Get Up to Speed on SQL's Fast Delete in Db2 for i

Scott Forstie
November, 2021

Diagnose and optimize fast delete using SQL on Db2 for i

Fast and slow are subjective terms. As an avid runner, I sometimes feel like I'm running fast until a youngster breezes by me. If only I could explain that they should fully consider my age and other factors before they give me that look that screams "move over, slowpoke." It's with this awareness that I stride into the topic of fast delete using SQL on Db2 for i.

Db2 for i has supported fast delete for SQL users since IBM i 5.3. Fast delete occurs when the DELETE SQL statement is coded to delete all rows within the table and certain conditions are met. When these conditions are met, the delete is implemented with the same support used by the Clear Physical File Member (CLRPFM) command. The performance difference between a fast and slow delete can be dramatic. As with most performance issues, however, the difference between fast and slow deletes depends on many factors. Fast deletes are more important in tables that contain more rows.

Even though support for fast deletes has existed for a long time, many users weren't aware of the conditions placed on fast deletes. When customers unexpectedly encountered slow deletes, they didn't know how to determine the cause. Recent improvements to IBM i 6.1 and 7.1 clarified the fast delete rules and instrumentation within the database monitor facility. This article explains those rules and how to use monitoring enhancements to diagnose DELETE performance issues.

In this article, a fast delete refers to a DELETE SQL statement that can be implemented by the database in one fell swoop through the file. A slow delete refers to every other successful execution of a DELETE SQL statement, where the delete processing is implemented row by row. So when I use the terms fast and slow delete, I'm simply distinguishing the type of implementation used by the database. The so-called slow delete can actually complete very quickly.

Query Option Controls

Before I delve into the fast delete enhancements, I need to cover a mechanism that will influence fast delete behavior, the SQL_FAST_DELETE_ROW_COUNT query option. Database technologies typically include such performance controls. Consider the OPTIMIZE FOR n ROWS optional clause, which can be incorporated into individual SQL query-based statements. Other controls come outside of the application, scoping to such things as the session, job, connection, or entire database. Db2 for i provides many external controls via the QAQQINI query options file, including a fast delete control mechanism.

Figure 1 defines how you can use the SQL_FAST_DELETE_ROW_COUNT query option. You can use this to direct the database to never attempt to fast delete or, more likely, to set the minimum number of rows that must exist in a table for fast delete to be considered. When this support was added, we determined that 1,000 rows was a reasonable break-even point. By default, deletes against tables with fewer than 1,000 rows will not even attempt to use fast delete.

Figure 1: Using SQL_FAST_DELETE_ROW_COUNT QAQQINI query option

Parameter	Value	Description
SQL_FAST_DELETE_ROW_COUNT Specifies how the database manager implements the delete. This value is used when processing a DELETE FROM table-name SQL statement without a WHERE clause.	*DEFAULT	Indicates that the database manager chooses how many rows to consider when determining whether to attempt fast delete. When using the default value, the database manager will most likely use 1,000 as a minimum row count.
	*NONE	This value prevents the database manager from attempting to fast delete, without regard for the table row count.
	*OPTIMIZE	This value is the same as using *DEFAULT.
	Integer Value	Specifying a value for this option lets the user tune the behavior of DELETE. For fast delete to be attempted, the target table for the DELETE statement must match or exceed the number of rows specified on the option. A fast delete does not write individual rows into a journal. Valid values are 1–999,999,999,999,999.

Slow Delete Example

The SQL script shown below takes the QAQQINI support for a test drive, demonstrating how to use the Start Database Monitor (STRDBMON) command to collect and observe the fast delete reason code. This script should work as-is on any IBM i 6.1 or 7.1 machine that meets the service-level requirements defined in the box at the end of this article (see "Service Level"). You can execute the script using System i Navigator's Run SQL Scripts or the Run SQL Statements (RUNSQLSTM) command.

Instead of relying on the Change Query Attributes (CHGQRYA) command, this example takes advantage of the QSYS2.OVERRIDE_QAQQINI() procedure to use different SQL_FAST_DELETE_ROW_COUNT values within a single workstream. Explanations follow each part of the script.

```
CL: CRTLIB DELETLIB; CL: STRDBMON QGPL/SLOWDLT; CALL
QSYS2.OVERRIDE_QAQQINI(1, '', ''); CALL QSYS2.OVERRIDE_QAQQINI(2,
'SQL_FAST_DELETE_ROW_COUNT', '*NONE');
```

I overrode the default setting and instruct the database to disallow fast deletes.

```
DROP TABLE DELETLIB.SYSIXADV; CREATE TABLE DELETLIB.SYSIXADV AS (SELECT *
FROM QSYS2.SYSIXADV) WITH DATA; DELETE FROM DELETLIB.SYSIXADV;
```

Slow delete occurred because fast delete was disabled.

```
INSERT INTO DELETLIB.SYSIXADV SELECT * FROM QSYS2.SYSIXADV; CALL
QSYS2.OVERRIDE_QAQQINI(2, 'SQL_FAST_DELETE_ROW_COUNT', '999999999'); DELETE
FROM DELETLIB.SYSIXADV;
```

Fast delete is enabled, but has a very large minimum row count value. Fast deletes will be attempted only if the table's row count exceeds 999,999,999.

```
CALL QSYS2.OVERRIDE_QAQQINI(2, 'SQL_FAST_DELETE_ROW_COUNT', '*DEFAULT');
INSERT INTO DELETLIB.SYSIXADV SELECT * FROM QSYS2.SYSIXADV; DELETE FROM
DELETLIB.SYSIXADV WHERE 1=1;
```

Slow delete occurred because a WHERE clause was used. It doesn't matter that the clause would have found every row in the table.

```
CREATE OR REPLACE VIEW DELETLIB.SYSIXADV_VIEW AS SELECT * FROM
DELETLIB.SYSIXADV; INSERT INTO DELETLIB.SYSIXADV_VIEW SELECT * FROM
QSYS2.SYSIXADV; DELETE FROM DELETLIB.SYSIXADV_VIEW;
```

SQL views can be targets of inserts, updates, and deletes. In this step, the slow delete occurred because the target of the DELETE was a view.

```
INSERT INTO DELETLIB.SYSIXADV SELECT * FROM QSYS2.SYSIXADV; CREATE OR REPLACE
TRIGGER DELETLIB.ADVTRIG1 AFTER UPDATE ON DELETLIB.SYSIXADV REFERENCING
OLD_TABLE AS OLD_T NEW_TABLE AS NEW_T NEW AS NEW_ROW FOR EACH ROW MODE DB2SQL
BEGIN DECLARE TEMP_VAR BIGINT; IF (NEW_ROW.TIMES_ADVISED = 0) THEN SET
TEMP_VAR = NEW_ROW.TIMES_ADVISED; END IF; END; DELETE FROM DELETLIB.SYSIXADV;
```

Slow delete occurred because a trigger existed over the target table. The type of trigger is irrelevant to this check.

```
DROP TRIGGER DELETLIB. ADVTRIG1; INSERT INTO DELETLIB.SYSIXADV SELECT * FROM
QSYS2.SYSIXADV; ALTER TABLE DELETLIB.SYSIXADV ADD PARTITION BY
RANGE(TIMES_ADVISED) ( STARTING FROM(1) ENDING(9999999999) EVERY(1000000000)
); CREATE OR REPLACE ALIAS DELETLIB.SYSPART1 FOR
DELETLIB.SYSIXADV(PART000001); DELETE FROM DELETLIB.SYSPART1;
```

Slow delete occurred because the target of the DELETE was an alias referring to a partitioned table member.

```
CL: ENDDBMON;
```

SQL Performance Monitor

You can examine the monitor data collected in the example above through direct SQL queries or by using the System i Navigator SQL Performance Monitor's analyze feature. I recommend using the IBM-supplied graphical tools in cases where the monitor file has a reasonably limited number of rows within it. The best practice for limiting database monitor size is to use the monitor pre-filters.

Figure 2 provides a simple query to connect the dots between the fast delete reason code and what it indicates. Figure 3 shows the query result. I structured this example to show some slow delete scenarios but no successful fast deletes. Slow deletes might happen because of a table structure, the style of SQL DELETE statement used, or an environmental reason.

Figure 2: Querying database monitor for fast delete reason code

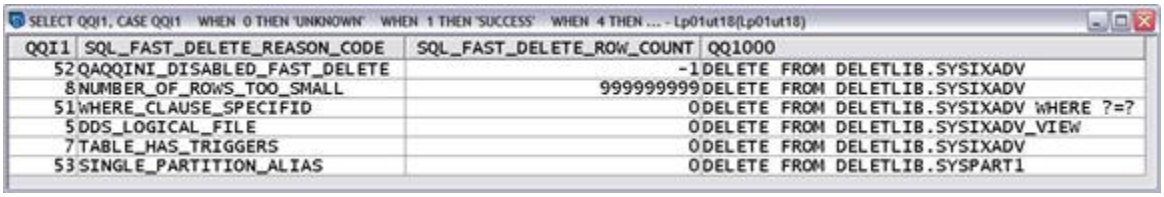


Figure 3: Querying database monitor fast delete reason code

```
SELECT QQI1, CASE QQI1 WHEN 0 THEN 'UNKNOWN' WHEN 1 THEN 'SUCCESS' WHEN 4
THEN 'DISTRIBUTED_TABLE' WHEN 5 THEN 'DDS_LOGICAL_FILE' WHEN 6 THEN
'REFERENTIAL_INTEGRITY_PARENT' WHEN 7 THEN 'TABLE_HAS_TRIGGERS' WHEN 8 THEN
'NUMBER_OF_ROWS_TOO_SMALL' WHEN 9 THEN 'CONCURRENT_OPERATION_CONFLICT' WHEN
10 THEN 'LOCK_CONFLICT_ON_TABLE' WHEN 11 THEN 'LOCK_CONFLICT_ON_DATA_SPACE'
WHEN 51 THEN 'WHERE_CLAUSE_SPECIFID' WHEN 52 THEN
'QAQQINI_DISABLED_FAST_DELETE' WHEN 53 THEN 'SINGLE_PARTITION_ALIAS' ELSE
CHAR(QQI1) END AS "SQL_FAST_DELETE_REASON_CODE", QVP151 AS
```

```
"SQL_FAST_DELETE_ROW_COUNT", QQ1000 FROM QGPL.SLOWDLT WHERE QQRID = 1000 AND
QQC21 = 'DL'
```

Figures 4 and 5 show how to drill down into the fast-delete reason code information using the SQL Performance Monitor. The first time you drill into statement-level detail for DELETE statements, you'll see every possible column. Figure 5 shows a limited number of columns. If you look under the View pull-down menu, you can limit the number of columns and set their order by using the Columns option. Limiting the columns will provide direct control over the columns and improve the performance of the Navigator session because the host will send only the data you want to see.It might seem odd that the reason codes have a gap between 11 and 51, but this was an intentional design choice. The lower-numbered reasons are provided by different parts of the database than the higher-numbered reasons.

Figure 4: Accessing delete detail using SQL Performance Monitor

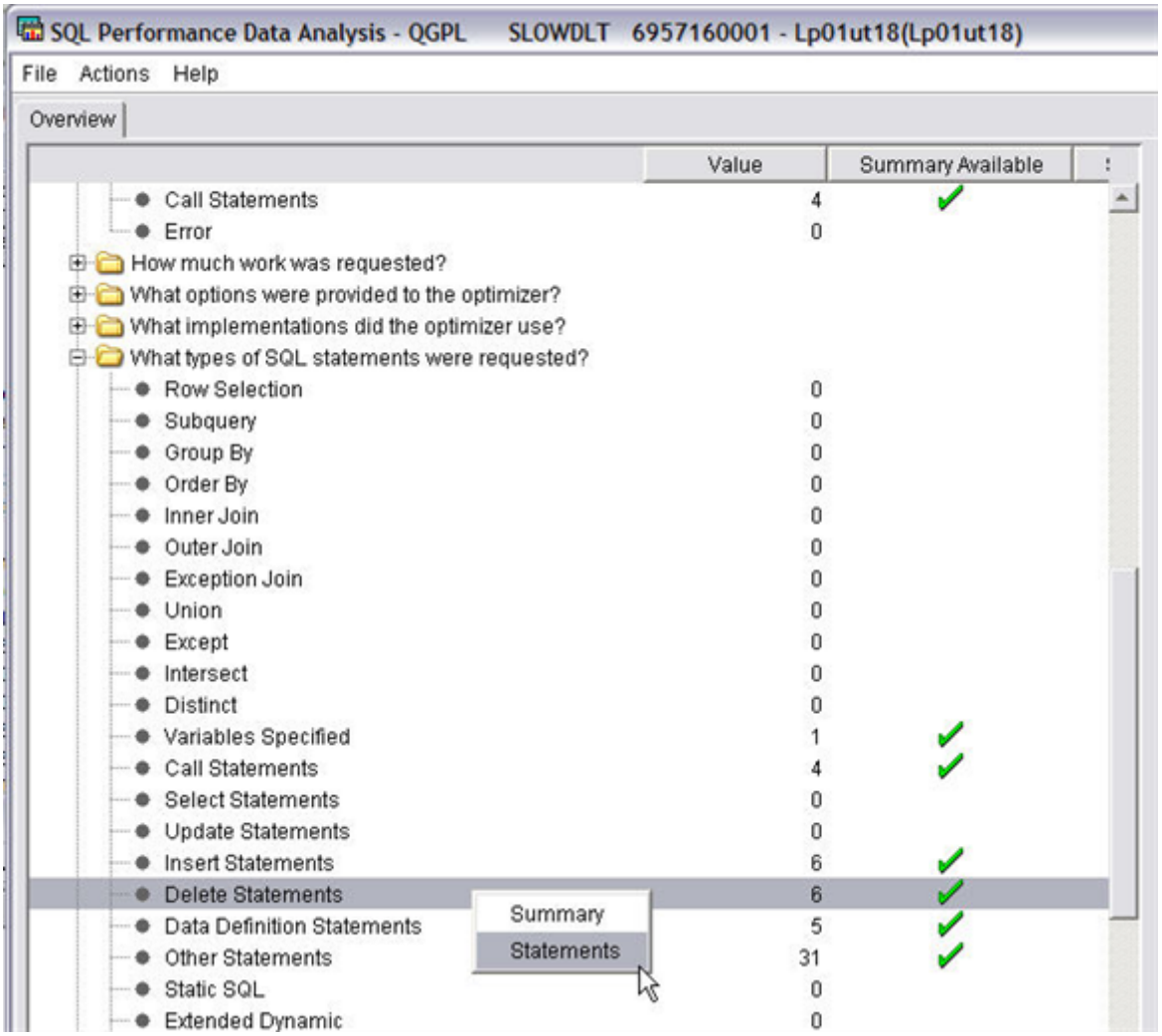
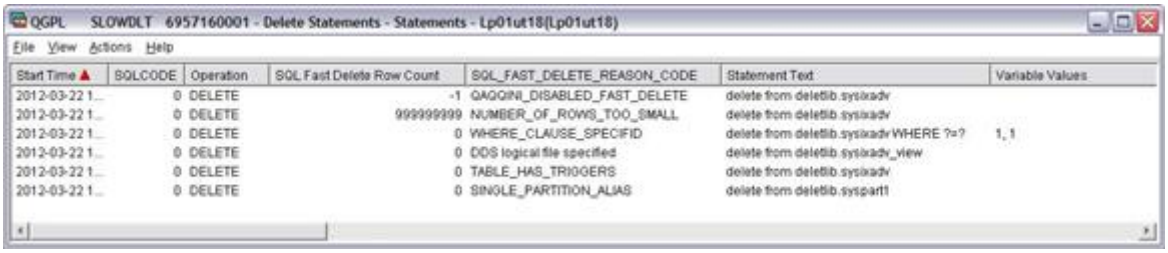


Figure 5: Fast delete results in System i Navigator



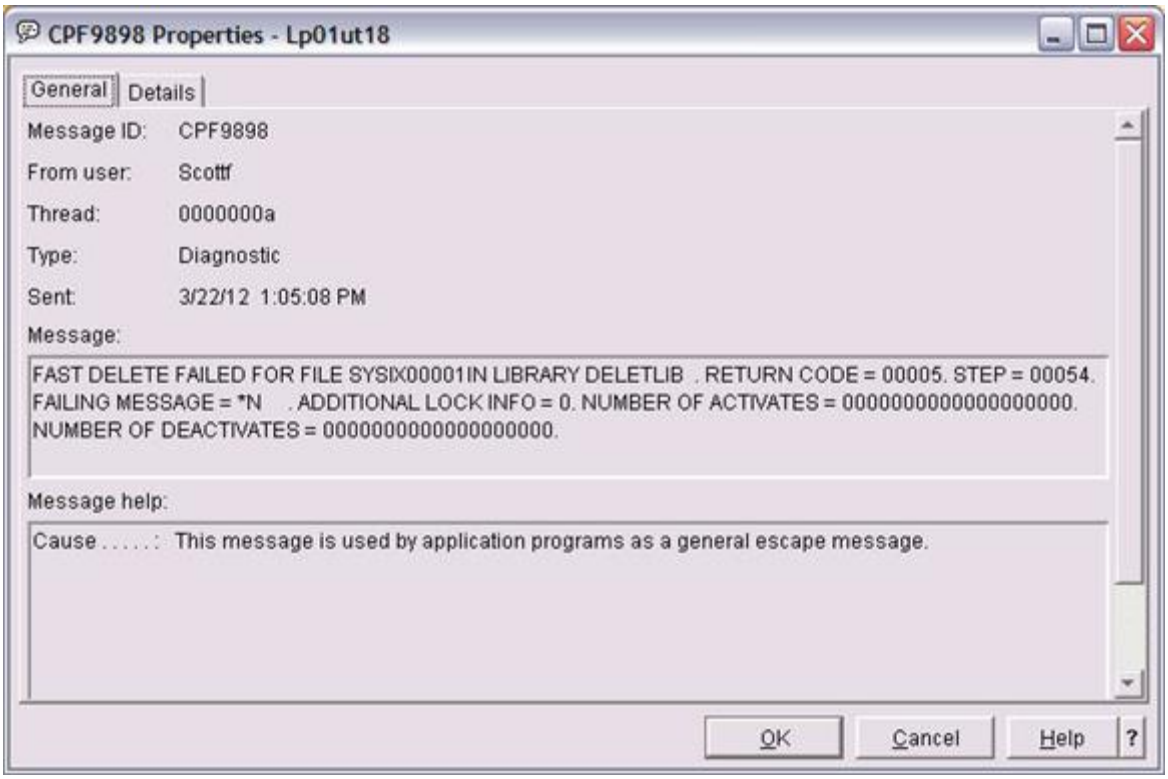
Helpful Job Log Messages

In addition to the database monitor instrumentation, a diagnostic message might be sent to the job log and to the machine's history log when a slow delete occurs under certain conditions. If a slow delete happens with a reason code between 2 and 12, a CPF9898 message will be sent if either of the following conditions hold true:

1. The number of rows in the table is greater than 100,000.
2. The number of rows in the table is greater than the SQL_FAST_DELETE_ROW_COUNT QAQQINI value in effect for the job.

The CPF9898 message contains enough information to diagnose why fast delete was not achieved. Figure 6 shows an example message.

Figure 6: Fast delete failed message



The message text includes the system name for the target table (i.e., the file name) and schema (i.e., the library name). The RETURN CODE value corresponds to the QQI1 values. The STEP information is helpful only to IBM; it indicates the internal step you were on within the database delete processing. FAILING MESSAGE might contain a preceding message identifier, if applicable.

ADDITIONAL LOCK INFO could contain three values:

- 0 indicates no SPACE LOCATION lock timeouts were encountered.
- 1 indicates a SPACE LOCATION lock timeout was encountered.
- 2 indicates that the fast delete request conflicts with an open cursor in this job.

The NUMBER OF ACTIVATES and NUMBER OF DEACTIVATES values are counts of the previous and current usage of the file within this job. The example message in Figure 6 contains RETURN CODE = 00005, which corresponds with DDS_LOGICAL_FILE, which happened because the DELETE was executed against a view.

If you experience unexpectedly slow delete operations and don't have the option of collecting an SQL Performance Monitor, check the job log or history log for this CPF9898.

Commitment Control

A common misconception about fast delete processing is that you have to avoid commitment control (i.e., use COMMIT(*NONE)) to achieve fast deletes. This view is understandable; commitment control implies slower processing, and users previously had few clues from the database regarding whether fast delete was successful. To set the record straight, fast-delete processing is possible when commitment control is used, as long as the transaction doesn't have a pending change for the target table.

If the previous paragraph left you underwhelmed, read on before you judge the relevancy of fast delete to DELETE statements run under commitment control. When a DELETE statement cannot be implemented using fast delete and commitment control is used, a journal entry is written for every row deleted. If the table contains a million rows, a million journal entries will dutifully reflect the changes to the file. Fast delete processing has a key advantage in this regard, because fast delete operations are recorded in the journal using a single entry. The table below compares the journal entry details.

Journal entries for fast and slow deletes

Delete implementation	Number of journal entries	Journal code	Entry type	Description
Slow delete	1 per row	R	DL	Record deleted in the physical file member
Fast delete	1	D	CG	Change file

Note: The fast delete CG entry-specific data include the Change field type, which will be set to 3 = SQL DELETE FROM table (without a WHERE clause).

The fast delete performance savings are compounded if a logical-replication, high-availability product is being used. Instead of sending many individual record-deleted entries to the backup and replaying them one by one, a single journal entry covers the entire operation.

Implement Fast Deletes

This article was intended for anyone who works with performance-critical applications that clear data from large tables using SQL. Hopefully, the rules and tooling I described will help you understand why some deletes end up being slower than others. I know the chances are remote that I will become a fast runner, but I hope I can help you better implement fast deletes.

Service Level

The support described in this article is available on IBM i 6.1 and higher releases.

Find Out More

The following resources and web pages contain information related to this topic:

- Database performance and query optimization book
- Db2 for i Lab Services Offerings
- IBM i Technology Updates wiki
- IBM i journal entries by code and type