MsSQL: Basal performance tuning, part 1

Hvordan man skriver "God SQL" for bedre performance.

Skrevet den **03. Feb 2009** af **trer** I kategorien **Databaser / MS SQL** | ★★☆☆☆

Disse artikler om basal performance tuning er uddrag af en intern anbefaling om SQL som jeg har skrevet til udviklerne på min arbejdsplads, NNIT a/s.

Bemærk i øvrigt: Artiklen her er oprettet i kategorien MSSQL og handler derfor primært om Microsoft SQL Server. Flere af optimeringstippene er dog generiske og gælder på mange platforme.

Den største gevinst under performance tuning opnår man ikke ved at skrue på serverens indstilninger, men ved at få SQL'en til at køre godt. Når der skrives SQL til triggers, stored procedures og ad-hoc queries til dataudtræk / rapportgenerering er det af stor betydning om SQL'en er velskrevet for at data udtrækkes hurtigt og med mindst serverbelastning.

Hver SQL udtryk der afvikles skal kompileres og en queryplan - dvs. en plan for hvordan data skal findes og bearbejdes - skal laves. Dårlig SQL vil gøre, at unødige data læses og bearbejdes mens god SQL vil minimere både datamængder og antal operationer.

Man bør være opmærksom på, at et SQL udtryk der fungerer på en single-processor maskine ikke nødvendigvis vil give en god ydelse på en multi-processor maskine og vice versa.

Nogle af de ting jeg gennemgår her i artiklen er rent performancemæssige, andre er primært af hensyn til driftsikkerhed.

I øvrigt, når jeg skriver "Undgå et-eller-andet" bør det læses "Undgå om muligt". Man må først fokusere på at få de korrekte data ud, dernæst at optimere forespørgslen.

Brug af hints
På SQL Server kan man angive hints til optimizeren, f.eks. vedr. valg af index, jointyper etc. Denne type hints bør man under ingen omstændigheder benytte i et driftssystem, de er beregnet til troubleshooting etc.

Problemet er, at efterhånden som data ændres (og hardwaren udvides) vil optimizeren rekompilere sql'en og lave nye queryplaner som matcher data bedre. Bruger man hints vil optimizeren over tid typisk ænde med suboptimale planer.

Brug af database defaults
Alle specielle indstillinger som er krævet for korrekt opførsel bør sættes i forbindelses-strengen når applikationen forbinder til databasen. Kun såfremt man er sikker på, at varierende indstillinger ikke har betydning for brugen af databasen bør man ignorere disse.

De forskellige indstillings-muligheder på en database kan ses ved kommandoen *sp_dboption* som er beskrevet i *Microsoft Books Online*.

Begræns kommunikation mellem server og klient
Indsæt altid udtrykket SET NOCOUNT ON i starten af en stored procedure eller trigger. Udtrykket undertrykker SQL Servers normale tilbagemelding om antal rækker berørt / opdateret etc. hvormed der

spares et roundtrip fra klient til server.

Benyt "Show Execution plan"
Query Analyzers "Show Execution Plan" viser en eksekveringsplan for et givent udtryk og kan også bruges til at foretage en sammenligning af queries - de forskellige udtryk skrives blot i samme query vindue og afvikles.

Execution plan fortæller derefter hvor stor del af den samlede omkostning det enkelte statement har. Dermed kan man nemt sammenligne forskellige sql-formuleringer og finde det mest effektive.

Prefix altid tabeller, views og stored procedures med ejer.
Når en tabel eller lignende uden ejer-prefix skal benyttes søger optimizeren først efter tabellen med aktuelle bruger som ejer. Findes den ikke søges der så efter tabellen med [DBO] som ejer.

Ejer-prefixet sikrer altså hurtigere tilgang til det søgte objekt og tillader desuden optimizeren at genbruge eksekveringsplaner således at en ny plan ikke skal gene-reres for hver bruger der skal udføre en given query.
Eks.
Korrekt SQL:

```
SELECT [col1], [col2], [col3]
FROM [dbo].[mytable]
```

Fejlagtig SQL

```
SELECT [col1], [col2], [col3]
FROM [mytable]
```

Benyt table-alias'er til at øge læsbarheden.
Ved større eller mere komplekse SQL udtræk hvor data hentes fra flere tabeller eller views kan man med fordel benytte alias'er fremfor at skrive det fulde tabel-navn (inkl. ejer) for hver brugt kolonne.
Eks.

```
SELECT p.[navn],a.[navn]
FROM [dbo].[persontabel] AS p
INNER JOIN [dbo].[afdelingstabel] AS a
ON p.[personafdeling_id] = a.[afdelingsid]
```

Undgå rå SQL i applikationer
Man bør undgå at fyre rå SQL af fra applikationer / websider. Pak i stedet alt ind i procedurer og funktioner. Dermed vil al SQL normalt ligge pre-kompileret og med færdige queryplaner fremfor at skulle dannes fra gang til gang.

Dette giver også en sikkerhedsmæssig fordel da det så kun er de pre-definerede operationer man kan udføre på databasen.

Parameteriser rå SQL
Når man benytter rå sql fra klienten / applikationen, vil SQL Serveren forsøge at parameterisere disse. Parameteriseringen består i, at konstanter erstattes med placeholders så den kompilerede SQL og Queryplanen kan genbruges.

På udtryk som de her viste er det oplagt at placere placeholders ved 17, 'Hans Hansen', 100 og 22.

```
DELETE FROM dbo.mytable
WHERE id = 17
```

```
SELECT navn, adresse
FROM dbo.medlemmer
WHERE navn = 'Hans Hansen'
```

```
UPDATE dbo.mytable
SET value = 100
WHERE id = 22
```

Desværre er det langt fra altid, at SQL Server kan gøre det, selv ved så simple udtryk som ovenfor. Tester man med et DELETE udtryk som ovenfor vil man opdage, at den nogen gange parameteriserer andre gange gør den ikke!

Man bør derfor altid parameterisere sin SQL fra applikationen,

Benyt altid kolonne-specifikationer.
Skriv altid fuld liste af kolonne-navne i SELECT og INSERT statements. Benyttes wildcards vil man typisk læse unødige data hvilket giver et overhead og senere ændringer / opdateringer af tabellerne kunne få applikationen (eller dens stored procedures og views etc) til at fejle når de underlæggende tabeller ændres.

Korrekt SQL

```
SELECT [UserName]
FROM [dbo].[mytable]
```

```
INSERT INTO [dbo].[mytable] ([col1],[col2])
VALUES ('value1','value2')
```

Fejlagtig SQL

```
SELECT *
FROM [dbo].[mytable]
```

```
INSERT INTO [dbo].[mytable]
VALUES ('value1','value2')
```

En undtagelse for ovenstående er ved en COUNT(), hvor brugen af wildcards frem for et kolonne-navn normalt sikrer optimal performance.

Angiver man i en COUNT() et kolonnenavn vil SQL Server optælle den angivne kolonne - uanset om den er indekseret. Bruges i stedet wildcard (*) vil SQL Ser-ver altid vælge en indekseret kolonne hvorfra antal

rækker hurtigt kan returneres.

Undgå brug af dynamisk SQL
Bruges dynamisk SQL i stored procedures, user-defined functions og/eller triggers vil disse objekter skulle rekompileres og en ny query plan generes for hver afvikling.

Forbehold brug af dynamisk SQL til specielle stored procedures / funktioner som kun vil blive kaldt få gange af få brugere.

Begræns brug af OUTER JOINs.
Ved outer joins tvinges SQL Server til at gennemgå hele tabel-sættet, derfor skal den join type benyttes med omtanke - specielt i forbindelse med store tabeller.

Grundlæggende; Der bør aldrig forekomme situationer hvor det er nødvendigt at foretage flere outer joins på samme tabel - er man i sådan en situation er det ofte et tegn på, at database designet er forfejlet.

I visse tilfælde vil man med fordel kunne skifte flere OUTER JOINS ud med en enkelt CROSS JOIN kombineret med INNER JOINs og WHERE-betingelser.

Undgå unødige sorterings-operationer
Sortering tager CPU tid og kræver, på SQL Server 7, plads i TEMPDB, mens det, på SQL Server 2000, primært forøger hukommelsesbruget (større sorteringer vil dog stadig, helt eller delvist, blive skrevet til TEMPDB).

Visse operationer kræver en implicit eller eksplicit sortering, og bør derfor generelt undgås såfremt applikationen / systemet ikke har behov for denne funktionalitet.

Eksplicit sortering: ORDER BY
Implicit sortering: DISTINCT, UNION, GROUP BY samt IN (sub-select)

Bemærk i øvrigt:
1) at UNION ALL ikke inkluderer en implicit sortering - ved man, at der ikke er dubletter i tabellerne kan UNION ALL benyttes frem for UNION. UNI-ON ALL kan dog stadig kræve brug af TEMPDB til mellemlagring af data.
2) At ORDER BY er nødvendig for at sikre at data returneres i en given rækkefølge, selv i forbindelse med en GROUP BY operation.
3) At ORDER BY altid bør understøttes af et index der dækker de kolonner der sorteres på.

Undgå at benytte DISTINCT
Grundlæggende bør DISTINCT ikke bruges. Generelt kan det siges, at brug af DISTINCT antyder at WHERE betingelsen ikke er korrekt angivet (resultatet af en meget gammel og meget lang diskussion om relationel database-teori).

Ved DISTINCT er der desuden et overhead pga. en implicit sortering som beskrevet ovenfor.

Benyt "EXISTS (sub-select)" frem for "IN (sub-select)"
Et udtryk der baserer sig på en IN operator kan omskrives til at benytte en EXISTS operator.

Benyttes IN operatoren skal SQL Server søge gennem det returnerede datasæt for hver resultat-række, ved EXISTS operatoren kontrolleres blot om der returne-res et datasæt eller ej, hvorved søgning undgås.

Da checket reelt set dermed er et check af NULL eller NOT-NULL kan man lade sub-select'en returnere en konstant hvilket yderligere sænker server-belastningen.
Eks. kan udtrykket

```
SELECT column
FROM [dbo].[mytable]
WHERE column IN (SELECT column FROM [yourtable])
```

Erstattes med

```
SELECT column
FROM [dbo].[mytable] AS mt
WHERE EXISTS(
SELECT 1 FROM [dbo].[yourtable] AS yt
WHERE yt.column = mt.column
)
```

På SQL Server 2000 vil Query Rewriteren - en del af Query Optimizeren - som regel kunne ændre et IN udtryk til et EXISTS hvorved performance bibeholdes. Dette er dog ikke altid tilfældet - specielt ikke ved komplekse statements.

Undgå ikke-optimerbar SQL
De ikke-optimerbare betingelser forhindrer enten helt brug af indeks eller forhindrer effektiv brug af indeks. Dermed kommer man nemt ud i full-table scan for at finde blot få rækker.

Statements der benytter OR eller IN (value,value,,,) vil typisk tvinge SQL Ser-ver til gentagne indeksopslag med en efterfølgende merge eller hash-operation for at udfinde data.

Eks. vil følgende WHERE betingelse give 3 søgninger gennem index med efterfølgende merge:

```
WHERE x = 8 OR y = 'A' OR z = 'Hans'
```

Statements der benytter NOT vil skulle negere WHERE betingelsens udsagn, det kan f.eks. betyde at betingelsen omskrives til en OR (i.e. NOT X = 5 omskrives til X < 5 OR X > 5), hvorefter ovenstående "problem" opstår.

Ved brug af < (mindre end) og > (større end) samt BETWEEN vil optimizeren ofte vælge et tablescan frem for indexscan såfremt der udtrækkes data der ikke er omfattet af index.

Undgå non-deterministiske funktioner i WHERE betingelser
Non-deterministiske funktioner er i følge Microsofts definition funktioner der returnerer forskellige data med samme (eller ingen) input parametre.

Benyttes sådanne funktioner kan et statement reelt set ikke optimeres. Eksempler på non-deterministiske funktioner er (se flere i Books Online):

```
GETDATE(),
RAND() og
PATINDEX().
```

I viewet INFORMATION_SCHEMA.ROUTINES viser kolonnen IS_DETERMINISTIC om SQL Server opfatter en given procedure eller funktion som deterministisk eller ej.

Undgå brug af cursors
Performancemæssigt er en cursor-baseret løsning altid langsommere end en "ren" SQL løsning - og kræver tillige flere ressourcer under afviklingen.

Er en cursor påkrævet til løsning af en given opgave, skal man naturligvis vælge en cursor type som kan løse opgaven. Man skal blot sikre sig, at det er den letteste cursor man vælger.

Ofte vil det være nok at benytte en cursor erklæret med LOCAL FAST_FORWARD - hvor cursor data er read-only og der kun kan traverseres en vej gennem datasættet. I Books Online gennemgås de forskellige cursor typer og deres fordele og hvilke keywords der kan kombineres.

Bemærk at såfremt der vælges en cursortype som ikke understøtter de funktioner som efterfølgende benyttes, vil SQL Server fore-tage en implicit type-konvertering af den valgte cursor.

Man kan derfor normalt med fordel altid angive FAST_FORWARD.
Det bør bemærkes at man altid uanset den valgte cursor type bør definere cursor specifikt som LOCAL eller GLOBAL frem for at forlade sig på database default.

Ofte kan man undgå en cursor ved at tænke lidt utraditionelt, overvej f.eks følgende:

Opbyg en kommasepareret streng over alle fornavne i en tabel.

Oplagt at lave den således

```
declare @liste varchar(8000), @fornavn varchar(50)
declare crsr cursor local fast_forward
  select fornavn from data
open crsr
fetch next from crsr into @fornavn
while @@fetch_status=0 begin
  set @liste=@liste+@fornavn+','
  fetch next from crsr into @fornavn
end
close crsr
deallocate crsr
select @liste as [output]
```

Men en utraditionel løsning er langt hurtigere og kræver mindre kode:

```
declare @liste varchar(8000)
select @liste=@liste+fornavn+','
from data
select @liste as [output]
```

God fornøjelse
Troels

Kommentar af driis d. 19. Dec 2004 | 1

God artikel !

Kommentar af skwat d. 15. Mar 2005 | 2

Det er ganske simpelt pisse godt, har du med mssql at gøre SKAL du læse alle trers artikler.

Kommentar af nih d. 13. Aug 2004 | 3

Kommentar af lomse d. 26. Jan 2004 | 4

Hvis, jeg kendte noget til emnet, er det sikkert en rigtig fed artikkel. men det kunne lige så godt være skrevet på kinesisk, jeg fatter intet af det. Kunne du ikke skrive en mere sigende tittel på den. Feks. basal performance tuning af SQL.

Kommentar af arne_v d. 18. Dec 2004 | 5

Glimrende.

Kommentar af rosenbeck d. 16. Feb 2004 | 6

Kommentar af squashguy d. 25. Feb 2004 | 7

Kommentar af vallemanden d. 24. Feb 2004 | 8

sql er mange ting! skriv dog hvad det handler om(MSSQL, MySQL, O.S.V.) inden man smider 5 point væk(Heldigvis er det jo ingen ting)