sql Records

T-SQL Techniques and Tuning by Regan Wick

Thursday, January 10, 2013

Identifying and Eliminating Key Lookups in Query Execution Plans

There are four sections in this post
1. Definition of Key Lookup
2. How to eliminate a Key Lookup
3. Further aspects
4. DMV Query to identify and prioritize Key Lookups

1. Definition of Key Lookup

A Key Lookup occurs when the optimizer elects to use a non-clustered index but must reference back to the clustered index to read data for columns not in the non-clustered index.

Consider a Customer table as follows which I have populated with 1M records.

```
CREATE TABLE [dbo].[Customer](
	[IDCustomer] [int] IDENTITY(1,1) NOT NULL,
	[TXFirstName] [varchar](50) NOT NULL,
	[TXLastName] [varchar](50) NOT NULL,
	[TXAddress] [varchar](100) NOT NULL,
	[TXPhone] [varchar](50) NOT NULL,
	[DTCreated] [datetime] NOT NULL,
CONSTRAINT [PK_Customer] PRIMARY KEY CLUSTERED
(
	[IDCustomer] ASC
)
) ON [PRIMARY]
GO
```

We want to query this table for all customers' First Name and Last Name who have a DTCreated in the first week of January 2010.

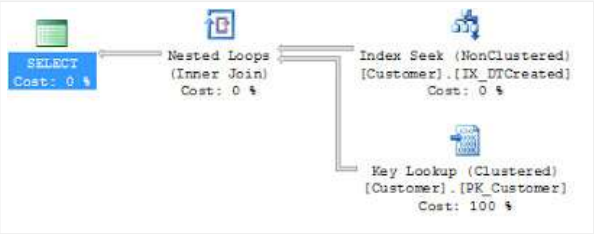
```
CREATE PROCEDURE dbo.GetData
AS
	SELECT
		TXFirstName
		,TXLastName
	FROM
		dbo.Customer
	WHERE
		DTCreated >= '2010-01-01'
	AND	DTCreated < '2010-01-08';
```

We can see that this performs a Clustered Index Scan. Without an index on DTCreated, it has no choice but to scan the entire base table picking off records that match the DTCreated criteria.

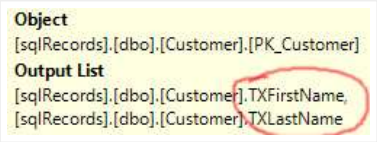
Next, we create an index on DTCreated. This will improve the query by allowing for a seek operation on the DTCreated values and thereby eliminating the full table scan of all 1M records.

```
CREATE NONCLUSTERED INDEX [IX_DTCreated] ON [dbo].[Customer]
(
	[DTCreated] ASC
) ON [PRIMARY]
GO
```

But this is where we see the introduction of the Key Lookup operation. Because the index does not specify TXFirstName and TXLastName (which are in the SELECT clause) the plan must reference the base table to lookup those values.



Key Lookup Properties

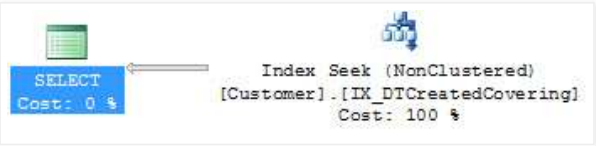


The Output List specifies the columns not in the index which are causing the Key Lookup to occur. This is essential information for eliminating the Key Lookup.

2. How to eliminate a Key Lookup

While in most cases better than a full table scan, a Key Lookup still involves additional reads, and so if possible Key Lookups should be eliminated. We can eliminate the Key Lookup which references the base table by INCLUDE-ing TXFirstName and TXLastName (detailed in the Key Lookup properties) in the index thereby covering the query. Essentially, the index is functioning as a separate table which contains all the columns required for the search criteria.

```
CREATE NONCLUSTERED INDEX [IX_DTCreatedCovering] ON [dbo].[Customer]
(
	[DTCreated] ASC
)
INCLUDE ([TXFirstName],[TXLastName]) ON [PRIMARY]
GO
```



Index Seek Properties

3. Further aspects

This section covers 2 additional aspects of covering indexes and Key Lookup elimination: (a) Index Key versus INCLUDEd columns and (b) index benefit versus cost.

Prior to SQL Server 2005, the only way to have an index cover a query was to append the needed columns to the index key. Using our example, such an index would look like this.

```
CREATE NONCLUSTERED INDEX [IX_DTCreatedTXFirstNameTXLastName] ON [dbo].[Customer]
(
	[DTCreated] ASC,
	[TXFirstName] ASC,
	[TXLastName] ASC
) ON [PRIMARY]
GO
```

The problem with this approach is a wide index key reduces the number of records that can fit on intermediate pages within the index b-tree, and so the depth of the tree likely will increase requiring additional reads to get to the same records.

Beginning with SQL Server 2005, indexes support an INCLUDE clause which specifies column values that should be written only to leaf-level pages. This is having it both ways: keeping the index key narrow while being able to cover queries with columns not relevant to the index key itself. The IX_DTCreated index has a depth of 3 whereas the wide index IX_ DTCreatedTXFirstNameTXLastName has a depth of 4. However, the IX_DTCreatedCovering has a depth of 3 (same as the IX_DTCreated) because the additional columns are INCLUDEd only on the leaf-level pages of the index.

The lesson here is to put search-interesting columns (aligned to the WHERE clause) in the index key and other columns referenced (SELECT and ORDER clauses) in the included section. Depending on the situation it may make sense to widen the index key with ORDER clause columns.

The second further aspect of indexes to consider is cost versus benefit. An index provides benefit for reads but costs on inserts, updates, and deletes – because these operations must update all affected indexes. So whereas IX_DTCreated is not updated when a value of TXFirstName has been changed, IX_DTCreatedCovering is updated. It is always wise to review Key Lookups for possible elimination, but it will make sense to accept one if the cost outweighs the benefit.

4. DMV Query to identify and prioritize Key Lookups

Even the best maintained system will end up having unnecessary Key Lookups as sprocs are updated to return additional columns not accounted for when indexes were originally created. And as we all know, many databases exhibit no evidence that tuning items such as covering indexes have been even considered.

Fortunately, the dynamic management views (DMVs) offer visibility into Key Lookups. The DMV query below returns data for database objects with Key Lookup operators in their execution plan. One of the columns returned is the XML representation of the query plan, which when selected within an SSMS query window will display the graphical query plan. I do not claim that this is best DMV query for this data, but it is simple and works well.

```
WITH DataCTE
AS
(
SELECT
	 CAST(ISNULL(db_name(QueryText.dbid),'') AS NVARCHAR(128)) AS [Database]
	,CAST(ISNULL(object_name(QueryText.objectid, QueryText.dbid),'') AS NVARCHAR(128))AS
Object

,sys.dm_exec_cached_plans.plan_handle

	,SUM(QueryStats.execution_count) AS ExecutionCount
	,MAX(QueryStats.plan_generation_num) AS Recompilation_Total
	,SUM(QueryStats.total_elapsed_time)	AS WallClock_Total
	,SUM(QueryStats.total_worker_time) AS CPU_Total
	,SUM(QueryStats.total_logical_reads) AS LogicalReads_Total
```

```
        ,SUM(QueryStats.total_logical_writes) AS LogicalWrites_Total
        ,SUM(QueryStats.total_physical_reads) AS PhysicalReads_Total
FROM
        sys.dm_exec_cached_plans
        INNER JOIN sys.dm_exec_query_stats  QueryStats
               ON QueryStats.plan_handle = sys.dm_exec_cached_plans.plan_handle
        CROSS APPLY sys.dm_exec_sql_text (sql_handle) QueryText
WHERE
        ISNULL(db_name(QueryText.dbid),'')  NOT IN ('','master','msdb')
GROUP BY
         CAST(ISNULL(db_name(QueryText.dbid),'') AS NVARCHAR(128))
        ,CAST(ISNULL(object_name(QueryText.objectid, QueryText.dbid),'') AS NVARCHAR(128))

,sys.dm_exec_cached_plans.plan_handle

)
SELECT
         [Database]
        ,Object
        ,query_plan AS QueryPlan
        ,plan_handle
        ,ExecutionCount
        ,Recompilation_Total
        ,WallClock_Total
        ,CPU_Total
        ,LogicalReads_Total
        ,LogicalWrites_Total
        ,PhysicalReads_Total
FROM
        DataCTE
        CROSS APPLY sys.dm_exec_query_plan(plan_handle)
WHERE
        CAST(query_plan AS NVARCHAR(MAX)) LIKE '%Lookup=%'
ORDER BY
      (WallClock_Total / ExecutionCount) DESC --Avg Wall Clock
        --(CPU_Total / ExecutionCount) DESC --Avg CPU
        --(LogicalReads_Total / ExecutionCount) DESC --Avg Logical Reads
        --(LogicalWrites_Total / ExecutionCount) DESC --Avg Logical Writes
        --(PhysicalReads_Total / ExecutionCount) DESC --Avg Physical Reads
```

Having removed from our example all non-clustered indexes except IX_DTCreated which we know uses the Key Lookup, we can see that this DMV query returns a record for the GetData sproc.

Database	Object	QueryPlan	plan_handle	ExecutionCount	Recompilation_Total	WallClock_Total	CPU_Total	LogicalReads_Total	LogicalWrite...	PhysicalReads
sqlRecords	GetData	<ShowPlanXML xmlns="ht...	0x050007008...	5	1	401150	399378	30218	0	0

Upon selecting the QueryPlan XML link, we see the graphical representation with a Key Lookup operator (same as from the end of section 1) and can investigate and update the index to eliminate the Key Lookup - again if it makes sense.

It should be noted that this DMV covers only sprocs and UDFs. In-line SQL queries that use Key Lookups would not be reported here.

Hope this is of some help.

Later

-Regan

No comments:

Post a Comment

Enter your comment...

Comment as: Google Accour

Publish Preview

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

Blog Archive

▶ 2018 (2)
▶ 2017 (1)
▶ 2016 (1)
▶ 2014 (1)
▼ 2013 (3)
▶ October (1)
▶ February (1)
▼ January (1)
Identifying and Eliminating Key Lookups in Query E...
▶ 2012 (1)

About Me

Regan Wick

T-SQL techniques & tuning

View my complete profile