SQL Server 2005: Tips and Tricks to Tuning for High Performance

SPR302

Kimberly L. Tripp
SQLskills.com
Kimberly@SQLskills.com

Kimberly L. Tripp

- Consultant/Trainer/Speaker/Writer
- Founder, ***SYS*olutions, Inc.** (www.SQLskills.com)
  - e-mail: Kimberly@SQLskills.com
  - blog: http://www.SQLskills.com/blogs/Kimberly
- Microsoft Regional Director and SQL Server MVP
- Writer/Editor for SQL Magazine www.sqlmag.com
- Author of several SQL Server 2005 Whitepapers on MSDN/TechNet – Table and Index Partitioning, Snapshot Isolation and .NET SQLCLR for DBAs
- Author/Presenter for more than 25 online webcasts on MSDN and TechNet
- Coauthor MSPress Title: *SQL Server 2000 High Availability*
- Presenter/Technical Manager for SQL Server 2000 High Availability DVD
- I still love this stuff… Feel free to ask questions!

Session Overview

- Methodologies
- Optimization and Data Access Patterns
- General Strategies for Tuning
- Designing for Performance
- Indexing for Performance
- Optimizing Procedural Code

Methodologies

What impacts performance?

- The “Process”
- The Application
- The Database
- The Software
- The Hardware

Where do you start?

How much effort?

Costs?

Is hardware the problem or only a symptom of another problem?

Which give you the most gains?

The “Process”

- Process improvements can yield some of the biggest gains…can you streamline your business needs?
- How can the process be improved?
  - Can steps be eliminated
    - Are you returning data within the transaction…
    - Verifying data before modification
  - Can steps be removed from the “online” and transactional process
    - Does an entire order need to be processed in one transaction?
  - Can you compartmentalize pieces of the transaction

NOTE: Consider SODA – Service Oriented Database Architectures

The “Process”

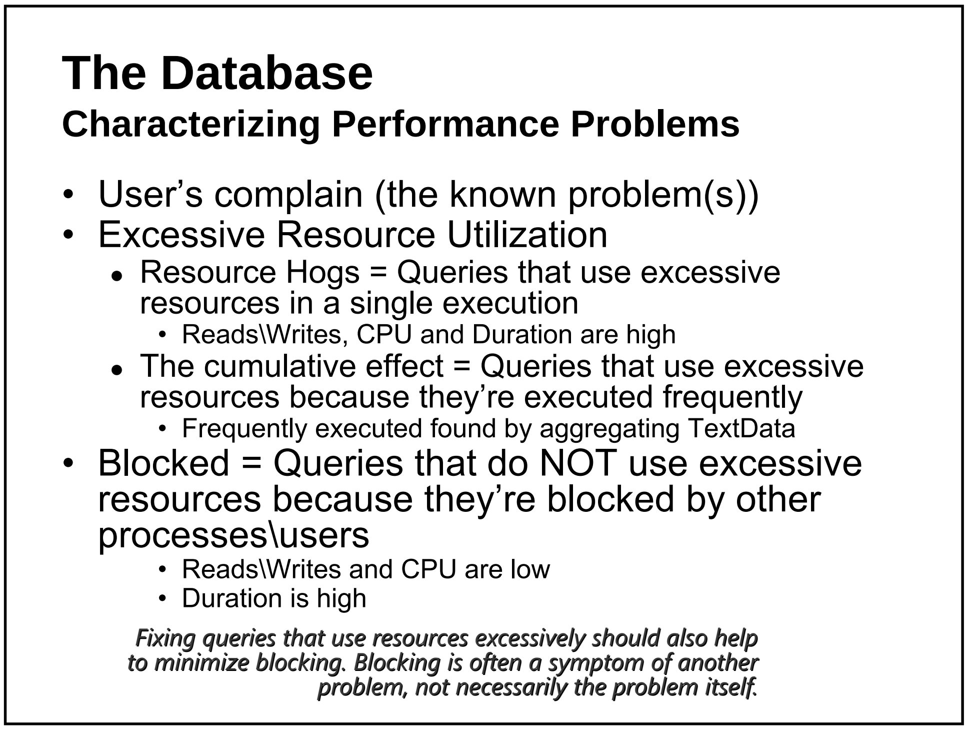
“Technology is nothing more than an enabler of a business process. If your process isn’t efficiently designed and you don’t have the data available when you need it, what you’ve done is used technology to drive up costs.”

Cathy Ellwood
Director of Corporate Strategy
Nationwide Insurance

The Application

- Ask a bad question…
- SELECT *
  - Limit the columns selected
  - Limit the rows returned
- Isolating the column to one side of the expression
  - Use MonthlySalary > value/12 (seekable)
  - NOT MonthlySalary * 12 > value (must scan)
- Too many round-trips
- Poorly designed transactions – waiting for user input
- Too much “expertise” in the application language – minimal in SQL Server (misunderstanding where\what the server should do v. the application)

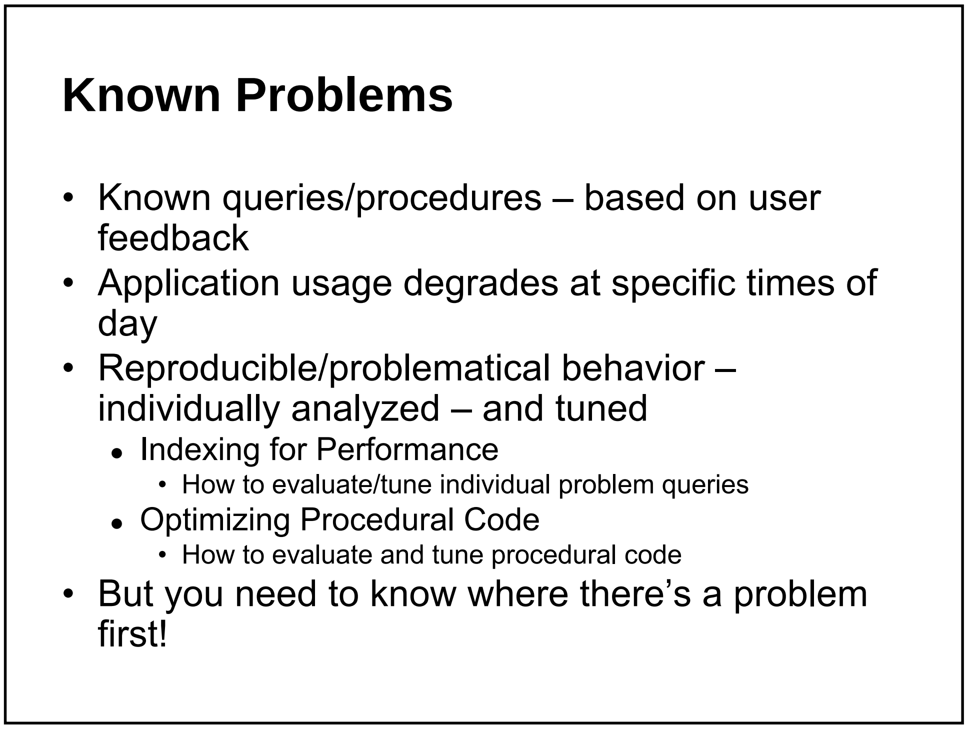
SQL skills.com



The Database
Characterizing Performance Problems

- User’s complain (the known problem(s))
- Excessive Resource Utilization
  - Resource Hogs = Queries that use excessive resources in a single execution
    - Reads\Writes, CPU and Duration are high
  - The cumulative effect = Queries that use excessive resources because they’re executed frequently
    - Frequently executed found by aggregating TextData
- Blocked = Queries that do NOT use excessive resources because they’re blocked by other processes\users
    - Reads\Writes and CPU are low
    - Duration is high

Fixing queries that use resources excessively should also help to minimize blocking. Blocking is often a symptom of another problem, not necessarily the problem itself.



Known Problems

- Known queries/procedures – based on user feedback
- Application usage degrades at specific times of day
- Reproducible/problematical behavior – individually analyzed – and tuned
  - Indexing for Performance
    - How to evaluate/tune individual problem queries
  - Optimizing Procedural Code
    - How to evaluate and tune procedural code
- But you need to know where there’s a problem first!

Unknown Problems

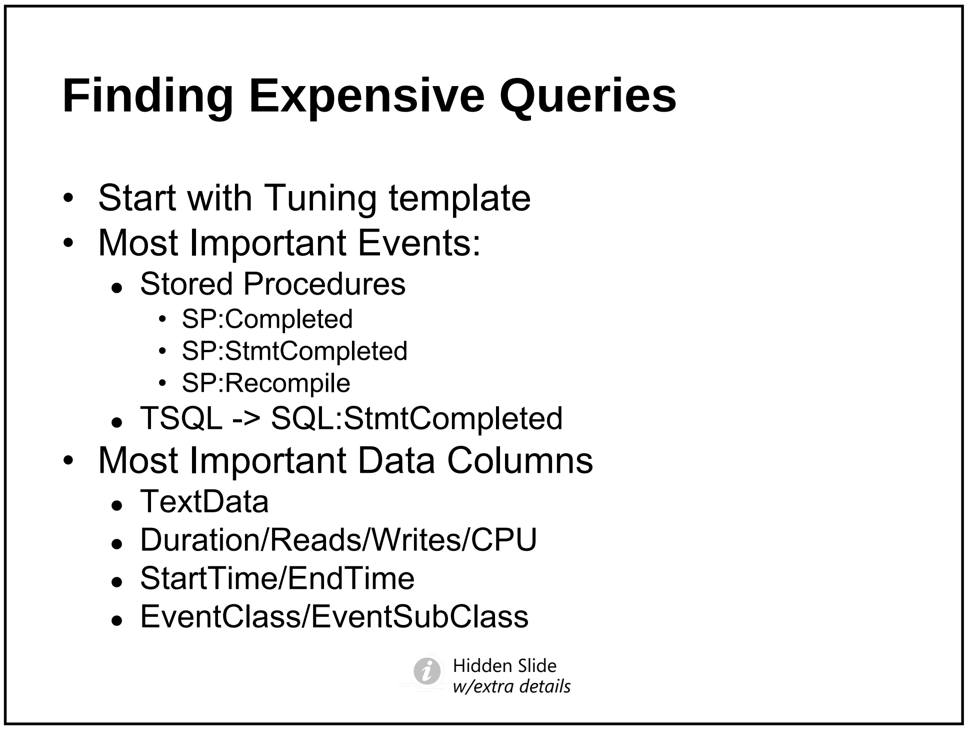
- Expensive Queries
  - Determined by excessive durations
    - Locking problems?
    - Excessive resource access?
  - Use the same practices to tune these as known problems
  - Use Profiler to find them (filter on duration)
- Frequently executed queries
  - Which would you tune?

```
SELECT * FROM Table1 WHERE Col3 = 12
```
or
```
SELECT * FROM Table6 WHERE Col4 = 19
```
What if you knew that Query 2 was run more than 1000 times per hour…

Profiler

Check out MSDN Webcast, Part 9
for details/demos on
Server-side Trace Queues

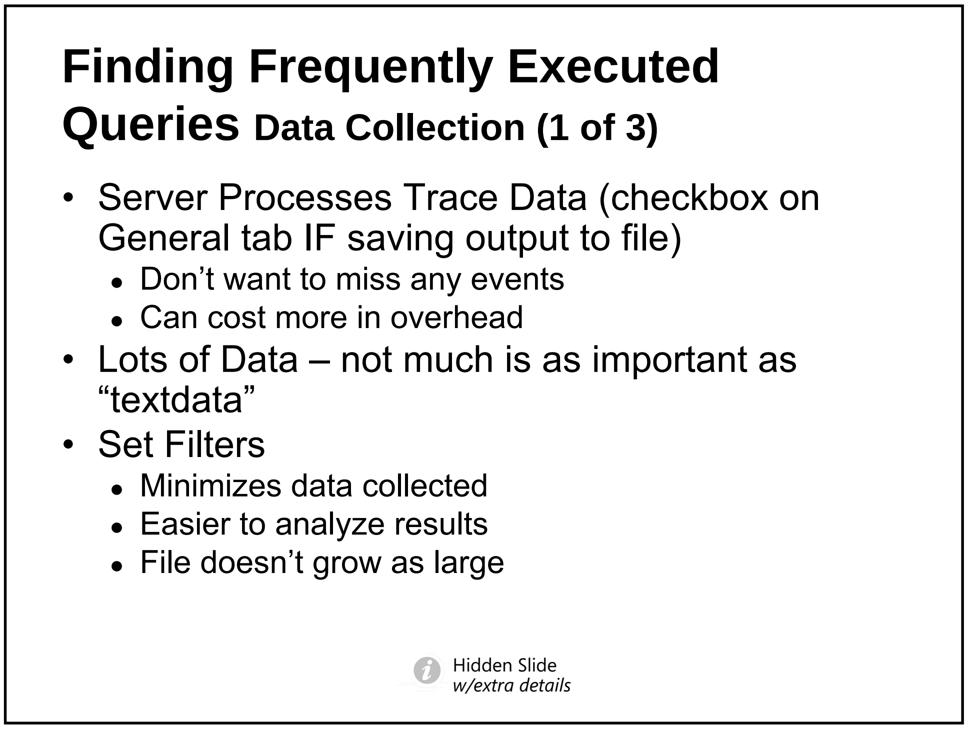
- Get a feel for how Profiler works…play!
- Do not add too many events/data columns
  - Too many can negatively impact performance
  - Event type can also negatively impact performance
    - The more “internal” the type, the more expensive
    - TSQLBatchCompleted
    - SP:Completed
    - SP:StmtCompleted
      - …
    - Lock:Acquired (unless filtered – a lot of data/overhead)
- Don’t get overwhelmed by Events and Data Columns – you don’t need most of them most of the time!



Finding Expensive Queries

- Start with Tuning template
- Most Important Events:
  - Stored Procedures
    - SP:Completed
    - SP:StmtCompleted
    - SP:Recompile
  - TSQL -> SQL:StmtCompleted
- Most Important Data Columns
  - TextData
  - Duration/Reads/Writes/CPU
  - StartTime/EndTime
  - EventClass/EventSubClass

Hidden Slide
w/extra details



Finding Frequently Executed Queries **Data Collection (1 of 3)**

- Server Processes Trace Data (checkbox on General tab IF saving output to file)
  - Don’t want to miss any events
  - Can cost more in overhead
- Lots of Data – not much is as important as “textdata”
- Set Filters
  - Minimizes data collected
  - Easier to analyze results
  - File doesn’t grow as large

Hidden Slide
w/extra details

Finding Frequently Executed Queries Filters (2 of 3)

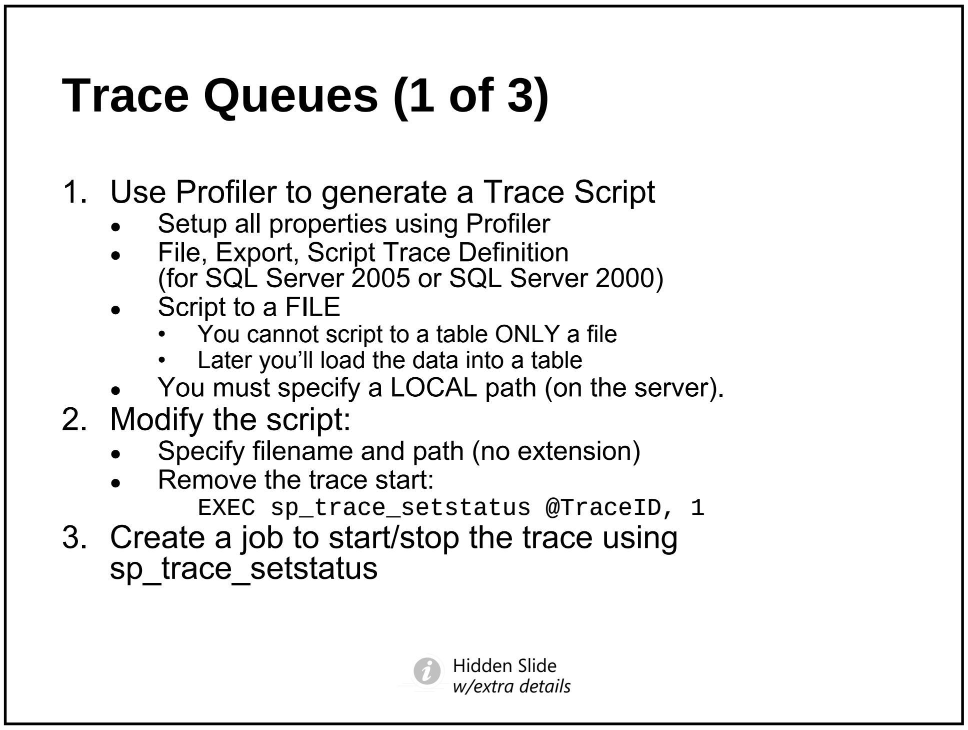
- Exclude system IDs by excluding all object ids less than 100 (and new filter IsSystem = 0 for user)
- DatabaseName – only the database in which you are interested
- And add a few more to remove other non-essential data…do a few tests – what do you see that you don't need?
- May want to remove sp_% or at least sp_sqlagent_% (you'll learn what your server does that you don't want to see after a few iterations and tests)

Hidden Slide
w/extra details

Finding Frequently Executed Queries Analysis (3 of 3)

- Save Trace to a File (using Trace Queue)
  - sp_trace_create
  - sp_trace_setstatus
- Later pull results into a table
  - fn_trace_gettable
  - See KB 270599: How to Programmatically Load Trace Files into Tables
- Aggregate textdata by a substring of characters to find "query classes"

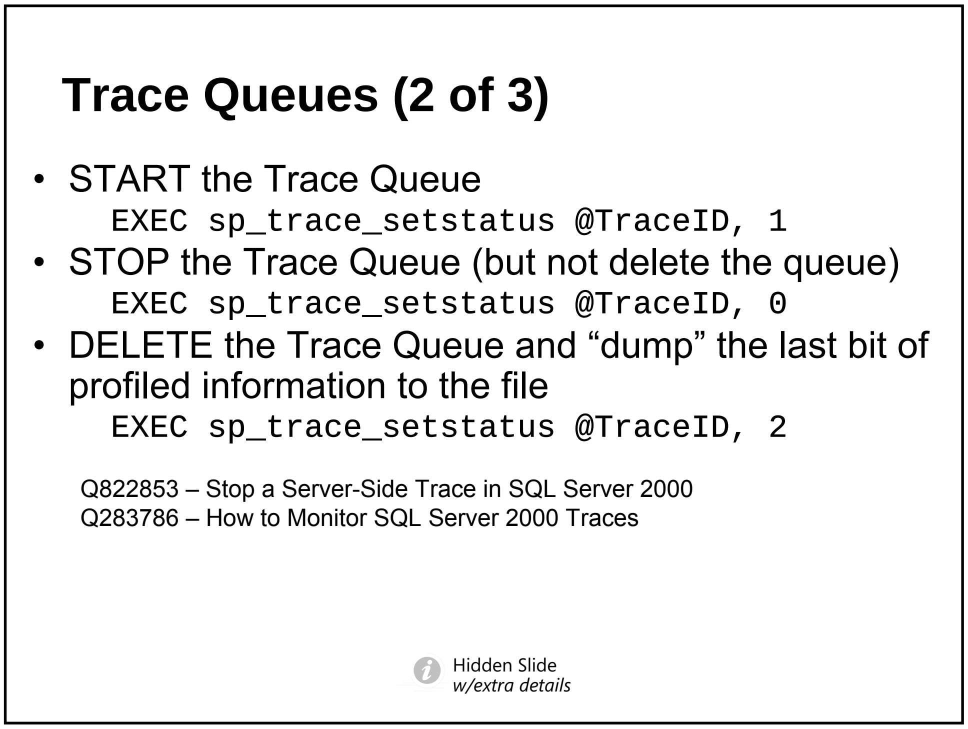
Hidden Slide
w/extra details



Trace Queues (1 of 3)

1. Use Profiler to generate a Trace Script
   - Setup all properties using Profiler
   - File, Export, Script Trace Definition
     (for SQL Server 2005 or SQL Server 2000)
   - Script to a FILE
     - You cannot script to a table ONLY a file
     - Later you'll load the data into a table
   - You must specify a LOCAL path (on the server).
2. Modify the script:
   - Specify filename and path (no extension)
   - Remove the trace start:
     `EXEC sp_trace_setstatus @TraceID, 1`
3. Create a job to start/stop the trace using sp_trace_setstatus

Hidden Slide
w/extra details

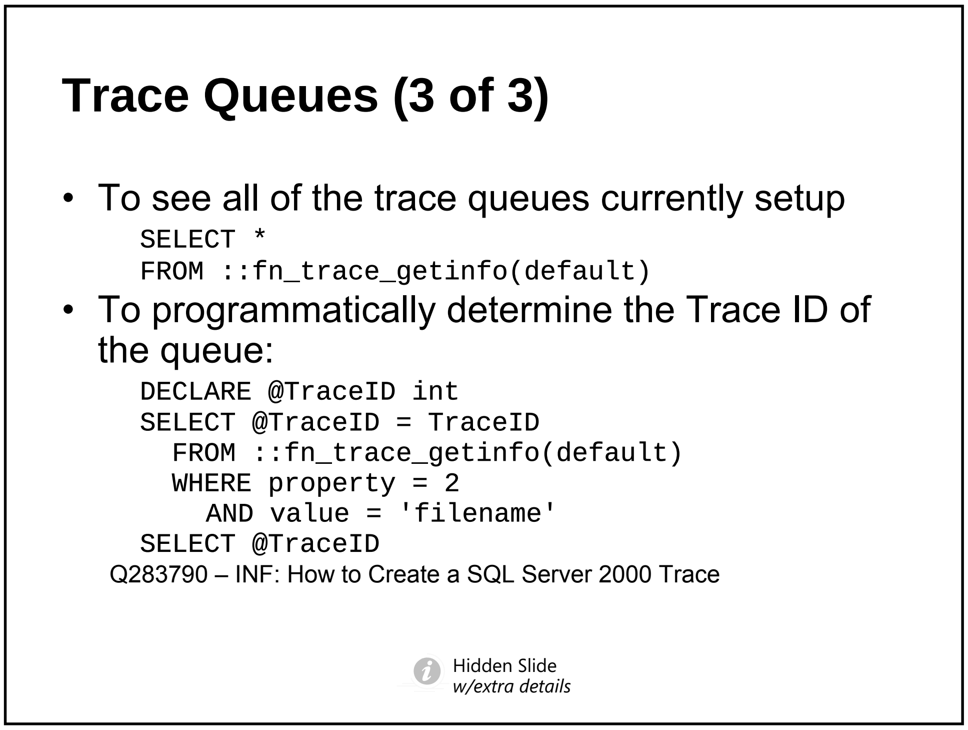


Trace Queues (2 of 3)

- START the Trace Queue
  `EXEC sp_trace_setstatus @TraceID, 1`
- STOP the Trace Queue (but not delete the queue)
  `EXEC sp_trace_setstatus @TraceID, 0`
- DELETE the Trace Queue and "dump" the last bit of profiled information to the file
  `EXEC sp_trace_setstatus @TraceID, 2`

Q822853 – Stop a Server-Side Trace in SQL Server 2000
Q283786 – How to Monitor SQL Server 2000 Traces

Hidden Slide
w/extra details



Trace Queues (3 of 3)

- To see all of the trace queues currently setup

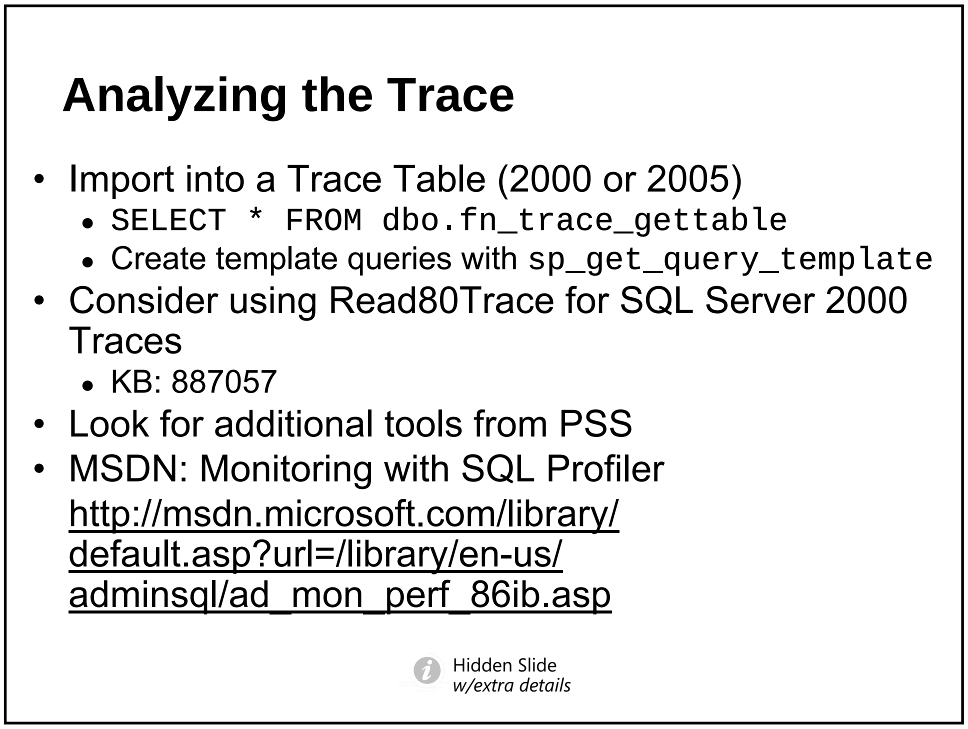
```
SELECT *
FROM ::fn_trace_getinfo(default)
```

- To programmatically determine the Trace ID of the queue:

```
DECLARE @TraceID int
SELECT @TraceID = TraceID
  FROM ::fn_trace_getinfo(default)
  WHERE property = 2
    AND value = 'filename'
SELECT @TraceID
```

Q283790 – INF: How to Create a SQL Server 2000 Trace

Hidden Slide
w/extra details



Analyzing the Trace

- Import into a Trace Table (2000 or 2005)
  - `SELECT * FROM dbo.fn_trace_gettable`
  - Create template queries with `sp_get_query_template`
- Consider using Read80Trace for SQL Server 2000 Traces
  - KB: 887057
- Look for additional tools from PSS
- MSDN: Monitoring with SQL Profiler
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/adminsql/ad_mon_perf_86ib.asp

Hidden Slide
w/extra details

Application Profiling

- Remember – Profiler Only helps you "Fish"
- Finding the problem is sometimes a bigger problem!
  - ➯ SQL Server Profiler
- Limit the scope – stay focused on problem!
- Consider automating Traces through SQL Server Agent
  - Script Trace using Trace Queue
  - Automate job to use `sp_trace_setstatus` to programmatically start and stop trace
  - Consider Alerts/`RAISERROR` and/or `WAITFOR DELAY` to create patterns (when not a continuous trace)
- What about solving the problem?

Optimization & Data Access Patterns

Consider Data Access Strategies

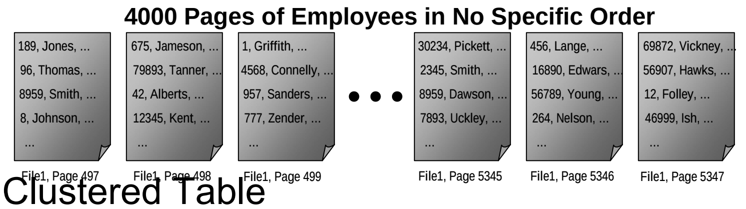
- How would you process data?
  - Table Scan
  - Clustered Index Seek
  - Non-clustered Index Seek to Data Lookup
  - Understanding Covering
- How does SQL Server Know?
- How can you make sure that SQL Server has the best information?

Table Scan in Heap v. Clustered

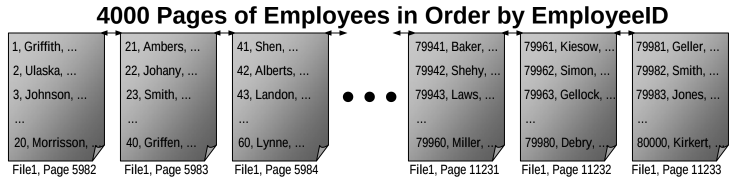
- Heap Structure
  - Table without a Clustered Index
  - Records are NOT ORDERED
  - No Doubly-Linked List
  - If NO Indexes exist – a full Table Scan required. At least 4000 I/Os on the Employee Table Heap
- Clustered Table
  - Table with a Clustered Index
  - Records are ordereded by the clustering key
  - Structured as a B-Tree with doubly-linked lists
  - Exactly 4000 I/Os if 4000 pages in the data (leaf) level

Table Scan in Heap v. Clustered

- Heap Structure



- Clustered Table



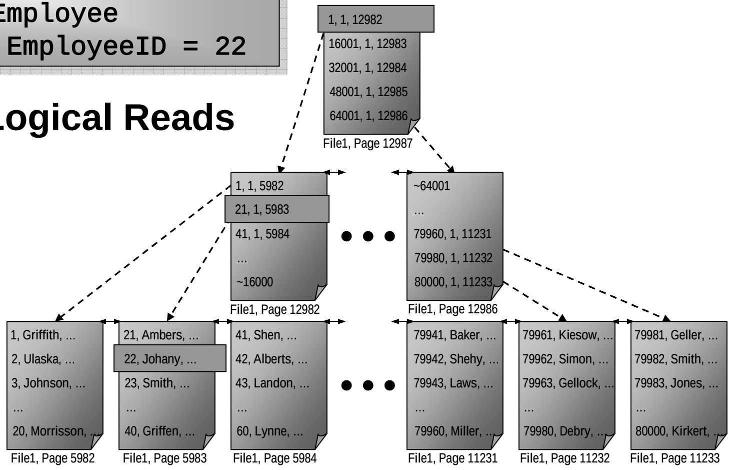
demo

The Added Cost of Forwarding Pointers

Clustered Index to Seek

```
SELECT *
FROM Employee
WHERE EmployeeID = 22
```

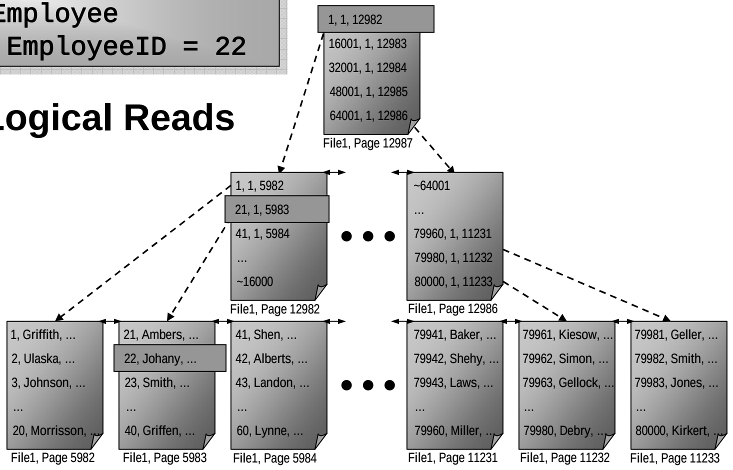
= 3 Logical Reads



Clustered Index to Seek

```
SELECT *
FROM Employee
WHERE EmployeeID = 22
```

= 3 Logical Reads



Non-clustered Index to Data Lookup

```
SELECT *
FROM Employee
WHERE SSN = '123-45-6789'
```

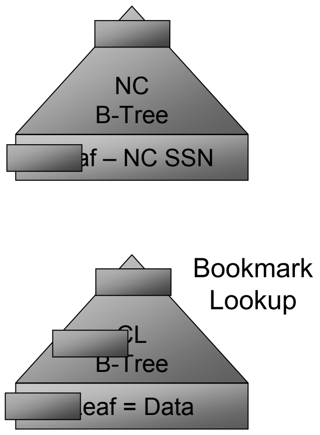
Root, then Leaf in NC Index on SSN to yield EmployeeID

= 2 Logical Reads

Root, intermediate, leaf in CL to yield *

= 3 Logical Reads

Total of 5 Logical Reads



Query Specific Index Usage

```
SELECT *
FROM Employee
WHERE SSN BETWEEN '123-45-6789' AND '123-45-6800'
```

Assumption – 12 Rows

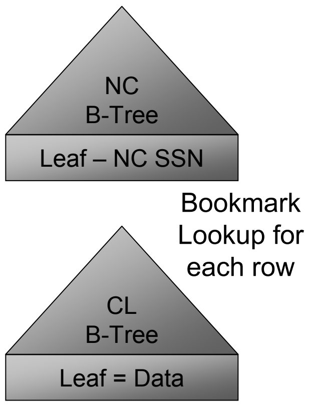
Root, then Leaf in NC Index on SSN to yield 12 EmployeeIDs

= 2 or 3 Logical Reads

Root, intermediate, leaf for each row to access data in CL yields

= 3*12 Logical Reads

Total of 38/39 Logical Reads



Query Specific Index Usage

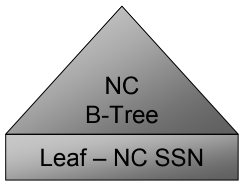
At what Range would Bookmark Lookup be useless?

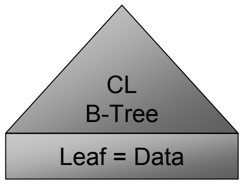
```
SELECT *
FROM Employee
WHERE SSN x AND y
```

n Rows

Table Scan = 4000 Pages
When *n Rows* >= 4000
Table Scan seems better!

Probably well before 4000 sequential reads are better than ?? random reads.





At what percentage is a bookmark lookup TOO expensive?

What is selective enough?

a) 60 percent
b) 30 percent
c) 20 percent
d) 10 percent
e) 5 percent
f) 1 percent
g) None of the above are selective enough?

The real answer is that it depends on the table and it’s all relative to row size. The “selectivity” threshold varies table to table BUT – the closest correct answer is _____.

demo

Bookmark Lookups are too expensive at what percentage

Understanding Covering

Key Points

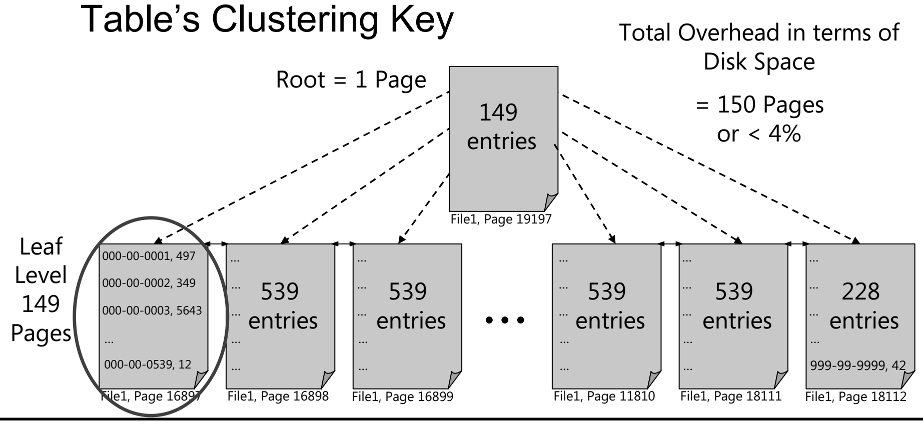
- Only nonclustered indexes are considered "covering" indexes. A clustered index covers everything – but is unnecessarily wide (wrt covering)
- The leaf level of a nonclustered index contains one entry for *every* row of the table ALL the time. Every insert and every delete impacts every nonclustered index on that table
- An index covers a query when the index includes all columns referenced by the query – somewhere in the index; column order is irrelevant (with regard to the definition of covering)

Understanding Covering
Internal Structures

- Leaf level contains one “entry” for every row of the table – in indexed order
  - Without new INCLUDE feature
    - Leaf level “entry” is the index key + bookmark lookup value (RID if head, Clustering Key if table is clustered)
  - With new INCLUDE
    - Leaf level “entry” is the index key + bookmark lookup value (RID if head, Clustering Key if table is clustered) + non-key columns included solely to help create more covering
- The nonclustered index effectively becomes the SAME as a clustered index but with fewer columns than the table
- Nonclustered indexes are a mini version of the table

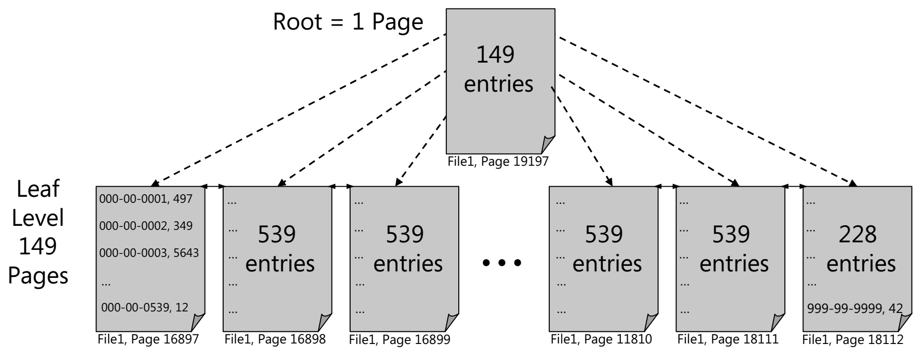
Nonclustered Index
Unique Key SSN

- Leaf level contains the nonclustered key(s) – index indexed order
- Includes either the Heap’s Fixed RID or the Table’s Clustering Key



What if you didn’t know? Unique Key SSN

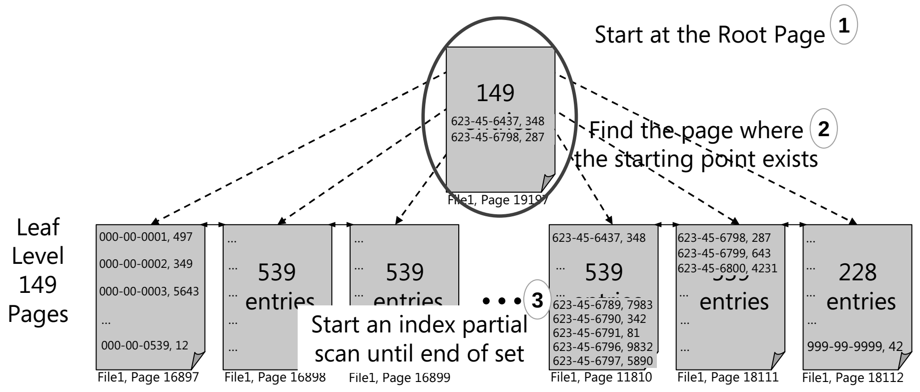
- Could this structure be anything else?
- What if you created a table with JUST EmpID and SSN and then clustered it on SSN



Nonclustered Index

Fairly Obvious Index Access

```
SELECT EmpID, SSN
FROM Employee
WHERE SSN BETWEEN ‘623-45-6789’
      AND ‘623-45-6800’
```



Similar Query – How to Process?

Less Obvious Index Access

```
SELECT EmpID, SSN
FROM Employee
WHERE EmpID < 10000
```

- Clustered Index on EmpID = 500 I/Os
  - Seekable with Partial scan
  - If the table has 80,000 rows at 20 rows per page then the table has 4,000 pages. If 10,000 is 1/8 of 80,000 then this SEEKABLE query will cost 1/8 of the 4,000 pages.
- Nonclustered Index on SSN, EmpID = 149 I/Os
  - Not seekable, must scan
  - If the leaf level has 80,000 rows at 539 rows per page then the leaf level of the nonclustered index has 149 pages. If this index is not seekable, then we must scan all 149 pages.

Other Less Obvious Index Access Patterns

- Scanning nonclustered indexes to cover the query
- Nonclustered Index Intersection to join multiple indexes to cover the query
- Hash aggregates to scan and build aggregates out of order – still better to scan a covering index rather than a clustered

- Certainly, the BEST case is when you cover ONLY the necessary data AND it's seekable

But you can't cover everything…

INCLUDE to add non-Key Columns in the Leaf Level of nonclustered Index

- Key is limited to 900 bytes or 16 columns (whichever comes first)
  - Allows the tree to be more scalable
  - Used to apply to the entire tree
- Leaf-level can include non-key columns – with NO limitations (can include LOB types – use sparingly!)
  - Allows the leaf level to cover more queries
  - Can cover ANYTHING
- In SQL Server 2005 you CAN cover anything and everything… but just because you can, should you?

Indexed Views

- A data set, defined by the SELECT statement of the view, that has been duplicated in the leaf level of a clustered index (*materialized view*)
- Not every view *can* be indexed
- Not every view *should* be indexed
- Allows a "wider" leaf level because only the key is limited to 900 bytes or 16 columns (whichever comes first)
  - If the Indexed View is against only a single table – and does not include functions, computations, or aggregations, then it's exactly the same as INCLUDE
  - But an indexed view can include all of those and more…

Many ways to Cover Queries

- Clustered Index – always covers (only one way to seek or partially scan, full scan is the most expensive here as it's the entire row/set)
- Nonclustered indexes – covers the query with narrower rows = "just the data you need"
- Nonclustered indexes with INCLUDE – can allow you to cover ANY query and can even include LOB types in the leaf level of the index
- Indexed Views – can cover wide queries (as does include) but these can include joins, aggregates, computations, deterministic functions, etc.
- Do you need to cover EVERY query?

NO!

General Strategies for Tuning

How do you fix the problem?

- Things to Try Before Adding More Indexes
- DTA for Workload Tuning
- Manually
  - Rewrite Transactional Code/Queries
  - Designing for Performance
  - Indexing for Performance
  - Optimizing Procedural Code

Things to Try BEFORE Adding More Indexes (1 of 3)

1. Are your statistics up to date? (use `stats_date` function)
   - If not, then **`UPDATE STATISTICS tablename`** and try again
   - Did that help?
     - Yes – then make sure that statistics are being updated regularly
       - Index rebuilds update statistics (**`ALTER INDEX REBUILD`**)
       - Statistics can be updated manually (**`UPDATE STATISTICS`**)
     - No – did the statistics update based on a full scan or a sampling (use **`DBCC SHOW_STATISTICS`** to see)
       - If a sampling was used, try updating with a full scan (**`UPDATE STATISTICS tablename WITH FULLSCAN`**) and try again… did that help?

Things to Try BEFORE Adding More Indexes (2 of 3)

- Is this in a stored procedure?
  - sp_recompile procedure to see if it changes based on the prior index adjustments
  - EXECUTE procedure param, param, param, … WITH RECOMPILE
    - If the problem goes away, might need statement (SQL Server 2005 ONLY) or procedure recompilation (see MSDN Webcast Part 7)
  - Might be recompiling too much
    - Helpful KBs in this area: Q243586, Q308737

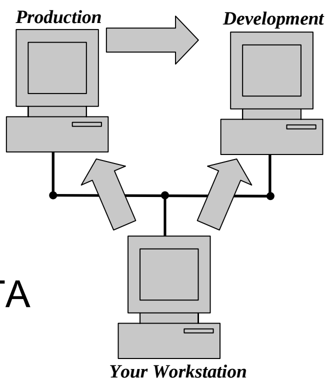
Things to Try BEFORE Adding More Indexes (3 of 3)

- Consider rewriting the code/query
  - Temp table v. views v. derived table v. table variable
  - If written as a join, try rewriting as a subquery
  - If written as a subquery, try rewriting as a join
  - If includes an OR SARG, consider rewriting with UNION or UNION ALL (more on this coming up!)
  - Using Cursors? Can you use SET algorithms instead

Finding the Right Balance

Workload Tuning with Database Tuning Advisor

- Create a TRACE of your Production Server (on the server or from your pc)
  Realistic Sample Workload
- Backup/Restore Production Data to Development Environment
  Real Data/Real Statistics for DTA
- Point Profiler at Development Server in order to perform Database Tuning Advisor
  Does not Impact Production Server



Database Tuning Advisor

Simple Workload

- Create a workload using Tuning template
  - Defines the Events to monitor
  - Defines the Data Columns to monitor
- Capture a realistic workload to a FILE
- Set a max file size < 1GB and do NOT enable "rollover" as it creates a NEW file once the first file fills and could fill your harddrive
- Set filters
  - DatabaseName LIKE YourDBName
  - Exclude system objects

SQL skills.com

Executing Database Tuning Advisor
Simple Route (1 of 2)

- Launch DTA from…
  - Profiler: Tools, Database Tuning Advisor
  - SSMS: Tools, Database Tuning Advisor
  - Launch Database Tuning Advisor Application
- Select the DB to tune (DTA can tune multiple databases and tune a workload that spans databases)
- For the simple route, keep all of the default values
  - Default values are a great start and will likely find some issues/problems…

Executing Database Tuning Advisor
Simple Route (2 of 2)

- Select to tune "all tables"
- Let DTA generate recommended indexes and then use Actions, Save (or Apply) Recommendations to create a.sql script that you can review and modify, if necessary)
  - Might want to choose different index names
  - Might want to set FILLFACTOR values
- Execute the script immediately (in SSMS) or schedule the script to run at off hours (as a SQL Agent Job)

Are you done?

- DTA won’t solve all problems
- DTA only helps you to design and implement more effective structures and indexes – it doesn’t create maintenance scripts/plans
- Once indexes are in place, data and distribution statistics can change – making indexes fragmented and/or their statistics out of date

- Need to have solid maintenance practices in place to ensure optimal performance over time

Designing for Performance

Not the primary focus for today – but lots of additional resources to review!

The Most Common Problem

The system was not designed for what actually happens – either in terms of availability or scalability…

- Causes of *problem* are not properly understood
- Users end up doing things you didn't expect

For Scalability, Reliability, Availability,…
Three Primary Focus Areas

1) Know your data
   - Designing for Performance (MSDN Webcast Series)
2) Know your users
   - Indexing for Performance (sessions 4/5 + **this session!**)
   - Optimizing Procedural Code (session 7 + **this session!**)
   - Changing Isolation where appropriate (session 6)
3) Users lie!
   - Profiling the user requests (session 9 + **this session!**)
   - Profiling the server responses

The rest of the day has only one primary theme
Solving the Problem(s)!

Designing for Performance
Know your Data

- Database Creation
  - Placement of Data/Log (session 1)
  - Minimizing internal and external file fragmentation (session 1)
  - Setting/Changing Recovery Models (session 2)
- Table Creation
  - Choosing the Data Type for the job (session 3)
  - Row Size – Vertical Partitioning (session 4)
  - Number of rows – Horizontal Partitioning (session 8)

Indexing for Performance
Know your Users

- Transaction Processing
  - LESS indexes (session 4) but requires MORE maintenance (session 5)
  - Stored procedures probably play a large role (session 7) Recovery and Availability are Critical (sessions 1-2)
- Read-only/Decision Support
  - MORE indexes (session 4) but requires LESS maintenance (session 5)
  - Ad Hoc queries probably play a larger role (session 9) in Read-only/Decision Support
  - Must find the right balance of indexes (session 4)
  - Must design to keep objects available due to maintenance (session 5)
  - Should consider looking into Snapshot Isolation (session 6)

Optimizing Procedural Code
Know your Users

- Recompiling enough?
  - Plans are saved – that's not *always* a good thing (session 7)
  - Stored procedures with complex/condition logic may not have an optimal plan saved (session 7)
- Recompiling too much?
  - Inconsistencies in environment (session settings) http://msdn.microsoft.com/library/en-us/dnsql2k/html/sql_queryrecompilation.asp?frame=true
  - Not using *new* statement-level recompilation options (session 7)

Data Access Patterns and Analysis
Know your Users

- Decision Support Choices
  - Complex Business Intelligence
    ⇨ Create a separate data warehouse, build with SSIS, use Analysis Services to Analyze/Mine
  - Read-only version of exact OLTP environment
    ⇨ Backup/Restore
    ⇨ Database Snapshots
- Mixed Workload Choices
  - Vertical Partitioning (session 3)
  - Trading Consistency for Concurrency (session 6)
  - Snapshot Isolation (readers won't block writers, writers won't block readers) (session 6)

Profiling
Users lie!

- Targeting the Known problems
- Targeting resource problems
  - Tip: Profile for high reads, writes, duration
- Targeting blocking problems
  - Tip: Profile for LOW reads/writes with HIGH duration
- Correlate this with server resources and/or middle-tier/web resources

MSDN Webcast Series
www.microsoft.com/events/series/msdnsqlserver2005.mspx

- Session 1: Interaction Between Data and Log
- Session 2: Recovery Models
- Session 3: Table Optimization Strategies
- Session 4: Optimization Through Indexes
- Today! Advanced Indexing Strategies
- Session 5: Optimization Through Maintenance
- Session 6: Isolation, Locking, and Blocking
- Session 7: Optimizing Procedural Code
- Session 8: Table and Index Partitioning
- Session 9: Profiling/Server-side Trace Queues
- Session 10: Common Roadblocks, A Series Wrapup

Indexing for Performance

Solving Performance Problems with Advanced Indexing Strategies

- Appropriate Indexing Strategies
  - Indexing for AND
  - Indexing for OR
  - Indexing for Joins
  - Indexing for Aggregations
  - Indexed Views
- Other Indicators for Index Changes
- Design Considerations

Prerequisite Key Points

Be sure to go back to the series for the details

- ***Most*** tables perform better with a clustered index, especially one that meets these criteria:
  - Unique, Narrow, Static = most important
  - Ever-increasing (ideal when monotonically increasing)
  - Examples of good clustering keys:
    - Identity
    - Date, Identity
    - GUID (only when using **`newsequentialid()`** function)
- Heaps are great for staging data (when there are NO indexes, not just the lack of a clustered index)
  - Excellent for high performance data loading (parallel bulk load and parallel index creation after load)
  - Excellent as a partition to a partitioned view or a partitioned table

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

demo

Indexing for AND

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

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.

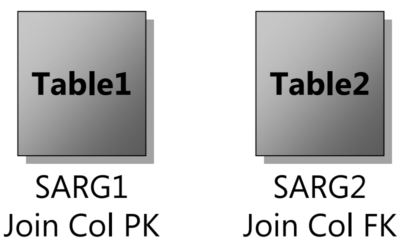
demo

Indexing for OR

Indexing for Joins

- Tables are joined two tables at a time
- Dissect your query into the individual table requests – key components are the SARGs and the join conditions
- 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 from which to choose…

Best Options for Joins – Phase I

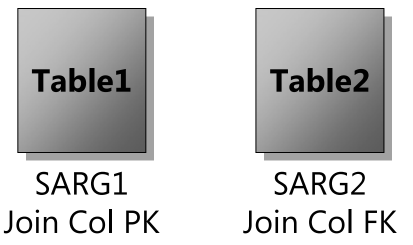


Do you already have individual indexes on each and all of these columns?

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 4 Indexes exist then the optimizer has the best choices

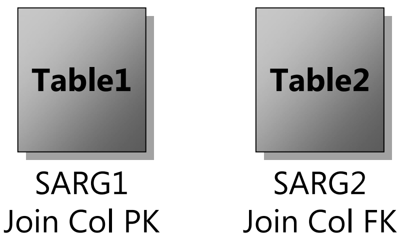
Cover the Combination – Phase II



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 Tables' Queries – 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

SQL skills.com

demo

Indexing for Joins

Indexing for Aggregations

- Two types of Aggregates:
  Stream and Hash
- Try to Achieve Stream to Minimize Overhead in temp table creation
  - To achieve stream create an index whose key is the GROUP BY clause and whose INCLUDE list is the rest of the columns
  - Using INCLUDE for the aggregates reduces the cost of keeping this index up to date when modified
- Computation of the Aggregate Still Required
- Lots of Users, Contention and/or Minimal Cache can Aggravate the problem

↳ **Indexed Views**

demo

Indexing for Aggregates

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
```

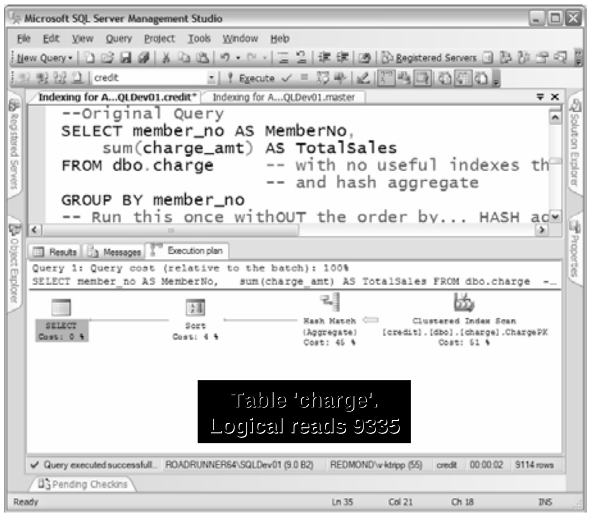
Aggregate Query (cont'd)
Table scan + hash aggregate

```
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 – 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


Microsoft SQL Server Management Studio
File Edit View Query Project Tools Window Help
New Query Registered Servers
credit Execute
Indexing for A...QLDev01.credit* Indexing for A...QLDev01.master
Results Messages Execution plan
SELECT Cost: 0 %
Sort Cost: 4 %
Hash Match (Aggregate) Cost: 46 %
Clustered Index Scan [credit].[dbo].[charge].ChargePK Cost: 51 %
Pending Checkins
Ready Ln 35 Col 21 Ch 18 INS

```
--Original Query
SELECT member_no AS MemberNo,
     sum(charge_amt) AS TotalSales
FROM dbo.charge      -- with no useful indexes th
                     -- and hash aggregate
GROUP BY member_no
-- Run this once withOUT the order by... HASH ac
```

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

Table 'charge'.
Logical reads 9335

Query executed successfull... ROADRUNNER64\SQLDev01 (9.0 B2) REDMOND\v-kltripp (55) credit 00:00:02 9114 rows

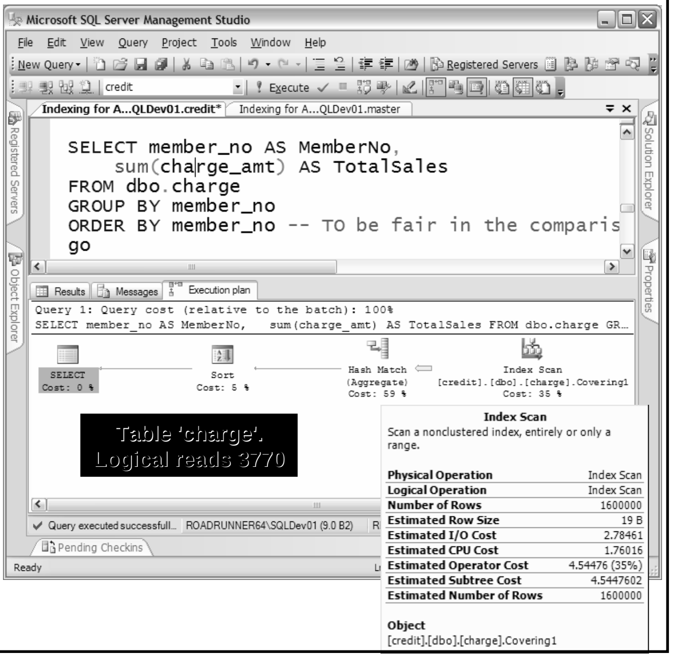
Aggregate Query
Index scan + hash aggregate

```
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 – 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
File Edit View Query Project Tools Window Help
New Query
Registered Servers
credit
Execute
Indexing for A...QLDev01.credit*
Indexing for A...QLDev01.master
Registered Servers
Object Explorer
Solution Explorer
Properties
Results
Messages
Execution plan
SELECT
Cost: 0 %
Sort
Cost: 5 %
Hash Match
(Aggregate)
Cost: 59 %
Index Scan
[credit].[dbo].[charge].Covering1
Cost: 35 %
Query executed successfull...
ROADRUNNER64\SQLDev01 (9.0 B2)
Pending Checkins
Ready

```
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
```

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

Table 'charge'.
Logical reads 3770

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

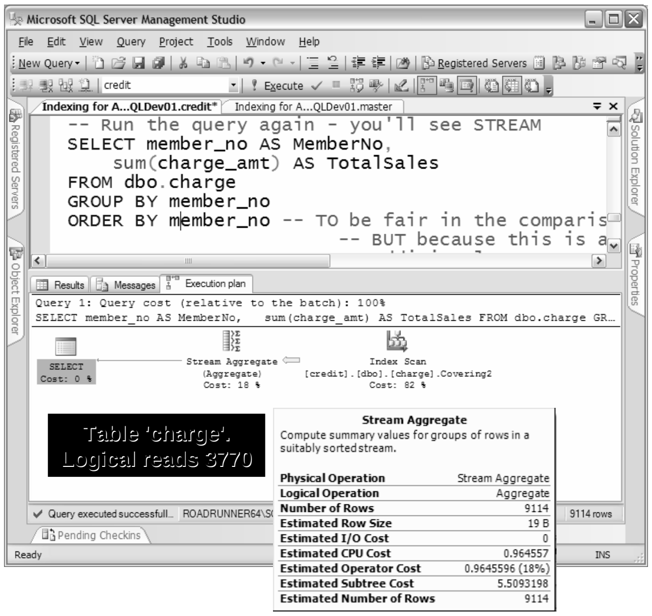
Aggregate Query
Index scan + stream aggregate

```
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



```
-- 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
```

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

SELECT
Cost: 0 %

Stream Aggregate
(Aggregate)
Cost: 18 %

Index Scan
[credit].[dbo].[charge].Covering2
Cost: 82 %

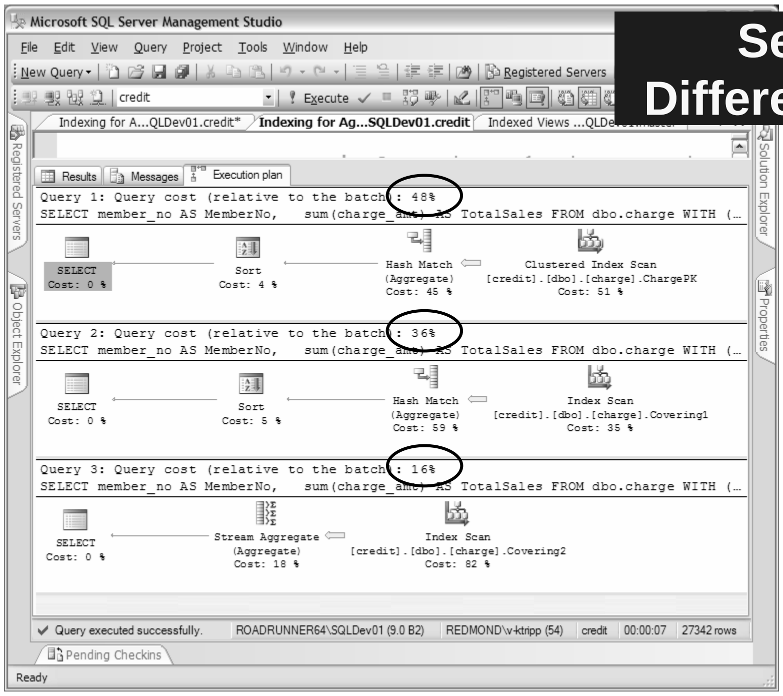
Table 'charge'.
Logical reads 3770

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

Query executed successfull... ROADRUNNER64\S 9114 rows

See the Difference?


Microsoft SQL Server Management Studio
File Edit View Query Project Tools Window Help
New Query
Registered Servers
credit
Execute
Indexing for A...QLDev01.credit*
Indexing for Ag...SQLDev01.credit
Indexed Views ...QLDev01.master
Results
Messages
Execution plan
Registered Servers
Object Explorer
Solution Explorer
Properties
Pending Checkins
Ready

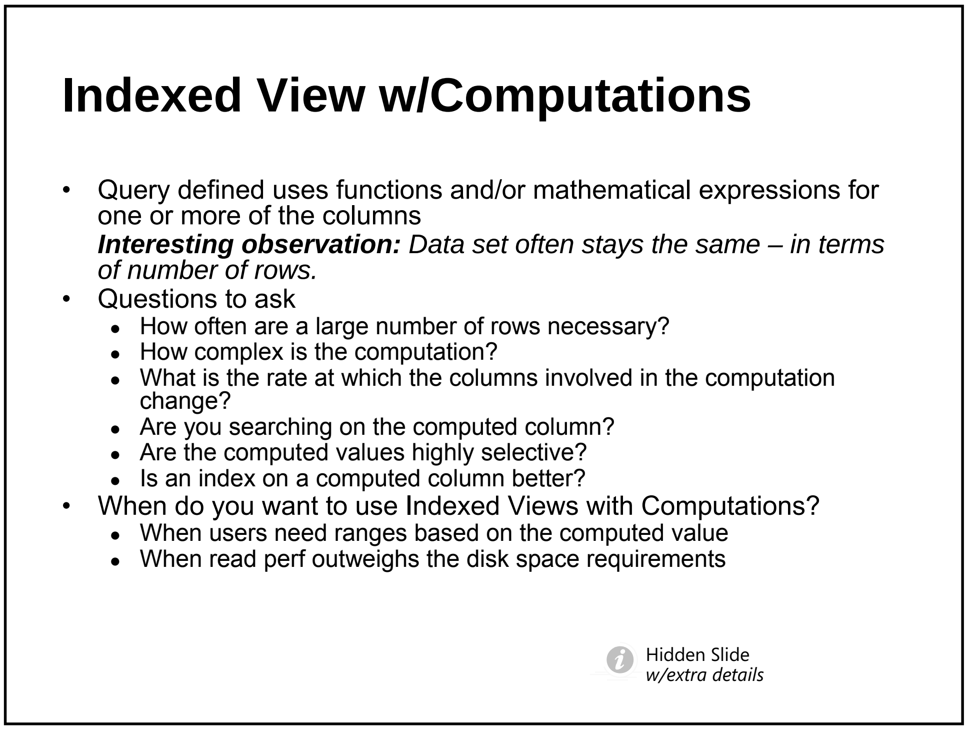
```
Query 1: Query cost (relative to the batch): 48%
SELECT member_no AS MemberNo,   sum(charge_amt) AS TotalSales FROM dbo.charge WITH (…

SELECT        Sort        Hash Match     Clustered Index Scan
Cost: 0 %     Cost: 4 %   (Aggregate)    [credit].[dbo].[charge].ChargePK
                          Cost: 45 %     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 (…

SELECT        Sort        Hash Match     Index Scan
Cost: 0 %     Cost: 5 %   (Aggregate)    [credit].[dbo].[charge].Covering1
                          Cost: 59 %     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 (…

SELECT        Stream Aggregate   Index Scan
Cost: 0 %     (Aggregate)        [credit].[dbo].[charge].Covering2
              Cost: 18 %         Cost: 82 %

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

Indexed View with Joins

- Query defined accesses multiple tables
  Interesting observation: *Result set often contains a significant amount of duplicated data.*
- Questions to ask
  - What is the amount of data duplication?
  - What is the rate at which that data is modified?
  - Is it really better than having the right indexes to support the join?
- When do you want to use Indexed Views with Joins?
  - When the overly duplicated data is relatively static (and the data set is relatively small)
  - When the volatile data is not overly duplicated
  - When read performance outweighs the disk space requirements
  - You've made sure that indexes don't help
  - The data set is relatively small (could end up wasting A LOT of cache with duplicated information)

Hidden Slide
w/extra details

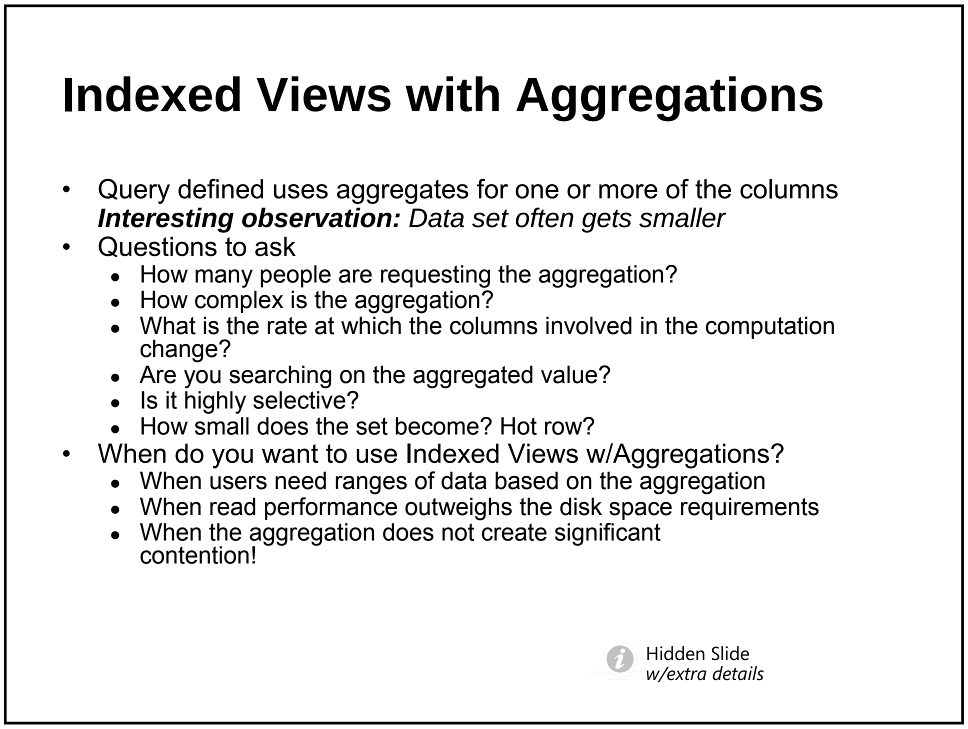
SQL skills.com



Indexed View w/Computations

- Query defined uses functions and/or mathematical expressions for one or more of the columns
  Interesting observation: *Data set often stays the same – in terms of number of rows.*
- Questions to ask
  - How often are a large number of rows necessary?
  - How complex is the computation?
  - What is the rate at which the columns involved in the computation change?
  - Are you searching on the computed column?
  - Are the computed values highly selective?
  - Is an index on a computed column better?
- When do you want to use Indexed Views with Computations?
  - When users need ranges based on the computed value
  - When read perf outweighs the disk space requirements

Hidden Slide
w/extra details

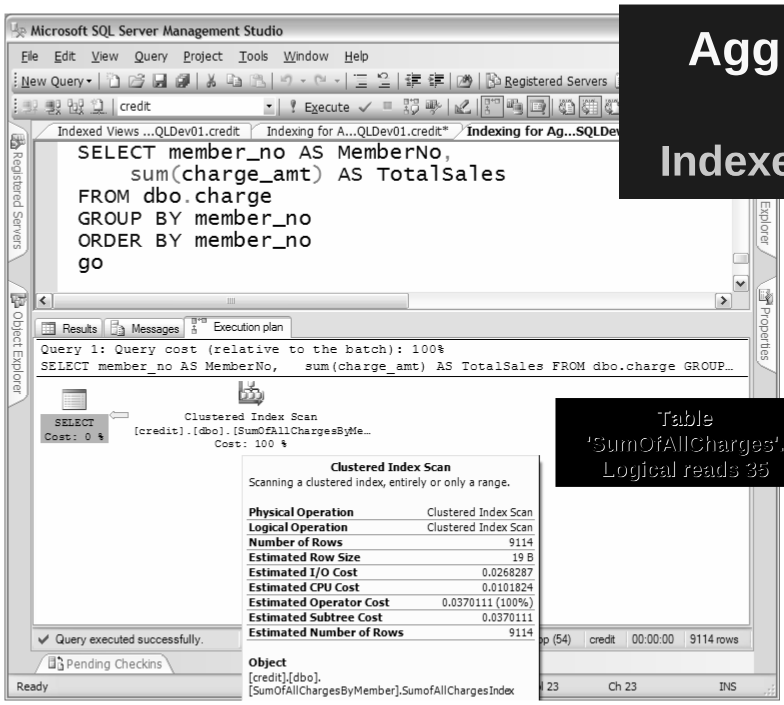


Indexed Views with Aggregations

- 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 at which the 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 w/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!

Hidden Slide
w/extra details

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…
```

SELECT Cost: 0 % ← Clustered Index Scan [credit].[dbo].[SumOfAllChargesByMe… Cost: 100 %

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

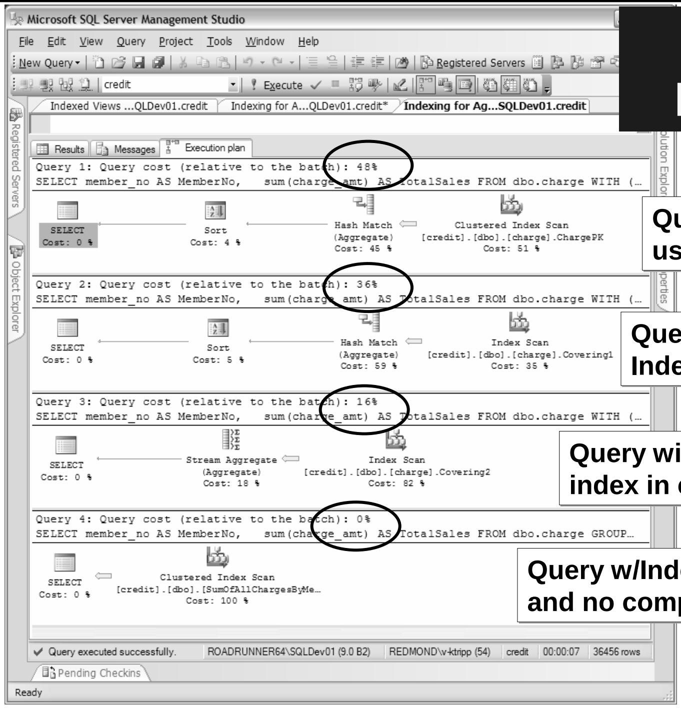
Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Number of Rows	9114
Estimated Row Size	19 B
Estimated I/O Cost	0.0268287
Estimated CPU Cost	0.0101824
Estimated Operator Cost	0.0370111 (100%)
Estimated Subtree Cost	0.0370111
Estimated Number of Rows	9114

Object
[credit].[dbo].[SumOfAllChargesByMember].SumofAllChargesIndex

Query executed successfully. | op (54) | credit | 00:00:00 | 9114 rows

Table 'SumOfAllCharges'. Logical reads 35

See the Difference?


Microsoft SQL Server Management Studio
File Edit View Query Project Tools Window Help
New Query
Registered Servers
credit
Execute
Indexed Views ...QLDev01.credit
Indexing for A...QLDev01.credit*
Indexing for Ag...SQLDev01.credit
Registered Servers
Object Explorer
Results
Messages
Execution plan
Pending Checkins
Ready

```
Query 1: Query cost (relative to the batch): 48%
SELECT member_no AS MemberNo,   sum(charge_amt) AS TotalSales FROM dbo.charge WITH (…
```

SELECT Cost: 0 % ← Sort Cost: 4 % ← Hash Match (Aggregate) Cost: 45 % ← Clustered Index Scan [credit].[dbo].[charge].ChargePK Cost: 51 %

Query with NO useful indexes

```
Query 2: Query cost (relative to the batch): 36%
SELECT member_no AS MemberNo,   sum(charge_amt) AS TotalSales FROM dbo.charge WITH (…
```

SELECT Cost: 0 % ← Sort Cost: 5 % ← Hash Match (Aggregate) Cost: 59 % ← Index Scan [credit].[dbo].[charge].Covering1 Cost: 35 %

Query with covering Index in wrong order

```
Query 3: Query cost (relative to the batch): 16%
SELECT member_no AS MemberNo,   sum(charge_amt) AS TotalSales FROM dbo.charge WITH (…
```

SELECT Cost: 0 % ← Stream Aggregate (Aggregate) Cost: 18 % ← Index Scan [credit].[dbo].[charge].Covering2 Cost: 82 %

Query with covering index in correct order

```
Query 4: Query cost (relative to the batch): 0%
SELECT member_no AS MemberNo,   sum(charge_amt) AS TotalSales FROM dbo.charge GROUP…
```

SELECT Cost: 0 % ← Clustered Index Scan [credit].[dbo].[SumOfAllChargesByMe… Cost: 100 %

Query w/Indexed View and no computations!

Query executed successfully. | ROADRUNNER64\SQLDev01 (9.0 B2) | REDMOND\v-ktripp (54) | credit | 00:00:07 | 36456 rows

SQL skills.com

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!

Indexed View with Aggregates

- 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

Can you cover EVERYTHING?

- Yes, you CAN cover virtually anything (for a table)
  - Any query's requested data can be covered against that table
  - Nonclustered indexes can include LOB types (SQL Server 2005 only)
- Just because you can, should you?
  - Use INCLUDE (and covering indexes) ONLY for queries that excessively use resources and are high priority
  - Use Indexed Views sparingly in OLTP and be sure to test for the hot row problem caused by aggregates
- Are there any other concerns?

Indexing-related Concerns & Considerations

SQL skills.com

Other Indicators for Index Changes

- Showplan Indicators of poor performance – where indexes may help
  - “Scan” – not always a bad thing but typically more expensive than a seek
    - Scan is OK if result set is large
    - Scan is problematic if result set is small (especially when less than 1%)
    - Scan on an index isn’t as bad but if it’s a high priority query and there are any limiting conditions, consider an index that’s seekable
  - “Spool” operators – indicates a temporary/worktable
  - “Hash” – indicates a temporary/worktable was created and that the best option may not have been available

These are not always bad but start by asking the DTA

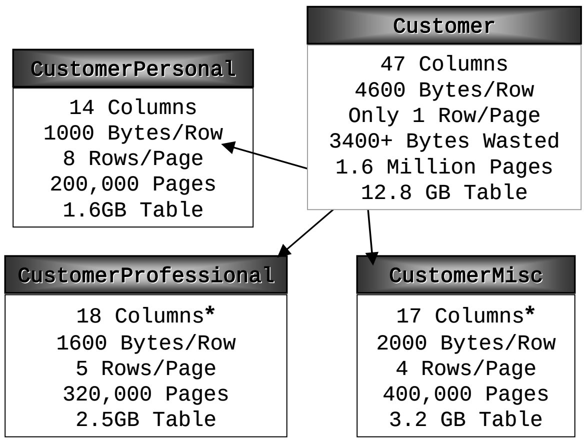
Design Considerations

Index-related features may warrant design changes

- LOB columns in the leaf level of an index do NOT allow that index to be built/rebuilt online (reason: LOB compaction is automatic now)
  - Design strategy to circumvent => Vertical Partitioning
- A partition of a partitioned table cannot be rebuilt online (reason: row versioning – which is what an online index operation uses behind the scenes – is at the table level, so while the ENTIRE partitioned table’s indexes CAN be rebuilt online a partition can only be rebuilt offline….)
  - Design strategy to circumvent => Combine horizontal partitioning strategies

Benefits of Vertical Partitioning

Customer Table with 1,600,000 Rows



*****The Primary key column must be made redundant for these two additional tables. 47 Columns in Customer. 49 Columns total between 3 tables.

Customer
= 12.8 GB

Partitioned Tables
= 7.3 GB

✓ Savings in Overall Disk Space (5.5 GB Saved)
✓ Not reading data into cache when not necessary
✓ Locks are table specific therefore less contention at the row level
✓ **LOB data can be isolated to support online index operations for more critical data**

Partitioning Scenario

- If RW portion is only current month then that's the only place where fragmentation will occur
- If current month is June then only rebuild June (partition = 6):

```
ALTER INDEX ChargesPTPK ON ChargesPT
REBUILD PARTITION = 6
WITH (ONLINE = ON)
```

'ONLINE' is not a recognized ALTER INDEX REBUILD PARTITION option.

Partitioning for Performance & Online Index Rebuilds

- Separate your read-only into a partitioned table
- Keep read-write as standalone table or additional partitioned table
- For simplified user access, consider using a partitioned view over the RO partitioned table and RW standalone table to simplify user access
- OR use application directed inserts, updates, deletes and selects (better for availability)
- Other benefits too:
  - Partitioned Tables are easier for the optimizer to optimize
  - Using separate tables allows you to index the RO data one way (more indexes with no maintenance) and the RW another way (fewer indexes with maintenance)

Finding the Right Balance

Index Strategies – Summary

- Create your clustered index first, choose wisely (session 4)
- Create your constraints
  - Primary Key – automatically gets a unique CL index
  - Unique Key – automatically gets a unique NC index
- Create (manually) NC Indexes on the columns that have foreign key constraints
- Capture a Workload(s), use Database Tuning Advisor, work through the suggestions (iteratively) to determine additional indexes needed
- Add additional indexes to help improve SARGs, Joins, Aggregations and use DTA as an advisor in “sandbox” tuning
- Are you done? **NO!**

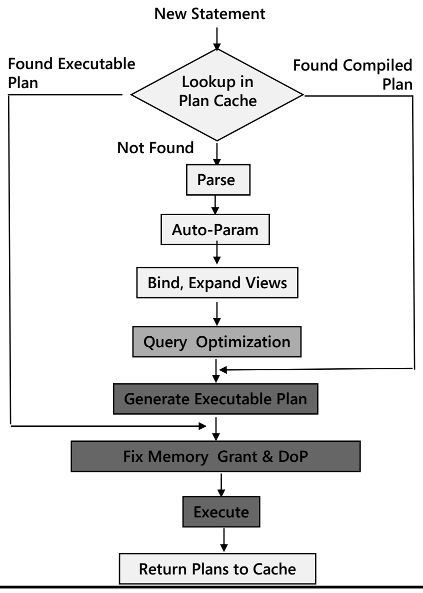
Watch MSDN Webcast session 5 for details on fragmentation and index maintenance best practices

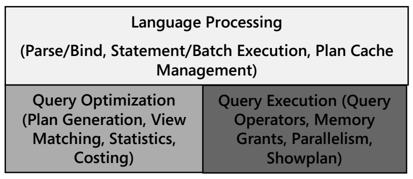
SQL skills.com

Plans, Plan Caching *and* Optimizing Procedural Code

FYI – I suspect that we'll only be able to do ***Demo Madness*** here… not sure if we'll have time for all of the slides but I wanted to add them in for completeness!
Check out MSDN Webcast, Part 7 for more details on this topic!

Statement Execution




Language Processing
(Parse/Bind, Statement/Batch Execution, Plan Cache Management)
Query Optimization (Plan Generation, View Matching, Statistics, Costing)
Query Execution (Query Operators, Memory Grants, Parallelism, Showplan)

Statement Auto-parameterization

- Evaluates your query to determine if it's “safe”
  - Needs to be a fairly straightforward plan
  - Parameters do not change plan choice
    - Equality for a lookup of PK value is probably safe
    - Searching with an IN is not safe
- Automatically parameterizes search arguments
- Places statement in cache for subsequent executions

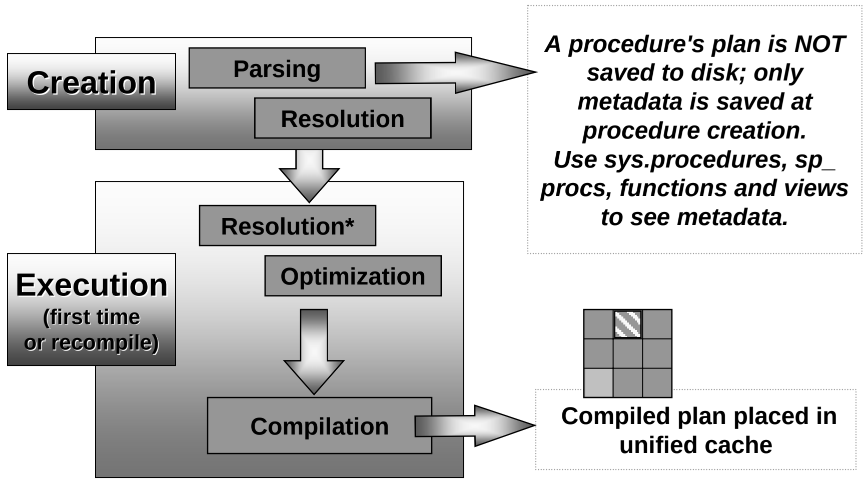
How do you know?

- SQL Server 2000/2005
  - Access master.dbo.syscacheobjects
- SQL Server 2005 – DMVs
  - sys.dm_exec_cached_plans
    - For list of sql statements in cache
  - sys.dm_exec_sql_text(sql_handle)
    - For the text of the sql statement executed
  - sys.dm_exec_query_plan(plan_handle)
    - For the execution plan of the sql statement executed
  - sys.dm_exec_query_stats
    - For a variety of query statistics – like number of executions, plan creation time (first execution into cache), last execution time, etc.

But is this *really* the best way?

- Plan caching is not all that optimal for reuse when:
  - Different parameters cause different plans
  - Ad-hoc needs to be textual match
- For better/controlled plan re-use consider writing Optimized Procedural Code:
  - Forced statement caching through `sp_executesql`
  - Stored procedures

Processing Stored Procedures

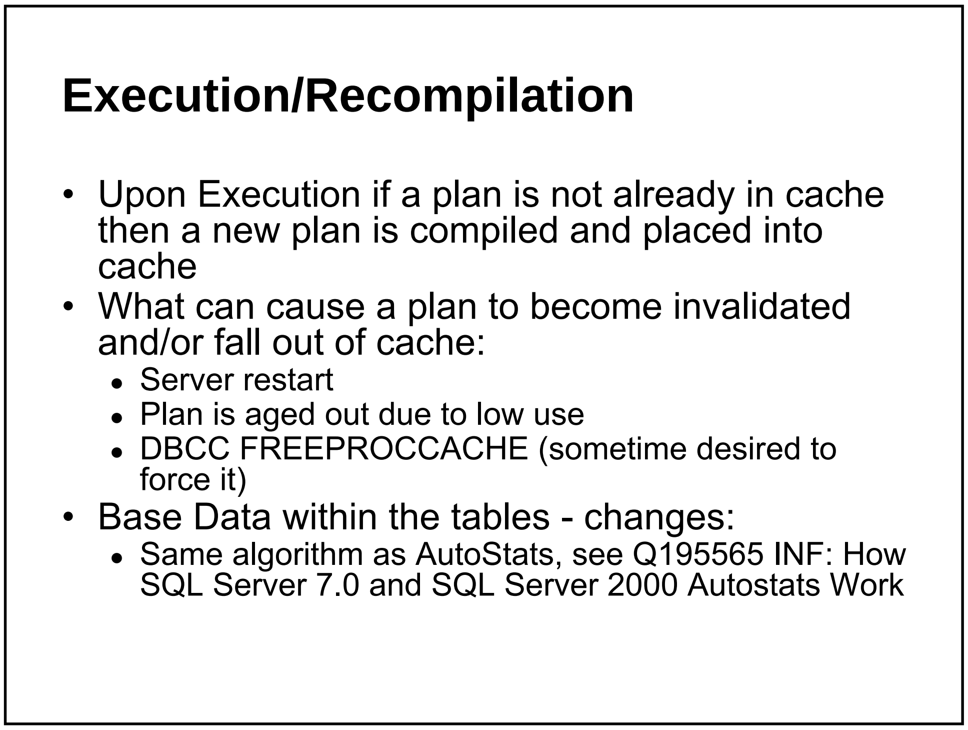

Creation
Parsing
Resolution
A procedure's plan is NOT saved to disk; only metadata is saved at procedure creation. Use sys.procedures, sp_ procs, functions and views to see metadata.
Execution (first time or recompile)
Resolution*
Optimization
Compilation
Compiled plan placed in unified cache

Resolution

- When a stored procedure is created all objects referenced are resolved (checked to see whether or not they exist).
- The create succeeds if the objects dne
  - Procedures called that do not exist generate error
  - Cannot add rows to sysdepends… The sp will still be created.
  - Benefit: Recursion is allowed!
  - Tables, Views, Functions called that do not exist - do NOT generate error (unless in 6.5 compatibility mode)
- Verify dependencies with sp_depends before dropping an object

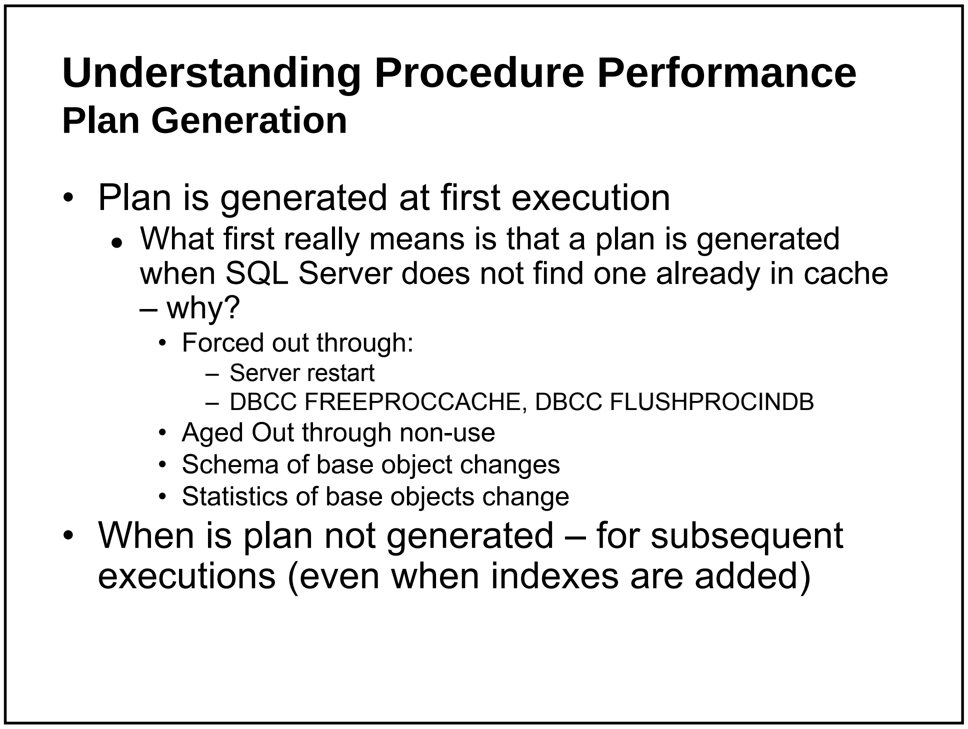
Compilation/Optimization

- Based on parameters supplied
- Future executions will reuse the plan
- Complete optimization of all code passed (more on this coming up…modular code!)
- Poor coding practices can cause excessive locking/blocking
- Excessive recompilations can cause poor performance



Execution/Recompilation

- Upon Execution if a plan is not already in cache then a new plan is compiled and placed into cache
- What can cause a plan to become invalidated and/or fall out of cache:
  - Server restart
  - Plan is aged out due to low use
  - DBCC FREEPROCCACHE (sometime desired to force it)
- Base Data within the tables - changes:
  - Same algorithm as AutoStats, see Q195565 INF: How SQL Server 7.0 and SQL Server 2000 Autostats Work



Understanding Procedure Performance
Plan Generation

- Plan is generated at first execution
  - What first really means is that a plan is generated when SQL Server does not find one already in cache – why?
    - Forced out through:
      - Server restart
      - DBCC FREEPROCCACHE, DBCC FLUSHPROCINDB
    - Aged Out through non-use
    - Schema of base object changes
    - Statistics of base objects change
- When is plan not generated – for subsequent executions (even when indexes are added)

Recompilation Issues

RECOMPILATION = OPTIMIZATION
OPTIMIZATION = RECOMPILATION

- When do you want to recompile?
- What options do you have Recompilation?
- How do you know you need to recompile?
- Do you want to recompile the entire procedure or only part of it?
- Can you test it?

When to recompile?

- When the plan for a given statement within a procedure is not consistent in execution plan–due to parameter changes
- Cost of recompilation might be significantly less than the execution cost of a bad plan!
- Why?
  - Faster Execution with a better plan
  - Saving plans for reuse is NOT always beneficial
  - Some plans should NEVER be saved
- Do you want to do this for every procedure?
  - No, but start with the highest priority/expensive procedures first!

Options for Recompilation

- CREATE … WITH RECOMPILE
- EXECUTE … WITH RECOMPILE
- sp_recompile objname
- Statement Recompilation
  - The old way
    - Dynamic String Execution (sometimes the only option)
    - Modularized Code (still a good idea)
  - The new way (excellent for complex single statements)
    - OPTION(RECOMPILE)
    - OPTION(OPTIMIZE FOR
      (@variable_name = literal_constant, ...))

How do you know?

- You Test!
  - Test optimization plans consistency using EXECUTE WITH RECOMPILE
  - Choose what needs to be recompiled
    - Whole Procedure
    - Portions of the procedure
  - Test final performance using strategy
    - Procedure Recompilation
      (CREATE with RECOMPILE)
    - Statement Recompilation
      (Dynamic String Execution)
    - Modularized Code
      (Sub procedures created with or without WITH RECOMPILE)

EXECUTE WITH RECOMPILE

- Excellent for Testing
- Verify plans for a variety of test cases

```
EXEC dbo.GetMemberInfo 'Tripp' WITH RECOMPILE
EXEC dbo.GetMemberInfo 'T%' WITH RECOMPILE
EXEC dbo.GetMemberInfo '%T%' WITH RECOMPILE
```

- Do the execution plans match?
- Are they consistent?
- Yes ⇨ then create the procedure normally
- No ⇨ Determine what should be recompiled

What Should be Recompiled?

- Whole Procedure
  - CREATE with RECOMPILE
    - Procedure is recompiled for each execution
  - EXECUTE with RECOMPILE
    - Procedure is recompiled for that execution
      - NOTE: Consider forcing recompilation through another technique – you should not expect users will know when/why to use EXECUTE … WITH RECOMPILE
- Statement(s) Recompilation
  - If limited number of statements cause recompile
    - Dynamic String Execution
    - Modular Code
    - New (SQL Server 2005 only) statement-level recompilation options

CREATE … WITH RECOMPILE

- Use when the procedure returns drastically different results based on input parameters.
- May not be the only – or even the best option…
- How do you know?

```
CREATE PROCEDURE GetMemberInfo
( @LastName     varchar(30) )
  WITH RECOMPILE
AS
SELECT m.*
  FROM dbo.Member AS m
  WHERE m.LastName LIKE @LastName
go
EXEC dbo.GetMemberInfo 'Tripp' -- index+bookmark
EXEC dbo.GetMemberInfo 'T%'    -- optimally, a table scan
EXEC dbo.GetMemberInfo '%T%'   -- optimally, a table scan!
```

Statement-level Recompilation

- What if only a small number of statements need to be recompiled?
- The SQL Statement is not likely safe (i.e. it will not be saved/parameterized)
- Dynamic String Execution!
  - Amazingly Flexible
  - Permission Requirements
  - Potentially Dangerous
  - Advanced Examples
    - Complex large strings
    - Changing database context
    - Output parameters

Statement-level Recompilation

- The old way: Modularizing your code
  - Doesn't hurt!
  - May allow better reuse of code "snippets"
  - Allows "block recompilation" in conditional logic
- The new way: "inline recompilation"
  - OPTION(RECOMPILE)
    - Excellent when parameters cause the execution plan to widely vary
    - Bad because EVERY execution will recompile
  - OPTIMIZE FOR (@variable_name = constant, ...)
    - Excellent when large majority of executions generate the same optimization time
    - You don't care that the minority may run slower with a less than optimal plan?

Modular Code

An Excellent Solution!

```
IF (expression operator expression)
  SQL Statement Block1
ELSE
   SQL Statement Block2
```

Scenario 1 – upon first execution…
- Parameters are passed such that the ELSE condition executes – BOTH Block1 and Block2 are optimized with the input parameters

Scenario 2 – upon first execution…
- Parameters are passed such that the IF condition executes – ONLY Block1 is optimized. Block2 will be optimized when a parameter which forces the ELSE condition is passed.

Solution?
Do not use a lot of conditional SQL Statement Blocks
Call separate stored procedures instead!

See ModularProcedures.sql

Plan Caching Options

- Stored Procedures and sp_executesql have the same potential for problems
- Forcing a recompile can be warranted/justified
- Always recompile the smallest amount possible!
- But can you have too many recompiles?
  - Yes, but it's not quite as bad as 2000 because only the statement is recompiled, instead of the entire procedure
  - Yes, but following some best practices can help to minimize that!

Recompilations

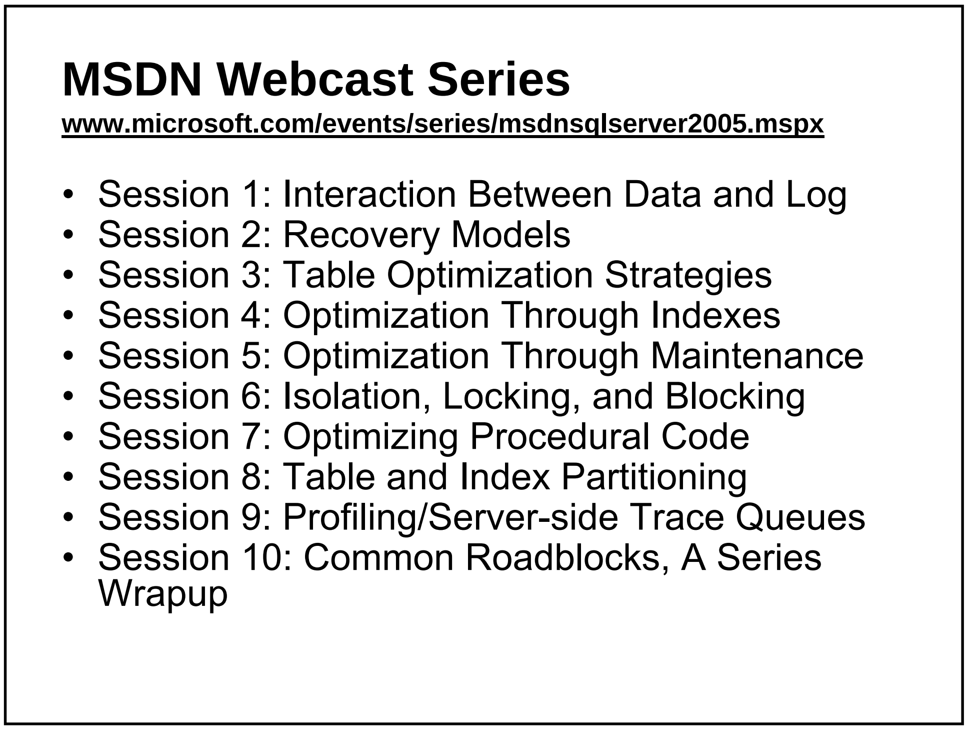
- Possibly too few
  - Widely varying result sets
  - Parameterization
  - Conditional logic
- Could there be too many
  - Understanding why – Profiling
  - Stored Procedure Best Practices
    - Setting Session Settings
    - Interleaving DML/DDL
    - Temp Tables within stored procedures
    - Temp Tables v. Table Variables

Profiling SP Performance

- Create New Trace (SQLProfilerTSQL_sps)
- Replace **SP:StmtStarting** w/**SP:StmtCompletion**
  - Better if you want to see a duration (starting events don't have a duration)
  - Add Duration as a Column Value
- If short term profiling for performance:
  - Add columns: Reads, Writes, Execution Plan
- Always use Filters
  - Database Name (only the db you want)
  - Exclude system IDs (checkbox on filter dialog)
- Resources:
  - Query Recompilation in SQL Server 2000
    http://msdn.microsoft.com/library/en-us/dnsql2k/html/sql_queryrecompilation.asp?frame=true
  - Troubleshooting stored procedure recompilation
    http://support.microsoft.com/default.aspx?scid=kb;en-us;Q243586

Session Summary

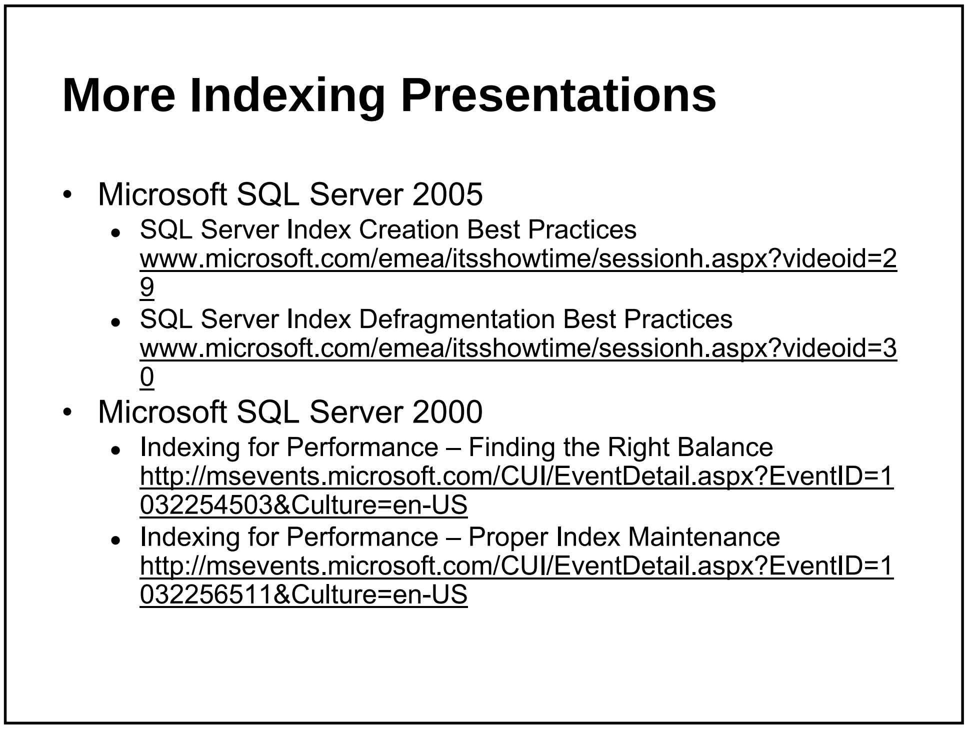
- Indexing is the closest thing to a magic bullet but a truly scalable system also requires design and optimizing procedural code
- You can *over* index, you can *under* index – you need to "find the right balance" and prioritize:
  ↳ Know your data, know your users, know your workload
- Good maintenance procedures are what's going to keep your system running smoothly
- Good design practices and transactional coding practices will give you better performance AND access to more powerful features and capabilities!
- Test, Test, Test!
- **Final words:** May all your code be compiled and optimized! ☺



MSDN Webcast Series

www.microsoft.com/events/series/msdnsqlserver2005.mspx

- Session 1: Interaction Between Data and Log
- Session 2: Recovery Models
- Session 3: Table Optimization Strategies
- Session 4: Optimization Through Indexes
- Session 5: Optimization Through Maintenance
- Session 6: Isolation, Locking, and Blocking
- Session 7: Optimizing Procedural Code
- Session 8: Table and Index Partitioning
- Session 9: Profiling/Server-side Trace Queues
- Session 10: Common Roadblocks, A Series Wrapup



More Indexing Presentations

- Microsoft SQL Server 2005
  - SQL Server Index Creation Best Practices www.microsoft.com/emea/itsshowtime/sessionh.aspx?videoid=29
  - SQL Server Index Defragmentation Best Practices www.microsoft.com/emea/itsshowtime/sessionh.aspx?videoid=30
- Microsoft SQL Server 2000
  - Indexing for Performance – Finding the Right Balance http://msevents.microsoft.com/CUI/EventDetail.aspx?EventID=1032254503&Culture=en-US
  - Indexing for Performance – Proper Index Maintenance http://msevents.microsoft.com/CUI/EventDetail.aspx?EventID=1032256511&Culture=en-US

TechNet Webcast Series

http://www.microsoft.com/events/series/technetsqlserver2005.mspx

- Session 1: A Fast-paced Feature Overview and Series Introduction
- Session 2: Security, presented by Bob Beauchemin, SQLskills.com
- Session 3: Understanding Installation Options and Initial Configuration
- Session 4: Upgrade Considerations and Migration Paths
- Session 5: Effective Use of the New Management Tools
- Session 6: New Application Design Patterns for Scalability and Availability and the Operational Impacts of Service Broker, presented by Bob Beauchemin, SQLskills.com
- Session 7: Technologies and Features to Improve Availability
- Session 8: Implementing Database Mirroring, Part 1 of 2, presented by Mark Wistrom, Database Mirroring Program Manager, Microsoft Corp.
- Session 9: Implementing Database Mirroring, Part 2 of 2
- Session 10: Recovering from Human Error
- Session 11: Best Practices and Series Wrap-up

Session Specific Resources

Demo Scripts, Resource Links, Additional Materials
http://www.SQLskills.com
http://www.SQLskills.com, Past Events

SQLskills Immersion Events
http://www.SQLskills.com, Events, Immersion Events

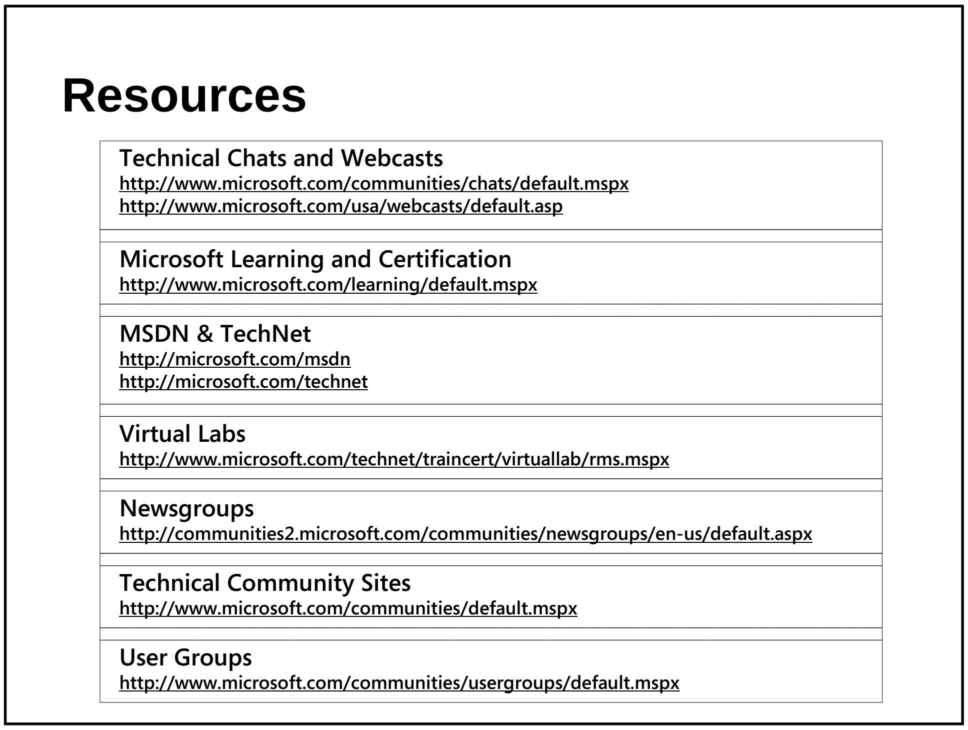
SQL Server Always On Technologies
http://www.microsoft.com/sql/AlwaysOn

SQL Server High Availability Technologies
http://www.microsoft.com/sql/technologies/highavailability/

SQL Server VLDB Case Studies and Other Information
http://www.microsoft.com/sql/bigdata

Microsoft SQL Server Developer Center on MSDN
http://msdn.microsoft.com/sql/

Microsoft SQL Server TechCenter on TechNet
http://www.microsoft.com/communities/usergroups/default.mspx



Resources

Technical Chats and Webcasts
http://www.microsoft.com/communities/chats/default.mspx
http://www.microsoft.com/usa/webcasts/default.asp

Microsoft Learning and Certification
http://www.microsoft.com/learning/default.mspx

MSDN & TechNet
http://microsoft.com/msdn
http://microsoft.com/technet

Virtual Labs
http://www.microsoft.com/technet/traincert/virtuallab/rms.mspx

Newsgroups
http://communities2.microsoft.com/communities/newsgroups/en-us/default.aspx

Technical Community Sites
http://www.microsoft.com/communities/default.mspx

User Groups
http://www.microsoft.com/communities/usergroups/default.mspx

Q&A

Thank you!

Please take a moment to fill out your evaluation.

Kimberly L. Tripp

Consultant . Trainer . Writer . Speaker

email: **Kimberly@SQLskills.com**

Make sure to register for special offers and other helpful information and resources!

www.SQLskills.com