Slow Deletes Caused by Large Mapping Index involving Column Store

Last updated by | Vitor Tomaz | Jun 8, 2022 at 5:34 AM PDT

Contents

Slow deletes caused by large mapping index involving column store

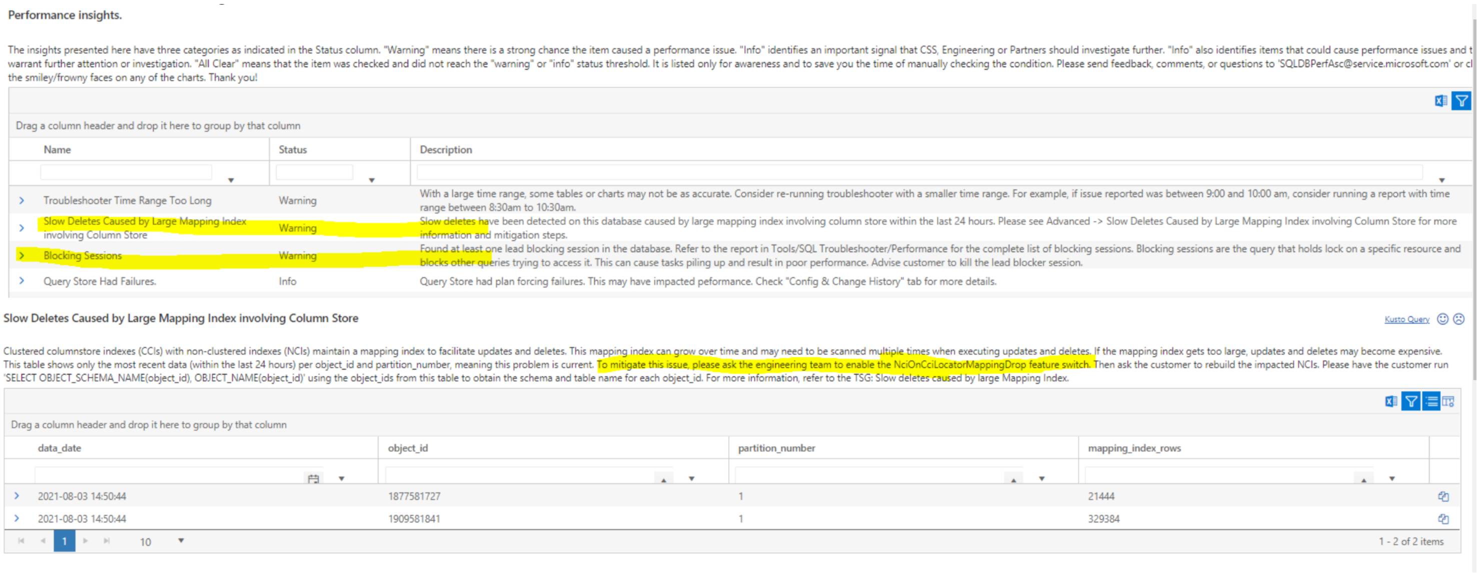
Issue

Customer experiences slowness when application delete rows in the table using column store.

Troubleshooting

ASC SQL Troubleshooter

From ASC -> SQL Troubleshooter -> Performance Insight, you can see a warning triggered for "Slow Deletes Caused by Large Mapping Index involving Column Store".



Performance insights.

The insights presented here have three categories as indicated in the Status column. "Warning" means there is a strong chance the item caused a performance issue. "Info" identifies an important signal that CSS, Engineering or Partners should investigate further. "Info" also identifies items that could cause performance issues and t warrant further attention or investigation. "All Clear" means that the item was checked and did not reach the "warning" or "info" status threshold. It is listed only for awareness and to save you the time of manually checking the condition. Please send feedback, comments, or questions to 'SQLDBPerfAsc@service.microsoft.com' or cl the smiley/frowny faces on any of the charts. Thank you!

Drag a column header and drop it here to group by that column

Name	Status	Description
Troubleshooter Time Range Too Long	Warning	With a large time range, some tables or charts may not be as accurate. Consider re-running troubleshooter with a smaller time range. For example, if issue reported was between 9:00 and 10:00 am, consider running a report with time range between 8:30am to 10:30am.
Slow Deletes Caused by Large Mapping Index involving Column Store	Warning	Slow deletes have been detected on this database caused by large mapping index involving column store within the last 24 hours. Please see Advanced -> Slow Deletes Caused by Large Mapping Index involving Column Store for more information and mitigation steps.
Blocking Sessions	Warning	Found at least one lead blocking session in the database. Refer to the report in Tools/SQL Troubleshooter/Performance for the complete list of blocking sessions. Blocking sessions are the query that holds lock on a specific resource and blocks other queries trying to access it. This can cause tasks piling up and result in poor performance. Advise customer to kill the lead blocker session.
Query Store Had Failures.	Info	Query Store had plan forcing failures. This may have impacted peformance. Check "Config & Change History" tab for more details.

Slow Deletes Caused by Large Mapping Index involving Column Store

Clustered columnstore indexes (CCIs) with non-clustered indexes (NCIs) maintain a mapping index to facilitate updates and deletes. This mapping index can grow over time and may need to be scanned multiple times when executing updates and deletes. If the mapping index gets too large, updates and deletes may become expensive. This table shows only the most recent data (within the last 24 hours) per object_id and partition_number, meaning this problem is current. To mitigate this issue, please ask the engineering team to enable the NciOnCciLocatorMappingDrop feature switch. Then ask the customer to rebuild the impacted NCIs. Please have the customer run 'SELECT OBJECT_SCHEMA_NAME(object_id), OBJECT_NAME(object_id)' using the object_ids from this table to obtain the schema and table name for each object_id. For more information, refer to the TSG: Slow deletes caused by large Mapping Index.

Drag a column header and drop it here to group by that column

data_date	object_id	partition_number	mapping_index_rows
2021-08-03 14:50:44	1877581727	1	21444
2021-08-03 14:50:44	1909581841	1	329384

Kusto query

1. If the following query does not return any results try lowering the rcrows filter, but below 1k it is almost for sure not related to mapping index. Note, this is a generic query, DB and DW specific version available below.

```
let myAppName = 'a549145a44f0';
let mappingIndexRowCounThreshold=10000;
MonDatabaseMetadata
| where table_name == 'sysrowsets'
| where AppName has myAppName
| where ownertype  == 5 and rcrows > mappingIndexRowCounThreshold
| summarize max(rcrows) by bin(TIMESTAMP, 1d), AppTypeName, AppName, LogicalServerName, logical_db_name,
```

2. SQL DB: to map objects to user facing name use this query

```
let myAppName = 'a549145a44f0';
let mappingIndexRowCountThreshold=10000;
let FilteredResults=materialize(MonDatabaseMetadata
| where AppName has myAppName
| where (table_name=='sysclsobjs' and class==50) or (table_name=='sysschobjs' and (substring(['type'],0,1
| project TIMESTAMP, table_name, class, ['type'], id, name, nsid, created, modified, indid, idmajor, idmi
let schemas=FilteredResults
| where (table_name=='sysclsobjs' and class==50)
| summarize by schema_id=id, schema_name=tolower(name);
let tables=FilteredResults
| where (table_name=='sysschobjs' and (substring(['type'],0,1)=='U' or ['type']=='ET'))
| extend is_external=iif(['type']=='ET',1,0)
| summarize by schema_id=nsid, object_id=id, table_name=name, is_external, created, modified, data_date=b
let indexes=FilteredResults
| where (table_name=='sysidxstats' and indid in (0,1))
| extend index_type_desc=iff(['type']==0, 'HEAP', iff(['type']==1, 'CLUSTERED', iff(['type']==5, 'CCI', t
| summarize by object_id=id,index_id=indid,index_type=type,index_type_desc;
let index_columns=FilteredResults
| where (table_name=='sysiscols' and idminor==1 and binary_and(status, 0x10)==0 and tinyprop2==0)
| summarize by object_id=idmajor, index_id=idminor, index_column_id=subid, column_id=intprop, key_ordinal
let partition_columns=FilteredResults
| where (table_name=='sysiscols' and idminor in (0,1) and tinyprop2<>0)
| summarize by object_id=idmajor, index_id=idminor, index_column_id=subid, column_id=intprop, partition_o
let object_PSch=FilteredResults
| where (table_name=='syssingleobjrefs' and class==7)
| summarize by object_id=depid, data_space_id=indepid;
let partition_schemes=FilteredResults
| where (table_name=='sysclsobjs' and class==31 and ['type']=='PS')
| summarize by data_space_name=name, data_space_id=id;
let PSch_PFunc=FilteredResults
| where (table_name=='syssingleobjrefs' and class==31)
| summarize by data_space_id=depid, function_id=indepid;
let partition_functions=FilteredResults
| where (table_name=='sysclsobjs' and class==30)
| summarize by function_name=name, function_id=id, boundary_value_on_right=status, partition_count=intpro
let columns=FilteredResults
| where (table_name=='syscolpars')
| summarize by object_id=id, column_id=colid, column_name=name;
let partition_details=partition_columns
| join kind=inner (object_PSch) on object_id
| join kind=inner (partition_schemes) on data_space_id
| join kind=inner (PSch_PFunc) on data_space_id
| join kind=inner (partition_functions) on function_id
| join kind=inner (columns) on object_id, column_id
| project object_id, index_id, column_id, column_name, partition_ordinal, data_space_id, data_space_name,
let clustred_index_columns=index_columns
| join kind=inner (columns) on object_id, column_id
| sort by object_id asc, index_id asc, key_ordinal asc
| summarize index_columns=makeset(todynamic(iif(is_descending_key==0,strcat('[',column_name, '] asc'),str
| summarize by object_id, index_id, index_columns=tostring(index_columns)
| project object_id, index_id, index_columns=replace(@',',', ',replace(@'^\[','',replace(@'\]$','',transl
let db_user_tables = materialize(
tables
| join kind=inner (schemas) on schema_id
| join kind=leftouter (indexes) on object_id
| join kind=leftouter (clustred_index_columns) on object_id, index_id
| join kind=leftouter (partition_details) on object_id, index_id
| project schema_id, schema_name, object_id, table_name, is_external, table_type=index_type_desc, index_c
| sort by schema_name asc, table_name asc, data_date asc);
let rowsets_with_large_mapping_indices = materialize(MonDatabaseMetadata
| where table_name == 'sysrowsets' and  AppName has myAppName
| where ownertype  == 5 and rcrows > mappingIndexRowCountThreshold
| summarize max(rcrows) by bin(TIMESTAMP, 1d), AppName, idmajor, numpart, rowsetid
| project-rename mapping_index_rows=max_rcrows, object_id=idmajor, data_date = TIMESTAMP, partition_numbe
rowsets_with_large_mapping_indices
| join kind=leftouter  (db_user_tables) on object_id, data_date
| project-away data_date1, object_id1, is_external, table_type
| project AppName,object_id, rowsetid, partition_number, mapping_index_rows, data_date, schema_name=iff(i
```

3. SQLDW query

```
let myAppName = 'fed802b75e5c';
let mappingIndexRowCounThreshold=10000;
let problematic_objects=materialize(
MonDatabaseMetadata
| where AppName has myAppName
| where table_name == 'sysrowsets'
| where ownertype  == 5 and rcrows > mappingIndexRowCounThreshold
| summarize max(rcrows) by bin(TIMESTAMP, 1d), AppTypeName, AppName, LogicalServerName, logical_db_name,
problematic_objects
| join kind=leftouter  (
    dw_user_table_mappings(myAppName)
    | extend logical_db_name = strcat("Distribution_", distribution)
) on $left.logical_db_name == $right.logical_db_name and $left.idmajor == $right.physical_object_id
| summarize sum(max_rcrows) by object_id, schema_name, table_name, TIMESTAMP
```

Root cause

The mapping index has an inherent cost, because we need to map rows from NCI to CCI and we need to maintain the mapping index, when a row is deleted. Performance can significantly degrade, because the iterator model requires a row to be deleted first in the base index and it's locator to be returned to the NCI so we can delete it from there too. To find the locators for each delete we open the Mapping Index at the beginning of the delete and we keep a cursor at the last row used to look up the original locator of the deleted using the mapping index, while we can move the cursor in only one direction. Thus if your next row to be delete is above your current cursor position you'll scan until the end, then restart from the beginning. This leads to (# mapping index entry count) x (# rows to deleted) worst case runtime as opposed to the best case of (#rows to delete).

Mitigation

- Make sure FS NciOnCciLocatorMappingDrop is enabled
- The customer can self-diagnose this issue. select object_id ,partition_id ,OBJECT_NAME(object_id) as Table ,partition_number ,rows as MappingIndexRowCount from sys.internal_partitions where internal_object_type_desc = 'COLUMN_STORE_MAPPING_INDEX' and rows > 1000 -- this threshold isn't an absolute number; the higher the rowcount, the more likely the problem will be seen
- You can also tell how many scans did a delete cost (look for scan range count):

```
SELECT op.* FROM sys.dm_db_index_operational_stats(db_id(), object_id('t'), NULL, NULL) op JOIN sys.inter
```

- Depending on what's the goal of the customer is, there are two options - please see FAQ for whys:
    - If customer want to get rid of the mapping index for good:
        - Drop all NCI's

    - Rebuild the CCI on the table (partitioned rebuild will not remove the mapping index for good, see FAQ). Works WITH (ONLINE = ON) option as well. OR
    - Create a new partition aligned empty table that doesn't have an NCI. Let's call it TargetTable.
    - Switch partitions from the SourceTable to TargetTable, as part of the switch partition we'll drop the existing Original Locator column and the Mapping Index, see CIndexDDL::SwitchRowsets. This for large tables this will be significantly faster thant the CCI rebuild option.
  - If customer want to manage the mapping index size, since they want to keep using NCIs:
    - Rebuild the CCI per partition, starting with the ones with the highest rowcount from the query above OR
    - For CCIs with no existing NCI, create and drop a NCI to trigger the mapping drop
    - For CCIs with existing NCIs that are required, re-create the NCI - note that online NCI build and rebuild will not drop the mapping index.

Additional Information

What is mapping index?

Every nonclustered index must be able to map its rows back to the base index in case some columns are only available in the base index or for queries to satisfy DMLs - i.e. you need to delete rows from base and non-clustered indexes. In Btrees we store the base index key next to each nonclustered index key, but for columnstore the base index (rowgroups) can change. A new immutable column, called original locator was introduced to designate the location where a row was first inserted, this column value is stored next to the NCI index's key column(s) and we use it to look up rows in the columnstore index. To track original locators when rows move between row groups (TM compresses delta store, merge by TM or REORG) we need an intermediate mapping structure, called the mapping index.

Tuple Mover and REORGANIZE contribute to the growth of the mapping index because they add rows to the mapping index, the gist of it is: we track continuous ranges of row movements from one rowgroup to another (so we don't need to map every row), thus running reorganize will create as many entries in the mapping index as many holes your delete created in a rowgroup.

Related blog:http://www.nikoport.com/2015/09/06/columnstore-indexes-part-65-clustered-columnstore-improvements-in-sql-server-2016/

FAQ

- Q: When all the NCIs are dropped do we still use the mapping index? A: Yes, as long as a mapping index is present we will maintain the mapping index. Given that in the presence of NCI we first must delete rows from CCI we will always try to return the row handle so it can be deleted from the NCI(s) as well.
- Q: Is it applicable to Nonclustered columnstore indexes? A: No
- Q: Can I tell how many times will the mapping index be scanned for a delete, so I can estimate the cost? A: It's quite hard to tell upfront, because it depends on the order of how we find rows to be deleted. You can however tell the cost post delete, by consulting the sys.dm_db_index_operational_stats

- Q: Are there any negative effects of having NciOnCciLocatorMappingDrop enabled? A: This FS will ensure that we drop the mapping index and the original locator column when the NCI is created. Naturally, after dropping these constructs we will create a new mapping index and original locator. Even though the original locator has default (null) values, hence we can take advantage of default column optimizations, it still has some cost, since we need to build and persist these column segments for each rowgroup. The cost of adding and dropping the mapping index rowset is negligible.

- Q: How can I get rid of the mapping index for good? A: You need to rebuild the CCI on all partitions on the table (ALTER INDEX REBUILD ...), rebuilding one partition will remove the mapping index on that partition, but we still keep the original locator column in the table definition, so on first row movement between rowgroups we'll create a new mapping index. And alternative way is to create a new table with the same partition schema as the original table without an NCI, then switch partitions to the new table, at this point the partition switch logic will drop the original locator and the mapping index.

- Q: When is the mapping index added? A: We create the mapping index rowset lazily, the first time we need to insert a row into it. Thus after applying the mitigations below you might see the mapping index gone, then come back.

How good have you found this content?

🙂 🙁