

2019

Zusammenfassung DAB2

SEP FS 2019

PASCAL BRUNNER

1 Inhaltsverzeichnis

2	Grundlegendes	4
2.1	3-Ebene-Architektur	4
2.1.1	Logische Datenunabhängigkeit	5
2.1.2	Physische Datenunabhängigkeit.....	5
2.2	3-Phasen-DB-Entwurf.....	5
2.3	Trigger.....	5
2.3.1	Syntax	6
2.3.2	ECA-Prinzip	6
2.3.3	Sinnvoller Einsatz.....	7
2.3.4	Probleme	7
2.4	Stored Procedures.....	7
2.4.1	Vorteile	9
2.4.2	Nachteile.....	10
2.5	Funktionen.....	10
2.6	5-Schichten-Architektur eines RDBMS.....	11
2.7	Speicherhierarchie.....	12
2.7.1	Betriebssystem	13
2.8	Satztypen	14
2.9	Speichersystem.....	15
2.9.1	Statische Verfahren	15
2.9.1	Dynamisches Verfahren	17
2.10	RAID	17
2.10.1	Stufe 0.....	17
2.10.2	Stufe 1.....	18
2.10.3	Stufe 10 (1+0)	18
2.10.4	Stufe 5.....	18
2.10.5	Stufe 6.....	18
3	Optimierungen	19
3.1	Index erstellen	20
3.2	Zugriff auf Datensätze	20
3.2.1	Tupelzugriff mittels TID	20
3.2.1	Tupelzugriff mittels Schlüsselwert	21
3.2.1	Relationen-Scan (Full table scan)	21
3.2.1	Index-Scan	22
3.2.1	Externes Sortieren	22

3.2.1	Join.....	22
3.3	Ausgangslage der Optimierung	25
3.4	Logischen Optimierung.....	25
3.4.1	Grundprinzipien.....	25
3.4.2	Ablauf	26
3.5	Eigenschaften der Relationalen Algebra	26
3.6	Vom SQL-Statement bis zur Ausführung des Befehls.....	27
3.6.1	Übersetzung und Sichtauflösung.....	27
3.6.2	Standardisierung & Vereinfachung	28
3.6.3	Logische Optimierung.....	32
3.6.4	Physische Optimierung.....	35
3.6.5	Kostenbasierte Auswahl	35
3.6.6	Planparametrisierung	35
3.6.7	Code-Erzeugung.....	35
3.6.8	Code-Ausführung.....	35
3.7	Ausführungsplan.....	35
3.7.1	Kostenbasierte Auswahl	36
4	Transaktionen.....	37
4.1	ACID-Eigenschaften	37
4.1.1	Isolation / Nebenläufigkeit (Concurrency)	37
4.2	Isolationsebenen	39
4.3	Transaktionen in der Praxis	39
4.4	Schedules.....	40
4.4.1	Serieller Schedule	40
4.4.2	Serialisierbarer Schedule	40
4.5	Transaktionsverwaltung: Sperrverfahren.....	42
4.6	Recovery	44
4.6.1	Fehlerklassifikation.....	45
5	Glossar	46
	 Abbildung 1 Datenbankbegriffe	4
	Abbildung 2 drei Ebenen Architektur einer DB	4
	Abbildung 3 Trigger Beispiel.....	6
	Abbildung 4 Beispiel Stored Procedure.....	7
	Abbildung 5 Die 5-Schichten Architektur	11
	Abbildung 6 Verfeinerte Speicherhierarchie.....	12
	Abbildung 7 Speicherhierarchie	13
	Abbildung 8 übliche Form der Abbildung.....	13

Abbildung 9 Abbildung aus Dateisystem	13
Abbildung 10 statische Verfahren im Überblick.....	16
Abbildung 11 B Baum	17
Abbildung 12 Auswahl an Statistik-Auswertung im SQL	19
Abbildung 13 Beispiel Parse-Baum.....	27
Abbildung 14 Beispiel Nested Schachtelung	29
Abbildung 15 "IN" Umwandlung in rel. Algebra.....	30
Abbildung 16 Beispiel Entschachtelung Join	30
Abbildung 17 Typ-J Umwandlung in rel. Algebra	31

2 Grundlegendes

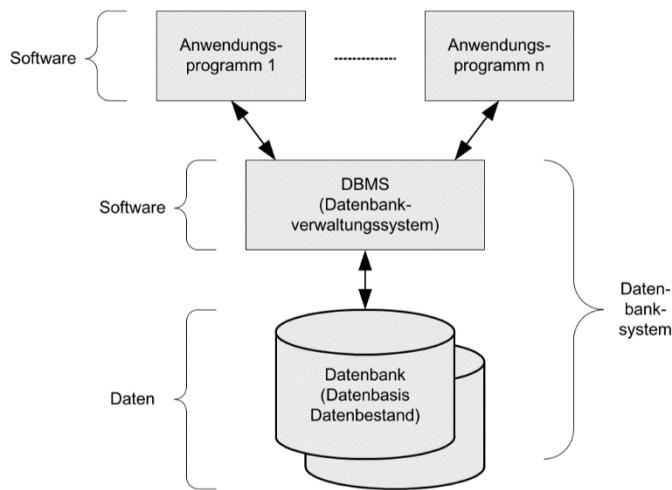


Abbildung 1 Datenbankbegriffe

2.1 3-Ebene-Architektur

Nicht zur verwechseln mit dem 3-Phasen-DB-Entwurf!

- Konzeptionell - Logische Gesamtsicht
 - Wie sind die Daten strukturiert
 - Bauen des idealtypischen Datenbanksystem (ohne Rücksicht auf Performance oder physikalischen Grenzen)
 - Extern - Sicht einer einzelnen Anwendung / Benutzergruppe
 - Wie sieht ein Benutzer / Programm die Daten
 - Wird mit den VIEWS realisiert
 - Intern - Speicherung, Datenorganisation, Zugriffsstrukturen
 - Wie sind die Daten physisch gespeichert
 - Normalisierung
 - Demormalisierung, falls es zu Performanceprobleme führt
- ➔ DBMS ermöglichen diese drei Ebenen

Das Data Dictionary beschreibt die Daten in einer Datenbank (Metadaten)

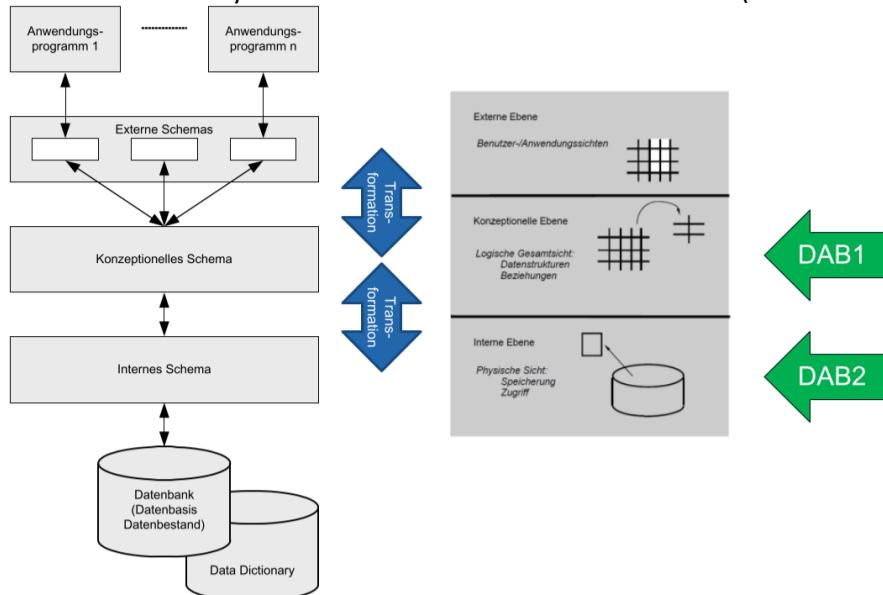


Abbildung 2 drei Ebenen Architektur einer DB

Zwischen den verschiedenen Ebenen erfolgen Transformationen. → in der Praxis wird die konzeptionelle und interne Schicht nicht getrennt.

2.1.1 Logische Datenunabhängigkeit

Änderungen (auf logischer / konzeptioneller Ebene) in einer Datenbank (bspw. Einfügen eines Attributes) hat keinen Einfluss auf die Applikationen.

2.1.2 Physische Datenunabhängigkeit

Änderungen (auf physischer, interner Ebene) in einer Datenbank (bspw. Entfernen eines Indexes) hat keinen Einfluss auf die Applikation

2.2 3-Phasen-DB-Entwurf

1. Konzeptionelles Datenmodell
 - a. Technologie unabhängige Spezifikation der Daten, die in der DB gespeichert werden sollen
 - b. ERM → korrektes Diagramm (DAB1)
2. Logisches Datenmodell
 - a. Übersetzung des konzeptionellen Schemas in Strukturen, welche mit DBMS implementiert werden
 - b. RM, SQL (DAB1)
3. Physisches Datenmodell
 - a. Umfasst alle Anpassungen
 - b. Datenverteilung, Indexierung etc. → SQL, Tools (DAB2)

Der physische Entwurf einer Datenbank ist von sehr grosser Bedeutung. Dieser ist stark abhängig von dem gewählten Produkt

- Physische Datenunabhängigkeit
- Sicherheit
- Performance

2.3 Trigger

Ein Trigger ist ein Programmcode, der vom DBMS ausgeführt wird. Auslöser des Triggers ist eine Änderung am Datenbankzustand (INSERT, UPDATE, DELETE). Dieser wird vor allem für die Realisierung von Integritätsbedingungen, Berechnungen (bspw. Nachführen von Summen) oder Protokollierung der Änderungen gut geeignet. Der Vorteil dadurch ist eine Entlastung der Applikation, was eine effizientere Ausführung ermöglicht.

Trigger können geschachtelt werden → Ein Trigger, kann ein anderer Trigger auslösen

```

CREATE TRIGGER NewPODetail ON Purchasing.PurchaseOrderDetail
AFTER INSERT
AS
IF @@ROWCOUNT = 1
BEGIN
    UPDATE Purchasing.PurchaseOrderHeader
    SET SubTotal = SubTotal + LineTotal
    FROM inserted
    WHERE PurchaseOrderHeader.PurchaseOrderID = inserted.PurchaseOrderID
END
ELSE
BEGIN
    UPDATE Purchasing.PurchaseOrderHeader SET SubTotal = SubTotal +
    (SELECT SUM(LineTotal) FROM inserted
     WHERE PurchaseOrderHeader.PurchaseOrderID = inserted.PurchaseOrderID)
    WHERE PurchaseOrderHeader.PurchaseOrderID IN (SELECT PurchaseOrderID FROM inserted)
END;

```

Der Trigger wird ausgelöst, nachdem in der Tabelle PurchaseOrderDetail (KaufAuftragsPosition) ein Datensatz oder mehrere Datensätze eingefügt wurden. Es wird das Attribut SubTotal (Gesamtbetrag des Kaufauftrags) der Tabelle PurchaseOrderHeader (KaufAuftragKopf) um den, resp. die hinzugefügten Beträgen (LineTotal) erhöht.

Abbildung 3 Trigger Beispiel

2.3.1 Syntax

Trigger werden in SQL-Syntax geschrieben und haben einen Namen

- CREATE Trigger
- DROP Trigger
- ALTER Trigger ... {DISABLE | ENABLE}

Als Auslöser werden folgende Befehlsworte verwendet

- BEFORE → vor der DML-Operation
 - o **Achtung!** SQL Server unterstützt kein BEFORE und muss durch INSTEAD OF nachgebildet werden
- AFTER → nach der DML-Operation
 - o Weitere Operationen werden ausgelöst → bspw. bestimmte andere Daten löschen
 - o Berechnungen ausführen und speichern
 - o Historisierung
- INSTEAD OF → anstelle der DML-Operation
 - o Verhindern unerlaubter DML-Operationen → vermeiden von unnötiger Datenbank-Arbeit
 - o Wenn SQL-Constraints nicht ausreichend → Höchstkomplexe Überprüfungen können ausgeführt werden
 - o Nicht änderbare Views «änderbar» machen

- for each statement wird genau einmal ausgeführt → Statement Trigger
- for each row wird für jedes modifizierte Tupel ausgeführt → Row Trigger

2.3.2 ECA-Prinzip

Event – Condition – Action → ON Ereignis IF Bedingung DO Aktion

2.3.3 Sinnvoller Einsatz

- Überprüfung / Funktion oft ausgeführt wird
- SQL-Constraints nicht ausreichend sind
- Logik von der DB ausgeführt werden soll (Codd'sche Forderung)
- Lösung stark vereinfacht wird

2.3.4 Probleme

- Fehlerbehandlung
- Testen, Debuggen
- Unübersichtlich
- Evtl. grosse Abhängigkeit
- Terminierung bei geschachtem Aufruf

2.4 Stored Procedures

Ist eine Menge von SQL-Anweisungen die unter einem gemeinsamen Namen gespeichert und als Einheit ausgeführt werden. Diese werden direkt in der DB abgelegt und können von der Applikation und/oder Benutzer aufgerufen werden.

```
CREATE PROCEDURE HumanResources.uspGetEmployees
    @LastName nvarchar(50),
    @FirstName nvarchar(50)
AS
    SELECT FirstName, LastName, Department
    FROM HumanResources.vEmployeeDepartmentHistory
    WHERE FirstName = @FirstName AND LastName = @LastName AND EndDate IS NULL;
```

- Erzeugt eine Stored Procedure (SP) Namens uspGetEmployees im Schema HumanResources.
- @LastName und @FirstName sind Input-Parameter der SP.
- Die SP sucht den entsprechenden Mitarbeiter und gibt zusätzlich dessen Departement zurück. Der Mitarbeiter wird nur zurückgegeben, falls dieser in ungekündigtem Zustand ist (EndDate IS NULL).

Abbildung 4 Beispiel Stored Procedure

```

USE DAB2Query
GO

-- Löschen, falls vorhanden
IF OBJECT_ID('Sales.GetOrders', 'P') IS NOT NULL
    DROP PROCEDURE Sales.GetOrders;
GO

-- Erstellen
CREATE PROCEDURE Sales.GetOrders
    @custid AS INTEGER,
    @orderdatefrom AS DATE = '1900-01-01', -- Defaultwert, wenn kein Parameterwert geliefert wird
    @orderdateto AS DATE = '9999-12-31' -- Defaultwert, wenn kein Parameterwert geliefert wird
AS
BEGIN
    SET NOCOUNT ON; -- Unterdrücken der Angabe der betroffenen Zeilen

    SELECT
        *
    FROM
        Sales.Orders
    WHERE
        custid = @custid
        AND orderdate >= @orderdatefrom
        AND orderdate < @orderdateto;

    RETURN;
END
GO

-- Test

EXECUTE Sales.GetOrders 37, '2014-04-01', '2014-07-01'
EXECUTE Sales.GetOrders 37
EXECUTE Sales.GetOrders

```

Fibonacci

```

USE DAB2Query
GO

-- Löschen, falls vorhanden
IF OBJECT_ID('dbo.Fibonacci', 'P') IS NOT NULL
    DROP PROCEDURE dbo.Fibonacci;
GO

-- Erstellen
CREATE PROCEDURE dbo.Fibonacci
    @n AS INTEGER
AS
BEGIN
    DECLARE @f1 INTEGER
    DECLARE @f2 INTEGER
    DECLARE @f  INTEGER
    DECLARE @nr INTEGER

    IF @n <= 2
        BEGIN
            PRINT 1
            RETURN
        END

    SET @f1 = 1
    SET @f2 = 1

    SET @nr = 0

    WHILE @nr < @n - 2
        BEGIN
            SET @f  = @f1 + @f2
            SET @f2 = @f1
            SET @f1 = @f

            SET @nr = @nr + 1
        END

    PRINT @f
END
GO

-- Test
EXECUTE dbo.Fibonacci 1
EXECUTE dbo.Fibonacci 2

```

2.4.1 Vorteile

- Große Datenmengen können direkt auf dem Server verarbeitet werden → SQL-Server ist sehr mächtig und kann sehr grosse Mengen in kurzer Zeit verarbeiten.
- Große Auswahl von Operationen für die Ausführung von Aufgaben
- Mehr Sicherheit, da es nur Ausführrecht für das Stored Procedure benötigt → Verringerung von DB-Attacken
- Parametrisierung möglich → höhere Sicherheit, verhindert SQL Injections
- Einfach änderbar, da Business Rules zentral abgelegt sind
- Wiederverwendbarkeit

2.4.2 Nachteile

- Syntax und Semantik sind nicht standardisiert → Jeder DBMS-Hersteller hat eigene Sprache
- Verwaltungsaufwand
- Fehlerbehandlung nicht immer einfach
- Meist keine höheren Strukturmöglichkeiten

2.5 Funktionen

```
USE DAB2Query
GO

-- Löschen, falls vorhanden
IF OBJECT_ID('Sales.QtyRange','IF') IS NOT NULL
    DROP FUNCTION Sales.QtyRange;
GO

-- Erstellen
CREATE FUNCTION Sales.QtyRange(@qtylow AS INTEGER, @qtyhigh AS INTEGER)
RETURNS TABLE AS RETURN
(
    SELECT
        *
    FROM
        Sales.OrderDetails
    WHERE
        qty BETWEEN @qtylow AND @qtyhigh
);
GO

-- Ausführen
SELECT
    *
FROM
    Sales.QtyRange(100,200)
```

Fibonacci-Folge Rekursiv

```
USE DAB2Query
GO

-- Löschen, falls vorhanden
IF OBJECT_ID('dbo.FibonacciRec','FN') IS NOT NULL
    DROP FUNCTION dbo.FibonacciRec;
GO

-- Erstellen
CREATE FUNCTION dbo.FibonacciRec(@n AS INTEGER)
RETURNS INTEGER
AS BEGIN
    IF @n <= 2 RETURN 1

    RETURN dbo.FibonacciRec(@n - 1) + dbo.FibonacciRec(@n - 2)
END
GO

-- Ausführen
SELECT dbo.FibonacciRec(1)
SELECT dbo.FibonacciRec(2)
SELECT dbo.FibonacciRec(3)
SELECT dbo.FibonacciRec(4)
SELECT dbo.FibonacciRec(5)
SELECT dbo.FibonacciRec(6)
SELECT dbo.FibonacciRec(7)
SELECT dbo.FibonacciRec(8)
```

2.6 5-Schichten-Architektur eines RDBMS

Die 5-schichtige Architektur hat sich in der Praxis bewährt, es beschreibt die Transformationskomponenten und Schnittstellen

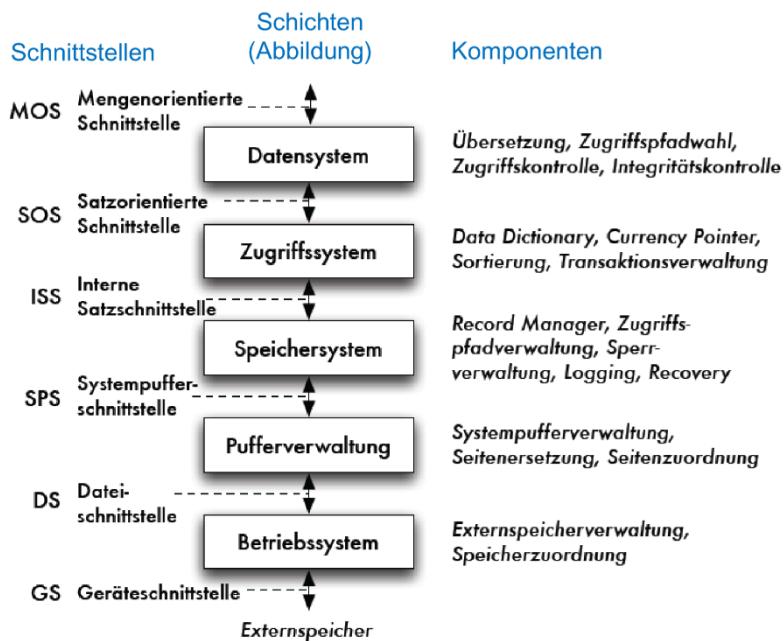


Abbildung 5 Die 5-Schichten Architektur

Die 5-Schichten-Architektur ist für die Modellierung einer Systemarchitektur eines RDBMS zuständig.

MOS: die mengenorientierte Schnittstelle wird für die Manipulation von Daten verwendet. Hierzu wird beispielsweise SQL als Sprache verwendet

Datensystem: Es wandelt die Anfragen aus dem MOS an das SOS um. Es braucht ein Parse-Baum (Übersetzer) den der SQL-Befehl verarbeitet. Des Weiteren wird noch der SQL Befehl optimiert. Der Zugriffspfad ist beispielsweise ein Index. Im besten Fall wenn man zwei Zugriffe.

Die Optimierung unterscheidet zwischen zwei grundsätzlichen Möglichkeiten:

- Logische Optimierung: Query Rewriting
- Physische Optimierung: sinnvollen Zugriffspfad

SOS: Die satzorientierte Schnittstelle stellt sicher, dass die Befehle satzweise verarbeitet werden

Zugriffssystem: Es wandelt die Anfrage von SOS an ISS um. Eine wichtige Aufgabe des Zugriffssystems ist die Abstraktion von internen Datensätzen und die Abstraktion des Zugriffs auf die Daten. Der Currency Pointer zeigt jeweils auf einen Datensatz und wird für die schrittweise Datenverarbeitung verwendet. Transaktionsverwaltung erfolgt nach dem ACID-Prinzip (Atomicity, Consistency, Isolation, Durability)

ISS: Die interne Satzschnittstelle verarbeitet die Datensätze einheitlich ohne auf die Relationentypen zu berücksichtigen (egal was für eine Tabelle, Tupel etc)

Speichersystem: Wandelt die Anfragen des ISS in Anfragen zum SPS um.

Locking: Sperrverwaltung, Logging: Buchführen, was für Änderungen wurden durchgeführt

SPS: Die Systempufferschnittstelle stellt Operationen zum Lesen und Speichern zur Verfügung

Pufferverwaltung: Wandelt die Anfragen des SPS in Anfragen zum DS um. Nimmt gespeicherte Blöcke und cached diese nach oben. Das DBMS nutzt Kenntnisse der auszuführenden Operationen um den Puffer optimaler verwenden zu können. → Relevant für die Performance.

DS: Die Datei- oder Seitenschnittstelle stellt Operationen zur Verfügung um Blöcke von Daten zu verarbeiten

Betriebssystem: Wandelt Anfragen des DS an GS weiter. Treiberprogramme zum Holen und Schreiben von Blöcken. Hier wird gespeichert und gelesen

GS: Die Geräteschnittstelle ergibt sich aus der Hardware

Der Optimizer führt Statistik über diverse Ausführungen etc., damit er bei neuen Fragen entscheiden kann welcher «Weg» der bessere ist.

Ein gutes DBMS wird immer versuchen die Performance zu verbessern

Daten und Log immer auf unterschiedlichen Disks ablegen

2.7 Speicherhierarchie

1. Prozessor mit Registern	(1 cycle)	Primärspeicher
2. Cache	(L1: 1-5 cycles, L2: 5-20 cycles, L3...)	Beispielsweise RAM / Cache → sehr schnell und sehr teuer, des Weiteren ein flüchtiger Speicher
3. Hauptspeicher (RAM)	(40 – 100 cycles)	
4. Sekundärspeicher:		Sekundärspeicher
Harddisk	(20'000'000 cycles)	Beispielsweise HD oder SSD → Faktor 10^5 – 10^6 langsamer als Primärspeicher (SSD immer noch 10^3 langsamer). Günstig bis Billig und Persistent
SSD	(200'000 cycles)	
5. Nearline-Tertiärspeicher (automatische Bereitstellung der Medien)	(Sekunden)	
6. Offline-Tertiärspeicher	(manuelle Bereitstellung der Medien, per Hand)	

Abbildung 6 Verfeinerte Speicherhierarchie

Teritärspeicher

Beispielsweise optische Speicher, Band → Langsam, billig und hohe Kapazität, ist ein Wechselmedium und nicht permanent direkt zugreifbar, dafür persistent

Ein DBMS ist hochparallel aufgebaut. Je mehr Prozessoren zur Verfügung stehen, desto besser wird die Zugriffszeit. Die Zugriffslücke wird auch durch SSD nicht bekämpft.

Für das RDBMS ist ein schneller Primärspeicher und Sekundärspeicher sehr wichtig. Cache stellt Speicherinhalte für den Prozessor zur Verfügung. Dies ist vor allem für das Lesen wichtig, jedoch beim Schreiben nicht wirklich vorteilhaft. Das Caching ist sinnvoll bei:

- Zeitliche Lokalität
 - Kurzer Zeit wiederholt auf die gleichen Daten zugegriffen werden
- Räumliche Lokalität

- Zusammen angefragte Daten sind auf dem Sekundärspeicher räumlich eng zusammen gespeichert
- Kann unter anderem mit Clustering realisiert werden
 - Fremdschlüssel eignen sich oft als Clustering

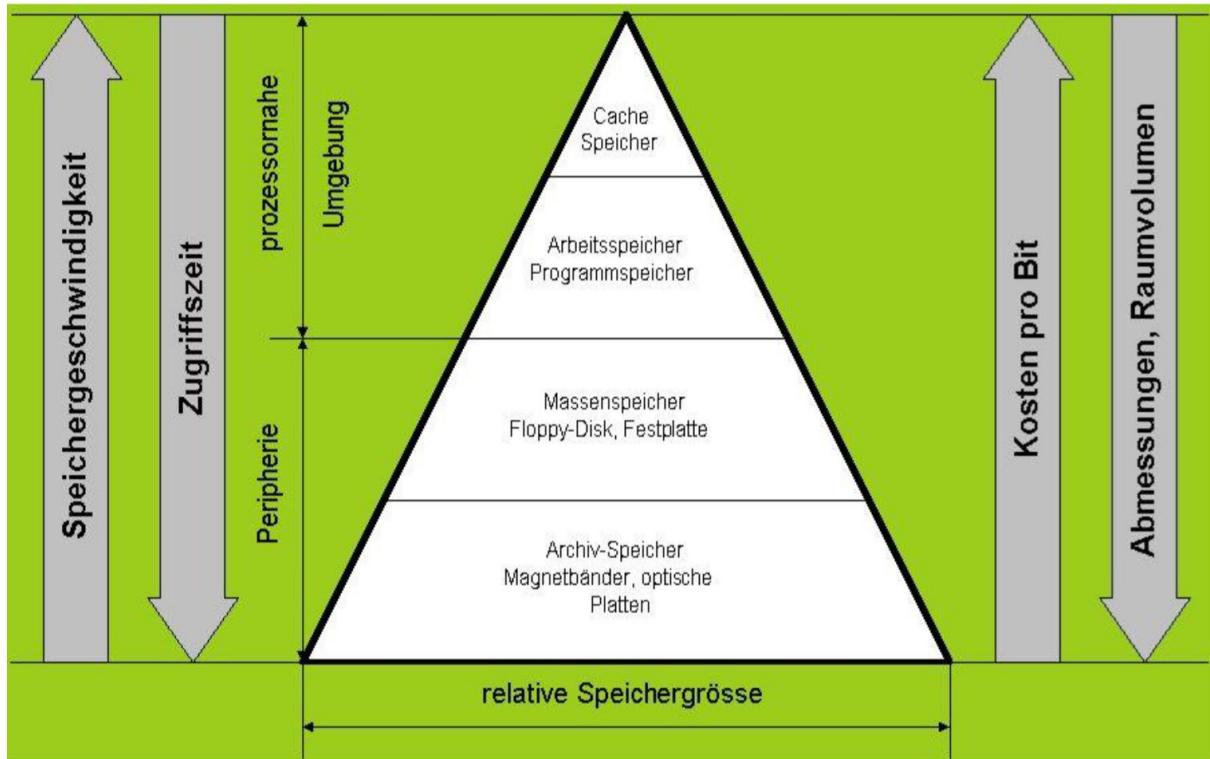


Abbildung 7 Speicherhierarchie

2.7.1 Betriebssystem

Aufgaben

- Zuordnung und Freigabe von Speicherplatz
- Holen und Speichern von Seiteninhalt
- Logische Zuordnung von Datenbereiche
- Freispeicherverwaltung → doppelt verkettete Liste von Seiten

Abbildung auf Dateisystem

Konz. Ebene	Interne Ebene	Dateisystem
Relationen	Log. Dateien	Phys. Dateien
Tupel	Datensätze	Seiten / Blöcke
Attributwerte	Felder	Bytes

Abbildung 9 Abbildung auf Dateisystem

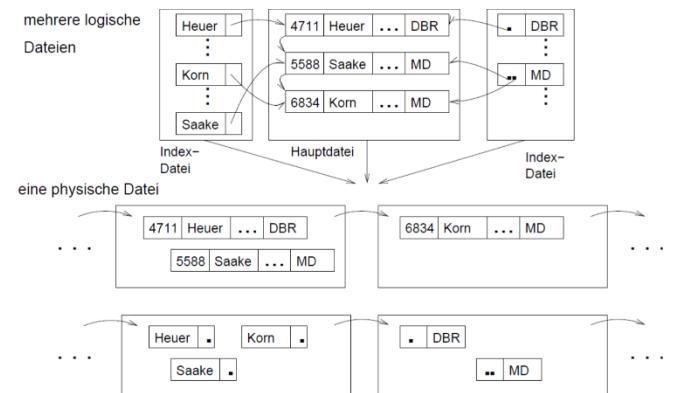


Abbildung 8 übliche Form der Abbildung

1. RDBMS jede Relation oder jeden Zugriffspfad in genau einer Betriebssystem-Datei (z.B. MySQL)
2. Eine oder mehrere Betriebssystem-Dateien. RDBMS verwaltet Relationen und Zugriffspfade selbst (z.B. SQL Server, Oracle)

Mit VARCHAR kann man zwar dynamische Größen von Feldern gewährleisten, jedoch braucht man auch mehr Rechenzeit, da man die Komprimierung rückgängig machen muss.

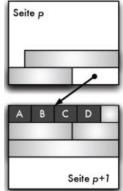
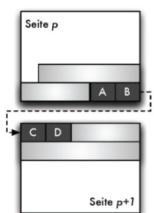
NVARCHAR speichert die Daten im UNICODE Format & VARCHAR im ASCII Code

Unterschied Block vs. Seiten

Daten werden **blockweise** gelesen, dabei werden mehrere Blöcke zu **Seiten** zusammengefasst.

Zwischen Hauptspeicher und Magnetspeicher werden immer mehrere **Blöcke** (= eine **Seite**) übertragen.

Verfahren des Blockens

- Satzorientierte Speicherung (Normalfall) → N-ary Storage Model (NSM)
 - o Nichtspannsätze: jeder Datensatz in maximal einem Block
 
 - o Spannsätze: Datensatz evtl. in mehreren Blöcken. Ist ein Datensatz zu lang für aktuellen Block, dann den noch passenden Anteil des Datensatzes in diesem Block und den Rest in einem neu anzufordernden Block speichern. → etwas unglücklich, da es zusätzlichen Overhead erzeugt.
 

- Spaltenorientierte (alt: Attributs-orientierte) Speicherung → Anwendung in Data-Warehouse

N-ary Storage Model

- Interner Aufbau der Seiten
 - o Header
 - Info über Vorgänger-/Nachgänger Seite
 - Evtl. Nummer der Seite selbst
 - Info über Typ der Seite
 - Freier Platz
 - o Datensätze
 - o Unbelegte Bytes
- Organisation
 - o Doppelt verkettete Liste
 - o Freispeicherverwaltung
 - o Satzadressen
 - Seitennummer + Offset

2.8 Satztypen

- **Fixe Satzlänge**
 - o Verwaltungsblock mit Typ und Länge eines Satzes
 - o Freiraum damit Nutzdaten immer an selber Position liegen
 - o Nutzdaten des Datensatzes
- ➔ Nachteil: Höherer Speicheraufwand

→ Bspw: SQL: char(n)

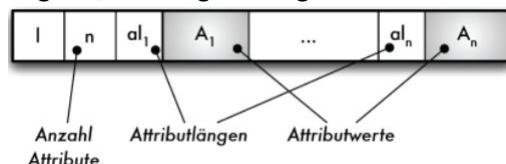
- **Variable Satzlänge**

- o Verwaltungsblock: Satzlänge l, um die Länge des Nutzdaten-Bereichs d zu kennen
 - o Nutzdaten des Datensatzes

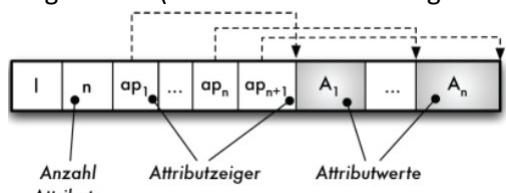
→ Nachteil: Höherer Verwaltungsaufwand beim Lesen und Schreiben, da Satzlänge immer neu ermittelt werden muss

→ Bspw: SQL: varchar(n)

- o Strategie a: Jedes Datenfeld variabler Länge A beginnt mit einem Längenzeiger al, der angeibt, wie lang das folgende Datenfeld ist



- o Strategie b: Am Beginn des Satzes wird nach dem Satz-Längenzeiger l und der Anzahl der Attribute ein Zeigerfeld ap₁, ..., ap_n für alle variabel langen Datenfelder eingerichtet (Vorteil: leichtere Navigation innerhalb des Satzes)



- **Fixiert (pinned)**

- o Addressierung über Seitennummer und Offset
 - o Problematisch bei Änderungen

- **Unfixiert (unpinned)**

- o TID-Konzept (Tupelidentifikator)
 - TID ist Datensatz-Adresse aus Seitennummer p und Index i in das Satzverzeichnis

2.9 Speichersystem

Meistens sind Daten statischer Natur und nicht dynamischer Natur

2.9.1 Statische Verfahren

Optimal nur bei bestimmter (fester) Anzahl von zu verwaltenden Datensätzen → periodische Reorganisation

Direkte Organisationsformen → keine Hilfsstruktur, keine Adressberechnung (Heap, sequenzielle Datei)

- Einfügen Heap O(1)
- Suchen Heap O(n)
- Löschen Heap O(n)
- Gutes Verfahren für bspw. Logging von Daten (man fügt sehr viel Daten ein und sucht selten etwas)
- Einfügen sequenzielle Datei O(n)
- Löschen sequenzielle Datei O(n)
- Suchen sequenzielle Datei O(n/2)
- Gutes Verfahren für Suchen (bspw. nach bestimmten Bereichen)

Statische Indexverfahren → für Primärindex (indexsequenzielle Datei)

- Kombination von sequenzieller Hauptdatei und Indexdatei
- Kann geclusterter, dünnbesetzter Index sein
- Mind. Zweistufiger Baum
- Probleme:
- Stufenzahl ist fix
- Index- und Hauptdatei-Seiten schrumpfen langsam
- Viele Mutationen führen zu unausgeglichenen Seiten in der Hauptdatei

Indexiert-nichtsequentieller Zugriffspfad

- Hauptdatei kann unsortiert sein
- Dichtbesetzt
- Nicht geclustered

Statisches Hashing

- Basis-Hashfunktion: $h(k) = k \bmod m$
- Anforderungen an h
 - o Effizient berechenbar
 - o Kollisionen minimieren
 - o Vernüftiger Bildbereich
- Abbilden der Primärschlüsselattributwerten auf sogenannte Bucket-Adresse

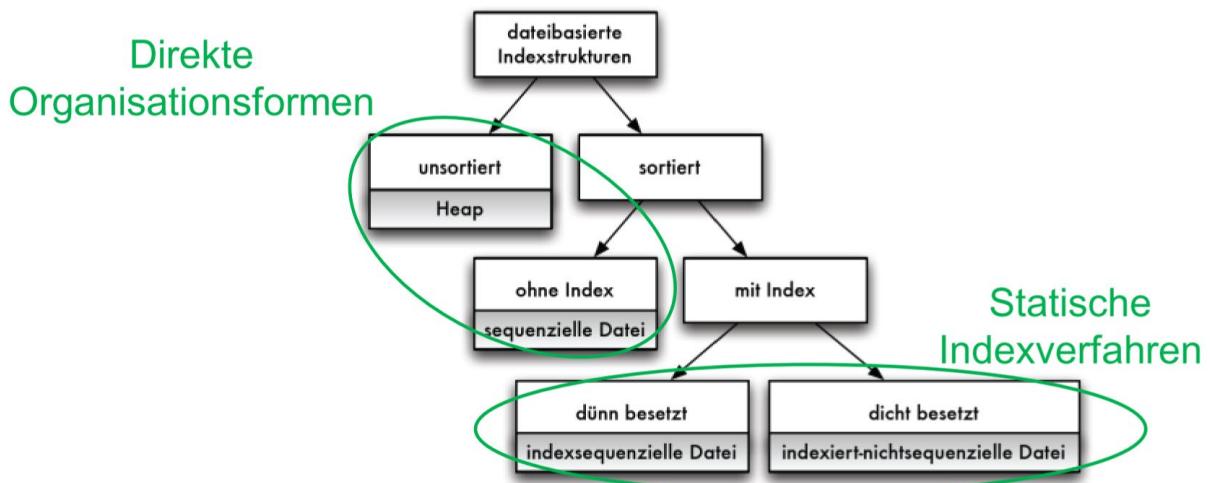


Abbildung 10 statische Verfahren im Überblick

Dünn = Index referenziert auf Seite

Dicht = Index referenziert auf Tupel

Jeder dünn-besetzte Index muss auch geclustert sein, jedoch nicht umgekehrt

Sekundärindexe können nur dicht-besetzt und nicht-geclustert sein

Abdeckender Index = ist ein Index mit mehreren Kriterien

2.9.1 Dynamisches Verfahren

Praktisch alles ist in Form eines B-Baumes realisiert

B Baum

- Dynamisches Verfahren: Anpassung der Anzahl Stufen
- Allgegenwärtig im Datenbankbereich (nicht nur RDBMS)
- Man macht einen Knoten (i) genau so gross wie eine Seite → mehrere Dateien pro Knoten
- Aufwand beim B-Baum → $O(\log_2(n))$
- Aufwand beim B+-Baumes → $O(\log(n))$ → bereits mit zwei Ebenen kann man sehr sehr viele Daten abspeichern
- Wir verwenden nie (!) B-Bäume, sondern immer B+-Bäume!
- Sind balanciert / ausgeglichen
- Vollständig balanciert

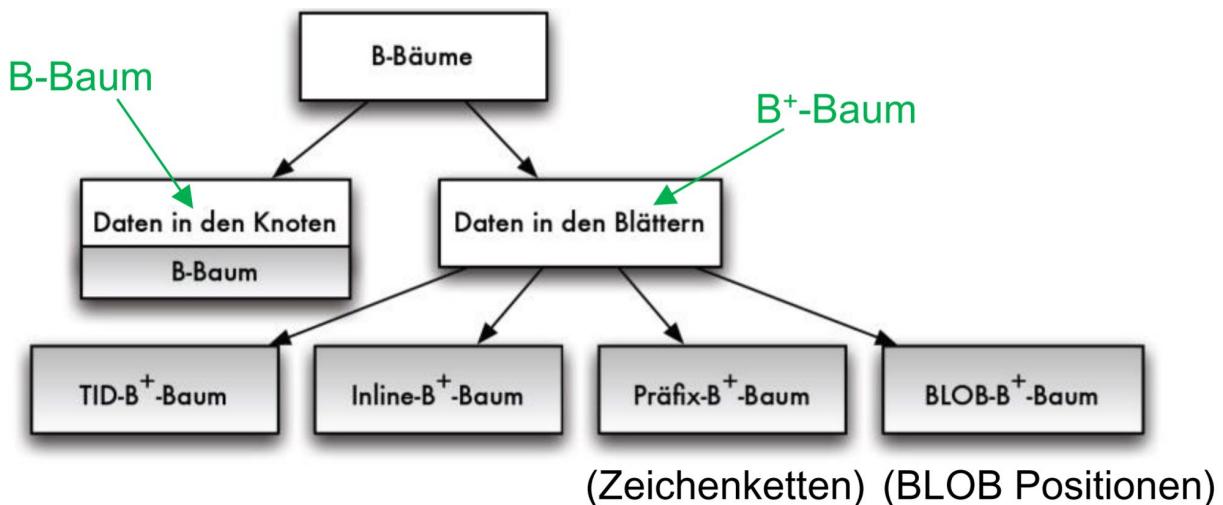


Abbildung 11 B Baum

2.10 RAID

RAID-Stufe	Anzahl der Festplatten $n \geq$	Netto-Kapazität (Anzahl Festplatten)	Ausfall von n Festplatten möglich
0	≥ 2	n	0
1	≥ 2	Grösse der kleinsten Festplatte	$n - 1$
5	≥ 3	$n - 1$	1
6	≥ 4	$n - 2$	2

2.10.1 Stufe 0

keine zusätzliche Sicherheit, man verteilt seine Daten auf zusätzliche Festplatten → Performanceverbesserung

2.10.2 Stufe 1

mega redundant, man spiegelt die Daten auf eine andere Festplatte ab → hohe Ausfallsicherheit

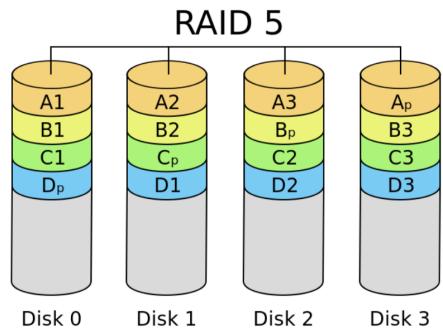
2.10.3 Stufe 10 (1+0)

Man verteilt die Daten auf mehrere Festplatten und Spiegelung der Daten

2.10.4 Stufe 5

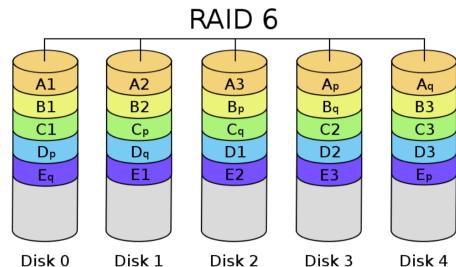
Parity-Konzept → für n Festplatten hat man noch eine zusätzliche Festplatte n+1. Durch die Verknüpfung von XOR aller Festplatte, kann man im Falle eines Ausfallen aufgrund des Kennens des Resultats wiederherstellen. Maximal ein Laufwerk darf ausfallen.

Mindestens ≥ 3 Festplatten

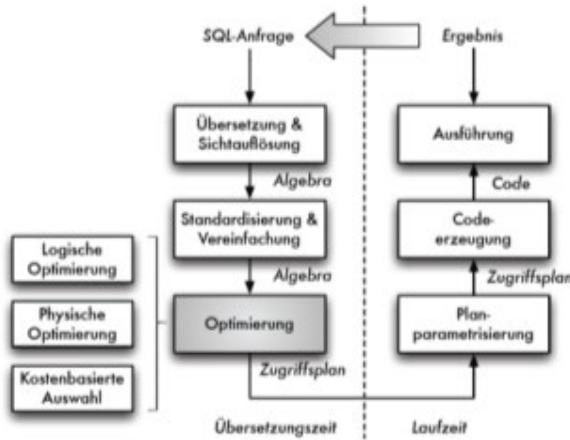


2.10.5 Stufe 6

Hat man doppelte Paritäts-Informationen → Ausfall von maximal zwei Laufwerken möglich



3 Optimierungen



Die Basis Algorithmen des Optimizers sitzen im Zugriffssystem und der Optimizer selbst eher «unten». Für einzelne Operationen (bspw. Join) werden dabei mehrere **Basisalgorithmen** implementiert, um für unterschiedliche Situationen jeweils den geeigneten Algorithmus auswählen zu können. Die Auswahl des geeigneten Basisalgorithmus und der besten Reihenfolge übernimmt der Optimizer (Komponente des Zugriffssystems).

Die vom DBMS ausgewählten Basisalgorithmen und Reihenfolge wird als **Ausführungsplan** (Execution Plan) bezeichnet. → Auswahl aufgrund einer Aufwand-Kostenabschätzung.

Aufwandsabschätzung Basisalgorithmen

- $|r|$ Anzahl Tupel in der Relation r
- b_r Anzahl von Blöcken die Tupel der Relation r beinhalten
- mem Puffergrösse in Anzahl der Blöcke
- $val_{A,r}$ Anzahl verschiedener Werte für das Attribut A in der Relation r («Cardinality»)
- $lev_{I(R(A))}$ Anzahl der Indexebenen eines B^+ -Baums für den Index $I(R(A))$ für das Attribut A des Relationenschemas R
- $bsize$ Blockgrösse
- $size_r$ (mittlere) Grösse von Tupeln aus r
- f_r Blockungsfaktor – wieviel Tupel aus r können in einem Block gespeichert werden: $f_r = bsize / size_r$

Abbildung 12 Auswahl an Statistik-Auswertung im SQL

- Interessant ist vor allem wie viele Seiten gelesen werden müssen. (**dominierender Kostenfaktor**)
- Die Statistik muss immer wieder aktualisiert werden, da diese bereits nach einigen Sekunden nicht mehr aktuell sind.
- Die Temp-DB wird automatisch von SQL erzeugt und wird vor allem für Zwischenablage bei grösseren Berechnungen verwendet
- JOIN ist sehr aufwändig
 - o Nested-Loop-Join: Doppelte Schleife (pro Element die ganze Tabelle durchgehen)
 - Verbesserungen:

- Geblocktes Lesen
- Indexe werden als B*-Bäume realisiert
- Für Zwischenergebnisse sind Seitenzugriffe notwendig (bspw. Sortierung)
- Zwischenrelationen werden im Hintergrund (nicht Hauptspeicher) angelegt

3.1 Index erstellen

```
USE AdventureWorks2012;
GO
-- Create a new table with three columns.
CREATE TABLE dbo.TestTable
(
    TestCol1 int NOT NULL,
    TestCol2 nchar(10) NULL,
    TestCol3 nvarchar(50) NULL
);
GO
-- Create a clustered index called IX_TestTable_TestCol1
-- on the dbo.TestTable table using the TestCol1 column.
CREATE CLUSTERED INDEX IX_TestTable_TestCol1
    ON dbo.TestTable (TestCol1);
GO
```

CREATE INDEX Syntax

Creates an index on a table. Duplicate values are allowed:

```
CREATE INDEX index_name
ON table_name (column1, column2, ...);
```

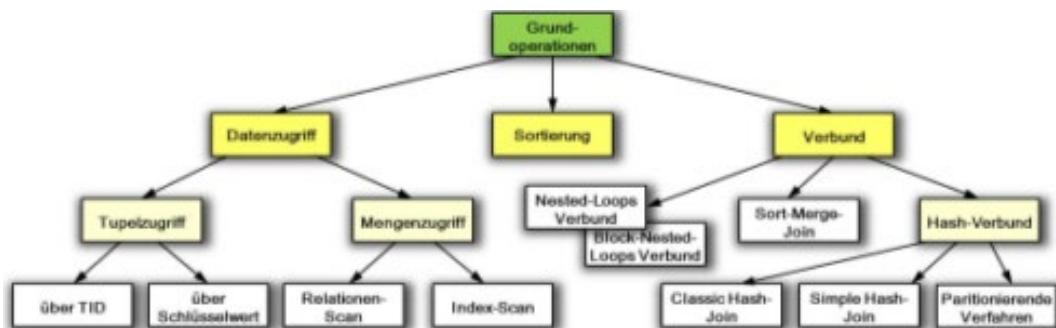
CREATE UNIQUE INDEX Syntax

Creates a unique index on a table. Duplicate values are not allowed:

```
CREATE UNIQUE INDEX index_name
ON table_name (column1, column2, ...);
```

```
DROP INDEX table_name.index_name;
```

3.2 Zugriff auf Datensätze



1. Beim Datenzugriff werden Daten gelesen / verändert. Beim Tupelzugriff wird maximal ein Datensatz anhand eines Wertes (TID oder Schlüsselwert) gesucht, während beim Mengenzugriff (SCAN, Durchlaufen einer Relation), alle oder eine Menge von Datensätzen gelesen wird.
2. Bei der Sortierung werden grosse Datenmengen ausserhalb des Hauptspeichers sortiert
3. Bei den Verbundoperationen werden mehrere Tabellen miteinander verbunden (Join)

3.2.1 Tupelzugriff mittels TID

Zugriff auf Tupel mittels RelationenID (Identifikator der Relation) und TID. Tupel wird im Tupelpuffer abgelegt.

Bemerkung: TID besteht aus der Seitennummer und einem Offset, wodurch direkt auf den im Hintergrundspeicher liegenden Datensatz zugegriffen werden kann

Operation: fetch-tuple(RelationenID; TID) → Tupelpuffer

Aufwand: O(1)

3.2.1 Tupelzugriff mittels Schlüsselwert

Es gibt zwei Arten von Indexen: 1. **Primärindex** (bspw. Index(KUNDE(KNr)): liefert max. ein Tupel; 2. **Sekundärindex** (bspw. Index(BESTELLUNG(KNr)): liefert evtl. mehrere Tupel. Der Zugriff erfolgt mittels IndexID

Bemerkung: Max. ein Datensatz gelesen, i.d.R. wird über Primärschlüssel zugegriffen. Sekundärindexe können nicht mittels fetch-TID abgehandelt werden -> Resultat ist eine TID-Liste

Operation: fetch-TID(IndexID; Attributwert) → TID

Aufwand: O(lev_{i(R(A))}) + O(1) entspricht → O(log_m(|r|))

Kann wie folgt umgesetzt werden:

```

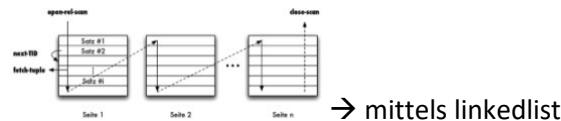
1 currTID ← fetch-TID(KUNDE-KNR-Index, 4711);
2 currRec ← fetch-tuple(KUNDE-RelationID, currTID);
3 put(currRec);
  
```

Anzeige des Ergebnisses

3.2.1 Relationen-Scan (Full table scan)

Scan: Durchlaufen von Tupeln einer Relation (Selektion / Projektion)

Relationen-Scan: Durchlaufen aller Tupel einer Relation in beliebiger Reihenfolge



Operation:

- Open-rel-scan(RelationenId) → ScanID
- Next-TID(ScanId) → liefert nächsten TID
- End-of-scan(scanId) → true, falls kein TID mehr abzuarbeiten
- Close-scan(ScanId) → schliesst scan

Aufwand: O(b_r) → b_r = Anzahl Blöcke die Tupel enthalten

SELECT * FROM Kunde **WHERE** Nachname **BETWEEN** 'Heuer' **AND** 'Jagellowsk'

Umgesetzt mittels Operationen:

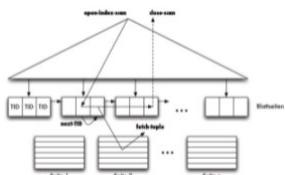
```

1 currScanID ← open-rel-scan(KUNDE-RelationID);
2 currTID ← next-TID(currScanID);
3 while ¬ end-of-scan(currScanID) do
4   currRec ← fetch-tuple(KUNDE-RelationID, currTID);
5   if currRec.Nachname ≥ 'Heuer' and currRec.Nachname ≤ 'Jagellowsk'
6     then
7       | put (currRec);
8     end
9   currTID ← next-TID(currScanID);
10 end
11 close-scan(currScanID);
  
```

```
DECLARE kunden_Cursor CURSOR FOR SELECT Name, Vorname FROM Kunde WHERE KNr > 100
OPEN kunden_Cursor
FETCH NEXT FROM kunden_Cursor INTO @name, @vorname
WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT @name + ' ' + @vorname
    ...
    FETCH NEXT FROM vendor_cursor INTO @name, @vorname
END
CLOSE kunden_Cursor;
DEALLOCATE kunden_Cursor;
```

3.2.1 Index-Scan

Nutzt Index zum Auslesen der Tupel in Sortierreihenfolge (Ausnahme: Hash-Index)



Zusätzliche Operation: open-index-scan(IndexID, Min, Max) → ScanID

Aufwand für bestimmten Wert mit Attribut A: $O(\text{lev}_{I(R(A))} + (|r| / \text{val}_{A,r})) \rightarrow \text{lev} = \text{Anz. Indexebenen}; \text{val} = \text{Anzahl versch. Werte für das Attribut A in der Relation r}$

```
SELECT * FROM Kunde WHERE Nachname BETWEEN 'Heuer' AND 'Jagellowsk'
```

Umgesetzt mittels Operationen:

```

1 currScanID ← open-index-scan(KUNDE-Nachname-IndexID,
  'Heuer','Jagellowsh');
2 currTID ← next-TID(currScanID);
3 while → end-of-scan(currScanID) do
4   currRec ← fetch-tuple(KUNDE-RelationID, currTID);
5   put(currRec);
6   currTID ← next-TID(currScanID);
7 end
8 close-scan(currScanID);

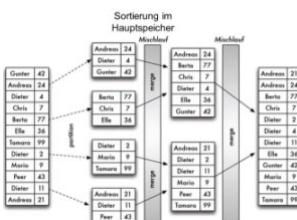
```

3.2.1 Externes Sortieren

Problem: Datenmengen die nicht in Hauptspeicher passen → Mergesort-Verfahren

Ablauf:

- Relationen werden in zwei gleich grosse Teile geteilt, die im Hauptspeicher sortiert werden können und werden mittels bspw. Heapsort sortiert
 - N Mischläufe mit Merge-Operationen, die zwei Zwischenergebnisse merged, so lange wieder nur noch eine Relation verbleibt



3.2.1 Join

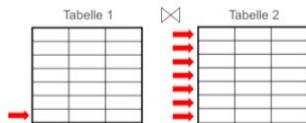
Einige Operation die Tupel aus unterschiedlichen Tabellen verknüpft. SQL Server bietet fünf Varianten von Basisalgorithmen (Loop bzw. Nested-Loop-Join, Merge-Join und drei Varianten des

Hash-Joins). Der Optimizer versucht aufgrund von statistischen Auswertungen den optimalen Join zu wählen.

Verfahren	Vorbedingung	Komplexität	Anwendbarkeit
Nested Loops Block Nested Loops	keine Relationen in Blöcken organisiert	$O(r \cdot s)$ $O(b_r \cdot b_s)$	alle Verbunde
Merge	Relationen sortiert nach Verbundattributen	$O(b_r + b_s)$	Equi-Join
Sort-Merge	keine	$O(b_r \log_{\text{mem}} b_r + b_s \log_{\text{mem}} b_s)$	Equi-Join
Classic Hash	keine	$O(b_r + k \cdot b_s)$	Equi-Join

Nested-Loop-Join

1. Satz der 1. Tabelle mit allen Sätzen aus Tabelle 2 usw.



Eine Verbesserung lässt sich mit geblocktem Lesen der Sätze beider Tabellen (statt über Tupel über Blöcke) realisieren

Kosten:

- ohne Indexunterstützung (geblocktes Lesen):

$$\text{cost}_{\text{LOOP}} = b_r + \left\lceil \frac{b_r}{\text{mem} - 1} \right\rceil \cdot b_s$$

Möglichst viele Blöcke von r im Speicher
- mit Primärindexunterstützung in s:

$$\text{cost}_{\text{LOOP}} = b_r + |r| \cdot (\text{lev}_{I(s)} + 1)$$
- mit Sekundärindexunterstützung in s:

$$\text{cost}_{\text{LOOP}} = b_r + |r| \cdot (\text{lev}_{I(s)} + \frac{|s|}{\text{val}_{B,s}})$$

□

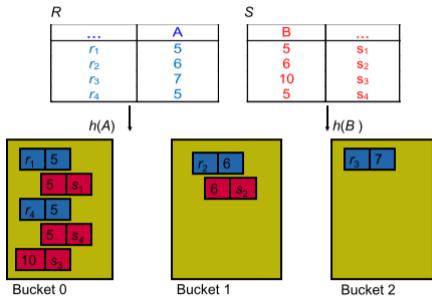
Sort-Merge-Join

Beide Tabelle werden sortiert



Hash-Join

Die Grundidee des Hash-Joins ist, dass durch gleiche Hashfunktion auf beiden Relationen die Verbundkandidaten in denselben Blöcken (im Hauptspeicher) zu finden sind. Dies setzt voraus, dass zumindest die Hashtabelle in den Hauptspeicher passt.



Vorgehen

1. Build-Phase: kleinere Relation durch Hashfunktion h in Buckets einordnen
 2. Probe-Phase: Durch Anwendung von h auf grössere Relation Buckets mit Verbundkandidaten identifizieren
 3. Verbundbedingung pro Bucket prüfen und Resultate ausgeben
- Dies ist nur passend für equi-join oder natural Join

Man unterscheidet von drei unterschiedlichen Hash-Join-Varianten

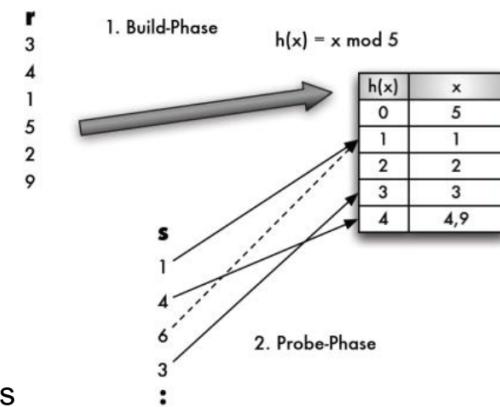
Classic Hash-Join (Überlauferkennung)

Hashtabelle für r wird so gross angelegt, dass diese im Hauptspeicher Platz hat, dadurch muss r allenfalls partitioniert werden und s für jede Partition vollständig gelesen werden

Aufwand: $O(b_r + p * b_s)$ → b_r = Anz. Blöcke; p = Anz. Partitionen von r = Anz. Scans über s

Ablauf:

1. Build-Phase (Scan über r)
Tupel mittels Hashfunktion $h(r.A)$ in Hashtabelle H einordnen.
2. Probe-Phase (Scan über s)
Wenn H voll oder vollständig gelesen:
Scan über s und mit $h(s.B)$ Verbundpartner suchen.
3. Falls Scan über r nicht abgeschlossen:
 H neu aufbauen und erneuten Scan über s



Simple Hash-Join (Begrenzung des Bildbereichs)

Ist eine Verbesserung des Classic Hash-Join. Der Hash-Adressbereich wird so gewählt, dass alle Tupel aus r in der Hashtabelle H Platz finden. Dafür muss h in mehrere Bildbereiche unterteilt werden.

- Beim Classic Join muss die grössere Tabelle s mehrfach gelesen werden. Um die Anzahl der Lese-Operationen zu verringern soll sowohl r wie auch s partitioniert werden – sie wird in mehrere Bildbereiche unterteilt (bspw. 0-10, 11-20 und 21-26). Die Verarbeitung erfolgt jetzt entlang den Partitionen der Hashtabelle.

Ablauf:

1. Build-Phase:
Scan über r: Tupel von r mittels Hashfunktion $h(r.A)$ in aktuellen, begrenzten Bildbereich der Hashtabelle H ablegen. Tupel ausserhalb Bildbereich in Überlaufbereich ablegen.
2. Probe-Phase:
Scan über s: Überprüfung mit $h(s.B)$ ob das Tupel im Bildbereich und Verbundpartner suchen. Tupel ausserhalb Bildbereich in Überlaufbereich ablegen.
3. Hashtabelle h löschen, Bildbereich ändern und Verfahren mit Tupeln aus Überlaufbereich wiederholen.

Hash-partitionierte Verfahren (Partitionierung des Bildbereichs)

Verbesserung des Simple Hash-Joins. Mehrfaches lesen wird vermieden, indem zu Beginn die Relationen mit einer Hashfunktion h auf dem Externen Speicher partitioniert werden.

- Ab dem zweiten Schritt wird beim simple Hash-Join die Tupel wiederholt gelesen und geschrieben, je mehr Operationen für die Hashtabelle, desto mehr Lese- und Schreiboperationen

Ablauf:

1. **Partitionierung:**
Die Relationen r und s werden mittels h_1 auf dem Externspeicher in Partitionen r_i und s_i eingeordnet.
2. Für jede Partition i werden folgende Schritte ausgeführt:
 1. Build-Phase:
Scan über r_i : Tupel von r_i mittels Hashfunktion $h_2(r.A)$ in Hashtabelle H ablegen.
 2. Probe-Phase:
Scan über s_i : Mit $h_2(s.B)$ Verbundpartner suchen.

3.3 Ausgangslage der Optimierung

Eingaben sind mengenorientierte Anfragen, die in die satzorientierte Schnittstelle des Zugriffssystems umgesetzt werden müssen

- Sehr hohes Abstraktionsniveau der mengenorientierten Schnittstelle (SQL)
- SQL ist **deklarativ**, nicht-prozedural → es wird spezifiziert **was** man finden möchte, aber **nicht wie**
- Das wie bestimmt sich aus der Abbildung der mengenorientierten Operatoren auf Schnittstellen-Operatoren der internen Ebene
- Zu einem was, kann es zahlreiche wie geben: Effiziente Anfrageauswertung durch Anfrageoptimierung erwünscht
- Im Allgemeinen wird aber nicht die optimale Auswertungsstrategie gesucht bzw. gefunden, sondern eine genügend gute

3.4 Logischen Optimierung

3.4.1 Grundprinzipien

Das Ziel ist eine möglichst schnelle Anfragebearbeitung. Hierfür können folgende Überlegungen hilfreich sein.

- Möglichst **kleine Zwischenergebnisse** → **Selektionen** so früh wie möglich ausführen
- Basisoperationen wenn möglich zusammenfassen (Selektion / Projektion) und ohne Zwischenspeicherung von Zwischenergebnissen als ein Berechnungsschritt realisieren
- Nur Berechnung ausführen, die auch einen Beitrag zum Gesamtergebnis liefern → **Redundante Operationen** oder nachweisbar leere Zwischenrelationen aus Berechnungen entfernen
- Zusammenfassen gleicher Teilausdrücke (Wiederverwendung von Zwischenergebnissen)
- Eigenschaften der relationalen Algebra ausnutzen

3.4.2 Ablauf

1. SQL in Parse-Baum umwandeln

2. Semantische Analyse

3. Parse-Baum in Operator-Baum umwandeln (Bag-Algebra)

4. Auflösen von Sichten (Views)

5. Standardisierung von Selektions- und Verbundbedingungen (konjunktive Normalform – KNF)

6. Vereinfachung / Transformation (inkl. Entschachtelung)

→ Anschliessend folgt die physische Optimierung. Es werden ausführbare Pläne (Execution Plan) erstellt (mit versch. Basisoperationen) und nach einer Kostenabschätzung der optimale Plan gewählt

3.5 Eigenschaften der Relationalen Algebra

Äquivalenzbeziehungen:

- $p_1 \wedge p_2 \Leftrightarrow p_2 \wedge p_1$
- $p_1 \vee p_2 \Leftrightarrow p_2 \vee p_1$
- $p_1 \wedge (p_2 \wedge p_3) \Leftrightarrow (p_1 \wedge p_2) \wedge p_3$
- $p_1 \vee (p_2 \vee p_3) \Leftrightarrow (p_1 \vee p_2) \vee p_3$
- $p_1 \vee (p_2 \wedge p_3) \Leftrightarrow (p_1 \vee p_2) \wedge (p_1 \vee p_3)$
- $\neg(\neg p_1) \Leftrightarrow p_1$
- ...

Verbund, Vereinigung, Durchschnitt und Produkt sind kommutativ:

- $R1 \bowtie R2 = R2 \bowtie R1$
- $R1 \cup R2 = R2 \cup R1$
- $R1 \cap R2 = R2 \cap R1$
- $R1 \times R2 = R2 \times R1$

Selektionen sind untereinander vertauschbar:

- $\sigma_P(\sigma_Q(R)) = \sigma_Q(\sigma_P(R))$

Idempotenz:

- $A \vee A \Leftrightarrow A$
- $A \wedge A \Leftrightarrow A$
- $A \vee \neg A \Leftrightarrow \text{true}$
- $A \wedge \neg A \Leftrightarrow \text{false}$

Verbund, Vereinigung, Durchschnitt und Produkt sind assoziativ:

- $R1 \bowtie (R2 \bowtie R3) = (R1 \bowtie R2) \bowtie R3$
- $R1 \cup (R2 \cup R3) = (R1 \cup R2) \cup R3$
- $R1 \cap (R2 \cap R3) = (R1 \cap R2) \cap R3$
- $R1 \times (R2 \times R3) = (R1 \times R2) \times R3$

Konjunktionen in einer Selektionsbedingung können in mehrere Selektionen aufgebrochen werden:

- $\sigma_{p1} \wedge \sigma_{p2} \wedge \dots \wedge \sigma_{pn}(R) = \sigma_{p1}(\sigma_{p2}(\dots(\sigma_{pn}(R))\dots))$

Eine Projektion kann mit der Vereinigung vertauscht werden:

- $\pi_L(R1 \cup R2) = \pi_L(R1) \cup \pi_L(R2)$

Eine Disjunktion kann in eine Konjunktion umgewandelt werden:

- $\neg(P1 \vee P2) = \neg P1 \wedge \neg P2$
- $\neg(P1 \wedge P2) = \neg P1 \vee \neg P2$

3.6 Vom SQL-Statement bis zur Ausführung des Befehls

1. Übersetzung und Sichtauflösung
 - a. SQL in Parse-Baum übersetzen (Syntax Analyse)
 - b. Semantische Analyse des Eingabetextes (Existieren Tabellen, Attribute, Rechte, ...)
 - c. Parse-Baum in relationale Bag-Algebra, respektive Operator-Baum übersetzen
 - d. Sichten (Views) auflösen (Sichtexpansion), Views durch SQL-Statements ersetzen
2. Standardisierung und Vereinfachung
 - a. Standardisierung (logische Ausdrücke und rel. Algebra)
 - b. Vereinfachung (Entfernung von redundanten Teilen, logische Ausdrücke, etc.)
 - c. Entschachtelung von Subqueries (spezieller Teil der Vereinfachung)
3. Optimierung
 - a. Logische Optimierung: Transformation mittels algebraischen, äquivalenzerhaltenden Regeln
 - b. Physische Optimierung: Entwicklung von potentiellen Ausführungsplanvarianten (Wahl der Basisalgorithmen), inkl. Berücksichtigung von parallelen Operationen
 - c. Kostenbasierte Bewertung der Ausführungsplanvarianten und Auswahl des Ausführungsplanes aufgrund statistischer Daten
4. Planparametrisierung: Vorcompilierte SQL-Anweisungen (Stored Procedure, Embedded SQL) müssen nicht mehr übersetzt und optimiert werden, für diese werden die gewählten Ausführungspläne gespeichert, welche als «Methoden» mit Parametern aufgerufen werden können. Die Phase der Übersetzung kann in diesen Fällen entfallen. Im SQL-Server werden auch dynamische Abfragen in einem Pool zwischengespeichert, so dass wiederkehrende SQL-Anfragen nicht erneut übersetzt und optimiert werden müssen.
5. Codeerzeugung: Die Ausführungspläne können in ausführbaren Code umgewandelt werden. Werden die Ausführungspläne durch einen Interpreter durchgeführt, entfällt dieser Schritt

3.6.1 Übersetzung und Sichtauflösung

- A) Die Zeichenfolge einer gegebenen Anfrage muss in eine Datenstruktur mit SQL-Syntax-Konstrukten, den Parse-Baum, überführt werden. Dies ist Aufgabe des Parsers.

```
SELECT k.KNr, k.Nachname
FROM KUNDE k, BESTELLUNG b
WHERE k.KNr = b.KNr AND b.Datum > '22-11-04'
```

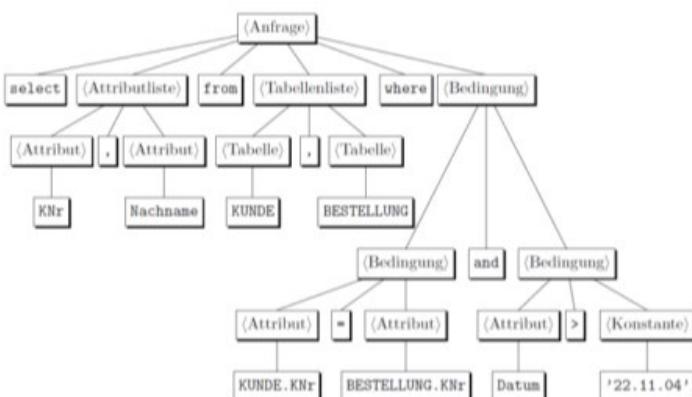


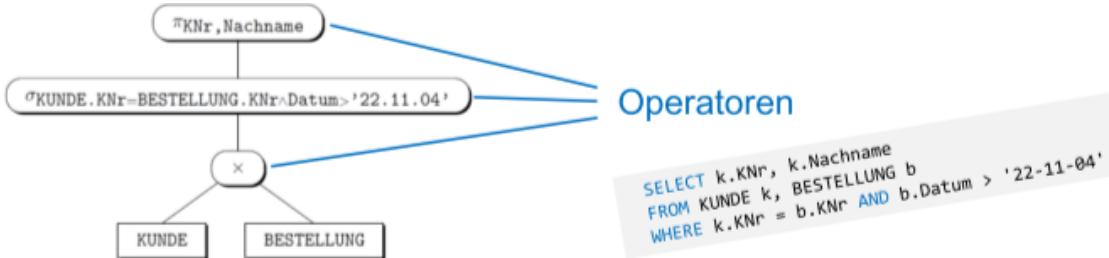
Abbildung 13 Beispiel Parse-Baum

- B) Der Parser überprüft die lexikalische (korrekte Angabe der Symbole [Schlüsselwörter, Variablenname etc.]) und syntaktische (korrekte Reihenfolge / Grammatik) Korrektheit des

SQL-Ausdrucks. Des Weiteren wird die semantische Korrektheit (bspw. existiert das Attribut in der Tabelle), sowie die Zugriffsberechtigung überprüft).

C) Übersetzung in rel. Bag-Algebra (Operator-Baum)

- Vom Parse-Baum zum Operator-Baum → Transformationsregeln
 - Relationen der Tabellenliste hinter FROM untereinander durch Kreuzprodukte verknüpfen
 - Bedingung im WHERE-Teil als Selektion übernehmen
 - Spaltenliste hinter SELECT als abschliessende Projektion
 - Berücksichtigung von SQL-Konstrukten wie ORDER BY; GROUP BY etc.
 - Später während Vereinfachung: Auflösen von Unteranfragen (Subqueries)



$\Pi_{KNr, Nachname}(\sigma_{KUNDE.KNr = BESTELLUNG.KNr \wedge Datum > '22.11.04'}(KUNDE \times BESTELLUNG))$

- D) Sichten über Sichten sollten generell vorsichtig eingesetzt werden, da hier die Komplexität sehr rasch wächst und die Wartung dadurch schwerfällt.

3.6.2 Standardisierung & Vereinfachung

- Logische Bedingungen werden in Normalform (konjunktive oder disjunktive) gebracht
- Vereinfachen: Durch Ausnutzung mathematischer Gesetzmäßigkeiten werden die Ausdrücke vereinfacht bspw. ist $B \vee \neg B$ immer true
- Entschachtelung: Bei der Entschachtelung werden Subqueries im WHERE-Teil (bspw. WHERE FK IN (SELECT PK FROM A)) aufgelöst, so dass diese als «normale» Join-Operationen ausgeführt werden können
 - Anstelle einer geschachtelten Iteration kann eine effizientere Verbundoperation ausgeführt werden
 - Gemeinsame Teilabfragen lassen sich besser erkennen

`SELECT * FROM KUNDE
WHERE KNr IN (SELECT KNr FROM BESTELLUNG)`

wird zu...



Jedoch sind die beiden Statements nicht identisch:

- Im zweiten Statement werden die Attribute beider Tabellen zurückgegeben, kann durch Angabe der Attributliste gelöst werden

- Falls für einen Kunden mehrere Bestellungen vorliegen, werden in der zweiten Abfrage auch mehrere Tupel pro Kunde im Resultat erscheinen. → braucht noch ein Group- oder Distinct Operation

Es gibt vier Typen von Schachtelung:

- **Typ-A-Schachtelung** (A = Aggregation)

- o Innerer Block Q enthält kein Verbundprädikat, das die äussere Relation referenziert
- o Q berechnet Aggregat

```
SELECT BestNr
FROM BESTELLUNG
WHERE ProdNr = (SELECT MAX(ProdNr)
                 FROM PRODUKT
                 WHERE Preis < 15)
```

- o Die Ausführung erfolgt durch «Innere Anfrage + Aggregat berechnen» dann das Ergebnis in äussere Anfrage einsetzen und diese berechnen

- **Typ-N-Schachtelung** (N = Nested)

- o Innerer Block Q enthält kein Verbundprädikat, das die äussere Relation referenziert
- o Q berechnet **kein** Aggregat

```
SELECT BestNr
FROM BESTELLUNG
WHERE ProdNr IN (SELECT ProdNr
                  FROM PRODUKT
                  WHERE Preis < 15)
```

- o Die Ausführung erfolgt entweder gemäss Typ-A oder durch Entschachtelung

```
SELECT BestNr
FROM BESTELLUNG
WHERE ProdNr IN (SELECT ProdNr
                  FROM PRODUKT
                  WHERE Preis < 15)
```

```
SELECT BestNr
FROM BESTELLUNG B JOIN PRODUKT P ON B.ProdNr = P.ProdNr
AND P.Preis < 15
```

Abbildung 14 Beispiel Nested Schachtelung

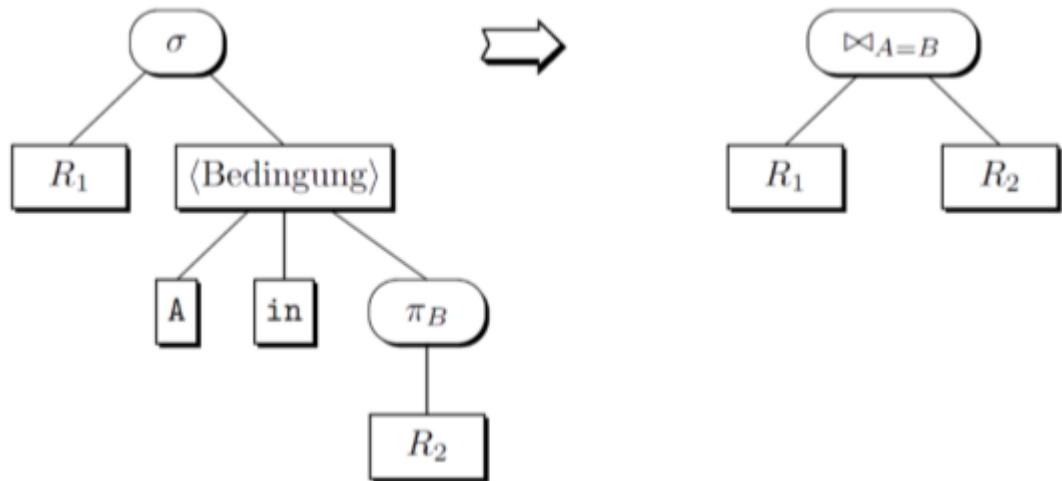


Abbildung 15 "IN" Umwandlung in rel. Algebra

- **Typ-J-Schachtelung** ($J = \text{Join}$)
 - o Innerer Block Q referenziert im Verbundprädikat die Relation des äusseren Blocks ('korrelierte Unterabfrage')
 - o Q liefert kein Aggregatwert

```

SELECT BestNr
FROM BESTELLUNG B
WHERE B.ProdNr IN (
    SELECT ProdNr
    FROM LIEFERUNG L
    WHERE L.LiefNr = B.LiefNr AND
        L.Datum = current_date)
  
```

Ohne Entschachtelung würde jeder Datensatz des äusseren, den inneren Block aufrufen.

```

SELECT BestNr
FROM BESTELLUNG B
WHERE B.ProdNr IN (
    SELECT ProdNr
    FROM LIEFERUNG L
    WHERE L.LiefNr = B.LiefNr AND
        L.Datum = current_date)

SELECT BestNr
FROM BESTELLUNG B JOIN LIEFERUNG L
ON B.ProdNr = L.ProdNr
    AND L.LiefNr = B.LiefNr AND L.Datum = current date
  
```

Abbildung 16 Beispiel Entschachtelung Join

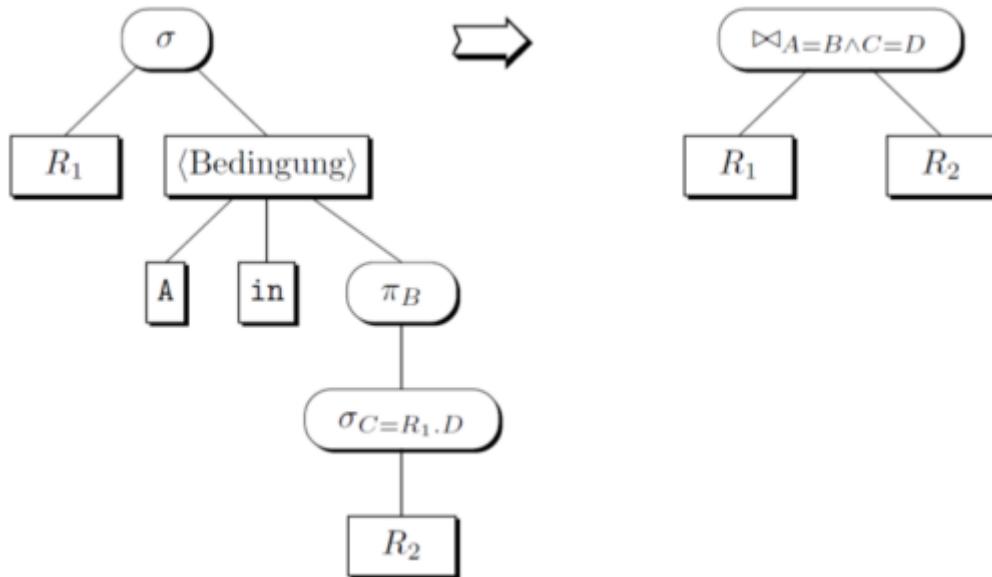


Abbildung 17 Typ-J Umwandlung in rel. Algebra

- Typ-JA-Schachtelung (JA = Join-Aggregation)
 - o Innerer Block Q referenziert im Verbundprädikat die Relation des äusseren Blocks ('korrelierte Unterabfrage')
 - o Q liefert Aggregatwert

```

SELECT BestNr
FROM BESTELLUNG B
WHERE B.BestNr IN (SELECT MAX(LiefNr)
                    FROM LIEFERUNG L
                    WHERE L.ProdNr = B.ProdNr AND
                          L.Versandart = 'Express')
  
```

- o Die Entschachtelung erfolgt in zwei Schritten. Der Block Q wird zunächst als Tabelle berechnet

```

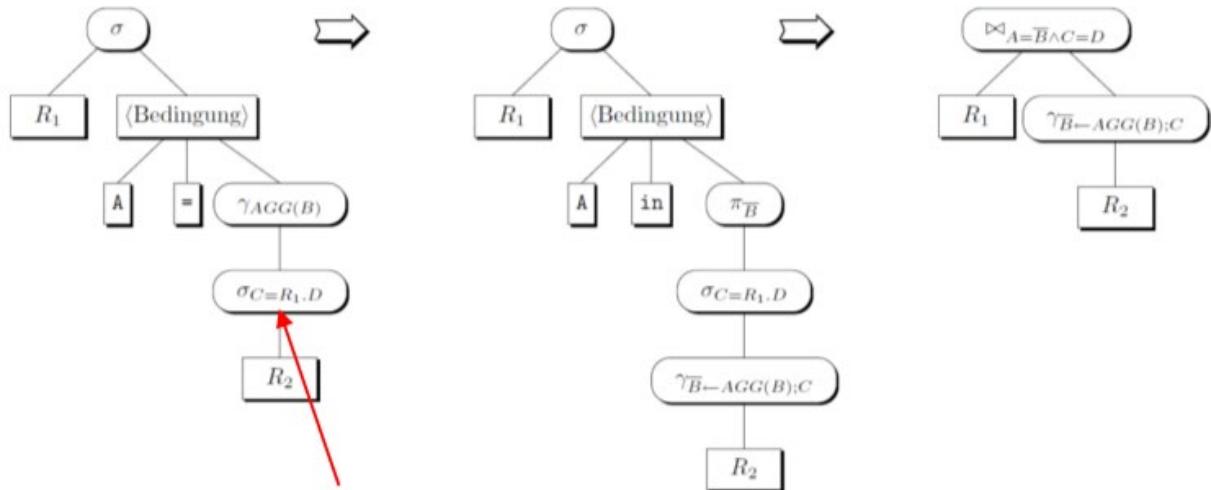
SELECT MAX(LiefNr) AS MAX_LiefNr, ProdNr FROM LIEFERUNG
WHERE Versandart = 'Express' GROUP BY ProdNr
  
```

```

SELECT BestNr
FROM BESTELLUNG B
WHERE B.BestNr IN (SELECT MAX(LiefNr)
                    FROM LIEFERUNG L
                    WHERE B.ProdNr = L.ProdNr AND
                        L.Versandart = 'Express')

SELECT B.BestNr
FROM BESTELLUNG B JOIN (SELECT MAX(LiefNr) AS MAX_LiefNr, ProdNr
                           FROM LIEFERUNG
                           WHERE Versandart = 'Express'
                           GROUP BY ProdNr) L
ON B.BestNr = L.MAX_LiefNr AND B.ProdNr = L.ProdNr

```



3.6.3 Logische Optimierung

Detaillierte Ausführung siehe Kapitel oben

1. Aufbrechen von Konjunktionen im Selektionsprädikat:
 $\sigma_{c_1 \wedge c_2 \wedge \dots \wedge c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\dots(\sigma_{c_n}(R))\dots))$
2. σ ist kommutativ:
 $\sigma_{c_1}(\sigma_{c_2}((R))) \equiv \sigma_{c_2}(\sigma_{c_1}((R)))$
3. π -Kaskaden: Falls $A_1 \subseteq A_2 \subseteq \dots \subseteq A_n$, dann gilt:
 $\pi_{A_1}(\pi_{A_2}(\dots(\pi_{A_n}(R))\dots)) \equiv \pi_{A_1}(R)$

4. Vertauschen von σ und π : Falls die Selektion sich nur auf die Attribute A_1, \dots, A_n der Projektionsliste bezieht, können die beiden Operationen vertauscht werden:

$$\pi_{A_1, \dots, A_n}(\sigma_c(R)) \equiv \sigma_c(\pi_{A_1, \dots, A_n}(R))$$

5. \times, \bowtie_j, \cup und \cap sind kommutativ. Wenn also Φ eine dieser Operationen bezeichnet, so gilt:

$$R \Phi S \equiv S \Phi R$$

6. Vertauschen von σ mit \bowtie_j (statt Join natürlich auch mit Cross-Join):

Falls das Selektionsprädikat c nur auf Attribute der Relation R zugreift, kann man die beiden Operationen vertauschen:

$$\sigma_c(R \bowtie_j S) \equiv \sigma_c(R) \bowtie_j S$$

Falls c_1 sich nur auf Attribute aus R und c_2 sich nur auf Attribute aus S bezieht, gilt folgende Äquivalenz:

$$\sigma_{c_1 \wedge c_2}(R \bowtie_j S) \equiv \sigma_{c_1}(R) \bowtie_j (\sigma_{c_2}(S))$$

7. Vertauschung von π mit \bowtie_j

Die Projektionsliste L sei: $L = \{A_1, \dots, A_n, B_1, \dots, B_m\}$, wobei A_i Attribute aus R und B_j Attribute aus S seien. Falls sich das Joinprädikat j nur auf Attribute aus L bezieht, gilt folgende Umformung:

$$\pi_L(R \bowtie_j S) \equiv (\pi_{A_1, \dots, A_n}(R)) \bowtie_j (\pi_{B_1, \dots, B_m}(S))$$

Falls das Joinprädikat sich auf weitere Attribute, sagen wir A'_1, \dots, A'_p aus R und B'_1, \dots, B'_q aus S bezieht, müssen diese für die Join-Operation erhalten bleiben und können erst danach herausprojiziert werden:

$$\pi_L(R \bowtie_j S) \equiv \pi_L(\pi_{A_1, \dots, A_n, A'_1, \dots, A'_p}(R) \bowtie_j \pi_{B_1, \dots, B_m, B'_1, \dots, B'_q}(S))$$

8. Die Operationen x , \bowtie , \cup , \cap sind jeweils (einzelnen betrachtet) assoziativ. Wenn also Φ eine dieser Operationen bezeichnet, so gilt:

$$(R \Phi S) \Phi T \equiv R \Phi (S \Phi T)$$

9. Die Operation σ ist distributiv mit \cup , \cap , $-$. Falls Φ eine dieser Operationen bezeichnet, gilt:

$$\sigma_c(R \Phi S) \equiv (\sigma_c(R)) \Phi (\sigma_c(S))$$

10. Die Operation π ist distributiv mit \cup :

$$\pi_A(R \cup S) \equiv (\pi_A(R)) \cup (\pi_A(S))$$

11. Die Join- und/oder Selektionsprädikate können mittels de Morgan's Regeln umgeformt werden:

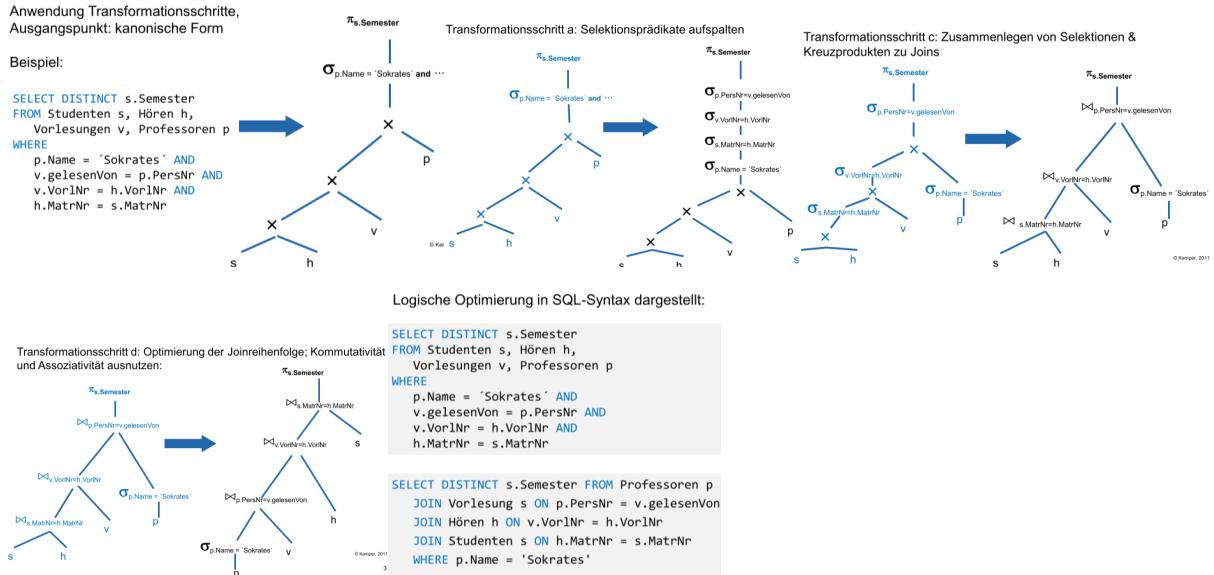
$$\neg(c_1 \wedge c_2) \equiv (\neg c_1) \vee (\neg c_2) \text{ bzw. } \neg(c_1 \vee c_2) \equiv (\neg c_1) \wedge (\neg c_2)$$

12. Ein kartesisches Produkt, das von einer Selektions-Operation gefolgt wird, deren Selektionsprädikat Attribute aus beiden Operanden des kartesischen Produktes enthält, kann in eine Join-Operation umgeformt werden.

Sei c eine Bedingung der Form $A \theta B$, mit A ein Attribut von R und B ein Attribut aus S : $\sigma_c(R \times S) \equiv R \bowtie_c S$

Die Transformationsschritte durch die Anwendung der 12-Regeln

- Mittels Regel 1 werden konjunktive Selektionsprädikate in Kaskaden von σ -Operationen zerlegt.
- Mittels Regeln 2, 4, 6, und 9 werden Selektionsoperationen soweit „nach unten“ propagiert wie möglich.
- Forme eine x -Operation, die von einer σ -Operation gefolgt wird, wenn möglich in eine \bowtie -Operation um (Regel 12).
- Mittels Regel 5 und 8 werden die Blattknoten so vertauscht, dass derjenige, der das kleinste Zwischenergebnis liefert, zuerst ausgewertet wird.
- Mittels Regeln 3, 4, 7, und 10 werden Projektionen soweit wie möglich nach unten propagiert.
- Versuche Operationsfolgen zusammenzufassen, wenn sie in einem „Durchlauf“ ausführbar sind (z.B. Anwendung von Regel 1, Regel 3, aber auch Zusammenfassung aufeinanderfolgender Selektionen und Projektionen zu einer „Filter“-Operation).



3.6.4 Physische Optimierung

→ Optimalen, physischen Plan mit Basisalgorithmen erstellen

Das Ergebnis der logischen Optimierung muss in einen ausführbaren und optimalen, physischen Plan umgeformt werden. Die algeb. Operatoren müssen dabei durch Basisalgorithmen ersetzt werden. Diese Umformung ist nicht eindeutig und es müssen verschiedenen Pläne generiert werden um anschliessend eine Kostenabschätzung durchzuführen.

3.6.5 Kostenbasierte Auswahl

Statistikinformationen für die Auswahl eines konkreten internen Planes nutzen. **Achtung:** logische, physische und kostenbasierte Auswahl werden nicht sequentiell ausgeführt, sondern bieten versch. Möglichkeiten aus denen die RDBMS auswählen kann.

3.6.6 Planparametrisierung

Bei vorkompilierten Anfragen (Embedded SQL, Stored Procedure): Ersetzen der Platzhalter durch Werte

3.6.7 Code-Erzeugung

Umwandlung des Zugriffsplans in ausführbaren Code, oder Ausführung mittels Interpreter

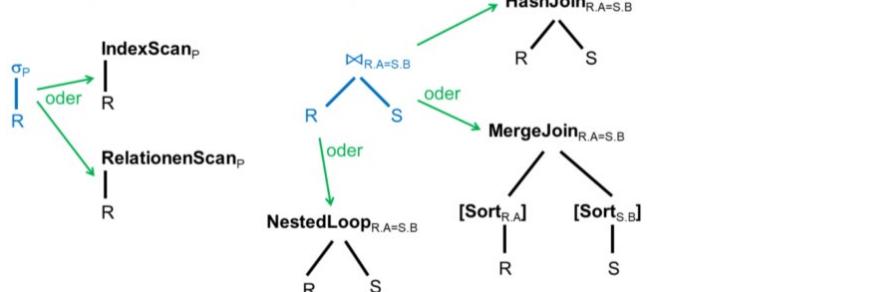
3.6.8 Code-Ausführung

Ausführung des Codes auf dem DB-Server

3.7 Ausführungsplan

Um den besten Plan zu finden, braucht es eine Aufwand-/Kostenanschätzung

Umsetzung eines QEP hat mehrere Möglichkeiten:

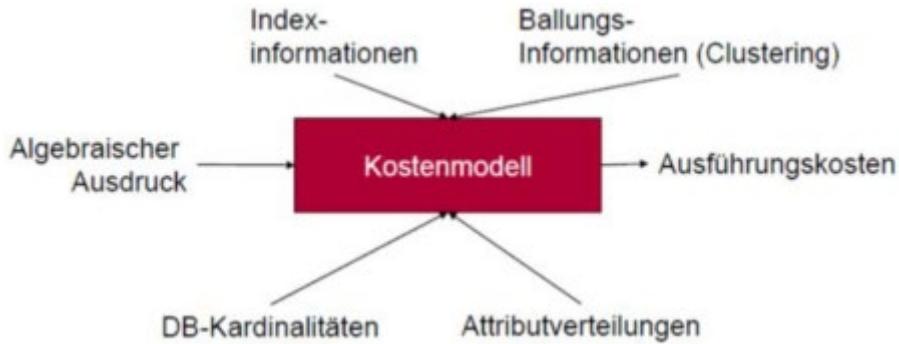


3.7.1 Kostenbasierte Auswahl

Ziel: den billigsten Plan auszuführen

Idee: Generierung aller denkbaren Pläne und den billigsten ausführen. Die Bewertung erfolgt anhand versch. Daten (Häufigkeiten, Verteilung, verfügbare Ressourcen etc.)

Mit dem Kostenmodell rechnet das RDBMS einige Pläne und die Ausführungskosten aus und wählt dann den billigsten.



Zur Kostenberechnung werden statistische Daten gesammelt.

Zu jeder Basisrelation:

- Anzahl der Tupel ($|r|$ = **Kardinalität der Relation R**), Tupelgröße, ...

Zu (jedem) Attribut:

- Min / Max, $val_{A,r}$: Anzahl verschiedener Werte für das Attribut A in der Relation r (= **Kardinalität des Attributes**); bei ungleichmäßiger Verteilung der Werte → **Werte-verteilung** nötig (**Histogramm**), ...

Zum System:

- Speichergröße, Bandbreite, I/O Zeiten, CPU Zeiten, ...

4 Transaktionen

Gilt nur für die relationale Datenbank

4.1 ACID-Eigenschaften

- Atomarität / **Atomicity**
 - o Man will das eine Transaktion entweder ganz oder gar nicht ausgeführt wird
- Konsistenz / **Consistency**
 - o Bedingungen: Constraints, Checks, NOT NULL, Datentypen
 - o SQL-Server überprüft die Bedingungen → Liefert eine Fehlermeldung, falls dies verletzt wird
- Isolation / Nebenläufigkeit (Concurrency)
 - o Gleichzeitiger Zugriff mehrerer Benutzer ermöglichen
 - o Auch bei nur einem CPU kann Pseudo-Concurrency gewährleistet werden
- Dauerhaftigkeit (Durability) / Recovery
 - o Daten sollen dauerhaft gespeichert werden

4.1.1 Isolation / Nebenläufigkeit (Concurrency)

Durch die Nebenläufigkeit kann es im Datenbanksystem zu Fehler kommen. Am besten wäre es natürlich, wenn gar keine Fehler auftreten – dies hätte jedoch negative Auswirkung auf die Performance. Aus diesem Grund «lebt man mit gewissen Fehlern»

Lost-Update

Überschreiben bereits getätigter Updates

Bei der Nebenläufigkeit kann ein Lost-Update-Problem auftauchen

	Transaktion Benutzer 1	Transaktion Benutzer 2
1	SELECT Wert INTO W FROM Tbl	
2		SELECT Wert INTO W FROM Tbl
3	UPDATE Tbl SET Wert = 100	
4		UPDATE Tbl SET Wert = 200
5	SELECT Wert INTO W FROM Tbl	
6		SELECT Wert INTO W FROM Tbl

- Keine Isolation der Transaktionen beider Benutzer → Update-Operation von Benutzer 1 geht verloren!
- Lösung: Durch andere Transaktion gelesene Daten dürfen bis zur deren Beendigung nicht verändert werden.

→ Grundsätzlich wollen wir das gar nicht, kann aber ein Anwendungfall bspw. bei read-only

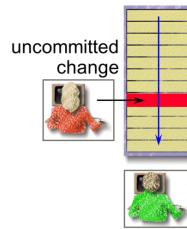
Wichtig: Lost-Update muss in jedem Fall behoben werden

Dirty-Read-Problem

Lesen von einer Veränderung noch nicht abgeschlossener Transaktion

Dirty-Read: Lesen von Veränderungen noch nicht abgeschlossener Transaktionen

	Transaktion Benutzer 1	Transaktion Benutzer 2
1	SELECT Wert INTO W FROM Tbl	
2	UPDATE Tbl SET Wert = NeuerWert	
3		SELECT Wert INTO W FROM Tbl
4	ROLLBACK	
5		UPDATE Tbl SET Wert = W + 1
6		SELECT Wert INTO W FROM Tbl
7	SELECT Wert INTO W FROM Tbl	



- Keine Isolation der Transaktionen beider Benutzer → Benutzer 2 arbeitet mit einem nicht bestätigten Wert, der später sogar zurückgenommen wird!
- Lösung: Nur Updates bestätigter Transaktionen lesen.

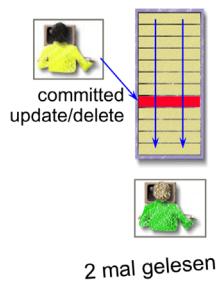
Wichtig: Dirty-Read ist in der Praxis zum Teil akzeptabel, die Konsequenzen müssen aber klar durchdacht werden

Non-Repeatable-Read

Lesen von zwischenzeitlich von anderen Transaktionen durchgeführten Veränderungen

Non-Repeatable-Read: Lesen von zwischenzeitlich von anderen Transaktionen durchgeführten Veränderungen

	Transaktion Benutzer 1	Transaktion Benutzer 2
1	SELECT Wert INTO W FROM Tbl	
2		UPDATE Tbl SET Wert = Wert + 5
3		COMMIT
4	SELECT Wert INTO W FROM Tbl	



- Keine Isolation der Transaktionen beider Benutzer → Benutzer 1 erhält nicht immer denselben Wert!
- Lösung: Nur den Datenbankzustand sehen, der bei Beginn einer Transaktion vorliegt.

Wichtig: Es ist vorstellbar, dass dies da und dort akzeptabel ist. Denn eine Verhinderung dieses Problems würde das Ziel von einem möglichst hohen Grad an Parallelität im Datenbanksystem konkurrenzieren. → Je höher den Isolationslevel, desto geringer die Parallelität.

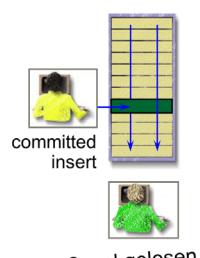
Dieser Fehlertyp wird in der Praxis daher häufig auch zugelassen.

Phantom-Read

Lesen von zwischenzeitlich von anderen Transaktionen durchgeführten Veränderungen

Phantom-Read: Lesen von zwischenzeitlich von anderen Transaktionen durchgeführten Veränderungen

	Transaktion Benutzer 1	Transaktion Benutzer 2
1	SELECT count(*) INTO cnt FROM Tbl	
2	N = cnt	
3		INSERT INTO Tbl VALUES (...)
4		COMMIT
5	SELECT count(*) INTO cnt FROM Tbl	
6		



- Keine Isolation der Transaktionen beider Benutzer → Benutzer 1 sieht Zwischenresultate, die von Benutzer 2 eingefügt wurden!
- Lösung: Nur den Datenbankzustand sehen, der bei Beginn einer Transaktion vorliegt

Wichtig: üblich diesen Fehler in der Praxis zuzulassen

4.2 Isolationsebenen

Ist im SQL-92 Standard

```
SET TRANSACTION ISOLATION LEVEL {READ UNCOMMITTED | ...}
```

Isolationsebene	Dirty Read	Non-Repeatable Read	Phantom Read	Lost Update	
READ UNCOMMITTED	möglich	möglich	möglich	möglich	
READ COMMITTED	Häufig in der Praxis	verhindert	möglich	möglich	verhindert
REPEATABLE READ	verhindert	verhindert	möglich	verhindert	
SERIALIZABLE	verhindert	verhindert	verhindert	verhindert	

Je strenger man ist, desto länger gehen die einzelnen Transaktionen (Serializable → ist am langsamsten)

4.3 Transaktionen in der Praxis

- COMMIT = Transaktion ist definitiv und permanent
- ROLLBACK / ABORT: Transaktion wird annulliert und ist für die anderen Transaktion unsichtbar

Wir müssen dem SQL-Server angeben wie viele Befehle kommen, daher *BEGIN* und *COMMIT*

```
BEGIN TRANSACTION
```

```
// Some SQL-Statement
```

```
COMMIT
```

→ Dadurch stellt man sicher, dass die ganze Transaktion durchgeführt wird.

Falls ein Befehl schief geht, kann SQL standardmäßig einen ROLLBACK, diesen kann man jedoch auch manuell einfügen

```
BEGIN TRANSACTION
```

```
// Some SQL-Statement
```

```
ROLLBACK
```

```
...
```

```
COMMIT
```

Wenn man nichts in SQL-Server einstellt, dann ist quasi jede einzelne Zeile eine Transaktion (→ nach einem DELETE ist es def. Gelöscht)

Es gibt verschiedene Zustände der Datenbank bei **offenen Transaktionen**:

- Vorhergehender Zustand wird in sogenannten **Before-Images** festgehalten und kann damit wiederhergestellt werden

- Betroffenen Sätze sind durch Schreib- / Lese-Sperren blockiert; sie können in anderen Transaktionen nicht gelesen / geändert werden
- Andere Transaktionen sehen geänderte Daten nicht (ausser bei Isolationsgrad *Uncommitted Read*)

Zustand DB nach COMMIT:

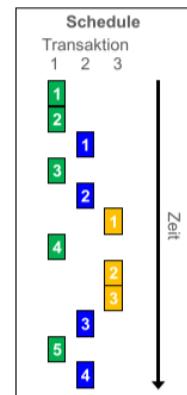
- Die geänderten Daten sind in der Datenbank festgeschrieben
- Alle Datenänderungen – alter Zustand (Before-Images) und neuer Zustand (After-Images) – sind in den Transaktionslogs protokolliert
- Vorhergehender Zustand kann **nicht** mittels ROLLBACK wiederhergestellt werden
- Alle Sperren der Transaktionen sind freigegeben
- Geänderte Daten können in Transaktionen mit *Read Committed Isolation* oder in nachfolgenden Transaktionen gelesen werden
- Geänderte Daten können in anderen Transaktionen geändert werden

Zustand DB nach ROLLBACK:

- Geänderte Daten sind vollständig zurückgesetzt
- Herstellen des alten Zustandes durch Einsetzen der Before-Images
- Rollback-Vorgang wird auch in den Transaktionslogs aufgezeichnet
- Sperren der betroffenen Daten sind freigegeben
- Daten können in anderen Transaktionen gelesen und geändert werden

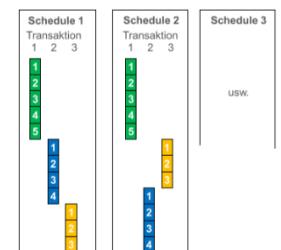
4.4 Schedules

- Schedule (Ablaufplan)
 - o **Nicht** das gleiche wie Ausführungsplan der Query
 - o Beschreibt wie mehrere Transaktionen in welcher Reihenfolge ihre Operationen ausführen/mischen
 - o Können ACID-Eigenschaften (eine oder mehrere) verletzen
- Vollständiger Schedule = History
 - o Sämtliche Schritte aller anstehenden Transaktionen inkl. Terminierung (COMMIT, ABORT)
 - o Für jede Transaktion ist festgehalten, ob sie erfolgreich endet oder abbricht (dies kann im Schedule noch offen sein)



4.4.1 Serieller Schedule

- Alle Transaktionen nacheinander
- Für n Transaktionen existieren $n!$ verschiedene serielle Schedules
- Serielle Schedules werden als konsistenzerhaltend betrachtet
- Sicher, aber schlechte Performance → verschränkte Ausführung ist erwünscht

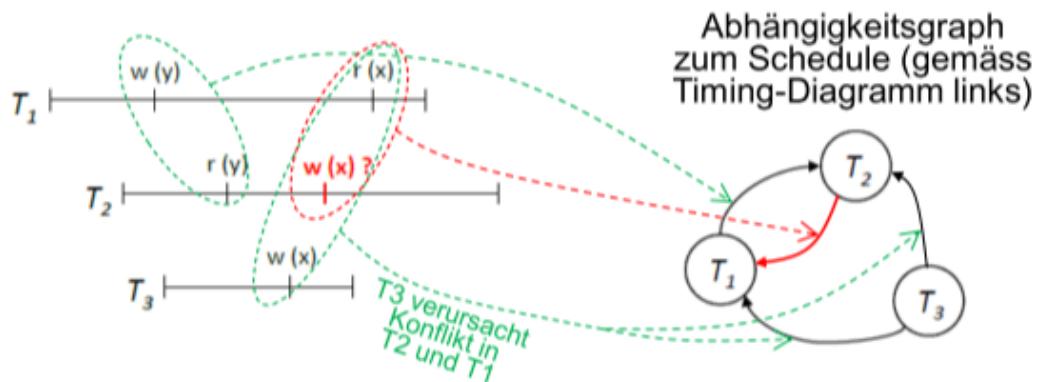


4.4.2 Serialisierbarer Schedule

- Ist auch konsistenzerhaltend
- Ist serialisierbar, falls dieser zu einem 'identischen' seriellen Schedule umgewandelt werden kann, so dass das Ergebnis dasselbe ist
- Konfliktserialisierbar → hat denselben Effekt auf die Datenbank, wie einer der seriellen Schedules. Hierzu existieren effiziente Algorithmen
 - o Abhängigkeit (Konflikt) besteht, wenn zwei Transaktionen auf dasselbe Objekt mit reihenfolgeabhängigen Operationen zugreifen. Konfliktarten:
 - Schreib-/Lese-Konflikt $w_1(x) r_2(x)$

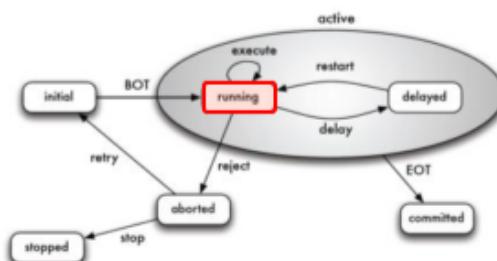
- Lese-/Schreib-Konflikt $r_1(x) w_2(x)$
- Schreib-/Schreib-Konflikt $w_1(x) w_2(x)$
- Ein Schedule s ist nun konfliktserialisierbar, falls ein serieller Schedule existiert, dessen Konflikte identisch sind
- Nachweis der Serialisierbarkeit: Führen von zeitlichen Abhängigkeiten zwischen Transaktionen in einem Abhängigkeitsgraphen (Konfliktgraphen)

Man kann zeigen, dass ein Schedule serialisierbar ist (d.h. er führt zum selben Resultat wie ein beliebiger serieller Schedule), wenn der Abhängigkeitsgraph **keine Zyklen** enthält (in $O(n^2)$ entscheidbar):



Dieser Schedule ist **nicht serialisierbar**, da der Graph einen Zyklus aufweist.

Eine Transaktion kann folgende **Zustände** einnehmen:



Für den Scheduler bestehen folgende drei Möglichkeiten zur Behandlung eines Schrittes einer laufenden (**running**) Transaktion:

- Ausführen (**execute**): Transaktion \rightarrow Zustand **running**
- Verzögern (**delay**): Transaktion \rightarrow Zustand **delayed**
- Zurückweisen (**reject**): Transaktion \rightarrow Zustand **aborted** (z.B. wenn Serialisierbarkeit gefährdet würde)

Scheduling-Verfahren:

1. Ein **Scheduler** arbeitet **aggressiv**, wenn er Konflikte zulässt und dann versucht, aufgetretene Konflikte zu erkennen und aufzulösen:
 - Ziel: Maximiert die Parallelität von Transaktionen
 - Risiko: Transaktionen werden erst am Ende ihrer Ausführung zurückgesetzt
 - Grenzfall: Im Extremfall ist keine Transaktion mehr erfolgreich
 - Beispiel: Oracle (Scheduling-Verfahren: Mehrversionensynchronisation)
2. Ein Scheduler arbeitet **konservativ**, wenn er Konflikte möglichst vermeidet, dafür aber Verzögerungen von Transaktionen in Kauf nimmt:
 - Ziel: Minimiert den Rücksetzungsaufwand für abgebrochene Transaktionen
 - Risiko: Erlauben eine geringere Parallelität von Transaktionen
 - Grenzfall: Im Extremfall findet keine Parallelisierung von Transaktionen mehr statt, d.h. die Transaktionen werden sequentiell ausgeführt
 - Beispiel: SQL Server (Scheduling-Verfahren: Sperrverfahren; nutzt 7 Haupt-Lock-Modi und diverse Untermodi zur Optimierung)

4.5 Transaktionsverwaltung: Sperrverfahren

- Sperren (Locks) auf Datenbankobjekten
 - o Abfrage der Sperre vor jedem Zugriff auf ein Datenbankobjekt und
 - o Setzen einer Sperre wenn verfügbar
- Erweiterung jeder Transaktion durch spezifische Operationen
 - o **Lese-Sperre** (Share Lock): read lock (rl) – read unlock (ru)
 - Andere Transaktionen weiterhin lesend auf das Datenbankobjekt zugreifen und Lesesperrern auf dieses Datenbankobjekt setzen. Sie können jedoch keine Schreibsperrre setzen
 - o **Schreib-Sperre** (Exclusive Lock): write lock (wl) – write unlock (wu)
 - Das betreffende Datenbankobjekt nur dieser Transaktions zur Verfügung und kann nur durch diese Transaktion geändert werden
 - o **Unlock**: read unlock (ru) und write unlock (wu)
 - Warden überlickweise zu unlock (u) zusammengefasst

Regeln zur **Sperrdisziplin**:

- Schreibzugriff $w(x)$ nur nach Setzen einer Schreibsperrre $wl(x)$ möglich.
- Lesezugriffe $r(x)$ nur nach Setzen einer Lesesperrre $rl(x)$ oder $wl(x)$ erlaubt.
- Eine Schreibsperrre $w(x)$ kann nur gesetzt werden, wenn auf x keine Sperren existiert.
- Eine Lesesperrre $r(x)$ kann nur gesetzt werden, wenn auf x keine Schreibsperrre existiert.
- Nach $u(x)$ darf die Transaktion kein erneutes $rl(x)$ oder $wl(x)$ ausführen.
- Eine Transaktion darf eine Sperre der selben Art auf demselben Objekt nicht nochmals anfordern.
- Beim Commit/Rollback müssen alle Sperren aufgehoben werden.

Sperren führen zu folgenden vier Problemen:

- Blocking
 - o Eine gesperrte Ressource zwingt andere Prozesse zu warten, bis diese wieder freigegeben wird. → Reduktion des Durchsatzes an Transaktionen

- Lösung: Keine, ohne Sperren funktioniert die Transaktionsverwaltung nicht
- Hinweise für Praxis
 - Lange laufende Abfragen vermeiden, kurze Transaktionen
 - Ineffiziente Abfragen optimieren
 - Vernünftig indizieren und Indexe verwenden
 - Keine Benutzereingaben innerhalb von Transaktionen
 - Sperr-Timeouts verwenden
 - ...
- Verhungern, Livelock
 - Eine Transaktion kommt nie dran, weil immer wieder andere vorher berücksichtigt werden
 1. T_1 sperrt x
 2. T_2 will x sperren, muss aber warten
 3. T_3 will danach x sperren, muss auch warten
 4. T_1 gibt x frei
 5. T_3 kommt vor T_2 an eine Zeitscheibe, sperrt x
 6. T_2 will weiterhin x sperren, muss aber warten
 7. T_4 will danach x sperren, muss auch warten
 8. T_3 gibt x frei
 9. T_4 kommt vor T_2 an die nächste Zeitscheibe ...
 - Lösung: RDBMS muss geeignet («fair») auswählen
- Deadlock (Verklemmung)
 - Eine Menge von Transaktionen sperren sich gegenseitig, wenn jede Transaktion der Menge auf ein Ereignis wartet, das nur durch eine andere Transaktion der Menge ausgelöst werden kann. Da alle am Deadlock beteiligten Transaktionen warten, kann keine ein Ereignis auslösen, so dass eine andere geweckt wird. Also warten alle beteiligten.
 - Lösung: RDBMS erkennt Deadlocks durch Zyklensuche im sog. «wer wartet auf wen»-Graphen
 - Opfer-Transaktion auswählen und zurücksetzen
 - Zu einem späteren Zeitpunkt neustartet
 - Hinweise für Praxis
 - Objekte immer in **derselben Reihenfolge** ansprechen
 - 'Teure' Objekte zuletzt ansprechen
 - Angepassten Isolationslevel verwenden
- Phantom-Read

Dabei gibt es verschiedene Lösungsansätze

- Lösung 1
 - Zusätzlich zu den Tupeln muss auch der Zugriffsweg, auf dem man zu den Objekten gelangt ist, gesperrt werden

Beispiel: `SELECT COUNT(*) INTO X FROM Mitarbeiter`

 - Alle Mitarbeiter (bzw. deren Primärschlüssel-Index) müssen mit einer RL-Sperre belegt werden.
 - Beim Einfügen eines neuen Mitarbeiter wird dies erkannt und T2 muss warten.

Sperre kann ggf. auch selektiver sein – z.B.: `SELECT COUNT(*) INTO X FROM Mitarbeiter WHERE PNr BETWEEN 1000 AND 2000`

 - Nur die Mitarbeiter mit der entsprechenden PNr müssen gesperrt werden (z.B. Index-Bereich von PNr[1000,2000])
- Lösung 2

- Von der Tabelle vorgängig einen Snapshot erstellen
- Lösung 3 (unabhängig vom Isolationslevel)
 - Die Tabelle ganz sperren

4.6 Recovery

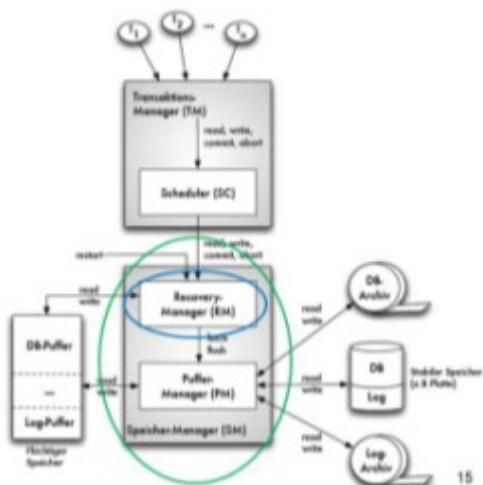
Alle Massnahmen zur Wiederherstellung verloren gegangener Datenbestände. Je nach Fehlerart müssen unterschiedliche Behandlungsstrategien ausgeführt werden.

- Transaktions-Manager / Scheduler
 - Wahrt die Isolations- und Konsistenz-eigenschaft einer Transaktion
- Recovery-Manager
 - Sichert die Atomaritäts- und Dauerhaftigkeitseigenschaft einer Transaktion

Der Speicher-Manager, besonders wichtig für das Recovery, bildet die Schnittstelle zwischen flüchtigem und stabilem Speicher, er umfasst den Recovery-Manager und den Puffer-Manager

Der **Recovery-Manager** sorgt dafür, dass nach einem Fehler:

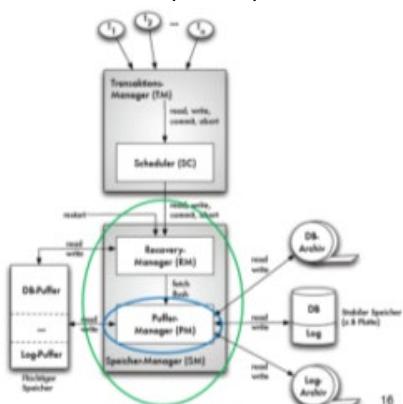
- Alle Änderungen «committeter» Transaktionen im stabilen Speicher abgelegt werden
- Keine Änderungen von aktiven oder abgebrochenen Transaktionen im stabilen Speicher



15

Der **Puffer-Manager** verwaltet den Puffer:

- Holt Daten (Seiten) vom stabilen Speicher in den Puffer
- Schreibt Daten (Seiten) vom Puffer in den stabilen Speicher
- Ersetzt Daten (Seiten) im Falle eines «Pufferüberlaufs»

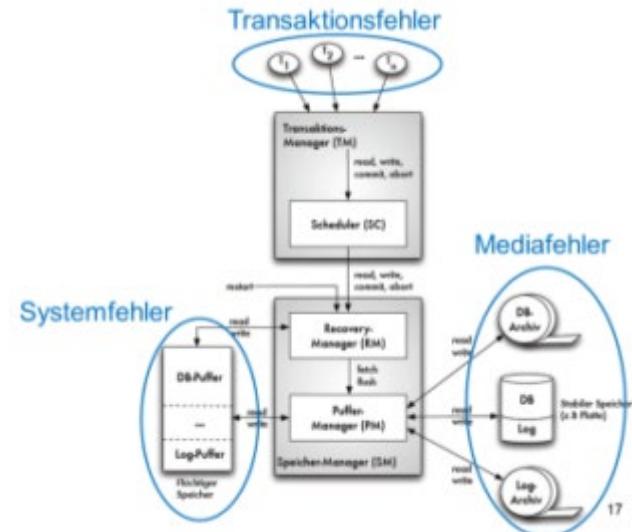


16

4.6.1 Fehlerklassifikation

- Transaktionsfehler
- Systemfehler
- Mediafehler

➔ Verschiedene Recovery-Massnahmen



17

5 Glossar

Fachbegriff	Beschreibung
DBS	Datenbanksystem
DBMS	Datenbankverwaltungssystem
AP	Anwendungsprogramm
IS	Informationssystem (DBS + AP)
Data Dictionary	Datenbeschreibung der Daten in einer Datenbank, jede Datenbank hat ein solches Dictionary. → Metadaten sind abgelegt
Datenunabhängigkeit	Obwohl Daten verändert werden, sollen alle Applikationen, welche dieses Attribut nicht verwenden keinen Einfluss haben. Programme laufen weiter als ob es keine Änderungen gab.
ECA	Event Condition Action → Prinzip beim Trigger
Statement Trigger	For each statement welches genau einmal ausgeführt wird
Row Trigger	For each row, welches für jedes modifizierte Tupel ausgeführt wird
RDBMS	
NSM	N-ary Storage Model