

# Seminar 7

## Performance Tuning in SQL Server (II)\*

\* Not among the exam topics

# Stored Procedures

## ■ Advantages

- Performance advantages
- Server side
- Reuse of execution plans

## ■ Note: Requirements for plan reuse

- Plan reuse is not always a good thing

## ■ New in SQL Server 2005:

- OPTIMIZE FOR / RECOMPILE query hints

# Stored Procedures – Optimization Tips

## ■ **SET NOCOUNT ON**

- Number of affected rows is not displayed
- Reduces network traffic

## ■ Use **schema name** with **object name**

- Helps finding directly the compiled plan

```
SELECT * FROM dbo.MyTable  
EXEC dbo.StoredProcedure
```

# Stored Procedures – Optimization Tips

- Do not use **sp\_** prefix
  - SQL Server first searches in the **master** database and then in the **current** database
- use UNION to implement an "OR" operation
- Avoid joins between two types of columns
  - Index is not used for a converted column!

# Stored Procedures – Optimization Tips

## ■ **sp\_executesql** vs **exec**

- Execution plan of a dynamic statement can be reused if ALL characters of two consecutive executions are exactly the same
- EXEC ('Select \* from Categories where ID = 1')
- EXEC ('Select \* from Categories where ID = 2')
- EXECUTE sp\_executesql N'Select \* from Categories where ID = @ID', N'@ID int', @ID=1;

# Stored Procedures – Optimization Tips

## Cursors

- Generally use a lot of SQL Server resources and reduce the performance and scalability of applications
  
- Scenarios where cursors are suitable/better:
  - Procedural logic / must access the data in a row-by-row manner
  - Ordered calculations

# Stored Procedures – Optimization Tips

- Do not use COUNT() in a subquery to do an existence check
- Use IF EXISTS (SELECT 1 FROM... )
  - The output of nested select is not used
  - Reduces processing time and network transfer
- Keep transactions short
  - Transactions' length affects blocking and deadlocking

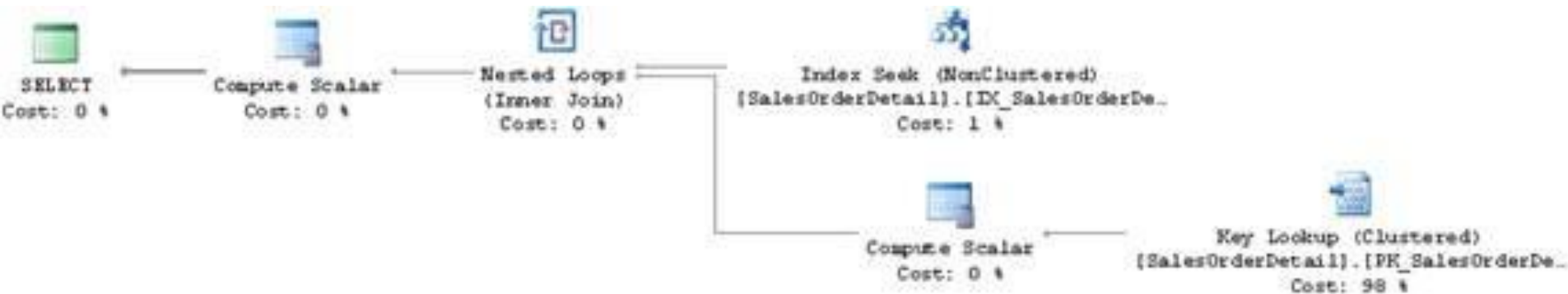
# Stored Procedures – Optimization Tips

## Reuse execution plans

```
CREATE PROCEDURE test (@pid int)
AS
```

```
    SELECT * FROM Sales.SalesOrderDetail
    WHERE ProductID = @pid
```

**exec test(897)**



**exec test(870)**





# Stored Procedures – Optimization Tips

## OPTIMIZE FOR / RECOMPILE query hints

```
ALTER PROCEDURE test (@pid int)
AS
    SELECT * FROM Sales.SalesOrderDetail
    WHERE ProductID = @pid
    OPTION (OPTIMIZE FOR (@pid = 870))
```

```
ALTER PROCEDURE test (@pid int)
AS
    SELECT * FROM Sales.SalesOrderDetail
    WHERE ProductID = @pid
    OPTION (RECOMPILE)
```

# Stored Procedures – Optimization Tips

## OPTIMIZE FOR UNKNOWN

- local variables are not known at optimization time
- example below: always generates the same execution plan

```
ALTER PROCEDURE test (@pid int)
AS
    DECLARE @lpid int
    SET @lpid = @pid
    SELECT * FROM Sales.SalesOrderDetail
    WHERE ProductID = @lpid
```

# Stored Procedures – Optimization Tips

## OPTIMIZE FOR UNKNOWN

- local variables are not known at optimization time
- example below: always generates the same execution plan

```
ALTER PROCEDURE test (@pid int)
AS
    SELECT * FROM Sales.SalesOrderDetail
    WHERE ProductID = @pid
    OPTION (OPTIMIZE FOR UNKNOWN)
```

# Stored Procedures – Optimization Tips

## OPTIMIZE FOR query hints

```
DECLARE @city_name nvarchar(30);  
DECLARE @postal_code nvarchar(15);  
  
SELECT * FROM Person.Address  
WHERE City = @city_name AND PostalCode =  
@postal_code OPTION  
(OPTIMIZE FOR (@city_name = 'Seattle',  
@postal_code UNKNOWN) );
```

# Stored Procedures – Optimization Tips

## Other query hints

- HASH GROUP / ORDER GROUP

```
SELECT ProductID, OrderQty, SUM(LineTotal) AS Total
FROM Sales.SalesOrderDetail
WHERE UnitPrice < $5.00
GROUP BY ProductID, OrderQty
ORDER BY ProductID, OrderQty
OPTION (HASH GROUP, FAST 10);
```

# Stored Procedures – Optimization Tips

## Other query hints

- MERGE UNION / HASH UNION / CONCAT UNION

```
SELECT ...  
UNION  
SELECT ...
```

```
OPTION ( MERGE UNION )
```

# Stored Procedures – Optimization Tips

## Join hints

- LOOP JOIN / MERGE JOIN / HASH JOIN

```
SELECT * FROM Sales.Customer AS c
INNER JOIN Sales.vStoreWithAddresses AS sa
        ON c.CustomerID = sa.BusinessEntityID
WHERE TerritoryID = 5
OPTION (MERGE JOIN);
GO
```

# Stored Procedures – Optimization Tips

## Join hints

- FAST n - focus on returning the first 'n' rows as fast as possible

```
SELECT * FROM Sales.Customer AS c
INNER JOIN Sales.vStoreWithAddresses AS sa
        ON c.CustomerID = sa.BusinessEntityID
WHERE TerritoryID = 5
OPTION (FAST 10);
GO
```



# Stored Procedures – Optimization Tips

## Join hints

- FORCE ORDER – “force” the optimizer to use the order of joins as they are listed in the query

```
SELECT * FROM Table1  
INNER JOIN Table2 ON Table1.a = Table2.b  
INNER JOIN Table3 ON Table2.c = Table3.d  
INNER JOIN Table4 ON Table3.e = Table4.f  
OPTION (FORCE ORDER);
```

# Stored Procedures – Optimization Tips

More about Controlling Execution Plans with Hints:

<https://www.simple-talk.com/sql/performance/controlling-execution-plans-with-hints/>

# Dynamic Execution

## ■ Disadvantages:

- Ugly code; hard to maintain
- Requires direct permissions (in 2000)
- Security risk of SQL Injection

## ■ Use smartly:

- Dynamic filters and sorting to get good plans
- And more....

# Temporary Tables

## ■ Useful when:

- You have intermediate result sets that need to be accessed several times
- You need a temporary storage area for data while running procedural code

## ■ Use temp tables when:

- You are handling large volumes of data, where plan efficiency is important and non-trivial

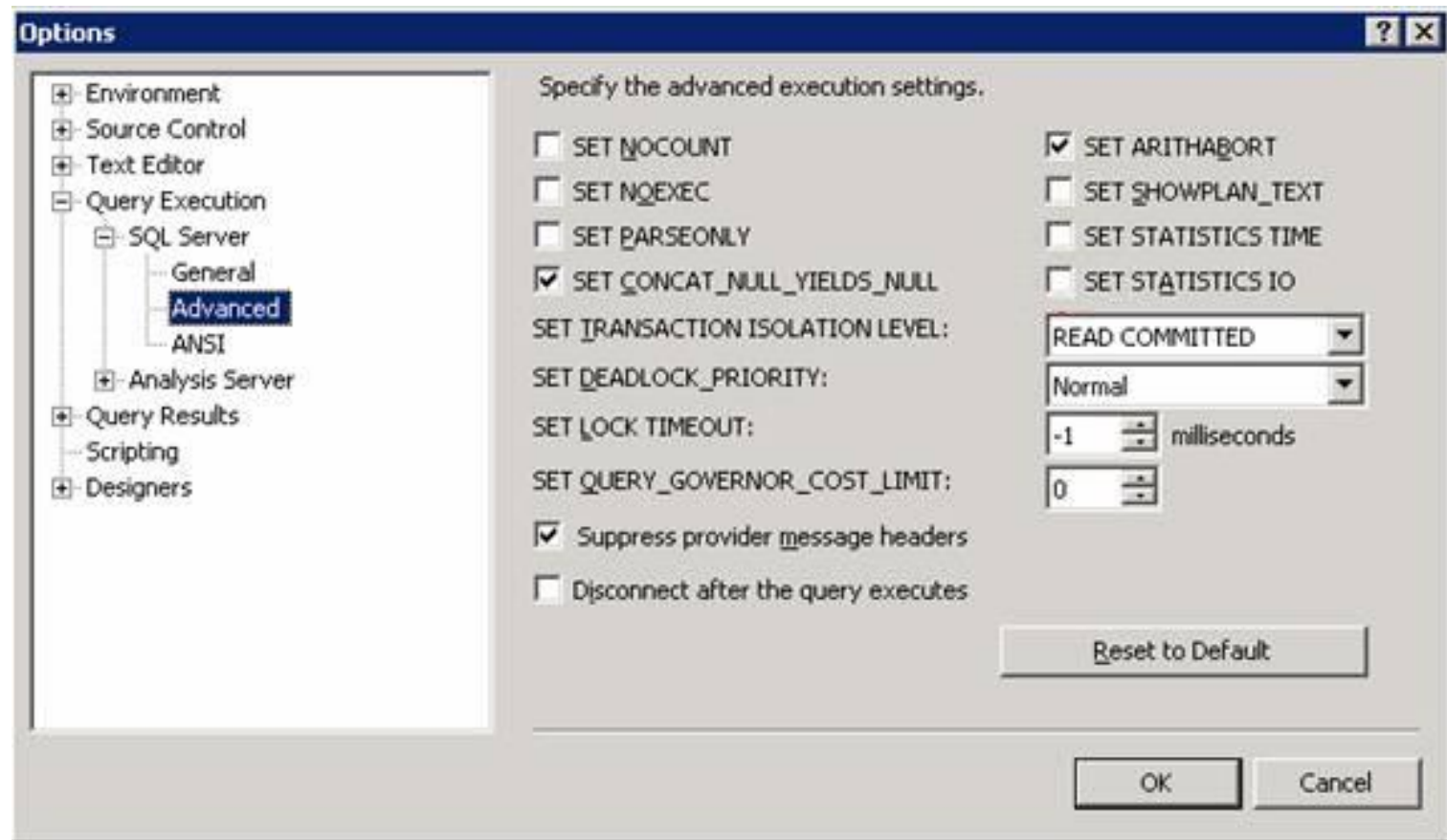
## ■ Use table variables when:

- You are handling small volumes of data, or when plans are trivial

# Triggers

- Expensive
- Main performance impact: accessing *inserted* and *deleted*
  - SQL Server 2000: transaction log
  - SQL Server 2005: row versioning (tempdb)
- Try to utilize set-based activities
- Identify affected rows and react accordingly
- UPDATE triggers record deletes followed by inserts in the log

# SQL Server Options



# Fragmentation

- Fragmentation: has a significant effect on query performance
  - *Logical fragmentation*: percent of *out-of-order* pages
  - *Page density*: page population
- Use DBCC SHOWCONTIG to get fragmentation statistics, and examine LogicalFragmentation and Avg. Page Density (full)
- Use the *sys.dm\_db\_index\_physical\_stats* function, and examine AvgFragmentation
- Rebuild indexes to handle fragmentation

## Other statistics

### ■ Update statistics asynchronously

- ***String summary statistics***: frequency distribution of substrings is maintained for character columns
- ***Asynchronous auto update statistics*** (default off)
- **Computed column statistics**



## Other statistics

- *sys.dm\_exec\_query\_stats* - performance statistics for cached query plans
  - *total\_logical\_reads* / *total\_logical\_writes* - total number of logical reads/writes performed by executions of a plan since it was compiled
  - *total\_physical\_reads* - total number of physical reads performed by executions of this plan since it was compiled
  - *total\_worker\_time* - total amount of CPU time, in microseconds, that was consumed by executions of plan since it was compiled
  - *total\_elapsed\_time* - total elapsed time, in microseconds, for completed executions of the plan