Db2 Event Store: A Purpose-Built IoT Database Engine

Christian Garcia-Arellano, Adam Storm, David Kalmuk
Hamdi Roumani, Ronald Barber◇, Yuanyuan Tian◇
Richard Sidle◇, Fatma Ozcan◇, Matthew Spilchen
Josh Tiefenbach, Daniel Zilio, Lan Pham
Kostas Rakopoulos, Alexander Cheung, Darren Pepper
Imran Sayyid, Gidon Gershinsky◇, Gal Lushi◇
Hamid Pirahesh◇

eventstore@ca.ibm.com

IBM Cloud and Cognitive Software ◇ IBM Research

ABSTRACT
The requirements of Internet of Things (IoT) workloads are unique in the database space. While significant effort has been spent over the last decade rearchitecting OLTP and Analytics workloads for the public cloud, little has been done to rearchitect IoT workloads for the cloud. In this paper we present IBM Db2 Event Store™, a cloud-native database system designed specifically for IoT workloads, which require extremely high-speed ingest, efficient and open data storage, and near real-time analytics. Additionally, by leveraging the Db2 SQL compiler, optimizer and runtime, developed and refined over the last 30 years, we demonstrate that rearchitecting for the public cloud doesn't require rewriting all components. Reusing components that have been built out and optimized for decades dramatically reduced the development effort and immediately provided rich SQL support and excellent run-time query performance.

PVLDB Reference Format:
Christian Garcia-Arellano, Adam Storm, David Kalmuk. Hamdi Roumani, Ron Barber, Yuanyuan Tian, Richard Sidle, Fatma Ozcan, Matthew Spilchen, Josh Tiefenbach, Daniel Zilio, Lan Pham, Kostas Rakopoulos, Alex Cheung, Darren Pepper, Imran Sayyid, Gidon Gershinsky, Gal Lushi, Hamid Pirahesh. Db2 Event Store: A Purpose-Built IoT Database Engine. *PVLDB*, 13(12): 3299-3312, 2020.
DOI: https://doi.org/10.14778/3415478.3415552

1. INTRODUCTION

1.1 The Needs of IoT systems

With the rapid proliferation of connected devices (smart phones, vehicles, buildings, industrial machinery, etc.) there has never been as great a need to persist data, and make it available for subsequent analytical processing, as quickly as possible. Traditionally, the data storage and analytics needs of the Internet of Things (IoT) space have been serviced by relational databases, time series databases, or more recently, elements of the Hadoop ecosystem [1]. Unfortunately, none of these technologies were designed for the specific demands of Internet of Things use cases, which include:

- **Extremely high-speed data ingestion**: It is not uncommon in IoT use cases to see data arriving at a rate of millions of data points per second. As the data is typically machine generated, and not generated by humans, it arrives steadily around the clock, resulting in hundreds of billions of events per day.
- **Efficient data storage**: Due to the large volume at which data is arriving, efficient storage is essential. This requires that the system stores data in a highly compressed format, leverages cost effective storage (such as cloud-based object storage), and ideally automates data retention through techniques such as Time to Live (TTL). Moreover, since IoT datasets grow so rapidly, it is desirable to store the data in an open data format, so that it can be directly queried by additional runtime engines and does not require migration should the system be re-platformed in the future.
- **Real-time, near real-time and deep analytics**: Persisting data is never the end goal. IoT systems require that the data be made available as quickly as possible to both queries which are interested in a given data point (or collection of data points from a given sensor), as well as more complex queries which leverage the full power of SQL. Additionally, IoT data often feeds ML models, such as those used for predictive maintenance.
- **Continuous Availability**: IoT data stores are fed by remote data sources, often with limited ability to store data locally. As a result, in cases where the data store is unavailable for a prolonged period of time, remote data sources may overflow their local storage, resulting in data loss. Similarly, queries being run on IoT systems are often critical to the business, and even a small interruption could result in significant financial impact [2]. To ensure a complete data history and consistent business insights, IoT systems must remain continuously available.

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415552

1.2 Related Work: Existing Technologies for IoT Workloads

Traditionally, relational database management systems (RDBMSes) have been used to handle the most challenging of data problems. As a result, over the last four decades they have evolved from a focus on high speed transaction processing [3], to servicing high performance analytics [4] [5], and more recently, hybrid transactional/analytical processing (HTAP) workloads [6], which require both fast transactions and analytics. In this time, they have evolved from running on a single machine, to scaling out to tens or hundreds of machines to leverage both MPP techniques, and/or for high availability. While this evolution has created shared data architectures that on the surface seem similar to what we propose in this paper [7] [8], the leveraging of shared storage in these systems is principally for availability, and is not an attempt to leverage cheap and plentiful storage, as is required for IoT workloads. Furthermore, the dependence on mutable data and strong consistency in most relational systems makes it difficult to leverage eventually consistent cloud-based object storage.

While RDBMSes are unquestionably versatile, their generalization prevents them from being ideal for several recently identified use cases [9], one of which is IoT workloads. In IoT workloads, machine generated data is often produced at a rate of millions of events per second – orders of magnitude faster than human generated transactions. For example, an IoT system producing 1M events/sec (a rate common in the IoT space) will generate more events in a single day than the total number of US stock market trades in a year [10]. This tremendous volume of arriving data plays against the strengths of traditional relational database systems whose WAL techniques favour transactional consistentcy at the expense of ingest performance. Additionally, since traditional relational database systems update and delete in-place and require strong consistency, they are not able to directly leverage cloud object storage - the most cost-effective way to store large volumes of data in the public cloud. While recent cloud-native relational database systems are now able to leverage cloud object storage [11] [12] and have made a concerted effort to separate compute and storage to improve transactional availability [13], they have not yet gained widespread adoption for IoT workloads.

Over the last decade, time series databases have dramatically increased in popularity [14] and have become the repositories of choice for IoT data. While time series databases are able to rapidly ingest data and store time series data efficiently, many of them, such as InfluxDB [15], Kdb+ [16], use a non-SQL compliant query language [17] [18], and struggle with true continuous availability [19]. There do exist time series databases which support SQL, such as TimescaleDB [20], however we show in section 3 that they have limitations in terms of ingest and query performance. Also, none of these time series databases leverages an open data format or can directly leverage object storage to efficiently handle the massive volume of data generated by IoT systems.

To counter the limitations of time series databases, a wave of Hadoop and open-source based systems, like Apache Druid [22], have been employed for high speed data use cases with some success [23] [24] [25]. These systems are typically architected according to the Lambda Architecture whereby one data store is used to persist data quickly (typically a time series database or KV store) and provide near real-time analytics, while a second system is used for deep analytics [26]. The Lambda Architecture however, suffers from complexity (multiple systems to maintain), stores a non-trivial portion of the data in two places (resulting in higher storage costs), and is difficult to query, as application designers have to understand which of the disparate systems to query for a given use case. Furthermore, many of the systems, on which Lambda is built, struggle to achieve the required ingest speeds required for IoT workloads, without a significant hardware footprint [27].

More recently systems have emerged which are tackling similar requirements to those found in IoT systems. Unfortunately, these newer systems are either restricted to internal use [28] or are opaque [29, 30].

1.3 A Purpose-Built IoT System

To address the specific needs of IoT use cases, we built Db2 Event Store [31], which has the following design principles:

- **Make ingest as fast as possible:** The system is designed to do the minimum amount of synchronous work required to persist data, ensure durability in the presence of node failures, and make it available to query processing.
- **Asynchronously refine and enrich data:** Once data has been ingested, it is further refined and enriched asynchronously, to make query processing more efficient.
- **Highly optimize query processing:** Efficient query processing leverages a highly optimized open data format, meta-data constructs which allow for data skipping, and a robust and mature query optimizer and runtime. Furthermore, queries leverage all available hardware by running in parallel across all nodes in the cluster and also multi-threaded within each node.
- **Ensure continuous availability for both ingest and queries:** Each component of the system is designed to be fully redundant. In the event of node failures, data is quickly assigned new leaders amongst the surviving members to ensure minimal disruption to ingest and queries.

In the remainder of this paper we describe the Db2 Event Store architecture in detail, as well as some of the challenges faced in building the system. We then compare it to two existing systems in wide used for IoT use cases, both from a functional and performance perspective. Finally, we discuss some of the remaining challenges to be overcome.

2. ARCHITECTURE

2.1 Architectural overview

Db2 Event Store leverages a hybrid MPP shared nothing / shared disk cluster architecture . The combination of shared nothing and shared disk architectures allows it to combine the linear scalability attributes of a traditional MPP shared nothing data store with the superior availability, and cloud-native characteristics of a shared data system. The system is constructed by combining a new cloud native storage layer with Db2's existing BLU MPP columnar database engine [5].

Table data is physically divided into micro-partitions based on a user defined hash partitioning key, and stored on a reliable shared storage medium such as cloud object storage (e.g. IBM Cloud Object Storage [32]) , a network attached storage device (such as IBM Cloud Block Storage [33]), or a cluster filesystem (such as

Ceph [34] or IBM Spectrum Scale [35]). The entire dataset is fully accessible by all compute nodes in the cluster, decoupling the storage from the compute and enabling each to be scaled independently.
MPP shared nothing scale-out for ingest and query processing is achieved by logically dividing the leadership of micro-partitions across the available compute nodes, coordinated through the consistent meta-store Apache Zookeeper [36]. Each micro-partition is logically owned by one and only one compute node at any given point in time and any requests to read or write that micro-partition are executed via the owning compute node, enabling query and ingest parallelism across the cluster. By ensuring a sufficiently large number of micro-partitions relative to the number of compute nodes the system ensures sufficient granularity in the data partitioning to allow an even distribution of data leadership across the compute nodes in the presence of node failures. When handling node failures, the affected micro-partitions are logically reassigned amongst the remaining compute nodes that are replicas of the failed micro-partitions, substantially reducing the failover time compared to a model that requires data migration [37], and allowing processing to continue with minimal disruption.

To ensure that database metadata does not become a single point of failure and to maintain the desired continuous availability, the system implements the concept of a “floating” catalog node where catalog information is stored on a reliable shared filesystem and exposed via a logical node that can be restarted on any of the compute servers in the event of a failure.

The architecture is depicted below:

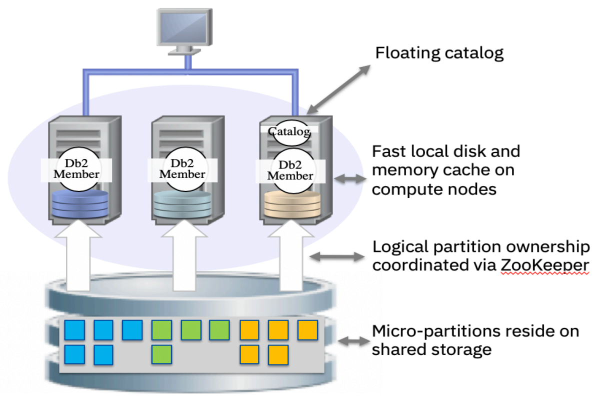
Floating catalog
Catalog
Db2 Member
Db2 Member
Db2 Member
Fast local disk and memory cache on compute nodes
Logical partition ownership coordinated via ZooKeeper
Micro-partitions reside on shared storage

Figure 1 MPP shared nothing / shared disk architecture

Fast ingest in this architecture is achieved through a combination of parallelism and optimized storage techniques. The system implements a headless cluster architecture where all compute nodes play the dual role of head and data nodes. This allows ingest to be parallelized across the entire cluster and removes any potential for a head node ingest bottleneck. Within the server, ingested records are mapped to specific micro-partitions by Db2’s MPP hash partitioning function and shipped to the owning members via Db2’s internal cluster communication infrastructure.

On the storage side, data is written into fast local storage (SSD or NVMe) devices present on each of the compute nodes, and it is further replicated and written to the local storage on at least two other compute nodes before acknowledging commits. This ensures availability of the data in the event of a compute node failure. The data is then asynchronously written to durable shared storage. This model allows the system to avoid any extra latency that might otherwise be incurred from the shared storage medium (particularly in the case of high latency cloud object storage) and allows the architecture to efficiently accommodate both small and large inserts. It should be noted that this model implies that micro-partition leadership must be aligned with the replica locations, but this affinity is also desirable as it has the beneficial side effect of enabling better data cache locality. Ingested data can be optionally aged out of the system via a time-to-live (TTL) mechanism, configured at the table level.

The ingested data is stored in an open data format (Apache Parquet [38]) which leverages a compressed PAX storage format which can be efficiently utilized for analytics processing by Db2’s BLU columnar runtime engine. The use of an immutable open data format also allows for the data to be queried directly by external runtime engines (e.g. Hive, Spark, Presto). In order to support the cost and scalability benefits of cloud object storage, which may not have strong consistency guarantees for modified data, the system leverages an append-only immutable storage model where data blocks are never re-written. Synopsis metadata information, which allows for data skipping during query processing, is automatically generated as part of the ingest process and is written in separate Parquet files, allowing for synopsis data to be accessed and cached separately from the table data. Indexing is also supported and is implemented as an UMZI index [39] leveraging an LSM-tree style format to adhere to the append-only requirement. Indexes are generated asynchronously as part of the data sharing process to minimize the latency on ingest processing. This also means that duplicate key elimination occurs during data sharing processing. Db2 Event Store implements a first-writer-wins (FWW) semantic when ingesting into tables with primary keys defined. FWW ensures that the first landed version of a row (where a distinct row is defined by its associated primary key) is never replaced if in the future different versions of the same row (i.e. the primary key is the same) are ingested.

Db2 Event Store query processing is SQL based, and is enabled through Db2’s industrial grade Common SQL Engine (CSE) which includes its cost-based SQL optimizer and BLU columnar runtime [4]. The Db2 CSE is integrated into the Db2 Event Store storage layer and allows for the ability to exploit Db2 Event Store’s synopsis-based data skipping as well as its indexes. This integration allows Db2 Event Store to offer high speed analytics using parallel SQL processing, JDBC/ODBC connectivity, and full Db2 SQL compatibility.

Last but not least, efficient data access is achieved through a multi-tiered caching layer that caches frequently accessed data, synopsis and index blocks in-memory, and also on fast local storage on the compute nodes, in order to absorb latencies that would otherwise be incurred when exploiting cloud object.

2.2 Data format and meta-data

2.2.1 Leveraging an open data format - Apache Parquet

When choosing a data storage format, it was desirable to leverage one that was columnar organized, due to the performance advantages when processing analytics workloads. In addition, it was highly desirable to utilize an open format, that would allow access by external readers and avoid data lock in. In search for an open column organized format we decided on Apache Parquet [38], as it is widely adopted and supported by many readers (e.g. Spark, Hive, Presto [40]). In addition, Apache Parquet is a self-describing format, which is accomplished by including the schema (column name and type) in each of the files.

Db2 Event Store uses Snappy [41] compression to reduce the storage footprint. Snappy compression was chosen as it represented the best trade-off between storage size and ingest/query impact. GZIP [42] and LZ4 [43] were also considered as options, but GZIP incurred a much higher overhead for both ingest and query performance, and at the time of our initial evaluation LZ4 was new to the Parquet specification and did not have widespread adoption in some of the Parquet readers. We plan to reinvestigate LZ4 compression now that it is more prevalent in the Parquet ecosystem.

As described above, the Db2 Event Store architecture leverages a micro-partitioned data model. With this architecture the finest granularity of a table is a *table micro-partition*. Parquet files belong to micro-partitions and thus a given Parquet file contains the data of exactly one table micro-partition. The Parquet files are immutable once written. Each Parquet file for a table micro-partition is assigned a monotonically increasing number, referred to as its *tablet identifier*. The Db2 Event Store runtime engine and external readers use this to infer that higher tablet identifiers represent newer data. The metadata for the Parquet files in shared storage is maintained in Apache Zookeeper. This includes the high watermark tablet identifier of each micro-partition.

Within the Db2 Event Store engine, tuples are identified through a Tuple Sequence Number (TSN), an integer that may be used to locate a given tuple within a table. A TSN in Db2 Event Store includes the tablet identifier, the zone (Pre-Shared vs Shared; when data is ingested it moves through several zones which are described in detail in section 2.3), and the offset of the tuple within the Parquet file.

Writing and reading of Parquet files within the database engine is done using an open source C++ parquet library [44]. To minimize the amount of read IO we implemented a custom reader in the C++ parquet library that serves read requests from a local cache. The local cache is discussed in more detail in Section 2.4 Multi-tiered caching.

2.2.1.1 *Encrypting Parquet Data*

Db2 Event Store is able to securely handle sensitive user data. Since data is kept in the Apache Parquet format, we have worked with the Parquet community to design and implement a security mechanism, built into the format itself. The specification of Parquet Modular Encryption [45] was released in the Apache Parquet Format repository, and its C++ open source implementation was merged in the Apache Arrow repository [46] and released as part of version 0.16.0.

Parquet is a complex format, comprised of different data and metadata modules that enable efficient filtering (columnar projection and predicate push-down) by the analytic engines that process the data. Parquet Modular Encryption encrypts each module separately, thus preserving the filtering capabilities and analytics efficiency with the encrypted data. It leverages the AES GCM cipher [47], supported in CPU hardware, in order to perform module encryption operations without slowing down the overall workflow. Besides protection of data privacy, AES GCM also allows to protect the integrity of the stored data, making it tamper-proof against malicious attempts to modify the file's contents. The size overhead of Parquet encryption is negligible, since only the modules (such as compressed data pages) are encrypted, and not the individual data values.

Encryption keys, used for securing privacy and integrity of Db2 Event Store data, are managed by the native Db2 Key Management System [48], that handles safe storage and rotation operations for these keys. Parquet files are encrypted before being sent to the shared storage – therefore, the encryption keys and the plaintext data are not visible to the storage backend. After retrieval of encrypted files from the shared storage, DB2 Even Store verifies cryptographic integrity of the processed data, using the Parquet Modular Encryption libraries. Additionally, the SSD/NVMe caching layer also uses the Parquet encryption format, ensuring that locally persisted files are protected against privacy and integrity attacks.

2.3 Ensuring fast ingestion

One of the key design characteristics to allow Db2 Event Store to handle the fast ingestion rate common in IoT scenarios is that it organizes the data in a table into multiple zones, and it evolves the data asynchronously from one zone to the next as the data ages. As described in Section 2.1, the data is first ingested into fast local storage. This is the first data zone called the Log Zone. Data in this zone is replicated to remote nodes to ensure durability and high availability. From the Log Zone, the data is moved to additional zones with the goal to persist it on cost effective shared storage, make it available to external readers, and most importantly, continuously optimize it for query performance. There are two additional zones: the data is first moved from the Log Zone to the Pre-Shared Zone, and then from the Pre-Shared Zone to the Shared Zone. All of these zones are transparent to the end user, who sees a single view of the table without having to worry about the continuous evolution of the data through the zones. All of these zones are immutable. One critical design point is that the log *is* the database – the Log Zone is not just written to local storage to guarantee durability like in traditional database systems that implement write-ahead logging, but is also directly utilized to service query results. Figure 2 shows a high-level view of the zones and the evolution of the data from one to the next one.

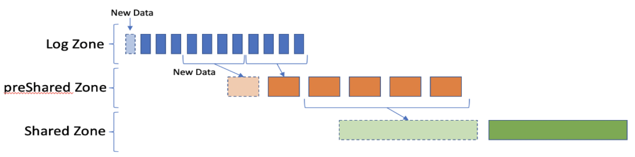

Figure 2 Data evolution through zones

2.3.1 *Logging ingestion*

Ingest processing starts with the Db2 Event Store client, which provides an API for asynchronous batch inserts. This client can connect to any of the nodes to perform ingest, and in the case of failures will automatically resubmit the batch insert to any of the other nodes. The node that the client is connected to is referred to as the "ingest coordinator". The role of the node acting as a coordinator for the batch is to perform the hash partitioning of the batch into the corresponding micro-partitions, and then direct the micro-partition batch to one of the micro-partition replicas. When the micro-partition batch is received by a micro-partition replica it is placed in the Log Zone in the form of log buffers, that are persisted to local storage and replicated to multiple remote replicas, each of which persist to local storage before acknowledging, to guarantee durability and availability.

Each micro-partition maintains its own set of replicas and the insert is only considered successful once it has been acknowledged by a quorum of replicas (R/2+1) for each micro-partition impacted. By default, the replication factor R is 3. Both replication and acknowledgements are batched to improve the efficiency of the replication and to guarantee both the persistence and availability of the ingested data. This data is available for querying as soon as it is replicated to a quorum of nodes (i.e. before the data is enriched through synopsis or indexing).

To ensure high performance ingest, log data must be stored on fast local storage (SSD or NVMe devices) to ensure low latency for the only synchronous portion of the data persistence lifecycle.

2.3.2 *Data enrichment*

The Log Zone enables quick persistence and durability, but it is not optimal for querying. As a result, recently ingested data must be moved to a more query friendly format as soon as possible and be enriched with additional data structures, like indexes and synopsis. This allows for more efficient querying, at the expense of the additional latency (in the order of seconds), which is sufficient for most IoT applications.

The next zone after the Log Zone is the Pre-Shared Zone, which is stored in shared storage. The process of "sharing" a table micro-partition can be done by any of the replicas, as all have a copy of all the log buffers, but Db2 Event Store gives preference to the micro-partition leader, which can be re-assigned dynamically on failures or due to load re-balancing. As an iteration of data persistence to the Pre-Shared Zone is completed, the transition point between zones, and the Pre-Shared Zone tablet metadata, is tracked in the consistent meta-store Apache Zookeeper. This allows for a seamless transfer of leadership between nodes and enables consistency of the objects in shared storage. Data that has been pre-shared is subsequently purged from the Log Zone, which is completed when Log Zone readers are drained out from the already persisted area (queries that are reading this data from the Log Zone must complete before the data can be purged).

Moving data from the Log Zone to the Pre-Shared Zone is fast (typically on the order of seconds). As a result, Parquet files written to the Pre-Shared Zone may be small, as each Pre-Share iteration can only consider the data written since the last Pre-Share iteration, and therefore not optimally sized for query processing (where large files – on the order of 10s or 100s of MBs are ideal). For this reason, once there is enough volume of data in the Pre-Shared Zone, the small files are consolidated to generate much larger Parquet files in what is called the Shared Zone. Larger files enable more efficient query processing and better overall data compression so typical consolidated files are in the range of 64MB in size. The Shared Zone is the final zone and so files remain there forever (or until the expiration time is reached if TTL is configured for the table). Finally, to avoid having multiple copies of data, the Pre-Shared Zone is also purged, like the Log Zone, once a set of files from the Pre-Shared Zone have been written successfully to the Shared Zone, the sharing state is registered in the consistent meta-store, and the pre-shared files purge is initiated once they are drained of concurrent queries.

2.3.3 *Persistence to cheap storage*

The database engine was designed to exploit a storage hierarchy that includes memory, fast local storage within each node for fast persistence, and finally cost-efficient storage to maintain the very large volume of data.

One of the challenges of supporting cost-efficient object stores is that their consistency guarantees vary. For that reason, the files written by Db2 Event Store are never updated, and Apache Zookeeper is used as consistent data store to record the state of objects in the cost-efficient object store. The other challenge of cost-efficient object stores is their performance, and for this reason in Section 2.4 we discuss the multi-tiered caching, that significantly reduces the performance impact of accessing files in persistent storage.

2.3.4 *Building Indexes and Synopsis*

Providing an indexing structure for an IoT data store is challenging for multiple reasons. First of all, at the rate of ingest that Db2 Event Store was designed to support, the volume of data grows rapidly. As an example, for a 3-node cluster, at 1 million inserts per second per node, with 40-byte events, the volume grows by 3.5 PB/year or 9.5 TB/day of uncompressed data. This volume of data is one of the motivating factors for supporting cost-efficient object storage, but this brings about new challenges: dealing with eventual consistency, and the high latency reads and writes discussed above. To get around the eventual consistency limitation of updates to object storage, index files must be written once, and never overwritten. A final challenge is to provide a unified index structure that can index the data across the multi-zone architecture. All of these requirements must be satisfied while still providing very fast index access for both range and point queries.

The index in Db2 Event Store is a multi-zone index, covering the Pre-Shared and Shared Zones. The index does not cover the Log Zone as it would require synchronous maintenance and maintaining the index synchronously would increase insert latency. Furthermore, it is more efficient to maintain the index asynchronously when large amounts of data are moved to the Pre-Shared Zone, which happens only seconds after being ingested into the Log Zone.

The index within a zone follows an LSM-like structure with both multiple runs and multiple levels, based on the UMZI index [39]. This kind of indexing structure is well suited for the high volume of writes, as it can be constantly re-optimized.

As the data pre-sharing is generating an Apache Parquet file from the data in the Log Zone, it also generates the corresponding compressed index run (See Figure 3 Index Runs together with the pre-sharing processing). Generating the index run at this time is also very efficient, as the complete run generation is done in memory, from the data in the Log Zone, which is also in memory.

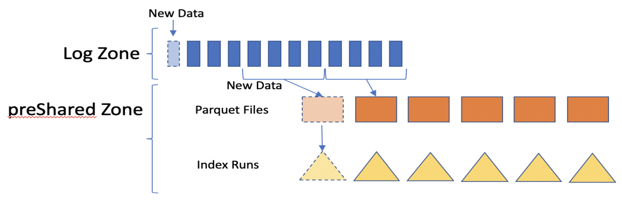

Figure 3 Index Runs together with the pre-sharing processing

With IoT data, duplicate values are relatively uncommon, and are typically the result of a sensor sending the same data value multiple times (often in close succession). For this reason, duplicates are most commonly found in the recently ingested data, which is still resident in memory. To ensure no duplicate data, when a new index run is being generated from the Log Zone, the system performs index lookups to ensure the primary key uniqueness is maintained. The system keeps the most recent index runs and an index synopsis for older runs in the local cache. The index synopsis data, which contains primary key ranges that for IoT systems always include a timestamp, is particularly helpful in ensuring efficient primary key uniqueness lookups since the bulk of the data already loaded into the table will have older timestamp values. Since the local cache is multi-level to exploit the storage hierarchy, the index runs are maintained both in memory and in the local SSD/NVMe devices, and the most efficient look ups are for runs that are still in the in-memory level of the cache. We will discuss the details of the multi-level cache management in Section 2.4.

In the same way that the data is evolved from the Pre-Shared to the Shared Zone, the index is also evolved, by merging multiple index runs into a single and much larger index run. As the volume of data grows, and the number of runs grows, the performance of the index would degrade without merging as more runs must be consulted for each index lookup. For this reason, the system maintains multiple levels of the index, continuously merging runs from one level into larger runs in the next level to reduce the number of overall runs. As new runs are generated in the next level, the runs in the previous level are purged by a background garbage collection process, to reduce the storage cost. All of this is done while still maintaining the consistency of the index, both for in-flight queries and in the persisted copy, so that the index can be rebuilt on restart.

Another important point to note is that all the runs from all levels are persisted to shared storage. This is required as the micro-partition assignment to nodes is dynamic, so the persistence of runs to shared storage is required to allow the transfer of micro-partitions from one node to another. When a micro-partition is re-assigned, the database engine initiates a background process to warm up the local cache of the new micro-partition leader, therefore enabling index access to reach top performance again as quickly as possible.

To enable data skipping in table scans, Db2 Event Store automatically creates and maintains an internal synopsis table for each user-created table. Similar to the data synopsis of Db2 BLU [4] and IBM Cloud SQL Query [49], each row of the data synopsis table covers a range of rows of the corresponding user table and contains the minimum and maximum values of the columns over that row range. Blocks of the data synopsis are also in Parquet format. Note that the data synopsis, when applicable, is likely to be cached (see Section 2.4) as it is small and accessed fully in each table scan that qualifies for data skipping. This data synopsis, which is distinct from the index synopsis, is populated as data are consolidated into the Shared Zone and so does not cover the data of the Pre-Shared and Log Zones. Maintaining data synopsis content for these zones would come at considerable additional cost (extra writes to shared storage) and would provide little value from data skipping given that the volume of data in these zones is small.

2.4 Multi-tiered caching

Traditionally modern high-performance DBMS systems either are in-memory, and thus rely on RAM for performance, with resiliency coming from multi-node replication and a weak story on power outages (with either slow full cache rebuilds or worse yet, data loss), or use high performance network storage like IBM Cloud Block Storage [33]. Given that cheap cloud Object Storage, on the order of $0.01 USD/GB/month [50] is high latency, and high-performance storage is generally at least 10X the cost of object storage, the challenge is how to leverage inexpensive storage and still provide optimal performance. The approach taken by Db2 Event Store is multi-tiered caching for both data and index objects, that is able to leverage both memory and fast local storage to insulate the system from the high-latencies of Object Storage.

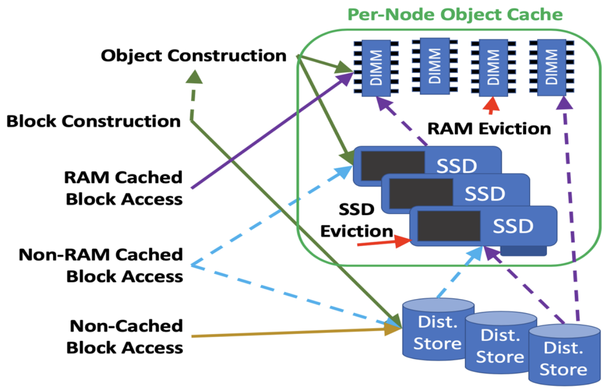

Figure 4 Caching Stages and Migration

Note that the DIMM memory indicated at the top of this figure shows the cache managed RAM memory. Naturally there is other memory used for processing and transient buffering.

2.4.1 Challenges of leveraging high-latency storage

Db2 Event Store employs many mechanisms to take full advantage of caching within its Cache Manager component.

- **Multi-layered Caching:** To insulate the system from Object Storage latencies, both local SSD/NVMe devices and RAM are used for caching of data block and index objects. While utilizing a main memory cache is of course not new, mixing it with SSD/NVMe and the introduction of an epoch-based lock free eviction technique does introduce novelty.
- **Directed Caching:** The multi-tiered caching offers several access methods as outlined above in Figure 4**.** Data and index block construction creates the objects in both the cache and object storage. Cache access may be directed based on request type to either RAM only, as is used by some short-lived index structures in the LSM tree, or directed to local SSD/NVMe devices with optional placement in the RAM cache as well. The cache manager also allows for non-RAM cached reads, utilizing only SSD/NVMe for which callers supply a private buffer when it is known that retrieved content is unlikely to be re-used, such as for retrieval of data that will be rewritten soon into a more concise format or data that is copied to data structures outside the cache manager (e.g. synopsis information for query acceleration). Also, the caching tiers may be bypassed completely allowing direct cloud storage access when accessing potentially cold table data. This is generally related to probabilistic caching described below.
- **Probabilistic Caching:** To deal with a limited cache size as compared with the table or index data, the Cache Manager utilizes probabilistic techniques similar to those used in Db2 BLU [4]. These techniques avoid the cache flooding issue when high volume accesses like large table scans occur, but will still build up a cache of the hot objects over time. The subsystem

leverages statistics on the total sizes of the different table and index data for the objects being accessed, and computes a probability of caching block requests relative to data used by a given query, and the total managed cache space. The decision is performed by the block storage layer utilizing statistics maintained by the cache manager. Making the decision up-stream allows for taking into account semantic information like a priority for synopsis caching vs. traditional table data caching.

- **Soft & Hard Limits:** The caching infrastructure uses a soft limit for memory while imposing a hard limit on non-volatile storage usage. RAM based caching evictions (discussed below) happen generally in a relaxed fashion, engaging a background process at 100% of target utilization and attempting to bring memory utilization to 80% before resting again. However, should the RAM target be found to exceed 125%, which may occur in a very busy system, then a more aggressive eviction technique is utilized by performing immediate releases upon dereference of any memory objects. For SSD/NVMe the Cache Manager provides extensions of space up to a requested target limit, but when that is reached, caching requests are blocked, and the users of the subsystem will fall back to RAM only caching, or skip caching altogether depending on the use case.
- **Epoch based eviction:** The eviction of both the RAM data and the SSD/NVMe data is managed by a lock free epoch-based technique that utilizes a small, on the order if 1 byte, epoch id. This both saves metadata storage space and allows for quick scans with fewer TLB misses. Atomic operations that obtain reference counts protect objects from going away at inopportune moments without the costs of mutex operations. More detail on the LRU cache replacement algorithm is provided in the next section.

2.4.2 Batchwise-LRU Cache Replacement

Caching for a cloud-native database system such as Db2 Event Store introduces distinct challenges:

- Object sizes vary widely in Db2 Event Store, from small objects such as index run meta-data objects (small number of KBs) and the small data blocks of the most recently data in the Pre-Shared Zone, to the large consolidated data blocks of the Shared Zone and the blocks of higher level runs of the index (both of which are on the order of 100 MBs). Existing cache replacement algorithms, such as those used in traditional DBMS buffer pools and operating systems, are designed to handle small, fixed-size pages and are reactive in nature, evicting a page on demand when a new page is needed. Such page replacement methods are not well-suited to handling objects of widely varying sizes.
- Adaptive cache replacement algorithms, such as LRU, LRU-2 [51] and ARC [52], that are known to work well for a wide variety of workloads, often require global locks at object access and/or eviction time. Such global locks limit scalability in the presence of concurrent accesses.

To address these caching challenges, Db2 Event Store implements a batchwise-LRU eviction algorithm using epochs, that is scalable and works well for variable-sized objects.

Cache eviction in Db2 Event Store is done by background threads, with one thread for object eviction from cache-managed RAM and a second thread for eviction from cache persistent node-local storage. Object eviction is triggered when cache usage for the given storage type reaches a configurable start threshold (e.g. 95%) and objects are evicted until usage is reduced to the stop threshold (e.g. 90%). This pro-active eviction ensures that there is space in the cache at all times for objects of all sizes at the cost of a small loss of cache space (<10%). Tracking of object accesses is done using epochs. The epochs (which wrap over time) are much smaller than full timestamps, with only 1 byte needed to be able to evict in 1% of cache size increments. The use of small epoch values minimizes the memory overhead for tracking object accesses. In addition, for objects in persistent cache storage, the access epochs are recorded contiguously in an array in the object directory memory of the cache. Contiguous storage and small epoch size enable the LRU objects to be identified efficiently via a scan for the *purge epoch*, which is the oldest active epoch in the system. At eviction time, objects last accessed in the purge epoch are evicted until either none remain, or the target threshold is reached. If more cache space is to be freed, the purge epoch is incremented, and objects last accessed in this epoch are evicted. Given that there is a single background thread, no global lock is needed for eviction. Epochs are implemented as atomics and as they are updated relatively infrequently, recording object accesses does not limit scalability.

2.5 Optimized query processing

For query processing Db2 Event Store leverages Db2's BLU MPP cluster query engine, combining the power of Db2's cost-based SQL optimizer [53] and the accelerated analytic processing of Db2's BLU acceleration columnar engine [4], with the fast ingest and cloud native storage capabilities of the Db2 Event Store data management layer. The BLU MPP engine was chosen due to its ability to efficiently process both low latency and complex queries, its mature query compilation/rewrite/optimization capabilities, and of course, its accessibility within IBM. At the project's outset (and for Db2 Event Store's first few releases) the system leveraged Apache Spark as a query engine but we found that it could not efficiently handle low latency queries, and its performance trailed traditional data warehouses as query complexity increased (principally because of its relatively immature query optimizer).

The use of Db2's BLU MPP SQL engine allows the system to offer a rich set of standards compliant SQL capabilities including full distributed joins, grouping + aggregation, ordering, and OLAP functions. Given this range of SQL functionality, it is relatively straightforward to leverage Db2's date and timestamp manipulation functions to group events into buckets of a desired time granularity and then leverage Db2's grouping and OLAP analytic functions to perform all manner of real-time, near-real-time or deep analytics on time series data.

2.5.1 Client Interfacing

Db2 Event Store's SQL interfacing is done through Db2's existing JDBC / ODBC client infrastructure supporting the full range of Db2 connectivity. Incoming statements are compiled and optimized through Db2's cost based multi-stage SQL optimizer [53] and leverage Db2's access plan caching infrastructure to avoid the need to recompile plans on every execution.

2.5.2 MPP Plan Generation and Optimization

The Db2 optimizer generates columnar query plans that are fully MPP aware producing a series of "subsections" (executable subplans) that are distributed across active compute members and executed in parallel. MPP specific optimization decisions include selecting appropriate distributed join strategies (collocated, broadcast, hash directed), application of partial vs. final aggregation and ordering, in addition to decisions on join strategies, join ordering, and predicate pushdown decisions, as well as the application of semantically equivalent query rewrites to improve performance.

2.5.3 *Query Execution*

The parallel execution of query plans is performed by Db2's MPP BLU runtime layer which allows us to take advantage of its existing highly parallelized column-store processing strategies. Execution in the BLU runtime layer is broken down into multiple stages where a stage consists of the sequential execution of a series of operators called "evaluators". Each stage takes a single table / column scan as input and employs vectorized processing where input data is passed in as column-oriented vectors, and each evaluator performs data transformations producing one or more output vectors as results. Each query thread has its own distinct evaluator chain and executes in parallel. Query threads perform work-stealing on the single input and fully process a batch of input rows through all evaluators before obtaining another batch. Constructs such as joins that combine input streams from multiple tables are executed in multiple stages – for example the build phase of a join taking input from one table would be executed as one stage and the probe stage of the join taking input from the other table would be executed as a subsequent stage. Cross-member communications during processing is achieved through several different classes of table-queue evaluators which act as either an input stream of columnar data from one or more members or an output stream of columnar data to one or more members distributing data based on the distribution required by the plan (e.g. broadcast vs. hash directed). Data is exchanged in a network optimized columnar format and the underlying communications infrastructure is optimized for highly parallel batch data exchange.

2.5.4 *Interfacing with the Micro-Partitioned Storage*

At the data management layer, the Db2 BLU runtime has been extended to support hash directed data flows, table / columns scans, and index lookups on the micro-partitioned storage layer.

In order to support distributed sub-section processing and hash directed data exchange, the logical micro-partition leadership is represented internally as a Db2 "partition map" which abstracts the mapping of hash key values to individual members. Interfacing at this layer enabled us to transparently leverage the existing Db2 MPP runtime without requiring significant changes. The partition mapping itself is generated by first generating a fixed mapping of keys to micro-partitions based on the Db2 MPP hash partitioning function, and then resolving the member associated with each micro-partition based on current ownership, producing a key to member mapping. Since the mapping of micro-partitions is logical and can change dynamically upon node failure / failback, the system also associates a version number with a given partition map. This number corresponds to a parallel version number maintained in Apache ZooKeeper which is incremented any time the micro-partition mapping is altered. By comparing the two versions it can quickly determine if the partition map is out of date and needs to be refreshed. At section initialization time the partition map version that a query is executing is stored to allow validation that the partition mapping remains current throughout the execution. In the event that a partition map version change is detected during query execution, the query execution will be aborted, but the query can be immediately re-submitted and will execute using updated mappings.

Table / column scans have been implemented by adapting the corresponding evaluators in BLU to scan column data from the set of micro-partitions associated with the member executing the scan and producing standard BLU column vectors as input to the BLU runtime. The lower level scanning is done through the multi-tier caching layer abstractions and also leverages the available synopsis data in order to do data skipping.

Index lookups are similarly implemented by adapting the existing index scan logic in Db2 to interface with the UMZI indexes and produce rows in the output format that the Db2 runtime expects. The physical scan of a given index starts in the smaller runs that have been most recently generated, and as it moves through it, it jumps from level to level as it finds the transition points.

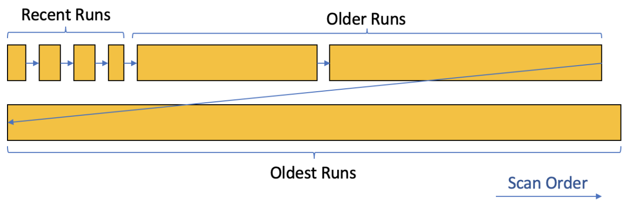

Figure 5 Scan of Index Runs

There is one other element that Db2 Event Store implements to be able to provide accelerated query performance when using the index: for each run it also maintains an in-memory synopsis of the index run, that allows it to quickly eliminate the run without actually performing any IO. This could be considered an additional level in the index, but unlike the others, this is a level that is only kept in memory.

2.6 Continuous Availability

One of the key design principles for Db2 Event Store was to create a database system that would be able to provide continuous availability. This required not only designing a system without single points of failure, but also one that was always available even during planned outages. Since all replicas of a table micro-partition are able to process ingest, the ingest processing is minimally impacted when a replica node is lost, as long as a majority of replicas are still available. The main impact for clients that were connected directly to the failed node is that the client must handle such a failure and re-try the operation in a different node.

The rest of the operations on a table micro-partition have a higher dependency on the availability of the table micro-partition leader. These include queries and background tasks, like data movement between zones, data enrichment and index optimization processes. For these, through decoupling table micro-partitions from the nodes that act both as replicas and replica leaders, Db2 Event Store is able to maintain the availability, as long as a majority of replicas are still available to guarantee that only durable data is processed. By maintaining the node states and database meta-data in a consistent meta-store like Apache Zookeeper, the system is able to quickly identify table micro-partition leaders that become unavailable and seamlessly transfer the leadership of micro-partitions to other nodes to continue processing both queries and background tasks.

The graph in Figure 6 shows an example of the impact to both ingest and query processing during a single node failure for a 3-node cluster. The impact to ingest processing is partial, as only those clients connected to the failed node are impacted. In the case of queries, they are impacted until the table micro-partitioning leadership is restored for all micro-partitions. In this diagram, the failed node recovers with minimal impact to the workloads.

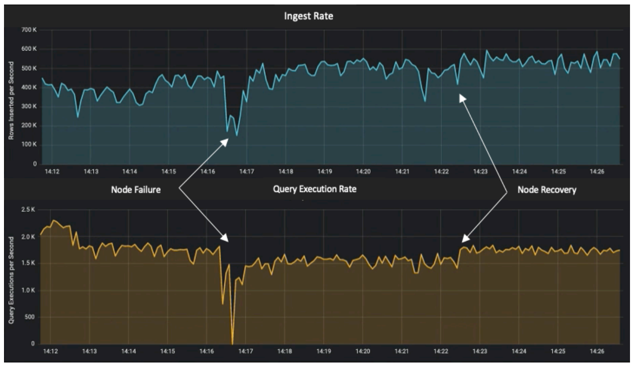

Figure 6 Performance during node failure

The other components of the system are also designed for continuous availability. In the case of the database catalog node, the system is able to continue processing ingest and queries when the catalog service is down, as the table metadata is cached by data nodes and they only depend on the catalog node to perform initial cache population. The database catalog node uses shared storage for persistence, which allows it to fail over to any of the hosts that are part of the cluster. In the case of Apache Zookeeper, it is also continuously available as long as a majority of the Apache Zookeeper nodes are still active.

3. PERFORMANCE

In this section we examine the performance of Db2 Event Store in two dimensions. First, we illustrate the scalability characteristics of Db2 Event Store, comparing the load and query performance for different cluster sizes against a single node baseline. Second, we compare Db2 Event Store with competitors in the time series DBMS space.

3.1 Db2 Event Store Cluster Scalability

For the Db2 Event Store scalability experiments workload we used the Time Series Benchmark Suite (TSBS) [54, 55], created by TimescaleDB [20]. TSBS is a complete IoT benchmarking suite that is based on a model of a trucking company. The TSBS IoT schema consists of three tables, shown in Table 1 TSBS Schema Fields. Tags is a dimension table containing metadata about the trucks. Readings and Diagnostics are large fact tables containing sensor metrics from the trucks. Readings contains the current truck location metrics, and Diagnostics the current truck status metrics. The data size can be scaled through three parameters: the number of trucks, the data sample rate (seconds between metrics), and the time duration, where the first parameter determines the size of Tags and all three determine the sizes of Readings and Diagnostics. Also provided is a suite of 13 queries, described in Table 3 that each examine one of the fact tables and the dimension table, to answer different styles of questions. To ease the comparisons, we chose to categorize these queries into three clusters: global analysis, localized analysis and targeted queries. Global analysis queries (Q1-Q6) consider data over the life of the vehicle, for example average driver driving duration per day or average versus projected fuel consumption per fleet. Localized analysis queries (Q7-Q11) filter more on specific events like trucks with high load or trucks with longer driving sessions. The last category is targeted analysis (Q12-Q13) where these queries focus on a narrow selection of data, for example last location by specific truck and stationary trucks at a specified interval. We made one change to the TSBS query set by adding a low-bound time predicate to the localized analysis queries. This is explained in more detail below.

In the Db2 Event Store scalability test we created a large workload using parameters based on a major shipping vendor, with 84,700 trucks, 10 days of data, and sensor metrics every 10 seconds, resulting in 13.16 Billion rows.

Table 1 TSBS Schema Fields

TSBS Shema Overview		
Readings	**Diagnostics**	**Tags (Metadata)**
Tags Id	Tags Id	Id
Time	Time	Name
Latitude	Fuel state	Fleet
Longitude	Current load	Driver
Fuel consumption	Truck status (moving or stationary)	Nominal fuel consumption
Elevation		Fuel capacity
Velocity		Load capacity
Heading		Truck model
Grade		Device version

The multi-node tests were performed on clusters of 3, 6, and 9 physical machines, each with 20 2.8GHz cores, 386GBs of RAM, 1 local SSD (NVMe) and using a remote NFS server using a RAID array of 4 SSDs. The single node test was performed on a larger node of 28 2.4GHz cores, 1.5TB of RAM and 3 direct attached SSDs. This system was configured for Db2 Event Store cache to use 200GB SSD and 100GB memory. For TimescaleDB the buffer pool was restricted to 100GB.

First, we look at the scalability of ingestion by collecting timings from a single node system and comparing this to clusters of 3, 6, and 9 nodes. These results show over 75% scalability when looking at 3, 6 and 9 nodes. The single node system exhibits the best per-node performance as it has the advantage of not performing log replication on ingest. Cluster configurations, on the other hand, provide triple log replication for durability, as described in Section 2.3.1.

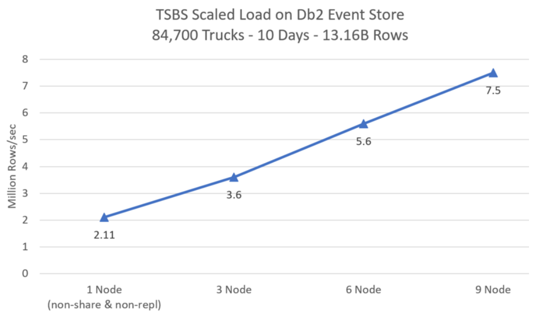

Figure 7 TSBS Scaled Load on Db2 Event Store

Second, we look at the scalability of query processing in a cluster by collecting timings from the single node system and comparing this to a cluster of 3, 6, and 9 nodes. These results show good scaling at up to 6.9X for 9 nodes, as well as the same benefits of a single node system, which has the advantage of no network overhead when running the queries.

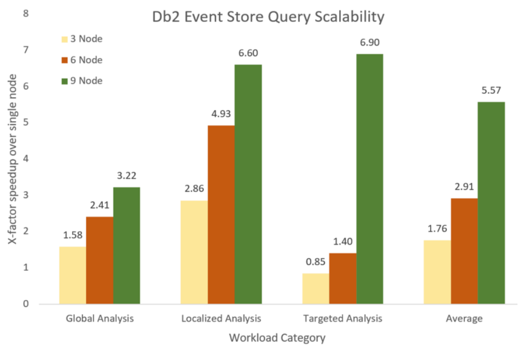

Figure 8 DB2 Event Store Query Scalability

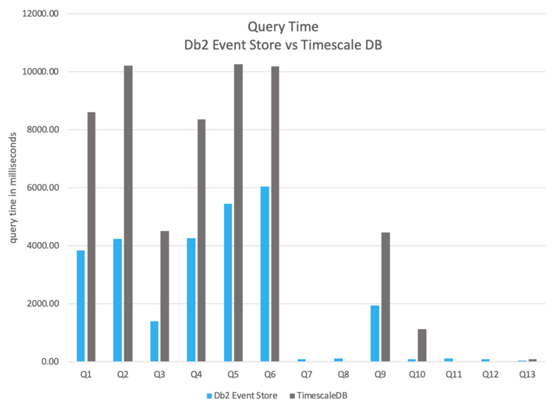

Figure 9 Query Time - Db2 Event Store vs TimescaleDB

3.2 Comparison with Other DBMSes

As described previously, there are many existing database management systems (DBMSes) that can be used to service IoT workloads [56, 16]. In our experience however, time series databases are most commonly used for IoT workloads and as a result, they serve as the most suitable comparison point. We chose two of the more popular time series database, Apache Druid and TimescaleDB for the evaluation [22] [20]. Apache Druid is in widescale use at many large-scale customers. TimescaleDB has both devoted significant effort to benchmarking, developing and publishing a complete IoT benchmarking suite which we leverage in our scalability tests in Section 3.1, as well as publishing benchmark results against other time series DBMSes. A recent report comparing TimescaleDB with InfluxDB [57] demonstrated that TimescaleDB outperforms InfluxDB (currently the most popular open source time series database <reference db-engines>) in both ingest and query performance.

3.2.1 Comparison with TimescaleDB

When comparing with TimescaleDB, we use the TSBS benchmark but this time at a smaller scale and focus on a single node setup for comparison of both Db2 Event Store and TimescaleDB, as at the current time, TimescaleDB does not support clustering. We built a data set with 4,000 trucks and 3 days of data with sensor metrics every 10 seconds resulting in 187M rows.

In terms of load time we found that TimescaleDB performed best using a 10K row batch size and 6 workers. We also loaded Db2 Event Store using a 10K row batch size and 6 clients with 18 shards. This comparison showed Db2 Event Store to load data in less than 2 minutes vs the 37 minutes, representing a 21X performance advantage.

Table 2 Ingest Time - Db2 Event Store vs TimecaleDB

Load Time	Time (sec)	Rows/sec	Metrics/sec
TimescaleDB	2256.1	82,755	413,771
Db2 Event Store	105.0	1,776,159	8,879,020

Next, for the query test, we took one representative query instance from each of the 13 query types and ran it 5 times, after a warmup run, which allowed data to be in memory for both Db2 Event Store and TimescaleDB, and then we computed an average time over the 5 runs. The query descriptions, and performance results, are shown in Table 3. The results show that Db2 Event Store is approximately 2X faster than TimescaleDB on all but a few of the very low latency queries.

Table 3 TSBS Benchmark Query Results

Description	Query	Db2 Event Store (ms)	TimescaleDB (ms)
Average driver driving duration per day	Q1	3,843.35	8,622.34
Average driver driving session without stopping per day	Q2	4,253.82	10,221.31
Average load per truck model per fleet	Q3	1,396.73	4,527.74
Average vs projected fuel consumption per fleet	Q4	4,265.14	8,376.96
Truck breakdown frequency per model	Q5	5,452.62	10,275.07
Daily truck activity per fleet per model	Q6	6,037.50	10,183.94
Trucks with high load	Q7	90.39	15.68
Last location per truck	Q8	129.32	15.61
Trucks with longer daily sessions	Q9	1,947.86	4,478.34
Trucks with longer driving sessions	Q10	90.39	1,143.39
Trucks with low fuel	Q11	129.32	16.51
Last location for specific truck	Q12	105.45	0.44
Stationary trucks in an interval	Q13	48.84	85.76

On these very low latency queries (all of which complete in less than 150ms) TimescaleDB benefited from its partitioned Hypertable indexes, while Db2 Event Store was not able to leverage its index due to an existing limitation with its index plan generation. We plan to address this limitation in the future.

We chose to add a low-bound time predicate to the queries analyzing the latest truck status (Q7, Q8, Q11, Q12) as we have found that having such bounds is common in customer scenarios, either driven by a UI that shows status for a selected window of time, or by query guarantees like all queries being limited to activity in the last hour. The lower bound can be easily obtained by periodically running a query to find the minimum report time from the set of maximum report times of the desired vehicles. We ran these slightly modified queries in Db2 Event Store and TimescaleDB for all our comparisons.

3.2.2 Benchmark with Apache Druid

Apache Druid is similar to Db2 Event Store in its mandate to ingest large volumes of data and perform near real-time analytics. Additionally, it is also designed for high availability, and built to scale to very large clusters. Finally, Apache Druid has been around for a number of years, has been performance optimized in that time, outperforms other Hadoop-based options by a large margin [21], and has become increasingly popular for a number of large use cases in that time [22].

For the performance comparison with Apache Druid we also leveraged an existing benchmark, in this case a benchmark that was developed by the Apache Druid team [27] to compare their performance to MySQL. The benchmark contains nine single table SQL queries based on the LINEITEM table from the TPC-H benchmark. Each of the queries either perform a count, sum, group by, or order by, commonly found in IoT workloads.

```
-- count_star_interval
SELECT COUNT(*) FROM LINEITEM WHERE L_SHIPDATE BETWEEN
'1992-01-03' AND '1998-11-30';
-- sum_price
SELECT SUM(L_EXTENDEDPRICE) FROM LINEITEM;
-- sum_all
SELECT SUM(L_EXTENDEDPRICE), SUM(L_DISCOUNT), SUM(L_TAX),
SUM(L_QUANTITY) FROM LINEITEM;
-- sum_all_year
SELECT YEAR(L_SHIPDATE), SUM(L_EXTENDEDPRICE),
SUM(L_DISCOUNT), SUM(L_TAX), SUM(L_QUANTITY) FROM
LINEITEM GROUP BY YEAR(L_SHIPDATE);
-- sum_all_filter
SELECT SUM(L_EXTENDEDPRICE), SUM(L_DISCOUNT), SUM(L_TAX),
SUM(L_QUANTITY) FROM LINEITEM WHERE L_SHIPMODE LIKE
'%AIR%';
-- top_100_parts
SELECT L_PARTKEY, SUM(L_QUANTITY) FROM LINEITEM GROUP BY
L_PARTKEY ORDER BY SUM(L_QUANTITY) DESC LIMIT 100;
-- top_100_parts_details
SELECT L_PARTKEY, SUM(L_QUANTITY), SUM(L_EXTENDEDPRICE),
MIN(L_DISCOUNT), MAX(L_DISCOUNT) FROM LINEITEM GROUP BY
L_PARTKEY ORDER BY SUM(L_QUANTITY) DESC LIMIT 100;
-- top_100_parts_filter
SELECT L_PARTKEY, SUM(L_QUANTITY), SUM(L_EXTENDEDPRICE),
MIN(L_DISCOUNT), MAX(L_DISCOUNT) FROM LINEITEM WHERE
L_SHIPDATE BETWEEN '1996-01-15' AND '1998-03-15' GROUP BY
L_PARTKEY ORDER BY SUM(L_QUANTITY) DESC LIMIT 100;
-- top_100_commitdate
SELECT L_COMMITDATE, SUM(L_QUANTITY) FROM LINEITEM GROUP
BY L_COMMITDATE ORDER BY SUM(L_QUANTITY) DESC LIMIT 100;
```

3.2.2.1 Initial Benchmark and Results

While results exist for Druid runs on this benchmark, none are recent, so we reran the workload with Druid before comparing with Db2 Event Store. To do this we setup an Apache Druid cluster (using Hortonworks HDP version 3.1.0 which contains Druid version 0.12.1) on 3 physical machines, each with 28 2GHz cores, 386 GBs of RAM and 2 direct attached SSDs running HDFS. On the same hardware, we setup Db2 Event Store version 2.0 using the same directly attached SSDs for local storage, and an NFS storage target also on SSDs to closely match the I/O characteristics of the Druid system. We then configured Druid for the environment by moving the segment cache location to the locally attached SSDs and increasing the number of processing threads to 55. Db2 Event Store used its default configuration, without any modifications.

3.2.2.2 Ingesting data

We completed the data load using the TPC-H Scale Factor 100 data set (100 GB, over 600 million rows), which we generated using the publicly available tools for TPC-H [58], and ingested it into both Druid (using the Hadoop Batch method) and Db2 Event Store (via insert-sub-select from an external table in CSV format).

The ingest performance difference between Druid and Db2 Event Store was significant. Inserting the data into Druid took 2 hours and 8 minutes while the insert into Db2 Event Store took 12 minutes and 10 seconds — a 10.6x difference.

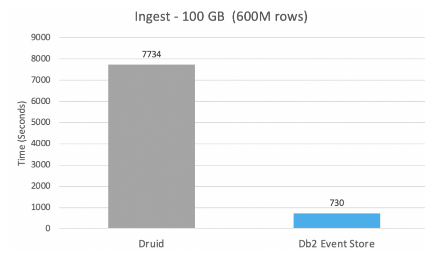

Figure 10 100 GB data set ingestion vs DRUID

3.2.2.3 Initial query performance

We then ran the nine benchmark queries in a batch and found that Druid took 76.4 seconds to complete the run, while Db2 Event Store completed in only 40.4 seconds - nearly 1.9x faster than Druid, as shown below.

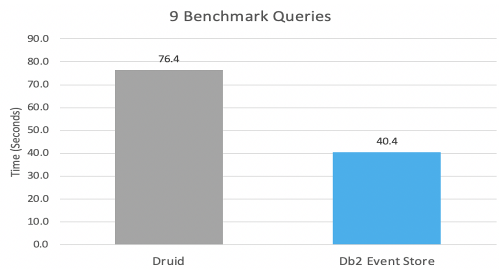

Figure 11 Query performance with 9 query workload vs DRUID

When we looked at the queries one-by-one, we found that in all but two queries, Db2 Event Store outperformed Druid, often significantly. In the case of Q1, the query includes a count(*), and Druid can make use of metadata it keeps to more efficiently compute the result. It is also important to note that for the queries that use LIMIT 100 to produce an exact result in Db2 Event Store (Q6, Q7, Q8, and Q9), the same queries in DRUID are written using their TopN function, which uses an approximation algorithm that is not guaranteed to produce the exact query results and must be configured explicitly [59]. The benchmark with the query response time of each of the queries can be seen in Figure 12.

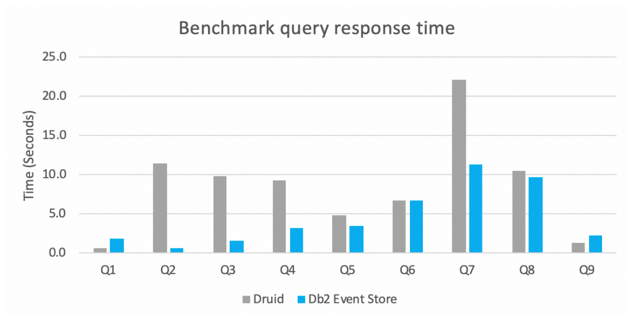

Figure 12 Query response time vs DRUID

3.2.2.4 Adding more complex queries

Analysis of the benchmark's queries revealed that from an analytics workload perspective, the queries had fairly low complexity. As a result, we supplemented the workload with two additional queries which would provide slightly more significant complexity (grouping by one or two non-key columns):

```
SELECT SUM(L_EXTENDEDPRICE) FROM LINEITEM GROUP BY
L_ORDERKEY ORDER BY SUM(L_EXTENDEDPRICE) DESC FETCH FIRST
100 ROWS ONLY
```

```
SELECT SUM(L_EXTENDEDPRICE) FROM LINEITEM GROUP BY
L_PARTKEY, L_ORDERKEY ORDER BY SUM(L_EXTENDEDPRICE) DESC
FETCH FIRST 100 ROWS ONLY
```

When these queries were run against Db2 Event Store and Druid, we found that Db2 Event Store further outperformed Druid. The first more complex query took 31 seconds to run on Db2 Event Store and 861 seconds to run on Druid (a difference of more than 27x), while the second query took 159 seconds to run on Db2 Event Store, and 1819 seconds to run on Druid (a difference of more than 11x). See the comparison in Figure 13**.** These results illustrate the benefits of leveraging Db2's mature query compilation, optimization and runtime layers as query complexity increases.

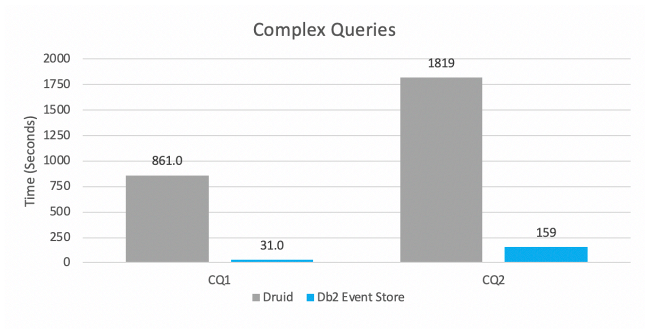

Figure 13 Complex query response time vs DRUID

While these additional queries are slightly more complex than the original benchmark's queries, they are by no means complex. For truly complex queries we'd need to include joins, which are generally not supported by IoT databases, including Apache Druid [60].

4. FUTURE WORK

There are several improvements we are considering for future releases of Db2 Event Store.

Faster queries, enhanced by secondary indexes - Currently Db2 Event Store tables only support a single unique index, created using a primary key. While this serves most IoT queries well, there are some queries which benefit from index access on columns not included in the primary key. To improve the performance of these types of queries we're investigating secondary index approaches which achieve the desired query performance gains, while not sacrificing the system's ability to rapidly ingest data.

Online scalability - While Db2 Event Store can scale from a single node to larger clusters, it currently hardens cluster size at installation. We are currently investigating the modifications required to allow Db2 Event Store clusters to scale both up and down instantly. As the system was architected for scalability, this work is largely an engineering effort.

Fine grained update/delete - As described in section 2.3.2, Db2 Event Store currently only supports removal of data through a time-to-live mechanism (TTL). To allow for a broader set of use cases, we have work underway to modify our data format to allow for fine grained update and deletes. This work is challenging as we plan to continue to use Apache Parquet (which does not natively support data modification), continue to support cloud-based object storage which is eventually consistent, and allow for high performance fine grained updates and deletes which may only modify a single row in a given file.

5. CONCLUSIONS

In this paper we present a novel, cloud-native database system designed specifically for IoT workloads to provide extremely high speed ingest, efficient data storage, and near real-time analytics. The system is continuously available to both ingestion and queries in the presence of node failures, and stores data in an encrypted open data format on cloud native object storage. By leveraging the Db2 compiler, optimizer and runtime layers, the system significantly outperforms existing IoT database systems – especially as queries get more complex – and is able to perform complex analytics queries which include joins, sorting and aggregation. Future work will enhance the system to allow it to scale online and provide fine grained updates and deletes.

6. ACKNOWLEDGMENTS

The Parquet encryption work in this paper has been funded in part by the EU Horizon2020 ProTego project (n 826284).

7. REFERENCES

[1] Cloudera, "Apache Hadoop Ecosystem," [Online]. Available: https://www.cloudera.com/products/open-source/apache-hadoop.html. [Accessed 2 March 2020].

[2] N. Cravotta, "IoT unavailable impact to the business reference," 28 October 2019. [Online]. Available: https://www.embedded-computing.com/guest-blogs/redundancy-in-the-internet-of-things. [Accessed 1 March 2020].

[3] P. Selinger, M. Astrahan, D. D. Chamberlin, R. Lorie and T. Price, "Access Path Selection in a Relational Database Management System," *In Proceedings of the 1979 ACM SIGMOD international conference on Management of data (SIGMOD '79),* p. 23–34, 1979.

[4] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm and L. Zhang, "DB2 with BLU acceleration: so much more than just a column store," *PVLDB,* 6(11):1080–1091, 2013.

[5] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi and C. Bear, "The vertica analytic database: C-store 7 years later.," *PVLDB,* 5(12):1790–1801, 2012.

[6] F. Farber, N. May, W. Lehner, P. Große, I. Muller, H. Rauhe and J. Dees, "The SAP HANA database - An architecture overview.," *IEEE Data Eng. Bull.,* vol. 35, no. 1, p. 28–33, 2012.

[7] "Oracle Real Application Clusters (RAC)," [Online]. Available: https://www.oracle.com/database/technologies/rac.html. [Accessed 18 May 2020].

[8] J. M. Nick, J.-Y. Chung and N. S. Bowen, "Overview of IBM system/390 parallel sysplex-a commercial parallel processing system," in *Proceedings of International Conference on Parallel Processing*, Honolulu, HI, USA, 1996.

[9] M. Stonebraker and U. Cetintemel, "One Size Fits All: An Idea Whose Time Has Come and Gone," *In Proceedings of the 21st International Conference on Data Engineering (ICDE '05),* p. 2–11, 2005.

[10] Cboe, "Market History Monthly - 2019," [Online]. Available: https://markets.cboe.com/us/equities/market_statistics/historical_market_volume/market_history_monthly_2019.csv-dl. [Accessed 18 May 2019].

[11] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis and P. Unterbrunner, "The Snowflake Elastic Data Warehouse," *In Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16),* p. 215–226, 2016.

[12] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani and a. V. Srinivasan, "Amazon Redshift and the Case for Simpler Data Warehouses," *In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15),* p. 1917–1923, 2015.

[13] 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 Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17),* p. 1041–1052, 2017.

[14] "DB-Engines," [Online]. Available: https://db-engines.com/en/ranking_categories. [Accessed 1 March 2020].

[15] "InfluxDB," [Online]. Available: https://www.influxdata.com. [Accessed 18 May 2020].

[16] "Kdb+," [Online]. Available: https://kx.com/why-kx/. [Accessed 1 March 2020].

[17] "Queries: q-sql," [Online]. Available: https://code.kx.com/q4m3/9_Queries_q-sql/. [Accessed 1 March 2020].

[18] "Influx Query Language (InfluxQL)," [Online]. Available: https://docs.influxdata.com/influxdb/v1.7/query_language. [Accessed 1 March 2020].

[19] "Disaster-recovery planning for kdb+ tick systems," [Online]. Available: https://code.kx.com/q/wp/disaster-recovery/. [Accessed 1 March 2020].

[20] "TimescaleDB," [Online]. Available: https://www.timescale.com. [Accessed 18 May 2020].

[21] J. Correia, C. Costa and M. Y. Santos, "Challenging SQL-on-Hadoop Performance with Apache Druid," *Lecture Notes in Business Information Processing,* vol. 353, 2019.

[22] "Powered by Apache Druid," [Online]. Available: https://druid.apache.org/druid-powered. [Accessed 1 March 2020].

[23] R. Shiftehfar, "Uber's Big Data Platform: 100+ Petabytes with Minute Latency," 17 10 2018. [Online]. Available: https://eng.uber.com/uber-big-data-platform/. [Accessed 1 3 2020].

[24] S. Krishnan, "Genie is out of the bottle!," 21 June 2013. [Online]. Available: https://netflixtechblog.com/genie-is-out-of-the-bottle-66b01784752a. [Accessed 1 March 2020].

[25] K. Weil, "Hadoop at Twitter," 8 4 2010. [Online]. Available: https://blog.twitter.com/engineering/en_us/a/2010/hadoop-at-twitter.html. [Accessed 1 March 2020].

[26] "Lambda Architecture," [Online]. Available: http://lambda-architecture.net/. [Accessed 1 March 2020].

[27] X. Léauté, "Benchmarking Druid," Druid, 17 03 2014. [Online]. Available: https://druid.apache.org/blog/2014/03/17/benchmarking-druid.html. [Accessed 14 07 2020].

[28] B. Chattopadhyay, P. Dutta, W. Liu, O. Tinn, A. Mccormick, A. Mokashi, P. Harvey, H. Gonzalez, D. Lomax, S. Mittal, R. Ebenstein, H. Lee, X. Zhao, T. Xu, L. Perez, F. Shahmohammadi, T. Bui, N. McKay, N. Mikhaylin, S. Aya and V. Lychagina, "Procella: Unifying serving and analytical data at YouTube," *PVLDB,* 12(12):2022-2034, 2019.

[29] "Amazon Timestream," [Online]. Available: https://aws.amazon.com/timestream/. [Accessed 1 March 2020].

[30] "Amazon S3 pricing," [Online]. Available: https://aws.amazon.com/s3/pricing/. [Accessed 1 March 2020].

[31] IBM, "IBM Db2 Event Store product page," [Online]. Available: https://www.ibm.com/products/db2-event-store. [Accessed 2 March 2020].

[32] IBM, "IBM Cloud Object Storage," IBM, [Online]. Available: https://www.ibm.com/products/cloud-object-storage-system. [Accessed 14 07 2020].

[33] IBM, "IBM Cloud Block Storage," IBM, [Online]. Available: https://www.ibm.com/cloud/block-storage. [Accessed 14 07 2020].

[34] The Ceph Foundation, "Ceph," [Online]. Available: https://ceph.io/. [Accessed 14 07 2020].

[35] IBM, "IBM Spectrum Scale," IBM, [Online]. Available: https://www.ibm.com/products/scale-out-file-and-object-storage. [Accessed 14 07 2020].

[36] "Apache ZooKeeper," [Online]. Available: https://zookeeper.apache.org/. [Accessed 1 March 2020].

[37] Amazon, "Amazon Redshift FAQs," Amazon, [Online]. Available: https://www.amazonaws.cn/en/redshift/faqs/. [Accessed 14 07 2020].

[38] "Apache Parquet," [Online]. Available: https://parquet.apache.org/. [Accessed 1 March 2020].

[39] C. Luo, P. Tözün, Y. Tian, R. Barber, V. Raman and R. & Sidle, "Umzi: Unified Multi-Zone Indexing for Large-Scale HTAP," in *EDBT '19*, 2019.

[40] "Parquet Files," [Online]. Available: https://spark.apache.org/docs/latest/sql-data-sources-parquet.html. [Accessed 1 March 2020].

[41] "Snappy, a fast compressor/decompressor," [Online]. Available: https://github.com/google/snappy. [Accessed 1 March 2020].

[42] "GNU Gzip," [Online]. Available: https://www.gnu.org/software/gzip/. [Accessed 1 March 2020].

[43] "LZ4 - Extremely fast compression," [Online]. Available: https://github.com/lz4/lz4. [Accessed 1 March 2020].

[44] "Apache Parquet for C++: a C++ library to read and write the Apache Parquet," [Online]. Available: https://github.com/apache/parquet-cpp. [Accessed 1 March 2020].

[45] "Parquet modular encryption," [Online]. Available: https://issues.apache.org/jira/browse/PARQUET-1300. [Accessed 1 March 2020].

[46] "Implement encrypted Parquet read and write support," [Online]. Available: https://github.com/apache/arrow/pull/2555. [Accessed 2 March 2020].

[47] J. Salowey, A. Choudhury and D. McGrew, "AES Galois Counter Mode (GCM) Cipher Suites for TLS," August 2008. [Online]. Available: https://tools.ietf.org/html/rfc5288. [Accessed 1 March 2020].

[48] "Db2 native encryption," [Online]. Available: https://www.ibm.com/support/knowledgecenter/SSEPGG_11.5.0/com.ibm.db2.luw.admin.sec.doc/doc/c0061758.html. [Accessed 1 March 2020].

[49] IBM, "Data Skipping for IBM Cloud SQL Query," IBM, [Online]. Available: Query https://www.ibm.com/cloud/blog/data-skipping-for-ibm-cloud-sql-query. [Accessed 14 07 2020].

[50] IBM, "IBM Cloud Object Storage Pricing," [Online]. Available: https://www.ibm.com/cloud/object-storage/pricing. [Accessed 14 07 2020].

[51] E. O'Neil J., P. E. O'Neil and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," *In Proceedings of the 1993 ACM SIGMOD international conference on Management of data (SIGMOD '93),* p. 297–306, 1993.

[52] N. Megiddo and D. S. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," in *In Proceedings of the 2nd Usenix Conference on File and Storage Technologies (FAST '03)*, San Francisco, CA, USA, 2003.

[53] P. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie and T. G. Price, "Access Path Selection in a Relational Database Management System," *In Proceedings of the 1979 ACM SIGMOD international conference on Management of data (SIGMOD '79),* p. 23–34, 1979.

[54] "Time Series Benchmark Suite (TSBS) github page," [Online]. Available: https://github.com/timescale/tsbs.

[55] "Time Series Benchmark Suite (TSBS)," 28 Aug 2019. [Online]. Available: https://blog.timescale.com/blog/how-to-benchmark-iot-time-series-workloads-in-a-production-environment/.

[56] Apache Software Foundation, "Apache Cassandra," [Online]. Available: http://cassandra.apache.org/. [Accessed 2 March 2020].

[57] "TimescaleDB vs. InfluxDB: Purpose built differently for time-series data," 15 June 2019. [Online]. Available: https://blog.timescale.com/blog/timescaledb-vs-influxdb-for-time-series-data-timescale-influx-sql-nosql-36489299877/. [Accessed 18 May 2020].

[58] "TPC-H," [Online]. Available: http://www.tpc.org/tpch/. [Accessed 1 March 2020].

[59] A. Druid, "Apache Druid TopN operator," [Online]. Available: https://druid.apache.org/docs/latest/querying/topnquery.html . [Accessed 21 05 2020].

[60] "Druid Joins," [Online]. Available: https://druid.apache.org/docs/latest/querying/joins.html. [Accessed 1 March 2020].