# LegoIndex: A Scalable and Modular Indexing Framework for Efficient Analysis of Extreme-Scale Particle Data

Chang Guo
Arizona State University
Tempe, USA
cguo51@asu.edu

Ning Yan
Georgia State University
Atlanta, USA
nyan1@student.gsu.edu

Lipeng Wan
Georgia State University
Atlanta, USA
lwan@gsu.edu

Zhichao Cao
Arizona State University
Tempe, USA
Zhichao.Cao@asu.edu

## Abstract

Particle-in-Cell (PIC) simulations play a critical role in various scientific domains, including plasma physics, astrophysics, and fusion energy research, by enabling the modeling of complex interactions between charged particles and electromagnetic fields. As PIC simulations scale up in size and complexity, they generate massive volumes of particle data at enormous speeds (TBs/hour). This enormous amount of data presents significant challenges for post-simulation analysis, as existing analysis tools (typically designed for smaller datasets) struggle with low query performance and high resource utilization. While incorporating indexes for PIC data can alleviate some of these inefficiencies, current indexing solutions often fall short of addressing the diverse analysis needs of scientists, substantial index construction overhead during simulation runs, and inefficient small I/O operations.

To address these challenges, we present **LegoIndex**, a scalable, modular, and elastic post-simulation indexing framework designed to improve data access efficiency while minimizing unnecessary data I/Os and memory usage. **LegoIndex** offers users the flexibility to customize indexing components, structures, and statistic metrics based on the data scale and specific analysis needs. **LegoIndex** parallelizes the index construction process, enabling efficient processing of datasets of varying sizes. To further enhance query performance, **LegoIndex** intelligently clusters scattered index results that are spatially or temporally related and optimizes computation logic to effectively reduce data I/Os and memory usage. We conducted comprehensive evaluations of **LegoIndex** based on large-scale real-world PIC datasets. Integrating **LegoIndex** with the existing analysis tool can achieve up to a **2276×** improvement in overall performance, a **3068×** reduction in memory usage, and a **3001×** decrease in I/O numbers for large datasets.


*HPDC '25', Notre Dame, IN, USA*

ACM ISBN 979-8-4007-1869-4/2025/07
https://doi.org/10.1145/3731545.3731591

## CCS Concepts

• **Information systems → Record storage systems**; • **Computing methodologies → Parallel computing methodologies**.

## Keywords

Scientific Data, Indexing, High-Performance Computing, Particle Data

**ACM Reference Format:**
Chang Guo, Ning Yan, Lipeng Wan, and Zhichao Cao. 2025. LegoIndex: A Scalable and Modular Indexing Framework for Efficient Analysis of Extreme-Scale Particle Data. In *The 34th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '25), July 20–23, 2025, Notre Dame, IN, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3731545.3731591

## 1 Introduction

Particle-in-Cell (PIC) simulations are fundamental for modeling the interactions between charged particles and electromagnetic fields, providing deep insights into complex phenomena such as plasma dynamics, particle acceleration, cosmic ray propagation, and magnetic reconnection [31, 51, 55]. Modern PIC simulation tools such as WarpX [71], EPOCH [6], and Geant4 [2] leverage sophisticated numerical methods to resolve both the fields and particle dynamics at fine temporal and spatial resolutions. The continuous simulation space is discretized into a computational grid, with each unit (cell) containing electric and magnetic fields that affect particle motion. These tools are designed to harness the power of high-performance computing (HPC) systems, utilizing thousands of processors or even GPUs in parallel to perform the simulation tasks. As a result, PIC simulations generate enormous volumes of data, with large-scale runs often producing terabytes of particle data per hour [34].

Once a PIC simulation is completed, scientists engage in *post-simulation* analysis to extract meaningful insights from the massive datasets generated. Various types of analytic tasks can be involved in post-simulation analysis. One of the most common tasks is performing range queries, where scientists search for particles based on specific criteria, such as position, momentum, energy, or charge [14, 28]. For instance, researchers might query particles within a particular spatial region or those with a momentum range exceeding a threshold to isolate interesting phenomena like shock

formation or high-energy events [61, 82]. Another example is particle tracking, where the focus is on monitoring the evolution of specific subsets of particles over time. This is particularly useful in understanding dynamic processes like particle acceleration, scattering, or confinement in magnetic fields [42, 62].

However, existing analysis tools, such as openPMD-viewer [47], ParaView [10], and H5py [20], face serious performance challenges as the scale of particle data increases. These tools, while effective for smaller datasets, struggle with the rapid growth in both the size and complexity of modern PIC simulations. Three critical issues arise: 1) As the dataset size increases, querying becomes prohibitively slow. For example, running a simple range query on a 1 TB particle dataset can take over an hour to complete due to the lack of efficient indexing and data retrieval mechanisms as discussed in subsubsection 3.1.1. 2) Existing analysis tools struggle to handle the growing scale of data, as they are designed to load the entire dataset into memory, even when only a small subset is needed for analysis. This approach causes serious read amplification and can quickly lead to memory exhaustion when working with multi-terabyte datasets, as shown in subsubsection 3.1.2. And 3) many existing analysis tools suffer from frequent small reads and unnecessary data loading, which cannot benefit from the high throughput of the underlying HPC parallel file system, as discussed in subsubsection 3.1.3, causing significant delays in accessing relevant data.

Motivated by the index schemes used in Online Analytical Processing (OLAP) scenarios, building an efficient index for PIC data analysis can be a promising solution. Analysis tools can dramatically improve query performance by pinpointing the relevant data based on the index searching result. Using index can also reduce overall memory and CPU usage by minimizing unnecessary data loading and processing. However, designing such an index for PIC data analysis comes with several key challenges. *First*, the scale, complexity, and structures of PIC data can be different. At the same time, scientists may use different types of queries during PIC data analysis, from point lookup, and range query, to partial tracking. One predefined single index (e.g., a B-tree) cannot ensure efficiency in all scenarios. Therefore, how to design an index framework that can adapt to diverse query operations and data characteristics is challenging. *Second*, the index is created after the PIC simulation is finished, and it is stored, transferred, and migrated together with the PIC data. Therefore, it is critical to ensure a fast and scalable index construction to avoid bottlenecks in the analysis pipeline, and to achieve lightweight and portable index structures for a low storage footprint and efficient transfer. *Third*, PIC data is usually stored in parallel file systems like Lustre [13, 66, 83], which prefer large and consecutive I/Os. How to leverage the information from the index to plan the efficient I/Os during analysis must be explored.

To address the aforementioned three challenges, we propose **LegoIndex**, a highly scalable and modular indexing framework designed for efficient analysis of extreme-scale particle data. *First*, **LegoIndex** draws inspiration from the modularity of a Lego model to fit diverse PIC data and analysis requirements, enabling users to customize indexing granularity (e.g., cell-level or multi-level in-cell indexing for specific cells) and index structures (e.g., linear-based or tree-based). The heterogeneous indexing structures at different levels are seamlessly connected through unique keys. Additionally, users can specify customizable statistical metrics at each level (e.g., minimum x-position or maximum y-momentum), enabling fine-grained support for data analysis.

*Second*, **LegoIndex** ensures efficient index construction, storage, and migration through several key strategies. To optimize construction efficiency, we introduce the *Bulk Load Scanner*, which dynamically adjusts read chunk sizes based on available memory and splits chunks into smaller tasks for multiple worker threads. For efficient storage and retrieval, **LegoIndex** partitions the index by levels, links them using the *key composition mechanism*, and stores the index as key-value pairs, ensuring compatibility with existing key-value stores or simple file-based storage. Since raw datasets are often shared with other scientists and later migrated to low-cost storage, **LegoIndex** is persisted alongside simulation data to remain accessible and operational.

*Third*, in **LegoIndex**, we introduce the *Dynamic Scanner* and *LegoMask* to enhance query efficiency. The *Dynamic Scanner* analyzes index traversal results to dynamically generate optimized I/O plans, grouping nearby I/O requests within a file into larger, more efficient read operations based on the characteristics of the underlying file system. Meanwhile, *LegoMask* improves efficiency by eliminating irrelevant cells in memory loaded during the grouping process, reducing unnecessary particle-level computations and filtering. Additionally, **LegoIndex** periodically processes historical performance of the underlying file system to update *Dynamic Scanner* parameters, ensuring adaptability to changing conditions and maintaining optimal performance.

We implemented a prototype of **LegoIndex** as a pluggable library, available as open-source on **GitHub**[1] and integrated with one of the most widely used PIC analyzing tools, *OpenPMD-viewer* [47]. We evaluated the query performance of **LegoIndex** using multiple particle datasets with various sizes (generated by running WarpX simulation on a supercomputer at Lawrence Berkeley National Laboratory(LBNL)) and compared it with the default *OpenPMD-viewer* [47] and a modified version of *OpenPMD-viewer* with *Min-Max Index (Zonemap)* enabled [9, 32, 38, 63]. Our results demonstrate significant improvements in query efficiency and resource utilization: **LegoIndex** achieves up to **2276×** and **6.8×** query performance improvement compared to *OpenPMD-viewer* and the *Min-Max Index* version, while reducing memory usage by up to **3068×** compared to *OpenPMD-viewer*. Even when the target data proportion increases to 90%, **LegoIndex** consistently delivers up to **44×** better performance than these baselines. The multi-level index structure provides up to **2.7×** better performance than a single-level index. Additionally, the *Bulk Load Scanner* design and parallelization strategies enable index construction speeds that are up to **3788×**, while the group read improves I/O efficiency by up to **2014×** over *OpenPMD-viewer* and **21.7×** over *Min-Max Index*. For particle visualization and tracking, **LegoIndex** achieves up to **3×** and **260×** better performance than *OpenPMD-viewer*, respectively.

## 2 Background

### 2.1 Data Generated by PIC Simulations

The Particle-in-Cell (PIC) technique is a widely used computational method for simulating the behavior of charged particles

[1] https://github.com/asu-idi/LegoIndex

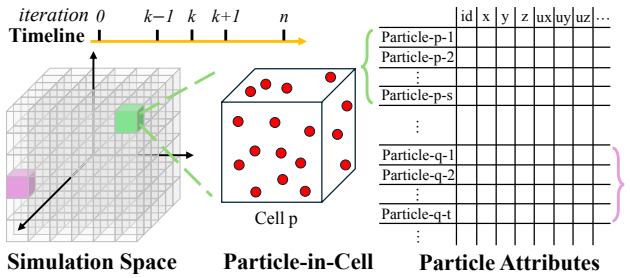



**Figure 1: Particle data organization in PIC simulations.**

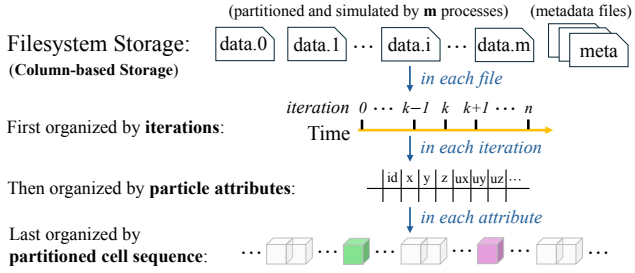



**Figure 2: Column-based storage of particle data on filesystem.**

interacting with electromagnetic fields. It has enabled the development of numerous advanced simulation codes, including WarpX [71, 72, 76], EPOCH [12], LSP [67], and Geant4 [7], which run on high-performance computing (HPC) systems with thousands of processors, simulating billions to trillions of particles in multi-dimensional space. These large-scale simulations are crucial for research in plasma physics, space science, and particle physics [5, 77, 81].

To gain insights from these simulations, scientists rely on output data capturing the dynamic state of each particle over time. Figure 1 illustrates how particle data is structured during the PIC simulations. The continuous simulation space is discretized into a computational grid, with each unit (cell) containing electric and magnetic fields that affect particle motion. Each particle within a cell is characterized by attributes such as position (x, y, z) and momentum (ux, uy, uz), representing its dynamic behavior. By periodically recording these attributes (called *iteration*), scientists can track individual particle trajectories and study their interactions with both electromagnetic fields and neighboring particles.

As depicted in Figure 2, particle data are stored in a distributed manner [27], with each process writing its assigned portion of the simulation space in parallel to optimize I/O performance on distributed parallel file systems such as Lustre [13, 66, 83] and GPFS [39, 65]. The data files are structured by simulation iteration, with each iteration's data appended incrementally as the simulation progresses, as shown in Figure 1. Within each iteration, particle attributes are serialized in a columnar format, storing each attribute contiguously across all particles in the sequence of their respective cells. Additionally, metadata files are generated to facilitate efficient data retrieval, and indexing details such as file locations, attribute start offsets, and sizes. While this storage strategy is optimized for high-throughput output in HPC environments, it often introduces inefficiencies during data access and analysis.

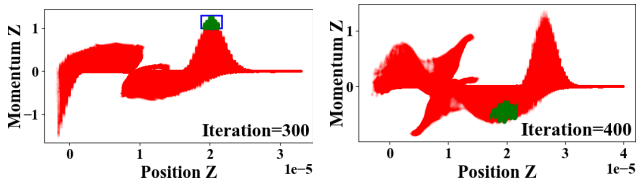
## 2.2 Workflows of Particle Data Analysis

Once particle data from PIC simulations is generated and stored, scientists often need to analyze specific particles (e.g., those whose momentum falls within a certain range) [37, 52], or perform more complex queries spanning multiple attributes. Figure 3 provides an example of a typical particle analysis workflow. Scientists often begin by examining overall particle distributions before zooming in on regions of interest with range queries. For instance, 3(a) shows the full particle distribution at iteration 300, where the focus is on particles exhibiting high momentum along the z-axis. By applying range queries, scientists can isolate these high-momentum particles and trace their positions and momentum over iterations, as illustrated in 3(b), which is the key to understanding particle dynamics. Currently, this analysis workflow requires at least two separate read operations from the file system: one to load the entire dataset for distribution calculations and another to reload it for the range query. This approach has two explicit drawbacks: 1) loading the entire dataset into memory is infeasible for large-scale particle data due to memory constraints, and 2) re-reading the entire dataset for each query is redundant and inefficient, introducing unnecessary overhead and significantly slowing down the analysis process. As PIC simulations grow in size and complexity, volumes of particle data reach petabyte scales, further exacerbating the inefficiencies of existing analysis tools.

## 2.3 Indexing Techniques for Particle Data

As mentioned in subsection 2.1, columnar storage formats, common in OLAP systems, are often used to store particle data to optimize particle data output performance. However, column-based storage introduces performance and efficiency drawbacks. On one hand, when queries target a small subset of rows (e.g., a few particles), columnar storage can be inefficient because large data blocks must still be scanned, leading to unnecessary I/Os. On the other hand, analytic queries that filter data based on one column (e.g., momentum) and retrieve data from another (e.g., position) require multiple reads across different blocks, increasing both I/O and CPU costs.

To mitigate these inefficiencies, existing columnar storage systems employ various indexing techniques to boost query performance. For instance, Parquet [38, 63] and Redshift [9, 32] employ block-level statistics (e.g., minimum and maximum values) to prune irrelevant blocks, reducing I/O and speeding up range queries. Systems like HBase support secondary indexes, providing faster lookups across distributed datasets [23, 44]. Additionally, partitioning techniques, as seen in BigQuery [4], allow datasets to be segmented based on attributes to improve overall query performance. While these techniques offer potential performance enhancements for columnar data, directly applying them to PIC data presents unique challenges. The multi-dimensional nature of particle attributes and the complexity of particle-based queries – such as spatial relationships, dynamic interactions, and trajectory tracking – require more advanced indexing strategies tailored to the unique demands of particle data analysis.




(a) Particle Overview and Selection (b) Particle Tracking

**Figure 3: Particle distribution, selection, and tracking.**

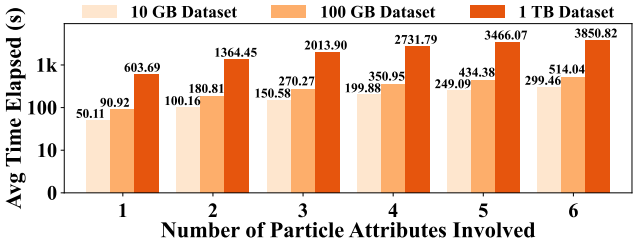
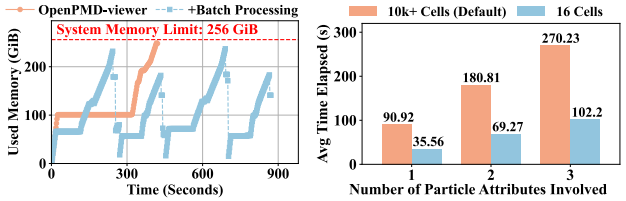
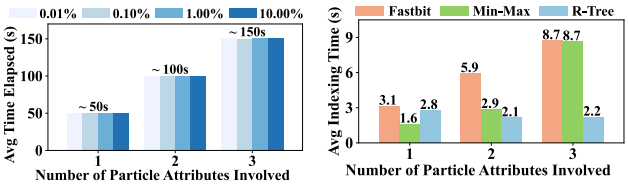



**Figure 4: Average query time across different dataset scales.**




(a) Memory footprint of existing analysis tools (b) Impact of I/O Block Size

**Figure 5: Memory usage and impact of I/O size.**




(a) Different Selection Proportions (b) Different Index Performance

**Figure 6: Influence of selection percentage and index types.**

# 3 Motivations and Challenges

## 3.1 Motivations

Our study is motivated by the pressing need to overcome the performance and resource limitations that domain scientists encounter when analyzing large-scale particle data with existing tools. Specifically, there are three key requirements motivating this effort.

*3.1.1 Reducing Particle Query Latency on Large Datasets.* As particle datasets continue to grow, query latency has increased substantially, slowing down the overall analysis workflow. To quantify this impact, we conducted 180 range query operations on three PIC datasets of varying sizes, generated by production runs of the WarpX simulation code on Perlmutter [54], an HPE Cray EX supercomputer at Lawrence Berkeley National Laboratory. The queries were randomly generated using our query generator (see subsubsection 5.2.2), each selecting 0.01% of the total particles and involving from 1 to 6 particle attributes. As shown in Figure 4, querying the 1TB dataset can take over an hour. Latency also increases with more particle attributes, adding computational overhead.

*3.1.2 Minimizing the Memory Footprint of Analysis Tools.* Existing analysis tools such as *OpenPMD-viewer* [47] allocate large memory spaces to load entire particle attributes from the filesystem. While effective for small datasets, this approach becomes unfeasible for larger datasets. For instance, as shown by the red curve in 5(a), querying a 1TB dataset initially required about 100GB of memory for a single attribute. When the system tried to allocate more memory for the next attribute, it exceeded the available memory, causing the process to terminate unexpectedly due to an out-of-memory error. Adopting a memory allocation strategy that loads smaller data batches and merges partial query results can mitigate memory usage, as shown by the blue curve in 5(a). However, this approach still traverses the entire dataset, which contributes to unnecessary memory consumption, I/O amplifications, and processing overhead. Consequently, there is a pressing need for a more efficient solution to minimize memory usage during particle query and analysis workflows.

*3.1.3 Improving I/O Efficiency.* To optimize data output performance for large-scale simulations on HPC systems, particle data is typically stored in chunked layouts, with each chunk containing one or more cells of the simulation space [27, 29]. However, small chunk sizes can trigger numerous small I/O operations during query operations, severely degrading efficiency. For example, reorganizing a 100 GB dataset from over 10,000 small cells into 16 larger cells significantly improved I/O efficiency, reducing query time to nearly one-third, as shown in 5(b). Inefficiencies also arise from unnecessary data reads, as detailed in subsection 2.2. An evaluation of query latency with varying selection proportions (0.01%, 0.10%, 1.00%, and 10.00%) on a 10 GB dataset in 6(a) revealed that latency is influenced more by the number of attributes and selection criteria than by the size of the target data. Even for queries selecting just 0.01% of the dataset, the existing analyzing tools (e.g., *OpenPMD-viewer* [47]) read the entire dataset into memory, resulting in query times similar to much larger selections, such as 10.00%. This underscores the urgent need for efficient mechanisms to minimize small I/O operations and eliminate unnecessary data reads.

## 3.2 Research Objective and Challenges

To achieve these required improvements in particle data analysis workflows, we propose to ***design and develop a scalable and modular post-simulation indexing framework, which indexes key attributes to speed up the queries and reduce resource utilization*** for facilitating query operations on large-scale particle data. However, developing this indexing framework requires addressing the following three inherent challenges due to the unique nature of the particle data.

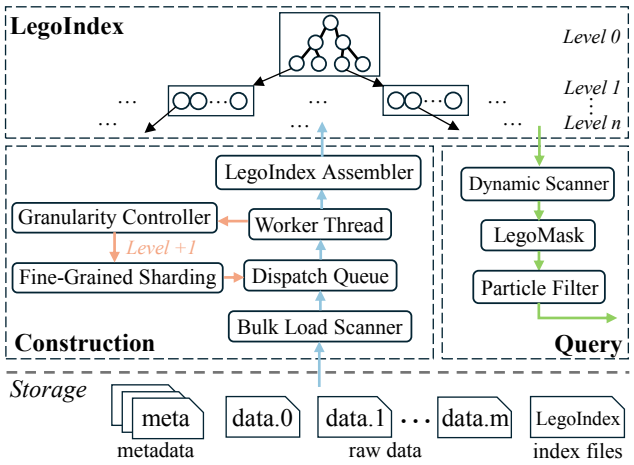



**Figure 7: LegoIndex: structure and workflow overview.**

**(1) Capability of Adapting to Various Analysis Tasks.** Scientists may perform various types of analysis queries, from range queries and particle tracking to more advanced operations like clustering, pattern recognition, and correlation studies. Different indexing structures exhibit varying performance across different types of queries, as shown in 6(b). Therefore, one of the key challenges is designing a modular indexing framework that allows scientists to customize index types and structures based on their specific analysis requirements. In particular, the framework should support configurable indexing granularity (e.g., multi-level fine-grained indexing for large or special cells) and offer various indexing structures (e.g., tree-based, bitmap-based, hash-based indexes, etc.) to optimize performance for different query patterns.

**(2) Efficient Index Construction, Storage, and Migration.** Index creation must be both efficient and scalable to avoid bottlenecks in the analysis pipeline, requiring algorithms that minimize computational overhead, even for massive datasets. After construction, the indices must be stored efficiently on persistent storage devices for future analysis tasks. Moreover, since scientific simulation data often transfers or migrates between storage systems, the framework must support seamless index migration.

**(3) Query Optimizations with Intelligent I/O Operation Planning and Scheduling.** While indexes significantly improve the identification and localization of data chunks that contain target particles, accessing these data often triggers inefficient I/O operations due to their small sizes and random distribution across the dataset. This strains the file system and leads to prolonged query times. To overcome these challenges, intelligent I/O planning and scheduling mechanisms are needed to dynamically optimize the overall query processing time. These mechanisms should optimize data access patterns by considering the characteristics of the queries, the dataset, and the underlying storage architecture.

## 4 LegoIndex

To tackle the challenges of building efficient indexes for large-scale particle datasets to improve query operations, we introduce LegoIndex, a scalable and modular indexing framework, as shown in Figure 7. LegoIndex offers three key features: 1) Inspired by Lego's modularity, LegoIndex allows users to customize the index, such as the number of index levels, selection of different index structures for each level, and embedded statistical metrics. This enables fine-tuning for specific analysis needs like spatial locality, temporal tracking, or attribute-based filtering. 2) LegoIndex speeds up the index construction process for massive particle datasets by efficiently parallelizing the process and distributing workloads across multiple processes or threads. And 3) LegoIndex employs a dynamic scanner and LegoMask mechanism to optimize data access patterns and reduce unnecessary I/Os during read operations.

### 4.1 A Modular Indexing Framework

*4.1.1 Adjustable Indexing Granularity.* Relying on a single-level index for each data cell still leads to prolonged query times, especially for large or skewed cells that often contain vast numbers of particles and are queried frequently. Without an internal index, analysis tools must load the entire cell into memory for exhaustive searches, causing high I/O and CPU overhead. For example, in formats like HDF5, a single cell can contain billions of particles. When only a small subset of particles is queried, the entire cell is read into memory, leading to significant read amplification and inefficiency.

To resolve this, LegoIndex introduces a customizable multi-level indexing approach combining a primary cell-level index with optional in-cell indexes for finer granularity, as shown in Figure 8. Unlike traditional two-level indexing systems, LegoIndex allows an arbitrary number of index levels within particularly large or dense cells. This design improves query performance by reducing read amplification: the primary index identifies relevant cells, and the in-cell index refines the search range within the cell.

Since constructing in-cell indexes for every cell is computationally expensive, LegoIndex introduces a *Granularity Controller* that selectively determines which cells require in-cell indexes based on data distribution and query patterns. By default, LegoIndex applies fixed-size partitioned linear in-cell indexes for large or frequently accessed cells. For more advanced use cases, users can define custom indexing policies to fine-tune the trade-off between indexing granularity and overhead. These policies allow users to specify conditions for triggering in-cell indexing, partitioning strategies, index structures, and statistical metrics, etc.

*4.1.2 Customizable Indexing Structures.* Given the complexity of particle data analyses, no single indexing structure can effectively handle all query scenarios. Traditional indexing like Zonemap [9, 32, 38, 63] and Bitmap [78] treat attributes independently, resulting in costly intersection operations when querying multiple attributes in the same query. Multi-dimensional index structures such as R-trees [33, 43, 46, 74] and Kd-trees [41, 53, 58] efficiently handle range queries across multiple attributes. However, their performance can degrade significantly when attribute ranges heavily overlap, like particle ID. Furthermore, queries like point lookups or particle tracking require different indexing structures (e.g., hash or special filters), highlighting the need for customizable indexing.

LegoIndex addresses this challenge by supporting various indexing structures, including linear-based, tree-based, bitmap, and Bloom filters, as shown in Figure 8, enabling users to select the

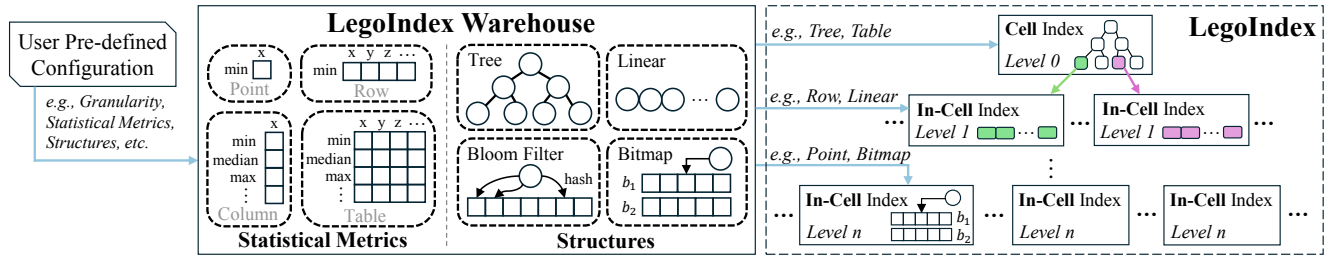



**Figure 8: Architecture and design overview of LegoIndex.**

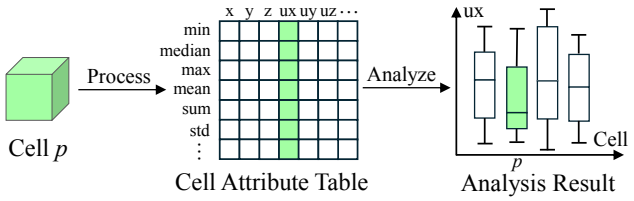



**Figure 9: Various cell statistics help analysis.**

optimal index and data partition for each level based on their workload characteristics. Each index level operates independently and is linked by unique keys, allowing seamless integration of heterogeneous indexing structures. For example, when tracking particles across iterations, a tree-based index at the top level (i.e., cell-level) can quickly narrow the search space, while an in-cell Bloom Filter index enables fast and precise data retrieval. Moreover, **LegoIndex** provides an open interface for integrating custom index structures (similar to operator overload in C++), ensuring flexibility to accommodate a wide range of query workloads.

Currently, users need to decide the number of levels and the index structures of each level before constructing the LegoIndex. By default, **LegoIndex** builds a single-level, cell-based R-Tree index, which provides strong performance for most multi-dimensional analysis workloads. For scenarios involving large cells or particle tracking across iterations, enabling multi-level indexing and selecting alternative structures, such as Bloom filters, at appropriate levels can better capture the data access patterns. In other scenarios like spatial k-nearest neighbor (kNN) queries, using a Kd-tree, or a hybrid approach that employs an R-Tree for coarse-grained pruning and an in-cell Kd-tree for fast, enables fine-grained search. The capability of supporting dynamic selection during index construction based on the data characteristics is planned for future work.

*4.1.3 Diverse Indexing Statistical Metrics.* Traditional indexing methods like the Min-Max index are optimized for range queries but often fall short for the diverse analytical needs in particle data analysis. Advanced metrics, such as standard deviation and emittance, are essential for identifying particle patterns, as shown in Figure 9, but indexing all possible metrics can lead to significant processing and storage overheads. **LegoIndex** addresses this by enabling customizable indexing metrics tailored to specific analysis needs, as shown in Figure 8, minimizing overhead while optimizing query performance. Users can select from various metrics, such as points (single attribute with one statistic), rows (multiple attributes with one statistic), columns (single attribute with multiple statistics), or tables (multiple attributes with multiple statistics).

**LegoIndex** supports a range of statistical metrics by three vectors: the first represents statistical methods (e.g., min, max, std), and the second corresponds to particle attributes (e.g., x, y, z). **LegoIndex** computes the interaction between these vectors to derive the necessary metrics for index construction. The third vector captures complex statistics derived from multiple attributes (e.g., emittance). During index construction, the selected metrics are extracted and organized based on the chosen index structure. To efficiently handle complex metrics, **LegoIndex** employs tailored integration strategies: in linear indexes, metrics are treated as unified entities for efficient traversal, whereas in tree-based structures (e.g., R-trees), they are represented as independent dimensions. In contrast, bitmap and hash-based indexes manage each metric separately, which can be inefficient for multi-criteria queries.

## 4.2 Index Construction, Storage, and Migration

Intuitively, **LegoIndex** can be constructed during simulations. However, it can introduce high latency and synchronization issues across thousands of Message Passing Interface (MPI) processes. It also limits the customizing possibilities of **LegoIndex** based on the analysis needs of different users. To address this, we propose constructing **LegoIndex** in a post-simulation phase, during the system's off-peak time. If users have a clear idea of the desired index levels, index structure, and statistical metrics, they can predefine a configuration file and initiate **LegoIndex** construction immediately after the simulation concludes. The construction workflow of **LegoIndex** is illustrated in Figure 10, which consists of loading, job creation, index creation, and index linking between levels.

*4.2.1 Post-simulation Index Construction.* Traditionally, index generation is integrated into the data creation process. For example, the ADIOS2 engine can calculate Min-Max values for particle data within each cell during simulation and write them alongside metadata. However, this inline indexing introduces an overhead of around 10-15% of total simulation time [21, 36], which is unacceptable for large-scale scientific simulation. To mitigate this, **LegoIndex** adopts a post-simulation index construction approach. Once the simulation completes and data is written to the file system, the index constructor can read the data from storage. However, as datasets continue to scale, loading all the data into memory is no longer feasible. Although cell-by-cell reading is supported, the small cell sizes can result in frequent small I/O operations, significantly impairing index construction performance.

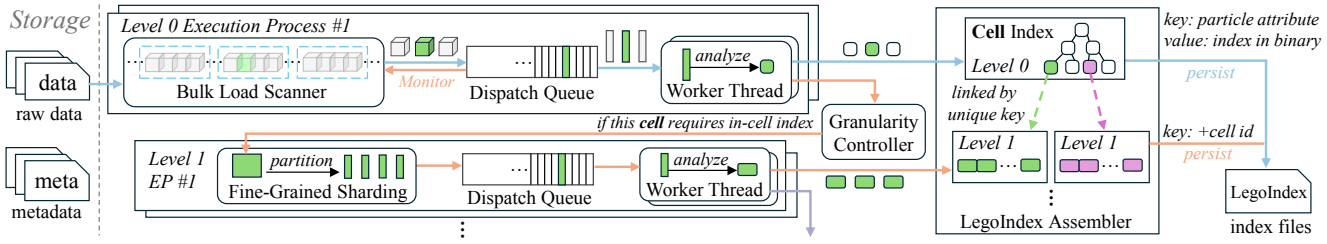



**Figure 10: LegoIndex construction workflow.**

To overcome this, **LegoIndex** introduces the *Bulk Load Scanner* as shown in Figure 10, a dedicated thread that optimizes data retrieval by balancing memory efficiency and I/O performance. It dynamically determines the optimal read chunk size based on available memory, fetching data in large chunks to reduce small I/O overhead. For large cells exceeding a single chunk's capacity, the *Bulk Load Scanner* processes them in multiple passes. Each chunk is then subdivided into smaller tasks, typically one cell per task, and placed in the *Dispatch Queue*. Our evaluations in subsubsection 5.4.2 show that a bulk load chunk size of 100MB to 1GB achieves an effective balance between I/O and CPU efficiency.

*Worker Threads* dequeue tasks from the Dispatch Queue, analyze data based on the demands of pre-defined index structure and statistical metrics, and send results to the ***LegoIndex*** *Assembler*, which integrates index components in the same level or across adjacent levels. Upon completion, the ***LegoIndex*** *Assembler* serializes each index as a key-value pair and stores it for future use. Additionally, *Worker Threads* report cell metadata (e.g., size, skewness) to the *Granularity Controller* to determine whether finer-grained in-cell indexing is required. If necessary, the cell data is further partitioned, analyzed, and submitted to the ***LegoIndex*** *Assembler*, while also being passed to the next-level *Granularity Controller* for refinement.

Throughout this process, *Bulk Load Scanner* continuously monitors the *Dispatch Queue*, prefetching data as needed to maximize I/O and CPU utilization. Furthermore, index construction at each level is performed in parallel by multiple processes, each leveraging multiple worker threads to enhance efficiency. This dynamic approach optimizes data handling, increases parallelism, and minimizes bottlenecks during index construction.

*4.2.2 Index Persistence and Storage.* As data scale increases, the size of the multi-level **LegoIndex** index grows accordingly, making it inefficient to store and load the entire index at once. To address this, **LegoIndex** introduces a *key composition mechanism* that leverages a key-value store to separate index levels and link them via unique keys. This design allows the query engine to load only the necessary index levels on demand, enabling fast lookups, efficient retrieval, and scalable index management.

For the top-level cell index, **LegoIndex** generates a unique key based on particle attributes such as species, simulation iteration, and attribute. For lower in-cell indexes, the key is further extended with the cell ID or global offset, ensuring uniqueness. **LegoIndex** partitions the index across levels and links them through these keys. For instance, when a leaf node in an upper-level index points to a finer-grained index at the next level, the corresponding key is derived by appending the current cell ID or global offset to the previous key. Furthermore, **LegoIndex** stores each index level separately, allowing users to retain frequently accessed levels (e.g., the top-level index) alongside the source data and dynamically reconstruct lower-level indexes on demand. During the index persistence, index data is serialized as values and stored as key-value pairs, either in a single file or in key-value stores such as Redis [17] or RocksDB [16, 24, 45, 69, 80], as illustrated in Figure 10.

*4.2.3 Index Sharing and Migration.* The PIC simulation data is usually stored on parallel distributed file systems in HPC environments. That data is often shared with and analyzed by scientists across multiple institutions all over the world. Additionally, when large-scale datasets are migrated to long-term, cost-effective storage solutions, it is crucial to maintain the accessibility and functionality of the associated index. To address these challenges, **LegoIndex** ensures that its data is persisted alongside the simulation data, as shown in Figure 10. This tight integration guarantees that the index remains fully operational, even when the dataset is transferred to remote or archival storage systems. Moreover, **LegoIndex** allows different scientists to construct custom indexes for the same dataset, tailored to their specific analysis needs, without disrupting the underlying data or other users' indexing schemes. This flexibility enhances collaborative data analysis while preserving the index's integrity across migrations.

## 4.3 Query Optimizations with LegoIndex

To accelerate PIC data analysis, **LegoIndex** can be integrated with existing tools and queried when analysis requests are made. When a user initiates a query to find a set of cells, **LegoIndex** is searched to pinpoint precise addresses of target cells in the files, avoiding the need to search through all the data and reducing read amplification. These cells are often dispersed across multiple files or scattered at different offsets within a single file. Reading these cells individually can result in inefficient, small random I/Os, especially for smaller cells or when in-cell indexing is used, exacerbating the issue.

One potential solution is to aggregate multiple targeted cells into a larger read operation. However, determining the optimal read size and strategy remains a challenge. **LegoIndex** addresses this with two key components in its query logic: the *Dynamic Scanner* and *LegoMask*, which work together to optimize query performance at runtime, as illustrated in Figure 11.

The *Dynamic Scanner* leverages particle count metadata embedded in **LegoIndex** to estimate read costs without accessing the actual data. To mitigate small I/O inefficiencies, the *Dynamic Scanner* groups nearby cells or in-cell data and reads them in a single

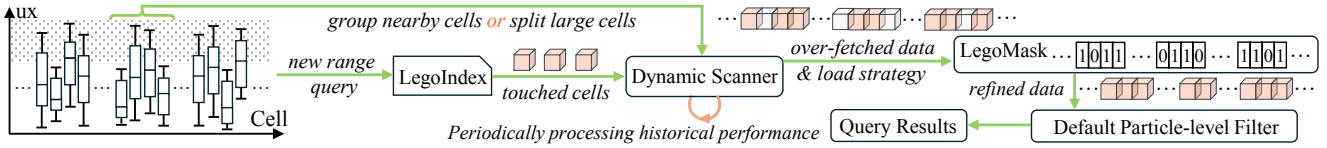



**Figure 11: LegoIndex intelligent I/O scheduling workflow.**

I/O operation. Additionally, **LegoIndex** dynamically adjusts its strategy in real-time based on historical performance feedback, ensuring optimized read performance across varying dataset sizes and system conditions. However, grouping nearby cells may inadvertently include irrelevant data that has already been filtered out by the index, leading to unnecessary computational overhead for particle-level filtering in memory. To address this, **LegoIndex** employs *LegoMask*, an efficient data masking technique that uses index information to generate a binary mask, eliminating irrelevant cells or in-cell segments before fine-grained filtering occurs. This significantly reduces CPU overhead from in-memory computations and boundary checks, thereby boosting query efficiency.

The overall query workflow with **LegoIndex** is illustrated in Figure 11. When a query arrives, **LegoIndex** first examines the top-level index to narrow down the target cells, then refines the search using the lower-level in-cell index to pinpoint finer-grained ranges. Once the target cells or data subsets within a cell are identified, the *Dynamic Scanner* employs a recursive algorithm to optimize the read strategy. The Scanner begins by identifying gaps between adjacent cells and determining the largest contiguous range. It then evaluates whether reading the entire range or splitting at the largest gap would yield better performance, based on filesystem I/O performance metrics obtained through historical queries. This recursive refinement continues until further splitting no longer improves performance. **LegoIndex** then generates a *LegoMask* as a binary mask based on the index results and applies it to the data chunk read using the Scanner's strategy. Finally, a particle-by-particle filter is applied to refine the query results.

When responding to the queries, iterating through a large-scale index can introduce noticeable delays once the number of index nodes exceeds one million. To address this, **LegoIndex** can support constructing coarser-grained indexes by grouping batches of cells together at the top-level and building finer-grained, cell-level indexes at the lower-level (e.g., level-1). During query execution, the system will adaptively load and traverse the multi-level index hierarchy as needed, effectively mitigating the performance impact of large-scale indexes. Additionally, users can adjust the *Granularity Controller* rules to selectively build next-level indexes only where needed, further reducing the overall index size.

# 5 Implementation and Evaluations

## 5.1 Prototype Implementation Details

We developed the prototype of **LegoIndex** as a library built on several robust technologies to ensure flexibility and performance. The prototype leverages ADIOS 2 v2.8.3 [27] as the PIC data I/O engine, GEOS v3.12.0 [25] for R-tree indexing support, Fastbit [78] for bitmap-based index support, Protobuf [50] for efficient index serialization/deserialization, Pybind11 v2.10 [40] for seamless integration with analysis tools, and RocksDB v8.0.0 [24] for index storage and migration. This design allows **LegoIndex** to be easily integrated with a wide range of PIC analysis tools. The source code for **LegoIndex**, along with the openPMD-viewer version integrated with **LegoIndex**, is publicly available on GitHub [1].

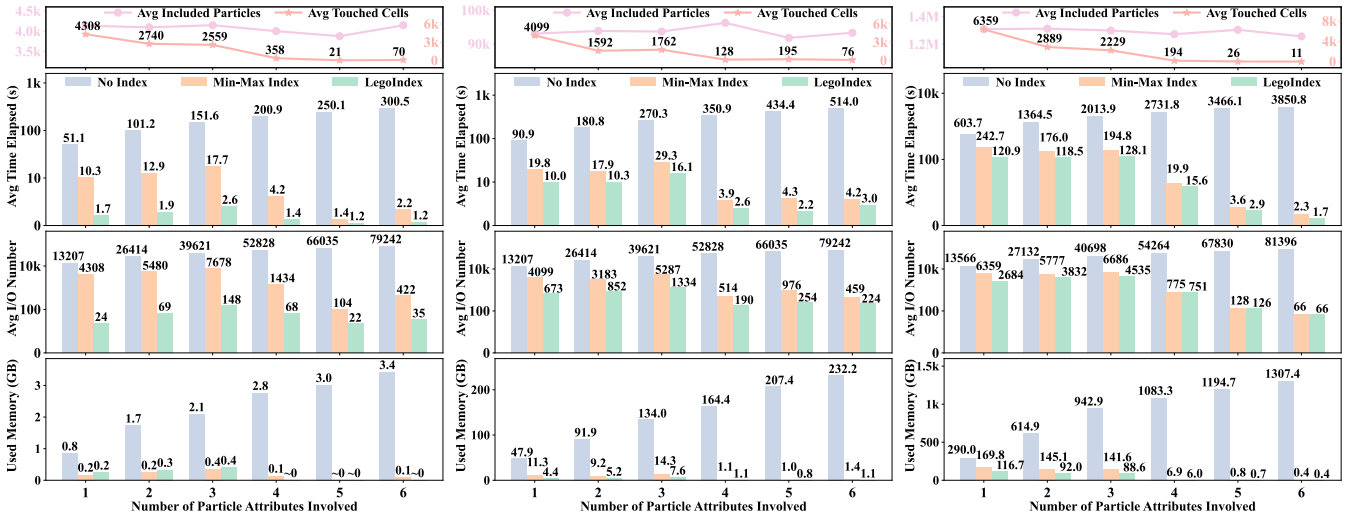
## 5.2 Experimental Setup

*5.2.1 Testbed and Datasets.* To ensure consistent and reliable performance measurements, all experiments were conducted in a controlled, dedicated environment to minimize system noise and variability. We used Dell EdgePower 650 servers equipped with Intel(R) Xeon(R) Silver 6330 CPUs, 256 GB of memory, and 480 GB of SSD storage, running Ubuntu Linux 22.04 LTS. The storage system utilized a 2 TB Samsung 980 PRO NVMe SSDs, offering sequential read and write speeds of 7,000 MB/s and 5,100 MB/s, respectively [64], as the backend for all testing datasets. The datasets used in our experiments consisted of three particle simulation datasets generated by the WarpX simulation code on the Perlmutter supercomputer at LBNL. These datasets varied in size, measuring 10 GB, 100 GB, and 1 TB per iteration, with each containing approximately 12,000 cells. All experiments were performed using these datasets to ensure consistency and scalability across varying data sizes.

*5.2.2 Particle Data Analysis Query Workloads and Baselines.* To comprehensively evaluate **LegoIndex**, we developed a particle data analysis query generator as a benchmark tool. This generator creates queries with varying query ranges and selects different numbers of particle attributes and selection proportions. The generator begins by randomly selecting attribute ranges and filters the data accordingly. It then evaluates the results against a desired selection percentage and dynamically adjusts the query range using machine learning-based convergence functions to align with the target proportion. For example, the generator can create a query to select N% of the dataset based on attributes such as momentum in the $x$ and $y$ directions. Unless otherwise specified, each experimental result presented represents the average of at least 10 random queries, maintaining consistent selection sizes, attribute counts, and proportions across the entire dataset. We use *OpenPMD-viewer* [37, 47], one of the most widely used particle analysis tools, as our primary baseline for comparison.

## 5.3 Overall Query Performance

*5.3.1 Query Performance Across Datasets of Different Sizes.* Using the query generator, we evaluated query performance by selecting a small proportion of data (0.01% of the total) with varying numbers of particle attributes across three datasets of different sizes.




(a) 10 GB Dataset – Small Size Cells (b) 100 GB Dataset – Medium Size Cells (c) 1 TB Dataset – Large Size Cells

**Figure 12: Overall query performance comparison.**

This comparison measured the performance of three configurations: the default openPMD-viewer (***No Index***[2]), openPMD-viewer with Min-Max indexing support (***Min-Max Index***), and openPMD-viewer integrated with **LegoIndex** (***LegoIndex***). For simplicity, **LegoIndex** constructs a top-level cell index using an R-tree structure and employs the minimum and maximum attribute values as statistical metrics.

The results, shown in Figure 12, use the *pink line* to indicate the average number of particles selected by the query and the *orange line* to represent the average number of relevant cells accessed. These are followed by three bar plots depicting the average query time, average I/O operations, and average memory usage for each approach. To emphasize performance differences, all bar plots are presented on a ***logarithmic scale***.

Across all dataset sizes, the default *No Index* exhibits a linear increase in query time as the number of selected attributes grows, primarily due to its inefficiency in loading the entire dataset regardless of query size. In contrast, both *Min-Max Index* and *LegoIndex* significantly enhance performance by processing only the relevant cells. These optimizations yield improvements of up to **1759**× and **2276**× in query time, and **635**× and **3001**× in I/O efficiency, **3216**× and **3068**× in memory usage, respectively.

The advantages of using indexing become increasingly significant as the number of accessed cells decreases. As shown in Figure 12, increasing the number of attributes involved in a query typically results in particles that satisfy the query conditions being concentrated in fewer cells. This phenomenon arises from the inherent locality of particle data—for example, particles with similar momentum values often reside within the same cell. In such scenarios, index structures are highly effective in filtering out unnecessary cell accesses, significantly improving query performance.

For instance, in 12(a), the query performance improvement provided by the *LegoIndex* increases substantially, from **30**× to **250**×, as the number of accessed cells decreases.

For smaller datasets, as in 12(a), the performance of *Min-Max Index* is influenced by both the number of touched cells and particle attributes. Increased attributes can inflate I/O operations, raising elapsed time (e.g., from 10.3s to 17.7s as attributes grow from 1 to 3), despite reduced cell access. *LegoIndex*, however, mitigates these effects through its *Dynamic Scanner* and *LegoMask*, which optimize performance by aggregating small I/O operations. When fewer cells are touched, these mechanisms have diminished impact, resulting in a reduction in performance improvement over *Min-Max Index* (e.g., from **6.8**× to **1.2**×).

As dataset sizes increase to 100 GB and 1 TB (12(b) and 12(c)), query times increase across all approaches. With total 12,000 cells, the particle number within each cell grows significantly, causing I/O time to dominate the overall query execution time. Consequently, the performance gains from indexing become less pronounced. As shown in Figure 12, the average number of I/O operations for *No Index* and *Min-Max Index* remains similar across different dataset sizes, while for *LegoIndex*, it increases significantly, approaching that of *Min-Max Index*. This reduces the impact of the *Dynamic Scanner*. However, despite this effect, *LegoIndex* maintains a performance advantage, though its improvements over *Min-Max Index* decrease from **2**× to **1.2**× as larger datasets mean fewer small I/Os.

*5.3.2 Memory Footprint of Query Operations.* We also monitored the memory usage during query processing, as shown in Figure 12. The default *No Index* exhibits significantly higher memory consumption because it loads the entire dataset into memory for each query operation, regardless of the query's scope. In contrast, *LegoIndex*, which employs the *Dynamic Scanner* to aggregate nearby cells into larger I/O units for improved performance, demonstrates memory efficiency comparable to the lightweight *Min-Max Index*. This

[2]Smaller batch processing was used for the 1 TB dataset to ensure completion

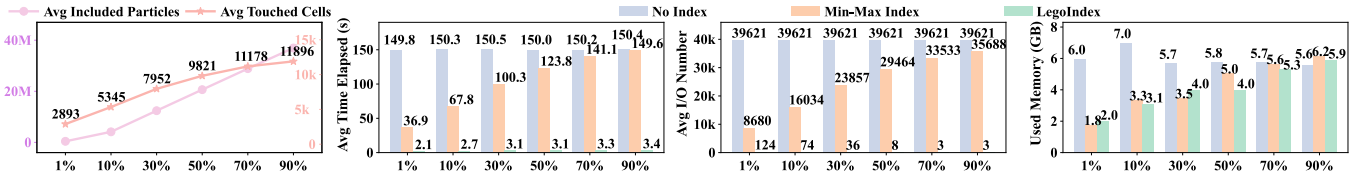



**Figure 13: Query performance at different selection proportions (x-axis: selection proportions).**

indicates that the optimizations introduced by the *Dynamic Scanner* do not incur any noticeable memory overhead. Notably, *LegoIndex* achieves substantial reductions in accumulated memory usage compared to the default *No Index*, with up to **259**× and **3068**× lower memory consumption on the 100 GB and 1 TB datasets, as shown in 12(b) and 12(c) respectively.

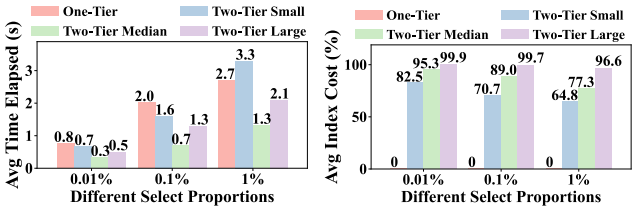
*5.3.3 Impact of Query Selectivity.* To thoroughly evaluate the impact of varying query selectivity on **LegoIndex**, we used the query generator to simulate queries with different selectivity levels (i.e., 1%, 10%, 30%, 50%, 70%, and 90% of the total dataset) involving three particle attributes, focusing on the 10 GB dataset, as illustrated in Figure 13. The default *No Index* exhibits no sensitivity to query selectivity, consistently requiring approximately 150 seconds, 40k I/Os, and 6 GB of Memory regardless of the proportion of data selected. This behavior is caused by loading and processing the entire dataset, resulting in uniform query costs.

In contrast, the *Min-Max Index* shows increasing query execution times as selectivity decreases (i.e., more data is selected), as a larger number of cells must be accessed. At high selectivity levels (e.g., 90%), the query performance of *Min-Max Index* converges with that of the default *No Index*, as nearly all cells are accessed, diminishing the effectiveness of the indexing. Conversely, *LegoIndex* demonstrates robust performance across all levels of query selectivity. By leveraging the *Dynamic Scanner* and *LegoMask*, *LegoIndex* effectively mitigates the impact of high selectivity by merging small I/O operations and pre-filtering irrelevant data. As a result, even when selecting 90% of the dataset, *LegoIndex* achieves a remarkable **44**× performance improvement over both the *Min-Max Index* and the default *No Index*, while maintaining a similar memory footprint.

## 5.4 Feature-Wise Performance Analysis

To understand how different features of **LegoIndex** impact performance, we conducted a series of experiments focused on various aspects such as multi-level indexing, construction performance, and intelligent I/O scheduling.

*5.4.1 Multi-level Indexing Design in LegoIndex.* We further explored the effectiveness of **LegoIndex** with different levels of indexing on a 100 GB dataset. The configurations included a single cell-level index (*One-Tier, **LegoIndex** default configuration*) and multi-level indices (*fine-tuned*) that partitioned cells into slices of varying granularities, ranging from coarse-grained (*Two-Tier Small*), to mid-grained (*Two-Tier Median*), to fine-grained (*Two-Tier Large*). As shown in Figure 14, the use of multi-level indexing resulted in up to **2.7**× better performance than the *One-Tier* configuration. Performance gains increased with finer-grained indexing (from *One-Tier* to *Two-Tier Median*), but at the finest granularity (*Two-Tier Large*,




(a) Query Average Cost (In-Memory). (b) LegoIndex Cost Proportion (In-Memory)

**Figure 14: Query performance across LegoIndex granularity.**

where slices contained only 100 particles), indexing overheads became noticeable. A detailed breakdown (14(b)) highlights a trade-off between granularity and indexing overhead: finer granularity enhances query precision but accounts for a larger proportion of the overall cost. Additionally, finer-grained indexing incurs significant storage overhead, with *Two-Tier Large* consuming 1.7 GB compared to just 179 KB for *Two-Tier Small*.

*5.4.2 Performance of LegoIndex Construction.* We evaluated the construction cost on the 10 GB dataset, which involves building a top-level cell index using an R-tree structure, to assess the effectiveness of the *Bulk Load Scanner* design and parallelization strategies. We tested *Bulk Load Scanner* by scanning between 1 and 10,000 cells per batch while varying the number of worker threads from 1 to 128 to analyze the performance impact. As shown in Figure 15, we separate the overall cost into three components: I/O time (left), representing the *Bulk Load Scanner* cost; CPU time (middle), reflecting the impact of different worker configurations; and total construction time (right).

On the left side of Figure 15, the I/O time decreases nearly linearly (by approximately *10*×) as the number of scanned cells increases from 1 to 100 per batch. However, the benefits diminish as the batch size grows further. Increasing the scan size from 100 to 1,000 cells results in only a *2*× reduction in I/O time, and beyond 1,000 cells, the improvement becomes negligible. This is because as the scan chunk size increases, the overhead of small I/O operations diminishes. Additionally, the I/O cost remains relatively stable across different numbers of worker threads (x-axis) since in-memory processing has minimal influence on I/O performance.

The CPU cost, shown in the middle of Figure 15, follows a similar trend. Increasing the number of threads reduces in-memory processing time, but the benefits diminish as thread count grows, eventually leading to lock contention overhead. This effect is particularly evident when scanning just 1 cell at a time. With a small number of threads, workers frequently remain idle, waiting for I/O

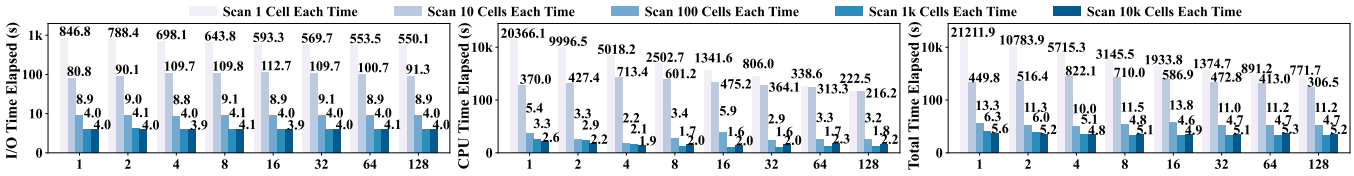
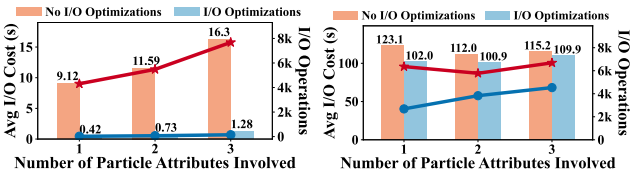
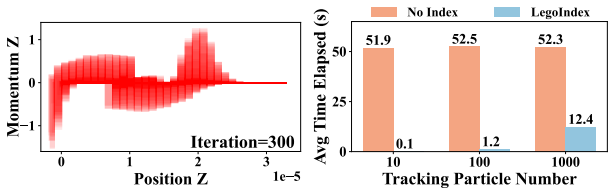



**Figure 15: Construction performance with varying scan sizes and worker thread numbers (x-axis: number of worker threads).**




(a) 10 GB Dataset – Small Size Cells (b) 1 TB Dataset – Large Size Cells

**Figure 16: LegoIndex intelligent I/O scheduling.**




(a) LegoIndex Approximate Visualization (b) Particle Tracking across Iterations

**Figure 17: Particle distribution visualization and tracking.**

operations to complete. Even with 128 threads, the total execution time still exceeds 200 seconds due to lock contention.

From the total construction time (right side of Figure 15), we observe that for the 10 GB dataset, scanning 100 to 1,000 cells per batch (i.e., 100 MB to 1 GB) with 4 to 8 worker threads achieves the highest efficiency. Further increasing resources does not yield significant performance improvements.

*5.4.3 Impact of Intelligent I/O Scheduling.* Finally, we evaluated the impact of *LegoIndex*'s *Dynamic Scanner* by comparing query performance with and without I/O optimization (Figure 16, where the red line shows the average touched cells and the blue line shows the actual I/Os with *Dynamic Scanner*). On smaller datasets, where query execution is dominated by small I/Os, *LegoIndex* demonstrated up to a **21.7**× improvement in average I/O time. For larger datasets, the benefits of grouping diminished as the number of small I/Os decreased, but *LegoIndex* still achieved a 10–20% improvement due to its efficient handling of I/O operations. These results highlight the adaptability of the *Dynamic Scanner* across different data scales, ensuring performance gains as dataset sizes grow (16(b)).

## 5.5 End-to-End Evaluation of Integrating LegoIndex with Real-World Analysis

In this section, we evaluate how **LegoIndex** enhances two critical particle analysis workflows: 1) approximate particle distribution visualization, and 2) particle tracking.

*5.5.1 Approximate Particle Distribution Visualization.* We visualized the particle distribution of the 10 GB dataset at iteration 300 using Python Matplotlib [8], leveraging cell-level information provided by **LegoIndex**. By varying opacity to represent particle density, the visualization was significantly faster compared to *No Index*. As shown in 17(a), *LegoIndex* reduced the visualization time to just 7.3 seconds, achieving up to a **3**× performance improvement over *No Index*, which required 23.1 seconds to plot all 40 million particles (3(a)). This acceleration is made possible by using aggregated metadata from *LegoIndex*, such as the minimal and maximal $z$-values of momentum and position, eliminating the need to load all particles. This efficient approach enables rapid approximate visualizations, making it a practical solution for exploratory data analysis in large-scale particle datasets.

*5.5.2 Particle Tracking.* We also evaluated particle tracking workflows by selecting and tracing particles' momentum in the $x$-direction between Iterations 400 and 500. The experiments involved tracking different numbers of particles—approximately 10, 100, and 1,000—from the 10 GB dataset. The performance of *LegoIndex*, using its tree-based index structure with integrated Bloom filters, was compared against the default *No Index*, as shown in 17(b). The *No Index* exhibited stable performance regardless of the number of particles being tracked, as it consistently loaded all particle IDs and sorted them to identify the targets. This exhaustive approach becomes inefficient for highly selective queries. In contrast, *LegoIndex* leveraged its indexing mechanism to localize particles efficiently, achieving up to a **260**× speedup when tracking only 10 particles. Nonetheless, the tracking cost of *LegoIndex* grows linearly with the number of particles being tracked, as each particle must traverse the Bloom filters. This highlights that *LegoIndex* is more efficient when tracking a limited number of particles, which aligns with typical use cases in scientific analysis and visualization [47].

# 6 Related Work

## 6.1 Indexing Techniques for Scientific Data

Various indexing techniques, such as Min-Max indexing, bitmaps, and tree-based structures (e.g., R-trees, Kd-trees), have been applied to improve data access for large scientific datasets. These methods aim to optimize query performance by organizing data spatially or storing attribute-based statistics.

Several research efforts (e.g., Slalom [56, 57], NoDB [3]) have explored adaptive partitioning strategies and fine-grained indexing on raw data, leveraging data distribution, query patterns, and attribute characteristics to enhance performance and flexibility for tabular

datasets. While particle attributes can be logically represented in a tabular format, particle data is typically stored in a columnar manner. Applying these partitioning methods without considering the underlying data organization can introduce overheads. In contrast, **LegoIndex** inherently utilizes the natural cells generated by the engine while remaining compatible with various partitioning strategies, ensuring efficient data access and adaptability.

In the context of particle data, several spatial indexing techniques have been explored. For example, k-d trees are used to preserve spatial locality, improving performance for nonuniform distributions and supporting low-latency reads and attribute-based filtering [70]. Balanced k-d trees also optimize range queries and traversal efficiency for real-time visualization [75]. For cosmological simulations, inverted indices efficiently track dark matter particles by exploiting physical properties of particle movement [22]. Additionally, techniques for optimizing query performance on shared parallel file systems, through dynamic scheduling and partial sorting, are explored in [79]. In HPC environments, the Indexed Massive Directory within DeltaFS [84, 85] enables high concurrency and efficient trajectory queries, while FastQuery [19, 29], which integrates FastBit indexing, demonstrates substantial performance improvements on supercomputing platforms.

Static indexing approaches often struggle with the dynamic nature of PIC datasets, where data distribution and access patterns evolve over time. Tree-based structures, while efficient for multidimensional queries, can degrade in performance when attribute ranges overlap significantly, leading to costly linear scans. **LegoIndex** addresses these issues with a flexible, multi-level indexing system combined with dynamic in-cell indexing. This approach allows researchers to tailor the indexing strategy to the evolving data and query characteristics, ensuring improved performance across diverse analysis tasks and data access patterns.

## 6.2 I/O Optimization for Scientific Applications

Efficient I/O is critical for large-scale data analysis, especially when dealing with the massive datasets generated by PIC simulations. Traditional parallel I/O libraries, such as HDF5 [29], PnetCDF [48], and ADIOS2 [27], have introduced various strategies to optimize data reading and writing, including bulk data aggregation and parallel data writing to minimize I/O costs during data generation.

H5bench [49] provides I/O benchmarks to evaluate and optimize HDF5 performance on parallel file systems. By identifying performance bottlenecks, H5bench significantly enhances I/O efficiency in HPC environments. The tool captures a range of I/O patterns, including data locality, array structures, and I/O modes, improving overall I/O performance. Two data layout reorganization strategies are introduced in [76], which optimize read and write performance for particle-mesh simulations, such as WarpX. In a separate study [18], an alternative approach to overcoming I/O bottlenecks was proposed, where an in-memory query system aggregates distributed memory across compute nodes. This technique eliminates disk-based data transfers, leading to substantial improvements in query performance. DLS [59] was introduced as a method to improve I/O efficiency by significantly reducing disk page accesses when querying complex tetrahedral mesh datasets. This approach accelerated query performance while preserving spatial query accuracy. Tree-based particle compression techniques are proposed in [35], enhancing I/O efficiency by large dataset sizes while preserving high reconstruction quality. Their traversal and partitioning schemes optimize memory usage and minimize reconstruction errors, thereby accelerating query performance and enabling faster data processing and analysis. A lightweight in situ framework is presented in [30], which enhances I/O efficiency by enabling visualization of sub-sampled snapshots without downloading full datasets or requesting extra compute resources. By minimizing data movement and enabling local access, it improves query performance and streamlines analysis. In [15], I/O efficiency improvements are achieved through the use of H5Part and FastQuery, enabling efficient analysis of trillion-particle datasets.

However, many existing scientific databases (e.g., SciDB [26, 68], TileDB [60], rasdaman [11]) face challenges when handling raw PIC data. The data's size and the specific nature of PIC queries introduce delays and storage overhead, making these systems less efficient compared to **LegoIndex**. The size of the data is often too large for these systems to handle efficiently, and the specific nature of PIC queries makes it difficult for these databases to support them directly. Furthermore, tools like HDF5 [29], ADIOS 2 [27], and Parquet [73] often rely on block-by-block reading, which leads to frequent small I/O operations and performance bottlenecks, especially with scattered access patterns. **LegoIndex** addresses this by using an intelligent read scheduler that dynamically groups or splits data into larger, more efficient I/O operations, improving memory usage and I/O performance during query execution.

## 7 Conclusion and Future Work

In this paper, we presented LegoIndex, a scalable and flexible indexing framework to significantly improve query performance and resource efficiency for the analysis of extreme-scale PIC datasets. By allowing users to customize indexing components and structures, **LegoIndex** minimizes unnecessary I/O operations and memory usage. Its multi-process architecture further accelerates index construction and query execution across diverse dataset scales.

Looking forward, we plan to extend **LegoIndex** to support a broader range of data types beyond PIC, making it applicable to a wider set of scientific and simulation domains. We also aim to integrate predictive heuristics and locality-aware strategies for dynamic index type selection and next-level detection, further optimizing indexing for varying access patterns. Additionally, we are exploring distributed index construction using MPI, enabling parallelized building of large-scale indices across multiple nodes to better support massive datasets.

## Acknowledgments

We extend our sincere gratitude to the anonymous reviewers for their constructive feedback. We are especially grateful to Dr. Axel Huebl and Dr. Remi Lehe at Lawrence Berkeley National Laboratory for their guidance and for providing access to computational resources on the Perlmutter supercomputer. We also acknowledge the members of the ASU-IDI Lab for their thoughtful comments and contributions. This work was partially funded by the National Science Foundation under Grant Number #2412436 and #2443219.

## References


[1] 2025. LegoIndex. https://github.com/asu-idi/LegoIndex

[2] Sea Agostinelli, John Allison, K al Amako, John Apostolakis, Henrique Araujo, Pedro Arce, Makoto Asai, D Axen, Swagato Banerjee, GJNI Barrand, et al. 2003. GEANT4—a simulation toolkit. *Nuclear instruments and methods in physics research section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 506, 3 (2003), 250–303.

[3] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. 2012. NoDB: efficient query execution on raw data files. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 241–252.

[4] Md Husen Ali, Md Sarwar Hosain, and Md Anwar Hossain. 2021. Big Data analysis using BigQuery on cloud computing platform. *Australian JofEng Inno Tech* 3, 1 (2021), 1–9.

[5] Raul E Angulo and Oliver Hahn. 2022. Large-scale dark matter simulations. *Living Reviews in Computational Astrophysics* 8, 1 (2022), 1.

[6] TD Arber, Keith Bennett, CS Brady, A Lawrence-Douglas, MG Ramsay, Nathan John Sircombe, Paddy Gillies, RG Evans, Holger Schmitz, AR Bell, et al. 2015. Contemporary particle-in-cell approach to laser-plasma modelling. *Plasma Physics and Controlled Fusion* 57, 11 (2015), 113001.

[7] Pedro Arce, Antonio Muñoz, Montserrat Moraleda, José María Gomez Ros, Fernando Blanco, José Manuel Perez, and Gustavo García. 2015. Integration of the low-energy particle track simulation code in Geant4. *The European Physical Journal D* 69 (2015), 1–5.

[8] Niyazi Ari and Makhamadsulton Ustazhanov. 2014. Matplotlib in python. In *2014 11th International Conference on Electronics, Computer and Computation (ICECCO)*. IEEE, 1–6.

[9] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J Green, Monish Gupta, Sebastian Hillig, et al. 2022. Amazon Redshift re-invented. In *Proceedings of the 2022 International Conference on Management of Data*. 2205–2217.

[10] Utkarsh Ayachit. 2015. *The paraview guide: a parallel visualization application*. Kitware, Inc.

[11] Peter Baumann. 2014. Rasdaman: Array databases boost spatio-temporal analytics. In *2014 Fifth International Conference on Computing for Geospatial Research and Application*. IEEE, 54–54.

[12] Keith Bennett, Chris Brady, Holger Schmitz, Christopher Ridgers, Tony Arber, Roger Evans, and Tony Bell. 2017. Users Manual for the EPOCH PIC codes. *University of Warwick* (2017).

[13] Peter Braam. 2019. The Lustre storage architecture. *arXiv preprint arXiv:1903.01955* (2019).

[14] Dominic AS Brown, Matthew T Bettencourt, Steven A Wright, Satheesh Maheswaran, John P Jones, and Stephen A Jarvis. 2021. Higher-order particle representation for particle-in-cell simulations. *J. Comput. Phys.* 435 (2021), 110255.

[15] Surendra Byna, Jerry Chou, Oliver Rubel, Prabhat, Homa Karimabadi, William S. Daughter, Vadim Roytershteyn, E. Wes Bethel, Mark Howison, Ke-Jou Hsu, Kuan-Wu Lin, Arie Shoshani, Andrew Uselton, and Kesheng Wu. 2012. Parallel I/O, analysis, and visualization of a trillion particle simulation. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–12. doi:10.1109/SC.2012.92

[16] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, modeling, and benchmarking RocksDB Key-Value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 209–223.

[17] J Carlson. 2013. *Redis in Action*. Manning.

[18] Hsuan-Te Chiu, Jerry Chou, Venkat Vishwanath, and Kesheng Wu. 2015. In-memory query system for scientific dataseis. In *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 362–371.

[19] Jerry Chou, Kesheng Wu, et al. 2011. Fastquery: A parallel indexing system for scientific data. In *2011 IEEE International Conference on Cluster Computing*. IEEE, 455–464.

[20] Andrew Collette, James Tocknell, Thomas A Caswell, Darren Dale, Ulrik Kofoed Pedersen, Aleksandar Jelenak, Andrea Bedini, Martin Raspaud, Jialin Lei, Laurence Hole, et al. 2017. H5Py/H5Py: 2.7. 1. (2017).

[21] ADIOS 2 contributors. 2024. ADIOS 2 Supported Engines. https://adios2.readthedocs.io/en/latest/engines/engines.html Accessed: 2024-10-16.

[22] Daniel Crankshaw, Randal Burns, Bridget Falck, Tamás Budavári, Alexander S. Szalay, and Jie Wang. 2013. Inverted indices for particle tracking in petascale cosmological simulations. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management* (Baltimore, Maryland, USA) *(SSDBM '13)*. Association for Computing Machinery, New York, NY, USA, Article 25, 10 pages. doi:10.1145/2484838.2484882

[23] Chen CUI, Linjiang ZHENG, Fengping HAN, and Mujun HE. 2018. Design of secondary indexes in HBase based on memory. *Journal of Computer Applications* 38, 6 (2018), 1584.

[24] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Transactions on Storage (TOS)* 17, 4 (2021), 1–32.

[25] GEOS contributors. 2024. *GEOS computational geometry library*. Open Source Geospatial Foundation. doi:10.5281/zenodo.11396894

[26] L Gerhardt, CH Faham, and Y Yao. 2015. Accelerating scientific analysis with SciDB. In *Journal of Physics: Conference Series*, Vol. 664. IOP Publishing, 072019.

[27] William F Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip Davis, Jong Choi, Kai Germaschewski, Kevin Huck, et al. 2020. Adios 2: The adaptable input output system. a framework for high-performance data management. *SoftwareX* 12 (2020), 100561.

[28] Sara Gomez, Jaime Humberto Hoyos, and Juan Alejandro Valdivia. 2023. Particle-in-cell method for plasmas in the one-dimensional electrostatic limit. *American Journal of Physics* 91, 3 (2023), 225–234.

[29] Luke Gosink, John Shalf, Kurt Stockinger, Kesheng Wu, and Wes Bethel. 2006. HDF5-FastQuery: Accelerating complex queries on HDF datasets using fast bitmap indices. In *18th International Conference on Scientific and Statistical Database Management (SSDBM'06)*. IEEE, 149–158.

[30] Pascal Grosset and James Ahrens. 2021. Lightweight interface for in situ analysis and visualization of particle data. In *ISAV'21: In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*. 12–17.

[31] Fan Guo, Yi-Hsin Liu, William Daughton, and Hui Li. 2015. Particle acceleration and plasma dynamics during magnetic reconnection in the magnetically dominated regime. *The Astrophysical Journal* 806, 2 (2015), 167.

[32] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1917–1923.

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

[34] W. Daughton B. Loring H. Karimabadi, W. H. Matthaeus. [n. d.]. Heating in the Solar Wind: A Study of PIC Simulations and Data Generation. https://sdm.lbl.gov/sdav/highlights/29-highlights/visualization/62-pic-sim.html. Accessed: 2025-02-07.

[35] Duong Hoang, Harsh Bhatia, Peter Lindstrom, and Valerio Pascucci. 2021. High-quality and low-memory-footprint progressive decoding of large-scale particle data. In *2021 IEEE 11th Symposium on Large Data Analysis and Visualization (LDAV)*. IEEE, 32–42.

[36] Axel Huebl. 2021. Issue of StatsLevel Influence the Write Performance in ADIOS 2. https://github.com/ornladios/ADIOS2/issues/2880 Accessed: 2024-10-16.

[37] Axel Huebl, Rémi Lehe, Jean-Luc Vay, David P. Grote, Ivo Sbalzarini, Stephan Kuschel, David Sagan, Christopher Mayes, Frédéric Pérez, Fabian Koller, Franz Poeschel, Carsten Fortmann-Grote, Angel Ferran Pousa, Juncheng E, Maxence Thévenet, and Michael Bussmann. 2015. openPMD: A meta data standard for particle and mesh based data. https://github.com/openPMD. doi:10.5281/zenodo.591699

[38] Todor Ivanov and Matteo Pergolesi. 2020. The impact of columnar file formats on SQL-on-hadoop engine performance: A study on ORC and Parquet. *Concurrency and Computation: Practice and Experience* 32, 5 (2020), e5523.

[39] Reshu Jain, Prasenjit Sarkar, and Dinesh Subhraveti. 2013. Gpfs-snc: An enterprise cluster file system for big data. *IBM Journal of Research and Development* 57, 3/4 (2013), 5–1.

[40] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. 2017. pybind11–Seamless operability between C++ 11 and Python. *URL: https://github. com/pybind/pybind11* (2017).

[41] You Jia, Jingdong Wang, Gang Zeng, Hongbin Zha, and Xian-Sheng Hua. 2010. Optimizing kd-trees for scalable visual descriptor indexing. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. IEEE, 3392–3399.

[42] Xiaolin Jin, Li Lei, Jibo Li, Tao Huang, Xiaoyan Zhang, Zhonghai Yang, and Bin Li. 2020. 3D particle-in-cell simulations of electron magnetic confinement in electron cyclotron resonance ion sources. *Review of Scientific Instruments* 91, 1 (2020).

[43] Ibrahim Kamel and Christos Faloutsos. 1992. Parallel R-trees. *ACM SIGMOD Record* 21, 2 (1992), 195–204.

[44] Zhiwen Kang, Peng Fang, Hongkui Xu, Bo Zhou, Fangcun Li, Sheng Zhou, Mingzhao Zheng, and Yao Li. 2022. Design of secondary index scheme for large-scale crowd behavior analysis data based on Elasticsearch and HBase. In *CAIBDA 2022; 2nd International Conference on Artificial Intelligence, Big Data and Algorithms*. VDE, 1–4.

[45] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald Dreslinski. 2021. Improving performance of flash based Key-Value stores using storage class memory as a volatile memory extension. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 821–837.

[46] Jinwoong Kim, Sul-Gi Kim, and Beomseok Nam. 2013. Parallel multi-dimensional range query processing with R-trees on GPU. *J. Parallel and Distrib. Comput.* 73, 8 (2013), 1195–1207.

[47] Remi Lehe. 2015. openPMD-viewer. https://github.com/openPMD/openPMD-viewer Accessed: 2024-10-16.

[48] Jianwei Li, Wei-keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Rob Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. 2003.

Parallel netCDF: A high-performance scientific I/O interface. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. 39.

[49] Tonglin Li, Suren Byna, Quincey Koziol, Houjun Tang, Jean Luca Bez, and Qiao Kang. 2021. h5bench: HDF5 I/O kernel suite for exercising HPC I/O patterns. In *Proceedings of Cray User Group Meeting, CUG*, Vol. 2021.

[50] Google LLC. 2023. openPMD-viewer. https://github.com/protocolbuffers/protobuf Accessed: 2024-10-16.

[51] Alexandre Marcowith, Gilles Ferrand, Mickael Grech, Zakaria Meliani, Illya Plotnikov, and Rolf Walder. 2020. Multi-scale simulations of particle acceleration in astrophysical systems. *Living Reviews in Computational Astrophysics* 6 (2020), 1–182.

[52] Mitchell Nelson, Zachary Sorenson, Joseph M Myre, Jason Sawin, and David Chiu. 2020. Parallel acceleration of CPU and GPU range queries over large data sets. *Journal of Cloud Computing* 9, 1 (2020), 44.

[53] Matheus Agio Nerone, Pedro Holanda, Eduardo C De Almeida, and Stefan Manegold. 2021. Multidimensional adaptive & progressive indexes. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 624–635.

[54] NERSC. 2022. Perlmutter. https://www.nersc.gov/systems/perlmutter Accessed: 2024-10-16.

[55] Kenichi Nishikawa, Ioana Duțan, Christoph Köhn, and Yosuke Mizuno. 2021. PIC methods in astrophysics: simulations of relativistic jets and kinetic physics in astrophysical systems. *Living Reviews in Computational Astrophysics* 7, 1 (2021), 1.

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

[57] Matthaios Olma, Manos Karpathiotakis, Ioannis Alagiannis, Manos Athanassoulis, and Anastasia Ailamaki. 2020. Adaptive partitioning and indexing for in situ query processing. *The VLDB Journal* 29, 1 (2020), 569–591.

[58] Beng C Ooi. 1987. Spatial kd-tree: A data structure for geographic database. In *Datenbanksysteme in Büro, Technik und Wissenschaft: GI-Fachtagung Darmstadt, 1.–3. April 1987 Proceedings*. Springer, 247–258.

[59] Stratos Papadomanolakis, Anastassia Ailamaki, Julio C Lopez, Tiankai Tu, David R O'Hallaron, and Gerd Heber. 2006. Efficient query processing on unstructured tetrahedral meshes. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 551–562.

[60] Stavros Papadopoulos, Kushal Datta, Samuel Madden, and Timothy Mattson. 2016. The tiledb array data storage manager. *Proceedings of the VLDB Endowment* 10, 4 (2016), 349–360.

[61] Jaehong Park, Jared C Workman, Eric G Blackman, Chuang Ren, and Robert Siller. 2012. Particle-in-cell simulations of particle energization from low Mach number fast mode shocks. *Physics of Plasmas* 19, 6 (2012).

[62] A Ferran Pousa, Ralph Assmann, and A Martinez de la Ossa. 2019. Wake-T: a fast particle tracking code for plasma-based accelerators. In *Journal of Physics: Conference Series*, Vol. 1350. IOP Publishing, 012056.

[63] Alice Rey, Michael Freitag, and Thomas Neumann. 2023. Seamless Integration of Parquet Files into Data Processing. In *BTW 2023*. Gesellschaft für Informatik eV, 235–258.

[64] Samsung. 2023. Samsung 980Pro. https://semiconductor.samsung.com/consumer-storage/internal-ssd/980pro/. Accessed: 2024-10-16.

[65] Frank Schmuck and Roger Haskin. 2002. {GPFS}: A {Shared-Disk} file system for large computing clusters. In *Conference on file and storage technologies (FAST 02)*.

[66] Philip Schwan et al. 2003. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium*, Vol. 2003. 380–386.

[67] Joseph R Smith, Chris Orban, Nashad Rahman, Brendan McHugh, Ricky Oropeza, and Enam A Chowdhury. 2021. A particle-in-cell code comparison for ion acceleration: EPOCH, LSP, and WarpX. *Physics of Plasmas* 28, 7 (2021).

[68] Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. 2011. The architecture of SciDB. In *Scientific and Statistical Database Management: 23rd International Conference, SSDBM 2011, Portland, OR, USA, July 20-22, 2011. Proceedings 23*. Springer, 1–16.

[69] Viraj Thakkar, Madhumitha Sukumar, Jiaxin Dai, Kaushiki Singh, and Zhichao Cao. 2024. Can Modern LLMs Tune and Configure LSM-based Key-Value Stores?. In *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems*. 116–123.

[70] Will Usher, Xuan Huang, Steve Petruzza, Sidharth Kumar, Stuart R. Slattery, Sam T. Reeve, Feng Wang, Chris R. Johnson, and Valerio Pascucci. 2021. Adaptive Spatially Aware I/O for Multiresolution Particle Data Layouts. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 547–556. doi:10.1109/IPDPS49936.2021.00063

[71] J-L Vay, A Almgren, J Bell, L Ge, DP Grote, M Hogan, O Kononenko, R Lehe, A Myers, C Ng, et al. 2018. Warp-X: A new exascale computing platform for beam–plasma simulations. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 909 (2018), 476–479.

[72] J-L Vay, A ea Huebl, A Almgren, LD Amorim, J Bell, L Fedeli, L Ge, K Gott, DP Grote, M Hogan, et al. 2021. Modeling of a chain of three plasma accelerator stages with the WarpX electromagnetic PIC code on GPUs. *Physics of Plasmas* 28, 2 (2021).

[73] Deepak Vohra and Deepak Vohra. 2016. Apache parquet. *Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools* (2016), 325–335.

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

[75] Ingo Wald, Aaron Knoll, Gregory P. Johnson, Will Usher, Valerio Pascucci, and Michael E. Papka. 2015. CPU ray tracing large particle data with balanced P-k-d trees. In *2015 IEEE Scientific Visualization Conference (SciVis)*. 57–64. doi:10.1109/SciVis.2015.7429492

[76] Lipeng Wan, Axel Huebl, Junmin Gu, Franz Poeschel, Ana Gainaru, Ruonan Wang, Jieyang Chen, Xin Liang, Dmitry Ganyushin, Todd Munson, et al. 2021. Improving I/O performance for exascale applications through online data layout reorganization. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2021), 878–890.

[77] Jeremy J Williams, Ashish Bhole, Dylan Kierans, Matthias Hoelzl, Ihor Holod, Weikang Tang, David Tskhakaya, Stefan Costea, Leon Kos, Ales Podolnik, et al. 2024. Understanding large-scale plasma simulation challenges for fusion energy on supercomputers. *arXiv preprint arXiv:2407.00394* (2024).

[78] Kesheng Wu. 2005. FastBit: an efficient indexing technology for accelerating data-intensive science. In *Journal of Physics: Conference Series*, Vol. 16. IOP Publishing, 556.

[79] Tzuhsien Wu, Jerry Chou, Shyng Hao, Bin Dong, Scott Klasky, and Kesheng Wu. 2017. Optimizing the Query Performance of Block Index Through Data Analysis and I/O Modeling. In *SC17: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–10.

[80] Qiaolin Yu, Chang Guo, Jay Zhuang, Viraj Thakkar, Jianguo Wang, and Zhichao Cao. 2024. CaaS-LSM: compaction-as-a-service for LSM-based key-value stores in storage disaggregated infrastructure. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–28.

[81] Kedeng Zhang and Hui Wang. 2023. Observations and simulations of large-scale traveling ionospheric disturbances during the January 14-15, 2022 geomagnetic storm. *Frontiers in Astronomy and Space Sciences* 10 (2023), 1297632.

[82] Wen-shuai Zhang, Hong-bo Cai, Bao Du, Dong-guo Kang, Shi-yang Zou, and Shao-ping Zhu. 2021. Full particle-in-cell simulation of the formation and structure of a collisional plasma shock wave. *Physical Review E* 103, 2 (2021), 023213.

[83] Tiezhu Zhao, Verdi March, Shoubin Dong, and Simon See. 2010. Evaluation of a performance model of lustre file system. In *2010 Fifth Annual ChinaGrid Conference*. IEEE, 191–196.

[84] Qing Zheng, George Amvrosiadis, Saurabh Kadekodi, Garth A. Gibson, Charles D. Cranor, Bradley W. Settlemyer, Gary Grider, and Fan Guo. 2017. Software-defined storage for fast trajectory queries using a deltaFS indexed massive directory. In *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems* (Denver, Colorado) *(PDSW-DISCS '17)*. Association for Computing Machinery, New York, NY, USA, 7–12. doi:10.1145/3149393.3149398

[85] Qing Zheng, Charles D. Cranor, Danhao Guo, Gregory R. Ganger, George Amvrosiadis, Garth A. Gibson, Bradley W. Settlemyer, Gary Grider, and Fan Guo. 2018. Scaling Embedded In-Situ Indexing with DeltaFS. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 30–44. doi:10.1109/SC.2018.00006

Received 7 February 2025; accepted 4 April 2025.