Incremental Partitioning for Efficient Spatial Data Analytics

Tin Vu, Ahmed Eldawy, Vagelis Hristidis, Vassilis Tsotras
Department of Computer Science & Engineering
University of California, Riverside, USA
tin.vu@email.ucr.edu,{eldawy,vagelis,tsotras}@cs.ucr.edu

ABSTRACT

Big spatial data has become ubiquitous, from mobile applications to satellite data. In most of these applications, data is continuously growing to huge volumes. Existing systems for big spatial data organize records at either the record-level or block-level. Systems that use record-level structures include key-value stores and LSM-Tree stores, which support insert and delete operations and they are optimized for highly-selective queries. On the other hand, systems like GeoSpark that use block-level structures (e.g. 128 MB each) are more efficient for analytical queries, but they cannot incrementally maintain the partitioned data and do not support delete operations. This paper proposes a general framework that enables block-level systems to incrementally maintain spatial partitions, in the presence of bulk insertions and deletions, in distributed file system (DFS) blocks. We first formally study the incremental spatial partitioning problem for big data and demonstrate its NP-hardness. Then, we propose a cost model to estimate the performance of queries on the partitioned data and the effect of modifying it as the data grows. After that, we provide three different implementations of the incremental partitioning framework. Comprehensive experiments on large real datasets show that our proposed partitioning algorithms outperforms state-of-the-art spatial partitioning methods.

PVLDB Reference Format:
Tin Vu, Ahmed Eldawy, Vagelis Hristidis, Vassilis Tsotras. Incremental Partitioning for Efficient Spatial Data Analytics. PVLDB, 15(3): 713-726, 2022.
doi:10.14778/3494124.3494150

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://bitbucket.org/tvu032/beast-tv/src/inc-update/beast-spark/src/main/java/edu/ucr/cs/bdlab/beast/operations/update/ and UCR STAR [30].

1 INTRODUCTION

Spatial data is being produced at increasing rates from various sources such as mobile applications and satellite data. For example, there is an average of 500 million tweets sent every day [55] from users at different spatial locations. NASA EOSDIS adds about 6.4 TB of data to its archives every day [27]. In all these applications, data is not only large in volume, but it is also continuously growing and changing. These characteristics urged the research community and industry to develop new systems for big spatial data [24, 59, 61].

Proceedings of the VLDB Endowment, Vol. 15, No. 3 ISSN 2150-8097.
doi:10.14778/3494124.3494150

When organizing spatial data, there are two main approaches, depending on the query processing needs of the system. If the focus is on highly selective queries (e.g. point look-up, top-k), the data is indexed; in the first approach (termed *record-level*) every record finds its exact position in the index structure (e.g., R-tree [31] and quad-tree [51]) so that the highly selective query will access a very few records using the index structure. While such queries are fast, there is an overhead in maintaining the index. On the other hand, if the focus is on analytical queries (e.g. aggregates, spatial joins [50], kNN joins [40], polygon union [22], convex hull, Voronoi diagram [37], DBSCAN [33], etc.), it is better to partition the data at a coarser granularity. Here, the exact record position is not important; rather records are organized in partitions (e.g., hash or range partitioning). After a record's partition is identified, its position within the partition is not important. This is because an analytical query will read all records in each partition that it accesses. As a result, the second approach (termed *block-level*) incurs less overhead in creating and maintaining the partitions, compared to the index maintenance of the record-level approach.

In order to support high ingestion or deletion rates while providing indexed access to files (*record-level* approach) systems use the Log-Structured Merge-tree (LSM-Tree) data organization [4, 15, 48]. In an LSM-Tree, new records are inserted sequentially in (fast) main memory to create a component file (also called memtable). After a component file gets full, it is written sequentially to pages. Eventually component files in the pages are merged together and records find their exact position in the index. Record-level approaches using the LSM-Tree include Apache HBase[32], Accumulo[1], AsterixDB [5], MD-HBase [46], Parallel Secondo [38], BBoxDB [44], and GeoMesa [35], among others. In these systems, a new record is sent to one of the participating nodes and is then indexed by a spatial index residing in that node. A highly selective query will run in parallel on each node accessing only relevant records through each node's spatial index. While LSM-Tree allows such systems to have high update rates, the amortized maintenance cost per record remains high.

On the other hand, distributed query processing engines that focus on analytical queries, such as Spark and Hadoop, follow the *block-level* approach. Typically the block size is 128 MB, which is much larger than a disk page (4-8KB) of the *record-level* storage engines. Example systems that use this approach include Slalom [47], for general purpose data analytics, and systems that are tailored towards spatial data analytics (i.e. using spatial partitions) such as SpatialHadoop [23], Simba [59], and GeoSpark [61], among others [24]. Unfortunately, due to the sequential write limitation of DFS (files are written sequentially to avoid expensive random writes), like HDFS [52], Amazon S3 [8] and Microsoft Azure [10] Blob storage, there is no mechanism that current block-level systems can use to maintain their spatial partitions incrementally.

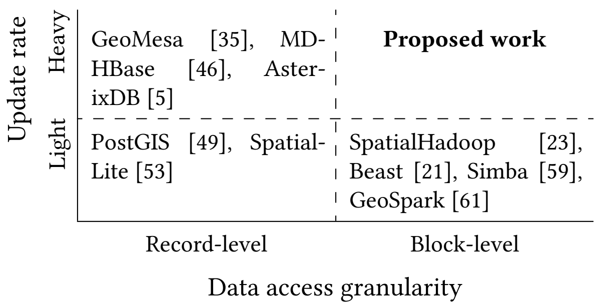

Update rate
Heavy
Light
GeoMesa [35], MD-HBase [46], AsterixDB [5]
Proposed work
PostGIS [49], SpatiaLite [53]
SpatialHadoop [23], Beast [21], Simba [59], GeoSpark [61]
Record-level
Block-level
Data access granularity

Figure 1: Position of the proposed work in the context of existing systems

The problem is exacerbated by the presence of *updates*. For example, a Twitter analytics system must periodically delete bot tweets, which are discovered by bot classifiers that run periodically on the data. This requires efficiently handling batch updates, which are currently not adequately handled by existing block-level platforms.

In this paper, we study how a system can combine both efficient spatial analytic queries and high ingestion rates, as shown in Figure 1. Single-machine Spatial DBMS systems, e.g., PostGIS, provide record-level indexing for relatively low ingestion rate that a single machine can provide. Big-data Management Systems (BDMS) and key-value stores, e.g., AsterixDB and GeoMesa, are able to support very high ingestion rates for record-level indexes by using distributed LSM-Tree indexes. On the other hand, block-level partitioning on big spatial data is supported by systems like SpatialHadoop, GeoSpark and Simba but they can only support applications with low ingestion rates where the data can be occasionally repartitioned. Furthermore, block-level systems naturally do not support delete operation. The proposed framework in this paper supports high insertion and deletion rates while incrementally partitioning the data in blocks. *Hence, our proposed work can be viewed as the LSM-equivalent for block-level spatial data, where the goal is to avoid the high overhead of the record-level LSM merges.*

This paper proposes a new framework that can efficiently support incremental big spatial datasets with batch ingestion, addressing the limitations of the state-of-the-art. We focus on ingestion/ deletion since for the applications we consider new data is continuously added, deleted, or updated. A *first key property* of the proposed framework is to facilitate partitioning of data based on their spatial attributes, so as to perform fewer accesses during query time. The partitions are created and stored directly in DFS which allows MapReduce and RDD programs to run directly on the partitions. A *second key property* is to facilitate a pay-as-you-go partition maintenance mechanism that can be optimized based on the objectives of the application and the system workload.

To achieve these properties, the proposed framework has two phases, namely, (i) *data flushing*, where new or deleted data is periodically pushed to secondary storage, and (ii) *partition optimization*, where the newly added/deleted data and the old partitions are jointly maintained given cost budget constraints.

Initially, we formalize the partition optimization problem and prove its NP-Hardness, which implies that building optimal partitions is non-trivial. Therefore, we break the problem into two subproblems, namely: *partition selection* and *partition reorganization*, which can be solved separately. The former gives us a set of potential partitions to reorganize and the latter physically reorganizes the records in the selected partitions. We focus on the partition selection problem which is critical in the reorganization process and has two key challenges. *(a)* how to formalize and estimate the benefit of selecting a specific set of partitions to reorganize, and *(b)* how to efficiently navigate the combinatorial search space of partition subsets, given a reorganization budget. For that, we introduce a novel block-based cost model for spatial partitioning that estimates the processing cost for a range query, and propose an algorithm to select a set of partitions to optimize this cost.

We implement three solutions to the *partition selection* problem, termed, *R*-Tree-Inspired Partitioning (R*P)*, *LSM-Tree-Inspired Partitioning (LSMP)*, and *Cost Based Partitioning (CBP)*. R*P is inspired by the node insert and node split algorithm in R*-tree. Similarly, LSMP utilizes the idea of LSM-Tree merge policy to maintain its partitions. Finally, CBP utilizes the proposed range query cost model so that it can minimize the estimated query cost, given a limited reorganization budget. The experimental results indicate that the proposed cost model allows us to create a high performance incremental partitioning scheme in terms of both partitioning time and query throughput, as compared to our two straightforward solutions and to state-of-the-art big spatial data systems.

In summary, the main contributions of this paper are:

- We introduce a comprehensive framework for incremental spatial partitioning of large-scale spatial datasets.
- We formalize the partition optimization problem for incremental spatial partitioning systems and prove its NP-Hardness.
- We propose three different implementations of the partitioning frameworks, including an approximate algorithm to solve partition optimization problem.
- We perform an extensive experimental evaluation on incremental big spatial systems to measure their performance and partitioning quality.

The rest of this paper is organized as follows: Section 2 reviews related works. Section 3 gives an overview of the proposed framework. Section 4 formalizes the partition optimization problem and proves its NP-Hardness. Section 5 proposes a new cost model for distributed spatial indexes. Section 6 shows the different incremental spatial partitioning techniques which implemented the proposed framework. Section 7 shows experimental results to validate our proposed work while Section 8 concludes the paper.

2 RELATED WORK

The existing work for supporting big spatial data can be broadly categorized into two categories, *record-level* systems and *block-level* systems, as depicted in Figures 2(a) and 2(b).

Record-level Systems: Systems in this category address the write limitation of DFS by utilizing the LSM-Tree [48] which converts random writes to sequential writes and merges. This idea was originally used to build key-value stores with a single-dimensional index, e.g., BigTable [15], HBase [32], and Accumulo [1]. To extend this idea to spatial data, some systems use space-filling-curves (SFC) to convert the high-dimensional coordinates to a single dimension,

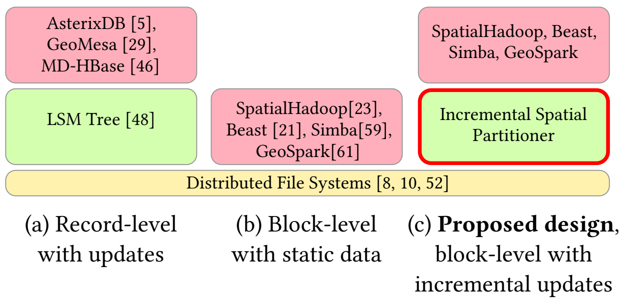

AsterixDB [5], GeoMesa [29], MD-HBase [46]
LSM Tree [48]
SpatialHadoop[23], Beast [21], Simba[59], GeoSpark[61]
SpatialHadoop, Beast, Simba, GeoSpark
Incremental Spatial Partitioner
Distributed File Systems [8, 10, 52]
(a) Record-level with updates
(b) Block-level with static data
(c) Proposed design, block-level with incremental updates

Figure 2: Architecture of big spatial data access methods

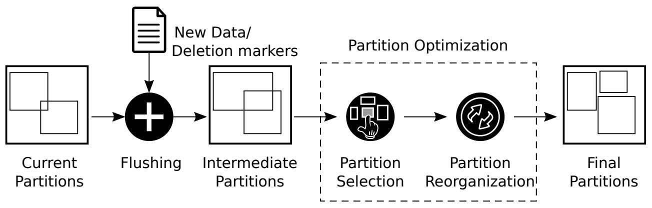


Figure 3: Overview of the proposed framework

e.g., MD-HBase [46], MongoDB [43], and GeoMesa [35]. Alternatively, AsterixDB [5, 9] hash-partitions the records based on their IDs and then builds an R-tree index for each LSM component.

The drawback of this category is the huge overhead in processing the records since records have to be retrieved one-by-one through the index which makes this approach suitable for highly-selective queries that returns only a few records but does not scale for analytic queries that tend to process most or all the records.

Block-level Systems: Figure 2(b) indicates how the existing *block level approach* builds spatial partitions at the block level directly on top of a DFS. The work in this category is geared towards analytical queries that process huge amounts of data. Instead of organizing the data at a record level, these systems organize them in large blocks of 128 MB each. Examples include MapReduce-based systems like SpatialHadoop [23] or Spark RDD systems like GeoSpark [61] and Simba [59], streaming-based systems like Tornado [41], and data warehousing systems like Sphinx [26]. Some of these systems, e.g., Kangaroo [6], account for query workload but are still limited to static data, i.e., all data needs to be repartitioned.

The main limitation of all these systems is that they cannot incrementally maintain the data in a spatially partitioned way. Whenever new records are inserted or old records are deleted, the data has to be *fully repartitioned* to reach the best performance. To address this limitation, there are several techniques on adaptive spatial data partitioning such as AQWA [7], Schism [17], and others [2, 19, 56]. However, none of these techniques address the problem of incremental partitioning in a distributed file system. They are either limited to traditional database systems [2, 17, 19, 56] or in-memory partitioning [7],

Proposed Work: Figure 2(c) shows the proposed work which introduces the incremental spatial partitioner into the block-level approach to enable incremental partitioning on DFS. In analogy with record-level systems, this work plays the same role that the LSM-Tree did to enable record-level indexing on the limited DFS, i.e., it enables block-level systems to incrementally maintain spatial partitions of the data without the need to modify files in DFS.

3 A GENERIC INCREMENTAL SPATIAL PARTITIONING FRAMEWORK

Figure 3 gives an overview of the proposed framework for incremental partitioning of big spatial data. This is a generic framework in the sense that its steps can be implemented differently to produce various types of incremental partitioning schemes. This paper provides three different implementations that follow this framework. The framework consists of two phases, *data flushing* and *partition optimization*. The *data flushing* phase ingests a set of new records and deletion markers into an existing, initially empty, partition structure. To adhere with HDFS limitations, this step can only append to existing files or write new files to produce an intermediate unoptimized partitioning.

The second *partition optimization* phase takes the intermediate partitioning and partially reorganizes it to produce an optimized final partitioning. This phase first identifies a subset of partitions, then it reorganizes their contents into a new set of partitions. In Section 4, we prove that the *partition optimization* problem is NP-hard. As it is impractical to find an optimal solution, we break the problem into two sub-problems, *partition selection* and *partition reorganization*, solved separately as discussed below.

The *partition selection* step identifies a subset of *bad* partitions to be reorganized. The second step, *partition reorganization*, processes the selected partitions by reorganizing their contents into a set of *new* partitions; old partitions are then deleted. Since HDFS does not allow random updates, the modified files have to be completely rewritten.

We proceed by first describing the layout of the partition on the distributed file system; then we provide more details about the three main steps of our framework (data flushing, partition selection, and partition reorganization).

3.1 Partition layout

This paper focuses on block-level partitioning in which each partition fits in one 128-MB HDFS block [52]. Local indexing can be employed to determine the internal format of each 128 MB block. However, this is outside the scope of this paper since we focus on analytical queries which are only marginally improved by local indexes [23]. Each partition is stored on disk as a separate file. Additionally, a *master* file stores the metadata of the partitions which consists of a partition ID, the minimum bounding rectangle (MBR) of the partition, number of records, total size of records marked for deletion, and total size of non-deleted records. The master file acts like the global index of partitioned data. A spatial query starts by examining the master file to decide which partitions to process, e.g., the partitions that overlap the area of interest. Then, the selected partitions are processed in parallel using MapReduce [18] or RDD [62]. If multiple master files exist, the most recent one is used to adhere with multiversion concurrency control schemes.

3.2 Data flushing phase

As shown in Figure 3, the data flushing phase takes a batch of new data or deletion markers and ingests it to the partitioning. Typically, systems that deal with big data, buffer these updates in memory and trigger the flushing phase when the in-memory component reaches a pre-specified threshold, e.g., 4GB. Since this phase is triggered while the system is *hot* and accepting updates to the data, it prioritizes the insertion time over the quality of the partition. This allows the system to continue accepting new records at the highest rate with minimal partition maintenance overhead. In this proposed framework, we limit the flushing phase to appending to existing files and writing new files. This limitation is driven by the DFS limitation and also helps in reducing the amount of disk IO, which is equal to the batch size. The *master* file is not changed until the flushing phase is complete which makes all the changes hidden to the query processing. Once the flushing process completes, all the changes become visible by writing a new version of the master file that reflects the updates.

This paper considers two flushing techniques, namely, *append*, which appends records to existing partitions; and *LSM flush*, which creates a new set of partitions (that logically form a new LSM component). Both are described in detail in Section 6.

3.3 Partition selection

The partition selection step identifies *target* partitions that need to be deleted and reorganized. We design this step to run under a system constraint which limits the amount of disk IO in this process (read + write). The goal is to choose a small subset of *bad* partitions, e.g., overlapping partitions, that are lowering the quality of the partitioning and reorganize them to boost its quality. For efficiency, this step operates only on the partition metadata, e.g., MBR and size, from the latest master file. Therefore, this step only takes a fraction of a second which is negligible on the performance but its result has a significant impact on the quality of the partition and the overall reorganization time.

3.4 Partition reorganization

Given a list of target partitions from the previous step, the partition reorganization step completes the partition optimization phase by physically rewriting those partitions into highly-optimized partitions, with the removal of records which are marked for deletion. This step is generic and can use any existing static partition construction algorithm for big spatial data. The cost of this step is linear in the total size of selected partitions since it runs in one scan over the repartitioned data.

4 PARTITION OPTIMIZATION PROBLEM

4.1 Preliminaries and problem definition

To formulate the problem, we first present the following definitions; notations are summarized in Table 1:

- $r(mbr, size, is\_deleted)$: a record r is represented by its minimum bounding rectangle (MBR) and size in bytes. In addition, $is\_deleted$ is a Boolean tombstone flag with value $\{0, 1\}$ to indicate whether this is a deleted record.
- b: default block size in the file system, e.g., 128 MB.

Table 1: Table of notations

Notation	Description
r	a record.
b	default block size, e.g., 128 MB.
D	a dataset D is a set of records.
$MBR(D)$	the MBR of a set of records D.
$psize(D)$	the physical size of D.
$csize(D)$	the condensed size of D.
$pblocks(D)$	the number of physical blocks of D.
$cblocks(D)$	the number of condensed blocks of D.
P	A global partitioning (set of partitions)
$T(P, P')$	The disk IO cost to transform P to P'
$C(P)$	Function to compute query cost on P

- $D = \{r_1, \ldots, r_n\}$: a dataset D is a set of records.
- $MBR(D) = \bigcup_{i=1}^{n} mbr(r_i)$: the MBR of a set of records is the minimum MBR that contains the MBRs of all its records.
- $psize(D) = \sum_{i=1}^{n} size(r_i)$: the physical size of a set of records is the sum of all record sizes. This indicates the disk space needed to store these records.
- $csize(D) = \sum_{i=1}^{n} (1 - 2 \cdot is\_deleted_i) \cdot size(r_i)$: the condensed size of a set of records is the size this set would occupy if deleted records are removed. It is computed by subtracting the size of deleted (tombstone) records.
- $pblocks(D) = \left\lceil \frac{psize(D)}{b} \right\rceil$: is the number of physical blocks for a set of records.
- $cblocks(D) = \left\lceil \frac{csize(D)}{b} \right\rceil$: is the number of physical blocks D will occupy if condensed..
- $P = \{p_1, \ldots, p_m\}$: A global partitioning state, hereafter will be simply called a *partitioning*, of a dataset is a set of partitions p_i, where each partition is a subset of the input D. Partitions satisfy the following three constraints, (1) $p_i \subseteq D$, (2) $p_i \cap p_j = \emptyset, \forall i \neq j$, and (3) $\bigcup_i p_i = D$. Similar to the dataset D, we can also compute *MBR*, *psize*, *csize*, *pblocks* and *cblocks* of every $p_i \in P$. In this work, we assume the records inside a partition are not indexed (i.e., there is no local index), but this decision is orthogonal to the reorganization problem that we study. Previous work[23] showed that local indexes have little impact on the overall performance of analytical queries on big spatial data.
- $T(P, P')$: Given two partitions, P and P', the cost to transform P to P' is defined as the number of blocks to read from P and the cost of writing new condensed partitions in P'.

$$T(P, P') = \sum_{p_i \in P \setminus P'} pblocks(p_i) + \sum_{p_i \in P \setminus P'} cblocks(p_i)$$

- $C(P, s)$ is a function that measures the average cost of running a range query with size $s \times s$ on P. This metric, formally defined in Section 5.1, reflects how good the partitioning is for spatial query processing.

Partition Optimization Problem *Given a dataset D with partitioning P, a fixed number of accessed blocks B and a query size $s \times s$, find a new partitioning P' which can be transformed from P where $T(P, P') \leq B$ and $C(P', s)$ is minimized.*

4.2 The NP-Hardness of the problem

We proceed to prove that a simplified version of the partition optimization problem is NP-Hard (and hence the partition optimization problem is also NP-Hard). In particular, we show that if we have a quality function that is defined independently for each partition, the problem can be reduced from the well-known 0-1 Knapsack problem.

(0-1 Knapsack) Given a set of n items numbered from 1 *to n, each with weight w_i and value v_i, and a maximum weight capacity W, find a vector $X = \{x_1, \ldots, x_n\}$ that:*

$$\text{maximize} \sum_{i=1}^{n} v_i x_i$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i \leq W \text{ and } x_i \in \{0, 1\}$$

Reduction Algorithm: We transform the 0-1 Knapsack problem to the following partition optimization problem:

- Define D as a set of n records where the size of each record r_i is w_i.
- The current partitioning P contains n partitions, where each partition contains only one record, $p_i = \{r_i\}$.
- The budget B is equal to the weight capacity W, and the block size $b = 1$.
- The cost function for one partition $c(p, s)$ is:

  $$c(p,s) = \begin{cases} 0 & if\ |p| = 1 \\ \sum_{r_i \in P} v_i - \sum_{r_i \in p} v_i & otherwise \end{cases}$$

  Finally, we define $C(P, s) = \sum_{p_i \in P} c(p_i, s)$. In particular, the cost for a partitioning state P is the total of cost for each partition in P.

To complete the proof, we need to show that (1) the optimal answer can be mapped between the two problems, and (2) both the problem reduction and the answer mapping require a polynomial time.

(1) Assume that the optimal solution for the 0-1 Knapsack problem is $X = \{x_1, \ldots, x_n\}$. The set of items with $x_i = 1$ correspond to a subset of records $R = \{r_i | x_i = 1\}$. In this case, there is an optimal answer P' where each item $x_i = 0$ maps to a partition with one record $p_i = \{r_i\}$, and all the items with $x_i = 1$ are combined in one partition R. On the other hand, given an optimal answer P' to the partition optimization problem, we can map it to an optimal answer to the 0-1 Knapsack as follows: Each record $r_i \in p'_j$ where $|p'_j| = 1$ will be mapped to $x_i = 0$, otherwise, if $|p'_j| \geq 2$, r_i is mapped to $x_i = 1$. The key idea is that all partitions selected to be modified by the partition optimization problem as an optimal solution will have a total cost equal to the weights of their records and the answer will have a quality equal to the total value of those selected records.

(2) Both the reduction algorithm and the answer mapping above require $O(n)$ time complexity, i.e., the reduction process is a polynomial-time algorithm.□

5 COST-BENEFIT ANALYSIS OF THE PARTITION SELECTION PROCESS

Given that the partition optimization problem is NP-hard, we break it into two smaller sub-problems: *partition selection*, which selects a subset of partitions to be reorganized, and *partition reorganization*, which reorganizes the records in the selected partitions. Previous work on static indexes for big spatial data [20, 23, 39, 57, 59] can be used to solve the second problem while there has been little attention to the first one. This section focuses on the first problem, provides a theoretical analysis, and develops a cost model for it.

The key idea is to create an accurate cost model for range queries and use it as a proxy for the quality of the partitions (Section 5.1). We use range queries as they are the most fundamental operation in spatial data analytics and it was shown to strongly predict the performance of indexes for other queries [20, 34]; range query is commonly used as the building block for other spatial operations such as joins or aggregations, as we discuss below. Then, we define a *benefit function* that uses the cost model to estimate the improvement in the partition quality for any subset of selected partitions (Section 5.2). The next section will show how to use this cost model to solve the partition selection problem.

5.1 Range query cost model in HDFS

We build on previous research which showed that the cost of a range query is a good proxy for the quality of a spatial partitioning method (i.e., a partitioning state *P*) even for analytical queries such as spatial join [20, 34]. Given a fixed range query of size $s \times s$, our cost model estimates the total number of blocks and total size of data that it will process. This cost model allows us to maintain optimized partitions without the awareness of query workload. Building an adaptive spatial partitioning method is outside the scope of this paper.

Traditional range query cost models [3, 12, 16, 54] focused on estimating the number of disk pages required to answer the query or the query size. This made sense for traditional DBMS algorithms which access data from a regular disk with a relatively small disk page, e.g., 8 KB. The assumption was that a disk page is the smallest access unit to a disk which makes the cost uniform on all disk pages regardless of how many actual records are in each page.

When transitioning to the distributed file system, the previous assumptions no longer hold. In HDFS, data is stored in blocks which can vastly vary in size from a few megabytes up-to 128 MB. Therefore, the cost of accessing each block is no longer uniform. In addition, we have an opportunity to build a more accurate estimate as compared to traditional models. In traditional DBMS, the cost estimation is part of the query optimization which should take only a few milliseconds. However, this work uses this cost model as part of the index optimization step which can take tens or hundreds of seconds so we have an opportunity to run a model that takes a second or two without significantly hurting the overall performance.

Our analysis starts with an $s \times s$ square-shaped query and tries to estimate its cost.This cost model considers square queries for the sake of simplicity but it can be extended for general rectangular range queries. s is a system parameter that can be set to the most common query size expected by the application. Assume that $P = \{p_1, \ldots, p_m\}$ is a partitioning state of a spatial dataset D. Suppose that $w(p_i)$ and $h(p_i)$ are the width and height of a partition p_i, respectively. Figure 4 shows the two possible relationships of a query q to the partition p_i. As shown in the figure, the query q_1 is disjoint with the partition p_i which means that such query does

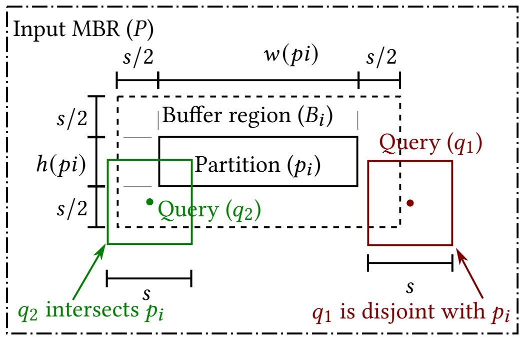


Figure 4: Relationship of a range query with a partition

not have to process the partition p_i. On the other hand, the query q_2 overlaps p_i and hence needs to process that partition to produce the answer. We compute the probability of a query of size $s \times s$ being disjoint or overlapping with the partition p. To do that, we define a *buffer region* B_i that expands p_i with a buffer size of $s/2$ in all directions. We can easily see that if the center of the query falls inside the buffer region (e.g., q_2), it overlaps the partition; otherwise, it is disjoint. For a random square range query q_i of size $s \times s$ with its center in the domain space $MBR(D)$, the probability that the center of q_i falls inside the buffer region is the ratio of the area of B_i over the area of the domain space $MBR(D)$. Thus the average number of blocks that are contributed from p_i for query q_i is:

$$c_b(p_i, s) = \frac{(w(p_i) + s)(h(p_i) + s)}{w(P)h(P)} \cdot pblocks(p_i) \tag{1}$$

In addition, the average amount of data in bytes that is scanned from p_i for query q_i is:

$$c_s(p_i, s) = \frac{(w(p_i) + s)(h(p_i) + s)}{w(P)h(P)} \cdot psize(p_i) \tag{2}$$

To process a query in HDFS, first, there is a fixed cost to locate overlapping blocks, which correlates with number of partition's blocks. Second, there is a cost to scan the entire partition, which correlates with partition size. Therefore, the total cost (running time) to complete a square query with size $s \times s$ can be represented as the following:

$$C(p_i, s) = k_b \cdot c_b(p_i, s) + k_s \cdot c_s(p_i, s) \tag{3}$$

The hardware-specific coefficients k_b and k_s have units of seconds/block and seconds/byte, respectively; together, they unify the cost units to seconds. We determine these constants as the follows: 1) run a small number of range queries with varying number of blocks and sizes, 2) collect c_b, c_s and C, 3) fit these data points to a linear regression model. Overall, the average processing time to answer a query q_i on the partitions of P can be estimated as follows:

$$C(P, s) = \sum_{p_i \in P} C(p_i, s) = k_b \cdot \sum_{p_i \in P} c_b(p_i, s) + k_s \cdot \sum_{p_i \in P} c_s(p_i, s) \tag{4}$$

5.2 Reorganization benefit

This section shows how to use the range query cost model described above to estimate the *benefit* of the reorganization step. We define the benefit as the reduction in the range query cost after the reorganization process, i.e., cost after subtracted from the cost before. To formalize the benefit calculation, suppose that the state of partitioning at timestamp t is P_t (intermediate partitions) and after reorganization it will be P_{t+1} (final partitions) as presented in Figure 3. Now, assume that the partition selection step has selected a group of partitions $G_t = \{p_1, \ldots, p_n\} \subseteq P_t$ to be reorganized. After these partitions are reorganized, they will produce a new set of partitions $G_{t+1} = \{p'_1, \ldots, p'_m\} \subseteq P_{t+1}$. We define the *reorganization benefit* of G_t as the reduction of query cost when the partitions are reorganized from P_t to P_{t+1}.

$$Benefit(G_t, s) = C(P_t, s) - C(P_{t+1}, s) \tag{5}$$

We can rewrite P_t as $(P_t - G_t) \cup G_t$ and similarly P_{t+1}.

$$Benefit(G_t,s)=C((P_t-G_t)\cup G_t,s)-C((P_{t+1}-G_{t+1})\cup G_{t+1},s) \tag{6}$$

Since our cost function C is linear, we can apply super position as follows.

$$\begin{aligned} Benefit(G_t, s) &= C(G_t, s) - C(G_{t+1}, s) \\ &+C(P_t - G_t, s) - C(P_{t+1} - G_{t+1}, s) \end{aligned} \tag{7}$$

But $P_t - G_t \equiv P_{t+1} - G_{t+1}$, which are the set of non-selected partitions. Their cost is the same which means that the benefit of reorganizing G_t is:

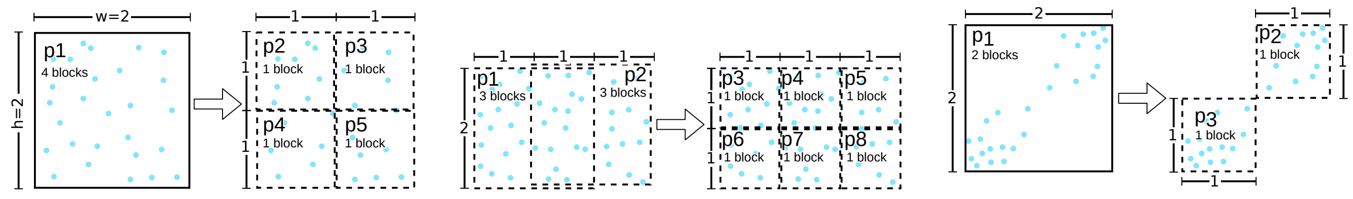
$$Benefit(G_t, s) = C(G_t, s) - C(G_{t+1}, s) \tag{8}$$

Finally, we can utilize Equation 8 to compute the benefit of the reorganization step that transforms P_t to P_{t+1}. To understand the key idea behind the computation of cost reduction, i.e., benefit, Figure 5 illustrates three examples of partitions before and after reorganization. For simplicity, we assume $k_b = 1$ and $k_s = 0$, then the total cost in Equation 4 becomes $C(P, s) = \sum_{p_i \in P} c_b(p_i, s)$. In all three examples, we assume that the area of data space is $w(P) \cdot h(P) = 10$ and query size $s = 0.5$. In Figure 5(a), a single partition p_1 with four blocks is reorganized into four single-block partitions. According to the simplified cost model and Equation 1, $C(p_1, s) = \frac{(2+0.5)(2+0.5)}{10} \cdot 4 = 2.5$ and $\sum_{i=2\ldots5} C(p_i, s) = 4 \cdot \frac{(1+0.5)(1+0.5)}{10} \cdot 1 = 0.9$. So, we say that the reduction in cost is $2.5 - 0.9 = 1.6$. This value means that if we reorganize p_1 into 4 smaller partitions, we would reduce 1.6 block accesses on average to answer a square query with size 0.5×0.5. This case indicates that partitioning a multi-block partition into several single-block partitions improves the cost.

The second case in Figure 5(b) gives an example of partitioning two overlapping partitions. According to our cost model the cost before reorganizing is $C(p_1, s) + C(p_2, s) = 2 \cdot \frac{(2+0.5)(2+0.5}{10} \cdot 3 = 3.75$ while the cost after reorganizing is $\sum_{i=3\ldots8} C(p_i, s) = 6 \cdot \frac{(1+0.5)(1+0.5)}{10} \cdot 1 = 1.35$. The cost reduction is 2.4. This case indicates that partitioning overlapping partitions provides additional cost reduction.

Figure 5(c) shows an example of splitting one partition p_1 with empty regions into two blocks p_2 and p_3 while uncovering that empty region. In this case, we can calculate the cost as $C(p_1, s) = \frac{(2+0.5)(2+0.5)}{10} \cdot 2 = 1.25$ and $C(p_2, s) + C(p_3, s) = 2 \cdot \frac{(1+0.5)(1+0.5)}{10} \cdot 1 = 0.45$. The reduction in the cost is 0.8. Thus reorganizing a partition that contains *dead space* reduces the cost.

In summary, the proposed benefit model favors the creation of the smallest number of single-block partitions. This fits well with



(a) Split a partition with uniform data (b) Split overlapping partitions (c) Uncover dead area by splitting skewed data

Figure 5: Different scenarios for reorganizing a group of partitions to reduce the estimated cost and improve the quality

the DFS design that will always split large files into single blocks that are processed individually. Further, the overhead that we add on each partition is minimal, i.e., MBR and size.

To estimate the benefit of a reorganization scheme (estimated cost reduction), we would like to take these three cases into account. Nevertheless, it would be hard to account for case 3 (Figure 5(c)) since it requires extra information about the data distribution inside the partition which is not available in the master file. Therefore, our benefit computation only accounts for the first two cases. The challenge here is we would not know how G_{t+1} looks like to compute the benefit in Equation 8 until we actually reorganize G_t. However, our goal is to use that equation to select the best subset $G_t \subseteq P_t$ in the partition selection process, which promises the maximum benefit. Therefore, we need to be able to estimate $Benefit(G_t, s)$ without physically reorganizing G_t, i.e., without knowing G_{t+1}. To resolve this issue, we compute a *prediction* $\hat{G}_{t+1}$ of G_{t+1} and use it to calculate a *prediction* (estimate) of the benefit. In this case, $\hat{G}_{t+1}$ is a set of partitions that we predict to produce after the reorganization step runs. In order to estimate $\hat{G}_{t+1}$, we assume that the partition reorganization step will produce $m = cblocks(G_t)$ single-block, square-shaped, equi-sized, and non-overlapping partitions. Since all the estimated resulting partitions are identical, their total area is equal to the area of the selected partitions G_t. Hence, the estimated side length of each of the new partitions $p'_i, \forall i = 1 \ldots m$ is calculated as:

$$\hat{w}(p'_i) = \hat{h}(p'_i) = \sqrt{\frac{w(G_t) \cdot h(G_t)}{cblocks(G_t)}} \tag{9}$$

Although this is an ideal case that might not always happen, it is a good indicator of how far the cost might go down. The actual output of the reorganization step might have a higher cost (if the partitions are overlapping and not square) or a lower cost (if a dead space was uncovered). Finally, the estimated benefit when we reorganize G_t to $\hat{G}_{t+1}$ is:

$$\widehat{Benefit}(G_t, s) = C(G_t, s) - C(\hat{G}_{t+1}, s) \tag{10}$$

Because we can compute G_t and $\hat{G}_{t+1}$ before the partition reorganization process, the estimated benefit in Equation 10 is a good indicator to help us choose the partitions for reorganization with maximum benefit, given a limited disk IO budget. We proved that this is a NP-Hard problem. Therefore, we will use a greedy strategy to solve this problem, which will be discussed in Section 6.3.

6 PROPOSED INCREMENTAL PARTITIONING ALGORITHMS

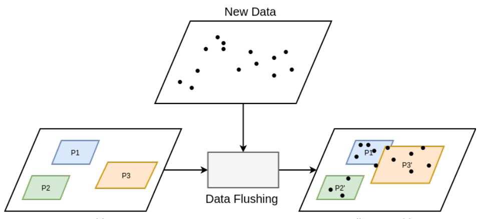
This section presents three approaches to instantiate the incremental partitioning framework in Section 3. The first two approaches, R*P and LSM-P, are inspired by the R*-tree [11] and LSM-tree [48], respectively, to implement the three steps in Figure 3. The third approach, Cost Based Partitioning (CBP), uses the proposed benefit model to maximize the estimated benefit and lowers the cost of the partitions.

Comparing the three approaches, R*P is designed to only split overflowing partitions as they get larger but cannot merge small partitions when they become partly empty. In contrast, LSMP only merges partitions together based on its merging policy but cannot split a single large partition. As a result, R*P and LSMP might not perform well for the incremental datasets with spatially overlapping insertion/deletion batches. This motivated us to design CBP, which can overcome these limitations. In particular, CBP periodically reorganizes the partitions, essentially both splitting and merging, measuring the benefit using Equation 10. This strategy allows CBP to behave well for a wider range of spatial ingestion workloads. We use R*-Grove [58] as the spatial partitioning algorithm in all approaches to guarantee a good partition load balance, but it could be replaced by other spatial partitioning techniques such as Grid File [45], Kd-tree [13], Quad-Tree [28, 51], or Hilbert R-Tree [36]. All of these techniques are sample-based partitioning [20], which supports common spatial data type such as point, line, polygon, circle, etc, with either disjoint or non-disjoint objects.

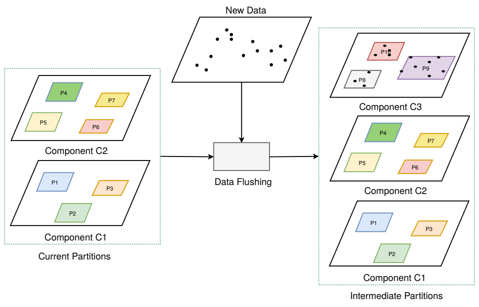
6.1 R*-Tree-inspired partitioning (R*P)

This section describes an adaptation of the traditional R*-tree index [11] to the incremental partitioning problem by implementing the three steps of the partitioning framework: *flushing*, *partition selection*, and *partition reorganization*. The general idea is to treat each partition as a leaf node in the R*-tree. The details of the three steps are described below.

Data Flushing: The flushing step uses the R*-tree insertion algorithm to choose a partition for each record and append it, as shown in Figure 6(a). The inputs to this flushing process are the current partition and a non-partitioned batch of new or to-be-deleted records. Based on the MBRs of the current partitions, the flushing process scans the records in the non-partitioned batch and appends each record to one of the partitions following the R*-tree ChooseSubtree method (i.e., choose the partition that requires the least

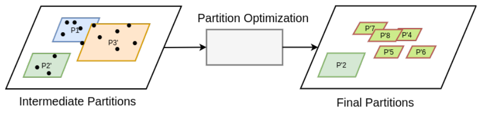


(a) R*P and CBP Flushing

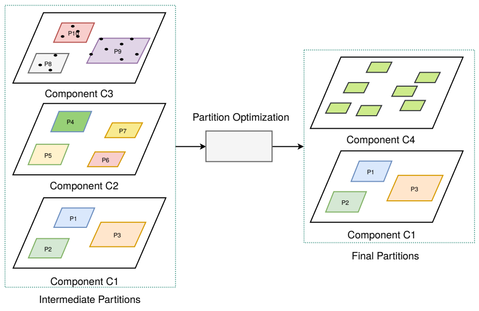


(b) LSMP Flushing

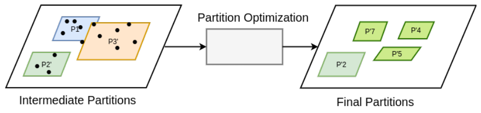
Figure 6: Data flushing in different techniques



(a) R*P partition reorganization



(b) LSMP partition reorganization



(c) CBP partition reorganization

Figure 7: Partition optimization in different techniques

overlap enlargement as detailed in [11]). The output of this flushing phase is a set of intermediate partitions which contains the same number of partitions as the input but the contents of these partitions include the new data records and the markers for deleted records.

Partition Selection: After the flushing phase is complete, the partition selection step identifies the partitions that need to be reorganized. Following the standard R*-tree design, this step simply selects all the *overflow* partitions that go beyond a maximum size M, in this case, 128 MB. For example, in Figure 7(a), partition p_1, p_3 is selected for reorganization because they went beyond the maximum partition size. Notice that we cannot implement the forced reinsert technique in R*-tree since random updates are not allowed in HDFS.

Partition Reorganization: This step reorganizes the partitions selected by the previous step using the R*-Grove [58] partitioning method. Overall, R*P is not suitable for datasets which require a high rate of record deletion or update. In particular, R*P only splits partitions into smaller partitions, while there is no merge mechanism for partitions which contain deleted records. As a result, there might be many small partitions when the datasets are updated over time as shown in Figure 7(a).

6.2 LSM-tree-inspired partitioning (LSMP)

In the traditional LSM-Tree [48], each batch is flushed and indexed as a separate LSM component. An LSM compaction policy merges these components depending on their sizes and order of creation. We adapt our generic partitioning framework to support an LSM variation where each component is indexed by an R*-Tree.

Data Flushing: In the *data flushing* phase, the new batch is partitioned as an R*-Tree, similar to [23]. Since the batch can be bigger than an HDFS block, it might consist of multiple partitions. Figure 6(b) shows a data flushing process (termed as LSMP Flushing) in which the new batch is partitioned into a new component C_3, besides the current components C_1, C_2 of our LSM partition. In an auxiliary *component file*, we store the component ID and creation order for each component. LSMP flushing requires the same disk IO as R*P Flushing, but it creates new partitions while the number of partitions does not change in R*P.

Partition Selection: This step scans the components' metadata, i.e., creation order and size, from the master file and the auxiliary component file. The standard LSM compaction policy from H-Base [32] is applied to the LSM components to identify the components that need to be merged. If there is more than one component to merge, all partitions in those components would be selected.

Partition Reorganization: In this step, a new partition component is reconstructed from all the partitions in all the selected components. This results in replacing all these components with one component which is considered the new merged component. Figure 7(b) shows how components C_2 and C_3 are reconstructed into a new component C_4.

The advantage of the LSM-tree-inspired partitioning is that it contains highly optimized partition components. It also works well for the datasets with update workloads, since the merging process always produces the optimized partitions. Thus it will provide a good query performance if the query range completely falls into only one component. However its performance will be negatively affected with large query ranges, when they require scanning multiple components. In order to address this drawback, the number of components should be reduced by the compaction process, with a trade-off of reconstructing time for multiple components.

Algorithm 1 Greedy Partition Selection Algorithm

```
1: function PartitionSelection($P = \{p_1, \dots, p_m\}$, $B$, s)
2:     $G = \{\}$                                        ▷ Set of selected partitions
3:     while $nblocks(G) \leq B$ do
4:         max-benefit = 0
5:         for each $p_i \in P$ do
6:             $b = \max\{\widehat{Benefit}(G \cup \{p_i\}, s),$
7:                  $\widehat{Benefit}(G, s) + \widehat{Benefit}(p_i, s)\}$
8:             if $b >$ max-benefit then
9:                 max-benefit = $b$
10:                $p* = p_i$                            ▷ Update selected partition
11:            $G = G \cup \{p*\}$
12:            $P = P - \{p*\}$
13:    return $C$
```

6.3 Cost-based partitioning (CBP)

In this approach, after each flush, we reorganize the partitions given a reorganization budget (we use a budget similar to the one spent by R*P in our experiments). The cost model in Section 5.2 is used to estimate the benefit of each candidate reorganzation.

Data Flushing: In general, the data flushing phase for CBP partitioning works in a same way with R*P, which was described in Section 6.1 and Figure 6(a).

Partition Selection: Based on Equation 10, we design a greedy algorithm to select a group of partitions that will likely lead to high benefit. We start with an empty set of selected partitions. Then, we scan the set of available partitions and choose the one that maximizes the benefit function if added to the selected group. We repeat this until the allocated budget B is used. Notice that once a partition is selected, the benefit of all other partitions change so the next iteration of the loop will have to recalculate all of them.

Algorithm 1 shows the pseudo code for the proposed partition selection algorithm. The input parameters include the list of current partitions, the budget of number of blocks B and the desired size s of range query. Line 2 initializes the set of selected partitions G to the empty set. Then, the loop in Lines 5-12 iterates over all the partitions in P to compute the benefit of each one. Line 7 calculates the benefit of each partition when added to the set of selected partitions. In Equation 10, the benefit of a group is not necessarily equal to the total benefit of its individual partitions. Therefore, we recompute the group benefit in each iteration. This is acceptable because the calculation of Equation 10 is not expensive. The partition that results in the maximum benefit increase is chosen (Line 10). The chosen partition $p*$ is then added to the set of selected partitions G and removed from the set of available partitions P. Notice that once a partition is added to G, the benefit of all other partitions might change so the next iteration of the loop will recalculate all of them. Once the total number of blocks in G is larger than the budget B the algorithm terminates and returns the set of selected partitions.

Partition Reorganization: When a large number of partitions is selected for reorganization, we need to decide whether to consider all of them in one group or we can split them into smaller groups. If we consider all of them in one group and apply Equation 9, we assume that the resulting partitions cover the entire space which might be inaccurate if there is a large gap between partitions. Therefore, we first split the selected partitions into groups, by adding all overlapping partitions in one group, and then partition each group independently. The reason that we reorganize the partitions in groups is to minimize the overlapping of reorganized partitions with existing partitions. This reduces the skewness of selected partitions, then makes the estimated benefit being more realistic. CBP might be able to create less reorganized partitions than R*P as shown in Figure 7(c) since it can merge under-utilized partitions into a single-block partition.

7 EXPERIMENTS

In this section, we perform a comprehensive experimental evaluation to highlight the advantages of proposed work over existing spatial data management systems. For record-level approaches, we compare to two state-of-the-art baselines, namely, AsterixDB 0.9.6 [5] and GeoMesa 3.1.2 [35]. For block-level approaches, we use Sedona 1.0.1 [60, 61] and Beast 0.9.1 [21], which are Spark-based spatial data systems. These baselines are compared to one of the three techniques that are proposed in this paper, namely, R*P, LSMP, and CBP. These systems are evaluated based on the ingestion time, partitioning quality, and query performance on the partitioned data. We use range query and spatial join as query workloads.

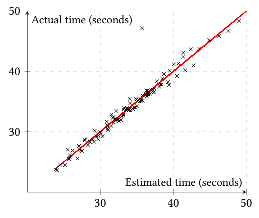
Datasets: We use the following spatial datasets in our experiments: (1) MS-Buildings dataset [42] with size 96 GB (751 million polygons). This dataset is synthesized from the original MS-Buildings dataset from UCR STAR, in which we extract the rectangles of building's geometries. In order to increase dataset size, new records are created by randomly shifting the rectangles of existing records. (2) OSM-All Objects dataset [63] with size 340GB. This dataset contains all map objects in the OpenStreetMap datasets. (3) OSM-Parks dataset [25] with size 8 GB (10 million polygons), a real dataset which represents green areas all over the world. This OSM-Parks dataset is only used in the spatial join performance evaluation, in a combination with MS-Buildings dataset. This combination makes the total processed data large enough for an expensive operation like spatial join.

Workloads: We evaluate different systems using insertion, deletion, and spatial analytical query workloads. For the insertion, we split input datasets into batches and keep adding these batches to the each of the systems. There are two type of deletion workloads: delete by batch, which deletes an entire batch that was inserted earlier, and delete by sample, which deletes a set of random records from all inserted records. The analytical queries include both range query and spatial join.

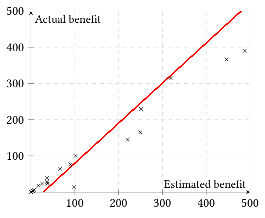
System specs: All experiments are executed on a cluster of one head node and 12 worker nodes, each having 12 cores, 64 GB of RAM, and a 10 TB HDD. They run CentOS 7, Oracle Java 1.8.0_131, Hadoop 3.2.1 and Spark 3.0.0.

7.1 Cost model and benefit model validation

7.1.1 Cost model validation. The proposed cost model in Section 5 mainly estimates the average execution time of a square range query. If the model is accurate, we expect that the estimated cost (provided by the model) and the actual cost (the total time to run the query) will have a high correlation. To verify the cost model, we partitioned subsets of the MS-Buildings dataset of different sizes. Then, we run square range queries of various sizes on all of them and measure both the estimated cost and the actual running time.



(a) Estimated/actual query time (s)



(b) Estimated/actual benefit

Figure 8: Cost model and benefit model validation

Figure 8(a) shows the relationship between the estimated cost and actual cost of range queries for various query size $s \times s$. The value of $s \times s$ varies from $0.0001 \times MBR(D)$ to $0.05 \times MBR(D)$, with $MBR(D)$ is the area of the input dataset. For each query, we compute the *estimated* running time using Equation 4 and measure the *actual* running time as we process the query on the partitioned data. Then, we plot all those queries on a scatter plot with the estimated and actual running time on the x and y-axes, respectively. From the trend lines, we observe that there is a linear correlation between estimated cost and actual cost with the slope being very close to 1.0. This observation indicates that our proposed cost model is accurate in reality. The correlation between the two axes is nearly 94%. In other words, the estimated cost can be used in the partitioning optimization process in order to minimize the actual cost of range query processing.

7.1.2 Reorganization benefit model validation. To verify the reorganization benefit model, we record the actual benefit (Equation 8) and estimated benefit (Equation 10) as we reorganize partitions using the three proposed methods, R*P, LSMP, and CBP. Figure 8(b) depicts a scatter plot with the estimated and actual benefits on the x and y axis, respectively. The correlation is 97% which indicates that the proposed estimated benefit is reliable to be integrated into our partition selection algorithm. This high correlation confirms that the estimated benefit can be used for partition selection step even with skewed datasets like MS-Buildings.

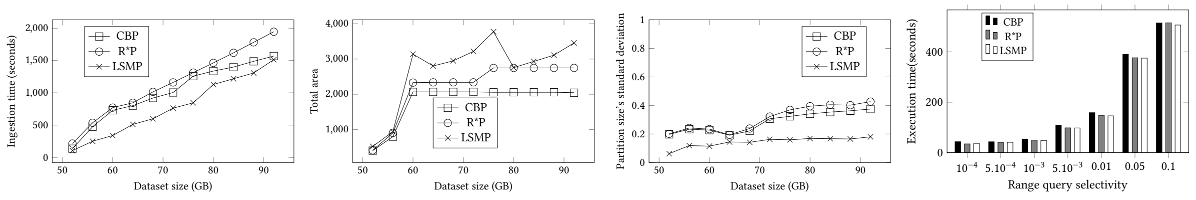
7.2 Performance of proposed partitioning algorithms

In this experiment, we compare the performance of proposed techniques in two workloads: insert-only and insert+delete. Figure 9 and 10 show the behavior of our proposed partitioning algorithms. We execute the experiment on MS-Buildings dataset with the comparison in ingestion performance, quality metrics and range query performance. To simulate the dataflow in Figure 3, we split the original MS-Buildings into batches of 8GB. Each batch cover a subarea of the entire dataset. This step increases the skewness of the data and guarantee that the distribution of the ingested data will be changing overtime. Figure 9(a) and 10(a) show the ingestion time for different partitioning techniques. One can observe that CBP outperforms R*P and is slower than LSMP in terms of ingestion time. By design, CBP and R*P have the same data flushing mechanism, which appends the new records to existing partitions. The difference lies the reorganization process where R*P splits each overflowing partition independently while CBP can reorganize many overlapping partitions together which reduces the number of required reorganization jobs. On the other hand, LSMP is different than CBP and R*P in both data flushing and reorganization process. First, the LSMP flushing step partitions the flushed data and writes it to a separate component on disk which is generally more efficient than appending to many files. Second, the reorganization process is triggered periodically based on the LSM merging policy and it always consists of one job that reorganizes all the selected components together. This explains why the update operation on LSMP is generally faster than CBP and R*P, since sometimes there is no merge operation triggered.

Figure 9(b) and 10(b) show the total area, as a quality measure, of different partitioning techniques in the same set of MS-Buildings batches. Since CBP is optimized for estimated range query cost, we expect that it creates partitions with good quality. R*P keeps splitting partitions so the total area will keep increasing as new batches coming. LSMP's total area increases as new components are added, and periodically drops when a compaction step is triggered which increases the partitioning quality. Figure 9(c) shows the partition load balance of the proposed techniques for the insert-only workload, which is the normalized standard deviation of partition size. LSMP is better than CBP and R*P since all of its components are optimally partitioned. In addition, CBP and LSMP's block utilization will be better than R*P for insert+delete workload as shown in Figure 10(c), since R*P can only split partitions into smaller partitions, but cannot merge them together when they become smaller after deletion. Overall, the partition quality of LSMP is not stable due to its merge policy. In particular, the partition quality will be good after a merge operations but gets worse as new components are flushed. To choose the best among all proposed partitioning techniques, we execute several batches of range query with different query sizes on the datasets partitioned by the proposed techniques. Figure 9(d) shows that all the techniques perform well for the insert-only workload. However, for the insert+delete workload in Figure 10(d), CBP is consistently better, especially for bigger data. Therefore, to keep the remaining experiments simple, we will use CBP as a representative of the three proposed techniques to compare with other state-of-the-art spatial partitioning systems.

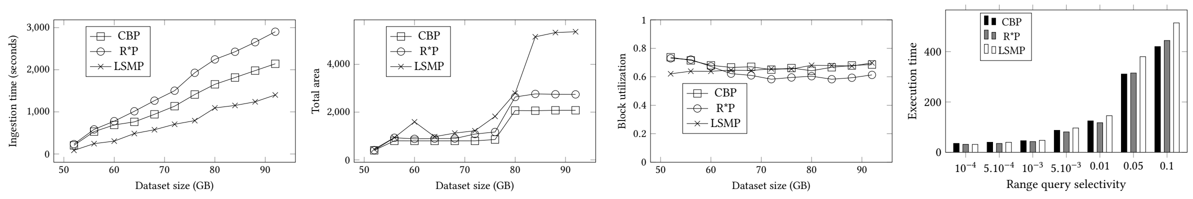
7.3 Comparison with state-of-the-art systems

This part compares the proposed CBP algorithm to existing record-level and block-level systems for big spatial data based on both the ingestion time and query execution time. We compare to GeoMesa and AsterixDB as record-level systems and to Sedona and Beast as block-level systems. GeoMesa is a geospatial database systems that is built on top of distributed databases, which is expected to work well with highly selective queries. We use XZ2-16bits [14] as the index scheme for spatial objects in GeoMesa which allows each partitioned file maintain a size of few KBs. AsterixDB is a big-data management system that uses record-level indexing with an R-tree implementation on-top of LSM components. Sedona and Beast only work with static data so we must rebuild the entire index after each batch. Similar to Sedona and Beast, the proposed methods is implemented in Spark. To make the experiments fair, we use HDFS as the storage layer for all systems, except AsterixDB, which comes with its own storage layer.



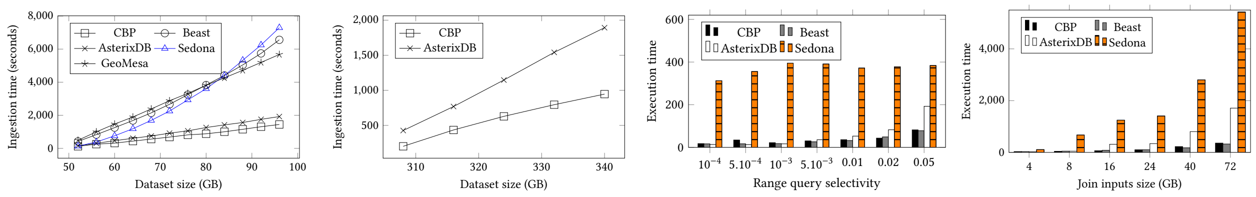
(a) Ingestion performance (b) Total area (c) Partition load balance (d) Range query performance

Figure 9: Performance of the three proposed implementations with insert-only workload



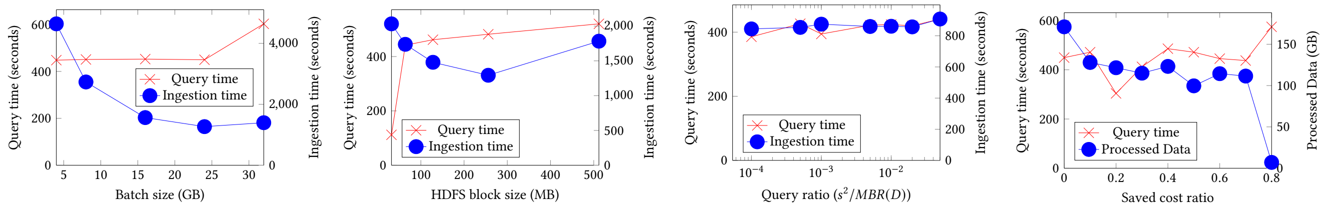
(a) Ingestion performance (b) Total area (c) Disk utilization (d) Range query performance

Figure 10: Performance of the three proposed implementations with insert-delete workload



(a) Ingestion performance (b) Scalability of proposed system (c) Range query performance (d) Spatial join performance

Figure 11: Performance comparison of CBP with existing techniques: AsterixDB, GeoMesa, Beast



(a) Effect of batch size (b) Effect of HDFS block size (c) Effect of query size s (d) Effect of saved cost ratio

Figure 12: Effect of different partitioning parameters in CBP

7.3.1 Ingestion performance. Figure 11(a) shows the ingestion performance in different systems. To measure the steady state performance when data is big, we start with a 48 GB data of the MS-Buildings dataset and append batches of 4GB. We measure the accumulated time to ingest these batches to the systems we are comparing. We observe that CBP outperforms other systems in terms of ingestion performance. Sedona and Beast are the slowest systems since they repartition all the data for each new batch so the ingestion time grows super linearly. On the other hand, AsterixDB and GeoMesa only insert the new records but they suffer from the overhead of record-level indexing where the index structure must find the exact position of each single record. CBP reaches a sweet spot since it adopts a block-level approach that has a lower overhead, yet, it does not require a complete reorganization as in Sedona and Beast. Since CBP and AsterixDB are clearly winners for ingestion workloads, we run another experiment starting with 300 GB of OSM-All Objects dataset. Then we append several batches of 8 GB data to verify that they are able to work on very large datasets as shown in Figure 11(b). We observe that CBP is still relatively faster than AsterixDB at this scale.

7.3.2 Range query performance. Figure 11(c) shows the performance of range query in different systems. In this experiment, we ingest MS-Buildings dataset in CBP, Beast, AsterixDB and Sedona. We omit the results of GeoMesa because it was way slower than all other techniques. After the data is fully ingested, we execute a batch of range queries with different query sizes and measure the total running time in seconds. We observe that AsterixDB is better than others with highly selective queries and the performance significantly drops for large size queries. This is a direct result of the record-level index scheme of AsterixDB which better supports highly-selective queries where the index can quickly locate the individual records in the result. As the query size increases, the cost of accessing records individually becomes too high. On the other hand, CBP and Beast show a reasonable performance for small queries and much better than AsterixDB for large queries since they can still use the partition information to reduce the number of accessed partitions and then simply scan the matching partitions in parallel which reduces the total processing time for range query.

7.3.3 Spatial join performance. In this experiment, we partition different batches of data which are extracted from MS-Buildings with size from 2GB to 64GB and OSM-Parks with size from 2GB to 8GB. We execute the spatial join algorithm which finds all the intersected pairs of two datasets. Figure 11(d) shows the spatial join query performance in CBP, Beast, AsterixDB and Sedona. In particular, we measure the total time to complete the query. We can easily observe that CBP and Beast outperform AsterixDB and Sedona in all join operations of large datasets, while AsterixDB is only the winner in the join of small datasets. Sedona is slow due to its design and implementation. The explanation is similar to the range query performance. Since AsterixDB partitions size is small, they will require more jobs to complete the same query when compared to block-level partitioning techniques. We also observe that Beast is only slightly faster than CBP because it always rebuilds the entire index so it becomes highly optimized. However, CBP can reach almost the same performance by smart selection of a few partitions to reorganize after each batch.

7.4 Effects of partitioning parameters on CBP

7.4.1 Effect of batch size. Figure 12(a) shows the effect of batch size on ingestion time and query performance in the CBP technique. We ingest 96 GB of the MS-Buildings dataset using CBP while varying the batch size from 4GB to 32GB. We use a set of 100 square range queries with various query ratios from 0.0001 to 0.05 for this and the following experiments. We observe that when the batch size increases, the ingestion time is reduced because fewer flush and reorganization steps are needed. At the same time, the query performance is not significantly affected because CBP can adjust the reorganization work to keep the index of high quality. However, when the batch size is very large, the range query performance starts to decrease because of the less frequent reorganization.

7.4.2 Effect of block size. This experiment measure the effect of HDFS block size to the partitioning quality. We ingest the MS-Buildings dataset with while varying the block size from 32 MB up to 512 MB. Figure 12(b) shows that the ingestion time will be high if the block size is too small or too large, while the range query performance is slower for large block size, which is expected since a query have to process more data in average. Based on this observation, we would suggest to use the block size 128 MB or 256 MB for spatial partitioning techniques on HDFS.

7.4.3 Effect of query size in the cost model. This experiment studies the effect of the parameter s in the cost model which represents the size of the square query range. We vary the parameter s while measuring the ingestion time and the query processing time for the same set of range queries used in Section 7.4.1. In Figure 12(c), we vary the query ratio ($s^2/MBR(D)$) from 10^{-4} to 0.05 where $MBR(D)$ is the area of the input MBR. As observed, both the ingestion time and processing time are relatively stable which indicates that the parameter s is easy to tune. This makes sense since optimizing the partitions for one query size is expected to make the partitions work well for other query sizes a well.

7.4.4 Effect of saved cost ratio. This experiment highlights the effect of changing the frequency of the reorganization step. In particular, we introduce a parameter called *saved cost ratio*, where CBP only triggers the reorganization process if the saved cost after reorganization is greater than that ratio. Figure 12(d) shows the query time and the amount of data being reorganized as the saved cost ratio is increased from 0 (always reorganize) to 0.8. As the ratio increases, the total amount of data being reorganized decreases because of fewer reorganization. On the other hand, the query time becomes significantly worse only for very high saved cost ratio. This indicates that we can further reduce the reorganization time while maintaining the same query performance and we plan to further study this effect in the future.

8 CONCLUSION

This paper proposed a generic framework for incremental partitioning of big spatial data. We used this framework to implement three incremental spatial partitioning techniques to show its feasibility. Then, we provided a deeper study to the partition optimization problem and proved its NP-hardness. Based on this, we split it into two smaller problems, partition selection and partition reorganization. We then showed that the partition selection problem is crucial for the partition quality. To solve the partition selection problem, we proposed a new range query cost model and used it to build an approximate greedy algorithm for the partition selection problem. Finally, we carried out an extensive experimental evaluation using large scale real data to evaluate the efficiency of the proposed work. The experiments showed that the proposed techniques minimize the partition construction time while maintaining high quality partitions. The source code and datasets are made publicly available for reproducibility. In the future, we can extend the proposed work to be aware of the workload by using a workload-aware cost model while reusing the benefit model and CBP algorithm.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation (NSF) under grant IIS-1838222, IIS-2046236, IIS-1954644, IIS-1901379, SES-1831615 and CNS-1924694.

REFERENCES

[1] accumulo [n.d.]. Apache Accumulo. https://accumulo.apache.org/. Visited on 15-Sep-2021.

[2] Daniar Achakeev, Bernhard Seeger, and Peter Widmayer. 2012. Sort-based query-adaptive loading of r-trees. In *Proceedings of the 21st ACM international conference on Information and knowledge management.* 2080–2084.

[3] Swarup Acharya, Viswanath Poosala, and Sridhar Ramaswamy. 1999. Selectivity Estimation in Spatial Databases. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA.* 13–24.

[4] Sattam Alsubaiee et al. 2014. Storage management in AsterixDB. *Proceedings of the VLDB Endowment* 7, 10 (2014), 841–852.

[5] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, et al. 2014. Asterixdb: A scalable, open source BDMS. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1905–1916.

[6] Ahmed M. Aly, Hazem Elmeleegy, Yan Qi, and Walid G. Aref. 2016. Kangaroo: Workload-Aware Processing of Range Data and Range Queries in Hadoop. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining, San Francisco, CA, USA, February 22-25, 2016*, Paul N. Bennett, Vanja Josifovski, Jennifer Neville, and Filip Radlinski (Eds.). ACM, 397–406. https://doi.org/10.1145/2835776.2835841

[7] Ahmed M. Aly, Ahmed R. Mahmood, Mohamed S. Hassan, Walid G. Aref, Mourad Ouzzani, Hazem Elmeleegy, and Thamir Qadah. 2015. AQWA: Adaptive Query-Workload-Aware Partitioning of Big Spatial Data. *Proc. VLDB Endow.* 8, 13 (2015), 2062–2073. https://doi.org/10.14778/2831360.2831361

[8] amazons3 [n.d.]. Amazon S3. https://aws.amazon.com/s3/. Visited on 15-Sep-2021.

[9] asterixdb [n.d.]. Apache AsterixDB. https://asterixdb.apache.org. Visited on 15-Sep-2021.

[10] azure [n.d.]. Microsoft Azure. https://azure.microsoft.com. Visited on 15-Sep-2021.

[11] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. 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* (Atlantic City, New Jersey, USA) *(SIGMOD '90)*. 322–331.

[12] Alberto Belussi and Christos Faloutsos. 1995. Estimating the Selectivity of Spatial Queries Using the 'Correlation' Fractal Dimension. In *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland.* 299–310.

[13] Jon Louis Bentley. 1979. Multidimensional Binary Search Trees in Database Applications. *IEEE Trans. Software Eng.* 5, 4 (1979), 333–340. https://doi.org/10.1109/TSE.1979.234200

[14] Christian BÖxhm, Gerald Klump, and Hans-Peter Kriegel. 1999. Xz-ordering: A space-filling curve for objects with spatial extension. In *International Symposium on Spatial Databases.* Springer, 75–90.

[15] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 4.

[16] Yong-Jin Choi and Chin-Wan Chung. 2002. Selectivity estimation for spatio-temporal queries to moving objects. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002.* 440–451.

[17] Carlo Curino, Evan Philip Charles Jones, Yang Zhang, and Samuel R Madden. 2010. Schism: a workload-driven approach to database replication and partitioning. (2010).

[18] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[19] Gisbert Dröge and Hans-Jörg Schek. 1993. Query-adaptive data space partitioning using variable-size storage clusters. In *International Symposium on Spatial Databases.* Springer, 337–356.

[20] Ahmed Eldawy, Louai Alarabi, and Mohamed F Mokbel. 2015. Spatial partitioning techniques in SpatialHadoop. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1602–1605.

[21] Ahmed Eldawy, Vagelis Hristidis, Saheli Ghosh, Majid Saeedan, Akil Sevim, A.B. Siddique, Samriddhi Singla, Ganesh Sivaram, Tin Vu, and Yaming Zhang. 2021. Beast: Scalable Exploratory Analytics on Spatio-temporal Data. In *CIKM.* ACM.

[22] Ahmed Eldawy, Yuan Li, Mohamed F Mokbel, and Ravi Janardan. 2013. CG_Hadoop: computational geometry in MapReduce. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems.* ACM, 294–303.

[23] Ahmed Eldawy and Mohamed F Mokbel. 2015. Spatialhadoop: A mapreduce framework for spatial data. In *2015 IEEE 31st International Conference on Data Engineering.* IEEE, 1352–1363.

[24] Ahmed Eldawy and Mohamed F. Mokbel. 2016. The Era of Big Spatial Data: A Survey. *Foundations and Trends in Databases* 6, 3-4 (2016), 163–273.

[25] Ahmed Eldawy and Mohamed F. Mokbel. 2019. Boundaries of parks and green areas from all over the world as extracted from OpenStreetMap. https://doi.org/10.6086/N1RX994T#mbr=9qh2s0vm,9qhf060f Retrieved from UCR-STAR https://star.cs.ucr.edu/?OSM2015/parks&d#mbr=9qh2s0vm,9qhf060f.

[26] Ahmed Eldawy, Ibrahim Sabek, Mostafa Elganainy, Ammar Bakeer, Ahmed Abdelmotaleb, and Mohamed F. Mokbel. 2017. Sphinx: Empowering Impala for Efficient Execution of SQL Queries on Big Spatial Data. Arlington, VA, 65–83.

[27] EOSDIS 2017. The Common Metadata Repository: The Foundation of NASA's Earth Observation Data. https://earthdata.nasa.gov/the-common-metadata-repository.

[28] Raphael A. Finkel and Jon Louis Bentley. 1974. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Inf.* 4 (1974), 1–9. https://doi.org/10.1007/BF00288933

[29] Anthony Fox, Chris Eichelberger, James Hughes, and Skylar Lyon. 2013. Spatio-temporal indexing in non-relational distributed databases. In *Big Data, 2013 IEEE International Conference on.* IEEE, 291–299.

[30] Saheli Ghosh, Tin Vu, Mehrad Amin Eskandari, and Ahmed Eldawy. 2019. UCR-STAR: The UCR spatio-temporal active repository. *SIGSPATIAL Special* 11, 2 (2019), 34–40.

[31] 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.

[32] hbase [n.d.]. Apache HBase. http://hbase.apache.org/. Visited on 15-Sep-2021.

[33] Yaobin He, Haoyu Tan, Wuman Luo, Huajian Mao, Di Ma, Shengzhong Feng, and Jianping Fan. 2011. MR-DBSCAN: An Efficient Parallel Density-Based Clustering Algorithm Using MapReduce. Tainan, Taiwan, 473–480.

[34] Erik G. Hoel and Hanan Samet. 1994. Performance of Data-Parallel Spatial Operations. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile.* 156–167.

[35] James N Hughes, Andrew Annex, Christopher N Eichelberger, Anthony Fox, Andrew Hulbert, and Michael Ronquest. 2015. Geomesa: a distributed architecture for spatio-temporal fusion. In *SPIE Defense+ Security.* International Society for Optics and Photonics, 94730F–94730F.

[36] Ibrahim Kamel and Christos Faloutsos. 1994. Hilbert R-tree: An Improved R-tree using Fractals. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile.* 500–509. http://www.vldb.org/conf/1994/P500.PDF

[37] Yuan Li, Ahmed Eldawy, Jie Xue, Nadezda Knorozova, Mohamed F. Mokbel, and Ravi Janardan. 2019. Scalable Computational Geometry in MapReduce. (16 Jan 2019). https://doi.org/10.1007/s00778-018-0534-5

[38] Jiamin Lu and Ralf Hartmut Guting. 2012. Parallel secondo: boosting database engines with hadoop. In *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on.* IEEE, 738–743.

[39] Peng Lu, Gang Chen, Beng Chin Ooi, Hoang Tam Vo, and Sai Wu. 2014. ScalaGiST: Scalable Generalized Search Trees for MapReduce Systems [Innovative Systems Paper]. *PVLDB* 7, 14 (2014), 1797–1808.

[40] Wei Lu, Yanyan Shen, Su Chen, and Beng Chin Ooi. 2012. Efficient Processing of k Nearest Neighbor Joins using MapReduce. 5, 10 (2012), 1016–1027.

[41] Ahmed R. Mahmood, Ahmed M. Aly, Thamir Qadah, El Kindi Rezig, Anas Daghistani, Amgad Madkour, Ahmed S. Abdelhamid, Mohamed S. Hassan, Walid G. Aref, and Saleh M. Basalamah. 2015. Tornado: A Distributed Spatio-Textual Stream Processing System. 8, 12 (2015), 2020–2023.

[42] Microsoft. 2020. Computer generated building footprints in all 50 US states. https://doi.org/10.6086/N1C24TGK Retrieved from UCR-STAR https://star.cs.ucr.edu/?MSBuildings&d.

[43] mongodb [n.d.]. MongoDB. https://www.mongodb.com/. Visited on 15-Sep-2021.

[44] Jan Kristof Nidzwetzki and Ralf Hartmut Güting. 2018. BBoxDB-A Scalable Data Store for Multi-Dimensional Big Data. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management.* ACM, 1867–1870.

[45] Jürg Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. 1984. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Trans. Database Syst.* 9, 1 (1984), 38–71.

[46] Shoji Nishimura, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2013. MD-HBase: design and implementation of an elastic data infrastructure for cloud-scale location services. *Distributed and Parallel Databases* 31, 2 (2013), 289–319.

[47] Matthaios Olma, Manos Karpathiotakis, Ioannis Alagiannis, Manos Athanassoulis, and Anastasia Ailamaki. 2017. Slalom: Coasting through raw data via adaptive partitioning and indexing. *Proceedings of the VLDB Endowment* 10, 10 (2017), 1106–1117.

[48] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.* 33, 4 (1996), 351–385.

[49] postgis [n.d.]. PostGIS. https://postgis.net/. Visited on 15-Sep-2021.

[50] Ibrahim Sabek and Mohamed F. Mokbel. 2017. On Spatial Joins in MapReduce. Redondo Beach, CA, 21:1–21:10.

[51] Hanan Samet. 1984. The Quadtree and Related Hierarchical Data Structures. *ACM Comput. Surv.* 16, 2 (1984), 187–260.

[52] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) (MSST '10)*. IEEE Computer Society, Washington, DC, USA, 1–10. https://doi.org/10.1109/MSST.2010.5496972

[53] spatiallite [n.d.]. SpatialLite. https://www.gaia-gis.it/fossil/libspatialite/index. Visisted on 15-Sep-2021.

[54] Yannis Theodoridis and Timos Sellis. 1996. A model for the prediction of R-tree performance. In *Proceedings of the fifteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM, 161–171.

[55] twitterstatistic 2018. Twitter Usage Statistics. http://www.internetlivestats.com/twitter-statistics/. Visisted on 15-Sep-2021.

[56] Kostas Tzoumas, Man Lung Yiu, and Christian S Jensen. 2009. Workload-aware indexing of continuously moving objects. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1186–1197.

[57] Tin Vu and Ahmed Eldawy. 2018. R-Grove: growing a family of R-trees in the big-data forest. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 532–535.

[58] Tin Vu and Ahmed Eldawy. 2020. R*-Grove: Balanced Spatial Partitioning for Large-Scale Datasets. *Frontiers in Big Data* 3 (2020), 28. https://doi.org/10.3389/fdata.2020.00028

[59] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. 2016. Simba: Efficient In-Memory Spatial Analytics. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. 1071–1085.

[60] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. 2015. Geospark: A cluster computing framework for processing large-scale spatial data. In *Proceedings of the 23rd SIGSPATIAL international conference on advances in geographic information systems*. 1–4.

[61] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. 2016. A demonstration of GeoSpark: A cluster computing framework for processing big spatial data. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*. 1410–1413.

[62] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*. USENIX Association, 15–28.

[63] Yaming Zhang and Ahmed Eldawy. 2021. OpenStreetMap All Objects. Retrieved from UCR-STAR https://star.cs.ucr.edu/?osm21/all_objects&d.