

SQLskills Immersion Event

IE1: Internals and Performance

Module 11: Indexing Strategies

Kimberly L. Tripp
Kimberly@SQLskills.com



Overview

- Indexing for performance
- Indexing for AND
- Indexing for OR
- Indexing for joins
- Indexing for aggregates
- Indexed views (for aggregates)



2

© SQLskills, All rights reserved.
<http://www.SQLskills.com>



Indexing for Performance

- **Extremely challenging**
 - Users lie ☺
 - Workload specific
 - Data modifications are impacted by indexes (indexes add overhead to INSERTs/UPDATEs/DELETEs)
 - The type and frequency of the queries needs to be considered
 - This can change over time
 - This can change over the course of the business cycle
 - Need to have an understanding of how SQL Server works in order to create the “RIGHT” indexes – you CANNOT just guess!
- **To do a good job at tuning you must:**
 - Know your data
 - Know your workload
 - Know how SQL Server works!

Indexing Strategies at Design

- **First and foremost: choose a GOOD clustering key**
- **Create your primary keys and unique keys**
- **Create your foreign keys**
 - Manually index your foreign keys with nonclustered indexes
- **Create any nonclustered indexes needed to help with highly selective queries (lookups are OK for highly selective queries)**
- **STOP: this is your “design” base**
- **Add indexes slowly and iteratively over time while learning and understanding your workload as well as query priorities and always re-evaluate if/when things change!**

But Will YOUR Queries Use Them?

- **Subset of columns = projection**
 - Do not use * (unless against view)
 - Optimizer has more chances for optimizing query when result set is NARROW (only the required columns)
- **Subset of rows = selection**
 - Use positive search arguments
 - Isolate the column to one side of the expression
 - **USE:** MonthlySalary > value/12 (constant, seekable)
 - **DO NOT USE:** MonthlySalary * 12 > value (must scan)
 - Be cautious with LEADING wildcards
 - **USE:** LastName LIKE 'S%'
 - Avoid just appending %val% to every value (from the app)
- **Consider using views, stored procedures and functions to limit the columns/rows**

Indexing Strategies Overall

- **Good base table indexes and a very small number of indexes to start**
(some performance improvements should be handled by good design strategies)
- **General strategies:**
 - Narrow indexes have **very few** uses!
 - Be careful that your general strategy is NOT:
 - See a WHERE clause, create a single-column index on it
 - To automatically create an index on every column (horrible!)
 - Guessing... or tuning queries randomly (without workload/index analysis)
 - Wider indexes have MANY, MANY more uses!
 - I'm not saying that you need to create indexes that have all of your columns in them but understanding a lot more about internals and how SQL Server works is VERY important for better performance!

Indexing Pitfalls

- **USE the tools!**
 - STATISTICS IO
 - Showplan/Missing Index DMVs
 - Database [Engine] Tuning Advisor
 - BEWARE of the limitations of the tools!
 - Missing Index DMVs (and therefore showplan) only tune the plan that was executed
 - They do not “hypothesize” about alternatives (like DTA does)
 - All of the index recommendation from tools tend to go for “the best” choice rather than good enough choices
 - NONE of the tools do index consolidation...

Indexing for AND

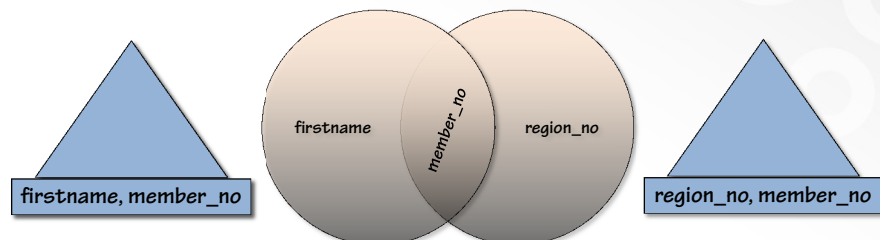
- **AND progressively limits the SET**
- **All conditions MUST be true**
- **Indexing strategies**
 - Evaluate columns in WHERE clause
 - Index any single highly selective set
 - Index a combination of columns to yield a highly selective set
 - Order should be based on the most commonly combined criteria (if all SARGs use equality)
 - Order should be based on the most selective criteria (if SARGs use varying operators such as >, < or LIKE)
 - If no combination of criteria create a selective set AND it's a high priority query, consider covering the query
 - SQL Server may use index intersection to intersect two relatively small sets (HASH Join), this is likely to be achieved without trying

Index Options

```
SELECT m.Member_No, m.FirstName, m.Region_No
FROM dbo.Member AS m
WHERE m.FirstName LIKE 'K%'
      AND m.Region_No > 6
      AND m.Member_No < 5000
```

- **Table scan (always an option)**
 - Clustered on member_no so a full table scan is unnecessary
 - SQL Server can “seek” with a partial table scan
- **NC index on firstname (K% is not very selective)**
- **NC index on region_no (region_no > 6 is 1/3 of the table)**
- **What does SQL Server do?**

Index Intersection



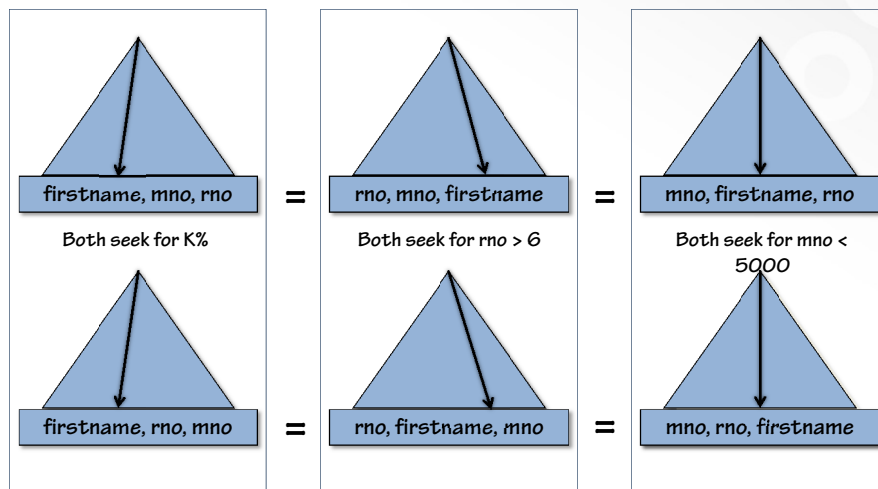
- Think of each of your nonclustered indexes as sets (as mini tables ordered by the key of the index)
- All nonclustered indexes “include” the clustering key in the leaf level of the index
- If we could “join” (or intersect) these sets on their common element (member_no) then we could find the data that we need...
- And, our query only wants these columns
- The intersection of these two indexes covers our query!

Index Intersection

- **Not the *fastest* option available for performance**
 - Requires more than 1 index to get to the data
 - Potentially requires tempdb space (HASH match)
 - Not something for which I strategize but something that might happen with lower priority queries that aren't covered and for those it's PERFECT
- **If it's a high priority query, you should consider doing with 1 index what you're currently doing with 2!**
 - No temp table
 - Only one index to seek/scan

Index Options

rno = region_no
mno = member_no



All indexes are the same size (same columns) but the ORDER of the columns is different

What's Best Depends On the QUERY!

- An index that has **firstname** first is better for THIS query because it's the most selective SET (based on the query, not the data itself)
- An index with **region_no** first is good, possibly better if the **firstname** might accept leading wildcards such as
 - WHERE `firstname LIKE '%e%'`
- Not as big of a fan of having **member_no** first
 - It's the most selective data column (it's unique) but, we already have a clustered index on **member_no**
 - If a highly selective query were to run then SQL Server could seek into the clustered index...
 - If ALL of the queries supply all three of the parameters then **region_no** or **firstname** first would help more queries!
- Remember, **ALL 6 are better than 2 or even a partial table scan!**

Summary: Key Order – How Do You Decide?

- **First and foremost – it depends on the usage of the columns**
 - If you ALWAYS supply **LastName** and sometimes supply **FirstName**
 - **LastName, FirstName** is better than **FirstName, LastName**
- **Second – it depends on the types of predicates (equality?)**
 - If EVERY query ALWAYS supplies ALL conditions and those conditions are accessed with equality conditions, it does NOT matter:
`WHERE LastName = 'Tripp' AND FirstName = 'kimberly'`
 - Then, it doesn't matter (these two indexes are REDUNDANT **in this case**):
 - **LastName, FirstName**
 - **FirstName, LastName**
- **Third – what about inequality?**
 - Once you start adding predicates that want inequality (**LIKE, <, >**, etc.) then you might only benefit (or, be able to seek on) the first condition. So, the 2nd and 3rd condition might be OK just to be in the INCLUDE

Indexing for OR

- **What is OR doing?**
 - Gather individual sets
 - Bring together and ensure that any row that appears in multiple places is only displayed once
 - Sound familiar?
- **IN is just a simplified series of OR conditions**
 - If an index exists to help search on each condition and EVERY specific value is HIGHLY selective, then it will use an index every condition
 - If any condition is not selective enough to use the index then a scan will be performed

Indexing for OR

- **For ideal performance tuning, treat each OR as a different query**
- **Each condition CAN use an index**
 - Each condition has to be *selective enough* to use the index
 - If there are 6 conditions and 5 are selective but 1 isn't then why would SQL Server use 5 different indexes and then still do a table scan...
 - If you have an IN then SQL Server can use the *same* index multiple times but if some of the conditions are selective and one are more are not then you hit the same issue as above – why would SQL Server use an index AND do a table scan!
- **The final step is that an OR cannot return duplicates – SQL Server MUST determine if any rows are in more than one result set.**
 - This often requires a temp table and a sort...

Indexing for OR

OR is Similar to UNION

- OR removes duplicate rows based on row's unique identifier (RID or clustering key)
- UNION removes duplicate rows based on the SELECT list
- This is NOT good enough... you must add the row's key to the SELECT list if you choose to use UNION
- If you're joining multiple tables, you should consider adding EACH table's key to the query
- OR always removes duplicates
 - What if you know there are no duplicates
 - What if you don't care if duplicates are returned
- Consider UNION ALL

Be sure to test this thoroughly as your queries are semantically different when you change from OR to UNION

Demo

Indexing for OR

On your own...

(play with the scripts)

Using the Tools for Join Tuning

- Understand how to break down a join
- Understand how to force your join for performance comparisons
- Understand the pros/cons of the showplan recommendations
- Understand how to best use DTA for a more well-rounded recommendation
 - Use DTA from SSMS to see all of the recommendations
 - Know how to use DTA's recommendations iteratively!

Indexing for Joins

- Multiple possible join strategies: do you need to care?
- Items on which to focus:
 - Most expensive table in the join (you have to start somewhere?!)
 - Most expensive join in the plan (it's probably downstream from the most expensive table and a join on that table)
 - Once you know the problem table AND the problem join, focus on tuning that particular table within that specific join!

Join Strategies

- **Loop join**
 - Iterative search on the inner table based on the number of rows that match in the driving table
- **Merge join**
 - Processing both tables at the same time using suitably sorted indexes
- **Hash join**
 - Build table (smaller set)
 - Probe table (larger set)
 - Either can use indexes to make the sets smaller
- **Key points: strategies will work for all three**
 - You do not need to know or care about the specific strategy... just give SQL Server the best information from which to choose!

Best Options for Joins: Phase I



SARG1
Join Col PK



SARG2
Join Col FK

*Do you already have
individual indexes on
each and all of these
columns?*

Foreign key???

- One join strategy might use Table1's SARG1 index to Table2's join key index (loop join)
- Another could use Table2's SARG2 index to Table1's join key index (loop join)
- Another could use only the join key indexes (merge)
- What's best depends on the data!
- If ALL 4 Indexes exist then the optimizer has the best choices

Cover the Combination: Phase II



SARG1
Join col PK



SARG2
Join col FK

Still not working?

- Not using these indexes?
- Performance still stinks?
- Cover the combo
 - Problem table (SARG, join): priority for the SARG
 - Problem table (join, SARG): priority for the join
- Only works when the cardinality of the join is low

Cover the Query: Phase III



SARG1
Join col PK



SARG2
Join col FK

- Covering the query/queries
- Cover the combo first, THEN add the additionally requested columns, with INCLUDE
 - Problem table (SARG, join): priority for the SARG
 - Problem table (join, SARG): priority for the join

Indexing for Aggregations

- Two types of aggregates:
stream and hash
- Try to achieve stream to minimize overhead in temp table creation
- Computation of the aggregate still required
- Lots of users, contention and/or minimal cache can aggravate the problem!

Aggregate Query

- Member has 10,000 rows
- Charge has 1,600,000 rows

```
SELECT c.member_no AS MemberNo,  
       sum(c.charge_amt) AS TotalSales  
FROM dbo.charge AS c  
GROUP BY c.member_no
```

Aggregate Query Table Scan + Hash Aggregate

```
SELECT c.member_no AS MemberNo,
       sum(c.charge_amt) AS TotalSales
FROM dbo.charge AS c
GROUP BY c.member_no
```

- Table scan of charge table
 - Largest structure to evaluate
 - Worst case scenario
- Worktable created to store intermediate aggregated results: OUT OF ORDER (HASH)
- Data returned OUT OF ORDER unless ORDER BY added
- Additional ORDER BY causes another step for SORT, and sorting can be expensive!

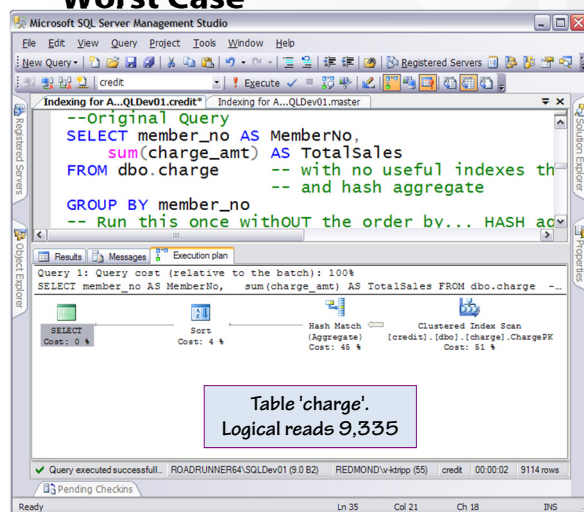
Worst Case

Clustered index scan
(table scan)
1,600,000 rows

Hash aggregate
yields 9,114 rows
out of order

Sort
only has to sort
9,114 rows instead
of 1,600,000 rows

Return data



Aggregate Query Index Scan + Hash Aggregate

```
SELECT c.member_no AS MemberNo,
       sum(c.charge_amt) AS TotalSales
FROM dbo.charge AS c
GROUP BY c.member_no
```

- Out of order covering index on charge table
 - Index exists which is narrower than base table
 - Used instead of table to cover the query
- Worktable still created to store intermediate aggregated results: OUT OF ORDER (HASH)
- Data returned OUT OF ORDER unless ORDER BY added
- Additional ORDER BY causes another step for SORT, and sorting can be expensive!

Not as Bad

COVERING

Index scan
1,600,000
narrower rows

Hash aggregate
yields 9,114
rows out of
order

Sort

only has to sort
9,114 rows
instead of
1,600,000 rows

Return data

Microsoft SQL Server Management Studio

Query 1: Query cost (relative to the batch): 100%

SELECT member_no AS MemberNo, sum(charge_amt) AS TotalSales FROM dbo.charge GR...

Execution plan:

- Sort (Cost: 6 %)
- Hash Match (Aggregate) (Cost: 59 %)
- Index Scan (Covering) (Cost: 35 %)

Table 'charge'. Logical reads 3,770

Index Scan

Scan a nonclustered index, entirely or only a range.

Physical Operation	Index Scan
Logical Operation	Index Scan
Number of Rows	1600000
Estimated Row Size	19 B
Estimated I/O Cost	2.78461
Estimated CPU Cost	1.76016
Estimated Operator Cost	4.54476 (35%)
Estimated Subtree Cost	4.5447602
Estimated Number of Rows	1600000

Object: [credit].[dbo].[charge].Covering1

Aggregate Query Index Scan + Stream Aggregate

```
SELECT c.member_no AS MemberNo,
       sum(c.charge_amt) AS TotalSales
FROM dbo.charge AS c
GROUP BY c.member_no
```

- **Covering index on charge table (in ORDER of GROUP BY clause)**
 - Index exists which is narrower than base table
 - Used instead of table to cover the query
 - Covers the GROUP BY so data is grouped
- **Less work to aggregate results IN ORDER**
- **Data returned IN ORDER unless ORDER BY/ joins added**
- **Adding an ORDER BY identical to the GROUP BY does NOT cause any additional step for sorting!**

Much Better!

COVERING
Index scan
1,600,000
narrower rows

Stream
aggregate
also yields
9,114 rows
IN ORDER

NO SORT
REQUIRED
Return data

The screenshot shows the SQL Server Management Studio interface with a query window and an execution plan. The query is:

```
-- Run the query again - you'll see STREAM
SELECT member_no AS MemberNo,
       sum(charge_amt) AS TotalSales
FROM dbo.charge
GROUP BY member_no
ORDER BY member_no -- TO be fair in the comparison
-- BUT, because this is a
```

The execution plan shows a 'SELECT' operator connected to a 'Stream Aggregate' operator, which is connected to an 'Index Scan' operator. The 'Index Scan' operator is marked as 'Covering2'. A tooltip for the 'Table 'charge'' shows 'Logical reads 3,770'. The 'Stream Aggregate' operator has a tooltip that says 'Compute summary values for groups of rows in a suitably sorted stream.' The 'Physical Operation' table shows the following details:

Physical Operation	Stream Aggregate
Logical Operation	Aggregate
Number of Rows	9114
Estimated Row Size	19 B
Estimated I/O Cost	0
Estimated CPU Cost	0.964557
Estimated Operator Cost	0.9645596 (18%)
Estimated Subtree Cost	5.5093198
Estimated Number of Rows	9114

At the bottom of the window, it says 'Query executed successfully... ROADRUNNER64S'.

See the Difference?

Query 1: Query cost (relative to the batch): 48%
 SELECT member_no AS MemberNo, sum(charge_amt) AS TotalSales FROM dbo.charge WITH (...)
 Execution plan: SELECT (Cost: 0%) → Sort (Cost: 4%) → Hash Match (Aggregate) (Cost: 46%) → Clustered Index Scan [credit].[dbo].[charge].ChargePK (Cost: 51%)

Query 2: Query cost (relative to the batch): 36%
 SELECT member_no AS MemberNo, sum(charge_amt) AS TotalSales FROM dbo.charge WITH (...)
 Execution plan: SELECT (Cost: 0%) → Sort (Cost: 5%) → Hash Match (Aggregate) (Cost: 59%) → Index Scan [credit].[dbo].[charge].Covering1 (Cost: 35%)

Query 3: Query cost (relative to the batch): 16%
 SELECT member_no AS MemberNo, sum(charge_amt) AS TotalSales FROM dbo.charge WITH (...)
 Execution plan: SELECT (Cost: 0%) → Stream Aggregate (Aggregate) (Cost: 18%) → Index Scan [credit].[dbo].[charge].Covering2 (Cost: 82%)

Query executed successfully. ROADRUNNER64\SQLDev01 (9.0 B2) REDMOND\w-ktripp (54) credit 00:00:07 27342 rows

Concerns

- **Hash aggregates**
 - More temp tables
 - More contention in tempdb
 - Larger tempdb required
 - Performance varies on each execution
- **Stream/hash aggregate**
 - Aggregate needs to be computed

Is there a better way?
Indexed views!

Demo

What kinds of gains can you get?
Will it be worth it?



Views/Indexes: Quick Review

- **Views**
 - Named, saved SELECT statement
 - Tabular data set
 - Data definition (no ORDER BY unless TOP is used)
- **Indexes**
 - Clustered (only 1 per table)
 - Defines order and structure of data
 - LEAF Level = Data (of the table)
 - Nonclustered (249/999 per table)
 - Separate and duplicated data
 - Automatically maintained
 - Order and structure defined per index

Indexed Views Defined

- A materialized view = meaning a named, saved query
- whose data is structured and ordered by the definition
- of the clustered index on the view.
- Pros = Speed (usually exponential gains)
 - No calculations
 - No joins
 - No aggregations
 - No temp tables
- Cons
 - Disk Space (minimal, affected by type of view)
 - INSERT/DELETE may affect the index
 - UPDATES to the data used by the view will also be update in the index
 - Goal: Don't index everything, find the right use

Indexed Views with Aggregations

Query defined uses aggregates for one or more of the columns
Interesting observation: data set often gets smaller

- Questions to ask
 - How many people are requesting the aggregation?
 - How complex is the aggregation?
 - What is the rate that columns involved in the computation change?
 - Are you searching on the aggregated value?
 - Is it highly selective?
 - How small does the set become? Hot row?
- When do you want to use indexed views with aggregations?
 - When users need ranges of data based on the aggregation
 - When read performance outweighs the disk space requirements
 - When the aggregation does not create significant contention!

Real Power of Indexed Views

- Optimizer is aware of them even when view is NOT referenced in the query (Enterprise/Developer Ed.)
- Calculations, aggregations and complex joins are already computed so only an index SEEK or SCAN is necessary
 - Processing steps might be saved
 - Possibly less disk access
 - Less CPU
 - Less memory (not necessarily with joins)
 - Tempdb space not necessary
- Nonclustered indexes can be created!
- It's a mini clustered table of all the data (including aggregations, joins, computations) that you need!

The Bad News *kind of...*

- Full support in Enterprise Edition
(and Developer Edition) ONLY
 - View does NOT need to be named and optimizer hints are not required
 - ANY query that CAN benefit WILL
- Reduced functionality on other Editions
 - VIEW must be named and an optimizer hint is REQUIRED

```
FROM viewName WITH (NOEXPAND)
```

 - Base table queries will NOT benefit from Index
 - ONLY queries that directly name and directly access the view will benefit
- Review complete details in the whitepaper

Seems Complex?

- Stay focused on purpose
- Keep indexes on views to a minimum unless READ ONLY database
- Consider impact to INSERT, UPDATE, DELETE performance
- Add indexed views to maintenance scripts

Test, test, test!



Aggregate Query Indexed View

```
SELECT member_no AS MemberNo,
       sum(charge_amt) AS TotalSales
FROM dbo.charge
GROUP BY member_no
ORDER BY member_no
go
```

Query 1: Query cost (relative to the batch): 100%

SELECT member_no AS MemberNo, sum(charge_amt) AS TotalSales FROM dbo.charge GROUP...

Cost: 0 %

Clustered Index Scan
Scanning a clustered index, entirely or only a range.

Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Number of Rows	9114
Estimated Row Size	19 B
Estimated I/O Cost	0.0268287
Estimated CPU Cost	0.0101824
Estimated Operator Cost	0.0370111 (100%)
Estimated Subtree Cost	0.0370111
Estimated Number of Rows	9114

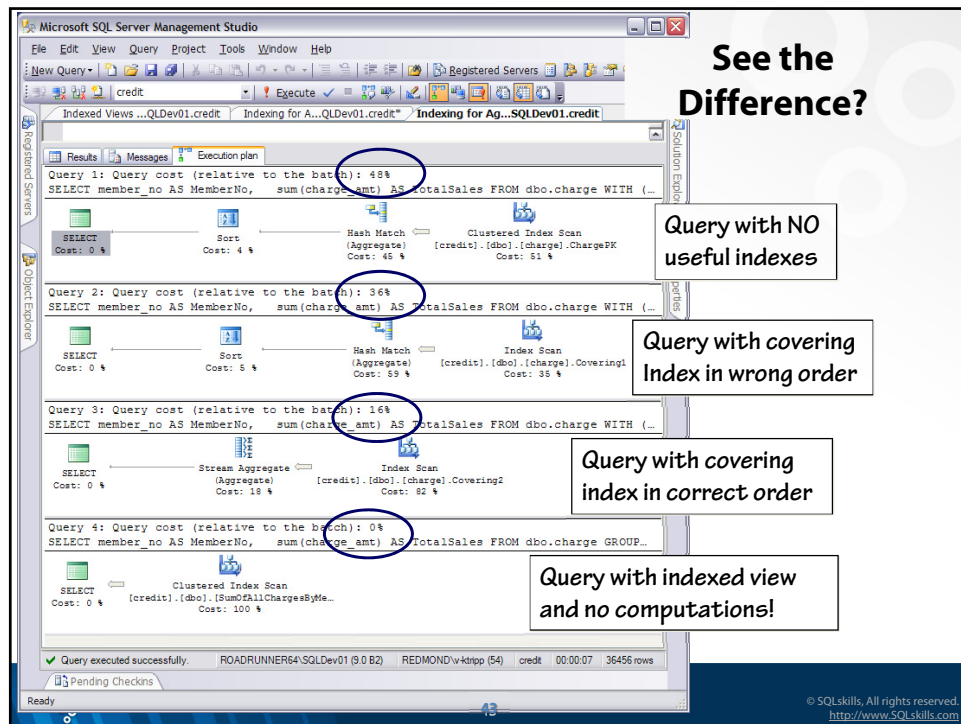
Object
[credit].[dbo].[SumOfAllChargesByMember].SumOfAllChargesIndex

Table 'SumOfAllCharges'.
Logical reads 35

Query executed successfully.

Ready

© SQLskills. All rights reserved.
<http://www.SQLskills.com>



Indexed View with Aggregates

Key points:

- Tempdb access not necessary
- NO worktables are necessary
- Aggregated set should be small but not too small as to create a hot ROW spot of activity (which can create excessive blocking)
 - GROUP BY member_no – probably OK
 - GROUP BY state – too few rows in aggregate
 - GROUP BY country – **AVOID** like the plague!
- Performance of data modification statements should be tested

Database Tuning Advisor

- DTA will include numerous configurable settings:
 - Length of TIME to tune
 - Physical design structures:
 - Indexes and **indexed views**
 - Indexes only
 - **Indexed views only**
 - Nonclustered indexes
 - Partitioned tables and indexes
 - Aligned partitioning
 - Full partitioning
 - Advanced: online vs. offline

Summary

- **Ask for ONLY the data you need**
 - Limit the rows requested with effective WHERE clause criteria
 - Limit the columns requested with effective SELECT lists
- **Prioritize OLTP/OLAP and choose the clustering key based on the clustered index debate**
- **Add “key” indexes**
 - Nonclustered for PK (if not the clustered)
 - Nonclustered for the unique keys
 - MANUALLY index your foreign keys
- **Add nonclustered for SARGs**
- **Consider covering for high priority/low selectivity**
- **Test, test, test!**

Review

- Indexing for performance
- Indexing for AND
- Indexing for OR
- Indexing for joins
- Indexing for aggregates
- Indexed views (for aggregates)

Questions!