Rottnest: Indexing Data Lakes for Search

Ziheng Wang*+, Conor Kennedy* Rain Jiang* Weston Pace Huayi Zhang, Alex Aiken
Sasha Krassovsky* *Stanford University* *Bytedance, Inc.* *LanceDB, Inc.* Chenyu Jiang, Wei Xu *Stanford University*
Stanford University, *Bytedance, Inc.*
Anthropic PBC

***Abstract*—Data lakes have become widely popular in managing enterprise data. Their widespread integration with query engines has allowed them to displace specialized data warehouses as the single source of truth for enterprise data. While the columnar storage format and block min-max indices allow query engines to achieve competitive performance on relational data analytics queries, they are not yet suitable for other search-oriented queries like full text and vector nearest neighbor search. We present Rottnest, a general system that builds additional lightweight indices on top of data lakes. We show that our system is more cost efficient compared to un-indexed data lakes or specialized databases across several orders of magnitude of total query loads and operating time horizons.**

I. Introduction

Data lakes have become mainstream in analytical data processing. They typically consist of Parquet files on object storage such as AWS S3 organized by a table format like Delta Lake, Iceberg or Hudi [1]–[5]. The Parquet files are queried on-demand with engines such as Trino, Databricks, Snowflake, or DuckDB [6]–[10]. For OLAP SQL workloads, the performance of such query engines is often better than specialized data warehouses [11], [12]. The widespread support of Parquet among data science and machine learning tools have made data lakes ideal for the analytical "single source of truth" in modern enterprise data stacks.

However, data lakes are still unsuitable for emerging workloads that are **search-oriented**. These workloads, such as high-cardinality time series, text, and embedding analytics typically need to quickly drill down to a very small subset of the data and perform complex aggregations on this subset. In most cases, the minimal indexing available in current Parquet-based data lakes do not allow efficient evaluation of the filter conditions (e.g. UUID match, text regex, approximate nearest neighbors), making these workloads inefficient for query engines like SparkSQL or Trino [6], [13]. As a result, the lakehouse paradigm breaks for these workloads – organizations have to duplicate their data to a specialized data management system like Clickhouse, ElasticSearch or Qdrant to perform efficient search analytics, as shown in Figure 1.

Modern data lakes store data primarily as immutable Parquet files in object storage, optimized for large-scale analytics. Data lakes like Delta Lake and Apache Iceberg added transactional support to data lakes by maintaining lightweight metadata files alongside the Parquet files [4], [5], [14]. These systems achieve ACID properties efficiently through a combination of metadata logging and background compaction operations - transaction logs track file-level changes while periodic compactions merge small files to optimize read performance. Building on these principles, we explore if we could support novel search workloads through additional lightweight index files on top of transactional data lakes. We build Rottnest, a system that augments data lakes with external indices resident on object storage for workloads such as high cardinality analytics, full text search, and vector embedding search.

Although individual components of Rottnest's novel architecture such as inverted indices and LSM-style compaction are well studied, our key insight is that combining them enables efficient search over data lakes with low query cost and storage overhead. Rottnest's architecture is enabled by a novel consistent-on-demand indexing protocol that can be bolted on to any data lake and object storage backend. We also propose a novel evaluation methodology that demonstrates Rottnest's effectiveness across a wide range of operational scenarios.

Just as data lakes provide "good enough" transactional semantics for offline big data workloads without replacing OLTP databases, Rottnest provides "good enough" search capabilities for offline workloads without aiming to replace specialized search systems in latency-sensitive online serving. We believe it can be the most compelling option for most offline workloads such as historical log analytics and LLM pretraining data exploration.

In summary, we make three contributions:

1) A novel bolt-on, consistent-on-demand protocol for concurrent index maintenance. Our protocol, with formal correctness proofs, enables consistent indexing even as the underlying data lake undergoes operations like compaction and deletion. Unlike existing alternatives, it only relies on strong read-after-write consistency, making it compatible with all major cloud providers.
2) A new system architecture to support search-oriented workloads over data lakes using our lazy indexing protocol that combines in-situ querying of Parquet files directly and object-store native indices to deliver low query cost and low storage overhead.
3) A novel evaluation framework using physics-inspired *phase diagrams* to analyze total cost of ownership (TCO) across different query loads and time horizons. This methodology reveals insights about system trade-offs that would be difficult to obtain through traditional performance evaluations, and can be applied to evaluate other cloud data systems where TCO is the relevant metric.

* Equal contribution. + Currently at Anthropic PBC.

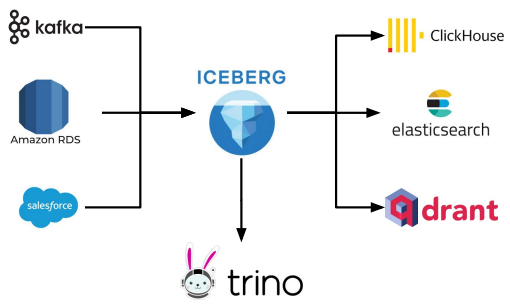

Fig. 1. Typical enterprise data stack. For workloads where directly querying the data lake with a query engine like Trino is too inefficient, a specialized system like Clickhouse, ElasticSearch or Qdrant is used.

II. Motivations

In this section, we describe the access characteristics of Parquet-based data lakes to explain why search-based workloads can be challenging. We also describe existing approaches commonly taken to address those challenges.

A. Parquet Data Lakes on Object Storage

Organizations have started using object-storage-based data lakes based on Parquet files as the single source of truth for enterprise data, due to their wide ecosystem integrations, compelling cost characteristics and transactional semantics. As depicted in Figure 1, operational data is first ETLed into the data lake from OLTP databases, CRMs, and custom connectors. Once the data lands in the data lake, a query engine such as Trino or SparkSQL is used to query the data lake directly for analytical workloads.

Parquet files are well optimized for OLAP workloads against object storage. A Parquet file can consist of multiple row groups, where each row group in turn consists of a collection of column chunks. A column chunk in a row group consists of all the values of a column in the row range corresponding to the row group. Min-max statistics on the column chunks assist query engines in performing predicate pushdown, reading only column chunks necessary for a particular query. These column chunks, which range from several to hundreds of MB in size, can be downloaded in parallel via byte-range requests against object storage. With VM instance bandwidth reaching 100Gbps in major public clouds, analytics on Parquet files residing in object storage has reached interactive speeds.

B. Search Workloads

While data lakes are designed for SQL OLAP use cases, they still fall short in emerging search-oriented workloads, such as:

- **High-cardinality filtering**, such as filtering on a UUID column. Examples include observability workloads (filtering by Kubernetes pod name) and blockchain analytics (filtering by transaction hash).
- **Substring search**, such as log search or text analytics. For example, to detect if evaluation datasets are leaked in the pretraining corpus, one could perform a substring search against the training records, which might be stored as a text column in a data lake.
- **Vector embedding search**, i.e. approximate nearest neighbor search. Examples include retrieval-augmented-generation (RAG) and recommendation systems.

While query engines can efficiently use Parquet-based data lakes for OLAP workloads, they struggle to do so on these new workloads. There are two main issues:

1) **Useless indices.** In search workloads, it is often infeasible to keep the data sorted based on a high-cardinality column and maintain insertion performance. For example, time series data such as IT monitoring and blockchain logs are naturally sorted by timestamp, and sorting them based on UUID tags amount to an extremely expensive transpose. Natural sort-orders do not exist for text or vector embeddings. These two factors render min-max indices on column chunks useless for search workloads.
2) **Read granularity.** Even if an appropriate index exists, column chunks corresponding to text or binary data types typically tend to be tens to hundreds of MBs in size due to Parquet's design (further explored in Section V-A). This means for highly selective search queries, current query engines have to retrieve many MBs of data from object storage just to return a result that is tens of bytes.

"Lakehouse" query engines today like Spark, Trino, or DuckDB today typically have to perform expensive full-column scans which read GBs to TBs of data to perform search queries, making them very expensive.

C. Existing Approaches

Today practitioners commonly employ two approaches to tackle such search workloads, copy data (ETL) and brute force scan. We also mention an emerging third approach, which is new data formats.

1) Copy Data: Practitioners today commonly copy data from the data lake back into specialized systems tailor-made for their workload, such as Clickhouse [15], ElasticSearch [16], vector databases like Qdrant [17] (Figure 1). While this approach guarantees the best performance for those queries by leveraging specialized systems, it largely negates the benefits of the data lake for the data in question. These systems tend to be compute-storage integrated and are expensive to manage. In addition, the data pipelines duplicate data and reintroduce data consistency and staleness issues that data lakes seek to address in the first place.

While we believe that this is the best approach if the search queries are frequent or require strict latency SLAs, e.g. a search engine like Google, we observe that many important workloads do not have such requirements. For example, an LLM pretraining data exploration system might serve only up to 100 internal users while query latencies up to a minute are acceptable. Similarly, a log management system might serve only a particular SRE team where query latencies up to a few seconds are acceptable.

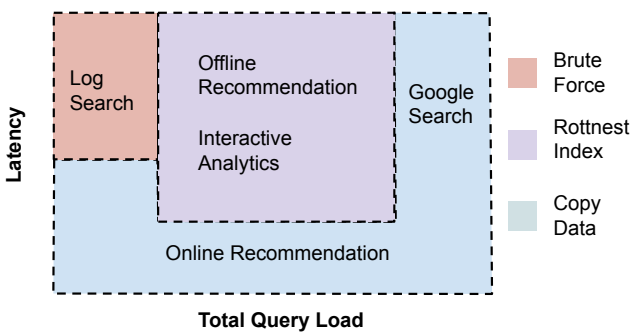

Fig. 2. The most economical approach given latency and throughput requirements of the application. Some example applications are shown.

2) Brute Force Scan: Another popular approach is to simply pay the high IO and compute cost of these queries by horizontally scaling the query engine, such as with Spark SQL [13], [18]. Cloud vendors are especially incentivized to take this approach as they typically charge based on the amount of data scanned. For example, AWS Athena has been heavily used by cybersecurity professionals to query logs, with a pricing model that reflects the brute-force scanning approach.

This approach is the simplest and most economical if querying is highly infrequent, since it incurs no upfront indexing or storage costs. However, it simply defers those costs to query time. While we believe this may be the best approach for workloads that might not ever read most of the data (e.g. log analytics), if most data will be read at least a couple of times over its lifetime, this quickly becomes expensive. In addition, this approaches incurs a minimum latency of typically a few hundred milliseconds due to its need to spin up tasks on demand and access object storage, making it unsuitable for worklaods with strict latency SLAs.

3) New File Format: A third approach is to replace Parquet files with new file formats, such as Lance and Nimble [19], [20]. While these new file formats are purposefully designed to excel at these new search-style workloads, they have not yet achieved the widespread ecosystem integrations that made Parquet appealing in the first place. For example, all three major data lake formats only support Parquet today [2], [3], [5]. As a result, it is impractical today for most organizations to adopt them as the ground truth data store. This means in practice organizations still have to keep data in Parquet, making this no different than the copy-data approach.

Another similar approach is recent modifications to the Parquet file itself, for example column indices and bloom filters [21], [22]. While these approaches could be somewhat effective in some search queries that involve high cardinality tags, they do not help for text or vector searches. More importantly, they assume a Parquet writer that generates these additional indexing structures, which in practice is quite rare.

D. Motivation for a New Approach

The limitations of existing approaches suggest the need for a new design that simultaneously satisfies:

- **Lower indexing and storage cost** compared to copying data into dedicated indices that require expensive persistent RAM/disk resources.
- **Cheaper querying** compared to the brute force approach.

Satisfying these two objectives would provide the most economical solution for a large class of offline or human-interactive workloads with medium total query load, some examples of which are shown in Figure 2. Our objective of easy integration with existing data lakes to reduce data silos leads us to two more design goals:

- **Non-interference guarantee** with existing data lake operations. As the data lake lives at the center of an organization's data architecture, the system must not impact the performance or complexity of existing data lake ingestion, maintenance, or read workloads.
- **Broad compatibility** with all major cloud providers, which might offer varying consistency properties for their object storage offerings.

III. Rottnest Search Indices

Rottnest addresses these four objectives by ***maintaining lightweight index files on top of data lakes.*** Inspired by how data lakes like Delta Lake or Iceberg achieve ACID properties on top of Parquet files in object storage by maintaining log files [3], [4], [23], we propose a novel lazy indexing protocol that constructs secondary indices on data lakes which satisfies the latter two design goals.

Instead of updating indices upon write, Rottnest maintains consistency with the underlying data lake on-demand. Rottnest index maintenance and querying can be performed completely independently of other data lake operations. This approach shares similarities with Postgres concurrent indices and background compactions in LSM-based databases like RocksDB [24], [25]. Like RocksDB, this means the newest data often remains unindexed and requires full scanning during queries. While our protocol is conceptually similar to Azure Hyperspace [26], a key difference lies in the consistency primitives used. Hyperspace relies on atomic rename operations, which are not universally supported across major cloud providers like AWS S3. In contrast, Rottnest's protocol requires only strong read-after-write consistency, making it compatible with all major cloud object stores.

Building on our novel lazy indexing protocol, Rottnest's overall search architecture satisfies our first two design goals: minimizing both index construction/storage costs and query costs. Optimizing along these two objectives allows Rottnest to maximize the area of the purple-shaded area of Figure 2 where it wins over the brute force and copy data approaches. In the red-shaded area, brute force scanning requires no indexing cost but incurs high search costs - lower index costs help Rottnest reach its break-even point with fewer queries. In the blue-shaded area, dedicated systems like ElasticSearch require high upfront costs to store indices in expensive persistent SSDs/memory but offer cheap searches - lower query costs allow Rottnest to remain competitive at higher query loads while also meeting stricter latency requirements.

To satisfy these dual requirements, our search architecture combines two key engineering innovations: in-situ querying of Parquet files to minimize index storage costs, and object-storage-native index structures optimized for low-latency access to minimize query costs. These innovations are put together in a novel way which enables, for the first time, full-text and vector search on Parquet-based data lakes with latencies of just a few seconds, without requiring persistent memory or disk resources.

To reduce indexing and storage cost, Rottnest does not store a copy of the raw data in the index, opting instead to query the data *in situ* in the Parquet files, inspired by previous work such as NoDB [27]. Assuming the index structure is smaller than the compressed raw data (which is typically the case), this drastically lowers the index storage overhead. While recent work has suggested that Parquet files have efficiency issues that limit their performance on highly selective search queries [28], [29], we show that contrary to popular wisdom, most of these issues can be circumvented with a custom Parquet reader implementation described in Section V-A, leading to search performance that rivals custom formats like Lance [19].

To eliminate the need for disk caching to further lower storage cost, Rottnest stores and accesses index files directly on object storage. While the indices are accessed randomly via S3 byte range GET requests, we optimize latency through a technique we call componentization, based on principles from cache- and disk-efficient algorithms [30], [31].

To rigorously analyze the system's performance characteristics, we introduce a novel methodology using phase diagrams that quantitatively maps the performance boundaries where Rottnest outperforms both brute-force scanning and dedicated search systems (shown qualitatively as the purple region in Figure 2). While existing literature on cost comparisons of cloud data systems typically focus on aspects like storage cost and query cost separately, or the total cost of executing a fixed benchmark set of queries over a fixed cluster configuration [32]–[34], our TCO framework jointly considers index construction cost, storage overhead, and per-query cost over varying time horizons and query volumes to identify optimal operating regions for different approaches. This analytical framework could also be used to evaluate future data lake indexing systems.

In the following three sections, we will explore our novel contributions in turn: index protocol (IV), optimized access architecture (V) and TCO evaluation (VI).

IV. Index Protocol

Similar to delta-rs [35], the Rust-based client of Delta Lake, Rottnest is designed as an embedded Rust library with index management APIs in Python. The Rottnest client library supports four APIs: `index`, `search`, `compact` and `vacuum`. Of these, `index`, `compact`, and `vacuum` mutate the Rottnest index, maintaining two invariants that together guarantee correct `search`:

- **Existence**: all indexed files referenced in the metadata table are present in the object storage bucket.

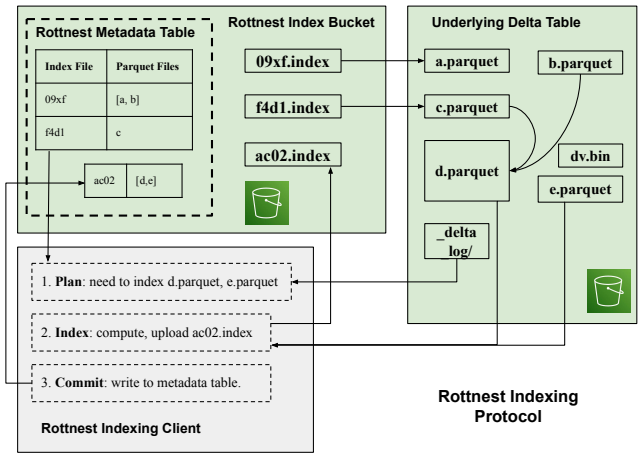

Fig. 3. Rottnest Indexing Protocol. Since the last `index` call b.parquet and c.parquet has been compacted into d.parquet, and an update was written with the update file e.parquet and deletion vector dv.bin.

- **Consistency**: an index file correctly indexes the associated Parquet files if they still exist.

A. Building a Rottnest Index

The `index` API can be called from any VM instance or serverless function with access to the underlying data lake to keep a Rottnest index at an object storage bucket `index_dir` up to date with the current *snapshot* of the data lake table.[1] The design would be simple for append-only data lakes where Parquet files can only be added: when `index` is called, build a new index file covering the new Parquet files. However, data lakes support operations that may invalidate existing Parquet files (e.g., compactions, vacuums, Z-order, and CRUD operations) as well as produce custom files such as *deletion vectors* which record individual rows of a table that have been deleted [36], [37]. Because our indices point to physical locations, index files may be invalidated by such operations.

We propose a simple protocol to ensure consistency in the face of these different data lake operations: index all new Parquet files written to the data lake, regardless of whether they result from insertions, compactions or updates. While searching, search only index files that include physical locations included in the snapshot. An index file might also map a query term to physical locations not in the snapshot – such locations are filtered out during the search.

To facilitate this process, Rottnest keeps track of the list of Parquet files it has already indexed in the Rottnest metadata table, which is implemented as a Delta Lake table itself resident on object storage. In principle, any transactional data store, e.g. Postgres, can be used for this purpose.

The indexing works as follows (example in Figure 3):

1) **Plan**: Look at the manifest list of the latest data lake snapshot and find the Parquet files that are in the current snapshot but not yet indexed. Rottnest only indexes new

[1]Data lakes support time travel; a snapshot is a point-in-time copy of the data represented by a list of Parquet files in the snapshot.

data files (d, e.parquet in Figure 4) and not deletion files (dv.bin).

2) **Index**: Rottnest proceeds to build an index file (ac02.index) that covers all the new Parquet files, and uploads it to `index_dir`.[2] If, for some reason, a file is no longer available to read during the indexing process, e.g. due to garbage collection of the data lake, the index API aborts and needs to be retried.
3) **Commit**: After the new index file has been uploaded to object storage, the indexer inserts a record into the Rottnest metadata table transactionally. Note that the metadata table shown in Figure 3 is simplified, in practice other metadata information such as total number of rows indexed and index timestamp can be recorded as well.
4) **Timeout**: If the index operation is not completed within a set timeout, it will abort and needs to be retried. The timeout is critical for garbage collection, as described later in `vacuum`.

We do not require all index files present in the Rottnest index bucket to have an associated entry in the metadata table, which might occur if the indexer fails during commit. These index files will be garbage collected separately, described Section IV-C. We also do not require all Parquet files referenced by index files to still be active in the underlying data lake. Indeed, it is expected that some index files might cover Parquet files removed by compactions.

Although the indexing API is internally parallel, it should not be called on the same table column across multiple processes. While doing so will not violate any safety guarantees, the same Parquet files would be needlessly re-indexed multiple times.

One might argue that indexing every new Parquet file in the data lake is inefficient, as new Parquet files written by special processes such as Z-order or compaction could be more efficiently indexed by simply remapping the existing posting list values without recomputing the entire index. However, this procedure is significantly more complex with dangerous pitfalls: e.g. the original Parquet files that were compacted might have been removed, making the remapping impossible.

B. Searching a Rottnest Index

Rottnest's client library offers a `search` API, which can be called on any process with access to the data lake and the Rottnest index at `index_dir`, similar to `index`. Rottnest search queries search a specified data lake snapshot to retrieve top-K results.[3] The search procedure follows these steps, with an example illustrated in Figure 4:

1) **Plan**: Rottnest first queries the data lake for the manifest list for the specified snapshot, which contains a list of Parquet files that make up the snapshot along with potential deletion vector files. Rottnest queries the metadata table in `index_dir` to determine which index files cover the Parquet files and which Parquet files have no index and must be brute-force scanned. In Figure 4, 09xf and ac02.index must be queried.
2) **Query Index**: Each Rottnest index file is queried independently in parallel for physical locations in the underlying Parquet files. The index files are queried on object storage directly, with optimization techniques described in Section V-B. For example, 09xf.index might return [(a.parquet, 10), (a.parquet, 20)], where 10 and 20 denote locations in the Parquet file. Rottnest indices are allowed to return false positives (e.g. bloom filter), so the top-K filter is not applied at this stage. Instead, we just filter out locations that correspond to Parquet files not in the specified `snapshot`.
3) **In-situ Probing**: The physical locations in the Parquet files are downloaded and scanned with the actual predicate to filter out false positives. Rottnest efficiently random accesses Parquet files, explained in more detail in Section V-A. Deletion files, i.e. dv.bin, are applied if they exist. The unindexed Parquet files are only scanned if the filtered results are not sufficient to satisfy an exact-match top K query or for a scoring query, which must rank all data items.

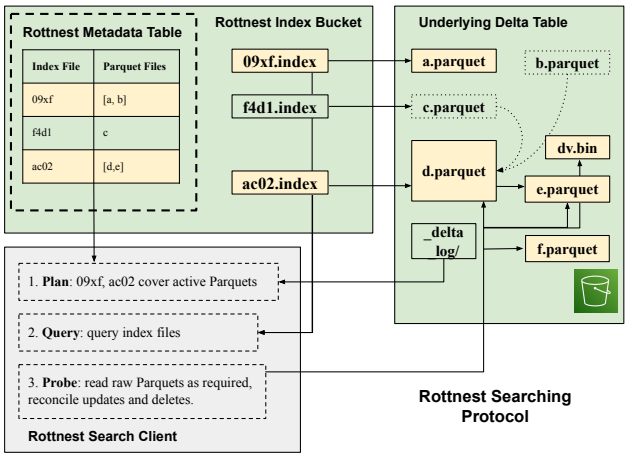

Fig. 4. Rottnest Search Protocol based on the running example in Figure 3. Assume that after the `index` call, f.parquet is added to the table and is un-indexed.

The correctness of this procedure follows from the two invariants. No row in the data lake will be "missed" since it is either covered by an index file or a Parquet file that is exhaustively scanned. Different from the `index` API, the `search` API is read-only for the data lake and the Rottnest index. It is meant to be called in parallel by independent processes and concurrently with the `index`, `compaction` and `vacuum` APIs described in the next section.

C. Index Maintenance

To avoid querying many small index files as the data grows, Rottnest supports compacting indices similar to log-structured

[2]Some types of indexes (such as vector indexes) might have a minimum size limit. If the total number of rows in the new files falls below this limit, the indexing will be aborted in favor of brute force scan.

[3]Top-K could have different meaning for exact match queries like regex (any K that satisfy predicate) and scoring queries like vector search (top K ranked based on score).

merged tree (LSM) mechanisms employed in databases such as RocksDB and Clickhouse [15], [24]. Multiple small index files are compacted into larger files and older index files can be garbage collected. In Rottnest, the indices are compacted independently of the data maintenance process of the underlying data lake, which might perform its own LSM-style compactions. Similar to `index`, Rottnest's client library supports a `compact` API, which can be run from any server that has access to the object storage bucket. It proceeds in three steps:

1) **Plan:** determine which index files to merge. In general, index files that cover small numbers of Parquet files or rows can be merged into larger ones, while it maybe less important (and more expensive) to merge indices that already cover a large number of files. The default behavior of Rottnest is to merge index files that is smaller than a configurable threshold in a bin-packing strategy.
2) **Merge:** build the merged index files. This step could be computationally intensive and might require reading the raw Parquet files. After the new index files are built, upload them to `index_dir`.
3) **Commit:** insert metadata about the merged index files into the metadata table.

It is important to note that the compaction process does not consult the log of the data lake at all, and is completely decoupled from the compaction process of the data lake itself. How exactly multiple index files are combined into one depends on the index type, some examples of which will be given in Section V-C.

Similar to how compaction works in data lakes, Rottnest index compaction does *not* delete index files, which is the responsibility of a separate garbage collection process, commonly referred to as vacuum. There are multiple reasons an index file may be eligible for garbage collection:

- It was written by a failed indexer or compactor before a successful commit to the metadata table.
- It has undergone compaction, i.e. a new index file covers the Parquet files that this file covers.
- It points to Parquet files that are no longer part of a supported snapshot of the underlying data lake.

To physically delete those files, the Rottnest client library provides the vacuum API, which proceeds in three steps:

1) **Plan:** determine which index files in the metadata table to keep based on `snapshot_id`. Rottnest currently uses a simple heuristic: it first computes all Parquet files included in all the snapshots past `snapshot_id`. Then it greedily selects index files that cover the most number of active Parquet files. The procedure stops when the number of covered Parquet files cannot be increased. This procedure maximizes the number of covered Parquet files while heuristically minimizing the number of index files.
2) **Commit:** Delete the rows in the metadata table corresponding to index files that are no longer necessary as determined by the last step.
3) **Remove:** Physically remove the index files from object storage that are no longer in the metadata table *and older than the index timeout*. Removing these "invisible" files require an expensive LIST operation against `index_dir`, which is acceptable because `vacuum` calls are not expected to be frequent or real-time.

We remove after commit in `vacuum` instead of commit after upload in `index` and `compact`. This ensures index files included in the metadata table are physically present to preserve the first invariant.

Vacuum is critical in reducing the storage overhead of the Rottnest index after index compaction. In addition, vacuum reduces index size also after operations on the underlying data lake. For example, if a compaction occurred in the underlying data lake, data in old Parquet files will be copied to new Parquet files. While new index files will be built for the new Parquet files, the old index files pointing to the old Parquet files will be deleted upon vacuum.

It is critical that vacuum uses the timestamps associated with objects to only remove objects older than the `index` timeout, since from its perspective there is no difference between index files written but not yet committed and index files that were written but failed to commit. Since the `index` operation has a timeout, `vacuum` knows files older than this timeout are in the second category and can be removed. Note that this timeout is against the object store's clock, which is valid because modern object stores provide strong consistency, and thus have a single global clock [38].

D. Invariants Proof of Correctness

Due to the loose synchronization between the index files, the underlying data lake, and the processes that modify them, Rottnest's protocols are carefully designed to ensure that data is either indexed correctly, or not at all, by maintaining the existence and consistency invariants. If M is the set of files referenced by the metadata table, and B is the set of files in the bucket, the following holds:

Lemma 1. ***Existence****: all indexed files referenced in the metadata table are present in the object storage bucket (i.e.* $\forall f \in M : f \in B$*)*

Proof. We prove the result by induction. Initially, $M = \varnothing$ and the invariant trivially holds.

For the inductive step, assume the invariant holds before launching some number of index-modifying processes (either `index`, `compact`, or `vacuum`). There are three states these processes can be in: `before_upload`, `before_commit`, and `during_delete`. First, notice that the indexing and compaction processes follow the same pattern of plan, upload, and commit. For both indexing and compaction in the `before_upload` state, both M and B are unmodified, so the invariant holds. In the `before_commit` state, B can only have grown with the new file f_{new}, and so $\forall f \in M : f \in B \cup \{f_{new}\}$ is true. Lastly, the commit will update M to contain f_{new}, and so

$\forall f \in M \cup \{f_{new}\} : f \in B \cup \{f_{new}\}$. Note that concurrent updates do not change the nature of the proof, since updates to M are transactional and files uploaded to B are owned exclusively by the process.

For `vacuum`, suppose we decide to delete some files F. First note that a concurrent indexing or compaction operation may have introduced some files to B but not M. Since these operations are guaranteed to take less time than the global timeout (they abort otherwise), we know that it is not safe to delete files younger than the timeout, and any such files are skipped by the vacuum process. These files therefore cannot be in F, and so will never be deleted before they are written to M. By the induction hypothesis, in the `before_commit` state, the invariant holds. After transitioning to the `during_delete` state, M is updated to $M \setminus F$; since M shrank, $\forall f \in M \setminus F : f \in B$ holds. Following the `during_delete` state, B is updated to $B \setminus F$ and we have $\forall f \in M \setminus F : f \in B \setminus F$. □

Lemma 2. ***Consistency****: an index file correctly indexes the associated Parquet files if they still exist. In other words, let d_f be the associated data file for index file f. Then $\forall f \in B$: $\neg exists(d_f) \vee indexes(f, d_f)$.*

Proof. Take an arbitrary Rottnest file $f \in B$. Once f is built, it correctly indexes d_f. Since both Rottnest and underlying data files are both immutable, the $indexes(f, d_f)$ will always be true unless either f or d_f is deleted. If f is deleted, then $f \neg \in B$, so the invariant holds. If d_f is deleted, then $\neg exists(d_f)$ is true, and so the invariant also holds. □

While the proof relies on reasoning about system states, similar to the design of Hyperspace, our approach fundamentally differs in how these states are managed. While Hyperspace uses atomic rename operations to switch between state files representing transitioning and stable states [26], Rottnest achieves the same guarantees through a combination of immutable files with append-only metadata updates, transactional updates to the metadata table and careful use of timestamps for failure detection. This design eliminates the need for atomic rename operations, which are not universally supported across cloud providers, while maintaining the same strong consistency guarantees through primitives commonly available in all major object stores.

V. Index Implementation

We have so far focused on maintaining consistency between indices and physical data locations in a data lake. We now present Rottnest's novel architecture that enables efficient search directly over data lake formats through two key components working in concert:

1) An in-situ querying mechanism that enables efficient access to native formats like Parquet to avoid storing a copy of the data in a different format.
2) A componentization strategy that optimizes index layouts for fast cold-access on object storage to avoid persistent disk/RAM caching for index data structures.

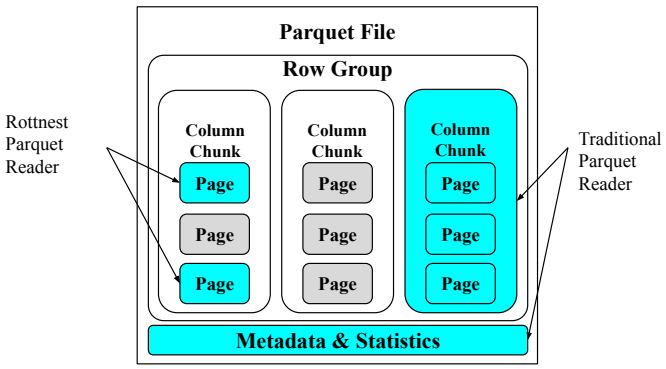

Fig. 5. Traditional Parquet readers read entire column chunks. Rottnest's reader reads individual pages, and notably bypasses the file metadata.

Combined, these techniques enable a fully object-storage-based architecture while maintaining low query latencies, satisfying both cheap search and storage as described in Section III. We will demonstrate that this architecture enables us to perform efficient search queries over data lakes on object storage with latencies previously thought impractical.

A. In-situ Querying

It might seem surprising from our discussion in Section II-B that keeping the original data in Parquet on object storage could lead to efficient querying. A typical layout of a Parquet file is shown in Figure 5. Since Parquet writers typically write 128MB row groups, and we are interested in indexing wide columns (vectors and text), the row group's space is dominated by our column's column chunk (typically 100 of 128 MB goes to our column).[4] For highly selective search queries, reading and decompressing 100MB of data from object storage to retrieve just a few rows is not efficient.

To mitigate this issue, we observe that the minimal access granularity in a Parquet file is actually a *data page*, whose size is independent of the row group size. Typically, the physical size of a data page is equal to the compressed size of 1MB of raw data, which is around a few hundreds KBs for text or vector data types. We will show in Section VII-C that reading at this granularity is as efficient as using custom data formats like Lance [19].

Similar to NoDB which maintains *position zone maps* on raw data [27], Rottnest maintains *page tables* that associate a unique ID for each data page to the offsets and sizes of the data page. Rottnest's indices are built at the granularity of these pages. In other words, the posting lists do not point to individual rows but to data pages. While this adds additional filtering work at query time and might complicate the design of indices which rely on posting list intersections (e.g. BM25), it significantly reduces the index storage footprint and speeds up index construction.

[4]This is an inherent flaw in Parquet's design, because all column chunks in a row group must have the same number of rows.

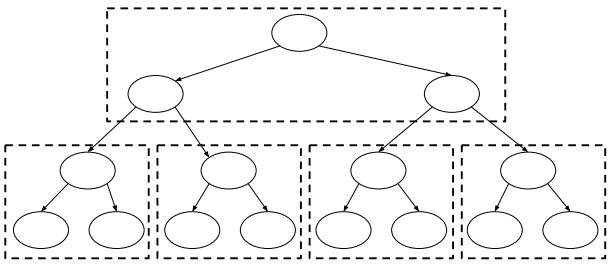

Fig. 6. Breaking a BST into serializable components. Each query must read the "root" component and one leaf component. The other three leaf components are not read. The memory-mapped approach for each query would have required four sequential requests, while this approach requires only two.

B. Optimizing for Object Storage Accesses

Object storage is a high-latency but high-throughput storage medium that favors large sequential range requests issued in parallel (access *width*) over sequences of small requests (access *depth*) [4], [39], [40].

The straightforward approach is to take the in-memory data structure, serialize and compress it, and then upload it to object storage. To query the data structure, simply download and decompress it in memory. Compression significantly reduces both storage costs and read amplification, with IO savings typically outweighing decompression overhead. While this approach leads to large sequential reads and adequate parallelism assuming simultaneous querying of multiple index files, it can be wasteful for random-access indices where most of the data structure is not accessed.

An opposite approach could consist of "memory-mapping" the data structure to object storage, where memory accesses are directly translated to object store requests. This approach has the benefit of reading only the bytes required, however it could lead to long chains of dependent object-storage requests. In addition, it foregoes the compression benefits offered by the former approach.

An intermediate approach that we employ, which we term **componentization**, builds on classical cache and disk blocking techniques [30], [31]. We break up a data structure into several components to optimize access patterns. However, unlike traditional blocking which focused purely on locality, componentization must additionally consider compression and object store access patterns. Each component is compressed and concatenated to form the index file, which also contains an offset array of the location of each component. Each data structure access only reads the components it needs, reducing the total number of dependent requests because each component could capture multiple requests. An example of this approach applied to a binary search tree is shown in Figure 6.

The key observation that enables componentization is that most data structures employed in indices have some degree of "access locality". This approach would not work if after an access into the data structure, the next access address is completely random or data dependent. In these cases, alternative data structures might have to be considered.

C. Example Rottnest Indices

We now describe how to apply the componentization principle to build Rottnest indices for the search workloads described in Section II-B: UUID, substring and vector search, where current data lake query engines struggle. In addition, we explain how to support efficient merging of those indices, required for compaction.

1) High-cardinality UUIDs: We design an index to facilitate fast exact matching of UUIDs via a binary trie, where each UUID corresponds to a path in the trie. In accordance to the design principles in Section V-B, we break the binary trie into components similar to Figure 6. To save space, the binary trie only indexes for each UUID as many bits as required to uniquely identify it, which corresponds to its *longest common prefix* (LCP). However, an index can be merged with others, so we do not know the LCP of any UUID *a priori*. We thus index up to 8 extra bits of the LCP for each UUID while allowing leaf nodes to map to multiple UUIDs in case that is not yet enough. We employ an additional optimization by replacing the first 8 layers of the trie with a look up table to reduce the number of sequential requests.

2) Exact Substring Matching: We employ the FM-index based on the Burrows-Wheeler Transform [41], [42] with a sampled suffix array. The data structures are adapted to object storage with the componentization approach described in Section V-B. To merge indices, we employ the technique described in [43] with bounded interleave iterations.

3) Vector ANN Index: Vector indices are typically graph based (e.g. Vamana, HNSW) or clustering based (IVF-PQ) [44]–[46]. While graph-based approaches like HNSW have been gaining in popularity, they typically require fewer distance computations but more sequential data accesses (due to the random traversal of the graph) to achieve the same recall target compared to centroid-based approaches like IVF-PQ [44], [46]. This is very beneficial for disk/RAM scenarios where the distance computations can dominate search cost, but can be harmful when IO access costs dominate, as is the case with object storage. As a result, popular object-storage based vector databases typically rely on centroid-based indices like IVF-PQ or SPFresh [47]–[49]. We choose to use IVF-PQ index in Rottnest, and tune the *nprobe* and *refine* parameters to hit different recall targets: *nprobe* controls how many centroids to query and *refine* controls how many full precision vectors to download for reranking.

VI. Evaluating Rottnest

In Figure 2, we presented intuition for how Rottnest compares to two other approaches: **copy data** into a dedicated system or **brute force** scan the data lake. We now follow up with a much more quantitative evaluation framework that seeks to answer under what exact conditions is one approach better than another to ground our evaluation results.

All three approaches have drastically different storage and query cost characteristics. While prior comparison frameworks have focused on comparing indexing cost, storage footprint and query cost separately [32], [34], or the total cost of running

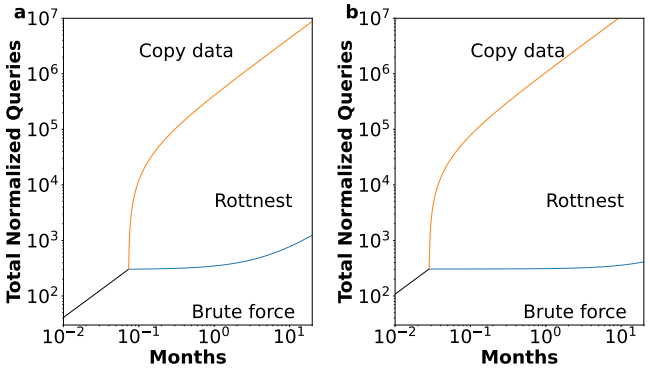

Fig. 7. Phase change diagrams for a) Substring search and b) UUID search. Note log-log axes. Explained in Section VII-B1.

a fixed benchmark of queries [33], we believe practitioners need a more intuitive way to choose between approaches based on their expected workload. Our phase diagram framework provides this by visually showing which approach minimizes total cost of ownership under different operating conditions.

We assume we are operating in a regime where we are not latency constrained in Figure 2. While the **minimum latency threshold** of an approach is an important metric, a better way to evaluate these three approaches beyond the threshold is using cost: the **total cost of ownership (TCO)** of the system under a fixed query load and operating duration. In this situation, comparing query latency is less appropriate since the brute force approach can trivially reduce query latency by scaling search horizontally.[5]

Since all three approaches can be more efficient if the query only has to search part of the data lake, e.g. due to a filter on a structured attribute like timestamp, we consider the cost per **normalized query**, where the brute force approach must scan the entire dataset. Note that Rottnest can leverage structured filters by building indices on different partitions of the data clustered by the structured attribute [50], [51].

To compare the TCO of these three approaches, we can plot a quadrant with the total number of normalized queries on the y-axis and the number of months we are operating the system on the x-axis. For a particular point (months, queries) on this quadrant, we can estimate the TCO of each of the three approaches as $index\_cost + cost\_per\_month \times months + cost\_per\_query \times queries$:

- **Copying data** into an dedicated system typically incurs a high cost per month for a cluster of always-on servers. On the other hand, the indexing and query cost can be folded into this constant monthly operating cost. $TCO = cpm\_i \times months$, where cpm_i is the cost per month.
- **Brute force** incurs no indexing cost and a very low cost per month (S3 storage of compressed data). However it has very high $cost\_per\_query$: $TCO = cpm\_bf \times months + cpq\_bf \times queries$, where cpm_bf and cpq_bf represent the cost per month and per query respectively.
- **Rottnest indices** incur a one-time index cost, relatively higher cost per month (to store and maintain the index structures) but a much lower $cost\_per\_query$ compared to brute force. $TCO = ic\_r + cpm\_r \times months + cpq\_r \times queries$, where ic_r denotes the index cost.

Using this model we can plot a *phase change*[6] diagram that indicates the most economical solution for each point in the quadrant, a couple examples of which are shown in Figure 7. The lines indicate boundaries between regions where a different approach is optimal in terms of TCO. This phase change graph allows a practitioner to easily figure out the most economical approach based on the estimated usage characteristics of the search workload. For example, at 10 months and 10^4 total normalized queries, Rottnest is the most cost efficient approach for substring search.

Rottnest typically becomes most economical when:

1) The dataset needs to be queried more than some minimum number of times to amortize the index cost.
2) The number of queries is large enough that brute force is too expensive but not large enough to justify copying the data into an always-on index.

The parameters cpm_i, cpm_bf, cpq_bf, ic_r, cpm_r and cpq_r, which directly determine the shape of the graph, are generally dependent on the search workload and data distribution. We will discuss how changing them affects the phase change diagram in Section VII-D.

VII. Results

We now apply this evaluation framework to the Rottnest indices we constructed for three example applications: UUID, substring and vector search. For substring search, we use the C4 dataset from FineWeb [52], containing 0.8 trillion characters (304GB compressed) of web crawl data that reflects real-world text search patterns. Substring search is commonly used in LLM data curation to retrieve data related to a particular domain or detect copyright violation. For UUID search, we generated 2 billion 128-byte hashes to mirror enterprise workloads involving unique identifier lookups common in observability queries. Vector search evaluation uses the SIFT dataset [53] of 1 billion 128-dimensional vectors, a commonly-used industry-standard benchmark.

For the copy data approach, we use AWS OpenSearch for substring and UUID search (cluster of 3 r6g.large.search instances with data in EBS), and LanceDB [19] for vector search (3 r6g.xlarge instances with data in S3 with index cached in memory). Our brute force baseline uses PySpark on AWS EMR on a cluster of r6i.4xlarge instances. Substring and UUID search are implemented in SQL whereas vector search is implemented with a UDF using the mapInArrow API for optimal performance.

For cpm_i, we include the EBS cost required to replicate the primary index three times for OpenSearch or LanceDB as well as three r6g.large instances. cpm_r and cpm_bf are

[5]In practice, the scaling is not perfect and the cost can still increase. We will examine this later in Section VII-A.

[6]Phase change diagrams in physics plot the region of temperature and pressure where a molecule forms a solid, liquid, or gas.

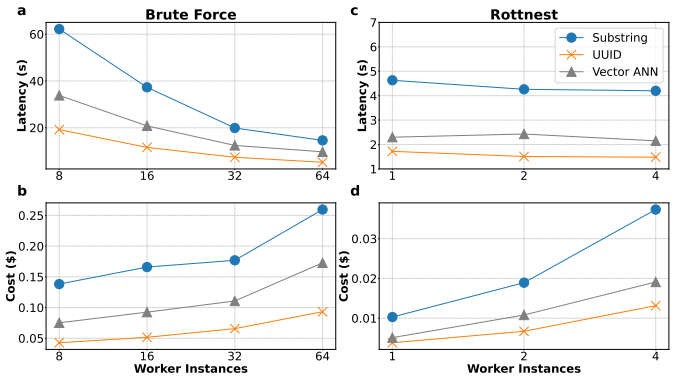

Fig. 8. Brute force approach latency (a) and cost (b) scaling with cluster size. Rottnest latency (c) and cost (d) scaling with cluster size. Worker instance used is r6i.4xlarge with 16 vCPUs.

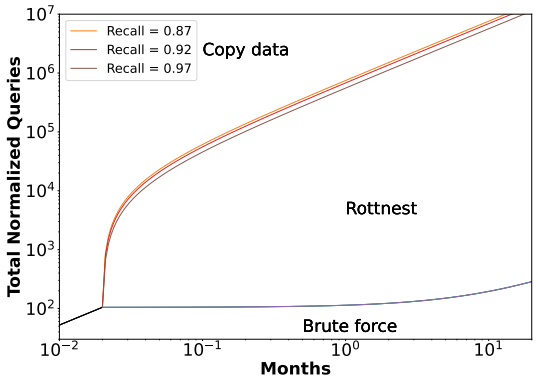

Fig. 9. Phase change diagrams for vector search at different recall targets. Note log-log axes.

computed based on the cost of storing the raw data and the raw data plus the Rottnest index on S3 respectively. cpq_bf and cpq_r are computed from query latency times the hourly cost of the EC2 instances on which the queries are executed. The indexing cost ic_r, includes both the EC2 instance cost for Rottnest to compute initial indices and adequate compaction.[7] Technically, ic_r, cpq_bf and cpq_r should include the cost of S3 requests. In practice, we find them eclipsed by compute resource costs so focus on the latter for the evaluation.

A. Minimum Latency Thresholds

Before we address TCO, we seek to determine the minimum latency threshold of Rottnest and the brute force approach for the three applications. We examine how horizontally scaling the number of machines impact the latency and the query cost, as described in the last section, in Figure 8.

From Figure 8a and 8b, we see that Spark is fairly horizontally scalable up to 32 worker instances across the three query types. Scaling to 64 workers leads to a marked decrease in latency improvement and therefore a large increase in cost per query. The latency at 64 workers can be taken as an estimate of the minimum latency threshold where brute force becomes a viable approach.

Rottnest is not easily horizontally scalable, since it is often latency bound instead of throughput bound: as discussed in Section V-B, we find ourselves bottlenecked by the *depth* of our object storage reads instead of the *width*. As a result, Figure 8c and 8d show the latency barely improving with more searchers, while the cost almost linearly increases. Indeed, Rottnest is designed to be operated in practice with a shared-nothing architecture.

We note that for all three query types, Rottnest's latency on one worker still outperforms brute force latency on 64 workers by a large margin: 4.3x for substring and UUID search, and 5.4x for vector search. This means Rottnest has a much lower minimum latency threshold: 4.6s for substring search, 1.7s for UUID search and 2.3s for vector search.

[7]Compaction could also be counted in cpm_r. For the append-heavy datasets here, data no longer needs to be compacted once they are in adequately sized indices, making it more appropriate to include compaction cost in the upfront indexing cost.

B. Total Cost of Ownership

We now apply our TCO evaluation framework described in Section VI. We use 8 worker instances for brute force and a single instance for Rottnest. Note they are the most cost efficient configurations explored in the last section for both approaches. We separate the three applications into **exact queries** (substring and UUID) and **scoring queries** (vector), where the evaluation is complicated slightly by recall tradeoffs.

1) Exact Queries: We first evaluate the exact match queries, i.e. UUID and substring search. The phase diagrams are plotted on Figure 7. We see that the point at which Rottnest becomes a competitive option starts very early for both applications (2 days for substring search and ¡1 day for UUID search). As the number of months increase, the range of total query numbers in which Rottnest is most economical grow to span more than 4 orders of magnitude: from around 8×10^2 to 4×10^6 total queries at 10 months for substring search and from 3×10^2 to 10^7 for UUID search.

We see a curvature up in the boundary between Rottnest and brute force for substring search since the Rottnest indices are almost as large as the compressed Parquets in this case. This makes brute force increasingly economical as the operating duration increases. For the UUID search, the indices are much smaller, so the boundary stays flat. This behavior will be discussed in detail in Section VII-D.

2) Scoring Queries: The evaluation approach for vector search needs to be modified since query cost can be traded off with recall for Rottnest. We assume that changing the recall target has negligible effect on the TCO of the other two approaches. This is definitely the case for brute force, where the recall is always perfect. This is less true for copy data approach, where the recall target could degrade the throughput of the vector database which might require more servers to hit a QPS target. We ignore this consideration here, which makes the copy data baseline stronger in the evaluation presented.

The Rottnest vector index is based on IVF-PQ [47], [54]. We tune the *nprobe* and *refine* parameters to hit different recall@10 targets: 0.87, 0.92 and 0.97. The former controls

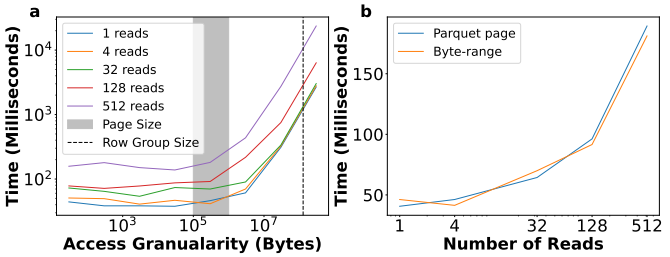

Fig. 10. Parquet reading benchmarks showing how a) read latency increases with read granularity at different number of concurrent reads and b) the latency of reading 300KB byte ranges compares to reading real Parquet pages.

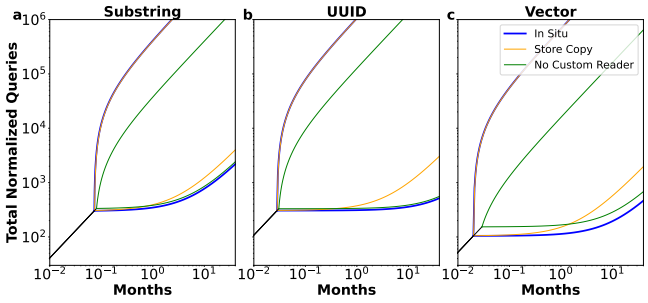

Fig. 11. Changes to the phase diagram if Rottnest keeps a copy of the data in custom format or if it did not use an optimized custom Parquet reader.

how many centroids to probe and PQ vectors to rank, whereas the latter controls how many full precision vectors to download and rerank. Increasing these parameters improves recall but also increases search latency and cpq_r.

We show the phase diagrams corresponding to the different recall targets in Figure 9. In our experiments, a recall target of 0.97 leads to a higher search latency, and thus cpm_r, 35% worse compared to a recall of 0.87. However this difference barely changes the area Rottnest wins in the log-log plot, which covers around 4 orders of magnitude at 10 months. Indeed, given the large orders of magnitude differences between cpm_i and cpm_r, the small changes in cpq_r does not move the boundary significantly. Concretely, this means building a Rottnest index is most likely still a good decision if recall target changes due to business requirements or if different queries have different recall requirements.

C. In-situ Querying

A key decision point for Rottnest is to read raw data from the underlying Parquet files, instead of copying the raw data into a custom storage format. This decreases both ic_r and cpm_r as it reduces storage requirements. The Rottnest index is typically much smaller than the compressed data itself, so storing a copy of the data would multiply the storage overhead several fold. In Figure 11, we show the effect including a copy of the data would have on the phase diagram of the UUID search. On longer time horizons at around 10 months, it reduces the range of total queries where Rottnest is more cost-effective than brute force by a few times and the cost benefit compared to brute force in general.

The catch is that keeping the data in Parquet makes querying more challenging, as open source Parquet reader implementations cannot efficiently perform random access on this data, increasing cpq_r. In Figure 11 we show that without a custom reader, Rottnest becomes less efficient than the copy data approach over several orders of magnitude. Rottnest resolves this by designing its own optimized Parquet reader that reads only the required column pages (∼300KB) vs entire row groups (∼128MB).

Compared to an ideal custom format that allows the reader to fetch only the bytes necessary for a data item (typically 0.1-4KB) without decompression, our Parquet reader has a read granularity of 300KB and must decompress the whole read to fetch any item. We experimentally validate that our Parquet reader makes our in-situ querying likely as efficient as using this ideal custom format. We first show in Figure 10a that byte-range read request latency to S3 is stable in terms of read granularity until around 1MB, at which point it increases linearly with the read size. This observation holds for different numbers of concurrent reads from 1 to 512. While the size of Parquet row groups puts it in the throughput bound regime, the size of Parquet pages puts it squarely in the latency bound regime. Concretely, this means using a custom storage format to perform more granular reads is unlikely to result in improved performance. In addition, we show in Figure 10b that there is little difference in terms of performance between reading 300KB byte ranges and reading and decoding actual Parquet pages in our Parquet reader, showing that decompression overhead is not a concern.

To further evaluate this key design choice, we directly compare Rottnest's query performance to LanceDB cold cache mode which uses its own custom Lance format. In contrast to the LanceDB configuration used to benchmark the copy-data approach above where the index is kept in memory, we keep the index on S3 and query it directly similar to Rottnest, using optimized configurations tuned by a core LanceDB maintainer. Rottnest achieves comparable search latency at all recall targets: 2.09s (Rottnest) vs 1.90s (Lance) at 0.87, 2.30s vs 1.94s at 0.92 and 2.81s vs 2.72s at 0.97. This experimentally validates that using a custom format is unlikely to significantly improve query performance compared to in-situ querying with Rottnest's custom Parquet reader.

D. Sensitivity Analysis

1) Parameter Robustness: The last section has demonstrated that changing cpq_r and cpm_r have dramatic effects on the phase diagram. In this section, we systematically examine the impact of changing cpq_r, ic_r and cpm_r on the phase diagram to understand the effects of optimizing each. Figure 12 demonstrates how the phase diagrams shift for vector search (0.92 recall) when each of these parameters are multiplied by the shown factor. For cpm_r, we show the result of scaling $cpm\_r - cpm\_bf$, or just the storage cost associated with the Rottnest index files. Two observations:

1) Decreasing Rottnest search latency (cpq_r) makes it more competitive against copy data, with virtually no benefit

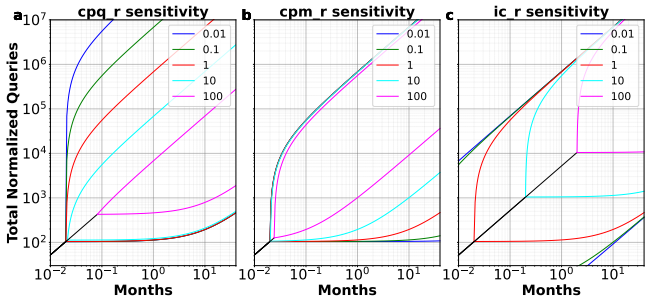

Fig. 12. Sensitivity analysis of cpq_r, ic_r and cpm_r for vector search application at recall 0.92. Contours indicate phase diagrams if each of the parameter is multiplied by the denoted factor. The actual diagram is in red.

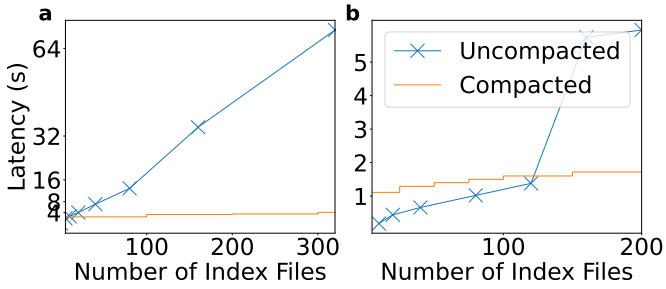

Fig. 13. Search latency on uncompacted vs. compacted index files for a) substring (100x compaction factor) and b) UUID search (25x compaction factor). Compaction greatly reduces search latency when there is a large number of index files.

against brute force. Decreasing the Rottnest index size (cpm_r) does exactly the opposite.

2) Reducing the indexing cost reduces the minimum operating time at which Rottnest becomes worthwhile compared to the other approaches. On the other hand, it does little to the asymptotic boundary between Rottnest and the other two approaches at longer operating time horizons.

These observations inform how optimizations in Rottnest directly benefit different classes of use cases: making the search faster benefits high query load applications; making the index smaller benefits low load applications and making the index construction cheaper benefits applications with short operational lifetimes.

2) Dataset Size Robustness: So far, all the parameters are computed based on fixed dataset sizes. While the key parameters cpm_i, cpm_bf, cpq_bf, ic_r, cpm_r and cpq_r are evidently dependent on the *data distribution* in complex nonlinear ways[8], most are almost perfectly linearly correlated with *dataset size* assuming the same data distribution, which would imply no change in the phase diagram.

While cpq_r generally scales with the number of index files queried, which generally increases linearly with dataset size, Rottnest compactions could greatly reduce this number to dramatic effect as seen in Figure 13. For the case of UUID search, we also see nonlinear scaling due to AWS list request throttling issues. Post compaction, the Rottnest search latency is effectively constant irrespective of the dataset size, which means that as data volume increases, cpq_r stays relatively constant while all other parameters increase linearly, making Rottnest more attractive against the copy data approach as shown in Figure 12a.

3) Throughput Limitations: While our evaluation framework covers concerns such as search latency, total operating cost and operating duration, we did not discuss the maximum throughput supported by the three approaches. While the copy data approach is typically bottlenecked by the disk IOPs and CPUs of the dedicated servers, Rottnest and the brute force approach are bottlenecked by S3's limit of 5500 GET RPS per prefix. While the number of requests Rottnest makes is heavily dependent on the query, this limit typically caps Rottnest's QPS at 10-100. However, from Figure 7 and Figure 9, Rottnest already underperforms copy-data approach at these QPS levels (10QPS = 2.52×10^7 total queries at 10 months). As a result, these throughput limits do not significantly change the conclusions drawn in this paper.

VIII. Related Work and Discussion

In this paper, we present Rottnest, a bolt-on indexing system for data lakes that supports general-purpose search workloads. While database systems have long used indices for various workloads from OLTP to full-text and vector search [55]–[59], data lakes are just beginning to adopt similar capabilities. Existing solutions like Microsoft's Hyperspace [26] and Apache Hudi [60] focus primarily on OLAP workloads and have key limitations: Hyperspace requires atomic rename operations not available in all cloud providers, while Hudi requires tight integration with the data lake's metadata. In contrast, Rottnest supports a broader range of search workloads while maintaining compatibility with any data lake.

We show Rottnest is the most cost-effective solution across several orders of magnitude of total query load across a wide range of operating durations for our exemplar applications. While Rottnest incurs a couple seconds minimum latency, this is acceptable for human-interactive applications or those (e.g. retrieval-augmented generation) where search latency is dominated by other factors like LLM generation.

Besides the TCO considerations presented, there are practical benefits to deploying Rottnest in the aforementioned, typically spiky workloads. Dedicated clusters like ElasticSearch take minutes to scale up and down. Brute force approaches requiring distributed compute is hard to deploy in practice: either a shared cluster is used or each query spins up its own cluster. The former leads to poor performance isolation between different user queries and the latter incurs significant spinup overheads. Since Rottnest provides acceptable search latencies from *one* search instance with object storage as the only shared state, it easily supports scalable shared-nothing deployment architectures with serverless functions like AWS Lambda, greatly reducing infrastructure complexity and cost.

We have open sourced Rottnest[9] to facilitate further research in data lake indexing systems.

[8]For example, entropy influences compression efficacy for text datasets.

[9]https://github.com/marsupialtail/rottnest

REFERENCES

[1] Apache Software Foundation, "Apache parquet," https://parquet.apache.org, 2024, accessed: 2024-08-31.

[2] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, J. Torres, H. van Hovell, A. Ionescu, A. Łuszczak *et al.*, "Delta lake: high-performance acid table storage over cloud object stores," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3411–3424, 2020.

[3] Apache Software Foundation, "Apache iceberg," https://iceberg.apache.org, 2024, accessed: 2024-08-31.

[4] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, J. Torres, H. van Hovell, A. Ionescu, A. Łuszczak *et al.*, "Delta lake: high-performance acid table storage over cloud object stores," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3411–3424, 2020.

[5] Apache Software Foundation, "Apache hudi," https://hudi.apache.org, 2024, accessed: 2024-08-31.

[6] Trino Software Foundation, "Trino," https://trino.io, 2024, accessed: 2024-08-31.

[7] M. Armbrust *et al.*, "Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics," in *Proceedings of CIDR*, vol. 8, 2021.

[8] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner, "The snowflake elastic data warehouse," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 215–226. [Online]. Available: https://doi.org/10.1145/2882903.2903741

[9] "What is delta lake?" https://docs.databricks.com/en/delta/index.html, 2024.

[10] R. Ortloff and S. Herbert, "Unifying iceberg tables on snowflake," https://www.snowflake.com/blog/unifying-iceberg-tables/, Aug 2023.

[11] R. Xin and M. Mokhtar, "Databricks sets official data warehousing performance record," https://www.databricks.com/blog/2021/11/02/databricks-sets-official-data-warehousing-performance-record.html, Nov 2021, accessed: 2024-04-08.

[12] Starburst, "Snowflake alternative: Open source alternative trino," https://www.starburst.io/blog/snowflake-alternatives/, Mar 2024, accessed: 2024-04-08.

[13] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark sql: Relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, 2015, pp. 1383–1394.

[14] A. S. Foundation, "Apache iceberg," https://github.com/apache/iceberg, 2024, accessed: 2024-05-04.

[15] ClickHouse, Inc., "Clickhouse," https://github.com/ClickHouse/ClickHouse, 2024, accessed: 2024-08-31.

[16] ElasticSearch NV, "Elasticsearch," https://github.com/elastic/elasticsearch, 2024, accessed: 2024-08-31.

[17] Qdrant, Inc., "Qdrant," https://github.com/qdrant/qdrant, 2024, accessed: 2024-08-31.

[18] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[19] LanceDB, "Introducing lance v2," https://blog.lancedb.com/lance-v2/, 2024, accessed on [Insert access date here]. [Online]. Available: https://blog.lancedb.com/lance-v2/

[20] Facebook, Inc., "Nimble," https://github.com/facebookincubator/nimble, 2024, accessed: 2024-08-28.

[21] D. at CERN, "Enhancing apache spark and parquet efficiency: A deep dive into column indexes and bloom filters," *CERN Database Blog*, 2024, accessed: 2024-08-31. [Online]. Available: https://db-blog.web.cern.ch/node/194

[22] InfluxData, "Using parquet's bloom filters for efficient query performance," *InfluxData Blog*, 2024, accessed: 2024-08-31. [Online]. Available: https://www.influxdata.com/blog/using-parquets-bloom-filters/

[23] S. Xu and S. Narayanan. (2023) Record level index: Hudi's blazing fast indexing for large-scale datasets. Apache Hudi. Accessed: 2024-04-08. [Online]. Available: https://hudi.apache.org/blog/2023/11/01/record-level-index/

[24] S. Dong, A. Kryczka, Y. Jin, and M. Stumm, "Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications," *ACM Transactions on Storage (TOS)*, vol. 17, no. 4, pp. 1–32, 2021.

[25] *CREATE INDEX - PostgreSQL Documentation*, PostgreSQL Global Development Group, 2024, accessed: 2024-02-03. [Online]. Available: https://www.postgresql.org/docs/current/sql-createindex.html#SQL-CREATEINDEX-CONCURRENTLY

[26] R. Potharaju, T. Kim, E. Song, W. Wu, L. Novik, A. Dave, A. Fogarty, P. Pirzadeh, V. Acharya, G. Dhody, J. Li, S. Ramanujam, N. Bruno, C. A. Galindo-Legaria, V. Narasayya, S. Chaudhuri, A. K. Nori, T. Talius, and R. Ramakrishnan, "Hyperspace: the indexing subsystem of azure synapse," *Proc. VLDB Endow.*, vol. 14, no. 12, p. 3043–3055, Jul. 2021. [Online]. Available: https://doi.org/10.14778/3476311.3476382

[27] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki, "Nodb: efficient query execution on raw data files," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 241–252.

[28] M. Kuschewski, D. Sauerwein, A. Alhomssi, and V. Leis, "Btrblocks: efficient columnar compression for data lakes," *Proceedings of the ACM on Management of Data*, vol. 1, no. 2, pp. 1–26, 2023.

[29] X. Zeng, Y. Hui, J. Shen, A. Pavlo, W. McKinney, and H. Zhang, "An empirical evaluation of columnar storage formats," *arXiv preprint arXiv:2304.05028*, 2023.

[30] M. A. Bender, E. D. Demaine, and M. Farach-Colton, "Cache-oblivious b-trees," in *Proceedings 41st Annual Symposium on Foundations of Computer Science*. IEEE, 2000, pp. 399–409.

[31] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," *ACM Transactions on Algorithms (TALG)*, vol. 8, no. 1, pp. 1–22, 2012.

[32] J. Tan, T. Ghanem, M. Perron, X. Yu, M. Stonebraker, D. DeWitt, M. Serafini, A. Aboulnaga, and T. Kraska, "Choosing a cloud dbms: architectures and tradeoffs," *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 2170–2182, 2019.

[33] Databricks, "Databricks sets official data warehousing performance record," 11 2021, databricks Engineering Blog. [Online]. Available: https://www.databricks.com/blog/2021/11/02/databricks-sets-official-data-warehousing-performance-record.html

[34] A. Van Renen and V. Leis, "Cloud analytics benchmark," *Proceedings of the VLDB Endowment*, vol. 16, no. 6, pp. 1413–1425, 2023.

[35] D. rs Contributors, "delta-rs: Native rust bindings for delta lake," https://github.com/delta-io/delta-rs, 2024, accessed: 2024-05-27.

[36] D. L. Team, "Introducing deletion vectors in delta lake: Streamlined data deletion for faster queries," *Delta.io Blog*, 2023, accessed: 2024-08-31. [Online]. Available: https://delta.io/blog/2023-07-05-deletion-vectors/

[37] A. Merced, "Understanding apache iceberg delete files," *Medium*, 2022, accessed: 2024-08-31. [Online]. Available: https://alexmercedcoder.medium.com/understanding-apache-iceberg-delete-files-0b445df5872f

[38] J. Barr, "Amazon s3 update – strong read-after-write consistency," *AWS News Blog*, 2020, accessed: 2024-08-31. [Online]. Available: https://aws.amazon.com/blogs/aws/amazon-s3-update-strong-read-after-write-consistency/

[39] Amazon Web Services. (Latest) Amazon Simple Storage Service (S3) Documentation. Amazon Web Services. [Online]. Available: https://docs.aws.amazon.com/AmazonS3/latest/userguide/optimizing-performance-guidelines.html

[40] D. Durner, V. Leis, and T. Neumann, "Exploiting cloud object storage for high-performance analytics," *Proceedings of the VLDB Endowment*, vol. 16, no. 11, pp. 2769–2782, 2023.

[41] M. Burrows, "A block-sorting lossless data compression algorithm," *SRS Research Report*, vol. 124, 1994.

[42] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Proceedings 41st annual symposium on foundations of computer science*. IEEE, 2000, pp. 390–398.

[43] J. Holt and L. McMillan, "Merging of multi-string bwts with applications," *Bioinformatics*, vol. 30, no. 24, pp. 3524–3531, 2014.

[44] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE transactions on pattern analysis and machine intelligence*, vol. 42, no. 4, pp. 824–836, 2018.

[45] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2019.

[46] S. Jayaram Subramanya, F. Devvrit, H. V. Simhadri, R. Krishnawamy, and R. Kadekodi, "Diskann: Fast accurate billion-point nearest neighbor search on a single node," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[47] "Index ivfpq - lancedb documentation," 2024, accessed: 2024-08-26. [Online]. Available: https://lancedb.github.io/lancedb/concepts/index_ivfpq/

[48] Y. Xu, H. Liang, J. Li, S. Xu, Q. Chen, Q. Zhang, C. Li, Z. Yang, F. Yang, Y. Yang *et al.*, "Spfresh: Incremental in-place update for billion-scale vector search," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 545–561.

[49] Turbopuffer, "Turbopuffer system architecture," 2024, turbopuffer Documentation. [Online]. Available: https://turbopuffer.com/architecture

[50] L. Patel, P. Kraft, C. Guestrin, and M. Zaharia, "Acorn: Performant and predicate-agnostic search over vector embeddings and structured data," *Proceedings of the ACM on Management of Data*, vol. 2, no. 3, pp. 1–27, 2024.

[51] M. Wang, L. Lv, X. Xu, Y. Wang, Q. Yue, and J. Ni, "An efficient and robust framework for approximate nearest neighbor search with attribute constraint," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[52] G. Penedo, H. Kydlíček, A. Lozhkov, M. Mitchell, C. Raffel, L. Von Werra, T. Wolf *et al.*, "The fineweb datasets: Decanting the web for the finest text data at scale," *arXiv preprint arXiv:2406.17557*, 2024.

[53] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, pp. 91–110, 2004.

[54] M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvasy, P.-E. Mazaré, M. Lomeli, L. Hosseini, and H. Jégou, "The faiss library," *arXiv preprint arXiv:2401.08281*, 2024.

[55] J. J. Levandoski, D. B. Lomet, and S. Sengupta, "The bw-tree: A b-tree for new hardware platforms," in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, 2013, pp. 302–313.

[56] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson, "Bztree: a high-performance latch-free range index for non-volatile memory," *Proc. VLDB Endow.*, vol. 11, no. 5, p. 553–565, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3164135.3164147

[57] A. Prout, "The story behind singlestore's skiplist indexes," 2019. [Online]. Available: https://www.singlestore.com/blog/what-is-skiplist-why-skiplist-index-for-memsql/

[58] C. Chen, C. Jin, Y. Zhang, S. Podolsky, C. Wu, S.-P. Wang, E. Hanson, Z. Sun, R. Walzer, and J. Wang, "Singlestore-v: An integrated vector database system in singlestore," *Proc. VLDB Endow.*, vol. 17, pp. 3772–3785, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:272727885

[59] A. Prout, S.-P. Wang, J. Victor, Z. Sun, Y. Li, J. Chen, E. Bergeron, E. Hanson, R. Walzer, R. Gomes, and N. Shamgunov, "Cloud-native transactions and analytics in singlestore," in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 2340–2352. [Online]. Available: https://doi.org/10.1145/3514221.3526055

[60] S. Sumit, "Asynchronous indexing using hudi." [Online]. Available: https://www.onehouse.ai/blog/asynchronous-indexing-using-hudi