**Db2 v12.1.1.0: Making Reorg Rebuild Better for Your Business**

**By Zach Hoggard, Chris Stojanovski**

[Db2 pureScale](https://www.ibm.com/docs/en/db2/12.1?topic=editions-introduction-db2-purescale-environment) is the shared-disk cluster solution that aims to achieve zero-downtime while running your mission-critical, high-performance OLTP workloads. To meet these stringent high-availability expectations, Db2 pureScale incorporates several features – multiple active members, full redundancy across all components, geographically dispersed clusters, HADR. However, true high availability isn’t just about resilience – it must also extend to database maintenance operations. Any disruption caused by maintenance activities can impact to the overall availability of the cluster, making it essential to minimize downtime during these processes as well. In this blog, we are going to talk about one of such maintenance operations, index reorganization, and the improvements that we made in Db2 v12.1 to support it as a fully online operation for all flavours of index maintenance.

**The importance of Indexes and Index Reorganization**

As you know, Db2 indexes are critical for optimal OLTP performance. OLTP workloads often contain many inserts, updates, and deletes which will correspondingly change the index. Over time many of these changes may begin to degrade the index structure (e.g. not optimal number of index tree levels, sparse index pages, etc.), which can impact query performance and space utilization.  Index reorganization using the [REORG INDEXES](https://www.ibm.com/docs/en/db2/12.1?topic=commands-reorg-indexindexes) command is the solution for keeping indexes in their optimal form and therefore is typically run often to keep databases performing well. Since REORG INDEXES may be run often, it is critical for it to be a fully online operation. This maintenance operation has been supported as a fully online operation in non-pureScale environments, and now since v12.1 this design has also been extended to pureScale environments, which results in causing very little disruption to concurrently running workloads and can even be run automatically through the automatic maintenance feature. With this, running index reorganization online will allow your pureScale OLTP databases to remain fully available while continually optimizing their performance.

**Types of Index Reorganization**

There are three types of index reorganization supported by Db2: CLEANUP, RECLAIM EXTENTS, and REBUILD.

A CLEANUP is used to "compact” the index tree, which increases the space utilization in the index and reduces the number of pages accesses required to get to the requested key. This is accomplished by freeing empty/deleted keys and merging adjacent index pages.

A RECLAIM EXTENTS is used to release free space reserved by the index back to the tablespace. This is done by shuffling around empty index pages in the index to group together freed pages which then are released back to the tablespace in groups of extents.

Both CLEANUP and RECLAIM EXTENTS are often used at the same time to optimize the layout of the index, and both have been supported to be run fully online, that is, while maintaining full read and write access to the table and index, for a long time in both Db2 and Db2 pureScale environments.

 The third type of index reorganization, REBUILD, performs more extensive changes, as it is used to fully rebuild an index from scratch into its most optimal form for both space utilization and fastest access time. It is the only type of index reorganization that results in an index tree with the minimum number of levels. A rebuild is typically run less often as it is more heavy-handed, but important to run sometimes for the long-term health of indexes. Like CLEANUP and RECLAIM EXTENTS, REBUILD has been supported to be run while maintaining full read and write access to the table and index in Db2, but pureScale environments was the exception, as REBUILD did only support running when there is no access allowed to the table or index. However, this is the case no longer in Db2 v12.1.1. It’s finally here! We’ve enhanced Index Reorganization Rebuild in pureScale to let your tables remain read and/or write accessible while the rebuild is running. In the rest of the blog, we are going to talk about the ins and outs of these changes, and with this, give you an opportunity to exploit this new behaviour in your pureScale environments.

**When should I run an Index Reorganization?**

This is an important topic to discuss before we move any further, as even with Index reorganization being now a fully available maintenance operation with minimal impact to concurrent workloads, these operations do require additional processing, and it should be considered when deciding when is the right time to run these. The first thing is to understand whether the index would benefit from a reorganization, and in particular, from a REBUILD, which would deliver the most benefit but also would incur in the most cost and impact during its processing. As always, we rely on the [REORGCHK](https://www.ibm.com/docs/en/db2/12.1?topic=commands-reorgchk) utility for this purpose, which is a tool that uses the statistical information collected by the RUNSTATS utility to offer suggestions on when to reorganize your indexes.

REORGCHK has several formulas that are index specific (F4-F8), and that show in which ways your index may not be optimal and has an indicator to say when indexes have reached pre-defined thresholds which suggest the need of a specific type of reorganization. This is primarily how you should be determining when to run index reorganization. If you want to dig further, you can use the [REORGCHK\_IX\_STATS](https://www.ibm.com/docs/en/db2/12.1.0?topic=mrv-reorgchk-ix-stats-reorganization-index-statistics) procedure to retrieve the index statistics that are used to indicate whether there is a need for an index reorganization. We will leave a discussion on the different statistics provided by this for a future blog. The important thing for this is that through this procedure users can get access to the recommendations based on the pre-defined thresholds for the F4-F8 formulas.

As mentioned earlier, the automatic maintenance feature can even automate this process. Automatic maintenance uses the statics from REORGCHK and will automatically run index reorganization when the formula thresholds have been crossed.

**How do I run a run an Index rebuild in pureScale?**

Running an index reorganization rebuild in pureScale is just as straightforward as on a non-pureScale setup. Similar to non-pureScale there are a few prerequisites necessary to ensure the unimpacted operation of concurrent transactions while a rebuild with write access is running, as well as prevent any impact to the index reorg rebuild while its running. From a high level when rebuilding an index, a shadow index is built which acts as a hidden copy of the original index until the operation is complete and it becomes the main index. If write access is used, the index may change during the rebuild and these changes need to be tracked and replayed into the shadow index to ensure that it contains all changes made during the rebuild; this is done using special log records. Thus, before starting ensure the database has enough log space to handle the additional logging during the rebuild with write access and sufficient disk space for the shadow object (covered in detail in the next section).

There are only a few pureScale-specific limitations to keep in mind. First, log archiving must be enabled because any insert, update, or delete (IUD) operations during the rebuild must be logged and available for replay.  Additionally, index reorganization rebuild is blocked in certain scenarios: when the cluster has mixed software levels, when a pureScale member is being added online, or when the table has more than 200 indexes.

With these considerations in place, running an index reorganization rebuild in pureScale is as simple as it is in non-pureScale. No syntax changes are required—you use the exact same command:

$db2 -v "REORG INDEXES ALL FOR TABLE T1 ALLOW READ ACCESS"

REORG INDEXES ALL FOR TABLE T1 ALLOW READ ACCESS

DB20000I  The REORG command completed successfully.

$ db2 -v "REORG INDEXES ALL FOR TABLE T1 ALLOW WRITE ACCESS"

REORG INDEXES ALL FOR TABLE T1 ALLOW WRITE ACCESS

DB20000I  The REORG command completed successfully.

Monitoring of index reorganization rebuild in pureScale is very similar to how any type (rebuild, cleanup, or reclaim extents) would be monitored regardless of if it is pureScale. This is done using the db2pd –reorgs index. For pureScale environments db2pd will only print information from the member it is attached to, thus, this may not result in a complete set of all index reorganizations happening in the cluster. In order to get a listing of all index reorganization rebuilds in the pureScale cluster db2\_all should be used with db2pd. For example:

db2\_all "db2pd -db TESTDB -reorg index"

This can be used to both monitor in progress and completed index reorganization rebuilds. The output to the above command will look similar to the following:

Index Reorg Stats:   
Retrieval Time: 2024-02-27 13.15.14   
TbspaceID: 2        TableID: 4   
Schema: REGRESS1 TableName: T1   
Access: Allow write   
Status: In Progress   
Start Time: 2024-02-27 13.12.24   End Time: -   
Total Duration:  -   
Prev Index Duration:  -   
Cur Index Start: 2024-02-27 13.12.24   
Cur Index: 1            Max Index: 1            Index ID: 1   
Cur Phase: 1          (Scan   )   Max Phase: 3   
Cur Count: 2                      Max Count: 22   
Total Row Count:  -

**How does Index Reorganization Rebuild work in pureScale?**

A diagram of a diagram

Description automatically generated with medium confidence

Index Reorganization Rebuild can be broken down into four phases:

1. Initialization
2. Build
3. Catchup
4. Completion and Cleanup

Of these four phases the first and last phases (Initialization, Completion and Catchup) occur a single time at the initialization and completion of the rebuild. The second phase (Build) and third phase (Catchup) occur once per index being rebuilt.

**Initialization**

The initialization phase gathers all resources and performs the setup necessary to begin the index reorganization rebuild. A new transaction is created, a table (or partition) lock is acquired, and shared memory is allocated in the CF for coordination purposes. The access type of the rebuild will dictate what table or partition lock is necessary. For index reorganization rebuilds with allow write access an IN lock is acquired, for allow read access a U lock is acquired, and for no access a Z lock is acquired. For index reorganization rebuild with allow write access this phase includes a wait for all open write access transactions to either commit or rollback.

**Build**

During the build phase a duplicate shadow index is built using the table data. For index reorganization rebuild with allow write access this phase includes additional logging for use during the next phase, catchup.

**Catchup**

The catchup phase allows for changes that have occurred against the table since the start of the build phase to be applied to the shadow index. Note that this phase is only done for index reorganization rebuild with allow write access. The purpose of this phase is to allow for the shadow index to be brought back into sync with the table. During this phase the additional log records created in the previous phase are applied in iterations up to the end of the logs. If a catchup iteration is unable to reach the end of the logs due to the rate of incoming new log records the rebuild will move on to building the next index. However, if this is the last index and catchup is still unable to complete the table would be put into read-only (U lock mode) to allow catchup to complete for all indexes.

**Completion and Cleanup**

 Once the final catchup iteration has completed the table or partition lock is upgraded to a Z lock. With the Z lock the shadow index is able to be switched with the original index object. The original index object is invalidated on all members and the memory allocated on the CF for coordination purposes is freed. Finally, the original index is dropped, the Z lock is released, and the transaction commits.

**A real-world example**

Now that you have an understanding of the design of an index reorganization rebuild, let’s go through a real-world example so that you can see some data. This example walks through a rebuild with ALLOW WRITE ACCESS of an index on a not-partitioned table in a pureScale environment, that is, online index reorganization rebuild. The table has been pre-populated with 10,000,000 rows and has a unique index on an INT column. While the online index reorganization rebuild is running a separate connection will do 200,000 updates on the column that has been indexed. This example will explore the log space usage and tablespace usage over the course of the online index reorganization rebuild.

Let’s start by taking a look at the initial state of the relevant information used in this example.

**Table/Index Definition:**

CREATE TABLE REG\_TBL (\

          I1 INT NOT NULL \

          S2 INT NOT NULL \

          B3 BIGINT NOT NULL \

          D4 DECIMAL(9,2) NOT NULL \

          V5 VARCHAR(255));

CREATE INDEX REG\_IDX ON TABLE\_REG (I1);

**Transaction log space:**

LOGFILSIZ 1000: Number of 4K pages per log file

LOGPRIMARY 20 : Number of preallocated log files

LOGSECOND 20  : Number of on-demand log files

This results in a total log space of 163,039,348 bytes or about 155 MB.

**Tablespace configuration:**

PAGESIZE   4K

EXTENTSIZE 32

The load of 10,000,000 rows results in an index with 41984 pages (164 MB).

**Phase 1: Initialization**

Right after the REORG INDEXES ... REBUILD is issued the “initialization” phase begins. As mentioned earlier, this creates a new transaction and does some setup such as taking locks and allocating memory. The locks acquired look like this:

|  |  |  |
| --- | --- | --- |
| **Transaction Handle** | **Lock Type** | **Lock Mode** |
| 3 | ExtentMoveLock | ..S |
| 3 | PoolLock | .IX |
| 3 | RowLock | .NS |
| 3 | AlterTableLock | ..X |
| 3 | TableLock | .IN |
| 3 | TableLock | .IS |

At this time the log space used is minimal as only log records pertaining to the creation of the transaction and the REORG INDEX have been written. This results in 652 bytes of used log space:

|  |  |
| --- | --- |
| **TOTAL\_LOG\_AVAILABLE (bytes)** | **TOTAL\_LOG\_USED (bytes)** |
| 163,039,348 | 652 |

At this time the shadow index has not yet been created so there is no change in tablespace usage from the initial state:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **TBSP\_USED\_PAGES** | **TBSP\_TOTAL\_PAGES** | **TBSP\_PENDING\_FREE\_PAGES** | **TBSP\_USED\_MB** | **TBSP\_TOTAL\_MB** | **TBSP\_PENDING\_FREE\_MB** |
| 224,064 | 229,376 | 0 | 875 | 896 | 0 |

**Phase 2: Build**

The build phase creates the shadow index in the same tablespace and starts building it from a table scan. This means that there is an increase in tablespace usage (the index usage almost doubles to 324 MB) and a small amount of log space is used to write log records indicating the start location for the catchup phase. There are no locking changes.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **TOTAL\_LOG\_AVAILABLE (bytes)** | | | **TOTAL\_LOG\_USED (bytes)** | |
| 163,034,004 | | | 5,996 | |
| **TBSP\_USED\_PAGES** | **TBSP\_TOTAL\_PAGES** | **TBSP\_PENDING\_FREE\_PAGES** | | **TBSP\_USED\_MB** | **TBSP\_TOTAL\_MB** | **TBSP\_PENDING\_FREE\_MB** |
| 266,784 | 270,336 | 0 | | 1,042 | 1,056 | 0 |

**Phase 3: Catchup**

During catchup any concurrent transactions that altered the index during the build phase are replayed on the shadow object. This is facilitated using special log records specifically for online index reorganization rebuild. The writing of these additional log records creates a slight overhead that should be accounted for. To understand the overhead associated to these new log records, we can compare it to another scenario where updates were done prior to the online index reorganization rebuild. Looking at the logging associated where updates were done concurrently with online index reorganization rebuild, we see the following usage:

|  |  |
| --- | --- |
| **TOTAL\_LOG\_AVAILABLE (bytes)** | **TOTAL\_LOG\_USED (bytes)** |
| 162,026,832 | 1,013,168 |

Comparing this to the log usage when updates were done prior to online index reorganization rebuild reveals the additional overhead represented by the difference:

|  |  |
| --- | --- |
| **TOTAL\_LOG\_AVAILABLE (bytes)** | **TOTAL\_LOG\_USED (bytes)** |
| 162232493 | 807,507 |

The difference of 205,661 bytes accounts for the additional overhead due to the special logging done during concurrent transactions. During this stage there are no changes in locks until the final iteration of catchup where the table enters read only mode through a U lock in preparation for completion and cleanup.

|  |  |  |
| --- | --- | --- |
| **Transaction Handle** | **Lock Type** | **Lock Mode** |
| 3 | ExtentMoveLock | ..S |
| 3 | PoolLock | .IX |
| 3 | RowLock | .NS |
| 3 | AlterTableLock | ..X |
| 3 | TableLock | .IN |
| 3 | TableLock | ..U |
| 3 | TableLock | .IS |

**Phase 4: Completion and Cleanup**

Once catchup has completed the U lock acquired is momentarily upgraded to a Z lock. With exclusive access the old index object is switched with the newly created shadow index object. The old index object is dropped and the associated memory freed. Thus, the used tablespace memory has returned to the state it was at the start of the maintenance operation:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **TBSP\_USED\_PAGES** | **TBSP\_TOTAL\_PAGES** | **TBSP\_PENDING\_FREE\_PAGES** | **TBSP\_USED\_MB** | **TBSP\_TOTAL\_MB** | **TBSP\_PENDING\_FREE\_MB** |
| 224,000 | 270,336 | 42,784 | 875 | 1,056 | 167 |

Once the switch and free is complete locks associated with online index reorganization rebuild are released and the transaction completes. Online index reorganization rebuild has therefore successfully completed and a newly created index is present.

**Conclusions**

In this blog we reviewed index reorganization concepts and the work done for Db2 v12.1.1 to enhance index reorganization rebuild to allow read and write access in pureScale without impacting the table and index availability. We also dove into some of the design considerations and went through some usage examples so that you can see how it works and how you might use it in your own environments.

This blog highlights the advantages of using index reorganization rebuild and how it is now much more accessible for pureScale environments.

**About the Authors**

Zach Hoggard is a Senior Software Developer for IBM Db2 in the IBM Data & AI organization at the IBM Canada Lab. Zach has over 10 years of Db2 Kernel development experience and his areas of expertise are the Buffer Pool and Storage Manager components of Db2, but also has extensive experience with all Db2 Kernel components. Zach has strong interest in Db2 problem determination as well as Db2 system design in the Kernel area. Zach is an IBM Certified Database Associate for Db2 v11.1 and v10.5, and holds a Bachelor's degree in Software Engineering from Western University in London, Ontario, Canada.  
  
Chris Stojanovski is a Staff Software Developer for IBM Db2 in the IBM Data & AI organization at the IBM Canada Lab. Chris has been with IBM for five years, contributing primarily to the Index Manager team while also working with Data Management Services and XML. As a member of the Db2 Kernel team, he is passionate about expanding his knowledge of database internals and system architecture. Chris holds a Bachelor of Computer Science from the University of Waterloo, a Bachelor of Business from Wilfrid Laurier University, and a Master’s in Computer Science from Western University.