Helios: Hyperscale Indexing for the Cloud & Edge

Rahul Potharaju, Terry Kim, Wentao Wu, Vidip Acharya, Steve Suh, Andrew Fogarty, Apoorve Dave, Sinduja Ramanujam, Tomas Talius, Lev Novik, Raghu Ramakrishnan

Microsoft Corporation

{rapoth, terryk, wentwu, viachary, stsuh, anfog, apdave, sindujar, tomtal, levn, raghu}@microsoft.com

ABSTRACT

Helios is a distributed, highly-scalable system used at Microsoft for flexible ingestion, indexing, and aggregation of large streams of real-time data that is designed to plug into relational engines. The system collects close to a quadrillion events indexing approximately 16 trillion search keys per day from hundreds of thousands of machines across tens of data centers around the world. Helios use cases within Microsoft include debugging/diagnostics in both public and government clouds, workload characterization, cluster health monitoring, deriving business insights and performing impact analysis of incidents in other large-scale systems such as Azure Data Lake and Cosmos. Helios also serves as a reference blueprint for other large-scale systems within Microsoft. We present the simple data model behind Helios, which offers great flexibility and control over costs, and enables the system to asynchronously index massive streams of data. We also present our experiences in building and operating Helios over the last five years at Microsoft.

PVLDB Reference Format:
Rahul Potharaju, Terry Kim, Wentao Wu, Vidip Acharya, Steve Suh, Andrew Fogarty, Apoorve Dave, Sinduja Ramanujam, Tomas Talius, Lev Novik, Raghu Ramakrishnan. Helios: Hyperscale Indexing for the Cloud & Edge. *PVLDB*, 13(12): 3231-3244, 2020.
DOI: https://doi.org/10.14778/3415478.3415547

1. INTRODUCTION

International Data Corporation (IDC) estimates that data will grow from 0.8 to 163 ZBs this decade [31]. As an example, Microsoft's Azure Data Lake Store already holds many EBs [61] and is growing rapidly. Users seek ways to focus on the few things they really need, but without getting rid of the original data. This is a non-trivial challenge since a single dataset can be used for answering a multitude of questions. As an example, inside Microsoft, telemetry (e.g., logs, heartbeat information) from services such as Azure Data Lake are stored and analyzed to support a variety of developer tasks (e.g., monitoring, reporting, debugging). With the monetary cost of downtime ranging from $100k to millions of dollars per hour [37], real-time processing and querying of this service data becomes critical.

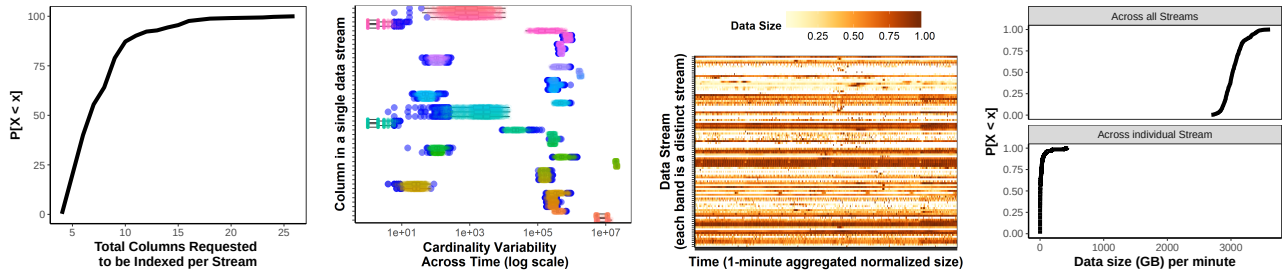
Figure 1(a) shows the distribution of the total number of columns requested to be indexed for streams that users identified as important for debugging a large internal analytics system called Cosmos. The indexes are used to quickly retrieve relevant logs when debugging failures. Over 50% of streams have more than seven columns that are frequently queried, and thus need to be indexed for faster retrieval. Coupled with the high cardinality (see Figure 1(b)) of the underlying columns, primary indexes or simple partitioning schemes are not sufficient; secondary indexes are a necessity to support such diverse workloads. Service telemetry also exhibits skewed temporal behavior — data streams are both diverse (Figure 1(c)) and high-volume (Figure 1(d)) — incoming telemetry can go as high as 4 TB/minute. None of our existing systems could handle these requirements adequately; either the solution did not scale or it was too expensive to capture all the desired telemetry.

In this paper, we present our experiences in building and operating Helios, a system for inexpensive and flexible ingestion, indexing, and aggregation of large streams of real-time data. Helios has evolved over the last five years to support several capabilities. First, it gives users an easy-to-understand data to memory hierarchy mapping based on their query needs providing them with a cost/benefit trade-off. This fits well with *write-once, read-often, long-lifetime* scenarios, which are increasingly common. There is a recency bias (since long-lived data becomes irrelevant, inaccurate, or outdated as it ages) and users can cache recent data in faster SSDs and spill older data to disks or remote storage. Second, we support computation offloading in that we can distribute computation (including filtering, projections, and index generation) to host machines, allowing the user to optimize costs. As a concrete example, for telemetry in our data centers, we handle ingestion rates of 10s of PBs/day by distributing index generation and online aggregation to the nodes where the data is generated (see Section 1.1), thus utilizing computational slack on those machines rather than incurring additional costs in the indexing infrastructure. Our approach therefore applies naturally to edge applications, since we can similarly distribute the task of data summarization — including filtering, projections, index generation, and online aggregation — to edge devices. Third, since we build indexes on-the-fly, a much wider range of queries, including point look-ups and aggregations, can be supported efficiently on the full spectrum of data from the freshest to the oldest.

At query time, we want users to leverage existing relational engines without having to choose one over another. While Helios is an indexing and aggregation subsystem, it is designed to provide query engines an abstraction of relations with secondary indexes, leveraged through traditional *access path selection* [63] at query optimization time. We have successfully verified the practicality of such an abstraction through integration with Apache Spark [11] and Azure SQL Data Warehouse [1].

We view this work as our first effort on building a new genre of distributed big-data processing systems that operate in the context

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415547



(a) (b) (c) (d)

Figure 1: (a) Distribution of the number of columns queried per stream; (b) Distribution of the cardinality for columns being queried; (c) Heatmap of a subset of data streams in our ingestion workload; (d) Distribution of raw data size ingested.

of cloud **and** edge. We see a number of future research problems that are worth investigating. One major problem is how to distribute computation between cloud and edge. In this respect, Helios provides one example for a specific type of computation, i.e., indexing of data. There are a number of related problems, such as resource allocation, query optimization, consistency, and fault tolerance. We believe that this is a fertile ground for future research.

1.1 An End-to-End Tour of Helios

In this section, we discuss Helios through an example.

Example (Log Analytics). Consider a data-parallel cluster such as Microsoft's internal service Cosmos [19], which is similar to a fully managed collection of Hadoop clusters, each with tens of thousands of nodes, containing several exabytes of data. The scope of analytics on these datasets ranges from traditional batch-style queries (e.g., OLAP) to explorative, "finding a needle in a haystack" type queries (e.g., point-look ups, summarization). Users (denoted by *user_id*) submit user jobs. A *user job* (denoted as *job_id*) is a distributed program execution that runs on multiple machines, and can be thought of as a graph in which each vertex (denoted *vertex_id*) is a task that runs on a single machine and edges denote precedence constraints between tasks.

Example users of log analytics over Cosmos telemetry include:

User A, a service engineer interested in quickly locating all logs generated by a single failing vertex.

User B, a sales representative computing an hourly report for jobs belonging to a specific user.

User C, a data explorer looking for all relevant streams that contain latency information pertaining to a specific user so she may compute a per-tenant resource prediction model.

Figure 3 shows an end-to-end example. A four-column table is generated across a set of sources. Users can specify a *loose* schema to the incoming data and provide hints about search patterns that could be interesting at query time with the help of a `CREATE STREAM` statement (Figure 2).

With Helios, the end-to-end experience is divided as follows:

1. Data Ingestion. At ingestion time, the `CREATE STREAM` statement consumes the data streams being generated on a list of machines specified by *SourceList* and indexes the relevant columns on-the-fly as shown in Figure 3. The statement generates two (system) jobs: One is packaged into a self-containing executable called an agent (see Section 5.1) that gets deployed onto every machine producing the log stream, and the other gets deployed onto the actual ingestion machines. In this example, streams (from each machine) are accumulated and chunked every minute into *data blocks* by the local agent (Step 1 in Figure 3). These data blocks are then sent to a collection of ingestion machines, where they are accumulated into larger blocks and persisted into a collection of append-only streams on a distributed file system (e.g., ADLS). Therefore, each block is persisted as an independently addressable chunk of data with a unique durable URI (Step 2 in Figure 3).

Figure 2: Example CREATE

```
CREATE STREAM Table
FROM FileSystemMonitorSource(SourceList, "d:\logs_*.txt")
USING DefaultTextExtractor("-d", ",")
(
        Timestamp: datetime PRIMARY KEY,
        user_id: string INDEX,
        job_id: string INDEX,
        vertex_id: string INDEX
)
OUTPUT TO "abfs://bigdata/logdata"
PARTITION BY FORMAT(Timestamp, "yyyy/mm/dd/hh")
CHUNK EVERY 1 MINUTE;
```

Subsequently, a collection of index blocks derived from these data blocks (by simply extracting the values of the corresponding "to be indexed" columns *tenant_id*, *job_id* and *vertex_id*) are merged into a global index, which holds the (Column, Value) $\rightarrow$ Data Block URI mapping (Step 3 in Figure 3). This step allows us to provide indexed access to massive data streams. While we discussed the case where indexing happens on the ingestion layer for simplicity, in reality, it gets pushed down into the agent. We discuss the complete model in Section 2.1.

2. Query. Users can query the underlying data using SQL. The optimizer will *transparently* prepare the best execution plan, leveraging the indexed columns if they help.

```
SELECT Timestamp
FROM Table WHERE user_id == 'adam'
```

If the optimizer picks a plan that leverages the indexes, relevant (indexed) predicates are used to identify candidate data blocks using the global index service (Step A in Figure 3), in order to prune the original data stream for subsequent query execution (Step B in Figure 3). For instance, in Spark, the query is distributed to workers running on different machines for data block retrieval, exploiting task-data locality where applicable (Step B). Each block is then read by leveraging existing optimizations (e.g., pushing down predicates) to further obtain any potential data reduction. Any shuffling or join is handled by the underlying query engine before serving the final result to the user. We discuss the complete query processing model in Sections 3 and 4.

1.2 Production Impact

Over the last five years we have designed, implemented, and operated Helios as a distributed ingestion and indexing system for structured and unstructured data at Microsoft. Helios is a highly available service (internally classified as a Ring-0 debugging service) that *scales reliably* to petabytes of data, collects close to a

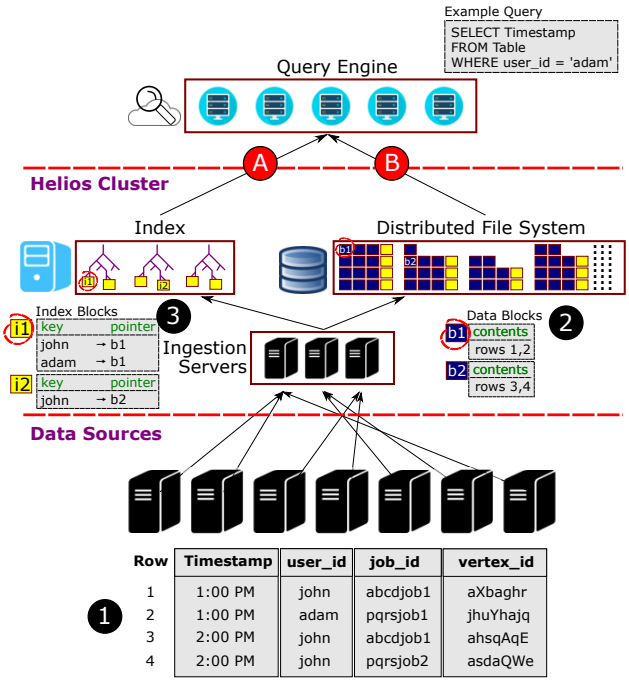


Figure 3: E2E Helios scenario: (1) Data is generated; (2) Chunks of data are independently committed; (3) A block-level index maps search key space to data blocks; (A) Query engine obtains candidate data blocks from block-level index; (B) Relevant data blocks accessed via traditional access methods.

quadrillion log lines per day from hundreds of thousands of machines, indexing approximately 16 trillion keys per day, and has a *wide geo-distributed footprint* (> 15 data centers around the world). It serves as a *reference blueprint* for other internal large-scale systems within Microsoft.

Helios is used widely for several internal use cases at Microsoft, e.g., debugging and diagnostics, workload characterization, cluster health monitoring, deriving business insights and performing impact analysis of incidents in Azure Data Lake (ADL) [61] and Cosmos [19]. In addition, there are several other interesting applications being based on ideas from Helios including debugging for several production clusters, secondary indexing for data stored on ADL/Cosmos, etc. The Helios clusters used by these applications span from a handful to hundreds of machines and store petabytes of data over a specified retention period.

1.3 Our Contributions

To summarize, the key contributions of this paper are:

- **Hierarchical Data Abstraction**. The hierarchical nature of the Helios index allows each layer to be (1) computed at different stages of the data collection life cycle, (2) held at a different tier in a storage layer, and (3) utilized independently at query time. This flexibility allows for a full spectrum of *cost* and *performance* trade-offs that users can choose from, while operating reliably at scale.
- **Production System**. We describe Helios in production use at Microsoft, highlighting the following advantages: First, it allows users to aggressively push down indexing and meta-data extraction to the edge (i.e., sources generating data) so they can be performed on-the-fly during ingestion time. Second, it builds on lower-level abstractions such as ADL's Tiered Store [61] that provides scalability, synchronous replication, and strong consistency. Finally, in our system, *index is data*. Helios indexes are represented and exposed as data in easily accessible formats, available for any query engine. This allows us to democratize index usage outside of Helios, e.g., query engines can perform their own cost-based index access path selection, and index reads can independently scale without being constrained by compute resources provided by Helios. In fact, we were able to integrate Apache Spark [11] with no changes to its optimizer/compiler. Some of the core query optimization techniques from Helios have been open-sourced as part of Hyperspace [3].
- **Operational Experiences**. We discuss our operational experience with a large-scale deployment of Helios at Microsoft for enabling log analytics for ADL and Cosmos.

Paper organization. We discuss the end-to-end user experience in Section 2.1, and describe how this has been implemented in Section 2. We describe the indexing data structures in Section 3 and explain how indexes fit into query processing in Section 4. We describe practical considerations in deploying a large-scale system in production in Section 5 and illustrate real-world use cases in Section 6. We summarize lessons we learned in Section 7, discuss related work in Section 8, and conclude in Section 9.

2. HELIOS SYSTEM ARCHITECTURE

Helios is an end-to-end system that allows users to ingest data and query it through a distributed query engine. In this section, we describe the individual layers within the broader system.

2.1 Data and Query Model

Helios exposes a semi-relational data model to applications. Data is organized as tables of records loosely schematized with some attributes exposed for indexing — there may be additional attributes, and not all records have values defined for all attributes. Records can be stored in any storage system with the following requirement:

The storage system provides an atomic append operation that takes a group of one or more records, herewith referred to as a block*, and returns a handle (e.g., URI) that can be used to efficiently retrieve the appended bytes.*

Such a data model is widely supported by commonly available storage systems in a data lake ecosystem. For instance, HDFS [64] and ADLS [61] can address data using ⟨offset, length⟩ within a file, while WAS [17] can address data using a blob URI.

Helios provides global *secondary indexes* (herewith referred to as index or global index) on top of the data stored in the underlying storage, stored as *index blocks* on the same reliable storage. A secondary index maps the indexed columns of a record to the handle of the data block that contains the record. Specifically, a user ingests data into the underlying storage system (where it is stored as data blocks), and Helios continuously builds and maintains the secondary indexes (persisted as index blocks) on the ingested data. Query engines can leverage Helios's secondary indexes at query time to prune data at the granularity of data blocks.

Helios currently supports indexing all primitive data types such as integer, float, string, bytes, and so on. For more complex nested objects (e.g., JSON), Helios allows user to promote the elements into a flattened top-level column which can then be indexed.

2.2 Overview

Ingestion in Helios consists of three operations: (1) building the index, (2) persisting both the index and data, and (3) orchestrating fault-tolerance protocols between data sources and ingestion servers. Building an index is computationally intensive — this is aggravated in the case of unstructured streams since they need to be pre-processed (e.g., parsing and imposing schema) before they can be indexed. The sheer size of data ingested in our production environments necessitates large ingestion clusters. The cost of setting

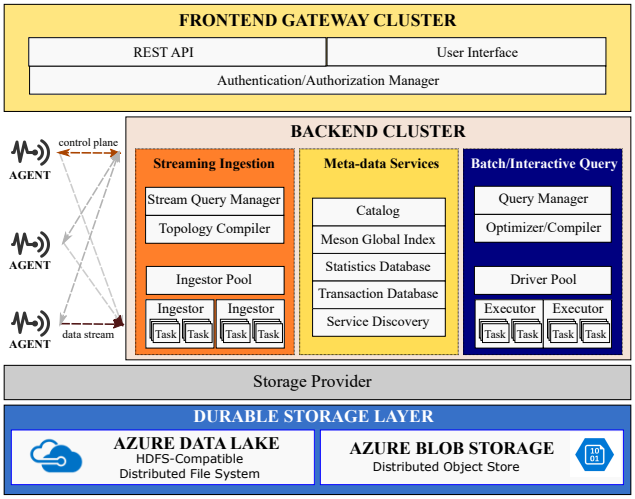


Figure 4: Helios Architecture.

up and maintaining such clusters adds to the capital and operational expenses of the host clusters (e.g., Cosmos), and it is very desirable to reduce the size of ingestion clusters. Our key idea here is to split the computation so that the expensive part (e.g., parsing, indexing, aggregation) can happen outside the ingestion clusters. Concretely, we package them into *agents* that can be deployed, e.g., where the data to be indexed is generated, if there is computational slack we can utilize on those machines. We transport the resulting index entries to the ingestion clusters in a compact format, and utilize the ingestion clusters for relatively cheaper operations (e.g., meta-data and index management). This yields many additional advantages, e.g., reducing network, computation, and storage requirements.

As the data and indexes are both write-heavy and require stringent read latency guarantees, we designed a new global indexing mechanism (Section 3). This mechanism is optimized for workloads that are *write-heavy*. It also provides users with flexibility to trade off query consistency and latency.

At query time, the multi-layered index built by Helios presents query engines an abstraction of relations with secondary indexes. Such an abstraction can be easily integrated into existing query engines through traditional *access path selection* [63]. We explain how we translate query requests to operations over the data and indexes stored in distributed file systems, with various levels of consistency guarantees in Section 4.

2.3 Architectural Details

Figure 4 shows the high-level architecture of Helios. Users interact with Helios using an *agent*, which prepares their data for ingestion and sends requests to one of many *backend* servers.[1] These servers are responsible for reading/writing data from remote data sources, updating the catalog, building global indexes, and running user queries. Clients can choose to persist their data either in Azure Data Lake Store (ADLS) or Azure Blob Storage (WAS) with custom retention policies based on data stream type (e.g., preserve error logs) and/or time (e.g., store for at least one week). Other storage integrations are possible through the storage provider abstraction that Helios exposes.

Data Processing Agents. Data collection happens through a *data processing agent*, an independently-running standalone entity intended to be executed as close as possible to the data source (e.g., host machines outside the cloud), though we could also run the agent in the ingestion cluster. Because of the geo-distributed architecture of Helios, an agent can run in any pod/rack/datacenter/-country and thus, special care must be taken to avoid unnecessarily increasing write latency — an incorrectly chosen server (e.g., in a datacenter on the other side of the planet) might lead to backpressure scenarios that may be hard to mitigate. To alleviate this problem, an agent periodically utilizes a *service discovery* component to obtain the latest network topology and picks the list of servers that are *healthy* and *nearby*. In addition, the agent also takes care of fault tolerance, in terms of both local failures (e.g., machine restart) or protocol-level failures (e.g., ingestion server being unavailable).

Backend Cluster. The *backend cluster* consists of ingestion machines, meta-data services, and query execution machines. The ingestion and query execution runtimes are co-hosted on the same set of physical machines to exploit locality whenever possible — if the user queries for recent data, there is no need to fetch data from remote machines. Backend servers are typically co-located in the same datacenter but within different fault-tolerance domains. This ensures that Helios servers have fast access to the underlying data and also increases availability and reliability. Data, unless deemed critical, is typically not geo-replicated at this layer — if the user chooses to, she can enable geo-replication features in the underlying storage system.

Ingestion servers are *stateless*. An agent can communicate with a different ingestion server for each request. This allows a wide variety of load distribution mechanisms (e.g., round-robin, consistent hashing, etc.). It is also easy to quickly add (or remove) servers from our system in response to the total instantaneous load. Any changes in the ingestion topology will automatically re-balance the load. When a block is received from an agent, the ingestion server is responsible for executing the corresponding user-defined *ingestion dataflow*, which can consist of indexing operations. The Helios indexing component (Section 3) is responsible for providing a distributed global index that helps locate data blocks (both in cache and remote storage) for a set of given predicates. Statistics about the data stream are persisted into a light-weight *statistics database* for providing insights to users about their data streams.

The backend clusters have several additional components that allow for the execution of distributed SQL queries. In an example instantiation, Helios supports large-scale interactive data processing through Apache Spark [11]. For performance reasons, the Spark workers are allowed to communicate directly with the ADLS remote storage to fetch data but are configured to exploit local caches on the ingestion servers.

Conceptually, the ingestion layer can be considered a best-effort temporary cache — we do not implement active replication in this layer. Instead, we rely on passive replication, i.e., if a block of data cannot be located in the cache tier during an incoming query, it will be fetched from the remote store and cached within the server. Since the servers do not implement active replication, the process of adding/removing servers is usually quick (i.e., there is no explicit hydration or draining required). This feature is disabled for stores that natively support tiering (e.g., ADLS [61]).

[1] In our design, data is shipped (from the agent running on the edge) and stored at the Helios side, and we do not assume that both the data producers (e.g., Cosmos machines) and Helios clusters access the same durable storage (e.g., Azure Storage). It is true that in reality Cosmos machines can also access Azure Storage. However, they do not ingest data directly to Azure Storage. Instead, they send data to Helios and it is Helios's responsibility to store the received data in Azure Storage and build index on top of the data. Since Cosmos has hundreds of thousands of machines, writing (small log records) directly to Azure Storage may stress the system too much. Therefore, we use Helios ingestion servers, as also a layer for performing intermediate aggregations of data so we can write larger blocks to the underlying durable storage.

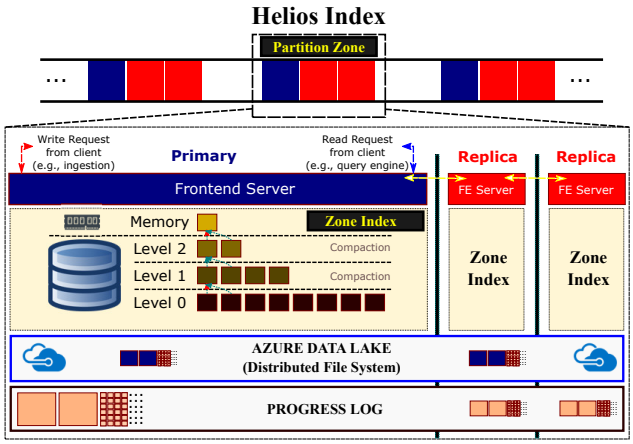


Figure 5: Helios leverages a combination of tree- and hash-based indexing techniques to build its distributed global index.

Finally, the catalog provides the latest view of the data streams. This gives the user the ability to either interactively query (relatively smaller) portions of their dataset or resort to traditional batch processing. The ADLS-based remote storage model and our geo-distributed presence leads to latency and throughput characteristics that can support extremely high ingestion rates.

Frontend Cluster. Users submit queries via a *frontend gateway* cluster that is responsible for authenticating and authorizing users. In production clusters where we expose Spark as the query engine, several Spark application drivers take incoming user queries, create new user sessions as necessary, and distribute them to the Spark workers running on the backend machines. This layer is also responsible for providing the necessary query-level fault tolerance. We do this through a traditional *log-and-replay* mechanism, i.e., log every incoming user query in a session, and if a session is not found upon the next query, provide the user with an option to replay their entire session. User sessions are fair-shared and automatically expire in an admin-configured duration.

3. ASYNCHRONOUS INDEXING

Helios maintains indexes on thousands of columns over petabytes of data per day. A key factor in Helios's success is asynchronous index management: Helios maintains *eventually consistent* indexes against the data, but expose a strongly consistent queryable view of the underlying data with best-effort index support.

3.1 The Helios Index

Helios leverages a combination of classic tree-based and hash-based indexing techniques to build its distributed global index (see Figure 5). A Helios index $\mathcal{I}$ contains two components $\mathcal{I}\langle\mathcal{P}, \mathcal{Z}\rangle$:

- *Partition Zone* $\mathcal{H}$ – Helios distributes indexed data by hashing on a user-specified *partitioning key* or simply using round-robin distribution, if no key is specified. Each partition falls into its own *reliability zone*, with multiple replicas to ensure fault tolerance and improve availability.
- *Tree-based Zone Index* $\mathcal{Z}$ – Within each reliability zone, Helios maintains a tree-structured index on the data.

3.1.1 Tree-based Zone Index

The design of the zone index is inspired by a variety of classic indexes, including B-trees [27], log-structured merge (LSM) trees [58], and Merkle trees [53]. Figure 6 presents its structure:

- *Leaf nodes* – A leaf node maps search keys to pointers of the corresponding data with the search keys. Pointers here could be paths to files stored in the underlying file system, or offsets inside files that locate addressable data chunks.

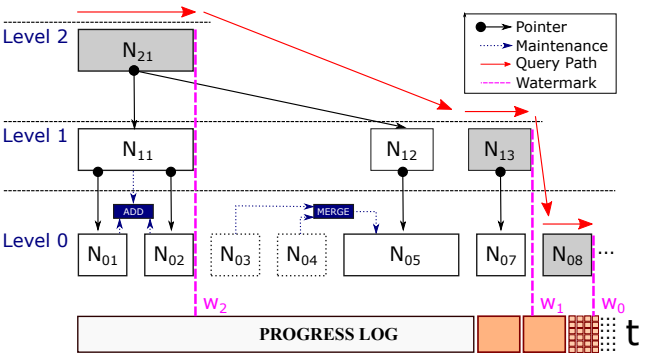


Figure 6: The zone index structure in Helios.

- *Internal nodes* – An internal node contains pointers to internal/leaf nodes at the next lower level.

In contrast to classic tree-structured indexes, there is *no global order* on the indexed keys in any of the levels hosted by the zone index, though one is free to sort keys within each individual node (leaf or internal node) *locally*. This will become clear after we present the index construction algorithm (Section 3.1.2). Moreover, each level of the index maintains a *watermark* that records the timestamp (implemented as a sequence number) of the latest event being ingested *and* indexed *by this level*. Since Helios adopts an asynchronous, merge-based approach for index updates, these watermarks serve as safeguards that ensure the correctness of a novel *hybrid index scan* operator that can be employed by query execution plans (see Section 4).

3.1.2 Index Construction

The Helios index is constructed in a *bottom-up* manner. This is different from building classic tree-based indexes such as B-trees, where data is inserted into the index in a top-down manner, even if the tree grows bottom-up (e.g., the bulk loading algorithm [60]).

Helios uses a *progress log* that records data events being ingested so far (see Section 3.3). We start by scanning the progress log from the latest checkpoint. Data records are grouped in a uniform fashion, by using fixed-size (e.g., 100 records per chunk) or fixed-time (e.g., chunk every 1 minute) policies, and we construct an *initial* leaf node (i.e., nodes at level L_0) for each group/chunk. A fundamental difference with respect to classic tree-based indexes such as a B-tree is that no global order is enforced over the leaf nodes: Indexed keys are packed into the leaf nodes based on the arrival order of the corresponding data records. After this initialization stage, an index node at level L_i ($i \geq 0$) is *finalized* by *combining* indexing information from nodes at either level L_{i-1} (for $i \geq 1$) or level L_i (for $i \geq 0$). Specifically, we represent the content of an index node N as a *collection* of pairs $\langle K, P\rangle$, where K is a set of search keys and P is a set of pointers. We use the notation $N\{\langle K, P\rangle\}$ hereafter. The following property mus be satisfied by every index node to ensure correctness of index-based search:

PROPERTY 1. *For any search key $k \in K$, there exists* at least one *pointer $p \in P$ such that k appears in the index or data block (e.g., file or chunk) pointed to by p.*

Given two index nodes $N_1\{\langle K_1, P_1\rangle\}$ and $N_2\{\langle K_2, P_2\rangle\}$, we can combine N_1 and N_2 to produce a new index node $N\{\langle K, P\rangle\}$:

- *Merge* two nodes N_1 and N_2 from L_i, denoted by $N_1 \cup N_2$ (Algorithm 1), which simply looks for common keys and takes a union of corresponding pointers, i.e.,

$$K = K_1 = K_2 \text{ and } P = P_1 \cup P_2.$$

Algorithm 1: *Merge* two index nodes $N_1 \cup N_2$.

Input: Two index nodes $N_1\{\langle K_1, P_1\rangle\}$ and $N_2\{\langle K_2, P_2\rangle\}$ from level L_i of the index.

```
N ← N1;
foreach ⟨K2, P2⟩ ∈ N2 do
    if there exists ⟨K, P⟩ ∈ N s.t. K = K2 then
        P ← P ∪ P2;
    else
        N ← N ∪ {⟨K2, P2⟩};
Insert N⟨K, P⟩ into Li of the index;
Delete N1 and N2 from the index;
```

Any $\langle K_1, P_1\rangle$ (resp. $\langle K_2, P_2\rangle$) such that K_1 (resp. K_2) only appears in N_1 (resp. N_2) remains unchanged in N. N_1 and N_2 will be removed from L_i after merging.

- *Add* two nodes N_1 and N_2 from L_{i-1}, denoted by $N_1 \oplus N_2$ (Algorithm 2), which takes a union of the set of keys and creates a new set with two pointers that point to N_1 and N_2, i.e.,

$$K = K_1 \cup K_2 \text{ and } P = \{p(N_1), p(N_2)\},$$

where $p(N)$ is a function that returns a pointer to index node N. However, N_1 and N_2 will not be removed after N is inserted.

Clearly, both *Merge* and *Add* are *closed* under Property 1:

PROPERTY 2. *The combined node N by either $N_1 \cup N_2$ or $N_1 \oplus N_2$ preserves Property 1.*

Algorithm 2: *Add* two index nodes $N_1 \oplus N_2$.

Input: Two index nodes $N_1\{\langle K_1, P_1\rangle\}$ and $N_2\{\langle K_2, P_2\rangle\}$ from level L_{i-1} of the index.

```
K ← (⋃K1) ∪ (⋃K2);
P ← {p(N1), p(N2)};
Insert N{⟨K, P⟩} into level Li of the index;
```

In our current implementation of the index construction phase, we apply the *Merge* operator to the leaf nodes and use the *Add* operator for growing the index upwards. We note that *Merge* is not just limited to leaf nodes, it is also used on nodes at the same level to compact smaller blocks into larger ones, as we will see later. Algorithm 3 illustrates building the entire index using the two operators *Merge* and *Add*.

Algorithm 3: Construction of the zone index $\mathcal{Z}$.

Input: $\mathcal{D}$, the set of data blocks; n, the total number of index levels.
Output: Return the zone index $\mathcal{Z}$.

```
foreach B ∈ D do
    Create an index node N{⟨{k}, {p(B)}⟩} for each
      search key k found in B;
    Add N to the leaf level L0;
Apply Merge (Algorithm 1) to the leaf level L0 w.r.t. some
  merge policy;
for 1 ≤ i ≤ n do
    Build the level Li by applying Add (Algorithm 2) to
      nodes from Li−1 w.r.t. some add policy;
return The zone index Z constructed;
```

Both *Merge* and *Add* follow some policies. The *merge policy* determines when to merge nodes into a potential compact representation, whereas the *add policy* determines when to add a new layer that grows the tree. These policies are highly flexible, and different levels can employ different policies. As an example, at the leaf level, we currently use a size-based policy for *Merge*: If both sizes of N_1 and N_2 are below a (configurable) threshold (e.g., 64MB), then N_1 and N_2 will be merged. Similarly, the policy for *Add* is also size-based — we keep adding nodes (at a higher level) until the size of the resulting node reaches a certain (configurable) threshold. We discuss more policies in Section 3.1.4.

Due to the size-based policies, a non-root index level may contain index nodes that are *orphans*, i.e., they do not have parent nodes in the next higher level. For example, in Figure 6 we have marked all orphan nodes using gray. Also note that by definition all index nodes in the root level are orphans. In Helios, we implement a *hybrid index scan* operator that utilizes these orphan nodes to provide stronger consistency for query semantics (Section 4). Of course, one can reduce the number of orphan nodes by choosing different merge/add policies, or simply eliminate orphan nodes by allowing *underutilized* index nodes.

Hash-based Key Space Reduction. One problem of *Merge* and *Add* is that the resulting index node may contain many index keys (after taking the union). In particular, this is a critical issue when the index keys are from a large domain consisting of billions (e.g., Job ID, Device ID) or trillions (e.g., Vertex ID, Task ID) of search keys. To avoid this phenomenon of "cascading explosion," instead of directly taking a union over the index keys, we first apply a hash function on the index keys and then take the union over the hashed values. Each internal level of the index uses a different hash function, where the hash function used by a higher level further reduces the key space generated by the hash function used by the previous lower level.

3.1.3 Index Maintenance and Compaction

Maintaining the zone index $\mathcal{Z}$ when new data arrives is straightforward. We keep accumulating data blocks until a desired number is reached, which triggers an *index compaction* procedure as illustrated in Algorithm 4.

Algorithm 4: Updates and compaction of the index.

Input: $\mathcal{Z}$, the current zone index with n levels; $\mathcal{D}$, the set of new data blocks.

```
foreach B ∈ D do
    Create an index node N{⟨{k}, {p(B)}⟩} by collecting
      search keys k from B;
    Merge N with orphans in the leaf level L0;
for 1 ≤ i ≤ n do
    Add new nodes in the level Li, starting from the existing
      orphans in Li.
```

We again perform compaction in a bottom-up fashion. We first construct the new leaf nodes by merging from the new data blocks *and* any old orphans that are below the size threshold. Since more leaf nodes are available now, it may trigger adding more internal nodes in the next higher level. If so, we apply the *Add* operator starting from the old orphan nodes. This procedure is recursive and we keep adding more internal nodes level by level until no more actions can be performed.

3.1.4 A Generic Indexing Model

Thus far, we have presented the index structure along the lines of our current implementation in Helios. However, the index model can be generalized to a form that has little dependency on our specific implementation. We now discuss this generic index model.

Index Structure. The generic index model still contains two components: (1) a *partitioning scheme* $\mathcal{P}$ that distributes data to different zones, and (2) a *hierarchical index* $\mathcal{Z}$ within each partition zone. The partitioning scheme $\mathcal{H}$ is not restricted to the hash-based or round-robin based schemes currently implemented in Helios, though. One can use various other partitioning strategies such as *range-based* ones. In fact, these strategies are more favorable in the presence of data skew. On the other hand, while the zone index $\mathcal{Z}$ remains hierarchical, like the one implemented in Helios, it does not necessarily need to be a tree. Indeed, the zone index can be generalized to a directed acyclic graph (DAG). This would enable more flexible merge policies for merging non-orphan index nodes.

Properties of Index Nodes. While index nodes are currently stored as HDFS files in Helios, this is not necessary in general. Indeed, an index node only needs to be an *atomically addressable chunk* (AAC) that conforms to Property 1. By "atomically addressable" we mean that an index node or data chunk is indivisible and has its own uniform resource identifier (URI), which is unique in a hierarchical storage system. For example, if we wish, we could even use a database table to store an index node — in this case, the table identifier would become the URI for this index node.

Primitives for Index Construction. In addition to the two primitives, *Merge* and *Add*, for index construction, one can introduce other primitives. The only requirement for an index construction primitive is the closure property (i.e., Property 2). As an example for other primitives, consider a *Split* operator that splits an index node $N\{\langle K, P\rangle\}$ into multiple ones $N_1\{\langle K_1, P\rangle\}$, ..., $N_l\{\langle K_l, P\rangle\}$, where K_1, ..., K_l form a *partition* of K using hash partitioning. *Split* is useful in situations where the key space is large and index nodes tend to contain many distinct keys. In such cases we can apply *Split* to index nodes before performing *Merge* or *Add*, and design merge/add policies that tend to combine index nodes with overlapping key sets. Clearly, the *Split* operator follows the closure property.

Policies for Applying Primitives. For each of the index construction primitives, one can employ a policy that specifies when and how to execute the primitive. We have discussed two specific policies, both size-based, for the *Merge* and *Add* primitives. Indeed, due to the closure property followed by the primitives, these policies can be made as generic interfaces that allow for various implementations. As another example of a merge policy, one can first *cluster* the index nodes based on some metric that measures the *similarity* between the key sets of the index nodes. One then simply merges index nodes that fall into the same cluster.

Comparison with Learned Indexes. The recent emergence of learned index structures has spawned extensive research [43]. In some ways, the Helios index is closer to the spirit of learned index structures than traditional tree-based indexes, though we do not employ any learning-based techniques at present. At the highest level, a learned index tries to learn a function that maps an index search key to the leaf nodes containing the search key. The Helios structure in its generic form performs exactly the same function, with the difference that the mapping is not learned from data but composed from the (typically, hash) partitioning functions used by different index levels. While learned indexes might learn this mapping more accurately on static data or evolving data without drift of data distribution, the Helios index is perhaps more robust in a dynamic environment where the underlying data distribution can drift from time to time, which is typical in workloads faced by large-scale data ingestion systems.

3.2 Key Properties

Asynchrony-friendly. Building and maintaining indexes consists of atomic writes of both data and its associated index into the underlying storage systems. Maintaining indexes synchronously requires that two atomic writes (i.e., one for the data block and one for the index block) be in the same transaction. Applications can use Helios for high availability by replicating their index and data. However, replication requires distributed consensus. This requirement makes our write transaction relatively heavy and may lead to increased latency or back-pressure. Our initial customer was Cosmos [19] where ingestion can happen simultaneously from more than a quarter-million machines. Detecting and dealing with back-pressure in a synchronous ingestion model can be very expensive.

To balance scale/user requirements with a reasonable end-to-end experience, our approach decouples index building from data ingestion, making them asynchronous. This asynchrony also allows us to utilize optimizations such as buffering index updates for ingested data, thereby amortizing the replication and consensus cost. Although not mandatory, this design decision to make the index only eventually consistent with the underlying data allows us to ensure that data ingestion latency will not be affected by index management operations, i.e., users will not experience any latency increase even with extra index management.

Load balancing-friendly. Since the Helios indexing model is recursive, any node can offload its computation to another node (sibling/parent/child), giving us great flexibility in how the model is implemented in practice. For instance, if the source machine (which could be an edge node) is overloaded, it can choose to trade-off more network bandwidth for an opportunity to offload its 'parsing' or 'indexing' work to a parent node (inside the cloud).

3.3 Durability, Scalability & Consistency

Achieving durability through the progress log. The key to decoupling yet systematically orchestrating the index building and data management asynchronously is a layer that does the necessary book-keeping. For this, we introduce the *progress log* to track the progress of both data ingestion and index building. Since all writes to the underlying storage system go through Helios (either directly or indirectly, through another write log), a write transaction can now be modeled as follows. Helios first durably appends the data into the storage system. Once the write is acknowledged with the handle: <*Timestamp*, *Sequence No*, *URI*, *Filename*, *Start Offset*, *Length*>, we append atomically a log entry consisting of the handle into the progress log, and only acknowledge the user upon a successful log append. Any failure during the data append or the log append will abort the write transaction and require retry from the caller. Compared to a traditional write transaction in a storage system without Helios, Helios only adds the cost of an extra log append operation to the write transaction, which causes minimal additional latency. The progress log also serves as the naming service for the data stored, i.e., there is a one-to-one mapping between the log entries and the corresponding data blocks written to storage.

Note that the progress log is different from a traditional Write-Ahead-Log (WAL) — a traditional WAL logs the operations *before* they are executed, while our progress log only records successful operations afterwards, not failed ones. This design of the progress log avoids the need to include the appended data into the log (thus, reducing its size), and simplifies the maintenance and cost of error handling (i.e., caller just retries). As a consequence, the log is much more compact, and is blazingly fast to maintain, even in a large-scale distributed environment. Although this design may incur duplicated data caused by retries at failures, queries can use

the progress log for a consistent view of the underlying data. We have observed that the amount of duplication incurred due to failures is hardly noticeable in practice. Not dealing with duplicates was a conscious decision driven by two considerations: (1) Helios strives to achieve "at least once" semantics, and (2) we have to deal with data streams that are out of our control (e.g., generated by an external process that does not offer any standard means to detect duplicates, e.g., as in the case of logging).

Achieving scalability through partitioning. Once a write transaction of a data block succeeds, *index hydration* (herewith, hydration) — the process of building and durably persisting indexes of the newly ingested data blocks and exposing them for query engines — happens asynchronously in the background. Specifically, the indexing layer maintains an *offset* into the progress log, to keep track of the handle to the latest data block whose index has been built. The indexing layer periodically triggers hydration on the newly inserted data blocks, that is, the data blocks after the index-offset. Such a hydration process retrieves the data block's columns that were configured to be indexed and inserts the <*Column Name*, *Column Value*, *Data block URI*> into the index. The frequency of triggering the hydration process can be tuned by users to trade-off throughput and latency of index building. In general, a longer period can buffer more records and insert their indexes in a batch, improving the throughput. However, the index will lag the data for a longer period that in turn may affect the cost of querying.

Two issues in the data ingestion process are worth additional discussion. First, the progress log holds only the handles to the data. Thus, retrieving the data for building indexes requires reading data from storage. Such an indirection may incur extra (and expensive) I/O. Second, index building may require heavy ETL operations such as parsing and transformation on the data blocks, consuming significant computation resources on a set of centralized index servers. While these problems seem simple, they can cause the cost of indexing solution to become unacceptable at scale. In our implementation, we tackle these problems through a combination of caching (e.g., caching data blocks on the ingestion servers), and edge-enabled computing (e.g., on-the-fly parsing and indexing).

Achieving desired consistency. Since both the index and data are continuously changing, we explored two read APIs (for use by a query engine) with different consistency guarantees:

- *Read committed*: Returns the latest view of the dataset, consisting of all records that have been committed at the time of the query. In this mode, the query engine will exploit the indexes, to the extent possible, to prune unnecessary data reads, but still trigger a linear scan on any newly inserted data.
- *Read snapshot*: Returns a possibly stale version of the dataset. In this mode, the query engine will use *only* the indexes to get records that satisfy the specified predicates on a possibly stale version of the dataset.

4. QUERY PROCESSING

Due to the existence of orphans in internal levels, index access in Helios cannot follow a strict "*move down*" protocol as in classic tree-based indexes. We therefore describe a *hybrid index scan* operator that supports both "*move down*" and "*move right*" when accessing index nodes—we assume, conceptually, that orphans are always on the right of non-orphans in each level of the index.

Algorithm 5 illustrates the details. The main loop (lines 13 to 15) basically executes the "move right" operation by scanning the orphans level by level in a top-down manner. When scanning each index level, we invoke the **Search** function (lines 2 to 10) over each orphan node. For example, the red arrows in Figure 6 showcase

Algorithm 5: A hybrid index scan.

Input: $\mathcal{Z}$, zone index with n levels; k, the search key.

```
1
2  function Search(N{⟨K, P⟩}, k):
3  if N ∈ L_0 then
4  |  foreach ⟨K, P⟩ s.t. k ∈ K do
5  |  |  foreach p(B) ∈ P do
6  |  |  |  Scan the data block B for records matching k;
7  else
8  |  foreach ⟨K, P⟩ s.t. k ∈ K do
9  |  |  foreach p(N') ∈ P do
10 |  |  |  Search(N', k);
11
12 main loop:
13 for n ≥ i ≥ 1 do
14 |  foreach N{⟨K, P⟩} ∈ L_i s.t. N is orphan do
15 |  |  Search(N, k);
```

how index search marches along the *frontier* line formed by orphans. The **Search** function executes the "move down" operation by *recursively* inquiring the child nodes pointed to by the current index node being searched, if the search key has been found within the current node (lines 8 to 10).

The index search algorithm naturally supports the two aforementioned consistency levels for queries: *read committed* and *read snapshot*,. If a query is only interested in reading the latest snapshot, then it only needs to scan the index until the rightmost leaf node has been accessed. The watermark of that leaf node serves as the *version number* of the corresponding snapshot, due to the following property:

PROPERTY 3. *Let $\mathcal{Z}$ be a zone index with n levels. Let w_i be the watermark of the index level L_i ($0 \leq i \leq n$). Then $w_0 \leq w_1 \leq \cdots \leq w_n$.*

The rationale is that, since index updates are asynchronous and bottom-up, a lower index level always contains fresher data/index information. Indeed, a query can further access *any* snapshot rather than just the latest one. Given the version number (i.e., timestamp) of the desired snapshot, one can simply follow the index search algorithm until an orphan node with a larger timestamp is reached.

On the other hand, if a query wants to fetch all committed data, then after finishing to search the index, we can switch to scanning the progress log to search for the latest data blocks that have not been indexed yet. This linear scan, however, is *bounded* – only the log tail between the watermark w_0 of the leaf level and the latest committed data time t is interesting. Such bounded property also offers opportunities for further performance tuning: If the time range $[w_0, t]$ becomes so large that hurts the overall query execution performance, then one should trigger the index compaction procedure more frequently to allow more fresh data blocks to be absorbed into the index.

Cost-based Index Access Path Selection. The *hybrid index scan* operator introduces an interesting performance trade-off. Indeed, one can stop exploring orphan nodes at any level and switch to scanning the remaining leaf nodes afterwards. For example, in Figure 6 one can skip N_{13} and instead directly scan the remaining leaf nodes N_{05} to N_{08}. A natural question is then which strategy can lead to faster overall access performance. If N_{13} does not contain the search key k, then one can just skip N_{07} since it must not contain k as well. In this case, accessing N_{13} is better than skipping it. On the other hand, if N_{13} does contain k, then one has to also access N_{07} for an additional look-up. In this case, accessing

N_{13} is worse than skipping it, as the access to N_{13} results in additional overhead. One can leverage a cost-based approach to address this access path selection problem, by maintaining statistics (e.g., bloom filters) that can provide hints to whether an index node is worth accessing. We leave this interesting direction as future work for exploration. Note that, while one can in theory also skip internal nodes and jump to leaf nodes directly in classic tree-based indexes, it is seldom beneficial as each index level maintains a global order and only one of the child node needs further look-up.

5. PRACTICAL CONSIDERATIONS

In this section, we discuss some critical design questions and their associated trade-offs that helped towards operationalizing the ideas behind Helios and making the system scale in production.

5.1 How to Scale-out Index Generation?

Indexing is a computation-intensive step and may include decompression, parsing, and transformation of the raw data ingested, as well as any indexing prerequisites such as sorting and summarization. Collectively, Helios collects $\approx$ 12 PB/day of raw data across 15 data centers. Our initial approach towards data and index management focused towards bringing in data from all the sources into an ingestion cluster and performing the required post-processing operations (e.g., parsing, indexing etc.). However, this approach incurs significant costs (e.g., transmitting all compressed data to ingestion and then decompressing etc.), defeating the important purpose of reducing the capital and operational expenses of maintaining our production clusters.

The hierarchical/recursive indexing model gives the freedom of distributing our computation across the agents and ingestion backend servers, sharing the same intuition as the current trend of edge computation. Following this idea, we extracted the core data processing and index building logic as a minimalistic data processing library called Quark, implemented in C++. Quark is shared by agents, ingestion backend servers, and multiple query engines. At a high-level, Quark consists of data writers and readers.

- Writers offer a dataflow execution engine that allows our users to specify how data from the producers should be processed before it is stored in an appropriate (e.g., columnar) format. To this extent, Quark allows users to construct data and index blocks by leveraging one or more of its built-in components: parsers (that chunk incoming data into individual records, e.g., line parsers), extractors (that allow extraction of column-value pairs from a record, e.g., JSON, regex extractors), indexers (that allow user to summarize and/or hold pointers to their data, e.g., inverted index, sketches), partitioners (that partition the incoming data by a specific column, e.g., timestamp partitioner), serializer (that serializes the data and index), and compressor (that allows for columnar compression).
- Readers expose query processing operators such as SELECT and PROJECT with the ability to push down predicates. The engine allows for columnar data retrieval and vectorized execution.

The distribution of ingestion workload is a trade-off of COGS and resource constraints on the data source machines — having an agent will reduce the size of the ingestion cluster needed, but will consume processing power from the source machine.[2] As agents can be deployed to heterogeneous sources, including computation-intensive service machines or resource-constrained devices, carefully constraining the resource utilization of the computation on the agent is critical. In practice, we also allow users to impose strict restrictions on the agent's CPU and memory consumption.

[2]COGS ("cost of goods sold") is a standard economic term used widely in the technology industry to refer to the cost of procuring and maintaining systems. Our goal is to have lower COGS.

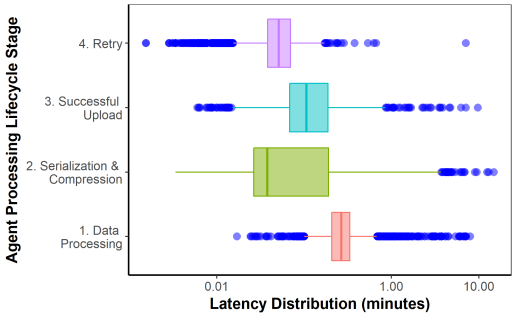
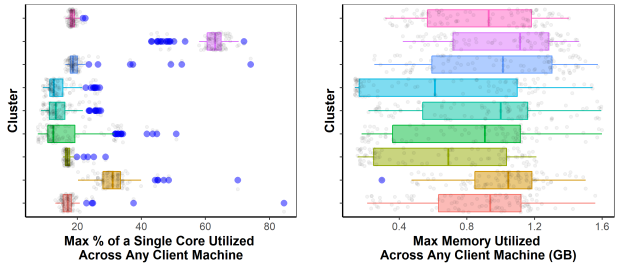


Figure 7: Agent latency distribution for various stages in its data processing lifecycle.



(a) (b)

Figure 8: (a) CPU consumption; (b) Memory consumption.

Figure 7 shows the distribution of the time spent by the agent at various stages in the data processing lifecycle. The "Data Processing" stage captures the latency between the agent observing a change in the underlying file on the source machine and finishing its processing. Next, the "Serialization/Compression" stage captures the time to compress the processed data (e.g., columnar, indexed) and serialize into a binary format. Finally, the "Successful Upload" and "Retry" capture the time it takes to transmit the binary blob to the ingestion backend during and in the presence of a failure, respectively. We observe that the important CPU cycles correspond to the first two stages — our goal has been to minimize these to the extent possible so we can satisy user provided resource constraints. As shown, our current agent deployments take up 10%-30% of the allowed batching time (2-5 minutes) for the multiple streams being ingested on each machine. Figure 8(a) and Figure 8(b) show the respective CPU and memory consumption distributions of our agents across all our deployments. We observe that the agent is consuming 15%-65% of *a single CPU core* to do all the processing of tens of high volume data stream from each customer machine and remains under an allocated memory budget of 1.6 GB.

(Fixed Time vs. Fixed Size) In the example index creation DDL shown in Figure 2, we demonstrated the "fixed chunk (time interval)" policy. It is possible to replace it by a "fixed (chunk) size" policy. However, there is certain performance impact by doing that. If the threshold of chunk size is too small, then it would result in very frequent index updates that can hurt index accessibility. On the other hand, if the threshold is too large, then the index may significantly lag behind the progress log due to untimely index updates.

5.2 How to Provide Fault Tolerance?

The progress log (described in Section 3.3), plays a central role in coordinating the data and the index layers. The progress log is the only strongly synchronized module in the Helios design. It directly affects the write throughput that users experience as writing

to progress log is part of the transaction of committing data into the data layers. For low or moderate ingestion rate, we found it reasonable to rely on an existing database service (Azure SQL, in our case). However, for high ingestion scenarios, we quickly realized that relying on a heavy component such as a remote database, would cause tremendous back-pressure on the entire system.

In our production environments, partly inspired by the need to have a fast durable log [14], we implement our progress log using the RSL-HK library [61] that exposes viewstamped replication [57, 67], consensus [47, 48], checkpointing, and recovery mechanisms. The progress log of <*SequenceId*, *URL*>, is maintained as replicated in-memory Hekaton [36] tables, with an auto-increment *SequenceId* as the primary key.[3] This design guarantees the global datacenter-scoped uniqueness and monotonicity on the sequence Id. Write operations into the log are transactions with ACID semantics realized via optimistic, lock-free techniques in Hekaton, which include writing the transaction log for the primary and replicating it to the replicas. The underlying replicated log is modeled as an infinite redundant stream – an append-only ordered collection of 4 KB blocks. As periodic checkpoints are created, the logs are truncated. The progress log is deployed in quorum-based rings, usually consisting of an odd number of servers (5, in our case) that are appropriately distributed across failure domains. All the meta-data updates are routed to the primary node in each ring/partition.

Despite fault-tolerance from Paxos, there are limitations to using a single log: (1) communication latencies limit overall throughput, especially when replicas are spread over a wide area and (2) progress slows down when a majority fail to acknowledge writes. Therefore, to improve availability and throughput, we use multiple replicated logs, each governing its own partition of the data set. This design allowed us to increase our write throughput from a few hundred writes to hundreds of thousands of writes per second. In our production environments, we have observed a consistent rate of 55,000 writes/sec and 90,000 reads/sec for the meta-data information to be persisted and retrieved from a single partition of the underlying meta-data layer.

5.3 How to Optimize Index Reads/Writes?

Our initial design of the global index contained a strongly synchronized and centralized component (i.e., all updates had to be applied on a set of central servers) and a primary focus on tuple-level strong consistency guarantees. Index creation was synchronously done with data ingestion. While this design kept up with production loads during the first year, we started observing operational problems as we expanded and the number of inserts crossed trillions per day — while partitioning helped to some extent, the overall solution was increasingly becoming expensive as a single replicated partition was unable to keep up with the required write throughput. What's worse was reads were expensive as they had to sift through a large search space consisting of trillions of keys.

To improve writes, we exploited the fact that Helios does not require strong consistency between the index and the data layer, and decoupled the index maintenance from data ingestion. Although the index layer is always lagging the data layer, Helios can leverage the index to the best extent possible, can utilize the progress log

[3]To give a bit more details, in our production environment each datacenter manages its local progress log, which is sufficient since all Cosmos jobs (and their related Helios log analysis queries) are confined in their local datacenters. In other words, there are no cross-datacenter Cosmos jobs and hence no Helios queries demanding a span across datacenters. Within each datacenter, its progress log is multi-partitioned, and the partitions are strongly consistent using our implementation based on RSL-HK.

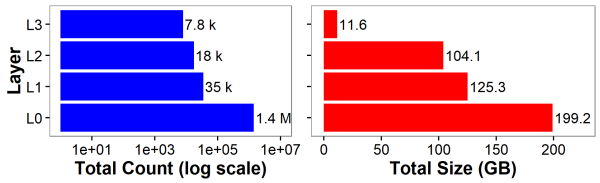


Figure 9: Index size variation across levels.

to figure out the lagging part and can linearly scan the lagging part. Such approach removes the synchronization between the data layer and the index layer, speeding up writes and reducing back-pressure. One side-effect of making everything asynchronous is that the index servers (both primary and replicas) will now need to have access to the underlying index blocks on-demand. A naive implementation would require multiple accesses to the underlying storage to retrieve these index blocks. To avoid this problem, We utilize a local cache spanning memory and SSD across the index servers. Therefore, if an index block was ever retrieved in the course of a hydration process of any server, that block would be available for every other server, saving extra I/Os to the storage system.

To improve reads, we resort to building multiple layers within the index. Upon accumulation of enough data at the leaf layers, the index triggers either a Merge or Add (described in Section 3.1.3) as part of index maintenance. The higher layers in the index were small enough to be held in main memory and support fast queries on the index. In our experiments, we observed that all the necessary compactions (L_0: Base $\rightarrow$ L_1: Base Compacted $\rightarrow$ L_2: 2nd Compaction) for 24 hours worth of data take 2 hours and the final compaction (L_2: 2nd Compaction $\rightarrow$ L_3: Hashed) for the same volume takes less than 30 minutes. Figure 9 presents the sizes of different index levels currently seen in our production environment.

(Read Committed vs. Read Snapshot) While an experimental evaluation that compares *read-committed* and *read-snapshot* is attractive, performing such an evaluation in our production environment is difficult, as customers would choose either read-committed or read-snapshot, but not both of them. In general, read-snapshot is more efficient compared to read-committed, and we observe customers sometimes prefer it. Meanwhile, read-committed allows for reading more recent data and is favorable for applications with more stringent requirements on data freshness.

6. REAL-WORLD USE CASES OF HELIOS

Over the last few years, we have witnessed a large variety of applications built by internal customers using the Helios blueprint. The core technology behind Helios allows for an adjustable cost model and has been used as a substrate for building several verticals (domain-specific applications that cater to different market segments) such as interactive log search and analytics, indexing-as-a-service and IoT analytics in various teams across Microsoft. In this section, we present two representatives among such use cases of Helios within Microsoft.

Monitoring/Debugging Microsoft's Big Data Systems. Cosmos (a.k.a, Azure Data Lake) is the largest data parallel cluster within Microsoft. In such large systems, problems become more of a norm than an exception. As a consequence, debugging becomes very difficult due to the complexity of the system and the scale of the log information being collected — query engines have to be in-place for both quickly pinpointing problems or for more sophisticated root cause analysis. The computational costs for scanning, filtering, and joining the log data in its raw form are prohibitive. The Helios Log Analytics Service (HLAS) is regularly used by 100s of personas (e.g., incident managers, service reliability engineers,

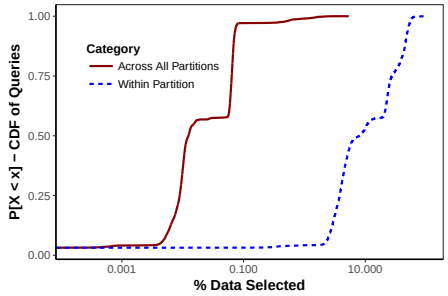


Figure 10: Reduction in Query I/O due to indexes.

and developers) internally for debugging Cosmos/ADL to collect 10s of PBs of log data per day (service footprint) and provides a first-point-of-entry for debugging Cosmos and Azure Data Lake services. We have naturally observed and tuned HLAS for a wide variety of workloads that we summarize below:

- *Needle-in-a-haystack Workloads*: For persons on the critical path of operations (e.g., directly responsible individuals [55], site reliability engineers [39]), the primary goal becomes searching for specific lines (needle) in a large stream of logs (haystack). These style of queries often contain multiple attributes (e.g., timestamps, application identifiers such as JobId, UserId) and touch only a portion of data (typically, less than 10 GBs).
- *Impact and Drill-down Analysis*: An important first question answered when an alert is triggered is whether there was an *impact*, a term that is highly domain-dependent. We have observed domain-experts express complex impact calculation through sophisticated HLAS scripts. These are highly expressive workloads, often retrieving multiple TBs of data, performing complex JOINs, and outputting domain-specific metrics.
- *Performance Monitoring and Reporting*: HLAS is also used for sending out daily reports to service owners (through recurring jobs) and enabling *smart alerts* (i.e., every alert with a link to a query script that can retrieve relevant logs).

Figure 10 shows the CDF of the percentage of data selected (or more specifically, *index selectivity*) for all our query workloads for queries of two types: (1) user queries that contain predicates that are not part of the partition keys of the original data (e.g., JobId == 'job123'), i.e., partition pruning is not possible; and (2) user queries containing predicates that are partition keys and indexed columns (e.g., Filename == 'cosmosErrorLog' ∩ JobId == 'job123'), i.e., partition pruning is possible. We observe that for the former case, the index allows the underlying query engine to prune more than 60% of a partition at the 95^{th} percentile. For queries where only the indexed column is specified as part of the predicates, the index is even more effective — it helps prune more than 99.99% of the data, allowing us to provide queries that execute in less than a few seconds over multiple terabytes of data. We would like to note that we calculated the % data selected assuming a one month retention, i.e., we evaluate what portion of the one month data did the index help prune. As the retention of the data increases, the index may help prune a larger portion of the underlying data.

Indexing Privacy Attributes for Achieving GDPR Compliance. In an effort to protect the personal data and privacy of EU citizens for transactions that occur within EU member states, the European Parliament adopted the General Data Protection Regulation (GDPR) in April 2016, a protection directive expected to set a new standard for consumer rights regarding their data. The GDPR directive requires that companies holding EU citizen data provide a reasonable level of protection for personal data (e.g., identity information, browsing habits, biometric data etc.), including erasing *all* personal data upon request (a.k.a., right to be forgotten). Within Microsoft [54], this is implemented as a "*Forget Me*" button in the user's personal dashboard. Finding the list of data streams that contain the user's information requires a provenance graph (as the number of streams is in the order of 10s of billions) and an index (as each stream can span multiple peta-bytes) to avoid expensive linear scans. As part of processing incoming customer data, there are 1000s of daily indexing jobs that process 100s of TBs. Each indexing job builds an exact multi-level local index (<*Column*, *Value*> → *Stream Region*) and an approximate global index (<*Column*, *Value*> → *Streams*). An entity termed the "Delete Processor" then collects a batch of "Forget Me" requests, utilizes the indexes to resolve all the streams and records that contain data to be deleted, and then issues delete requests to the underlying store.

7. DISCUSSION & LESSONS LEARNED

The technology behind Helios has evolved over the last five years from feedback spanning hundreds of personas including developers, site reliability engineers, customer support personnel and data scientists. In this section, we focus on scenarios and expectations that are leading us to explore new directions.

Less is More for Massive Streams. The immense data volumes and limited human attention necessitates techniques to help identify the most relevant data and trends to allow non-expert users to issue queries. However, due to limited attention span, data access is still governed by reactive patterns (e.g., root cause analyses). In fact, we have observed that for our own case, less than 5% of the collected raw data is actually ever accessed. Of course, this does not reduce demand for collection of raw data. There is, however, growing interest from our users in techniques that summarize data streams. One way of approaching the problem of summarizing massive data streams is to abstract away a new class of streaming *"interpretation"* operators that allow users to aggregate and highlight commonalities of interest, and package them in a way where non-expert users can utilize them. While there has been extensive research in sketching and streaming data structures [23, 25, 26, 29, 30], adapting these to high volumes and deriving useful explanations is still an active research area [12].

Unified Query Experience and Optimizations. Today, it is customary for users to first figure out how to collect data into a big data system. Subsequently, there are a number of telemetry pipelines written that process this data and produce results for reporting or dashboards. A significant drawback of this approach is the extra IO (data needs to be written to disk first) and the delay between getting the data and computing the result (in our experience, a delay of 24 hours is not uncommon). We are observing significant demand from users in terms of avoiding batch telemetry pipelines altogether. While moving all these pipelines into existing standard streaming engines is possible, it may be costly and expensive to do all the processing in the cloud. The ingestion engine behind the Helios Log Analytics service evolved from a simple data collection layer to a layer performing full-fledged indexing on high volume streams. It is not unreasonable to imagine running batch queries on the same architecture [44] instead of having a dedicated batch query engine. In Helios, this translates to coming up with efficient techniques for splitting computation between end devices, edge, and cloud. This solves the problem of maintaining code that needs to produce the same result in two complex distributed systems. Based on our production experience, we posit that a single engine model can (1) enable optimizations such as automatically converting recurring batch queries into streaming queries, dynamically mapping operators/computations to various layers (e.g., end

device, edge, cloud), automated A/B testing for figuring out the best query execution plans, joint query optimization in multi-tenant scenarios and automatic cost-based view materialization, and (2) help significantly reduce user and operational burden of having to learn and maintain multiple complex stacks.

IoT Scenarios. In IoT Analytics-related scenarios where it is not unusual to have billions of devices that can sense, communicate, compute, and potentially actuate. Data streams coming from these devices will challenge traditional approaches to data management and contribute to the emerging paradigm of big data. Given the volume of data, the blueprint of Helios is a natural fit — agents can be deployed to customer-owned IoT sensors and edge devices to summarize data on-the-fly and make it available for querying with low-latency. This also opens up new challenges for making the agent operate under resource-constrained (e.g., power) environments.

Augment, Not Automate. The role of a data engineer is complex, error-prone and involves a lot of experimentation with data. When setting up big data pipelines, many human and compute years are lost in fine tuning them for target workloads, e.g., the common case of incorrectly partitioning the dataset without considering the incoming queries. Traditional solutions to these problems incorporate recommendation (e.g., for indexes [24], views [5], and partitions [6]) as a first-class user experience. While it is interesting to integrate index recommendation into Helios, this is perhaps insufficient — our mailing lists would still be filled with questions on why a certain query is slow. We believe the space is ripe for disruption. While there is already significant research happening in fully automating database concepts [59], we would like to see more research in the area of "explainable recommendations" that provides users with not only recommendation results, but also explanations to clarify why such strategies have been recommended.

8. RELATED WORK

Helios is a blueprint that builds on a long history of techniques from distributed systems (e.g., [4, 15, 16, 19, 21, 38, 51, 52, 61, 66, 69]), specialized for processing and indexing massive data streams with reduced capital and operational costs. The query execution techniques within Helios are similar to those described in the shared-nothing literature [34, 35]. The data storage and layout within Helios share properties with other well-described systems including BigTable [22], HBase [7], Cassandra [46] Megastore [13], Yahoo! PNUTS [28] and Dynamo [33]. Rather than striving towards becoming the "one-size-fits-all" system, Helios adopts the philosophy of "made-to-measure" — users have total (and easy) control over splitting their streaming indexing job between end devices, edge, and cloud. This view enables users to choose between a wide range of cost models. For instance, users can choose to pay less (by allowing the agent to do pre-processing and indexing on-premise) or pay more (by allowing the cloud do all the indexing).

To avoid concurrent actions across data centers, some systems, such as Amazon's Dynamo [33], resort to weaker consistency semantics like eventual consistency where state can temporarily diverge. Others, such as Yahoo! PNUTS [28], avoid state divergence by requiring all operations that update the state to be funneled through a primary site and thus incurring increased latency. Our global index offers timeline/snapshot consistency [13, 28]. The design ideas behind the global index are similar to Deuteronomy [49] and Aurora [68]. The key idea behind scaling the global index was to leverage a *replicated log* to store the physical location of the delta index rather than the index itself.

The emergence of edge computing has raised new challenges for big data systems [32, 56]. In typical application scenarios of edge computing, such as smart cities, operational monitoring, and Internet of Things, continuous data streams must be processed under stringent latency constraints. In recent years, a number of distributed streaming systems have been built via either open-source or industry effort (e.g., Storm [65], Spark Streaming [10], Flink [18], MillWheel [8], Dataflow [9], Quill [20]). These systems, however, adopt a cloud-based, centralized architecture that does not include any "edge computing" component — they typically assume an external, independent data ingestion pipeline that directs edge streams to cloud storage endpoints such as Kafka [45] or Event Hubs [2]. From this point of view, Helios is an effort towards building a new genre of big data systems that combine the cloud and the edge as a single, holistic data processing platform, by pushing down significant computation to the Helios *agents* running on the edge. Moreover, unlike Helios, existing distributed streaming systems (e.g., Storm, Spark Streaming) do not support indexing as far as we know.

Tree-structured indexes are ubiquitous in database management systems for accelerating query processing. Typical examples include ones that target general relational queries such as B-trees [27] and LSM tress [58], and ones that target special classes of queries such as R-trees [40] and quad-trees [62] that target spacial data. Due to this ubiquity, there were even proposals of generalized search trees (GiST) for database systems [41] with its own concurrency and recovery models [42], which have been implemented in PostgreSQL. The design and implementation of the zone indexes in Helios is mostly inspired by these previous works on tree-structured indexes. However, as Helios operates in hyper-scale data ingestion scenarios, we have to loosen the requirement of a global sort-order within each index level, as typically enforced by tree-structured indexes. We draw inspiration from LSM trees by employing policy-based mechanisms to merge newly ingested data into Helios indexes periodically and asynchronously. We have also made the operations over Helios indexes and the corresponding policies as general as possible, in the spirit of GiST. In recent years, there has also been work on index structures that utilize modern hardware. For example, Bw-tree [50] is such an index structure that is mainly designed for in-memory transaction processing. It cannot be easily adapted for the big data scenarios that Helios tackles, where the query workloads are analytical and I/O (instead of memory and CPU) is the key performance factor to be optimized. Moreover, in the big data context, indexes need to be partitioned and distributed across multiple machines, which is also not supported by Bw-tree to the best of our knowledge.

9. CONCLUSION

We have described Helios, a distributed system for flexible ingestion, indexing, and aggregation of large streams of real-time data at Microsoft. We described *Helios Index*, the data structure backing our production system. The hierarchical and recursive nature of the Helios index allows each layer to be (1) computed at different stages of the data collection life cycle, (2) held at a different tier in a storage layer, and (3) utilized independently at query time. This flexibility allows for a full spectrum of *cost* and *performance* tradeoffs that users can choose from, while operating reliably at scale.

Helios builds and maintains indexes on thousands of columns over petabytes of data per day. Helios clusters have been in production for the last five years, collecting close to a quadrillion log lines per day from hundreds of thousands of machines spread across tens of datacenters. Helios serves several internal use cases at Microsoft and has been deployed for mission critical debugging/diagnostic needs. The system has also served as a reference blueprint for other large-scale systems within Microsoft. Some of the core query optimization techniques from Helios have been open-sourced as part of Hyperspace [3].

10. REFERENCES

[1] Azure sql data warehouse. https://docs.microsoft.com/en-us/azure/sql-data-warehouse/.

[2] Event hubs. https://azure.microsoft.com/en-us/services/event-hubs/.

[3] Hyperspace. https://github.com/microsoft/hyperspace.

[4] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB*, 2003.

[5] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *VLDB*, pages 496–505, 2000.

[6] S. Agrawal, V. R. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD*, pages 359–370, 2004.

[7] A. S. Aiyer, M. Bautin, G. J. Chen, P. Damania, P. Khemani, K. Muthukkaruppan, K. Ranganathan, N. Spiegelberg, L. Tang, and M. Vaidya. Storage infrastructure behind Facebook messages: Using HBase at scale. *IEEE Data Eng. Bull.*, 2012.

[8] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *PVLDB*, 6(11):1033–1044, 2013.

[9] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB*, 8(12):1792–1803, 2015.

[10] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. Structured streaming: A declarative API for real-time applications in apache spark. In *SIGMOD*, pages 601–613, 2018.

[11] M. Armbrust et al. Spark sql: Relational data processing in spark. In *SIGMOD*, pages 1383–1394, 2015.

[12] P. Bailis, E. Gan, K. Rong, and S. Suri. MacroBase, A Fast Data Analysis Engine. In *ACM SIGMOD*, 2017.

[13] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, 2011.

[14] P. A. Bernstein, C. W. Reid, and S. Das. Hyder-a transactional record manager for shared flash. In *CIDR*, volume 11, pages 9–20, 2011.

[15] T. Bingmann, M. Axtmann, E. Jöbstl, S. Lamm, H. C. Nguyen, A. Noe, S. Schlag, M. Stumpp, T. Sturm, and P. Sanders. Thrill: High-performance algorithmic distributed batch data processing with c++. In *IEEE Big Data*, 2016.

[16] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: cloud computing for everyone. In *Procs. of SOCC*. ACM, 2011.

[17] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *SOSP*, 2011.

[18] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[19] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.

[20] B. Chandramouli, R. C. Fernandez, J. Goldstein, A. Eldawy, and A. Quamar. Quill: Efficient, transferable, and rich analytics at scale. *PVLDB*, 9(14):1623–1634, 2016.

[21] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *PVLDB*, 8(4):401–412, 2014.

[22] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *TOCS*, 2008.

[23] M. Chao. A general purpose unequal probability sampling plan. *Biometrika*, 69(3):653–656, 1982.

[24] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for microsoft SQL server. In *VLDB*, pages 146–155, 1997.

[25] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewer. Failure diagnosis using decision trees. In *Procs. of Autonomic Computing*. IEEE, 2004.

[26] J. Cheng, Y. Ke, and W. Ng. A survey on algorithms for mining frequent itemsets over data streams. *KIS*, 2008.

[27] D. Comer. The ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[28] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.

[29] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 2012.

[30] G. Cormode, F. Korn, and S. Tirthapura. Exponentially decayed aggregates on data streams. In *ICDE*. IEEE, 2008.

[31] I. D. Corporation. What will we do when the world's data hits 163 zettabytes in 2025? *https://goo.gl/iwsPww*, 2017.

[32] M. D. de Assunção, A. D. S. Veith, and R. Buyya. Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. *J. Network and Computer Applications*, 103:1–17, 2018.

[33] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS*, 2007.

[34] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The gamma database machine project. *TKDE*, 1990.

[35] D. J. DeWitt, A. Halverson, R. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling. Split query processing in polybase. In *Procs. of SIGMOD*. ACM, 2013.

[36] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Procs. of SIGMOD*. ACM, 2013.

[37] L. Fife. How much does 1 hour of downtime cost the average business? https://goo.gl/fqqvTW, 2017.

[38] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SIGOPS*. ACM, 2003.

[39] Google. Site reliability engineering. *https://goo.gl/YwqcQL*, 2017.

[40] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[41] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *VLDB*, pages 562–573, 1995.

[42] M. Kornacker, C. Mohan, and J. M. Hellerstein. Concurrency and recovery in generalized search trees. In *SIGMOD*, pages 62–72, 1997.

[43] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *SIGMOD*, pages 489–504, 2018.

[44] J. Kreps. Questioning the lambda architecture. *https://goo.gl/5Es6N9*, 2014.

[45] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

[46] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[47] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[48] L. Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[49] J. J. Levandoski, D. B. Lomet, M. F. Mokbel, and K. Zhao. Deuteronomy: Transaction support for cloud data. In *CIDR*, 2011.

[50] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The bw-tree: A b-tree for new hardware platforms. In *ICDE*, pages 302–313, 2013.

[51] L. Mai, L. Rupprecht, A. Alim, P. Costa, M. Migliavacca, P. Pietzuch, and A. L. Wolf. Netagg: Using middleboxes for application-specific on-path aggregation in data centres. In *Procs. of CoNEXT*. ACM, 2014.

[52] L. Mai, K. Zeng, R. Potharaju, L. Xu, S. Suh, S. Venkataraman, P. Costa, T. Kim, S. Muthukrishnan, V. Kuppa, S. Dhulipalla, and S. Rao. Chi: A scalable and programmable control plane for distributed stream processing systems. *PVLDB*, 11(10):1303–1316, 2018.

[53] R. C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, pages 369–378, 1987.

[54] Microsoft. GDPR Compliance. *https://goo.gl/2KkwMv*, 2017.

[55] Microsoft. Rotating devops role improves engineering service quality. *https://goo.gl/x63caG*, 2017.

[56] M. Mohammadi, A. I. Al-Fuqaha, S. Sorour, and M. Guizani. Deep learning for iot big data and streaming analytics: A survey. *IEEE Communications Surveys and Tutorials*, 20(4):2923–2960, 2018.

[57] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Procs. of SPDC*. ACM, 1988.

[58] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 1996.

[59] A. Pavlo et al. Self-driving database management systems. In *CIDR*, 2017.

[60] R. Ramakrishnan and J. Gehrke. *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[61] R. Ramakrishnan, B. Sridharan, J. R. Douceur, P. Kasturi, B. Krishnamachari-Sampath, K. Krishnamoorthy, P. Li, M. Manu, S. Michaylov, R. Ramos, et al. Azure data lake store: A hyperscale distributed file service for big data analytics. In *Procs. of ICMD*, pages 51–63. ACM, 2017.

[62] H. Samet. Hierarchical spatial data structures. In *SSD*, pages 193–212, 1989.

[63] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Readings in Artificial Intelligence and Databases*, pages 511–522. Elsevier, 1988.

[64] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *MSST*, 2010.

[65] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. V. Ryaboy. Storm@twitter. In *SIGMOD*, pages 147–156, 2014.

[66] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm@ twitter. In *Procs. of SIGMOD*. ACM, 2014.

[67] R. Van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, pages 91–104, 2004.

[68] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Procs. of ICMD*. ACM, 2017.

[69] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*. ACM, 2013.