i \in I_i$ for $\forall i \in [1, d]$ and it does not hold for $\forall (p, w(p)) \in D \backslash Q$. Then, the answer to this query is $f(Q)$.

We define the I/O cost as the number of times that the program reads or writes the disk, and introduce 3 aspects of our evaluation of the database:

- Write efficiency: The average I/O cost of a single record insertion into a data structure.
- Query efficiency: The worst-case I/O cost in the data structure to answer a query.
- Space efficiency C: The number of blocks occupied by the data structure in disk.

Note that in write efficiency, we consider the average case, but in query efficiency, we consider the worst case, which is consistent with [10, 12, 16, 23, 30, 37], because we focus on write-intensive workloads, which contain a lot more insertion than query. There is an intuition in real-world applications: When we have tons of data to insert, we only consider the cumulative time to store them. However, when we need to answer some queries, every query should be responded in a reasonable time. In addition, since we focus on write-intensive workloads, we only consider linear-space solutions i.e. $C = O(N)$, because when the number of records comes large, a super-linear-space data structure will make it hard to store them all on disk, and a sub-linear-space data structure cannot answer all queries with accurate results.

3 RELATED WORK

To study the problem in a commutative semigroup for one dimension and two dimensions, Andrew Yao [38] gave a result that any dynamic algorithm must take $\Omega(N \log N / \log \log N)$ time in the worst case in processing an $O(N)$ one-dimensional instruction sequence and there is a space-time trade-off $t = \Omega(\log N / \log(C \log N/N))$ for the two-dimensional static problem. Tao [34] gave a very substantial result in 2D range searching. However, it is not easy to extend this work to higher dimensional space and for insertion-intensive workload, since it used the 1D interval tree which lacks write efficiency. Different from his work, we focus on 3D or higher dimensions.

To study the orthogonal range searching problem for higher dimensions in the static case, Chazelle proposed the lower bounds for the d-dimensional cases: $O(K + polylog(N))$ query I/O cost with $\Omega(N(\log N/\log \log N)^{d-1}))$ storage cost in [16] and $\Omega((\log N/\log(2C/N))^{d-1})$ query I/O cost with C units storage cost in [17] where the lower bound of query I/O cost is provably tight for $C = \Omega(N(\log N)^{d-1+\varepsilon})$. More specifically, the first lower bound is for the orthogonal range reporting problem. Unlike semigroup orthogonal range searching, we need to report all the points in this problem instead of just returning the aggregate value. Chazelle [16] proved that any d-dimensional data structure with $O(\mathrm{polylog}(N))$ query efficiency should use $\Omega(N(\log N/\log \log N)^{d-1})$ space. However, since reporting is obviously more complicated than semigroup

Source	Write efficiency	Query efficiency	Space efficiency
k-d tree [15]	$O(d^2 \log_2 \frac{N}{B} \log_2 \frac{N}{M}/B)$	$O((N/B)^{\frac{d-1}{d}} \log_2 \frac{N}{M})$	$O(N)$
LSM R-tree [11]	$O(d \log_M \frac{N}{B} \log_2 \frac{N}{M}/B)$	$O(N/B)$	$O(N)$
PH tree [39]	$O(dw)$	$O(N/B)$	$O(N)$
Priority R-tree [12]	$O(d^2 \log_M \frac{N}{B} \log_2 \frac{N}{M}/B)$	$O((N/B)^{\frac{d-1}{d}} \log_2 \frac{N}{M})$	$O(N)$
Bkd tree [30]	$O(d^2 \log_M \frac{N}{B} \log_2 \frac{N}{M}/B)$	$O((N/B)^{\frac{d-1}{d}} \log_2 \frac{N}{M})$	$O(N)$
Ishiyama's method [22]	$O(d \lg N \log N + \lg N \lg \lg N \log N)$	$O(N^{\frac{d-2}{d}} \lg N/\lg \lg N)$	$O(dN \lg N + N \lg N \lg \lg N)$
Alstrup's method [10]	$O(\log^d N)$	$O(\log^{d-1} N)$	$O(N \log^{d-1} N)$
Chazelle's method [18]	$O(\log^d N)$	$O(\alpha(N) \log^{d-1} N)$	$O(N \log^{d-1} N)$
WORQ-tree without LM	$O(d \log_M \frac{N}{B} \log_2 \frac{N}{M}/B)$	$O((N/B)^{\frac{2d-3}{2d-1}} \log_B \frac{N}{B} \log_2 \frac{N}{M})$	$O(N)$
WORQ-tree with LM	$O(d \log_M \frac{N}{B} \log_2 \frac{N}{M}/B/\log \log N)$	$O((N/B)^{\frac{2d-3}{2d-1}} \log_B \frac{N}{B} \log_2 \frac{N}{M} \log_2 N)$	$O(N)$

Table 2: Bounds for the semigroup orthogonal range searching

searching, we still have no idea if semigroup orthogonal range searching can be answered with both $O(\text{polylog}(N))$ query efficiency and $O(N)$ space efficiency. The second lower bound is only provably tight when C is large enough. In addition, there is another limitation: It only works in idempotent semigroups. This lower bound is for the semigroup dominance searching problem. In this problem, we are given a query point and search one side of the query point in each dimension.

To better discuss the existing method, we list some typical existing methods and our new methods in Table 2 with their write efficiency, query efficiency and space efficiency. We choose some of the existing methods as the baselines in our experiment. In the following, we discuss the existing methods in Table 2 and show the reason of our choices.

Zäschke et al. proposed the PH-tree based on binary PATRICIA-tries combined with hypercubes. They combine each bit in all the dimensions to make the search key and store it in the linear or hypercube presentation. The insertion complexity is $O(dw)$ where w is the number of bits in a record. This data structure is focused on in-memory usage and considers the register width of the CPU. It does not have a good query bound for range query (i.e. $O(N)$) and the performance on disk is not that efficient. To study the efficiency on disk, we choose it as a baseline.

To study this problem with $O(N)$ storage space on disk, we have the widely-used R-tree [21] to index the keys in linear space. Researchers further studied the LSM-based R-tree [24]. Mao et al. [27] did a study of the LSM R-tree. They studied the three existing ways of organizing the data in an LSM R-tree, some of which can offer very good write efficiency, but none can support a provable query I/O guarantee. Alsubaiee et al. [11] shows how to "LSM-ify" the R-tree to an LSM R-tree. The LSM R-tree consists of in-memory and on-disk components. When the memory is full, the in-memory components will be flushed to the disk. To bulk-write an R-tree, they sort all the records by a space-filling curve comparator (e.g., Hilbert curve or Z-order curve) and pack them into tree-node blocks. The calculation of the space-filling curve comparator can be time-consuming and in the worst case, a range query will lead to very low locality and even need to traverse all the blocks. We take the LSM R-tree with the Hilbert curve comparator as a baseline.

To tackle the problem of slow query, Arge et al. [12] proposed the Priority R-tree, which basically use the same technique as the k-d tree to get the same worst-case guarantee as k-d trees i.e., $O((N/B)^{\frac{d-1}{d}})$. In LSM R-tree, each d-dimensional rectangles are stored as a $2d$-dimensional record in the same manner as the kd-tree. Another data structure which is widely used for this problem is k-d tree [15] itself. Different from the R-tree, the k-d tree offers an $O((N/B)^{\frac{d-1}{d}})$ range query guarantee. One good attempt to make k-d tree dynamic is made by Robinson [32]. This new data structure is called k-d-b tree. However, an insertion may cause several split operations on the path from the leaf to the root to maintain the query performance just like B-tree. Obviously, this split operation significantly affects the performance when the data structure is on disk. LSM-tree [28] proposed by O'Neil et al. introduced another way of balancing the trade-off of write efficiency and query efficiency. Instead of maintaining only one tree, an LSM-tree maintains a set of trees and reduces the I/Os to rebalance the tree after each insertion. Bkd-tree proposed by Procopiuc [30] et al. invoked this design. Unlike k-d-b tree, a set of k-d trees are maintained in a Bkd tree. Motivated by B-tree [14], he proposed an I/O efficient bulk-loading algorithm for k-d tree and used the logarithmic method to make the static data structure dynamic. However, its write efficiency deteriorates fast with a factor of d^2 when d comes large. We choose the Bkd-tree as a baseline since it is the best among the kd-tree-like data structures for this problem.

There are also studies such as [10, 18] discussing about the solutions with $O(\text{polylog(N)})$ query efficiency. Most of such studies used a static data structure to solve their offline problem, and all of their algorithms cost super-linear storage space, which makes them relatively far from our present real-world applications. Ishiyama [22] et al. studied a seldom-studied problem: succinct data structures for multi-dimensional orthogonal range searching. They proposed a data structure can answer the range counting queries in $O(N^{\frac{d-2}{d}} \lg N/\lg \lg N)$ time with $O(dN \lg N + N \lg N \lg \lg N)$ space cost, which requires less space partitioning than the k-d tree. Chazelle [18] proposed a RAM-based data structure of size $O(N \log^{d-1} N)$ which can answer any range query in $O(\alpha(N) \log^{d-1} N)$ time over a semigroup, where $\alpha(\cdot)$ is the functional inverse of Ackermann's function defined by Tarjan [36]. Alstrup [10] proposed a new general technique for static orthogonal range searching problems in higher dimensions. They showed a method to extend their d-dimensional data structure to support $(d+1)$-dimensional range

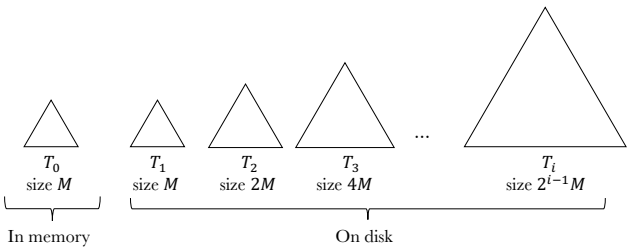


Figure 2: An Illustration of the LSM-Segment Tree

Algorithm 1: `Insert`(p, w)

Input: a point p with its weight w

```
1 insert (p, w) into T_0;                          // store it in memory
2 if memory is full then
3     T'_1 ← Seg-Merge(T_0, ∅, 1);                 // flush T_0 into T'_1
4     delete T_0; i ← 1;
5     while there exist both T_i and T'_i do
6         T'_{i+1} ← Seg-Merge(T_i, T'_i, 1); i ← i + 1;
7     T_i ← T'_i; delete T'_i;
```

reporting. This iterative method has $O(\log^{d-1} N)$ query efficiency and $O(N \log^{d-1} N)$ space efficiency. Since it is a static data structure, it needs $O(N \log^{d} N)$ pre-processing time. Since Alstrup's method has the best query efficiency among the super-linear-space methods, we choose it as a baseline.

4 DESIGN OF THE WORQ-TREE

In this section, we present this data structure step by step. In Section 4.1, we discuss the most uncomplicated case, where one-dimensional data is the only input. In Section 4.2, we present how to modify this data structure to work for two-dimensional input. In Section 4.3, we further extend our data structure to higher dimensions. In Section 4.4, we propose a lazy merge strategy to further improve the write efficiency. In Section 4.5, we discuss the deletion and concurrency control based on existing methods.

4.1 LSM-Segment Tree

Motivated by LSM-tree [28], we use an LSM-like structure to deal with the 1-dimensional cases. Different from the original LSM-tree, the LSM-segment tree consists of several segment trees [20]. To store these segment trees in the disk, we first sort all the points in key order and pack B points into one leaf block with its aggregate results on disk, where B is a parameter based on the disk and the file system. Then, we scan the sequence and pack B blocks into a new point in the upper level, iteratively.

Formally, a database on disk with N points consists of $\lceil \log_2 \frac{N}{M} \rceil$ segment trees as shown in Figure 2, which are $T_1, T_2, \cdot, T_{\lceil \log_2 \frac{N}{M} \rceil}$. T_i is a segment tree contains $M \cdot 2^{i-1}$ points. When the point is inserted, it will be first inserted into an in-memory tree T_0, and when the memory is full, the tree will be flushed into the disk to make a new T'_1. For each i, if there exists both T'_i and T_i, then T'_i and T_i should be merged into T'_{i+1}. Otherwise, we let T'_i be a new T_i. Algorithm 1 shows the procedure.

To merge T'_i and T_i, we first sort all $M \cdot 2^i$ points in key order on disk. We divide them into $M \cdot 2^i/B$ blocks where each block has at most B points. Then, we calculate the aggregate results for each block. The leaf level is constructed. Next, we do a bottom-up construction. We scan each level to construct the upper level. Each block in the upper level consists of an aggregate result of B blocks in the lower level and B indexes to them. We do the step iteratively until a level created has one block. Algorithm 2 shows the procedure and Figure 3 shows an example. We insert the records in Table 1. In the first step, we sort all the points by the d-th dimension. Since the points are all 1-dimensional, we have $d = 1$. In the second step, we construct the second level of the tree. In the second level, the key is a segment showing the range of B blocks in the lower level and the value is the aggregate value of the B blocks. In the third step, we construct the third level in the same way. Since there is only one block in the highest level, the algorithm stops.

Algorithm 2: `Seg-Merge`(T^1, T^2, d)

Input: two segment trees T^1 and T^2 to sort by the d-th dimension

Output: a new tree's root

```
1 sort all the N points in T^1 and T^2 by the d-th dimension to
    get a queue Q = (p_1, p_2, ··· , p_N) by x_d;
2 while Q.size > 1 do
3     (p_1, p_2, ··· , p_B) ← pop out first B points in Q;
4     p ← the disk address of (p_1, p_2, ··· , p_B);
5     w(p) ← f(p_1, p_2, ··· , p_B); Q.push(p);
6 return Q.head
```

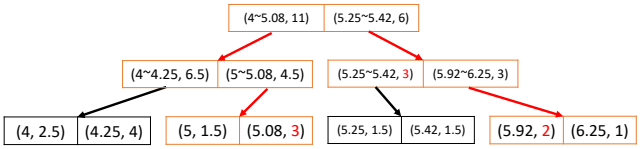


Figure 3: Seg-Query in the range $[5.04, 5.96]$: The arrow in red marks the traverse of the algorithm. We need to access the blocks in red and fetch the aggregate values.

Note that the keys are sorted in each level of the segment trees, a merge sort costs $O(M \cdot 2^i/B)$ I/Os. In addition, the number of blocks in the upper levels should be smaller than the number of blocks in the leaf level, so the upper-level construction costs $O(M \cdot 2^i/B)$ I/Os. The merge cost is $O(M \cdot 2^i/B)$. Hence, the amortized cost of an insertion is $O(\log_2 \frac{N}{M}/B)$.

As shown in Figure 3 , to answer the query, we do a top-down search on the segment tree. The smallest range such that all the points in a block are in it is called the block range of this block. If all the points in a block range are also in the query range, then the two ranges are fully overlapped. If some of the points in a block range are also in the query range and these two ranges are not fully overlapped, then the two ranges are partially overlapped. In each block, we can check how the query range overlaps with the block range. If the query range fully overlaps with the block range, we simply return the aggregate result stored in the block. If the query range overlaps with the block range only partially, we recursively search all the children blocks which overlap with the query range, and aggregate the results returned. Algorithm 3 shows the procedure.

Since the keys make a segment on disk, there is an observation that we access at most two blocks in each level in the query. The

Algorithm 3: `Seg-Query(r, p)`

Input: a 1-dimensional range r and a point p in the LSM-segment tree
Output: the aggregate value

```
1 (p_1, p_2, ·, p_B) ← the children of p;
2 w ← 0;                                   // 0 is the identity in S
3 for i ← 1 to B do
4     if p_i is fully overlapped with r then
5         w ← f(w, w(p_i));
6     else if p_i is partially overlapped with r then
7         w ← f(w, Seg-Query(r, p_i));
8 return w
```

Algorithm 4: `Part-Merge(T^1, T^2)`

Input: two segment trees T^1 and T^2
Output: a new tree's root

```
1  sort all the N points in T^1 and T^2 to get (p_1, p_2, ··· , p_N) by
     the first dimension;
2  T ← ∅; l ← 1; r ← ⌈(N/B)^{2/3}⌉ · B; i ← 1;
3  while l ≤ N do
4      T[i] ← (p_l, ··· , p_r);
5      T[i].seg ← Seg-Merge((p_l, ··· , p_r),∅,2);
6      l' ← l; r' ← l + ⌈(N/B)^{1/3}⌉ · B; j ← 1;
7      while l' ≤ N do
8          T[i][j] ← (p_{l'}, ··· , p_{r'});
9          T[i][j].seg ← Seg-Merge((p_{l'}, ··· , p_{r'}),∅,1);
10         l' ← l' + ⌈(N/B)^{1/3}⌉ · B; r' ← r' + ⌈(N/B)^{1/3}⌉ · B; j
             ← j + 1;
11     l ← l + ⌈(N/B)^{2/3}⌉ · B; r ← r + ⌈(N/B)^{2/3}⌉ · B; i ← i + 1;
12 return T
```

reason is that the query range will partially overlap with at most two blocks as the two ends on a segment. To answer the query in T_i, it costs $O(\log_B(M \cdot 2^i/B))$ I/Os. The total cost to obtain the whole aggregate result is $O(\log_2 \frac{N}{M} \log_B \frac{N}{B})$.

4.2 LSM-Partition-Segment Tree

When it comes to 2-dimensional cases, we cannot simply use the same methods as above since it is not easy to sort the 2-dimensional keys in order. Motivated by Bkd-tree [30], we use the grid method to deal with the problem. For convenience, we write the LSM-segment tree with sorting by the d-th dimension as d-th-dimension segment tree. Assume that we have N points to build a tree. We first divide them into $\lceil (N/B)^{1/3} \rceil$ strips by the first dimension. In each strip, we construct a second-dimension segment tree in the same way as mentioned above. Then, we divide each strip into $\lceil (N/B)^{1/3} \rceil$ grid by the second dimension. In each grid, we construct a first-dimension segment tree. Algorithm 4 shows the procedure. Line 9 is to build a first-dimension segment tree and Line 5 is to build a second-dimension segment tree.

Note that different from the 1-dimensional cases, the points are not sorted here, so the construction of the segment tree is not $O(N)$. The cost depends on the division of the points, where the external sorting costs $O(\frac{N}{B} \log_M \frac{N}{B})$. Hence, when there are $\lceil \log_2 \frac{N}{M} \rceil$ trees, the amortized cost of insertion is $O(\log_M \frac{N}{B} \log_2 \frac{N}{M}/B)$.

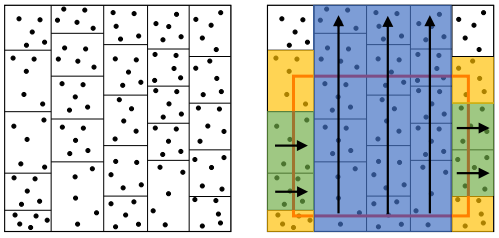


Figure 4: An example of Algorithm 5: The fully overlapped strips are in the blue area. The fully overlapped grids in the partially overlapped strips are in the green area. The partially overlapped grids in the partially overlapped strips are in the yellow area.

To answer the query, we first find the strips which overlap with the query range. There are two kinds of strips: strips that are fully overlapped with query range in the first-dimension and partially overlapped. For the at most $\lceil (N/B)^{1/3} \rceil$ fully overlapped ones, search the second-dimension segment trees, which cost at most $O((N/B)^{\frac{1}{3}} \log_B \frac{N}{B})$ I/Os. For the at most 2 partially overlapped ones, we then use find the grids which overlap with the query range. There are two kinds of grids: girds fully overlapped with the query range in the second-dimension and partially overlapped ones. For the at most $\lceil (N/B)^{1/3} \rceil$ fully overlapped ones, search the first-dimension segment trees, which cost at most $O((N/B)^{\frac{1}{3}} \log_B \frac{N}{B})$ I/Os. For at most 2 partially overlapped ones, scan the whole grid the obtain the answer, which cost $O((N/B)^{\frac{1}{3}})$ I/Os. Algorithm 5 shows the procedure.

For example, we append more data records to Table 1 to make $N = 125$, then the strips and grids should be constructed as shown in Figure 4. We divide all the points into 5 strips and divide each strip into 5 grids. To answer a range query in the red rectangle, we divide the query range into 3 areas and calculate the aggregate value in different ways:

(1) In the blue area, search the 2^{rd}-dimension segment trees.
(2) In the green area, search the 1^{st}-dimension segment trees.
(3) In the yellow area, scan the whole grids.

For a summary, it costs $O((N/B)^{\frac{1}{3}} \log_B N)$ I/Os to query in a partitioned segment tree. Hence, considering there are $\lceil \log_2 \frac{N}{M} \rceil$ trees, the cost of query is $O((N/B)^{\frac{1}{3}} \log_B N \log_2 \frac{N}{M})$.

4.3 WORQ-Tree

When it comes to high-dimensional cases, things will become more difficult. Given a d-dimensional dataset, we propose a new method to divide and conquer the problem. Assume that we have N points to build a tree, we first divide them into $\lceil (N/B)^{\frac{2}{2d-1}} \rceil$ strips by the first dimension, then build the $(d-1)$-dimensional trees in the strips in the same way recursively, until we meet $d < 3$. When $d < 3$, we use the same method as we proposed in Section 4.2 to build the tree.

Given that there are d levels of construction and in each level the time complexity depends on the division, the construction costs $O(d\frac{N}{B} \log_M \frac{N}{B})$ I/Os. Hence, considering there are $\lceil \log_2 \frac{N}{M} \rceil$ trees,

Algorithm 5: `Part-Query(r, T)`

Input: a 2-dimensional range r and the LSM-partition-segment tree T
Output: the aggregate value

$w \leftarrow 0$;
for $i \leftarrow 1$ *to* $\lceil (N/B)^{\frac{1}{3}} \rceil$ **do**
  if $T[i]$ *is fully overlapped with* r **then**
    $w \leftarrow f(w,$ `Seg-Query(r, T[i].seg)`$)$;
  else if $T[i]$ *is partially overlapped with* r **then**
    for $j \leftarrow 1$ *to* $(N/B)^{\frac{1}{3}}$ **do**
      if $T[i][j]$ *is fully overlapped with* r **then**
        $w \leftarrow f(w,$ `Seg-Query(r, T[i][j].seg)`$)$;
      else if $T[i][j]$ *is partially overlapped with* r **then**
        foreach $p \in T[i][j]$ **do**
          if $p \in r$ **then**
            $w \leftarrow f(w, w(p))$;
return w

Algorithm 6: `WORQ-Merge(`T^1, T^2,d`)`

Input: two segment trees T^1 and T^2 and the dimension d
Output: a new tree's root

if $d \leq 2$ **then**
  return `Part-Merge(`T^1,T^2`)`;
else
  sort all the N points in T^1 and T^2 to get $(p_1, p_2, \cdots, p_N)$ by the d-th dimension;
  $T \leftarrow \emptyset$; $l \leftarrow 1$; $r \leftarrow \lceil (N/B)^{\frac{2d-3}{2d-1}} \rceil \cdot B$; $i \leftarrow 1$;
  while $l \leq N$ **do**
    $T[i] \leftarrow (p_l, \cdots, p_r)$;
    $T[i].worq \leftarrow$ `Seg-Merge(`$(p_l, \cdots, p_r)$,$\emptyset$,$d-1$`)`;
    $l \leftarrow l + \lceil (N/B)^{\frac{2d-3}{2d-1}} \rceil \cdot B$; $r \leftarrow r + \lceil (N/B)^{\frac{2d-3}{2d-1}} \rceil \cdot B$; $i \leftarrow i+1$;
  return T

the amortized cost of insertion is $O(d \log_M \frac{N}{B} \log_2 \frac{N}{M}/B)$. Algorithm 6 shows the procedure.

To answer the query, we first find the strips which overlap with the query range in the first dimension. We scan the whole strip for the strips partially overlapped with the query range to obtain the answer. For the strips fully overlapped with the query range, we further search the results in the other $(d-1)$ dimensions. Algorithm 7 shows the procedure. Figure 5 shows an example of Algorithm 7. The red rectangles are the query range in the corresponding dimensions. We have a fully overlapped strip marked in orange and a partially overlapped strip marked in yellow in the first dimension. We need to scan the yellow strip and then search down from the orange one. In the second dimension, we meet the same case, and we scan the yellow strip again and search down from the orange strip. Since there are two remaining dimensions, the orange strip is actually an LSM-Partition-Segment Tree, so we use Algorithm 5 to obtain the aggregate value. To discuss the read efficiency and write efficiency, we give the following Theorem 10 and Theorem 2. The detailed proofs are in the technical report.

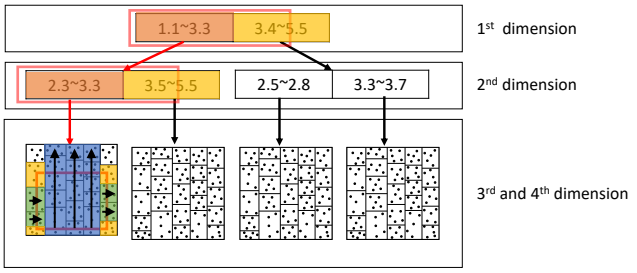


Figure 5: An example of the WORQ-tree: The red rectangles show the query range in each dimension. The yellow areas in the first and second dimensions are the partially overlapped strips in its dimension. The orange areas in the first and second dimensions are the fully overlapped strips in its dimension.

Algorithm 7: `WORQ-Query(r, T, d)`

Input: a d-dimensional range r, the WORQ-tree T and the dimension d
Output: the aggregate value

if $d \leq 2$ **then**
  return `Part-Query(r, T)`;
else
  $w \leftarrow 0$;
  for $i \leftarrow 1$ *to* $\lceil (N/B)^{\frac{2}{2d-1}} \rceil$ **do**
    if $T[i]$ *is partially overlapped with* r **then**
      $w \leftarrow f(w,$ `WORQ-Query(r, T[i].worq, d-1)`$)$;
    else if $T[i]$ *is fully overlapped with* r **then**
      foreach $p \in T[i]$ **do**
        if $p \in r$ **then**
          $w \leftarrow f(w, w(p))$;
  return w

Theorem 1. *The amortized insertion I/O cost of WORQ-tree is* $O(d \log_M \frac{N}{B} \log_2 \frac{N}{M}/B)$.

Proof sketch. We show that the amortized cost of Algorithm 2 is $O(\log_2 \frac{N}{M}/B)$, and then the amortized cost of Algorithm 4 is $O(\log_M \frac{N}{B} \log_2 \frac{N}{M}/B)$. Based on these two amortized costs, we can calculate that the amortized cost of Algorithm 6 is $O(d \log_M \frac{N}{B} \log_2 \frac{N}{M}/B)$. □

Theorem 2. *The worst-case query I/O cost of WORQ-tree is* $O((N/B)^{\frac{2d-3}{2d-1}} \log_B \frac{N}{B} \log_2 \frac{N}{M})$.

Proof sketch. We show that the complexity of Algorithm 3 is $O(\log_B \frac{N}{B})$, and then the complexity of Algorithm 5 is $O((N/B)^{\frac{1}{3}} \log_B N)$. By induction, we can derive that the complexity of Algorithm **??** is $O((N/B)^{\frac{2d-3}{2d-1}} \log_B \frac{N}{B})$. Since there are $\lceil \log_2 \frac{N}{M} \rceil$ trees, the I/O cost of query is $O((N/B)^{\frac{2d-3}{2d-1}} \log_B \frac{N}{B} \log_2 \frac{N}{M})$. □

4.4 Lazy Merge

Although the write efficiency for d-dimension is now $O(d \log_M \frac{N}{B} \log_2 \frac{N}{M}/B)$, we want to further improve it to support more write-intensive workloads by re-balancing the trade-off between the write

Algorithm 8: `Lazy-Merge`(T^1, T^2, d, K)

Input: two segment trees T^1 and T^2 of size N, the dimension d and the lazy constant K
Output: a new tree's root

if $d = 1$ **then**
  return `Seg-Merge`(T^1, T^2, d)
else
  sort all the strips $T^1[\cdot]$ and $T^2[\cdot]$ to get a $T[\cdot]$
  $i \leftarrow 1$;
  while $T[\cdot].size > \lceil (2N/B)^{\frac{2}{2d-1}} \rceil$ **do**
    while $T[i].size > \lceil 2(N/B)^{\frac{2d-3}{2d-1}} \rceil \cdot B$ **do**
      $i \leftarrow i + 1$;
    $j \leftarrow i + 1$;
    while $T[j].size > \lceil 2(N/B)^{\frac{2d-3}{2d-1}} \rceil \cdot B$ **do**
      $j \leftarrow j + 1$;
    $T[j] \leftarrow$ `Lazy-Merge`$(T[i], T[j], d\text{-}1, K)$;
    delete $T[i]$; $i \leftarrow j + 1$;
  foreach $T[a_1], T[a_2], \cdots, T[a_k]$ *overlap at a point* **do**
    if $k \leq K$ **then**
      for $i \leftarrow 2$ *to* k **do**
        $T[a_1] \leftarrow$ `WORQ-Merge`$(T[a_1], T[a_i], d)$;
        delete $T[a_i]$;
  return T

efficiency and query efficiency. The most commonly-used merge operation for high-dimensional LSM-tree is the same as the one for the Bkd-tree: When we need to merge two trees T^1 and T^2, we just put all the points in the two trees together and rebuild the whole tree based on the data. The existing methods do not take advantage of T^1 and T^2 balanced before, which cause more I/O cost of insertion. To tackle this problem, we propose a new merge algorithm named lazy merge (LM) to reduce the amortized I/O cost of merging on SSD. Intuitively, we want to take the advantage of the existing data structures by maintaining some of the sub-structures and postponing the rebuild operation. This method will significantly reduce the sequential I/Os but add a small amount of random I/Os. Since the random I/O performance of SSDs is much better than HDDs, WORQ-tree with LM becomes able to support higher-intensive workloads on SSD.

The main idea is to merge two strips into one strip instead of rebuilding all $\lceil (N/B)^{\frac{2}{2d-1}} \rceil$ strips after sorting. Assume that we need to merge two trees T^1 and T^2 each of size N. Since there are $2N$ points, there should be $\lceil (2N/B)^{\frac{2}{2d-1}} \rceil$ strips in the second level. In the previous method, we simply sort all points to divide them into $\lceil (2N/B)^{\frac{2}{2d-1}} \rceil$ strips. However, with lazy merge, we first list all the $2\lceil (N/B)^{\frac{2}{2d-1}} \rceil$ strips from the second level of T^1 and T^2, then merge the closest two strips into one strip iteratively until there are at most $\lceil (2N/B)^{\frac{2}{2d-1}} \rceil$ strips. We do the same lazy merge to merge strips, but there is a constraint: If the size of the strip is larger than $2\lceil (N/B)^{\frac{2d-3}{2d-1}} \rceil \cdot B$, then it cannot be merged into another strip. The lazy merge takes only $O(dN)$ I/O costs, because in each level, it only needs $O(N/B)$ I/Os to scan the strips and merge them.

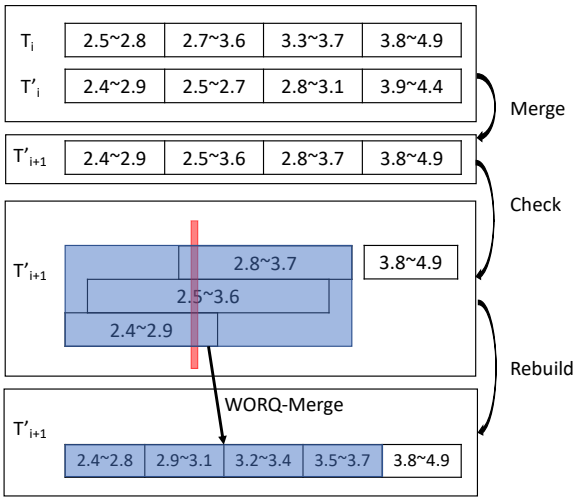


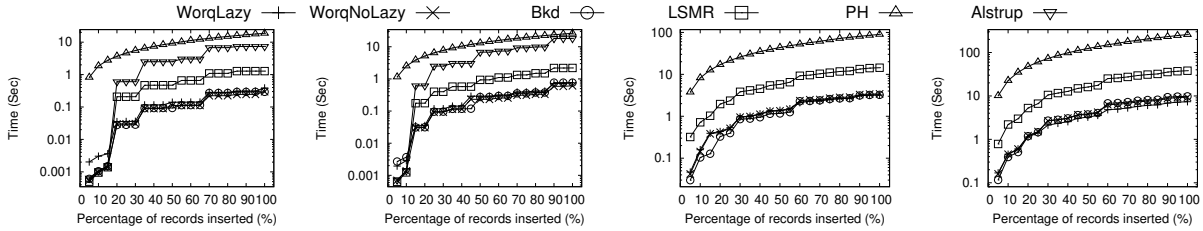
Figure 6: The steps of lazy merge

However, in the previous method, we need a sorting operation here and it costs $O(\frac{N}{B} \log_M \frac{N}{B})$ I/Os. Algorithm 8 shows the procedure.

However, if we get rid of all the sorting, the query efficiency cannot be guaranteed. Note that in the previous description in Section 4.3, all the strips should not be overlapped, but after a lazy merge operation, two strips may be overlapped. After k times of lazy merge, at most 2^k strips may be overlapped at a point. If 2^k strips are overlapped at the starting point of the query range, the query I/O cost will become roughly 2^k larger. Since k can be at most $\lceil \log_2 \frac{N}{M} \rceil$, the query efficiency will become unacceptable. To tackle this problem, we set a lazy constant K for a lazy merge. When the lazy merge in a level is done, we scan all the strips and check the overlapped strips. If more than K strips are overlapped at a point, we use Algorithm 6 to rebuild all the strips overlapped at this point. Figure 6 shows the steps of lazy merge. After the first merge, we have 3 strips overlapped at one point. If the lazy constant $K = 2$, then a rebuild operation is needed. We use Algorithm 6 to rebuild the 3 strips so that no strip overlapped at this point. Instead of rebuilding all the strips in WORQ-tree without lazy merge, we reduce a lot of I/O costs. Note that a point will meet a rebuild operation after at most $\lceil \log_2 K \rceil$ times of lazy merge, and once after a rebuild operation, there will be no overlapped strips at this point. We make the write efficiency at least $\log_2 K$ times faster, which is $O(d \log_M \frac{N}{B} \log_2 \frac{N}{M} / B / \log_2 K)$. In addition, in each level of the WORQ-tree, there should be at most K strips overlapped at a point. We make the query efficiency at most K times slower, which is $O(K(N/B)^{\frac{2d-3}{2d-1}} \log_B \frac{N}{B} \log_2 \frac{N}{M})$.

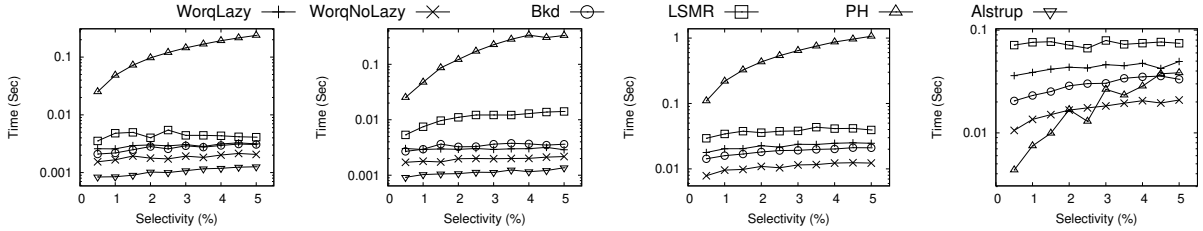
4.5 Deletion and Concurrency Control

To support deletion, we can perform a point query before the point deletion to get the address. Then, we can update the aggregate value from the bottom to the top in the WORQ-tree. This method will cost $O(\log_2 \frac{N}{M} (\log_B N + d))$ I/Os. However, we can use an update memo structure to accelerate this process if G is a commutative group. We build an update memo structure in memory, which keeps the records of deletions and updates into the WORQ-tree. This method was also studied in [33]. It can significantly reduce the deletion cost.



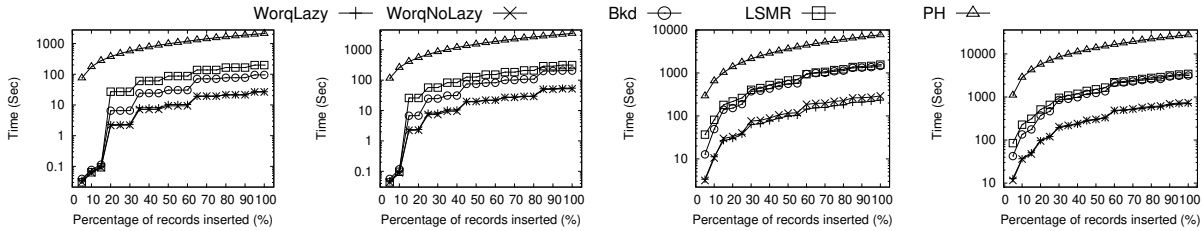
(a) Gowalla-small (b) London-small (c) Chicago-small (d) Berlin-small

Figure 7: Insertion time for small-datasets on SSD



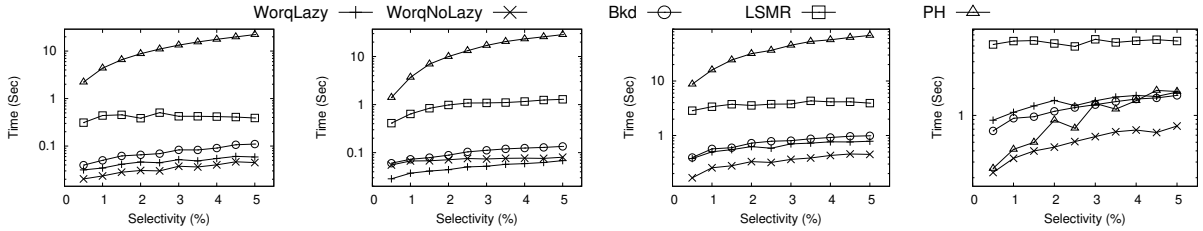
(a) Gowalla-small (b) London-small (c) Chicago-small (d) Berlin-small

Figure 8: Query time for small-datasets on SSD



(a) Gowalla (b) London (c) Chicago (d) Berlin

Figure 9: Insertion time for original-datasets on SSD



(a) Gowalla (b) London (c) Chicago (d) Berlin

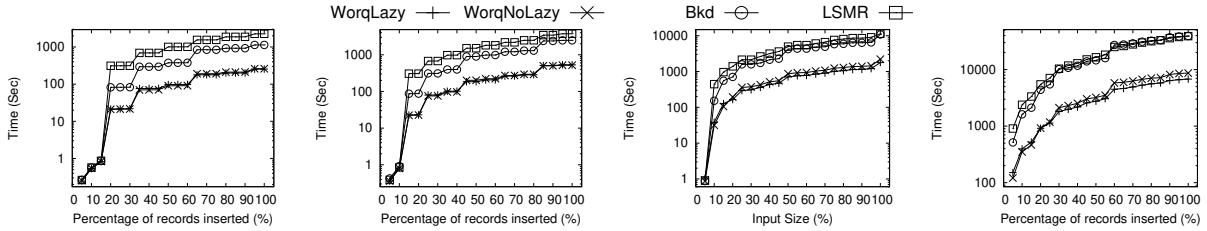
Figure 10: Query time for original-datasets on SSD

Besides, all the above discussion is based on one-thread. We can also extend this method for support of concurrent reads and writes. For the on-disk part, we can use a multi-version scheme to ensure the correctness. Since the data will not change the position in a WORQ-tree, we can keep an old version while merging. And also in merging, we can erase the outdated data easily. For the in-memory part, write-ahead logging can be used to keep the durability of the data. Since our methods are immutable similar to [28, 30, 33], dirty reads can be avoided.

5 EXPERIMENT

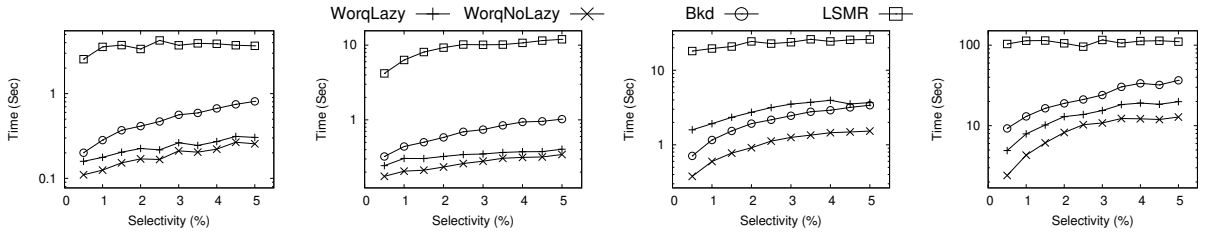
The experiments were conducted on a machine running Mac OS 11.1 on Intel Core i7 with 2.6 GHz, 32 GB memory, and 512 GB SSD. According to [13, 19, 33], without loss of generality, we choose four high-dimensional datasets on Stanford Network Analysis Project (SNAP) [4], Fernuni-Hagen [5], City of Chicago [6] and RMIT University [7] with different scales of insertions and dimensions.

(1) **Gowalla**: A sequence of check-ins of users in a location-based social website over the period from Feb. 2009 to Oct. 2010.
(2) **BerlinMOD**: This dataset consists of simulated car trips in the German capital Berlin.



(a) Gowalla-large (b) London-large (c) Chicago-large (d) Berlin-large

Figure 11: Insertion time for large-datasets on SSD



(a) Gowalla-large (b) London-large (c) Chicago-large (d) Berlin-large

Figure 12: Query time for large-datasets on SSD

Dataset	Insertions	Dimension
Chicagotaxi	28.7M	5
Gowalla	6.4M	3
LondonCourier	9.9M	3
BerlinMOD	56.1M	5
Chicagotaxi-small	0.3M	5
Gowalla-small	60K	3
LondonCourier-small	90K	3
BerlinMOD-small	0.6M	5
Chicagotaxi-large (real data)	167M	5
Gowalla-large	64M	3
LondonCourier-large	99M	3
BerlinMOD-large	561M	5

Table 3: Detailed information of chosen datasets

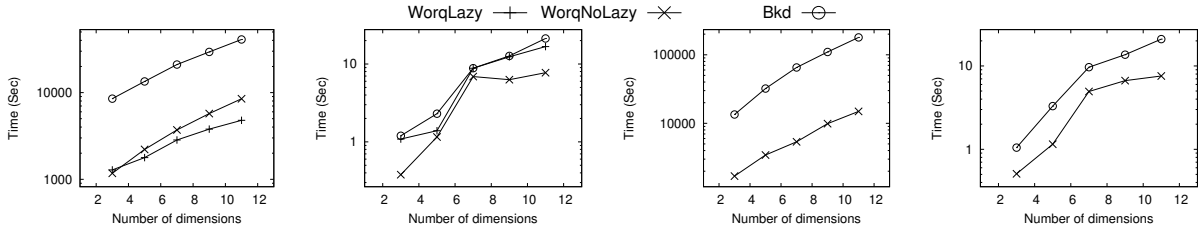
(3) **Chicagotaxi**: This dataset includes time and location information of taxi trips in Chicago.
(4) **LondonCourier**: This dataset is a collection of courier trajectories, captured principally around London over a continuous eight-week period.

Due to the lack of the corresponding orthogonal range query sets, we choose to randomly choose the query range in each dimension by uniform distribution. According to related studies [29, 35, 39], we set the selectivity of each query between 0.5% ∼ 5%. We study the average trip time for different queries in Chicagotaxi as a real-world application example. For each insertion for other datasets, we use COUNT() as the aggregate function. Note that there is no difference in the performance from other aggregate functions. Here, we call the original datasets original-datasets. We add 4 small-datasets and 4 large-datasets, since the original scale of the datasets are too large for some algorithms on HDD and too small for our algorithms on SSD. We randomly pick the records in the original-datasets, where the selectivity is 1%, to make the small-datasets. We repeat the same records 10 times to make the large-datasets. Note that the dataset Chicagotaxi-large is totally based on real-world data, and the other 3 large-datasets are based on the repeated real-world data. More information on these 12 datasets is listed in Table 3.

We evaluate the write efficiency, query efficiency of the WORQ-tree against four baseline data structures with different numbers of dimensions and different sizes of datasets. For write efficiency, we study how the running time increases with the increase of the percentage of records inserted from the dataset. The selectivity is the percentage of the records covered in the query range, and we study how the running time increases with the increase of the selectivity of the query range for query efficiency. Our data structures are WORQ-tree without lazy merge (*WorqNoLazy*) and WORQ-tree with lazy merge (*WorqLazy*), and the following is the other 4 baselines:

(1) **Bkd-tree** [30] (*Bkd*): A multi-dimensional data structure based on the LSM-tree and the kd-tree.
(2) **LSM R-tree** [11] (*LSMR*): A multi-dimensional data structure based on the LSM-tree and the R-tree. To bulk-load an R-tree, we take the commonly-used Hilbert curve as a space-filling curve comparator to sort the records and pack them into R-tree nodes.
(3) **PH-tree** [39] (*PH*): A multi-dimensional data structure based on the quadtree. Different from LSM R-tree using Hilbert value, it sorts the record by Z-order and takes the same multi-dimensional approach as quadtrees.
(4) **Alstrup** [10] (*Alstrup*): A static orthogonal range searching techniques proposed by Alstrup. Here we use the same logarithmic method in LSM-trees to make it dynamic.

The block size is set to 4KB for all the data structures. The lazy constant K for the WORQ-tree with lazy merge is set to $\log_2 \log_2 N$. The number of memory blocks used by the program is M. We set $M = 500$ for the small-datasets, $M = 10000$ for original-datasets and $M = 100000$ for large-datasets. The reason why we cannot let the program use all the 32GB ram is that the algorithms will find



(a) Insertion on SSD (b) Query on SSD (c) Insertion on HDD (d) Query on HDD

Figure 13: Efficiency with different numbers of dimensions

that all the data can be stored in the memory and will not write the data to the disk. Therefore we set M based on the dataset sizes and the number of dimension to better show the performance of the algorithms. All the coordinates for the records in the datasets are stored in Long Double, and all the weights and aggregate values are stored in Long Int. The results about deletion are shown in the technical report, since to support deletion will not change the ranks of all the algorithms. Similar to [28, 30, 33], due to the immutable nature of LSM-trees or variants, we do not do the experiment for the concurrency aspect, since they must return correct answers.

5.1 Efficiency on SSD

The results of write efficiency and query efficiency for small-datasets on SSD are shown in Figure 7 and Figure 8, respectively. In Figure 7(a) and Figure 7(b), we can find that Alstrup's data structure is far too much slower than other data structures in the insertion test, which is corresponding to its theoretical insertion time complexity $O(\log^{d+1} N/B)$. In addition, we find that due to its $O(N \log^{d-1} N)$ space complexity and the actual space efficiency shown in Table 4, we cannot even further test this data structure in the larger and higher-dimensional datasets. Except for Alstrup's data structure, we find that WORQ-tree with LM has the highest query efficiency, and the WORQ-tree without LM has the highest write efficiency. Since these datasets are very small-scale, the difference between the algorithms is not very significant.

The results of write efficiency and query efficiency for original-datasets on SSD are shown in Figure 9 and Figure 10, respectively. We find that Bkd-tree has the highest write efficiency and the highest query efficiency among the baselines. However, both WORQ-tree with LM and WORQ-tree without LM have higher write efficiency and higher query efficiency in average cases. In detail, WORQ-tree without LM shows 3.6×, 4.0×, 5.0× and 4.3× faster write efficiency than the other 4 baselines in Figure 9. Meanwhile, it also shows 2.5×, 1.7×, 2.2× and 2.2× faster query efficiency in Figure 10.

The large-datasets are the most large-scale datasets in the experiment, where the largest input data is about 31.4GB. The results of write efficiency and query efficiency for large-datasets on SSD are shown in Figure 11 and Figure 12. For these datasets, WORQ-tree without LM has faster write efficiency than Bkd-tree but slower than WORQ-tree with LM, and WORQ-tree with LM has faster query efficiency than BKd-tree but slower than WORQ-tree without LM. In detail, WORQ-tree without LM shows 4.4×, 4.8×, 5.8× and 4.5× faster write efficiency in Figure 11. Meanwhile, it also shows 3.2×, 3.0×, 2.2× and 2.9× faster query efficiency in Figure 12. In the largest dataset, WORQ-tree with LM can support insertions with 30% higher write efficiency than WORQ-tree without LM. Take Alibaba's 11.11 shopping festival [1] as an example. Cainiao network [8] handled 2.32 billion delivery orders which handled 27k orders per second on average. Under this workload, we estimate that the WORQ-tree will cost about $500GB$ storage space, which is reasonable with its substantially higher write efficiency. In the test with the largest dataset Berlin-large, WORQ-tree with LM inserts 82k records per second while any other baseline can insert at most 14k records per second, which means that the WORQ-tree has high availability for high-intensive workloads. Higher efficiency will help the couriers to reduce the delivery time as Cainiao network [8] has been steadily doing to meet future needs.

The space efficiency for all the datasets are shown in Table 4. The WORQ-tree with LM costs almost the same space as the WORQ-tree without LM. Both versions of the WORQ-trees show a linear growth of space usage with dataset size N and dimension d.

5.2 Efficiency on HDD

Besides SSD, we also conduct experiments on HDD to evaluate the efficiency and effectiveness of our data structures.

On HDD, WORQ-tree keeps showing a 6.0× and 5.5× write efficiency and 1.6× and 1.4× query efficiency. The detailed results and figures are in the technical report.

5.3 Efficiency with Different Number of Dimensions

We use the dataset Chichagotaxi-large to study the efficiency with different numbers of dimensions. We choose 3, 5, 7, 9, and 11 attributes in the data to further test the WORQ-tree and the Bkd-tree which has the best efficiency among the baselines. We evaluate the write efficiency and query efficiency both on SSD and HDD. 100 range queries for each number of dimensions are also generated with the selectivity of about 2.5% (2.35% $\sim$ 2.65%).

Figure 13 shows the results. Figure 13(a) shows the write efficiency on SSD. The advantage of lazy merge is very obvious, which can make the write efficiency up to 2.1× higher. The write efficiency of WORQ-tree is at least 4.8× faster than Bkd-tree by the number of dimensions varying from 3 to 11. Since the write efficiency of WORQ-tree is $O(d \log_M \frac{N}{B} \log_2 \frac{N}{M}/B)$ and the write efficiency of Bkd-tree is $O(d^2 \log_M \frac{N}{B} \log_2 \frac{N}{M}/B)$, it seems that WORQ-tree's insertion time grows slower than Bkd-tree with d increasing. Figure 13(b) shows the query efficiency on SSD. Except for the case of

Datasets	WorqNoLazy	WorqLazy	Bkd	LSMR	PH	Alstrup
Chicagotaxi-small	57M	61M	17M	18M	208M	N/A
Gowalla-small	8.5M	9.1M	2.3M	2.4M	44M	435M
LondonCourier-small	12M	13M	3.5M	3.6M	59M	638M
BerlinMOD-small	111M	116M	31M	32M	656M	N/A
Chicagotaxi	5.3G	5.3G	1.5G	1.5G	14G	N/A
Gowalla	825M	836M	233M	249M	4.3G	N/A
LondonCourier	1.2G	1.2G	352M	348M	5.8G	N/A
BerlinMOD	11G	10G	2.9G	3.0G	63G	N/A
Chicagotaxi-large	30G	33G	8.4G	8.4G	N/A	N/A
Gowalla-large	7.4G	7.4G	2.3G	2.3G	N/A	N/A
LondonCourier	11G	11G	3.4G	3.4G	N/A	N/A
BerlinMOD-large	101G	102G	29G	30G	N/A	N/A

Table 4: Storage space for different datasets

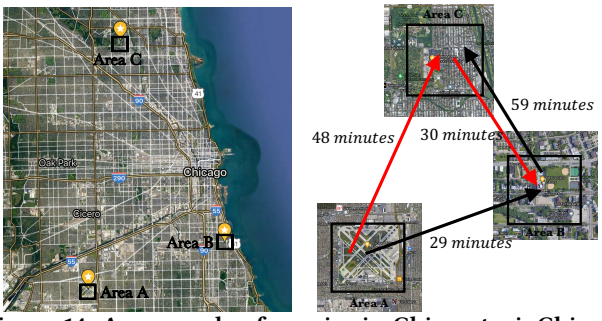


Figure 14: An example of queries in Chicagotaxi: Chicago Midway International Airport in Area A, NEIU CCICS in Area B and NEIU's main campus in Area C. Area A Latitude in [41.7724, 41.8047], Area A Longitude in [-87.7741, -87.7390], Area B Latitude in [41.9708, 42.0034], Area B Longitude in [-87.7300, -87.7007], Area C Latitude in [41.8002, 41.8362], and Area C Longitude in [-87.6248, -87.5902].

7 dimensions, the query efficiency of WORQ-tree is at least 2.1× faster than Bkd-tree. Figure 13(c) and Figure 13(d) shows the write efficiency and query efficiency on HDD. The results are consistent with SSD.

5.4 Case Study

We use Chicagotaxi as a real-world application example. Assume there is a truck driver who needs to transport goods from Chicago Midway International Airport (Area A) to NEIU CCICS (Area B) and NEIU's main campus (Area C) at 8 am. We study the average trip time for different solutions. Figure 14 shows the 3 areas in Chicago. In the map, we can observe that it is a much shorter journey from Area A to Area C via Area B than to Area B via Area C. The truck driver may choose to drive to CCICS first because of the shorter geographic distant. If we have a query of semigroup orthogonal range searching about the average trip time, we can know that it is 78 minutes from the airport to CCICS via the main campus but 88 minutes from the airport to the main campus via CCICS as shown in Figure 14. With this information, a courier can optimize its goods transport routes.

5.5 Summary

In this section, we show the numerical experimental results to discuss the write efficiency, query efficiency, and space efficiency of our data structures compared with 4 baselines. Alstrup's data structure shows the best query efficiency, but its $O(N \log^{d-1} N)$ space complexity makes it occupy 100× or even 1000× more space than other methods. In addition, it also has a very low write efficiency, which makes it unusable for large-scale datasets. PH-tree is not designed for on-disk usage, so it is not so competitive among the other baselines. LSM R-tree has a relatively high write efficiency, but it cannot support with a short query latency since it is by order of a space-filling curve, which is not always optimal for the query range. Bkd-tree has the highest write efficiency and query efficiency among the baselines because it is designed for write-intensive workloads and has a query I/O bound. However, Bkd-tree is not designed for the semigroup orthogonal range searching. Our data structures WORQ-tree outperformed all the baseline in the experiments. In addition, the version with lazy merge has the highest write efficiency (up to 110% higher than the version without lazy merge), and the version without lazy merge has the highest query efficiency, which offers more choices to make trade-offs between write efficiency and query efficiency. For insertions, our algorithm is faster than the Bkd-tree, LSM R-tree, PH-tree and Alstrup's method, up to 5.8×, 5.8×, 39× and 30×, respectively. For queries, our algorithm is faster than the Bkd-tree, LSM R-tree and PH-tree, up to 2.9×, 8.6× and 152.7×, respectively.

6 CONLUSION

In this paper, we studied how to solve the classical semigroup orthogonal range searching problem under a high-intensive workload. We propose a new data structure named WORQ-tree. Its write efficiency is $O(d \log_M \frac{N}{B} \log_2 \frac{N}{M}/B)$ and query efficiency is $O((N/B)^{\frac{2d-3}{2d-1}} \log_B \frac{N}{B} \log_2 \frac{N}{M})$, which is the best query I/O guarantee among all the $O(N)$-space multi-dimensional data structures. To further improve the write efficiency on SSD, we proposed a new lazy merge strategy, which adds tolerance of range overlap. We conducted experiments on 4 real-world datasets, and WORQ-tree outperforms all the baselines in the aspect of both write efficiency and write efficiency, which is at least 4.4 times faster on SSD in heavy insertion tests and at least 2.2 times faster on SSD in query tests. Furthermore, the lazy merge method can improve the write efficiency up to 110%.

REFERENCES

[1] [n.d.]. https://www.alizila.com/by-the-numbers-2020-11-11-global-shopping-festival/.

[2] [n.d.]. https://docs.google.com/spreadsheets/d/1qF5vJ2-jkRygMQQIk3fj6G-WEW0ScDiKjyGseovFGqc.

[3] [n.d.]. https://www.backblaze.com/blog/hard-drive-cost-per-gigabyte/.

[4] [n.d.]. http://snap.stanford.edu.

[5] [n.d.]. http://dna.fernuni-hagen.de.

[6] [n.d.]. https://data.cityofchicago.org.

[7] [n.d.]. http://www.rmit.edu.au/.

[8] [n.d.]. https://www.alizila.com/cainiao-sets-1111-delivery-speed-record/.

[9] Peyman Afshani. 2019. A new lower bound for semigroup orthogonal range searching. *arXiv preprint arXiv:1903.07967* (2019).

[10] Stephen Alstrup, G Stolting Brodal, and Theis Rauhe. 2000. New data structures for orthogonal range searching. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*. IEEE, 198–207.

[11] Sattam Alsubaiee, Alexander Behm, Vinayak Borkar, Zachary Heilbron, Young-Seok Kim, Michael J Carey, Markus Dreseler, and Chen Li. 2014. Storage management in AsterixDB. *Proceedings of the VLDB Endowment* 7, 10 (2014), 841–852.

[12] Lars Arge, Mark De Berg, Herman Haverkort, and Ke Yi. 2008. The priority R-tree: A practically efficient and worst-case optimal R-tree. *ACM Transactions on Algorithms (TALG)* 4, 1 (2008), 1–30.

[13] Mohamed S Bakli, Mahmoud A Sakr, and Taysir Hassan A Soliman. 2018. A spatiotemporal algebra in Hadoop for moving objects. *Geo-spatial information science* 21, 2 (2018), 102–114.

[14] Rudolf Bayer and Edward McCreight. 2002. Organization and maintenance of large ordered indexes. In *Software pioneers*. Springer, 245–262.

[15] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

[16] Bernard Chazelle. 1990. Lower bounds for orthogonal range searching: I. the reporting case. *Journal of the ACM (JACM)* 37, 2 (1990), 200–212.

[17] Bernard Chazelle. 1990. Lower bounds for orthogonal range searching: II. The arithmetic model. *Journal of the ACM (JACM)* 37, 3 (1990), 439–463.

[18] B. Chazelle and B. Rosenberg. 1989. Computing Partial Sums in Multidimensional Arrays. In *Proceedings of the Fifth Annual Symposium on Computational Geometry* (Saarbruchen, West Germany) *(SCG '89)*. Association for Computing Machinery, New York, NY, USA, 131–139. https://doi.org/10.1145/73833.73848

[19] Eunjoon Cho, Seth A Myers, and Jure Leskovec. 2011. Friendship and mobility: user movement in location-based social networks. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1082–1090.

[20] H. Edelsbrunner. 1980. *Dynamic Data Structures for Orthogonal Intersection Queries*. Inst. f. Informationsverarbeitung, TU Graz.

[21] Antonin Guttman. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. 47–57.

[22] Kazuki Ishiyama and Kunihiko Sadakane. 2017. A succinct data structure for multidimensional orthogonal range searching. In *2017 Data Compression Conference (DCC)*. IEEE, 270–279.

[23] Kazuki Ishiyama and Kunihiko Sadakane. 2020. Compact and succinct data structures for multidimensional orthogonal range searching. *Information and Computation* 273 (2020), 104519.

[24] Young-Seok Kim, Taewoo Kim, Michael J Carey, and Chen Li. 2017. A comparative study of log-structured merge-tree-based spatial indexes for Big Data. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 147–150.

[25] Feifei Li. 2019. Cloud-native database systems at Alibaba: Opportunities and challenges. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2263–2272.

[26] Chen Luo and Michael J. Carey. 2019. Efficient Data Ingestion and Query Processing for LSM-Based Storage Systems. *Proc. VLDB Endow.* 12, 5 (Jan. 2019), 531–543. https://doi.org/10.14778/3303753.3303759

[27] Qizhong Mao, Mohiuddin Abdul Qader, and Vagelis Hristidis. 2020. Comprehensive Comparison of LSM Architectures for Spatial Data. In *2020 IEEE International Conference on Big Data (Big Data)*. 455–460. https://doi.org/10.1109/BigData50022.2020.9377919

[28] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[29] Jinglin Peng, Dongxiang Zhang, Jiannan Wang, and Jian Pei. 2018. AQP++ Connecting Approximate Query Processing With Aggregate Precomputation for Interactive Analytics. In *Proceedings of the 2018 International Conference on Management of Data*. 1477–1492.

[30] Octavian Procopiuc, Pankaj K Agarwal, Lars Arge, and Jeffrey Scott Vitter. 2003. Bkd-tree: A dynamic scalable kd-tree. In *International Symposium on Spatial and Temporal Databases*. Springer, 46–65.

[31] Raghu Ramakrishnan, Baskar Sridharan, John R Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, et al. 2017. Azure data lake store: a hyperscale distributed file service for big data analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 51–63.

[32] John T Robinson. 1981. The KDB-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*. 10–18.

[33] Jaewoo Shin, Jianguo Wang, and Walid G Aref. 2021. The LSM RUM-Tree: A Log Structured Merge R-Tree for Update-intensive Spatial Workloads. In *2021 IEEE 37rd International Conference on Data Engineering (ICDE)*. IEEE.

[34] Yufei Tao. 2012. Indexability of 2d range search revisited: constant redundancy and weak indivisibility. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*. 131–142.

[35] Yufei Tao, Yi Yang, Xiaocheng Hu, Cheng Sheng, and Shuigeng Zhou. 2014. Instance-level worst-case query bounds on R-trees. *The VLDB journal* 23, 4 (2014), 591–607.

[36] Robert Endre Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM* 22, 2 (April 1975), 215–225. https://doi.org/10.1145/321879.321884

[37] Andrew C Yao. 1982. Space-time tradeoff for answering range queries. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*. 128–136.

[38] Andrew C Yao. 1985. On the complexity of maintaining partial sums. *SIAM J. Comput.* 14, 2 (1985), 277–288.

[39] Tilmann Zaschke, Christoph Zimmerli, and Moira C Norrie. 2014. The PH-tree: a space-efficient storage structure and multi-dimensional index. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 397–408.

TECHNICAL REPORT

In the technical report, we give detailed proofs of the complexity of our algorithms. And then, we show the result of the experiment on HDD.

Proof

We present Lemma 1 and Lemma 2 to discuss the write efficiency and query efficiency of the LSM-segment tree.

Lemma 1. *The amortized insertion I/O cost of LSM-Segment tree is* $O(\log_2 \frac{N}{M}/B)$.

Proof. We first calculate the I/O cost of merging two segment trees. To build a segment tree of $M \cdot 2^i$ leaf nodes, we need a merge sort of all the leaf nodes, which costs $O(M \cdot 2^i/B)$ I/Os. Starting from the leaf level, we perform a linear scan of all the nodes in a level to build the upper level, which means we need $O(M \cdot 2^i/B)$ I/Os to scan a level with $M \cdot 2^i$ nodes and create a new level with $M \cdot 2^i/B$ nodes. So, the total cost of the level construction is $O(M \cdot 2^i/B) + O(M \cdot 2^i/B^2) + O(M \cdot 2^i/B^3) \cdots = O(M \cdot 2^i/B)$. Therefore, to build a segment tree of $M \cdot 2^i$ leaf nodes costs $O(M \cdot 2^i/B)$ I/Os.

Next,the i-th segment tree with $M \cdot 2^{i-1}$ leaf nodes will need at most $N/(M \cdot 2^{i-1})$ times of merging, so it totally needs $O(M \cdot 2^{i-1}/B) \cdot N/(M \cdot 2^{i-1}) = O(N/B)$ IOs. There are $\log_2 \frac{N}{M}$ segment trees, so the total I/O cost is $\log_2 \frac{N}{M} \cdot O(N/B) = O(\log_2 \frac{N}{M} N/B)$. Hence, for each insertion, the amortized I/O cost is $O(\log_2 \frac{N}{M} N/B)/N = O(\log_2 \frac{N}{M}/B)$. □

Lemma 2. *The worst-case query I/O cost of LSM-Segment tree is* $O(\log_2 \frac{N}{M} \log_B \frac{N}{B})$.

Proof. We first calculate the I/O cost of query on one segment tree. We access at most two blocks in each level in the query. The reason is that the query range will partially overlap with at most two blocks as the two ends on a segment. Since a segment tree can only have $\log_B(N/B)$ levels, to answer the query in one segment tree costs $O(\log_B(N/B))$ I/Os.

Since there are $\log_2 \frac{N}{M}$ segment trees, the total cost to obtain the whole aggregate result is $O(\log_2 \frac{N}{M} \log_B \frac{N}{B})$. □

By Lemma 1 and Lemma 2, we know that the write efficiency of the LSM-Segment tree is $O(\log_2 \frac{N}{M}/B)$ and the query efficiency is $O(\log_2 \frac{N}{M} \log_B \frac{N}{B})$.

Next, we present Lemma 3 and Lemma 4 to discuss the write efficiency and query efficiency of the LSM-Partition-segment tree.

Lemma 3. *The amortized insertion I/O cost of LSM-Partition-Segment tree is* $O(\log_M \frac{N}{B} \log_2 \frac{N}{M}/B)$.

Proof. We first calculate the I/O cost of one Partition-Segment tree. To build a partition-segment tree, we need an external sorting by $1 - st$ dimension. The external sorting costs $O(\frac{N}{B} \log_M \frac{N}{B})$ I/Os. Then we need to sort each strip and build a segment tree for each strip, which totally costs $O(\frac{N}{B} \log_M \frac{N}{B})$ I/Os. Finally, we need to sort each grid and build a segment tree for each grid, which totally costs $O(\frac{N}{B} \log_M \frac{N}{B})$ I/Os. Therefore, it costs $O(\frac{N}{B} \log_M \frac{N}{B})$ I/Os to build a partition-segment tree.

Similar to Lemma 1, for each insertion, the amortized I/O cost is $O(\log_M \frac{N}{B} \log_2 \frac{N}{M}/B)$. □

Lemma 4. *The worst-case query I/O cost of LSM-Partition-Segment tree is* $O((N/B)^{\frac{1}{3}} \log_B N \log_2 \frac{N}{M})$.

Proof. We first calculate the I/O cost of query on one partition-segment tree. For the at most $\lceil (N/B)^{1/3} \rceil$ fully overlapped strips, search the second-dimension segment trees, which cost at most $O((N/B)^{\frac{1}{3}} \log_B \frac{N}{B})$ I/Os. For the at most 2 partially overlapped ones, there are two kinds of grids. For the at most $\lceil (N/B)^{1/3} \rceil$ fully overlapped ones, search the first-dimension segment trees, which cost at most $O((N/B)^{\frac{1}{3}} \log_B \frac{N}{B})$ I/Os. For at most 2 partially overlapped ones, scan the whole grid the obtain the answer, which cost $O((N/B)^{\frac{1}{3}})$ I/Os. Hence, the total query I/O cost is $O((N/B)^{\frac{1}{3}} \log_B \frac{N}{B}) + 2 \cdot O((N/B)^{\frac{1}{3}} \log_B \frac{N}{B}) + 4 \cdot O(N^{\frac{1}{3}}/B) = O((N/B)^{\frac{1}{3}} \log_B \frac{N}{B})$.

Since there are $\log_2 \frac{N}{M}$ partition-segment trees, the total cost to obtain the whole aggregate result is $O((N/B)^{\frac{1}{3}} \log_B N \log_2 \frac{N}{M})$. □

By Lemma 3 and Lemma 4, we know that the write efficiency of the LSM-Partition-Segment tree is $O(\log_M \frac{N}{B} \log_2 \frac{N}{M}/B)$ and the query efficiency is $O((N/B)^{\frac{1}{3}} \log_B N \log_2 \frac{N}{M})$.

Next, we present Theorem 1 and Theorem 2 to discuss the write efficiency and query efficiency of the WORQ-tree.

Theorem 1. *The amortized insertion I/O cost of WORQ-tree is* $O(d \log_M \frac{N}{B} \log_2 \frac{N}{M}/B)$.

Proof. Given that there are d levels of construction and in each level the construction costs $O(\frac{N}{B} \log_M \frac{N}{B})$ I/Os, it costs $O(d \frac{N}{B} \log_M \frac{N}{B})$ I/Os to build one tree in a WORQ-tree.

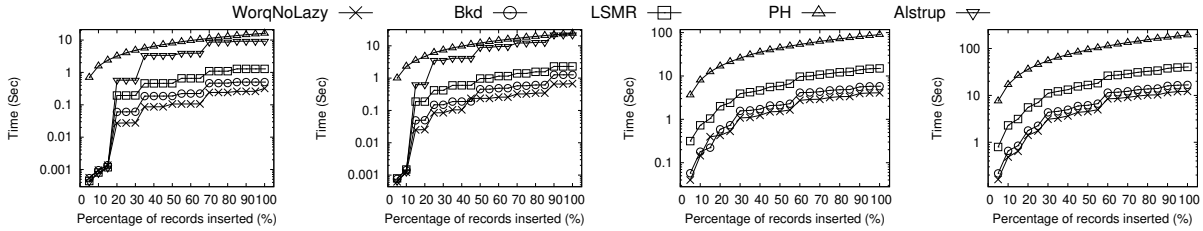
Hence, similar to Lemma 1, considering there are $\lceil \log_2 \frac{N}{M} \rceil$ trees, the amortized cost of insertion is $O(d \log_M \frac{N}{B} \log_2 \frac{N}{M}/B)$. □

Theorem 2. *The worst-case query I/O cost of WORQ-tree is* $O((N/B)^{\frac{2d-3}{2d-1}} \log_B \frac{N}{B} \log_2 \frac{N}{M})$.

Proof. We prove it by induction. Assume that the time complexity for d-dimension queries in one tree is $O((N/B)^{\frac{2d-3}{2d-1}} \log_B \frac{N}{B})$. When $d = 2$, it obviously holds. When $d > 2$, there are two kinds of strips: at most two strips partially overlapped with the query range in the d-th dimansion and at most $\lceil (N/B)^{\frac{2}{2d-1}} \rceil$ strips fully overlapped. For the at most two partially overlapped strips, it costs $O((N/B)^{\frac{2d-3}{2d-1}})$ I/Os to scan the whole strip to obtain the result. For the at most $\lceil (N/B)^{\frac{2}{2d-1}} \rceil$ fully overlapped strips, it costs $O(\left((N/B)^{\frac{2d-3}{2d-1}}\right)^{\frac{2d-5}{2d-3}} \log_B N) = O((N/B)^{\frac{2d-5}{2d-1}} \log_B \frac{N}{B})$ I/Os in each strip, so it totally cost $O((N/B)^{\frac{2d-3}{2d-1}} \log_B \frac{N}{B})$ I/Os.

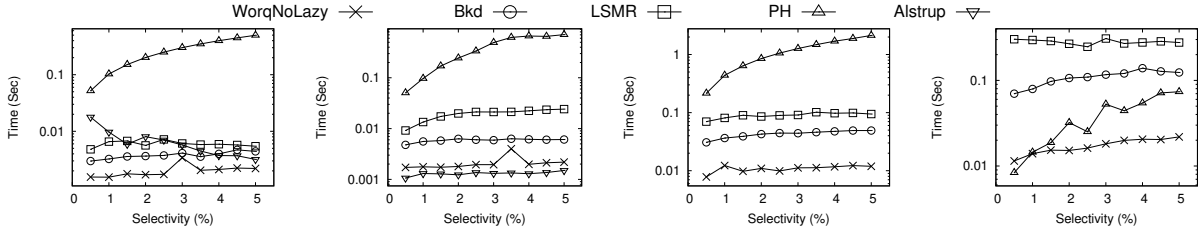
Hence, since there are $\lceil \log_2 \frac{N}{M} \rceil$ trees, the I/O cost of query is $O((N/B)^{\frac{2d-3}{2d-1}} \log_B \frac{N}{B} \log_2 \frac{N}{M})$. □

By Theorem 1 and Theorem 2, we know that the write efficiency of the WORQ-tree is $O(d \log_M \frac{N}{B} \log_2 \frac{N}{M}/B)$ and the query efficiency is $O((N/B)^{\frac{2d-3}{2d-1}} \log_B \frac{N}{B} \log_2 \frac{N}{M})$.



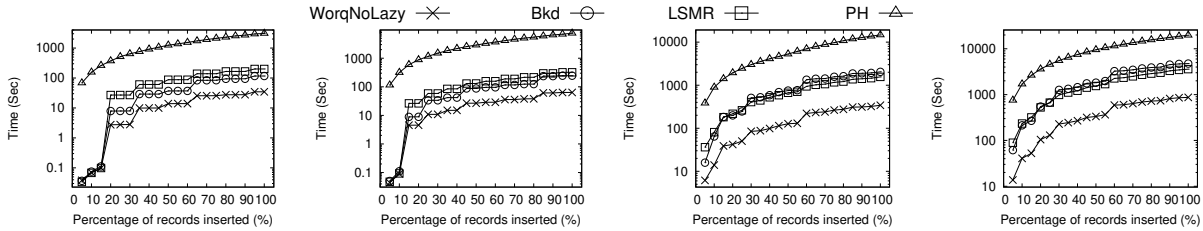
(a) Gowalla-small (b) London-small (c) Chicago-small (d) Berlin-small

Figure 15: Insertion time for small-datasets on HDD



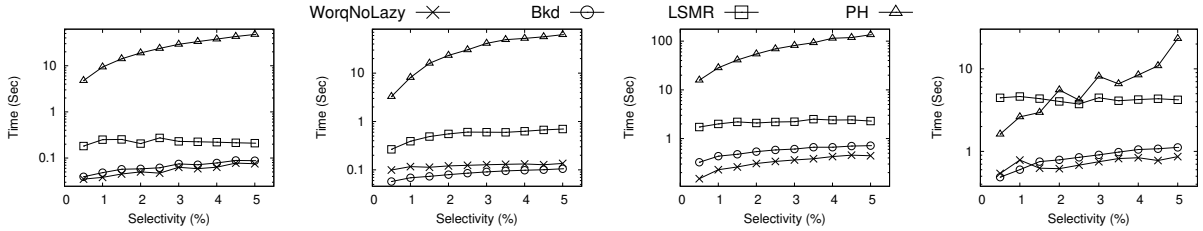
(a) Gowalla-small (b) London-small (c) Chicago-small (d) Berlin-small

Figure 16: Query time for small-datasets on HDD



(a) Gowalla (b) London (c) Chicago (d) Berlin

Figure 17: Insertion time for original-datasets on HDD



(a) Gowalla (b) London (c) Chicago (d) Berlin

Figure 18: Query time for original-datasets on HDD

Additional Experiment: Efficiency on HDD

Besides SSD, we also conduct experiments on HDD to evaluate the efficiency and effectiveness of our data structures.

The write efficiency and query efficiency results for small-datasets on SSD are shown in Figure 15 and Figure 16, respectively. Although the efficiency is not so high as the results on SSD, we can also get the same observation as above. Alstrup's data structure theoretically has high query efficiency, but it is not that usable due to its unacceptable write efficiency and space efficiency.

The results of write efficiency and query efficiency for original-datasets on SSD are shown in Figure 17 and Figure 18, respectively. There is an exception that in Figure 18(b), our WORQ-tree shows 1.2× slower than Bkd-tree. However, it is for a lower-dimensional and smaller-scale dataset. For the other larger and higher dimensional datasets, WORQ-tree keeps showing a 6.0× and 5.5× write efficiency and 1.6× and 1.4× query efficiency in Figure 17 and Figure 18.

Additional Experiment: Deletions

We also test the efficiency of deletions and find that there is no observable difference between all the mothods. We choose Chicago-large as the dataset, randomly perform 1000 point deletions and calculate the running time. Figure 19 shows the results. Since all these four methods are using the same LSM technique, they are

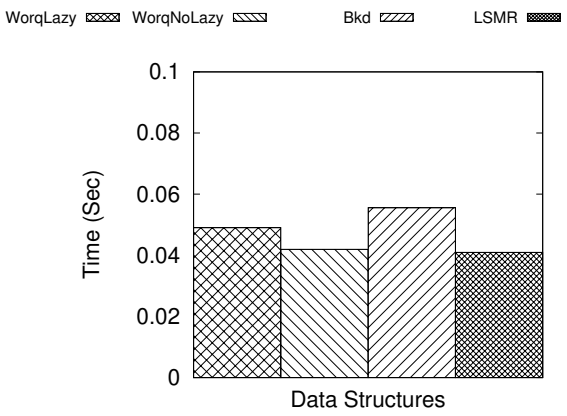


Figure 19: Deletion time for Chincago-large on SSD

supposed to perform the same on deletions. In addition, we focus on write-intensive workloads, so the deletion will not change the ranks of our methods and the competitors.