An I/O efficient approach for concurrent spatio-temporal range queries over large-scale remote sensing image data

Ze Deng, Yue Wang, Tao Liu, Schahram Dustdar, *Fellow, IEEE,* Rajiv Ranjan, Albert Zomaya, *Fellow, IEEE,* Yizhi Liu and Lizhe Wang†, *Fellow, IEEE,*

Abstract—High-performance remote sensing analytics workflows require ingesting and querying massive image archives to support real-time spatio-temporal applications. While modern systems utilize window-based I/O reading to reduce data transfer, they face a dual bottleneck: the prohibitive overhead of runtime geospatial computations caused by the decoupling of logical indexing from physical storage, and severe storage-level I/O contention triggered by uncoordinated concurrent reads. To address these limitations, we present a comprehensive I/O-aware query processing approach based on a novel "Index-as-an-Execution-Plan" paradigm. We introduce a dual-layer inverted structure that serves as a deterministic I/O planner, pre-materializing grid-to-pixel mappings to completely eliminate runtime geometric calculations. Furthermore, we design a hybrid concurrency-aware I/O coordination protocol that adaptively integrates Calvin-style deterministic ordering with optimistic execution, effectively converting I/O contention into request merging opportunities. To handle fluctuating workloads, we incorporate a Surrogate-Assisted Genetic Multi-Armed Bandit mechanism for automatic parameter tuning. Evaluated on a distributed cluster with Sentinel-2 datasets, our approach reduces end-to-end latency by an order of magnitude compared to standard window-based reading, achieves linear throughput scaling under high concurrency, and demonstrates superior convergence speed in automatic tuning.

Index Terms—Remote sensing data management, Spatio-temporal range queries, I/O-aware indexing, Concurrency control, I/O tuning

1 Introduction

A Massive amount of remote sensing (RS) data, characterized by high spatial, temporal, and spectral resolutions, is being generated at an unprecedented speed due to the rapid advancement of Earth observation missions [1]. For instance, NASAs AVIRIS-NG acquires nearly 9 GB of data per hour, while the EO-1 Hyperion sensor generates over 1.6 TB daily [2]. Beyond the sheer volume of data, these datasets are increasingly subjected to intensive concurrent access from global research communities and real-time emergency response systems (e.g., multi-departmental coordination during natural disasters). Consequently, modern RS platforms are required to provide not only massive storage capacity but also high-throughput query capabilities to satisfy the simultaneous demands of numerous spatio-temporal analysis tasks.

Existing RS data management systems [3], [4], [5] typically decompose a spatio-temporal range query into a decoupled two-phase execution model. The first phase is the metadata filtering phase, which utilizes spatio-temporal metadata (e.g., footprints, timestamps) to identify candidate image files that intersect the query predicate. Recent advancements have transitioned from traditional tree-based indexes [6], [7] to scalable distributed schemes based on grid encodings and space-filling curves, such as GeoHash [8], GeoSOT [4], and GeoMesa [9]. By leveraging these high-dimensional indexing structures, the search complexity of the first phase has been effectively reduced to $O(\log N)$ or even $O(1)$, making metadata discovery extremely efficient even for billion-scale datasets.

- *Z. Deng, L. Wang (Corresponding author, lizhe.Wang@gmail.com), Y. Wang, T. Liu, and Y. Liu are with the School of Computer Science, China University of Geosciences, Wuhan, 430078, P.R.China.*
- *Z. Deng, and L. Wang (Corresponding author, lizhe.Wang@gmail.com) are also with Hubei Key Laboratory of Intelligent Geo-Information Processing, China University of Geosciences, Wuhan 430074, China.*
- *Schahram Dustdar is with the Technische Universität Wien, Austria.*
- *R. Ranjan is with School of Computing, Newcastle University, U.K.*
- *A. Zomaya is with the School of Information Technologies, The University of Sydney, Sydney, Australia.*

The second phase is the data extraction phase, where the system reads the actual pixel data from the identified raw image files stored in distributed file systems or object stores. A critical observation in modern high-performance RS analytics is that the primary system bottleneck has fundamentally shifted from the first phase to the second. While the metadata search completes in milliseconds, the end-to-end query latency is now dominated by the massive I/O overhead required to fetch, decompress, and process large-scale raw images. Traditional systems attempted to reduce I/O overhead by pre-slicing tiles and building pyramids (e.g., approaches used in Google Earth Engine [10] that store metadata in HBase and serve pre-tiled image pyramids), but aggressive tiling increases management complexity and produces many small files. More recent Cloud-Optimized GeoTIFF (COG) formats and COG-aware frameworks [3], [11] exploit internal overviews and window-based I/O to read only the portions of files that spatially intersect a query.

While window-based I/O effectively reduces raw data transfer, it introduces a new "computation wall" due to the decoupling of logical indexing from physical storage. Current state-of-the-art systems operate on a "Search-then-Compute-then-Read" model: after identifying candidate files, they must perform fine-grained, per-image geospatial computations at runtime to map query coordinates to precise file offsets and clip boundaries. This runtime geometric resolution (C_{geo}) becomes computationally prohibitive when processing a large volume of candidate images, often negating the benefits of I/O reduction. Moreover, under concurrent workloads, the lack of coordination among these independent read requests leads to severe I/O contention and storage thrashing, rendering traditional indexing-centric optimizations insufficient for real-time applications.

To address the problems above, we propose a novel "Index-as-an-Execution-Plan" paradigm to strictly bound the query latency. Unlike conventional approaches that treat indexing and I/O execution as separate stages, our approach integrates fine-grained partial querying directly into the indexing structure. By pre-materializing the mapping between logical spatial grids and physical pixel windows, our system enables deterministic I/O planning without run-

time geometric computation. To further ensure scalability, we introduce a concurrency control protocol tailored for spatio-temporal range queries and an automatic I/O tuning mechanism. The principal contributions of this paper are summarized as follows:

1) We propose an I/O-aware "Index-as-an-Execution-Plan" schema. Instead of merely returning candidate image identifiers, our index directly translates high-level spatio-temporal predicates into concrete, byte-level windowed read plans. This design bridges the semantic gap between logical queries and physical storage, eliminating expensive runtime geospatial computations and ensuring that I/O cost is proportional strictly to the query footprint.
2) We propose a hybrid concurrency-aware I/O coordination protocol. This protocol adapts transaction processing principles by integrating Calvin-style deterministic ordering [12] with optimistic execution [13]. It shifts the focus from protecting database rows to coordinating shared I/O flows. This protocol dynamically switches strategies based on spatial contention, effectively converting "I/O contention" into "request merging opportunities."
3) We proposed an automatic I/O tuning method to improve the I/O performance of spatio-temporal range queries over remote sensing data. The method extends an existing AI-powered I/O tuning framework [14] based on a surrogate-assisted genetic multi-armed bandits algorithm [15].

The remainder of this paper is organized as follows: Section 2 presents the related work. Section 3 proposes the definition concerning the spatio-temporal range query problem. Section 4 proposes the indexing structre. Section 5 proposes the hybrid concurrency control protocol. Section 6 proposes the method of I/O stack tuning. Section 7 presents the experiments and results. Section 8 concludes this paper with a summary.

2 Related Work

This section describes the most salient studies of I/O-efficient spatio-temporal query processing, concurrency control and I/O Performance Tuning.

2.1 I/O-Efficient Spatio-Temporal Query Processing

Efficient spatio-temporal query processing for remote sensing data has been extensively studied, with early efforts primarily focusing on metadata organization and index-level pruning in relational database systems. Traditional approaches typically extend tree-based spatial indexes, such as R-tree [6], quadtree [16], and their spatio-temporal variants [7], to organize image footprints together with temporal attributes, and are commonly implemented on relational backends (e.g., MySQL and PostgreSQL). These methods provide efficient range filtering for moderate-scale datasets, but their reliance on balanced tree structures often leads to high maintenance overhead and limited scalability as the volume of remote sensing metadata grows rapidly. With the continuous increase in data volume and ingestion rate, recent systems have gradually shifted toward grid-based spatio-temporal indexing schemes deployed on distributed NoSQL stores. By encoding spatial footprints into uniform spatial grids using GeoHash [8], GeoSOT [4], or space-filling curves [9], [5], and combining them with temporal identifiers, these approaches enable lightweight index construction and better horizontal scalability on backends such as HBase and Elasticsearch. Such grid-based indexes can effectively reduce the candidate search space through coarse-grained pruning and are more suitable for large-scale, continuously growing remote sensing archives.

However, index pruning alone is insufficient to guarantee end-to-end query efficiency for remote sensing workloads, where individual images are usually large and query results require further pixel-level processing. To reduce the amount of raw I/O, Google Earth system [10] rely on tiling and multi-resolution pyramids that physically split images into small blocks. While more recent solutions leverage COG and window-based I/O to enable partial reads from monolithic image files. Frameworks such as OpenDataCube [3] exploit these features to read only the image regions intersecting a query window, thereby reducing unnecessary data transfer. Nevertheless, after candidate images are identified, most systems still perform fine-grained geospatial computations for each image, including coordinate transformations and precise pixel-window derivation, which may incur substantial overhead when many images are involved.

2.2 Concurrency Control

Concurrency control has long been studied to provide correctness and high throughput in multi-user database and storage systems, with two broad paradigms dominating the literature: deterministic scheduling [12] and non-deterministic schemes [17], [18]. Hybrid approaches [19], [20] that adaptively combine these paradigms seek to exploit the low-conflict efficiency of deterministic execution while retaining the flexibility of optimistic techniques. More recent proposals such as OOCC target read-heavy, disaggregated settings by reducing validation and round-trips for read-only transactions, achieving low latency under OLTP-like workloads [21]. These CC families are primarily optimized for record- or key-level access patterns: their metrics and designs emphasize transaction latency, abort rates, and throughput under workloads with small, well-defined read/write sets.

Overall, existing concurrency control mechanisms are largely designed around transaction-level correctness and throughput, assuming record- or key-based access patterns and treating storage I/O as a black box. Their optimization objectives rarely account for I/O amplification or fine-grained storage contention induced by concurrent range queries. Consequently, these approaches are ill-suited for data-intensive spatio-temporal workloads, where coordinating overlapping window reads and mitigating storage-level interference are critical to achieving scalable performance under multi-user access.

2.3 I/O Performance Tuning in Storage Systems

I/O performance tuning has been extensively studied in the context of HPC and data-intensive storage systems, where complex multi-layer I/O stacks expose a large number of tunable parameters. These parameters span different layers, including application-level I/O libraries, middleware, and underlying storage systems, and their interactions often lead to highly non-linear performance behaviors. As a result, manual tuning is time-consuming and error-prone, motivating a wide range of auto-tuning approaches.

Several studies focus on improving the efficiency of the tuning pipeline itself by reformulating the search space or optimization objectives. Chen et al. [22] proposed a meta multi-objectivization (MMO) model that introduces auxiliary performance objectives to mitigate premature convergence to local optima. While such techniques can improve optimization robustness, they are largely domain-agnostic and do not explicitly account for the characteristics of I/O-intensive workloads. Other works, such as the contextual bandit-based approach by Bez et al. [23], optimize specific layers of the I/O stack (e.g., I/O forwarding) by exploiting observed access patterns. However, these methods are primarily designed for administrator-level tuning and target

isolated components rather than end-to-end application I/O behavior.

User-level I/O tuning has also been explored, most notably by H5Tuner [24], which employs genetic algorithms to optimize the configuration of the HDF5 I/O library. Although effective for single-layer tuning, H5Tuner does not consider cross-layer interactions and lacks mechanisms for reducing tuning cost, such as configuration prioritization or early stopping.

More recently, TunIO [14] proposed an AI-powered I/O tuning framework that explicitly targets the growing configuration spaces of modern I/O stacks. TunIO integrates several advanced techniques, including I/O kernel extraction, smart selection of high-impact parameters, and reinforcement learningdriven early stopping, to balance tuning cost and performance gain across multiple layers. Despite its effectiveness, TunIO and related frameworks primarily focus on single-application or isolated workloads, assuming stable access patterns during tuning. Query-level I/O behaviors, such as fine-grained window access induced by spatio-temporal range queries, as well as interference among concurrent users, are generally outside the scope of existing I/O tuning approaches.

3 DEFINITION

This section formalizes the spatio-temporal range query problem and establishes the cost models for query execution. We assume a distributed storage environment where large-scale remote sensing images are stored as objects or files.

Definition 1 (Spatio-temporal Remote Sensing Image). A remote sensing image R is defined as a tuple:

$$R = \langle id, \Omega, \mathcal{D}, t \rangle, \tag{1}$$

where id is the unique identifier; $\Omega = [0, W] \times [0, H]$ denotes the pixel coordinate space; $\mathcal{D}$ represents the raw pixel data; and t is the temporal validity interval. The image is associated with a spatial footprint $MBR(R)$ in the global coordinate reference system.

Definition 2 (Spatio-temporal Range Query). Given a dataset $\mathbb{R}$, a query Q is defined by a spatio-temporal predicate $Q = \langle S, T \rangle$, where S is the spatial bounding box and T is the time interval. The query result set $\mathcal{R}_Q$ is defined as:

$$\mathcal{R}_Q = R \in \mathbb{R} \mid MBR\,(R) \cap S \neq \emptyset \wedge R.t \cap T \neq \emptyset. \tag{2}$$

For each $R \in \mathcal{R}_Q$, the system must return the pixel matrix corresponding to the intersection region $MBR(R) \cap S$.

Definition 3 (Query Execution Cost Model). The execution latency of a query Q, denoted as $Cost(Q)$, is composed of two phases: metadata filtering and data extraction.

$$Cost\,(Q) = C_{meta}\,(Q) + \sum_{R \in \mathcal{R}_Q} (C_{geo}\,(R, Q) + C_{io}\,(R, Q)). \tag{3}$$

Here, $C_{meta}(Q)$ is the cost of identifying candidate images $\mathcal{R}_Q$ using indices. The data extraction cost for each image consists of two components: geospatial computation cost (C_{geo}) and I/O access cost (C_{io}). C_{geo} is the CPU time required to calculate the pixel-to-geographic mapping, determine the exact read windows (offsets and lengths), and handle boundary clipping. In window-based partial reading schemes, this cost is non-negligible due to the complexity of coordinate transformations. C_{io} is the latency to fetch the actual binary data from storage.

Definition 4 (Concurrent Spatio-temporal Queries). Let $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_N\}$ denote a set of spatio-temporal range queries issued concurrently by multiple users. Each query Q_i independently specifies a spatio-temporal window $\langle S_i, T_i \rangle$ and may overlap with others in both spatial and temporal dimensions. Concurrent execution of $\mathcal{Q}$ may induce overlapping partial reads over the same images or image regions, leading to redundant I/O and storage-level contention if queries are processed independently.

Problem Statement (Latency-Optimized Concurrent Query Processing). Given a dataset $\mathbb{R}$ and a concurrent workload $\mathcal{Q}$, the objective is to minimize the total execution latency:

$$\min \sum_{Q_i \in \mathcal{Q}} \left(C_{meta}\,(Q_i) + \sum_{R \in \mathcal{R}_{Q_i}} (C_{geo}\,(R, Q_i) + C_{io}\,(R, Q_i)) \right), \tag{4}$$

subject to:

1) *Correctness:* The returned data must strictly match the spatio-temporal predicate defined in Eq. (2).
2) *Isolation:* Concurrent reads must effectively share I/O bandwidth without causing starvation or excessive thrashing.

4 I/O-AWARE INDEXING STUCTURE

This section introduces the details of indexing structre for spatio-temporal range query over remote sensing image data.

4.1 Index schema design

To enable I/O-efficient spatio-temporal query processing, we first decompose the global spatial domain into a uniform grid that serves as the basic unit for query pruning and data access coordination. Specifically, we adopt a fixed-resolution global tiling scheme based on the Web Mercator (or EPSG:4326) coordinate system, using zoom level 14 to partition the Earths surface into fine-grained grid cells (experiments show that the 14-level grid has the highest indexing efficiency which can be referred to Section 7.2.3). This resolution strikes a practical balance between spatial selectivity and index size: finer levels would significantly increase metadata volume and maintenance cost, while coarser levels would reduce pruning effectiveness and lead to unnecessary image I/O. At this scale, each grid cell typically corresponds to a spatial extent comparable to common query footprints and to the internal tiling granularity used by modern raster formats, making it well suited for partial data access.

Grid-to-Image Mapping (G2I). Based on the grid decomposition, we construct a grid-centric inverted index to associate spatial units with covering images. In our system, each grid cell is assigned a unique *GridKey*, encoded as a 64-bit Z-order value to preserve spatial locality and enable efficient range scans in key-value stores such as HBase. The *G2I table* stores one row per grid cell, where the row key is the GridKey and the value maintains the list of image identifiers (ImageKeys) whose spatial footprints intersect the corresponding cell, as illustrated in Fig. 1(a).

This grid-to-image mapping allows query processing to begin with a lightweight enumeration of grid cells covered by a query region, followed by direct lookups of candidate images via exact GridKey matches. By treating each grid cell as an independent spatial bucket, the G2I table provides efficient metadata-level pruning and avoids costly geometric intersection tests over large image footprints.

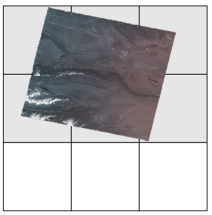
However, the G2I table alone is insufficient for I/O-efficient query execution. While it identifies which images are relevant to a given grid cell, it does not capture how the grid cell maps to pixel regions within each image. As a result, a grid-only representation cannot directly guide partial reads and would still require per-image geospatial computations at query time. Therefore, the G2I table functions as a coarse spatial filter and must be complemented by an image-centric structure that materializes the correspondence between grid cells and pixel windows, enabling fine-grained, window-based I/O.

Key	Value
Grid Key	Remote Sening Image List
$grid_1$	img_2, img_{30}
$grid_2$	img_1, img_2, img_5
...	...
$grid_n$	img_9, img_{60}, img_7

(a) Grid to Image Mapping

Key	Value		
Image Key	Grid–Window Mapping	Temporal Properties	Storage Path
img_1	$< grid_2, W_{1\_2} >, < grid_5, W_{1\_5} >$	t_1	p_1
img_2	$< grid_2, W_{2\_2} >$	t_2	p_2
...	...	...	...
img_n	$< grid_{30}, W_{n\_30} >, < grid_8, W_{n\_8} >$	t_n	p_n

(b) Image to Grid Mapping



(c) Image Signature

Fig. 1. Index schema design.

Image-to-Grid Mapping (I2G). To complement the grid-centric G2I table and enable fine-grained, I/O-efficient data access, we introduce an image-centric inverted structure, referred to as the Image-to-Grid mapping (I2G). In contrast to G2I, which organizes metadata by spatial grids, the I2G table stores all grid-level access information of a remote sensing image in a single row. Each image therefore occupies exactly one row in the table, significantly improving locality during query execution.

As illustrated in Fig. 1(b), the row key of the I2G table is the *ImageKey*, i.e., the unique identifier of a remote sensing image. The row value is organized into three column families, each serving a distinct role in query-time pruning and I/O coordination:

GridWindow Mapping. This column family records the list of grid cells intersected by the image together with their corresponding pixel windows in the image coordinate space. Each entry has the form

$$\langle GridKey, W_{ImageKey\_GridKey} \rangle,$$

where *GridKey* identifies a grid cell at the chosen global resolution, and $W_{ImageKey\_GridKey}$ denotes the minimal pixel bounding rectangle within the image that exactly covers that grid cell.

These precomputed window offsets allow the query executor to directly issue windowed reads on large raster files without loading entire images into memory or recomputing geographic-to-pixel transformations at query time. As a result, grid cells become the smallest unit of coordinated I/O, enabling precise partial reads and effective elimination of redundant disk accesses.

Temporal Metadata. To support spatio-temporal range queries, each image row includes a lightweight temporal column family that stores its acquisition time information, such as the sensing timestamp or time interval. This metadata enables efficient temporal filtering to be performed jointly with spatial grid matching, without consulting external catalogs or secondary indexes.

Storage Pointer. This column family contains the information required to retrieve image data from the underlying storage system. It stores a stable file identifier, such as an object key in an object store (e.g., MinIO/S3) or an absolute path in a POSIX-compatible file system. By decoupling logical image identifiers from physical storage locations, this design supports flexible deployment across heterogeneous storage backends while allowing the query engine to directly access image files once relevant pixel windows have been identified.

The I2G table offers several advantages. First, all grid-level access information for the same image is colocated in a single row, avoiding repeated random lookups and improving cache locality during query execution. Second, by materializing grid-to-window correspondences at ingestion time, the system completely avoids expensive per-query geometric computations and directly translates spatial overlap into byte-range I/O requests. Third, the number of rows in the I2G table scales with the number of images rather than the number of grid cells, substantially reducing metadata volume and maintenance overhead.

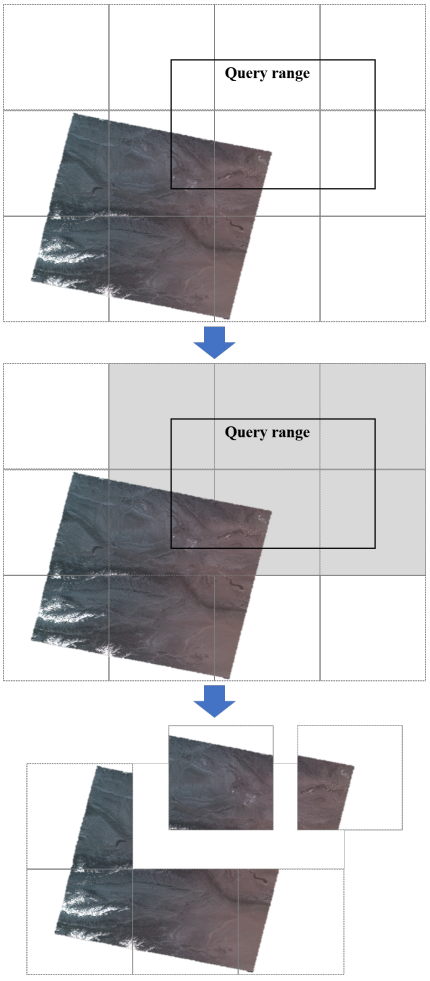


Fig. 2. Query-time Execution

During data ingestion, the gridwindow mappings are generated by projecting grid boundaries into the image coordinate system using the images georeferencing parameters. This process requires only lightweight affine or RPC transformations and does not involve storing explicit geometries or performing polygon clipping. As a result, the I2G structure enables efficient partial reads while keeping metadata compact and ingestion costs manageable.

4.2 Query-time Execution

The I/O-aware index enables efficient spatio-temporal range queries by directly translating query predicates into windowed read plans, while avoiding both full-image loading and expensive geometric computations.

Given a user-specified spatio-temporal query $q = \langle [x_{\min}, y_{\min}, x_{\max}, y_{\max}], [t_s, t_e] \rangle$, the system resolves the query through three consecutive stages: *Grid Enumeration*, *Candidate Image Retrieval with Temporal Pruning*, and *Windowed Read Plan Generation*. As illustrated in Fig. 2, this execution pipeline bridges high-level query predicates and low-level I/O operations in a fully deterministic manner.

Grid Enumeration. As shown in Step 1 and Step 2 of Fig. 2, the query execution starts by rasterizing the spatial footprint of q into the fixed global grid at zoom level 14. Instead of performing recursive space decomposition as in quadtrees or hierarchical spatial indexes, our system enumerates the minimal set of grid cells $\{g_1, \ldots, g_k\}$ whose footprints intersect the query bounding box.

Each grid cell corresponds to a unique 64-bit *GridKey*, which directly matches the primary key of the G2I table. This design has important implications: grid enumeration has constant depth and low computational cost and the resulting GridKeys can be directly used as lookup keys without any geometric refinement. Consequently, spatial key generation is reduced to simple arithmetic operations on integer grid coordinates.

Candidate Image Retrieval with Temporal Pruning. Given the enumerated grid set $\{g_1, \ldots, g_k\}$, the query processor performs a batched multi-get on the G2I table. Each G2I row corresponds to a single grid cell and stores the identifiers of all images whose spatial footprints intersect that cell. For each grid g_i, the lookup returns:

$$G2I[g_i] = \{imgKey_1, \ldots, imgKey_m\}.$$

All retrieved image identifiers are unioned to form the spatial candidate set $C_s = \bigcup_{i=1}^{k} G2I[g_i]$. This step eliminates the need for per-image polygon intersection tests that are commonly required in spatial databases and data cube systems.

To incorporate the temporal constraint $[t_s, t_e]$, each candidate image in C_s is further filtered using the temporal column family of the Image-to-Grid (I2G) table. Images whose acquisition time does not intersect the query interval are discarded early, yielding the final candidate set C. This lightweight temporal pruning is performed without accessing any image data and introduces negligible overhead.

Windowed Read Plan Generation. As shown in Step 3 of Fig. 2, the final stage translates the candidate image set into a concrete I/O plan. For each image $I \in C$, the query executor issues a selective range-get on the I2G table to retrieve only the gridwindow mappings relevant to the query grids:

$$I2G\left[I, \{g_1, ..., g_k\}\right] = \{W_{I\_g_i} \mid g_i \cap I \neq \emptyset\}. \tag{5}$$

Each $W_{I\_g_i}$ specifies the exact pixel window in the original raster file that corresponds to grid cell g_i. Since these window offsets are precomputed during ingestion, query execution requires only key-based lookups and arithmetic filtering. No geographic coordinate transformation, polygon clipping, or rastervector intersection is performed at query time.

The resulting collection of pixel windows constitutes a *windowed read plan*, which can be directly translated into byte-range I/O requests against the storage backend. This approach avoids loading entire scenes and ensures that the total I/O volume is proportional to the queried spatial extent rather than the image size.

4.3 Why I/O-aware

The key reason our indexing design is I/O-aware lies in the fact that the index lookup results are not merely candidate identifiers, but constitute a concrete I/O access plan. Unlike traditional spatial indexes, where query processing yields a set of objects that must still be fetched through opaque storage accesses, our Grid-to-Image and Image-to-Grid lookups deterministically produce the exact pixel windows to be read from disk. As a result, the logical query plan and the physical I/O plan are tightly coupled: resolving a spatio-temporal predicate directly specifies which byte ranges should be accessed and which can be skipped.

This tight coupling fundamentally changes the optimization objective. Instead of minimizing index traversal cost or result-set size, the system explicitly minimizes data movement by ensuring that disk I/O is proportional to the querys spatio-temporal footprint. Consequently, the index serves as an execution-aware abstraction that bridges query semantics and storage behavior, enabling predictable, bounded I/O under both single-query and concurrent workloads.

Theoretical Cost Analysis. To rigorously quantify the performance advantage, we revisit the query cost model defined in Eq. (3):

$$Cost(Q) = C_{meta}(Q) + \sum_{R \in \mathcal{R}_Q} \left(C_{geo}(R, Q) + C_{io}(R, Q)\right).$$

In traditional full-image reading systems, although the geospatial computation cost is negligible ($C_{geo} = 0$) as no clipping is performed, the I/O cost C_{io} is determined by the full file size. Consequently, the total latency is entirely dominated by massive I/O overhead, rendering C_{meta} (typically milliseconds) irrelevant.

Existing window-based I/O systems (e.g., ODC or COG-aware libraries) successfully reduce the I/O cost to the size of the requested window. However, this reduction comes at the expense of a significant surge in C_{geo}. For every candidate image, the system must perform on-the-fly coordinate transformations and polygon clipping to calculate read offsets. When a query involves thousands of images, the accumulated CPU time ($\sum C_{geo}$) becomes a new bottleneck (e.g., hundreds of milliseconds to seconds), often negating the benefits of I/O reduction (detailed quantitative comparisons are provided in Sec. 7.2.2).

In contrast, our I/O-aware indexing approach fundamentally alters this trade-off. By materializing the grid-to-pixel mapping in the I2G table, we effectively shift the computational burden from query time to ingestion time. Although the two-phase lookup (G2I and I2G) introduces a slight overhead compared to simple tree traversals, C_{meta} remains in the order of millisecondsorders of magnitude smaller than disk I/O latency. Since the precise pixel windows are pre-calculated and stored, the runtime geospatial computation is effectively eliminated, i.e., $C_{geo} = 0$. The system retains the minimal I/O cost characteristic of window-based approaches, fetching only relevant byte ranges. Therefore, our design achieves the theoretical minimum for both computation and I/O components within the query execution critical path.

5 Hybrid Concurrency-Aware I/O Coordination

In this section, we propose a hybrid coordination mechanism that adaptively employs either lock-free non-deterministic execution or deterministic coordinated scheduling based on the real-time contention level of spatio-temporal workloads.

5.1 Query Admission and I/O Plan Generation

When a spatio-temporal range query Q arrives, the system first performs index-driven plan generation. The query footprint is rasterized into the global grid to enumerate the intersecting grid cells. The G2I table is then consulted to retrieve the set of candidate images, followed by selective

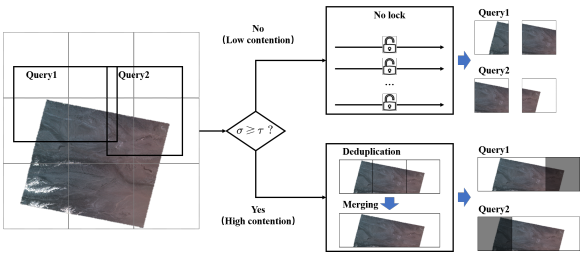


Fig. 3. Hybrid Concurrency-Aware I/O Coordination.

lookups in the I2G table to obtain the corresponding pixel windows.

As a result, each query is translated into an explicit *I/O access plan* consisting of imagewindow pairs:

$$Plan\,(Q) = \{(img_1, w_1)\,, (img_1, w_2)\,, (img_3, w_5)\,, ...\}\,, \quad (6)$$

where each window w denotes a concrete pixel range to be accessed via byte-range I/O. Upon admission, the system assigns each query a unique *QueryID* and records its arrival timestamp.

5.2 Contention Estimation and Path Selection

To minimize the overhead of global ordering in low-contention scenarios, the system introduces a Contention-Aware Switch. Upon the arrival of a query batch $\mathcal{Q} = \{Q_1, Q_2, ..., Q_n\}$, the system first estimates the Spatial Overlap Ratio (σ) among their generated I/O plans.

Let $A(Plan(Q_i))$ be the aggregate spatial area of all pixel windows in the I/O plan of query Q_i. The overlap ratio σ for a batch is defined as:

$$\sigma = 1 - \frac{\mathrm{A}(\bigcup_{i=1}^{n} Plan(Q_i))}{\sum_{i=1}^{n} \mathrm{A}(Plan(Q_i))}, \quad (7)$$

where $\sigma \in [0, 1]$. A high σ indicates that multiple queries are competing for the same image regions, leading to high I/O amplification if executed independently.

The system utilizes a rule-based assignment mechanism similar to HDCC [20] to select the execution path:

1) Path A (Non-deterministic/OCC-style): If $\sigma < \tau$ (where τ is a configurable threshold), queries proceed directly to execution to maximize concurrency.
2) Path B (Deterministic/Calvin-style): If $\sigma \geq \tau$, queries are routed to the Global I/O Plan Queue for coordinated merging.

5.3 Deterministic Coordinated and Non-deterministic Execution

When $\sigma \geq \tau$, the system switches to a deterministic path to mitigate storage-level contention and I/O amplification, as shown in Fig. 3. To coordinate concurrent access to shared storage resources, we introduce a *Global I/O Plan Queue* that enforces a deterministic ordering over all admitted I/O plans. Each windowed access (img, w) derived from incoming queries is inserted into this queue according to a predefined policy, such as FIFO based on arrival time or lexicographic ordering by $(timestamp, QueryID)$.

This design is inspired by deterministic scheduling in systems such as Calvin, but differs fundamentally in its scope: the ordering is imposed on *window-level I/O operations* rather than on transactions. As a result, accesses to the same image region across different queries follow a globally consistent order, preventing uncontrolled interleaving of reads and reducing contention at the storage layer. The deterministic ordering also provides a stable foundation for subsequent I/O coordination and sharing.

The core of our approach lies in coordinating concurrent windowed reads at the image level. Windows originating from different queries may overlap spatially, be adjacent, or even be identical. Executing these requests independently would lead to redundant reads and excessive I/O amplification.

To address this, the system performs three coordination steps within each scheduling interval. Stage 1: Global Deduplication. The system first extracts all windowed access pairs (img, w) from the admitted queries and inserts them into a global window set ($\mathcal{W}_{total}$). If multiple queries $Q_1, Q_2, ..., Q_n$ request the same pixel window w from image img, the system retains only one unique entry in $\mathcal{W}_{total}$. This stage ensures that any specific byte range is identified as a single logical requirement, effectively preventing the redundant retrieval of overlapping spatial grids. Stage 2: Range Merding. After de-duplication, the system analyzes the physical disk offsets of all unique windows in $\mathcal{W}_{total}$. Following the principle of improving access locality, windows that are physically contiguous or separated by a gap smaller than a threshold θ are merged into a single read. Stage 3: Dispatching. This stage maintains a mapping between the physical byte-offsets in the buffer and the logical window requirements of each active query. Each query Q_i receives only the exact pixel windows $w \in Plan(Q_i)$ it originally requested. This is achieved via zero-copy memory mapping where possible, or by slicing the shared system buffer into local thread-wise structures. This ensures that while the physical I/O is shared to reduce amplification, the logical execution of each query remains independent and free from irrelevant data interference.

For example, when Q_1 requests grids $\{1, 2\}$ and Q_2 requests grids $\{2, 3\}$, Stage 1 identifies the unique requirement set $\{1, 2, 3\}$. Stage 2 then merges these into a single contiguous I/O operation covering the entire range $[1, 3]$. In Stage 3, the dispatcher identifies memory offsets corresponding to grids 1 and 2 within the buffer and maps these slices to the private cache of Q_1. For Q_2, similarly, the dispatcher extracts and delivers slices for grids 2 and 3 to Q_2.

Through these mechanisms, concurrent queries collaboratively share I/O, and the execution unit becomes a coordinated window read rather than an isolated request. Importantly, this coordination operates entirely at the I/O planning level and does not require any form of locking or transaction-level synchronization.

When contention remains below the threshold ($\sigma < \tau$), the system prioritizes low latency over merging efficiency by adopting an optimistic dispatch mechanism, as shown in Fig. 3. Instead of undergoing heavy-weight sorting, I/O plans are immediately offloaded to the execution engine. By utilizing thread-local sublists, each thread independently handles its byte-range requests.

5.4 Optimistic Read Execution and Completion

Once a coordinated window read is scheduled, the system issues the corresponding byte-range I/O request immediately. Read execution is fully optimistic: there is no validation phase, no abort, and no rollback. This is enabled by the immutability of remote-sensing imagery and by the deterministic ordering of I/O plans, which together ensure consistent and repeatable read behavior.

A query is considered complete when all windows in its I/O plan have been served and the associated local processing (e.g., reprojection or mosaicking) has finished. By eliminating validation overhead and allowing read execution to proceed independently once scheduled, the system achieves low-latency query completion while maintaining predictable I/O behavior under concurrency.

Overall, this concurrency-aware I/O coordination mechanism reinterprets concurrency control as a problem of *coordinating shared I/O flows*. By operating at the granularity of windowed reads and leveraging deterministic ordering

and optimistic execution, it effectively reduces redundant I/O and improves scalability for multi-user spatio-temporal query workloads.

6 I/O Stack Tuning

We first describe an I/O stack tuning problem and then the surrogate-assisted GMAB algorithm is proposed to solve the problem.

6.1 Formulation of Online I/O Tuning

We study a concurrency spatio-temporal query engine that processes many range queries at the same time. The system works on large remote sensing images stored in shared storage. Different from traditional HPC jobs or single-application I/O workloads, the system does not run one fixed job. Instead, it keeps receiving a stream of user queries. Each query is turned into many small I/O operations that often touch overlapping regions in large raster files.

Let $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_N\}$ denote a stream of spatio-temporal range queries submitted by multiple users. Each query q is decomposed by the I/O-aware index into a set of grid-aligned spatial windows based on a predefined global grid system. These windows are further mapped to sub-regions of one or more large remote sensing images. In this way, every query produces an I/O execution context $c = \langle W, M, S \rangle$, where W describes the set of image windows to be accessed, including their sizes, spatial overlap, and distribution across images. M captures window-level coordination opportunities, such as window merging, deduplication, or shared reads across concurrent queries. S represents system-level execution decisions, including batching strategies, I/O scheduling order, and concurrency limits. Importantly, the I/O behavior of the system is not determined solely by static application code, but emerges dynamically from the interaction between query workloads, execution plans, and system policies.

The goal of I/O tuning in this system is to optimize the performance of query-induced I/O execution under continuous, concurrent workloads. We focus on minimizing the observed I/O cost per query, which may be measured by metrics such as average query latency, effective I/O throughput, or amortized disk read time. Let $\theta \in \Theta$ denote a tuning configuration, where each configuration specifies a combination of system-level I/O control parameters, including window batching size, merge thresholds, queue depth, concurrency limits, and selected storage-level parameters exposed to the engine. Unlike traditional I/O tuning frameworks, the decision variables θ are applied at the query execution level, rather than at application startup or compilation time.

For a given tuning configuration θ and execution context c, the observed I/O performance is inherently stochastic due to: interference among concurrent queries; shared storage contention; variability in window overlap and access locality. We model the observed performance outcome as a random variable:

$$Y(\theta, c) = f(\theta, c) + \epsilon, \tag{8}$$

where $f(\cdot)$ is an unknown performance function and ϵ captures stochastic noise. Moreover, as query workloads evolve over time, the distribution of execution contexts c may change, making the tuning problem non-stationary.

Given a stream of queries $\mathcal{Q}$ and the resulting sequence of execution contexts $\{c_t\}$, the problem is to design an online tuning strategy that adaptively selects tuning configurations θ_t for query execution, so as to minimize the long-term expected I/O cost:

$$\min_{\{\theta_t\}} \mathbb{E}\left[\sum_{t=1}^{T} Y(\theta_t, c_t)\right], \tag{9}$$

subject to practical constraints on tuning overhead and system stability.

6.2 Surrogate-Assisted GMAB for Online I/O Tuning

Algorithm 1: Surrogate-Assisted Genetic Multi-Armed Bandit (SA-GMAB)

Input : Configuration space Θ, Initial population size P, Exploration parameter α, Surrogate update interval Δ
Output: Online selection of I/O coordination configuration θ_t

```
// Initialization
Initialize memory table M = ∅;
Initialize surrogate model f̃ with empty training data;
Generate an initial population P_0 ⊂ Θ;
Set tuning step counter t ← 0;
// Online Tuning Loop
while arrival of query q_t with execution context c_t do
    // Candidate Generation
    Apply genetic operators (selection, crossover, mutation) on current population to generate candidate set C_t ⊂ Θ;
    // Surrogate-based Pre-evaluation
    foreach θ ∈ C_t do
        r̂_θ ← f̃(θ, c_t);
    end
    // Candidate Filtering
    Select top-K configurations C'_t ⊂ C_t based on r̂_θ or uncertainty;
    // Bandit-based Selection
    foreach θ ∈ C'_t do
        Score(θ) = μ̂_θ + α sqrt(log(t+1) / (n_θ + 1));
    end
    Select configuration: θ_t = arg max_{θ∈C'_t} Score(θ);
    // Query Execution & Reward Observation
    Execute query q_t using I/O coordination policy θ_t;
    Measure performance outcome and compute reward r_t;
    // State Update
    Update memory entry for θ_t: n_{θ_t} ← n_{θ_t} + 1;
    μ̂_{θ_t} ← μ̂_{θ_t} + (r_t − μ̂_{θ_t}) / n_{θ_t};
    Update population P by inserting θ_t (optionally evicting low-performing ones);
    if t mod Δ = 0 then
        Retrain surrogate model f̃ using observations in M;
    end
    t ← t + 1;
end
```

To address the online I/O tuning problem, we use a Surrogate-Assisted Genetic Multi-Armed Bandit (SA-GMAB) framework. It combines genetic search, bandit-style exploration, and a simple performance model. The goal is to handle workloads where behavior changes over time, where results are random, and where queries may affect each other. The main steps of this framework are shown in Algorithm 1.

We first initialize the memory table and the surrogate model, and then generate an initial population of configurations (lines 1-4). In our system, each arm is an I/O tuning configuration $\theta \in \Theta$. A configuration is a group of I/O control parameters, such as merge thresholds, batch

size, queue depth, and limits on parallel requests. The space of possible configurations is large and discrete. It is not possible to list or test all of them. So we do not fix all arms in advance. Instead, new configurations are created dynamically by genetic operators during candidate generation (line 6). Each configuration acts as a policy that tells the system how to run I/O plans during a scheduling period.

When a query q_t with context c_t arrives, the framework enters the online tuning loop (line 5). For this query, a set of candidate configurations is created through selection, crossover, and mutation (line 6). For every candidate configuration, the surrogate model predicts its reward under the current context (lines 7-9). These predicted rewards are then used to filter and keep only the top promising configurations, or those with high uncertainty (line 10).

When a configuration θ is used to process a query q_t with context c_t, the system observes a random performance result $Y_t = Y(\theta, c_t)$. We define the reward as a simple transformation of I/O cost so that a higher reward means better performance. A common form is the negative latency of the query, or the negative I/O time per unit work. Because other queries run at the same time, the reward may change even for the same configuration. Thus, many samples are needed to estimate the expected reward.

For the remaining candidates, the framework computes a bandit score using both historical average reward and exploration term (lines 11–13), and then selects the configuration with the highest score (line 14). In this way, the method prefers configurations that have performed well before, but it also tries configurations that have been used only a few times.

The selected configuration is then applied to execute the query (line 15). After execution, the system observes the performance result and converts it into a reward value (line 16). For each configuration θ, the system keeps a memory entry that records how many times it has been used and its average reward. These values are updated after each execution (lines 17-18). This keeps all historical observations instead of discarding older ones, so estimates become more accurate over time, and poor configurations are not repeatedly tried.

The selected configuration may also be added into the population, while poor ones may be removed (line 19). The surrogate model is retrained periodically using data stored in memory (lines 20-22), so that its predictions follow the most recent workload. The tuning step counter is then increased (line 23), and the framework continues with the next query (line 24).

7 Performance Evaluation

First, we introduce the experimental setup, covering the dataset characteristics, query workload generation, and the distributed cluster environment. Then, we present the experimental results evaluating the proposed I/O-aware indexing structure, the hybrid concurrency-aware I/O coordination mechanism, and the online I/O tuning framework, respectively.

7.1 Experimental Setup

7.1.1 Dataset

We employed a large-scale real-world remote sensing dataset derived from the Sentinel-2 mission [1], specifically the Level-2A atmospherically corrected products. The dataset comprises multi-spectral images covering global land surfaces from 2019 to 2023. To simulate a cloud-native storage environment, all images are converted into Cloud-Optimized GeoTIFF (COG) format and stored in a distributed object store. The statistics of the dataset are summarized in Table 1.

1. https://sentinel.esa.int/web/sentinel/missions/sentinel-2

TABLE 1
Dataset Statistics

Dataset	Resolution	Time Span	Total Volume
Sentinel-2	10m - 60m	2019–2023	15.4 TB
Landsat-8	30m	2020–2022	4.2 TB

TABLE 2
Cluster Configurations

Hardware Configuration (Per Node)	
CPU	Dual Intel Xeon Gold 6248 (20 cores, 2.50GHz)
Memory	128GB DDR4 ECC
Storage	4TB NVMe SSD (Data) + 500GB SSD (OS)
Network	10 Gigabit Ethernet (10GbE)
Software Stack	
OS	Ubuntu 20.04 LTS
Storage	Hadoop 3.3.1, HBase 2.4.5, Lustre
Framework	OpenJDK 11, Spark 3.2.1

7.1.2 Query Workload

To evaluate the system performance under diverse scenarios, we developed a synthetic workload generator that simulates concurrent spatio-temporal range queries. The query parameters are configured as follows:

- **Spatial Extent:** The spatial range of queries follows a log-uniform distribution, ranging from small tile-level access (0.001% of the scene) to large-scale regional mosaics (1% to 100% of the scene).
- **Temporal Range:** Each query specifies a time interval randomly chosen between 1 day and 1 month.
- **Concurrency & Contention:** The number of concurrent clients N varies from 1 to 64. To test the coordination mechanism, we control the Spatial Overlap Ratio $\sigma \in [0, 0.9]$ to simulate workloads ranging from disjoint access to highly concentrated hotspots.

7.1.3 Experimental Environment

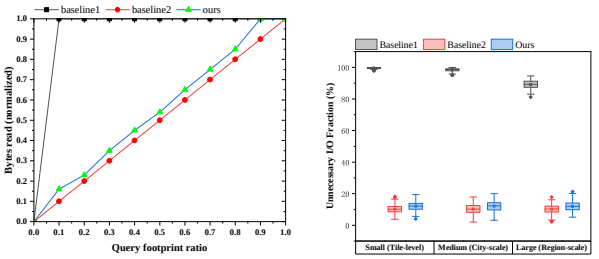
All experiments are conducted on a cluster with 9 homogenous nodes (1 master node and 8 worker nodes). The cluster is connected via a 10Gbps high-speed Ethernet to ensure that network bandwidth is not the primary bottleneck compared to storage I/O. Table 2 lists the detailed hardware and software configurations. The I/O-aware index (G2I/I2G) is deployed on HBase, while the raw image data is served by a MinIO distributed object storage cluster.

7.2 Evaluating the data indexing structure

In the following experiments, we measured the indexing on a single node in the cluster, bacause each nodes needs to the indexing for spatial query. We investigated of query performance of the indexing for remote sensing images.

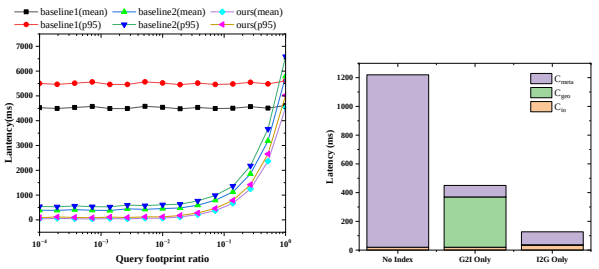
7.2.1 I/O Selectivity Analysis

First, we evaluated the effectiveness of data reduction by measuring the I/O selectivity, defined as the ratio of the retrieved data volume to the total file size. Fig. 4 compares our method against Baseline 1 (full-file retrieval) and Baseline 2 (exact window-based reading, e.g., OpenDataCube). As illustrated in Fig. 4(a), Baseline 1 exhibits a linear increase in I/O volume proportional to the file size, resulting in poor selectivity regardless of the query footprint. In contrast, both Baseline 2 and Ours significantly reduce I/O traffic by



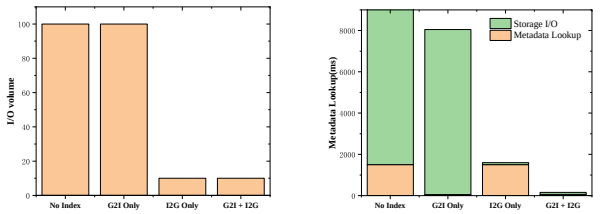
(a) I/O Selectivity (b) [Unnecessary data fraction

Fig. 4. The computing cost of spatial-keyword skylines



(a) The query latency (b) Latency Breakdownn

Fig. 5. End-to-End Query Latency



(a) I/O Reduction Analysis (b) Latency Breakdown

Fig. 6. Ablation Analysis

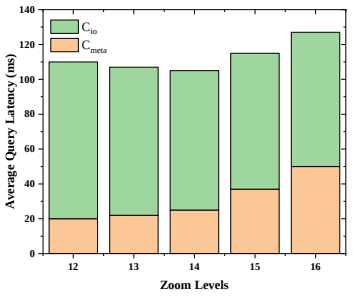


Fig. 7. Impact of grid resolution on query latency

enabling partial reads. It is worth noting that our method incurs slightly higher I/O volume (approximately $16\% - 23\%$ of the file size for small queries) compared to the theoretically optimal Baseline 2 ($10\% - 20\%$). This marginal data redundancy is attributed to the grid alignment effect: our index retrieves pixel blocks based on fixed grid boundaries, whereas Baseline 2 performs precise geospatial clipping. Fig. 4(b) further presents the distribution of unnecessary data fraction. While our method introduces a small amount of "over-reading" due to grid padding, it successfully avoids the massive data waste observed in Baseline 1. As we will demonstrate in the next section, this slight compromise in I/O precision is a strategic trade-off that eliminates expensive runtime computations.

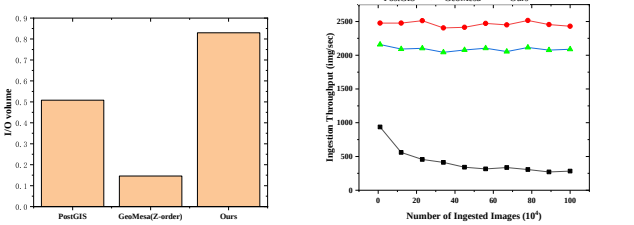
7.2.2 End-to-End Query Latency

We next measured the end-to-end query latency to verify whether the I/O reduction translates into time efficiency. Fig. 5(a) reports the mean and 95th percentile (P95) latency across varying query footprint ratios (log scale).The results reveal three distinct performance behaviors:Baseline 1 shows a high and flat latency curve (≈ 4500 ms), dominated by the cost of transferring entire images.Baseline 2, despite its optimal I/O selectivity, exhibits a significant latency floor ($\approx$ 380 ms for small queries). This overhead stems from the on-the-fly geospatial computations required to calculate precise read windows.Ours achieves the lowest latency, ranging from 34 ms to 59 ms for typical tile-level queries (10^{-4} coverage).Crucially, for small-to-medium queries, our method outperforms Baseline 2 by an order of magnitude. The gap between the two curves highlights the advantage of our deterministic indexing approach: by pre-materializing grid-to-window mappings, we eliminate runtime coordinate transformations. Although our I/O volume is slightly larger (as shown in Sec. 7.2.1), the time saved by avoiding computational overhead far outweighs the cost of transferring a few extra kilobytes of padding data.

To empirically validate the cost model proposed in Eq. 3, we further decomposed the query latency into three components: metadata lookup (C_{meta}), geospatial computation (C_{geo}), and I/O access (C_{io}). Fig. 5(b) presents the time consumption breakdown for a representative medium-scale query (involving approx. 50 image tiles). As expected, the latency of Baseline 1 is entirely dominated by C_{io} ($> 99\%$), rendering C_{meta} and C_{geo} negligible. The massive data transfer masks all other overheads. While C_{io} of Baseline 2 is successfully reduced to the window size, a new bottleneck emerges in C_{geo}. The runtime coordinate transformations and polygon clipping consume nearly 70% of the total execution time (approx. 350 ms). This observation confirms our theoretical analysis that window-based I/O shifts the bottleneck from storage to CPU. The proposed method exhibits a balanced profile. Although C_{meta} increases slightly (approx. 60 ms) due to the two-phase index lookup (G2I + I2G), this cost is well-amortized. Crucially, C_{geo} is effectively eliminated ($<$ 1 ms) thanks to the pre-computed grid-window mappings. Consequently, our approach achieves a total latency of approx. 150 ms, providing a $3\times$ speedup over Baseline 2 by removing the computational bottleneck without regressing on I/O performance.

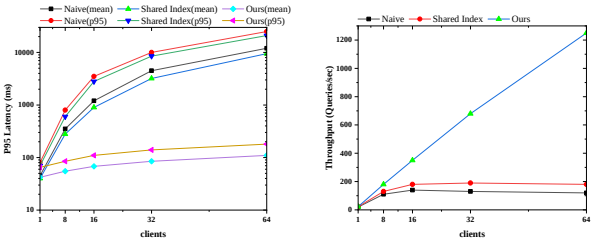
7.2.3 Ablation Study

To quantify the individual contributions of the G2I (coarse filtering) and I2G (fine-grained access) components, we decomposed the system into four variants. Fig. 6 breaks down the performance in terms of I/O volume and latency components (Metadata Lookup vs. Storage I/O). Fig. 6(a) confirms that removing either component leads to sub-optimal I/O behavior. The "No Index" and "G2I Only" variants result in 100% I/O volume (full-file reads), as they lack the window information required for partial access. Conversely, "I2G Only" and "Full" (Ours) achieve minimal I/O volume ($\approx 10\%$).However, I/O volume alone does not tell the full story. Fig. 6(b) reveals the latency breakdown:No Index: Suffers from both high metadata scanning cost (full table scan) and high storage I/O cost.G2I Only: Efficiently reduces metadata lookup time ($\approx$ 50 ms) but fails to reduce storage I/O ($\approx$ 8000 ms).I2G Only: Although it minimizes storage I/O ($\approx$ 100 ms), it incurs prohibitive metadata lookup overhead ($\approx$ 1500 ms) because the system must scan the entire I2G table to identify relevant images without spatial pruning.G2I + I2G (Ours): Achieves the "best of both worlds," maintaining low metadata latency ($\approx$ 60 ms) via



(a) Ingestion Scalability (b) Storage Consumption Overhead

Fig. 8. Index Construction and Storage Overhead



(a) The query latency (b) Aggregate Throughput

Fig. 9. The computing cost of spatial-keyword skylines

G2I pruning while ensuring minimal storage I/O ($\approx$ 100 ms) via I2G windowing.

Moreover, the choice of grid resolution (Zoom Level) is a critical parameter that dictates the trade-off between metadata management overhead (C_{meta}) and I/O precision (C_{io}). To justify our selection of Zoom Level 14, we conducted a sensitivity analysis by varying the grid resolution from Level 12 to Level 16 under a fixed workload of medium-scale range queries. Fig. 7 illustrates the latency breakdown across different resolutions. The results reveal a clear convex trajectory in total query latency, driven by two opposing forces. For coarse-grained grids (Level ≤ 13), while metadata lookup is extremely fast ($C_{meta} < 30$ ms) due to the small number of grid keys, the I/O cost (C_{io}) is prohibitively high. Large grid cells force the system to read significant amounts of irrelevant pixel data outside the actual query boundary (high read amplification), serving as the dominant bottleneck. Conversely, finer grids (Level 15, 16) maximize I/O precision, reducing C_{io} to its theoretical minimum. However, this comes at the cost of an explosion in metadata volume. A single query may intersect thousands of Level 16 micro-grids, causing C_{meta} to surge drastically (> 280 ms) due to the overhead of scanning and processing massive key lists in the G2I/I2G tables. As evidenced by the trough in the total latency curve, Zoom Level 14 represents the optimal "sweet spot" for our dataset. At this resolution, the grid cell size (approx. 20×20 meters at the equator) roughly matches the typical internal tile size of remote sensing images, keeping I/O waste low while maintaining a manageable number of index keys. Consequently, our system adopts Level 14 as the default global configuration.

7.2.4 Index Construction and Storage Overhead

Finally, we evaluated the scalability and cost of maintaining the index. Fig. 8 compares our method against PostGIS (R-tree) and GeoMesa (Z-order) during the ingestion of 10^6 images.Fig. 8(a) illustrates the ingestion throughput. PostGIS exhibits a degrading trend as the dataset grows, bottlenecked by the logarithmic cost of R-tree rebalancing. In contrast, Ours maintains a stable throughput ($\approx$ 2100 img/sec). Although slightly lower than the lightweight GeoMesa ($\approx$ 2500 img/sec) due to the dual-table write overhead, our method demonstrates linear scalability suitable for high-velocity streaming data.Regarding storage cost (Fig. 8(b)), our index occupies approximately 0.83% of the raw data size. While this is higher than GeoMesa (0.15%) and PostGIS (0.51%) due to the storage of grid-window mappings, it remains strictly below the 1% threshold. This result validates that the proposed method achieves significant performance gains with a negligible storage penalty.

7.3 Evaluating the Concurrency Control

In this section, we evaluate the proposed hybrid coordination mechanism on a distributed storage cluster to assess its scalability, robustness under contention, and internal storage efficiency. We investigated end-to-end latency, throughput, tail latency, and I/O amplification under varying degrees of concurrency and spatial contention.

To systematically control the workload characteristics, we developed a synthetic workload generator. We define the *Spatial Overlap Ratio* (σ) to quantify the extent of shared data regions among concurrent queries, ranging from $\sigma = 0$ (disjoint) to $\sigma = 0.9$ (highly concentrated hotspots). The number of concurrent clients varies from $N = 1$ to $N = 64$. For comparison, we evaluate the following execution schemes:

1) **Baseline A (Naive):** Queries function as isolated threads with independent I/O execution.
2) **Baseline B (Shared Index):** Metadata access is shared, but data retrieval remains uncoordinated, representing the state-of-the-practice in systems like GeoMesa.
3) **Ours:** The proposed mechanism featuring contention-aware switching, global I/O plan ordering, and window merging.

7.3.1 Concurrency Scalability

First, we investigated the system scalability by increasing the number of concurrent clients from 1 to 64 under a high-overlap scenario ($\sigma \approx 0.8$). Fig. 9 reports the mean latency, P95 tail latency, and aggregate throughput. Note that the latency axes in Fig. 9(a) are plotted on a log scale to visualize the orders-of-magnitude difference.

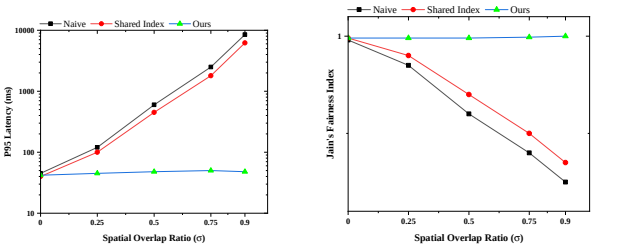
As shown in Fig. 9(a), both Baseline A and Baseline B exhibit exponential latency degradation. At 64 clients, the mean latency of Baseline A spikes to 12,000 ms, indicating severe storage saturation. This bottleneck arises from the "I/O blender effect," where randomized concurrent reads trigger severe disk seek thrashing. In contrast, Ours maintains a stable latency profile, increasing only marginally to 110 ms at 64 clients.

Fig. 9(b) further demonstrates the throughput advantage. While Baselines saturate at approximately 16–32 clients, Ours demonstrates super-linear throughput scaling relative to logical requests. This is attributed to the request collapse mechanism, where higher concurrency increases the probability of window merging, thereby reducing the physical I/O cost per query.

7.3.2 Tail Latency and Contention Sensitivity

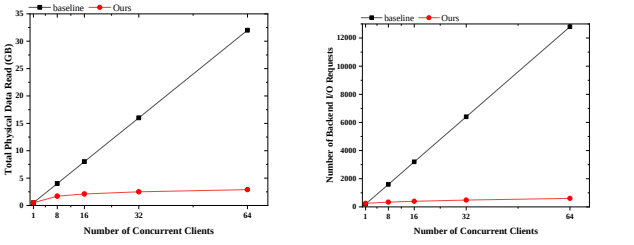
Next, we fixed the concurrency at $N = 32$ and swept the Spatial Overlap Ratio σ from 0 to 0.9 to evaluate the system's resilience to hotspots. Fig. 10 depicts the P95 latency and fairness index.

Intuitively, higher contention typically degrades performance. However, Fig. 10(a) reveals a *counter-intuitive* phenomenon for our system: the P95 latency remains flat (≈ 48 ms) even as σ approaches 0.9. This indicates that our coordination mechanism successfully converts contention" into optimization opportunities" via window merging. Conversely, both Baselines exhibit a sharp "performance cliff" when $\sigma > 0.5$, with Baseline A reaching 8,500 ms at $\sigma = 0.9$.



(a) Tail Latency Sensitivity (b) Fairness under Contention

Fig. 10. Tail Latency and Contention Sensitivity



(a) Data Volume Reduction (b) Request Collapse (IOPS)

Fig. 11. Storage-Level Effects and Request Collapse

Furthermore, Fig. 10(b) shows that our system maintains a Jains Fairness Index near 1.0, whereas Baselines drop to 0.25–0.35. This confirms that the deterministic plan queue effectively prevents the starvation of queries accessing contended regions.

7.3.3 Storage-Level Effects and Request Collapse

To explain the performance gains observed above, we analyzed the internal I/O behavior. Fig. 11 compares the physical data movement against logical query demands. Note that in this experiment, Baseline A and Baseline B are grouped as a single baseline, as neither implements window-level coordination.

Fig. 11(a) and Fig. 11(b) demonstrate the Request Collapse effect. While 64 concurrent clients generate 12,800 IOPS in the baseline, our system collapses them into fewer than 600 physical operations. Fig. 12 quantifies this using the Merging Efficiency. As the overlap ratio σ increases, the I/O amplification factor of our system drops linearly from 1.0 to 0.15. This mathematically proves that the throughput gains are derived from a fundamental reduction in physical I/O volume rather than mere CPU scheduling.

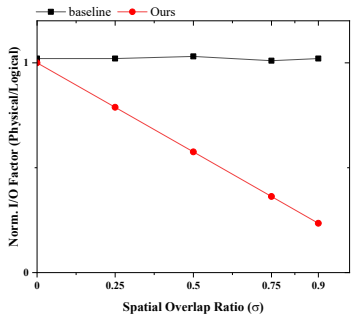


Fig. 12. Merging Efficiency

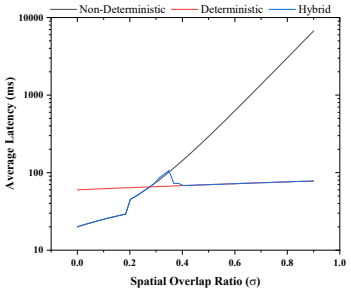


Fig. 13. Adaptive Performance Switching

7.3.4 Deterministic vs Non-Deterministic Modes

We then validated the effectiveness of the hybrid switching logic by comparing it against static Forced Optimistic" and Forced Deterministic" policies. As shown in Fig. 13, the static policies exhibit distinct weaknesses: the Deterministic mode incurs high coordination overhead ($\approx$ 60 ms) at low σ, while the Optimistic mode suffers from exponential thrashing at high σ. The Hybrid curve successfully tracks the lower performance envelope of the two.

7.3.5 Microbenchmark of Window Merging

Finally, we dissected the efficiency of the three-stage reduction pipeline. Fig. 14(a) shows that the combination of Deduplication (Stage 1) and Range Merging (Stage 2) achieves a cumulative reduction in request count consistent with the findings in Section 5.3.3.

Fig. 14(b) presents the Run Length Distribution (CDF) of I/O requests. The proposed mechanism shifts the I/O pattern from small, fragmented reads (typical in baselines) to larger, sequential chunks, which significantly amortizes disk seek times.Fig. 14(c) presents the cost-benefit analysis. The CPU overhead of the dispatcher remains negligible ($< 2.5\mu s$ per window) compared to the benefit of achieving a $> 90\%$ zero-copy ratio, verifying that the algorithmic complexity of coordination yields a high return on investment in terms of system throughput.

7.4 Evaluating the I/O tuning

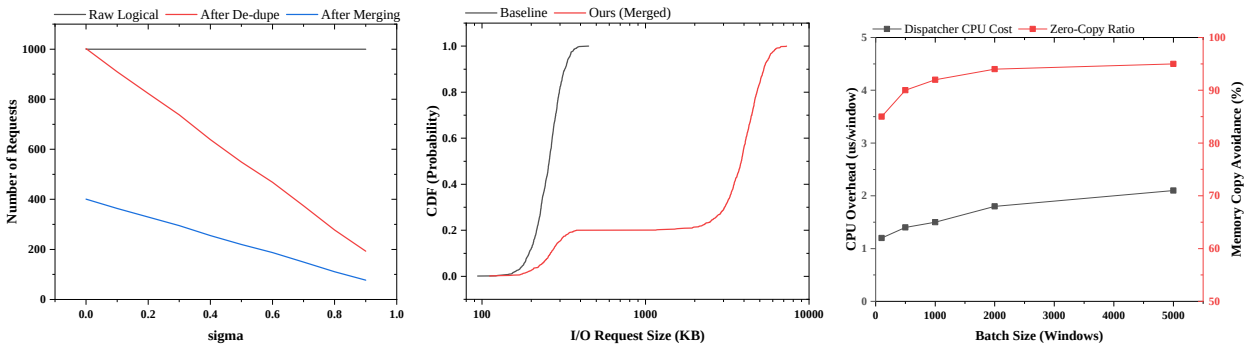
In this section, we evaluate the effectiveness of the proposed SA-GMAB tuning framework. The experiments are designed to verify four key properties: fast convergence speed, robustness against stochastic noise, adaptability to workload shifts, and tangible end-to-end performance gains.

7.4.1 Convergence Speed and Tuning Cost

First, we initiated a cold-start tuning session to evaluate how efficiently each method identifies high-quality configurations. Fig. 15 reports the convergence trajectory, cumulative tuning cost, and search efficiency.

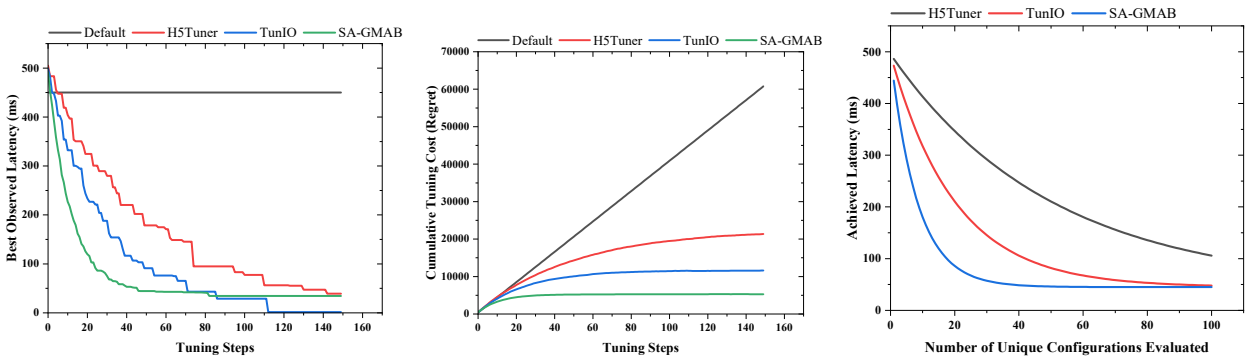
As shown in Fig. 15(a), the **Default** configuration remains trapped in a high-latency state (≈ 450 ms). While **H5Tuner** and **TunIO** gradually improve performance, they exhibit slow decay rates, requiring over 80 steps to stabilize. In contrast, **SA-GMAB** achieves a sharp drop in best-observed latency within the first 15–20 steps. This acceleration is attributed to the surrogate model, which effectively prunes unpromising configurations before costly execution.

Fig. 15(b) plots the cumulative tuning overhead (regret). The steep slope of the GA-based baselines indicates that they repeatedly explore poor configurations due to their memory-less nature. Our method exhibits the flattest curve, minimizing the cumulative performance loss during exploration. Furthermore, Fig. 15(c) confirms the high sample efficiency: SA-GMAB reaches the near-optimal zone (≈ 50 ms) after evaluating significantly fewer unique configurations compared to H5Tuner and TunIO.



(a) Reduction Pipeline (b) Run Length Distribution (c) Cost-Benefit Analysis

Fig. 14. Microbenchmark of Window Merging



(a) Convergence Speed (b) Cumulative Tuning Overhead (c) Search Efficiency

Fig. 15. Convergence Speed and Tuning Cost

7.4.2 Robustness under Stochastic Interference

In concurrent I/O environments, performance measurements are inherently noisy. Fig. 16 evaluates the robustness of the tuning algorithms under such stochastic interference.

Fig. 16(a) tracks the instantaneous reward over time. **H5Tuner** exhibits high variance, frequently dropping to low-reward regions because it discards good configurations that perform poorly once due to transient noise. In contrast, **SA-GMAB** maintains a stable high-reward trajectory. By aggregating historical observations in the memory table, our method "smooths out" the noise and correctly identifies optimal configurations despite fluctuations. Fig. 16(c) further breaks down the decision quality. Our method selects the **Optimal Configuration** for **88%** of the rounds, whereas H5Tuner selects it only **35%** of the time, wasting the majority of its budget on suboptimal or poor parameters.

7.4.3 Adaptation to Workload Shifts

We then investigated the system's ability to adapt to non-stationary environments. We introduced a sudden workload shift at $t = 60$, changing the query pattern from sparse random access to dense sequential scans.

As illustrated in Fig. 17(a), the shift causes an immediate latency spike (> 300 ms) for all methods. The **Default** policy fails to adapt. **H5Tuner** reacts sluggishly, requiring many generations to evolve parameters for the new regime. **SA-GMAB**, however, detects the context change and leverages its surrogate model to rapidly propose new candidates, achieving a full recovery to the new optimal latency (≈ 80 ms) within fewer than 15 batches (Fig. 17(c)).Fig. 17(b) traces the evolution of the *Merge Threshold* parameter. While baselines drift slowly, our method executes a decisive shift from 0.2 to 0.8, effectively locking onto the new optimal region required by the sequential workload.

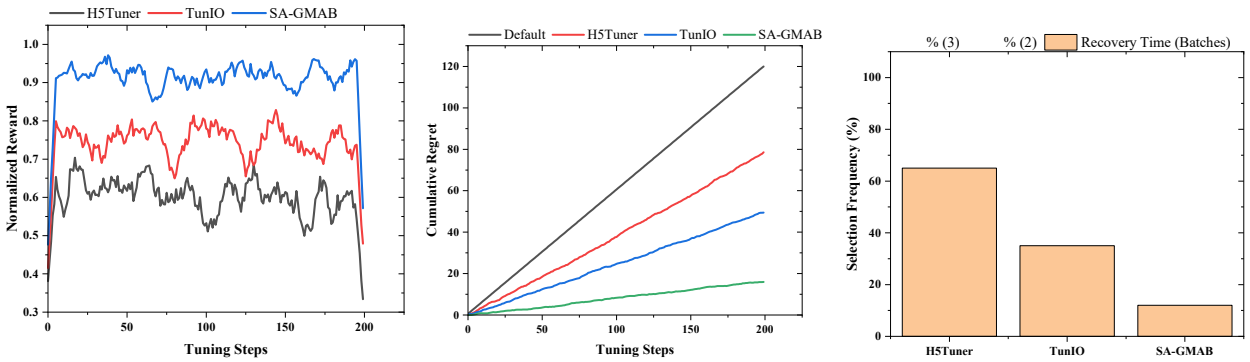
7.4.4 Impact on End-to-End Query Performance

Finally, we measured the steady-state performance of the fully optimized system. Fig. 18 compares the end-to-end metrics across different tuning methods.

Fig. 18(a) presents a latency trace during steady-state operation. While **Default** suffers from high latency and **GA-based** methods exhibit jitter due to unstable exploration, **SA-GMAB** maintains a consistently low and smooth latency profile (≈ 45 ms). This stability is critical for meeting SLA requirements in real-time analytics. Fig. 18(b) summarizes the aggregate throughput gain. Our method achieves a **5.6**$\times$ improvement over the default configuration. Fig. 18(c) reveals the underlying reason: under high contention, the tuner automatically selects aggressive batching and merging parameters, driving the I/O amplification factor down to **0.2**. This confirms that SA-GMAB effectively aligns the system configuration with real-time workload characteristics to maximize I/O efficiency.

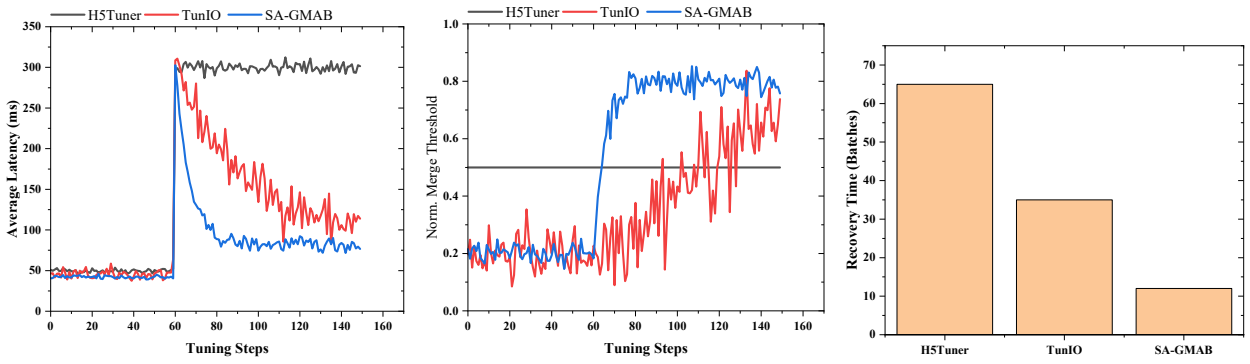
8 Conclusions

Modern high-performance remote sensing data management systems face a critical bottleneck shift from metadata discovery to data extraction, driven by prohibitive runtime geospatial computations (C_{geo}) and severe I/O contention under concurrent access. This paper presents a comprehensive I/O-aware query processing framework designed to strictly bound query latency and maximize throughput for large-scale spatio-temporal analytics. By introducing the "Index-as-an-Execution-Plan" paradigm and a dual-layer inverted structure (G2I and I2G), we bridge the semantic gap between logical indexing and physical storage, effectively shifting the computational burden from query time to ingestion time.To address the scalability challenges in multi-user environments, we developed a hybrid concurrency-



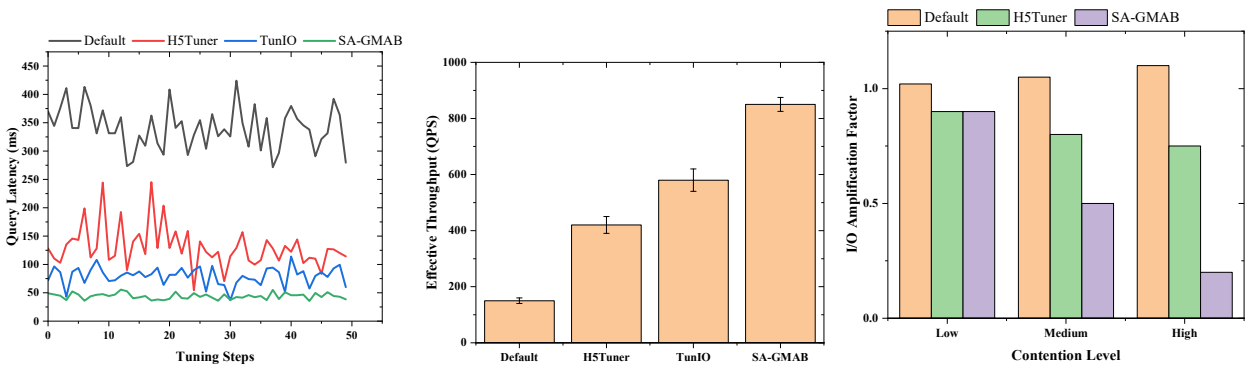
(a) Reward Stability under Noise (b) Regret Growth (c) Configuration Stability

Fig. 16. Robustness under Stochastic Interference



(a) Response to Workload Shift (b) Parameter Adaptation (c) Speed of Adaptation

Fig. 17. Adaptation to Workload Shifts



(a) Steady-State Stability (b) End-to-End Throughput (c) I/O Efficiency Tuning

Fig. 18. Impact on End-to-End Query Performance

aware I/O coordination protocol that adaptively switches between deterministic ordering and optimistic execution based on spatial contention. Furthermore, to handle the complexity of parameter configuration in fluctuating workloads, we integrated a Surrogate-Assisted Genetic Multi-Armed Bandit (SA-GMAB) mechanism for online automatic I/O tuning.Our empirical evaluation on large-scale Sentinel-2 datasets demonstrates that the proposed I/O-aware index reduces end-to-end latency by an order of magnitude compared to standard window-based reading approaches. The hybrid coordination mechanism effectively converts I/O contention into request merging opportunities, achieving linear throughput scaling significantly superior to traditional isolated execution. Additionally, the SA-GMAB tuning method exhibits faster convergence speed and greater robustness against stochastic noise compared to existing genetic baselines. These findings provide a scalable and predictable path for next-generation remote sensing platforms to support real-time, data-intensive concurrent workloads.

ACKNOWLEDGMENTS

This work is supported in part by the National Natural Science Foundation of China (No. U21A2013, No. 41925007 and No. 62076224), Open Research Project of The Hubei Key Laboratory of Intelligent Geo-Information Processing(KLIGIP-2019B14).

REFERENCES

[1] "Remote sensing big data computing: Challenges and opportunities," *Future Generation Computer Systems*, vol. 51, pp. 47–

60, 2015, special Section: A Note on New Trends in Data-Aware Scheduling and Resource Provisioning in Modern HPC Systems.

[2] J. M. Haut, M. E. Paoletti, S. Moreno-Álvarez, J. Plaza, J. Rico-Gallego, and A. Plaza, "Distributed deep learning for remote sensing data interpretation," *Proc. IEEE*, vol. 109, no. 8, pp. 1320–1349, 2021.

[3] "The australian geoscience data cube foundations and lessons learned," *Remote Sensing of Environment*, vol. 202, pp. 276–292, 2017, big Remotely Sensed Data: tools, applications and experiences.

[4] J. Yan, Y. Liu, L. Wang, Z. Wang, X. Huang, and H. Liu, "An efficient organization method for large-scale and long time-series remote sensing data in a cloud computing environment," *IEEE J. Sel. Top. Appl. Earth Obs. Remote. Sens.*, vol. 14, pp. 9350–9363, 2021.

[5] H. Liu, J. Yan, J. Wang, D. Zhang, J. Li, L. He, and X. Yu, "Mstgi: a multi-scale spatio-temporal grid index model for remote-sensing big data retrieval," *Remote Sensing Letters*, vol. 15, no. 1, pp. 44–54, 2024.

[6] C. Strobl, "Postgis," in *Encyclopedia of GIS*, S. Shekhar and H. Xiong, Eds. Springer, 2008, pp. 891–898.

[7] R. E. O. Simões, G. R. de Queiroz, K. R. Ferreira, L. Vinhas, and G. Câmara, "Postgis-t: towards a spatiotemporal postgresql database extension," in *XVII Brazilian Symposium on Geoinformatics - GEOINFO 2016, Campos do Jordão, SP, Brazil, November 27-30, 2016*, C. E. C. Campelo and L. M. Namikawa, Eds. MCTIC/INPE, 2016, pp. 252–262.

[8] I. S. Suwardi, D. Dharma, D. P. Satya, and D. P. Lestari, "Geohash index based spatial data model for corporate," in *2015 International Conference on Electrical Engineering and Informatics (ICEEI)*. IEEE, 2015, pp. 478–483.

[9] J. N. Hughes, A. Annex, C. N. Eichelberger, A. Fox, A. Hulbert, and M. Ronquest, "Geomesa: a distributed architecture for spatio-temporal fusion," in *Geospatial informatics, fusion, and motion video analytics V*, vol. 9473. SPIE, 2015, pp. 128–140.

[10] N. Gorelick, M. Hancher, M. Dixon, S. Ilyushchenko, D. Thau, and R. Moore, "Google earth engine: Planetary-scale geospatial analysis for everyone," *Remote sensing of Environment*, vol. 202, pp. 18–27, 2017.

[11] "rio-tiler: User friendly rasterio plugin to read raster datasets," 2025. [Online]. Available: https://github.com/opendatalab/Earth-Agent

[12] A. Thomson, T. Diamond, S. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: fast distributed transactions for partitioned database systems," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, and A. Fuxman, Eds. ACM, 2012, pp. 1–12.

[13] H. Lim, M. Kaminsky, and D. G. Andersen, "Cicada: Dependably fast multi-core in-memory transactions," ser. SIGMOD '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2135.

[14] N. Rajesh, K. Bateman, J. L. Bez, S. Byna, A. Kougkas, and X. Sun, "Tunio: An ai-powered framework for optimizing HPC I/O," in *IEEE International Parallel and Distributed Processing Symposium, IPDPS 2024, San Francisco, CA, USA, May 27-31, 2024*. IEEE, 2024, pp. 494–505.

[15] D. Preil and M. Krapp, "Genetic multi-armed bandits: A reinforcement learning inspired approach for simulation optimization," *IEEE Trans. Evol. Comput.*, vol. 29, no. 2, pp. 360–374, 2025.

[16] J. Tang, Z. Zhou, K. Ning, Y. Sun, and Q. Wang, "A novel spatial indexing mechanism leveraging dynamic quad-tree regional division," in *Proceedings of the 2012 International Conference on Information Technology and Software Engineering*, W. Lu, G. Cai, W. Liu, and W. Xing, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 909–917.

[17] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Comput. Surv.*, vol. 13, no. 2, pp. 185–221, 1981.

[18] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Trans. Database Syst.*, vol. 6, no. 2, pp. 213–226, 1981.

[19] T. Wang and H. Kimura, "Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores," *Proc. VLDB Endow.*, vol. 10, no. 2, pp. 49–60, 2016.

[20] Y. Hong, H. Zhao, W. Lu, X. Du, Y. Chen, A. Pan, and L. Zheng, "A hybrid approach to integrating deterministic and non-deterministic concurrency control in database systems," *Proc. VLDB Endow.*, vol. 18, no. 5, pp. 1376–1389, 2025.

[21] H. Wu, M. Zhang, K. Chen, X. Liao, Y. Shan, and Y. Wu, "OOCC: one-round optimistic concurrency control for read-only disaggregated transactions," in *41st IEEE International Conference on Data Engineering, ICDE 2025, Hong Kong, May 19-23, 2025*. IEEE, 2025, pp. 238–250.

[22] T. Chen and M. Li, "Multi-objectivizing software configuration tuning (for a single performance concern)," *CoRR*, vol. abs/2106.01331, 2021.

[23] J. L. Bez, F. Z. Boito, R. Nou, A. Miranda, T. Cortes, and P. O. A. Navaux, "Adaptive request scheduling for the I/O forwarding layer using reinforcement learning," *Future Gener. Comput. Syst.*, vol. 112, pp. 1156–1169, 2020.

[24] B. Behzad, J. Huchette, H. V. T. Luu, R. A. Aydt, S. Byna, Y. Yao, Q. Koziol, and Prabhat, "A framework for auto-tuning HDF5 applications," in *The 22nd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'13, New York, NY, USA - June 17 - 21, 2013*, M. Parashar, J. B. Weissman, D. H. J. Epema, and R. J. O. Figueiredo, Eds. ACM, 2013, pp. 127–128.