

Persisted Version Store

Last updated by | Vitor Tomaz | Jun 8, 2022 at 5:32 AM PDT

Contents

- [PVS \(Persisted Version Store\)](#)
- [Version tracking and cleanup](#)
- [In-row Versions:](#)
- [Off-row in-database version store](#)
- [Version tracking and cleanup:](#)
- [Queries](#)
- [Common Issues w/ PVS in Pool environment.](#)

PVS (Persisted Version Store)

Persistent Version Store (PVS) in SQL Server is a mechanism for keeping the existing versions generated for snapshot isolation in the database itself instead of the original tempdb version store that was introduced with versioning in SQL 2005. The main benefit of PVS is that facilitates the Accelerated Database Recovery.

There are three main components to Persistent Version Store:

Version tracking and cleanup

In-row Versions:

In-row versions are a payload that is appended to the versioning information in the record which allows the previous image of the row to be generated from the current image. We use two kinds of payload: diff and full-row. Full-row diffs are used for small rows where the CPU cost of generating a diff versus just copying the row makes it more sensible to just copy the row. Additionally, there is overhead for a diff so it is possible that the full-row could actually be smaller in some cases. For off-row, the layers below VersionMgr are abstracted from knowing where the version will end up. For in-row versions the BTreeRow level needs to know if in-row could potentially be used so that it can split the page to make enough room for the updated row with the in-row content if needed. The BTreeRow layer makes worst case assumptions about the potential row size to guarantee that there will be space. The IndexPageRef/HeapPageRef/PageRef level can make the final decisions about whether to actually generate an in-row version or not depending on the actual size of the payload that will be needed.

Off-row in-database version store

Off row version store will be persisted in the database where the transaction generates versions. Each recovery unit will host a PVS manager and provide the AddRecord(...) and GetRecord(...) interface. We keep the interface unchanged from version store to upper layer versioning system. PVS in database

also optimized for insert only operation. The insert operation will be logged in non-transactional manner, then during crash recovery the normal redo process will make sure the version durability

Version tracking and cleanup:

Cleanup of in-row versions: If the stale (versions no longer needed by any snapshot scan) in-row versions are not cleaned up quickly, they would occupy space on the page. This might cause new inserts/updates to split the page (in case of btrees) and new allocations (in case of heaps) leading to fragmentation that can impact the performance of the database. As such it becomes important to track the stale versions and cleanup aggressively.

Cleanup of persisted off-row versions: Stale versions in off-row version store also lead to increased database size. Cleanup of persisted off-row versions is in many ways similar to that of cleanup of versions in tempdb. The older versions of the rows can be cleaned up if it is not needed by any snapshot scan. The off-row versions are stored in dedicated heap pages.

Please note that even before PVS, tempdb size was taken into account with customers quota of data/log file. Now the data merely moves into the database so yes it still accounts for the quota. Here's how you could track the usage from our telemetry :

Queries

```
MonSqlTransactions
| where AppName in( "" )
| where database_id == XX
| where persisted_version_store_kb > 0
| project PreciseTimeStamp , persisted_version_store_kb
| order by PreciseTimeStamp asc nulls last
```

The following kusto query can be used to determine if active transaction/scan is holding up PVS cleanup. If the value returned is above 100GB or so, then this is most likely the cause of PVS holdup. If true, then check if the database is close to full (step b).

```
// PVS holdup trend
MonSqlTransactions
| where AppName == "e7e22f28bdb5"
| where database_id == 7
| where isnotempty(pvs_off_row_page_skipped_oldest_active_xdesid)
| where TIMESTAMP > ago(3h)
| extend pvs_size_gb_heldup_by_active_scan_or_tx = (pvs_off_row_page_skipped_oldest_active_xdesid+pvs_
| project TIMESTAMP, pvs_size_gb_heldup_by_active_scan_or_tx
```

Step b.

Check the database/pool space usage with the following Kusto query.

<https://sqlazurescus2.kustomfa.windows.net/sqlazure1> 

```

MonSqlRgHistory
| where AppName == "e7e22f28bdb5"
| where database_id == 7
| where TIMESTAMP >= ago(60min)
| where isnotempty(physical_database_name)
| summarize arg_max(TIMESTAMP, size_on_disk_bytes, max_size_mb, code_package_version, hotpatch_data_pa
| extend data_size_on_disk_kb = iff (type_desc == 'ROWS', size_on_disk_bytes / 1024, 0), data_size_ma
| summarize max(TIMESTAMP), sum(data_size_on_disk_kb), sum(data_size_max_kb) by LogicalServerName, dat
| join kind = leftouter(
  MonDmIoVirtualFileStats
  | where TIMESTAMP >= ago(120min)
  | where isnotempty(logical_database_id)
  | summarize arg_max(TIMESTAMP, spaceused_mb) by LogicalServerName, database_id, AppName, type_desc, fi
  | extend data_size_used_kb = iff (type_desc == 'ROWS', spaceused_mb * 1024, 0)
  | summarize sum(data_size_used_kb) by LogicalServerName, database_id, AppName ) on LogicalServerName,
  join kind = leftouter( MonResourcePoolStats
  | where TIMESTAMP >= ago(60min)
  | summarize arg_max(TIMESTAMP, data_storage_percent, resource_pool_storage_limit) by LogicalServerName
  | project LogicalServerName, AppName, pool_size_used_kb = toreal(resource_pool_storage_limit) * data_s
  | project LogicalServerName, database_id, AppName, db_space_left = strcat( round(sum_data_size_max_kb/
    ,round((pool_size_max_kb - pool_size_used_kb)/1024.0/1024.0, 1), 'GB'))

```


Example query result:

LogicalServerName	database_id	AppName	db_space_left	pool_space_left
lci-frigg-appdbserver-22c1853c	7	e7e22f28bdb5	4096.0GB - 3235.0GB = 861.0GB	0.0GB - 0.0GB = 0.0GB

Common Issues w/ PVS in Pool environment.

Active transaction was causing Persisted version store (PVS) to grow. In an elastic pool, if there are long running scans or transactions, that can cause version store to grow for that duration.

To handle, keep track of long running transactions and kill them if they are not necessary. If they are necessary, then ensure that sufficient disk space is allocated to the elastic pool so that databases do not run into database full situations.

The version store size for the database 'lold' is now reclaimed. Please refer to the troubleshooting section of Accelerated Database Recovery section Manage accelerated database recovery - SQL Server | [Microsoft Docs to identify the sessions/transactions that are causing the version store growth.](#) 

How good have you found this content?

