Persistent versus Logged Properties

Document Version 1.0
October 2021

Copyright © 2021 PTC Inc. and/or Its Subsidiary Companies. All Rights Reserved.

User and training guides and related documentation from PTC Inc. and its subsidiary companies (collectively "PTC") are subject to the copyright laws of the United States and other countries and are provided under a license agreement that restricts copying, disclosure, and use of such documentation. PTC hereby grants to the licensed software user the right to make copies in printed form of this documentation if provided on software media, but only for internal/personal use and in accordance with the license agreement under which the applicable software is licensed. Any copy made shall include the PTC copyright notice and any other proprietary notice provided by PTC. Training materials may not be copied without the express written consent of PTC. This documentation may not be disclosed, transferred, modified, or reduced to any form, including electronic media, or transmitted or made publicly available by any means without the prior written consent of PTC and no authorization is granted to make copies for such purposes.

Information described herein is furnished for general information only, is subject to change without notice, and should not be construed as a warranty or commitment by PTC. PTC assumes no responsibility or liability for any errors or inaccuracies that may appear in this document.

The software described in this document is provided under written license agreement, contains valuable trade secrets and proprietary information, and is protected by the copyright laws of the United States and other countries. It may not be copied or distributed in any form or medium, disclosed to third parties, or used in any manner not provided for in the software licenses agreement except with written prior approval from PTC.

UNAUTHORIZED USE OF SOFTWARE OR ITS DOCUMENTATION CAN RESULT IN CIVIL DAMAGES AND CRIMINAL PROSECUTION. PTC regards software piracy as the crime it is, and we view offenders accordingly. We do not tolerate the piracy of PTC software products, and we pursue (both civilly and criminally) those who do so using all legal means available, including public and private surveillance resources. As part of these efforts, PTC uses data monitoring and scouring technologies to obtain and transmit data on users of illegal copies of our software. This data collection is not performed on users of legally licensed software from PTC and its authorized distributors. If you are using an illegal copy of our software and do not consent to the collection and transmission of such data (including to the United States), cease using the illegal version, and contact PTC to obtain a legally licensed copy.

Important Copyright, Trademark, Patent, and Licensing Information: See the About Box, or copyright notice, of your PTC software.

United States Governments Rights

PTC software products and software documentation are "commercial items" as that term is defined at 48 C.F.R. 2.101. Pursuant to Federal Acquisition Regulation (FAR) 12.212 (a)-(b) (Computer Software) (MAY 2014) for civilian agencies or the Defense Federal Acquisition Regulation Supplement (DFARS) at 227.7202-1 (a) (Policy) and 227.7202-3 (a) (Rights in commercial computer software or commercial computer software documentation) (FEB 2014) for the Department of Defense, PTC software products and software documentation are provided to the U.S. Government under the PTC commercial license agreement. Use, duplication or disclosure by the U.S. Government is subject solely to the terms and conditions set forth in the applicable PTC software license agreement.

PTC Inc., 121 Seaport Boulevard, Boston, MA 02210 USA

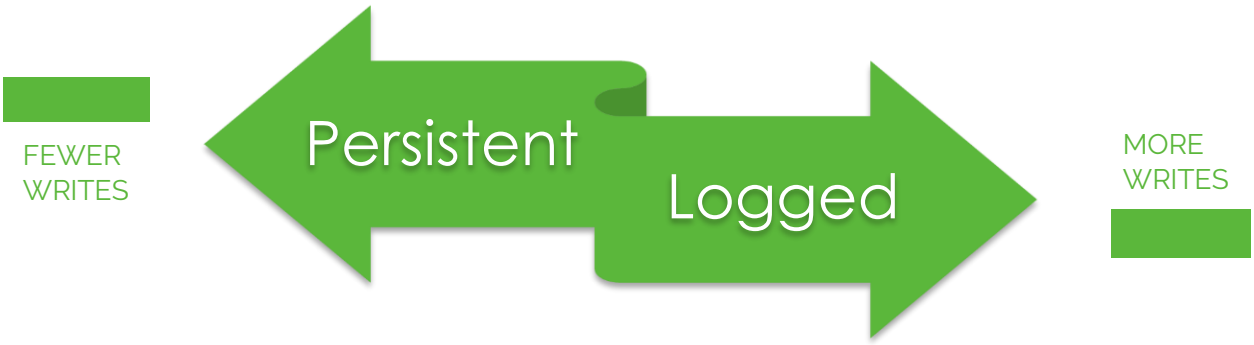
Table of Contents

Document Version History

Revision Date	Version	Description of Change
September 2021	0.2	Draft produced on EDC letterhead
October 2021	0.9	Final Draft
October 14th	1.0	Published to PTC Community

Executive Summary

ThingWorx provides several different "aspects" (or storage options) for how property values are saved. These options each have different implications for performance and scalability. Understanding those implications is important for designing a scalable IOT solution.



Persistent Properties are best used for non-telemetry data which will change infrequently (for example only a few times in a day) and where historical values are not required. When overused, Persistent properties can put significant pressure on the database layer of your ThingWorx implementation, leading to poor performance of your IOT application. As the number of Things in your IOT application scales up, the quantity or frequency of persistent properties per Thing needs to be carefully considered.

Logged Properties are best used for telemetry data where historical values need to be retained, but also for any other value that is expected to change frequently. Logged properties can create some additional requirements: a process for handling null/default values after restarts, more disk space, and a data retention policy. There are benefits as well, though, like more flexibility and scalability for the ingestion of larger volumes of data.

Persistent + Logged Properties perform database operations of both aspects. Combined use should be very limited – only properties that update infrequently (a few times a day), and that must be in-memory in the event of a ThingWorx restart.

In-Memory Only Properties are neither persistent nor logged – they are not stored to the database. These properties can greatly improve scale for values that need to be available for the application to drive UIs or compute other derived values that will be stored. However, high-frequency updates of in-memory properties can create scale challenges in HA (high availability) ThingWorx configurations where memory state needs to be constantly shared between multiple ThingWorx nodes.

Introduction

As the Enterprise Deployment Center has worked with a variety of teams on system design and optimization, it has become clear that the different options for storing property types, called "aspects", impact scalability. One of the most impactful effects is the difference between a "persistent" and a "logged" property. Both property aspects result in the value of the changed property being stored in the database, but the way they are stored, as well as the effort required to store them, is quite different.

Understanding these differences can help you make decisions on which is best for your use case, and what administration and/or application design considerations each option presents. The Thing Properties section of the ThingWorx Help Center provides details on how the different aspects behave. This article will focus on the two most used options:

Aspect	Persistent	Logged
Ideal Use-Case	**Non-Telemetry:** Accessed often Does not change often Latest value only (not time-series)	**Telemetry:** Changes often Time-Series Data
Where is it stored?	Persistent Provider Database	Value Stream … Data Provider Database
When is it updated?	Any and every value change with a newer timestamp	Based on selected "Data Change Type": *Most commonly "Always" or "Value" (with optional threshold)*
How is it persisted?	UPDATE TABLE property_vtq …	INSERT INTO value_stream …
ThingWorx Restart?	Most-recent value loads into memory at startup automatically	Default Value is used if configured until first new value is received
Things to Consider	• UPDATEs can be slower than INSERTs • Does not retain historical values • Dead/Ghost row cleanup needed in database (PostgreSQL or SQLServer)	• Stores historical values • Latest value not automatically loaded into memory when ThingWorx starts • Data Retention Policy: How long to keep data before downsampling, archiving, or purging?

From a scalability perspective, when comparing "persistent" and "logged", it is important to think about how data flows from the edge through any busines logic your application applies as it is ingested before being stored.

Example Scenario 1 – Simple Property Example

Let's take the following data scenario as a simple example, and compare the database activity for this scenario with three different priority aspect configurations:

- There is a fleet of 1,000 connected machines ("assets") that send data every minute.
- Assets send the `isServicePanelOpen` property every minute, whether changed or not.
- On average, the service panel is opened and then closed twice per day on each asset.

With "persistent", each value from the edge triggers a database UPDATE, whether or not it is different than the previous value. This scenario generates 1,440,000 UPDATEs per day.

$$(1\ UPDATE) \times\ (60\ minutes) \times (24\ hours) \times (1{,}000\ assets)$$

$$= \mathbf{1{,}440{,}000\ UPDATEs\ per\ day}$$

Also remember that each UPDATE is treated like two database operations: a new row is added, and the old row is marked as a ghost row (in SQL Server terms) or dead tuple (in PostgreSQL terms) for later cleanup. Think of this like the database applying an "immediate, zero data retention policy" – you are purging historical data from your system as quickly as the database engine can get to its maintenance tasks.

That's a lot of database activity for one property on a fairly small asset fleet. This activity multiplies quickly as asset count, property count, and data frequency increase.

Now, let's change this scenario to a "logged" property set to the "Always" data change type. This still generates 1,440,000 database operations per day, but now they are INSERTs.

INSERTs are typically quicker than UPDATEs, as they do not result in creating a ghost row / dead tuple. However, you are trading speed for storage; your value stream will grow by 1,440,000 rows every day. It is important to determine if this historical data has analytical value, how long it needs to be in "hot" storage within ThingWorx for your use-cases, and after that time if it should be archived to a secondary database, downsampled to reduce storage requirements, or simply deleted it from the system.

Finally, let's look at the third, and likely best option for this particular scenario: a "logged" property set to the "Value" data change type. Now, no matter how many times the edge sends the same value for the same property, it will only be stored in the value stream if it has changed:

$$(1\ INSERT) \times\ (on\ average, 2\ open + 2\ close\ actions\ per\ day) \times (1{,}000\ assets)$$

$$= \mathbf{4{,}000\ INSERTs\ per\ day}$$

We have gone from 1,440,000 database transactions down to 4,000, on average, per day. This is an enormous reduction in database activity, and yet none of the information that this specific sensor provides is lost in the process.

Example Scenario 2 - Monitoring a Large Connected Product Fleet

Let's apply this logic to a more complete scenario: a fleet of refrigerated vending machines that send data about operations (keeping the product cold) and consumables (drinks, coins, and bills) to plan service visits.

75,000 Refrigerated Vending Machines		
Data Refresh Rate : All Properties in a Batch, once every 10 minutes		
Property	**Type**	**Details**
1. `serialNumber`	String	Unique identifier of this vending machine
2. `locationAddress`	String	Machine location for service
3. `productid_1`	String	Used to indicate which beverage type is stocked in the machine
4. `productid_2`		
5. `productid_3`		
6. `productid_4`		
7. `itemsRemaining_1`	Integer 0 … 24	Generate alert if running out of one product type
8. `itemsRemaining_2`		
9. `itemsRemaining_3`		
10. `itemsRemaining_4`		
11. `interiorTemperature`	Number	Track product temperature and refrigerator operations.
12. `exteriorTemperature`		
13. `runtimeSinceService`	Integer	Seconds the machine's cooling system has been active since last service
14. `serviceDoorStatus`	String	"Open", "Closed", "Locked"
15. `numCoin_5`	Integer 0 … 100	Generate alert if machine has too much or too little of one currency type
16. `numCoin_10`		
17. `numCoin_25`		
18. `numCoin_100`		
19. `numBill_100`		
20. `numBill_500`		

As the devices send once every ten minutes, the average data rate is fairly low.

$$(75{,}000\ assets) \times \frac{(20\ properties)}{(60\ seconds) \times (10\ minutes)} = 2{,}500\ Writes\ per\ Second$$

However, if several machines were to send updates at roughly the same time the server could see large pulses of data all at once, so we should still be thoughtful about how we store these properties to scale effectively.

In this scenario, the only properties that likely should be "persistent" would be `serialNumber` and `locationAddress` as these will rarely, if ever change. The others could potentially provide a different value every data update, so "logged" is the best choice.

Depending on how often the type of drink in the machines is changed, the four `productid_<num>` fields could be "persistent" or "logged", but having them as "logged" creates potential for analytical use later (for example, how did the rate of drink purchases increase over time for cola, root beer, or ginger ale)?

When Should I Use Persistent Properties Then?

Persistent Properties make sense for properties that change *very* infrequently – typically non-telemetry data that changes only a handful of times per day – and for use cases where the latest value for that property must always be in memory, including if ThingWorx has been restarted, for your application logic to function.

If a property value changes more frequently than that, say once an hour for example, then you may have a more scalable solution if such a property is "logged". Write your application business logic to handle the situation that the latest value may not be in memory if ThingWorx is restarted for some reason (i.e. the application logic should first check if the value is in memory, and if not, then query it from the database).

Also, keep in mind the planned size of your asset fleet when looking at the quantity and frequency of persistent properties. "10 Persistent properties that update once an hour" is likely fine if you only have dozens of connected assets… but if you plan to have hundreds or thousands of connected assets, this is likely to become a scale issue.

The Importance of a Data Retention Policy for Logged Properties at Scale

As the examples illustrated, logged properties will typically scale far more effectively for data that is expected to change. However, by their nature logged properties will create additional application and infrastructure requirements that you need to address.

As logged properties are a Value Stream (time-series data, i.e. a stream of time-stamped values), your database will now be storing multiple data points for each Thing-property pair, whereas with a persistent property it was overwriting or replacing the previous value. If that time-series data is allowed to build up, it can lead to poor IOT application performance for query operations, as well as excessive disk space requirements.

Defining your data retention policy becomes important as it helps you define how much data history your use case(s) require to deliver their business value. Once data ages past this point, you can:

- **Downsample:** Could older "one data point every five minutes" data be replaced with a single value for the hour? Could use the: Last value? Highest or Lowest value?
- **Aggregate:** Could older "one data point every five minutes" be reduced to a set of derived values? Could use the: Minimum, Maximum, Median, Average…
- **Archive:** If older data may still be needed in the future, where should it be transferred to? Perhaps a simple file or secondary data store/lake, something with better compression for long-term "cold storage".
- **Purge:** Once no longer needed and/or archived/moved elsewhere, how/when will data be removed from the system?

Performing these types of data retention tasks will require some compute resources to perform the desired aggregation or data transfer/removal operations, but this type of data clean-up allows to do have much more control over the performance of your IOT application as the scale of your overall implementation increases.

The Best – and Worst – of Both Worlds: Persistent + Logged Properties

For scenarios where a property changes infrequently, but is used by business logic frequently, the "if in memory get, else query from database" approach may not be viable. In these scenarios, combining Persistent and Logged together appears to satisfy both of these requirements, but it is important to understand what is happening behind-the-scenes with this combination and its implications for scalability.

Quite simply, a property marked as both persistent and logged will do **both** sets of database operations. It will UPDATE the database table used to store persistent values, *and* it will INSERT into the database table used to store logged value stream data. If "persistent+logged" is mistakenly used on a property where there is a high frequency of data change, this can result in significant scalability limitations. A "persistent+logged" property makes sense in similar scenarios as a "persistent only" property – the same "very infrequent data change" guidance would apply here (i.e. my business logic needs the current value of this property in memory at all times), with the added business requirement that the history of this property needs to also be retained.

Using In-Memory Properties … With Caution!

One alternative design pattern is a property that is *neither* persistent nor logged – stored in-memory only, with a second "logged" property that stores data when necessary. Then, in the event of a ThingWorx restart, a "pre-load" service can be written to run during the restart, read the latest values out of the "logged" property, and put them into the in-memory cache property.

This approach can work for a small number of properties, helping to relieve database pressure at the cost of ThingWorx memory; more properties means greater memory requirements for the system. At larger volumes, this approach could yield worse performance than the initial problem you were trying to solve – especially in HA (high availability) ThingWorx configurations. In a clustered (HA) ThingWorx implementation, this impact is multiplied by the fact that multiple nodes share a common memory state; the increase in memory and network traffic to replicate these changes across all ThingWorx and Ignite nodes can result in performance issues by itself.

Summary

The storage mechanism you choose for the properties in your ThingModel can have a large impact on the scalability of your solution as a whole. For any data that will change frequently (more than a few times per day), logged properties are typically the most scalable solution. However, logged property values can require some additional thought towards your disk space and data retention requirements, as well as some additional handling within your application for scenarios where the value being requested has not been sent since the last ThingWorx restart. Once addressed, this approach uses the more efficient database operation to store values, and takes advantage of the ValueStream Subsystem, providing the most flexibility in terms of database options and tuning.