Machine Learning Data Platform

PHASE I RESEARCH AND DEVELOPMENT REPORT

Osprey DCS
January 10, 2023

Open for public release.

Osprey Distributed Control Systems

OSPREYDCS/TM-01/2023

Osprey Distributed Control Systems

MACHINE LEARNING DATA PLATFORM*
Description, Operations, Performance, and Limitations

Authors:
Christopher K. Allen
George McIntyre
Michael Davidsaver
Craig McChesney
Bob Dalesio

Date Published: January 10, 2023

Prepared by
OSPREY DISTRIBUTED CONTROL SYSTEMS
304 Blue Heron Court
Ocean City, MD 21842-2452

*Work performed under the auspices of the U.S. Department of Energy with funding by the Office of High Energy Physics SBIR Grant DE-SC0022583.

CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

ABSTRACT

Osprey Distributed Control Systems has initiated development of a *Machine Learning Data Platform* (MLDP) for the high-speed collection, storage, retrieval, and management of heterogeneous, time-correlated data available from the controls system of large particle accelerator facilities and other larges experimental physics facilities. It is intended as a "machine-learning ready" host platform for data mining, data science, and building machine learning applications for the diagnosis, modeling, control, and optimization of such facilities. There are two primary functions of the platform, 1) high-speed data acquisition and storage of time-correlated, time-series, heterogeneous data, and 2) broad retrieval of archived heterogeneous data by time correlation and other associations relevant to data science and machine learning applications. The Machine Learning Data Platform consists of two separate systems, the *Aggregator* and the *Datastore*. The first function of the platform is partially realized with the *Aggregator*, which is deployed within the Experimental Physics and Industrial Control System (EPICS). It is the front end of the platform providing high-speed, real-time acquisition of online facility data. The remaining functionality of the platform is realized with the *Datastore*. The Datastore is a standalone system intended for high-speed archiving of acquired data and rapid retrieval of the time-series data in broad forms suitable for machine learning and data science applications. Moreover, it partially supports the annotation of data sets and the storage of post-processed data; full support is intended in future development. In addition to the Aggregator and Datastore systems, a standalone *Web Application* was developed as a companion to the Datastore, allowing independent inspection and interaction with the data archive using a standard web browser. The performance goal of the platform was stated as the continuous acquisition of 4,000 signals at a sampling rate of 1 kHz. Osprey DCS has now completed the initial development phase of the Machine Learning Data Platform resulting in a working prototype. A comprehensive description of the design, the operation, and the performance of the MLDP prototype is provided in this report, the results are then summarized and recommendations for future development are offered. Briefly, the developmental status of the Aggregator system is mature whereas the Datastore, although operational, still has issues that need to be addressed in future efforts in particular regarding the above performance goals. This report is part of the evaluation effort for the MLDP prototype and is intended to provide guidance for the courses of future development; a focused critique is made of all the Datastore systems establishing its status so that strategic future efforts can be made.

PART 1: Overview

1. PROJECT OVERVIEW

1.1 BACKGROUND

Machine learning offers alternative approaches to prediction, control, and operation of particle accelerator systems; these techniques are data driven rather than physics centric. Moreover, due to the modern interest in Artificial Intelligence (AI) and Machine Learning (ML) there is a wide availability of off-the-shelf and open-source tools for building AI and ML based applications (e.g., (1) (2) (3) (4)). The current situation provides the opportunity for the rapid development of AI and ML based models and techniques for the prediction and control of accelerator systems, specifically, for those based upon data science. Yet one immediate difficulty in applying data science techniques to accelerators is the sheer scope of the data to be collected, processed, and maintained. Particle accelerator systems are typically large, sophisticated facilities containing potentially thousands of beam control and diagnostic points. Machine learning and general data science algorithms specific to accelerator systems thus require enormous data collection, archiving, and processing capabilities. Moreover, for online control applications, data acquisition must be fast enough for data-science based control algorithms to recognize and correct for changing machine conditions during operations. Thus, for the successful implementation of data-science techniques applied to an operating facility, access to operations data must be comprehensive, timely, and extensively catalogued. Furthermore, there is not a standardized method for supporting such applications. The current demand for machine learning and artificial intelligence applications asserts the need for development of next generation data acquisition, archiving, and management technologies for particle accelerator systems (5) (6) (7) (8).

Upon motivation from accelerator community to apply advanced machine learning techniques in the diagnosis, prediction, control, and optimization of particle accelerator systems (e.g., see (9) (10) (11), etc.), Osprey DCS began the development of a "machine-learning ready" user platform, called the Machine Learning Data Platform (MLDP). Funding for the initial project was obtained through a Phase I Small Business Innovative Research (SBIR) research grant sponsored by the United States Department of Energy (DOE) Office of High Energy Physics[1]. From this funding opportunity a prototype for the MLDP was developed. Osprey was able to leverage off expertise in accelerator systems and the Experimental Physics and Industrial Control System (EPICS) to rapidly develop the platform into an operational prototype.

The MLDP is designed to support general machine learning and data science applications for particle accelerator facilities. The platform is capable of ingesting large data sets available from the control system then presenting data scientists and application developers with a *consistent*, *data-centric* programming interface to the archived facility data. Data science and machine learning algorithms can then be quickly prototyped, tested, and deployed in organized fashion. The platform is designed to be facility independent, where applications developed at one facility would operate with minimal or no modification at other facilities. This aspect is motivated by the nature of machine learning, where algorithms condition themselves upon the data they are given, irrespective of the source. The platform is designed for deployment at any EPICS-based facility. However, there are Application Programming Interfaces (APIs) available for any control system

[1] Grant #DE-SC0022583 - "A Data Science and Machine Learning Platform Supporting Large Particle Accelerator Control and Diagnostics Applications."

to populate the MLDP data archive. The archiving and querying component of the platform, the Datastore, is standalone and can be deployed anywhere.

The Web Application was initially built as a development tool for archive inspection. The utility of this tool became immediately apparent as it provides *universal access* to the data archive. Data scientists and researchers can inspect and interact with a common archive from any remote location using a standard internet web browser. Consequently, Osprey DCS pursued development of the Web Application as a complement to the MLDP providing an additional suite of data science capabilities. It is an independent, standalone system running on a separate host platform providing universal access to the data archive through a common URL.

1.2 PROJECT GOALS

Several goals were stated at project initiation, for both performance and operation.

1. Acquire and archive 4,000 signals at a sample rate of 1 kHz.
2. Archive and query time-correlated heterogeneous data.
3. Annotate archived data with additional user metadata.
4. Support the storage and management of processed data obtained post ingestion.
5. Wide query capabilities of the data archive.

1. The first goal is a straightforward performance goal. For concreteness, let the signals be float values requiring 8 bytes each. Then we must be able to acquire and archive 32 Kbytes every millisecond. Or more simply, the data ingestion rate must be minimally 32 Mbytes/second. For Java implementations where float objects require 24 bytes of storage, the minimum ingestion data rate becomes 96 Mbytes/second.

2. The archive contains heterogeneous data. This includes scalar data of different types such as Booleans, integers, floats, and character strings. Additionally, complex data types such as numeric arrays, tables, statistical data samples, complex data structures, and images are to be archived. These differing data types must be correlated and available within the same results of a single data request.

3. Users of the MLDP must be able to annotate archived data with notes, data associations, and other artifacts both during acquisition and post-acquisition. Such information is then available to other data science applications through the archive.

4. Users of the MLDP must also be able to annotate the archive with calculations obtained from the data archive itself. The overall objective is to provide users with the ability to annotate the archive with information obtained by data mining, machine-learning, and other general data processing applications post-acquisition. These post-ingestion results are then available to other users of the platform.

5. The wide data query objective is more general, the ability to request heterogeneous data via time-correlations, metadata, relationships, and other data properties, within a single query request. That is, the data requests are not restricted by the data type, data source, or time range. Additionally, Single requests containing disparate types such as scalars, arrays, images, and data structures can be queried according to their common properties, such as hardware system,

equipment, alarm status, or user attributes. We expect the data rates for the query operations to be on par with that of the ingestion service.

In addition to the original objectives stated at project inception, the following supplemental achievements were also realized during the project:

6. An independent Web Application was developed for inspection and interaction with the MLDP data archive using a standard internet web browser.
7. The Datastore component is a fully standalone system and can be adapted to a variety of data providers. Thus, data science applications developed for the MLDP have broader applicability.

6. The Web Application is useful as a standalone tool for data scientists. It allows for independent inspection, interaction, and annotation of the MLDP without any other resources other than a standard web browser. Important properties within data sets can be identified without any programming or scripting. In addition, it provides universal access to the MLDP archive from remote locations via a common URL.

7. The Datastore can be used to archive data from sources other than the Aggregator system; it is not limited to EPICS-based facilities. All that is required is that data providers conform to one of the available ingestion APIs offered by the Datastore (there are currently two). Thus, any accelerator or large experimental physics facility can utilize the archiving and search capabilities of the Datastore system. The Datastore system itself is versatile having broader applicability.

1.3 PROTOTYPE

An operational prototype of the MLDP has been developed in the Phase I effort. The MLDP has two separate components, the data aggregator service, or *Aggregator*, and the data archive service, or *Datastore*. The two subsystems were developed and tested independently. Development of the Aggregator was leveraged off previous efforts to create a similar system for synchronous acquisition of beamline data. Thus, the Aggregator system is far more mature. The major focus of the current development effort was the Datastore archive management system. The Datastore must be able to ingest and archive data at rates provided by the Aggregator. It is also the interaction point between the MLDP and data scientists. It requires continuing interaction with data scientists and other users to build out desirable features. Thus, the Datastore utilized the major portion of the development effort.

The Datastore component has utility as a standalone system. To demonstrate its versatility, the data ingestion capabilities were tested using two very different simulated data sources. A data simulator that emulates the operation of the Aggregator system was used in initial operations and performance testing. Additionally, Osprey DCS had previously developed a data simulator emulating the MPEX facility at Oak Ridge National Laboratory (12). The latter simulator provides data in a very different format than the Aggregator system, however, the prototype Datastore system operated consistently over the two differing data sources. Thus, the Datastore system has broader applicability.

In addition to the MLDP prototype, the Web Application was also developed as a companion utility for the Datastore. The Web Application was initially intended as a development tool for independent inspection and verification of the data archive without the requirements of a formal API, or other supporting services. However, the Web Application was found to have independent

merit as it allows anyone, including developers and data scientists, to inspect the data archive without any programming or scripting requirements. That is, it provides inspection and ad hoc data mining of the archive. Additionally, it provides universal access to the data archive as users have remote access requiring only a standard internet web browser.

1.4 DEVELOPMENT STRATEGY

To expedite initial development and accelerate implementation of the MLDP prototype, it was decided to exploit all potential Components Off-The-Shelf (COTS) software and available third-party technologies. However, use of third-party components was almost exclusive to the Datastore system which required most of the development effort. The Aggregator component was built only with available EPICS components, leveraging off our experience with EPICS control system, and synchronous acquisition systems in general.

The Aggregator system is better defined than the Datastore. Its requirements are primarily technical in nature, being dictated by specified data rates and subsequent hardware operations. This is a subject well understood by Osprey DCS and, consequently, we were able to develop the Aggregator system in-house without external technologies (i.e., other than EPICS).

Development of the Datastore system involved significant research as well as development; thus, we were prepared to explore all available design and implementation options. The use of third-party systems within the Datastore component could potentially yield sub-optimal results in initial development, however, doing so allows us to quickly evaluate the prototype and identify which areas require future attention. It also allows data scientists to evaluate a working prototype and provide feedback. Our approach first provides us with a working prototype from which we can then optimize to obtain stated project goals. Thus, the current design and implementation for the Datastore system is the most dynamic, evolving from both performance evaluations and from feedback from the machine learning and data science community.

As a specific example of the implementation strategy, we chose the gRPC technology as the communication protocol for the Datastore component, both for speed of development, and for broader compatibility (i.e., application to control systems other than EPICS). Although it appears that gRPC satisfies our current data throughput requirements (see Subsection 10.3.4), ultimately it may be necessary to replace gRPC with the EPICS pvAccess protocol to increase performance. Such an action would constitute a tradeoff between performance and limiting the scope of the Datastore applicability.

We identify specific third-party components and systems that are utilized in the MLDP prototype. The Aggregator requires the EPICS environment for operation. It also optionally produces output in the form of HDF5 files, which are self-describing data files based upon the Hierarchical Data Format specification (13). The Spring and Spring Boot Java libraries are frameworks for general application development and service development (13) (14). They contain code libraries, standalone tools, and other resources for rapid application development, including service development, within their pre-defined paradigms. Much of the core Datastore code library is based upon these frameworks. Also used within the Datastore core is the InfluxDB database system. The InfluxDB database is specifically intended for real-time data acquisition applications (15). It is designed for the storage and retrieval of time-series data and was used as a primary component in the data archive. The InfluxDB database system is based upon the model of a “measurement” and uses a proprietary language *Influx* to store, retrieve, and analyze time-series data. The data

archive also uses a non-relational, document-based (i.e., "NoSQL") database to store metadata associated with the data archive and the acquisition process. The MongoDB system was selected for this document-based database (16). It has its own API supported by multiple programming languages, but also supports SQL-like query statements. The Datastore communications are realized through gRPC (17). This is a remote procedure call (RPC) technology based upon Google's Protocol Buffers framework (18). The RPC protocol is defined in the "proto" meta language then compiled into standard programming languages which implement the Protocol Buffers framework. Finally, the React Javascript library is used within the Web Application (19). This library is a framework for building services and user interfaces in internet web browsers.

1.5 PROJECT STATUS

The status of the project is assessed from a detailed evaluation of the current MLDP prototype provided in this report. This includes detailed critiques of the system architecture, its performance, and any operational limitations of the prototype. The results are summarized, and outstanding issues are identified within the Project Summary of Chapter 10. Recommendations for future development efforts to remedy these issues are offered in the final Chapter 11.

Regarding the stated project objectives, the following observations are made concerning the current prototype:

1. The Aggregator system is operating close to the stated performance goals. It was tested to successfully acquire 3,200 signals sampled at 1 kHz and produce data tables with 1,000 rows every second for an overall data rate of 26 Mbytes/second. The Datastore, however, still requires significant performance improvements. Maximum continuous, sustained ingestion rates ranged between 0.2 to 7 Mbytes/second. Continuous data rates of 2.0 Mbytes/second were obtained for large data streams, however, equivalent amounts of data ingested as "wide data frames" was as slow as 0.2 Mbytes/second. Burst rates were substantially higher, in the 30 Mbyte/second range. Network data transmission rates through gRPC communications were seen as high as 380 Mbytes/second.
2. The Datastore system can successfully ingest scalars of type Boolean, integer, and floating point and complex data of either byte arrays or images. Numeric arrays and complex data structures are transmitted properly, both through ingestion and query, but are archived incorrectly. Additionally, there are subtle issues with timestamp ingestion and assignment.
3. The ability to provide user attributes and other metadata during ingestion is available. However, post-ingestion modification of the data archive is not yet available.
4. The feature to annotate archived data *post-ingestion* (or "post archiving") has not yet been implemented. Thus, the ability to include calculated data has also not been implemented. However, the framework for post-ingestion annotation is in place, as the Datastore contains a separate repository for metadata capable of storing such information.
5. The prototype supports wide query capabilities of the data archive, and the operation is robust. However, data rates are still below performance goals, maximum rates are seen in the 2.0 to 2.7 Mbytes/second range. Data transmission rates averaged about 170 Mbytes/second.

Regarding the additional projection achievements, consider the following observations:

6. The web application prototype can inspect all archived data and metadata, but still requires additional functionality. Current inspection methods are via time ranges and associated metadata properties. It is launched by simply connecting to a specified host URL.
7. Datastore ingestion operation was confirmed using two different data simulators, a simulator mimicking the Aggregator system and a data simulator mimicking the MPEX facility at Oak Ridge National Laboratory.

Each of the above issues is discussed in further detail within the sequel, including causes and remedies. Here we briefly outline some of the findings.

The data aggregation service, or Aggregator, is at a mature state of development. Osprey DCS was able to leverage off previous efforts in the areas of fast data acquisition and its expertise in the EPICS control system to expedite development. The current Aggregator implementation is essentially functioning at specification. However, additional testing on a platform offering 4,000 signals is necessary to confirm full performance capabilities.

The Aggregator was tested on an EPICS platform simulating the Linac Coherent Light Source (LCLS) Beam Position Monitoring (BPM) system at Stanford Linear Accelerator (SLAC). Specifically, the platform contains 200 nodes each supplying 16 separate time-series signals, representing the data obtained from BPMs at 200 locations. The Aggregator was able to collect data from the 200 locations and aggregate it into a single composite data table for transmission to the Datastore. Signal rates varied up to a maximum of 1 kHz such that the Aggregator produced composite tables of 3,200 signals and 1,000 rows delivering them at a 1 Hz rate.

Although functional, the Datastore component still requires important development efforts. Most features of the original specification are operational within the current prototype, with the notable exception of post-ingestion data annotations. Although the framework for post-ingestion archive annotation is in place, the feature itself has not yet been implemented. Additionally, the mis-archiving of numeric arrays and data structures must be corrected, along with a timestamping issues. Datastore performance is a key issue for further development, as current data rates are significantly below specification.

Although archiving rates perform adequately under burst conditions, continuous, sustained data rates average about 1 Mbytes/second for ingestion and 2.5 Mbytes/second for query operations. To meet specification these rates should be on the order of 33 Mbytes/second for native implementations and 100 Mbytes/second for the current Java implementation. Data transmission rates through network gRPC communications were seen between 170 and 380 Mbytes/second. This condition indicates that data transmission is performing as anticipated, however data processing is not. Performance characteristics could be immediately improved with hardware solutions, for example, adding more and faster servers. The InfluxDB database system is also being underutilized, for example, the full InfluxDB query capabilities and the multi-partitioning/multi-write head feature for InfluxDB installations are not being exploited. By using hardware solutions along with improved implementation, it is expected that performance requirements can be met. The InfluxDB database system should work, as it is specifically designed for these applications. However, alternative implementation solutions must also be considered if the expectation cannot be met. Employing an archive based upon the HDF5 self-describing file library, although developmentally intensive, is an alternative. These solutions and options are discussed further in Chapter 11.

Datastore ingestion was tested on two separate platforms, one simulating the Aggregator system, and one simulating the MPEX facility at Oak Ridge National Laboratory. Although the two platforms produce very different data formats, the Datastore ingestion tested consistently with comparable results. This result demonstrates the utility of maintaining the Datastore as an independent system within the Machine Learning Data Platform.

The Web Application is still in its initial development phase. It has a functional interface containing basic search capabilities for the MLDP data archive. However, it still requires stylization for enhanced user interaction and ease of interaction. The eventual addition of standard data science features, such as visualization, statistics, fitting, etc., would also greatly benefit the utility of the application, as well as the ability to export data. Features for data archive annotation are not yet available as this capability is not yet implemented in the Datastore core services.

Detailed performance evaluations of each component of the MLDP are provided in Chapters 6, 7, 8, and 9, along with descriptions of current operational limitations. A summary of these findings is included in Chapter 10 and recommendation for future development efforts are provided in Chapter 11.

1.6 OUTLINE

This report is divided into four parts, 1) a project overview, 2) MLDP design and operation, 3) prototype testing and evaluations, and 4) summary and recommendation. Part 1 provides a project overview and basic description of the Machine Learning Data Platform. It is comprised of the first two chapters. Part 2 describes the design, implementation, and operation of the MLDP in greater detail. It is comprised of Chapters 3, 4, and 5, each describing the Aggregator system, the Datastore system and the Web Application, respectively. Part 3 contains four chapters evaluating the MLDP prototype by category; Chapter 6 focuses on the Aggregator system, Chapter 7 evaluates the operation and performance of the Datastore ingestion service, Chapter 8 contains data integrity testing of the Datastore archive, and Chapter 9 evaluates the performance of the Datastore query service. The final Part 4 consists of Chapter 10 containing a summary of the prototype evaluations, and Chapter 11 containing recommendations for future design and development efforts for the MLDP based upon summary results.

Note that the organization of the report is from general to specific. The basic design, general operation, and status of the MLDP is covered in Part 1. Chapter 1 contains the project overview while Chapter 2 within Part 1 contains a concise, self-contained description of the MLDP intended for casual readers. Part 2 covers the design and function of the MLDP with increasingly greater depth. It contains a detailed description of the Datastore operations, as these systems are of primary concern in future development efforts. This part is intended for readers requiring greater depth of understanding (e.g., developers, project management, etc.). The evaluation section of Part 3 contains one chapter on the Aggregator system, then three chapters concerning various aspects of the Datastore performance and operation. The results of the evaluations section allow developers to identify operational issues with the current prototype and provide sensible strategies for improvement efforts. Thus, the final part contains both general observations and specific courses of future action. Casual readers can skip from Part 1 to Part 4 to obtain a general understanding of the Machine Learning Data Platform and then a summary of the prototype operations and the current project status.

2. MACHINE LEARNING DATA PLATFORM

The Machine Learning Data Platform (MLDP) is an integrated system for data science applications to model, diagnosis, control, and optimize the operation of charged-particle accelerator systems. The platform is designed for high-speed, real-time acquisition and archiving of all data available from the facility control system, in particular EPICS-based control system popular with large-scale facilities (20). The MLDP stores time-correlated, time-series, heterogeneous data to a central archive where broad access is available to data scientists. It provides a standardized hosting platform for management and mining of the resulting data archive. The archive is presented to data scientists through wide query capabilities including time ranges, data sources, data properties, user attributes, and other relationships through well-defined Applications Programming Interfaces (APIs). A further capability of the MLDP is to allow data scientists to annotate data with extraneous properties such as user notes, calculations, data set relationships, and other associations as discovered through post-acquisition analysis.

The data platform is divided into two primary systems, the *Aggregator* and the *Datastore*. The Aggregator provides fast real-time data acquisition and collection capabilities. Heterogeneous data is acquired from potentially disparately different data sources distributed within the EPICS control system. It is then collected, correlated in time, and realized as EPICS NTTable normative types (21). These tables are then coalesced at a central location where they are staged for transport to the Datastore system as composite data frames. The Datastore system ingests the heterogeneous data frames and stores them into the central data archive. It then provides data scientists with broad access to the archive through well-defined APIs.

A companion tool was also developed for the MLDP, the Web Application. The Web Application allows universal access to the MLDP data archive requiring only a standard internet web browser. It is an independent, standalone system running on a separate host platform allowing data scientists and engineers to search and interact with the archive from any remote location.

Although MLDP development is directed toward accelerator facilities employing the EPICS control system, EPICS is only required for the Aggregator system. Data transport within Aggregator system are via the EPICS pvAccess protocol (22) and, consequently, the EPICS system is required. However, the Datastore component is fully independent and standalone. All communications with the Datastore are based upon Remote Procedure Call (RPC) protocol. Thus, any data source capable of supplying time-correlated heterogeneous data may utilize the Datastore component independently. This versatility of the Datastore system was verified under testing evaluations with two different data simulators.

2.1 DATA FLOW

Here we describe the basic data flow and transport within the MLDP. This section also serves to introduce some basic terminology used within the MLDP, such as *data source*, *data provider*, *data archive*, and *data consumer*.

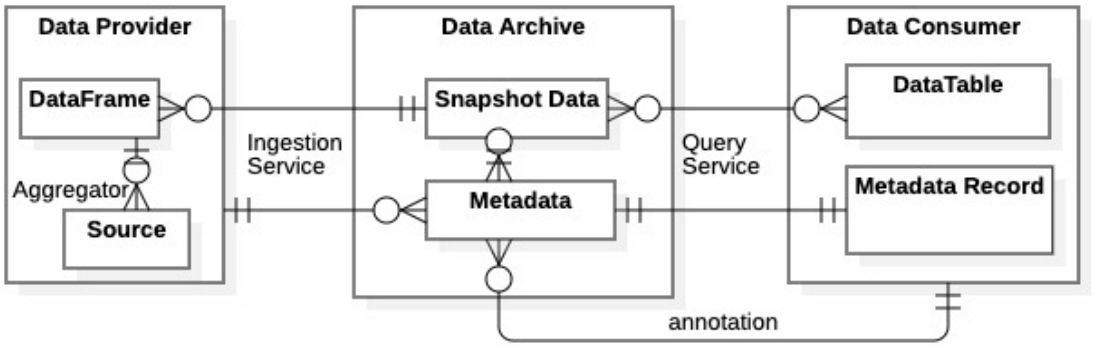


Figure 1: MLDP data transport and flow

The basic structure of the MLDP consists of a central data archive with a data acquisition and ingestion service at one end, offered to *data providers*, and a query service at the other end, which is offered to *data consumers*. Data transport through the MLDP is shown in Figure 1. At the left-hand side of the figure are the data providers, and specifically within the Data Provider box is the Aggregator service that acquires data from *data sources*. Data sources may be any device producing a signal, such as beamline hardware, diagnostic equipment, or any EPICS process variable (PV). A data provider is a more general term indicating a standalone system that collects and supplies formatted signal data from one or more data sources. The Aggregator service, as a data provider, acquires and collects data then stages it as a *data frame* for transport to the data archive. Shown in the diagram is the many-to-one relationship between the data sources, through the Aggregator, to the collected data frames. There is an additional many-to-one relationship between the data frames collected by the Aggregator and the data archive managed by the Datastore *Ingestion Service*; data frames may be simultaneously ingested from multiple data sources, as to be discussed. On the right-hand side of the diagram are the data consumers, which could represent data scientists, machine learning algorithms, data mining operations, or simply any interested party inspecting the data archive (e.g., the Web Application). The Datastore *Query Service* transports data from the archive to the data consumers. The data relationships for consumers are more complex, as will be covered in the sequel. Suffice it to say that data consumers have broad query capability over the data archive, especially concerning snapshot data.

Referring to Figure 1 there are two data channels, one for *snapshot data* and one for *metadata*. Specifically, the data archive contains both snapshot data and metadata. The term "snapshot data" refers to the heterogeneous, time-series data that is acquired from hardware devices attached to the facility control system. (Use of the modifier "snapshot" references the operation and archive structure of the Datastore which is covered below.) Metadata refers to any properties, annotations, or associations attributed to the snapshot data, the data sources, the data providers, the data acquisition process, or other artifacts that may be added post-acquisition. Within the figure, the one-to-many relationship between data providers and archived metadata is intended to illustrate that any single data provider may annotate the archived data with a multitude of metadata. Additionally, the metadata can describe multiple aspects of the data and data provider. Shown in

the diagram, data consumers may also provide metadata to the archive. The pictorial is intended to indicate that data consumers can provide metadata concerning both snapshot data and the data consumer itself.

Snapshot data requests are delivered to data consumers in the form of data tables, indicated by the DataTable entity in the diagram, while metadata requests are delivered as metadata records, indicated by the MetadataRecord entity in the diagram. The many-to-many relationship between archived snapshot data and data tables indicate that the table may reference multiple snapshots, and vice-versa. Metadata records are strictly one-to-one. Although there may be multiple metadata records associated with a single request, these are essentially atomic entities, all data consumers would receive identical records. As shown in the diagram, data consumers can add metadata to the data archive. Currently this feature is not fully implemented; only pre-defined attribute values can be modified although the intent is to broaden this capability in future implementations to allow full snapshot data annotations to data consumers. Additionally, post-ingestion annotation is not currently implemented, although the framework for doing so is available.

To frame the performance goals in the context of MLDP data transport, consider the following sequence of operations: 4,000 hardware signals are all sampled at 1 kHz. The values are time correlated and collected into a data frame with 4,000 columns, one for each signal. If one such frame is created for every second of acquisition, then these data frames contain 1,000 rows of data. The MLDP must be able to archive one such 1,000 x 4,000 frame every second.

2.2 BASIC DESIGN

The basic design of the full Machine Learning Data Platform is shown in the block diagram of Figure 2. As previously mentioned, there are two primary subsystems to the platform, the Aggregator and the Datastore, both independent. The data flow around the diagram is clockwise from the bottom left to the bottom right, that is, from the data source to the data consumer. The front end of the MLDP is the data Aggregator system, denoted Aggregator in the diagram. It sits within the EPICS control system and acquires data from beamline instruments and diagnostics equipment attached to the control system (i.e., the data sources). As indicated in the diagram, the Aggregator system functions as the *data provider* to the Datastore. It acquires heterogeneous data from various data sources and coalesces them into manageable collections, or *data frames*, that can be ingested by the Datastore. Within the Aggregator, these data frames are realized as EPICS NTTable normative types which support transport of heterogeneous, time-series data over the pvAccess protocol.

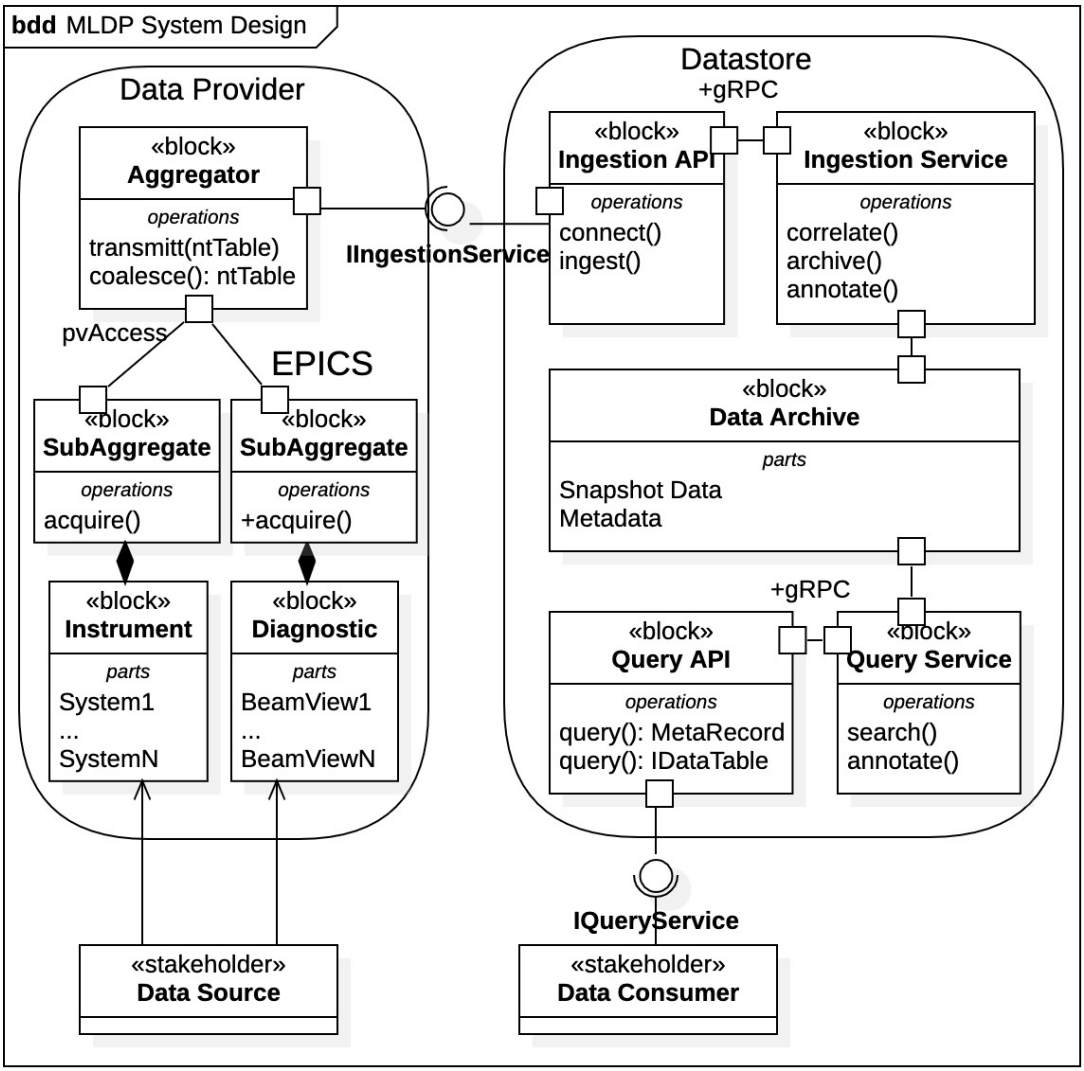


Figure 2: MLDP system design

Note that the Aggregator system is composed of both distributed components, labeled SubAggregate, and a centralized component, labeled Aggregator. The distributed components are dispersed within the EPICS control system, typically within proximity of the data sources. They perform the actual data acquisition from the beamline instruments and diagnostic equipment. Once acquired, data is sent to the central Aggregator where it is coalesced into composite NTTable objects for staging to the Datastore ingestion service. All communications with the Datastore component are performed by the central Aggregator which acts a single data provider. It is possible to simultaneous connect multiple Aggregator systems to a single Datastore system. Administration of the full Aggregator service would be the task of control systems engineers.

The Datastore system provides the data ingestion and archiving service, as well as the querying services for the MLDP. As indicated in Figure 2, it is a fully independent system and is hosted on a separate platform. It is also the primary point of interaction for data scientists and data science applications, the *data consumer* within the diagram. Communication with the Datastore ingestion and query services is made through Remote Procedure Call (RPCs). However, external connections to these services are made with well-defined APIs available in API libraries shown in the diagram. Thus, Datastore users are isolated from any underlying communications protocols and need only be concerned with the API operations within the libraries.

Referring to Figure 2, the ingestion side of the Datastore has two components, the ingestion API exposed to data providers and the ingestion service which manages the archiving of ingested data. The data archive contains both the snapshot data, and the metadata associated with the snapshot data. The query side of the Datastore also has two components, the query service which performs the requested search and query operations, and the query API facing the data consumers. The query API translates user data requests into appropriate search and query operations for the query service, then returns the requests in standard formats for data science and machine learning applications. Both the ingestion service and the query service have access to the data archive, and both may modify its contents.

Not shown in the diagram is the Web Application. It runs on a separate host platform and connects directly to the Datastore query service, that is, it does not require a query API library. Users may connect to the Web Application using a predetermined URL from a standard internet web browser.

2.3 DEPLOYMENT

A typical deployment scenario for the MLDP is shown in Figure 3. The data sources are instrument and diagnostic hardware systems connected to EPIC Input/Output Controllers (IOCs), shown in the top right-hand side of the diagram. Each hardware device within a facility typically has a unique IOCs controller, and IOC controllers are hosted on network servers. A server may host one or more IOC controllers depending upon the resource requirements of the hardware. The diagram includes the use of Field Programmable Gate Array (FPGA) instrument controllers to clarify the high-speed capabilities for data acquisition and control. The asterisk (*) decorations given to the hardware and IOC connections are meant to indicate that multiple servers and IOCS are available throughout the beamline. The distributed Aggregator components of the MLDP are hosted within IOC servers connected to hardware (not shown). As indicated in the diagram, the Central Aggregator is hosted on a single IOC of particular importance, this is the point of contact for the Datastore ingestion service. The Datastore system is hosted on a separate server platform. It ingests data from the Central Aggregator through the IIngestionService interface exposed by an

API library. The InfluxDB and MongoDB database systems used by the Datastore are also hosted on the Datastore platform. However, the multiple servers seen in the diagram are intended to demonstrate that the Datastore hosting platform may contain multiple servers (i.e., for performance) and, consequently, these databases can be deployed on independent platforms.

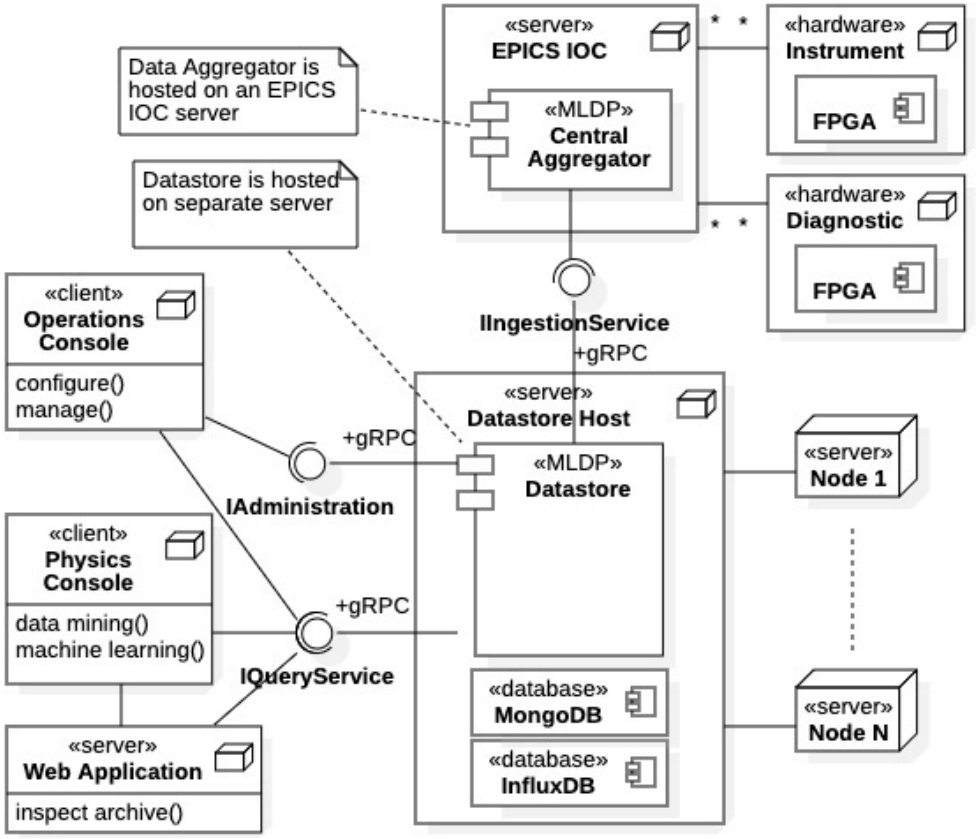


Figure 3: typical MLDP deployment

The left-hand side of Figure 3 shows a set of data consumers: the operations console, the physics console, and the Web Application. The operations console is a client platform for hosting Datastore administration utilities. The physics console is a client platform for data-centric applications utilizing the Datastore archive and query services. The Web Application is a hybrid platform, it is both a client to the Datastore and a service to the physics console. The clients connect to the Datastore using the `IAdministration` and `IQueryServce` interfaces exposed by the appropriate API libraries. The `IAdministration` interface offers methods to manage the Datastore operations and the Datastore archive, while the `IQueryService` interface provides the archive search and query operations required for data science and machine learning applications.

Clearly other deployment configurations are possible for data consumers and would depend greatly on the nature of the facility utilizing the MLDP and the desired applications. The important point to note is that the Aggregator and the Datastore components are deployed separately, hosted on separate nodes. The Aggregator system is deployed within EPICS IOCs for data acquisition and transport. It then supplies collected data to the Datastore through the `IIngestionService` API for transporting coalesced data frames. The Datastore has a separate host platform that supports all internal communications through gRPC and well-defined API interfaces for external

communications. The supporting InfluxDB and MongoDB databases may be deployed on the same server host or separate hosts. The Web Application is also deployed on a separate host server.

2.4 CODE REPOSITORIES

Before continuing to the detailed descriptions of the individual MLDP components, we provide the locations of the code repository for the MLDP project. Listed are all the major projects and systems comprising the current MLDP implementation, along with some implementation specifics. This repository is available online for inspection.

All source code repositories are hosted on GitHub (24). The Aggregator is contained in the single repository https://github.com/mdavidsaver/bsas/tree/russo-vfield. It is implemented in the C++ language and built using the Unix Make utility. It also uses some Python tools for testing. The Web Application is contained in the repository https://github.com/craigmcchesney/datastore-web-app. It is implemented in Java and JavaScript and utilizes the React JavaScript library. The Datastore system is in multiple repositories, all are located under the Osprey DCS organization (https://github.com/osprey-dcs) branch; see Subsection 4.2 for details. It is implemented primarily in the Java language utilizing the third-party components previously described. Builds are managed by the Maven Java project utility (23).

PART 2: Design and Operations

3. THE AGGREGATOR

The Aggregator system is the front end of the Machine Learning Data Platform. Its function is to acquire machine data from hardware and software systems along the beamline, then coalesce it into manageable collections for transport to the Datastore ingestion service. Thus, it has several requirements that are immediately apparent:

1) Acquire thousands of simultaneous signals from heterogeneous data sources.
2) Correlate the signals into time-series data channels.
3) Aggregate the time-series data into composite structures at a central location.
4) Transport the resulting framed data to the Datastore for archiving.

Thus, the Aggregator system begins with low-level hardware acquisition and ends with high-level data management and transport. As data moves through the transport stack the Aggregator adds organization and form. The final output of the Aggregator system is a stream of time-correlated, time-series, heterogeneous data frames recognized by the Datastore ingestion service.

The Aggregator is implemented in C++ and builds using the standard Unix Make utility. It requires only those resources already available in EPICS, that is, no third-party components or resources are used. Installation of the Aggregator requires expertise with the EPICS control system (20), as well as site-specific facility knowledge.

For the sake of context, a brief description of the EPICS environment is provided below. A cursory understanding of the EPICS environment clarifies the Aggregator design and operation.

3.1 THE EPICS ENVIRONMENT

EPICS is a sophisticated control system designed for operation and data acquisition of large facilities with thousands of control and observations points (21). The Aggregator is hosted within the EPICS control system. It has distributed systems that function as services within EPICS, and a centralized system operating as a client of EPICS.

The EPICS control system is implemented in layers, starting from the hardware systems at the bottom layer to Supervisory Control And Data Acquisition (SCADA) client applications at the top layer. Control signals and acquired data move through front-end computers connected to the hardware layer, to client applications distributed over computer networks. There are front-end computers connected to hardware which are called Input/Output Controllers (IOCs) in EPICS terminology. The interfaces between hardware systems and the EPICS IOCs are special programs called EPICS *device drivers*, which recognize signals from specific hardware devices. The device driver makes hardware signals available to clients of EPICS as *Process Variables* (or "PVs") within the EPICS environment. EPICS has multiple proprietary network communication protocols, including *ChannelAccess*, *pvAccess*, and *pvData*, which transport control signals and acquired data as process variables (22). EPICS communications are bi-directional, that is process variables can be both *control signals* and *acquired data*. Control signals originate from high-level EPICS clients and are transported to low-level hardware. Acquired data originates from low-level diagnostic hardware and are transported to high-level EPICS client applications. In either case, front-end IOCs managing the device drivers provide access to all hardware systems within the

facility. Thus, EPICS clients can control and monitor all available hardware as a part of the operations of an accelerator facility.

The data Aggregator system is one such client of the control system; however, it is also more than a standard EPICS client. It also contains distributed components within the EPICS environment that function as data acquisition services. A single, centralized Aggregator component then operates as a specialized client of the distributed services. This operation is described below.

3.2 OPERATION

The operation of the Aggregator system is illustrated with help of the example shown in Figure 4. The figure illustrates a situation where data is acquired from multiple hardware devices at the top and is systematically collected and processed to a central location at the bottom center. Each block within the figure represents a separate EPICS IOC server. The Aggregator has local subsystems deployed within the IOCs at the top, collecting and managing data, and a central system within the bottom IOC which performs the final processing and transport. Thus, the full Aggregator system is distributed throughout the EPICS control system, being hosted on multiple EPICS front-end IOC computers at the top of the figure, and on a single IOC at the bottom.

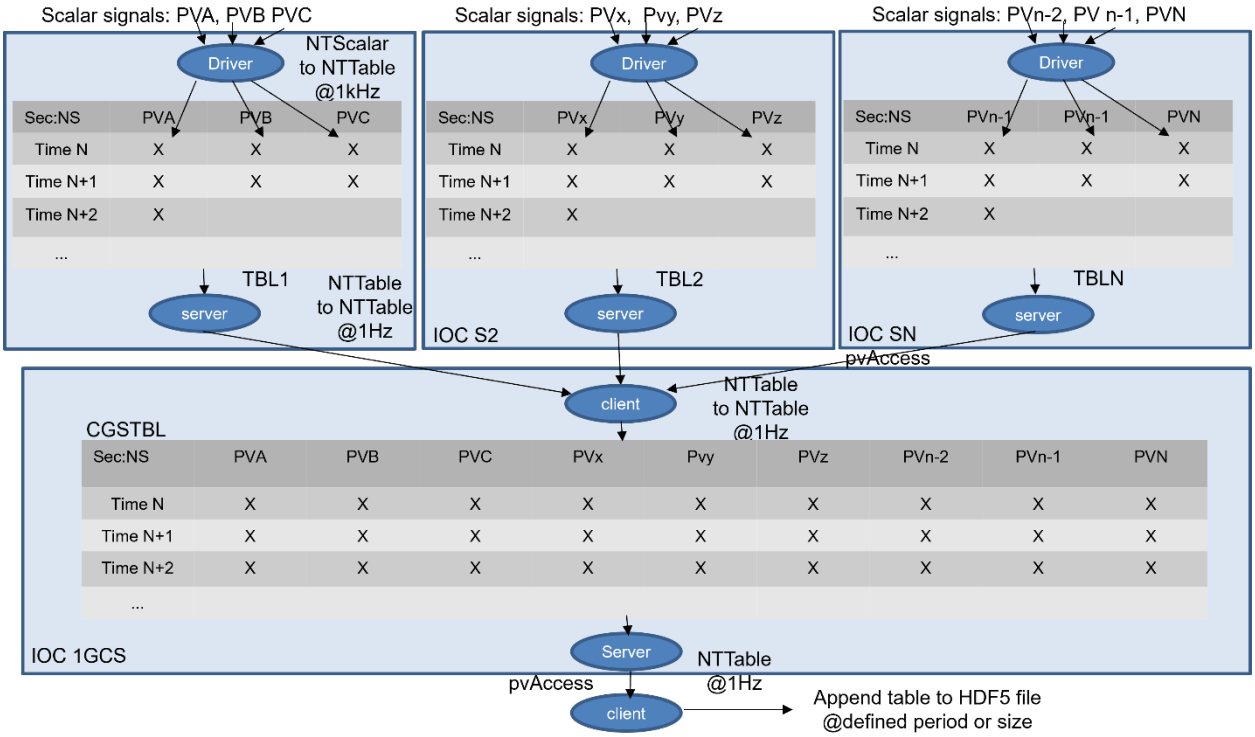


Figure 4: example of the aggregation process

Referring to the example in Figure 4, scalar values are sampled at 1 kHz from multiple hardware data sources and acquired by the EPICS device drivers. The scalar values are then collected by the front-end Aggregator components deployed within the IOCs hosting the device drivers. Note that the scalar data is collected into tables realized as EPICS NTTable objects, part of the EPICS normative types and supported by the pvAccess transport protocol (21) (24). The figure is intended to demonstrate this operation for an arbitrary number of separate IOC servers S_1, S_2, … each connected to a hardware device which supplies multiple signals. The total number of signals in

the example is N, labeled PVA, PVB, PVC, PV_{N-n}, …, PV_N. Note that for a large facility the value for N could be in the thousands. The first two IOCs, S_1 and S_2, have 3 input signals each, {PVA, PVB, PVC} and {PVx, PVy, PVz}, respectively. The final IOC at the top right has the last 3 input signals PV_{N-2}, PV_{N-1}, PV_N. The important point is that there exists multiple front end IOCs each producing NTTables containing data from a subset of the N hardware signals.

The top-half of Figure 4 demonstrates that the multiple front-end Aggregator components act as servers to the central Aggregator system, shown at the bottom of the figure. The data tables produced by the front-end components, containing potentially thousands of N signal values, are collected by the central Aggregator system acting as an EPICS client utilizing the pvAccess protocol. The incoming tables are coalesced into a single, composite data table containing an aligned time-series for all N hardware signals. The resultant table may be stored to an HDF5 data file or sent to the Datastore by way of the ingestion service. Metadata associated with the collected data, or the collection process itself, are also attached to the composite data tables.

Note that a continuous data stream is maintained by the full Aggregator system. Specifically, time-series data are collected into data tables near the hardware sources, coalesced into composite data tables, then sent to the Datastore at a continuous rate. For example, consider $N = 4{,}000$ signals all sampled at a common rate of 1 kHz. The signals are collected into multiple NTTables of 1,000 rows each. The tables are then sent to the central Aggregator where they are coalesced into a composite table of 4,000 columns (one for each signal) and 1,000 rows (one for each sample). If the composite tables are ingested by the Datastore at a rate of 1 frame per second, this yields an overall data transmission rate of 4,000 signals at 1 kHz, conforming to stated project goals.

3.3 ARCHITECTURE

The Aggregator architecture is shown in Figure 5, this architecture realizes the operational description given above. The beamline hardware and diagnostic equipment are shown on the left side of the diagram. Field Programmable Gate Array (FPGA) controllers are used as hardware interfaces signifying the high-speed aspect of acquisition. The hardware is connected to front-end IOC servers through the EPICS device drivers, where they are made accessible to the entire EPICS control system. Components of the Aggregator system are deployed throughout the control system and are identified with the "MDLP" stereotype within the diagram,.

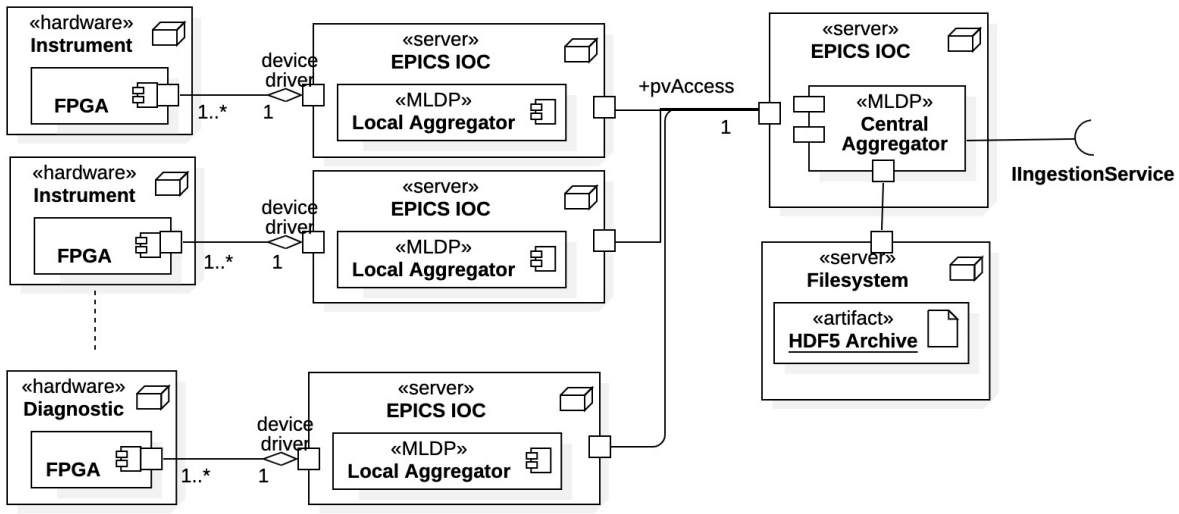


Figure 5: Aggregator architecture

The Aggregator architecture has both distributed components and a centralized component. The distributed front-end of the aggregation services, labeled Local Aggregator, sit in IOCs spread across the facility, preferably in proximity to the hardware acquisition system (as shown in the diagram). The centralized component, labeled Central Aggregator, can be deployed almost anywhere within the control system; however, it must have access to the Datastore ingestion service (i.e., the IIngestionService interface in the diagram). It acts as an EPICS pvAccess client, receiving and coalescing the tabular data produced by the distributed front-end components. As indicated in the figure, the Central Aggregator has the capability to store the composite table data in HDF5 files for inspection (a useful feature for development and testing) or later transport to the Datastore. However, the intent is to immediately transport acquired data to the Datastore archive through its ingestion service via the IIngestionService API interface.

Both the distributed front-end components and centralized component of the Aggregator are configured using the EPICS real-time database system. For the front-end system the database contains the signals to be acquired and the method of acquisition. It also defines the PVs for transporting the local data tables to the central Aggregator. The Central Aggregator real-time database identifies the sub-tables to be collected, the collection rates, and the format of the composite table.

4. THE DATASTORE

This chapter is a self-contained description of the Datastore system within the Machine Learning Data Platform. The Datastore is a standalone service and may be used as such; in this regard the Aggregator system is one of many possible data providers. The design and operation of the Datastore is explained in detail as sustained development efforts are expected here. Additionally, expanded descriptions of the APIs and the query operations are provided since the Datastore is the primary interaction point of data scientists with the Machine Learning Data Platform.

4.1 OVERVIEW

The Datastore system is responsible for data archiving, archive management, and all query operations of the Machine Learning Data Platform. Unlike the Aggregator, it is a fully independent system hosted on a separate platform of one or more server nodes. Although it was originally designed for use in EPICS-based control systems, EPICS is no longer a requirement. It has its own communications protocol and Applications Programming Interfaces (APIs) independent of EPICS. Thus, the Datastore may be deployed at any facility so long as the data providers conform to the ingestion APIs described in Subsection 4.7.

The Datastore is implemented almost exclusively in the Java programming language. The only exception being the "proto" meta-language for Remote Procedure Call (RPC) definitions. Specifically, remote procedure calls and messages are defined in this meta-language then used to generate Java source code in the *datastore-grpc* project (described below).

The Datastore implementation contains a core of services that manage the data archive, and an additional set of API libraries for external communications and for Datastore management. The core implementation utilizes the Spring and Spring Boot Java application frameworks. The core also requires two database services, MongoDB and InfluxDB, both must be available on the host platform network. All communications to the Datastore core are realized through the gRPC framework using Google's Protocol Buffers technology (17) (18). Internal Datastore communications between the core services and the MongoDB and InfluxDB databases occur via network sockets supported by the database APIs.

4.2 CODE REPOSITORIES

The Datastore code base is composed of multiple projects contained in multiple code repositories, all hosted on GitHub (26). The structure of the Datastore project within the repository system is shown in Figure 6. The central location is the Osprey DCS organization common branch https://github.com/org/osprey-dcs, as shown in the figure. Seven principal projects are identified in the figure along with their hierarchical relationships and the services and/or functions they provide (projects are identified with the <<project>> stereotype).

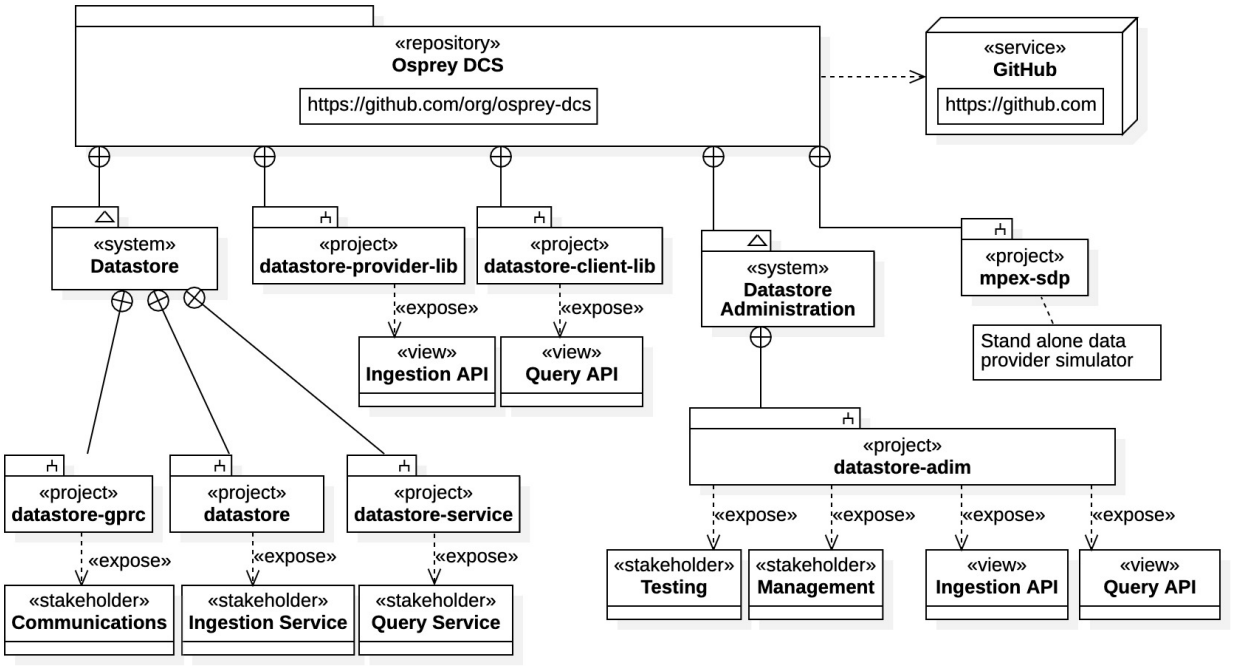


Figure 6: Datastore repository and project structure

The diagram identifies two high-level models (i.e., <<system>> stereotypes), *Datastore* and *Datastore Administration*. The *Datastore* system contains all internal operations of the Datastore core service and is composed of three projects, *datastore-grpc*, *datastore*, and *datastore-service*. These projects realize the communications protocol, the data ingestion service, and the data query service, respectively. The other high-level model, *Datastore Administration*, is composed of a single project, *datastore-admin*, and realizes the administrative requirements for Datastore operations, including connections services and testing, data integrity testing, performance testing, and data management. The project also contains applications programming interfaces (APIs) for both ingestion and query services, as APIs are required for management and testing. These are narrow APIs supporting ingestion and query operations using *data frames* and *data tables*, respectively (described in Subsection 4.5). Moreover, the ingestion API within this project is that used by the Aggregator.

The projects *datastore-provider-lib* and *datastore-client-lib* are two standalone API libraries for user interaction with the Datastore ingestion service and the Datastore query service, respectively. They provide wide interfaces supporting more generalized ingestion and query operations, specifically, operations based upon Java native types and RPC protocol, respectively. Note that the *datastore-provider-lib* ingestion API is offered for data providers other than the Aggregator, which may require a more generalized interface. Two accompanying projects, not shown in the

diagram but also belonging to the Datastore repository collection, are *datastore-provider-spring-boot-starter* and *datastore-client-spring-boot-starter*. These are two standalone service applications based upon the Spring Boot (14) framework supporting the *datastore-provider-lib* and *datastore-client-lib* APIs, respectively.

Finally, the project *mpex-spd* is a standalone application that simulates data produced by the MPEX facility at Oak Ridge National Laboratory (ORNL) (12). This application creates simulated data which can be ingested by the Datastore through the *datastore-provider* API. The *datastore-admin* project contains a scenarios-based data simulator, which can simulate the Aggregator system. The scenarios simulator sends data to the Datastore ingestion service using the *datastore-admin* ingestion API. Both data simulators may be configured for a variety of data-production and ingestion-loading conditions, each in their respective domain. Both simulators were used for Datastore performance testing.

The *datastore-grpc* repository is fundamental to all other Datastore projects. It is required to generate all communications protocols across the core system and is based upon the gRPC remote procedure call framework (17). The protocol is defined using Google's Protocol Buffers proto meta-source language and compiled using the *protobuf* compiler (18). The proto source is compiled into Java source code and subsequently built into Java bytecode by the Maven utility. Note that proto source code may be compiled into a variety of programming languages, such as JavaScript or Python. Because this project is a dependency of all other Datastore projects, it should be built first then installed on any host platform.

The two projects *datastore* and *datastore-service*, once compiled, run as separate services on the host platform. They realize the ingestion service and the query service, respectively. They are currently configured to build into shaded Java archives ("fat jars") to be launched using the Java virtual machine. Henceforth they monitor pre-defined host ports for connection requests. These two services maintain the Datastore archive and comprise the core operation of the Datastore. These core services require access to the InfluxDB and MongoDB database systems where configuration is managed through project properties files.

All projects contain the configuration file *application.yml* in their *src/main/resources* directory. These are application properties files required by their respective projects for site-specific builds. They contain property values for proper building, deployment, and service connections within the host environment. Default values are available for most properties but, typically, some user-specific and/or site-specific additions are required for proper project configuration (e.g., user credentials for databases, machine locations for file repositories, URLs and port numbers for service connections, etc.). All projects also support property value assignment through host environment variables or Java virtual machine command-line options.

4.3 CORE ARCHITECTURE

The Datastore core is composed of three primary components, the data archive, the ingestion service, and the query service. The basic architecture is depicted graphically in the component diagram of Figure 7 where the three components are labeled Data Archive, Ingestion Service, and Query Service, respectively. Also shown in the figure is that external interaction with the Datastore core is facilitated through independent API libraries, one exposing the ingestion service and one exposing the query service. Recall from Subsection 4.2 that there exist multiple API libraries

available for both the Datastore ingestion and query interactions. (All API libraries utilize the gRPC connection library *datastore-grpc* for cross-platform communications.) Regarding the figure, the libraries `Ingestion API` and `Query API` and their respective API interfaces `IIngestionService` and `IQueryService` are simply abstractions for the available concrete API libraries and interfaces. Specifics for some API libraries are covered in Subsection 4.7.

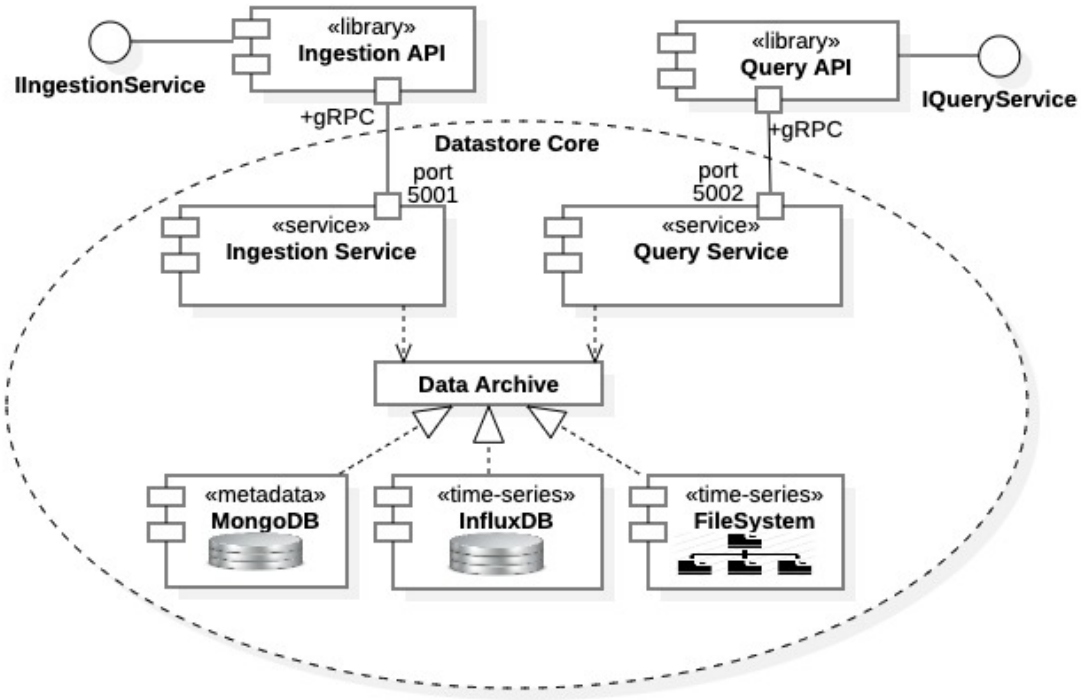


Figure 7: Datastore architecture, components, and relationships

Referring to Figure 7 the data archive is realized by an additional three sub-components, a MongoDB NoSQL database, an InfluxDB time-series database, and the host file system (i.e., system files located on the host platform). As indicated in the figure the archive metadata is maintained in the MongoDB database, while the time-series data is maintained in both the InfluxDB database and the file system. For reasons of performance and manageability, raw data and images are stored as system files rather than data blocks within the InfluxDB database. The archive itself is managed by the two core services, the Ingestion Service, and the Query Service. Both these components run as independent services on the host platform, and in the default configuration, they are attached to ports 5001 and 5002 of the host server, respectively. Again, external communications with these services are facilitated through the gRPC remote procedure call framework.

The Datastore core is capable of operating either synchronously or asynchronously through the gRPC connections, for both ingestion and query. Synchronous communications are blocking communications; remote procedure calls are established and not released until all data is exchanged. Asynchronous communications are non-blocking; data exchange is initiated with an exchange of gRPC interfaces, then remote procedure calls return before data exchange completes, or potentially even begins. Here interactions between the calling client and the service are realized as data streams that must be properly managed to maintain data stream integrity. The asynchronous data stream must be managed in both the Datastore core service and the external

client API library. The advantage of synchronous communication is that it is simple and avoids state and connection errors. However, it is slower and limited in the size of data that may be exchanged. Asynchronous communications are faster and essentially unlimited in size but runs the risk of overloading and data corruption through protocol or network errors.

Again, as shown in Figure 7, external communications to the Datastore core are established with API libraries. There exist two separate libraries for both the ingestion service and to the query service. Each client library supports both synchronous and asynchronous communication. The API libraries offer well-defined interfaces for interacting with the Datastore and isolate users from the gRPC communications protocol used by the core (although the gRPC library *datastore-grpc* must be installed in the API platform).

A Datastore host platform must have the InfluxDB and MongoDB database services pre-installed. These databases may be hosted on separate servers, but they must be accessible within the host network. Moreover, the databases must be properly administered to provide access to the Datastore through user credentials. Specifically, the Datastore ingestion and query services must have user credentials allowing read/write access to the InfluxDB and MongoDB databases. The Datastore administration library, covered in Subsection 4.7.4, must have administrative access to both databases. Typically, the credentialling process is facilitated through the *application.yml* configuration file, or host platform environment variables.

4.4 COMMUNICATIONS

Since the Datastore has a unique communications protocol, we describe the system in some detail. Developers using API libraries exclusively will generally be isolated from the details of this protocol (e.g., data scientists). However, the *datastore-client-lib* query API library has low-level features exposing some Datastore communications elements. Also, Datastore developers should be familiar with the paradigm to make engineering and design decisions.

Client communication with the Datastore services, both ingestion and query, are managed through Remote Procedure Call (RPC). The communications protocol is defined entirely within the *datastore-grpc* project. Defined are the complete set of messages and types that may be exchanged, and the interfaces by which they are exchanged (e.g., the remote procedure calls). The project is based upon the gRPC framework using the Protocol Buffers technology.

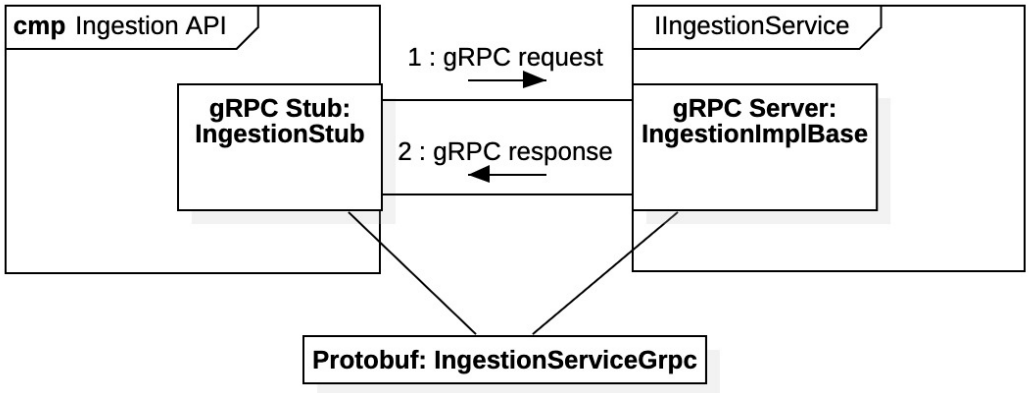


Figure 8: communications with Protocol Buffers

A basic illustration of the gRPC Protocol Buffers framework is shown in Figure 8. The figure depicts the case for the ingestion side of the Datastore, specifically, for communications between an ingestion API client library on the left side and the Datastore ingestion service on the right. A more expanded view, including implementation details, is shown in Figure 9. Situations for the Datastore query service are directly analogous. Referring again to Figure 8, Protocol Buffers provides an *implementation base* for gRPC services to communicate with clients, labeled `IngestionImplBase` in the diagram. The implementation base acts as an internal gRPC server providing hooks for the ingestion service to manage all incoming and outgoing messages. Clients of the ingestion service obtain a gRPC communications "stub", labeled `IngestionStub` in the diagram, which exposes all remote procedure calls and messages supported by the service. Both the gRPC implementation base and the gRPC communications stub are supplied by a common resource `IngestionServiceGrpc`, generated by the Protocol Buffer framework. As shown in the diagram, client requests are passed to the stub, transmitted through the gRPC framework across the network, then arrive at the implementation base where the service is notified, and the message is recovered. The service response follows a complementary path where it is delivered to the stub and recovered by the client as a remote procedure call result.

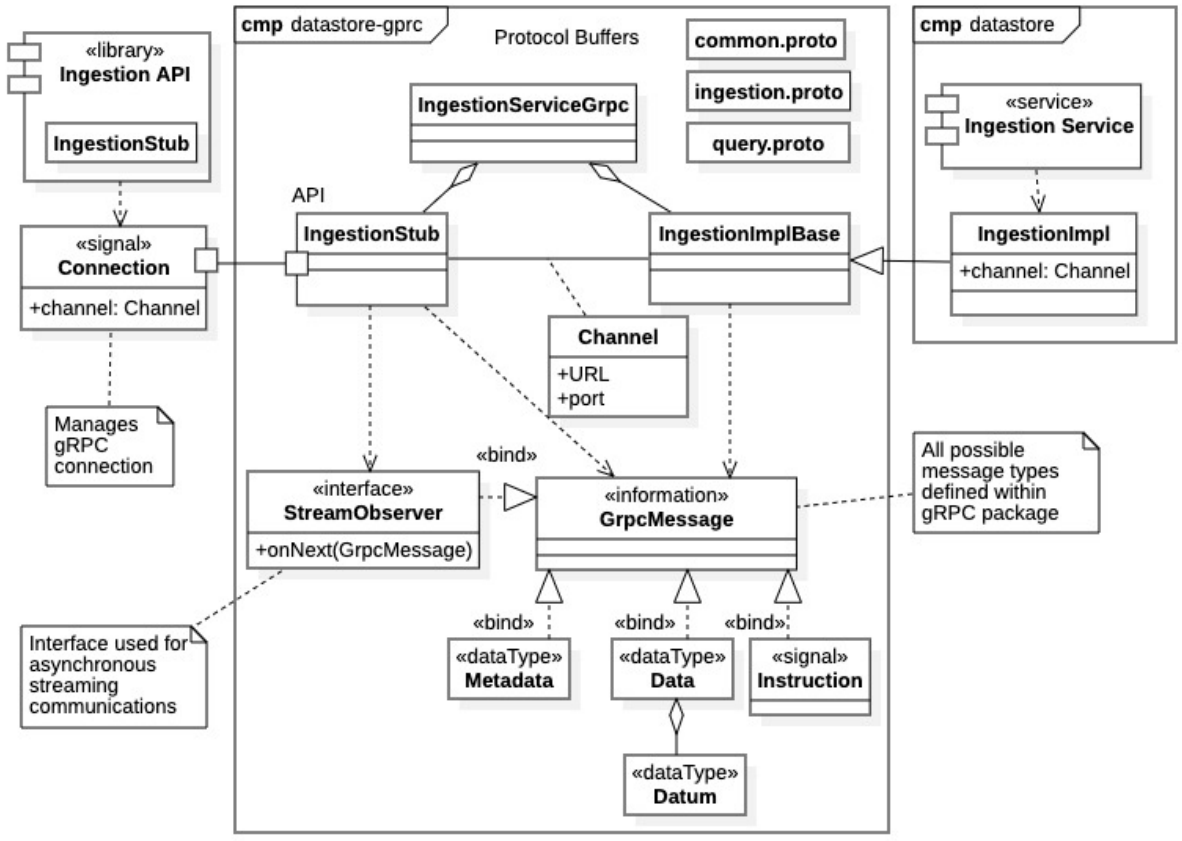


Figure 9: Datastore communications protocol

The graphic in Figure 9 illustrates how the *datastore-grpc* package library implements the gRPC infrastructure explicitly for the Datastore service. The figure specifically depicts the Datastore ingestion service, the query service implementation is directly analogous. The entities within the package *datastore-grpc* are explicitly identified within the center box of the diagram, along with their relationships. The `IngestionServiceGrpc` resource, along with its ingestion stub and

implementation bases components, are all elements of the Protocol Buffers technology identified in Figure 8. The external Ingestion API library on the left-hand side utilizes the ingestion stub through a Connection object. The external Datastore Ingestion Service, on the right-hand side, directly inherits from the gRPC implementation base using class IngestionImpl. A subset of available gRPC messages is also shown in the diagram, all of which are bound to the generic type GrpcMessage. These message types include data, metadata, and instructions. The diagram also formally indicates that the internally defined gRPC message types (i.e., through the GrpcMessage binding) are the only ones recognized by the communications stub and the implementation base.

The remote procedure calls supported by the ingestion stub and the implementation base, and their respective message types, are all defined in the "proto" meta programming language of the Protocol Buffers technology. Three source files within the project, common.proto, ingestion.proto, and query.proto, define the entire protocol, specifically, the common resources, the ingestion service protocol, and the query service protocol, respectively. These are the only source files within the *datastore-grpc* project. Instructions for the *protobuf* compiler are found in the Maven project file *pom.xml*. When building the project, Maven first instructs the protobuf compiler to create Java source code based upon the Protocol Buffers framework and the proto source protocol descriptions. The Java source code is then, in turn, compiled into Java bytecode and packaged into a Java archive file (i.e., a "jar" file). The jar file contains the entire gRPC communications library and should be installed in the API client platform as well as the Datastore host platform. Note that once built, the *datastore-grpc* project will contain all the Java source files for inspection.

Referring again to Figure 9, after the project build the framework contains the common resource IngestionServiceGrpc which provides both a communications stub for external communications and an implementation base used as a hook for the ingestion service to send and receive messages. (An analogous situation is found for the query service side of the Datastore and the corresponding common resource QueryServiceGrpc.) As indicated on the left-hand side, an external client connection to the Datastore is made by requesting a communications channel Channel which is then used to create the ingestion stub from the IngestionServiceGrpc resource. The channel object is maintained within a Connection utility to explicitly release gRPC resources once communications are terminated. Again, only the data and messages explicitly defined in the *datastore-grpc* package are recognized.

Both the synchronous and asynchronous communication protocols are defined in the *datastore-grpc* package. Both communications are bi-directional and often use the same gRPC messages. However, the implementations are quite different. Synchronous communication is essentially standard RPC message passing where procedure calls are made directly on the communications stub. Asynchronous communications consist of *data streams* that are implemented by exchanging pre-defined gRPC interfaces within the Protocol Buffers framework. These interfaces establish the data stream using operations with gRPC message arguments defined in the proto source files. The interface StreamObserver identified in Figure 9 is the common interface used by asynchronous gRPC clients to both send and receive messages through a data stream. It is a generic interface where specific gRPC message types are identified as template parameters. For example, the interface StreamObserver<Data> would be used to pass arguments of type Data.

As an example of asynchronous communications, consider the ingestion scenario where a client offers a large data set to the Datastore ingestion service as a stream of Data objects. In contrast

with a synchronous remote procedure call where a concrete Data object is offered; an asynchronous remote procedure involves the exchange of interfaces. In this case, the client requests an interface of type StreamObserver<Data> (or a subtype of this interface) from the ingestion stub. To do so it first passes the interface StreamObserver<Instruction> to the ingestion service. This interface allows the ingestion service to send the client instructions regarding the open data stream. It is the client's responsibility to provide a StreamObserver<Instruction> implementation that properly manages the stream of Data message to the Datastore ingestion service. The client is then provided with an interface StreamObserver<Data> as the response to the RPC method. This interface, implemented by the ingestion service, contains an operation onNext(Data) called by the client to offer data when instructed. For the full ingestion operation, the client initiates the data stream by invoking the appropriate RPC method (in the ingestion stub) which accepts the StreamObserver<Instruction> argument. The client is returned a StreamObserver<Data> interface as the RPC result. After receiving a ready acknowledgement, the client would then send Data objects as instructed by the ingestion service. Received instructions would typically be acknowledgements of ingested Data objects and ready conditions for continued streaming.

The current *datastore-grpc* project is configured to generate a Java-based implementation of the Protocol Buffers framework (i.e., as specified in the Maven *pom.xml* project file). However, it is possible to generate the communications framework for other languages, particularly the Python language which is popular in machine learning applications. It is also worth noting that the communications framework was also implemented in JavaScript, which was used in the Web Application augmenting the MLDP.

4.5 DATA FRAMES, SNAPSHOTS, AND DATA TABLES

Fundamental to Datastore operations are the notations of *data frames*, *snapshots*, and *data tables*. Data frames and snapshots are particular to ingestion operations while data tables are the results of query operations. Since these objects are essential to Datastore ingestion and query APIs covered next, we define them here in some detail. The basic relationships between data frames, snapshots, and data tables, along with their implementations, are illustrated in the diagram of Figure 10. We return to the diagram after first clarifying the basic concepts.

4.5.1 Data Frames

Data frames are units of snapshot data used for ingestion by the Datastore. They are collections of tabulated heterogeneous, time-series data analogous to the EPICS NTTable normative type familiar to EPICS users. (In fact the Java implementation DataFrame directly constructs from an NTTable argument.) Like the NTTable, a data frame may contain columns of heterogeneous data from different data sources. The commonality being that rows of the data frame are all correlated by the same time instant; that is, they all have the same timestamp. Although data within a data frame may originate from disparate sources and may have contrasting formats, they are all time correlated. The Datastore can process and archive a variety of differing data types within a single data frame, including scalars, numeric arrays, data structures, and images.

The ingestion API library within the *datastore-admin* project directly supports the data frame as a unit of data ingestion. In fact, this is a narrow interface accepting only data frames as a source of ingestion, consistent with the Aggregator as a data provider. The *datastore-provider-lib* ingestion API library, exposing a much wider ingestion interface, and accepts data in the form of Java native

types. Data frames may be broken into component columns and timestamps then sent separately. This library is offered to support data providers requiring more flexible methods of ingestion.

4.5.2 Snapshots

A snapshot is an aggregation of related data frames, typically from the same data provider. Snapshots are supported within the Datastore archive and the core operations, whereas data frames are only supported at the API level. This condition effected the term "snapshot data" used to describe the time-series data archived within the Datastore.

A "snapshot" is meant to encapsulate data received by a data provider over an interval of time (e.g., a "snapshot of data"). The data frames within a snapshot all originate from the same data provider. The provenance of a snapshot is preserved within the Datastore archive, however, individual data frames within a snapshot are not. The API library recognizes data frames and uses them to creates snapshots according to the method of ingestion and the behavior of the client data provider. As to be discussed in Subsection 4.6.4, the exact nature of the situation is determined by the size of the data frames, which communication method is used (synchronous or asynchronous), and the data provider actions.

4.5.3 Data Tables

A data table contains the results of a Datastore query service request. Like data frames, data tables contain time-correlated sets of heterogeneous data in tabular form. The data table is analogous to the DataFrame object in the Python Pandas library (25), familiar to many data scientists and machine learning applications developers. Both data frames and data tables inherit from a common base class sharing many of the same properties and operations, as seen in Figure 10. However, data frames differ in that they may contain metadata, data tables contain only snapshot data. Data tables contain additional *operations* particular to query requests, such as loading completed and loading error notifications. Both data tables and data frames consist of time-series, columnated heterogeneous data.

The query API library within the *datastore-admin* project directly supports the data table as the result of a query operation. This is a high-level library intended for data science applications requiring no familiarity with internal Datastore operations. The *datastore-client-lib* project contains a low-level query API returning results in their native gRPC message format. This library is offered to developers requiring low-level access to Datastore operations.

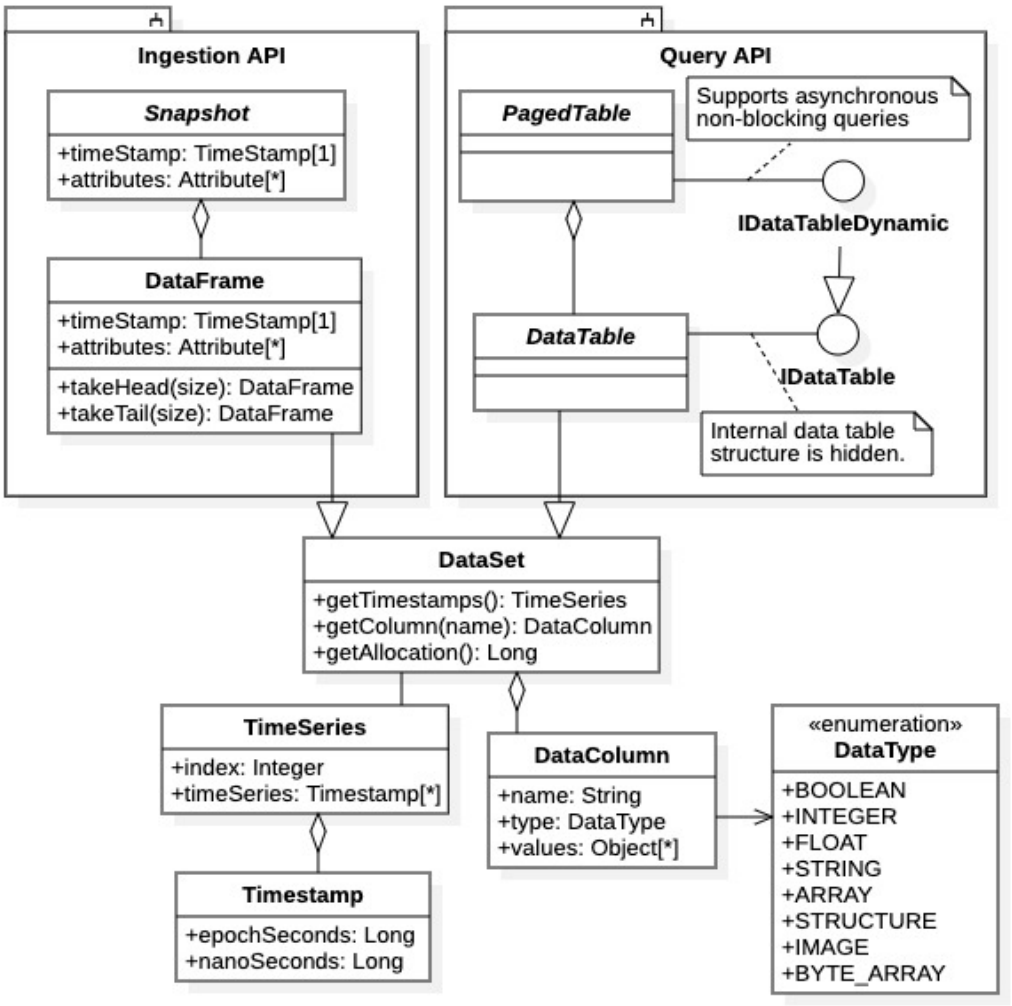


Figure 10: data frame, data tables, and snapshots

4.5.4 Relationships

Figure 10 illustrates the relationships between data frames, data tables, and snapshots. The specific implementation depicted is for the ingestion and query APIs in the *datastore-admin* project. In the figure we see that the data frame, represented by the `DataFrame` class, and the data table, represented by the *`DataTable`* class, both inherit from a common base `DataSet`. As seen in the diagram, data frames are used within the ingestion API library while data tables are used within the query API library. The common base `DataSet` consists of an indexed time series and an aggregation of data columns containing snapshot data from the various data sources. The snapshot data values may be of the various supported types. The common base class `DataSet` implements the common operations for data frames and data tables, such as retrieving timestamps and data values for desired rows or retrieving entire data columns. Snapshots, as represented by the *`Snapshot`* class in the diagram, are simply aggregations of data frames, with the addition of a timestamp ("trigger time") and possible user attributes. However, within the Datastore implementation *there is no formal snapshot type*, it is an abstraction (indicated with italicization in Figure 10). That is, a snapshot is only conceptual. Specifically, within the Datastore core implementation snapshots

are realized as archive operations, but they are not concrete objects. However, with the appropriate query request, it is possible to recover a DataTable instance containing all the snapshot data within a snapshot.

Within the query API shown in Figure 10 it is seen that the data tables themselves are not explicitly exposed. Alternately, an interface, IDataTable, is provided as the result of a query operation. Thus, the internal implementation of a data table is hidden. This situation is desirable since there are multiple data table implementations available, the specific implementation returned by the query API depends upon the type of query and the type of communications used (synchronous or asynchronous). Figure 10 specifically identifies a *paged data table,* which is typically returned from a large, asynchronous request. Paged data tables are essentially ordered collections of sub-tables, each independently loaded from an asynchronous data stream. It is typically of no consequence to the user whether the table is paged or not, all the same operations should be available through the IDataTable interface. However, for the asynchronous case, the derived interface IDataTableDynamic offers additional notification operations for the user, for example when the table has completed loading or if an exception has occurred.

During ingestion operations data frames may require "binning" to accommodate gRPC message size limitations. There, source data frames are divided into subframes meeting size requirements before being transported to the ingestion service. Within the figure it is implied that data frames support data binning through the takeHead() and takeTail() operations, which detach a partition of pre-determined size for packaging into gRPC messages. The memory allocation required for each data frame is available in the base class operation getAllocation(). More information on data binning is covered in Subsection 4.6.4.

4.6 INGESTION OPERATION

We now begin to describe the basic operation of the Datastore using concepts previously covered. Here the operation is with regards to the ingestion transport stack, from the ingestion service to the data archive. We also provide a functional description of the data retrieval process in Subsection 4.6.5 to clarify this operation with respect to the archiving process. A detailed description of the query API are described in Subsection 4.7.3 and the query service operation in Subsection 4.8, these being the specific concern of data scientists and data science applications.

4.6.1 Overview

The Datastore ingestion service is designed to accept heterogeneous, time-series data in the form of scalars, numeric arrays, data structures, images, and raw data (byte arrays). Scalars themselves may be of various types, including Booleans, integers, floats, and character strings. Ingestion is performed in discrete *data frames*[3], where each frame is essentially a time-series table consisting of columns of the above data types. Each row of the data frame has a distinct timestamp indicating the time instant at which the column entries were acquired.

Ingestion by the Datastore can be performed synchronously or asynchronously. Synchronous ingestion is a blocking operation where data frames are sent individually, the operation does not release until the frame is transmitted, processed, and archived. Asynchronous ingestion is non-

[3] The gRPC message type is SnapshotData. For ease of discussion, here we equivocate this gRPC message and the "data frame" programming language structure.

blocking. Multiple data frames may be sent through a persistent data stream while the Datastore simultaneously processes and archives the incoming data frames. Thus, data transport and data processing are decoupled. In fact, the Datastore supports ingestion through multiple simultaneous data streams. Synchronous ingestion is simple and safe and is suitable for small data frames and slow data rates. Asynchronous ingestion is faster and essentially size unlimited; however, it runs the risk of overwhelming the Datastore processing capabilities if data sizes, and data rates, are not properly tuned. Asynchronous ingestion is also more prone to network and protocol errors and runs the risk of deadlock and race conditions if stream communications are improperly managed.

4.6.2 Data Transmission

Data may be transmitted to the Datastore both synchronously and asynchronously. In either case a gRPC connection to the Datastore is established and maintained while data frames are transmitted. Additionally, the Datastore supports multiple simultaneous connections for ingestion by either means. Once all data has been ingested, the data provider closes the connection releasing all gRPC resources. In synchronous transmission data frames are sent one at a time, blocking while the data is transported, processed, and archived. Acknowledgements are simply in the form of returned remote procedure calls. For asynchronous transmission data transport and data processing are decoupled. Data frames can be transported as fast as the Datastore will accept them through an open "data stream." Acknowledgements are handled through a gRPC "ready" signal obtained through the stream. Additionally, both the *datastore-admin* and *datastore-provider-lib* ingestion API libraries provide an ingestion data queue where incoming data is buffered before asynchronous transport. This buffering isolates the Datastore from provider fluctuations and ensures that data is transmitted at maximum rates during sustained ingestion (i.e., as fast as Datastore will accept it). Thus, the processing and archiving capabilities of the Datastore are the primary limitation for asynchronous ingestion, whereas transmission and data size are also factors in synchronous ingestion.

As covered in Subsection 4.4, synchronous communications are straightforward, they handle gRPC messages directly. Remote procedure calls are performed directly on the gRPC communications stubs and accept gRPC message types as arguments. The result is returned directly from the procedure call as a gRPC message of proper type. On the other hand, asynchronous streaming communications involved the exchange of gRPC interfaces, specifically, of type StreamObserver or a derived class. Data exchange with these interfaces is strictly dictated by the gRPC protocol; however, it is also dependent upon the expectations of the Datastore. Asynchronous clients of the Datastore services, specifically the API libraries, must adhere to both the gRPC mechanism and to the designed behavior of the Datastore services. Thus, it is crucial to define the operation of the Datastore core early in development, as all asynchronous transmission throughout the data stack reflects the core implementation. Both the gRPC communications protocols and the Datastore data transmission protocol (e.g., requests, acknowledgements, synchronizations) are managed through these gRPC interfaces. Asynchronous data streaming, although fast and unlimited in data capabilities, is more fragile.

For an explicit explanation of the asynchronous streaming process see the example provided in Subsection 4.4 concerning the asynchronous streaming of a large data set. The example directly describes the interaction between the ingestion API library and the ingestion services through the exchange of gRPC interfaces StreamObserver<Instruction> and StreamObserver<Data>.

4.6.3 Snapshots and UIDs

As discussed previously, a *snapshot* is a construct used by the Datastore to represent a collection of heterogeneous time-series data acquired within a given time range from a given data provider. This construct is fundamental to data ingestion and the Datastore archive. The Datastore organizes the archive according to snapshots, generating metadata regarding the data types, attributes, and data providers producing the snapshots.

Within the archive, each snapshot data set has a unique *snapshot ID*. This is an integer value used as a unique identifier (UID) for storage and query operations. Again, however, snapshots themselves are purely conceptual and are not concrete objects within the archive. Nevertheless, as a data set, a specific snapshot may be recovered as a data table resulting from a query operation requesting the snapshot UID.

The realization of a snapshot differs according to the process of ingestion. In the synchronous case a snapshot UID is assigned to every data frame sent to the Datastore. In this case the data frame and the snapshot are essentially equivalent. If data is synchronously transmitted, every data frame is assigned a snapshot UID, unless data binning is invoked (discussed below). In the case of data binning, a UID is assigned to each bin of the original data frame.

In the asynchronous case one snapshot UID is assigned to the entire data stream. To clarify consider the process of asynchronous data streaming. First an asynchronous data stream is opened by a data provider, say for example the Aggregator service. The Aggregator then streams all its available data using as many data frames as required, for as long as required (e.g., continuous acquisition). Once data ingestion is completed, the Aggregator closes the data stream and it is discarded. Upon data stream closure, all streamed data is saved as a single snapshot and assigned a single snapshot UID.

4.6.4 Data Binning

A feature provided in the *datastore-admin* ingestion API library is *data binning*. This feature is used to circumvent the gRPC message size limitations. Currently gRPC message sizes have a maximum memory allocation of approximately 4 Mbytes. With this restriction, data frames greater than maximum size are rejected by gRPC, potentially causing the ingestion service to fail altogether. (On the other hand, the query service truncates query requests larger than the maximum limit but does not fail.) The solution is to bin large data frames into an ordered collection of smaller frames before transmission, each binned frame meeting the size requirements.

Before a data frame is sent to the Datastore ingestion service, the binning process computes the frame's memory allocation. If a data frame is too large it is binned into smaller frames meeting the size requirements. However, this can be an expensive process depending upon the structure of the data frame (in particular, for wide frames with large column counts). For asynchronous streaming, performance is improved by simultaneously binning and streaming; that is, the data bins are sent as they are partitioned from the source frame. Thus, while the Datastore is processing a binned data frame the client API library is building and buffering subsequent bins, rather than waiting for a Datastore acknowledgement (the primary benefit in asynchronous communication).

4.6.5 Archiving and Retrieval

Consider the archiving of snapshot data by the Datastore and its subsequent retrieval. Within the standalone query API *datastore-provider-lib* project, snapshot data is ingested as low-level objects, or collections of low-level objects, as Java native types. This API is oriented toward data

providers requiring more flexible ingestion (e.g., other than the Aggregator). Within the *datastore-admin* project snapshot data is ingested in the form of data frames, of type `DataFrame`, as outlined in Subsection 4.5. The API library is designed specifically for ingestion of Aggregator output. However, it accepts data from any provider that can supply data in the `DataFrame` format. Additionally, the same project contains a query API library oriented towards data science applications. Thus, we focus on snapshot data archiving and retrieval as it is implemented there.

Recall that the term *snapshot data* refers to the time-series data within the Datastore archive (i.e., not metadata). Although ingested data is archived according to time intervals, the Datastore has broad query capabilities; it need not be requested as a "snapshot" of data. The situation is illustrated with the help of Figure 11.

Figure 11 illustrates an example where multiple data providers supply data in separate snapshots using separate, and potentially concurrent, data streams. However, the snapshot data is queried across all snapshots. In the figure we see that multiple data frames are ingested into single snapshots, and each frame contains time-series data from multiple data sources (e.g., EPICS PVs). In the left-hand side we see that snapshot `S1` is composed of the sequence of data frames `F1` through `Fn`. Each of the data frames contain a portion of the time-series data acquired from data sources named `pv1` through `pv5`. Likewise, snapshot `S2` is composed of the sequence of data frames `G1` through `Gm`, each containing a portion of the time-series data from data sources named `pv6` through `pv10`. Within the Datastore archive both snapshots are simply stored as time-series data, although all the relationships between snapshots `S1` and `S2` and data sources `pv1` through `pv10` are maintained.

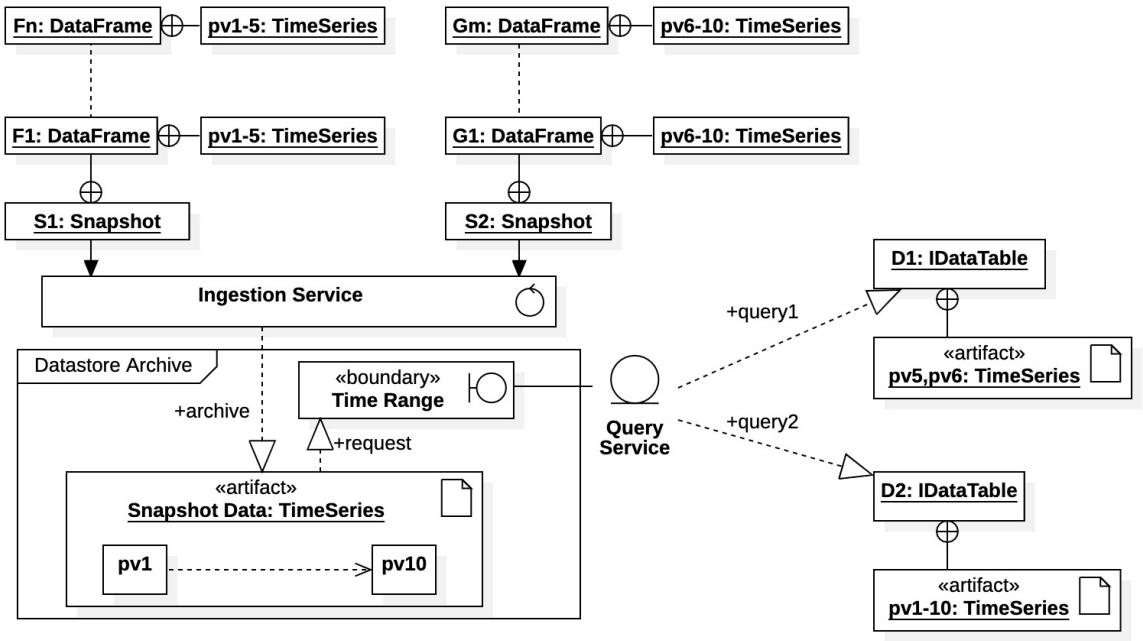


Figure 11: snapshot data archiving and snapshot data queries

The query service has access to the full archive, facilitating snapshot data requests using a spectrum of query parameters. The example shown in Figure 11 demonstrates query operations over time ranges and data sources. The results of two query requests are shown: `query1` produces a data table `D1` containing time-series data for data sources `pv5` and `pv6` over some unspecified (but bounded)

time duration, while `query2` produces a data table `D2` containing time-series data for data sources `pv1` through `pv10` over some unspecified (but bounded) time duration. In each case the snapshot data is originally sourced from both snapshots `S1` and `S2`. The point is to illustrate that data tables are general. Regardless of the ingestion configuration, data is available across data providers, snapshots, data sources, time intervals, and other attributes (including metadata).

4.6.6 Further Operation

Further operation is best described through the APIs seen by clients of the Datastore core. The next subsection provides an overview of the Datastore APIs and their supporting operations. The ingestion, query, and administration APIs are covered in detail.

4.7 APIs

Currently there are two parallel sets of Applications Programming Interfaces (APIs) available for the Datastore core, for both the ingestion service and the query service. The separate API libraries provide different perspectives to the Datastore archive. The projects *datastore-provider-lib* and *datastore-client-lib* contain standalone API libraries for the Datastore ingestion service and query service, respectively. These libraries expose wide communications interfaces suitable for general applications programming. The ingestion interface exposed within *datastore-provider-lib* supports data ingestion via Java native types, rather than explicit data frame objects. The query interface within *datastore-client-lib* supports data requests using a proprietary SQL-like language, the Datastore Query Language (DQL) covered in Subsection 4.8.5. Request results are returned as collections of low-level gRPC message types.

A second set of API libraries is offered in the *datastore-admin* project. The ingestion API is specific to the Aggregator accepting explicit data frame objects (i.e., of type `DataFrame`). The query API is orientated toward data science and machine-learning applications development where requests are created with a "request builder" utility and results are returned as data tables. Utilizing the data frame and data table constructs, the APIs hide many of the underlying operations of the Datastore, which are directly visible in the former API libraries. Since the intended focus of the Datastore is on Aggregator ingestion and data science applications, the latter interfaces are presented here.

4.7.1 Datastore Connection

The general procedure for connection to any Datastore API library is shown graphically in Figure 12. The situation applies both to an ingestion service API or a query service API, either of which is denoted generally as IServiceApi in the diagram. Each API library contains a connection factory, denoted ConnectionFactory, which supplies an appropriate API interface implementation when requested. A user requests the desired API simply by choosing the default service connection, or optionally specifying the network address for a particular Datastore deployment. The connection factory then initiates the required gRPC framework by creating the appropriate gRPC communications stub and establishing the network connections to the desired service within the Datastore core (i.e., either the ingestion service or the query service). An appropriate interface implementation (labeled ServiceApiImplementation) is created and fitted with the communications stub. The user is then returned the handle to the newly created interface object.

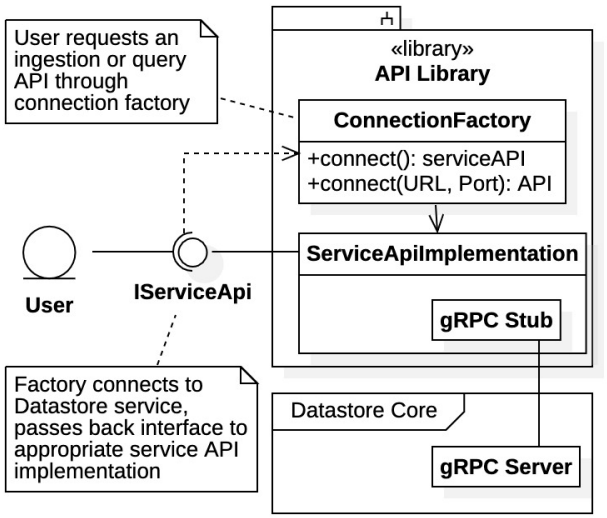


Figure 12: API interface connection

Note that the user sees only the exposed API interface IServiceApi and that multiple implementations may be available for the same interface. It is desirable to isolate the user from the actual interface implementations, the connection factory decides which implementation to deliver. The objective here is to establish concrete API interfaces early in development, preferably narrow ones allowing only a minimum of well-defined operations. This facilitates implementation upgrades later in the development process without affecting the API definition and, hence, any user applications.

The ingestion API interface is covered in Subsection 4.7.2 and the query API interface is covered in Subsection 4.7.3. These subsections discuss the use of the APIs and, consequently, illustrate user interaction with the respective Datastore services. Additionally, an administration API is available for Datastore management; its use and corresponding Datastore operations is covered in Subsection 4.7.4.

4.7.2 Ingestion Process and API

The ingestion API within the *datastore-admin* project is tailored to the Aggregator service. The library provides narrow API interfaces for data exchange based upon data frames. Ingestion operations for this API is shown in Figure 13. More broadly, the entire Datastore ingestion process is described with the help of the diagram.

The left-hand side of Figure 13 contains a generic data provider, while the right-hand side contains the ingestion API library and the Datastore ingestion service. As indicated in the diagram, the data provider is assumed to acquire data (i.e., from hardware data sources) then collect it into data frames for ingestion by the Datastore. The Aggregator system is the data provider *specific to the Machine Learning Data Platform*. However, all that is required of data providers using this API library is that they produce data frames; data providers are simply sources of data frames.

Appropriately, recall that one can create DataFrame objects directly from EPICS NTTable objects produced by the Aggregator; DataFrames and NTTables are essentially equivalent classes.

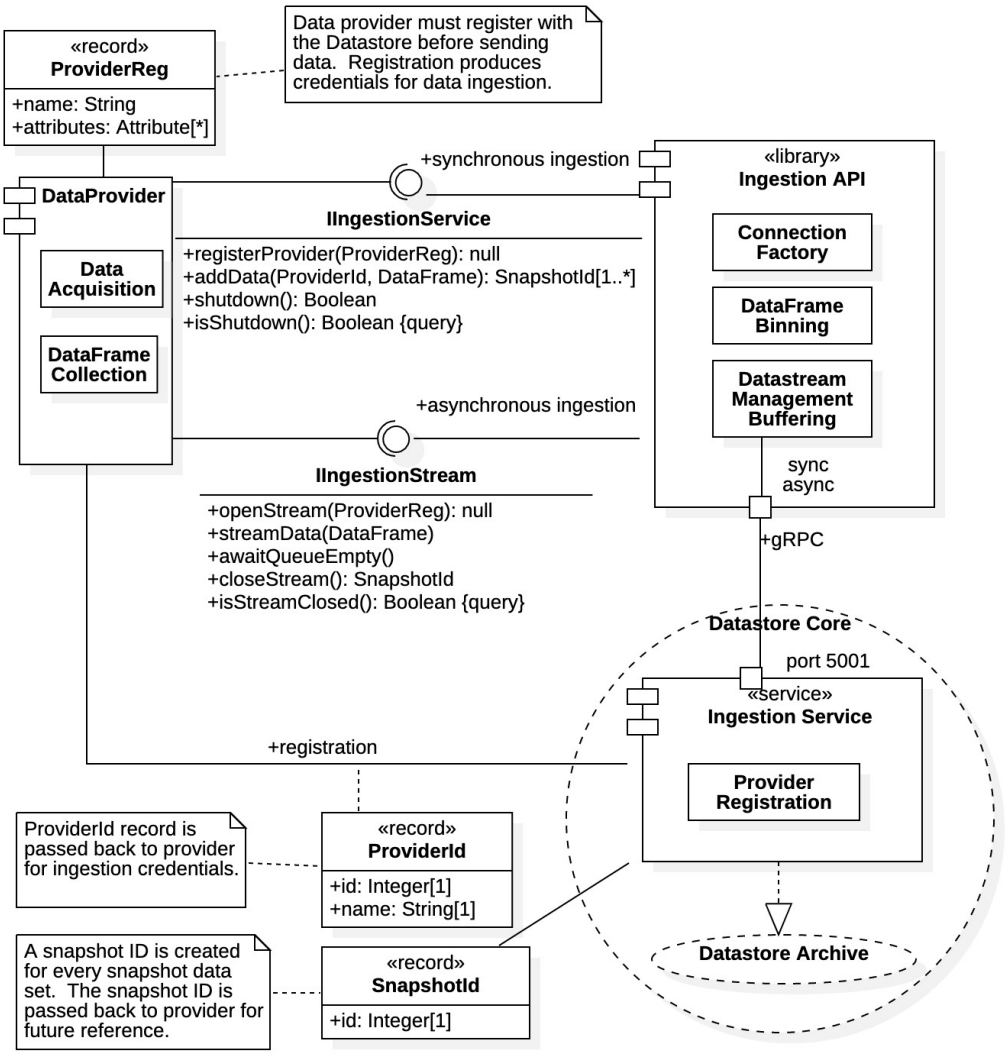


Figure 13: Datastore ingestion process and ingestion APIs

As seen in the right-hand side of Figure 13, the ingestion API library explicitly provides a connection factory, data frame binning, data stream management, and buffering services, as well as gRPC communications with the Datastore ingestion service. To initiate the ingestion process, the data provider requests an ingestion API to the Datastore via a connection factory with the process described in Subsection 4.7.1 and Figure 12. Namely, an ingestion API is requested from the ingestion connection factory (DsIngestionServiceFactory in the *datastore-admin* project) which makes all the appropriate connections with the ingestion service and passes back an active API interface implementation. The interface provided depends upon the connection request, either an IIngestionService interface for synchronous ingestion, or an IIngestionStream interface for asynchronous ingestion. Henceforth all the operations and activities shown in Figure 13 are available. As seen in the diagram, these interfaces are extremely narrow, only providing operations for provider registration, data frame ingestion, and shutdown operations.

Once the desired API is obtained, the first requirement of the data provider is registration. Referring to Figure 13 provider registration is a component of the ingestion service; the provider must register itself with the Datastore as a unique data provider. To do so it creates a ProviderReg record, populating it with the provider's unique name and any attributes it wishes to establish. Attributes can be assigned to the provider itself, or to incoming data (e.g., facility, location, subsystem, equipment type, experiment, etc.). The registration record is passed to the ingestion service through the ingestion API, after which the data provider receives a unique identifier (UID) in the form of a ProviderId record. If the provider has previously registered with the Datastore it will be returned the original registration UID (provider names are unique). For any ingestion process to commence, this UID record must first be established; all incoming snapshot data will be associated with the provider UID. Henceforth data ingestion proceeds through the acquired ingestion API.

As stated above, synchronous ingestion is performed using the IIngestionService interface while asynchronous ingestion is performed using the IIngestionStream interface. The operations within each interface demonstrate the difference between the two communication paradigms. For synchronous ingestion the data provider must first register with the registerProvider(ProviderReg) operation to obtain its UID. Then it may send as many data frames as desired using the addData(ProviderId, DataFrame) operation, but always identifying itself as the data provider with its UID record. This is a blocking operation and the addData() operation will not release until the data frame is fully transmitted, processed, and archived. Note also that at least one snapshot ID is returned with every data frame offered. Multiple snapshot UIDs may be returned if the data frame required binning, in that case one snapshot UID is returned for each data bin. Once the provider has finished transmitting all data frames the connection should be shut down using the shutdown() operation, then the interface discarded.

Asynchronous ingestion involves the management of a data stream between the provider and the ingestion service. Here the registration process is synonymous with data stream initiation. The provider opens the data stream by passing its registration record to the openStream(ProviderReg) operation. Henceforth the data stream remains open for frame ingestion until explicitly shut down with the closeStream() operation. The provider does not need to overtly identify itself afterwards since the gRPC connection to the ingestion service remains open. The provider may then send as many data frames as desired to the Datastore archive through the streamData(DataFrame) operation. The operation will block only if the data stream becomes backlogged. Normally operations will return immediately, as data frames are buffered within the ingestion API library. Specifically, ingested frames are queued for transmission to the ingestion service so that, once initiated, the ingestion API library attempts to maintain a continuous stream of available data frames to the Datastore. If data is offered at a rate faster than the ingestion service can process the incoming data, the data queue will eventually fill then block.

To facilitate optimization of the data stream from the data provider side, the ingestion API exposes the awaitQueueEmpty() operation. This operation allows the provider to monitor the buffering of the data stream. The operation will block until the data frame queue empties. By sending a set amount of data frames and measuring queue wait times, the provider can optimize streaming. That is, by monitoring this process, the provider can adjust its data rate to that of the current Datastore ingestion rate. The *datastore-admin* project provides several performance parameters (in the configuration file) for tuning of the data buffer size, bin sizes, and queueing time limits.

Once the provider has finished transmitting all data frames the data stream should be closed using the closeStream() operation. Note that the operation returns the snapshot UID for the entire data stream. After the close stream operation, the interface is no longer active and can be discarded. Note that within the ingestion API library, asynchronous data streams are managed using StreamObserver interfaces as described within Subsection 4.4 and 4.6.2. Data providers need only be concerned with the external streaming operations exposed by the ingestion API.

4.7.3 Query Process and API

The query API library within the *datastore-admin* project is provided for data science and machine-learning applications development. It exposes a narrow API interface where query results are returned as high-level data tables. This is the interface shown in Figure 14. The alternate query API library is contained in the *datastore-client-lib* project; it is a low-level programming interface requiring advanced knowledge of the Datastore. Aspects of this API are treated in Subsection 4.8 where the Datastore query service operations are covered in further depth.

Figure 14 depicts the query API library and its basic operations. Query operations are the aspect of the Datastore seen by *data consumers*, shown on the right-hand side of the diagram. Data consumers are data scientists, data science applications, or any other interested party including other archive-dependent applications such as the Web Application. Data consumers interact with the Datastore archive through the query API library shown in the left-hand side of Figure 14. It explicitly provides a Datastore connection factory service, data stream management, and requested data table loading services. Access to the data archive is through the Datastore query service through both synchronous and asynchronous gRPC communications. The library exposes two separate query interfaces, IQueryServiceMeta for requesting metadata from the Datastore archive, and IQueryServiceData for requesting snapshot data from the Datastore archive. Machine learning and other data science operations would likely involve repeated invocation of both metadata requests and snapshot data requests to isolate relevant data properties and associations. Thus, unlike the ingestion interfaces, clients would likely use both interfaces concurrently.

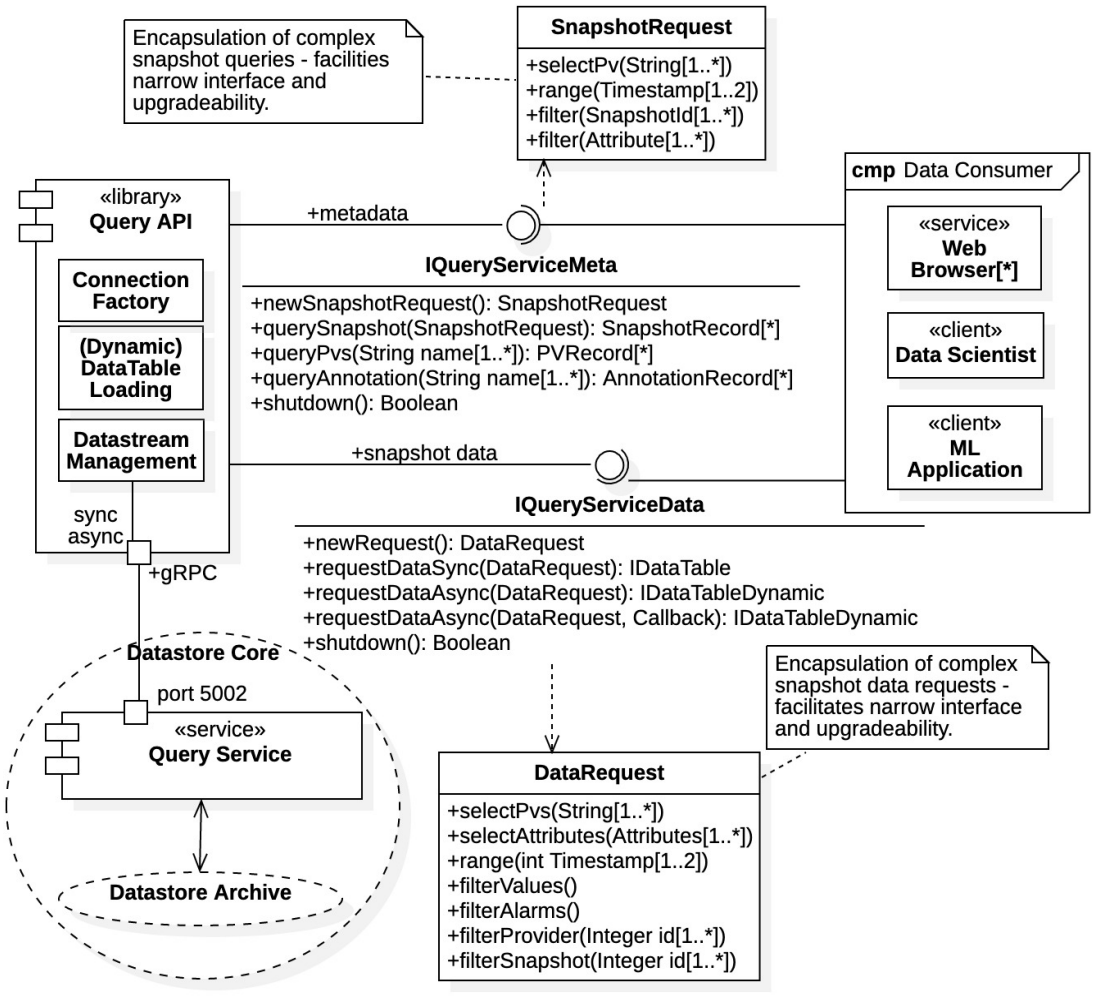


Figure 14: Datastore query operations and query APIs

The process of obtaining these interfaces is exactly analogous to that for obtaining an ingestion API interface and is described in Subsection 4.7.1. The data consumer requests an API interface from the connection factory, in this case the `DsQueryServiceFactory` class, which subsequently makes the connection to the Datastore query service then passes back the appropriate interface implementation. The query API library handles all connections and all data stream management with the Datastore query service. It also interprets all user data requests, formulating them into the appropriate gRPC messages for the query service, then marshalling the query service results into the appropriate response.

As indicated in the diagram, the metadata query interface `IQuerySeriviceMeta` exposes separate operations for each type of metadata requested. This condition is necessary because each metadata type is represented by a unique record type. Note that the interface has a special mechanism for handling *snapshot* metadata requests, as they can be complex. Snapshot requests are encapsulated by special objects, `SnapshotRequest` objects, which are configured by the user and then passed to the `querySnapshot()` operation to perform the metadata request. This mechanism is covered in

further detail within Subsection 4.9.2. All metadata requests are performed using synchronous, blocking communications.

On the other hand, the *snapshot data* query interface IQueryServiceData is quite narrow. It employs a technique for constructing snapshot data requests analogous to that for snapshot metadata requests. A method newRequest() is provided for creating new snapshot data requests objects, realized as DataRequest objects. The DataRequest class contains methods for generating specific snapshot data requests. Once configured, the request is then passed to the requestDataSync() operation to perform a synchronous query, or the requestDataAsync() operation to perform an asynchronous query. The DataRequest utility is cover in more detail within Subsection 4.8.4. Both operations return requests as data tables. The specific data table implementation depends upon the request method as was shown in Figure 10, IDataTable for synchronous requests and IDataTableDynamic for asynchronous requests. (Recall that these are actually interfaces, the concrete data table implementations are hidden.) Synchronous data requests block until the request is completed and the data table is fully loaded upon return. However, synchronous requests are only suitable for query results producing small data tables (less than 4 Mbytes) as they are gRPC message size limited.

Asynchronous data requests do not block, returning a data table instance that is dynamically loaded. Users have access to table data during the dynamic loading process, however, not all data may be available at any given instant. There are notification methods for table loading complete and for table loading exceptions (not shown). Two techniques are available for receiving loading completed notification, one way is through the data table itself (within the IDataTableDynamic interface) and the other through the query API. For the latter case note that the IQueryServiceData interface contains an operation requestDataAsync() accepting the argument type Callback. This argument is a callback function provided by the data consumer for such notification, it receives the fully loaded data table when invoked by the API library. Asynchronous data requests are essentially unlimited as the query API library maintains an active data stream from the Datastore to the data table until the request is completed. The query API library manages all data connections, table loading, and data streams between data tables and the Datastore query service.

Figure 14 identifies the operations within, and consequently the function of, the snapshot request SnapshotRequest and snapshot data request DataRequest classes associated with the two query API interfaces. These classes are very similar and are essentially "builders" for creating their respective request types (i.e., either snapshot metadata requests or snapshot data requests, respectively). New SnapshotRequest and DataRequest objects are always created for the "open query". That is, upon initial creation, the request objects always query for all data within the Datastore archive. For example, in the case of a newly created DataRequest object, it requests all snapshot data currently within the archive. The user narrows the search request by invoking their select, range, and filter methods, samples of which are shown in the diagram. Once the snapshot request or snapshot data request object is configured according to the data consumer's conditions, they are offered to the API for the desired query request.

Note that each query interface has a shutdown() operation. After obtaining a query interface from the connection factory all query operations are active and data consumers can continue archive inspection for as long as desired. However, once query operations are completed, data consumers

should invoke the shutdown() operation to close the connection to the query service and release all gRPC resources. Afterwards the API interfaces can be safely discarded.

4.7.4 Datastore Administration

Basic Datastore administration services are provided in the *datastore-admin* project. The project contains resources, tools, and services for data management, data integrity testing, and performance testing, as well as fully supported ingestion and query APIs based upon the data frame and data table frameworks. There are additional facilities for inspection and manipulation of the InfluxDB and MongoDB databases required by the Datastore. The *datastore-admin* project has minimal dependencies, requiring only the *datastore-grpc* project for direct communications with the Datastore services. It does not employ any Spring or Spring Boot application frameworks. The Datastore administration library is still under development; although not refined, most features are currently available and used in the Datastore evaluations.

A basic outline of the Datastore administration service is shown in Figure 15. The figure indicates that the administration service is a library, labeled DatastoreAdministration, containing explicit components for data management, archive integrity, and performance testing. For ease of illustration, these services are exposed in a single interface although in practice each component exposes a separate interface. In the diagram the administration service is connected to a single Datastore instance, although in practice it may be connected to multiple Datastore deployments and offers operations for individual instance testing.

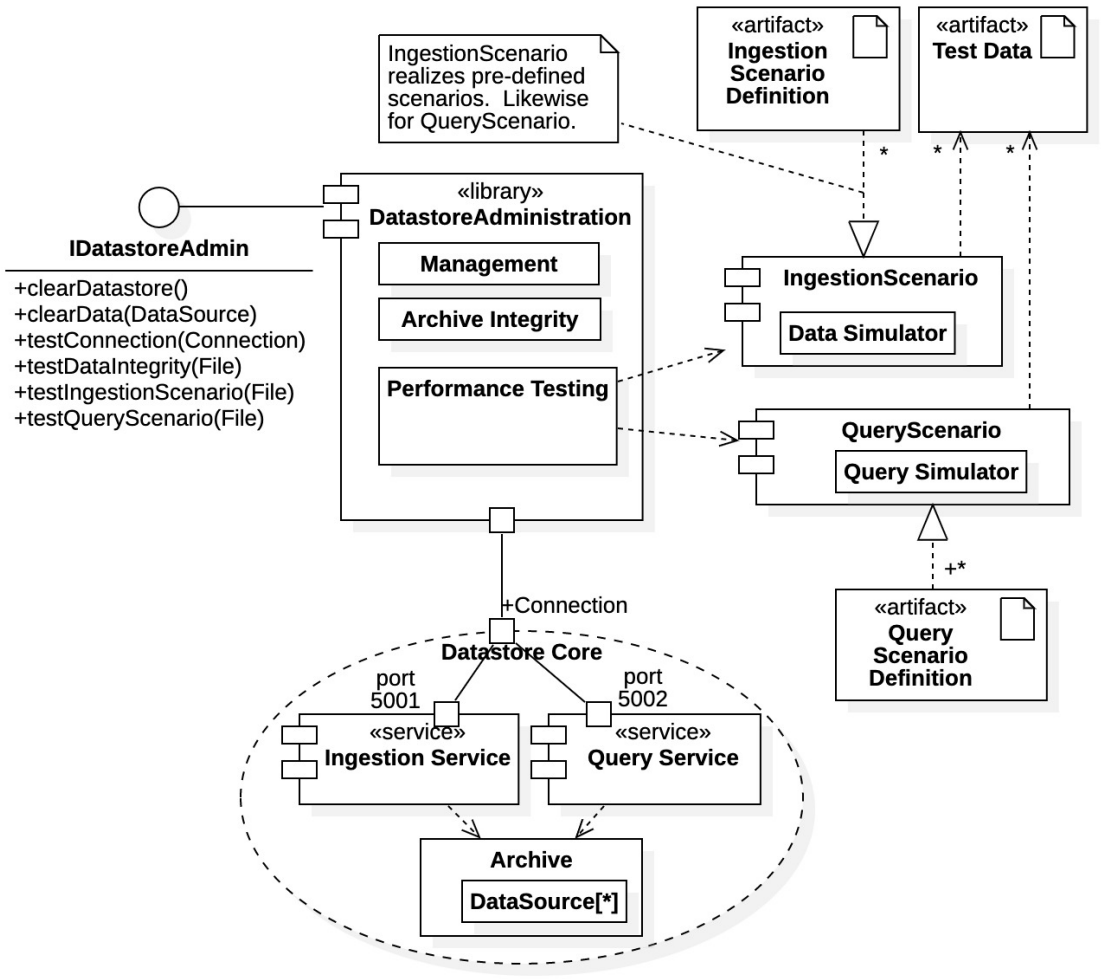


Figure 15: Datastore administration service and API

The Datastore administration project contains "scenario execution" facilities for Datastore operation and performance evaluations. For ingestion testing, there are data ingestion scenarios described in YAML files and executed by a simulated data provider in real time. Performance characteristics of the Datastore were obtained using these scenarios. Several such scenarios mimic the Aggregator service. Test fixtures are also available for accuracy and performance testing of the query service. These text fixtures are used in the performance evaluation of the Datastore query service. Currently, as indicated in the diagram, the fixtures are being refactored into "query scenarios" analogous to that for the ingestion service. These scenarios are descripted in YAML files and realized as simulated data consumers. Development of the query scenarios engine is more complex since it involves the full data stack; operations for data ingestion, data query, and data inspection are necessary to determine scenario success, as well as its performance.

Data integrity tests are an important aspect of Datastore administration. They confirm that ingested data is archived correctly and requested data is queried correctly. Integrity testing includes methods for determining that heterogenous data is stored correctly and that time correlations are preserved. It also requires that metadata, such as user attributes, snapshot IDs, and provider attributes, are recognized during ingestion and correctly recovered. Tools for inspecting and managing the MongoDB and InfluxDB databases are also required in integrity testing.

Data management is essentially provided as a service to Datastore users. Tools for testing communications services, inspection and management of databases, and clearing of data archives are currently available. The administration service also contains numerous unit tests for subsystems and individual components and classes. This includes unit tests for connection services, data streams, data frames, data tables, and data table loading operations. Tools for cleaning the archive of corrupted data or data otherwise deemed unusable are currently under development.

The administration service can manage multiple Datastore systems and archives. This is useful when separate Datastore systems are deployed. For example, a Datastore service may function as a development platform while a separate version locked Datastore service may be deployed for application development. That is, it may be useful to deploy a test Datastore whose archive can be manipulated without consequence, while simultaneously maintaining a secure Datastore system available to data science application developers. Or consider the following situation familiar to data scientists: a separate Datastore system may be deployed as a training service to verify machine learning applications on a predetermined, learning data archive.

Recall that the *datastore-admin* project also contains full Datastore ingestion service and query service APIs. These APIs were described in detail in the previous subsections and depicted in Figure 13 and Figure 14. The ingestion API has full capability based upon the *data frame* and supports both synchronous and asynchronous ingestion. The query service API also operates both synchronously and asynchronously, producing request results in the form of *data tables* and metadata records. It was desirable to include the full API libraries within the *datastore-admin* project since many of the testing resources were developed using these APIs. Some parallel development was necessary to avoid collision with other projects within the Datastore. However, it may be desirable to break out the ingestion and query APIs as standalone libraries now that all the interface definitions have been established.

4.8 DATASTORE QUERY

The Datastore query service provides broad search and query access to the data archive, which contains both snapshot data and associated metadata. Multiple query APIs are available for query operations. The basic operations of the *datastore-admin* query API were covered in Subsection 4.7.3 and we further describe these operations in the sections below, specifically, the data paging operation, the snapshot request utility, the snapshot data request utility. Note that a full section on available metadata is included in Subsection 4.9; this section applies to all available query APIs. The operations with the query API presented by the *datastore-client-lib* project are also covered, however, not so explicitly as this library requires specific knowledge of the gRPC communications framework and a new query language. There, snapshot data requests are formulated in the Datastore Query Language (DQL) which is described in Subsection 4.8.5. Snapshot data request results are returned as paged collections of native gRPC message types. Metadata requests are also returned in their native gRPC message types, which are essentially records containing the metadata properties analogous to those described in Subsection 4.9.

The ability to annotate archived data *post-ingestion* is currently unavailable, specifically, the ability to annotate existing data with user comments, data associations, and post-ingestion information. This feature stated in the initial project goals, but not implemented due to limited resources in Phase I. This should be a primary focus for future development. However, the

framework for archive annotation does exists as the Datastore metadata repository is specifically designed for this feature. Additionally, the Datastore prototype does have the ability to associate separate data sets as "annotated queries," where complex data sets can be identified as saved query requests.

4.8.1 Operation

The basic query operations offered to data consumers was covered in Subsection 4.7.3 when describing the query APIs. Here we provide a brief overview of these operations.

Metadata queries are all performed using the synchronous gRPC communications protocol. Metadata is returned in the form of specialized records particular to the metadata requested (see Subsection 4.9). Metadata results are typically not large and the sole reliance on synchronous communications presented no operational or performance limitations so far. It also serves to keep the interfaces and implementations simple and robust. Snapshot data may be queried using either synchronous or asynchronous communications. Synchronous queries are intended for small query results, specifically, when request results are smaller than the maximum gRPC message size of 4 Mbytes. They return static data tables that are complete at the time of the response. Asynchronous queries can accommodate extreme data requests. They return dynamic data tables which remain connected to the query service and continue to populate until fully loaded. Users may immediately begin accessing the table data before it is fully populated.

The *datastore-admin* project provides separate interfaces for metadata queries and snapshot data queries, `IQueryServiceMeta` and `IQueryServiceData`, respectively (shown in Figure 14). The metadata request interface is larger, supporting all the available record types containing metadata. The snapshot data interface is narrow but supports both synchronous queries and asynchronous queries. Snapshot data request can be made using either a Datastore Query Language (DQL) statement (see Subsection 4.8.5) or using a `DataRequest` object. The `DataRequest` is an auxiliary tool used to encapsulate a DQL statement, as these requests can be quite sophisticated. Thus, the broad query capabilities for snapshot data are embodied in DQL and the `DataRequest` utility.

4.8.2 Data Paging

The asynchronous snapshot data query mechanism is based upon the notion of *data paging*. Although the implementation for paging is hidden behind the common data table interface `IDataTable` for both synchronous and asynchronous queries, the data tables for asynchronous queries are quite different from those generated using synchronous queries (see Figure 10). Synchronous request results are limited in size by the maximum gRPC message size and can be implemented as static data tables. Asynchronous query requests are not size limited and are loaded dynamically without knowledge of any predetermined response size. Thus, their implementation is significantly more complex.

Asynchronous data tables are comprised of an aggregation of *data pages*. Data pages are themselves fully implemented static data tables, the tables resulting from synchronous query requests. They are implemented as dynamically loaded lists containing static data tables, or *pages*. Thus, the size of a paged data table is essentially unlimited. Paged data tables are also referred to as *dynamic data tables*.

Once the asynchronous query is initiated, gRPC messages containing paged data begin streaming from the Datastore query service to a buffer managed within the API library. So as not to interfere with the gRPC streaming process, separate processing threads manage the incoming data

messages. Incoming gRPC messages are retrieved from the message buffer by processing threads, converted to data pages, then loaded into data tables. Once all requested data has streamed and the processing threads have completed, the table is notified of completion.

The contents of a dynamic data table are available during the page loading process. They can be accessed using the common interface IDataTable. Notifications for loading complete and loading errors are available through the derived interface IDataTableDynamic. Loading completed notifications may also be received from the API library through the use of a user-provided callback function.

4.8.3 Metadata Requests

Metadata query requests are relatively straightforward. Most query requests are direct, identifying a single metadata property. This condition is shown in the IQueryServiceMeta interface of Figure 14, which contains several metadata request operations (not all). Moreover, the results of metadata requests are returned as sets of metadata records, with the record type specific to the metadata requested. Each record in the set contains metadata that matches a particular request parameter. For example, to obtain a set of PV records whose property name has a value matching a given regular expression (or list of PV names), one simply invokes the queryPvs(pvRegex) operation with the given regular expression pvRegex. The only exception to this general request process is the snapshot record query mechanism, which is significantly more complex. Snapshot records contain metadata concerning the data providers, the data sources, and the snapshot ingestion process itself. Snapshot records are requested using a SnapshotRequest object, which is analogous to the snapshot data request object DataRequest. This latter condition is necessary because snapshot metadata is significantly broader and can be searched using a much larger set of parameters.

Recall that all metadata requests are processed using synchronous, blocking gRPC communications. Thus, the full request is always returned by the query operation response. Since metadata requests produce relatively small result sets (as compared to snapshot data requests), synchronous metadata requests have, so far, presented no issues in either operation or performance. If, however, future applications require metadata result sets greater than the maximum gRPC message allocation of 4 Mbytes, an asynchronous mechanism would be required. This would entail a straightforward addition of a metadata query interface identical to the current one but backed by an asynchronous communications implementation.

The metadata contained in the Datastore archive is covered in significant detail within Subsection 4.9. As metadata is an important aspect of the Datastore query capabilities an entire section is dedicated here. Familiarity with all available metadata is important for an understanding of the overall query capabilities of the Datastore and the operations of both query API libraries.

4.8.4 Snapshot Data Requests

The search capabilities of the snapshot data archive are very broad. Snapshot data requests can be complex, combining data providers (e.g., the Aggregator), data sources (i.e., hardware), time ranges, metadata properties, user attributes, and other parameters. There are currently two methods of formulating snapshot data requests: 1) using the proprietary Datastore Query Language (DQL), and 2) using the data request utility realized by the class DataRequest. The DQL language is a computer-scienced based SQL-like query language oriented toward advanced users familiar with the Datastore internal mechanisms (e.g., Datastore developers). It was created for its brevity, as a complex request can be realized in a single DQL statement and passed directly through gRPC.

The data request utility is provided for data scientists and data science applications. It requires only familiarity with the Datastore query API, as all possible query options are available through the methods exposed by the `DataRequest` class.

Within the *datastore-client-lib* project query API used by Datastore developers, snapshot data requests are made exclusively using the Datastore Query Language (DQL). The *datastore-admin* project offers a query API that supports both DQL requests and requests made using the `DataRequest` utility. Both snapshot data request methods are described below.

4.8.5 Datastore Query Language

A Datastore Query Language (DQL) request statement supports the broadest range of snapshot data query options. The request is performed using a single query statement. The statement very much resembles an abridged Sequential Query Language (SQL) statement. A DQL statement employs the general predicate-based grammar using the following format:

SELECT <source>, <source>, … WHERE <time-predicate> AND <sub-predicate> <AND> …

The <source> parameters refer to data source and are typically EPICS process variables or other hardware devices. However, they may also include properties of data sources, such as alarm status and conditions. Snapshot data requests return data only from the sources specifically identified within the SELECT clause.

The WHERE clause contains a collection of predicate clauses <time-predicate>, <sub-predicate>, <sub-sub-predict>, etc., each narrowing the search request. Each predicate clause is defined as follows:

<predicate> := <parameter> <operator> <value>.

That is, the predicate clause <predicate> is composed of a parameter <parameter>, an operation <operation>, and a value <value>. Parameters can include data source values, timestamps, user attribute names, data providers, alarm conditions, etc. Operators are the standard logic predicates such as greater than >, greater than or equal >=, equal ==, less than <, etc. In particular, the time range of the snapshot request must be included as the WHERE time clause <time-predicate>. For example, the clause "WHERE time >= -1d" dictates that only snapshot data archived within the last day is to be returned.

As an example of DQL, consider the following legal DQL statement:

```
SELECT Cam1-Image, `BPM-*.*` WHERE time >= '2022-10-01T01:23:45Z' AND BPM-15.value > 10
```

The statement will produces a resulting data table with columns containing data for the source "`Cam1-Image`" and all others data sources with names matching the regular expression `` `BPM-*.*` ``. Note the requirement of forward quotes enclosing `` `BPM-*.*` ``, which must be used whenever a regular expression is specified. Single quotes must be used for predicate values containing character strings, such as that used for the ISO date-time specification in the time clause. Also, the suffix ".*" in the regular expression `` `BPM-*.*` `` is required because, by default, DQL assumes a suffix "`.value`" to identify the measurement value associated with the data source, rather than any other property of the source (such as alarm limit, status, etc.). Thus, "`Cam1-Image`" is shorthand for "`Cam1-Image.value`". Returning to the statement, the results are narrowed in scope by the WHERE predicates. The table will contain column data only with timestamps occurring at or after October 1, 2022, 1:23.45 AM, and where BPM-15 has a value greater than 10.

Consider the additional DQL example statement:

```
SELECT `PV*.*` WHERE time > -5h AND PV01.value < 0 AND `PV02:alarm-severity` == 'MAJOR_ALARM'
```

The second statement will yield data table columns for all sources matching regular expression `PV*.*` whose timestamps have occurred within the last 5 hours and where source PV01 has measure value less than 0, while simultaneously source PV02 has a major alarm status.

For more details on the DQL language see the documentation at https://github.com/osprey-dcs/datastore-service.

4.8.6 DataRequest Utility

As seen in the above examples, DQL query statements can be complex. Thus, DQL is intended for users familiar with the Datastore internal structure. The DataRequest utility has been offered to expediate use of Datastore query capabilities without advanced DQL language requirements. The utility is intended to offer data scientists and applications developers a user-friendly alternative to DQL.

The data request utility is located within the query service API found in the *datastore-admin* project. It can be viewed as a sub-interface to the snapshot data query interface IQueryServiceData as shown in Figure 14. Here snapshot data requests are realized with the builder class, DataRequest, which offers the query parameters and combinations of DQL through its operations. Instances of the DataRequest class are available through the newRequest() operation within the interface.

A DataRequest object is configured by the data consumer to create a specific snapshot data request. Once configured, it is then passed to either the requestDataSync() or requestDataAsync() operations in the IQueryServiceData interface (seen in Figure 14). The data request may be performed synchronously using the requestDataSync() operation or asynchronously using the requestDataAsync() operation. It should be mentioned that the IQueryServiceData interface also supports data query using DQL (not shown in Figure 14). These are offered as requestDataSync(String) or requestDataAsync(String) operations requiring only the DQL query string.

Use of the DataRequest utility class requires no knowledge of DQL. It realizes equivalent snapshot data requests through a set of grouped method types, one group for selecting data sources or annotations, one group for specifying time ranges, and another group for general filtering operations. Each group of methods is prefixed with select, range, and filter, respectively, analogous to the SELECT, WHERE <time-predicate>, WHERE <sub-predict> statements of DQL. A subset of these operations is shown in Figure 14. Each DataRequest object is initialized to the "open query" requesting all archived data (equivalent to the DQL statement "SELECT `*.*`"). That is, any DataRequest instance newly created by the newRequest() operation will return all snapshot data currently within the Datastore. The "selector" methods narrow the data sources to specific names, or name-matching regular expressions. Snapshot data requests are further narrowed by "range" methods that restrict the intervals of allowable timestamps in the resultant data. Finally, there are "filter" methods which allow the result set to be filtered by additional parameter constraints, such as data source values, alarm conditions, user attribute values, and other available metadata.

4.9 METADATA

There are currently four categories of metadata within the Datastore archive: 1) snapshot, 2) data source, or process variable, 3) annotation, and 4) data provider. The available metadata records and their relationship to the Datastore archive, the query service, and the ingestion service are shown in Figure 16. To provide contrast, the basic snapshot data mechanism is also shown in the figure. All the current metadata record types are shown in the left-hand side of the diagram. We provide a description of each metadata record in separate subsections below.

Metadata query requests are returned as record sets containing all the requested parameters and associations. These records can, in turn, be parsed to create more sophisticated metadata queries identifying desired data correlations and associations. Relationships between snapshots, data providers, data sources, user attributes, and timestamps are all available. All records shown in Figure 16 are available through the query service, except the ProviderRecord type which remains internal to the Datastore. The ProviderId record type is generated by the ingestion service during provider registration as was described in Subsection 4.7.2. Its property values are populated from the appropriate ProviderRecord by the Datastore ingestion service, which has access to the metadata archive.

Recall that the Datastore ingestion services allows data providers to assign user attributes to data sets. The ingestion service also creates properties associated with the ingestion process, such as snapshot UIDs, data source names, initial timestamps, final timestamps, etc. All these attributes and properties are contained within the metadata repository.

Before proceeding, note the presence of the Attribute type at the top of Figure 16. An attribute consists of a simple (name, value) pair and represents a general property that can be assigned to any metadata. In fact, *attribute lists* within metadata records are currently the primary mechanism of "data annotation" within the Datastore prototype. (Recall annotation of the data archive with comments, data relationships, and post-ingestion results is currently unsupported.) As discussed below, the AnnotationRecord seen in Figure 16 is somewhat of a misnomer as it represents a *named query*, not an actual archive annotation. The framework for post-ingestion annotation of archived data is available, the feature is intended for future implementations.

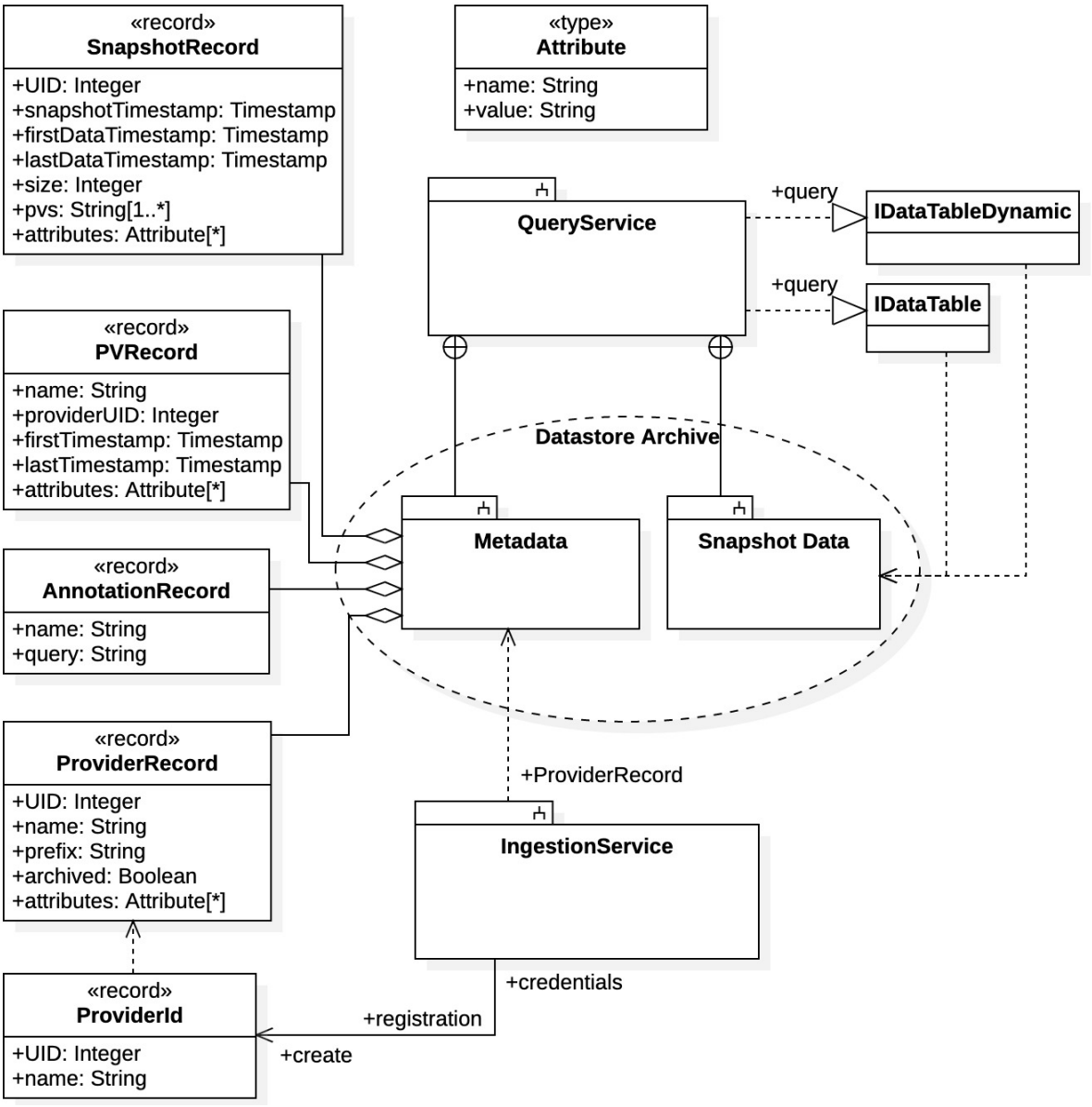


Figure 16: Datastore metadata

4.9.1 Snapshot Records

Metadata requests that return information about the data providers, data sources, and other attributes associated with an ingested "snapshot" are referred to as *snapshot queries*. The results of such queries are returned in records of type `SnapshotRecord`. Each record contains metadata associated with the ingestion process, but no actual snapshot data.

Referring to Figure 16 it is seen that each `SnapshotRecord` instance contains a unique snapshot identifier `UID`, this is the UID assigned to the snapshot at the time of ingestion discussed in Subsections 4.6.3 and 4.7.2. The record also contains several timestamps, a timestamp `snapshotTimestamp` for the snapshot itself (the "trigger time"), the timestamp `firstDataTimestamp` for the first data entry within the snapshot data set, and a timestamp `lastDataTimestamp` for the last data

entry within the snapshot data set. The snapshot timestamp is the time instant at which the snapshot was created and may differ from any of the data timestamps. Within the current Datastore implementation *the snapshot timestamp must be unique*. Specifically, the snapshot UID creation process requires a unique snapshot timestamp to function correctly. A more robust snapshot UID creation method is recommended for future implementation.

Also contained within the snapshot record is the snapshot size `size` (number of data rows), a list of all data source unique names contributing to the snapshot data `pvs` (e.g., EPICS process variables), and a list of attributes for the snapshot `attributes`. Thus, snapshot queries can be complex as there are many query parameters and, hence, query combinations. To simply the query mechanism for snapshot metadata a `SnapshotRequest` utility was implemented; it is directly analogous to the `DataRequest` utility for snapshot data.

4.9.2 `SnapshotRequest` Utility

Instances of `SnapshotRequest` are required to perform all snapshot queries within the *datastore-admin* query API library. As shown in Figure 14, instances of this class are available from the `IQueryServiceMeta` interface through the `newSnapshotRequest()` operation. Like their snapshot data request counterpart `DataRequest` used in the `IQueryServiceData` interface and discussed in Subsection 4.8.6, `SnapshotRequest` objects are essentially builder utilities for snapshot metadata queries. They isolate data consumers from the complexities of internal snapshot queries while maintaining a narrow query interface. Additionally, any updates to the snapshot metadata query mechanism will be seen in the snapshot request utility rather than the query interface.

Analogous to `DataRequest` objects, newly created `SnapshotQuery` objects are configured for the "open query"; specifically, the new request object will return all snapshot records within the Datastore metadata archive. Like the `DataRequest` class, the `SnapshotQuery` class contains `select`, `range`, and `filter` methods to narrow snapshot queries. The `select` methods restrict snapshot result records to those from specified data sources. The `range` methods restrict snapshot result records to those that have snapshot timestamps within a given interval, or those that contain snapshot data acquired within a given interval. The `filter` methods can be used to further reduce the result set by filtering on other properties within the snapshot records, such as UIDs and user attribute values.

Once the `SnapshotRequest` instance has been configured to the desired request, it is offered to the `querySnapshot()` operation of the `IQueryServiceMeta` interface. Within the *datastore-admin* API library the query parameters within `SnapshotRequest` are interpreted to create a gRPC snapshot query request message recognized by the Datastore query service. The query is performed, and the results are returned as a set of snapshot records. Within the *datastore-client-lib* this gRPC message must be directly constructed to perform a snapshot request. Note that the snapshot request gRPC message is a complex data structure containing parameters, subclauses, and substructures; the snapshot request utility insulates the data consumer from the complexities of the gRPC communications constructs.

4.9.3 PV Records

The Datastore maintains metadata on all data sources seen during ingestion. This type of metadata is contained in PV records, or `PVRecord` instances, as shown in Figure 16. Note that the type `PVRecord` is a legacy name as PV records may also represent metadata for other data sources. In

intial development EPICS was the intended platform for Datastore operation and, consequently, all data sources were assumed to be process variables. Referring to the figure, each PVRecord contains a unique name name, the unique identifier (UID) providerUID of the associated data provider, two timestamps associated with the data source, firstTimestamp and lastTimestamp, and an attributes list attributes. The timestamp firstTimestamp is the first instant when the data source was active, the timestamp lastTimestamp is the last instant the data source was active.

It is important to note here that only a single snapshot data provider is associated with each data source. Thus, *if different data providers attempt to add data for the same data source the operation will fail*, potentially catastrophically. Thus, each data source must have a unique data provider, and this condition must be maintained throughout the lifetime of the data archive.

Any time a new data source is encountered during ingestion a new PV record is created. Moreover, PV records can be updated and modified if new properties are encountered during the ingestion process. That is, each time a new data frame is ingested, the data sources within the frame are compared with the existing data sources in the metadata archive. Thus, *PV record comparison may require significant processing and resources during ingestion*. This situation is especially significant for wide data frames containing thousands of data columns. When the Datastore ingests data frames, the entire archive of PV records is queried and compared to each PV within the frame.

The above situation indicates immediate remedies for increasing ingestion performance. Although the current implementation safeguards data integrity within the Datastore archive, the process of PV record comparison could be streamlined. An active cache of PV records could be maintained within heap memory for immediate comparisons with incoming data frames, then stored to the archive upon termination of the API connection. Alternatively, a tradeoff between data integrity and performance is also possible, but not as attractive.

4.9.4 Annotations Records

Currently the Datastore supports *named queries*, which are represented as annotation records of type AnnotationRecord shown in Figure 16. Named queries consist simply of a unique name identifier name and the query string query it represents. The query string is a Datastore Query Language (DQL) statement, which was described in Subsection 4.8.5.

Note that the annotation record does represent post-ingestion annotation of the data archive. Current support for user annotation of archive data is in the form of attribute lists. Specifically, metadata records (except for annotations records) contain a set of user attributes that may be assigned to snapshots, snapshot data providers, and data sources. The assignment of attributes is currently possible only during ingestion, not post ingestion. However, it is a straightforward extension. Moreover, full search capabilities are possible for user attributes.

4.9.5 Provider Records

The Datastore maintains metadata regarding all snapshot data providers that have contributed to the data archive. Referring to Figure 16, data provider metadata is contained in provider records of type ProviderRecord. As also indicated in the figure, these records are created and maintained by the ingestion service. Each provider has a unique identifier, the property UID in the provider record. This unique identifier (UID) is used by the Datastore ingestion service to register providers and manage their data.

Along with the UID of the data provider, the provider record also contains the provider's unique name, any standard prefix associated with the provider, and any user attributes list associated with the provider. An "archived" flag is also contained within the record for internal use by the Datastore; the flag indicates whether the provider data is to be permanently stored regardless of its lifetime. Except for the name and UID, the additional properties within the provider record can be augmented during the registration process. For example, the provider may wish to add additional user attributes to the attributes list indicating, say for example, an upgrade to the underlying hardware.

Provider records are used internally by the Datastore and are currently unavailable for query. However, it is a straightforward process to open the metadata query service to provider record queries if warranted. Additionally, it is possible to obtain the UID of a provider during the registration process.

5. WEB APPLICATION

During development it became desirable to have an independent tool for inspection of the Datastore archive. A particularly attractive option was that of a web browser, allowing developers to connect to a URL and inspect the archive using a standard internet web browser. Recognizing its greater utility, this tool was subsequently developed into a prototype application, the Web Application. This tool is now a complement to the Machine Learning Data Platform allowing universal access to the data archive from remote locations using a standard internet web browser. The Web Application runs standalone as a web service on a separate platform, connections are made using a predetermined URL. Some outstanding features of the prototype are outlined here, along with the basic architecture and operation. Some screenshots of the browser web pages are also included for operational reference. Note that the web pages have not yet been stylized but are fully functional.

The prototype Web Application allows data consumers to interact with datastore query service, requesting data and metadata and displaying the results interactively in a web browser. The application allows users to navigate through archived metadata and snapshot data sets, but does not currently provide any features for processing, manipulating, or exporting data. There are, however, current efforts to export data to various file formats (NumPy, Excel, etc.). Additionally, there are intentions to add features to manipulate data within the metadata archive, specifically, the ability to annotate data with user attributes.

5.1 IMPLEMENTATION

The web application is implemented in Java and JavaScript, utilizing the React JavaScript library for building services and Graphical User Interfaces (GUIs) within web browsers (19). Communications with the Datastore query service are made directly through two separate gRPC libraries, one in Java and one in JavaScript. That is, no external API libraries are required. The `proto` source files within the *datastore-grpc* project are compiled into an additional JavaScript gRPC library used by the Web Application for web browser communications. Thus, the *datastore-grpc* Java library is not used exclusively. (The situation illustrates the utility of the Protocol Buffers technology for supporting alternate programming and scripting languages.)

The Web Application is essentially composed of two separate components, the *application server*, and the *web browser application* as is shown in Figure 17. The application server is implemented in Java while the browser application is implemented in JavaScript utilizing both the React framework and the JavaScript gRPC communication library.

5.2 ARCHITECTURE

The basic architecture of the Datastore Web Application architecture is shown in Figure 17. The two components of the Web Application are shown on either side of the diagram, the application server is shown in the right-hand side, and the web browser application is shown on the left-hand side. The application server is independent and runs on a separate host platform. When launched, it connects to the Datastore query service across the network, then subsequently runs as an independent service connected to both the web browser and the application server. To initiate the Web Application clients as internet web browsers, one connects to this service via a predetermined network URL currently attached to port 3000 (i.e., in the default configuration). Note that the web

browser application itself also typically runs on a separate platform, the platform hosting the web browser.

The function of the application server is twofold. Upon initial connection, it loads the web browser application into the web browser. Henceforth it acts as an intermediary between the web application running in the browser and the Datastore query service, marshalling query requests and their responses. Thus, after the Web Application is launched, the web server essentially functions as a query service API, hence its dependence upon the *datastore-grpc* library.

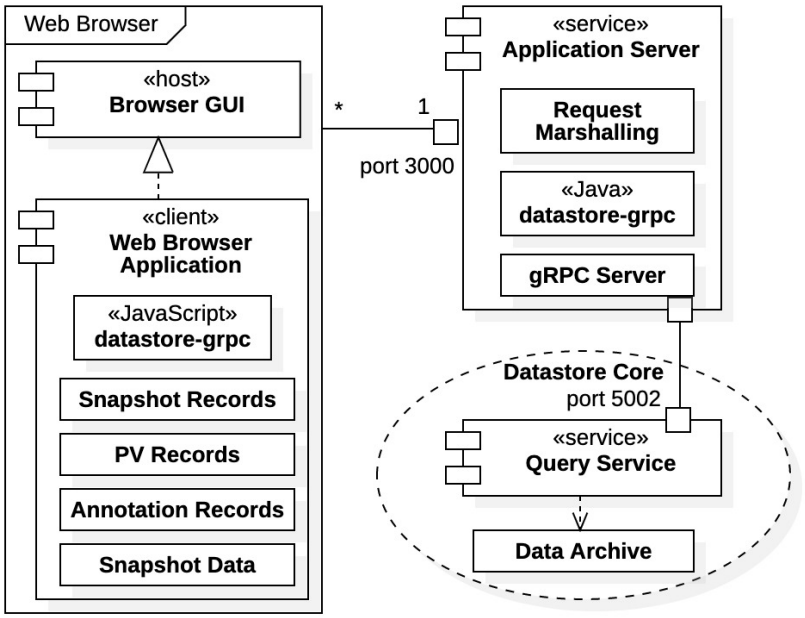


Figure 17: web application architecture

After being loaded into the web browser, the web browser application acts as a standalone client. It exposes the features for archive inspection, which are realized as separate browser pages, covered in the next subsection. As shown in the diagram, currently snapshot records, PV records, annotation records, and snapshot data are all available for inspection by data consumers within the Web Application. Also shown in the diagram is the use of the JavaScript gRPC communications library within the web browser application. Specifically, the web browser application directly packages all query requests into the appropriate gRPC request messages, without the use of an external API library. Conversely, all data responses are unpacked directly from their respective gRPC data messages and used to populate the browser pages. This configuration keeps the Web Application modular and standalone.

5.3 OPERATION

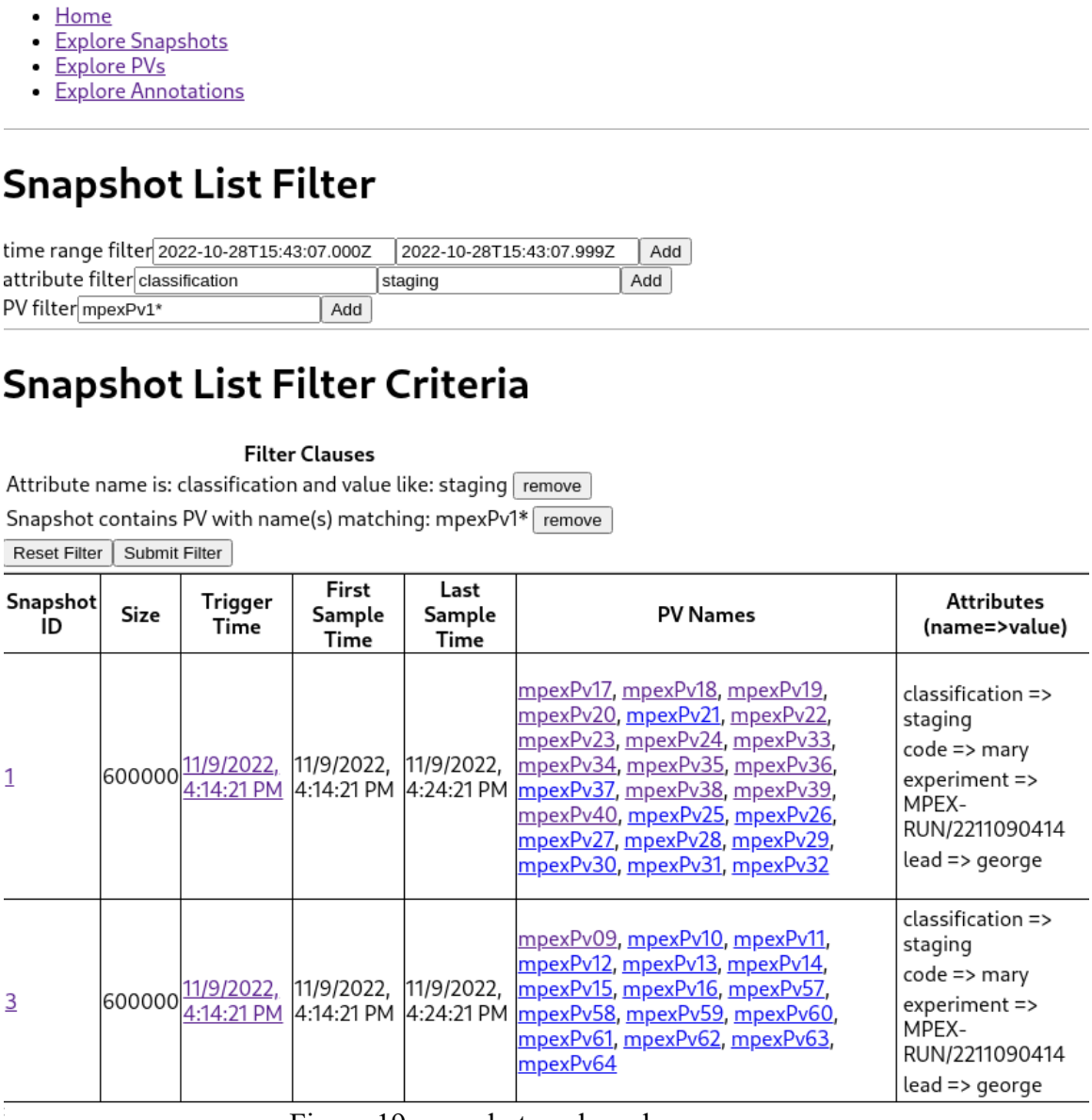
Figure 18 is a screen shot of the Web Application home page, the first page seen in the web browser after connection to the application server. As seen in the figure, there are currently three *perspectives* for browsing the Datastore archive, each according to metadata type. The archive may be inspected by snapshots, by data sources (e.g., EPICS PVs), and by annotations. Each option links to a separate browser page which opens the perspective. Although each perspective is based upon a specific metadata type, it also allows data consumers to request snapshot data associated with the metadata properties. That is, the metadata is not only recovered from the Datastore archive, but it is also used to retrieve snapshot data.

Figure 18: web application home page

To explain the operation of the Web Application we demonstrate inspection of snapshot records within the archive, the most complex metadata type. If one follows the explore snapshots link on the home page, it brings up the *snapshot explorer* page, which is shown in Figure 19. The snapshot explorer is the *snapshot perspective* on the data archive: it allows data consumers to browse snapshot metadata as record lists, that is, snapshot records are shown as lists within the browser page. Users may browse forwards and backwards through the set of all available snapshot records in the Datastore archive. Additionally, as seen in the Snapshot List Filter section, the view of available snapshots can be narrowed by filtering on time range, PV data sources, and attribute values. That is, the set of visible snapshot records can be restricted by their associated metadata property values or by timestamp ranges. Also seen in the screen shot is the ability to remove filters and open the current snapshot request. Snapshots records are displayed by their unique snapshot ID along with their property values. The screenshot in Figure 19 displays the properties for snapshots with UIDs 1 and 3.

At the top of the screen shot are links to the other Web Application perspectives. This a convenience feature found in all other browser application perspectives. Thus, it is possible to immediately jump from perspective to perspective. Another important convenience, all query requests are amended to page URLs so that browser page configurations can be bookmarked within

the web browser. Thus, if a particular perspective configuration is found desirable it can be immediately recalled using a browser bookmark.



- Home
- Explore Snapshots
- Explore PVs
- Explore Annotations

Snapshot List Filter

time range filter 2022-10-28T15:43:07.000Z 2022-10-28T15:43:07.999Z Add
attribute filter classification staging Add
PV filter mpexPv1* Add

Snapshot List Filter Criteria

Filter Clauses

Attribute name is: classification and value like: staging remove
Snapshot contains PV with name(s) matching: mpexPv1* remove
Reset Filter Submit Filter

Snapshot ID	Size	Trigger Time	First Sample Time	Last Sample Time	PV Names	Attributes (name=>value)
1	600000	11/9/2022, 4:14:21 PM	11/9/2022, 4:14:21 PM	11/9/2022, 4:24:21 PM	mpexPv17, mpexPv18, mpexPv19, mpexPv20, mpexPv21, mpexPv22, mpexPv23, mpexPv24, mpexPv33, mpexPv34, mpexPv35, mpexPv36, mpexPv37, mpexPv38, mpexPv39, mpexPv40, mpexPv25, mpexPv26, mpexPv27, mpexPv28, mpexPv29, mpexPv30, mpexPv31, mpexPv32	classification => staging code => mary experiment => MPEX-RUN/2211090414 lead => george
3	600000	11/9/2022, 4:14:21 PM	11/9/2022, 4:14:21 PM	11/9/2022, 4:24:21 PM	mpexPv09, mpexPv10, mpexPv11, mpexPv12, mpexPv13, mpexPv14, mpexPv15, mpexPv16, mpexPv57, mpexPv58, mpexPv59, mpexPv60, mpexPv61, mpexPv62, mpexPv63, mpexPv64	classification => staging code => mary experiment => MPEX-RUN/2211090414 lead => george

Figure 19: snapshot explorer browser page

Returning to the bottom of the screen shot in Figure 19, all fields within a snapshot record are displayed, specifically, the snapshot UID, the size of the snapshot (number of data rows), the snapshot timestamp (i.e., "trigger time"), the first and last timestamps seen within the snapshot data set, all sources providing data to snapshot, and any user attributes associated with the data provider. The actual snapshot data is displayed on a separate browser page.

It should be noted that the snapshot explorer page also includes hyperlinks within the above metadata property values. These hyperlinks are attached to the metadata property values. They navigate to Web Application *inspector* pages providing detailed inspection of metadata, such as the *PV inspector* page for individual data sources, and the *snapshot inspector* page for individual

snapshots. For example, clicking on a particular PV name will automatically formulate the appropriate query request and bring up the *PV inspector* page (not shown), pre-populated with the PV record field values. Again, it is noted that whenever such a request query is created it is also appended to the URL for browser bookmarking.

By choosing an individual snapshot, either by clicking on the snapshot UID hyperlink on the bottom left-hand side of Figure 19, or by direct entry, the *snapshot inspector* page is invoked. This browser page is shown in the screen shot of Figure 20. As seen there, the metadata properties of the snapshot are listed at the top, including the UID, the size (in timestamped rows), the trigger timestamp, the time range of all ingested snapshot data, the data sources (i.e., the PVs producing the data), and any user attributes associated with the snapshot. The snapshot inspector also allows the user to inspect the snapshot data within the snapshot. Additionally, the snapshot inspector allows users to filter the displayed snapshot data with a set of options, either with specified time range, or by data source, or with a combination of both. The snapshot inspector has forward and previous buttons allowing the user to scroll through requested data sets, while also adding or removing additional filtering operations.

- Home
- Explore Snapshots
- Explore PVs
- Explore Annotations

Snapshot Details

ID:

3

Size:

600000

Trigger Timestamp:

11/9/2022, 4:14:21 PM

First Sample Time:

11/9/2022, 4:14:21 PM

Last Sample Time:

11/9/2022, 4:24:21 PM

PV Names:

mpexPv09, mpexPv10, mpexPv11, mpexPv12, mpexPv13, mpexPv14, mpexPv15, mpexPv16, mpexPv57, mpexPv58, mpexPv59, mpexPv60, mpexPv61, mpexPv62, mpexPv63, mpexPv64

Attributes:

classification => staging
code => mary
experiment => MPEX-RUN/2211090414
lead => george

Snapshot Data Filter

time range filter 2022-11-09T23:14:21.000Z 2022-11-09T23:24:21.000Z Add
PV filter mpexPv6* Add

Snapshot Data Filter Criteria

Filter Clauses

Snapshot contains PV with name(s) matching: mpexPv6* remove

Reset Filter Submit Filter

Previous Page Next Page

timestamp	mpexPv60	mpexPv61	mpexPv62	mpexPv63	mpexPv64
11/9/2022, 4:14:21 PM 410345849	0.2000922879974548	0.7954638333980324	0.46197445952178695	0.05000639473895334	0.4384815885930552
11/9/2022, 4:14:21 PM 411345849	0.20434387306691187	0.7877282527942066	0.4597394547181328	0.06844811729016374	0.4398517393214379
11/9/2022, 4:14:21 PM 412345849	0.19759005721193956	0.8012339410911983	0.4618919765901268	0.0627376510656032	0.43912225607437494
11/9/2022, 4:14:21 PM 413345849	0.1880360592131297	0.793082402718566	0.46018597128954103	0.06652349086722817	0.43969551411869945
11/9/2022, 4:14:21 PM 414345849	0.20873551770091836	0.7834345145507549	0.4617124709401858	0.0766654158461611	0.439526297232016
11/9/2022, 4:14:21 PM 415345849	0.20042730735210165	0.7907224870693857	0.4594787551819115	0.06827897711460335	0.439677780373476
11/9/2022, 4:14:21 PM 416345849	0.20464365849220018	0.7839412570105436	0.46004178516494537	0.053037929879839935	0.4398249444835905
11/9/2022, 4:14:21 PM 417345849	0.20743229909426886	0.800926109461076	0.4616846519208024	0.06931637779358465	0.4370356310654396
11/9/2022, 4:14:21 PM 418345849	0.20702778942698005	0.7863188613115656	0.46225796582779693	0.08184514507713089	0.43640700244952996
11/9/2022, 4:14:21 PM 419345849	0.19835248125359436	0.7970922872157786	0.46147342131047847	0.07202651442633193	0.4382694987705031
11/9/2022, 4:14:21 PM 420345849	0.20341489340784238	0.8004789775530455	0.4608705577707226	0.08252377842439837	0.43963681096032625
11/9/2022, 4:14:21 PM 421345849	0.1934599316792456	0.7970824174106597	0.46173177053841635	0.07101954686424491	0.43778187453776984

Figure 20: snapshot inspector browser page

Not shown are the *PV Explorer* and the *Annotations Explorer* perspectives. The *PV Explorer* perspective allows users to navigate through all PV records within the Datastore archive while filtering on property values and timestamps. It also brings up the *PV Inspector* page which allows users to identify all snapshots containing the data source and, consequently, inspect all snapshot data offered by the data source.

The Annotations Explorer is relatively straightforward. It simply displays a list of all the named queries within the Datastore archive, as there are currently no user annotation capabilities. However, this perspective can be significantly augmented once the post-ingestion annotations features become available with the Datastore.

PART 3: Evaluations

6. AGGREGATOR EVALUATION

6.1 TEST PLATFORM

The data Aggregator system has been deployed and tested on an EPICS platform simulating the Linac Coherent Light Source (LCLS) within the Stanford Linear ACcelerator (SLAC) facility. Specifically, the platform contains 200 nodes each supplying 16 separate time-series signals, representing the data obtained from Beam Position Monitors (BPMs) at 200 locations along a simulated beamline. The sampling rates for each signal are varied up to 1 kHz.

6.2 OPERATION

The Aggregator performed as required, no operational issues were observed. The distributed front-end Aggregator subsystems were fed signals from the 200 test nodes as described in Subsection 3.2 and shown in Figure 4. The front-end components created NTTable instances containing the signal data which were then transported to the central aggregation service by the EPICS pvAccess protocol. There they were coalesced by the central aggregation service and correctly correlated in time. The aggregate data tables were written to HDF5 files where they could be inspected for accuracy.

6.3 PERFORMANCE

The current performance of the data aggregation system is close to that of our specified goals. Specifically, the Aggregator was able to process the 3,200 scalar signals of the test platform at a sampling rate of 1 kHz. This is a comfortable figure slightly below the stated goal of 4,000 signals at 1 kHz.

The overall data rate for the double-valued scalar data was 25.6 Mbytes/second. As the Aggregator is implemented in C++ and compiled into native binary (rather than Java bytecode), the observed data rate is slightly below the desired 32 Mbytes/second required for 8-byte double representation. However, at the time of this writing the Aggregator has yet to be evaluated with a test platform containing 4,000 signals. Thus, it is unknown if the Aggregator would see any performance drop with the additional 800 signals.

The Aggregator component of the Machine Learning Data Platform is mature. We do not expect significant development efforts in the future, however, testing against a full 4,000 signal data source has yet to be performed. Additional testing of MLDP full data throughput is also required once the Datastore ingestion service is advanced enough in performance to accept data from the Aggregator service.

7. INGESTION EVALUATION

Unlike the Aggregator system of the Machine Learning Data Platform, the Datastore component is still early in development. However, with the working prototype we can accurately determine the status of this development. Extensive evaluations of the prototype were independently performed for both the ingestion service and the query service. Here the results of the ingestion evaluations are presented. This includes both operational limitations of the current prototype as well as performance results.

Performance results were conducted independently with two separate data simulators on two different platforms. The first simulator presents "ingestion scenarios" and was used to evaluate Datastore ingestion of various data types. It also emulates the operation of the Aggregator system, explicitly producing data frames of various sizes and rates. The second data simulator imitates the MPEX facility at Oak Ridge National Laboratory producing data in a low-level format. Outputs from the scenarios-based simulator are included in APPENDIX A for synchronous ingestion and APPENDIX B for asynchronous ingestion.

7.1 LIMITATIONS

Within the current Datastore prototype there are notable operational limitations of the ingestion capabilities which are identified here. Operational limitations are described in this subsection and operations errors are described in the next subsection. The descriptions are detailed and technical. The intent is to document the current implementation issues as clearly as possible providing scope for future development.

7.1.1 Time References

Time references are the method by which the Datastore ingestion service identifies the timestamps within ingested snapshot data. They are realized as gRPC message types within the *datastore-grpc* project communications library. Currently there are two separate types of time references available. One type, the *timestamp iterator*, contains only two parameters: 1) a timestamp and 2) a sampling rate. All data within the incoming data frames (i.e., every row in every frame) is assumed to be sampled at the given sampling rate with the sampling process initiated at the given timestamp. The other type, the *timestamp list*, contains the entire set of timestamps for the incoming snapshot data set. That is, a timestamp is explicitly assigned for each row of each data frame. A gRPC data message containing the snapshot data can contain only one type of time reference. The use of the timestamp iterator is convenient when sampling rates are known a priori and do not vary, specifically, when we have a continuous, uninterrupted data stream. On the other hand, the timestamp list is necessary when data acquisition rates vary, or when acquisition may be subject to interruptions.

At current, Datastore data streaming operates as expected when using timestamp iterators. Specifically, it is possible to continuously stream data to the Datastore under the assumption that all incoming data is sampled at the same rate and that acquisition process started at the given time instant. However, to use a timestamp list *one must know all timestamps for the entire data stream a priori*. One cannot stream timestamps as data arrives. To further complicate matters, for asynchronous communications a practical buffering technique to mitigate this limitation is not possible. The situation is explained below.

To communicate with the Datastore ingestion service, a gRPC connection must first be established. For asynchronous data streaming, a type of "handshake" is established where the Datastore is configured for incoming asynchronous data ingestion (recall that this action is performed with an exchange of gRPC interfaces). Henceforth the Datastore expects to be sent a time reference by which it assigns *all timestamps* for the forthcoming data. If a timestamp list is sent, it must contain *all timestamps for the entire data stream* maintained under the current gRPC connection. Otherwise, an exception is thrown during streaming. Thus, the best one can do is buffer all incoming data at the API library level, send all the timestamps for the buffered data, then send the snapshot data itself. The connection must then be terminated, closing the streaming session as no further timestamps can be sent. A new gRPC connection would need to be established to send additional snapshot data; however, establishing gRPC connections requires time and resources.

It may be possible to maintain a queue of repeating connections by which data is buffered, sent, then terminated. It was seen that simultaneous data streams incur no performance overhead (see Subsection 7.3.6). Thus, the overhead in establishing multiple connections might be minimal. However, this model essentially defeats the advantage of data streaming. In the preferred model an open ingestion connection is maintained, then data is continuously streamed as it is acquired.

7.1.2 Message Size

In the current configuration, gRPC message allocation sizes are restricted to a maximum of 4 Mbytes (specifically 2^{22} bytes). It is unknown if this is a hard limit of Protocol Buffers, or if the message size can be adjusted. Regardless, one must be aware of gRPC message size limitations and how they affect data frames transmitted over networks. Attempting to send data messages larger than the size limitation results in a gRPC connection termination for both synchronous and asynchronous communication.

7.1.3 Data Binning

The gRPC message size restriction can be circumvented using "data binning." This feature was added to the *datastore-admin* project and was described in Subsection 4.6.4. Briefly, before a data frame is sent to the Datastore, its memory allocation is computed. If a data frame is too large it is binned into smaller frames meeting the size requirements. However, this can be an expensive process depending upon the structure of the data frame (specifically, for frames with large column counts). For asynchronous streaming, performance is improved by simultaneously binning and streaming, that is, the data bins are created and buffered while the data stream buffer is active. Thus, while the Datastore is processing incoming binned data frames the client library is building subsequent bins (the primary benefit in asynchronous communication). Parameters for performing tuning of the data binning feature are available in the *datastore-admin* configuration file.

7.2 Ingestion Errors

Notable implementation errors were observed with the Datastore ingestion service during evaluations and are identified below. These issues should be addressed immediately in future efforts.

7.2.1 Timestamp Lists

In addition to the timestamp list limitation described above, another unresolved issue exists concerning the use of timestamp lists with synchronous gRPC communication. If one attempts to send multiple data frames (using multiple RPC calls) over the same gRPC connection using

timestamp lists the operation fails. However, single data frames over one connection operate correctly.

Conceptually, sending a timestamp list and snapshot data within a single gRPC message should not present any difficulty. Snapshot data and timestamp lists contained within a single gRPC message is consistent by design, and within synchronous communications single remote procedure calls are not released until all data has been exchanged. However, this process also fails to operate correctly

Considering the repeated issues concerning timestamp lists, they were not used in any performance testing.

7.2.2 Snapshot UIDs

The generation of snapshot UIDs appear to be mishandled by the Datastore ingestion service. This condition is most conspicuous with synchronous data transmission where a single snapshot UID is produced for each data frame ingested. This issue has also been observed within the metadata archive, specifically within snapshot records. The Datastore ingestion service sometimes generates multiple snapshot UIDs for a single data frame (specifically, for the gRPC data message). It also sometimes fails to generate unique snapshot UIDs for data frames, that is, multiple data frames are assigned the same snapshot UID. The root cause of this error must be identified and corrected.

The sporadic nature of snapshot UID assignments by the Datastore ingestion service can be viewed in the synchronous performance results listed in APPENDIX A.

Although not an error, in principle, for synchronous communications it should be possible for API libraries to identify multiple data frames as belonging to a single snapshot. This condition would be advantageous when data binning is invoked, and all bins could be associated with the same data frame. This behavior can, perhaps, be facilitated by amending the time reference gRPC message. It is worth consideration in future implementations.

7.3 PERFORMANCE

In the current Java implementation, the performance goal of ingesting 4,000 signals at a sampling rate of 1 kHz requires a data rate of 96 Mbytes/second. This number is found by considering the memory allocation for a Java double object (i.e., type `Double`), 24 bytes. We must be able to ingest 4,000x24 bytes every millisecond, the sample period for a 1 kHz rate. Thus, *the overall data rate for the performance goal is 100 Mbytes/second* with a Java implementation. Note that a C++ native implementation requires 8 bytes for a double representation and, consequently, would reduce the data rate to 32 Mbytes/second.

Initial ingestion performance tests were executed using "scenarios." Each scenario consists of a set of preconfigured data frames to be sent to the Datastore at a continuous rate slightly faster than they can be processed, ensuring that the Datastore is operating at maximum capacity. The various data frame configurations are meant to test ingestion performance over a wide variety of conditions. Each scenario case is described in Subsection 7.3.4 where results are summarized and tabulated. Simulation of the Aggregator system is also presented there.

An additional performance study was performed using the data simulator available in the *mpex-spd* project. This data simulator emulates the Material Plasma Exposure eXperiment (MPEX)

facility at Oak Ridge National Laboratory (ORNL) (12). Specifically, the data simulator produces a data stream of 64 scalar PVs up to a 1kHz rate, with an additional separate stream of 240x640 pixel images at a variable rate from 4 to 20 Hz. Performance studies here corroborate the scenario studies. Results of this study are presented in Subsection 7.3.6.

In all scenario evaluations, data is streamed to the Datastore both synchronously and asynchronously (i.e., for each scenario). For each test a connection to the Datastore is established and maintained while data frames are continuously transmitted. Once all data has been transmitted the connection is closed and performance calculations are made. The results are summarized within tables in the case studies subsection. The output files of the scenario executions are included in the appendices in the order that they appear in the case studies. The summary tables include the scenario names for reference within the appendices.

7.3.1 Platforms

Performance testing was executed on development platforms, no specialized testing hardware was used. The Datastore ingestion service was hosted on the development platform, along with the InfluxDB and MongoDB databases. Connections to the ingestion service were made using the appropriate ingestion API library for the given data simulator. All Datastore services and database installations were hosted on a single node where communications were performed in loop-back mode, as shown in Figure 21. That is, all Datastore gRPC connections were made using the network loopback, as were for the database connections.

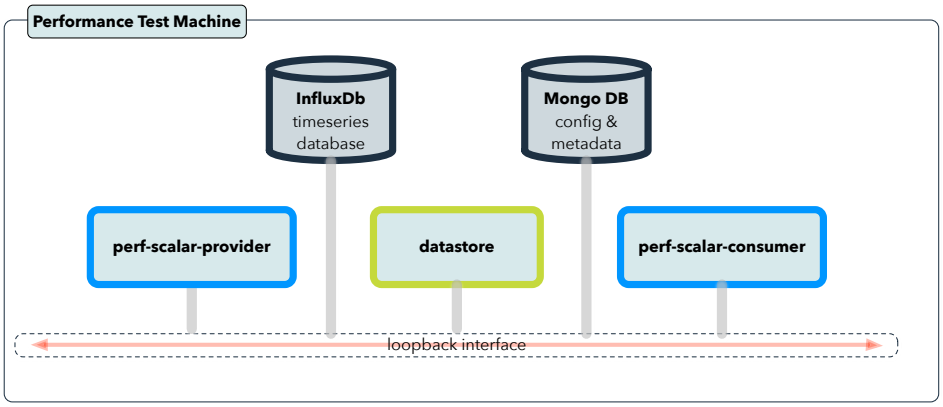


Figure 21: ingestion evaluations platform configuration

All tests were performed on a single-node platform with non-partitioned InfluxDB installation and a single-node MongoDB installation. For the scenarios testing, the CPU was an intel i7 processor running MacOS 11.4 with 6 cores running at 2.7 GHz with 16 GBytes of RAM. The test platform for the MPEX simulator was similar but hardware systems were more advanced consisting of an intel i7 processor with 10 cores and 32 GBytes of RAM memory.

Note that the data simulators were also hosted on the same node. Thus, there may have been competition between the data simulator and the Datastore for host resources, although the data simulator was designed to preprocess all simulated data and avoid conflict as much as possible. All performance characteristics should be considered with this test configuration in mind. It is expected that performance data rates would significantly improve using a dedicated host platform for the Datastore, along with a high-speed data network.

7.3.2 Data Simulators

The scenarios data simulator is available in the *datastore-admin* project, (see src/test/com/ospreydcs/datastore/admin/grpc/ingestion/model/TestIngestionScenario); it was briefly discussed in Subsection 4.7.4 and shown in Figure 15. The scenarios simulator requires the ingestion API within the *datastore-admin* project, it is designed to send pre-configured data frames at designated rates. It can be configured to simulate a variety of ingestion scenarios: various data types, data frame shapes and sizes, number of frames sent, rates at which they are sent, and the duration of the test. Each frame is sent on separate process threads which block if the previous thread is still active, threads do not compete for resources. With this implementation the ingestion service may be continuously supplied data. Burst rates are found when the simulator offers a batch of data frames then terminates. Maximum data processing rates are determined when the simulator offers a sustained stream of data frames faster than the ingestion service can process it. Thus, the frame rates and overall data rates can be accurately measured for maximum performance. Burst rates determine data transmission limits while sustained transmission determines data processing rates. Scenarios are defined in YAML files within the *datastore-admin* project.

The MPEX data simulator is available in the *mpex-sdp* project. It sends scalar and image data to the Datastore ingestion using the *datastore-provider-lib* ingestion API library. This data simulator is designed to test Datastore ingestion with narrow data frames but extended time intervals (e.g., minutes to hours). Both data transmission rates and data processing rates can be independently measured with this simulator. There are configuration options available within the application properties files of the *mpex-sdp* project.

7.3.3 Data Binning

Several different bin sizes were used during scenarios testing (1 Mbyte, 2 Mbyte, and 4 Mbyte). It was found that the largest bin size, 4 Mbytes, either outperforms or performs as well as other bin sizes. This corresponds to the largest gRPC message size that can be transmitted. For synchronous streaming it always appears best to transmit the maximum data over a single remote procedure call. Moreover, one cannot bin wide data frames with row memory allocation larger than the maximum bin size. Thus, all performance tests were executed with a maximum bin size of 4 Mbytes.

7.3.4 Scenarios Cases

Discussed below are the performance results for multiple scenarios case studies. Each case was performed for both synchronous and asynchronous streaming. In all cases (except the debugging case) the scenarios simulator was configured to deliver data frames at a rate greater than the processing capabilities of the Datastore ingestion service; thus it is always operating at maximum capacity. Burst rates are determined by sending small data sets. The raw output results for each scenario are listed in the appendices; synchronous ingestion scenario results are listed in APPENDIX A, and asynchronous ingestion scenario results are listed in APPENDIX B.

Heterogeneous Tables

Three scenarios containing data frames of differing data types and sizes were executed to test overall operation of Datastore ingestion over a variety of circumstances. The data frame memory allocation was greatly varied. Only two frames were streamed to the Datastore in each case. Note that the last case contains extremely large data frames with an overall data transmission of over 350 Mbytes.

1. 1 column ea. Boolean/Integer/Double/String; 10 rows; 2.2 Kbyte frames (for debugging)
2. 10 columns ea. Boolean/Integer/Double/String/4K ByteArray; 100 rows; 5.2 Mbyte frames
3. 10 columns ea. Boolean/Integer/Double/String/4K ByteArray/65K image; 250 rows; 177 Mbyte frames

The first scenario is the base scenario for testing and debugging the data simulator, it demonstrates basic operation where small mixed-scalar data frames are streamed at a rate that does not challenge Datastore ingestion. The second case is designed to test ingestion performance and the binning facility of the API library, while the third case is meant to overwhelm the ingestion service with extremely large data frames of heterogeneous data containing both raw data (byte arrays) and images. Results for the three cases are collected in Table 1 below.

Case No.	Scenario Name	Frame size (Mbytes)	Synchronous rate (Mbytes/sec)	Asynchronous rate (Mbytes/sec)
1 (debug)	TWO_SMALL_FRAMES	0.022	0.005	0.083
2	TWO_MODEST_FRAMES	5.2	0.623	37.0
3	TWO_HUGE_FRAMES	177	1.09	2.07

Table 1: baseline ingestion rates

The first case suggest a minimum allocation for data frame sizes to achieve ingestion rate performance. Specifically, if the frame allocations are too small, disproportionate resources are spent on routine connection and transport rather than actual data processing. The asynchronous data rate seen in case 2 is suspicious and warrants further inspection, although anomalously high rates are seen in another case below. It is assumed that this is a burst rate since the total stream size is only 10 Mbytes, requiring minimal data processing. Thus, the 37 Mbytes/second figure likely reflects data transmission rates more than data processing.

Note the data rate of 2.1 Mbytes/second seen for asynchronous ingestion in case 3. Total stream allocation here is over 350 Mbytes and thus requires substantial processing. This case reflects the best data rate for continuous, sustained ingestion of mixed data, as data binning would result in a data stream of almost 90 frames requiring almost 3 minutes of real time for ingestion. The inclusion of image and raw data is likely a significant performance boost as processing of images appears to be efficient (see case below).

Data Arrays, Data Structure, and Images

The Datastore is capable of archiving and retrieving one-dimensional arrays of double values; they are treated as single entries within a data frame (i.e., analogous to a single scalar value). Higher dimensional arrays are currently not supported. Data structures are also supported by the Datastore, also being stored as single entries within a data frame. Likewise with image data, which is stored as a single entry consisting of a raw byte array with an image format identifier.

The scenario cases below were presented to the Datastore to test fundamental archiving of complex data types (i.e., non-scalar data). Note that the frame memory allocation is nontrivial, but the overall dimensions are small; the frames contain only 10 columns and 10 rows. In each case 10 frames were streamed to the Datastore so total stream allocation is 10 times larger. Thus, the last case contains the largest overall ingestion size of 260 Mbytes.

1. 10 columns of 38 Kbyte arrays; 10 rows; 3.8 Mbyte frames
2. 10 columns of 82 Kbyte data structures; 10 rows; 8.9 Mbyte frames
3. 10 columns of 260 Kbyte images; 10 rows; 26 Mbyte frames

The performance results are listed in Table 2 below.

Case	Scenario Name	Frame size (Mbytes)	Synchronous rate (Mbytes/sec)	Asynchronous rate (Mbytes/sec)
1 (arrays)	LARGE_ARRAYS	3.8	2.39	5.7
2 (structures)	LARGE_STRUCTURES	8.9	2.03	48.2
3 (images)	LARGE_IMAGES	26	4.6	5.35

Table 2: ingestion data rates for complex data

Again, we see a suspicious overperformance in case 2 for asynchronous streaming. However, the trend is clear, the ingestion of large data frames with small column counts (i.e., "narrow frames") performs best.

Upon scrutiny of ingestion operation, it was discovered that the data structures were being transmitted through the ingestion service, however, storage of the data structure was incomplete. Data structures are being processed incorrectly (see Subsection 8.2.3 on data integrity testing). Thus, the figure of 48.2 Mbytes/second is indicative of data transmission without full processing.

For the last case, ingestion of large images, the Datastore maintains an asynchronous data rate over 5 Mbytes/seconds for continuous, sustained ingestion. This is indicative of efficient processing and archiving of image data, which is stored in system files rather than the InfluxDB database. Thus, it may be possible to increase all ingestion performance by archiving snapshot data to system files, say using the HDF5 format. The synchronous data rate of 4.6 Mbytes/second is also impressive for the image data case. It appears as though synchronous communications is equally as efficient when archiving raw data to system files

Scalar Tables

These cases are intended to represent the ingestion of large NTTable objects from the Aggregator system; the *datastore-admin* ingestion API library accepts NTTable objects as data frames. Data frames consisting of scalar, time-series data were streamed to the Datastore in the following configurations:

1. 100 columns of Boolean/Integer/Double/String values; 100 rows; 370 Kbyte frames
2. 500 columns of Double values; 500 rows; 6 Mbyte frames
3. 500 columns of String values (avg. 60 chars); 500 rows; 41 Mbyte frames (avg.)
4. 1,000 columns of Double values; 1,000 rows; 24 Mbyte frames
5. 2,000 columns of Double values; 1,000 rows; 48 Mbyte frames

The first case sets a baseline for table data and establishes that differing mixed-scalar types may be ingested in the same snapshot (i.e., wider than the previous 10 column mixed scalar frame). The second case tests Datastore performance for basic scalar table ingestion (i.e., NTTables) using

moderately wide frames. Additionally, the 6 Mbyte data frame memory allocation is just over the 4 Mbyte gRPC message limit, so that data binning is invoked. The third case is meant to compare performance of double-value ingestion to that of large character-string ingestion. The frames have the same column width, but the character strings require more than 5 times more allocation. Both should contain the same number of metadata operations; comparison between the two cases provides an estimate of the Datastore metadata processing requirements.

The fourth and fifth cases are intermediaries between the stated goal of 4,000 signal acquisition at 1 kHz sampling. Since it became clear the performance goal was not possible, a 4,000 column scenario was never executed. (Note Case 5 for the total ingestion of 480 Mbytes requires over 30 minutes to complete at the asynchronous rate of 0.256 Mbytes/second. Synchronous ingestion takes over 3.5 hours for completion.)

These scenarios specifically test the Datastore ingestion processing and archiving performance *under continuous, sustained ingestion*. Frames were intentionally sent fast enough to back log the ingestion service and invoke frame buffering by the API library. Henceforth frames are offered at a rate maintaining the queue backlog so that the ingestion service is continuously processing data frames; there is always a queued frame ready for transmission whenever a "ready" is acknowledged from the ingestion service. Thus, the data rates reflect continuous, sustained ingestion regardless of the actual data frame memory allocation. The maximum data rates achieved in the scenarios are summarized in Table 3.

Case	Scenario Name	Frame size (Mbytes)	Synchronous rate (Mbytes/sec)	Asynchronous rate (Mbytes/sec)
1 (100 cols)	SMALL_TABLES	0.370	0.046	0.178
2 (500 cols)	MODEST_TABLES	6.0	0.107	0.206
3 (500 cols)	STRING_TABLES	41	0.103	1.00
4 (1,000 cols)	LARGE_TABLES	24	0.061	0.143
5 (2,000 cols)	HUGE_TABLES	48	0.037	0.256

Table 3: ingestion rates for wide scalar tables

As seen above, the data rates here are the poorest. However, they are the most consistent and no overtly anomalous values are seen. Data frames with large numbers of columns apparently overwhelm the Datastore ingestion processing, even if the overall data size is relatively small (for continuous, sustained ingestion). Note that the asynchronous ingestion of wide data frames appears consistent around 0.2 Mbytes/second regardless of the width, for double values. The performance boost seen in the 2,000 column case is interesting, but so far unexplained.

A clue as to this performance issue is indicated in case 3. Rather than double, here scalar values are character strings with an average allocation of ~200 bytes (2 bytes per character plus 50 overhead). This is compared to case 2 where the scalars are doubles with an allocation of 24 bytes (in Java). Even with this large allocation difference case 3 outperforms its analogue by a factor 5 in the asynchronous case with no performance change in the synchronous case. The situation strangely suggests that the processing of character strings is more efficient than that of doubles,

either within Datastore ingestion service or the InfluxDB database, as archived string value were verified.

The third case is also interesting because it overwhelmed the data queue buffering of the ingestion API library (using a queue size of 100 gRPC data messages). However, the buffering operations performed as expected. The data provider is blocked from sending additional data until the Datastore ingestion service processes enough data to open the queue for additional streaming. The mechanism is intentional, to expose data providers to streaming back pressure isolating the Datastore from overactive data providers.

Array Table

This singular case is meant to compare to previous cases of continuous, sustained ingestion of wide data frames in Case 5 above. Here we ingest an equivalent amount of data in numeric array format.

1. 1 column of 1-dimension 2,700 Double value arrays; 1,000 rows, 38 Mbyte frames

A data frame containing a single column containing a one-dimensional array of double values is constructed. The array contains 2,700 double values, larger than the 2,000 columns of the widest scalar frame previously considered. Ten frames were sent to the Datastore.

The memory allocated for each array is measured at approximately 38 Kbytes. The measurement is obtained by serializing the numeric array and counting bytes. It is interesting to note that the measured serialized value for memory allocation is significantly less than that required for 2,700 individual Java Double values, about 65 Kbytes. This fact is most likely due to efficient object serialization within the Java library. The results of the test are shown in Table 4.

Case	Scenario Name	Frame size (Mbytes)	Synchronous rate (Mbytes/second)	Asynchronous rate (Mbytes/second)
1 (1 array)	ARRAY_TABLES	38	4.7	7.96

Table 4: comparison of wide scalar table and equivalent array

Thus, we see when sending an equivalent amount of data in an array format, as compared to a table format, the Datastore performance is increased over 30-fold for the asynchronous case and over 100-fold in the synchronous case. This finding may suggest that performance improvements can be made for table data with alternate implementations within the Datastore. However, during integrity testing it was found that numeric arrays are archived as single character strings, thus, the results do reflect full data processing of numeric values within the array.

7.3.5 Scenarios Cases: Addendum

Upon further testing of large data structures for asynchronous streaming (case 2 of "Data Arrays, Data Structure, and Images" in the previous subsection), it was found that the Datastore ingestion service is accepting data much faster than it is processing them. The ingestion service will accept approximately 30 gRPC messages containing data structures, of maximum size, as fast as they can be streamed. This would represent the maximum burst transmission. Thus, the ~50 Mbyte/second data rate reflects the burst data transmission rate without processing, as it includes data that was never fully processed by the Datastore. It has been determined that the full data structure is being

sent through the gRPC communications system. however, the ingestion service is simply not processing data structures correctly (see Subsection 8.2.3 case study).

When sending large numbers of data frames (~100) at a 10 Hertz rate the ingestion service eventually throttles then blocks. It will continue to accept frames afterward but at a much-reduced rate, in batches of about 30 gRPC messages. Thus, we see burst transmission of 30 frames, or about 270 Mbytes, after which we see the ingestion data processing rates. The best data rate estimate for this case appears to be approximately 5 Mbytes/second, which is consistent with other observed testing. (This case is included in APPENDIX B under "ADDENDUM".) It is also important to note that during this testing data rates as high as 100 Mbytes/second were achieved so long as the ingestion service was not processing data, that is simply receiving it. This behavior, although anomalous, indicates the gRPC communications is capable of transmitting data at the rate of 100 Mbytes/second. This fact was confirmed in the ingestion testing with the MPEX data simulator where data transmission rates were measured as high as 380 Mbytes/second (see Subsection 7.3.6).

Additionally for the above case, when attempting to send larger frames, with 50 to 100 rows having allocations of 450 to 900 Mbytes, the scenario fails to execute due to heap space limitations. This failure occurs because the data simulator allocates all scenario data before execution so as not to interfere with ingestion processing during scenario execution. The condition indicates that a substantial amount of data is being sent to the Datastore, it is simply not processing it in a timely fashion.

Another indication that the Datastore is not fully processing the received data occurs when attempting to close the data stream. Even when a "ready" acknowledgement is received by the Datastore, a timeout is encountered when attempting to close the stream. This suggests that the Datastore is still processing data and is unwilling, or unable, to respond to a close stream event. This same exception is seen in the second case of the "Heterogeneous Tables" cases. The situation suggests that the extreme data rate of 37 Mbytes/second seen there for the asynchronous case is also inaccurate, as the Datastore has not finished processing data.

The close-stream exception was also seen in data integrity testing and is described further in Subsection 8.1.1. It might be avoided through refactoring of the *datastore-admin* ingestion API library; however, it may be attributed to an ingestion service error. In any event, the issue should be addressed in future implementation.

7.3.6 MPEX Simulator Case

The MPEX data simulator available in the *mpex-spd* project emulates the Material Plasma Exposure eXperiment (MPEX) facility at Oak Ridge National Laboratory (ORNL) (12). The simulator produces a data stream from 64 scalar process variables (PVs) up to 1kHz, along with a separate stream of 240x640 pixel images from 4 Hz to 20 Hz. Performance studies using this simulator were executed using a series of "ramped" ingestion test cases. The MPEX data simulator connects to the Datastore ingestion service through the ingestion API library within the *datastore-provider-lib* project. Within that API library, rather than explicit *data frames*, data is transmitted in equivalent *batches* containing native Java types or gRPC data messages. Asynchronous communications were used for all ingestion tests here.

Two different test batteries were performed with the MPEX data simulator. One test battery consisted of pre-configured data frames ingested serially over a *single asynchronous data stream*.

The other test battery consisted of ingesting an equivalent amount of data concurrently over *multiple asynchronous data streams*. That is, the MPEX simulator test fixture can evaluate the Datastore ingestion for simultaneous, concurrent data streams.

Single Data Stream

Scalar data transmission rates and data processing rates were both tested using a ramp-up process depicted in Figure 22. Specifically, each test consisted of a predetermined number of data frames, or "batches", that were staged for ingestion. The number of batches ranged from 1 to 100. Each data frame (batch) contained 1,000 rows from 64 data sources (PVs) for a total of 64,000 double values. The full set of test frames is then transmitted at maximum rate. The number of test frames is gradually increased to contrast transmission rates and data processing rates. Ten such scenarios were performed starting with a frame count of 1 and finishing with a frame count of 100. This condition yields an overall data transmission from 64,000 doubles to 6.4 million doubles, or from 1.5 Mbytes to 153 Mbytes total allocation for Java double objects.

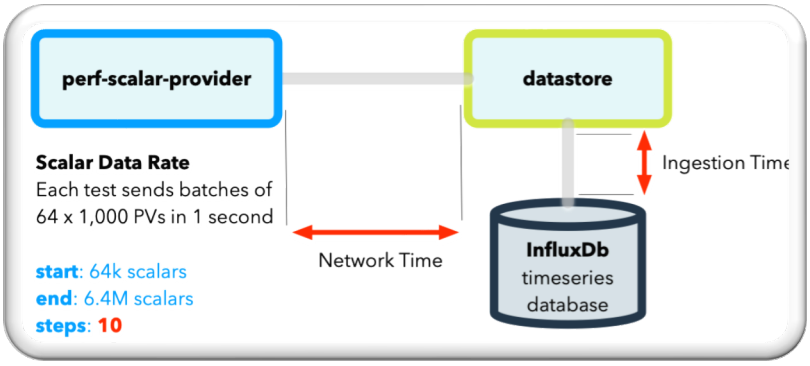


Figure 22: scalar ramping ingestion tests configuration

The MPEX simulator test platform was able to independently measure data transmission times and data ingestion times independently, as indicated in Figure 22. Thus, in addition to the burst rates measured for small ingestion sets, the transmission rates can also be determined.

The results of the scalar ramping tests are shown in the graph of Figure 23. The graphical depiction of the results illustrates that the transmission times over the network and the ingestion service processing times scale relatively well, with the size of the transmitted data. However, the processing of the scalar data is still well below our desired goal.

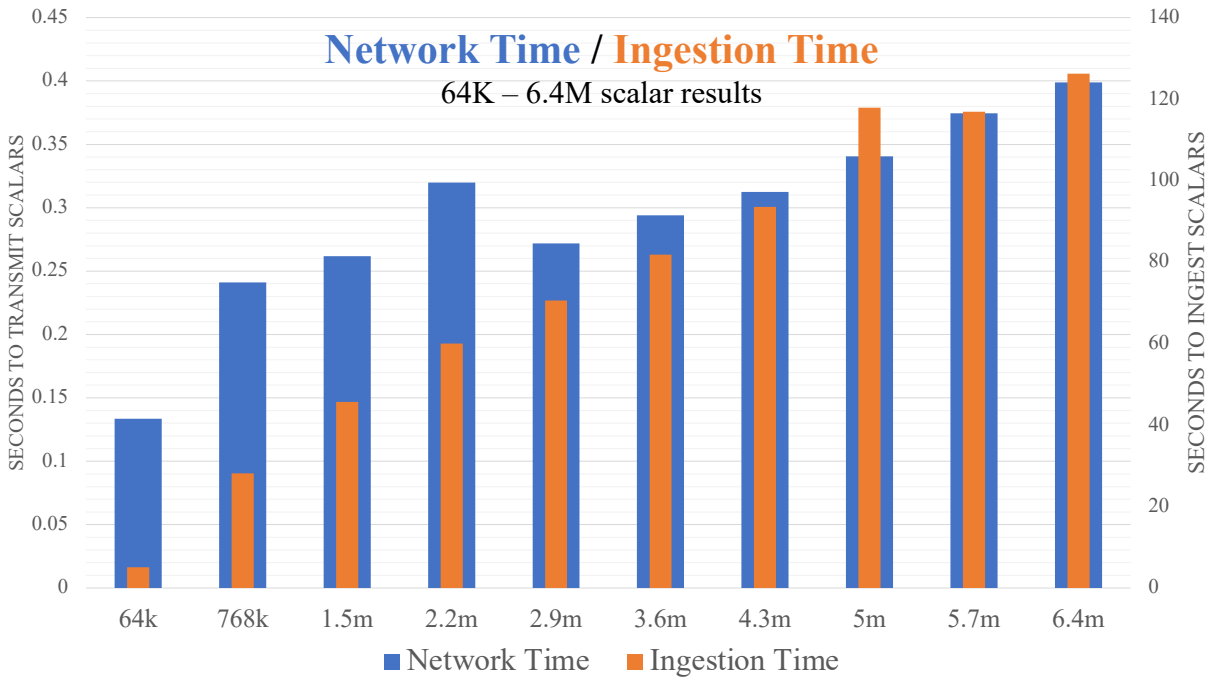


Figure 23: scalar ramping tests results

To summarize the single stream performance results we find that 50,600 doubles can be processed each second during continuous, sustained ingestion. This yields a maximum ingestion rate of 1.2 Mbytes/second, larger than that for wide scalar frames, but well below the objective of 100 Mbytes/second. However, we find that 16 million double values can be transmitted per second. That is, our maximum transmission rate on this platform is 380 Mbytes/second for Java double values, well above the 100 Mbytes/second rate.

Note that here the data frames are 64 columns wide. Comparing the above results to the scenarios performance testing, the 1.2 Mbyte/second data rates seen are between the 0.2 Mbyte/second rate obtained for wide data frames and the maximum 5.0 Mbyte/second rate seen for image data. Thus, the results appear consistent, as total data transmission is roughly on the same order. They also suggest that the increased performance in the narrow data frame case is likely due to the reduced processing of metadata.

The measured transmission rates of 380 Mbytes/second are encouraging, well above that which is required. Thus, the gRPC communications implementation appears to present no obstacle in our performance goals. Ingestion data processing and archiving should be the focus of future development efforts.

Concurrent Data Streams

In addition to the single-stream ramp-up testing, a set of concurrency tests was also performed using the MPEX data simulator. The test fixture is shown in Figure 24. Rather than a single data stream transmitting data frames with 64 scalar columns, here we have 8 data streams each transmitting frames with 8 scalar columns for a total of 64 data sources. Specifically, each data frame consisted of 8 columns and 1,000 rows for a total size of 8,000 double values. The frames were all transmitted at a continuous rate of 1 frame/second simulating ingestion of 64 data sources at 1 kHz sampling.

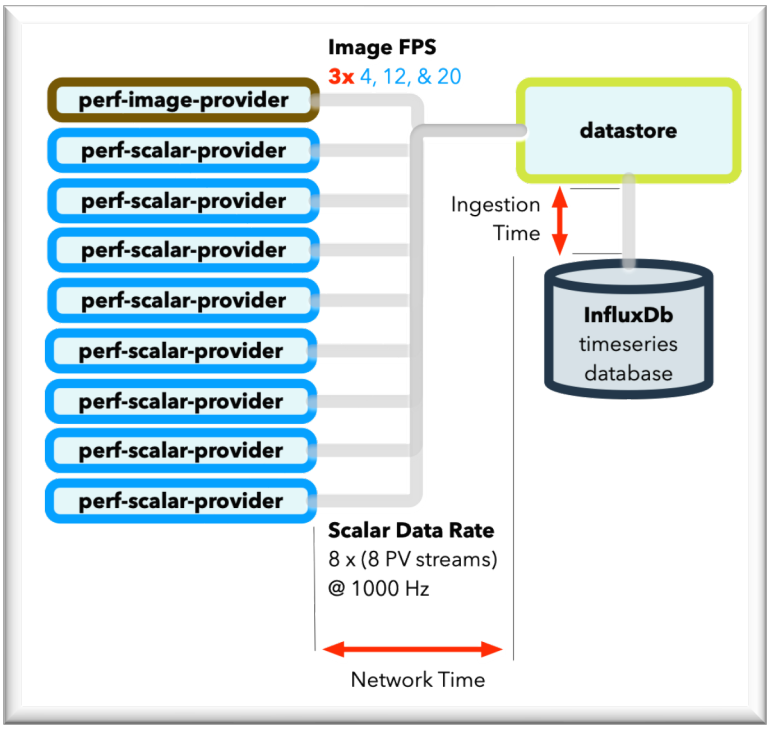


Figure 24: concurrent data streams test configuration

In addition to the scalar frames, an additional stream was established which transmits image data, specifically, three 240x640 pixel images. The image transmission rate was then ramped from an initial rate of 4 frames per second to a final rate of 20 frames per second.

To summarize the concurrency findings, for multiple scalar data streams the same data rates were observed as that for a single data stream. Thus, the results of the currency tests show no increase in performance when using multiple data streams. However, there was *no performance decrease* either. Thus, the Datastore ingestion service performs consistently under either circumstance.

During the concurrency testing, the data stream for the image data was isolated and performance numbers were determined. The results are shown in the graph of Figure 25. There the ingestion times for 100 image batches are shown with a transmission rate of 12 frames per second. These are times were measured while the 8 other concurrent scalar streams were active. In the graph we see an initial performance lag while the ingestion service is apparently initializing the image archiving process. After that time the ingestion processing performance settles into a steady state. The transmission times remain relatively constant except for initialization, where gRPC connections must be established.

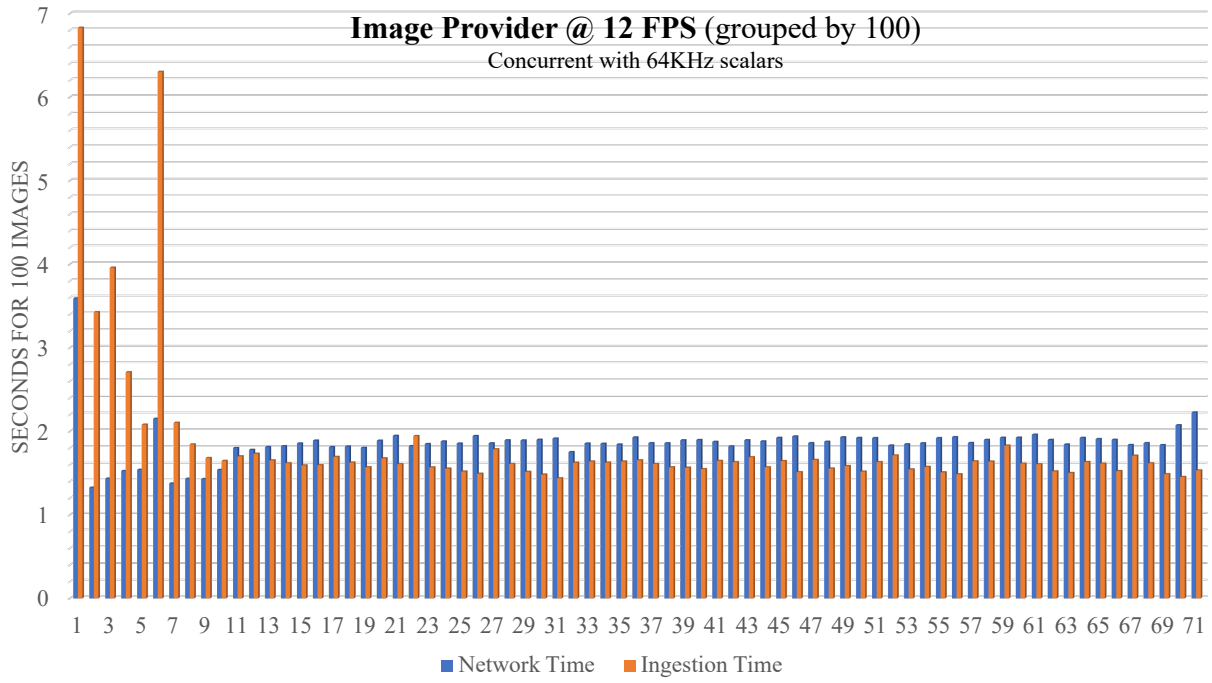


Figure 25: image data ingestion performance over time

Consider that each 240x640 image requires an allocation of 154 Kbytes. Thus, after the steady state is reached, we see an image ingestion rate of about 7.5 Mbytes/second (100 frames every 2s). This is consistent with the 5 Mbyte/second rate seen on the slower platform used for scenarios testing. However, the former rate was obtained while concurrently processing concurrent streams containing a total of 64,000 doubles in addition to the 12 images.

It should be noted that ingested rates for the three different image rates, 4 Hz, 12 Hz, and 20 Hz were relatively consistent. The 20 Hz case did see the best performance overall with ingestion times for 100 images slightly better than the 12 Hz case, sometimes approaching 1.5 seconds. Thus, image ingestion processing and archive appears to be the most efficient in all cases observed.

7.4 SUMMARY

A significant observation is that performance of the Datastore ingestion service greatly depends upon the shape of the data frames ingested. Specifically, continuous, sustained ingestion of wide data frames is significantly slower than that for narrow frames. This suggests that the processing of metadata is an expensive process for wide data frames. Another significant observation is that the best performance is seen for data frames containing few columns but large data, such as images and raw data. These data types are stored in system files on the host platform rather than the InfluxDB database, apparently the mechanism is more efficient for archiving large blocks of raw data. In all cases it is found that asynchronous streaming outperforms synchronous streaming as expected.

Of particular interest to the MLDP is the continuous, sustained ingestion of wide data frames containing thousands of process variables. This situation is essentially that of operation with the Aggregator system, requiring a data rate of 100 Mbytes/second to meet project goals. For the scenarios simulator the maximum rate observed for ingestion of wide data frames was 0.25

Mbytes/second (2,000 columns). The MPEX simulator obtained 1.5 Mbytes/second on the faster platform using narrower frames (64 columns). (Again, suggesting expensive metadata processing.) Thus, the performance of the Datastore ingestion service must be increased by at least a factor 100 to meet the stated project goals. Due to the significant underperformance seen in the 2,000-column case study, a 4,000-column case was never performed. (These case studies require significant real time executions.)

The data rates for continuous sustained ingestion of images were 5.4 Mbytes/second for the scenarios simulator and 7.5 Mbytes/second for the MPEX data simulator. The processing and archiving of images appear to be most efficient. Numeric arrays and data structures showed better performance for the scenarios simulator but the integrity of the archiving for these types is questionable (see Subsection 8.1.3). It is interesting to note that the ingestion of wide frames (500 columns) containing character strings had data rate of 1 Mbyte/second, about 5 times faster than for equivalent frames of double values. Thus, it also appears that the processing of character strings is more efficient than that for double values.

Although the Datastore ingestion service prototype has overall underperformance issues, concurrent ingestion does not appear to be a concern. Under testing with concurrent, asynchronous data streams the Datastore performance was consistent with that of a single data stream containing an equivalent amount of data. Additionally, the gRPC communications system is not factor in ingestion performance. Data transmission rates as high as 380 Mbytes/second were observed in MPEX simulator testing.

7.5 CONCLUSIONS

Overall, the current implementation of the Datastore ingestion service functions as specified, but with limitations. The most notable limitation is the inability to explicitly ingest timestamped rows within data frames (i.e., the use of timestamp lists). This issue must be addressed in future efforts.

The gRPC maximum message size limitation can be overcome using data binning; however, it may be worth investigating if larger message sizes are possible as it was found that data bins with the largest possible size perform best. Larger gRPC message sizes may further increase performance.

Another issue of less significance is the mismanagement of snapshot UIDs when archiving data. This error is seen during synchronous data transmission; sometimes snapshot UIDs are not unique and sometimes multiple snapshot UIDs are generated for the same data frame. Additionally, the ingestion service apparently requires unique snapshot timestamps to generate snapshot UIDs, this is a brittle implementation. These issues should eventually be addressed as it affects data integrity within the Datastore.

The current performance of the ingestion service does not meet the requirements of a 100 Mbyte/second data rate. Unfortunately, the worst performance for Datastore ingestion occurs for the target objective of wide data frames containing thousands of columns. The maximum data rates for wide data frames are on the order of 0.2 Kbytes/second. As testing was done on development platforms, these rates can certainly be improved with hardware by using dedicated host servers and high-speed data networks. However, simply by improving implementation the ingestion service should process table data at a much faster rate. For example, advanced features of the InfluxDB system were not utilized, such as multi-partitioning with multiple write heads. This should be employed in future testing.

The processing and archiving of image data appears most efficient under sustained, continuous ingestion. Data rates between 5 and 7 Mbytes/second are observed in this case. Since images are stored as system files within the host platform this condition suggested that performance can be improved using direct file archiving rather than the InfluxDB database. However, this approach would entail a major redesign of the Datastore archive requiring intense development effort. A more immediate approach would be to investigate the advanced features of the InfluxDB system as described above.

During ingestion testing, data transmission rates were seen in far excess of 100 Mbytes/second, up to 380 Mbytes/second. This transmission rate, observed on a limited testing platform, indicates that the gRPC communications are not a limitation for target performance.

8. DATA INTEGRITY EVALUATION

Extensive data integrity testing of the Datastore archive was performed. The objective is to ensure that 1) snapshot data and metadata is correctly ingested, 2) ingested snapshot data and metadata is correctly archived, and 2) archived data is correctly retrieved during query operations. Thus, data integrity evaluations are essentially a critique of both the ingestion and query services, as well as the data archive itself.

To perform data integrity testing it was necessary to have both a functioning ingestion API library and a query API library. That is, it was first necessary to confirm that the API libraries operated as expected and that any integrity issues were to be attributed to the Datastore core services and data archive. API library operations were confirmed before the data integrity testing.

The ingestion and query API libraries with the *datastore-admin* project were used exclusively for the data integrity testing. It is a natural choice to include data integrity testing capabilities within this project since these services are an important aspect of Datastore administration. Additionally, the project already contained most of the necessary tools for data integrity testing. Specifically, one must perform both data ingestion and query of the Datastore archive to ensure correct data throughput. Furthermore, the project also contains API libraries for direct inspection and management of the InfluxDB and MongoDB databases. Thus, one can create integrity tests that ingest known data then directly inspect it within the databases, determining the integrity of ingestion operations. Likewise, one can create query integrity tests by identifying InfluxDB and MongoDB data within the archives then directly requesting it through the query service for comparison.

Both the metadata and snapshot data archives were tested. Metadata testing was straightforward as a single test fixture was constructed containing all aspects of available metadata. The fixture supplied pre-defined metadata to populate the archive and then comparisons were made with the archived metadata. Snapshot data testing was more complex. For each test the datastore archive was cleared then populated with a known set of data frames of a particular data type. The archive was inspected for integrity, then the entire archive was queried for all data to ensure query operations were successful.

8.1 METADATA

A battery of data integrity tests was executed on the metadata query service. The Datastore was populated with a predetermined set of data frames with known metadata and attributes. The data archive was then inspected for integrity using the MongoDB database API within the *datastore-admin* project. Finally, all the available metadata requests were made to the Datastore query service to recover the metadata records and compared to the known values.

A significant portion of the metadata integrity testing consists of verification of the metadata query operations themselves. Such testing might be more appropriately included in Chapter 9 concerning query service evaluation. However, it was easier to design a test fixture that simultaneously evaluated the operation of the metadata query service and the integrity of the results. Moreover, query service validation, in general, appears to be part of data integrity testing, as it is impossible to conduct the latter without use of the former. Including query service validation here also allows us to focus primarily on query performance issues within Chapter 9.

8.1.1 Operational Errors

The most serious issue with the metadata service concerns multiple snapshot ID generation, which was previously identified in the ingestion evaluation. Otherwise, it was found that all metadata was stored and recovered correctly except for timestamps.

The Datastore core service has general operational issues with timestamps. This issue is seen as a implementation error within the Datastore core ingestion service, rather than a data integrity issue. Hence the timestamp issue is also seen in the metadata. As covered in more detail below, incorrect timestamps were seen for both the PV records queries and the snapshot time range queries. The problem of incorrectly assigning timestamp values is covered in further detail within Subsection 8.2.1 of this chapter. No other serious errors were seen within the metadata operations.

8.1.2 Test Fixture

Within the *datastore-admin* project a library was implemented to create a consistent test fixture for metadata query service testing. The test fixture contains a utility for clearing the Datastore archive and then *monitoring all ingestion activities within the testing process*. While monitoring the ingestion data stream, it identifies and records all Datastore metadata seen within the data stream. The external metadata cache is then used for comparison with the metadata archive and the metadata recovered from the query service.

The text fixture can be configurated to create a variety of metadata ingestion and query scenarios. Recall that synchronous communications are always used for metadata queries, but data ingestion may be performed either synchronously or asynchronously. The following process is used to initialize the Datastore for all metadata tests:

1) The Datastore archive is completely cleared then initialized to a common archive state.
2) Three separate data providers are registered with the Datastore
3) The Datastore is populated with four distinct data frames using synchronous communication:

    1. a frame of 5 scalar columns, all different types; size 10 rows
    2. a frame of 1 array column; size 10 rows
    3. a frame of 1 data structure column, size 5 rows
    4. a frame of 10 double columns and 5 numeric array columns; size 100

The specific configuration is used for the above process:

- The first three frames are always identical; their timestamps are fixed.
- The values within the last frame are random; their timestamps reflect the creation instant.
- The following data providers are used:
    - Frames 1 and 4 are populated by Data Provider 1
    - Frame 2 is populated by Data Provider 2
    - Frame 3 is populated by Data Provider 3

Synchronous ingestion is used to populate the archive. User attributes for all data frames are distinct. The data source names (i.e., PV names) within, and across, all data frames are distinct. The timestamps across all data frames are distinct.

8.1.3 Cases

Once the Datastore archive is initialized to the common state described above, separate tests were performed for each metadata type. That is, each test requested specific metadata records and compared the results to the known values cached by the library ingestion utility. The output of the metadata tests is found in APPENDIX C.

PV Record Queries

After the archive was initialized with the test fixture data, the open request for all PV names correctly returned all PV data sources that contributed to the archive. Additional testing was done for single PV names and some regular expressions. In all test cases the metadata query mechanism correctly identified PV name queries. Specifically, `PVRecord` instances were returned containing the correct PV name attribute. All other record properties were also correct except the timestamp properties (i.e., fields `firstTimestamp` and `lastTimestamp`).

The metadata query service was also able to correctly identify all attributes names associated with all data sources within the test fixture.

Annotations Queries

There were no annotated queries (i.e., "named queries") within the test fixture and, thus, none were available for query. An open query for annotation queries was performed and none were returned, as expected.

Data Providers

Currently there is no direct mechanism to query snapshot data providers through the metadata query service. Snapshot data provider UID records can be obtained through the snapshot data ingestion API during provider registration. Correct snapshot data provider UIDs were confirmed with the library's ingestion utility.

Snapshot Queries

Snapshot queries are by far the most complex of the metadata query service. This complexity is managed through the `SnapshotRequest` utility described in Subsection 4.9.2 Thus, the battery of snapshot query tests was performed on this utility.

The Datastore timestamping error is seen in the snapshot metadata, most predominantly in the time range query. (This condition is also covered in Subsection 9.3 for snapshot data requests.) Interestingly, the snapshot timestamp (i.e., the "trigger time") is being stored and returned correctly. The timestamps assigned to snapshot data are being archived incorrectly. Thus, what is observed in the snapshot records are correct snapshot timestamps, but erroneous first and last timestamps for the snapshot data. Thus, the timestamping error within the Datastore core can be isolated to the snapshot data archiving mechanism.

The snapshot data timestamps seen within the Datastore archive are recovered accurately by the query service. That is, although snapshot data timestamps are being misassigned by the ingestion service, the query service is accurately returning the incorrect values. Thus, further indication that the error can be isolated to the ingestion archiving mechanism.

It should be noted that within the test fixture data, all snapshot timestamps occurred before the first timestamp of the snapshot data. That is, the trigger time of the snapshot always preceded the acquisition times of the snapshot data. Therefore, the first timestamp attribute within any snapshot record was always identical to the snapshot timestamp. The last timestamp attribute does equal the timestamp assigned to the last data value of the snapshot within the archive (although it is incorrect). All other snapshot queries obtained correct results, as outlined below.

The test data was configured so that all snapshots had unique attribute values, although they may have shared attribute names. The Datastore was able to correctly identify a requested snapshot by its unique attribute values in each test case. When requesting all attributes assigned to all snapshots, the Datastore correctly returned all attribute names assigned across all snapshots.

The data management ingestion utility also records all snapshot UIDs generated by the Datastore upon frame ingestion. In each snapshot request where snapshots are directly requested by UID the Datastore returned the correct snapshot. Snapshot records were requested both by single UID value and using a list of UIDs (which returns a list of records). The data sets used in the metadata test fixture did not invoke the snapshot UID errors seen when performing the ingestion testing (described in Subsection 7.2.2). Thus, all tests performed here were successful.

It is possible to request all snapshot records that contain specific data source (PV) names, or name regular expressions. Within the test data, all data source names were unique. Partial testing of snapshot requests by data source name was performed. When requesting a single source name and requesting all source names the metadata query service performed correctly. Testing for snapshots that share a common data source name was not performed.

The Datastore query service currently contains no mechanism for requesting snapshot records by the data provider (i.e., via the UID of the provider producing the snapshot).

8.1.4 Summary

Except for the timestamp misassignments, all metadata is being ingested, archived, and retrieved correctly. Thus, there are no data integrity issues for metadata. The timestamps misassignments occur within the snapshot data archive. This timestamping has been isolated to the Datastore ingestion service. The snapshot UID error seen in ingestion testing (Subsection 7.2.2) did not materialize.

8.2 SNAPSHOT DATA

A battery of data integrity tests for snapshot data ingestion, archiving, and query operations was implemented within the *datastore-admin* project. The test library was used to determine that 1) timestamps were accurately ingested, 2) snapshot data was being accurately ingested, and that 3) snapshot data was being properly retrieved. Issues were detected in all three categories. The raw output of the snapshot data testing is listed in APPENDIX D.

8.2.1 Operational Errors

There are operational issues with asynchronous ingestion and with timestamp integrity. These issues are described separately below.

Asynchronous Ingestion

An initial attempt was made to use asynchronous ingestion for snapshot data integrity testing, since asynchronous ingestion is much faster. However, attempting to use asynchronous ingestion

produced several query errors, including responses containing partial results, responses containing null values, total data request failure, and crashing of the Datastore ingestion service. These issues appear to be the consequence of simultaneous use of the ingestion and query services within the Datastore core.

The Datastore appears to continue ingestion processing after asynchronous ingestion has terminated, that is, after acknowledgement of a closeStream() operation. Or it continues to maintain gRPC resources after a closeStream() operation. Thus, either the Datastore continues to process ingested data after acknowledging data receipt and stream closure, or it simply fails to release resources in a timely fashion. In the former case one could expect collisions with query operations if data is still being processed to the archive, which is the likely case. In any event, the snapshot data query operations were inconsistent after using asynchronous ingestion, sometimes crashing the Datastore ingestion service, sometimes corrupting the Datastore archive, and once corrupting the entire InfluxDB installation. This issue must be addressed in future development efforts. The solution could be as straightforward as sending a close stream acknowledgement only after all ingested data processing and archiving has been confirmed.

Timestamps

Most concerning is the fact that *snapshot data timestamps are not correctly assigned* within the archive. There is a lag between the timestamp of the incoming data and the timestamp assigned to the snapshot data within the Datastore archive. This time difference is not consistent. Time differences from as small as seconds to as large as days have been observed during testing. However, the difference is always a lag between the true timestamp and the recorded timestamp. Upon inspection of the InfluxDB time-series database, this time lag is being archived. Thus, the time lag must be occurring in the pre-archive processing. The query service does, however, operate correctly. Retrieval of archived timestamps is correct, that is, the query results contain the timestamps within the InfluxDB archive.

Additionally, issues with the *snapshot data timestamp ingestion* have been previously addressed regarding the time reference gRPC message. Recall from Subsection 7.1.1 that use the *timestamp list* feature of the ingestion service is not practical. It is not possible to directly assign individual timestamps to row data within data frames. One must use a *timestamp iterator* where an initial timestamp is assigned, and all subsequent data is assumed to be acquired at a common sampling rate. Thus, all testing was performed using timestamp iterators.

The timestamp assignment error is seen in all snapshot data integrity tests. It is likely that the error is a simple implementation "bug" within the ingestion service. However, it creates a corrupt data archive and must be located and corrected.

8.2.2 Test Fixture

The *datastore-admin* library contains services for defining and creating different testing resources. One such feature is the ability to create pre-defined data frame instances from YAML definition files. The parser service supports YAML formats which realize all possible data types and structures supported by the Datastore core services. Thus, any type of data frame can be tested in this manner. The facility was used to create data integrity tests for a variety of different data frames, three of which are covered here.

For the snapshot data integrity testing a fixture was implemented that first clears the Datastore of all archived data and metadata. Predefined, individual data frames are then sent to Datastore using

the test facility described above. The open query is used to retrieve the entire Datastore snapshot data archive, whose result should be that of the single frame ingested.

8.2.3 Cases

The results for three test cases, each containing different types of data frames, are presented. They reflect all snapshot data integrity issues seen in the prototype so far. All test data was sent to the Datastore using synchronous data streams due to the operational errors with asynchronous ingestion described above. The data integrity tests produced identical results for both synchronous and asynchronous queries.

Consider the following cases:

1. Mixed Scalar Data: frame with columns of Booleans, integers, floats, doubles, strings
2. Array Data: frame with a single column of array data
3. Structured Data: frame with a single column of 5 different data structures.

We address each case separately in succession.

Mixed Scalar Data

An excerpt of the mixed-scalar data-integrity test output is shown in Figure 26. The incoming data frame is shown at the top under the "Test Data Frame:" header. Note that all metadata of the incoming data frame is also shown above the frame. The result of the open query request is shown at the bottom under the "Open Query Result:" header. The test frame at the top contains six columns, one column containing the timestamps and five columns containing the actual snapshot data from data sources TEST-PV00, TEST-PV01, TEST-PV02, TEST-PV03, and TEST-PV04. Each data column contains data of a different scalar type: Boolean, integer, float, string, and double, respectively, for

Test Data Frame:
Snapshot Data Provider UID = null
DataFrame UID = null
DataFrame Timestamp = 2022-10-01T01:23:40.100Z
DataFrame Attributes = {duration=10000000000, period=1000000000, file=test-dataframe-scalars.yml, name=Test DataFrame Scalars, type=test data, frequency=1}

timestamp	TEST-PV00	TEST-PV01	TEST-PV02	TEST-PV03	TEST-PV04
2022-10-01T01:23:45.100Z	true	0	0.0	str0	0.0
2022-10-01T01:23:46.100Z	true	1	0.1	str1	0.01
2022-10-01T01:23:47.100Z	true	2	0.2	str2	0.02
2022-10-01T01:23:48.100Z	true	3	0.3	str3	0.03
2022-10-01T01:23:49.100Z	true	4	0.4	str4	0.04
2022-10-01T01:23:50.100Z	true	5	0.5	str5	0.05
2022-10-01T01:23:51.100Z	true	6	0.6	str6	0.06
2022-10-01T01:23:52.100Z	true	7	0.7	str7	0.07
2022-10-01T01:23:53.100Z	true	8	0.8	str8	0.08
2022-10-01T01:23:54.100Z	false	9	0.9	str9	0.09

Open Query Result:

timestamp	TEST-PV00	TEST-PV02	TEST-PV01	TEST-PV04	TEST-PV03
2022-10-01T01:24:00.100Z	true	0.0	0	0.0	null
2022-10-01T01:24:01.100Z	true	0.1	1	0.01	null
2022-10-01T01:24:02.100Z	true	0.2	2	0.02	null
2022-10-01T01:24:03.100Z	true	0.3	3	0.03	null
2022-10-01T01:24:04.100Z	true	0.4	4	0.04	null
2022-10-01T01:24:05.100Z	true	0.5	5	0.05	null
2022-10-01T01:24:06.100Z	true	0.6	6	0.06	null
2022-10-01T01:24:07.100Z	true	0.7	7	0.07	null
2022-10-01T01:24:08.100Z	true	0.8	8	0.08	null
2022-10-01T01:24:09.100Z	false	0.9	9	0.09	null

Figure 26: data integrity test - mixed scalar data

columns TEST-PV00 to TEST-PV04. Note that the bottom data table containing the query result does not necessarily maintain the original column order. This condition is typical within Datastore query operations.

When retrieving mixed scalar data, *character strings are not properly recognized*. This is seen within the result column TEST-PV03 containing character string data; the string values within the query request are all returned as null values. Inspection of the InfluxDB archive shows that string values are being stored correctly. However, the query service is not recognizing them when retrieving the request. Upon inspection of the gRPC data messages, query results are left empty for the character string values. Thus, the integrity error can be isolated to the Datastore query service. The Boolean and mixed numeric data within the same data frame was ingested, archived, and retrieved correctly.

Also seen in Figure 26 is the appearance of the timestamping error. In this case all timestamps of the requested data occur 15 seconds after the original timestamps. This time lag is uniform across all timestamps in the request results. The misassigned timestamps appear within the InfluxDB time-series database, revealing that they are being incorrectly assigned during ingestion. The Datastore query service returns the timestamps accurately as they appear in the InfluxDB database, thus the query operation appears to be performing correctly with regards to timestamps.

Array Data

The Datastore can store one-dimensional arrays of double values. The ingestion service realizes such arrays as lists of double values, and they are transmitted as such within their respective gRPC data messages. Likewise, the ingestion APIs also represent one-dimensional numeric array data as lists of doubles (i.e., of Java type List<Double>). For clarity, we defined a one-dimensional numeric array A of length N as follows:

$$A := [\, x_1, x_2, \ldots, x_N \,],$$

where the x_i are double values. This is also the format used to represent numeric arrays within the YAML definition files in the *datastore-admin* testing library.

An excerpt of the numeric-array data-integrity testing output is shown in Figure 27. The top of the excerpt contains the test data frame and its metadata as before. The incoming data frame contains two columns: a column of timestamps and a single data column, TEST-Array-PV00, containing one-dimensional array values. The frame has row size 10, that is, there are 10 arrays each having 10 values, for a total of 100 double values within the frame. Additional output is included in this excerpt to demonstrate some additional features of the test facility. At the top we see the query time, the request size, the achieved data rate, and the assigned name of the test (testRequestData2Sync_IntegrityArrays). At the bottom we see that the facility checks for missing data sources, timestamp difference, snapshot data value differences, along with other integrity verification checks not included in the excerpt.

```
TEST: testRequestData2Sync_IntegrityArrays FROM
com.ospreydcs.datastore.admin.model.IQueryServiceDataTest
  Query time (seconds)    : 0.031804
  Request size (bytes)    : 1660
  Data rate (bytes/second): 52194.692491510505
Test Data Frame:
Snapshot Data Provider UID = null
DataFrame UID = null
DataFrame Timestamp = 2022-10-03T01:23:40.100Z
DataFrame Attributes = {duration=10000000000, period=1000000000, file=test-dataframe-
arrays.yml, name=Test DataFrame Arrays, type=test data, frequency=1}
timestamp      TEST-Array-PV00
2022-10-03T01:23:45.100Z      [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
2022-10-03T01:23:46.100Z      [1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9]
2022-10-03T01:23:47.100Z      [2.0, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9]
2022-10-03T01:23:48.100Z      [3.3, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9]
2022-10-03T01:23:49.100Z      [4.0, 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9]
2022-10-03T01:23:50.100Z      [5.0, 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9]
2022-10-03T01:23:51.100Z      [6.0, 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, 6.9]
2022-10-03T01:23:52.100Z      [7.0, 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.7, 7.8, 7.9]
2022-10-03T01:23:53.100Z      [8.0, 8.1, 8.2, 8.3, 8.4, 8.5, 8.6, 8.7, 8.8, 8.9]
2022-10-03T01:23:54.100Z      [9.0, 9.1, 9.2, 9.3, 9.4, 9.5, 9.6, 9.7, 9.8, 9.9]

Open Query Result:
timestamp      TEST-Array-PV00
2022-10-03T01:24:00.100Z      [0, null, null, null, null, null, null, null, null, null]
2022-10-03T01:24:01.100Z      [1, null, null, null, null, null, null, null, null, null]
2022-10-03T01:24:02.100Z      [2, null, null, null, null, null, null, null, null, null]
2022-10-03T01:24:03.100Z      [null, null, null, null, null, null, null, null, null, null]
2022-10-03T01:24:04.100Z      [4, null, null, null, null, null, null, null, null, null]
2022-10-03T01:24:05.100Z      [5, null, null, null, null, null, null, null, null, null]
2022-10-03T01:24:06.100Z      [6, null, null, null, null, null, null, null, null, null]
2022-10-03T01:24:07.100Z      [7, null, null, null, null, null, null, null, null, null]
2022-10-03T01:24:08.100Z      [8, null, null, null, null, null, null, null, null, null]
2022-10-03T01:24:09.100Z      [9, null, null, null, null, null, null, null, null, null]


Query results are missing providers: []
Timestamp maximum time difference: PT15S
Query results differ at the following locations:
  TEST-Array-PV00: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Figure 27: data integrity test - array data

The table at the bottom of Figure 27 shows the result of the open query request, under the header "Open Query Result:". There we see an almost total failure of the data integrity test for numeric array data, only 9 of the 100 scalar values were recovered. However, regarding these 9 values, the data types were *long integer* rather than *double*, so essentially the operation fails altogether.

Upon inspection of the InfluxDB archive, *one-dimensional array types are archived as character strings*. Specifically, consider an array with value [x_1, x_2, …, x_N], where the x_i are the double-valued elements. It is archived in the format "[str_1, str_2, …, str_N]" where str_i is the string representation of the double value x_i. When retrieving the array elements, the Datastore query service *fails to parse the string values correctly*. As a result, the query result is either incorrect, or fails altogether. For example, when ingesting the array with value [0.0, 0.1, …, 0.9], the archived InfluxDB representation is the character string "[0, .1, …, .9]". The returned query result is [0, null, null,…, null]. And upon closer inspection the returned data types are [Long, null, null, …, null]. Thus, one possibility is that the parsing process is initially open, parsing numeric values into whichever format fits the first representation; then all subsequent numeric parsing would fail in this case.

To correct the integrity regarding numeric array data archiving a more robust implementation must be adopted. The current implementation is frail. Even if the string representation contained in the

archive was parsed correctly, the loss of numeric precision is inevitable. The preservation of numeric precision is crucial to many data science applications.

Finally, we again see the appearance of the timestamping error within snapshot data. As with the Mixed Scalar Data test case, all the timestamps given to the requested data lag 15 seconds behind the original timestamps. The 15 second lag is also seen within the InfluxDB archive as before. Although, the same time lag was seen in the two integrity tests, this condition is atypical, usually different time lags appear without any consistency. It should be noted that the two cased were run consecutively in the same test battery.

Structured Data

The Datastore core recognizes complex data structures in both the ingestion and query services. Moreover, the transmission of data structures is well-defined within the *datastore-grpc* project communications library. The general format of a data structure S is defined as follows:

$$S := \{ n_1=v_1, n_2=v_2, \ldots, n_N=v_N \},$$

where the n_i are string-valued *field names* and the v_i are *field values* of any supported type. This is also the format used to represent data structures within the YAML definition files in the *datastore-admin* testing library.

The field values v_i may be of any data type supported by the Datastore, including other data structures. Thus, any field value v_i may itself be a data structure which, in turn, may contain other sub-structures as fields. Thus, it is possible to form very complex data structures recognized by the Datastore. Data structures can have a tree-like format supporting recursive operations. At least in principle, it is possible to ingest, archive, and query complex data structures of arbitrary width and depth.

A sample excerpt for the structured-data integrity test is shown in Figure 28. Again, the incoming test frame is shown at the top, along with its metadata, while the result of the open query request is shown in the data table at the bottom. The incoming test frame has two columns, one column of timestamps and one column of data structures, labeled `TEST-Structure-PV00`. The test data frame has 5 rows, that is, there are 5 data structures within the frame. Note that each data structure has a different form, and forms become increasingly complex. The Datastore will respect data structures with different formats produced by a single data source, so long as all field names are unique.

We should mention that structured-data integrity tests were also performed with data frames containing data structures from multiple data sources, that is, multiple columns. Each different source produced a data structure with a unique format, but with varying field values. However, the results were essentially identical to those found using the simple frame shown in the excerpt.

As seen in the output of Figure 28, there are serious issues with data structures, including their field names, their archiving, and their retrieval. Most of these issues appear to be a direct consequence of the format used to archive the data structure within the InfluxDB database. The storage method is convoluted and involves archiving field values in both double and string formats. However there also appear to be direct issues within the query service implementations

```
TEST: testRequestData3Sync_IntegrityStructs FROM
com.ospreydcs.datastore.admin.model.IQueryServiceDataTest
  Query time (seconds)    : 0.030208000000000002
  Request size (bytes)    : 240
  Data rate (bytes/second): 7944.915254237288
Test Data Frame:
Snapshot Data Provider UID = null
DataFrame UID = null
DataFrame Timestamp = 2022-10-04T01:23:40.100Z
DataFrame Attributes = {duration=10000000000, period=1000000000, file=test-dataframe-structs.yml,
name=Test DataFrame Structures, type=test data, frequency=1}
timestamp      TEST-Structure-PV00
2022-10-04T01:23:45.100Z      {f1=0.0}
2022-10-04T01:23:46.100Z      {g1=1.0, g2=2.0, g3=3.0, g4=4.0, g5=5.0}
2022-10-04T01:23:47.100Z      {h1=1.0, h2={h21=0.1, h22=0.2}, h3={h31=0.1, h32=0.2, h33=0.3}}
2022-10-04T01:23:48.100Z      {i1=1.0, i2={i21=0.1, i22=0.2}, i3={i31=0.1, i32={i321=0.01,
i322={i3221=0.001, i3222=0.002}}}}
2022-10-04T01:23:49.100Z      {j1={j11={j111={j1111={j11111=1.0E-5, j11112=2.0E-5, j11113=3.0E-
5}}}}}

Open Query Result:
timestamp      TEST-Structure-PV00
2022-10-04T01:23:50.100Z      0.0
2022-10-04T01:23:51.100Z      {g1=1.0, g2=2.0, g3=3.0, g4=4.0, g5=5.0}
2022-10-04T01:23:52.100Z      {h1=1.0, h2={h21=null, h22=null}, h3={h31=null, h32=null, h33=null}}
2022-10-04T01:23:53.100Z      {i1=1.0, i2={i21=null, i22=null}, i3={i31=null, i32=null}}
2022-10-04T01:23:54.100Z      {j11=null}


Query results are missing providers: []
Timestamp maximum time difference: PT5S
Query results differ at the following locations:
  TEST-Structure-PV00: [0, 2, 3, 4]
```

Figure 28: data integrity test - data structures

Data structures with only a single depth and multiple width maintain full data integrity through the Datastore ingestion and recovery process. This is seen in the second row of the result table within Figure 28, the data structure with field names `g1`, …, `g5` is correctly recovered. However, it is the only one. Note that even the very simple data structure `{ f1=0.0 }` is incorrectly recovered; the field name is lost.

To be more precise, if a data structure contains multiple fields and each field value v_i is a scalar, then the data structure is retrieved correctly. This is the only case which is recovered correctly. If a structure contains a single field, the field name is lost. If a multi-field structure has a field with value containing a substructure, then the values within that substructure are lost. For example, say field value v_1 contains a substructure of the form $v_1 = \{f_{11}{=}v_{11}, f_{12}{=}v_{12}\}$; then the values v_{11} and v_{22} are lost, the recovered substructure is $v_1 = \{f_{11}{=}\text{null}, f_{12}{=}\text{null}\}$ (note the field names remain). Any substructures at greater depth are lost completely.

Upon inspection of the InfluxDB archive it is seen that the Datastore flattens data structures. If a top-level field contains a scalar value, that value is stored as a scalar. If the value contains a more complex data type, including substructures, then it is stored as a character string. The Datastore query service is apparently failing to parse the character strings correctly. Moreover, in the case

of the single field structure, the structure is archived correctly but the query service fails to recover the field name.

Again, we see the timestamp misassignment error in the output of Figure 28. In this case the timestamps of the requested data lag the incoming data by 5 seconds. During the integrity and query service testing we have seen time lags in the ranges of hours and sometimes days.

When inspecting the gRPC messages containing the data structures, it was found that the message contents were correct for the ingestion side. It was difficult to confirm correctness on the query side as the archive was incorrectly parsed. However, this fact does validate the correct handling of complex data structures within the gRPC communications mechanism.

The Datastore requires a more robust implementation for archiving data structures. An implementation that does not involve string conversion of numeric values would be preferrable. Archiving data structures in system files rather than within InfluxDB may be an option. Perhaps the use of HDF5 file formats would be a viable solution for this case.

Image Data

The Datastore is capable of processing images. Images are realized as a type consisting of a raw byte array containing the image data, and a format specifier indicating the format of the image data (e.g., JPEG, GIFF, PNG, etc.). The ingestion API within the *datastore-admin* project also includes attributes for specifying dimension information for user-specific imagines, but the Datastore core does not yet support user formats.

The Datastore ingestion and archiving of images is straightforward. They are transmitted as byte arrays augmented with an image format enumeration. They are then archived as system files on the host platform using a file extension indicating the image format. File location information and metadata is stored in the Datastore archive for later query and retrieval. The query service is then able to recover images and properties using timestamps, data sources, and other metadata.

No extensive integrity testing of image data was performed, that is, the accuracy of the raw data was not tested. However, it was verified that the byte arrays of proper size were being archived and recovered.

Raw Data

Raw data consists simply of byte arrays. The Datastore treats raw data in an analogous fashion as with raw image data. Specifically, raw data is transmitted directly as a byte array message then archived as a system file within the host platform. It was confirmed that raw data of proper size was stored and recovered by the Datastore, but the accuracy of the data was not evaluated.

8.2.4 Summary

Several important errors and issues were identified in the snapshot data integrity testing. They are summarized here as follows:

- Asynchronous ingestion and query operations cannot be used simultaneously, or immediately after a close stream operation. The exact wait time before query operations can be initiated after an asynchronous close stream operation is unknown.
- Timestamps of snapshot data are being incorrectly assigned by the ingestion service.
- Scalar character strings are not recognized by the query service.

- Numeric arrays are archived as single character strings. The string values are parsed incorrectly by the query service. This implementation may be efficient, but it is brittle and results in a loss of numeric precision. Transport of numeric arrays through the Datastore generally fails altogether.
- Data structures are flattened during archiving, stored as both float values and character strings. The query service fails to recognize all but one type of data structure, the single-depth multiple-field structure.

The above issues require correction and/or refactoring of both the Datastore ingestion and query service. Additionally, more robust storage mechanisms for both numeric arrays and data structures should be considered.

8.3 CONCLUSIONS

Except for the snapshot data timestamps, all metadata is being ingested, archived, and retrieved correctly. The trigger time for a snapshot data record is being archived correctly. Thus, there are no data integrity issues for metadata as timestamps are a property of snapshot data. The timestamp error seen within the Datastore core operation has been isolated to the ingestion service. The snapshot UID error seen in ingestion testing (Subsection 7.2.2) did not materialize.

Regarding snapshot data integrity, several issues were observed. The most pressing, although likely the least difficult to correct, is the incorrect assignment of snapshot data timestamps by the ingestion service. Also, character strings are not properly recognized by the Datastore query service and must be corrected.

The storage and retrieval of numeric arrays and data structures failed the data integrity testing. The solution to these issues will likely involve significant refactoring of the archiving mechanism. The gRPC transmission of these data types did appear to function correctly.

Image data and raw data operations were verified, but the accuracy of the data was not.

The complete output from the metadata integrity testing is included in APPENDIX C while output of the snapshot data integrity testing is included in APPENDIX D. The results of the snapshot data query performance testing are also included in APPENDIX D.

9. QUERY EVALUATION

The *performance* of the Datastore query service is evaluated here. Most operational issues were covered in Chapter 8 on data integrity testing. Thus, the primary focus of this chapter is on the snapshot data query service performance, however any additional operation issues are presented as they are observed.

Two separate snapshot data query performance evaluations were performed, a case-by-case scenarios test battery and a ramp-up test battery. Different test platforms were used for each set of evaluations. As to be shown, no significant performance differences were seen between the two evaluations.

The case-by-case scenarios performance evaluations covered first all utilize the *datastore-admin* query API library. Recall that within the *datastore-admin* project, the query API has separate interfaces for snapshot data query requests and metadata query requests. The snapshot data interface supports both synchronous and asynchronous communications while the metadata service utilizes the synchronous communications exclusively. Thus, both synchronous and asynchronous snapshot requests can all be evaluated using the same interface within the same test battery.

The additional set of ramp-up query evaluations were made using test data generated by the MPEX data simulator. These evaluations are described in Subsection 9.4. The query API library within the *datastore-client-lib* project is used for the MPEX data query testing. This is a wide API supporting requests for both metadata and snapshot data using the DQL language. All tests with this data set were performed for snapshot data using asynchronous query requests.

9.1 PLATFORMS

The case-by-case scenarios performance testing was executed on a development platform, the same as for scenarios ingestion testing. Specifically, a single node platform using an Intel i7 processor running MacOS 11.4. There were 6 cores running at 2.7 GHz with 16 GBytes of RAM. The Datastore core services were hosted on this machine, along with the non-partitioned InfluxDB database and single-node MongoDB database. All communications were made using the network loopback mode shown in Figure 21.

The test platform for the MPEX data query testing was the same as that used for the MPEX data simulator ingestion testing. Specifically, a single node platform utilizing an Intel i7 processor with 10 cores and 32 GBytes of RAM memory. The platform hosted the Datastore services, the non-partitioned InfluxDB installation, the single-node MongoDB installation, and the query test fixture. Loopback network communications were used for all communications as shown in Figure 21.

Any competition between query test fixtures and Datastore operations would likely be minimal for query testing. The Datastore archive is always pre-populated before the query operation is initiated. The test fixtures simply formulate the queries, collect snapshot data requests, then measure performance. Note that performance results for a platform utilizing dedicated servers, a multi-partitioned InfluxDB database, and high-speed networks would improve performance significantly.

9.2 METADATA

Metadata testing consisted entirely of data integrity tests already discussed in Chapter 8. No performance evaluations were considered for metadata queries. So far, metadata query requests are quite small compared to snapshot data requests. Consequently, overall performance is primarily constrained by snapshot data queries. (This condition may change in the future.)

9.3 SCENARIOS SNAPSHOT DATA PERFORMANCE

Snapshot data query results are returned in data tables of the form described in Subsection 4.5.3. Briefly, each result table contains a single time series column identifying timestamps for all other data columns in the table. The other table columns contain either heterogenous data from a data source (e.g., an EPICS process variable), or properties of the heterogeneous data (e.g., alarm condition, alarm status, etc.). Data columns may contain scalars (Booleans, integers, floats, strings), numeric arrays, data structures, raw data (i.e., byte arrays), and image data.

9.3.1 Errors

Operational issues with strings, arrays, and data structures were previously discussed in Subsection 8.2. The misassignment of snapshot data timestamping was discussed in Subsection 8.2.1.

9.3.2 Test Fixture

Performance was evaluated for both synchronous and asynchronous queries. However, data requests for synchronous queries were limited to 4 Mbytes due to gRPC message size limitations. The Datastore does not support data streaming for synchronous requests, thus, synchronous request sizes are limited by the maximum size of a single gRPC message (i.e., ~4Mbytes). Data streaming is supported for asynchronous query requests and, consequently, they are essentially unlimited in size.

There are two supported methods for making snapshot data requests: 1) the use of a Datastore Query Language (DQL) statement or 2) the use of a `DataRequest` query object. Either request can be made synchronously or asynchronously. The use of these two query methods was described in Subsection 4.8. Both methods were used for snapshot data query tests, no differences in results were seen. The results shown here employed the snapshot data request utility.

In each performance test the test fixture first clears the Datastore archive of all snapshot data and all metadata. The Datastore archive is then populated with data frames specified for the given scenario case. The test data frames are always ingested synchronously to avoid issues with concurrent asynchronous ingestion and query operations (thus, ingestion times can be lengthy). Then various query scenarios are performed on the test data. Asynchronous query tests are always performed. Synchronous query tests are performed whenever request size permits it.

9.3.3 Scenario Cases

Three different scenarios were performed, 1) requests for wide scalar table data, 2) requests for single data sources within wide tables, and 3) small query requests. In Cases 1 and 2 the Datastore archive is populated with wide data frames, that is, frames with large column counts. However, the data within each column is always of type double. The column counts are increased from 500 to 2,000.

In Case 1 multiple frames of the same type are also sent to the archive so that large queries results can be generated. In Case 2 only one frame is sent to the archive then queries for single data columns are requested and compared to queries for all archive data. Finally, in Case 3 the Datastore archive is populated with single, small data frames consisting of mixed scalar data, arrays, and structures, the same data frames used in the snapshot data integrity tests of Subsection 8.2.3.

Note that the performance testing here focuses on the evaluation of wide query requests, that is, for query results containing large numbers of columns. Narrow query requests, but with large data volumes, are considered in Subsection 9.4, the performance testing using the MPEX data.

Wide Scalar Tables

In this case three different types of data frames were used to populate the Datastore archive, all containing scalar data of type double. The data frames contained large column counts used to simulate wide query requests. Only one type of frame is used for each query test. Consider the following cases:

1) 500 columns of random double values; 100 rows
2) 1,000 columns of random double values; 100 rows
3) 2,000 columns of random double values; 100 rows

Thus, we have a total of 50,000 double values comprising the first request, 100,000 double values in the second, and 200,000 double values in the third.

For asynchronous queries, additional tests were performed using larger archived data sets composed of multiple frames. Specifically, 10 data frames for each frame type listed above were sent to the Datastore archive, then all data was subsequently queried using the open request. Thus, we have the following additional cases for asynchronous query tests:

1)+ 500 columns of random double values; 100 rows; **10 frames** (total size 1,000 rows)
2)+ 1,000 columns of random double values; 100 rows; **10 frames** (total size 1,000 rows)
3)+ 2,000 columns of random double values; 100 rows; **10 frames** (total size 1,000 rows)

The first request contains 500,000 double values, the second contains 1,000,000, and the third contains 2,000,000. (Compare these cases with the total request sizes in the MPEX data query tests in Subsection 9.4, the maximum being 6,400,000 doubles.)

The data rates are computed using the total real time required to complete the data request. In the performance tests below, the open query was used to recover all data in the Datastore archive. In all performance tests the random double values were correctly recovered from the Datastore (using the integrity test utility). However, the data timestamps where always incorrect, as was discussed in Subsection 8.2.1.

The results of the Datastore query service performance tests for wide queries are summarized in the three tables below. Each table contains results from the six test Datastore archive states described above. However, different results are seen for the same battery of query performance tests on the same platform, but *during different test runs*. Thus, to get an appreciation in the differences in performance the three tables are results from the same test fixture, but over multiple runs (each run takes approximately 45 minutes real time). The last summary table contains the

best overall performance results obtained in a single run. The raw output from the best performance run is included in APPENDIX D.

Case	Frame Count	Allocation (Mbytes)	Synch. Rate (Mbytes/sec)	Async. Rate (Mbytes/sec)	Async. Callback (Mbytes/sec)
1) 500 Columns	1	1.202	1.401	1.224	1.304
	10	12.02	-	1.560	1.511
2) 1k Columns	1	2.402	1.323	1.671	1.553
	10	24.02	-	1.375	1.556
3) 2k Columns	1	4.802	-	1.347	1.672
	10	48.02	-	1.415	1.549

Table 5: data rates for wide queries

The results shown in Table 5 are for a typical execution of the snapshot data performance test battery on a "hot" test platform. Each of the 6 test cases are represented by the table rows, subdivided by the frame count sub-case. That is, each test case contains the results for both a single data frame archive and the 10-frame archive available for asynchronous queries. The missing data rates for the synchronous query are due to the resultant set being too large for a single gRPC message; therefore, the test was not permitted.

The columns of Table 5 respectively list the test case, the sub-case (i.e., single frame or multi-frame), the total memory allocation of the query request, the data rate for a synchronous query, the data rate for an asynchronous query, and the data rate for an asynchronous callback request. The last column of Table 5, the asynchronous callback request, demonstrates a feature of the asynchronous query API where the query result is returned within a user-provided callback function. The table is returned fully loaded at the time of the callback function invocation. As seen in the results table, this operation performs more-or-less equivalent to that of a dynamically loaded data table, whose result is shown in the preceding column.

As seen from the table, the asynchronous data rates tend to average around 1.5 Mbytes/second, almost independent of data request size. There is some preference for the asynchronous callback method, but it is slight. In this test run the available synchronous query requests performed almost on par with the corresponding asynchronous requests, which is atypical.

Case	Frame Count	Allocation (Mbytes)	Synch. Rate (Mbytes/sec)	Async. Rate (Mbytes/sec)	Async. Callback (Mbytes/sec)
1) 500 Columns	1	1.202	0.772	1.635	1.634
	10	12.02	-	1.853	1.923
2) 1k Columns	1	2.402	1.629	1.812	1.844
	10	24.02	-	1.903	1.663
3) 2k Columns	1	4.802	-	1.764	1.900
	10	48.02	-	1.415	1.549

Table 6: data rates for wide queries – additional run

Table 6 above shows the results of another typical run of the same snapshot data query test battery for a hot platform. Notice the increase in performance for all but the largest query request in Case 3+, and the smallest synchronous request. As confirmed in the last result-summary table below, there appears to be a "sweet spot" for data rates concerning allocation request size. The decrease in performance for the smallest synchronous data requests is also seen in the table below, and in the mixed scalar data testing. This is likely due to a disproportionate rationing of gRPC resources during the initialization for synchronous query requests.

Table 7 below contains the best performance results seen in a single execution of the snapshot data query performance test battery. This was executed on a cold platform, that is, immediately after machine start up and login. Thus, there was a minimum of system processes running on the host platform at the time of the testing.

Case	Frame Count	Allocation (Mbytes)	Synch. Rate (Mbytes/sec)	Async. Rate (Mbytes/sec)	Async. Callback (Mbytes/sec)
1) 500 Columns	1	1.202	0.5927	2.018	1.945
	10	12.02	-	1.632	1.972
2) 1k Columns	1	2.402	1.590	2.152	2.192
	10	24.02	-	1.811	1.896
3) 2k Columns	1	4.802	-	2.057	1.874
	10	48.02	-	1.670	1.427

Table 7: data rates for wide queries – best performance

As seen above, the maximum data rates for query operations for wide scalar data frames are about 2 Mbytes/second. It is possible these values could be increased somewhat using a tuning parameter, the data page size. Recall from Subsection 4.8.2 that in asynchronous query operations the result sets are streamed to the data table in pages, that is, each gRPC message contains a page of the overall result. The Datastore query service allows users to specify the page size, in rows, used in the gRPC data message. Since the query tests utilized wide data requests, conservative values of page sizes were used, 50, 25, and 20, respectively for each of the three frame cases. Use of larger page sizes would require less gRPC messages to be streamed, potentially increasing

performance. However, one runs the risk of creating data pages with allocations greater than the maximum gRPC message limit and consequently corrupting the results table.

Single Data Source Queries

All the previous query tests were performed using the open query, specifically, querying for all contents of the Datastore. Recall that the test fixture first clears the Datastore archive of all data before populating it with the test data for the specific test case. Thus, the open query should return all the test data, and it should be the fastest query to do so (since no selection, range, or filtering operations are invoked).

Query tests here were conducted to request snapshot data from individual data sources (PVs). These tests were performed on a Datastore archive containing a single wide scalar data frame. Specifically, the Datastore archive was populated with a single wide data frame then a request for data from a single data source was performed. The result is compared to the open query request for the entire archive.

Consider the follow cases:

1) 500 columns of random double values; 100 rows
    a. 1 PV Query: 1 column result (total size 100 doubles)
    b. Open Query: 500 column result (total size 50,000 doubles)
2) 1,000 columns of random double values; 100 rows
    a. 1 PV Query: 1 column result (total size 100 doubles)
    b. Open Query: 1,000 column result (total size 100,000 doubles)
3) 2,000 columns of random double values; 100 rows
    a. 1 PV Query: 1 column result (total size 100 doubles)
    b. Open Query: 2,000 column result (total size 200,000 doubles)

Note that these are exactly the cases of the previous single-frame scenarios, with the addition of the single source request. Note also that all single data source requests are the same size, 100 double values. All tests were performed using the asynchronous query mechanism.

Case (Archive size)	**Sub Case** (Column Count)	**Allocation (Mbytes)**	**Query Time (seconds)**	**Async. Rate (Mbytes/sec)**
1) 500 Columns	a. 1 (single PV)	0.0048	0.549	0.00874
	b. 500 (all PVs)	1.202	0.735	1.635
2) 1k Columns	a. 1 (single PV)	0.0048	0.965	0.00497
	b. 1,000 (all PVs)	2.402	1.326	1.812
3) 2k Columns	a. 1 (single PV)	0.0048	2.768	0.00173
	b. 2,000 (all PVs)	4.802	2.723	1.764

Table 8: comparison of single PV query versus open query

Table 8 above shows the comparison of a single data source query request to that of an open query request for snapshot data with large column counts. The data rates for single source queries were significantly worse than the open query cases, indicating an implementation issue within the query service. The query times are of particular interest. From the above table we can see that query times for single PVs are on the same order as those for the entire data archive. Even though the size of the query results were orders of magnitude smaller for the single PV queries, the query times were comparable to that of the full archive request.

Note that the open query times scale with the size of the request, but the single PV query times do not. All single PV query results sizes are the same, 100 double values. Ideally, they should all have the same query time. The single PV query times appear to scale with the size of the Datastore archive, not with the size of the request. Although not tested directly, it is assumed that all query requests scale with the snapshot data archive size contained within the requested time range.

This condition indicates a serious issue within the Datastore query service. The query result size and query times do not scale correctly. However, this fact does suggest potential for greatly enhanced performance simply by improving query service implementation. The Datastore ingestion service does not appear to utilize the full proficiency of InfluxDB system.

Small Query Requests

It is interesting to compare the query times found for the wide data frames to those for the data frames used in integrity testing. Here, in each test the Datastore archive is cleared, populated with a single data frame of the described type, then an open query is performed.

Consider the following cases:

1) Mixed Scalar Data: frame of Booleans, integers, floats, and strings; 10 rows
2) Array Data: frame of 1 column array data; 10 rows
3) Structured Data: frame of 1 column data structures; 5 rows.
4) 500 columns of random double values; 100 rows

There first three cases are those used in the data integrity tests, the final case is the 500-column wide scalar data case considered above.

Case	Allocation (Kbytes)	Sync. Time (seconds)	Sync. Rate (Kbytes/sec)	Async. Time (seconds)	Async. Rate (Kbytes/sec)
1) Scalars	1.120	0.738	1.517	0.0290	38.60
2) Arrays	1.660	0.044	37.34	0.0255	65.06
3) Structures	0.240	0.0327	7.334	0.0243	9.891
4) 500 Doubles	1,202.4	1.559	771.0	0.7653	1,571.0

Table 9: query times and data rates for small queries

Table 9 shows the results of the small query results testing. Note the conversion to Kbytes units for these cases. The case results are ordered by rows. The columns now contain both the total query times as well as the corresponding data rates. The memory allocation for each request is also shown; note that the sizes of the small data requests are about 3 orders of magnitude smaller

than that for the wide scalar frame request. The maximum data rate is seen for the wide scalar asynchronous query, again suggesting a sweet spot for maximum data rates and request size.

From the above table we see that data rates for small data requests drop by 2 to 3 orders of magnitude from that maximum rate, generally being in the 10s of Kbyte/second range. This condition reflects the overhead in the query process. Specifically, the Datastore is spending a disproportionate amount of time with the query process rather than the query response. However, the overall wait times for the small queries tested are still fractions of a second, less than 30 milliseconds for the asynchronous case. Thus, although data rates are significantly smaller, the response times are good.

The higher data rates seen for the numeric arrays case are likely anomalous and due to the integrity issues seen with this data type in the previous chapter. Recall that entire arrays are being archived as single character strings within the InfluxDB database. The query service is incorrectly parsing these strings and, typically, returning `null` values rather than the double-valued array elements. Thus, although the query time is consistent with the other small frames, the actual data size recovered is smaller than the original data frame confusing the data rate calculation.

9.4 MPEX Snapshot Data Performance

Query service ramp-up testing was performed using the data simulator within the *mpex-sdp* project. Specifically, the Datastore archive was populated with data from the MPEX data simulator then various query tests were performed. The query service API within the *datastore-client-lib* project was used for all MPEX data query testing. The test battery differed significantly from the previous test setup. Whereas the previous test fixtures focused on wide query requests, the current tests involved narrow queries with large row counts. However, the results compare well.

9.4.1 Test Fixture

The MPEX data simulator emulates the data sources within the MPEX facility at Oak Ridge. The facility has 64 scalar process variables providing data as double values at sampling rate up to 1 kHz. It also has cameras supplying 240x640 byte images at sampling rates up to 20 frames per second. Thus, the query service test fixture first clears the Datastore archive then populates it with sample data from the 64 scalar sources and images from 3 cameras. The data simulator run for at least 10 seconds to populate the archive with the test data. Then queries for snapshot data with increasing time ranges were performed to ramp up the results size.

The query test using the MPEX data were performed using a series of increasing request sizes, as shown in Figure 29. Also shown in the diagram is that the test fixture isolated query time operation and transmission time over the network. No image data was queried, only the scalar data within

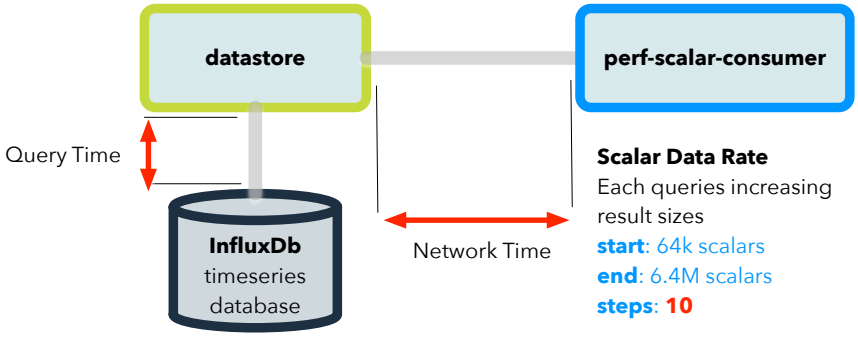


Figure 29: MPEX data query tests configuration

the MPEX data set. Query sizes varied starting from a 64,000 double request to a 6,400,000 double request. Specifically, the Datastore archive was populated with snapshot data from the 64 scalar process variables. Then 10 request cases were performed, ramping from an initial request for 1,000 rows of snapshot data to a final request of 100,000 rows of snapshot data. All requests were loaded into a single data table.

9.4.2 Ramping Cases

The results of the ramping query tests are shown in Figure 30. As with the previous evaluations, we see again that the transmission times and the query times scale well with the size of the query request. In the graph of Figure 30 the scaling is even more evident.

To summarize the graph in Figure 30, the query service can provide up to 113,000 double values per second. This results in a maximum data rate of 2.7 Mbytes/second, below the goal of 100 Mbytes/second, but relatively consistent with the previous query testing on the slower platform. Recall that for wide query requests but of similar size, we found maximum data rates around 2.0 Mbytes/second. The increased performance for narrow queries could be attributed to the increased hardware capabilities of the test platform. However, it is also possible that wide queries are simply more expensive.

The best transmission rate for query operations is 7.1million double values per second. This translates to a maximum transmission rate of 170 Mbytes/second for Java double objects, again well above our performance goal of 100 Mbytes/second.

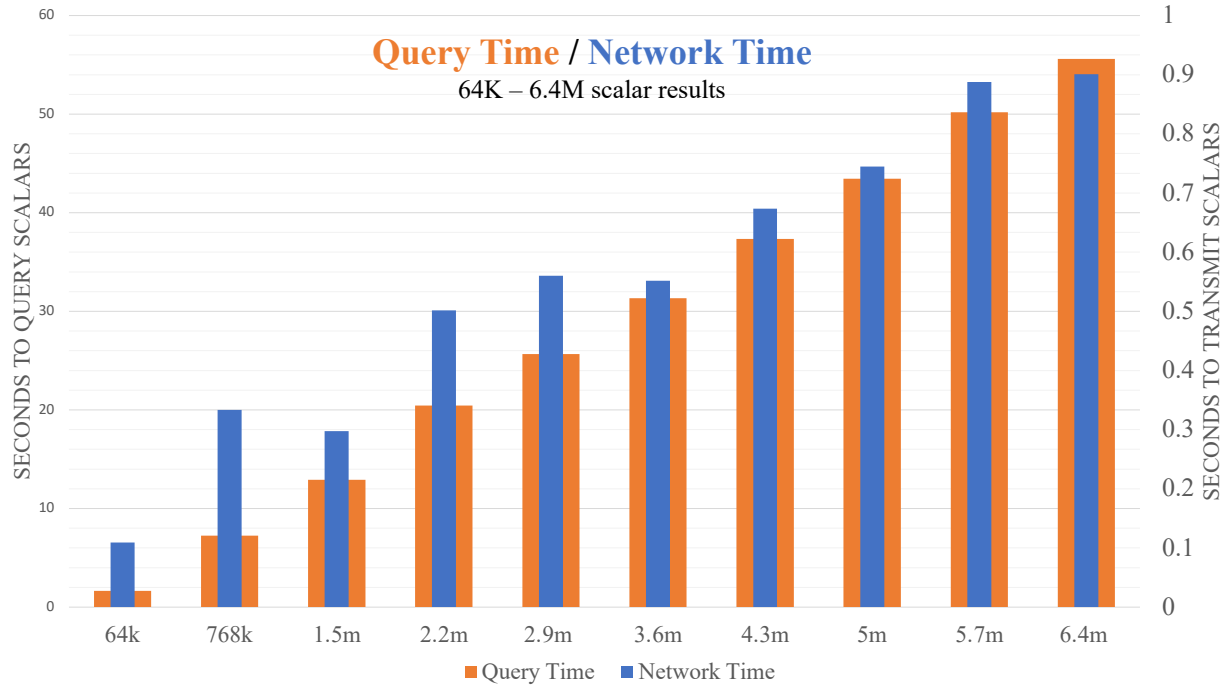


Figure 30: MPEX data query results

Single data source queries were not performed within these request cases. Thus, the performance drop for individual PV snapshot data requests was not identified here.

9.5 CONCLUSIONS

The maximum data rates for the query service are about 2 Mbytes/second for wide data requests and 2.7 Mbytes/second for narrow requests of similar size. However, in both cases these rates still produce a considerable real-time wait for large requests. For example, a scalar data request producing a results table 500 columns wide and 1,000 rows long has a total allocation of approximately 12 Mbytes in Java (each Double requires 24 bytes). Thus, the request would take 6 seconds at maximum data rates. An equivalent request for 2,000 columns of double values would require 24 seconds to complete. A significant performance improvement is necessary for data science applications requiring large data requests.

When querying for single data sources within larger data sets a significant performance drop is observed. The query times appear to scale with the size of the overall data set within the requested time range, rather than the request size itself. It is expected that this condition is the result of under-utilization of the underlying InfluxDB database querying capabilities.

PART 4: Summary & Recommendations

10. PROJECT SUMMARY

10.1 MACHINE LEARNING DATA PLATFORM

Osprey DCS has developed a working prototype for the Machine Learning Data Platform (MLDP). The MLDP is composed of two separate systems each which function independently. The first is the *Aggregator* system, the front-end of the MLDP, which acquires data from the beamline hardware and diagnostics then stages it for archiving by the second component. The second component, the *Datastore*, manages the data archive and provides wide query capabilities for data science and machine learning applications. The Aggregator system is EPICS based and is necessarily deployed within the EPICS control system of an accelerator facility. The Datastore is fully standalone and hosted on a separate platform.

The operation of each MLDP component was independently verified. The Aggregator service was tested on a platform emulating the LCLS facility at SLAC. The Datastore was tested on two different platforms, one simulating the Aggregator system and one simulating the MPEX facility at ORNL. The Aggregator system performed as expected meeting all requirements of the LCLS simulation. The Datastore functioned correctly, however there were notable errors in some operations. Additionally, Datastore performance is below requirements. Exhaustive testing of the Datastore systems was performed to identify the prototype status and determine future development strategies.

In addition to the MLDP prototype, a Web Application was also developed in the Phase I effort. This application was not part of the original project description, it was originally built as a development tool for independent inspection of the Datastore archive. However, its standalone utility was recognized and pursued. The Web Application provides universal access to the data archive, data scientists can use the tool for independent archive inspection without need of any programming or scripting. Thus, additional development efforts were included.

In summary, the Machine Learning Data Platform prototype *is operational*. Data throughput is confirmed, from the front-end hardware system, through the Aggregator system, to the Datastore archive, and available to back-end users from the query service. There is still one outstanding feature yet to be implemented, post-ingestion annotation of the data archive. Additionally, there are archiving errors within the current Datastore prototype that need to be corrected. The data rates for the Datastore ingestion service need improvement to function in conjunction with the Aggregator. The data rates for the Datastore query service need improvement for practical data science applications; wait times for large data requests are too large.

Since the Datastore is the least mature component of the MLDP, the current report focused heavily on the Datastore prototype status. Accordingly, the summary reflects this emphasis.

10.2 DATA AGGREGATOR

The Aggregator has both a distributed aspect and a centralized component. The front-end of the Aggregator consists of distributed local aggregator components that acquire signal data from beamline hardware and processes it for transport to the central component. The central aggregator collects the distributed data and assembles it for staging to the Datastore.

The Aggregator system must be deployed directly within the EPICS environment. Unlike the Datastore system, its implementation explicitly requires EPICS utilities and communications protocols. However, in its current form, it requires no other third-party resources. It is implemented fully in the C++ programming language and its builds using the UNIX make utility.

The Aggregator component of the MLDP is at a mature state of development. It performed as required within a testbed simulating the Linac Coherent Light Source (LCLS) facility at Stanford Linear Accelerator. The test bed simulated 3,200 signals at a sampling frequency of 1 kHz. Thus, evaluation on the LCLS test platform verified operation slightly below the stated performance goal of 4,000 scalars at 1 kHz acquisition. All signals were successfully coalesced into composite EPICS NTTable objects with 1,000 timestamped rows. These tables were offered at 1 Hz rate for a scalar data transmission of 3,200,000 double values every second, or an overall transmission rate of approximately 25 Mbytes/second (the equivalent rate for a Java implementation is 77 Mbytes/second).

We do not expect significant development effort for the Aggregator system going forward. However, the Aggregator system should be tested on a platform capable of producing larger data frames (i.e., data frames containing more than 3,200 signals). If performance issues manifest were are prepared to implement alternative designs presented in the next chapter. Alternatively, it may be possible to employ standard data compression techniques to improve performance if needed.

10.3 DATASTORE

The Datastore component manages the MLDP data archive and functions as an independent system within the overall MLDP. The Datastore itself has separate subsystems for data ingestion and archiving, and for the query service offered to data science and machine learning applications. The Datastore is fully independent and is deployed on a separate host platform. It does not require any EPICS tools or services. However, it does utilize third-party database systems and software frameworks.

10.3.1 Standalone Versatility

The Datastore is a standalone system and capable of data ingestion from data providers other than the Aggregator service. Thus, any experimental or industrial facility able to supply data in the format required by the ingestion service (the Datastore front-end) can populate the MLDP data archive. In addition to the ingestion of data frames provided by the Aggregator, an alternate ingestion interface is available, one which accepts data in the Java native format. Consequently, data ingestion is not restrictive. Once the data archive is populated, all data science and machine learning features of the MLDP are available to a wide variety of facilities through the back end query service. This condition demonstrates the utility of maintaining the Datastore as an independent system within the Machine Learning Data Platform.

The MLDP data archive is available to any data consumer through the Datastore query service. That is, whatever the method of data ingestion, the query service presents a consistent interface for all data science applications. Currently two API libraries are available, both in the Java language. One is a narrow, high-level interface providing data tables appropriate for data science applications. The other is a low-level interface utilizing a proprietary query language DQL specific to the Datastore and producing results in their native gRPC message format. An additional narrow

query API is intended to be built for the Python language based upon the Pandas DataFrame object popular in data science applications.

10.3.2 Heterogeneous Data

The Datastore is capable of ingesting, archiving, and retrieving heterogeneous data. Specifically, the Datastore can ingest heterogeneous data within the same data stream and retrieve heterogeneous data within a single query request. The current prototype is designed to recognize scalar data of type Boolean, integer, float, and character string, and complex data in the form of raw data, one-dimensional numeric arrays, arbitrary data structures, and images. However, during evaluations it was found that the treatment of character strings, numeric arrays, and data structures had implementation errors. These errors are discussed more below.

10.3.3 Concurrent Ingestion

The versatility of the Datastore system is further enhanced by its ability to ingest concurrent streams of heterogeneous data. That is, a single Datastore installation can accept data from multiple data providers simultaneously. This feature was verified using the MPEX data simulator. Further, the Datastore ingestion performance appears unaffected when utilizing concurrent streams. No performance loss was seen when ingesting equivalent amounts of data either through a single stream or through multiple concurrent streams. However, the overall ingestion performance of the Datastore needs improvement (discussed below).

10.3.4 Data Transmission

The Datastore has a unique communications mechanism based upon the gRPC framework. Data transmission through this communications system was measured during evaluations and found to present no limitation in performance. Data transmission rates as high as 380 Mbytes/second were seen in ingestion testing and 170 Mbytes/second during query testing.

10.3.5 API Libraries

To accommodate wide application of the Datastore system, two separate ingestion API libraries are available. One contains a narrow API interface specifically for ingesting data frames, the format required by the Aggregator. The other contains a wide, general-purpose interface accepting data in native Java formats. Ingestion operations were confirmed using two separate data simulators, one simulating the Aggregator service and one simulating the MPEX facility at Oak Ridge National Laboratory. This ability to ingest data from widely differing data sources illustrates the utility of maintaining the Datastore as an independent system.

The Datastore query service also has two available query API libraries. One library is designed for data science applications, providing utilities for simplifying data requests, and offering requested results in the form of heterogeneous data tables. The alternate query API library supports Datastore developers and users with advanced knowledge of Datastore operation. It is a wide interface offering responses in the form of gRPC messages native to Datastore communications. All data requests are made in the Datastore Query Language (DQL), an SQL-like query language particular to the Datastore. No significant performance differences were seen between any of the API libraries.

A Datastore administration library is also available. The library provides tools for Datastore management, data integrity testing, and performance testing as well as unit test suites for individual

Datastore components. The administration library also contains the data science ingestion and query APIs described above.

10.3.6 Missing Features

The MLDP is missing one outstanding feature stated in the project goals. The current prototype does not support post-ingestion annotation of the data archive, or "back annotation". This feature would allow data scientists to modify the data archive with notes, data associations, calculations, and other artifacts after ingestion. For example, data science computations obtained from the current data archive could produce results which are then subsequently stored within the archive itself. In this manner, post-ingestion data relationships and computed data properties found within the archive would be available to other data science applications.

One aspect of the back annotation feature was realized, the ability to save and recall *named queries*. Specifically, if a user finds a query result particularly useful the query itself can be stored within the archive for later retrieval. Another feature is the ability to assign user attribute lists within metadata.

Although the post-ingestion feature is not currently implemented, the framework for doing is available. The data archive is composed of both a repository of snapshot data and a repository of associated metadata. The metadata is contained in a NoSQL database system which is well-suited for managing post-ingestion annotations, including calculations and data associations. Being an important aspect of the MLDP data science application, suggestions for full implementation are described in Subsection 11.7.

10.3.7 Operational Errors

The Datastore prototype has notable operational issues that were observed during evaluation. These ranged from simple implementation "bugs" to more serious design weaknesses. Issues concerning data archiving are covered in the next subsection.

The Datastore appears to continue ingestion processing after asynchronous ingestion has terminated (i.e., after acknowledging a close stream operation), or it continues to maintain gRPC resources post termination. That is, either the Datastore continues to process ingested data after acknowledging data receipt and stream closure, or it simply fails to release resources in a timely fashion. In the former case one would expect collisions with query operations if data were still being processed to the archive. The condition has resulted in many errors during testing, including crashing of the ingestion service, corruption of the data archive, and corruption of the entire InfluxDB installation. It is important that this issue be resolved. The solution may be as simple as restricting stream closure acknowledgements to occur only after all ingested data has been processed.

The Datastore misassigns timestamps to incoming data. It was seen that snapshot data was archived with incorrect timestamps, always lagging that of the original timestamp but not in any consistent fashion. However, when testing metadata, it was found that snapshots themselves were correctly timestamped (i.e., the "trigger time" was correct). Thus, the timestamping error within the Datastore core can be isolated to the snapshot data ingestion mechanism. The snapshot data timestamps seen within the Datastore archive are, however, correctly recovered by the query service, no timestamping errors are observed there.

The current mechanism by which the Datastore ingests timestamps has limitations. This issue is technical, but the consequence can be described concisely as follows. Only incoming data that is

sampled at a constant rate can be continuously streamed to the Datastore ingestion service. Incoming data with inconsistent sampling, say containing interruptions, cannot be continuously streamed. Data can always be ingested statically; that is, it is correctly ingested when sent synchronously as pre-acquired frame. However, this method is slow.

Character strings are not correctly recognized. Specifically, character strings are being ingested and archived correctly but the query service does not correctly retrieve them. This is likely a straightforward implementation "bug." More serious is the inability to recognize numeric arrays and data structures. These are treated as data integrity issued covered in the next subsection.

There is an error in the snapshot UID assignment mechanism. Sometimes snapshot UIDs are repeated, which destroys the integrity of the metadata archive. Sometime multiple snapshot UIDs are assigned to a single data frame. Moreover, the Datastore appears to use the snapshot trigger time when creating snapshot UIDs; this is an unnecessary frailty requiring unique snapshot trigger times.

10.3.8 Archive Data Integrity

During evaluations it was found that the archiving mechanism for some data types was suspect, specifically for numeric arrays and data structures. The transport of these data types to the archive was confirmed, however, the archiving and retrieval was brittle.

The Datastore is currently archiving numeric array data as a single character string. Upon retrieval the character string is parsed then returned as a numeric array. Although the archiving is accurate (i.e., the array string representation is accurate), string parsing fails during query operations. This implementation was likely chosen due to performance considerations; however, more robust implementations are available. The implementation is also subject to a loss of numeric precision depending upon how the array string representation is managed. Loss of numeric precision is a serious limitation to many data science applications.

The treatment of data structures by the archiving mechanism is convoluted. Data structured are partially flattened and archived in both numeric format and as character strings. Communications within the Datastore core correctly transported data structures, however, the query mechanism was unable to correctly reassemble data structures of arbitrary form. A more robust archiving mechanism would help mitigate the inability of the query service to restore complex data structures. Additionally, there is an inherent loss of numeric precision when reassembling data structures from character strings.

10.3.9 Ingestion Service Performance

Two separate test fixtures were used in performance evaluations, one simulating the Aggregator system and one simulating the MPEX facility at Oak Ridge National laboratory. However, both platforms consisted of a single node hosting both the data simulator and the Datastore. No dedicated servers, networks, or other hardware systems were used (i.e., they were development platforms). Communications between the data simulators, Datastore, and internal database systems all utilized an internal network loop back. Significant performance enhancements can be expected with the use of dedicated hardware. Additionally, the InfluxDB database system used in both platforms was single partitioned (multiple partitioning would increase the number of available write heads).

The project goal of archiving 4,000 signals at a sampling rate of 1 kHz translates to a data rate of 32 Mbytes/second for native implementations and 100 Mbytes/second for a Java implementation.

(Recall that the Datastore is implemented in Java.) During performance evaluations, maximum data rates observed for continuous, sustained ingestion ranged from 0.2 to 7 Mbytes/second.

Most important is that data rates varied greatly depending upon the format of the ingested data. Data frames containing thousands of data sources performed worst. Unfortunately, this is the stated case for the Aggregator system. Data rates also varied when ingesting data of different types. Data types archived to system files rather than the InfluxDB database system appear to produce better performance numbers.

Regarding data formats, the best performance occurs when sending large data sets but with few data columns (up to 15). The maximum data rates seen here range between 2 and 5 Mbytes/second. When ingesting data frames containing equivalent amounts of data, but with large column counts, the performance drops significantly. A maximum data rate of 1.2 Mbytes/second was observed for sustained ingestion of data with 64 columns, while a maximum data rate of 0.25 Mbytes/second was observed for sustained ingestion of data with 2,000 columns.

Concerning data types, the best sustained ingestion rates were seen for image data, in the 5 to 7 Mbyte/second range. Images are archived as system files on the host platform. Higher rates were observed for array data but, due to the current integrity issue with arrays, they remain unconfirmed.

10.3.10 Query Service Performance

It is desirable to have the Datastore query service perform at the same data rates as the ingestion service, 100 Mbytes/second. Query performance is of lesser concern than that of the ingestion service, which must perform at the level of the Aggregator system. However, it still presents an issue with respect to the applicability of real-time machine learning application. The objective is to make archived data available as fast as possible so that data science applications can control facility operations. Thus, large data sets must be made available as quickly as possible.

The maximum data rates for the query service are about 2 Mbytes/second for wide data requests (2,000 columns) and 2.7 Mbytes/second for narrow requests of similar size (64 columns). These rates are significantly below performance objectives, yielding significant real-time waits for large requests. For example, a 12 Mbytes scalar data request producing a results table 500 columns wide and 1,000 rows long would take 6 seconds at maximum data rates. An equivalent request for 2,000 columns of double values would require 24 seconds to complete (~50 Mbyte result). A significant performance improvement is necessary for data science applications requiring large data requests.

When querying for a single data column within a larger request set, a significant performance drop is observed. Specifically, when considering requests for data within a specified time interval, the query time for the single data column is almost identical to the query time for the entire set. The query times appear to scale with the size of the overall data set within the time range, rather than the request size itself.

It was observed, however, that query times for requests containing all data columns do scale appropriately, query times scale with the overall request size. This fact suggests under-utilization of the InfluxDB database querying capabilities. Thus, there is potential for enhanced performance simply by improving the implementation within the query service.

10.4 WEB APPLICATION

A prototype Web Application was developed as a companion utility for the Datastore archive. It is a standalone system deployed on a separate host server platform. Once deployed it requires only a standard internet web browser for operation.

The Web Application is an independent tool providing universal access to the MLDP data archive from remote locations. It does not require programming interfaces or any programming knowledge, only a standard web browser. Thus, data scientists can inspect the data archive for properties and patterns before any code is written.

The Web Application is comprised of two components, an application server which connects to the Datastore query service, and a client which runs within the web browser. To initiate the Web Application the user directs the web browser to the server URL which then loads the client system providing access to the Datastore archive.

The web application offers users several "perspectives" on the Datastore archive. Perspectives allow the user to browse the snapshot data archive based upon metadata properties. One perspective is available for each metadata type. Inspection by timestamp is also available. The Web Application also enables users to browse metadata using "inspector" pages, offering a more detailed view of archived metadata. Operation of the Web Application was verified for each supported perspective and inspector page.

The Web Application is functional but not stylized. Enhancements of the user interface would provide easier access to the Datastore archive, that is, improve the "look and feel" of the application. Additional features would also greatly enhance the usefulness of the applications, such as the ability to export selected data in differing formats (e.g., CSV, Excel, Pandas, etc.). Offering standard data analysis tools would also be beneficial, such as visualization, fitting, and statistical properties.

11. FUTURE EFFORTS

From the design considerations, the evaluations, and their summaries, we can identify specific efforts concerning future development. Here we isolate outstanding implementation and performance issues with the current prototype then offer recommendations for future development. Recommendations are catalogued by Machine Learning Data Platform component.

11.1 MACHINE LEARNING DATA PLATFORM

The Aggregator and the Datastore systems of the Machine Learning Data Platform (MLDP) were tested on separate platforms. Eventually a test platform capable of simulating full data throughput of the MLDP should be implemented before deployment. However, such testing is only practical when the Datastore ingestion service is performing on par with the Aggregator service.

11.2 THE AGGREGATOR

Although performance of the Aggregator system so far meets expectation, alternative designs to improve performance are provided. On such approach is shown in Figure 31. Contrast the figure with the current aggregation process demonstrated in Figure 4.

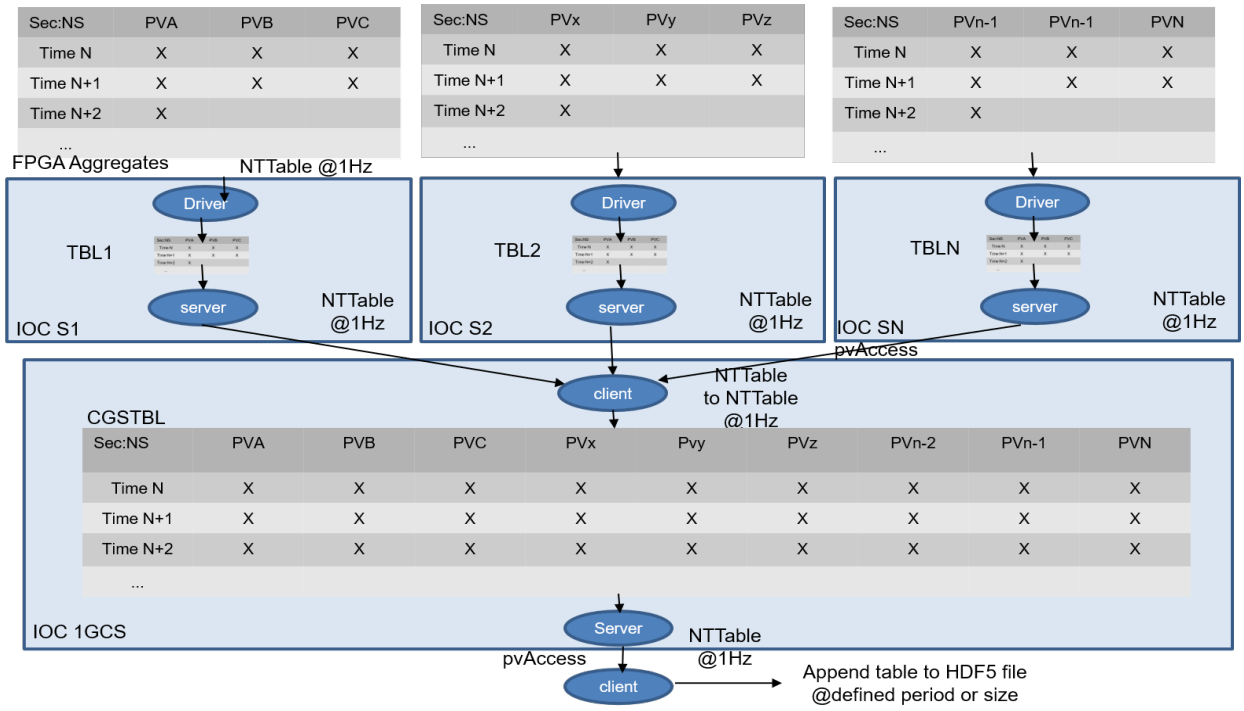


Figure 31: data aggregation using FPGAs

Rather than performing the initial signal aggregation in the IOC as seen in Figure 4, in Figure 31 signal aggregation is now performed within the controlling hardware of the beamline device, specifically the Field Programmable Gate Arrays (FPGAs). That is, the first stage of the aggregation process is pushed farther up the data transport stack, into the FPGA controller. Since these devices are programmable they may be configured to create the data table of hardware signals, rather than the EPICS IOC controller. The FPGA generated data tables are then fed to the EPICS device driver which loads the tables into the IOC controller where they are staged for transport. The hosting IOC then sends the data tables to the central Aggregator system for time-correlation and coalescing. Once assembled by the central aggregator, the final composite table is sent to the Datastore for ingestion.

Alternatively, a simpler option is also available. Rather than full data tables, the FPGA controller may create data vectors for each signal. These data vectors would then be provided to the device driver within the IOC. The distributed aggregator components within the local IOC would assemble the signal vectors into correlated, time-series data tables for collection by the central aggregation service. Although likely not as fast, this strategy is attractive since it requires less implementation within the FPGA where development is more intensive.

Another possiblity is to apply standard data compression techniques to the signal data tables before transmitting them to the central aggregator. It would first need to be determined whether or not compression offers any benefit; that is, determine if the EPICS pvAccess communications potocol is a performance bottleneck. Tabular data is well-suited to data compression techniques, however, application may be more difficult regarding EPICS NTTables as they are normative types, essentally complex data structures.

11.3 DATASTORE API LIBRARIES

Complementary to the two query API libraries currently available, an additional API should be developed for the Python programming language. Specifically, a query API that produces results in the form of Pandas `DataFrame` objects. The Pandas library is popular in modern data science applications, it should be supported.

The ingestion and query APIs within the *datastore-admin* project should be broken into separate libraries now that their operations are well-defined. This would simultaneously facilitate API accessibility and establish Datastore administration as a standalone service. Additionally, the Datastore administration operations should be built out and formalized with well-defined API interfaces.

The operation and use of the API libraries should be documented. The operation and use of the administration library and services should be documented once they are formalized. The APIs are the aspects of the MLDP seen by the largest user group and should be well documented.

11.4 DATASTORE ARCHIVING

The Datastore should adopt a more robust implementation for the archiving of numeric arrays and data structures. The current implementation is frail and easily subject to errors, as was observed in the evaluations.

Although fast, the current method of archiving numeric arrays as character strings is dubious. The current archiving implementation is fast since InfluxDB efficiently stores character strings. Storing the individual elements of the array would likely create additional storage time and overhead, although no more so than with the storage of equivalent double values. However, even if the query service did parse character strings correctly (it does not) there is an inherent loss of numeric precision in the process. The tradeoff between performance and data precision is not recommended for data science applications.

The Datastore also requires a more robust implementation for archiving data structures. Currently data structures are being flattened and archived partially as floats for field values, and as character strings for substructures. The query service is unable to recognize the substructures within a data structure, parsing them incorrectly. An implementation that does not involve string conversion of numeric values would be preferrable. Considering the organization of the InfluxDB time-series database, an alternate archive location for structured data may be warranted, as was done for image data. Perhaps the use of HDF5 file formats would be a viable option in this case.

A more sweeping approach toward the data archiving problem, which may also address the ingestion performance problem, would be a redesign of the time-series data storage system and archive. The most radical approach would be to replace the InfluxDB database with an archive of data files on the host platform. Snapshot data could be stored in HDF5 format that would easily accommodate numeric arrays of arbitrary dimension. Complex data structures could also be stored in HDF5 format, in a similar fashion as was done for the Aggregator system and `NTTable` data (see Figure 5). This approach would require significant design and development effort but would also eliminate some third-party dependencies. Additionally, as was seen in the ingestion evaluations, the archiving of snapshot data to files was also faster (e.g., for images and raw data) although a multiple write-head InfluxDB installation was not tested.

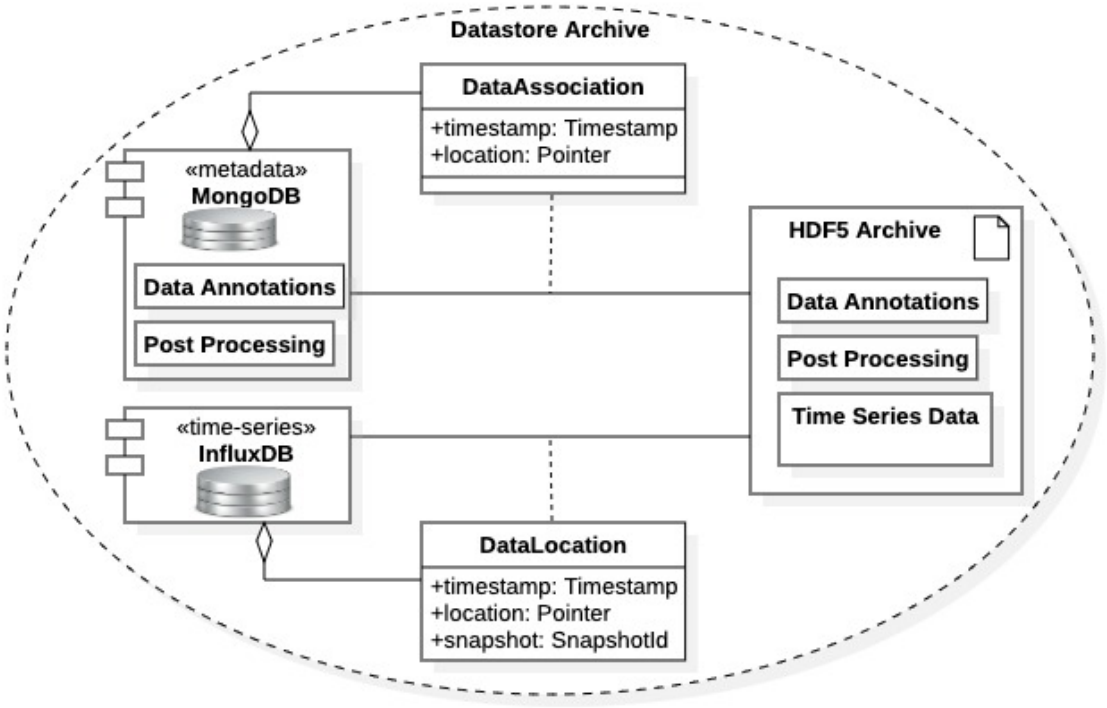


Figure 32: alternate archive design

Rather than completely abandon the InfluxDB model, a first compromise might be something like that shown in Figure 32. There we see a hybrid system where the snapshot data is stored in an HDF5 file archive on the host file system, but the InfluxDB database is still used for partial storage. The data sources and timestamps of the snapshot data would be stored in InfluxDB, along with the location pointers of the data within the HDF5 archive. This configuration would allow for quick lookups of snapshot data locations by the query service. As shown in the diagram, the metadata could be stored in the MongoDB database as before or it could be stored within the HDF5 archive. Data associations between time-series data can be created and stored within the metadata archive as per the original design. The InfluxDB database is efficient at storing scalar time-series data but storing more complex data types requires more sophisticated archiving techniques. The above design attempts to combine the efficiency of InfluxDB time-series storage with the broad applicability of HDF5.

To accommodate the redesigned archive, standard HDF5 file formats would need to be designed to accommodate all forms of snapshot data. An organizational structure would also need to be determined to allow proper assignment and back-referencing of metadata (as shown in Figure 32). However, HDF5 can maintain the structure and relationships of all data types currently supported by the Datastore. Furthermore, as a self-describing file format, HDF5 allows post-ingestion augmentation of the archive, that is, it readily supports the MLDP feature of annotating existing archives. As shown in the diagram, and user annotations could be stored in the HDF5 archive or the MongoDB database.

11.5 DATASTORE INGESTION OPERATIONS

The Datastore ingestion operations are of particular importance since it must be capable of accepting data at the rate offered by the Aggregator system. Full archiving capabilities of data by the Machine Learning Data Platform cannot be tested until the ingestion service performs on par with the Aggregator service. Query performance can be addressed later in development.

The best verified data rate observed for sustained, continuous ingestion on the single-node test platform is between 2 and 5 Mbytes/second, more than 20-fold below the objective of 100 Mbytes/seconds. However, the worst observed rates occur for the ingestion wide data frames containing thousands of columns, those produced by the Aggregator. A data rate of approximately 0.25 Mbytes/second is observed on single-node platforms requiring a factor 400 speedup. The best performance, stated above, occurs for large data sets but with few data columns.

Thus, either the data processing implementation within the Datastore ingestion service is under performing, or the InfluxDB database system is under performing, or there is a combination of both conditions. We explore each situation with possible remedies. Consider first the InfluxDB system.

The InfluxDB database system is specifically designed to efficiently archive time-series data in real time. It is a promising technology for the archiving operations and, in principle, should work well. Thus, it is likely that the system is being used inefficiently. A case in point is immediately apparent: the InfluxDB database system supports simultaneous, multiple write head archiving and multiple read head retrieval. In fact, a single InfluxDB installation can be partitioned to support multiple write head operation with multi-threading. This advanced feature of multi-partitioning was not exploited in the performance evaluations. An immediate performance improvement would be expected simply by partitioning the InfluxDB archive.

Another immediate and simple method to increase performance of the Datastore ingestion service is with a hardware solution. Specifically, we add more server nodes to the host platform as demonstrated in Figure 33. Combined with the multi-partitioning of the InfluxDB database, a multi-server host platform is extremely promising. This form of parallelism allows for an immediate increase in write performance, essentially proportional to the number of servers. Thus, adding *N* InfluxDB servers (each containing a separate partition) would increase performance by a factor *N*. Additional subpartitioning within each server would likely increase performance even more, depending upon the hardware and the format of the data frame being ingested.

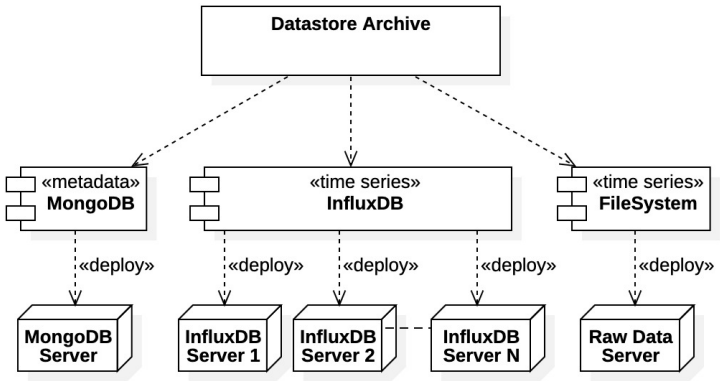


Figure 33: Datastore multi-server deployment

By combining improved ingestion service implementation along with a hardware solution the required performance objective may be realized. Note that a by employing a multi-node server

platform, the ingestion implementation performance requirements are significantly lessened. For example, to mitigate an underperformance factor 20, a factor 10 is immediately available by using 10 server nodes, reducing the required ingestion performance improvement to a factor 2. This fact is especially encouraging and suggests that *it is the most promising approach*. In summary, the InfluxDB database system is being underutilized and immediate performance enhancements should be seen simply with better application and with better hardware.

A specific issue of ingestion performance concern which may be immediately addressed is that of PV record comparison. To ensure data integrity all incoming data sources are compared to all existing data sources within the metadata archive. As observed during evaluations, this process appears to require significant processing time and resources. The situation is especially weighty for wide data frames containing thousands of data columns. The process of data source comparison could see significant performance enhancement with alternative implementations, some form of improvement should be investigated. One option is to create a memory cache of recent PV records to improve real-time accessibility. Another option, although not as preferrable, would be a reasonable tradeoff between integrity requirements and performance. It is expected that more efficient PV record comparison could easily provide a doubling in ingestion performance.

To consider the worst-case scenario, although unlikely, it could be that the sheer volume of data being ingested simply overwhelms the InfluxDB system. If this is the case, then it may be necessary to implement the snapshot data archive in house. Rather than relying solely on data files to implement the Datastore archive, the hybrid design of Figure 32 was suggested. It was discussed in the previous subsection that direct archiving to the file system is fast and, thus, a protentional remedy for ingestion underperformance. However, this is a significant redesign requiring extensive development effort. Multi-partitioning of the InfluxDB installation should be attempted first.

11.6 DATASTORE QUERY OPERATIONS

From the results of the query performance studies, it was found that the size of the query results and the length of the query time do not scale correctly. The single data source query times scale with the size of the time range, not with the size of the actual request. This suggests that query times, and thus data rates, can be greatly enhanced with better query service implementation. An immediate reimplementation to reflect proper scaling should be prioritized in future development.

Upon scrutiny of the InfluxDB operation it was found that the Influx language provides more fine-grain control over data requests than is being utilized by the Datastore query service. The query service is performing InfluxDB requests that are larger than necessary, then subsequently post-processing results. Thus, fine-grained query operations can be pushed down to the InfluxDB database where, in principle, they should be more efficient. Better utilization of the InfluxDB query capabilities would increase query performance and solve the scaling issue.

Adding multiple InfluxDB servers to the Datastore host platform (as shown in Figure 33) would also increase query performance. Again, we could expect increased performance scaling with the number of read head and partitioned servers. Thus, the hardware and multi-partitioning solution is available for both ingestion and query performance improvements.

The query service performance is of lower priority than the ingestion service performance, as ingestion performance must match that of the Aggregator system. The query performance can be addressed later in development and may of the performance issues corrected with the ingestion

service can likely be then applied to the query service. However, it is desirable to have the query service perform on par with the ingestion service with the objective that data science applications be fast enough for online facility interaction.

11.7 DATA ARCHIVE ANNOTATION

Clearly the addition of the data archive annotation feature must be included in future efforts. Data scientists and applications must be able to annotate archived data with post-ingestion information which is subsequently available to other users. Eventually, the ability to add post-ingestion data calculations to the archive should be available. However, the implementation could be performed in stages, adding additional functionality in each step.

Consider the following implementation strategy:

1. First implement the ability to add user notations to archive data (notes, comments, etc.).
2. Notations would then be broadened to include source data locations within the archive.
3. Add the ability to identify associations between data sets within the archive.
4. The inclusion of calculations obtained from source data can be added as notations to the above data associations.

Thus, the feature of archive annotation with user attributes, data associations, and post-ingestion calculations would be fully implemented at the final stage. Each stage within the process would provide additional functionality toward that goal, and each stage could be tested and verified independently before proceding to the next.

11.8 DATASTORE ERRORS

The inability to simultaneously ingest timestamps and snapshot data must be resolved. It is desirable to maintain a continuous data stream from the data provider to the ingestion service where snapshot data and timestamps are transmitted concurrently. This condition is especially important for situations where data is sampled at irregular rates (e.g., interruptions). The issue concerns the use of timestamp lists, which must be implemented more robustly. The ability to maintain snapshot data and timestamps within a single gRPC message is consistent by design. However, this process also fails to operate correctly. If one attempts to send multiple data frames over the same gRPC connection using attached timestamp lists the operation fails. However, a single data frame over one connection operates correctly in the synchronous case.

During ingestion even when a “ready” acknowledgement is received by the Datastore, a timeout is sometimes encountered when attempting to close the stream if large data sets are involved. Additionally, sometimes a close stream event is acknowledged by the ingestion service and the service appears to continue processing data, or simply fails to release gRPC resources in a timely fashion. The query service then becomes deadlocked when attempting to access the archive and subsequently fails. This suggests that the ingestion service maintains a lock on archive data and is unwilling or unable to respond to the close stream event, or the close stream event is not well defined. This close-stream exception might be avoided through refactoring of the ingestion API; however, it should be addressed in the ingestion service implementation.

The Datastore ingestion service has a sporadic error concerning snapshot UIDs. Sometimes multiple snapshot UIDs are generated for a single data frame and sometimes snapshot UIDs are

not unique (i.e., separate data frames are assigned the same snapshot UID). The condition is most conspicuous with synchronous communications where a single snapshot UID should be produced for each data frame ingested (specifically, a data bin). The error must be corrected.

Additionally, the current Datastore implementation appears to require a unique snapshot timestamp (i.e., "trigger time") to create snapshot UIDs. This mechanism is not robust and should be refactored.

Although not an error per se, in principle for synchronous communications it should be possible to identify multiple data frames as belonging to a single snapshot, perhaps by amending time reference. This would be advantageous when data binning is invoked where all bins could be associated with the same data frame (i.e., snapshot UID). Addition of such a feature would be worth consideration in future efforts.

11.9 WEB APPLICATION

Although the Web Application was not initially intended for external use, continued development should be persued. It has standalone utility as an independent tool for data scientists providing universal access to the MLDP data archive from any remote location. Additionally, the Web Application allows data scientists to evaluate the data archive without any programming or programming knowledge, only a standard web browser is required. To establish the utility of the Web Application, additional features are recommended, as well as building out the existing ones. Also, stylistically the interface should be improved with better "look and feel" supporting simpler interactions with users.

The Web Application should implement the exporting of archived snapshot data and metadata. The ability to isolate and export archive data in different formats (e.g., such as CSV, Excel, NumPy, etc.) would greatly enhance the applicability of the Web Application. Data sets of interest could be identified remotely then downloaded for local analysis and processing.

The addition of common statistical features would also greatly enhance the usefulness of the Web Application. The usefulness of data visualization in the form of graphs and charts is immediately apparent. The ability to analyze time-series data using averages, data fitting, and correlations are extremely useful in data analysis. Additionally, the ability to identify gaps and anomalies in time-series data would also be useful. However, extensive development in this area may be unwarranted as specialized data analysis would be available through data exporting and common spreadsheet applications, or using more advanced data analysis and machine learning applications through the API query interface.

Once the post-ingestion archive annotations capability of the Datastore is implemented, it would be desirable to have a subset of these features available within the web application. Features of particular interest would be the ability to annotate archived data with user notes and comments, and to provide data associations within the archive. However, these additions cannot be developed until the Datastore query service is mature enough and offers a well-defined interface for such activities.

12. REFERENCES

1. Keras. [Online] https://keras.io/.
2. TensorFlow. [Online] https://www.tensorflow.org/.
3. PyTorch. [Online] https://pytorch.org/.
4. scikit-learn. [Online] https://scikit-learn.org/stable/index.html.
5. Machine Learning at SLAC. [Online] SLAC National Accelerator Laboratory. https://ml.slac.stanford.edu/.
6. Artificial Intelligence and Machine Learning. [Online] Thomas Jefferson National Accelerator Facility. https://www.jlab.org/AI.
7. Buongiorno, Caitlyn. Machine Learning Applications. [Online] Fermi National Accelerator Laboratory, January 8, 2021. https://news.fnal.gov/2021/01/fermilab-receives-doe-funding-to-develop-machine-learning-for-particle-accelerators/.
8. Dietrich, Tamara. DOE Funding Boosts Artificial Intelligence Research at Jefferson Lab. [Online] Thomas Jefferson National Accelerator Facility, September 22, 2020. https://www.jlab.org/news/releases/doe-funding-boosts-artificial-intelligence-research-jefferson-lab.
9. Machine Learning, Artificial Intelligence, and Particle Accelerators. [Online] Accelerator Technology and Applied Physics Division, Berkeley Lab. https://atap.lbl.gov/machine-learning-artificial-intelligence-and-particle-accelerators/.
10. Gnida, Manuel. AI learns physics to optimize particle accelerator performance. [Online] Stanford Linear Accelerator National Laboratory, July 29, 2021. https://www6.slac.stanford.edu/news/2021-07-29-ai-learns-physics-optimize-particle-accelerator-performance.aspx.
11. Machine Learning System Improves Accelerator Diagnostics. [Online] Thomas Jefferson National Laboratory/US Dept of Energy, June 3, 2021. https://www.energy.gov/science/np/articles/machine-learning-system-improves-accelerator-diagnostics.
12. Laboratory, Oak Ridge National. Material Plasma Exposure eXperiment (MPEX). [Online] 2022. https://www.ornl.gov/mpex.
13. Spring. [Online] VMware Tanzu, 2022. https://spring.io/.
14. Spring Boot. [Online] VMware Tanzu, 2022. https://spring.io/projects/spring-boot.
15. InfluxDB. [Online] InfluxData Inc., 2022. https://www.influxdata.com/.
16. MongoDB. [Online] MongoDB Inc., 2022. https://www.mongodb.com/home.
17. gRPC. [Online] gRPC Authors, 2022. https://grpc.io/.
18. Protocol Buffers. [Online] Google, 2022. https://developers.google.com/protocol-buffers/.
19. React . [Online] Meta Platforms, Inc., 2022. https://reactjs.org/.

20. EPICS - Experimental Physics and Industrial Control System. [Online] 2022. https://epics-controls.org/.

21. White, Greg, et al. EPICS V4 Normative Types. [Online] 2015. https://docs.epics-controls.org/en/latest/specs/Normative-Types-Specification.html.

22. pvAccess and pvData. [Online] EPICS. https://epics-controls.org/resources-and-support/documents/pvaccess/.

23. Apache Maven Project. *Apache Maven Project.* [Online] Apache Software Foundation, 2022. https://maven.apache.org/.

24. normativeTypesCPP Reference. [Online] EPICS Documentation, 2019. https://docs.epics-controls.org/projects/normativetypes-cpp/en/latest/ntCPP.html.

25. pandas.DataFrame. [Online] numFocus, Inc., 2022. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html.

26. A high performance, open source universal RPC framework. [Online] 2022. https://grpc.io/.

ACKNOWLEDGEMENTS

Work here was performed under the auspices of the United States Department of Energy (DOE). Development of the Machine Learning Data Platform prototype was funded with a DOE Small Business Innovation Research Grant (Grant DE-SC0022583) made available by the Office of High Energy Physics. The MPEX data simulator was built while under contract with Oak Ridge National Laboratory, managed by the US Department of Energy.

The authors wish to thank Greg White of Stanford Linear Accelerator facility for suggestions and support in developing the Aggregation system and the testing platform. We also thank Daniele Filippetto and Carlos Serrano of Lawrence Berkeley National Laboratory for their support in the initial research and development of the Machine Learning Data Platform, especially the many useful suggestions identifying practical data science and machine learning requirements.

APPENDIX A: SYNCHRONOUS INGESTION SCENARIOS

```
TEST: testScenario1TwoSmallFrames FROM
com.ospreydcs.datastore.admin.IngestionServiceScenariosTest
2022-11-11T16:33:01.992953
Scenario TWO_SMALL_FRAMES Results
  Scenario description      : 1.3 KByte frames (2) - 1 column Boolean, Integer, Double, String,
10 Rows
  Scenario fully completed  : true
  Scenario time limit (sec) : 2
  Scenario time active (sec): 1.227673
  Frame size avg. (bytes)   : 3170
  Frame count offered       : 2
  Frame count sent          : 2
  Frame rate offered  (f/s) : 100.0
  Frame rate achieved (f/s) : 1.6290983022352044
  Data size offered (bytes) : 6340
  Data size sent    (bytes) : 6340
  Data rate achieved  (b/s) : 5164.241618085598
  Execution exception       : null
  Provider Registration ID  : ProviderId(id=1, name=TEST-Provider1, status=OpStatus(message=,
status=NO_STATUS, severity=NO_ALARM))
  Snapshot IDs :
    SnapshotId(id=1, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=2, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))


TEST: testScenario2TwoModestFrames FROM
com.ospreydcs.datastore.admin.IngestionServiceScenariosTest
2022-11-11T16:33:16.203443
Scenario TWO_MODEST_FRAMES Results
  Scenario description      : 4.34 MByte frames (2) - 10 Columns Boolean, Integer, Double,
String, ByteArray, 100 Rows
  Scenario fully completed  : true
  Scenario time limit (sec) : 15
  Scenario time active (sec): 13.957901
  Frame size avg. (bytes)   : 4350900
  Frame count offered       : 2
  Frame count sent          : 2
  Frame rate offered  (f/s) : 5.0
  Frame rate achieved (f/s) : 0.14328802016864858
  Data size offered (bytes) : 8701800
  Data size sent    (bytes) : 8701800
  Data rate achieved  (b/s) : 623431.846951773
  Execution exception       : null
  Provider Registration ID  : ProviderId(id=2, name=TEST-Provider2, status=OpStatus(message=,
status=NO_STATUS, severity=NO_ALARM))
  Snapshot IDs :
    SnapshotId(id=3, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=3, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=5, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=5, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))


TEST: testScenario3TwoHugeFrames FROM com.ospreydcs.datastore.admin.IngestionServiceScenariosTest
2022-11-11T16:38:46.343234
Scenario TWO_HUGE_FRAMES Results
  Scenario description      : 177 MByte frames (2) - 40 scalar columns, 10 byte arrays, 10
images, 250 rows
  Scenario fully completed  : true
  Scenario time limit (sec) : 400
  Scenario time active (sec): 319.797124
  Frame size avg. (bytes)   : 175155500
  Frame count offered       : 2
  Frame count sent          : 2
  Frame rate offered  (f/s) : 0.1
  Frame rate achieved (f/s) : 0.006253964935594605
  Data size offered (bytes) : 350311000
  Data size sent    (bytes) : 350311000
  Data rate achieved  (b/s) : 1095416.3552765409
  Execution exception       : null
  Provider Registration ID  : ProviderId(id=3, name=TEST-Provider3, status=OpStatus(message=,
status=NO_STATUS, severity=NO_ALARM))
  Snapshot IDs :
    SnapshotId(id=7, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=7, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=8, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=8, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=9, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
```

```
SnapshotId(id=9, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=10, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=10, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=11, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=11, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=12, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=12, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=13, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=13, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=14, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=14, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=15, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=15, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=16, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=16, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=17, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=17, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=18, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=18, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=19, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=19, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=20, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=20, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=21, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=21, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=22, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=22, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=23, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=23, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=24, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=24, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=25, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=25, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=26, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=26, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=27, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=27, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=28, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=28, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=29, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=29, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=30, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=30, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=31, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=31, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=33, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=33, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=34, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=34, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=35, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=35, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=36, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=36, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=37, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=37, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=38, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=38, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=39, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=39, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=40, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=40, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=41, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=41, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=42, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=42, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=43, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=43, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=44, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=44, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=45, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=45, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=46, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=46, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=47, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=47, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=48, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=48, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=49, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=49, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=50, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=50, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
```

```
    SnapshotId(id=51, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=51, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=52, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=52, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=53, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=53, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=54, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=54, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=55, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=55, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=56, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=56, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=57, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=57, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))


TEST: testScenario4LargeArrays FROM com.ospreydcs.datastore.admin.IngestionServiceScenariosTest
2022-11-11T16:39:02.943127
Scenario LARGE_ARRAYS Results
  Scenario description      : 3.8-Mbyte frames (10) - 10 columns of 64 KByte arrays, 10 rows
  Scenario fully completed  : true
  Scenario time limit (sec) : 30
  Scenario time active (sec): 16.017196
  Frame size avg. (bytes)   : 3835740
  Frame count offered       : 10
  Frame count sent          : 10
  Frame rate offered  (f/s) : 5.0
  Frame rate achieved (f/s) : 0.6243290024046656
  Data size offered (bytes) : 38357400
  Data size sent    (bytes) : 38357400
  Data rate achieved  (b/s) : 2394763.727683672
  Execution exception       : null
  Provider Registration ID  : ProviderId(id=2, name=TEST-Provider2, status=OpStatus(message=,
status=NO_STATUS, severity=NO_ALARM))
  Snapshot IDs :
    SnapshotId(id=59, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=60, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=61, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=62, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=63, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=64, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=65, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=66, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=67, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=68, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))


TEST: testScenario5LargeStructures FROM
com.ospreydcs.datastore.admin.IngestionServiceScenariosTest
2022-11-11T16:39:48.067880
Scenario LARGE_STRUCTURES Results
  Scenario description      : 8.2-Mbyte frames (10) - 10 columns of 81.8 KByte structures, 10
rows
  Scenario fully completed  : true
  Scenario time limit (sec) : 60
  Scenario time active (sec): 43.77788
  Frame size avg. (bytes)   : 8893536
  Frame count offered       : 10
  Frame count sent          : 10
  Frame rate offered  (f/s) : 5.0
  Frame rate achieved (f/s) : 0.2284258625588996
  Data size offered (bytes) : 88935360
  Data size sent    (bytes) : 88935360
  Data rate achieved  (b/s) : 2031513.6319986256
  Execution exception       : null
  Provider Registration ID  : ProviderId(id=3, name=TEST-Provider3, status=OpStatus(message=,
status=NO_STATUS, severity=NO_ALARM))
  Snapshot IDs :
    SnapshotId(id=69, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=69, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=70, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=71, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=71, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=72, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=73, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=73, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=74, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=75, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=75, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=76, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=77, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
```

```
    SnapshotId(id=77, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=78, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=79, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=79, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=80, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=81, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=81, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=82, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=83, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=83, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=84, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=85, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=85, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=86, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=87, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=87, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=88, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))


TEST: testScenario6LargeImages FROM com.ospreydcs.datastore.admin.IngestionServiceScenariosTest
2022-11-11T16:40:24.416727
Scenario LARGE_IMAGES Results
  Scenario description      : 26.2 Mbyte frames (6) - 10 Columns of 260 KByte images, 10 Rows
  Scenario fully completed  : true
  Scenario time limit (sec) : 90
  Scenario time active (sec): 34.235131
  Frame size avg. (bytes)   : 26232203
  Frame count offered       : 6
  Frame count sent          : 6
  Frame rate offered  (f/s) : 0.5
  Frame rate achieved (f/s) : 0.17525856699657436
  Data size offered (bytes) : 157393220
  Data size sent    (bytes) : 157393220
  Data rate achieved  (b/s) : 4597418.365362761
  Execution exception       : null
  Provider Registration ID  : ProviderId(id=1, name=TEST-Provider1, status=OpStatus(message=,
status=NO_STATUS, severity=NO_ALARM))
  Snapshot IDs :
    SnapshotId(id=89, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=89, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=90, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=90, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=91, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=91, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=92, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=92, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=93, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=93, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=95, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=95, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=96, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=96, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=97, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=97, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=98, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=98, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=99, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=99, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=101, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=101, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=102, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=102, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=103, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=103, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=104, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=104, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=105, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=105, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=107, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=107, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=108, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=108, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=109, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=109, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=110, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=110, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=111, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=111, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=113, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=113, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=114, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
```

```
    SnapshotId(id=114, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=115, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=115, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=116, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=116, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=117, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=117, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=119, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=119, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=120, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=120, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=121, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=121, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=122, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=122, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=123, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=123, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))


TEST: testScenario7SmallTables FROM com.ospreydcs.datastore.admin.IngestionServiceScenariosTest
2022-11-11T16:41:44.365690
Scenario SMALL_TABLES Results
  Scenario description      : 363 KByte frames (10) - 100 Scalar Columns (10 Bool, 10 Int, 10
Str, 70 Dbl); 100 Rows
  Scenario fully completed  : true
  Scenario time limit (sec) : 90
  Scenario time active (sec): 79.819054
  Frame size avg. (bytes)   : 369960
  Frame count offered       : 10
  Frame count sent          : 10
  Frame rate offered  (f/s) : 10.0
  Frame rate achieved (f/s) : 0.1252833690562156
  Data size offered (bytes) : 3699600
  Data size sent    (bytes) : 3699600
  Data rate achieved  (b/s) : 46349.835216037514
  Execution exception       : null
  Provider Registration ID  : ProviderId(id=1, name=TEST-Provider1, status=OpStatus(message=,
status=NO_STATUS, severity=NO_ALARM))
  Snapshot IDs :
    SnapshotId(id=125, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=126, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=127, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=128, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=129, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=130, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=131, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=132, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=133, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=134, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))


TEST: testScenario8ModestTables FROM com.ospreydcs.datastore.admin.IngestionServiceScenariosTest
2022-11-11T16:51:04.266747
Scenario MODEST_TABLES Results
  Scenario description      : 6 MByte frames (10) - 500 Scalar Columns (All Doubles); 500 Rows
  Scenario fully completed  : true
  Scenario time limit (sec) : 600
  Scenario time active (sec): 559.565027
  Frame size avg. (bytes)   : 6012000
  Frame count offered       : 10
  Frame count sent          : 10
  Frame rate offered  (f/s) : 5.0
  Frame rate achieved (f/s) : 0.01787102395160947
  Data size offered (bytes) : 60120000
  Data size sent    (bytes) : 60120000
  Data rate achieved  (b/s) : 107440.59599707613
  Execution exception       : null
  Provider Registration ID  : ProviderId(id=1, name=TEST-Provider1, status=OpStatus(message=,
status=NO_STATUS, severity=NO_ALARM))
  Snapshot IDs :
    SnapshotId(id=135, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=135, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=137, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=137, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=139, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=139, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=141, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=141, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=143, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=143, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=145, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
```

```
    SnapshotId(id=145, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=147, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=147, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=149, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=149, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=151, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=151, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=153, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=153, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))


TEST: testScenario9LargeTables FROM com.ospreydcs.datastore.admin.IngestionServiceScenariosTest
2022-11-11T17:04:24.441374
Scenario LARGE_TABLES Results
  Scenario description      : 24 MByte frames (10) - 1000 Scalar Columns (All Doubles); 1000 Rows
  Scenario fully completed  : false
  Scenario time limit (sec) : 800
  Scenario time active (sec): 789.971399
  Frame size avg. (bytes)   : 24024000
  Frame count offered       : 5
  Frame count sent          : 2
  Frame rate offered  (f/s) : 0.1
  Frame rate achieved (f/s) : 0.0025317372281221033
  Data size offered (bytes) : 120120000
  Data size sent    (bytes) : 48048000
  Data rate achieved  (b/s) : 60822.4551684054
  Execution exception       : null
  Provider Registration ID  : ProviderId(id=1, name=TEST-Provider1, status=OpStatus(message=,
status=NO_STATUS, severity=NO_ALARM))
  Snapshot IDs :
    SnapshotId(id=155, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=155, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=156, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=156, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=157, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=157, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=158, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=159, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=159, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=160, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=160, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=161, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=161, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=162, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))


TEST: testScenario9HugeTables FROM com.ospreydcs.datastore.admin.IngestionServiceScenariosTest
2022-11-12T12:53:17.482978
Scenario HUGE_TABLES Results
  Scenario description      : 48 MByte frames (5) - 2000 Scalar Columns (All Doubles); 1000 Rows
  Scenario fully completed  : true
  Scenario time limit (sec) : 1500
  Scenario time active (sec): 1295.948966
  Frame size avg. (bytes)   : 48024000
  Frame count offered       : 1
  Frame count sent          : 1
  Frame rate offered  (f/s) : 0.1
  Frame rate achieved (f/s) : 7.716353237940699E-4
  Data size offered (bytes) : 48024000
  Data size sent    (bytes) : 48024000
  Data rate achieved  (b/s) : 37057.01478988641
  Execution exception       : null
  Provider Registration ID  : ProviderId(id=1, name=TEST-Provider1, status=OpStatus(message=,
status=NO_STATUS, severity=NO_ALARM))
  Snapshot IDs :
    SnapshotId(id=20, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=20, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=21, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=21, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=22, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=22, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=23, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=23, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=24, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=24, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=25, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=25, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=26, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))


TEST: testScenarioStringTables FROM com.ospreydcs.datastore.admin.IngestionServiceScenariosTest
```

```
2022-11-11T19:10:48.918582
Scenario STRING_TABLES Results
  Scenario description      : 41 MByte frames avg (10) - 500 Columns of 60 character string
(avg); 500 Rows
  Scenario fully completed  : false
  Scenario time limit (sec) : 400
  Scenario time active (sec): 399.023862
  Frame size avg. (bytes)   : 40634200
  Frame count offered       : 10
  Frame count sent          : 1
  Frame rate offered  (f/s) : 1.0
  Frame rate achieved (f/s) : 0.002506115787130545
  Data size offered (bytes) : 406342000
  Data size sent    (bytes) : 40920000
  Data rate achieved  (b/s) : 102550.2580093819
  Execution exception       : null
  Provider Registration ID  : ProviderId(id=1, name=TEST-Provider1, status=OpStatus(message=,
status=NO_STATUS, severity=NO_ALARM))
  Snapshot IDs :
    SnapshotId(id=12, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=12, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=13, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=13, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=14, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=14, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=15, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=15, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=16, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=16, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=17, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))


TEST: testScenarioArrayTables FROM com.ospreydcs.datastore.admin.IngestionServiceScenariosTest
2022-11-12T13:07:04.847214
Scenario ARRAY_TABLES Results
  Scenario description      : 32 MByte frames avg (10) - 1 Column of 32 Kbyte double arrays;
1,000 Rows
  Scenario fully completed  : true
  Scenario time limit (sec) : 600
  Scenario time active (sec): 81.590742
  Frame size avg. (bytes)   : 38379000
  Frame count offered       : 10
  Frame count sent          : 10
  Frame rate offered  (f/s) : 2.0
  Frame rate achieved (f/s) : 0.12256292509265328
  Data size offered (bytes) : 383790000
  Data size sent    (bytes) : 383790000
  Data rate achieved  (b/s) : 4703842.50213094
  Execution exception       : null
  Provider Registration ID  : ProviderId(id=1, name=TEST-Provider1, status=OpStatus(message=,
status=NO_STATUS, severity=NO_ALARM))
  Snapshot IDs :
    SnapshotId(id=27, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=27, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=28, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=28, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=29, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=29, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=30, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=30, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=31, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=31, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=33, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=33, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=34, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=34, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=35, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=35, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=36, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=36, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=37, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=37, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=39, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=39, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=40, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=40, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=41, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=41, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=42, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=42, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
    SnapshotId(id=43, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
```

```
SnapshotId(id=43, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=45, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=45, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=46, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=46, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=47, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=47, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=48, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=48, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=49, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=49, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=51, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=51, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=52, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=52, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=53, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=53, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=54, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=54, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=55, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=55, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=57, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=57, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=58, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=58, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=59, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=59, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=60, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=60, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=61, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=61, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=63, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=63, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=64, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=64, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=65, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=65, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=66, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=66, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=67, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=67, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=69, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=69, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=70, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=70, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=71, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=71, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=72, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=72, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=73, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=73, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=75, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=75, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=76, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=76, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=77, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=77, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=78, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=78, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=79, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=79, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=81, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=81, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=82, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=82, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=83, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=83, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=84, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=84, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=85, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
SnapshotId(id=85, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
```

APPENDIX B: ASYNCHRONOUS INGESTION SCENARIOS

```
TEST: testScenario1TwoSmallFrames FROM com.ospreydcs.datastore.admin.IngestionStreamScenariosTest
2022-11-11T17:15:32.392978
Scenario TWO_SMALL_FRAMES Results
  Scenario description      : 1.3 KByte frames (2) - 1 column Boolean, Integer, Double, String,
10 Rows
  Scenario fully completed  : true
  Scenario time limit (sec) : 2
  Scenario time active (sec): 0.062413
  Frame size avg. (bytes)   : 2610
  Frame count offered       : 2
  Frame count sent          : 2
  Frame rate offered  (f/s) : 100.0
  Frame rate achieved (f/s) : 32.04460609167962
  Data size offered (bytes) : 5220
  Data size sent    (bytes) : 5220
  Data rate achieved  (b/s) : 83636.4218992838
  Execution exception       : null
  Provider Registration ID  : ProviderId(id=1, name=null, status=OpStatus(message=,
status=NO_STATUS, severity=NO_ALARM))
  Snapshot IDs :
    SnapshotId(id=1, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))


TEST: testScenario2TwoModestFrames FROM
com.ospreydcs.datastore.admin.IngestionStreamScenariosTest
2022-11-11T17:15:37.885844
Scenario TWO_MODEST_FRAMES Results
  Scenario description      : 4.34 MByte frames (2) - 10 Columns Boolean, Integer, Double,
String, ByteArray, 100 Rows
  Scenario fully completed  : true
  Scenario time limit (sec) : 15
  Scenario time active (sec): 0.23662300000000003
  Frame size avg. (bytes)   : 4375400
  Frame count offered       : 2
  Frame count sent          : 2
  Frame rate offered  (f/s) : 5.0
  Frame rate achieved (f/s) : 8.452263727532825
  Data size offered (bytes) : 8750800
  Data size sent    (bytes) : 8750800
  Data rate achieved  (b/s) : 3.6982034713447124E7
  Execution exception       : java.util.concurrent.TimeoutException:
DsIngestionStream#awaitStreamEndResponse() - Timeout out waiting on response from Datastore
  Provider Registration ID  : ProviderId(id=2, name=null, status=OpStatus(message=,
status=NO_STATUS, severity=NO_ALARM))
  Snapshot IDs :


TEST: testScenario3TwoHugeFrames FROM com.ospreydcs.datastore.admin.IngestionStreamScenariosTest
2022-11-11T17:18:42.622681
Scenario TWO_HUGE_FRAMES Results
  Scenario description      : 177 MByte frames (2) - 40 scalar columns, 10 byte arrays, 10
images, 250 rows
  Scenario fully completed  : true
  Scenario time limit (sec) : 400
  Scenario time active (sec): 169.355028
  Frame size avg. (bytes)   : 175098000
  Frame count offered       : 2
  Frame count sent          : 2
  Frame rate offered  (f/s) : 0.1
  Frame rate achieved (f/s) : 0.011809510609865093
  Data size offered (bytes) : 350196000
  Data size sent    (bytes) : 350196000
  Data rate achieved  (b/s) : 2067821.6887661582
  Execution exception       : null
  Provider Registration ID  : ProviderId(id=3, name=null, status=OpStatus(message=,
status=NO_STATUS, severity=NO_ALARM))
  Snapshot IDs :
    SnapshotId(id=3, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))


TEST: testScenario4LargeArrays FROM com.ospreydcs.datastore.admin.IngestionStreamScenariosTest
2022-11-11T17:18:51.352933
Scenario LARGE_ARRAYS Results
  Scenario description      : 3.8-Mbyte frames (10) - 10 columns of 64 KByte arrays, 10 rows
  Scenario fully completed  : true
  Scenario time limit (sec) : 30
  Scenario time active (sec): 6.726521
```

```
  Frame size avg. (bytes)   : 3835740
  Frame count offered       : 10
  Frame count sent          : 10
  Frame rate offered  (f/s) : 5.0
  Frame rate achieved (f/s) : 1.4866526098706894
  Data size offered (bytes) : 38357400
  Data size sent    (bytes) : 38357400
  Data rate achieved  (b/s) : 5702412.881785398
  Execution exception       : null
  Provider Registration ID  : ProviderId(id=2, name=null, status=OpStatus(message=,
status=NO_STATUS, severity=NO_ALARM))
  Snapshot IDs :
    SnapshotId(id=4, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))


TEST: testScenario5LargeStructures FROM
com.ospreydcs.datastore.admin.IngestionStreamScenariosTest
2022-11-11T17:18:59.288994
Scenario LARGE_STRUCTURES Results
  Scenario description      : 8.2-Mbyte frames (10) - 10 columns of 81.8 KByte structures, 10
rows
  Scenario fully completed  : true
  Scenario time limit (sec) : 60
  Scenario time active (sec): 1.844144
  Frame size avg. (bytes)   : 8893536
  Frame count offered       : 10
  Frame count sent          : 10
  Frame rate offered  (f/s) : 5.0
  Frame rate achieved (f/s) : 5.42257003791461
  Data size offered (bytes) : 88935360
  Data size sent    (bytes) : 88935360
  Data rate achieved  (b/s) : 4.822582184471495E7
  Execution exception       : java.util.concurrent.TimeoutException:
DsIngestionStream#awaitStreamEndResponse() - Timeout out waiting on response from Datastore
  Provider Registration ID  : ProviderId(id=3, name=null, status=OpStatus(message=,
status=NO_STATUS, severity=NO_ALARM))
  Snapshot IDs :


TEST: testScenario6LargeImages FROM com.ospreydcs.datastore.admin.IngestionStreamScenariosTest
2022-11-11T17:19:32.897461
Scenario LARGE_IMAGES Results
  Scenario description      : 26.2 Mbyte frames (6) - 10 Columns of 260 KByte images, 10 Rows
  Scenario fully completed  : true
  Scenario time limit (sec) : 90
  Scenario time active (sec): 29.381435
  Frame size avg. (bytes)   : 26232203
  Frame count offered       : 6
  Frame count sent          : 6
  Frame rate offered  (f/s) : 0.5
  Frame rate achieved (f/s) : 0.20421058399632286
  Data size offered (bytes) : 157393220
  Data size sent    (bytes) : 157393220
  Data rate achieved  (b/s) : 5356893.562210287
  Execution exception       : null
  Provider Registration ID  : ProviderId(id=1, name=null, status=OpStatus(message=,
status=NO_STATUS, severity=NO_ALARM))
  Snapshot IDs :
    SnapshotId(id=6, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))


TEST: testScenario7SmallTables FROM com.ospreydcs.datastore.admin.IngestionStreamScenariosTest
2022-11-11T17:19:58.389686
Scenario SMALL_TABLES Results
  Scenario description      : 363 KByte frames (10) - 100 Scalar Columns (10 Bool, 10 Int, 10
Str, 70 Dbl); 100 Rows
  Scenario fully completed  : true
  Scenario time limit (sec) : 90
  Scenario time active (sec): 20.357379
  Frame size avg. (bytes)   : 363060
  Frame count offered       : 10
  Frame count sent          : 10
  Frame rate offered  (f/s) : 10.0
  Frame rate achieved (f/s) : 0.4912223719959234
  Data size offered (bytes) : 3630600
  Data size sent    (bytes) : 3630600
  Data rate achieved  (b/s) : 178343.19437683997
  Execution exception       : java.util.concurrent.TimeoutException:
DsIngestionStream#awaitStreamEndResponse() - Timeout out waiting on response from Datastore
  Provider Registration ID  : ProviderId(id=1, name=null, status=OpStatus(message=,
status=NO_STATUS, severity=NO_ALARM))
```

```
  Snapshot IDs :


TEST: testScenario8ModestTables FROM com.ospreydcs.datastore.admin.IngestionStreamScenariosTest
2022-11-11T17:24:55.059718
Scenario MODEST_TABLES Results
  Scenario description      : 6 MByte frames (10) - 500 Scalar Columns (All Doubles); 500 Rows
  Scenario fully completed  : true
  Scenario time limit (sec) : 600
  Scenario time active (sec): 291.345432
  Frame size avg. (bytes)   : 6012000
  Frame count offered       : 10
  Frame count sent          : 10
  Frame rate offered  (f/s) : 5.0
  Frame rate achieved (f/s) : 0.0343235173839966
  Data size offered (bytes) : 60120000
  Data size sent    (bytes) : 60120000
  Data rate achieved  (b/s) : 206352.98651258755
  Execution exception       : java.util.concurrent.TimeoutException:
DsIngestionStream#awaitStreamEndResponse() - Timeout out waiting on response from Datastore
  Provider Registration ID  : ProviderId(id=1, name=null, status=OpStatus(message=,
status=NO_STATUS, severity=NO_ALARM))
  Snapshot IDs :


TEST: testScenario9LargeTables FROM com.ospreydcs.datastore.admin.IngestionStreamScenariosTest
2022-11-11T17:39:05.321318
Scenario LARGE_TABLES Results
  Scenario description      : 24 MByte frames (10) - 1000 Scalar Columns (All Doubles); 1000 Rows
  Scenario fully completed  : false
  Scenario time limit (sec) : 800
  Scenario time active (sec): 840.070769
  Frame size avg. (bytes)   : 24024000
  Frame count offered       : 5
  Frame count sent          : 5
  Frame rate offered  (f/s) : 0.1
  Frame rate achieved (f/s) : 0.0059518795136175
  Data size offered (bytes) : 120120000
  Data size sent    (bytes) : 120120000
  Data rate achieved  (b/s) : 142987.95343514683
  Execution exception       : java.util.concurrent.TimeoutException:
DsIngestionStream#awaitQueueEmpty() - Timeout out waiting for data queue to empty
  Provider Registration ID  : ProviderId(id=1, name=null, status=OpStatus(message=,
status=NO_STATUS, severity=NO_ALARM))
  Snapshot IDs :


TEST: testScenario9HugeTables FROM com.ospreydcs.datastore.admin.IngestionStreamScenariosTest
2022-11-11T18:29:05.524670
Scenario HUGE_TABLES Results
  Scenario description      : 48 MByte frames (5) - 2000 Scalar Columns (All Doubles); 1000 Rows
  Scenario fully completed  : true
  Scenario time limit (sec) : 900
  Scenario time active (sec): 938.939448
  Frame size avg. (bytes)   : 48024000
  Frame count offered       : 5
  Frame count sent          : 5
  Frame rate offered  (f/s) : 0.1
  Frame rate achieved (f/s) : 0.0053251570275914106
  Data size offered (bytes) : 240120000
  Data size sent    (bytes) : 240120000
  Data rate achieved  (b/s) : 255735.3410930499
  Execution exception       : null
  Provider Registration ID  : ProviderId(id=1, name=null, status=OpStatus(message=,
status=NO_STATUS, severity=NO_ALARM))
  Snapshot IDs :


TEST: testScenarioStringTables FROM com.ospreydcs.datastore.admin.IngestionStreamScenariosTest
2022-11-11T17:45:56.557555
Scenario STRING_TABLES Results
  Scenario description      : 41 MByte frames avg (10) - 500 Columns of 60 character string
(avg); 500 Rows
  Scenario fully completed  : true
  Scenario time limit (sec) : 400
  Scenario time active (sec): 408.37489
  Frame size avg. (bytes)   : 40934100
  Frame count offered       : 10
  Frame count sent          : 10
  Frame rate offered  (f/s) : 1.0
  Frame rate achieved (f/s) : 0.02448730381047669
```

```
  Data size offered (bytes) : 409341000
  Data size sent    (bytes) : 409341000
  Data rate achieved  (b/s) : 1002365.742908434
  Execution exception       : null
  Provider Registration ID  : ProviderId(id=1, name=null, status=OpStatus(message=,
status=NO_STATUS, severity=NO_ALARM))
  Snapshot IDs :


TEST: testScenarioArrayTables FROM com.ospreydcs.datastore.admin.IngestionStreamScenariosTest
2022-11-12T13:45:19.731035
Scenario ARRAY_TABLES Results
  Scenario description      : 32 MByte frames avg (10) - 1 Column of 32 Kbyte double arrays;
1,000 Rows
  Scenario fully completed  : true
  Scenario time limit (sec) : 600
  Scenario time active (sec): 48.203727
  Frame size avg. (bytes)   : 38379000
  Frame count offered       : 10
  Frame count sent          : 10
  Frame rate offered  (f/s) : 2.0
  Frame rate achieved (f/s) : 0.20745283865706068
  Data size offered (bytes) : 383790000
  Data size sent    (bytes) : 383790000
  Data rate achieved  (b/s) : 7961832.494819332
  Execution exception       : null
  Provider Registration ID  : ProviderId(id=1, name=null, status=OpStatus(message=,
status=NO_STATUS, severity=NO_ALARM))
  Snapshot IDs :
    SnapshotId(id=87, status=OpStatus(message=, status=NO_STATUS, severity=NO_ALARM))
```

ADDENDUM – Large Data Structures

Note the execution exception indicated a timeout waiting for the Datastore to respond to a close stream request. This indicates that the Datastore is still processing data and the data rates are likely inaccurate.

```
TEST: testScenario4LargeStructures FROM
com.ospreydcs.datastore.admin.IngestionStreamScenariosTest
2022-11-12T14:29:33.501276
Scenario LARGE_STRUCTURES Results
  Scenario description      : 8.2-Mbyte frames (10) - 10 columns of 81.8 KByte structures, 10
rows
  Scenario fully completed  : true
  Scenario time limit (sec) : 800
  Scenario time active (sec): 136.968987
  Frame size avg. (bytes)   : 9111573
  Frame count offered       : 75
  Frame count sent          : 75
  Frame rate offered  (f/s) : 10.0
  Frame rate achieved (f/s) : 0.5475692099555354
  Data size offered (bytes) : 683368040
  Data size sent    (bytes) : 683368040
  Data rate achieved  (b/s) : 4989217.303622169
  Execution exception       : java.util.concurrent.TimeoutException:
DsIngestionStream#awaitStreamEndResponse() - Timeout out waiting on response from Datastore
  Provider Registration ID  : ProviderId(id=1, name=null, status=OpStatus(message=,
status=NO_STATUS, severity=NO_ALARM))
  Snapshot IDs :
```

APPENDIX C: METADATA QUERY TESTS

```
TEST: testQuerySnapshots6CompareTimestamps FROM
com.ospreydcs.datastore.admin.model.IQueryServiceMetaTest
DataFrame Properties:
  duration = 10000000000
  period = 1000000000
  file = test-dataframe-scalars.yml
  name = Test DataFrame Scalars
  type = test data
  frequency = 1
  Frame timestamp: 2022-10-01T01:23:40.100Z
  First timestamp: 2022-10-01T01:23:45.100Z
  Last timestamp : 2022-10-01T01:23:54.100Z
Snapshot Query for snapshot with Attributes: {duration=10000000000, period=1000000000, file=test-
dataframe-scalars.yml, name=Test DataFrame Scalars, type=test data, frequency=1}
  Snapshot UID     : 1
  type = test data
  name = Test DataFrame Scalars
  period = 1000000000
  frequency = 1
  duration = 10000000000
  file = test-dataframe-scalars.yml
  Snapshot timestamp: 2022-10-01T01:23:40.100Z
  First timestamp   : 2022-10-01T01:23:40.100Z
  Last timestamp    : 2022-10-01T01:23:49.100Z
DataFrame Properties:
  duration = 10000000000
  period = 1000000000
  file = test-dataframe-arrays.yml
  name = Test DataFrame Arrays
  type = test data
  frequency = 1
  Frame timestamp: 2022-10-03T01:23:40.100Z
  First timestamp: 2022-10-03T01:23:45.100Z
  Last timestamp : 2022-10-03T01:23:54.100Z
Snapshot Query for snapshot with Attributes: {duration=10000000000, period=1000000000, file=test-
dataframe-arrays.yml, name=Test DataFrame Arrays, type=test data, frequency=1}
  Snapshot UID     : 2
  type = test data
  name = Test DataFrame Arrays
  period = 1000000000
  frequency = 1
  duration = 10000000000
  file = test-dataframe-arrays.yml
  Snapshot timestamp: 2022-10-03T01:23:40.100Z
  First timestamp   : 2022-10-03T01:23:40.100Z
  Last timestamp    : 2022-10-03T01:23:49.100Z
DataFrame Properties:
  duration = 10000000000
  period = 1000000000
  file = test-dataframe-structs.yml
  name = Test DataFrame Structures
  type = test data
  frequency = 1
  Frame timestamp: 2022-10-04T01:23:40.100Z
  First timestamp: 2022-10-04T01:23:45.100Z
  Last timestamp : 2022-10-04T01:23:49.100Z
Snapshot Query for snapshot with Attributes: {duration=10000000000, period=1000000000, file=test-
dataframe-structs.yml, name=Test DataFrame Structures, type=test data, frequency=1}
  Snapshot UID     : 3
  type = test data
  name = Test DataFrame Structures
  period = 1000000000
  frequency = 1
  duration = 10000000000
  file = test-dataframe-structs.yml
  Snapshot timestamp: 2022-10-04T01:23:40.100Z
  First timestamp   : 2022-10-04T01:23:40.100Z
  Last timestamp    : 2022-10-04T01:23:44.100Z
DataFrame Properties:
  duration = 100000000
  period = 1000000
  file = test-framefactory-basic.yml
  name = BASIC
  type = DataFrame Factory
  frequency = 1000.0
  Frame timestamp: 2022-12-07T21:09:49.479910Z
```

```
  First timestamp: 2022-12-07T21:09:49.470626Z
  Last timestamp : 2022-12-07T21:09:49.569626Z
Snapshot Query for snapshot with Attributes: {duration=100000000, period=1000000, file=test-
framefactory-basic.yml, name=BASIC, type=DataFrame Factory, frequency=1000.0}
  Snapshot UID     : 4
  type = DataFrame Factory
  name = BASIC
  period = 1000000
  frequency = 1000.0
  duration = 100000000
  file = test-framefactory-basic.yml
  Snapshot timestamp: 2022-12-07T21:09:49.479910Z
  First timestamp   : 2022-12-07T21:09:49.479910Z
  Last timestamp    : 2022-12-07T21:09:49.578910Z


TEST: testQuerySnapshots3ByTimestamp FROM
com.ospreydcs.datastore.admin.model.IQueryServiceMetaTest
All Snapshot Timestamps returned by data manager  :
  2022-10-01T01:23:40.100Z
  2022-10-03T01:23:40.100Z
  2022-10-04T01:23:40.100Z
  2022-12-07T21:09:49.479910Z
Snapshot Records for range after snapshot timestamp: 2022-10-01T01:23:40.100Z
  SnapshotRecord(id=2, timestamp=1664760220100000000, size=10, pvNames=[TEST-Array-PV00],
attributes={type=test data, name=Test DataFrame Arrays, period=1000000000, frequency=1,
duration=10000000000, file=test-dataframe-arrays.yml}, firstTimestamp=1664760220100000000,
lastTimestamp=1664760229100000000, dsManagerClass=null)
  [Snapshot ID=2, Timestamp=2022-10-03T01:23:40.100Z, 1st timestamp=2022-10-03T01:23:40.100Z,
Last timestamp=2022-10-03T01:23:49.100Z, ]
  SnapshotRecord(id=3, timestamp=1664846620100000000, size=5, pvNames=[TEST-Structure-PV00],
attributes={type=test data, name=Test DataFrame Structures, period=1000000000, frequency=1,
duration=10000000000, file=test-dataframe-structs.yml}, firstTimestamp=1664846620100000000,
lastTimestamp=1664846624100000000, dsManagerClass=null)
  [Snapshot ID=3, Timestamp=2022-10-04T01:23:40.100Z, 1st timestamp=2022-10-04T01:23:40.100Z,
Last timestamp=2022-10-04T01:23:44.100Z, ]
  SnapshotRecord(id=4, timestamp=1670447389479910000, size=100, pvNames=[Test-BASIC-SCALAR00,
Test-BASIC-SCALAR01, Test-BASIC-SCALAR02, Test-BASIC-SCALAR03, Test-BASIC-SCALAR04, Test-BASIC-
SCALAR05, Test-BASIC-SCALAR06, Test-BASIC-SCALAR07, Test-BASIC-SCALAR08, Test-BASIC-SCALAR09,
Test-BASIC-ARRAY00, Test-BASIC-ARRAY01, Test-BASIC-ARRAY02, Test-BASIC-ARRAY03, Test-BASIC-
ARRAY04], attributes={type=DataFrame Factory, name=BASIC, period=1000000, frequency=1000.0,
duration=100000000, file=test-framefactory-basic.yml}, firstTimestamp=1670447389479910000,
lastTimestamp=1670447389578910000, dsManagerClass=null)
  [Snapshot ID=4, Timestamp=2022-12-07T21:09:49.479910Z, 1st timestamp=2022-12-
07T21:09:49.479910Z, Last timestamp=2022-12-07T21:09:49.578910Z, ]

Snapshot Records for range after snapshot timestamp: 2022-10-03T01:23:40.100Z
  SnapshotRecord(id=3, timestamp=1664846620100000000, size=5, pvNames=[TEST-Structure-PV00],
attributes={type=test data, name=Test DataFrame Structures, period=1000000000, frequency=1,
duration=10000000000, file=test-dataframe-structs.yml}, firstTimestamp=1664846620100000000,
lastTimestamp=1664846624100000000, dsManagerClass=null)
  [Snapshot ID=3, Timestamp=2022-10-04T01:23:40.100Z, 1st timestamp=2022-10-04T01:23:40.100Z,
Last timestamp=2022-10-04T01:23:44.100Z, ]
  SnapshotRecord(id=4, timestamp=1670447389479910000, size=100, pvNames=[Test-BASIC-SCALAR00,
Test-BASIC-SCALAR01, Test-BASIC-SCALAR02, Test-BASIC-SCALAR03, Test-BASIC-SCALAR04, Test-BASIC-
SCALAR05, Test-BASIC-SCALAR06, Test-BASIC-SCALAR07, Test-BASIC-SCALAR08, Test-BASIC-SCALAR09,
Test-BASIC-ARRAY00, Test-BASIC-ARRAY01, Test-BASIC-ARRAY02, Test-BASIC-ARRAY03, Test-BASIC-
ARRAY04], attributes={type=DataFrame Factory, name=BASIC, period=1000000, frequency=1000.0,
duration=100000000, file=test-framefactory-basic.yml}, firstTimestamp=1670447389479910000,
lastTimestamp=1670447389578910000, dsManagerClass=null)
  [Snapshot ID=4, Timestamp=2022-12-07T21:09:49.479910Z, 1st timestamp=2022-12-
07T21:09:49.479910Z, Last timestamp=2022-12-07T21:09:49.578910Z, ]

Snapshot Records for range after snapshot timestamp: 2022-10-04T01:23:40.100Z
  SnapshotRecord(id=4, timestamp=1670447389479910000, size=100, pvNames=[Test-BASIC-SCALAR00,
Test-BASIC-SCALAR01, Test-BASIC-SCALAR02, Test-BASIC-SCALAR03, Test-BASIC-SCALAR04, Test-BASIC-
SCALAR05, Test-BASIC-SCALAR06, Test-BASIC-SCALAR07, Test-BASIC-SCALAR08, Test-BASIC-SCALAR09,
Test-BASIC-ARRAY00, Test-BASIC-ARRAY01, Test-BASIC-ARRAY02, Test-BASIC-ARRAY03, Test-BASIC-
ARRAY04], attributes={type=DataFrame Factory, name=BASIC, period=1000000, frequency=1000.0,
duration=100000000, file=test-framefactory-basic.yml}, firstTimestamp=1670447389479910000,
lastTimestamp=1670447389578910000, dsManagerClass=null)
  [Snapshot ID=4, Timestamp=2022-12-07T21:09:49.479910Z, 1st timestamp=2022-12-
07T21:09:49.479910Z, Last timestamp=2022-12-07T21:09:49.578910Z, ]

Snapshot Records for range after snapshot timestamp: 2022-12-07T21:09:49.479910Z

TEST: testQuerySnapshots4ByAttributes FROM
com.ospreydcs.datastore.admin.model.IQueryServiceMetaTest
Attributes used for Snapshot Query: {duration=10000000000, period=1000000000, file=test-
dataframe-scalars.yml, name=Test DataFrame Scalars, type=test data, frequency=1}
```

```
Snapshot Records for attributes query: {duration=10000000000, period=1000000000, file=test-
dataframe-scalars.yml, name=Test DataFrame Scalars, type=test data, frequency=1}
  SnapshotRecord(id=1, timestamp=1664587420100000000, size=10, pvNames=[TEST-PV00, TEST-PV01,
TEST-PV02, TEST-PV03, TEST-PV04], attributes={type=test data, name=Test DataFrame Scalars,
period=1000000000, frequency=1, duration=10000000000, file=test-dataframe-scalars.yml},
firstTimestamp=1664587420100000000, lastTimestamp=1664587429100000000, dsManagerClass=null)
  [Snapshot ID=1, Timestamp=2022-10-01T01:23:40.100Z, 1st timestamp=2022-10-01T01:23:40.100Z,
Last timestamp=2022-10-01T01:23:49.100Z, ]
```

APPENDIX D: SNAPSHOT DATA QUERY TESTS

```
TEST: testRequestData1Sync_IntegrityScalars FROM
com.ospreydcs.datastore.admin.model.IQueryServiceDataTest
  Query time (seconds)    : 0.29376
  Request size (bytes)    : 1120
  Data rate (bytes/second): 3812.6361655773417
Test Data Frame:
Snapshot Data Provider UID = null
DataFrame UID = null
DataFrame Timestamp = 2022-10-01T01:23:40.100Z
DataFrame Attributes = {duration=10000000000, period=1000000000, file=test-dataframe-scalars.yml,
name=Test DataFrame Scalars, type=test data, frequency=1}
timestamp      TEST-PV00      TEST-PV01      TEST-PV02      TEST-PV03      TEST-PV04
2022-10-01T01:23:45.100Z      true    0      0.0     str0    0.0
2022-10-01T01:23:46.100Z      true    1      0.1     str1    0.01
2022-10-01T01:23:47.100Z      true    2      0.2     str2    0.02
2022-10-01T01:23:48.100Z      true    3      0.3     str3    0.03
2022-10-01T01:23:49.100Z      true    4      0.4     str4    0.04
2022-10-01T01:23:50.100Z      true    5      0.5     str5    0.05
2022-10-01T01:23:51.100Z      true    6      0.6     str6    0.06
2022-10-01T01:23:52.100Z      true    7      0.7     str7    0.07
2022-10-01T01:23:53.100Z      true    8      0.8     str8    0.08
2022-10-01T01:23:54.100Z      false   9      0.9     str9    0.09

Open Query Result:
timestamp      TEST-PV00      TEST-PV02      TEST-PV01      TEST-PV04      TEST-PV03
2022-10-01T01:24:40.100Z      true    0.0     0       0.0     null
2022-10-01T01:24:41.100Z      true    0.1     1       0.01    null
2022-10-01T01:24:42.100Z      true    0.2     2       0.02    null
2022-10-01T01:24:43.100Z      true    0.3     3       0.03    null
2022-10-01T01:24:44.100Z      true    0.4     4       0.04    null
2022-10-01T01:24:45.100Z      true    0.5     5       0.05    null
2022-10-01T01:24:46.100Z      true    0.6     6       0.06    null
2022-10-01T01:24:47.100Z      true    0.7     7       0.07    null
2022-10-01T01:24:48.100Z      true    0.8     8       0.08    null
2022-10-01T01:24:49.100Z      false   0.9     9       0.09    null


Query results are missing providers: []
Timestamp maximum time difference: PT55S
Query results differ at the following locations:
  TEST-PV01: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
  TEST-PV03: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

TEST: testRequestData2Sync_IntegrityArrays FROM
com.ospreydcs.datastore.admin.model.IQueryServiceDataTest
  Query time (seconds)    : 0.03406
  Request size (bytes)    : 1660
  Data rate (bytes/second): 48737.52201996477
Test Data Frame:
Snapshot Data Provider UID = null
DataFrame UID = null
DataFrame Timestamp = 2022-10-03T01:23:40.100Z
DataFrame Attributes = {duration=10000000000, period=1000000000, file=test-dataframe-arrays.yml,
name=Test DataFrame Arrays, type=test data, frequency=1}
timestamp      TEST-Array-PV00
2022-10-03T01:23:45.100Z      [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
2022-10-03T01:23:46.100Z      [1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9]
2022-10-03T01:23:47.100Z      [2.0, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9]
2022-10-03T01:23:48.100Z      [3.3, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9]
2022-10-03T01:23:49.100Z      [4.0, 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9]
2022-10-03T01:23:50.100Z      [5.0, 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9]
2022-10-03T01:23:51.100Z      [6.0, 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, 6.9]
2022-10-03T01:23:52.100Z      [7.0, 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.7, 7.8, 7.9]
2022-10-03T01:23:53.100Z      [8.0, 8.1, 8.2, 8.3, 8.4, 8.5, 8.6, 8.7, 8.8, 8.9]
2022-10-03T01:23:54.100Z      [9.0, 9.1, 9.2, 9.3, 9.4, 9.5, 9.6, 9.7, 9.8, 9.9]

Open Query Result:
timestamp      TEST-Array-PV00
2022-10-03T01:24:40.100Z      [0, null, null, null, null, null, null, null, null, null]
2022-10-03T01:24:41.100Z      [1, null, null, null, null, null, null, null, null, null]
2022-10-03T01:24:42.100Z      [2, null, null, null, null, null, null, null, null, null]
2022-10-03T01:24:43.100Z      [null, null, null, null, null, null, null, null, null, null]
2022-10-03T01:24:44.100Z      [4, null, null, null, null, null, null, null, null, null]
2022-10-03T01:24:45.100Z      [5, null, null, null, null, null, null, null, null, null]
2022-10-03T01:24:46.100Z      [6, null, null, null, null, null, null, null, null, null]
```

```
2022-10-03T01:24:47.100Z      [7, null, null, null, null, null, null, null, null, null]
2022-10-03T01:24:48.100Z      [8, null, null, null, null, null, null, null, null, null]
2022-10-03T01:24:49.100Z      [9, null, null, null, null, null, null, null, null, null]


Query results are missing providers: []
Timestamp maximum time difference: PT55S
Query results differ at the following locations:
  TEST-Array-PV00: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

TEST: testRequestData3Sync_IntegrityStructs FROM
com.ospreydcs.datastore.admin.model.IQueryServiceDataTest
  Query time (seconds)    : 0.05065600000000001
  Request size (bytes)    : 240
  Data rate (bytes/second): 4737.839545167403
Test Data Frame:
Snapshot Data Provider UID = null
DataFrame UID = null
DataFrame Timestamp = 2022-10-04T01:23:40.100Z
DataFrame Attributes = {duration=10000000000, period=1000000000, file=test-dataframe-structs.yml,
name=Test DataFrame Structures, type=test data, frequency=1}
timestamp      TEST-Structure-PV00
2022-10-04T01:23:45.100Z      {f1=0.0}
2022-10-04T01:23:46.100Z      {g1=1.0, g2=2.0, g3=3.0, g4=4.0, g5=5.0}
2022-10-04T01:23:47.100Z      {h1=1.0, h2={h21=0.1, h22=0.2}, h3={h31=0.1, h32=0.2, h33=0.3}}
2022-10-04T01:23:48.100Z      {i1=1.0, i2={i21=0.1, i22=0.2}, i3={i31=0.1, i32={i321=0.01,
i322={i3221=0.001, i3222=0.002}}}}
2022-10-04T01:23:49.100Z      {j1={j11={j111={j1111={j11111=1.0E-5, j11112=2.0E-5, j11113=3.0E-
5}}}}}

Open Query Result:
timestamp      TEST-Structure-PV00
2022-10-04T01:24:10.100Z      0.0
2022-10-04T01:24:11.100Z      {g1=1.0, g2=2.0, g3=3.0, g4=4.0, g5=5.0}
2022-10-04T01:24:12.100Z      {h1=1.0, h2={h21=null, h22=null}, h3={h31=null, h32=null,
h33=null}}
2022-10-04T01:24:13.100Z      {i1=1.0, i2={i21=null, i22=null}, i3={i31=null, i32=null}}
2022-10-04T01:24:14.100Z      {j11=null}


Query results are missing providers: []
Timestamp maximum time difference: PT25S
Query results differ at the following locations:
  TEST-Structure-PV00: [0, 2, 3, 4]

TEST: testRequestData3Sync_IntegrityStructs2 FROM
com.ospreydcs.datastore.admin.model.IQueryServiceDataTest
  Query time (seconds)    : 0.038505000000000005
  Request size (bytes)    : 4305
  Data rate (bytes/second): 111803.66186209582
Test Data Frame:
Snapshot Data Provider UID = null
DataFrame UID = null
DataFrame Timestamp = 2022-10-05T01:23:40.100Z
DataFrame Attributes = {duration=10000000000, period=1000000000, file=test-dataframe-
structs2.yml, name=Test DataFrame Structures 2, type=test data, frequency=1}
timestamp      TEST-Structure2-PV00   TEST-Structure2-PV01   TEST-Structure2-PV02   TEST-
Structure2-PV03        TEST-Structure2-PV04
2022-10-05T01:23:45.100Z      {f1=0.0}       {g1=1.0, g2=2.0, g3=3.0, g4=4.0, g5=5.0}
       {h1=1.0, h2={h21=1.1, h22=1.2}, h3={h31=1.01, h32=1.02, h33=1.03}} {i1=1.0, i2={i21=1.1,
i22=1.2}, i3={i31=0.1, i32={i321=0.01, i322={i3221=0.001, i3222=1.002}}}}
       {j1={j11={j111={j1111={j11111=1.00001, j11112=1.00002, j11113=1.00003}}}}}
2022-10-05T01:23:46.100Z      {f1=1.0}       {g1=1.1, g2=2.1, g3=3.1, g4=4.1, g5=5.1}
       {h1=2.0, h2={h21=2.1, h22=2.2}, h3={h31=2.01, h32=2.02, h33=2.03}} {i1=2.0, i2={i21=2.1,
i22=2.2}, i3={i31=0.2, i32={i321=0.02, i322={i3221=0.002, i3222=2.002}}}}
       {j1={j11={j111={j1111={j11111=2.00001, j11112=2.00002, j11113=2.00003}}}}}
2022-10-05T01:23:47.100Z      {f1=2.0}       {g1=1.2, g2=2.2, g3=3.2, g4=4.2, g5=5.2}
       {h1=3.0, h2={h21=3.1, h22=3.2}, h3={h31=3.01, h32=3.02, h33=3.03}} {i1=3.0, i2={i21=3.1,
i22=3.2}, i3={i31=0.3, i32={i321=0.03, i322={i3221=0.003, i3222=3.002}}}}
       {j1={j11={j111={j1111={j11111=3.00001, j11112=3.00002, j11113=3.00003}}}}}
2022-10-05T01:23:48.100Z      {f1=3.0}       {g1=1.3, g2=2.3, g3=3.3, g4=4.3, g5=5.3}
       {h1=4.0, h2={h21=4.1, h22=4.2}, h3={h31=4.01, h32=4.02, h33=4.03}} {i1=4.0, i2={i21=4.1,
i22=4.2}, i3={i31=0.4, i32={i321=0.04, i322={i3221=0.004, i3222=4.002}}}}
       {j1={j11={j111={j1111={j11111=4.00001, j11112=4.00002, j11113=4.00003}}}}}
2022-10-05T01:23:49.100Z      {f1=4.0}       {g1=1.4, g2=2.4, g3=3.4, g4=4.4, g5=5.4}
       {h1=5.0, h2={h21=5.1, h22=5.2}, h3={h31=5.01, h32=5.02, h33=5.03}} {i1=5.0, i2={i21=5.1,
i22=5.2}, i3={i31=0.5, i32={i321=0.05, i322={i3221=0.005, i3222=5.002}}}}
       {j1={j11={j111={j1111={j11111=5.00001, j11112=5.00002, j11113=5.00003}}}}}

Open Query Result:
```

```
timestamp      TEST-Structure2-PV03  TEST-Structure2-PV04  TEST-Structure2-PV00  TEST-
Structure2-PV01       TEST-Structure2-PV02
2022-10-05T01:24:10.100Z      {i1=1.0, i2={i21=null, i22=null}, i3={i31=null, i32=null}}
       {j11=null}     0.0     {g1=1.0, g2=2.0, g3=3.0, g4=4.0, g5=5.0}     {h1=1.0, h2={h21=null,
h22=null}, h3={h31=null, h32=null, h33=null}}
2022-10-05T01:24:11.100Z      {i1=2.0, i2={i21=null, i22=null}, i3={i31=null, i32=null}}
       {j11=null}     1.0     {g1=1.1, g2=2.1, g3=3.1, g4=4.1, g5=5.1}     {h1=2.0, h2={h21=null,
h22=null}, h3={h31=null, h32=null, h33=null}}
2022-10-05T01:24:12.100Z      {i1=3.0, i2={i21=null, i22=null}, i3={i31=null, i32=null}}
       {j11=null}     2.0     {g1=1.2, g2=2.2, g3=3.2, g4=4.2, g5=5.2}     {h1=3.0, h2={h21=null,
h22=null}, h3={h31=null, h32=null, h33=null}}
2022-10-05T01:24:13.100Z      {i1=4.0, i2={i21=null, i22=null}, i3={i31=null, i32=null}}
       {j11=null}     3.0     {g1=1.3, g2=2.3, g3=3.3, g4=4.3, g5=5.3}     {h1=4.0, h2={h21=null,
h22=null}, h3={h31=null, h32=null, h33=null}}
2022-10-05T01:24:14.100Z      {i1=5.0, i2={i21=null, i22=null}, i3={i31=null, i32=null}}
       {j11=null}     4.0     {g1=1.4, g2=2.4, g3=3.4, g4=4.4, g5=5.4}     {h1=5.0, h2={h21=null,
h22=null}, h3={h31=null, h32=null, h33=null}}


Query results are missing providers: []
Timestamp maximum time difference: PT25S
Query results differ at the following locations:
  TEST-Structure2-PV03: [0, 1, 2, 3, 4]
  TEST-Structure2-PV04: [0, 1, 2, 3, 4]
  TEST-Structure2-PV00: [0, 1, 2, 3, 4]
  TEST-Structure2-PV02: [0, 1, 2, 3, 4]

TEST: testRequestData4Sync_ScalarTable500 FROM
com.ospreydcs.datastore.admin.model.IQueryServiceDataTest
Single Data Frame: 500 PV wide frame, double values
Number of frames sent to Datastore : 1
Test data frame column count       : 500
Test data frame row count          : 100
Test data frame value count        : 50000
Test data frame allocation (bytes) : 1202400
Total number of rows in Datastore  : 100
Total number of values in Datastore: 50000
Total data in Datastore (bytes)    : 1202400
Result table column count      : 500
Result table row count         : 100
Result table value count total : 50000
Result table null value count  : 0
Result table allocation (bytes): 1202400
Query duration                 : PT2.02864S
Query time (seconds)           : 2.02864
Query data rate (bytes/second) : 592712.35901885

Query results are missing providers: []
Timestamp maximum time difference: PT-40H-57M-40.707555S
Query results differ at the following locations:

Callback Function Test: 500 wide frame, double values
Open query request at 2022-12-10T14:34:47.372199Z
Table loading wait completed at 2022-12-10T14:34:48.764599Z
Total loading wait duration : PT1.3924S
Result table allocation (bytes): 1202400
Query data rate (bytes/second) : 863544.9583453031
FncNotify(Boolean, IDataTableDynamic) called at 2022-12-10T14:34:48.764589Z
  bolSuccess = true
  (tblResult==dtdResult) = true
  Callback duration : PT1.39239S
  Result table column count      : 500
  Result table row count         : 100
  Result table value count total : 50000
  Result table null value count  : 0
  Result table allocation (bytes): 1202400
  Query duration                 : PT1.39239S
  Query time (seconds)           : 1.39239
  Query data rate (bytes/second) : 863551.1602352789

Single PV Select Query:
Single query request        : SELECT `Test-Table500-DOUBLE499.value`  WHERE time >= '1970-01-
01T00:00:00Z'
Total loading wait duration : PT0.813552S
Total wait time (seconds)   : 0.813552
Result allocation (bytes)   : 4800
Data rate (bytes/second)    : 5900.053100477904

PV Timestamps                  PV Values  Query Timestamps              Query Values
2022-12-10T14:33:46.226762Z    0.931985   2022-12-08T21:36:05.519207Z    0.931985
2022-12-10T14:33:46.226763Z    0.627492   2022-12-08T21:36:05.519208Z    0.627492
```

2022-12-10T14:33:46.226764Z	0.109195	2022-12-08T21:36:05.519209Z	0.109195
2022-12-10T14:33:46.226765Z	0.927798	2022-12-08T21:36:05.519210Z	0.927798
2022-12-10T14:33:46.226766Z	0.570167	2022-12-08T21:36:05.519211Z	0.570167
2022-12-10T14:33:46.226767Z	0.285980	2022-12-08T21:36:05.519212Z	0.285980
2022-12-10T14:33:46.226768Z	0.420664	2022-12-08T21:36:05.519213Z	0.420664
2022-12-10T14:33:46.226769Z	0.178648	2022-12-08T21:36:05.519214Z	0.178648
2022-12-10T14:33:46.226770Z	0.640004	2022-12-08T21:36:05.519215Z	0.640004
2022-12-10T14:33:46.226771Z	0.239614	2022-12-08T21:36:05.519216Z	0.239614
2022-12-10T14:33:46.226772Z	0.848371	2022-12-08T21:36:05.519217Z	0.848371
2022-12-10T14:33:46.226773Z	0.809563	2022-12-08T21:36:05.519218Z	0.809563
2022-12-10T14:33:46.226774Z	0.356303	2022-12-08T21:36:05.519219Z	0.356303
2022-12-10T14:33:46.226775Z	0.844814	2022-12-08T21:36:05.519220Z	0.844814
2022-12-10T14:33:46.226776Z	0.254462	2022-12-08T21:36:05.519221Z	0.254462
2022-12-10T14:33:46.226777Z	0.0338123	2022-12-08T21:36:05.519222Z	0.0338123
2022-12-10T14:33:46.226778Z	0.721949	2022-12-08T21:36:05.519223Z	0.721949
2022-12-10T14:33:46.226779Z	0.641291	2022-12-08T21:36:05.519224Z	0.641291
2022-12-10T14:33:46.226780Z	0.661831	2022-12-08T21:36:05.519225Z	0.661831
2022-12-10T14:33:46.226781Z	0.260617	2022-12-08T21:36:05.519226Z	0.260617
2022-12-10T14:33:46.226782Z	0.237786	2022-12-08T21:36:05.519227Z	0.237786
2022-12-10T14:33:46.226783Z	0.368218	2022-12-08T21:36:05.519228Z	0.368218
2022-12-10T14:33:46.226784Z	0.0506968	2022-12-08T21:36:05.519229Z	0.0506968
2022-12-10T14:33:46.226785Z	0.169267	2022-12-08T21:36:05.519230Z	0.169267
2022-12-10T14:33:46.226786Z	0.114390	2022-12-08T21:36:05.519231Z	0.114390
2022-12-10T14:33:46.226787Z	0.545707	2022-12-08T21:36:05.519232Z	0.545707
2022-12-10T14:33:46.226788Z	0.661868	2022-12-08T21:36:05.519233Z	0.661868
2022-12-10T14:33:46.226789Z	0.0765773	2022-12-08T21:36:05.519234Z	0.0765773
2022-12-10T14:33:46.226790Z	0.224069	2022-12-08T21:36:05.519235Z	0.224069
2022-12-10T14:33:46.226791Z	0.722427	2022-12-08T21:36:05.519236Z	0.722427
2022-12-10T14:33:46.226792Z	0.600156	2022-12-08T21:36:05.519237Z	0.600156
2022-12-10T14:33:46.226793Z	0.241992	2022-12-08T21:36:05.519238Z	0.241992
2022-12-10T14:33:46.226794Z	0.899537	2022-12-08T21:36:05.519239Z	0.899537
2022-12-10T14:33:46.226795Z	0.147469	2022-12-08T21:36:05.519240Z	0.147469
2022-12-10T14:33:46.226796Z	0.0402431	2022-12-08T21:36:05.519241Z	0.0402431
2022-12-10T14:33:46.226797Z	0.677517	2022-12-08T21:36:05.519242Z	0.677517
2022-12-10T14:33:46.226798Z	0.0851805	2022-12-08T21:36:05.519243Z	0.0851805
2022-12-10T14:33:46.226799Z	0.0630171	2022-12-08T21:36:05.519244Z	0.0630171
2022-12-10T14:33:46.226800Z	0.877298	2022-12-08T21:36:05.519245Z	0.877298
2022-12-10T14:33:46.226801Z	0.233741	2022-12-08T21:36:05.519246Z	0.233741
2022-12-10T14:33:46.226802Z	0.277443	2022-12-08T21:36:05.519247Z	0.277443
2022-12-10T14:33:46.226803Z	0.0341104	2022-12-08T21:36:05.519248Z	0.0341104
2022-12-10T14:33:46.226804Z	0.235957	2022-12-08T21:36:05.519249Z	0.235957
2022-12-10T14:33:46.226805Z	0.218579	2022-12-08T21:36:05.519250Z	0.218579
2022-12-10T14:33:46.226806Z	0.152819	2022-12-08T21:36:05.519251Z	0.152819
2022-12-10T14:33:46.226807Z	0.732770	2022-12-08T21:36:05.519252Z	0.732770
2022-12-10T14:33:46.226808Z	0.853808	2022-12-08T21:36:05.519253Z	0.853808
2022-12-10T14:33:46.226809Z	0.571947	2022-12-08T21:36:05.519254Z	0.571947
2022-12-10T14:33:46.226810Z	0.775079	2022-12-08T21:36:05.519255Z	0.775079
2022-12-10T14:33:46.226811Z	0.375982	2022-12-08T21:36:05.519256Z	0.375982
2022-12-10T14:33:46.226812Z	0.989312	2022-12-08T21:36:05.519257Z	0.989312
2022-12-10T14:33:46.226813Z	0.667459	2022-12-08T21:36:05.519258Z	0.667459
2022-12-10T14:33:46.226814Z	0.797767	2022-12-08T21:36:05.519259Z	0.797767
2022-12-10T14:33:46.226815Z	0.580500	2022-12-08T21:36:05.519260Z	0.580500
2022-12-10T14:33:46.226816Z	0.169139	2022-12-08T21:36:05.519261Z	0.169139
2022-12-10T14:33:46.226817Z	0.141311	2022-12-08T21:36:05.519262Z	0.141311
2022-12-10T14:33:46.226818Z	0.133696	2022-12-08T21:36:05.519263Z	0.133696
2022-12-10T14:33:46.226819Z	0.742178	2022-12-08T21:36:05.519264Z	0.742178
2022-12-10T14:33:46.226820Z	0.786170	2022-12-08T21:36:05.519265Z	0.786170
2022-12-10T14:33:46.226821Z	0.758891	2022-12-08T21:36:05.519266Z	0.758891
2022-12-10T14:33:46.226822Z	0.150048	2022-12-08T21:36:05.519267Z	0.150048
2022-12-10T14:33:46.226823Z	0.173491	2022-12-08T21:36:05.519268Z	0.173491
2022-12-10T14:33:46.226824Z	0.286970	2022-12-08T21:36:05.519269Z	0.286970
2022-12-10T14:33:46.226825Z	0.221234	2022-12-08T21:36:05.519270Z	0.221234
2022-12-10T14:33:46.226826Z	0.565877	2022-12-08T21:36:05.519271Z	0.565877
2022-12-10T14:33:46.226827Z	0.140156	2022-12-08T21:36:05.519272Z	0.140156
2022-12-10T14:33:46.226828Z	0.446323	2022-12-08T21:36:05.519273Z	0.446323
2022-12-10T14:33:46.226829Z	0.986315	2022-12-08T21:36:05.519274Z	0.986315
2022-12-10T14:33:46.226830Z	0.878222	2022-12-08T21:36:05.519275Z	0.878222
2022-12-10T14:33:46.226831Z	0.626218	2022-12-08T21:36:05.519276Z	0.626218
2022-12-10T14:33:46.226832Z	0.0164554	2022-12-08T21:36:05.519277Z	0.0164554
2022-12-10T14:33:46.226833Z	0.634250	2022-12-08T21:36:05.519278Z	0.634250
2022-12-10T14:33:46.226834Z	0.323680	2022-12-08T21:36:05.519279Z	0.323680
2022-12-10T14:33:46.226835Z	0.126711	2022-12-08T21:36:05.519280Z	0.126711
2022-12-10T14:33:46.226836Z	0.859863	2022-12-08T21:36:05.519281Z	0.859863
2022-12-10T14:33:46.226837Z	0.0408606	2022-12-08T21:36:05.519282Z	0.0408606
2022-12-10T14:33:46.226838Z	0.0942359	2022-12-08T21:36:05.519283Z	0.0942359
2022-12-10T14:33:46.226839Z	0.0179800	2022-12-08T21:36:05.519284Z	0.0179800
2022-12-10T14:33:46.226840Z	0.527199	2022-12-08T21:36:05.519285Z	0.527199
2022-12-10T14:33:46.226841Z	0.640610	2022-12-08T21:36:05.519286Z	0.640610
2022-12-10T14:33:46.226842Z	0.769288	2022-12-08T21:36:05.519287Z	0.769288
2022-12-10T14:33:46.226843Z	0.0879469	2022-12-08T21:36:05.519288Z	0.0879469
2022-12-10T14:33:46.226844Z	0.125475	2022-12-08T21:36:05.519289Z	0.125475

```
2022-12-10T14:33:46.226845Z    0.357824   2022-12-08T21:36:05.519290Z    0.357824
2022-12-10T14:33:46.226846Z    0.700222   2022-12-08T21:36:05.519291Z    0.700222
2022-12-10T14:33:46.226847Z    0.622132   2022-12-08T21:36:05.519292Z    0.622132
2022-12-10T14:33:46.226848Z    0.236842   2022-12-08T21:36:05.519293Z    0.236842
2022-12-10T14:33:46.226849Z    0.364177   2022-12-08T21:36:05.519294Z    0.364177
2022-12-10T14:33:46.226850Z    0.389569   2022-12-08T21:36:05.519295Z    0.389569
2022-12-10T14:33:46.226851Z    0.0359182  2022-12-08T21:36:05.519296Z    0.0359182
2022-12-10T14:33:46.226852Z    0.594551   2022-12-08T21:36:05.519297Z    0.594551
2022-12-10T14:33:46.226853Z    0.680774   2022-12-08T21:36:05.519298Z    0.680774
2022-12-10T14:33:46.226854Z    0.743667   2022-12-08T21:36:05.519299Z    0.743667
2022-12-10T14:33:46.226855Z    0.419551   2022-12-08T21:36:05.519300Z    0.419551
2022-12-10T14:33:46.226856Z    0.200178   2022-12-08T21:36:05.519301Z    0.200178
2022-12-10T14:33:46.226857Z    0.784794   2022-12-08T21:36:05.519302Z    0.784794
2022-12-10T14:33:46.226858Z    0.445948   2022-12-08T21:36:05.519303Z    0.445948
2022-12-10T14:33:46.226859Z    0.519590   2022-12-08T21:36:05.519304Z    0.519590
2022-12-10T14:33:46.226860Z    0.575035   2022-12-08T21:36:05.519305Z    0.575035
2022-12-10T14:33:46.226861Z    0.803461   2022-12-08T21:36:05.519306Z    0.803461

TEST: testRequestData5Sync_ScalarTable1k FROM
com.ospreydcs.datastore.admin.model.IQueryServiceDataTest
Single Data Frame: 1,000 PV wide frame, double values
Number of frames sent to Datastore : 1
Test data frame column count       : 1000
Test data frame row count          : 100
Test data frame value count        : 100000
Test data frame allocation (bytes) : 2402400
Total number of rows in Datastore  : 100
Total number of values in Datastore: 100000
Total data in Datastore (bytes)    : 2402400
Result table column count      : 1000
Result table row count         : 100
Result table value count total : 100000
Result table null value count  : 0
Result table allocation (bytes): 2402400
Query duration                 : PT1.510836S
Query time (seconds)           : 1.510836
Query data rate (bytes/second) : 1590113.023518105

Query results are missing providers: []
Timestamp maximum time difference: PT-40H-57M-41.182807S
Query results differ at the following locations:

Callback Function Test: 1,000 PV wide frame, double values
Open query request at 2022-12-10T14:36:51.552179Z
Table loading wait completed at 2022-12-10T14:36:53.426545Z
Total loading wait duration : PT1.874366S
Result table allocation (bytes): 2402400
Query data rate (bytes/second) : 1281713.3900209456
FncNotify(Boolean, IDataTableDynamic) called at 2022-12-10T14:36:53.426531Z
  bolSuccess = true
  (tblResult==dtdResult) = true
  Callback duration : PT1.874352S
  Result table column count      : 1000
  Result table row count         : 100
  Result table value count total : 100000
  Result table null value count  : 0
  Result table allocation (bytes): 2402400
  Query duration                 : PT1.874352S
  Query time (seconds)           : 1.874352
  Query data rate (bytes/second) : 1281722.9634561704

Single PV Select Query:
Single query request        : SELECT `Test-Table1K-DOUBLE999.value`  WHERE time >= '1970-01-
01T00:00:00Z'
Total loading wait duration : PT0.893523S
Total wait time (seconds)   : 0.8935230000000001
Result allocation (bytes)   : 4800
Data rate (bytes/second)    : 5371.993781917197

PV Timestamps                  PV Values  Query Timestamps               Query Values
2022-12-10T14:34:52.705941Z    0.0250501  2022-12-08T21:37:11.523134Z    0.0250501
2022-12-10T14:34:52.705942Z    0.580117   2022-12-08T21:37:11.523135Z    0.580117
2022-12-10T14:34:52.705943Z    0.465856   2022-12-08T21:37:11.523136Z    0.465856
2022-12-10T14:34:52.705944Z    0.395095   2022-12-08T21:37:11.523137Z    0.395095
2022-12-10T14:34:52.705945Z    0.668779   2022-12-08T21:37:11.523138Z    0.668779
2022-12-10T14:34:52.705946Z    0.961178   2022-12-08T21:37:11.523139Z    0.961178
2022-12-10T14:34:52.705947Z    0.266544   2022-12-08T21:37:11.523140Z    0.266544
2022-12-10T14:34:52.705948Z    0.136619   2022-12-08T21:37:11.523141Z    0.136619
2022-12-10T14:34:52.705949Z    0.390005   2022-12-08T21:37:11.523142Z    0.390005
2022-12-10T14:34:52.705950Z    0.395384   2022-12-08T21:37:11.523143Z    0.395384
2022-12-10T14:34:52.705951Z    0.471163   2022-12-08T21:37:11.523144Z    0.471163
```

2022-12-10T14:34:52.705952Z	0.0353424	2022-12-08T21:37:11.523145Z	0.0353424
2022-12-10T14:34:52.705953Z	0.725533	2022-12-08T21:37:11.523146Z	0.725533
2022-12-10T14:34:52.705954Z	0.216469	2022-12-08T21:37:11.523147Z	0.216469
2022-12-10T14:34:52.705955Z	0.718766	2022-12-08T21:37:11.523148Z	0.718766
2022-12-10T14:34:52.705956Z	0.371872	2022-12-08T21:37:11.523149Z	0.371872
2022-12-10T14:34:52.705957Z	0.113064	2022-12-08T21:37:11.523150Z	0.113064
2022-12-10T14:34:52.705958Z	0.109058	2022-12-08T21:37:11.523151Z	0.109058
2022-12-10T14:34:52.705959Z	0.968012	2022-12-08T21:37:11.523152Z	0.968012
2022-12-10T14:34:52.705960Z	0.485850	2022-12-08T21:37:11.523153Z	0.485850
2022-12-10T14:34:52.705961Z	0.112406	2022-12-08T21:37:11.523154Z	0.112406
2022-12-10T14:34:52.705962Z	0.818364	2022-12-08T21:37:11.523155Z	0.818364
2022-12-10T14:34:52.705963Z	0.523466	2022-12-08T21:37:11.523156Z	0.523466
2022-12-10T14:34:52.705964Z	0.0363931	2022-12-08T21:37:11.523157Z	0.0363931
2022-12-10T14:34:52.705965Z	0.153244	2022-12-08T21:37:11.523158Z	0.153244
2022-12-10T14:34:52.705966Z	0.139289	2022-12-08T21:37:11.523159Z	0.139289
2022-12-10T14:34:52.705967Z	0.675251	2022-12-08T21:37:11.523160Z	0.675251
2022-12-10T14:34:52.705968Z	0.883254	2022-12-08T21:37:11.523161Z	0.883254
2022-12-10T14:34:52.705969Z	0.524335	2022-12-08T21:37:11.523162Z	0.524335
2022-12-10T14:34:52.705970Z	0.324847	2022-12-08T21:37:11.523163Z	0.324847
2022-12-10T14:34:52.705971Z	0.847131	2022-12-08T21:37:11.523164Z	0.847131
2022-12-10T14:34:52.705972Z	0.897094	2022-12-08T21:37:11.523165Z	0.897094
2022-12-10T14:34:52.705973Z	0.437186	2022-12-08T21:37:11.523166Z	0.437186
2022-12-10T14:34:52.705974Z	0.961075	2022-12-08T21:37:11.523167Z	0.961075
2022-12-10T14:34:52.705975Z	0.906838	2022-12-08T21:37:11.523168Z	0.906838
2022-12-10T14:34:52.705976Z	0.180098	2022-12-08T21:37:11.523169Z	0.180098
2022-12-10T14:34:52.705977Z	0.646649	2022-12-08T21:37:11.523170Z	0.646649
2022-12-10T14:34:52.705978Z	0.0440409	2022-12-08T21:37:11.523171Z	0.0440409
2022-12-10T14:34:52.705979Z	0.0825397	2022-12-08T21:37:11.523172Z	0.0825397
2022-12-10T14:34:52.705980Z	0.632258	2022-12-08T21:37:11.523173Z	0.632258
2022-12-10T14:34:52.705981Z	0.291557	2022-12-08T21:37:11.523174Z	0.291557
2022-12-10T14:34:52.705982Z	0.464305	2022-12-08T21:37:11.523175Z	0.464305
2022-12-10T14:34:52.705983Z	0.750602	2022-12-08T21:37:11.523176Z	0.750602
2022-12-10T14:34:52.705984Z	0.534020	2022-12-08T21:37:11.523177Z	0.534020
2022-12-10T14:34:52.705985Z	0.330433	2022-12-08T21:37:11.523178Z	0.330433
2022-12-10T14:34:52.705986Z	0.355177	2022-12-08T21:37:11.523179Z	0.355177
2022-12-10T14:34:52.705987Z	0.212365	2022-12-08T21:37:11.523180Z	0.212365
2022-12-10T14:34:52.705988Z	0.832573	2022-12-08T21:37:11.523181Z	0.832573
2022-12-10T14:34:52.705989Z	0.190459	2022-12-08T21:37:11.523182Z	0.190459
2022-12-10T14:34:52.705990Z	0.793303	2022-12-08T21:37:11.523183Z	0.793303
2022-12-10T14:34:52.705991Z	0.461415	2022-12-08T21:37:11.523184Z	0.461415
2022-12-10T14:34:52.705992Z	0.116918	2022-12-08T21:37:11.523185Z	0.116918
2022-12-10T14:34:52.705993Z	0.0223298	2022-12-08T21:37:11.523186Z	0.0223298
2022-12-10T14:34:52.705994Z	0.153638	2022-12-08T21:37:11.523187Z	0.153638
2022-12-10T14:34:52.705995Z	0.857897	2022-12-08T21:37:11.523188Z	0.857897
2022-12-10T14:34:52.705996Z	0.845968	2022-12-08T21:37:11.523189Z	0.845968
2022-12-10T14:34:52.705997Z	0.259241	2022-12-08T21:37:11.523190Z	0.259241
2022-12-10T14:34:52.705998Z	0.0155398	2022-12-08T21:37:11.523191Z	0.0155398
2022-12-10T14:34:52.705999Z	0.572688	2022-12-08T21:37:11.523192Z	0.572688
2022-12-10T14:34:52.706Z	0.254253	2022-12-08T21:37:11.523193Z	0.254253
2022-12-10T14:34:52.706001Z	0.0152195	2022-12-08T21:37:11.523194Z	0.0152195
2022-12-10T14:34:52.706002Z	0.892744	2022-12-08T21:37:11.523195Z	0.892744
2022-12-10T14:34:52.706003Z	0.641789	2022-12-08T21:37:11.523196Z	0.641789
2022-12-10T14:34:52.706004Z	0.501431	2022-12-08T21:37:11.523197Z	0.501431
2022-12-10T14:34:52.706005Z	0.226134	2022-12-08T21:37:11.523198Z	0.226134
2022-12-10T14:34:52.706006Z	0.217160	2022-12-08T21:37:11.523199Z	0.217160
2022-12-10T14:34:52.706007Z	0.480099	2022-12-08T21:37:11.523200Z	0.480099
2022-12-10T14:34:52.706008Z	0.0546550	2022-12-08T21:37:11.523201Z	0.0546550
2022-12-10T14:34:52.706009Z	0.336155	2022-12-08T21:37:11.523202Z	0.336155
2022-12-10T14:34:52.706010Z	0.944941	2022-12-08T21:37:11.523203Z	0.944941
2022-12-10T14:34:52.706011Z	0.743743	2022-12-08T21:37:11.523204Z	0.743743
2022-12-10T14:34:52.706012Z	0.996932	2022-12-08T21:37:11.523205Z	0.996932
2022-12-10T14:34:52.706013Z	0.763098	2022-12-08T21:37:11.523206Z	0.763098
2022-12-10T14:34:52.706014Z	0.872989	2022-12-08T21:37:11.523207Z	0.872989
2022-12-10T14:34:52.706015Z	0.501454	2022-12-08T21:37:11.523208Z	0.501454
2022-12-10T14:34:52.706016Z	0.939975	2022-12-08T21:37:11.523209Z	0.939975
2022-12-10T14:34:52.706017Z	0.337980	2022-12-08T21:37:11.523210Z	0.337980
2022-12-10T14:34:52.706018Z	0.255034	2022-12-08T21:37:11.523211Z	0.255034
2022-12-10T14:34:52.706019Z	0.778020	2022-12-08T21:37:11.523212Z	0.778020
2022-12-10T14:34:52.706020Z	0.486645	2022-12-08T21:37:11.523213Z	0.486645
2022-12-10T14:34:52.706021Z	0.407308	2022-12-08T21:37:11.523214Z	0.407308
2022-12-10T14:34:52.706022Z	0.588663	2022-12-08T21:37:11.523215Z	0.588663
2022-12-10T14:34:52.706023Z	0.666589	2022-12-08T21:37:11.523216Z	0.666589
2022-12-10T14:34:52.706024Z	0.735943	2022-12-08T21:37:11.523217Z	0.735943
2022-12-10T14:34:52.706025Z	0.311020	2022-12-08T21:37:11.523218Z	0.311020
2022-12-10T14:34:52.706026Z	0.508135	2022-12-08T21:37:11.523219Z	0.508135
2022-12-10T14:34:52.706027Z	0.371712	2022-12-08T21:37:11.523220Z	0.371712
2022-12-10T14:34:52.706028Z	0.152588	2022-12-08T21:37:11.523221Z	0.152588
2022-12-10T14:34:52.706029Z	0.504658	2022-12-08T21:37:11.523222Z	0.504658
2022-12-10T14:34:52.706030Z	0.562557	2022-12-08T21:37:11.523223Z	0.562557
2022-12-10T14:34:52.706031Z	0.741220	2022-12-08T21:37:11.523224Z	0.741220
2022-12-10T14:34:52.706032Z	0.233598	2022-12-08T21:37:11.523225Z	0.233598

```
2022-12-10T14:34:52.706033Z    0.681272   2022-12-08T21:37:11.523226Z    0.681272
2022-12-10T14:34:52.706034Z    0.284977   2022-12-08T21:37:11.523227Z    0.284977
2022-12-10T14:34:52.706035Z    0.0726497  2022-12-08T21:37:11.523228Z    0.0726497
2022-12-10T14:34:52.706036Z    0.795870   2022-12-08T21:37:11.523229Z    0.795870
2022-12-10T14:34:52.706037Z    0.866724   2022-12-08T21:37:11.523230Z    0.866724
2022-12-10T14:34:52.706038Z    0.365148   2022-12-08T21:37:11.523231Z    0.365148
2022-12-10T14:34:52.706039Z    0.238647   2022-12-08T21:37:11.523232Z    0.238647
2022-12-10T14:34:52.706040Z    0.381629   2022-12-08T21:37:11.523233Z    0.381629

TEST: testRequestDataAsync1_IntegrityScalars FROM
com.ospreydcs.datastore.admin.model.IQueryServiceDataTest
  Query time (seconds)    : 0.019939000000000002
  Request size (bytes)    : 1120
  Data rate (bytes/second): 56171.32253372786
Test Data Frame:
Snapshot Data Provider UID = null
DataFrame UID = null
DataFrame Timestamp = 2022-10-01T01:23:40.100Z
DataFrame Attributes = {duration=10000000000, period=1000000000, file=test-dataframe-scalars.yml,
name=Test DataFrame Scalars, type=test data, frequency=1}
timestamp      TEST-PV00      TEST-PV01      TEST-PV02      TEST-PV03      TEST-PV04
2022-10-01T01:23:45.100Z      true    0      0.0     str0    0.0
2022-10-01T01:23:46.100Z      true    1      0.1     str1    0.01
2022-10-01T01:23:47.100Z      true    2      0.2     str2    0.02
2022-10-01T01:23:48.100Z      true    3      0.3     str3    0.03
2022-10-01T01:23:49.100Z      true    4      0.4     str4    0.04
2022-10-01T01:23:50.100Z      true    5      0.5     str5    0.05
2022-10-01T01:23:51.100Z      true    6      0.6     str6    0.06
2022-10-01T01:23:52.100Z      true    7      0.7     str7    0.07
2022-10-01T01:23:53.100Z      true    8      0.8     str8    0.08
2022-10-01T01:23:54.100Z      false   9      0.9     str9    0.09

Open Query Result:
timestamp      TEST-PV00      TEST-PV02      TEST-PV01      TEST-PV04      TEST-PV03
2022-10-01T01:24:50.100Z      true    0.0    0       0.0     null
2022-10-01T01:24:51.100Z      true    0.1    1       0.01    null
2022-10-01T01:24:52.100Z      true    0.2    2       0.02    null
2022-10-01T01:24:53.100Z      true    0.3    3       0.03    null
2022-10-01T01:24:54.100Z      true    0.4    4       0.04    null
2022-10-01T01:24:55.100Z      true    0.5    5       0.05    null
2022-10-01T01:24:56.100Z      true    0.6    6       0.06    null
2022-10-01T01:24:57.100Z      true    0.7    7       0.07    null
2022-10-01T01:24:58.100Z      true    0.8    8       0.08    null
2022-10-01T01:24:59.100Z      false   0.9    9       0.09    null


Query results are missing providers: []
Timestamp maximum time difference: PT1M5S
Query results differ at the following locations:
  TEST-PV01: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
  TEST-PV03: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

TEST: testRequestDataAsync2_IntegrityArrays FROM
com.ospreydcs.datastore.admin.model.IQueryServiceDataTest
  Query time (seconds)    : 0.020027
  Request size (bytes)    : 1660
  Data rate (bytes/second): 82888.10106356419
Test Data Frame:
Snapshot Data Provider UID = null
DataFrame UID = null
DataFrame Timestamp = 2022-10-03T01:23:40.100Z
DataFrame Attributes = {duration=10000000000, period=1000000000, file=test-dataframe-arrays.yml,
name=Test DataFrame Arrays, type=test data, frequency=1}
timestamp      TEST-Array-PV00
2022-10-03T01:23:45.100Z      [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
2022-10-03T01:23:46.100Z      [1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9]
2022-10-03T01:23:47.100Z      [2.0, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9]
2022-10-03T01:23:48.100Z      [3.3, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9]
2022-10-03T01:23:49.100Z      [4.0, 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9]
2022-10-03T01:23:50.100Z      [5.0, 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9]
2022-10-03T01:23:51.100Z      [6.0, 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, 6.9]
2022-10-03T01:23:52.100Z      [7.0, 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.7, 7.8, 7.9]
2022-10-03T01:23:53.100Z      [8.0, 8.1, 8.2, 8.3, 8.4, 8.5, 8.6, 8.7, 8.8, 8.9]
2022-10-03T01:23:54.100Z      [9.0, 9.1, 9.2, 9.3, 9.4, 9.5, 9.6, 9.7, 9.8, 9.9]

Open Query Result:
timestamp      TEST-Array-PV00
2022-10-03T01:24:50.100Z      [0, null, null, null, null, null, null, null, null, null]
2022-10-03T01:24:51.100Z      [1, null, null, null, null, null, null, null, null, null]
2022-10-03T01:24:52.100Z      [2, null, null, null, null, null, null, null, null, null]
2022-10-03T01:24:53.100Z      [null, null, null, null, null, null, null, null, null, null]
```

```
2022-10-03T01:24:54.100Z      [4, null, null, null, null, null, null, null, null, null]
2022-10-03T01:24:55.100Z      [5, null, null, null, null, null, null, null, null, null]
2022-10-03T01:24:56.100Z      [6, null, null, null, null, null, null, null, null, null]
2022-10-03T01:24:57.100Z      [7, null, null, null, null, null, null, null, null, null]
2022-10-03T01:24:58.100Z      [8, null, null, null, null, null, null, null, null, null]
2022-10-03T01:24:59.100Z      [9, null, null, null, null, null, null, null, null, null]


Query results are missing providers: []
Timestamp maximum time difference: PT1M5S
Query results differ at the following locations:
  TEST-Array-PV00: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

TEST: testRequestDataAsync3_IntegrityStructs FROM
com.ospreydcs.datastore.admin.model.IQueryServiceDataTest
  Query time (seconds)    : 0.018214
  Request size (bytes)    : 240
  Data rate (bytes/second): 13176.677281212254
Test Data Frame:
Snapshot Data Provider UID = null
DataFrame UID = null
DataFrame Timestamp = 2022-10-04T01:23:40.100Z
DataFrame Attributes = {duration=10000000000, period=1000000000, file=test-dataframe-structs.yml,
name=Test DataFrame Structures, type=test data, frequency=1}
timestamp      TEST-Structure-PV00
2022-10-04T01:23:45.100Z      {f1=0.0}
2022-10-04T01:23:46.100Z      {g1=1.0, g2=2.0, g3=3.0, g4=4.0, g5=5.0}
2022-10-04T01:23:47.100Z      {h1=1.0, h2={h21=0.1, h22=0.2}, h3={h31=0.1, h32=0.2, h33=0.3}}
2022-10-04T01:23:48.100Z      {i1=1.0, i2={i21=0.1, i22=0.2}, i3={i31=0.1, i32={i321=0.01,
i322={i3221=0.001, i3222=0.002}}}}
2022-10-04T01:23:49.100Z      {j1={j11={j111={j1111={j11111=1.0E-5, j11112=2.0E-5, j11113=3.0E-
5}}}}}

Open Query Result:
timestamp      TEST-Structure-PV00
2022-10-04T01:24:15.100Z      0.0
2022-10-04T01:24:16.100Z      {g1=1.0, g2=2.0, g3=3.0, g4=4.0, g5=5.0}
2022-10-04T01:24:17.100Z      {h1=1.0, h2={h21=null, h22=null}, h3={h31=null, h32=null,
h33=null}}
2022-10-04T01:24:18.100Z      {i1=1.0, i2={i21=null, i22=null}, i3={i31=null, i32=null}}
2022-10-04T01:24:19.100Z      {j11=null}


Query results are missing providers: []
Timestamp maximum time difference: PT30S
Query results differ at the following locations:
  TEST-Structure-PV00: [0, 2, 3, 4]

TEST: testRequestDataAsync3_IntegrityStructs2 FROM
com.ospreydcs.datastore.admin.model.IQueryServiceDataTest
  Query time (seconds)    : 0.017487000000000003
  Request size (bytes)    : 4305
  Data rate (bytes/second): 246182.87870989874
Test Data Frame:
Snapshot Data Provider UID = null
DataFrame UID = null
DataFrame Timestamp = 2022-10-05T01:23:40.100Z
DataFrame Attributes = {duration=10000000000, period=1000000000, file=test-dataframe-
structs2.yml, name=Test DataFrame Structures 2, type=test data, frequency=1}
timestamp      TEST-Structure2-PV00  TEST-Structure2-PV01  TEST-Structure2-PV02  TEST-
Structure2-PV03       TEST-Structure2-PV04
2022-10-05T01:23:45.100Z      {f1=0.0}       {g1=1.0, g2=2.0, g3=3.0, g4=4.0, g5=5.0}
       {h1=1.0, h2={h21=1.1, h22=1.2}, h3={h31=1.01, h32=1.02, h33=1.03}} {i1=1.0, i2={i21=1.1,
i22=1.2}, i3={i31=0.1, i32={i321=0.01, i322={i3221=0.001, i3222=1.002}}}}
       {j1={j11={j111={j1111={j11111=1.00001, j11112=1.00002, j11113=1.00003}}}}}
2022-10-05T01:23:46.100Z      {f1=1.0}       {g1=1.1, g2=2.1, g3=3.1, g4=4.1, g5=5.1}
       {h1=2.0, h2={h21=2.1, h22=2.2}, h3={h31=2.01, h32=2.02, h33=2.03}} {i1=2.0, i2={i21=2.1,
i22=2.2}, i3={i31=0.2, i32={i321=0.02, i322={i3221=0.002, i3222=2.002}}}}
       {j1={j11={j111={j1111={j11111=2.00001, j11112=2.00002, j11113=2.00003}}}}}
2022-10-05T01:23:47.100Z      {f1=2.0}       {g1=1.2, g2=2.2, g3=3.2, g4=4.2, g5=5.2}
       {h1=3.0, h2={h21=3.1, h22=3.2}, h3={h31=3.01, h32=3.02, h33=3.03}} {i1=3.0, i2={i21=3.1,
i22=3.2}, i3={i31=0.3, i32={i321=0.03, i322={i3221=0.003, i3222=3.002}}}}
       {j1={j11={j111={j1111={j11111=3.00001, j11112=3.00002, j11113=3.00003}}}}}
2022-10-05T01:23:48.100Z      {f1=3.0}       {g1=1.3, g2=2.3, g3=3.3, g4=4.3, g5=5.3}
       {h1=4.0, h2={h21=4.1, h22=4.2}, h3={h31=4.01, h32=4.02, h33=4.03}} {i1=4.0, i2={i21=4.1,
i22=4.2}, i3={i31=0.4, i32={i321=0.04, i322={i3221=0.004, i3222=4.002}}}}
       {j1={j11={j111={j1111={j11111=4.00001, j11112=4.00002, j11113=4.00003}}}}}
2022-10-05T01:23:49.100Z      {f1=4.0}       {g1=1.4, g2=2.4, g3=3.4, g4=4.4, g5=5.4}
       {h1=5.0, h2={h21=5.1, h22=5.2}, h3={h31=5.01, h32=5.02, h33=5.03}} {i1=5.0, i2={i21=5.1,
i22=5.2}, i3={i31=0.5, i32={i321=0.05, i322={i3221=0.005, i3222=5.002}}}}
       {j1={j11={j111={j1111={j11111=5.00001, j11112=5.00002, j11113=5.00003}}}}}
```

```
Open Query Result:
timestamp      TEST-Structure2-PV03  TEST-Structure2-PV04  TEST-Structure2-PV00  TEST-
Structure2-PV01       TEST-Structure2-PV02
2022-10-05T01:24:15.100Z      {i1=1.0, i2={i21=null, i22=null}, i3={i31=null, i32=null}}
       {j11=null}     0.0     {g1=1.0, g2=2.0, g3=3.0, g4=4.0, g5=5.0}     {h1=1.0, h2={h21=null,
h22=null}, h3={h31=null, h32=null, h33=null}}
2022-10-05T01:24:16.100Z      {i1=2.0, i2={i21=null, i22=null}, i3={i31=null, i32=null}}
       {j11=null}     1.0     {g1=1.1, g2=2.1, g3=3.1, g4=4.1, g5=5.1}     {h1=2.0, h2={h21=null,
h22=null}, h3={h31=null, h32=null, h33=null}}
2022-10-05T01:24:17.100Z      {i1=3.0, i2={i21=null, i22=null}, i3={i31=null, i32=null}}
       {j11=null}     2.0     {g1=1.2, g2=2.2, g3=3.2, g4=4.2, g5=5.2}     {h1=3.0, h2={h21=null,
h22=null}, h3={h31=null, h32=null, h33=null}}
2022-10-05T01:24:18.100Z      {i1=4.0, i2={i21=null, i22=null}, i3={i31=null, i32=null}}
       {j11=null}     3.0     {g1=1.3, g2=2.3, g3=3.3, g4=4.3, g5=5.3}     {h1=4.0, h2={h21=null,
h22=null}, h3={h31=null, h32=null, h33=null}}
2022-10-05T01:24:19.100Z      {i1=5.0, i2={i21=null, i22=null}, i3={i31=null, i32=null}}
       {j11=null}     4.0     {g1=1.4, g2=2.4, g3=3.4, g4=4.4, g5=5.4}     {h1=5.0, h2={h21=null,
h22=null}, h3={h31=null, h32=null, h33=null}}


Query results are missing providers: []
Timestamp maximum time difference: PT30S
Query results differ at the following locations:
  TEST-Structure2-PV03: [0, 1, 2, 3, 4]
  TEST-Structure2-PV04: [0, 1, 2, 3, 4]
  TEST-Structure2-PV00: [0, 1, 2, 3, 4]
  TEST-Structure2-PV02: [0, 1, 2, 3, 4]

TEST: testRequestDataAsync4_ScalarTable500 FROM
com.ospreydcs.datastore.admin.model.IQueryServiceDataTest
Single Data Frame: 500 PV wide frame, double values
Number of frames sent to Datastore : 1
Test data frame column count       : 500
Test data frame row count          : 100
Test data frame value count        : 50000
Test data frame allocation (bytes) : 1202400
Total number of rows in Datastore  : 100
Total number of values in Datastore: 50000
Total data in Datastore (bytes)    : 1202400
Result table column count      : 500
Result table row count         : 100
Result table value count total : 50000
Result table null value count  : 0
Result table allocation (bytes): 1202400
Query duration                 : PT0.595764S
Query time (seconds)           : 0.5957640000000001
Query data rate (bytes/second) : 2018248.8367877211

Query results are missing providers: []
Timestamp maximum time difference: PT-41H-1M-4.094004S
Query results differ at the following locations:

Callback Function Test: 500 wide frame, double values
Open query request at 2022-12-10T14:38:13.376798Z
Table loading wait completed at 2022-12-10T14:38:13.994985Z
Total loading wait duration : PT0.618187S
Result table allocation (bytes): 1202400
Query data rate (bytes/second) : 1945042.5194965277
FncNotify(Boolean, IDataTableDynamic) called at 2022-12-10T14:38:13.994973Z
  bolSuccess = true
  (tblResult==dtdResult) = true
  Callback duration : PT0.618175S
  Result table column count      : 500
  Result table row count         : 100
  Result table value count total : 50000
  Result table null value count  : 0
  Result table allocation (bytes): 1202400
  Query duration                 : PT0.618175S
  Query time (seconds)           : 0.618175
  Query data rate (bytes/second) : 1945080.276620698

Single PV Select Query:
Single query request        : SELECT `Test-Table500-DOUBLE499.value`  WHERE time >= '1970-01-
01T00:00:00Z'
Total loading wait duration : PT0.453802S
Total wait time (seconds)   : 0.45380200000000004
Result allocation (bytes)   : 4800
Data rate (bytes/second)    : 10577.300232259884

PV Timestamps                  PV Values  Query Timestamps               Query Values
```

2022-12-10T14:37:09.613311Z	0.762613	2022-12-08T21:36:05.519307Z	0.762613
2022-12-10T14:37:09.613312Z	0.790143	2022-12-08T21:36:05.519308Z	0.790143
2022-12-10T14:37:09.613313Z	0.116374	2022-12-08T21:36:05.519309Z	0.116374
2022-12-10T14:37:09.613314Z	0.325579	2022-12-08T21:36:05.519310Z	0.325579
2022-12-10T14:37:09.613315Z	0.222622	2022-12-08T21:36:05.519311Z	0.222622
2022-12-10T14:37:09.613316Z	0.668450	2022-12-08T21:36:05.519312Z	0.668450
2022-12-10T14:37:09.613317Z	0.560950	2022-12-08T21:36:05.519313Z	0.560950
2022-12-10T14:37:09.613318Z	0.661735	2022-12-08T21:36:05.519314Z	0.661735
2022-12-10T14:37:09.613319Z	0.156326	2022-12-08T21:36:05.519315Z	0.156326
2022-12-10T14:37:09.613320Z	0.151829	2022-12-08T21:36:05.519316Z	0.151829
2022-12-10T14:37:09.613321Z	0.420613	2022-12-08T21:36:05.519317Z	0.420613
2022-12-10T14:37:09.613322Z	0.565620	2022-12-08T21:36:05.519318Z	0.565620
2022-12-10T14:37:09.613323Z	0.477866	2022-12-08T21:36:05.519319Z	0.477866
2022-12-10T14:37:09.613324Z	0.103225	2022-12-08T21:36:05.519320Z	0.103225
2022-12-10T14:37:09.613325Z	0.301309	2022-12-08T21:36:05.519321Z	0.301309
2022-12-10T14:37:09.613326Z	0.895247	2022-12-08T21:36:05.519322Z	0.895247
2022-12-10T14:37:09.613327Z	0.909542	2022-12-08T21:36:05.519323Z	0.909542
2022-12-10T14:37:09.613328Z	0.343432	2022-12-08T21:36:05.519324Z	0.343432
2022-12-10T14:37:09.613329Z	0.416665	2022-12-08T21:36:05.519325Z	0.416665
2022-12-10T14:37:09.613330Z	0.0276130	2022-12-08T21:36:05.519326Z	0.0276130
2022-12-10T14:37:09.613331Z	0.805125	2022-12-08T21:36:05.519327Z	0.805125
2022-12-10T14:37:09.613332Z	0.958280	2022-12-08T21:36:05.519328Z	0.958280
2022-12-10T14:37:09.613333Z	0.931042	2022-12-08T21:36:05.519329Z	0.931042
2022-12-10T14:37:09.613334Z	0.287248	2022-12-08T21:36:05.519330Z	0.287248
2022-12-10T14:37:09.613335Z	0.910493	2022-12-08T21:36:05.519331Z	0.910493
2022-12-10T14:37:09.613336Z	0.578218	2022-12-08T21:36:05.519332Z	0.578218
2022-12-10T14:37:09.613337Z	0.742736	2022-12-08T21:36:05.519333Z	0.742736
2022-12-10T14:37:09.613338Z	0.370117	2022-12-08T21:36:05.519334Z	0.370117
2022-12-10T14:37:09.613339Z	0.764743	2022-12-08T21:36:05.519335Z	0.764743
2022-12-10T14:37:09.613340Z	0.539757	2022-12-08T21:36:05.519336Z	0.539757
2022-12-10T14:37:09.613341Z	0.403167	2022-12-08T21:36:05.519337Z	0.403167
2022-12-10T14:37:09.613342Z	0.153306	2022-12-08T21:36:05.519338Z	0.153306
2022-12-10T14:37:09.613343Z	0.188111	2022-12-08T21:36:05.519339Z	0.188111
2022-12-10T14:37:09.613344Z	0.370435	2022-12-08T21:36:05.519340Z	0.370435
2022-12-10T14:37:09.613345Z	0.251656	2022-12-08T21:36:05.519341Z	0.251656
2022-12-10T14:37:09.613346Z	0.0787347	2022-12-08T21:36:05.519342Z	0.0787347
2022-12-10T14:37:09.613347Z	0.577420	2022-12-08T21:36:05.519343Z	0.577420
2022-12-10T14:37:09.613348Z	0.352484	2022-12-08T21:36:05.519344Z	0.352484
2022-12-10T14:37:09.613349Z	0.260471	2022-12-08T21:36:05.519345Z	0.260471
2022-12-10T14:37:09.613350Z	0.378281	2022-12-08T21:36:05.519346Z	0.378281
2022-12-10T14:37:09.613351Z	0.535466	2022-12-08T21:36:05.519347Z	0.535466
2022-12-10T14:37:09.613352Z	0.0150599	2022-12-08T21:36:05.519348Z	0.0150599
2022-12-10T14:37:09.613353Z	0.531074	2022-12-08T21:36:05.519349Z	0.531074
2022-12-10T14:37:09.613354Z	0.768013	2022-12-08T21:36:05.519350Z	0.768013
2022-12-10T14:37:09.613355Z	0.697641	2022-12-08T21:36:05.519351Z	0.697641
2022-12-10T14:37:09.613356Z	0.280920	2022-12-08T21:36:05.519352Z	0.280920
2022-12-10T14:37:09.613357Z	0.860069	2022-12-08T21:36:05.519353Z	0.860069
2022-12-10T14:37:09.613358Z	0.880862	2022-12-08T21:36:05.519354Z	0.880862
2022-12-10T14:37:09.613359Z	0.487281	2022-12-08T21:36:05.519355Z	0.487281
2022-12-10T14:37:09.613360Z	0.650612	2022-12-08T21:36:05.519356Z	0.650612
2022-12-10T14:37:09.613361Z	0.302121	2022-12-08T21:36:05.519357Z	0.302121
2022-12-10T14:37:09.613362Z	0.629895	2022-12-08T21:36:05.519358Z	0.629895
2022-12-10T14:37:09.613363Z	0.930620	2022-12-08T21:36:05.519359Z	0.930620
2022-12-10T14:37:09.613364Z	0.245753	2022-12-08T21:36:05.519360Z	0.245753
2022-12-10T14:37:09.613365Z	0.197155	2022-12-08T21:36:05.519361Z	0.197155
2022-12-10T14:37:09.613366Z	0.898468	2022-12-08T21:36:05.519362Z	0.898468
2022-12-10T14:37:09.613367Z	0.592910	2022-12-08T21:36:05.519363Z	0.592910
2022-12-10T14:37:09.613368Z	0.752690	2022-12-08T21:36:05.519364Z	0.752690
2022-12-10T14:37:09.613369Z	0.700464	2022-12-08T21:36:05.519365Z	0.700464
2022-12-10T14:37:09.613370Z	0.438868	2022-12-08T21:36:05.519366Z	0.438868
2022-12-10T14:37:09.613371Z	0.159169	2022-12-08T21:36:05.519367Z	0.159169
2022-12-10T14:37:09.613372Z	0.308794	2022-12-08T21:36:05.519368Z	0.308794
2022-12-10T14:37:09.613373Z	0.456834	2022-12-08T21:36:05.519369Z	0.456834
2022-12-10T14:37:09.613374Z	0.627686	2022-12-08T21:36:05.519370Z	0.627686
2022-12-10T14:37:09.613375Z	0.176198	2022-12-08T21:36:05.519371Z	0.176198
2022-12-10T14:37:09.613376Z	0.155897	2022-12-08T21:36:05.519372Z	0.155897
2022-12-10T14:37:09.613377Z	0.693652	2022-12-08T21:36:05.519373Z	0.693652
2022-12-10T14:37:09.613378Z	0.773975	2022-12-08T21:36:05.519374Z	0.773975
2022-12-10T14:37:09.613379Z	0.996897	2022-12-08T21:36:05.519375Z	0.996897
2022-12-10T14:37:09.613380Z	0.448835	2022-12-08T21:36:05.519376Z	0.448835
2022-12-10T14:37:09.613381Z	0.672951	2022-12-08T21:36:05.519377Z	0.672951
2022-12-10T14:37:09.613382Z	0.00539157	2022-12-08T21:36:05.519378Z	0.00539157
2022-12-10T14:37:09.613383Z	0.0236185	2022-12-08T21:36:05.519379Z	0.0236185
2022-12-10T14:37:09.613384Z	0.951792	2022-12-08T21:36:05.519380Z	0.951792
2022-12-10T14:37:09.613385Z	0.138654	2022-12-08T21:36:05.519381Z	0.138654
2022-12-10T14:37:09.613386Z	0.367834	2022-12-08T21:36:05.519382Z	0.367834
2022-12-10T14:37:09.613387Z	0.934145	2022-12-08T21:36:05.519383Z	0.934145
2022-12-10T14:37:09.613388Z	0.481122	2022-12-08T21:36:05.519384Z	0.481122
2022-12-10T14:37:09.613389Z	0.122825	2022-12-08T21:36:05.519385Z	0.122825
2022-12-10T14:37:09.613390Z	0.665713	2022-12-08T21:36:05.519386Z	0.665713
2022-12-10T14:37:09.613391Z	0.692112	2022-12-08T21:36:05.519387Z	0.692112

```
2022-12-10T14:37:09.613392Z    0.860546   2022-12-08T21:36:05.519388Z    0.860546
2022-12-10T14:37:09.613393Z    0.233509   2022-12-08T21:36:05.519389Z    0.233509
2022-12-10T14:37:09.613394Z    0.742835   2022-12-08T21:36:05.519390Z    0.742835
2022-12-10T14:37:09.613395Z    0.793065   2022-12-08T21:36:05.519391Z    0.793065
2022-12-10T14:37:09.613396Z    0.843101   2022-12-08T21:36:05.519392Z    0.843101
2022-12-10T14:37:09.613397Z    0.336787   2022-12-08T21:36:05.519393Z    0.336787
2022-12-10T14:37:09.613398Z    0.357991   2022-12-08T21:36:05.519394Z    0.357991
2022-12-10T14:37:09.613399Z    0.378008   2022-12-08T21:36:05.519395Z    0.378008
2022-12-10T14:37:09.613400Z    0.923943   2022-12-08T21:36:05.519396Z    0.923943
2022-12-10T14:37:09.613401Z    0.366978   2022-12-08T21:36:05.519397Z    0.366978
2022-12-10T14:37:09.613402Z    0.544659   2022-12-08T21:36:05.519398Z    0.544659
2022-12-10T14:37:09.613403Z    0.558250   2022-12-08T21:36:05.519399Z    0.558250
2022-12-10T14:37:09.613404Z    0.219087   2022-12-08T21:36:05.519400Z    0.219087
2022-12-10T14:37:09.613405Z    0.851424   2022-12-08T21:36:05.519401Z    0.851424
2022-12-10T14:37:09.613406Z    0.360898   2022-12-08T21:36:05.519402Z    0.360898
2022-12-10T14:37:09.613407Z    0.538459   2022-12-08T21:36:05.519403Z    0.538459
2022-12-10T14:37:09.613408Z    0.617386   2022-12-08T21:36:05.519404Z    0.617386
2022-12-10T14:37:09.613409Z    0.620119   2022-12-08T21:36:05.519405Z    0.620119
2022-12-10T14:37:09.613410Z    0.233278   2022-12-08T21:36:05.519406Z    0.233278

TEST: testRequestDataAsync5_ScalarTable1k FROM
com.ospreydcs.datastore.admin.model.IQueryServiceDataTest
Single Data Frame: 1,000 PV wide frame, double values
Number of frames sent to Datastore : 1
Test data frame column count       : 1000
Test data frame row count          : 100
Test data frame value count        : 100000
Test data frame allocation (bytes) : 2402400
Total number of rows in Datastore  : 100
Total number of values in Datastore: 100000
Total data in Datastore (bytes)    : 2402400
Result table column count      : 1000
Result table row count         : 100
Result table value count total : 100000
Result table null value count  : 0
Result table allocation (bytes): 2402400
Query duration                 : PT1.116363S
Query time (seconds)           : 1.116363
Query data rate (bytes/second) : 2151988.197387409

Query results are missing providers: []
Timestamp maximum time difference: PT-41H-1M-5.606428S
Query results differ at the following locations:

Callback Function Test: 1,000 PV wide frame, double values
Open query request at 2022-12-10T14:40:26.523828Z
Table loading wait completed at 2022-12-10T14:40:27.620019Z
Total loading wait duration : PT1.096191S
Result table allocation (bytes): 2402400
Query data rate (bytes/second) : 2191588.8745665676
FncNotify(Boolean, IDataTableDynamic) called at 2022-12-10T14:40:27.620008Z
  bolSuccess = true
  (tblResult==dtdResult) = true
  Callback duration : PT1.09618S
  Result table column count      : 1000
  Result table row count         : 100
  Result table value count total : 100000
  Result table null value count  : 0
  Result table allocation (bytes): 2402400
  Query duration                 : PT1.09618S
  Query time (seconds)           : 1.09618
  Query data rate (bytes/second) : 2191610.8668284407

Single PV Select Query:
Single query request        : SELECT `Test-Table1K-DOUBLE999.value`  WHERE time >= '1970-01-
01T00:00:00Z'
Total loading wait duration : PT1.133155S
Total wait time (seconds)   : 1.133155
Result allocation (bytes)   : 4800
Data rate (bytes/second)    : 4235.960658515384

PV Timestamps                  PV Values  Query Timestamps               Query Values
2022-12-10T14:38:17.129662Z    0.660633   2022-12-08T21:37:11.523234Z    0.660633
2022-12-10T14:38:17.129663Z    0.449879   2022-12-08T21:37:11.523235Z    0.449879
2022-12-10T14:38:17.129664Z    0.0934336  2022-12-08T21:37:11.523236Z    0.0934336
2022-12-10T14:38:17.129665Z    0.559412   2022-12-08T21:37:11.523237Z    0.559412
2022-12-10T14:38:17.129666Z    0.248453   2022-12-08T21:37:11.523238Z    0.248453
2022-12-10T14:38:17.129667Z    0.826494   2022-12-08T21:37:11.523239Z    0.826494
2022-12-10T14:38:17.129668Z    0.00340653 2022-12-08T21:37:11.523240Z    0.00340653
2022-12-10T14:38:17.129669Z    0.369451   2022-12-08T21:37:11.523241Z    0.369451
2022-12-10T14:38:17.129670Z    0.508015   2022-12-08T21:37:11.523242Z    0.508015
```

2022-12-10T14:38:17.129671Z	0.749118	2022-12-08T21:37:11.523243Z	0.749118
2022-12-10T14:38:17.129672Z	0.208295	2022-12-08T21:37:11.523244Z	0.208295
2022-12-10T14:38:17.129673Z	0.130552	2022-12-08T21:37:11.523245Z	0.130552
2022-12-10T14:38:17.129674Z	0.706394	2022-12-08T21:37:11.523246Z	0.706394
2022-12-10T14:38:17.129675Z	0.513258	2022-12-08T21:37:11.523247Z	0.513258
2022-12-10T14:38:17.129676Z	0.329093	2022-12-08T21:37:11.523248Z	0.329093
2022-12-10T14:38:17.129677Z	0.408907	2022-12-08T21:37:11.523249Z	0.408907
2022-12-10T14:38:17.129678Z	0.843263	2022-12-08T21:37:11.523250Z	0.843263
2022-12-10T14:38:17.129679Z	0.493823	2022-12-08T21:37:11.523251Z	0.493823
2022-12-10T14:38:17.129680Z	0.0273487	2022-12-08T21:37:11.523252Z	0.0273487
2022-12-10T14:38:17.129681Z	0.408973	2022-12-08T21:37:11.523253Z	0.408973
2022-12-10T14:38:17.129682Z	0.933771	2022-12-08T21:37:11.523254Z	0.933771
2022-12-10T14:38:17.129683Z	0.211255	2022-12-08T21:37:11.523255Z	0.211255
2022-12-10T14:38:17.129684Z	0.948057	2022-12-08T21:37:11.523256Z	0.948057
2022-12-10T14:38:17.129685Z	0.950639	2022-12-08T21:37:11.523257Z	0.950639
2022-12-10T14:38:17.129686Z	0.847162	2022-12-08T21:37:11.523258Z	0.847162
2022-12-10T14:38:17.129687Z	0.427929	2022-12-08T21:37:11.523259Z	0.427929
2022-12-10T14:38:17.129688Z	0.303126	2022-12-08T21:37:11.523260Z	0.303126
2022-12-10T14:38:17.129689Z	0.612902	2022-12-08T21:37:11.523261Z	0.612902
2022-12-10T14:38:17.129690Z	0.880840	2022-12-08T21:37:11.523262Z	0.880840
2022-12-10T14:38:17.129691Z	0.223326	2022-12-08T21:37:11.523263Z	0.223326
2022-12-10T14:38:17.129692Z	0.892837	2022-12-08T21:37:11.523264Z	0.892837
2022-12-10T14:38:17.129693Z	0.0985128	2022-12-08T21:37:11.523265Z	0.0985128
2022-12-10T14:38:17.129694Z	0.242767	2022-12-08T21:37:11.523266Z	0.242767
2022-12-10T14:38:17.129695Z	0.778720	2022-12-08T21:37:11.523267Z	0.778720
2022-12-10T14:38:17.129696Z	0.996459	2022-12-08T21:37:11.523268Z	0.996459
2022-12-10T14:38:17.129697Z	0.921936	2022-12-08T21:37:11.523269Z	0.921936
2022-12-10T14:38:17.129698Z	0.139498	2022-12-08T21:37:11.523270Z	0.139498
2022-12-10T14:38:17.129699Z	0.165613	2022-12-08T21:37:11.523271Z	0.165613
2022-12-10T14:38:17.129700Z	0.228259	2022-12-08T21:37:11.523272Z	0.228259
2022-12-10T14:38:17.129701Z	0.852033	2022-12-08T21:37:11.523273Z	0.852033
2022-12-10T14:38:17.129702Z	0.222797	2022-12-08T21:37:11.523274Z	0.222797
2022-12-10T14:38:17.129703Z	0.776849	2022-12-08T21:37:11.523275Z	0.776849
2022-12-10T14:38:17.129704Z	0.151402	2022-12-08T21:37:11.523276Z	0.151402
2022-12-10T14:38:17.129705Z	0.151260	2022-12-08T21:37:11.523277Z	0.151260
2022-12-10T14:38:17.129706Z	0.734725	2022-12-08T21:37:11.523278Z	0.734725
2022-12-10T14:38:17.129707Z	0.165344	2022-12-08T21:37:11.523279Z	0.165344
2022-12-10T14:38:17.129708Z	0.169804	2022-12-08T21:37:11.523280Z	0.169804
2022-12-10T14:38:17.129709Z	0.544751	2022-12-08T21:37:11.523281Z	0.544751
2022-12-10T14:38:17.129710Z	0.809225	2022-12-08T21:37:11.523282Z	0.809225
2022-12-10T14:38:17.129711Z	0.962566	2022-12-08T21:37:11.523283Z	0.962566
2022-12-10T14:38:17.129712Z	0.976764	2022-12-08T21:37:11.523284Z	0.976764
2022-12-10T14:38:17.129713Z	0.0255319	2022-12-08T21:37:11.523285Z	0.0255319
2022-12-10T14:38:17.129714Z	0.0376698	2022-12-08T21:37:11.523286Z	0.0376698
2022-12-10T14:38:17.129715Z	0.237407	2022-12-08T21:37:11.523287Z	0.237407
2022-12-10T14:38:17.129716Z	0.629274	2022-12-08T21:37:11.523288Z	0.629274
2022-12-10T14:38:17.129717Z	0.0561740	2022-12-08T21:37:11.523289Z	0.0561740
2022-12-10T14:38:17.129718Z	0.586469	2022-12-08T21:37:11.523290Z	0.586469
2022-12-10T14:38:17.129719Z	0.125789	2022-12-08T21:37:11.523291Z	0.125789
2022-12-10T14:38:17.129720Z	0.138144	2022-12-08T21:37:11.523292Z	0.138144
2022-12-10T14:38:17.129721Z	0.462603	2022-12-08T21:37:11.523293Z	0.462603
2022-12-10T14:38:17.129722Z	0.866790	2022-12-08T21:37:11.523294Z	0.866790
2022-12-10T14:38:17.129723Z	0.812338	2022-12-08T21:37:11.523295Z	0.812338
2022-12-10T14:38:17.129724Z	0.316681	2022-12-08T21:37:11.523296Z	0.316681
2022-12-10T14:38:17.129725Z	0.332323	2022-12-08T21:37:11.523297Z	0.332323
2022-12-10T14:38:17.129726Z	0.999572	2022-12-08T21:37:11.523298Z	0.999572
2022-12-10T14:38:17.129727Z	0.977680	2022-12-08T21:37:11.523299Z	0.977680
2022-12-10T14:38:17.129728Z	0.514389	2022-12-08T21:37:11.523300Z	0.514389
2022-12-10T14:38:17.129729Z	0.527212	2022-12-08T21:37:11.523301Z	0.527212
2022-12-10T14:38:17.129730Z	0.0934409	2022-12-08T21:37:11.523302Z	0.0934409
2022-12-10T14:38:17.129731Z	0.638397	2022-12-08T21:37:11.523303Z	0.638397
2022-12-10T14:38:17.129732Z	0.777247	2022-12-08T21:37:11.523304Z	0.777247
2022-12-10T14:38:17.129733Z	0.000112316	2022-12-08T21:37:11.523305Z	0.000112316
2022-12-10T14:38:17.129734Z	0.658807	2022-12-08T21:37:11.523306Z	0.658807
2022-12-10T14:38:17.129735Z	0.901388	2022-12-08T21:37:11.523307Z	0.901388
2022-12-10T14:38:17.129736Z	0.922370	2022-12-08T21:37:11.523308Z	0.922370
2022-12-10T14:38:17.129737Z	0.964825	2022-12-08T21:37:11.523309Z	0.964825
2022-12-10T14:38:17.129738Z	0.318328	2022-12-08T21:37:11.523310Z	0.318328
2022-12-10T14:38:17.129739Z	0.530392	2022-12-08T21:37:11.523311Z	0.530392
2022-12-10T14:38:17.129740Z	0.992433	2022-12-08T21:37:11.523312Z	0.992433
2022-12-10T14:38:17.129741Z	0.110684	2022-12-08T21:37:11.523313Z	0.110684
2022-12-10T14:38:17.129742Z	0.342084	2022-12-08T21:37:11.523314Z	0.342084
2022-12-10T14:38:17.129743Z	0.379609	2022-12-08T21:37:11.523315Z	0.379609
2022-12-10T14:38:17.129744Z	0.563991	2022-12-08T21:37:11.523316Z	0.563991
2022-12-10T14:38:17.129745Z	0.475766	2022-12-08T21:37:11.523317Z	0.475766
2022-12-10T14:38:17.129746Z	0.435233	2022-12-08T21:37:11.523318Z	0.435233
2022-12-10T14:38:17.129747Z	0.687890	2022-12-08T21:37:11.523319Z	0.687890
2022-12-10T14:38:17.129748Z	0.423903	2022-12-08T21:37:11.523320Z	0.423903
2022-12-10T14:38:17.129749Z	0.296875	2022-12-08T21:37:11.523321Z	0.296875
2022-12-10T14:38:17.129750Z	0.636583	2022-12-08T21:37:11.523322Z	0.636583
2022-12-10T14:38:17.129751Z	0.946621	2022-12-08T21:37:11.523323Z	0.946621

```
2022-12-10T14:38:17.129752Z    0.199330   2022-12-08T21:37:11.523324Z    0.199330
2022-12-10T14:38:17.129753Z    0.139803   2022-12-08T21:37:11.523325Z    0.139803
2022-12-10T14:38:17.129754Z    0.148406   2022-12-08T21:37:11.523326Z    0.148406
2022-12-10T14:38:17.129755Z    0.555487   2022-12-08T21:37:11.523327Z    0.555487
2022-12-10T14:38:17.129756Z    0.629324   2022-12-08T21:37:11.523328Z    0.629324
2022-12-10T14:38:17.129757Z    0.879961   2022-12-08T21:37:11.523329Z    0.879961
2022-12-10T14:38:17.129758Z    0.411018   2022-12-08T21:37:11.523330Z    0.411018
2022-12-10T14:38:17.129759Z    0.762447   2022-12-08T21:37:11.523331Z    0.762447
2022-12-10T14:38:17.129760Z    0.299826   2022-12-08T21:37:11.523332Z    0.299826
2022-12-10T14:38:17.129761Z    0.651340   2022-12-08T21:37:11.523333Z    0.651340

TEST: testRequestDataAsync6_ScalarTable2k FROM
com.ospreydcs.datastore.admin.model.IQueryServiceDataTest
Single Data Frame: 2,000 PV wide frame, double values
Number of frames sent to Datastore : 1
Test data frame column count       : 2000
Test data frame row count          : 100
Test data frame value count        : 200000
Test data frame allocation (bytes) : 4802400
Total number of rows in Datastore  : 100
Total number of values in Datastore: 200000
Total data in Datastore (bytes)    : 4802400
Result table column count      : 2000
Result table row count         : 100
Result table value count total : 200000
Result table null value count  : 0
Result table allocation (bytes): 4802400
Query duration                 : PT2.334258S
Query time (seconds)           : 2.334258
Query data rate (bytes/second) : 2057356.1277288112

Query results are missing providers: []
Timestamp maximum time difference: PT-40H-57M-32.332526S
Query results differ at the following locations:

Callback Function Test: 2,000 PV wide frame, double values
Open query request at 2022-12-10T14:45:43.887866Z
Table loading wait completed at 2022-12-10T14:45:46.450922Z
Total loading wait duration : PT2.563056S
Result table allocation (bytes): 4802400
Query data rate (bytes/second) : 1873700.7697061633
FncNotify(Boolean, IDataTableDynamic) called at 2022-12-10T14:45:46.450914Z
  bolSuccess = true
  (tblResult==dtdResult) = true
  Callback duration : PT2.563048S
  Result table column count      : 2000
  Result table row count         : 100
  Result table value count total : 200000
  Result table null value count  : 0
  Result table allocation (bytes): 4802400
  Query duration                 : PT2.563048S
  Query time (seconds)           : 2.563048
  Query data rate (bytes/second) : 1873706.6180578747

Single PV Select Query:
Single query request        : SELECT `Test-Table2K-DOUBLE1999.value`  WHERE time >= '1970-01-
01T00:00:00Z'
Total loading wait duration : PT1.713104S
Total wait time (seconds)   : 1.713104
Result allocation (bytes)   : 4800
Data rate (bytes/second)    : 2801.931464756372

PV Timestamps                  PV Values  Query Timestamps               Query Values
2022-12-10T14:40:34.554405Z    0.639960   2022-12-08T21:43:02.221879Z    0.639960
2022-12-10T14:40:34.554406Z    0.149034   2022-12-08T21:43:02.221880Z    0.149034
2022-12-10T14:40:34.554407Z    0.217559   2022-12-08T21:43:02.221881Z    0.217559
2022-12-10T14:40:34.554408Z    0.434104   2022-12-08T21:43:02.221882Z    0.434104
2022-12-10T14:40:34.554409Z    0.840019   2022-12-08T21:43:02.221883Z    0.840019
2022-12-10T14:40:34.554410Z    0.0308556  2022-12-08T21:43:02.221884Z    0.0308556
2022-12-10T14:40:34.554411Z    0.985007   2022-12-08T21:43:02.221885Z    0.985007
2022-12-10T14:40:34.554412Z    0.989694   2022-12-08T21:43:02.221886Z    0.989694
2022-12-10T14:40:34.554413Z    0.951906   2022-12-08T21:43:02.221887Z    0.951906
2022-12-10T14:40:34.554414Z    0.712961   2022-12-08T21:43:02.221888Z    0.712961
2022-12-10T14:40:34.554415Z    0.385616   2022-12-08T21:43:02.221889Z    0.385616
2022-12-10T14:40:34.554416Z    0.321724   2022-12-08T21:43:02.221890Z    0.321724
2022-12-10T14:40:34.554417Z    0.686109   2022-12-08T21:43:02.221891Z    0.686109
2022-12-10T14:40:34.554418Z    0.898196   2022-12-08T21:43:02.221892Z    0.898196
2022-12-10T14:40:34.554419Z    0.0642037  2022-12-08T21:43:02.221893Z    0.0642037
2022-12-10T14:40:34.554420Z    0.714807   2022-12-08T21:43:02.221894Z    0.714807
2022-12-10T14:40:34.554421Z    0.610156   2022-12-08T21:43:02.221895Z    0.610156
2022-12-10T14:40:34.554422Z    0.0158997  2022-12-08T21:43:02.221896Z    0.0158997
```

2022-12-10T14:40:34.554423Z	0.209783	2022-12-08T21:43:02.221897Z	0.209783
2022-12-10T14:40:34.554424Z	0.276372	2022-12-08T21:43:02.221898Z	0.276372
2022-12-10T14:40:34.554425Z	0.122124	2022-12-08T21:43:02.221899Z	0.122124
2022-12-10T14:40:34.554426Z	0.811797	2022-12-08T21:43:02.221900Z	0.811797
2022-12-10T14:40:34.554427Z	0.313338	2022-12-08T21:43:02.221901Z	0.313338
2022-12-10T14:40:34.554428Z	0.610678	2022-12-08T21:43:02.221902Z	0.610678
2022-12-10T14:40:34.554429Z	0.404207	2022-12-08T21:43:02.221903Z	0.404207
2022-12-10T14:40:34.554430Z	0.771378	2022-12-08T21:43:02.221904Z	0.771378
2022-12-10T14:40:34.554431Z	0.154572	2022-12-08T21:43:02.221905Z	0.154572
2022-12-10T14:40:34.554432Z	0.689127	2022-12-08T21:43:02.221906Z	0.689127
2022-12-10T14:40:34.554433Z	0.489157	2022-12-08T21:43:02.221907Z	0.489157
2022-12-10T14:40:34.554434Z	0.403396	2022-12-08T21:43:02.221908Z	0.403396
2022-12-10T14:40:34.554435Z	0.0181432	2022-12-08T21:43:02.221909Z	0.0181432
2022-12-10T14:40:34.554436Z	0.299467	2022-12-08T21:43:02.221910Z	0.299467
2022-12-10T14:40:34.554437Z	0.0656444	2022-12-08T21:43:02.221911Z	0.0656444
2022-12-10T14:40:34.554438Z	0.692360	2022-12-08T21:43:02.221912Z	0.692360
2022-12-10T14:40:34.554439Z	0.273690	2022-12-08T21:43:02.221913Z	0.273690
2022-12-10T14:40:34.554440Z	0.00432817	2022-12-08T21:43:02.221914Z	0.00432817
2022-12-10T14:40:34.554441Z	0.672931	2022-12-08T21:43:02.221915Z	0.672931
2022-12-10T14:40:34.554442Z	0.0923673	2022-12-08T21:43:02.221916Z	0.0923673
2022-12-10T14:40:34.554443Z	0.618996	2022-12-08T21:43:02.221917Z	0.618996
2022-12-10T14:40:34.554444Z	0.368655	2022-12-08T21:43:02.221918Z	0.368655
2022-12-10T14:40:34.554445Z	0.490947	2022-12-08T21:43:02.221919Z	0.490947
2022-12-10T14:40:34.554446Z	0.279339	2022-12-08T21:43:02.221920Z	0.279339
2022-12-10T14:40:34.554447Z	0.481938	2022-12-08T21:43:02.221921Z	0.481938
2022-12-10T14:40:34.554448Z	0.325930	2022-12-08T21:43:02.221922Z	0.325930
2022-12-10T14:40:34.554449Z	0.0187829	2022-12-08T21:43:02.221923Z	0.0187829
2022-12-10T14:40:34.554450Z	0.504714	2022-12-08T21:43:02.221924Z	0.504714
2022-12-10T14:40:34.554451Z	0.304375	2022-12-08T21:43:02.221925Z	0.304375
2022-12-10T14:40:34.554452Z	0.944701	2022-12-08T21:43:02.221926Z	0.944701
2022-12-10T14:40:34.554453Z	0.552739	2022-12-08T21:43:02.221927Z	0.552739
2022-12-10T14:40:34.554454Z	0.800877	2022-12-08T21:43:02.221928Z	0.800877
2022-12-10T14:40:34.554455Z	0.689963	2022-12-08T21:43:02.221929Z	0.689963
2022-12-10T14:40:34.554456Z	0.529764	2022-12-08T21:43:02.221930Z	0.529764
2022-12-10T14:40:34.554457Z	0.524532	2022-12-08T21:43:02.221931Z	0.524532
2022-12-10T14:40:34.554458Z	0.477066	2022-12-08T21:43:02.221932Z	0.477066
2022-12-10T14:40:34.554459Z	0.687881	2022-12-08T21:43:02.221933Z	0.687881
2022-12-10T14:40:34.554460Z	0.946458	2022-12-08T21:43:02.221934Z	0.946458
2022-12-10T14:40:34.554461Z	0.440882	2022-12-08T21:43:02.221935Z	0.440882
2022-12-10T14:40:34.554462Z	0.314242	2022-12-08T21:43:02.221936Z	0.314242
2022-12-10T14:40:34.554463Z	0.805770	2022-12-08T21:43:02.221937Z	0.805770
2022-12-10T14:40:34.554464Z	0.291887	2022-12-08T21:43:02.221938Z	0.291887
2022-12-10T14:40:34.554465Z	0.377650	2022-12-08T21:43:02.221939Z	0.377650
2022-12-10T14:40:34.554466Z	0.618372	2022-12-08T21:43:02.221940Z	0.618372
2022-12-10T14:40:34.554467Z	0.697717	2022-12-08T21:43:02.221941Z	0.697717
2022-12-10T14:40:34.554468Z	0.553483	2022-12-08T21:43:02.221942Z	0.553483
2022-12-10T14:40:34.554469Z	0.724828	2022-12-08T21:43:02.221943Z	0.724828
2022-12-10T14:40:34.554470Z	0.823807	2022-12-08T21:43:02.221944Z	0.823807
2022-12-10T14:40:34.554471Z	0.856897	2022-12-08T21:43:02.221945Z	0.856897
2022-12-10T14:40:34.554472Z	0.693157	2022-12-08T21:43:02.221946Z	0.693157
2022-12-10T14:40:34.554473Z	0.153141	2022-12-08T21:43:02.221947Z	0.153141
2022-12-10T14:40:34.554474Z	0.651914	2022-12-08T21:43:02.221948Z	0.651914
2022-12-10T14:40:34.554475Z	0.0575413	2022-12-08T21:43:02.221949Z	0.0575413
2022-12-10T14:40:34.554476Z	0.674422	2022-12-08T21:43:02.221950Z	0.674422
2022-12-10T14:40:34.554477Z	0.578626	2022-12-08T21:43:02.221951Z	0.578626
2022-12-10T14:40:34.554478Z	0.924849	2022-12-08T21:43:02.221952Z	0.924849
2022-12-10T14:40:34.554479Z	0.737171	2022-12-08T21:43:02.221953Z	0.737171
2022-12-10T14:40:34.554480Z	0.310196	2022-12-08T21:43:02.221954Z	0.310196
2022-12-10T14:40:34.554481Z	0.238530	2022-12-08T21:43:02.221955Z	0.238530
2022-12-10T14:40:34.554482Z	0.0103176	2022-12-08T21:43:02.221956Z	0.0103176
2022-12-10T14:40:34.554483Z	0.987593	2022-12-08T21:43:02.221957Z	0.987593
2022-12-10T14:40:34.554484Z	0.450596	2022-12-08T21:43:02.221958Z	0.450596
2022-12-10T14:40:34.554485Z	0.859988	2022-12-08T21:43:02.221959Z	0.859988
2022-12-10T14:40:34.554486Z	0.530433	2022-12-08T21:43:02.221960Z	0.530433
2022-12-10T14:40:34.554487Z	0.331931	2022-12-08T21:43:02.221961Z	0.331931
2022-12-10T14:40:34.554488Z	0.952907	2022-12-08T21:43:02.221962Z	0.952907
2022-12-10T14:40:34.554489Z	0.302368	2022-12-08T21:43:02.221963Z	0.302368
2022-12-10T14:40:34.554490Z	0.406211	2022-12-08T21:43:02.221964Z	0.406211
2022-12-10T14:40:34.554491Z	0.00500069	2022-12-08T21:43:02.221965Z	0.00500069
2022-12-10T14:40:34.554492Z	0.0799768	2022-12-08T21:43:02.221966Z	0.0799768
2022-12-10T14:40:34.554493Z	0.984842	2022-12-08T21:43:02.221967Z	0.984842
2022-12-10T14:40:34.554494Z	0.529011	2022-12-08T21:43:02.221968Z	0.529011
2022-12-10T14:40:34.554495Z	0.0256488	2022-12-08T21:43:02.221969Z	0.0256488
2022-12-10T14:40:34.554496Z	0.458684	2022-12-08T21:43:02.221970Z	0.458684
2022-12-10T14:40:34.554497Z	0.872662	2022-12-08T21:43:02.221971Z	0.872662
2022-12-10T14:40:34.554498Z	0.351182	2022-12-08T21:43:02.221972Z	0.351182
2022-12-10T14:40:34.554499Z	0.952182	2022-12-08T21:43:02.221973Z	0.952182
2022-12-10T14:40:34.554500Z	0.496678	2022-12-08T21:43:02.221974Z	0.496678
2022-12-10T14:40:34.554501Z	0.788744	2022-12-08T21:43:02.221975Z	0.788744
2022-12-10T14:40:34.554502Z	0.470404	2022-12-08T21:43:02.221976Z	0.470404
2022-12-10T14:40:34.554503Z	0.845835	2022-12-08T21:43:02.221977Z	0.845835

```
2022-12-10T14:40:34.554504Z    0.203485   2022-12-08T21:43:02.221978Z    0.203485

TEST: testRequestDataAsync7_Multiframe500 FROM
com.ospreydcs.datastore.admin.model.IQueryServiceDataTest
Multiple Data Frame: 500 PV wide frame, double values
  Data frame factory         : BASIC
  Number of data frames sent : 10
  Query page size (rows)     : 50
Number of frames sent to Datastore : 10
Test data frame column count       : 500
Test data frame row count          : 100
Test data frame value count        : 50000
Test data frame allocation (bytes) : 1202400
Total number of rows in Datastore  : 1000
Total number of values in Datastore: 500000
Total data in Datastore (bytes)    : 12024000
Result table column count      : 500
Result table row count         : 1000
Result table value count total : 500000
Result table null value count  : 0
Result table allocation (bytes): 12024000
Query duration                 : PT7.367866S
Query time (seconds)           : 7.367866
Query data rate (bytes/second) : 1631951.5040040086

Query results are missing providers: []
Timestamp maximum time difference: PT-41H-10M-1.399705S
Query results differ at the following locations:

Callback Function Test: 500 PV wide frame, double values
  Data frame factory         : BASIC
  Number of data frames sent : 10
  Query page size (rows)     : 50
Open query request at 2022-12-10T14:52:49.198342Z
Table loading wait completed at 2022-12-10T14:52:55.296824Z
Total loading wait duration : PT6.098482S
Result table allocation (bytes): 12024000
Query data rate (bytes/second) : 1971638.1879949798
FncNotify(Boolean, IDataTableDynamic) called at 2022-12-10T14:52:55.296817Z
  bolSuccess = true
  (tblResult==dtdResult) = true
  Callback duration : PT6.098475S
  Result table column count      : 500
  Result table row count         : 1000
  Result table value count total : 500000
  Result table null value count  : 0
  Result table allocation (bytes): 12024000
  Query duration                 : PT6.098475S
  Query time (seconds)           : 6.098475
  Query data rate (bytes/second) : 1971640.4510963808

Single PV Select Query: Test-Table500-DOUBLE499
Single query request        : SELECT `Test-Table500-DOUBLE499.value`  WHERE time >= '1970-01-
01T00:00:00Z'
Total loading wait duration : PT5.475526S
Total wait time (seconds)   : 5.475526
Result allocation (bytes)   : 48000
Data rate (bytes/second)    : 8766.281084228254

TEST: testRequestDataAsync8_Multiframe1k FROM
com.ospreydcs.datastore.admin.model.IQueryServiceDataTest
Multiple Data Frame: 1k PV wide frame, double values
  Data frame factory         : BASIC
  Number of data frames sent : 10
  Query page size (rows)     : 25
Number of frames sent to Datastore : 10
Test data frame column count       : 1000
Test data frame row count          : 100
Test data frame value count        : 100000
Test data frame allocation (bytes) : 2402400
Total number of rows in Datastore  : 1000
Total number of values in Datastore: 1000000
Total data in Datastore (bytes)    : 24024000
Result table column count      : 1000
Result table row count         : 1000
Result table value count total : 1000000
Result table null value count  : 0
Result table allocation (bytes): 24024000
Query duration                 : PT13.26531S
Query time (seconds)           : 13.26531
Query data rate (bytes/second) : 1811039.4706192317
```

```
Query results are missing providers: []
Timestamp maximum time difference: PT-41H-16M-1.403304S
Query results differ at the following locations:

Callback Function Test: 1k PV wide frame, double values
  Data frame factory         : BASIC
  Number of data frames sent : 10
  Query page size (rows)     : 25
Open query request at 2022-12-10T15:06:56.995194Z
Table loading wait completed at 2022-12-10T15:07:09.663713Z
Total loading wait duration : PT12.668519S
Result table allocation (bytes): 24024000
Query data rate (bytes/second) : 1896354.2620885677
FncNotify(Boolean, IDataTableDynamic) called at 2022-12-10T15:07:09.663700Z
  bolSuccess = true
  (tblResult==dtdResult) = true
  Callback duration : PT12.668506S
  Result table column count      : 1000
  Result table row count         : 1000
  Result table value count total : 1000000
  Result table null value count  : 0
  Result table allocation (bytes): 24024000
  Query duration                 : PT12.668506S
  Query time (seconds)           : 12.668506
  Query data rate (bytes/second) : 1896356.208064313

Single PV Select Query: Test-Table1K-DOUBLE999
Single query request        : SELECT `Test-Table1K-DOUBLE999.value`  WHERE time >= '1970-01-
01T00:00:00Z'
Total loading wait duration : PT9.823226S
Total wait time (seconds)   : 9.823226
Result allocation (bytes)   : 48000
Data rate (bytes/second)    : 4886.378466707373

TEST: testRequestDataAsync9_Multiframe2k FROM
com.ospreydcs.datastore.admin.model.IQueryServiceDataTest
Multiple Data Frame: 2k PV wide frame, double values
  Data frame factory         : BASIC
  Number of data frames sent : 10
  Query page size (rows)     : 20
Number of frames sent to Datastore : 10
Test data frame column count       : 2000
Test data frame row count          : 100
Test data frame value count        : 200000
Test data frame allocation (bytes) : 4802400
Total number of rows in Datastore  : 1000
Total number of values in Datastore: 2000000
Total data in Datastore (bytes)    : 48024000
Result table column count      : 2000
Result table row count         : 1000
Result table value count total : 2000000
Result table null value count  : 0
Result table allocation (bytes): 48024000
Query duration                 : PT28.761841S
Query time (seconds)           : 28.761841
Query data rate (bytes/second) : 1669712.3108357354

Query results are missing providers: []
Timestamp maximum time difference: PT-41H-25M-0.741489S
Query results differ at the following locations:

Callback Function Test: 2k PV wide frame, double values
  Data frame factory         : BASIC
  Number of data frames sent : 10
  Query page size (rows)     : 20
Open query request at 2022-12-10T15:44:16.258595Z
Table loading wait completed at 2022-12-10T15:44:49.913245Z
Total loading wait duration : PT33.65465S
Result table allocation (bytes): 48024000
Query data rate (bytes/second) : 1426964.773069992
FncNotify(Boolean, IDataTableDynamic) called at 2022-12-10T15:44:49.913228Z
  bolSuccess = true
  (tblResult==dtdResult) = true
  Callback duration : PT33.654633S
  Result table column count      : 2000
  Result table row count         : 1000
  Result table value count total : 2000000
  Result table null value count  : 0
  Result table allocation (bytes): 48024000
  Query duration                 : PT33.654633S
```

```
  Query time (seconds)           : 33.654633
  Query data rate (bytes/second) : 1426965.4938742016

Single PV Select Query: Test-Table2K-DOUBLE1999
Single query request        : SELECT `Test-Table2K-DOUBLE1999.value`  WHERE time >= '1970-01-
01T00:00:00Z'
Total loading wait duration : PT26.670255S
Total wait time (seconds)   : 26.670255
Result allocation (bytes)   : 48000
Data rate (bytes/second)    : 1799.7578200883343
```