

SQL Programmierung

DB Design

DB Design

- Normalisierung
- Redundanz
- Generalisierung

- Praxis
 - Architektur innerhalb des SQL Servers

Lesson 1: Designing Tables

- Normalizing Data
 - Splittung von Stammdaten und Bewegungsdaten
 - Mehrere Normalformen
 - 1.,2.,3, Codd,4.5.
- Im Regelfall wird bis zum Grad 3 normalisiert
-
- 1.NF: jede Zelle enthält einen Wert
- 2. NF jede Zeile wird durch einen Primärschlüssel eindeutig
- 3. NF alle Zeilen ausserhalb des PK enthalten keine direkten Abhängigkeiten
- PK gehen Beziehungen mit FK (1:N) ein

Common Normalization Forms

- First Normal Form
 - Eliminate repeating groups in individual tables
 - Create a separate table for each set of related data
 - Identify each set of related data by using a primary key
- Second Normal Form
 - Non-key columns should not be dependent on only part of a primary key
 - These columns should be in a separate table and related by using a foreign key
- Third Normal Form
 - Eliminate fields that do not depend on the key

Primary Keys

- The primary key uniquely identifies each row within a table
- Candidate key could be used to uniquely identify a row
 - Must be unique and cannot be NULL (unknown)
 - Can involve multiple columns
 - Should not change
 - Primary key is one candidate key
 - Most tables will only have a single candidate key
- Debate surrounding natural vs. surrogate keys
 - Natural key: formed from data related to the entity
 - Surrogate key: usually codes or numbers

Foreign Keys

- Foreign keys are references between tables:
 - Foreign key in one table holds the primary key from another table
 - Self-references are permitted
- Rows that do not exist in the referenced table cannot be inserted in a referencing table
- Rows cannot be deleted or updated without cascading options
- Multiple foreign keys can exist in one table

Vorteile der Normalisierung

- SQL Server verwendet Sperren:
 - Sperren gibt es auf Zeilen, Seiten und Tabellen
 - Sperren werden benötigt um konkurrierende Zugriffe zu steuern
 - NF unterstützt den SQL Server feineres Sperrniveau zu erhalten
- OLTP databases sollten daher immer normalisiert sein:
 - Transactions sollten kurz und schnell sein
- Datawarehouse Tabellen sollten denormalized sein

Nachteile

- Normalisierung verhindert Redundanz
- Redundanz dagegen schafft Geschwindigkeit
- Je mehr Normalisiert desto mehr JOINS werden notwendig sein
- Redundanz muss allerdings gepflegt werden

Seiten und Blöcke

- Datensätze werden in Seiten gespeichert
- 8 Seiten werden zu Blöcken zusammengefasst
- 1 Seite = 98192 bytes
- 1 Seiten max 700 Datensätze
- 1 Datensatz max 8060 bytes
- 1 Seite max 8072 bytes nutzbarer Platz für Daten
- SQL Server liest Daten seitenweise von der HDD in RAM!!! 1:1!!
- Ziel: Reduktion der Seiten!
- Evtl DB Design abweichend von Normalisierung anpassen zu Gunsten geringere Seitenzahlen
- Messung:
 - Set statistics io on
 - Dbcc showcontig(,Tabelle')
 - Select * from sys.dm_db_index_physical_stats(
db_id(DBName), object_id(,TABELLE'), ,NULL, NULL, ,Detailed'
)

Abfragen auf mehrere Tabellen

Die FROM Klausel und virtuelle Tabellen

- FROM Klausel legt die Ursprungstabellen fest, die im SELECT verwendet werden
- FROM Klausel kann sowohl Tabellen als auch Operator beinhalten
- Ergebnis der FROM Klausel ist eine virtuelle Tabelle
 - Nachfolgende Anweisungen verwenden diese virtuelle Tabelle
- FROM Klausel kann Tabellen Aliase beinhalten
 - Nützlich für spätere Phasen der Abfrage

Überblick über Joins

Join Type	Description
Cross	Kombiniert alle Zeilen beider Tabellen miteinander (Erstellt ein Kartesisches Produkt)
Inner	Startet mit einem kartesischen Produkt, setzt später aber Filter und filtert nach dem angegebenen Prädikat
Outer	Beginnt mit kartesischem Produkt; Alle Zeilen aus der zugewiesenen Tabelle werden beibehalten und die übereinstimmenden Zeilen aus anderen Tabellen abgerufen. Zusätzliche NULL als Platzhalter eingefügt

TSQL Join Syntax

- ANSI SQL-92

- Tabellen werden über JOIN Operator in der FROM Klausel verbunden

```
SELECT ...  
FROM   Table1 JOIN Table2  
       ON <on_predicate>
```

- ANSI SQL-89

```
SELECT ...  
FROM   Table1, Table2  
WHERE  <where_predicate>
```

verknüpft

Wie funktionieren Inner Joins?

- Rückgabe nur von Zeilen, die in beiden Tabellen einen Wert haben
- Vergleicht die Zeilen basierend auf einem Attribut das als Prädikat mitgegeben wird
 - ON Klausel in SQL-92 Syntax (bevorzugt)
 - WHERE Klausel in SQL-89 Syntax (nicht empfohlen)
- Wieso die Filterung in der ON Klausel?
 - Logische Unterscheidung zwischen der Filterung des Joins (ON) und des Ergebnisses (WHERE)
 - Abfrage wird nicht optimiert

Inner Join Syntax und Beispiel

- Einzelnen Tabellen werden in FROM Klausel aufgelistet
 - Getrennt durch JOIN Operatoren
- Tabellen Aliase sind empfehlenswert

```
SELECT o.orderid,  
       o.orderdate,  
       od.productid,  
       od.unitprice,  
       od.qty  
FROM Sales.Orders AS o  
     JOIN Sales.OrderDetails AS od  
     ON o.orderid = od.orderid;
```

Wie funktionieren Outer Joins?

- Gibt alle Zeilen aus einer Tabelle und alle übereinstimmenden Zeilen aus der zweiten Tabelle zurück
- Zeilen aus einer Tabelle werden beibehalten
 - Bezeichnet mit LEFT, RIGHT, FULL Schlüsselwort
 - Alle Zeilen aus der beibehaltenen Tabelle werden in der Ergebnismenge ausgegeben
- Passende Zeilen aus anderen Tabellen werden zusätzlich in der Ergebnismenge ausgegeben
- Zusätzliche Zeilen wurden zu Ergebnissen für nicht übereinstimmende Zeilen hinzugefügt
 - NULLs wurden an Stellen hinzugefügt, an denen Attribute nicht übereinstimmen

Outer Join Syntax und Beispiele

- Gibt alle Zeilen aus der ersten Tabelle zurück, nur Treffer aus der zweiten

```
FROM t1 LEFT OUTER JOIN t2 ON  
t1.col = t2.col
```

- Gibt alle Zeilen aus der zweiten Tabelle zurück, nur die Übereinstimmungen von der ersten Tabelle:

```
FROM t1 RIGHT OUTER JOIN t2 ON  
t1.col = t2.col
```

```
FROM t1 LEFT OUTER JOIN t2 ON  
t1.col = t2.col  
WHERE t2.col IS NULL
```

- Gibt nur Zeilen aus der ersten Tabelle zurück, ohne Übereinstimmung in der zweiten Tabelle:

Wie funktionieren Cross Joins?

- Kombiniert jede Zeile aus der ersten Tabelle mit jeder Zeile aus der zweiten Tabelle
- Alle möglichen Kombinationen werden ausgegeben
- Logische Grundlage für inner und outer Joins
 - Inner Join beginnt mit kartesischem Produkt, fügt Filter hinzu
 - Outer-Join verwendet kartesische Ausgabe, gefiltert, fügt nicht übereinstimmende Zeilen zurück
(mit NULL-Platzhaltern)
- Aufgrund der kartesischen Produktausgabe ist dies normalerweise keine gewünschte Join-Form

Cross Join Syntax

- Es wird nicht verglichen, keine ON Klausel benötigt
- Gibt alle Zeilen aus der linken Tabelle zusammen mit jeder Zeile aus der rechten Tabelle zurück (ANSI SQL-92 syntax):

```
SELECT ...  
FROM t1 CROSS JOIN t2
```


- Gibt alle Zeilen aus der linken Tabelle zusammen mit jeder Zeile aus der rechten Tabelle zurück (ANSI SQL-92 syntax):

```
SELECT ...  
FROM t1, t2
```

Wie funktionieren Self Joins?

- Wieso Self Joins benutzen?
 - Vergleich von zwei Spalten in gleicher Tabelle möglich
- Erstellt zwei Instanzen derselben Tabelle in der F
- Es wird mindestens ein Alias benötigt

```
SELECT e.empid, e.lastname,  
       e.title, e.mgrid, m.lastname  
FROM   HR.Employees AS e  
JOIN   HR.Employees AS m  
ON     e.mgrid=m.empid;
```



empid
lastname
firstname
title
titleofcourtesy
birthdate
hiredate
address
city
region
postalcode
country
phone
mgrid

Gruppieren und Aggregieren von Daten

Verwendung von Aggregate Funktionen

- Aggregate Funktionen:
 - Geben einen Skalar Wert zurück (ohne Spaltenname)
 - Ignoriert NULL Werte (außer in der COUNT Funktion)
 - Verwendung möglich in:
 - SELECT, HAVING und ORDER BY Klausel
 - Häufig verwendet in Verbindung mit der GROUP BY Klausel

```
SELECT AVG(unitprice) AS avg_price,  
MIN(qty) AS min_qty,  
MAX(discount) AS max_discount  
FROM Sales.OrderDetails;
```

avg_price	min_qty	max_discount
26.2185	1	0.250

Übersicht über die Aggregate Funktionen

Funktion	Beschreibung
SUM	Summiert alle Werte der Spalten auf
MIN	Gibt den kleinsten Wert der Spalte zurück
MAX	Gibt den größten Wert der Spalte zurück
AVG	Gibt den Durchschnittswert der Spalte zurück
COUNT	Gibt die Anzahl der vorhandenen Datensätze zurück

DISTINCT in Verbindung mit Aggregate Funktionen

- DISTINCT sorgt dafür, dass nur verschiedene Werte von der Funktion betrachtet werden
- DISTINCT in Aggregate Funktionen löscht NUR Werte und keine kompletten Zeilen

Beispiel:

```
SELECT empid, YEAR(orderdate) AS orderyear,  
COUNT(custid) AS all_custs,  
COUNT(DISTINCT custid) AS unique_custs  
FROM Sales.Orders  
GROUP BY empid, YEAR(orderdate);
```

empid	orderyear	all_custs	unique_custs
1	2006	26	22
1	2007	55	40
1	2008	42	32
2	2006	16	15

Wie benutze ich die GROUP BY Klausel?

- GROUP BY erstellt Gruppen für Ausgabezeilen gemäß einer eindeutigen Kombination von Werten, die in der GROUP BY-Klausel angegeben sind

```
SELECT <select_list>  
FROM <table_source>  
WHERE <search_condition>  
GROUP BY <group_by_list>;
```

- GROUP BY berechnet einen Summenwert für Aggregatfunktionen in nachfolgenden Phasen (für jede einzelne Zeile)

```
SELECT empid, COUNT(*) AS cnt  
FROM Sales.Orders  
GROUP BY empid;
```

Logische Abfolge von Operationen

Logical Order	Phase	Comments
5	SELECT	
1	FROM	
2	WHERE	
3	GROUP BY	Creates groups
4	HAVING	Operates on groups
6	ORDER BY	

- Wenn eine Abfrage GROUP BY verwendet, arbeiten alle nachfolgenden Phasen mit den Gruppen und nicht mit den Quellzeilen

GROUP BY Ablauf

```
SELECT SalesOrderID,  
SalesPersonID, CustomerID  
FROM Sales.SalesOrderHeader;
```

SalesOrder ID	SalesPerson ID	CustomerID
43659	279	29825
43660	279	29672
43661	282	29734
43662	282	29994
43663	276	29565
...
75123	NULL	18759



```
WHERE  
CustomerID IN  
(30097,  
30098)
```

SalesOrder ID	SalesPerson ID	Customer ID
51803	290	29777
69427	290	29777
44529	278	30010
46063	278	30010

```
GROUP BY  
SalesPersonID
```



SalesPersonID	Count(*)
278	2
290	2

GROUP BY mit Aggregate Funktionen

- Aggregate Funktionen werden normalerweise in der SELECT Klausel verwendet

```
SELECT custid, COUNT(*) AS count  
FROM Sales.Orders  
GROUP BY custid;
```

- Aggregate Funktionen beziehen sich auf alle Spalten, nicht nur auf die der SELECT Klausel

Filtern von Gruppierten Daten

- HAVING Klausel:

```
SELECT custid, COUNT(*) AS count_orders  
FROM Sales.Orders  
GROUP BY custid  
HAVING COUNT(*) > 10;
```

- Jeder Gruppe muss die in der HAVING Klausel benannte Bedingung erfüllen
- HAVING Klausel wird nach GROUP BY Klausel ausgeführt

Modul 10 - Unterabfragen

Was sind Unterabfragen?

- Unterabfragen sind Abfragen integriert in einer anderen Abfrage
- Ergebnisse der Unterabfrage werden an die Hauptabfrage weitergeleitet
 - Unterabfrage ist wie eine Aussage für Hauptabfrage
- Unterabfragen können in sich abgeschlossen oder korreliert sein
 - Unabhängige Unterabfragen haben keine Abhängigkeit von der äußeren Abfrage (in sich abgeschlossene)
 - Korrelierte Unterabfragen verlassen sich auf Werte aus der Hauptabfrage in der sie stehen
- Unterabfragen können an vielen Stellen auftreten:
 - Als Spalte: dann darf sich nur eine „Zelle“ aus er Abfrage ergeben
 - Als Tabellenergebnis: dann ist ein FROM (Unterabfrage) möglich, wobei die

Vergleich zwischen Unterabfragetypen

Abgeschlossene Unterabfrage:

Hauptabfrage:

```
SELECT orderid, productid,  
unitprice, qty  
FROM Sales.OrderDetails  
WHERE orderid = ( )
```



Unterabfrage:

```
SELECT MAX(orderid) AS  
lastorder  
FROM Sales.Orders
```

Hauptabfrage

```
SELECT orderid, empid, orderdate  
FROM Sales.Orders AS 01  
WHERE  
orderdate = ( )
```



Unterabfrage:

```
SELECT MAX(orderdate) FROM  
Sales.Orders AS 02  
WHERE 02.empid = 01.empid
```



EXISTS Parameter

- Wenn eine Unterabfrage mit dem EXISTS Parameter verwendet wird, dann führt sie einen Existenztest durch
 - Gibt lediglich wahr (true) oder falsch (false) zurück
- Wenn Zeilen als Ergebnis der Unterabfrage zurückgegeben werden würden, dann gibt die Unterabfrage ein true zurück anstatt der Zeilen
- Andererseits gibt die Unterabfrage ein false zurück
- `WHERE [NOT] EXISTS (subquery)`

Exists Syntax und Beispiele

- EXISTS folgt keine Spalte oder ähnliches sondern immer eine

```
SELECT custid, companyname  
FROM Sales.Customers AS c  
WHERE EXISTS (  
    SELECT *  
    FROM Sales.Orders AS o  
    WHERE c.custid=o.custid);
```

```
SELECT custid, companyname  
FROM Sales.Customers AS c  
WHERE NOT EXISTS (  
    SELECT *  
    FROM Sales.Orders AS o  
    WHERE c.custid=o.custid);
```

CTE

Andreas Rauch ppedv AG

Common Table Expression

[WITH <common_table_expression> [,...n]]

<common_table_expression>::=

expression_name [(column_name [,...n])]

AS

(CTE_query_definition)

CTE

- Temporäres Resultset
- Verwendung für:
 - Auflösung von Rekursionen
 - Vereinfachung von komplexeren Abfragen
 - bei Unterabfragen, die häufiger wiederholt werden
 - Bei mehreren Unterabfragen

CTE

- Sollten mit einem ; beginnen
 - Oder enden

Beispiel

```
;with MyCTE(x)
as
(select x='hello')
select x from MyCTE
```

```
with MyCTE(x) as
(
select x = convert(varchar(8000), 'hello')
union all
select x + 'a' from MyCTE where len(x) < 100
)
select x from MyCTE order by x;
```

```
with MyCTE(x) as
(
select x = convert(varchar(8000),'hello')
-- ANKER und Intialisierung des Einstiegspunkts der Rekursion
union all
select x + 'a' from MyCTE where len(x) < 100
--um die Rekursion zu betreiben muss zw der Tabelle dem bereits aus
dem Anker bestehenden Ergebnis (=CTE) ein Join verwendet werden
)
select x from MyCTE order by x;
--im folgenden Select kann auf die gesamte CTE bezug genommen
werden
```


Vorteile

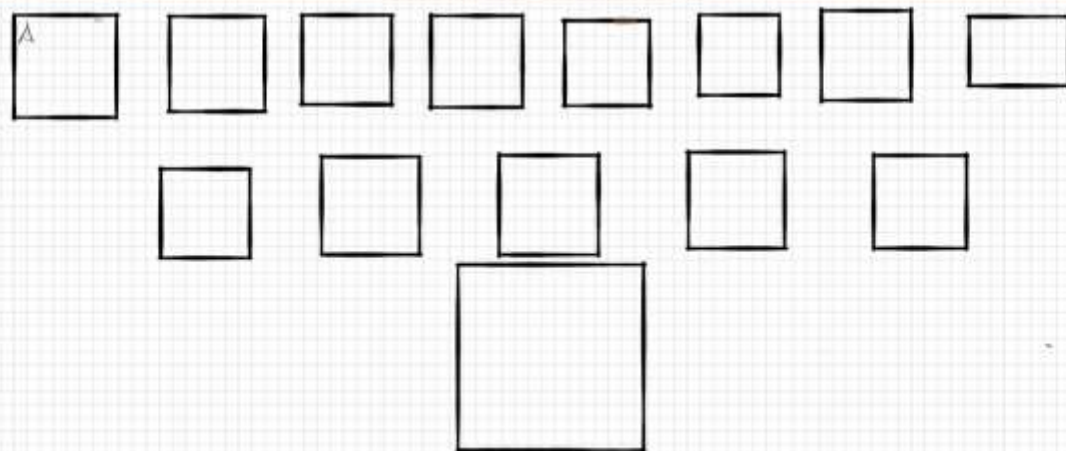
- Unterabfragen können deutlich übersichtlicher umschrieben werden
 - Lesbarkeit
- Rekursionen sind wesentlich einfacher auflösbar
- Es können Aggregatsberechnungen gemacht werden, die nicht so einfach ohne Group by gemacht werden können
-

SQL Programmierung

Indizes

Olto	EVI	PLOM	SEMP	KADJ	EVA	Olto	DILON	WACO
EVA	KADJ							

PAGES.
100.00
Datensatz pro Page
200



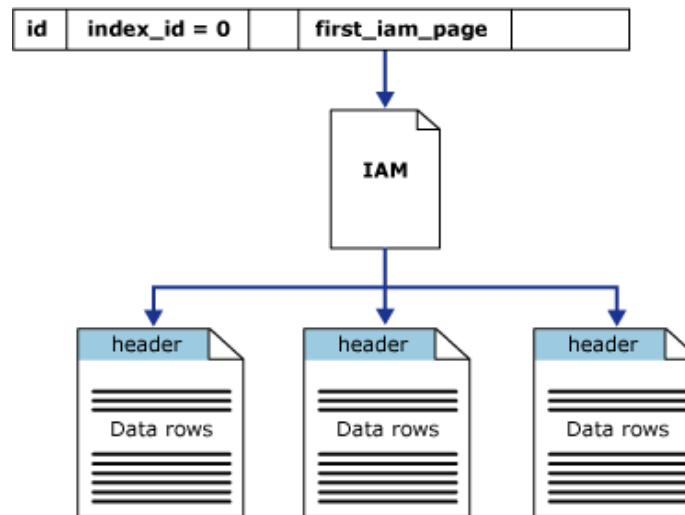
Indizes und Statistiken

Arbeitsweise der Indizes

- Indizes werden wie Datenbanken in Seiten verwaltet
- Seiten enthalten 8192 bytes
- Tabellen ohne Clustered Index = Heap
- B-Tree (balancierter Baum)
- Suche ab Wurzelknoten
 - Wie Telefonbuch

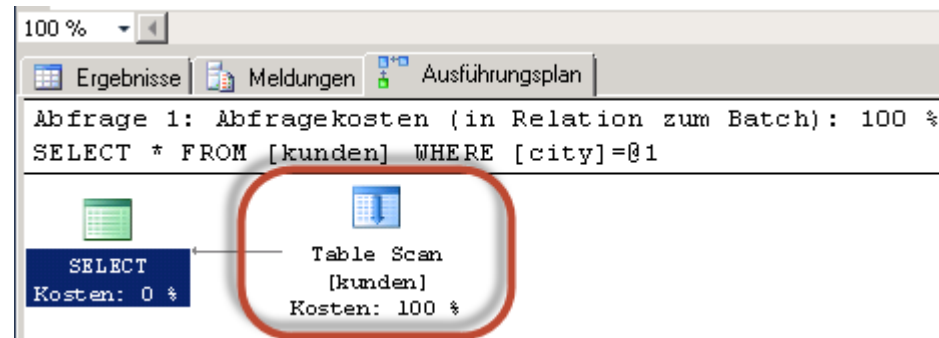
Heap

- Ein „Sau“-Haufen an Daten
- Eigtl keine Reihenfolge der Datensätze vorhersagbar
- Heap besteht aus vielen Seiten



Heap

- Suche nach bestimmten Datensätzen muss immer den kompletten Heap durchlaufen
- Suche = Durchsuchen aller Seiten
 - SET STATISTICS IO ON
- Suche = TABLE SCAN

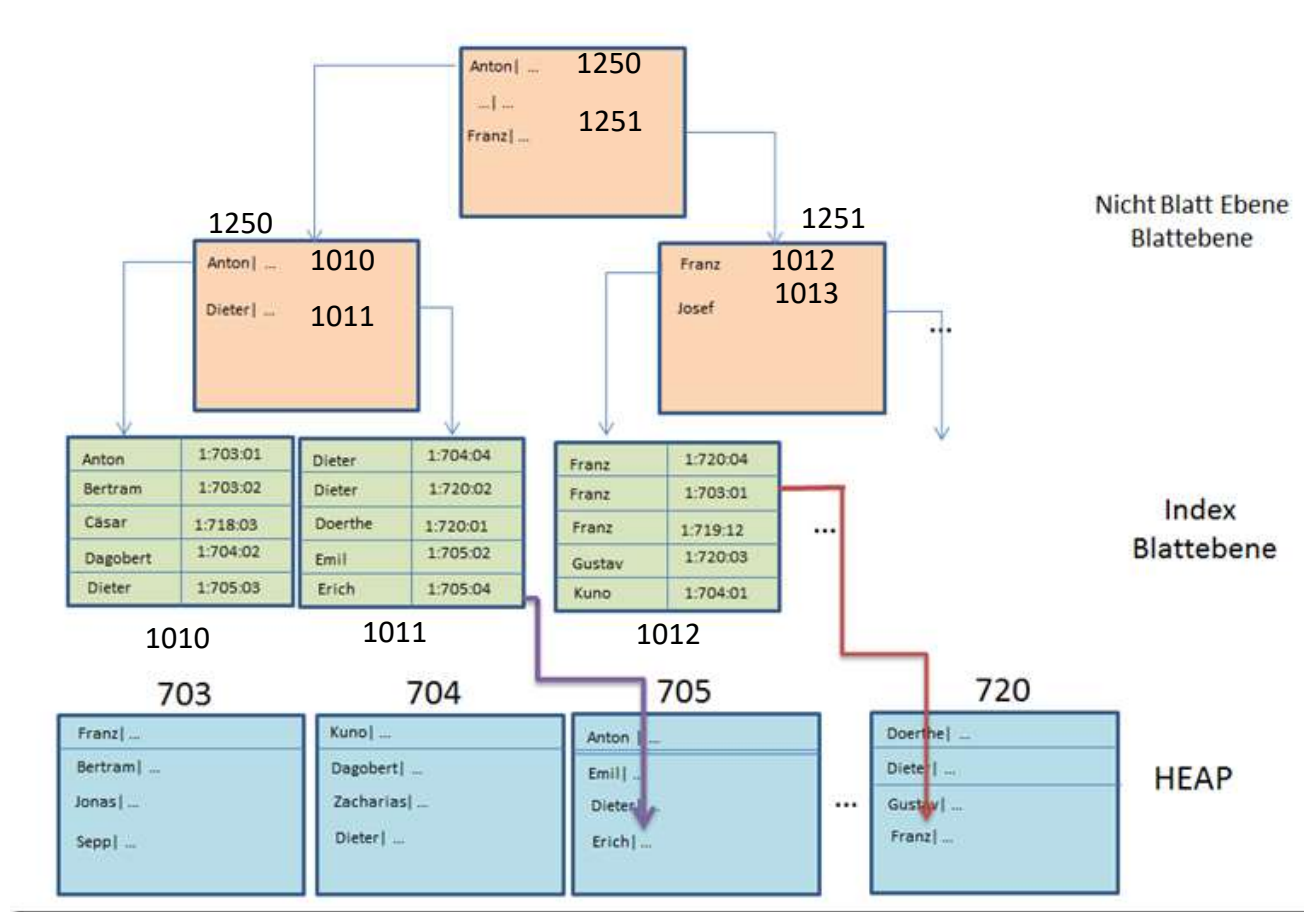


Wie funktioniert denn der Index?

- Wer das weiss, weiss auch welcher Index verwendet werden sollte
- Indizes werden ähnlich wie Telefonbücher verwaltet
 - Suche nach Tel von „Maier Hans“
Gezieltes Suchen im Telefonbuch..
...Treffer.. TelNr gefunden.
- Gezieltes Suchen im Index ist ein „Seek“



Wie funktioniert der Index?



Wie funktionieren Indizes

- Man kann auch nachschauen ;-)
 - sys.dm_db_index_physical_stats
 - DBCC IND (DB, Tabelle, 1)
 - DBCC PAGE (DB, Datei, Seite, [1,2,3])
 - DBCC TRACEON (3604)

Welche Indizes gibt es denn?

- Nicht gruppierter Index
- Gruppierter Index
- Zusammengesetzter Index (max 16 Spalten)
- Eindeutiger Index
- Index mit eingeschlossenen Spalten
- Gefilterter Index
- Partitionierter Index
- Columnstored Index
- Indizierte Sicht
- Abdeckender Index
- Realer hypothetischer Index

Welche Indizes gibt es denn?

- Spaß bei Seite!

- Nur 2!

- Bzw. 3

Nicht gruppierter Index

Gruppierter Index

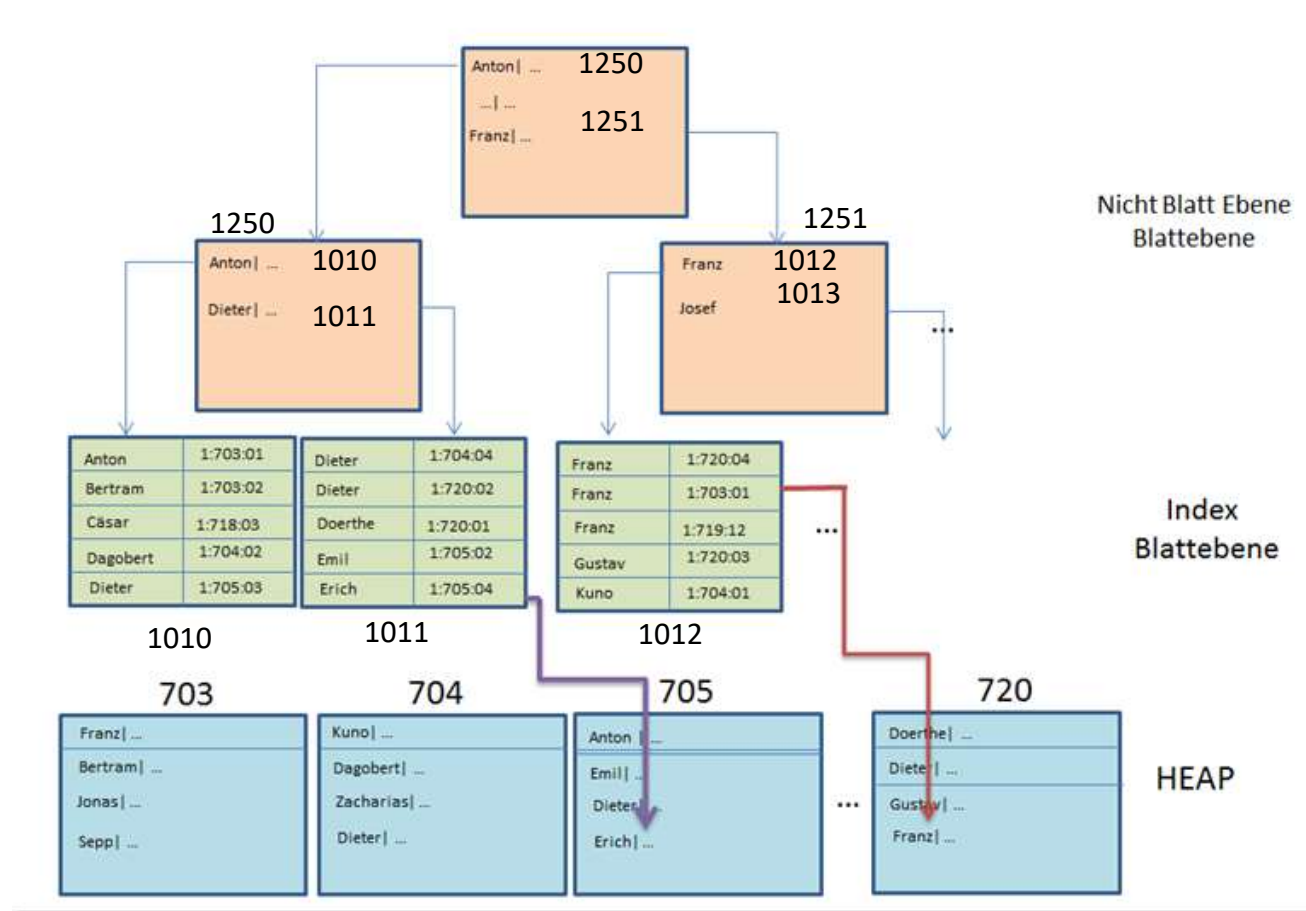
Columnstored Index

Spezialindizes: XML, Geo-Indizes

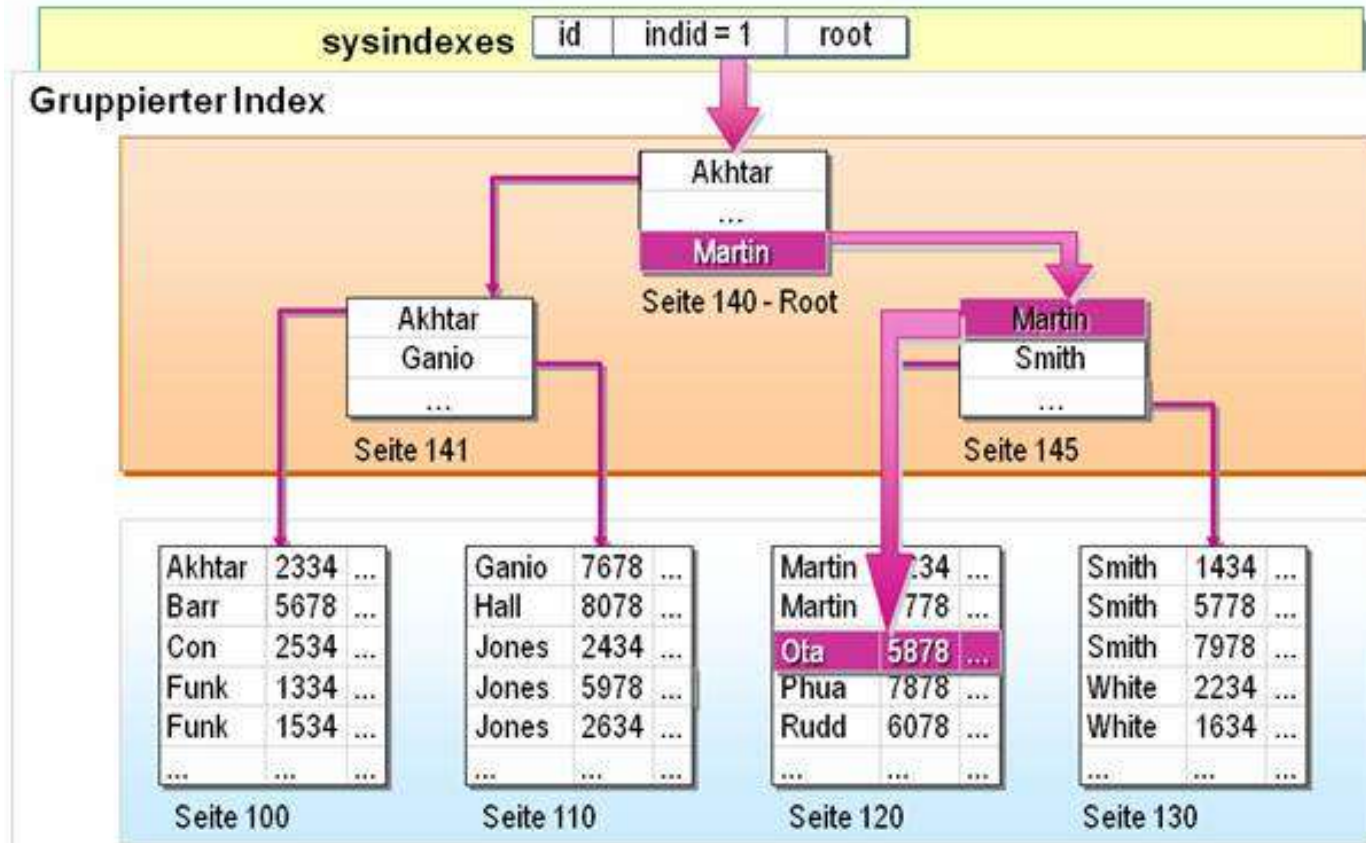
Wie funktioniert der Index?

- Nicht gruppierter Index lediglich sortierte Kopie der Indexspalten mit Zeiger auf den Originaldatensatz (1:204:02)
- Gruppierter Index ist Tabelle in physikalischer sortierter Form

Nicht gruppierter Index



Gruppiertes Index



Indizes

- Nicht gruppierte Indizes besitzen Kopien der Daten und verwenden Zeiger auf den Originaldatensatz
- Gruppierte Indizes sind die Tabellen!
...in physikalisch sortierter Form

Einsatzgebiete

- Gruppierter Index
 - Sehr gut bei Abfragen nach Bereichen und rel. Großen Ergebnismengen: < , > , between, like
Kandidaten: Bestelldatum, PLZ,..
Gibt's nur 1-mal, daher zuerst vergeben!
- Nicht gruppierter Index
 - Sehr gut bei Abfragen auf rel. eindeutige Werte bzw. geringen Ergebnismengen: =
Kandidaten: ID; Firmenname, ...
kann mehrfach verwendet werden (999-mal)
 - ➔ PK oft Gruppierter Index!! = Verschwendung

..und die anderen Indizes?

- Gefilterter Index:
 - Es müssen nicht mehr alle Datensätze in den Index mit aufgenommen werden.
- Mit Eingeschlossenen Spalten
 - Der Index kann zusätzliche Werte enthalten (→ SELECT), der Indexbaum wird dadurch nicht belastet.
- Partitionierter Index
 - Physikalische Verteilung der Indexdaten per Partitionierung

..und die anderen Indizes?

- Eindeutiger Index
 - Erzwingt eindeutige Werte.
Kandidat: Primary Key
- Zusammengesetzter Index
 - Index besteht aus mehreren Spalten. Auch im Indexbaum enthalten.
 - Kandidat: where umfaßt mehrere Spalten
 - Land , Stadt
- Abdeckender Index
 - ;-) leider nicht per „CREATE“, sondern ergibt aus der Abfrage. Bester Index! Alle Ergebnisse werden aus dem Index geliefert.
Keine Lookup Vorgänge!

..und die anderen Indizes?

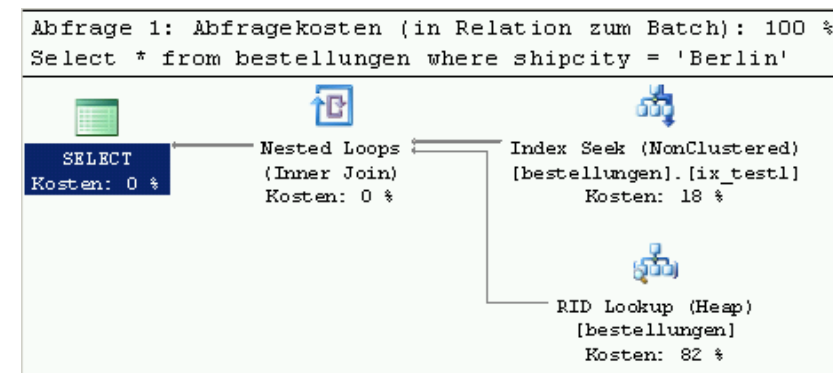
- Indizierte Sicht
 - Perfekt für Aggregate!
 - = Clustered Index (materialized View)
 - Viele Bedingungen
 - Schemabinding, big_count()
 - In Enterprise Version können Statements „überschrieben“ werden
Statt Abfrage auf Tabelle, verwendet SQL Server die Sicht
 - Aber auch Probleme: Locks

..und die anderen Indizes?

- Columnstored Index (ab SQL 2012)
 - Statt Datenätze werden Spalten in Seiten verwaltet
 - Sehr gut bei Datawarehouse Szenarien
 - Mehrfach vorkommende Werte lassen sich gut komprimieren
 - Abfragen verwenden nur noch die Seiten, in denen die entsprechenden Daten vorhanden sind

Welchen Indizes sollte man denn erstellen?

- Nur die, die man benötigt!
 - Jeder weitere Index stellt bei INS, UP ...eine Last dar
 - Keine überflüssigen Indizes (ABC, AB, A)
 - Wieviele Telefonbücher benötigen man pro Stadt?
- Die, die fehlen!
 - SQL Server merkt sich fehlende Indizes
- Nicht nur das WHERE ist entscheidend
 - Sondern auch der SELECT



Wie wirken sich Indizes auf die Leistung aus?

- Hervorragend,
 - Sofern keine Messdatenerfassung erfolgt
- Entscheidend ist die Anzahl der Indexebenen
 - Statt 100000 Seiten im Heap für 1 DS durchlaufen zu müssen, benötigt man über den Index so viele Seiten wie Ebenen vorhanden sind. (3 bis 4 Ebenen)
 - Ob 1 Mio oder 100 Mio DS, oft kaum mehr als 3 Ebenen

Worauf sollte man Indizes achten?

- Indizes müssen gewartet werden?
 - Reorg oder Neuerstellung
- Suche nach korrekten Indizes
- Suche nach doppelten, überflüssigen, fehlenden Indizes
- Gute Übersicht durch Systemsichten
 - Sys.dm_db_index_physical_stats

ColumnStore Index

- Für DEV
 - Optimal für Datenarchiv
 - Deutlich geringere Platzbedarf und daher auch deutlich geringere RAM Bedarf
 - Geringere CPU Last
 - Keine Notwendigkeit verschiedenste Indizes pro Abfrage anzulegen
 - INS UP und DEL dagegen führen zu nicht indizierten Heapstrukturen
 - Langsam und pflegebedürftig
 - Ab SQL 2014 als gr Columnstore IX (updatebar)
 - In SQL 2012 nicht updatebar
 - AB SQL 2016 SP1 in allen Versionen