CARP: Range Query-Optimized Indexing for Streaming Data

Ankush Jain[†], Charles D. Cranor[†], Qing Zheng[‡], Bradley W. Settlemyer[§], George Amvrosiadis[†], Gary A. Grider[‡]
[†]Carnegie Mellon University, [‡]Los Alamos National Laboratory, [§]Nvidia
{ankushj, chuck, gamvrosi}@cmu.edu, {qzheng, ggrider}@lanl.gov, bsettlemyer@nvidia.com

***Abstract*—Ingestion of data generated by high-performance scientific applications continues to stress available storage resources. Efficient range-based analyses on this data can be enabled by reordering it on attributes of interest, but require expensive post-processing sorts to realize the query benefits of reordering. In-situ indexing techniques, while write-efficient, are orders of magnitude slower at range queries than sorted indices. Range queries are necessary for analyzing continuous physical attributes and tracking phenomena such as energy bands and wave fronts.**

We present CARP, a scalable data partitioner for range queries that reorders data in-situ as it is streamed to storage during application I/O. Motivated by our findings that real application distributions tend to be highly skewed and dynamic, CARP dynamically discovers and adapts its data partitions to track these characteristics. As a result, CARP can approximate the query performance of a sort without any ingestion overhead, making it 5× faster than prior work.

***Index Terms*—Data Analysis, Sorting, Overlay Networks, In-situ Indexing**

I. Introduction

Organization of data generated by high-performance applications continues to be a challenge due to storage bandwidth constraints. Many high-performance applications consist of distinct computation phases, often called *epochs* or *timesteps*, that are periodically interrupted by checkpointing I/O phases. Examples of such applications include particle-in-cell frameworks [1], mesh-based codes [2–4], and machine learning model training. Periodic I/O phases enable scientists to perform queries with a temporal component, such as tracing trajectories of high-energy particles, or tracking shock waves in a fluid simulation. For efficient execution of such queries, persisted data is typically indexed in a post-processing stage. This post-processing requires additional time and resources, increasing the time to discovery. It also increases with the scale of the simulation, especially as computational bandwidth growth outpaces that of storage bandwidth [5].

To reduce the time to discovery, in-situ approaches that index data as it is written by the parallel application have been proposed [6] as an alternative to post-processing. DeltaFS [7] intercepts application writes and organizes them into hash-based partitions. This enables point queries that retrieve individual items by key, but it breaks key locality. Range queries require key locality to efficiently find all items with keys in a given range. Range queries are an important tool for scientists interacting with continuous attributes such as temperature and velocity. They can be used to filter data into ranges of relevant values, such as isolating blast wave fronts or tracking energy bands. However, existing solutions for range query indexing force undesirable tradeoffs between write performance, query performance, and resource utilization (§II).

To enable efficient range queries without post-processing, it is necessary to develop reordering-based in-situ indexing mechanisms. Reordering creates key locality in the data layout, which enables range queries to be served efficiently via a small number of contiguous writes. However, reordering of on-disk data stresses write performance by requiring I/Os for index maintenance. This leads to slower checkpointing and idling of compute nodes. Enabling efficient querying without adversely affecting write performance requires reordering data in the network data plane, before writing to storage. Doing so without requiring additional compute resources requires a lightweight embedded reorganization mechanism with a low memory footprint. These challenges are further compounded by the nature of scientific key distributions — our analysis of real-world codes (§III) shows that scientific key distributions are highly skewed and vary over time. Reorganization mechanisms processing such data need to be adaptive to ensure that load hotspots in the reordering pipeline do not slow down application writes.

We present CARP (*Continuous, Adaptive, Range Partitioner*), an adaptive in-situ indexing system that enables range queries without requiring data post-processing. CARP requires no changes to scientific applications and operates by intercepting writes in flight to storage. User input on key distribution characteristics is not required. Partitions are instead discovered at runtime and adapted as the key distribution changes. CARP is write-optimized and does not divert any storage bandwidth from the application for reordering. Our evaluation shows that CARP incurs no overhead on application runtime and can achieve sub-second range query latency that is up to two orders of magnitude faster than state-of-the-art auxiliary indexing approaches. By partitioning data at runtime, CARP is up to 5× faster than post-processing approaches.

The contributions of this work are summarized as follows:

- After a brief discussion on background (§II), we characterize workload distributions and their drifts in modern scientific applications (§III) and demonstrate that static partitioning is inadequate for such workloads (§VII-B).

SC24, November 17-22, 2024, Atlanta, Georgia, USA
979-8-3503-5291-7/24/$31.00 ©2024 IEEE

TABLE I: Range query indexing approaches.

Approach	In-situ	Efficient Indexing	Efficient Querying
Bulk Sorting (Clustered)	×	×	✓
Bulk Sorting (Auxiliary)	×	×	×
Bitmap Indexes (FastQuery)	×	×	×
DB Indexes (LSM-Tree, B-Tree)	✓	×	✓
DeltaFS	✓	✓	×
CARP (This paper)	✓	✓	✓

- We present CARP (§V), an in-situ data processing framework guaranteeing data locality and load balancing at scale. We identify requirements for a CARP storage backend and present KoiDB (§V-D), a reference backend implementation.
- We show that partial locality in data layout can be created in a single pass without requiring any reorganization and provide nearly-optimal range query performance (§VII-A), followed by a discussion on adapting CARP for different requirements (§VIII).

We have released the source code [8] for both the CARP framework and the KoiDB backend.

II. Background and Motivation

Data generated by many scientific applications consists of simulated entities such as particles and mesh cells, each storing several attributes (e.g. energy, velocity) into a *record*. This data is written out at fixed intervals and not mutated once written, making bulk indexing feasible. The data is then analyzed using point queries (e.g., retrieving particles by ID) or range queries (e.g., retrieving particles with energy in a certain range). Range queries are needed for interacting with continuous attributes and for filtering data using ranges of relevant values (e.g. energy bands, temperature and blast wave fronts).

An efficient range-query index must locate ranges of values quickly. Sorted indexes do this by creating indexed attribute (henceforth referred to as *key*) locality in the data layout, so that the keys for a range can be located in one or few data partitions. Clustered indexes further improve query performance by storing all records with their corresponding keys. This allows for data for a query to be retrieved efficiently via fewer, contiguous reads. In contrast, auxiliary indexes need multiple random reads to retrieve all attributes of interest in the range. In general, a database table can have only one clustered index. The clustered index is typically reserved for the attribute most commonly used in queries [9, 10].

Table I shows that no current approach enables both in-situ efficient indexing and efficient querying for ranges. We summarize the characteristics of the indexing approaches below. We discuss their *index construction* performance and break down their query performance into *index lookup* and *data retrieval*.

Index Construction. Bulk parallel/distributed sorting [11–13] approaches support range queries but require post-processing. These can be used to build either clustered or auxiliary sorted indexes. FastQuery [14], on the other hand, builds auxiliary bitmap structures where keys are sorted and stored separately with a pointer to their records. Clustered indexes enable more efficient queries but require more I/O to reorder and write the entire dataset. FastQuery does not reorder data, but results in data scans, computing large bitmap vectors, and writing auxiliary structures to disk.

To avoid expensive post-processing, techniques have been proposed to intercept application writes and index them in-situ before data is written to disk. DeltaFS [7] shuffles data into hash-based partitions and requires no post-processing, but it can only support point queries on the partitioned data. Distributed databases maintain online indexes but offer lower write efficiency as they periodically reorganize on-disk data [15]. Databases are designed for mixed workloads with updates, while lightweight indexes are better suited to scientific data.

Index Lookup and Data Retrieval. Efficient range queries require quickly finding matching records using the index and retrieving the records from storage using large sequential reads. Only sorted and clustered indexes provide both of these properties. Auxiliary indexes can return the offsets at which matching rows are present, but retrieving those records requires multiple random reads. Additionally, FastQuery queries require provisioning enough compute nodes to fit large index structures in memory, and queries can only be processed once these indexes are loaded.

With CARP, we aim to pair the query latency of a sorted and clustered index with the reduced overhead of an in-situ approach. We discuss the challenges of meeting these goals in the next section.

III. Range Query Challenges

Sorted, clustered indexes provide the lowest query latency but require post-processing to reorder data. Sorting large on-disk datasets is even more expensive than building auxiliary indices, as it requires multiple read/write passes over the data [13].

As an alternative to reordering via post-processing, data can be partitioned into *range-based partitions* while in transit from the application to storage. This accelerate queries, as only a small number of relevant partitions need to be retrieved from storage for each query. While range partitioning is a well-known technique in distributed databases, adapting it to in-situ partitioning of scientific data requires overcoming two key challenges we describe next.

Write Rate. To organize data in a way that does not reduce the application write throughput, it is not sufficient to partition data online. The indexing pipeline must be designed to not require any storage bandwidth for reordering or index maintenance. Conventional databases use reordering to maintain their partitions, which reduces effective storage bandwidth [16, 17]. As on-disk partitions grow beyond configured sizes, they are split into smaller partitions and migrated across nodes.

The maximum achievable write rate is a function of the average number of I/O operations performed for each application write, also called the *Write Amplification Factor* (WAF).

A WAF of 10 will turn an I/O phase lasting 12 minutes into 2 hours, which is wasteful for write-intensive large-scale scientific applications. WAF for write-optimized single-node database indexes has been measured at 19-37 [15] and is much higher for B-tree indexes and distributed databases [18]. In-situ strategies that have a high WAF would not outperform post-processing approaches (WAF of 2-3×). To maximize gains from an in-situ indexing approach, we constrain our design to a WAF of 1×.

Adaptivity. Effective range-partitioning requires the key space to be partitioned such that the generated data partitions are relatively balanced. This is important for both write and query performance. On the write path, this ensures that the application is not slowed down by a straggler writing a significantly larger partition than others. On the query path, imbalanced partitions will lead to much slower performance for queries corresponding to those partitions.

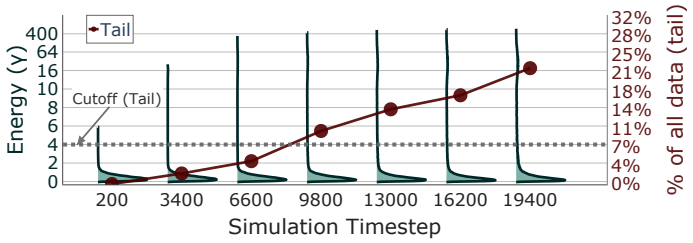
However, we find that keys in scientific data are often skewed in unpredictable ways and change as the simulation progresses. Specifically, we studied the energy distributions of two applications — VPIC and Phoebus. VPIC is a plasma physics code that simulates particle physics phenomena such as magnetic reconnection [1]. Phoebus is a mesh-based hydrodynamics code that we run with a Sedov blast wave setup [4]. We ran both codes at 512 ranks and studied the energy distributions in the output data.

Fig. 1 shows the energy distributions of both the codes over time. For VPIC (Fig. 1a), we find that the energy distributions are highly skewed with most particles falling between 0 and 1 with long tails that get longer and heavier as the simulation progresses. Towards the second half of the simulation, 20-30% of the data is contained within the tail generating a bimodal distribution with a second mode between 16 and 64. For Phoebus (Fig. 1b), we also see a highly skewed distribution with a tail. Initially, there is a high energy explosion but most of the mesh has no energy. Over time, the energy from the explosion dissipates into a bigger area and moves the distribution into a medium energy band.

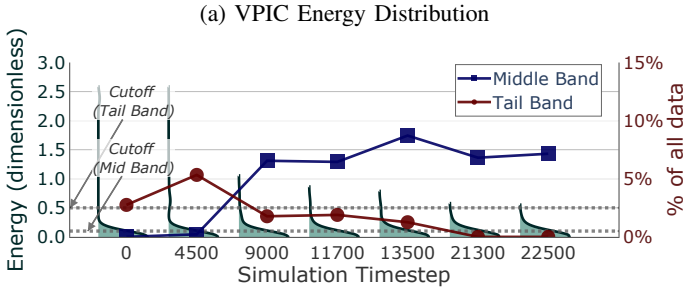
These findings highlight the challenges with partitioning keys from scientific workloads. A highly precise reconstruction of the key distribution is necessary to divide the keyspace into a large number of balanced partitions. It is also impractical for any user-provided static range partitioning scheme to accommodate such skewed and dynamic key distributions. A distributed database would only be able to balance its partitions after a series of splitting, rewriting, and merging operations, incurring prohibitive index construction costs. An ideal range query index would discover and adapt to workload characteristics and reorder data without consuming excess storage bandwidth.

IV. Design Principles of CARP

We have designed a streaming in-situ partitioner for large parallel applications, CARP, which intercepts writes from applications, transforms them, and persists them in a range query-friendly layout. A logical view of CARP's partitioning



(a) VPIC Energy Distribution



(b) Adaptive Mesh Refinement Energy Distribution

Fig. 1: Energy distributions across 7 timesteps of VPIC (top) and AMR (bottom) simulations. Distributions are shows as histograms overlaid with line plots for interesting bands. Both VPIC and AMR distributions are highly skewed and shift significantly over time.

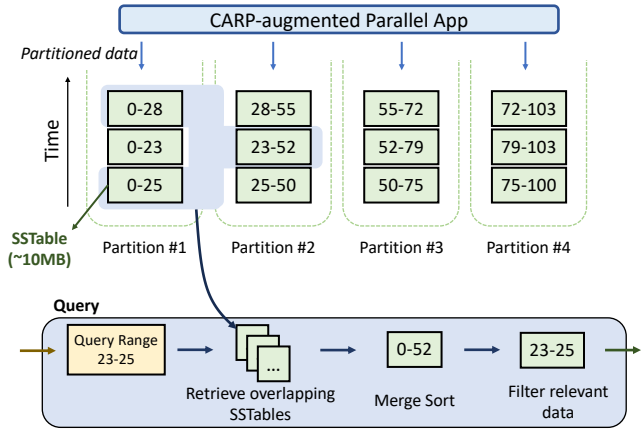


Fig. 2: A logical view of CARP's data layout and querying. Incoming data is partitioned and stored as SSTables, and partition boundaries shift with key distribution changes. Queries retrieve relevant SSTables and merge them.

process is shown in Fig. 2. CARP has no impact on application runtime, as it uses spare CPU and network capacity to transform data at a rate necessary to saturate storage. This partitioned output requires no post-processing and can be queried directly once the application terminates.

CARP augments conventional range partitioning with novel techniques to achieve its write and query performance goals. We build upon a simple insight — data does not need to be fully sorted to achieve the performance benefits of a sorted, clustered index. It is sufficient to partially order data in a way that allows queries to be processed via a small number of large reads. CARP exploits this relaxation to achieve its write performance and adaptivity goals.

We employ a greedy approach to generate best-effort range

partitions in a single streaming pass so as to not require expensive reorganization of written data. To generate balanced partitions, CARP employs a novel primitive to construct and monitor the global key distribution. This distribution is divided into partitions with equal areas under the curve, and each partition is assigned to a rank. In the common case, all ranks shuffle data to corresponding partitions. As the workload's key distribution drifts, CARP recomputes partition boundaries and continues shuffling. Shuffled data is written to disk via a storage backend called *KoiDB* — KoiDB further optimizes CARP output to improve range query performance. Next, we describe how CARP achieves the design goals laid out in the previous section.

Write Performance. To ingest data at storage layer throughput, CARP is designed to not divert any storage bandwidth for reorganization. Even when partitions need to be recomputed, CARP does not touch previously written data — it simply records a change in partition boundaries on each rank and continues appending data. CARP also employs standard storage optimization practices such as batching small writes into large immutable units called *SSTables* and having its on-disk layout be an append-only log.

Query Performance. Efficient queries in CARP are enabled by 1) creating a highly partitioned layout, and 2) enabling efficient retrieval via large sequential writes. CARP creates one partition per application rank, automatically ensuring more data partitions as application scale increases. A selective range query only needs to retrieve a fraction of the total data from overlapping partitions. In addition, data is written in large SSTables, which can be read efficiently via large read requests. CARP merges its partially ordered SSTs at query time via a merge-sort operation, which is cheap compared to the I/O cost of retrieving data.

Adaptivity. To adapt to skewed and dynamic key distributions, we recompute CARP partitions as the incoming distribution changes. Unlike conventional databases, we do not repartition data that has already been written to disk. We simply record the change in partition boundaries and continue appending new data. Over the lifetime of a run, the range allocated to a CARP partition changes multiple times. As a corollary, data belonging to one point in the range may end up in different partitions at different times in the run. Instead of incurring write amplification to consolidate this data, we simply postpone this consolidation (also called *compaction*) to query time.

V. Design of CARP

In this section we provide an overview of CARP's architecture. We describe the scalable way data is routed to ranks by key (§V-A), the triggers that allow CARP to determine that data needs to be repartitioned (§V-B), the protocol that allows ranks to determine a new partitioning for the data (§V-C), and a reference storage backend design for logging partition data efficiently to storage (§V-D).

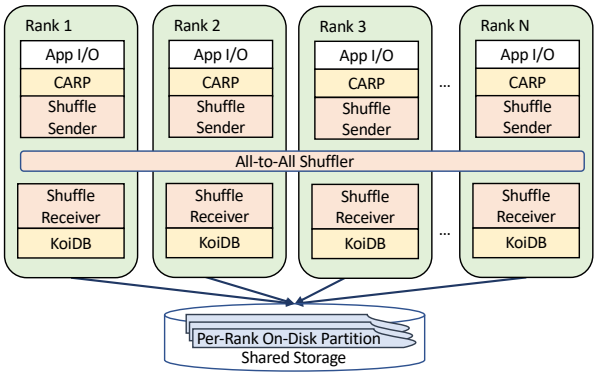


Fig. 3: Example of an N-rank application using CARP. Writes are intercepted and partitioned via an all-to-all shuffler. Each rank is both a sender and receiver. Partitioned data collected by receivers is stored by a local storage backend (KoiDB).

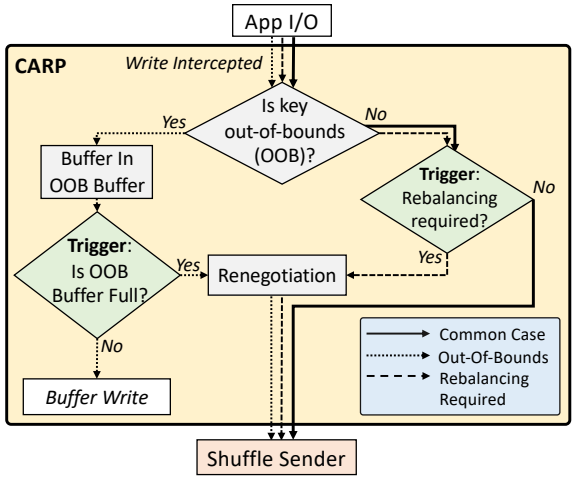


Fig. 4: CARP data and control flow. Application data is routed by the shuffle sender to its partition. But, if its key is outside all partition bounds, then data is added to an Out-Of-Bounds (OOB) buffer. When the OOB buffer fills up or partitions become imbalanced, CARP renegotiates the partition table.

A. Communication: Scalable data flow

Figure 3 shows the architecture of CARP using an example parallel application with N ranks. Application data is intercepted by CARP as a stream of records, one per each particle or cell. Each application rank owns one partition and acts as both a producer and a consumer of data. These records are routed to the corresponding partition on the basis of the key via an operation called *shuffling*. On the receiving end, the local partition is managed via a local instance of a storage backend called KoiDB (§V-D). This KoiDB instance is responsible for collecting data and writing it out to a per-rank on-disk log. For efficient shuffling, we leverage the scalable all-to-all communication overlay from the DeltaFS project [7], as it has been demonstrated to scale up to 131,072 ranks.

Due to the transmission delay introduced by shuffling, data written through CARP is only available to query at the end of a checkpointing *epoch*. At the end of each epoch, CARP flushes all data to the disk. By doing so, it aligns its fault-tolerance semantics with those of typical scientific applications.

B. Detection: Triggering repartitioning

CARP uses a triggering mechanism to determine if the current partitioning of data is leading the system to a state of load imbalance (i.e., routing more data to a subset of the partitions). Imbalanced loads affect the write path by creating stragglers and affect the query path when larger partitions are traversed to return query data. Figure 4 shows the data flow from the application to the shuffle sender layer. This is the path that CARP monitors to determine whether to trigger data repartitioning. We describe CARP's control flow through the three possible cases of data flow from the application to the shuffle sender layer below.

Common Case. In the common case CARP shuffles data as per the assigned partition ranges as described previously in §V-A. Collectively these ranges form a *partition table* that is replicated across all ranks. A distribution of keys transmitted by each rank is stored locally and used when it becomes necessary to participate in a global renegotiation of the partition table (§V-C1).

Out-Of-Bounds. As simulations progress new keys are generated. If a rank is asked to insert a key that is currently out of the partition table bounds, then there is no valid destination for the data. In this case CARP temporarily buffers the data in an in-memory buffer on the sender called the *Out-Of-Bounds (OOB) Buffer*. Once the OOB buffer reaches a predetermined threshold, a renegotiation of the partition table is triggered.

Rebalancing Required. This trigger is only needed to adapt to distribution changes *within* an epoch — for new epochs CARP bootstraps partitions from scratch. To resolve intra-epoch drift, a renegotiation should be triggered to compute a new and more relevant partition table. Instead of using background communication mechanisms to robustly detect load outliers, we have found it simpler to assume that a rebalancing is required at periodic intervals within an epoch. Some applications such as AMR (Adaptive Mesh Refinement) codes are aware of when they refine and can signal CARP for more precise control over renegotiation. We explore the impact of a fixed interval further in our evaluation.

C. Renegotiation: Determining new partitions

The goal of renegotiation is to rebalance the partition boundaries used by the shuffle layer to route data to its destination. Renegotiation replaces the partition table stored in each rank with a new more balanced one based on the latest statistics of the global distribution of the keyspace. Renegotiation is similar to distributed snapshotting algorithms such as *Chandy-Lamport* [19] but is designed to trade off some accuracy for scalability and performance. As a result, the computed global distribution is not a perfect representation of the actual distribution, but our evaluation shows that any estimation errors result in negligible imbalance in partition load. Our scalable implementation of renegotiation is described in §VI. The steps involved in the renegotiation protocol are as follows:

1) A rank triggers a renegotiation round by notifying all ranks to start the process. This pauses shuffling.

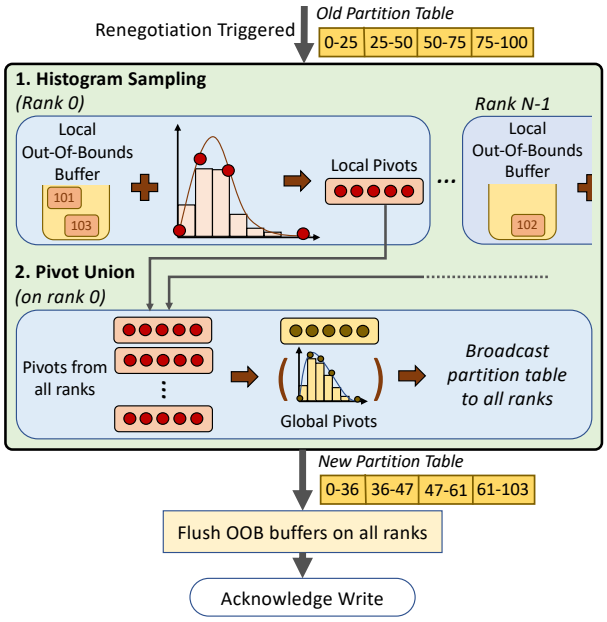


Fig. 5: Partition table renegotiation relies on pivots computed from histograms of keys tracked on each rank. Pivots are merged, and the new global histogram is divided into equal-mass bins before being broadcasted.

2) Local distribution estimates are computed on all ranks and then collected and merged on rank 0 to form an estimate of the global distribution.
3) A new partition table is computed and broadcast.
4) All ranks switch to the new partition table and flush their Out-Of-Bounds buffers according to the new partitions.
5) Ranks reset local distribution statistics, resume shuffling.

Local distribution estimates are computed and merged using summary statistics primitives. Renegotiation is initiated by trigger that can fire on any rank. We describe CARP's summary statistics primitives and trigger design next.

1) Summary Statistics: CARP ranks use summary statistics primitives to construct the global distribution during renegotiation. We use *histogram-based sampling* to track and aggregate ranks' local key distributions. Different quantile estimation techniques can be plugged into CARP, but we have found that histogram-based sampling is efficient to compute and allows for control over trading compactness for accuracy. CARP tracks rank-local key distributions using histograms. Each rank's histogram consists of one bin per application rank (or partition). For each processed key the corresponding bin counter is incremented. This histogram represents a lightweight, lossy representation of the local key distribution. When a renegotiation is invoked the global distribution is constructed using two primitives: histogram sampling and pivot union, as shown in Fig. 5.

When ranks are notified of a renegotiation round, they compute *pivots* via the **histogram sampling** primitive. Pivots are a compact and lossy representation of a distribution that can be efficiently computed, communicated, and merged. Pivots are a set of k points in the keyspace that divide the histogram into k partitions of equal mass, i.e., bins containing the

same number of samples. Increasing k reduces representation lossiness but requires more network communication for the additional information. For skewed distributions, this results in more pivots concentrated in histogram regions where bin counts are high and vice versa. Pivots are calculated by linear interpolation between bin boundaries. We also factor in the keys in the local OOB buffer for pivot computation.

Pivots from all ranks are collected at a designated rank (Rank 0) and aggregated to form pivots representing the global distribution via the **pivot union** primitive. This operation uses the information captured in rank-local pivots to construct a global distribution. This global distribution is then resampled to compute the new partition table that is broadcast to all ranks.

2) Triggers: Renegotiation triggers are evaluated at each rank to determine when to renegotiate, as shown in Fig. 4.

The **Out-of-Bounds trigger** is used to extend partition boundaries when many keys outside the partition table boundaries are encountered. We partition the known keyspace without any gaps, so an invocation of this trigger strictly extends the allocated keyspace. An in-memory *Out-of-Bounds* (OOB) buffer is used on each rank to temporarily store keys that are currently outside the bounds of the current partition table. The OOB buffer has a predetermined size and when it fills up a renegotiation is triggered. The contents of all ranks' OOB buffers are factored into the new partition table so that the newly computed table has destinations for those keys. Keys in the OOB buffers are flushed to their respective destinations once the renegotiation completes. We have found OOB buffers with a capacity of 512–1024 items per rank sufficiently effective.

This trigger is also used to bootstrap CARP at the beginning of each epoch. As there is no partition table when an epoch begins, all ranks collect writes into their OOB buffers, and invoke a renegotiation to find the first partition table to start shuffling with.

The **Rebalancing trigger** is used to account for key distribution drift over time causing partition load to become imbalanced. In principle this trigger could be designed to fire only when key distribution drift is detected, however we have found that invoking it periodically is both simpler and effective. We found frequent fixed-interval renegotiation to provide effective partitioning without having a measurable runtime overhead (§VII-C4).

D. Storage: Logging for efficiency

KoiDB is our reference implementation of CARP's storage backend. Its output is written to a parallel file system and can be directly queried by a query client on a single node. As query clients access files in read-only mode, multiple concurrent query clients are automatically supported. KoiDB collects data from the shuffle receiver and writes it to an append-only log on shared storage. KoiDB instances are local to each rank and operate independently of each other. We discuss using CARP with other storage backends in §VIII.

Fig. 6 shows KoiDB's on-disk log format. KoiDB produces this output by first collecting shuffled records in a memory buffer. When the buffer fills, KoiDB compacts the data into an *SSTable* (or SST) that is then appended to the log. Compaction operations include (optionally) sorting the contents by key, serializing the keys and values into separate sub-blocks for more efficient query-time parsing, and writing the serialized SST to storage. A manifest entry containing the SST's key range and location is also written into a manifest block that is used to find relevant SSTs for queries. Storing this manifest results in a small space amplification (~.01%). KoiDB uses double-buffering to allow compaction to run in the background while shuffling continues in the foreground.

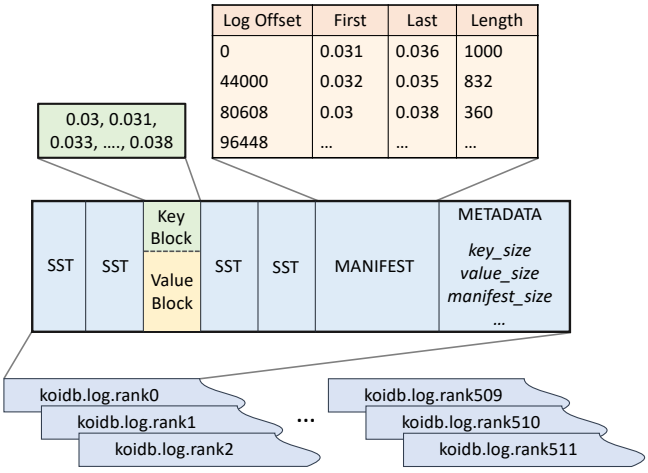


Fig. 6: KoiDB on-disk log format. Each log consists of SSTables organized into contiguous key and value blocks. Each SST has an entry in the manifest along with metadata.

We now describe two additional optimizations KoiDB applies to SSTs for query performance:

Repartitioning to refine SSTs. CARP partitioning quality can be degraded by *stray keys* created when CARP partition tables are updated. These keys end up being misdelivered because their correct shuffle destination changes between dispatch and delivery due to renegotiation. Letting them be written to the wrong partition reduces SST selectivity and will increase query latency as more SSTs need to be read for each query. One way to avoid stray keys is by flushing the network before each renegotiation, but this increases the cost of invoking the primitive. KoiDB mitigates this by simultaneously maintaining multiple open SSTs and separating stray keys into a different SST. As we show in §VII-C3, this improves the selectivity of CARP partitions by up to 48×.

Subpartitioning to create smaller SSTs. To further reduce the read amplification for highly selective queries, KoiDB can subdivide a rank's SSTs into smaller ranges before writing them out. Smaller SSTs can be retrieved faster, but a large number of subpartitions can also increase CARP's CPU and metadata overhead and adversely affect runtime.

VI. Implementation

In this section we describe CARP's implementation and its relationship to scalability, memory footprint, and runtime overhead.

CARP [8] consists of ~10,000 lines of C/C++ code. Application writes are intercepted using a dynamically preloaded shared library. The CARP API can also be called directly without using a preload library. CARP uses two instances of the DeltaFS shuffle service [20] for communications: the *data* instance batches messages into 32KB buffers for high throughput, while the *control* instance is used for renegotiation control messages and uses no batching for low latency. The shuffle service is built using the Mochi [21] Mercury RPC library [22]. We use RPCs as both shuffling and renegotiation are asynchronous operations better suited to RPC semantics (it also allows us to avoid complications with multiple concurrent MPI communicators). CARP creates its own threads for network communication, RPC callbacks, and asynchronous compaction in KoiDB. This setup allows straggler ranks to use multiple cores and provides some tolerance for load imbalance.

Scalability. CARP is designed to scale by composing scalable operations — the shuffle service has been demonstrated to scale to 131,072 ranks [7], and the renegotiation protocol is designed as a reduction operation with a logarithmic scaling factor (evaluated in §VII-C1). Each instance of CARP's storage backend (KoiDB) is an independent instance and can scale trivially. By default, CARP creates one file per rank, but at larger scales, the total number of files can be reduced (if needed) by having a subset of ranks be shuffle receivers.

We call CARP's scalable implementation of renegotiation the *Tree-based Renegotiation Protocol (TRP)*. A naive implementation of renegotiation, as described in §V-C, would require directly collecting all ranks' pivots on one rank before computing the new partition table. Such an implementation will have limited scalability, as its memory and network communication footprint will scale linearly with ranks. Our TRP implementation, on the other hand, uses a reduction tree-based design [23, 24] which scales logarithmically.

TRP is built on the observation that pivot unions (§V-C1), being associative and commutative have all the properties of a lossy reduction operation. Pivots represent distributions and can be merged in any order, but the pivots lose some information in the process. To limit the impact of this lossiness, TRP employs a shallow tree hierarchy with a large fan-out (depth of 3, fan-out of up to 64). The tree's leaf layer consists of all CARP ranks, while intermediate layers consist of equally spaced ranks. Subsets of pivots are reduced on intermediate layers, and a final reduction happens at the root.

Memory Footprint. As CARP runs in application processes, it needs to have a low memory footprint to avoid competing with the application for memory. CARP achieves this by streaming application data directly to the shuffle pipeline. The primary source of memory overhead is due to buffers used to batch I/O for network and storage efficiency. To illustrate this overhead, we use a run with 4096 ranks and default CARP parameters that results in 27MB per rank[1]. This is less than 1% of the per-core memory budget on the LANL Trinity supercomputer [25].

Runtime Overhead. CARP only runs during application I/O, so it does not impact the compute phase. Data shuffling and triggered renegotiations are potential sources of additional runtime overhead. Shuffle bandwidth increases with the number of writes and quickly outstrips storage bandwidth in our experiments. Thus, the shuffle service itself does not have any runtime overhead. CARP could underutilize storage by pausing I/O for renegotiation, but our experiments show that shuffle receivers buffer enough outstanding writes to keep storage busy and mask this overhead. If needed, CARP can continue routing data using the old partition table while a renegotiation is underway, however we have not found this necessary in our experiments.

VII. Evaluation

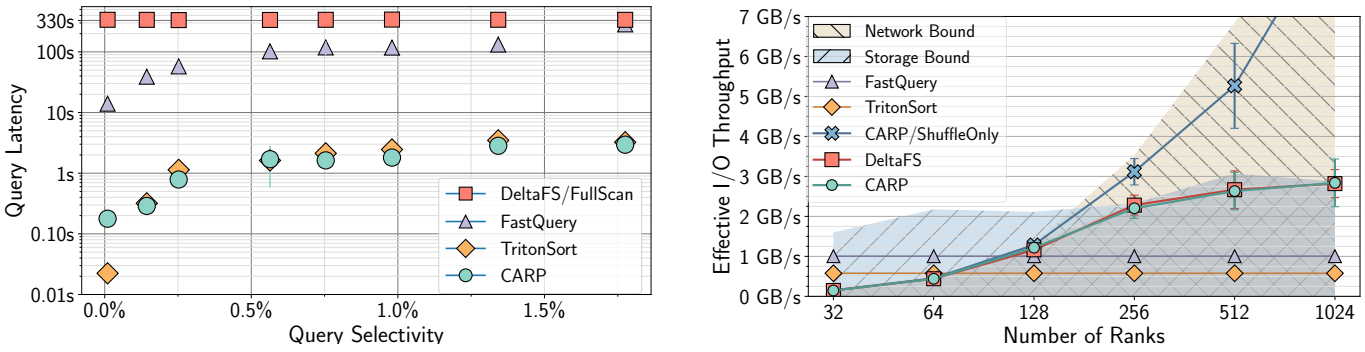
Baselines. We use DeltaFS [7], FastQuery [14], and TritonSort [13] as baselines representing the state of the art from different research communities. DeltaFS represents the state of the art for write-efficient online hash-partitioning, FastQuery employs auxiliary indices, and TritonSort performs data reordering.

Benchmark workloads. We evaluate the performance of CARP using traces collected from a 512-rank VPIC simulation [1], a popular particle simulation code described in §III, and YCSB [26], a standard benchmark suite with varying key-value workloads. To eliminate variability across simulation runs and ensure reproducibility, we replay traces captured from VPIC rather than using it as a streaming workload. We use the computational plasma trace described in section III for all our experiments. We index VPIC traces using energy as the key. This results in a 4 byte floating-point key and a 56 byte record for the rest of the particle data.

Experimental setup. We use a 32-node compute cluster with each node having two 8-core Intel Xeon E5-2670 CPUs and 64GB DRAM. The nodes are connected via 40Gbps Infiniband QDR. Simulation output is written to a shared Lustre storage cluster consisting of 20 nodes identical to those in the compute cluster, each having a 240GiB SSD. DeltaFS, FastQuery, and CARP experiments use the compute cluster, while TritonSort runs directly on the Lustre nodes. TritonSort has identical storage resources as other approaches and is able to achieve higher throughput out of storage by not incurring the overhead of Lustre's coordination. While TritonSort appears to have lower compute resources, all applications are bottlenecked by the storage bandwidth and their runtime is dictated by their write amplification. FastQuery provides a number of tunable parameters, and we report the numbers from the best performing parameters on our cluster.

The rest of this section is organized as follows: In §VII-A we evaluate CARP's query performance. We evaluate CARP's runtime overhead and compare it with existing approaches in §VII-B. Finally, in §VII-C, we show selected results of a sensitivity analysis aimed at understanding the impact of

[1]Each rank uses 2MB for shuffle RPC buffers [7], 24MB for two KoiDB memtables, 16KB for 4096*4B partition table entries, 16KB for partition shuffle counts, and 32KB for 512-entry OOB buffer with 64B records.



(a) Query Latency

(b) I/O Throughput

Fig. 7: (Left) CARP matches the query latency of TritonSort's fully-sorted data layout and outperforms both FastQuery by 100×. (Right) CARP incurs no overhead over unpartitioned I/O and is 3-5× faster than post-processing.

tunable parameters and optimizations on CARP's runtime overhead, load balancing effectiveness, and index quality.

A. Query Performance

In this subsection we evaluate CARP's partitioned output using two aspects: query latency and read amplification. Query latency is a function of how much data the index needs to read and how efficient the storage layout is in serving that data. For CARP, minimum effective query selectivity is capped by the size of one CARP partition — 0.18% (or $\frac{1}{512}$) for 512 ranks. This percentage decreases with scale as the number of partitions increases and when subpartitioning is enabled. We now discuss CARP's query performance over two query suites: one constructed by us, and a YCSB query suite.

Observation 1: *CARP can serve most queries as efficiently as a fully sorted layout and 1-2 orders of magnitude faster than state of the art auxiliary indexing approaches.*

Query Suite. For query latency experiments we index 12 timesteps from the simulation. Each timestep is 188GB of data, and the total simulation data is 2.2TB. Query latencies for eight queries with different selectivity are shown in Fig. 7a (each query is repeated 3 times and averaged). We compare the query latency for CARP's partitioned ordered output with that of FastQuery and a sorted, clustered index. We also provide numbers for a *full scan* over unindexed data for reference. The sorted output uses 12MB SSTs and a manifest with one entry per SST, a format similar to KoiDB. We refer to the sorted, clustered index as TritonSort for convenience, but all sorts generate identical outputs. Queries for both CARP and TritonSort are processed by the same query client. The manifest is read first to find corresponding SSTs and then key blocks of corresponding SSTs are read in parallel. CARP SSTs overlap and must be merge-sorted for ordered range query semantics. The latency numbers for CARP include this sorting cost. The query client for CARP and TritonSort is run on a single compute node with access to the Lustre filesystem. FastQuery numbers are measured using its own query client reading from its natively supported HDF5 format. All query latency numbers include the time taken to fetch keys matching a query from a storage cluster to a query client and the compute time to subsequently process/filter the data fetched.

For both CARP and TritonSort, query latency appears to be linearly proportional to the amount of data read. Despite incurring an additional sorting cost, CARP's total query latency is similar to TritonSort. CARP is slower for highly selective queries (177ms vs 22ms for a query with 0.01% selectivity) because it is forced to read full partitions, but it is able to match (and even outperform) TritonSort for larger query ranges with selectivity $> 0.05\%$. FastQuery is much slower than either of the two approaches. This is partially due to it needing to read large bitmap indices, but largely due to it being an auxiliary index, requiring small, random reads.

Observation 2: *Despite incurring extra sorting overhead, CARP matches TritonSort's query latency performance for all except extremely selective queries ($< 0.05\%$ selectivity).*

YCSB Benchmark Suite. To thoroughly evaluate the query performance of CARP's approximate partitioning we use Workload E from YCSB (Fig. 8). Workload E consists of short range queries and inserts in a 19:1 ratio. The range queries consist of up to 100 keys generated from a Zipfian distribution and are reordered by a random hash function. We drop inserts from our benchmark suite as CARP and TritonSort are not online stores but are instead transient services with different insert and query pathways. We define the workload's ranges in terms of fully ordered SSTs. We interpret YCSB query ranges as SST numbers in TritonSort's output and translate them into equivalent key ranges that we use for both CARP and TritonSort. The range SST# $(251, 350)$ would therefore query the key range stored in TritonSort SSTs numbered 251 to 350. Since KoiDB SSTs are smaller and more fragmented, the same query with CARP would result in more SSTs read.

We construct 4 query batches of different fixed widths (5, 20, 50, and 100 SSTs) corresponding to query selectivity of 0.03%–0.6%. The starting index is drawn from YCSB's Zipfian distribution in the range SST# (0, 18266), where 18266 is the number of SSTs in each timestep. We use 16 I/O threads to fetch SSTs for each query in parallel. For each width, we construct a batch of 1000 queries per timestep with the order randomized by YCSB's hash function (`fnvhash`). Fig. 8

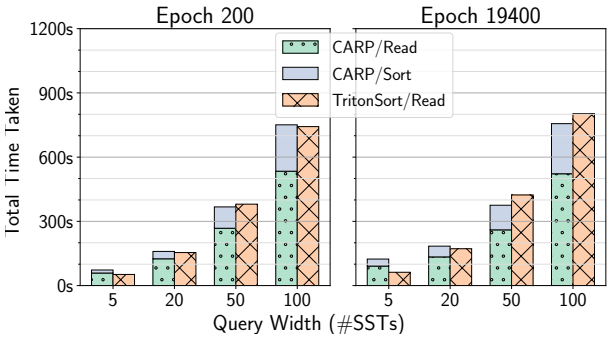


Fig. 8: Time taken for a YCSB query suite, demonstrating similar trends to the left-hand figure. CARP is slower for highly selective queries but comparable/better for larger queries despite the sorting overhead.

compares the total time taken by a batch of queries for two different timesteps and 4 query widths for each timestep, and it reaffirms the trends from Fig. 7a. CARP's median selectivity of 0.07% (with 4-way KoiDB subpartitioning enabled) enables us to be competitive for queries with a width of 20 SSTs and greater. For larger queries, the cost of sorting starts to become a greater proportion of CARP's query latency, but the aggregate latency is still close.

A surprising takeaway from these latency experiments is that CARP's partial ordering seems to result in the most efficient storage layout. CARP reads are faster than the fully sorted layout, and query performance is similar despite us incurring the extra overhead of merging SSTs. We attribute this to our layout being amenable to parallel reads from different storage nodes of a parallel filesystem — it has enough contiguity to be read efficiently vs small random I/Os, but is distributed enough to allow for parallel processing of a query.

B. Write Path

Observation 3: *CARP is 2.8–4.9× faster than post-processing approaches with no overhead on application performance.*

Fig. 7b measures the effective I/O throughput of four different index building approaches ingesting 188 GB of VPIC data (corresponding to 3.5B particles or one timestep at 512 ranks). The effective I/O throughput is computed by dividing the application data volume by the total runtime. For in-situ approaches, their runtime overhead is included in the application runtime, while for post-processing approaches, it is obtained by adding the post-processing time. Each run is repeated 9 times and averaged.

To measure I/O throughput, we keep the total amount of data generated constant across different scales as the write performance of SSDs decreases as they fill up. Allowing the amount of data written to scale up with the number of ranks results in higher scales unfairly getting a lower storage throughput. The achievable storage bandwidth from our cluster (*Storage Bound* in Fig. 7b) increases with the number of application ranks: from 1.6 GB/s at 32 ranks to saturating the storage with 3 GB/s at 512 ranks. At 1024 ranks, the increased contention from a large number of parallel writers causes a small dip in achievable throughput. For 1024 ranks, we divide the 512-rank trace into two halves to keep the total data constant.

We compare CARP with DeltaFS, FastQuery, and TritonSort. CARP and DeltaFS are embedded within the application and hence are evaluated at different scales. FastQuery and TritonSort are post-processing approaches and are run after VPIC completes. For the post-processing approaches, effective throughput is obtained by dividing the total data written by the total time taken (VPIC time and post-processing time). For the in-situ approaches VPIC time includes the online index building time.

FastQuery and TritonSort. As shown in Fig. 7b, the additional time taken by post-processing approaches (FastQuery and TritonSort) significantly reduces the effective application bandwidth from 3GB/s to ~1 GB/s and ~0.6 GB/s respectively. This represents a slowdown of 2.8× for FastQuery, and 4.9× for TritonSort. TritonSort incurs a much larger slowdown as it creates a clustered index, which requires four passes over all data for out-of-core sorts. FastQuery scans the data once, creates index structures, and writes them to storage, but takes an additional 24% space to store indexes for a single attribute.

DeltaFS. As discussed in §II, DeltaFS embeds with VPIC and reorganizes data via an all-to-all shuffle while it is in transit from the application to storage. DeltaFS uses the same 3-hop shuffle used by CARP (§V-A). DeltaFS uses a hash of the particle ID to partition incoming data. At smaller scales, DeltaFS is bound by the available shuffle throughput (*Network Bound* in Fig. 7b). As the aggregate shuffle throughput increases with scale, DeltaFS is able to fully saturate the storage layer, incurring no overhead over raw VPIC throughput.

CARP. We measure CARP network overhead (*CARP/ShuffleOnly*) and end-to-end performance (*CARP*) separately in Fig. 7b. For CARP/ShuffleOnly, we drop data once it is received at a shuffle receiver so as to not be bound by storage performance. We see that CARP/ShuffleOnly scales with the available shuffle bandwidth, incurring a small overhead for renegotiation rounds and the residual load imbalance. However, this overhead does not matter for the end-to-end performance as CARP quickly becomes bound by the available storage bandwidth. When the network bandwidth exceeds storage bandwidth, CARP has no impact on end-to-end performance. By avoiding post-processing, CARP enables indexes to be built 2.8–4.9× faster.

Observation 4: *Static partitioning can quickly devolve to reading orders of magnitude more data than CARP by not adapting to key distribution drift.*

Fig. 9 is a study of how frequently partition tables need to be recomputed for our VPIC trace to generate balanced partitions. We construct partitions from a perfect knowledge of different simulation timesteps and measure how well they fit subsequent timesteps. The green line simulates a static partitioning approach where partitions are computed from a perfect knowledge of the first timestep but are not changed afterwards. The load balance of a static partition scheme worsens as the

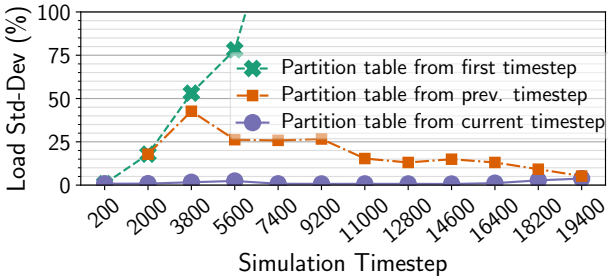


Fig. 9: Simulated performance of different static partitioning schemes in generating balanced partitions (lower load std-dev implies better partition balance). Reusing a partition table from the first timestep results in the worst load-balance. Tables from the previous timestep fare better, but perform poorly when the simulation is the most active. Tables from the current timestep fit the best (true by definition).

key distribution drifts over time. Next, we measure how well a partition table from the previous timesteps fits the current timestep. This works better but still demonstrates a significant load imbalance around timestep 3800 when simulation entropy is high. As the simulation converges and the entropy between adjacent timesteps reduces, reusing older partition tables fares better. Finally, we see that partition tables obtained from the current timestep fit that timestep very well. This is true by definition, and the minor load imbalance arises from the lossiness of the histogram approach to capture distributions.

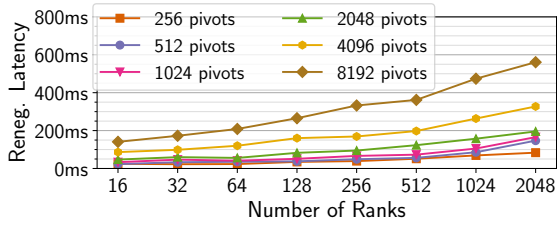
We conclude that any partitioning approach should recompute its tables at least once every timestep, more if there is intra-timestep entropy. We emphasize that the purple line in Fig. 9 represents an upper bound on CARP's load balance, as the partitions used for this benchmark are *oracle partitions*. Since CARP partitions data as it arrives, it does not have the required information to compute these.

C. Sensitivity Analyses and Microbenchmarks

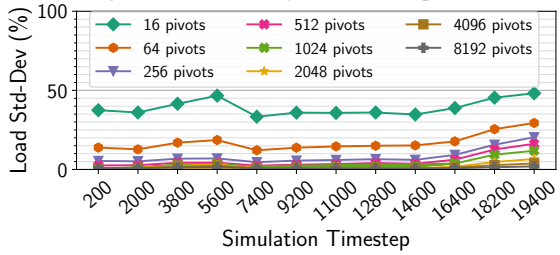
1) Renegotiation Protocol Scalability: Fig. 10a shows the time taken by a single renegotiation round across different scales and pivot counts. As described in §VI, we implement renegotiation using a reduction tree and hence expect it to scale logarithmically. Fig. 10a shows the logarithmic scalability of our implementation from 16 to 2048 ranks for 6 different values of the pivot count parameter. Increasing the number of pivots computed increases the size of the messages exchanged. This results in proportionally higher round latency. 512 pivots are sufficient for CARP to accurately track distributions (discussed below), and the latency of a single round is only 150ms even at 2048 ranks.

The absolute values of renegotiation latency are handicapped by us using an emulated network stack (IPoIB) for a fair comparison with baseline approaches that use sockets. We expect these numbers to be much lower if run natively on a modern interconnect.

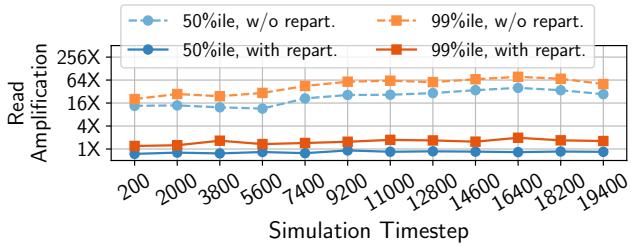
2) Impact of Pivot Count: Fig. 10b shows the results from a micro-benchmark that tests the lossiness of our pivot calculation scheme. Pivots are an approximate representation of a distribution. The higher the pivot count, the better the



(a) Renegotiation scalability at different pivot counts



(b) Pivot Count vs Histogram Fidelity



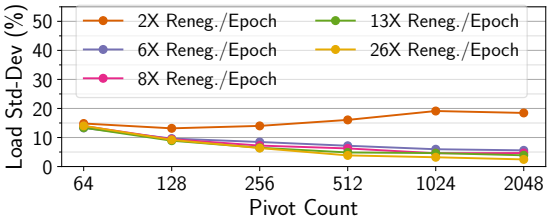
(c) Impact of repartitioning on 50th and 99th percentile RAF

Fig. 10: (Top) Renegotiation scales logarithmically with the number of ranks and takes longer if more pivots are exchanged. (Center) 512 pivots enable reasonably accurate reconstruction for even highly skewed distributions, with diminishing returns after. (Bottom) Both median and tail RAFs are significantly lowered by KoiDB partitioning, from 16-64× to 1-2×, across all timesteps.

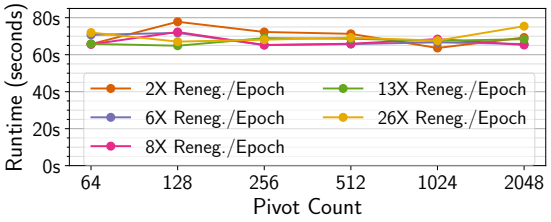
approximation. We compute oracle pivots from a full key distribution of each of the 12 different timesteps for different pivot counts and check how well the partition table calculated from those pivots fits that timestep's keys. We use standard deviation in partition load (`std-dev`) as a proxy for the lossiness of the pivot calculation. A lossless scheme would result in perfect partitions, resulting in zero `std-dev`. Higher `std-dev` implies more lossiness.

In Fig. 10b we see that higher pivot counts lead to lower load imbalance with diminishing returns beyond 256 pivots. We also see that the last two timesteps are harder to reconstruct than the others. This is because of the extremely skewed nature of those timesteps' distributions (Fig. 1). All VPIC timesteps have long tails, but when they become extremely long (towards the end of the simulation) more pivots are required to accurately capture those distributions.

3) Impact of KoiDB: We now summarize how KoiDB (§V-D) improves partition quality. We introduce a measure called *Read Amplification (RA)* to measure CARP partition quality. We define read amplification as the ratio of the size of actual CARP partitions vs (hypothetical) perfectly balanced partitions. An RA of 1× for all partitions is ideal, as balanced



(a) Impact of CARP parameters on partition load balance



(b) Impact of CARP parameters on write performance

Fig. 11: (Top) Renegotiating multiple times within a timestep is necessary for load-balance, but provides marginal benefit beyond a point. More pivot counts produce a better load balance, with diminishing returns. (Bottom) Up to 1024 pivots, increasing the pivot count leads marginal improvement in runtime. Renegotiating frequently does not penalize runtime — the cost of frequent renegotiations is made up by the gains in load balance.

partitions are better on both the write and the query paths. It is also unachievable as CARP tries to predict future key distributions and partition in a single pass, which is inevitably an imperfect operation.

We find that repartitioning, by separating out stray keys, reduces the average RA by up to 48× (Fig. 10c). For *subpartitioning*, we find that 2-way and 4-way subpartitioning improve average latencies for highly selective queries by 28% and 43% respectively with no observable runtime overhead. We omit detailed results for subpartitioning as we have observed its impact to be largely predictable, as described in §V-D.

4) Tuning CARP: We now summarize key takeaways on the performance impact of CARP's two main tunable parameters: renegotiation frequency and pivot counts (Fig. 11). We vary renegotiation frequencies between 2×/epoch to 26×/epoch and pivot count from 64 to 2048. We then measure CARP's partition load balance and application runtime performance. Note that it is important to optimize partition imbalance even if it does not affect runtime to provide more predictable and uniform query performance.

We find that these parameters have a noticeable impact on partition load balance but not on CARP's shuffling runtime. At the lowest values of these parameters (64 pivots and 2× renegotiations per epoch), CARP partitions have a normalized standard deviation of 14%. At the highest values (2048 epochs and 26×/epoch) this drops down to 2%. More pivots provide noticeable load balance improvement up to 512 pivots and then returns diminish. It is beneficial to increase the renegotiation intervals from 2×/epoch to 6×/epoch, but we get minimal gains beyond that interval. Surprisingly, none of these parameters seem to impact runtime in any measurable way. We attribute this to CARP's imbalance-tolerant implementation (idle ranks can cede CPU time to straggling ranks, §VI).

Tuning Conclusion. While CARP can be tuned for incremental performance benefits, it provides stable performance across a wide range of parameters. Even with the worst performing parameters, CARP partitions only deviate by 14% on average. CARP will still provide excellent query performance for queries not reading from larger partitions. A moderate number of pivots (~512) can represent even extremely skewed distributions with long tails, and renegotiation frequency has marginal impact on partition quality as it only addresses intra-epoch drift.

VIII. Discussion

In this section, we discuss how CARP can be adapted for different query types, analyses, and architectures. We also describe how CARP can interoperate with other storage formats and indexing techniques.

Multi-attribute Queries. In this paper, we focus on using CARP to build a sorted, clustered index on a preferred attribute. For applications such as VPIC, indexing on a single attribute is sufficient to identify and retrieve points of interest (high-energy particles), and apply subsequent transformations in memory. However, CARP can be extended to build sorted, auxiliary attributes on additional attributes. This would require CARP to track distributions of all configured attributes and use a two-stage shuffling pipeline as described below:

- First, the entire row is shuffled using the primary attribute as usual. The receiver serializes all rows, assigns them a unique `row_id`, and writes them to the local storage backend. Each receiver then computes a `(key, partition_id, row_id)` tuple for each additional indexed attribute and shuffles it via the same pipeline.
- Each receiver of the auxiliary attribute tuples writes them to separate storage backend instances, where each row points to the full row on the primary key partition.

The additional attributes will not have the query performance of the primary attribute but will still benefit from the space efficiency and index lookup performance of sorted indices (vs bitmap indexes).

Applications and architectures. CARP can be used to ingest data from any parallel application where storage bandwidth is the bottleneck. The assumptions we make are typical of such analysis workflows:

1) Data is not modified once ingested.
2) Spare network bandwidth is available for shuffling. This is true by definition for any storage-bound workflow and holds for most HPC clusters. The Aurora cluster [27] at ANL, for example, has a bisection bandwidth of 690 TB/s, more than 20× the storage bandwidth (31 TB/s).
3) Write performance is paramount, and the goal is to provide acceptable query performance without compromising write performance. Where the goal is optimal query performance (as with a web service), a distributed database is more appropriate.

Storage Formats. Developing our own storage backend allowed us to identify the most important properties for

an efficient storage backend and on-disk format. CARP-partitioned output can be directly written to columnar formats like Parquet. This would automatically accelerate queries, as Parquet maintains statistics (min/max etc.) on each rowgroup, and CARP-partitioned rowgroups would have a tighter range and require less I/O at query time.

However, as we show in §VII-C, repartitioning on shuffle receivers significantly improves partition quality. A writer that can receive repartitioning hints from CARP and use them to further optimize data before writing to Parquet or other storage formats can improve query latencies by 1-2 orders of magnitude.

Indexing Techniques. As an alternative to auxiliary CARP indices as described above, it is also possible to combine CARP with other indexing techniques. This also requires query engines that can generate efficient query plans by leveraging the strengths of different indexing techniques. We provide some examples below:

1) Different auxiliary index structures (such as bitmap indexes) can be built either in-situ on auxiliary nodes, or as a post-processing stage.
2) On the query path, raw-data indexing approaches can further improve index quality incrementally, in response to user queries.
3) CARP's approximately sorted output can be incrementally converted into a fully sorted layout on the query path by writing back the merged SSTs that are computed for user queries.

To leverage the full potential of different indexing techniques, it is necessary to develop end-to-end analysis engines that can understand user analysis requirements and generate an appropriate combination of in-situ embedded, in-situ auxiliary, and (if necessary) post-processing transformations using different techniques.

IX. Related Work

Data structures such as LSM-Trees [28], B+ Trees [29], and their variants [15, 30] are employed by online databases to order data. These data structures heavily reorganize data internally and are more inefficient at writes than bulk post-processing approaches [18]. Multiple in-situ analysis frameworks have been proposed including PreDatA [31], GLEAN [32, 33], NESSIE [34], DataSpaces [35], and ADIOS [36]. These systems are designed such that auxiliary nodes are used to perform analysis tasks. ADIOS supports in-situ generation of range query indices using FastBit [37] (the same bitmap index used by FastQuery) using auxiliary nodes. In-situ generation of bitmap indices would be faster than post-processing via FastQuery, but at the cost of dedicated resources, and the space overhead and query performance limitations of bitmap indices would still remain.

Usher et al. [38] describes an in-situ indexing system that aggregates application data into spatial locality-preserving data layouts. Their approach requires provisioning dedicated aggregator nodes which collect spatially-partitioned particle data from the application, add a bitmap index to the data, and write it to storage. Particle codes already have spatial partitions because of how they decompose the problem. This system only preserves these pre-existing partitions. CARP is able to create partitioning along arbitrary dimensions using all-to-all shuffling. Further, CARP does not require dedicated resources, introduces more powerful distribution monitoring constructs, overlaps indexing and I/O for extremely high storage utilization, and confers maximum benefits of an in-situ partitioning approach. Since CARP does not require any dedicated resources, these two approaches can be composed together for richer partitioning capabilities.

Slalom [39] and VETI [40] describe raw-data indexing approaches. Raw data indexing works on the query path to exploit pre-existing order in data and adaptively reorder data in response to query patterns. Scalable variants of these techniques can complement approximate indexes on the write-path to further optimize query performance for exploratory queries, but they are not a substitute for in-situ indexing on the write path. WiscKey [41] introduces the notion of separating keys and values in LSM-Trees for lower write amplification, similar to our discussion on clustered vs auxiliary indexes. SuRF [42] can help reduce redundant reads for range queries with sparse keyspaces but can not help speed up queries for data with no keyspace gaps.

X. Conclusion

CARP demonstrates that a sorted, clustered layout for range queries over a parallel application can be generated in a single streaming pass without any impact on application runtime. This accelerates subsequent analyses regardless of the workflow employed — it can provide highly efficient queries by itself, or accelerate a query plan with auxiliary indices, or provide tighter partitions for columnar storage formats to exploit. Due to its low memory footprint, CARP can co-exist with other in-situ approaches running on the same system. End-to-end analysis stacks that produce an optimal combination of in-situ and post-hoc indexes for a given set of query requirements remain future work.

Acknowledgment

We thank the anonymous reviewers of our paper for their valuable comments on improving the manuscript. This manuscript has been approved for unlimited release and has been assigned LA-UR-24-29173. This work has been authored by an employee of Triad National Security, LLC which operates Los Alamos National Lab with the U.S. Department of Energy/National Nuclear Security Administration. We also thank the member companies of the PDL Consortium (Amazon, Google, Honda, IBM Research, Intel, Jane Street, Meta, Microsoft Research, Oracle, Pure Storage, Salesforce, Samsung Semiconductor, Two Sigma, and Western Digital) for their interest, insights, feedback, and support.

References

[1] S. Byna, J. Chou, O. Rübel, Prabhat, H. Karimabadi, W. S. Daughter, V. Roytershteyn, E. W. Bethel, M. Howison, K. J. Hsu, K. W. Lin, A. Shoshani, A. Uselton, and K. Wu, "Parallel I/O, analysis, and visualization of a trillion particle simulation," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2012. DOI: 10.1109/SC.2012.92.

[2] P. Grete, J. C. Dolence, J. M. Miller, J. Brown, B. Ryan, A. Gaspar, F. Glines, S. Swaminarayan, J. Lippuner, C. J. Solomon, G. Shipman, C. Junghans, D. Holladay, J. M. Stone, and L. F. Roberts, "Parthenon—a performance portable block-structured adaptive mesh refinement framework," *The International Journal of High Performance Computing Applications*, vol. 37, no. 5, pp. 465–486, Sep. 1, 2023. DOI: 10.1177/10943420221143775. [Online]. Available: https://doi.org/10.1177/10943420221143775 (visited on 08/21/2024).

[3] W. Zhang, A. Myers, K. Gott, A. Almgren, and J. Bell, "AMReX: Block-structured adaptive mesh refinement for multiphysics applications," *The International Journal of High Performance Computing Applications*, vol. 35, no. 6, pp. 508–526, Nov. 1, 2021. DOI: 10.1177/10943420211022811. [Online]. Available: https://doi.org/10.1177/10943420211022811 (visited on 06/19/2024).

[4] *Phoebus*, https://github.com/lanl/phoebus, Aug. 2024.

[5] A. C. Bauer, H. Abbasi, J. Ahrens, H. Childs, B. Geveci, S. Klasky, K. Moreland, P. O'Leary, V. Vishwanath, B. Whitlock, and E. W. Bethel, "*In Situ* Methods, Infrastructures, and Applications on High Performance Computing Platforms," *Computer Graphics Forum*, vol. 35, no. 3, pp. 577–597, Jun. 2016. DOI: 10.1111/cgf.12930. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1111/cgf.12930 (visited on 04/01/2024).

[6] H. Childs, "In Situ Processing," Nov. 1, 2012. [Online]. Available: https://escholarship.org/uc/item/3st8x19d (visited on 08/23/2024).

[7] Q. Zheng, C. D. Cranor, D. Guo, G. R. Ganger, G. Amvrosiadis, G. A. Gibson, B. W. Settlemyer, G. Grider, and F. Guo, "Scaling embedded in-situ indexing with DeltaFS," in *Proceedings - International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018*, 2018. DOI: 10.1109/SC.2018.00006.

[8] A. Jain and C. D. Cranor, *Carp: Range query-optimized indexing for streaming data*, version v1.7, Jun. 2024. DOI: 10.5281/zenodo.13314155. [Online]. Available: https://doi.org/10.5281/zenodo.13314155.

[9] "MySQL :: MySQL 8.0 Reference Manual :: 17.6.2.1 Clustered and Secondary Indexes." (Aug. 14, 2023), [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/innodb-index-types.html.

[10] M. Ray. "Clustered and nonclustered indexes - SQL Server." (Aug. 14, 2023), [Online]. Available: https://learn.microsoft.com/en-us/sql/relational-databases/indexes/clustered-and-nonclustered-indexes-described?view=sql-server-ver16 (visited on 08/23/2024).

[11] H. Sundar, D. Malhotra, and G. Biros, "Hyksort: A new variant of hypercube quicksort on distributed memory architectures," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13, Eugene, Oregon, USA: Association for Computing Machinery, 2013, pp. 293–302. DOI: 10.1145/2464996.2465442. [Online]. Available: https://doi.org/10.1145/2464996.2465442.

[12] B. Dong, S. Byna, and K. Wu, "SDS-sort: Scalable dynamic skew-aware parallel sorting," in *HPDC 2016 - Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, 2016. DOI: 10.1145/2907294.2907300.

[13] A. Rasmussen, G. Porter, M. Conley, H. V. Madhyastha, R. N. Mysore, A. Pucher, and A. Vahdat, "Tritonsort: A balanced large-scale sorting system," in *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Boston, MA: USENIX Association, Mar. 2011. [Online]. Available: https://www.usenix.org/conference/nsdi11/tritonsort-balanced-large-scale-sorting-system.

[14] J. Chou, M. Howison, B. Austin, K. Wu, J. Qiang, E. W. Bethel, A. Shoshani, O. Rübel, and R. D. Ryne, "Parallel index and query for large scale data analysis," in *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, 2011, pp. 1–11.

[15] P. Raju, V. Chidambaram, R. Kadekodi, and I. Abraham, "PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees," in *SOSP 2017 - Proceedings of the 26th ACM Symposium on Operating Systems Principles*, 2017. DOI: 10.1145/3132747.3132765.

[16] P. Ganesan, M. Bawa, and H. Garcia-Molina, "Online balancing of range-partitioned data with applications to peer-to-peer systems," in *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, ser. VLDB '04, Toronto, Canada: VLDB Endowment, 2004, pp. 444–455.

[17] I. Konstantinou, D. Tsoumakos, and N. Koziris, "Fast and cost-effective online load-balancing in distributed range-queriable systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 8, pp. 1350–1364, 2011. DOI: 10.1109/TPDS.2010.200.

[18] Y. Qiao, X. Chen, N. Zheng, J. Li, Y. Liu, and T. Zhang, "Closing the b+-tree vs. LSM-tree write amplification gap on modern storage hardware with built-in transparent compression," in *20th USENIX Conference on File and Storage Technologies (FAST 22)*, Santa Clara, CA: USENIX Association, Feb. 2022, pp. 69–82. [Online]. Available: https://www.usenix.org/conference/fast22/presentation/qiao.

[19] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 1, pp. 63–75, 1985.

[20] Q. Zheng, G. Amvrosiadis, S. Kadekodi, G. A. Gibson, C. D. Cranor, B. W. Settlemyer, G. Grider, and F. Guo, "Software-defined storage for fast trajectory queries using a deltaFS indexed massive directory," in *Proceedings of PDSW-DISCS 2017 - 2nd Joint International Workshop on Parallel Data Storage and Data Intensive Scalable Computing Systems - Held in conjunction with SC 2017: The International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017. DOI: 10.1145/3149393.3149398.

[21] R. B. Ross, G. Amvrosiadis, P. Carns, C. D. Cranor, M. Dorier, K. Harms, G. Ganger, G. Gibson, S. K. Gutierrez, R. Latham, *et al.*, "Mochi: Composing data services for high-performance computing environments," *Journal of Computer Science and Technology*, vol. 35, pp. 121–144, 2020.

[22] J. Soumagne, D. Kimpe, J. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. Ross, "Mercury: Enabling remote procedure call for high-performance computing," in *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, IEEE, 2013, pp. 1–8.

[23] R. Rabenseifner, "Optimization of collective reduction operations," in *International Conference on Computational Science*, Springer, 2004, pp. 1–9.

[24] R. Rabenseifner and J. L. Träff, "More efficient reduction algorithms for non-power-of-two number of processors in message-passing parallel systems," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, Springer, 2004, pp. 36–46.

[25] B. Vigil, "Trinity Advanced Technology System Overview," LA-UR-14-28143, 1160100, Oct. 20, 2014. DOI: 10.2172/1160100. [Online]. Available: https://www.osti.gov/servlets/purl/1160100/ (visited on 08/23/2024).

[26] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[27] *Aurora — argonne leadership computing facility*, https://www.alcf.anl.gov/aurora, Aug. 2024.

[28] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[29] D. Comer, "Ubiquitous b-tree," *ACM Computing Surveys (CSUR)*, vol. 11, no. 2, pp. 121–137, 1979.

[30] W. Zhong, C. Chen, X. Wu, and S. Jiang, "{Remix}: Efficient range query for lsm-trees," in *19th {USENIX} Conference on File and Storage Technologies ({FAST} 21)*, 2021, pp. 51–64.

[31] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, Q. Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf, "PreDatA - preparatory data analytics on peta-scale machines," in *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPS 10)*, 2010, pp. 1–12. DOI: 10.1109/IPDPS.2010.5470454.

[32] V. Vishwanath, M. Hereld, and M. E. Papka, "Toward simulation-time data analysis and i/o acceleration on leadership-class systems," in *Proceedings of the 2011 IEEE Symposium on Large Data Analysis and Visualization (LDAV 11)*, 2011, pp. 9–14. DOI: 10.1109/LDAV.2011.6092178.

[33] V. Vishwanath, M. Hereld, V. Morozov, and M. E. Papka, "Topology-aware data movement and staging for I/O acceleration on Blue Gene/P supercomputing systems," in *Proceedings of the 2011 International*

Conference for High Performance Computing, Networking, Storage, and Analysis (SC 11), 2011, pp. 1–11. DOI: 10.1145/2063384.2063409.

[34] R. A. Oldfield, G. D. Sjaardema, G. F. Lofstead II, and T. Kordenbrock, "Trilinos i/o support trios," *Sci. Program.*, vol. 20, no. 2, pp. 181–196, Apr. 2012. DOI: 10.1155/2012/842791.

[35] J. C. Bennett, H. Abbasi, P. T. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, P. Pebay, D. Thompson, H. Yu, F. Zhang, and J. Chen, "Combining in-situ and in-transit processing to enable extreme-scale scientific analysis," in *Proceedings of the 2012 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 12)*, 2012, pp. 1–9. DOI: 10.1109/SC.2012.31.

[36] J. Gu, S. Klasky, N. Podhorszki, J. Qiang, and K. Wu, "Querying Large Scientific Data Sets with Adaptable IO System ADIOS," in *Supercomputing Frontiers*, R. Yokota and W. Wu, Eds., Cham: Springer International Publishing, 2018, pp. 51–69. DOI: 10.1007/978-3-319-69953-0_4.

[37] K. Wu, "FastBit: An efficient indexing technology for accelerating data-intensive science," *Journal of Physics: Conference Series*, vol. 16, no. 1, p. 556, Jan. 2005. DOI: 10.1088/1742-6596/16/1/077. [Online]. Available: https://dx.doi.org/10.1088/1742-6596/16/1/077 (visited on 08/22/2024).

[38] W. Usher, X. Huang, S. Petruzza, S. Kumar, S. R. Slattery, S. T. Reeve, F. Wang, C. R. Johnson, and V. Pascucci, "Adaptive spatially aware i/o for multiresolution particle data layouts," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2021, pp. 547–556.

[39] M. Olma, M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, and A. Ailamaki, "Slalom: Coasting through raw data via adaptive partitioning and indexing," *Proceedings of the VLDB Endowment*, vol. 10, no. 10, pp. 1106–1117, 2017.

[40] S. Maroulis, N. Bikakis, G. Papastefanatos, P. Vassiliadis, and Y. Vassiliou, "Resource-aware adaptive indexing for in situ visual exploration and analytics," *The VLDB Journal*, vol. 32, no. 1, pp. 199–227, Jan. 1, 2023. DOI: 10.1007/s00778-022-00739-z. [Online]. Available: https://doi.org/10.1007/s00778-022-00739-z (visited on 05/23/2024).

[41] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: Separating keys from values in ssd-conscious storage," *ACM Transactions on Storage (TOS)*, vol. 13, no. 1, pp. 1–28, 2017.

[42] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo, "Surf: Practical range query filtering with fast succinct tries," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 323–336.

[43] O. Tange, "Gnu parallel - the command-line power tool," *;login: The USENIX Magazine*, vol. 36, no. 1, pp. 42–47, Feb. 2011. [Online]. Available: http://www.gnu.org/s/parallel.