Are deleted rows wasting resources on your IBM i system?

A task for database engineers on IBM DB2 for i

Tom McKinley November 06, 2014

Historically, IBM i systems have not had a database administrator or engineer. This reduced focus on the database can often lead to very inefficient use of system resources. This article explains how to address one of the key performance issues, *deleted row space*, caused by a lack of focus on database management. You can learn how to identify tables that have lots of deleted row which causes system resources to be wasted on specific IBM® DB2® access methods. The article also describes the steps to be taken to reduce the deleted row space and reduce system resource usage.

Introduction

Do you have an IBM DB2 table with a lot of deleted rows in it? Though this seems to be a basic question, the interesting thing is that many IBM i shops don't know the answer. Typically, most IBM i shops don't have a database administrator to watch these types of details. The worst I have seen during one of my customer engagements is a table with 2.6 billion (yes, that is billion) deleted rows. And that table was being scanned frequently. Not only are many customers not sure how many deleted rows there are in their database, but they also don't understand the impact those deleted rows are having.

Let's see if we can help you answer two questions:

1. What tables have a large number of deleted rows?
2. Are those deleted rows wasting processor and memory resources?

Of course, the deleted rows are taking up disk space, but how much? We also want to understand if those deleted rows are causing extra disk operations, taking up room in memory, and wasting processor utilization on scan operations. It might also be good to know if the storage consumed by the deleted rows is being reused when new rows are added to a table in the future.

First, let's find the DB2 tables and physical files that have the largest number of deleted rows in them. Before you start this analysis, you need to perform the following data collection for one

or more of your main production data libraries. Create a library for this collection data with the following command:

```
CRTLIB DELETDROWS
```

Run the following CL commands for each library (replace `LIBNAME` with your data library name or `*ALLUSR`). These commands collect information about the files in the named library.

```
DSPFD FILE(LIBNAME/*ALL) TYPE(*MBR) OUTPUT(*OUTFILE) FILEATR(*PF)
OUTFILE(DELETDROWS/DSPFD_MBR) OUTMBR(*FIRST *ADD)

DSPFD FILE(LIBNAME/*ALL) TYPE(*ATR) OUTPUT(*OUTFILE) FILEATR(*PF)
OUTFILE(DELETDROWS/DSPFD_ATR) OUTMBR(*FIRST *ADD)
```

Create a couple of indexes over these two newly created out files to improve the performance of your analysis.

```
CREATE INDEX DELETDROWS.DSPFD_ATR_IX ON DELETDROWS.DSPFD_ATR(PHFILE);
CREATE INDEX DELETDROWS.DSPFD_MBR_IX
  ON DELETDROWS.DSPFD_MBR(MBFTYP, MBFILE, MBNDTR);
```

Run the following procedure call from any SQL interface. This stored procedure dumps SQL plan cache entries to a table named PCSS in the DELETDROWS library.

```
CALL QSYS2.DUMP_PLAN_CACHE('DELETDROWS', 'PCSS')
```

Step 1 – Identify tables that have a large number of deleted rows

The following SQL query returns the top 25 tables with the largest number of deleted rows. If you see tables in the query output that you know are used heavily, consider reorganizing them to eliminate the deleted rows.

```
SELECT DISTINCT F.MBFILE AS FILENAME,
            F.MBNAME as Member,
            MBLIB AS LIBRARY,
            MBNRCD AS "Number Non-Deleted Rows",
            MBNDTR AS "Number Deleted Rows",
            PHRUSE AS "Reusing Deleted Rows",
            integer(mbndtr/(Case when mbnrcd+mbndtr=0 then 1
            else mbnrcd+mbndtr end) * 100) as "Percent Deleted",
            Case when MBDSSZ >0 then MBDSSZ ELSE MBDSZ2 END as Size,
            Integer(Case when MBDSSZ >0 then MBDSSZ ELSE MBDSZ2 END *
            (mbndtr/(Case when mbnrcd+mbndtr=0 then 1
            else mbnrcd+mbndtr end))) as "Deleted Space"

  FROM DELETDROWS.DSPFD_MBR AS F JOIN DELETDROWS.DSPFD_ATR AS A
    ON F.MBFILE = A.PHFILE
  WHERE MBFTYP = 'P' AND MBNDTR > 10000
    ORDER BY MBNDTR DESC FETCH FIRST 25 ROWS ONLY
```

You can modify this query to look for tables that exceed some threshold of deleted rows. In this example, the threshold is set to 10,000 rows(`mbndtr` > 10000). You can also sort the results by the **Percent Deleted** value to list tables with the highest percentage of deleted rows at the top.

Figure 1 – Example output from step 1 query

FILENAME	MEMBER	LIBRARY	Number Non-Deleted Rows	Number Deleted Rows	Reusing Deleted Rows	Percent Deleted	SIZE	Deleted Space
DELTEST	DELTEST	MCKINLEY2	3,700,000	4,300,000	Y	53	392,011,776	210,706,329
ITEM_FACT	ITEM_FACT	PURGETEST	3,568,632	2,432,583	Y	40	1,612,763,136	653,730,984
QAPZREQ	QAPZREQ	QUSRSYS	44,236	121,803	N	73	16,781,312	12,310,446
QAPZSYM	QAPZSYM	QUSRSYS	43,824	118,801	N	73	50,335,744	36,771,324
SELECT1P	TEST1	MCKINLEY2	550,000	50,000	Y	8	7,221,248	601,770
QAPZGRP	QAPZGRP	QUSRSYS	8,212	2,389	N	22	3,690,496	831,675
QAPZPTF	QAPZPTF	QUSRSYS	11,308	361	N	3	1,581,056	48,912
QAPMCCCNAT	QAPMCCCNA	QUSRSYS	37	299	Y	88	217,088	193,182
CLASSDBMON	CLASSDBMOI	DENTON	430,959	256	Y	0	1,122,041,856	666,124
KENTJOB	KENTJOB	DENTON	271	126	Y	31	3,309,568	1,050,391
QZG0000777	KENTJOB	DBQTEAM03	271	126	Y	31	3,145,728	998,392
QZG0000777	KENTJOB	DBQTEAM04	271	126	Y	31	3,145,728	998,392
QZG0000777	KENTJOB	DBQTEAM05	271	126	Y	31	3,145,728	998,392
QZG0000777	KENTJOB	DBQTEAM06	271	126	Y	31	3,145,728	998,392
QZG0000777	KENTJOB	DBQTEAM07	271	126	Y	31	3,145,728	998,392
QZG0000777	KENTJOB	DBQTEAM08	271	126	Y	31	3,145,728	998,392
QZG0000777	KENTJOB	DBQTEAM09	271	126	Y	31	3,145,728	998,392
QALZALIC	QALZALIC	QUSRSYS	143	105	N	42	32,768	13,873

Step 2 – Look for SQL statements that are wasting resources

Some SQL statements consume more system resources to process deleted rows based on access methods used in the runtime implementation. This section focuses on a couple of the methods that are more expensive because of the deleted rows in the table.

SQL statements performing table scans

Table scan operations are clearly more expensive because of the deleted rows. Each deleted row is accessed as DB2 scans the table to find the non-deleted rows that meet the search criteria. Even if more advanced methods are used to skip the rows that do not meet the search criteria, it is likely that there are more disk I/O operations required. More operations are needed because in general, the pages in the table are more sparsely populated with non-deleted rows.

The following SQL query identifies the tables that were the target of a table scan and accounts for the largest total of deleted rows processed.

```
with bigdel as (
      SELECT  distinct
             mbfile    as File,
             mbname as Member,
             mblib     as Library,
             mbnrcd  as NumberRecords,
             mbndtr   as NumberDeletedRecords,
             phruse   as ReuseDeleted
```

```
      FROM deletdrows.DSPFD_MBR F
           join deletdrows.DSPFD_ATR a on f.MBFILE=a.PHFILE
     WHERE mbftyp ='P' AND mbndtr >10000 )
-- Join table scan info from the plan cache
SELECT QQTLN as LIBrary, QQTFN as "Table Name", D.REUSEDELETED as
"Reusing Deleted Rows" , Max(QQTOTR )as "Number Non-Deleted Rows",
max(d.numberDeletedRecords) as "Number Deleted Rows" ,
SUM(d.numberDeletedRecords) as "Total Deleted Rows Scanned",
count(*) as TotalScans
FROM deletdrows.PCSS M JOIN bigdel d
  ON d.mblib=m.qqtln AND d.mbfile=m.qqtfn AND d.Member=m.qqtmn
WHERE qqrid = 3000 AND QQC11 <>'Y'
GROUP BY qqtln, qqtfn, d.reusedeleted
ORDERY BY  SUM(d.numberDeletedRecords) DESC
FETCH FIRST 25 ROWS ONLY OPTIMIZE FOR ALL ROWS
```

It is outside the scope of this article, but you should look for the SQL statements that are causing these table scan operations and see if they creating additional indexes can eliminate the table scan method from being used. Still, that does not remove the need to reduce the size of these very frequently used DB2 tables in your database. As you can see in Figure 1, each table scan of DELTEST processes 210 MB of storage occupied by deleted rows.

Figure 2 – Example output from Step 2 query

LIBRARY	Table Name	Reusing Deleted Rows	Number non-Deleted Rows	Number Deleted Rows	Total Deleted Rows Scanned	Total Scans
MCKINLEY2	DELTEST	Y	3,700,000	4,300,000	18,434,100,000	4,287

SQL statements performing temporary index builds

Index builds have to perform more I/O operations to bring in storage pages with deleted rows only to skip over the deleted rows during the creation of the index. Some of these index builds might occur over small tables, but creating a temporary index many times can amplify the expense of those deleted rows.

The following SQL query identifies the SQL statements that involve temporary index builds over tables having a large number of deleted rows (greater than 10000).

```
with bigdel as (
Select  mbfile, mblib,F.MBNAME as Member, mbftyp, MBNRCD as
NumberRecords, MBNDTR as numberDeletedRecords,
  PHRUSE as REUSEDELETED,
  MBDSSZ as MBRSIZE
  FROM deletdrows.DSPFD_MBR F join deletdrows.DSPFD_ATR a
  ON f.MBFILE=a.PHFILE
WHERE MBFTYP ='P' AND mbndtr >10000)
-- Join Index builds to the tables with large number of deleted rows
Select QQTLN as Library, QQTFN as "Table Name", D.REUSEDELETED as
"Reusing Delweted Rows", Max(QQTOTR )as "Non-Deleted Rows",
max(d.numberDeletedRecords) as "Deleted Rows", SUM(d.numberDeletedRecords) as
"Total Deleted Rows Scanned", SUM(M.QQRIDX) as "Total index entries created",
QQIDXD as Index_Advised_Columns,
Sum(case when QQC16='N' then qqi6 else 0 end) as total_keys_built,
Sum(case when QQC16='N' then 1 else 0 end ) as indexCreated,
Sum(case when QQC16='Y' then 1 else 0 end ) as indexreused,
```

```
Count(*) as TotalIXsCreated
FROM deletdrows.PCSS M JOIN bigdel d
  ON d.mblib=m.qqtln and d.mbfile=m.qqtfn and d.Member= m.qqtmn
WHERE qqrid = 3002
GROUP BY qqtln, qqtfn ,qqtmn, d.REUSEDELETED, m.QQRCOD , QQIDXD
ORDER BY SUM(d.numberDeletedRecords)DESC
FETCH FIRST 25 ROWS ONLY OPTIMIZE FOR ALL ROWS
```

The output from this query helps to identify those tables that are good candidates to be reorganized. An additional benefit of this report is that this can also be a good way to see some indexes that might need to be created to eliminate the temporary index builds.

Step 3 – Look at highly accessed tables

If you are running IBM i 7.1 or later versions on your system, you can use the new access counters to identify tables that are frequently accessed. These counters are automatically incremented by DB2 for both SQL and non-SQL interfaces. To identify the more highly accessed DB2 tables, run the following query and look for tables that have a high count of deleted rows.

```
SELECT table_schema, Table_name, Data_size, Number_Deleted_Rows,
Logical_Reads, Physical_reads, Sequential_reads, Random_reads
FROM qsys2.systablestat
ORDER BY Logical_reads DESC  FETCH FIRST 25 rows ONLY
```

In this example, the query output in Figure 3 shows only one table, DELTEST, that has a significant number of deleted rows.

Figure 3 – Example result from a highly accessed table query

SCHEMA	TABLE_NAME	DATA SIZE	NUMBER DELETED ROWS	LOGICAL READS	PHYSICAL READS	SEQUENTIAL READS	RANDOM READS
lib1	P_GHDBC1AP	3,269,562,368	0	39,900,656,317	433,691	46,790,480,530	1,844,127
lib2	PCSNAPSHTA	1,558,278,144	0	9,304,974,458	152,204	9,964,856,190	2,763,927
lib4	QAPMJOBMI	505,421,824	0	6,353,286,018	77,577	44,771,132	6,318,137,958
LST	PCSNAPSHTA	7,396,790,272	0	5,677,495,916	2,279,696	5,625,878,765	22,029,410
QBENA	PCSNAPSHTA	547,446,784	0	2,232,019,202	35,158	2,230,301,674	1,717,560
QIBMDB2OLD	PNCDBMON2	2,086,768,640	0	2,119,777,056	747,709	2,113,766,480	1,094,904
lib5	PCSNAPSHAI	411,131,904	0	2,007,957,207	58,466	2,004,946,526	2,469,189
lib6	PCSNAPSHBI	419,520,512	0	1,240,550,047	65,244	1,239,118,848	661,150
lib127	P_DRRDG	348,217,344	0	683,868,040	26,158	683,126,680	269,308
lib5	DBWIZEMON	130,113,536	0	555,389,777	8,302	554,785,300	332,238
lib5	QAPMJOBMI	14,164,201,472	0	472,206,401	1,400,034	449,789,044	22,417,357
Lib270	P_DRGH	304,177,152	0	408,329,291	18,453	407,742,720	181,145
LIB270	P_GHFIELD	295,788,544	0	391,143,424	28,007	390,609,005	135,664
lib2	PCSNAPSHTA	304,177,152	0	304,559,247	20,740	302,861,827	1,346,458
lib2	DBWIZEMON	249,651,200	0	287,803,295	28,773	287,034,476	425,945
lib4	QAPMJOBOS	5,926,572,032	0	203,388,552	465,419	111,958,801	91,446,423
MCKINLEY2	DELTEST	392,011,776	4,300,000	192,028,663	6,066	192,023,207	5,456
DLT	COLUMNINFO	15,907,110,912	0	173,894,575	596,960	173,894,575	0
DELETDROWS	PCSS	186,736,640	0	145,870,203	5,808	145,443,854	171,703
lib4	QAPMJOBWT	4,500,504,576	0	107,827,463	115,542	65,803,729	42,023,734
LST	QAPMJOBMI	3,827,314,688	0	103,728,920	589,972	93,474,880	10,254,296
LST	QAIDRCSTL_Q254(	555,757,568	0	42,322,734	39,295	42,322,734	0
lib4	QAIDRCSTL_Q228(	488,648,704	0	40,299,150	36,335	40,299,150	0
lib2	QAPMJOBOS	387,981,312	0	35,854,917	48,721	27,012,604	8,088,137
DBQTEAM09	ORDERS	169,881,600	137	35,815,667	1,412	30,030,530	380,213

Step 4 – Fix the problematic tables

To eliminate the deleted rows from the tables identified by the previous analysis methods, we need to use the Reorganize Physical File Member (RGZPFM) system command. However, there are some important points to consider before performing a reorganize operation. Some applications might fail to work properly if one or more of the tables they use are reorganized. There are two reasons that this might occur:

1. The application has a dependency on the physical location of some rows. You would have to obtain the Relative Record Number (RRN) of a specific row and store the value else where so that you can use the value to access the row later. You might need it for some native record-level access request or the SQL RRN function. This case is unlikely, but it is a possibility. However, this is **not** a recommended programming practice.
2. The application might have a dependency on the rows being stored in the arrival sequence. That is, a row inserted after another row will have a higher RRN value.

The other key consideration is the file attribute that controls whether deleted record space is reused for new rows being inserted into the table. If a file is reusing deleted rows, the applications cannot have the arrival sequence dependency unless rows are never deleted.

There are two basic options for reorganizing. The first option is to slide all the rows up filling the deleted holes and then truncate the set of deleted rows at the end. Performing the reorganize in this manner maintains the arrival sequence, and therefore, applications with an arrival order sequence dependency continue to function.

The second option is to move the non-deleted rows from the end of table to the deleted row locations at the beginning of the table. This method breaks the arrival order sequence of the rows, but usually performs much better than the first method. For more details on the RGZPFM command, refer to the IBM Redpaper™ listed in the Resources section.

Monitoring and managing deleted row space is one of the key activities that a DB2 for i database engineer should be performing on a regular basis. If you do not have a DB2 for i database engineer, you can read the blog entry to better understand the role and responsibilities of this position.

Hopefully, this article might have given you some motivation to start paying attention to deleted row space and allow you to take focused action to minimize the system resources being wasted on deleted rows.

You can also use the Phase 2 System Limits support to track changes in your biggest files, including delete operations. Refer to the Gain Big Insights into DB2 for i with System Limits, Phase 2 article for more information about System Limits support.

Resources

- IBM i Reorganize Physical File Member Redpaper
- DB2 for i Center of Excellence Services
- DB2 for i blog
- DB2 for i SQL Performance Workshop