dp3: multimodal log database

Wyatt Alt wyatt.alt@gmail.com Jainil Ajmera jainil@berkeley.edu

May 8, 2024

Abstract

dp3 is a multimodal log database for robotics data infrastructure, with goals to support low-latency playback, timeseries analytics, precomputed multigranular field-level statistical summaries, and efficient integration with distributed compute under a single system.

The first section of this paper will introduce key concepts, background, and challenges in the multimodal logging domain. The second section will present the architecture of dp3.

1 Introduction

dp3 [30] is a row-oriented multimodal log database structured around a time-partitioned copy-on-write tree, which is stored in S3-compatible storage and fronted by an in-memory cache of inner tree nodes. It is designed for storage and retrieval of robotics data with three "user personas" in mind:

The **robotics developer**, whose primary workload consists of alternating between searches for interesting log events and doing replay-based analysis of those events using a playback visualization tool, or possibly replaying logs through their software stack.

The **ML developer**, whose primary workload consists of large-scale compute jobs against narrow selections of topics.

The **data infrastructure admin**, who is responsible for administering the database and satisfying the other two customers in a cost-effective and maintainable manner.

One person may of course assume multiple personas.

For the robotics developer, dp3 provides low-latency playback for data recorded by a "producer" from a point in time on a selection of "topics", as well as a query language for ergonomic expression of the "as-of join" conditions allowing the user to quickly pivot between searching for interesting events, and replaying those events in full context.

For the ML developer, dp3 supports both client/server and embedded modes of operation. The query executor may be embedded directly into jobs so that jobs can operate directly against S3. This isolates heavy and bursty ML-related read workloads from the comparatively lighter-weight streaming playback and summarization workloads supported by the database server, and reduces the cost and complexity of infrastructure management by eliminating the need to scale a database deployment in step with the job workloads.

dp3 minimizes operational burden for the data infrastructure admin by offloading primary storage and index structure to S3 and allowing the admin to use S3 tooling (lifecycle policies, retention policies, etc.) to govern the garbage collection of the tree. To keep users from being exposed to deleted objects, the admin must ensure the tree storage is always truncated ahead of the physical deletion schedule. Since dp3 supports any S3-compatible storage, it can be deployed in on-premise environments in situations where cloud offload bandwidth is limited at the point of ingestion.

dp3 is initially targeting a smooth integration experience with the ROS [23] ecosystem and the MCAP [10] format, to become a viable "plug and play" infrastructure primitive for ROS and MCAP users. To facilitate this, dp3 accepts MCAP files as its input format, stores data in MCAP format in the leaves of the tree, and streams MCAP out during playback. The messages recorded by the user are repackaged and compressed, but not otherwise transformed from their original encoding, making dp3's output compatible with any internal log tooling the user has developed. Once dp3's supported set of message encodings is expanded, it should be fully usable as a storage component by any ROS user.

The main contributions of dp3 are: (1) A single system for playback, search, and analytics of multimodal logs. While these purposes are individually satisfiable by other solutions, dp3 consolidates them under one platform. (2) A database system leveraging the MCAP file format for data storage. To our knowledge dp3 is the

only such system. (3) A stream-oriented query language for expression of merges and as-of joins. (4) A database that directly targets compatibility with existing ROS infrastructure.

The first section of this paper will elaborate on some of the design goals and high-level context surrounding the project. The second section will describe the implementation of key components in greater detail.

2 Background

2.1 Multimodal log management

In robotics, it is common to use the word "logging" to refer to the process of writing multimodal diagnostic messages to network or disk for followup analysis. Perhaps by intent, the term serves to emphasize the parallels with traditional server application logging, i.e applications log text, robots log point clouds.

Server application log management platforms are plentiful and follow well-established patterns. Some examples of these include Sumo Logic [29], Datadog logging [8], and the log management interfaces of the three major cloud providers: Google Cloud Logging [14], AWS Cloudwatch Logs [1], and Azure Monitor Logs [21].

From the user's point of view, these log management solutions are mostly interchangeable. The core interface consists of, (1) Search controls: dropdowns for error levels, date picking, application names, tags, and a text box for "advanced" searches in a limited query language; (2) A listing of log lines matching the search results in descending timestamp order; (3) A "view in context" feature, usually accessed by right-clicking a particular log line in the listing. This feature will show the log line of interest highlighted (or "pinned") in a wider scope, perhaps by dropping the search restriction. For instance if the original query was restricted to a single application, the "context" may now include all applications. This feature is used for tracing cascading failures and events across services.

Users of multimodal logs have a similar set of requirements, with additional challenges.

First, there are more kinds of content-based search, such as geographical search, image similarity search, or other kinds of search unanticipated by the database designer (i.e UDFs), and perhaps a more frequent need to correlate multiple signals in a single search. The need for timestamp-based search controls and filtering on high-level dimensions remains as great as before.

Second, the meaning of "listing log lines" or "filling a screen with logs" becomes a lot less clear and is an interesting question in user interface design. Log lines can still be listed as JSON, with bytes-valued fields rendered in base64 or similar – but more creative approaches can likely do better by incorporating images, pointcloud visualizations, maps, audio, and more.

Third, text rendering is clearly insufficient for the "view in context" function, which now represents replaying "what the robot saw" at the associated time. Instead of a pinned message in a larger resultset, "view in context" is implemented using a complex visualization application such as Foxglove [11].

Beyond these parallels, there are other points of divergence:

Different organizational profile. Server application logs are generally browsed only by infrastructure teams and application administrators. The read activity on those databases is sparse and high latencies are tolerable. In a robotics development organization, a multimodal log database is a business-critical, user-facing service and may be depended on by most of the development team outside of infrastructure. High latencies in playback and search are a direct impediment to product development speed.

Different scales of volume. Depending on what kind of data the user is logging, multimodal data volumes can be orders of magnitude larger than server application log volumes for the same organization. Large producers may produce hundreds of terabytes or petabytes of logs per day. This makes it relatively more important for administrators to have a clear understanding of data access patterns, and granular control over data retention.

Support for distributed compute. Typical server log databases do not expect you to run distributed compute jobs on your log data and are not designed with that use-case in mind, whereas in multimodal log management, it is a core workload.

Support for statistical aggregates. In many cases, multimodal logs are similar to metrics. It can be attractive to be able to plot statistical aggregates such as quantiles or averages of field values over selected windows of time. The APM portions of Datadog enable this kind of operation at low latency even over trillions of points. To drive a responsive interface these aggregations must rely on precomputation.

While there are undoubtedly large organizations where these concerns apply to application log

management, in multimodal logging they become relevant at much smaller levels of company scale, when infrastructure teams are less developed.

2.2 ROS and MCAP

dp3 integrates closely with two existing software projects: the ROS (Robot Operating System) ecosystem, and the MCAP log file format.

ROS is a software framework frequently used in academia and industry for developing robotics applications. A ROS-based system is built from "nodes", which are independent processes that communicate via "topics" in pub/sub fashion over a message bus. Nodes may be attached to sensor drivers or process the outputs of other nodes. ROS provides various pre-built nodes for common kinds of robotics operations in Python and C++, along with extension support for writing your own nodes.

The ROS project has recently completed a transition from ROS1 to ROS2 [19]. One important aspect of this transition was a switch from a prescribed message serialization format (ros1msg) to a pluggable recording architecture where the user is free to choose whatever serialization they want. For example CDR [16], protobuf [15], and flatbuffers [13] are common choices. Flexible message serialization is a benefit for users but a complication for tooling authors, who must support whatever format the user chooses instead of focusing on just one.

MCAP is a container format for multimodal logging developed at Foxglove, intended to support the pluggable serialization goals of ROS2 while still standardizing format features such as time-based indexing, chunk compression, integrity checks, and efficient remote summarization. The goal of MCAP is to enable a shared ecosystem of multimodal log tooling in spite of the variation among organizations in message serialization choice. MCAP has been the default log format of ROS2 since version "Iron Irwini", released in May 2023 [25].

dp3 targets integration with MCAP because the ROS ecosystem, and by extension the MCAP ecosystem, is significantly populated by educational applications and early-stage startups in position to try new approaches to data management. Accordingly, dp3 is built around MCAP's domain model:

A **schema** is a named schema definition in a format consistent with the encoding format of the log's messages.

A **channel** is a logical stream of messages within a log file. A channel is associated with a schema, as well as a topic.

A **topic** is a name for a logical data stream, for instance "/images". Each channel has exactly one topic, multiple channels may have the same topic, and channels with the same topic may have different schemas (unusual in a single recording but common over time as an organization's schemas evolve).

A **message** associates a channel with an array of bytes, which encodes a message in the format described by the schema of the channel.

2.3 As-of joins

A central aspect of multimodal log search workloads is a similar operation to what some SQL dialects call an "as-of join". The idea of an as-of join is to join records in one table with those in another based on a sequential or temporal relationship - for instance, join records in one table with the immediately preceding or succeeding records in another table, based on a timestamp column.

As a result of the message-oriented architecture described in the previous section, these kinds of queries are common in robotics. For an example query, consider working on a self-driving car and needing to locate recent instances where the car was taking a left hand turn in the rain, with more than 10 pedestrians present in an intersection, in order to debug some misbehavior that has been noticed. The log messages that would indicate the left turn, the rain, and the number of tracked pedestrians will likely occur on separate topics, at unaligned timestamps and different logging frequencies. This makes the desired result the cases where variously-defined "adjacent" readings on these topics match your search criteria.

As-of joins are a nonstandard SQL feature, and are not supported in all analytics databases. In dialects where they are unsupported they are generally emulatable with more complex SQL syntax. None of BigQuery [12], Redshift [26], or Azure Synapse [22], support them. DuckDB [24], Clickhouse [6], and Snowflake [28] all do.

Regardless of whether an engine has explicit support for as-of joins, there are complexities with putting SQL in front of the user and telling them to write complex N-way as-of joins. First, while the robotics developer may have some basic SQL experience, they are not a SQL expert, and particularly if the dialect does not support explicit as-of joins the queries can become extremely complex. Although one workable option is to hire SQL experts to interface between the database and the robotics developer, one of dp3's experimental goals is to avoid this solution and attempt instead to empower the

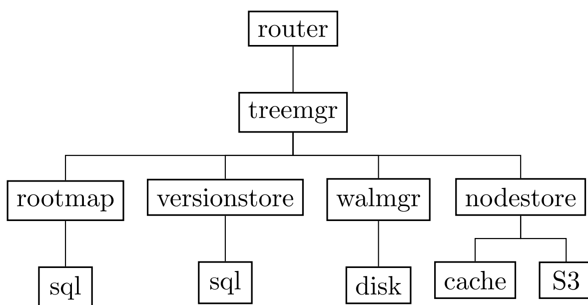

Figure 1: Component structure of dp3

robotics developer directly. Second, analytics databases are not general-purpose databases and often have non-obvious performance footguns and tricks that you need to understand to really use them effectively. A typical user with basic SQL experience has usually gained that experience from a general purpose RDBMS, and may be aggravated to find that the SQL implementation of an analytics DBMS comes with a host of unwritten guidance along the lines of "don't use joins" or "manually push down where clauses", or that join algorithms must be directly specified. SQL knowledge is portable to a degree, but perhaps less than might initially appear.

In dp3 we target a narrower language feature set with explicit focus on ergonomic as-of join support. While we compile these queries to our own execution logic, the binding of the language to dp3 is not very tight, and there would be no conceptual barrier to instead compiling to e.g Clickhouse SQL in a way that transparently handles the quirks of Clickhouse for the user and results in a statement that should execute efficiently. This could be a useful extension strategy for a platform built on dp3.

3 Implementation

3.1 Program architecture

The component structure of dp3 is depicted in Figure 1. This section will briefly outline the main data flows, and the following sections will explain components in greater detail.

3.1.1 Write path

On the write path, the router receives an MCAP data stream and passes it into the tree manager, which splits it into per-topic MCAP-format buffers in memory, while parsing the messages and computing statistics. Buffers are flushed into the WAL as a configurable size threshold or on full consumption of the input.

The WAL manager monitors data outstanding in WAL for both size and time since last update. When the outstanding data of a stream exceeds a size threshold, or when the stream has not received a new write within a configurable amount of time, leaf pages from the WAL are merged into a copy-on-write tree overlay, and written into storage as a single object. This helps to decouple storage write dimensions from the user's recording pattern: if the user is recording a lot of small files, we will batch data from multiple input files into larger storage writes.

Following a successful write to storage, the location of the root, along with a version from the version store, and the server's timestamp, are written to the rootmap database.

dp3 does not support updates, and writes to storage must be serialized per-tree.

3.1.2 Read path

On the read path, the router receives a request as a string in the dp3 query language. The query is parsed and transformed into an iterator-style execution plan, with a root location retrieved from the tree manager for each topic involved in the query. Scan nodes are instantiated with a timerange and a filter expression and are executed by walking the relevant tree leaves and applying the filter. Leaf nodes are structured as linked lists and sometimes have multiple "insert" and "delete range" operations, which the scan operator resolves into a set of file read and merge operations.

Node lookups go through the nodestore, with a read-through byte-capacity bounded LRU cache of inner nodes. If the important inner nodes of the tree are in the cache, then the only reads from storage are those for the leaf data.

After streaming through the tree of query operators, records are written out to the user in a single topic-multiplexed MCAP data stream.

3.2 Public API

The public API of dp3 consists of the following core operations:

Import(database string, producer string, objectID string) imports an object from storage on behalf of a producer by telling dp3 a location from which to read. The term "producer" is intentionally vague: it is up to the user to decide whether to associate producers with "devices" or "simulation runs" or another internal concept. dp3 supports multiple logical databases within a single instance, and operations must be scoped with a database.

Query(database string, query string) evaluates the query and returns results in MCAP format.

StatRange(database string, producer string, topic string, start uint64, end uint64, granularity int) returns a range of field-level statistical aggregations at a granularity at least as granular as the one requested. These statistics are stored in the inner nodes of the tree, so under optimal circumstances StatRange queries should mostly be served from RAM.

Delete(database string, producer string, topic string, start uint64, end uint64) performs a logical deletion of data between two timestamps. Physically this is an insert of a "mask" rather than a true delete. Physical deletion is handled by storage retention policies only.

Truncate(database string, producer string, topic string, timestamp uint64) perform a logical data truncation at a particular timestamp. This makes data inserted prior to that timestamp unreachable by read requests. To prevent reads from accessing deleted files, it is necessary to truncate tables prior to the time when storage retention policies would delete their objects.

Messages(database string, producer string, topics map[string]uint64), start uint64, end uint64 returns a merged result of messages on the provided set of topics in timestamp order. The values of the passed map are minimum tree versions, and the HTTP headers in the response of the method indicate the versions of the topics in the response. This API is used to emulate "log tailing", by polling for messages since the last tree version seen. We will probably remove this once we decide how to incorporate the tailing functionality into the query executor.

3.3 Multigranular tree

The core storage structure of dp3 is a versioned, time-partitioned, copy-on-write tree. One tree is stored for each producer/topic combination. Internally we refer to this combination as a "table", however the user deals only with producers and topics. Trees are constructed based on a start and end time, branching factor, and target leaf width in seconds. The constructor then picks a height that results in a tree satisfying the dimensional constraints. The height obtained by a default tree is 5.

The nodes of the tree are divided between inner and leaf nodes. All data is stored in the leaf nodes in MCAP format, with the inner nodes responsible for storing time bounds along with an array of children, each associated with statistics when present. Within dp3 and in the child arrays of inner nodes, nodes are addressed with a 24-byte ID, the structure of which is depicted in Figure 2.

bytes	value	description
8	uint64	version
8	uint64	offset
8	uint64	length

Figure 2: Structure of a node ID

The tree module implements three core write-related methods: Insert, DeleteRange, and Merge. Insert and DeleteRange are both physical inserts. They each construct a path from a root to a leaf and serialize that path to the WAL. When the WAL manager chooses to sync the WAL data to storage, the Merge method is called on these partial trees, along with the relevant inner node structure of the destination tree in storage, to form a copy-on-write overlay.

3.3.1 Truncation

We support efficient, synchronous, logical deletion from the tail of the tree of data before a timestamp. This is implemented by looking up the tree version at that timestamp in the rootmap, and setting the table's minimum version to that version plus one. Subsequent scan operations are made aware of the minimum version and ignore the data.

Truncating the table ensures that no reader will access deleted objects after storage lifecycle policies physically reap the files. Truncation must be scheduled by the administrator to stay ahead of the lifecycle policy-initiated deletion.

3.3.2 Range-based deletion and overlapping inserts

In addition to truncation from the tail of the tree, we support the ability to delete messages from an arbitrary time range. Range-based deletion is accomplished with a combination of two strategies. Any tree node impacted in a deletion scenario is either fully or partly covered. If a node is fully covered by the deletion, the partial tree created by the DeleteRange function will omit the node and include a tombstone indicator in the corresponding location in its parent's child array. This rule can be applied down the tree until we are left with the case of partially-split leaf nodes to resolve.

We implement leaf nodes using a linked list structure. The node has a header prior to the

bytes	value	description
1	uint8	physical node version
24	node ID	ancestor node ID
8	uint64	ancestor version
8	uint64	ancestor delete start
8	uint64	ancestor delete end
n	bytes	mcap leaf data

Figure 3: Byte serialization of a leaf node

start of the data that includes an ancestor node ID and version, as well as an ancestor delete and end time.

If the bytes of the ancestor node ID are all zero, there is no ancestor. Otherwise, the header indicates that readers must merge the ancestor node's data with the data of the node in hand. If the deletion end time is nonzero, that range is interpreted by the reader as a mask to apply over the merge of ancestor data.

Overlapping inserts use the same linked list structure and simply leave the deletion range zero. If we receive a write for a leaf that already has data on it, we handle that as a physical list append rather than a physical merge of message data.

Consequently our tree merge operation never needs to read message data from storage, only inner nodes. However, we are subject to a form of "bloat" from heavy overwrites until we implement some kind of compaction process.

3.3.3 Storage object structure

Our writes to both the WAL and storage are based around an internal structure called a "memtree". The memtree is a linked structure of node pointers amenable to in-place manipulation. The memtree's byte serialization is the structure of both our storage objects and the "data" portion of the WAL insert records.

In the memtree, nodes are associated with random, temporary IDs. To serialize the memtree to disk, we walk it breadth-first and collect a path of nodes. We reverse the path and serialize the nodes to an output buffer in reverse-dependency order, so that the offset and length used in construction of the real node IDs are known at the time when they need to be recorded into the child arrays of their parents.

The final node serialized to the output is the root node. After the root node is serialized, we write its ID to the final 24 bytes of the output, enabling a standalone reader to interpret the file by reading the last 24 bytes and seeking to the root offset from which to start traversing.

The byte serialization of a leaf node is depicted in Figure 3.

value	description
uint64	start time
uint64	end time
uint8	height
[]child	children

Figure 4: Structure of an inner node

value	description
node ID	child node ID
uint64	child version
map[string]statistics	per-schema statistics

Figure 5: Structure of an inner node child array element

The ancestor-related fields may be zero depending on whether there is an ancestor and whether the leaf represents a logical insert or delete.

Our inner nodes are currently serialized as a uint8 physical node version followed by the JSON serialization of the inner node structure, which is depicted in Figure 4.

The structure of an inner node child is depicted in Figure 5.

We anticipate switching the inner nodes from JSON to a packed format before long.

The statistics stored on inner node children are stored per-schema, with schemas identified using a sha1 content hash. Storing statistics for multiple schemas is necessary because the schemas of our tables can evolve over time, including in ways that typical notions of schema evolution would call incompatible, meaning a field could (as far as we are concerned) have different meanings or types at two different points in time. In order to resolve queries and provide useful statistics in this situation, we store data on each schema separately and leave it to the consumer to decide what to do.

The statistics themselves are computed for all fields except variable-length arrays, and different statistics are supported depending on whether the field is numeric or text. The major constraint on the statistics we store is that they are associative: it must be possible to derive a correct updated statistic from an existing statistic and the new data being inserted, without rescanning old data. This property holds for some common statistics such as sum, min, max, count, and average (assuming count is also maintained), but does not hold for others such as exact quantiles. There are approximate alternatives that may be useful for addressing this gap, such as the DDSketch [20].

The text statistics we store today are very

bytes	value	description
1	uint8	record type
8	uint64	data length (**n**)
n	bytes	record data
4	uint32	crc32 of preceding fields

Figure 6: WAL record serialization

limited: only min and max are supported. However, some useful-seeming structures obey the associative law and could be used for text, such as Bloom filters, unique sets, and trigram bitmaps.

The first byte of both the leaf and inner node serializations is the physical node version. This is distinct from the "version" referred to elsewhere in the system, such as in the "ancestor version" field of the leaf node. The physical node version serves both to evolution of the physical node format and indicate to readers whether they are dealing with a leaf or inner node. Leaf and inner nodes split the range of the unsigned byte, enabling 128 possible versions for each.

Storage objects are named with the decimal representation of the tree version they correspond to. WAL manager The write ahead log is a single, append-only stream of binary records. The format of a WAL record is depicted in Figure 6.

As with storage objects, WAL filenames are decimal integers, and WAL records have 24-byte addresses similar to node IDs, which can be resolved to a WAL filename, offset, and length.

They byte identifier of the WAL record takes one of three types, indicating how the data should be parsed: **Insert** contains a serialized partial tree covering one leaf node. **MergeRequest** contains a list of insert record addresses that should be merged as a batch into storage, as well as a UUID-format batch ID. **MergeComplete** records the completion of a merge operation into storage by batch ID.

On startup, the WAL manager scans all outstanding WAL files in the WAL directory, replaying their records through its accounting until it arrives at a final state representing unmerged work in the WAL. Thereafter, writes to the WAL are accounted for in memory and this state is used for scheduling of flushes to storage.

Garbage collection of the WAL is handled with a process that polls the in-memory state of the WAL manager to determine if any files on disk are no longer referenced. Any such files are deleted.

3.4 Rootmap

The rootmap fronts a database of root locations used by the tree manager. Every write that is finalized to storage records a write to the rootmap, and every read from dp3 (when operating in server mode) does a read from the rootmap. While the rootmap is technically rebuildable from storage operations by listing objects and reading root IDs from the final 24 bytes, it is essential for the functioning of the database and effectively a central point of failure.

The rootmap interface is fairly straightforward and should be portable to a range of database solutions:

Get(database string, producer string, topic string, version uint64) gets a root for in a database a table at a version.

GetLatestByTopic(database string, producer string, topics map[string]uint64) gets the latest roots in a database for a set of tables. The passed map of topics indicates minimum tree versions for each topic.

GetHistorical(database string, producer string, topic string) gets all root versions in the database for a single table.

Put(database string, producer string, topic string, version uint64, nodeID nodeID) inserts a new root into the rootmap.

One dp3 instance can contain multiple logical "databases", allowing users to segregate simulation data from real-world data or maintain other such divisions. All operations against the rootmap are therefore scoped with a logical database identifier.

Today, dp3 uses SQLite [17] for the rootmap.

3.5 Versionstore

For a single tree, writes must be versioned in monotonically increasing order to support range deletion as well as querying for new data since a prior read.

The version store is backed by a persistent counter, from which a fixed number of versions are reserved at a time. On startup, a dp3 instance acquires a lock on the counter and increments it, thus reserving a set of versions before another lock must be taken. When the end of the reserved range is reached, a new reservation is made. This batching minimizes contention over the shared counter.

This mechanism exists to ensure that within a tree, a single node will only record increasing versions for a table. When write clustering support for dp3 is developed, we will still maintain an invariant that writes for a single table flow through only one node at a time. If responsibility for a table transitions from one node to another (i.e during resharding), a new

reservation operation or service restart will be required.

The versionstore is backed by SQLite today, and is colocated in the same database as the rootmap. The only required feature is the ability to lock and set the shared counter, which is supported by any mainstream SQL database, and activity on it is very sparse. To reduce component sprawl, the versionstore and rootmap will likely remain colocated as long as the rootmap is also on SQL.

3.6 Query language

The kinds of searches executed on dp3 are expected to be heavy on multiway as-of joins. In natural language, a typical kind of search might be, "show me the times in the last month where we were taking an unprotected left hand turn, in the rain, with dogs and bicycles in the intersection."

Each of these criteria (unprotected left, is it raining, what objects are tracked) will be stored on different tables, and the timestamps of each table will not be in alignment, as the messages will be logged at different frequencies. This means the utility of typical equijoins is diminished, and query conditions based on relative time proximity or immediate precedence are more useful – in this case, locating instances where we see dogs and bicycles and the prior instance of /raining reported "true", and the prior instance of /unprotected_left indicated the beginning of a turn. These examples are of course simplified.

In dp3 we have developed a query language for ergonomic expression of as-of joins. The dp3 query language includes the following features: (1) Where clauses with support for nested fields and typical SQL binary operators. (2) Parenthetical groupings. (3) Time-ordered merging of tables. (4) As-of joins, based on either a time window, immediate succession, or both. (5) Reversal of scan order. (6) Time-based restriction on log time. (7) Limit and offset.

Currently, all queries must be scoped with a single device. All queries are terminated with a semicolon. Some examples of valid queries are listed below.

Scan all messages on a topic.

```
from my-robot /topic;
```

Scan all messages on a topic with a where clause.

```
from my-robot /topic as t
where t.my_field > 10;
```

Return a time-ordered merge of two topics.

```
from my-robot /topic1, topic2;
```

Restrict from clause with a time range.

```
from my-robot between
"2020-01-01" and "2021-01-01"
/topic;
```

Return merged results with conditions on both tables.

```
from my-robot
/topic1 as t1, /topic2 as t2
where t1.field = "foo"
or t2.field = "bar";
```

As-of join with where clause with condition on either child.

```
from my-robot
/topic1 as t1 precedes /topic2 as t2
by less than 5 seconds
where t1.field = "foo"
or t2.field = "bar";
```

As-of join with immediate modifier.

```
from my-robot
/topic1 as t1
precedes immediate /topic2 as t2
where t1.field = "foo"
or t2.field = "bar";
```

Reversal of scan order.

```
from my-robot /topic desc;
```

These queries get compiled into a typical iterator-style query execution plan, and executed by repeatedly calling Next on a root executor node.

The current operators we implement are "merge", as-of join, scan, filter, limit, and offset. The merge operation is associated with the comma operator in the language grammar, and it executes an N-way streaming merge of time-ordered children through a binary heap. This is the same way a conventional RDBMS may implement an ordered union, when indexes exist on the ordering column.

The as-of join operations (precedes or succeeds keyword) are executed by first inverting "succeeds" terms to "precedes", and then feeding the two sides into an operator that caches the previously-observed LHS value, and evaluates on each RHS observation whether the previously-observed LHS and current RHS satisfy the as-of condition. A specification of "immediate" causes only the first RHS value to be returned for each LHS. Without the immediate modifier, all matches will be returned.

4 Discussion

4.1 Related Work

The copy-on-write tree structure of dp3 borrows heavily from btrdb [2], a timeseries database for storing streams of high-frequency univariate floats. Comparing the two, btrdb's use of univariate floats makes it effectively columnar, and enables it to use compression techniques that are not available to dp3, which stores multivariate rows with chunked zstd compression.

Another system that uses a similar storage scheme is Apache Iceberg [9]. Iceberg is a table format specification rather than an implementation, but the format specifies a height-three copy-on-write tree consisting of metadata files, manifest lists, manifest files, and data files. The metadata is a JSON file storing a list of "snapshots" that reference manifest lists. Manifest lists include summary information about manifest files, and manifest files contain a list of data files in Avro format. Iceberg has built-in support for Parquet, Avro, and ORC format data files. As with dp3 and btrdb, it seems logical in deployment scenarios to cache the upper nodes of the tree.

The primary intent of Iceberg is to enable multiple database engines to operate against the same data safely at the same time, but as a useful side-effect of having data in Iceberg format you gain the ability to use an off-the-shelf database engine for free, and the potential for pluggable integrations in the future if more databases add support for external Iceberg tables, as Snowflake has recently done. In addition, dp3 may at some point want to have multiple engines operating at the same time, for instance if we were able to integrate with Spark SQL [3]. These possibilities make Iceberg a worthwhile direction to explore as backing storage for dp3. In comparing the two, Iceberg stands out for its far more developed SQL and transactional capabilities, while dp3 is distinguished in supporting message storage and playback in the user's native choice of recording format, without on-the-fly transcoding during read. dp3 also has a more granular time index due to having a deeper tree.

The query language and goals of dp3 bear similarity to query languages and systems previously developed in the areas of "stream databases" and "sequence databases", such as CQL [4], the GSQL language for the Gigascope database [7], Aurora [5], and SEQUIN for the SEQ system [27]. These languages are broadly SQL-like with a focus on support for merging and as-of join operations, although those go by different names.

CQL has a focus on continuous queries over unbounded data streams. CQL supports a "sliding window join" with the syntax

```
Select Istream(*)
From S1 [Range 5 seconds], S2 [Range 10
    seconds]
Where S1.A = S2.A
```

in this example, a tuple from S1 will be sent to the output if it joins with a tuple in S2 from within a 10-second sliding window, and a tuple from S2 will be sent if it joins with a 5-second sliding window from S1.

dp3 does not support this kind of bidirectional windowing today, but this query is pretty similar to what dp3 would express as

```
from my-device
S1 precedes S2 by less than 10 seconds
where S1.field = "foo"
and S2.field = "bar"
```

The concept of joining columns between tables in dp3 is not presumed to make sense today, because the fields of different tables are typically incomparable. However, support for this may become important as we develop more focus on subqueries.

Continuous queries are also not supported in dp3 today, however support would not require much work. The tree structure makes it very cheap to check if there is any new data since the last time observed. This should make it simple to offer a "tail" operator as an alternative to scan, which will run continuously and watch the tree root for new tuples as they come in. The rest of the query plan should remain the same.

In the SEQUIN language for the SEQ database, an as-of join is expressed in this way:

```
// first define the moving average as a
    view
CREATE VIEW MovAvgStockl AS (
PROJECT avg(C.high) as avghigh
FROM stock1 c
OVER \$P-23 TO \$p.);

// then use the view in the query
PROJECT A.avghigh - B.high
FROM MovAvgStockl A,
Previous(PROJECT D.high
FROM Stock2 D
WHERE D.volume > 25,000) B
WHERE.\$P > 2000;
```

The *Previous* function returns the most recent hour at a point during evaluation when the volume on Stock2 exceeded 25,000. In dp3 we have no support for aggregations at this time, so we are unable to compute the moving average represented

in the view. In dp3, "Previous" is implemented with "precedes immediate".

4.2 Work remaining

dp3 is on github [30] in partially implemented form. A license is not yet chosen. The database is implemented in Go. dp3 compiles to a single binary used for both the server and the command-line interface.

That interface includes various functions for introspecting the storage, such as the "treeinspect" and "walinspect" commands, as well as an interactive terminal similar to psql (from Postgres). From the client interface it is possible to list tables, view statistics, and run interactive queries.

Communication between client and server is over an HTTP API mapping closely to the functions described in 3.2. We view this API as fairly ad-hoc and temporary, and intend to either firm it up or switch to grpc once we are in a better position to evaluate.

Significant work remains, both in design validation and feature development, to get dp3 to a useful state.

4.2.1 Design validation

The largest outstanding design question relates to dp3's choice to use MCAP as the data format in leaf storage. There are a few benefits to this choice: (1) No transcoding required on ingestion or query. (2) Automatically compatible with user's own internal tooling. (3) Streaming playback is a row-oriented workload. (4) Automatically compatible with common robotics recording workflows

There are also drawbacks, primarily in the performance department. (1) Columnar storage would realize better compression in general. (2) A standard columnar format may allow us to leverage an off-the-shelf query executor, and could probably support more performant queries for our search workload. (3) Building generic systems on MCAP requires supporting query operations on multiple different binary encoding formats in place, which is tedious to implement and support, and may carry some performance risks.

The question is also complicated by workload-specific factors. If a columnar format is used, it is likely that small "row groups" (or whatever equivalent concept exists) will be required, to enable queries to merge large numbers of tables at once without exceeding available memory. Using small row groups will tend to work against the interests of good OLAP performance on the data files, obviating some of the benefits we would be hoping to get with the columnar format.

Furthermore, a significant portion of the heavy read volume the database will serve for ML purposes, will be image and point-cloud data. In these kinds of messages, there is effectively one column: a byte array, possibly up to megabytes in size. This means that row-oriented and columnar files for these kinds of data are physically pretty similar, so the impact of the change to columnar for these cases may not be that great.

The best way to resolve all these questions seems like an implementation spike. Parquet is an obvious candidate for a columnar format to test as it would also be compatible with Iceberg. The LanceDB [18] project is another one that is focused on some of the more exotic kinds of searches users might be interested in, such as image similarity search. Once we have a good sense of how all the options perform, we can choose the best option and have a strong rationale.

A second area where outstanding questions exist is whether the structure we have, with the statistics we can attach, is really sufficient to accelerate the kinds of queries users want to run to be fast enough for a good user experience. To get a better feel for this we will need to incorporate more statistics and then get some users to evaluate.

4.2.2 Feature development

Major feature development priorities are listed below.

- Message parsing support for all the message encoding formats we intend to support, not just ros1msg. This includes protobuf and CDR at a minimum, and likely flatbuffers as well.
- Support in the query language for descending into variable-length complex arrays. Today we only parse fixed-length arrays. This is important because it is relatively common for data producers to batch multiple logical messages into a single message using an array, and they will still require the ability to execute queries on that data. Likewise, we should (under some circumstances) gather statistics on these arrays as well and store those on the inner nodes.
- The query language must be extended with a "neighbors" keyword or similar, for a bidirectional as-of join.
- The set of collected statistics needs to be expanded, and also used to accelerate search.

Today we have a small handful of associative statistics (min, max, sum, count) for numeric and text fields. We will want to add bloom filters, trigrams, possibly distinct sets. Ways of accelerating spatial search would also be useful, and in many cases may correlate well with time. However, this will require a way for the user to tell us what data is spatial data, which we currently lack.

- dp3 is currently single-node only. While supporting clustered reads is simple, supporting multiple writers will require some design to ensure that only one node writes to a table at a time.

Acknowledgements

Thanks to James Smith for review.

References

[1] Amazon Web Services. *Amazon CloudWatch homepage*. URL: https://aws.amazon.com/cloudwatch.

[2] Michael P Andersen and David E. Culler. "BTrDB: Optimizing Storage System Design for Timeseries Processing". In: *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA: USENIX Association, Feb. 2016, pp. 39–52. ISBN: 978-1-931971-28-7. URL: https://www.usenix.org/conference/fast16/technical-sessions/presentation/andersen.

[3] Apache Spark Developers. *Apache Spark homepage*. URL: https://spark.apache.org.

[4] Arvind Arasu, Shivnath Babu, and Jennifer Widom. "The CQL Continuous Query Language: Semantic Foundations and Query Execution". In: *VLDB Journal* (June 2006). URL: https://www.microsoft.com/en-us/research/publication/the-cql-continuous-query-language-semantic-foundations-and-query-execution/.

[5] Donald Carney et al. "Monitoring Streams - A New Class of Data Management Applications". In: *Very Large Data Bases Conference*. 2002. URL: https://api.semanticscholar.org/CorpusID:7816630.

[6] Clickhouse. *Clickhouse home page*. URL: https://clickhouse.com.

[7] Chuck Cranor et al. "Gigascope: a stream database for network applications". In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. SIGMOD '03. San Diego, California: Association for Computing Machinery, 2003, pp. 647–651. ISBN: 158113634X. DOI: 10.1145/872757.872838. URL: https://doi.org/10.1145/872757.872838.

[8] Datadog. *Datadog log management homepage*. URL: https://docs.datadoghq.com/logs.

[9] Apache Iceberg Developers. *Apache Iceberg specification*. URL: https://iceberg.apache.org/spec.

[10] Foxglove Developers. *MCAP: serialization-agnostic log container file format*. Available from https://github.com/foxglove/mcap. 2024. URL: https://mcap.dev.

[11] Foxglove Technologies. *Foxglove*. 2024. URL: https://foxglove.dev.

[12] Google. *BigQuery*. URL: https://cloud.google.com/bigquery.

[13] Google. *Flatbuffers*. URL: https://flatbuffers.dev.

[14] Google. *Google Cloud Logging homepage*. URL: https://cloud.google.com/logging.

[15] Google. *Protocol Buffers*. URL: https://protobuf.dev.

[16] Object Management Group. *Common Data Representation specification*. May 1, 2024. URL: https://www.omg.org/cgi-bin/doc?formal/02-06-51.

[17] Richard D Hipp. *SQLite*. 2024. URL: https://www.sqlite.org/index.html.

[18] LanceDB Developers. *LanceDB project page*. URL: https://github.com/lancedb/lancedb.

[19] Steven Macenski et al. "Robot Operating System 2: Design, architecture, and uses in the wild". In: *Science Robotics* 7.66 (2022), eabm6074. DOI: 10.1126/scirobotics.abm6074. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[20] Charles Masson, Jee E. Rim, and Homin K. Lee. "DDSketch: A Fast and Fully-Mergeable Quantile Sketch with Relative-Error Guarantees". In: *Proceedings of the VLDB Endowment, Vol. 12, No. 12*. 2019. URL: https://www.vldb.org/pvldb/vol12/p2195-masson.pdf.

[21] Microsoft. *Azure Monitor Logs homepage*. URL: https://learn.microsoft.com/en-us/azure/azure-monitor/logs/data-platform-logs.

[22] Microsoft. *Azure Synapse Analytics*. URL: https://azure.microsoft.com/en-us/products/synapse-analytics.

[23] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*. Kobe, Japan, May 2009.

[24] Mark Raasveldt and Hannes Muehleisen. *DuckDB*. URL: https://github.com/duckdb/duckdb.

[25] ROS2 authors. *ROS2 Iron Irwini*. URL: https://docs.ros.org/en/iron/Releases/Release-Iron-Irwini.html.

[26] Amazon Web Services. *Redshift*. URL: https://aws.amazon.com/redshift.

[27] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. "The Design and Implementation of a Sequence Database System". In: *Proceedings of the 22th International Conference on Very Large Data Bases*. VLDB '96. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996, pp. 99–110. ISBN: 1558603824.

[28] Snowflake. *Snowflake home page*. URL: https://snowflake.com.

[29] Sumo logic. *Sumo logic home page*. URL: https://www.sumologic.com.

[30] Wyatt Alt and Jainil Ajmera. *dp3 project page*. URL: https://github.com/wkalt/dp3.