

Halloween Problem




Last updated by | Vitor Tomaz | Feb 24, 2023 at 3:28 AM PST

Contents

- [Issue](#)
- [Investigation / Analysis](#)
- [Mitigation](#)
- [More information](#)
 - [Simulation](#)

Issue

The Halloween Problem is a well documented issue, and it affects other database systems, not just SQL Server:

- [Halloween Protection](#) 
- [The Halloween Problem in SQL Server and suggested solutions](#) 
- [The Halloween Problem](#) 

The symptoms seen by the customer include never-ending execution of stored procedures or queries, or command timeouts for operations that "never" finish.

Investigation / Analysis

Essentially the Halloween problem in general occurs when the read for an update isn't protected from the update itself, and the update causes rows to move within the index used by the read, thereby causing the same row(s) to get updated repeatedly. In most common scenarios, we see this problem within the update, insert, delete statement where read and write cursors are used.

Mitigation

To solve this issue, SQL server need to introduce blocking operators (like SPOOL) to avoid the same data being update multiple times. A blocking operator would cause all the rows coming from the query against our primary table to pool in that operator. This blocking operators will copy all the rows (eager spool) or just some rows at the time (lazy spool) to tempdb.

More information

In most common scenarios, we see this problem within the update, insert, delete statement where read and write cursors are used. In this example here, the read in question isn't the one at the bottom of the update itself, but the one used in the cursor's fetch operation.

The update changes the key column of the index, causing the updated row to move forward in the index, thus causing that updated row to be fetched again in the second fetch after the update. This process repeats until all rows have been processed from the source.

(0,1) <= first fetch, updated to (2,1), and this row moves below the next one
(1,2)

which leads to

(1,2) <= second fetch, updated to (3,2), which in turn moves this row below the next one, and so on
(2,1)

Simulation

```
CREATE TABLE EMPLOYEE
  (ID INT IDENTITY(1,1),
   [NAME] VARCHAR(80),
   SALARY FLOAT)

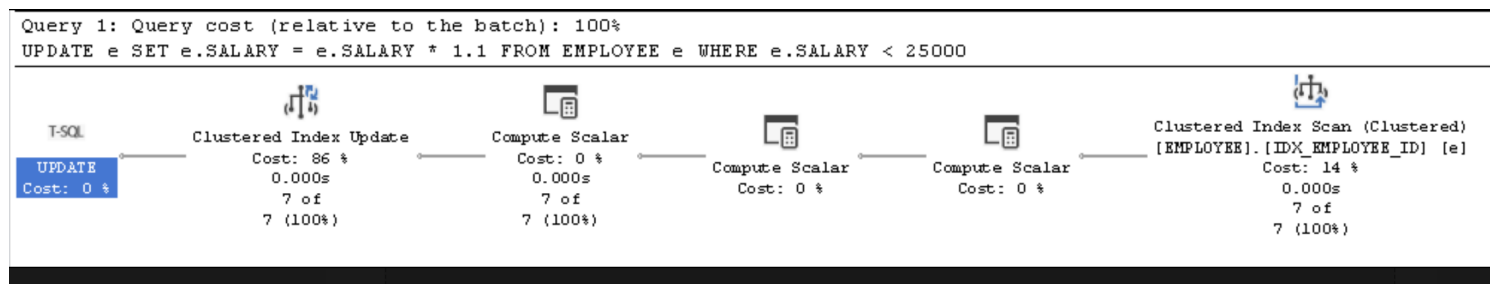
CREATE CLUSTERED INDEX IDX_EMPLOYEE_ID ON EMPLOYEE(ID)
CREATE INDEX IDX_EMPLOYEE_SALARY ON EMPLOYEE(SALARY)

INSERT INTO EMPLOYEE (NAME,SALARY) VALUES ('LUKE', 5000)
INSERT INTO EMPLOYEE (NAME,SALARY) VALUES ('LUKE', 7000)
INSERT INTO EMPLOYEE (NAME,SALARY) VALUES ('LUKE', 9000)
INSERT INTO EMPLOYEE (NAME,SALARY) VALUES ('LUKE', 11000)
INSERT INTO EMPLOYEE (NAME,SALARY) VALUES ('LUKE', 13000)
INSERT INTO EMPLOYEE (NAME,SALARY) VALUES ('LUKE', 15000)
INSERT INTO EMPLOYEE (NAME,SALARY) VALUES ('LUKE', 17000)
```

Now execute the "historical" update:

```
UPDATE e
SET e.SALARY = e.SALARY * 1.1
FROM EMPLOYEE e
WHERE e.SALARY < 25000
```

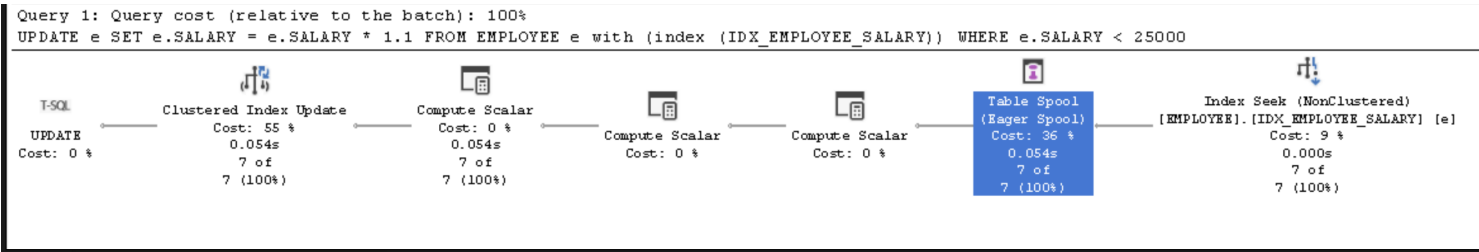
Looking at the execution plan we can see that the index is not used



To force the use of the index we will use

```
UPDATE e
SET e.SALARY = e.SALARY * 1.1
FROM EMPLOYEE e WITH (INDEX (IDX_EMPLOYEE_SALARY))
WHERE e.SALARY < 25000
```

The index is used and the Table Spool is used to make sure the same field is not read several times before the update.



Note: The Spool operators (blocking ones) **can** create suboptimal execution plans leading to performance issues.

How good have you found this content?

