Pyro: A Spatial-Temporal Big-Data Storage System

Shen Li* Shaohan Hu* Raghu Ganti† Mudhakar Srivatsa† Tarek Abdelzaher*
*University of Illinois at Urbana-Champaign †IBM Research

Abstract

With the rapid growth of mobile devices and applications, geo-tagged data has become a major workload for big data storage systems. In order to achieve scalability, existing solutions build an additional index layer above general purpose distributed data stores. Fulfilling the semantic level need, this approach, however, leaves a lot to be desired for execution efficiency, especially when users query for moving objects within a high resolution geometric area, which we call geometry queries. Such geometry queries translate to a much larger set of range scans, forcing the backend to handle orders of magnitude more requests. Moreover, spatial-temporal applications naturally create dynamic workload hotspots[1], which pushes beyond the design scope of existing solutions. This paper presents Pyro, a spatial-temporal big-data storage system tailored for high resolution geometry queries and dynamic hotspots. Pyro understands geometries internally, which allows range scans of a geometry query to be aggregately optimized. Moreover, Pyro employs a novel replica placement policy in the DFS layer that allows Pyro to split a region without losing data locality benefits. Our evaluations use NYC taxi trace data and an 80-server cluster. Results show that Pyro reduces the response time by 60X on $1km \times 1km$ rectangle geometries compared to the state-of-the-art solutions. Pyro further achieves 10X throughput improvement on $100m \times 100m$ rectangle geometries[2].

1 Introduction

The popularity of mobile devices is growing at an unprecedented rate. According to the report published by the United Nations International Telecommunication Union [1], mobile penetration rates are now about equal to the global population. Thanks to positioning modules in mobile devices, a great amount of information generated today is tagged with geographic locations. For example, users can share tweets and Instagram images with location information with family and friends; taxi companies collect pick-up and drop-off events data with geographic location information as well. The abundances of geo-tagged data give birth to a whole range of applications that issue spatial-temporal queries. These queries, which we call geometry queries, request information about moving objects within a user-defined geometric area. Despite the urgent need, no existing systems manage to meet both the scalability and efficiency requirements for spatial-temporal data. For example, geospatial databases [2] are optimized for spatial data, but usually fall short on scalability on handling big-data applications, whereas distributed data stores [3, 4, 5, 6] scale well but quite often yield inefficiencies when dealing with geometry queries.

Distributed data stores, such as HBase [3], Cassandra [4], and DynamoDB [5], have been widely used for big-data storage applications. Their key distribution algorithms can be categorized into two classes: random partitioning and ordered partitioning. The former randomly distributes keys into servers, while the latter divides the key space into subregions such that all keys in the same subregion are hosted by the same server. Compared to random partitioning, ordered partitioning considerably benefits range scans, as querying all servers in the cluster can then be avoided. Therefore, existing solutions for spatial-temporal big-data applications, such as MD-HBase [7], and ST-HBase [8], build index layers above the ordered-partitioned HBase to translate a geometry query into a set of range scans. Then, they submit those range scans to HBase, and aggregate the returned data from HBase to answer the query source, inheriting scalability properties from HBase. Although these solutions fulfill the semantic level requirement of spatial-temporal applications, moving hotspots and large geometry queries still cannot be handled efficiently.

Spatial-temporal applications naturally generate moving workload hotspots. Imagine a million people simultaneously whistle taxis after the New Year's Eve at NYC's Times Square. Or during every morning rush hour, people driving into the city central business district search for the least congested routes. Ordered partitioning data stores usually mitigate hotspots by splitting an overloaded region into multiple daughter regions, which can then be moved into different servers. Nevertheless, as region data may still stay in the parent region's server, the split operation prevents daughter regions from enjoying data locality benefits. Take HBase as an example. Region servers in HBase usually co-locate with HDFS datanodes. Under this deployment, one replica of all region data writes to the region server's storage disks, which allows get/scan requests to be served using local

[1] The hotspot in this paper refers to a geographic region that receives a large amount of geometry queries within a certain amount of time.

[2] The reason of using small geometries in this experiment is that the baseline solution results in excessively long delay when handling even a single large geometry.

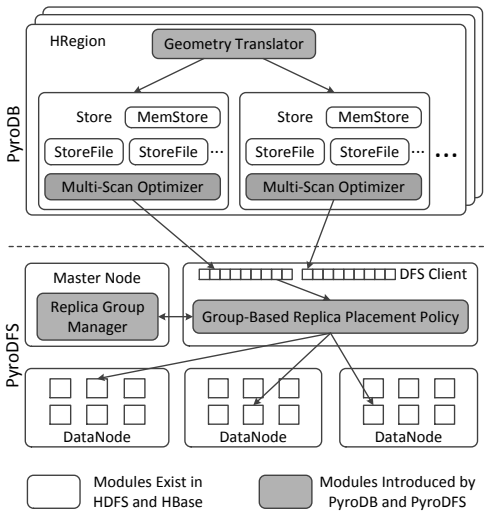


Figure 1: Pyro Architecture

data. Other replicas spread randomly in the entire cluster. Splitting and moving a region into other servers disable data locality benefits, forcing daughter regions to fetch data from remote servers. Therefore, moving hotspots often lead to performance degradation.

In this paper, we present Pyro, a holistic spatial-temporal big-data storage system tailored for high resolution geometry queries and moving hotspots. Pyro consists of PyroDB and PyroDFS, corresponding to HBase and HDFS respectively. This paper makes three major contributions. First, PyroDB internally implements Moore encoding to efficiently translate geometry queries into range scans. Second, PyroDB aggregately minimizes IO latencies of the multiple range scans generated by the same geometry query using dynamic programming. Third, PyroDFS employs a novel DFS block grouping algorithm that allows Pyro to preserve data locality benefits when PyroDB splits regions during hotspots dynamics. Pyro is implemented by adding 891 lines of code into Hadoop-2.4.1, and another 7344 lines of code into HBase-0.99. Experiments using NYC taxi dataset [9, 10] show that Pyro reduces the response time by 60X on $1km \times 1km$ rectangle geometries. Pyro further achieves 10X throughput improvement on $100m \times 100m$ rectangle geometries.

The remainder of this paper is organized as follows. Section 2 provides background and design overview. Then, major designs are described in Section 3. Implementations and evaluations are presented in Sections 4 and 5 respectively. We survey related work in Section 6. Finally, Section 7 concludes the paper.

2 Design Overview

Pyro consists of PyroDB and PyroDFS. The design of PyroDB and PyroDFS are based on HBase and HDFS respectively. Figure 1 shows the high-level architecture, where shaded modules are introduced by Pyro.

2.1 Background

HDFS [11] is an open source software based on GFS [12]. Due to its prominent fame and universal deployment, we skip the background description.

HBase is a distributed, non-relational database running on top of HDFS. Following the design of BigTable [13], HBase organizes data into a 3D table of rows, columns, and cell versions. Each column belongs to a column family. HBase stores the 3D table as a key-value store. The key consists of row key, column family key, column qualifier, and timestamp. The value contains the data stored in the cell.

In HBase, the entire key space is partitioned into regions, with each region served by an HRegion instance. HRegion manages each column family using a *Store*. Each Store contains one MemStore and multiple StoreFiles. In the write path, the data first stays in the MemStore. When the MemStore reaches some pre-defined flush threshold, all key-value pairs in the MemStore are sorted and flushed into a new StoreFile in HDFS. Each StoreFile wraps an HFile, consisting of a series of data blocks followed by meta blocks. In this paper, we use meta blocks to refer to all blocks that store meta, data index, or meta index. In the read path, a request first determines the right HRegions to query, then it searches all StoreFiles in those regions to find target key-value pairs.

As the number of StoreFiles increases, HBase merges them into larger StoreFiles to reduce the overhead of read operations. When the size of a store increases beyond a threshold, its HRegion splits into two daughter regions, with each region handles roughly half of its parent's key-space. The two daughter regions initially create reference files pointing back to StoreFiles of their past parent region. This design postpones the overhead of copying region data to daughter region servers at the cost of losing data locality benefits. The next major compaction materializes the reference files into real StoreFiles.

HBase has become a famous big-data storage system for structured data [14], including data for location-based services. Many location-based services share the same request primitive that queries information about moving objects within a given geometry, which we call geometry queries. Unfortunately, HBase suffers inefficiencies when serving geometry queries. All cells in HBase are ordered based on their keys in a one-dimensional space. Casting a geometry into that one-dimensional space inevitably results in multiple disjoint range scans. HBase handles those range scans individually, preventing queries to be aggregately optimized. Moreover, location-based workloads naturally create moving hotspots in the backend, requiring responsive resource elasticity in every HRegion. HBase handles workload hotspots by efficiently splitting regions, which sacrifices data locality benefits for newly created daugh-

ter regions. Without data locality, requests will suffer increased response time after splits. Above observations motivate us to design Pyro, a data store specifically tailored for geometry queries.

2.2 Architecture

Figure 1 shows the high-level architecture of Pyro. Pyro internally uses Moore encoding algorithm [15, 16, 17, 18] to cast two-dimensional data into one-dimensional Moore index, which is enclosed as part of the row key. For geometry queries, the *Geometry Translator* module first applies the same Moore encoding algorithm to calculate scan ranges. Then, the *Multi-Scan Optimizer* computes the optimal read strategy such that the IO latency is minimized. Sections 3.1 and 3.2 present more details.

Pyro relies on the group-based replica placement policy in PyroDFS to guarantee data locality during region splits. To achieve that, each StoreFile is divided into multiple shards based on user-defined pre-split keys. Then, Pyro organizes DFS replicas of all shards into elaborately designed groups. Replicas in the same group are stored in the same physical server. After one or multiple splits, each daughter region is guaranteed to find at least one replica of all its region data within one group. To preserve data locality, Pyro just need to move the daughter region into the physical server hosting that group. The details of group-based replica placement are described in section 3.3.

Pyro makes three major contributions:

- Geometry Translation: Apart from previous solutions that build an index layer above HBase, Pyro internally implements efficient geometry translation algorithms based on Moore encoding. This design allows Pyro to optimize a geometry query by globally processing all its range scans together.
- Multi-Scan Optimization: After geometry translation, the multi-scan optimizer aggregately processes the generated range scans to minimize the response time of the geometry query. By using storage media performance profiles as inputs, the multi-scan optimizer employs a dynamic programming algorithm to calculate the optimal HBase blocks to fetch.
- Block Grouping: To deal with moving hotspots, Pyro relies on a novel data block grouping algorithm in the DFS layer to split a region quickly and efficiently, while preserving data locality benefits. Moreover, by treating meta block and data block differently, block grouping helps to improve Pyro's fault tolerance.

3 System Design

We first present the geometry translation and multi-scan optimization in Sections 3.1 and 3.2 respectively. These two solutions help to efficiently process geometry

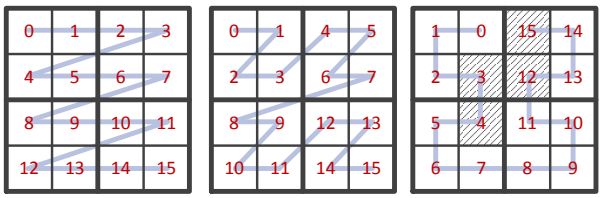


Figure 2: Spatial Encoding Algorithms of Resolution 2

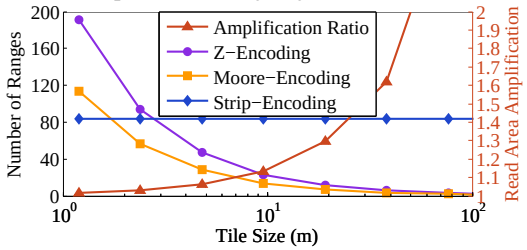


Figure 3: Translate Geometry to Key Ranges

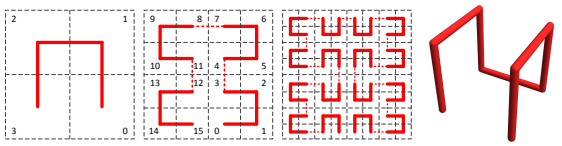
queries. Then, Section 3.3 describes how Pyro handles moving hotspots with the block grouping algorithm.

3.1 Geometry Translation

In order to store spatial-temporal data, Pyro needs to cast 2D coordinates (x, y) into the one-dimensional key space. A straightforward solution is to use a fixed number of bits to represent x, and y, and append x after y to form the spatial key. This leads to the *Strip*-encoding as shown in Figure 2 (a). Another solution is to use ZOrder-encoding [7] that interleaves the bits of x and y. An example is illustrated in Figure 2 (b). These encoding algorithms divide the 2D space into $m \times m$ tiles, and index each tile with a unique ID. The tile is the spatial encoding unit as well as the unit of range scans. We define the resolution as $\log_2(m)$, which is the minimum number of bits required to encode the largest value of x and y.

In most cases, encoding algorithms inevitably break a two-dimensional geometry into multiple key ranges. Therefore, each geometry query may result in multiple range scans. Each range scan requires a few indexing, caching, and disk operations to process. Therefore, it is desired to keep the number of range scans low. We carry out experiments to evaluate the number of range scans that a geometry query may generate. The resolution ranges from 25 to 18 over the same set of randomly generated disk-shaped geometry queries with $100m$ radius in a $40,000,000m \times 40,000,000m$ area. The corresponding tile size ranges from 1.2m to 153m. Figure 3 shows the number of range scans generated by a single geometry query under different resolutions. It turns out that Strip-encoding and ZOrder-encoding translate a single disk geometry to a few tens of range scans when the tile size falls under 20m.

To reduce the number of range scans, we developed the Geometry Translator module. The module employs the *Moore*-Encoding algorithm which is inspired by the Moore curve from the space-filling curve fam-



(a) (b)

Figure 4: Moore Curve

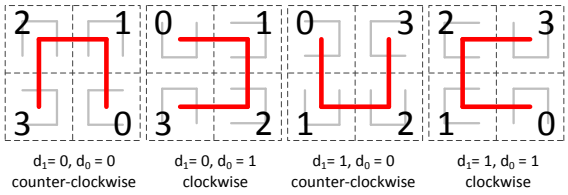


Figure 5: Moore Encoding Unit

ily [15, 16, 17, 18]. A simple example is shown in Figure 2 (c). A Moore curve can be developed up to any resolution. As shown in Figure 4 (a), resolutions 1 and 2 of Moore encoding are special cases. The curve of resolution 1 is called a unit component. In order to increase the resolution, the Moore curve expands each unit component according to a fixed strategy as shown in Figure 5. Results plotted in Figure 3 show that Moore-Encoding helps to reduce the number of range scans by 40% when compared to ZOrder-Encoding. Moore curves may generalize to higher dimensions [19], Figure 4 (b) depicts the simplest 3D Moore curve of resolution 1. Implementations of the Moore encoding algorithm are presented in Section 4.

3.2 Multi-Scan Optimization

The purpose of multi-scan optimization is to reduce read amplification. Below, we first describe the phenomenon of read amplification, and then we present our solution to this problem.

3.2.1 Read Amplification

When translating geometry queries, range scans are generated respecting tile boundaries at the given resolution. But, tile boundaries may not align with the geometry query boundary. In order to cover the entire geometry, data from a larger area is fetched. We call this phenomenon *Read Area Amplification*. Figure 3 plots the curve of read area amplification ratio, which is quantitatively defined as the total area of fetched tiles over the area of the geometry query. The curves show that, solely tuning the resolution cannot achieve both a small number of range scans and a low ratio of read area amplification. For example, as shown in Figure 3, restricting each geometry query to generate less than 10 scans forces Pyro to fetch data from a 22% larger area. On the other hand, limiting the area amplification ratio to less than 5% leads to more than 30 range scans per geometry query. The problem gets worse for larger geometries.

Moreover, encoding tiles are stored into fixed-size DB blocks on disks, whereas DB blocks ignore the boundaries of encoding tiles. An entire DB block has to be loaded even when there is only one requested key-value pair fallen in that DB Block, which we call the *Read Volume-Amplification*. Please notice that, DB blocks are different from DFS blocks. DB blocks are the minimum read/write units in PyroDB (similar to HBase). One DB block is usually only a few tens of KiloBytes. In contrast, a DFS block is the minimum replication unit in PyroDFS (similar to HDFS). DFS blocks are orders of magnitudes larger than DB blocks. For example, the default PyroDFS block size is 64MB, which is 1024 times larger than the default PyroDB block size.

Besides read area and volume amplifications, using a third-party indexing layer may also force the data store to unnecessarily visit a DB block multiple times, especially for high resolution queries. We call it the *Redundant Read Phenomenon*. Even though a DB block can be cached to avoid disk operations, the data store still needs to traverse DB block's data structure to fetch the requested key-value pairs. Therefore, Moore encoding algorithm alone is not enough to guarantee the efficiency.

For ease of presentation, we use the term *Read Amplification* to summarize the read area amplification, read volume amplification, and redundant read phenomena. Read amplification may force a geometry query to load a significant amount of unnecessary data as well as visiting the same DB block multiple times, leading to a much longer response time. In the next section, we present techniques to minimize the penalty of read amplification.

3.2.2 An Adaptive Aggregation Algorithm

According to Figure 3, increasing the resolution helps to alleviate read area amplification. Using smaller DB block sizes reduces read volume amplification. However, these changes require Pyro to fetch significantly more DB blocks, pushing disk IO to become a throughput bottleneck. In order to minimize the response time, Pyro optimizes all range scans of the same geometry query aggregately, such that multiple DB blocks can be fetched within fewer disk read operations. There are several reasons for considering IO optimizations in the DB layer rather than relying on asynchronous IO scheduling in the DFS layer or the OS layer. First, issuing a DFS read request is not free. As a geometry query may potentially translate into a large number of read operations, maintaining those reads alone elicits extra overhead in all three layers. Second, performance of existing IO optimizations in lower layers depend on the timing and ordering of request submissions. Enforcing the perfect request submission ordering in the Geometry Translator is not any cheaper than directly performing the IO optimization in PyroDB. Third, as PyroDB servers have the global knowledge about all p-reads from the same geometry request, it is the natural place to implement IO optimizations.

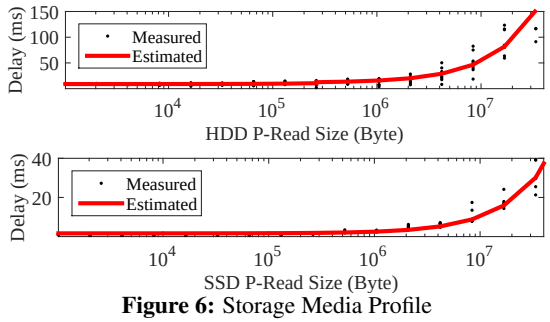


Figure 6: Storage Media Profile

Pyro needs to elaborately tune the trade-off between unnecessarily reading more DB blocks and issuing more disk seeks. Figure 6 shows the profiling results of Hadoop-2.4.1 position read (p-read) performance on a 7,200RPM Seagate hard drive and a Samsung SM0256F Solid State Drive respectively. In the experiment, we load a 20GB file into the HDFS, and measure the latency of p-read operations of varies sizes at random offsets. The disk seek delay dominates the p-read response time when reading less than 1MB data. When the size of p-read surpasses 1MB, the data transmission delay starts to make a difference. A naïve solution calculates the disk seek delay and the per-block transmission delay, and directly compares whether reading the next unnecessary block helps to reduce response time. However, the system may run on different data storage media, including hard disk drives, solid state drives, or even remote cloud drives. There is no guarantee that all media share the same performance profile. Such explicit seek delay and transmission delay may not even exist.

In order to allow the optimized range scan aggregation to work for a broader scenarios, we propose the Adaptive Aggregation Algorithm (A^3). A^3 uses the p-read profiling result to estimate delay of p-read operations. The profiling result contains the p-read response time of various sizes. A^3 applies interpolation to fill in gaps between profiled p-read sizes. This design allows the A^3 algorithm to work for various storage media.

Before diving into algorithm details, we present the abstraction of the block aggregation problem. Suppose a geometry query hits shaded tiles (3, 4, 12, 15) in Fig 2 (c). For the sake of simplicity, assume that DB blocks align perfectly with encoding tiles, one block per tile. Figure 7 shows the block layout in the StoreFile. A^3 needs to determine what block ranges to fetch in order to cover all requested blocks, such that the response time of the geometry query is minimized. In this example, let us further assume each block is 64KB. According to the profiling result shown in Figure 6, reading one block takes about 9 ms, four blocks takes 14 ms, while reading thirteen blocks takes 20 ms. Therefore, the optimal solution reads blocks 3-15 using one p-read operation.

A^3 works as follows. Suppose a geometry query translates to a set $\mathbf{Q}$ of range scans. Block indices help to convert those range scans into another set $\mathbf{B}'$ of blocks, sorted in the ascending order of their offsets. By removing all cached blocks from $\mathbf{B}'$, we get set $\mathbf{B}$ of n requested but not cached blocks. Define $\mathbf{S}[i]$ as the estimated minimum delay of loading the first i blocks. Then, the problem is to solve $\mathbf{S}[n]$. For any optimal solution, there must exist a k, such that blocks k to n are fetched using a single p-read operation. In other words, $\mathbf{S}[n] = \mathbf{S}[k-1] + \text{ESTIMATE}(k,n)$, where $\text{ESTIMATE}(k,n)$ estimates the delay of fetching blocks from k to n together based on the profiling result. Therefore, starting from $\mathbf{S}[0]$, A^3 calculates $\mathbf{S}[i]$ as $\min\{\mathbf{S}[k-1] + \text{ESTIMATE}(k,i) | 1 \le k \le i\}$. The pseudo code of A^3 is presented in Algorithm 1.

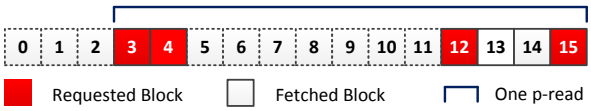


Figure 7: Block Layout in a StoreFile

Algorithm 1: A^3 Algorithm

```
   Input: blocks to fetch sorted by offset B
   Output: block ranges to fetch R
1  S ← an array of size |B|; initialize to ∞
2  P ← an array of size |B|; S[0] ← 0
3  for i ← 1 ~ |B| do
4      for j ← 0 ~ i − 1 do
5          k = i − j;    s ← ESTIMATE(k, i) + S[k − 1]
6          if s < S[i] then
7              S[i] ← s;    P[i] ← k
8  i ← |B|; R ← ∅
9  while i > 0 do
10     R ← R ∪ (P[i], i);    i ← P[i] − 1
11 return R
```

In Algorithm A^3, the nested loop between line $3-7$ leads to $O(|\mathbf{B}|^2)$ computational complexity. If $\mathbf{B}$ is large, the quadratic computational complexity explosion can be easily mitigated by setting an upper bound on the position read size. For example, for the hard drive profiled in Figure 6, fetching 10^7 bytes result in about the same delay as fetching 5×10^6 bytes twice. Therefore, there is no need to issue position read larger than 5×10^6 bytes. If block size is set to $64KB$, the variable j on the 5^{th} line in Algorithm 1 only needs to loop from 0 to 76, resulting in linear computational complexity.

3.3 Block Grouping

Moore encoding concentrates range scans of one geometry query into fewer servers. This may lead to performance degradation when spatial-temporal hotspots exist. To handle moving hotspots, a region needs to gracefully split itself to multiple daughters to make use of resources on multiple physical servers. Later, those daughter regions may merge back after their workloads shrink.

In HBase, the split operation creates two daughter regions on the same physical server, each owning reference

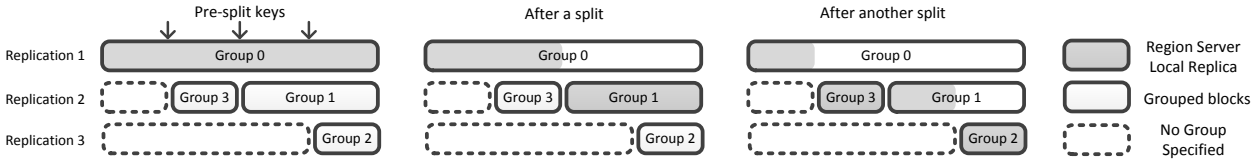


Figure 8: Split Example

files pointing back to StoreFiles of their parent region. Daughter regions are later moved onto other servers during the next cluster balance operation. Using reference files on one hand helps to keep the split operation light, but on the other hand prevents daughter regions from taking advantage of data locality benefits. Because, after leaving the parent region's server, the two daughter regions can no longer find their region data in their local disks until daughters' reference files are materialized. HBase materializes reference files during the next major compaction, which executes at a very low frequency (*e.g.*, once a day). Forcing earlier materialization does not solve the problem. It could introduce even more overhead to the already-overwhelmed region, as materialization itself is a heavy operation.

An ideal solution should keep both split and materialization operations light, allowing the system to react quickly when a hotspot emerges. Below, we present our design to achieve such ideal behaviors.

3.3.1 Group Based Replica Placement

Same as HBase, Pyro suggests users to perform pre-split based on expected data distribution to gain initial load balancing among region servers. Pyro relies on the expected data distribution to create more splitting keys for potential future splits. Split keys divide StoreFiles into shards, and help to organize DFS block replicas into replica groups. PyroDFS guarantees that DFS blocks respect predefined split keys. To achieve that, PyroDFS stops writing into the current DFS block and start a new one as soon as it reaches a predefined split key. This design relies on the assumption that, although moving hotspots may emerge in spatial-temporal applications, the long-round popularity of different geographic regions changes slowly. Results presented in evaluation Section 5.1 confirm the validity of this assumption.

Assume blocks are replicated r times and there are $2^{r-1}-1$ predefined split keys within a given region. Split keys divide the region key space into 2^{r-1} shards, resulting in $r \cdot 2^{r-1}$ shard replicas. Group 0 contains one replica from all shards. Other groups can be constructed following a recursive procedure:

1 Let Ψ be the set of all shards. If Ψ contains only one shard, stop. Otherwise, use the median split key κ in Ψ to divide all shards into two sets A and B. Keys of all shards in A are larger than κ, while keys of all shards in B are smaller than κ. Perform step 2, and then perform step 3.

2 Create a new group to contain one replica from all shards in set A. Then, let $\Psi \leftarrow A$, and recursively apply step 1.

3 Let $\Psi \leftarrow B$, and then recursively apply step 1.

Replicas in the same group are stored in the same physical server, whereas different groups of the same region are placed into different physical servers. According to the construction procedure, group 1 starts from the median split key, covering the bottom half of the key space (*i.e.*, 2^{r-2} shards). Group 1 allows half of the regions workload to be moved from group 0's server to group 1's server without sacrificing data locality. Figure 8 demonstrates an example of $r = 3$. PyroDFS is compatible with normal HDFS workload whose replicas can be simply set as no group specified. Section 3.3.2 explains why group 1 and 2 are placed at the end rather than in the beginning of the StoreFile.

Figure 8 also shows how Pyro makes use of DFS block replicas. The shaded area indicates which replica serves workloads falling in that key range. In the beginning, there is only one region server. Replicas in group 0 take care of all workloads. As all replicas in group 0 are stored locally in the region's physical server, data locality is preserved. After one split, the daughter region with smaller keys stays in the same physical server, hence still enjoys data locality. Another daughter region moves to the physical server that hosts replica group 1, which is also able to serve this daughter region using local data. Subsequent splits are carried out under the same fashion.

To distinguish from the original split operation in HBase, we call the above actions the *soft* split operation. Soft splits are designed to mitigate moving hotspots. Daughter regions created by soft splits eventually merge back to form their parent regions. The efficiency of the merge operation is not a concern as it can be performed after the hotspot moves out of that region. Please notice that the original split operation, which we call the *hard* split, is still needed when a region grows too large to fit in one physical server. As this paper focuses on geometry query and moving hotspots, all splits in the following sections refer to soft splits.

3.3.2 Fault Tolerance

As a persistent data store, Pyro needs to preserve high data availability. The block grouping algorithm presented in the previous section affects DFS replica placement schemes, which in turn affects Pyro's fault tolerance properties. In this section, we show that the block grouping algorithm allows Pyro to achieve higher data availability compared to the default random replica

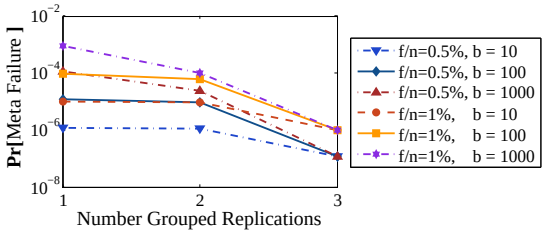


Figure 9: Unavailability Probability

placement policy in HDFS.

Pyro inherits the same HFile format [3] from HBase to store key-value pairs. According to HFile Format, meta blocks are stored at the end of the file. Losing any DFS block of the meta may leave the entire HFile unavailable, whereas the availability of key-value DFS blocks are not affected by the availability of other key-value DFS blocks. This property makes the last shard of the file more important than all preceding shards. Therefore, we choose two different objectives for their fault tolerance design.

- Meta shard: Minimize the probability of losing any DFS block.
- Key-value shard: Minimize the expectation of the number of unavailable DFS blocks.

Assume there are n servers in the cluster, and f nodes are unavailable during a cluster failure event. For a given shard, assume it contains b blocks, and replicates r times, where g out of r replications are grouped. PyroDFS randomly distributes the grouped g replications into g physical servers. The remaining $(r-g)b$ block replicas are randomly and exclusively distributed in the cluster. If the meta fails, it must be the case that the g servers hosting the g grouped replications all fail (*i.e.*, $\binom{f}{g}/\binom{n}{g}$), and at least one block in each $r-g$ ungrouped replications fails. Hence, the probability of meta failure is

$$\mathbf{Pr}\left[\text{meta failure}\right] = \frac{\binom{f}{g}}{\binom{n}{g}}\left(1-\left(1-\frac{\binom{f-g}{r-g}}{\binom{n-g}{r-g}}\right)^{b}\right). \quad (1)$$

Figure 9 plots how the number of grouped replications g affects the failure probability. In this experiment, n and r are set to 10,000, and 3 respectively. According to [20, 21, 22], after some power outage, 0.5%-1% of the nodes fail to reboot. Hence, we vary f to be 50, and 100. The results show that the meta failure probability decreases when g increases. Pyro sets g to the maximum value for the meta shard, therefore achieves higher fault tolerance compared to default HDFS where g equals 1.

For key-value shards, transient and small-scale failures are tolerable, as they do not affect most queries. It is more important to minimize the scale of the failure (*i.e.*, the number of unavailable DB blocks). The expected failure scale is,

$$\mathbf{E}\left[\text{failure scale}|\text{failure occurs}\right] = \frac{b\binom{f-g}{r-g}}{\binom{n-g}{r-g}}. \quad (2)$$

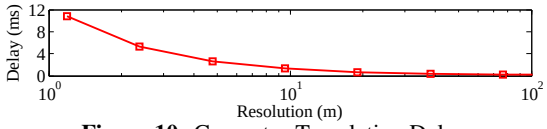


Figure 10: Geometry Translation Delay

The failure scale decreases with the increase of grouped replication number g. Therefore, placing replica groups 1 and 3 at the end of the StoreFile minimizes both the meta shard failure probability and the failure scale of key-value shards.

4 Implementation

PyroDFS and PyroDB are implemented based on HDFS-2.4.1 and HBase-0.99 respectively.

4.1 Moore Encoding

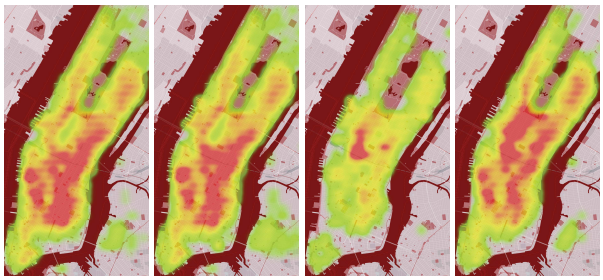
As previously shown in Figure 4 and Figure 5, each unit of Moore curve can be uniquely defined by the combination of its orientation (north, east, south, and west) and its rotation (clockwise, counter-clockwise). Encode the orientation with 2 bits, $d1$ and $d0$, such that 00 denotes north, 01 east, 10 south, and 11 west. With more careful observations, it can be seen that the rotation of a Moore curve component unit completely depends on its orientation. Starting from the direction shown in Figure 4 (a), the encodings in east and west oriented units rotate clockwise, and others rotate counter-clockwise. With a given integer coordinate (x,y), let x_k and y_k denote the k^{th} lowest bits of x and y in the binary presentation. Let $d_{k,1}d_{k,0}$ be the orientation of the component unit defined by the highest $r-k-1$ bits in x, and y. Then, the orientation $d_{k-1,1}d_{k-1,0}$ can be determined based on $d_{k,1}$, $d_{k,0}$, x_k, and y_k [15, 16, 17, 18].

$$\begin{aligned}
d_{k-1,0} &= \bar{d}_{k,1}\bar{d}_{k,0}\bar{y}_k \mid \bar{d}_{k,1}d_{k,0}x_k \\
&\mid d_{k,1}\bar{d}_{k,0}y_k \mid d_{k,1}d_{k,0}\bar{x}_k && (3)\\
&= \bar{d}_{k,0}\left(d_{k,1}\oplus\bar{y}_k\right) \mid d_{k,0}\left(d_{k,1}\oplus x\right) && (4)\\
d_{k-1,1} &= \bar{d}_{k,1}\bar{d}_{k,0}x_k\bar{y}_k \mid \bar{d}_{k,1}d_{k,0}\bar{x}_k y_k \\
&\mid d_{k,1}\bar{d}_{k,0}\overline{\bar{x}_k y_k} \mid d_{k,1}d_{k,0}\overline{x_k\bar{y}_k} && (5)\\
&= d_{k,1}\left(\bar{x}_k\oplus y_k\right) \mid \left(x_k\oplus y_k\right)\left(d_0\oplus x_k\right) && (6)
\end{aligned}$$

The formula considers all situations where $d_{k-1,0}$ and $d_{k-1,1}$ should equal to 1, and uses a logic *or* to connect them all. For example, the term $\bar{d}_{k,1}\bar{d}_{k,0}\bar{y}_k$ states that when the previous orientation is north ($\bar{d}_{k,1}\bar{d}_{k,0}$), the current unit faces east or west ($d_{k-1,0}=1$) if and only if $y_k=0$. The same technique can be applied to determine the final Moore encoding index ω.

$$\begin{aligned}
\omega_{2k+1} &= \bar{d}_{k,1}\bar{d}_{k,0}\bar{x}_k \mid \bar{d}_{k,1}d_{k,0}\bar{y}_k \\
&\mid d_{k,1}\bar{d}_{k,0}x_k \mid d_{k,1}d_{k,0}y_k && (7)\\
&= \bar{d}_{k,0}\left(d_{k,1}\oplus\bar{x}_k\right) + d_{k,0}\left(d_{k,1}\oplus\bar{y}_k\right) && (8)\\
\omega_{2k} &= \bar{x}_k\oplus y_k && (9)
\end{aligned}$$

Then, each geometry can be translated into range scans using a quad tree. Each level in the quad tree corresponds to a resolution level. Each node in the tree represents



(a) (b) (c) (d)

Figure 11: Manhattan Taxi Pick-up/Drop-off Hotspots

a tile, which is further divided into four smaller tiles in the next level. The translating algorithm only traverses deeper if the geometry query partially overlaps with that area. If an area is fully covered by the geometry, there is no need to go further downwards. Figure 10 shows the delay of translating a $5km \times 5km$ square geometry. The delay stays below 11ms even using the finest resolution.

4.2 Multi-Scan Optimization

After converting a geometry query into range scans, the multi-scan optimizer needs two more pieces of information to minimize the response time: 1) storage media performance profiles, and 2) the mapping from key ranges to DB blocks. For the former one, an administrator may specify an HDFS path under the property name *hbase.profile.storage* in the hbase-site.xml configuration file. This path should point to a file containing multiple lines of (p-read size, p-read delay) items, indicating the storage media performance profile result. Depending on storage media types in physical servers, the administrator may set the property *hbase.profile.storage* to different values for different HRegions. The file will be loaded during HRegion initialization phase. For the latter one, HBase internally keeps indices of DB blocks. Therefore, Pyro can easily translate a range scan into a serious of block starting offsets and block sizes. Then, those information will be provided as inputs for the A^3 algorithm.

4.3 Block Grouping

Distributed file systems usually keep replica placement policies as an internal logic, maintaining a clean separation between the DFS layer and higher layer applications. This design, however, prevents Pyro from exploring opportunities to make use of DFS data replications. Pyro carefully breaks this barrier by exposing a minimum amount of control knobs to higher layer applications. Through these APIs, applications may provide *replica group* information when writing data into DFS. It is important to choose the right set of APIs such that PyroDFS applications do not need to reveal too much about details in the DFS layer. At the same time, applications are able to fully make use of data locality benefits of all block replicas.

In our design, PyroDFS exposes two families of APIs which help to alter its internal behavior.

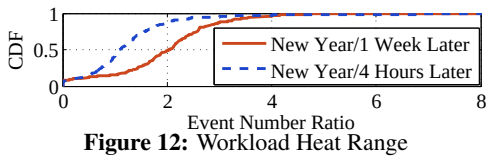


Figure 12: Workload Heat Range

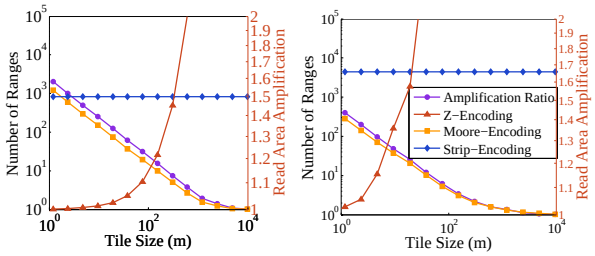
Figure 13: Row Key

- *Sealing a DFS Block:* PyroDB may force PyroDFS to seal the current DFS block and start writing into a new DFS block, even if the current DFS block has not reached its size limit yet. This API is useful because DFS block boundaries may not respect splitting keys, especially when there are many StoreFiles in a region and the sizes of StoreFiles are about the same order of magnitude of the DFS block size. The *seal* API family will help StoreFiles to achieve full data locality after splits.
- *Grouping Replicas:* PyroDB may specify *replica namespace* and *replica groups* when calling the *write* API in PyroDFS. This usually happens during MemStore flushes and StoreFile compactions. Under the same namespace, replicas in the same replica group will be placed into the same physical server, and replicas in different groups will be placed into different physical servers. If there are not enough physical servers or disk spaces, PyroDFS works in a best effort manner. The mapping from the replica group to the physical server and corresponding failure recovery is handled within PyroDFS. PyroDB may retrieve a physical server information of a given replica group using *grouping* APIs, which allows PyroDB to make use of data locality benefits.

5 Evaluation

Evaluations use NYC taxi dataset [9, 10] that contains GPS pickup/dropoff location information of 697,622,444 trips from 2010 to 2013. The experiments run on a cluster of 80 Dell servers (40 Dell PowerEdge R620 servers and 40 Dell PowerEdge R610 servers) [23, 24, 25, 26, 27, 28, 29, 30, 31, 32]. The HDFS cluster consists of 1 master node and 30 datanodes. The HBase server contains 1 master node, 3 zookeeper [33] nodes, and 30 region servers. Region servers are co-located with data nodes. Remaining nodes follow a central controller to generate geometry queries and log response times, which we call Remote User Emulators (RUE).

We first briefly analyze the NYC taxi dataset. Then, Sections 5.2, 5.3, and 5.4 evaluate the performance improvements contributed by Geometry Translator, Multi-Scan Optimizer, and Group-based Replica Placement respectively. Finally, in Section 5.5, we measure the overall response time and throughput of Pyro.



(a) radius = $1000m$ (b) $50m \times 628m$

Figure 14: Reducing the Number of Range Scans

5.1 NYC Taxi Data Set Analysis

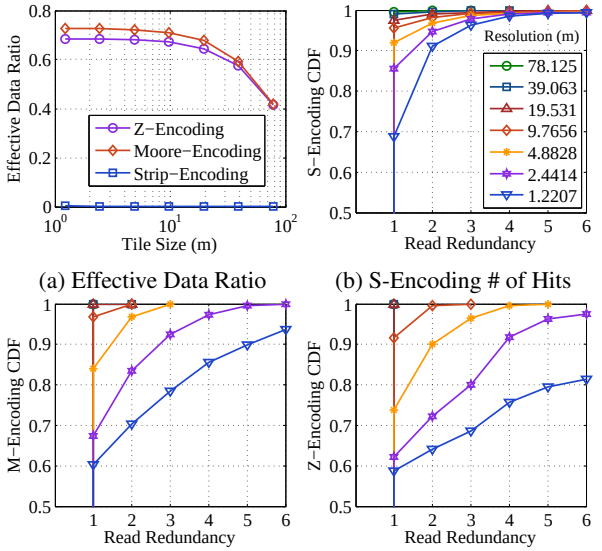
Moving hotspot is an important phenomenon in spatial-temporal data. Figure 11 (a) and (b) illustrate the heat maps of taxi pick-up and drop-off events in the Manhattan area during a 4 hour time slot starting from 8:00PM on December 31, 2010 and December 31, 2012 respectively. The comparison shows that the trip distribution during the same festival does not change much over the years. Figure 11 (c) plots the heat map of the morning (6:00AM-10:00AM) on January 1st, 2013, which drastically differs from the heat map shown in Figure 11 (b). Figure 11 (d) illustrates the trip distribution from 8:00PM to 12:00AM on July 4th, 2013, which also considerably differs from that of the New Year Eve in the same year.

Figures 11 (a)-(d) demonstrate the distribution of spatial-temporal hotspots. It is important to understand by how much hotspots cause event count to increase in a region. We measure the increase as the ratio, $\frac{\text{event count during peak hours}}{\text{event count during normal hours}}$. The CDF on 16X16 Manhattan area is shown in Figure 12. Although hotspots move over time, the event count of a region changes within a reasonably small range. During New Year midnight, popularity of more than 97% regions grow within four folds.

When loading the data into HBase, both spatial and temporal information contribute to the row key. The encoding algorithm translates the 2D location information of an event into a 32-bit spatial-key, which acts as the suffix of the row key. Then, the temporal strings are parsed to Linux 64-bit timestamps. We use the most significant 32 bits as the temporal-key. Each temporal key represents roughly a 50-day time range. Finally, as shown in Figure 13, the temporal-key is concatenated in front of the spatial key to form the complete row key.

5.2 Moore Encoding

Figure 14 shows how much Moore encoding helps to reduce the number of range scans at different resolutions when translating geometry queries in a $40,000,000m \times 40,000,000m$ area. Figures 14 (a) and (b) uses disk geometry and rectangle geometries respectively. The two figures share the same legend. For disk geometries, Moore encoding generates 45% fewer range scans when compared to ZOrder-encoding. When a long rectangle is in use, Moore encoding helps to reduce the number of range scans by 30%.



(a) Effective Data Ratio (b) S-Encoding # of Hits

(c) Z-Encoding # of Hits (d) M-Encoding # of Hits

Figure 15: Read Amplification Phenomenon

To quantify the read volume amplification, we encode the dataset coordinates with Moore encoding algorithm using the highest resolution shown in Figure 3, and populate the data using 64KB DB Blocks. Then, the experiment issues $1Km \times 1Km$ rectangle geometries. Figure 15 (a) shows the ratio of fetched key-value pairs volume over the total volume of accessed DB Blocks, which is the inverse of read volume amplification. As the Strip-encoding results in very high read volume amplification, using the inverse helps to limit the result in interval [0, 1]. Therefore, readers can easily distinguish the difference between Moore-encoding and ZOrder-encoding. We call the inverse metric the effective data ratio. As Moore encoding concentrates a geometry query into fewer range scans, and hence fewer range boundaries, it also achieves higher effective data ratio.

Figures 15 (b)-(d) plot the CDFs of redundant read counts when processing the same geometry query.It is clear that the number of redundant reads increases when using higher resolutions. Another observation is that, Moore-encoding leads to large read redundancy. Thanks to the multi-scan optimization design, this will not be a problem, as all redundant reads will be accomplished within a single DB block traverse operation.

5.3 Multi-Scan Optimization

In order to measure how A^3 algorithm works, we load data from the NYC taxi cab dataset using Moore encoding algorithm, and force all StoreFiles of the same store to be compacted into one single StoreFile. Then, the RUE generates $1Km \times 1Km$ rectangle geometry queries with the query resolution set to 13. We measure the internal delay of loading requested DB blocks individually versus aggregately.

The evaluation results are presented in Figure 16. The curves convey a few interesting observations. Let us look

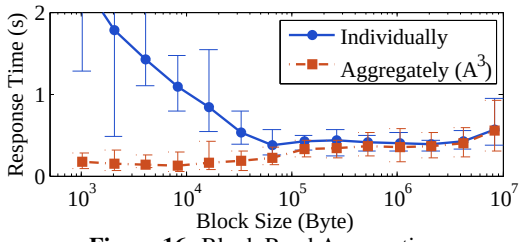


Figure 16: Block Read Aggregation

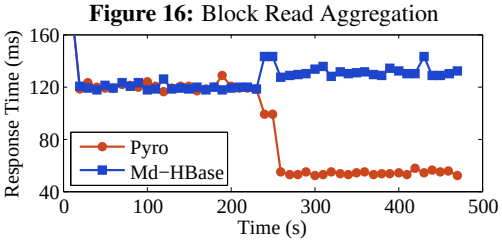


Figure 17: Response Time at Splitting Event

at the A^3 curve first. In general, this curve rises as the block size increases, which agrees with our intuition as larger blocks lead to more severe *read volume amplification*. The minimum response time is achieved at 8KB. Because the minimum data unit of the disk under test is 4KB, further decreasing block size does not help any more. On the computation side, using smaller block size results in larger input scale for the A^3 algorithm. That explains why the response time below 8KB slightly goes up as the block size decreases. The "*individually*" curve monotonically decreases when the block size grows from 1KB to 100KB. It is because increasing block size significantly reduces the number of disk seeks when the block is small. When the block size reaches between 128KB and 4MB, two facts become true: 1) key-value pairs hit by a geometry query tend to concentrate in less blocks; 2) data transmission time starts to make impacts. The benefits of reducing the number of disk seeks and the penalties of loading DB blocks start to cancel each other, leading to a flat curve. After 4MB, the data transmission delay dominates the response time, and the curve rises again. Comparing the nadirs of the two curves concludes that A^3 helps to reduce the response time by at least 3X.

5.4 Soft Region Split

To measure the performance of *soft* splitting, this experiment uses normal scan queries instead of geometry queries, excluding the benefits of Moore encoding and multi-scan optimization. A table is created for the NYC's taxi data, which initially splits into 4 regions. Each region is assigned to a dedicated server. The HBASE_HEAPSIZE parameter is set to 1GB, and the MemStore flush size is set to 256MB. Automatic region split is disabled to allow us to manually control the timing of splits. Twelve RUE servers generate random-sized small scan queries.

Figure 17 shows the result. The split occurs at the 240^{th} second. After the split operation, HBase suffers from even longer response time. It is because daughter region B does not have its region data in its own physical server, and has to fetch data from remote servers, including the one hosting daughter region A. When the group based replication is enabled, both daughter regions read data from local disks, reducing half of the pressure on disk, cpu, and network resources.

5.5 Response Time and Throughput

We measure the overall response time and throughput improved by Pyro compared to the state-of-the-art solution MD-HBase. Experiments submit rectangle geometry queries of size $1km \times 1km$ and $100m \times 100m$ to Pyro and MD-HBase. The request resolutions are set to 13 and 15 respectively for two types of rectangles. The block sizes vary from 8KB to 512KB. When using MD-HBase, the remote query emulator initiates all scan queries sequentially using one thread. This configuration tries to make the experiment fair, as Pyro uses a single thread to answer each geometry query. Besides, experiments also show how Pyro performs when using ZOrder-encoding or/and A^3 algorithm. Figures 18 and 19 plot experiment results. The legend on the upper-left corner shows the mapping from colors to block sizes. PyroM and PyroZ represent Pyro using Moore- and ZOrder- encoding respectively. PyroM-A^3 and PyroZ-A^3 correspond to the cases with the A^3 algorithm disabled.

When using PyroM and PyroZ, the response times grow with the increase of block size regardless of whether the rectangle geometry is large or small. It is because larger blocks weaken the benefits of block aggregation and force PyroM and PyroZ to read more data from disk. After disabling A^3, the response time rises by 6X for $1km \times 1km$ rectangles, and 2X for $100m \times 100m$ rectangles. MD-HBase achieves the shortest response time when using 64KB DB blocks, which is 60X larger compared to PyroM and PyroZ when handling $1km \times 1km$ rectangle geometries. Reducing the rectangle size to $100m \times 100m$ shrinks the gap to 5X. An interesting phenomenon is that using 512KB DB blocks only increases the response time by 5% compared to using 64KB DB blocks, when the request resolution is set to 13. However, the gap jumps to 33% if the resolution is set to 15. The reason is that, higher resolution leads to more and smaller range scans. In this case, multiple range scans are more likely to hit the same DB block multiple times. According to HFile format, key-value pairs are chained together as a linked-list in each DB block. HBase has to traverse the chain from the very beginning to locate the starting key-value pair for every range scan. Therefore, larger DB block size results in more overhead on iterating through the key-value chain in each DB block.

Figure 20 shows the throughput evaluation results of the entire cluster. Pyro regions are initially partitioned based on the average pick up/drop off event location distribution over the year of 2013. Literature [9] presents

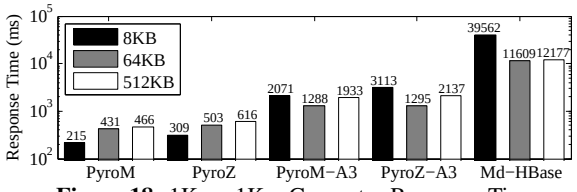


Figure 18: 1Km×1Km Geometry Response Time

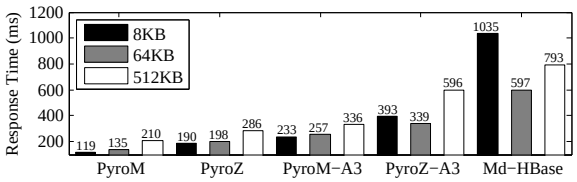


Figure 19: 100m×100m Geometry Response Time

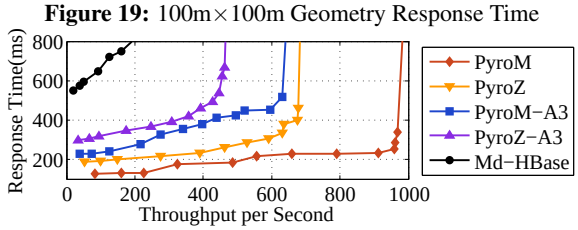


Figure 20: 100m×100m Geometry Throughput

more analysis and visualizations of the dataset. During the evaluation, each RUE server maintains a pool of emulated users who submit randomly located $100m \times 100m$ rectangle geometry queries. The reason of using small geometries in this experiment is that MD-HBase results in excessively long delays when handling even a single large geometry. The distribution of the rectangle geometry queries follows the heat map from 8:00PM to 11:59PM on December 31, 2013. The configuration mimics the situation where an application only knows the long-term data distribution, and is unable to predict hotspot bursts. When setting 600ms to be the maximum tolerable response time, Pyro outperforms MD-HBase by 10X.

6 Related Work

As the volume of spatial-temporal data is growing at an unprecedented rate, pursing a scalable solution for storing spatial-temporal data has become a common goal shared by researchers from both the distributed system community and the database community. Advances on this path will benefit a great amount of spatial-temporal applications and analytic systems.

Traditional relational databases understand high dimensional data well [17, 18, 34, 35] due to extensively studied indexing techniques, such as *R*-Tree [36], *Kd*-Tree [37], *UB*-Tree [38, 39], etc. Therefore, researchers seek approaches to improve the scalability. Wang *et al.* [40] construct a global index and local indices using Content Addressable Network [41]. The space is partitioned into smaller subspaces. Each subspace is handled by a local storage. The global index manages subspaces, and local indices manage data points in their own subspaces. Zhang *et al.* [42] propose a similar architecture using *R*-tree as global index and *Kd*-tree as local indices.

From another direction, distributed system researchers push scalable NoSQL stores [3, 4, 5, 6, 13, 43, 44, 45] to better understand high dimensional data. Distributed key-value stores can be categorized into two classes. One class uses random partition to organize keys. Such systems include cassandra [4], DynamoDB [5], etc. Due to the randomness on key distribution, these systems are immune to dynamic hotspots concentrated in a small key range. However, spatial-temporal data applications and analytic systems usually issue geometry queries, which translate to range scans. Random partitioning cannot handle range scans efficiently, as it cannot extract all keys within a range with only the range boundaries. Consequently, each range scan needs to query all servers. Other systems, such as BigTable [13], HBase [3], couchDB [46], use ordered partitioning algorithms. In this case, the primary key space is partitioned into regions. The benefits are clear. As data associated with a continuous primary key range are also stored consecutively, sorted partitioning helps to efficiently locate the servers that host the requested key range.

The benefits of ordered partitioning encouraged researchers to mount spatial-temporal application onto HBase. Md-HBase [7] builds an index layer on top of HBase. The index layer encodes spatial information of a data point into a bit series using ZOrder-encoding. Then, a row using that bit series as key is inserted into HBase. The ST-HBase [8] develops a similar technique. However when serving geometry queries, the index layer inevitably translates each geometry query into multiple range scans, and prevents data store from aggregately minimizing the response time.

As summarized above, existing solutions either organize multiple relational databases together using some global index, or build a separate index layer above some general purpose distributed data stores. This paper, however, takes a different path by designing and implementing a holistic solution that is specifically tailored for spatial-temporal data.

7 Conclusion

In this paper, we present the motivation, design, implementation, and evaluation of Pyro. Pyro tailors HDFS and HBase for high resolution spatial-temporal geometry queries. In the DB layer, Pyro employs Moore encoding and multi-scan optimization to efficiently handle geometry queries. In the DFS layer, group-based replica placement policy helps Pyro to preserve data locality benefits during hotspots dynamics. The evaluation shows that Pyro reduces the response time by 60X on $1km \times 1km$ rectangle geometries and improves the throughput by 10X on $100m \times 100m$ rectangle geometries compared to the state-of-the-art solution.

References

[1] B. Sanou, "The world in 2013: Ict facts and figures," in *International Communication Union, United Nations*, 2013.

[2] S. Steiniger and E. Bocher, "An overview on current free and open source desktop gis developments," *International Journal of Geographical Information Science*, vol. 23, no. 10, pp. 1345–1370.

[3] L. George, *HBase: The Definitive Guide*. O'Reilly, 2011.

[4] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *ACM SOSP*, 2007.

[6] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "Tao: Facebook's distributed data store for the social graph," in *USENIX ATC*, 2013.

[7] S. Nishimura, S. Das, D. Agrawal, and A. E. Abbadi, "Md-hbase: A scalable multi-dimensional data infrastructure for location aware services," in *IEEE International Conference on Mobile Data Management*, 2011.

[8] Y. Ma, Y. Zhang, and X. Meng, "St-hbase: A scalable data management system for massive geo-tagged objects," in *International Conference on Web-Age Information Management*, 2013.

[9] B. Donovan and D. B. Work, "Using coarse gps data to quantify city-scale transportation system resilience to extreme events," *Transportation Research Board 94th Annual Meeting*, 2014.

[10] New York City Taxi & Limousine Commission (NYCT&L), "Nyc taxi dataset 2010-2013," https://publish.illinois.edu/dbwork/open-data/, 2015.

[11] T. White, *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2009.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *ACM SOSP*, 2003.

[13] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *USENIX OSDI*, 2006.

[14] T. Harter, D. Borthakur, S. Dong, A. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis of hdfs under hbase: A facebook messages case study," in *USENIX FAST*, 2014.

[15] M. Bader, *Space-Filling Curves: An Introduction with Applications in Scientific Computing*. Springer Publishing Company, Incorporated, 2012.

[16] J. Lawder, "The application of space-flling curves to the storage and retrieval of multi-dimensional data," in *Ph.D. Thesis*, 2000.

[17] K.-L. Chung, Y.-L. Huang, and Y.-W. Liu, "Efficient algorithms for coding hilbert curve of arbitrary-sized image and application to window query," *Inf. Sci.*, vol. 177, no. 10, pp. 2130–2151.

[18] P. Venetis, H. Gonzalez, C. S. Jensen, and A. Halevy, "Hyper-local, directions-based ranking of places," *Proc. VLDB Endow.*, vol. 4, no. 5, pp. 290–301.

[19] R. Dickau, "Hilbert and moore 3d fractal curves," http://demonstrations.wolfram.com/HilbertAndMoore3DFractalCurves/, 2015.

[20] A. Cidon, S. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum, "Copysets: Reducing the frequency of data loss in cloud storage," in *USENIX ATC*, 2013.

[21] R. J. Chansler, "Data availability and durability with the hadoop distributed file system," in *The USENIX Magzine*, 2012.

[22] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in globally distributed storage systems," in *USENIX OSDI*, 2010.

[23] S. Li, S. Wang, F. Yang, S. Hu, F. Saremi, and T. F. Abdelzaher, "Proteus: Power proportional memory cache cluster in data centers," in *IEEE ICDCS*, 2013.

[24] S. Li, S. Wang, T. Abdelzaher, M. Kihl, and A. Robertsson, "Temperature aware power allocation: An optimization framework and case studies," *Sustainable Computing: Informatics and Systems*, vol. 2, no. 3, pp. 117 – 127, 2012.

[25] S. Li, S. Hu, and T. F. Abdelzaher, "The packing server for real-time scheduling of mapreduce workflows," in *IEEE RTAS*, 2015.

[26] S. Li, T. F. Abdelzaher, and M. Yuan, "TAPA: temperature aware power allocation in data center with map-reduce," in *IEEE International Green Computing Conference and Workshops*, 2011.

[27] S. Li, H. Le, N. Pham, J. Heo, and T. Abdelzaher, "Joint optimization of computing and cooling energy: Analytic model and a machine room case study," in *IEEE ICDCS*, 2012.

[28] S. Li, S. Hu, S. Wang, L. Su, T. F. Abdelzaher, I. Gupta, and R. Pace, "WOHA: deadline-aware map-reduce workflow scheduling framework over hadoop clusters," in *IEEE ICDCS*, 2014.

[29] M. M. H. Khan, J. Heo, S. Li, and T. F. Abdelzaher, "Understanding vicious cycles in server clusters," in *IEEE ICDCS*, 2011.

[30] S. Li, S. Hu, S. Wang, S. Gu, C. Pan, and T. F. Abdelzaher, "Wattvalet: Heterogenous energy storage management in data centers for improved power capping," in *USENIX ICAC*, 2014.

[31] S. Li, L. Su, Y. Suleimenov, H. Liu, T. F. Abdelzaher, and G. Chen, "Centaur: Dynamic message dissemination over online social networks," in *IEEE ICCCN*, 2014.

[32] CyPhy Research Group, "UIUC Green Data Center," http://greendatacenters.web.engr.illinois.edu/index.html, 2015.

[33] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *USENIX ATC*, 2010.

[34] J. K. Lawder and P. J. H. King, "Querying multi-dimensional data indexed using the hilbert space-filling curve," *SIGMOD Rec.*, vol. 30, no. 1, pp. 19–24.

[35] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-probe lsh: Efficient indexing for high-dimensional similarity search," in *VLDB*, 2007.

[36] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *ACM SIGMOD*, 1984.

[37] I. Wald and V. Havran, "On building fast kd-trees for ray tracing, and on doing that in o(n log n)," in *IEEE Symposium on Interactive Ray Tracing*, 2006.

[38] R. Bayer, "The universal b-tree for multidimensional indexing: General concepts," in *Proceedings of the International Conference on Worldwide Computing and Its Applications*, 1997.

[39] F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, and R. Bayer, "Integrating the ub-tree into a database system kernel," in *VLDB*, 2000.

[40] J. Wang, S. Wu, H. Gao, J. Li, and B. C. Ooi, "Indexing multi-dimensional data in a cloud system," in *ACM SIGMOD*, 2010.

[41] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *ACM SIGCOMM*, 2001.

[42] X. Zhang, J. Ai, Z. Wang, J. Lu, and X. Meng, "An efficient multi-dimensional index for cloud data management," in *International Workshop on Cloud Data Management*, 2009.

[43] B. Cho and M. K. Aguilera, "Surviving congestion in geo-distributed storage systems," in *USENIX ATC*, 2012.

[44] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "Mica: A holistic approach to fast in-memory key-value storage," in *USENIX NSDI*, 2014.

[45] R. Geambasu, A. A. Levy, T. Kohno, A. Krishnamurthy, and H. M. Levy, "Comet: An active distributed key-value store," in *USENIX OSDI*, 2010.

[46] J. C. Anderson, J. Lehnardt, and N. Slater, *CouchDB: The Definitive Guide Time to Relax*, 1st ed. O'Reilly Media, Inc., 2010.