Historian Engine Reference

Understand the historian engine architecture.

Reference Modules Historian UI | Engine

Historian Engine (Reference) manages the execution, archiving, and retrieval of time-series data within the FrameworX runtime environment.

Advanced Topic: This document provides deep technical insight into the Historian module execution engine. Most solutions don't require this level of understanding - the default engine behavior handles typical data archiving requirements automatically.

Overview
Archiving Process
Triggering Mechanisms
Archiving Data
Using Store and Forward
Store and Forward Modes
Configuring Store and Forward Batch Sizes
Row Count Indicator on Time-Series Databases
Reading History from Standard and Normalized Tables
Cross-Consumer Notification Order

Overview

The Historian Engine orchestrates:

- Event-driven and time-based data collection
- Multi-format storage management
- Store and Forward buffering
- Database schema optimization
- Concurrent table processing
- Data compression and retrieval

Understanding the engine helps when:

- Optimizing high-volume data collection
- Implementing custom storage strategies
- Managing distributed historian architectures
- Troubleshooting data gaps or delays

Archiving Process

The historian archiving process follows three main steps:

1. **Event Detection** - Trigger or tag change initiates archive request
2. **Data Storage** - Values written to configured storage location
3. **Synchronization** - Store and Forward ensures data integrity

Process Flow

Process Flow
Tag Value Change Event Detection Archive Queue Storage Writer Database Store & Forward Buffer Retry on Connection

---

Triggering Mechanisms

Trigger-Based Archiving

Triggers initiate archiving for entire historian table:

Supported Triggers:

- Tag values
- Tag properties
- Server namespace objects (e.g., Server.Minute)

Execution:

- Trigger change creates archive request
- ALL tags in table archived with same timestamp
- Values stored regardless of individual changes

Restrictions:

- Server-side objects only
- Cannot use client-side tags
- Single trigger per historian table

SaveOnChange Archiving

Continuous monitoring of individual tag changes:

Execution:

- Each tag monitored independently
- Archive on value change only
- Individual timestamps per tag
- More efficient for sparse data

Performance:

- Lower database writes
- Accurate timestamps
- Reduced storage requirements

Archiving Data

After the archive request is created, the platform system will check how the data will be stored depending on the StorageLocation of the current HistorianTable. You configure this option when creating the HistorianTable.

Archiving the data will differ if you use a SQL database or a TagProvider as a Historian.

Archiving to SQL database (Tag Historian)

The Datasets module has a pre-defined object named **TagHistorian**. By default, a SqlLite database is used, but you can choose other databases. Access the HistorianTables to learn how to do it.

When archiving to the SQL database defined by the TagHistorian object, you can choose between the Standard and Normalized table schemas.

Standard Tables

If you use standard tables, both Trigger and Tag Change events result in a single additional row in the database. Each column in the table corresponds to a Tag in the HistorianTable group, ensuring that all tags receive an entry, even if only one Tag has a new value.

The row's timestamp is determined by the Trigger object when the archive event is triggered. For the OnTagChange event, if there is only one tag in the table, it retrieves the tag's timestamp for the row. If there are two or more tags in the table, the timestamp will reflect the execution time of the code, which is always slightly later than the tag timestamp.

All tags listed in the associated HistorianTable are stored, independent of whether they have new values, sharing a single timestamp as defined earlier. In the case of OnTagChange events involving multiple tag value changes, a single row is inserted with all tags in the group, utilizing the timestamp of the Tag that triggered the event.



Avoid exponential database growth

To prevent rapid database growth, you can use the Time Deadband configuration to ensure that a new row is not created every time a Tag's value changes. The system will not archive a new Tag's value until the dead band time isn't reached. After the deadband, the new row is generated using the timestamp of the last event.

Standard Tables Schema

The following table describes all existing columns from a Standard SQL Table:

Column Name	Data Type	Size	Description

ID	BigInt	(8 Bytes)	The primary key used as a reference within the system.
UTCTimeStamp_Ticks	BigInt	(8 Bytes)	Date and time in Universal Time, represented in 64-bit .NET ticks. The value is based on 100-nanosecond intervals since 12:00 A.M., January 1, 0001, following the Microsoft .NET Framework standard.
LogType	TinyInt	(1 byte)	Auxiliary column indicating the insertion event: • 0: startup • 1: normal logging • 2:shutdown.
NotSync	Int	(4 Bytes)	Auxiliary column to show if the data was synchronized or not when the Redundancy option is enabled. See Deploying Redundant Systems.
TagName	Float	(8 Bytes)	Automatically generated column with the tag name as the title, storing data values using double precision.
_TagName_Q	Float	(8 Bytes)	Automatically generated column for the data quality of each tag, following the OPC quality specification.

You can usually assign up to 200 tags to each HistorianTable. However, the exact number can vary depending on how many columns your target database can accommodate. As a best practice, define tags in the same table if they have similar storing rates and process dynamics.

Normalized Tables

Normalized tables archive data only after an On Tag Change events. If you check the Normalized feature when creating or editing the HistorianTable, the Trigger option is disabled.

In this schema, each archived row carries only the timestamp, the Tag identifier, and the value of the Tag that generated the event.

For every normalized HistorianTable in the solution, the engine registers each of its tags in a single shared dictionary table named **TagsDictionary** and automatically creates four value tables whose names use the HistorianTable name as a prefix (`<HistorianTableName>_BIT`, `<HistorianTableName>_FLOAT`, `<HistorianTableName>_NTEXT`, and `<HistorianTableName>_REAL`). The **TagsDictionary** is shared by all normalized HistorianTables stored in the same database; its **TableName** column identifies which HistorianTable each tag belongs to.

TagsDictionary Schema

Column Name	Data Type	Size	Description
ID	BigInt	(8 Bytes)	Primary key used as a reference within the system.
TagName	NVarchar		The name of all the Tags configured as normalized databases on the Historian.
NotSync	Integer	(4 Bytes)	Auxiliary flag indicating whether the dictionary entry still needs to be synchronized to the target database during Store and Forward or Redundancy replication.
TableName	NVarchar	(256)	Name of the HistorianTable the tag belongs to, allowing a single TagsDictionary to serve every normalized HistorianTable in the database.

Normalized Value Tables Schema

The four value tables (`_BIT`, `_FLOAT`, `_NTEXT`, `_REAL`) share the following schema:

Column Name	Data Type	Size	Description
ID	BigInt	(8 Bytes)	The primary key of the table is used as a reference by the system.
UTCTimeStamp_Ticks	BigInt	(8 Bytes)	The date and time in Universal Time are expressed in 64-bit .NET ticks. The value represents 100-nanosecond intervals since 12:00 A.M., January 1, 0001, following the Microsoft .NET Framework's time standard.
ObjIndex	Integer	(4 Bytes)	Foreign key referencing the ID column in the TagsDictionary table, establishing a relationship.
ObjValue	It can be Bit, Float, NText, or Real, depending on which table it is.		It represents the value of the Tag at the specified timestamp. The data type varies based on the context of the associated table.
ObjQuality	TinyInt	(1 Byte)	Indicates the quality of the tag at the specified time, based on the OPC quality specification.
NotSync	Int	(4 Bytes)	Auxiliary flag indicating whether the row still needs to be synchronized to the target database during Store and Forward or Redundancy replication.

Redundancy replication of normalized tables

Normalized Historian tables can be replicated between the primary and secondary servers of a redundant pair. In the Redundancy settings, with redundancy enabled, check **TagHistorian** in the **Replication** row. When active, newly archived rows are flagged through the **NotSync** column and replayed to the partner server, keeping both historians in sync. See Deploying Redundant Systems.

Archiving Externally using a TagProvider

When archiving data externally using a TagProvider, the external system defines the schemas. It determines the structural organization, naming conventions, and other specific settings.

You need to specify the Protocol to add a new Storage Location using a TagProvider. The Protocol is an intermediary between the solution you build with the platform and the external data historian systems. They interpret and translate data formats, protocols, and other communication specifics to ensure seamless data archiving and retrieval. Currently, the platform provides three protocol options to connect using TagProviders:

- **CanaryLabs**: A robust data historian system that's optimized for real-time data collection and analytics. When archiving to CanaryLabs, the data is stored in a highly compressed format that facilitates faster retrieval and analytics.
- **InfluxDB**: An open-source time-series database designed for high availability and real-time analytics. InfluxDB is particularly useful when working with large sets of time-series data where timely data retrieval is of the essence.
- **GE Proficy**: A comprehensive platform that provides real-time data collection and advanced analytics capabilities. GE Proficy is a scalable system that integrates and analyzes vast amounts of industrial data.

You can use the Store and Forward feature when configuring a new StorageLocation using TagProvider.

Using Store and Forward

The Store and Forward feature buffers archive requests locally while the target database is unreachable and forwards them when the database returns.

When you define an StorageLocation using a TagProvider and disable Store and Forward, the archive requests events are sent directly to the external database as the events occur, independent of an existing working connection. A built-in protection exists for SQL-Dataset-Tag Historian targets with Normalized tables, buffering new rows and including them in the database every five seconds.

Store and Forward Process

When the Historian module receives an archive request, it'll try to store the data in the Storage Location. If unsuccessful, it stores the data in a locally created SQLite database. After an unsuccessful attempt, the Historian module checks the local SQLite database every second and retries the Target Database at most every 60 seconds. When the Target Database is reachable again, the module copies the rows inserted while it was inaccessible, in batches of up to 250 rows by default. With Keep Local Copy enabled, the forward cycle runs every 5 seconds instead. From FrameworX 10.1.5, this batch size is configurable via TServer.json — see *Configuring Store and Forward Batch Sizes* below.

HistorianTables are verified within a 4-second window. If not all tables are processed in time, the verification continues in the next cycle. If the copy process to the StorageLocation succeeds, meaning the connection was reestablished, the copied rows are removed from the temporary SQLite cache. If the temporary SQLite database is empty after the process, it is deleted.

In applications with a high volume of data and several tables to be synchronized, the data availability in the StorageLocation (external database) may take some time. The synchronization velocity depends on the insertion performance of the main database and the local databases' insertion performance (SQLite). In most applications, the Store and Forward synchronization process takes up to 1 second per table.

Due to the possible synchronization restrictions, it's essential to take the following points when deciding the database system to be used in your solution:

- For large projects with significant data volumes, it's recommended to use robust databases like SQL Server or Oracle for better performance.
- SQLite has a 10GB limit and limited performance and is suitable for smaller data models. The Keep a Local Copy feature works well for projects not requiring immediate synchronization, especially if the main database experiences occasional unavailability due to other projects or third-party software usage.

Multi-solution central database (SolutionName column)

When several solutions — for example multiple Edge nodes consolidating data through DataHub — forward their normalized Historian data to the same central database, add a **SolutionName** column to the central **TagsDictionary** table. This column is not created automatically; you add it once, manually, on the shared database.

While synchronizing through Store and Forward, the Historian engine fills **SolutionName** with the name of the originating solution for every dictionary entry it inserts. This keeps tags that share the same name, but come from different solutions, distinguishable in the shared dictionary instead of colliding. If the column is absent, synchronization still works, but the originating solution is not recorded.

Store and Forward Modes

Store and Forward is configured by two booleans on the StorageLocation: **StoreAndForward** and **KeepLocalCopy**. At module start the engine derives an internal mode from this pair, and that mode determines how often the local spool is checked and where the spool file lives. Store and Forward applies only to non-local (external) targets; when the configured Historian is itself a local SQLite database, Store and Forward is not engaged.

Store and Forward and database redundancy replication of the Tag Historian are mutually exclusive, so choose one of the two. The Designer blocks the combination, and if a solution has both, the runtime turns Store and Forward off at startup and logs the reason.

Configuration	Internal mode	Spool check interval	Behavior
StoreAndForward = off	0	—	Store and Forward disabled. Archive requests are sent to the target as they occur.
StoreAndForward = on, KeepLocalCopy = off	1	~1000 ms, with a retry of the target database at most every 60 s	Forward-only. Data is buffered to a local SQLite spool only while the target is unreachable, and the spool is replayed to the target once connectivity is restored. The spool is transient and is removed once empty.
StoreAndForward = on, KeepLocalCopy = on	2	~5000 ms	Hybrid keep-local-copy. A local copy of the data is retained in the local database in addition to forwarding to the target, suited to deployments that do not require immediate synchronization.

The spool and replay are local-database operations: data is buffered to a local SQLite database and replayed into the configured Historian (the external/target database) over the same database connection used for normal archiving — there is no separate network forwarding channel. Replay proceeds in batches (250 rows per pass by default — configurable from 10.1.5, see *Configuring Store and Forward Batch Sizes* below) as described under *Using Store and Forward* above.



Two modes, two intervals

The labels above are the actual configuration fields (StoreAndForward and KeepLocalCopy). The forward-only mode polls its spool roughly once per second and retries the target database at most once every 60 seconds; the keep-local-copy mode runs its forward cycle roughly every five seconds. There is no separate "Type 1 / Type 2" setting exposed to the user — the mode is derived internally from the two booleans.

Configuring Store and Forward Batch Sizes

Available from FrameworX 10.1.5c.

The number of rows flushed per synchronization cycle is configurable in `TServer.json` (located in the `FrameworX\MachineSettings` folder) under the `appSettings` object. Three independent keys control the Historian and Alarm engines:

Key	Default	Drives
`Historian.StoreAndForwardBatchSize`	250	Every Historian S&F sync site — both recovery passes after reconnect and the steady-state flagged-row drain.
`Alarm.StoreAndForwardRecoveryBatchSize`	100	Alarm S&F recovery pass (KeepLocalCopy = off — drains the local cache after remote reconnects).
`Alarm.StoreAndForwardNotSyncBatchSize`	1000	Alarm S&F replay pass (KeepLocalCopy = on — replays flagged rows while preserving the local copy).

All three values are validated to the range [50, 50000]. An out-of-range value keeps the default, and the engine writes a Warning to the trace log. Default values match the pre-10.1.5 hardcoded limits, so fresh installs behave identically to earlier versions until an operator edits the file.

How to change the batch size

1. Stop all running solutions in the machine (each solution runs its own TServer process, but all of them read from the same `TServer.json` configuration file).
2. Open `TServer.json` in the `FrameworX\MachineSettings` folder.
3. Inside the `appSettings` object, locate the `Historian` and `Alarm` sections and update the relevant values:

   Error rendering macro 'code': Invalid value specified for parameter 'com.atlassian.confluence.ext.code.render.InvalidValueException'

   ```
   "appSettings": {
     "Historian": {
       "StoreAndForwardBatchSize": 250
     },
     "Alarm": {
       "StoreAndForwardRecoveryBatchSize": 100,
       "StoreAndForwardNotSyncBatchSize": 1000
     }
   }
   ```

4. Restart all solutions. The new values take effect from the next sync cycle.

Performance considerations

Larger batches drain a backlog faster but increase CPU load per cycle — more rows are read from the local spool, deleted, and inserted into the remote database per pass. On Edge hardware with limited CPU headroom, oversized batches can interfere with real-time data acquisition. Tune in a staging environment before applying to production; for PostgreSQL/TimescaleDB targets the practical sweet spot is typically 1000–2500 rows.

Row Count Indicator on Time-Series Databases

Available from FrameworX 10.1.5d.

Each Historian table exposes a row-count indicator (the **Rows** value shown in the Historian Monitor). On most databases this is the exact number of rows currently in the table. On a **TimescaleDB / PostgreSQL** time-series backing, an exact `count(*)` over a very large table (a hypertable holding many millions of rows) is expensive, and running it while the Historian initializes can delay a table from opening. To keep startup and monitoring responsive regardless of table size, the row-count indicator on a time-series backing is **bounded at 100,000**:

- Below 100,000 rows, the indicator shows the exact count.
- At 100,000 rows or more, the indicator shows `100000`, meaning "100,000 or more".

> **Display indicator only — data is not limited**
>
> The 100,000 bound applies **only** to the displayed row-count indicator. It does not limit how much data is stored, archived, or retrieved — the actual number of stored rows is unbounded, and querying and trending return all data. On non-time-series databases (SQL Server, Oracle, SQLite, and PostgreSQL without the TimescaleDB extension) the indicator continues to show the exact, unbounded count.

Reading History from Standard and Normalized Tables

From FrameworX 10.1.6, history reads return the stored samples for Standard and Normalized tables alike:

- **Aggregated history** from the script toolkit (`TK.History.GetTagHistorianAggregated`), the REST API (`GET /api/v1/uns/{path}/aggregated`) and the AI Runtime tools computes its buckets from the samples stored for the tag, in Standard and Normalized tables and in external historians. In 10.1.5, the buckets come back empty in some cases even when samples are stored.
- **Long time windows:** when a window holds more stored samples than the historian reads in one request, each bucket carries a `samplesTruncated` flag. It is `true` on the buckets at and after the point where the historian stopped reading. Those buckets are incomplete or empty, so request a shorter window for them. In the script toolkit result, `samplesTruncated` is the last column.
- **Bad quality at a bucket start:** Average and Total no longer use the value of a sample with Bad quality at the start of a bucket.
- **Runtime start and stop markers:** on Normalized tables, the runtime start and stop markers are not counted as samples.
- **Text values:** text values that are not numbers are skipped, instead of counted as zero or as their leading digits.
- **REST history on Normalized tables:** `GET /api/v1/uns/{path}/history` returns the stored values of a tag in a Normalized table. In 10.1.5, the value field of each sample carries an internal synchronization flag (true or false) instead of the tag value.

For exporting a Historian table to a CSV file, see Historian Tables Reference.

Cross-Consumer Notification Order

A single tag value is observed by several runtime modules at once — for example the Alarm engine, the Historian, and scripts can all be bound to the same tag. When that tag changes, the runtime notifies the bound consumers in **registration order**: the order in which each consumer's binding was added to the tag. Notification is **not** ordered by consumer type or by any cross-consumer priority.

The practical consequence is a reliability contract worth designing around:

- The Alarm engine has **no** guaranteed precedence over the Historian (or over scripts) on the same tag change. Do not assume an alarm is evaluated before that same change is archived, or vice versa.
- If a design requires a guaranteed sequence — for example, "archive only after the alarm state has been evaluated" — build that dependency explicitly (for instance, have the first step write a derived tag that the second step is triggered from), rather than relying on the order in which consumers happen to be notified.

> **Event priority is per-consumer, not cross-consumer**
>
> An event priority does exist in the runtime, but it governs how an individual consumer handles its own event (for example, whether a handler runs immediately on the changing thread versus being queued for that module). It does not reorder which consumer is notified first across consumers. The cross-consumer fan-out is registration order regardless of priority, so it remains the case that no consumer type is guaranteed to run ahead of another on the same tag change.

In this section...
