Microsoft

CSS Academy

Performance Tuning - Database Design and Query Optimization

Ying Fan
Support Engineer
Customer Service & Support
Asia Pacific & Greater China Region

Microsoft

CSS Academy

Welcome!

Agenda

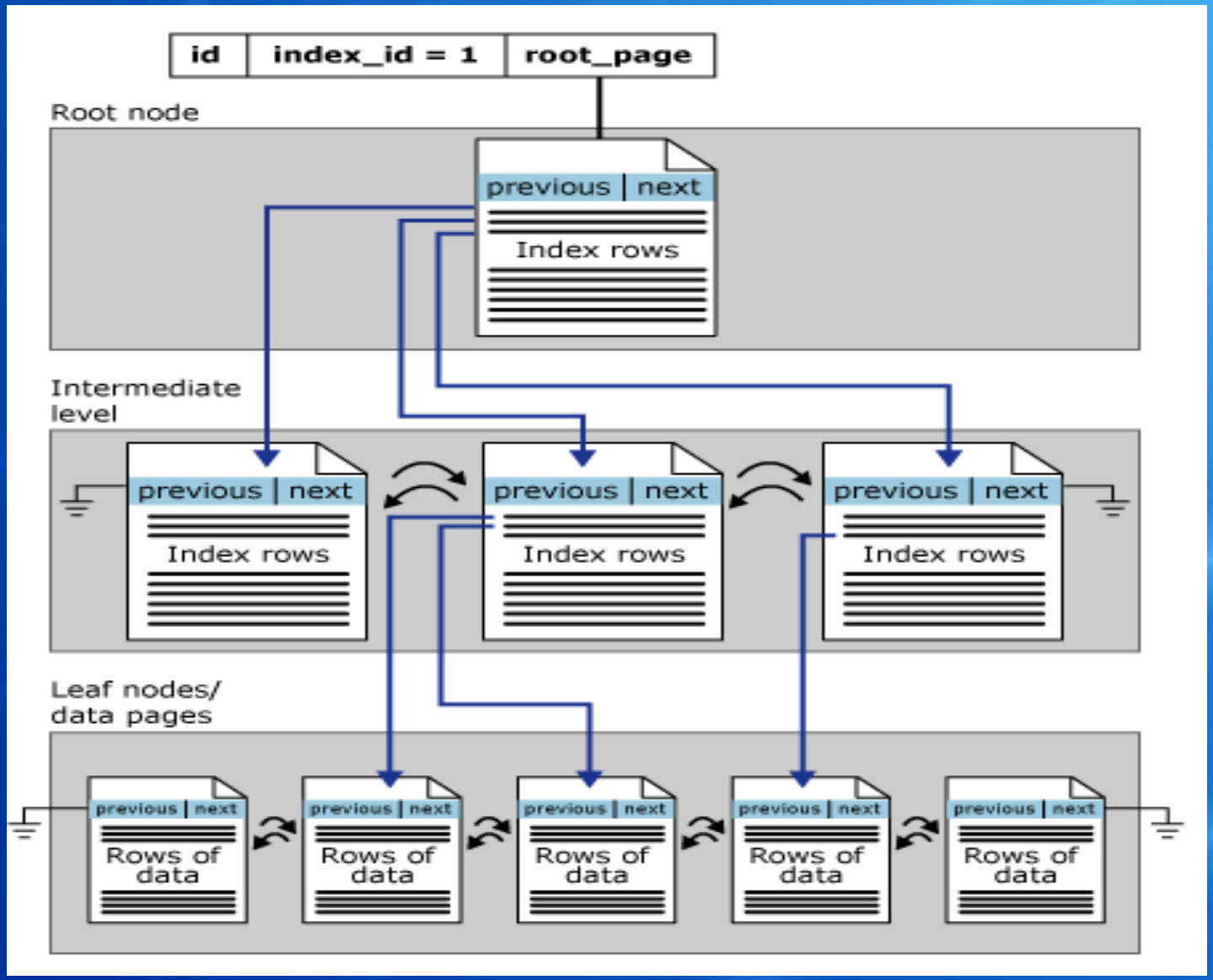
- Index structure
- Reading Query Plans
- Troubleshooting Query Plans
- Optimizing Query Plans
- Plan Caching

Index Structure

Index Types

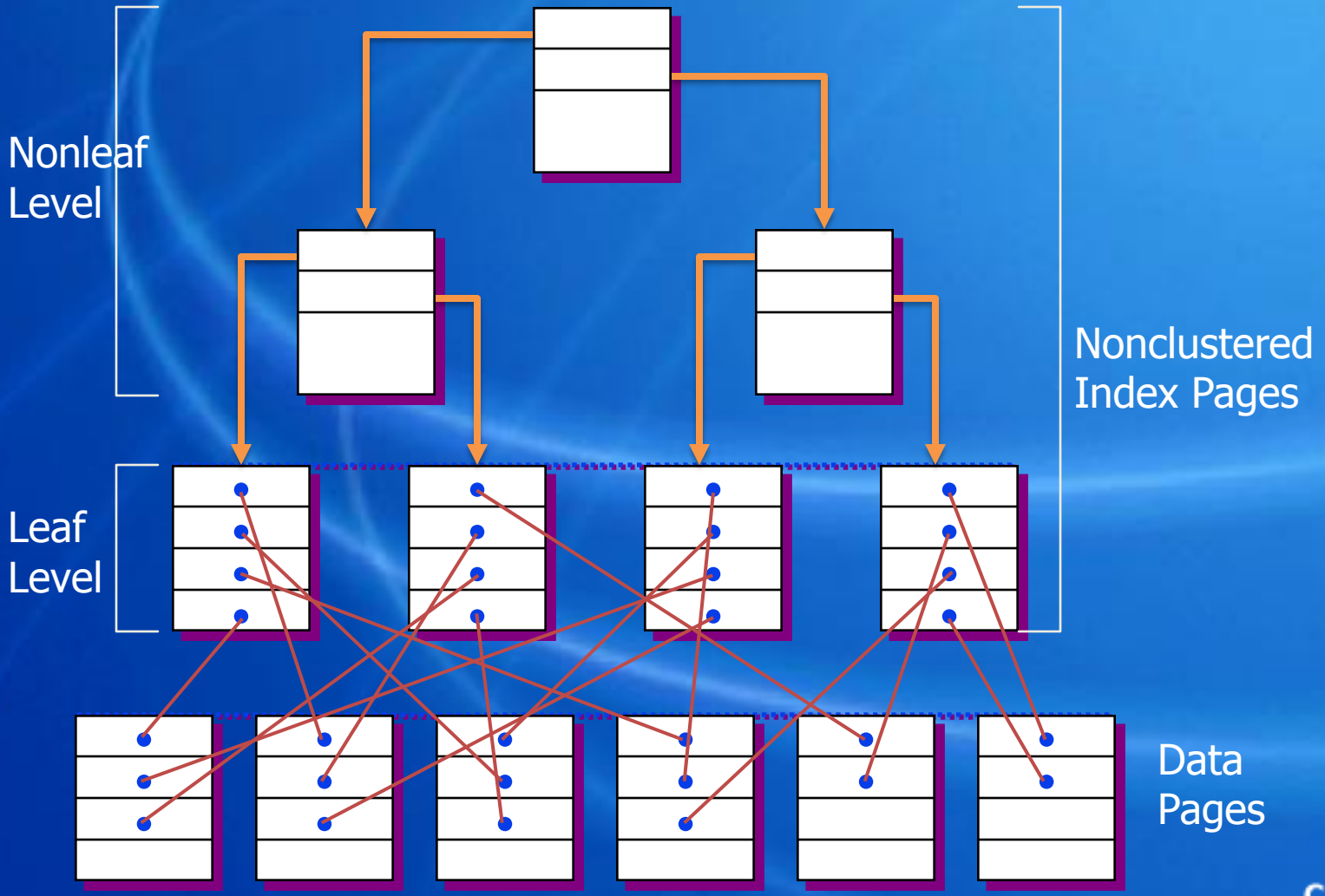
- SQL Server maintains indexes using a B-trees Structure
- Clustered Index
  - Data is physically ordered around the keys.
  - Actual data is stored at the leaf level.
- Nonclustered Index
  - An independent index structure that points to the underlying records
- A table without a clustered index is a “Heap”
  - Nonclustered indexes can exist on such a table

Clustered Index Characteristics



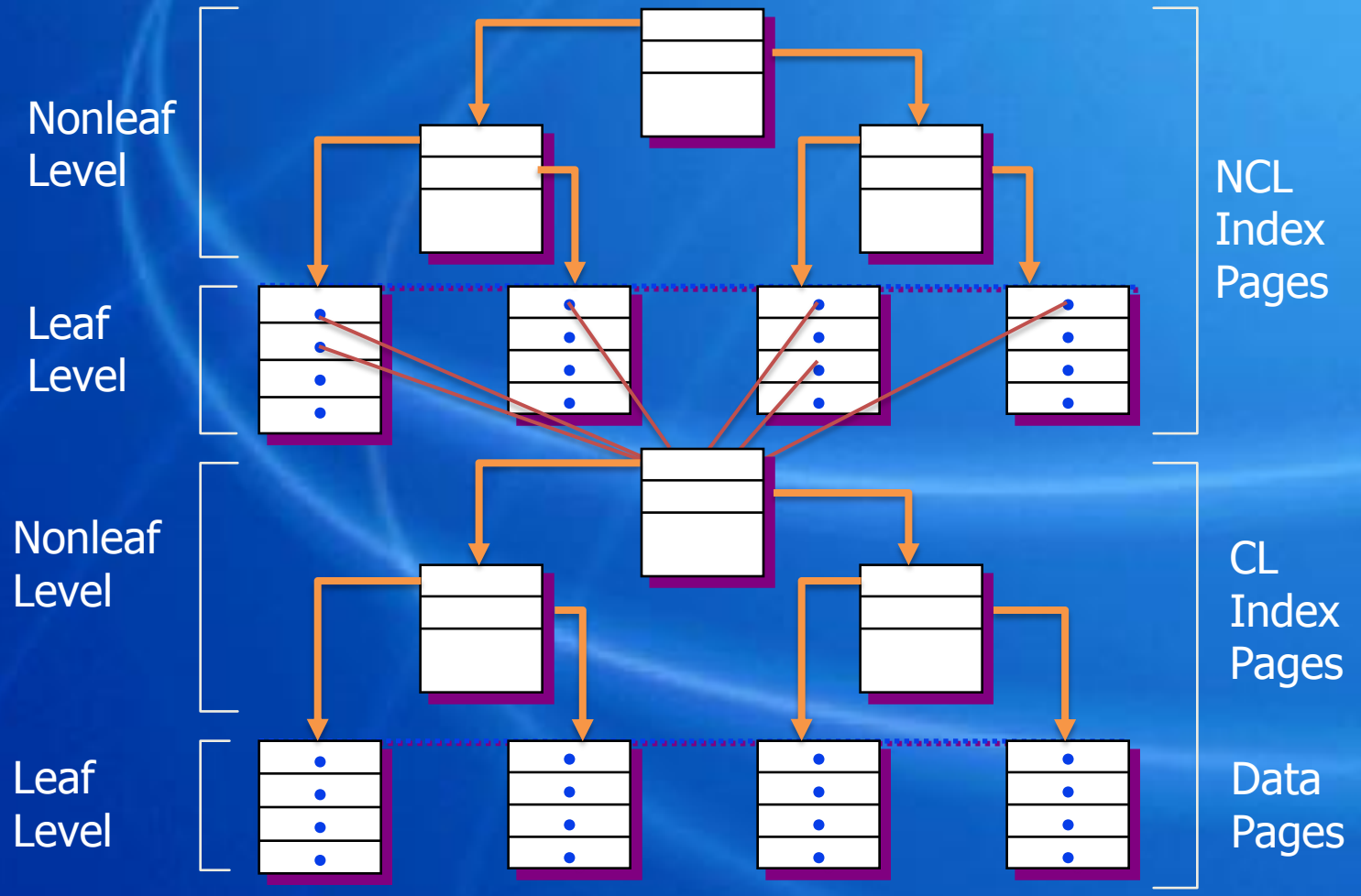
Nonclustered Indexes With a Heap

- All rows on leaf pages have pointers to rows



Nonclustered Index With a Clustered Index

- Entries in the nonclustered leaf pages contain clustered index key values



Covering Indexes

- A covering index contains all data needed by query, in its leaf level.
  - This may include the clustered index key(s).
  - A query satisfied by covering index is covered query
- So no bookmark lookup is needed, fastest data access possible.
- Meaningful only to non-clustered indexes.

Index Selection

- Clustered Index:
  - Smaller column that does not get modified and has a self ordered insertion.
  - To retrieve a range of data
- Non-Clustered Index:
  - Smaller column(s) with high selectivity.
  - For covering index, chose the most selective column as the first column in the list.
- Consider creating an index if column(s);
  - Frequently used for filtering
  - Is/Are part of foreign key constraints or join predicates
  - May cover some queries
- Use Index Tuning Wizard or Database Engine Tuning advisor for further accuracy.

Fragmentation Analyses

- DBCC SHOWCONTIG

Table: 'MyTable1' (1556968673); index ID: 1, database ID: 16
TABLE level scan performed.
- Pages Scanned.....: 18986
- Extents Scanned.....: 2443
- Extent Switches.....: 9238
- Avg. Pages per Extent.....: 7.8
- Scan Density [Best Count:Actual Count]....: 25.70%[2374:9239]
- Logical Scan Fragmentation .....: 44.58%
- Extent Scan Fragmentation .....: 87.07%
- Avg. Bytes Free per Page.....: 1658.7
- Avg. Page Density (full).....: 79.51%

Index Reorganization

- DBCC SHOWCONTIG Shows 3 Kinds of External Fragmentation
  - Extent scan fragmentation
  - Logical scan fragmentation
  - Scan Density
- DBCC INDEXDEFRAG Fixes Logical Scan Fragmentation
- Rebuilding Index Removes ALL Fragmentation (DBCC DBREINDEX)

DBCC IndexDefrag

- Low impact
  - ~20% degradation of medium OLTP workload
  - Only takes page level locks unlike DBREINDEX which takes a table lock.
- Uses minimal data space
- May be stopped and restarted
- Use analysis to decide when to reorg
- May generate a lot of log activity

Reading query plan

Why Read Query Plans?

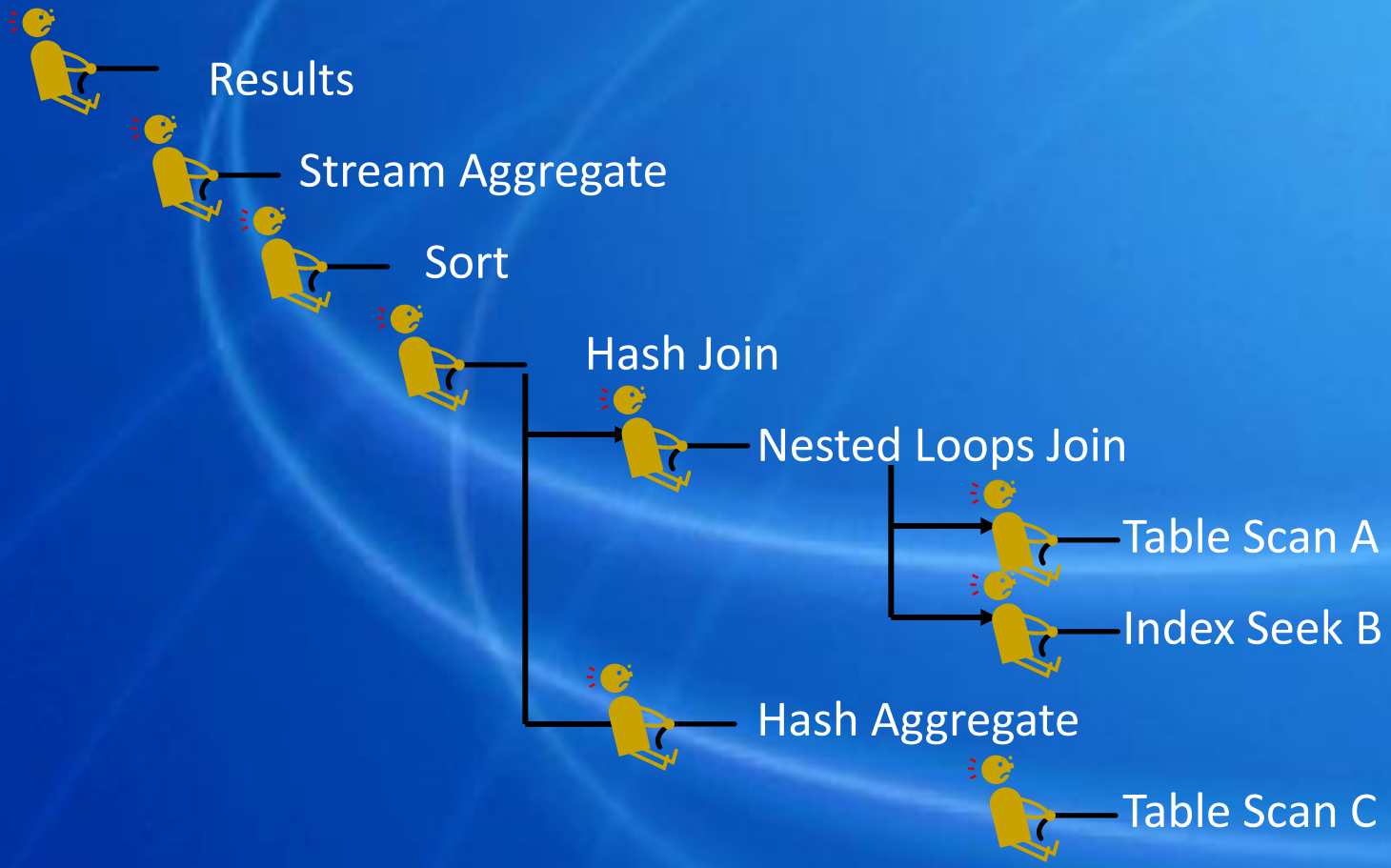
- Understand where most of the work is being done in important queries
  - Find opportunities to add or modify indexes, build covering indexes, or create indexed views
  - Potentially revisit schema design, denormalize, etc
- See how optimizer estimates compare to actuals
  - Explore hints to improve performance
  - Eliminate legacy hints that no longer needed
- Potentially write better SQL

Statistics and Estimates

- Statistics are used by the query optimizer to decide the query plan
- Estimates are derived using statistics
- Auto-create/update statistics
- The EstimateRows is per single execution, therefore you have to multiply it by EstimateExecutions before comparing to the ROWS value.

  Rows = EstimateRows * EstimateExcecutions

Query Plans are Trees



Reading Plans

Statement	Output Sample
STATISTICS TIME	SQL Server Execution Times: CPU time = 0 ms, elapsed time = 2 ms.
STATISTICS PROFILE	Rows Executes StmtText StmtId ---------- 47 1 SELECT * FROM [charge] 16 WHERE (([charge_amt]>=@1) . .
STATISTICS IO	Table 'member'. Scan count 1, logical reads 23, physical reads 0, read-ahead reads 0.

Execute the query directly to get the plan

Reading Plans (cont.)

	Command	Execute Query?	Include Estimated Row Counts & Stats	Include Actual Row Counts & Stats
Text Plan	SET SHOWPLAN_TEXT ON	No	No	No
	SET SHOWPLAN_ALL ON	No	Yes	No
	SET STATISTICS PROFILE ON	Yes	Yes	Yes
XML Plan	SET SHOWPLAN_XML ON	No	Yes	No
	SET STATISTICS PROFILE XML	Yes	Yes	Yes

Reading SHOWPLAN_ALL and STATISTICS PROFILE Output

- StmtText – description of what work is being done
- Estimate Rows – cardinality estimate
- TotalSubtreeCost – estimated cost of this operator and all nodes underneath it in the tree
- EstimateExecutions – how many times we'll run this operator
- ExecutionWarnings- are there missing join predicates or statistics
- Rows – Total rows returned by this operator at runtime
- Executes – total number of times this operator was executed

The most common operators

- Index seek
- Index scan
- Nested loop
- Hash match
- Merge join
- Bookmark lookup

Index Seek

- Nonclustered index seek retrieval means reading B-tree entries to determine the data page that is pointed to and then retrieving the page, going back to the B-tree, retrieving another data page, and so on.

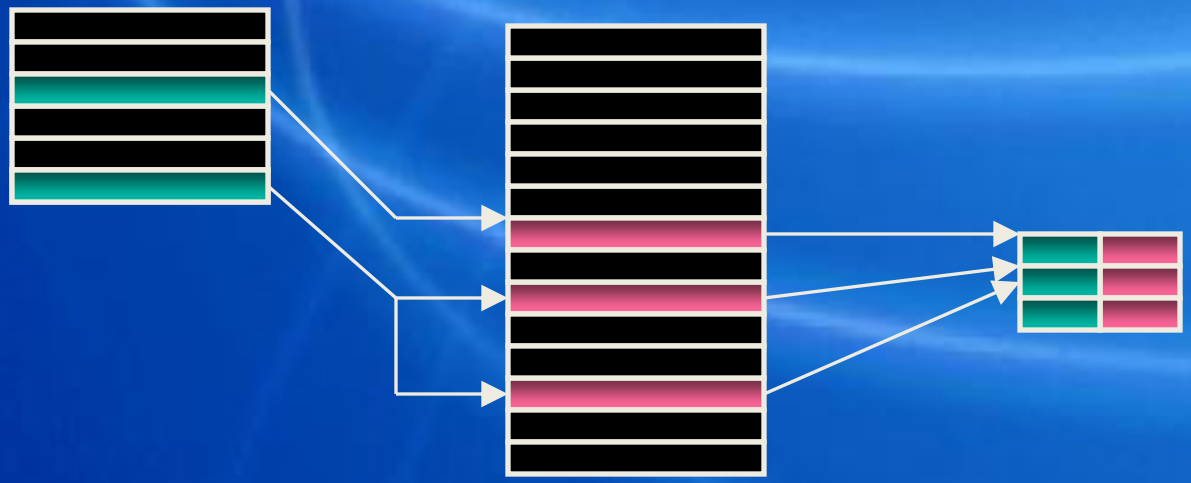
  Generally used when.
  - Highly selective.
  - Seek() predicate contains the leading columns of the index in SQL 2000 or index name in SQL 2005.
- Clustered Index Seek.
  - Uses the seeking ability of indexes to retrieve rows from a clustered index.
  - Seek() predicate contains the columns used for seeking in SQL 2000 or index name in SQL 2005.

Index Scan

- A scan, the alternative to Index Seek, means that SQL Server will stay at the leaf level and traverse just that one level of the index horizontally from the first page to the last.
- Clustered index scan (table scan)
  - Scans the clustered index specified in the Argument column.
  - The Argument column contains the name of the clustered index being used for scan

Nested Loop Algorithm

- Get Row From Outer Table
  - Get Matching Row From Inner Table
  - Output Composite Result
  - Loop Through Inner Table
  - When Inner Table Exhausted ,Loop on Outer Table



Nested loop

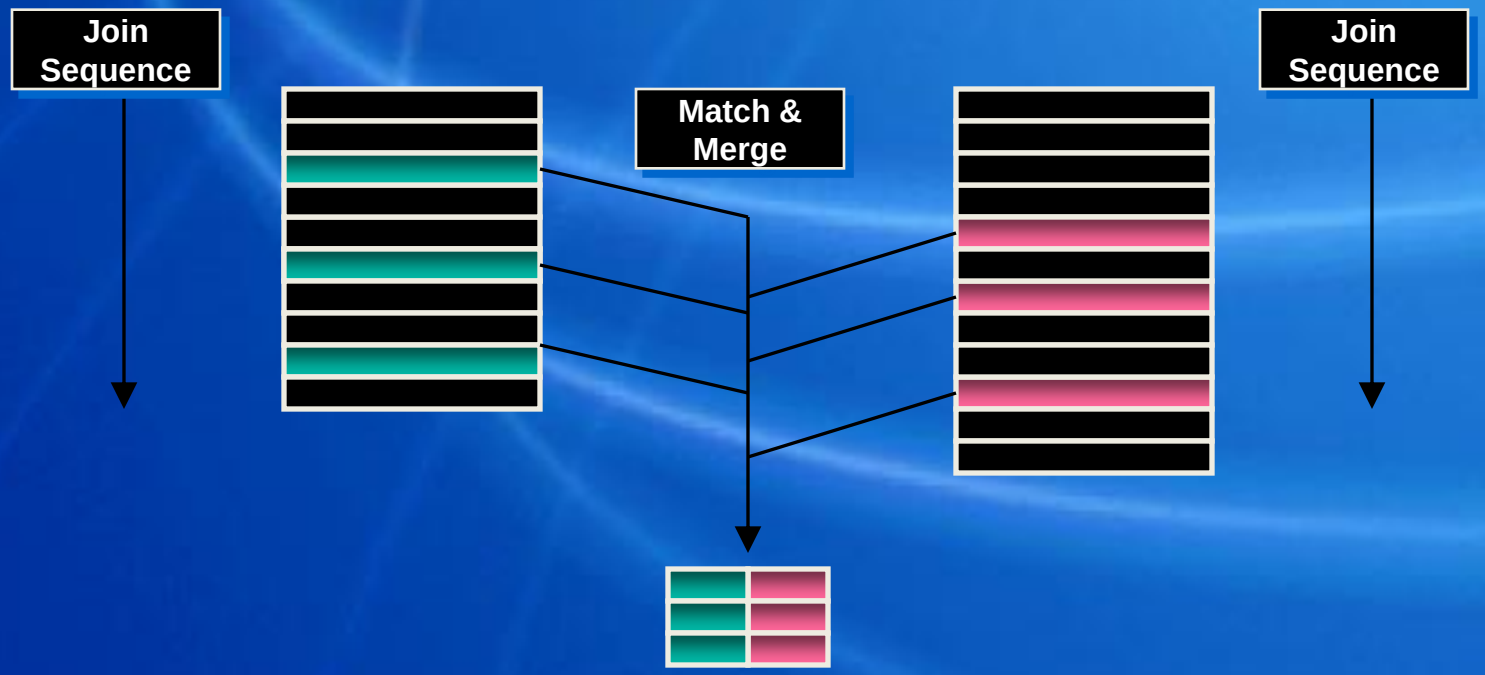
- The top input shown in the execution plan is outer input table.
- The bottom input table is inner input table.
- The outer loop consumes the outer input table row by row, for each outer row, searches for matching rows in the inner input table.
- **Effective if the outer input is *quite* small and the inner input is preindexed and quite large.**
- Low memory requirement.

- Sample query:

  SELECT O.[OrderId] FROM [Customers] C JOIN [Orders] O ON C.[CustomerId] = O.[CustomerId] WHERE C.[City] = N'London'

Merge Join Algorithm

- Get next row from outer table
- ✖ Get next row from inner table with same key
- ✖ If found, output it and go on looping on inner table
- ✖ If not found, loop on outer table



Merge Join

- **Requires that both inputs be sorted in the order of the merge column keys.**
- An index on proper set of columns is useful to provide the order.
- Small and Large tables may appear on either side of the join
- Low memory requirement.
- Usually Inner / Outer selection is not as important as in Nested Loop and Hash Joins.
- A many-to-many uses a temporary table to store rows.

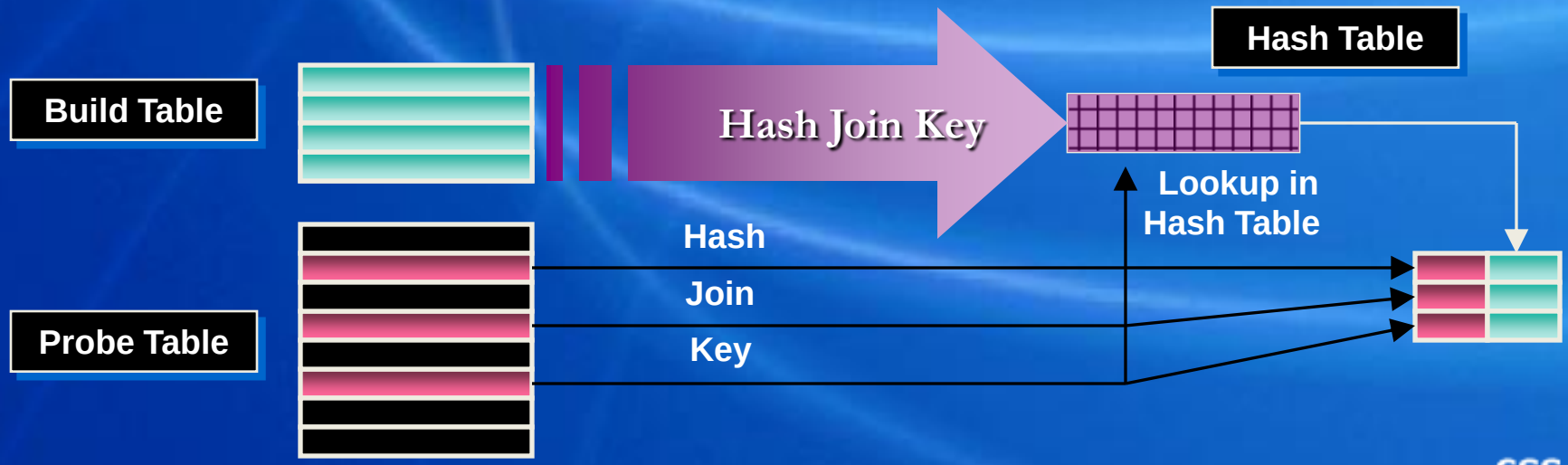
- Sample Query: SELECT O.[OrderId], C.[CustomerId], C.[ContactName]

  FROM [Orders] O JOIN [Customers] C

  ON O.[CustomerId] = C.[CustomerId]

Hash Join Algorithm

1 Scan Smaller (Build) Table
   - Hash Build Key Values; Store in Hash Table
2 Scan Larger (Probe) table
   - Hash Probe Key Value; Look Up in Hash Table
   - If Found, Output Result



Hash Join

- The top input is build input, **smaller of the two inputs.**
- The bottom input is probe input.
- Build table is completely read before first inner row is accessed.
- The larger the Build table, the more memory is needed.

- Sample Query:

```
SELECT O.[OrderId], O.[OrderDate], C.[CustomerId],
C.[ContactName]
FROM [Orders] O JOIN [Customers] C
ON O.[CustomerId] = C.[CustomerId]
```

	Nested Loops Join	Merge Join	Hash Join
Best for . . .	Relatively small inputs with an index on the inner table on the join key.	Medium to large inputs with indexes to provide order on the equijoin keys and/or where we require order after the join.	Data warehouse queries with medium to large inputs. Scalable parallel execution.
Concurrency	Supports large numbers of concurrent users.	Many-to-one join with order provided by indexes (rather than explicit sorts) supports large numbers of concurrent users.	Best for small numbers of concurrent users.
Stop and go	No	No	Yes (build input only)
Equal join required	No	Yes (except for full outer join)	Yes
Outer and semi-joins	Left joins only	All join types	All join types
Uses memory	No	No (may require sorts which use memory)	Yes
Uses tempdb	No	Yes (many-to-many join only)	Yes (if join runs out of memory and spills)
Requires order	No	Yes	No
Preserves order	Yes (outer input only)	Yes	No
Supports dynamic cursors	Yes	No	No

Multi-table joins

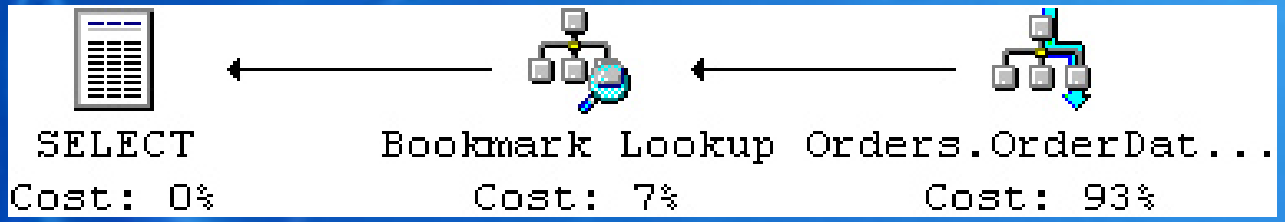
- Smaller joins first
- local predicates are applied before the join
- Joins reducing the number of rows first
- Aggregation may be performed before the join
- “FORCE ORDER” JOIN hint

Bookmark Lookups

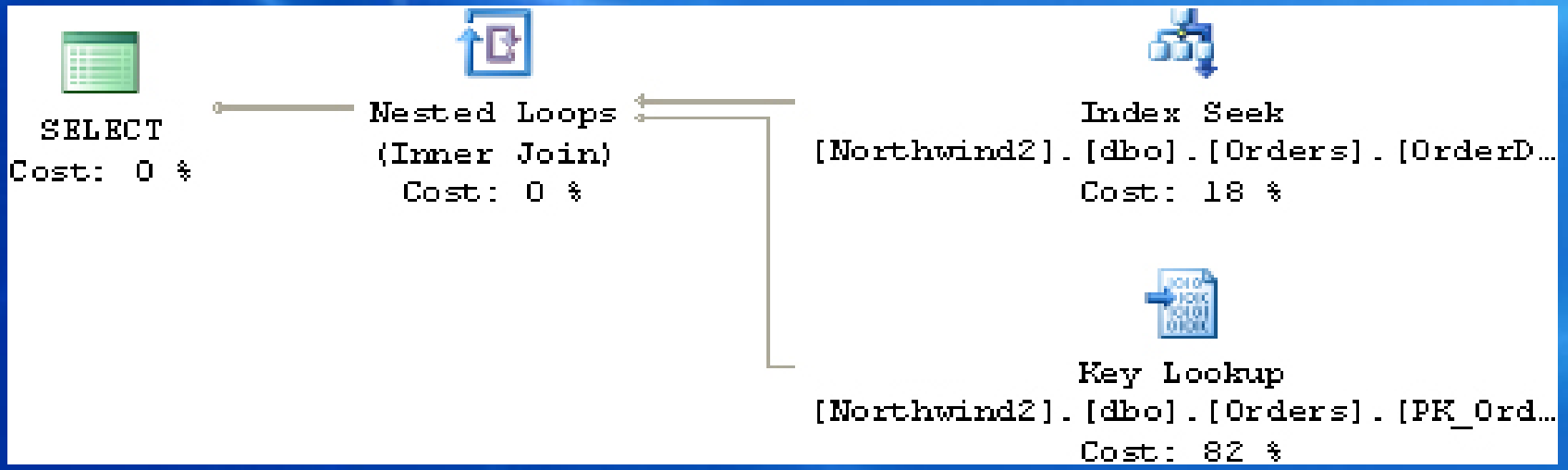
- Bookmark Lookup uses a Bookmark to look up a row in clustered index or table
- Bookmark Lookup can be removed just by adding columns to the existing index
- In some cases this may give great performance improvement

Bookmark Lookup

- SQL 2000



- SQL 2005/2008



How to eliminate the Bookmark

1. Bookmark Lookup operator shows the Table accessed with the bookmark
2. Locate Index Scan or Seek on an index on the same table following the branch from Bookmark Lookup to the right
3. Investigate what column(s) used in the query from the table are missing in the index identified in step 2 above
4. Add the columns identified in step 3 to the index found in step 2
5. Check the plan – if you didn't overlook any columns, the Bookmark Lookup is GONE

- Demo 1

What to look for?

- Large row counts or execution counts
- Large estimated costs
- Join techniques (hash, loop, merge)
- Access techniques (seeks, scans, bookmark lookups)
- Other operations (sort, top, …)

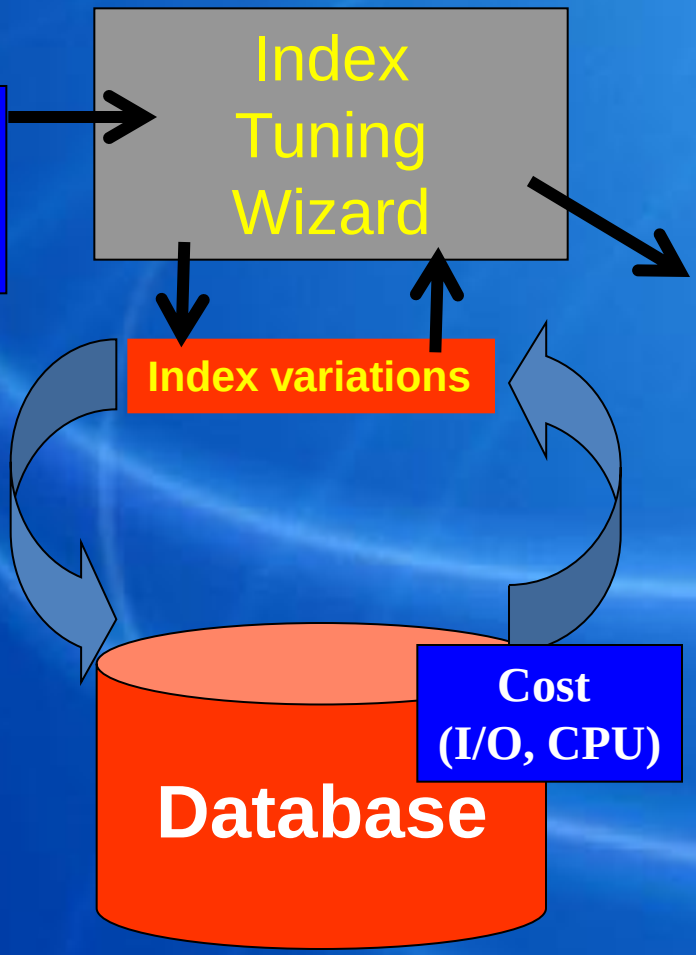
Complex Queries

- Since query plans are trees, any join branch can be as substantial as an entire separate query
- In those complex query cases
  - Identify major sub-branches first by looking top-down at the outermost joins
  - Examine leaves of the significant sub-branches separately
  - Total Subtree Cost tells you which branches are the most costly.

ITW and DTA

Input Workload:

- Trace File
- Trace Table
- SQL Script



Reports:

- Index Usage (recommend)
- Index Usage (current)
- Table Analysis
- View – Table Relations
- Query – Index Relations (Recommend)
- Query – Index Relations (Current)
- Query Cost
- Workload Analysis
- Tuning Summary

Lab1

- **Problem background:**
- Randomly a batch will bring SQL CPU 100% and can't finish. If we transfer the data in table KPI_TOL_CIF into a new table (KPI_TOL_CIF2), and execute the statements on the new table, the batch can come back to normal.

- In SQL trace TSB_20060510.trc, you could find the problematic statement is:

- INSERT INTO KPI_TOL_CIF(CUST_ID, AUM_04, AUM_04_MONTH, DATA_DATE) SELECT CUST_ID, BAL_AMT, LAST_DAY_AMT, '00950430' FROM KPI_TOL_CT_BAL03 WHERE DATA_DATE = '00950430' AND LEFT(CUST_ID,10) NOT IN(SELECT LEFT(CUST_ID,10) FROM KPI_TOL_CIF WHERE DATA_DATE = '00950430')

- The following one can be finished in 13 seconds.
- -- Dynamic SQL
- INSERT INTO KPI_TOL_CIF2(CUST_ID, AUM_04, AUM_04_MONTH, DATA_DATE) SELECT CUST_ID, BAL_AMT, LAST_DAY_AMT, '00950430' FROM KPI_TOL_CT_BAL03 WHERE DATA_DATE = '00950430' AND LEFT(CUST_ID,10) NOT IN(SELECT LEFT(CUST_ID,10) FROM KPI_TOL_CIF2 WHERE DATA_DATE = '00950430')

Lab1 (cont.)

- **Info gathered:**

- Plan1.rpt: The plan of the problematic statement (got by "set showplan_all on").
- Plan2.rpt: The plan of the statement with good table KPI_TOL_CIF2 by "set statistics profile on"
- sp_help.rpt: sp_help results of the two tables.
- sp_helpstats.rpt" sp_helpstats results of the two tables.

- **Question**
- 1. What's the difference of the two plans?
- 2. How to resolve the "hang" issue?

Troubleshooting Query plans

Is the estimate info correct?

- Check if TotalSubTreeCost is correct: EstimateRows vs Rows
- Showplan statistics profile
- No row returned will be estimated to 1 row
- Table variable is always estimated to 1 row
- Demo 2

Index seek or index/table scan?

- Seek will be good while the data returned is small compared with the data amount of whole table
- Otherwise, Scan could be beneficial
- Demo 3

Nested loop or Hash/merge join

- Nested loop is beneficial when two datasets are relatively small.
- Hash/Merge Join is beneficial when two datasets are relatively big.
- Nested loop impacts the performance while outer table is big.
- If we see executes is big, look into if nested loop matters.
- Demo 4

Where is the filter operation conducted?

- Use filter to make the dataset smaller before it is used for join.
- In most of conditions, SQL server will put the filter in the right place to make the plan optimized.

Optimizing Query Plans

The Query Optimization

- Determines the Most Efficient Execution Plan
  - Determining whether indexes exist and evaluating their usefulness
  - Determining which indexes or columns can be used
  - Determining how to process joins
  - Using cost-based evaluation of alternatives
    - Cost is estimated in terms of I/O and CPU cost

How to TUNE Query Plan

- Correct Statistics
- Create/Rebuid Index(es)
- Use different hints
- Plan Guide

Hints

- Frequent used query hints:

  | { LOOP | MERGE | HASH } JOIN

  | FORCE ORDER

  | MAXDOP number_of_processors

  | OPTIMIZE FOR

  | RECOMPILE

- Other hints: Index, nolock, rowlock, tablock

PLAN Guide

- Plan Guide is beneficial as it does not require to modify the database object.
- Take effect immediately, no need to restart service.
- Stored in each database.
- Plan Guide must exactly text match the query.
- Sp_create_plan_guide to create plan guide.
- sp_control_plan_guide to remove plan guide.

PLAN GUIDE SAMPLE 1

```
create proc Demo_Plan (@id int)
as
DECLARE @Table table (
            name nvarchar(256),
            [id] int
)
insert into @table
select a.name, a.id from
sysobjects a
inner join sysindexes b
on a.id = b.id
where b.indid =@id
```

```
EXEC sp_create_plan_guide
@name=N'Guide2',
@stmt=N'insert into @table
select a.name, a.id from
sysobjects a
inner join sysindexes b
on a.id = b.id
where b.indid =@id',
@type=N'Object',
@module_or_batch=N'Demo_Pla
n',
@params=NULL,
@hints=N'OPTION(loop join)'
```

PLAN GUIDE SAMPLE 2

```
insert into table1
select a.name, a.id from sysobjects a
inner join sysindexes b
on a.id = b.id
where b.indid =2
```

```
EXEC sp_create_plan_guide
@name=N'Guide2',
@stmt=N'insert into table1
select a.name, a.id from sysobjects a
inner join sysindexes b
on a.id = b.id
where b.indid =2',
@type=N'SQL',
@module_or_batch=N'insert into table1
select a.name, a.id from sysobjects a
inner join sysindexes b
on a.id = b.id
where b.indid =2',
@params=NULL,
@hints=N'OPTION(loop join)'
```

PLAN GUIDE SAMPLE 3

```
declare @id int
set @id = 2
DECLARE @Table table (
                name nvarchar(256),
                [id] int
)
insert into @table
select a.name, a.id from sysobjects a
inner join sysindexes b
on a.id = b.id
where b.indid =@id
```

```
EXEC sp_create_plan_guide
@name=N'Guide2',
@stmt=N'insert into @table
select a.name, a.id from sysobjects a
inner join sysindexes b
on a.id = b.id
where b.indid =@id',
@type=N'SQL',
@module_or_batch=N'declare @id int
set @id = 2
DECLARE @Table table (
                name nvarchar(256),
                [id] int
)
insert into @table
select a.name, a.id from sysobjects a
inner join sysindexes b
on a.id = b.id
where b.indid =@id',
@params=NULL,
@hints=N'OPTION(loop join)'
```

PLAN GUIDE SAMPLE 4

```
exec sp_executesql N'DECLARE @Table table (
                name nvarchar(256),
                [id] int
)
insert into @table
select a.name, a.id from sysobjects a
inner join sysindexes b
on a.id = b.id
where b.indid =@id',N'@id int',@id = 2
```

```
EXEC sp_create_plan_guide
@name=N'Guide2',
@stmt=N'insert into @table
select a.name, a.id from sysobjects a
inner join sysindexes b
on a.id = b.id
where b.indid =@id',
@type=N'SQL',
@module_or_batch=N'DECLARE @Table table (
                name nvarchar(256),
                [id] int
)
insert into @table
select a.name, a.id from sysobjects a
inner join sysindexes b
on a.id = b.id
where b.indid =@id',
@params=N'@id int',
@hints=N'OPTION(loop join)'
```

Lab2

- Problem:

  When run the query below, it also use nested loop to join tables regardless how much data in the table. However, the performance is bad and CPU is high when data is large.

  exec dbo.proc_MSS_GetCrawlHistory @ContentSourceID=4,@MaxRecords=1,@BeginTime='2009-09-07 05:26:35:360',@EndTime='2009-11-06 05:26:35:360',@CrawlStatus=11

- Question:

1. Why SQL Server always chooses nested loop even though the performance is bad.
2. How can we change SP using query hint to make it use hash match?
3. Can plan guide help on this problem? How can I create the plan guide?

Plan Caching issue

Key concept

- Compile
- Plan caching
- Recompile
- Plan reuse

How to trace plan Caching

- Profiler:
  - SP:CacheHit
  - SP:CacheInsert
  - SP:CacheMiss
  - SP:CacheRemove
  - SP:Recompile
  - SQL:StmtRecompile
- DMV:SYS.DM_EXEC_CACHED_PLANS

Plan CACHING

- **What's plan caching?**
- **How it helps?**
  - No need to recompile i.e plan reuse
- **SQL Server can cache query plans for many types of batches** :
  - Ad-hoc queries
  - Auto-parameterized queries
  - sp_executesql procedure
  - Stored procedures (including triggers)

Plan Caching (cont.)

- **SQL Server can not cache query plans when:**
  - Query plans used to create or update statistics
  - Query plans used to execute DBCC commands
  - Query plans used to create or rebuild indexes
  - Dynamic SQL executed via EXEC()
  - any stored procedure or query executed with RECOMPILE
  - Any query that contains a string or binary literal larger than 8KB
  - Zero cost plan (changed from 2005)

Plan caching' life cycle

- The cost is a measure of the server resources (CPU time and I/O) it takes the query optimizer to optimize the batch.
- The cost is aged using some algorithm.
- When a query plan or an execution context is reused, its cost is reset back to its compilation (or execution context generation) cost
- Remove Query Plan
  - Recompile (statement level)
  - With Recompile hint (sp level)
  - Memory pressure on buffer pool
  - Manually cleanup (DBCC FREEPROCCACHE)

Parameter Sniffing

- When a stored procedure is compiled for the first time, the values of the parameters supplied with the execution call are used to optimize the statements within that stored procedure.
- If these values are typical, then most calls to that stored procedure will benefit from an efficient query plan.
- At the mercy of whether the plan was first called with typical or atypical parameters as to what gets cached

Lab3

- Create test sp “sniff” following by the lab script.
- Test1: dbcc freeproccache; go;

  Exec sniff 50000;

  Exec sniff 75124
- Test2: dbcc freeproccahce; go;

  Exec sniff 75124

  Exec sniff 50000
- What is the difference? Why it is different?

How to resolve parameter sniffing

- Use EXEC() to call SQL query dynamically.
- Use local variable
- Use query hint:
  - “with recompile”
  - Force to use specified join { LOOP | MERGE | HASH }
  - “Optimize for”
- Plan guide

LAB3 (cont.)

- Use the lab script to create new SP and test how the parameter sniffing is resolved.

Review

- Index structure
- Reading Query Plans
- Troubleshooting Query Plans
- Optimizing Query Plans
- Plan Caching

Q & A

Thank You!

Microsoft®

Your potential. Our passion.™