Zusammenfassung der Vorlesungen

Einführung in Datenbanken (Prof. Jarke)
Implementierung von Datenbanken (Prof. Jarke)
Indexstrukturen von Datenbanken (Prof. Seidl)

zur Vorbereitung der Vertiefungsprüfung bei Herrn Prof. Jarke

Mir hat diese Zusammenstellung sehr geholfen.
Hoffe, dass sie auch Euch eine Hilfe sein kann.

Zusammenfassung von Indexstrukturen von Datenbanken eher grob,
da Prüfung bei Herrn Prof. Jarke

Daniel Dünnebacke
Aachen, Mai 2007

Anmerkungen an gromit56@web.de! Danke

Inhaltsverzeichnis

Einführung in Datenbanken

Datenbankentwurf

Ziele des Datenbankentwurfs: **(KEVAMS)**

- ***K**orrektheit*: Wiedergabe der Konzepte des Modells, Integrität, Konsistenz, Datenwiederherstellung
- ***E**ffizienz*: Reaktionszeit, Kosten von Anfragen, Ressourcen
- ***V**ollständigkeit* aller relevanten Anwendungsaspekte
- ***A**daptivität*: Anpaßbarkeit an neue oder unterschiedliche Anforderungen
- ***M**inimalität*: Keine nichtnotwendige Redundanz
- ***S**icherheit*: Datenschutz, Authorisierungsmechanismen, Verfügbarkeit

Vorgehen

- Anforderungsanalyse
- konzeptioneller Entwurf
  - Der Entwurf ist unabhängig vom Zielsystem
  - Definition von Objekten (Entitäten), Relationen, Anwendungsregeln, is_A, Constraints (ER-Diagramm $\leftrightarrow$ UML)
- logischer Entwurf
  - Konvertierung der Konzepte in ein konkretes Datenmodell
- Implementierung
- Validierung/ Akzeptanztest

ER-Diagramm

Entitäten
sind Objekte oder Personen der realen Welt. Sie werden durch Attribute beschrieben.

Attribute
charakterisieren und unterscheiden Entitäten voneinander. Weiterhin charakterisieren sie Beziehungen zwischen den Entitäten. Schlüsselattribute (unterstrichen) dienen als eindeutiges Identifizierungsmerkmal

Mehrwertige Attribute
können mehrere Werte aus ihrem Wertebereich annehmen

Beziehungen
verbinden Entitäten miteinander. Sie können ebenfalls Attribute besitzen

Kardinalitäten
geben an, wie viele Entitäten eines Typs an einer Beziehung teilnehmen können. Teilnahmezwang kann allerdings nicht dargestellt werden.

Arten:
- 1:1 – Beziehung
- 1:m – Beziehung
- n:m – Beziehung
- min:max – Beziehung (Definition unterer bzw. oberer Grenzen)

Generalisierungen und Spezialisierungen
werden mit Hilfe der isA-Beziehung ausgedrückt. Sie stehen für Vererbungsbeziehungen zwischen allgemeinen und speziellen Entitäten.

Möglichkeiten:
- disjunkt: à in Richtung Spezialisierung
- nicht disjunkt: Überlappungen möglich. à in Richtung Generalisierung
- total: vollständige Dekomposition: t neben dem Symbol
- teilweise: mehr zerlegbare Teile möglich: p neben dem Symbol

Aggregationen
eigenen sich zum Zusammenbau von komplexen Objekten. Dies erhöht die Lesbarkeit des Diagramms, interne Strukturen werden so versteckt.

UML vs. ER

(Kemper, S.57ff)
Anders als im ERD beschreibt eine Klasse nicht nur die strukturelle Repräsentation (Attribute) der Objekte sondern auch deren Verhalten in der Form von zugeordneten Operationen.

Objekte/ Klassen	Entitäten
Assoziation	Beziehungen mit Attributen
Generalisierung/ Spezialisierung	Generalisierung/ Spezialisierung
Aggregation	Kategorien (über existenzabhängige Entities)

Lässt sich Aggregation auch im ER-Diagramm ausdrücken?
Ja, Simulation über Assoziationen, aber etwas andere Semantik, da in UML ein Objekt nur an einer Aggregation teilnehmen kann, dieser Constraint lässt sich in ER nicht ausdrücken!

Algorithmus zur Übersetzung von ER in Relationenschema

relationale Entwurfsziele **(RINN)**
- geringe **R**elationenzahl (Joins teuer)
- einfacher Charakter von **I**ntegritätsbestimmungen (Schlüsselbedingungen,…)
- potentiell wenige **N**ullwerte (Platzverbrauch und Semantikprobleme)
- gute **N**ormalformeneigenschaft zur Verhinderung von Anomalien

Entitäten mit Attributen
Jede Entität wird zu einer Relation (Tabelle). Ihre Attribute bilden ihre Spalten. Zusammengesetzte Attribute werden aufgelöst und ihre atomaren Bestandteile werden in der Tabelle erfasst. Für jedes mehrwertige Attribut wird eine neue Tabelle erstellt, welche als Schlüssel den zusammengesetzten Schlüssel bestehend aus dem Primärschlüssel der Entität zusammen mit dem Attributsnamen erhält. Bei starken Entitätstypen bilden die unterstrichenen Attribute auch den Primärschlüssel. Bei schwachen Entitätstypen werden ebenfalls alle Attribute miteinbezogen. Als Fremdschlüssel wird der PK der Beziehung verwendet, die mit den Eigentümerentitätstypen korrespondiert. Der Primärschlüssel wird aus der Kombination des PK der Eigentümerentität, dem Fremd- und partiellen Schlüssel gewählt.
1:n – Beziehungen

Diese Beziehungen werden in die betroffene Entität auf der „n-Seite“ integriert. Die Relation der „n-Seite“ erhält als Fremdschlüssel den Schlüssel der Entität auf der „1- Seite“ und alle Attribute der Beziehung.
Grund: Jede Entitätsinstanz hängt mit genau einer Instanz auf der „1-Seite“ zusammen.

1:1 Beziehungen
Handelt es sich um zwei totale Teilnahmen, so werden beide Entitäten zu einer Relation verschmolzen. Die betroffenen Entitäten dürfen nicht an anderen Beziehungen teilnehmen.
Ansonsten erhält der Entitätstyp mit totaler Teilnahme als Fremdschlüssel den PK des anderen Typen und alle Attribute der Beziehung.

m:n - Beziehungen
Die Beziehung wird als neue Tabelle erstellt. Ihr Schlüssel setzt sich aus den PKs der teilnehmenden Entitäten zusammen. Die teilnehmenden Entitäten werden nach den bekannten Regeln übersetzt.
Anmerkung: Diese Methode ist auch bei allen anderen Beziehungstypen möglich. Es können so Nullwerte in Fremdschlüsseln vermieden werden.

Rekursive Beziehungen
Es werden zwei unabhängige Tabellen erstellt. Ihr Schlüssel ist jeweils die Kombination der PKs der beiden teilnehmenden Entitäten.

Generalisierung / Spezialisierung
Vier Möglichkeiten:
♦ Erstelle eine Tabelle für die Generalisierungsentität, dann für jede Spezialisierung eine Tabelle mit den Attributen der jeweiligen Spezialisierung und dem Primärschlüssel der Generalisierung.
♦ Erstelle eine Relation für jede Subklasse mit den passenden Attributen und dem PK der Generalisierung
♦ *disjunkte Subklassen:* Erstelle eine einzelne Relation mit allen Attributen, demselben PK und einem *diskriminierenden Attribut*, welches die Zugehörigkeit eines Tupels zu einer Subklasse ausdrückt (z.B. Stellentyp)
♦ *überlappende Subklassen:* Erstelle eine Relation mit demselben PK der Generalisierung. Es werden für jeden Subtyp diskriminierende Attribute eingeführt, die boolesche Werte die Zugehörigkeiten der Tupel zu den jeweiligen Subklassen anzeigen.

SQL-Abfrage

	RA-Algebra
Select…	Projektion (unsortiert n*log(n))
From…	Join; kartesisches Produkt (Auswahl der zu bearbeitenden Relationen)
Where…	Selektion
Group by…	
Order by…	
Having…	
	Umbenennung
	Vereinigung
	Mengendifferenz
	Schnitt

Views

Für Views gibt es keine Modellierung im rel. Modell. Views (in SQL) sind virtuelle Tabellen. Durch sie ist es möglich, Spalten aus verschiedenen Basistabellen zusammenzuführen und deren Spaltennamen umzubenennen

Relationales Datenmodell

Ein Datenmodell dient allgemein der Herstellung einer Abstraktion zum Zwecke der Beschreibung eines Problems (Realweltsituation)

Abstraktion

- Klassifikation
- Aggregation (Zusammenbau)
- Verallgemeinerung bzw. Spezialisierung

$D = (R, \sum_R)$ (siehe Aufgabe 3.2 der Übung zu EDB aus WS 05/06)

D = relationales Datenbankschema
R = Menge von relationalen Schemata
$\sum_R$ = Menge von interrelationalen Abhängigkeiten

Relationen: Teilmenge des kartesischen Produktes der Wertemengen der Attribute
Tupel: Elemente von diesen
Operationen: Algebra/ Kalkül

Operationen der Relationenalgebra:

- Schnitt, Vereinigung, Differenz benötigen verträgliche Attributmengen
- Projektion, Selektion, Join, Renaming

Integritätsbedingungen:

Interrellational: z.B. Foreign Keys
Intrarelational: Keys

Tupel- /Domaincalculus

Tuple:
$$\{n \mid (\exists a)(\exists b) : Angestellter(a) \wedge Abteilung(b) \wedge b.Ort = "Düsseldorf" \wedge a.name = n\}$$

Domain:
$$\{n \mid (\exists A\#)(\exists Abt\#)(\exists Mgr\#) : Angestellter(A\#, n, Abt\#) \wedge Abteilung(Abt\#, "Düsseldorf", Mgr\#)\}$$

Functional Dependencies

Eigenschaft:
Abschlusseigenschaft: Das Ergebnis einer relationalen Operation ist stets wieder eine Relation. Dies ist ein Vorteil gegenüber objektorientierten Anfragesprachen.

Definition: Sei V eine Menge von Attributen X, Y aus V, r ist aus Rel(V)

(x à y) (r) := 1, falls für alle u, w aus r: u(x) = w(x) => u(y) = w(y)
0 sonst

Eine funktionale Abhängigkeit (bezeichnet durch X à Y, zwischen zwei Attributmengen X und Y) die Teilmengen eines universellen Relationsschemas („eine große Tabelle mit eindeutigen Attributsnamen“) sind, bezeichnet die Abhängigkeit der Attributsmenge Y von der Menge X. Das heißt, dass die Werte von Y eindeutig durch die Werte von X bestimmt sind.

Key Definition:
Sei K eine Teilmenge von V

(K à V) (r) := 1, falls für alle u, w aus r: u(K) = w(K) => u(V) = w(V) = r

Armstrong Axiome

- Reflexivität $Y \subseteq X \Rightarrow X \rightarrow Y$
- Erweiterung $X \rightarrow Y \wedge Z \subseteq W \Rightarrow XW \rightarrow YZ$
- Transitivität $X \rightarrow Y \wedge Y \rightarrow Z \Rightarrow X \rightarrow Z$

von FDs zum Ziel der Erhaltung der funktionalen Abhängigkeiten.

Beweis direkt aus den Fds!

Weitere (RAP):

- Reflexivität $X \rightarrow X$
- Akkumulation $X \rightarrow YZ \wedge Z \rightarrow AW \Rightarrow X \rightarrow YZA$
- Projektivität $X \rightarrow Y \wedge Z \subseteq Y \Rightarrow X \rightarrow Z$

FDs korrekt und vollständig (Beweis siehe Vossen, S.158)

Normalformen

(Vossen, S.193ff)
Nachteile Normierung: Fragmentierung der Informationen

- wegen der Anomalien notwendig
- aber Zusammensetzen der Fragmente mit Join teuer

Die ersten drei Normalformen sowie die Boyce-Codd Normalform basieren auf den vorgestellten funktionalen Abhängigkeiten zwischen den Attributen einer Relation.

Ziele der Normalisierung sind die Erreichung von minimaler Redundanz und die Minimierung/ Eliminierung von Einfüge-, Lösch- und Update-Anomalien (Vossen, S.192).

Die Eigenschaft des **verlustfreien Joins** $((\forall r \in Sat(R))\ r =>< \bowtie_{i=1}^{k} \pi_{X_i}(r))$ gewährleistet, dass in Bezug auf die nach der Zerlegung erstellten Relationenschemas keine unechten Tupel erzeugt werden. (Test durch Tableau-Methode)

Die Eigenschaft der **Abhängigkeitswahrung** $((\bigcup_{i=1}^{k} F_i)^+ = F^+)$ gewährleistet, dass jede funktionale Abhängigkeit nach der Zerlegung in mindestens einer Relation dargestellt wird. (Test durch Abschlussberechnung durch Armstrong-Axiome)

Die Eigenschaft des verlustfreien Joins muss auf jeden Fall erreicht werden, während die Eigenschaft der Abhängigkeitswahrung manchmal geopfert werden kann (siehe später).

Weiterhin besteht kein Zwang, bis zur höchstmöglichen Normalform normalisieren zu müssen

1NF
Die Relation sollte keine nicht atomaren Attribute oder verschachtelte Relationen enthalten.
Lösung: Erstelle für jedes nicht atomare Attribut oder jede verschachtelte Relation neue Relationen.

2NF
In Relationen, deren Primärschlüssel mehrere Attribute enthalten, sollte kein Nichtschlüsselattribut funktional von einem Teil des Primärschlüssels abhängen. (l-minimal)
Lösung: Zerlege die Relation und erstelle eine neue für jeden partiellen Schlüssel mit seinen abhängigen Attributen. Erhalte die (restliche) Relation mit dem ursprünglichen Primärschlüssel und Attributen, die von diesem funktional voll abhängig sind.

3NF
Eine Relation sollte kein Nichtschlüsselattribut enthalten, das funktional von einem anderen Nichtschlüsselattribut (oder von einer Menge von Nichtschlüsselattributen) bestimmt wird. Das heißt, es sollte keine transitive Abhängigkeit eines Nichtschlüsselattributs vom Primärschlüssel bestehen.
Lösung: Zerlege die Relation und erstelle eine neue, die das bzw. die Nichtschlüsselattribute beinhaltet, die funktional von anderen Nichtschlüsselattributen bestimmt werden.

BCNF
Geringfügiger Unterschied zu 3NF: Das verletzende Attribut kann nur prim sein.
Problem: Bei der BCNF können funktionale Abhängigkeiten verloren gehen. Der später präsentierte Algorithmus stellt jedoch sicher, dass so wenige wie möglich bei der Zerlegung verloren gehen. Strenger als 3NF, da jede NF in BCNF auch in 3NF ist. Nur falls X à A in einem Relationsschema R gilt, während X kein Superschlüssel *und* A ein Prime-Attribut ist, befindet sich R nur in 3NF und nicht in BCNF.

Algorithmen zur Prüfung und Erstellung besserer Schemas

Die Dekomposition
Input: Ein universelles Schema R(U,F)
Output: Eine verlustfreie BCNF – Zerlegung D=(R,.) von R
à Allerdings können funktionale Abhängigkeiten verloren gehen (nicht immer abhängigkeitserhaltend), der Algorithmus stellt allerdings sicher, dass nur eine minimale Menge von FDs verloren geht.

Der Synthesealgorithmus
Die Idee des Verfahrens besteht grob gesagt darin, für jede (neue) linke Seite einer FD in einer Basis für die vorgegebene FD-Menge ein eigenes Schema zu erzeugen.

Input: Ein universelles Schema R(U,F)
Output: Eine verlustfreie, unabhängige Zerlegung in 3NF (mit beibehalten aller FDs) D=(R,.) von R. Dieser Algorithmus arbeitet schneller als der Dekompositionsalgorithmus.

Anmerkung: Die beiden Algorithmen sind nicht-deterministisch. Der Synthesealgorithmus hängt von der erzeugten minimalen Hülle von F ab, d. h. es existieren mehrere minimale Hüllen. Der Zerlegungsalgorithmus ist abhängig von der Reihenfolge, in der ihm die funktionalen Abhängigkeiten zugeführt werden.
à Einige Entwürfe können somit anderen unterlegen oder sogar gänzlich unerwünscht sein.

OO-Datenbanken (Kemper, S. 357)

Konzepte:

- Objekte und Klassen
- Generalisierung/ Spezialisierung und Vererbung
- OID (zustands- und ortsunabhängiger Objektidentifier) [a) + b)]
  à jedes Objekt ist eindeutig über seine automatisch vom System generierte Identität referenzierbar
- Polymorphität
- Integration von Verhaltens- und Struktur-Beschreibung [c) + d)]
  à Operationen werden integraler Bestandteil der Objektdatenbank
  à Vermeidung der umständlichen und i.A. ineffizienten Transformationen zwischen Datenbank und Programmiersprache
  à Durch Objektkapselung (Geheimnisprinzip), da die interne Struktur den Benutzern verborgen bleibt

Unzulänglichkeiten des relationalen Datenmodells (Kemper, S. 355)

a) Segmentierung
   - spätere Zusammenführung teuer

b) Künstliche Schlüsselattribute
   - häufig keine Anwendungssystematik, trotzdem müssen sie gewartet werden

c) Fehlende Verhaltens-Beschreibung

d) Externe Programmierschnittstelle

Darstellung von Entitäten nur als Tupel à besteht aus einer festen Anzahl von atomaren (nicht aus komplexeren Strukturen zusammengesetzt) Literalen à ist ein unveränderlicher Wert

Zum Vergleich von OODB und RDB siehe Aufgabe 9.1!!!!

XML

semi-strukturierte Daten (z.B. auch RDF): Darunter versteht man Daten, die zum großen Teil eine fest vorgegebene Struktur besitzen; gleichzeitig aber auch Elemente beinhalten, die diesem statischen Schema nicht unterliegen.

semi-strukturierte Daten weisen eine gewisse Struktur auf, können aber nicht in ein Datenbankschema eingeordnet werden. Das hat den Vorteil, dass sich Informationen einerseits strukturieren lassen, andererseits aber kein starres Schema eingehalten werden muss.

XML beschreibt die Struktur und die Semantik, nicht die Formatierung (im Ggs. zu HTML)

relationales Datenmodell à strukturiert
XML à semi-strukturiert
HTML à Formatierungssprache (kein Schema)

Bestandteile:

- optionale Präambel
- optionales Schema
- einziges Wurzelelement, welches beliebig viele und beliebig tief geschachtelte Unterelemente beinhalten kann

ohne Schema: wohl-geformt
mit Schema (umbedingt einhalten): valide oder gültig

Anfragesprache

XQuery	à flwr (gesprochen „flower“) - Ausdruck
for…	for $z in //zimmertyp return $z Bei der Auswertung wird $z mehrfach gebunden, nacheinander an alle Zimmertypen, und in jedem Schritt ausgegeben. Variablen werden sukzessive gebunden (Variablengültigkeit; SQL:from)
let…	let $z := //zimmertyp return $z Ergebnis ist eine Sequenz von Knoten, **$z** wird an eine Knotenmenge gebunden
where…	Selektion; optional
order by…	optional
return…	XML Struktur (z.B.<Kosten><VK>{$m/@Preis}</VK></Kosten>) ($--> Variablenbezeichnung)

Beispiel:
for $book in /medialist/book
where $book/author = „Nissen“
return $book

Rückgabe sind komplette Knoten mit daran hängenden Teilbäumen (strukturierte Rückgaben) gegenüber einfachen Werten in SQL (Werte von Attributen, die in der SELECT-Klausel stehen)!

XPath-Syntax als „Untersprache“!

Zugriff auf Elemente:
Das zentrale Konzept von XPath sind die so genannten Lokalisierungspfade, die aus aneinander gereihten - jeweils durch ein „/“ Zeichen voneinander abgetrennten - Lokalisierungsschritten bestehen. Ein Lokalisierungsschritt selektiert - ausgehend von einem Referenzknoten - eine Knotenmenge. Jeder der Knoten in dieser Menge dient dann als Referenzknoten für den nachfolgenden Lokalisierungsschritt in einem längeren Lokalisierungspfad. Alle im letzten Lokalisierungsschritt so selektierten Knotenmengen werden vereinigt und bilden das Ergebnis des gesamten Lokalisierungspfades.

Implementierung von Datenbanken

5-Schichten-Modell DBMS (siehe Übung 1.3 SS03, Folie 12)

Transaktion(sprogramm)

à Mengen-orientierte Schnittstelle (a) à

Translate and optimize queries — *logische Datenstruktur*

à Satz-orientierte Schnittstelle (b) à

Manage cursor, sort components and dictionary — *logische Zugriffsstrukturen*

à Interne Satzschnittstelle (c) à

Manage record and index — *Speicherstrukturen*

à System Buffer Schnittstelle (d) à

Manage buffer and segments — *Seitenzuweisung*

à Dateischnittstelle (e) à

Manage files and external memory — *Speicherzuweisungs-Strukturen*

à Geräteschnittstelle (f) à

physikalisches Laufwerk

Schnittstellen:
(a) set-oriented interfaces
- Objekte: Relationen, Views, Tuples
- Prozeduren: select, insert, delete, update (modify)

(b) record-oriented interfaces
- Objekte: record, set, keys, access paths
- Prozeduren: retrieve, dispose, change

(c) internal record interfaces
- Objekte: z.B. B*-Bäume
- Prozeduren: insert, delete, update (modify)

(d) system buffer interfaces
- Objekte: Seiten, Segmente
- Prozeduren: read, write

(e) file interfaces
- Objekte: blocks, files
- Prozeduren: read, write

(e) device interfaces

Objekte:	tracks, cylinder, channel
Prozeduren:	read, write

am Beispiel eines SQL-Ausdruckes (abgefragt=retrieved):
(a) Der SQL-Query wird in das System gegeben. Eine Liste von Titeln wird als Ergebnis des Queries abgefragt.
(b) Die Informationen des Titel-Feldes aller Records, die in dem Feld Autor „XYZ“ haben werden abgefragt.
(c) „XYZ“ wird in dem B*-Baum mit Autor im Index erreicht.
(d) Seiten und Segmente, wo die relevanten Records gespeichert werden, werden abgefragt.
(e) Blocks und Files,…
(f) Tracks und Zylinder,…

SQL-Anfrage über die verschiedenen Schichten:
1. Schicht ist mengenorientiert, dort startet die Anfrage und wird in entsprechenden Relationenalgebra-Ausdruck übersetzt. Dann wird Zugriffskontrolle für die entsprechenden Daten ausgeführt und die Überprüfung von Integritätsbedingen durchgeführt.
à Anfrageoptimierung
à Anfragegraphen, die auf dem Tupelkalkül basiert
à algebraische Optimierung, bei der Maß für die Güte der Optimierung die Größe der zu verarbeitenden Relationen bzw. die Anzahl der Tupel ist. Dazu wird ein Operatorbaum erzeugt und dort nach heuristischen Regeln die Operatoren der Relationenalgebra vertauscht. Dabei ist es wichtig Projektion und Selektion möglichst weit zu den Blättern zu verlagern und den Verbund so weit wie möglich nach oben (da teuer). Darüber hinaus sollte das kartesische Produkt ersetzt werden.

Tableau-Methode (auch in EDB)
High Level Optimierung zur Vermeidung überflüssiger Teilanfragen. (lossless-join)

Implementierung von Joins

		Komplexität
Hash-Join	Wenn die Basistabellen über die Joinspalten gehasht sind, Join in den Hashbuckets ausführen [sinnvoll, wenn Verwendung einer Hauptspeicher-Hashtabelle möglich ist]	O(n+m)
Merge/ Sort-Loop	Wenn die Basisrelationen passend sortiert sind, kommen die Tupel gleich in der passenden Reihenfolge [normalerweise linear] [WC: $O(n^2)$, wenn beide Relationen nur gleiche Attribute besitzen; Kreuzprodukt]	O(n*log(n))
Nested Loop	Wenn`s nichts besseres gibt [quadratisch]	O(n*m)

B/ B*-Bäume (auch in EDB)
„The Ideal Tree“:

- *Geringe Baumhöhe (Zugriffsschritte* $\log_k(n)$*)*
- *Balanciertheit*
- *Geringe Erhaltungs-/ und Änderungskosten*
- *Jeder Knoten mindestens zu 50% gefüllt*

Unterschiede zwischen B-Baum und B*-Baum:
Im B*-Baum sind die Daten in den Blättern gespeichert, der Rest dient als Wegweiser (Indexteil).
Im B-Baum sind die Daten in allen Knoten enthalten.

Im B*-Baum geschieht das Löschen und Einfügen nur in den Blättern.
Im B-Baum kann jeder Knoten verändert werden.

B*-Bäume (Mindestbelegung der Knoten von 2/3):
$k \le n \le 2k$ Datensätze in inneren Knoten (n+1 Pointer)
$k^* \le m \le 2k^*$ Datensätze in Blättern + 2 Pointer
$2(k+1)^{h-1} - 1 \le u \le (2k+1)^h - 1$ ist die Schlüsselanzahl bei Höhe $h \ge 1$

Kosten:
Suche h (genauer: $\log_k(n)$)

Eigenschaften:
- Wiederholen der Schlüssel in Blättern
- Mindestknotengröße
- Mindestzahl der Zeiger in Knoten
- **Ausgeglichenheit**

algebraische Optimierung

(Vossen, S. 306ff)
Anfrage-Optimierung, die algebraische Optimierung genannt wird, basiert auf der Anwendung algebraischer Regeln und hat insbesondere keinen Bezug auf die Datenstrukturen, in welchen die Operanden gespeichert sind.

Durch Anfrage-Bäume!

Äquivalenz:
Zwei Ausdrücke sind äquivalent, falls sie bei jeder Auswertung bezüglich des aktuellen Zustandes über dem vorgegebenen Datenbankschema das gleiche Ergebnis liefern.

Auf diese Weise ist es häufig möglich, die Auswertungs- bzw. Antwortzeit zu verkürzen, ohne die spezielle Wahl der Datenstrukturen für die Speicherung der Relationen, d.h. die konkrete Implementierung der Datenbank zu kennen.

Semi-Join

besonders effizient

$R \oplus S = \pi_R(R \otimes S)$

Ein Semi-Join Ausdruck ist ein Query in relational calculus das komplett auf Komponenten der folgenden Form reduziert werden kann:
{EACH r IN exp: SOME r* IN exp* (r.a=r*.b)}
à solche Ausdrücke sind gutartig (benign)!

Welche Ausdrücke können nicht durch Semi-Joins ausgewertet werden?
Solche mit Zyklen im Quantgraph. (hartnäckige, bösartige Teilausdrücke) à Zykel sind dann vorhanden, wenn eine Tupelvariable nicht nur in ihrem definierenden Niveau auftaucht, sondern zusätzlich auch in einem niedrigeren Niveau verwendet werden muss.

gutartige ß à bösartige Komponenten

Ein flacher Ausdruck im Relationenkalkül ist gutartig, wenn sein Ergebnis durch eine Semi-Join Reduktion berechnet werden kann, z.B. wenn es in eine äquivalente „Generalized Semi Join Expression“ (GSE) transformiert werden kann.
à
Ausdrücke im Relationenkalkül, die durch einen strikten „tree-like Quantgraph“ mit „EACH“-Wurzel, beliebigen Knoten und beliebigen Prädikaten repräsentiert werden können, sind gutartig

Quantgraphen (Kap2, Folien 51ff)

Zyklen im Quantgraphen
à bösartiger Ausdruck, wenn nicht wie nachfolgend beschrieben auflösbar.

Zyklen können aufgelöst werden, wenn:

- redundante Kanten gelöscht werden können
- wenn alle Absorber (Knoten mit zwei eintreffenden „predicate edges“) ALL-Absorber sind (Splitting-Absorber)
- durch „Comparison Doublets“, wenn es sich um SOME-Absorber handelt und die Joins durch Vergleiche $(>,<,\leq,\geq)$ vonstatten gehen. Dann neuer Knoten r* des Knoten r, der zuvor eine eingehende Kante hatte.

Kostenmodelle (Kap2, Folien 234ff)

Kostenmodelle des „access planning“ berücksichtigen in erster Linie Zugriffe auf den sekundären Speicher und zusätzlich die Kommunikationskosten bei verteilten Datenbanken.

Zwei Funktionen die einkalkuliert werden sollten:

- Query: DB à Anzahl der zugegriffenen Daten Records è Abschätzung der Größe des Zwischenergebnisses
- Access: {|zugegriffene Records|, Datenstruktur (#Blockzugriffe), Buffer des primären Speichers}

gegeben: Statistiken über den DB-Zustand

Konstruiere ein mathematisches Modell der Datenbank (Parameter System) und eine Menge von Formeln, welche die Größe des Ergebnisses für jede Sequenz von (algebraischer) Operationen abschätzen.

Transaktionsverwaltung

Grund für verzahnte Ausführung von Transaktionen:

- Performance
- Auslastung
- …

Bestandteile

- Serialisierung
- Fehlerrecovery (Gewährleistung von Atomarität und Durabilität [Dauerhaftigkeit])

Transaktion

Eine Transaktion ist eine Serie von Operationen (Lese- und Schreiboperationen), welche eine gegebene Datenbank von einem konsistenten Zustand in einen anderen (nicht unbedingt veränderten) konsistenten Zustand transformiert.

Alternativ: Der Zugriff auf eine Datenbank besteht aus einer Folge von Operationen (Lese-, Schreiboperationen). Das Programm oder die Gesamtheit dieser Operationen ist eine Transaktion.

Konflikte (Kemper, S.296ff)

- Dirty-Read $s = r_1(x)w_1(x)r_2(x)a_1w_2(x)c_2$
- Lost-Update $s = r_1(x)r_2(x)w_1(x)c_1w_2(x)c_2$
- Phantom-Problem $s = r_1(x)r_1(y)r_2(z)w_2(z)r_2(x)w_2(x)c_2r_1(z)c_1$

ACID-Prinzip

***A**tomicity:* Eine Transaktion wird entweder vollständig ausgeführt oder gar nicht. D.h. wenn im Laufe der Transaktion eine Operation fehlschlägt (o.ä.), dann werden alle Änderungen der DB vor der vorhergehenden Operation wieder rückgängig gemacht. Im anderen Fall werden die Änderungen beibehalten.

***C**onsistency:* Die Transaktion hinterlässt einen konsistenten Zustand, wenn sie aus einem solchen gestartet wurde, d.h. alle Integritätsbedingungen der DB werden eingehalten. Kommt es zu einem Fehler, so ist die DB in den Zustand wieder zurück zu versetzen, den die vor der Transaktion hatte.

***I**solation:* Die Transaktion läuft isoliert ab, also unabhängig von anderen Transaktionen.

***D**urability:* Nach erfolgreicher Durchführung einer Transaktion, sind die Daten gespeichert, so dass sie jeden folgend auftretenden Hard- oder Softwarefehler überstehen/ überleben.

Das ACID-Prinzip gewährleistet, dass eine Transaktion/ Programm korrekt ausgeführt wird, bzw. das gewünschte korrekte Ergebnis liefert. Tritt während der Transaktion ein Fehler auf, so hinterlässt sie keine „Spuren“ in der Datenbank, da sie atomar verarbeitet wird. Dieser wieder herzustellende Zustand ist aufgrund der Persistenz/ Durabilität stets wohl definiert. Durch die Isolation werden Anomalien im Mehrbenutzerbetrieb verhindert.

Read-Write-Modell (Kap3, Folien 10ff)

Eine Datenbank D={x,y,z,..} ist eine Menge von (Datenbank-) Objekten, zu denen die Schreib- und/ oder Leseopertionen Zugriff haben.

Eine Transaktion ist eine endliche Menge von Operationen der Form r(x) (read x) oder w(x) (write x) wie folgt: $t = p_1, p_2, ..., p_n$, mit $n < \infty, p_i \in \{r(x), w(x)\}$ *für* $1 \le i \le n$ *und* $x \in D$

Jeder von einer Transaktion t geschriebener Wert hängt von den Werten aller zuvor von t gelesenen Objekten ab.

Serialisierbarkeit

Wenn Transaktionen konkurrierend (parallel) ausgeführt werden, als wenn es seriell wären, dann wird dies serialisierte Schedules (Historien) genannt.

Ziel: Angabe eines Korrektheitskriteriums für Schedules:
Herbrand Semantik (Kap3, Folie 17); Herbrand Universum (Kap3, Folie 18)

Reads-From-Relation (RF(x))
$RF(s) = \{(t_i, x, t_j)\}$, d.h. dass $r_j(x)$ x von $w_i(x)$ liest

Life-Reads-From-Relation (LRF(x)
$LRF(s) = \{(t_i, x, t_j)\}$, d.h. dass ein lebendes $r_j(x)$ (es ist für ein t_∞ noch sinnvoll) x von $w_i(x)$ liest

FSR (Final State Serialisierbarkeit)
Herbrandsemantik der letzten Schreib-OP je Objekt/ LRF(s); nicht effizient überprüfbar (exponential in der Anzahl ?der Transaktionen?), Phantom-Probleme können auftreten

VSR (View Serialisierbarkeit)
Herbrandsemantik aller OPs/ RF(s); nicht effizient überprüfbar (NP-vollständig), nicht PCA

$VSR \subseteq FSR$; beide für praktische Anwendungen ungeeignet!!

CSR (Konfliktserialisierbarkeit)
Konflikt Relation/ Graph; PCA, effizient prüfbar (Zugehörigkeit zu CSR kann in polynominieller Zeit getestet werden; Graph bilden: O(n); auf Zyklen testen: $O(n^2)$), Problem: Dirty-Read (z.B.: $s = r_1(x)w_1(x)r_2(x)a_1w_2(x)c_2$)

Zwei Operationen $p \in t$ und $q \in t`$ mit $t, t` \in trans(s)$ und s Schedule (Historie) befinden sich in einem Konflikt, wenn sie auf dem selben Objekt (z.B. x,y,…) operieren und wenigstens einer von ihnen eine Schreib-Operation ist
à
C(s)={(p,q)|p,q sind im Konflikt und p steht vor q in s}
à
conf(s) bezeichnet die Konflikt-Relationen (aus C(s)), welche von den abgebrochenen Transaktionen bereinigt wurden.
à
Konflikt-Graph G(s)=(V,E) mit V=commit(s) [commited Transactions] und $(t, t`) \in E \Leftrightarrow t \neq t` \wedge (\exists p \in t)(\exists q \in t`)(p, q) \in conf(s)$
(Beispiel zu diesem Vorgehen: Kap3, Folien 27ff)
à
Serialisierungs-Theorem: $s \in CSR \Leftrightarrow G(s)$ hat keine Zyklen

Weitere: Commit-Serialisierbarkeit; gilt nur für korrekt ausgeführte Transaktionen

Deadlocks

Treten beim Zugriff auf die Daten auf, wenn Transaktionen parallel abgearbeitet werden und man einen sperrenden Scheduler verwendet. Werden von modernen DB nicht beachtet. Man kann ja die Transaktion einfach abbrechen.
Lösung: konservatives 2PL Scheduling

Ansätze zur Behebung von Deadlocks:

- Prevention
- Avoidance
- Detection + Recovery

Scheduler (Historie)

Beeinflusst eintreffenden Strom von Aktionen, so dass ausgehender Strom äquivalent zu seriellem unter Gewährleistung der Fehlersicherheitskriterien ist.

Aufgabe

- verzahnte Ausführung von Transaktionen (Wartezeiten von TA können von anderen TA genutzt werden à Beschleunigung gegenüber sequentiellen Fall)
- Korrektheit (Äquivalenz zu seriellem Schedule)

Eigenschaften:

- Rücksetzbarkeit (*RC*; Minimalanforderung)
  Eine Historie heißt rücksetzbar, falls immer die schreibende Transaktion vor der lesenden Transaktion ihr „commit“ ausführt, also $c_w <_H c_r$

  oder: Eine Transaktion darf erst dann ihr „commit“ durchführen, wenn alle Transaktionen, von denen sie gelesen hat, beendet sind.
- kaskadische Aborts vermeidend (*ACA*)
  mögliches Problem: Das Rücksetzen einer Transaktion setzt eine Lawine von weiteren Rollbacks in Gang.
  Eine Historie vermeidet kaskadisches Rücksetzen, wenn $c_w <_H r_r(A)$ gilt, wann immer T_r ein Datum A von T_w

  oder: Änderungen werden erst nach dem „commit“ freigegeben.
- strikt (*ST*)
  Bei strikten Historien dürfen auch veränderte Daten einer noch laufenden Transaktion nicht überschrieben werden.
  Wenn also für ein Datum A die Ordnung $w_w(A) <_H o_r(A)$ mit $o_i = r_i$ oder $o_i = w_i$ gilt, muss T_w zwischenzeitlich mit „commit“ oder „abort“ abgeschlossen worden sein.
- rigoros (*RG*)
  Eine Historie ist rigoros, wenn sie strikt ist und kein Objekt x wird überschrieben, bevor alle Transaktionen, welche x zuletzt gelesen haben beendet sind.

$RG \subseteq ST \subset ACA \subset RC$; $RG \subset CSR$; $RG \subset CO$

sperrende Scheduler

read lock (rl(x)) [read unlock (ru(x)]
write lock (wl(x)) [write unlock (wu(x)]

à ….rl(x)…..r(x)…..ru(x)

2-Phase-Locking (2PL) (Beispiele siehe Kap3, Folie 57; Kemper, S.310ff))
Ein Sperrprotokoll ist „2-Phase“ (Wachstumsphase (Sperren anfordern) und Schrumpfungsphase (Sperren freigeben)), wenn nach dem ersten ou_i keine weitere ql_i (mit $o,q \in \{r,w\}$.

Eigenschaften:
- einfache Implementierung
- bessere Performance als andere Protokolle
- Nicht Deadlock-frei!!! (Beispiel: $r_1(x)\ w_2(y)\ w_2(x)\ c_2\ w_1(y)\ c_1$)
  - Erkennung von Deadlocks durch:
    - Transaction timer (Zeitmarken)
    - Waiting Graphs (Zykel = Deadlock)
    - Timeout

strenges 2PL (S2PL): Alle Sperren (exclusive write) werden bis zum Ende gehalten!
(keine Schrumpfungsphase à CSR und ST, damit auch ACA)

konservatives 2PL: Alle Sperren sind seit dem Start gesetzt! (Preclaiming)
à Deadlockfrei, aber nicht praxistauglich
(woher soll der Scheduler am Anfang wissen, was er sperren soll)

Nachteile des 2PL Protokols:
- wenn 2PL Objekte „groß“ sind, so dass nur wenige Sperren zu managen sind, kommen Konflikte häufiger vor
- wenn 2PL Objekte „klein“ sind, so dass mehr Parallelität möglich ist, steigen die Kosten des Managements

Alternativen:
Tree locking (*TL*)
Multiple granularity locking (*MGL*)

nicht-sperrende Scheduler

- Zeitstempel
  Jeder Transaktion wird hierbei ein eindeutiger Zeitstempel zugeordnet
  wound-wait Wenn T_1 älter als T_2 ist, wird T_2 abgebrochen oder zurückgesetzt, so dass T_1 weiterlaufen kann. Sonst wartet T_1 auf die Freigabe der Sperre durch T_2.
  wait-die Wenn T_1 älter als T_2 ist, wartet T_1 auf die Freigabe der Sperre. Sonst wird T_1 abgebrochen oder zurückgesetzt.
- Optimistische Methode
  bisher: Konflikte kommen häufig vor à vermeide sie
  hier: Konflikte sind selten à teste nachher, ob welche aufgetreten sind

- Serialisierungsgraphtester (serialization graph tester)
  basiert auf der Charakteristik der Konflikt-Serialisierbarkeit
- Generic Validation
  Read-Phase
  Validation-Phase
    - backwards validation
    - forwards validation
  Write-Phase

Recovery

Recovery für Transaktionsfehler
System-Absturz (danach UNDO-REDO)
Verlust der Datenbank (Headcrash)

LOGs (Recovery Protocols) (Kap3, Folien 76ff)
Log-Einträge werden streng sequentiell angelegt. Daher lautet die Empfehlung, das Log auf einer eigenen Platte zu führen.

- à sicherer
- à vermeidet unnötige Platten-Suchzeiten
- à Erhöhung des Durchsatzes

UNDO-REDO

REDO notwendig, wenn Transaktionen bereits vor dem Fehler beendet wurden und die Ergebnisse nicht im stabilen Speicher sind.
à Recovery Manager braucht REDO, wenn es Transaktionen erlaubt ist zu terminieren, bevor alle geschriebenen Werte in die DB transferiert sind.

UNDO notwendig, wenn Transaktionen noch aktiv waren und einige Ergebnisse schon im Speicher sind.
à Recovery Manager braucht UNDO, wenn es noch laufenden Transaktionen erlaubt ist in die DB zu schreiben.

Undo-rule (Write-Ahead-Log-Protocol)
Das Before-Image einer Schreib-Operation (der alte Wert von x) muss ins Log geschrieben werden, bevor der neue Wert in der stabilen DB erscheint.

Redo-rule (Commit-rule)
Bevor eine Transaktion terminiert, muss jeder neue Wert, der von dieser geschrieben wurde, in den stabilen Speicher (DB oder Log).

Informationen:	UNDO	Before-Image (alter Wert von x vor der Änderung) geänderter Datenobjekte
	REDO	After Image geänderter Datenobjekte
	Beide: Welche Transaktion war für die Änderung verantwortlich?	
überflüssig?	UNDO	wenn geänderte Seiten erst zum Commit-Zeitpunkt in die stabile Datenbank geschrieben wurde
	REDO	sofern geänderte Seiten spätestens mit dem Commit in die stabile Datenbank geschrieben wurde

Indexstrukturen von Datenbanken

Braucht man Indexstrukturen eigentlich?
Theoretisch nein, sequentielle Suche ginge auch! (Komplexität: linear, O(n))
Praktisch ja, da man eine möglichst gute Leistung erzielen möchte!

Aufgaben:

- Lokalisierung der physischen Datensätze, die durch den Suchschlüssel (Wertekombination) spezifiziert werden
  à Ermittlung der physischen Seite(n), in denen diese Datensätze gespeichert sind
- Organisation der Seiten unter dynamischen Bedingungen
  (z.B. Überlauf der Seiten à Aufteilen der Seite auf zwei Seiten)

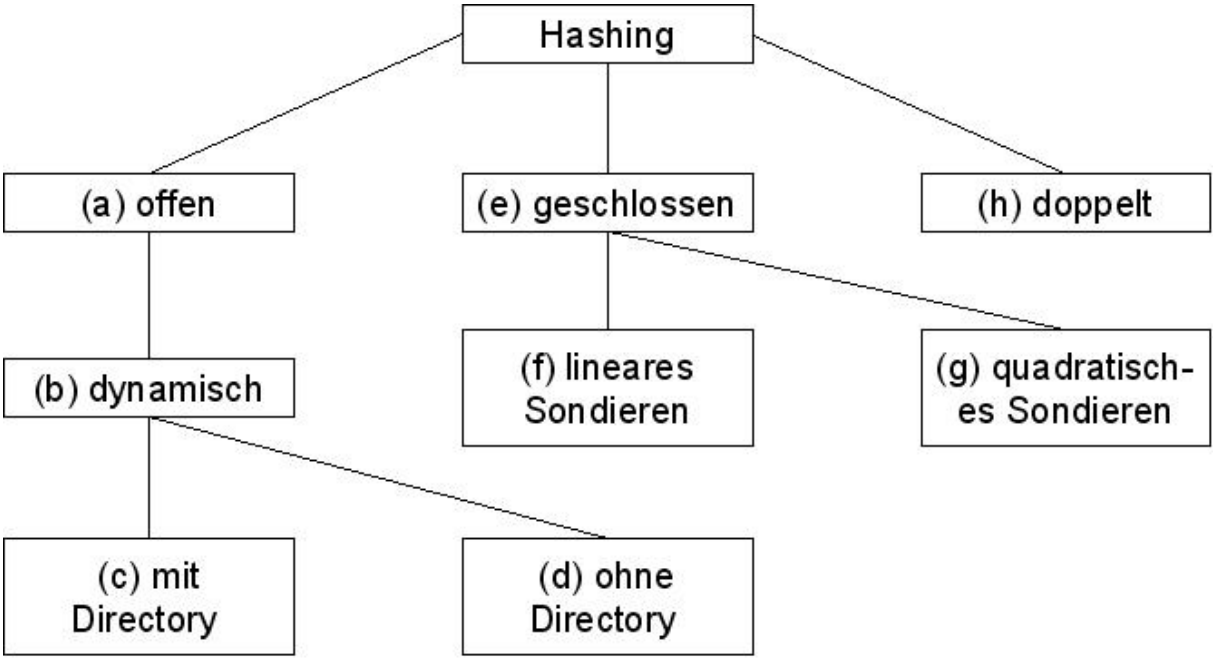
Anforderungen: **(EDOSIAN)**

- **E**ffizientes Suchen
- **D**ynamisches Einfügen, Löschen und Verändern von Datensätzen
- **O**rdnungserhaltung
- Gute **S**peicherplatzausnutzung
- **I**mplementierbarkeit
- **A**npassung an verändernde Datenverteilungen
- **N**ebenläufigkeit

Indexierung eindimensionaler Daten

Hashing

Ziel: direkte Berechnung eines möglichen Speicherortes (möglichst in O(1))
Grundprinzip: verschiedene Elemente mittels Hashfunktion auf denselben Indexwert abbilden



a) Speicherung der Schlüssel außerhalb der Tabelle, z.B. als verkettete Liste. Bei Kollisionen werden Elemente unter der selben Adresse abgelegt (im Überlaufbereich). Problem: Wie kann die externe Speicherung effektiv gelöst werden?

b) Die Hashtabelle wird bei Bedarf vergrößert, damit Datensätze weiterhin effizient eingefügt und gelöscht werden können.

c) Das Directory speichert Verweise auf Datenseiten, in denen die Schlüssel abgelegt werden. Ist eine Datenseite voll, so wird diese gespalten. Auch das Directory muss für hinreichend große Dateien auf dem Sekundärspeicher abgelegt werden. Jeder Datensatz wird mit zwei Zugriffen gefunden. (Speicherplatzausnutzung: ca. 70%)
Problem: Oft wächst das Directory superlinear (à Hashfunktion austauschen)

d) Hashfunktion liefert direkt die Seitenadressen [h: domain(K) à {Seitennummern}]
Überlaufsbehandlung:
- kein sofortiges Aufspalten (im Ggs. zu normalen erweiterbaren Hashing)
- besondere Kollisionsbehandlung der eingefügten Datensätze gemäß einer Überlaufstrategie (Suche dauert min. 2 Zugriffe)
- Verbreitet ist die Überlaufstrategie der „getrennten Verkettung“
  - Nutzung zweier verschiedener Seitentypen: Primär- und Überlaufseiten
  - Jede Überlaufseite ist genau einer Hashadresse h(K) zugeordnet
  - Neue Überlaufseiten werden mit existierenden (Überlauf- oder Primär) Seiten verkettet

Arten des Hashing ohne Dictionary:
- lineares Hashing
  benutze eine Folge von Hashfunktionen
- partielle Erweiterung
  Verdoppelung der Primärseiten

e) Bei Kollision wird mittels bestimmter Sondierungsverfahren [f) und g)] eine freie Adresse gesucht. Jede Adresse der Hashtabelle nimmt höchstens einen Schlüssel auf.
Problem: Finden geschickter bzw. effizienter Sondierungsverfahren, so dass nur wenige Sondierungsschritte notwendig sind.

Bei Kollision wird durch eine Folge h(x,j) mit j=1,2,… solange sondiert, bis eine freie Adresse gefunden wurde.

f) Prinzip: Verändern des Hashwertes um eine Konstante c [$h(x,j)=(h(x)+c*j) \bmod m$]
Problem: Cluster-Bildung (auch primäre Cluster) , d.h., dass Schlüssel mit ähnlichen Hashwerten liegen unter aufeinander folgenden Adressen, deshalb sind viele Sondierungsschritte nötig, um einen freien Platz zu finden.

g) $h(x,j)=(h(x)+ j^2) \bmod m$
Problem: Immer noch Bildung sekundärer Cluster

h) Doppelhashing soll Clusterbildung verhindern [siehe f) und g)], dafür werden zwei unabhängige Hashfunktionen verwendet. Nahezu ideales Verfahren aufgrund der unabhängigen Hashfunktionen.
Dabei heißen zwei Hashfunktionen unabhängig, wenn gilt:
Kollisionswahrscheinlichkeit P(h(x)=h(y))=1/m mit m der Größe der Hashtabelle

B-Baum (siehe auch IDB, S.13)
hierarchische Indexstruktur
Idee: Im Gegensatz zum Hashing eine dynamische Datenstruktur, die Bereichsanfragen ermöglicht (lokal Ordnungserhaltend) und effizient wachsen und schrumpfen kann.

Untersuchungen zeigen, dass die Knoten in B*-Bäumen zu 2/3 belegt sind
Wenn auf eine Unterlaufbehandlung verzichtet wird, wie beim R-Baum, sinkt der Füllgrad von ca. 50%-100% auf 40%-100%.

Degeneration?
Nein, da er, wenn überhaupt nach oben wächst à immer balanciert

Composite Index (Beispiel Telefonbuch) (siehe Folie 2-60)
Punktabfragen an die erste und Bereichsanfragen an die zweite Komponente sind effizient beantwortbar.
Gilt nicht für mehrdimensionale Bereichsanfragen. Punkt- oder Bereichsanfragen an die zweite Komponente sind nicht effizient beantwortbar.

Bitmap Indexing

- normalerweise für Attribute mit kleinen Wertebereichen (z.B. „Mann“/ „Frau“)
- ein Bit pro Attributwert
- Anordnung der Datensätze entsprechend ihrer physischen Speicherung.

Die Anzahl der entstehenden Bitvektoren entspricht der Dimensionen mal der Anzahl unterschiedlicher Werte, die für das jeweilige Attribut existieren.
à #Bitvektoren= #Attribute * #verschiedenerWerteJeAttribut

Vorteile
- effizienter sequentieller Scan (O(n); siehe Folie 2-35)
- auch logische Verknüpfungen durch sequentiellen Scan effizient möglich (siehe Folie 2-35)
- geringer Speicherbedarf
- besonders vorteilhaft bei kleinen Wertebereichen

Nachteile
- Reihenfolge der Bits und Reihenfolge der Daten müssen übereinstimmen
  à dadurch können hohe Aktualisierungskosten entstehen (Problematik bei Schreibanfragen)

Anfragen
- Standard-Bitmap-Indexe
- Mehrkomponenten-Bitmap-Indexe

Indexierung mehrdimensionaler Daten

Invertierte Listen (Beispiel siehe Folie 3-4)
Für anfragerelevante Attribute werden Sekundärindexe (invertierte Listen) angelegt.
Damit steht für jedes relevante Attribut eine eindimensionale Indexstruktur zur Verfügung.

Multiattributtssuche
- für jedes Attribut gibt es einen Sekundärindex
- Suche in allen Indexen, unabhängig von den anderen
- Kombiniere Ergebnisse über Durchschnittsbildung

Vorteile

- Invertierte Listen sind einigermaßen effizient, wenn die Antwortlisten sehr klein sind

Nachteile

- Die Attributswerte eines Datensatzes sind nicht in einer Struktur miteinander verbunden.
  - à Die Antwortzeit ist nicht proportional zur Anzahl der Antworten
  - à Die Suche dauert umso länger, je mehr Attribute spezifiziert sind
- Hohe Kosten für Update-Operationen
- Sekundärindexe beeinflussen die physische Speicherung der Datensätze nicht (es werden nur Verweise gespeichert)
  - à Ordnungserhaltung nicht möglich
  - à schlechtes Leistungsverhältnis

Raumfüllende Kurven (basierend auf Composite Indexes à zusammengesetzte Indexes)

Ziel: - beide Koordinaten spielen eine gleichberechtigte Rolle bei der Schlüsselbildung
- Nachbarschaftseigenschaft: im 2-dimensionalen benachbarte Punkte sollen im Index möglichst auch benachbart sein (auf benachbarten Plattenblöcken)

Idee: - Linearisierung des Raums durch raumfüllende Kurven
- jeder Punkt des Raums wird genau einmal durchlaufen
- jedem Punkt des Raums wird eine eindeutige Position auf der Kurve zugewiesen

Auswahl des Ansatzes abhängig von „access pattern“ à Welche Anfragen treten in der Datenbank auf?

Datenraum wird durch gleichmäßiges Raster partitioniert, wobei jede Zelle durch eine eindeutige Nummer, die ihre Position in der totalen Ordnung definiert, identifiziert wird
à daraus ergibt sich eine 1-dim. Einbettung

Ansätze z.B.:

Z-Ordnung [00 à 01 à 10 à 11]; diese Bitstrings dienen als Schlüssel in 1-dim. B+-Bäumen
- Berechnung des Z-Wertes durch Bit-Verzahnung

Gray-Codes [00 à 01 à 11 à 10]

Hilbert-Kurve
- Besonderheit: benachbarte Punkte auf der Kurve sind auch im 2-dimensionalen benachbart (Umkehrung gilt nur zum Teil)
- Anfragen: Raum teilen und Fallunterscheidungen um Felder auszuschließen

Quad-Tree (siehe Folien 3-15)

nicht-balancierte Datenstruktur für mehrdimensionale Punkt-Objekte
Ansatz: Jeder Knoten hat sowohl für die x- als auch für die y-Koordinate je zwei Sohnknoten

Aufbau:

- erster Punkt bildet Wurzel
- bei jedem weiteren Punkt
  - Bestimmung des Quadranten, in dem der Punkt liegt
  - Abstieg in den entsprechenden Sohnknoten
  - falls kein Sohnknoten für diesen existiert: Hinzufügen eines Sohnknotens

Problem: Entartet zu einer Liste, wenn alle Punkte auf einer Diagonalen liegen

Variationen:

- MX-Quadtree (siehe Folie 3-17)
  - diskreter Datenraum ($2^k * 2^k$ -Gitter)
  - Datenpunkte sind Elemente in einer quadratischen Matrix (à MX-Quadtree)
  - Splits jeweils in der Mitte
  - Daten nur in Blattknoten gespeichert
- PR-Quadtree (siehe Folie 3-18)
  - Erweiterung des MX-Quadtrees auf nicht-diskrete Daten
  - rekursive Dekomposition des Raums wie beim MX-QT solange, bis jeder Block maximal einen Punkt enthält
- Bucket PR-Quadtree (siehe Folie 3-19)
  - Erweiterung zu Sekundärspeicherstruktur
  à mehr als ein Datenpunkt pro Quadrant erlaubt
  - bei Überlauf: Bildung neuer Quadranten

Grid-File (siehe Folien 3-20ff)
Idee: Hashverfahren mit Directory für k-dimensionale Schlüssel; ordnungserhaltend
Struktur:

- Datenraum wird in ein k-dimensionales orthogonales Gitter unterteilt (Grid)
- Skalen, die als Suchbäume verwaltet werden, definieren die Einteilung des Gitters für jeweils eine Dimension
- Grid-Direcory (GD) ist ein k-dimensionales Array
- GD-Element ist eine Zelle des Directories mit dem Verweis auf die Datenseite, welche die entsprechenden Datensätze enthält
- Ebenfalls analog zum Hashing mit Directory können mehrere GD-Elemente auf eine Seite verweisen. Eine solche Menge von GD-Elementen heißt Seitenregion

Als Verfahren mit Directory vermeidet das Grid-File Überlaufseiten.

Wahl der Partitionierungslinie:

- Wahl der Splitachse
  - Ziel: möglichst quadratische Seitenregionen
  à Längste Achse der Seitenregion als Splitachse auswählen
  - Ziel: Anfrageverteilung berücksichtigen
  à Häufig angefragte Achsen feiner unterteilen
- Wahl der Splitposition
  - Ziel: den Datenraum möglichst gleichmäßig aufteilen
  à Mittelsplit: Halbieren der Datenseite
  - Ziel: die Daten möglichst gleichmäßig verteilen
  à Mediansplit: Auf allen Seitenregionen sollen möglichst gleich viele Datensätze stehen (Quantilverfahren)

MDB-Bäume (siehe Folien 3-26ff)
Multidimensionale B-Bäume
Ziel: Speicherung multidimensionaler Schlüssel in einer Indexstruktur
Idee: Hierarchie von Bäumen, wobei jeweils jede Hierarchiestufe einem Attribut entspricht

EQSON-Zeiger: Zeiger zwischen den B-Bäumen der Hierarchiestufen

Indexierung räumlicher Daten

Punkttransformation (siehe Folien 3-32)

n-dimensionale Rechtecke werden in 2*n-dimensionale Rechtecke überführt
Alternativen der Überführung:

- Mittentransformation:
  Das Rechteck wird durch den Mittelpunkt und die jeweils halbe Ausdehnung in x- und y-Richtung beschrieben
- Eckentransformation:
  Das Rechteck wird durch die diagonal gegenüberliegenden Eckpunkte beschrieben.

Anfragetypen

- Punktanfrage: In welchen Rechtecken liegt der Punkt?
  - Darstellung für x- und y-Achse (Liegt Koordinate im jeweiligen Intervall?)
  - beide 2-dim. Koordinatensysteme spannen ein 4-dim. Koordinatensystem auf
  - der Schnitt beider 4-dim. Lösungshyperkuben enthält Lösungsrechtecke
- Rechtecksschnitt: Welche Rechtecke werden geschnitten?
  - Analyse der Eckpunkte, ob q r schneidet ($r.x_{li} \le q.x_{re}$, $q.x_{li} \le r.x_{re}$, $r.y_{un} \le q.y_{ob}$ und $q.y_{un} \le r.y_{ob}$)
- Rechtecksenthaltensein:In welchen Rechtecken liegt das Rechteck?
  - Analyse der Eckpunkte, ob q in r enthalten ist ($r.x_{li} \le q.x_{li}$, $q.x_{re} \le r.x_{re}$, $r.y_{un} \le q.y_{un}$ und $q.y_{ob} \le r.y_{ob}$)

Clipping (Objektduplikation; siehe Folien 3-38f)

Idee: Rechteck in jede Region einfügen, die es schneidet

R-Bäume (Rectangle-Tree; Folien 3-40ff)

kein Clipping, um Mehrfachspeicherung zu vermeiden

höhenbalancierter Baum, der zur Speicherung von Rechteckdaten entworfen wurde

Idee:

- versuche mehr-dim. Strukturen so zu transformieren, dass sie ein-dim., also z.B. mit Bäumen abgebildet werden können
- basiert auf der Technik überlappender Seitenregionen
- Approximation der Objekte durch achsenparallele, minimalumgebende Rechtecke (**MUR**)
- verallgemeinert die Idee des B+-Baumes auf den 2-dim. Raum

Problem: Überlappung der Directoryregionen

à Fällt eine Anfrage in einen Überlappungsraum, müssen mehrere Pfade untersucht werden (im Zugriff geht log verloren)

à Die Überlappungen müssen möglichst klein gehalten werden

Grundsätzliches:

- R-Baum erlaubt die Indexierung von k-dim. achsenparallelen Rechtecke
- sowohl Datenrechtecke, als auch Datenseiten können sich überlappen
- innere Knoten: Directory Seiten; bestehen aus MUR und Verweisen auf andere Seiten

- Blätter: Datenseiten; bestehen aus MUR und Verweisen auf exakte Objekt-Repräsentanten
- R-Baum besitzt pro Knoten m Indexeinträge

Laufzeit: O(n) [Lesen aller Blätter O(n) + innerhalb des Baumes ca. genauso viele Elemente wie Blätter O(n) = 2*n = O(n)]

Komplexität des Suchens: logarithmisch (WC: O(n))

Parameter:
- *M*: maximale Anzahl von Einträgen pro Knoten (abhängig von Blockgröße)
- $m \leq M / 2$:Mindestbelegung pro Knoten

Eigenschaften:
- Anzahl der Index-Einträge pro Blatt-Knoten zwischen m und M
- Anzahl der Sohn-Knoten von Nichtblatt-Knoten zwischen m und M
- in inneren Knoten ist das kleinste Rechteck gespeichert, welches Rechtecke der Sohnknoten umfasst
- in Blatt-Knoten ist Verweis auf Objekt und sein kleinstes umschließendes Rechteck gespeichert
- höhenbalanciert
- Zerlegung des Datenraumes nicht disjunkt
- Höhe des Baums $h \leq (\log_m N) - 1$ (bei *N* gespeicherten Objekten]

Suchen:
- ähnlich zum B-Baum, nur das hier pro Knoten mehrere „Schlüssel“ (hier: Suchräume) zutreffen können

Einfügen:
- ähnlich wie im B*-Baum (in den Blattknoten, bei Überlauf Split des Knotens,…)
- im Ggs. zum B*-Baum kommen hier i.a. mehrere Blattknoten (überlappende Suchbereiche) in Frage
- Auswahl des geeigneten Blattknotens/ Teilbaumes:
  Jener, der durch das Einfügen nicht oder am geringsten vergrößert wird

Knotensplit:
Problem: große Anzahl von Möglichkeiten
Wunsch: möglichst wenig Überlappung der Rechtecke beider Knoten

Bestimmung der genauen Lösung zu aufwendig [$O(2^M)$]
à Heuristiken (gegeben: M+1 Rechtecke, die auf zwei Knoten verteilt werden müssen)
- Quadratic Split [$O(M^2)$]
  Suche zwei Rechtecke, die die größte Fläche „verschwenden“ würden, wenn sie zu einem Knoten hinzugefügt würden
  à verteile diese Rechtecke auf beide Knoten
  à weise restliche Rechtecke zu
- Linear Split [$O(M)$]
  Finde für jede Dimension die Rechtecke, welche die höchste aller Unteren und die niedrigste Koordinate bezüglich dieser Koordinate annehmen

à Normalisierung der gefundenen Rechtecke durch Division mit der Strecke zwischen min- und max-Wert in jeder Dimension
à Wähle das Paar, mit der größten normalisierten Separierung in einer Dimension

Löschen:

- Start bei Blattknoten, in dem Eintrag gelöscht wurde
- knotenweiser Aufstieg zur Wurzel, dabei
  - falls Minimalbelegung (<m) nicht mehr gegeben: Knoten und Eintrag in Vaterknoten löschen, Vormerken (in spezielle Menge Q) der übrigen Knoten (zum späteren Wiedereinfügen)
  à wurde die Wurzel erreicht, dann werden alle ausgelöschten Knoten, d.h. aus Menge Q wieder in den Baum eingefügt
  - Anpassen der MURs

Anfragetypen: (Folien 3-53ff)

- Punktanfrage
- Rechteckanfrage
- Bereichsanfrage
- nächster-Nachbar
- k-nächste-Nachbarn

Nächste Nachbar Verfahren: (Folien 3-59ff)

- Naiv
  - Nachteile: Start mit einem beliebigen Pfad, nicht mit dem Pfad, der möglichst nahe am Anfragepunkt liegt à das Verfahren schränkt seinen Suchraum erst sehr langsam ein
- nach Rossopolous (k-NN mit lokaler Sortierung)
  - bessere Einschränkung des Suchraumes durch
    - Verwendung von Seitenregionen zur Abschätzung der NN-Distanz
    - Priorisierung der Tiefensuche nach Abstand der Seitenregionen von q (d.h. vor dem rekursiven Abstieg: Sortieren der Sohnseiten nach Abständen)
- nach Samet (k-NN mit globaler Sortierung)
  - statt eines rekursiven Durchlaufes durch den Index hält der Algorithmus explizit eine Liste der aktiven Seiten
    - eine Seite p ist aktiv, wenn
      - sie noch nicht geladen wurde
      - die Elternseite bereits geladen wurde
      - die Distanz zwischen der Region p.region und dem Anfragepunkt nicht die Pruningdistanz übersteigt
  - Verfahren optimal in Bezug auf die Seitenzugriffe
  - Nachteil? zusätzlicher Speicher
    Wieviel? WC: Wahrscheinlich ein Verweis für jeden Eintrag (MUR) im Baum!

Kann ein R-Baum entarten?
Jein, der Baum selbst ist immer balanciert, aber bei starker Überlappung der Rechtecke durchsucht man trotzdem alle Zeiger auf Datenobjekte in den Blättern! Passiert bei hochdimensionalen Daten fast immer!

Lassen sich Überschneidungen verhindern?
Nein, aber minimieren

Was sind die Probleme bei hochdimensionalen Daten?
Überlappungen nehmen zu!

Variationen:

- R*-Baum
  - Änderung der Splitheuristik
  - Reorganisation des Baumes durch Löschen und Wiedereinfügen von Rechtecken
- Hilbert R-Baum
  Verbesserung des Splits durch Ordnung der Rechtecke gemäß des Verlaufs der Hilbert-Kurve
- SS-Baum (Similarity Search Tree)
  Anstelle von Rechtecken Verwendung von Kreisen in den Directory-Knoten
- SR-Baum (Sphere Rectangle Tree)
  Kombination von R- und SS-Baum

Approximation

Idee: möglichst viele Objekte, die keine Antworten sind, ohne Zugriff auf ihre exakte Repräsentation zu schliessen
Ziel: möglichst klein mit wenig Verschnitt!

Arten von Approximationen:

- achsenparalleles, minimal umgebendes Rechteck (BB) à MBR à MUR
- gedrehtes, minimal umgebendes Rechteck (RBB)
- minimal umgebender Kreis (CIR)
- minimal umgebene Ellipse (E)
- konvexe Hülle (CH)
- minimal umgebene konvexe Vier- oder Fünfecke (4-C bzw. 5-C)

Interpretation:

- CH hat im Unterschied zu 5-C beliebig viele Parameter, trotzdem approximiert 5-C fast gleich gut
- RBB: 1 Parameter mehr als BB à 31% Verbesserung
- 5-C: 6 Parameter mehr als BB à 60 % Verbesserung

Vorteile genauer Approximation:

- Zugriff auf die exakte Repräsentation für weniger Objekte (weniger Seitenzugriffe)
- Verfeinerungsschritt für weniger Objekte (weniger CPU-Zeit)
- Verbesserung der Verarbeitungszeit ist proportional zur Verbesserung der Approximationsgüte

Nachteile genauer Approximation:

- höherer Speicherplatzbedarf auf Datenseiten
- höherer Aufwand für die Berechnung der Approximationen und Tests auf den Approximationen

Bewertung:

- der Kreis ist für die Approximation nicht geeignet

- die Ellipse besitzt den niedrigsten break-even-Punkt
- Das 5-Eck liefert den besten Trade-Off (Speicher, Leistung, Break-Even)
- bei komplexeren Objekten (wachsendes c) lohnen sich komplexere Approximationen

Dekomposition

Idee: Verfeinerungsschritt (Test auf exakte Repräsentation) vereinfachen
à Dekomposition der EPL (einfache Polygone mit Löchern) in einfache Teilobjekte und Approximation der Teilobjekte durch minimal umgebende Rechtecke (MUR)

Vorteile:
- Teilobjekte lassen sich durch MUR besser approximieren als das Gesamtobjekt (Filterschritt arbeitet genauer)
- es entstehen einfachere Teilobjekte (Verfeinerungsschritt effizienter)

Nachteile:
- Rechen-Aufwand für die Dekomposition [O(n*log(n)]
- größerer Speicherbedarf
- komplexere Handhabung von Objekten

Indexierung hochdimensionaler Daten

neue Anwendungen behandeln hochdimensionale Daten
- Multimedia
- Biologie

à Transformation der Daten in Feature-Vektoren (meistens Punktdaten)

Lösungsansätze
- neue Heuristiken für den Split der Daten/ des Datenraums (X-Baum)
- Beschleunigung des sequentiellen Scans (VA-File)
- Transformation der Daten

X-Baum

Idee: - Weiterentwicklung des R*-Baums für hochdimensionale Daten
- der X-Baum vermeidet Überlappung mit Directory mittels:
  - einem Split mit minimaler Überlappung
    (bei Punktdaten existiert sogar ein überlappungsfreier Split)
  - dem Konzept der Superknoten (keine fixe Seitengröße)
    à Verbesserung des linearen Scans

Ziel: niedrige- und hochdimensionale Daten indexieren

Eigenschaften:
- Hybrid aus hierarchischer Indexstruktur (Baumartig) und linearem Scan
  - eine hierarchische Struktur ist günstig für niedrigdimensionale Daten ($\dim \leq 5$) [wenig Superknoten]
  - eine lineare Organisation ist günstig für hochdimensionale Daten ($\dim \leq 5$) [öfters Superknoten]
- durch dynamische Anpassung eine bestmögliche Kombination für jede Dimensionalität

Split:
Im Unterschied zum R*-Baum, bei dem hochdimensionale Räume zu einer hohen Überlappung der Directory-Seitenregionen führt, wird nun bei jedem Split zwei neue Seiten erzeugt und so lässt sich die Historie von Splitoperationen als binären Baum darstellen.

Vorteile:
- Überlappung wird gering gehalten, diesbezügliche Probleme bei den Operationen dadurch vermieden
- sequentieller Scan eines Superknotens ist schneller als mehrere einzelne Zugriffe
- Durchschnittliche Speicherplatzausnutzung ist höher (85% (m/(m+0,5) bei m=3) im Vergleich zu 66% bei R-Baum)
- durch die Superknoten (und dadurch eingesparte Verzweigungen im Index) ist das Directory kleiner à größere Teile können im Cache gepuffert werden

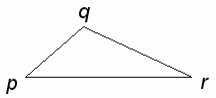
Nachteile:
- Gegen Überlappungen, die z.B. beim normalen Einfügen entstehen, wird nichts unternommen
- Konzept der Superknoten wird nur sehr begrenzt eingesetzt

Indexstrukturen für metrische Daten

Daten, deren Koordinaten nicht bekannt sind (nur noch Abstände/ Distanzen)
Grundeigenschaften:

Symmetrie
$$d(p,q) = d(q,p)$$

Definitheit	$d(p,q) = 0 \Leftrightarrow p = q$
Nichtnegativität	$d(p,q) \geq 0$
Dreiecksungleichung	$d(p,r) \leq d(p,q) + d(q,r)$



Ziel: Einsatz von Indexstrukturen ohne Dimensionsinformation (um I/O-Kosten zu reduzieren)
à Partitionierung der Daten, so dass nur einzelne Partitionen bi Anfragen durchsucht werden müssen

Abschätzen von Entfernungen:
Dreiecksungleichung (wichtig für „Abschneiden des Suchbaums“): Umweg über einen dritten Punkt kann niemals kürzer sein, als der direkte Weg zwischen zwei Punkten.

Pruning (bewusst bestimmte Informationen ignorieren, um eine höhere Effizienz zu erreichen) in metrischen Räumen (Folie 5-4)
$$|d(q,p) - d(p,o)| \leq d(q,o) \leq d(p,q) + d(p,o)$$
für Anfrageobjekt *q*, Referenz *p* und Objekt o

Varianten

- BK-Tree
  - diskrete Distanzfunktionen
- FQT
  - Unterschied zu BKT: pro Ebene wird ein festes Pivot-Element $(p \in U)$ gewählt

- Fixed Height QT
    - Unterschied zu FQT: alle Blätter befinden sich auf der selben Ebene
- MetricTree / VP-Tree
    - für kontinuierliche reelwertige Distanzfunktionen
    - für konstant große Samplemengen O(n*log(n))
- MVPT (Multi-Vantage-Point-Tree)
    Ziel: geringere Höhe, höherer Verzweigungsgrad
- VPF (Eluded middle vantage point forest)
    - Erweiterung des VPT zu einem Wald
    - Ziel: Optimierung der NN-Anfragen mit Maximalradius r*
- BST (Bisector Tree)
- GNAT (Geometric Near-Neighbor Access Tree)

M-Baum

Idee: Algorithmen des R-Baumes verwenden
- seitenorientiert, balanciert, für große dynamische Datenmengen geeignet
- Prototypimplementierung auf der Basis des GiST (Generalized Search Tree) Frameworks

Informationen können verwendet werden, um Distanzberechnungen zu sparen.

Struktur:

innere Knoten enthalten
- Routingobjekte
- Verweise auf die Wurzel des zugehörigen Teilbaumes
- Radius des abgedeckten Bereiches
- Distanz zum Vorgänger

Blätter beinhalten Objekte

Bereichsanfrage
- rekursiver Abstieg in die möglichen Teilbäume (ähnlich zum R-Baum)
- aber Vermeidung von Abstandsberechnungen durch Dreiecksungleichung

Einfügen analog zum R-Baum

Splitting:

a) Minimierung des Volumens (abgedeckter Bereich)
b) Minimierung des überlappenden Bereiches
    à reduziere damit die # zu verfolgenden Pfade bei Anfrage

Algorithmen für die Wahl der Routingobjekte (Promote)
zu a) „minimum Radii“
zu b) „minimum lower bound on DISTance“

Algorithmen für die Partitionierung der Menge (Partitionierung)
Generalized Hyperplane
Balanciert

Möglichkeiten nur wenige Dimensionen zu betrachten:
Dimensionsreduktion

Indexstrukturen für Intervalldaten

Intervalle sind ausgedehnte Objekte in 1D

Alternativen:

- Interval Tree
  gegeben: Menge von Rechtecken
  gesucht: alle Paare sich schneidender Rechtecke
  Idee: Transformation des 2D-Problems in eine Folge von 1D-Problemen
- Tile Tree
  - Repräsentation der Intervalle im 2-dim durch Punkttransformation
  - Quadrate erhalten Schlüssel: x- oder y-)Wert des rechten unteren Eckpunktes
- Segment Tree
  - 1D-Quadtree
  - Skelett ist ein balancierter Binärbaum
  - Speicherbedarf: O(N log(N))
  - Aufbau des Baumes: O(N log(N))
  - Anfrageoptimierung: $O((\log(N))^2 + t$, mit t= Anzahl der Lösungen
- Priority Search Tree
- Time Index
  - Sekundärspeicherbasiertes Verfahren

Interval B-Tree

- Sekundärspeicherbasiertes Verfahren
- Implementierung des Interval Trees mithilfe eines (erweiterten) B+-Baums
- Speicher: O(n) für n gespeicherte Intervalle (jeweils nur einmal gespeichert)
- Overlaps-Anfrage: O(log(n) + Ausgabe) (Baumdurchlauf)
- Einfügen und Löschen: O(log(n)) (Baumdurchlauf)

Relationalen Intervallbaum (Folien 6-28ff)

- kein „stand-alone“-Konzept
- Primärstruktur: Binärbaum mit nummerierten Knoten
- Virtualisierung: Primärstruktur wird nicht materialisiert (virtuell verwaltet, kein Baumaufbau)
- keine Reorganisation irgendeiner Struktur beim Einfügen/ Löschen notwendig
- Sekundärstruktur: Sortierte Listen (besser noch Bäume) von Intervallgrenzen an den Knoten
  à Speicher: O(n)
- Platzbedarf für n Intervalle: O(n/b) Plattenblöcke der Größe b
- Einfügen/ Entfernen: O($\log_b(n)$) Plattenzugriffe in den Indexen

Wurzel merken, die ein Wert 2^{n-1} ist und einen Bereiche von $2^n - 1$ abdeckt.
An den Knoten werden die Intervalle als 2 B*-Bäume der Anfangs- und Endpunkte gespeichert (à Speicherbedarf bei 2 Listen O(n))