Hyperspace: The Indexing Subsystem of Azure Synapse

Rahul Potharaju, Terry Kim, Eunjin Song, Wentao Wu, Lev Novik,
Apoorve Dave, Andrew Fogarty, Pouria Pirzadeh, Vidip Acharya, Gurleen Dhody, Jiying Li,
Sinduja Ramanujam, Nicolas Bruno, César A. Galindo-Legaria, Vivek Narasayya,
Surajit Chaudhuri, Anil K. Nori, Tomas Talius, Raghu Ramakrishnan
Microsoft Corporation
hyperspace@microsoft.com

ABSTRACT

Microsoft recently introduced Azure Synapse Analytics, which offers an integrated experience across data ingestion, storage, and querying in Apache Spark and T-SQL over data in the lake, including files and warehouse tables. In this paper, we present our experiences with designing and implementing Hyperspace, the indexing subsystem underlying Synapse. Hyperspace enables users to build multiple types of secondary indexes on their data, maintain them through a multi-user concurrency model, and leverage them automatically—without any change to their application code—for query/workload acceleration. Many requirements of Hyperspace are based on feedback from several enterprise customers. We present the details of Hyperspace's underlying design, the user-facing APIs, its concurrency control protocol for index access, its index-aware query processing techniques, and its maintenance mechanisms for handling index updates. Evaluations over standard industry benchmarks and real customer workloads show that Hyperspace can accelerate query execution by up to 10x and in certain real-world workloads, even up to two orders of magnitude.

PVLDB Reference Format:
Rahul Potharaju, Terry Kim, Eunjin Song, Wentao Wu, Lev Novik, Apoorve Dave, Andrew Fogarty, Pouria Pirzadeh, Vidip Acharya, Gurleen Dhody, Jiying Li, Sinduja Ramanujam, Nicolas Bruno, César A. Galindo-Legaria, Vivek Narasayya, Surajit Chaudhuri, Anil K. Nori, Tomas Talius, Raghu Ramakrishnan. Hyperspace: The Indexing Subsystem of Azure Synapse. PVLDB, 14(12): 3043 - 3055, 2021.
doi:10.14778/3476311.3476382

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/microsoft/hyperspace.

1 INTRODUCTION

The on-going convergence of data lakes and data warehouses is seeing the emergence of support for efficiently updatable and versioned relational data with change tracking [27, 28, 30], and competitive relational query capabilities at very large scale [2]. We are seeing out-of-box support for relational tool chains for reporting, data orchestration, security, sharing, compliance, and governance. At the same time, we observe data warehouses expanding their capabilities to support data diversity and scale to match lake capabilities. This convergence of warehouses and lakes adds value through service interoperability and consistent capabilities—including security, governance, lifecycle management, and cost management—over all data: in the cloud, on the edge, or on-prem.

Microsoft recently introduced Azure Synapse [2], which offers an integrated experience across data ingestion, storage, and querying in Apache Spark [15] and T-SQL, over data in the lake and warehouse tables. Our customers store datasets ranging from a few GBs to 100s of PBs and utilize several analytical engines to process this data. The scope of usage spans traditional data processing (e.g., ETL), analytical (e.g., OLAP), exploratory (e.g., "needle in a haystack" queries) and deep learning workloads.

Based on our experience working with customers, we observed that most enterprise workloads begin with data streaming continuously into the data lake via various means (e.g., telemetry from edge devices, usage data from business applications, click-stream data in web apps and search engines). Subsequently, complex ETL pipelines transform this data for downstream consumption and make it available to data and business analysts via shared *derived datasets* or traditional reporting applications. It is expected that the underlying data can be updated due to various business requirements (e.g., to enforce privacy policies like GDPR [42], as part of hybrid transaction/analytical processing pipelines [17], corrections due to accounting errors). To meet critical business and strict SLA requirements, users spend significant time in optimizing their pipelines. On one hand, to obtain desired performance, users spend a lot of time and energy creating several *derived datasets* each with their own sort-orders and/or partitioning, hand-tuned / optimized for their query patterns—increasingly assuming the role of *big data administrators*. On the other hand, to handle updates efficiently and avoid GDPR penalties, enterprise users implement custom solutions—increasingly assuming the role of *big data architects*—and spend significant effort in dataset lifecycle management.

While some of the difficulties around streaming ingestion and handling updates are addressed by recent work on updatable formats [27, 28, 30], the whole experience of having to manage lifecycle of *derived datasets* and then having to manually decide which dataset to use to obtain the required query acceleration is frustrating, error-prone and a far cry from what traditional databases have advocated for and provided. We argue that a system that offers a simple user experience, hides the complexity of building and managing these datasets and allows transparent usage in query processing is critical for any industrial offering such as Azure Synapse. Specifically, such a system needs to address two challenges in the context

Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476382

of data lakes: *how to efficiently store a derived dataset, maintain it and make it available to query engines in the lake*, and *how to automatically select the best derived dataset during query optimization*.

With these in mind, we designed Hyperspace, an indexing subsystem that we started offering to customers and open-sourced in 2020 [43]. The key idea behind Hyperspace is simple: Users specify the derived datasets (referred to henceforth as *indexes*) that they want to build.[1] Hyperspace builds these indexes using an existing scale-out engine, Apache Spark [15], and maintains metadata in its write-ahead log that is stored in the data lake. At runtime, Hyperspace *automatically* selects the best index to use for a given query without requiring users to rewrite their queries. Hyperspace indexes are exposed to users as an alternative performance enhancement option and are complementary to other query acceleration mechanisms such as data partitioning [61] available for data lakes. Our key contributions in this paper are:

- **Indexing Subsystem for Data Lakes.** We present the details of an extensible indexing engine, the user experience, the underlying physical layouts, how we leverage indexes for query acceleration, and describe the index management lifecycle.
- **Serverless Index Management.** To lower operational costs, we propose a novel "serverless" index management strategy. We also discuss how such a design enables us to provide multi-engine interoperability and multi-user concurrency.
- **Large-Scale Evaluation.** We present an evaluation of Hyperspace using both industry benchmarks and real customer workloads, **TPC-H**, **TPC-DS**, and **Customer**. Overall, we have observed gains of 40% to 75% for **TPC-H**, 50% for **TPC-DS**, and 44% to 94% for **Customer**, using commodity hardware. In some cases, we saw speedups of over two orders of magnitude.

In more detail, there were several considerations that went into designing Hyperspace:

(1) *Agnostic to data formats.* Hyperspace is format-agnostic and supports indexing most popular formats (e.g., CSV, JSON, Parquet [1]).

(2) *Indexes are data.* The index itself is stored in the lake like any other dataset in an open and efficient columnar format, Parquet[2], which has the following major benefits. First, in practice many queries only access subsets of (instead of all) columns from underlying tables. Second, users continue to benefit from a large number of optimizations being committed by ~200 developers from the open-source communities [7, 29]. For example, Parquet's min-max based pruning allows skipping irrelevant data blocks; as a result, scanning Hyperspace indexes can have similar performance benefits offered by other well-known techniques such as *zone maps*. Third, users also benefit from broader community efforts to bring hardware acceleration via GPUs [4, 51] and FPGAs [5].

(3) *Diverse use cases.* Developers are provided a flexible framework to extend and build their own auxiliary structures (e.g., B-tree, R-Tree) in addition to indexes already supported by Hyperspace.

(4) *Serverless and multi-engine indexes.* Hyperspace stores the index and the associated metadata on the lake. The metadata log contains (a) lineage that allows Hyperspace to enable seamless incremental index refresh operations and (b) statistics that allow very fast pruning of the search space. Since the log is stored in the lake, Hyperspace offers a "serverless" index management solution, i.e., users only need to launch a batch job to refresh the index and can benefit from scalable computation (to refresh the index). Further, such a design also enables providing multi-engine index experiences and integrating Hyperspace into Polaris [13].

(5) *Integration with query optimizer.* The current experience of Hyperspace inside Apache Spark is quite simple and requires no code change from its users. This was achieved through an integration with the underlying query optimizer that is agnostic to end users, allowing the optimizer to make good choices of which indexes to use to answer a query rather than forcing application developers to choose indexes, thereby simplifying usability.

Based on this simple design, we built several features in Hyperspace to address common customer pain points. First, the ability to *incrementally refresh* an index (Section 5.2) in the data lake is suitable for common streaming workloads. Using lineage mechanisms (Section 3.1), Hyperspace supports even the most advanced data formats such as Delta Lake [30] that support updates (Section 5.3) including their *time travel* feature. Second, Hyperspace offers out-of-box support for *history tracking* through its transaction log (Section 4.1), useful in audit logging for security and compliance. Third, to avoid problems with handling large numbers of files, Hyperspace offers an OPTIMIZE command that can bin-pack index files into a compact representation (Section 5.2). Finally, Hyperspace allows users to continue to exploit stale indexes[3] through its novel *hybrid scan* mechanism (Section 6.2), without sacrificing correctness.

Hyperspace offers users a simple framework to optimize the management and performance of their workloads. While indexing is not a silver bullet, we have observed significant workload acceleration, with some customers reporting speedups as high as two orders of magnitude. The open-source Hyperspace project [43] enables indexing support in Apache Spark and supports most widely used data formats such as CSV, JSON, Delta Lake, and Iceberg. It is currently in use by several enterprises, three of which are large enterprises including Microsoft. In the rest of this paper we present the design and implementation of Hyperspace.

Scope. There have been techniques for accelerating query performance in big data systems, in particular, by using various types of indexes (e.g., Z-order [47] in Delta Lake [30], R-tree in GeoSpark [65], Reflections in Dremio [24]). Hyperspace indexes share similar goals with these techniques in this aspect. However, Hyperspace is *not only* a query acceleration technology; rather, its emphasis is more on the automated *lifecycle management* of such query accelerators in the big data and/or data lake world, enabling *simplified index creation/maintenance* and *automated index utilization*. It is in this latter aspect that we are not aware of existing technology that shares similar goals. In a sense, Hyperspace is complementary to the above specific indexing techniques and allows for incorporation of *any* type of index into the end-to-end index utilization and management experience. The extensible and open design of Hyperspace allows it to take advantage of state-of-the-art in big data technology while offering simplified end-user experiences to customers and staying open to community contributions.

[1] Note that we abuse terminology a bit for ease of exposition, as derived datasets can be more general than indexes in their traditional sense. For example, a materialized view is a type of derived dataset that is typically not considered an index in the literature.

[2] Hyperspace can easily be extended to store its indexes in other columnar formats.

[3] Indexes go stale when the underlying data is modified but the index was not refreshed.

Availability. Hyperspace has been open-sourced [43] and operates in Microsoft's own production environments [44].

Paper organization. This paper is organized as follows. In Section 2, we present real-world use cases of Hyperspace in Azure Synapse Analytics. In Section 3, we discuss the design goals behind Hyperspace and provide an overview of its capabilities along with the public APIs. In Section 4, we explain how we implement Hyperspace as a low-cost indexing subsystem. In Section 5, we present the incremental index maintenance mechanism in Hyperspace. In Section 6, we discuss how the query optimizer utilizes Hyperspace indexes in query processing. In Section 7, we present evaluation results of Hyperspace. We discuss related work in Section 8, and present our conclusions in Section 9.

2 HYPERSPACE USE CASES

Hyperspace has been evolving continuously since its inception almost three years ago based on customer feedback. While Hyperspace is broadly applicable to any query acceleration scenarios, in this section, we present some representative patterns leveraged by customers within Azure Synapse Analytics [44].

High-Concurrency Interactive Analytics and Data Export. A common scenario we observed in enterprise applications in various sectors such as finance, accounting, and sales, is the following: *Users can input some predicates and they can either look at the results or export the data.* Since there are multiple query patterns, a single layout (e.g., data partitioning, distribution, or sort-order) may not suffice to meet all SLAs. We observed customers utilizing Hyperspace to create multiple indexes on their data (~8 TB/day) and leveraging its *index caching* mechanism to preemptively load indexes (~700 GB/day) into an active set of nodes (~60). Subsequently, they would build an intermediate *index serving layer* that takes in queries, writes the results into a blob store, and returns a URI to the end user so they can download. In some cases, customers obtained speedups of up to two orders of magnitude.

Indexing Privacy Attributes for GDPR Compliance. The General Data Protection Regulation (GDPR) [42] is a protection directive introduced by the European parliament that requires companies holding EU citizen data to provide a certain level of protection for personal data (e.g., biometric data, user activity data, etc.), including erasing all personal data upon request. This requirement applies to any product or service that collects user information (e.g., health and fitness, chat, etc.). We observed customers utilizing Hyperspace indexes to speed up searches for which data blobs hosted by Azure Storage (i.e., WASB [11]) contain the given user's data (via "lookup" style queries). Once these blobs are deleted, users perform an index refresh to delete the entries from the index. Since Hyperspace offers incremental refresh with support for deletes, this operation is efficient (as it only rewrites index blocks that are affected).

Time-series Analytics. Many customers focus on turning IoT data into actionable insights using products such as Time Series Insights [45] that allow users to specify the sensors from which to collect time series data and allow them to slice and dice so they can derive insights. A common pattern of requests we observed is that for data spanning multiple years, users issue queries in ways that do not align with how the data is partitioned/distributed on the data lake leading to sub-optimal performance. For instance, data is generally partitioned by timestamp but most queries select by *sensor id* instead. To avoid linear scans, we have observed customers using Hyperspace to create secondary indexes to efficiently answer such queries. An interesting observation here is that Hyperspace maintenance works well with any retention rates associated with the underlying raw data.

Complex View Support over Lake Data. With data from multiple sources flowing into the lake, in near real-time, applications are increasingly leveraging lake data for operational processing and analytics. Operational data processing requires support for data abstractions from source operational system and support for complex read processing. SQL views[13] are a useful way to provide such abstractions. These abstractions themselves can be complex. For example, consider a User (or Contact) entity in the lake, which is composed from multiple source datasets like user core data, user address data, user profile data, and so on, requiring multi-way joins. Efficient support for such views requires efficient query processing over source datasets. Hyperspace can provide support for efficient query processing over source datasets and over other views, without requiring them to be physically materialized.

Framework for Derived Dataset Maintenance. Many enterprise workloads are migrating ETL and data warehouse workloads to the cloud to simplify their management. In certain use cases, enterprises take regular snapshots of data in OLTP systems (e.g., sales) and join it with other richer sources (e.g., product usage) to derive other useful insights. We find many of these enterprises still prefer storing their raw data in CSV files as they do not want to impose restrictions for downstream pipelines. We observed users using Hyperspace to define materialized views—a type of derived datasets/indexes in Hyperspace—as optimized representations of the raw data. This not only allows downstream pipelines to benefit from query acceleration but also allows the teams to exploit Hyperspace's maintenance framework to keep everything up-to-date when the underlying data changes.

Needle-in-a-haystack Queries. Many customers are interested in text searches in a large set of files on the lake across all columns. Bloom filter indexes can be used to address the problem. An interesting side-effect would be their benefits in join optimizations, which we plan to address in future work.

3 HYPERSPACE INDEXING SUBSYSTEM

In this section, we present an overview of Hyperspace and its APIs, and discuss the underlying index structures. Hyperspace is designed to work with multiple query engines, but for concreteness, we explain the indexing subsystem and its use in the context of Apache Spark, without loss of generality.

3.1 Hyperspace Indexes

Hyperspace supports traditional *non-clustered indexes* (i.e., index is separated from data). Hyperspace indexes have the following physical properties (see Figure 1):

(1) **Columnar:** Index is stored in a columnar format (we use Parquet, but in principle any other format can be used). This not only allows us to benefit from community contributed optimizations and hardware advancements but also leverage techniques

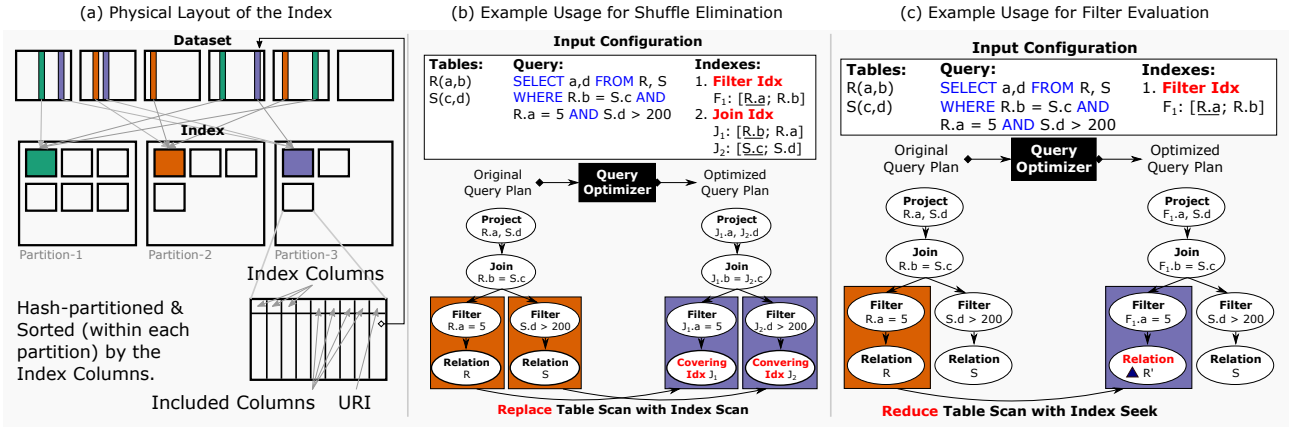


Figure 1: Hyperspace Indexes. (a) describes the physical layout; (b) and (c) show examples of shuffle and filter acceleration.

like vectorization along with min-max pruning [46] to accelerate scans over indexes.

(2) **Hash-Partitioned & (Optionally) Sorted:** Index is hash partitioned and sorted by the index columns within each individual partition, based on user-provided index configuration. Index columns can hold generic data types, e.g., bloom filters. This allows index scans to reduce the amount of data to be accessed for filter with equality predicates (i.e., point look-ups) that reference the indexed columns. If the indexed columns match join keys and the optimizer chooses to use a shuffle-based join (e.g., a hash join or a sort-merge join), then the shuffle stage of the join can be avoided due to the index being pre-partitioned [3].

(3) **(Optional) Lineage Tracking:** Every index can also optionally hold pointers/references to the underlying base table (incidentally, these are also useful for lineage tracking and enforcing deletes; see Section 5.2). However, since data in the lake does not typically have a primary key or a clustered index, in Hyperspace we exploit the following observation for data lake implementations [19, 57, 59]: *Each file/blob/folder in the underlying storage system is addressible through a handle (e.g., URI) that can be used to efficiently retrieve the underlying data.* Users can create secondary indexes that map the indexed columns to the handle of a data block that contains the full record. We note that Hyperspace allows the URI to be either fine-grained (e.g., directly pointing to a row-group in Parquet) or coarse-grained (e.g., pointing to a file or a folder).

Such indexes allow the following key query optimizations:

Replace *Table Scans* with *Index Scans*. When the index contains all information required to resolve a query as part of its key columns (a.k.a., "*indexed columns*") and data/payload columns (a.k.a, "*included columns*")—a *covering index*—they can be tremendously useful for "index-only" access paths (including for optimizing joins); see Figure 1(b). For instance, in cases where one or more joins have appropriate partition-aligned indexes, global shuffles (some of the most expensive operations in data lakes [54]) can be entirely eliminated. We illustrate these benefits through an empirical study that we conducted to measure the benefits of avoiding data shuffling under two scenarios: (i) query engine is aware of the underlying data distributions of the two tables involved in a join so it can avoid

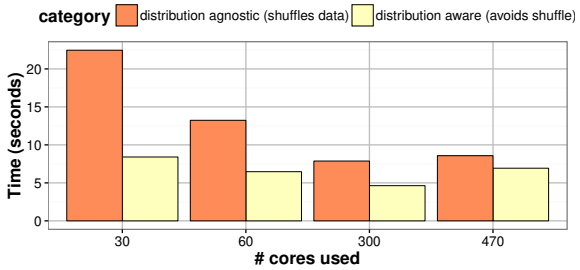


Figure 2: Benefits of getting rid of shuffles.

a global distributed shuffle, and (ii) query engine is not aware of the underlying data distributions so it will perform a full distributed shuffle. We compare the query execution time as the number of CPU cores grow, for both cases (i) and (ii) with the following micro-benchmark query:

```
SELECT count(*) FROM lineitem, orders
WHERE l_orderkey = o_orderkey.
```

Here `lineitem` and `orders` are the two largest tables in the 1TB **TPC-H** benchmark [10]. Notice the difference of up to 2.5x in query execution time between the two cases. This difference will increase significantly as the amount of shuffling increases; see Section 7.

Reduce *Table Scans* with *Index Seeks*. When the index does not cover the given query, it may still be useful for reducing the amount of data accessed by a table scan, as shown in Figure 1(c). During query optimization, Hyperspace uses such indexes to eliminate portions of the table that can be skipped, based on column selectivity.

Remark. In practice, it is easy to include other types of indexes into Hyperspace as long as they can be used for index scans or seeks. Indeed, with this point of view, we are embedding several other well-known data structures, such as materialized views, Z-ORDER, data skipping, sketches, inverted lists and even bloom filters, into Hyperspace [43]. One obvious challenge is how to use these indexes to accelerate query processing, which implies that the query optimizer needs to be able to *rewrite* the query plan when such indexes are available and estimate their execution cost accordingly; see Section 6. Another challenge is to ensure the freshness of such indexes when the underlying datasets are updated, which implies that index maintenance is critical in Hyperspace; see Section 5.

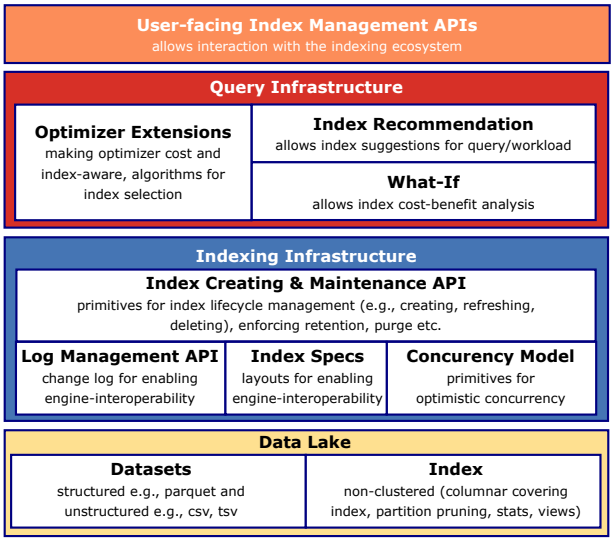


Figure 3: Overview of Hyperspace Indexing Subsystem.

3.2 Architectural Overview

Figure 3 shows an architectural overview of Hyperspace. At its core, Hyperspace offers two layers:

Indexing Infrastructure: At a bare minimum, users can utilize the indexing infrastructure (available as a *service* or a *library*) to create and maintain indexes on their data through the index creation and maintenance API (discussed in Section 3.4). Since indexes and their metadata are stored in the data lake (see Section 3.3 for justification), users can parallelize index scans to the extent that their query engine scales and their environment/business allows. Index metadata management is another important part of the indexing infrastructure. Internally, index metadata maintenance is managed by an *index manager*. The index manager takes charge of index metadata creation, update, and deletion when corresponding modification happens to the indexed data, and thus governs consistency between index and its metadata.

For developers and contributors of Hyperspace, the system also offers access to the underlying primitive components. Of particular interest are the following:

- *Log Management API:* A critical design decision we took in order to support multi-engine interoperability was to store all the indexes and their metadata in the lake. To track the lineage of the operations that take place over an index, Hyperspace records user operations in an *operation log* (Section 4.1).
- *Index Specs:* To support extensibility, Hyperspace requires certain properties from the underlying indexes. These are exposed via the lifecycle management API and anyone wanting to add an additional auxiliary data structure must implement this API.
- *Concurrency Model:* To support multi-user and incremental maintenance scenarios, we use optimistic concurrency control mechanisms (Section 4.3).

Query Infrastructure: Without loss of generality, we discuss the components of the query infrastructure implemented in the Scala version of Hyperspace. The library is written as an extension of the Spark optimizer (a.k.a. Catalyst) to make it index-aware, i.e., given a query along with existing indexes, Hyperspace-enabled Spark can perform *transparent query rewriting* to utilize the indexes. The only step required is to enable Hyperspace's optimizer extensions via

```
sparkSession.enableHyperspace()
```

after creating the Spark session. Since we treat an index as another dataset on the lake, users can exploit Spark's distributed nature to automatically scale out index scans.

Index Tuning. While Hyperspace introduces the notion of indexing, an important aspect of big data administration that critically influences performance is the ability to select indexes to build for a given query/workload. To decide the right indexes for a workload, users must be able to perform a cost-benefit analysis of the existing indexes and any "hypothetical" indexes [21] they have in mind. Hyperspace's "what if" functionality [18, 21, 32–34] allows users to quantitatively analyze the impact of existing or hypothetical indexes on performance of the system. In addition, Hyperspace also exposes an implementation of index recommendation [20, 37, 63] for automating the choice of indexes for query acceleration in big data workloads. Given certain constraints, such as the maximum number of indexes allowed and the storage budget, this utility selects the best Hyperspace indexes for an input workload in terms of percentage improvement of workload execution cost.

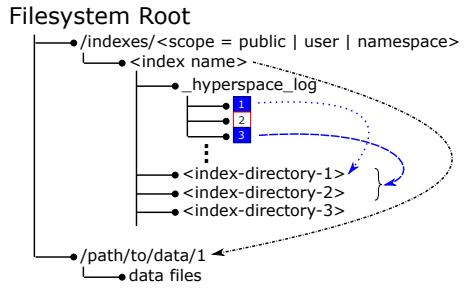


Figure 4: Index Organization on the Data Lake.

3.3 Index Organization on the Lake

To meet the design goals outlined in the introduction, we decided to store all index metadata in the lake, without any external dependencies (details in Section 4). Figure 4 shows the organization of the index metadata on the data lake file system. Each index (listed under `/indexes/*/<index name>` in Figure 4) has two components:

(1) the directory named `_hyperspace_log` that contains the operational log of the index, i.e., the list of all operations that happened on this index since its inception;
(2) the actual contents of the index.

Notice that the contents are captured through multiple directories. This is to support functionality such as concurrent index management (e.g., snapshot isolation) and incremental maintenance (e.g., the latest index is a union of the contents of multiple directories).

3.4 Usage API & Customization Knobs

Hyperspace offers a list of public APIs that can be grouped into the following categories:

Index maintenance APIs such as *create*, *delete*, *restore*, *vacuum*, *refresh*, and *cancel*. The *delete* API performs a "soft delete," which tells the optimizer to not consider this index during optimization. The actual index is not deleted, thus allowing the user to recover the deleted index using the *restore* API. Alternately, the user can

permanently delete the index (already in a soft-delete state) using the *vacuum* API. Users can cancel on-going index maintenance operations using the *cancel* API; this is useful if the user suspects that the maintenance job is stuck or has failed.

Utility APIs for debugging and recommendation such as *explain*, *whatIf*, and *recommend*. The *explain* API allows users to obtain various useful information from the optimizer, e.g., which part of the plan was modified, which indexes were chosen, why they were chosen, etc. The *whatIf* API allows users to provide the indexing subsystem with hypothetical index configurations and get an explanation of how useful it would be if the indexes in the hypothetical configuration were built. The *recommend* API allows users to get a ranked recommendation of indexes that can be built for a workload.

Storage and query optimizer configuration to allow the user to override the behavior of the query optimizer and index management. For instance, by default, every index that gets created will be discoverable. If this is not acceptable, the user can choose private index locations and namespaces to create their indexes and provide hints to the query optimizer so that it will only consider these indexes during query optimization.

4 INDEX MANAGEMENT

Our primary goal was to implement Hyperspace as a low-cost indexing subsystem that allows for concurrent index maintenance operations on an index that can be leveraged by multiple engines. In this section, we start with an important design decision taken to simplify the implementation, which was to make index management "*serverless*," i.e., there is no standalone server (in an Apache Spark cluster or more broadly in an Azure Synapse Analytics workspace) dedicated to managing indexes. We achieve this by storing all information pertaining to the index (e.g., metadata, operations on an index) in the data lake, and capturing changes to an index through an operation log. We discuss how we handle concurrent updates using optimistic concurrency control.

4.1 Foundation: Metadata in the Lake

Interoperability is complex as it requires every query engine to agree on what constitutes an index; this requires agreement between developers (and organizations/companies) working in different silo-ed ecosystems. Since the latter problem is much harder in reality, we prioritized finding a low-friction design for exposing index-related metadata (e.g., contents, state etc.) in a way that allows for easy integration. Exposing the state of an index or the list of operations invoked on an index through traditional means, such as a catalog service or a transaction manager service, guarantees strong consistency. However, this approach has a few major operational implications. First, it brings in service dependencies and live-site support overheads. Second, it makes integration complex since now every new engine has to depend on a third-party service. Finally, it introduces operational costs of running the service.

We decided to trade-off metadata consistency for easier operational maintenance, i.e., we store the ground truth of information of an index in the data lake. Figure 5(a) shows the full specification of the information we store for a given index, and Figure 5(b) provides a concrete example of an index metadata entry. The specification is divided into the following regions:

- **Contents**, which captures the type and type-specific information of the derived dataset that is useful in instantiating appropriate index interpretation logic, such as name, kind, configuration (e.g., indexed and included columns plus their types), content (e.g., physical location and layout);
- **Lineage**, which captures information required to track lineage of the derived dataset, e.g., HDFS data source being indexed, information needed to refresh the index with minimal information from the user, information needed to perform index selection, and descriptive history of an index;
- **State**, which captures state pertaining to the derived dataset, e.g., global information such as `Active` and `Disabled`, and transient information such as `Creating` and `Deleted`.

4.2 Index State Management

Index state management is at the center of serverless index management for *stateful* index operations such as index creation and deletion. Figure 6 presents the details of the index state transition diagram under the Hyperspace index manipulation APIs:

- `Creating`: When a user invokes the Hyperspace `create()` API, the index being created enters the `Creating` state; if the user cancels index creation by issuing the Hyperspace `cancel()` API, then the index goes back to the `DNE` (meaning 'Do Not Exist') state.
- `Active`: The index has been created successfully and is ready to use. Note that the index is not visible when it is in the `Creating` state until its state turns to `Active`.
- `Refreshing`: A user can refresh an existing index via the Hyperspace `refresh()` API upon data updates. However, refreshing does not block index visibility—readers can keep accessing the current active copy of the index until refreshing is finished.
- `Restoring`: A user can restore an index that has been deleted via the Hyperspace `restore()` API. Again, the index is not visible when it is in the `Restoring` state.
- `Deleting`: A user can delete an index using the `delete()` API. Recall that deletion in Hyperspace is *soft* and therefore fast. The state of the index moves from `Active` to `Deleted` after deletion and the index becomes invisible. Existing readers, however, will not be affected, similar to `Refreshing`. A deleted index can be vacuumed to free its storage space via the `vacuum()` API.
- `Optimizing`: A user can further choose to optimize the index via the Hyperspace `optimize()` API. This is an API call reserved for incremental refreshing. For example, one optimization is *index compaction*, where (small) index blocks generated incrementally will be merged into larger ones to improve index read efficiency.

Note that the states in the above list are *transitioning*, while the other states `Active`, `Deleted`, and `DNE` are *stable*.

Clearly, some index states conflict with each other. For example, if an index state is `Deleting`, `Refreshing`, or `Optimizing` (in one user session), it cannot be `Restoring` at the same time (in another concurrent user session), because the index can only move to `Deleting`, `Refreshing`, or `Optimizing` from `Active` whereas it can only enter `Restoring` from `Deleted` (as shown in Figure 6). If two Hyperspace APIs can lead to conflicting index states, then they are *incompatible*. Table 1 shows the *compatibility matrix* of Hyperspace APIs—incompatible entries are marked with 'N' whereas compatible ones are marked with 'Y.'

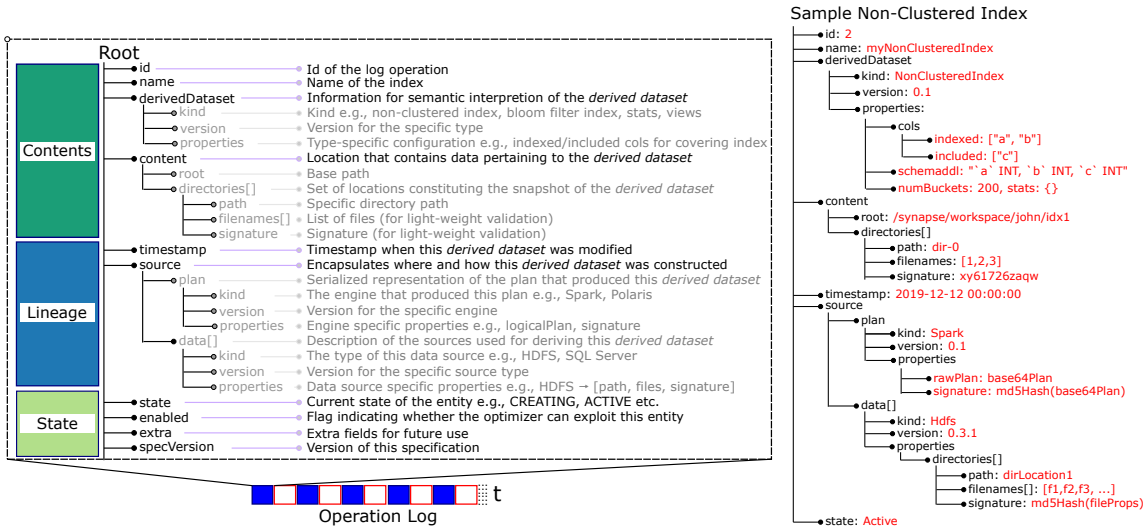


Figure 5: (a) Specification for a single metadata entry of a derived dataset (e.g., non-clustered index); (b) Sample index entry.

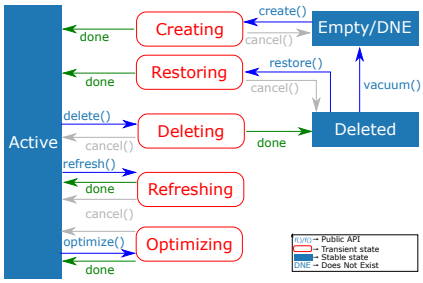


Figure 6: Index State Machine

API	C	D	O	RF	RS	V
C	Y	N	N	N	N	N
D	N	Y	Y	Y	N	N
O	N	Y	Y	Y	N	N
RF	N	Y	Y	Y	N	N
RS	N	N	N	N	Y	N
V	N	N	N	N	N	Y

Table 1: Compatibility matrix of Hyperspace index APIs: (1) 'C' for '*create*', (2) 'D' for '*delete*', (3) 'O' for '*optimize*', (4) 'RF' for '*refresh*', (5) 'RS' for '*restore*', and (6) 'V' for '*vacuum*'.

4.3 Multi-user Concurrency Control

Hyperspace allows multiple users to manipulate the same index simultaneously. Hyperspace ensures the correctness of the index (based on the index state transition diagram in Figure 6) using *optimistic* concurrency control. In the following, we provide a brief overview of the Hyperspace concurrency model.

We employ the *operation log* depicted in Figure 5(a) that supports the following log operations (Figure 7):

- `LogOp()`, which simply records the index manipulation (by a single user) that is going to happen;
- `Validate()`, which validates whether the index is in a suitable state that allows for the desired manipulation (e.g., one cannot delete an index that does not exist);

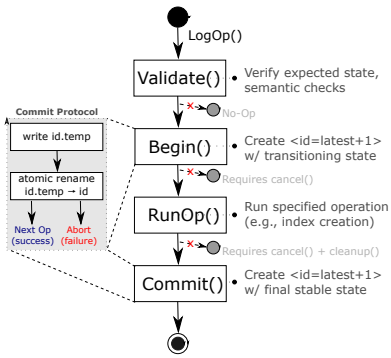


Figure 7: Log Operation

- `Begin()`, which assigns an id to the index manipulation with the corresponding transitioning state;
- `RunOp()`, which records that the manipulation is now running;
- `Commit()`, which records the id of the finished index manipulation with the corresponding final stable state.

Hyperspace relies on the *atomicity* of renaming a file in a cloud file system (including HDFS, Azure Storage, or Azure Data Lake) to ensure that altering index state from a transitioning state to a stable state during `Commit()` is atomic. If during `Commit()` Hyperspace detects that the corresponding file that stores the index transitioning state has been renamed, Hyperspace can simply *abort* the ongoing index manipulation. The user can choose to *retry* the index manipulation upon receiving an abort message.

Hyperspace currently supports *multiple writers* using the above concurrency control mechanism, and *multiple readers*. Readers simply choose a stable snapshot of the index data that has been committed. To ensure consistency between the index and the corresponding data being indexed, Hyperspace further employs a signature-based mechanism (see Figure 5(a)) where the latest timestamps of the data files are used to generate a *signature* for the index. During query processing (see Section 6), it is possible that a query may touch a table as well as its indexes simultaneously. Using the signature of the data that is stored in the index metadata, the optimizer

can ensure that the index is not stale. However, ensuring *external consistency* remains as a challenge, as Hyperspace is a serverless indexing service that does not manage the data. For example, it is possible that a query is accessing data that has been updated since it validated the consistency of the index, as Hyperspace does not hold any lock over the data during query execution. External consistency is beyond the scope of functionalities that Hyperspace currently supports. Some coordination mechanism across data lake users is required to ensure external consistency. One naive approach could be to enforce that each user has their own workspace and cannot access other users' workspaces. This prevents inconsistency but also restricts any possible data sharing.

5 INCREMENTAL INDEX MAINTENANCE

Hyperspace enables index maintenance over mutable datasets regardless of the underlying data format. In this section, we discuss the use cases and incremental index maintenance features.

5.1 Mutable Datasets and Semantics

Enterprise datasets get modified over time. The following are common patterns we have observed with our customers:

(1) **Streaming ingestion.** Teams that need to deploy data streaming pipelines [41, 64] would collect data into data lakes partitioned by a certain key, e.g., timestamp, writer-id, etc. This data is then consumed by downstream pipelines and modified to fit their needs. More frequently, a number of users query these raw datasets in ways that force linear scans.
(2) **GDPR Compliance.** To comply with data privacy regulations such as GDPR [42], enterprises have to delete data corresponding to a particular user, when requested to do so (typically through features like Forget-Me [42]).
(3) **Data corrections.** Data may need to be updated due to errors in data collection or late-arriving data or simply user satisfaction (e.g., in cases where users are charged due to a software bug, it is typical in cloud providers to issue refunds).

Typically, each data team has their own mechanisms to apply these updates to data lake datasets ranging from maintaining custom secondary indexes to blobs that are then used to update appropriate portions to using more recently introduced ACID table formats [27, 28, 30]. Since in-place update semantics are not so common [11], nearly all solutions to handling updates use some form of journaling (e.g., versioning, delete-and-append, etc.).

5.2 Index REFRESH & OPTIMIZE

Hyperspace captures detailed metadata of the underlying data source at the time it indexes them. This includes information such as filename, modified time, and file size. While appends to original datasets can be supported without any special metadata, to handle deletes, users need to create an index with lineage tracking enabled that tracks which source file a particular record originates from.

To accommodate a variety of enterprise workloads, Hyperspace supports multiple index maintenance modes captured in Table 2.

Full Refresh. The simplest mode is a full rebuild of the index that re-scans the original data to build a new version of the index. This is particularly efficient if the underlying source data is relatively stable and the index is being heavily used. We observed this mode being typically used by customers who capture *snapshots* of their data in the data lake. In these cases, the underlying data is entirely rewritten to the point where incremental indexing is not very beneficial (as it anyways rewrites the entire index).

Incremental Refresh. If the user is infrequently appending/deleting large amounts of data to the underlying source, it is beneficial to use the incremental refresh mode, where Hyperspace operates as follows. First, Hyperspace scans the underlying source and records the list of appended and deleted files/partitions by comparing it against the lineage information recorded in its metadata (recall that Hyperspace stores the filename, last modified timestamp and file size at the time of index creation). Second, for the appended files, Hyperspace starts an indexing job to sort only that portion of the data. To handle deletes, it utilizes index lineage to detect which portions of the data these affected index blocks were generated from and simply rewrites those index blocks. Third, all the new index blocks are committed as a new incremental version of the index. Finally, the metadata is updated to reflect the latest snapshot.

Quick Refresh. If the user frequently appends small amounts of data to the underlying source, it may not be cost-efficient to keep running indexing jobs since the benefit from partially sorting this newly added data may be negligible. For such cases, Hyperspace supports a quick refresh mode where it simply scans for the list of files appended/deleted and updates the metadata to capture this information. At query time, Hyperspace resorts to its *hybrid scan* operator (see Section 6.2) to derive the latest index.

Optimize. If a user invokes incremental or quick refresh too often, the index may get fragmented (similar to indexes in traditional DBMS systems). To handle this scenario, Hyperspace allows users to optimize their indexes by compacting smaller index files based on user-provided criterion (e.g., compact files less than 10 MB).

5.3 Support for ACID Data Formats

An inherent complexity inside Hyperspace is caused by the mechanisms to determine if the underlying data source is up-to-date and if it is not, determining what changed since the time the data source was indexed. Recent systems such as Linux Foundation's Delta Lake [30], Apache Hudi [27], and Apache Iceberg [28] have defined data formats and access protocols to implement transactional operations on cloud object stores. These systems maintain a transaction log that records all operations that have taken place on the data being managed. When indexing such data formats, Hyperspace can exploit the respective transaction logs to skip some expensive metadata checks. For instance, when Hyperspace is indexing data in Delta Lake format, it relies on the latter's transaction log to quickly obtain the list of partitions/files that have been added/removed and updates the Hyperspace log appropriately.

6 QUERY PROCESSING USING INDEXES

During query optimization, Hyperspace explores potential benefits of leveraging indexes by making the query optimizer "index-aware." For example, for the case of Apache Spark, Hyperspace has incorporated new optimizer rules into Catalyst — Spark's rule-based query optimizer [23]. In this section, we present the design and implementation of this "index-aware" query optimizer extension. We also

		Full Rebuild	Quick Query	Fast Refresh
Append	**Characteristic**	Slowest refresh/fastest query	Slow refresh/fast query	Fast refresh/moderately fast query
	API	**refreshIndex(mode="full")**	**refreshIndex(mode="incremental")**	**refreshIndex(mode="quick")**
	What it does?	Rebuilds the index	Builds index on newly added data	Captures metadata for appended files and leverages hybrid scan
	When to use?	Underlying source data is relatively stable	Infrequently appending large amounts of data	Frequently appending small amounts of data
Delete	**Characteristic**	Creates a new index (by reshuffling the source data)	Slow refresh/fast query	Fast refresh/moderately fast query
	API		**refreshIndex(mode="incremental")**	**refreshIndex(mode="quick")**
	What it does?		Drops rows from index immediately; Avoids shuffling the source data using index lineage	Captures file/partition predicates and deletes entries at query time
	When to use?		Infrequently deleting large amount of data	Frequently deleting small amounts of data
Optimize			**Faster Optimize Speed**	**Slower Optimize Speed**
	API		**optimizeIndex(mode="quick")**	**optimizeIndex(mode="full")**
	What it does?		Best-effort merge of small index files within a partition; DOES NOT refresh the index	Create a single file per partition by merging small & large files; DOES NOT refresh the index
	When to use?		When perf starts degrading	When perf starts degrading

Table 2: Index maintenance modes. Since workloads vary by requirements, Hyperspace supports multiple index maintenance mechanisms to allow customers to choose the options that will help them meet their business SLAs.

propose *hybrid scan*, a novel access path that allows Hyperspace to utilize an index for query processing even if it is not up to date.

6.1 Index-aware Optimizer Extension

6.1.1 A Generic Framework. Figure 8 presents the architecture of the query optimizer extension made by Hyperspace. The input to optimizer is now the query *and* the set of available *candidate* indexes. The optimizer then considers all indexes from the given candidates that could be utilized to accelerate query performance to select the best ones. To fulfill this, Hyperspace employs *index strategy* to perform index selection. The index strategies are customized for different query optimizers, and we will present their implementations in Apache Spark shortly. After applying the index strategies, the resulting query plans (with indexes) are sent to the *plan ranker*, which will compare/rank the plans and pick the best one (e.g., with the lowest execution cost). The estimation of the execution cost of a plan is delegated to a component called "what if" utility that is shared with the index recommender. The "what if" utility [21] is an extended version of the usual query optimizer cost model that takes index access costs into consideration. It can take both actual indexes (as in Figure 8) and *hypothetical* indexes as in the context of index recommendation [20].

6.1.2 Hyperspace Index Strategies in Spark. Since Catalyst is a rule-based query optimizer, Hyperspace implements two index strategies as optimizer rules, **FilterIndexRule** and **JoinIndexRule**, that target accelerating filter and join operators in Spark query execution plans using Hyperspace indexes. Hyperspace indexes may also be beneficial for other operators, such as aggregates on top of group-bys, which could be interesting directions for future work.

Indexing Rule for Filters. The **FilterIndexRule** works as follows. If a table scan has a filter f on top of it, we replace it by a Hyperspace index I if the following conditions meet:

- The *leading* column in the indexed (key) columns of I is referenced by *some* predicate in f;
- All columns referenced by predicates in f are covered by I, i.e., appear in either the indexed or included columns of I.

We implement the condition checking using *pattern matching* [8].

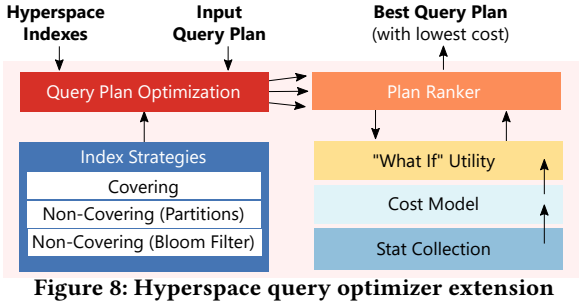


Figure 8: Hyperspace query optimizer extension

Indexing Rule for Joins. The **JoinIndexRule** works in a similar manner by looking for candidate indexes via pattern matching. However, unlike the **FilterIndexRule**, we are not able to match a specific pattern except for merely matching individual join operators. We then examine that for each matched join operator, it satisfies the *equi-join condition*, i.e., the join condition is restricted to be a *conjunction* of equality predicates between join columns.

After matching an eligible join operator O with join condition c, the next step is to find usable indexes for O. Given that both the left and right sub-plans are linear, we only have two base tables in the plan tree under O. For each base table T, we then check the following conditions for each candidate index I on top of T:

- All join columns in T that are referenced by c should be the same as the indexed columns of I;
- All other columns referenced by the left or right sub-plan that accesses T are contained by the included columns of I.

Let $\mathcal{I}_l$ and $\mathcal{I}_r$ be the candidate indexes found for the left and right sub-plan, respectively. If more than one index pair $(I_l, I_r) \in (\mathcal{I}_l, \mathcal{I}_r)$ exists, Hyperspace currently picks the one that would result in the least execution cost based on the following criteria:

(1) If there exist index pairs (I_l, I_r) such that I_l and I_r have the same number of partitions, then pick the index pair with the *largest* number of partitions.
(2) Otherwise, if no such index pair exists, then pick an arbitrary index pair from the eligible pairs.

The rationale is the following. First, when two indexes have the same number of partitions, there is no shuffling when performing the (sort-merge) join; if the number of partitions differ, one index

gets reshuffled into the number of partitions equal to the other. Second, in general more number of partitions can lead to better parallelism in join execution, assuming no resource constraint.

Finally, **JoinIndexRule** replaces table scans with corresponding index scans on top of the best index pairs found.

6.2 Hybrid Scan

Since refreshing an index is modeled as an indexing job (so it can scale), it may not always be beneficial to keep running refresh jobs (e.g., when only a small amount of data is appended/deleted). However, if an index is not refreshed, Hyperspace will detect that the index has gone stale since the underlying data has changed. For most of our customers, this was quite unreasonable (in fact, some have asked that we allow using a stale index as it provides significant acceleration). To circumvent these problems, Hyperspace provides *hybrid scan* that allows users to leverage indexes even when they have gone stale.

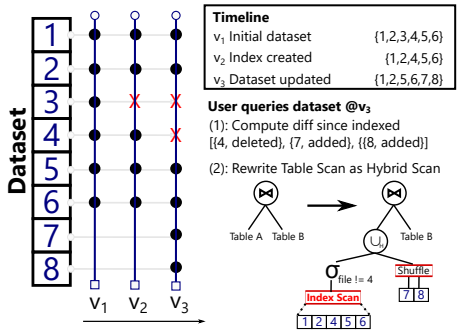


Figure 9: Hyperspace Hybrid Scan. $\bigcup_H$ is a special partition-aware physical union operator implemented in Hyperspace.

The key idea behind the *hybrid scan* technique is to utilize the existing index and then apply the changes observed in the underlying data source. At query time, if Hyperspace detects that the index has gone stale, it first obtains a list of files/partitions that have been appended/deleted from the underlying data source. Next, it modifies the query plan to exclude the deleted rows and "merge" the appended data in the following way (see Figure 9):

- *Handling Appends.* Since indexes are hash partitioned, any newly appended files should be partitioned the same way and then merged (i.e., union-ed) with the existing index data to avoid a full shuffle. To do this, we introduce a new partition-aware union operator, which detects that the two sides are partitioned, sorts within partitions, and performs a merge. Note that the sort is cheap since data is mostly sorted and we use Timsort [16], which has linear running time for mostly sorted data.
- *Handling Deletes.* Since we capture lineage information in indexes (i.e., which file a particular row originates from), the hybrid scan operation can modify the query plan to introduce a *filter* that eliminates all rows belonging to the files that have been deleted.

While hybrid scan allows users to exploit stale indexes, its performance is dependent on the extent of staleness of the index, i.e., if the underlying dataset has undergone too many changes, hybrid scan might end up causing regressions. To alleviate this issue, we have user-configurable properties that allow the optimizer to disable hybrid scan when the staleness has exceeded a certain threshold. We leave automatically determining this aspect to future work.

Compute Cluster Settings					
Type	**Nodes**	**VM**	**Cores**	**RAM**	**SSD**
Driver	2	D12V2	8	28 GB	200 GB
Worker	16	D14V2	256	112 GB	800 GB
Storage Settings					
Type	WASB Storage (general purpose v1)				
Repl	Locally-redundant storage replication (LRS)				

Table 3: Compute and storage characteristics

Support of "Time Travel". As described in Section 5.3, Hyperspace supports ACID data formats such as Linux Foundation's Delta Lake [30]. An interesting feature offered by such data format is the ability to "time travel," i.e., users can query point-in-time snapshots or roll back erroneous updates to their data. Time travel brings interesting challenges to index management. For instance, during optimization, Hyperspace should be able to detect that version of the underlying dataset that is being queried and should decide the best index to use (recall that Hyperspace supports multiple snapshots of indexes, i.e., every time the user invokes a full refresh, there is a new version of the index that is generated). Hybrid scan forms the core backbone for supporting data formats offering time travel. Once the *best* index is determined, Hyperspace utilizes hybrid scan to reconstruct the latest state by comparing the transaction logs.

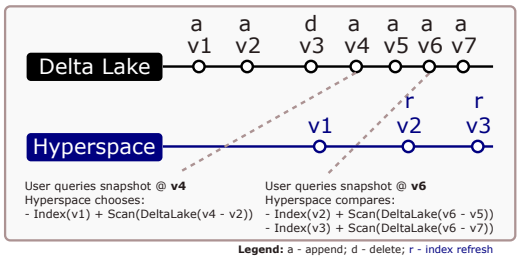


Figure 10: Hyperspace support for Time-travel Queries. Since multiple index snapshots can be used, Hyperspace compares the cost of hybrid scan over those snapshots.

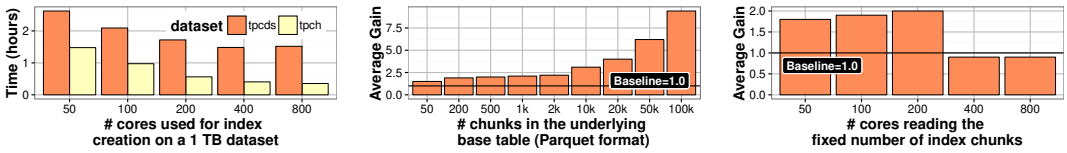
Figure 10 shows an example of how Hyperspace co-exists with ACID data formats. In formats like Delta Lake, every time a user appends (denoted by *a*) or deletes (denoted by *d*) files, a new version is created. Subsequently, user can specify the version of the dataset they want to time-travel to during query time. When Hyperspace is enabled, it determines the list of indexes that are applicable to this Delta Lake source and relies on simple cost comparisons (e.g., number of files linearly scanned for a specific index version) to pick the best index. For instance, if the user queries the data snapshot $v4$, Hyperspace has only one option whereas if they choose to query snapshot $v6$, Hyperspace performs the appropriate cost comparison between using its own index at $v2$ and $v3$.

7 EVALUATION

We evaluate the performance of Hyperspace using both industry benchmarks and real customer workloads. We start by discussing index creation time, present the gains achieved for queries over each of the datasets, and finally study the utility of indexes.

7.1 Experimental Environment

We present results obtained using Apache Spark [15, 66] for both index creation and utilization. Our experimental setup has a combination of D12V2 and D14V2 instances in Azure. The details of the



(a) (b) (c)

Figure 11: (a) Index creation time; (b) Performance gains on TPC-H with respect to increasing number of base table chunks; (c) Performance gains on TPC-H with respect to fixed number of index partitions (i.e., 200) and increasing number of CPU cores.

Dataset	CSV Size	Parquet Size	Index Count	Index Size
TPC-H	1.2 TB	391 GB	16	691 GB
TPC-DS	1.0 TB	510 GB	37	1315 GB
Customer	60 GB	18 GB	13	52 GB

Table 4: Dataset and Hyperspace index characteristics

memory and storage configuration of each of these VM instances are provided in Table 3; D12v2 instances power the job submission endpoint while D14v2 instances are used for running the Spark driver and workers. We store all our datasets on Azure Storage (WASB [11]) configured for locally-redundant storage (LRS) replication. We pre-generated data for two standard industry benchmarks, **TPC-H** (22 queries) [10] and **TPC-DS** (99 queries) [9], and a real-world customer workload **Customer** (15 queries). Compared to **TPC-H** and **TPC-DS**, the **Customer** workload is relatively simple as its' queries contain fewer joins. For all datasets, we also pre-generate the necessary indexes output by our index recommendation engine. The details of these datasets along with their index configurations are provided in Table 4. We run each query five times consecutively, discard the first iteration (to warm up the storage cache), and average the remaining runs.

7.2 Index Creation Time

We begin by answering the simplest question raised by the majority of our customers: *How long does it take to create Hyperspace indexes*? Figure 11(a) shows the index creation time for **TPC-H** (16 indexes) and **TPC-DS** (50 indexes) when varying the numbers of CPU cores. The indexes are selected based on the index recommendation component of Hyperspace. An index creation job reads the underlying base tables, performs any necessary projections and shuffles, and sorts the output based on the key columns determined by the index recommendation algorithm. Based on the results, we observe a linear speedup with increasing number of CPU cores.

We note that index creation remains an area in Hyperspace that is worth further improvement, since it requires scanning the underlying table once, repartitioning/shuffling the data based on the indexed columns (for generating the buckets), and then writing the index to cloud storage (Azure Storage in our case). We leave this as one interesting direction for future work.

7.3 Query Execution Performance

Figure 12 shows the gain achieved for each query from the three workloads. Overall, we observe gains of 75.0% for **TPC-H**, 50.6% for **TPC-DS**, and 89.9% for **Customer** when the underlying datasets are in CSV format, and 39.8% for **TPC-H**, 50.5% for **TPC-DS**, and 43.6% for **Customer** when the underlying base table datasets are in Parquet format. We now deep dive into two representative TPC-H queries to explain where the gains come from.

(TPC-H Q6) It scans the `lineitem` table, applies range selection predicates on `l_shipdate`, `l_discount`, and `l_quantity`, and performs an aggregation on a projection of a few columns to calculate revenue. When scanning `lineitem`, Spark pushes all predicates down into the Parquet file reader. Without Hyperspace indexes, the pushed-down filters are ineffective as Spark has to fetch all row groups, as the records are randomly arranged in Parquet files. With Hyperspace, Spark can use an index on `l_shipdate` that helps skip fetching more than 67% of data, resulting in 2x to 3x speedup.

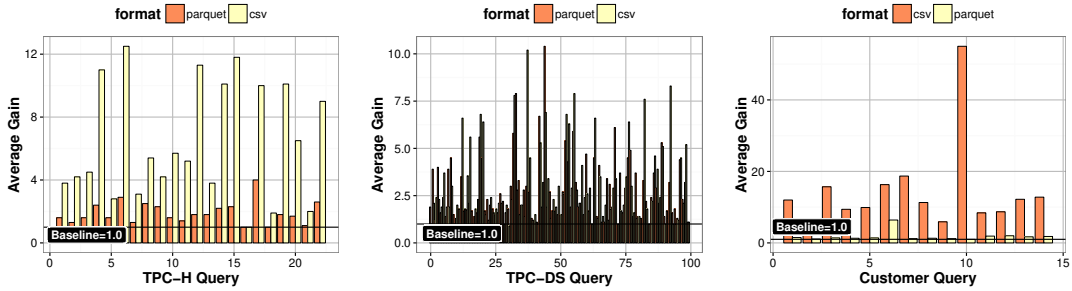
(TPC-H Q17) The `lineitem` table is first filtered and aggregated to find average `l_quantity` per `l_partkey` (AGG). It then joins `lineitem` with filtered `part` on `l_partkey` and `p_partkey` (JOIN). The results are further joined with filtered AGG results on `part_key`. Spark chose to use a shuffled hash-join for JOIN; as a result the tables need to be shuffled on `l_partkey` and `p_partkey`. Although the filter on `part` is highly selective, all records of `lineitem` – the largest table – are shuffled (179 GB). AGG computation is done through hash aggregation, which includes shuffling partially aggregated `lineitem` records on `l_partkey` before the global aggregation. This is another large shuffle (168 GB) during query execution. With Hyperspace, for both AGG and JOIN Spark exploits Hyperspace indexes on `lineitem` and `part` to eliminate the shuffles.

7.4 Hybrid Scan

While Hyperspace supports incremental refresh, one of the most popular customer requests has been to allow exploiting indexes to the extent possible. However, allowing stale indexes will lead to correctness issues in production and unexpected complaints from customers who may not fully understand why they are being served results from stale indexes. To address this, Hyperspace implements *hybrid scan* that allows exploiting a stale index to the extent possible and then adjusts for any appends/deletes to the underlying datasets. By far, the most frequently asked question was: *is hybrid scan performant? how can one understand its characteristics?*.

We study these question by simulating a real-world customer workload. Our customer has a large table into which they stream updates (which include appends and deletes). Deletes are quite infrequent (few batches a day) but appends are frequent (thousands of files being added into the dataset daily). To simulate this workload, we use the `lineitem` from TPC-H as the large table and run all the 22 queries as we continue to maintain (i.e., append/delete) it. As we perform this experiment, there are four situations to consider: our baseline performance *without indexes*, performance with *partial indexes* (i.e., since the dataset was updated, Hyperspace will invalidate any indexes built on `lineitem`), performance with *hybrid scan*, and finally, performance with *indexes fully refreshed*.

Figure 13 shows the performance of hybrid scan in comparison to the baseline, partial indexes, and fully up-to-date indexes. Each



(a) (b) (c)

Figure 12: Average gain for (a) 1 TB TPC-H workload; (b) 1 TB TPC-DS workload; (c) Customer workload.

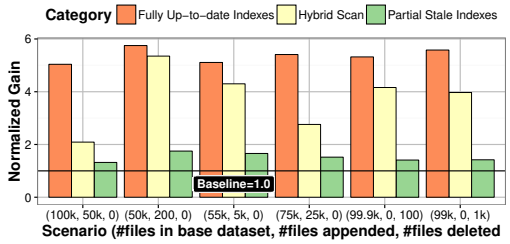


Figure 13: Efficiency of Hyperspace's Hybrid Scan

of the tuples of the form (a, b, c) in the x-axis denotes the number of files in the base dataset, number of files appended and deleted respectively. Since the base dataset contains a large number of files, Hyperspace offers significant acceleration (as high as 5x) as can be observed from the *fully up-to-date indexes* bar. Notice, however, that the performance of *partial indexes* approaches the baseline since Hyperspace is unable to utilize the most valuable indexes. There are several observations one can make from the performance of *hybrid scan*. First, when the number of updates is small relative to the number of files in the base dataset, e.g., (50k, 200, 0), (99.9k, 0, 100), and (99k, 0, 1k), the performance of hybrid scan is close to that obtained using fully refreshed indexes. The performance starts deteriorating when the number of files appended starts increasing significantly, as in the case of $(75l, 25k, 0)$. In the worst case, e.g., more than 50% data got appended as in $(100k, 50k, 0)$, the performance approaches that of the partial indexes (but still better).

8 RELATED WORK

Indexing is a standard performance acceleration technique in classic database systems. Examples include tree-based techniques (e.g., B-tree [22], LSM tree [50], R-tree [31], quad-tree [58]), hash-based techniques (e.g., extendible hash [26], linear hash [39]), and bitmap-based techniques [48], among others. In the past decade, column-stores [60] have become increasingly popular, especially for read-heavy OLAP workloads. Major database vendors such as Microsoft SQL Server [6] now provide column-stores as an alternative indexing technique in addition to traditional ones, e.g., B+ tree or hash indexes. Our design of Hyperspace indexes is inspired from both traditional and column-store indexes, with adaptations to big data query processing systems (e.g., partitioning on indexed columns).

Indexing support in modern big data systems remains limited, especially for OLAP-style data analytics workloads. Recently, Delta Lake [14] proposed using Z-order [47] to optimize for queries with multi-dimensional range filtering predicates. by requiring that the table itself be re-organized. Hyperspace instead supports Z-Order secondary indexes [43] which allows users to build several such indexes for different query patterns. Needless to say, this would raise new challenges in index recommendation when choosing between Z-order and other types of indexes. Moreover, our incremental index maintenance and hybrid query processing mechanisms are inspired from Helios [52], which targets indexing massive data injected into the cloud from the edge in a timely manner.

In the context of Spark, recent work by Uta et al. [62] showcases the use of in-memory cTrie indexes [53] to accelerate graph processing queries. GeoSpark [65] considers indexing support (e.g., R-tree, quad-tree, and KDB-tree) for querying spatial data. Hyperspace is in many ways complementary to these efforts in that we are not proposing new index structures but an extensible framework that can support these different types of indexes.

There have also been various other techniques that aim for accelerating query performance in big data and/or data lake systems. For example, data partitioning and partition pruning [40, 49], data shuffling [54], as well as materialized views and view selection [35, 36]. Hyperspace is complementary to these technologies by focusing on the indexing and index management aspect. There has also been much work devoted to *polystores* [25] in recent years where data is located on multiple heterogeneous data stores, where cross-platform query processing techniques have been developed (e.g., [12, 38]). It will be interesting to see how Hyperspace indexes could be further applied in polystore scenarios given that Hyperspace was designed with multi-engine use cases in mind.

9 CONCLUSION

We have presented Hyperspace [43, 44, 55, 56], the indexing subsystem of the Azure Synapse Analytics service from Microsoft. In the spirit of heterogeneity in data lakes, our goal is to democratize indexes and their management by storing all index-related data/metadata in the lake, allowing users to reuse them across multiple engines. We discussed Hyperspace's indexing infrastructure, its incremental maintenance mechanisms, and its query processing architecture. We presented evaluation results that show it improves query execution by 2x to 10x for complex workloads. Compared to existing indexing technologies in the big data world, Hyperspace offers not only query performance acceleration but also an automated lifecycle management framework that significantly improves user experience related to index usage.

REFERENCES

[1] [n.d.]. Apache Parquet. https://parquet.apache.org/.
[2] [n.d.]. Azure Synapse Analytics. https://azure.microsoft.com/en-us/services/synapse-analytics/.
[3] [n.d.]. Bucketing in Spark. https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-bucketing.html.
[4] [n.d.]. Deep Dive into GPU Support in Apache Spark 3.x. https://databricks.com/session_na20/deep-dive-into-gpu-support-in-apache-spark-3-x.
[5] [n.d.]. FPGA-Based Acceleration Architecture for Spark SQL. https://databricks.com/session/fpga-based-acceleration-architecture-for-spark-sql.
[6] [n.d.]. Microsoft SQL Server Columnstore Indexes. https://docs.microsoft.com/en-us/sql/relational-databases/indexes/columnstore-indexes-overview?view=sql-server-ver15.
[7] [n.d.]. Parquet MR. https://github.com/apache/parquet-mr.
[8] [n.d.]. Scala Pattern Matching. https://docs.scala-lang.org/tour/pattern-matching.html.
[9] [n.d.]. The TPC-DS Benchmark. http://www.tpc.org/tpcds/.
[10] [n.d.]. The TPC-H Benchmark. http://www.tpc.org/tpch/.
[11] [n.d.]. Windows Azure Storage Blob (WASB). https://gerardnico.com/azure/wasb.
[12] Divy Agrawal, Sanjay Chawla, Bertty Contreras-Rojas, Ahmed K. Elmagarmid, Yasser Idris, et al. 2018. RHEEM: Enabling Cross-Platform Data Processing - May The Big Data Be With You! -. *Proc. VLDB Endow.* 11, 11 (2018), 1414–1427.
[13] Josep Aguilar-Saborit, Raghu Ramakrishnan, Krish Srinivasan, Kevin Bocksrocker, Ioannis Alagiannis, Mahadevan Sankara, Moe Shafiei, Jose Blakeley, Girish Dasarathy, Sumeet Dash, et al. 2020. POLARIS: the distributed SQL engine in azure synapse. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3204–3216.
[14] Michael Armbrust, Tathagata Das, Sameer Paranjpye, Reynold Xin, Shixiong Zhu, Ali Ghodsi, Burak Yavuz, Mukul Murthy, Joseph Torres, Liwen Sun, Peter A. Boncz, Mostafa Mokhtar, Herman Van Hovell, Adrian Ionescu, Alicja Luszczak, Michal Switakowski, Takuya Ueshin, Xiao Li, Michal Szafranski, Pieter Senster, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proc. VLDB Endow.* 13, 12 (2020), 3411–3424.
[15] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *Procs. of SIGMOD*. ACM.
[16] Nicolas Auger, Cyril Nicaud, and Carine Pivoteau. 2015. Merge strategies: from merge sort to Timsort. *URL https://hal-upec-upem. archives-ouvertes. fr/hal-01212839, working paper or preprint* (2015).
[17] Azure. 2021. Synapse Link. https://docs.microsoft.com/en-us/azure/cosmos-db/synapse-link.
[18] Lorenzo Baldacci and Matteo Golfarelli. 2019. A Cost Model for SPARK SQL. *IEEE Trans. Knowl. Data Eng.* 31, 5 (2019), 819–832.
[19] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. 2011. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *SOSP*.
[20] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*. 146–155.
[21] Surajit Chaudhuri and Vivek R. Narasayya. 1998. AutoAdmin 'What-if' Index Analysis Utility. In *SIGMOD*. 367–378.
[22] Douglas Comer. 1979. The Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (1979), 121–137.
[23] DataBricks. 2015. Deep Dive into Spark SQL's Catalyst Optimizer. https://goo.gl/GZtSWC.
[24] Dremio. 2020. Reflections. https://docs.dremio.com/acceleration/reflections.html.
[25] Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, Magdalena Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stanley B. Zdonik. 2015. The BigDAWG Polystore System. *SIGMOD Rec.* 44, 2 (2015), 11–16.
[26] Ronald Fagin, Jürg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. 1979. Extendible Hashing - A Fast Access Method for Dynamic Files. *ACM Trans. Database Syst.* 4, 3 (1979), 315–344.
[27] Apache Foundation. 2020. Apache Hudi. https://github.com/apache/hudi.
[28] Apache Foundation. 2020. Apache Iceberg. https://iceberg.apache.org/spark/.
[29] Apache Foundation. 2021. Apache Spark. https://github.com/apache/spark.
[30] Linux Foundation. 2020. Delta Lake. https://github.com/delta-io/delta.
[31] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*. 47–57.
[32] Herodotos Herodotou. 2011. Hadoop Performance Models. *CoRR* abs/1106.0940 (2011).
[33] Herodotos Herodotou and Shivnath Babu. 2011. Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. *PVLDB* 4, 11 (2011), 1111–1122.
[34] Herodotos Herodotou and Shivnath Babu. 2013. A What-if Engine for Cost-based MapReduce Optimization. *IEEE Data Eng. Bull.* 36, 1 (2013), 5–14.
[35] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. 2018. Selecting Subexpressions to Materialize at Datacenter Scale. *Proc. VLDB Endow.* 11, 7 (2018), 800–812.
[36] Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc Friedman, Yifung Lin, Konstantinos Karanasos, and Sriram Rao. 2018. Computation Reuse in Analytics Job Service at Microsoft. In *SIGMOD*. 191–203.
[37] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. 2020. Magic mirror in my hand, which is the best in the land? An Experimental Evaluation of Index Selection Algorithms. *Proc. VLDB Endow.* 13, 11 (2020), 2382–2395.
[38] Jeff LeFevre, Rui Liu, Cornelio Inigo, Lupita Paz, Edward Ma, Malú Castellanos, and Meichun Hsu. 2016. Building the Enterprise Fabric for Big Data with Vertica and Spark Integration. In *SIGMOD*. 63–75.
[39] Witold Litwin. 1980. Linear Hashing: A New Tool for File and Table Addressing. In *VLDB*. 212–223.
[40] Yi Lu, Anil Shanbhag, Alekh Jindal, and Samuel Madden. 2017. AdaptDB: Adaptive Partitioning for Distributed Joins. *Proc. VLDB Endow.* 10, 5 (2017), 589–600.
[41] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanan Muthukrishnan, Vamsi Kuppa, et al. 2018. Chi: A scalable and programmable control plane for distributed stream processing systems. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1303–1316.
[42] Microsoft. 2017. GDPR Compliance. *https://goo.gl/2KkwMv* (2017).
[43] Microsoft. 2020. Hyperspace. https://github.com/microsoft/hyperspace.
[44] Microsoft. 2020. Hyperspace in Azure Synapse Analytics. https://aka.ms/synapse/hyperspace.
[45] Microsoft. 2020. Time Series Insights. https://azure.microsoft.com/en-us/services/time-series-insights/.
[46] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *VLDB*. 476–487.
[47] G. M. Morton. 1966. A computer oriented geodetic data base; and a new technique in file sequencing. *IBM Technical Report* (1966).
[48] Patrick E. O'Neil and Dallan Quass. 1997. Improved Query Performance with Variant Indexes. In *SIGMOD*. 38–49.
[49] Laurel J. Orr, Srikanth Kandula, and Surajit Chaudhuri. 2019. Pushing Data-Induced Predicates Through Joins in Big-Data Clusters. *Proc. VLDB Endow.* 13, 3 (2019), 252–265.
[50] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* (1996).
[51] Rahul Potharaju. 2021. NVIDIA GPU Acceleration for Apache Spark™ in Azure Synapse Analytics. *http://aka.ms/synapse-spark-gpu* (2021).
[52] Rahul Potharaju, Terry Kim, Wentao Wu, Vidip Acharya, Steve Suh, Andrew Fogarty, Apoorve Dave, Sinduja Ramanujam, Tomas Talius, Lev Novik, and Raghu Ramakrishnan. 2020. Helios: Hyperscale Indexing for the Cloud & Edge. *Proc. VLDB Endow.* 13, 12 (2020), 3231–3244.
[53] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. 2012. Concurrent tries with efficient non-blocking snapshots. In *PPOPP*. 151–160.
[54] Shi Qiao, Adrian Nicoara, Jin Sun, Marc Friedman, Hiren Patel, and Jaliya Ekanayake. 2019. Hyper Dimension Shuffle: Efficient Data Repartition at Petabyte Scale in Scope. *PVLDB* 12, 10 (2019), 1113–1125.
[55] Eunjin Song Rahul Potharaju, Terry Kim. 2021. Hyperspace for Delta Lake. *https://aka.ms/sais2021-hyperspace-for-delta-lake* (2021).
[56] Terry Kim Rahul Potharaju. 2020. Hyperspace: An Indexing Sub-system for Apache Spark. *https://aka.ms/sais2020-hyperspace* (2020).
[57] Raghu Ramakrishnan, Baskar Sridharan, John R Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, et al. 2017. Azure Data Lake Store: A Hyperscale Distributed File Service for Big Data Analytics. In *Procs. of ICMD*. ACM, 51–63.
[58] Hanan Samet. 1989. Hierarchical Spatial Data Structures. In *SSD*. 193–212.
[59] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *MSST*.
[60] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. 2005. C-Store: A Column-oriented DBMS. In *VLDB*. 553–564.
[61] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1626–1629.
[62] Alexandru Uta, Bogdan Ghit, Ankur Dave, and Peter A. Boncz. 2019. [Demo] Low-latency Spark Queries on Updatable Data. In *SIGMOD*. 2009–2012.
[63] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. 2000. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *ICDE*. 101–110.
[64] Le Xu, Shivaram Venkataraman, Indranil Gupta, Luo Mai, and Rahul Potharaju. 2021. Move Fast and Meet Deadlines: Fine-grained Real-time Stream Processing with Cameo. In *NSDI*.
[65] Jia Yu, Zongsi Zhang, and Mohamed Sarwat. 2019. Spatial data management in apache spark: the GeoSpark perspective and beyond. *GeoInformatica* 23, 1 (2019).
[66] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*. 15–28.