

Module 5: Indexing for Performance

**Designing for Performance
Kimberly L. Tripp**

Overview

- **Table Structure**
- **Index Concepts**
- **Index Architecture**
- **The Query Optimizer**
- **Using Showplan**
- **Isolated Query Tuning**
- **Finding the Right Balance with Workload Tuning**
- **Manually Tuning Queries**
 - **SARGs: AND/OR**
 - **JOINS**
- **Fragmentation/Maintenance**

Table Concepts

- Base unit for data storage = table
- Collection of unordered pages by default = heap
- Heap access solely through exhaustive table scans...why exhaustive?
 - No indexes means no guarantee of uniqueness
 - No order to data and no uniqueness – could be more records even if you find a match on the first row of the first page (ex. where ID = 1)
- Locking cannot guarantee data modification consistency without table level locks

Table Structure

- HEAP – A table without a clustered index
- Clustered Table – A table with a clustered index
- Non-clustered Indexes DO NOT affect the base table's structure
- However, Non-clustered Indexes are affected by whether or not the table is Clustered...

Index Concepts – Book Analogy

- Think of a book – with indexes in the back
- The book has one form of logical ordering
- For references – you use the indexes in the back... to find the data in which you are interested you look up the key
- When you find the key – you must lookup the data based on its location (“bookmark” lookup)
- The bookmark always depends on the (book) content order

Index – Species Common Name

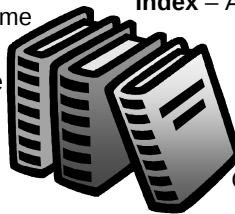
Index – Animals by Habitat, Name
Air, Land, Water

Index – Animal by Type, Name
Bird, Mammal, Reptile, etc...

Index – Species Scientific Name

Index – Animal by
Country, Name

Index – Animal by
Continent, Country, Name



Index Concepts – Tree Analogy

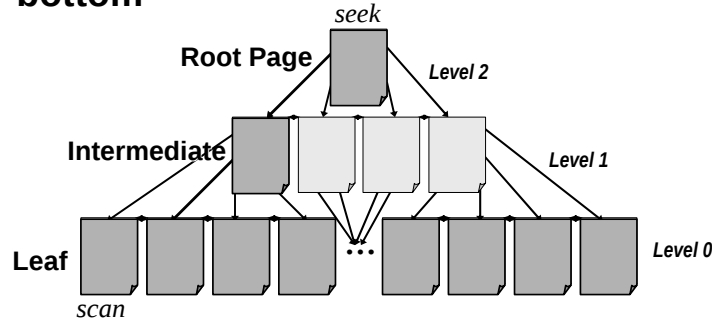
If a tree were data and you were looking for leaves with a certain property, you would have two options to find that data....

- 1) Touch every leaf – interrogating each one to determine if they held that property...SCAN
- 2) If those leaves (which had that property) were grouped such that you could start at the root, move to the branch and then directly to those leaves...SEEK



Seek vs. Scan

- **Seek** – Starts at the Root and uses the Balanced Structure to move from top to bottom



- **Scan** – Moves through the Leaf Level from Left to Right (possibly Right to Left)

Index Concepts

- **PROs**
 - Can speed up access to data – more options over exhaustive table scans
 - Can guarantee uniqueness of data
 - Can offer better lock granularity
 - Generally lead to better balanced performance – when indexed appropriately
- **Cons**
 - Add necessary overhead in INSERTs and DELETEs, add overhead in UPDATEs only when indexed column is modified
 - Add overhead in terms of Disk space
 - Add overhead in terms of Maintenance
(there are strategies for minimizing this)

Index Architecture

- **Every table should have a clustered index – choosing correct one very important!**
- **Non-clustered indexes are critical for overall performance**
- **But how do you find the right balance between over indexing and not having enough?**

Clustered Index Overview

- **Not Required – Although Highly Recommended**
- **Only One Per Table**
- **Physical Order Applied at Creation**
- **Logical Order Maintained through a Doubly-Linked List**
- **Requires ongoing and automated maintenance**
- **Key Value must be unique – either explicitly or implicitly**

Clustered Index Criteria

- **Unique**
 - Yes – No overhead, data takes care of this criteria
 - NO – SQL Server must “uniquify” the rows on INSERT. This costs time and space. Each duplicate has a 4-byte uniqifier. When CL is rebuilt all uniqifiers are regenerated causing all NC indexes to be rebuilt as well
 - **Narrow**
 - Yes – Keeps the NC indexes narrow
 - NO – Possibly wastes space
 - **Static**
 - Yes – Improves Performance
 - NO – Costly to maintain during updates to the key especially if row movement and/or splits
- In fact, an identity column that's ever increasing is ideal...*

Key Constraints Create Indexes

- **Primary Key Constraint**
 - Defaults to Unique Clustered
 - Only One Per Table

```
ALTER TABLE Employee
ADD CONSTRAINT EmployeePK
PRIMARY KEY CLUSTERED (EmployeeID)
```
- **Unique Key Constraints**
 - Default to Unique Non-Clustered
 - Up to 249 Per Table

```
ALTER TABLE Employee
ADD CONSTRAINT EmployeeSSNUK
UNIQUE NONCLUSTERED (SSN)
```
- **Foreign Key Constraints**
 - Have NEVER had indexes automatically created
 - Good idea to MANUALLY create non-unique non-clustered indexes on foreign keys to help improve join performance

Non-Clustered Index Overview

- Not Required – Although Critical to Achieving Optimal Performance
- Maximum of 249 Per Table
- Leaf Structure Separate from Base Table
- Based on the Heap's Fixed RID or Clustering Key
- Logical Order of Index Entries Maintained through a Doubly-Linked List
- By Far – the **FASTEST** Type of Range Index with very few exceptions!

*Don't ask for *, limit your queries!!!*

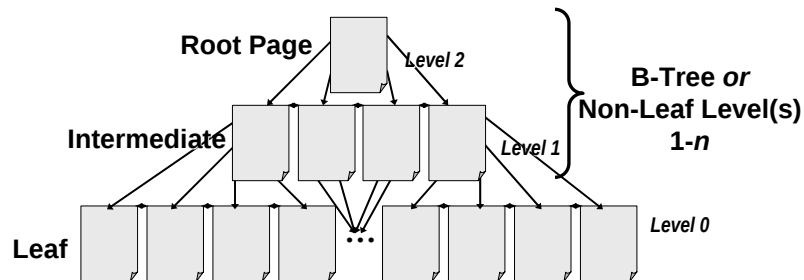
Non-clustered Covering

- Non-Clustered Indexes are BETTER for low selectivity/range queries because
 - Non-clustered indexes ARE EXACTLY the same as a clustered index – but just the columns your queries need
 - NO – this does NOT mean that you need to cover every query!!!
 - What it does mean – is that you can make one or two existing indexes slightly wider and you probably will not hurt INSERT, UPDATE, and DELETE performance and may RADICALLY improve SELECT performance

Physical Index Levels

Generic Overview

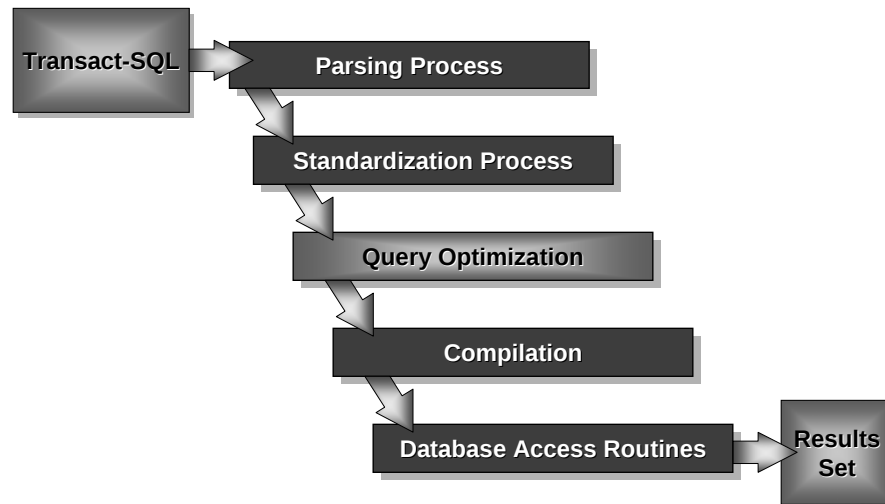
- Leaf Level - Contains one *entry* for every row of the table – in indexed order.
- Non-leaf Level(s) or Balanced-Tree - Contains one entry – specifically the FIRST entry – from every page of the level below. Always at least one non-leaf level. If only one, then it's the Root and only one page. Intermediate Levels are not a certainty.



The Query Optimizer

- Phases of Query Processing – Where is Optimization?
- Query Optimization
- Obtaining Execution Plan Information
 - Statistics statements
 - Graphical Showplan
- Index Tuning Wizard
 - Query Analyzer
 - Profiler

Phases of Processing Where is Optimization?



Query Optimization

- **Query Analysis**
 - Identifies the search and join criteria of the query
- **Index Selection**
 - Determines whether an index or indexes exist
 - Assesses the usefulness of the index or indexes
- **Join Selection**
 - Evaluates which join strategy to use

Obtaining Execution Plan Information

- Viewing STATISTICS Statements Output
 - SET STATISTICS IO ON
 - SET STATISTICS TIME ON
- Graphically Viewing the Execution Plan
- Viewing/sending Plan information, use text based showplan
 - ❖ SET STATISTICS PROFILE ON
 - SET SHOWPLAN_ALL ON
 - SET SHOWPLAN_TEXT ON

STATISTICS Statements

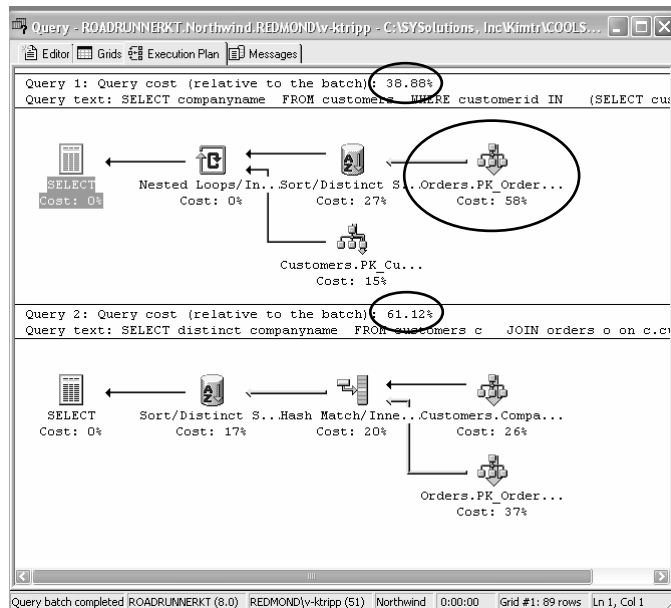
<i>Statement</i>	<i>Output Sample</i>
STATISTICS TIME	SQL Server Execution Times: CPU time = 0 ms, elapsed time = 2 ms.
STATISTICS IO	Table 'member'. Scan count 1, logical reads 23, physical reads 0, read-ahead reads 0.

- Scan Count DOES NOT mean Table Scan. Instead it means the number of times the query processor needed to access this table.
- Logical Reads are most important – they describe the path to access the data. Regardless of whether or not the I/Os are physical, if you can decrease logical you can usually improve performance. These are also consistent whereas physical I/Os are not.

Elements of the Graphical Execution Plan

- Steps Are Units of Work to Process a Query
- Sequence of Steps Is the Order in Which the Steps Are Processed – RIGHT to Left
- If multiple statements execute then each will have a percentage relative to what was sent (relative to “batch”)
- Each statement will have items that will add up to 100% for THAT statement within the batch

Reading the Plan

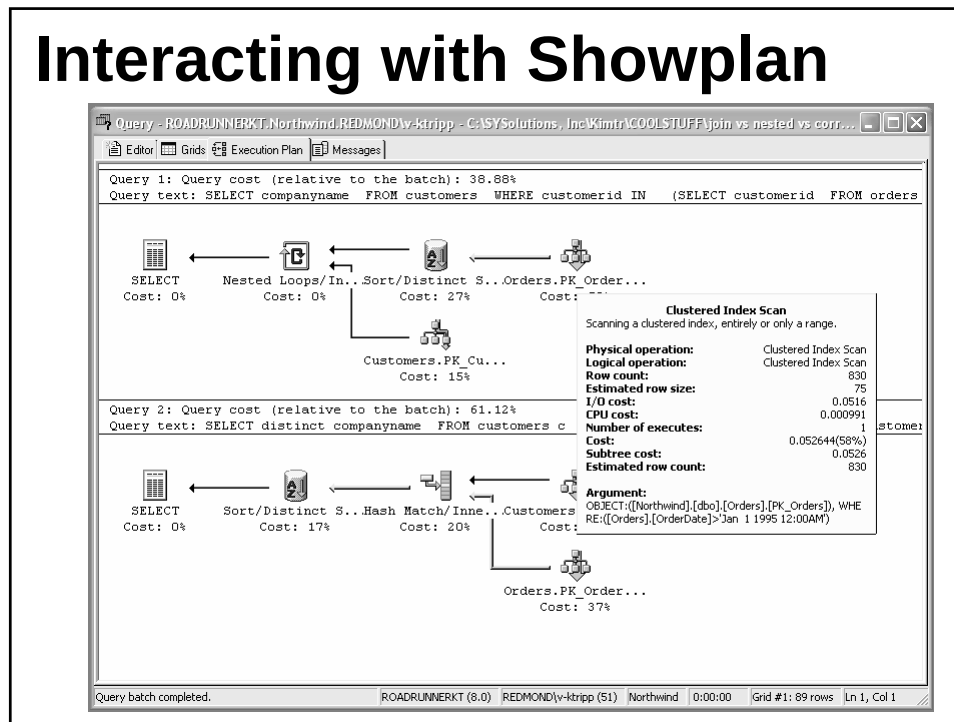


Two queries executed:

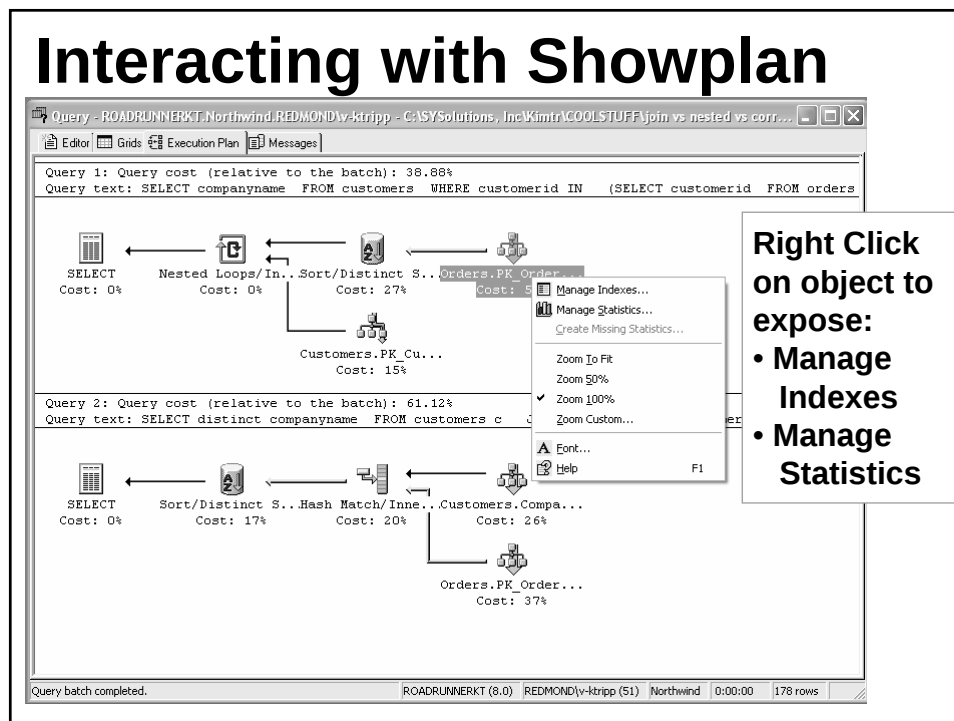
38.88% v.
61.12%

Of the 38.88% the most expensive step was the Access of Orders table

Interacting with Showplan



Interacting with Showplan



Query Tuning

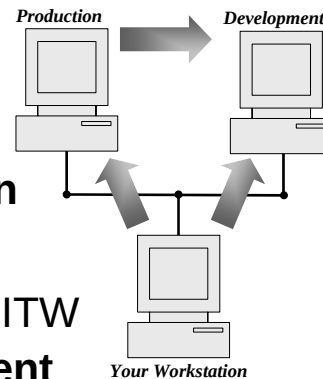
- Simple Strategy - ITW
- Understanding Selectivity
- How to Improve Queries with Varying Search Arguments (SARGs)
 - Indexing for AND
 - Indexing for OR
- How to Improve Joins

Index Strategies

- Determine Primary Usage of Table – OLTP vs. OLAP vs. Combo? This determines Clustered Index
- Create Constraints – Primary Key and Alternate/Candidate Keys
- Manually Add Indexes to Foreign Key Constraints
- Capture a Workload and Run through Index Tuning Wizard
- Continue to test, tune and troubleshoot and Add Additional Indexes using these strategies

Finding the Right Balance Using Index Tuning Wizard

- **Create a TRACE of your Production Server from your main workstation**
Realistic Sample Workload
- **Backup/Restore Production Data to Development Environment**
Real Data/Real Statistics for ITW
- **Point Profiler at Development Server in order to perform Index Tuning Wizard**
Does not Impact Production Server



Could the Query go Faster?

- ITW in Query Analyzer is ISOLATED query tuning
- Tuning a query to the n^{th} degree is NOT always a good idea
- Great way to specifically see if a query can be helped... might even run it multiple times
- Each additional time you should use optimizer hints to see if (and by how much) the query improves

Isolated Query Tuning

- Review the plan, keep track of all indexes used (this will be helpful for comparisons)
- Highlight the query in QA
- Run ITW (Ctrl + I)
- Next until you can next no more ☺
- Choose “Select All tables” then next
- Do you see an improvement?
- Can review it and/or just apply the changes...

Isolated Query Tuning

Did it work?

- Apply changes
- Go back to query window
- Copy original query so there are two copies
- Modify the first query to have the same plan as first execution by adding an optimizer hint to force the same index choice
 - WITH (INDEX (index_name))
- Run both queries and compare!

Manually Tuning Queries

Understanding Selectivity

- Not just based on the Number of Rows Returned
- Always Relative to the Number of Rows in the Table (usually expressed as a percentage)
- Low Number = High Selectivity
 - Any Index is Useful if even ONE condition is highly selective!
- High Number = Low Selectivity
 - This is harder!

Low Selectivity Queries

- Limited Select List
- Index On Columns Requested

```
SELECT LastName, FirstName, PhoneNo
FROM dbo.Member
WHERE LastName LIKE '[S-Z]%'
-- 10000 Rows, 3072 in S-Z Range
```

Covering
*A Mini Clustered Table of
 Just the Data You Need!*

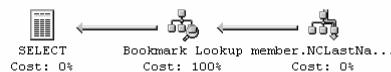
Options to Access Data

- **Table Scan**
- **Nonclustered on LastName**
 - **Bookmark Lookups for Every Row**
- **Nonclustered on LastName, FirstName and PhoneNo**
- **Nonclustered on FirstName, LastName, PhoneNo**

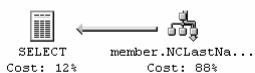
Query 1: Query cost (relative to the batch): 2.35%
 Query text: SELECT LastName, FirstName, Phone No FROM Member



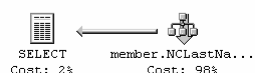
Query 2: Query cost (relative to the batch): 96.49%
 Query text: SELECT LastName, FirstName, Phone No FROM Member



Query 3: Query cost (relative to the batch): 0.25%
 Query text: SELECT LastName, FirstName, Phone No FROM Member



Query 4: Query cost (relative to the batch): 0.91%
 Query text: SELECT LastName, FirstName, Phone No FROM Member



IO Costs

Table Scan
= 184 Reads

NC LastName
= 6354 Reads

NC Covering Seek
= 19 Reads

NC Covering SCAN
= 59 Reads

Indexing for AND

- **AND Progressively Limits the SET**
- **All Conditions MUST be true**
- **Find the SMALLEST Set and work from there.**
- **Evaluate Columns in WHERE**
 - **A Single Highly Selective Condition**
 - **Any combination of Highly Selective Criteria**
- **If NOTHING Yields a Selective Set Consider Covering**

Indexing for OR

- **OR Is Similar to UNION**
- **ANY Condition must be true**
- **Optimally find each set or scan for the whole thing – requires useful indexes for EACH AND EVERY condition**
- **Consider re-writing as UNION**
 Include a Row Identifier – such as the Primary Key – to get same results (OR only yields the same result set as UNION if the SELECT list has a UNIQUE row identifier)
- **Test!**

Indexing for Joins

- Tables are Joined Two Tables at a Time
- Each Table has a Join Condition
- Optionally Each Table has a SARG
- Usually the Join is Between the Primary and Foreign Key
- Always Try to Give SQL Server as Many Options to Choose From...

Best Options for Joins



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 SARG1 to Table2 Join
- Another could use Table2 SARG2 to Table1 Join
- Another could use the Join Indexes on Both Tables
- BUT if ALL 4 Indexes are there then the Join has the best Options to choose from!

Cover the Combination



SARG1
Join Col PK



SARG2
Join Col FK

Still not working?

- **Not Using Any of These Indexes?**
- **Performance Still Awful?**
- **Cover the Combo – Create 4 Indexes (2 Per Table)**
 - Table1 (SARG, Join) – Priority to the Search Condition
 - Table1 (Join, SARG) – Priority to the Join Condition
- **Test!**

Cover One/Both of the Tables



SARG1
Join Col PK



SARG2
Join Col FK

Still not working?

- **Not Using Any of These Indexes?**
- **Performance Still Awful?**
- **Try Covering the queries – Create Indexes to Cover the Individual Tables requested...**
 - Trying them in both priority orders:**
 - **Priority for the Join**
 - **Priority for the SARG**

Finding the Right Balance

Index Strategies – Summary

- Determine Primary Usage of Table – OLTP vs. OLAP vs. Combo? This determines Clustered Index
- Create Constraints – Primary Key and Alternate/Candidate Keys
- Manually Add Indexes to Foreign Key Constraints
- Capture a Workload(s) and Run through Index Tuning Wizard
- Are you done? **NO!**

Keeping Performance Optimal

- For the Optimizer to use indexes appropriately you must have Statistics!
 - Auto Update Statistics
 - Auto Create Statistics
- For the database to stay compact and for indexes to stay efficient you must maintain indexes:
 - Defrag indexes
 - + Doesn't take the table offline
 - - Creates a lot of log space
 - Rebuild indexes
 - - May take the table offline (depends on index type)
 - + Completely restructures an index

Index Maintenance

	Option	If Used with Clustered Index Rebuilds NonClustered	Updates Statistics	Lock on Base Table for CL	Lock on Base Table for NC	Transaction Log Impact
Rebuilding	DROP and re-CREATE	Yes, twice	Yes	Exclusive	Shared	Longest Time, Most Activity
	DBCC DBREINDEX	Pre-sp2 YES (and only once) 2000sp2 NO - unless the CL Key is NON-unique. When the CL is not unique, the NC indexes are rebuilt EACH time the CL is rebuilt (the unqiifiers are regenerated).	Yes	Exclusive	Shared	Cannot Clear Until Complete
	CREATE with DROP_EXISTING	Same as above EXCEPT when the Clustering Key CHANGES - then all NC indexes are rebuilt only once. Otherwise, NO*	Yes	Exclusive	Shared	Cannot Clear Until Complete
	DBCC INDEXDEFRAG	No	No	N/A	N/A	Minimal Impact in sp1+
* In SQL Server 2000 sp2 DBCC DBREINDEX and CREATE with DROP_EXISTING are almost identical. The only exception is that CREATE with DROP_EXISTING can be used to CHANGE the definition of the clustered index.						

**See DBM303: Preserving SQL Server Performance
through Proper Index Maintenance**

Key Points

- **Choose Clustered Index Wisely!**
 - Avoid wide clustering keys
 - Consider clustering on an identity column
- **Add non-clustered for better performance**
 - Manually index foreign key
 - Use isolated query tuning for some
 - Use ITW with workload for most
- **Talk to your users – understand data access patterns and usage**
- **Create Indexes That Support Search Arguments**
- **Put Maintenance Jobs into place**

Review

- **Table Structure**
- **Index Concepts**
- **Index Architecture**
- **The Query Optimizer**
- **Using Showplan**
- **Isolated Query Tuning**
- **Finding the Right Balance with Workload Tuning**
- **Manually Tuning Queries**
 - **SARGs: AND/OR**
 - **JOINS**
- **Fragmentation/Maintenance**