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

Chang Guo[1], Ning Yan[2], Lipeng Wan[2], Zhichao Cao[1]

Intelligent Data Infrastructure Lab (***ASU-IDI***)

[1] Arizona State University

[2] Georgia State University

Background

- What is PIC Data?
  - Particle-In-Cell (PIC) is a widely used simulation method in plasma physics and other scientific domains.
- Scale of PIC Simulations
  - PIC simulations generate **TB to PB data per hour.**
- Popular Simulation Frameworks:
  - **WarpX**, **EPOCH**, and **Geant4**.
- Common Analysis Tools:
  - **openPMD-viewer**, **ParaView**, and **H5py**

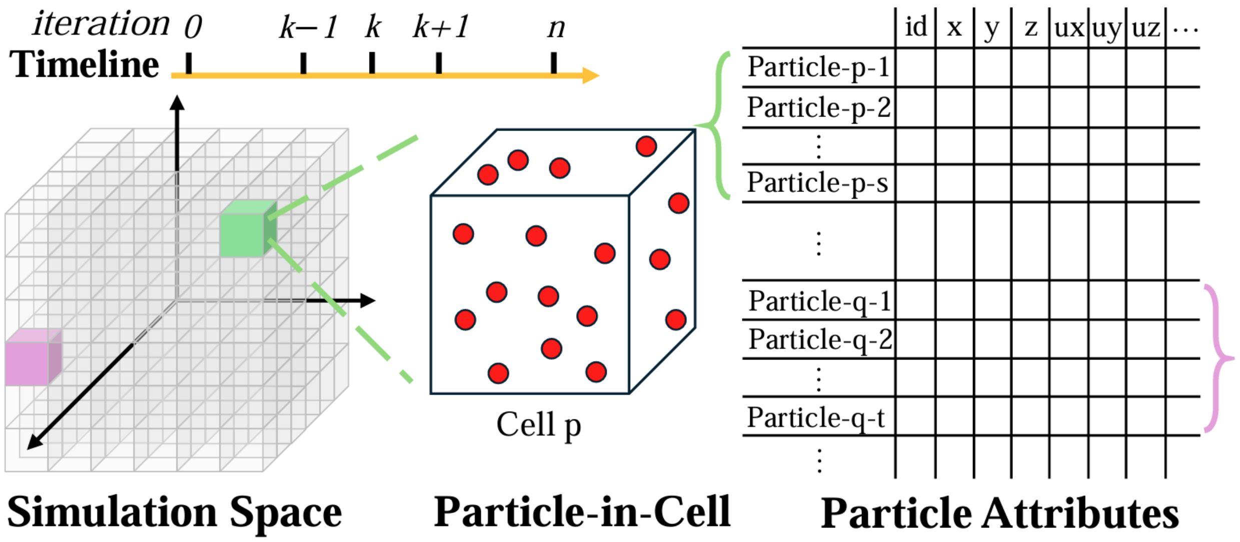


Figure 1: Particle data organization in PIC simulations.

Background

- How is PIC data stored on HPC clusters?
  - PIC data is typically stored in **column-based format** to optimize output performance.

- However, this leads to inefficiencies in analysis:
  - **The entire cell must be scanned** even targeting a few particles.
  - **Filtering by one attribute** and retrieving another results in **scattered reads and high I/O overhead**.

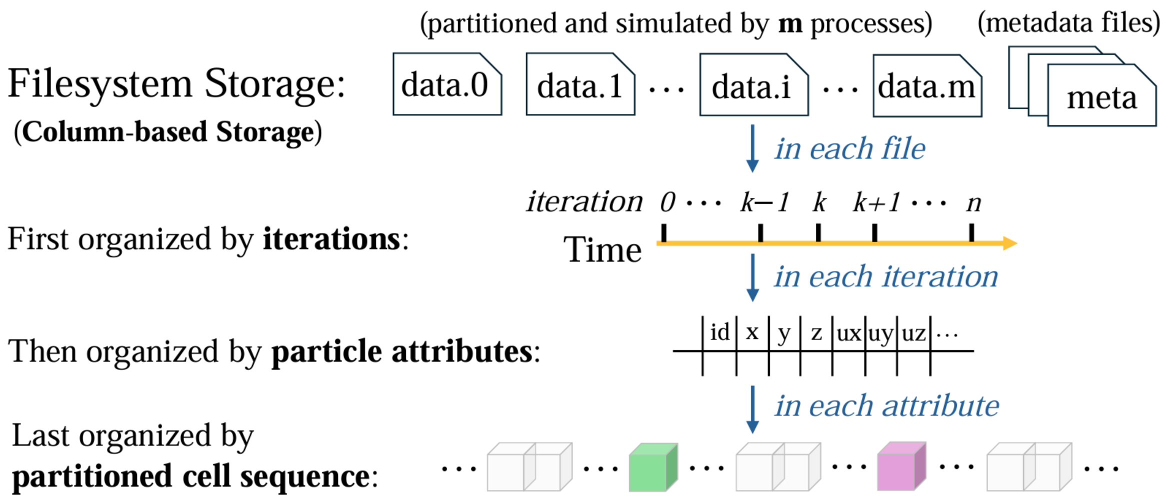
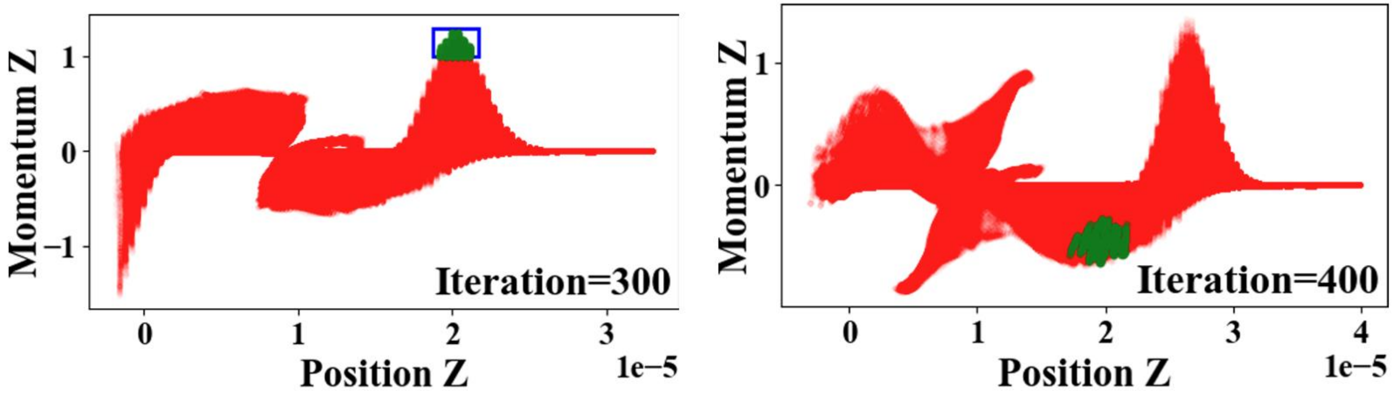


Figure 2: Column-based storage of particle data on filesystem.

Background

- Workflows of Particle Data Analysis
  - **Overview Visualization** — observe the global distribution of particles
  - **Particle Selection** — perform range queries based on particle attributes
  - **Particle Tracking** — follow selected particles across iterations



(a) Particle Overview and Selection

(b) Particle Tracking

Figure 3: Particle distribution, selection, and tracking.

Motivation

- Existing analysis tools load the entire dataset into memory, leading to:

1. **High Particle Query Latency on Large Datasets.**
    - A single query on a 1 TB dataset can **take over an hour**
    - Re-reading the full dataset for each query is **redundant and inefficient**

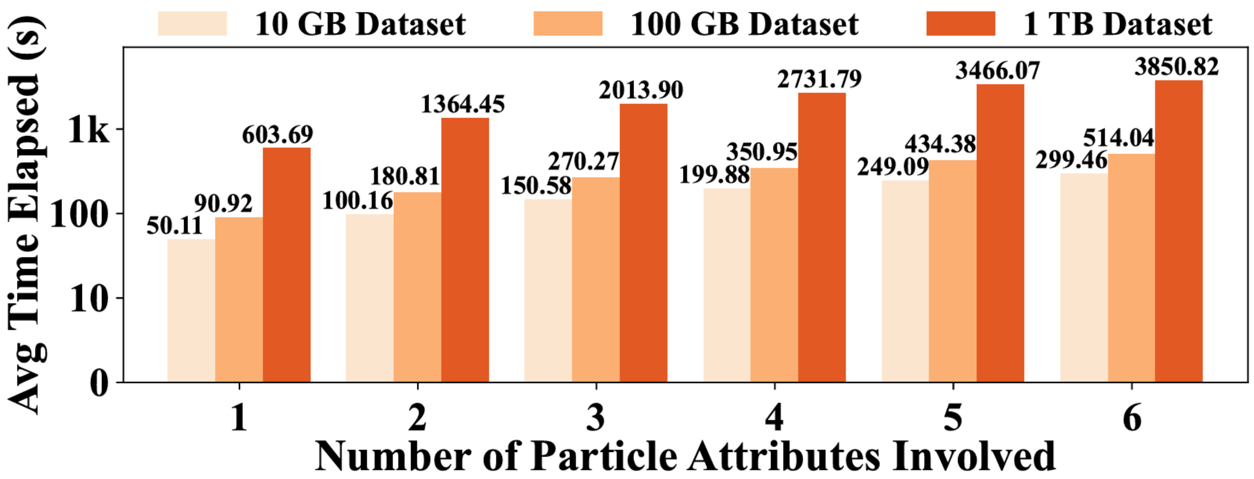


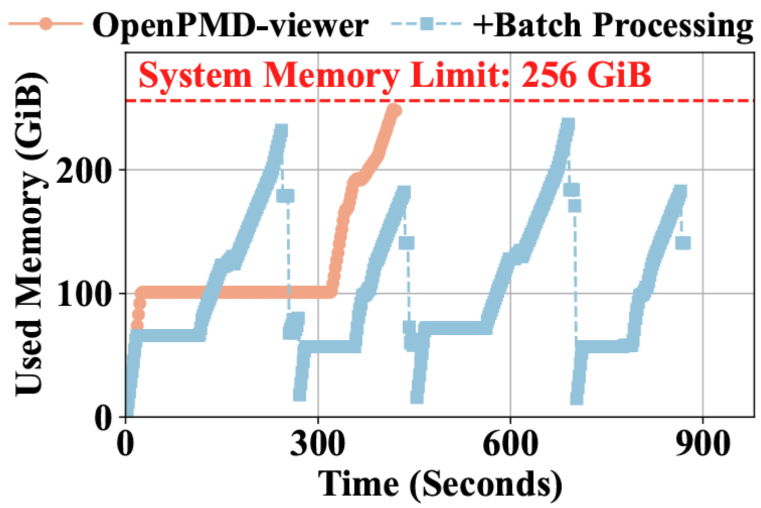
Figure 4: Average query time across different dataset scales.

Motivation

- Existing analysis tools load the entire dataset into memory, leading to:

2. Large Memory Footprint.

  - Loading large-scale particle data is **infeasible** due to memory limits. (red line)
  - Batch loading with partial result merging (blue curve) reduces memory usage but still **scans the entire dataset.**

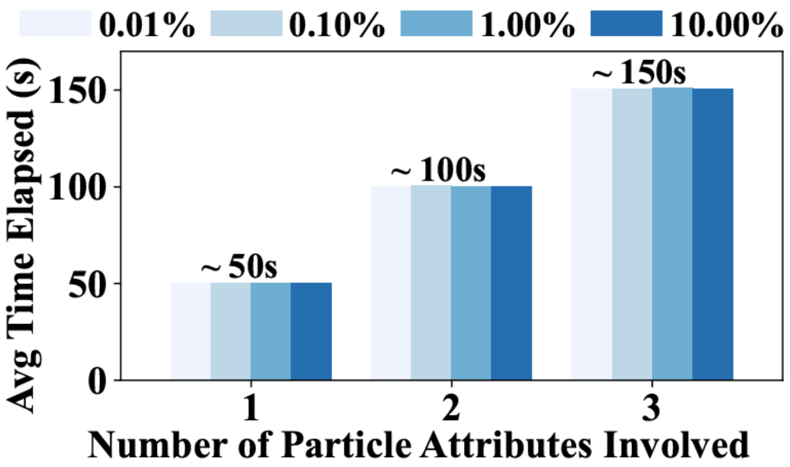


(a) Memory footprint of existing analysis tools

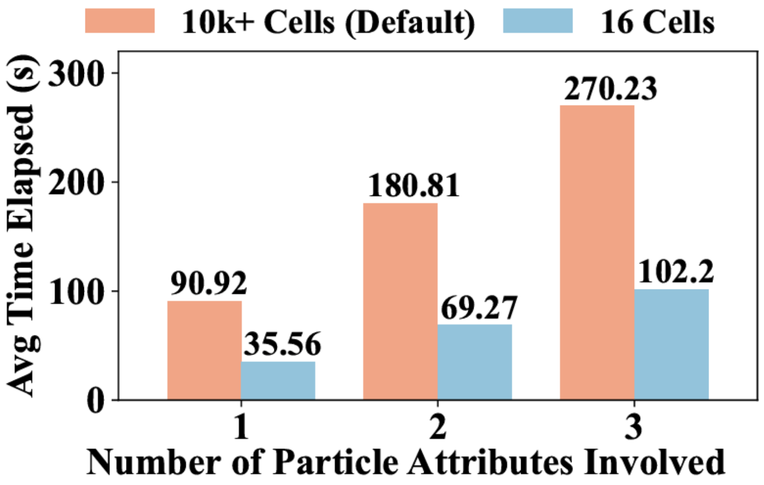
Motivation

- Existing analysis tools load the entire dataset into memory, leading to:

3. **I/O Inefficiency.**
   - **Unnecessary Reads**: Query latency remains **constant** even with varying selection proportions (left).
   - **Small I/O:** Reorganizing 10k+ cells into 16 larger cells significantly reduces query time—**up to 3× faster** (right).



(a) Different Selection Proportions



(b) Impact of I/O Block Size

Insights

- Using indexes can help filter and selectively read target data efficiently.

- However, existing indexing mechanisms for PIC simulations face challenges:

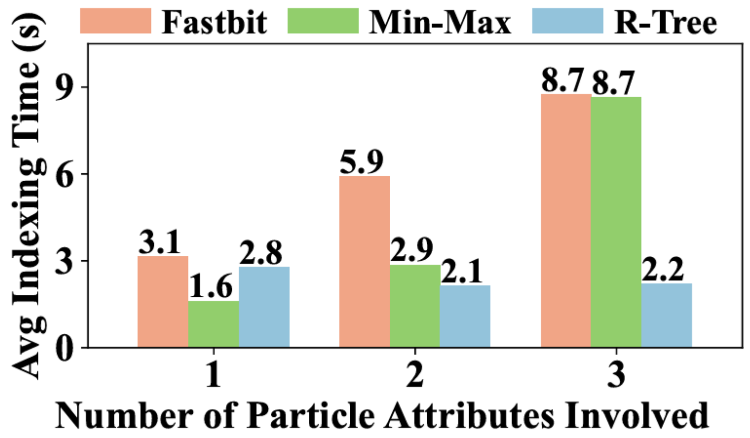
  1. **Single-purpose indexes** perform well for specific tasks but lack flexibility for diverse query patterns.
  2. **Online indexing** adds 10–15% overhead to simulations, while **post-simulation indexing** requires reading the entire dataset again—consuming extra resources.
  3. **Indexed results are often scattered**, leading to **small, fragmented I/O**, which reduces efficiency.

Research Objective

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.

Challenges

1. Capability of Adapting to Various Analysis Tasks.
    - **Single-purpose indexes** perform well for specific tasks but lack flexibility for diverse query patterns.
2. Efficient Index Construction, Storage, and Migration.
    - **Online indexing** adds 10–15% overhead to simulations, while **post-simulation indexing** requires reading the entire dataset again—consuming extra resources.
3. Query Optimizations with Intelligent I/O Operation Planning and Scheduling.
    - **Indexed results are often scattered**, leading to **small, fragmented I/O**, which reduces efficiency.



(b) Different Index Performance

LegoIndex Design

① Modular for Various Analysis Tasks

② Efficient Index Construction, Storage, and Migration

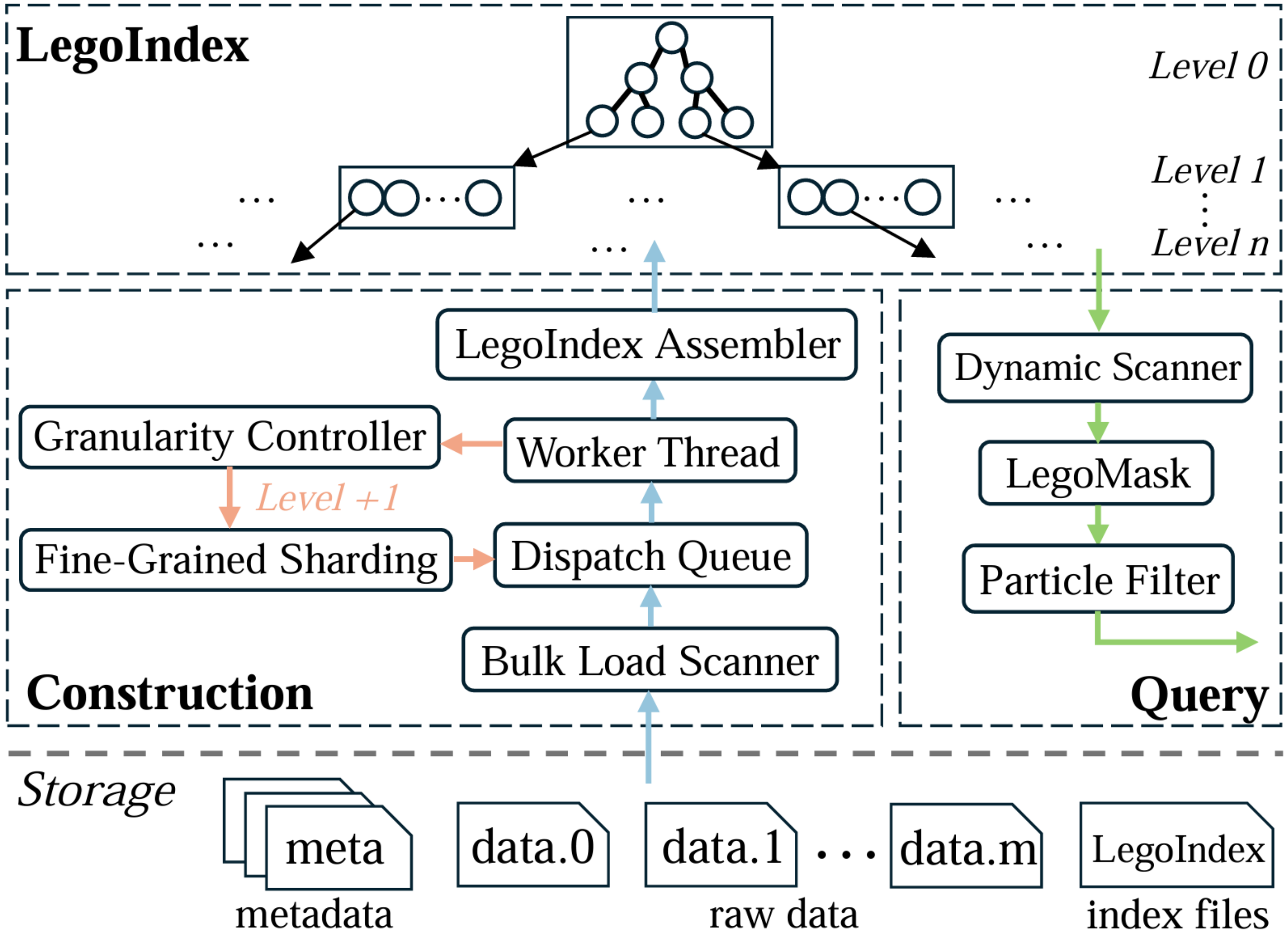


Figure 7: LegoIndex: structure and workflow overview.

③ Query Optimizations

LegoIndex Design

1. A Modular Indexing Framework

Various cell statistics can help analysis

However, indexing all possible statistics leads to:

- Longer construction time
- Increased storage and migration overhead
- Higher query load time

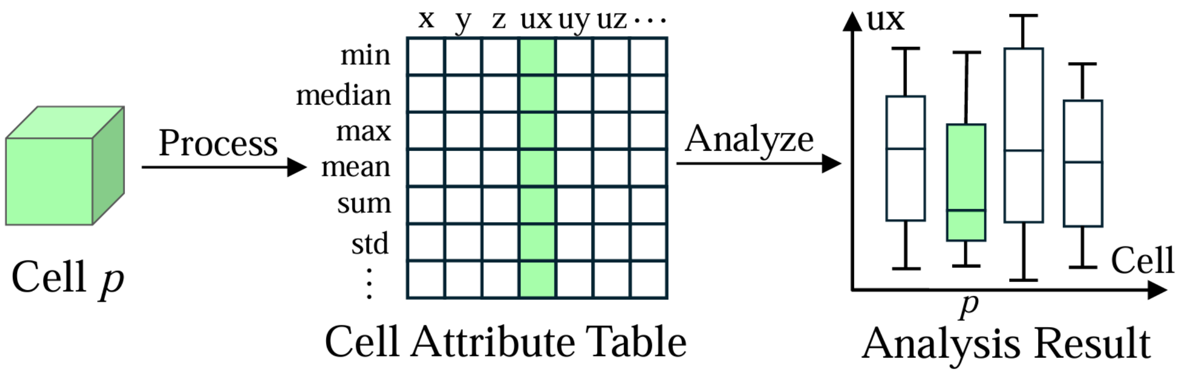


Figure 9: Various cell statistics help analysis.

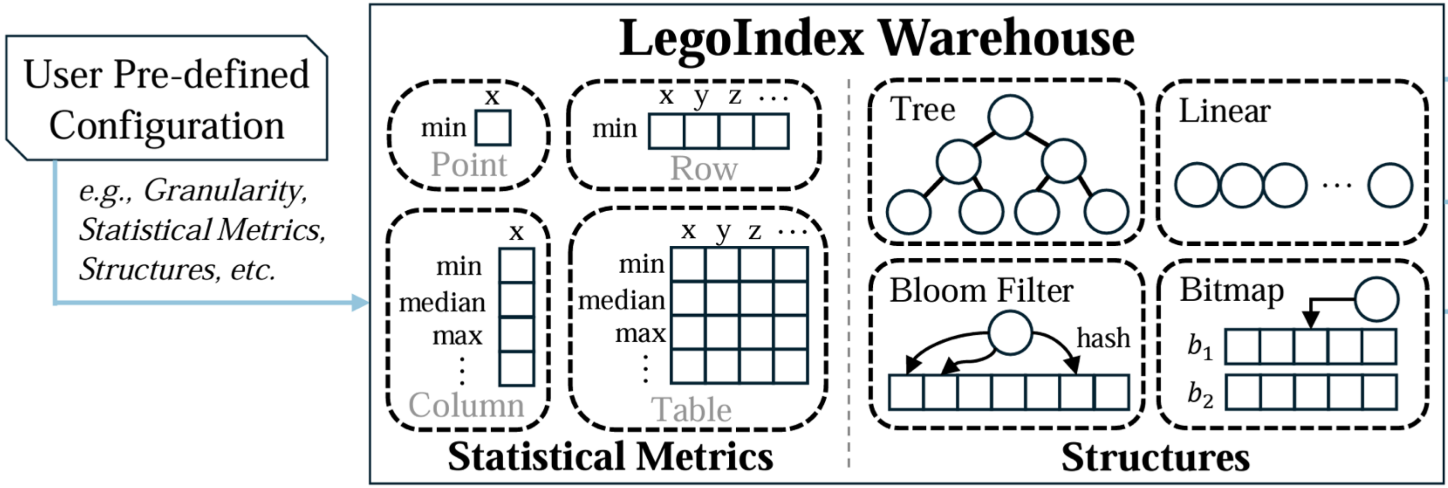
LegoIndex Design

1. A Modular Indexing Framework

LegoIndex provide an index warehouse with pre-defined Statistics Metrics and Structures

It allows users to customize:

- Indexing granularity (e.g., max-level-num, granularity conditions, etc.)
- Statistics metrics for each level
- Index structure for each level



LegoIndex Design

1. A Modular Indexing Framework

By default, **LegoIndex** constructs only the top-level cells using a tree-based index.

- Users can customize configurations as needed.
- In future, utilizing predictive heuristics for automatic adaptive indexing.

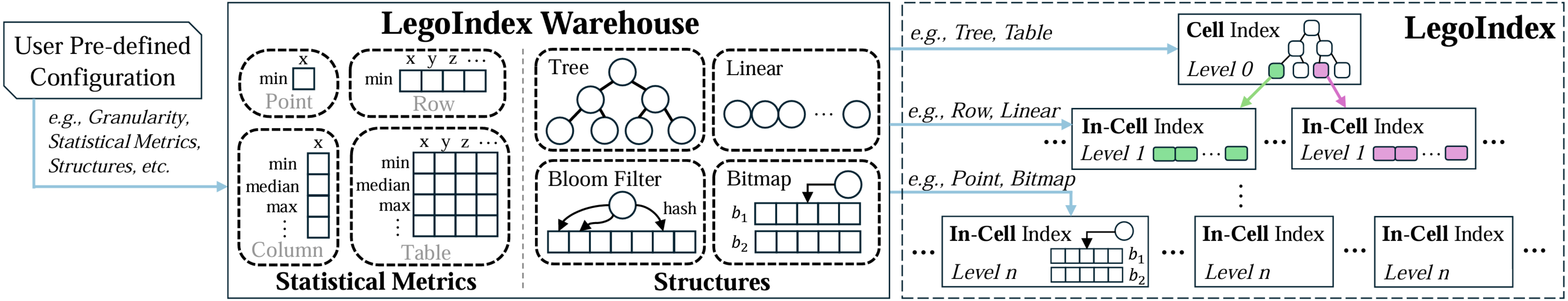


Figure 8: Architecture and design overview of LegoIndex.

LegoIndex Design

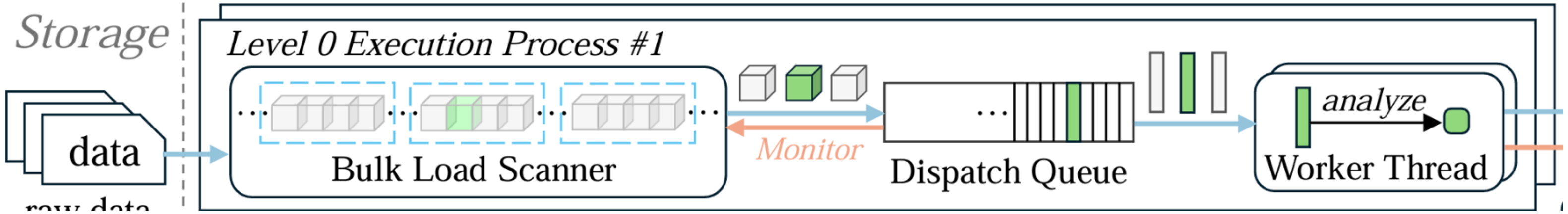
2. Efficient Index Construction, Storage, and Migration

Loading the entire dataset is infeasible for large-scale data,

while loading data cell-by-cell incurs inefficient small I/Os.

LegoIndex introduces a **Bulk Load Scanner** thread to

- Loads data in large chunks
- Dispatches the data to lower-level workers for processing



LegoIndex Design

2. Efficient Index Construction, Storage, and Migration

LegoIndex introduces

- **Granularity Controller**: Manages construction of the next-level index based on predefined rules
- **Assembler**: Integrates results from workers and builds the index
- **Key-Value Mechanism**: Links multiple index levels and simplifies storage and retrieval

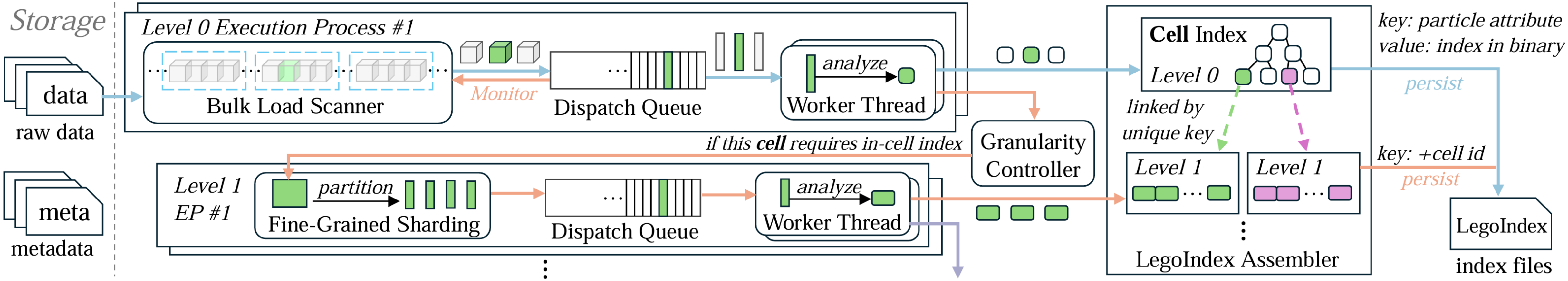


Figure 10: LegoIndex construction workflow.

LegoIndex Design

3. Query Optimizations with LegoIndex

Index results are scattered across the dataset.

- Directly fetching them leads to inefficient small I/Os.

LegoIndex introduces

- **Dynamic Scanner**: Groups nearby cells for efficient bulk reads or splits large cells into multiple I/Os
  - Adjusts fetching strategies based on historical performance
- **LegoMask**: Filters out unrelated in-memory data to reduce processing overhead

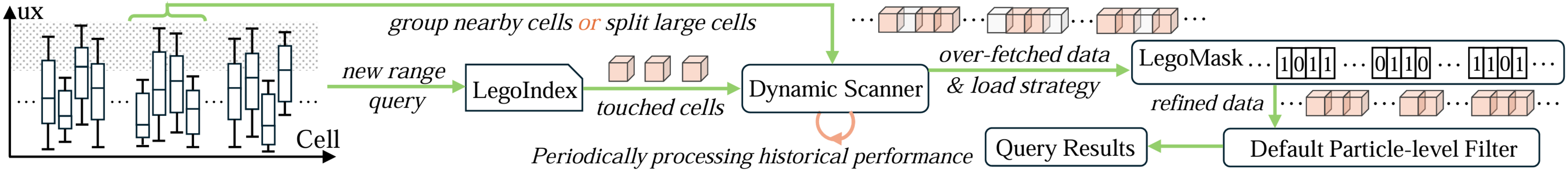


Figure 11: LegoIndex intelligent I/O scheduling workflow.

Evaluation Setup

Dataset: Generated using WarpX on the Perlmutter supercomputer at LBNL.

Dataset Sizes: 10GB, 100GB, and 1TB per iteration (~10k cells for all datasets)

Analysis Application: *openPMD-viewer*

Query Generator: Produces queries that select N% of the dataset based on attribute (e.g., momentum x and y).

Baseline:

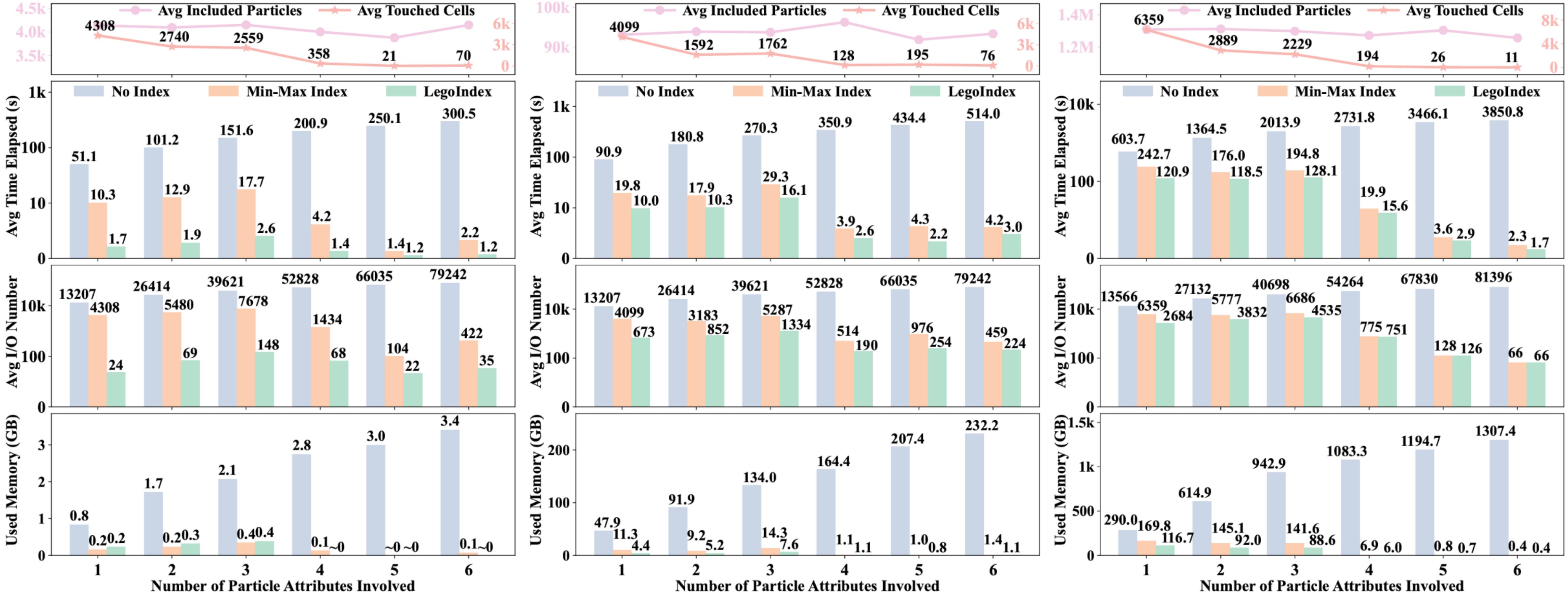
- **No Index**: default *openPMD-viewer* without indexing
- **Min-Max Index**: *openPMD-viewer* with Min-Max indexing support

Metrics:

- Query execution time
- Memory usage
- Number of I/O operations

Evaluation

Overall Query Performance (in logarithmic scale)



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

Evaluation

Query Performance at different selection proportions (10GB Dataset)

- Left: Increase in included particles and touched cells with higher selection rates
- Middle-left: LegoIndex and Min-Max significantly reduce query latency compared to no index
- Middle-right: Min-Max I/O count along with the select proportion, LegoIndex reduce I/O by dynamic scanner
- Right: LegoIndex achieves similar memory usage with better performance

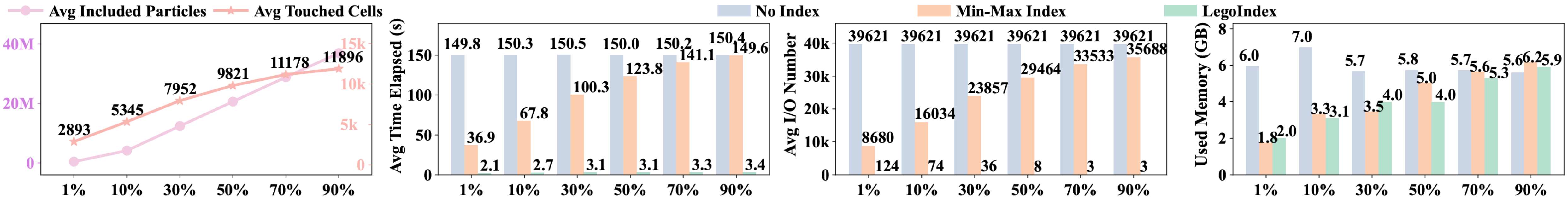


Figure 13: Query performance at different selection proportions (x-axis: selection proportions).

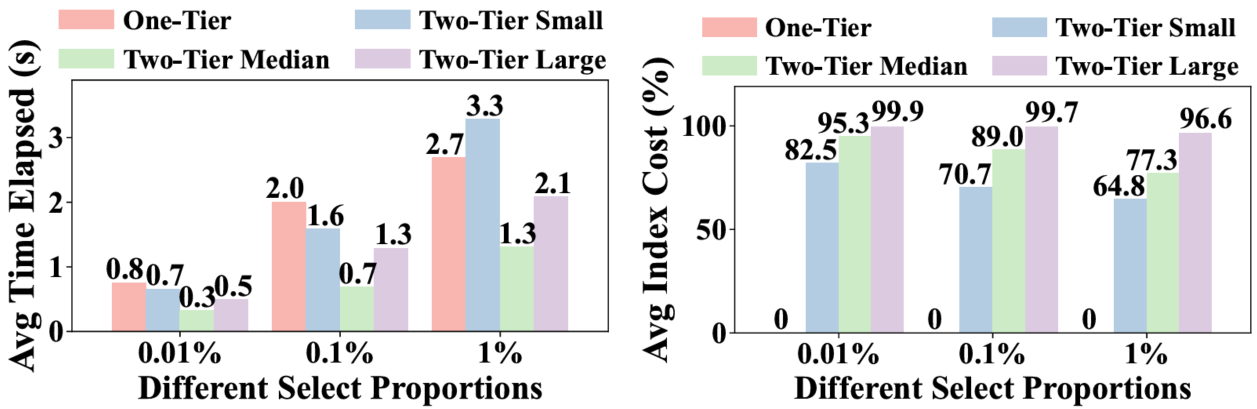
Evaluation

Performance of Different LegoIndex Configurations

- Comparison: **Default (one-tier)** vs. **Fine-tuned (two-tier)**

(a) Query Average Latency (In-Memory)

(b) Index Cost Proportion (In-Memory)



(a) Query Average Cost (In-Memory).

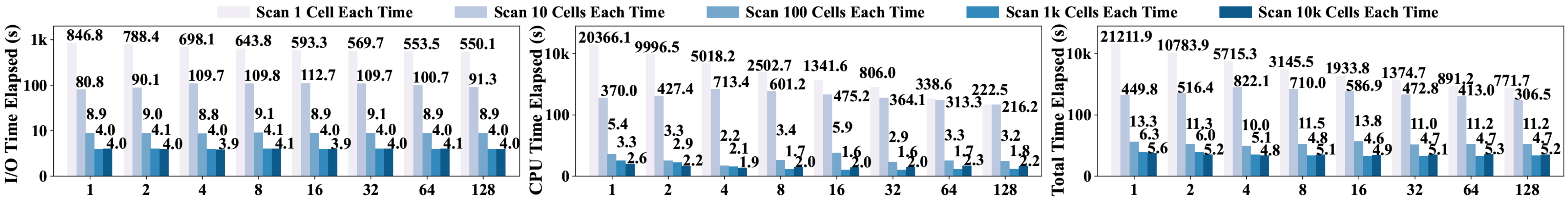
(b) LegoIndex Cost Proportion (In-Memory)

Figure 14: Query performance across LegoIndex granularity.

Evaluation

Index Construction Performance

1. I/O Time Elapsed: Larger scan sizes significantly reduce I/O time across all thread counts. However, the benefits diminish as the batch size grows further.
2. CPU Time Elapsed: Increasing the number of threads reduces in-memory processing time, but the benefits diminish as thread count grows
3. Total Time Elapsed: for the 10GB dataset, scanning100 to1,000 cells per batch (i.e., 100MB to 1GB) with 4 to 8 worker threads achieves the highest efficiency.



(a) I/O Time Elapsed (b) CPU Time Elapsed (c) Total Time Elapsed

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

Evaluation

Performance Improvement by I/O optimization

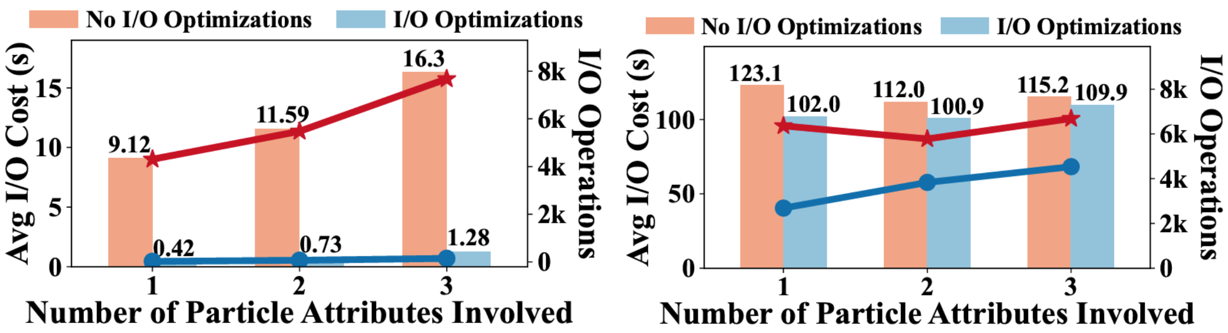
- **10GB Dataset – Small Cells:**

  LegoIndex achieves up to 21.7× speedup by reducing small I/O overhead with its Dynamic Scanner.

- **1TB Dataset – Large Cells:**

  Though the benefit of grouping diminishes, LegoIndex still provides 10–20% improvement by efficiently managing I/O.

Dynamic Scanner adapts to dataset scale, ensuring consistent performance gains.



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

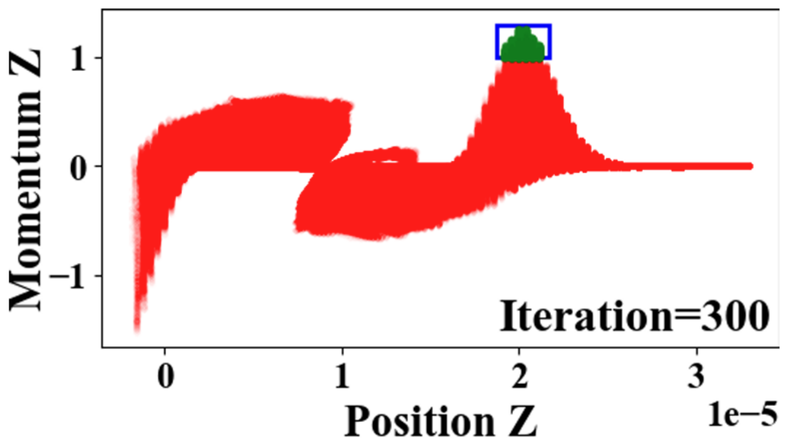
Figure 16: LegoIndex intelligent I/O scheduling.

Evaluation

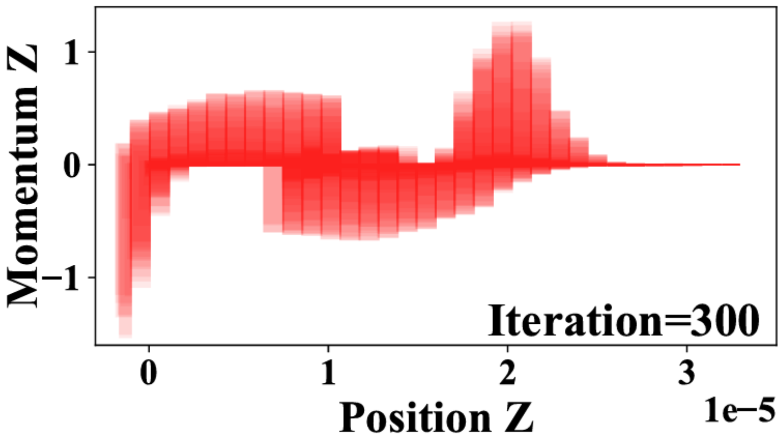
Other use cases – Approximate Visualization

On a 10GB dataset at iteration 300:

- **No Index** (left) plots all 40M particles in 23.1s
- **LegoIndex** (right) visualizes using aggregated cell metadata in just 7.3s (3× faster)



(a) Particle Overview and Selection



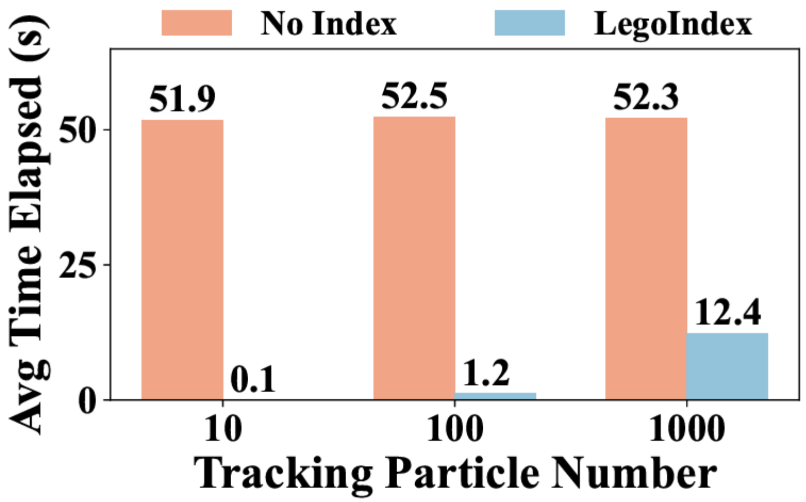
(a) LegoIndex Approximate Visualization

Evaluation

Other use cases – Particle Tracking

Figure 17(b): Tracking particles (10, 100, 1000) from a 10GB dataset

- **No Index:** Always scans all IDs → stable but inefficient
- **LegoIndex:** Uses tree + Bloom filters for fast localization
- Up to 260× speedup when tracking 10 particles
- Performance scales linearly with number of tracked particles
- Best suited for selective tracking in scientific analysis



(b) Particle Tracking across Iterations

Conclusion and Future Work

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

- Scalable and Modular Indexing Framework
- Accelerates post-simulation index construction
- Enhances query performance with **Dynamic I/O Scanner** and **LegoMask**
- Supports particle **visualization** and **tracking** workflows

Next Steps:

- Broaden support beyond PIC to other scientific and simulation data types
- Add predictive heuristics and locality-aware strategies for automatic adaptive indexing
- Enable cluster-level parallel index construction and distributed querying

Thank You!

Q & A

ASU-IDI

Contact

Chang Guo (cguo51@asu.edu)

Zhichao Cao (Zhichao.Cao@asu.edu)

https://asu-idi.github.io/contact/