Set-based programming

(and procedural programming)

www.everyloop.com

Procedural programming

Många programmeringsspråk, exempelvis c# och java, är så kallade procedural programming languages. Den typen av språk bygger på att funktioner exekveras i en given ordningsföljd – en efter en.

Genom att förse kompilatorn med en lång lista av instruktioner så talar vi om för den **HUR** vi vill att något ska utföras.

Denna typ av programkonstruktion går att använda även i T-SQL, men vi ska kolla på varför detta oftast inte är den bästa lösningen …

Set-based programming

SQL är ett i grunden set-baserat språk. Det är konstruerat för att behandla ett helt set av data åt gången, snarare än att iterera igenom data ett element i taget.

När vi skickar en query till databashanteraren så talar vi bara om för den **VAD** den ska göra, **INTE HUR** den ska göra det. Databashanteraren är byggd för att på egen hand ta fram en optimal plan över hur våran query ska exekveras, och kommer därefter utföra operationer på ett helt set av data enligt den planen.

Flödes-kontroll i T-SQL

Även om T-SQL i grunden är set-baserad så finns där också många av de klassiska konstruktionerna för flödeslogik (if-satser, while-loopar, try-catch) som vi finner i språk som Java och c#.

Vi kommer i denna lektion kolla på hur man använder dessa, då de kan vara användbara i vissa fall, men ha i åtanke att relationsdatabaser är byggda och optimerade för att främst hantera data i set.

If – else

```
if exists (select 1 from people where id = @id)
begin
        print 'updating record’;
        update people set name = @name where id = @id;
end
else
begin
        print 'inserting record’;
        insert into people (id, name) values(@id, @name);
end;
```

while

```
declare @count int = 0;

while @count <= 10
begin
        print 'Inserting row #' + convert(varchar, @count);
        insert into testdata (id, data) values(@count, 'test');
        set @count = @count + 1;
end
```

Cursors

Cursors är ett sätt att iterera genom ett recordset – alltså resultat av en select-sats – rad för rad. Det gör man genom att deklarera en cursor som definerar ett recordset, ”öppnar” cursorn och använder kommandot **fetch next** i en loop för att få ut nästa rad – tills alla man gått igenom alla rader.

Det finns olika typer av cursors i SQL med olika för och nackdelar. Vi kommer inte fördjupa oss i cursors, men ska kolla på ett exempel på en cursor av typen FAST_FORWARD, som är den snabbaste och som inte påverkas av förändringar i datakällan. Den är FORWARD_ONLY och READ_ONLY.

FAST_FORWARD

```
declare @id int;
declare @name varchar(100);

declare myCursor CURSOR FAST_FORWARD for
        select id, name from people;

open myCursor;
fetch next from myCursor into @id, @name;

while @@FETCH_STATUS = 0
Begin
        print convert(varchar, @id) + ' = ' + @name;
        fetch next from myCursor into @id, @name;
end;

close myCursor;
deallocate myCursor;
```

Problemet med loopar i SQL.

När databasmotorn tar emot en set-baserad instruktion så kommer den bryta ner den i sina beståndsdelar och för varje del försöka välja den mest effektiva algoritmen. Den bygger en så kallad exekverings-plan. Detta är en komplex process som tar hänsyn till en mängd variabler för att komma fram till det mest optimala sättet att utföra uppgiften. Det är så här relationsdatabaser är byggda för att fungera.

Genom att försöka skriva en rad-för-rad lösning med hjälp av cursors eller while-loopar motarbetar man ofta syftet med relationsdatabaser.

En set-baserad lösning är i princip alltid snabbare!

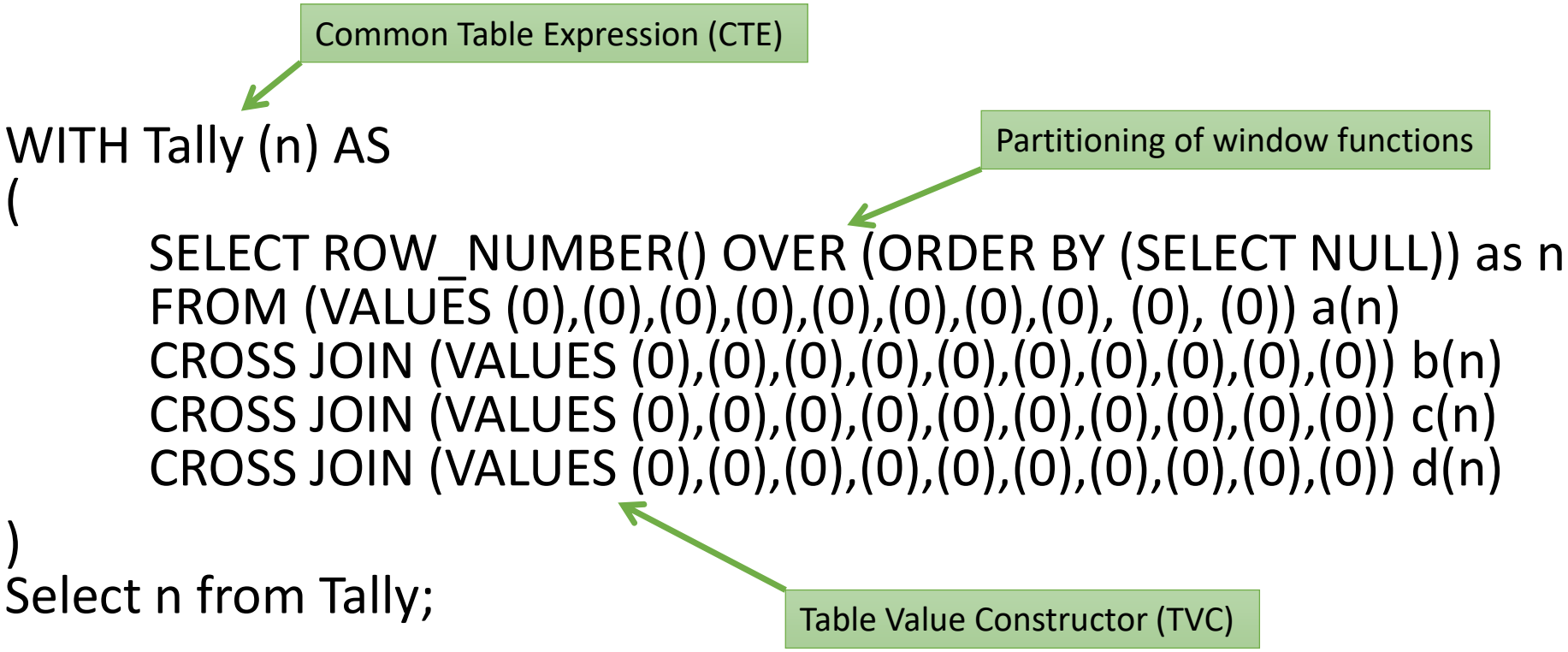
Tally table

När man i SQL vill lösa uppgifter där man i andra språk hade använt en for-loop, så kan man ha nytta av en så kallad ”Tally table”. Detta är en tabell med endast en kolumn, där första raden har värde 1, nästa värde 2, sedan 3, 4, 5, etc…

Denna tabell kan man sedan använda i set baserade uttryck för att i ”ett svep” utföra samma sak som man annars hade gjort med en loop.

Det finns en mängd problem som kan lösas väldigt snabbt i SQL med hjälp av en Tally table.

Set based generation of a tally table




```
WITH Tally (n) AS
(
        SELECT ROW_NUMBER() OVER (ORDER BY (SELECT NULL)) as n
        FROM (VALUES (0),(0),(0),(0),(0),(0),(0),(0), (0), (0)) a(n)
        CROSS JOIN (VALUES (0),(0),(0),(0),(0),(0),(0),(0),(0),(0)) b(n)
        CROSS JOIN (VALUES (0),(0),(0),(0),(0),(0),(0),(0),(0),(0)) c(n)
        CROSS JOIN (VALUES (0),(0),(0),(0),(0),(0),(0),(0),(0),(0)) d(n)
)
Select n from Tally;
```

Sub-queries

Ofta använder vi en query i en annan query. Detta kallas för sub-query. En subquery skrivs inom paranteser () och kan användas i bland annat ”where”, ”having”, och ”from”. De kan även användas för att beskriva specifika kolumner i en select-sats.

Vanligtvis exekveras subquery:n först och resultatet används sedan i den yttre query:n.

Om den inre query:n använder värden från den yttre, så kommer den behöva exekveras för varje rad i den yttre, vilket oftast går långsamt.