Time-series Tensor Organizer (Torg): a metadata and query system for complex time series data

Matthew James Bryan, Anuradha Ramachandran, Raman SV

Introduction

A common problem in the sciences is the need to index complex time series data which is scattered across many files and devices, and to deliver it as a single mathematical object, such as a PyTorch Tensor. For example, neuroscientific data collected across many sensor types (e.g., implanted sensors, cameras, wrist accelerometers) is typically stored across many files on a per-experiment and/or per-sensor basis. Additionally, we need ways to query events which occur at various points in that data set, for example: times when a specific patient initiated a specific movement.

Approach

By leveraging a SQL database as a metadata store, we can allow users to query across time series and events, and deliver to them a single object (e.g. tensor) containing the result of their query. The data rests in an arbitrary storage format, on an arbitrary storage technology. For example it may be a collection of HDF5 files resting in AWS S3 buckets. The metadata store provides a way to query and to address that data. A Python layer built on top provides the mechanism by which the user declares and writes the metadata, and queries that metadata to produce the output tensor. See Figure 3.

Our system revolves around two concepts: 1.) time series data, which can be multidimensional; and 2.) events, which are intervals on the timeline, and which involve 1 or more time series. Both concepts allow for metadata which can later be used for querying. A time series is identified by a user provided string, which they will be able to map back to the series' access paths. For example: it may be a fully qualified path on a filesystem. In our system we will refer to this as the time series's universal resource identifier (URI). An event is provided in reference to one or more URIs, together with a start/stop time, and any event-level metadata.

The two types of objects can be queried jointly or independently, using the power of SQL on the back end. An example of a query involving only the time series directly: "Give me data for patient 100 on day Y, for all sensors of type Z." An event query can involve both time series and event metadata, for example: "give me sensor data from neural implants only, for patient 100, where they were moving their left hand." The user specifies these queries using an object relational mapping (ORM). The latter query is depicted in Figs. 1, 2.

```
# 'loader' is an adapter for the hdf5 format
loader = H5demo.H5TensorLoader
sf = StaticFilter("Neural", metadata={"patientID": 100})
ef = EventFilter("hand movement", metadata={"whichHand": "left"})
# 'db' is our database adapter
events, event_timestamps = query.query(db, loader, static_filter=sf,
                                       event_filter=ef)
```

Figure 1. Object relational mapping (ORM) specifying the query

```
select o.uri, e.eid, e.start_time, e.end_time
from events as e, eolink as eo, objects as o,
     eventclasses as ec, objectclasses as oc
where e.eid = eo.eid and eo.oid = o.oid and
      o.oclid = oc.oclid and e.eclid = ec.eclid and
      oc.name = 'Neural' and ec.name = 'feed_drops'
      and e.metadata->>'whichHand' = 'left'
order by e.eid, o.uri;
```

Figure 2. Generated SQL query

SQL is a natural choice of technology for this use case. While it is plausible to store the time series data directly in a SQL database, we believe this result in unnecessary duplication of the data, and it is unclear whether SQL scales well to that use case. We suspect not, given SQL's focus on bag semantics and joins. Leveraging a database also allows us to share and secure a single metadata store for a large organization. That reduces redundancy, and improves reproducibility of results. The use of stored procedures adds a level of modularity, abstraction and security as this limits the end user's ability to interact with other database tables, including confidential data if any. That creates logical independence. Further, user roles with the stored procedures further enhances the security.

Software platform

We implemented the first version of this project using a local SQL server (Postgres) instance, which may be necessary for some scientific applications. A large proportion of scientific data analysis tools run local to the data, or on a

machine accessing the data over a network attached filesystem. However, the project will scale well to cloud deployments such as AWS or Azure, which may better reflect production ML pipelines. Our product is agnostic to storage medium and access path, since we require the user to provide adapters for those aspects. Hence: it is not tied to a particular environment per se. Additionally, SQL is widely available across software as a service (SaaS), container (e.g. Docker), virtual machine, and bare metal deployments, implying this design choice is not a strong constraint to adoptability.

Programming language

Our initial implementation of the application library is written in Python. C++ is a more general choice, since one can build a cpython library on top of C++, but due to our compressed timeline and the concentration of data science tools in Python we decided to aim directly for a Python implementation first. We believe this doesn't hamstring the extensibility of our approach since we could in theory perform an interface-compatible refactor of our library to be implemented in C++.

Testing and collaboration

We collaborate our development and test our system manually using Jupyter notebooks. Our notebooks generate demo data, and provide example user workflows. For this project, we have access to "AWS Educate" which provides us with cloud credits for the initial phases of the development.

Design details

The user provides time series data collected from diverse sources, storing it either locally or on cloud services like AWS S3 buckets. The user's responsibility is to declare that data and its corresponding metadata to Torg. Central to our architecture is a Python layer that mediates between user requests, the SQL database, and the storage. This layer supports three primary Python workloads: metadata ingestion, event metadata ingestion, and querying. A visual summary of our system design is illustrated in Figure 3.

Database adapter

To enable the use of various SQL flavors, we provide a layer of indirection between the Python code and the database. An abstract base class in Python (`DatabaseConnection`) defines the interface between the database and the rest of the library. We additionally provide two child classes for two specific implementations: Postgres (`db_postgres.py`), and Microsoft Open Database Connectivity (`db_odbc.py`). An additional SQL-less implementation using in-memory dictionaries is also provided for the purpose of debugging (see `H5Demo.py`).

Time series and event classes

Prior to performing any ingestion, the user first declares data and event classes. The declaration is a Python call, where the user provides the name of the class, as well as the attribute names and data types for any metadata they may provide later related to instances of that class. All time series and events ingested belong to exactly one such class.

For example: if a time series contains data for patient X from sensor Y on datetime Z, the class to which that time series belongs may be called "sensorData" and have metadata fields "patientID", "sensorID", "startTime" and "endTime".

The declaration in advance allows the database layer to set up any relevant schema and input validation. We chose not to set up any such validation for our initial version, but it is straightforward to implement in the future.

Ingestion layer

The user provides the URIs of the time series objects, together with their corresponding class and metadata. All time series must have a start/end time provided. This process is called "ingestion." Metadata is handed to the Python layer as a dictionary object, then encoded in a database-specific way and sent to the database for insert. For example, in the Postgres version of our demo we JSON-encode the dictionary as a JSON object and send it verbatim to the database.

Event filtering / ingestion

Event filtering involves the user defining criteria or conditions through a callback function to select specific events of interest based on some filter. The callback receives the classes and URIs of all objects matching some query. The callback function returns the class, URIs, start time, end time, and any other metadata associated with the event. As with time series ingestion, that information is sent to the database via the database adapter layer.

Storage adapter

As alluded to above, the user provides a storage adapter, which maps a URI and time range to an object which can be cast as a PyTorch tensor. This is the layer where the user maps the URI on to an access path, and handles authentication and authorization for accessing the data. If the data all rests in a single HDF5 file on the local disk for example, the user can parse the URI to determine the fully qualified path of the file, and the path in the HDF5 file's namespace to the object. In this case the filesystem's permissions / access

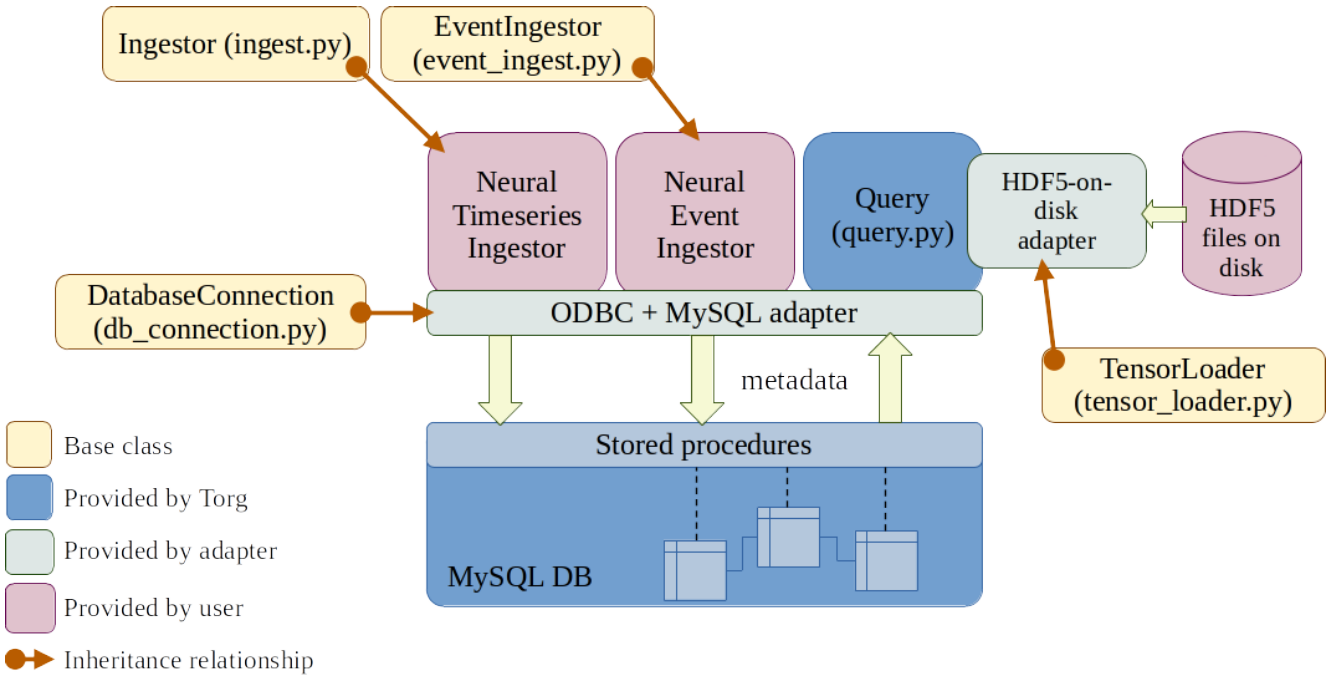

Figure 3. System Design diagram

control will be used to gate the adapter's access of the data. The layer of indirection provided by this adapter abstracts out the storage, allowing our product to solve the precisely the problem for which it was designed: the querying of the time series, rather than the storage.

We provide two reference implementations: one for HDF5 files stored on an accessible filesystem (`H5demo.py: H5TensorLoader`), and one for CSV files (`csv_demo.py: CSVTensorLoader`).

Querying

The query operation retrieves information related to both time series and events from the data source. The result of the query operation is returned as PyTorch tensors, which are multi-dimensional arrays suitable for use in PyTorch. In this operation, additional PyTorch specific parameters can also be passed such as the ID of the device where the data should be materialized (e.g. main memory or GPU memory). The shape of the tensors returned will depend on the nature of the query. In general, the rules for assembling the tensors are as follows:

1. If there are multiple instances of something returned, such as multiple instances of an event, those instances are returned as a list of tensors, rather than a single tensor. It is common convention to concatenate tensors together into a single tensor and for the "batch dimension" to be the first, but we want to leave the door open for events to have different lengths of time. PyTorch only allows concatenation of tensors if their non-concatenated dimensions have equal shapes.

2. The time dimension is the first dimension of each returned tensor. Within a given instance (e.g. event), individual time series are concatenated along the second dimension, regardless of the total number of dimensions. Note that there can exist other dimensions which we don't specifically address. PyTorch concatenation rules require that those dimensions have the same shape across all concatenated tensors.

3. Any individual time series which are combined to create a single instance are cropped on the left and right side to avoid "ragged tensors". That is - only the time intersection of the data series is returned. This is ineffect a "time inner join", but performed using time range objects in Python rather than in SQL. A simple SQL approach would be a poor choice here due to e.g. jitter, where two timestamps may be nearly equal but not quite. Data structures to index in time and perform time joins would be an excellent subject of future work!

For example: suppose we have two sensors, whose data is stored in tensors of shape (time, 3, 2), where the the last two dimensions relate to some aspects of the sensed data, e.g. space or frequency. Suppose further that we are querying events which are related to those two sensors. When the

query is performed, we take the following steps, following the steps above:

1. The SQL system is queried, which returns a list of event instances, including the URIs to which those events relate. Those URIs correspond to our two sensors.
2. The user-provided storage adapter is leveraged to load the data from the corresponding URIs, and corresponding time ranges.
3. For each event, we crop the provided time series to avoid the ragged tensor problem, and concatenate them along the last dimension. That results in a per-event tensor of shape (time, 3, 4).
4. The tensors for all events are put in a list.

Database schema

While the Python layer is not opinionated about the database's schema, we nevertheless provide two specific implementations: one for Postgres, and one for SQL Server via ODBC. Both implementations have a similar schema, which you can find in Fig. 4.

Note that we added a clustered index to depict one potential speed up, but we have not performance tested it. Our reasoning is that clustering based on object class is sensible because of the nature of our queries: we will tend to query based on one object class. This is a design decision to be tested later, and this may be premature optimization.

Postgres adapter

This was our first reference implementation, and it excludes some of the more fully-featured aspects of our ODBC approach, in order to provide the simplest initial implementation. See `db_postgres.py`.

This approach formats SQL queries directly in the Python code. We are aware this has a number of downsides: 1.) it leaks implementation details to any casual reader, which risks breaking encapsulation and security; and 2.) it has little protection against SQL injection. It also has very little input sanitization, and thus it may allow exceptions to bubble up to the user which are difficult to understand, and which may leak important details. We are aware of these issues, but elided addressing them in favor of quickly providing a solution. The ODBC adapter below more thoroughly addresses some of them.

ODBC adapter

The team had two different installations of MS SQL Server for the database operations. One version was on a MS Windows system and used the native version of MS SQL Server Management Studio (SSMS) v18.10. For this version, we used the ODBC Adapter to set up a secure connection to the SQL Server database (local server) via the ODBC data source administrator in Windows. Since SSMS is not available on Mac systems, another team member used Docker and Azure Data Studio. The ODBC connection properties was similar between the two. The code for this connection is present in the notebook – `db_odbc.py`.

One of the key reasons for us to choose ODBC was that it provides a standardized interface for accessing different database management systems, making it easier to develop applications that can work with various database platforms without significant changes to the codebase. Further, ODBC is widely supported across various operating systems like Windows, Linux, and MacOS. Similar to ODBC, SQLAlchemy offers a great ORM approach that would have allowed us to work with databases using Python objects instead of directly writing SQL queries. However, for purposes of this project, we believe that ODBC was the better approach as we were able to develop queries and structures in a DBMS and integrate that with our code.

We created stored procedures that automated the data ingestion and table creation process. These stored procedures provide security by preventing unauthorized access to the database, and offering physical and logical independence. They only allow mutations that are coded in the procedure itself, thereby preventing leakage of implementation details or schema changes. In future implementations, we plan on isolating the stored procedures to a specific schema and limiting access of the ODBC connection to just this schema, further enhancing security.

A few examples of the stored procedures created and their details are below:

1. The `check_table_exists` procedure checks for the existence of a table and returns the result.
2. The `create_object` procedure parses the input from the user (using a user defined SQL function) to dynamically create a SQL Table based on the columns and datatypes provided.
3. The `check_object_class_exists` procedure checks for the existence of the `Class_Metadata` or `Event_Class` tables based on a parameter for each.

We have metadata values that are JSON fields stored as `nvarchar` datatypes in our tables. These are queried using SQL's JSON parser, which has a different syntax compared to the approach used by Postgres. An example of this code syntax is "`JSON_VALUE(e.metadata, '$.{k}') = '{v}'`", where metadata is the column name, the parameter "k" is the JSON header, and "v" is the JSON value. This is implemented in ODBC adapter class.

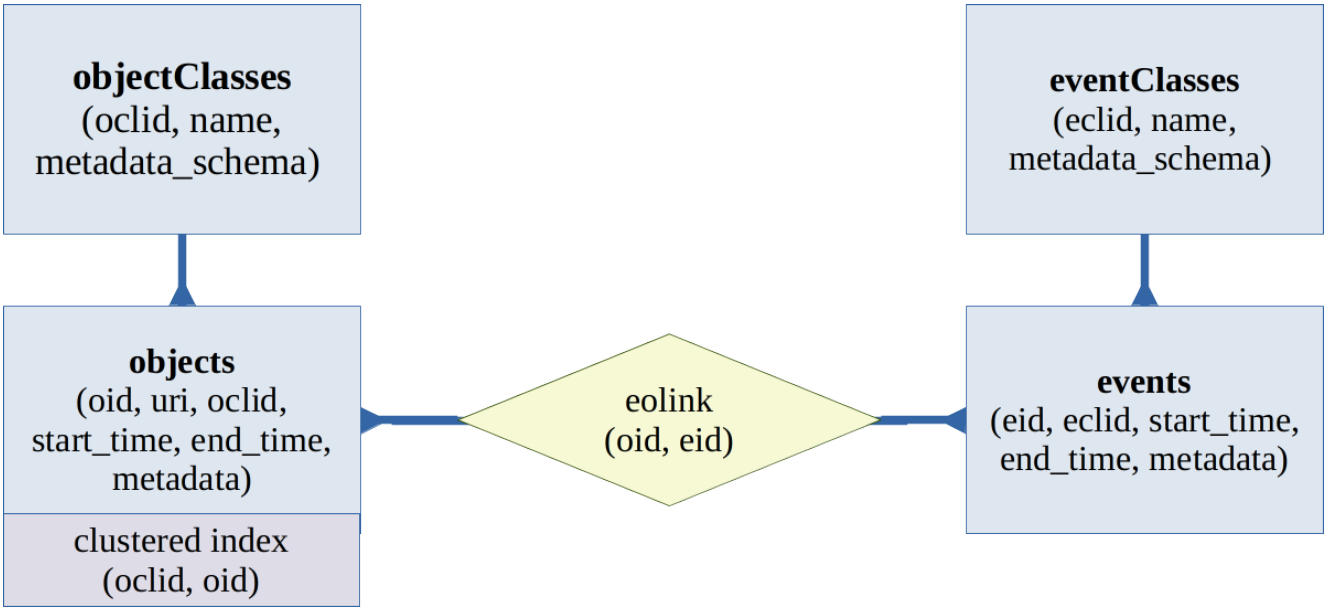

Figure 4. **Reference schema** *oclid*: object class id, *eclid*: event class id, *start_time, end_time*: timestamps encoded as UNIX epochs (i.e. floats), *metadata* a JSON-valued metadata field, *metadata_schema*: a varchar containing a JSON schema specification

Code examples

For the user's reference we apply the setup steps for our demo in `code/README.md`. Additionally, our `code/demo.ipynb` Jupyter notebook provides an example of full workflow, centered around HDF5 files which have schema resembling a real-world neural engineering dataset.

Connecting to a database

See `db_postgres.py` for an example implementation of a database adapter. In general, the user provides the database name and any authentication credentials to the datapter's initializer. In the demo Postgres case, it is assumed the running Python process has permissions to access a Postgres server running in the local host, through the corresponding socket. Hence creating the database connection looks like:

```
demo_db = DatabaseConnectionPostgres(
    database="torg",
    host="/var/run/postgresql/")
```

Note that object initialization does not imply a literal connection to the database. Torg uses this object to establish a connection for context of individual queries, and drops the connection once the query is completed. See `ingest.py:Ingestor.ingest` for an example usage.

Time series ingestion

Data class declaration and ingestion looks as follows:

```
ingestor = H5demo.IngestorH5Demo(demo_db)
ingestor.declare_object_class(
    'Neural',
    {'patientID': int, 'sensorID': int})
ingestor.declare_object_class(
    'Stimulation',
    {'patientID': int,
     'sensorID': int})
ingestor.declare_object_class(
    'WatchIMU',
    {'patientID': int,
     'sensorID': int,
     'manufacturer': str})
```

The `IngestorH5Demo` provides one example implementation for reference. Finally the user calls the convenience `ingest_files` method, which loops over the files in a directory and ingests any objects in them which are HDF5 format.

```
ingestor.ingest_files(paths_to_data)
```

Event ingestion

Event ingestion parallels the time series ingestion. In this demo case, the ingestor identifies time ranges where one of the time series had drop outs of its data stream, indicated by NaN values.

```
event_ingestor = H5demo. \
    EventIngestorH5Demo(
    demo_db)
event_ingestor.declare_event_class(
    "feed_drops",
    {"some_metadata": int})
event_ingestor.ingest_files(
    "feed_drops",
    paths_to_data)
```

Providing a storage adapter

The user leverages a storage adapter for the data access. They simply need to provide a reference to its class:

```
loader = H5demo.H5TensorLoader
```

In this case the adapter loads HDF5 objects from files accessible over a filesystem path. The URIs contain both the filesystem path, and a namespace path inside the HDF5 file, separated by a pipe symbol.

Query

The query interface accepts two ORM-style objects: `StaticFilter` and `EventFilter`. The former selects time series, the latter selects events. In both cases the user provides a class, and optionally a time range, a URI list, and/or metadata parameters. The predicates making up the filter object are combined with conjunctions, and the URI list is built up with disjunctions. That is - a URI list is built up to mean "give me all the URIs in this list", and then all filter terms are combined conjunctively.

```
sf = StaticFilter("Neural",
    metadata={"patientID": 100})
data, timestamps = query.query(
    demo_db, loader,
    static_filter=sf)
```

The 'data' object here is a PyTorch tensor or list of PyTorch tensors containing the time series data, assembled using the method enumerated in Approach. The 'timestamps' object contains a tensor or list of tensors, of the same length as 'data', which lists the timestamps for for the given time series objects. The timestamps are generated assuming that all time steps are equally spaced. That is a somewhat limiting assumption, but works for most applications.

Events are queried similarly, and can be queries in conjunction with the time series. See above Fig. 1.

Evaluation

We qualitatively evaluated several dimensions of our solution:

- **Support for multiple DB flavors**: we demonstrated this by providing adapters for multiple types of databases.
- **Cross-platform support**: our demos demonstrate that Torg runs on Linux, Windows, and in a containerized Azure environment.
- **Support for multiple time series data formats**: we illustrated the ability to swap in CSV and HDF5 formatted data.
- **Querying time series based on custom metadata**: we showed through manual tests that the user can query time series based on data class, time range, and custom metadata
- **Querying events based on custom metadata**: similar to time series querying, we showed that the user can query events based on event classes, data classes, time ranges, and metadata.
- **Truncating ragged tensors**: we showed that Torg provides effectively a "time inner join" by truncating ragged tensors along the time dimension to a contiguous span shared by all of them.

We did not quantitatively evaluate our proposal, but doing so would be high priority if this was a semester-long course. Our primary concern would be performance at scale: how well does our approach work when scaled to order-of 100k events? Would we need to redesign the schema, or the order that we pull data from disk? It's plausible the storage callbacks would need to be altered somehow to better support performance.

Related approaches

A related methodology has been explained in this paper [4] where metadata for each time series can be stored in a relational database and the links to the data that contains unique identifiers for each sensor can be created during the data ingestion phase. This paper mentions InfluxDB [2, 3], which solves the same problem we are solving, but in a more monolithic way. It provides data storage and a query language (similar to SQL), in addition to the metadata storage and querying. Our proposal is different in the sense that it isn't monolithic: it drops in on top of a user's existing data store. That may in-fact be required, since the existing data store may be subject to HIPAA or FDA requirements. In trade-off, our solution probably also results in data accesses which are inefficient. InfluxDB presumably has sophisticated caching and storage mechanisms; we assume our users will handle those details, together with any e.g. filesystem caches which may be on the user's system.

An additional advantage of our approach is the very low ramp up time and adoption cost compared to InfluxDB.

Dell Pravega [1] is a real time streaming and storage platform with built in querying and retrieval. Its concentration is on high bandwidth streaming ingest and storage. For example: it can be used to ingest lidar and RGB data from a smart car in real time. It allows for InfluxDB querying of the streams it stores. Thus: it is intended to solve the storage performance aspect of the problem (and to sell Dell products in the process, presumably).

Conclusions

Our project was an exploration aimed at addressing the myriad challenges associated with indexing and querying complex time series data, and storing and retrieving events and metadata without forcing the user to choose a particular storage technology or DBMS flavor.

We implemented the Torg Python library that allows a user to declare time series and events, along with corresponding metadata. An ORM query interface allows the user to query those metadata and to receive the results as PyTorch tensors. A multilayered architecture and use of callbacks allows Torg to provide a simple user experience while allowing for the use of myriad storage technologies and SQL flavors. Accomplishing this required us to deepen our understanding of SQL schema design, and of the relationship between an application and the DB with which it communicates. That lead us to the use of stored procedures, and a two-tiered ORM: one tier which is agnostic to the choice of database flavor, and another which isn't.

ORM Design and indirection: Our project necessitated querying databases from an object-oriented perspective, that also includes meticulous ORM design. By leveraging built-in ORMs from libraries like `pyodbc`, alongside a custom-built ORM tailored for querying time-series data, we achieved a versatile solution capable of interacting seamlessly with multiple flavours of database management systems.

Schema design and time representation: The conceptual representation of time, time series, and events formed the core of our project. Our schema design was meticulously crafted to ensure flexibility for concatenating tensors while maintaining conceptual simplicity. This underscored the importance of a thoughtful schema design.

To conclude, our project represents a nuanced exploration of ORM design, schema architecture, and temporal data processing, that not only challenged but also enriched our understanding of database management systems and object-oriented programming.

Division of work

It is difficult to track the nature of each member's contributions on the written materials and poster; likewise with the high-level idea development. However we present the technical contributions here:

Matthew Bryan:

- Created CSV and HDF5 demo datasets
- Implemented the Python layer
- Implemented the HDF5, CSV, and Postgres adapters
- Designed and implemented the Postgres schema
- Designed and implemented the demo workflow

Anuradha Ramachandran:

- Co-created the ODBC adapter
- Co-created the SQL Server stored procedures
- Completed demo workflow for Azure Data Studio
- Managed our team's GitLab repo

Raman SV:

- Co-created the ODBC adapter
- Co-created the SQL Server stored procedures

References

[1] Dell. Why Pravega. https://cncf.pravega.io.

[2] InfluxDB. Explore Data using InfluxQL. https://docs.influxdata.com/influxdb/v1/query_language/explore-data/#the-basic-select-statement.

[3] InfluxDB. InfluxDB Time Series Data Platform — InfluxDB. https://www.influxdata.com/.

[4] Brian McBride and Dave Reynolds. Survey of Time Series Database Technology, 2020.