# SQLskills Immersion Event

## IEPTO1: Performance Tuning and Optimization

## Module 10: Indexing Strategies

Kimberly L. Tripp
Kimberly@SQLskills.com

# Overview

- **Indexing for performance**
  - Design strategies
  - Overall strategies
- **Using the tools for tuning**
  - SET STATISTICS IO ON
  - Showplan
  - Missing indexes
- **Indexing for AND**
- **Indexing for OR**
- **Indexing for joins**
- **Indexing for aggregates**
- **Indexed views v. columnstore indexes**
- **Rowstore indexes v. columnstore indexes**

# 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!**

# Indexing Foreign Keys (1 of 2)

- **Helps referential integrity management**
  - When a primary key row is deleted, ALL foreign key references must be checked
    - When the foreign key column does not have an index whose key LEADS with the foreign key definition, then *something* has to be scanned
      - If there's no index that has the foreign key column in it, the table has to be scanned
    - Can be very expensive if there are many foreign key references and/or references from large tables
- **Helps the query optimizer better understand the relationship between tables when they're joined**
  - Foreign key values must exist in the referenced table
  - Foreign key values will reference exactly one row
- **Can help join performance**
  - When the more selective criteria is on the primary key table and SQL Server wants to join TO the foreign key reference

# Indexing Foreign Keys (2 of 2)

**Employee**

| ID | LN | FN | MI | … | DID |
|---|---|---|---|---|---|
| 1 | … | … | . | … | … |
| 2 | … | … | . | … | 63 |
| … | … | … | . | … | … |
| 345 | … | … | . | … | … |

```
SELECT [e].[LN], [e].[FN], [d].[DID]
FROM [Employee] AS [e]
    JOIN [Department] AS [d]
        ON [e].[DID] = [d].[DID]
WHERE [d].[City] = 'Bellingham'
```

**Department**

| DID | Name | … | City | State |
|---|---|---|---|---|
| 1 | … | … | … | … |
| 2 | … | … | … | … |
| … | … | … | … | … |
| 63 | … | … | Bellingham | WA |

```
SELECT [e].[LN], [e].[FN], [d].[DID]
FROM [Employee] AS [e]
    JOIN [Department] AS [d]
        ON [e].[DID] = [d].[DID]
```

But joining from Department to Employee doesn't work optimally without an index on Employee.DID

# 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 **<u>very few</u>** 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!
  - Columnstore should be considered for large aggregations but lots of other considerations (SQL Server version, reads v. updates, types of queries)

# 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**

# Using the Tools

- **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…
- **Resources:**
  - Search "Bart Duncan Missing"
  - Glenn's DMV Toolkit
  - A bit of searching – lots of good stuff out there!

# SET STATISTICS IO ON (1)

- **Scan count: does not mean table scan**
  - Nothing to do with actual type of access
  - Refers to the number of “accesses” an object
- **Logical reads: number of page accesses in the data cache – specific to this query’s execution**
  - A single page can be accessed many times and EVERY one of these will be counted

  * NOTE: Profiler vs. STATISTICS IO = Profiler includes I/Os performed during the execution of the query (for example, lookups into the plan cache, accessing metadata, security information, etc.). Profiler should always be greater than or equal to STATISTICS IO.
- **Physical reads: number of page reads that this query had to wait for – from disk**
- **Read-ahead reads: secondary process which accesses pages from disk (“reading ahead” of the query) so that SQL Server/CPU doesn’t have to wait**
- **Lob (logical, physical, and read-ahead) reads: same as the above but for all LOB [(n)text, MAX, XML] as well as limited-LOB data types that have overflowed**

# SET STATISTICS IO ON (2)

- **Use logical reads as a general "total"**
  - The cost of getting from A to B in "steps" alone
  - Similar to distance
    - Does not include any traffic [blocking] encountered along the way
    - Does not include any worktables required
  - Doesn't give you the complete picture
- **Use as a piece in the query execution information/puzzle**
  - Usually set in script
  - Can change it in SSMS
    - Tools, Options, Query Execution, Advanced:
      - SET STATISTICS IO ON
      - SET NOCOUNT ON
    - NOTE: Some of these options can have a profound affect on query performance. Should not change the ANSI options.

# Showplan

- **Estimated plan**
  - Gives you the plan that SQL Server came up with through optimization – without actually executing it
- **Actual plan**
  - Gives you the plan that SQL Server came up with through optimization – and, executed it
  - This is EXACTLY the same plan [shape] as estimated but includes actual numbers
    - Extremely beneficial in finding cardinality estimation issues
- **Definitely NOT perfect…**
  - What are you really seeing with cached plans and stored procedures…
    - Plans for COMPILED values not the actual value – these can be the most incorrect

# Missing Index Hints

- **The "green hint" in showplan, comes from the missing index DMVs**
- **Helpful, but**
  - Not always listed with the query it benefits (consider using SQL Sentry's Plan Explorer)
  - Gives you the index that reduces the most I/O for the plan that was executed
  - Doesn't consider other join types or join orders; doesn't always give the best plan
- **Good to try**
  - If you're ready to believe it and implement the suggestion consider checking to see what the Database [Engine] Tuning Advisor (DTA) recommends
  - Don't just trust it; must consider consolidation
    - Review existing indexes
    - Could you create a slightly-wider but better index? Possibly removing one or more existing indexes?
    - The more you tune – the more you'll find "similar" recommendations

# 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 ***predicate*** 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

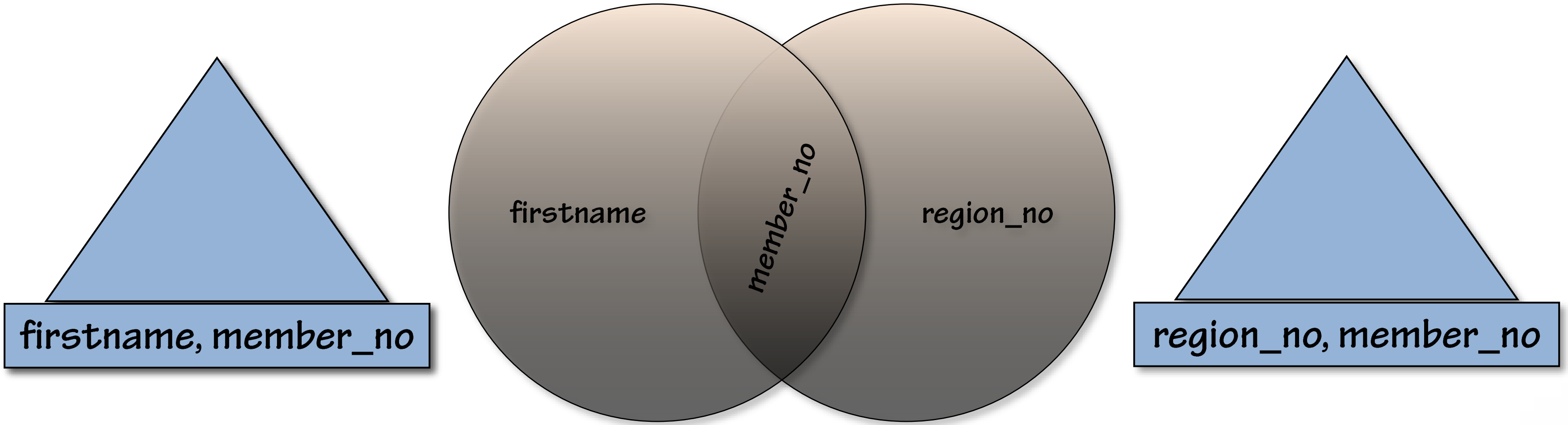
# Index Options

```
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



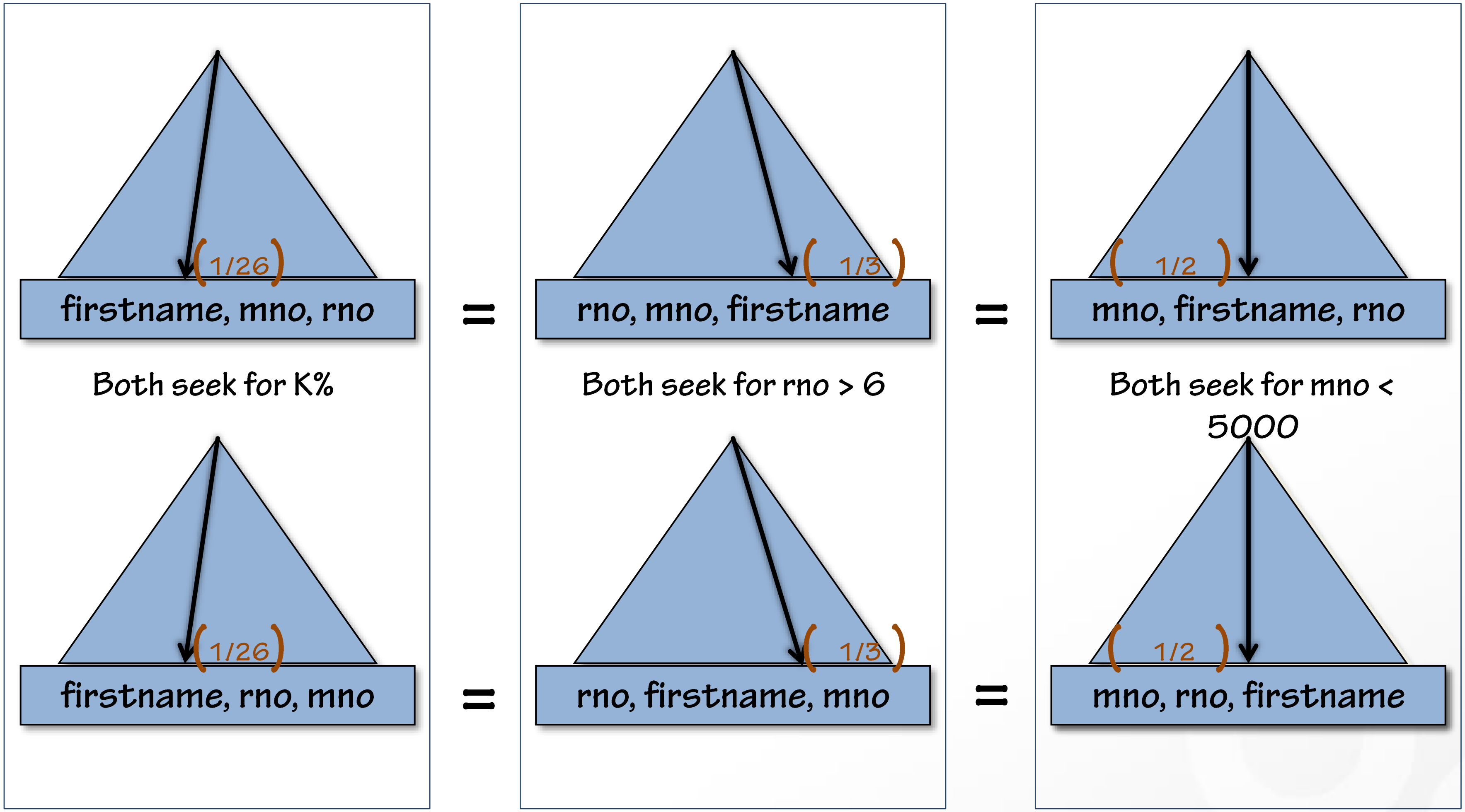

- Think of each of your nonclustered indexes as sets (as mini tables ordered by the key of the index)
- All nonclustered indexes "include" the clustering key in the leaf level of the index
- If we could "join" (or intersect) these sets on their common element (member_no) then we could find the data that 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

rno = region_no
mno = member_no

# Index Options




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*

# 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!

# Best Options for Joins: Phase I

Table1

SARG1
Join Col PK

Table2

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 SARG1 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 four indexes exist then the optimizer has the best choices**

# Cover the Combination: Phase II

Table1

SARG1
Join Col PK

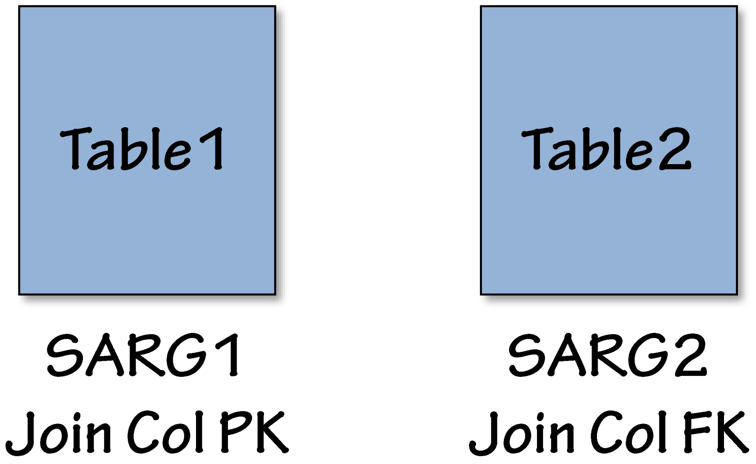
Table2

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

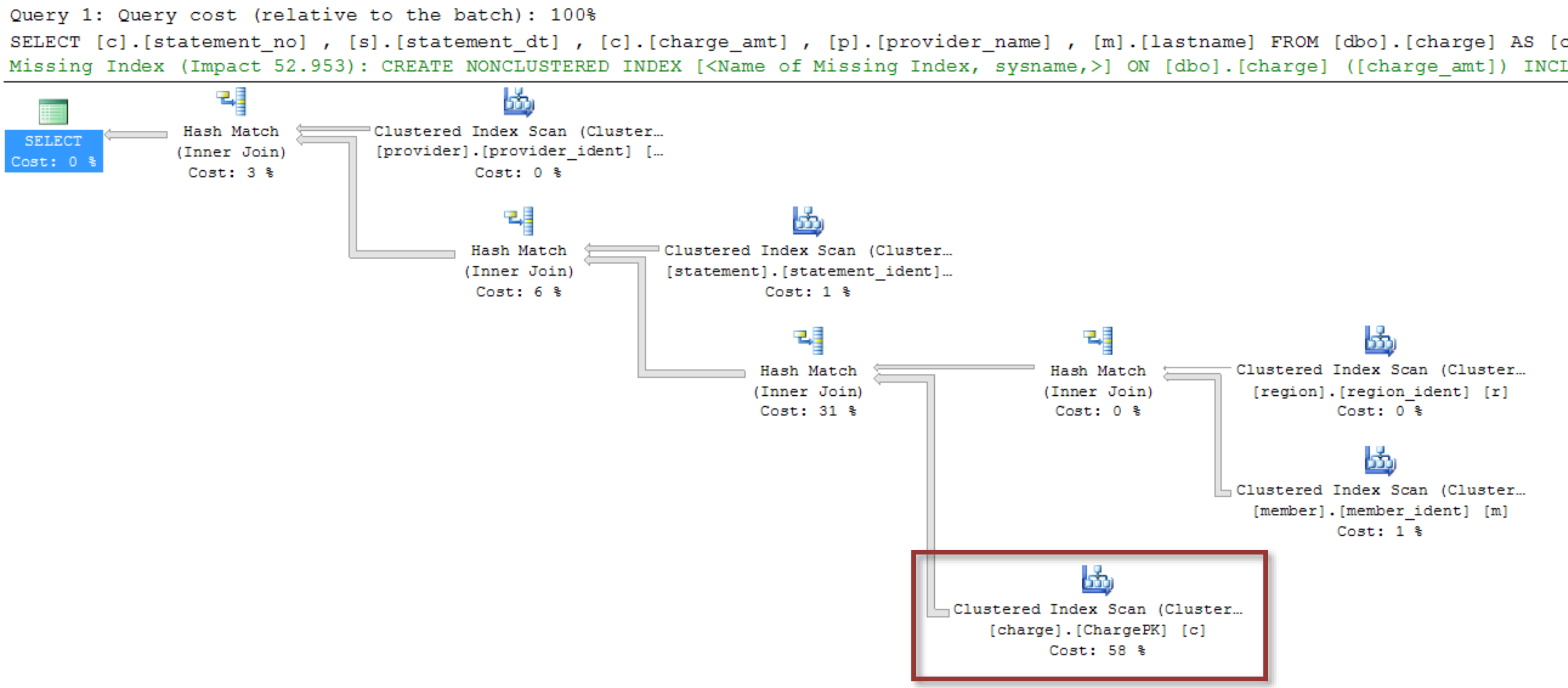



- **Covering the query/queries**
- **Cover the combo first, THEN add the additionally requested columns, with INCLUDE**
  - Problem table (SARG, join): priority for the SARG
  - Problem table (join, SARG): priority for the join

# Bringing It All Together (Long Demo)

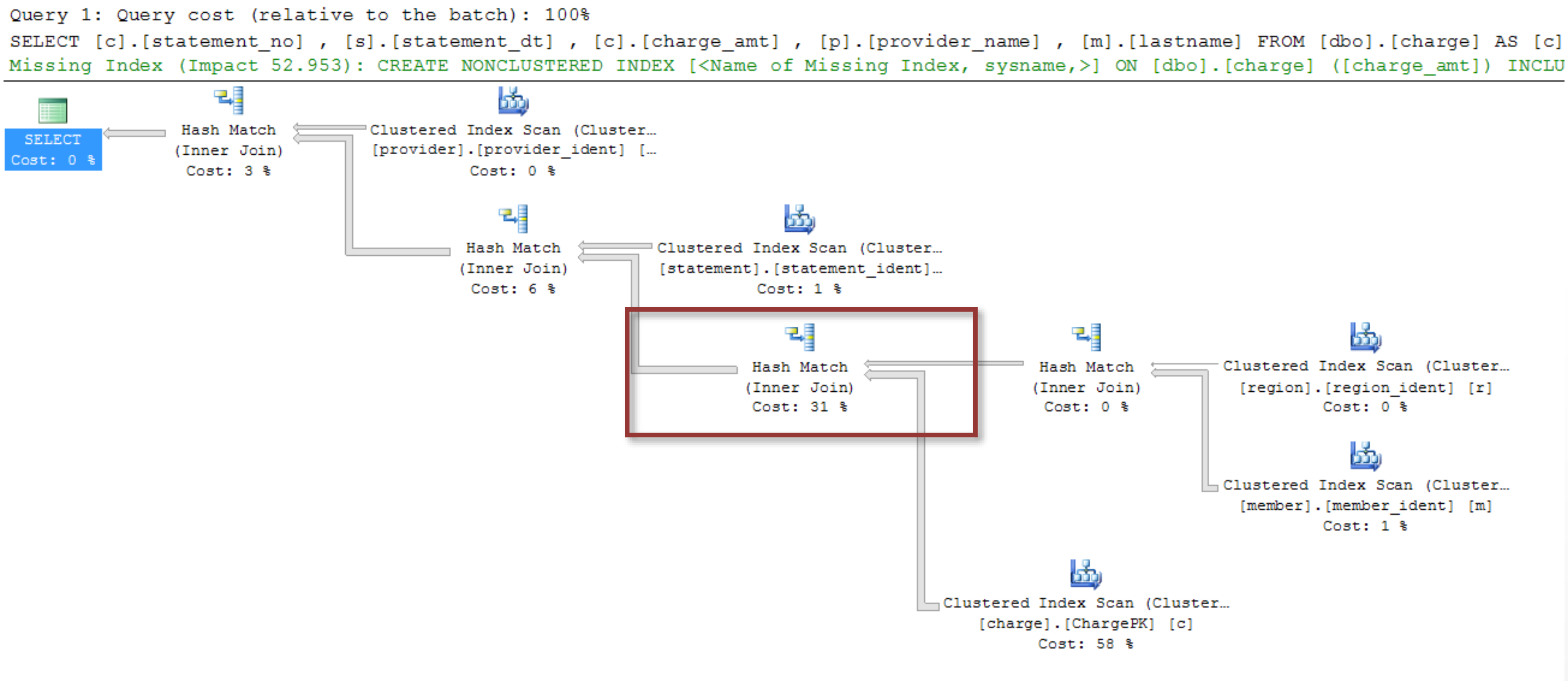
- **Pulling apart a plan and describing a lot while I do it…**
- **Hard-coding a query to create a base-line to go against**
- **Deciding where to start**
  - Analyzing where we have a problem(s)
  - Finding the problem table
  - Finding the problem join
- **Evaluating whether or not an index is a good idea**
  - Reviewing/debating the "green hint"
  - Using DTA from SSMS to see if the hint is different

# Table Scans – Are They Necessary?




- Is the table scan because you're returning the entire table/all columns?
- Or, it is because the right indexes don't exist?
- What table has the highest cost?

# Joins – Are All of the Joins Hash Joins?




- Are all of these hash joins because you're tables are large?
- Or, are they because the right indexes don't exist?
- What join has the highest cost?

# Manual Tuning Process

- **Find the most expensive table in the query (charge)**
  - Are there any SARGs – consider what the key would look like with these SARGs independent of the join conditions
- **Find the most expensive join in the query (charge joining to member)**
  - Figure out which join is the join that your problem table is joining to
- **Phase I should be already done**
- **Phase II should be considered**

```
CREATE NONCLUSTERED INDEX Charge_PriorityForSARG
ON [dbo].[charge] ([charge_amt], [member_no])

CREATE NONCLUSTERED INDEX Charge_PriorityForJoin
ON [dbo].[charge] ([member_no], [charge_amt])
```

- **Phase III adds any columns not already present, to the INCLUDE**

```
INCLUDE ([statement_no], [provider_no])
```

# Tuning Goal

- **Find the most expensive table in the query (charge)**
- **Find the most expensive join in the query (charge joining to member)**
- **Try to tune CHARGE for its join to member… How?**
  - Review the green hint:

```
CREATE NONCLUSTERED INDEX MissingIndexDMVRecommendation
ON [dbo].[charge] ([charge_amt])
INCLUDE ([member_no],[provider_no],[statement_no])
```

  - Double-check using DTA (Query, Analyze Query in DTA):

```
CREATE NONCLUSTERED INDEX [DTA_K6_K7_K3_K2]
ON [dbo].[charge]
([member_no], [charge_amt], [statement_no], [provider_no])
```

- **Notice how similar these indexes are?**
- **What do they do, how do they differ?**

# Result of Manual and Tool-based Tuning

- The missing index DMVs (via the green hint in showplan) came up with:

```
CREATE NONCLUSTERED INDEX MissingIndexDMVRecommendation
ON [dbo].[charge] ([charge_amt])
INCLUDE ([member_no],[provider_no],[statement_no])
```

- Manually, we came up with:

```
CREATE NONCLUSTERED INDEX Charge_PriorityForSARG
ON [dbo].[charge] ([charge_amt], [member_no])
INCLUDE ([statement_no], [provider_no])

CREATE NONCLUSTERED INDEX Charge_PriorityForJoin
ON [dbo].[charge] ([member_no], [charge_amt])
INCLUDE ([statement_no], [provider_no])
```

- The Database Tuning Advisor came up with:

```
CREATE NONCLUSTERED INDEX [DTA_K6_K7_K3_K2]
ON [dbo].[charge]
([member_no], [charge_amt], [statement_no], [provider_no])
```

# Benefits of These Indexes?

- **The green hint/the Missing Index DMVs recommendation:**
  - Leads with the column charge_amt
  - This removes the table scan and changes to an index seek
  - This allows filtering by our search argument (charge_amt > 2500)
  - PRO: This helps tune the plan that was chosen/executed
  - CON: They did not hypothesize for alternatives
- **The DTA's recommendation:**
  - Leads with the column member_no
  - This removes the table scan and changes to an index seek
  - This allows the join to change to a loop join
  - PRO: This significantly reduces the cost/time for the join
- **Of the tools – what's better? What gives better performance?**
  - DTA, but, our index was even a bit better given that you can't seek beyond charge_amt (as the SARGs against it are range-based)

# Database [Engine] Tuning Advisor

- **It's not always perfect**
- **It sometimes yields the same index recommendation that the missing index DMVs**
- **It often OVER recommends indexes (this is why you want to use it ITERATIVELY after really analyzing where to begin)**
- **It doesn't recommend any forms of index consolidation**
  - This is one of the reasons that a lot of development environments end up over-indexed
- **IF you end up creating the index that was recommended for a particular table, then, go ahead and create the statistics that are recommended for that table**
  - DTA can create multi-column statistics
  - The can give the optimizer other (sometimes VERY useful) ways of using the recommended index!

# Join Strategies

- **Loop join**
  - Iterative search on the inner table based on the number of rows that match in the driving table
  - Usually best when the driver (outer table [chosen by SQL Server]) is small
- **Merge join**
  - Processing both tables at the same time using suitably sorted indexes
  - Usually best when the RIGHT indexes exist
    - An index on EACH table that LEADs with the same column (the column being joined) is necessary
- **Hash join**
  - Two-phase operation (build, then probe): build table (smaller set) and probe table (larger set) allowing SQL to join extremely large sets – in MEMORY (can spill)
  - Either side *can* use indexes to make the sets smaller
  - When this occurs on reasonably small tables then it sometimes mean that good indexes don't exist
- **Key points: the strategy I demonstrated works for ALL join types**
  - You do not *need* to know or care about the specific strategy… just give SQL Server the best information from which to choose!

# 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

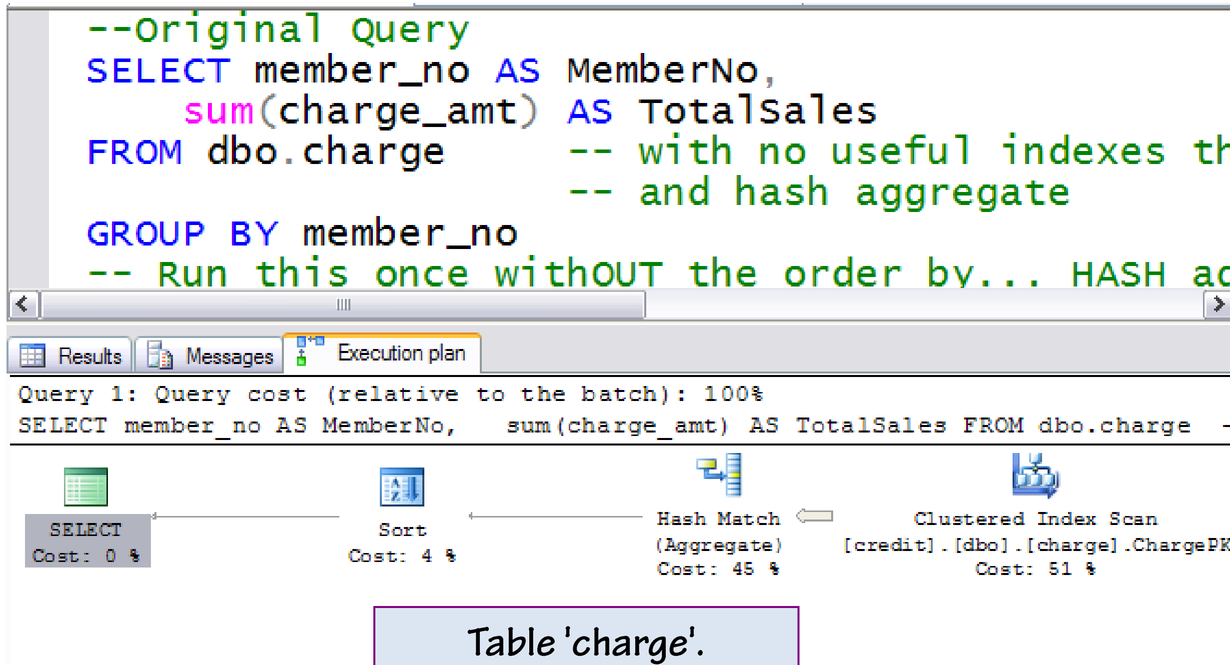



Table 'charge'.
Logical reads 9,335

# 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**

```sql
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 comparis
go
```

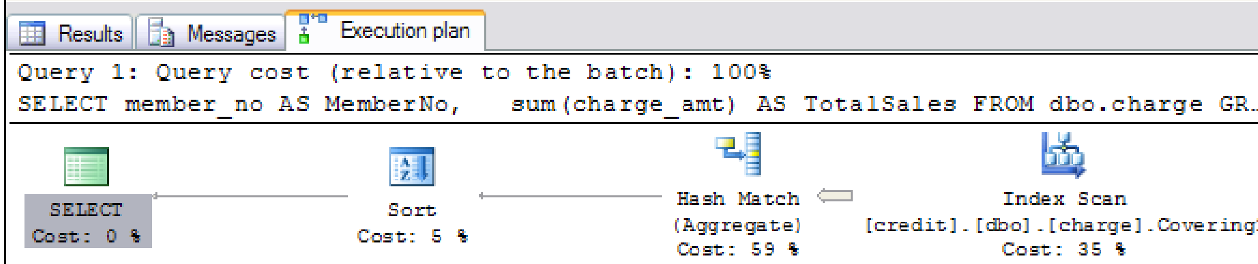



Table 'charge'.
Logical reads 3,770

**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!**

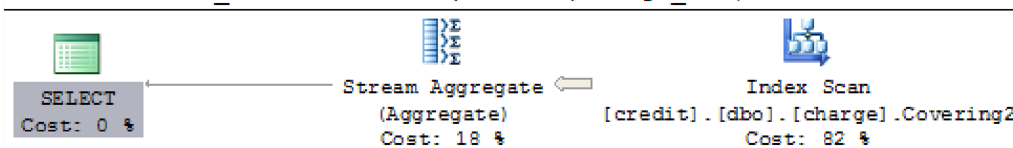
# Much Better!

- COVERING
  - Index scan
  - 1,600,000
  - narrower rows
- Stream
  - aggregate
  - also yields
  - 9,114 rows
  - IN ORDER
- NO SORT
  - REQUIRED
- Return data

```sql
-- 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 comparis
                          -- BUT because this is a
```

Results | Messages | Execution plan

```
Query 1: Query cost (relative to the batch): 100%
SELECT member_no AS MemberNo,   sum(charge_amt) AS TotalSales FROM dbo.charge GR
```



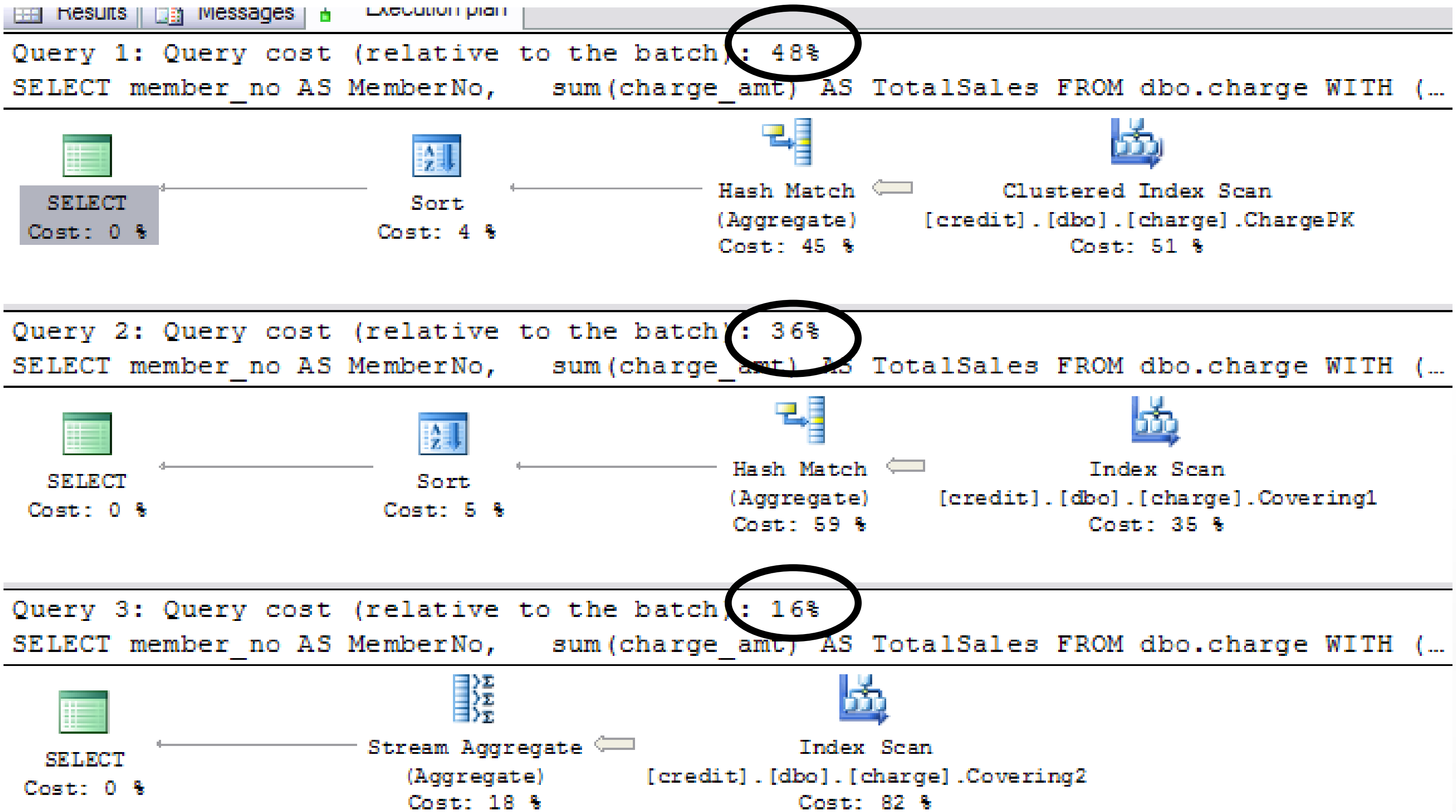

**Stream Aggregate**

Compute summary values for groups of rows in a suitably sorted stream.

| | |
|---|---|
| **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 |

Table 'charge'.
Logical reads 3,770

# See the Difference?



# 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?

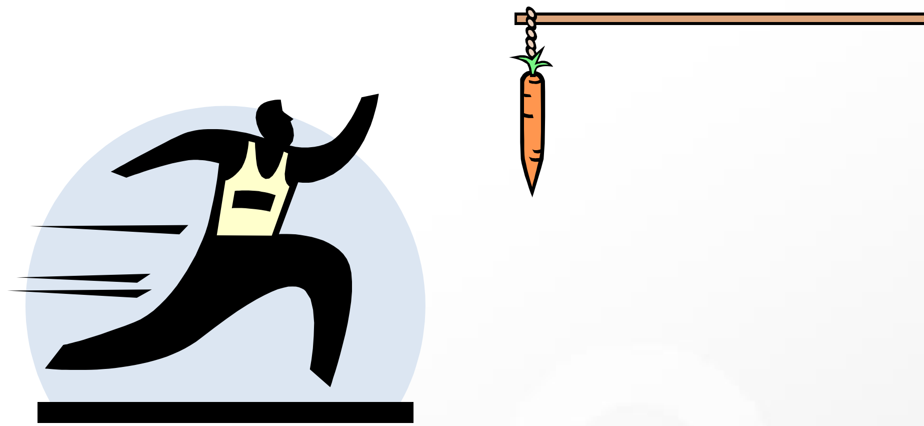
Indexed views (2000+)

Columnstore indexes (2012+)

# 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 v. Columnstore

## Indexed views

- Limited uses in non-Enterprise Editions
- Must be analyzed / created "per query"
  - More complicated to create
  - More storage required
  - More administrative overhead / maintenance
  - More costly to maintain during inserts / updates
- Requires certain session settings to be set

## Columnstore indexes

- Some limitations across versions:
  - BOL: [Features Supported by Editions]
- Only one can be created per table
  - Super easy to create
  - A lot LESS storage required (compression)
  - Less administrative overhead / maintenance
  - Might not be able to do inserts / updates
    - 2012: read-only nonclustered columnstore ONLY
    - 2014: adds read-write CLUSTERED columnstore indexes but these don't allow any other indexes / keys
    - 2016+: is really a MUCH better option
- No session setting requirements

# Columnstore Indexes by SQL Server Version

- **SQL Server 2008 is the lowest (IMO) version for large tables, performance, scalability**
  - Added data compression (row and page compression)
  - Added filtered indexes / filtered statistics
  - Fixed fast-switching for partition-aligned, indexed views
- **SQL Server 2012 adds read-only, nonclustered columnstore indexes**
  - Some frustrating “batch-mode” limitations for partitioned views (UNION ALL)
    - If you’re using PVs then you should upgrade!
- **SQL Server 2014 adds updateable, clustered columnstore indexes**
  - Many of the most frustrating limitations with CS fixed – for example, UNION ALL supports batch mode (which means you can use these with partitioned views)
  - Added “incremental statistics” to help reduce when to rebuild as well as time to rebuild
- **SQL Server 2016+ takes columnstore indexes even further with updateable, nonclustered, columnstore indexes and row-based, nonclustered indexes with clustered, columnstore indexes!**

# Row-based Indexes v. Column-based Indexes

## Rowstore indexes

- **Support data compression**
  - Row compressed
  - Page compressed
- **Can support point queries / seeks**
- **Wide variety of supported scans**
  - Full / partial table scans (CL)
  - Nonclustered covering scans (NC)
  - Nonclustered covering seeks with partial scans (NC)
- **Biggest problems**
  - More tuning work for analysis: must create appropriate indexes per query and then consolidate
  - Must store the data (not as easily compressed)

## Columnstore indexes

- **Significantly better compression**
  - Columnar data stored together, often allows much higher level of compression
  - COLUMNSTORE / COLUMNSTORE_ARCHIVE
- **Supports large scale aggregations**
- **Support partial scans w/"segment" elimination**
  - Only the needed columns are scanned
  - Data is broken down into row groups (roughly 1M rows) and segments can be eliminated
  - Combine w/partitioning for further elimination
  - Parallelization through batch mode processing
- **Biggest problems**
  - Minimum set for reads is a row group (no seeks)
  - Limitations of features for batch mode by version (fixes in 2014 and 2016)
  - Limitations with other features (less and less by SQL Server version)

# 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
- **Work with your developers / architects to create better base structures**
- **Be sure to use key indexes / constraints**
  - Nonclustered for primary key (if, it's not the clustered)
  - Nonclustered for the unique keys (one might actually be your clustered?)
  - MANUALLY index your foreign keys
- **Add nonclustered for highly selective SARGs**
- **Consider covering for high priority/low selectivity**
- **Test, test, test!**

# Review

- **Indexing for performance**
  - Design strategies
  - Overall strategies
- **Using the tools for tuning**
  - SET STATISTICS IO ON
  - Showplan
  - Missing indexes
- **Indexing for AND**
- **Indexing for OR**
- **Indexing for joins**
- **Indexing for aggregates**
- **Indexed views v. columnstore indexes**
- **Rowstore indexes v. columnstore indexes**

# Questions!

SQLskills
immerse yourself in sql server