Optimering

Index, exekveringsplaner och distributionsstatistik

www.everyloop.com

Heap table

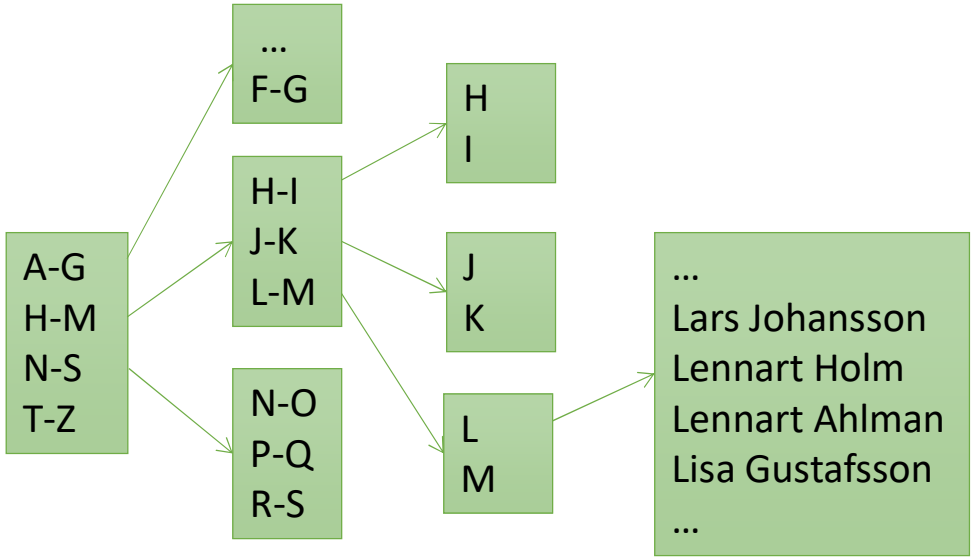
Heap tables lagrar ostrukturerad data – en ”hög” av osorterade, ostrukturerade records. När man lägger till nya rader slängs ny data bara högst upp på högen, vilket gör det snabbt.

Om man däremot vill hitta rader med ett specifikt värde (where) så är enda lösningen att databashanteraren går igenom alla rader i tabellen eftersom den annars inte kan veta vilka rader som har det specifika värdet.

Clustered index

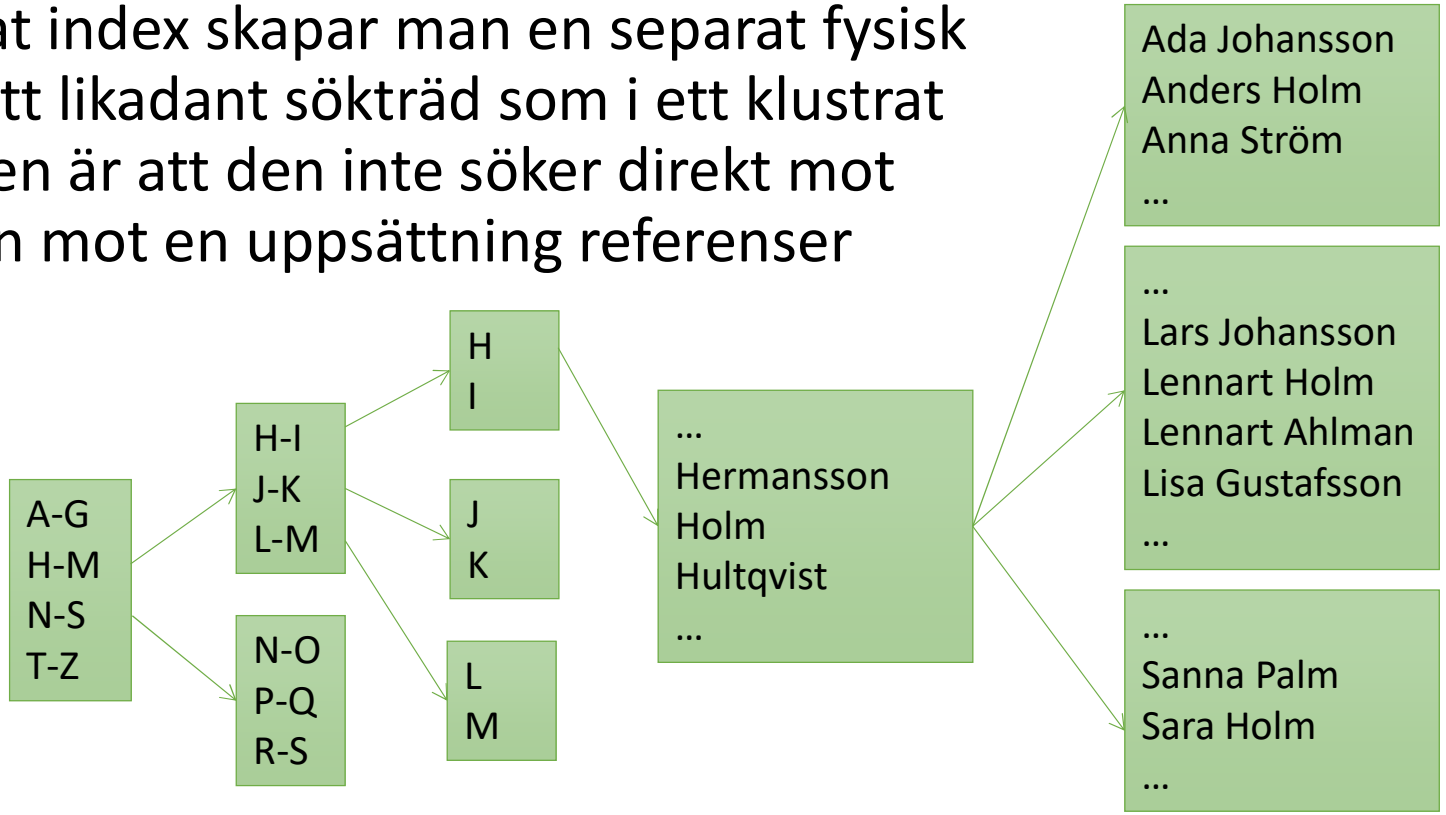
I en tabell med ett klustrat index så är datan lagrad sorterad efter indexet. Man kan bara ha ett klustrat index per tabell eftersom datan förstås bara kan vara sorterad fysiskt efter en kolumn åt gången.

Med index menas att man har information om var i tabellen det data man söker finns lagrat. Databashanteraren går igenom ett sökträd.



Non clustered index

För ett oklustrat index skapar man en separat fysisk struktur med ett likadant sökträd som i ett klustrat index. Skillnaden är att den inte söker direkt mot tabelldata, utan mot en uppsättning referenser till tabelldata.



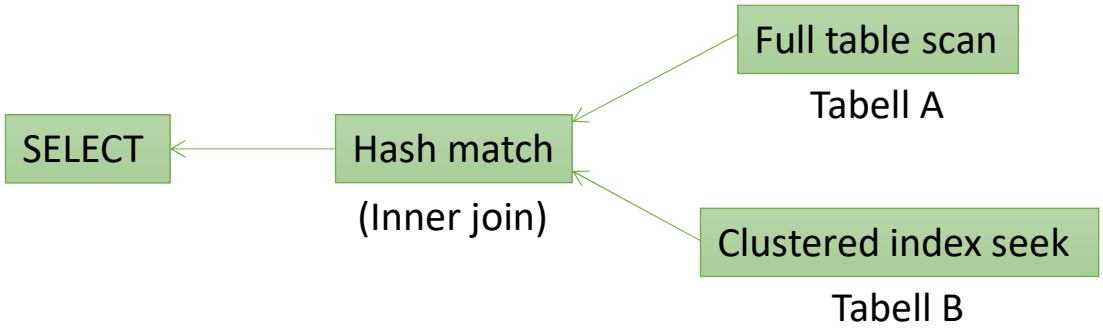
Fragmentation

SQL server lagrar data i sidor per 8KB. När man sätter in ny data i en tabell så kommer den i första hand skriva på befintliga sidor, och i andra hand skapa nya sidor. När man tar bort och sätter in data i en tabell så kommer det med tiden uppstå fragmentering. Denna beror antingen på att många sidor bara till viss del är fyllda (intern fragmentering), eller att sidorna hamnat i oordning (extern fragmentering).

Ju mer fragmenterat ett index är, ju längre tid kommer det ta att söka i det. Man kan därför behöva organisera eller bygga om index efter många skrivoperationer. Med en underhållsplan kan det automatiseras.

Execution plan

Innan databashanteraren verkställer en query så måste den ta fram en plan för hur den ska exekveras. Kom ihåg att i set-baserad programmering beskriver vi **VAD** som ska utföras, men inte **HUR**.

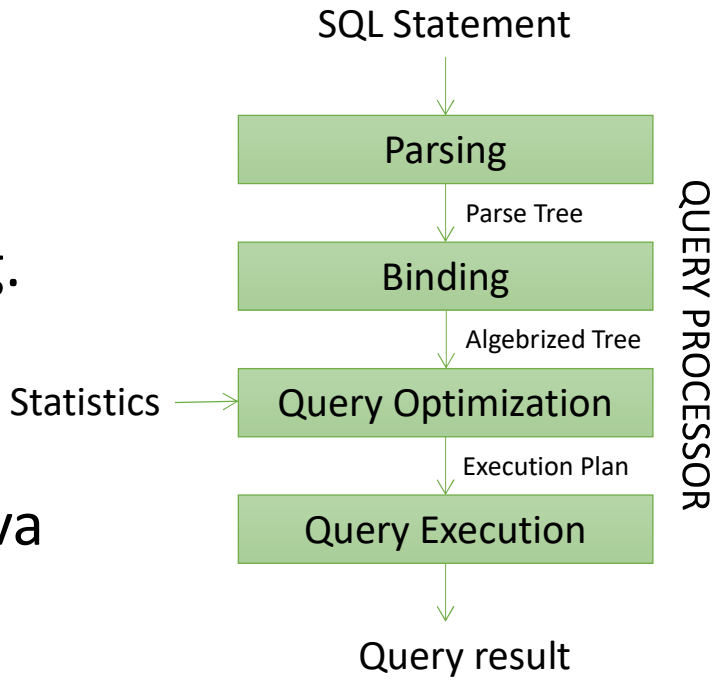


Grafisk visualisering av en exekveringsplan

Query processing

Vid exekvering så skickas SQL-koden genom de två första stegen – parsing och binding – som tolkar syntax och binder till faktiska databasobjekt. Resultatet blir ett träd av de **logiska operationer** som krävs för exekvering.

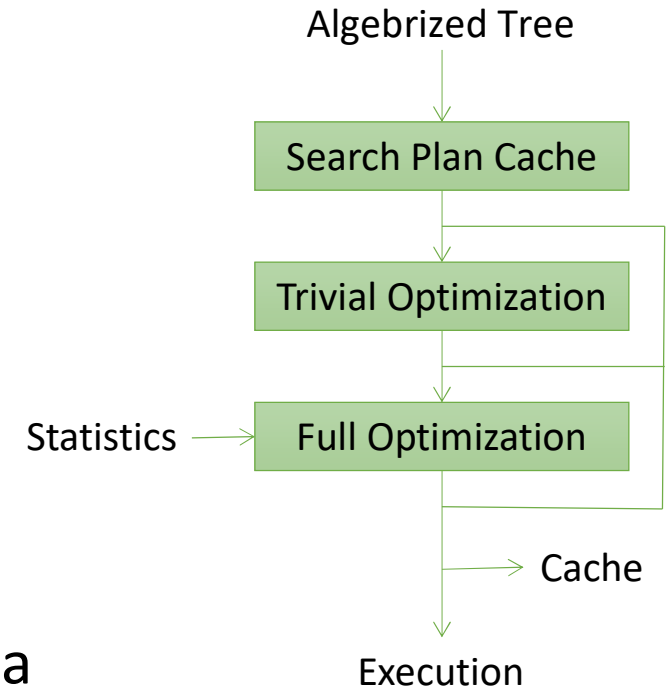
Utifrån det trädet, och tillgänglig statistik försöker optimizern ta fram den mest effektiva planen, med de **fysiska operationer** som beskriver utförandet.



Query optimzer

Om det inte finns en cachad plan för trädet så blir första steget att avgöra om det finns en trivial lösning. För vissa enkla queries så finns bara ett rimligt sätt att lösa uppgiften på, och man kan då hoppa över de steg som krävs för att dra igång en full optimeringsprocess.

För komplexa queries utförs mer syntaktiska transformationer, och insamling av tillgänglig statisik för kolumner och index, för att förbereda för full optimering.



Cost-based optimization

Genom transformeringsregler kan en mängd alternativa planer tas fram. Dessa planer kan sedan analyseras; för varje plan uppskattas en ”kostnad” (avseende resurser som I/O, CPU, Minne), och den plan som har lägst kostnad väljs.

Kostnaden beror dels på val av algoritmer, dels på uppskattat antal rader som behöver processas, vilket beräknas från distributionsstatistik.

Den mest optimerade planens kostnad måste samtidigt balanseras mot kostnaden att hitta planen i sig. Ibland lönar det sig att använda en mindre optimal plan, om det går fortare att hitta den.

Statistics

SQL server för statistik över hur kolumnvärden är distribuerade över tabellens rader. Denna statistik hjälper query-processorn att uppskatta antalet rader en query kommer returnera, vilket ger ledtrådar till hur queryn bäst processas, och vilka algorithmer som bör väljas.

Vanligtvis uppdaterar SQL server statistik automatisk när 20% + 500 rader har uppdaterats. Man kan även välja att uppdatera statistik manuellt, eller skapa statistik som är specifikt optimerad för de queries man avser köra.